IT5002

Computer Systems and Applications

Caches

colintan@nus.edu.sg

NUS
National University
of Singapore

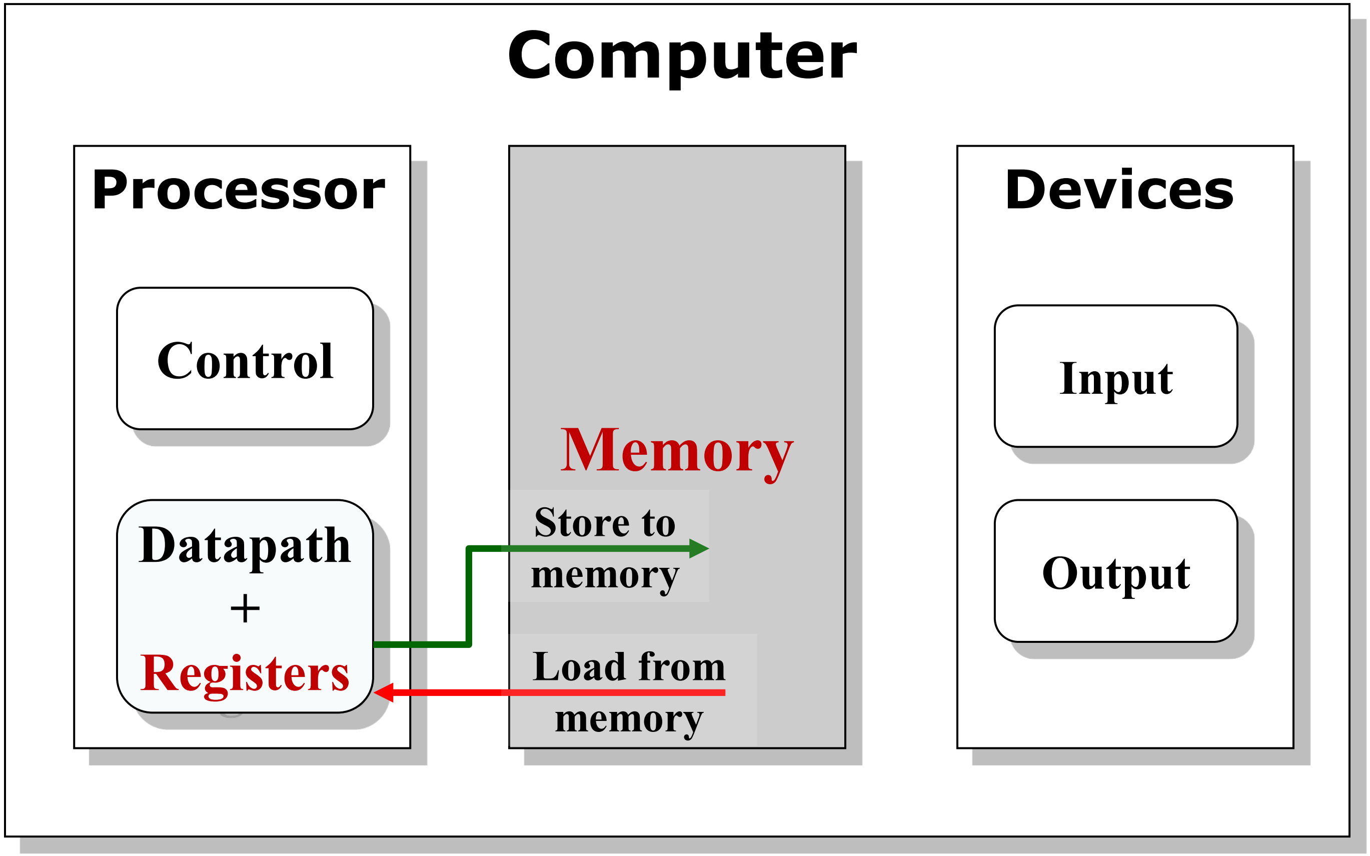
School of Computing

Q & A

- **DO NOT use the Zoom chat for questions. It doesn't appear in the video recordings.**
- **Please ask questions at https://sets.netlify.app/module/61597486a7805d9fb1b4accd**

← OR scan this QR code (may be obscured on some slides)

1. Data Transfer: The Big Picture

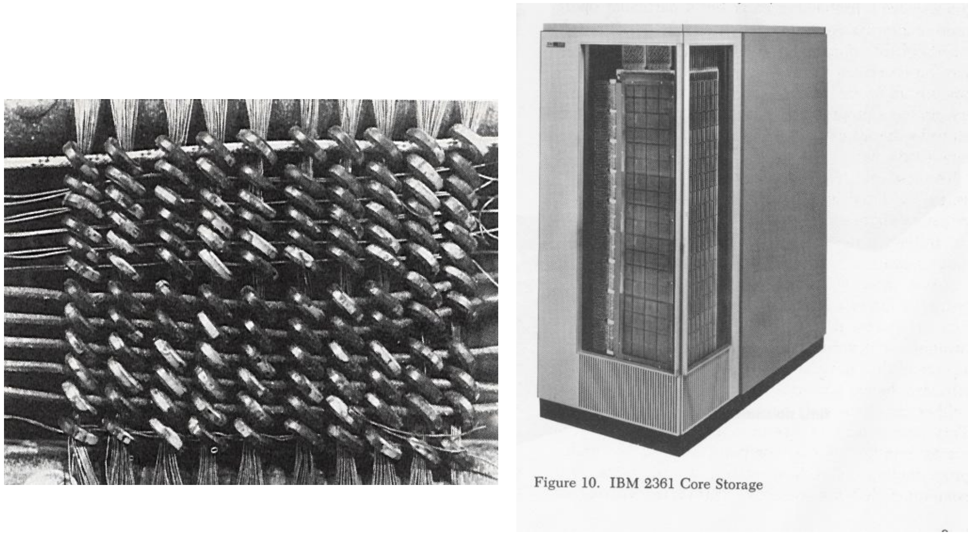


Registers are in the datapath of the processor. If operands are in memory we have to **load** them to processor (registers), operate on them, and **store** them back to memory.

1. Memory Technology: 1950s

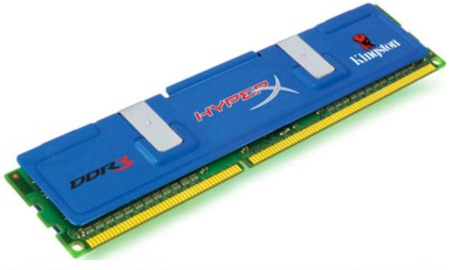
1948: Maurice Wilkes examining EDSAC's delay line memory tubes 16-tubes each storing 32 17-bit words

Maurice Wilkes: 2005



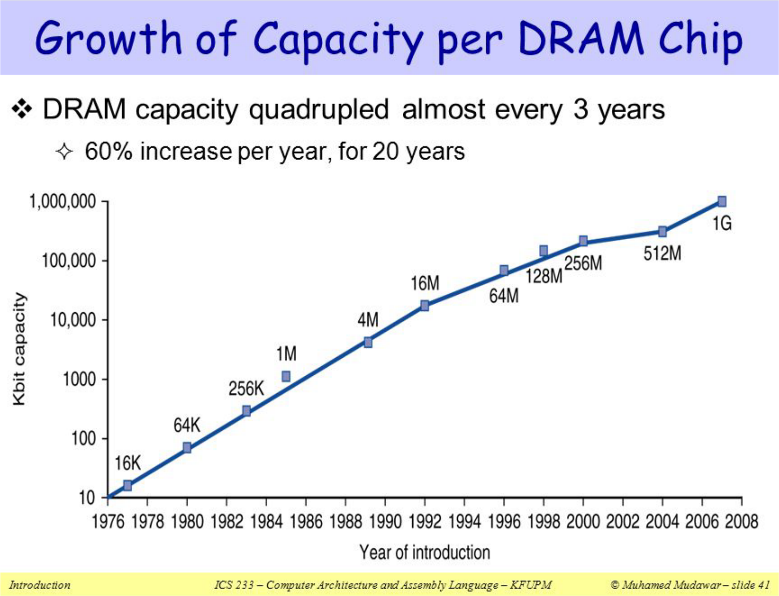
1952: IBM 2361 16KB magnetic core memory

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 (Lower power consumption, higher bandwidth)**

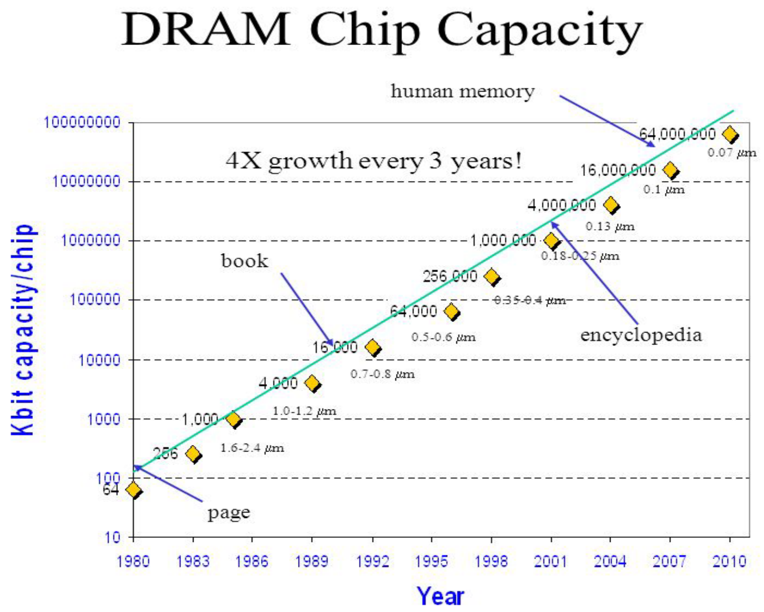
1. DRAM Capacity Growth



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*



DRAM Chip Capacity

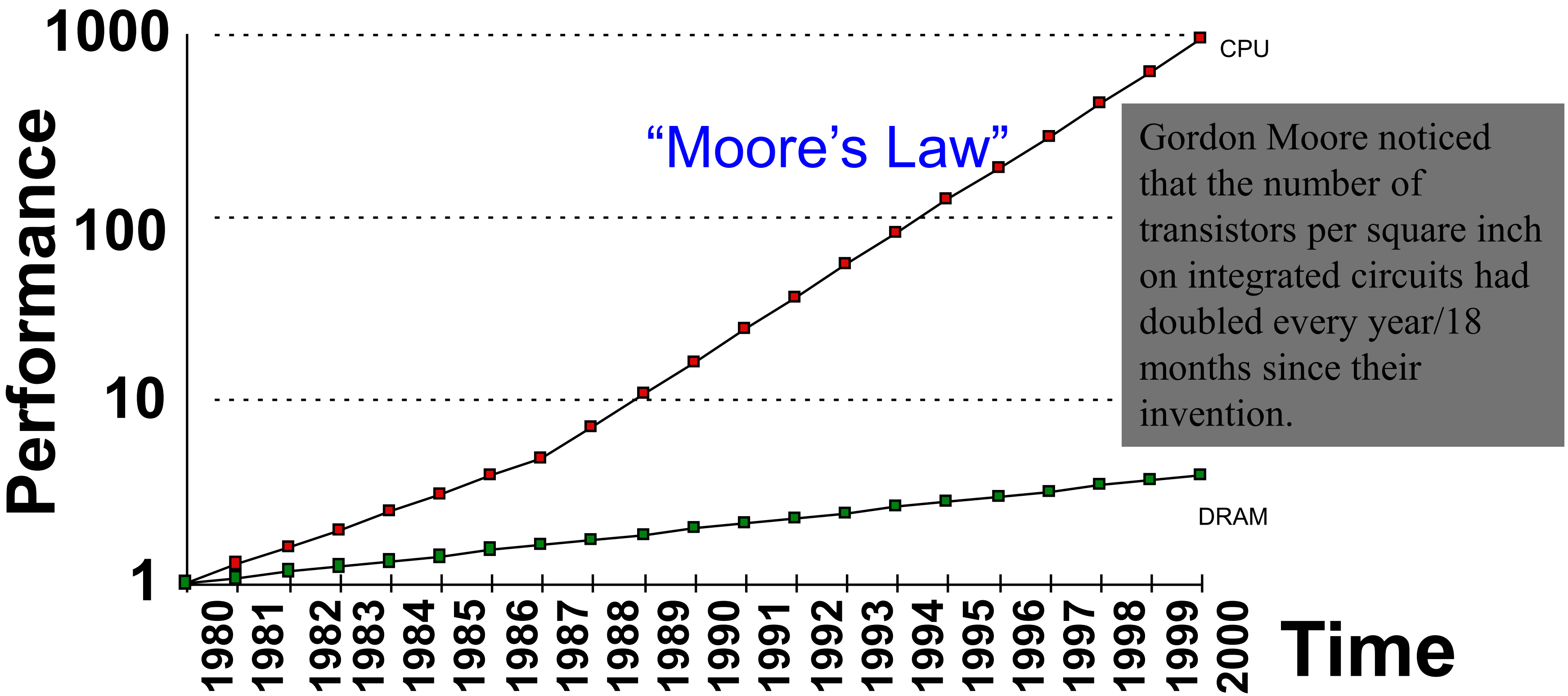
- Unprecedented growth in density, but we still have a problem

