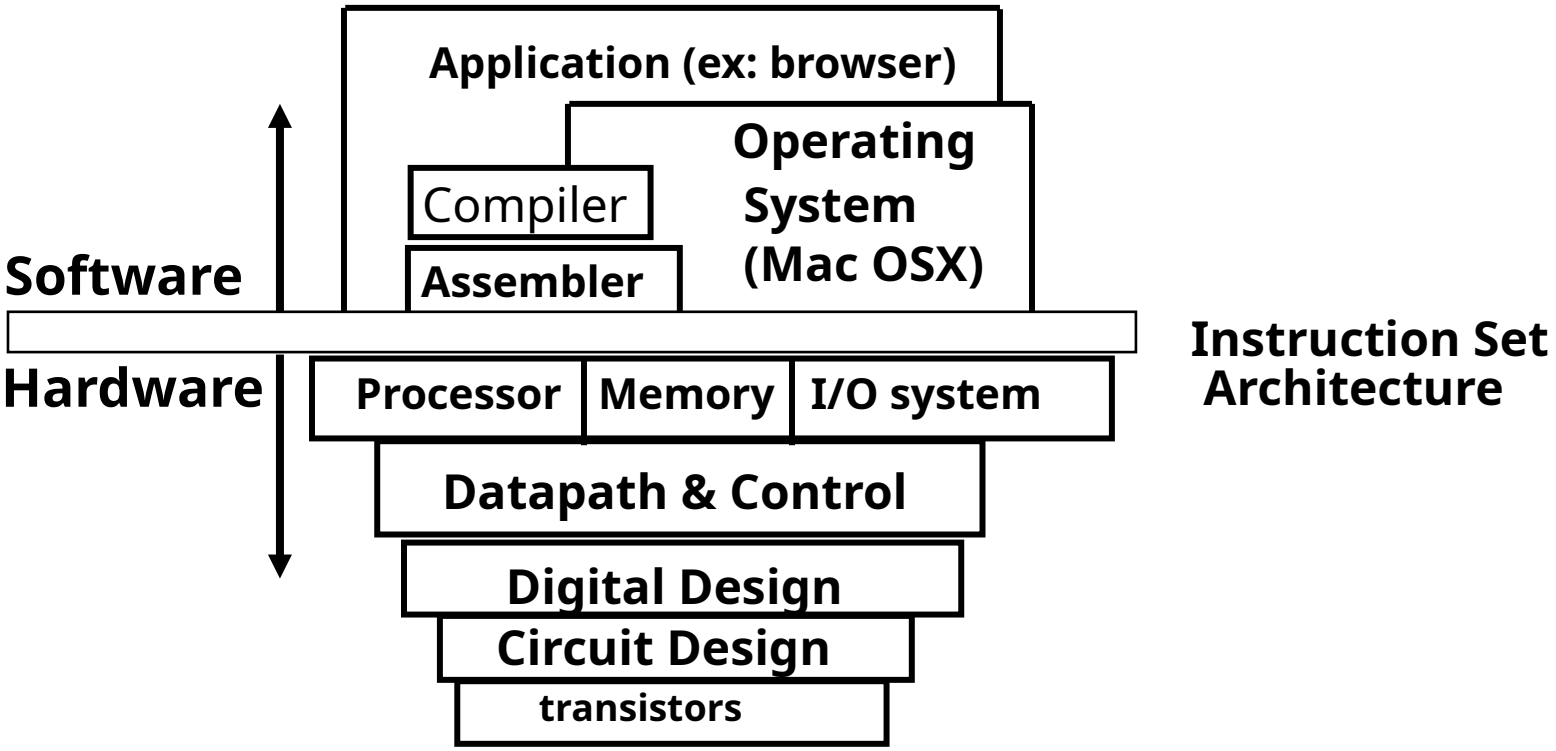
Chapter 5

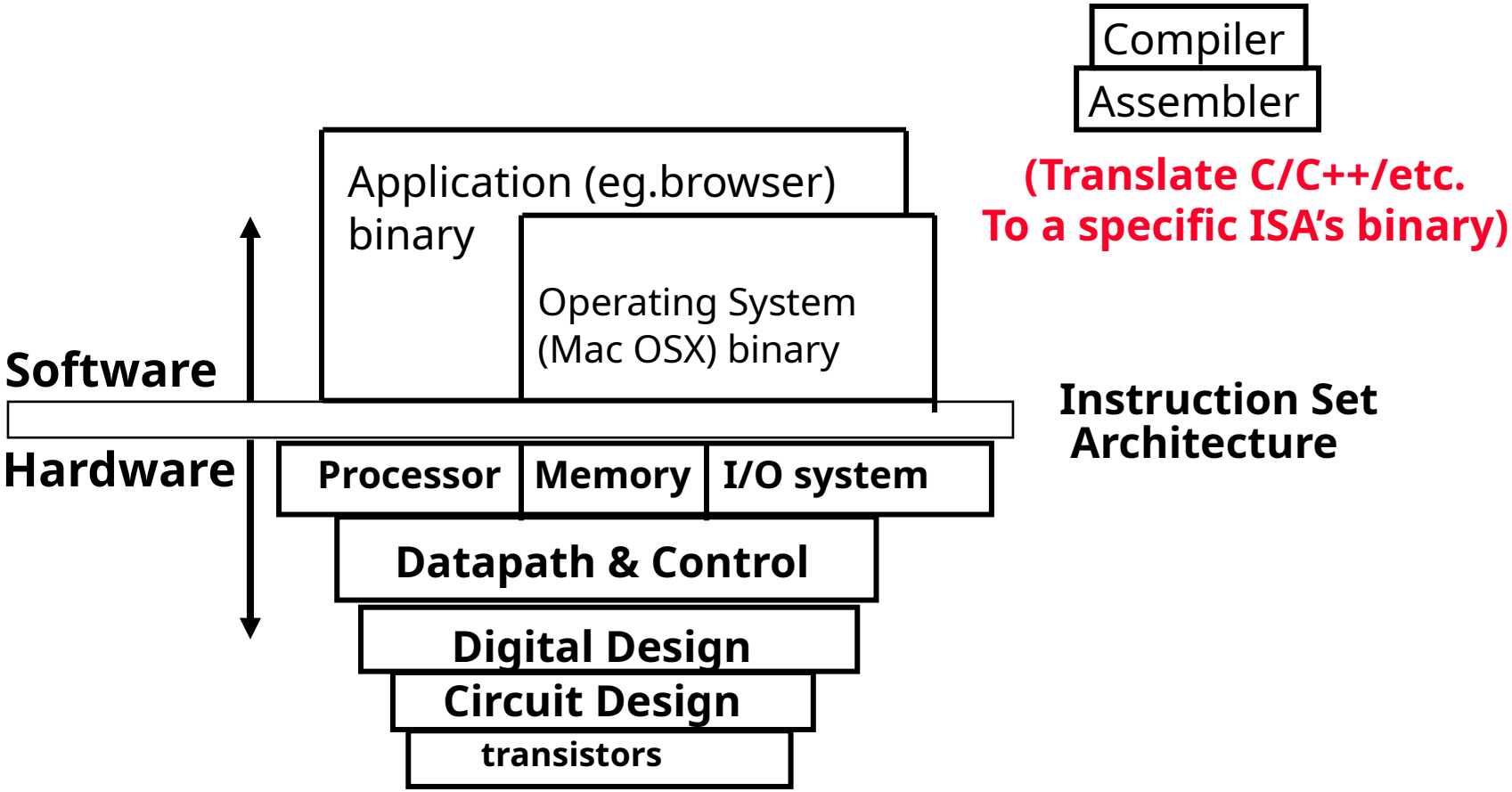
Large and Fast: Exploiting Memory Hierarchy

[Some slides adapted from A. Sprintson, M. Irwin, D. Paterson and others]

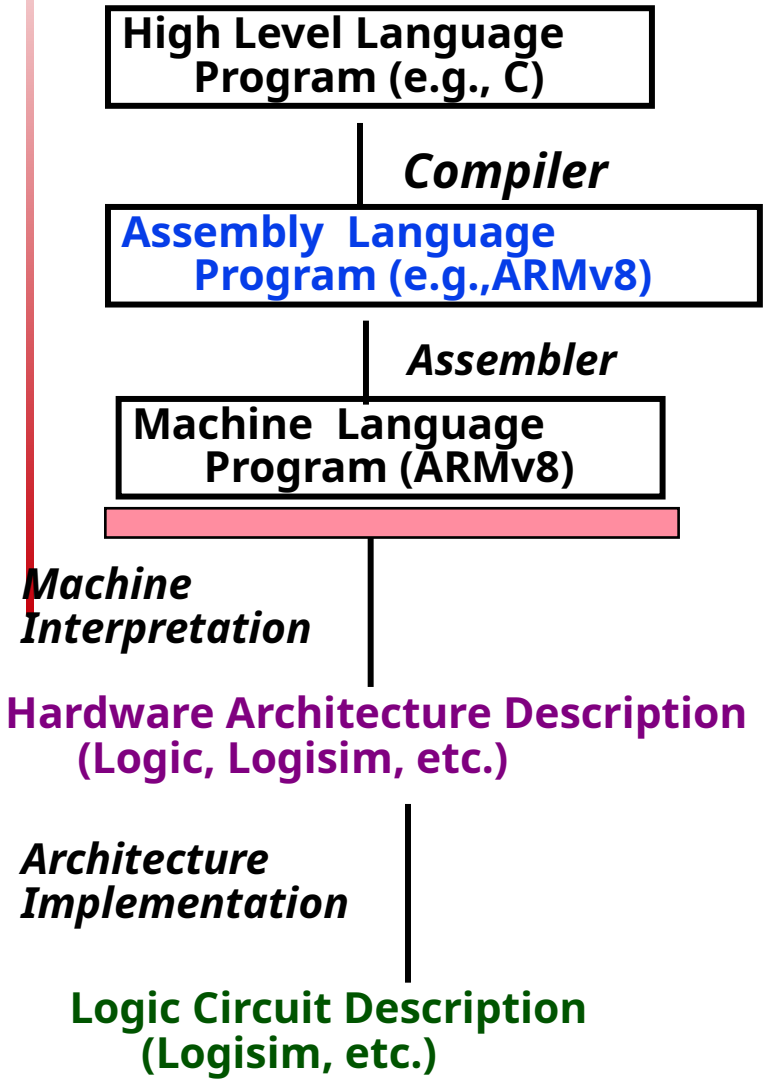
Computer organization



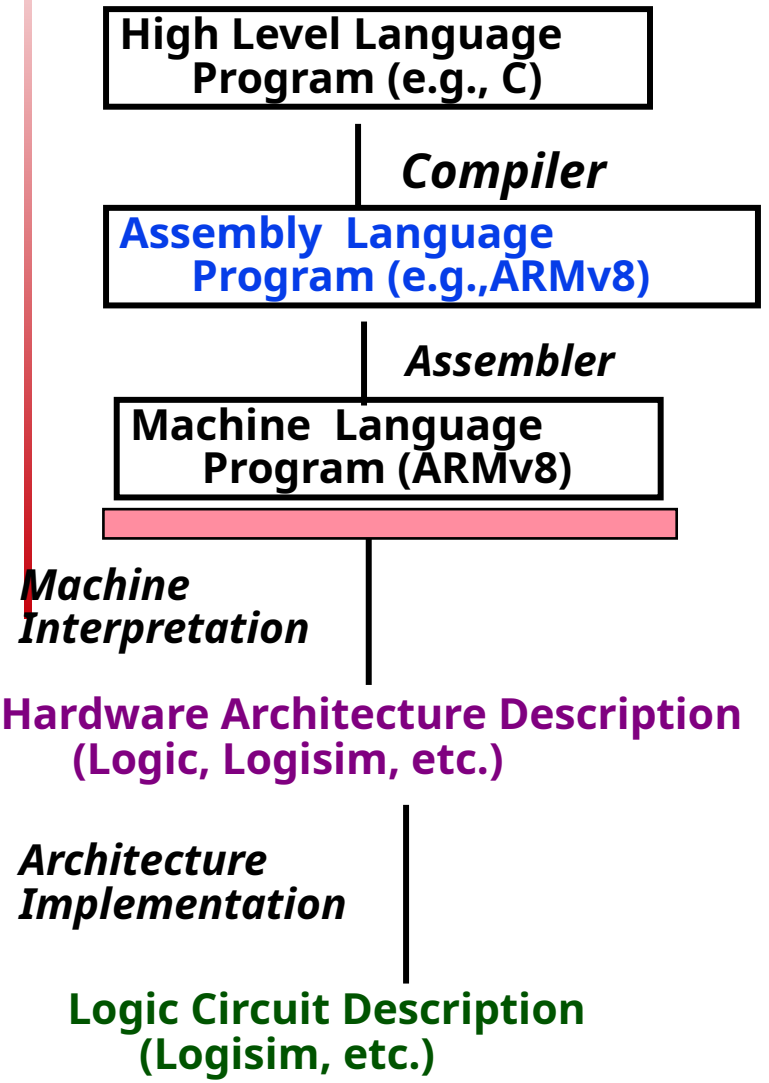
Computer organization



Levels of Representation

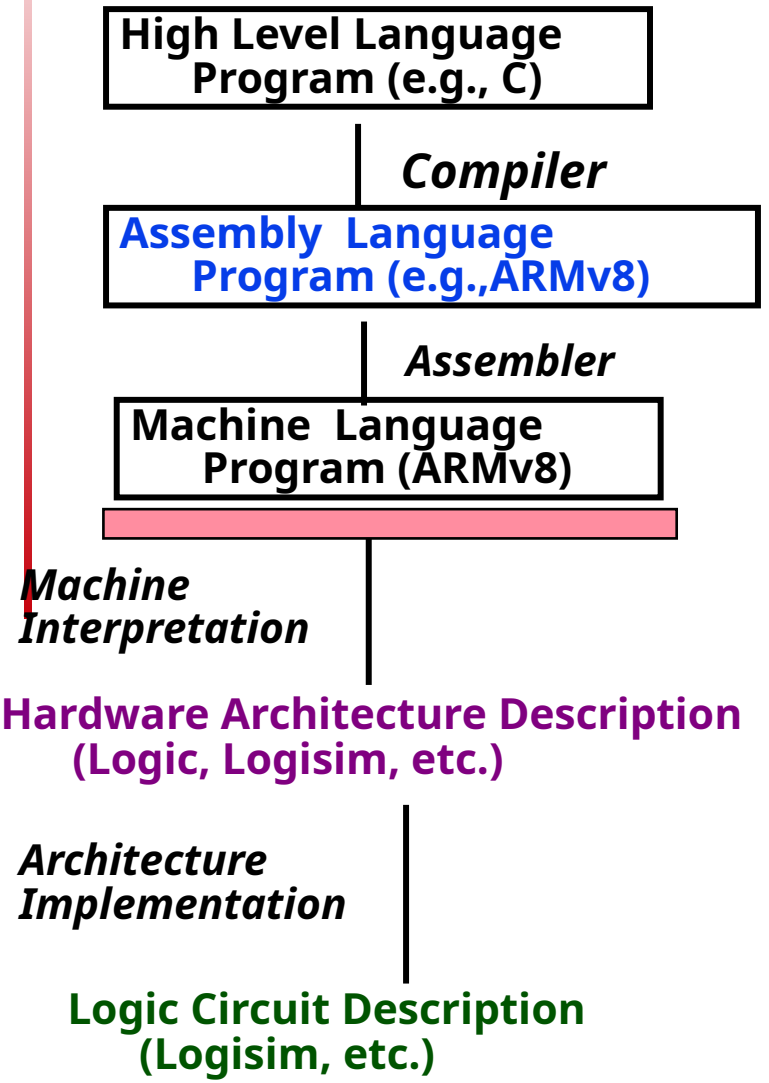


Levels of Representation




```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

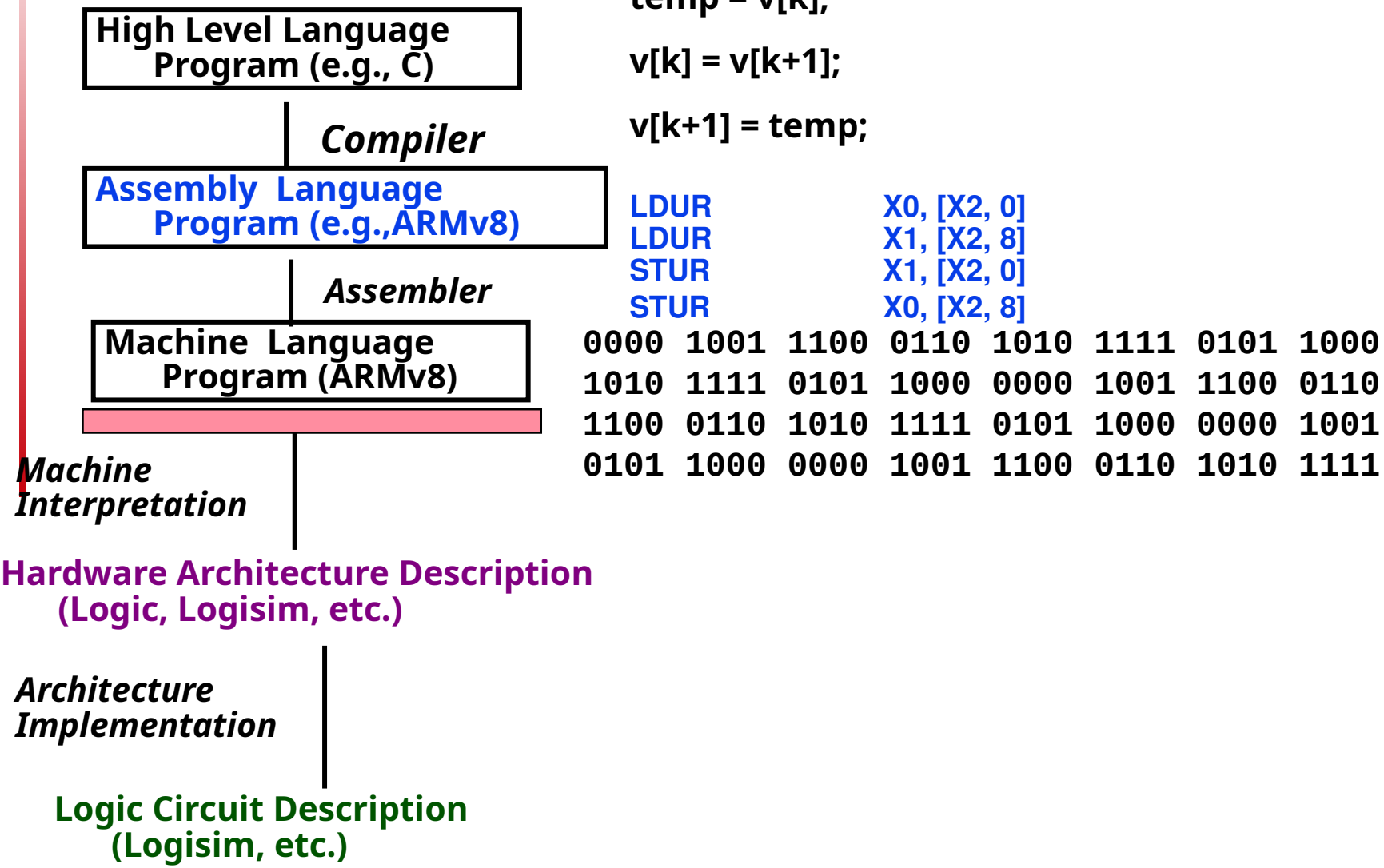
Levels of Representation




