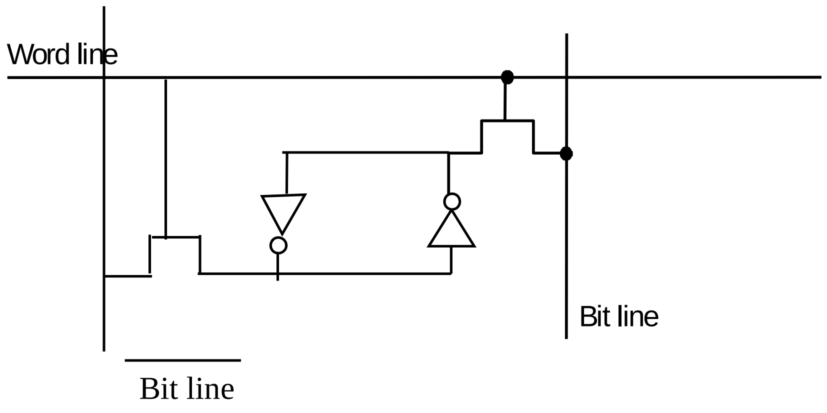
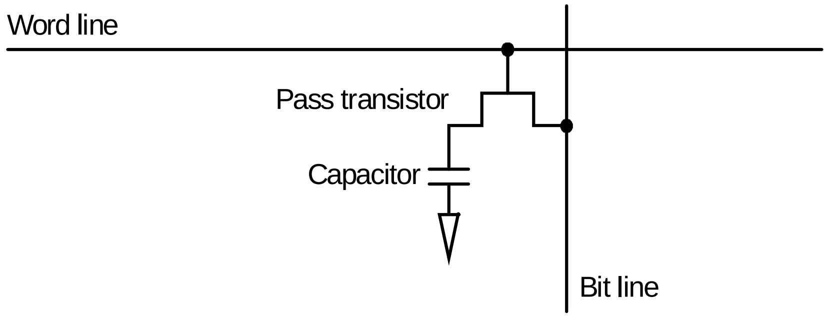
Chapter 5 Memory Hierarchy--Outline

- Basic idea of memory hierarchy
- Principle of locality
- Cache
- Virtual memory

Memories: Review

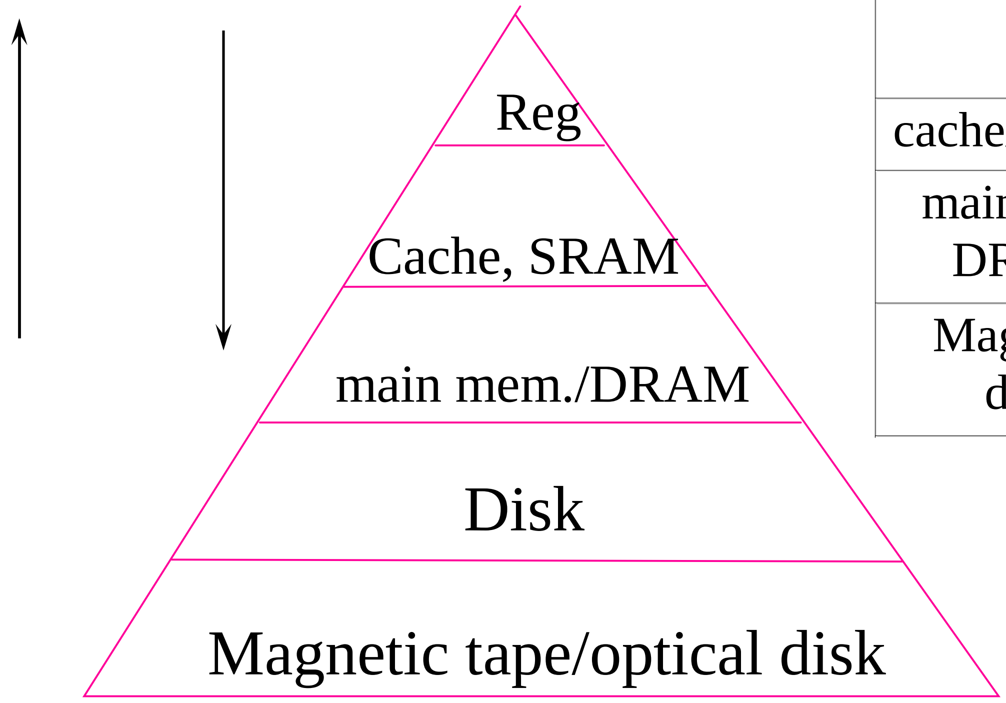
- SRAM:
  - value is stored on a pair of inverting gates
  - very fast but takes up more space than DRAM (4 to 6 transistors)
- DRAM:
  - value is stored as a charge on capacitor (must be refreshed)
  - very small but slower than SRAM (factor of 5 to 10)





Memory Hierarchy

- Multi-levels of memories with different speeds & sizes



	Access time	$/Gbyte in 2008
cache/SRAM	0.5~2.5 ns	$2k~5k
main mem DRAM	50~70 ns	$20~75
Magnetic disk	5-20 million ns	$.2~$2

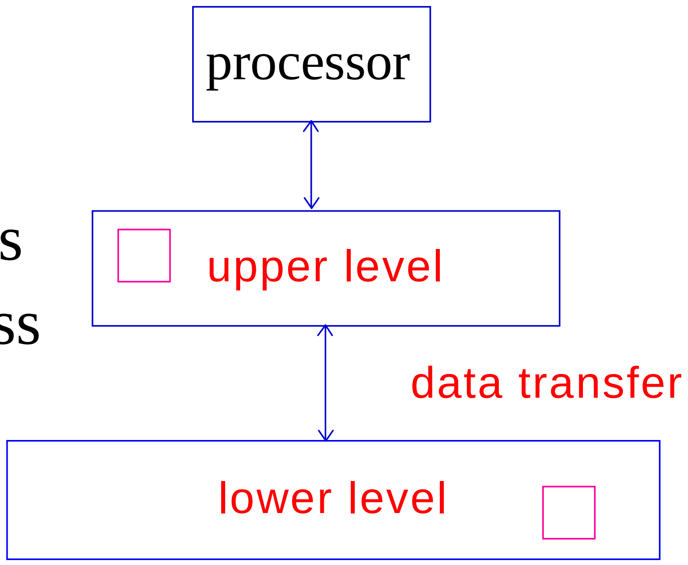
- Upper level & lower level
- any upper level $\subset$ lower level

Why Memory Hierarchy?

