CS2100 COMPUTER ORGANISATION

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

Lecture #22

Cache

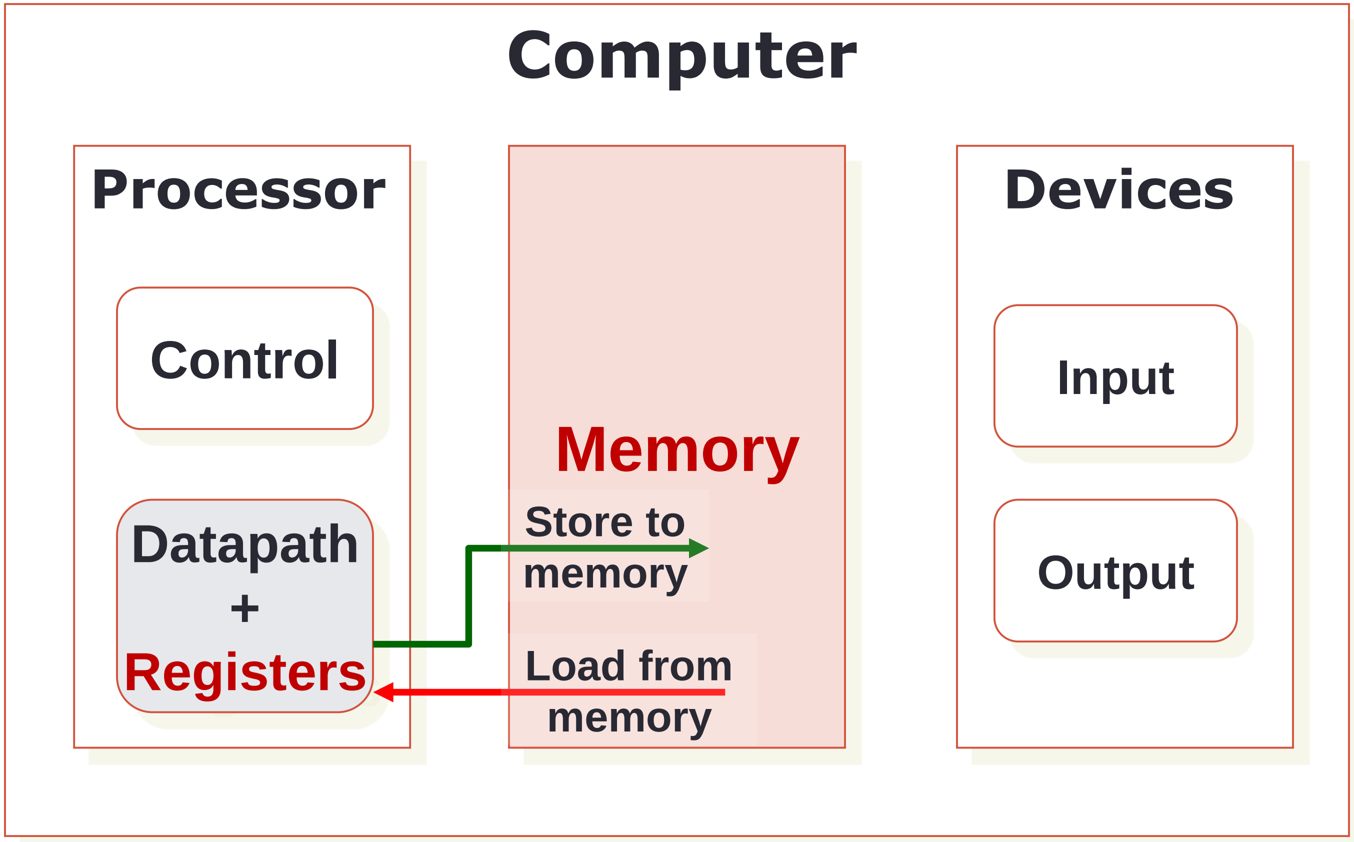
Part I: Direct Mapped Cache

NUS National University of Singapore | School of Computing

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

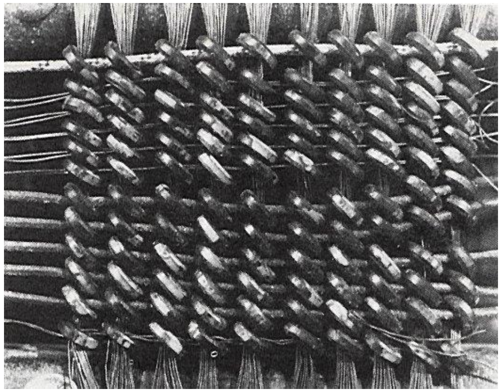


Registers are in the datapath of the processor. If operands are in memory we have to **load** them to processor (registers), operate on them, and **store** them back to memory.

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



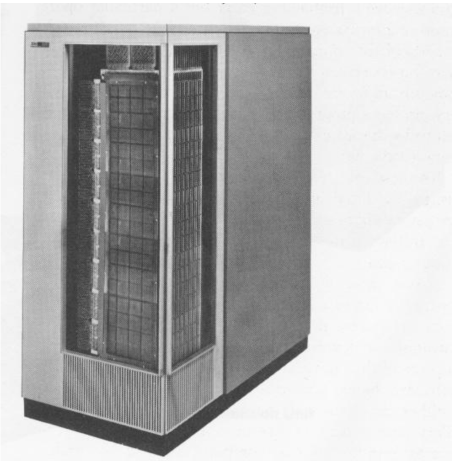
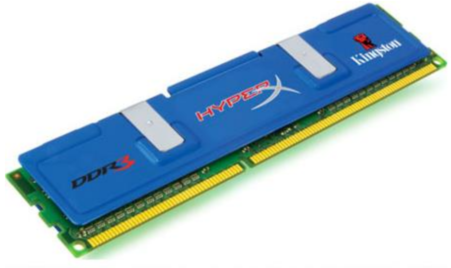


Figure 10. IBM 2361 Core Storage

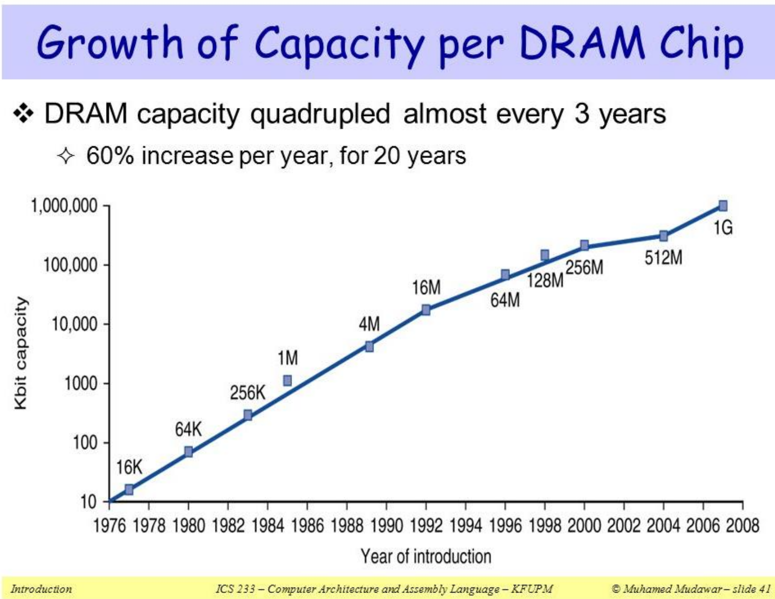
1952: IBM 2361 16KB magnetic core memory