```
temp = v[k];

v[k] = v[k+1];

v[k+1] = temp;
```

```
LDUR    X0, [X2, 0]
LDUR    X1, [X2, 8]
STUR    X1, [X2, 0]
STUR    X0, [X2, 8]
```

Levels of Representation



Levels of Representation

High Level Language Program (e.g., C)

Compiler

Assembly Language Program (e.g.,ARMv8)

Assembler

Machine Language Program (ARMv8)

Machine Interpretation

Hardware Architecture Description (Logic, Logisim, etc.)

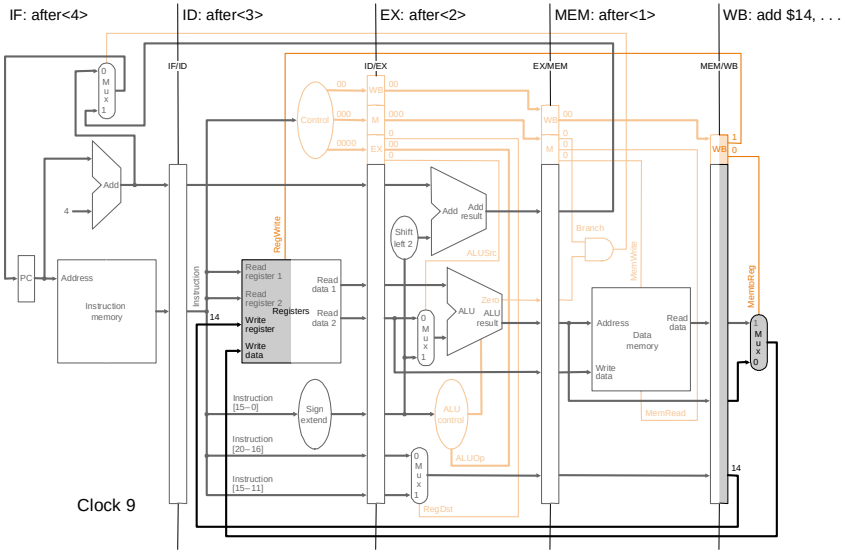
Architecture Implementation

Logic Circuit Description (Logisim, etc.)

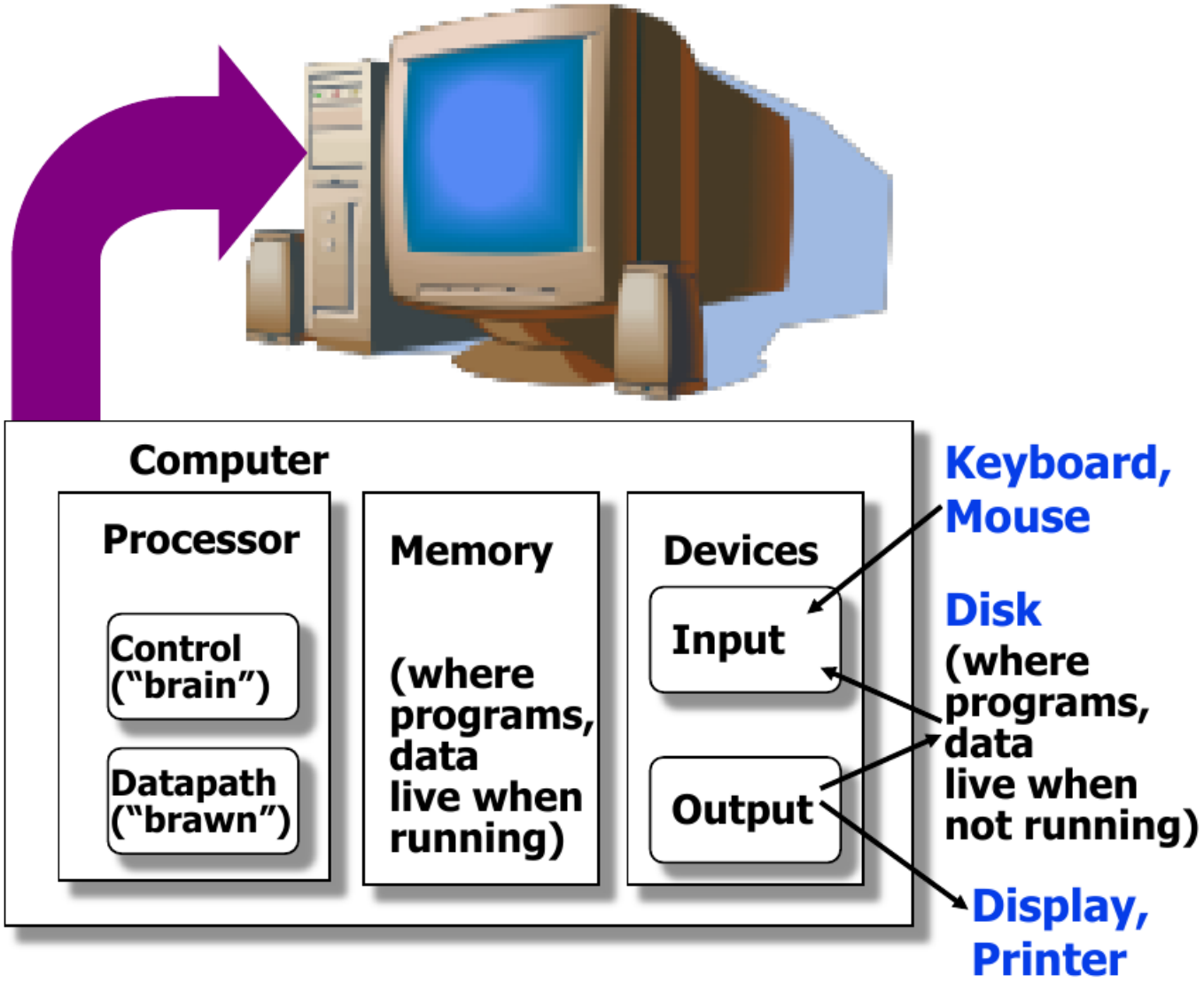
```
temp = v[k];

v[k] = v[k+1];

v[k+1] = temp;
```

```
LDUR        X0, [X2, 0]
LDUR        X1, [X2, 8]
STUR        X1, [X2, 0]
STUR        X0, [X2, 8]
```

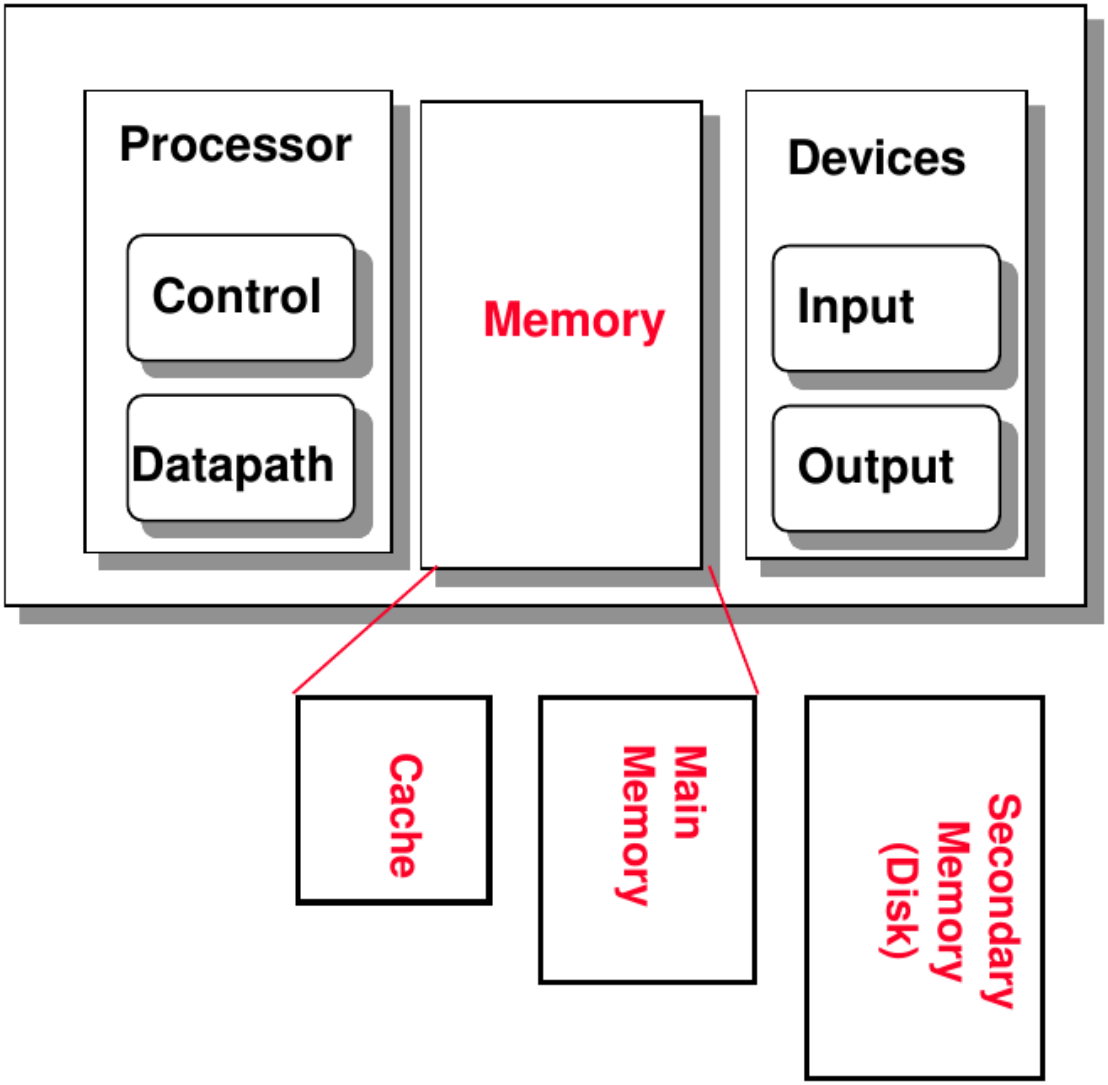
```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```



Anatomy: Components of any Computer

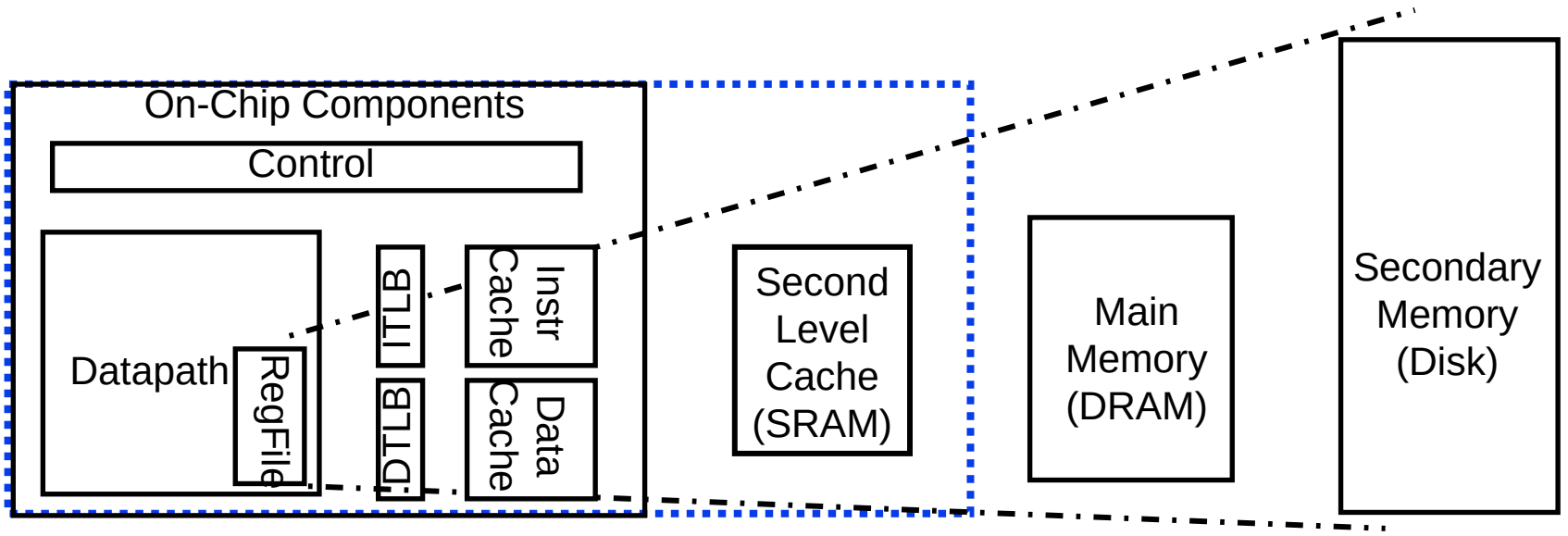


Review: Major Components of a Computer



A Typical Memory Hierarchy

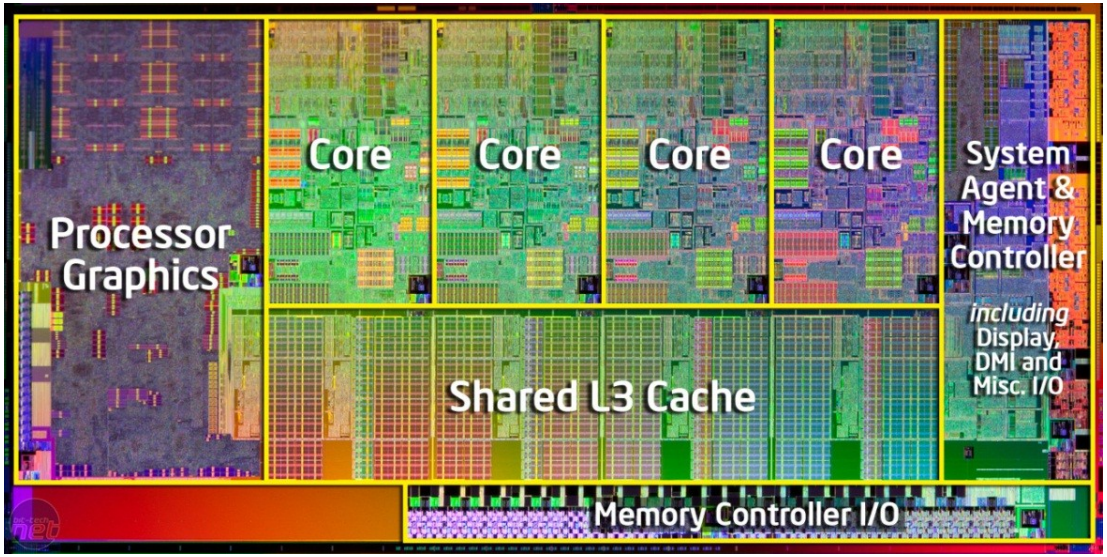
- By taking advantage of the principle of locality:
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.



Speed (ns):	.1's	1's	10's	100's	1,000's
Size (bytes):	100's	K's	10K's	M's	G's
Cost:	highest				lowest

Sandy Bridge/Ivy Bridge (Intel i7) Die Photo

- 32KB L1 I and L1 D caches (per core)
- 256KB L2 caches (per core)
- (up to) 15MB L3 cache (shared)



Memory Hierarchy Technologies

- Random Access Memories (RAMs)
  - "Random" is good: access time is the same for all locations
  - DRAM: Dynamic Random Access Memory
    - High density (1 transistor cells), low power, cheap, slow
    - Dynamic: need to be "refreshed" regularly (~ every 4 ms)
  - SRAM: Static Random Access Memory
    - Low density (6 transistor cells), high power, expensive, fast
    - Static: content will last "forever" (until power turned off)
  - Size: DRAM/SRAM ratio of *4 to 8*
  - Cost/Cycle time: SRAM/DRAM ratio of *8 to 16*

- "Non-so-random" Access Technology
  - Access time varies from location to location and from time to time (e.g., disk, CDROM, flash)

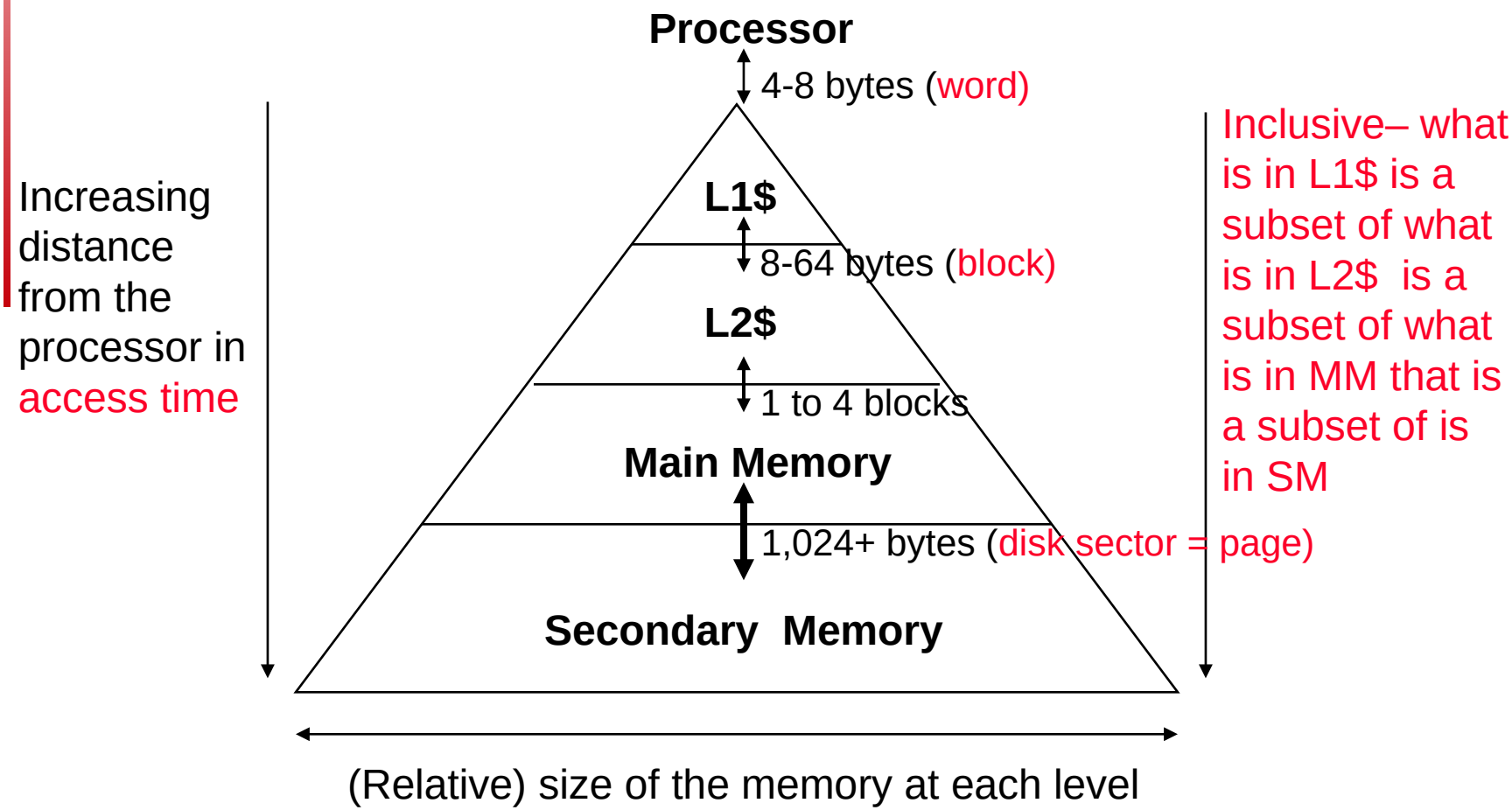
RAM Memory Uses and Performance Metrics

- Caches use *SRAM* for speed
- Main Memory uses *DRAM* for density

- Memory performance metrics
  - Latency: Time to access one word
    - *Access Time*: time between request and when word is read or written (read access and write access times can be different)
  - Bandwidth: How much data can be supplied per unit time
    - width of the data channel * the rate at which it can be used

