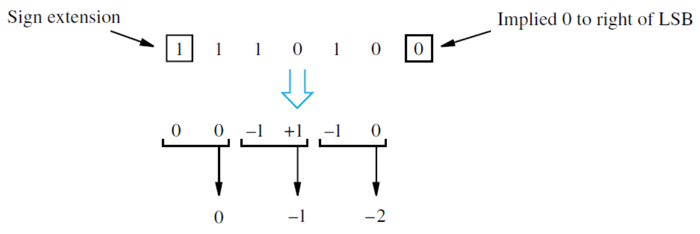
Fast Multiplication method 1: Bit-Pair Recoding of Multipliers:

This technique guarantees that the maximum number of summands (versions of the multiplicand) that must be added is not n but *n*/2 for *n*-bit operands. This technique called *bit-pair recoding* of the multiplier results in using at most one summand for each pair of bits in the multiplier. It is derived directly from the Booth algorithm.



(a) Example of bit-pair recoding derived from Booth recoding

The encoding values are computed as follows:

(-1)*(2^1) + (0)*(2^0) = -2 (-1)*(2^1) + (+1)* (2^0) = -1 (0)*(2^1) + (0))* (2^0) = 0

OR

Other method to compute bit pair values:

Multiplier bit-pair		Multiplier bit on the right	Multiplicand selected at position i
$i+1$	i	$i-1$	
0	0	0	$0 \times M$
0	0	1	$+1 \times M$
0	1	0	$+1 \times M$
0	1	1	$+2 \times M$
1	0	0	$-2 \times M$
1	0	1	$-1 \times M$
1	1	0	$-1 \times M$
1	1	1	$0 \times M$

(b) Table of multiplicand selection decisions

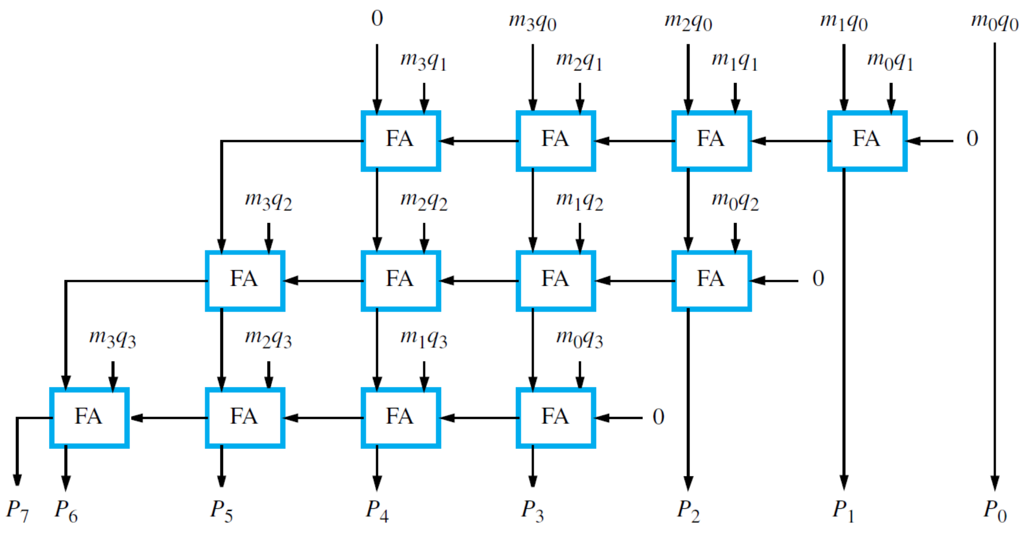
```
               0  1  1  0  1   (+13)
             × 1  1  0  1  0   (−6)
             ---------------

                     ⇓

               0  1  1  0  1
               0 −1 +1 −1  0
               --------------
0  0  0  0  0  0  0  0  0  0
1  1  1  1  1  0  0  1  1
0  0  0  0  1  1  0  1
1  1  1  0  0  1  1
0  0  0  0  0  0
-----------------------------
1  1  1  0  1  1  0  0  1  0   (−78)

                     ⇓

               0  1  1  0  1
               0     −1    −2
               --------------
1  1  1  1  1  0  0  1  1  0
1  1  1  1  0  0  1  1
0  0  0  0  0  0
-----------------------------
1  1  1  0  1  1  0  0  1  0
```

Figure 9.15 Multiplication requiring only $n/2$ summands.

Fast Multiplication method 2:

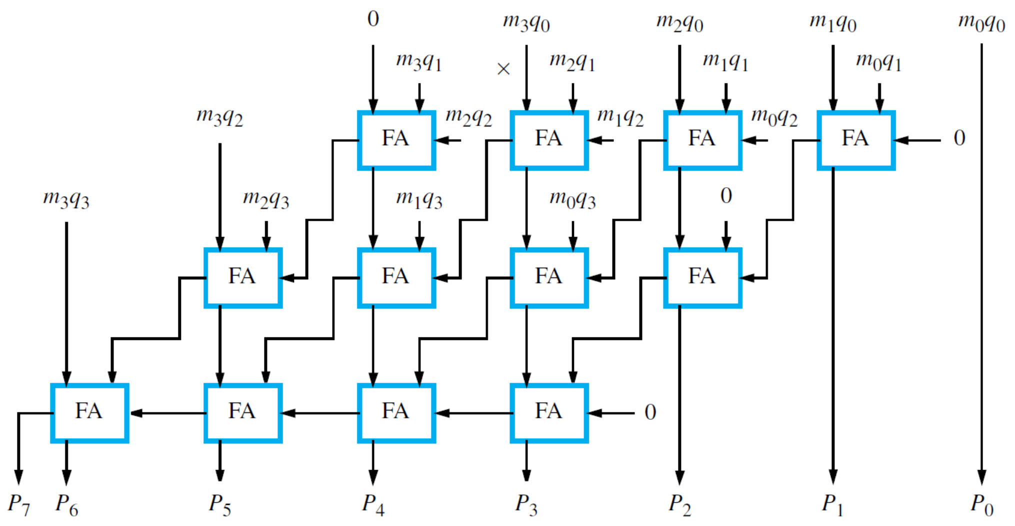
Carry-Save Addition of Summands:

This technique leads to adding the summands in parallel. Multiplication requires the addition of several summands. A technique called *carry-save addition* (CSA) can be used to speed up the process. Consider the 4 × 4 multiplication array shown in Figure 9.16*a*. This structure is in the form of the array shown in Figure in which the first row consists of just the AND gates that produce the four inputs *m*3*q*0, *m*2*q*0, *m*1*q*0, and *m*0*q*0.



(a) Ripple-carry array

Instead of letting the carries ripple along the rows, they can be "saved" and introduced into the next row, at the correct weighted positions, as shown in Figure 9.16*b*. This frees up an input to each of three full adders in the first row. These inputs can be used to introduce the third summand bits *m*2*q*2, *m*1*q*2, and *m*0*q*2. Now, two inputs of each of three full adders in the second row are fed by the sum and carry outputs from the first row. The third input is used to introduce the bits *m*2*q*3, *m*1*q*3, and *m*0*q*3 of the fourth summand. The high-order bits *m*3*q*2 and *m*3*q*3 of the third and fourth summands are introduced into the remaining free full-adder inputs at the left end in the second and third rows. The saved carry bits and the sum bits from the second row are now added in the third row, which is a ripple-carry adder, to produce the final product bits.



(b) Carry-save array

Consider the example shown in Figure 9.17. It involves adding the six shifted versions of the multiplicand for the case of multiplying two, 6-bit, unsigned numbers, where all six bits of the multiplier are equal to 1. The six summands, *A*, *B*, . . . , *F* are added by carry-save addition in Figure 9.18. The blue boxes in these two figures indicate the same operand bits, and show how they are reduced to sum and carry bits in Figure 9.18 by carry-save addition. Three levels of carry-save addition are performed, as shown schematically in Figure 9.19. This figure shows that the final two vectors *S*4 and *C*4 are available in three adder delays after the six input summands are applied to level 1. The final regular addition operation on *S*4 and *C*4, which produces the product, can be done with a carry-lookahead adder.

