5. Large and Fast: Exploiting Memory Hierarchy

Hyunok Oh
Hanyang University

5.1 Introduction

- **Memory hierarchy**
  - Exploiting the principle of locality
  - Multiple levels of memory with different speeds and sizes
    - Faster memory: Close to the processor
    - Slower and less expensive memory: Below that
- **2-level model for simplicity**
  - Upper level

    Smaller, faster, more expensive
  - Lower level

    Larger, slower, less expensive

Principle of locality

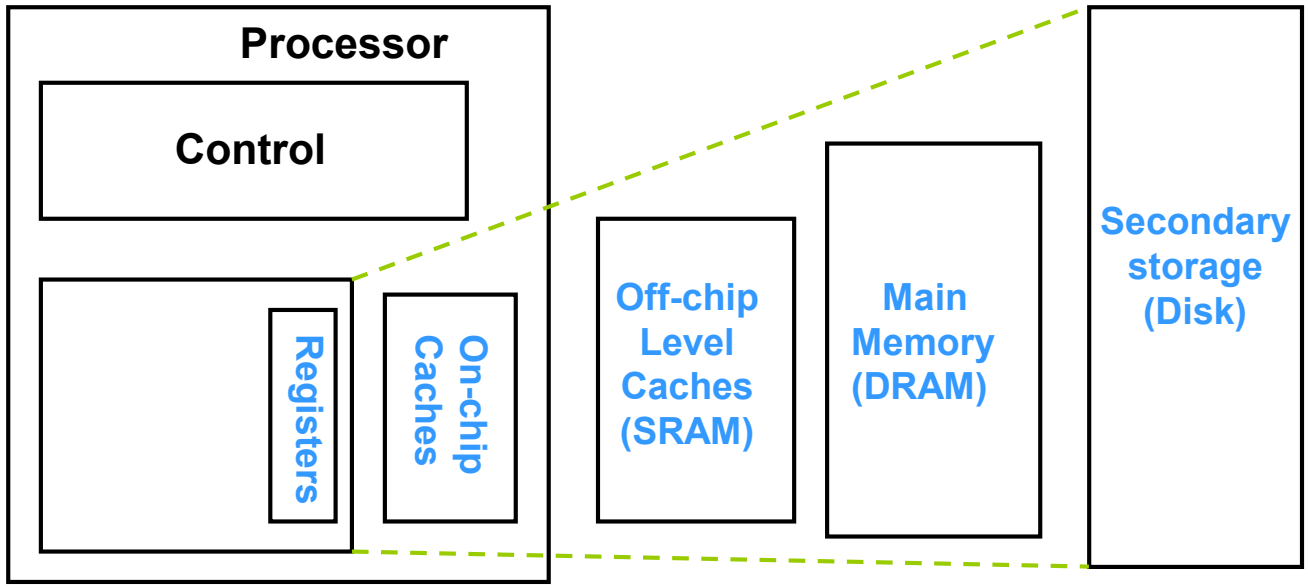
1. **Temporal locality**
   - Tendency to re-use recently accessed data items

     [ex] loops
2. **Spatial locality**
   - Tendency to reference data items that are close to other recently accessed items

     [ex] sequential execution

          array or record accesses

Memory Hierarchy in Reality



	Register	Cache	Main Memory	Disk Memory
Speed	<1ns	<10ns	100ns	107ns (10ms)
Size	100B	KB→MB	MB→GB	GB→TB
Management	Compiler	Hardware	OS	OS

Memory Technology (in 2004)

Memory technology	Typical access time	$ per GB in 2004
SRAM	0.5–5 ns	$4000–$10,000
DRAM	50–70 ns	$100–$200
Magnetic disk	5,000,000–20,000,000 ns	$0.50–$2

Block

- Minimum unit of information transfer between the two levels

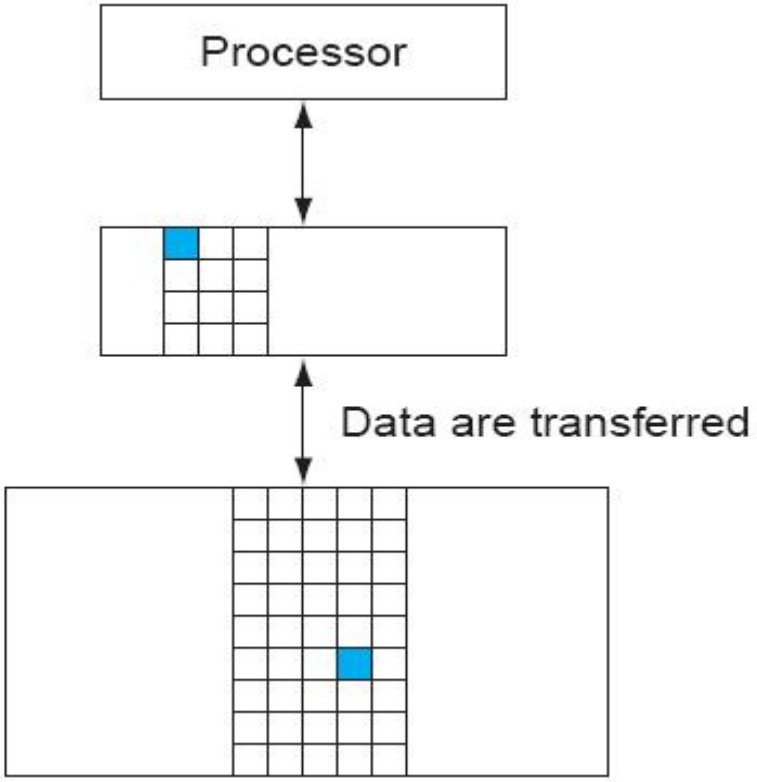


Figure 7.2

Hit and Miss

- **Hit**
  - when the data requested by the processor appears in some block in the upper level
- **Miss**
  - when the data is not found in the upper level
  - then access the lower level
- **Hit rate (=hit ratio)**
  - the fraction of memory accesses found in the upper level
- **Miss rate**
  - 1 - hit rate

Performance of the Memory Hierarchy

- **Hit time**
  - the time to access the upper level of the memory hierarchy
  - including the time needed to determine whether the access is a hit or a miss
- **Miss penalty**
  - the time to replace a block in the upper level with the corresponding block from the lower level

    + the time to deliver this block to the processor
- **Average memory access time (AMAT)**
  - hit time + miss rate x miss penalty

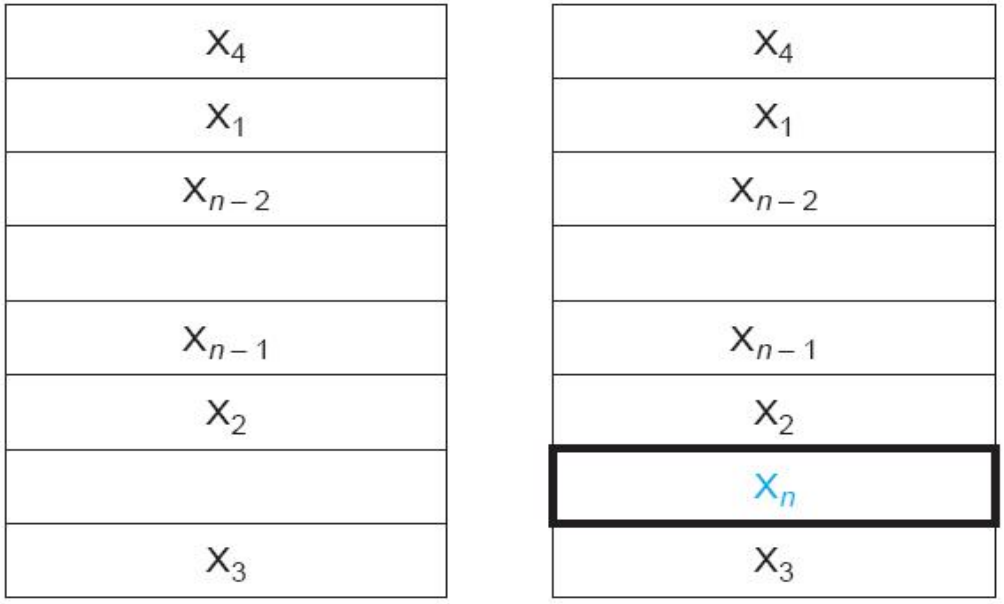
The Big Picture

- **Temporal locality and memory hierarchy**
  - Keeping more recently accessed data items closer to the processor
- **Spatial locality and memory hierarchy**
  - Moving blocks consisting of multiple contiguous words to upper level
- **Memory hierarchy with high hit rate**
  - Access time: close to that of the highest level
  - Size: equal to that of the lowest level
- **Multi-level inclusion property**
  - Level(i) $\subset$ Level(i+1)

5.2 The Basics of Caches

- **Definition of Cache**
  - The level of memory hierarchy between CPU and main memory
  - Any storage managed to take advantage of locality of access
- **The first paper … Wilkes[1965]**
  - "Slave memories and dynamic storage allocation"
- **The first implementation**
  - At the University of Cambridge by Scarott
- **The first commercial machine with a cache**
  - IBM 360/85, late 1960s
- **The first usage of the term "cache"**
  - Conti, Gibson and Pitkowsky
  - a paper in IBM Systems Journal in 1968

Simple Cache Scenario



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

Figure 7.4

- **2 questions**
  - How do we know if a data item is in the cache?
  - If it is, how do we find it?

Direct Mapped Cache

- Cache address (i.e. Index)

  (Block address) *modulo* (Number of blocks in the cache)

- When the number of entries in the cache is 2^N,

  Cache address = low-order N bits of the memory address

- Each memory location is mapped to exactly one cache location.

Fields on a Cache Line

- **Data**
- **Tag**
  - The information required to identify the original location
  - Used to decide if the word in cache is the requested one or not
  - High-order part of memory address except index
- **Valid bit**
  - Indicate whether an entry contains a valid address
- **See Figure 7.7**

Accessing a Cache

- **Example: 8-word direct mapped cache**

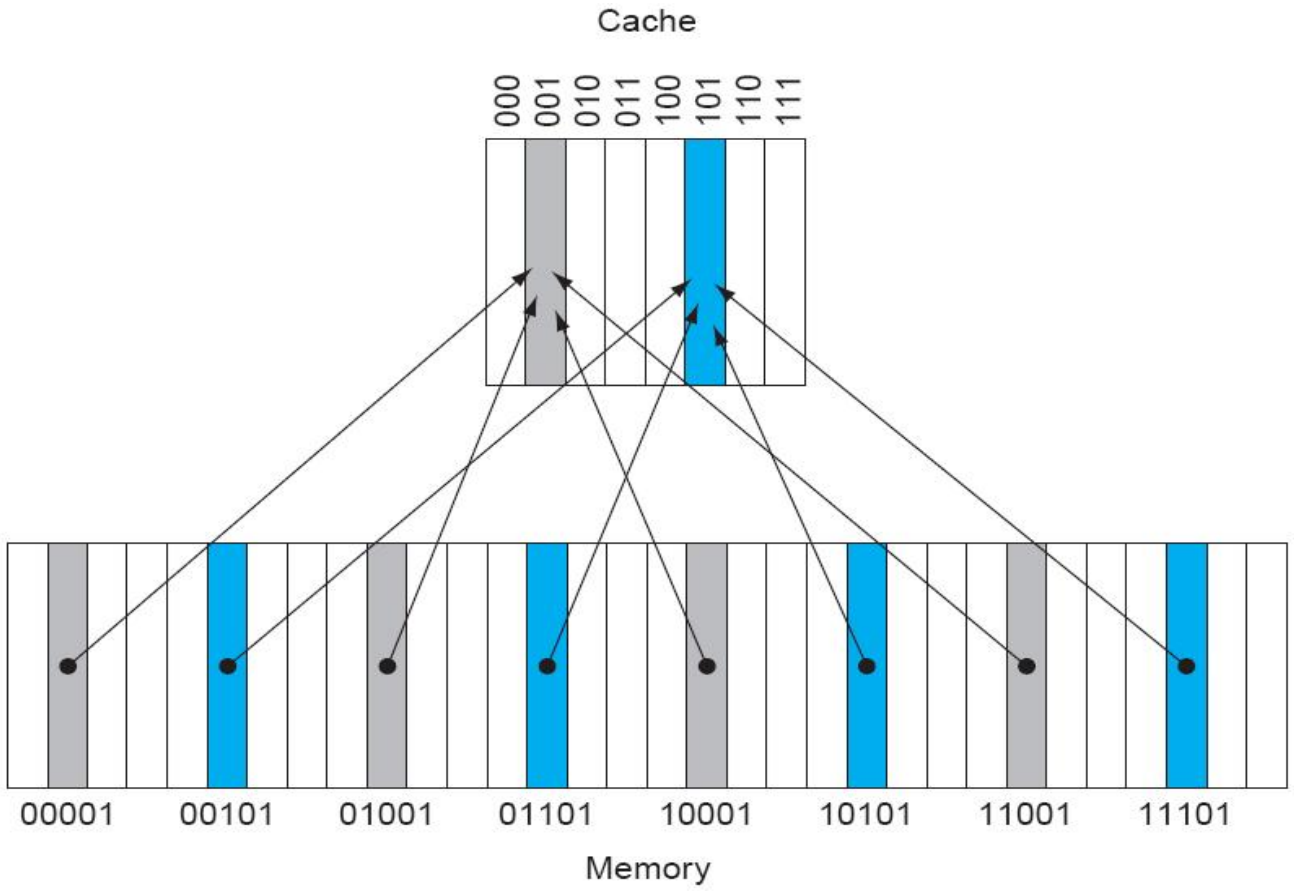


Figure 7.5

States of the Cache

Index	V	Tag	Data
000	Y	10	M[16]
001			
010	Y	11	M[26]
011	Y	00	M[3]
100			
101			
110	Y	10	M[22]
111			

22	10110
26	11010
22	10110
26	11010
16	10000
3	00011
16	10000
18	10010

Example Sequence of References

Decimal addr.	Binary addr.	Hit or miss	Assigned cache block
22	10110	miss	110 = 6
26	11010	miss	010 = 2
22	10110	hit	110 = 6
26	11010	hit	010 = 2
16	10000	miss	000 = 0
3	00011	miss	011 = 3
16	10000	hit	000 = 0
18	10010	miss	010 = 2

Cache with 4-Byte Block

- **32-bit address**
- **2^n words with one-word blocks**
- **Byte offset**

  Least significant 2 bits (not used in cache)
- **Index**

  Next n bits
- **Tag**

  32 - (n + 2) bits
- **Total number of bits**

  2^n x (valid bit + tag + block size)

  $= 2^n$ x (1 + (32-n-2) + 32)

  $= 2^n$ x (63-n)

Structure of an Example Cache

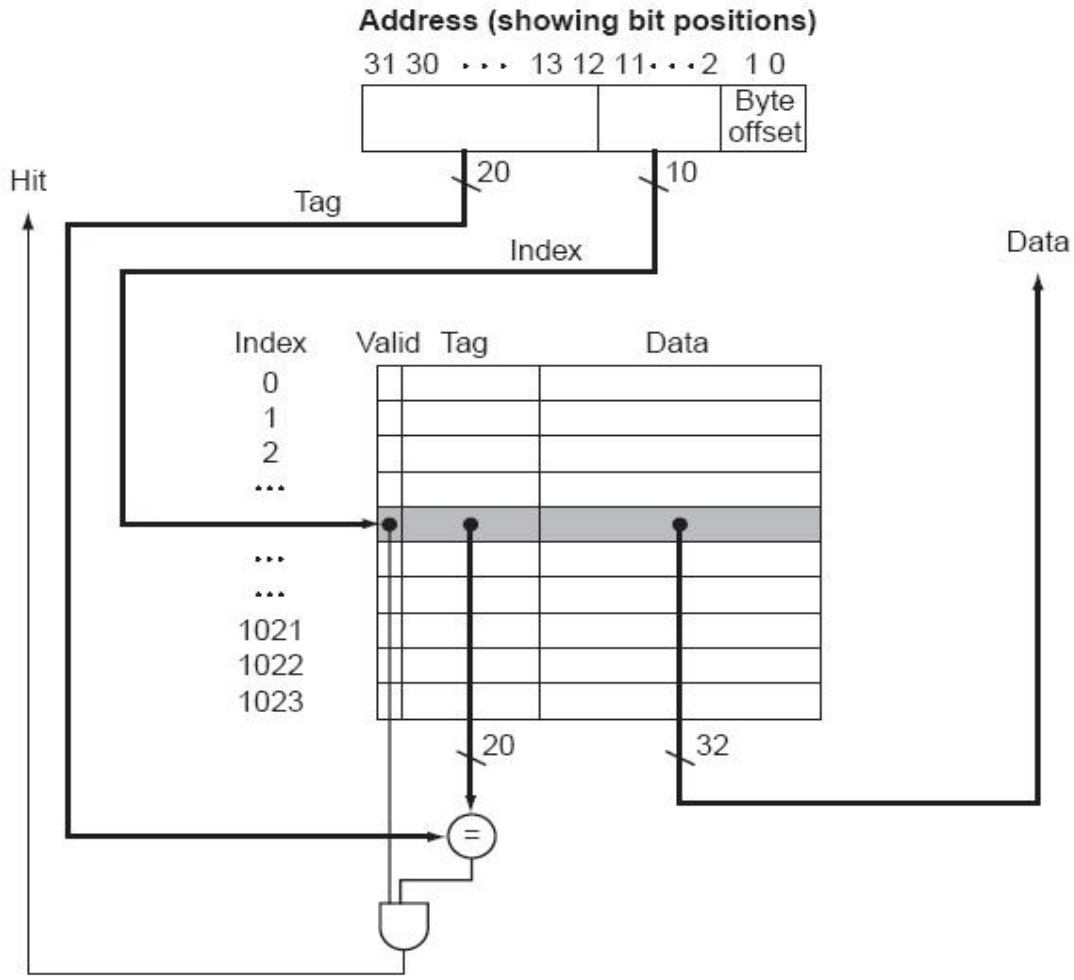


Figure 7.7

Cache with 2^m-Word Block

- 32-bit address
- 2^n blocks with 2^m-word blocks
- Byte offset : 2 bits
- Block offset : m bits
- Index : n bits
- Tag : 32 - (n + m + 2) bits
- Total number of bits

2^n x (block size + tag size + valid field size)

$= 2^n \times (2^m \times 32 + (32\text{-}n\text{-}m\text{-}2) + 1)$

$= 2^n \times (2^m \times 32 + 31 - n - m)$

Example: Bits in a Cache

- **Calculate total number of bits for the cache.**
  - Direct-mapped
  - 16 KB of data with 4-word blocks
  - 32-bit address

[Answer]

16 KB = 4K words = 2^{12} words

Number of blocks = 2^{12} / 4 = 2^{10}

Block size = 4 words = 4x4 bytes = 4x4x8 bits = 128 bits

Cache size = 2^{10} x (128 + (32-10-2-2) + 1)

= 2^{10} x 147

= 147 Kbits = 18.4 KBytes

Almost 1.15 times as large as data size of the cache.