The Memory Hierarchy

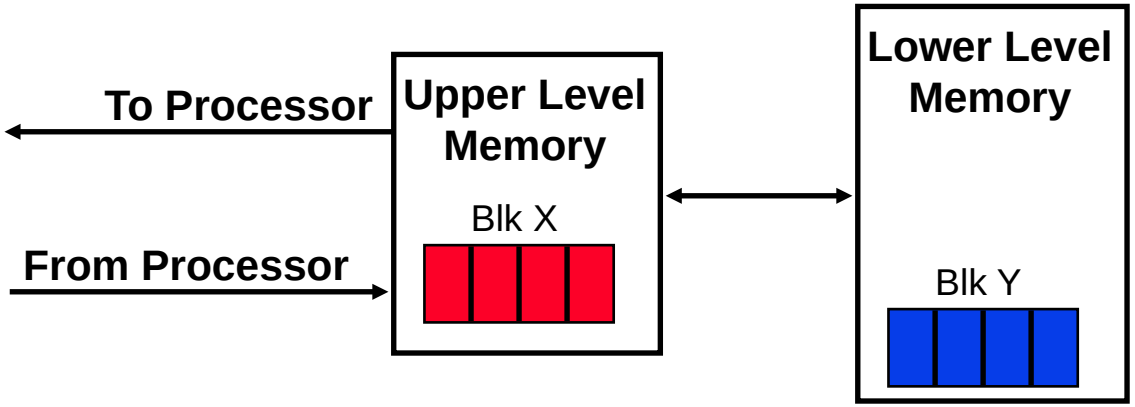
- Take advantage of the principle of locality to present the user with as much memory as is available in the cheapest technology at the speed offered by the fastest technology



The Memory Hierarchy: Why Does it Work?

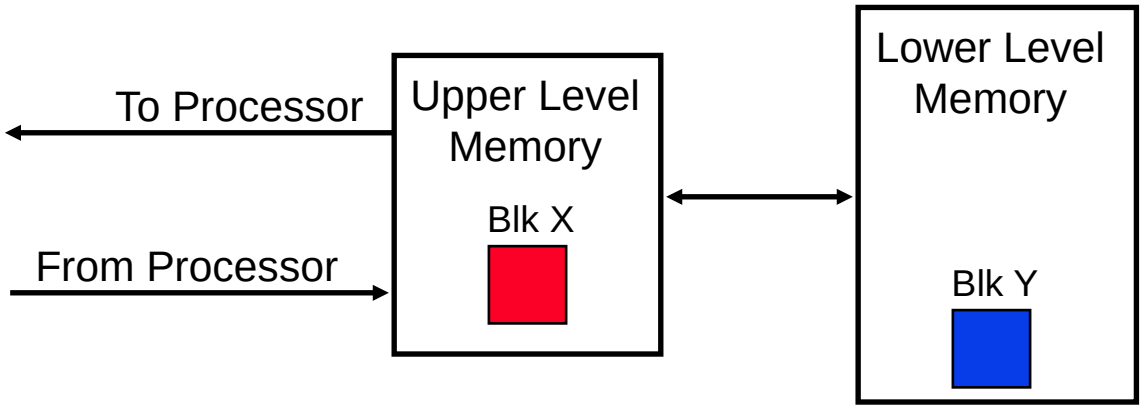
- Temporal Locality (Locality in Time):
  Keep most recently accessed instructions/data closer to the processor

- Spatial Locality (Locality in Space):
  Move blocks consisting of contiguous words to the upper levels



The Memory Hierarchy: Terminology

- Hit: data is in some block in the upper level (Blk X)
  - Hit Rate: the fraction of memory accesses found in the upper level
  - Hit Time: Time to access the upper level which consists of
    - Upper level acc. time + Time to determine hit/miss



- Miss: data is not in the upper level so needs to be retrieved from a block in the lower level (Blk Y)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level
    + Time to deliver the block the processor
  - Hit Time << Miss Penalty

How is the Hierarchy Managed?

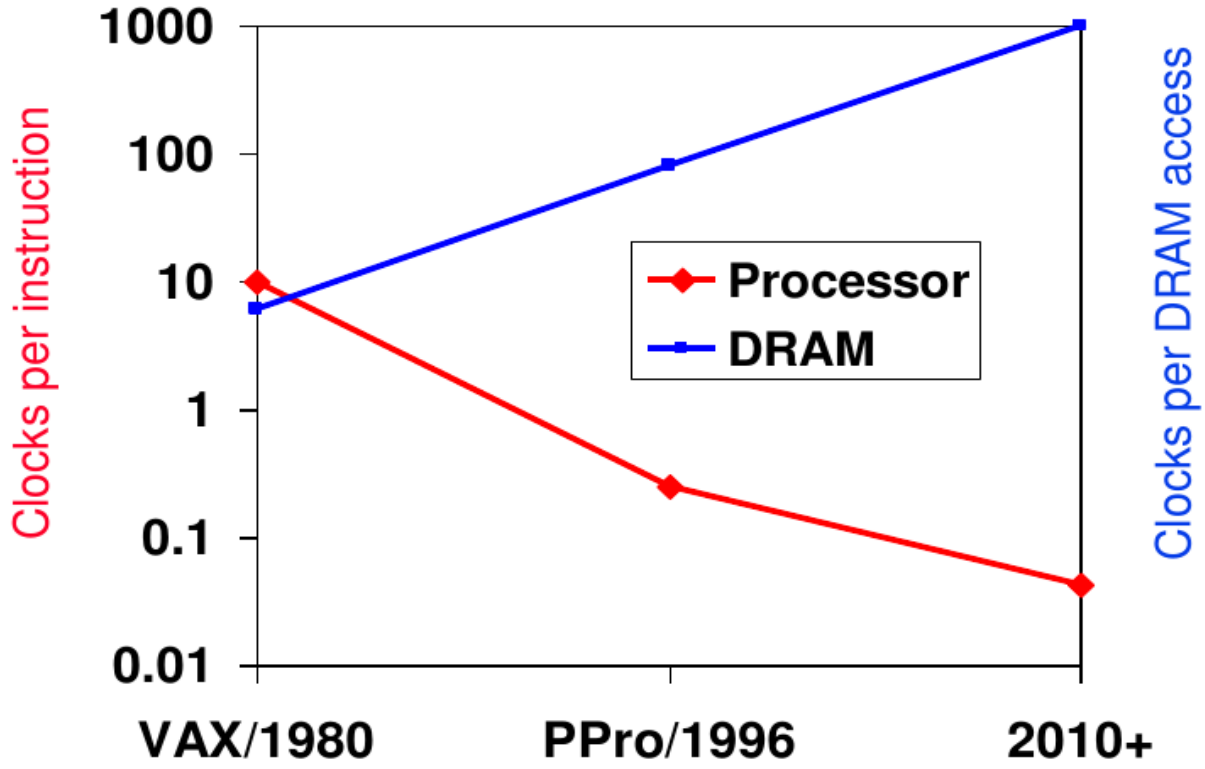
- registers ↔ memory
  - by compiler (programmer?)
- cache ↔ main memory
  - by the cache controller hardware
- main memory ↔ disks
  - by the operating system (virtual memory)
  - virtual to physical address mapping assisted by the hardware (TLB)
  - by the programmer (files)

How is the Hierarchy Managed?

- registers ↔ memory
  - by compiler (programmer?)
- cache ↔ main memory
  - by the cache controller hardware
- main memory ↔ disks
  - by the operating system (virtual memory)
  - virtual to physical address mapping assisted by the hardware (TLB)
  - by the programmer (files)

Why? The "Memory Wall"

- The Processor vs DRAM speed "gap" continues to grow



- Good cache design is increasingly important to overall performance

Cut point

The Cache

- Two questions to answer (in hardware):
  - Q1: How do we know if a data item is in the cache?
  - Q2: If it is, how do we find it?

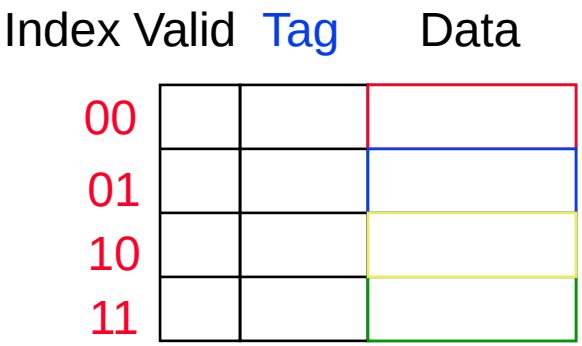
- Direct mapped
  - For each item of data at the lower level, there is exactly one location in the cache where it might be - so lots of items at the lower level must share locations in the upper level
  - Address mapping:

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

  - First consider block sizes of one word

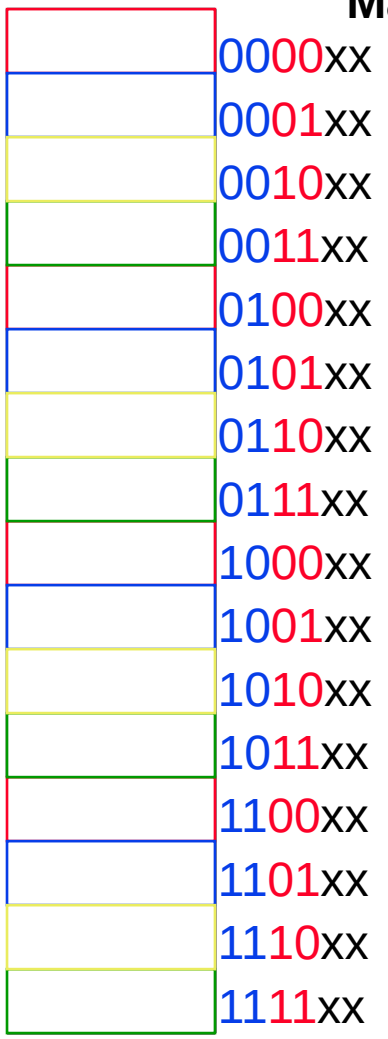
Caching: A Simple First Example

Cache



Q1: Is it there?

Main Memory

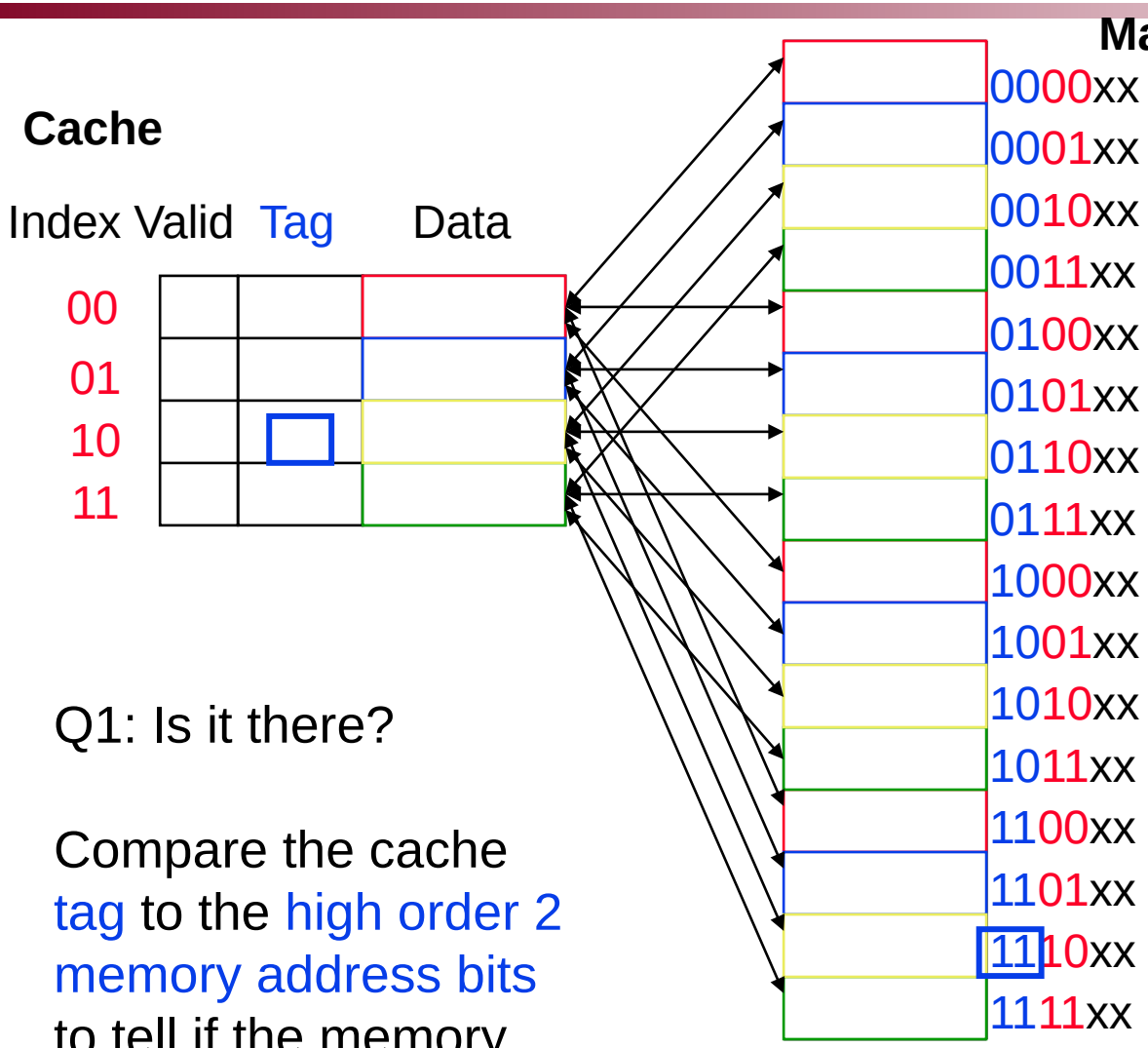


Two low order bits define the byte in the word (32-b words)

Q2: How do we find it?

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

Caching: A Simple First Example



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)

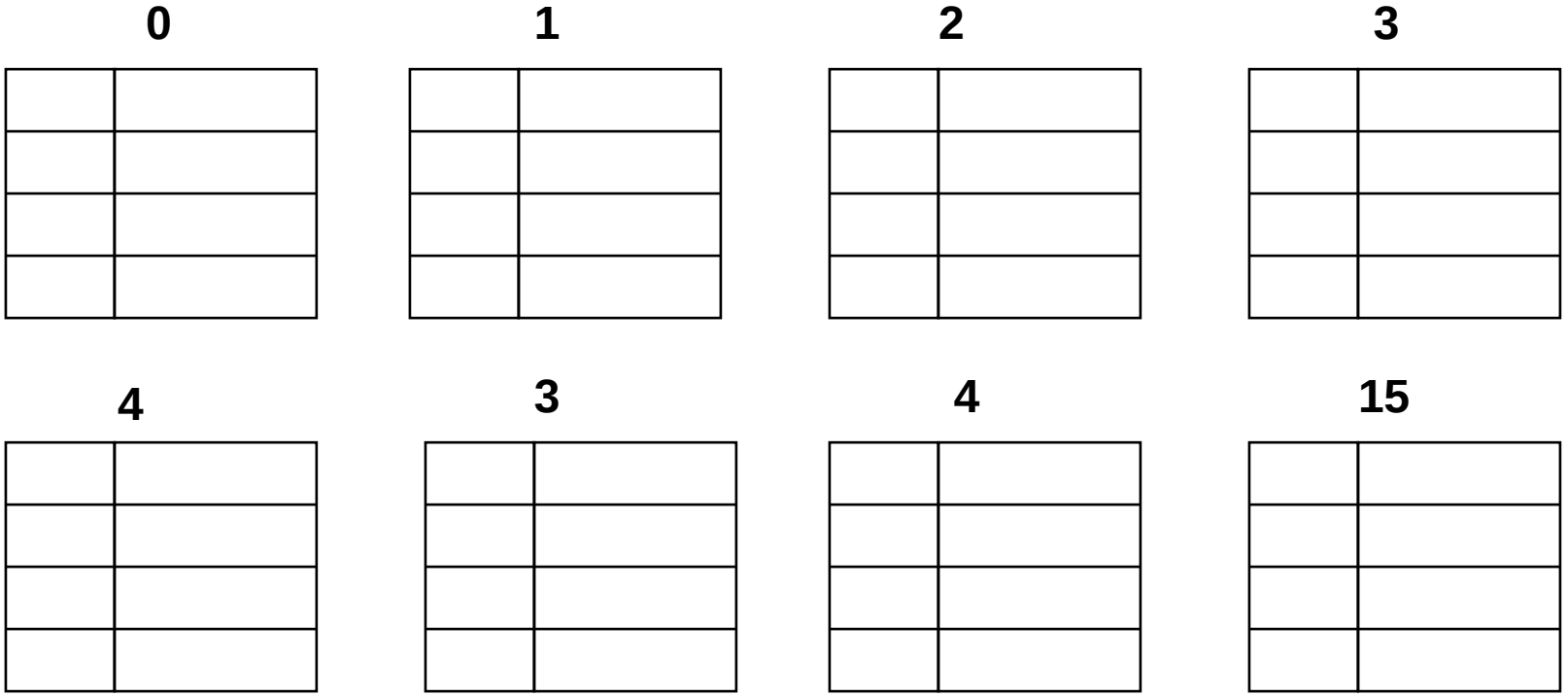
(block address) modulo (# of blocks in the cache)

Direct Mapped Cache

- Consider the main memory word reference string

0 1 2 3 4 3 4 15

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



Direct Mapped Cache

- Consider the main memory word reference string

0 1 2 3 4 3 4 15

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

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) 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)

