ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

COMPUTER ORGANISATION
(TỔ CHỨC MÁY TÍNH)

Cache Part I

Acknowledgement

- The contents of these slides have origin from School of Computing, National University of Singapore.
- We greatly appreciate support from Mr. Aaron Tan Tuck Choy for kindly sharing these materials.

Policies for students

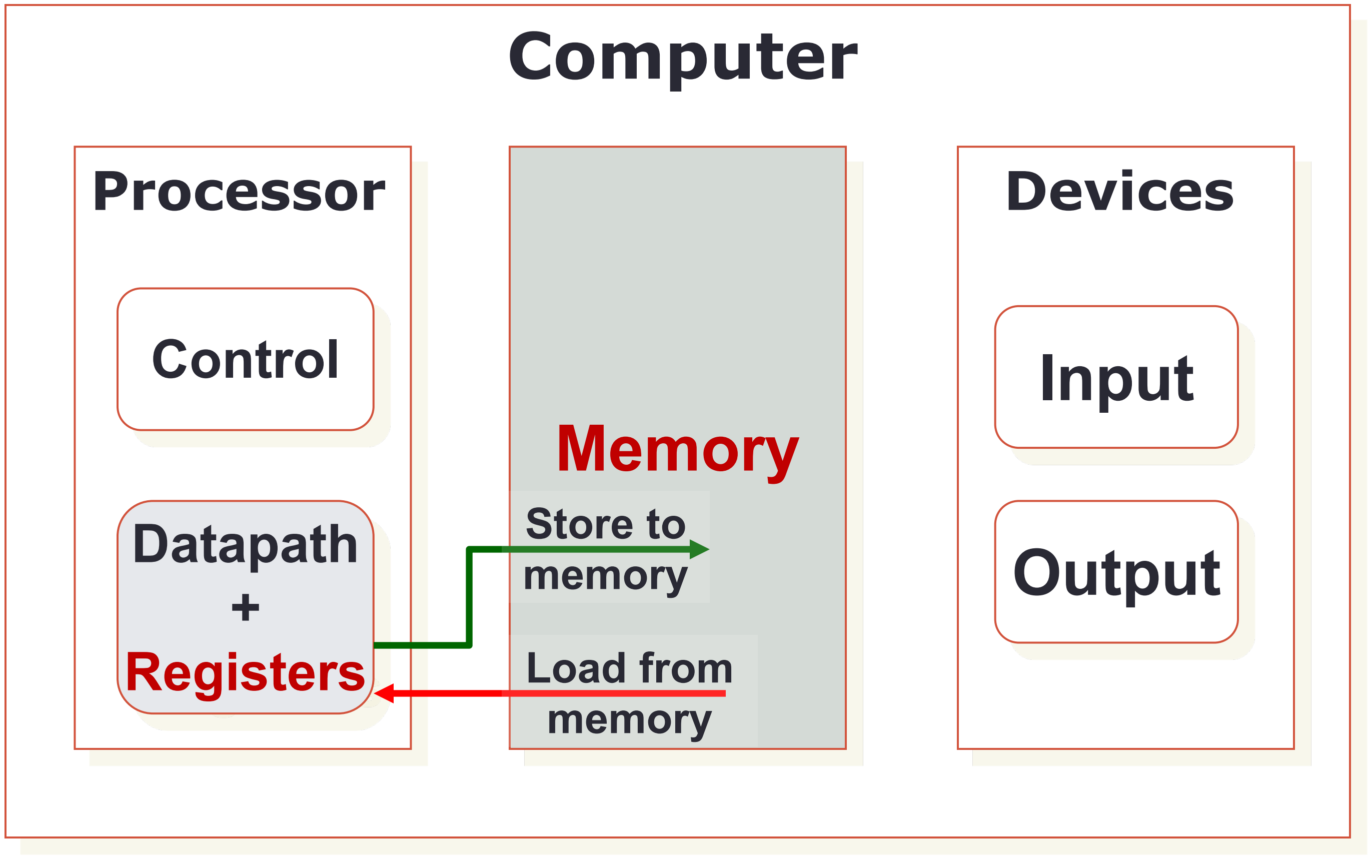
- These contents are only used for students PERSONALLY.
- Students are NOT allowed to modify or deliver these contents to anywhere or anyone for any purpose.

Road Map: **Part II**

- Performance
- Assembly Language
- Processor: Datapath
- Processor: Control
- Pipelining
- **Cache**

- **Cache Part I**
  - Memory Hierarchy
  - The Principle of Locality
  - The Cache Principle
  - Direct-Mapped Cache
  - Cache Structure and Circuitry
  - Cache Write and Write Miss Policy

Data Transfer: The Big Picture

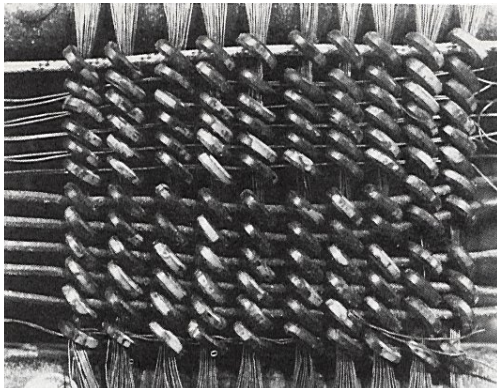


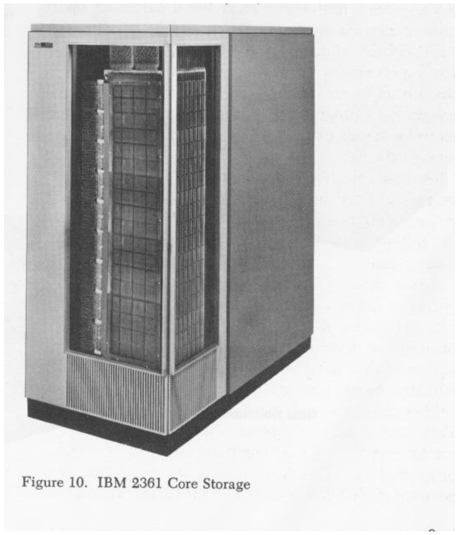
Registers are in the datapath of the processor. If operands are in memory we have to **load** them to processor (registers), operate on them, and **store** them back to memory

Memory Technology : 1950s

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

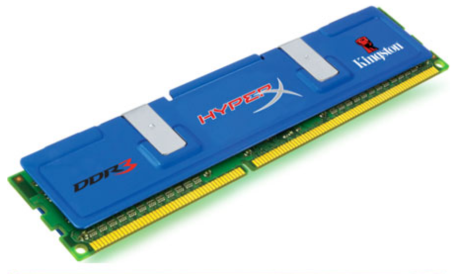
Maurice Wilkes: 2005





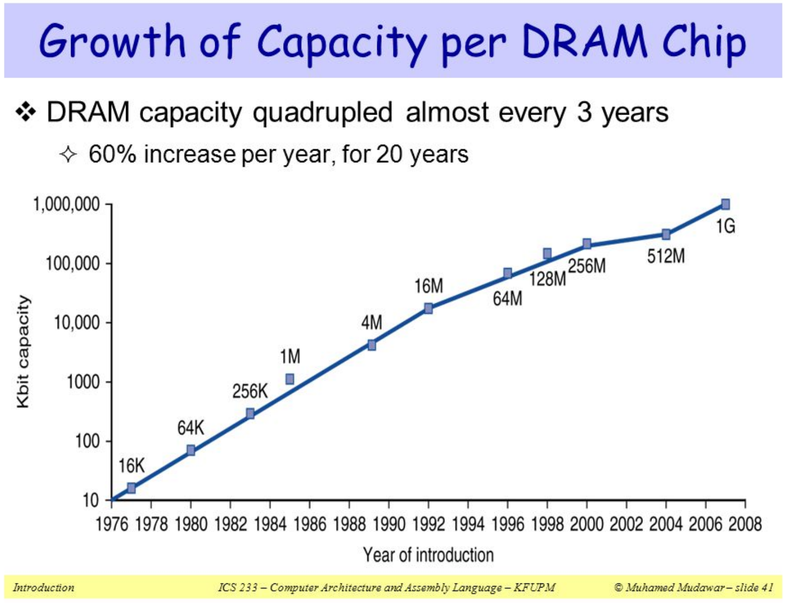
1952: IBM 2361 16KB magnetic core memory

Memory Technology Today: **DRAM**



- **DDR SDRAM**
  - **D**ouble **D**ata **R**ate
    - – **S**ynchronous **D**ynamic **RAM**
  - The dominant memory technology in PC market
  - Delivers memory on the positive and negative edge of a clock (double rate)
  - Generations:
    - DDR (MemClkFreq x 2(double rate) x 8 words)
    - DDR2 (MemClkFreq x 2(multiplier) x 2 x 8 words)
    - DDR3 (MemClkFreq x 4(multiplier) x 2 x 8 words)
    - DDR4 (due 2014)

