Chapter 6: Memory

Ngo Lam Trung, Pham Ngoc Hung, Hoang Van Hiep

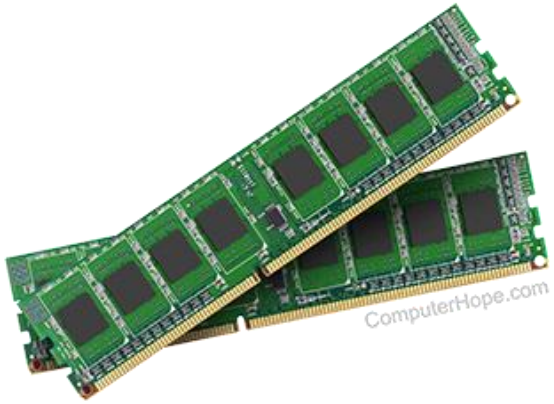
[with materials from *Computer Organization and Design*, MK and M.J. Irwin's presentation, PSU 2008]

Content

- Memory hierarchy
- Principal of locality
- Cache
- Virtual memory

Memory

- Memory: where data are stored.

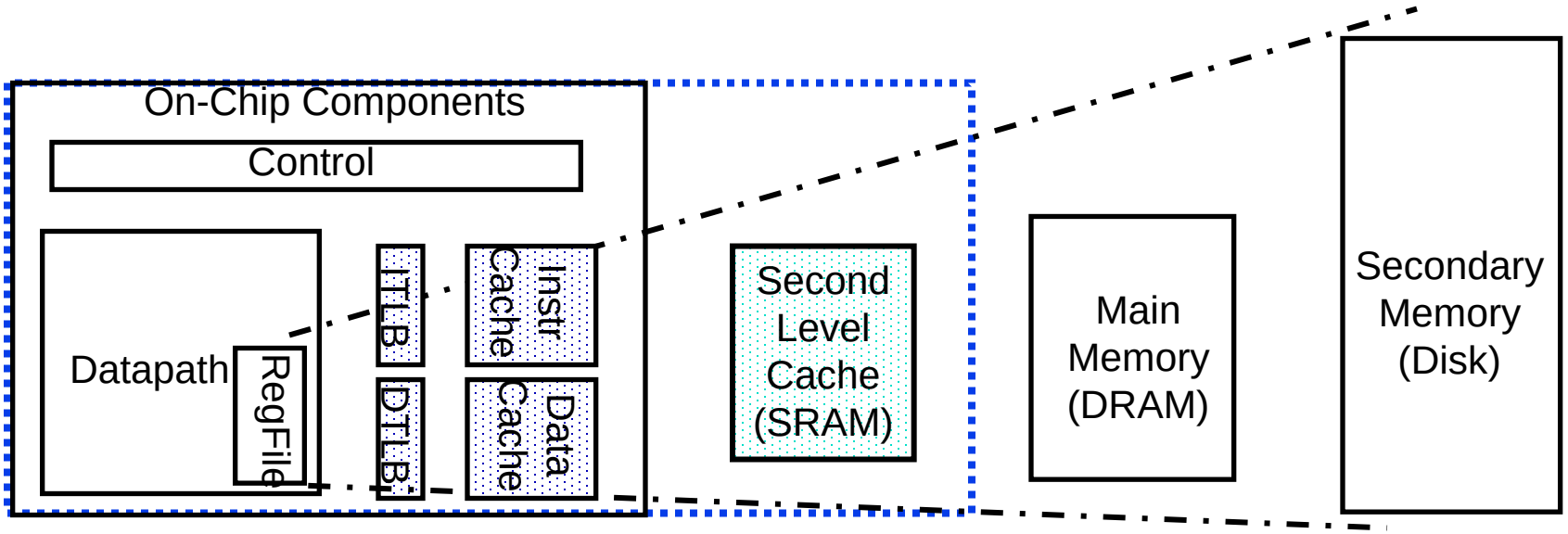


Why is memory critical to performance?

Memory technology

- Static RAM (SRAM)
  - 0.5ns – 2.5ns, $500 – $1000 per GB
- Dynamic RAM (DRAM)
  - 50ns – 70ns, $10 – $20 per GB
- Flash memory
  - 5,000ns – 50,000ns, $0.75 – $1 per GB
- Magnetic memory
  - 5,000,000ns – 20,000,000ns, $0.05 – $0.1 per GB
- Fact:
  - Large memories are slow
  - Fast memories are small (and expensive)

A Typical Memory Hierarchy



Speed (%cycles):	½'s	1's	10's	100's	10,000's
Size (bytes):	100's	10K's	M's	G's	T's
Cost:	highest				lowest

- How to get an ideal memory
  - As fast as SRAM
  - As cheap as disk?

The Memory Hierarchy: Locality Principal

- C program

```
int x[1000], temp;
for (i = 0; i < 999; i++)
    for (j = i+1; j < 1000; j++)
        if (x[i] < x[j])
        {
            temp = x[i];
            x[i] = x[j];
            x[j] = temp;
        }
```

Data memory at location of temp and x are accessed multiple times

Instruction memory at location of the two **for** loops are used repeatedly

The Memory Hierarchy: Locality Principal

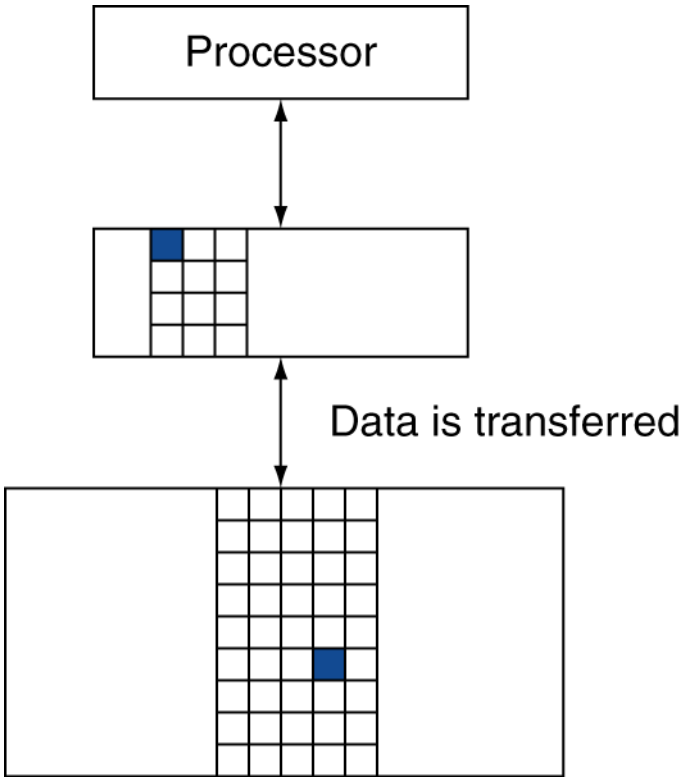
- Temporal Locality (locality in time)
  - If a memory location is referenced then it will tend to be referenced again soon

  ⇒ Keep most recently accessed data items closer to the processor

- Spatial Locality (locality in space)
  - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon

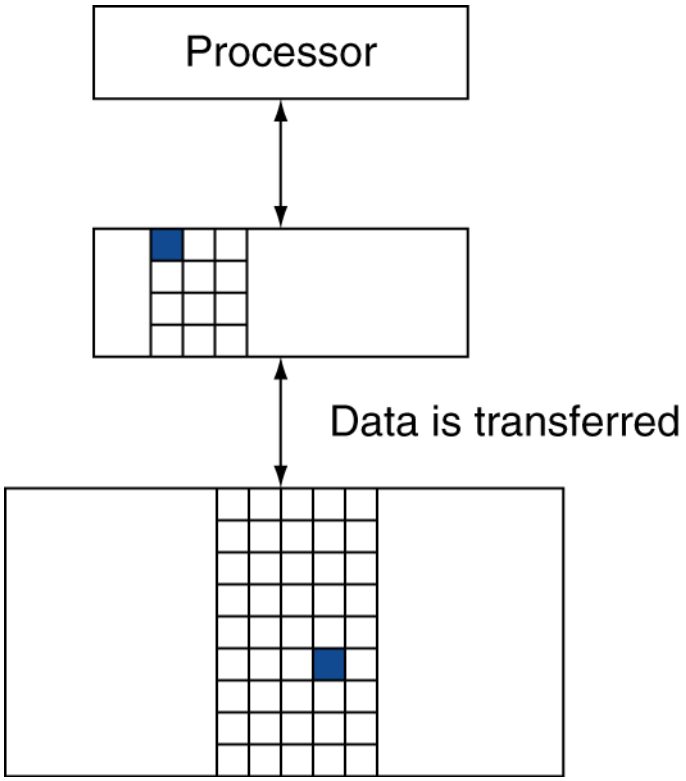
  ⇒ Move blocks consisting of contiguous words closer to the processor

Hierarchical memory access



- Data are stored in multiple levels.
  - High level: fast but small
  - Low level: slow but large
- Data are transferred in units of block (of multiple words) between levels, through the hierarchy.
- Frequently used data are stored closer to processor.

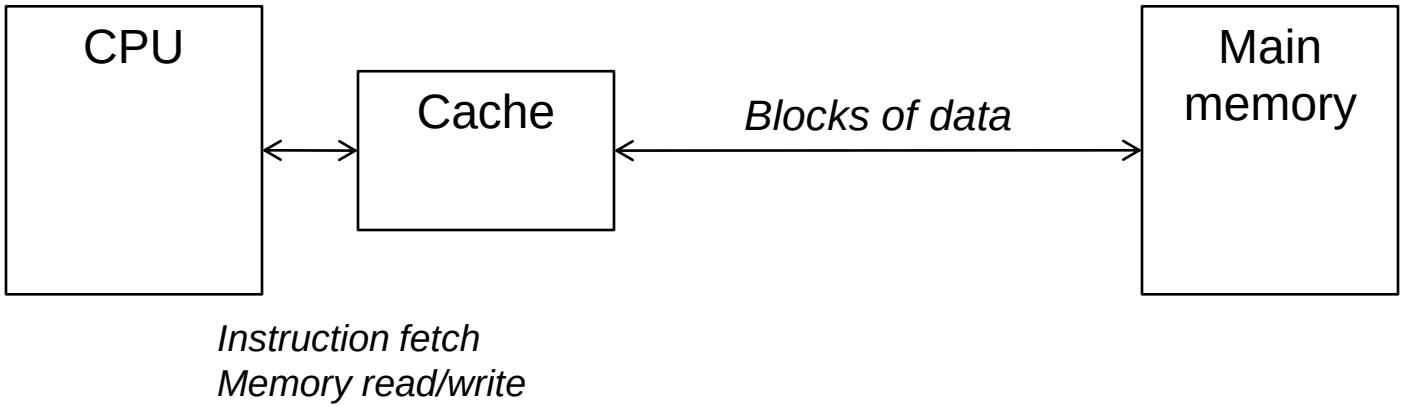
Hierarchical memory access



- Associative data access:
  - Processor access data in lower level
  - Data transfer from lower level to processor via upper level(s)
- 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

Cache

- The memory hierarchy between the processor and main memory
  - CPU fetch instructions and data from cache, if found (cache hit) → fast access (hit time).
  - If not found (cache miss) → load a block from main memory into cache, then access in cache → slower access time (miss penalty)
  - Hit time << miss penalty



Cache Basics

- CPU needs to access a data item in memory

➔Two questions to answer (in hardware):

  - Q1: How does CPU know if the data item is in the cache?
  - Q2: If it is, how does CPU find it?

- To answer the first question
  - Adding set of tags fields into cache: each block in cache has a tag
  - The tags contain address information to identify whether a word in cache is corresponding to the requested one in memory.
- To answer the second question
  - Depends on how a block in memory is mapped into block (line) in cache
    - methods for mapping: Direct mapping, Fully associative mapping, N-way set associative mapping

Cache Basics

- CPU needs to access a data item in memory

➔Two questions to answer (in hardware):

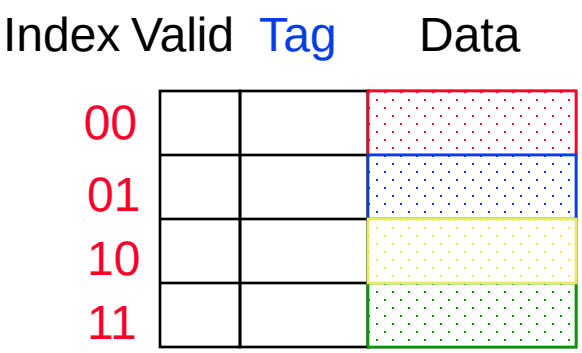
  - Q1: How does CPU know if the data item is in the cache?
  - Q2: If it is, how does CPU find it?

- Direct mapped
  - Each memory block is mapped to exactly one block in the cache
    - lots of lower level blocks must share blocks in the cache
  - Address mapping (to answer Q2):

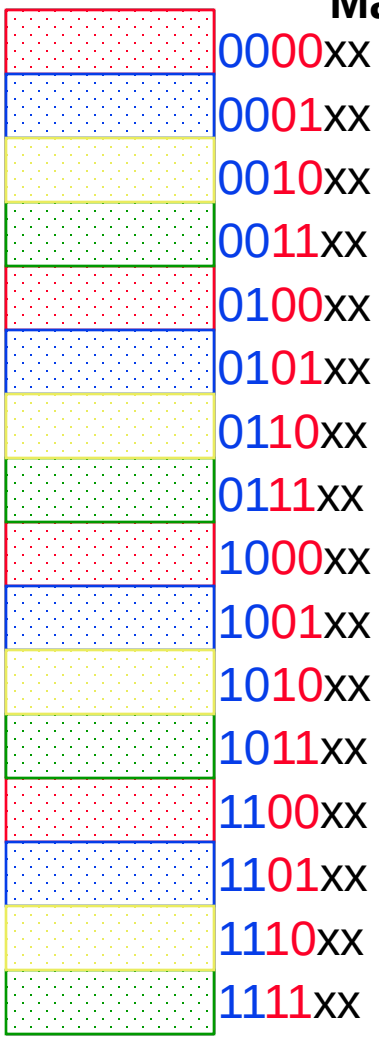
    (block address) modulo (# of blocks in the cache)
  - The tag field: associated with each cache block that contains the address information (the upper portion of the address) required to identify the block (to answer Q1)
  - The valid bit: if there is data in the block or not

Caching: A Simple First Example

Cache



Main Memory



One-word blocks
Two low order bits define the byte in the word (32b words)

Q1: Is it there?

