2
Von Neumann Architecture

2.1 INTRODUCTION

Computer architecture has undergone incredible changes in the past 20 years, from the number of circuits that can be integrated onto silicon wafers to the degree of sophistication with which different algorithms can be mapped directly to a computer's hardware. One element has remained constant throughout the years, however, and that is the von Neumann concept of computer design.

The basic concept behind the von Neumann architecture is the ability to store program instructions in memory along with the data on which those instructions operate. Until von Neumann proposed this possibility, each computing machine was designed and built for a single predetermined purpose. All programming of the machine required the manual rewiring of circuits, a tedious and error-prone process. If mistakes were made, they were difficult to detect and hard to correct.

Von Neumann architecture is composed of three distinct components (or sub-systems): a central processing unit (CPU), memory, and input/output (I/O) interfaces. Figure 2.1 represents one of several possible ways of interconnecting these components.

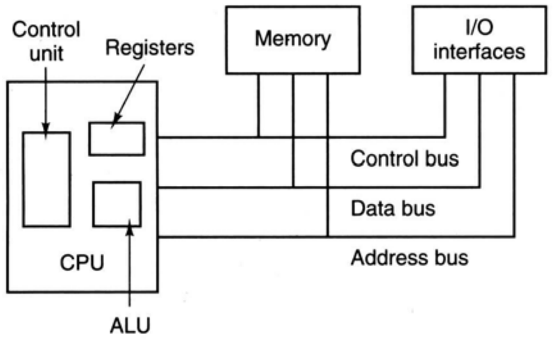


Figure 2.1 Basic Computer Components.

1. The CPU, which can be considered the heart of the computing system, includes three main components: the *control unit* (CU), one or more *arithmetic logic units* (ALUs), and various *registers*. The control unit determines the order in which instructions should be executed and controls the retrieval of the proper operands. It interprets the instructions of the machine. The execution of each instruction is determined by a sequence of control signals produced by the control unit. In other words, the control unit governs the flow of information through the system by issuing control signals to different components. Each operation caused by a control signal is called a microoperation (MO). ALUs perform all mathematical and Boolean operations. The registers are temporary storage locations to quickly store and transfer the data and instructions being used. Because the registers are often on the same chip and directly connected to the CU, the registers have faster access time than memory. Therefore, using registers both as the source of operands and as the destination of results will improve the performance. A CPU that is implemented on a single chip is called a *microprocessor*.

2. The computer's *memory* is used to store program instructions and data. Two of the commonly used type of memories are *RAM* (random-access memory) and *ROM* (read-only memory). RAM stores the data and general-purpose programs that the machine executes. RAM is temporary; that is, its contents can be changed at any time and it is erased when power to the computer is turned off. ROM is permanent and is used to store the initial boot up instructions of the machine.

3. The *I/O interfaces* allow the computer's memory to receive information and send data to output devices. Also, they allow the computer to communicate to the user and to secondary storage devices like disk and tape drives.

The preceding components are connected to each other through a collection of signal lines known as a *bus*. As shown in Figure 2.1, the main buses carrying information are the *control bus*, *data bus*, and *address bus*. Each bus contains several wires that allow for the parallel transmission of information between various hardware components. The address bus identifies either a memory location or an I/O device. The data bus, which is bidirectional, sends data to or from a component. The control bus consists of signals that permit the CPU to communicate with the memory and I/O devices.

The execution of a program in a von Neumann machine requires the use of the three main components just described. Usually, a software package, called an *operating system*, controls how these three components work together. Initially, a program has to be loaded into the memory. Before being loaded, the program is usually stored on a secondary storage device (like a disk). The operating system uses the I/O interfaces to retrieve the program from secondary storage and load it into the memory.

Once the program is in memory, the operating system then schedules the CPU to begin executing the program instructions. Each instruction to be executed must first be retrieved from memory. This retrieval is referred to as an *instruction fetch*. After an instruction is fetched, it is put into a special register in the CPU, called the *instruction register* (IR). While in the IR, the instruction is decoded to determine what type of operation should be performed. If the instruction requires operands, these are fetched from memory or possibly from other registers and placed into the proper location (certain registers or specially designated storage areas known as *buffers*). The instruction is then performed, and the results are stored back into memory and/or registers. This process is repeated for each instruction of the program until the program's end is reached.

This chapter describes the typical implementation techniques used in von Neumann machines. The main components of a von Neumann machine are explained in the following sections. To make the function of the components in the von Neumann architecture and their interactions clear, the design of a simple microcomputer is discussed in the next section. In later sections, various design techniques for each component are explained in detail. Elements of a datapath, as well as the hardwired and microprogramming techniques for implementing control functions, are discussed. Next, a hierarchical memory system is presented. The architectures of a memory cell, interleaved memory, an associative memory, and a cache memory are given. Virtual memory is also discussed. Finally, interrupts and exception events are addressed.