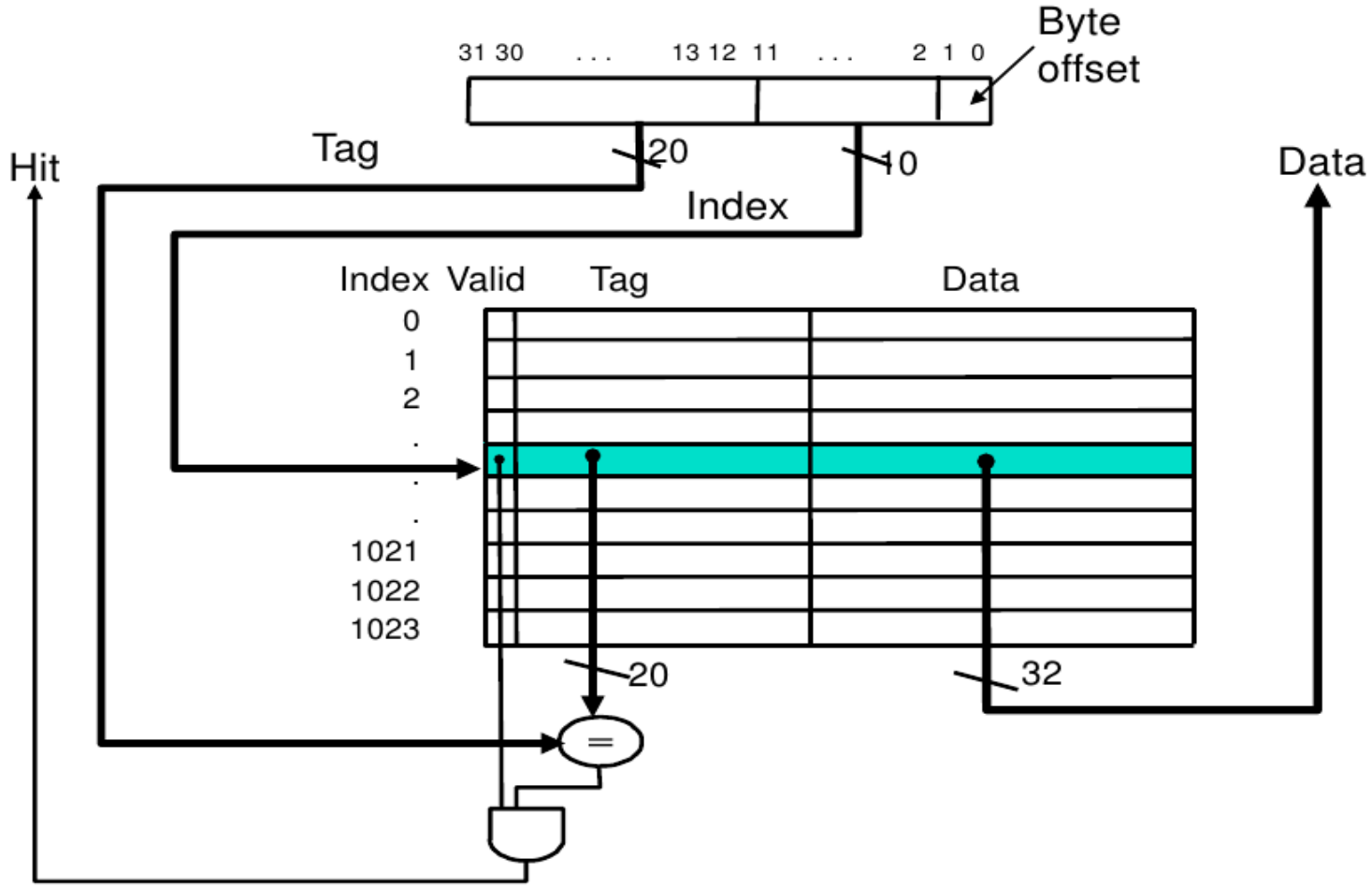
15 miss

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

- 8 requests, 6 misses

ARMv8 Direct Mapped Cache Example

- One word/block, cache size = 1K words



What kind of locality are we taking advantage of?

Cutpoint

Handling Cache Hits

- Read hits (I$ and D$)
  - this is what we want!
- Write hits (D$ only)
  - allow cache and memory to be inconsistent
    - write the data only into the cache (then write-back the cache contents to the memory when that cache block is “evicted”)
    - need a dirty bit for each cache block to tell if it needs to be written back to memory when it is evicted
  - require the cache and memory to be consistent
    - always write the data into both the cache and the memory (write-through)
    - don’t need a dirty bit
    - writes run at the speed of the main memory - slow! – or can use a write buffer, so only have to stall if the write buffer is full

Sources of Cache Misses

Compulsory (cold start or process migration, first reference):

- First access to a block, “cold” fact of life, not a whole lot you can do about it
- If you are going to run “millions” of instructions, compulsory misses are insignificant
  - Solution 1: predict future accesses (prefetching)
  - Solution 2: increase block size

Conflict (collision)

- Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity

Capacity

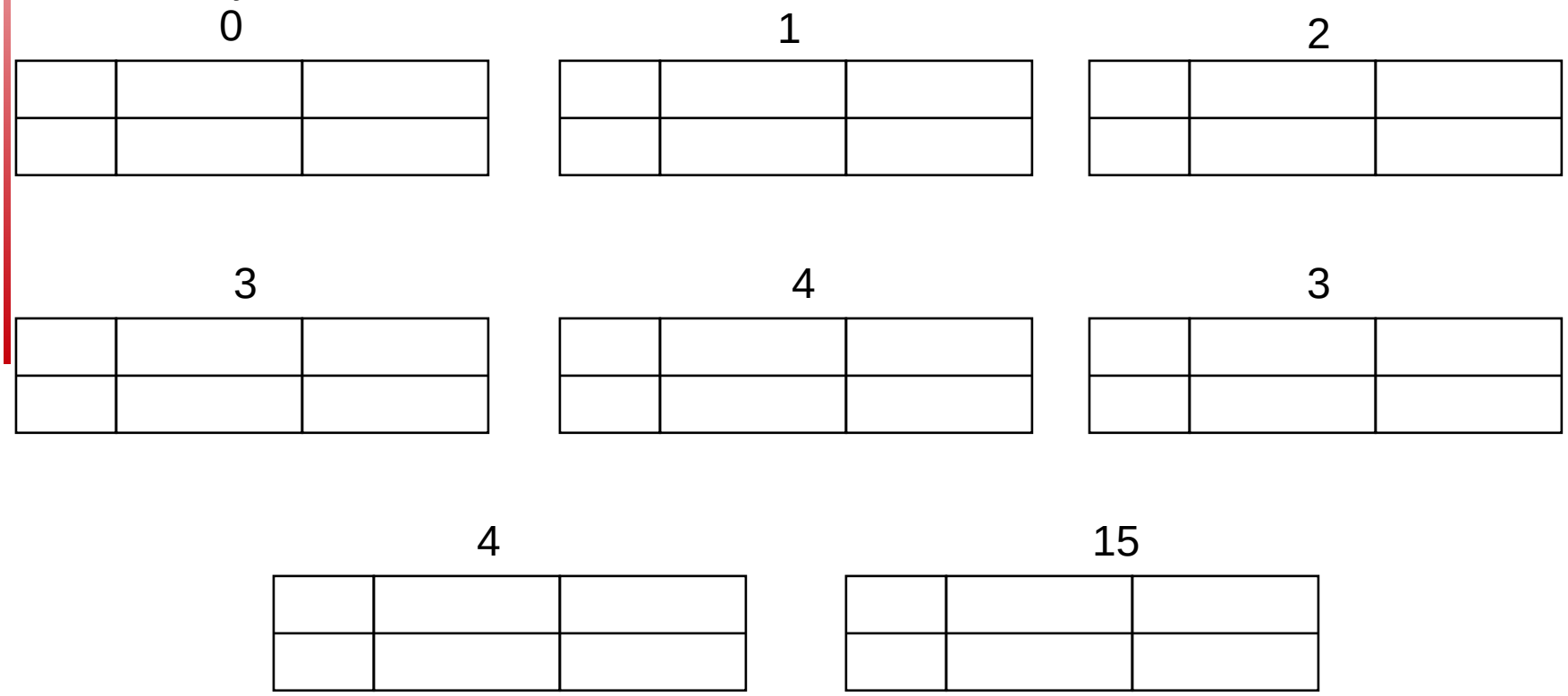
- Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size

Increasing Block Size

- Let cache block hold more than one word

0 1 2 3 4 3 4 15

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



What kind of locality are we taking advantage of?

Increasing Block Size

- Let cache block hold more than one word

0 1 2 3 4 3 4 15

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

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

~~01~~ 00	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)

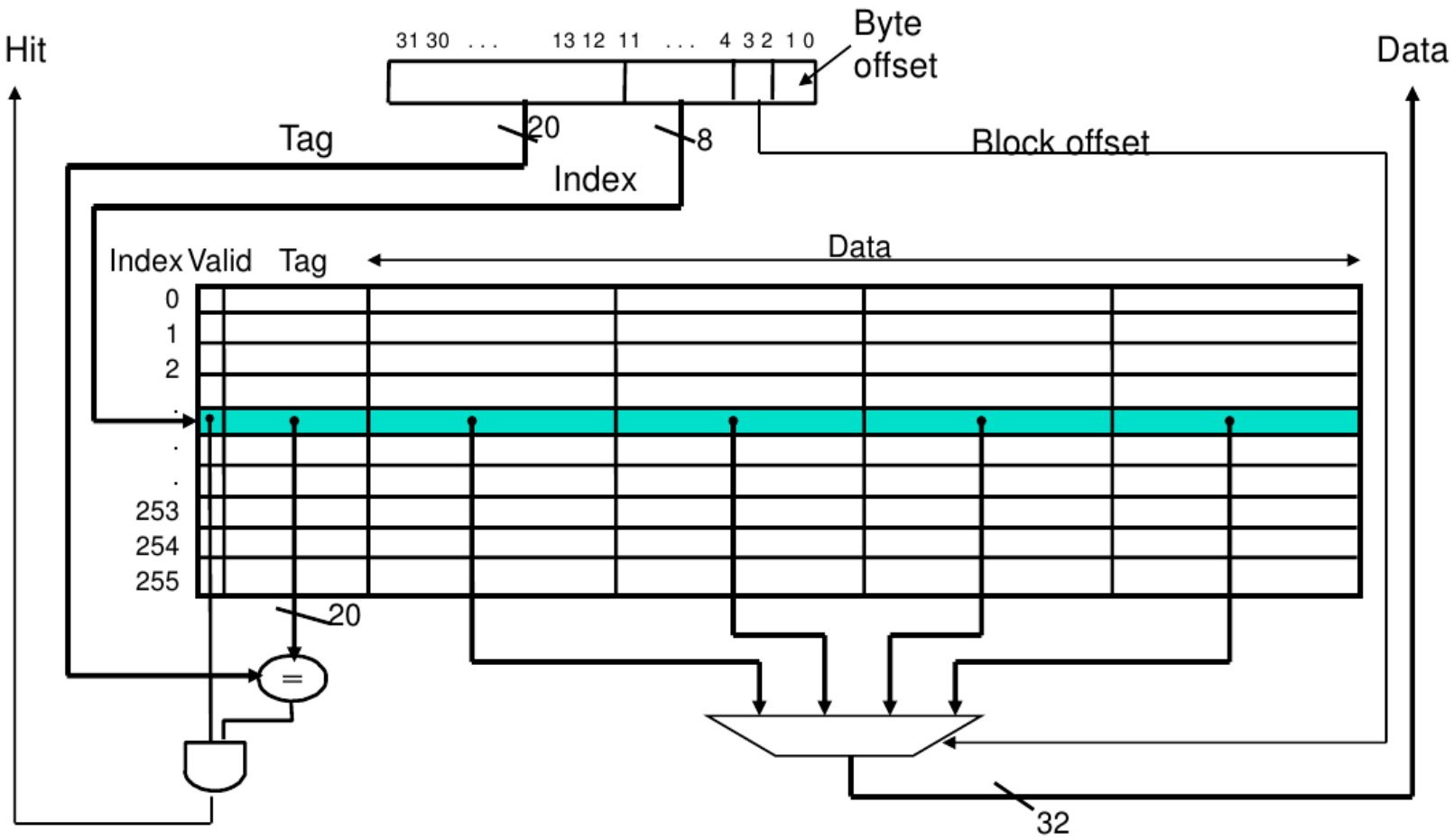
15 miss

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

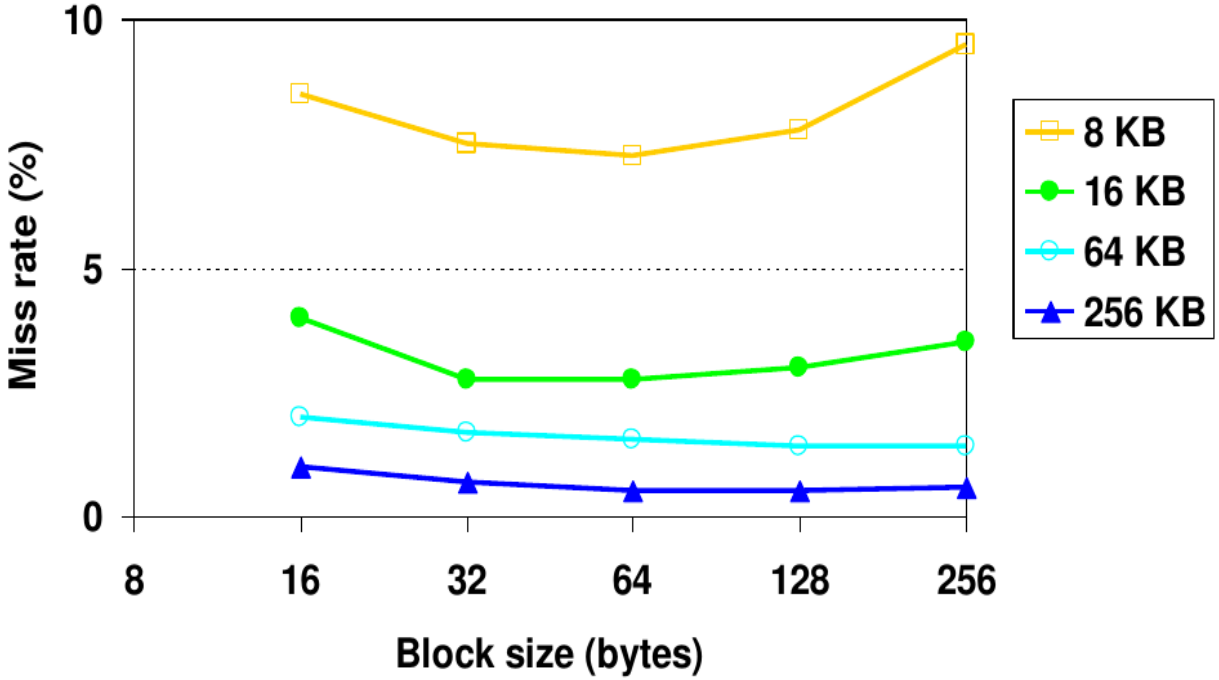
- 8 requests, 4 misses

Multiword Block Direct Mapped Cache

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



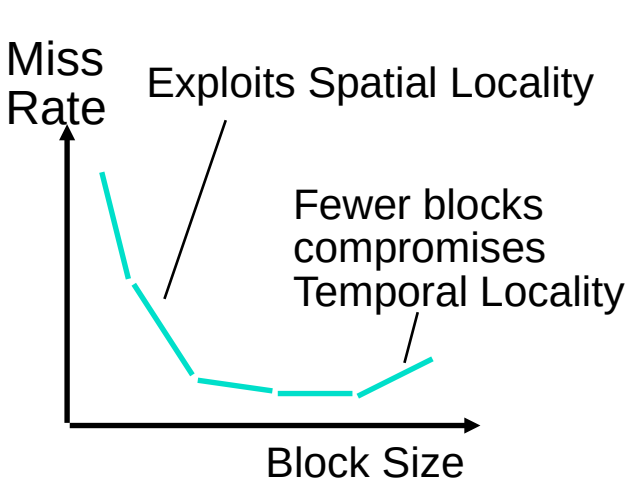
Miss Rate vs Block Size vs Cache Size

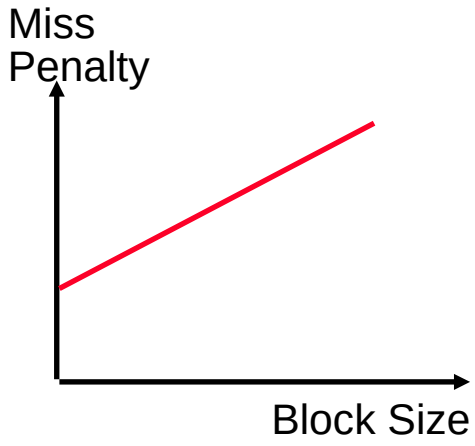


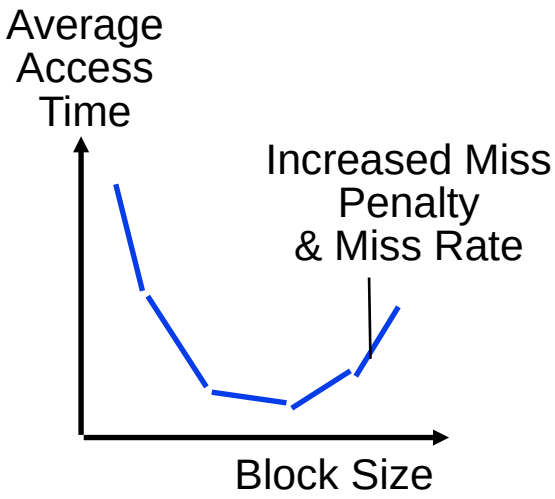
- Miss rate goes up if the block size becomes a significant fraction of the cache size because the number of blocks that can be held in the same size cache is smaller (increasing capacity misses)

Block Size Tradeoff

- Larger block sizes take advantage of spatial locality but
  - If the block size is too big relative to the cache size, the miss rate will go up
  - Larger block size means larger miss penalty
  - Latency to first word in block + transfer time for remaining words







- In general, Average Memory Access Time

  = Hit Time + Miss Penalty x Miss Rate

Cutpoint

Handling Cache Misses

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

  or (normally used in write-back caches)

  2 Write allocate – just write the word+tag into the cache (may involve having to evict a dirty block), no need to check for cache hit, no need to stall

  or (normally used in write-through caches with a write buffer)