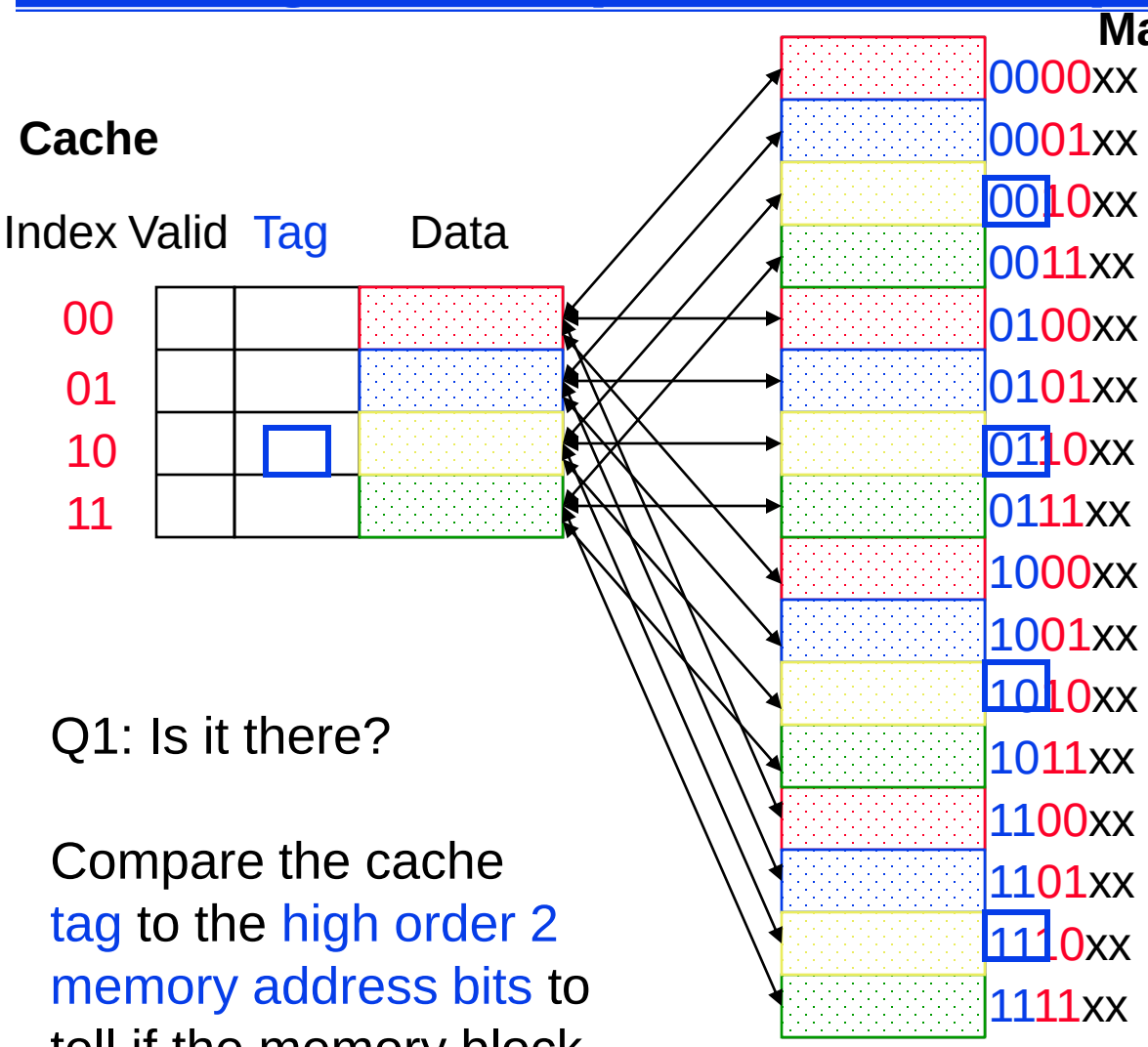
Compare the cache tag to the high order 2 memory address bits to tell if the memory block is in the cache

Q2: How does CPU find it?

Use next 2 low order memory address bits – the index – to determine which cache block (i.e., modulo the number of blocks in the cache)

(block address) modulo (# of blocks in the cache)

Caching: A Simple First Example



One word blocks
Two low order bits define the byte in the word (32b words)

Q1: Is it there?

Compare the cache tag to the high order 2 memory address bits to tell if the memory block is in the cache

Q2: How do we find it?

Use next 2 low order memory address bits – the index – to determine which cache block (i.e., modulo the number of blocks in the cache)

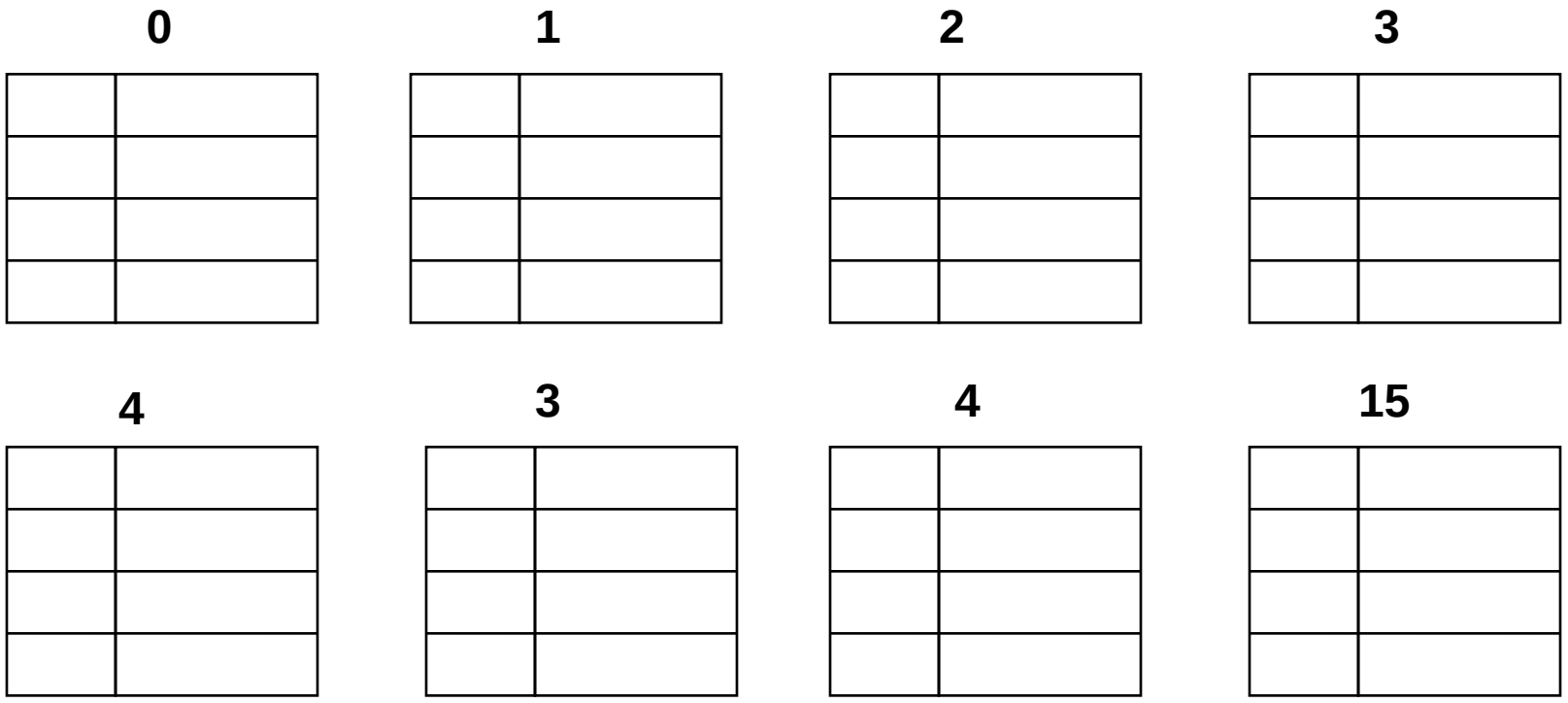
(block address) modulo (# of blocks in the cache)

Direct Mapped Cache

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

0 1 2 3 4 3 4 15



Direct Mapped Cache

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

0 1 2 3 4 3 4 15

0 miss

00	Mem(0)

1 miss

00	Mem(0)
00	Mem(1)

2 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)

3 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 miss

~~00~~ 01	~~Mem(0)~~ Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

3 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

15 miss

01	Mem(4)
00	Mem(1)
00	Mem(2)
~~00~~ 11	~~Mem(3)~~ Mem(15)

- 8 requests, 6 misses
- What if we repeatedly request 1,000,000 times

Cache performance

- Given a RISC-V CPU running a program with the miss rate of instruction cache is 2% and the miss rate of data cache is 4%. The processor has CPI of 2 without any memory stalls and the miss penalty is 100 cycles for all misses
- Determine how much faster that processor would run with a perfect cache that never missed. Assume the frequency of all loads and stores is 36%.
- Solution:

Cache performance

- Given a RISC-V CPU running a program with the miss rate of instruction cache is 2% and the miss rate of data cache is 4%. The processor has CPI of 2 without any memory stalls and the miss penalty is 100 cycles for all misses.
- Determine how much faster that processor would run with a perfect cache that never missed. Assume the frequency of all loads and stores is 36%.
- Solution:
- Given instruction count I

$$\text{Instruction miss cycles} = I * 2\% * 100 = 2.00 * I$$

$$\text{Data miss cycles} = I * 36\% * 4\% * 100 = 1.44 * I$$

- Total mem-stall cycles: $2.00\ I + 1.44\ I = 3.44\ I.$

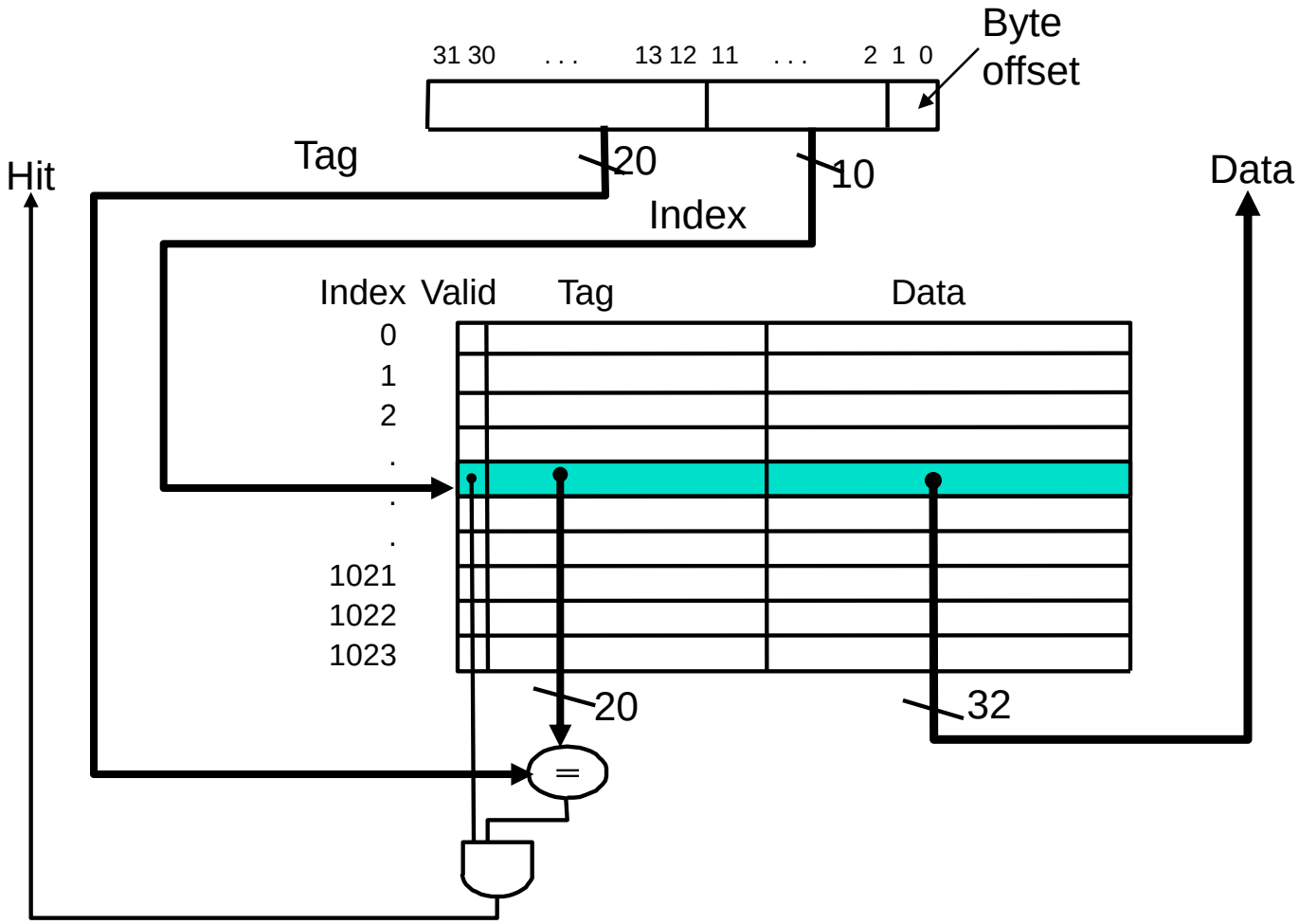
$$\frac{\text{CPU time with stalls}}{\text{CPU time with perfect cache}} = \frac{I \times \text{CPI}_{\text{stall}} \times \text{Clock cycle}}{I \times \text{CPI}_{\text{perfect}} \times \text{Clock cycle}} = \frac{\text{CPI}_{\text{stall}}}{\text{CPI}_{\text{perfect}}} = \frac{5.44}{2} = 2.72$$

Cache performance

- Given a RISC-V CPU running a program with the miss rate of instruction cache is 2% and the miss rate of data cache is 4%. The processor has CPI of 2 without any memory stalls and the miss penalty is 100 cycles for all misses
- Determine how much faster that processor would run with a perfect cache that never missed. Assume the frequency of all loads and stores is 36%.
- What is the speed up if the CPU now has faster CPI of 1 (instead of 2)?

