CO224 - 2020

Computer Architecture

Memory Systems

Isuru Nawinne

Early Computing Machines

- ENIAC-UPenn, Harvard Mark I (ASCC), etc. in 1940s
  - No concept of software / memory
- The first notion of a "stored program computer"
  - Alan Turing's hypothetical machine, around 1936
  - John Von Neumann, EDVAC (late 1940s, 1000×44-bit words)
- Von Neumann Architecture
  - Same storage and communication pathways for both instructions and data
  - EDSAC-Cambridge (late 1940s, 512×18-bit words)
  - Harvard Architecture – separate storages and pathways
  - Modern day: a mix of both with several levels (hierarchy)

MEMORY TECHNOLOGIES

Memory Types and Characteristics

- SRAM / DRAM / Flash / Magnetic disk / Magnetic Tape

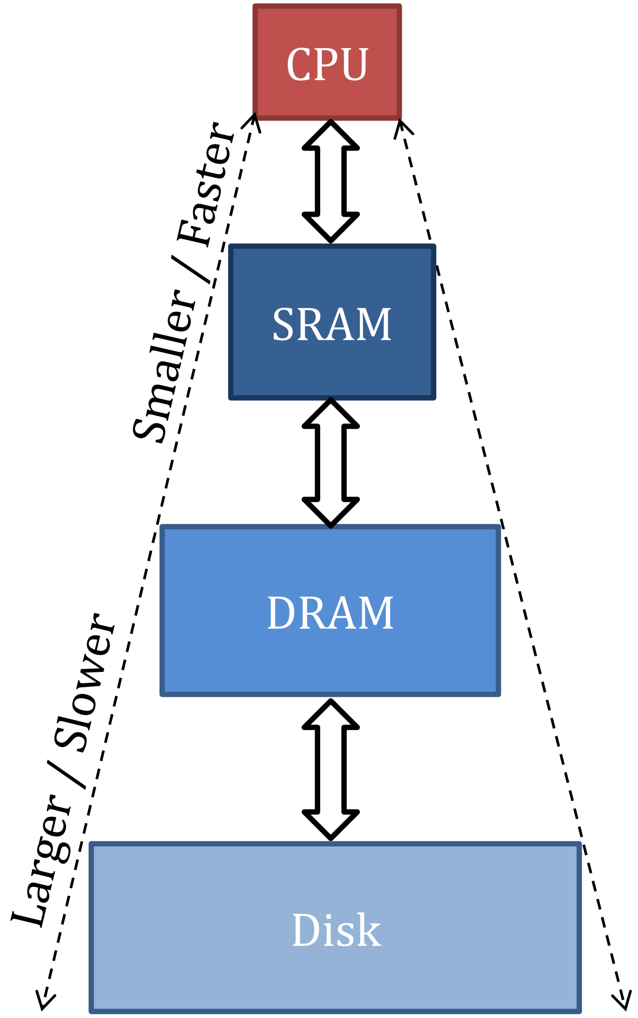
Memory Type	Technology	Access Time	Cycle Time	Capacity Order	Cost per GB
SRAM	Flip-flops	< 1 ns	< 1 ns	KB/MB	\$2000
DRAM	Tran + Cap	~ 25 ns	~ 50 ns	GB	\$10
Flash	NAND MOSFET	~ 70 ns	~ 70 ns	GB	<\$1
Disk	Magnetic	5-10 ms	5-10 ms	TB	<<<\$1

- Main memory (DRAM) → ~ 50ns
  but CPU cycle time → < 1 ns
- Can the MEM stage in the pipeline finish in one CPU cycle ?!?

MEMORY HIERARCHY

Illusion of large and fast memory

- CPU "sees" only the top level
- Each upper level contains a subset of data from the lower level
- What if the top level doesn't have the data that the CPU is asking for? (music library analogy)



Memory Hierarchy

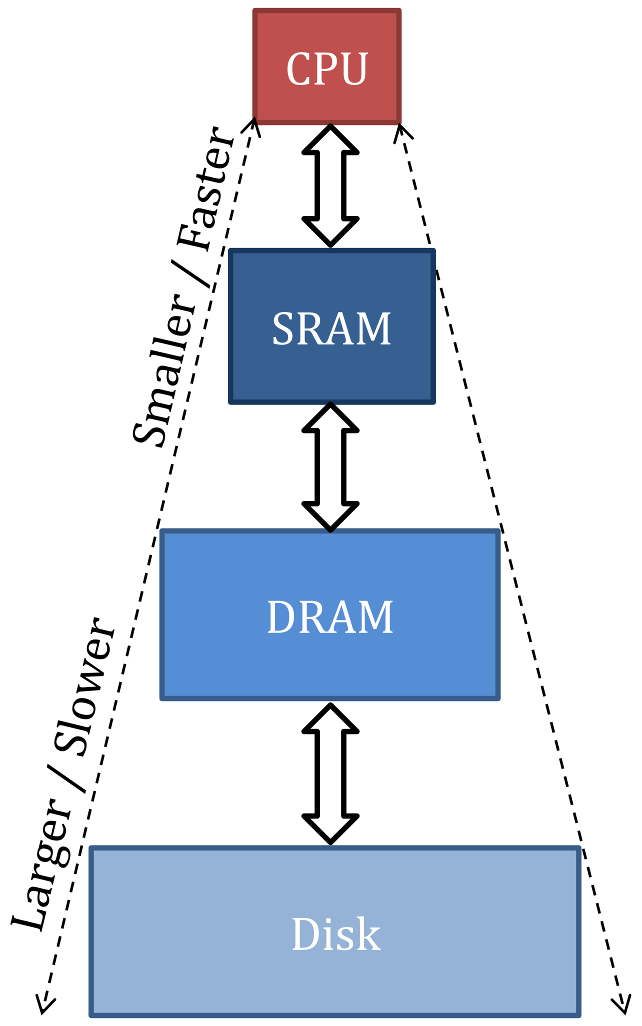
Illusion of large and fast memory

Some terminology:

- *Hits & Misses*
- *Hit-Rate & Miss-Rate*
- *Hit Latency & Miss Penalty*

- CPU performance? Stall on miss

- Do we get a lot of hits at top-level? (high hit-rate)



Memory Hierarchy

Principles of Locality

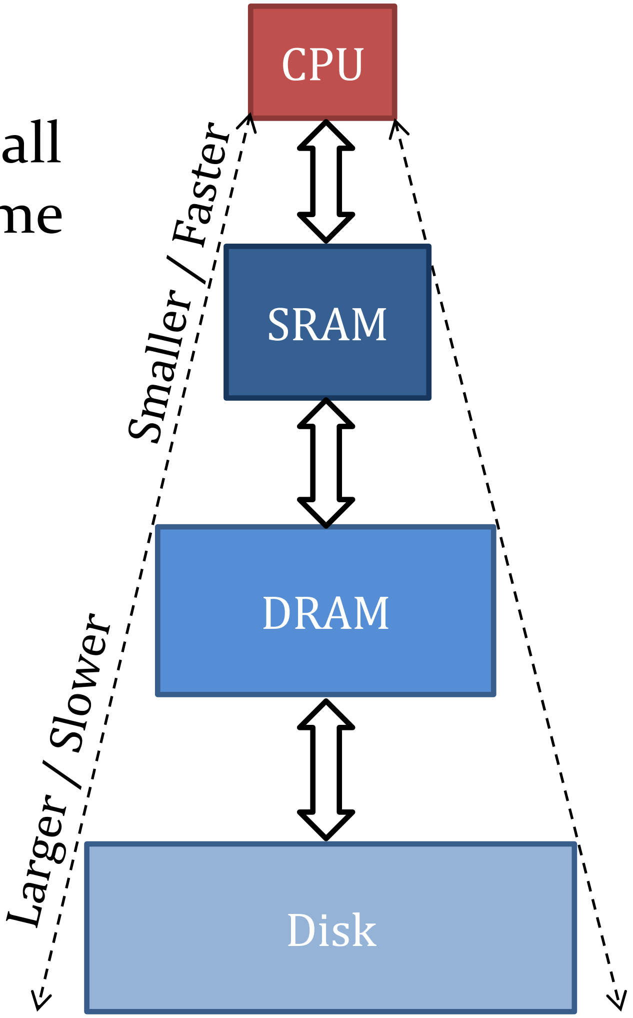
By their nature, programs access only a small portion of their address space at a given time

1. Temporal Locality

Recently accessed data are likely to be accessed again soon

2. Spatial Locality

Data located close to recently accessed data are likely to be accessed soon



Cache Memory

- A memory device used as the top-level in the memory hierarchy
- Exploits the principles of locality to improve memory access performance
  - Temporal locality → *Evicting* of data (cache controller)
  - Spatial locality → Handling *Blocks* of data
- *Average Access Time* for a cache:

$$T_{avg} = HL + (1 - HR) \times MP$$

Architecture of the cache

Program (Memory access pattern)

Architecture of the main memory

CACHE MEMORY

Data Placement

How do we usually find data in memory?

- By using an address
- Each byte-sized location in memory has an address (byte-address):

