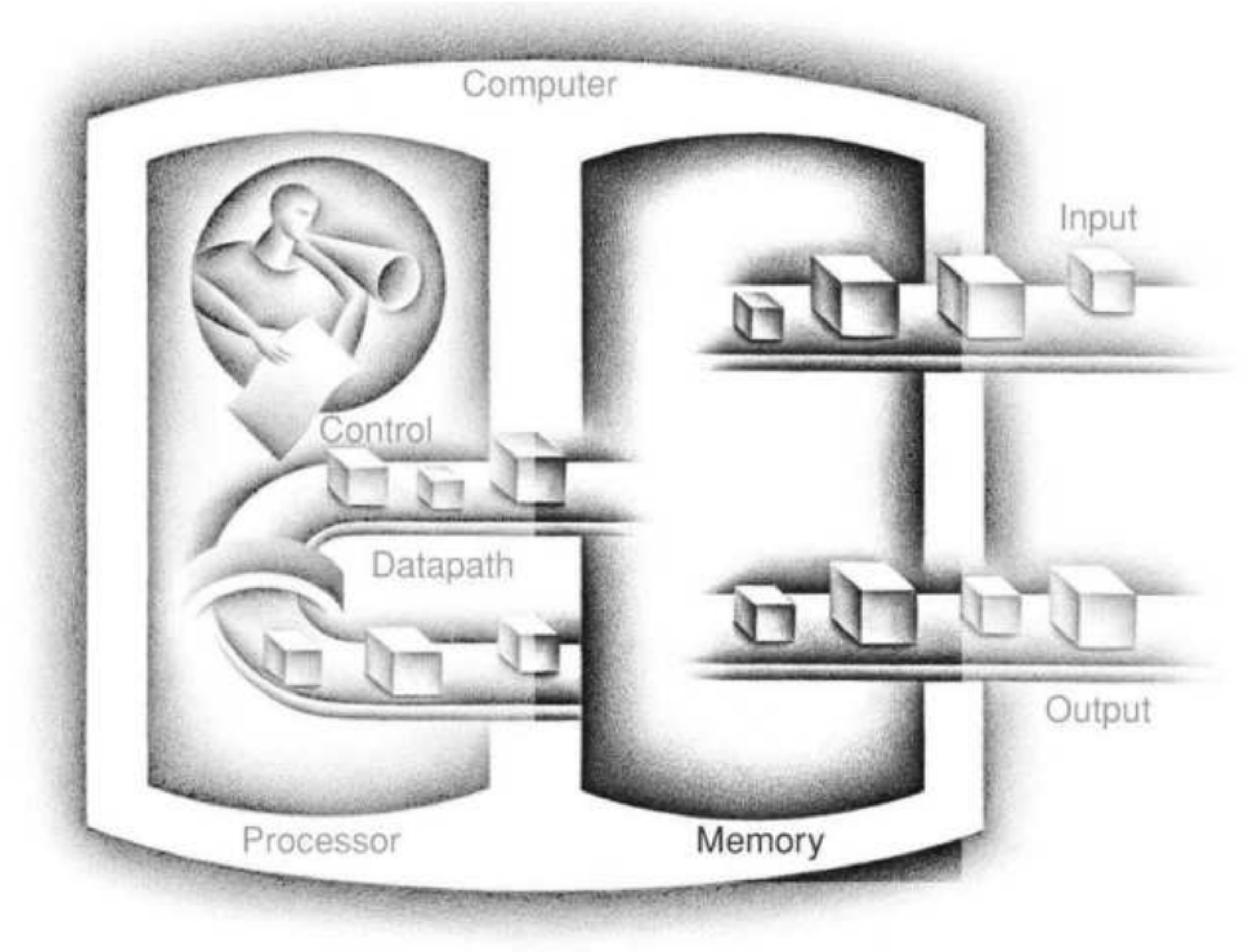
Computer Memory

Dr. S Pal, CSE, IIT(BHU)

Slide Sources: Patterson & Hennessy COD book site (copyright Morgan Kaufmann) adapted and supplemented

