

CS2100

<http://www.comp.nus.edu.sg/~cs2100/>

COMPUTER ORGANISATION

## Lecture #22

---

# Cache

## Part I: Direct Mapped Cache



**NUS**  
National University  
of Singapore

School of  
Computing



# Questions?

Ask at

<https://sets.netlify.app/module/676ca3a07d7f5ffc1741dc65>

**OR**

**Scan** and ask your questions here!  
(May be obscured in some slides)

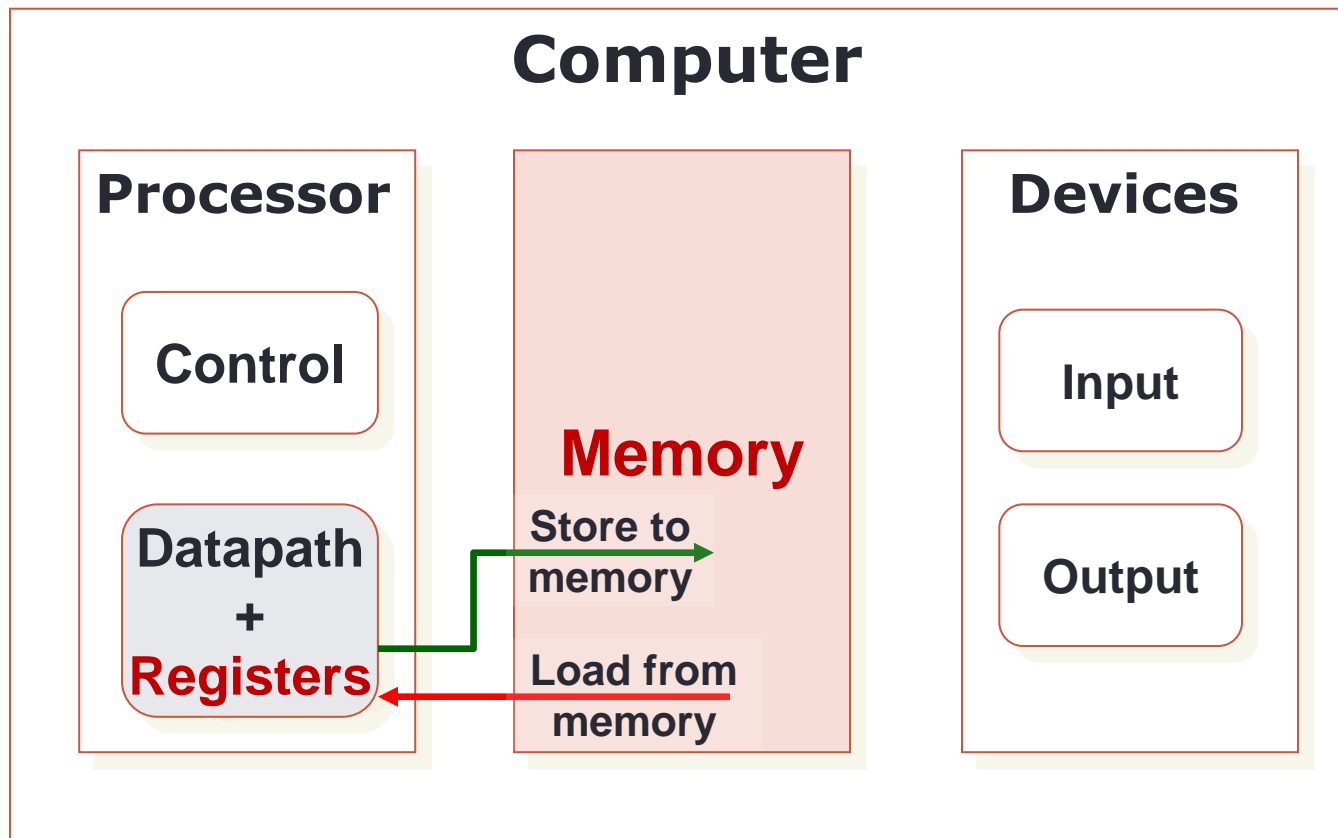


# Lecture #22: Cache I: Direct Mapped Cache

1. Introduction
2. Cache
  - 2.1 Locality
  - 2.2 Memory Access Time
3. Memory to Cache Mapping
4. Direct Mapping
5. Reading Data (Memory Load)
6. Types of Cache Misses
7. Writing Data (Memory Store)
8. Write Policy



# 1. Data Transfer: The Big Picture



Registers are in the datapath of the processor. If operands are in memory we have to **load** them to processor (registers), operate on them, and **store** them back to memory.



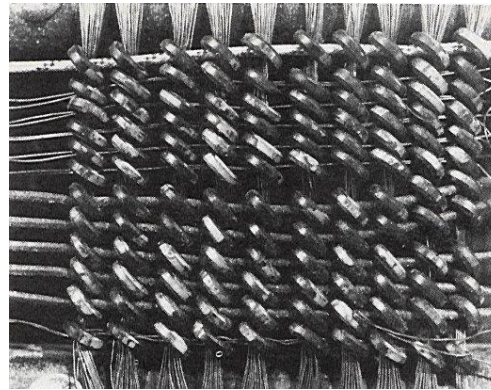
# 1. Memory Technology: 1950s



**1948: Maurice Wilkes examining EDSAC's delay line memory tubes  
16-tubes each storing 32 17-bit words**



**Maurice Wilkes: 2005**



**1952: IBM 2361 16KB magnetic core memory**

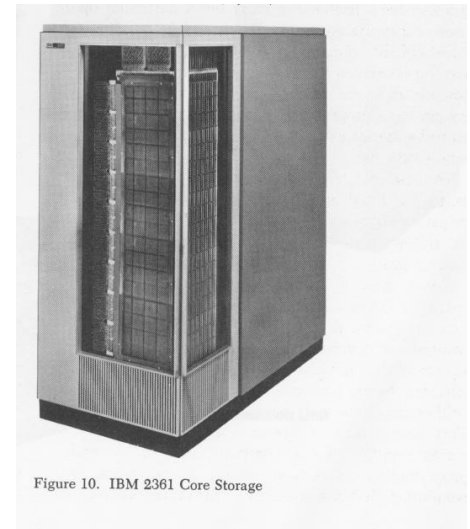


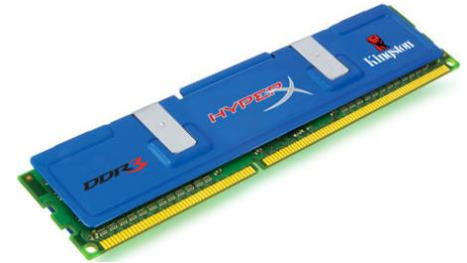
Figure 10. IBM 2361 Core Storage



# 1. Memory Technology Today: DRAM

## ■ **DDR SDRAM**

- **Double Data Rate**
  - **Synchronous Dynamic RAM**
- The dominant memory technology in PC market
- Delivers memory on the positive and negative edge of a clock (double rate)
- Generations:
  - DDR ( $\text{MemClkFreq} \times 2(\text{double rate}) \times 8 \text{ words}$ )
  - DDR2 ( $\text{MemClkFreq} \times 2(\text{multiplier}) \times 2 \times 8 \text{ words}$ )
  - DDR3 ( $\text{MemClkFreq} \times 4(\text{multiplier}) \times 2 \times 8 \text{ words}$ )
  - DDR4 (released in 2014)
  - DDR5 (in Q3 2021)