- Processor-DRAM performance gap growing
- Two goals (speed and size) conflict
- Principle of locality: program access a relative small portion of their address space --- if an item is referenced
  - it will tend to be referenced again soon --temporal locality
  - nearby items will tend to be referenced soon-- spatial locality
- How memory hierarchy works
- Goal of memory hierarchy: to achieve
  - speed of highest level (cache)
  - size of lowest level (disk)

Terminology

- How a processor access a memory hierarchy
  - access upper level first
  - if a miss, retrieve the whole block containing the requested data
- Block (line) a set of consecutive words in memory with the same block address
- Hit
- Miss
- Hit rate = #hit/#access
- Miss rate = 1 - hit rate
- miss penalty: time (transfer the block to upper level) + time (delivery data to processor)
- Why transfer the whole block instead of the requested data ?



Big Picture

- Principle of Locality
  - temporal
  - spatial
- Memory hierarchy
  - keep recently accessed data closer to processor --- temporal
  - moving blocks consisting of multiple contiguous words to upper level --- spatial
- Goal of memory hierarchy
  - memory access time $\rightarrow$ highest (fastest) level
  - memory size $\rightarrow$ lowest (largest) level

Four Questions in Designing Memory Hierarchy

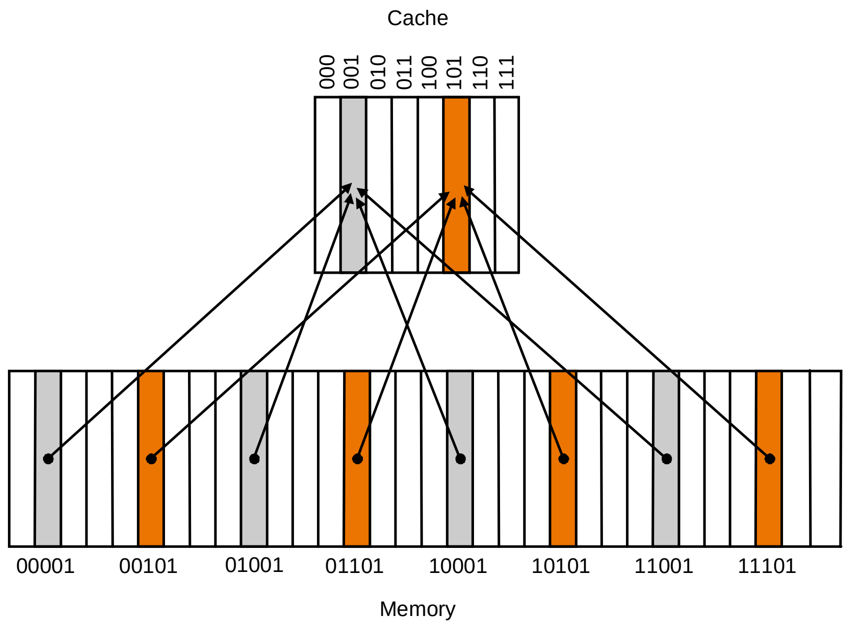
- Address mapping from lower level to upper level?
- How to find the requested data in the upper level?
- Which block to be replaced?
- Write?

Cache: interface b/w cache & main memory

- direct mapped cache
- fully associative cache
- set associative cache
- "direct mapped"
  - **For each item at the lower level, there is exactly one location in the cache where it might be.**
  - **e.g., lots of items at the lower level share locations in the upper level**

Direct Mapped Cache

- Mapping: address is modulo the number of blocks in the cache

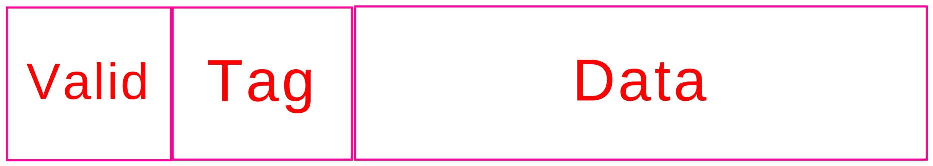


Block size =1

Cache with Direct Address Mapping

- Lower-order 3 bits of M. address = C. index
  - Memory locations with the same lower-order 3 bits share the same cache location– due to spatial locality
- The block in cache to be replaced is fixed
- How do I know if the data is what I want?
  - Tags: consist of higher-order bits of memory addresses

Format of a Word in Cache



- Valid = 1, valid address
  0, this cache location is empty
- Tag: remaining bits (higher order) of memory address
- Example :
  memory size = 2^5
  cache size = 2^3
  tags: 5-3=2 bits

- When CPU accesses cache, check both valid and tag

Example, block size = 1 word

- memory size = 32 blocks, cache size = 8 blocks
- Access sequence: 22, 26, 22, 18, 16, 18, 16, 18

Decimal address	Binary address	Hit/miss	Cache index
22			
26			
22			
18			
16			
18			
16			
18			

index	V	tag	data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Cache content

Block size =2

- How to form blocks: memory locations with the same block address
- Block offset:

Example 2, block size=2

- memory size = 32 words (16 blocks), cache size = 4 blocks,
- Access sequence: 22, 25, 23, 26, 16, 18, 16, 18

Decimal address	Binary address	Hit/miss	Cache index
22			
25			
23			
26			
16			
18			
16			
18			

index	V	tag	Data0	Data1
00	N			
01	N			
10	N			
11	N			

Direct Mapped Cache-- Address Translation

- Memory address with 2 bits byte offset: $x_{31}x_{30}x_{29}\ldots.x_i\ldots x_2x_1x_0$
- Cache size: 2^r blocks
- Block size : 2^b words
- Address translation: how to figure out cache index from the memory address?

- How many bits needed to implement the above cache?