  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

Reducing Cache Miss Rates #1

1. Allow more flexible block placement

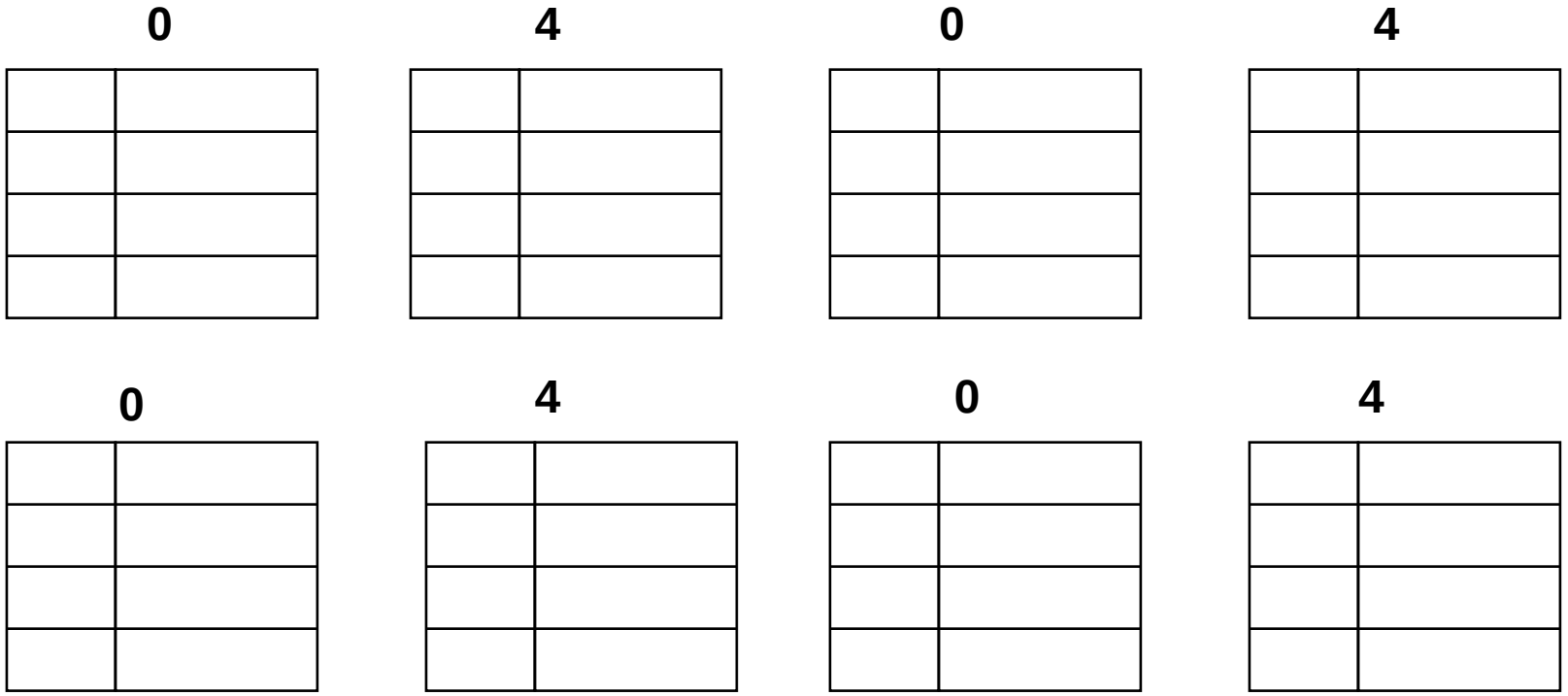
- In a direct mapped cache a memory block maps to exactly one cache block
- At the other extreme, could allow a memory block to be mapped to any cache block – fully associative cache
- A compromise is to divide the cache into sets each of which consists of n “ways” (n-way set associative). A memory block 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)

Another Reference String Mapping

- Consider the main memory word reference string

  0 4 0 4 0 4 0 4

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

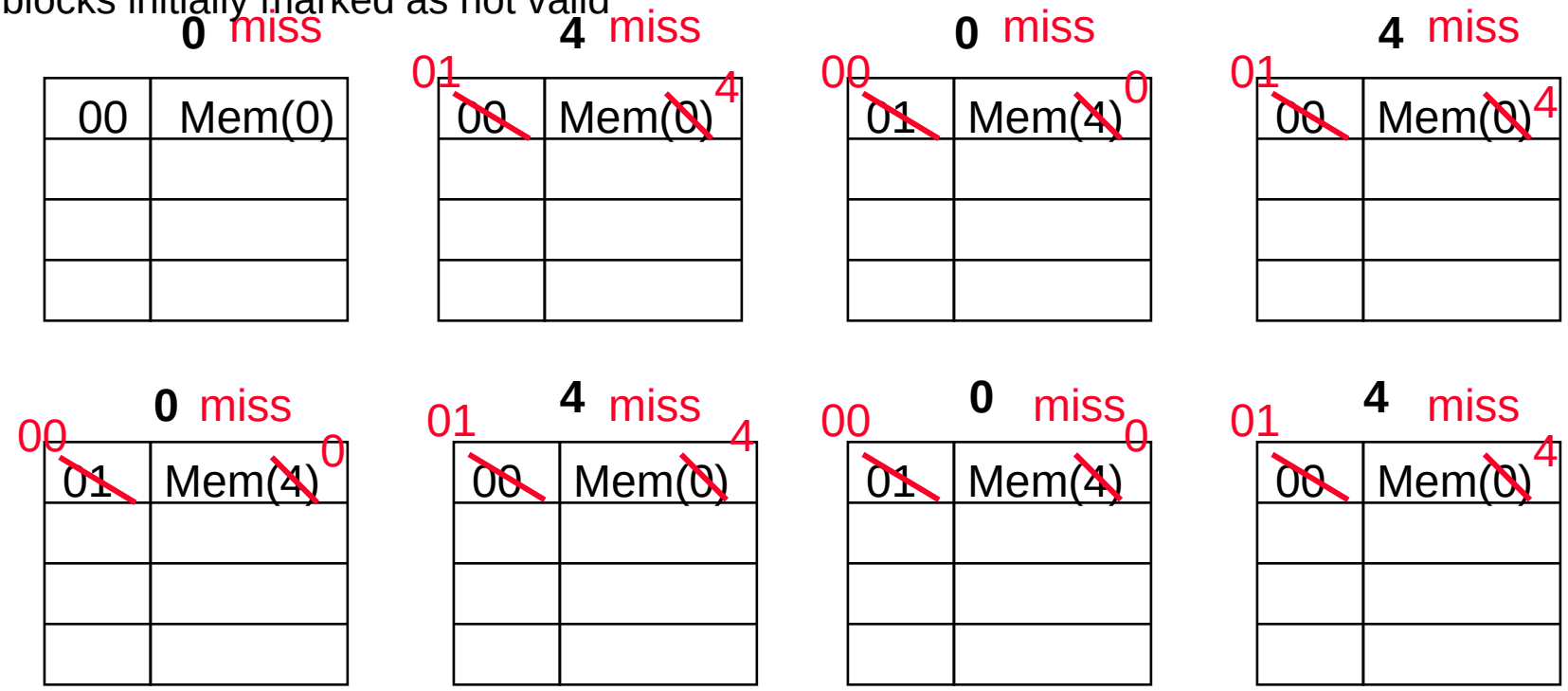


Another Reference String Mapping

- Consider the main memory word reference string

  0 4 0 4 0 4 0 4

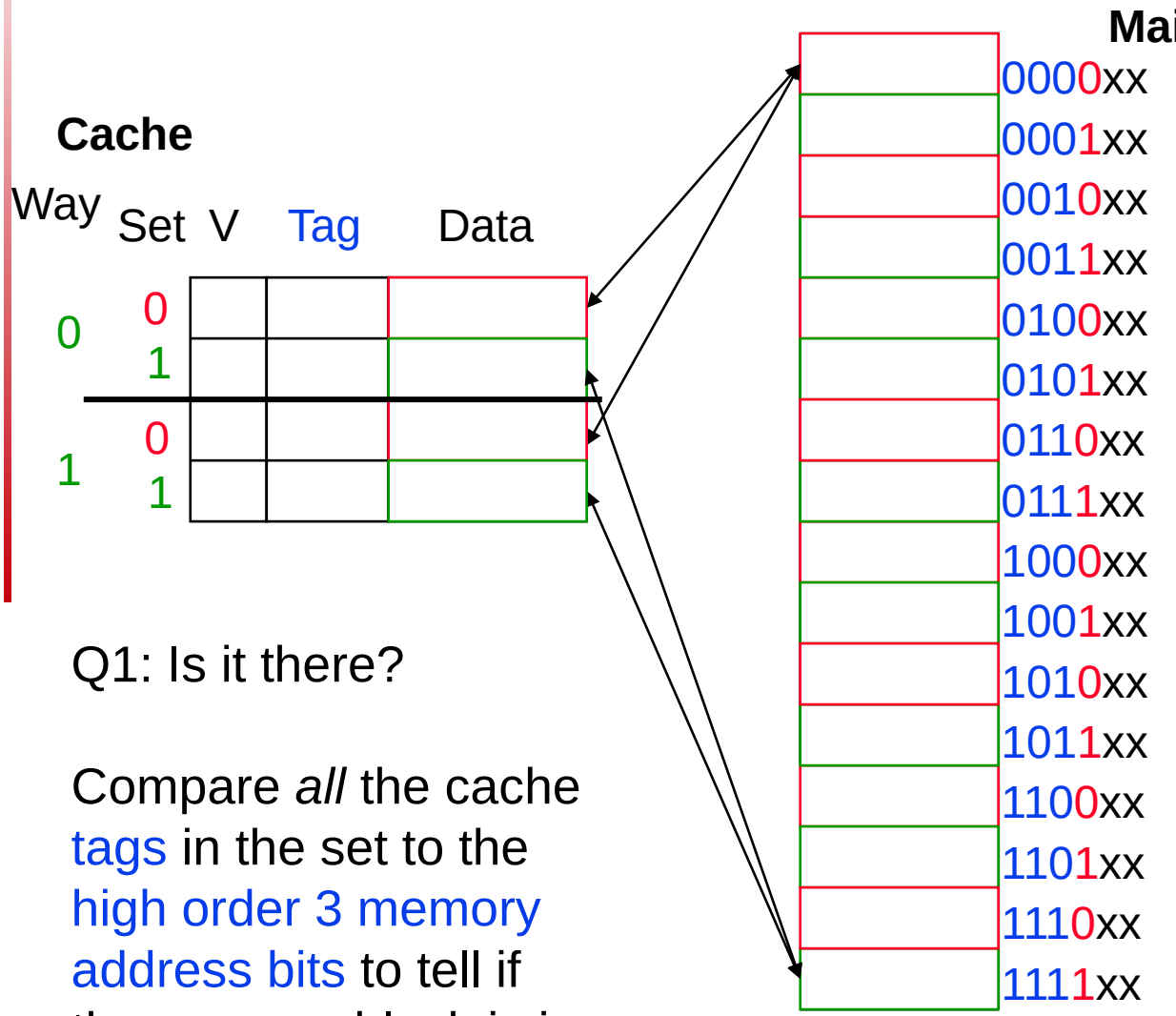
Start with an empty cache - all blocks initially marked as not valid



  - 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



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

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?

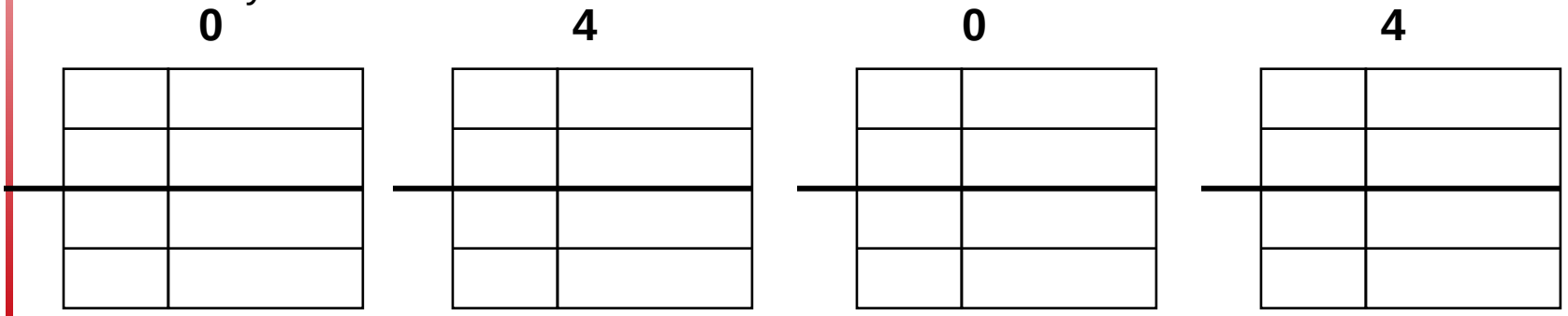
Use next 1 low order memory address bit to determine which cache set (i.e., modulo the number of sets in the cache)

Another Reference String Mapping

- Consider the main memory word reference string

0 4 0 4 0 4 0 4

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



Another Reference String Mapping

- Consider the main memory word reference string

  0 4 0 4 0 4 0 4

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

0 miss

000	Mem(0)

4 miss

000	Mem(0)
010	Mem(4)

0 hit

000	Mem(0)
010	Mem(4)

4 hit

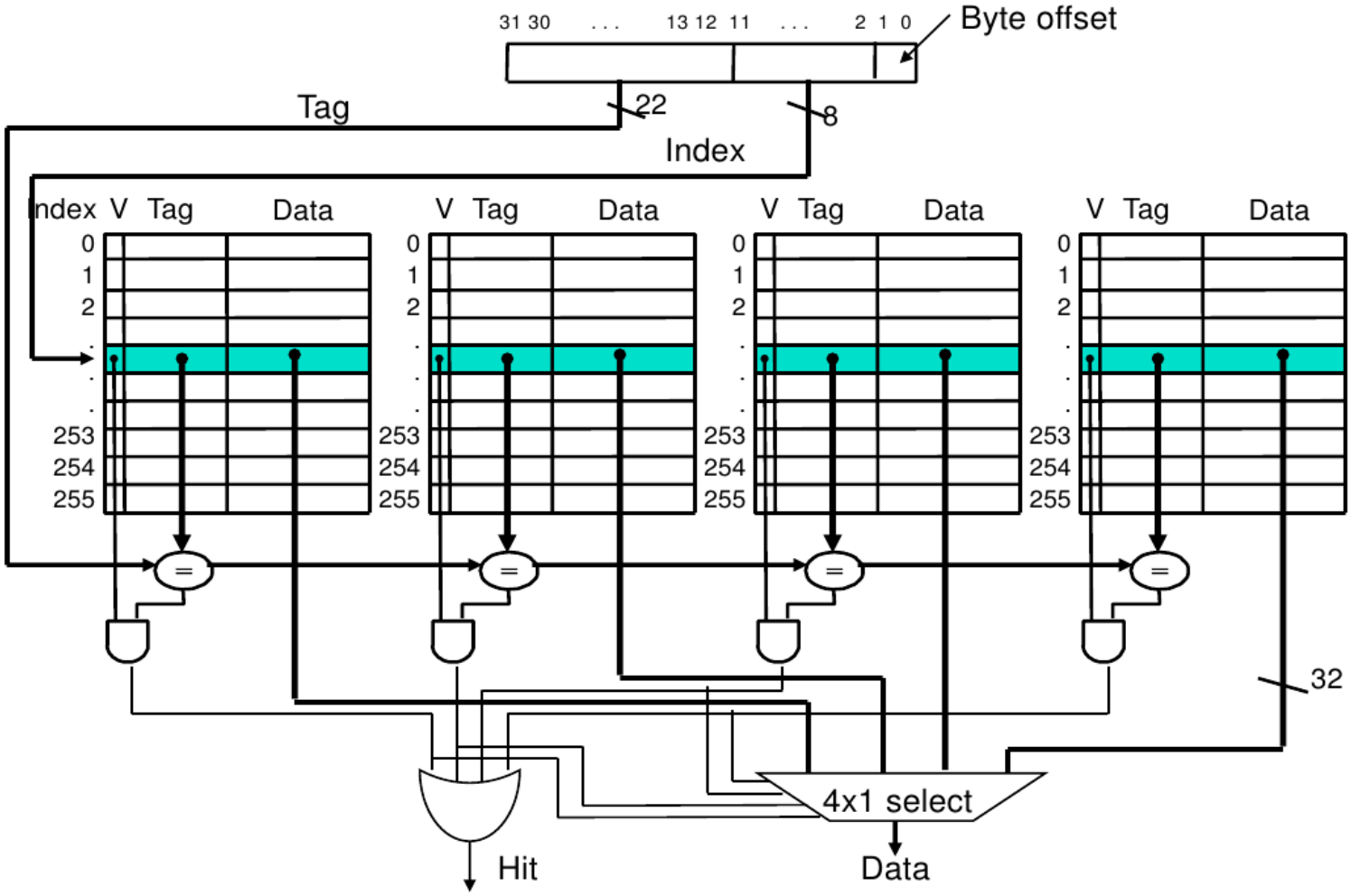
000	Mem(0)
010	Mem(4)

  - 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

- 28 = 256 sets each with four ways (each with one block)

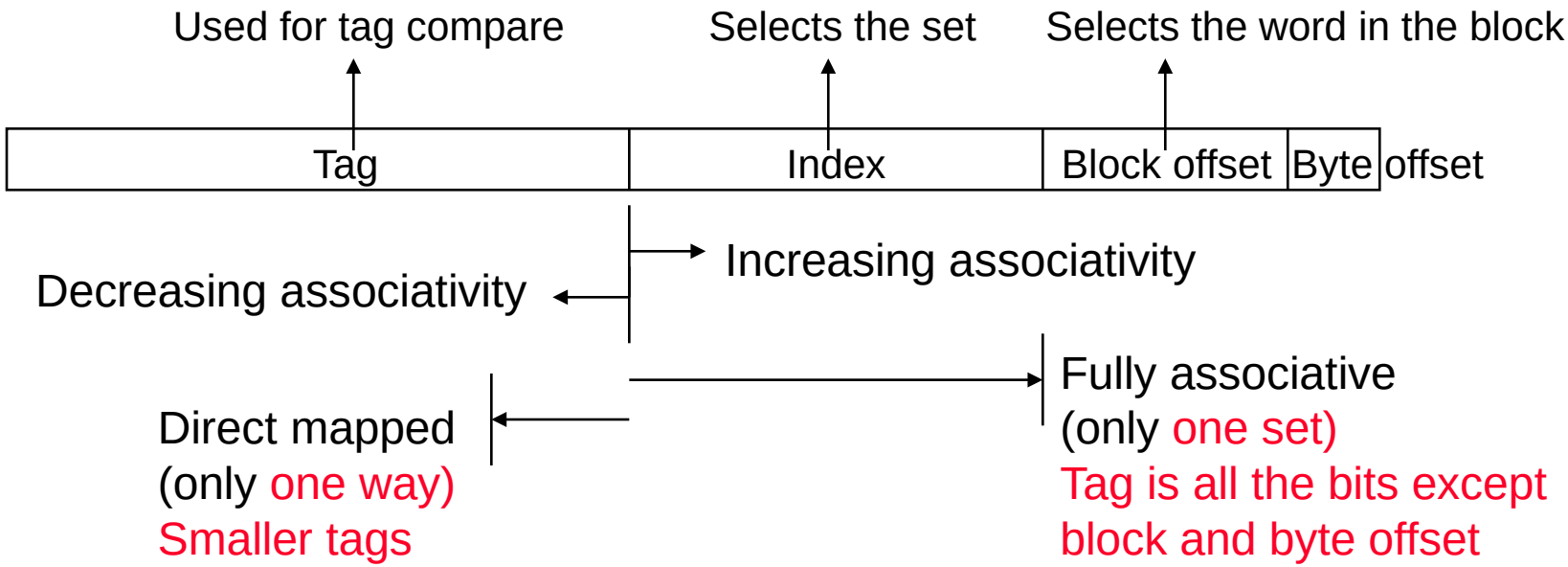


Range of Set Associative Caches

- For a fixed size cache, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number or ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit

Range of Set Associative Caches

- For a fixed size cache, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number or ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit

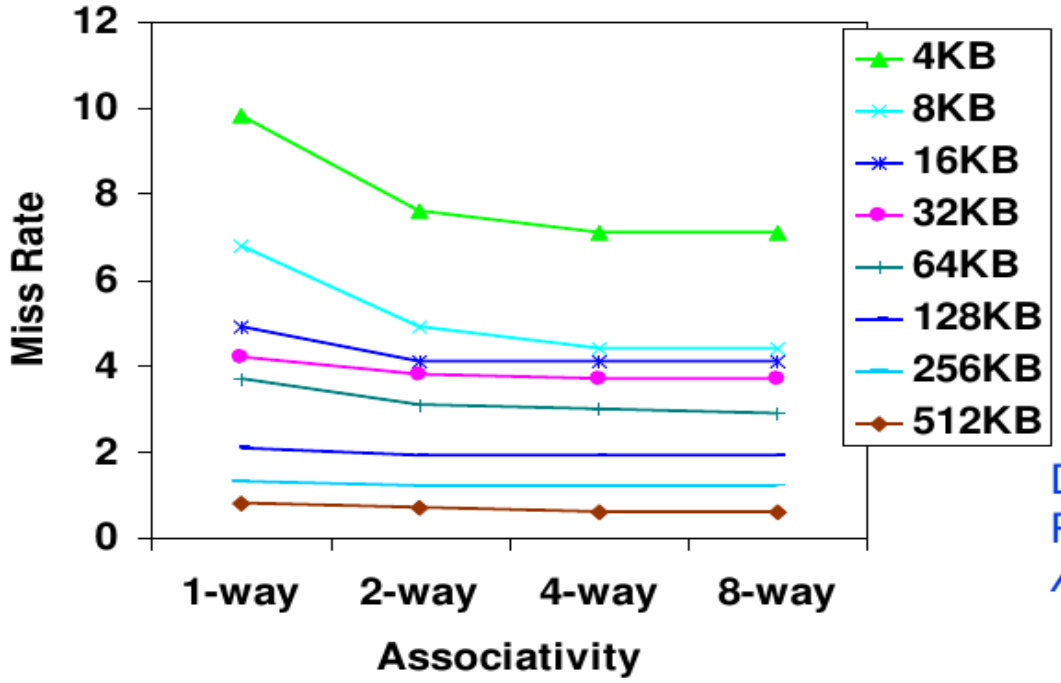


Costs of Set Associative Caches

- When a miss occurs, which way's block do we pick for replacement?
  - Least Recently Used (LRU): the block replaced is the one that has been unused for the longest time
    - Must have hardware to keep track of when each way's block was used relative to the other blocks in the set
    - For 2-way set associative, takes one bit per set → set the bit when a block is referenced (and reset the other way's bit)

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



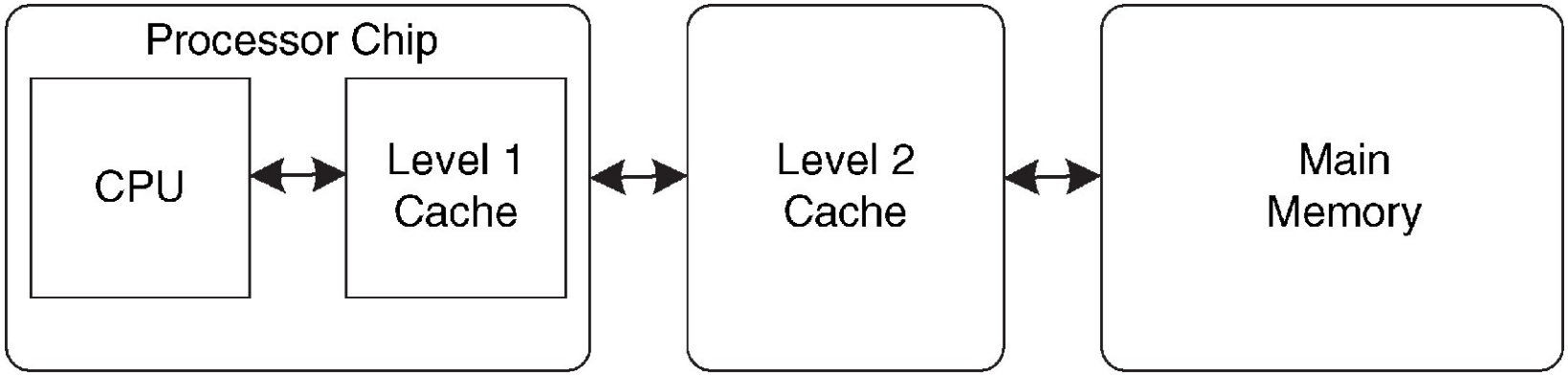
Data from Hennessy & Patterson, *Computer Architecture*, 2003

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

Cutpoint

Using multiple levels of caches

- With advancing technology have more than enough room on the die for bigger L1 caches *or* for a second level of caches – normally a unified L2 cache (i.e., it holds both instructions and data) and in some cases even a unified L3 cache



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
- The miss penalty of the L1 cache is significantly reduced by the presence of an L2 cache – so it can be smaller (i.e., faster) 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

Measuring Cache Performance

- Assuming cache hit costs are included as part of the normal CPU execution cycle, then

$$\text{CPU time} = \text{IC} \times \text{CPI} \times \text{CC}$$

$$= \text{IC} \times (\underbrace{\text{CPI}_{\text{ideal}} + \text{Memory-stall cycles}}_{\text{CPI}_{\text{stall}}}) \times \text{CC}$$

- Memory-stall cycles come from cache misses (a sum of read-stalls and write-stalls)

$$\text{Read-stall cycles} = \text{reads/program} \times \text{read miss rate} \times \text{read miss penalty}$$

$$\text{Write-stall cycles} = (\text{writes/program} \times \text{write miss rate} \times \text{write miss penalty}) + \text{write buffer stalls}$$

- For write-through caches, we can simplify this to

$$\text{Memory-stall cycles} = \text{miss rate} \times \text{miss penalty}$$

Using multiple levels of caches

- Average Memory Access Time (AMAT) = Hit TimeL1 + Miss RateL1 x Miss PenaltyL1
  - Miss PenaltyL1 = Hit TimeL2 + Miss RateL2 x Miss PenaltyL2
  - Average Memory Access Time = Hit TimeL1 + Miss RateL1 x (Hit TimeL2 + Miss RateL2 x Miss PenaltyL2)
- For our example, CPIideal of 2, 100 cycle miss penalty (to main memory), 36% load/stores, a 2% (4%) L1 I$ (D$) miss rate, add a L2$ that has a 25 cycle miss penalty and 15% miss rate.
  - CPIstalls = 2 + .02×(25 + .15×100) + .36×.04×(25 +.15×100)
    = 2 + .8 + .576 = 3.376
    (as compared to 5.44 with no L2$)
  - Note: CPIideal includes the hit time of the L1 for a pipelined processor.

Key Cache Design Parameters

	L1 typical	L2 typical
Total size (blocks)	250 to 2000	4000 to 250,000
Total size (KB)	16 to 64	500 to 8000
Block size (B)	32 to 64	32 to 128
Miss penalty (clocks)	10 to 25	100 to 1000
Miss rates (global for L2)	2% to 5%	0.1% to 2%

Two Machines' Cache Parameters

	Intel P4	AMD Opteron
L1 organization	Split I$ and D$	Split I$ and D$
L1 cache size	8KB for D$, 96KB for trace cache (~I$)	64KB for each of I$ and D$
L1 block size	64 bytes	64 bytes
L1 associativity	4-way set assoc.	2-way set assoc.
L1 replacement	~ LRU	LRU
L1 write policy	write-through	write-back
L2 organization	Unified	Unified
L2 cache size	512KB	1024KB (1MB)
L2 block size	128 bytes	64 bytes
L2 associativity	8-way set assoc.	16-way set assoc.
L2 replacement	~LRU	~LRU
L2 write policy	write-back	write-back

Cutpoint

4 Questions for the Memory Hierarchy

- Q1: Where can a block be placed in the upper level? *(Block placement)*
- Q2: How is a block found if it is in the upper level? *(Block identification)*
- Q3: Which block should be replaced on a miss? *(Block replacement)*
- Q4: What happens on a write? *(Write strategy)*

Q1&Q2: Where can a block be placed/found?

	# of sets	Blocks per set
Direct mapped	# of blocks in cache	1
Set associative	(# of blocks in cache)/ associativity	Associativity (typically 2 to 16)
Fully associative	1	# of blocks in cache

	Location method	# of comparisons
Direct mapped	Index	1
Set associative	Index the set; compare set's tags	Degree of associativity
Fully associative	Compare all blocks tags	# of blocks

Q3: Which block should be replaced on a miss?

- Easy for direct mapped – only one choice
- Set associative or fully associative
  - Random
  - LRU (Least Recently Used)

- For a 2-way set associative cache, random replacement has a miss rate about 1.1 times higher than LRU.
- LRU is too costly to implement for high levels of associativity (> 4-way) since tracking the usage information is costly

Q4: What happens on a write?

- *Write-through* – The information is written to both the block in the cache and to the block in the next lower level of the memory hierarchy
  - Write-through is always combined with a write buffer so write waits to lower level memory can be eliminated (as long as the write buffer doesn't fill)
- *Write-back* – The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - Need a dirty bit to keep track of whether the block is clean or dirty
- Pros and cons of each?
  - Write-through: read misses don't result in writes (so are simpler and cheaper)
  - Write-back: repeated writes require only one write to lower level

Improving Cache Performance

0. Reduce the time to hit in the cache
   - smaller cache
   - direct mapped cache
   - smaller blocks
   - for writes
     - no write allocate – no “hit” on cache, just write to write buffer
     - write allocate – to avoid two cycles (first check for hit, then write) pipeline writes via a delayed write buffer to cache

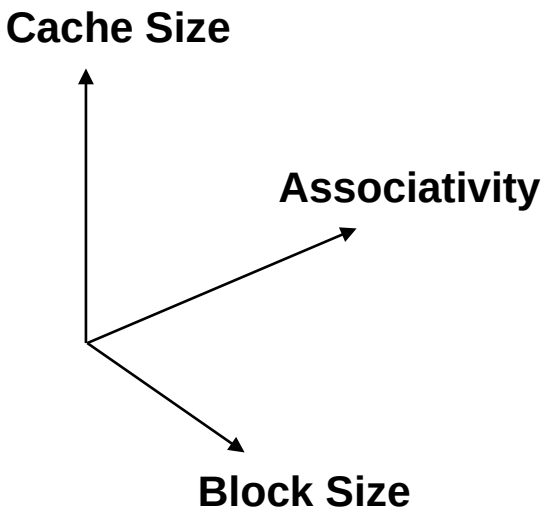
1. Reduce the miss rate
   - bigger cache
   - more flexible placement (increase associativity)
   - larger blocks (16 to 64 bytes typical)
   - victim cache – small buffer holding most recently discarded blocks

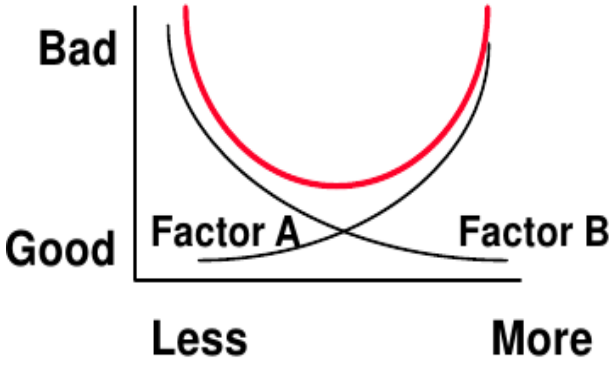
Improving Cache Performance

2. Reduce the miss penalty
   - smaller blocks
   - use a write buffer to hold dirty blocks being replaced so don’t have to wait for the write to complete before reading
   - check write buffer (and/or victim cache) on read miss – may get lucky
   - for large blocks fetch critical word first
   - use multiple cache levels – L2 cache not tied to CPU clock rate
   - faster backing store/improved memory bandwidth
     - wider buses
     - memory interleaving, page mode DRAMs

Summary: The Cache Design Space

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back
  - write allocation
- The optimal choice is a compromise
  - depends on access characteristics
    - workload
    - use (I-cache, D-cache, TLB)
  - depends on technology / cost
- Simplicity often wins





Other Ways to Reduce Cache Miss Rates

1. Allow more flexible block placement
   - In a direct mapped cache a memory block maps to exactly one cache block
   - At the other extreme, could allow a memory block to be mapped to any cache block – fully associative cache
   - A compromise is to divide the cache into sets each of which consists of n “ways” (n-way set associative)
2. Use multiple levels of caches
   - Add a second level of caches on chip – normally a unified L2 cache (i.e., it holds both instructions and data)
     - L1 caches focuses on minimizing hit time in support of a shorter clock cycle (smaller with smaller block sizes)
     - L2 cache focuses on reducing miss rate to reduce the penalty of long main memory access times (larger with larger block sizes)

Cache Summary

- The Principle of Locality:
  - Program likely to access a relatively small portion of the address space at any instant of time
  - Temporal Locality: Locality in Time
  - Spatial Locality: Locality in Space
- Three major categories of cache misses:
  - Compulsory misses: sad facts of life, e.g., cold start misses
  - Conflict misses: increase cache size and/or associativity
    Nightmare Scenario: ping pong effect!
  - Capacity misses: increase cache size
- Cache design space
  - total size, block size, associativity (replacement policy)
  - write-hit policy (write-through, write-back)
  - write-miss policy (write allocate, write buffers)

Improving Cache Performance

Reduce the hit time

- smaller cache
- direct mapped cache
- smaller blocks
- for writes
  - no write allocate – just write to write buffer
  - write allocate – write to a delayed write buffer that then writes to the cache

Reduce the miss penalty

- smaller blocks
- for large blocks fetch critical word first
- use a write buffer
- check write buffer (and/or victim cache) on read miss – may get lucky
- use multiple cache levels – L2 cache not tied to CPU clock rate
- faster backing store/improved memory bandwidth
  - wider buses
  - SDRAMs

Reduce the miss rate

- bigger cache
- associative cache
- larger blocks (16 to 64 bytes)
- use a victim cache – a small buffer that holds the most recently discarded blocks

Cutpoint

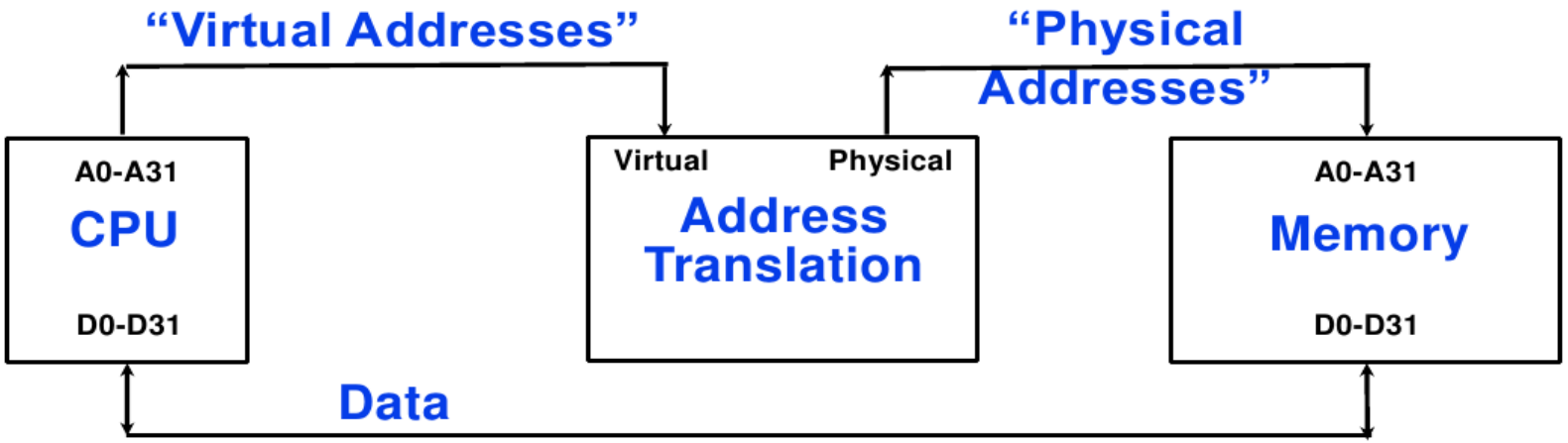
Virtual Memory

- The program's view of memory is not the actual physical DRAM
  - All complier/linkers place program TXT, global, heap and stack in the same place
  - Physical DRAM memory often smaller than program visible space!
- Use main memory as a “cache” for secondary memory
  - Allows efficient and safe sharing of memory among multiple programs
  - Provides the ability to easily run programs larger than the size of physical memory
  - Simplifies loading a program for execution by providing for code relocation (i.e., the code can be loaded anywhere in main memory)

Virtual Memory (cont)

- What makes it work? – again the Principle of Locality
  - A program is likely to access a relatively small portion of its address space during any period of time
- Each program is compiled into its own address space – a “virtual” address space
  - During run-time each virtual address must be translated to a physical address (an address in main memory)