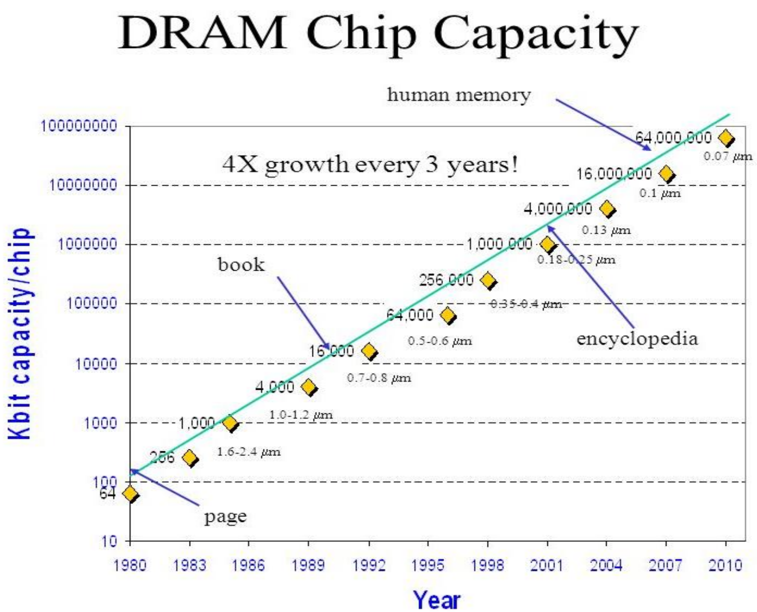
1. Memory Technology Today: **DRAM**



- **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 (released in 2014)
    - DDR5 (in Q3 2021)

1. DRAM Capacity Growth





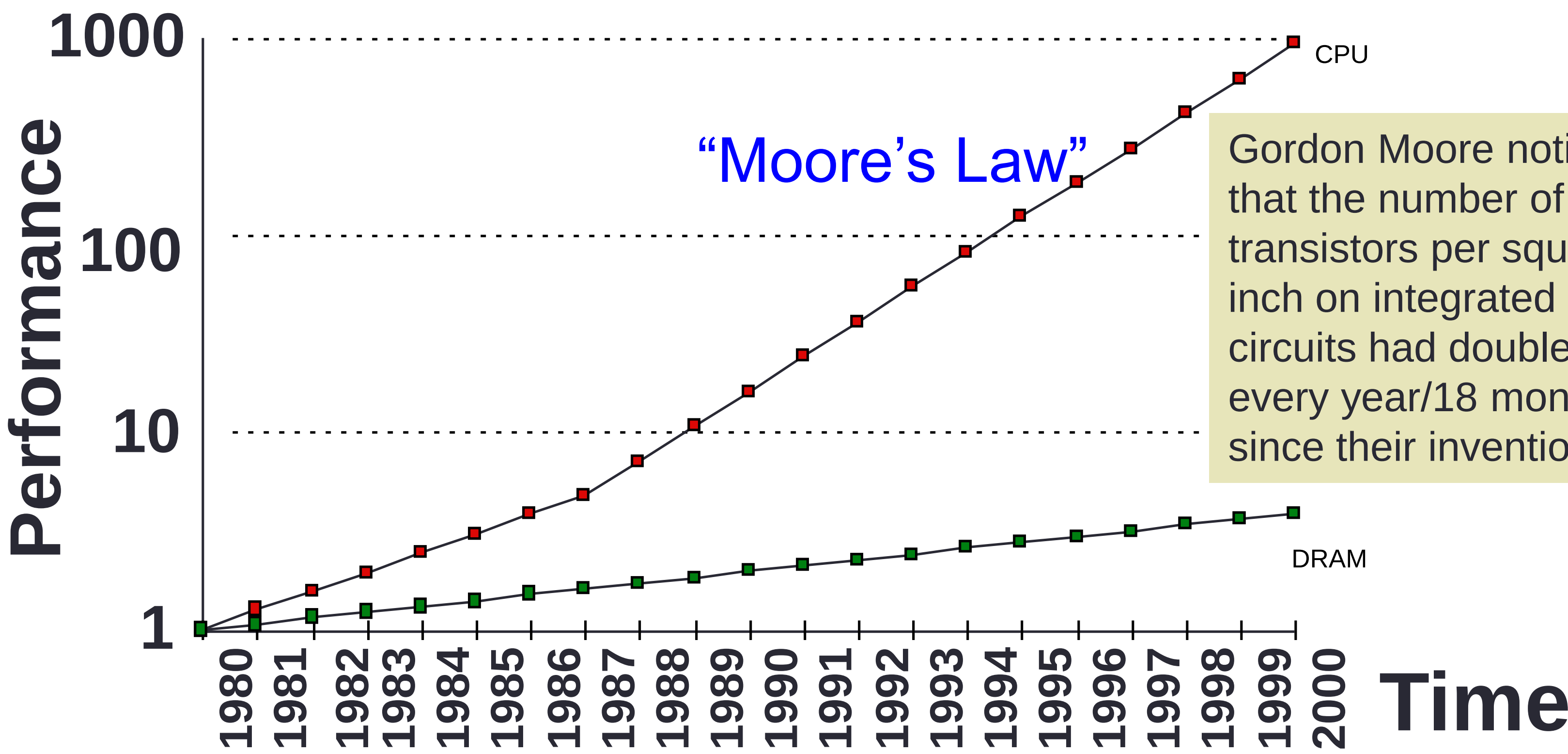
- Unprecedented growth in density, but we still have a problem

1. Processor-DRAM Performance Gap

Memory Wall:

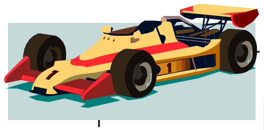
1GHz Processor ➔ 1 ns per clock cycle

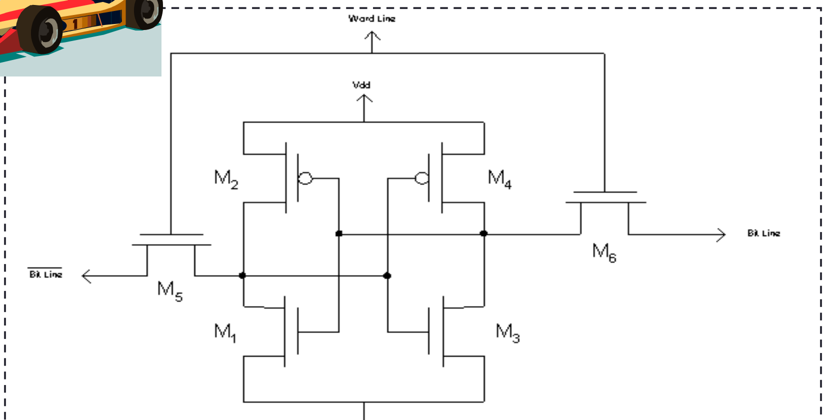
50ns for DRAM access ➔ 50 processor clock cycles per memory access!



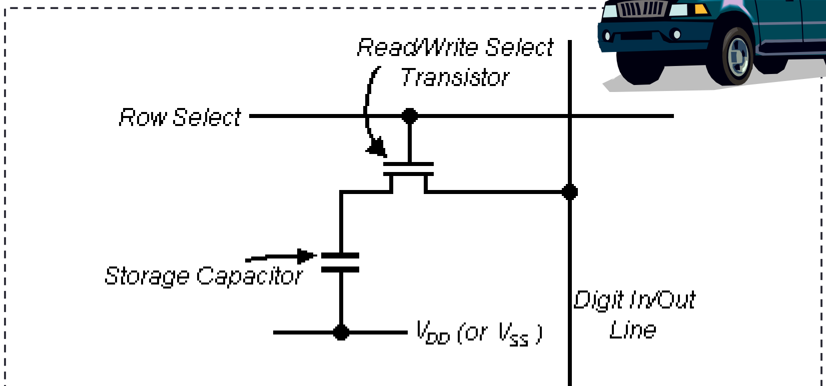
Gordon Moore noticed that the number of transistors per square inch on integrated circuits had doubled every year/18 months since their invention.