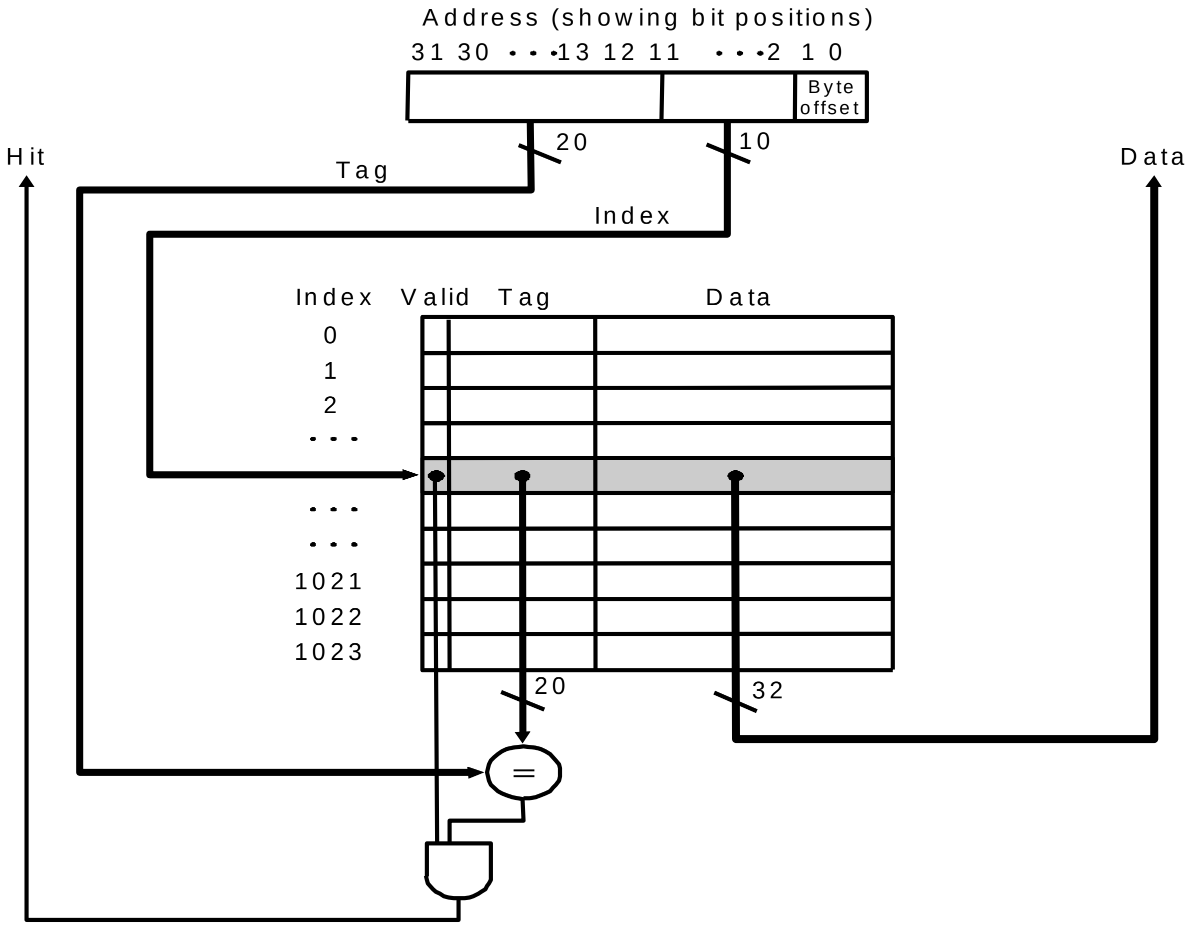
Direct Mapped Cache – MIPS (block size=1)

b=0, r=10

Low portion: select cache entry

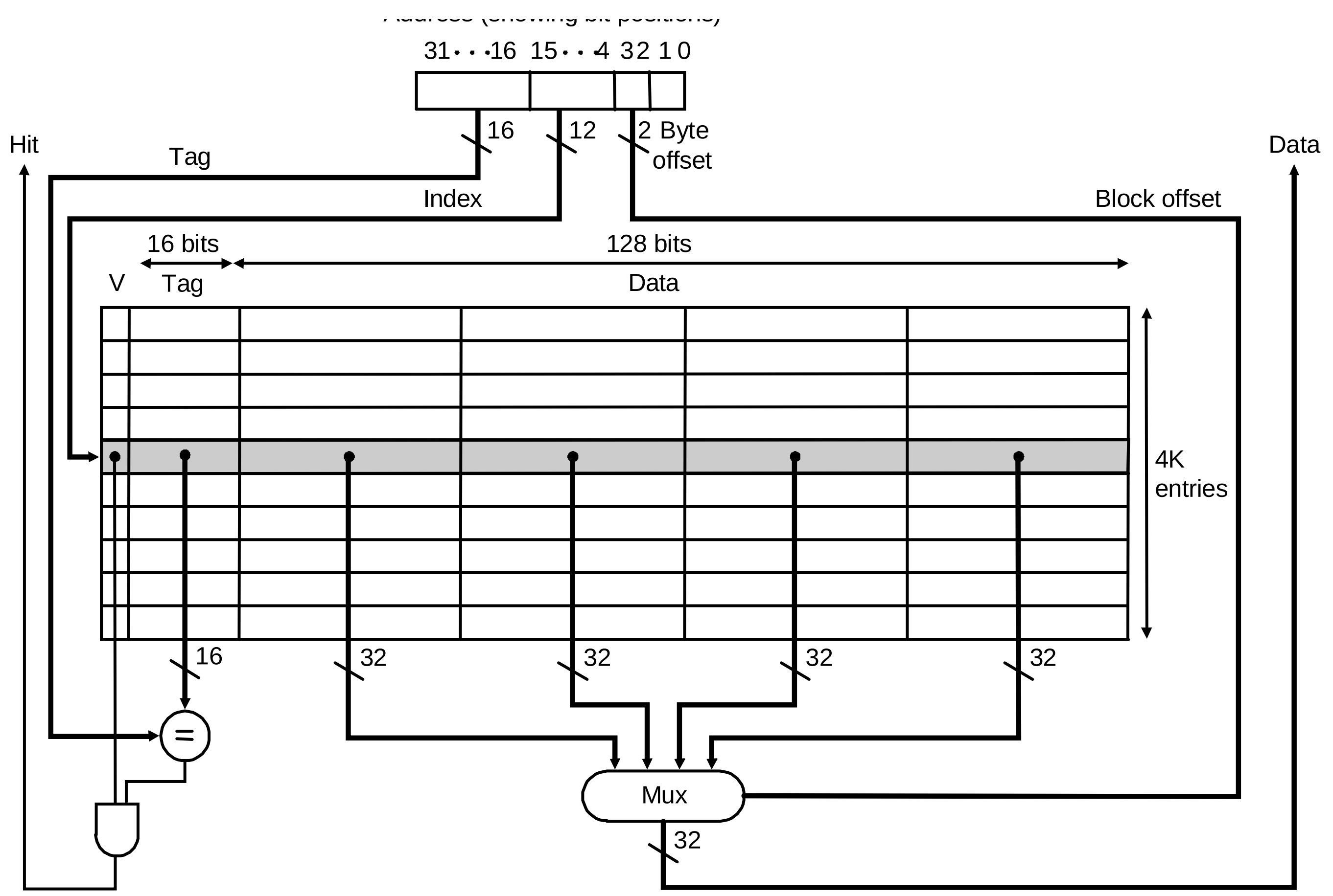
higher 20 pits: compared with tag

total # bits in the cache?



What kind of locality are we taking advantage of?