Direct Mapped Cache Example

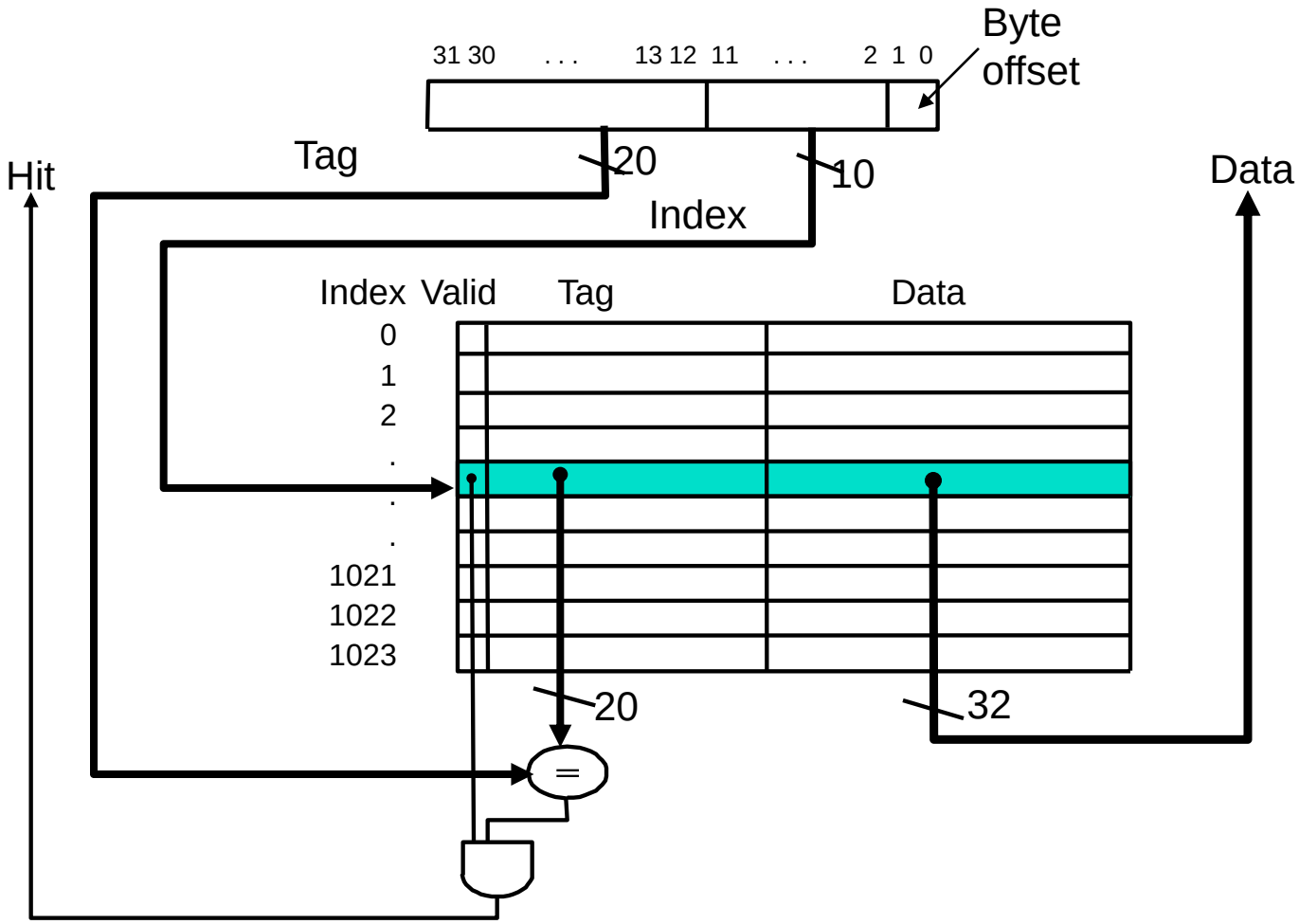
- One-word blocks, cache size = 1K words (or 4KB)



What kind of locality are we taking advantage of?

Direct Mapped Cache Example

- One-word blocks, cache size = 1K words (or 4KB)



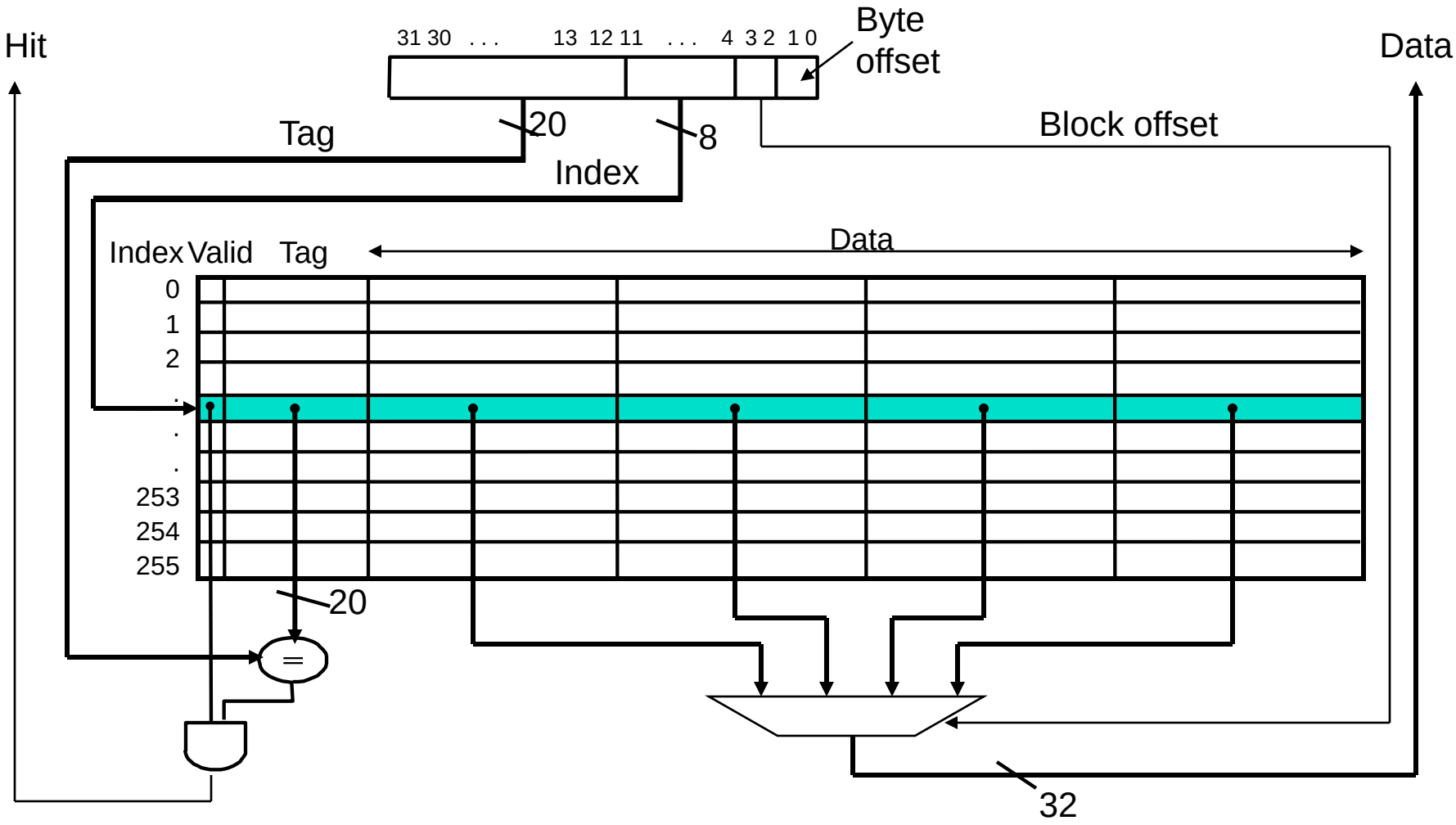
Calculate the total size of this cache in Kilobits

Exercise

- How many total bits are required for a direct-mapped cache with 16 KiB of data and 1-word blocks, assuming a 32-bit address?

Multiword Block Direct Mapped Cache

- Four words/block, cache size = 1K words



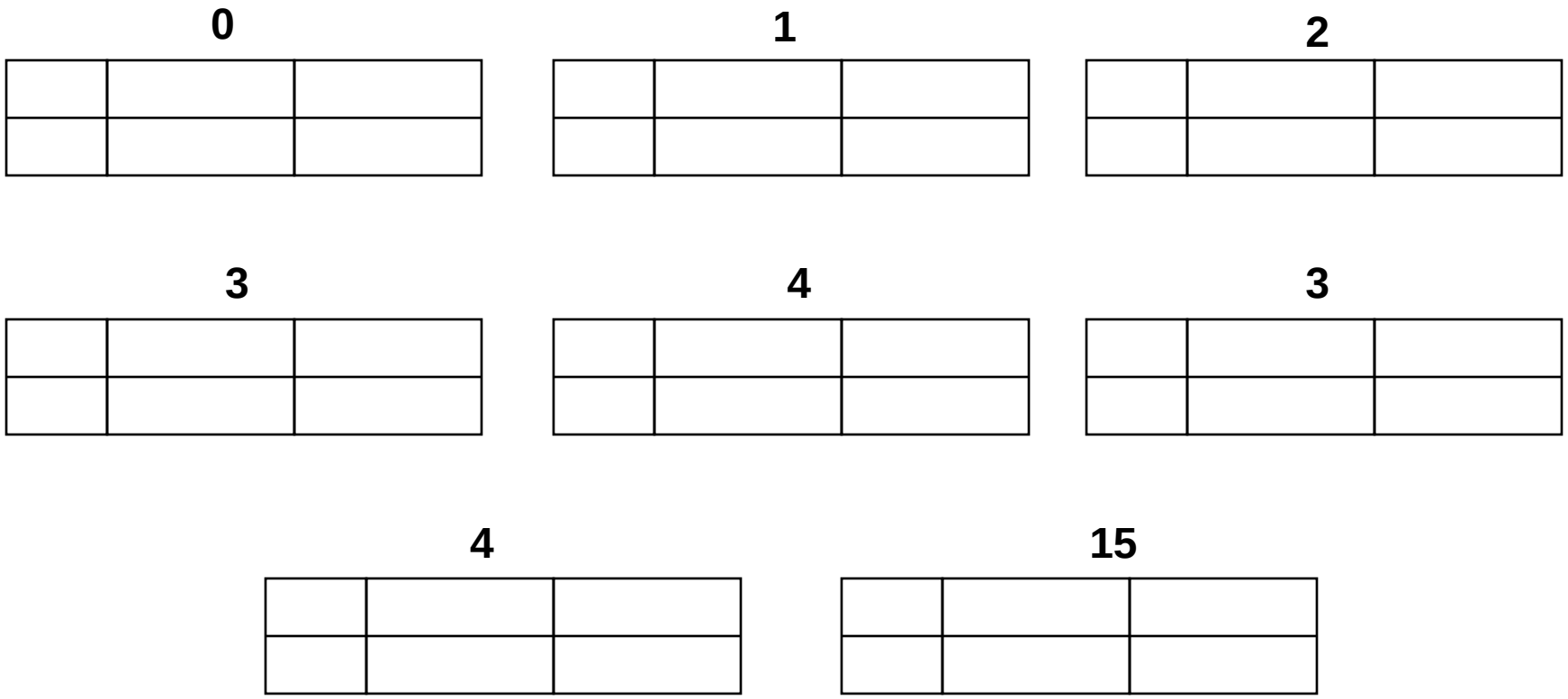
What kind of locality are we taking advantage of?

Taking Advantage of Spatial Locality

- Let cache block hold more than one word

Start with an empty cache - all blocks initially marked as not valid

0 1 2 3 4 3 4 15



Taking Advantage of Spatial Locality

- Let cache block hold more than one word

Start with an empty cache - all blocks initially marked as not valid

0 1 2 3 4 3 4 15

0 miss

00	Mem(1)	Mem(0)

1 hit

00	Mem(1)	Mem(0)

2 miss

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

3 hit

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

4 miss

~~00~~ 01	~~Mem(1)~~ 5	~~Mem(0)~~ 4
00	Mem(3)	Mem(2)

3 hit

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

4 hit

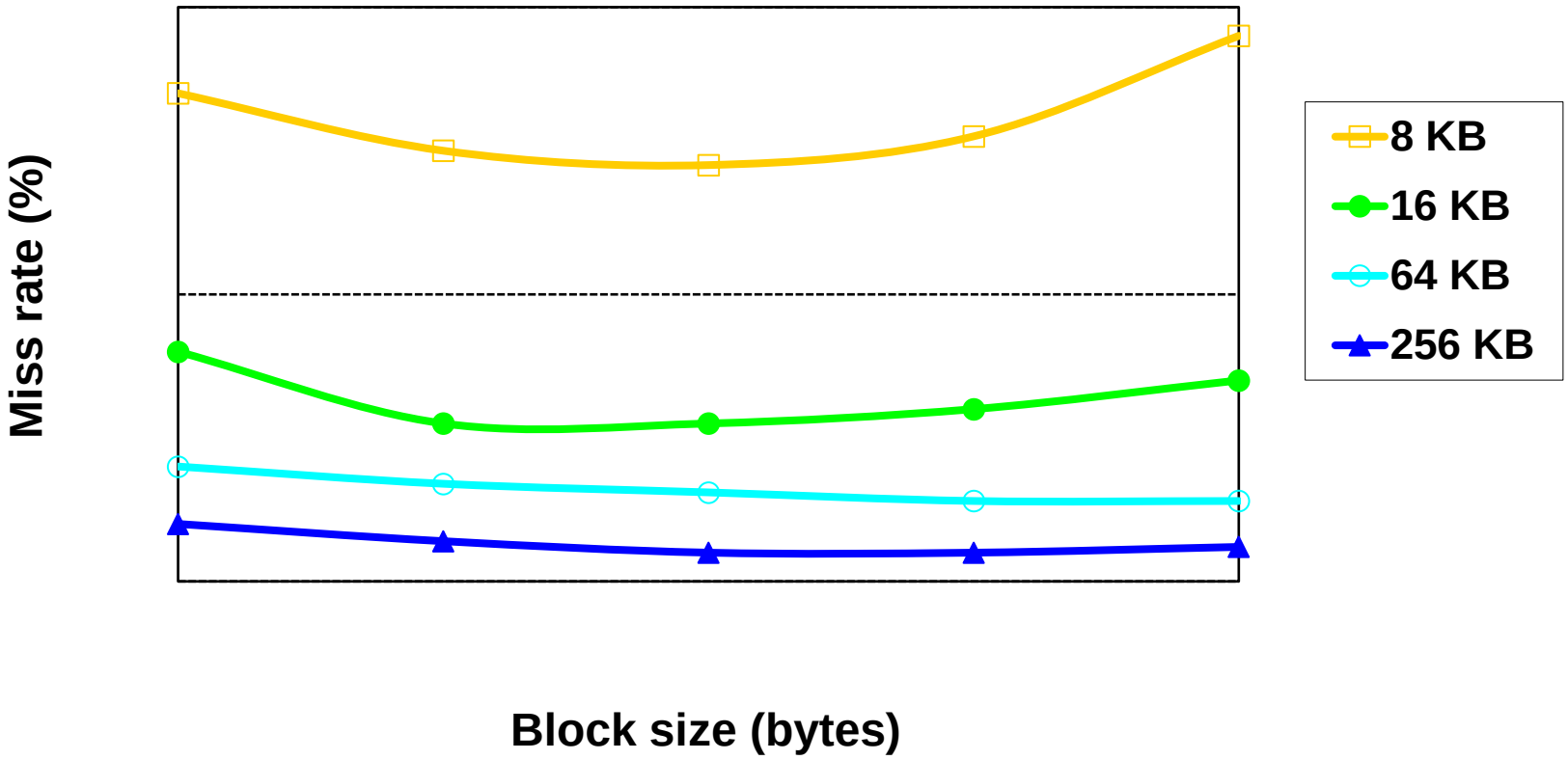
01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