0000_0000_0000_0000_0000_0000_0000_1010

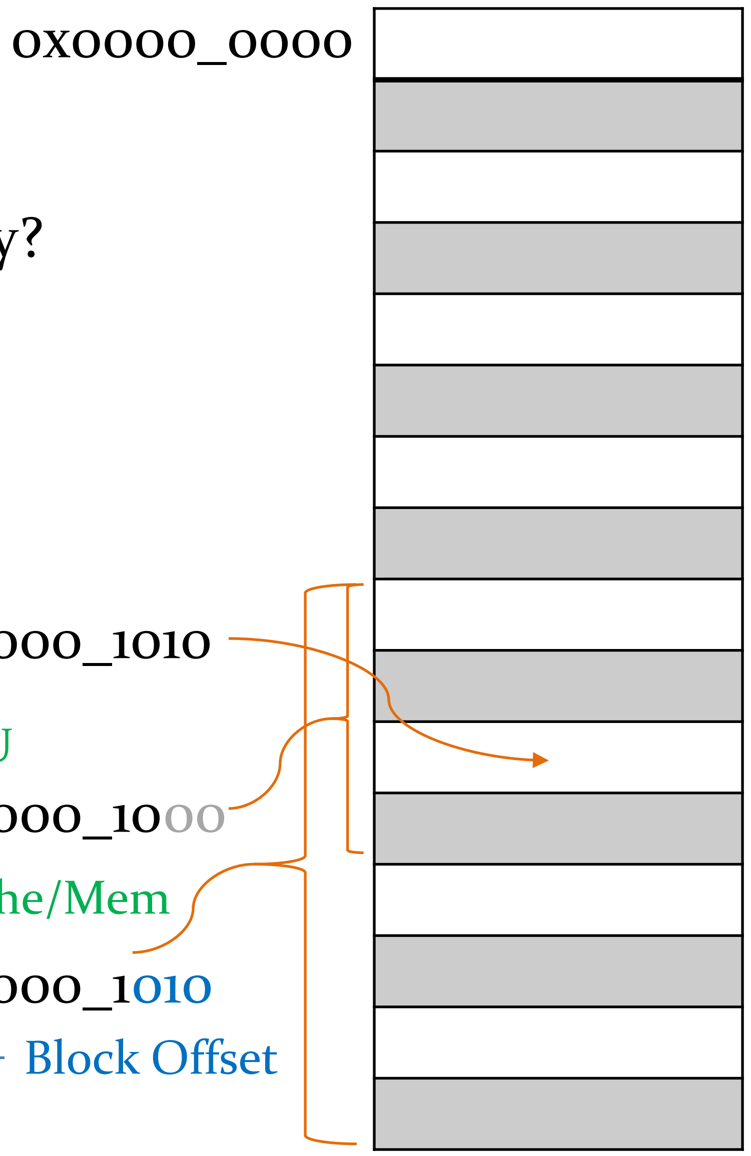
word-address (4-bytes in a word): *CPU

0000_0000_0000_0000_0000_0000_0000_1000

block-address (8-bytes in a block): *Cache/Mem

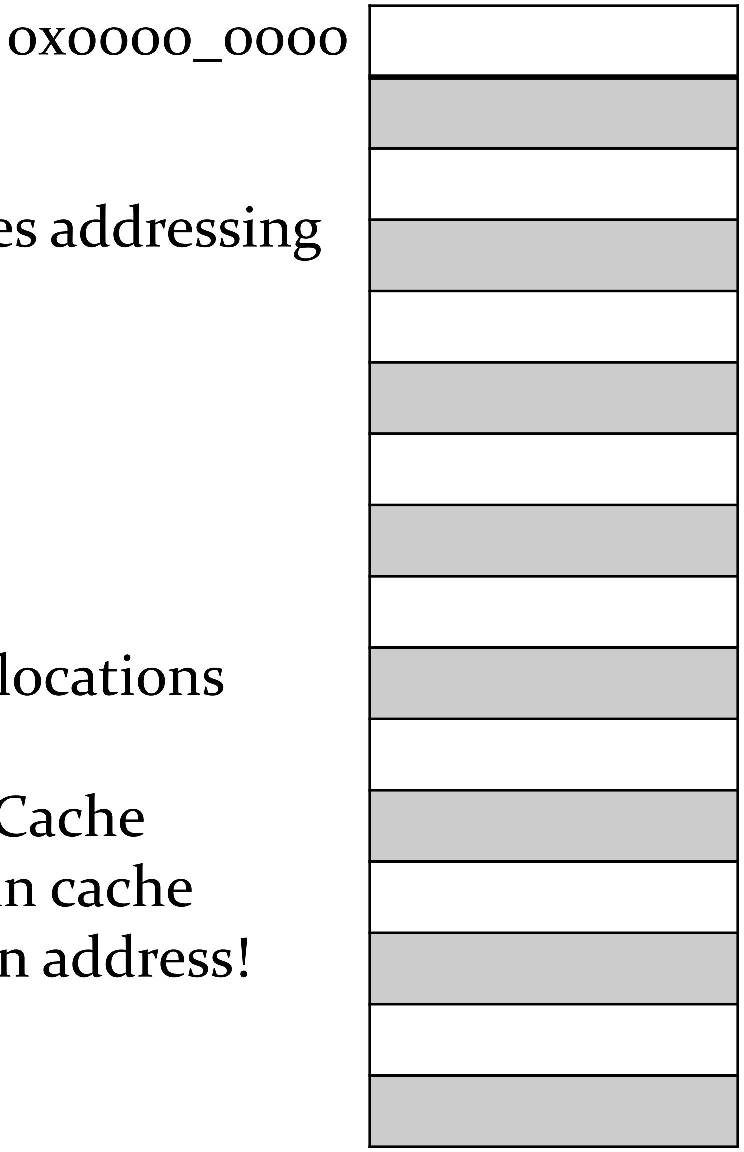
0000_0000_0000_0000_0000_0000_0000_1010

word-offset / byte-offset ⟵ Block Offset



CACHE MEMORY

Data Placement



Cache is smaller than memory, how does addressing work there?

- Can store address along with data.
  But that's a lot of space!
  And need to "search", takes time!

- Mapping memory addresses to cache locations

  Simplest mapping → Direct Mapped Cache
  Only one place in cache
  for a given address!

CACHE MEMORY

Direct Mapped Cache

A power of 2

$$\textit{Location in cache} = \textit{Memory block–address} \ \% \ \textit{No. of blocks in cache}$$

Index

- Assume **block-size = 8 bytes** and cache can hold **8 blocks**

0000_0000_0000_0000_0000_0000_0000_1XXX

% 8 = 001

Index	Data Block
000	
001	
010	
011	
100	
101	
110	
111	

- When CPU wants to access a data word in this block, "index bits" of the block-address are used to directly access the correct cache block.

CACHE MEMORY

Direct Mapped Cache

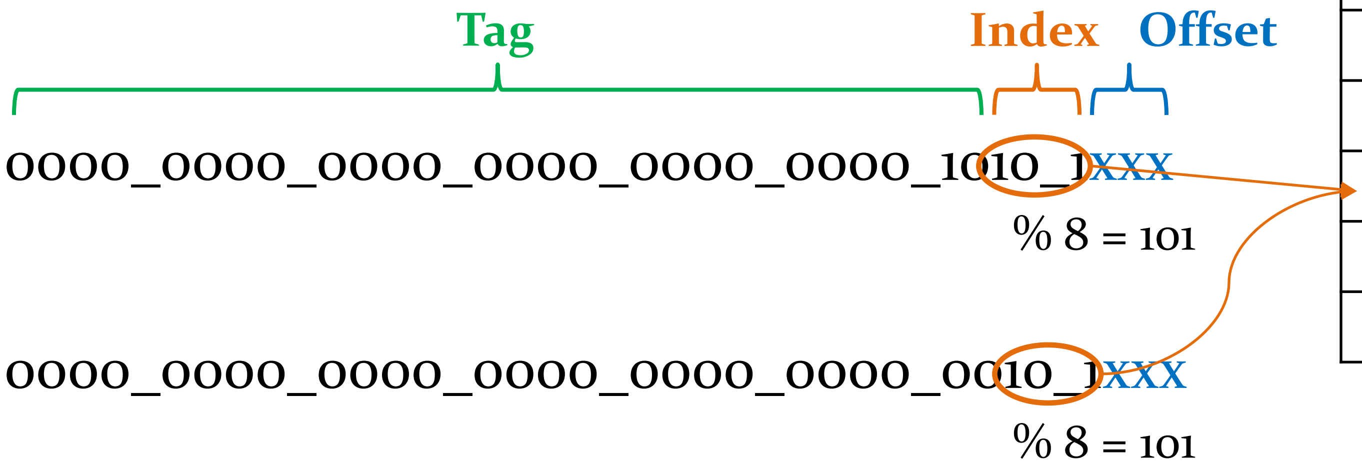
A power of 2

$$\underbrace{\text{Location in cache}}_{\text{Index}} = \text{Memory block-address} \;\%\; \overset{\text{A power of 2}}{\text{No. of blocks in cache}}$$

Index

- Assume **block-size = 8 bytes** and cache can hold **8 blocks**

Cache consists of Data Array + Tag Array



Tag
Index
Offset
0000_0000_0000_0000_0000_0000_1010_1XXX
% 8 = 101
0000_0000_0000_0000_0000_0000_0010_1XXX
% 8 = 101

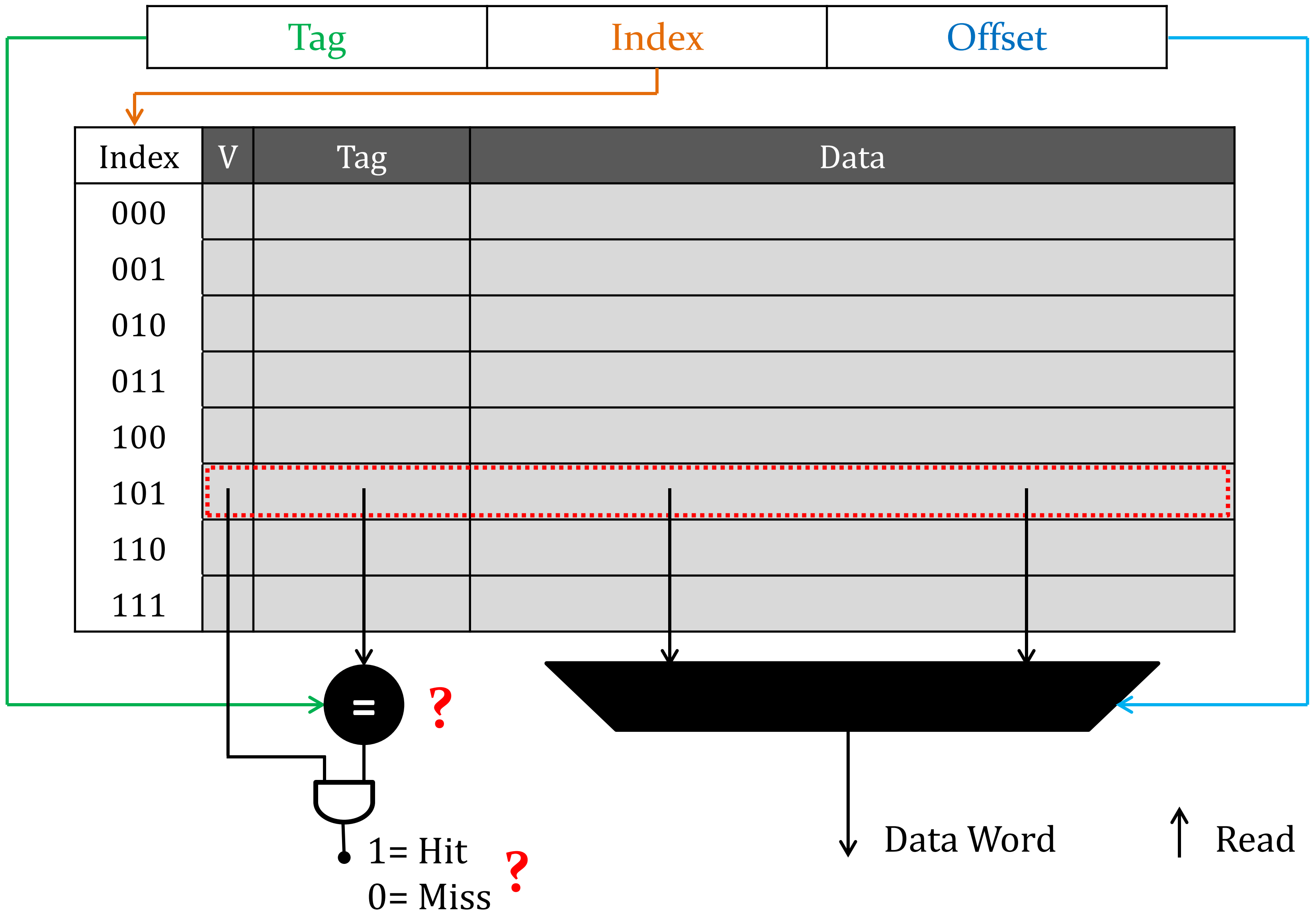
Index	Data Block
000	
001	
010	
011	
100	
101	?
110	
111	