DRAM Capacity Growth

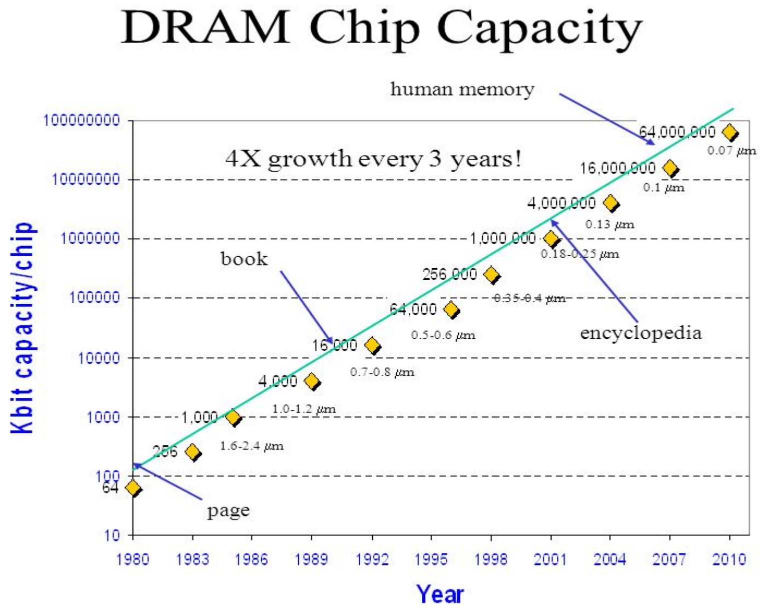


Growth of Capacity per DRAM Chip

❖ DRAM capacity quadrupled almost every 3 years

✧ 60% increase per year, for 20 years

Introduction *ICS 233 – Computer Architecture and Assembly Language – KFUPM* *© Muhamed Mudawar – slide 41*



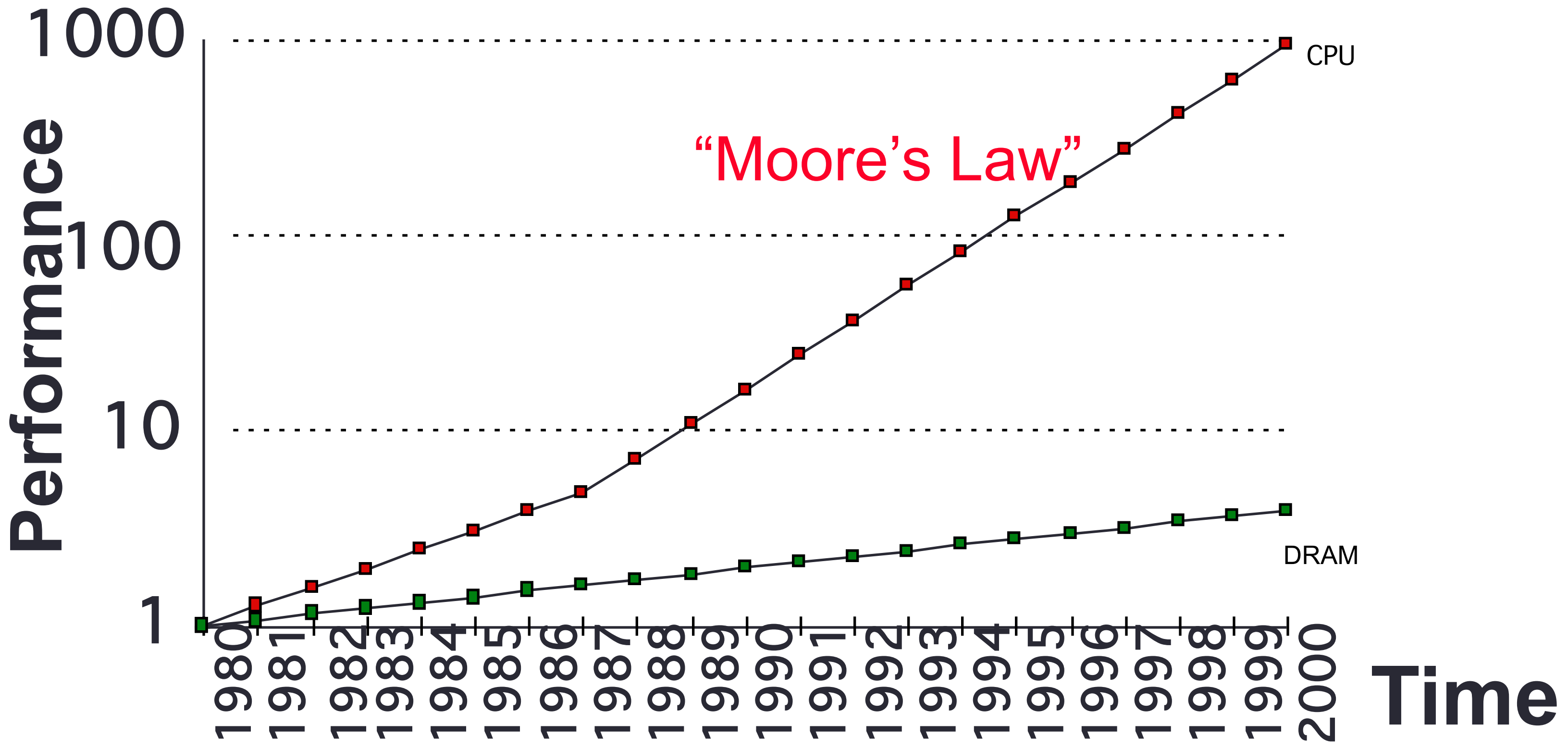
- Unprecedented growth in density, but we still have a problem

Processor-DRAM Performance Gap

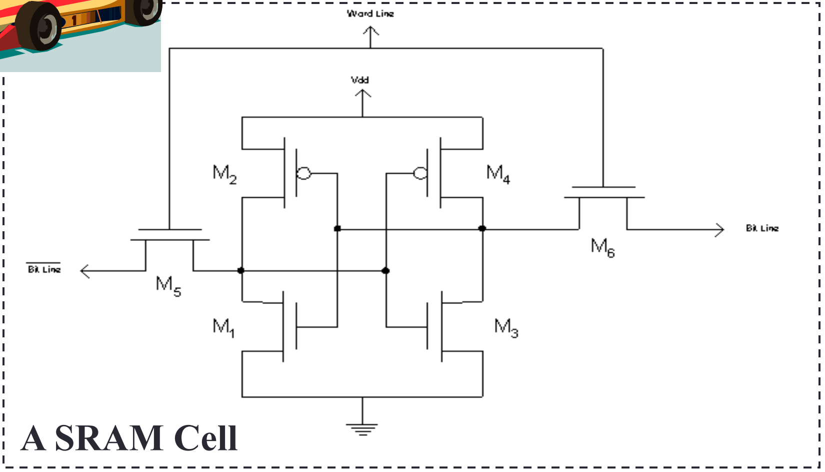
Memory Wall:

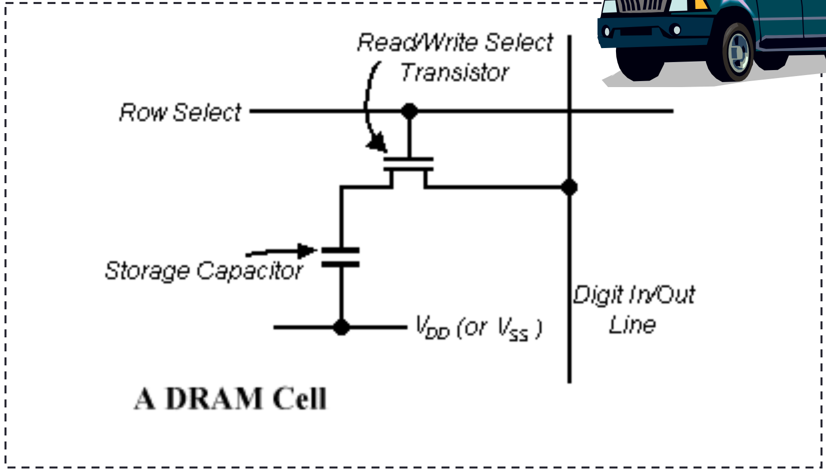
1 GHz Processor → 1 ns per clock cycle

50 ns for DRAM access → 50 processor clock cycles per memory access !!



Faster Memory Technology: **SRAM**





SRAM

6 transistors per memory cell

→ **Low density**

Fast access latency of 0.5 – 5 ns

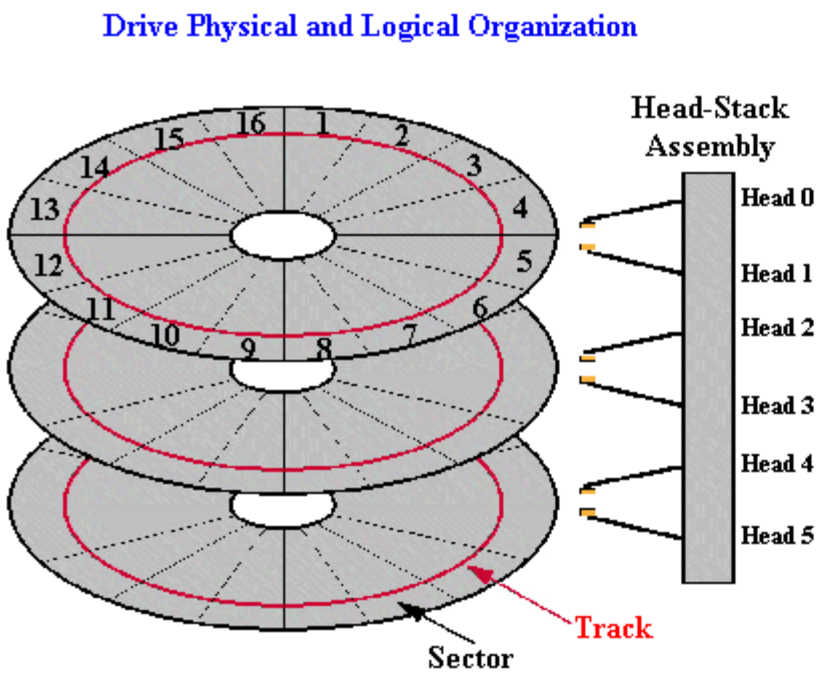
DRAM

1 transistor per memory cell

→ **High density**

Slow access latency of 50-70ns

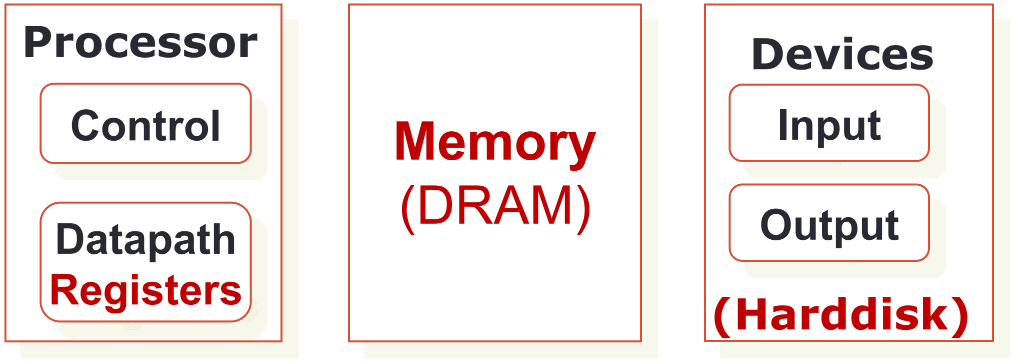
Slow Memory Technologies: **Magnetic Disk**