Example: Multiword Cache Block

- **What block number does byte address 1200 map to in a cache with 64 blocks and 16 bytes/block ?**

[Answer]

- Block number = $\lfloor 1200/16 \rfloor = 75$
- Cache block number
  = ($\lfloor$ byte address / bytes per block $\rfloor$)
  modulo (blocks in cache)
  = 75 modulo 64
  = 11

Miss rate vs. Block size

- **Larger blocks**
  - Increased spatial locality
  - Decreased miss rate

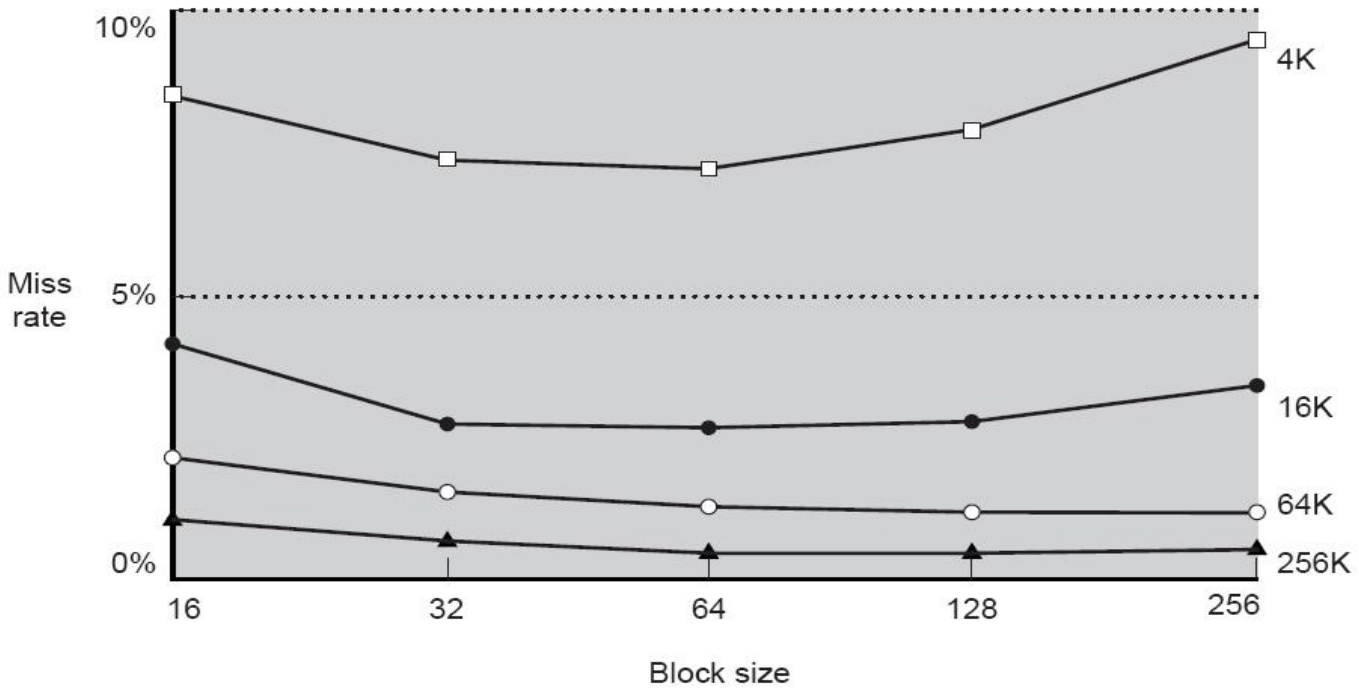


Figure 7.8

Block Size vs. Performance

- **Too large blocks**
  - Small number of blocks in the cache
  - Increased miss rate
  - Increased miss penalty
- **Miss penalty**
  - Miss penalty = block fetch time + cache load time
  - Block fetch time = latency to the first word
    + transfer time for the rest of the block
  - Transfer time $\propto$ block size
  - Increased as the block size increases

Elaboration: Reducing Miss Penalty

1. **Early restart**
   - Resume execution as soon as the requested word of the block is returned
   - Effective for instruction accesses due to the sequential access patterns
   - Less effective for data caches
2. **Requested word first (or critical word first)**
   - Even more sophisticated scheme
   - Organize the memory so that the requested word is transferred first
   - Slightly faster than early restart

Handling Cache Misses

- **Handling instruction cache miss**
  1) Send PC – 4 to the memory.
  2) Instruct main memory to perform a read and wait for the memory to complete its access.
  3) Write the cache entry. (Update data and tag fields and turn on the valid bit.)
  4) Restart the instruction execution at the first step.
- **Data cache miss**
  - Stall the processor until the memory responds with the data.

Handling Writes

- **Write-through policy**
  - Always write the data into both the memory and the cache.
  - Keep the main memory and the cache consistent.
- **On write-miss**
  1. Fetch the block containing the word.
  2. Overwrite the word into the cache block.
  3. Also write the word to main memory.
- **On write-hit**
  1. Overwrite the word into the cache block.
  2. Also write the word to main memory.

Problems of Write-through

- **Performance degradation**
  - Every write causes the data to be written to main memory.
  - Slow down the machine considerably.
- **SPEC2000 integer benchmarks**
  - 10% of the instructions are stores.
  - If the CPI without cache miss = 1.0,

    spending 100 extra cycles on every write,

    CPI = 1.0 + 100 x 10% = 11
  - Reducing performance by more than a factor of 10

Write Buffer

- One solution to the above problem
- Stores the data while it is waiting to be written to memory
- After writing the data into the cache and into the write buffer, the processor can continue execution

Write-Back

- The new value is written only to the cache.
- The modified block is written to the main memory when it is replaced.
- Better performance
- More complex to implement

Elaboration

- **Policies on write misses**
  1. Fetch-on-miss, fetch-on-write or allocate-on-miss
     - Most write-through caches
     - Allocating a cache block to the address that missed and fetching the rest of the block into the cache before writing the data
  2. Write-around
     - Allocate-on-miss but no-fetch-on-write
     - No-allocate-on-write
     - Motivation: sometimes write entire blocks before reading them
     - Complex to implement in a multiword block

An Example Cache: The Intrinsity FastMATH processor

- **Intrinsity FastMATH**
  - Embedded microprocessor
  - MIPS architecture with a simple cache implementation
  - 12-stage pipeline
  - Split caches
    - Each cache: 16KB with 16-word block and 256 blocks
  - Write policy
    - OS selectable: write-through or write-back
  - Miss rate
    - Instruction: 0.4%
    - Data: 11.4%
    - Effective combined: 3.2%

Cache of Intrinsity FastMATH

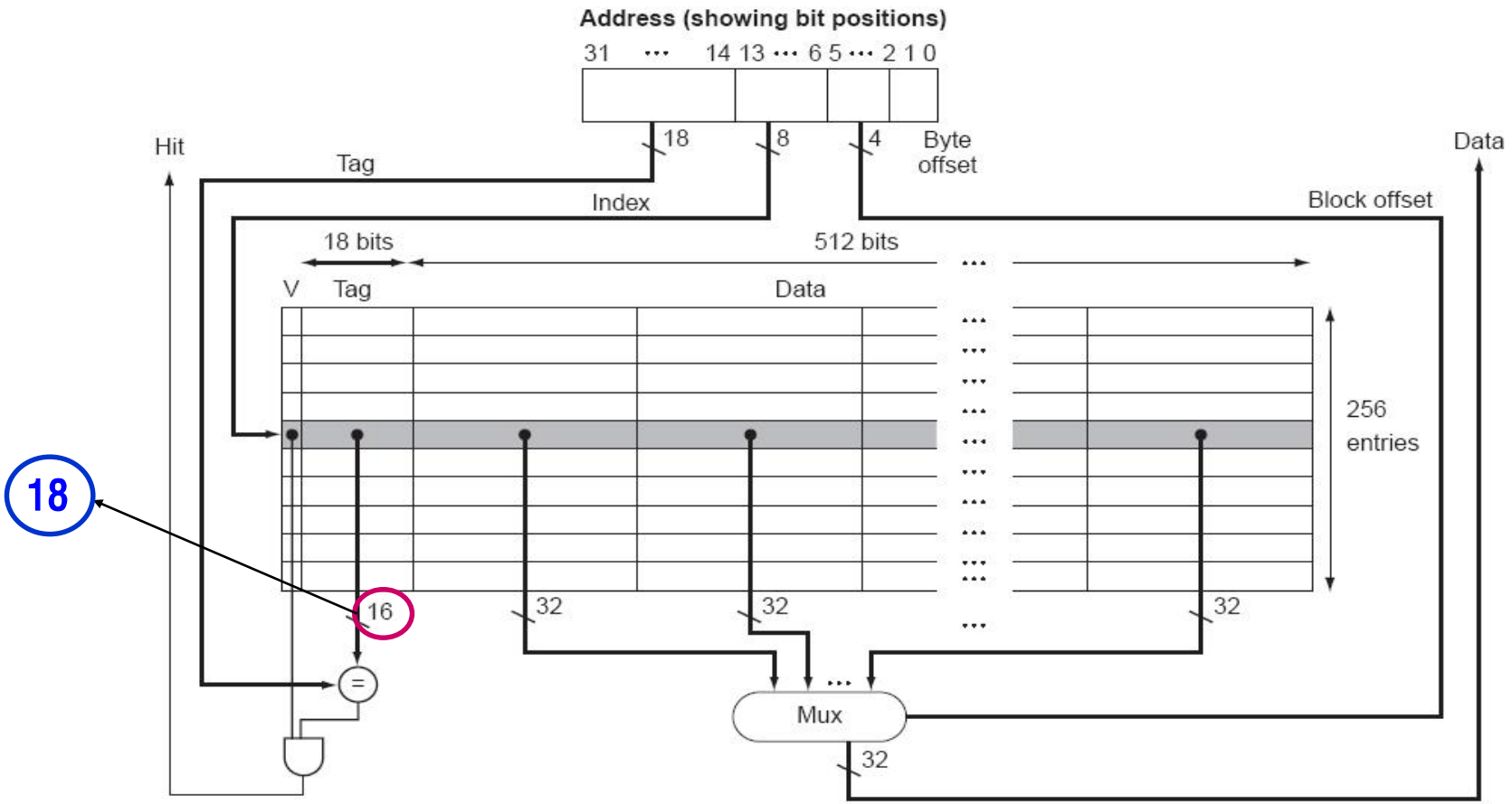


Figure 7.9

Elaboration

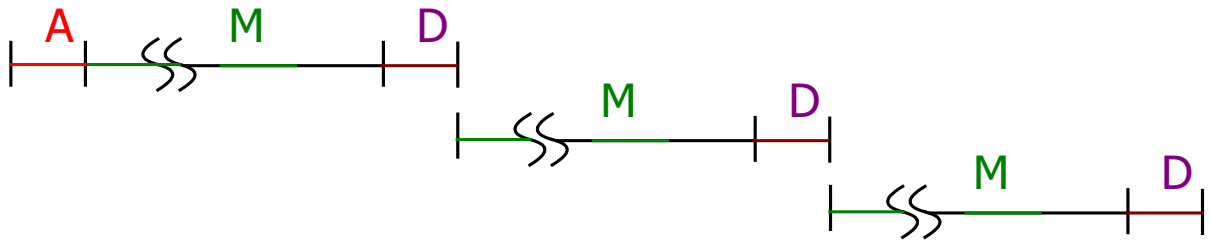
- Split caches vs. Combined (or Unified) cache
  - Combined cache: better hit rate
  - Split cache: larger bandwidth
- Miss rates for Intrinsity FastMATH
  - Total cache size: 32 KB
  - Split cache: 3.24%
  - Combined cache: 3.18%
- High bandwidth is more effective than high hit rate.
- Thus, we cannot use miss rate as the sole measure of cache performance.

Designing the Memory System to Support Caches

- **Increased bandwidth from memory to cache**
  - Reduced miss penalty
  - Larger block size with low miss penalty
- **Bus**
  - Connecting memory to processor
  - Much slower than the processor, by as much as a factor of 10
- **Assumptions**
  - 1 memory bus clock cycle to send address
  - 15 clock cycles for each DRAM access initiated
  - 1 clock cycle to send a word of data
  - Cache block of 4 words

One-word-wide Memory

- A one-word-wide bank of DRAMs
- All accesses are made sequentially.
- Miss penalty
  - 1 + 4 x 15 + 4 x 1 = 65 (clock cycles)
- The number of bytes transferred per clock cycle for a single miss
  - (4 x 4) / 65 = 0.25 (bytes/cycle)
- Timing



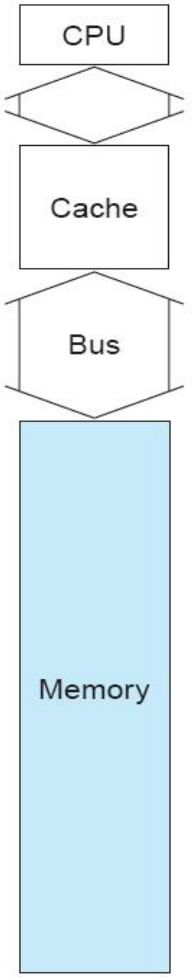


Figure 7.11a

Wide Memory (1/2)

- Increase the bandwidth to memory by widening the memory and the bus
  - Parallel access to all the words of the block
- **Major costs**
  - Wider bus
  - Potential increase in cache access time due to the multiplexor and control logic between processor and cache

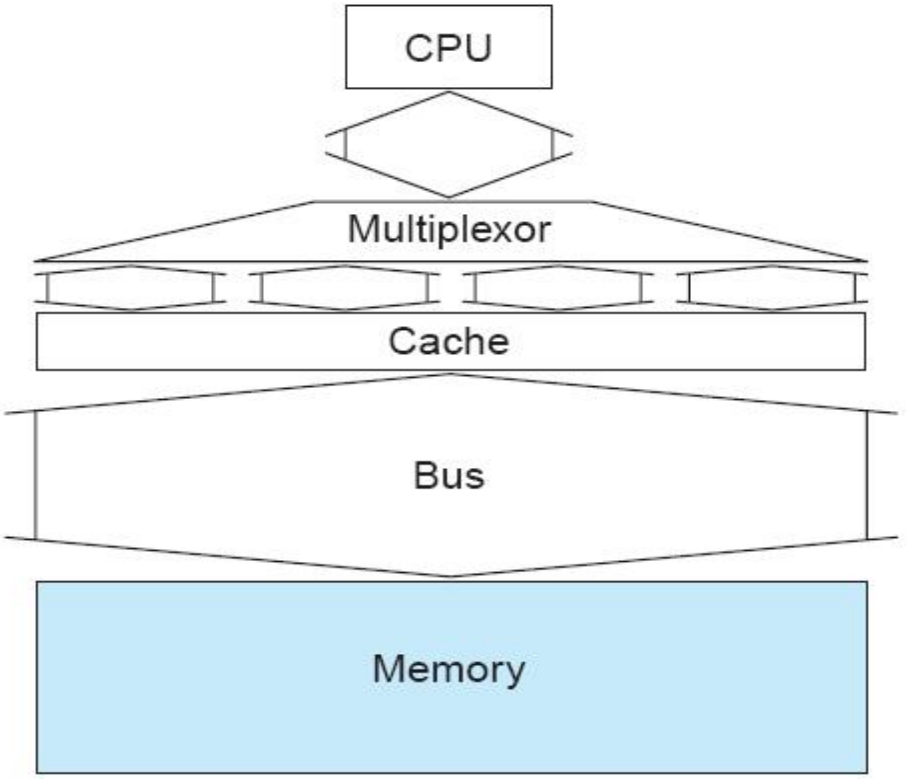
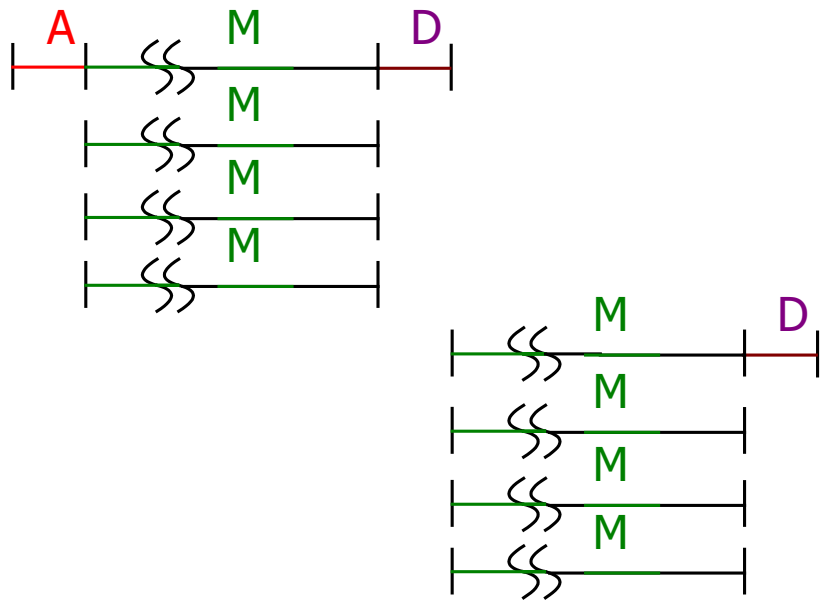


Figure 7.11b

Wide Memory (2/2)

- **If a main memory width = 2 words**
  - Miss penalty = 1 + 2×15 + 2×1 = 33 (clock cycles)
  - Bandwidth for a single miss = 4×4 / 33 = 0.48 (bytes/cycle)
- **If a main memory width = 4 words**
  - Miss penalty = 1 + 1×15 + 1×1 = 17 (clock cycles)
  - Bandwidth for a single miss = 4×4 / 17 = 0.94 (bytes/cycle)

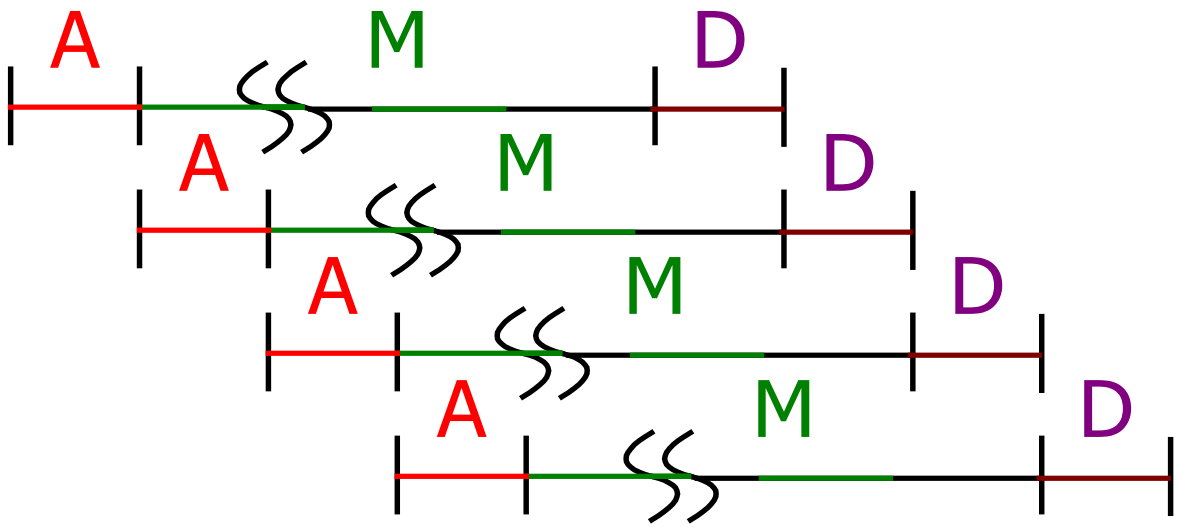


Interleaved Memory (2/1)

- Widening the memory but not the bus and cache
- Organize the memory chips in banks to read or write multiple words in one access time
- Advantage of incurring the full memory latency only once
- Each bank can be written independently
  - 4 times write bandwidth
  - Less stall occurs in write-through cache

Interleaved Memory (2/2)

