#8 : Memory Hierarchy

Computer Architecture 2021/2022

Ricardo Rocha

Computer Science Department, Faculty of Sciences, University of Porto

Slides based on the book

'Computer Organization and Design, The Hardware/Software Interface, 5th Edition

David Patterson and John Hennessy, Morgan Kaufmann'

Sections 5.1 – 5.5 and 5.7

Memory Hierarchy

"Ideally one would desire an indefinitely large memory capacity such that any particular […] word would be immediately available. […] We are […] forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible."

A. W. Burks, H. H. Goldstine, and J. von Neumann. *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument, 1946.*

Memory Hierarchy

Memory hierarchy is a **multi-level structure** that as the distance from the processor increases, the size of the memories and the access time both increase. **Performance** is the key reason for having a memory hierarchy.

The faster memories are more expensive per bit and thus tend to be smaller. The goal is to present the user with as **much memory as is available in the cheapest technology**, while **providing access at the speed offered by the fastest memory**.

The **data is similarly hierarchical** – a level closer to the processor is generally a subset of any level further away, and **all the data is stored at the lowest memory level**.

Memory Hierarchy

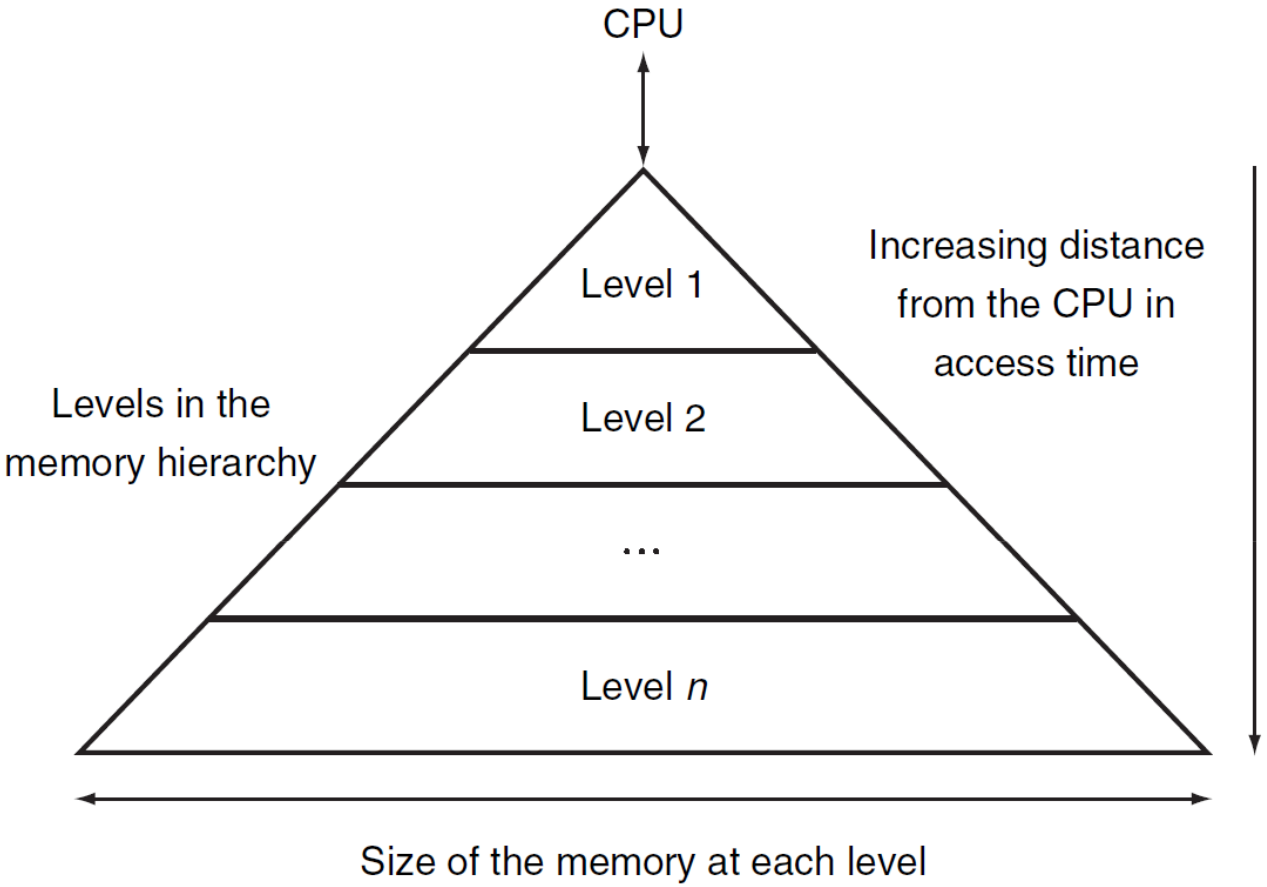


FIGURE 5.3 This diagram shows the structure of a memory hierarchy: as the distance from the processor increases, so does the size. This structure, with the appropriate operating mechanisms, allows the processor to have an access time that is determined primarily by level 1 of the hierarchy and yet have a memory as large as level *n*. Maintaining this illusion is the subject of this chapter. Although the local disk is normally the bottom of the hierarchy, some systems use tape or a file server over a local area network as the next levels of the hierarchy.

Principle of Locality

The principle of locality states that **programs access a relatively small portion of their address space at any instant of time**. There are two different types of locality:

- **Temporal locality** – principle stating that items referenced recently are likely to be referenced again soon (e.g., instructions in a loop, local variables) – memory hierarchies take advantage of temporal locality by **keeping more recently accessed data items closer to the processor**
- **Spatial locality** – principle stating that items near those referenced recently are likely to be referenced soon (e.g., sequential instruction access, data array) – memory hierarchies take advantage of spatial locality by **moving data items consisting of contiguous words in memory to upper memory levels**

Hits and Misses

Data is copied between only two adjacent levels at a time. Within each level, the **minimal unit of data is called a block (or line)**.

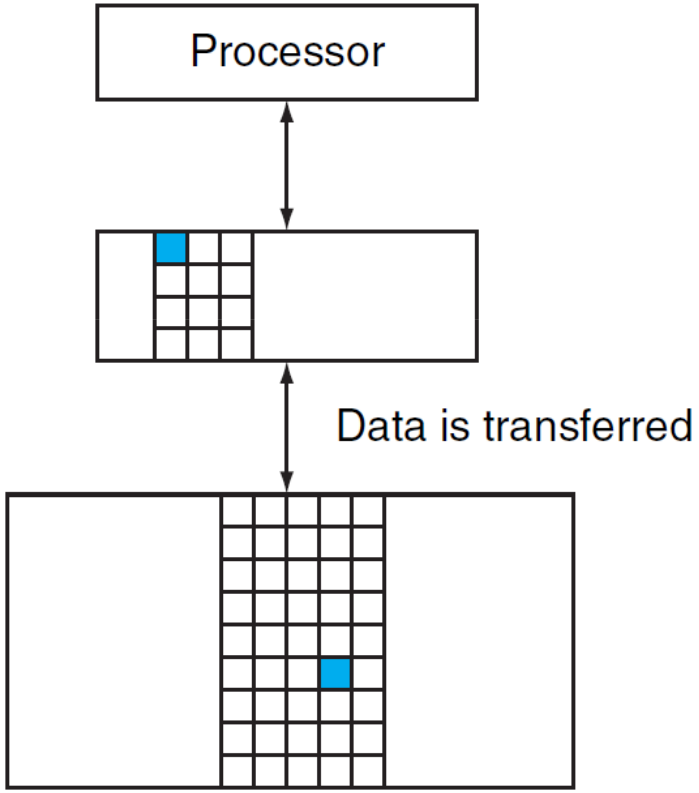


FIGURE 5.2 Every pair of levels in the memory hierarchy can be thought of as having an upper and lower level. Within each level, the unit of information that is present or not is called a *block* or a *line*. Usually we transfer an entire block when we copy something between levels.

Hits and Misses

If the data requested by the processor appears in some block in the upper memory level, this is called a **hit**. Otherwise, the request is called a **miss** and the next memory level is then accessed to retrieve the block containing the requested data.

The **hit rate** is the fraction of memory accesses found in the upper memory level – often used as a measure of the performance of the memory hierarchy. The **miss rate** (1–hit rate) is the fraction of memory accesses not found in the upper memory level.

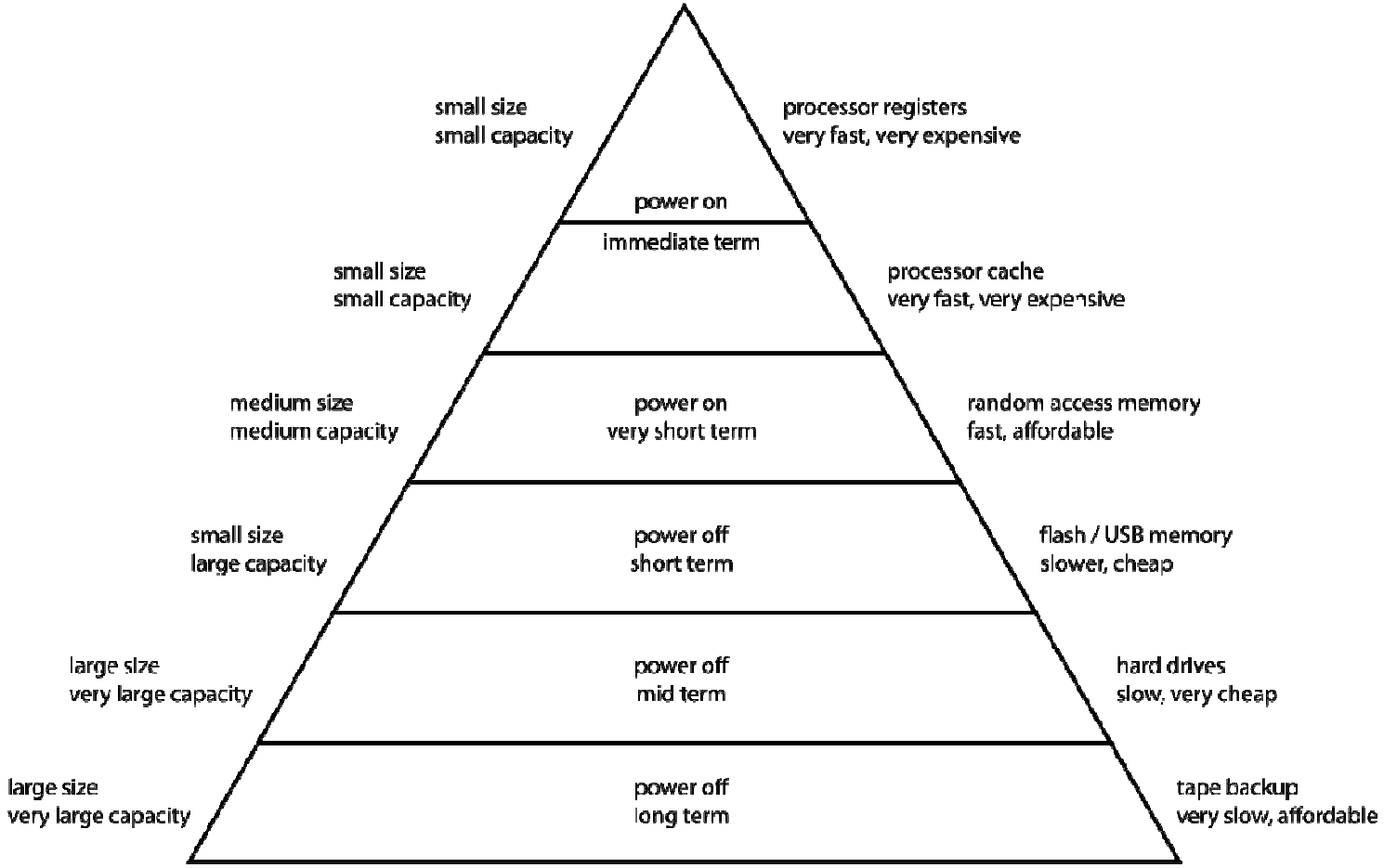
Hits and Misses

Hit time is the time to access the upper memory level, which includes the time needed to determine whether the access is a hit or a miss.

