Chapter 5

Large and Fast: Exploiting Memory Hierarchy

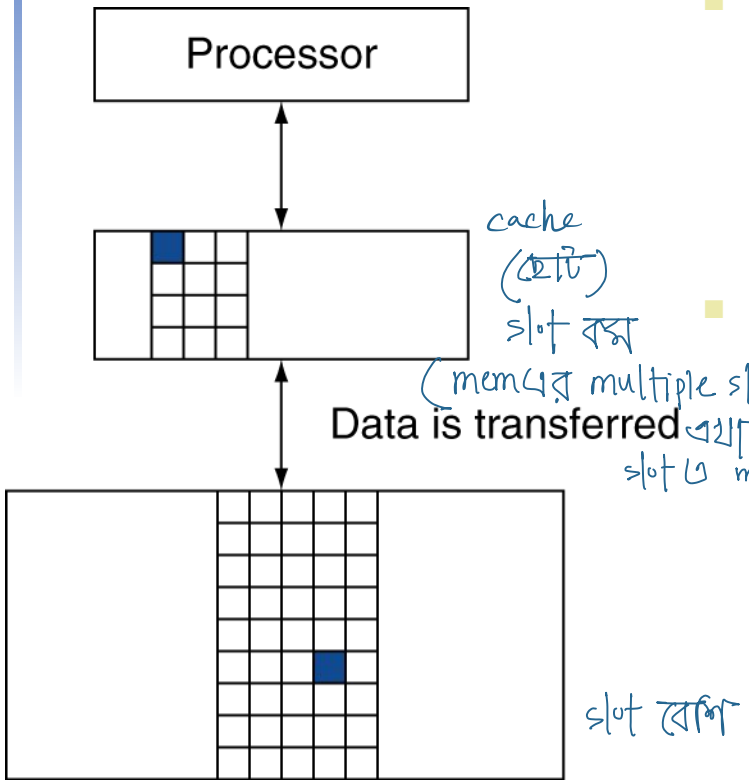
Principle of Locality

- Programs access a small proportion of their address space at any time
- Temporal locality
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, induction variables
- Spatial locality
  - Items near those accessed recently are likely to be accessed soon
  - E.g., sequential instruction access, array data

Taking Advantage of Locality

- Memory hierarchy
- Store everything on disk
- Copy recently accessed (and nearby) items from disk to smaller DRAM memory
  - Main memory
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
  - Cache memory attached to CPU

Memory Hierarchy Levels



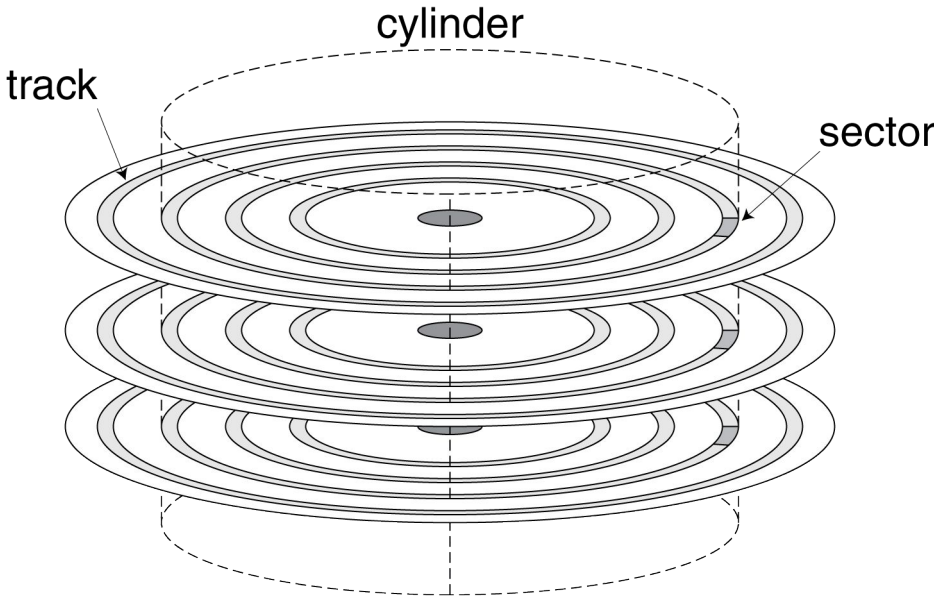
- Block (aka line): unit of copying
  - May be multiple words
- If accessed data is present in upper level
  - Hit: access satisfied by upper level
    - Hit ratio: hits/accesses
- If accessed data is absent
  - Miss: block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses = 1 – hit ratio
  - Then accessed data supplied from upper level

Memory Technology

- Static RAM (SRAM)
  - 0.5ns – 2.5ns, $2000 – $5000 per GB
- Dynamic RAM (DRAM)
  - 50ns – 70ns, $20 – $75 per GB
- Magnetic disk
  - 5ms – 20ms, $0.20 – $2 per GB
- Ideal memory
  - Access time of SRAM
  - Capacity and cost/GB of disk

Disk Storage

- Nonvolatile, rotating magnetic storage



Disk Sectors and Access

- Each sector records
  - Sector ID
  - Data (512 bytes, 4096 bytes proposed)
  - Error correcting code (ECC)
    - Used to hide defects and recording errors
  - Synchronization fields and gaps
- Access to a sector involves
  - Queuing delay if other accesses are pending
  - Seek: move the heads
  - Rotational latency
  - Data transfer
  - Controller overhead

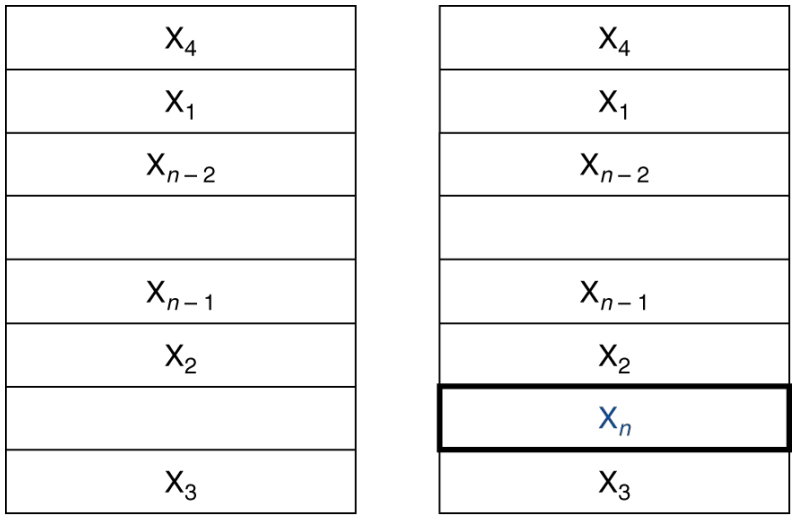
Disk Access Example

- Given
  - 512B sector, 15,000rpm, 4ms average seek time, 100MB/s transfer rate, 0.2ms controller overhead, idle disk
- Average read time
  - 4ms seek time
    + ½ / (15,000/60) = 2ms rotational latency
    + 512 / 100MB/s = 0.005ms transfer time
    + 0.2ms controller delay
    = 6.2ms
- If actual average seek time is 1ms
  - Average read time = 3.2ms

• seektime minimum 로 hashing 함

Cache Memory

- Cache memory
  - The level of the memory hierarchy closest to the CPU
- Given accesses X_1, …, X_{n-1}, X_n

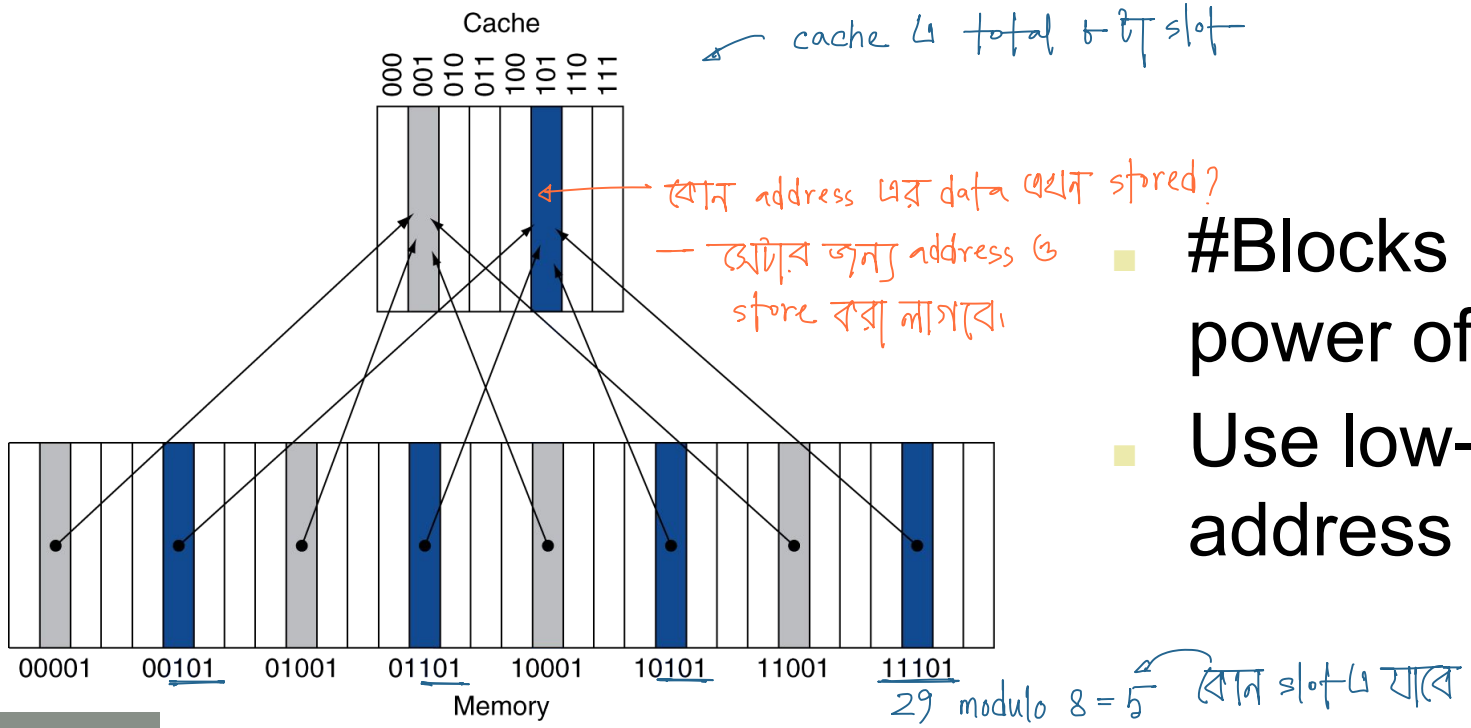


a. Before the reference to X_n b. After the reference to X_n

- How do we know if the data is present?
- Where do we look?

Direct Mapped Cache

- Location determined by address
- Direct mapped: only one choice
  - (Block address) modulo (#Blocks in cache)



- #Blocks is a power of 2
- Use low-order address bits

minimum time এ modulo বের করা লাগবে যেন এটার জন্য clock cycle না বাড়ে।

- ALU use করবো

$x \% 2^n \longrightarrow x \& (2^n - 1)$

$0\ \underbrace{1\ 1\ 1 \ldots}_{n \text{ সংখ্যক } 1}$

so last n bits পাবো

$2^n \rightarrow$ cache এ কতগুলো block আছে

last n bit দিয়ে → index বের করলাম

↳ যেসব address map হয় তাদের এই n bits same থাকে।

so, store করবো higher bit গুলো — tag bit বলে। (uniquely identify করার জন্য)

indexing যে bit দিয়ে করবো তার upper bit ?

valid — actually data এখানে store করা আছে কি না / valid কি না।

Tags and Valid Bits

- How do we know which particular block is stored in a cache location?
  - Store block address as well as the data
  - Actually, only need the high-order bits
  - Called the tag
- What if there is no data in a location?
  - Valid bit: 1 = present, 0 = not present
  - Initially 0

Cache Example

- 8-blocks, 1 word/block, direct mapped
- Initial state

address 나눠

22, decimal value

26,

22,

26,

16,

3,

16,

18,

16

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

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110

22,
26,
22,
26,
16,
3,
16,
18,
16

valid – N
so Miss
memory থেকে read করে
আনা লাগবে
then valid
Y করে দিবো

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
26	11 010	Miss	010

22,
26,
22,
26,
16,
3,
16,
18,
16

Index	V	Tag	Data
000	N		
001	N		
010	**Y**	**11**	**Mem[11010]**
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Hit	110
26	11 010	Hit	010

22,
26,
22,
26,
16,
3,
16,
18,
16

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

valid – yes
Tag – same
Hit
– আগেই load
করা আছে।

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

22,
26,
22,
26,
16,
3,
16,
18,
16

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

22,
26,
22,
26,
16,
3,
16,
18,
16

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

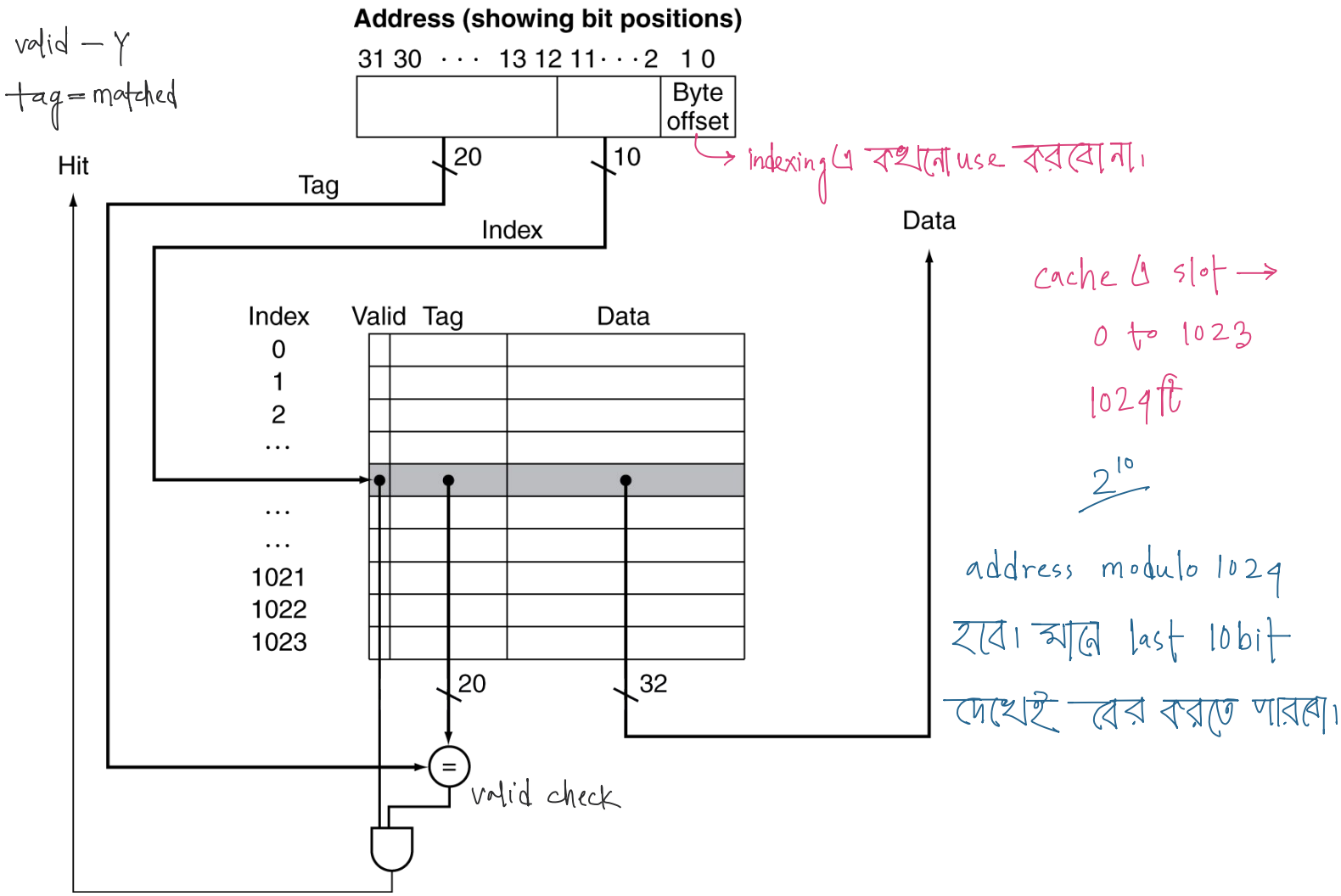
Cache Example

Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

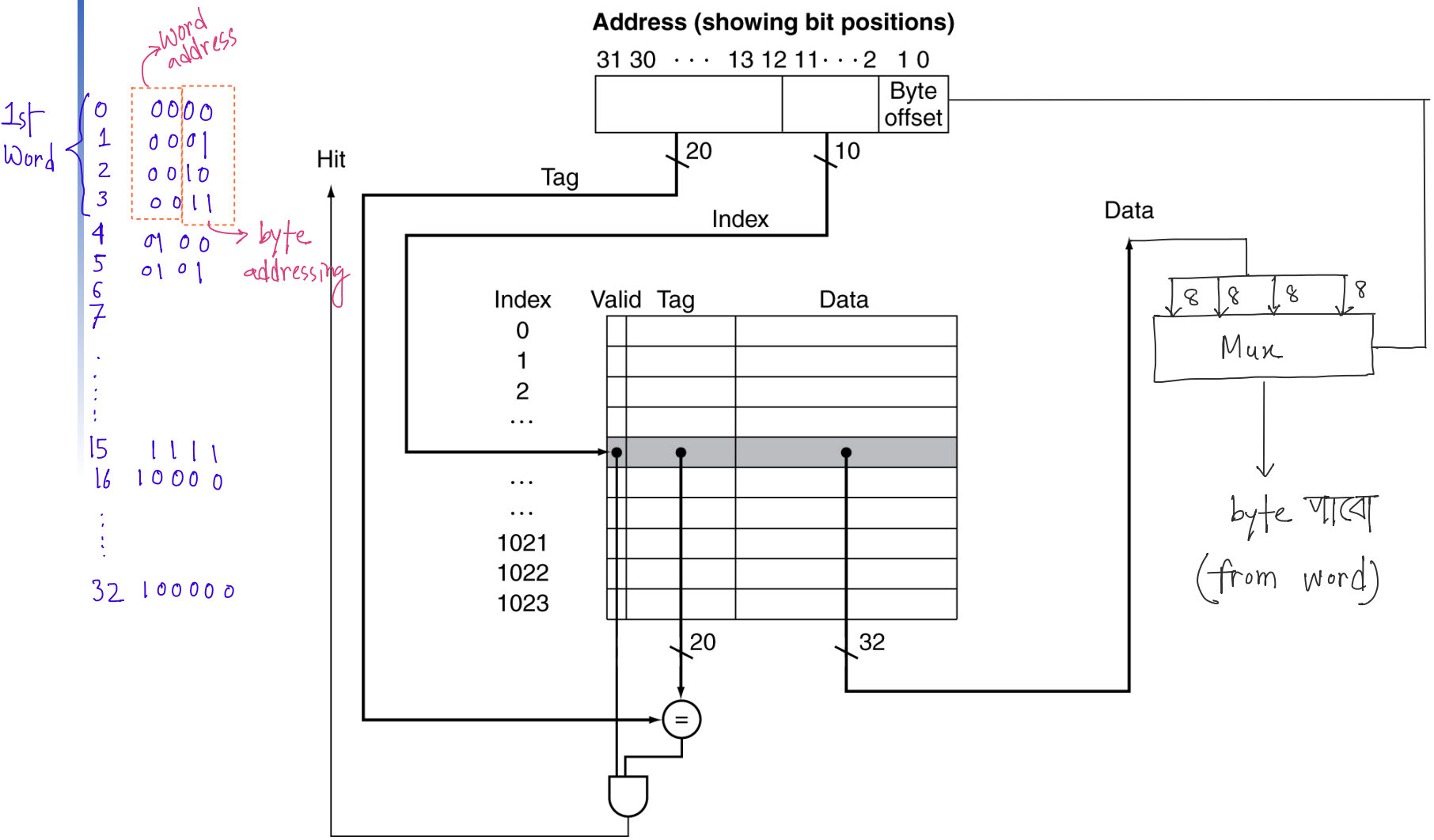
22,
26,
22,
26,
16,
3,
16,
18,
16

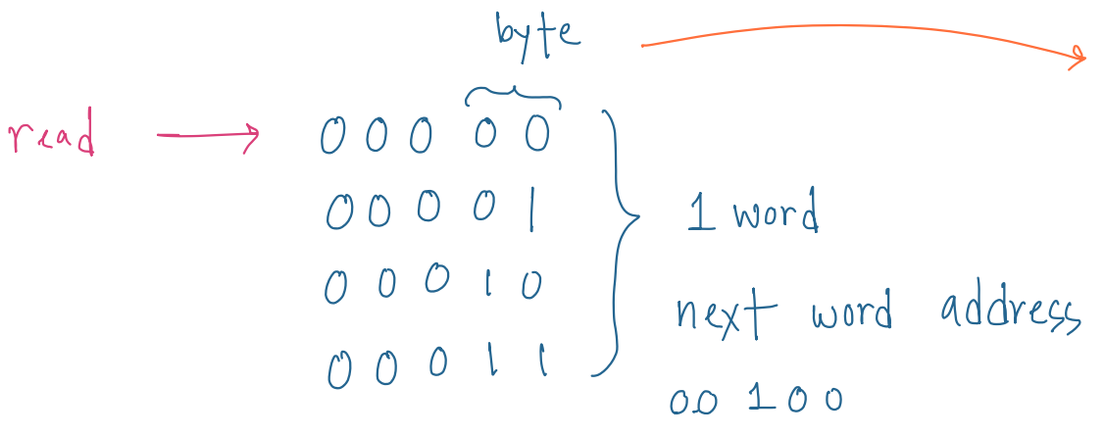
Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	10	Mem [10010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Address Subdivision