DIRECT MAPPED CACHE: STRUCTURE

Index	V	Tag	Data
000			
001			
010			
011			
100			
101			
110			
111			

V – Valid Bit

Direct Mapped Cache: Read Accesses



DIRECT MAPPED CACHE: READ ACCESSES

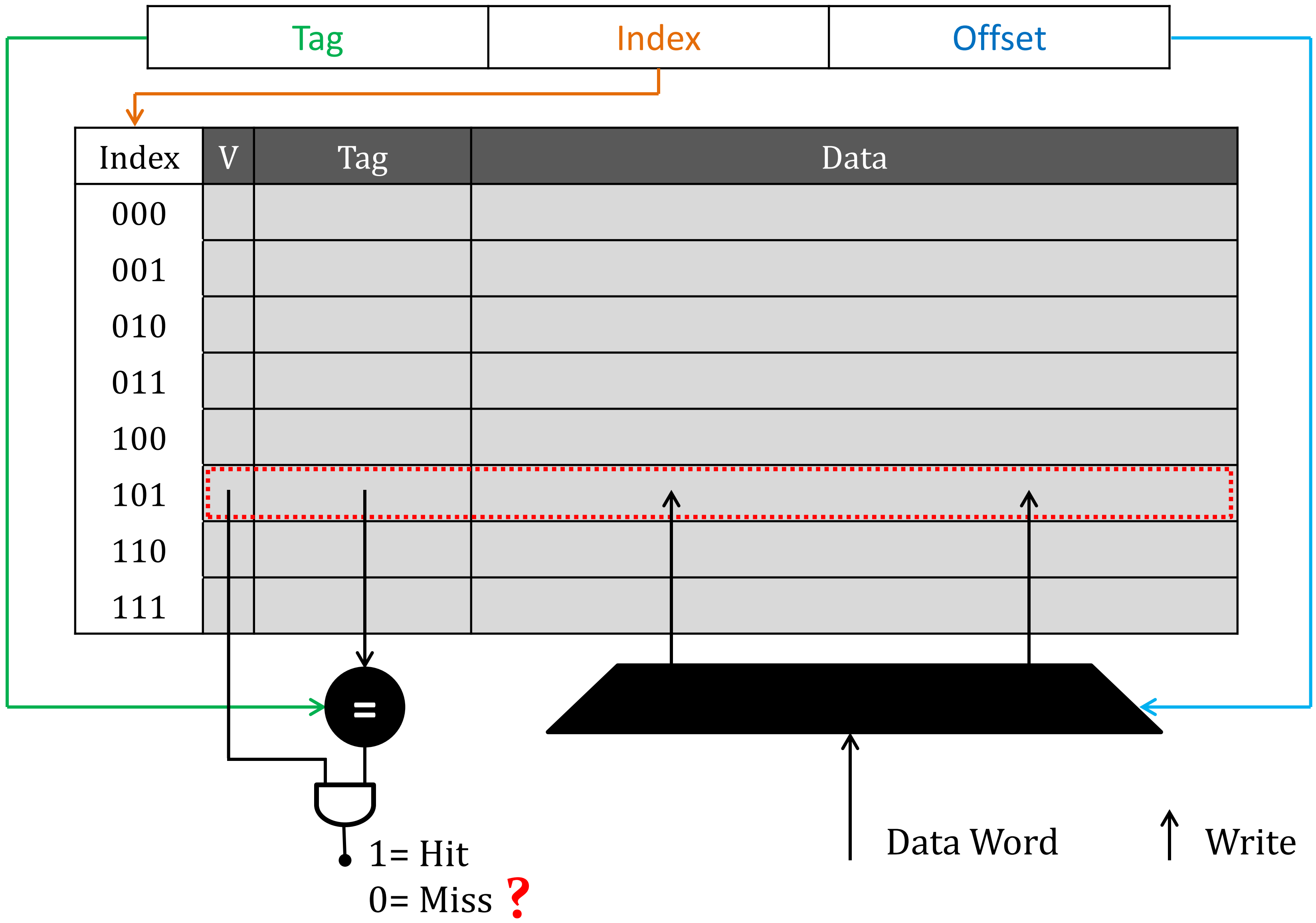
On a read-hit:

Send the data to the CPU

On a read-miss:

- Stall the CPU (need to send a signal to the CPU)
- Read the **block** from main memory (wait till memory responds)
- Update the cache entry: What about the old block ?
  - Update the data block of the entry
  - Update the tag field of the entry
  - Set the valid bit of the entry
- Send the data to the CPU & clear the stall signal

Direct Mapped Cache: Write Accesses



DIRECT MAPPED CACHE: WRITE ACCESSES

On a write-hit:

Block in cache is now different to the one in memory (*inconsistent*)

Solutions?

Simplest: always write to both cache and memory (**write-through**)

On a write-miss:

- Stall the CPU (need to send a signal to the CPU)
- Read the **block** from main memory (wait till memory responds)
- Update the cache entry
- Write the new data word to the cache entry
- Write the new data word to the memory

What about the old block ?

Okay to discard?
Yes!
(with *write-through*)

Writing and tag comparison latencies can overlap in the same clock cycle.

DIRECT MAPPED CACHE: WRITE POLICIES

Write-through:

- Simple to implement, but generates a lot of write traffic to memory
- Not very efficient when the program/CPU writes a lot
- Stalls the CPU on every write!
- A memory access: about 10-100 CPU clock cycles!
- Can use a *write-buffer*, and let the CPU carry on without stalling

Write-through works alright when the rate of write accesses are slow enough for the memory to handle

Write-buffer can help with small bursts of writes

Direct Mapped Cache: Write Policies

Write-back:

- **On a write-hit:** Update the cache entry (*inconsistent* with mem)
- **On a read-miss:** Write-back the old block (if it's **dirty**) before fetching the new block from memory
- **On a write-miss:** Write-back the old block (if it's **dirty**), fetch new block, and update only the cache entry

Write-back controller is complex to implement

But more efficient in general, specially when the rate of write accesses are too fast for the memory to handle

Write-buffer can be used for dirty blocks being written back

Direct Mapped Cache: Write-Back

Index	V	D	Tag	Data
000				
001				
010				
011				
100				
101				
110				
111				

D – Dirty Bit

CACHE PERFORMANCE

Average Access Time for a cache:

$$T_{avg} = HL + (1 - HR) \times MP$$

T_{avg}, *HL*, and *MP* may be expressed using <u>time</u> or <u>clock cycles</u>

Assume:

- MP = 20 CPU cycles
- HR = 95% (0.95) $\longrightarrow$ HR = 99.9% (0.999)
- HL = 1 CPU cycle
- CPU clock period = 1 ns

$$T_{avg} = 1 + (1 - 0.95) \times 20$$
$$= 2\ cycles\ (2\ ns)$$

$$T_{avg} = 1 + (1 - 0.999) \times 20$$
$$= 1.02\ cycles\ (\sim 1\ n$$

Cache Performance

36% of the instructions of a given program are loads and stores. CPI of the processor is 2, assuming ideal caches (100% hits, no stalls).

In reality, the actual instruction and data caches demonstrate 2% and 4% miss rates respectively for this program on this processor. Read/write miss penalties are 100 CPU cycles each, and the caches take 1 CPU cycle to determine hit status.

1. What is the actual CPI of the processor?
2. How much faster would the processor work if ideal caches were available, as opposed to using the real caches?
3. How much slower would the processor work if no caches are used, as opposed to using the real caches?

Cache Performance

36% of the instructions of a given program are loads and stores. CPI of the processor is 2, assuming perfect caches (100% hits, no stalls).

In reality, the actual instruction and data caches demonstrate 2% and 4% miss rates respectively for this program on this processor. Read/write miss penalties are 100 CPU cycles each, and the caches take 1 CPU cycle to determine hit status.

1. What is the actual CPI of the processor?

Assume the program has *i* instructions in total.

Cycles stalled due to instruction cache misses = $i \times 2\% \times 100 = 2i$

Cycles stalled due to data cache misses = $i \times 36\% \times 4\% \times 100 = 1.44i$

Total stall cycles = $3.44i$

$$\therefore \text{Actual CPI} = 2 + \frac{3.44i}{i} = 5.44$$

CACHE PERFORMANCE

36% of the instructions of a given program are loads and stores. CPI of the processor is 2, assuming perfect caches (100% hits, no stalls).

In reality, the actual instruction and data caches demonstrate 2% and 4% miss rates respectively for this program on this processor. Read/write miss penalties are 100 CPU cycles each, and the caches take 1 CPU cycle to determine hit status.

2. How much faster would the processor work if perfect caches were available, as opposed to using the real caches?

$$\text{Ideal speedup} = \frac{5.44}{2} = 2.72$$

Cache Performance

36% of the instructions of a given program are loads and stores. CPI of the processor is 2, assuming perfect caches (100% hits, no stalls).

In reality, the actual instruction and data caches demonstrate 2% and 4% miss rates respectively for this program on this processor. Read/write miss penalties are 100 CPU cycles each, and the caches take 1 CPU cycle to determine hit status.

3. How much slower would the processor work if no caches are used, as opposed to using the real caches?

$$\text{No cache CPI} = 2 + \frac{100i + 36i}{i} = 138$$

$$\therefore \text{Slowdown} = \frac{138}{5.44} = 25.37$$

Improve cache performance?
HR / HL / MP
Bigger cache $\rightarrow$ HR $\uparrow$

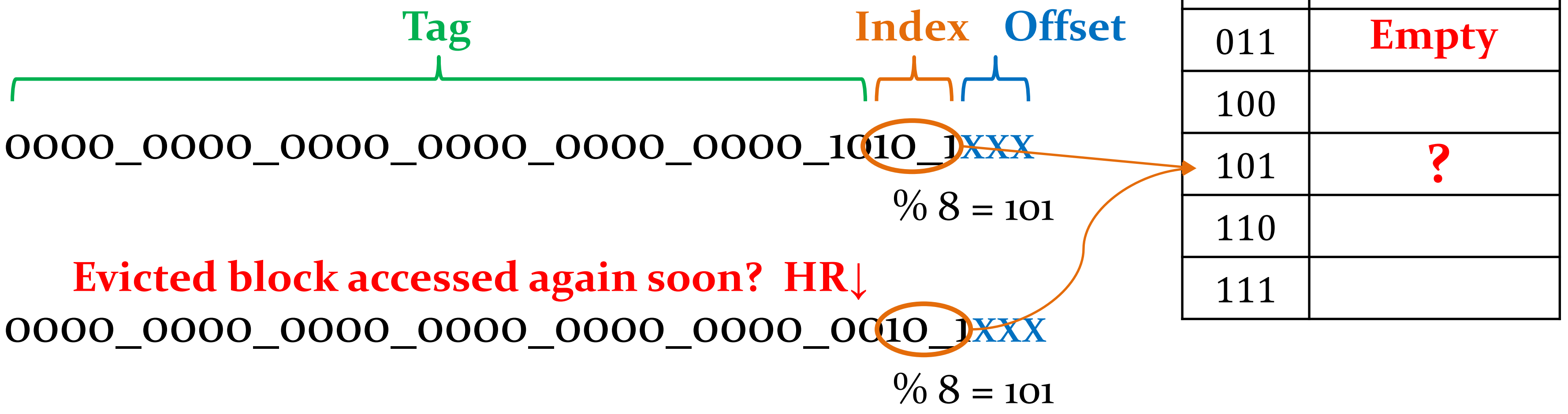
Recall: Direct Mapped Cache

$$\text{Location in cache} = \text{Memory block–address} \ \% \ \text{No. of blocks in cache}$$