1. Processor-DRAM Performance Gap

Memory Wall:

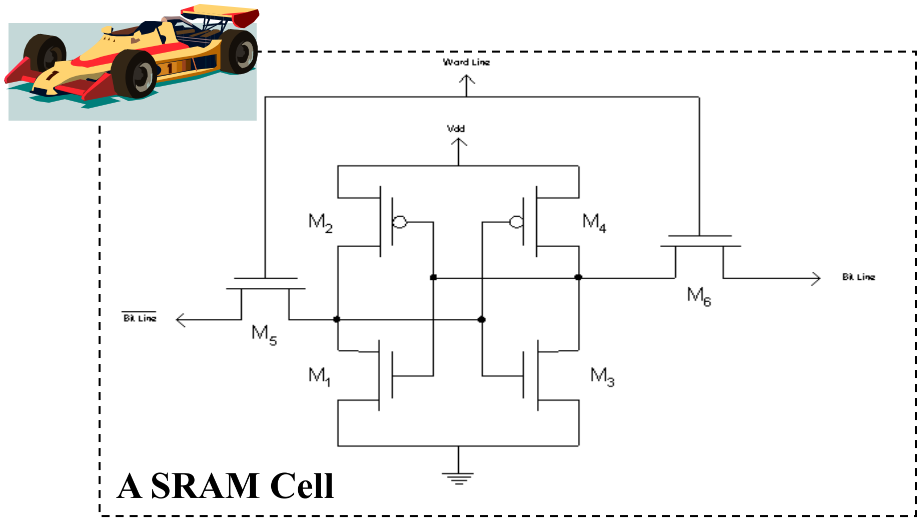
1GHz Processor ➔ 1 ns per clock cycle

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

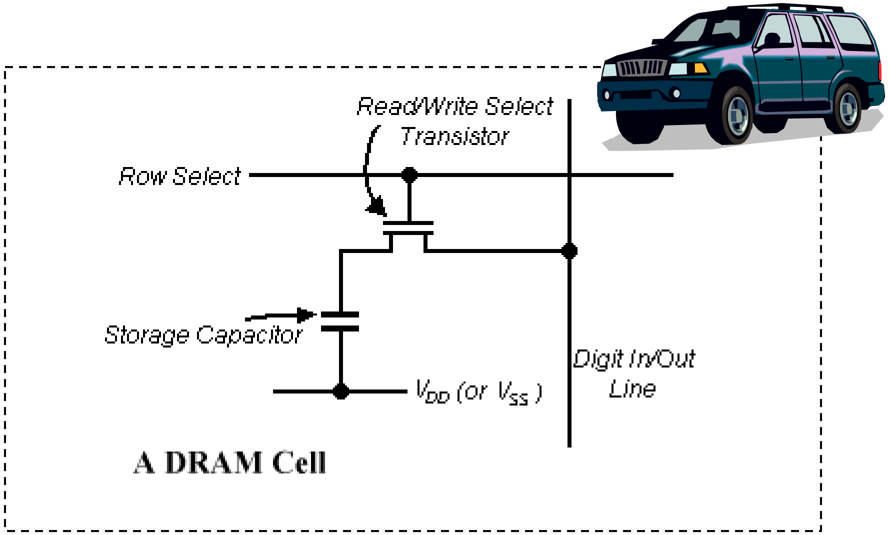


Gordon Moore noticed that the number of transistors per square inch on integrated circuits had doubled every year/18 months since their invention.

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

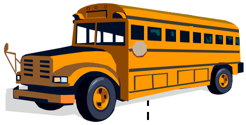
→ **High density**

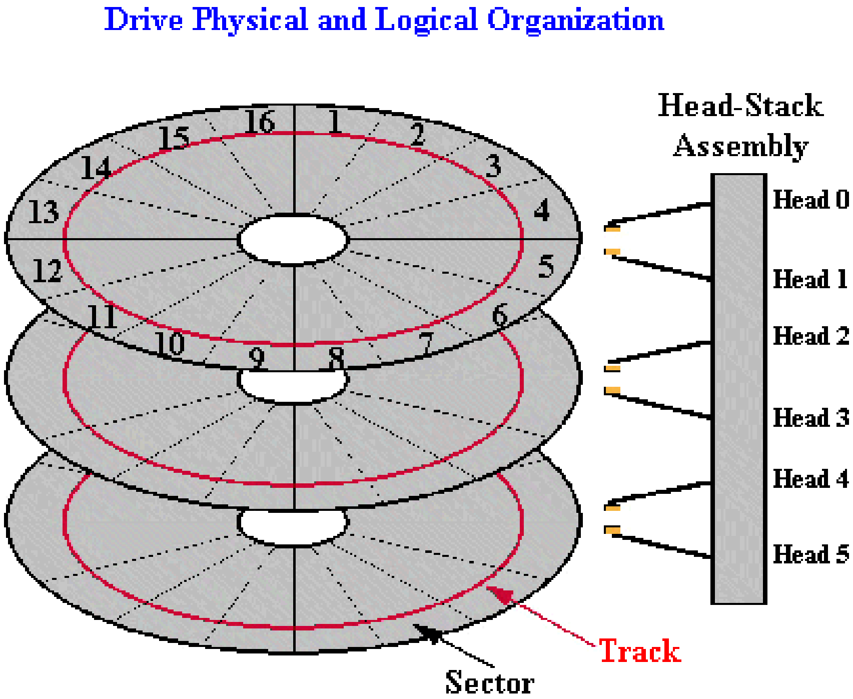
Slow access latency of 50-70ns

Less costly

Used in main memory

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

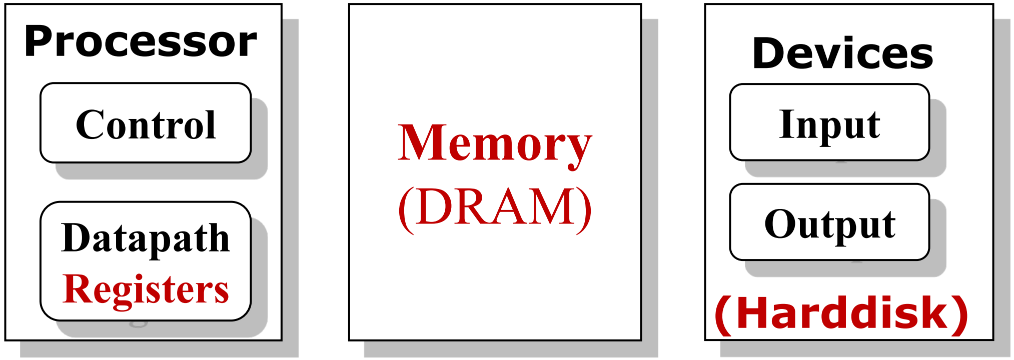


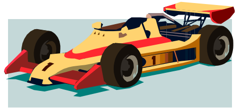


Typical high-end hard disk:

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

1. Quality vs Quantity





	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

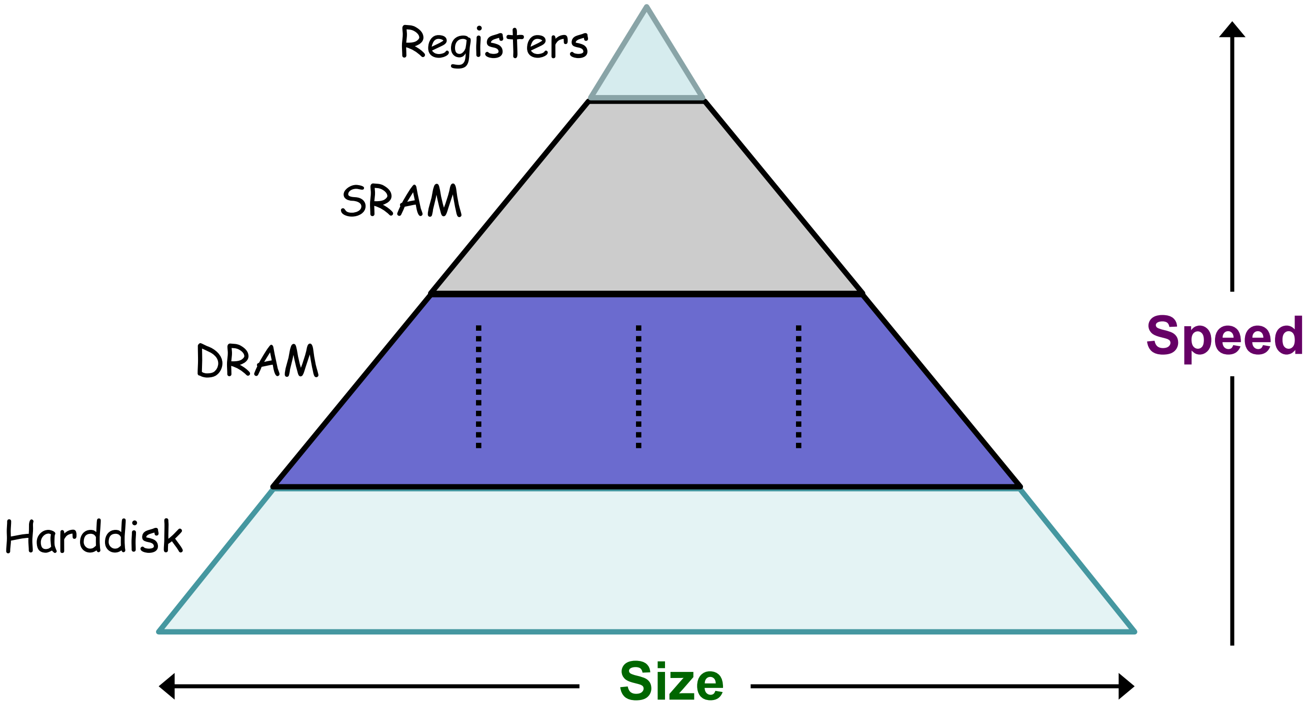
- What we want:
  - A **BIG** and **FAST** memory
  - Memory system should perform like 1GB of SRAM (1ns access time) but cost like 1GB of slow memory