Miss penalty is the time to replace a block in the upper memory level with the corresponding block from the next memory level, plus the time to deliver this block to the processor. The time to access the next memory level is the major component of the miss penalty.

If the **hit rate is high enough**, the memory hierarchy has an **effective access time close to that of the upper memory level** and therefore be **able to virtually represent a size equal to that of the lowest memory level**.

Memory Technologies



Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

Cache Memory

Cache: a safe place for hiding or storing things.

Webster's New World Dictionary of the American Language, 1988.

Cache memory is the **level of the memory hierarchy closest to the CPU**. Caches first appeared in research computers in the early 1960s and in production computers later in that same decade. Every general purpose computer built today, from servers to low-power embedded processors, includes caches.

Cache Memory

Caching is perhaps the most important example of the big idea of prediction.

The hit rates of the cache prediction on modern computers are often higher than 95%.

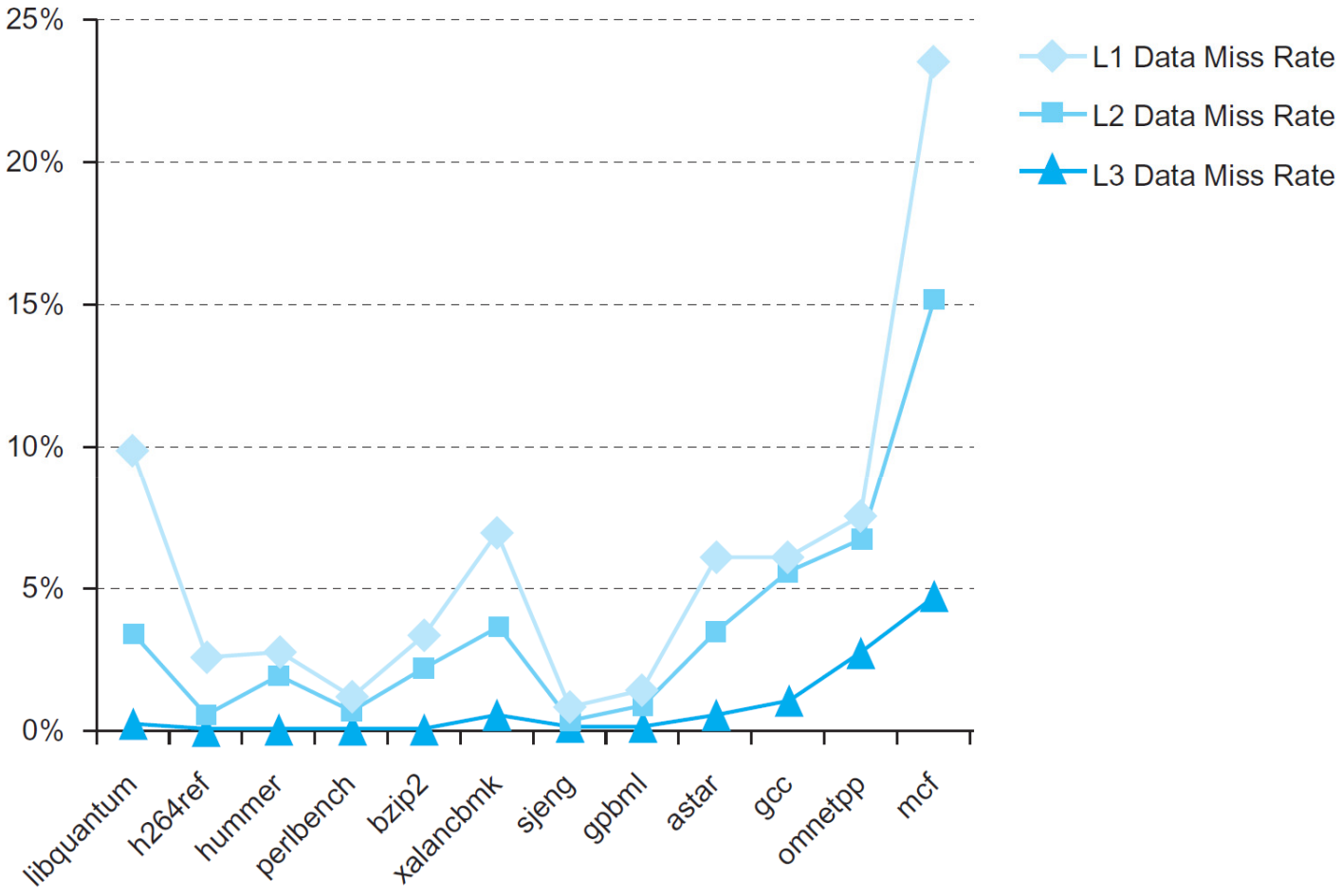


FIGURE 5.47 The L1, L2, and L3 data cache miss rates for the Intel Core i7 920 running the full integer SPECCPU2006 benchmarks.

Cache Memory

Questions to answer:

- How do we know if a data item is in cache?
- How do we find a data item in cache?

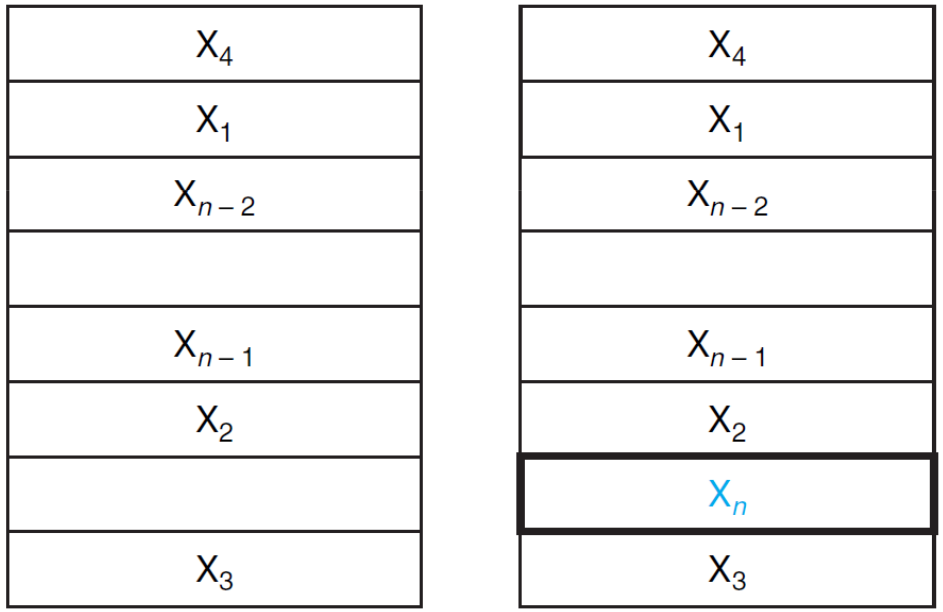
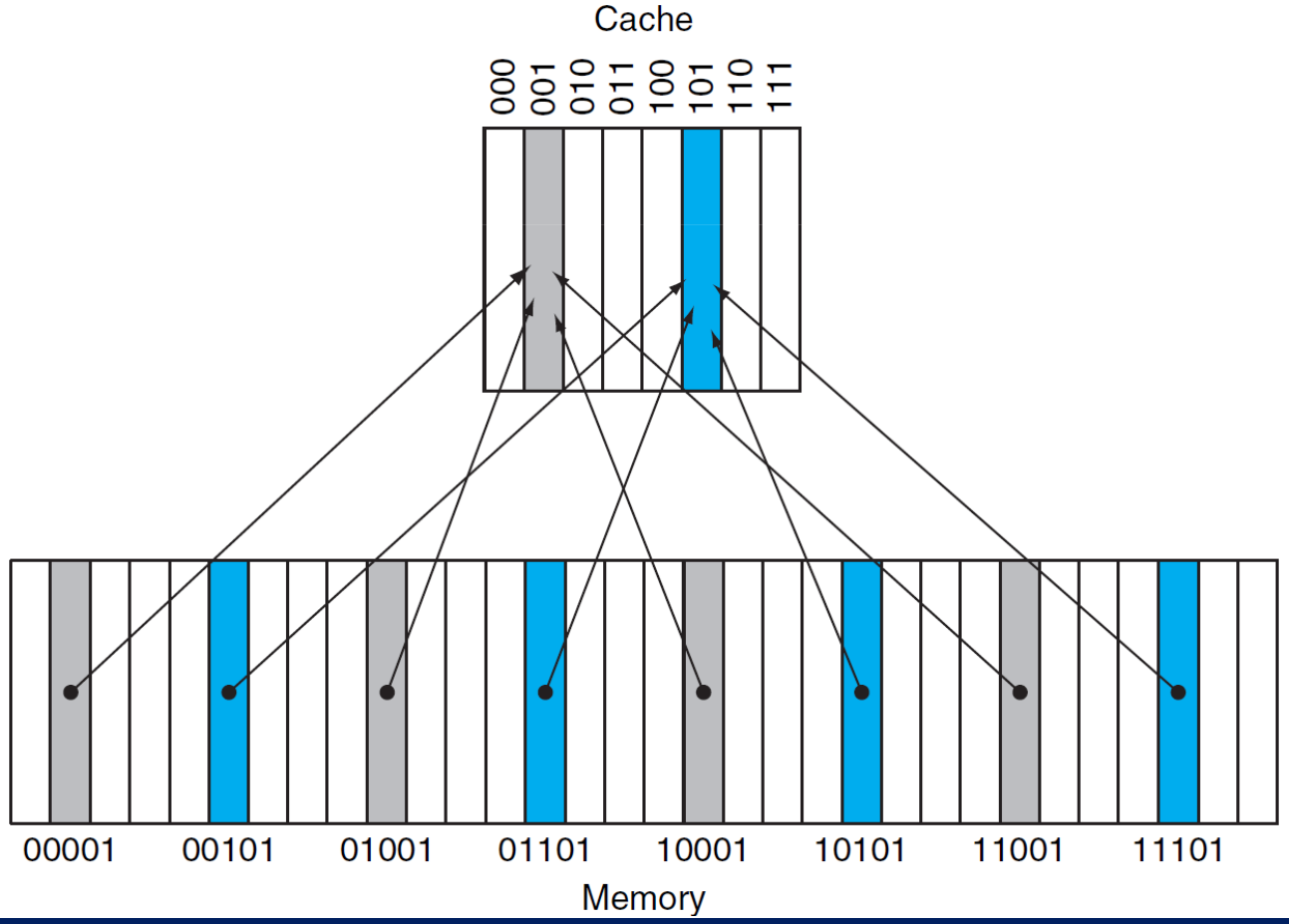


FIGURE 5.7 The cache just before and just after a reference to a word X_n that is not initially in the cache. This reference causes a miss that forces the cache to fetch X_n from memory and insert it into the cache.

Direct-Mapped Cache

In a direct-mapped cache, each block address is mapped to exactly one location in the cache. Almost all direct-mapped caches use the mapping:

(block address) modulo (#blocks in cache)



Direct-Mapped Cache

How do we compute the cache location for a given block address?

- Since the number of cache blocks is often a power of 2, use the **low-order bits of a block address** to compute its cache location

How do we know which particular block is in a cache location?

- Add a set of **tags to each cache location** identifying the block address in cache (actually, the tag only needs to contain the complementary higher-order bits)

How do we know there is valid data in a cache location?