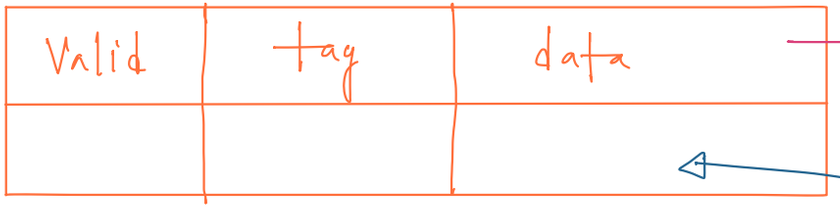


Address Subdivision





read ⟶ 00 1 00



→ block (minimum unit যেটা memory থেকে copy করে আনতে পারি)

2^m words আনলাম

→ design choice

4 words 2^2 → $m = 2$

copy করে আনলাম

10 bit এর last 2 bit ignore ⟶ byte এর জন্য

7th – 8th bit ⟶ word index

0 – 5 bit যদি নেই পাহলেই 4 টা word কে uniquely identify করা যাবে।

10 10 1 000
001
010
011
} 1st word

10 10 1 1 0 0 2nd word

1011 0 0 00 ⟶ 4 word পরে চলে আসলাম

8 words copy করে আনতে 3 bit লাগবে।

Tag size:

32 − (n + m + 2)

- indexing
- কতগুলো word
- byte offset

Total bits in a cache

- 32-bit addresses
- A direct-mapped cache
- The cache size is 2^n blocks
  - so *n* bits are used for the index
- The block size (b) is 2^m words (2^{m+2} bytes = 2^{m+2+3} bits)
  - *m* bits are used for the word within the block
  - two bits are used for the byte part of the address

- Total number of bits (C)
  - **C** = 2^n x (b + t + v)
  - t = 32 – (n + m + 2)
  - b = 2^{m+5}
- **C**

= 2^n x (2^{m+5} + 32 – (n + m + 2) + 1)

= 2^n x (2^m x 32 + 32 –n - m -1)

Size of tag field

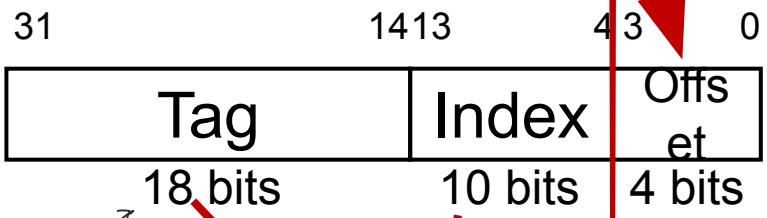
v is the valid field size, i.e., 1

Block size

Total bits in a cache

- How many total bits are required for a direct-mapped cache with 16 KiB of data and 4-word blocks (i.e., 16 bytes), assuming a 32-bit address?

- 16 KiB = 4 Ki Words = 1 Ki Blks = 2^{10} Blks
- C = 1024 x (b + t + v)
  = 1024x(4x32 + t + 1)
  = 1024x(4x32 +18+1)
  = 147 Kilo bits



$2^{10} * (1 + 18 + 4 \times 32)$

valid — 1

tag — 18

4 words — 4×32

32 bits each.

147 kiloBits $\longrightarrow$ 18 kiloBytes

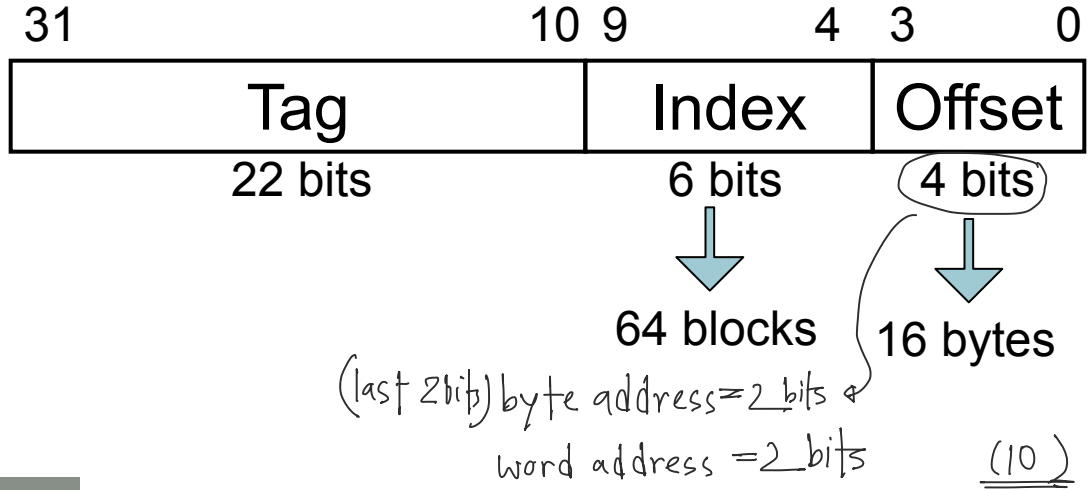
Tag, Valid bit — overhead

Example: Larger Block Size

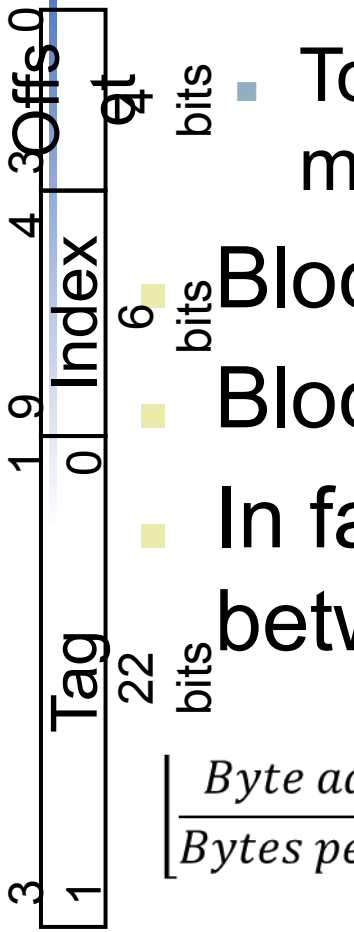
- 64 blocks, 16 bytes/block
  - To what block number does address 1200 map?
- Block address = ⌊1200/16⌋ = 75
- Block number = 75 modulo 64 = 11

Byte Address

[byte address / bytes per block]



Example: Larger Block Size



- 64 blocks, 16 bytes/block
  - To what block number does address 1200 map?
- Block address = $\lfloor 1200/16 \rfloor = 75$
- Block number = 75 modulo 64 = 11
- In fact Block 11 maps all addresses between 1200 and 1215.

$\lfloor$byte address / bytes per block$\rfloor$

$$\left\lfloor \frac{Byte\ address}{Bytes\ per\ block} \right\rfloor \times Bytes\ per\ block$$

$$\left\lfloor \frac{Byte\ address}{Bytes\ per\ block} \right\rfloor \times Bytes\ per\ block + (Bytes\ per\ block - 1)$$

Block Size Considerations

- Larger blocks should reduce miss rate
  - Due to spatial locality — বেশি word আনতে পারবো।
- But in a fixed-sized cache — ধরলাম 4 থেকে 16 করলাম — 12টা word বেশি রাখতে পারবো। miss rate কমবে।
  - Larger blocks ⇒ fewer of them
    - More competition ⇒ increased miss rate
- Larger miss penalty
  - Larger blocks ⇒ Larger transfer time
  - Can override benefit of reduced miss rate
  - Early restart critical-word-first can help

Block Size Considerations

- Larger blocks should reduce miss rate
  - Due to spatial locality
- But in a fixed-sized cache
  - Larger blocks ⇒ fewer of them
    - More competition ⇒ increased miss rate
- Larger miss penalty
  - Larger blocks ⇒ Larger transfer time
  - Can override benefit of reduced miss rate
  - Early restart critical-word-first can help

→ cache size fixed
block size বাড়ালাম
slot সংখ্যা কমবে।
multiple address
same slot এ map.
—miss rate বেড়ে গেল।

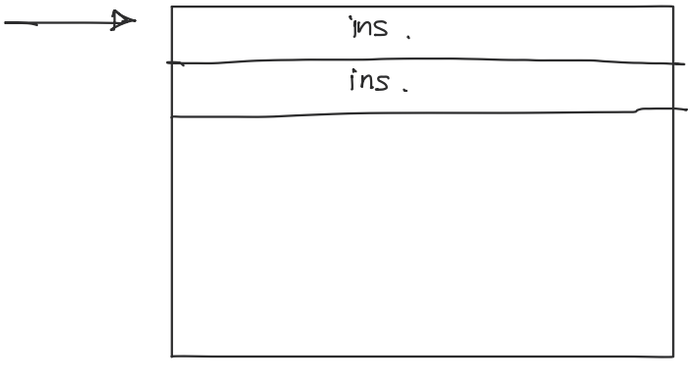
বেশি data
—memory থেকে
copy করে আনতে
সময় বেশি
লাগবে।
net benefit কমে গেল।

Early restart

→ যে word টা দরকার সেটা চলে আসলে CPU execution resume করবো। Parallaly বাকিটা loading হবে।

→ pipeline এ স্থা লাগে যখন memory থেকে data আনা লাগে।

- resume execution as soon as the requested word of the block is returned; Does not wait for the entire block
- For instruction access, it works best
  - Instruction accesses are largely sequential
  - so if the memory system can deliver a word every clock cycle, the processor may be able to restart operation when the requested word is returned, with the memory system delivering new instruction words just in time.
  - This technique is usually less effective for data caches because it is likely that the words will be requested from the block in a less predictable



ins. execute ঘ one clock cycle লাগে

best case: sequential instruction

lw → block copy হয়

Critical Word First

যে word টা চাই সেটা আগে copy করে আনলাম। এরপর CPU resume হবে।

- Organizes the memory so that the requested word is transferred from the memory to the cache first.
- The remainder of the block is then transferred, starting with the address after the requested word and wrapping around to the beginning of the block.
- Can be slightly faster than early restart
- but it is limited by the same properties that limit early restart.

same problem: data access যদি predictable না হয়।

Cache Misses

- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access

Write-Through

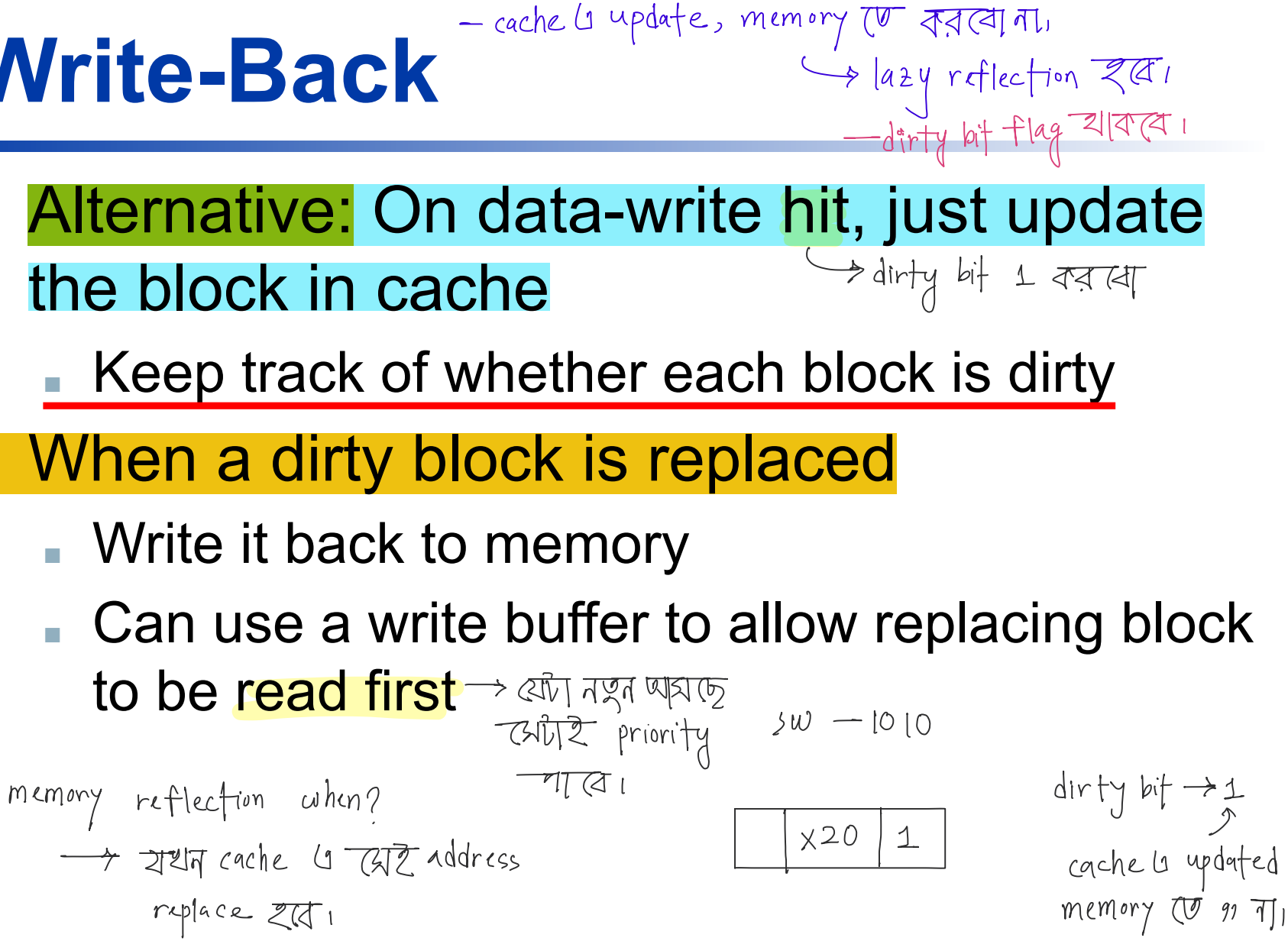
SW

→ cache এ hit হলে cache and memory তে update করবে

- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: also update memory
- But makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = 1 + 0.1×100 = 11
- Solution: write buffer → খুব বেশি write আসলে buffer full হবে। তখনও same problem. Buffer full হলে memory তে write so stall করা লাগবে।
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full

Write-Back

- Alternative: On data-write hit, just update the block in cache
  - Keep track of whether each block is dirty
- When a dirty block is replaced
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first



write এ miss → Write through scheme → memory, cache (update)

allocate on miss → memory তে update

যেটাই আবার cache এ আনবো।

cause যেটা আবার lw করতে পারি।

Cache এ load নাও করতে পারি। → eg: initialization এর case এ হয়

শুধু sw

sw

থাকে

Write Allocation

- What should happen on a write miss?
- Alternatives for write-through
  - Allocate on miss: fetch the block
  - Write around: don't fetch the block
    - Since programs often write a whole block before reading it (e.g., initialization)
- For write-back
  - Usually fetch the block
  - (See next slides)

Advantage for write-through

- we can write the data into the cache and then read the tag; — tag bit compare না করেই update করলাম কিন্তু এটা আসলে cache miss ছিল।
- if the tag mismatches, then a miss occurs.
- Because the cache is write-through, the overwriting of the block in the cache is not catastrophic
  - memory has the correct value and we can do it right.
- THIS CANNOT BE DONE FOR WRITE BACK → write back এ lazily update হয়। তাই এটা কাজ করবে না। dirty bit 1 ছিল...

Write Back

- If we have a cache miss, we must first write the block back to memory if the data in the cache is modified.
- stores require two cycles:
  - a cycle to check for a hit
  - followed by a cycle to actually perform the write
- Alternative: Write Buffer to hold that data
  -

data → SRAM এ
tag bit → SRAM এ থাকে

কেন hit check এ 1 cycle ? → check করার জন্য tag bit পড়তে হবে। সেখানে 1টা clock cycle যাবে।

— যেটা cache এ write করবে সেটা buffer এ রেখে CPU execution resume করবে। hit হলে পরে এখান থেকে write হবে।

(pipelining করে এক cycle এ করতে চাইলে) (next clock cycle এ) unused

Write Back- Write Buffer

- write buffer holds the data to write in
- effectively allowing the store to take only one cycle by pipelining it:
  - When a store buffer is used, the processor does the cache lookup and places the data in the store buffer during the normal cache access cycle.
  - Assuming a cache hit, the new data is written from the store buffer into the cache on the next unused cache access cycle.

Write Back- Write Buffer for miss

- the modified block is moved to a write-back buffer associated with the cache in case of a miss
  - while the requested block is read from memory.
  - The write-back buffer is later written back to memory.
- Assuming another miss does not occur immediately, this technique reduces the miss penalty

Example: Intrinsity FastMATH

processor

- Embedded MIPS processor
  - 12-stage pipeline
  - Instruction and data access on each cycle
- Split cache: separate I-cache and D-cache
  - Each 16KB: 256 blocks × 16 words/block
  - D-cache: write-through or write-back
- SPEC2000 miss rates
  - I-cache: 0.4%
  - D-cache: 11.4%
  - Weighted average: 3.2%

Example: Intrinsity FastMATH

16 words/block
16 × 1 mux
→ selection block offset

Address (showing bit positions)

256 blocks × 16 words/block

tag bit = 32 − 8 − 4 − 2 = 18

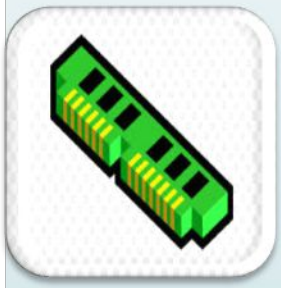
2^8 → 8 bit index

block offset = 4

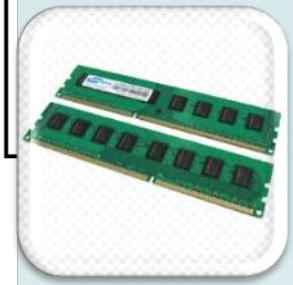
Hit

ta

In practice, to eliminate the multiplexor, caches use two RAMS



a smaller RAM for the tags



a large RAM for the data with the block offset supplying the extra address bits

Measuring Cache Performance

- Components of CPU time
  - CPU execution cycles
    - Includes cache hit time
  - Memory stall cycles
    - Mainly from cache misses

metrices → compare করার জন্য লাগবে।

→ execution করতে data দরকার যেটা cache এ নাই। memory থেকে আনা লাগবে।

Measuring Cache Performance

- $$\text{Read-stall cycles} = \frac{\{\text{Reads}\}}{\{\text{Program}\}} \times \text{Read miss rate} \times \text{Read miss penalty}$$

reads per program

- $$\text{Write-stall cycles} = \left(\frac{\{\text{Writes}\}}{\{\text{Program}\}} \times \text{Write miss rate} \times \text{Write miss penalty}\right) + \text{Write buffer stalls}$$

→ write through scheme ও buffer introduce করা হয়েছিল। Buffer full হয়ে গেলে buffer stall লাগবে।

For Write-through Cache

Notes: Write Buffer Stalls

- Write buffer stalls depend
  - on the proximity of writes
  - and not just the frequency

কাছাকাছি time এ কত দ্রুত আসছে

So not easy to deduce an equation

তাই ignore

- Usually we can ignore write buffer stalls
  - in systems with 4 or more words depth
  - With a memory capable of accepting writes at a rate that significantly exceeds the average write frequency in programs (by a factor of 2)

write buffer stall খুব বেশি হয় না।

- If a system did not meet these criteria, it would not be well designed;
  - instead, the designer should have used either a deeper write buffer or a write-back organization

– rethink/redesign করা লাগবে

Measuring Cache Performance

- With simplifying assumptions:
  - read and write miss penalties are same
    - In most write-through schemes this is the case
  - Write buffer stalls are negligible

$$\text{Memory stall cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

For Data

$$\text{Memory stall cycles} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

For Instruction

All instruction search করা লাগে। Cache এ কোথায় আছে।

instruction একটা করে পড়া লাগে

Cache Performance Example

- Given
  - miss rate: I-cache = 2% ; D-cache = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions
- Say total instruction: I
  - I-cache miss cycles: I × 0.02 × 100 = 2.00 × I
  - D-cache: I × 0.36 × 0.04 × 100 = 1.44 × I
- Actual CPI = 2 + 2 + 1.44 = 5.44
- Ideal CPU is 5.44/2 =2.72 times faster

How much faster is a processor with a perfect cache?