Solution: Add a Layer of Indirection

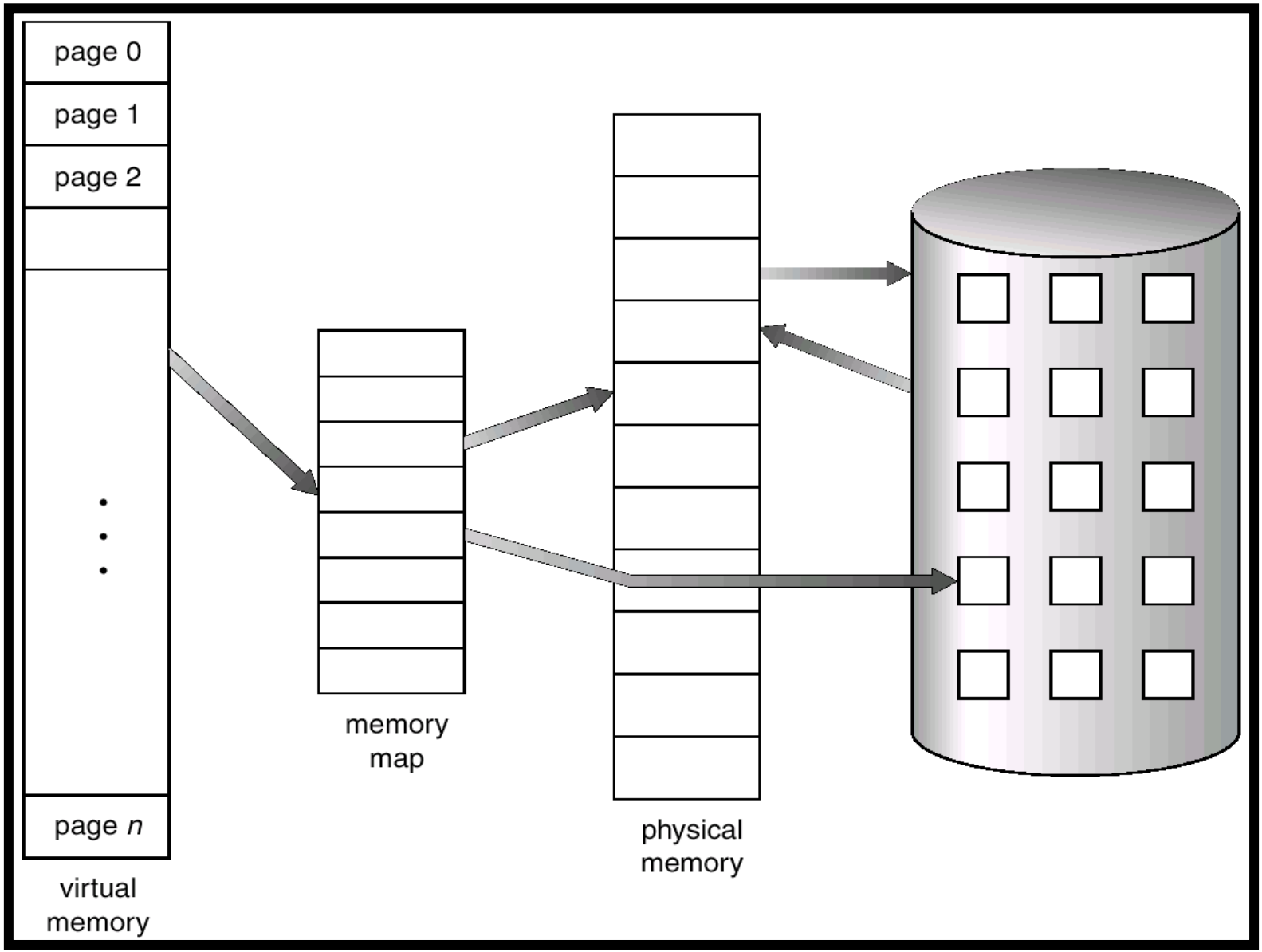


User programs run in an standardized virtual address space

Address Translation hardware managed by the operating system (OS) maps virtual address to physical memory

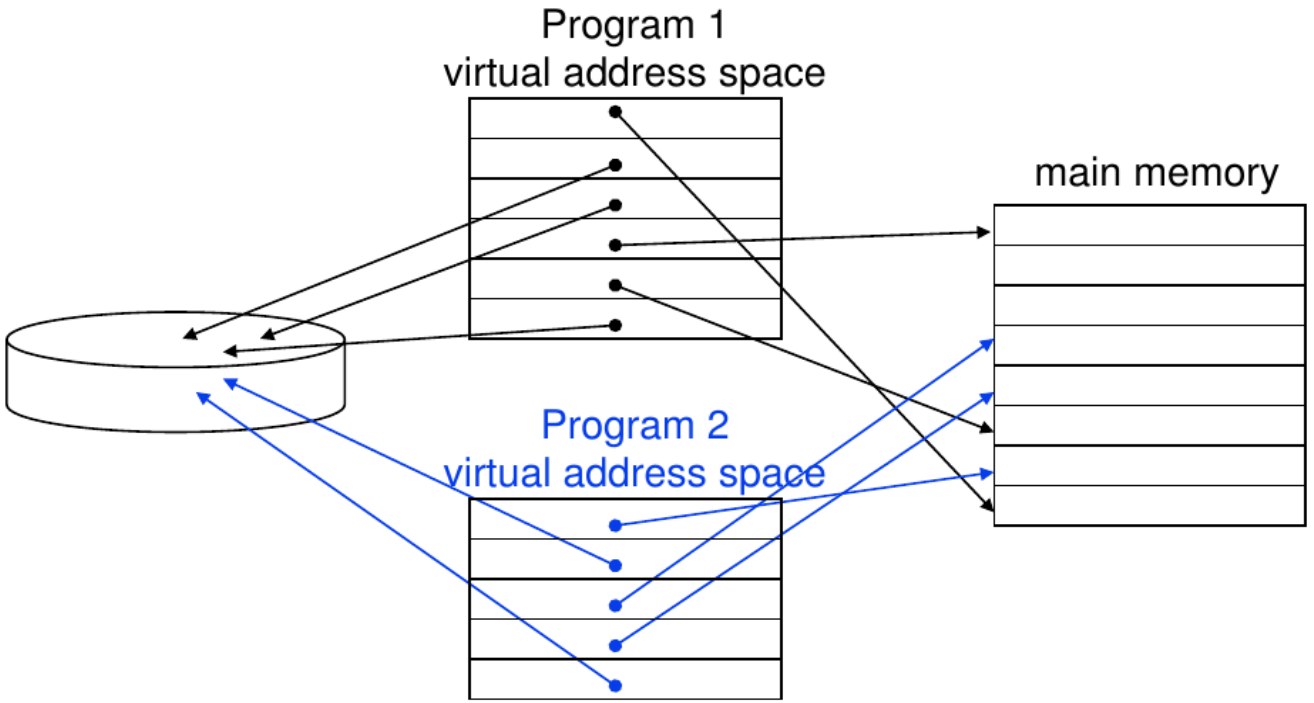
Hardware supports "modern" OS features: Protection, Translation, Sharing

Virtual Memory That is Larger Than Physical Memory



Two Programs Sharing Physical Memory

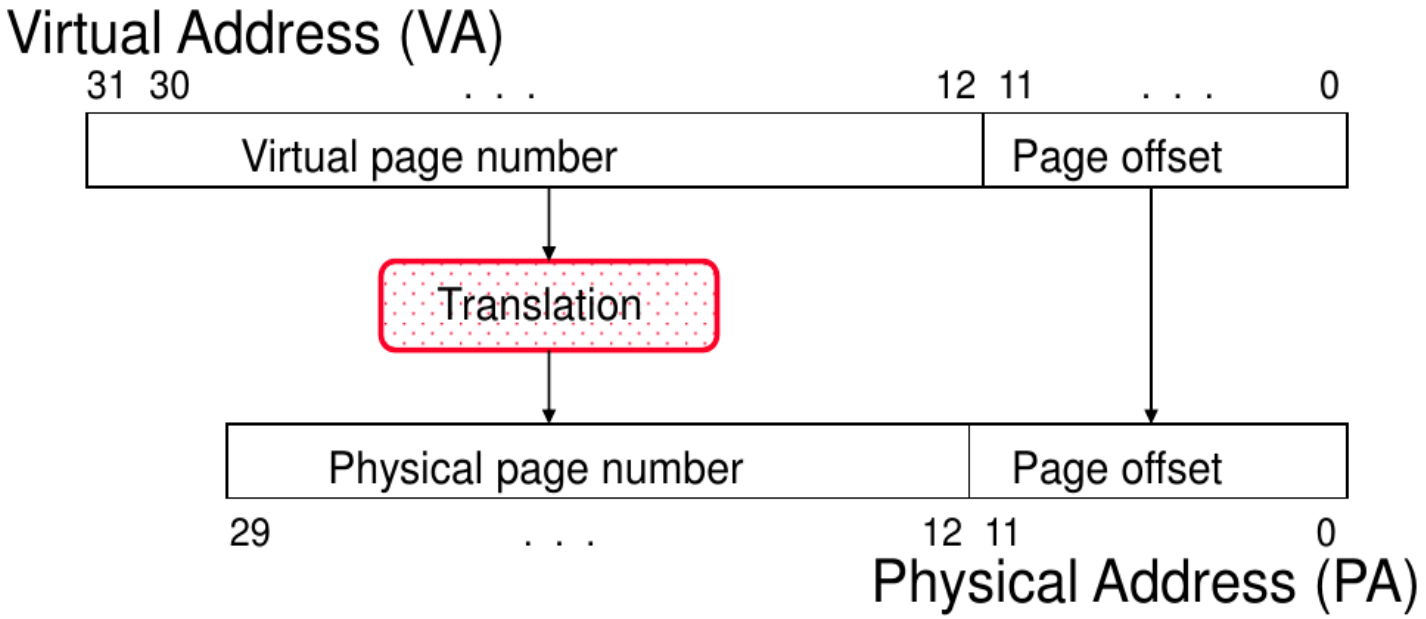
- A program's address space is divided into pages (all one fixed size) or segments (variable sizes)
  - The starting location of each page (either in main memory or in secondary memory) is contained in the program's page table



Cutpoint

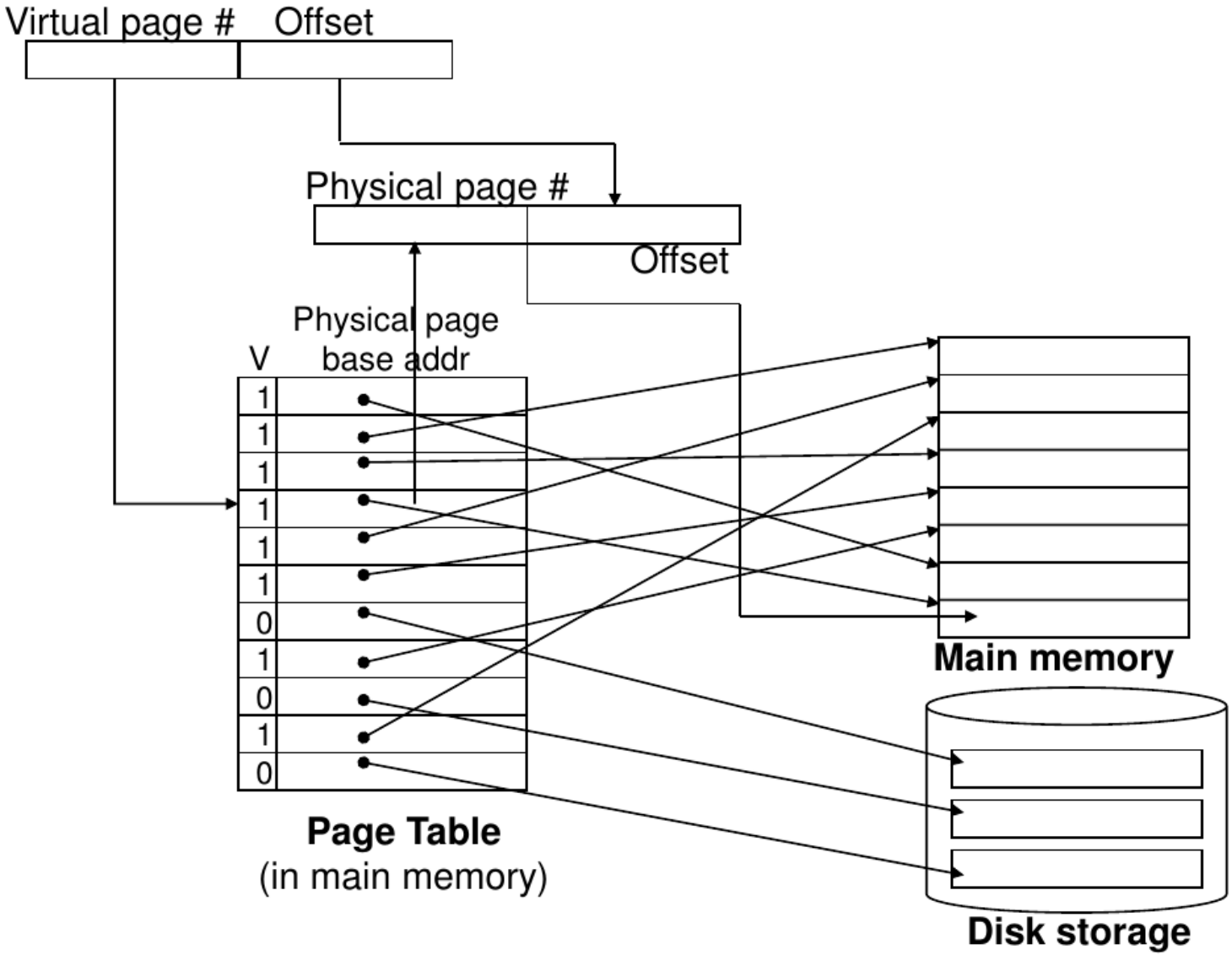
Address Translation

- A virtual address is translated to a physical address by a combination of hardware and software

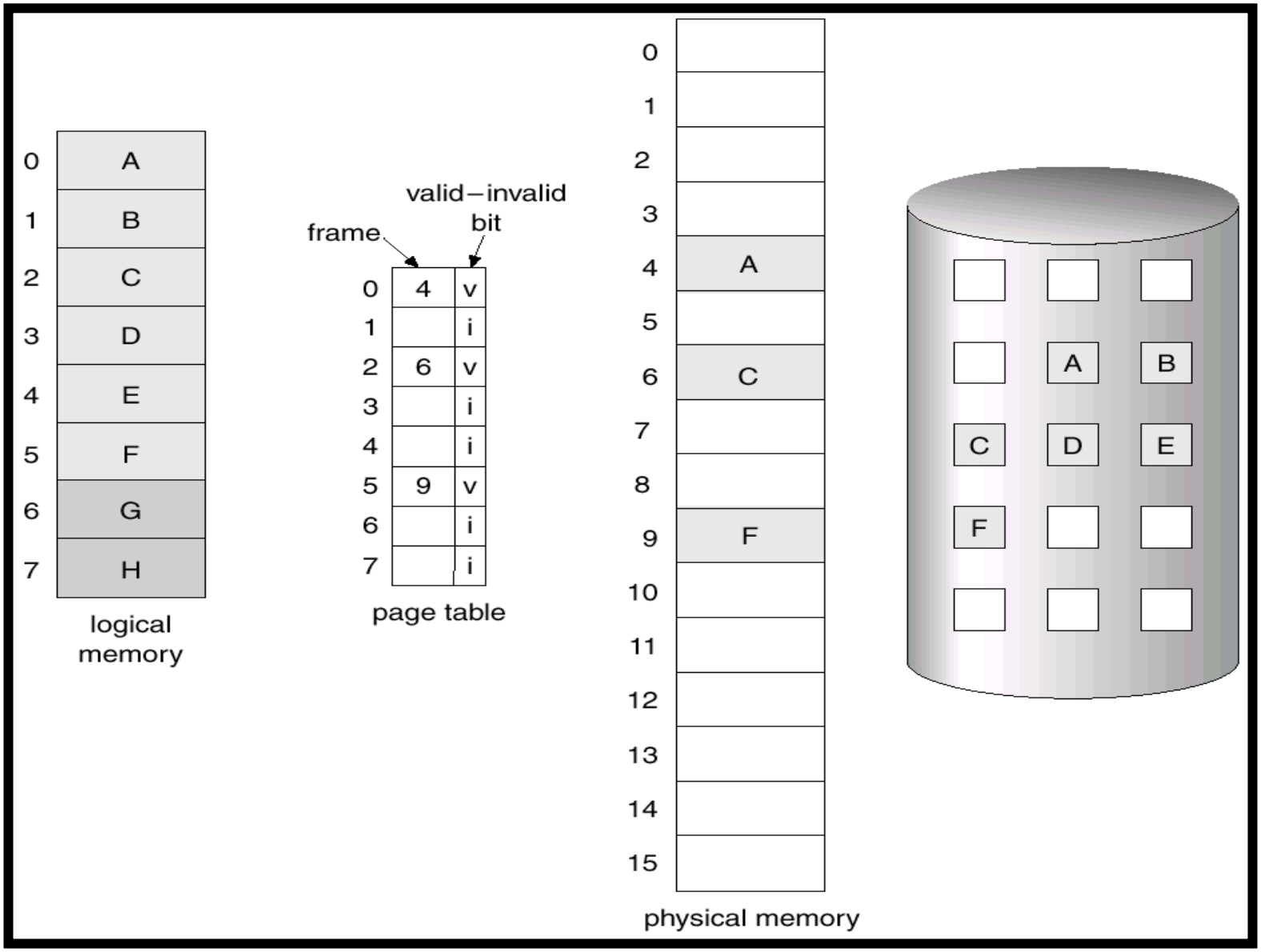


- So each memory request *first* requires an address translation from the virtual space to the physical space
  - A virtual memory miss (i.e., when the page is not in physical memory) is called a page fault

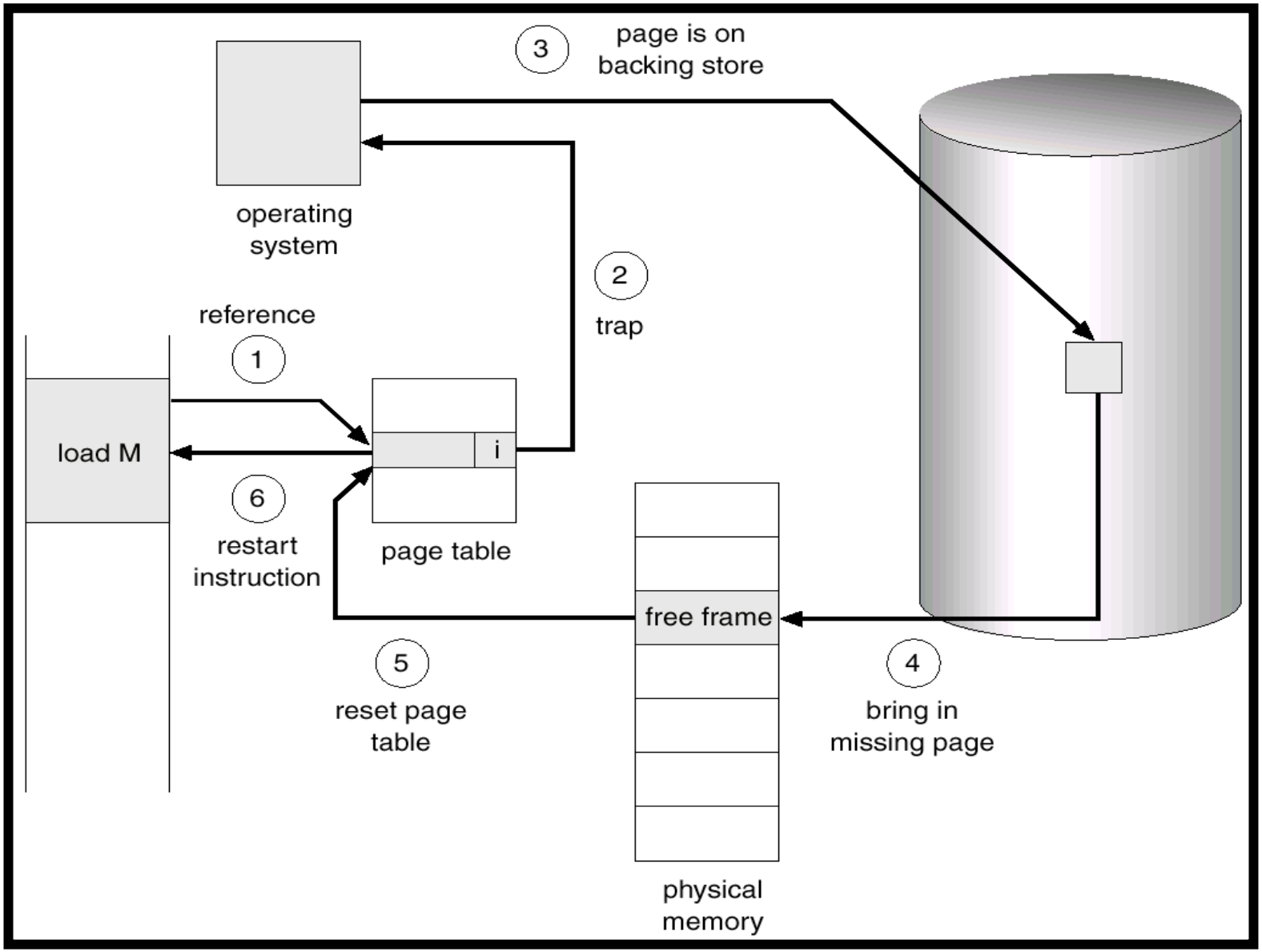
Address Translation Mechanisms



Page Table When Some Pages Are Not in Main Memory



Steps in Handling a Page Fault



Cutpoint

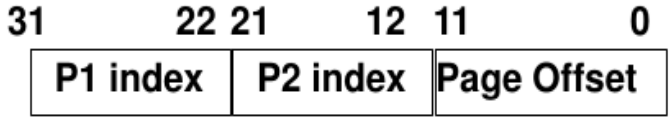
Page tables may not fit in memory!