1. Faster Memory Technology: **SRAM**





A SRAM Cell



A DRAM Cell

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

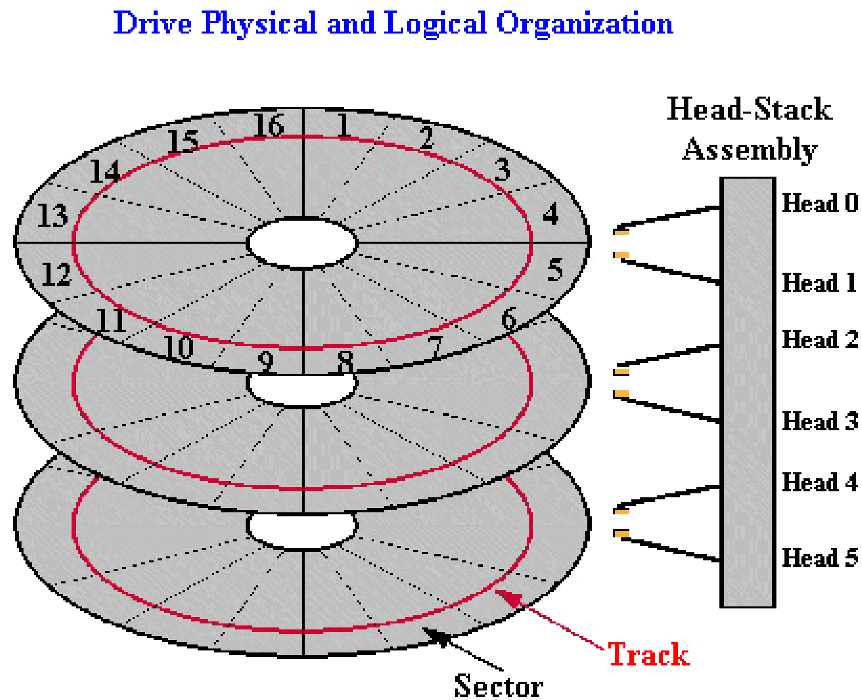
→ **High density**

Slow access latency of 50-70ns

Less costly

Used in main memory

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



Typical high-end hard disk:

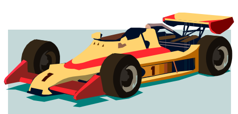
Average Latency: 4 - 10 ms
Capacity: 500-2000GB

1. Quality vs Quantity

Processor
Control
Datapath
Registers

Memory
(DRAM)

Devices
Input
Output
(Harddisk)



	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**

1. Best of Both Worlds

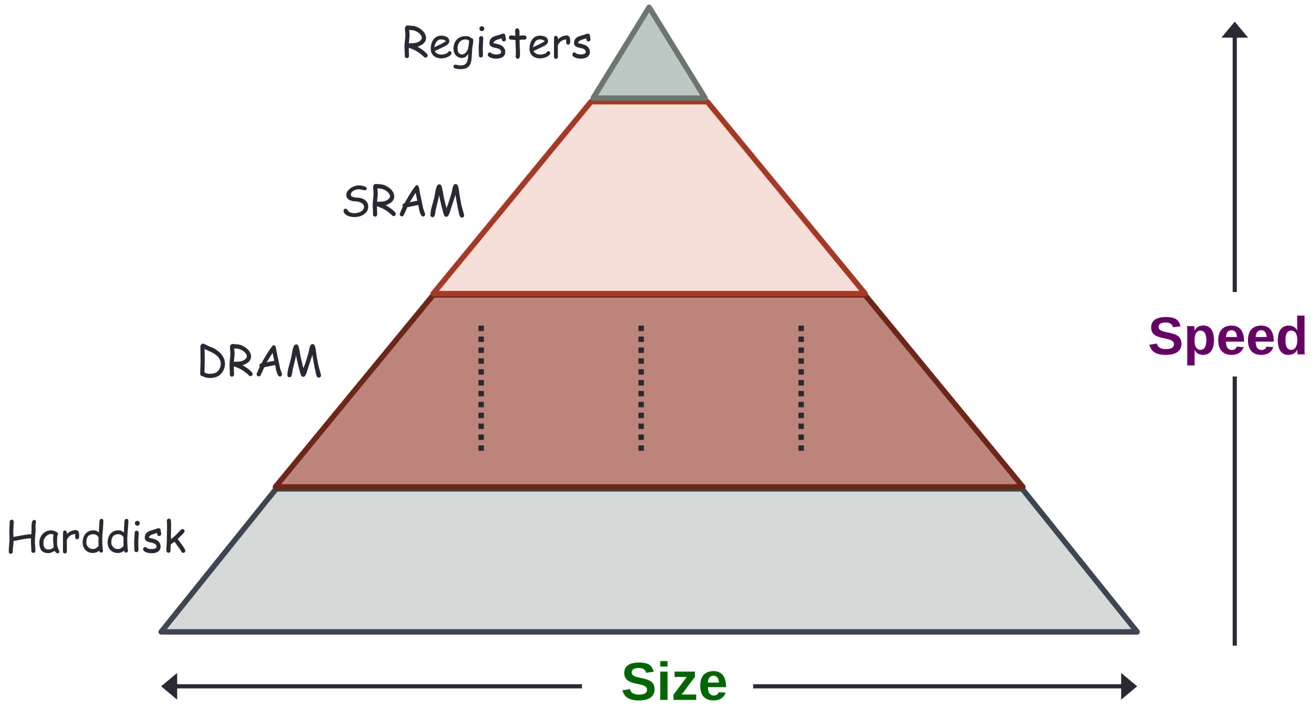
- What we want:
  - A **BIG** and **FAST** memory
  - Memory system should perform like 1GB of SRAM (1ns access time) but cost like 1GB of slow memory

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

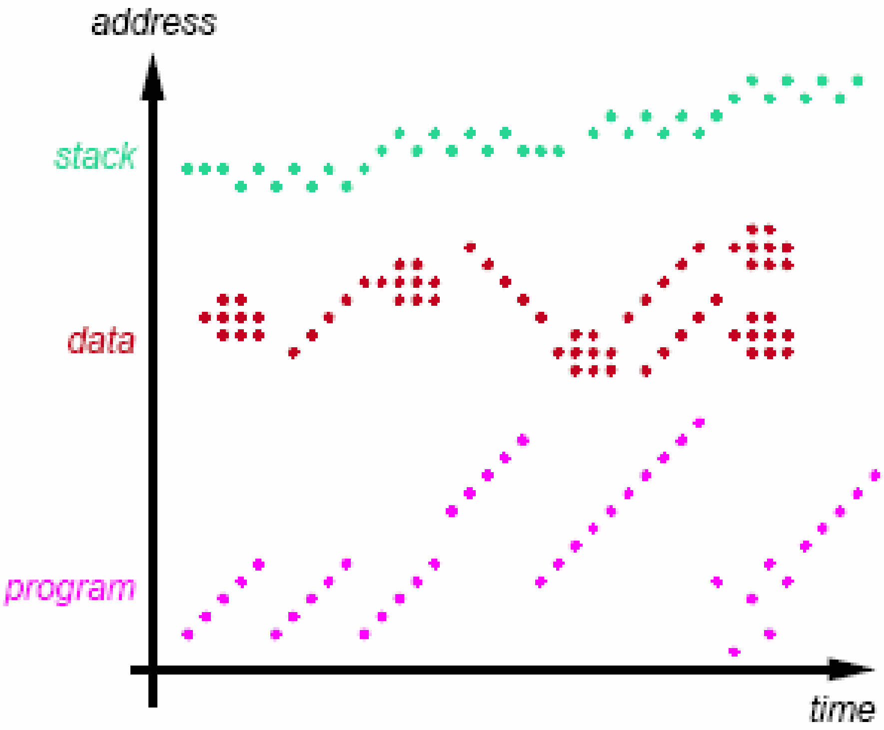
- Keep the frequently and recently used data in **smaller but faster** memory
- Refer to bigger and slower memory:
  - Only when you cannot find data/instruction in the faster memory
- Why does it work?