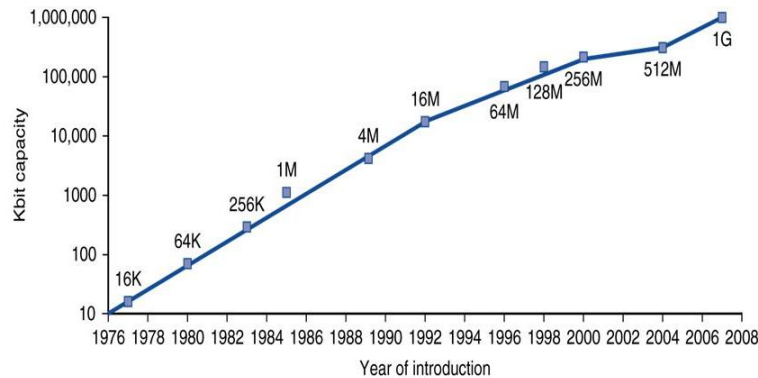


# 1. DRAM Capacity Growth

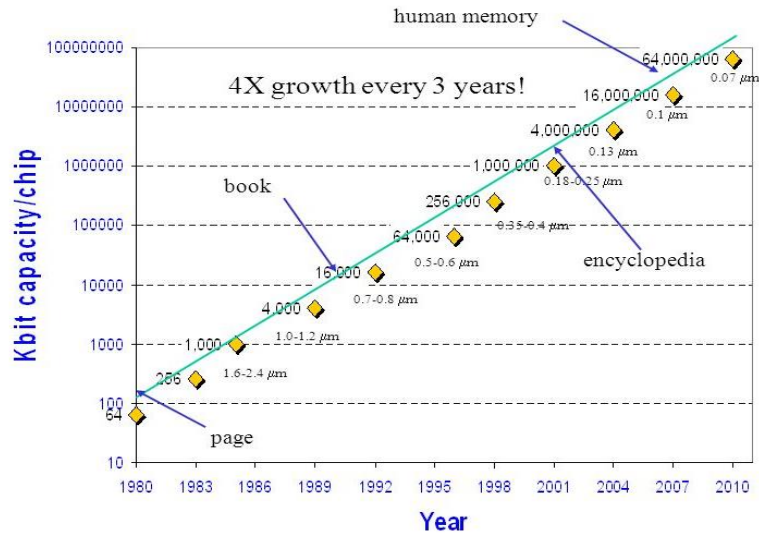
## Growth of Capacity per DRAM Chip

❖ DRAM capacity quadrupled almost every 3 years

✧ 60% increase per year, for 20 years



## DRAM Chip Capacity



- Unprecedented growth in density, but we still have a problem

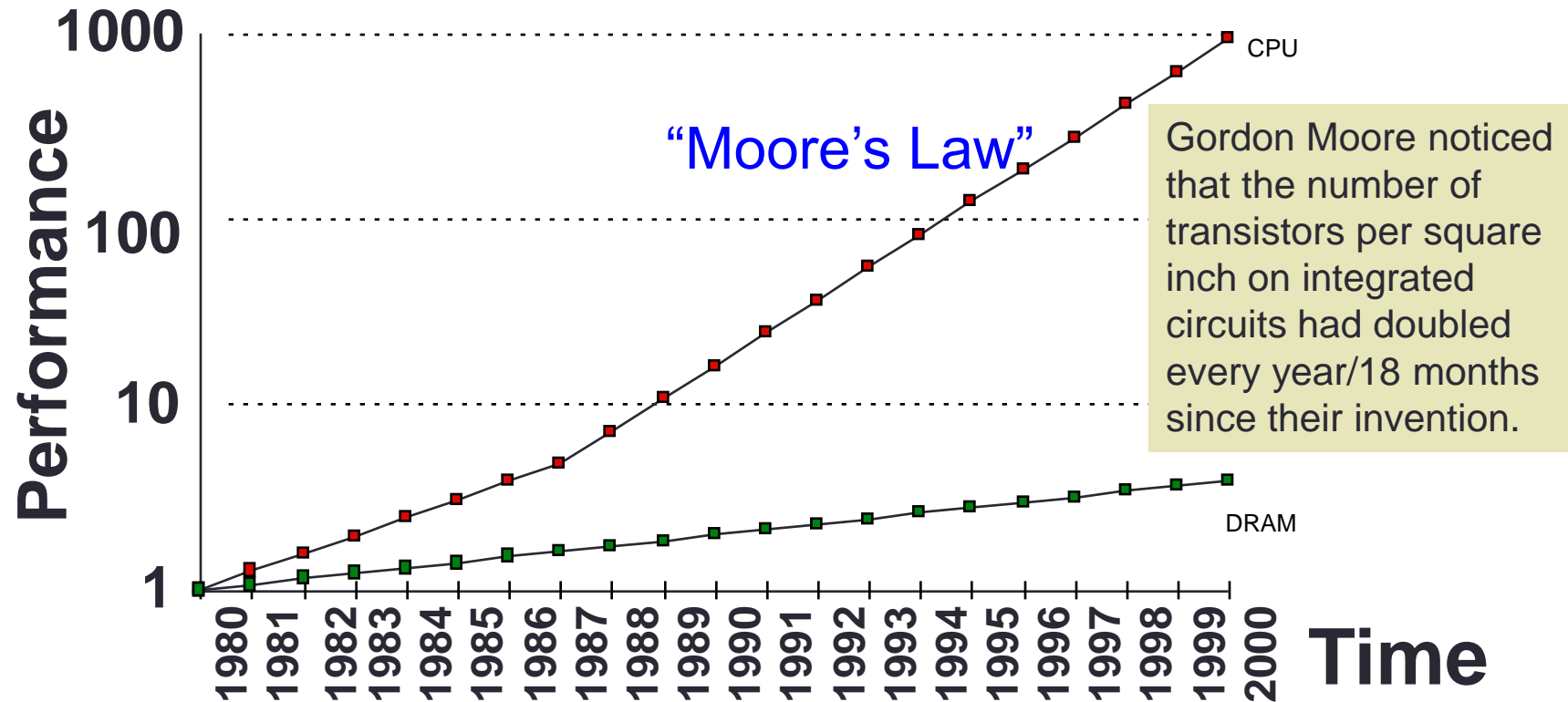


# 1. Processor-DRAM Performance Gap

## Memory Wall:

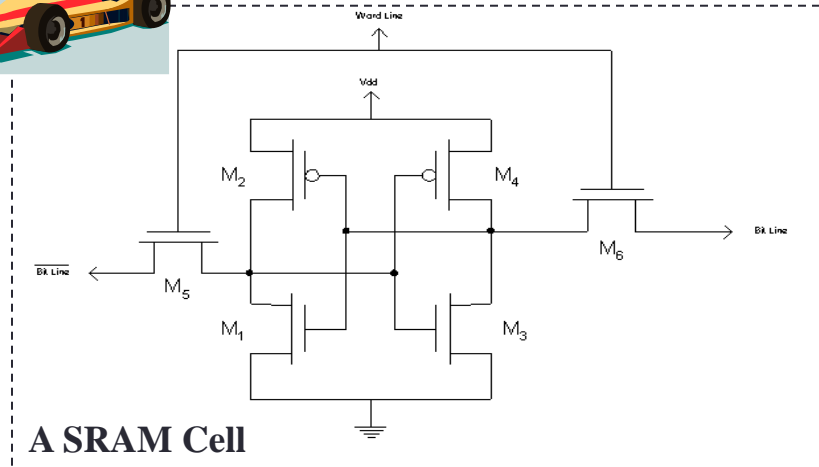
1GHz Processor  $\rightarrow$  1 ns per clock cycle

50ns for DRAM access  $\rightarrow$  50 processor clock cycles per memory access!





# 1. Faster Memory Technology: SRAM



## SRAM

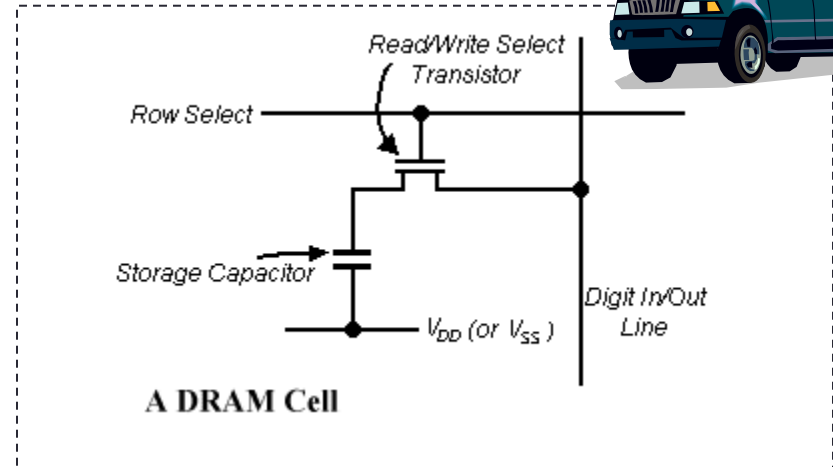
6 transistors per memory cell

→ **Low density**

**Fast access** latency of 0.5 – 5 ns

More costly

Uses flip-flops



## DRAM

1 transistor per memory cell

→ **High density**

**Slow access** latency of 50-70ns

Less costly

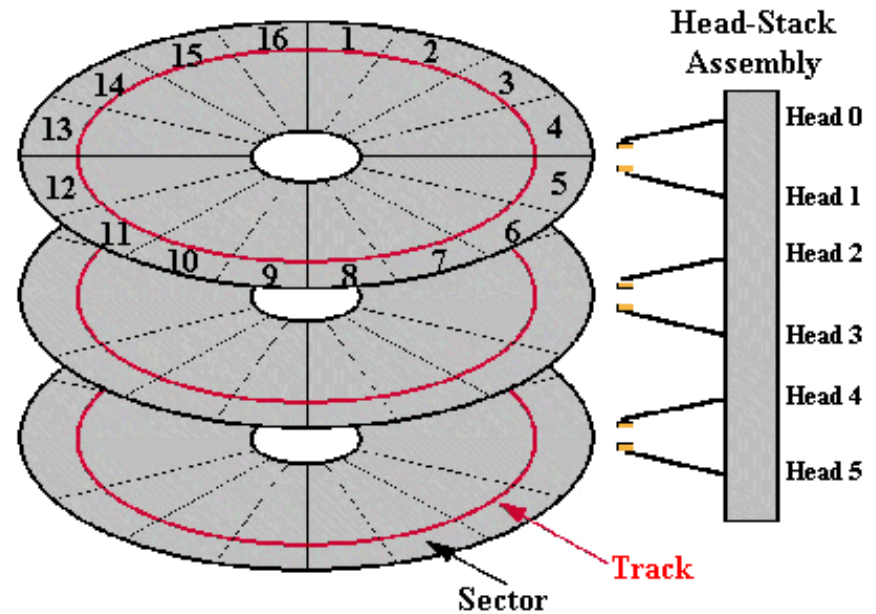
Used in main memory



# 1. Slow Memory Technology: **Magnetic Disk**



Drive Physical and Logical Organization



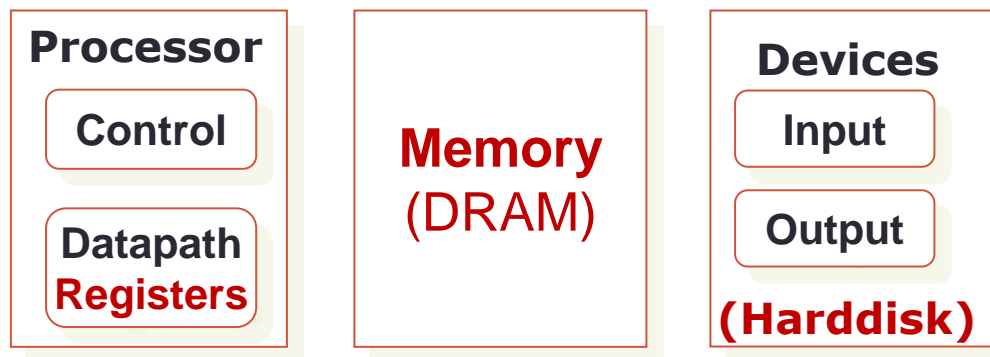
Typical high-end hard disk:

Average Latency: **4 - 10 ms**

Capacity: **500-2000GB**



# 1. Quality vs Quantity



	Capacity	Latency	Cost/GB
Register	100s Bytes	20 ps	\$\$\$\$
SRAM	100s KB	0.5-5 ns	\$\$\$
DRAM	100s MB	50-70 ns	\$
Hard Disk	100s GB	5-20 ms	Cents
<b>Ideal</b>	<b>1 GB</b>	<b>1 ns</b>	<b>Cheap</b>



# 1. Best of Both Worlds

- What we want:
  - A **BIG** and **FAST** memory
  - Memory system should perform like 1GB of SRAM (1ns access time) but cost like 1GB of slow memory

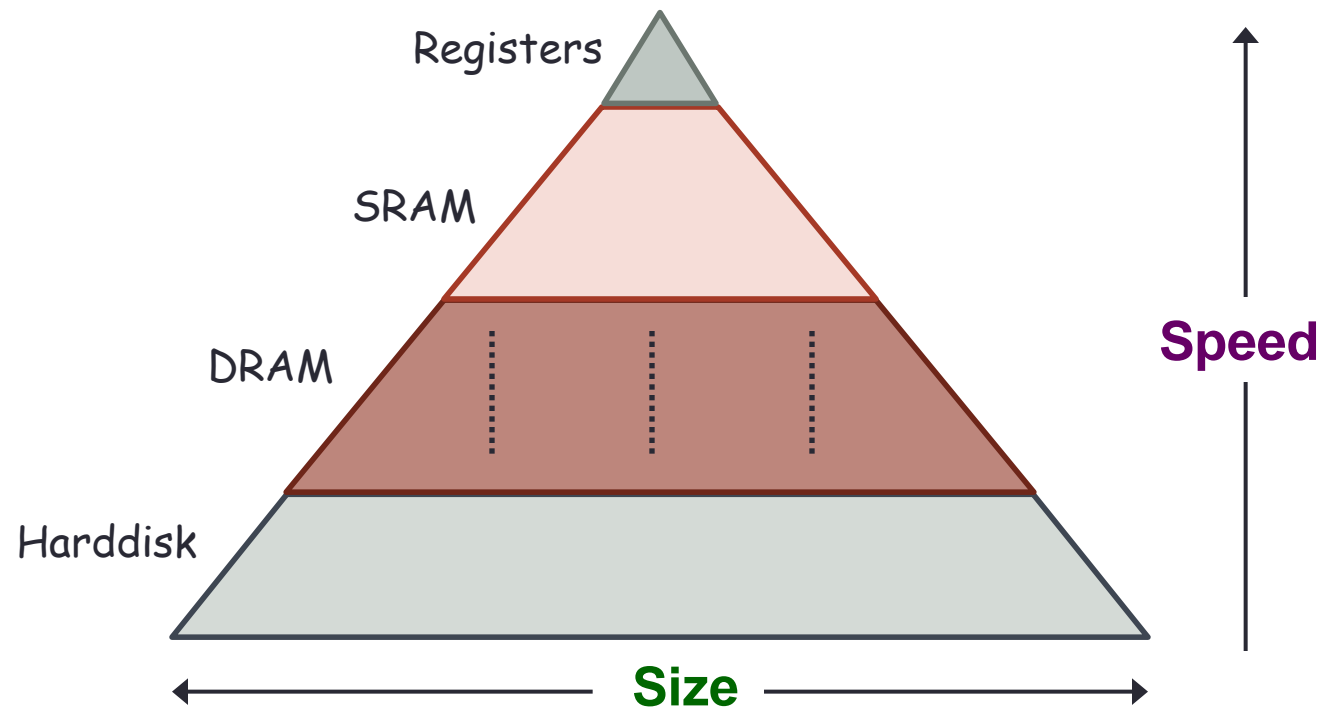
## Key concept:

Use a **hierarchy** of memory technologies:

- ❖ Small but fast memory near CPU
- ❖ Large but slow memory farther away from CPU



# 1. Memory Hierarchy



## 2. Cache: The Library Analogy



Imagine you are forced to put back a book to its bookshelf before taking another book.....



## 2. Solution: Book on the Desk!



What if you are allowed to take the books that are **likely to be needed soon** with you and place them nearby on the desk?



## 2. Cache: The Basic Idea

- Keep the frequently and recently used data in **smaller but faster** memory
- Refer to bigger and slower memory:
  - Only when you cannot find data/instruction in the faster memory
- Why does it work?

### Principle of Locality

Program accesses only a small portion of the memory address space within a small time interval





## 2.1 Cache: Types of Locality

### ■ Temporal locality

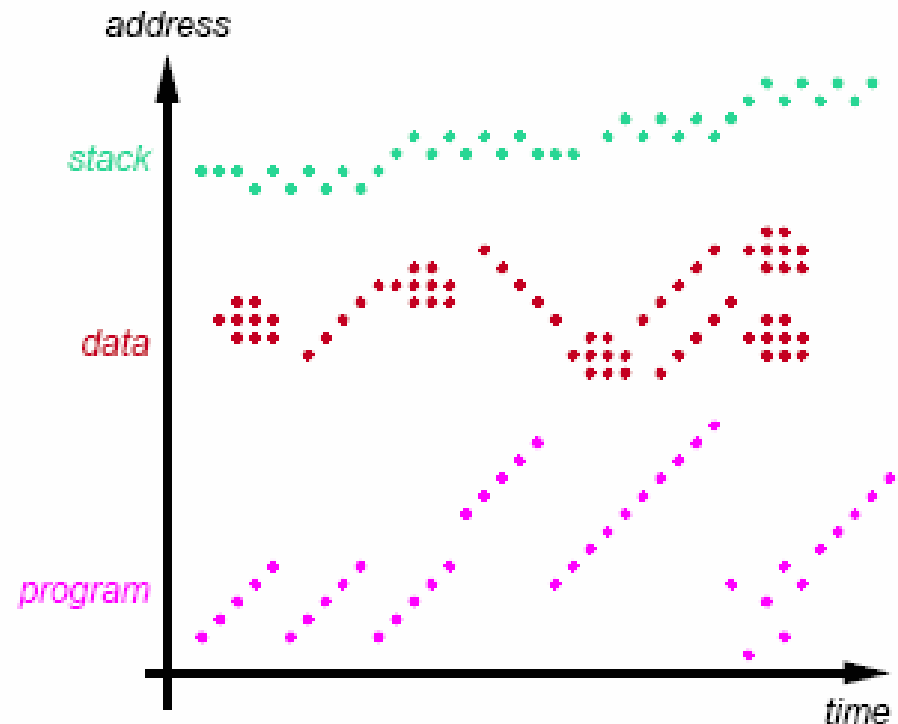
- If an item is referenced, it will tend to be referenced again soon