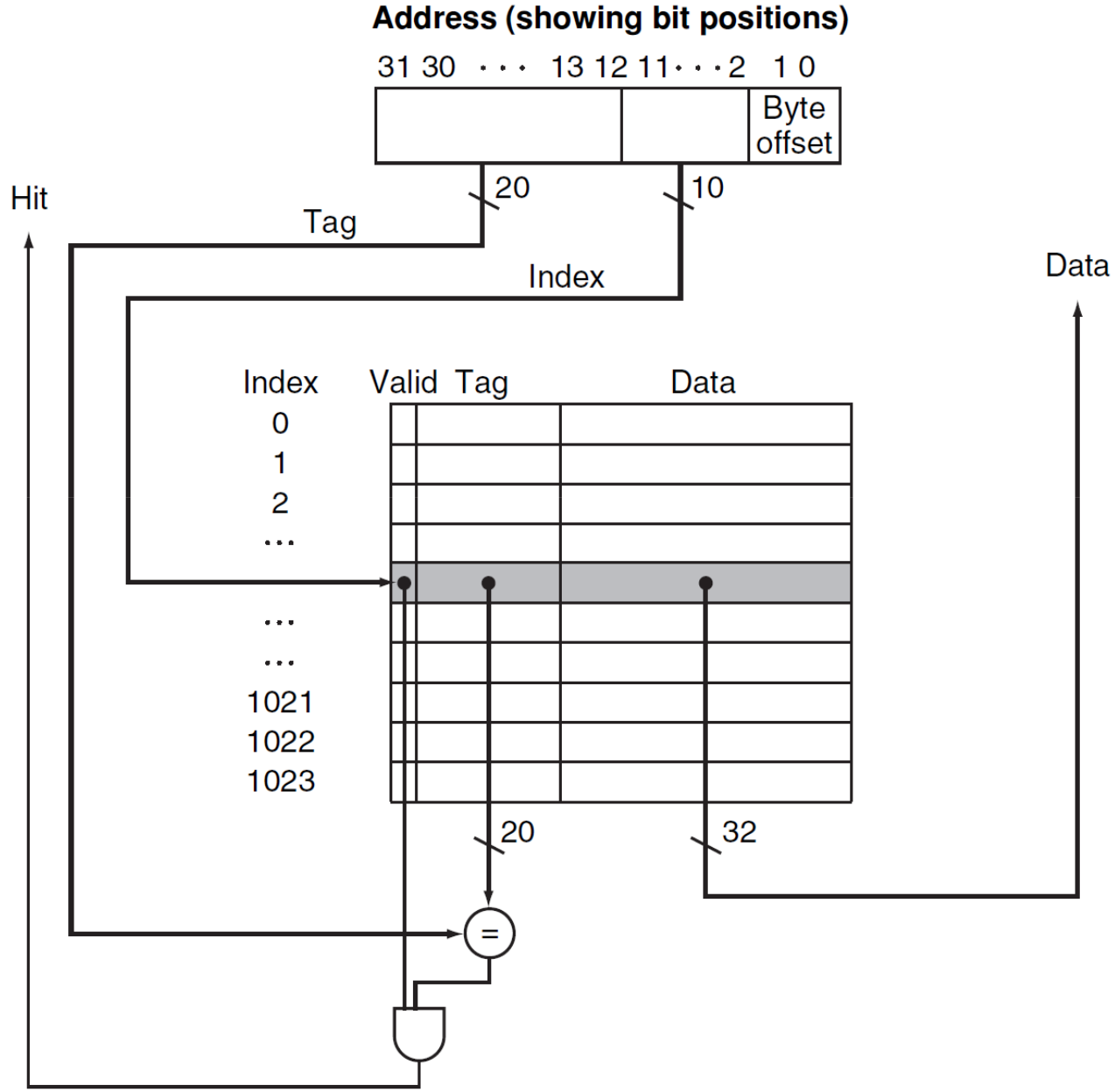
- Add a **valid bit to each cache location** indicating whether a location contains valid data (valid bit: 1 = valid data; 0 = invalid data; initially = 0)

8 x 1 Byte Blocks Direct-Mapped Cache

Decimal address of reference	Binary address of reference	Assigned cache block (where found or placed)	
22	10110_{two}	$(10110_{two} \bmod 8) = 110_{two}$	Miss
26	11010_{two}	$(11010_{two} \bmod 8) = 010_{two}$	Miss
22	10110_{two}	$(10110_{two} \bmod 8) = 110_{two}$	Hit
26	11010_{two}	$(11010_{two} \bmod 8) = 010_{two}$	Hit
16	10000_{two}	$(10000_{two} \bmod 8) = 000_{two}$	Miss
3	00011_{two}	$(00011_{two} \bmod 8) = 011_{two}$	Miss
16	10000_{two}	$(10000_{two} \bmod 8) = 000_{two}$	Hit
18	10010_{two}	$(10010_{two} \bmod 8) = 010_{two}$	Miss
16	10000_{two}	$(10000_{two} \bmod 8) = 000_{two}$	Hit

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

1024 x 4 Byte Blocks Direct-Mapped Cache



Bits in a Cache

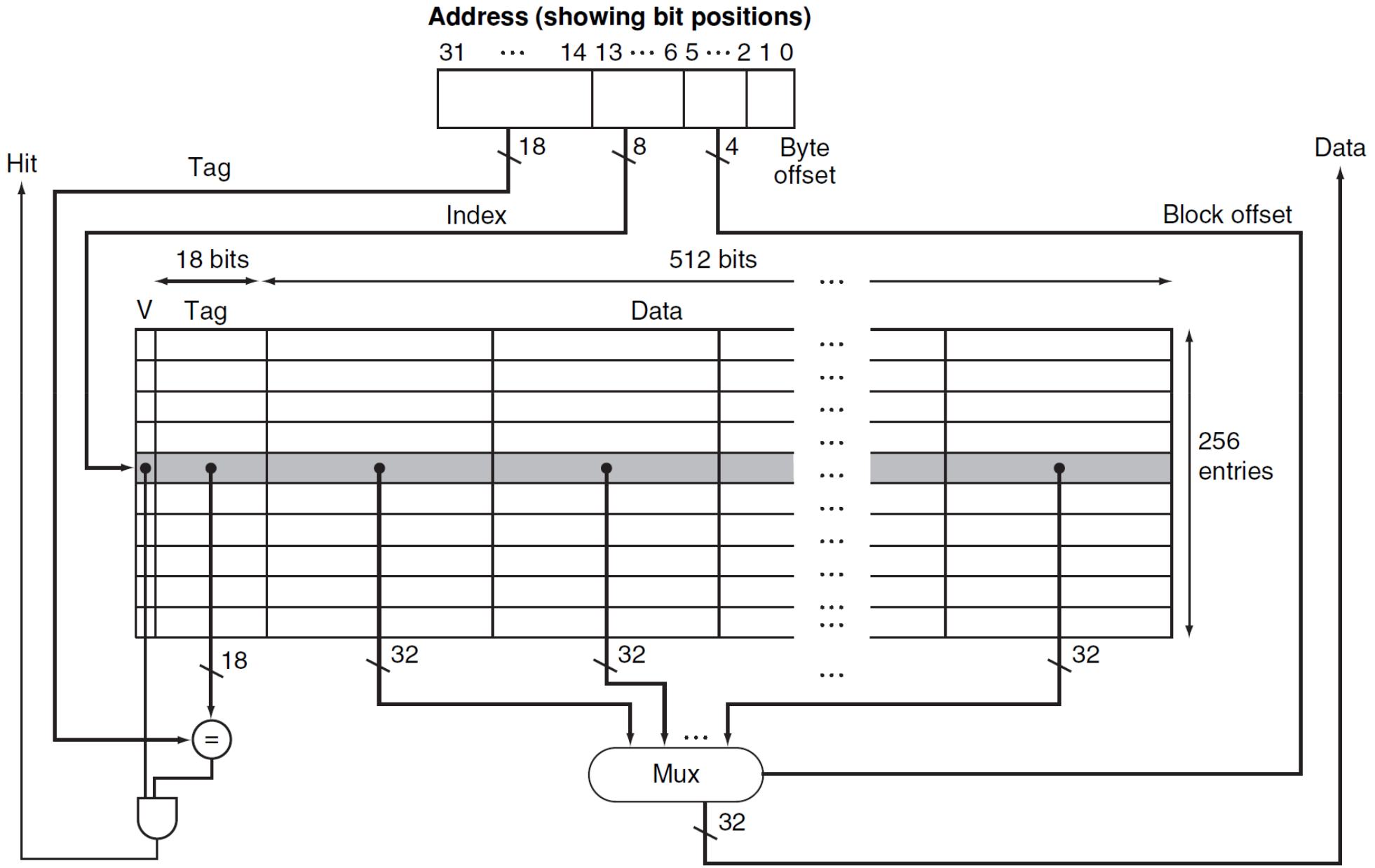
How many bits are required to implement the previous cache structure?

The cache size is 1024 entries (2^{10} blocks). Each block has 32 bits (4 bytes or 1 word) of data plus a tag with 20 bits and a valid bit. Thus, the actual size in bits is:

$$2^{10} \times (32 + 20 + 1) = 2^{10} \times 53 = 53 \text{ Kib } (= 1.656 \times 32 \text{ Kib })$$

The total number of bits in the cache is 1.656 times as many as needed just for the storage of the data. Regardless of the actual size in bits, the naming convention is to exclude the size of the tag and valid field and to count only the size of the data. Thus, **this cache is called a 4 KiB cache**.

256 x 64 Byte Blocks Direct-Mapped Cache



Block Size Considerations

In a fixed-size cache, are larger blocks better?

- **Larger blocks should reduce miss rate** as they exploit spatial locality
- But larger blocks will **reduce the total number of blocks**, which increases the competition for those blocks and eventually the miss rate
- In particular, the **miss rate may go up if the block size becomes a significant fraction of the cache size**
- A collateral problem is that the transfer time required to fetch a block from the next memory level **(miss penalty) will likely increase as the block size increases**

Block Size Considerations

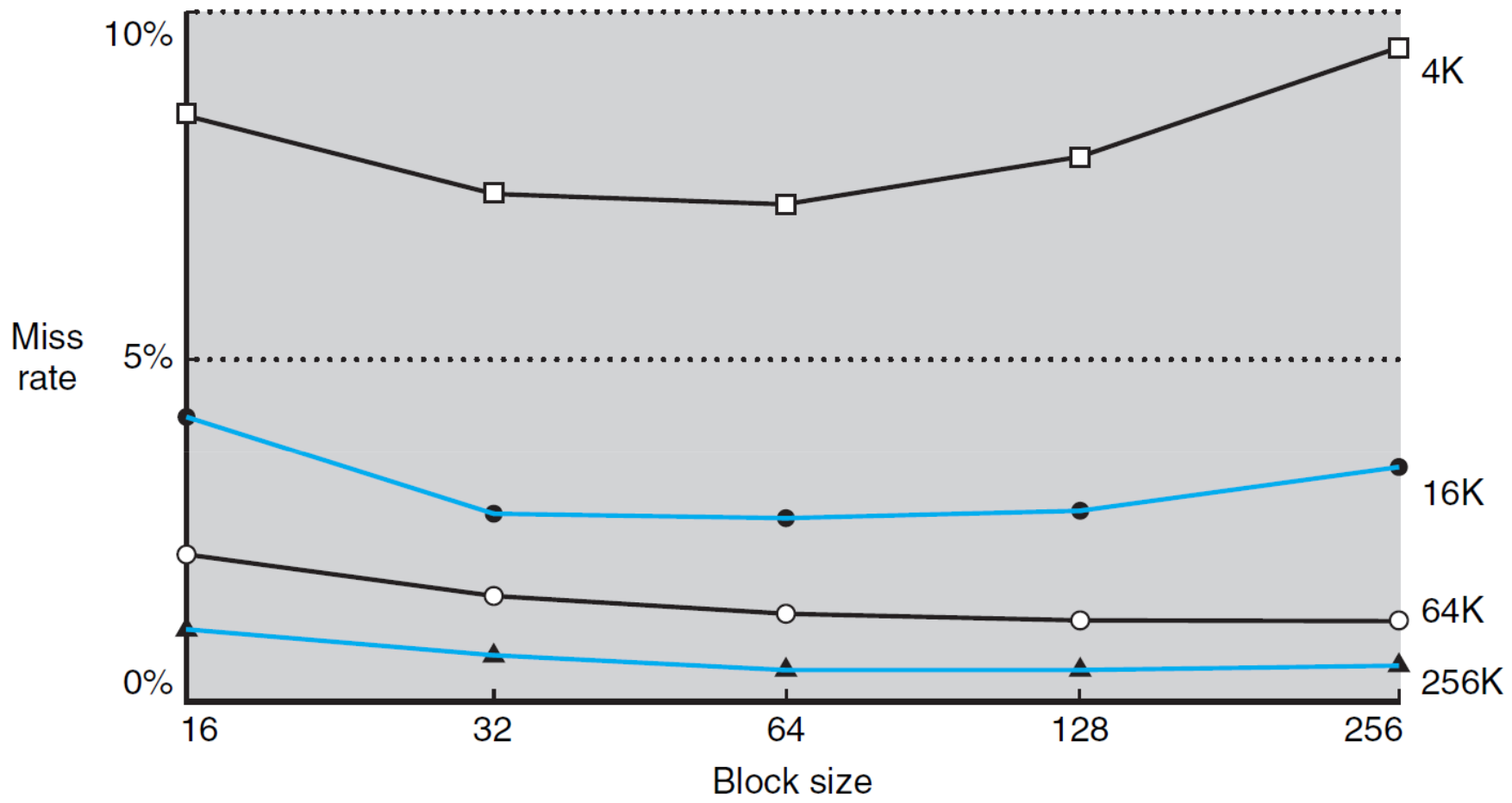


FIGURE 5.11 Miss rate versus block size. Note that the miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. (This figure is independent of associativity, discussed soon.) Unfortunately, SPEC CPU2000 traces would take too long if block size were included, so this data is based on SPEC92.