বোনো design কে faster করবো; সেটার কিছু dependency থাকে

Amdahl's Law

- A rule stating that the performance enhancement possible with a given improvement is limited by the amount that the improved feature is used.
- What happens if the processor is made faster, but the memory system is not?
  - The amount of time spent on memory stalls will take up an increasing fraction of the execution time

Cache Performance Example 2

- Given
  - miss rate: I-cache = 2% ; D-cache = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 1
  - Load & stores are 36% of instructions
- Say total instruction: I
  - I-cache miss cycles: I × 0.02 × 100 = 2.00 × I
  - D-cache: I × 0.36 × 0.04 × 100 = 1.44 × I
- Actual CPI = 1 + 2 + 1.44 = 4.44
- Ideal CPU is 4.44/1 =4.44 times faster

How much faster is a processor with a perfect cache?

Base CPI improved

cache থেকে কোনো improvement নেই। Processor থেকে আসছে।

miss না হলে → super fast

miss হলে → performance degrade.

Lesson (re)Learned

- So, in Example 2:
  - we have improved the processor architecture
    - Ideal CPI is 1 (previously it was 2)
  - Memory system was NOT improved
  - Now effective CPI has become 4.44

Previously it was 5.44

  - So, there is a small improvement; Not what we expected
- The amount of execution time spent on memory stalls:

Performance lost increases

  - Increases From $\frac{3.44}{5.44} = 63\%$ to $\frac{3.44}{4.44} = 77\%$

cache এর কারনে যেটা আসে।

processor idle থাকে। চলে।

↳ total time এর 63% cache এ

↳ total time এর 77% cache এ

↳ So আগেরটাই ভালো design, idle time বেশি ছিল, যদিও effective CPI কম ছিল।

Average Access Time

- Hit time is also important for performance
- Average memory access time (AMAT)
  - AMAT = Hit time + Miss rate × Miss penalty
- Example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
  - AMAT = 1 + 0.05 × 20 = 2ns
    - 2 cycles per instruction

Performance Summary

- When CPU performance increased
  - Miss penalty becomes more significant
- Decreasing base CPI
  - Greater proportion of time spent on memory stalls
- Increasing clock rate
  - Memory stalls account for more CPU cycles
- Can't neglect cache behavior when evaluating system performance

— cache improve করিনাই (but processor improve করলাম)

Associative Caches

directly mapped নাও হতে পারে
memory block & cache block

- Fully associative
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
  - Comparator per entry (expensive)
- *n*-way set associative
  - Each set contains *n* entries
  - Block number determines which set
    - (Block number) modulo (#Sets in cache)
  - Search all entries in a given set at once
  - *n* comparators (less expensive)

এটা address cache এর মধ্যে যেকোনো entry তে যেতে পারে। fixed mapping নেই।

parallely সব জায়গায় খুঁজবো

directly mapped, fully associated এর মাঝে

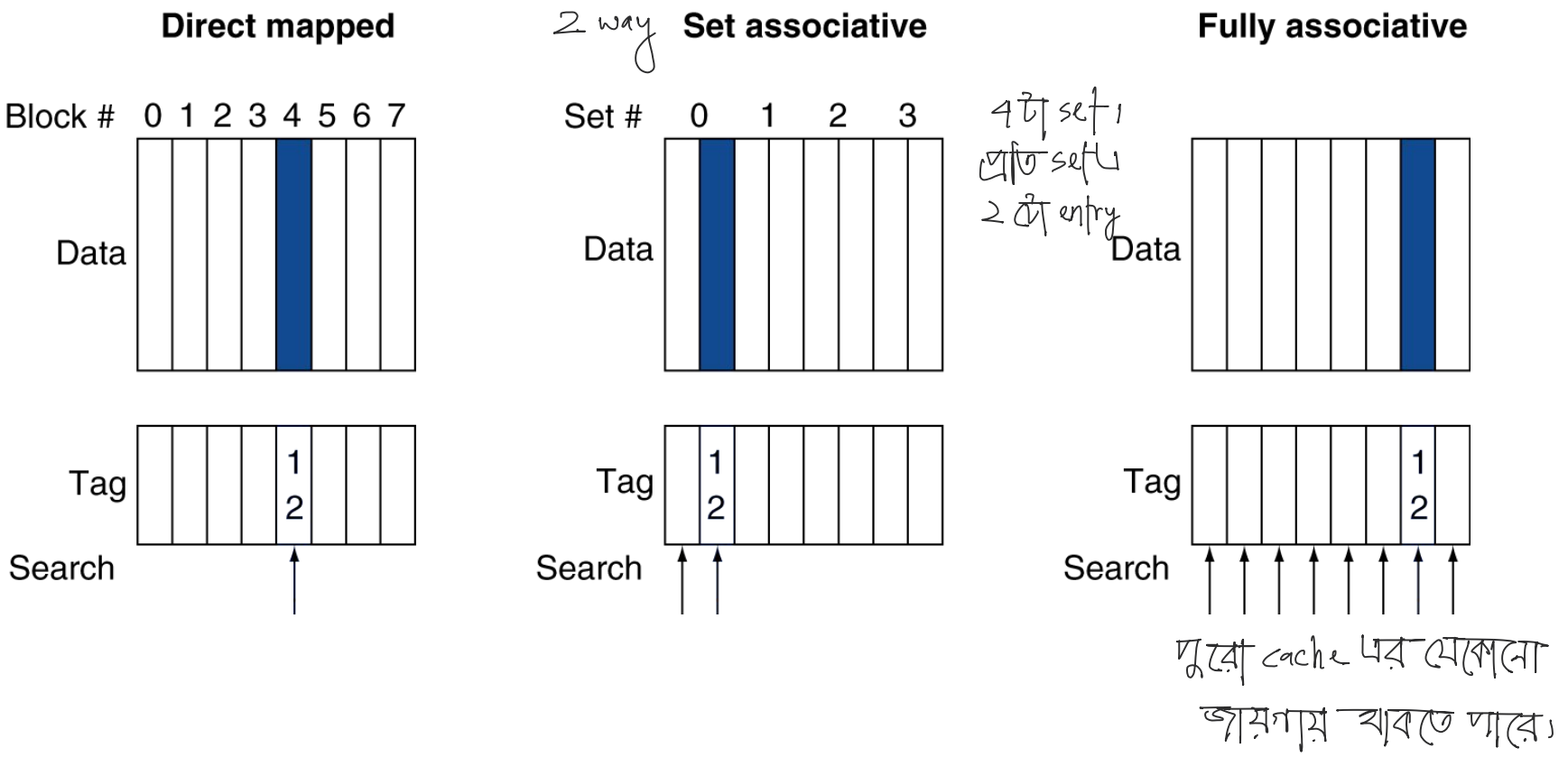
index bit কোন set যেটা determine করবে।

specific set search দুরো cache লাগবে না।

resource

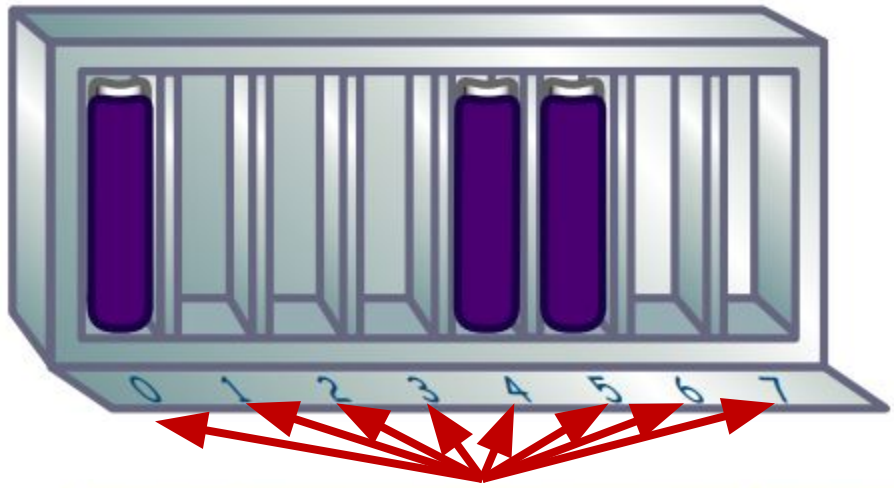
direct map এ → cache এ জায়গা ফাঁকা থাকলেও replace করা লাগছে। utilize করা যাচ্ছে না।

Associative Cache Example

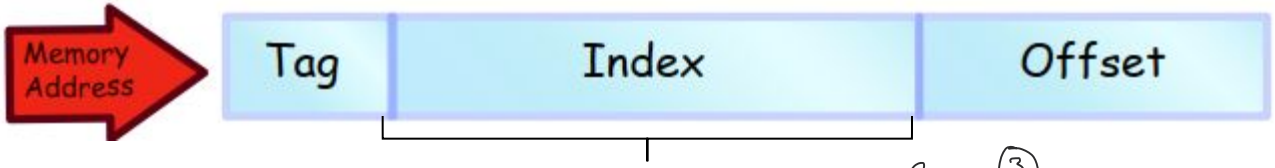


1-Way Set Associative

→ direct map

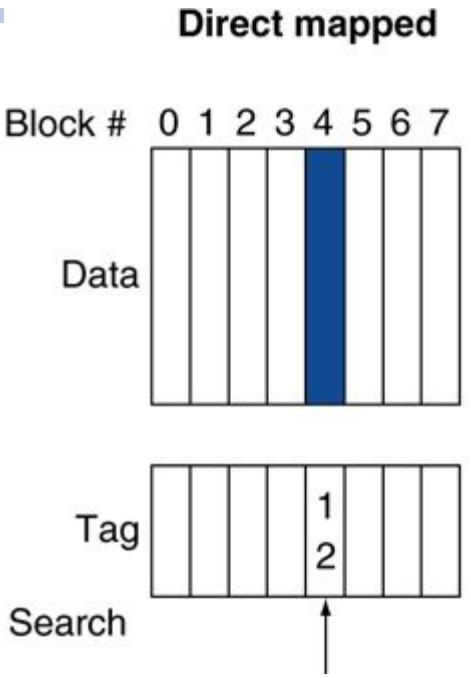


প্রতি block এ একটা entry



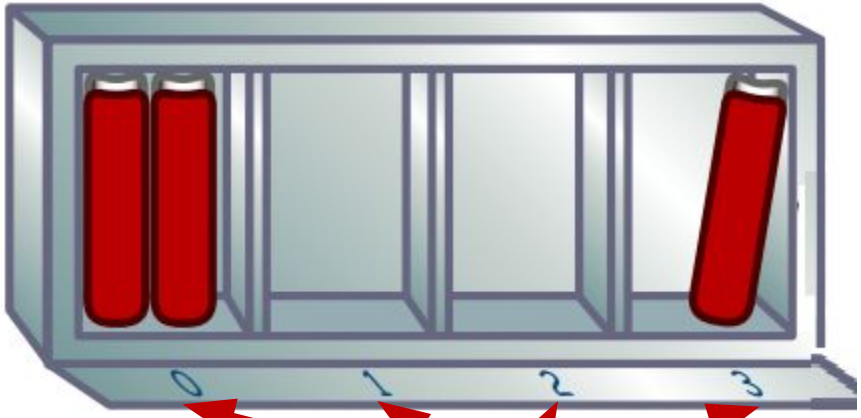
How many bits? 3 bits

$8 = 2^3$

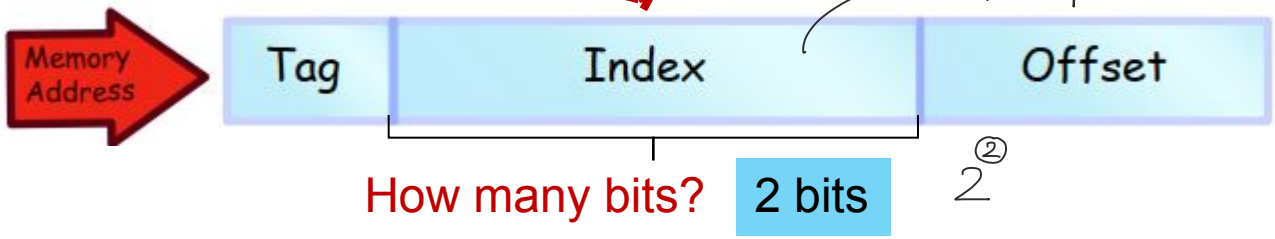


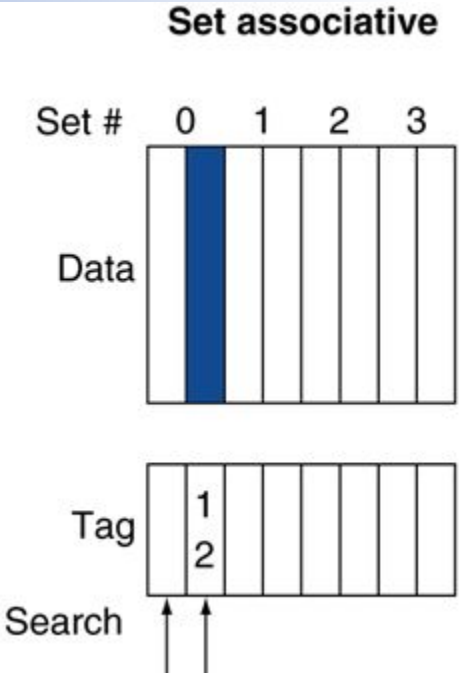
- A cache block can only go in one spot in the cache.
- It makes a cache block very easy to find
- but it's not very flexible about where to put the blocks.

2-Way Set Associative



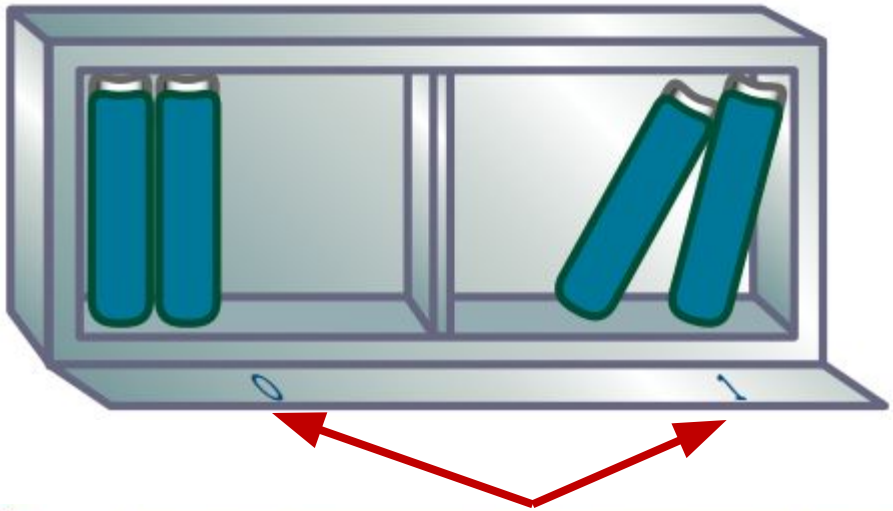
4 blocks বা set
প্রতি set এ ২টা
করে entry রাখা যাবে।
কোথায় রাখবে সেটা
random.

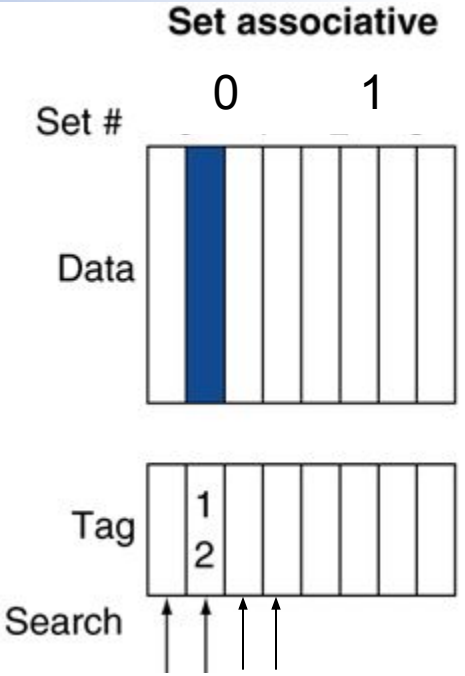




- This cache is made up of sets that can fit two blocks each.
- The index is now used to find the set
- The tag helps find the block within the set.

4-Way Set Associative





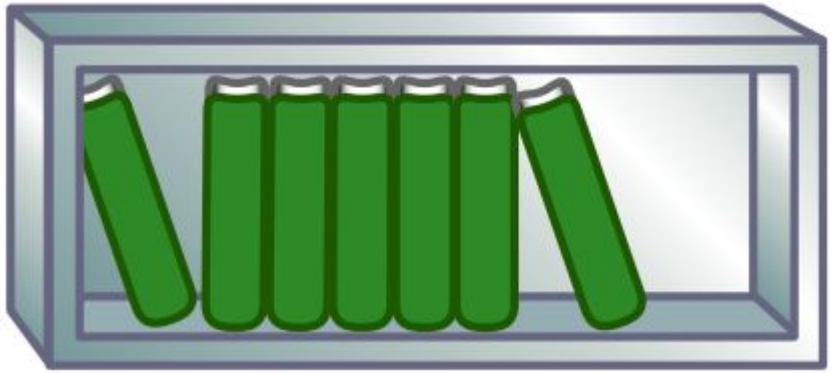
How many bits? 1 bit

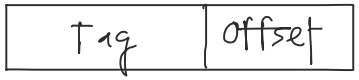
2^{1}

index bit ↓
tag bit ↑

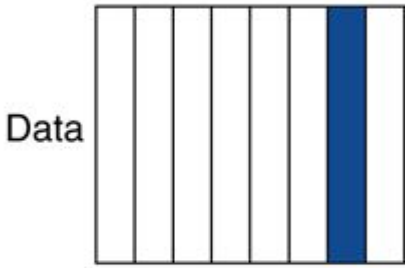
- Each set here fits four blocks,
- So there are fewer sets.
- As such, fewer index bits are needed.