2.2 DESIGN OF A SIMPLE MICROCOMPUTER USING VHDL

A computer whose CPU is a microprocessor is called a *microcomputer*. Microcomputers are small and inexpensive. Personal computers are usually microcomputers. Figure 2.2 represents the main components of a simple microcomputer. This microcomputer contains a CPU, a clock generator, a decoder, and two memory modules. Each memory module consists of 8 words, each of which has 8 bits. (A *word* indicates how much data a computer can process at any one time.) Since there are two memory modules, this microcomputer's memory consists of a total of sixteen 8-bit memory words. The address bus contains 4 bits in order to address these 16 words. The three least significant bits of the address bus are directly connected to the memory modules, whereas the most significant (leftmost) bit is connected to the select line of the decoder (S). When this bit is 0, M0 is chosen; when it is 1, M1 is chosen. (See Appendix B for information on decoders.) In this way, the addresses 0 to 7 (**0**000 to **0**111) refer to the words in memory module M0, and the addresses 8 to 15 (**1**000 to **1**111) refer to memory module M1. Figure 2.3 represents a structural view of our microcomputer in VHDL. (See Appendix C for information on VHDL. If you are not familiar with VHDL, you can skip this figure and also later VHDL descriptions.) The structural view describes our system by declaring its main components and connecting them with a set of signals. The structure_view is divided into two parts: the declaration part, which appears before the keyword *begin*, and the design part, which appears after *begin*. The declaration part consists of four component statements and two signal statements. Each component statement defines the input/output ports of each component of the microcomputer. The signal statements define a series of signals that are used for interconnecting the components. For example, the 2-bit signal *M* is used to connect the outputs of the decoder to the chip select (CS) lines of the memory modules. The design part includes a set of component instantiation statements. A component instantiation statement creates an instance of a component. An instance starts with a label followed by the component name and a portmap. Each entry of the portmap refers to one of the component's ports or a locally declared signal. A port of a component is connected to a port of another component if they have the same portmap entry. For instance, the CPU and memory unit M0 are connected because they both contain DATA as a portmap entry.

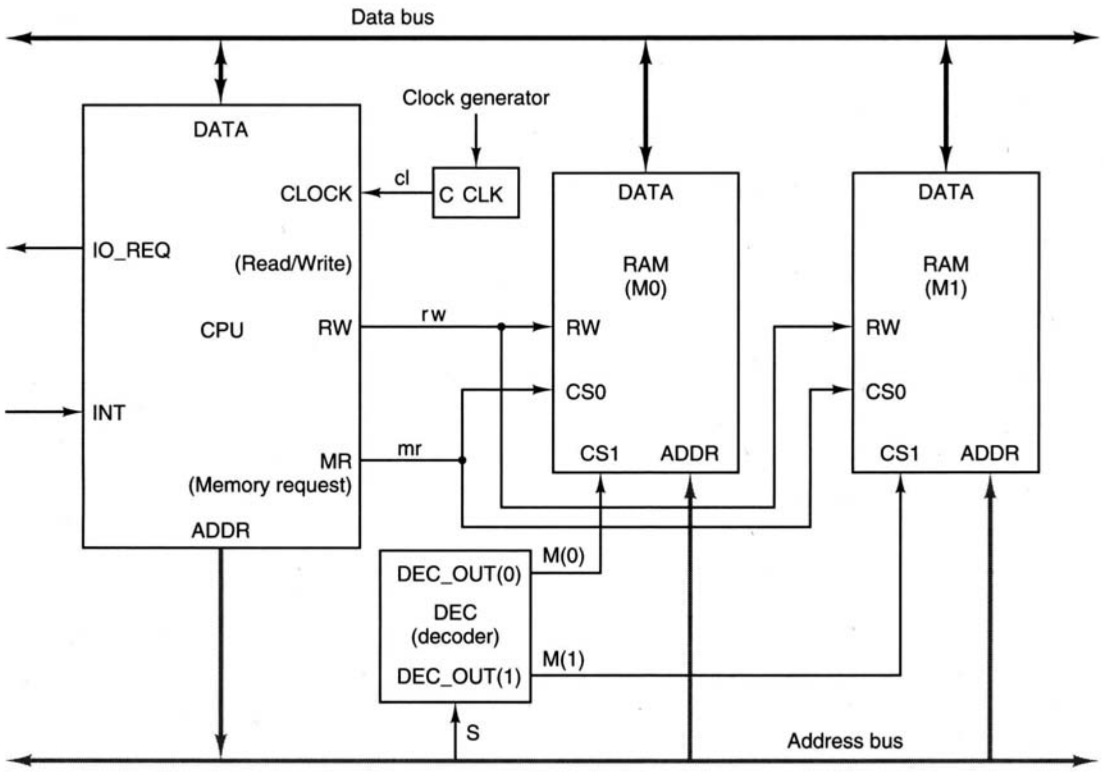


Figure 2.2 A simple microcomputer system.

Figure 2.4 describes the function (behavioral_view) of a memory module called random-access memory (RAM). In this figure, the variable memory stands for a memory unit consisting of 8 words, each of which has 8 bits. The while statement determines whether the RAM is selected or not. The RAM is selected whenever both signals CS0 and CS1 are 1. The case statement determines whether a datum should be read from the memory into the data bus (RW=0) or written into memory from the data bus (RW=1). When the signal RW is 0, the contents of the address bus (ADDR) are converted to an integer value, which is used as an index to determine the memory location that the data must be read from. Then the contents of the determined memory location are copied onto the data bus (DATA). In a similar manner, when the signal RW is 1, the contents of the data bus are copied into the proper memory location. The process statement constructs a process for simulating the RAM. The wait statement within the process statement causes the process to be suspended until the value of CS0 or CS1 changes. Once a change appears in any of these inputs, the process starts all over again and performs the proper function as necessary.