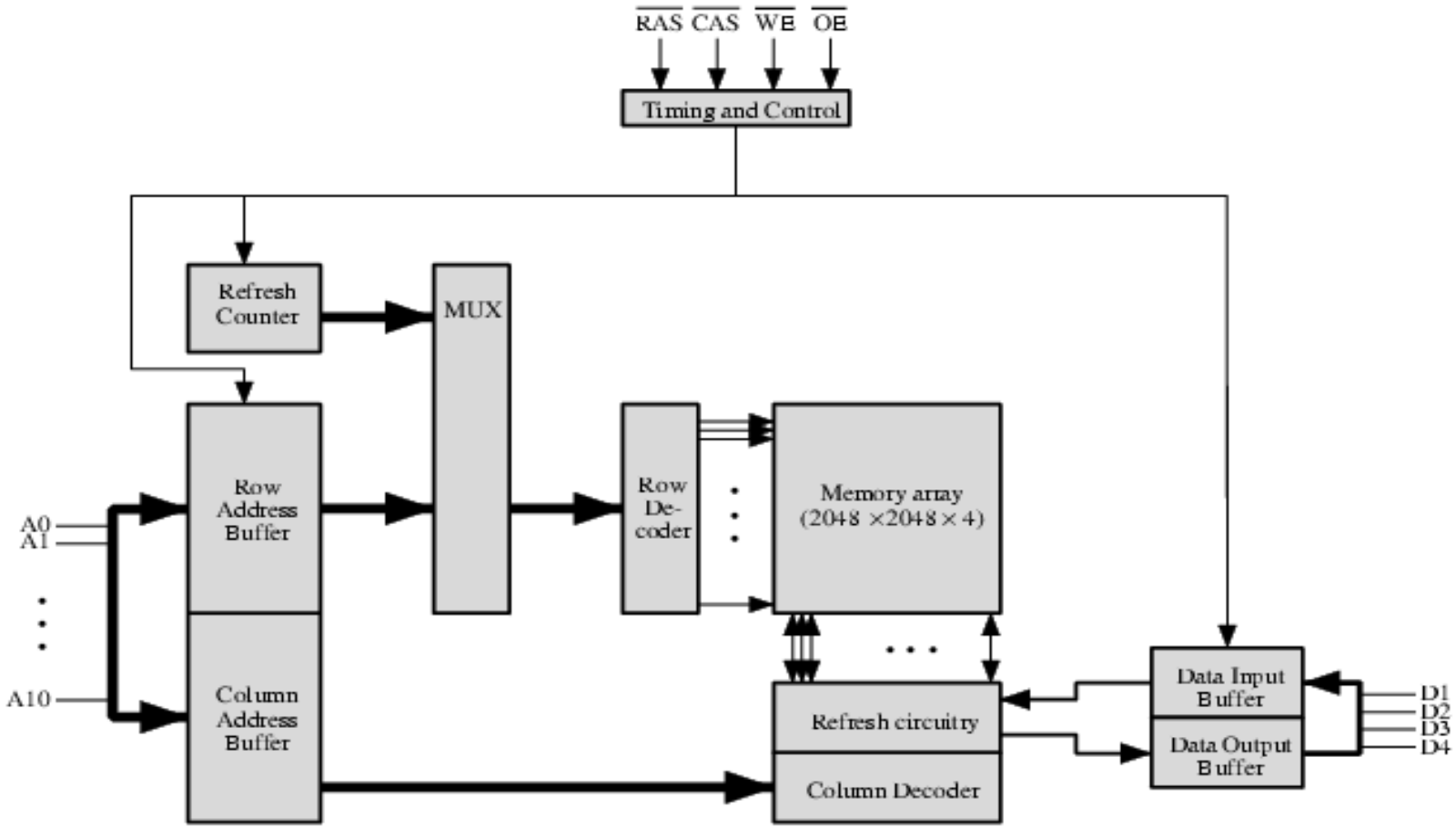
- **With 4 banks**
  - 4-way interleaved memory
  - Miss penalty = 1 + 1x15 + 4x1 = 20 (clock cycles)
  - Bandwidth for a single miss = 4×4 / 20 = 0.80



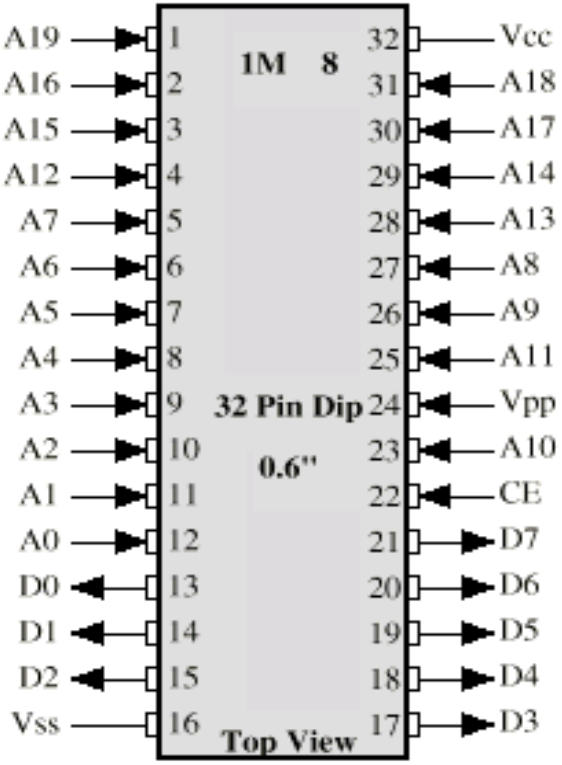
Elaboration

Year	Chip size	$/MB	Total access time to a new row/col.	Column access to a existing row
1980	64Kb	1500	250 ns	150 ns
1983	256Kb	500	185 ns	100 ns
1985	1Mb	200	135 ns	40 ns
1989	4Mb	50	110 ns	40 ns
1992	16Mb	15	90 ns	30 ns
1996	64Mb	10	60 ns	12 ns
1998	128Mb	4	60 ns	10 ns
2000	256Mb	1	55 ns	7 ns
2002	512Mb	0.25	50 ns	5 ns
2004	1024Mb	0.10	45 ns	3 ns

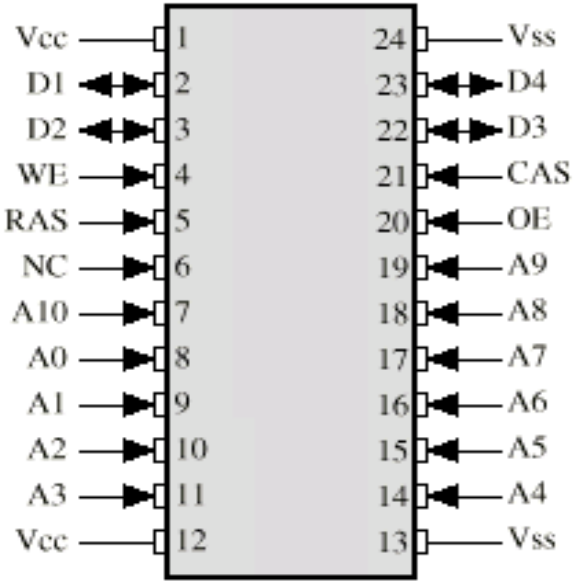
Typical 16 Mb DRAM (4M x 4)



Packaging

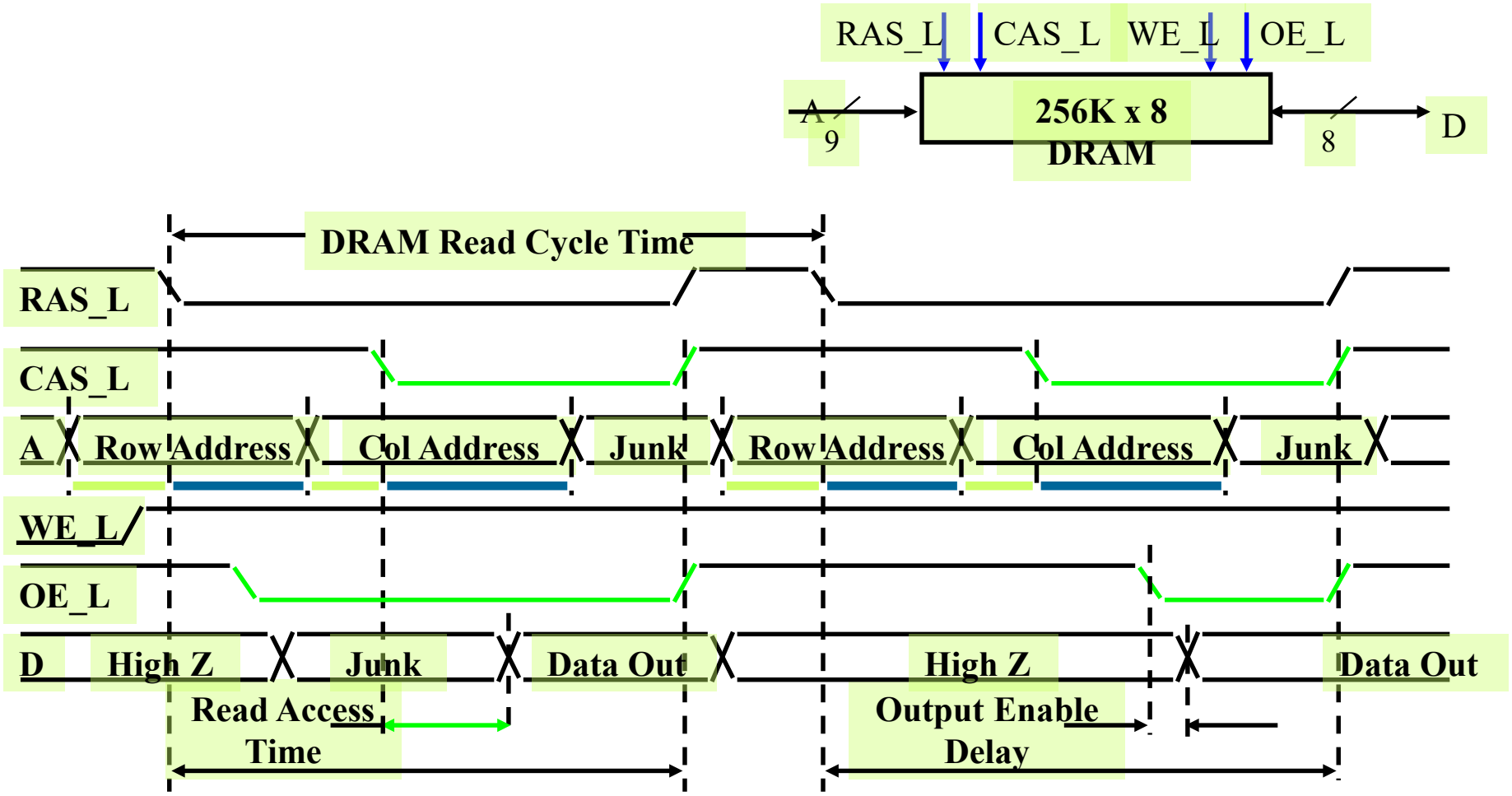


(a) 8 Mbit EPROM



(b) 16 Mbit DRAM

DRAM Read Timing



4 Key DRAM Timing Parameters

- **t_{RAC}: minimum time from RAS line falling to the valid data output.**
  - Quoted as the speed of a DRAM when buy
  - A typical 4Mb DRAM t_{RAC} = 60 ns
  - Speed of DRAM since on purchase sheet?
- **t_{RC}: minimum time from the start of one row access to the start of the next.**
  - t_{RC} = 110 ns for a 4Mbit DRAM with a t_{RAC} of 60 ns
- **t_{CAC}: minimum time from CAS line falling to valid data output.**
  - 15 ns for a 4Mbit DRAM with a t_{RAC} of 60 ns
- **t_{PC}: minimum time from the start of one column access to the start of the next.**
  - 35 ns for a 4Mbit DRAM with a t_{RAC} of 60 ns

5.3 Measuring and Improving Cache Performance

- **Two techniques for improving cache performance**
  1. Using associativity
  2. Multilevel caching
- **CPU time with cache**
  - CPU time = (CPU execution clock cycles + memory-stall clock cycles)×clock cycle time

    ,where CPU execution time includes cache hit time
  - Main cause of memory-stall is cache miss

Memory Stalls

- Memory-stall clock cycles
  = read-stall clock cycles + write-stall clock cycles
- Read-stall cycles
  = (reads/program) x read miss rate
  x read miss penalty
- Write-stall cycles
  = (writes/program) x write miss rate
  x write miss penalty + write buffer stalls

Simplification

- **Assumptions**
  - read miss penalty = write miss penalty
  - Read miss rate = write miss rate
  - negligible write buffer stalls
- **Memory-stall clock cycles**

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

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

Example: Calculating Cache Performance

- Instruction cache miss rate = 2%
- Data cache miss rate = 4%
- CPI = 2 without any memory stalls
- Miss penalty = 100 cycles
- Instruction frequencies
  - SPECint2000 from Figure 3.26
- **How much faster with a perfect cache?**

[Answer]

- Instruction count = I
- Instruction miss cycles = I × 2% × 100 = 2.00 × I
- Data miss cycles = I × 36% × 4% × 100 = 1.44 × I
  (∵ frequency of loads/stores = 36%)
- Total stall cycles = 2.00 × I + 1.44 × I = 3.44 × I
- Thus, CPI with memory stalls = 2 + 3.44 = 5.44
- Performance with perfect cache is better by
  5.44 / 2 = 2.72

When CPI = 1, speedup = (1+3.44)/1 = 4.44.

Example: With Increased Clock Rate

- **Doubling clock rate of above example with the same memory.**

[Answer]

- The new miss penalty = 100 x 2 = 200 clock cycles
- Total miss cycles per instruction
  = (2% x 200) + 36% x (4% x 200) = 6.88
- Faster machine's CPI = 2 + 6.88 = 8.88
- Performance $_{fast}$ / Performance $_{slow}$
  = Execution time $_{slow}$ / Execution time $_{fast}$
  = (IC x CPI $_{slow}$ x Clock cycle) / (IC x CPI $_{fast}$ x (Clock cycle /2))
  = 5.44 / ((8.88 x (1/2)) = 1.23
- Machine with faster clock is about 1.23 times faster rather than 2 times faster.

Increased Importance of Cache Performance

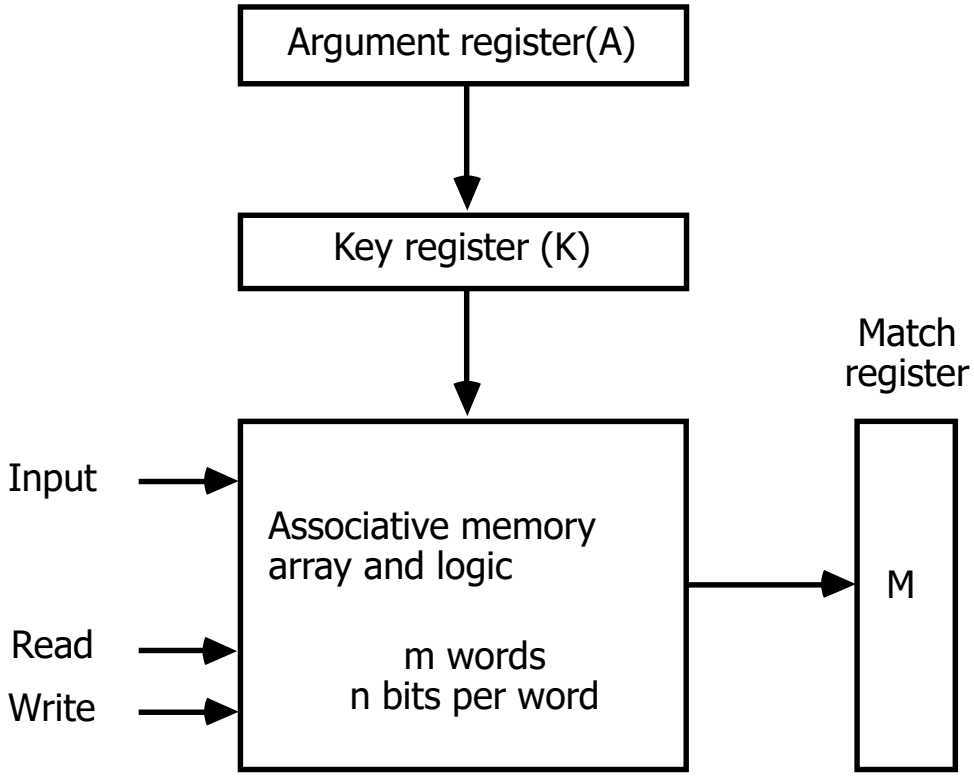
- **Increased cache penalties with faster processor**
  - With lower CPI, impact of stall cycles increases.
  - With higher clock rate, miss penalty increases.
    - Because miss penalty is measured in processor clock cycles.
- **Hit time**
  - Slow hit time increases processor cycle time.
  - With pipelines deeper than 5 stages,
    - More than 1 cycle for a cache hit
    - Likely to add another stage to the pipeline
  - What increases hit time
    - Large cache
    - Associative cache
    - Etc.

Reducing Cache Misses by More Flexible Placement of Blocks

- **3 placement schemes**
  1. Direct mapping
  2. Set-associative mapping
  3. (Fully) associative mapping
- **Direct mapped cache**
  - Direct mapping to a single location in the upper level
- **Fully associative cache**
  - A memory block can be placed anywhere in the cache.
  - Special hardware for parallel comparison
  - CAM (content addressable memory) = associative memory

Associative Memory

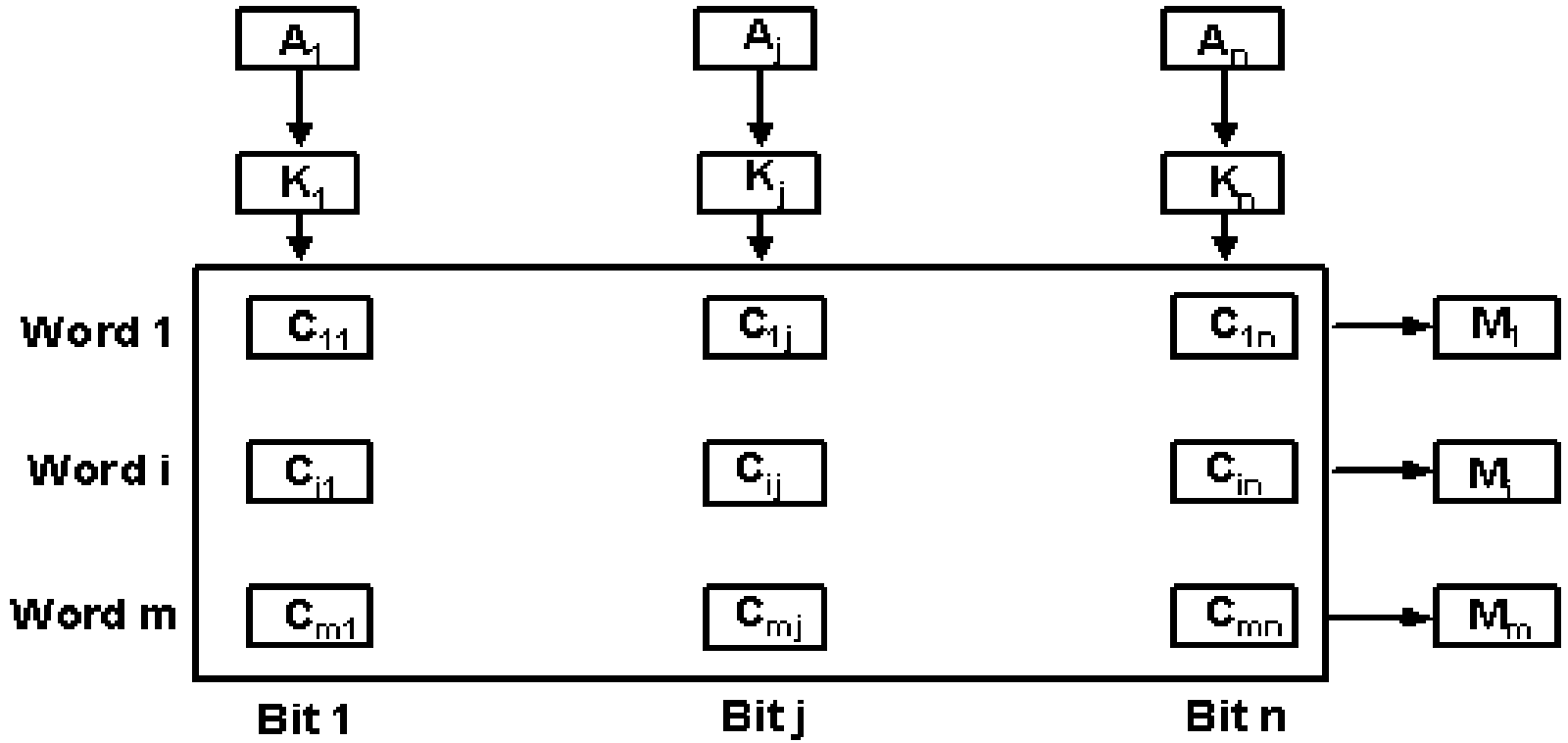
- Accessed by the content of the data rather than by an address
- Also called Content Addressable Memory (CAM)