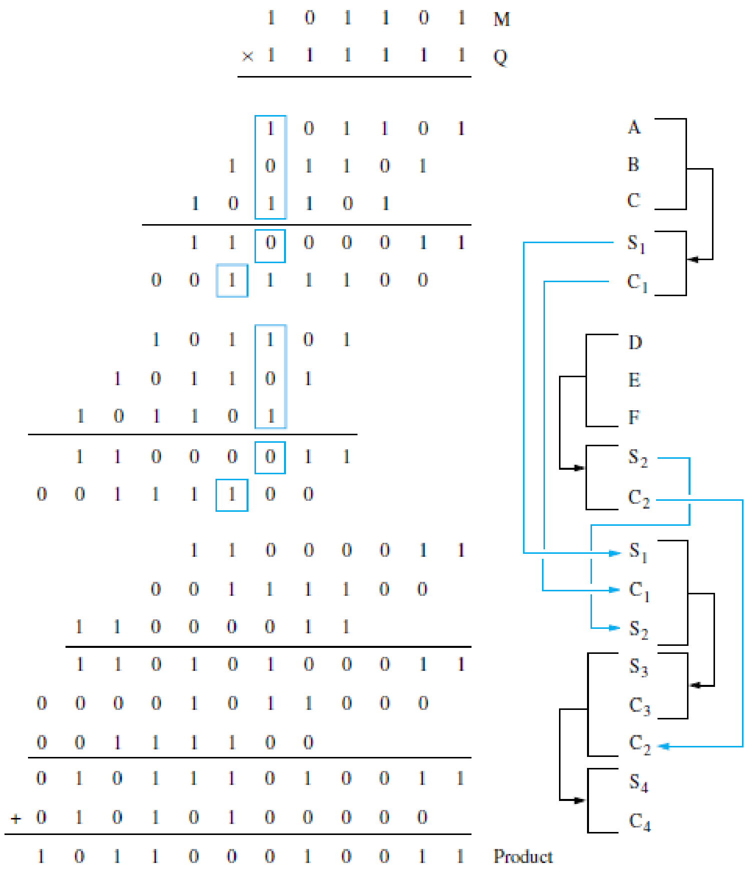


Figure 9.18 The multiplication example from Figure 9.17 performed using carry-save addition.

Integer Division

Figure 9.22 shows examples of decimal division and binary division of the same values. Consider the decimal version first. The 2 in the quotient is determined by the following reasoning: First, we try to divide 13 into 2, and it does not work. Next, we try to divide 13 into 27. We go through the trial exercise of multiplying 13 by 2 to get 26, and, observing that 27 − 26 = 1 is less than 13, we enter 2 as the quotient and perform the required subtraction. The next digit of the dividend, 4, is brought down, and we finish by deciding that 13 goes into 14 once, and the remainder is 1. We can discuss binary division in a similar way, with the simplification that the only possibilities for the quotient bits are 0 and 1.

```
      21                  10101
 13 )274        1101 )100010010
     26                1101
     ---               ----
      14                 10000
      13                  1101
     ---                 -----
       1                   1110
                           1101
                           ----
                              1
```

Figure 9.22 Longhand division examples.

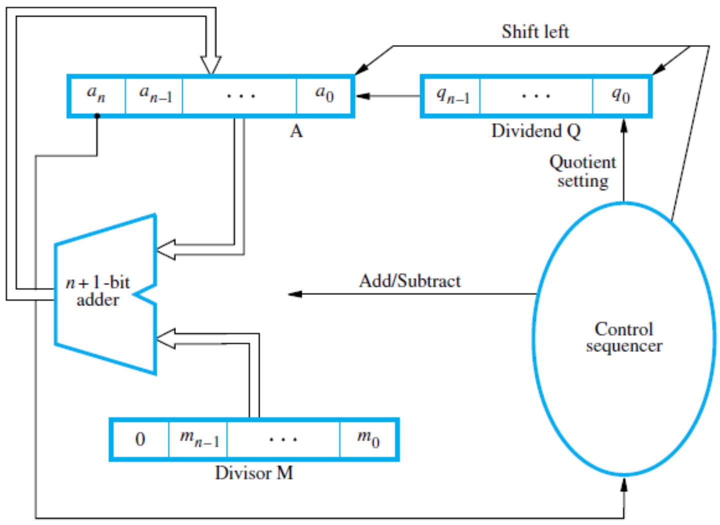


Figure 9.23 Circuit arrangement for binary division.

A circuit that implements division by this longhand method operates as follows: It positions the divisor appropriately with respect to the dividend and performs a subtraction. If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed. If the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is repositioned for another subtraction. This is called the *restoring division* algorithm.

Restoring Division

Figure 9.23 shows a logic circuit arrangement that implements the restoring division algorithm just discussed. Note its similarity to the structure for multiplication shown in Figure 9.7. An *n*-bit positive divisor is loaded into register M and an *n*-bit positive dividend is loaded into register Q at the start of the operation. Register A is set to 0. After the division is complete, the *n*-bit quotient is in register Q and the remainder is in register A. The required subtractions are facilitated by using 2's-complement arithmetic. The extra bit position at the left end of both A and M accommodates the sign bit during subtractions. The following algorithm performs restoring division.

Do the following three steps *n* times:

1. Shift A and Q left one bit position.

2. Subtract M from A, and place the answer back in A.

3. If the sign of A is 1, set *q*0 to 0 and add M back to A (that is, restore A); otherwise, set *q*0 to 1.

Figure 9.24 shows a 4-bit example as it would be processed by the circuit in Figure 9.23.

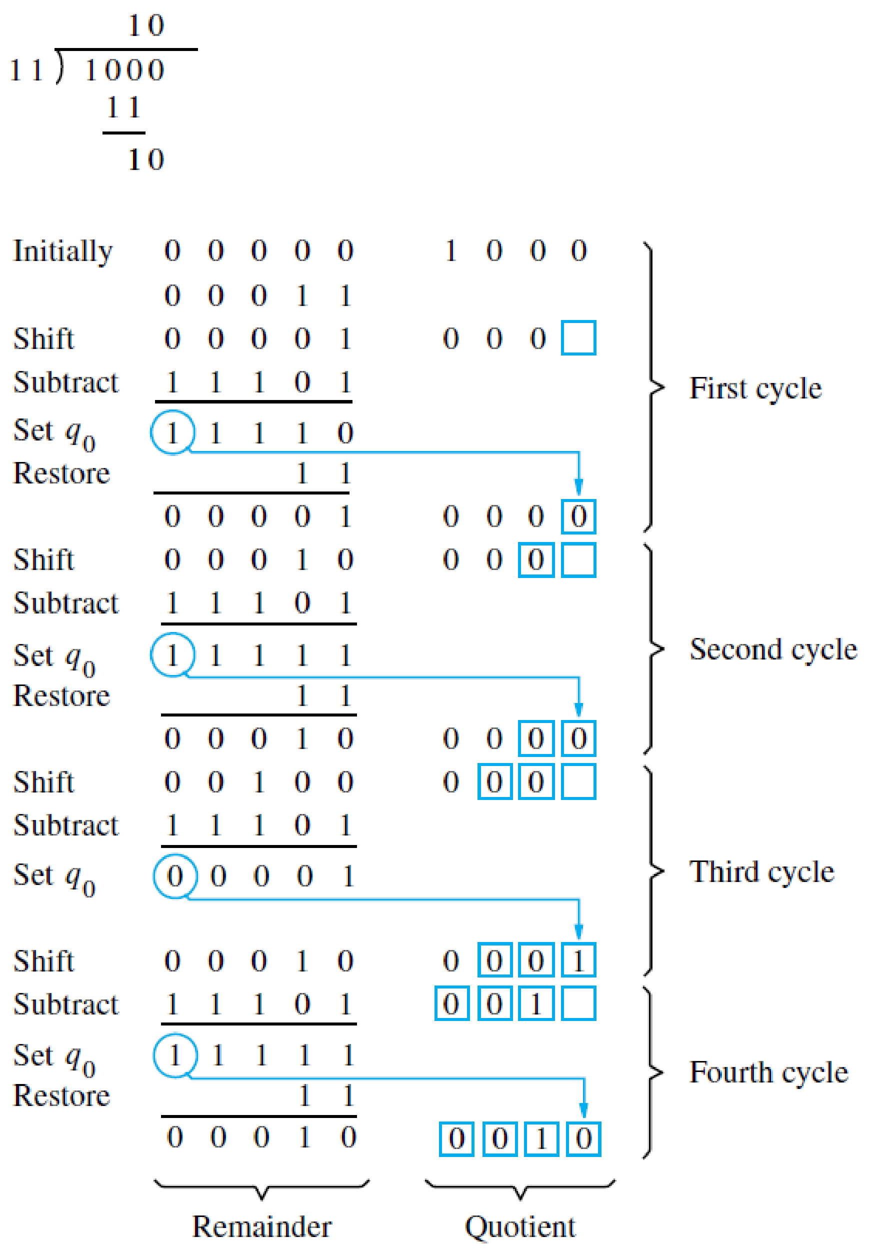


Figure 9.24 A restoring division example.