Five basic components of a computer



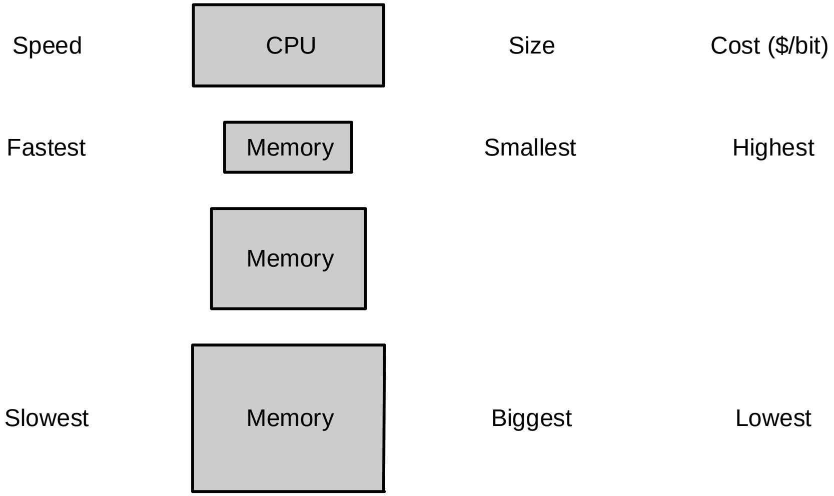
Memories: Review

SRAM (Static Random Access Memory):

- value is stored on a pair of inverting gates that will *exist indefinitely* as long as there is power, which is why it is called *static*
- very fast but takes up more space than DRAM – 4 to 6 transistors per bit
- used for *cache*

DRAM (Dynamic Random Access Memory):

- value is stored as a charge on capacitor that must be *periodically refreshed*, which is why it is called *dynamic*
- very small – 1 transistor per bit – but factor of 5 to 10 slower than SRAM
- used for *main memory*



Speed CPU Size Cost ($/bit)