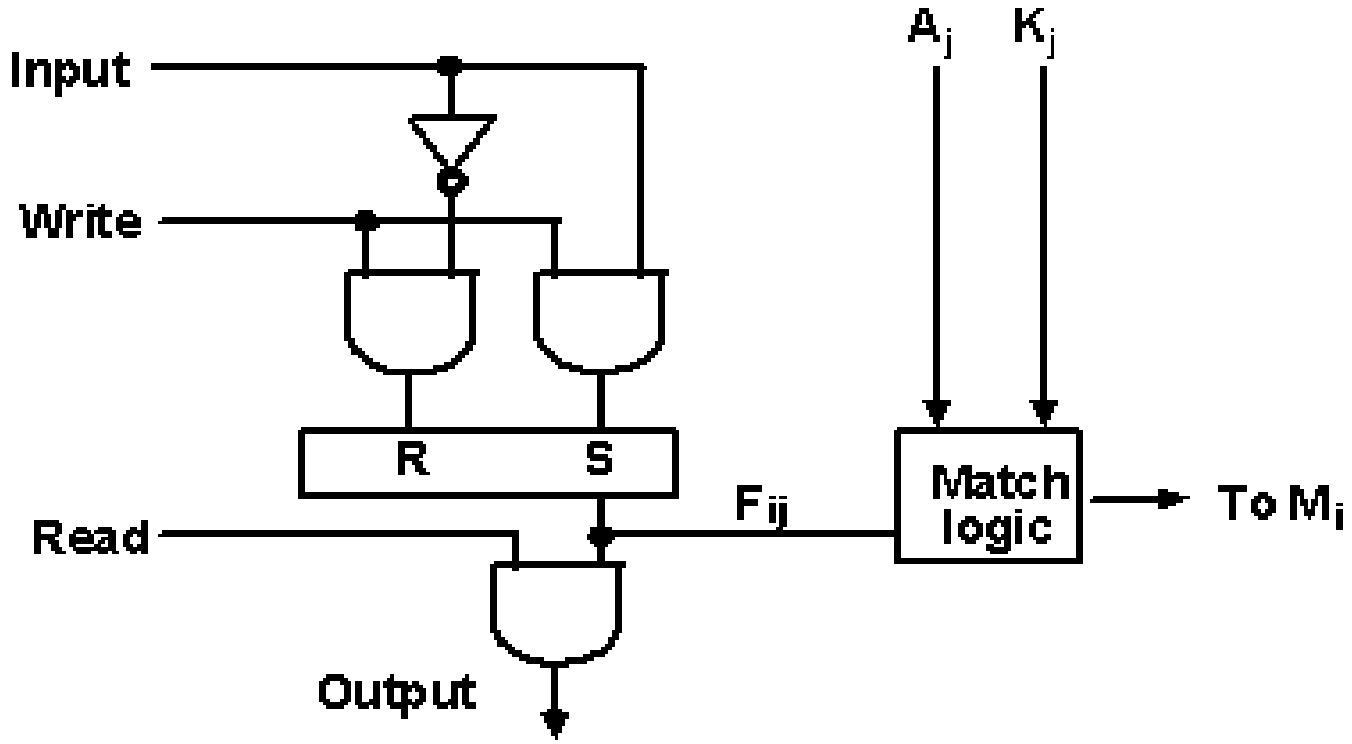
Operation of Associative Memory

- Compare each word in CAM in parallel with the contents of A (Argument Register)
- If CAM Word[i] = A, M(i) = 1
- Read sequentially accessing CAM for M(i) = 1
- K (Key Register) provides a mask for choosing a particular field or key in the argument in A
  - only those bits in the argument that have 1's in their corresponding position of K are compared

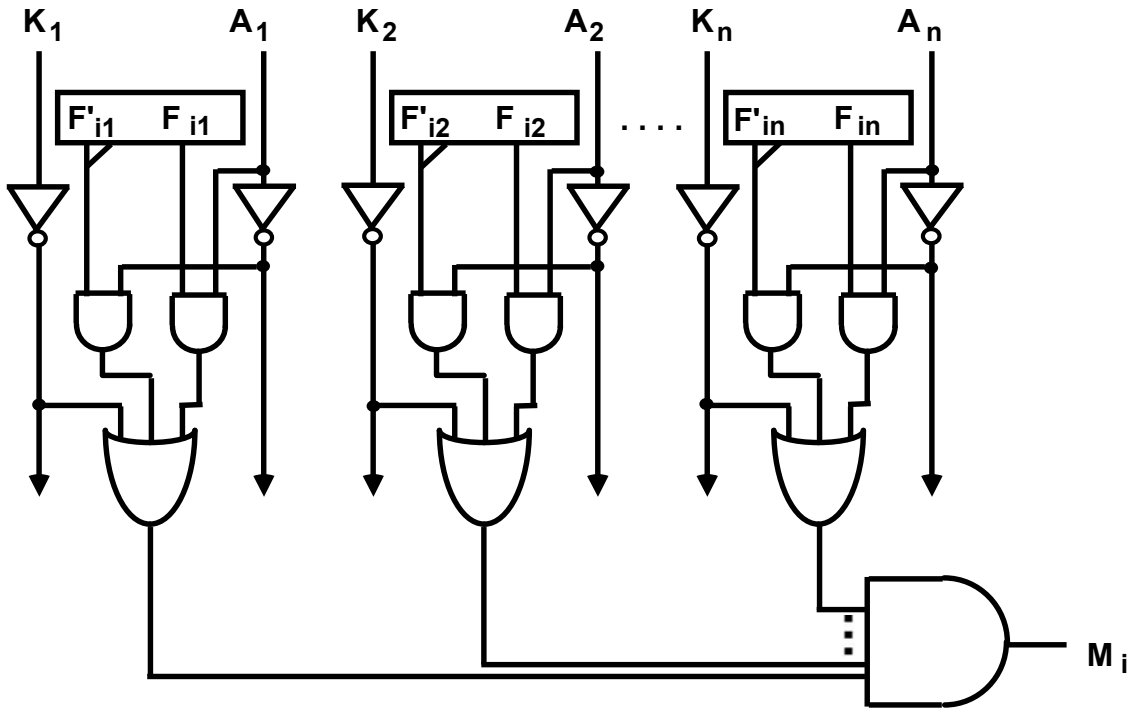
Internal Organization of CAM



Typical Cell C_{ij} of CAM



Match Logic



- $M_i = \Pi\{(F_{ij} \cdot A_j)+(F_{ij}' \cdot A_j')+K_j'\}$

n-Way Set Associative Cache

- Each block can be placed in one of n locations.
- Set size = n (blocks)

  Number of sets = cache size / n
- Each block in the memory maps to a unique set in the cache given by the index field.

  → A block is direct mapped into a set.
- A block can be placed in any element of that set.

  → All the blocks in the set are searched for a match.
- Increased degree of associativity (=n)

  → decreased miss rate

  increased hit time

Comparison of Three Block Placement Policies

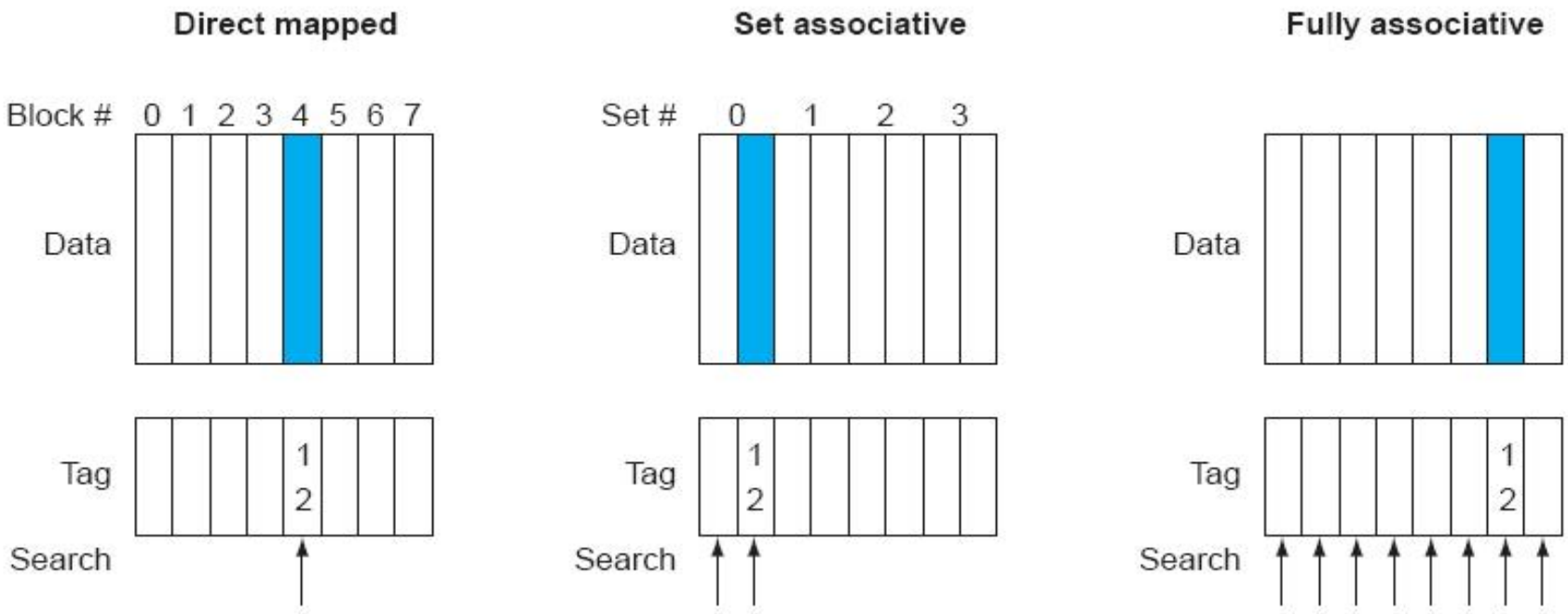


Figure 7.13

8-Block Cache Example

- **When cache size = m blocks**
  - 1-way set associative mapping = direct mapping
  - m-way set associative mapping = fully associative mapping

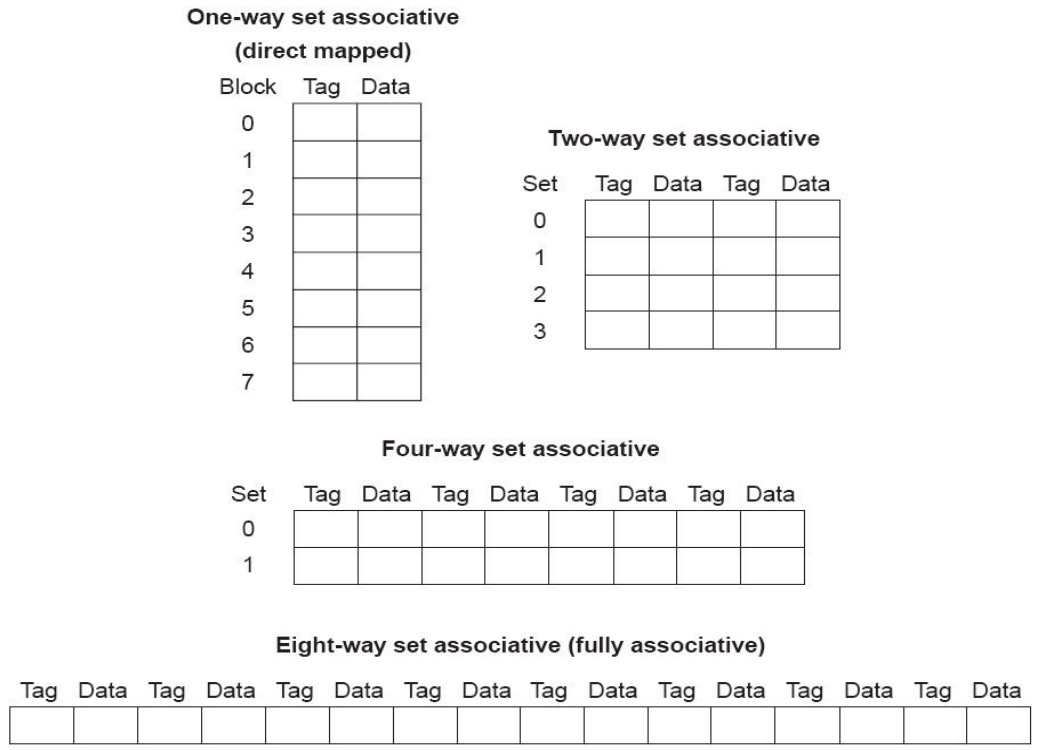


Figure 7.14

Example: Associativity in Caches

- Three small caches, each consisting of four one-word blocks
  - Direct mapped
  - Two-way set associative
  - Fully associative
- Replacement policy = LRU
- Access sequence = (0, 8, 0, 6, 8)
- **Calculate the number of misses for each cache.**

[Answer-1/3]

1. Direct mapped cache : 5 misses

Block address	Cache block
0	(0 mod 4) = 0
4	(4 mod 4) = 0
6	(6 mod 4) = 2
8	(8 mod 4) = 0

Block addr.	H/M	Contents of cache blocks			
		0	1	2	3
0	Miss	M[0]			
8	Miss	M[8]			
0	Miss	M[0]			
6	Miss	M[0]		M[6]	
8	Miss	M[8]		M[6]	

[Answer-2/3]

2. 2-way set associative cache : 4 misses

Block address	Cache block
0	(0 mod 2) = 0
4	(4 mod 2) = 0
6	(6 mod 2) = 0
8	(8 mod 2) = 0

Block addr.	H/M	Contents of cache blocks			
		Set 0		Set 1	
0	Miss	M[0]			
8	Miss	M[0]	M[8]		
0	Hit	M[0]	M[8]		
6	Miss	M[0]	M[6]		
8	Miss	M[8]	M[6]		

[Answer-3/3]

3. Fully associative cache : 3 misses

Block addr.	H/M	Contents of cache blocks			
0	Miss	M[0]			
8	Miss	M[0]	M[8]		
0	Hit	M[0]	M[8]		
6	Miss	M[0]	M[8]	M[6]	
8	Hit	M[0]	M[8]	M[6]	

Large Degree of Associativity

- Decreased miss rate

  but, becomes smaller when cache size becomes larger
- **Figure 7.15**
- **In case of gcc**
  - Direct -> 2-way : improved by 20%
  - 2-way -> 4-way : no further improvement
- **In case of spice**
  - Little opportunity for improvement

    because of too low miss rate

Miss Rates vs. Associativity

- **Data cache miss rate of Intrinsity FastMATH**
  - 10 SPEC2000 programs

Associativity	Data miss rate
1	10.3%
2	8.6%
4	8.3%
8	8.1%

Figure 7.15

Performance

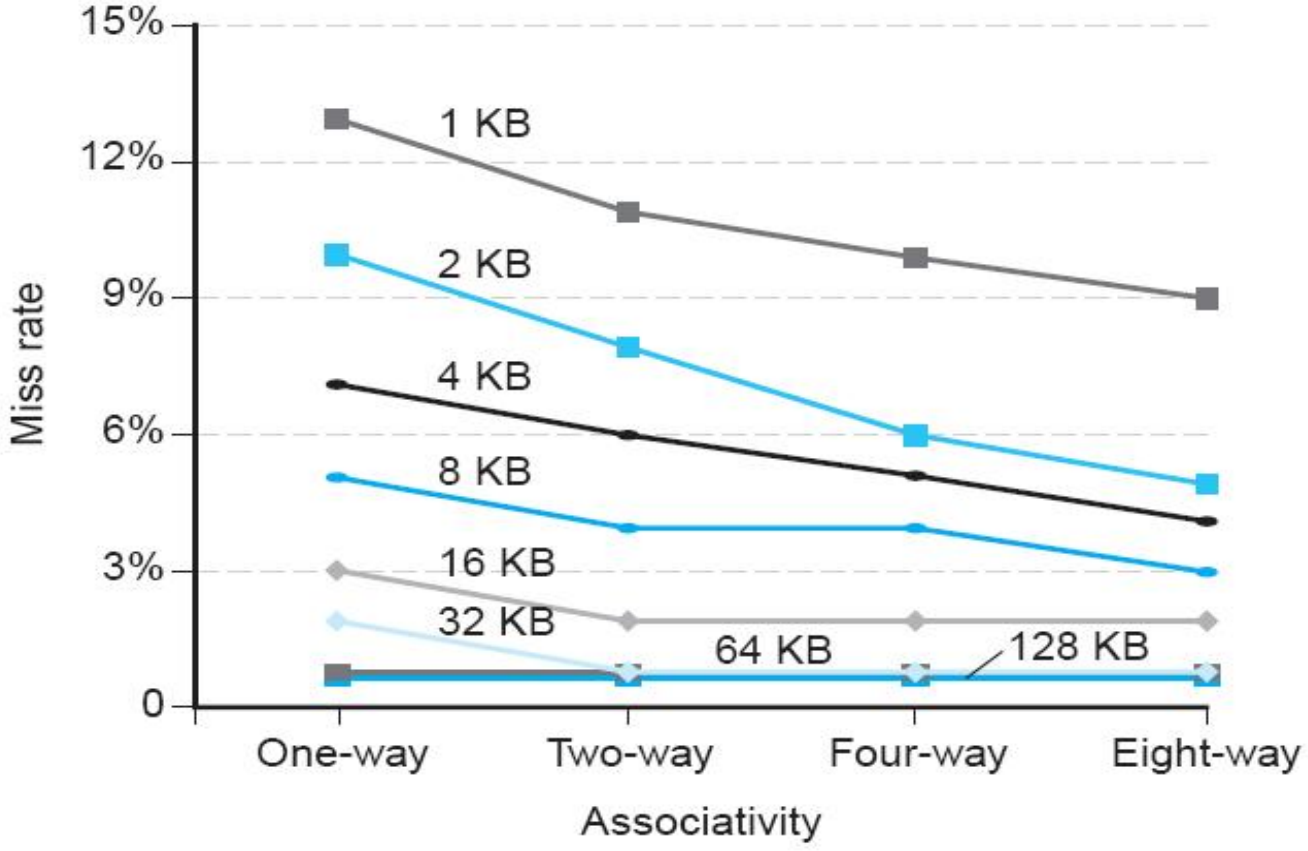
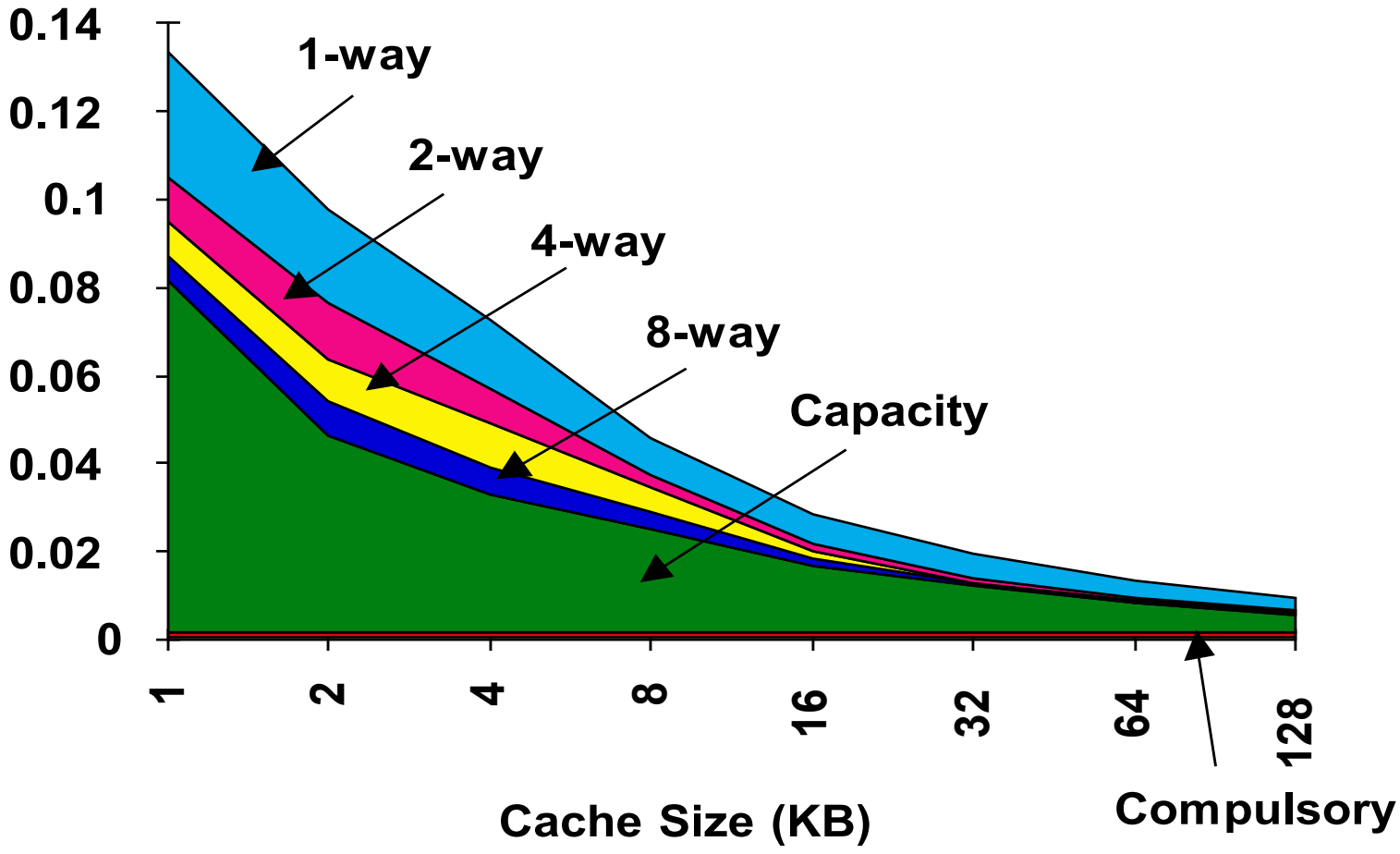