### ■ Spatial locality

- If an item is referenced, nearby items will tend to be referenced soon

### ■ Different locality for

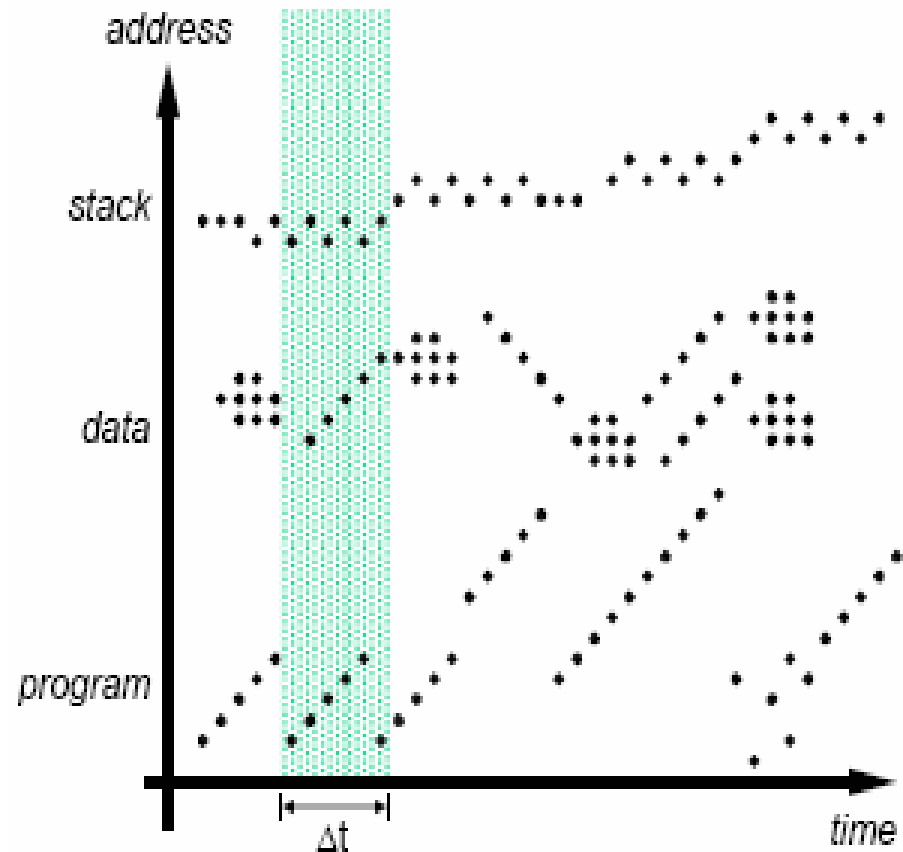
- Instructions
- Data



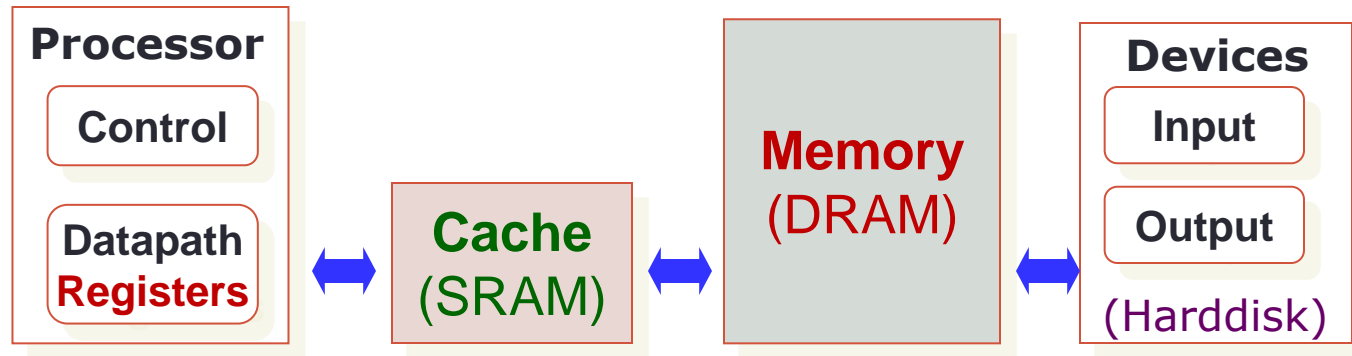
## 2.1 Working Set: Definition

- **Set of locations accessed during  $\Delta t$**
- Different phases of execution may use different working sets

Our aim is to **capture the working set and keep it in the memory closest to CPU**



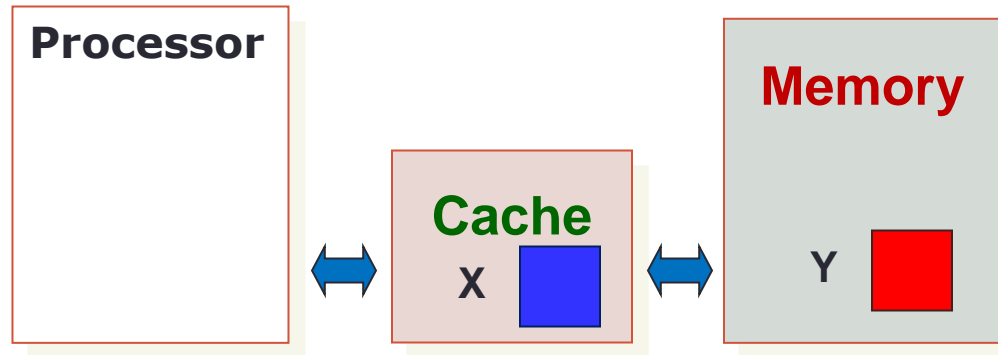
## 2.2 Two Aspects of Memory Access



- How to make SLOW main memory appear faster?
  - **Cache** – a small but fast SRAM near CPU
  - **Hardware managed:** Transparent to programmer
- How to make SMALL main memory appear bigger than it is?
  - **Virtual memory**
  - **OS managed:** Transparent to programmer
  - Not in the scope of this module (covered in CS2106)



## 2.2 Memory Access Time: Terminology



- **Hit**: Data is in cache (e.g., X)
  - **Hit rate**: Fraction of memory accesses that hit
  - **Hit time**: Time to access cache
- **Miss**: Data is not in cache (e.g., Y)
  - **Miss rate** =  $1 - \text{Hit rate}$
  - **Miss penalty**: Time to replace cache block + hit time
- Hit time < Miss penalty



## 2.2 Memory Access Time: Formula

### Average Access Time

$$= \text{Hit rate} \times \text{Hit Time} + (1 - \text{Hit rate}) \times \text{Miss penalty}$$

Example:

- Suppose our on-chip SRAM (cache) has **0.8 ns** access time, but the fastest DRAM (main memory) we can get has an access time of **10ns**. **How high a hit rate** do we need to sustain an average access time of **1ns**?

Let  $h$  be the desired hit rate.

$$1 = 0.8h + (1 - h) \times (10 + 0.8)$$

$$= 0.8h + 10.8 - 10.8h$$

$$10h = 9.8 \rightarrow h = 0.98$$

Hence we need a hit rate of **98%**.

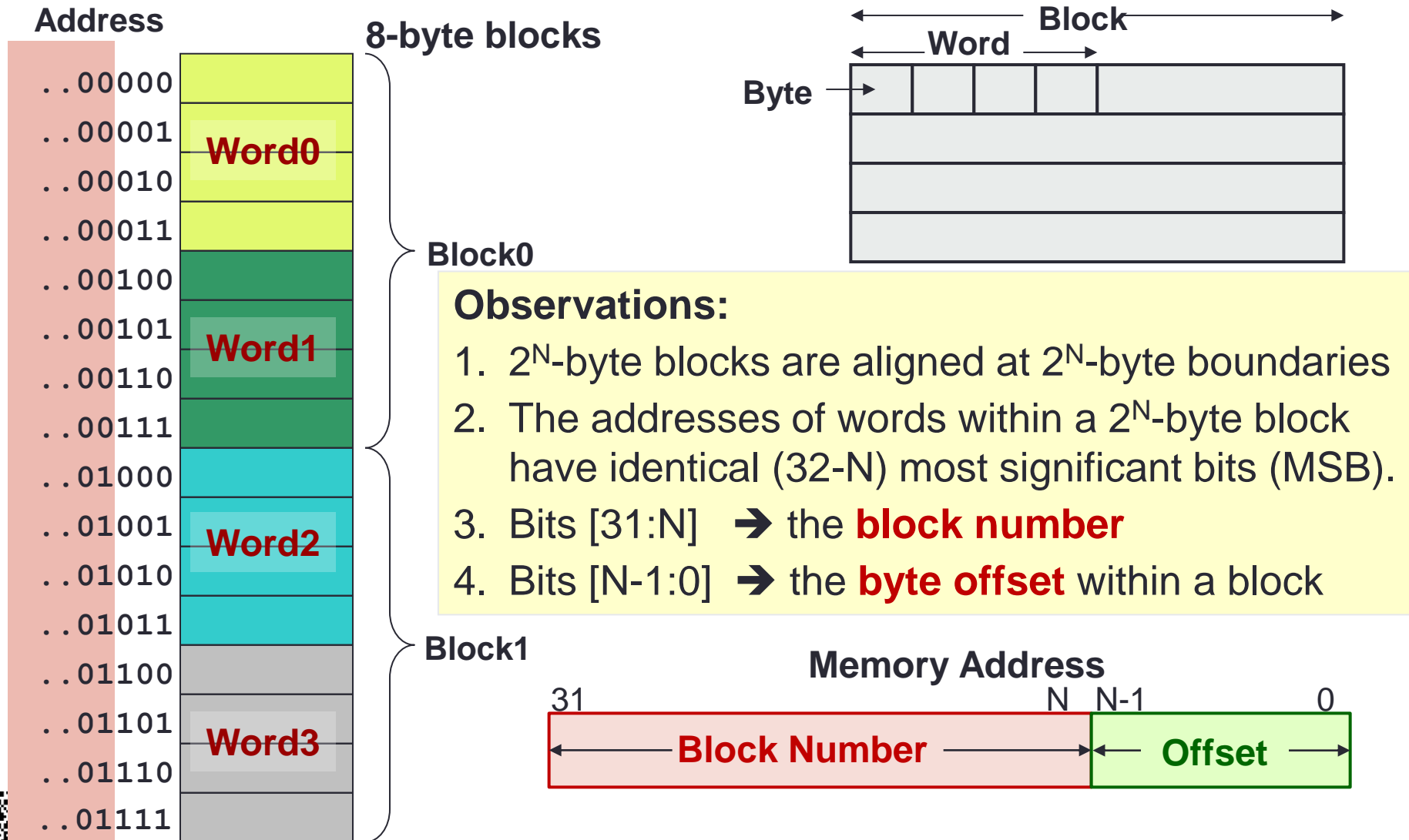


### 3. Memory to Cache Mapping (1/2)

- **Cache Block/Line:**
  - Unit of transfer between memory and cache
- Block size is typically one or more words
  - e.g.: 16-byte block  $\cong$  4-word block
  - 32-byte block  $\cong$  8-word block
- Why is the block size bigger than word size?