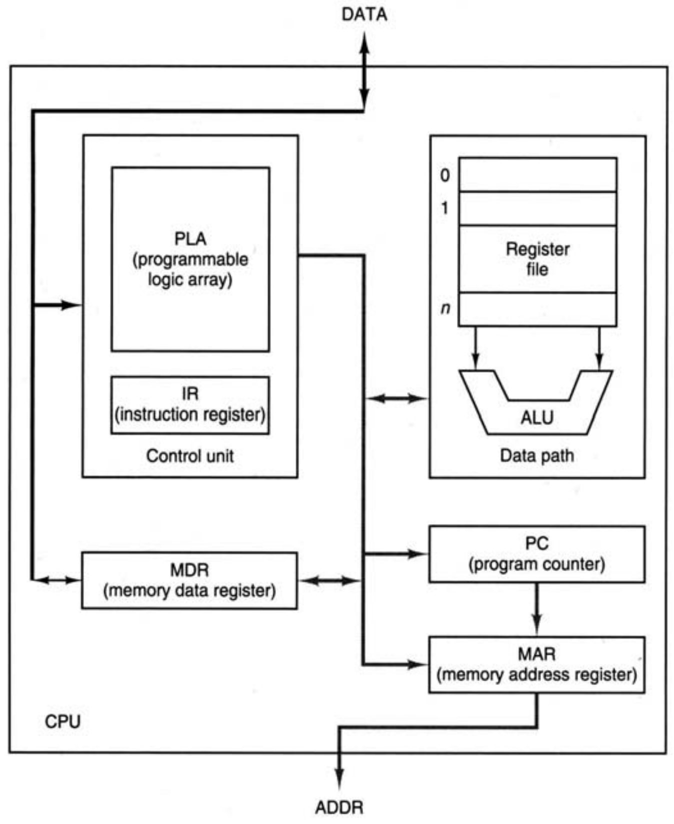
```
architecture  structure_view  of  microprocessor   is

        component  CPU
          port (DATA:   inout  tri_vector (0    to   7);
                ADDR:   out  bit_vector(3    downto   0);
                CLOCK, INT:   in  bit;
                MR, RW, IO_REQ:   out  bit);
        end component;

        component  RAM
          port (DATA:   inout  tri_vector(0   to   7);
                ADDR:  in  bit_vector(2   downto  0);
                CS0, CS1, RW:  in  bit);
        end component;

        component  DEC
          port (DEC_IN:   in  bit; DEC_OUT:   out  bit_vector(0   to  1));
        end component;

        component  CLK
          port (C:   out  bit);
        end component;

        signal  M: bit_vector(0   to  1);
        signal  cl, mr, rw: bit;

begin
        PROCESSOR: CPU  portmap (DATA, ADDR, cl, INT, mr, rw, IO_REQ);
        M0: RAM   portmap (DATA, ADDR(2 downto 0), mr, M(0), rw);
        M1: RAM   portmap (DATA, ADDR(2 downto 0), mr, M(1), rw);
        DECODER: DEC  portmap ( ADDR(3), M);
        CLOCK: CLK   portmap  (cl);
end  structure_view;
```

Figure 2.3 Structural representation of a simple microcomputer.

```
architecture behavioral_view of RAM is
begin
        process
            type   memory_unit  is  array(0 to 7) of bit_vector(0 to 7);
            variable  memory:  memory_unit;
        begin
            while  (CS0 = '1'   and  CS1 = '1')  loop
                case RW is                          --RW=0 means read operation
                                                    --RW=1 means write operation
                    when '0' => DATA <= memory(intval(ADDR)) after 50 ns;
                    when '1' => memory(intval(ADDR)) <= DATA after 60 ns;
                end case;
                wait on   CS0, CS1, DATA, ADDR, RW;
            end loop;
            wait on   CS0, CS1;
        end process;
end   behavioral_view;
```

Figure 2.4 Behavioral representation of an 8-by-8 RAM.

Figure 2.5 represents the main components of the CPU. These components are the data path, the control unit, and several registers referred to as the register file. The data path consists of the arithmetic logic unit (ALU) and various registers. The CPU communicates with memory modules through the memory data register (MDR) and the memory address register (MAR). The program counter (PC) is used for keeping the address of the next instruction that should be executed. The instruction register (IR) is used for temporarily holding an instruction while it is being decoded and executed.


DATA
PLA (programmable logic array)
IR (instruction register)
Control unit
0
1
Register file
n
ALU
Data path
MDR (memory data register)
PC (program counter)
MAR (memory address register)
CPU
ADDR
Figure 2.5 A simple CPU.

To express the function of the control unit, we will assume that our microcomputer has only four different instructions. Each instruction has a format as follows:

Opcode	Operand	···	Operand

The *opcode* (stands for operation code) field determines the function of the instruction, and the *operand* fields provide the addresses of data items. Figure 2.6 represents the opcode and the type of operands for each instruction. The *LOAD* instruction loads a memory word into a register. The *STORE* instruction stores a register into a memory word. The *ADDR* instruction adds the contents of two registers and stores the result in a third register. The *ADDM* instruction adds the contents of a register and a memory word and stores the result in the register.