Figure 7.30

Miss Rate for SPEC92 Benchmarks



Locating a Block in the Cache

- **Set-associative mapping**
  - The three portions of an address: Figure 7.16

  - Index … used to select the set
  - Tags … searched in parallel to determine hit or miss
- **Increasing the associativity by a factor of two**
  - Doubling the number of blocks per set

    -> doubling the number of comparators
  - Halving the number of sets

    -> decrease the size of the index by 1 bit

    increase the size of the tag by 1 bit

4-Way Set Associative Cache

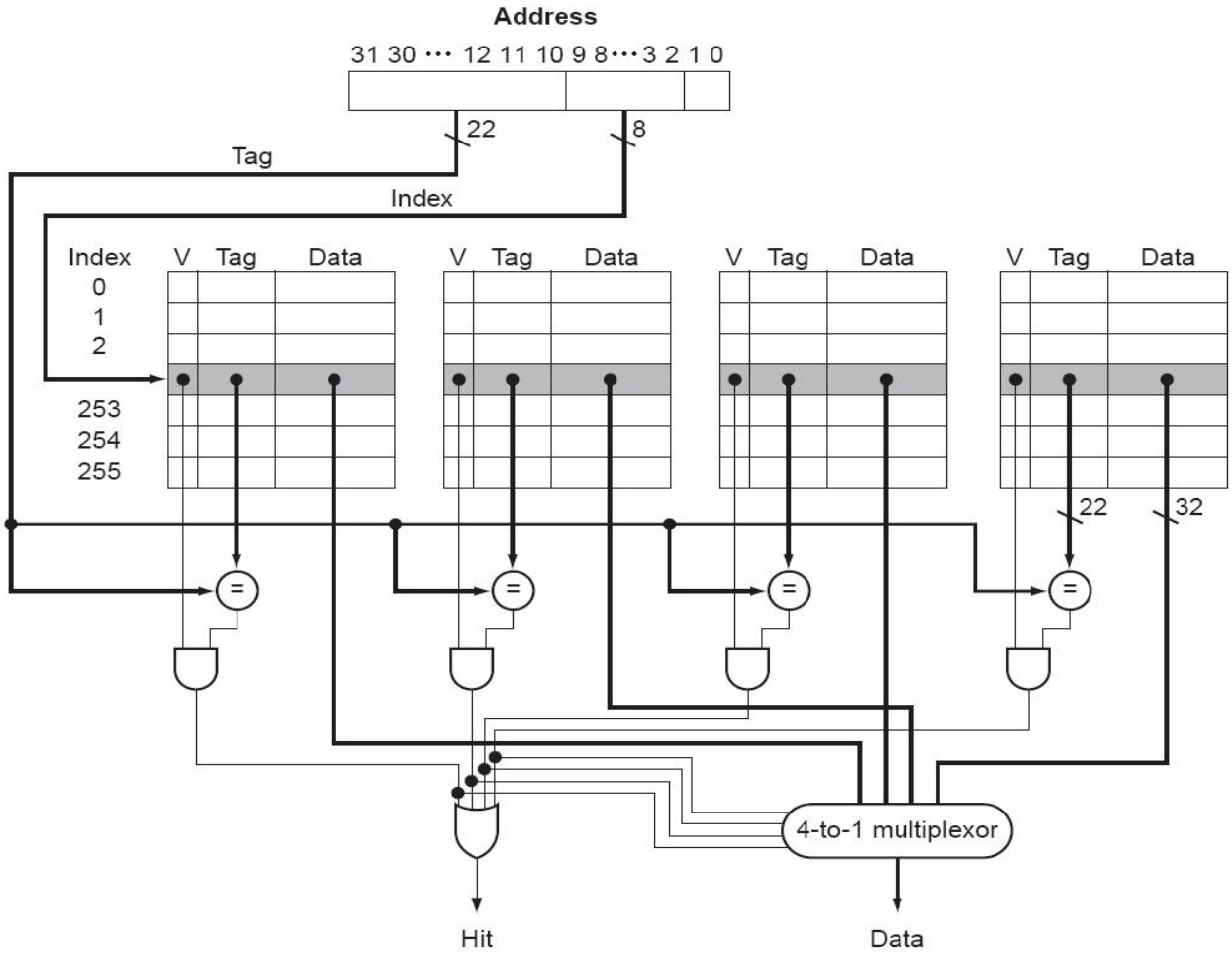


Figure 7.17

Example: Tag size vs. Associativity – 1

- 32-bit address, cache of 4K (=2^{12}) blocks
- Direct, 2-way, 4-way, fully associative

[Answer]

- Direct mapped
  4K sets -> 12-bit index
  tag = (32 – 12) x 4K = 80Kbits
- 2-way set associative
  2K sets -> 11-bit index
  tag = (32 – 11) x 2 x 2K = 84Kbits
- 4 -way set associative
  1K sets -> 10-bit index
  tag = (32 – 10) x 4 x 1K = 88Kbits
- Fully associative
  1 set, tag = 32 x 4K x 1 = 128Kbits

Example: Tag size vs. Associativity – 2

- Same as before, but 4-word block size

[Answer]

- 16 bytes per block -> 4-bit block offset
- Direct mapped
  4K sets -> 12-bit index
  tag = (28 – 12) x 4K = 64Kbits
- 2-way set associative
  2K sets -> 11-bit index
  tag = (28 – 11) x 2 x 2K = 68Kbits
- 4 -way set associative
  1K sets -> 10-bit index
  tag = (28 – 10) x 4 x 1K = 72Kbits
- Fully associative
  1 set, tag = 28 x 4K x 1 = 112Kbits

Choosing Which Block to Replace

- **Replacement algorithms**
  - Direct-mapped cache … uniquely specified
  - Set-associative cache … choice among the blocks in the set
  - Fully associative cache … choice among all the blocks
- **LRU (least recently used)**
  - Victim … the block that has been unused for the longest time
  - Implementation

    by keeping track of when each element was used
  - For 2-way set-associative cache

    1 reference bit
  - Higher degree of associativity

    Harder to implement

Reducing the Miss Penalty Using Multilevel Caches

- **Primary cache**
  - On-chip
- **Secondary cache**
  - Off-chip SRAMs
  - Reduce primary cache's miss penalty
- **Example : Performance of Multilevel Caches**
  - Clock rate = 5 GHz
  - CPI = 1.0 with a primary cache of 100% hit rate
  - Main memory access time = 100 ns (including miss handling)
  - Miss rate/instruction at the primary cache = 2%
  - **With a secondary cache below, how will it be faster?**

    miss rate = 0.5%, access time = 5 ns

[Answer]

- **With 1 level of cache**

  Miss penalty to main memory

  = 100ns/0.2ns = 500 clock cycles

  Total CPI

  = base CPI + memory-stall cycles per instruction

  = 1.0 + 2% x 500 = 11.0

- **With 2 levels of caches**

  Miss penalty to secondary cache

  = 5ns/0.2ns = 25 clock cycles

  Total CPI = 1 + primary stalls + secondary stalls

  = 1 + 2% x 25 + 0.5% x 500 = 4.0

- **Thus, speedup = 11.0/4.0 = 2.8.**

Design Considerations for a Primary and Secondary Cache

- **Primary caches**
  - Focus on minimizing hit time -> shorter clock cycles
  - Smaller → higher miss rate
  - Smaller block size → reduced miss penalty
- **Secondary caches**
  - Focus on reducing miss rate → to reduce the penalty of long memory access times
  - Larger → Access time is less critical.
  - Larger block size ← affecting miss penalty only

Understanding Program Performance

- **Impact of the memory hierarchy on performance**
  - A. LaMarca and R.E. Ladner, "The Influence of Caches on the Performance of Sorting," *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, Jan. 1997, pp.370-379.
  - Quicksort vs. Radix Sort
  - Alphastation 250
    - 2MB L2 cache
    - 32 byte blocks
    - Direct mapped
  - 8 byte keys
    - From 4000 to 4000000

Instructions per Item

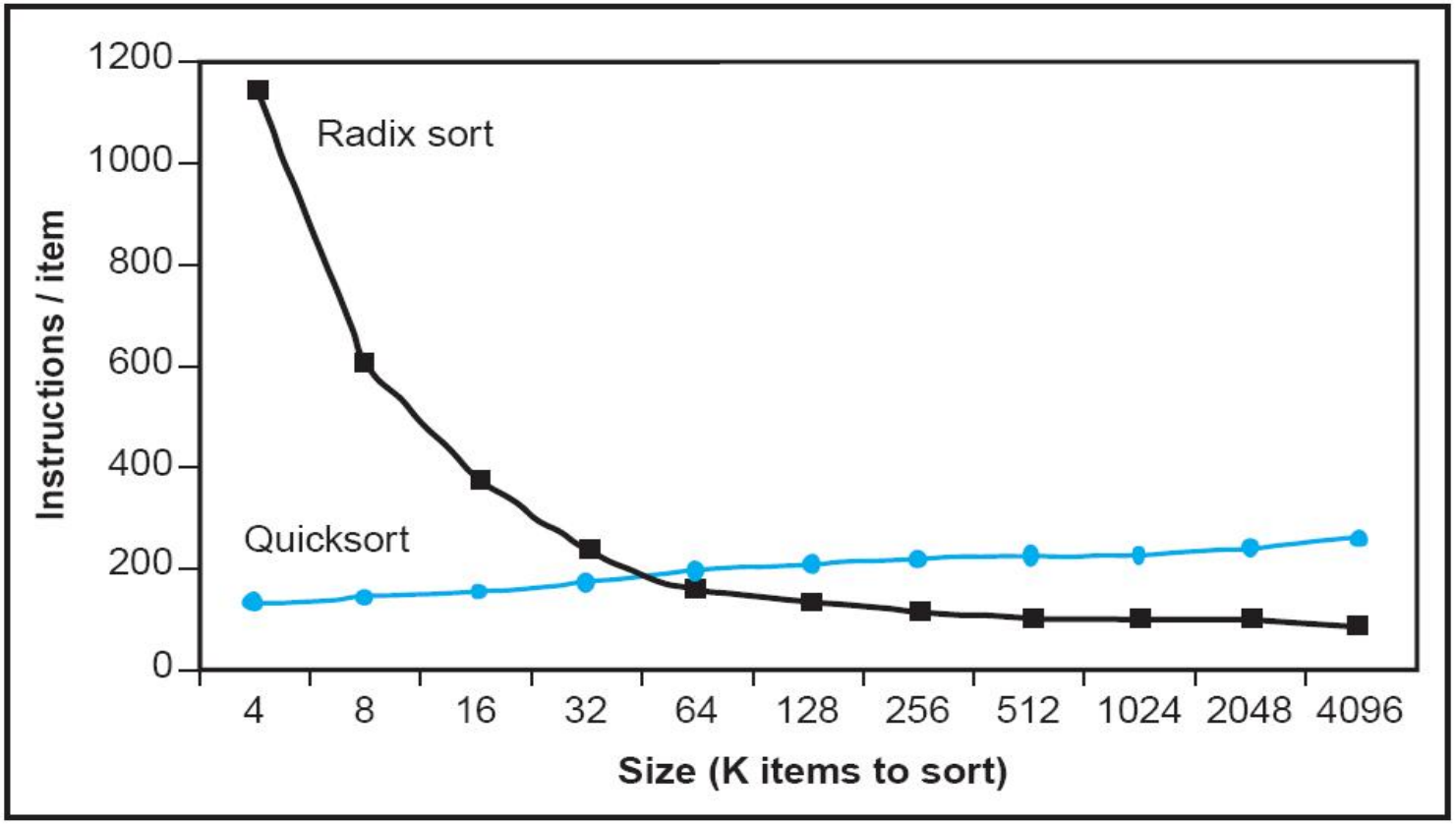


Figure 7.18a

Clock Cycles per Item

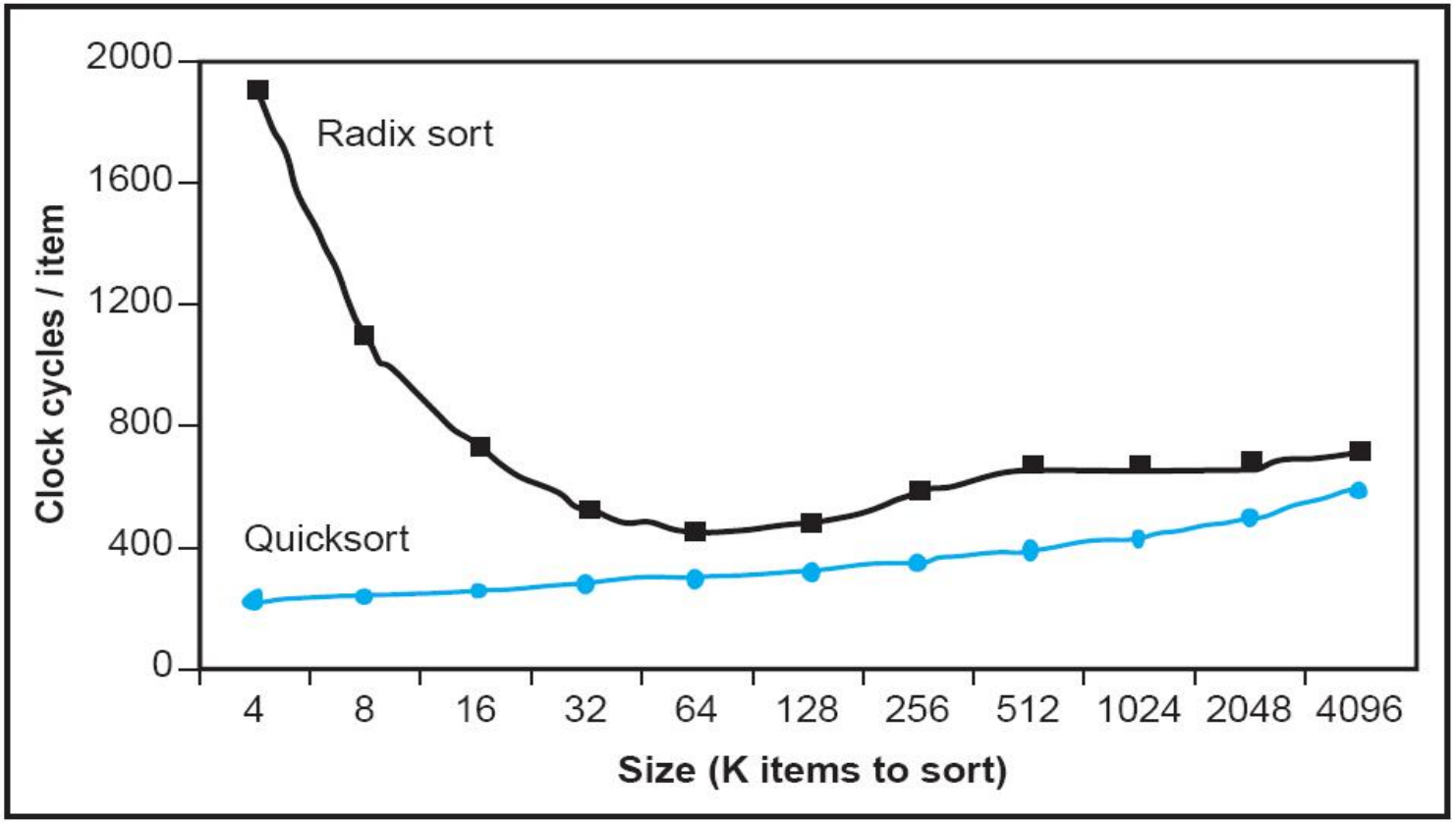


Figure 7.18b

Cache Misses per Item

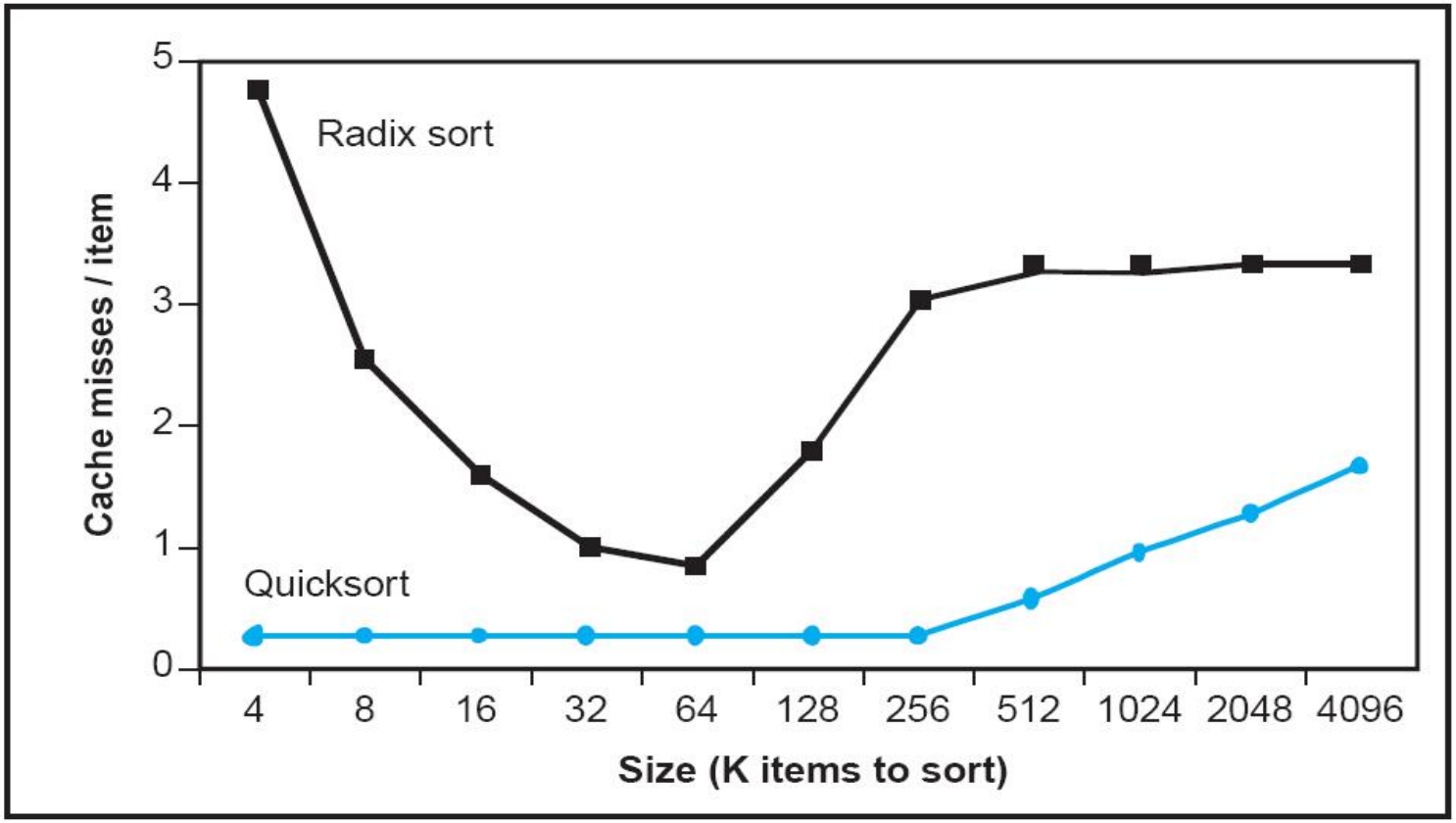


Figure 7.18c

Caches in Real Products

Processor	Type	Year of Introduction	L1 cache[a]	L2 cache	L3 cache
IBM 360/85	Mainframe	1968	16 to 32 KB	—	—
PDP-11/70	Minicomputer	1975	1 KB	—	—
VAX 11/780	Minicomputer	1978	16 KB	—	—
IBM 3033	Mainframe	1978	64 KB	—	—
IBM 3090	Mainframe	1985	128 to 256 KB	—	—
Intel 80486	PC	1989	8 KB	—	—
Pentium	PC	1993	8 KB/8 KB	256 to 512 KB	—
PowerPC 601	PC	1993	32 KB	—	—
PowerPC 620	PC	1996	32 KB/32 KB	—	—
PowerPC G4	PC/server	1999	32 KB/32 KB	256 KB to 1 MB	2 MB
IBM S/390 G4	Mainframe	1997	32 KB	256 KB	2 MB
IBM S/390 G6	Mainframe	1999	256 KB	8 MB	—
Pentium 4	PC/server	2000	8 KB/8 KB	256 KB	—
IBM SP	High-end server/ supercomputer	2000	64 KB/32 KB	8 MB	—
CRAY MTA[b]	Supercomputer	2000	8 KB	2 MB	—
Itanium	PC/server	2001	16 KB/16 KB	96 KB	4 MB
SGI Origin 2001	High-end server	2001	32 KB/32 KB	4 MB	—