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

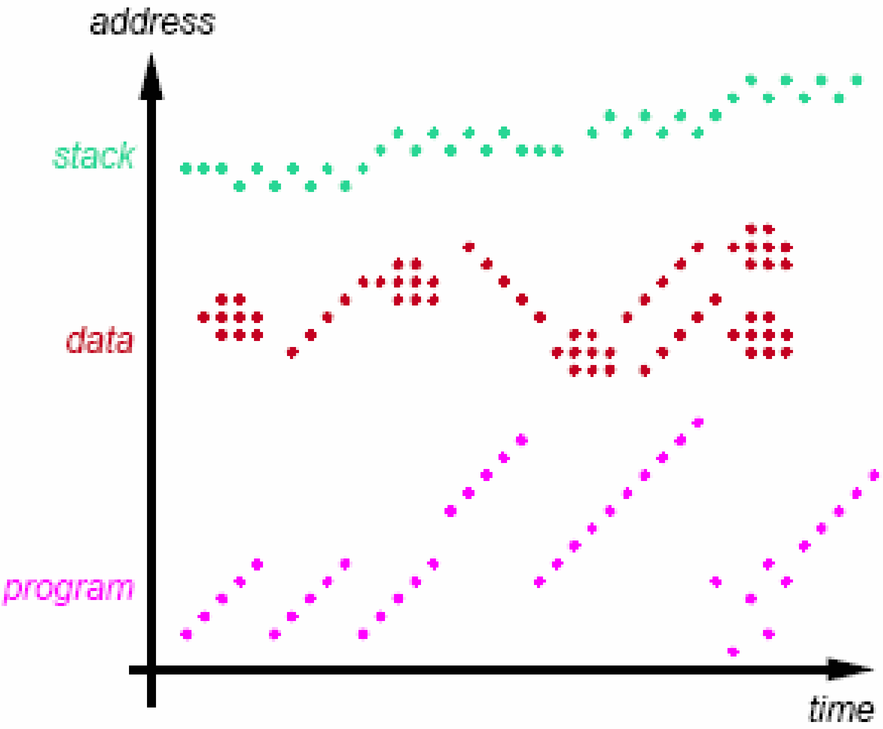
- **Keep the frequently and recently used data in smaller but faster memory**
- **Refer to bigger and slower memory:**
  - **Only when you cannot find data/instruction in the faster memory**
- **Why does it work?**

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

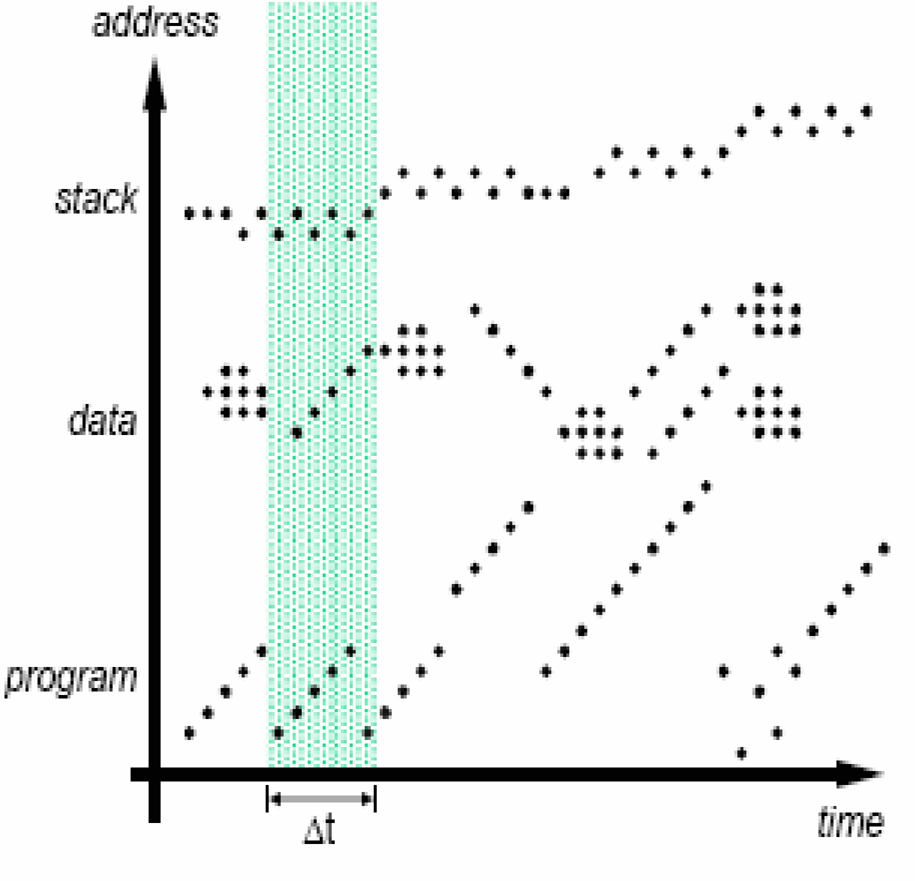
- **Temporal locality**
  - If an item is referenced, it will tend to be referenced again soon
- **Spatial locality**
  - If an item is referenced, nearby items will tend to be referenced soon
- Different locality for
  - Instructions
  - Data



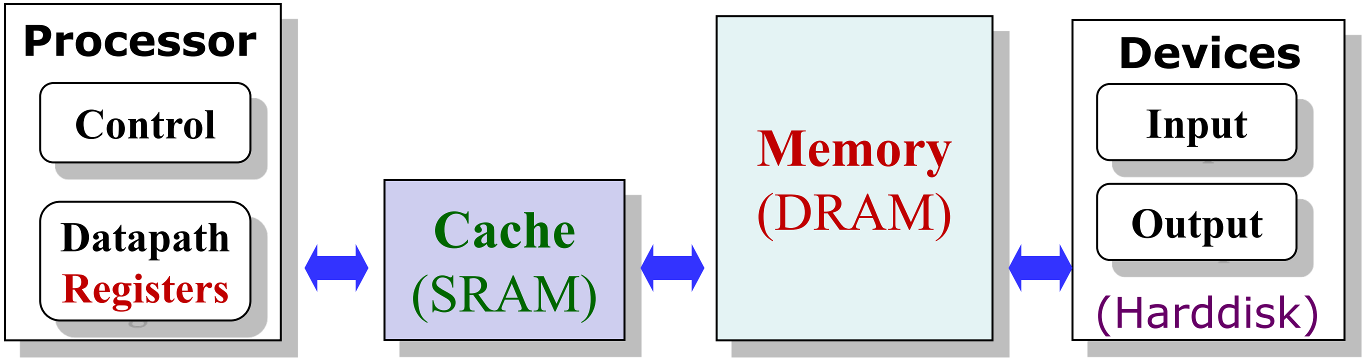
2.1 Working Set: Definition

- **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**

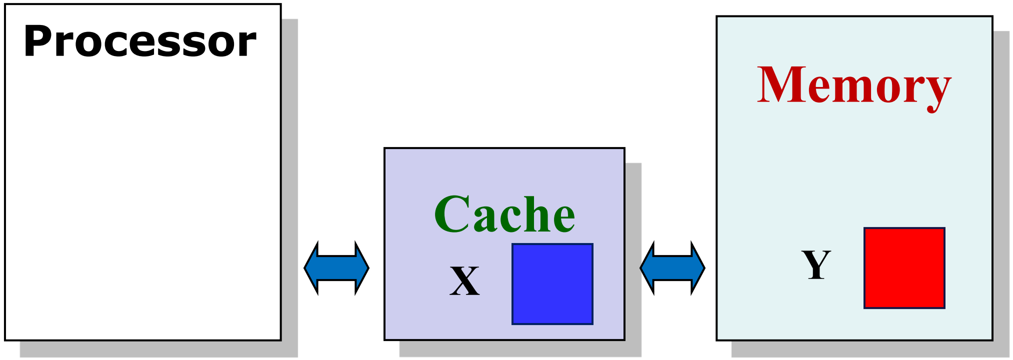


2.2 Two Aspects of Memory Access



- 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.

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

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

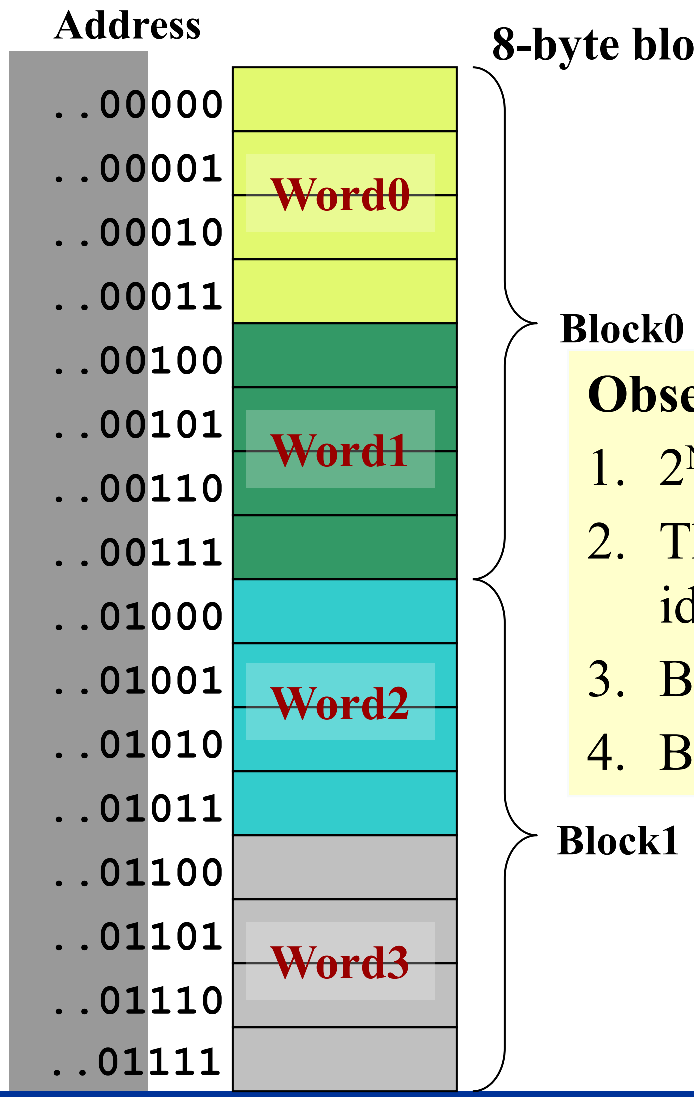
$10h = 9.8 \rightarrow h = 0.98$

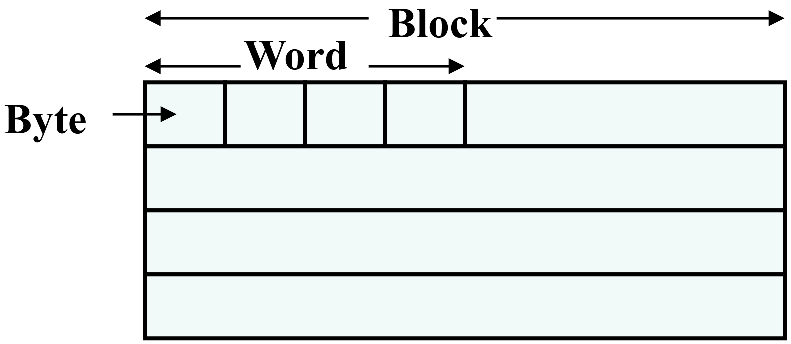
Hence we need a hit rate of **98%**.

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?