15 miss

~~01~~ 11	Mem(5)	Mem(4)
~~00~~	~~Mem(3)~~ 15	~~Mem(2)~~ 14

- 8 requests, 4 misses

Miss Rate vs Block Size vs Cache Size



- Miss rate goes up if the block size becomes a significant fraction of the cache size because the number of blocks that can be held in the same size cache is smaller (increasing capacity misses)

Note when increase the block size

- Decrease the miss rate, but
- Increase the miss penalty (# clock cycles to load data from memory into cache)
- Leads to capacity miss: the cache cannot store all the required blocks due to limited space (increase the block size means decrease the number of block inside the cache)

Cache Field Sizes

- The number of bits in a cache includes both the storage for data and for the tags
  - 32-bit byte address
  - For a direct mapped cache with 2^n blocks, *n* bits are used for the index
  - For a block size of 2^m words (2^{m+2} bytes), *m* bits are used to address the word within the block and 2 bits are used to address the byte within the word
- What is the size of the tag field?
- The total number of bits in a direct-mapped cache is then

  2^n x (block size + tag field size + valid field size)
- How many total bits are required for a direct mapped cache with 16KB of data and 4-word blocks assuming a 32-bit address?

Exercise

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

Sources of Cache Misses

- Compulsory (cold start, first reference):
  - First access to a block.
  - We cannot do much on this.
  - Solution: increase block size (but also increases miss penalty).
- Capacity:
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size (may increase access time)
- Conflict (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity (may increase access time)

Reducing Cache Miss Rates #1

➔Allow more flexible block placement

- Direct mapped cache: a memory block maps to exactly one cache block
- Fully associative cache allow a memory block to be mapped to *any* cache block
- A compromise is to divide the cache into sets each of which consists of n "ways" (n-way set associative). A memory block maps to a unique set (specified by the index field) and can be placed in any way of that set (so there are n choices)

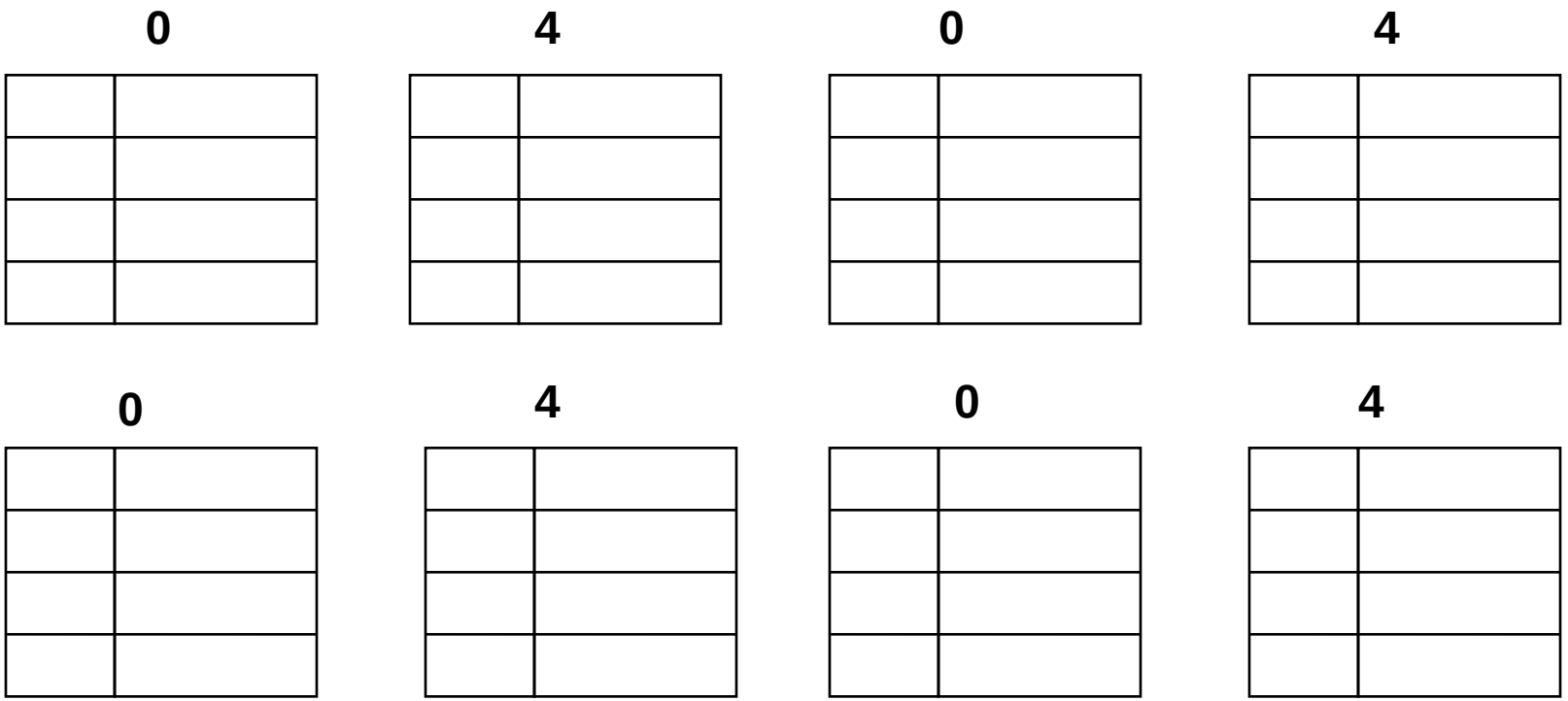
(block address) modulo (# sets in the cache)

Another Reference String Mapping

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

0 4 0 4 0 4 0 4

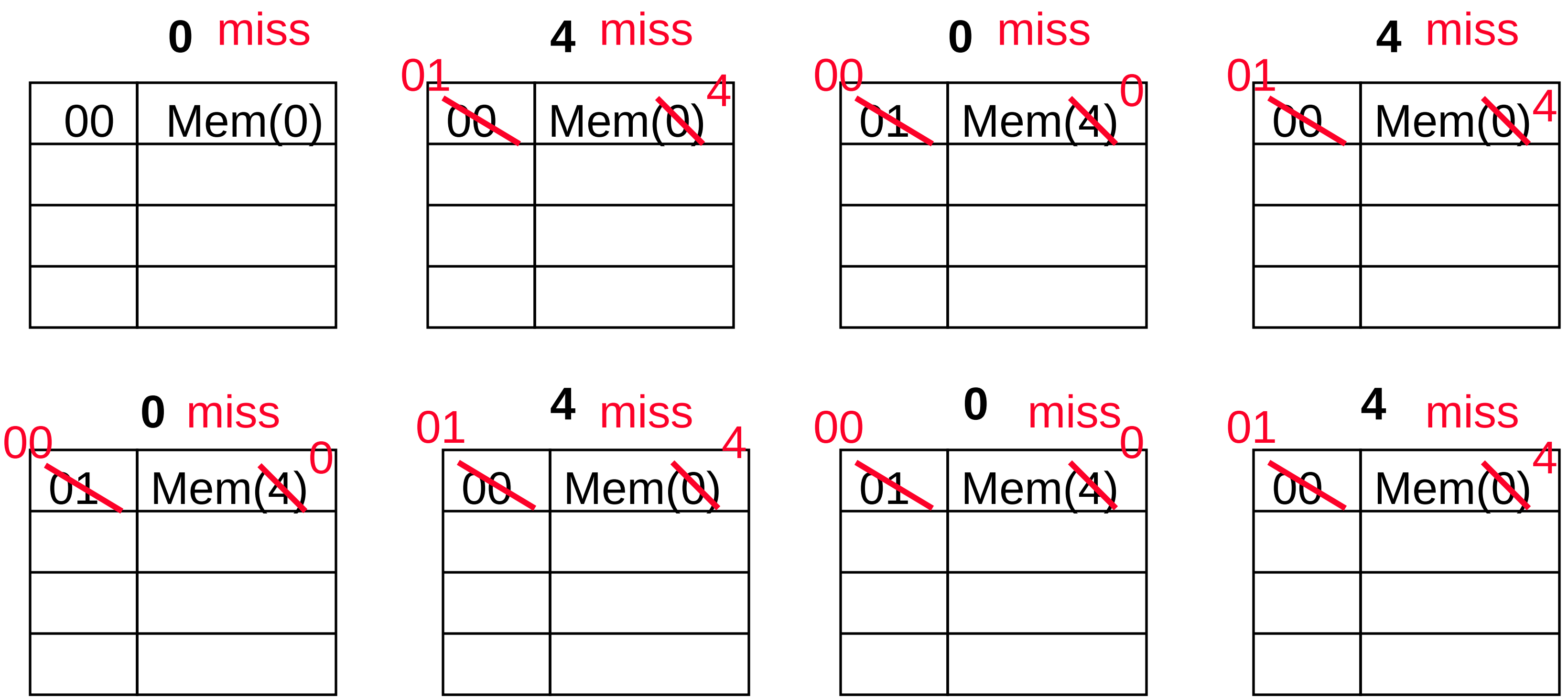


Another Reference String Mapping

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

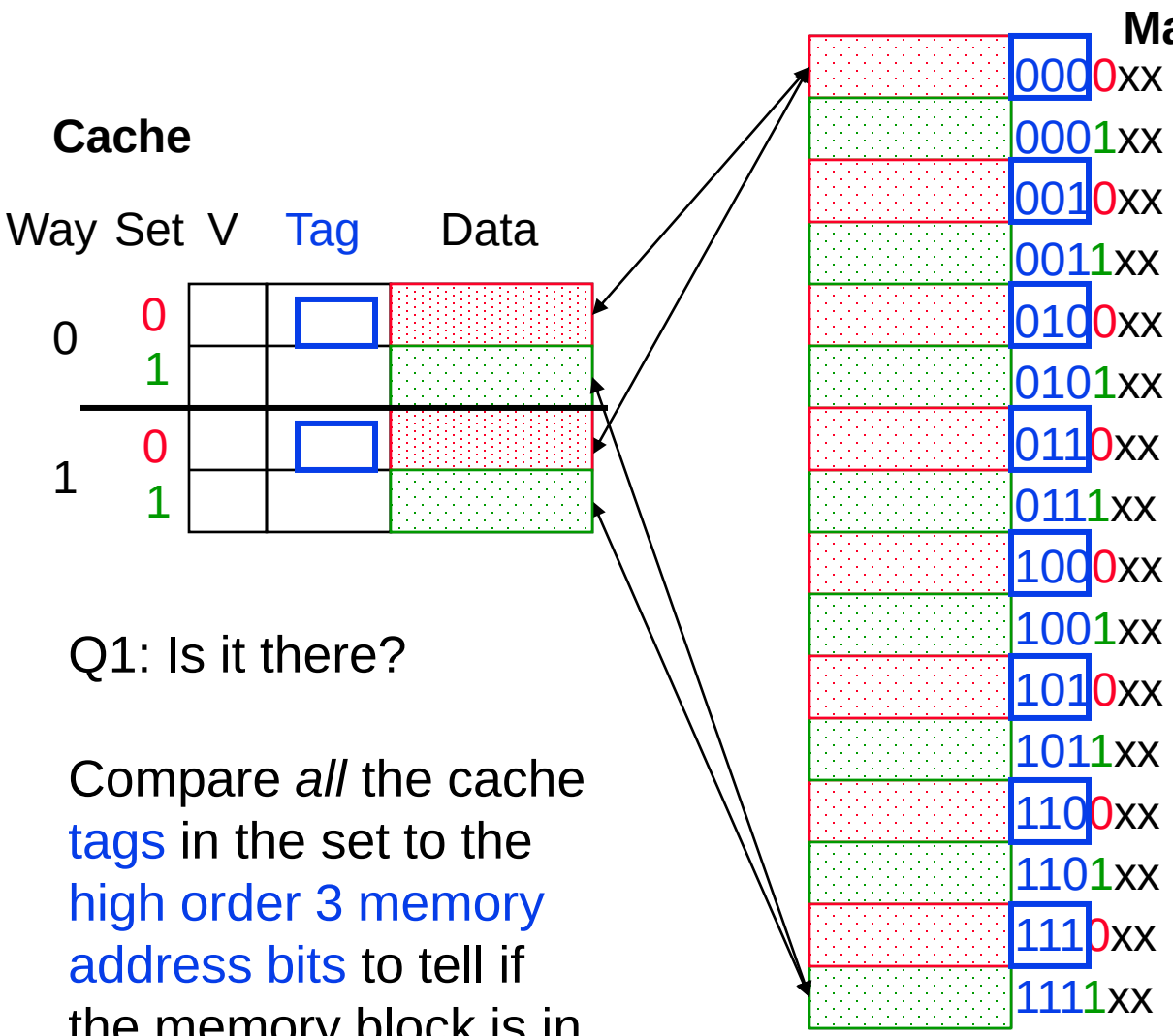
0 4 0 4 0 4 0 4



- 8 requests, 8 misses

- Ping pong effect due to conflict misses - two memory locations that map into the same cache block

Set Associative Cache Example



One word blocks
Two low order bits define the byte in the word (32b words)

Q1: Is it there?