[a] Two values seperated by a slash refer to instruction and data caches

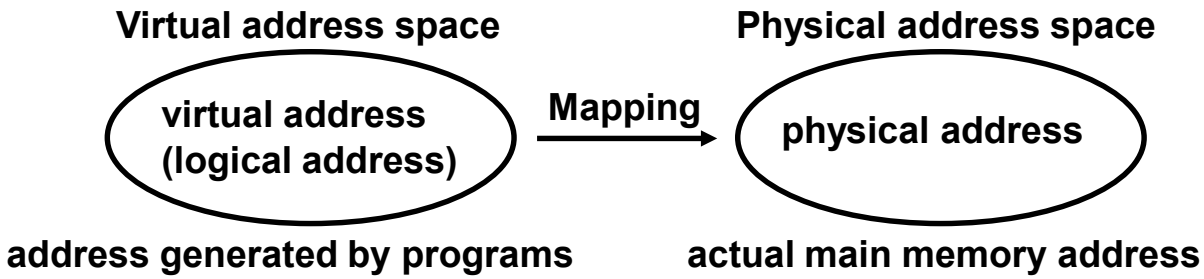
[b] Both caches are instruction only; no data caches

5.4 Virtual Memory

- **Virtual memory**
  - Use main memory as a "cache" for the secondary storage
- **2 major motivations**
  - To remove the programming burdens of a small, limited amount of main memory
  - To allow efficient sharing of memory among multiple programs
- **Goal of virtual memory**
  - Give the programmer the illusion that the system has a very large memory, even though the computer actually has a relatively small main memory

Address Space

- **Virtual address space and physical address space**



- **Virtual addresses**
  - Used by the processor (program, user)
- **Physical addresses**
  - Used to access main memory
- **Page and page frame**
  - Virtual and physical memory block
- **Page fault**
  - Virtual memory miss

Memory Mapping (or Address Translation)

- Change a virtual page number to a physical page number
- Performed by a combination of hardware and software

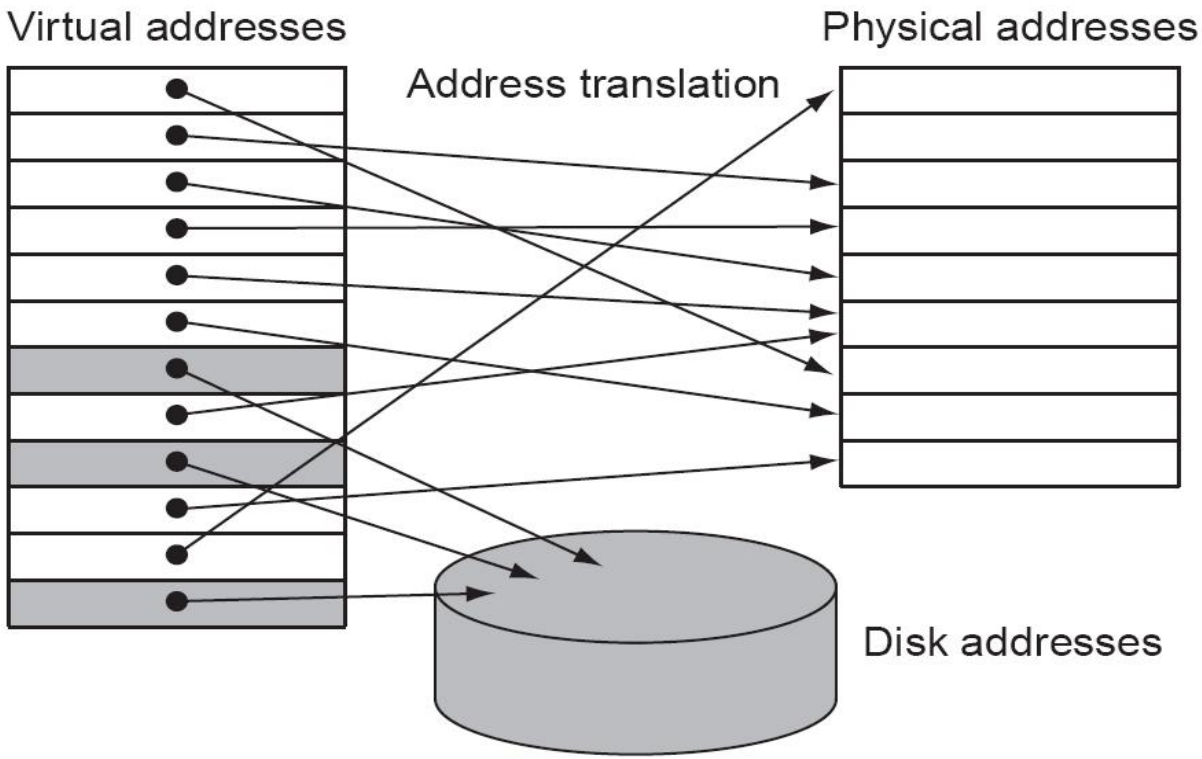


Figure 7.19

Relocation

- Advantage of virtual memory
- Simplifies program loading for execution
- Noncontiguous allocation
- Only to find a sufficient number of pages in main memory

Address in Virtual Memory System

- **Address = virtual page number || page offset**

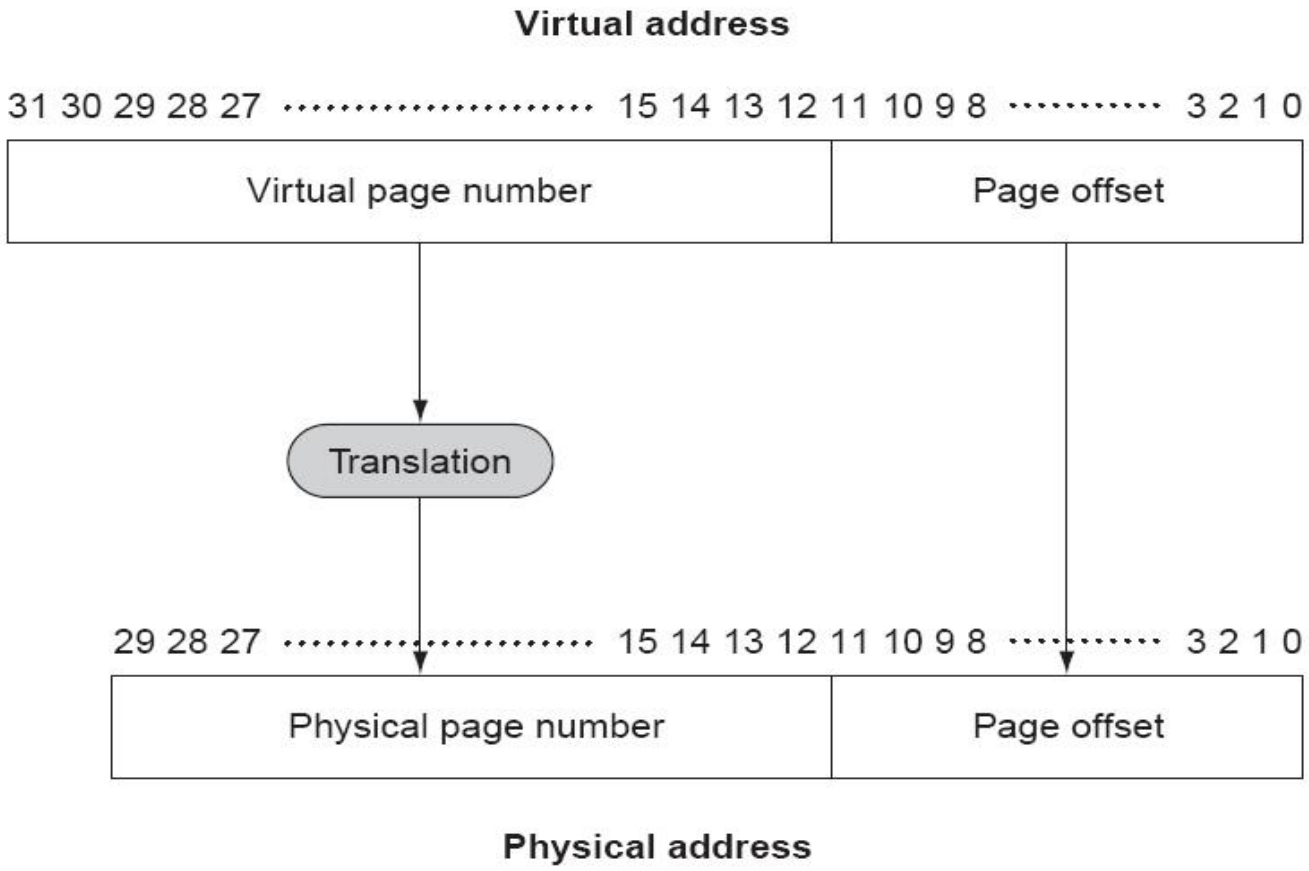


Figure 7.20

Design Considerations

- **Very large miss penalty (millions of cycles)**

1. Large page size

- Should be large enough to amortize the high disk access time
- Typically 4KB~16KB, recently 32KB or 64KB
- 1KB pages in the new embedded systems

2. Reducing page fault rate

- Fully associative placement of pages

3. Using clever software algorithms

- Because the overhead is small compared to disk access time

4. Write-back

- Too long a disk write time

Elaboration

- **Paging … fixed-size blocks**
- **Segmentation … variable-size blocks**
  - Address = (segment number, offset)
  - Addition rather than concatenation
  - Supporting more powerful methods of protection and sharing in an address space
  - Splitting address space into logically separate pieces

Placing a Page and Finding It Again

- **Fully associative mapping**
  - Reducing the page fault rate by optimizing the page placement
  - A virtual page can be mapped to any physical page.
- **Page table**
  - A full table that indexes the memory
  - Contains the virtual to physical address translation
  - Indexed with the virtual page number
  - Resides in memory
  - Each process has its own page table.
  - Table entry
    - Physical page number
    - Valid bit (or residence bit or presence bit)
  - Page table register
    - Starting address of the page table

Address Translation Hardware

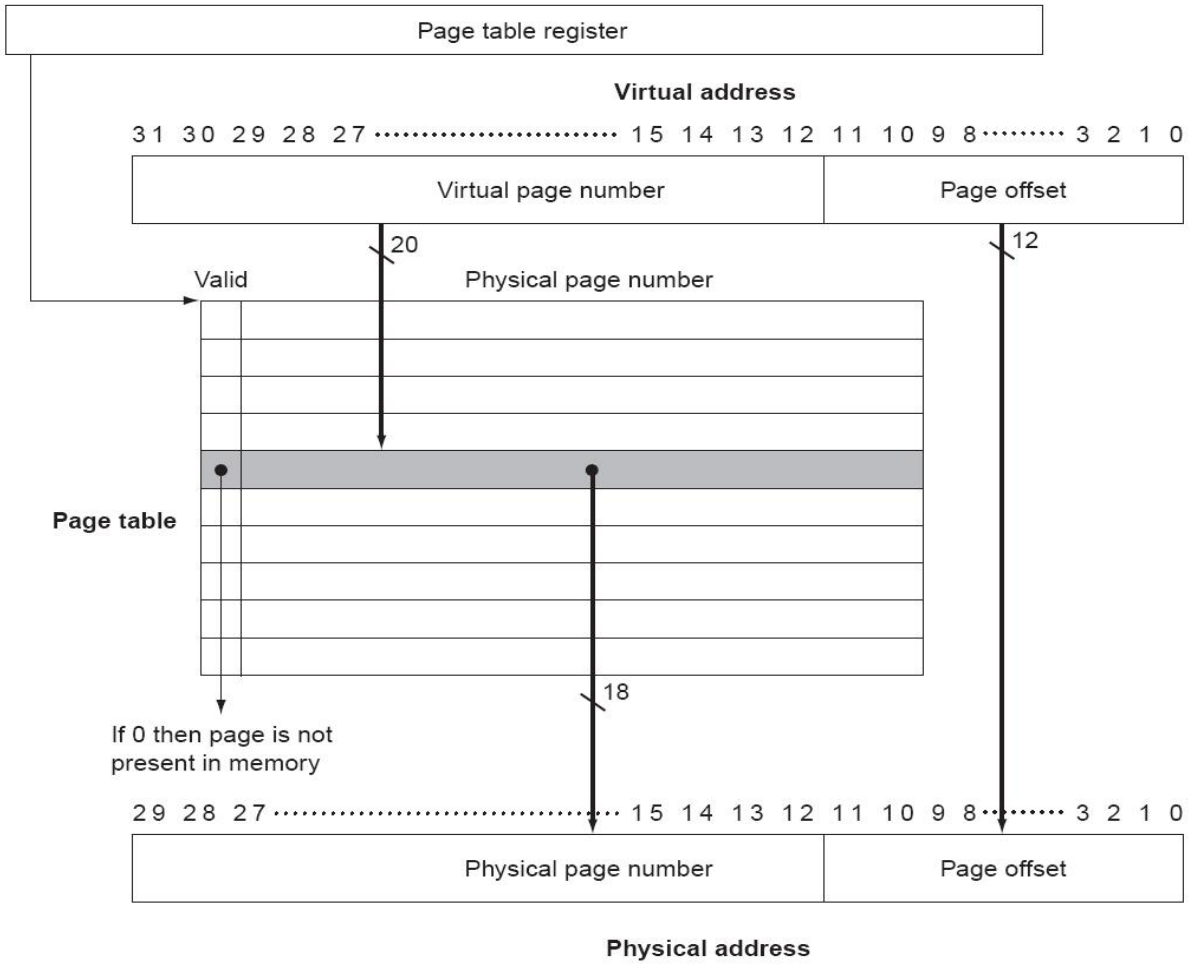
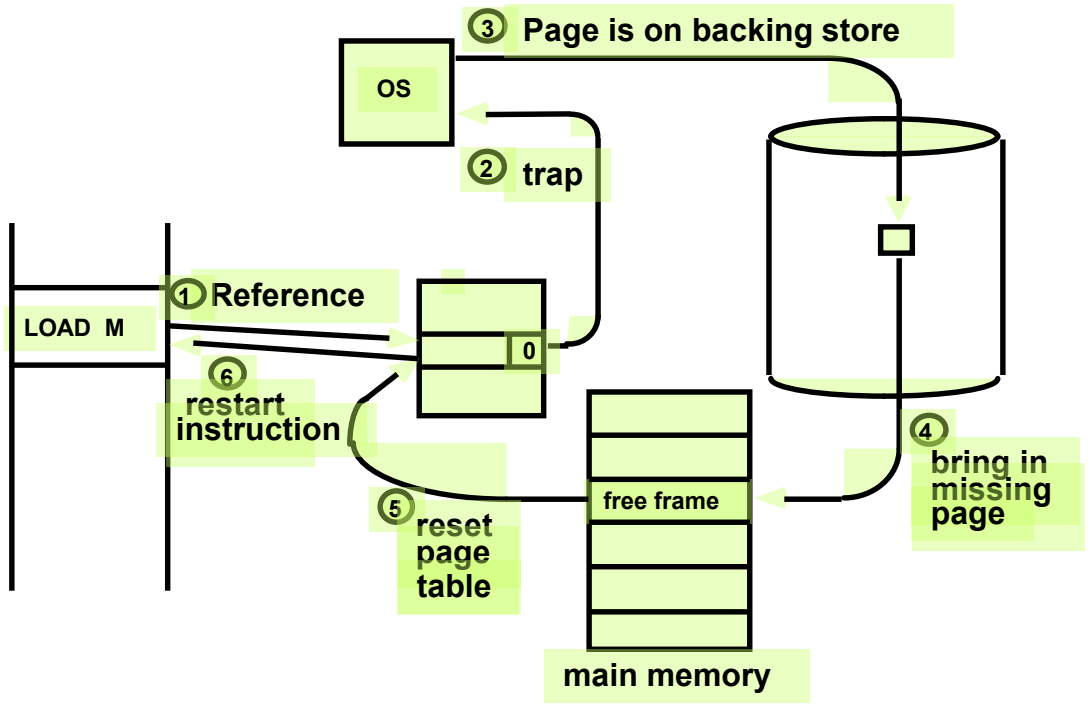


Figure 7.21

Page Faults

- **Page fault**
  - Occurs when the valid bit for a virtual page is off
  - OS must be given control through exception mechanism.
  - Find the page in the next level and decide where to place the requested page in memory



Processor architecture should provide the ability to restart any instruction after a page fault.

Handling a Page Fault

1. Trap to the OS by page fault exception
2. Save the user registers and program state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Choose a victim using page replacement algorithm
    a. Writeback to disk if dirty block
6. Issue a read from disk to the free frame
    a. Wait in a queue for this device until serviced
    b. Wait for the device seek and/or latency time
    c. Begin the transfer of the page to the free frame
7. While waiting, the CPU may be allocated to some other process
8. Interrupt from disk (I/O completed)
9. Save the registers and program state for the other user
10. Determine that the interrupt was from disk
11. Correct the page tables (the desired page is now in memory)
12. Wait for the CPU to be allocated to this process again
13. Restore the user registers, program state, and new page table, then resume the interrupted instruction.

Example: Virtual Memory Performance

- **Memory access time: 100 ns**
- **Disk access time: 25 ms**
- **Effective access time**
  - Let p = the probability of a page fault
  - Effective access time = 100(1-p) + 25,000,000p
  - If we want only 10% degradation
    - 110 > 100 + 25,000,000p
    - 10 > 25,000,000p
    - p < 0.0000004 (one fault every 2,500,000 references)
- **Lesson: OS had better do a good job of page replacement!**

Replacement Algorithms

- **LRU**
- **LRU approximation**
  - implemented using use bit (or reference bit)
  - set whenever the page is accessed and reset periodically

Page Table with Disk Addresses

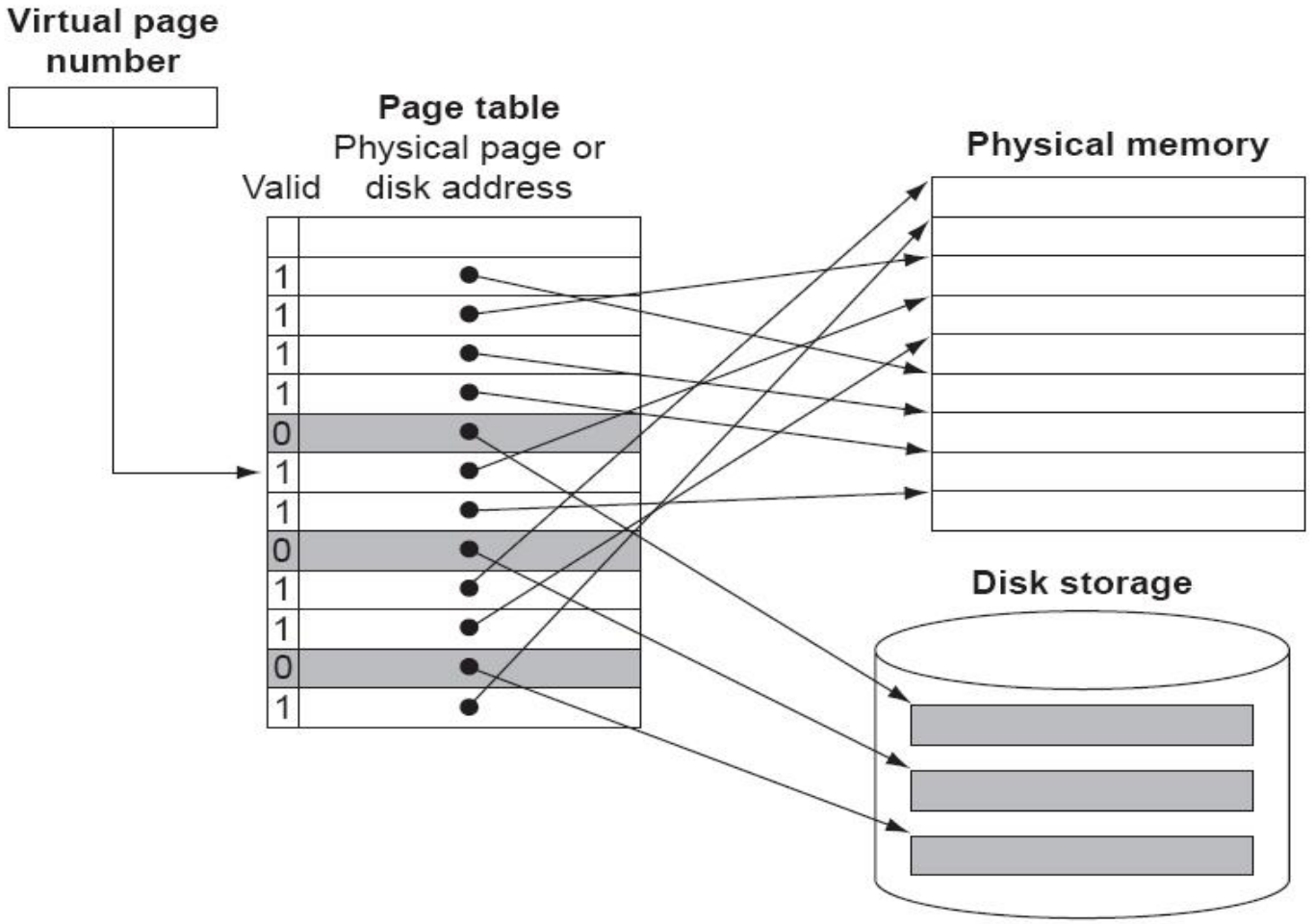


Figure 7.22

Elaboration

- **page table size for 32-bit virtual address, 4 KB pages, 4 bytes per page table entry**

$$\frac{2^{32}}{2^{12}}\ \text{pages} \times 4\ \frac{\text{bytes}}{\text{page}} = 4\ \text{MB (for each program in execution)}$$

- **5 techniques for reducing page table size**
  1. To keep a bound register that limits the size of page table for a given process
  2. Two page tables … one for heap and one for stack
  3. Inverted page table
  4. Multilevel page tables
  5. Paging of the page tables

What About Writes ?