A table for 4KB pages for a 32-bit address space has 1M entries

Each process needs its own address space!

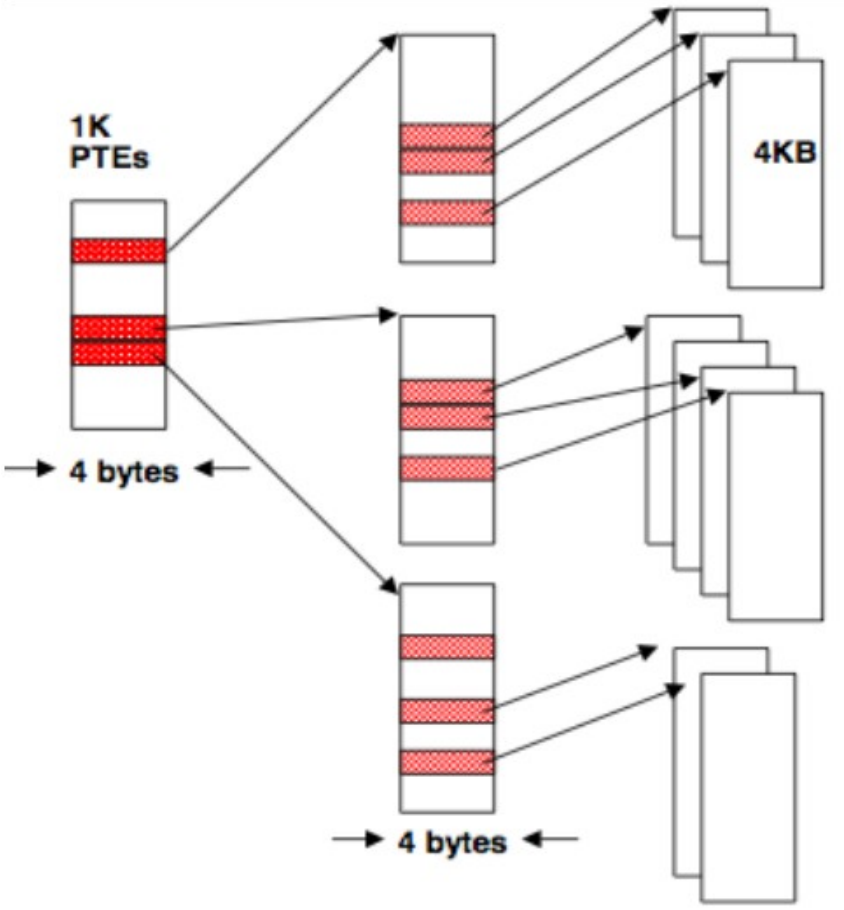
Two-level Page Tables

32 bit virtual address

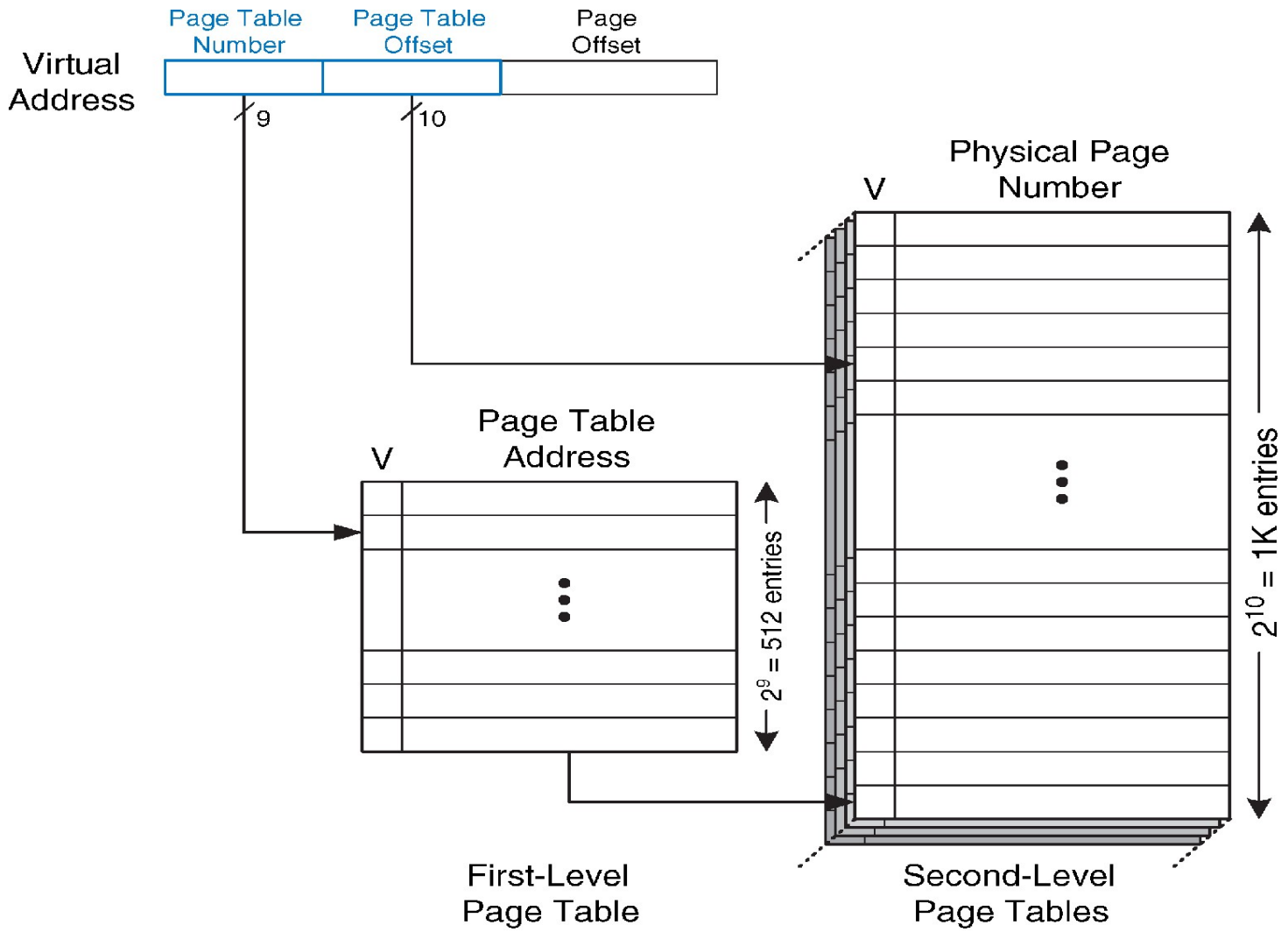


Top-level table wired in main memory

Subset of 1024 second-level tables in main memory; rest are on disk or unallocated

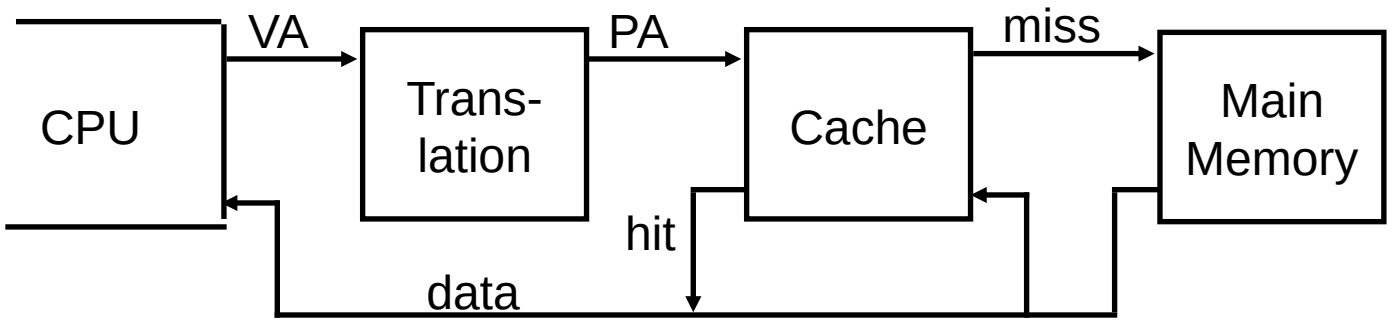


Two-level Page Table



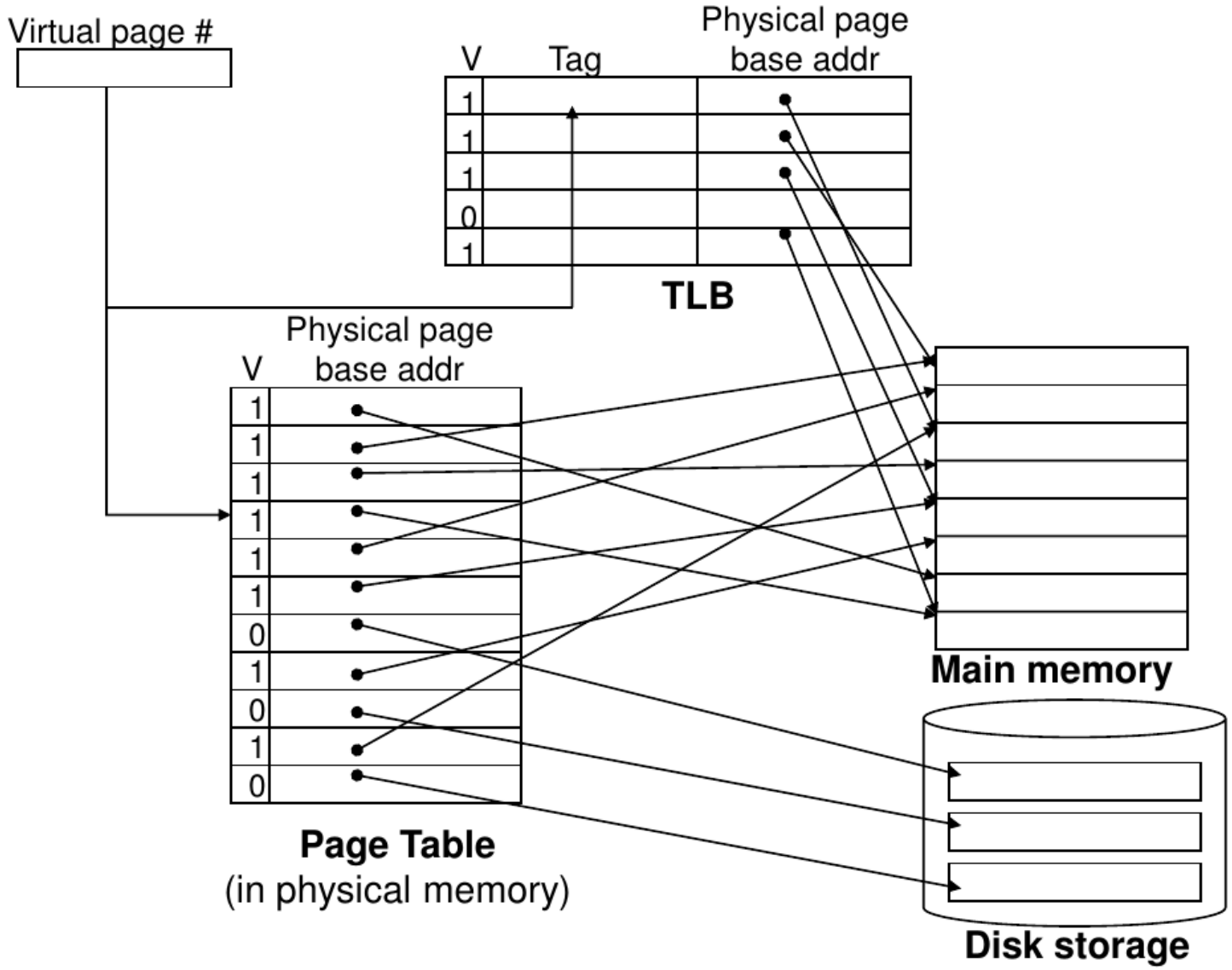
Virtual Addressing with a Cache

- Thus it takes an *extra* memory access to translate a VA to a PA



- This makes memory (cache) accesses very expensive (if every access was really *two* accesses)
- The hardware fix is to use a Translation Lookaside Buffer (TLB) – a small cache that keeps track of recently used address mappings to avoid having to do a page table lookup

Making Address Translation Fast



Translation Lookaside Buffers (TLBs)

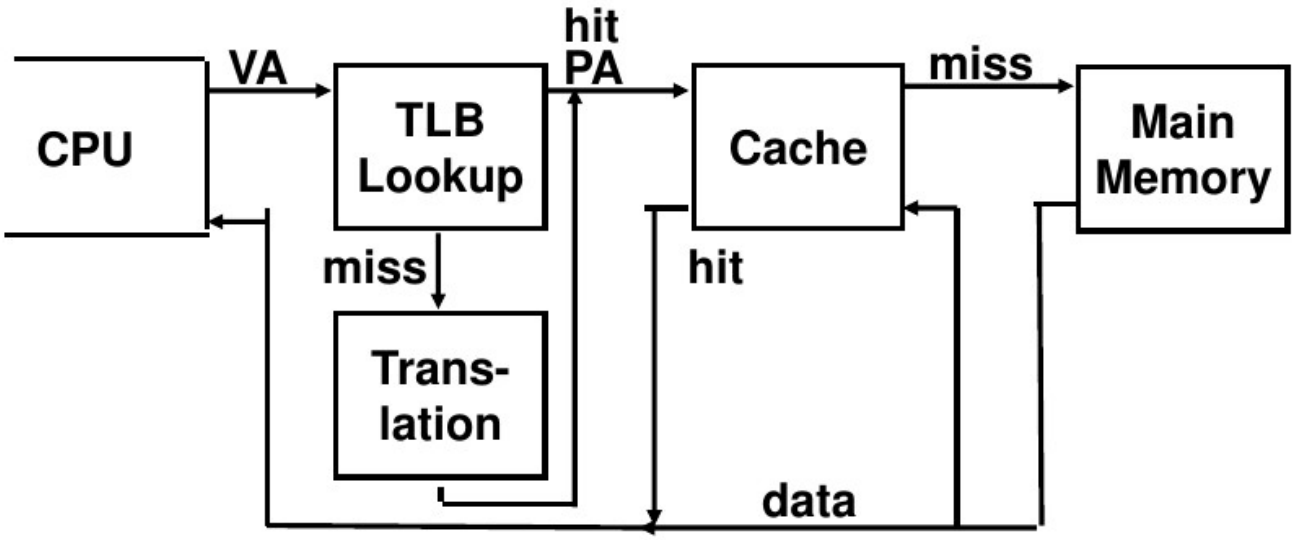
- Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped

V	Virtual Page #	Physical Page #	Dirty	Ref	Access

- TLB access time is typically smaller than cache access time (because TLBs are much smaller than caches)
  - TLBs are typically not more than 128 to 256 entries even on high end machines to maintain fast access times
  - Modern machines have introduced multi-level TLBs

A TLB in the Memory Hierarchy

- A TLB miss – is it a page fault or merely a TLB miss?
  - If the page is loaded into main memory, then the TLB miss can be handled (in hardware or software) by loading the translation information from the page table into the TLB
    - Takes 10's of cycles to find and load the translation info into the TLB
  - If the page is not in main memory, then it's a true page fault
    - Takes 10,000,000's of cycles to service a page fault
- TLB misses are much more frequent than true page faults



Cutpoint

Two Machines' Cache Parameters

	Intel P4	AMD Opteron
TLB organization	1 TLB for instructions and 1TLB for data	2 TLBs for instructions and 2 TLBs for data
	Both 4-way set associative	Both L1 TLBs fully associative with ~LRU replacement
	Both use ~LRU replacement	Both L2 TLBs are 4-way set associative with round-robin LRU
		Both L1 TLBs have 40 entries
	Both have 128 entries	Both L2 TLBs have 512 entries
		TBL misses handled in hardware
	TLB misses handled in hardware	

TLB Event Combinations

TLB	Page Table	Cache	Possible? Under what circumstances?
Hit	Hit	Hit	
Hit	Hit	Miss	
Miss	Hit	Hit	
Miss	Hit	Miss	
Miss	Miss	Miss	
Hit	Miss	Miss/ Hit	
Miss	Miss	Hit	

TLB Event Combinations

TLB	Page Table	Cache	Possible? Under what circumstances?
Hit	Hit	Hit	Yes – what we want!
Hit	Hit	Miss	Yes – although the page table is not checked if the TLB hits
Miss	Hit	Hit	Yes – TLB miss, PA in page table
Miss	Hit	Miss	Yes – TLB miss, PA in page table, but data not in cache
Miss	Miss	Miss	Yes – page fault
Hit	Miss	Miss/ Hit	Impossible – TLB translation not possible if page is not present in memory
Miss	Miss	Hit	Impossible – data not allowed in cache if page is not in memory

Summary

- The Principle of Locality:
  - Program likely to access a relatively small portion of the address space at any instant of time.
  - Temporal Locality: Locality in Time
  - Spatial Locality: Locality in Space
- Caches, TLBs, Virtual Memory all understood by examining how they deal with the four questions
  1. Where can block be placed?
  2. How is block found?
  3. What block is replaced on miss?
  4. How are writes handled?
- Page tables map virtual address to physical address
  1. TLBs are important for fast translation