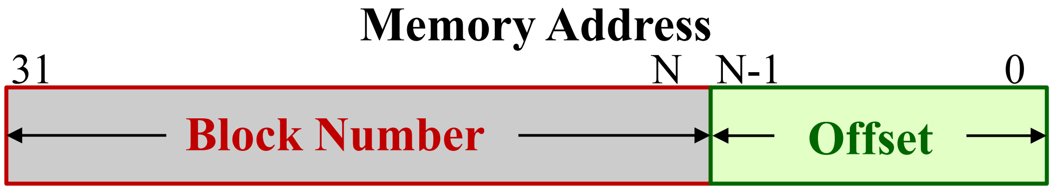
3. Memory to Cache Mapping (2/2)





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

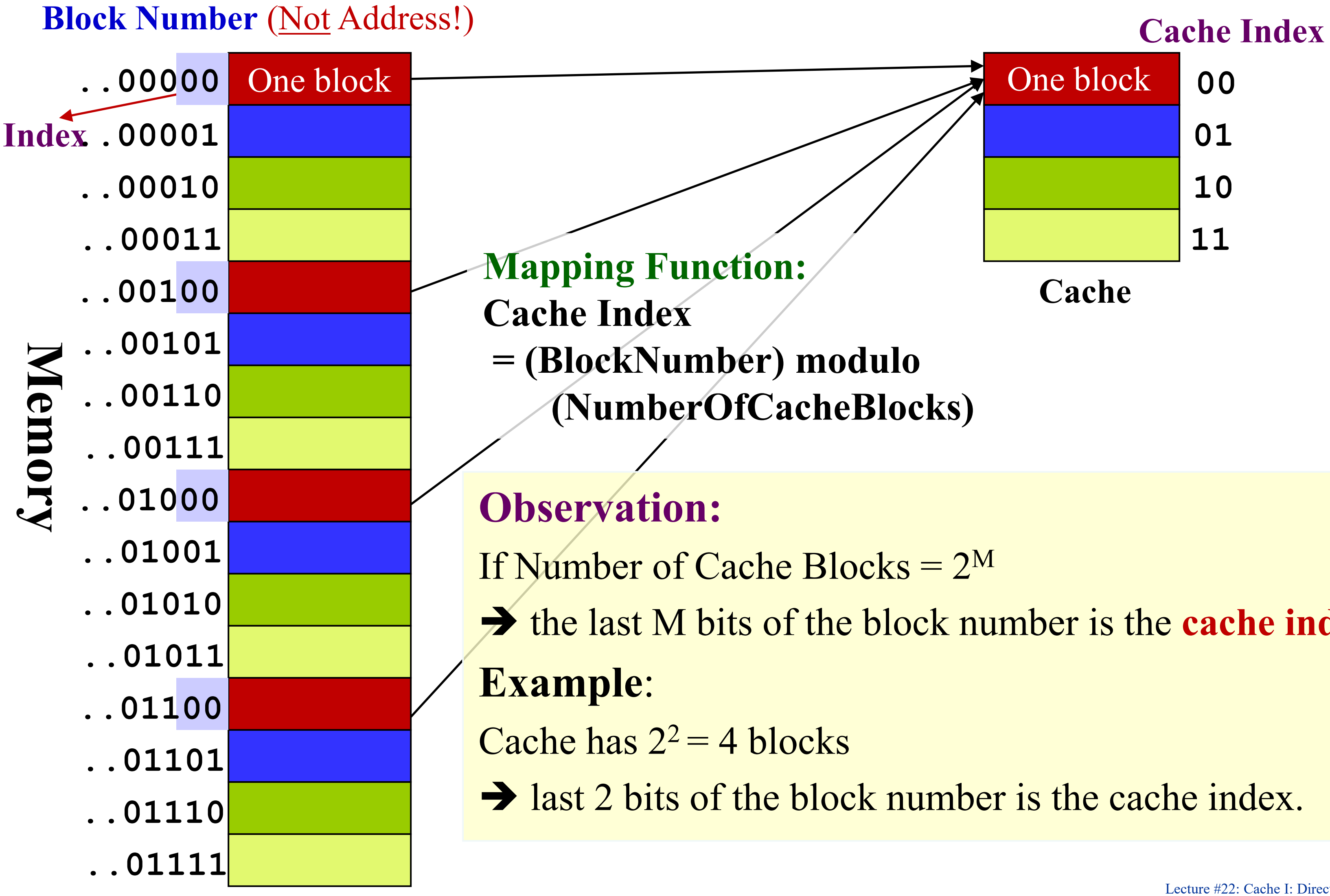


4. Direct Mapping Analogy

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

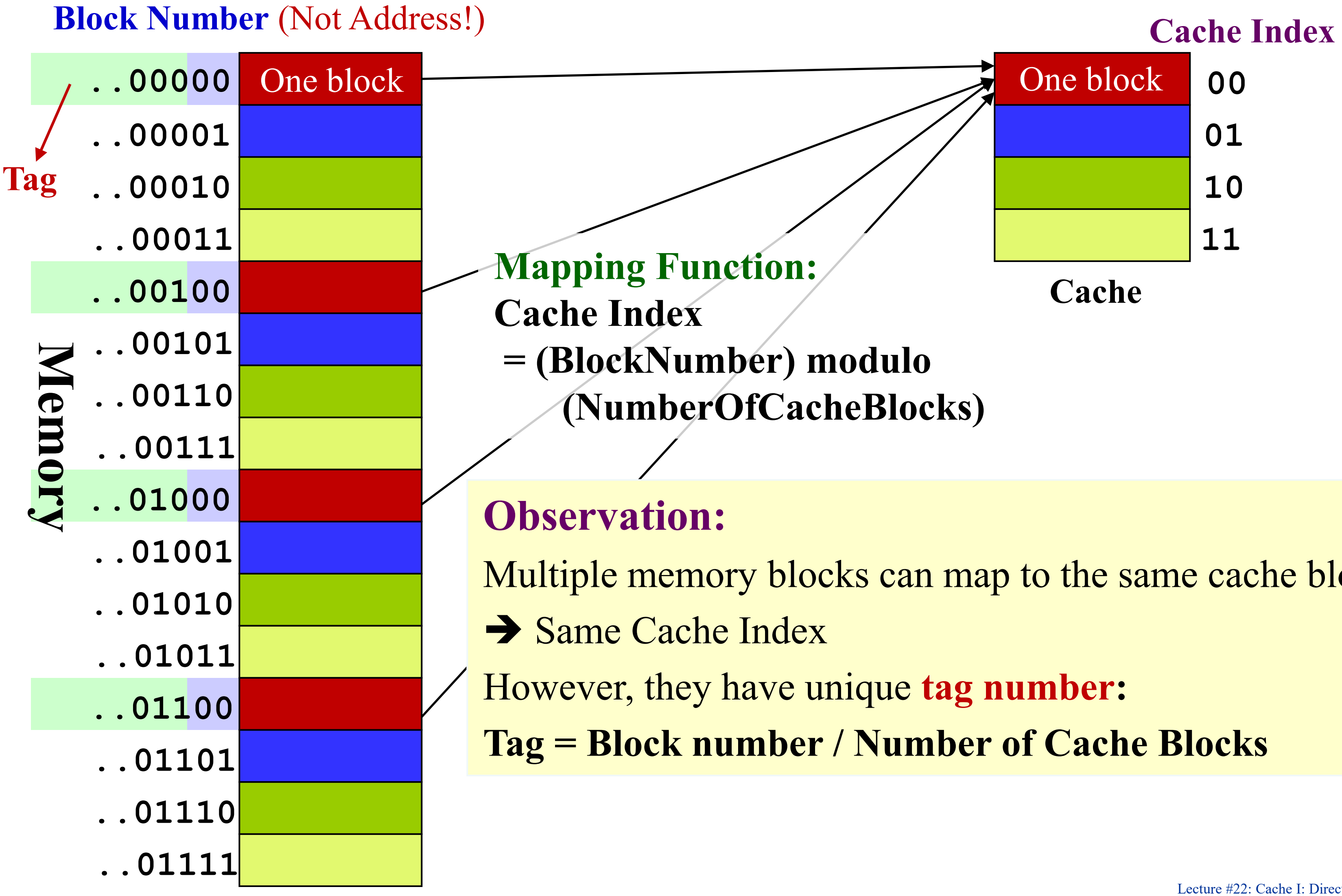
➔ the last M bits of the block number is the **cache index**

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

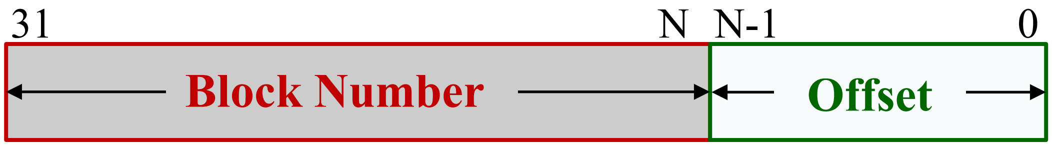
➔ Same Cache Index

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

Tag = Block number / Number of Cache Blocks

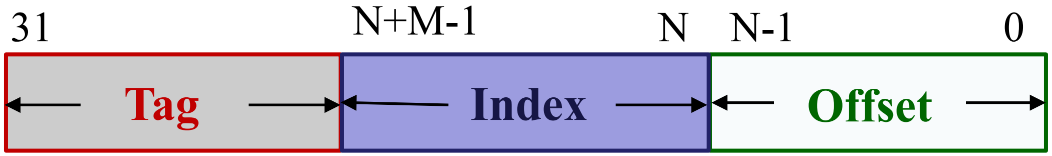
4. Direct Mapped Cache: **Mapping**

Memory Address



Cache Block size = 2^N bytes

Memory Address



Cache Block size = 2^N bytes

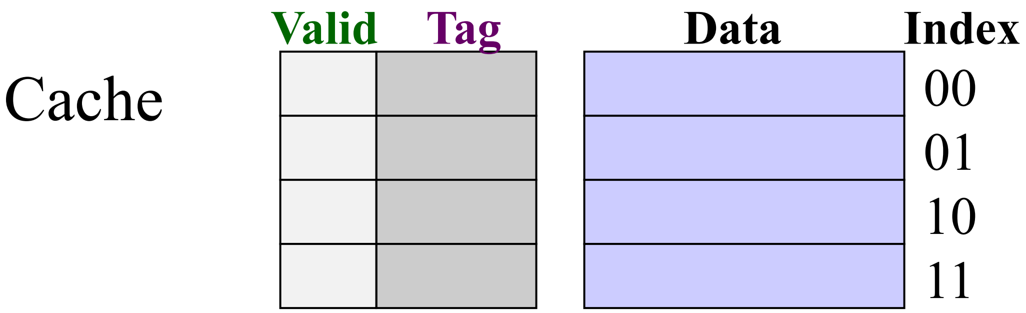
Number of cache blocks = 2^M

Offset = **N bits**

Index = **M bits**

Tag = **32 – (N + M) bits**

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



Cache

Valid	Tag	Data	Index
			00
			01
			10
			11

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

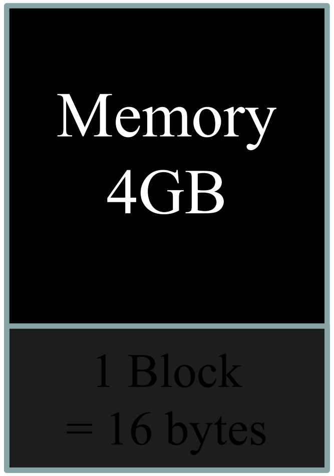
1. **Tag** of the memory block
2. **Valid bit** indicating whether the cache line contains valid data

When is there a cache hit?