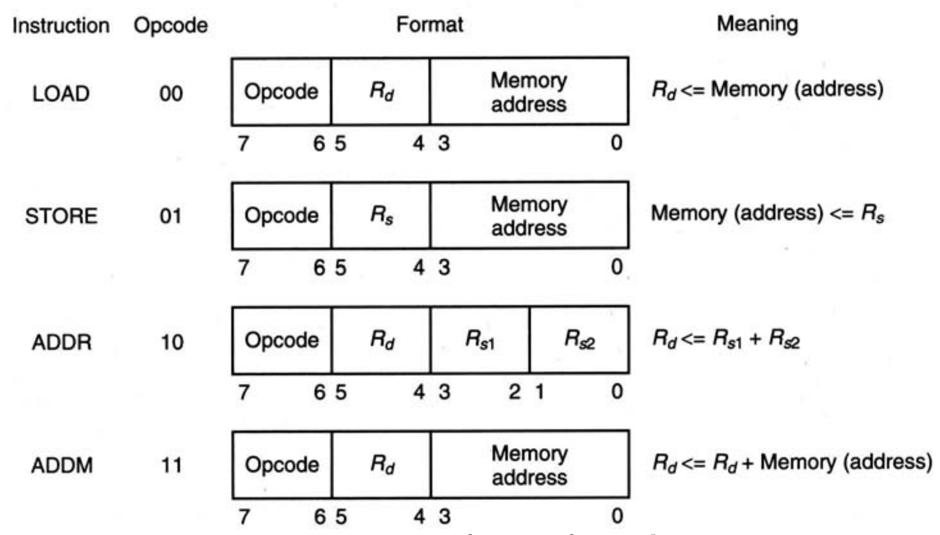
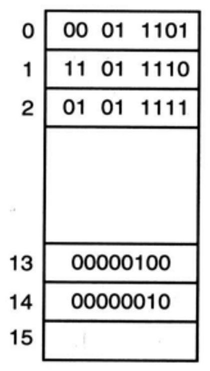


Figure 2.6 Instruction formats of a simple CPU.

To understand the roll of the control unit, let us examine the execution of a simple program in our microcomputer. As an example, consider a program that adds two numbers at memory locations 13 and 14 and stores the result at memory location 15. Using the preceding instructions, the program can be written as

LOAD 1,13 -- R_1 <= Memory (13)
ADDM 1,14 -- R_1 <= R_1 + Memory (14)
STORE 1,15 -- Memory (15) <= R_1

Let's assume that locations 13 and 14 contain values 4 and 2, respectively. Also, assume that the program is loaded into the first three words of memory. Thus, the contents of memory in binary are:



Memory

Figure 2.7 outlines the steps the computer will take to execute the program. Initially, the address of the first instruction (i.e., 0) is loaded into the program counter (PC). Next the contents of the PC are copied into the memory address register (MAR) and from there to the address bus. The control unit requests a read operation from the memory unit. At the same time, the contents of the PC are incremented by 1 to point to the next instruction. The memory unit picks up the address of the requested memory location (that is, 0) from the address bus and, after a certain delay, it transfers the contents of the requested location (that is, 00011101) to the memory data register (MDR) through the data bus. Then the contents of the MDR are copied into the instruction register (IR). The IR register is used for decoding the instruction. The control unit examines the leftmost 2 bits of the IR and determines that this instruction is a load operation. It copies the rightmost 4 bits of the IR into the MAR, such that it now contains 1101, which represents address 13 in decimal. The contents of memory location 13 are retrieved from the memory and stored in MDR in a similar manner to retrieving the instruction LOAD from memory. Next the contents of MDR are copied into the register R_1. At this time the execution of the LOAD instruction is complete.

The preceding process continues until all the instructions are executed. At the end of execution, the value 6 is stored in memory location 15.

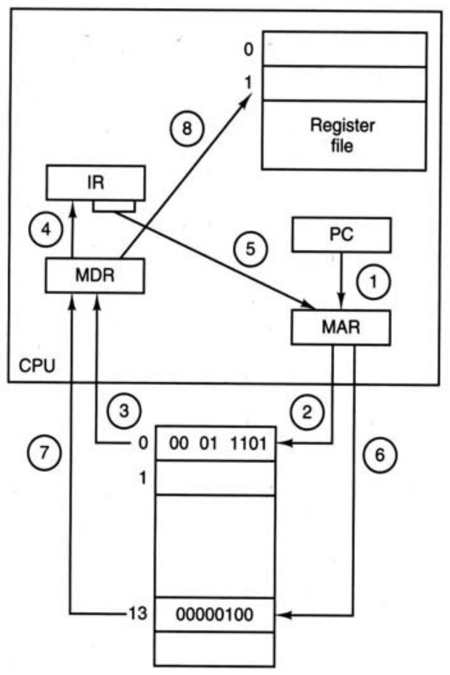


Figure 2.7. Flow of addresses and data for execution of the LOAD instruction.