- **Write-through in virtual memory**
  - Millions of processor clock cycles for a disk write
  - impractical
- **Write-back**
  - Performing write into the page in memory
  - Copying the page back to disk when it is replaced
  - Dirty bit (=Modified bit)
    - added to page table entry

Translation-Lookaside Buffer

- **Double memory accesses in virtual memory**
  (1) to obtain the physical address
  (2) to get the data
- **TLB (translation-lookaside buffer)**
  - Using locality of reference to the page table
  - Special address translation cache that keeps track of recently used translation
- **Typical values for a TLB**

TLB size	**16 ~ 512 entries**
block size	**1~2 page table entries (typically 4~8 bytes each)**
hit time	**0.5 ~ 1 clock cycle**
miss penalty	**10 ~ 100 clock cycles**
miss rate	**0.01 ~ 1%**

Address Translation with TLB

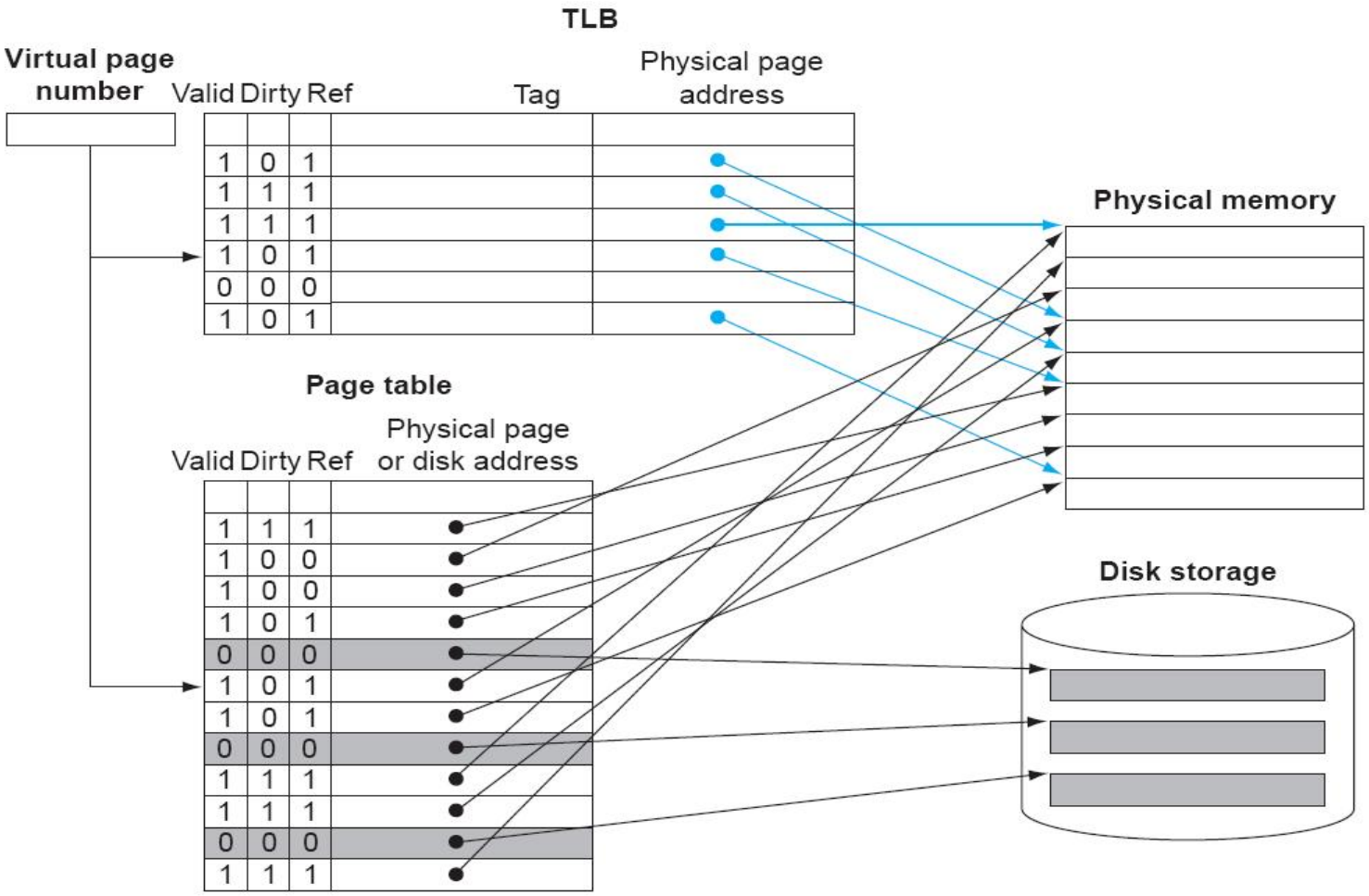


Figure 7.23

The Intrinsity FastMATH TLB

- **Memory system**
  - Page size = 4 KB
  - Virtual address = physical address = 32 bits
  - Virtual page number = physical page number = 20 bits
- **TLB**
  - Shared between instruction and data references
  - 16 entries, 64 bits/entry
  - Fully associative mapping
  - Entry = 20-bit tag + 20-bit physical page number
    + valid bit + dirty bit + other bookkeeping bits

TLB and Cache in Intrinsity FastMATH

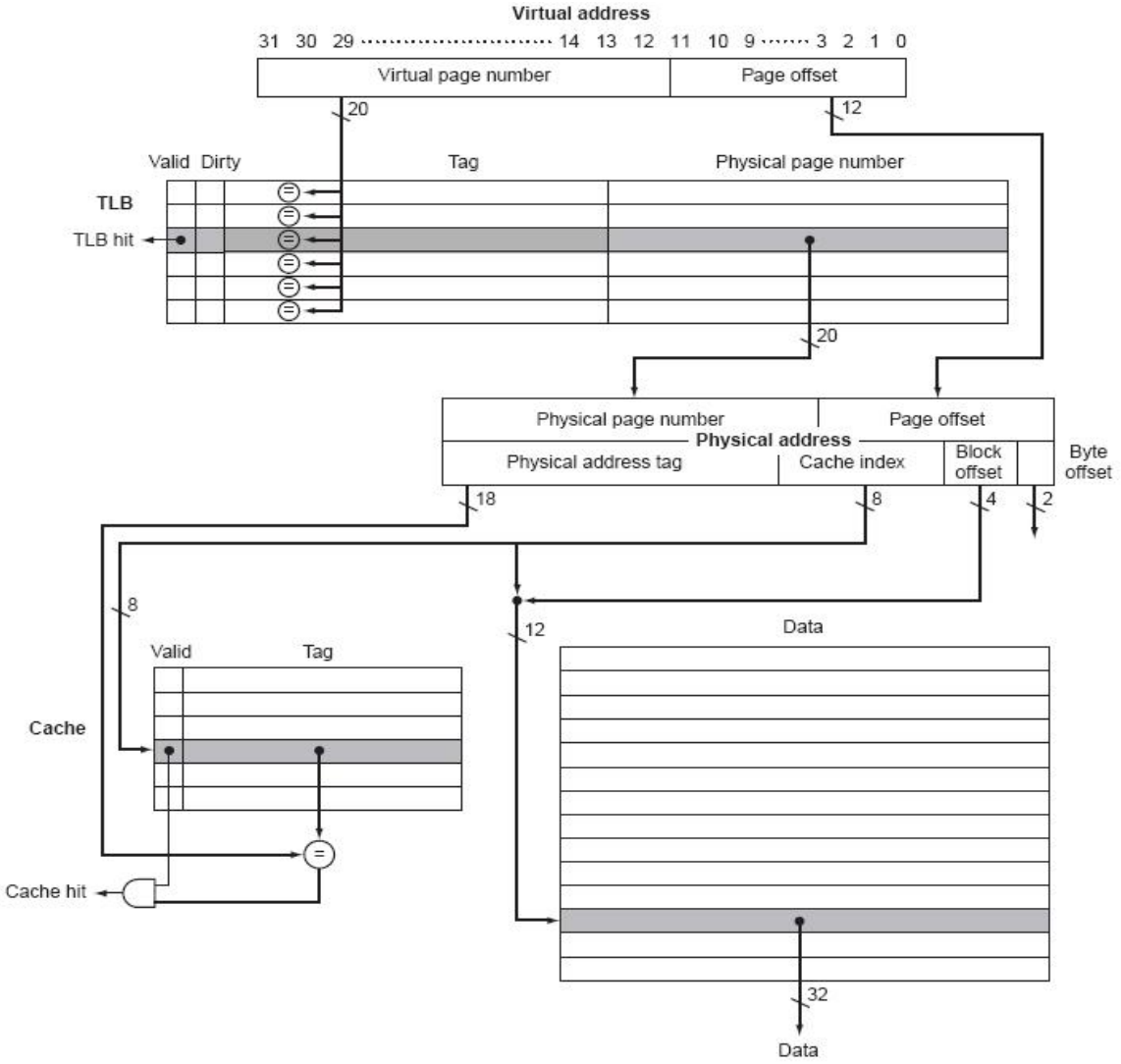


Figure 7.24

Processing a Read or Write Through in the Intrinsity FastMATH

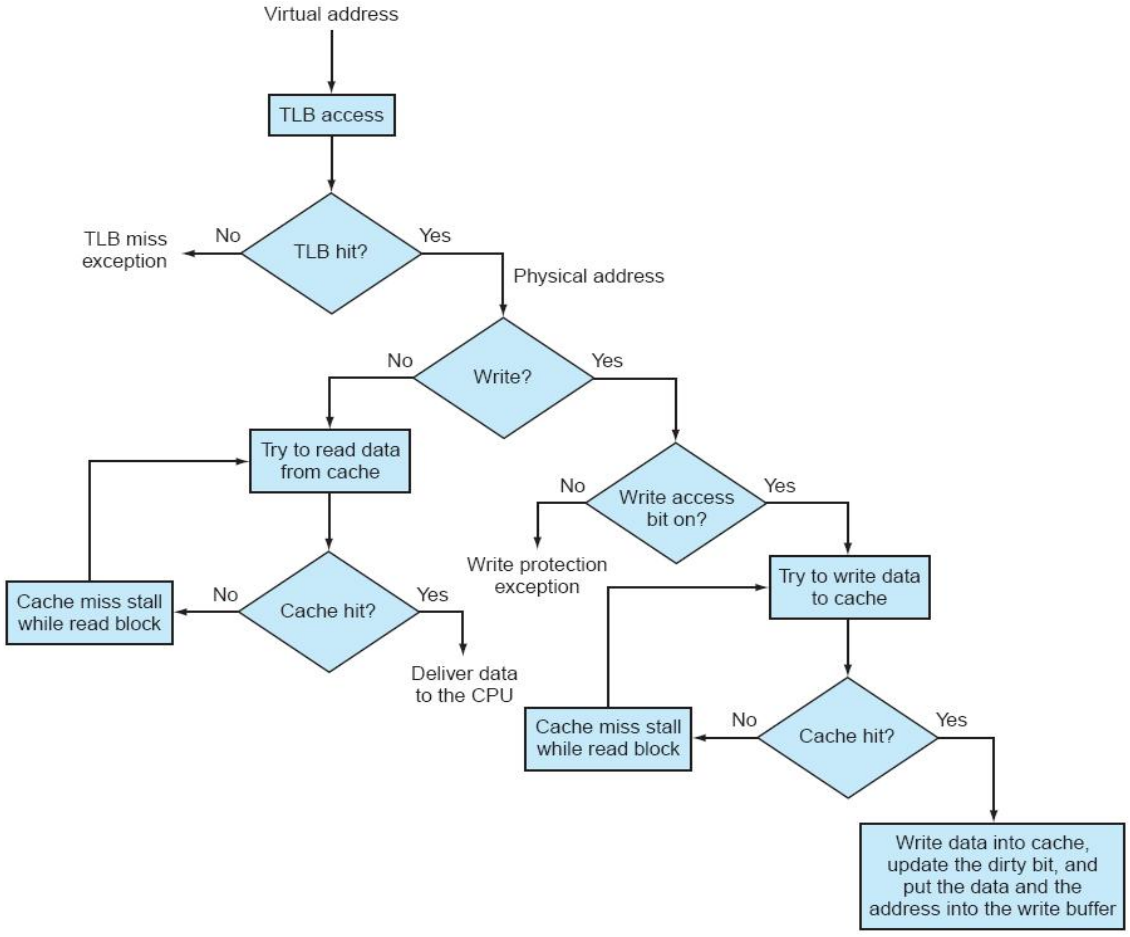
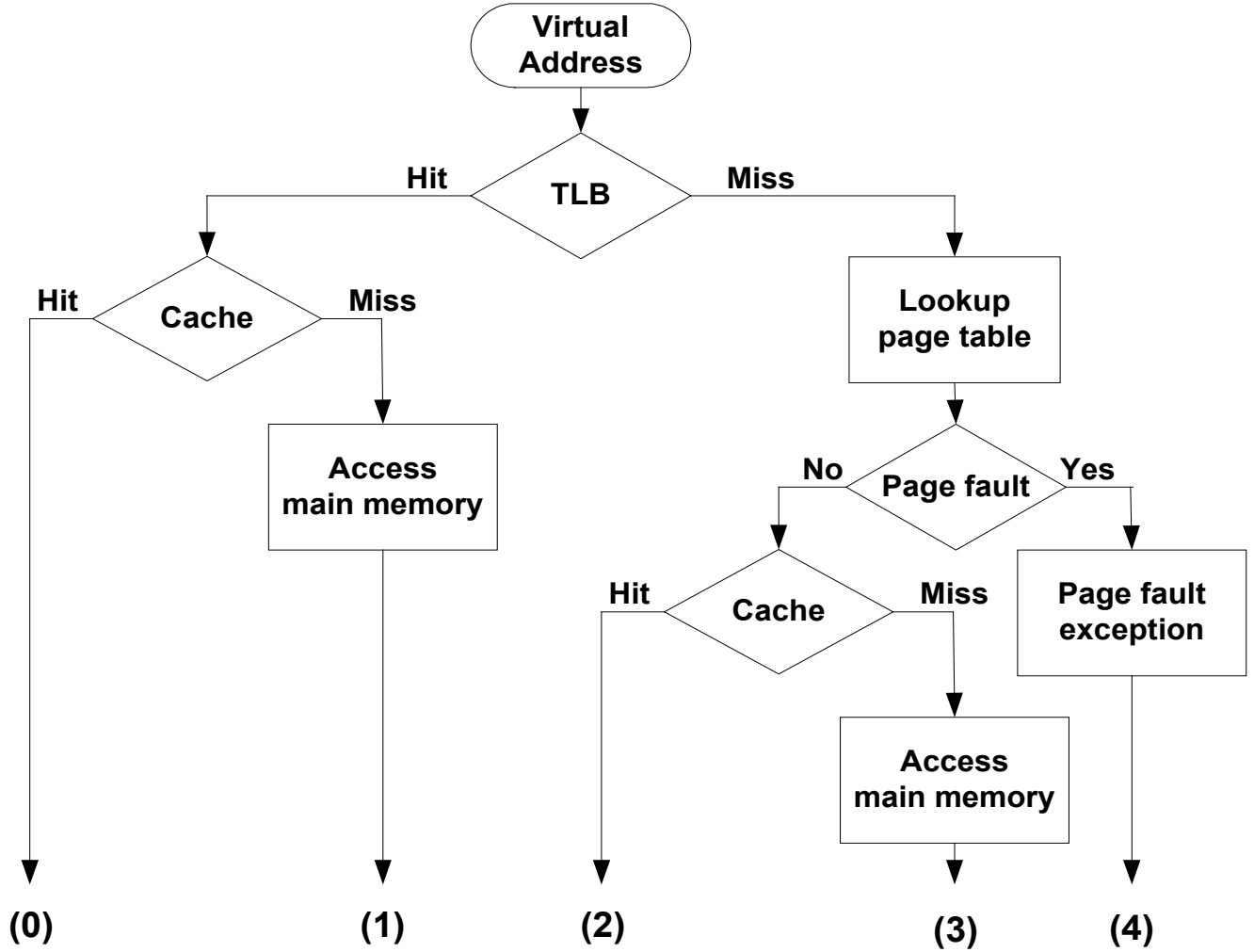


Figure 7.25

TLB, Cache and Virtual Memory



TLB Miss

- **Special register <- the page number of the reference**
- **Generates an exception**
  - Invoking OS
  - Finding the physical address for the missing page (by indexing the page table)
  - Updating TLB with a special set of system instructions
- **TLB miss penalty**
  - About 13 cycles, assuming the code and the page table entry are in the caches
- **True page fault if not in the page table**

Write Requests

- Check the write access bit in TLB
- If the write access bit is off, an exception is generated.

Integrating Virtual Memory, TLBs, and Caches

- **Example: Overall Operation of a Memory Hierarchy**
  - Possible combinations of cache miss, TLB miss and page fault

	cache	TLB	virtual memory	
(0)	hit	hit	hit	Best case
(1)	miss	hit	hit	Possible, no page table access
(2)	hit	miss	hit	TLB miss
(3)	miss	miss	hit	TLB miss and cache miss
(4)	miss	miss	miss	Page fault
	miss	hit	miss	Impossible
	hit	hit	miss	Impossible
	hit	miss	miss	Impossible

Elaboration (1)

- **Physically indexed and physically tagged cache**
  - Figure 7.25
  - Cache hit time = TLB access time + cache access time
  - VAX 8600, Intel 80486, MIPS R3000
- **Virtually indexed and virtually tagged cache**

  (= Virtually addressed cache = Virtual cache)
  - Address translation hardware is unused during normal cache access.
  - Aliasing (or synonyms) problem
  - Intel i860, SUN 3/200 series

Elaboration (2)

- **Virtually indexed and physically tagged cache**
  - Cache index $\subset$ page offset
  - Parallel accesses of cache and TLB
  - Physical address tags
  - Performance advantages of a virtually indexed cache
  - Architecturally simpler advantages of a physically addressed cache

Implementing Protection with Virtual Memory

- **Protection and page table**
  - Mapping each process' virtual address space to disjoint physical pages
    - cannot access other process' data
  - Preventing user process from table modification
  - Page table : in the protected address space of OS
- **Write protection bit**
  - Included in each page table entry
  - Checked on every memory access

Hardware/Software Interface

- **Hardware's 3 basic capabilities for the OS to implement protection**

  (1) 2 modes

  - user process
  - OS process (= kernel, supervisor or executive process)

  (2) A CPU state that a user process can read but not write

  (3) A mechanisms whereby the CPU can go from user mode to supervisor mode, and vice versa

  - system call exception (`syscall` instruction)
  - `ERET` (return from exception) instruction

Elaboration

- **Context switch or process switch from P1 to P2**
  - It must ensure that P2 cannot get access to the page tables of P1.
  - Without TLB, it suffices to change the page table register.
  - With TLB, TLB entries of P1 must be cleaned.
- **Process identifier (PID) or task identifier**
  - ID of currently running process
  - Concatenated to the tag portion of the TLB
  - Intrinsity FastMATH
    - 8-bit address space ID (ASID)
- **Similar problems in a cache**
  - Various solution such as PID

Handling TLB Misses and Page Faults

- **TLB misses**
  1) resident page → create the missing TLB entry
  2) non-resident page → page fault exception
- **TLB miss or page fault**
  - Handled with exception mechanism

Hardware/Software Interface

- **Exception disable**
  - When an exception occurs, disable all other exceptions.
  - To protect EPC and Cause registers from being destroyed

MIPS Control Registers

- **In coprocessor 0**

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

Figure 7.27

Handling Page Faults

0. Find out the virtual address that caused the page fault.
1. Look up the page table to find out the location on disk.
2. Choose a victim. If it is dirty, write back to disk.
3. Read the page from disk.
   - Processor executes another process.
4. OS restores the state of the process.
5. Execute `ERET` (return from exception) instruction.
   - Restore PC.
   - Reset processor from kernel to user mode.
6. Reexecute the instruction that faulted.
   - Access the requested page.