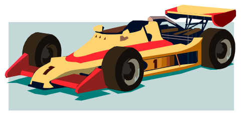


Typical high-end hard disk:

Average Latency: 4 - 10 ms
Capacity: 500-2000GB

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
Ideal	**1 GB**	**1 ns**	**Cheap**

Best of Both Worlds

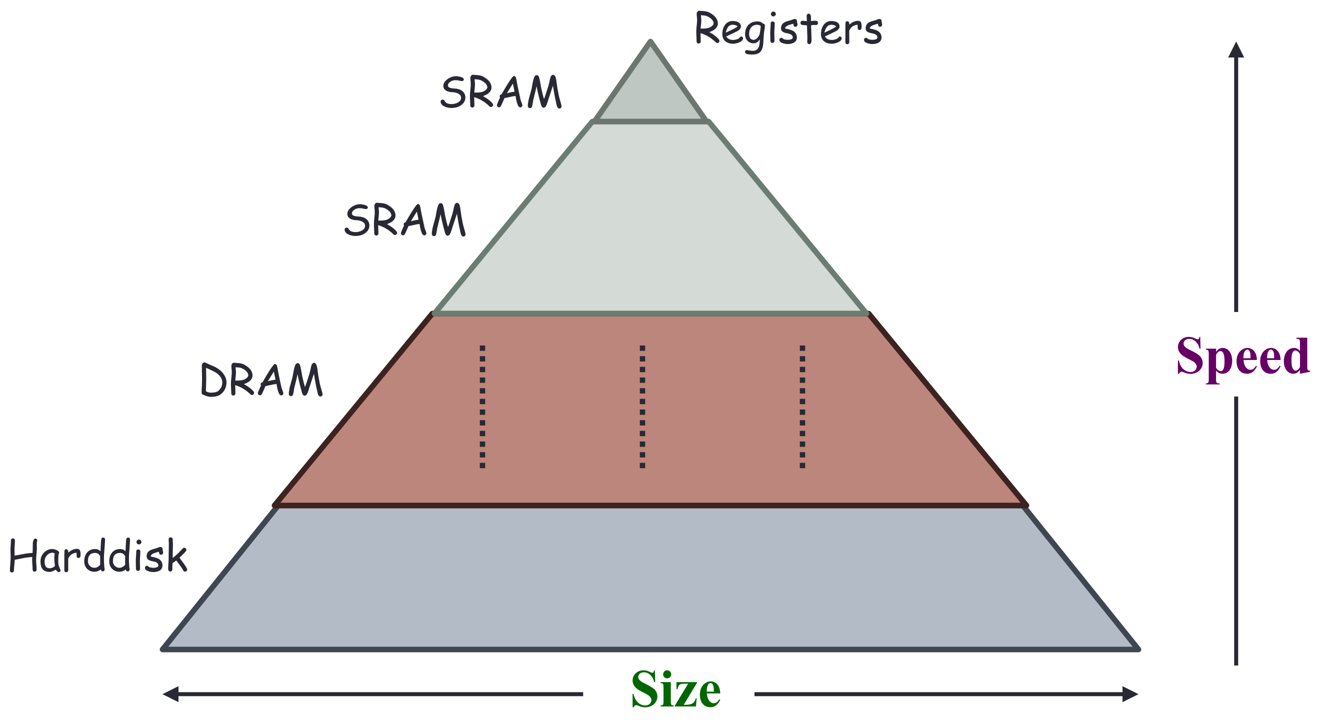
- What we want:
  - A **BIG** and **FAST** memory
  - Memory system should perform like 1GB of SRAM (1ns access time) but cost like 1GB of slow memory

Key concept:

Use a hierarchy of memory technologies:

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

Memory Hierarchy: Illustration



The Library Analogy

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

Solution: Book on the Table!

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?

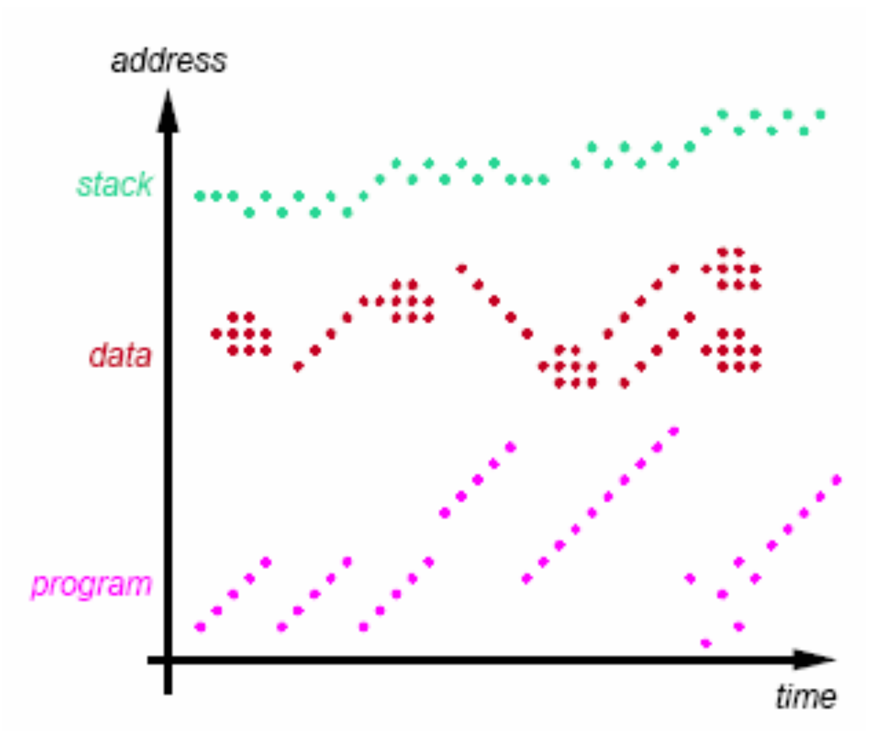
Cache: The Basic Idea

- 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

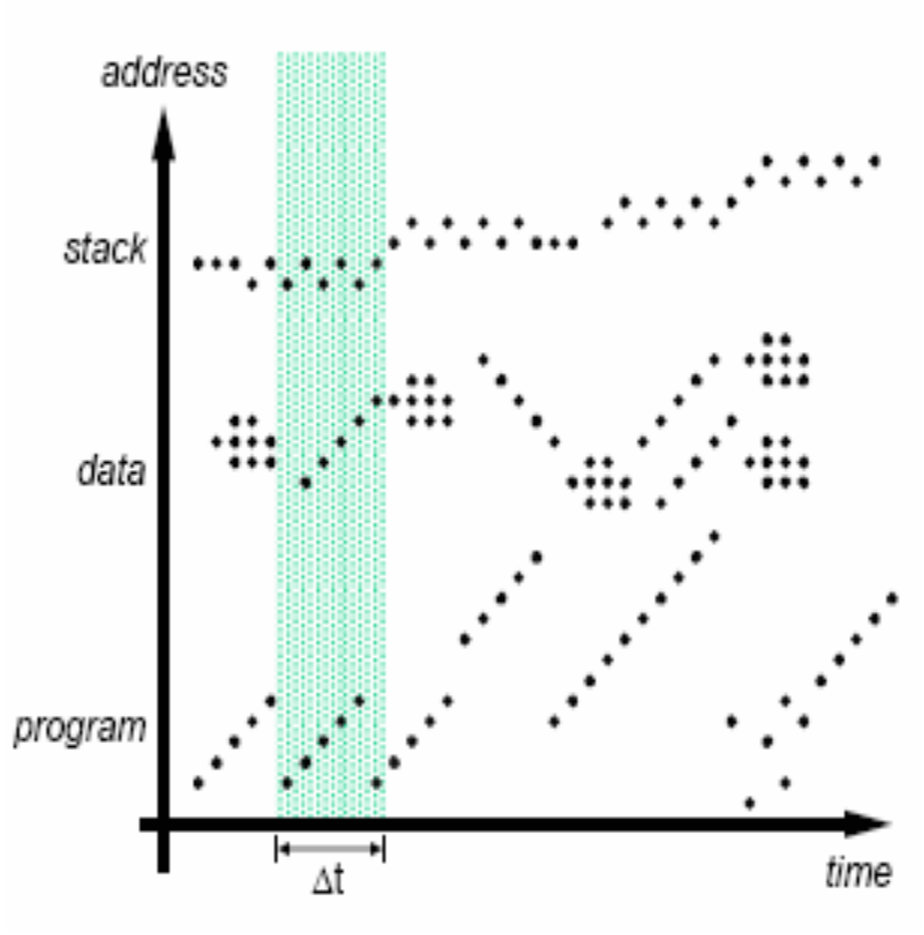
Program accesses only a small portion of the memory address space within a small time interval