# 3. Memory to Cache Mapping (2/2)



## 4. Direct Mapping Analogy

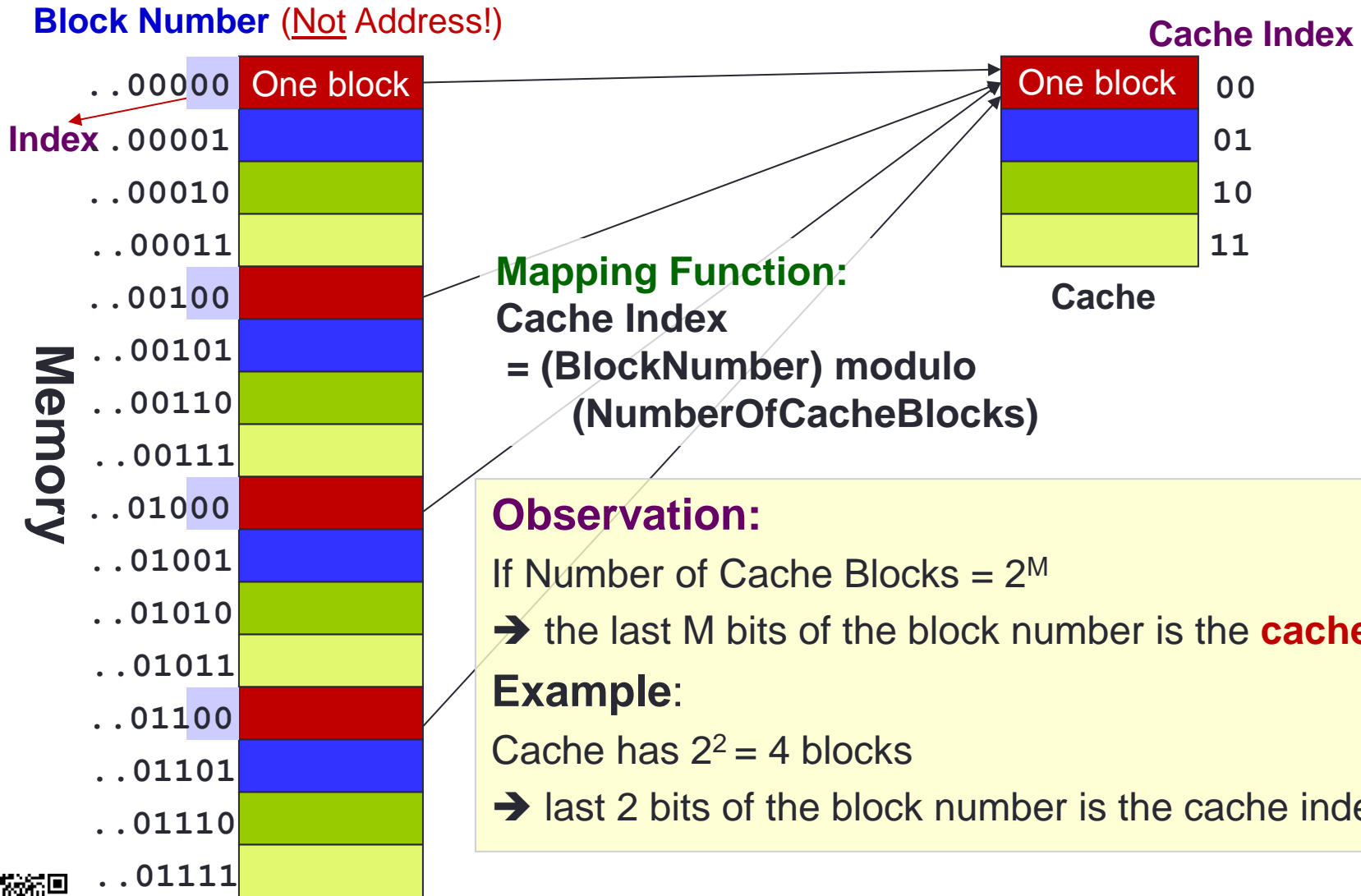


Imagine there are 26 “locations” on the desk to store books. A book’s location is determined by the first letter of its title.  
➔ Each book **has exactly one location**.

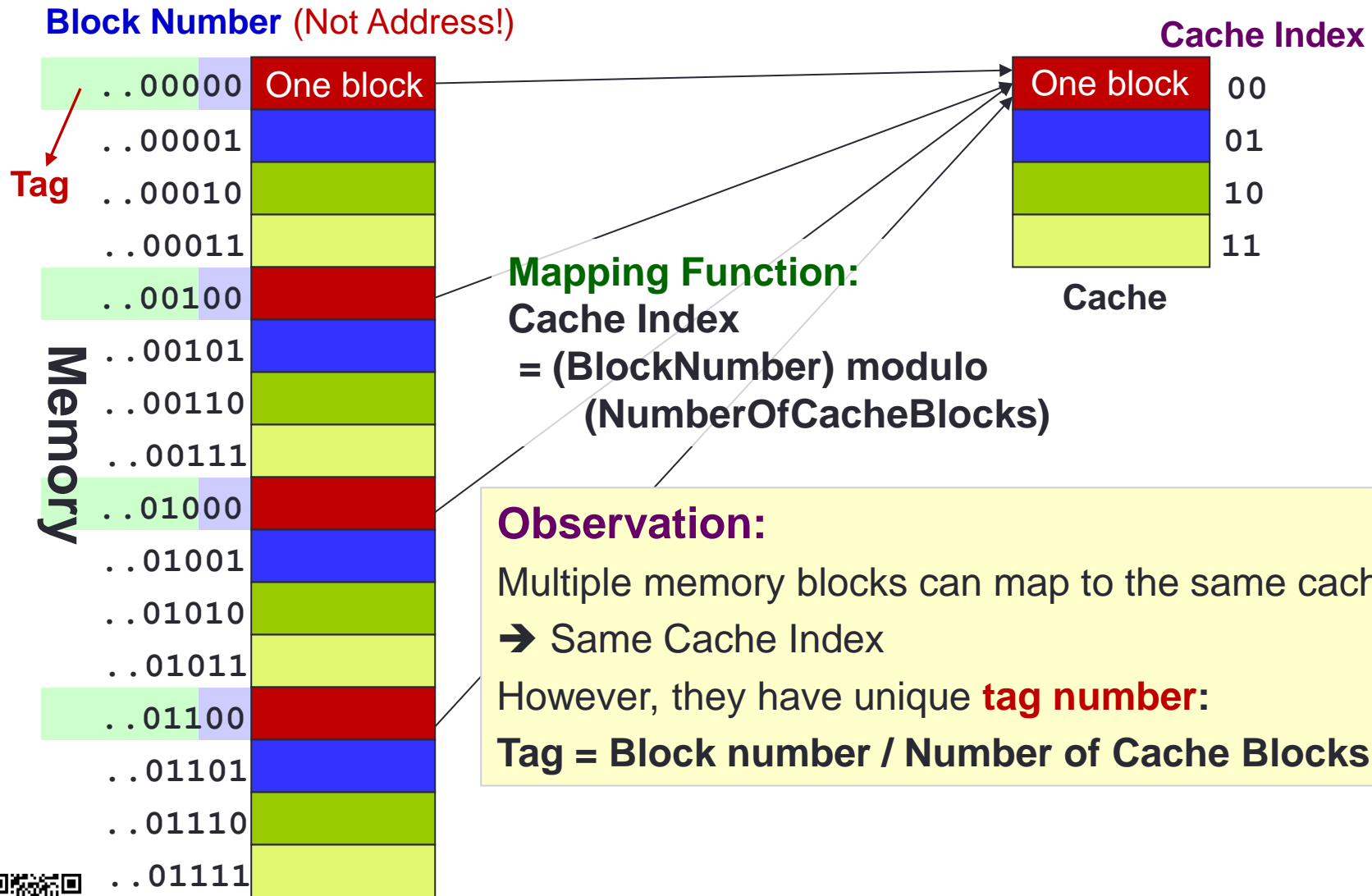




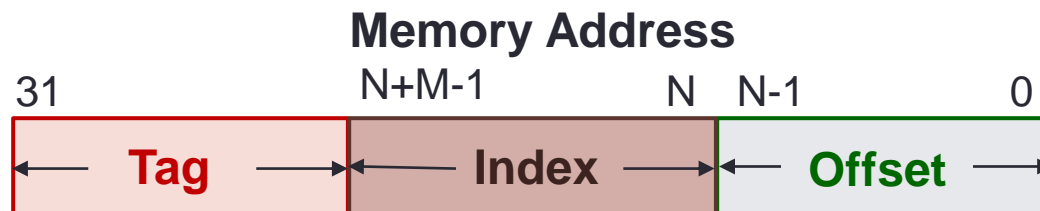
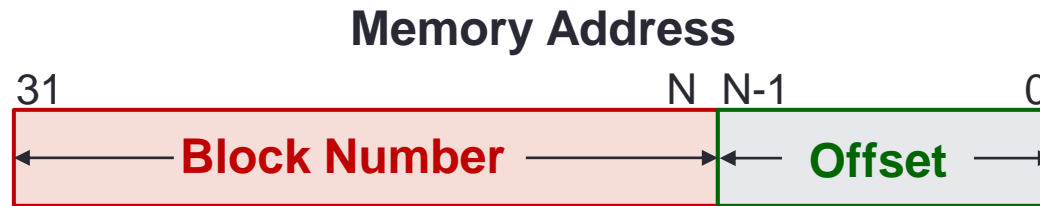
# 4. Direct Mapped Cache: Cache Index



# 4. Direct Mapped Cache: Cache Tag



## 4. Direct Mapped Cache: Mapping



Cache Block size =  $2^N$  bytes

Number of cache blocks =  $2^M$

**Offset** = **N bits**

**Index** = **M bits**

**Tag** =  **$32 - (N + M)$  bits**



## 4. Direct Mapped Cache: Cache Structure

Cache	Valid	Tag	Data	Index
				00
				01
				10
				11

Along with a data block (line), cache also contains the following administrative information (overheads):

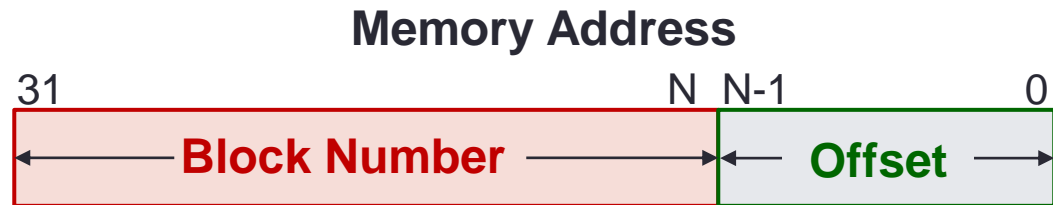
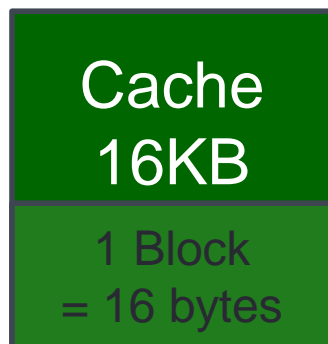
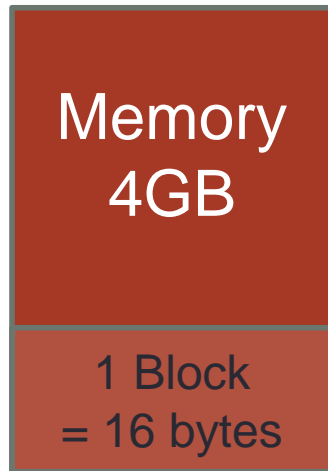
1. **Tag** of the memory block
2. **Valid bit** indicating whether the cache line contains valid data