In general, the execution process of an instruction can be divided into three main phases, as shown in Figure 2.8. The phases are *instruction fetch*, *decode_opfetch*, and *execute_opwrite*. In the fetch instruction phase, an instruction is retrieved from the memory and stored in the instruction register. The sequence of actions required to carry out this process can be grouped into three major steps.

1. Transfer the contents of the program counter to the memory address register and increment the program counter by 1. The program counter now contains the address of the next instruction to be fetched.
2. Transfer the contents of the memory location specified by the memory address register to the memory data register.
3. Transfer the contents of the memory data register to the instruction register.

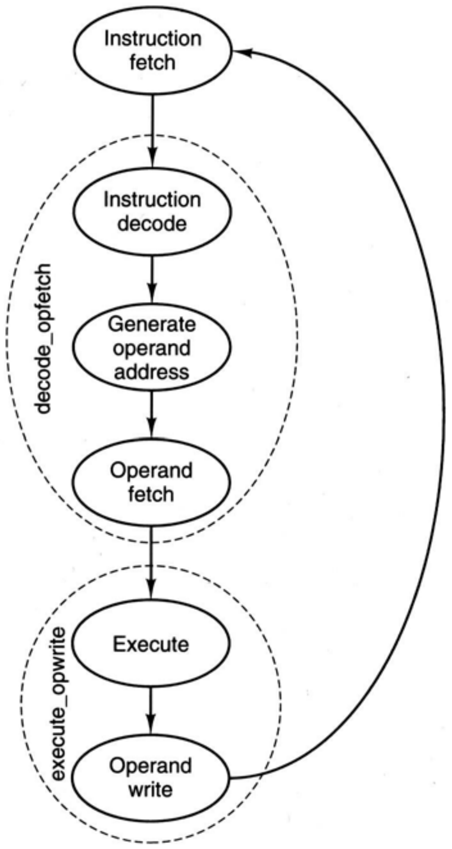


Figure 2.8 Main phases of an instruction process.

In the decode_opfetch phase, the instruction in the instruction register is decoded, and if the instruction needs an operand, it is fetched and placed into the desired location.

The last phase, execute_opwrite, performs the desired operation and then stores the result in the specified location. Sometimes no further action is required after the decode_opfetch phase. In these cases, the execute_opwrite phase is simply ignored. For example, a load instruction completes execution after the decode_opfetch phase.

The three phases described must be processed in sequence. Figure 2.9 presents a VHDL process for controlling the sequence of the phases. The function of each phase is described by a VHDL process. Figure 2.10 represents the steps involved in the instruction fetch phase. The function of decode_opfetch phase is presented in Figure 2.11. As shown in Figure 2.11, a case statement decodes a given instruction in the IR to direct the execution process to the proper routine. Figure 2.12 shows the process of the execute_opwrite phase. Take a moment to look over these figures to become familiar with instruction execution phases in the von Neumann architecture.

In the following sections, various design techniques for each component of the von Neumann architecture are explained.

```
control_state:  process (inst_fetch, decode_opfetch, execute_opwrite, CLOCK)
begin
        if   ((CLOCK = '1') and (not CLOCK'stable))     then
          if   ((not inst_fetch) and (not decode_opfetch) and
                (not execute_opwrite))    then
```

```
              case   ( next_state )    is
                    when   "inst_fetch_st" => inst_fetch <= true;
                                next_state := 'decode_opfetch_st';
                    when   "decode_opfetch_st" => decode_opfetch <= true;
                                next_state := 'execute_opwrite_st';
                    when   "execute_opwrite_st" => execute_opwrite <= true;
                                next_state := 'inst_fetch_st';
              end case;
           end if;
        end if;
end process   control_state;
```

Figure 2.9. Process for controlling the sequence of phases.

```
inst_fetch_state:    process
            begin
               wait on   inst_fetch     until    inst_fetch;
               MAR <= PC;
               ADDR <= MAR after 15 ns;       -- set the address for desired memory location
               MR <= '1' after 25 ns;         -- sets CS0 of each memory module to 1
               RW <= '0'    after   20 ns;     -- read from memory
               wait for 100 ns;                -- required time to read a data from memory
               MDR <= tri_vector_to_bit_vector (DATA);        -- since DATA has a tri_vector
                                                              -- type is converted to MDR
                                                              -- type which is bit_vector
               MR <= '0';
               IR <= MDR    after   15 ns;
               for   i    in   0    to   3    loop                -- increment PC by one
                     if   PC (i) = '0'    then
                        PC (i) := '1';
                        exit;
                     else
                        PC (i) := '0';
                     end if;
               end loop;
               inst_fetch <= false;
end process   inst_fetch_state;
```