Non-Restoring Division

The restoring division algorithm can be improved by avoiding the need for restoring A after an unsuccessful subtraction. Subtraction is said to be unsuccessful if the result is negative. Consider the sequence of operations that takes place after the subtraction operation in the preceding algorithm. If A is positive, we shift left and subtract M, that is, we perform 2A− M. IfAis negative, we restore it by performingA+ M, and then we shift it left and subtract M. This is equivalent to performing 2A+ M. The $q0$ bit is appropriately set to 0 or 1 after the correct operation has been performed. We can summarize this in the following algorithm for *non-restoring division*.

Stage 1: Do the following two steps *n* times:

1. If the sign of A is 0, shift A and Q left one bit position and subtract M from A;

otherwise, shift A and Q left and add M to A.

2. Now, if the sign of A is 0, set $q0$ to 1; otherwise, set $q0$ to 0.

Stage 2: If the sign of A is 1, add M to A.

Stage 2 is needed to leave the proper positive remainder in A after the *n* cycles of Stage 1.

The logic circuitry in Figure 9.23 can also be used to perform this algorithm, except that the Restore operations are no longer needed. One Add or Subtract operation is performed in each of the *n* cycles of stage 1, plus a possible final addition in Stage 2. Figure 9.25 shows how the division example in Figure 9.24 is executed by the non-restoring division algorithm.

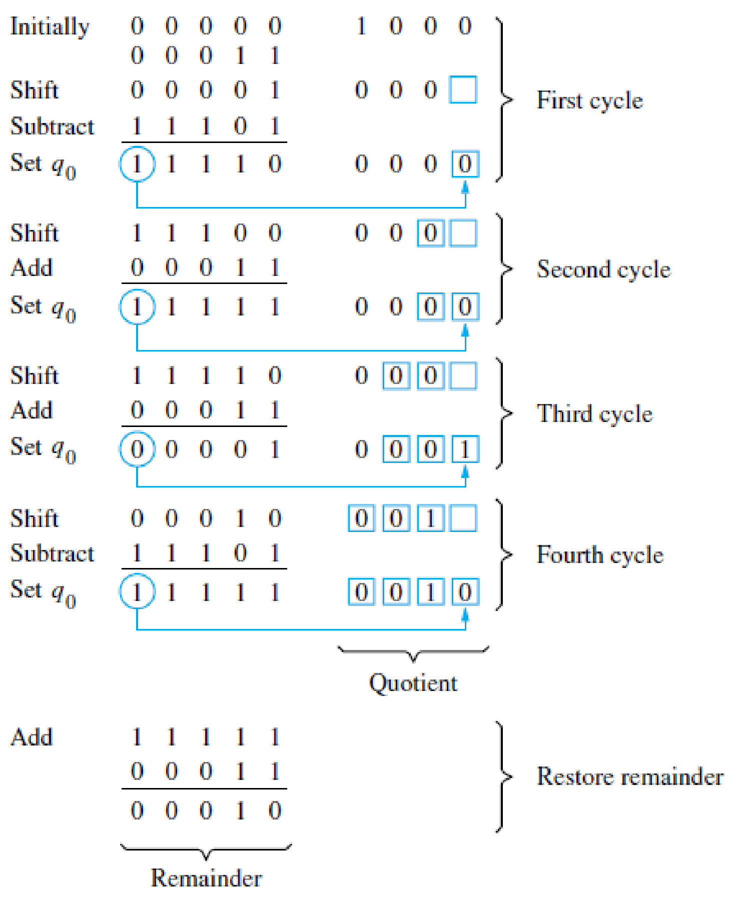


Figure 9.25 A non-restoring division example.

MEMORY SYSTEM

Basic Concepts

The maximum size of the memory that can be used in any computer is determined by the addressing scheme. For example, a computer that generates 16-bit addresses can address up to 2^16 = 64K (kilo) memory locations. Machines whose instructions generate 32-bit addresses can utilize a memory that contains up to 2^32 = 4G (giga) locations, whereas machines with 64-bit addresses can access up to 2^64 = 16E (exa) ≈ 16 × 1018 locations. The number of locations represents the size of the address space of the computer.

The memory is usually designed to store and retrieve data in word-length quantities. Consider, for example, a byte-addressable computer whose instructions generate 32-bit addresses. When a 32-bit address is sent from the processor to the memory unit, the high order 30 bits determine which word will be accessed. If a byte quantity is specified, the low-order 2 bits of the address specify which byte location is involved.

The connection between the processor and its memory consists of address, data, and control lines, as shown in Figure 8.1. The processor uses the address lines to specify the memory location involved in a data transfer operation, and uses the data lines to transfer the data. At the same time, the control lines carry the command indicating a Read or a Write operation and whether a byte or a word is to be transferred. The control lines also provide the necessary timing information and are used by the memory to indicate when it has completed the requested operation. When the processor-memory interface receives the memory's response, it asserts the MFC signal

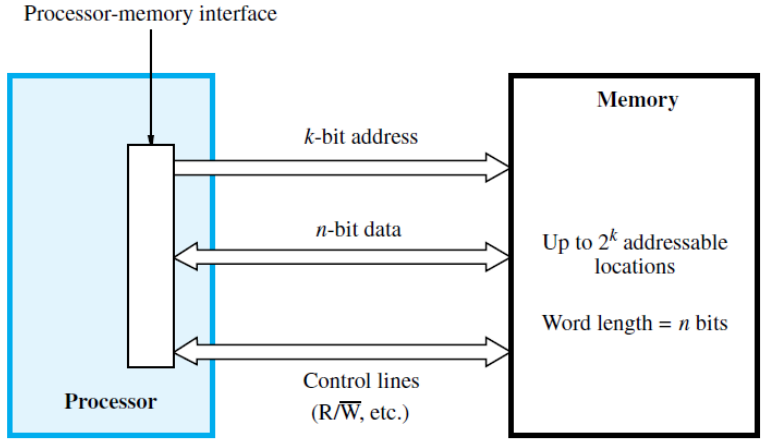


Figure 8.1 Connection of the memory to the processor.

A useful measure of the speed of memory units is the time that elapses between the initiation of an operation to transfer a word of data and the completion of that operation. This is referred to as the ***memory access time***. Another important measure is the ***memory cycle time***, which is the minimum time delay required between the initiation of two successive memory operations, for example, the time between two successive Read operations. The cycle time is usually slightly longer than the access time, depending on the implementation details of the memory unit.

A memory unit is called a ***random-access memory*** **(RAM)** if the access time to any location is the same, independent of the location's address. This distinguishes such memory units from serial, or partly serial, access storage devices such as magnetic and optical disks. Access time of the latter devices depends on the address or position of the data. The technology for implementing computer memories uses semiconductor integrated circuits.

Cache and Virtual Memory

Cache memory. This is a small, fast memory inserted between the larger, slower main memory and the processor. It holds the currently active portions of a program and their data.

Virtual memory is another important concept related to memory organization. With this technique, only the active portions of a program are stored in the main memory, and the remainder is stored on the much larger secondary storage device. Sections of the program are transferred back and forth between the main memory and the secondary storage device in a manner that is transparent to the application program. As a result, the application program sees a memory that is much larger than the computer's physical main memory.

Semiconductor RAM Memories:

Semiconductor random-access memories (RAMs) are available in a wide range of speeds. Their cycle times range from 100 ns to less than 10 ns

Internal Organization of Memory Chips

Memory cells are usually organized in the form of an array, in which each cell can store one bit of information. A possible organization is illustrated in Figure 8.2. Each row of cells constitutes a memory word, and all cells of a row are connected to a common line referred to as the *word line,* which is driven by the address decoder on the chip. The cells in each column are connected to a Sense/Write circuit by two-bit *lines,* and the Sense/Write circuits are connected to the data input/output lines of the chip. During a Read operation, these circuits sense, or read, the information stored in the cells selected by a word line and place this information on the output data lines. During a Write operation, the Sense/Write circuits receive input data and store them in the cells of the selected word.