**When is there a cache hit?**

( Valid[index] = TRUE ) **AND**  
( Tag[ index ] = Tag[ memory address ] )



## 4. Cache Mapping: Example



## Offset, **N** = 4 bits

**Block Number** =  $32 - 4 = 28$  bits

Check: Number of Blocks =  $2^{28}$



## Number of Cache Blocks

$$= 16\text{KB} / 16\text{bytes} = 1024 = 2^{10}$$

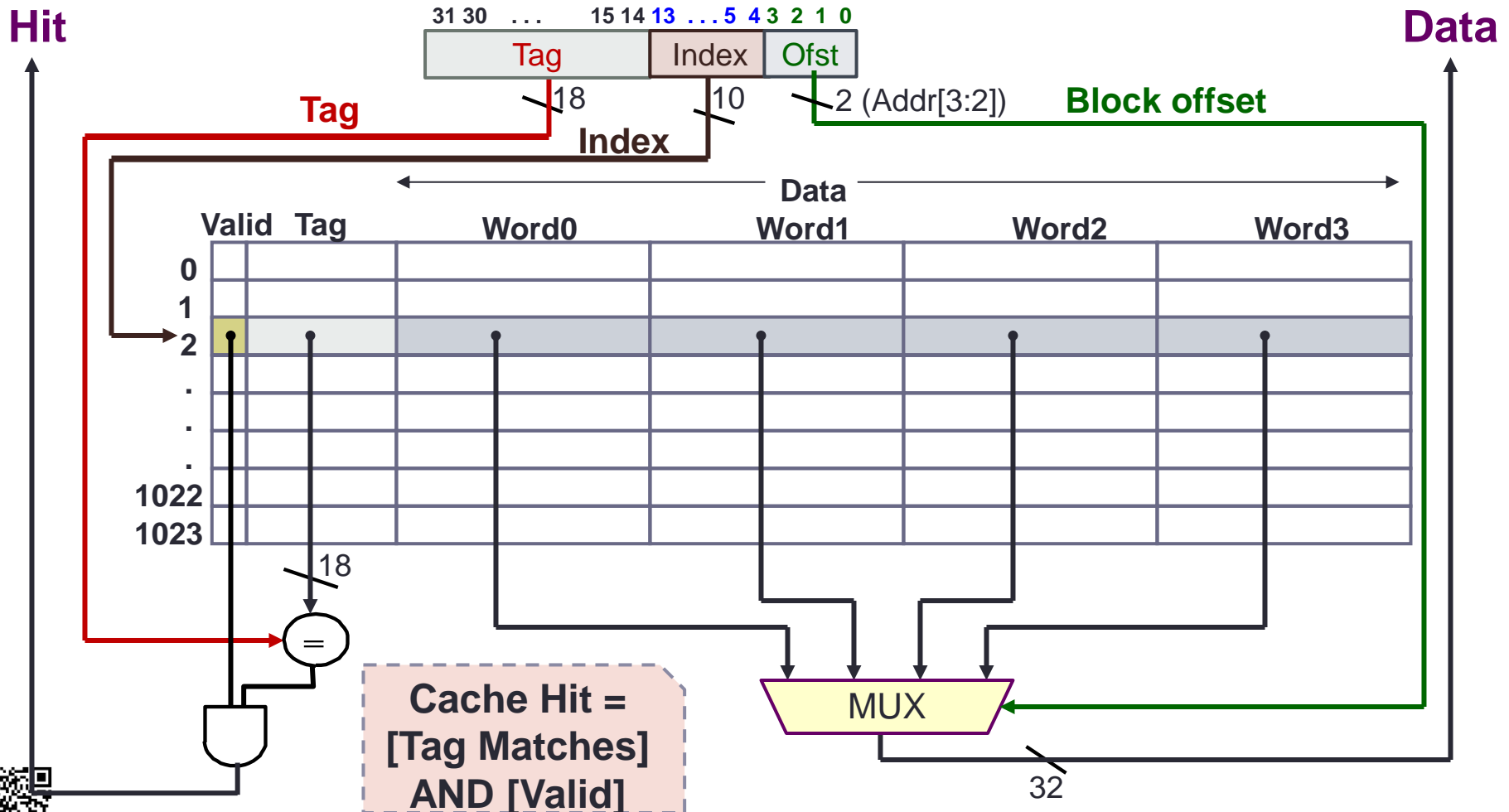
## Cache Index, **M** = 10bits

**Cache Tag** =  $32 - 10 - 4 = 18$  bits



# 4. Cache Circuitry: Example

16-KB cache:  
4-word (16-byte) blocks



## 5. Reading Data: Setup

- Given a direct mapped 16KB cache:
  - 16-byte blocks x 1024 cache blocks
- Trace the following memory accesses:

Tag														Index				Offset		
31														14	13		4	3	0	
000000000000000000000000														000000000001				0100		
000000000000000000000000														000000000001				1100		
000000000000000000000000														000000000011				0100		
000000000000000000000010														000000000001				1000		
000000000000000000000000														000000000001				0000		



## 5. Reading Data: Initial State

- Initially cache is empty  
→ All **valid** bits are zeroes (false)

		← Data →				
		Tag	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
Index	Valid					
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
			...	...	...	...
1022	0					
1023	0					





## 5. Reading Data: Load #1-1

Tag

Index

Offset

- Load from



**Step 1.** Check Cache Block at index 1

← Data →						
Index	Valid	Tag	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
...						
1022	0					
1023	0					



## 5. Reading Data: Load #1-2

Tag

Index

Offset

- Load from

00000000000000000000

0000000001

0100

**Step 2.** Data in block 1 is **invalid** [Cold/Compulsory Miss]

		Data			
		Word0	Word1	Word2	Word3
		Bytes 0-3	Bytes 4-7	Bytes 8-11	Bytes 12-15
Index	Valid	Tag			
0	0				
1	0				
2	0				
3	0				
4	0				
5	0				
...					
1022	0				
1023	0				



## 5. Reading Data: Load #1-3

- Load from 

Tag	Index	Offset
00000000000000000000	0000000001	0100

**Step 3.** Load 16 bytes from memory; Set **Tag** and **Valid** bit

			← Data →			
Index	Valid	Tag	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	0	A	B	C	D
2	0					
3	0					
4	0					
5	0					
... ..						
1022	0					
1023	0					



## 5. Reading Data: Load #1-4

Tag

Index

Offset

- Load from

00000000000000000000	0000000001	0100
----------------------	------------	------



**Step 4.** Return **Word1** (byte offset = 4) to Register

← Data →					
Index	Valid	Tag	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11
0	0				
1	1	0	A	B	C
2	0				
3	0				
4	0				
5	0				
...					
1022	0				
1023	0				



## 5. Reading Data: Load #2-1

Tag

Index

Offset

- Load from



**Step 1.** Check Cache Block at index 1

		← Data →				
Index	Valid	Tag	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	0	A	B	C	D
2	0					
3	0					
4	0					
5	0					
...						
1022	0					
1023	0					



## 5. Reading Data: Load #2-2

- Load from 

Tag	Index	Offset
00000000000000000000	00000000001	1100

**Step 2.** [Cache Block is Valid] AND [Tags match] → Cache hit!

		← Data →				
		Tag	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
Index	Valid					
0	0					
1	1	0	A	B	C	D
2	0					
3	0					
4	0					
5	0					
...						
1022	0					
1023	0					



## 5. Reading Data: Load #2-3

Tag

Index

Offset

- Load from

00000000000000000000	0000000001	1100
----------------------	------------	------

**Step 3.** Return **Word3** (byte offset = 12) to Register **[Spatial Locality]**

← Data →						
Index	Valid	Tag	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	0	A	B	C	D
2	0					
3	0					
4	0					
5	0					
...						
1022	0					
1023	0					



## 5. Reading Data: Load #3-1

Tag

Index

Offset

- Load from



**Step 1.** Check Cache Block at index 3

		← Data →				
Index	Valid	Tag	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	0	A	B	C	D
2	0					
3	0					
4	0					
5	0					
...						
1022	0					
1023	0					





## 5. Reading Data: Load #3-2

Tag

Index

Offset

- Load from

00000000000000000000

0000000011

0100

**Step 2.** Data in block 3 is **invalid** [Cold/Compulsory Miss]

		← Data →				
Index	Valid	Tag	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	0	A	B	C	D
2	0					
3	0					
4	0					
5	0					
...						
1022	0					
1023	0					



## 5. Reading Data: Load #3-3

- Load from 

Tag	Index	Offset
00000000000000000000	0000000011	0100

**Step 3.** Load 16 bytes from memory; Set **Tag** and **Valid** bit

		Data				
Index	Valid	Tag	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	0	A	B	C	D
2	0					
3	1	0	I	J	K	L
4	0					
5	0					
...						
1022	0					
1023	0					



## 5. Reading Data: Load #3-4

Tag

Index

Offset

- Load from

00000000000000000000	0000000011	0100
----------------------	------------	------



**Step 4.** Return **Word1** (byte offset = 4) to Register

		Data				
Index	Valid	Tag	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	0	A	B	C	D
2	0					
3	1	0	I	J	K	L
4	0					
5	0					
...						
1022	0					
1023	0					