Compare *all* the cache tags in the set to the high order 3 memory address bits to tell if the memory block is in the cache

Q2: How do we find it?

Use next 1 low order memory address bit to determine which cache set (i.e., modulo the number of sets in the cache)

Another Reference String Mapping

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

0 4 0 4 0 4 0 4

0 miss

000	Mem(0)

4 miss

000	Mem(0)
010	Mem(4)

0 hit

000	Mem(0)
010	Mem(4)

4 hit

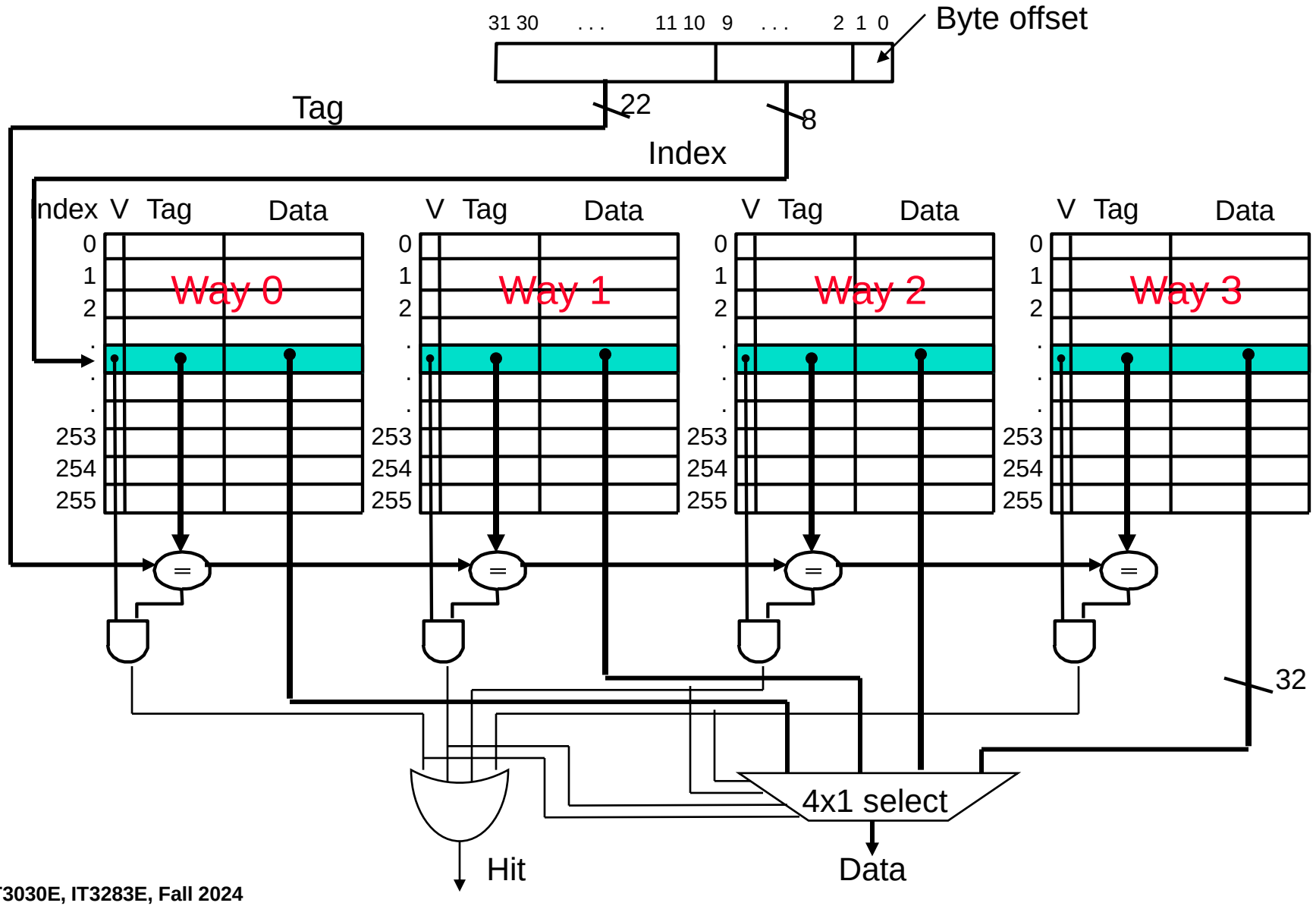
000	Mem(0)
010	Mem(4)

  - 8 requests, 2 misses

- Solves the ping pong effect in a direct mapped cache due to conflict misses since now two memory locations that map into the same cache set can co-exist!

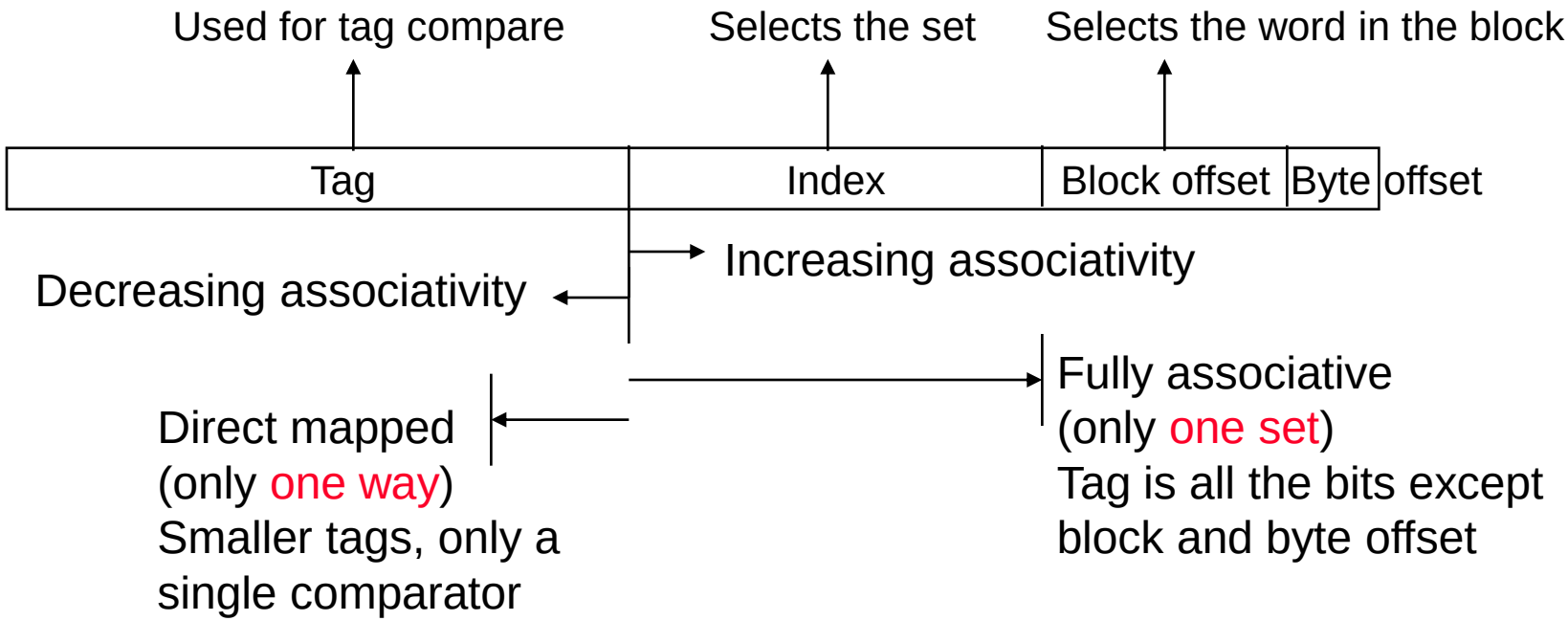
Four-Way Set Associative Cache

- 2^8 = 256 sets each with four ways (each with one block)



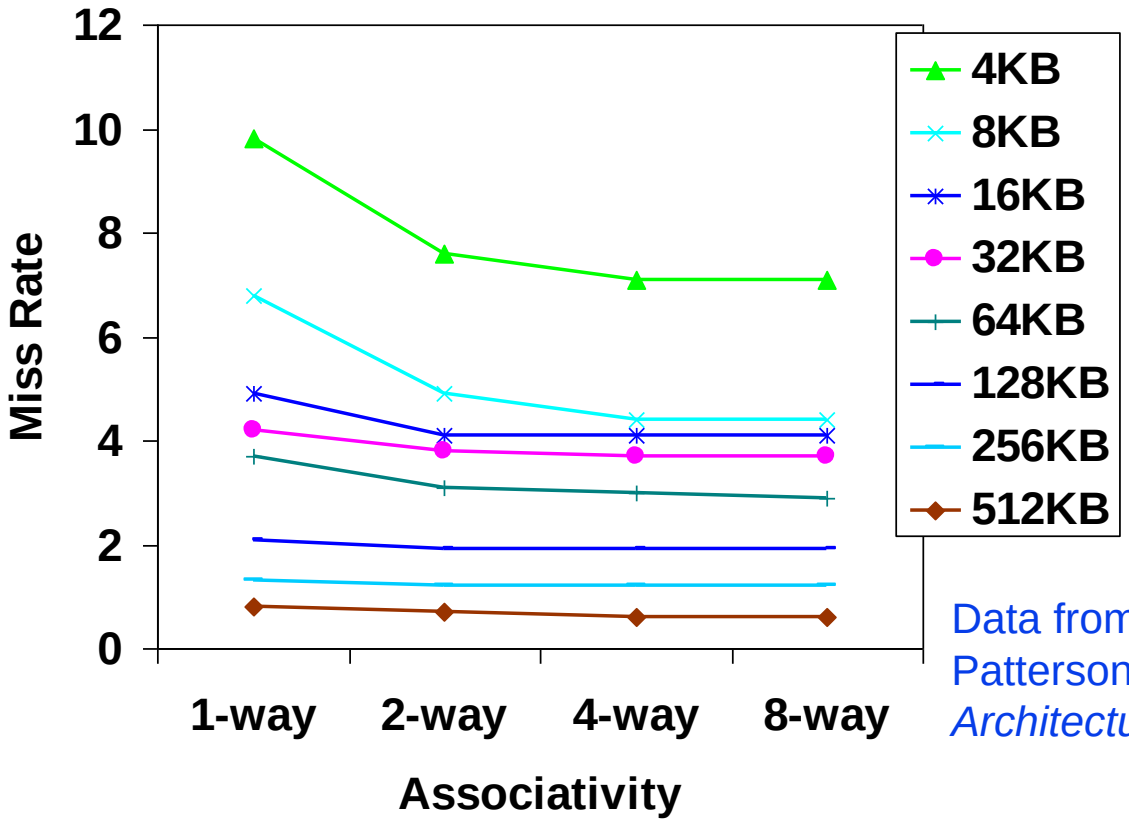
Range of Set Associative Caches

- For a fixed size cache, increase of the number of blocks per set results in decrease of the number of sets



Benefits of Set Associative Caches

- The choice of direct mapped or set associative depends on the cost of a miss versus the cost of implementation



Data from Hennessy & Patterson, *Computer Architecture*, 2003

- Largest gains are in going from direct mapped to 2-way (20%+ reduction in miss rate)

Block replacement

- Cache miss: a new block is loaded to cache, it will replace an old block

➔ Which block should be replaced?

- Direct-mapped cache: exactly one choice
- Associative cache: one of multiple blocks in the set must be selected
  - ➔ LRU scheme: (least recently used) block that has been unused the longest time is selected for replacement.
  - Mechanism for relative last time used tracking is necessary.

LRU block replacement

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

0 4 2 4 0 0 0 4

Last used

0 miss

x	000	Mem(0)

4 miss

	000	Mem(0)
x	010	Mem(4)

2 miss

x	001	Mem(2)
	010	Mem(4)

4 hit

	001	Mem(2)
x	010	Mem(4)

Last used

0 miss

x	000	Mem(0)
	010	Mem(4)

0 hit

x	000	Mem(0)
	010	Mem(4)

0 hit

x	000	Mem(0)
	010	Mem(4)

4 hit

	000	Mem(0)
x	010	Mem(4)

Reducing Cache Miss Rates #2

➔Use multiple levels of caches

- Very costly in 1990s: US$100000 or above
- Common in 2020s: ~US$500 machines

- Normally a unified L2 cache (holding both instructions and data, for each core) and a unified L3 cache shared for all cores

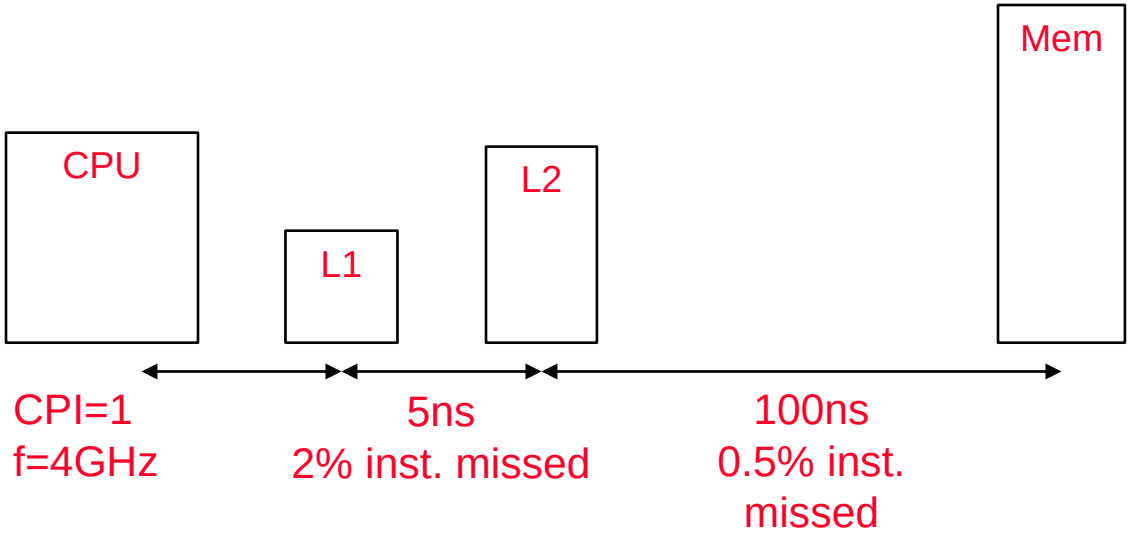
Multilevel Cache Design Considerations

- Design considerations for L1 and L2 caches are very different
  - Primary cache should focus on minimizing hit time in support of a shorter clock cycle
    - Smaller with smaller block sizes
  - Secondary cache(s) should focus on reducing miss rate to reduce the penalty of long main memory access times
    - Larger with larger block sizes
    - Higher levels of associativity
- The miss penalty of the L1 cache is significantly reduced by the presence of an L2 cache – so it can be smaller but have a higher miss rate
- For the L2 cache, hit time is less important than miss rate
  - The L2$ hit time determines L1$'s miss penalty
  - L2$ local miss rate >> than the global miss rate

Example

- Given a processor with a base CPI of 1.0 and clock rate of 4 GHz. Main memory access time is 100 ns.
  - All data references are hit in primary cache (L1).
  - Instruction miss rate of 2% in primary cache (L1).
- A new L2 is added
  - Access time from L1 to L2 is 5 ns.
  - Instruction miss rate (to main memory) reduced to 0.5%.
- What is speed-up after adding the L2?