A 64 kb Direct Mapped Cache with 4 Word Block



b=2, r=12, tag: 16 bits (=32-2-2-12)

cache index:12 bits

Block offset: selects one word in a block

Taking advantage of spatial locality

Cache -- Direct Mapped-- Summary

- Each block in main mem. $\rightarrow$ only one fixed location in cache
  - many $\rightarrow$ one mapping
  - replacing: replace the block occupying the location
  - cache index = lower-order bits of memory address, excluding byte and block offset
- Advantages
- Disadvantages

Hits vs. Misses

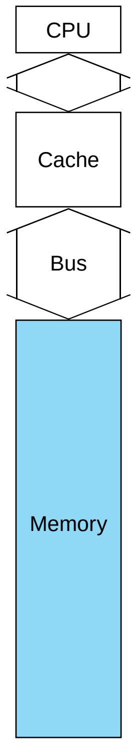
- Read hits
  - this is what we want!
- Read misses
  - stall the CPU, fetch block from memory, deliver to cache, restart
- Write hits: 2 write policies
  - write-through: update data in cache and memory right away
    - write buffer:a queue holding data to be written to memory
  - write-back: update data in cache only (update memory later)
- Write misses:
  - read the entire block into the cache, then write the word

How to Handle Cache Miss

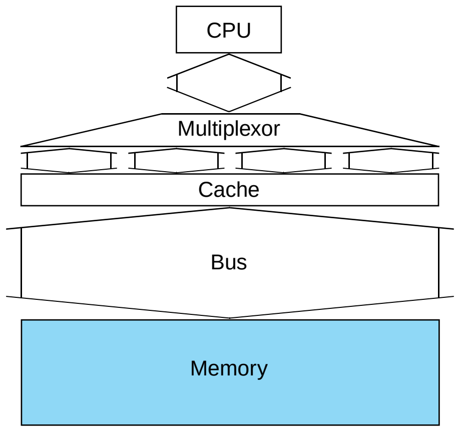
- Basic idea: stall CPU (not interrupt) & do the following
  - send the address of the miss item to memory
  - instruct memory to perform a read & wait for memory to complete the access
  - Write cache entry (data portion, tag, valid flag)
  - restart
- Miss penalty: setup time + block size

Designing Memory to Support Cache

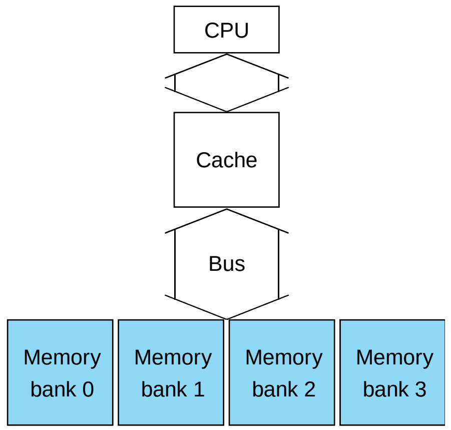
- Design a memory with higher bandwidth by increasing
  - physical width: wide memory
  - logic width: interleaved memory



a. One-word-wide memory organization



b. Wide memory organization



c. Interleaved memory organization

Performance

- Simplified model:

  execution time = (execution cycles + stall cycles) × cycle time

  stall cycles = #misses × miss penalty

  #misses= #I cache misses +#D cache misses

  miss penalty=

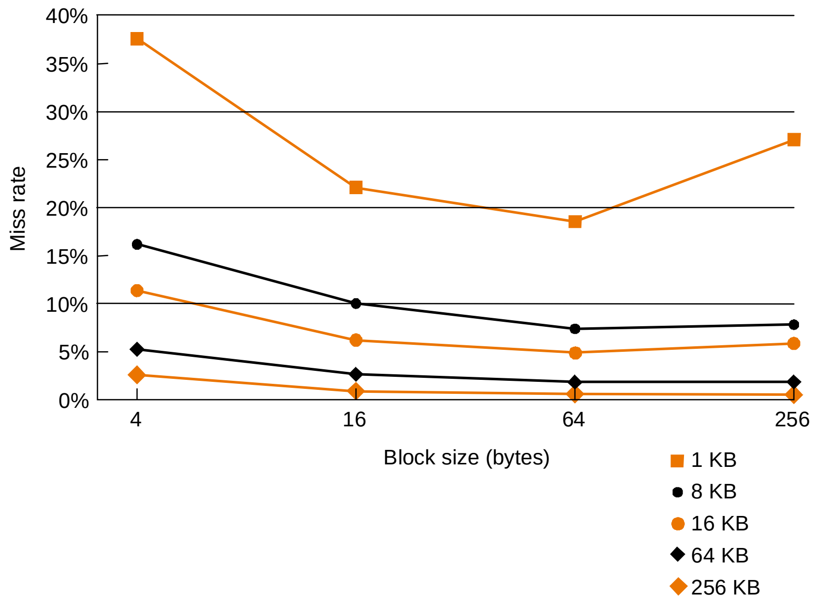
- Two ways of improving performance:
  - decreasing the miss ratio
  - decreasing the miss penalty

- Separate caches for instruction and data

Cache Misses: 3 C's

- Compulsory miss (cold start misses): A cache miss caused by the 1st access to a block that has never been in the cache.
- Capacity miss: A cache miss that occurs because of the cache size.
- Conflict miss (collision miss): A cache miss that occurs when multiple blocks compete for the same set while other sets are available.

- Increasing the block size tends to decrease miss rate:



Program	Block size in words	Instruction miss rate	Data miss rate	Effective combined miss rate
gcc	1	6.1%	2.1%	5.4%
	4	2.0%	1.7%	1.9%
spice	1	1.2%	1.3%	1.2%
	4	0.3%	0.6%	0.4%

Cache Performance -- Example

	loads	store	R-type	branch	jumps
gcc	24%	12%	44%	18%	2%

- Assume gcc
  inst cache miss: 2%, data cache miss: 4%
  CPI without memory stalls: 2
  miss penalty for all misses: 100 cycles
- Q1: how much faster a machine with a perfect cache (no misses)?
- Q2: Average penalty cycles per instruction
  - Inst miss cycles =
  - Data miss cycles =

$$\frac{CPU\ time\ with\ stalls}{CPU\ time\ with\ perfect\ cache}$$

Example

Find a reference string for which C2 has lower miss rate but spends more cycles on cache misses

Cache	C1	C2
mapping	direct	direct
block size	1	4
cache size	16words	16 words
#blocks	16	4
miss penalty	8 cycles	11 cycles
Initially	empty	empty

Average Memory 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 =
    - 2 cycles per instruction

- 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%

Set Associative Cache--Motivation

- Block size: 1 word
- Memory size: 16 blocks
- Cache size: 8 blocks
- Access sequence: 0,8,0,8,0,8
- Miss rate: 1
- Reorganize cache such that each cache index has two blocks-- 2 way set associative
  - miss rate reduced
- 2 way set associative cache
  - a cache that has 2 locations where each block can be placed