Organization of 16*8 Memory chip:

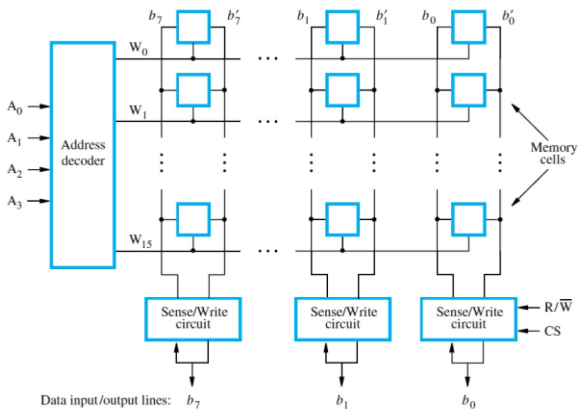


Figure 8.2 Organization of bit cells in a memory chip.

Figure 8.2 is an example of a very small memory circuit consisting of 16 words of 8 bits each. This is referred to as a 16 × 8 organization. The data input and the data output of each Sense/Write circuit are connected to a single bidirectional data line that can be connected to the data lines of a computer. Two control lines, R/W and CS, are provided. The R/W (Read/Write) input specifies the required operation, and the CS (Chip Select) input selects a given chip in a multichip memory system. The memory circuit in Figure 8.2 stores 128 bits and requires 14 external connections for address, data, and control lines. It also needs two lines for power supply and ground connections.

Consider now a slightly larger memory circuit, one that has 1K (1024) memory cells. This circuit can be organized as a 128 × 8 memory, requiring a total of 19 external connections. Alternatively, the same number of cells can be organized into a 1K×1 format. In this The required 10-bit address is divided into two groups of 5 bits each to form the row and column addresses for the cell array. A row address selects a row of 32 cells, all of which are accessed in parallel. But, only one of these cells is connected to the external data line, based on the column address.

Organization of 128*8 or 1k*1 Memory chip:

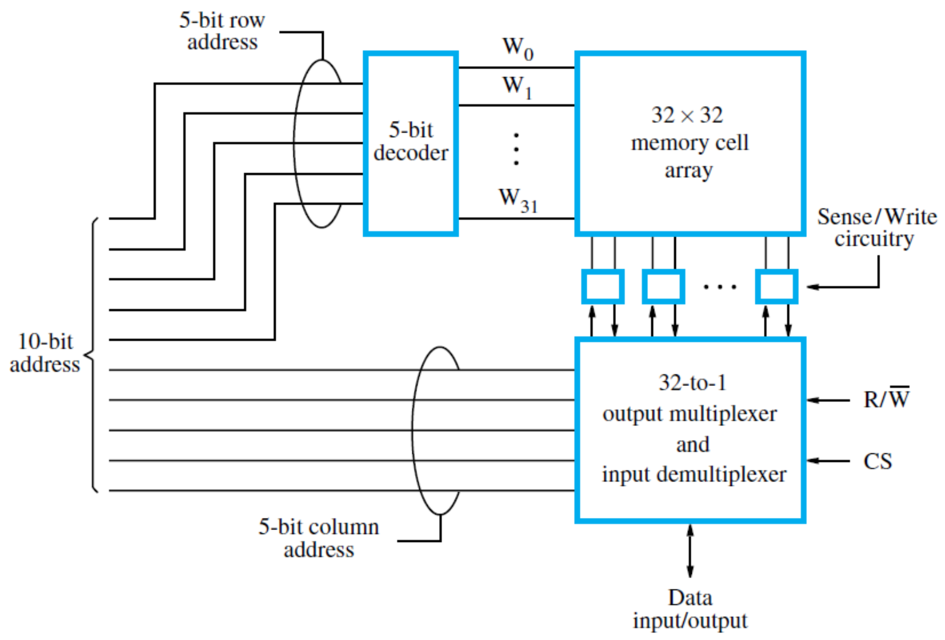


Figure 8.3 Organization of a 1K × 1 memory chip.

Static Memories:

Memories that consist of circuits capable of retaining their state as long as power is applied are known as *static memories. Static RAMs can be accessed very quickly. Access times on the order of a few nanoseconds are found in commercially available chips. SRAMs are used in applications where speed is of critical concern.*

A Static RAM Cell:

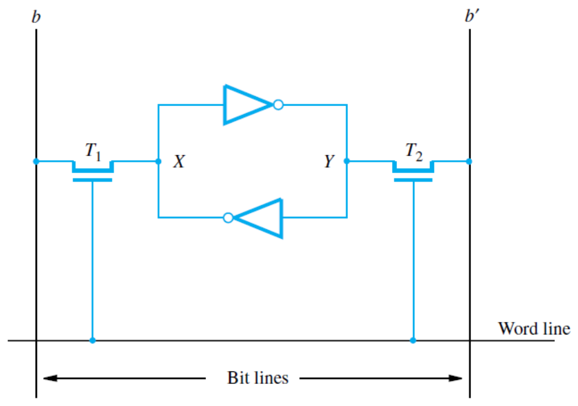


Figure 8.4 A static RAM cell.

Two inverters are cross-connected to form a latch. The latch is connected to two bit lines by transistors *T*1 and *T*2. These transistors act as switches that can be opened or closed under control of the word line. When the word line is at ground level, the transistors are turned off and the latch retains its state.

Read Operation

In order to read the state of the SRAM cell, the word line is activated to close switches $T1$ and $T2$. If the cell is in state 1, the signal on bit line b is high and the signal on bit line $b\_$ is low. The opposite is true if the cell is in state 0. Thus, b and $b\_$ are always complements of each other. The Sense/Write circuit at the end of the two bit lines monitors their state and sets the corresponding output accordingly.

Write Operation

During a Write operation, the Sense/Write circuit drives bit lines b and $b\_$, instead of sensing their state. It places the appropriate value on bit line b and its complement on $b\_$ and activates the word line. This forces the cell into the corresponding state, which the cell retains when the word line is deactivated.

Dynamic RAMs

Static RAMs are fast, but their cells require several transistors. Less expensive and higher density RAMs can be implemented with simpler cells. But, these simpler cells do not retain their state for a long period, unless they are accessed frequently for Read or Write operations. Memories that use such cells are called *dynamic RAMs* (DRAMs). Information is stored in a dynamic memory cell in the form of a charge on a capacitor, but this charge can be maintained for only tens of milliseconds. Since the cell is required to store information for a much longer time, its contents must be periodically *refreshed* by restoring the capacitor charge to its full value. This occurs when the contents of the cell are read or when new information is written into it.

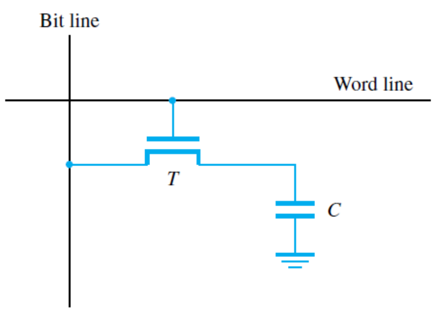


Figure 8.6 A single-transistor dynamic memory cell.