Associative Caches

Fully associative cache

- Blocks can be placed in any entry in the cache
- To find a given block, requires searching all entries in parallel
- To make search practical, each entry has a comparator (significantly increases the hardware cost)

N-way set associative cache

- Blocks can be placed in a fixed number of N entries (at least two), called a set
- Each block address is mapped to exactly one set in the cache
- Each set contains N entries and a block can be placed in any entry of the set
- To find a given block, requires searching the N entries in a set
- To make search practical, each entry has N comparators (less expensive)

Associative Caches

In a direct-mapped cache, the entry for a memory block is given by:

(block address) modulo (#blocks in cache)

In a set-associative cache, the set for a memory block is given by:

(block address) modulo (#sets in cache)

Associative Caches

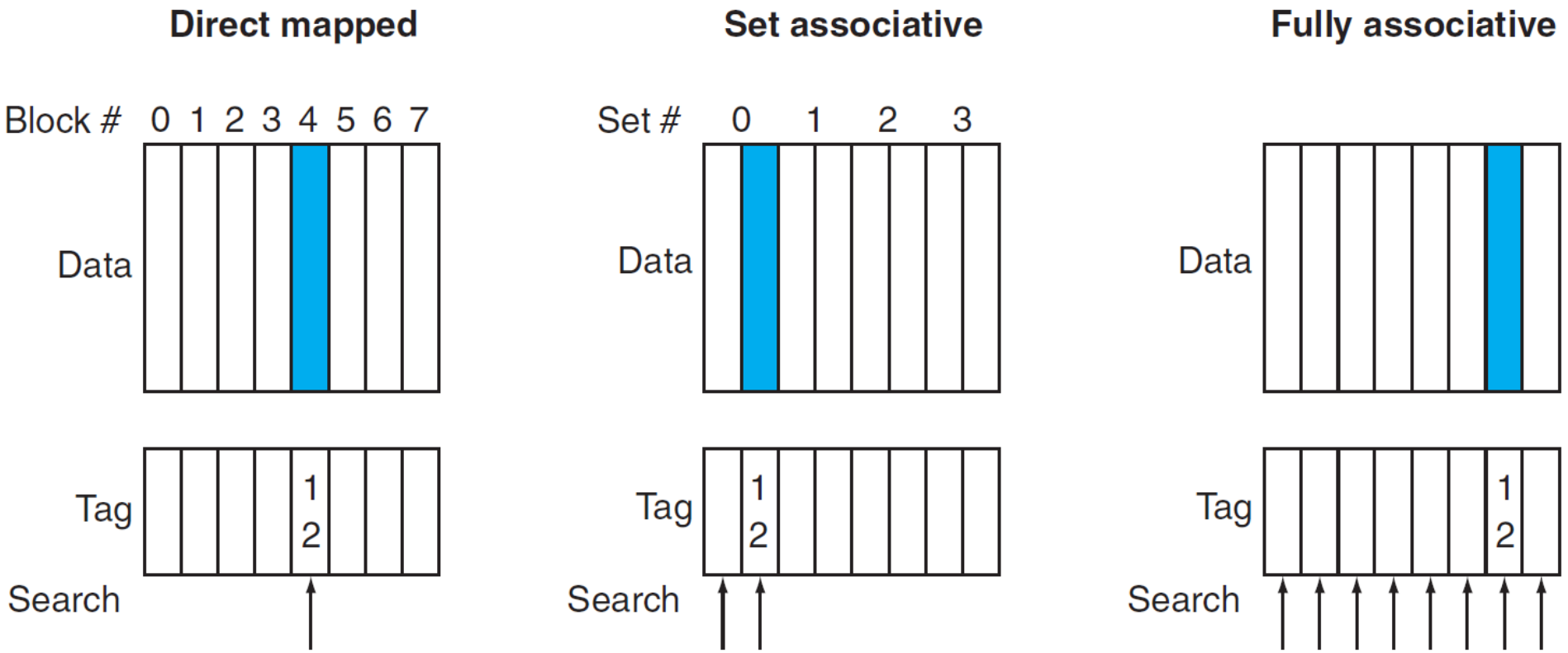


FIGURE 5.14 The location of a memory block whose address is 12 in a cache with eight blocks varies for direct-mapped, set-associative, and fully associative placement. In direct-mapped placement, there is only one cache block where memory block 12 can be found, and that block is given by (12 modulo 8) = 4. In a two-way set-associative cache, there would be four sets, and memory block 12 must be in set (12 mod 4) = 0; the memory block could be in either element of the set. In a fully associative placement, the memory block for block address 12 can appear in any of the eight cache blocks.

Spectrum of Associativity

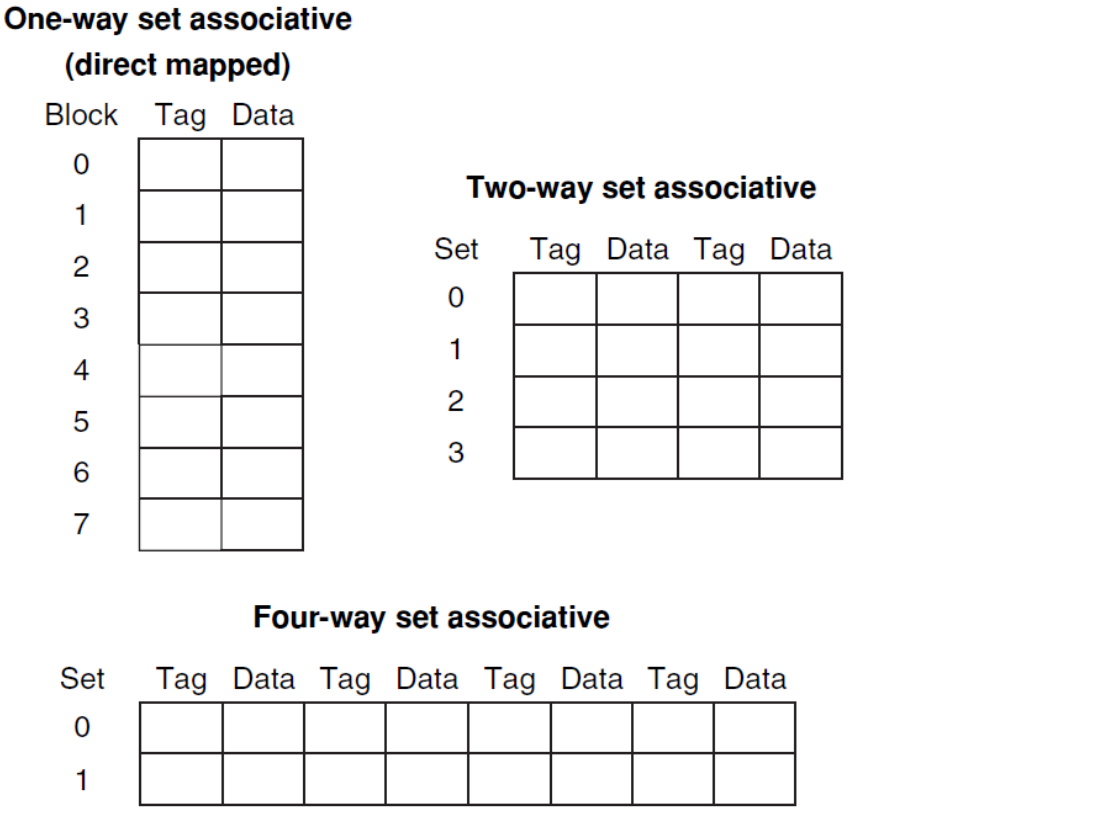


FIGURE 5.15 An eight-block cache configured as direct mapped, two-way set associative, four-way set associative, and fully associative. The total size of the cache in blocks is equal to the number of sets times the associativity. Thus, for a fixed cache size, increasing the associativity decreases the number of sets while increasing the number of elements per set. With eight blocks, an eight-way set-associative cache is the same as a fully associative cache.

Spectrum of Associativity

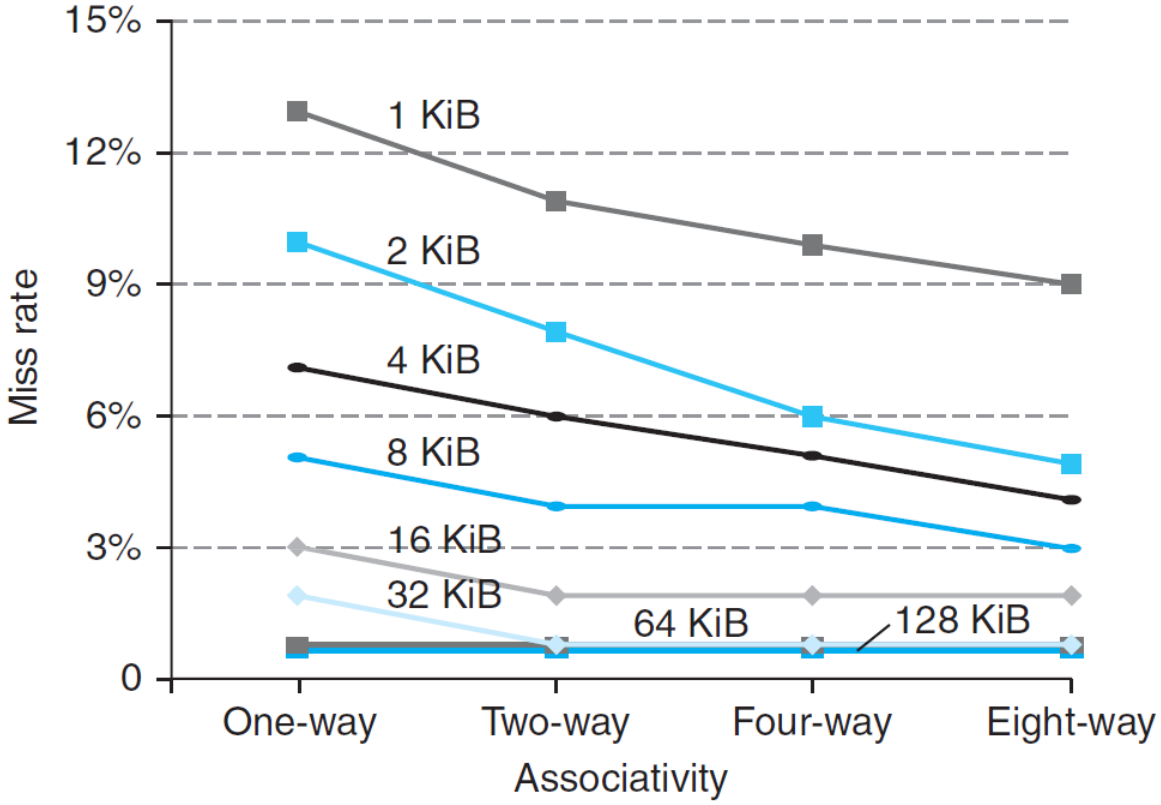


FIGURE 5.36 The data cache miss rates for each of eight cache sizes improve as the associativity increases. While the benefit of going from one-way (direct mapped) to two-way set associative is significant, the benefits of further associativity are smaller (e.g., 1%–10% improvement going from two-way to four-way versus 20%–30% improvement going from one-way to two-way). There is even less improvement in going from four-way to eight-way set associative, which, in turn, comes very close to the miss rates of a fully associative cache. Smaller caches obtain a significantly larger absolute benefit from associativity because the base miss rate of a small cache is larger. Figure 5.16 explains how this data was collected.