Index

- Assume **block-size = 8 bytes** and cache can hold **8 blocks**

Cache consists of Data Array + Tag Array


Tag
Index
Offset
0000_0000_0000_0000_0000_0000_1010_1XXX
% 8 = 101
Evicted block accessed again soon? HR↓
0000_0000_0000_0000_0000_0000_0010_1XXX
% 8 = 101

Index	Data Block
000	
001	
010	
011	Empty
100	
101	?
110	
111	

What if we remove the "direct mapped" concept?

$$\cancel{\begin{matrix}\textit{Location}\\ \textit{in cache}\end{matrix}} = \begin{matrix}\textit{Memory}\\ \textit{block–address}\end{matrix} \;\%\; \begin{matrix}\textit{No. of blocks}\\ \textit{in cache}\end{matrix}$$

Index

- Assume **block-size = 8 bytes** and cache can hold **8 blocks**

Cache consists of Data Array + Tag Array

~~Index~~	Data Block
~~000~~	
~~001~~	
~~010~~	
~~011~~	**Empty**
~~100~~	
~~101~~	**?**
~~110~~	
~~111~~	

Tag **~~Index~~** **Offset**

0000_0000_0000_0000_0000_0000_1010_1XXX

% 8 = 101

Evicted block accessed again soon? HR↓

0000_0000_0000_0000_0000_0000_0010_1XXX

% 8 = 101

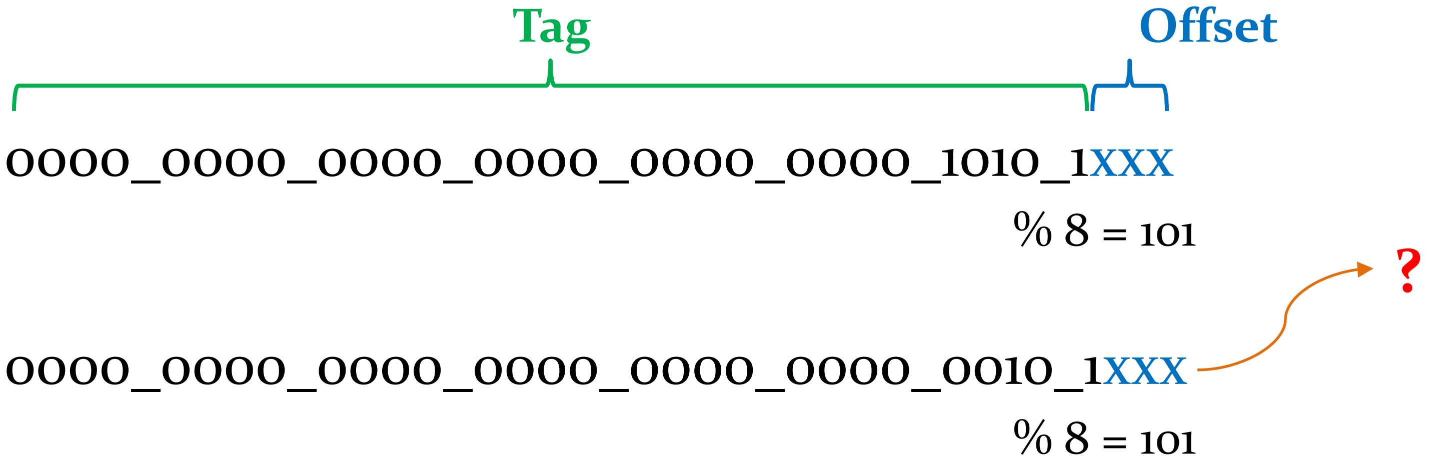
CACHE MEMORY

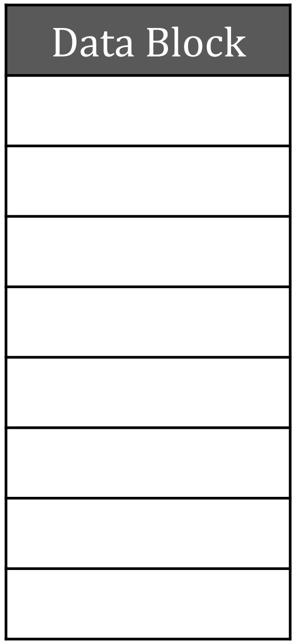
Fully Associative Cache (a block can be mapped anywhere, how to find)

- Need to search for the tag: *~~Sequential search?~~*
- Need comparator per entry! *Parallel search!* (practical only for small number of entries)

- Assume **block-size = 8 bytes** and cache can hold **8 blocks**

Cache consists of Data Array + Tag Array


Tag
Offset
0000_0000_0000_0000_0000_0000_1010_1XXX
% 8 = 101
0000_0000_0000_0000_0000_0000_0010_1XXX
% 8 = 101
?



CACHE MEMORY

Fully Associative Cache (a block can be mapped anywhere, how to find)

- Need to search for the tag: *~~Sequential search?~~*
- Need comparator per entry! *Parallel search!*
  (practical only for small number of entries)

Where to place a new block?

- Circuitry to find the first available invalid entry (sequential ☹)
- No invalid entries?

How to evict an existing valid block?

- **Replacement policy**:
  1. Least Recently Used (LRU)
  2. Pseudo Least Recently Used (PLRU)
  3. First-In First-Out (FIFO)

CACHE MEMORY

Fully Associative Cache

What we like:

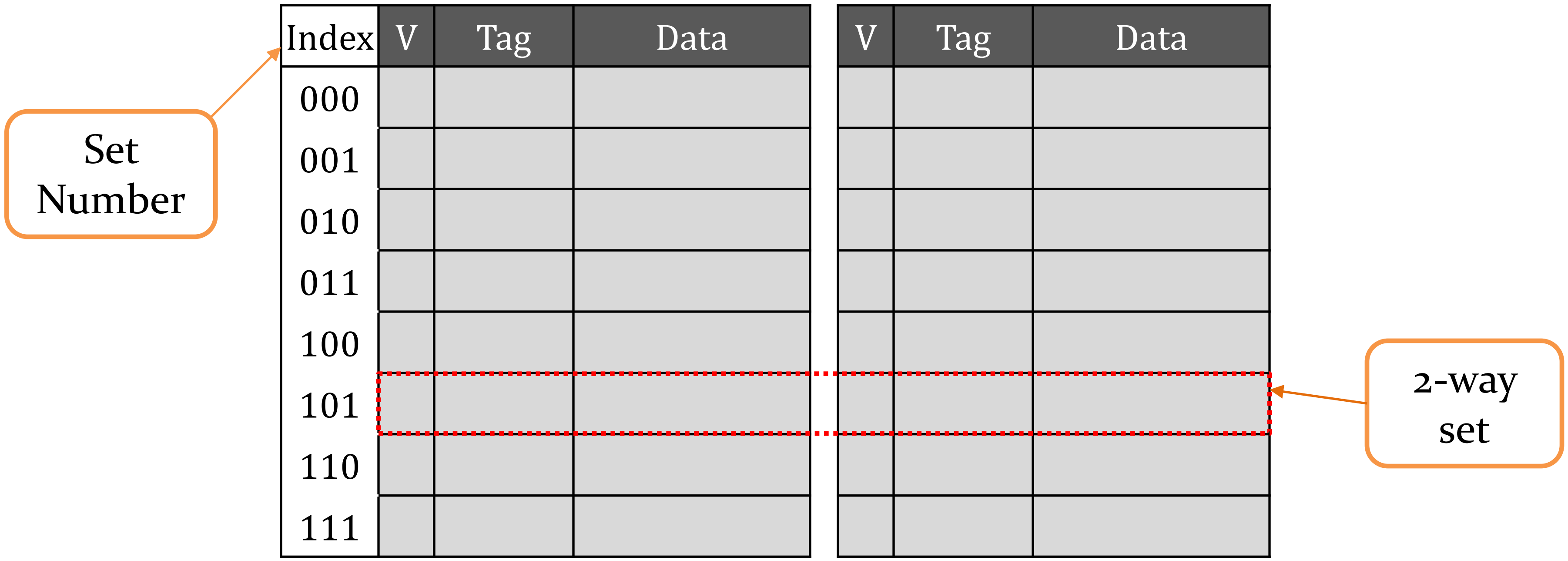
- High utilization of cache space, HR ↑
- Replacement Policy as needed (high performance, low cost/power)

What we don't like:

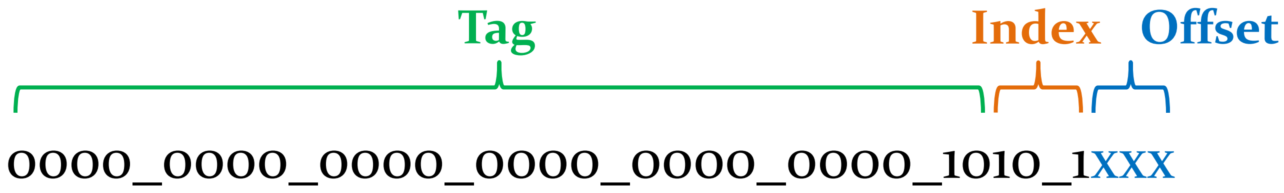
- Slow block placement (searching for an invalid entry)
- High power consumption by placement circuitry
- High cost, due to all the added hardware
  (placement, replacement, tag compare, etc.)