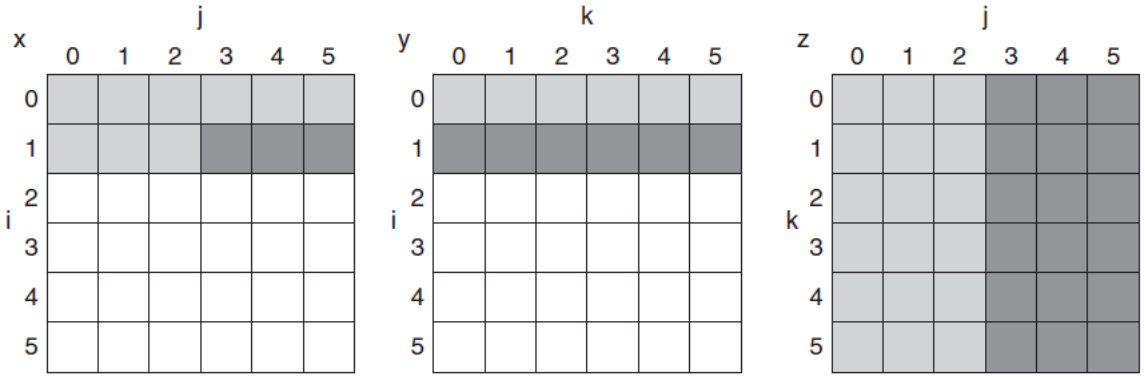
Answer



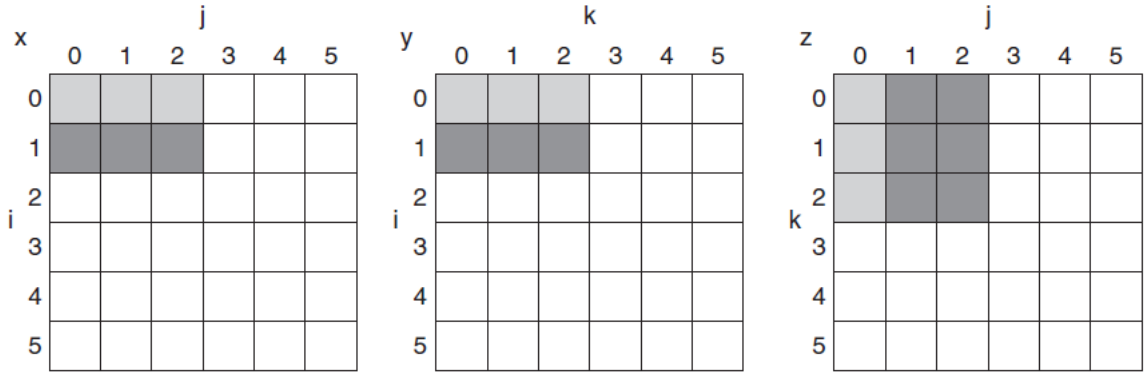
- $CPI = BaseCPI + StallCPI = BaseCPI + I_{Stall} + D_{Stall}$
- $BaseCPI = 1, D_{Stall} = 0,$
- $5\ ns = 20\ cycles, 100\ ns = 400\ cycles$
- Without L2: $I_{Stall} = 2\% * 400 = 8$
- With L2: $I_{Stall} = I_{Stall1} + I_{Stall2} = 2\% * 20 + 0.05\% * 400 = 2.4$
- $Speedup = \frac{1+8}{1+2.4} = 2.6$

Reducing cache miss: software approach

- Matrix multiplication: cache miss increases with large matrix (capacity miss)

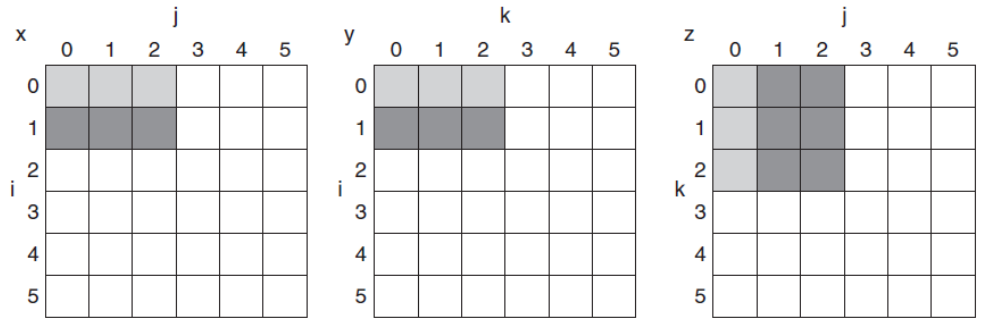


- Blocking
  - Software: localize to small block, not the whole row/column



Reducing cache miss: software approach

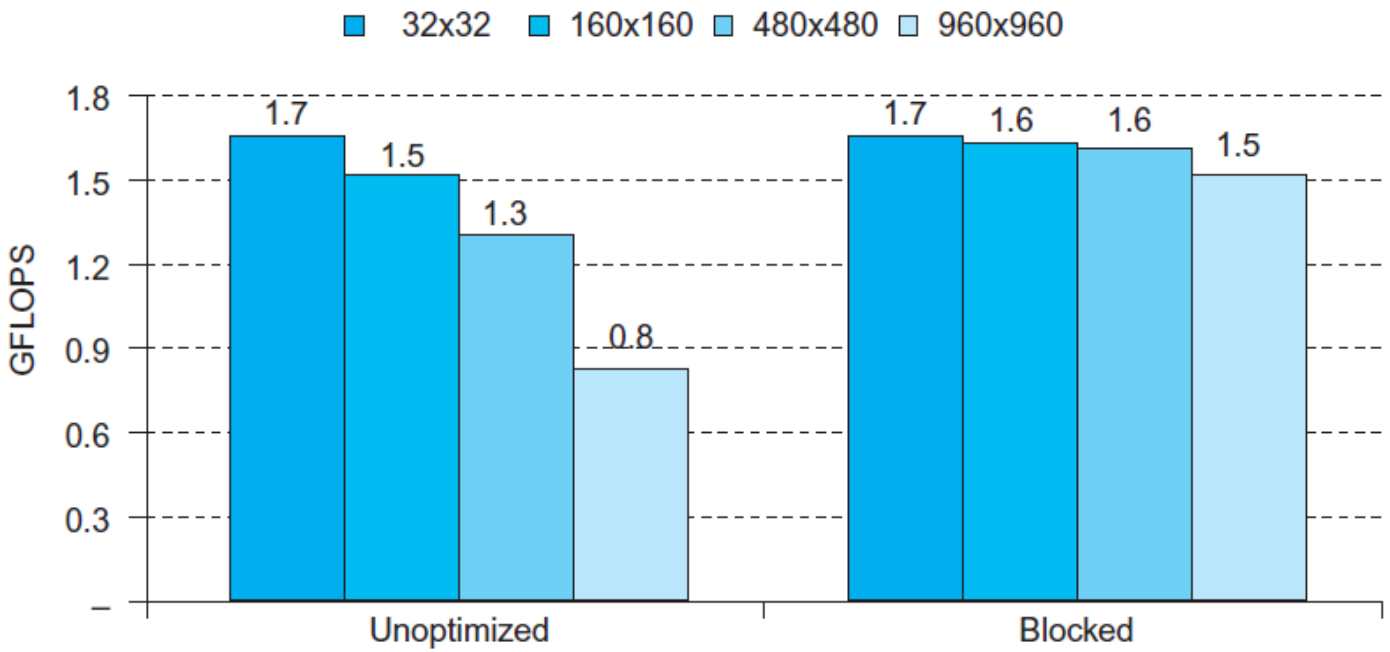
```
1  #define BLOCKSIZE 32
2  void do_block (int n, int si, int sj, int sk, double *A, double
3  *B, double *C)
4  {
5     for (int i = si; i < si+BLOCKSIZE; ++i)
6        for (int j = sj; j < sj+BLOCKSIZE; ++j)
7           {
8              double cij = C[i+j*n];/* cij = C[i][j] */
9              for( int k = sk; k < sk+BLOCKSIZE; k++ )
10                cij += A[i+k*n] * B[k+j*n];/* cij+=A[i][k]*B[k][j] */
11             C[i+j*n] = cij;/* C[i][j] = cij */
12          }
13 }
14 void dgemm (int n, double* A, double* B, double* C)
15 {
16    for ( int sj = 0; sj < n; sj += BLOCKSIZE )
17       for ( int si = 0; si < n; si += BLOCKSIZE )
18          for ( int sk = 0; sk < n; sk += BLOCKSIZE )
19             do_block(n, si, sj, sk, A, B, C);
20 }
```



n = 6
BLOCKSIZE = 3

Reducing cache miss: software approach

- Result of software optimization



Handling Cache Hits

- Read hits (I$ and D$)
  - this is what we want!

- Write hits (D$ only)
  - require the cache and memory to be consistent
    - always write the data into both the cache block and the next level in the memory hierarchy (write-through)
    - writes run at the speed of the next level in the memory hierarchy – so slow! – or can use a write buffer and stall only if the write buffer is full
  - allow cache and memory to be inconsistent
    - write the data only into the cache block (write-back the cache block to the next level in the memory hierarchy when that cache block is "evicted")
    - need a dirty bit for each data cache block to tell if it needs to be written back to memory when it is evicted – can use a write buffer to help "buffer" write-backs of dirty blocks

Handling Cache Misses (Single Word Blocks)

- Read misses (I$ and D$)
  - stall the pipeline, fetch the block from the next level in the memory hierarchy, install it in the cache and send the requested word to the processor, then let the pipeline resume
- Write misses (D$ only)
  1. stall the pipeline, fetch the block from next level in the memory hierarchy, install it in the cache (which may involve having to evict a dirty block if using a write-back cache), write the word from the processor to the cache, then let the pipeline resume

  or

  2. Write allocate – just write the word into the cache updating both the tag and data, no need to check for cache hit, no need to stall

  or

  3. No-write allocate – skip the cache write (but must invalidate that cache block since it will now hold stale data) and just write the word to the write buffer (and eventually to the next memory level), no need to stall if the write buffer isn't full

Multiword Block Considerations

- Read misses (I$ and D$)
  - Processed the same as for single word blocks – a miss returns the entire block from memory
  - Miss penalty grows as block size grows
    - Early restart – processor resumes execution as soon as the requested word of the block is returned
    - Requested word first – requested word is transferred from the memory to the cache (and processor) first
  - Nonblocking cache – allows the processor to continue to access the cache while the cache is handling an earlier miss
- Write misses (D$)
  - If using write allocate must *first* fetch the block from memory and then write the word to the block (or could end up with a "garbled" block in the cache (e.g., for 4-word blocks, a new tag, one word of data from the new block, and three words of data from the old block))

Exercise

- Given a CPU with 32 bits address and the below byte reference string.

  3, 180, 43, 2, 191, 88, 190, 14, 181, 44, 186, 253

- Identify the binary address, tag field, block index field, and hit ratio in the following cases.
  - The CPU has direct-mapped cache of 16 one-word blocks.
  - The CPU has direct-mapped cache of 8 two-word blocks.

Exercise

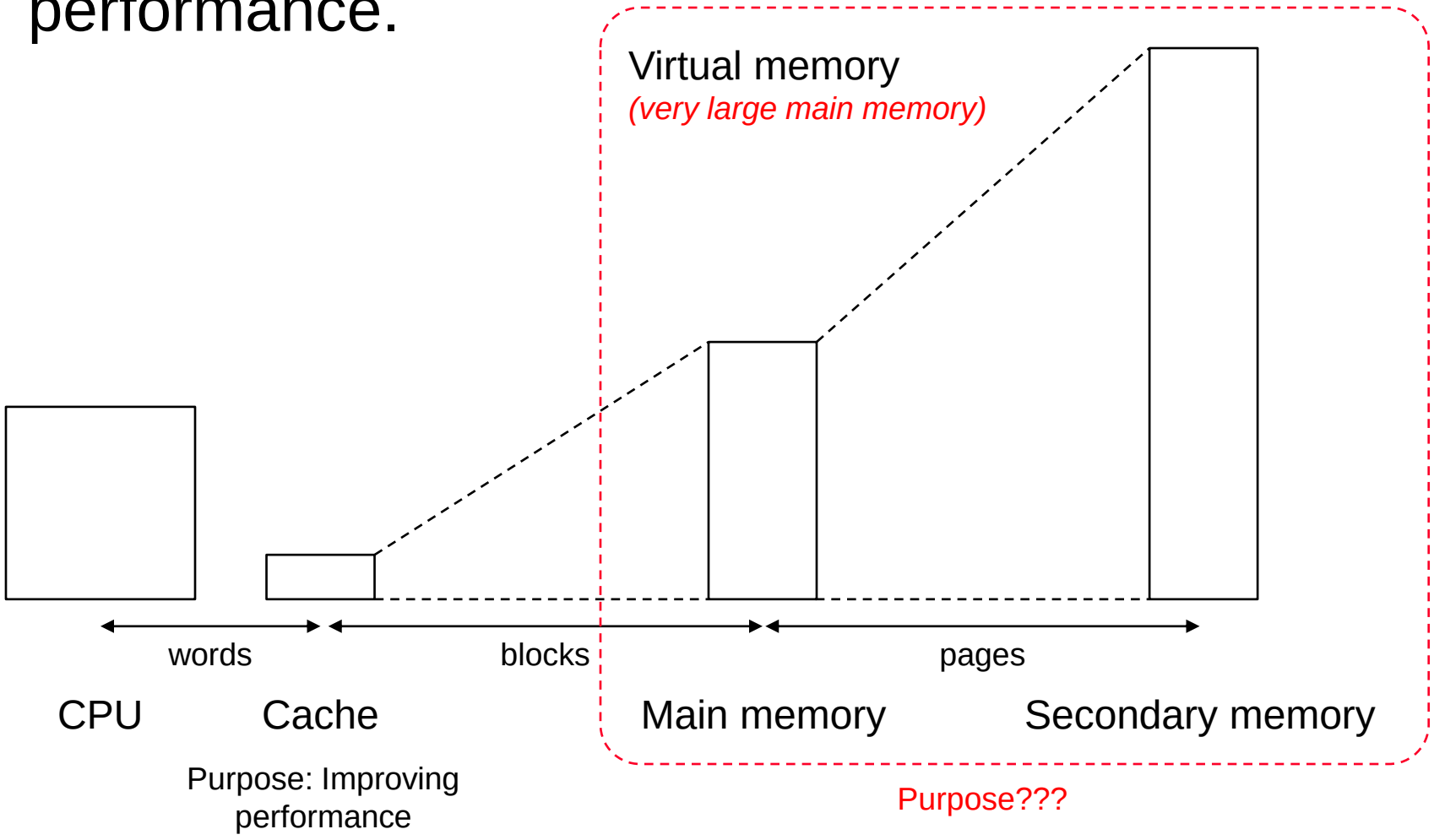
- Given a CPU with 32 bits address and the below byte reference string.