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

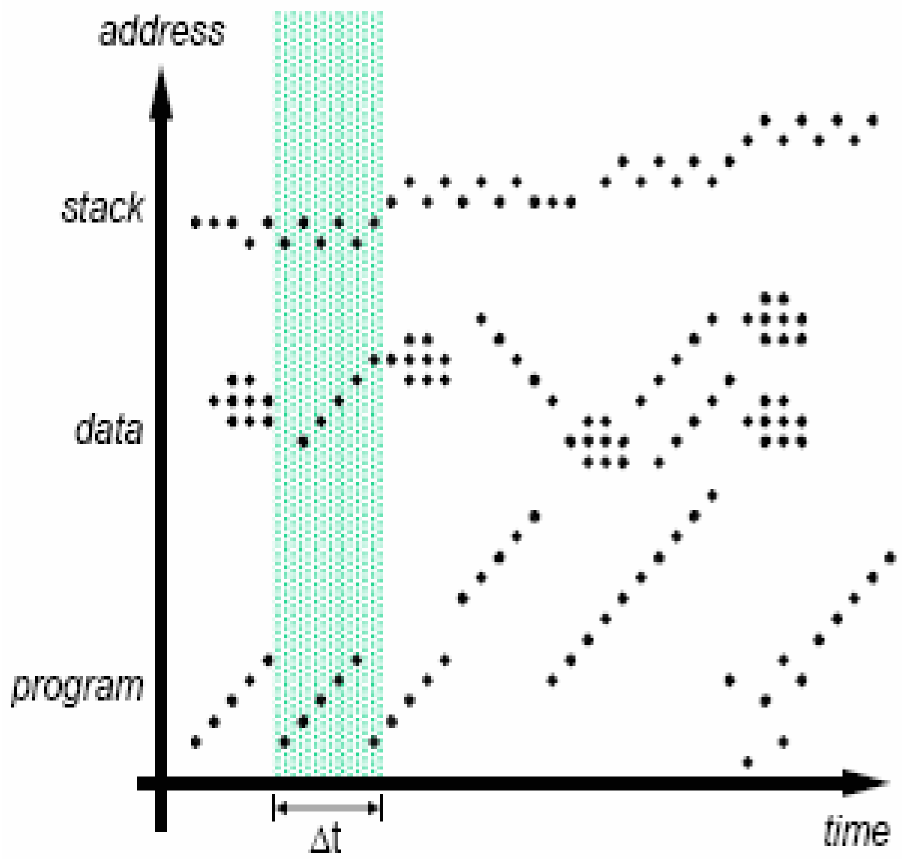
- **Temporal locality**
  - If an item is referenced, it will tend to be referenced again soon
- **Spatial locality**
  - If an item is referenced, nearby items will tend to be referenced soon
- Different locality for
  - Instructions
  - Data



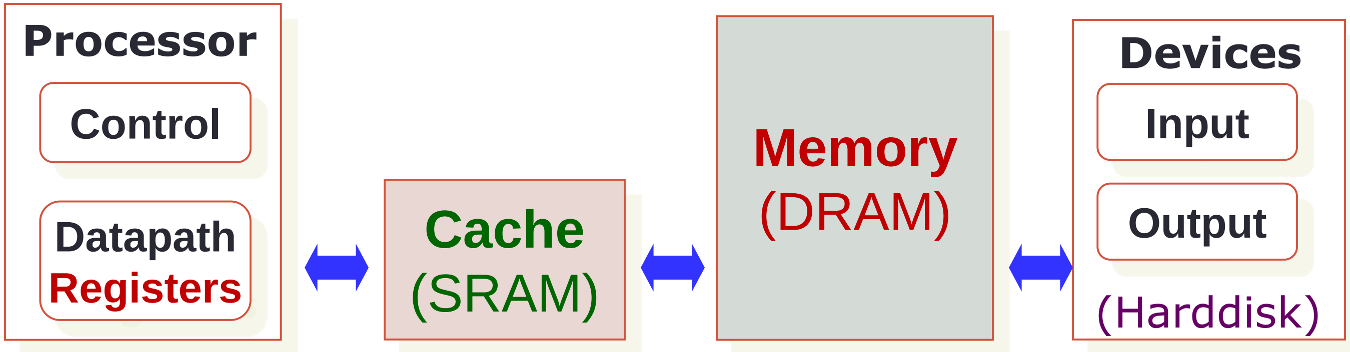
2.1 Working Set: Definition

- **Set of locations accessed during 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**

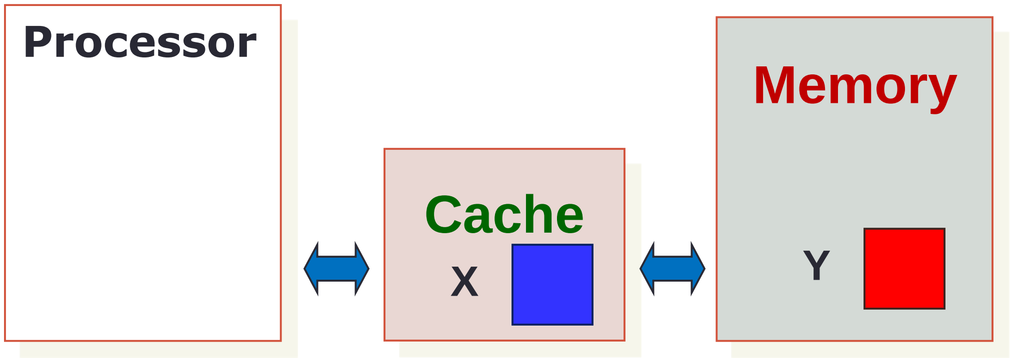


2.2 Two Aspects of Memory Access



- 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 – Hit rate
  - Miss penalty: Time to replace cache block + hit time
- Hit time < Miss penalty

2.2 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**?

Let h be the desired hit rate.