Associative Caches

- 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)

Comparison of 2 Caches with 8 words

- Direct mapped cache with block size=2
  - cache index: 2 bits
  - 4 cache sets, each set has **ONE** block of 2 words sharing same tag
  - address translation:

- 2-way set associative
  - cache index: 2 bits
  - 4 cache sets, each set has **TWO** blocks with different tags
  - use of replacement policy
  - address translation

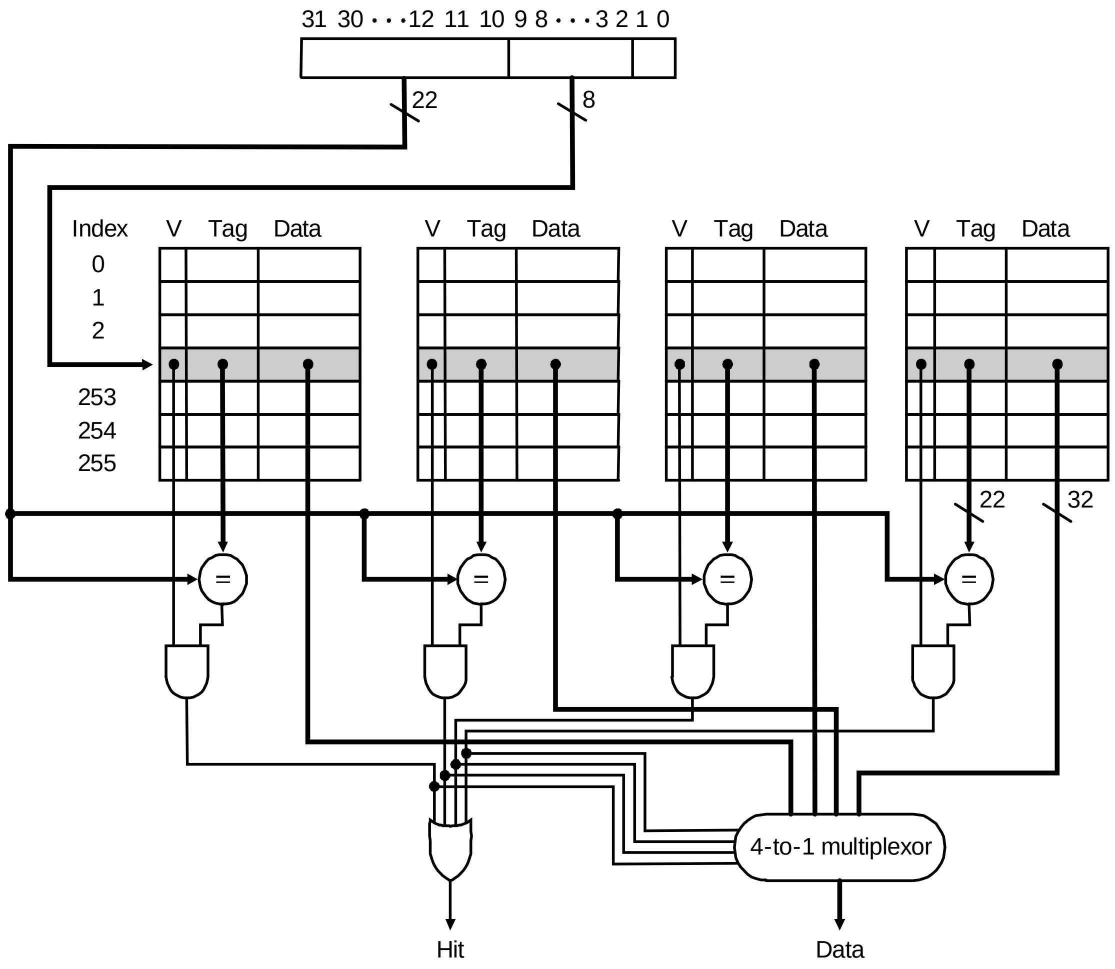
Example: two-way set associative vs direct

- memory size = 32, cache size = 8 words, direct cache has block size=2, set associative has block size =1
- Access sequence:

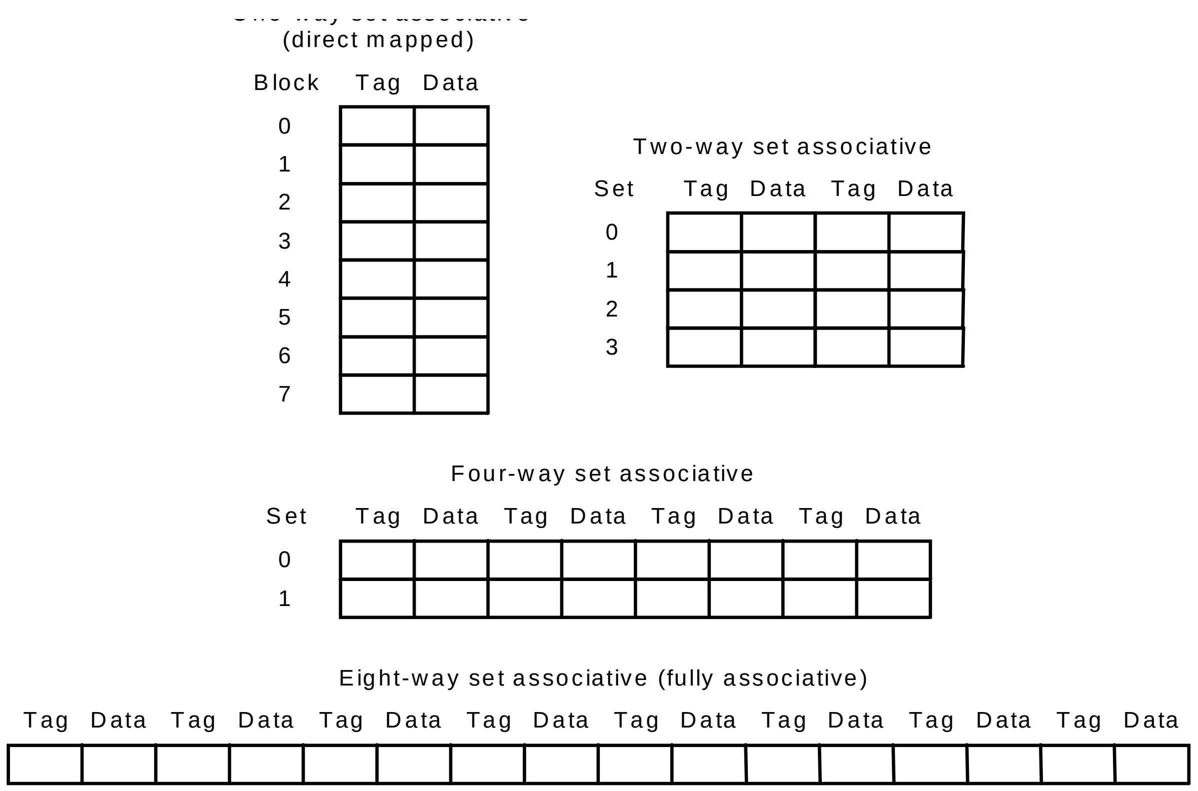
Implementation of 4-way associative cache