## 5. Reading Data: Load #4-1

Tag

Index

Offset

- Load from

00000000000000000010

0000000001

1000

**Step 1.** Check Cache Block at index 1

← Data →						
Index	Valid	Tag	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	0	A	B	C	D
2	0					
3	1	0	I	J	K	L
4	0					
5	0					
...						
1022	0					
1023	0					



## 5. Reading Data: Load #4-2

- Load from 

Tag	Index	Offset
00000000000000000010	00000000001	1000

**Step 2.** Cache block is **Valid** but **Tags mismatch** [Cold miss]

← Data →						
Index	Valid	Tag	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	0	A	B	C	D
2	0					
3	1	0	I	J	K	L
4	0					
5	0					
... ..						
1022	0					
1023	0					



## 5. Reading Data: Load #4-3

- Load from 

Tag	Index	Offset
0000000000000000000010	000000000001	1000

**Step 3.** Replace block 1 with new data; Set Tag

		← Data →				
Index	Valid	Tag	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	2	E	F	G	H
2	0					
3	1	0	I	J	K	L
4	0					
5	0					
...						
1022	0					
1023	0					



## 5. Reading Data: Load #4-4

Tag

Index

Offset

- Load from

00000000000000000010

0000000001

1000

**Step 4.** Return **Word2** (byte offset = 8) to Register

← Data →						
Index	Valid	Tag	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	2	E	F	G	H
2	0					
3	1	0	I	J	K	L
4	0					
5	0					
...						
1022	0					
1023	0					



## 5. Reading Data: Load #5-1

Tag

Index

Offset

- Load from

00000000000000000000	0000000001	0000
----------------------	------------	------



**Step 1.** Check Cache Block at index 1

← Data →						
Index	Valid	Tag	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	2	E	F	G	H
2	0					
3	1	0	I	J	K	L
4	0					
5	0					
...						
1022	0					
1023	0					





## 5. Reading Data: Load #5-2

- Load from 

Tag	Index	Offset
00000000000000000000	0000000001	0000

**Step 2.** Cache block is **Valid** but **Tags mismatch** [**Cold miss**]

		← Data →				
		Tag	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
Index	Valid					
0	0					
1	1	2	E	F	G	H
2	0					
3	1	0	I	J	K	L
4	0					
5	0					
...						
1022	0					
1023	0					



## 5. Reading Data: Load #5-3

- Load from 

Tag	Index	Offset
00000000000000000000	0000000001	0000

**Step 3.** Replace block 1 with new data; Set Tag

		← Data →				
Index	Valid	Tag	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	0	A	B	C	D
2	0					
3	1	0	I	J	K	L
4	0					
5	0					
...						
1022	0					
1023	0					



## 5. Reading Data: Load #5-4

- Load from 

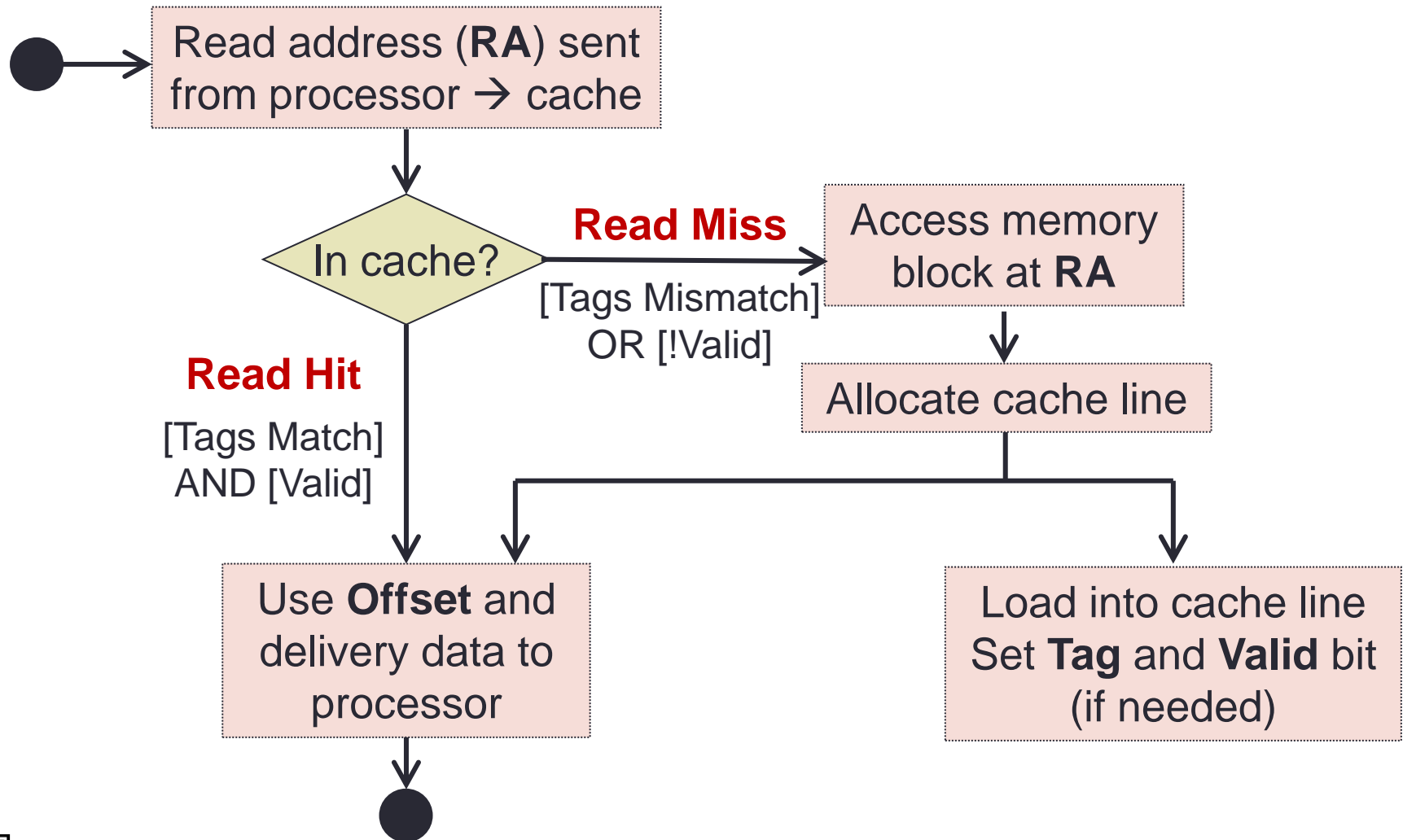
Tag	Index	Offset
00000000000000000000	0000000001	0000

**Step 4.** Return **Word0** (byte offset = 0) to Register

		← Data →				
Index	Valid	Tag	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	0	A	B	C	D
2	0					
3	1	0	I	J	K	L
4	0					
5	0					
...						
1022	0					
1023	0					



## 5. Reading Data: Summary

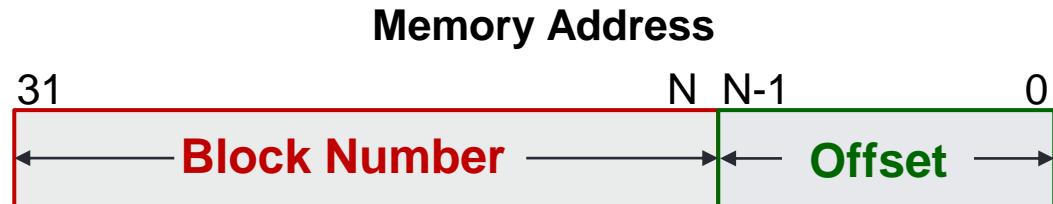
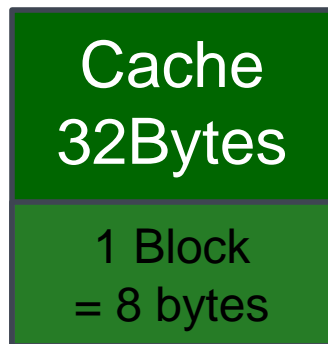
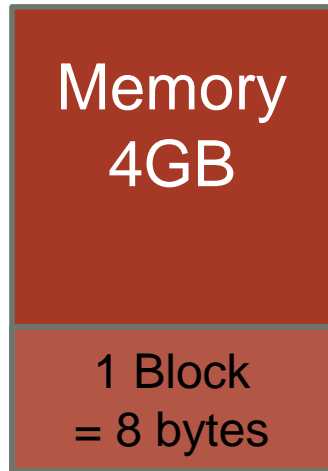


## 6. Types of Cache Misses

- **Compulsory misses**
  - On the first access to a block; the block must be brought into the cache
  - Also called **cold start misses** or **first reference misses**
- **Conflict misses**
  - Occur in the case of direct mapped cache or set associative cache, when several blocks are mapped to the same block/set
  - Also called **collision misses** or **interference misses**
- **Capacity misses**
  - Occur when blocks are discarded from cache as cache cannot contain all blocks needed



# Exercise #1: Setup Information



Offset,  $N = 3$

Block Number = 29



Number of Cache Blocks = 4

Cache Index,  $M = 2$

Cache Tag = 27



# Exercise #2: Tracing Memory Accesses

- Using the given setup in exercise #1, trace the following memory loads:
  - Load from addresses:  
**4, 0, 8, 12, 36, 0, 4**
- Note that “A”, “B”.... “J” represent word-size data
  - Assume 1 word = 4 bytes

## Memory Content

Addr	Data
0	A
4	B
8	C
12	D
...	...
32	I
36	J
...	...