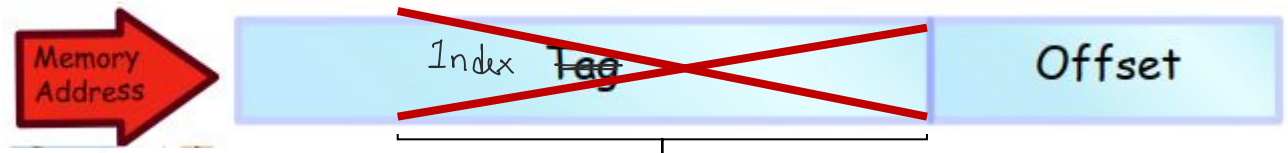
Fully Associative

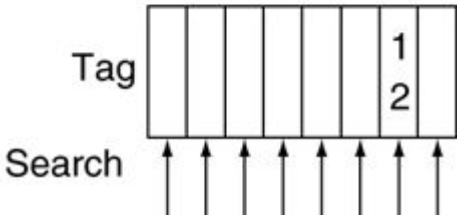




index — 0







How many bits? 0 bit

- No index is needed, since a cache block can go anywhere in the cache.
- Every tag must be compared when finding a block in the cache
- but block placement is very flexible!

Spectrum of Associativity

- For a cache with 8 entries

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				

한 set 에 2 blocks

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

· 4 entries per set

Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

8 entries
1 set 에

Associativity Example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8
- Direct mapped

0 modulo 4 = 0
8 modulo 4 = 0
0 modulo 4 = 0
6 modulo 4 = 2
8 modulo 4 = 0

as শুরুতে কোনো data নেই

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	Mem[0]			
8	0	miss	Mem[8]			
0	0	miss	Mem[0]			
6	2	miss	Mem[0]		Mem[6]	
8	0	miss	Mem[8]		Mem[6]	

Associativity Example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8
- 2-way set associative

replacement scheme :
least recently used ট replace.
recently accessed ট রাখবে।

0 modulo 2 = 0
8 modulo 2 = 0
0 modulo 2 = 0
6 modulo 2 = 0
8 modulo 2 = 0

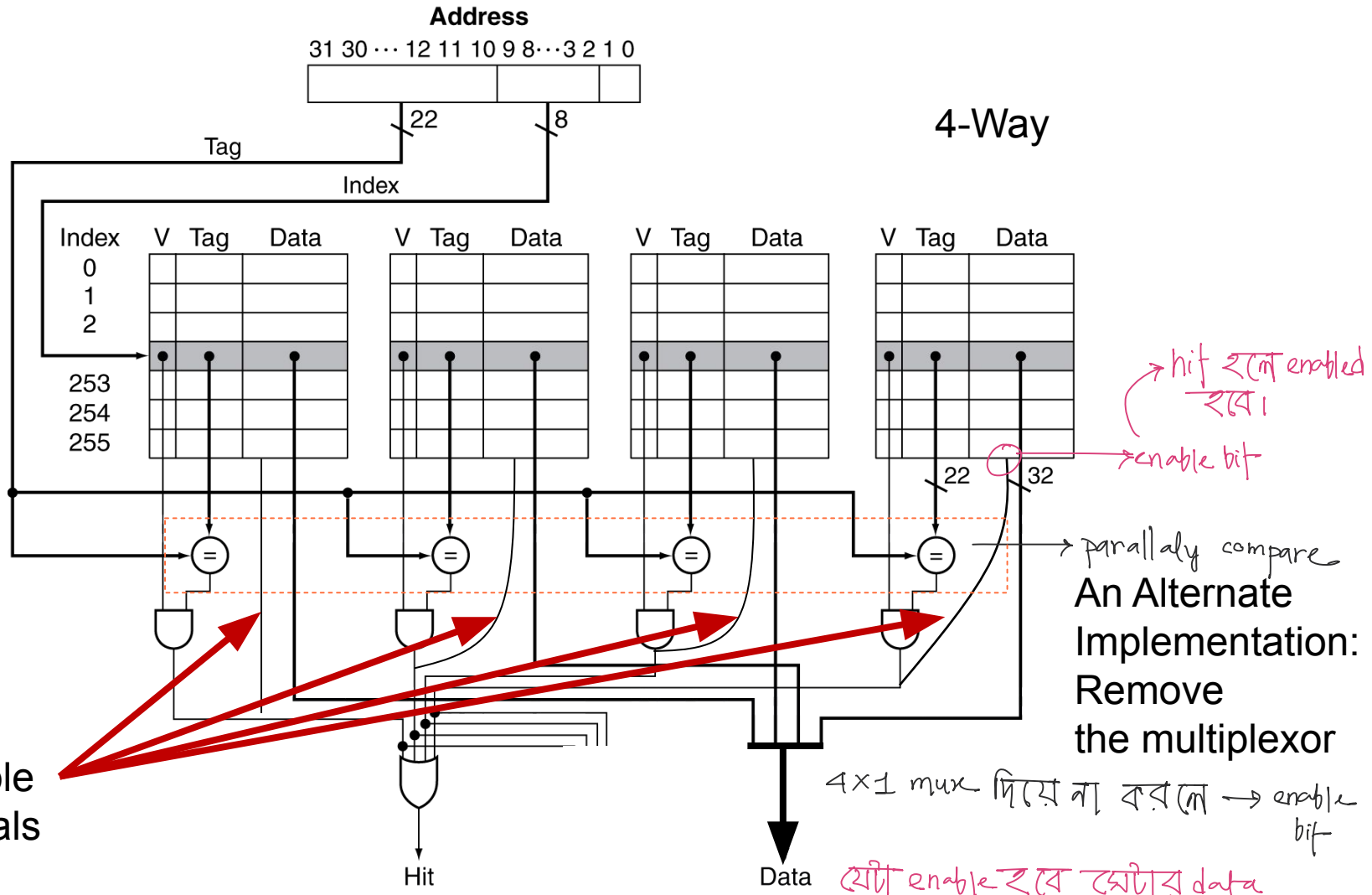
Block address	Cache index	Hit/miss	Cache content after access	
			Set 0	Set 1
0	0	miss	Mem[0]	
8	0	miss	Mem[0]	Mem[8]
0	0	hit	Mem[0]	Mem[8]
6	0	miss	Mem[0]	Mem[6]
8	0	miss	Mem[8]	Mem[6]

Associativity Example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8
- Fully associative

Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			
8		miss	Mem[0]	Mem[8]		
0		hit	Mem[0]	Mem[8]		
6		miss	Mem[0]	Mem[8]	Mem[6]	
8		hit	Mem[0]	Mem[8]	Mem[6]	

Set Associative Cache Organization



Replacement Policy

- Direct mapped: no choice
- Set associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
- Least-recently used (LRU)
  - Choose the one unused for the longest time
    - Simple for 2-way, manageable for 4-way, too hard beyond that
- Random
  - Gives approximately the same performance as LRU for high associativity

n-way set. → n সংখ্যক choice

কোনটা সবার আগে use হয় track করা লাগবে।

2 way হলে → 1টা extra bit রাখা লাগবে।

4 way → 2টা bit লাগবে

> 4 হলে → problem.

Imp (পরীক্ষার জন্য)

Tags versus Set Associativity

- Cache of 4096 blocks
- a 4-word block size
- 32-bit address,
- Find the total number of sets and the total number of tag bits for caches that are
  - direct mapped
  - two-way set associative
  - four-way set associative
  - fully associative.

Direct mapped:

$2^{12} = 4096$

block offset

	12 bits	4 bits

sets = 4096

$32 - 12 - 2 - 2 = 16$

↑

Tag bits (per block)

Total = 16×4096

2way set Associative:

index = 11

$32 - 11 - 2 - 2 = 17$

#sets = 2048

4 way set Associative:

$32 - 10 - 2 - 2 = 18$

#sets = 1024

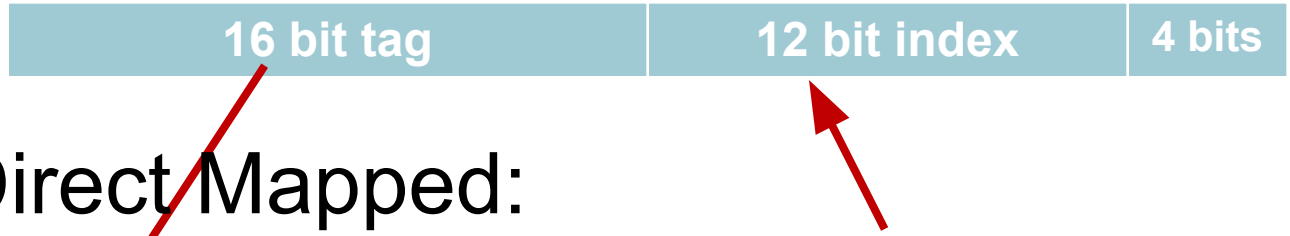
Fully Set Associative:

$32 - 0 - 2 - 2 = 28$

#sets = 1

Tags versus Set Associativity

- 4096 blocks; 32 bit address
- 4 words block = 16 (2^4) Bytes block
- So for tag + index 32 – 4 = 28 bits



- Direct Mapped:
  - 4096 (2^{12}) 1-way set => 12 bit index
  - 16 bit tag (28 -12)
  - 4096 entries × 16 bit tag = 64 K bits tag

Tags versus Set Associativity

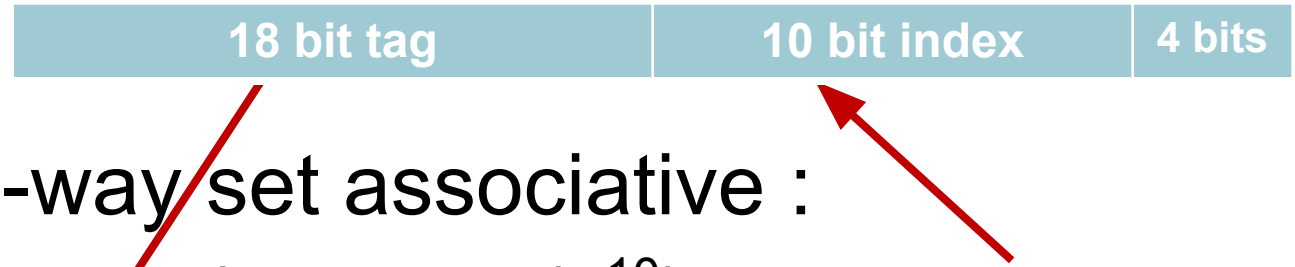
- 4096 blocks; 32 bit address
- 4 words block = 16 (2^4) Bytes block
- So for tag + index 32 – 4 = 28 bits

- 2-way set associative :
  - 4096/2 = 2048 (2^{11}) sets => 11 bit index
  - 17 bit tag (28 -11)
  - 4096 entries × 17 bit tag = 68 K bits tag

Associativity ↑ → Tag bit ↑ → overhead ↑ → costing ↑

Tags versus Set Associativity

- 4096 blocks; 32 bit address
- 4 words block = 16 (2^4) Bytes block
- So for tag + index 32 – 4 = 28 bits



- 4-way set associative :
  - 4096/4 = 1024 (2^{10}) sets => 10 bit index
  - 18 bit tag (28 -10)
  - 4096 entries × 18 bit tag = 72 K bits tag

Tags versus Set Associativity

- 4096 blocks; 32 bit address
- 4 words block = 16 (2^4) Bytes block
- So for tag + index 32 – 4 = 28 bits

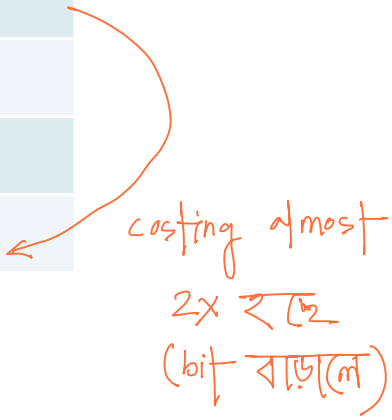
- Fully set associative :
  - No index
  - 28 bits tag (28 - 0)
  - 4096 entries × 28 bit tag = 112 K bits tag

Tags versus Set Associativity

- 4096 blocks; 4-words block; 32-bit address
- 4-words block = 16 (2^4) Bytes block
- So, for tag + index has 32 – 4 = 28 bits.

28 bits (tag + Index)	4 bits

Associativity	Tag bits (K)
Direct mapped (1 Way)	64
2 Way Set Associative	68
4 Way Set Associative	72
Fully Associative	112



Multilevel Caches

- Primary cache attached to CPU
  - Small, but fast → miss হলে level 2 এ যাবে
- Level-2 cache services misses from primary cache → miss হলে RAM এ যাবে
  - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache

Multilevel Cache Example

- Given
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction @ primary cache = 2%
  - Main memory access time = 100ns
  - 4 GHz => 0.25ns cycle length
- With just primary cache
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 × 400 = 9

Example (cont.)

- Now add L-2 cache
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
  - Penalty = 5ns/0.25ns = 20 cycles
- Primary miss with L-2 miss
- CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4
- Performance ratio = 9/3.4 = 2.6

CPU base CPI = 1,
clock rate = 4GHz

primary Miss rate = 2%

Multilevel Cache Considerations

- Primary cache
  - Focus on minimal hit time
- L-2 cache
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact
- Results
  - L-1 cache usually smaller than a single cache
  - L-1 block size smaller than L-2 block size

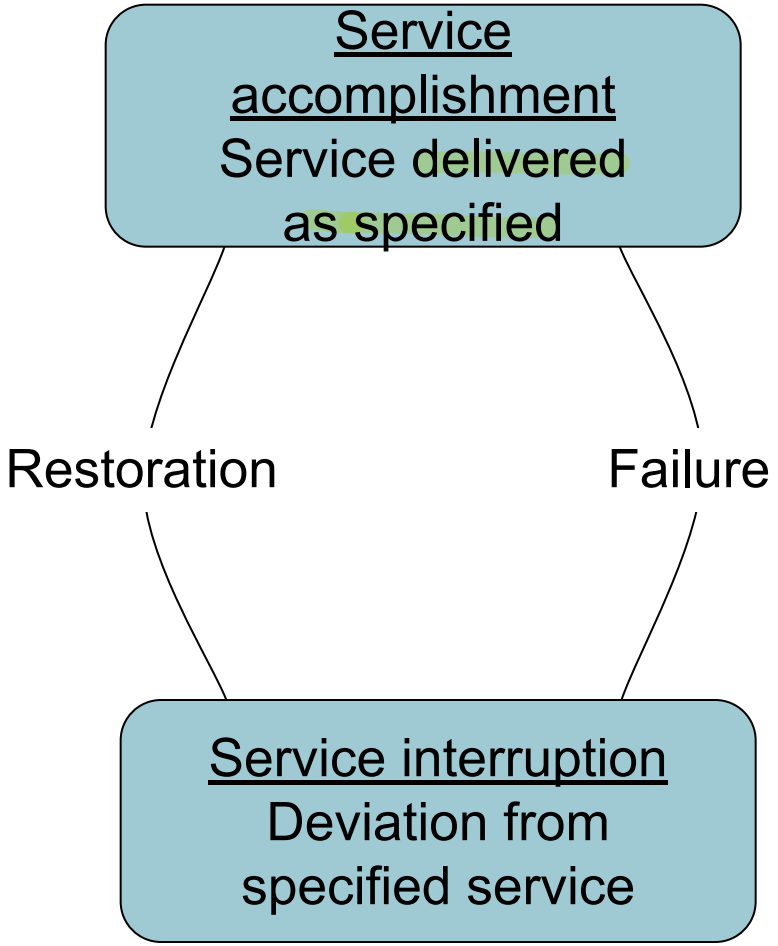
primary cache → minimal hit time (target)

L2 → miss rate কমানো target কারণ miss হলে mem এ যেতে হবে।

Interactions with Advanced CPUs

- Out-of-order CPUs can execute instructions during cache miss
  - Pending store stays in load/store unit
  - Dependent instructions wait in reservation stations
    - Independent instructions continue
- Effect of miss depends on program data flow
  - Much harder to analyse
  - Use system simulation

Dependability



- Fault: failure of a component
  - May or may not lead to system failure

Dependability Measures

- Reliability: mean time to failure (MTTF)
- Service interruption: mean time to repair (MTTR)
- Mean time between failures
  - MTBF = MTTF + MTTR
- Availability = MTTF / (MTTF + MTTR)
- Improving Availability
  - Increase MTTF: fault avoidance, fault tolerance, fault forecasting
  - Reduce MTTR: improved tools and processes for diagnosis and repair

Nines of Availability

- We want availability to be very high.
- One shorthand is to quote the number of "nines of availability" per year.

# of Nines	% of uptime
One	90
Two	99
Three	99.9
Four	99.99
Five	99.999

Nines of Availability

- Given 365 days per year, which is 365 * 24 * 60 = 525,600 minutes.
- Then the shorthand is decoded as follows:
- 90% => 525,600 * 0.1 downtime = 52560 minutes = 52560/(60*24) = 36.5 days.
-

# of Nines	% of uptime	Downtime/Year
One	90	36.5 days
Two	99	3.65 days
Three	99.9	526 min
Four	99.99	52.6 min
Five	99.999	5.26 min

MTTF and AFR

- MTTF is a reliability measure.
- A related term is *annual failure rate* (AFR)
  - The percentage of devices that would be expected to fail in a year for a given MTTF.
    - Hours in a year / MTTF
  - When MTTF gets large it can be misleading
  - while AFR leads to better intuition

High MTTF and AFR

- Some disks today are quoted to have a 1,000,000-hour MTTF.
  - 1,000,000/(365 * 24) = 114 years
  - they practically never fail???
- Warehouse scale computers that run Internet services such as Search might have 50,000 servers.
- Assume each server has 2 disks.
- How many disks we would expect to fail per year.

High MTTF and AFR

- One year => 365 * 24 = 8760 hours.
- A 1,000,000-hour MTTF means an AFR of 8760/1,000,000 = 0.876%.
- We have 50000 * 2 = 100,000 disks
- we would expect 0.00876 * 100,000 = 876 disks to fail per year
- On average more than (876/365) 2 disk failures per day.

tolerable ??