4 comparators

4-to-1 mux



Decreasing miss ratio with associativity



Fully associative: A cache in which a block can be placed in any location in cache

Miss rate decreases, access time increases

How Much Associativity

- Increased associativity decreases miss rate
  - But with diminishing returns
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%

Decreasing miss penalty with multilevel caches

- Add a second level cache:
  - often primary cache is on the same chip as the processor
  - use SRAMs to add another cache above primary memory (DRAM)
  - miss penalty goes down if data is in 2nd level cache
- Example: hit time (L1)= 1 cycle (cycle time =0.2 ns) with a 2% miss rate at L1 cache, 100 ns DRAM access. How much faster if we add L2 cache with 5 ns access time which decreases miss rate to main memory to 0.5%?

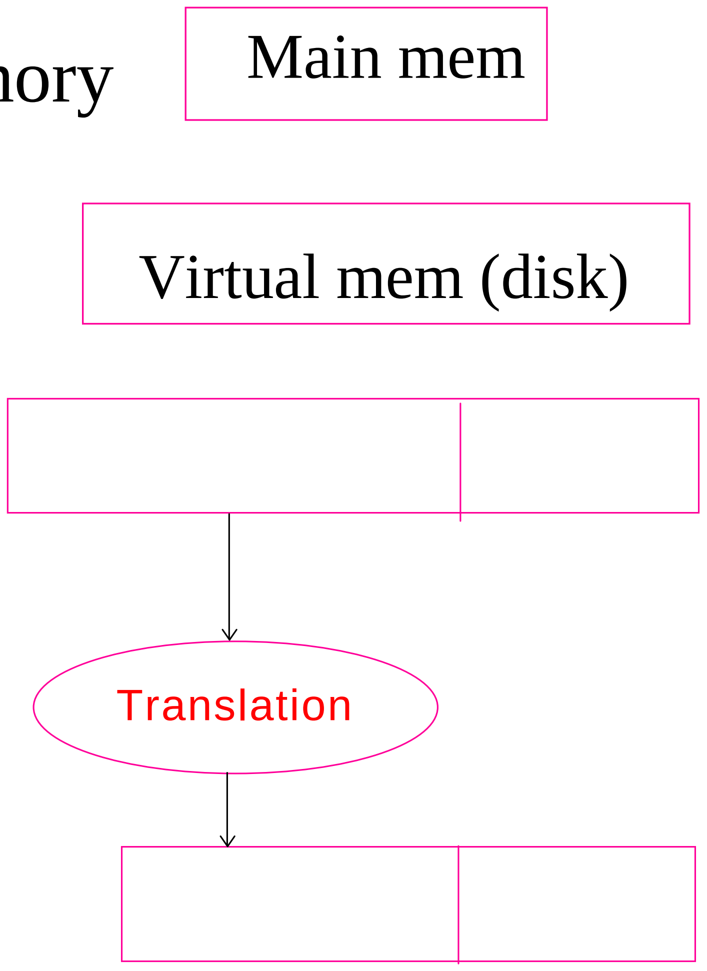
- Using multilevel caches:
  - try and optimize the hit time on the 1st level cache
  - try and optimize the miss rate on the 2nd level cache

Cache-- Summary

- Three address mapping:
  - Direct
  - Set associative
  - Fully associative
- Cache performance

Virtual Memory

- Upper level: main memory
  lower level: virtual mem. (disk)
- Motivations: increase size of memory system.
- Page: a virtual mem. block
- Page fault: a memory miss
- Virtual address
  physical address
  address translation