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

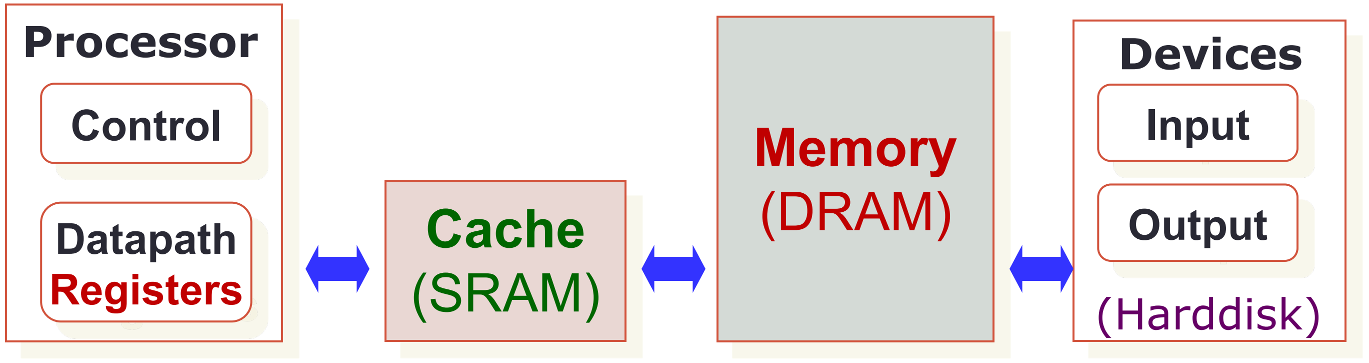
Working Set: Definition



- **Set of locations accessed during Δt**
- Different phases of execution may use different working sets

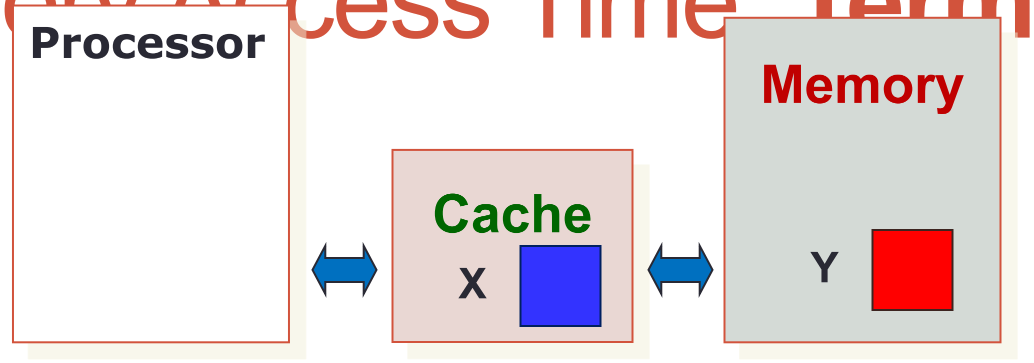
Our aim is to **capture the working set and keep it in the memory closest to CPU**

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

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 – Hit rate
  - **Miss penalty**: Time to replace cache block + deliver data
- Hit time < Miss penalty

Memory Access Time: **Formula**

Average Access Time

= Hit rate x Hit Time + (1-Hit rate) x 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**?

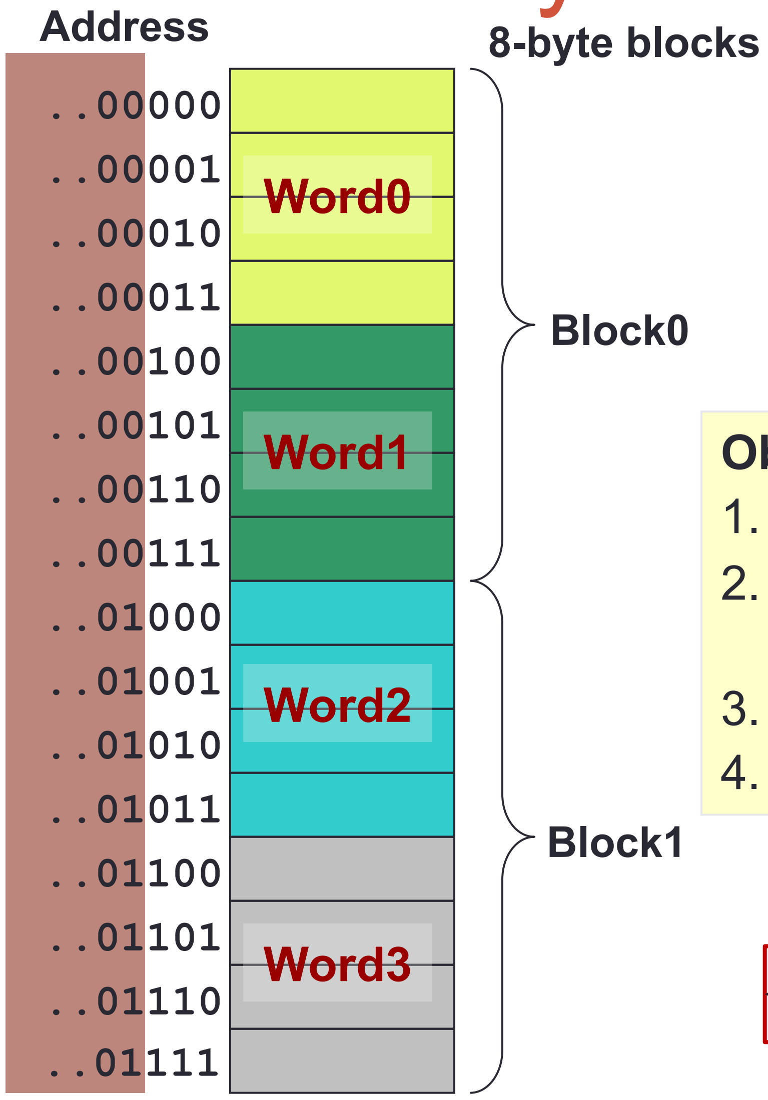
MEMORY ←→ CACHE MAPPING

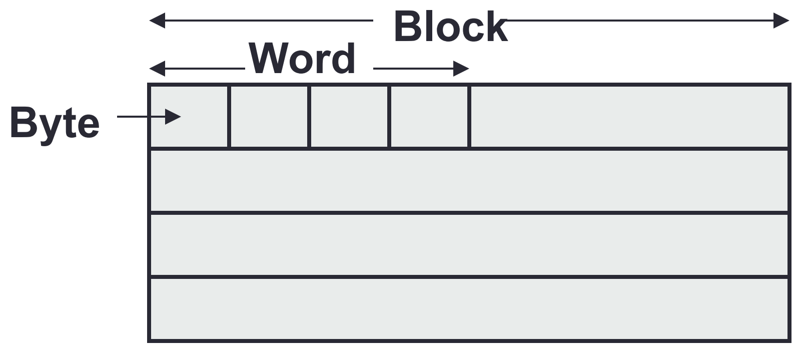
Where to place the memory block in cache?

Preliminary: Cache Block/Line (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 ≅ 4-word block
  - 32-byte block ≅ 8-word block
- Why block size is bigger than word size?

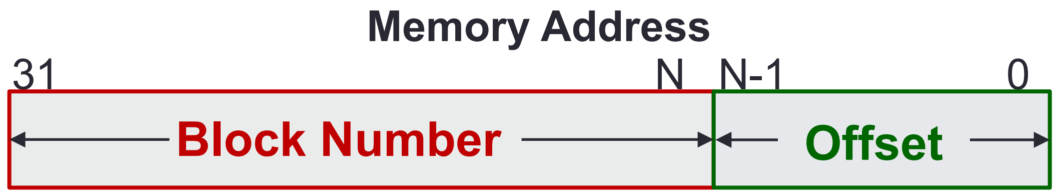
Preliminary: Cache Block/Line (2/2)





Observations:

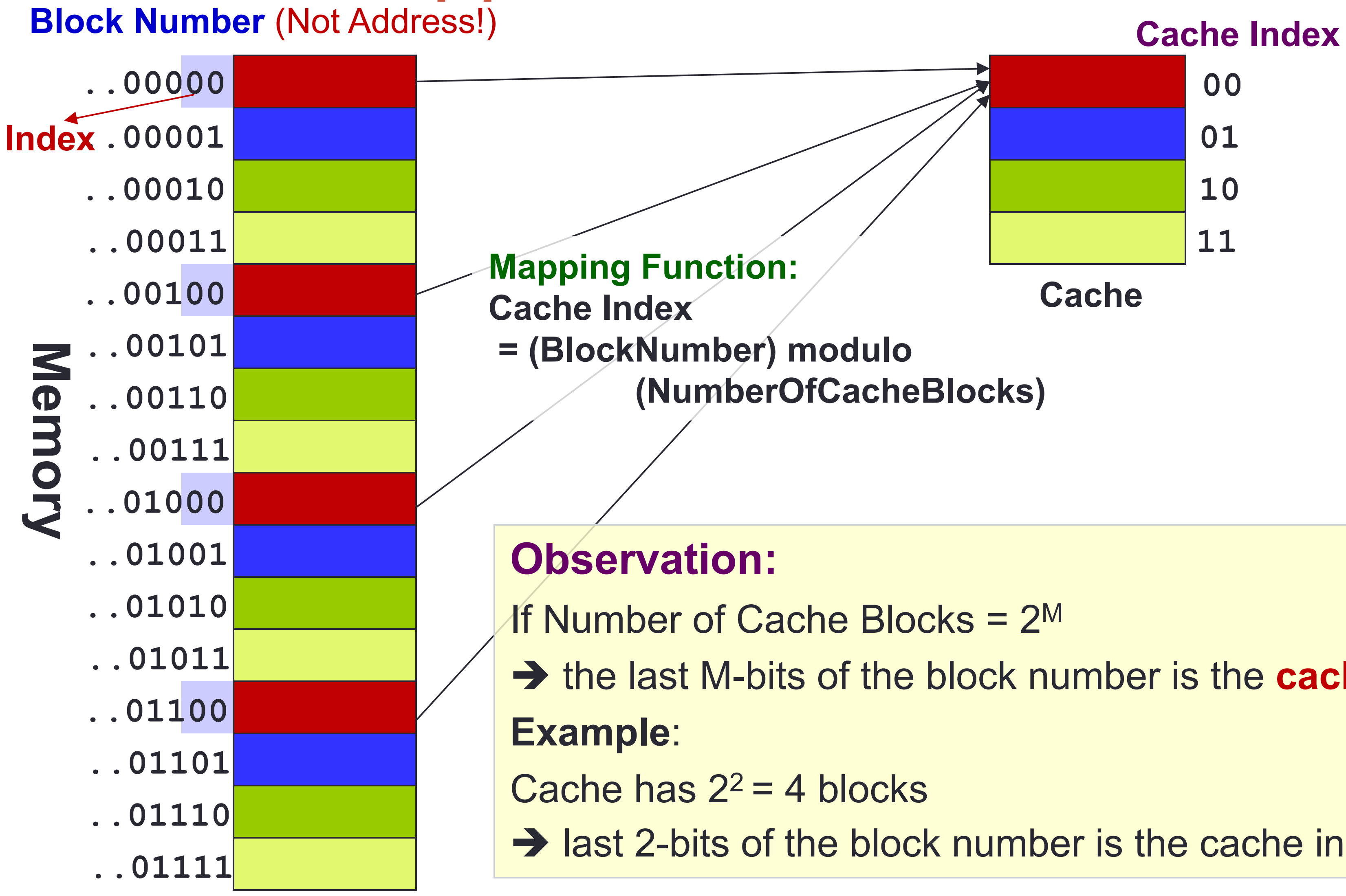
1. 2^N-byte blocks are aligned at 2^N-byte boundaries
2. The addresses of words within a 2^N-byte block have identical (32-N) most significant bits (MSB).
3. Bits [31:N] ➔ the **block number**
4. Bits [N-1:0] ➔ the **byte offset** within a block



Direct Mapping Analogy

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

Direct Mapped Cache: **Cache Index**



Mapping Function:
Cache Index
= (BlockNumber) modulo (NumberOfCacheBlocks)

Observation:

If Number of Cache Blocks = 2^M

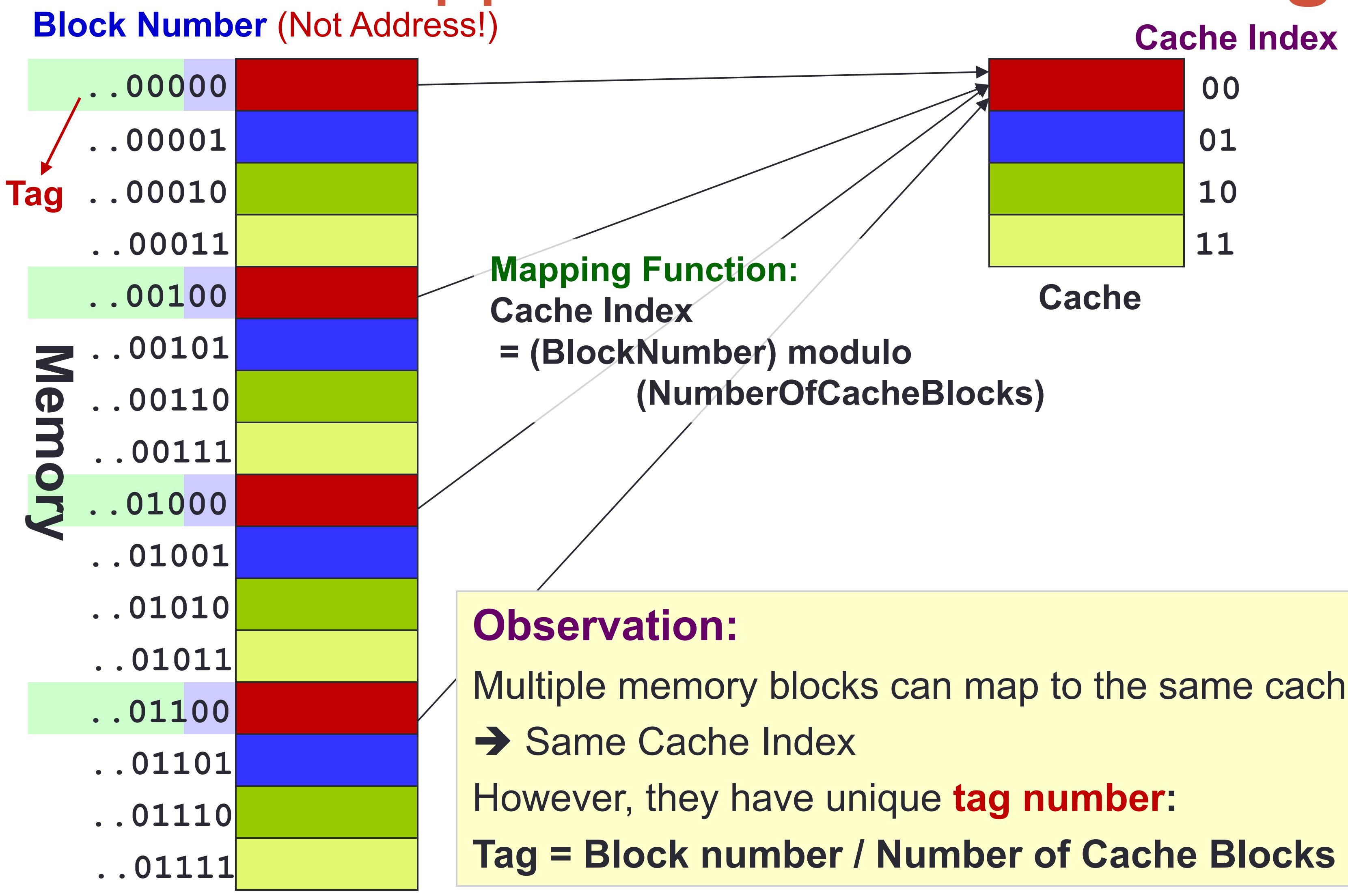
➔ the last M-bits of the block number is the **cache index**

Example:

Cache has $2^2 = 4$ blocks

➔ last 2-bits of the block number is the cache index.

Direct Mapped Cache: **Cache Tag**



Mapping Function:
Cache Index
= (BlockNumber) modulo (NumberOfCacheBlocks)

Observation:

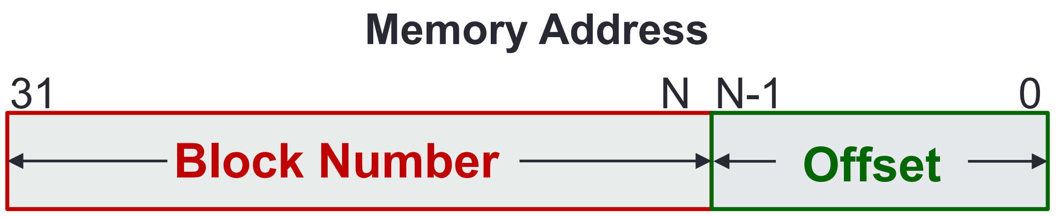
Multiple memory blocks can map to the same cache block

➔ Same Cache Index

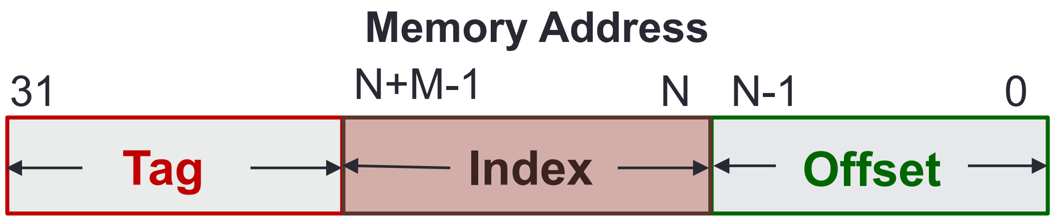
However, they have unique **tag number**:

Tag = Block number / Number of Cache Blocks

Direct Mapped Cache: **Mapping**



Cache Block size = 2^N bytes



Cache Block size = 2^N bytes

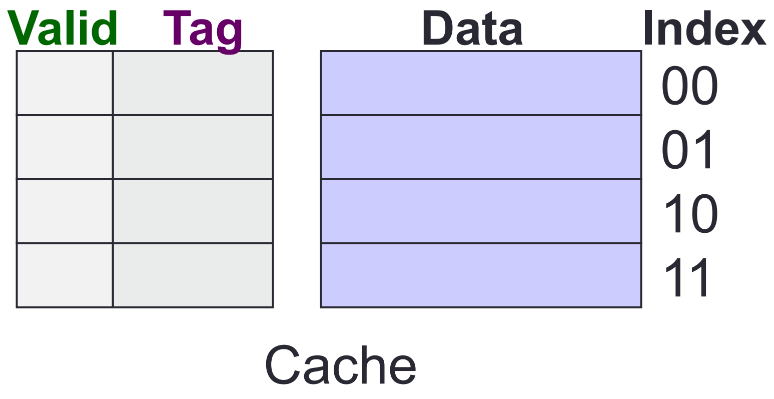
Number of cache blocks = 2^M

Offset = **N bits**

Index = **M bits**

Tag = **32 – (N + M) bits**

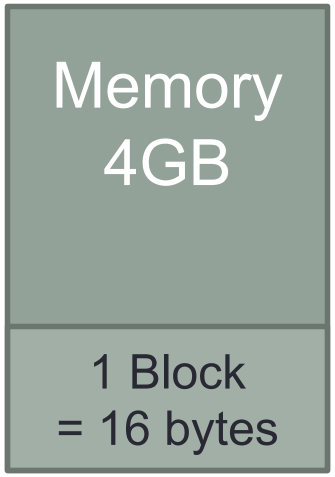
Cache Structure



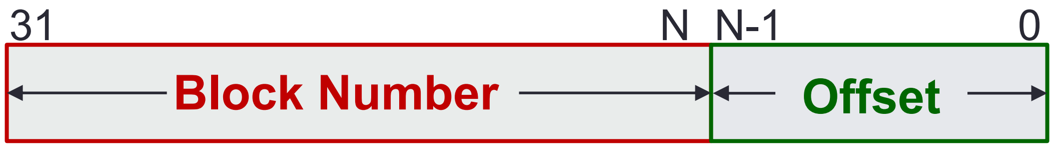
Along with a data block (line), cache contains
1. **Tag** of the memory block
2. **Valid bit** indicating whether the cache line contains valid data

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

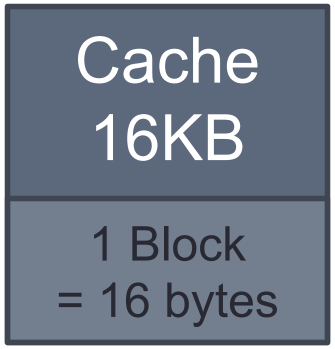
Cache Mapping: **Example**

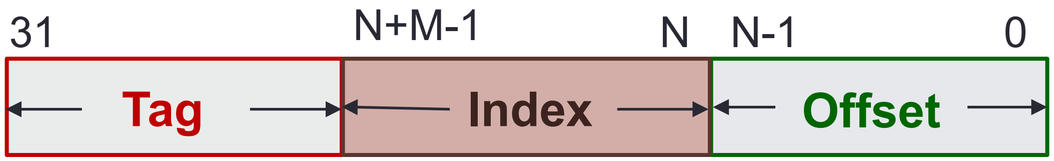


Memory Address



Offset, N = **4 bits**
Block Number = 32 – 4 = **28 bits**
Check: Number of Blocks = $\mathbf{2^{28}}$

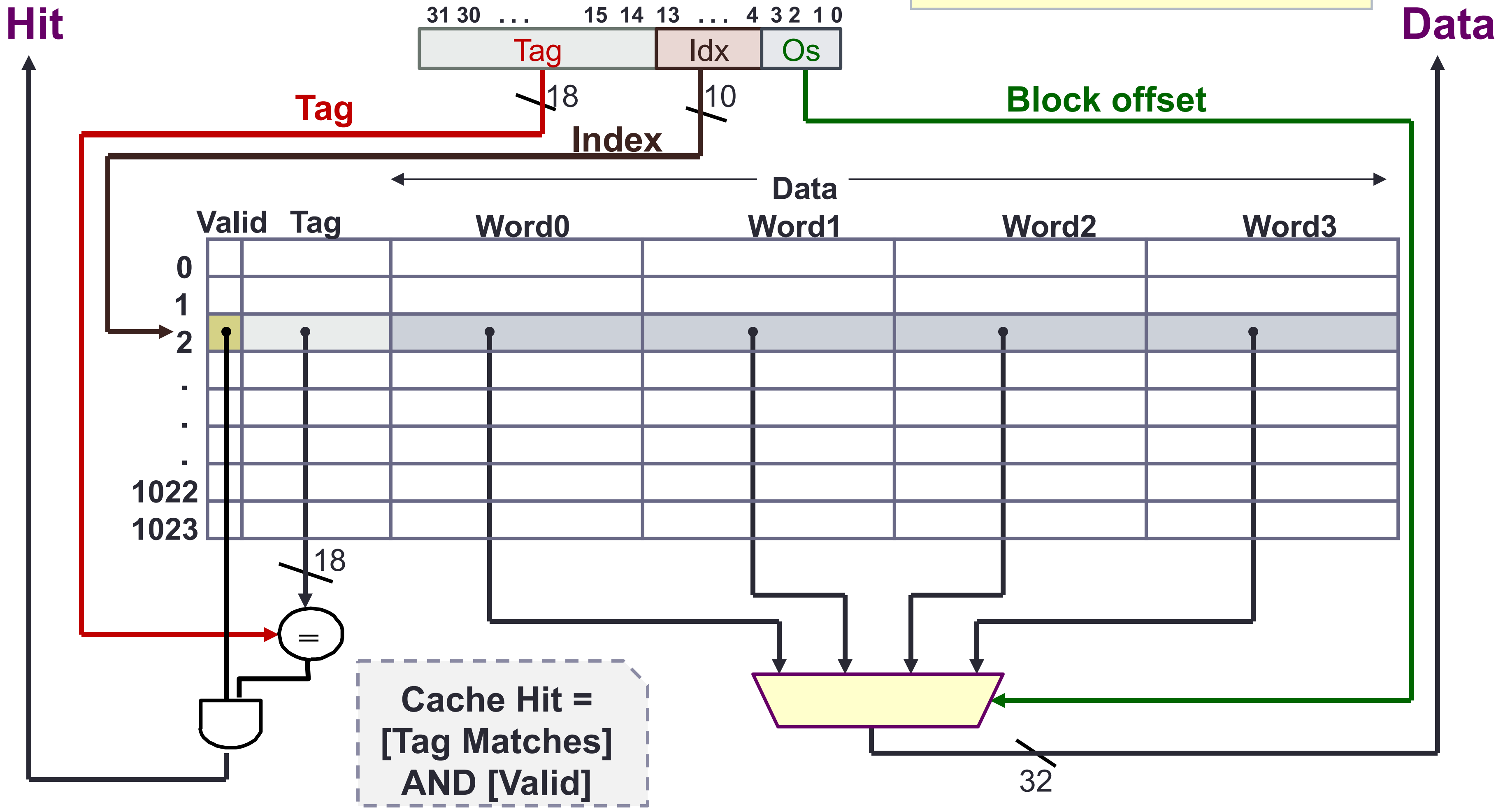




Number of Cache Blocks
= 16KB / 16bytes = 1024 = $\mathbf{2^{10}}$
Cache Index, M = **10bits**
Cache Tag = 32 – 10 – 4 = **18 bits**

Cache Circuitry: Example

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



Cache Hit =
[Tag Matches]
AND [Valid]

MEMORY LOAD INSTRUCTION

Trace and learn…..

Accessing 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
000000000000000000	0000000001	0100
000000000000000000	0000000001	1100
000000000000000000	0000000011	0100
000000000000000010	0000000001	1000
000000000000000000	0000000001	0000

Accessing Data: **Initial State**

- Initially cache is empty
  - ➔ All ***valid*** bits are 0

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

Accessing Data: **LOAD #1-1**

- Load from

Tag	Index	Offset
000000000000000000	0000000001	0100

Step 1. Check Cache Block at index 1

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

Accessing Data: **LOAD #1-2**

- Load from

Tag	**Index**	**Offset**
000000000000000000	0000000001	0100

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

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

Accessing Data: **LOAD #1-3**

- Load from

Tag	Index	Offset
0000000000000000000	0000000001	0100

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

			Data			
			Word0	**Word1**	**Word2**	**Word3**
Index	**Valid**	**Tag**	Byte 0-3	Byte 4-7	Byte 8-11	Byte 12-15
0	**0**					
1	**1**	**0**	**A**	**B**	**C**	**D**
2	**0**					
3	**0**					
4	**0**					
5	**0**					
		…	…	…	…	…
1022	**0**					
1023	**0**					

Accessing Data: **LOAD #1-4**

- Load from

Tag	Index	Offset
000000000000000000	0000000001	0100

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

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

Accessing Data: **LOAD #2-1**

- Load from

Tag	Index	Offset
0000000000000000000	0000000001	1100

Step 1. Check Cache Block at index 1

			Word0	Word1	Word2	Word3
Index	**Valid**	**Tag**	Byte 0-3	Byte 4-7	Byte 8-11	Byte 12-15
0	**0**					
1	**1**	**0**	**A**	**B**	**C**	**D**
2	**0**					
3	**0**					
4	**0**					
5	**0**					
…	…	…	…	…	…	…
1022	**0**					
1023	**0**					

Accessing Data: **LOAD #2-2**

- Load from

Tag	Index	Offset
0000000000000000000	0000000001	1100

Step 2. [Cache Block is valid] AND [Tag matches] ➔ Cache hit!