Replacement Policy

In an associative cache, we have a choice of where to place the requested block, and hence a **choice of which block to replace**.

- In a fully associative cache, all blocks are candidates for replacement
- In a set-associative cache, we must choose among the blocks in the selected set

The most commonly used scheme is **least recently used (LRU)** – the block replaced is the one that has been unused for the longest time.

- Simple for two-way, manageable for four-way, too hard beyond that
- For a two-way can be implemented by keeping a single bit in each set and setting the bit to indicate an element whenever that element is referenced

For high associativity caches, a **random scheme** gives approximately the same performance as LRU.

Example

Consider a small cache with four one-word blocks. Find the number of misses given the following sequence of block addresses: 0, 8, 0, 6, and 8.

Direct-mapped cache

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

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

Example

Consider a small cache with four one-word blocks. Find the number of misses given the following sequence of block addresses: 0, 8, 0, 6, and 8.

Two-way set associative cache (with LRU replacement)

Block address	Cache set
0	(0 modulo 2) = 0
6	(6 modulo 2) = 0
8	(8 modulo 2) = 0

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

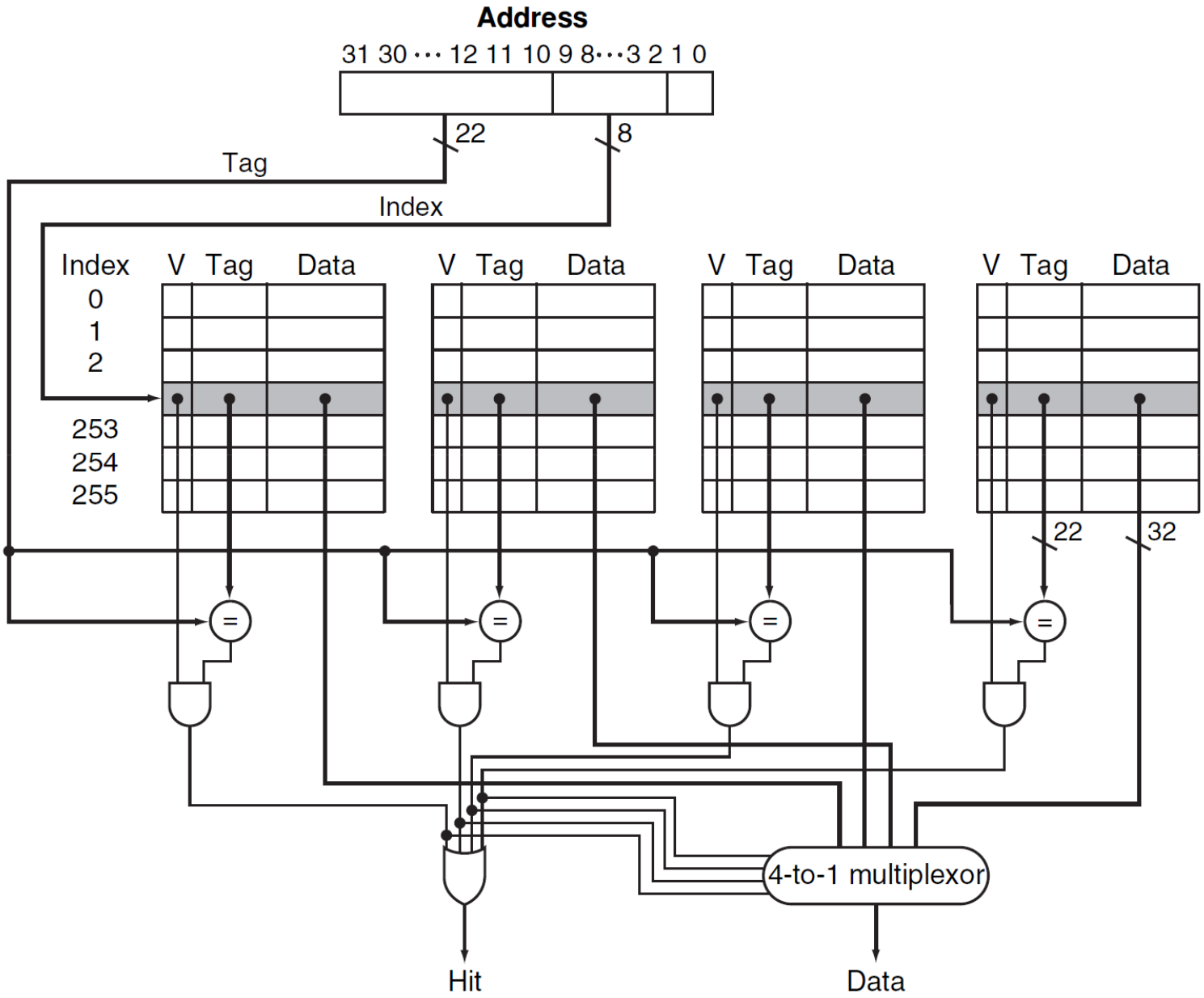
Example

Consider a small cache with four one-word blocks. Find the number of misses given the following sequence of block addresses: 0, 8, 0, 6, and 8.

Fully associative cache

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

Four-Way Set-Associative Cache



Handling Cache Misses

Handling cache misses is done in collaboration with the processor control unit. The processing of a cache miss **stalls the entire processor**, essentially freezing the contents of all registers while waiting for memory. The steps taken in a cache miss are:

- Instruct the next memory level to read the missing value
- Wait for the memory to respond (it can take multiple clock cycles)
- Update the corresponding cache line with the data received from memory
- Refetch and restart the instruction execution, this time finding it in the cache

More sophisticated processors can allow out-of-order execution of other instructions while waiting for a cache miss.

Handling Writes

How to handle write hits – consider a store instruction where a data-write hit is only wrote into the cache, without changing main memory. Then, **cache and memory would have different values**. In such a case, the cache and memory are said to be **inconsistent**.

How to handle write misses – should we fetch the corresponding block from memory to cache and then overwrite with the word that caused the miss (called **write allocate**) or should we simply write the word to main memory (called **no write allocate**)?

Caches Write-Through and Write-Back

Write-through is a scheme in which **write hits always update both the cache and the next memory level**, thus ensuring that **data is always consistent between the two**.

Write-back is a scheme that **handles write hits by updating only the cache**, then the **modified block is written to the next memory level when it is replaced** (need to keep track of modified blocks).

Write-Through Considerations

Despite its simplicity, this scheme **does not provide good performance**:

- Suppose that writes take 100 clock cycles longer and that 10% of the instructions are stores – if the base CPI (without cache misses) was 1.0, the 100 extra cycles on every write would lead to a CPI of 1.0 + 100 × 10% = 11.0, thus reducing performance by more than a factor of 10

One solution is to use a **write buffer** to hold the data waiting to be written to memory. **Execution can continue immediately** after writing the data into the cache and into the write buffer.

- The processor only stalls if the write buffer is full when reaching a write
- When a write to main memory completes, the entry in the write buffer is freed
- If the rate at which the memory can complete writes is less than the rate at which the processor is generating writes, no amount of buffering can help

Write-Back Considerations

Write-back **can improve performance** especially when processors can generate writes as fast or faster than the writes can be handled by main memory.

However, write-back is **more complex to implement** than write-through.

- If we simply overwrite a modified block on a store instruction before knew whether there is a write miss, we would **destroy the contents of the block in cache**, which is not backed up in the next memory level
- Stores either **require two cycles** (a cycle to check for a hit followed by a cycle to actually perform the write) or **require a write buffer** to hold the data to be written while the block is checked for a hit – thus allowing the store to take only one cycle by pipelining it

Cache Performance

Remember the performance equation:

$$CPUTime = InstructionCount \times CPI \times ClockPeriod$$

$$= \frac{Instructions}{Program} \times \frac{ClockCycles}{Instruction} \times \frac{Seconds}{ClockCycle}$$

CPU time can be divided into the clock cycles that the CPU spends executing the instructions with no misses (CPIPerfect) and the clock cycles that the CPU spends waiting for the memory system (CPIStall).

$$CPI = CPIPerfect + CPIStall$$

Cache Performance

Memory-stall clock cycles can be defined as the sum of the stall cycles coming from reads plus those coming from writes. For simplicity, let's assume that the read/write miss rates and miss penalties are the same:

$$CPIStall = \frac{MemoryAccesses}{Instructions} \times MissRate \times MissPenalty$$

If we consider separate caches/memories for instructions and data then:

$$\begin{aligned} CPIStall &= CPIStallInstructionAccess + CPIStallDataAcess \\ &= 1 \times MissRateInstructionAccess \times MissPenalty \\ &\quad + \frac{LoadStores}{Instructions} \times MissRateDataAccess \times MissPenalty \end{aligned}$$

Example

Assume a miss rate of 2% for the instruction cache and of 4% for the data cache, a miss penalty of 100 cycles for all misses, and a frequency of 36% of loads and stores. If the CPI is 2 without memory stalls, determine how much faster the processor runs with a perfect cache that never misses.

CPI memory stall for instructions and data access:

- CPIStallInstructionAccess = 1 × 0.02 × 100 = 2.00
- CPIStallDataAccess = 0.36 × 0.04 × 100 = 1.44
- CPIStall = 2.00 + 1.44 = 3.44

Accordingly, the total CPI including memory stalls is:

- CPI = 2 + 3.44 = 5.44

Performance with perfect cache is better by 2.72 = 5.44 / 2

Clock Rate Improvement

What happens if the processor is made faster but memory access is not?

Suppose we **speed-up the clock rate** by a factor of 2:

- The previous miss penalty of 100 cycles is now 200 cycles
- CPIStallInstructionAccess = 1 × 0.02 × 200 = 4.00
- CPIStallDataAccess = 0.36 × 0.04 × 200 = 2.88
- CPIStall = 4.00 + 2.88 = 6.88
- CPI = 2 + 6.88 = 8.88
- The fraction of time spent on memory stalls would have risen from 63% to 77% (3.44 / 5.44 = 63% and 6.88 / 8.88 = 77%)
- The total execution time of a program is better by 1.23 = 5.44 / (8.88 / 2)

CPI Improvement

What happens if the processor is made faster but memory access is not?

Suppose that an **improved pipeline reduces the CPI** from 2 to 1 without changing the clock rate:

- CPI = 1 + 3.44 = 4.44
- The fraction of time spent on memory stalls would have risen from 63% to 77% (3.44 / 5.44 = 63% and 3.44 / 4.44 = 77%)
- The total execution time of a program is better by 1.23 = 5.44 / 4.44

In both situations, the time spent on memory stalls takes an increasing fraction of the total execution time, which turns the expected impact on performance very low (x1.23) compared to the speed-up factor (x2).

Multilevel Caches

To close the gap between the fast clock rates of modern processors and the increasingly long time required to access DRAMs, all modern computers **support additional levels of caching**.

The second-level (L2) cache is accessed whenever a miss occurs in the first-level (L1) cache. If the L2 cache contains the desired data, the **miss penalty for the L1 cache will be essentially the access time of the L2 cache**, which will be much less than the access time of main memory. The same happens for a third-level (L3) cache, if it exists.

If neither the L1, L2 nor L3 cache contains the data, a main memory access is required, and a larger miss penalty is incurred.