Figure 2.10 Function of the instruction fetch phase.

```
decode_opfetch_state:   process
begin
            wait on  decode_opfetch  until  decode_opfetch;
            case  (IR (7 downto 6))  is
                                                                            -- LOAD
               when  "00" => MAR <= IR (3    downto   0);
                     ADDR <= MAR    after   15 ns;
                     MR <= '1'    after   25 ns;
                     RW <= '0'    after   20 ns;
                     wait for 100 ns;                 -- suppose 100 ns is required to read a
                                                      -- datum from memory
                     MDR <= tri_vector_to_bit_vector (DATA);
                     MR <= '0';
                     reg_file (intval (IR(5    downto   4))) <= MDR;
```

```
                                             -- copy MDR  to the destination register
                                                                             -- STORE
         when   "01" => MDR <= reg_file (intval (IR (5   downto   4)));
                DATA <= MDR   after  20 ns;
                MAR <= IR (3   downto  0);
                ADDR <= MAR   after  15 ns;
                MR <= '1'    after  25  ns;
                RW <= '1'   after  20 ns;
                wait for  110 ns;            -- suppose 110 ns is required to store
                                             -- a datum in memory
                MR <= "0";
                                                                             -- ADDR
         when "10" => ALU_REG1 <= reg_file (intval(IR(3  downto  2)));
                ALU_REG2 <= reg_file (intval(IR(1  downto  0)));
                add_op <= true   after 20  ns;
                                                                             -- ADDM
         when  "11" => ALU_REG1 <= reg_file (intval(IR(5 downto 4)));
                MAR <= IR (3  downto  0);
                ADDR <= MAR   after  15 ns;
                MR <= '1'   after  25   ns;
                RW <= '0'   after  20 ns;
                wait for  100 ns;            -- suppose 100 ns is required to read a
                                             -- datum from memory
                MDR <= tri_vector_to_bit_vector (DATA);
                MR <= '0';
                ALU_REG2 <= MDR;
                add_op <= true;
       end case;
       decode_opfetch <= false;
end process   decode_opfetch_state;
```

Figure 2.11. Function of the decode_opfetch phase.

```
execute_opwrite_state:   process
begin
       wait on  execute_opwrite   until  execute_opwrite;
         if  add_op   then
                reg_file(intval(IR(5 downto 4))) := ADD(ALU_REG1, ALU_REG2);
                add_op <= false;
          end if;
       execute_opwrite <= false;
end process   execute_opwrite_state;
```

Figure 2.12. Function of the execute_opwrite phase.

2.3 CONTROL UNIT

In general, there are two main approaches for realizing a control unit: the *hardwired* circuit and *microprogram* design.

Hardwired control unit. The hardwired approach to implementing a control unit is most easily represented as a sequential circuit based on different states in a machine; it issues a series of control signals at each state to govern the computers operation. (See Appendix B for information on sequential circuits.)

As an example, consider the design of a hardwired circuit for the load instruction of the simple microcomputer mentioned previously. This instruction has the format:

LOAD R_d, Address

As such, it would load the contents of a memory word into register R_d. Figure 2.13 represents the main registers and the control signals involved in this operation. The control signals are RW, MR, LD, AD, LA, WR, SR_0 and SR_1. The function of each is defined as follows:

RW:	perform read/write operation from/to memory (RW=0 means read, and RW=1 means write).
MR:	enable the chip select terminal (CS0) of the memory.
LD:	load data from data bus (DATA) to MDR.
AD:	load address from MAR to address bus (ADDR).
LA:	load the rightmost 4 bits of IR to MAR.
WR:	perform read/write operation from/to register file (WR=0 means read, and WR=1 means write).
SR0 and SR1:	select 2 bits of the IR as the address of register file.