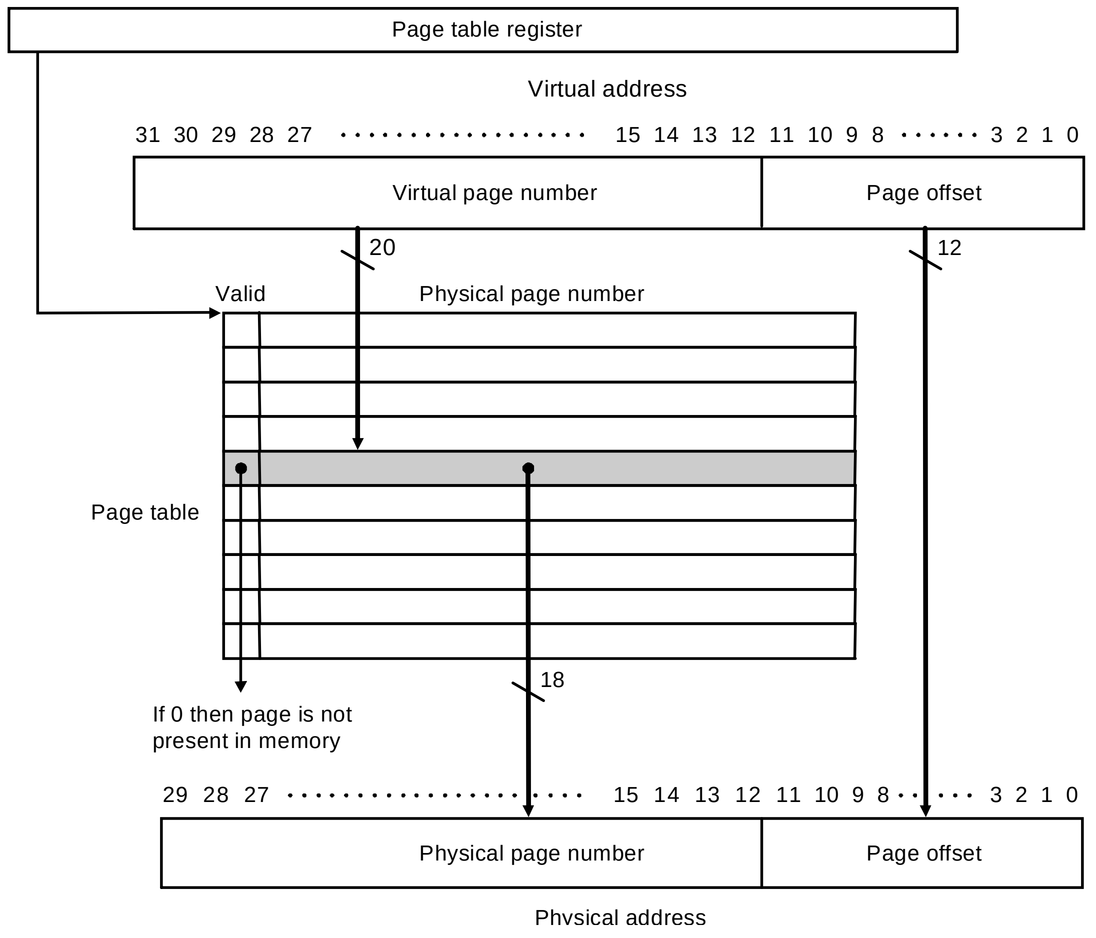


page size = 2^{12} = 4k

Addressing Mapping & Address Translation

- Fully associative: a page can go any page frame in physical memory
- Page table: tells if a page is in physical memory; if it is in, provides the physical address
  - each program has a page table
  - page table stored in main memory
- Page table register: pointing to the page table

Page Tables--Translating virtual to physical address



Page table (con't)

- Page size = 2^{12} bytes = 4 KB
- Virtual address space = 2^{32}
- Physical address space = 2^{30}
- # entries of page table = 2^{20}
- Valid =
  - 1: page table item indicates the physical page number
  - 0: page table item indicates the location on disk
    - have a page fault
    - bring the page from disk to main memory
    - insert the physical page # into the page table.

Replacement Policy

- Page faults:
  - how to find the missing page on disk?
  - choosing a page to replace? LRU scheme
- LRU (least recently used): replace the page least recently used
  - reference bit in page table: each time a page is accessed, set ref bit =1; OS periodically clears ref. bit
- Example Reference order

$$\xrightarrow{\qquad\qquad\qquad\qquad\qquad\qquad}$$

10, 12, 12, 9, 7, 11, 10

current access 8 is missing, which page to replace?

Writes

- Write through does not work
  - big page -- 4 kbytes
  - takes 100 ~ 1,000 cycles to write disk (startup penalty)
- Write back: individual writes are accumulated into a page. The page is copied back to disk only when the page is replaced
- Dirty bit : = 1 if the page has been written

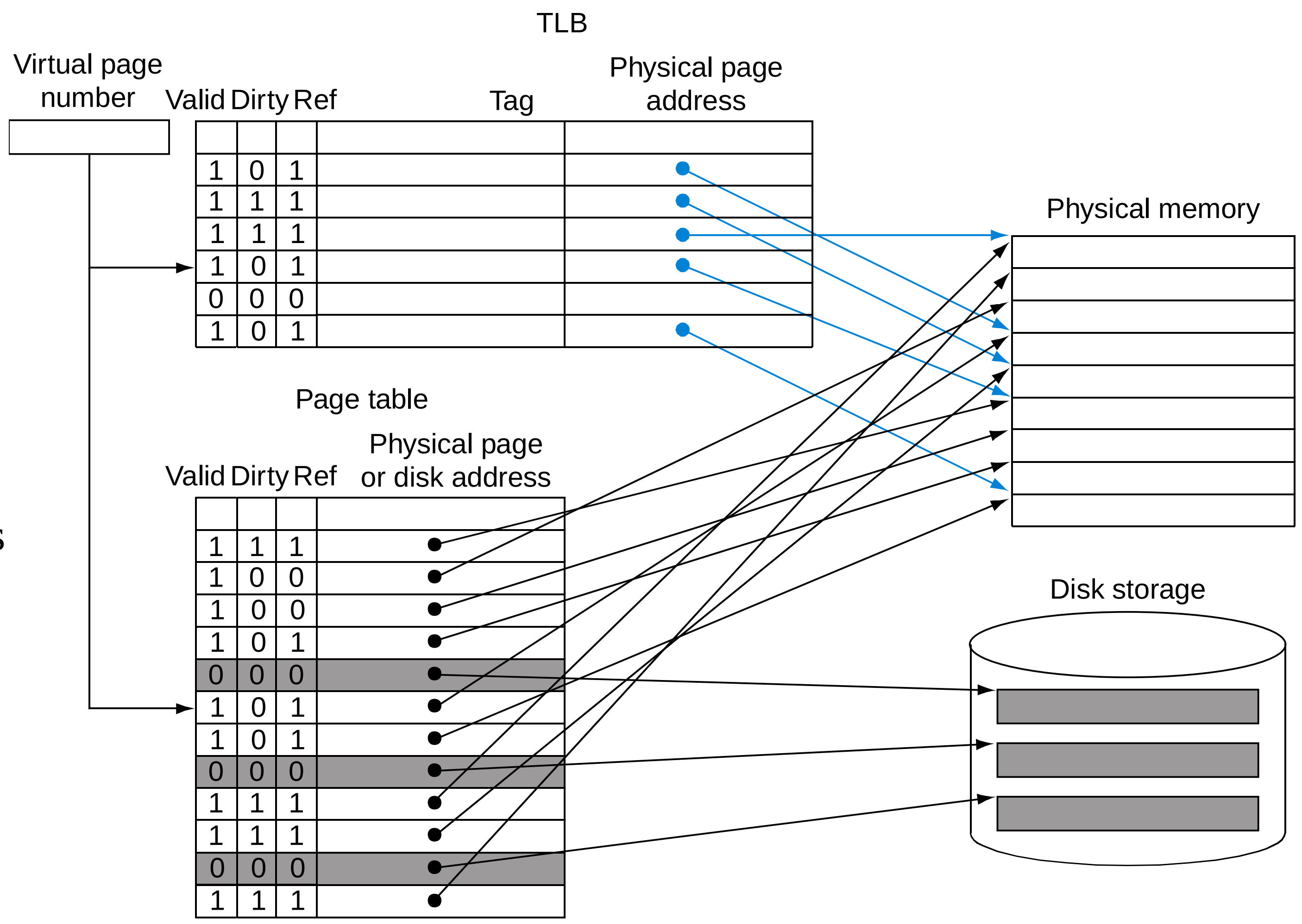
  0 if the page never changed
- Each entry in the page table contains:

How to make address translation faster?