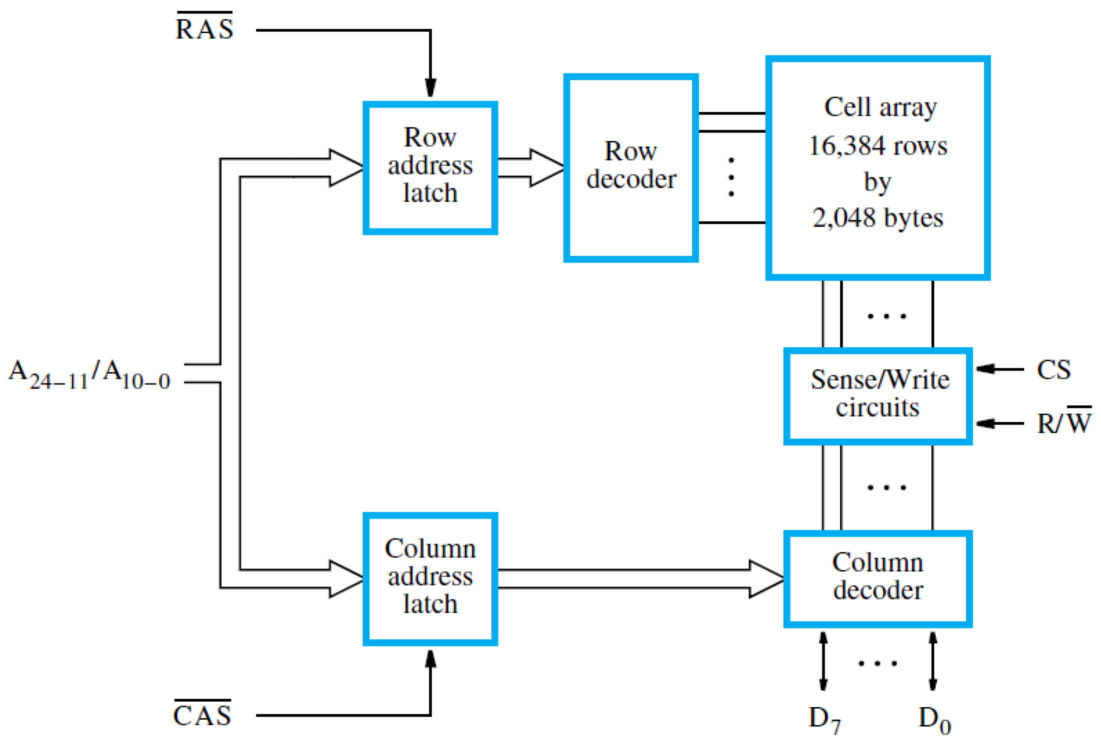


Figure 8.7 Internal organization of a 32M × 8 dynamic memory chip.

A256-Megabit DRAM chip, configured as 32M × 8, is shown in Figure 8.7. The cells are organized in the form of a 16K × 16K array. The 16,384 cells in each row are divided into 2,048 groups of 8, forming 2,048 bytes of data. Therefore, 14 address bits are needed to select a row, and another 11 bits are needed to specify a group of 8 bits in the selected row. In total, a 25-bit address is needed to access a byte in this memory. The high-order 14 bits and the low-order 11 bits of the address constitute the row and column addresses of a byte, respectively. To reduce the number of pins needed for external connections, the row and column addresses are multiplexed on 14 pins. During a Read or aWrite operation, the row address is applied first. It is loaded into the row address latch in response to a signal pulse on an input control line called the Row Address Strobe (RAS). This causes a Read operation to be initiated, in which all cells in the selected row are read and refreshed

All the above examples are Asynchronous

Synchronous DRAMs

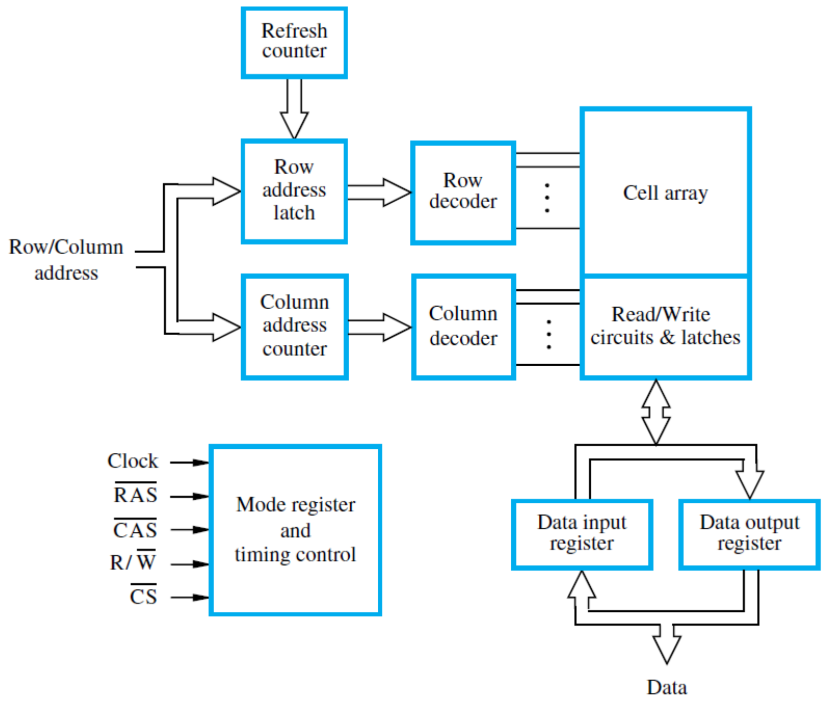


Figure 8.8 Synchronous DRAM.

The cell array is the same as in asynchronous DRAMs. The distinguishing feature of an SDRAM is the use of a clock ignal, the availability of which makes it possible to incorporate control circuitry on the chip hat provides many useful features. For example, SDRAMs have built-in refresh circuitry, with a refresh counter to provide the addresses of the rows to be selected for refreshing. As a result, the dynamic nature of these memory chips is almost invisible to the user.

Read-only Memories

A memory is called a read-only memory, or ROM, when information can be written into it only once at the time of manufacture. Figure 8.11 shows a possible configuration for a ROM cell. A logic value 0 is stored in the cell if the transistor is connected to ground at point *P*; otherwise, a 1 is stored

PROM

Some ROM designs allow the data to be loaded by the user, thus providing a *programmable ROM* (PROM). Programmability is achieved by inserting a fuse at point *P* in Figure 8.11. Before it is programmed, the memory contains all 0s. The user can insert 1s at the required locations by burning out the fuses at these locations using high-current pulses. Of course, this process is irreversible. PROMs provide flexibility and convenience not available with ROMs

EEPROM

An EPROM must be physically removed from the circuit for reprogramming. Also, the stored information cannot be erased selectively. The entire contents of the chip are erased when exposed to ultraviolet light. Another type of erasable PROM can be programmed, erased, and reprogrammed electrically. Such a chip is called an *electrically erasable* PROM, or EEPROM. It does not have to be removed for erasure. Moreover, it is possible to erase the cell contents selectively. One disadvantage of EEPROMs is that different voltages are needed for erasing, writing, and reading the stored data, which increases circuit complexity. However, this disadvantage is outweighed by the many advantages of EEPROMs. They have replaced EPROMs in practice.

Flash Memory

An approach similar to EEPROM technology has given rise to *flash memory* devices. A flash cell is based on a single transistor controlled by trapped charge, much like anEEPROM cell. Also like an EEPROM, it is possible to read the contents of a single cell. The key difference is that, in a flash device, it is only possible to write an entire block of cells. Prior to writing, the previous contents of the block are erased. Flash devices have greater density, which leads to higher capacity and a lower cost per bit. They require a single power supply voltage, and consume less power in their operation.

Memory Hierarchy

a very large amount of cost-effective storage can be provided by magnetic disks, and a large and considerably faster, yet affordable, main memory can be built with dynamic RAM technology. This leaves the more expensive and much faster static RAM technology to be used in smaller units where speed is of the essence, such as in cache memories. All of these different types of memory units are employed effectively in a computer system. The entire computer memory can be viewed as the hierarchy depicted in Figure 8.14.

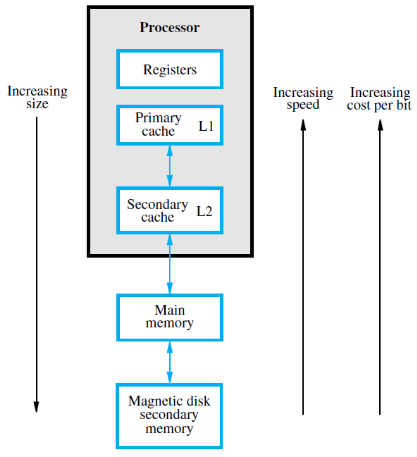


Figure 8.14 Memory hierarchy.

The fastest access is to data held in processor registers. At the next level of the hierarchy is a relatively small amount of memory that can be implemented directly on the processor chip. This memory, called a *processor cache*, holds copies of the instructions and data stored in a much larger memory that is provided externally. A primary cache is always located on the processor chip. This cache is small and its access time is comparable to that of processor registers. The primary cache is referred to as the *level 1* (L1) cache. A larger, and hence somewhat slower, secondary cache is placed between the primary cache and the rest of the memory. It is referred to as the *level 2* (L2) cache. The next level in the hierarchy is the *main memory*. This is a large memory implemented using dynamic memory components. The main memory is much larger but significantly slower than cache memories. In a computer with a processor clock of 2 GHz or higher, the access time for the main memory can be as much as 100 times longer than the access time for the L1 cache.