Page Fault Exceptions for Data Accesses

- **Why difficult to implement properly in a processor**
  1. They occur in the middle of the instructions.
  2. The instruction cannot be completed before handling the exception.
  3. After handling the exception, the instruction must be restarted as if nothing had occurred.
- **Restartable instruction**
  - An instruction that can resume execution after an exception is resolved without the exception's affecting the result of the instruction
  - Easy to implement in MIPS architecture
    - Each instruction writes only one data item.
    - Write occurs at the end of the instruction cycle.

    → Simply prevent the instruction from completing (by not writing) and restart the instruction at the beginning

Handling TLB Misses in MIPS

1. MIPS hardware saves the page number of the reference in **BadVAddr** register and generates an exception
2. Transfer control to TLB miss handler. ($8000\ 0000_{hex}$)
3. Find out physical address for the missing page.

- TLB[`Random`] <- M[`Context`] ; about 12 clock cycles

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

Handling Page Faults in MIPS

1. Transfer control to general exception handler. ($8000\ 0180_{hex}$)
2. Save the entire state of the process.
3. Find out the faulting virtual address.

Save State

```
Save GPR    addi $k1,$sp,-XCPSIZE # save space on stack for state
            sw   $sp,XCT_SP($k1)  # save $sp on stack
            sw   $v0,XCT_V0($k1)  # save $v0 on stack
            ...                   # save $v1, $ai, $si, $ti, ...on stack
            sw   $ra,XCT_RA($k1)  # save $ra on stack
Save        mfhi $v0              # copy Hi
Hi, Lo      mflo $v1              # copy Lo
            sw   $v0,XCT_HI($k1)  # save Hi value on stack
            sw   $v1,XCT_LI($k1)  # save Lo value on stack
Save        mfc0 $a0,$cr          # copy cause register
Exception   sw   $a0,XCT_CR($k1)  # save $cr value on stack
Registers   ...                   # save $v1, ....
            mfc0 $a3,$sr          # copy Status Register
            sw   $a3,XCT_SR($k1)  # save $sr on stack
Set sp      move $sp,$k1          # sp = sp - XCPSIZE
```

Call Exception Handler

Enable nested exceptions		
	`andi $v0,$a3,MASK1`	
		`# $v0 = $sr & MASK1, enable exceptions`
	`mtc0 $v0,$sr`	`# $sr = value that enables exceptions`
Call C exception handler		
`Set $gp`	`move $gp,GPINIT`	`# set $gp to point to heap area`
`Call C code`	`move $a0,$sp`	`# arg1 = pointer to exception stack`
	`jal xcpt_deliver`	`# call C code to handle exception`

Restoring State

```
Restore     move $at,$sp            # temporary value of $sp
most GPR,   lw   $ra,XCT_RA($at)    # restore $ra from stack
Hi, Lo      ...                     # restore $t0, ...., $a1
            lw   $a0,XCT_A0($k1)    # restore $a0 from stack
```

```
Restore     lw   $v0,XCT_SR($at)    # load old $sr from stack
Status      li   $v1,MASK2          # mask to disable exceptions
Register    and  $v0,$v0,$v1
                                    # $v0 = $sr & MASK2, disenable exceptions
            mtc0 $v0,$sr            # set Status Register
```

Exception Return

```
Restore     lw   $sp,XCT_SP($at)   # restore $sp from stack
$sp and     lw   $v0,XCT_V0($at)   # restore $v0 from stack
rest of     lw   $v1,XCT_V1($at)   # restore $v1 from stack
GPR used    lw   $k1,XCT_EPC($at) # copy old $epc from stack
as          lw   $at,XCT_AT($at)   # restore $at from stack
temporary
registers
```

```
Restore     mtc0 $k1,$epc          # restore $epc
ERC and     eret $ra              # return to interrupted instruction
return
```

Summary

- **Techniques for reducing page fault rate**
  (1) large page → exploiting spatial locality
  (2) fully associative mapping
  (3) LRU and a reference bit
- **Techniques for reducing disk writes**
  - write-back and dirty-bit
- **Memory protection**
  - restricting page table access
- **TLB**
  - reducing address translation time

5.5 A Common Framework for Memory Hierarchies

- Key design parameters for 4 major memory hierarchies

Features	Typical values for L1 caches	Typical values for L2 caches	Typical values for paged memory	Typical values for a TLB
Size (blocks)	250-2000	4000-250,000	16,000-250,000	16-512
Size (KB)	16-64	500-8000	250,000-1,000,000,000	0.25-16
Block size (B)	32-64	32-128	4000-64,000	4-32
Miss penalty (clocks)	10-25	100-1000	10,000,000-100,000,000	10-1000
Miss rates	2-5%	0.1-2%	10^{-5}-10^{-4}%	0.01-2%

Figure 7.29

Back to chapter overview

Q1 : Where Can a Block Be Placed?

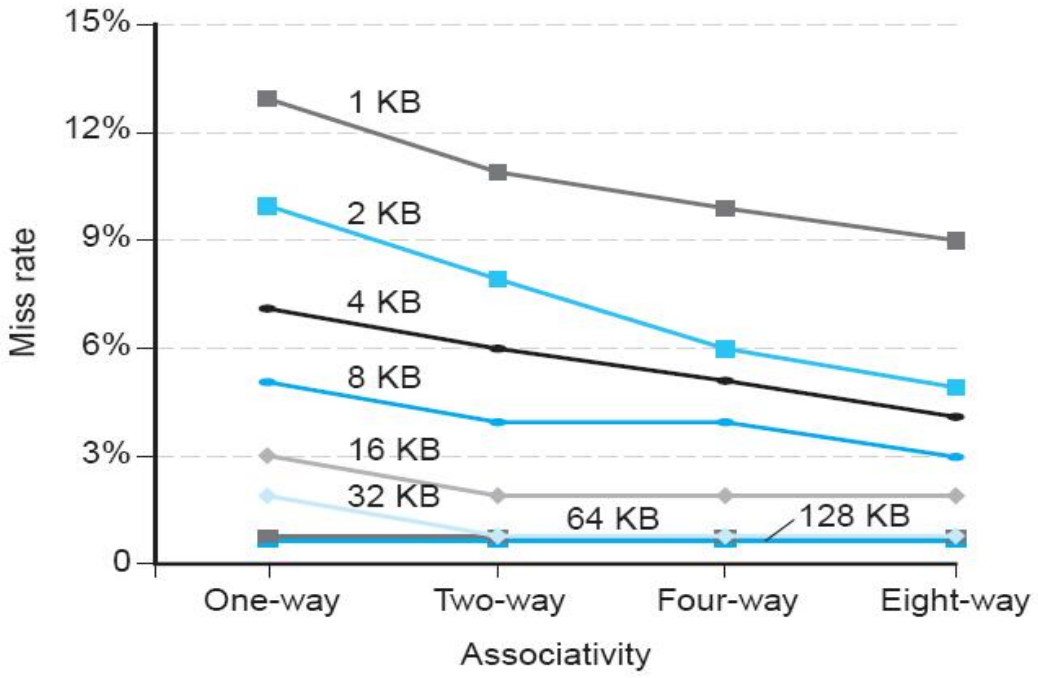
Scheme name	Number of sets	Blocks per set
Direct mapped	Number of blocks in cache	1
Set associative	Number of blocks in cache /Associativity	Associativity (typically 2~16)
Fully associative	1	Number of blocks in cache

- **n-way set associative cache**
  - Each block can be placed in one of n locations.
  - n = degree of associativity
  - Set size = n blocks
  - Number of sets = cache size / n
  - Each block in the memory maps to a unique set in the cache given by the index field, and a block can be placed in any element of that set.

Performance vs. Associativity

- **Increased associativity**
  - Decreased miss rate
  - Increased cost and slower access time
- **Miss rate vs. associativity**
  - Direct -> 2-way:
    20~30% reduction
  - Larger cache
    -> Low miss rate
    -> Small improvement

Figure 7.30



Q2 : How Is a Block Found?

Associativity	Location method	Comparisons required
Direct mapped	index	1
Set associative	index the set search among elements	degree of associativity
Fully associative	search all cache entries	size of the cache
	separate lookup table	0

- **Choice among the 3 mapping schemes**
  - the cost of a miss
  - the cost of implementing associativity
- **Direct-mapped caches**
  - Fast access time
  - simplicity

Fully Associativity with Extra Index Table in Virtual Memory

[disadvantages]

1) Extra storage for the table
2) Extra memory access

[motivations]

1) Expensive misses
2) Sophisticated replacement schemes
3) Easy indexing with no extra hardware and searching
4) Relatively small overhead (because of large page size)

Q3 : Which Block Should Be Replaced on a Cache Miss?

- **Candidates for replacement**
  - direct mapping … only one block
  - set associative mapping … blocks in the set
  - fully associative mapping … all the blocks
- **2 primary replacement strategies**

  (1) LRU

  (2) random
    - 1.1 times higher miss rate in 2-way set associative cache
- **In virtual memory**

  Large miss penalty -> Low miss rate is critical. ->LRU

Q4 : What Happens on a Write?

- **Two basic options**
  1) Write-through
  2) Write-back
- **Advantages of write-back**
  1) Writing at the rate of cache
  2) Only one write to the lower level
  3) Write-back with high-bandwidth transfer
     - Entire block write when write-back
- **Advantages of write-through**
  1) Misses are simpler and cheaper
     - No writes to the lower level
  2) Easier to implement (cf) write buffer

The Big Picture

Question 1 : Where can a block be placed?
Answer 1 : direct mapped, set associative, fully associative

Question 2 : How is a block found?
Answer 2 : indexing, limited search, full search,
separate lookup table

Question 3 : What block is replaced on a miss?
Answer 3 : LRU, random

Question 4 : How are writes handled?
Answer 4 : write through, write back

The 3 Cs : An Intuitive Model for Understanding the Behavior of Memory Hierarchies

- **Classifications of misses**
  1. Compulsory misses (= cold start misses)
     - The first access to a block
  2. Capacity misses
     - Cache cannot contain all the blocks needed by the process
     - When blocks are replaced and then later retrieved
  3. Conflict misses (= collision misses)
     - When too many blocks map to a set

Miss Rates and 3Cs

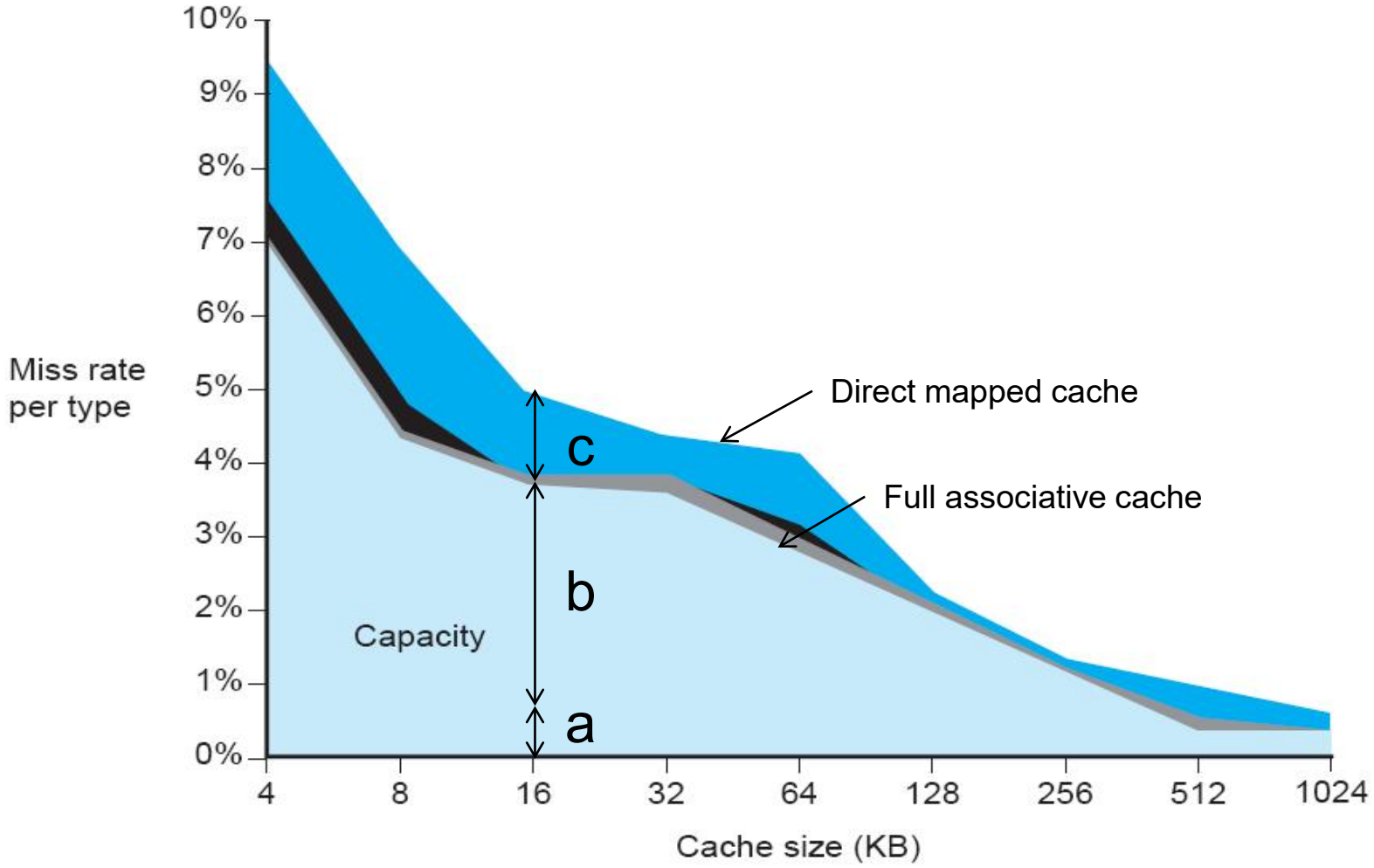


Figure 7.31

Design Challenges

Design change	Effect on miss rate	Possible negative performance effect
Increase cache size	decreases capacity misses	may increase access time
Increase associativity	decreases miss rate due to conflict misses	may increase access time
Increase block size	decreases miss rate for a wide range of block sizes due to spatial locality	increases miss penalty. Very large block increases miss rate

Figure 7.32

Reducing Misses

1. Conflict misses

- fully associative placement
- with increased access time

2. Capacity misses

- large cache
- with increased access time

3. Compulsory misses

- increased block size
- with increased miss penalty

5.8 Concluding Remarks

MPU	AMD Opteron	Intrinsity FastMATH	Intel Pentium 4	Intel PXA250	Sun UltraSPARC IV
Instruction set architecture	IA-32, AMD64	MIPS32	IA-32	ARM	SPARC v9
Intended application	server	embedded	desktop	low-power embedded	server
Die size (mm^2) (2004)	193	122	217		356
Instructions issued/clock	3	2	3 RISC ops	1	4×2
Clock rate (2004)	2.0 GHz	2.0 GHz	3.2 GHz	0.4 GHz	1.2 GHz
Instruction cache	64 KB, 2-way set associative	16 KB, direct mapped	12000 RISC op trace cache (~96 KB)	32 KB, 32-way set associative	32 KB, 4-way set associative
Latency (clocks)	3?	4	4	1	2
Data cache	64 KB, 2-way set associative	16 KB, 1-way set associative	8 KB, 4-way set associative	32 KB, 32-way set associative	64 KB, 4-way set associative
Latency (clocks)	3	3	2	1	2
TLB entries (I/D/L2 TLB)	40/40/512/ 512	16	128/128	32/32	128/512
Minimum page size	4 KB	4 KB	4 KB	1 KB	8 KB
On-chip L2 cache	1024 KB, 16-way set associative	1024 KB, 4-way set associative	512 KB, 8-way set associative	—	—
Off-chip L2 cache	—	—	—	—	16 MB, 2-way set associative
Block size (L1/L2, bytes)	64	64	64/128	32	32

Figure 7.36

Back to chapter overview

Growing Gap between CPU and DRAM

- **Processor speed**
  - 35%/year until 1986, 55%/year until 2003, then slow down
- **DRAM**
  - Density … x2 every 2 years
  - access time … 7%/year

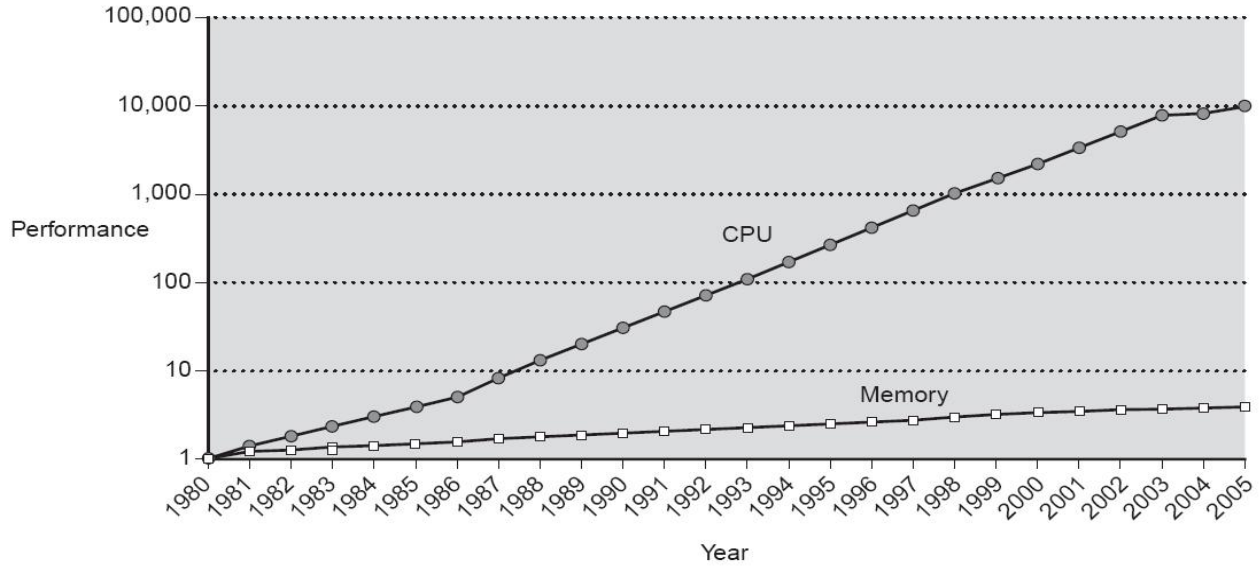


Figure 7.37

Recent Trends

- How to overcome the growing gap between processor speeds and lower levels of the memory hierarchy

1. **Multi-level caches**
   - L3 caches are becoming popular
2. **Software assists**
   1) Program reorganization
      - Enhancing spatial and temporal locality
      - Focus on loop-oriented programs
   2) Compiler-directed prefetching
      - Prefetch a block into the cache before it is actually referenced
      - Compiler identifies the blocks with special instructions