Set Associative Cache

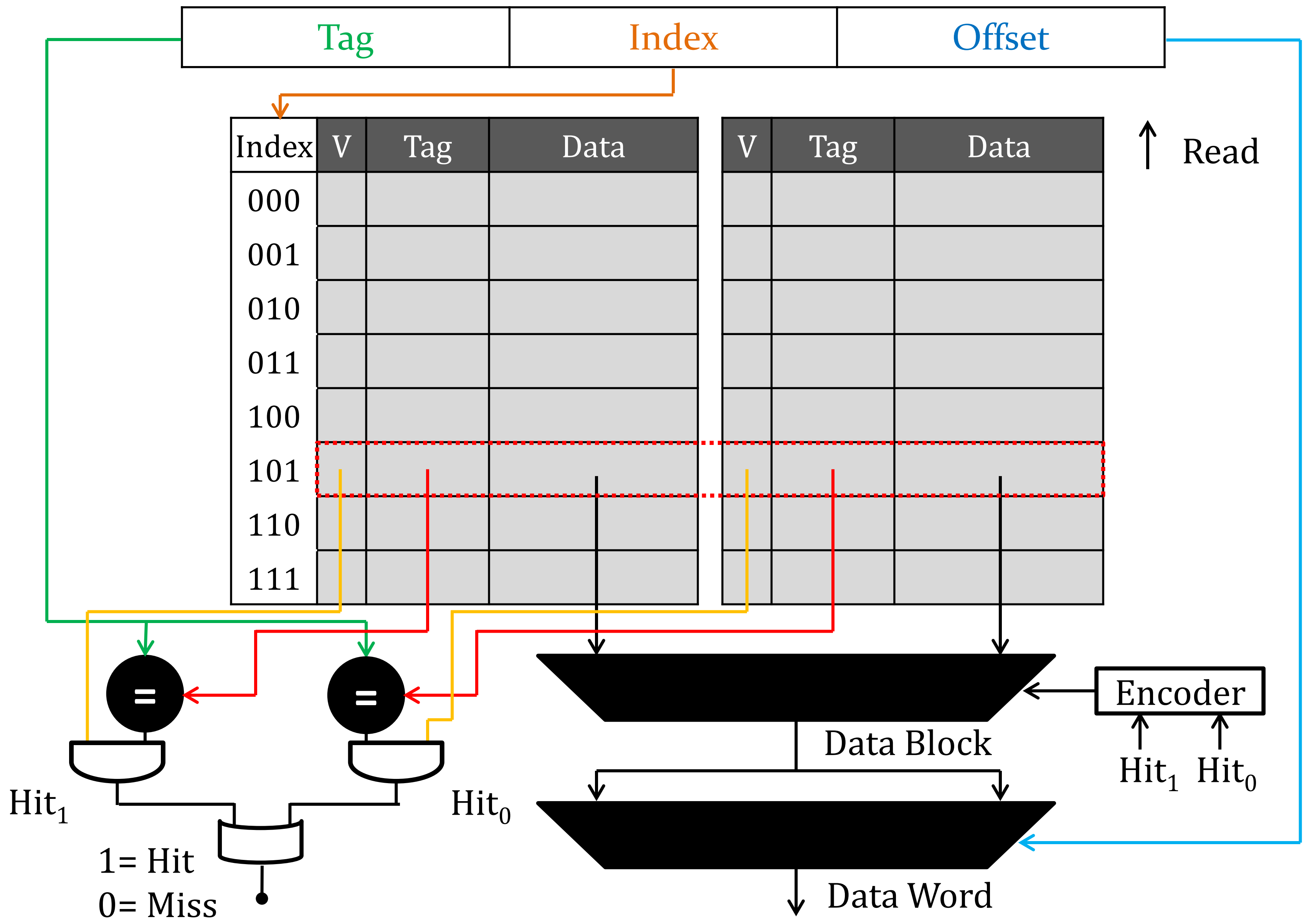
2-way Set Associative Cache



Assume **block-size = 8 bytes.** This cache can hold **16 blocks**



Set Associative Cache: Read Accesses



Set Associative Cache

Associativity: organization of an 8-block cache

Index	V	T	D
000			
001			
010			
011			
100			
101			
110			
111			

1-way set associative
(direct mapped)

Index	V	T	D	V	T	D
00						
01						
10						
11						

2-way set associative

Index	V	T	D	V	T	D	V	T	D	V	T	D
0												
1												

4-way set associative

V	T	D	V	T	D	V	T	D	V	T	D	V	T	D	V	T	D	V	T	D	V	T	D

8-way set associative
(fully associative)

Which one is the best?

SET ASSOCIATIVE CACHE

Comparison: 4-block caches

(block-size = 1 word, word-size = 4 bytes, 8-bit addresses)

000000xx

Miss

Index	V	Tag	Data
00	0	0000	?
01	0	0000	?
10	0	0000	?
11	0	0000	?

Direct mapped

Miss

Index	V	T	D	V	T	D
0	0	00000	?	0	00000	?
1	0	00000	?	0	00000	?

2-way set associative

Miss

V	T	D	V	T	D	V	T	D	V	T	D
0	000000	?	0	000000	?	0	000000	?	0	000000	?

Fully associative (4-way)

SET ASSOCIATIVE CACHE

Comparison: 4-block caches

(block-size = 1 word, word-size = 4 bytes, 8-bit addresses)

000000xx
001000xx

Miss Hits: 0 Misses: 1

Index	V	Tag	Data
00	1	0000	MEM[0]
01	0	0000	?
10	0	0000	?
11	0	0000	?

Direct mapped

Miss Hits: 0 Misses: 1

Index	V	T	D	V	T	D
0	1	00000	MEM[0]	0	00000	?
1	0	00000	?	0	00000	?

2-way set associative

Miss Hits: 0 Misses: 1

V	T	D	V	T	D	V	T	D	V	T	D
1	000000	MEM[0]	0	000000	?	0	000000	?	0	000000	?

Fully associative (4-way)

Set Associative Cache

Comparison: 4-block caches

(block-size = 1 word, word-size = 4 bytes, 8-bit addresses)

000000xx
001000xx
000000xx

Miss Hits: 0 Misses: 2

Index	V	Tag	Data
00	1	0010	MEM[8]
01	0	0000	?
10	0	0000	?
11	0	0000	?

Direct mapped

Hit Hits: 0 Misses: 2

Index	V	T	D	V	T	D
0	1	00000	MEM[0]	1	00100	MEM[8]
1	0	00000	?	0	00000	?

2-way set associative

Hit Hits: 0 Misses: 2

V	T	D	V	T	D	V	T	D	V	T	D
1	000000	MEM[0]	1	001000	MEM[8]	0	000000	?	0	000000	?

Fully associative (4-way)

Set Associative Cache

Comparison: 4-block caches

(block-size = 1 word, word-size = 4 bytes, 8-bit addresses)

000000xx
001000xx
000000xx
000110xx

Miss Hits: 0 Misses: 3

Index	V	Tag	Data
00	1	0000	MEM[0]
01	0	0000	?
10	0	0000	?
11	0	0000	?

Direct mapped

Miss Hits: 1 Misses: 2

Index	V	T	D	V	T	D
0	1	00000	MEM[0]	1	00100	MEM[8]
1	0	00000	?	0	00000	?

?

2-way set associative

Miss Hits: 1 Misses: 2

V	T	D	V	T	D	V	T	D	V	T	D
1	000000	MEM[0]	1	001000	MEM[8]	0	000000	?	0	000000	?

Fully associative (4-way)

Set Associative Cache

Comparison: 4-block caches

(block-size = 1 word, word-size = 4 bytes, 8-bit addresses)

000000xx
001000xx
000000xx
000110xx
001000xx

Miss Hits: 0 Misses: 4

Index	V	Tag	Data
00	1	0000	MEM[0]
01	0	0000	?
10	1	0001	MEM[6]
11	0	0000	?

Direct mapped

Miss Hits: 1 Misses: 3

Index	V	T	D	V	T	D
0	1	00000	MEM[0]	1	00011	MEM[6]
1	0	00000	?	0	00000	?

?

2-way set associative

Hit Hits: 1 Misses: 3

V	T	D	V	T	D	V	T	D	V	T	D
1	000000	MEM[0]	1	001000	MEM[8]	1	000110	MEM[6]	0	000000	?

Fully associative (4-way)

SET ASSOCIATIVE CACHE

Comparison: 4-block caches

(block-size = 1 word, word-size = 4 bytes, 8-bit addresses)

000000xx
001000xx
000000xx
000110xx
001000xx

- Higher associativity:
  HR ↑ ☺
  HL ↑ ☹

Hits: 0 Misses: 5

Index	V	Tag	Data
00	1	0010	MEM[8]
01	0	0000	?
10	1	0001	MEM[6]
11	0	0000	?

Direct mapped

Hits: 1 Misses: 4

Index	V	T	D	V	T	D
0	1	00100	MEM[8]	1	00011	MEM[6]
1	0	00000	?	0	00000	?

2-way set associative

Hits: 2 Misses: 3

V	T	D	V	T	D	V	T	D	V	T	D
1	000000	MEM[0]	1	001000	MEM[8]	1	000110	MEM[6]	0	000000	?

Fully associative (4-way)

The Configuration of a Cache

Parameters:

- Block-size
- Set-size (<u>number</u> of sets)
- Associativity (number of <u>ways</u> in a set)

- Replacement policy
- Write policy
- Many other factors…

Improving Performance

Improving HR:

- Bigger size (number of blocks the cache can hold) $\rightarrow$ cost $\uparrow\uparrow$
- Higher associativity $\rightarrow$ HL $\uparrow$ + cost $\uparrow$ + power $\uparrow$ compared to DM
- Cache prefetching
  (to reduce cold/compulsory misses as well as conflict misses)

Improving MP:

- Optimize the communication between cache and memory
- Add lower level caches!

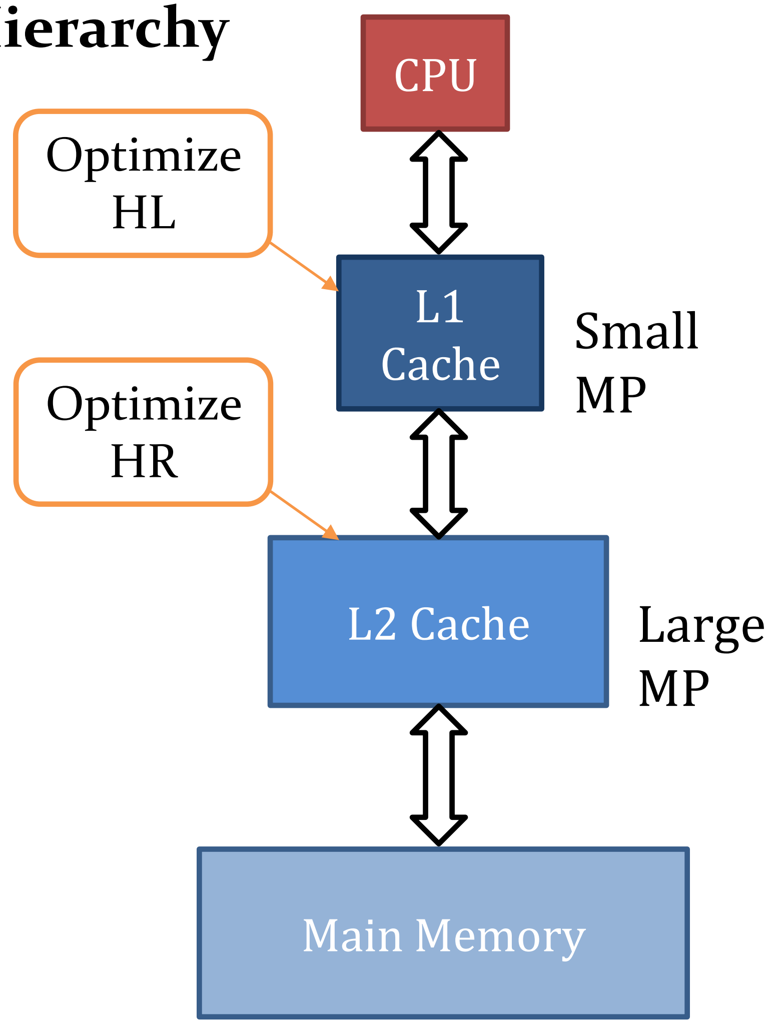
MULTI LEVEL CACHES

Cache Hierarchy within Memory Hierarchy

- Reduce the effective miss penalty

$$Avg.MP_{L_1} = HL_{L_2} + (1 - HR_{L_2}) \times MP_{L_2}$$

- Why not have one big cache?
- Reduce CPU cycle time, while minimizing the stall cycles
- L1 is on-chip (inside CPU)
- L2 is usually on-chip
  can be off-chip in some designs
- L3 is usually off-chip