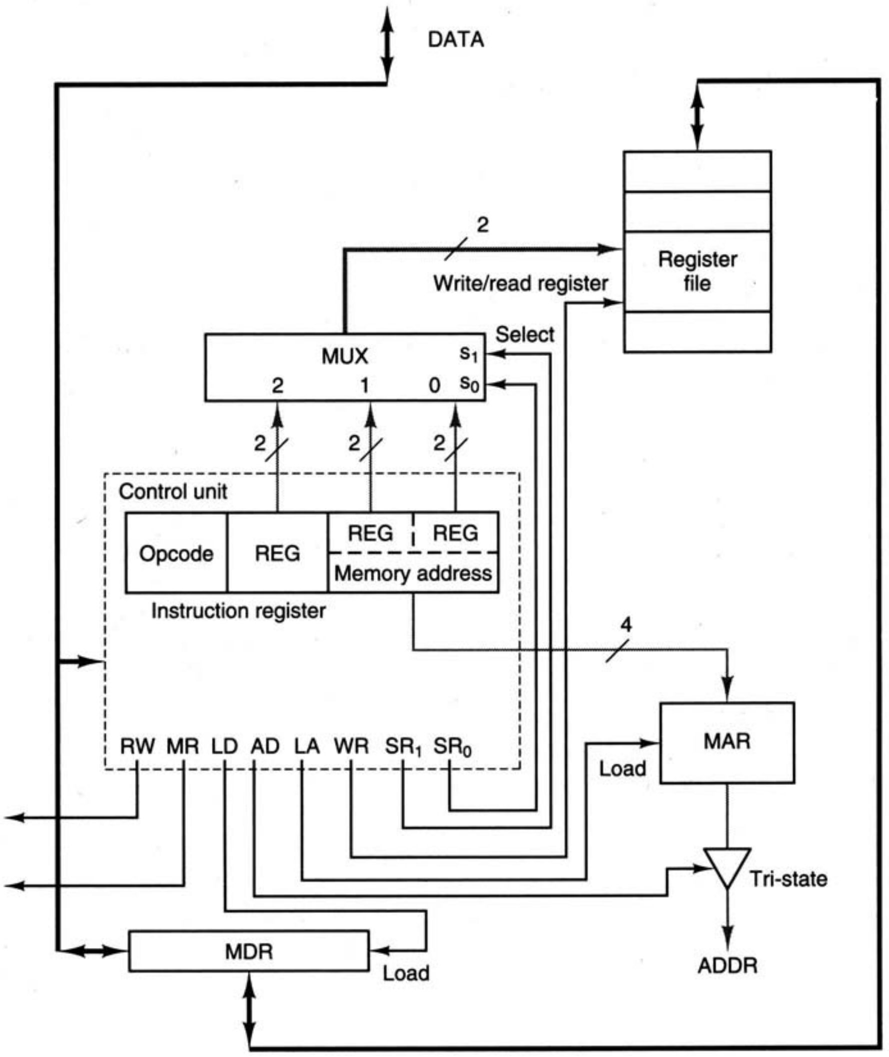
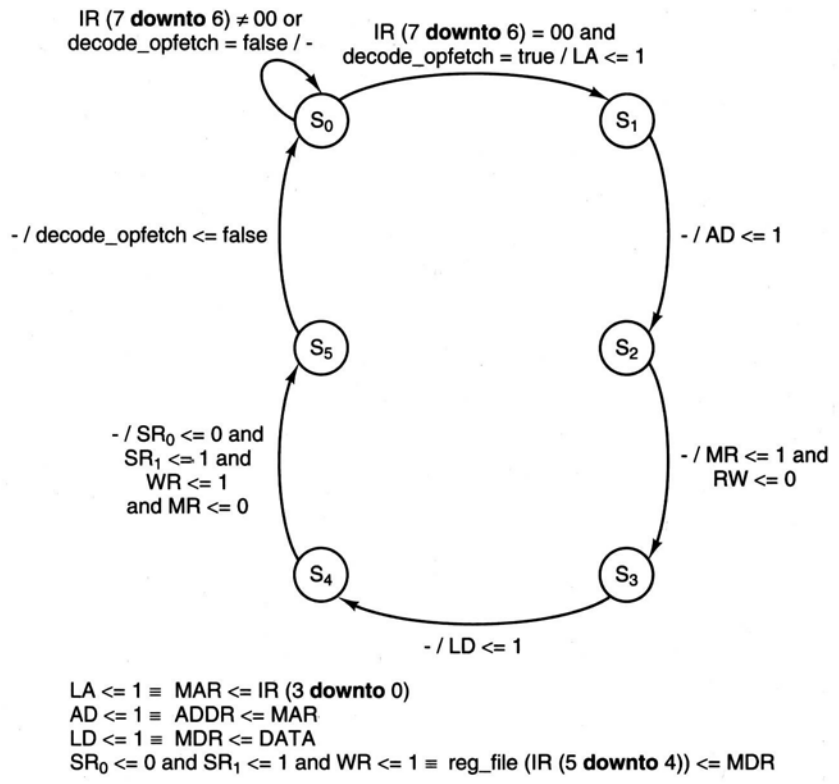


Figure 2.13 Control signals for some portions of the simple CPU in Figure 2.5.

The load operation starts by loading the 4 least significant bits of the instruction register (IR) into the memory address register (MAR). Then the desired data are fetched from memory and loaded into the memory data register (MDR). Next the destination register R_d is selected by setting s_0 and s_1 (select lines of the multiplexer, or MUX) to 0 and 1, respectively. The operation is completed by transferring the contents of the MDR to the register selected by SR_0 and SR_1.

Figure 2.14 represents a state diagram for the load operation. In this state diagram, the state S_0 represents the initial state. A transition from a state to another state is represented by an arrow. To each arrow a label in the form of *X/Z* is assigned; here *X* and *Z* represent a set of input and output signals, respectively. When there is no input signal or output signal, *X* or *Y* is represented as "-". A transition from S_0 to S_1 occurs whenever the leftmost 2 bits of IR are '00' and the signal decode_opfetch is true. (Note that although values of the signal decode_opfetch are represented as true and false, these values would actually be logical values 1 and 0, respectively, in a real implementation of this machine.) The signal decode_opfetch is set to true whenever the execution process of an instruction enters the decode_opfetch phase. When transition from S_0 to S_1 occurs, the control signal LA is set to 1, causing the rightmost 4 bits of IR to be loaded into the MAR. A transition from S_1 to S_2 causes the contents of MAR to be loaded on the address bus. The transition from S2 to S3 causes the target data to be read from the memory and loaded on the data bus. Note that it is assumed that a clock cycle (which causes the transition from S_2 to S_3) is enough for reading a datum from memory. A transition from S_3 to S_4 copies the data on the data bus into MDR. The transition from S_4 to S_5 sets the select lines of the multiplexer (MUX) to 1 and 0. This causes the third and fourth bits (from left) of IR to be selected as an address to the register file. By setting WR=1, the contents of MDR are copied into the addressed register. A transition from S_5 to S_0 completes the load operation by setting the signal decode-opfetch to false.