Question:

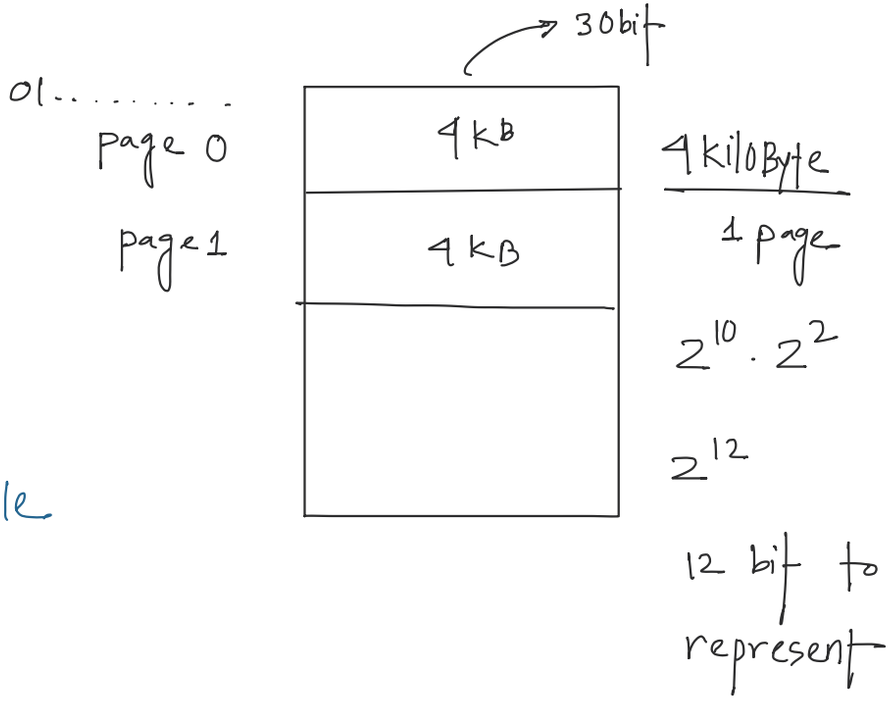
- Cache design
- math

Virtual Memory

প্রতিটা program এর নিজস্ব address space থাকবে। সেটার যেকোনো জায়গায় সে write করতে পারবে।

- Use main memory as a “cache” for secondary (disk) storage
  - Managed jointly by CPU hardware and the operating system (OS)
- Programs share main memory
  - Each gets a private virtual address space holding its frequently used code and data
  - Protected from other programs
- CPU and OS translate virtual addresses to physical addresses
  - VM “block” is called a page → unit of data harddisk থেকে পেজে RAM এ রাখবো
  - VM translation “miss” is called a page fault

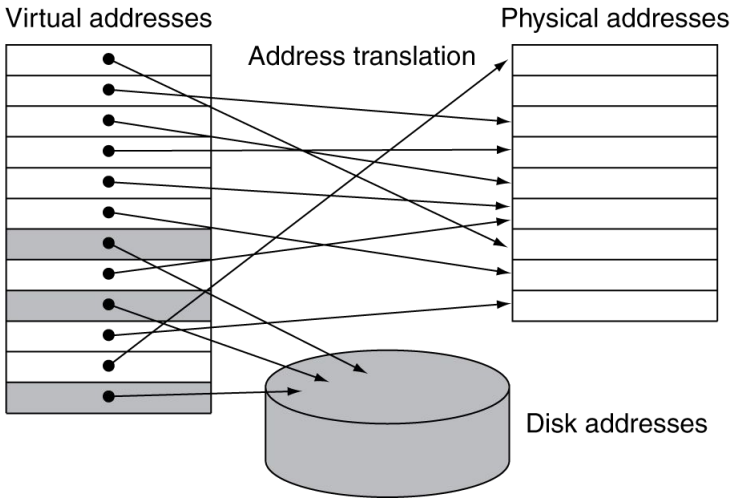
virtual address থেকে
physical address এ map করার
জন্য page table লাগে।



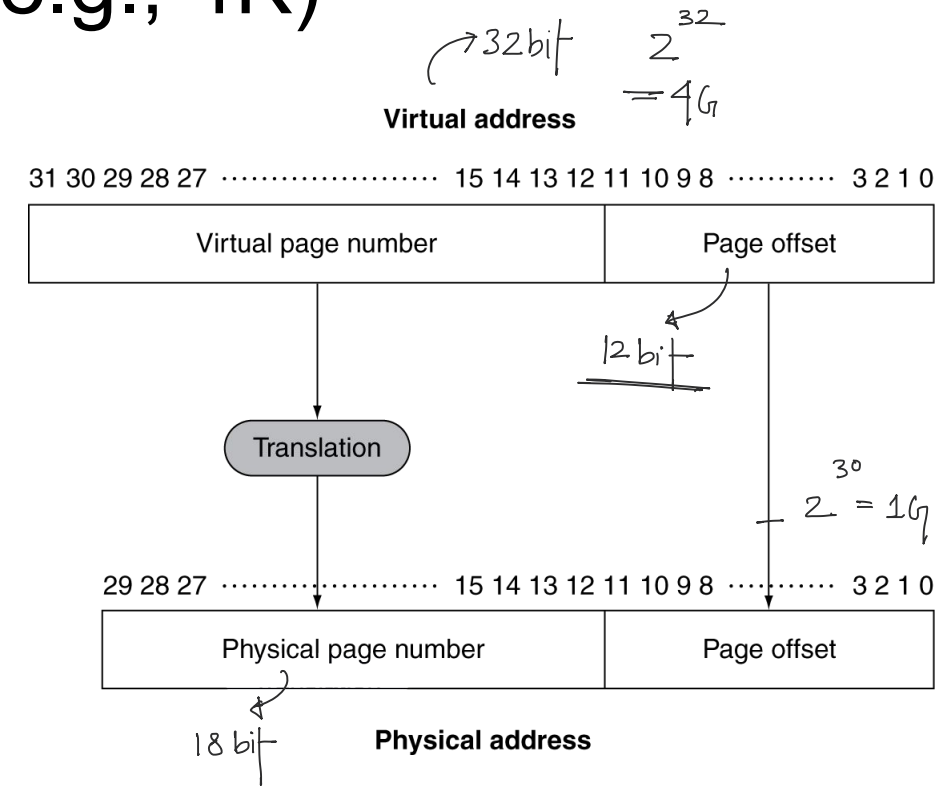
প্রতিটা process এর জন্য আলাদা page table
নিলে → no. of entries অনেক বেড়ে যায়।

Address Translation

- Fixed-size pages (e.g., 4K)



design focus → minimal page fault



OS protection দিবে যেন একটা program এর memory অন্য কোনো program access না করে।

Page Fault Penalty

- On page fault, the page must be fetched from disk
  - Takes millions of clock cycles
  - Handled by OS code
- Try to minimize page fault rate
  - Fully associative placement
    - But this needs costly search!!!
  - Smart replacement algorithms

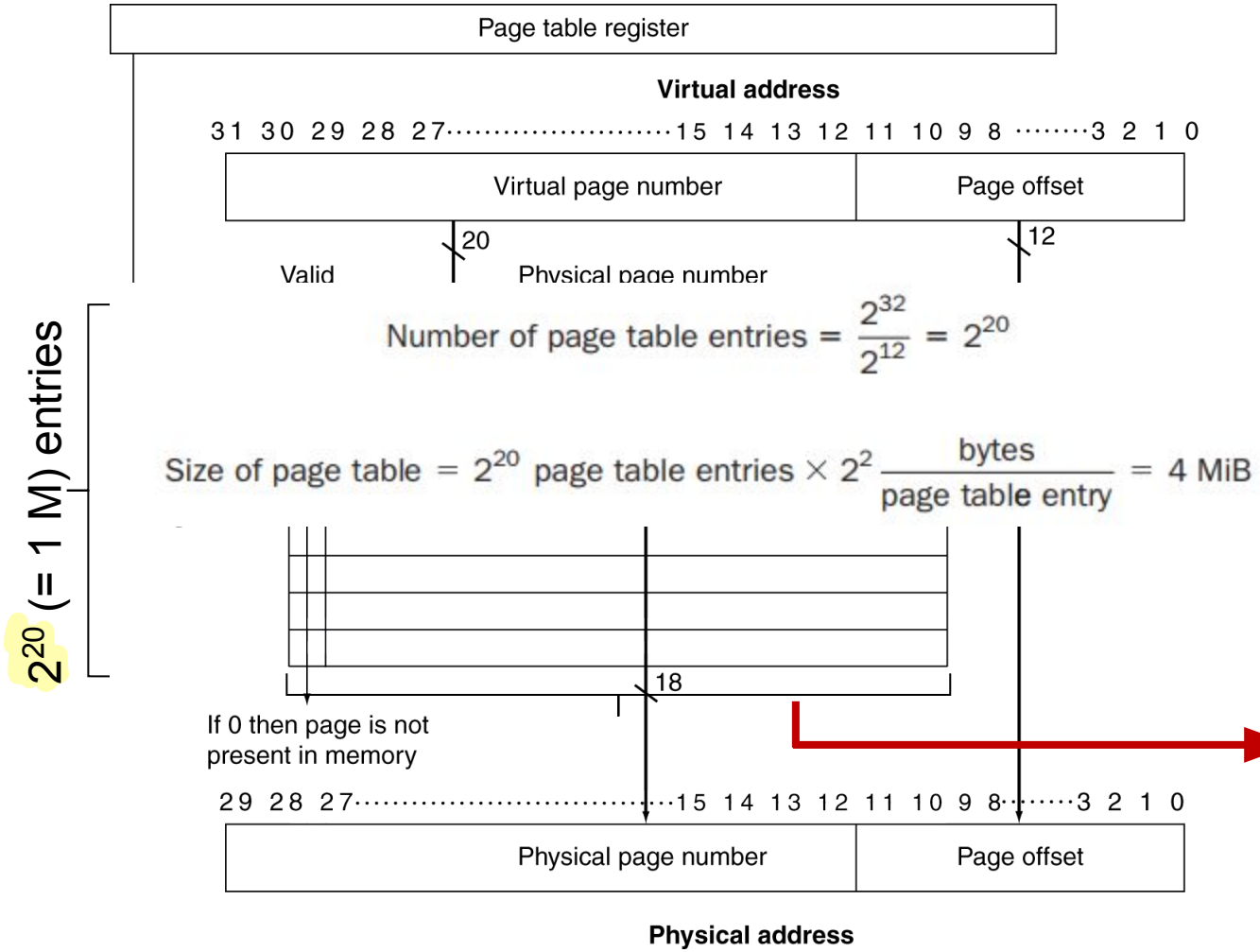
Page Tables

প্রতিটা process এর জন্য একটা করে page table থাকবে। → নিজস্ব virtual space আছে

- Stores placement information
  - Array of page table entries, indexed by virtual page number
  - Page table register in CPU points to page table in physical memory
- If page is present in memory
  - PTE stores the physical page number
  - Plus other status bits (referenced, dirty, …)
- If page is not present
  - PTE can refer to location in swap space on disk

valid bit → physically map করা আছে কি না।

Translation Using a Page Table



What is the size of the page table?

19 bits; but 32 bits wide usually

4 byte × 2^{20} entries

4 MB → size of a page table

Page Table Size Issues

- Size of page table => 4 MB (per process)
- What about 100 processes
  - Each with its own page table?
- What will happen if we have 64-bit addresses (according to prev. calc.)?
  - $2^{(64 - 12)} = 2^{52}$ entries!!!

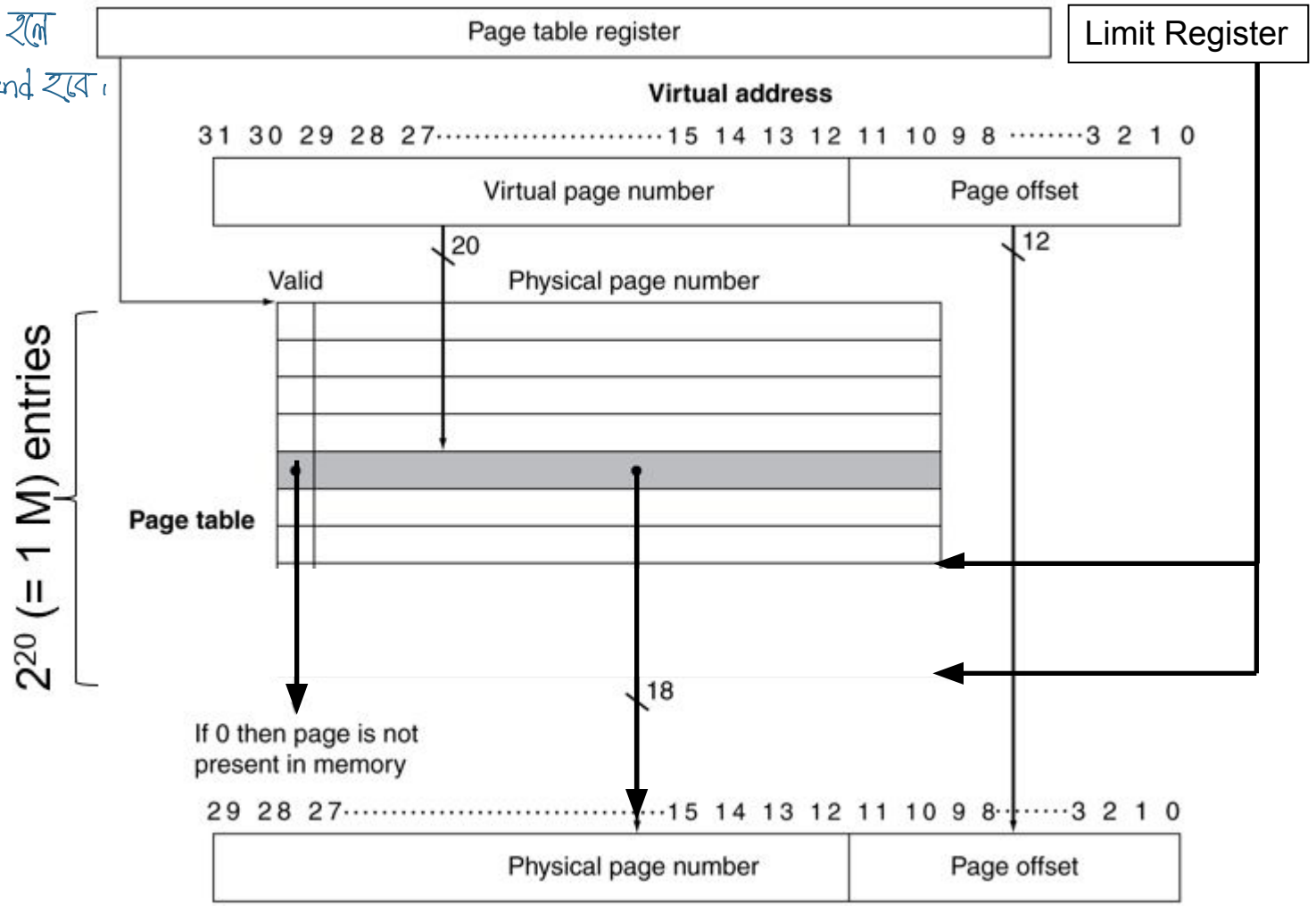
  - There are techniques to reduce the amount of storage required for the page table.

Techniques: Limit Registers

- Keep a limit register that restricts the size of the page table for a given process.
  - If the virtual page number becomes larger than the contents of the limit register, entries must be added to the page table.
  - This technique allows the page table to grow as a process consumes more space.
    - Thus, the page table will only be large if the process is using many pages of virtual address space.
  - This technique requires that the address space expand in only one direction.

Techniques: Limit Registers

higher order হলে
পুরোটা extend হবে।



Techniques: Two Limits

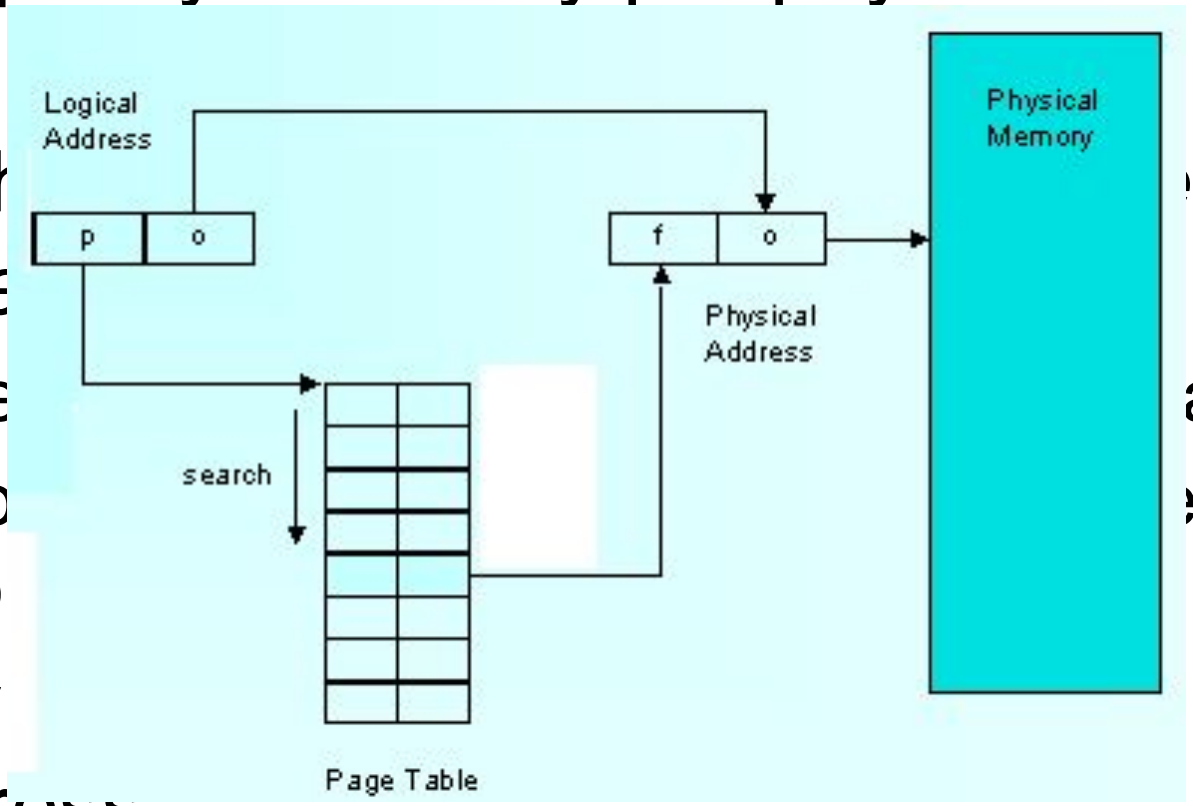
- Two separate page tables and two separate limits.
  - One grows from highest address down (Stack)
  - One grows from lowest address up (Heap)
  - So, address space is divided into 2 segments.
  - High-order bit of an address usually determines which segment
    - i.e., which page table to use for that address.
  - So, each segment can be as large as one-half of the address space.
  - A limit register for each segment specifies the current size of the segment, which grows in units of pages.

Allowing growth in only one direction is not sufficient

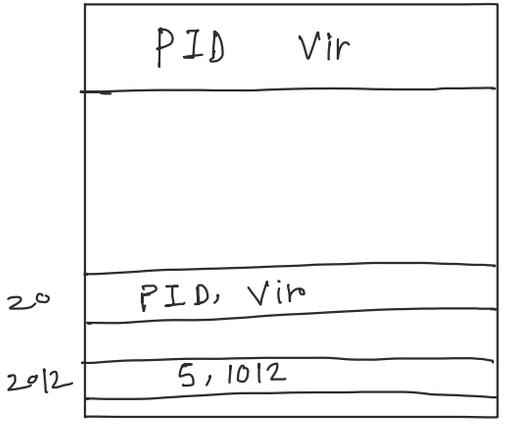
page → hard disk থেকে RAM এ read করে আনি যে unit

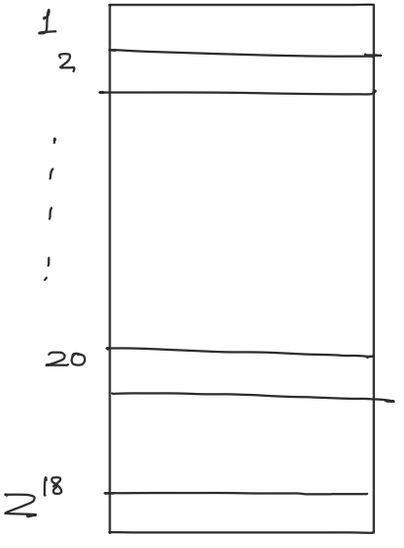
Techniques: Inverted Page Table

- Keep only one entry per physical block (i.e.,
- Such *d page table*
  - we able.
  - So e co
- May virtual address
  - To make the lookups faster



all process এর জন্য একটা page table maintain — inverted page table





$2^{18} \times 4 = 2^{22}$

↑

প্রতিটা entry 4 KB

PID → process identifier

Vir → virtual address

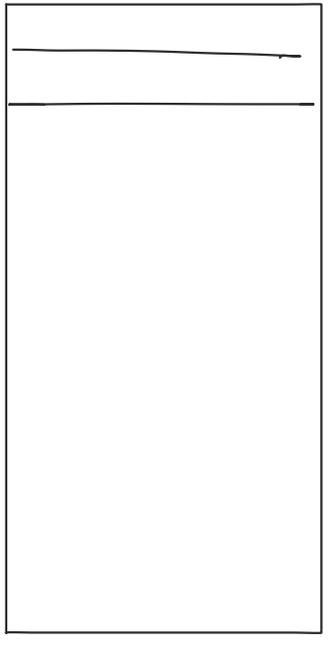
2012 page এ 5no process, 1012 virtual adress

5 খুঁজবো then check করবো vir 1012 কি না। index থেকে page no. (physical)

problem: দুরোটা search করা লাগে।

1MB দিয়েই সবগুলো process এর জন্যই page table বানাতে পারি।

V.A



2^{20}

1111 ...