Example on page 487 COD   www.wikichip.org

VIRTUAL MEMORY

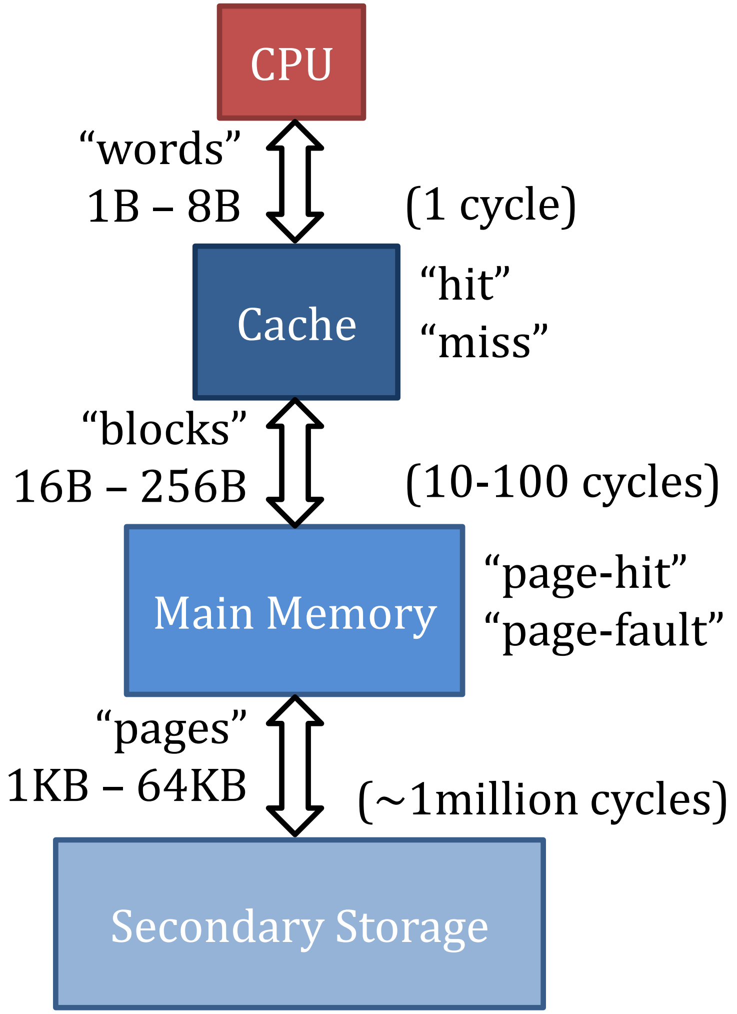
CPU Word-Size	Address Space Size
64-bit	16 Exabytes
32-bit	4 Gigabytes
16-bit	64 Kilobytes
8-bit	256 Bytes

- We assumed that main memory contains a slot for every Byte-address CPU/Program can generate.
- In reality, actual RAM sizes are smaller than the AS size.
- Virtual Memory allows programs to use more memory than available. How?
- By using main memory as a “cache” for secondary storage (disk)!
- VM also facilitates multiple programs, with safe and efficient memory sharing.

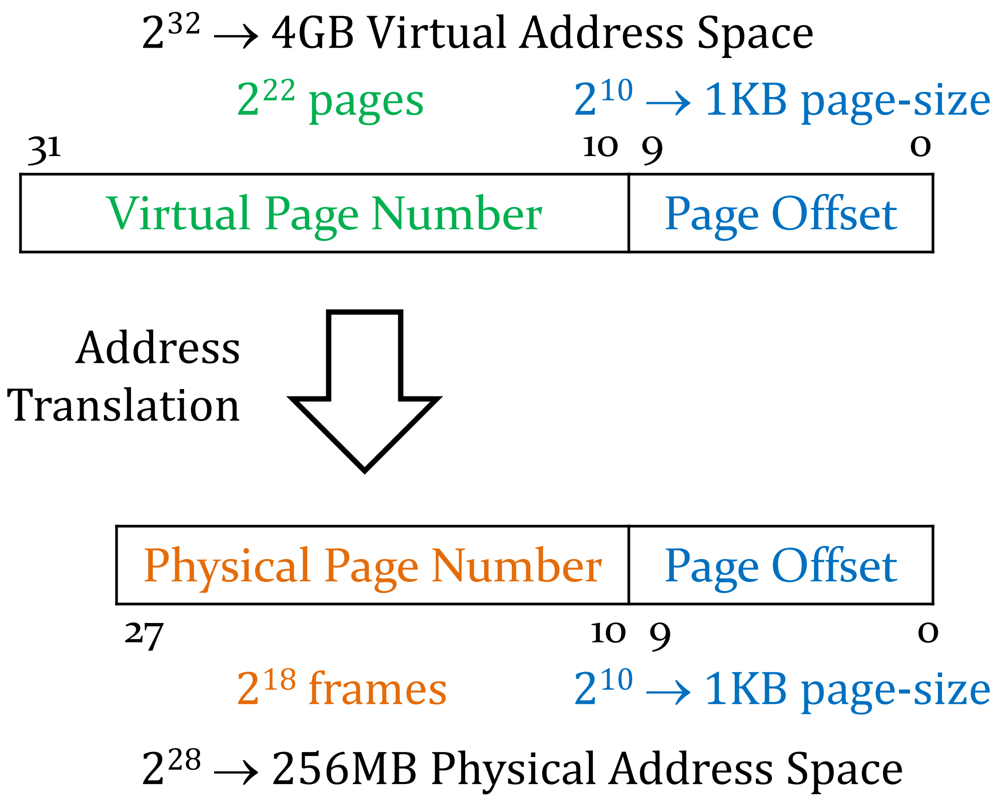
Virtual Memory

How does addressing work then?
Can we have tag-compare in memory?!?

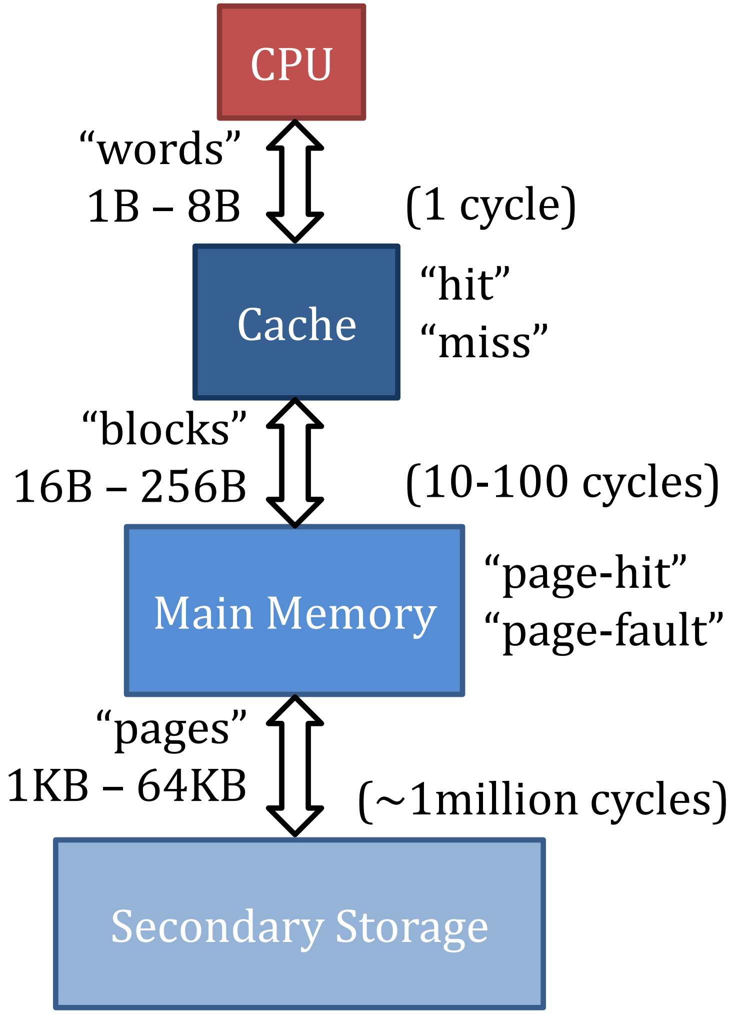
- CPU produces a **virtual address** from the full address space (remember, CPU has no idea about the memory hierarchy!)
- Memory is accessed using a **physical address** (in the range of the actual memory size)
- An **address translation** needs to be done whenever memory is being accessed.



Virtual Memory

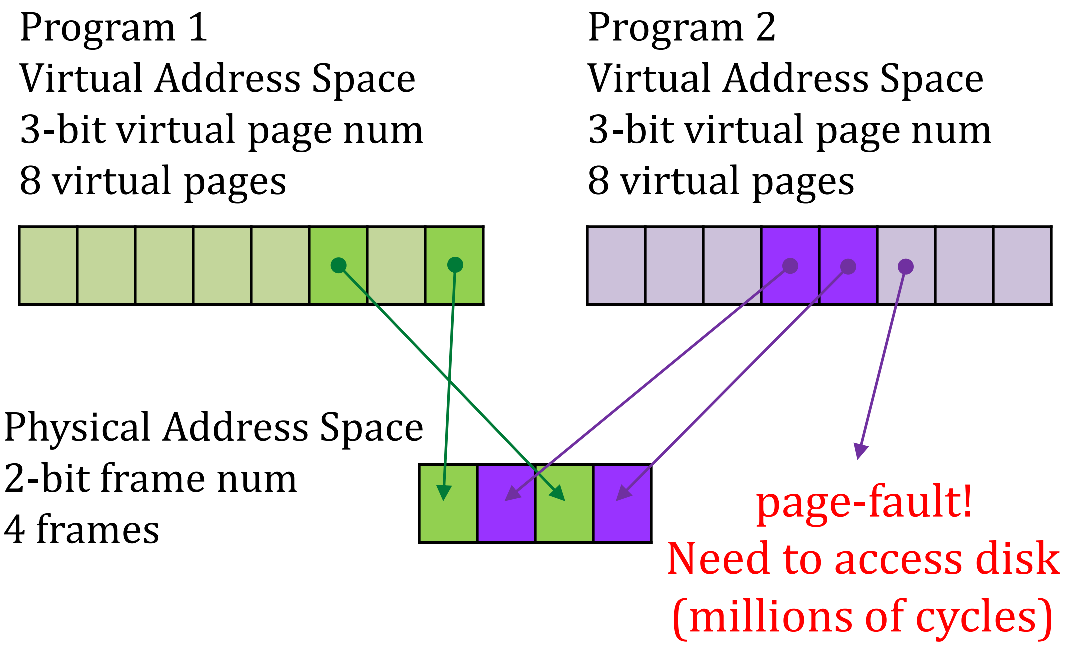

$2^{32} \rightarrow$ 4GB Virtual Address Space
2^{22} pages
$2^{10} \rightarrow$ 1KB page-size
31
10 9
0
Virtual Page Number
Page Offset
Address Translation
Physical Page Number
Page Offset
27
10 9
0
2^{18} frames
$2^{10} \rightarrow$ 1KB page-size
$2^{28} \rightarrow$ 256MB Physical Address Space