Disk devices provide a very large amount of inexpensive memory, and they are widely used as secondary storage in computer systems. They are very slow compared to the main memory. They represent the bottom level in the memory hierarchy.

Cache Memories

The cache is a small and very fast memory, interposed between the processor and the main memory. Its purpose is to make the main memory appear to the processor to be much faster than it actually is. The effectiveness of this approach is based on a property of computer programs called locality of reference. the point is that many instructions in localized areas of the program are executed repeatedly during some time period. Temporal locality suggests that whenever an information item, instruction or data, is first needed, this item should be brought into the cache, because it is likely to be needed again soon. Spatial locality suggests that instead of fetching just one item from the main memory to the cache, it is useful to fetch several items that are located at adjacent addresses as well. The term cache block refers to a set of contiguous address locations of some size

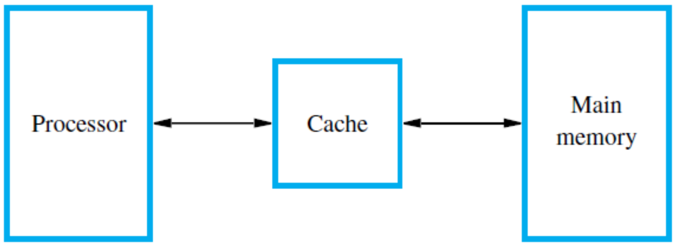


Figure 8.15 Use of a cache memory.

Cache hit/miss

The processor does not need to know explicitly about the existence of the cache. It simply issues Read and Write requests using addresses that refer to locations in the memory. The cache control circuitry determines whether the requested word currently exists in the cache. If it does, the Read or Write operation is performed on the appropriate cache location. In this case, a ***read* or *write hit*** is said to have occurred. A Read operation for a word that is not in the cache constitutes a *Read miss*.

Cache Mapping Functions

There are several possible methods for determining where memory blocks are placed in the cache. It is instructive to describe these methods using a specific small example. Consider a cache consisting of 128 blocks of 16 words each, for a total of 2048 (2K) words, and assume that the main memory is addressable by a 16-bit address. The main memory has 64K words, which we will view as 4K blocks of 16 words each. For simplicity, we have assumed that consecutive addresses refer to consecutive words.

Direct Mapping

The simplest way to determine cache locations in which to store memory blocks is the *direct-mapping* technique. In this technique, block *j* of the main memory maps onto block *j* modulo 128 of the cache, as depicted in Figure 8.16. Thus, whenever one of the main memory blocks 0, 128, 256, . . . is loaded into the cache, it is stored in cache block 0. Blocks 1, 129, 257, . . . are stored in cache block 1, and so on. Since more than one memory block is mapped onto a given cache block position, contention may arise for that position even when the cache is not full.

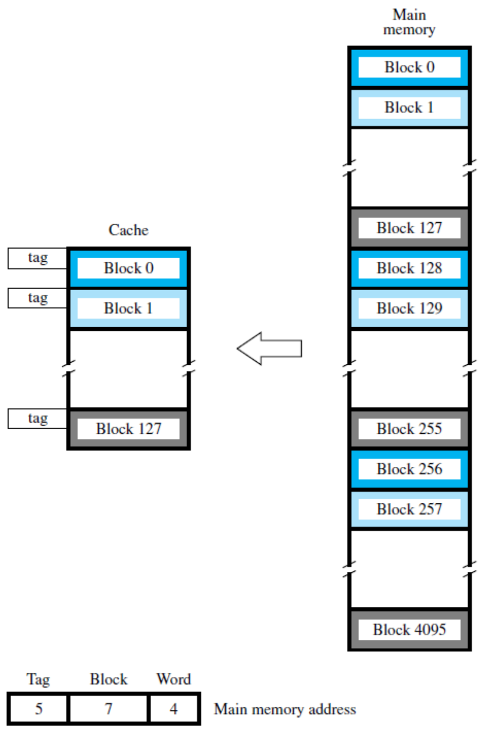


Figure 8.16 Direct-mapped cache.

Contention is resolved by allowing the new block to overwrite the currently resident block. With direct mapping, the replacement algorithm is trivial. Placement of a block in the cache is determined by its memory address. The memory address can be divided into three fields, as shown in Figure 8.16. The low-order 4 bits select one of 16 words in a block. When a new block enters the cache, the 7-bit cache block field determines the cache position in which this block must be stored. The high-order 5 bits of the memory address of the block are stored in 5 *tag* bits associated with its location in the cache. The tag bits identify which of the 32 main memory blocks mapped into this cache position is currently resident in the cache. As execution proceeds, the 7-bit cache block field of each address generated by the processor points to a particular block location in the cache. The high-order 5 bits of the address are compared with the tag bits associated with that cache location. If they match, then the desired word is in that block of the cache. If there is no match, then the block containing the required word must first be read from the main memory and loaded into the cache. The direct-mapping technique is easy to implement, but it is not very flexible.

Associative Mapping

most flexible mapping method, in which a main memory block can be placed into any cache block position. In this case, 12 tag bits are required to identify

a memory block when it is resident in the cache. The tag bits of an address received from the processor are compared to the tag bits of each block of the cache to see if the desired block is present. This is called the *associative-mapping* technique. It gives complete freedom in choosing the cache location in which to place the memory block, resulting in a more efficient use of the space in the cache. When a new block is brought into the cache, it replaces (ejects) an existing block only if the cache is full.

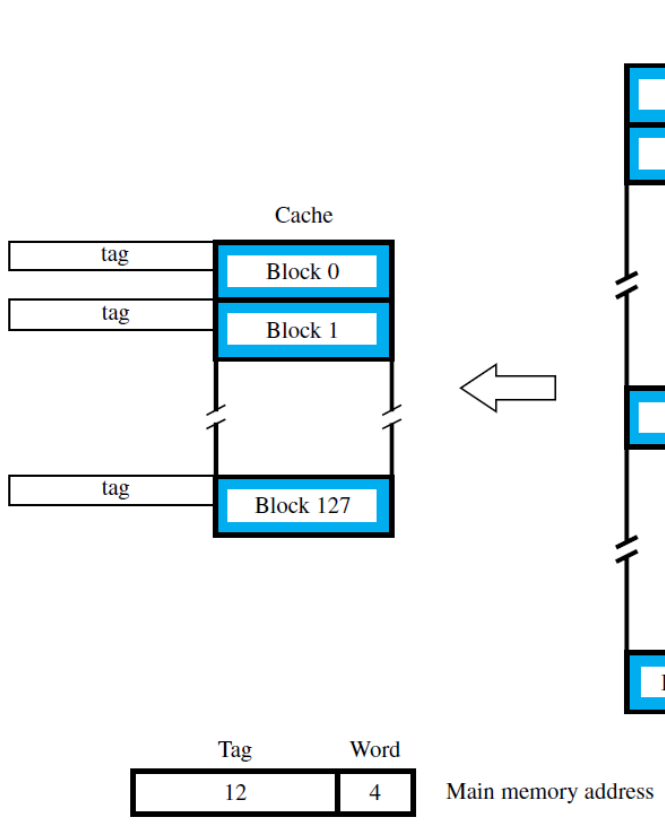


Figure 8.17 Associative-mapped cache.

Set-Associative Mapping

Another approach is to use a combination of the direct- and associative-mapping techniques. The blocks of the cache are grouped into sets, and the mapping allows a block of the main memory to reside in any block of a specific set. Hence, the contention problem of the direct method is eased by having a few choices for block placement. At the same time, the hardware cost is reduced by decreasing the size of the associative search. An example of this *set-associative-mapping* technique is shown in Figure 8.18 for a cache with two blocks per set. In this case, memory blocks 0, 64, 128, . . . , 4032 map into cache set 0, and they can occupy either of the two block positions within this set. Having 64 sets means that the 6-bit set field of the address determines which set of the cache might contain the desired block. The tag field of the address must then be associatively compared to the tags of the two blocks of the set to check if the desired block is present. This two-way associative search is simple to implement.