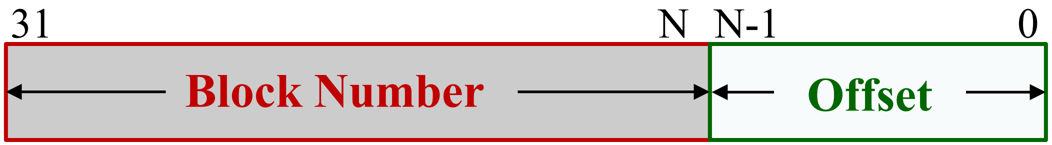
(Valid[index] = TRUE) **AND**

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

4. Cache Mapping: **Example**



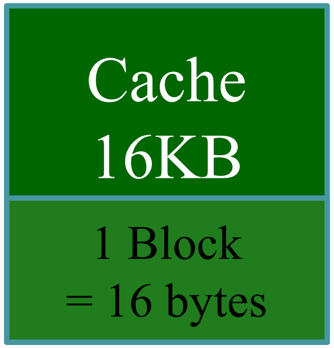
Memory Address

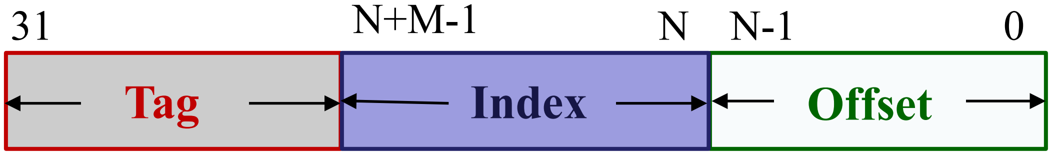


Offset, N = 4 bits

Block Number = 32 – 4 = **28 bits**

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





Number of Cache Blocks

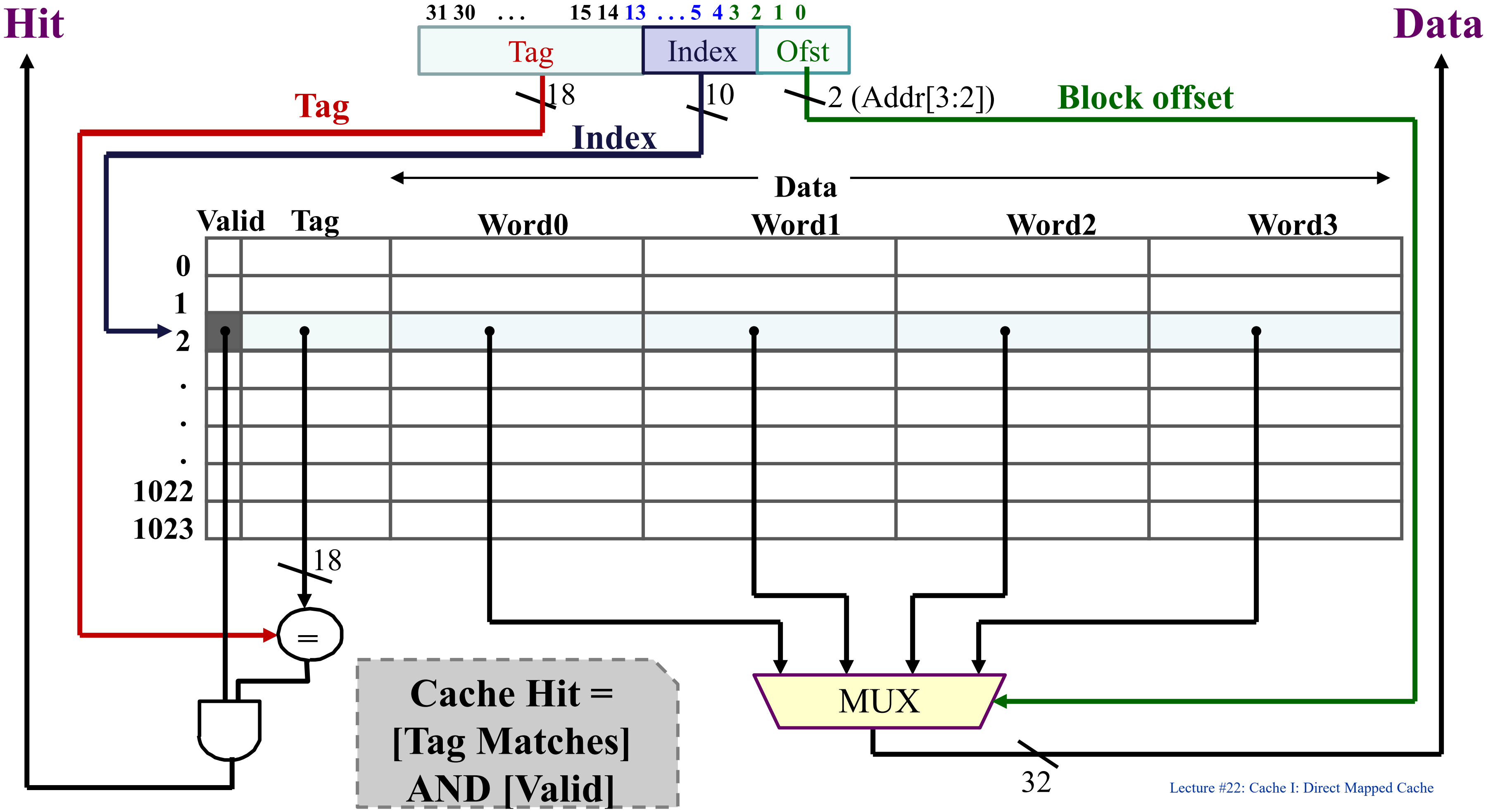
= 16KB / 16bytes = 1024 = $\mathbf{2^{10}}$

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)

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

- **Load from**

Tag	Index	Offset
000000000000000000	0000000001	0100

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	**0**					
2	**0**					
3	**0**					
4	**0**					
5	**0**					
			••• ••• •••	••• ••• •••	••• •••	
1022	**0**					
1023	**0**					

5. Reading 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** 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**

- **Load from**

Tag	Index	Offset
000000000000000000	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	**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**

- **Load from**

Tag	Index	Offset
000000000000000000	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!

			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 #2-3**

	Tag	Index	Offset
▪ **Load from**	000000000000000000	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**	000000000000000000	0000000011	0100

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

			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 #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**	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 #3-3**

- **Load from**

Tag	Index	Offset
000000000000000000	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**

- **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**	Bytes 0-3	Bytes 4-7	Bytes 8-11	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**	000000000000000010	0000000001	1000

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	**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**	000000000000000010	0000000001	1000

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

			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	**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			
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**

- **Load from**

Tag	Index	Offset
000000000000000010	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**	000000000000000000	0000000001	0000

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**	**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
000000000000000000	0000000001	0000

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**	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
0000000000000000000	0000000001	0000

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

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**

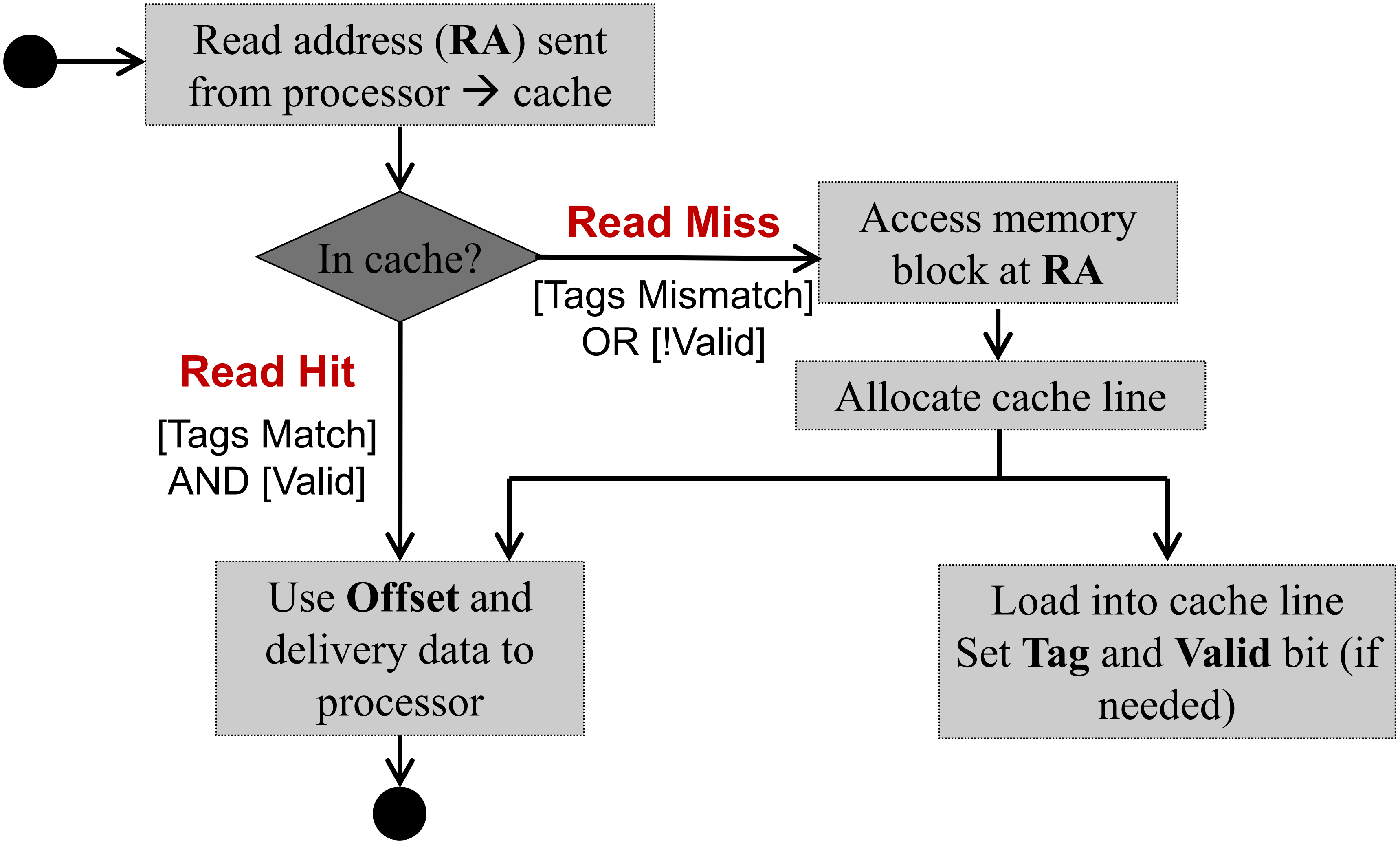
- **Load from**

Tag	Index	Offset
000000000000000000	0000000001	0000

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

			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	**1**	**0**	**I**	**J**	**K**	**L**
4	**0**					
5	**0**					
		••• ••• •••	••• •••	••• •••	•••	
1022	**0**					
1023	**0**					

5. Reading Data: Summary



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

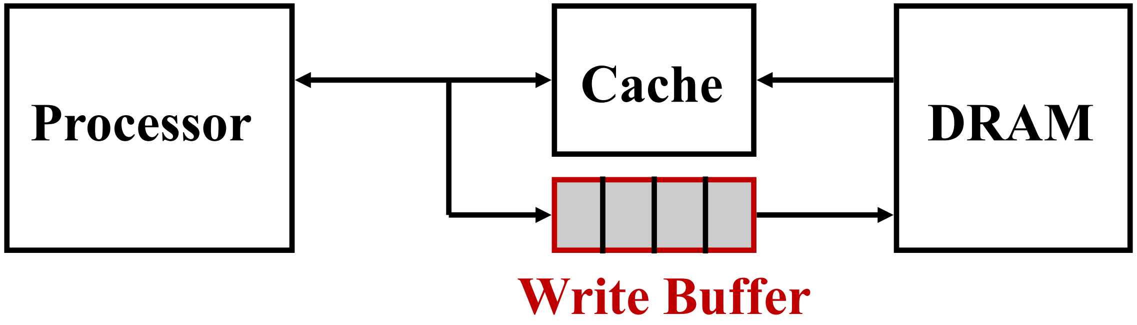
6. Types of Cache Misses

- **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

7. 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)