Multilevel Caches

With multilevel caches, memory-stall clock cycles can be defined as the sum of the stall cycles coming from the several cache levels. For simplicity, let's assume single instruction/data caches:

$$\begin{aligned} CPIStall &= MissRate \times MissPenalty \\ &= MissRateL1 \times MissPenaltyL1 \\ &+ GlobalMissRateL2 \times MissPenaltyL2 + \ldots \end{aligned}$$

The global miss rate for a level L represents the miss rate for the set of levels up to L:

$$GlobalMissRateL2 = MissRateL1 \times MissRateL2$$

$$GlobalMissRateL3 = MissRateL1 \times MissRateL2 \times MissRateL3$$

Multilevel Caches Considerations

A two-level cache structure allows the **L1 cache to focus on minimizing hit time**, to yield a shorter clock cycle or fewer pipeline stages, while allowing the **L2 cache to focus on miss rate**, to reduce the penalty of main memory access.

In comparison to a single level cache, the **L1 cache is often smaller** and the **L2 cache is often larger**. Given the focus of reducing miss rates, L2 often uses higher associativity than L1 and L2 may use a larger block size than L1.

Example

Assume a clock rate of 4GHz, a miss rate per instruction of 2% at the L1 cache and a memory access time of 100ns. If the base CPI is 1 without memory stalls, determine how much faster will the processor be if we add a L2 cache that has a 5ns access time and is large enough to reduce the global miss rate to main memory to 0.5%?

CPI with just L1 cache

- Clock duration = 1 / 4GHz = 0.25ns
- Miss penalty to main memory = 100ns / 0.25ns = 400 cycles
- CPIStall = 0.02 × 400 = 8
- CPI = 1 + 8 = 9

Example

Assume a clock rate of 4GHz, a miss rate per instruction of 2% at the L1 cache and a memory access time of 100ns. If the base CPI is 1 without memory stalls, determine how much faster will the processor be if we add a L2 cache that has a 5ns access time and is large enough to reduce the global miss rate to main memory to 0.5%?

CPI with L2 cache added

- Miss penalty to L2 = 5ns / 0.25ns = 20 cycles
- Miss penalty to main memory = 400 cycles
- CPIStall = 0.02 × 20 + 0.005 × 400 = 0.4 + 2.0 = 2.4
- CPI = 1 + 2.4 = 3.4

Performance with L2 cache is faster by 2.6 = 9.0 / 3.4

Virtual Memory

Virtual memory is a technique that uses **main memory as a cache for secondary storage**.

- Allows the execution of processes that are not entirely in memory
- Abstracts main memory into an extremely large uniform array of storage
- Frees programmers from the concerns of memory storage limitations

Code needs to be in memory to execute, but entire program rarely used:

- Code to handle unusual situations is almost never executed
- Large data structures often allocate more memory than they actually need
- Even if the entire program is used, it is not all needed at same time

Virtual Memory

Executing a process that is not entirely in memory benefits not only the users but also the operating system:

- Allows for less memory usage
- Allows for more efficient process creation
- Less I/O needed to load or swap processes into memory
- More programs could run concurrently
- Virtual address space can be much larger than physical address space

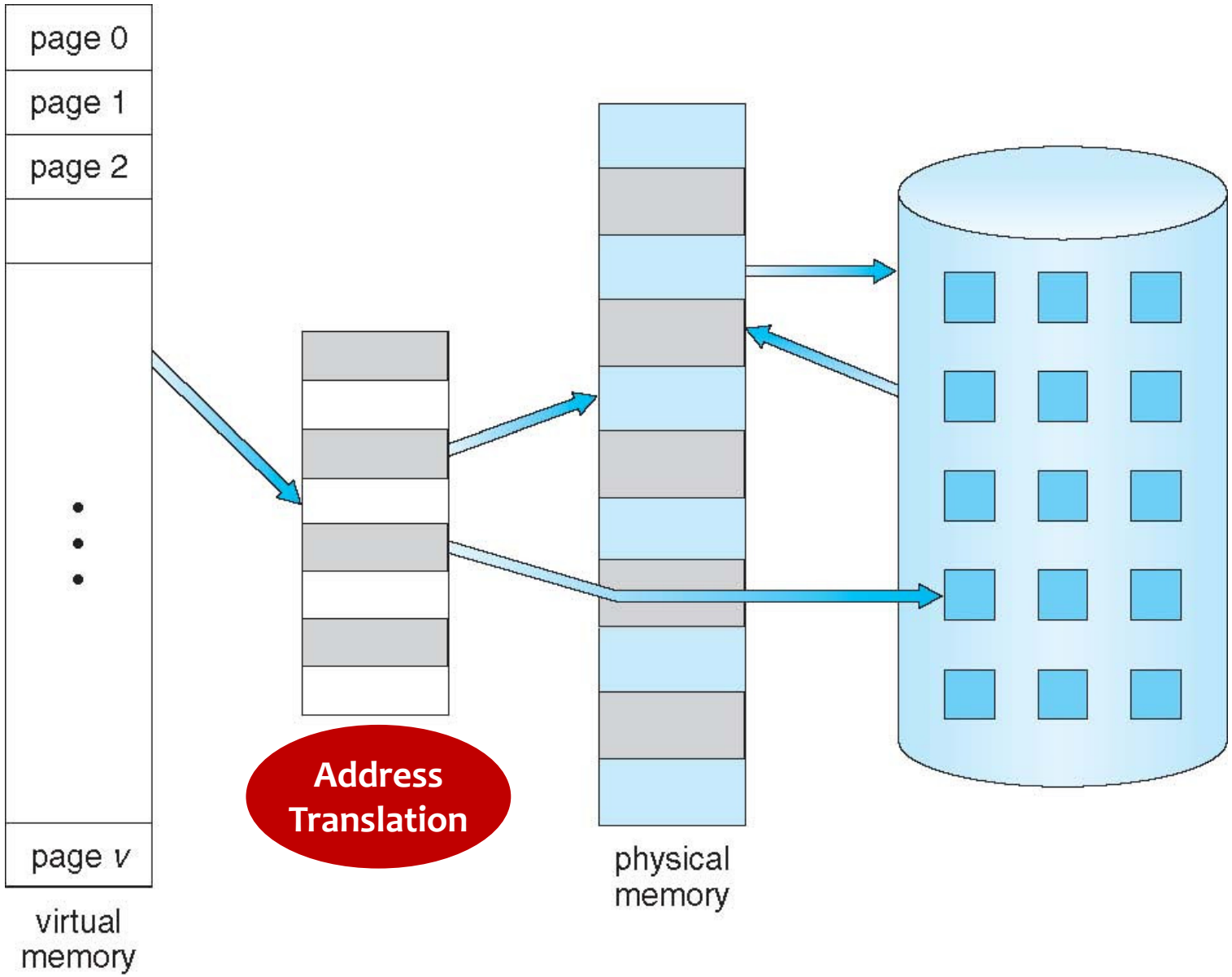
Virtual Memory

Although the concepts at work in virtual memory and in caches are the same, they use different terminology:

- **A virtual memory block is called a page**
- **A physical memory block is called a frame**
- **A virtual memory miss is called a page fault**

With virtual memory, the processor produces a virtual address, which is translated by a combination of hardware and software to a physical address, which in turn can be used to access main memory. This process is called **address mapping or address translation**.

Virtual to Physical Address Translation



Virtual to Physical Address Translation

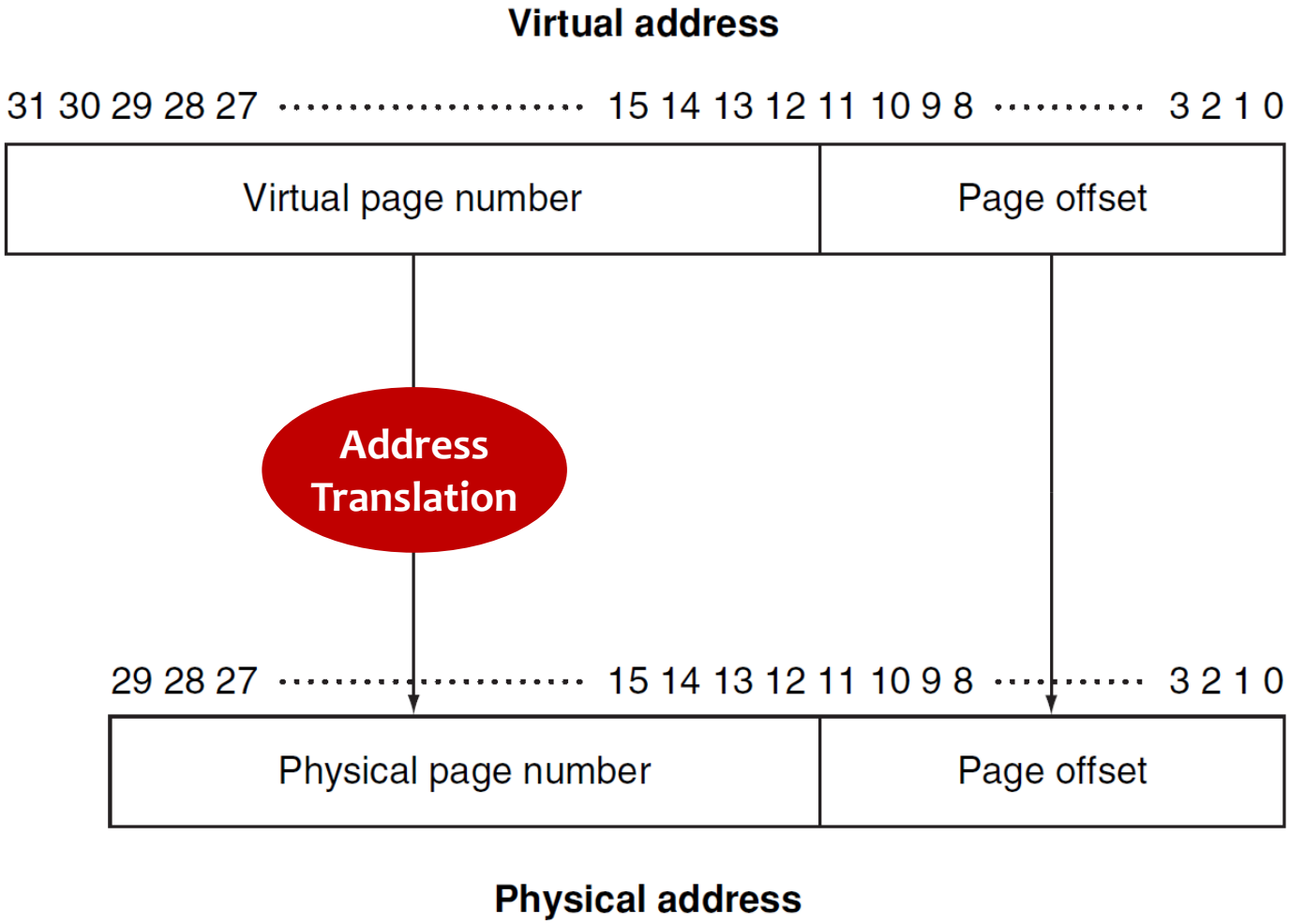


FIGURE 5.26 Mapping from a virtual to a physical address. The page size is $2^{12} = 4$ KiB. The number of physical pages allowed in memory is 2^{18}, since the physical page number has 18 bits in it. Thus, main memory can have at most 1 GiB, while the virtual address space is 4 GiB.

Page Table

The page table **contains the virtual to physical address translations** in a virtual memory system. The page table is typically indexed by the virtual page number – **each entry in the page table contains the physical page number for that virtual page, if the page is currently in memory**.

Each process has its own **page table register** that points to the corresponding page table. Page tables are stored in memory and thus **can also incur in page faults**.