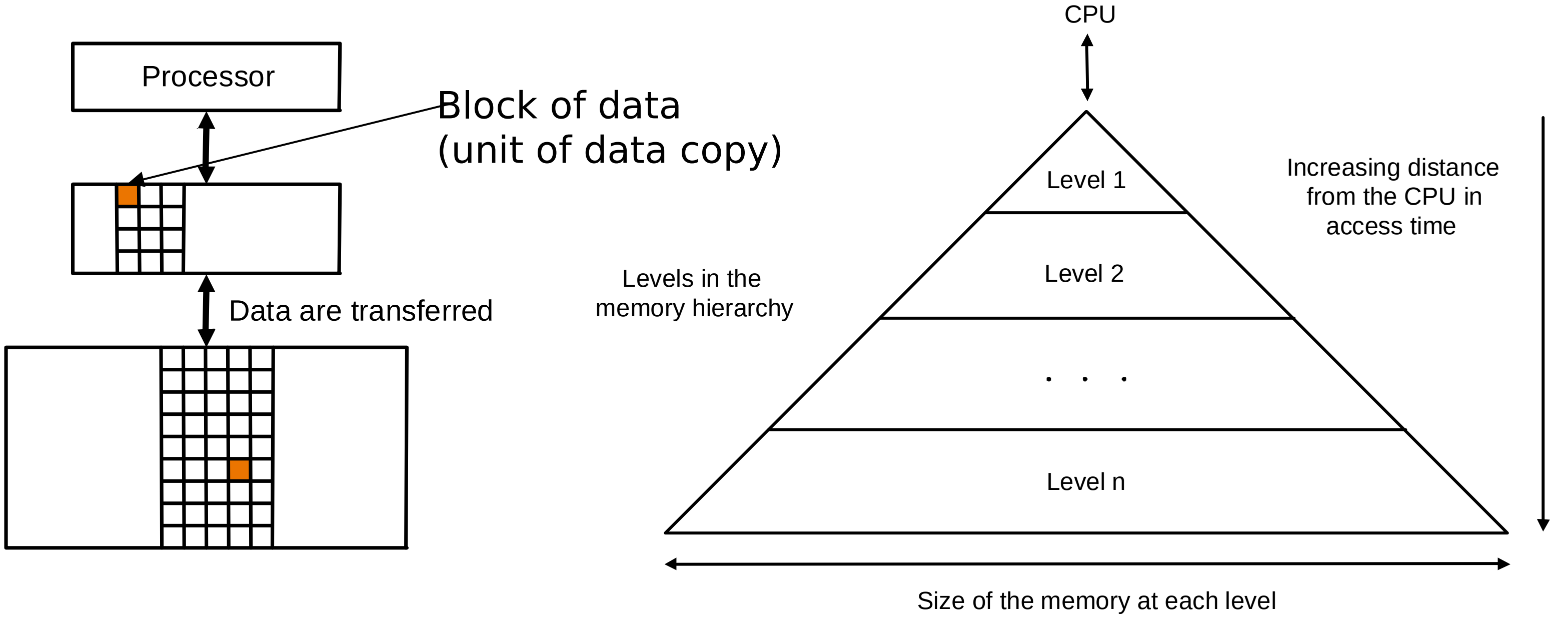
Fastest Memory Smallest Highest

Memory

Slowest Memory Biggest Lowest

Memory Hierarchy

- Users want **large** and **fast memories**…
  - expensive and they don’t like to pay…
- Make it seem like they have what they want…
  - *memory hierarchy*
  - hierarchy is *inclusive*, every level is *subset* of lower level
  - performance depends on *hit rates*


Processor
Block of data
(unit of data copy)
Data are transferred
CPU
Level 1
Level 2
. . .
Level n
Levels in the memory hierarchy
Increasing distance from the CPU in access time
Size of the memory at each level

Locality

- *Locality* is a principle that makes having a memory hierarchy a good idea
- If an item is referenced then because of
  - *temporal locality*: it will tend to be *again* referenced soon
  - *spatial locality*: *nearby items* will tend to be referenced soon
  - *why does code have locality – consider instruction and data?*

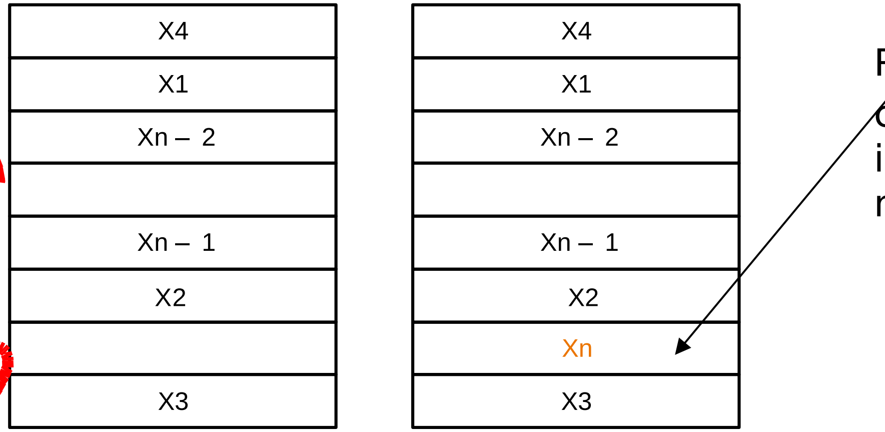
Hit and Miss

- Focus on *any two adjacent* levels – called, *upper* (closer to CPU) and *lower* (farther from CPU) – in the memory hierarchy, because each block copy is always between two adjacent levels
- Terminology:
  - *block*: minimum unit of data to move between levels
  - *hit*: data requested is available in upper level
  - *miss*: data requested is not available in upper level
  - *hit rate*: fraction of memory accesses that are hits (i.e., found at upper level)
  - *miss rate*: fraction of memory accesses that are not hits
    - miss rate = 1 – hit rate
  - *hit time*: time to determine if the access is indeed a hit + time to access and deliver the data from the upper level to the CPU
  - *miss penalty*: time to determine if the access is a miss + time to replace block at upper level with corresponding block at lower level + time to deliver the block to the CPU