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

Accessing Data: **LOAD #2-3**

- Load from

Tag	Index	Offset
000000000000000000	0000000001	1100

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

			Data			
			Word0	**Word1**	**Word2**	**Word3**
Index	**Valid**	**Tag**	Byte 0-3	Byte 4-7	Byte 8-11	Byte 12-15
0	**0**					
1	**1**	**0**	**A**	**B**	**C**	**D**
2	**0**					
3	**0**					
4	**0**					
5	**0**					
		…	…	…	…	…
1022	**0**					
1023	**0**					

Accessing Data: **LOAD #3-1**

- Load from

Tag	Index	Offset
000000000000000000	0000000011	0100

Step 1. Check Cache Block at index 3

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

Accessing Data: **LOAD #3-2**

- Load from

Tag	Index	Offset
000000000000000000	0000000011	0100

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

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

Accessing Data: **LOAD #3-3**

- Load from

Tag	Index	Offset
0000000000000000000	0000000011	0100

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

Index	Valid	Tag	Data: **Word0** Byte 0-3	**Word1** Byte 4-7	**Word2** Byte 8-11	**Word3** Byte 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**					

Accessing Data: **LOAD #3-4**

- Load from

Tag	Index	Offset
000000000000000000	0000000011	0100

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

			Data			
			Word0	**Word1**	**Word2**	**Word3**
Index	**Valid**	**Tag**	Byte 0-3	Byte 4-7	Byte 8-11	Byte 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**					

Accessing Data: **LOAD #4-1**

- Load from

Tag	Index	Offset
000000000000000010	0000000001	0100

Step 1. Check Cache Block at index 1

Index	Valid	Tag	Data **Word0** Byte 0-3	 **Word1** Byte 4-7	 **Word2** Byte 8-11	 **Word3** Byte 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					

Accessing Data: **LOAD #4-2**

- Load from

Tag	**Index**	**Offset**
000000000000000010	0000000001	0100

Step 2. Cache block is **Valid** but **Tag mism[illegible]ches** **[Cold Miss]**

			Data			
Index	**Valid**	**Tag**	**Word0** Byte 0-3	**Word1** Byte 4-7	**Word2** Byte 8-11	**Word3** Byte 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**					

Accessing Data: **LOAD #4-3**

- Load from

Tag	**Index**	**Offset**
0000000000000000010	0000000001	0100

Step 3. Replace block 1 with new data; Set ag

Data

Index	Valid	Tag	**Word0** Byte 0-3	**Word1** Byte 4-7	**Word2** Byte 8-11	**Word3** Byte 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**					

Accessing Data: **LOAD #4-4**

- Load from

Tag	Index	Offset
0000000000000000010	0000000001	0100

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

Index	Valid	Tag	Word0 Byte 0-3	Word1 Byte 4-7	Word2 Byte 8-11	Word3 Byte 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					

Accessing Data: **LOAD #5-1**

- Load from

Tag	Index	Offset
0000000000000000000	0000000001	0000

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

Index	Valid	Tag	**Word0** Byte 0-3	**Word1** Byte 4-7	**Word2** Byte 8-11	**Word3** Byte 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**					

Accessing Data: **LOAD #5-2**

Tag	**Index**	**Offset**
0000000000000000000	0000000001	0000

- Load from

Step 2. Cache block is **Valid** but **Tag mismatches** **[Conflict Miss]**

			Data			
Index	**Valid**	**Tag**	**Word0** Byte 0-3	**Word1** Byte 4-7	**Word2** Byte 8-11	**Word3** Byte 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**					

Accessing Data: **LOAD #5-3**

- Load from

Tag	**Index**	**Offset**
0000000000000000000	0000000001	0000

Step 3. Replace block 1 with new data; Set tag

			Data			
			Word0	**Word1**	**Word2**	**Word3**
Index	**Valid**	**Tag**	Byte 0-3	Byte 4-7	Byte 8-11	Byte 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**					

Accessing Data: **LOAD #5-4**

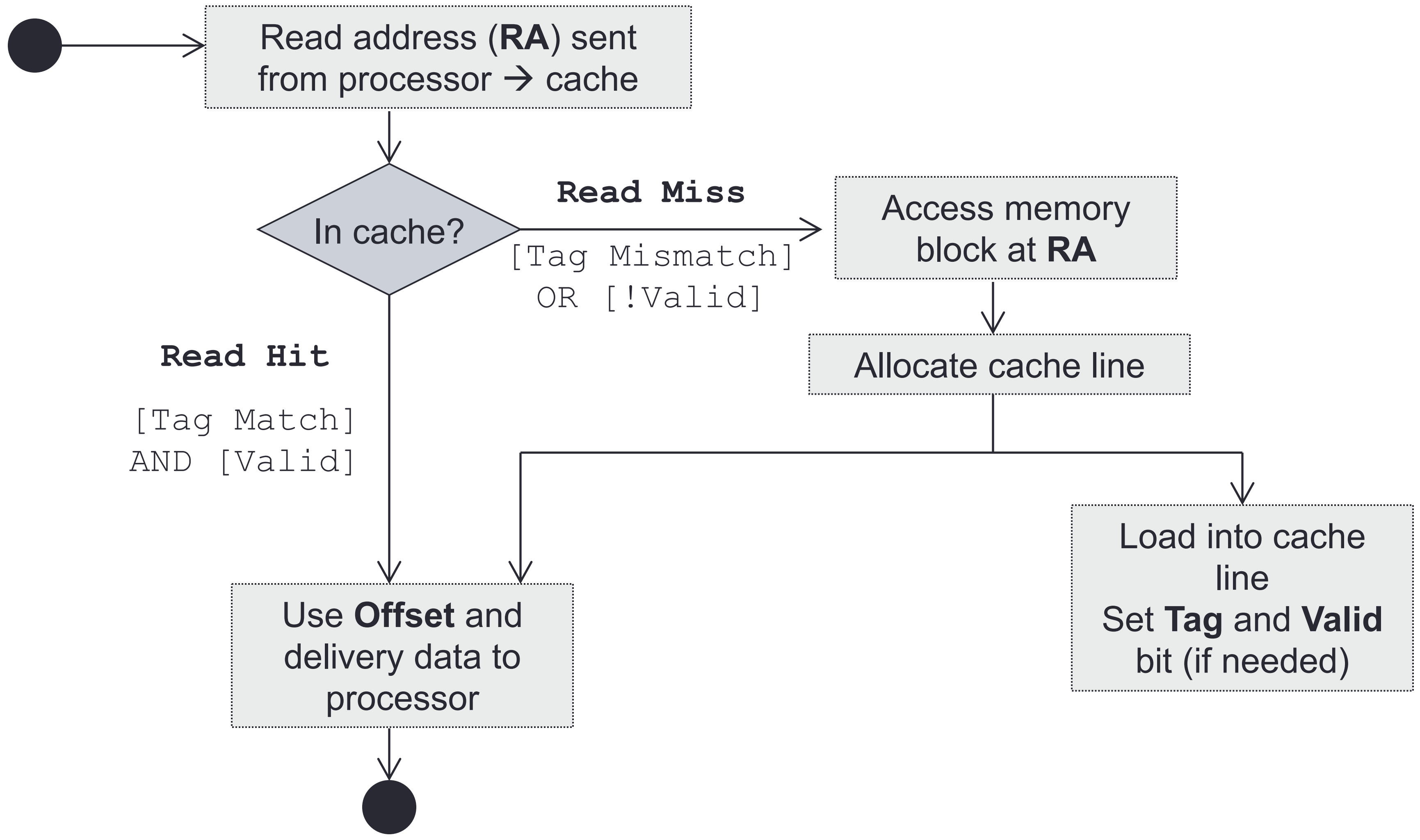
- Load from

Tag	Index	Offset
000000000000000000	0000000001	0000

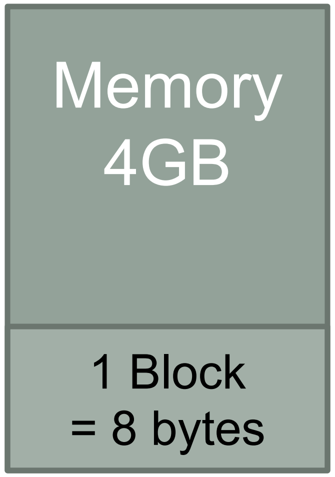
Step 4. Return **Word0** (byte offset = 0) to Register

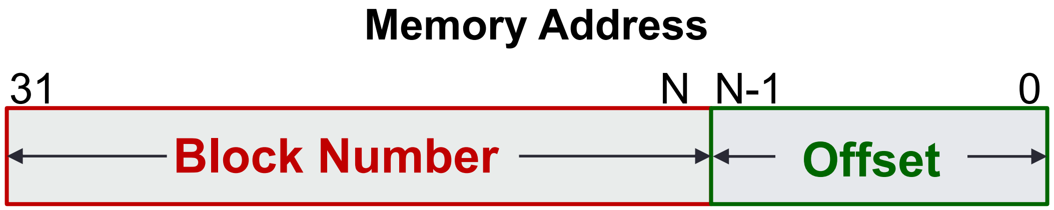
			Data			
Index	**Valid**	**Tag**	**Word0** Byte 0-3	**Word1** Byte 4-7	**Word2** Byte 8-11	**Word3** Byte 12-15
0	**0**					
1	**1**	**0**	**A**	**B**	**C**	**D**
2	**0**					
3	**1**	**0**	**I**	**J**	**K**	**L**
4	**0**					
5	**0**					
	...	...	...	...	...	... ...
1022	**0**					
1023	**0**					

Cache – Load Instruction: Summary


Read address (RA) sent from processor → cache
In cache?
Read Miss
[Tag Mismatch] OR [!Valid]
Access memory block at RA
Allocate cache line
Read Hit
[Tag Match] AND [Valid]
Load into cache line
Set Tag and Valid bit (if needed)
Use Offset and delivery data to processor