  3, 180, 43, 2, 191, 88, 190, 14, 181, 44, 186, 253

- The CPU has direct-mapped cache with a total of 8 data words. Miss penalty is 25 cycles.
- Which of the following designs is optimal given the above reference string?
  - 8x one-word blocks, access time of 2 cycles
  - 4x two-word blocks, access time of 3 cycles.
  - 2x four-word blocks, access time of 5 cycles.

Virtual Memory

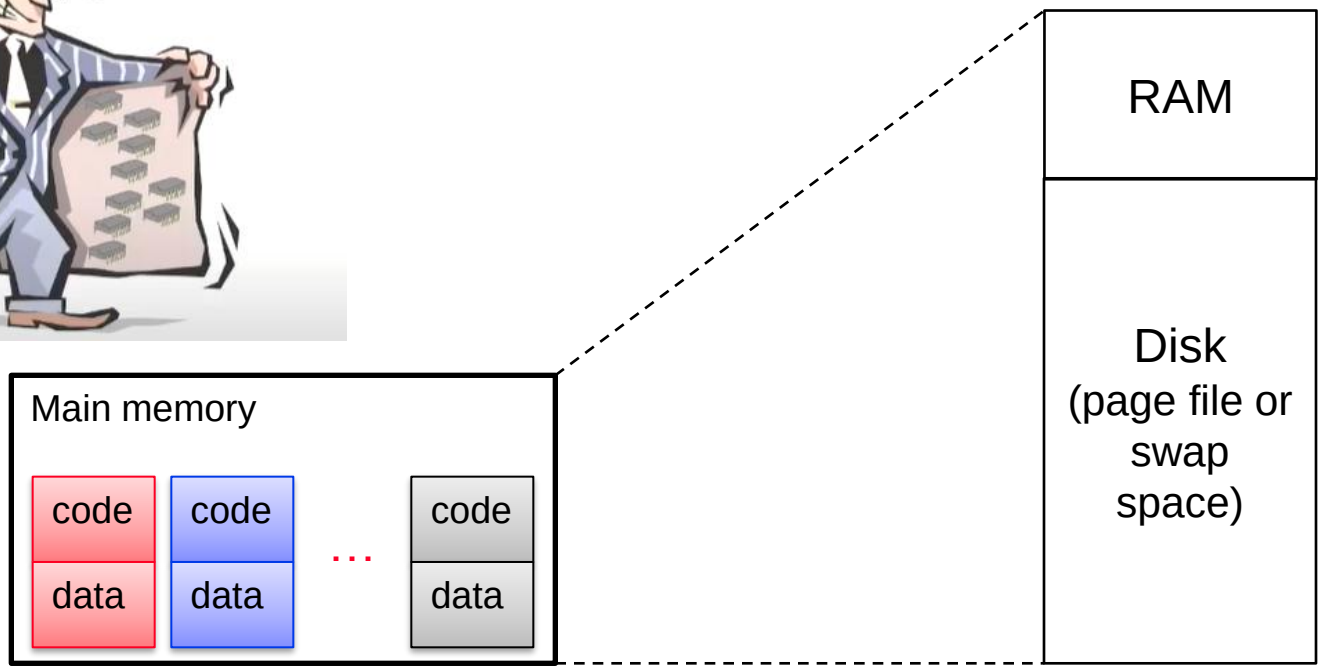
- Main memory (RAM) can be used as a “cache” for secondary storage (disk), but not mainly for performance.



Virtual Memory

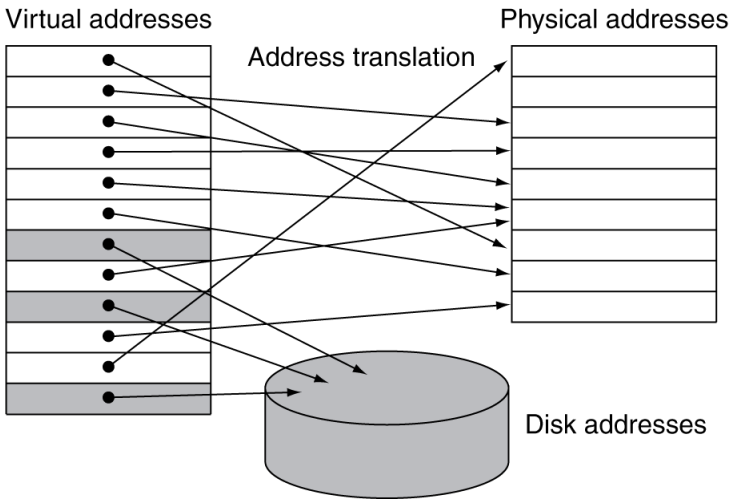
- Multiple programs (processes) share one main memory
- Large programs can run on computer with small main memory

Chris Terman MIT 6.004

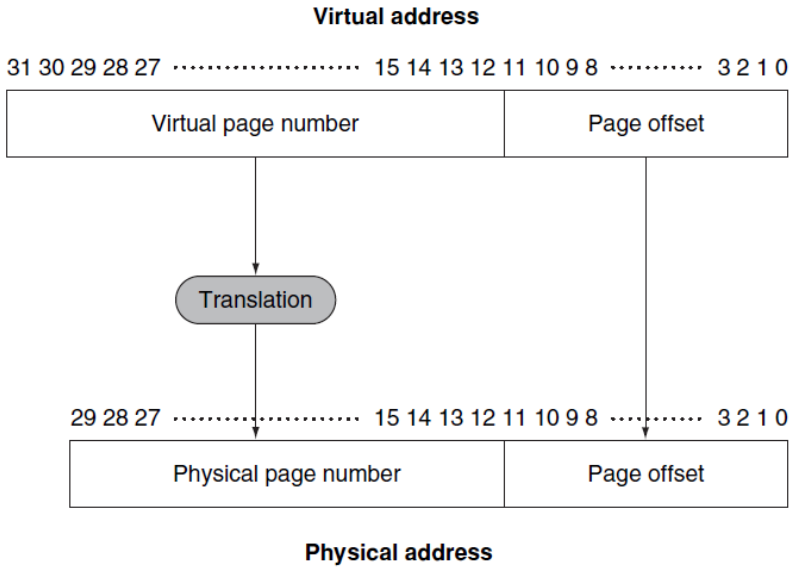


Relocation and Address translation

- Programs are located and run in virtual memory.
  - Each program has its own continuous address space (virtual address).
  - Virtual address are mapped to physical address via translation.
  - Memory is organized in pages of fixed size (4KB - 64KB).



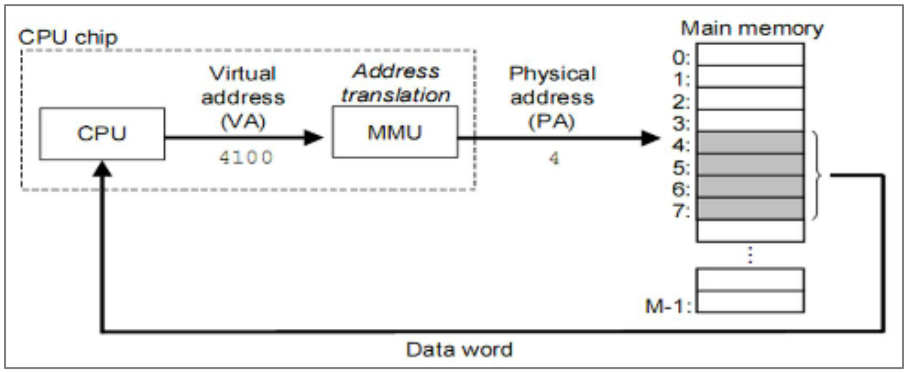
Do programs need to be allocated in contiguous physical pages?



Example: CPU with 32 bit address, but the computer has only 1GB of physical memory

Address Translation

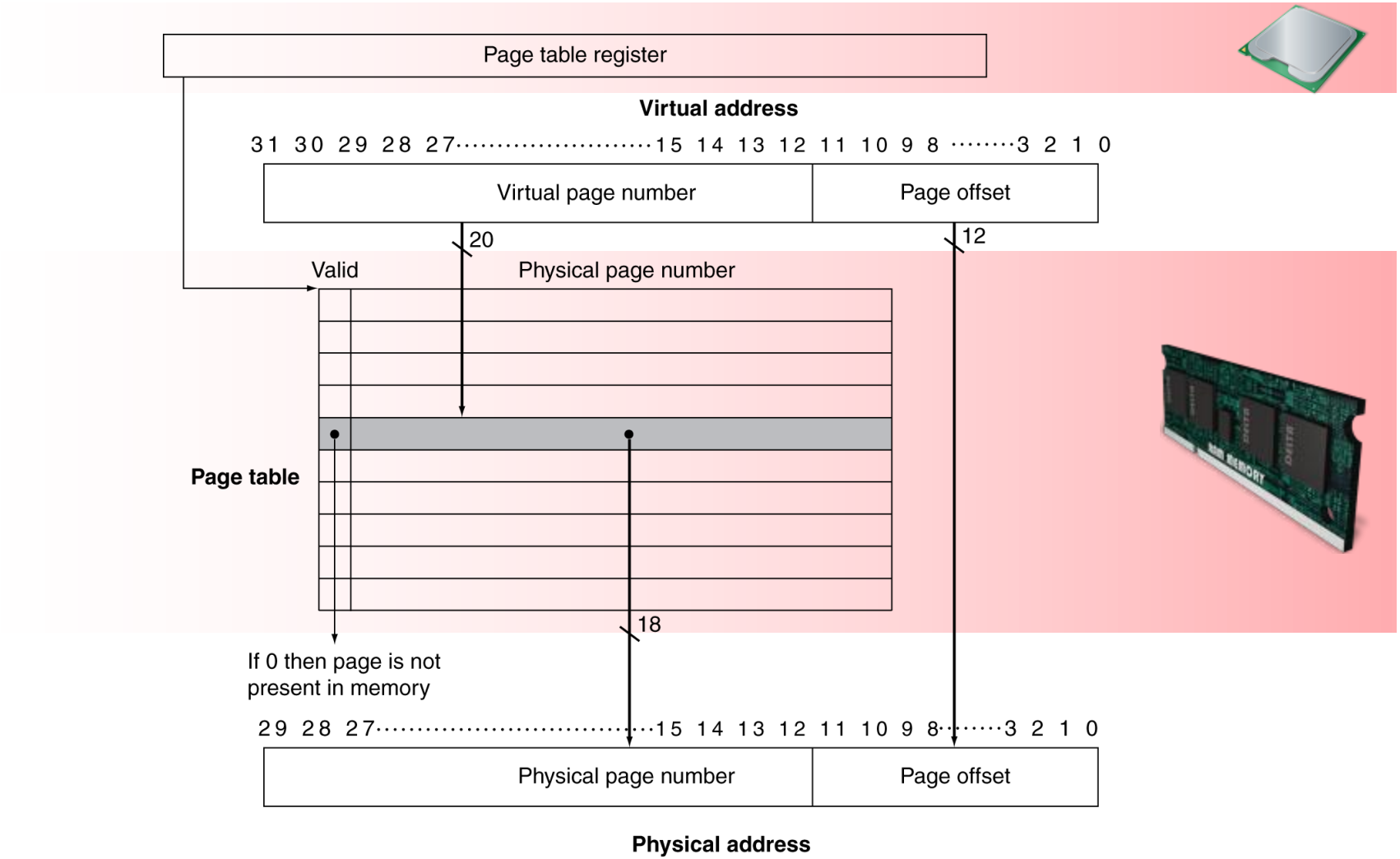
- CPU accesses a memory location based on virtual address: Virtual page number + Page offset
- If the virtual page number can be translated to physical page number (hit) → memory access can be done properly.
- Otherwise (miss): page fault → very expensive operation
  - New physical page is allocated for the running process
    - If no free physical pages is available, move an "old" page to disk to make space for the new page ➔ page replacement
  - Content for the new page is loaded from disk