7. 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

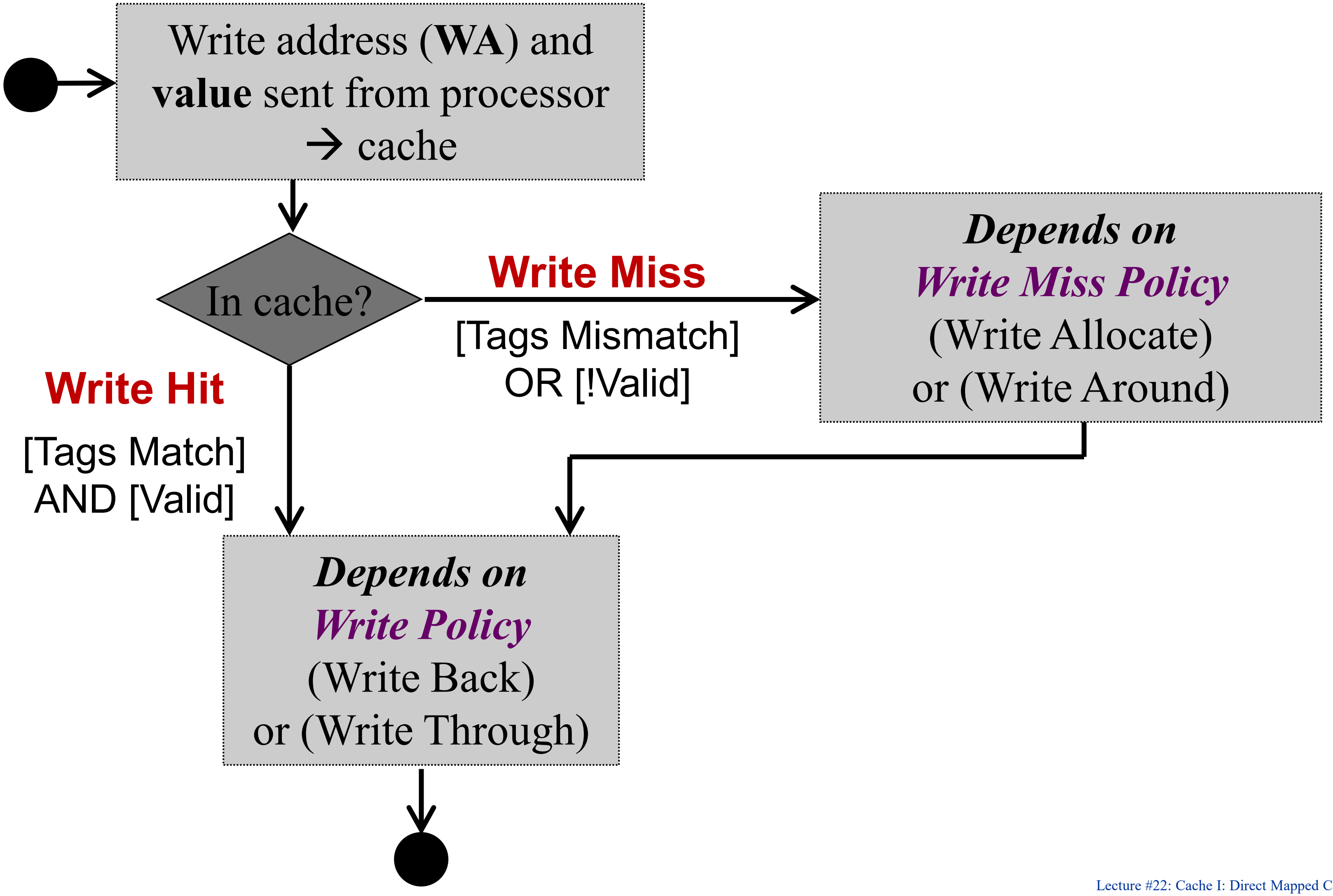
7. Write-Back Cache

- **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

7. 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**

7. Writing Data: Summary



Write address (**WA**) and **value** sent from processor → cache

In cache?

Write Miss

[Tags Mismatch] OR [!Valid]

Depends on

Write Miss Policy

(Write Allocate) or (Write Around)

Write Hit

[Tags Match] AND [Valid]

Depends on

Write Policy

(Write Back) or (Write Through)

8. Set Associative (SA) Cache

- **Compulsory misses**
  - On the first access to a ought into the cache
  - Also called cold start m ses

Solution:
Set Associative Cache

- **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

8. Set Associative Cache: Analogy

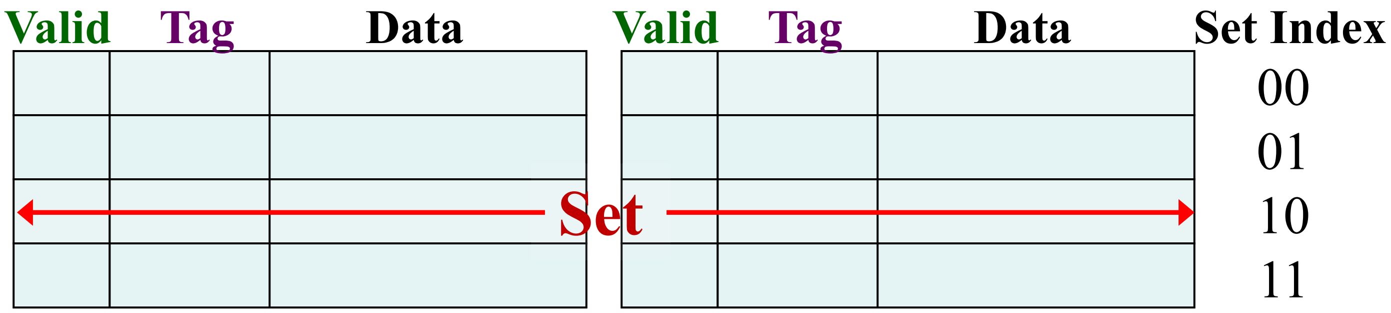
Many book titles start with “T”
➔ Too many conflicts!
Hmm… how about we give more slots per letter, 2 books start with “A”, 2 books start with “B”, …. etc?

8. Set Associative (SA) Cache

- **N-way Set Associative Cache**
  - A memory block can be placed in a fixed number (**N**) of locations in the cache, where **N** > 1

- **Key Idea:**
  - Cache consists of a number of sets:
    - **Each set contains N cache blocks**
  - Each memory block maps to a unique cache set
  - Within the set, a memory block can be placed in **any** of the **N** cache blocks in the set

8. Set Associative Cache: **Structure**



Valid	Tag	Data	Valid	Tag	Data	Set Index
						00
						01
						10
						11

Set

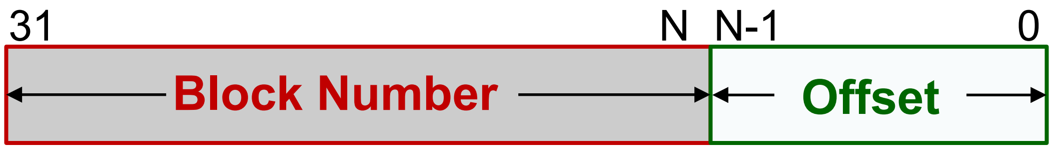
2-way Set Associative Cache

- **An example of 2-way set associative cache**
  - Each set has two cache blocks
- **A memory block maps to a unique set**
  - In the set, the memory block can be placed in **either of the cache blocks**

➔ Need to search both to look for the memory block

8. Set Associative Cache: **Mapping**

Memory Address



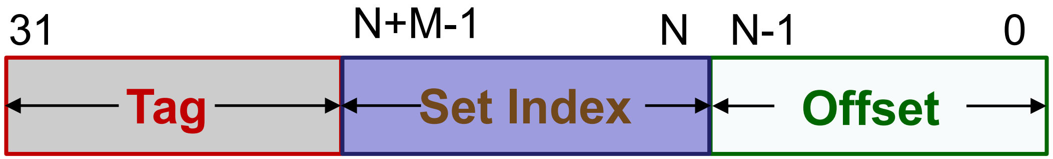
31 N N-1 0

Block Number

Offset

Cache Block size = 2^N bytes

Cache Set Index
= (BlockNumber) modulo (NumberOfCacheSets)