0... 10

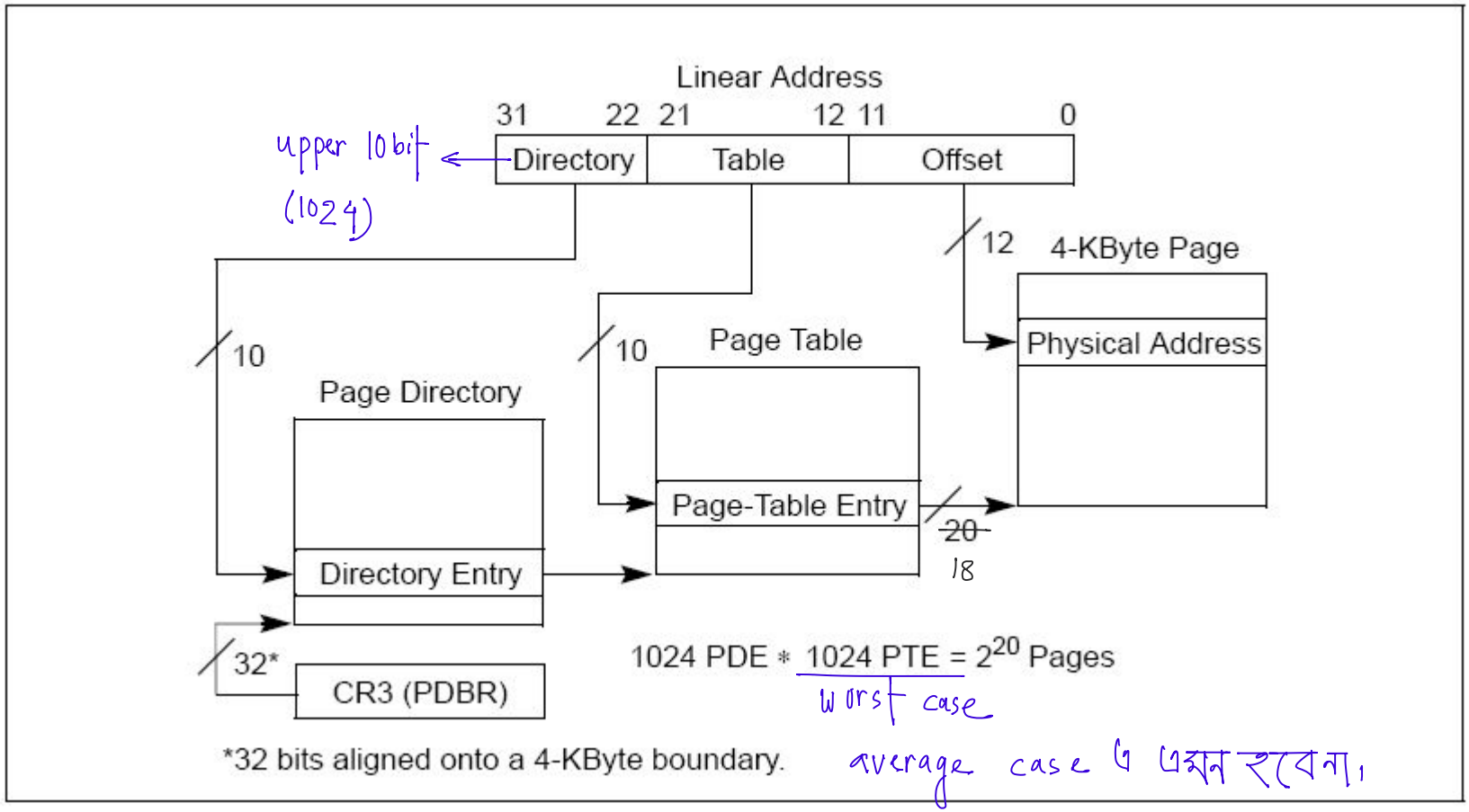
...11

প্রতিটা VA কখনোই লাগবে না।

Techniques: Multiple Levels

- The first level maps large fixed-size blocks of virtual address space
  Sometimes called segment
- Each entry in the segment table:
  - indicates whether any pages in that segment are allocated
  - if so, points to a page table for that segment.
- Address translation happens:
  - by first looking in the segment table, using the highest-order bits of the address.
  - If the segment address is valid, the next set of high-order bits is used to index the page table

Techniques: Multiple Levels

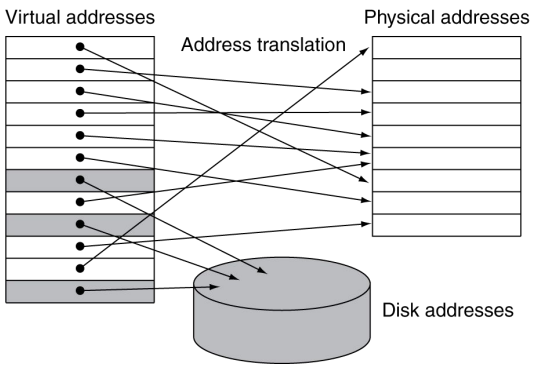


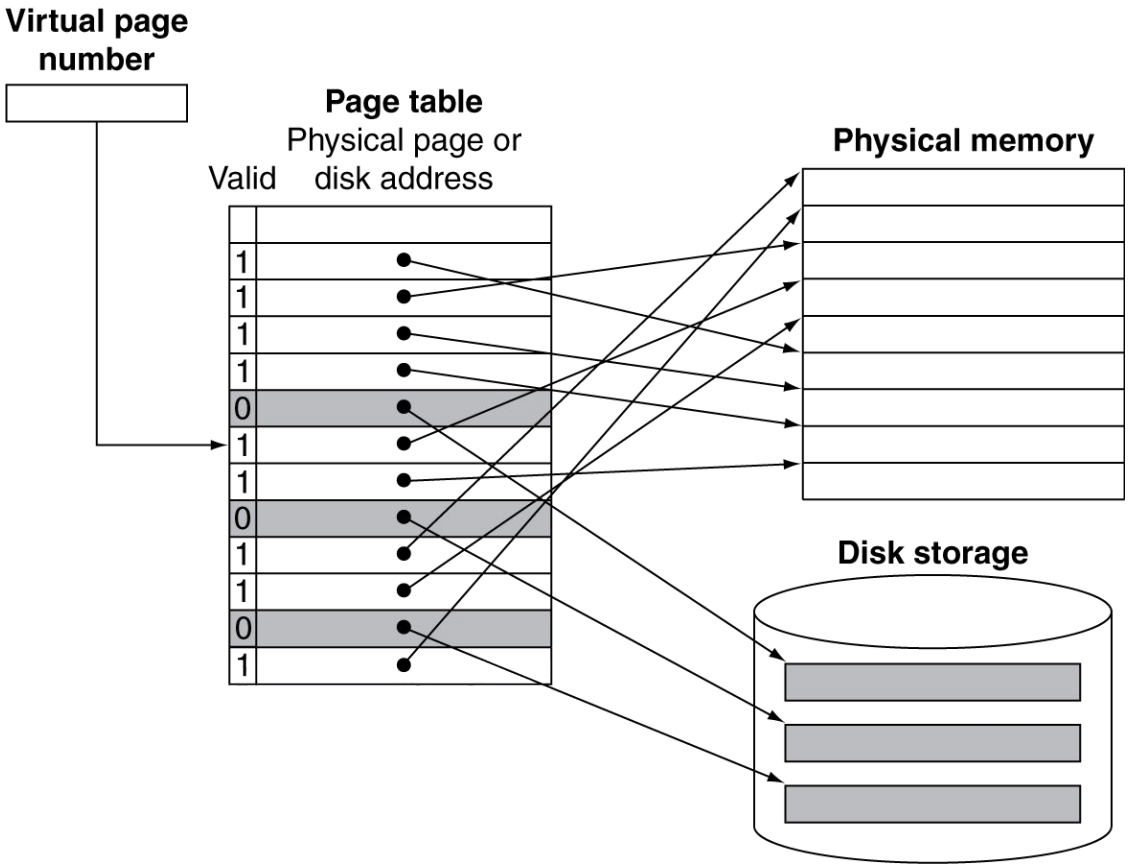
cache – memory থেকে → block

memory – hard disk থেকে → page

পড়ে আনে

Mapping Pages to Storage





Page Fault

কোনো একটা page যদি memory তে না থাকে — page fault
validbit 0 থাকে।
— OS কে control দিয়ে দিবে।

- If the valid bit for a virtual page is off, a page fault occurs.
- The operating system must be given control.
  - This transfer is done with the exception mechanism processor এর
  - OS must find the page in the next level of the hierarchy harddisk
  - and decide where to place the requested page in main memory.

Swap Space

processor এর যেসব page এ access থাকে ram এ
থাকে harddisk এও থাকে।

Swap Space

- Virtual addr. alone does not immediately tell us where the page is on disk.
- OS usually creates the space on flash memory/disk for all the pages of a process
- OS creates a data structure to record where each virtual page is stored on disk.
  - may be part of the page table
  - or an auxiliary data structure indexed in the same way as the page table
- OS also creates a data structure that tracks
  - which processes and which virtual addresses use each physical page

virtual address এর mapping থাকে harddisk and physical address এর সাথে।

Replacement and Writes

- To reduce page fault rate, prefer least-recently used (LRU) replacement
  - Reference bit (aka use bit) in PTE set to 1 on access to page
  - Periodically cleared to 0 by OS
  - A page with reference bit = 0 has not been used recently
- Disk writes take millions of cycles
  - Block at once, not individual locations
  - Write through is impractical
  - Use write-back
  - Dirty bit in PTE set when page is written

use bit = 1 (recently accessed)

0 হবে কখন? → OS করে। let's say 10 min পর পর করে দিবে। periodically tracking.

— ram and hard disk এ একসাথে update (use করা হয়না)

— dirty bit লাগবে।

— আগের page hard disk এ রেখে নতুন page নিয়ে আসতে হবে।

যদি মেমোরে dirty bit 1 থাকে

Translation Lookaside Buffer

Fast Translation Using a TLB

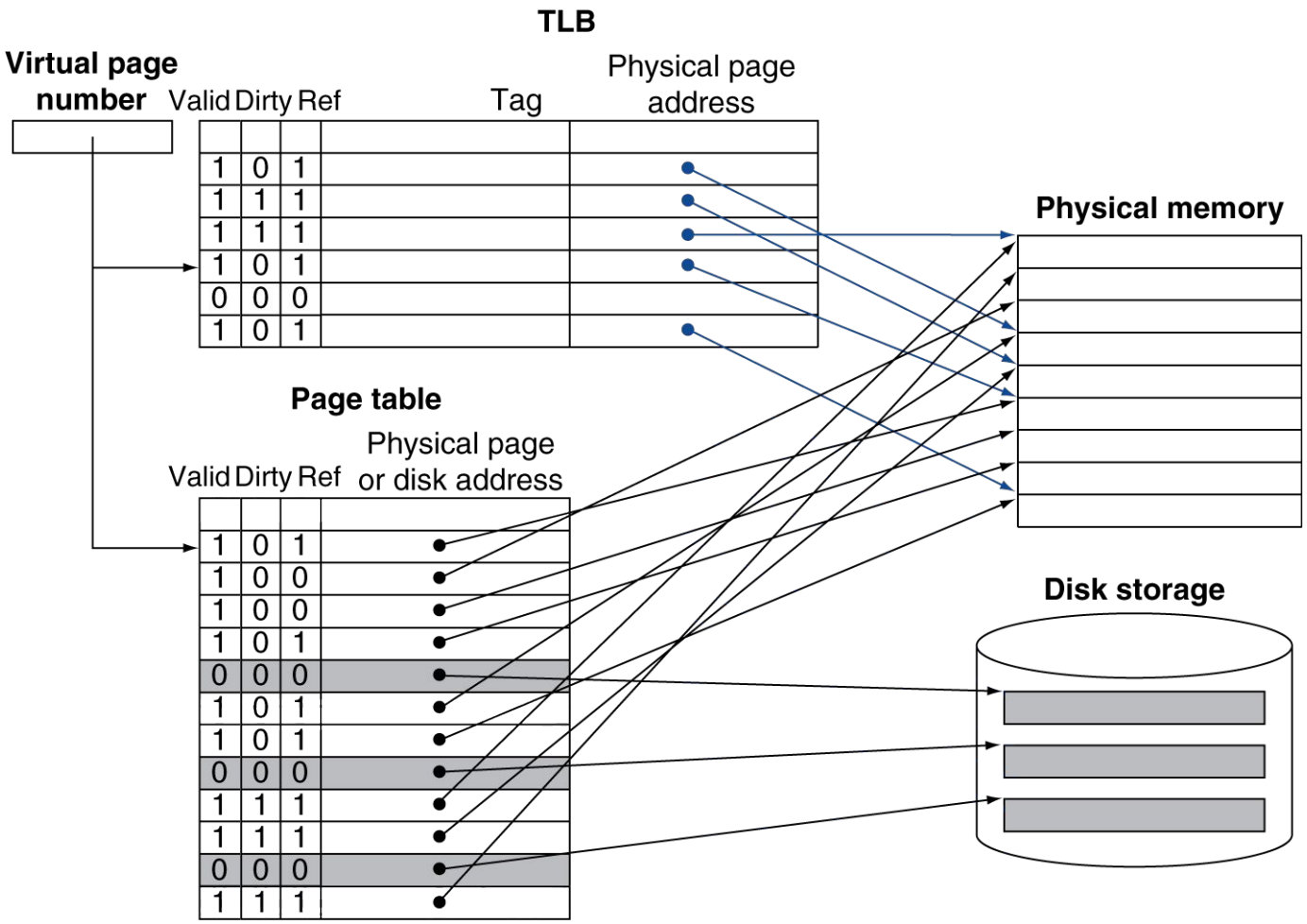
- Address translation would appear to require extra memory references
  - One to access the PTE
  - Then the actual memory access

Page Tables are in main memory

- But access to page tables has good locality
  - So use a fast cache of PTEs within the CPU
  - Called a Translation Look-aside Buffer (TLB)
  - Typical: 16–512 PTEs
  - Misses could be handled by hardware or software

TLB miss → corresponding page ram/harddisk এ থাকতে পারে।

Fast Translation Using a TLB



TLB Misses

- If page is in memory
  - Load the PTE from memory and retry
  - Could be handled in hardware
    - Can get complex for more complicated page table structures
  - Or in software (Kernel mode) OS করবে
    - Raise a special exception, with optimized handler
- If page is not in memory (page fault)
  - OS handles fetching the page and updating the page table
  - Then restart the faulting instruction

TLB Miss Handler

- TLB miss indicates
  - Page present, but PTE not in TLB
  - Page not present **True Page Fault**
- Handler copies PTE from memory to TLB
  - Then restarts instruction
  - If page not present, page fault will occur
- The reference and dirty bits may change in TLB
  - So they must also be copied back to PTE for the TLB entry that is replaced **Write-Back Scheme**

TLB: Associativity

Lower Miss Rate

- Some systems use small, fully associative TLBs

Not so high search cost

  - replacement choice becomes tricky
    - hardware LRU scheme is too expensive.
    - expensive software algorithm is also not feasible (unlike page faults)
    - Many systems provide some support for randomly choosing an entry to replace.

TLB misses are much more frequent

- Other systems use large TLBs, often with small associativity.

→ harddisk এ থাকে RAM এ নাই

Page Fault Handler

- Use faulting virtual address to find PTE
- Locate page on disk → os এর কাছে mapping থাকে
- Choose page to replace
  - If dirty, write to disk first
- Read page into memory and update page table
- Make process runnable again
  - Restart from faulting instruction

Intrinsity FastMATH TLB

- 4 KiB (2^{12}) pages; 32-bit address space → 12 bit page offset
  - So, virtual page number is (32 – 12 =) 20 bits long
  - Physical address is same size as virtual address. (32bit)
- TLB: 16 entries; fully associative
  - shared between the instruction and data
  - Each entry is 64 bits wide → data cache এ দিলে বসলে 64bit → 2 word
    - a 20-bit tag (virtual page number for that TLB entry)
    - the corresponding physical page number (also 20 bits),
    - a valid bit, a dirty bit, and other bookkeeping bits.
  - Like most MIPS systems, it uses software to handle TLB misses.

2 bit byte offset
1 bit word offset
∴ 29 bit tag

But TLB cache এ থাকলে না

এখানে 12 bit page offset
20 bit (indexing + tag)

∴ fully associative → no index bit so 20 bit tag.

TLB and Cache Interaction

CAM Implementation

TLB Cache যদি না থাকতো →

PT তে যেতে হবে। Virtual address থেকে physical address বানাতেই হবে (lw এ)। তখন 100 clock cycles লাগবে। যেটা 1/2 clock cycle এই হতে।