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

$\quad = 0.8h + 10.8 - 10.8h$

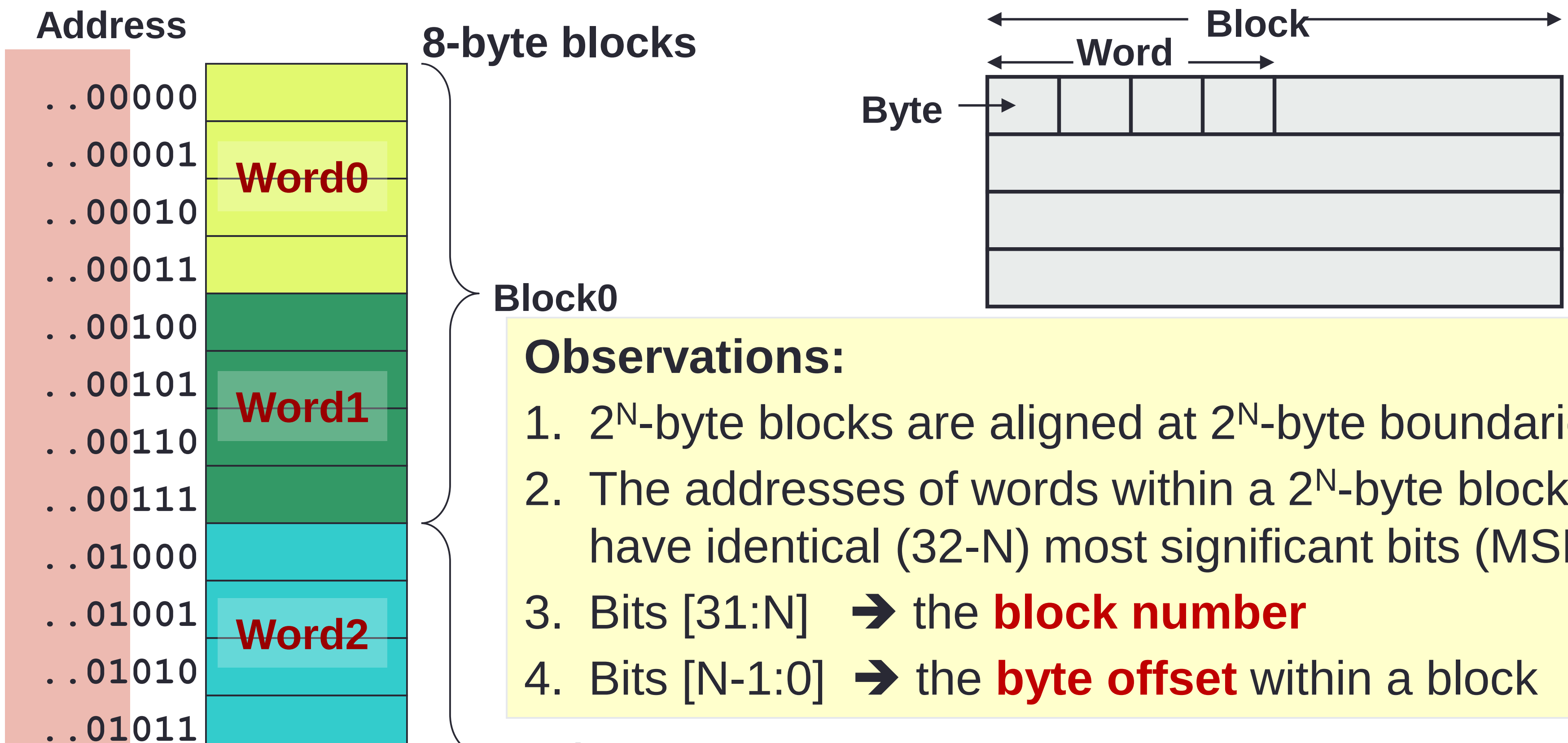
$10h = 9.8 \rightarrow h = 0.98$

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

3. Memory to Cache Mapping (1/2)

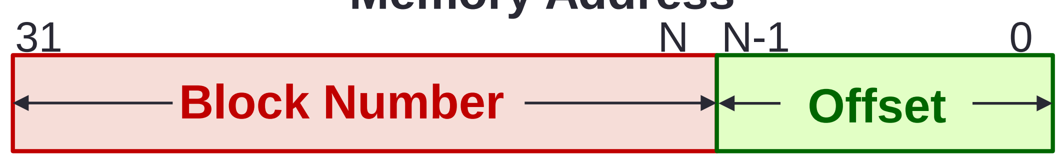
- **Cache Block/Line:**
  - Unit of transfer between memory and cache
- Block size is typically one or more words
  - e.g.: 16-byte block ≅ 4-word block
  - 32-byte block ≅ 8-word block
- Why is the block size bigger than word size?

3. Memory to Cache Mapping (2/2)



Observations:

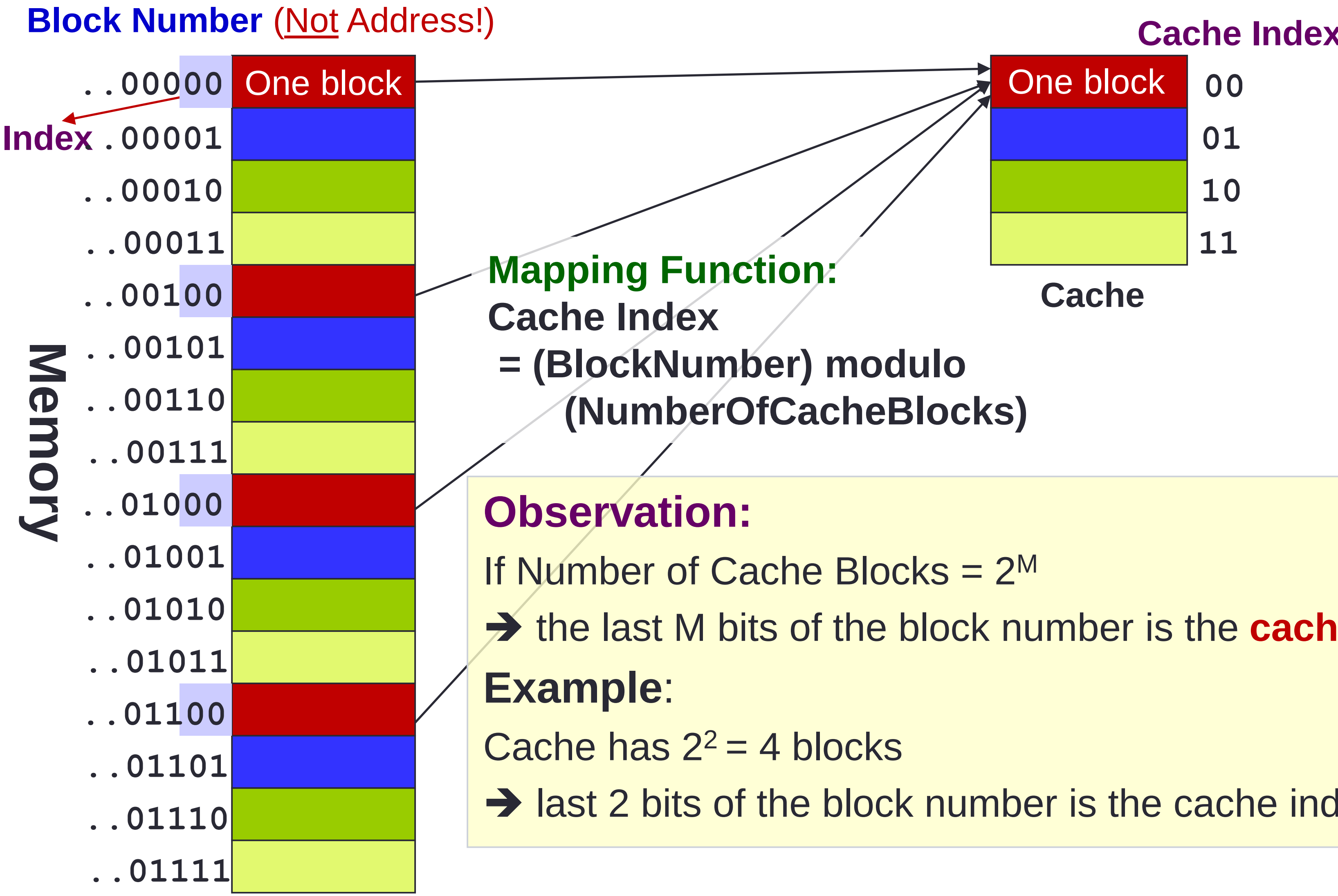
1. 2^N-byte blocks are aligned at 2^N-byte boundaries
2. The addresses of words within a 2^N-byte block have identical (32-N) most significant bits (MSB).
3. Bits [31:N] ➔ the **block number**
4. Bits [N-1:0] ➔ the **byte offset** within a block



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**



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

Observation:

If Number of Cache Blocks = 2^M

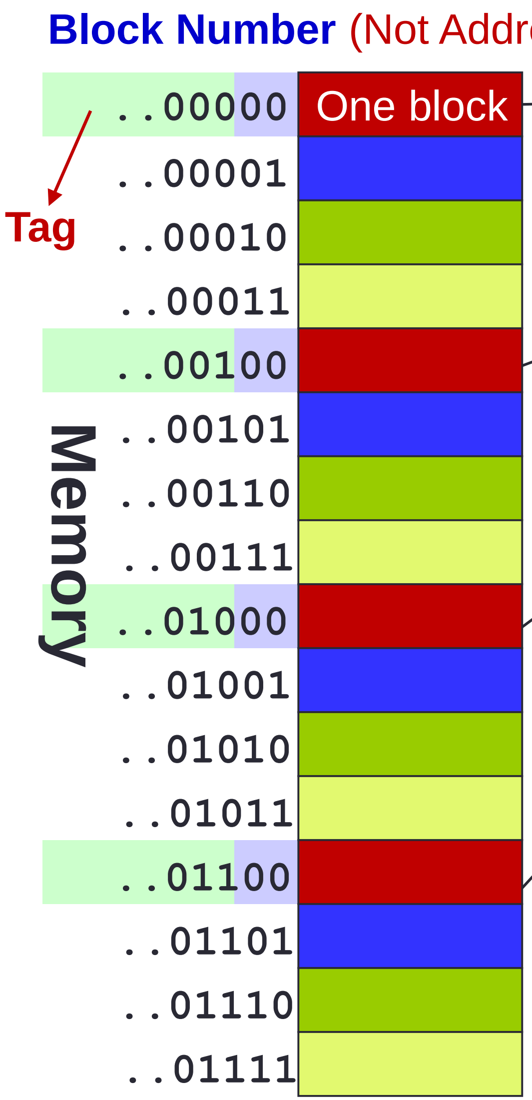
➔ the last M bits of the block number is the **cache index**

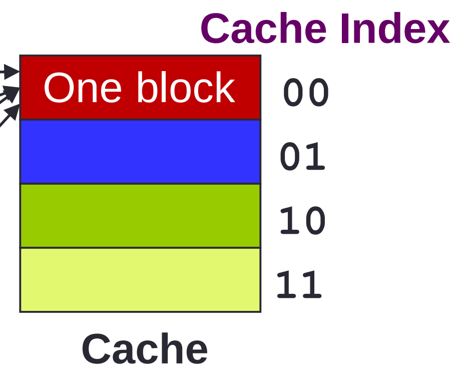
Example:

Cache has 2^2 = 4 blocks

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

4. 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**:

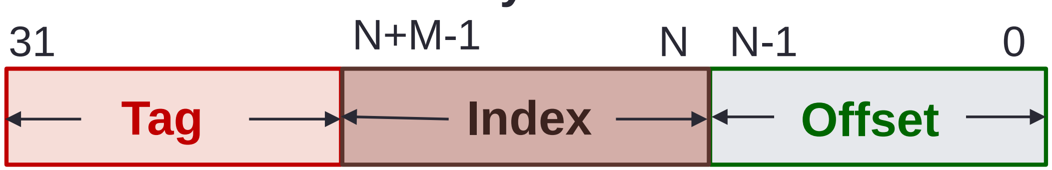
Tag = Block number / Number of Cache Blocks

4. Direct Mapped Cache: **Mapping**

Memory Address

Cache Block size = 2^N bytes

Memory Address



Cache Block size = 2^N bytes

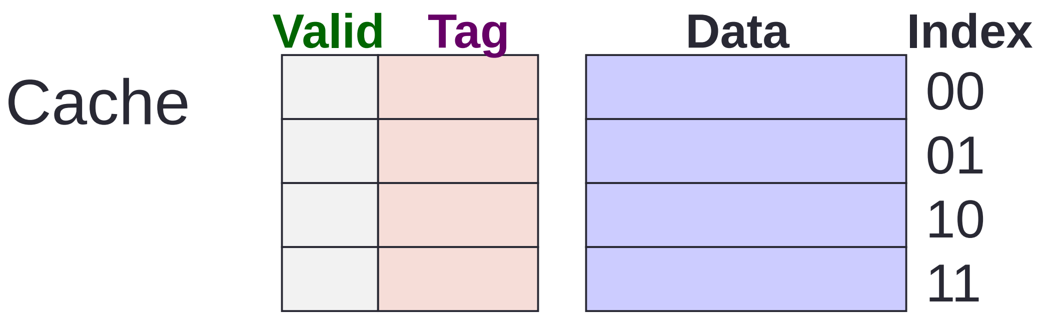
Number of cache blocks = 2^M

Offset = **N bits**

Index = **M bits**

Tag = **32 – (N + M) bits**

4. Direct Mapped Cache: **Cache Structure**



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

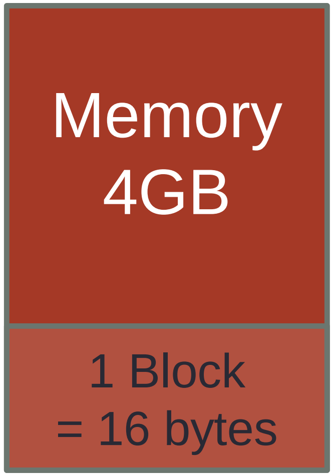
1. **Tag** of the memory block
2. **Valid bit** indicating whether the cache line contains valid data

When is there a cache hit?

(Valid[index] = TRUE) **AND**

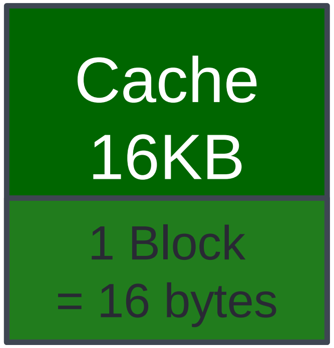
(Tag[index] = Tag[memory address])

4. Cache Mapping: **Example**



Memory Address

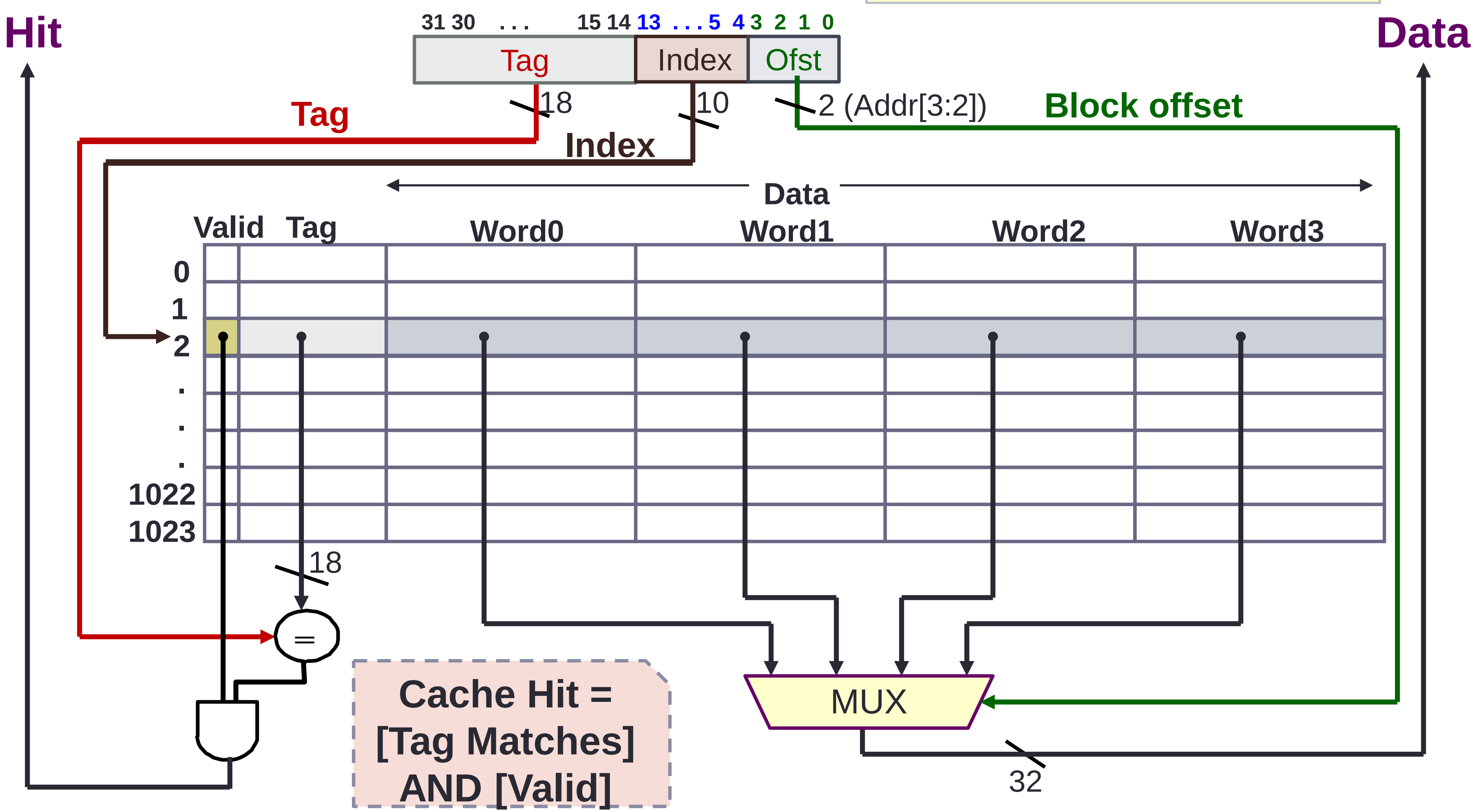
Offset, N = 4 bits
Block Number = 32 – 4 = **28 bits**
Check: Number of Blocks = $\mathbf{2^{28}}$



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

4. Cache Circuitry: **Example**

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



Cache Hit = [Tag Matches] AND [Valid]

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

5. Reading Data: **Initial State**

- Intially cache is empty
  - ➔ All ***valid*** bits are zeroes (false)

Index	Valid	Tag	Data: **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 #1-1**

	Tag	Index	Offset
▪ Load from	000000000000000000	0000000001	0100

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]**

Index	Valid	Tag	Data: 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 #1-3**

	Tag	Index	Offset
▪ Load from	000000000000000000	0000000001	0100

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

			Data			
			Word0	**Word1**	**Word2**	**Word3**
Index	**Valid**	**Tag**	Bytes 0-3	Bytes 4-7	Bytes 8-11	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	000000000000000000	0000000001	0100

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

Index	Valid	Tag	Data: **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 #2-1**

	Tag	Index	Offset
▪ Load from	0000000000000000000	0000000001	1100

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

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

	Tag	Index	Offset
▪ Load from	000000000000000000	0000000001	1100

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

Index	Valid	Tag	Data: Word0 Bytes 0-3	Data: Word1 Bytes 4-7	Data: Word2 Bytes 8-11	Data: 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 #2-3**

	Tag	Index	Offset
▪ Load from	000000000000000000	0000000001	1100

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

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	000000000000000000	0000000011	0100

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

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**					

(Data spans Word0–Word3.)