Caches

P

- By simple example
  - assume block size = one word of data



a. Before the reference to Xn

b. After the reference to Xn

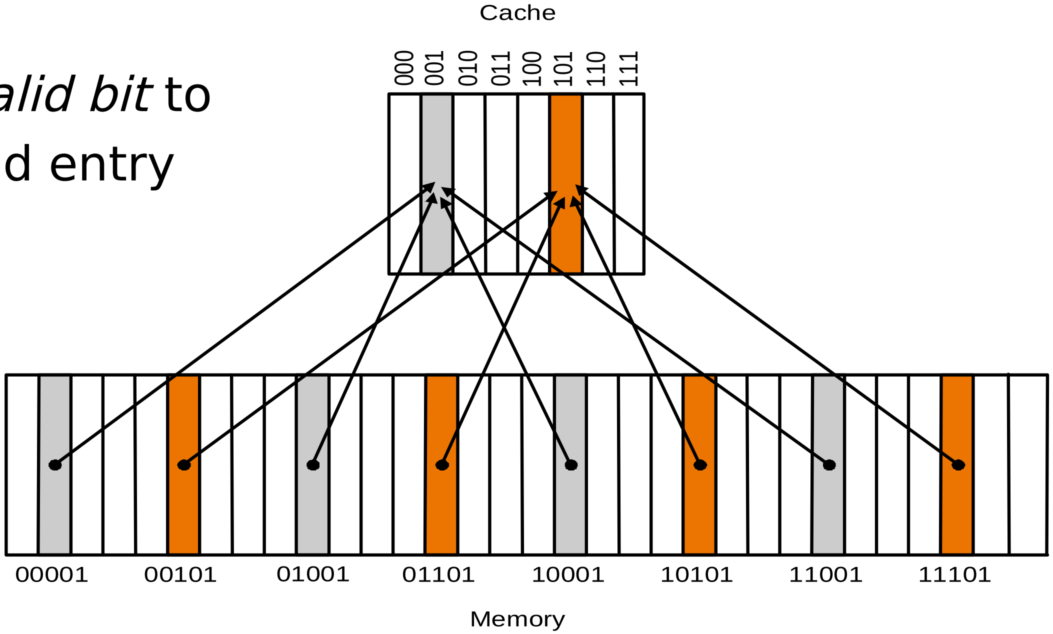
Reference to X_n causes miss so it is fetched from memory

load

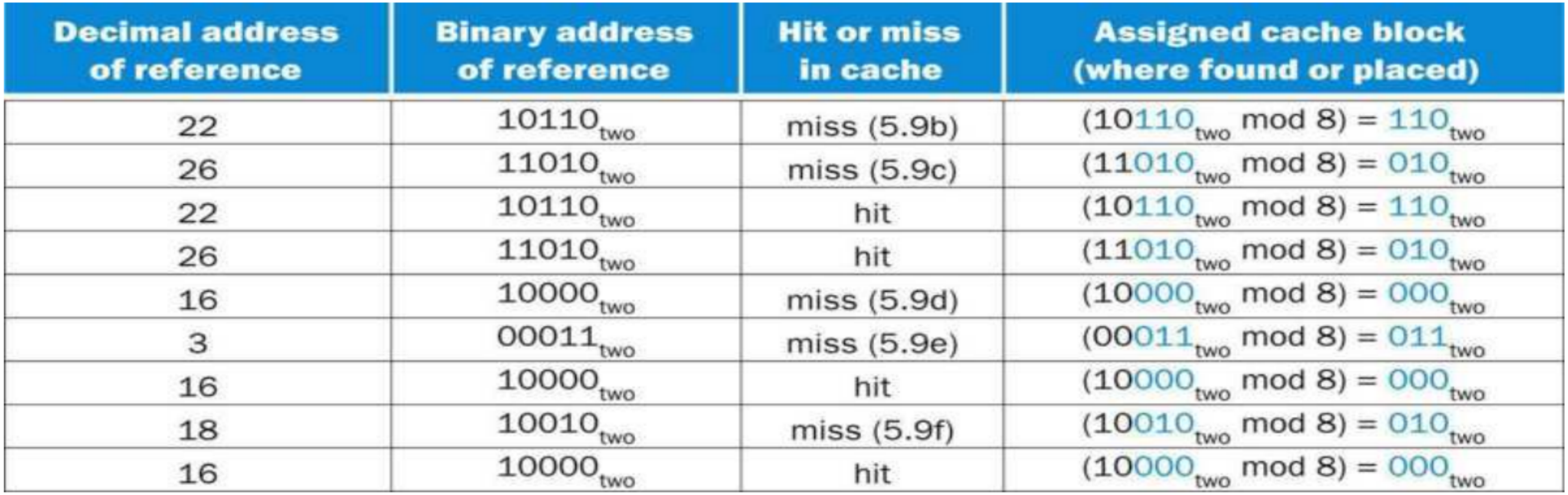
- Issues:
  - *how do we know if a data item is in the cache?*
  - *if it is, how do we find it?*
  - *if not, what do we do?*
- Solution depends on *cache addressing scheme…*