Page Tables

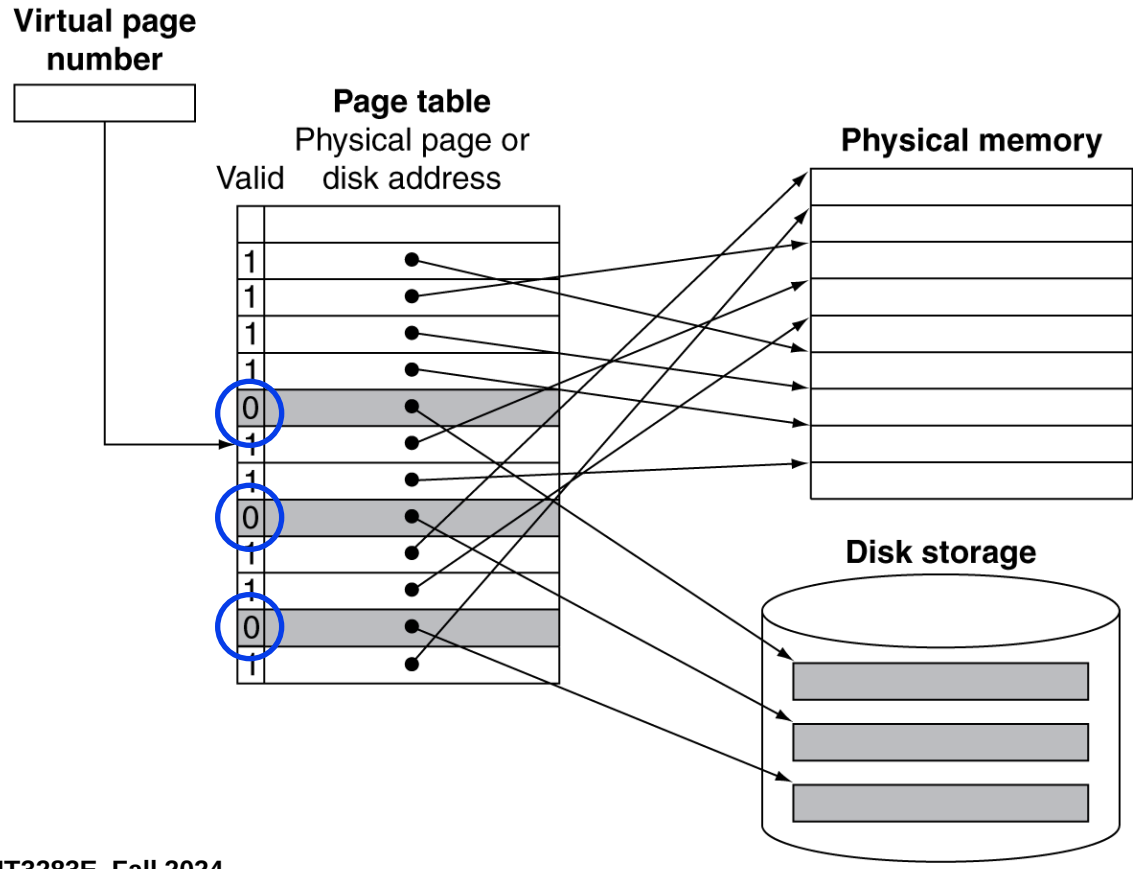
- Stores placement information of each program (process)
  - Array of **P**age **T**able **E**ntries, indexed by virtual page number
  - Located in main memory
  - **Page table register** *in CPU* points to page table in *physical memory*
- If page is present in memory
  - PTE stores the physical page number
  - Plus status bits (referenced, dirty, …)
- If page is not present
  - PTE can refer to location in swap space on disk

Translation Using a Page Table



Page Fault Penalty and Storage Mapping

- On page fault, the page must be fetched from disk
  - Usually together with page replacement
  - Takes millions of clock cycles
  - Handled by OS code



Memory pages can be stored in disk page-file or swap space
Managed by OS

Issues in virtual memory design

- Minimize cost for page fault and data write: minimize page fault rate, and minimize disk write frequency
  - Fully associative placement
  - Smart replacement algorithms
  - Write back approach
- Fast address translation: this happens for every memory access, it must be as fast as possible
  - Caching the page table: Translation Look-aside Buffer (TLB)

Page Replacement and Writes

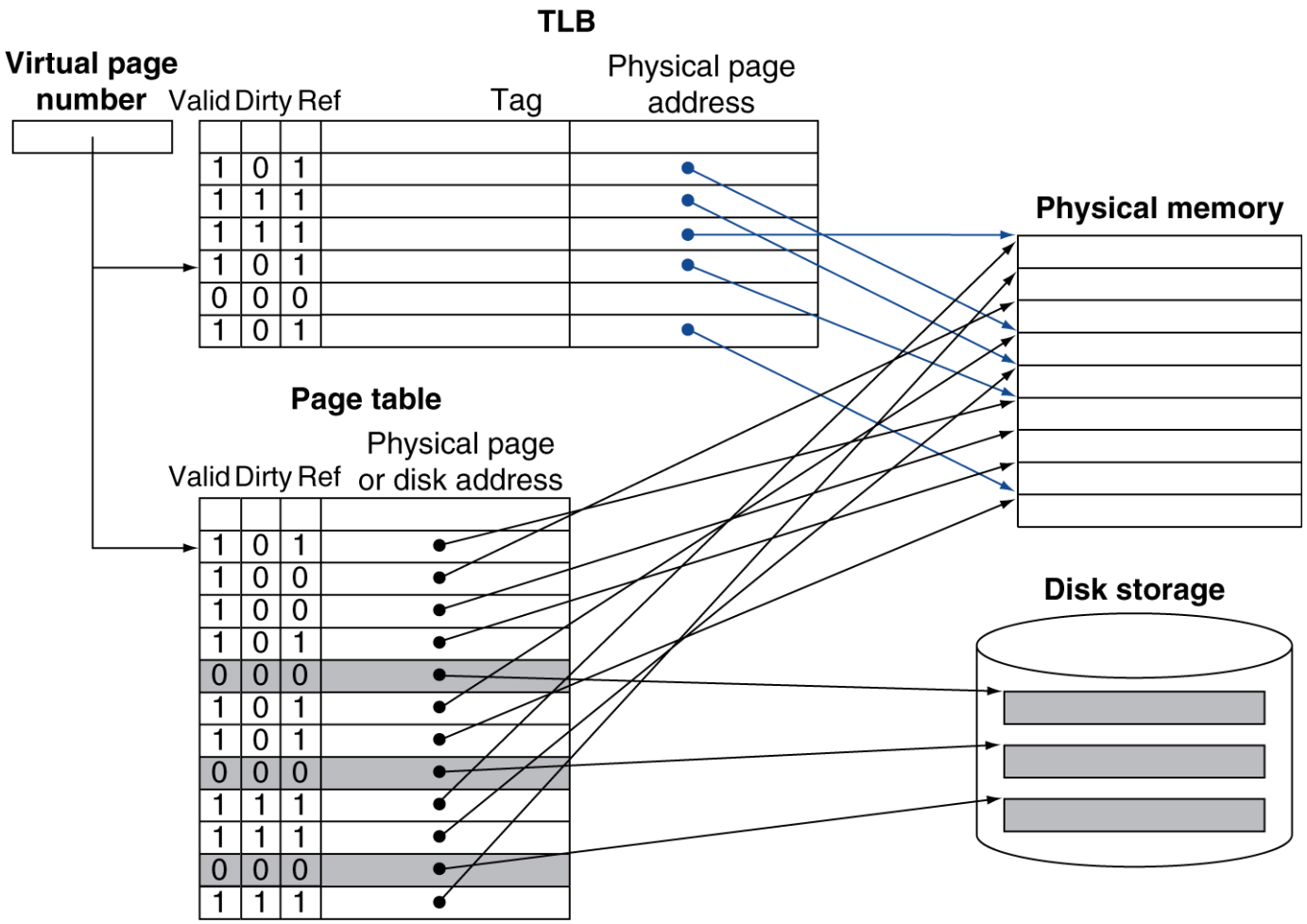
- Least-recently used (LRU) for page replacement
  - Can be quite slow when number of page is large
  - **Reference bit** (aka use bit) in PTE set to 1 on access to page
  - Periodically cleared to 0 by OS
  - Pages with reference bit = 0 are considered for replacement
- Disk writes take millions of cycles
  - Disk write is slow and should be done in batches of data.

  →Write through is impractical
  - Use write-back
  - **Dirty bit** in PTE set when page is written

Fast Translation Using TLB

- Address translation: two consecutive memory references
  - One to access the PTE, then the actual memory access
  - Has good locality → page table can be cached
- TLB (Translation Look-aside Buffer)
  - New component inside CPU
  - Provides fast access to the most recent PTEs
  - Typical: 16–512 PTEs, 0.5–1 cycle for hit, 10–100 cycles for miss, 0.01%–1% miss rate
  - Only contains PTEs corresponding to physical pages

Fast Translation Using a TLB



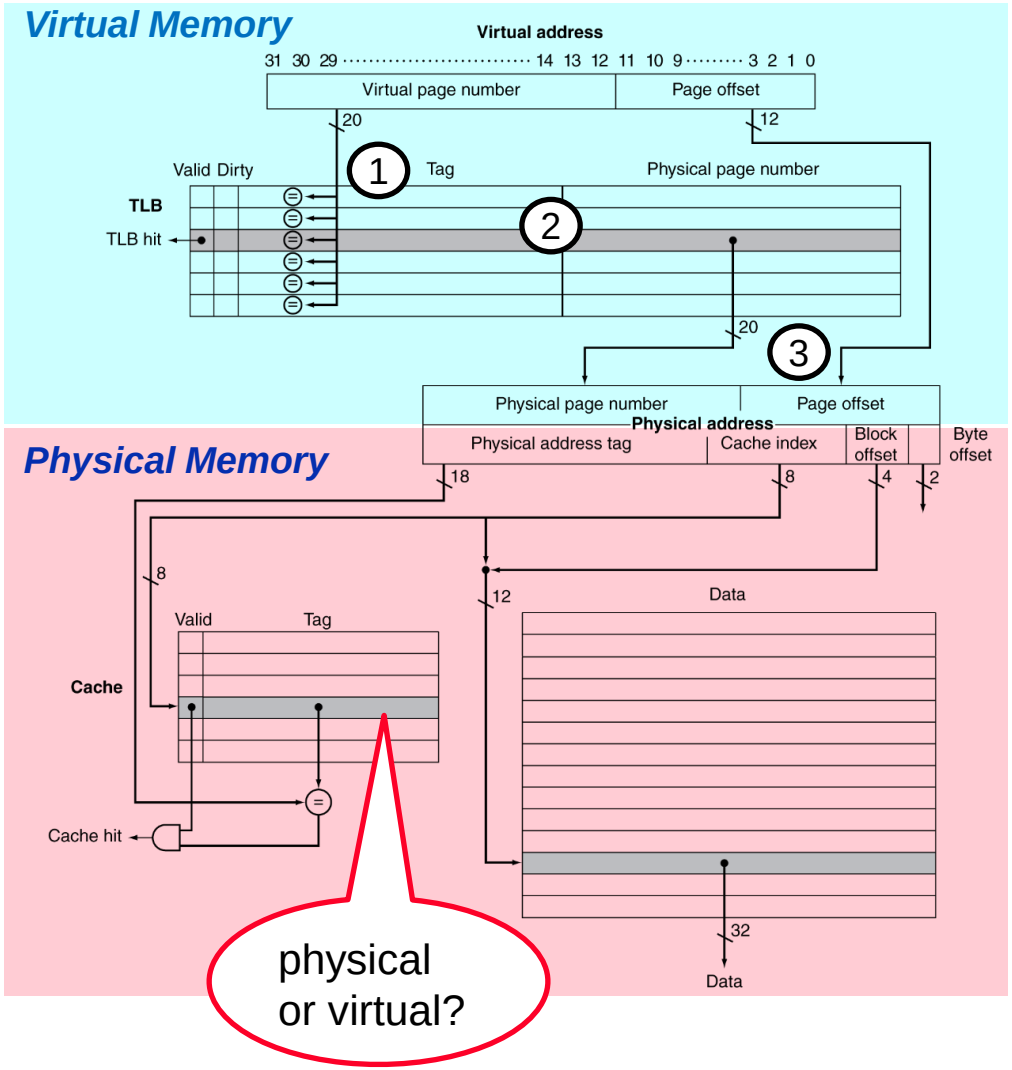
TLB Miss Handler

- TLB miss indicates
  - Page present, but PTE not in TLB
  - Page not present
- Page present:
  - Handler copies PTE from memory to TLB
  - Then restarts instruction
- If page not present: page fault will occur

Page Fault Handler

- Use faulting virtual address to find PTE (currently not valid)
- Locate page on disk
- Choose a page in physical memory to replace
  - If dirty, write-back the chosen page to disk first
- Read page into memory and update page table
- Make process runnable again
  - Restart from faulting instruction

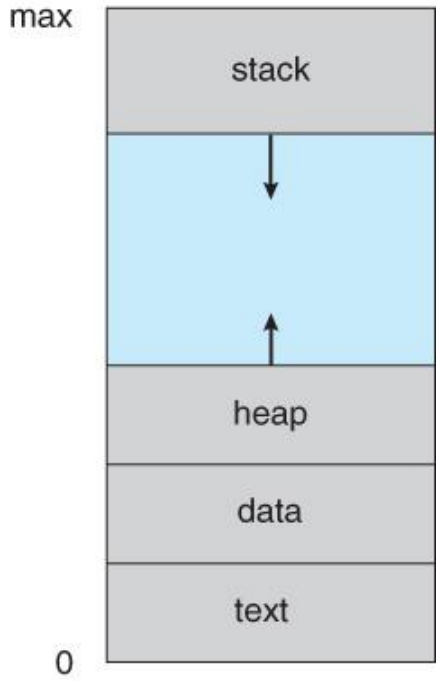
TLB and Cache Interaction



- Physically addressed cache
  - Cache uses physical address
  - Need to translate before cache lookup
  - Slow performance
- Virtually addressed cache
  - Skip TLB when in normal cache access
  - Aliasing problem:
    - Different virtual addresses for shared physical address
- Compromise: virtually indexed but physically tagged
  - No alias, but complicated design

Process and Memory protection

- Process: an instance of a program in execution
- (*Take the IT3070E - OS course for more details)*
- With separate memory space (virtual)
- Share the common physical memory
- Important data of a process: PC, register's values, page table
- Memory must be protected
  - Read protection: processes not able to read each other's memory
  - Write protection: processes prohibited from writing to other process's memory
- *Super process: the OS*



Memory Protection

- Read protection
  - Virtual pages of separate processes map to disjoint physical pages.
  - Placing page tables in protected address space of OS → processes are not allowed to modify page tables.
- Sharing data
  - OS creates a page table entry for a virtual page of one process to point to physical page of another page.
  - Write protection: use the write protection bit.
- Hardware support for protection (used by OS)
  - Special privileged supervisor mode (aka kernel mode) and privileged instructions.
  - Page tables and other state information only accessible in supervisor mode.
  - System call exception (e.g., ecall in RISC-V) to go from user mode to supervisor mode.

Summary

- Memory hierarchy and the locality principal
- Cache design
  - Direct mapped
  - Set associative
  - Memory access when cache hit and miss
- Virtual memory
  - Address translation
  - TLB
  - Protection