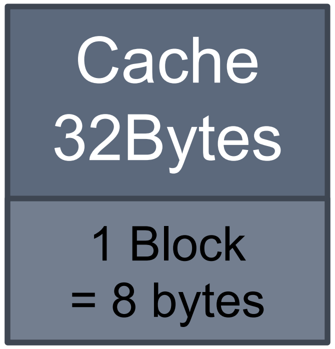
Exercise #1: Setup Information

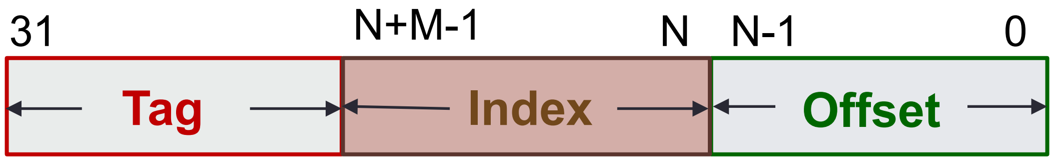




Offset, N =

Block Number =





Number of Cache Blocks =

Cache Index, M =

Cache Tag =

Exercise #2: Tracing Memory Access

- Using the given setup, trace the following memory loads:
  - Load from addresses:

    4, 0, 8, 12, 36, 0, 4

- Note that "A", "B"…. "J" represent word-size data

Memory Content

Address	Data
0	A
4	B
8	C
12	D
…	…
32	I
36	J
…	…

Exercise #2: **Load #1**

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

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

	Tag	Index	Offset
Address 4 =	000000000000000000000000000000	00	100

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

Exercise #2: **Load #2**

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

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

	Tag	Index	Offset
Address 0 =	000000000000000000000000000	00	000

Index	Valid	Tag	Word0	Word1
0	1	0000..000	A	B
1	0			
2	0			
3	0			

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

Exercise #2: **Load #3**

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

	Tag	Index	Offset
Address 8 =	000000000000000000000000000	01	000

Index	Valid	Tag	Word0	Word1
0	1	0000..000	A	B
1	1	0000..000	C	D
2	0			
3	0			

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

Exercise #2: **Load #4**

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

	Tag	Index	Offset
Address 12 =	00000000000000000000000000	01	100

Index	Valid	Tag	Word0	Word1
0	1	0000..000	A	B
1	1	0000..000	C	D
2	0			
3	0			

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

Exercise #2: **Load #5**

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

	Tag	Index	Offset
Address 36 =	0000000000000000000000000001	00	100

Index	Valid	Tag	Word0	Word1
0	1	0000..001	I	J
1	1	0000..000	C	D
2	0			
3	0			

Exercise #2: Load #6

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

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

	Tag	Index	Offset
Address 0 =	00000000000000000000000000000	00	000

Index	Valid	Tag	Word0	Word1
0	1	0000..001	I	J
1	1	0000..000	C	D
2	0			
3	0			

Exercise #2: **Load #7**

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

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

Address 4 =

Tag	Index	Offset
000000000000000000000000000	00	100

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

MEMORY STORE INSTRUCTION

A little trickier

Writing Data: **STORE #1-1**

- Store **X** to

Tag	Index	Offset
0000000000000000000	0000000001	1000

Step 1. Check Cache Block 1

			Data			
Index	**Valid**	**Tag**	**Word0** Byte 0-3	**Word1** Byte 4-7	**Word2** Byte 8-11	**Word3** Byte 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**					

Writing Data: **STORE #1-2**

Tag	Index	Offset
000000000000000000	0000000001	1000

- Store **X** to

Step 2. . [Cache Block is Valid] AND [Tag matches] ➔ Cache hit!

Index	Valid	Tag	Data: Word0 Byte 0-3	Word1 Byte 4-7	Word2 Byte 8-11	Word3 Byte 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					

Writing Data: **STORE #1-3**

- Store **X** to

Tag	Index	Offset
0000000000000000000	0000000001	1000

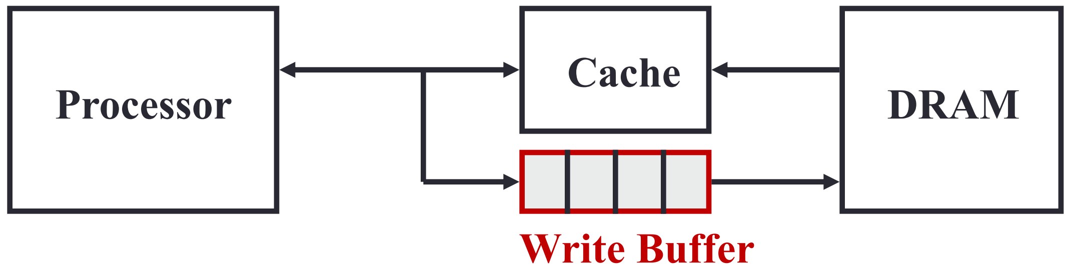
Step 3. Replace Word2 (offset = 8) with **X** **[See any problem here?]**

			Data			
			Word0	**Word1**	**Word2**	**Word3**
Index	**Valid**	**Tag**	Byte 0-3	Byte 4-7	Byte 8-11	Byte 12-15
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**					

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)

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

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

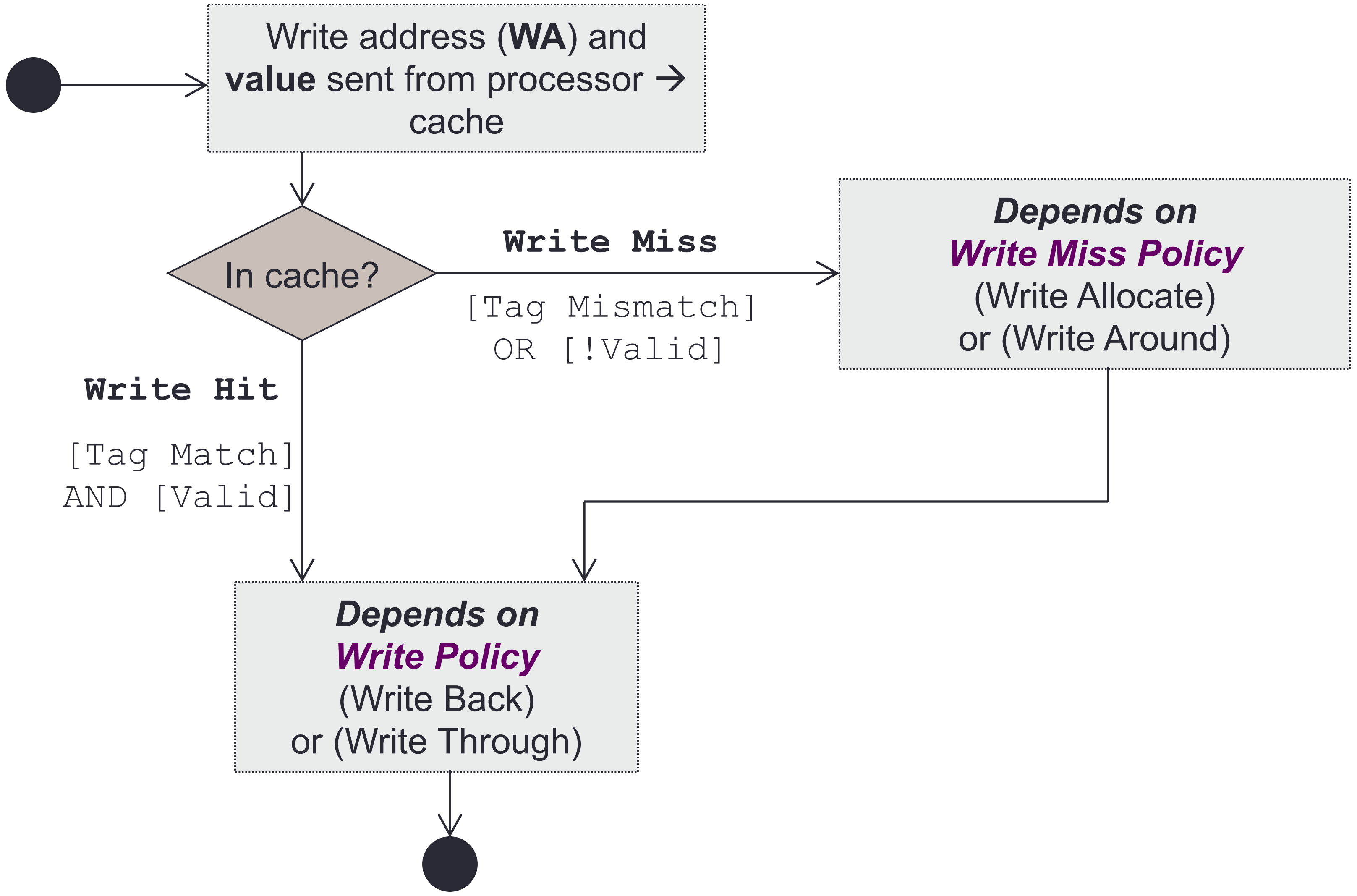
Handling Cache Misses

- 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**

Cache – Store Instructions: Summary


Write address (**WA**) and **value** sent from processor → cache
In cache?
Write Miss
[Tag Mismatch] OR [!Valid]
Depends on *Write Miss Policy* (Write Allocate) or (Write Around)
Write Hit
[Tag Match] AND [Valid]
Depends on *Write Policy* (Write Back) or (Write Through)

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 Assignment

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

Q&A