physical page number = frame number

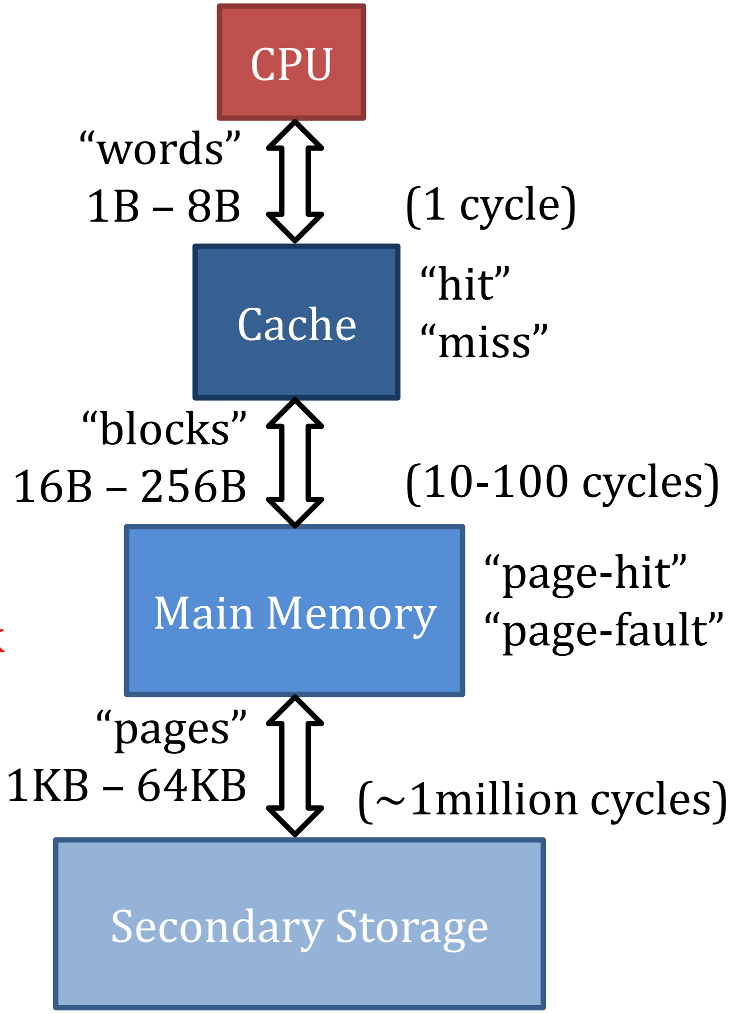


Virtual Memory

How are multiple programs supported?

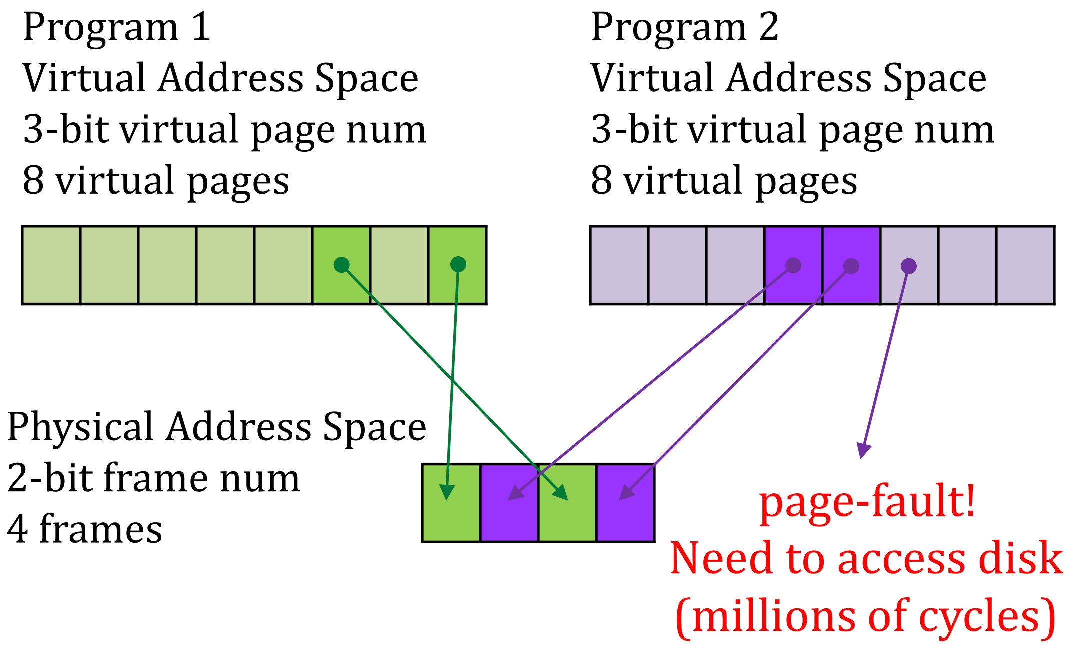


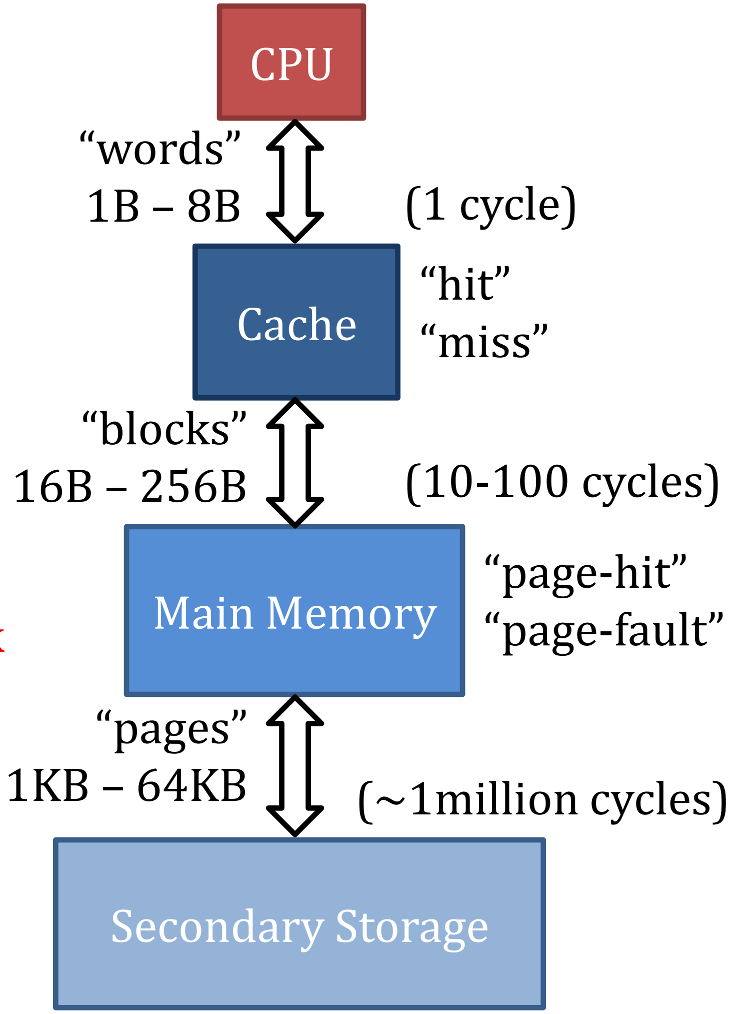
- Page faults are handled in software, by the operating system
- Address translation done in by OS with hardware support



VIRTUAL MEMORY

How are multiple programs supported?



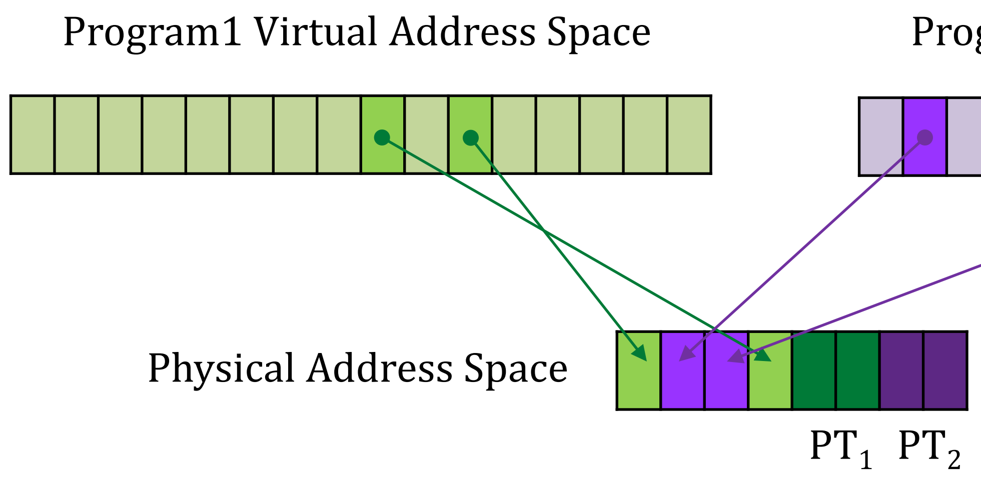


- VM typically uses *write-back* policy
- Minimize page-faults → fully associative page placement in memory
- How to find a page in memory? (how is address translation done?)