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

	Tag	Index	Offset
▪ Load from	000000000000000000	0000000011	0100

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

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

	Tag	Index	Offset
▪ Load from	000000000000000000	0000000011	0100

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

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	000000000000000000	0000000011	0100

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

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**

- Load from

Tag	Index	Offset
000000000000000000010	0000000001	1000

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

Index	Valid	Tag	Data: 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**

	Tag	Index	Offset
▪ Load from	0000000000000000010	0000000001	1000

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

Index	Valid	Tag	Data: 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**

	Tag	Index	Offset
▪ Load from	000000000000000010	0000000001	1000

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

			Data			
			Word0	**Word1**	**Word2**	**Word3**
Index	**Valid**	**Tag**	Bytes 0-3	Bytes 4-7	Bytes 8-11	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	000000000000000010	0000000001	1000

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

Index	Valid	Tag	Data: **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	000000000000000000	0000000001	0000

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

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

	Tag	Index	Offset
▪ Load from	000000000000000000	0000000001	0000

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

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					

(Word0–Word3 columns together: ← Data →)

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

Tag	Index	Offset
0000000000000000000	0000000001	0000

- Load from 0000000000000000000 0000000001 0000

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

Index	Valid	Tag	Data: 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**

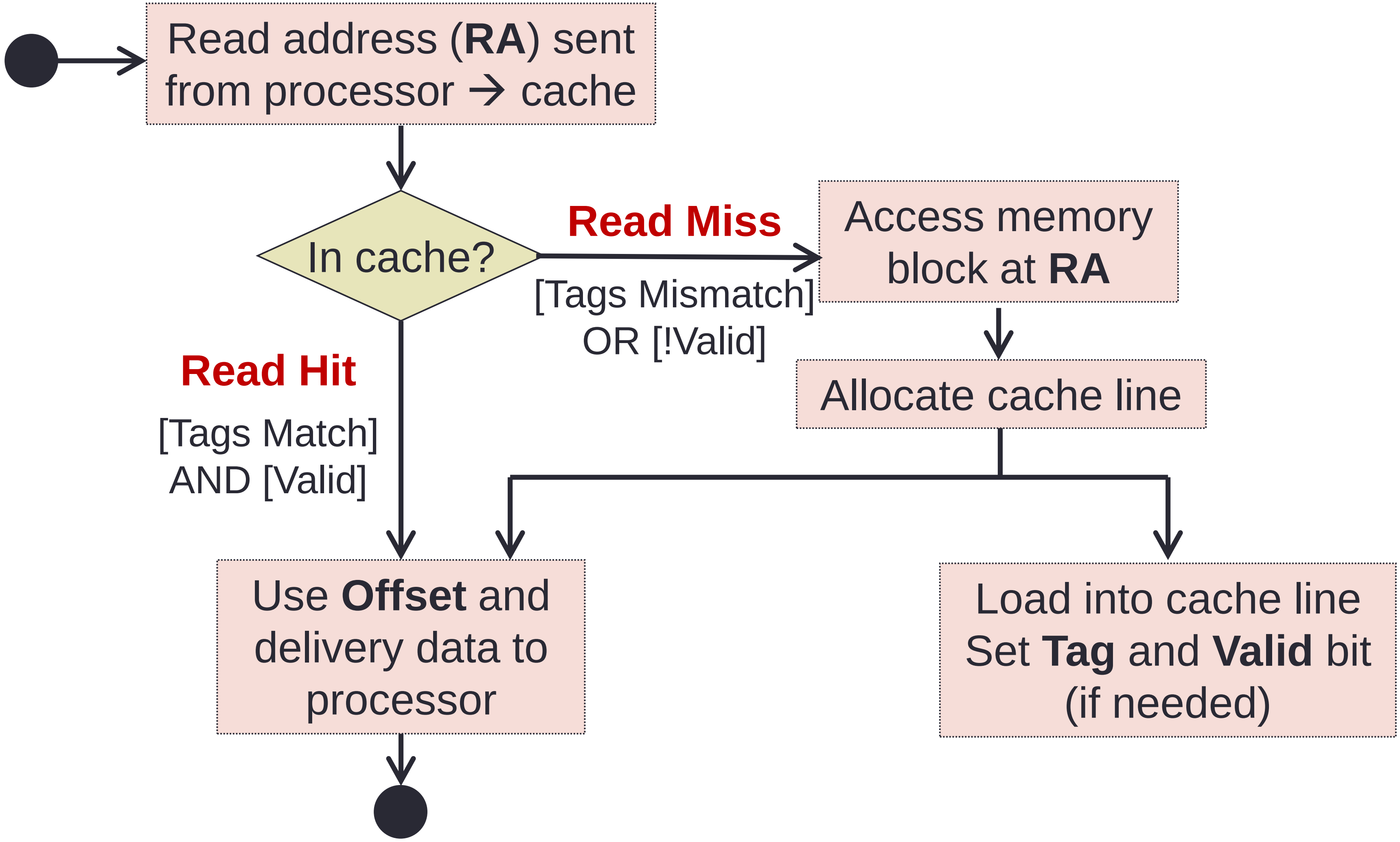
- Load from

Tag	Index	Offset
000000000000000000	0000000001	0000

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

Index	Valid	Tag	Data: 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					

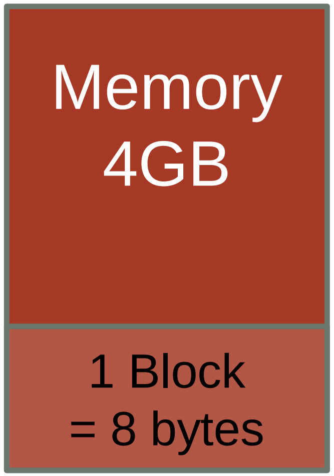
5. Reading Data: Summary

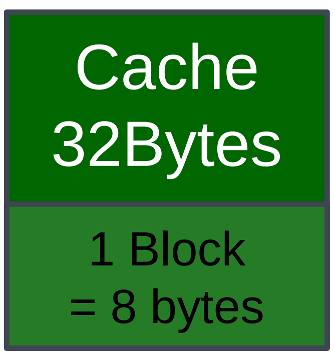


6. Types of Cache Misses

- 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

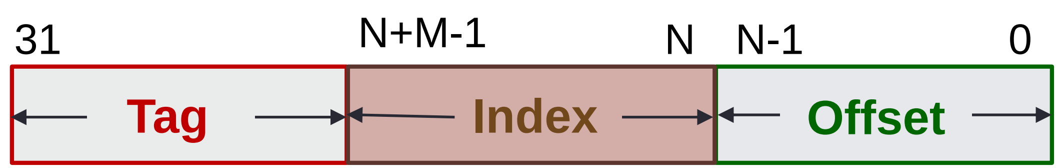




Memory Address

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
…	…

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

Exercise #2: Load #1

Miss

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

	Tag	Index	Offset
Address 4 =	000000000000000000000000000	00	100

Index	Valid	Tag	Word0	Word1
0	~~0~~ 1	0	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 #2

Miss Hit

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

	Tag	Index	Offset
Address 0 =	000000000000000000000000000	00	000

Index	Valid	Tag	Word0	Word1
0	1	0	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

Miss Hit Miss

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

	Tag	Index	Offset
Address 8 =	0000000000000000000000000000	01	000

Index	Valid	Tag	Word0	Word1
0	1	0	A	B
1	~~0~~ 1	0	C	D
2	0			
3	0			

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

Exercise #2: Load #4

Miss Hit Miss Hit

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

Address 12 =

Tag	Index	Offset
000000000000000000000000000	01	100

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

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

Exercise #2: Load #5

Miss Hit Miss Hit Miss

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

	Tag	Index	Offset
Address 36 =	0000000000000000000000000001	00	100

Index	Valid	Tag	Word0	Word1
0	1	~~0~~ 1	~~A~~ I	~~B~~ J
1	1	0	C	D
2	0			
3	0			

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

Exercise #2: Load #6

Miss Hit Miss Hit Miss Miss

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

	Tag	Index	Offset
Address 0 =	00000000000000000000000000000	00	000

Index	Valid	Tag	Word0	Word1
0	1	~~0~~ ~~1~~ 0	~~A~~ ~~I~~ A	~~B~~ ~~J~~ B
1	1	0	C	D
2	0			
3	0			