Direct Mapped Cache

- Addressing scheme in *direct mapped* cache:
  - cache block address = memory block address *mod* cache size (*unique*)
  - if cache size = 2^m, cache address = lower **m** bits of **n**-bit memory address
  - remaining upper **(n-m)** bits kept kept as *tag bits* at each cache block
  - also need a *valid bit* to recognize valid entry

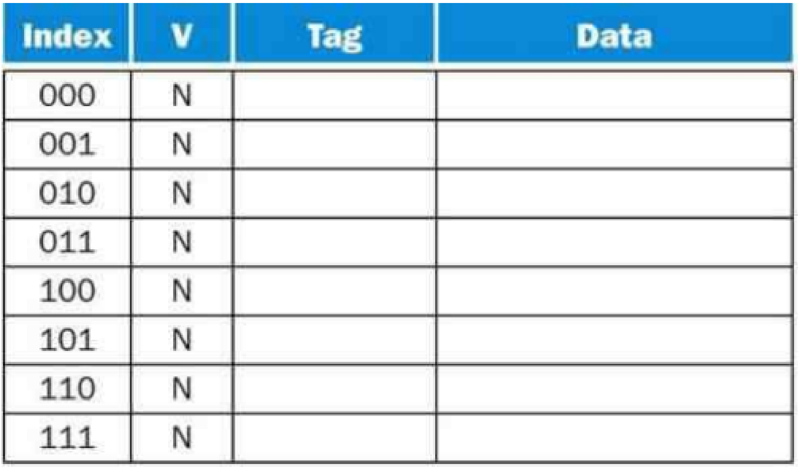


Accessing Cache



Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss (5.9b)	$(10110_{two}$ mod 8) = 110_{two}
26	11010_{two}	miss (5.9c)	$(11010_{two}$ mod 8) = 010_{two}
22	10110_{two}	hit	$(10110_{two}$ mod 8) = 110_{two}
26	11010_{two}	hit	$(11010_{two}$ mod 8) = 010_{two}
16	10000_{two}	miss (5.9d)	$(10000_{two}$ mod 8) = 000_{two}
3	00011_{two}	miss (5.9e)	$(00011_{two}$ mod 8) = 011_{two}
16	10000_{two}	hit	$(10000_{two}$ mod 8) = 000_{two}
18	10010_{two}	miss (5.9f)	$(10010_{two}$ mod 8) = 010_{two}
16	10000_{two}	hit	$(10000_{two}$ mod 8) = 000_{two}

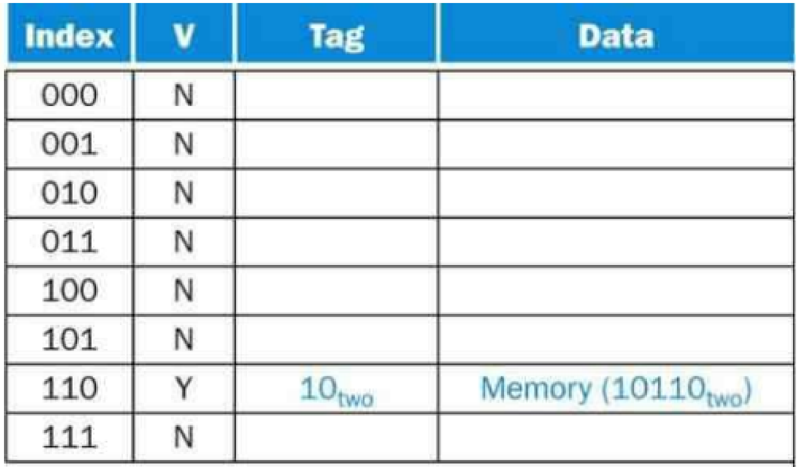
(0) Initial state:



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

a. The initial state of the cache after power-on

(1) Address referred 10110 (*miss*):



Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

b. After handling a miss of address (10110_{two})

Index	V	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

c. After handling a miss of address (11010_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

d. After handling a miss of address (10000_{two})

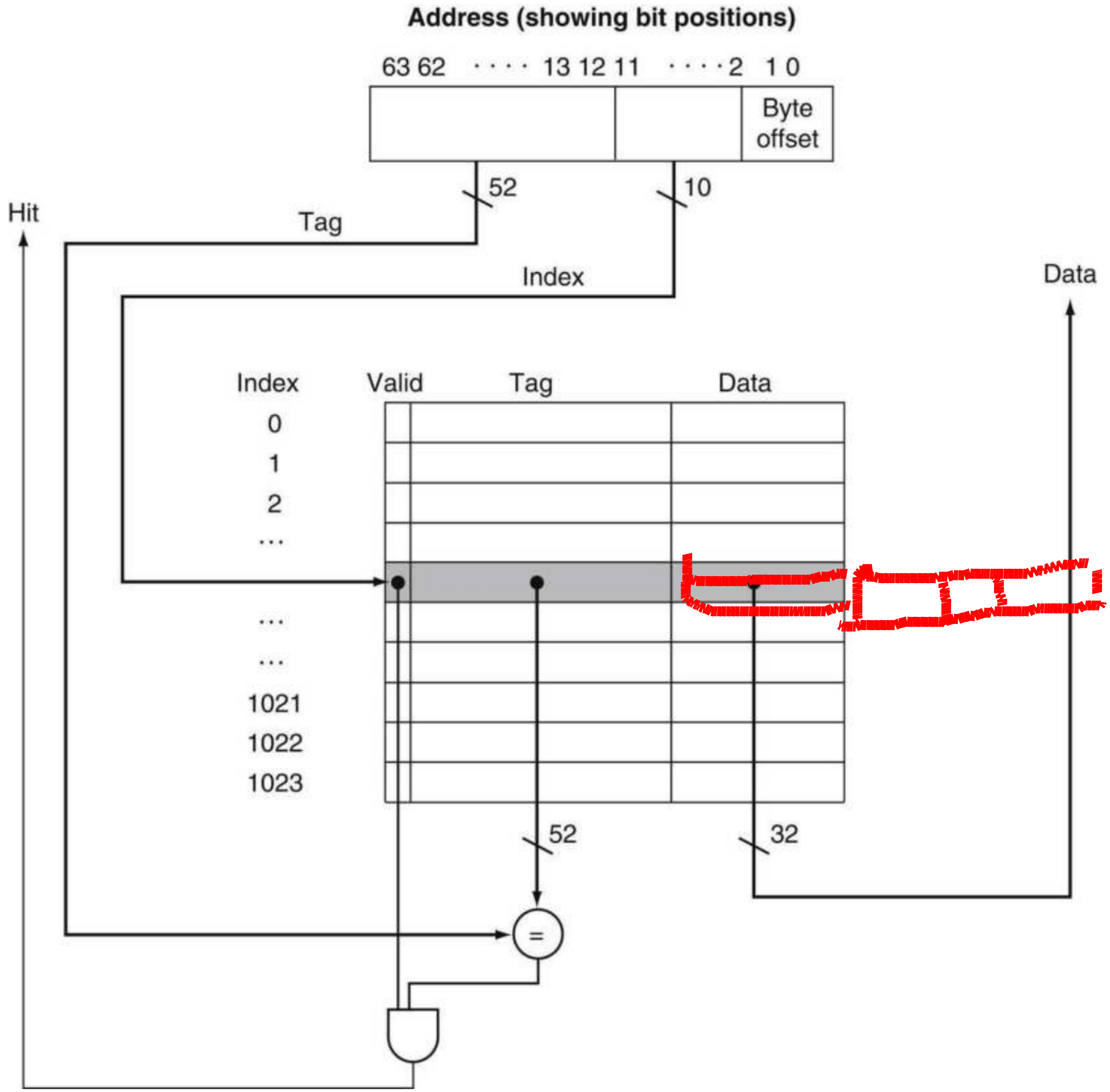
Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

e. After handling a miss of address (00011_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	10_{two}	Memory (10010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

f. After handling a miss of address (10010_{two})