↓

memory access লাগে তাই—

2 bit byte offset

4 bit word } 12 bit দিয়ে word খুঁজে পাবো।

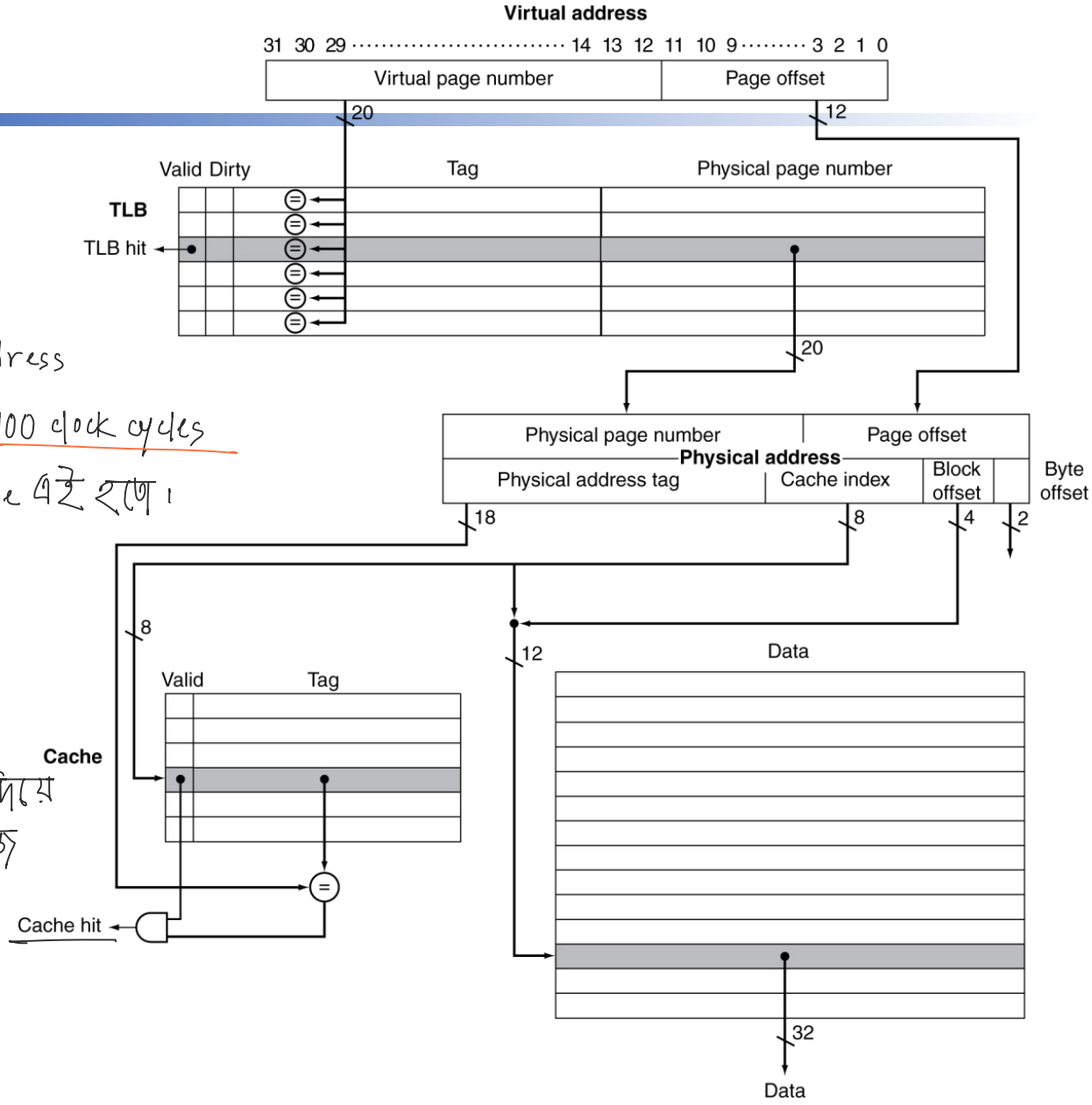
8 bit index

∴ tag 18 bit

Cache

2^8 2^4

256 blocks × 16 words/block



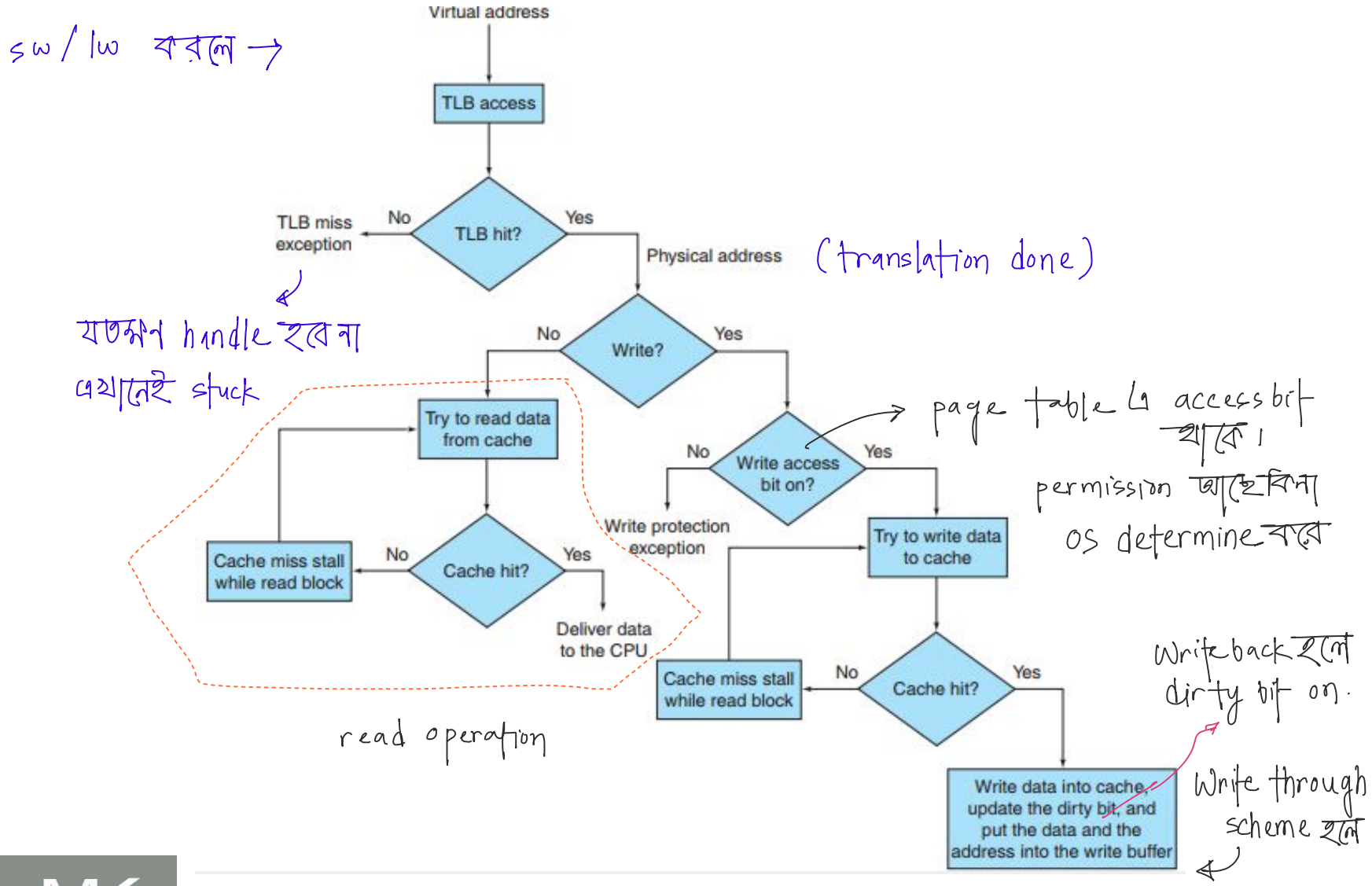
Content Addressable Memory (CAM)

- CAM is a circuit that combines comparison and storage in a single device.
  - Does not supply an address and read a word like a RAM
  - you supply the data and the CAM looks to see if it has a copy and returns the index of the matching row.
  - With CAMs higher set associativity in cache can be implemented

RAM → address ~~find~~ value return

CAM → value/ " address/ "
data index

Processing a read or a write-through in the Intrinsity FastMATH TLB and cache



TLB, Cache and VM Events Combined

Page ram এ, mapping TLB তে but
যে address এর data খুঁজতেছি cache এ আনা হয় নাই

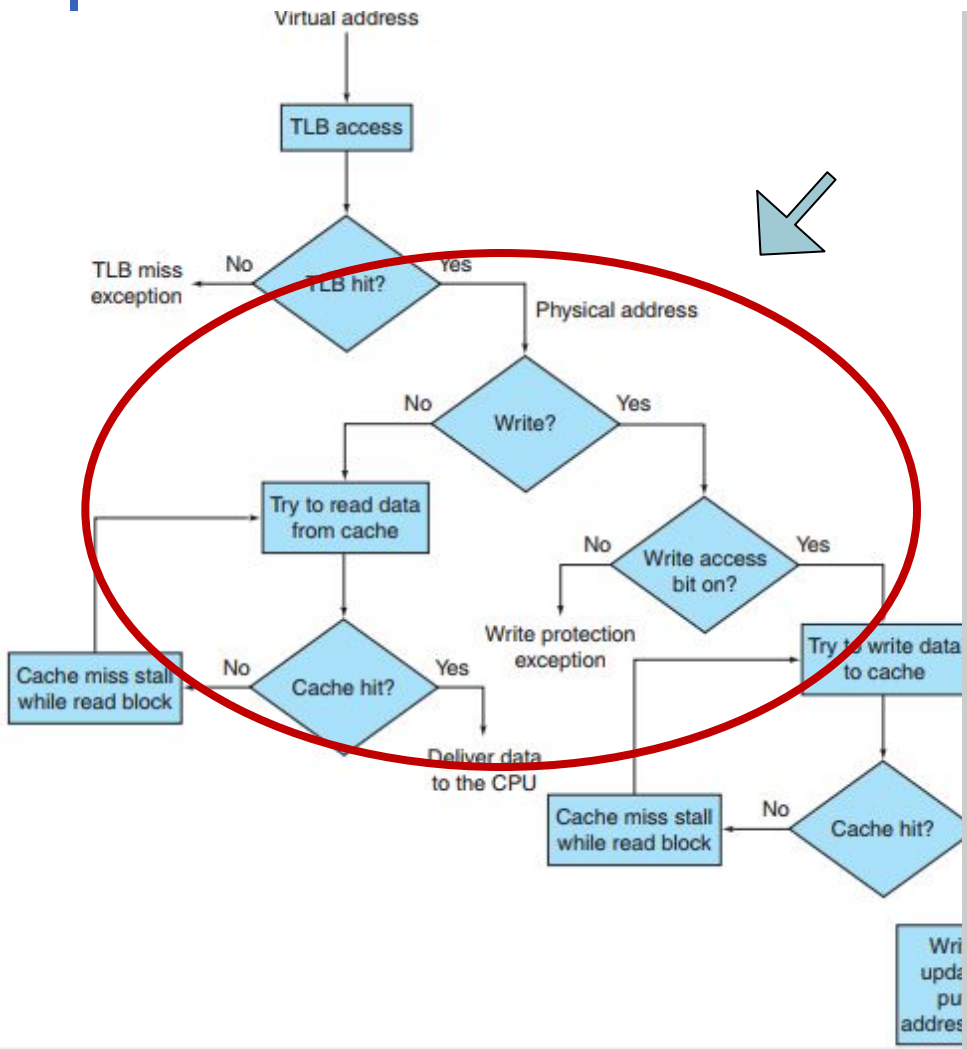
TLB	Page table	Cache	Possible? If so, under what circumstance?
Hit	Hit	Miss	Possible, although the page table is never really checked if TLB hits.
Miss	Hit	Hit	TLB misses, but entry found in page table; after retry, data is found in cache.
Miss	Hit	Miss	TLB misses, but entry found in page table; after retry, data misses in cache.
Miss	Miss	Miss	TLB misses and is followed by a page fault; after retry, data must miss in cache.
Hit	Miss	Miss	Impossible: cannot have a translation in TLB if page is not present in memory.
Hit	Miss	Hit	Impossible: cannot have a translation in TLB if page is not present in memory.
Miss	Miss	Hit	Impossible: data cannot be allowed in cache if the page is not in memory.

PT Hit → disk থেকে ram এ আনা হইছে যে page খুঁজতেছি

TLB miss হলে → PT Hit check করা হয়

hit " → PT check করা হয় না

Physically Addressed Cache



- Here, cache is "physically addressed" and "physically tagged"
- Time to access memory for a cache hit include:
  - TLB access time
  - Cache access time

  } Latency বাড়ে
- Of course, these accesses can be pipelined.

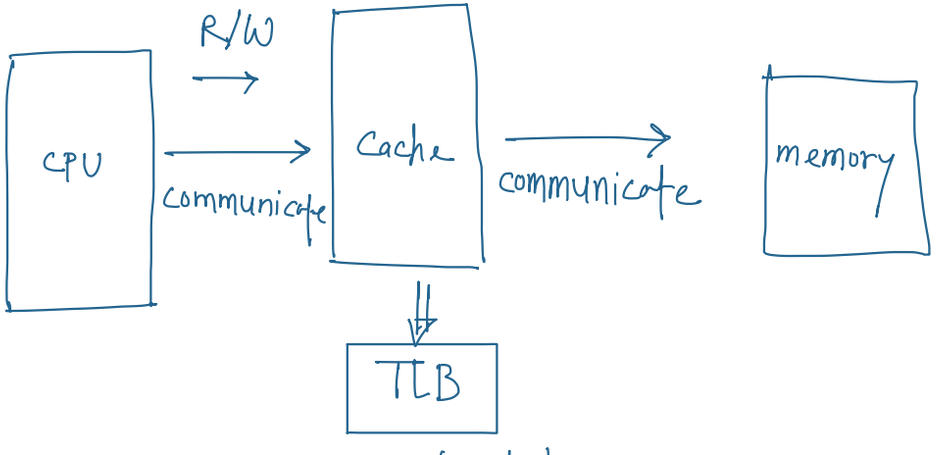
Virtually Indexed Cache

→ cache index করা হবে virtual address থেকে।
tag bit গুলোও virtual address থেকে আসবে।

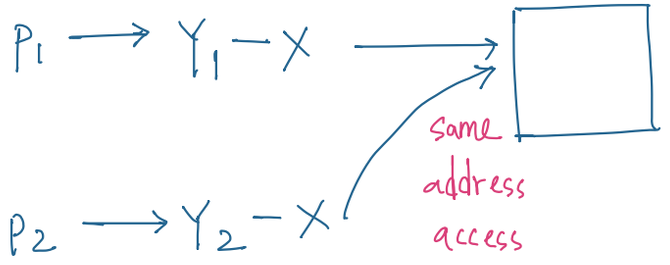
- Alternatively, the processor can index the cache with a virtual address
  - *Virtually indexed* and *Virtually tagged* cache

VIVT

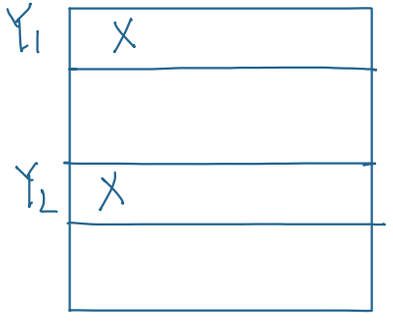
- Here, TLB is unused during the normal cache access
  - Reduce cache latency
- Cache miss=> the processor needs to translate the address to a physical address
  - so that it can fetch the cache block from main memory.



Cache এ data
না থাকলে mem
থেকে পড়ে আনা লাগবে। physical address লাগবে। আমাদের কাছে
virtual address আছে TLB সেটা physical address এ map করবে



Virtual Page



Problem:

P1 X ক update করলো।
cache ও হলো। But P2 এই reflection দেখতে পারবে না। — aliasing problem

virtually indexed

advantage — TLB লাগছে না

physical address লাগে নাই।

কিন্তু cache miss হলে TLB access করা লাগবে and memory থেকে data আনা লাগবে।

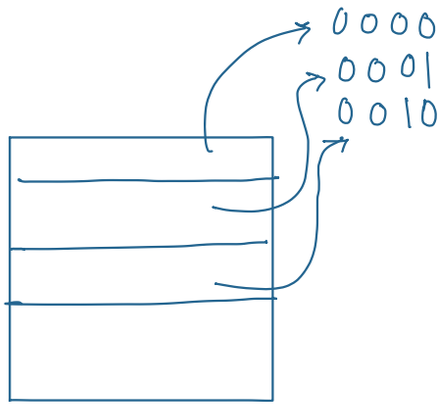
Soln: interprocess communication করতে দিবোনা।

Aliasing in VIVT Cache

- Aliasing occurs when the same object has two names
  - Two virtual addresses for the same page.
  - May happen for shared pages between processes
  - This ambiguity creates a problem:
    - A word on such a page may be cached in two different locations, each corresponding to different virtual addresses.
    - One program may write the data without the other program being aware that the data had changed.
- Solution to Aliasing Issues
  - either introduce design limitations on the cache and TLB to reduce aliases
  - or require the operating system, and possibly the user, to take steps to ensure that aliases do not occur

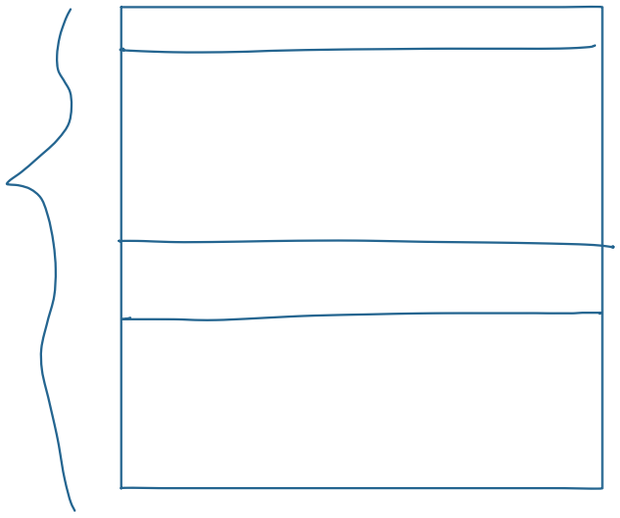
Virtually Indexed Physically Tagged

- Physical Tag using just the page-offset portion of the address, which is really a physical address since it is not translated
- These designs, which are *virtually indexed but physically tagged*, attempt to achieve the performance advantages of virtually indexed caches with the architecturally simpler advantages of a physically addressed cache.
  - There is no alias problem in this case.



4 bit indexing 0-15

8 bit 0-255 পর্যন্ত



indexing করে যাওয়ার পরে entry তে tag এর সাথে compare করা লাগে।

tag bit →

virtual address থেকে যে physical address পাবো সেটার upper bits

TLB access লাগবে

TLB access data cache access — in parallel হয়

tag bit offset

1 1 0 1 1 0 1 1 0 1 1 1

? 0 1 0 1 0 1 1 0 1 1 1

cache এর tag bit physical address এর tag bit match করে কিনা check.

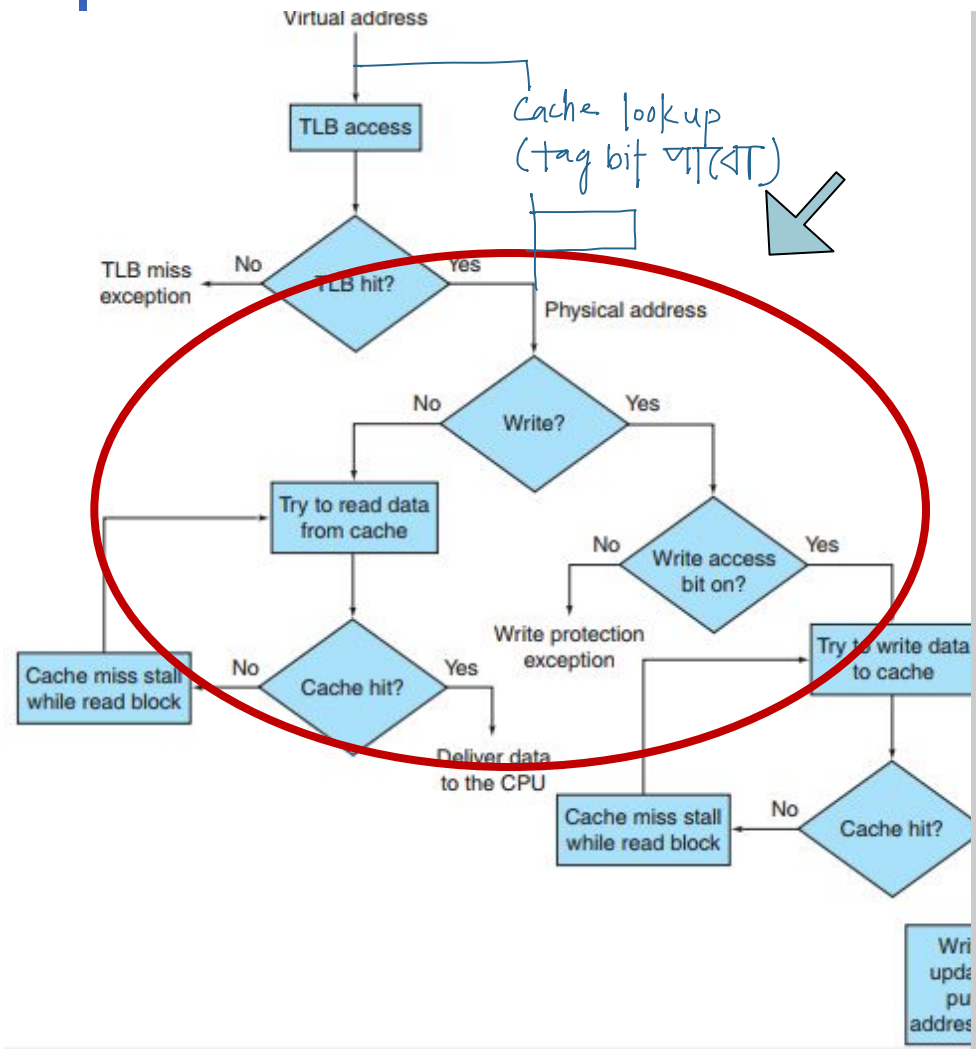
index virtual address থেকে tag physical থেকে।

তাই indexing পর্যন্ত TLB লাগে না

virtual address থেকে indexing আসা আলাদা কোনো meaning নেই।

indexing এর virtual address থেকে 8 bit নেয়া আর physical থেকে নেওয়া same। কোথাও store হয়না,

Physically Addressed Cache



- Here, cache is "physically addressed" and "physically tagged"
- Time to access memory for a cache hit include:
  - TLB access time
  - Cache access time

  } Latency বাড়ে
- Of course, these accesses can be pipelined.

Memory Protection

- Different tasks can share parts of their virtual address spaces
  - But need to protect against errant access
  - Requires OS assistance
- Hardware support for OS protection
  - Privileged supervisor mode (aka kernel mode)
  - Privileged instructions
  - Page tables and other state information only accessible in supervisor mode
  - System call exception (e.g., syscall in MIPS)

Memory Protection: H/W Suport

- Need to Support:
  - at least two modes:
    - Privileged supervisor mode (aka kernel mode)
    - User mode
  - Different Processor states:
    - Allow for a process to read only; No Write allowed
    - To write, privileged instructions are needed that are only available in supervisor mode
    - Page tables and other state information only accessible in supervisor mode
  - System call exception (e.g., syscall in MIPS):
    - This allows a process to change mode:
      - User mode => supervisor mode (and vice versa)

Page table এ access bit থাকে।

Memory Protection

- Each process has its own virtual address space.
- OS can keep the page tables organized so that the independent virtual pages map to disjoint physical pages
  - one process will not be able to access another's data.
- But what if the Process changes the mapping?
- Page tables are placed in the protected address space of the OS
  - So, user process is refrained from changing the page table mapping.
  - But OS is able to modify the page tables.