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

Exercise #2: Load #7

Miss Hit Miss Hit Miss Miss Hit

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

	Tag	Index	Offset
Address 4 =	000000000000000000000000000	00	100

Index	Valid	Tag	Word0	Word1
0	1	~~0~~ ~~1~~ 0	~~A~~ ~~I~~ A	~~B~~ ~~J~~ B
1	1	0	C	D
2	0			
3	0			

7. Writing Data: Store #1-1

- Store **X** to

Tag	Index	Offset
0000000000000000000	0000000001	1000

Step 1. Check Cache Block **1**

Index	Valid	Tag	Data: 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

	Tag	Index	Offset
▪ Store **X** to	000000000000000000	0000000001	1000

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

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 #1-3

	Tag	Index	Offset
▪ Store **X** to	000000000000000000	0000000001	1000

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

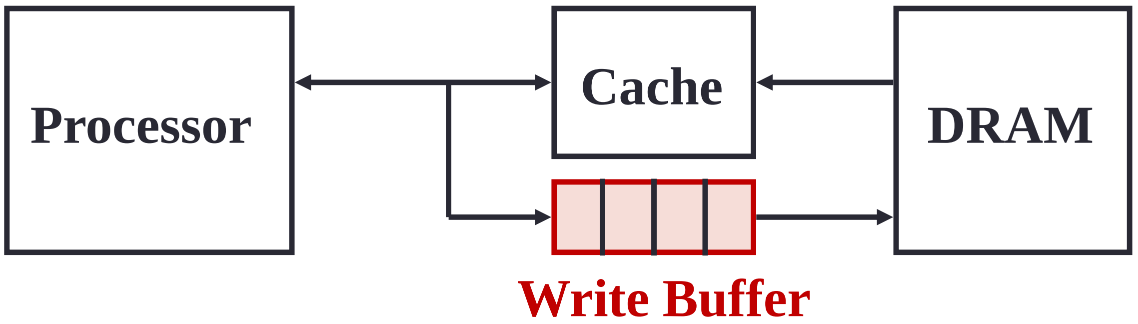
See any problem here?

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

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

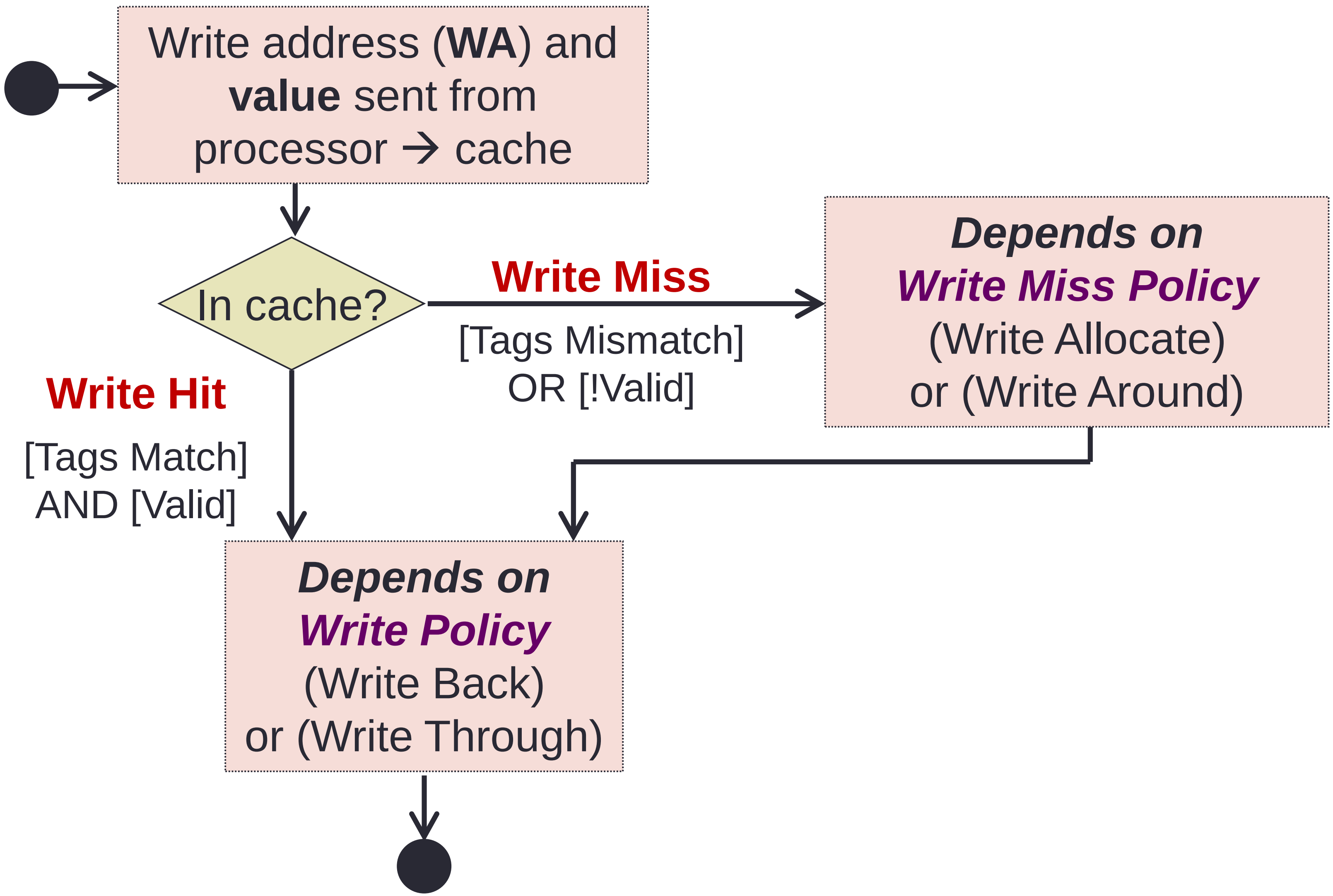
8. Handling Cache Misses

- 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



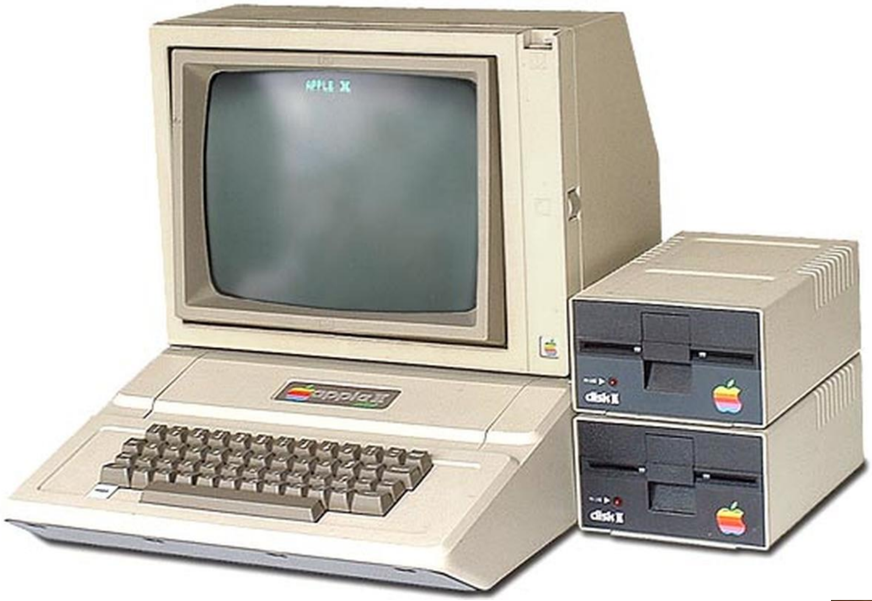
Summary

- 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^{rd} edition)
  - Chapter 5 sections 5.1 – 5.2 (4^{th} edition)

End of File




boedem
FLOPPY DISK
5,25" HD,DS


IBM
Diskette 2