Page Table

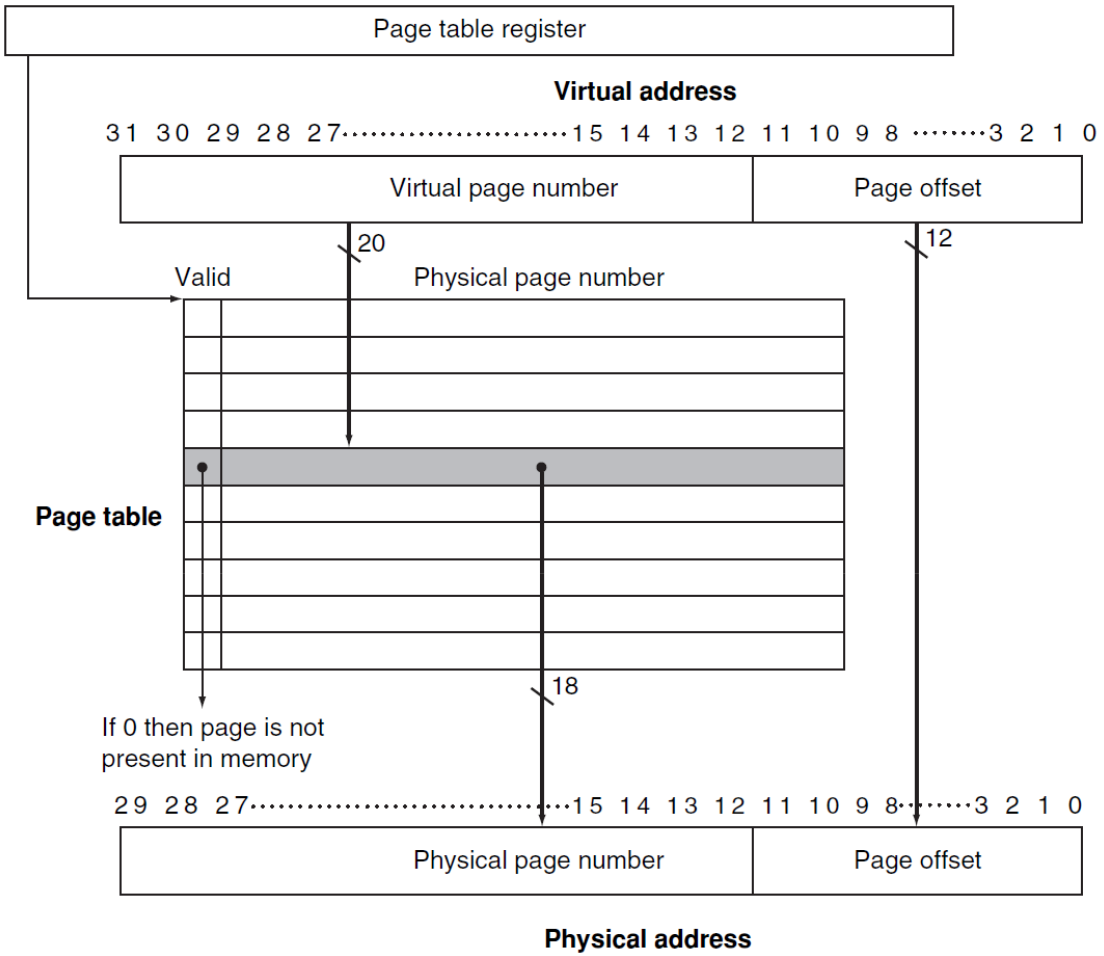


FIGURE 5.27 The page table is indexed with the virtual page number to obtain the corresponding portion of the physical address. We assume a 32-bit address. The page table pointer gives the starting address of the page table. In this figure, the page size is 2^{12} bytes, or 4 KiB. The virtual address space is 2^{32} bytes, or 4 GiB, and the physical address space is 2^{30} bytes, which allows main memory of up to 1 GiB. The number of entries in the page table is 2^{20}, or 1 million entries. The valid bit for each entry indicates whether the mapping is legal. If it is off, then the page is not present in memory. Although the page table entry shown here need only be 19 bits wide, it would typically be rounded up to 32 bits for ease of indexing. The extra bits would be used to store additional information that needs to be kept on a per-page basis, such as protection.

Handling Page Faults

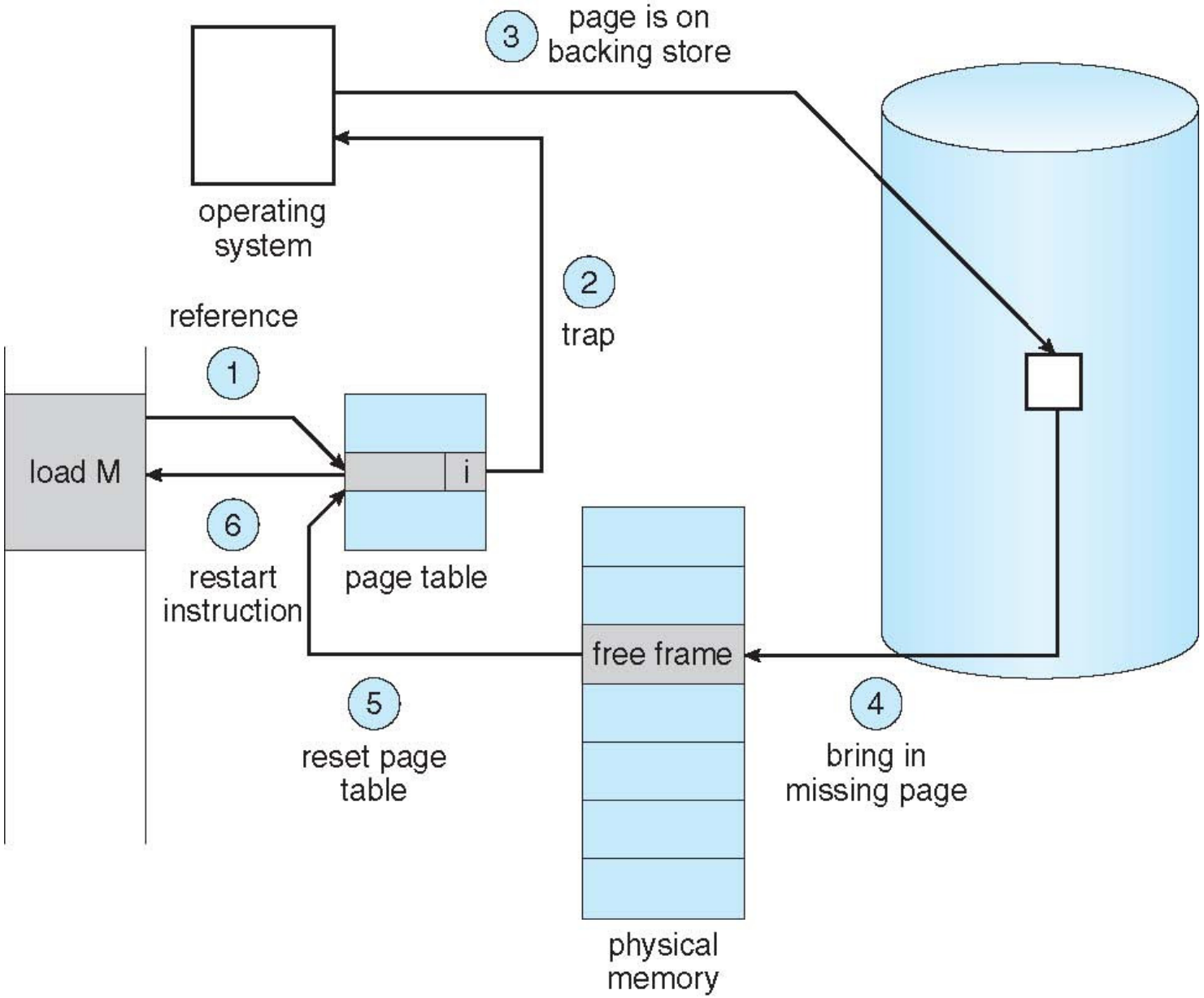
On a page fault, the page must be fetched from the next memory level (usually flash memory or magnetic disks), which can take **millions of clock cycles to process**.

The procedure for handling a page fault is straightforward:

- Find a free frame in memory and bring in the missing page from backing store
- Reset page table to indicate that page is now in memory
- Restart the instruction that caused the page fault

To reduce page fault rate, **fully associative placement of pages in memory** and **smart replacement algorithms** are used together with **large enough page sizes**. Sizes from 4 KiB to 16 KiB are typical today.

Handling Page Faults



Page Replacement

When a page fault occurs, if all pages in main memory are in use, the operating system must choose a page to replace. Because we want to minimize the number of page faults, most operating systems try to **choose a page that will not be needed in the near future**.

Two page transfers are required – one to write the replaced page back to secondary storage and another to bring the faulty page in. This overhead can be reduced by using a **modify (or dirty) bit** per page.

- The dirty bit is set whenever a page in memory is modified
- When a page is selected for replacement, if its dirty bit is unset, the page has not been modified since it was loaded and we can avoid writing it back to secondary storage since it is already there

Page Replacement Algorithms

Several page replacement algorithms exist:

- FIFO – First-In First-Out
- LRU – Least Recently Used
- Second chance
- Clock
- NRU – Not Recently Used
- LFU – Least Frequently Used
- Aging

Translation-Lookaside Buffer

Address translation appears to require extra memory references:

- One to access the page table
- Another to access the actual data

But access to page tables has good locality. Accordingly, modern processors include a fast cache, called translation-lookaside buffer (TLB), that **keeps track of recently used address translations to try to avoid access the page table**.

Translation-Lookaside Buffer

Some typical values for a TLB are:

- 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%

Translation-Lookaside Buffer

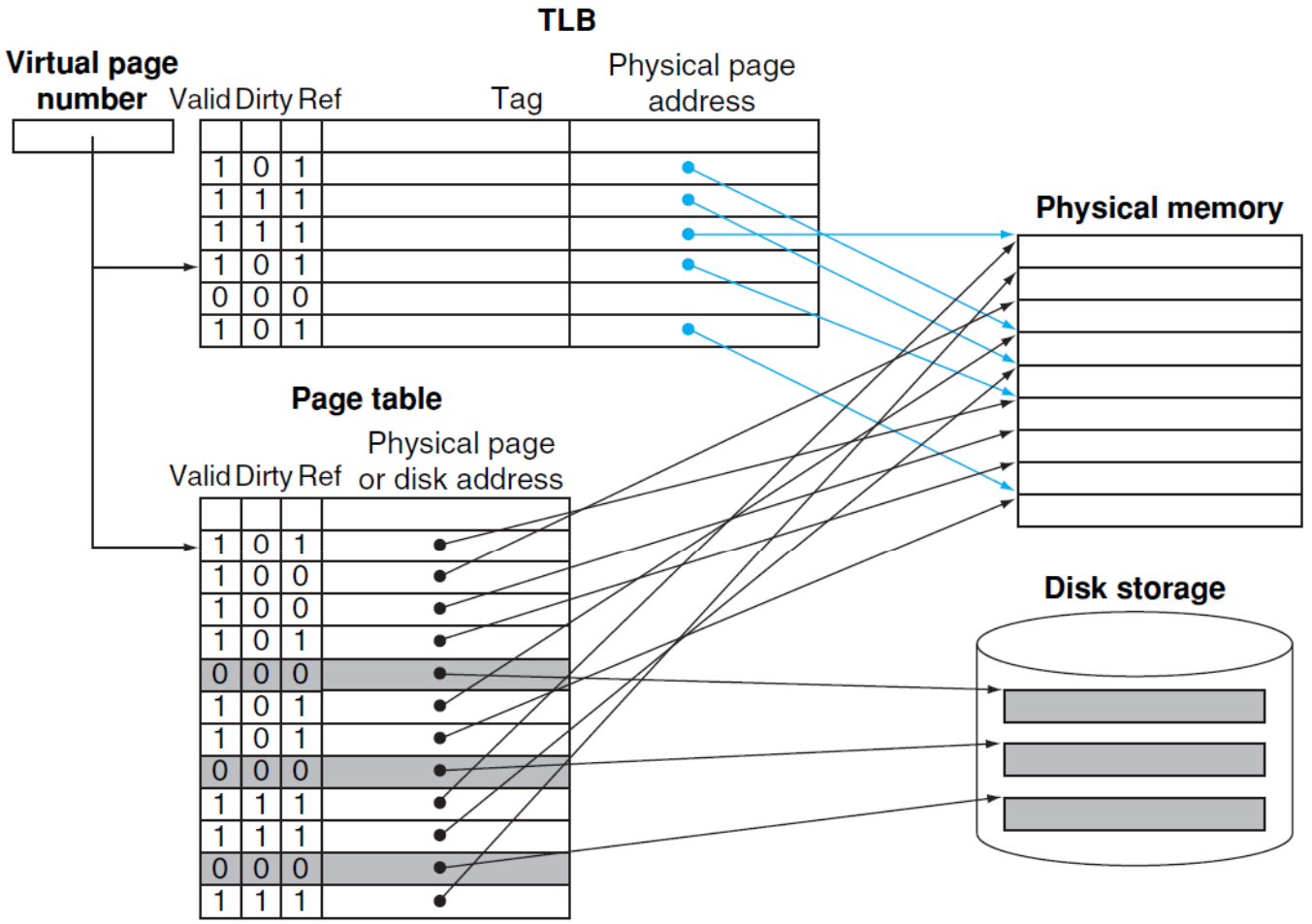
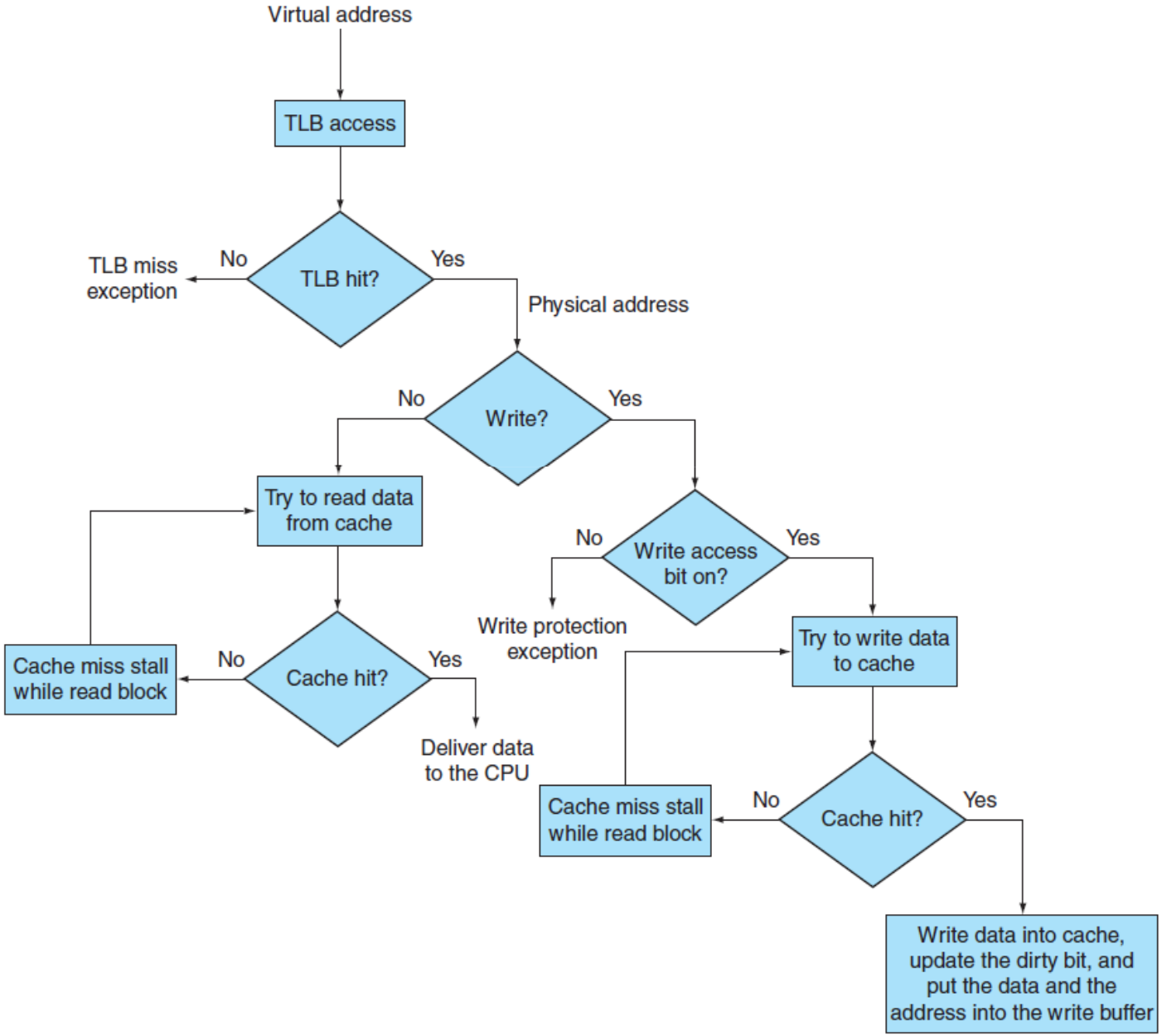


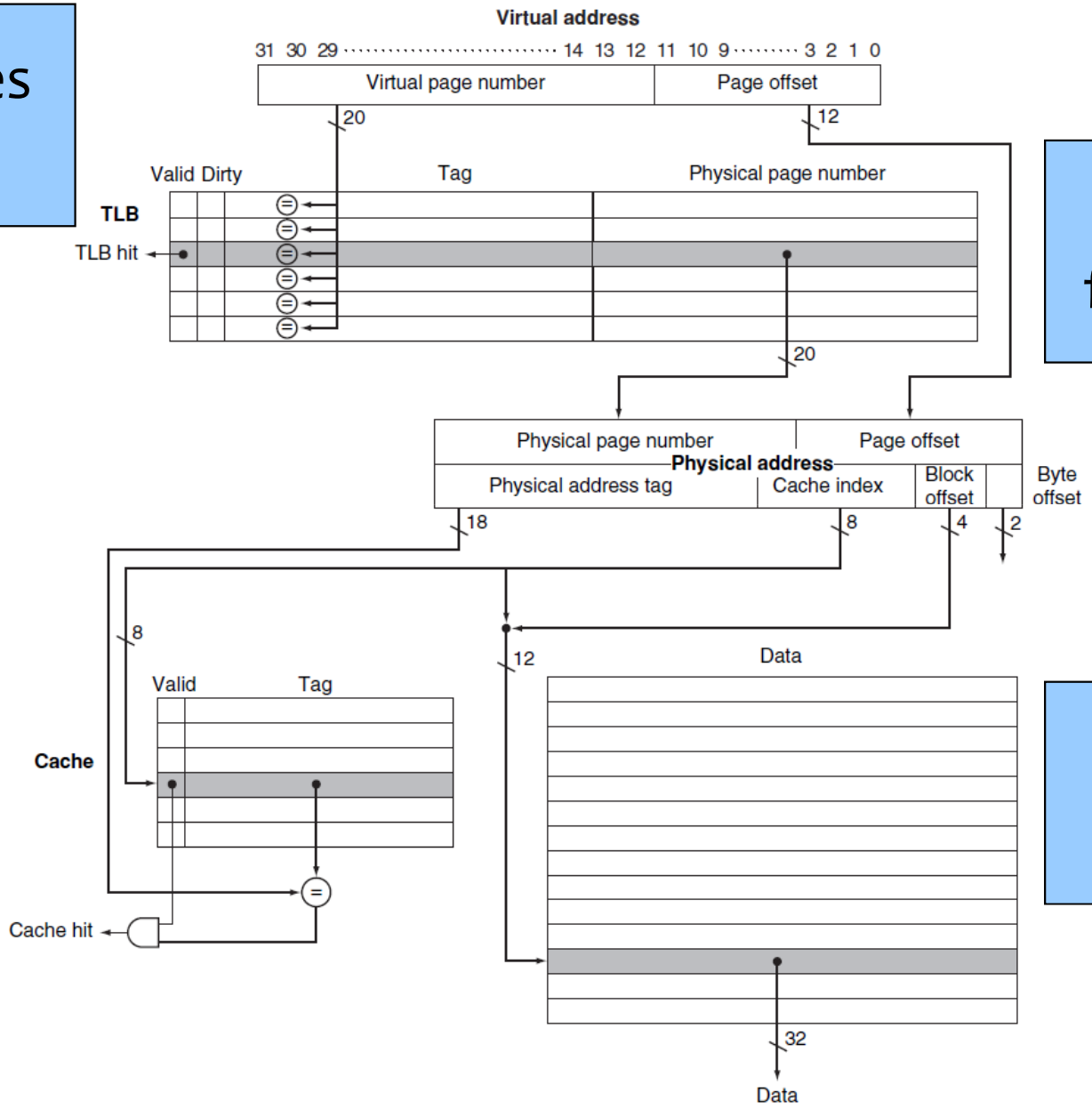
FIGURE 5.29 The TLB acts as a cache of the page table for the entries that map to physical pages only. The TLB contains a subset of the virtual-to-physical page mappings that are in the page table. The TLB mappings are shown in color. Because the TLB is a cache, it must have a tag field. If there is no matching entry in the TLB for a page, the page table must be examined. The page table either supplies a physical page number for the page (which can then be used to build a TLB entry) or indicates that the page resides on disk, in which case a page fault occurs. Since the page table has an entry for every virtual page, no tag field is needed; in other words, unlike a TLB, a page table is *not* a cache.

TLB and Cache Interaction



TLB and Cache Interaction

32-bit addresses
4KiB pages



TLB
fully associative

Cache
direct-mapped

TLB and Cache Interaction

TLB	Page table	Cache	Possible? If so, under what circumstance?
Hit	Hit	Miss	Possible, although the page table is never really checked if TLB hits.
Miss	Hit	Hit	TLB misses, but entry found in page table; after retry, data is found in cache.
Miss	Hit	Miss	TLB misses, but entry found in page table; after retry, data misses in cache.
Miss	Miss	Miss	TLB misses and is followed by a page fault; after retry, data must miss in cache.
Hit	Miss	Miss	Impossible: cannot have a translation in TLB if page is not present in memory.
Hit	Miss	Hit	Impossible: cannot have a translation in TLB if page is not present in memory.
Miss	Miss	Hit	Impossible: data cannot be allowed in cache if the page is not in memory.

FIGURE 5.32 The possible combinations of events in the TLB, virtual memory system, and cache. Three of these combinations are impossible, and one is possible (TLB hit, virtual memory hit, cache miss) but never detected.

Error Detection Code

Richard Hamming invented a popular redundancy scheme for memory, for which he received the Turing Award in 1968.

Hamming used a **parity bit for error detection** – the number of 1s in a word is counted and when the word is written into memory, the parity bit is also written (1 for odd, 0 for even). That is, the parity of the N+1 bit word should always be even.

Then, when the word is read out, the parity bit is read and checked. If the parity of the memory word and the stored parity bit do not match, an error has occurred. If there are 2 bits of error, then a 1-bit parity scheme will not detect any errors.

Error Correction Code

Later, Hamming came up with an easy to understand mapping of data, named **Hamming Error Correction Code (ECC)** in his honor, that uses extra parity bits to allow the **position identification of a single error**:

- Number bits from 1 on the left
- All bit positions that are a power of 2 are parity bits (p1, p2, p4, p8, …)
- Each parity bit checks a subset of data bits as follows

Bit position		1	2	3	4	5	6	7	8	9	10	11	12
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverage	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

Single Error Correcting

Single error correcting (SEC) then determines whether bits are incorrect by looking at the parity bits:

- If all parity bits are 0 then there is no error
- Otherwise, if the pattern is, say <p8,p4,p2,p1> = 1010, which is 10 in decimal, then Hamming ECC tells us that bit 10 (d6) was flipped – the error can be corrected just by inverting bit 10

For example, the SEC code for value 10011010_2 is 011100101010_2. Inverting bit 10 (d6) changes it to 011100101110_2. Looking at the parity bits, we get <p8,p4,p2,p1> = 1010_2 meaning that bit 10 (d6) must be wrong.

Single Error Correcting / Double Error Detecting

At the cost of one more parity bit for the whole SEC code, we can **correct single bit errors and detect double bit errors**.

Let H be the SEC parity bits and P the extra parity bit:

- If H is 0 and P is even then there is no error
- If H is 0 and P is odd then there is an error in bit P
- If H is not 0 and P is odd then a correctable single bit error occurred
- If H is not 0 and P is even then a double error occurred

Single Error Correcting / Double Error Detecting (SEC/DED) is common in memory for servers today. Eight byte (64 bits) data blocks can use SEC/DED with 8 parity bits, which is why many DIMMs are 72 bits wide.