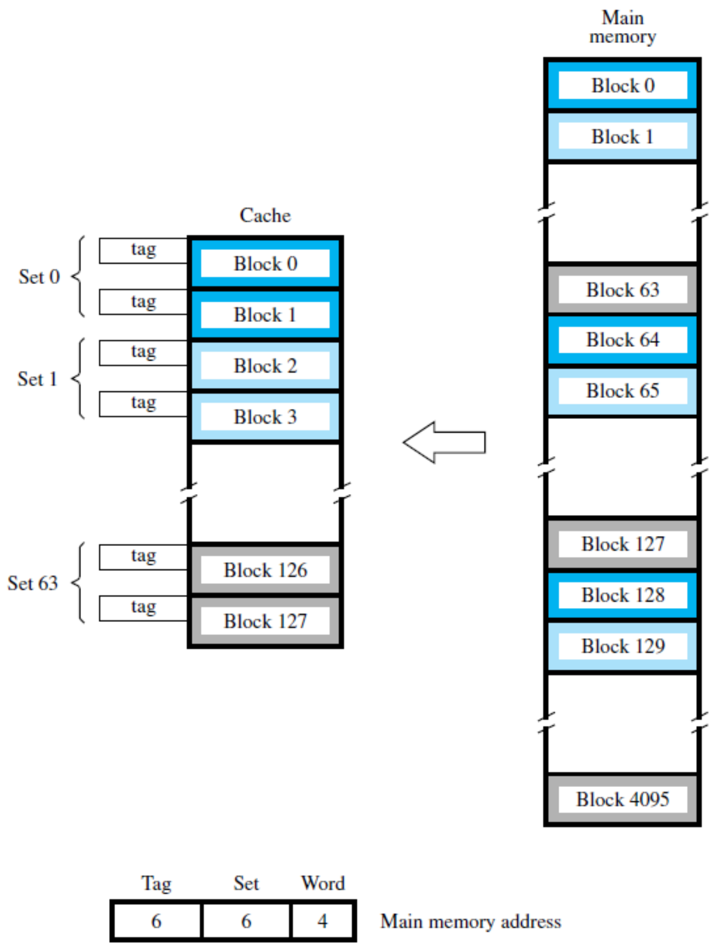


Figure 8.18 Set-associative-mapped cache with two blocks per set.

Virtual Memory

If a program does not completely fit into the main memory, the parts of it not currently being executed are stored on a secondary storage device, typically a magnetic disk. As these parts are needed for execution, they must first be brought into the main memory, possibly replacing other parts that are already in the memory. These actions are performed automatically by the operating system, using a scheme known as *virtual memory*. Application programmers need not be aware of the limitations imposed by the available main memory

a scheme known as *virtual memory*. Application programmers need not be aware of the limitations imposed by the available main memory. They prepare programs using the entire address space of the processor. Under a virtual memory system, programs, and hence the processor, reference instructions and data in an address space that is independent of the available physical main memory space. The binary addresses that the processor issues for either instructions or data are called *virtual* or *logical addresses*. These addresses are translated into physical addresses by a combination of hardware and software actions The binary addresses that the processor issues for either instructions or data are called *virtual* or *logical addresses*. These addresses are translated into physical addresses by a combination of hardware and software actions. If a virtual address refers to a part of the program or data space that is currently in the physical memory, then the contents of the appropriate location in the main memory are accessed immediately. Otherwise, the contents of the referenced address must be brought into a suitable location in the memory before they can be used.

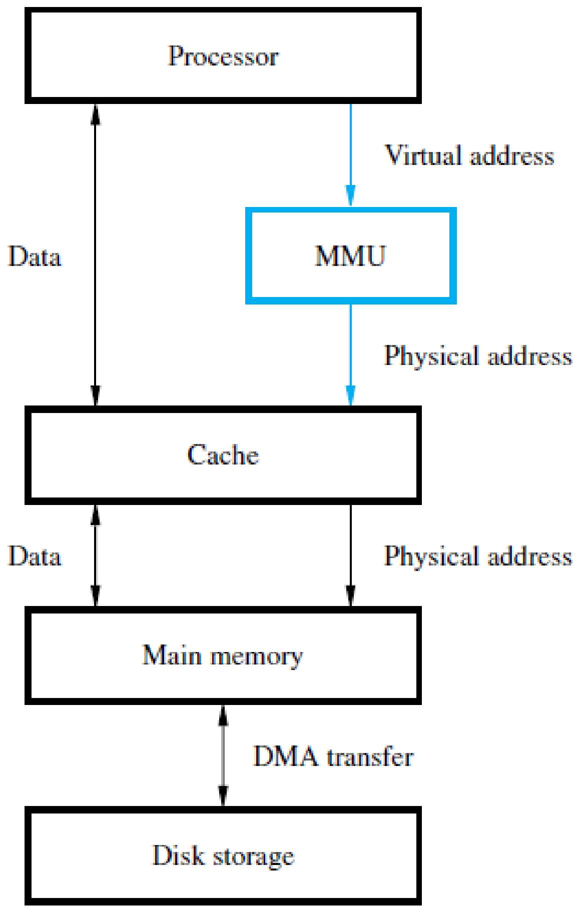


Figure 8.24 Virtual memory organization.

Address Translation

A simple method for translating virtual addresses into physical addresses is to assume that all programs and data are composed of fixed-length units called *pages*, each of which consists of a block of words that occupy contiguous locations in the main memory. Pages commonly range from 2K to 16K bytes in length. They constitute the basic unit of information that is transferred between the main memory and the disk whenever the MMU determines that a transfer is required. Pages should not be too small, because the access time of a magnetic disk is much longer (several milliseconds) than the access time of the main memory

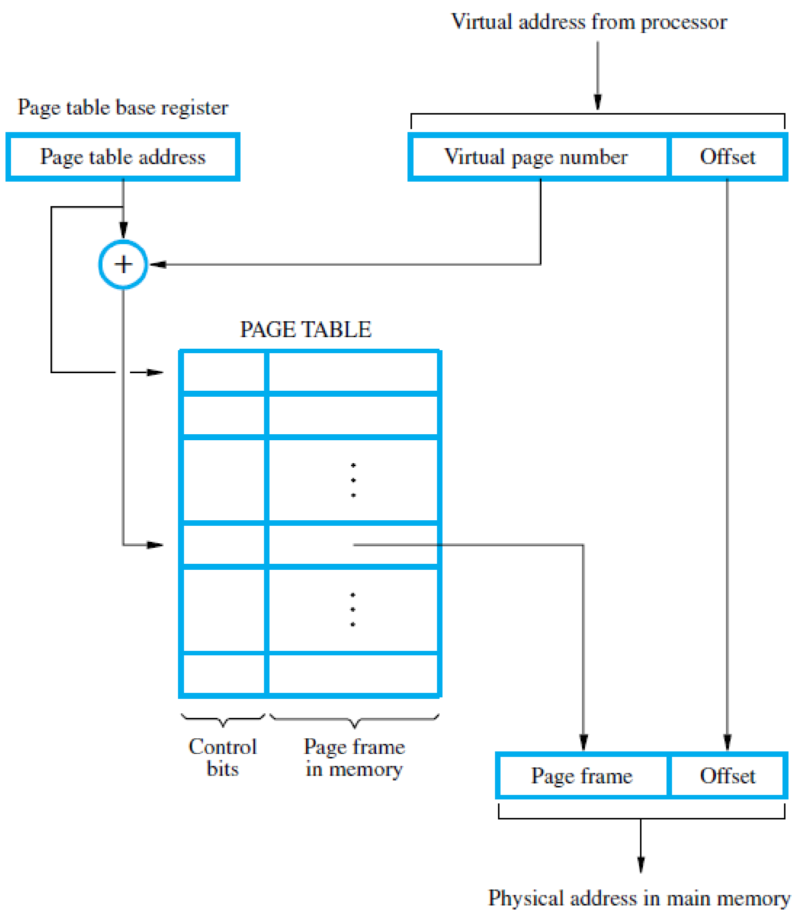


Figure 8.25 Virtual-memory address translation.