Direct Mapped Cache



Cache with 1024 1-word blocks: *byte offset* (least 2 significant bits) is ignored and next 10 bits used to index into cache

Cache arithmetic

- 64-bit addresses
- A direct-mapped cache
- The cache size is ***2^n*** blocks, so ***n*** bits are used for the index
- The block size is ***2^m*** words (***2^{m+2}*** bytes), so ***m*** bits are used for the word within the block, and 2 bits are used for the byte part of the address
- The size of the tag field is = ***64 - (n+m+2)***
- Total #bits $2^n \times (\text{block size} + \text{tag size} + \text{valid field size})$

  $= \quad 2^n \times (2^m \times 32 + (64 - n - m - 2) + 1) = 2^n \times (2^m \times 32 + 63 - n - m).$

- But *tag, valid* bits are ignored when cache size is referred (here, 4KB cache)

Example: How many total bits are required for a direct-mapped cache with 16 KiB of data and 1-word blocks, assuming a 64-bit address?

Ans: 16 KiB is $2^4 x 2^{10}$ bytes = 2^{12} words

With a block size of 1 word, there are 4096 (2^{12}) blocks

Each block has 1 word = 32 bits of data plus a tag, which is (64 −12 −2) bits, plus a valid bit.

Thus, the complete cache size is

= 2^{12} x {32 + (64-12-2) +1} = 2^{12} x83 bits

= **33.2 KB for a 16 KB** cache.

Example: How many total bits are required for a direct-mapped cache with 16 KiB of data and four-word blocks, assuming a 64-bit address?

Ans: 16 KiB is 4096 (2^{12}) words

With a block size of 4 words (2^2), there are 1024 (2^{10}) blocks

Each block has 4 ×32 = 128 bits of data plus a tag, which is (64 −10 −2 −2) bits, plus a valid bit.

Thus, the complete cache size is

= 2^{10} x {4x32 + (64-10-2-2) +1} = 2^{10} x179 bits

= **22.4 KB for a 16 KB** cache.

Consider a cache with 64 blocks and a block size of 16 bytes. To what block number does byte address 1200 map?

- address of the block is = *floor* [(Byte address) / (Bytes per block)] = floor(1200/16) = 75
- Block-id in the cache = (Block address) modulo (number of blocks in the cache)
- 75 mod 64 = 11th block
- 1200, 1201, ...1215

- block address is the block containing all addresses between *floor* [Byte address / Bytes per block] * Bytes per block + (no. of bytes in a block - 1)

Example Problem

- *Consider a cache with 64 blocks and a block size of 16 bytes. What block number does byte address 1200 map to?*

- As block size = 16 bytes:
  byte address 1200 ⇒ block address $\lfloor 1200/16 \rfloor$ = 75
- As cache size = 64 blocks:
  block address 75 ⇒ cache block (75 *mod* 64) = 11

Cache Read Hit/Miss

- *Cache read hit*: no action needed
- *Instruction cache read miss*:
  1. *Send original PC value* (*current PC – 4*, as PC has already been incremented in first step of instruction cycle) to memory
  2. Instruct main memory to perform read and wait for memory to complete access – *stall* on read
  3. After read completes *write cache* entry
  4. *Restart* instruction execution at first step to refetch instruction
- *Data cache read miss*:
  - Similar to instruction cache miss
  - To reduce data miss penalty allow processor to execute instructions while waiting for the read to complete *until* the word is required – *stall on use* (why won't this work for instruction misses?)

Cache Write Hit/Miss

- *Write-through* scheme
  - on *write hit*: replace data in cache *and* memory with *every* write hit to avoid *inconsistency*
  - on *write miss*: write the word into cache *and* memory – obviously no need to read missed word from memory!
  - Write-through is slow because of always required memory write
    - performance is improved with a *write buffer* where words are stored while waiting to be written to memory – processor can continue execution until write buffer is full
    - when a word in the write buffer completes writing into main that buffer slot is freed and becomes available for future writes
    - DEC 3100 write buffer has 4 words
- *Write-back* scheme
  - write the data block *only* into the cache and *write-back* the block to main *only when* it is replaced in cache
  - more efficient than write-through, more complex to implement