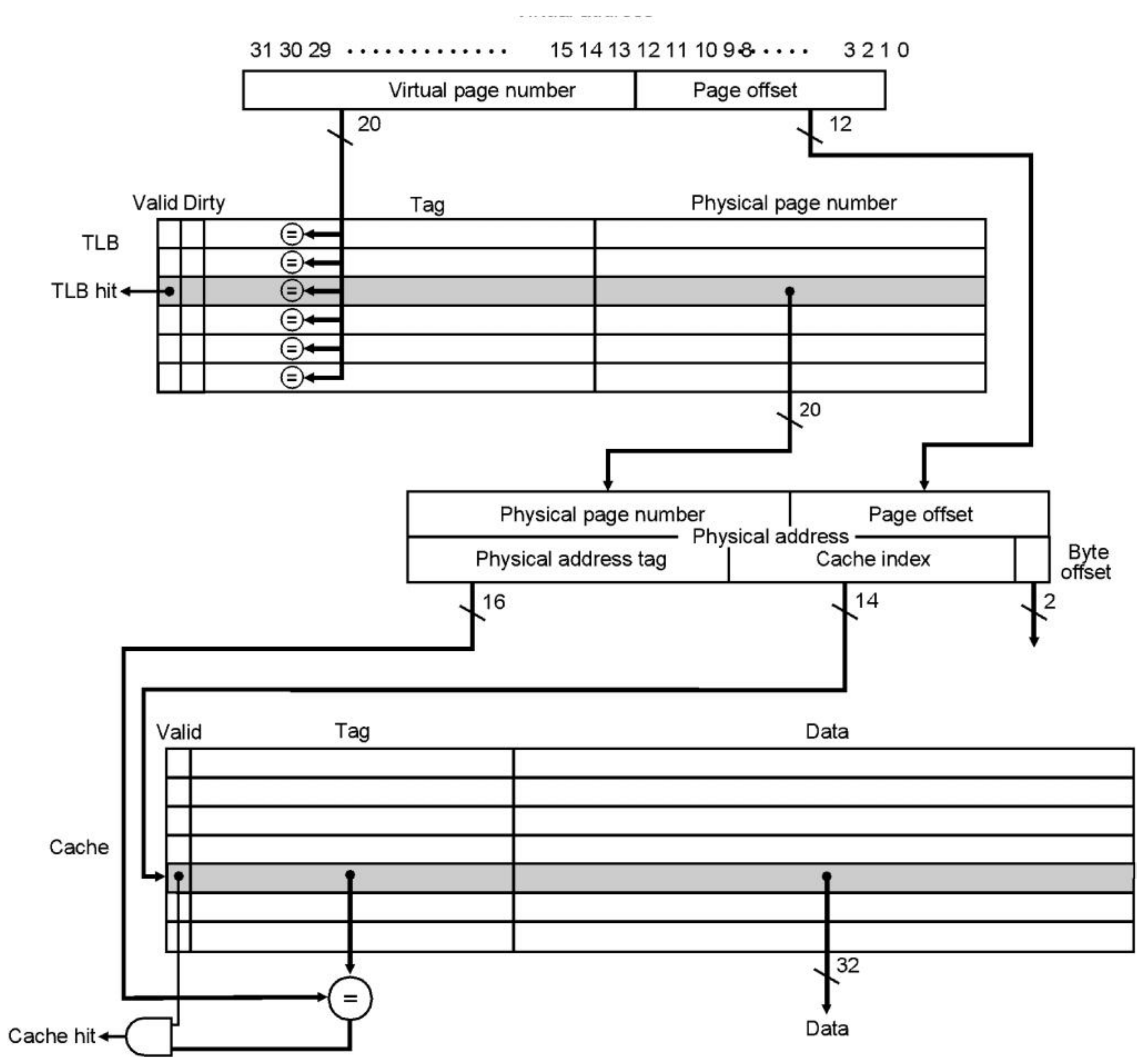
- One step in address translation
  - Use virtual page number to access page table—access main memory to translate address?
- Another cache to hold part translation items in the page table -- TLB

Making Address Translation Fast

- A cache for address translations: translation lookaside buffer TLB
- TLB: keeps track of recent address translations
- An associative memory: access by content



Typical values:	16-512 entries,
	miss-rate: .01% - 1%
	miss-penalty: 10 – 100 cycles


31 30 29 . . . . 15 14 13 12 11 10 9 8 . . . . 3 2 1 0
Virtual page number
Page offset
20
12
Valid
Dirty
Tag
Physical page number
TLB
TLB hit
20
Physical page number
Page offset
Physical address
Physical address tag
Cache index
Byte offset
16
14
2
Valid
Tag
Data
Cache
32
Cache hit
Data

4 questions for TLB vs. page table

- Mapping of page table items to TLB
- Finding an item in TLB
- Replace policy:
- Writes:

Virtual Memory Summary

- Concepts and motivations for virtual memory
- Block = page: large (4 ~ 16 KB)
- Mapping of virtual address to physical address
  - fully associative
  - page table
- Replacement policy: LRU + reference bit in page table
- Write-back
- TLB

Big Picture
A common Framework for Memory Hierarchy

- Q1: address mapping of lower level to upper level
  - direct mapping
  - set associative mapping
  - fully associative mapping
- Q2: how is a data found?
  - index + tag comparison --- cache
  - page table + page table register -- virtual memory
  - fully search -- TLB
- Q3: which block should be replaced?
  - in direct mapping: fixed
  - random
  - LRU + ref. Bit

* Q4: writes
- write through + write buffer
- write back + dirty bit

Imagine a system with the following parameters:

Virtual Address	20 bits	
Physical Address	18 bits	
Page size	1KB	
TLB	2 way set associative, 16 total entries	

TLB						
	Set 0			Set 1		
Index	Tag	PPN	Valid	Tag	PPN	Valid
0	03	C3	1	01	71	0
1	00	28	1	01	35	1
2	02	68	1	3A	F1	0
3	03	12	1	02	30	1
4	7F	05	0	01	A1	0
5	00	53	1	03	4E	1
6	1B	34	0	00	1F	1
7	03	38	1	32	09	0

Page Table		
VPN	PPN	Valid
000	71	1
001	28	1
002	93	1
003	AB	0
004	D6	0
005	53	1
006	1F	1
007	80	1
008	02	0
009	35	1
00A	41	0
00B	86	1
00C	A1	1
00D	D5	1
00E	8E	0
00F	D4	0
010	60	0
011	57	0
012	68	1
013	30	1
014	0D	0
015	2B	0
016	9F	0
017	62	0
018	C3	1
019	04	0
01A	F1	1
01B	12	1
01C	30	0
01D	4E	1
01E	57	1
01F	38	1

The content of the TLB and the first 32 entries of the page table are shown (all numbers are in hex)

1.Show the number of bits used for the virtual page number and the virtual page offset

2.Show the numbers of bits used for the TLB tag and TLB index

3.How many entries are in the page table?

4.Show the number of bits used for the physical page number and the physical page offset

5.For the virtual address 0x078E6 what is the physical page number