31 N+M-1 N N-1 0

Tag

Set Index

Offset

Cache Block size = 2^N bytes

Number of cache sets = 2^M

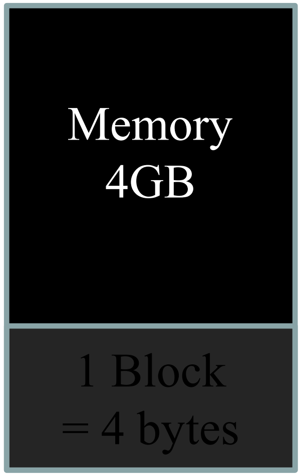
Offset = N bits

Set Index = M bits

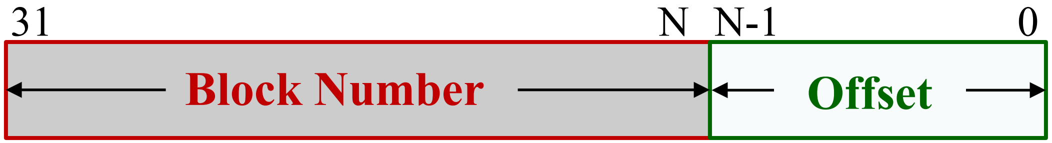
Tag = 32 – (N + M) bits

Observation:
It is essentially unchanged from the direct-mapping formula

8. Set Associative Cache: **Example**



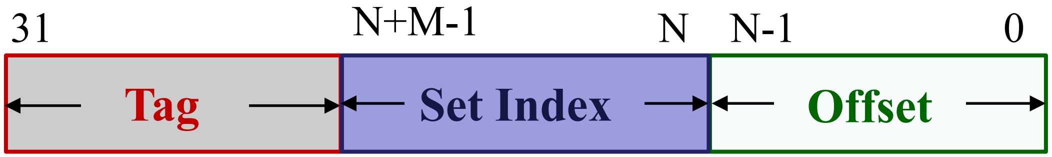
Memory Address

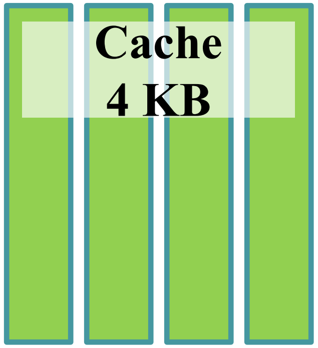


Offset, N = 2 bits

Block Number = 32 – 2 = **30 bits**

Check: Number of Blocks = **2^{30}**





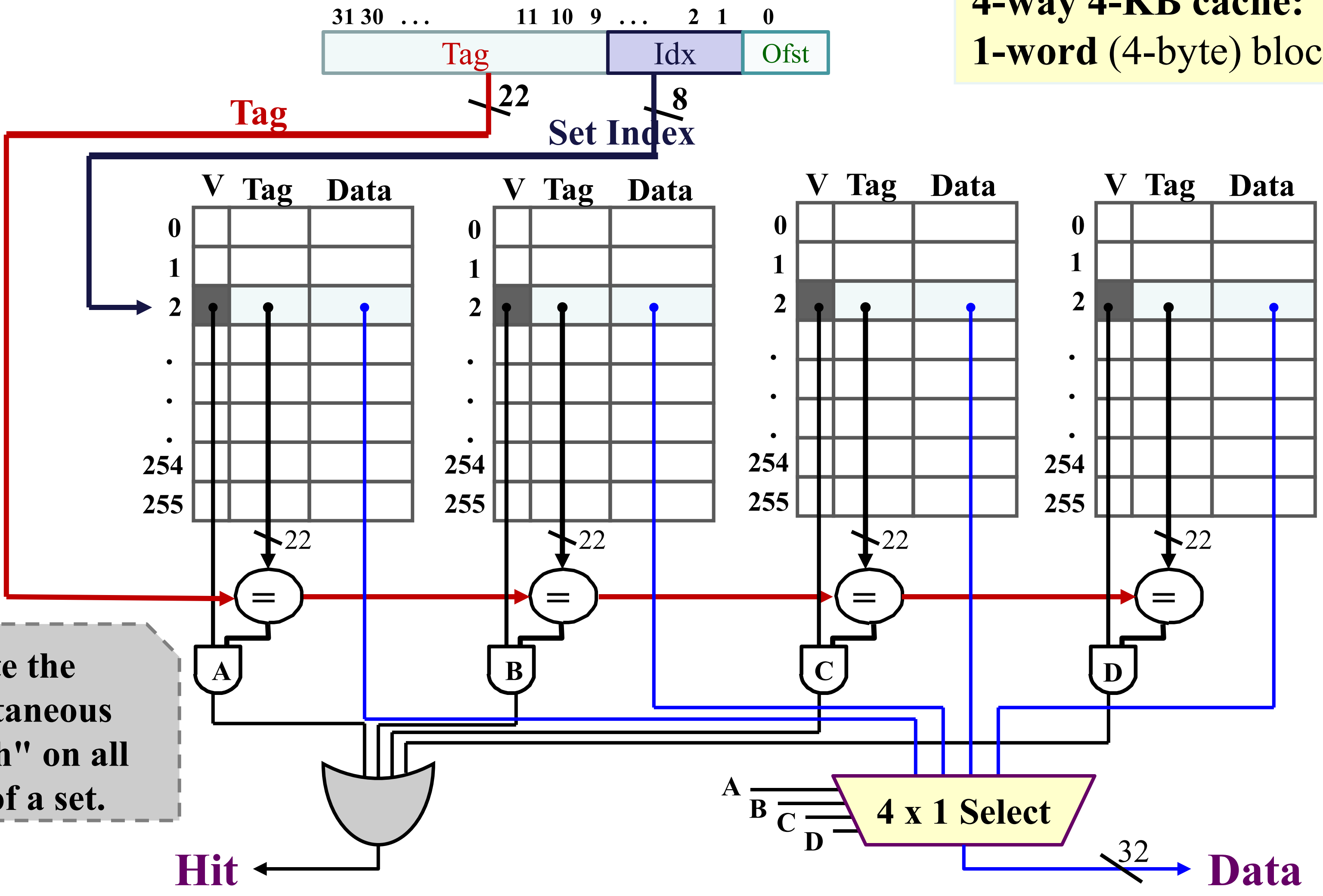
Number of Cache Blocks
= 4KB / 4bytes = 1024 = **2^{10}**

4-way associative, number of sets
= 1024 / 4 = 256 = **2^{8}**
Set Index, M = 8 bits

Cache Tag = 32 – 8 – 2 = **22 bits**

8. Set Associative Cache: **Circuitry**

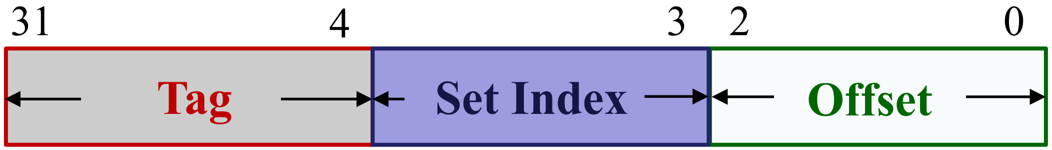
4-way 4-KB cache:
1-word (4-byte) blocks



Note the simultaneous "search" on all tags of a set.

8. SA Cache Example: **Setup**

- **Given:**
  - Memory access sequence: **4, 0, 8, 36, 0**
  - 2-way set-associative cache with a total of four 8-byte blocks ➔ **total of 2 sets**
  - Indicate hit/miss for each access



31 4 | 3 | 2 0

Tag | Set Index | Offset

Offset, N = 3 bits

Block Number = 32 – 3 = 29 bits

2-way associative, number of sets= 2 **= 2^1**

Set Index, M = 1 bits

Cache Tag = 32 – 3 – 1 = **28 bits**

8. SA Cache Example: **Load #1**

Miss

4, 0 , 8, 36, 0

- Load from **4** ➔

Tag	Index	Offset
000000000000000000000000000	0	100

Check: Both blocks in **Set 0** are invalid **[Cold Miss]**

Result: Load from memory and place in **Set 0 - Block 0**

Set Index	Block 0				Block 1			
	Valid	Tag	W0	W1	Valid	Tag	W0	W1
0	~~0~~ 1	0	M[0]	M[4]	0			
1	0				0			

8. SA Cache Example: **Load #2**

Miss Hit

4, 0, 8, 36, 0

Tag	Index	Offset
0000000000000000000000000000	0	000

- Load from **0** ➔

Result:

[Valid and Tags match] in **Set 0-Block 0** **[Spatial Locality]**

Set Index	Block 0				Block 1			
	Valid	Tag	W0	W1	Valid	Tag	W0	W1
0	1	0	M[0]	M[4]	0			
1	0				0			

8. SA Cache Example: **Load #3**

Miss Hit Miss

4, 0 , 8, 36, 0

Tag	Index	Offset
0000000000000000000000000000	1	000

- Load from **8** ➔

Check: Both blocks in **Set 1** are invalid **[Cold Miss]**

Result: Load from memory and place in **Set 1 - Block 0**

	Block 0				Block 1			
Set Index	**Valid**	**Tag**	**W0**	**W1**	**Valid**	**Tag**	**W0**	**W1**
0	1	0	M[0]	M[4]	0			
1	~~0~~ 1	0	M[8]	M[12]	0			

8. SA Cache Example: **Load #4**

Miss Hit Miss Miss

4, 0 , 8, 36, 0

	Tag	**Index**	**Offset**
- Load from **36** ➔	0000000000000000000000000010	0	100

Check: [Valid but tag mismatch] **Set 0 - Block 0**

[Invalid] **Set 0 - Block1** **[Cold Miss]**

Result: Load from memory and place in **Set 0 - Block 1**

	Block 0				**Block 1**			
Set Index	**Valid**	**Tag**	**W0**	**W1**	**Valid**	**Tag**	**W0**	**W1**
0	1	0	M[0]	M[4]	~~0~~ 1	2	M[32]	M[36]
1	1	0	M[8]	M[12]	0			

8. SA Cache Example: **Load #5**

Miss Hit Miss Miss Hit
4, 0 , 8, 36, 0

- Load from **0** ➔

Tag	**Index**	**Offset**
000000000000000000000000000	0	000

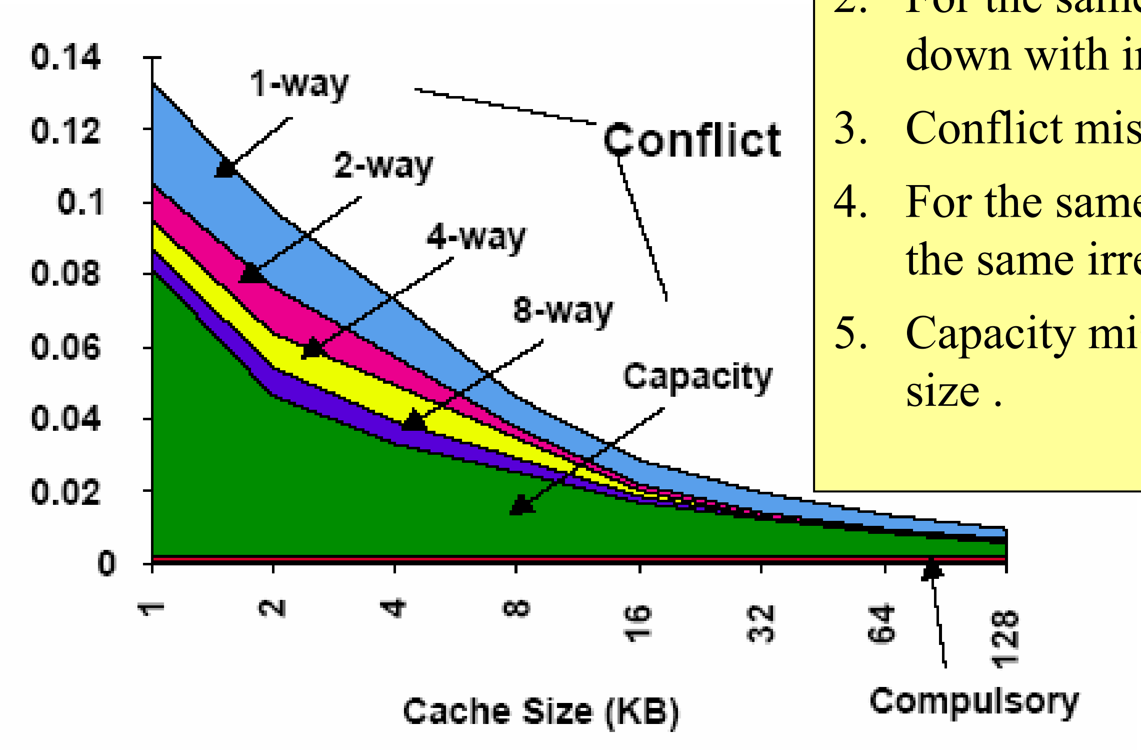
Check: [Valid and tags match] **Set 0-Block 0**
[Valid but tags mismatch] **Set 0-Block1**

[Temporal Locality]

	Block 0				**Block 1**			
Set Idx	**Valid**	**Tag**	**W0**	**W1**	**Valid**	**Tag**	**W0**	**W1**
0	1	0	M[0]	M[4]	1	2	M[32]	M[36]
1	1	0	M[8]	M[12]	0			

9. Cache Performance

Identical block size



Observations:

1. Cold/compulsory miss remains the same irrespective of cache size/associativity.
2. For the same cache size, conflict miss goes down with increasing associativity.
3. Conflict miss is 0 for FA caches.
4. For the same cache size, capacity miss remains the same irrespective of associativity.
5. Capacity miss decreases with increasing cache size .

Total Miss = Cold miss + Conflict miss + Capacity miss

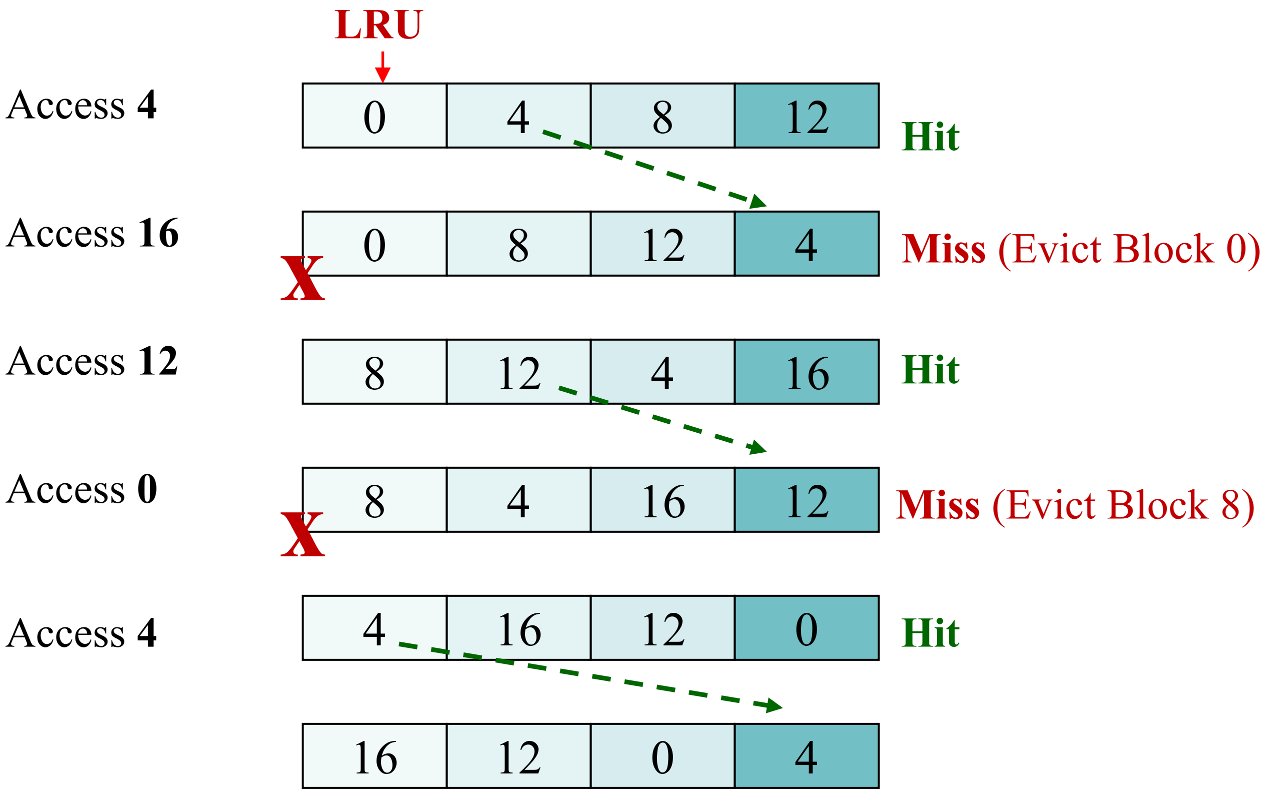
Capacity miss (FA) = Total miss (FA) – Cold miss (FA), when Conflict Miss→0