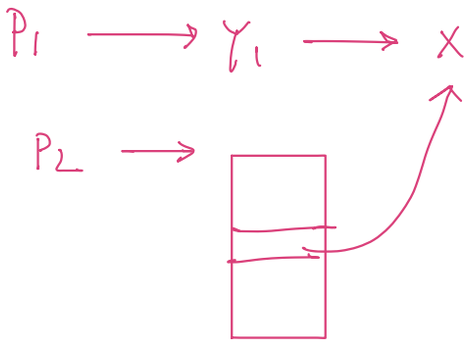
একটা process এর সাথে memory share করতে চাই

Memory Sharing and Protection

- OS assists processes for limited information sharing
  - OS can change the Page table as needed
  - The write access bit is used to restrict write
    - Can be changed only by OS

Example:

- P2 wants P1 to access its page
- P2 asks OS to create a page table entry for a virtual page in P1's address space that points to the same physical page that P2 wants to share.
- Any bits that determine the access rights for a page must be included in both the page table and the TLB because the page table is accessed only on a TLB miss.



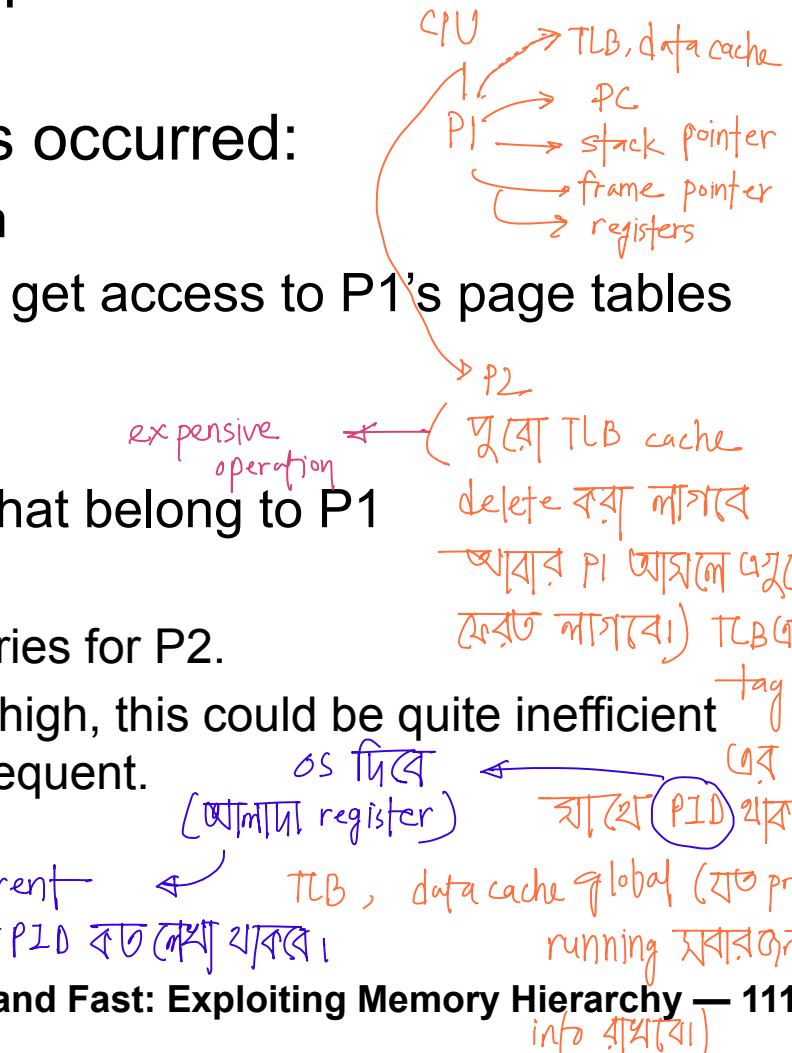
P1 X কে P2 এর সাথে share করতে চায়। P2 এর page table এ X এর জন্য ...

OS handle করবে।

OS middle ground

Memory Protection: Context Switch

- Context Switch: A changing of the internal state of the processor to allow a different process to use the processor
- Suppose a context switch has occurred:
  - P1 was running; now P2 will run
  - OS must ensure that P2 cannot get access to P1's page tables
    - Page Table register is changed
  - What about the TLB?
  - OS must clear the TLB entries that belong to P1
    - to protect the data of P1 and
    - to force the TLB to load the entries for P2.
    - If the process switch rate were high, this could be quite inefficient especially if context switch is frequent.



Thread বেশি হলে — context switching বেশি হয়।

data intensive operation বেশি হলে — single thread এ ভালো perform করবে।
কারন context switching এর সময় CPU কাজ না করে বসে থাকে

memory intensive operation বেশি (database এর সাথে communicate করা লাগে)
— তখন context switching বেশি হলেও prblm নেই। কারন CPU এমনই বসে থাকে।

Memory Protection: Context Switch

A common alternative:

- extend the virtual address space by adding a *process identifier* or *task identifier*.
- The Intrinsity FastMATH has an 8-bit address space ID (ASID) field for this purpose.
  - This small field identifies the currently running process
  - it is kept in a register loaded by OS when it switches processes.
  - The process identifier is concatenated to the tag portion of the TLB
  - TLB hit occurs only if both the page number *and* the process identifier match.
  - This eliminates the need to clear the TLB in context switch
- Similar problems can occur for a cache, since on a process switch the cache will contain data from the running process.

Exception Enable/Disable

- Suppose we have a page fault exception and OS is handling it
- What will happen, if a second exception occurs?
  - The control unit would overwrite the exception program counter, making it impossible to return to the instruction that caused the page fault!
- We need the ability to **disable** and **enable exceptions**.
- When an exception first occurs, the processor sets a bit that disables all other exceptions;
- this could happen at the same time the processor sets the supervisor mode bit.
- The OS will then save just enough state to allow it to recover if another exception occurs
  - the *exception program counter* (EPC) and Cause registers
  - The operating system can then re-enable exceptions.

Special control registers that help with exceptions, TLB misses, and page faults

Register	CP0 register number	Description
EPC	14	Where to restart after exception
Cause	13	Cause of exception
BadVAddr	8	Address that caused exception
Index	0	Location in TLB to be read or written
Random	1	Pseudorandom location in TLB
EntryLo	2	Physical page address and flags
EntryHi	10	Virtual page address
Context	4	Page table address and page number

FIGURE 5.33 MIPS control registers. These are considered to be in coprocessor 0, and hence are read using `mfc0` and written using `mtc0`.

When a TLB miss occurs, the MIPS hardware saves the page number of the reference here.

intel processor বিভাবে কাজ করে?
– self study.

TLB Miss in MIPS

- TLB Miss exception invokes OS, which handles the miss in software.
- Control is transferred to 8000 0000hex (TLB Miss Handler Address)
- To find the physical address for the missing page, the TLB miss routine indexes the page table using the page number of the virtual address and the page table register
- To make this indexing fast, MIPS hardware places the address of the Page Table Entry in a special Context Register
- Thus, the first two instructions copy the Context register into the kernel temporary register $k1 and then load the page table entry from that address into $k1.
- Recall that $k0 and $k1 are reserved for the operating system

TLB Miss in MIPS

```
TLBmiss:
   mfc0   $k1,Context    # copy address of PTE into temp $k1
   lw     $k1,0($k1)     # put PTE into temp $k1
   mtc0   $k1,EntryLo    # put PTE into special register EntryLo
   tlbwr                 # put EntryLo into TLB entry at Random
   eret                  # return from TLB miss exception
```

- TLB miss handler does not check to see if the page table entry is valid.
- If invalid, another and different exception occurs, and OS recognizes the page fault.
  - (frequent) TLB miss becomes fast
  - at a slight performance penalty for the (infrequent) page fault.

it transfers control to 8000 0180h

The Memory Hierarchy

The BIG Picture

- Common principles apply at all levels of the memory hierarchy
  - Based on notions of caching
- At each level in the hierarchy
  - Block placement
  - Finding a block
  - Replacement on a miss
  - Write policy

Block Placement

- Determined by associativity
  - Direct mapped (1-way associative)
    - One choice for placement
  - n-way set associative
    - n choices within a set
  - Fully associative
    - Any location
- Higher associativity reduces miss rate
  - Increases complexity, cost, and access time

Finding a Block

Associativity	Location method	Tag comparisons
Direct mapped	Index	1
n-way set associative	Set index, then search entries within the set	n
Fully associative	Search all entries	#entries
	Full lookup table	0

- Hardware caches
  - Reduce comparisons to reduce cost
- Virtual memory
  - Full table lookup makes full associativity feasible
  - Benefit in reduced miss rate

Replacement

- Choice of entry to replace on a miss
  - Least recently used (LRU)
    - Complex and costly hardware for high associativity
  - Random
    - Close to LRU, easier to implement
- Virtual memory
  - LRU approximation with hardware support

Write Policy

- Write-through
  - Update both upper and lower levels
  - Simplifies replacement, but may require write buffer
- Write-back
  - Update upper level only
  - Update lower level when block is replaced
  - Need to keep more state
- Virtual memory
  - Only write-back is feasible, given disk write latency

Sources of Misses

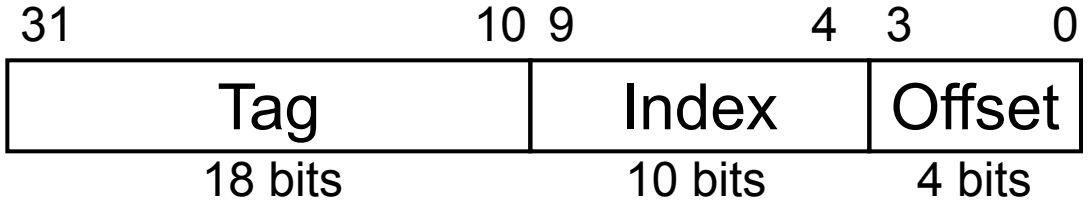
- Compulsory misses (aka cold start misses)
  - First access to a block
- Capacity misses
  - Due to finite cache size
  - A replaced block is later accessed again
- Conflict misses (aka collision misses)
  - In a non-fully associative cache
  - Due to competition for entries in a set
  - Would not occur in a fully associative cache of the same total size

Cache Design Trade-offs

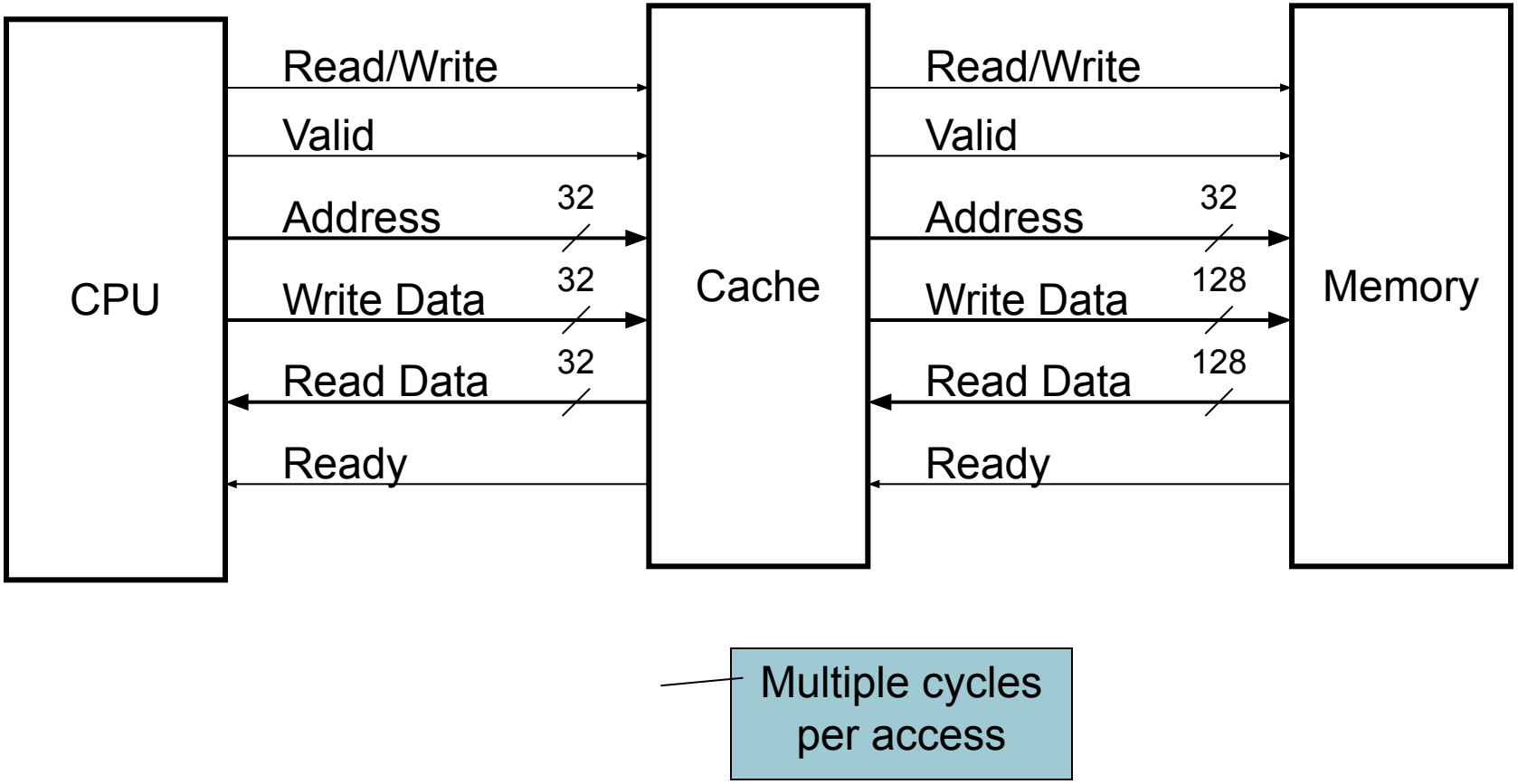
Design change	Effect on miss rate	Negative performance effect
Increase cache size	Decrease capacity misses	May increase access time
Increase associativity	Decrease conflict misses	May increase access time
Increase block size	Decrease compulsory misses	Increases miss penalty. Very large block could increase miss rate.

Cache Control

- Example cache characteristics
  - Direct-mapped, write-back, write allocate
  - Block size: 4 words (16 bytes)
  - Cache size: 16 KB (1024 blocks)
  - 32-bit byte addresses
  - Valid bit and dirty bit per block
  - Blocking cache
    - CPU waits until access is complete

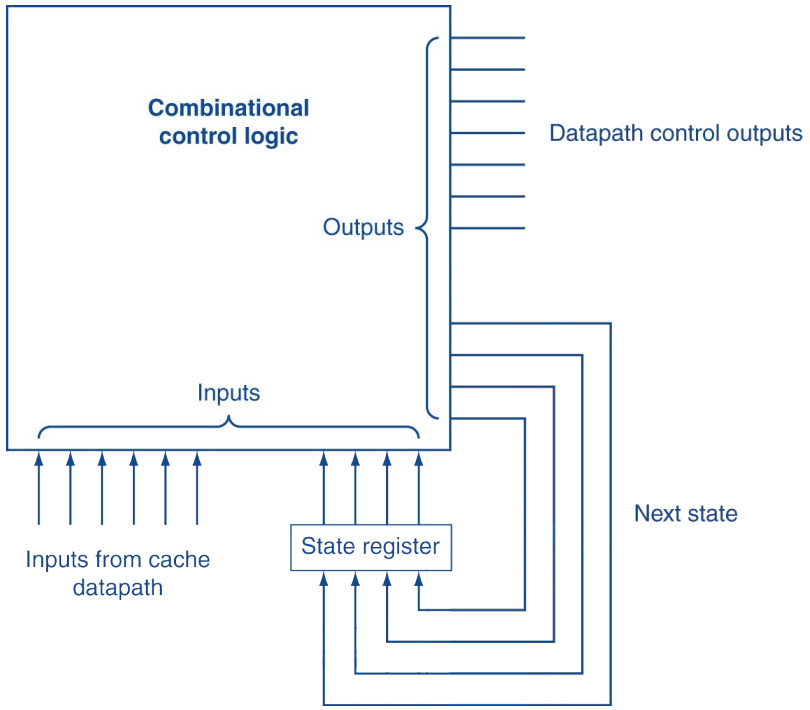


Interface Signals

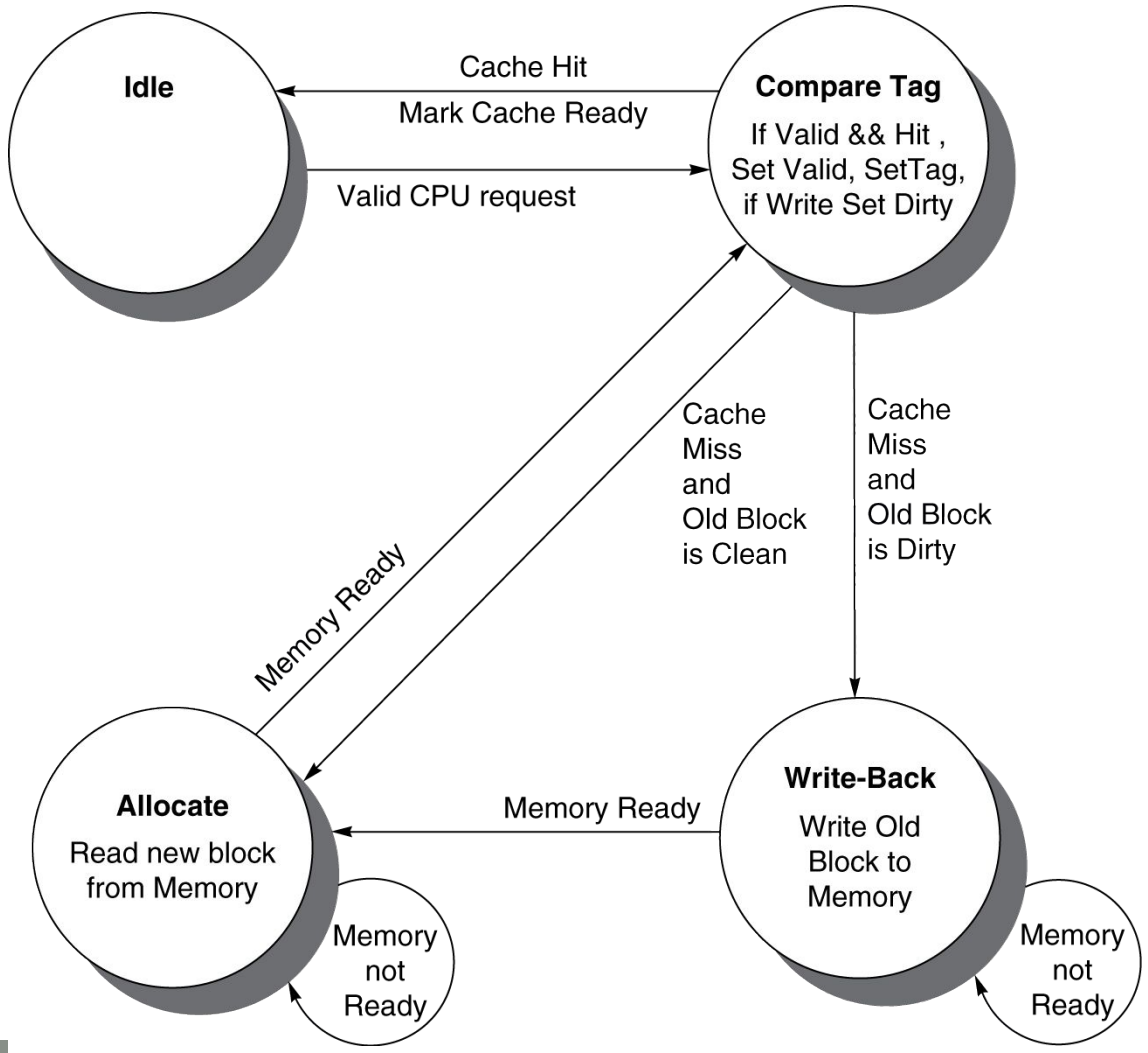


Finite State Machines

- Use an FSM to sequence control steps
- Set of states, transition on each clock edge
  - State values are binary encoded
  - Current state stored in a register
  - Next state = f_n (current state, current inputs)
- Control output signals = f_o (current state)



Cache Controller FSM



Could partition into separate states to reduce clock cycle time