Virtual Memory

Page Table

A section in the memory which stores the address translations

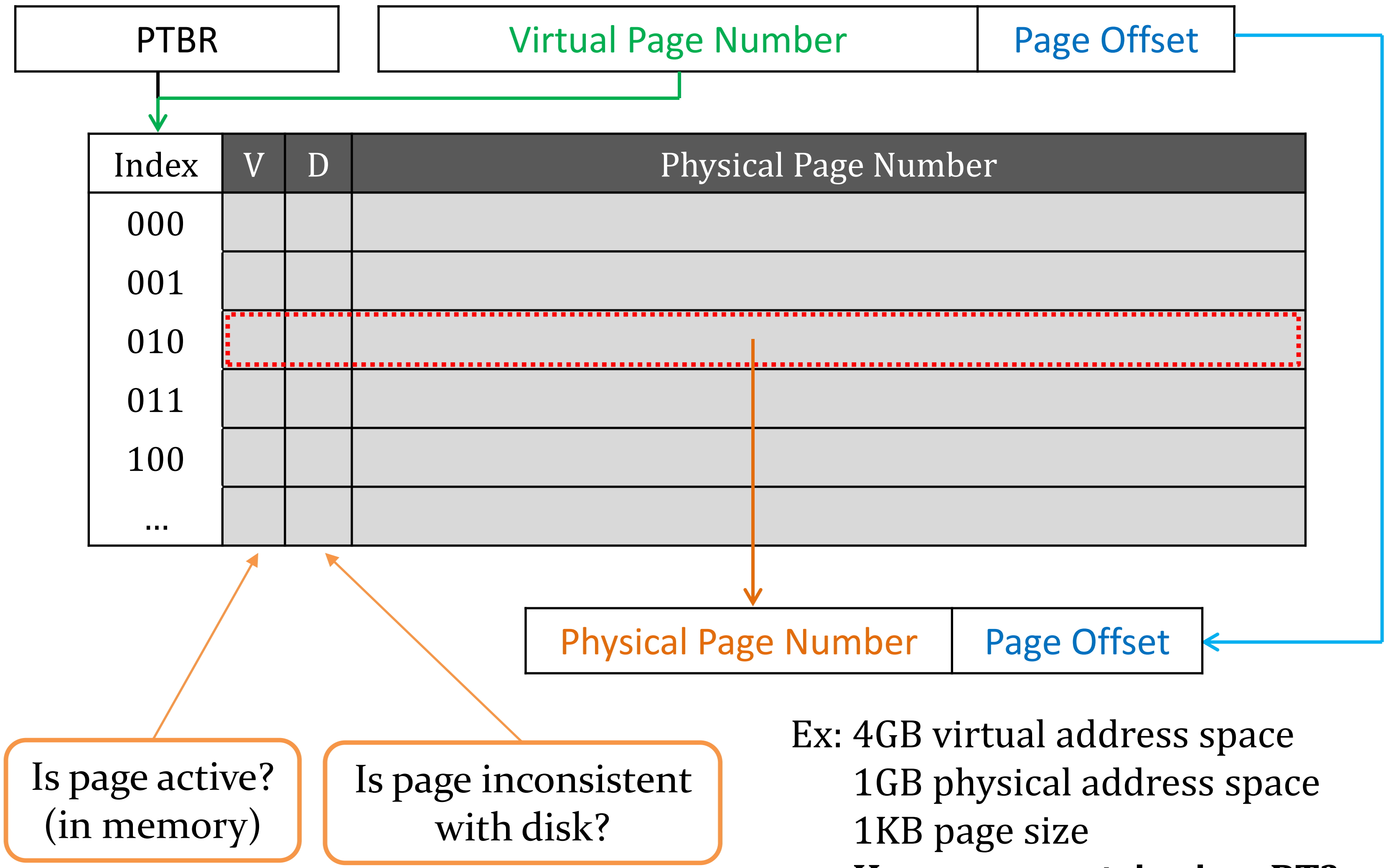

Program1 Virtual Address Space
Program 2 Virtual Address Space
Physical Address Space
PT_1
PT_2

Why program specific PTs?

- When a program/CPU wants to access memory, access PT first to get the address translated (2 memory accesses)
- Okay, but how to find PT ?!?
- Place it at a <u>fixed location</u>, known to CPU+OS

  PTBR (Page Table Base Register) in CPU $\rightarrow$ managed by OS

VIRTUAL MEMORY: PAGE TABLE



Ex: 4GB virtual address space
1GB physical address space
1KB page size
How many entries in a PT?
Size of a PT?

Virtual Memory

Page Faults

Entry in PT is invalid (page is inactive, not in memory)
Page faults are handled by the OS

- Fetch the page from disk (OS keeps track of disk locations)
- Find an unused frame in the physical memory (OS keeps track of utilization)
- If all frames are used, find an active page to replace:
  - Replacement policy (LRU/FIFO/LFU/etc.)
  - OS keeps track of page usage
  - Write-back replaced page if it's dirty
- Learn more about Page Handling in CO327

Translation Look-aside Buffer (TLB)

Can we avoid accessing memory for address translation?
Use a hardware cache for the PT!

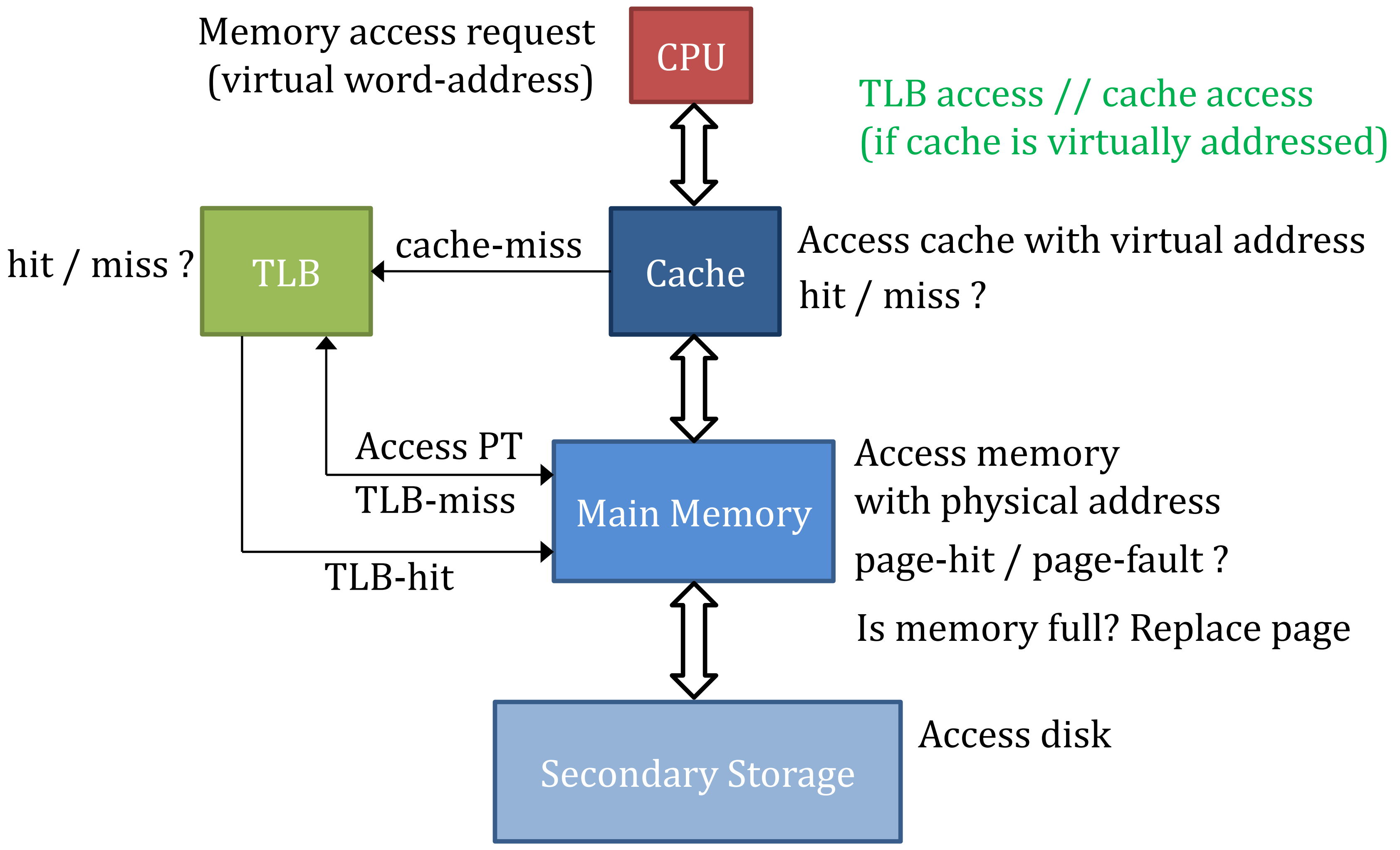
- Based on locality of PT entries

Is TLB entry valid?

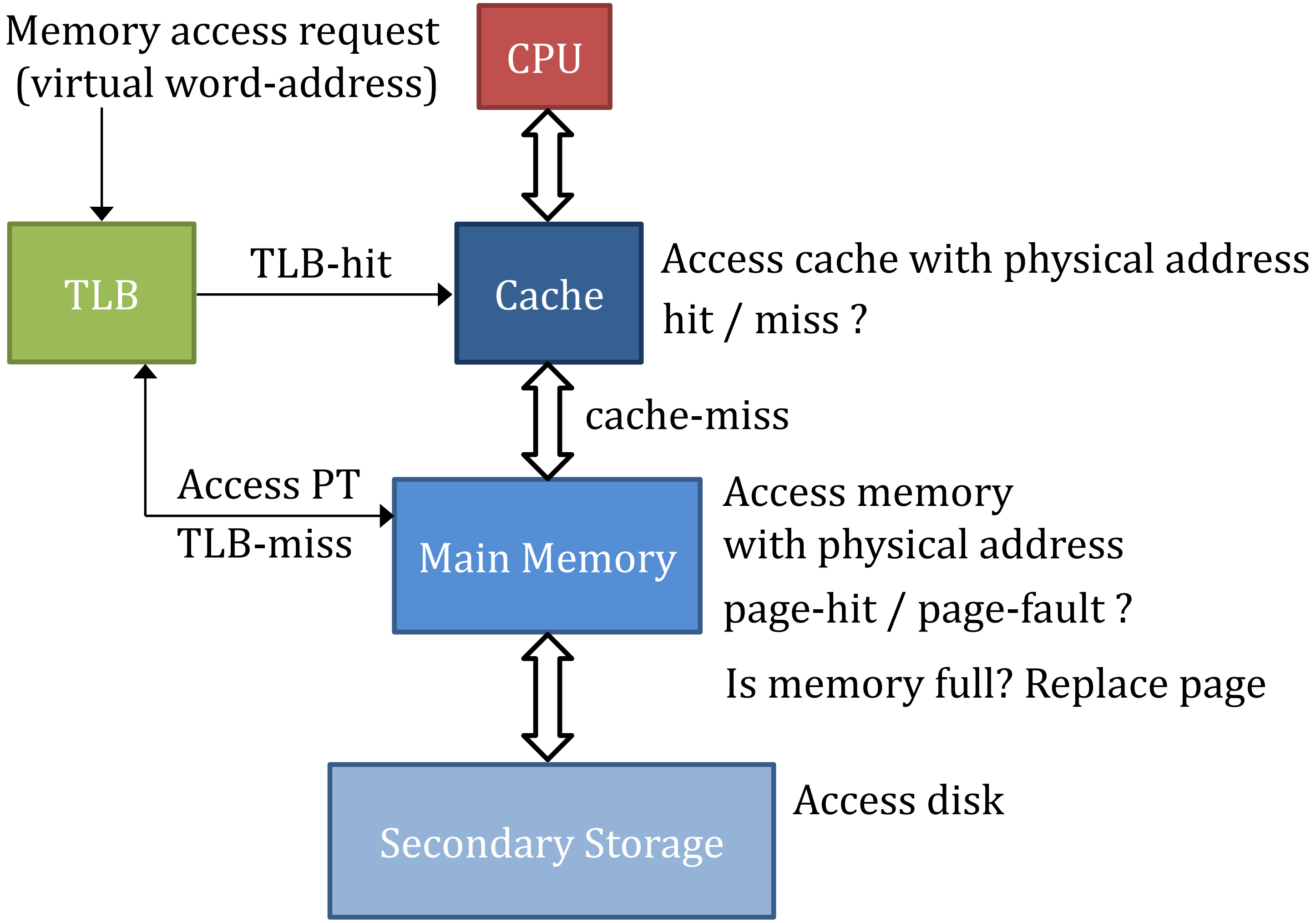
V	D	Tag	Physical Page Number

- Size: around 16 – 512 PT entries (address translations)
- Block-size: 1 – 2
- Placement: set/fully-associative (miss rate < 1 %)
- Hit latency: < 1 cycle
- Miss penalty: 10-100 cycles

TLB: Big Picture



TLB: BIG PICTURE


Memory access request
(virtual word-address)
CPU
TLB
TLB-hit
Cache
Access cache with physical address
hit / miss ?
cache-miss
Access PT
TLB-miss
Main Memory
Access memory
with physical address
page-hit / page-fault ?
Is memory full? Replace page
Secondary Storage
Access disk