10. Block Replacement Policy (1/3)

- Set Associative or Fully Associative Cache:
  - Can choose where to place a memory block
  - Potentially replacing another cache block if full
  - Need **block replacement policy**
- Least **R**ecently **U**sed (**LRU**)
  - **How:** For cache hit, record the cache block that was accessed
    - When replacing a block, choose one which has not been accessed for the longest time
  - **Why:** Temporal locality

10. Block Replacement Policy (2/3)

- Least Recently Used policy in action:
  - 4-way SA cache
  - Memory accesses: **0 4 8 12 4 16 12 0 4**



LRU

Access **4**	0	4	8	12	**Hit**
Access **16**	0	8	12	4	**Miss** (Evict Block 0)
Access **12**	8	12	4	16	**Hit**
Access **0**	8	4	16	12	**Miss** (Evict Block 8)
Access **4**	4	16	12	0	**Hit**
	16	12	0	4	

10. Block Replacement Policy (3/3)

- Drawback for LRU
  - Hard to keep track if there are many choices
- Other replacement policies:
  - First in first out (FIFO)
  - Random replacement (RR)
  - Least frequently used (LFU)

11. Summary: Cache Organizations

One-way set associative (direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

11. Summary: Cache Framework (1/2)

Block Placement: Where can a block be placed in cache?

Direct Mapped:

- Only one block defined by index

N-way Set-Associative:

- Any one of the **N** blocks within the set defined by index

Block Identification: How is a block found if it is in the cache?

Direct Mapped:

- Tag match with only one block

N-way Set Associative:

- Tag match for all the blocks within the set

11. Summary: Cache Framework (2/2)

Block Replacement: Which block should be replaced on a cache miss?

Direct Mapped:	N-way Set-Associative:
• No Choice	• Based on replacement policy

Write Strategy: What happens on a write?

Write Policy: Write-through vs write-back

Write Miss Policy: Write allocate vs write no allocate

12. Exploration: Improving Cache Penalty

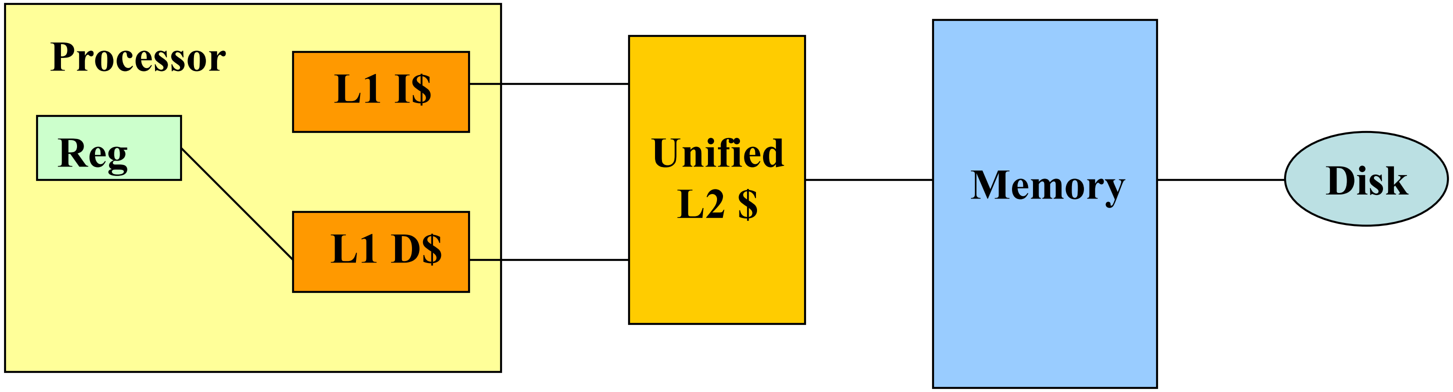
Average Access Time

= Hit rate x Hit Time + (1-Hit rate) x Miss penalty

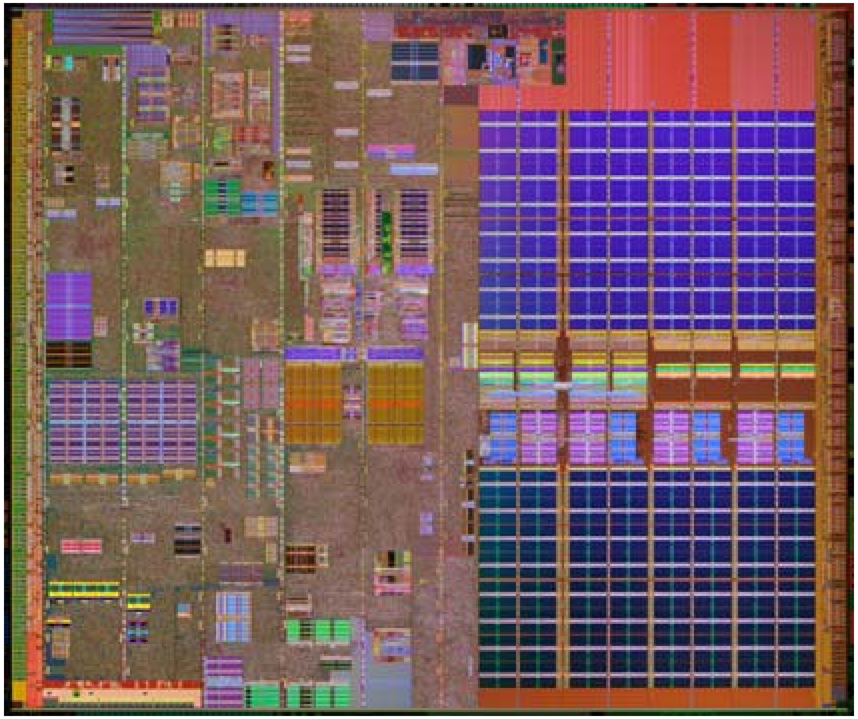
- **So far, we tried to improve Miss Rate:**
  - Larger block size
  - Larger Cache
  - Higher Associativity
- **What about Miss Penalty?**

12. Exploration: Multilevel Cache

- Options:
  - Separate data and instruction caches, or a unified cache
- Sample sizes:
  - **L1**: 32KB, 32-byte block, 4-way set associative
  - **L2**: 256KB, 128-byte block, 8-way associative
  - **L3**: 4MB, 256-byte block, Direct mapped



12. Exploration: Intel Processors

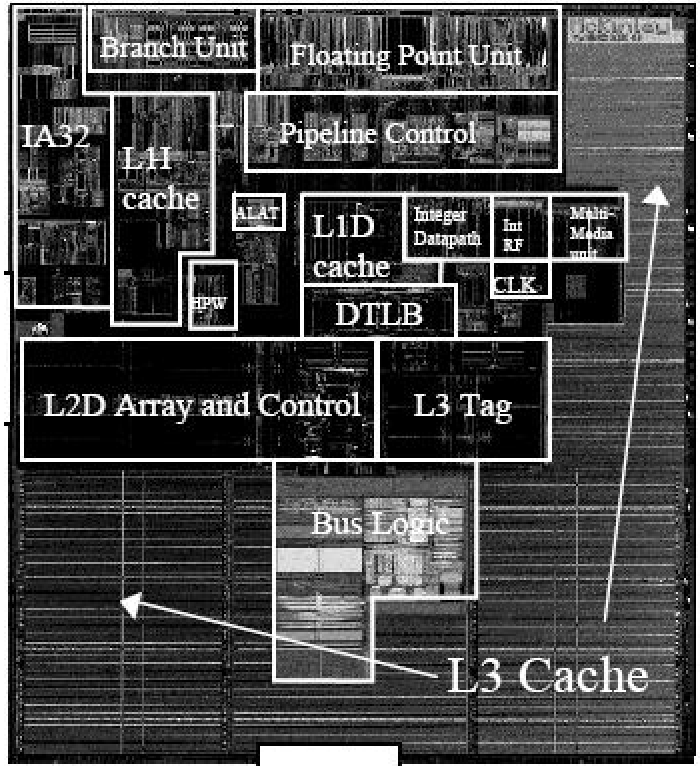


Pentium 4 Extreme Edition

L1: 12KB I$ + 8KB D$

L2: 256KB

L3: 2MB



Itanium 2 McKinley

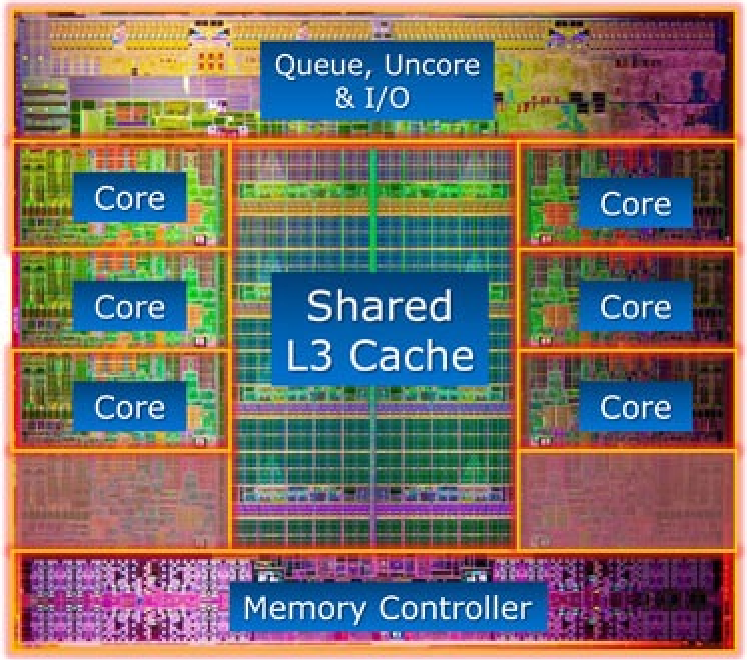
L1: 16KB I$ + 16KB D$

L2: 256KB

L3: 1.5MB – 9MB

12. Exploration: Trend: Intel Core i7-3960K

Intel® Core™ i7-3960X Processor Die Detail



Intel Core i7-3960K

per die:

-2.27 billion transistors

-15MB shared Inst/Data Cache (LLC)

per Core:

-32KB L1 Inst Cache

-32KB L1 Data Cache

-256KB L2 Inst/Data Cache

-up to 2.5MB LLC