# Exercise #2: Load #2

Addresses: 4, 0, 8, 12, 36, 0, 4

Miss Hit

Address 0 = 00000000000000000000000000000000 00 000

Addr.	Data
0	A
4	B
8	C
12	D
...	...
32	I
36	J

Index	Valid	Tag	Word0	Word1
0	1	0	A	B
1	0			
2	0			
3	0			



# Exercise #2: Load #3

Miss Hit Miss

Addresses: 4, 0, **8**, 12, 36, 0, 4

Tag

Index Offset

Address 8 =

00000000000000000000000000000000	01	000
----------------------------------	----	-----

Addr.	Data
0	A
4	B
8	C
12	D
...	...
32	I
36	J

Index	Valid	Tag	Word0	Word1
0	1	0	A	B
1	<del>0</del> 1	0	<b>C</b>	D
2	0			
3	0			







# Exercise #2: Load #6

Miss Hit Miss Hit Miss Miss

Addresses: 4, 0, 8, 12, 36, **0**, 4

Addr.	Data
0	A
4	B
8	C
12	D
...	...
32	I
36	J

Address 0 = 00000000000000000000000000000000 00 000

Tag Index Offset

Index	Valid	Tag	Word0	Word1
0	1	<del>0</del> 10	<del>A</del> I <b>A</b>	<del>B</del> JB
1	1	0	C	D
2	0			
3	0			



# Exercise #2: Load #7

Addresses: 4, 0, 8, 12, 36, 0, **4**

Miss Hit Miss Hit Miss Miss Hit

Addr.	Data
0	A
4	B
8	C
12	D
...	...
32	I
36	J

Address 4 = 00000000000000000000000000000000 00 100

Tag Index Offset

Index	Valid	Tag	Word0	Word1
0	1	<del>0</del> 10	<del>A</del> I A	<del>B</del> J <b>B</b>
1	1	0	C	D
2	0			
3	0			



# 7. Writing Data: Store #1-1

- Store **X** to 

Tag	Index	Offset
00000000000000000000	000000000001	1000



## Step 1. Check Cache Block 1

← Data →						
Index	Valid	Tag	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
0	0					
1	1	0	A	B	C	D
2	0					
3	1	0	I	J	K	L
4	0					
5	0					
... ..						
1022	0					
1023	0					



## 7. Writing Data: Store #1-2

- Store **X** to 

Tag	Index	Offset
00000000000000000000	0000000001	1000



**Step 2.** [Cache Block is Valid] AND [Tags match] → Cache hit!

← Data →

		Tag	Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
Index	Valid					
0	0					
1	1	0	A	B	C	D
2	0					
3	1	0	I	J	K	L
4	0					
5	0					
... ..						
1022	0					
1023	0					





## 7. Writing Data: Store #1-3

- Store **X** to 

Tag	Index	Offset
00000000000000000000	0000000001	1000

**Step 2.** Replace Word2 (offset = 8) with **X**

← Data →

See any problem here?

			Word0 Bytes 0-3	Word1 Bytes 4-7	Word2 Bytes 8-11	Word3 Bytes 12-15
Index	Valid	Tag				
0	0					
1	1	0	A	B	X	D
2	0					
3	1	0	I	J	K	L
4	0					
5	0					
... ..						
1022	0					
1023	0					

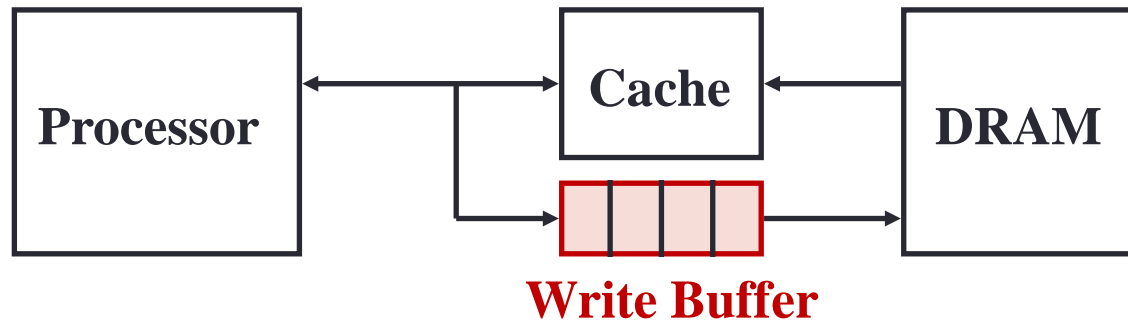


## 8. Changing Cache Content: **Write Policy**

- Cache and main memory are inconsistent
  - Modified data only in cache, not in memory!
- **Solution 1: Write-through** cache
  - Write data both to cache and to main memory
- **Solution 2: Write-back** cache
  - Only write to cache
  - Write to main memory only when cache block is replaced (evicted)



## 8. Write-Through Cache



- **Problem:**
  - Write will operate at the speed of main memory!
- **Solution:**
  - Put a write buffer between cache and main memory
    - Processor: writes data to cache + write buffer
    - Memory controller: write contents of the buffer to memory



## 8. Write-Back Cache

- **Problem:**

- Quite wasteful if we write back every evicted cache blocks

- **Solution:**

- Add an additional bit (**Dirty bit**) to each cache block
- Write operation will change dirty bit to 1
  - Only cache block is updated, no write to memory
- When a cache block is replaced:
  - Only write back to memory if dirty bit is 1

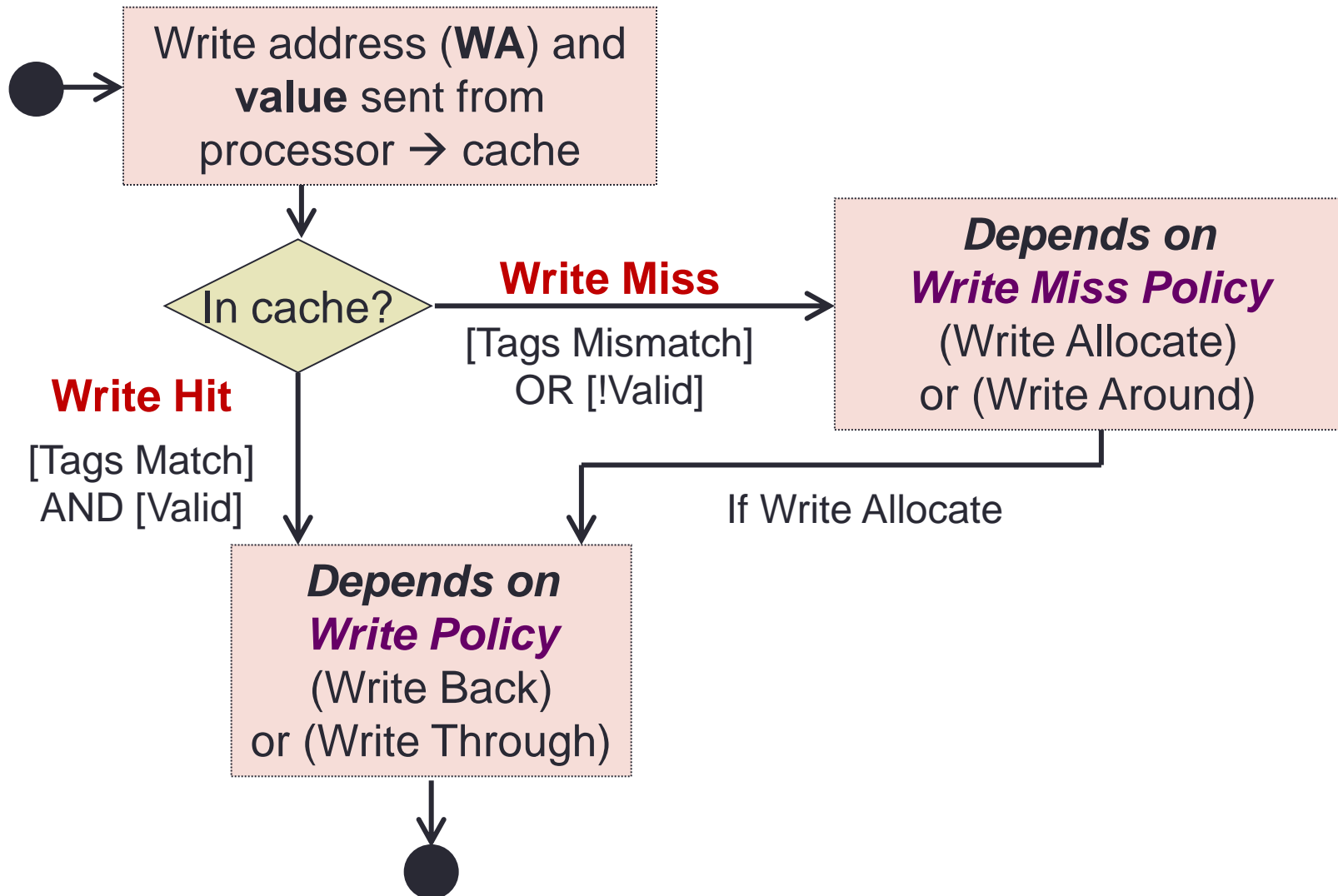


## 8. Handling Cache Misses

- On a **Read Miss**:
    - Data loaded into cache and then load from there to register
- 
- **Write Miss** option 1: **Write allocate**
    - Load the complete block into cache
    - Change only the required word in cache
    - Write to main memory depends on write policy
  - **Write Miss** option 2: **Write around**
    - Do not load the block to cache
    - Write directly to **main memory only**



## 8. Writing Data: Summary



# Summary

- Memory hierarchy gives the illusion of a fast and big memory
- Hardware-managed cache is an integral component of today's processors
- Next lecture: How to improve cache performance



# Reading

- **Large and Fast: Exploiting Memory Hierarchy**
  - Chapter 7 sections 7.1 – 7.2 (3<sup>rd</sup> edition)
  - Chapter 5 sections 5.1 – 5.2 (4<sup>th</sup> edition)





# End of File