A virtual-memory address-translation method based on the concept of fixed-length pages is shown schematically in Figure 8.25. Each virtual address generated by the processor, whether it is for an instruction fetch or an operand load/store operation, is interpreted as a *virtual page number* (high-order bits) followed by an *offset* (low-order bits) that specifies the location of a particular byte (or word) within a page. Information about the main memory location of each page is kept in a *page table*. This information includes the main memory address where the page is stored and the current status of the page. An area in the main memory that can hold one page is called a *page frame*. The starting address of the page table is kept in a *page table base register*. By adding the virtual page number to the contents of this register, the address of the corresponding entry in the page table is obtained. The contents of this location give the starting address of the page if that page currently resides in the main memory. Each entry in the page table also includes some control bits that describe the status of the page while it is in the main memory. One bit indicates the validity of the page, that is, whether the page is actually loaded in the main memory. It allows the operating system to invalidate the page without actually removing it. Another bit indicates whether the page has been modified during its residency in the memory.

Translation Lookaside Buffer

The page table information is used by the MMU for every read and write access. Ideally, the page table should be situated within the MMU. Unfortunately, the page table may be rather large. Since the MMU is normally implemented as part of the processor chip, it is impossible to include the complete table within the MMU. Instead, a copy of only a small portion of the table is accommodated within the MMU, and the complete table is kept in the main memory. The portion maintained within the MMU consists of the entries corresponding to the most recently accessed pages. They are stored in a small table, usually called the *Translation Lookaside Buffer* (TLB). The TLB functions as a cache for the page table in the main memory. Each entry in the TLB includes a copy of the information in the corresponding entry in the page table. In addition, it includes the virtual address of the page, which is needed to search the TLB for a particular page. Figure 8.26 shows a possible organization of aTLB that uses the associative-mapping technique. Set-associative mapped TLBs are also found in commercial products. Address translation proceeds as follows. Given a virtual address, the MMU looks in the TLB for the referenced page. If the page table entry for this page is found in the TLB, the physical address is obtained immediately. If there is a miss in the TLB, then the required entry is obtained from the page table in the main memory and the TLB is updated.

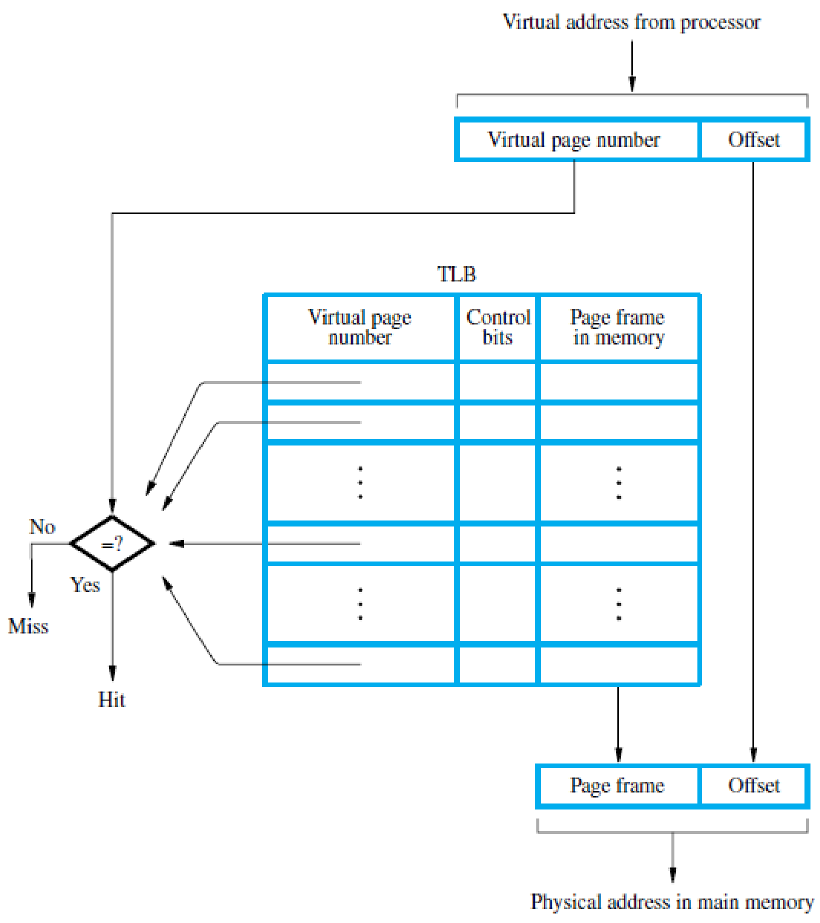


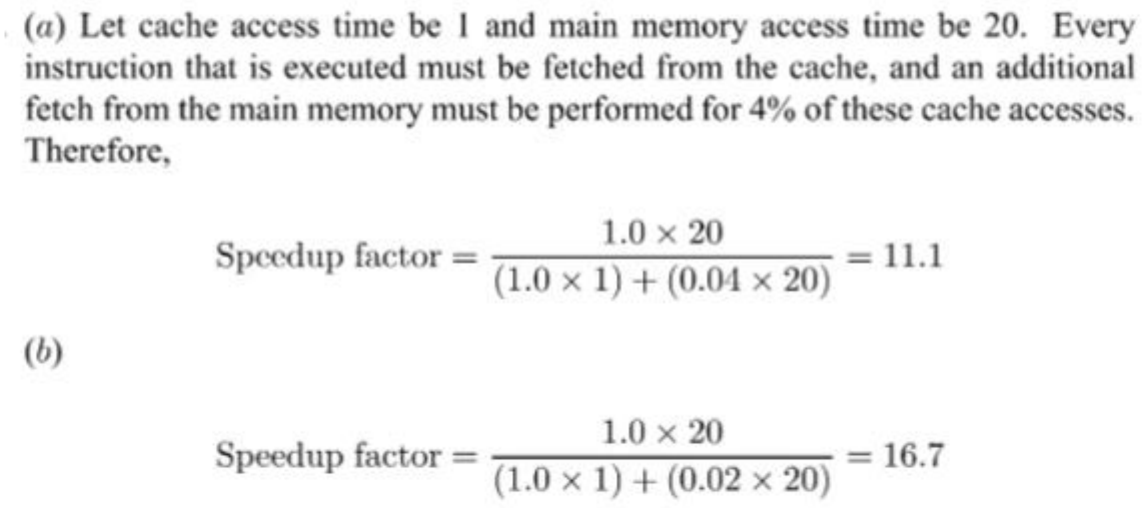
Figure 8.26 Use of an associative-mapped TLB

Performance Considerations

Let the cache access time be 1 unit and the main memory access time be 20 units.
Every instruction that is executed must be fetched from the cache, and an additional fetch from the main memory must be performed for 4% of these cache accesses.

(a) Calculate the speedup factor of the system with cache compared to a system without cache.
(b) If the miss rate is reduced to 2%, compute the new speedup factor.

Solution:



(*a*) Let cache access time be 1 and main memory access time be 20. Every instruction that is executed must be fetched from the cache, and an additional fetch from the main memory must be performed for 4% of these cache accesses. Therefore,

$$\text{Speedup factor} = \frac{1.0 \times 20}{(1.0 \times 1) + (0.04 \times 20)} = 11.1$$

(*b*)

$$\text{Speedup factor} = \frac{1.0 \times 20}{(1.0 \times 1) + (0.02 \times 20)} = 16.7$$

Example 5.2 Consider now the impact of the cache on the overall performance of the computer. Let h be the hit rate, M the miss penalty, that is, the time to access information in the main memory, and C the time to access information in the cache. The average access time experienced by the processor is

$$t_{ave} = hC + (1 - h)M$$

We use the same parameters as in Example 5.1. If the computer has no cache, then, using a fast processor and a typical DRAM main memory, it takes 10 clock cycles for each memory read access. Suppose the computer has a cache that holds 8-word blocks and an interleaved main memory. Then, as we showed in Section 5.6.1, 17 cycles are

needed to load a block into the cache. Assume that 30 percent of the instructions in a typical program perform a read or a write operation, which means that there are 130 memory accesses for every 100 instructions executed. Assume that the hit rates in the cache are 0.95 for instructions and 0.9 for data. Let us further assume that the miss penalty is the same for both read and write accesses. Then, a rough estimate of the improvement in performance that results from using the cache can be obtained as follows:

$$\frac{\textit{Time without cache}}{\textit{Time with cache}} = \frac{130 \times 10}{100(0.95 \times 1 + 0.05 \times 17) + 30(0.9 \times 1 + 0.1 \times 17)} = 5.04$$

This result suggests that the computer with the cache performs five times better.