IR (7 **downto** 6) ≠ 00 or decode_opfetch = false / -
IR (7 **downto** 6) = 00 and decode_opfetch = true / LA <= 1
S_0
S_1
- / decode_opfetch <= false
- / AD <= 1
S_5
S_2
- / SR_0 <= 0 and SR_1 <= 1 and WR <= 1 and MR <= 0
- / MR <= 1 and RW <= 0
S_4
S_3
- / LD <= 1
LA <= 1 ≡ MAR <= IR (3 **downto** 0)
AD <= 1 ≡ ADDR <= MAR
LD <= 1 ≡ MDR <= DATA
SR_0 <= 0 and SR_1 <= 1 and WR <= 1 ≡ reg_file (IR (5 **downto** 4)) <= MDR
Figure 2.14 State diagram for the load operation.

The state diagram of Figure 2.14 can be used to implement a hardwired circuit. Usually a programmable logic array (PLA) (defined in Appendix B) is used for designing such a circuit. Figure 2.15 represents the main components involved in such a design. When the size of the PLA becomes too large, it is often decomposed to several smaller-sized PLAs in order to save space on the chip.

One main drawback with the preceding approach (which is often mentioned in the literature) is that it is not flexible. In other words, a later change in the control unit requires the change of the whole circuit. The rationale for considering this a drawback is that a complete instruction set is usually not definable at the time that a processor is being designed, and a good design must allow certain operations, defined by some later user, to be executed at a very high speed. However, because microprocessor design changes so rapidly today, the lifetime of any particular processor is very short, making flexibility less of an issue. In fact, most of today's microprocessors are based on the hardwired approach.

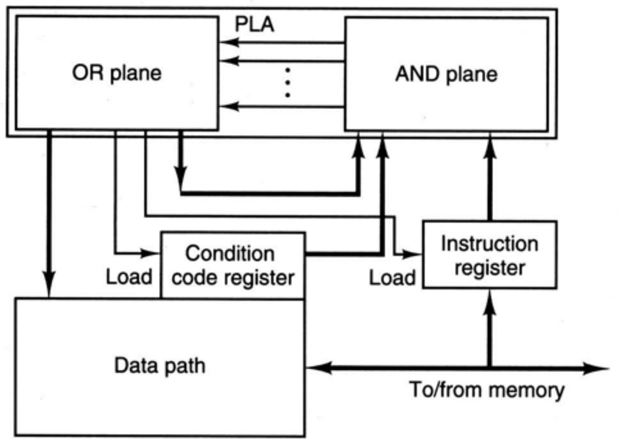


Figure 2.15 Structure of a CPU based on hardwired control unit design.

Microprogrammed control unit. To solve the inflexibility problem with a hardwired approach, in 1951 Wilkes invented a technique called microprogramming [WIL 51]. Today, while microprogramming has become less important as a widespread design method, the concept remains quite important. Hayes says that a microprogram design "resembles a computer within a computer; it contains a special memory from which it fetches and executes control information, in much the same manner as the CPU fetches and executes instructions from the main memory" [HAY 93].

In microprogramming, each machine instruction translates into a sequence of microinstructions that triggers the control signals of the machine's resources; these signals initiate fundamental computer operations. Microinstructions are bit patterns stored in a separate memory known as microcode storage, which has an access time that is much faster than that of main memory. The evolution of microprogramming in the 1970s is linked to the introduction of low-cost and high-density memory chips made possible by the advance of semiconductor technology. Figure 2.16 shows the main components involved in a microprogrammed machine. The microprogram counter points to the next microinstruction for execution, which, upon execution, causes the activation of some of the control signals. In Figure 2.16, instead of having direct connections from microcode storage and condition code registers to the decoder, a control circuit maps these connections to a smaller number of connections. This reduces the size of microcode storage. Because the microinstructions are stored in a memory, it is possible to add or change certain instructions without changing any circuit. Thus the microprogramming technique is much more flexible than the hardwired approach. However, it is potentially slower because each microinstruction must be accessed from microcode storage.

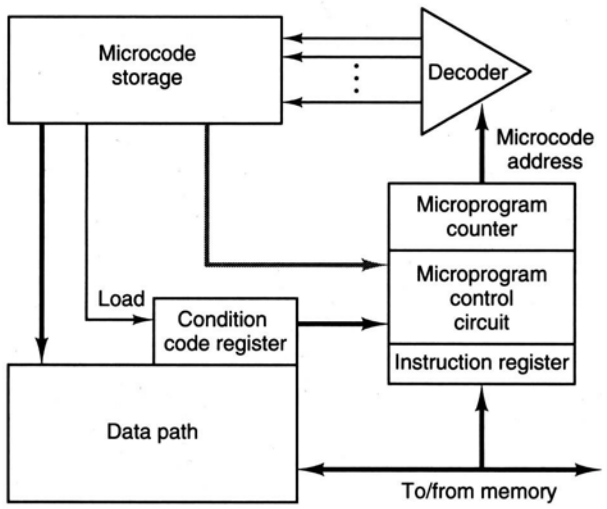


Figure 2.16 Structure of a CPU based on microprogrammed control unit design.

Microinstruction Word Design. The following criteria should be considered in designing a microinstruction format.

1. Minimization of the microcode storage word size (microword).
2. Minimization of the microprogram size.
3. Maximization of the flexibility of adding or changing microinstructions.
4. Maximization of the concurrency of the microoperations.

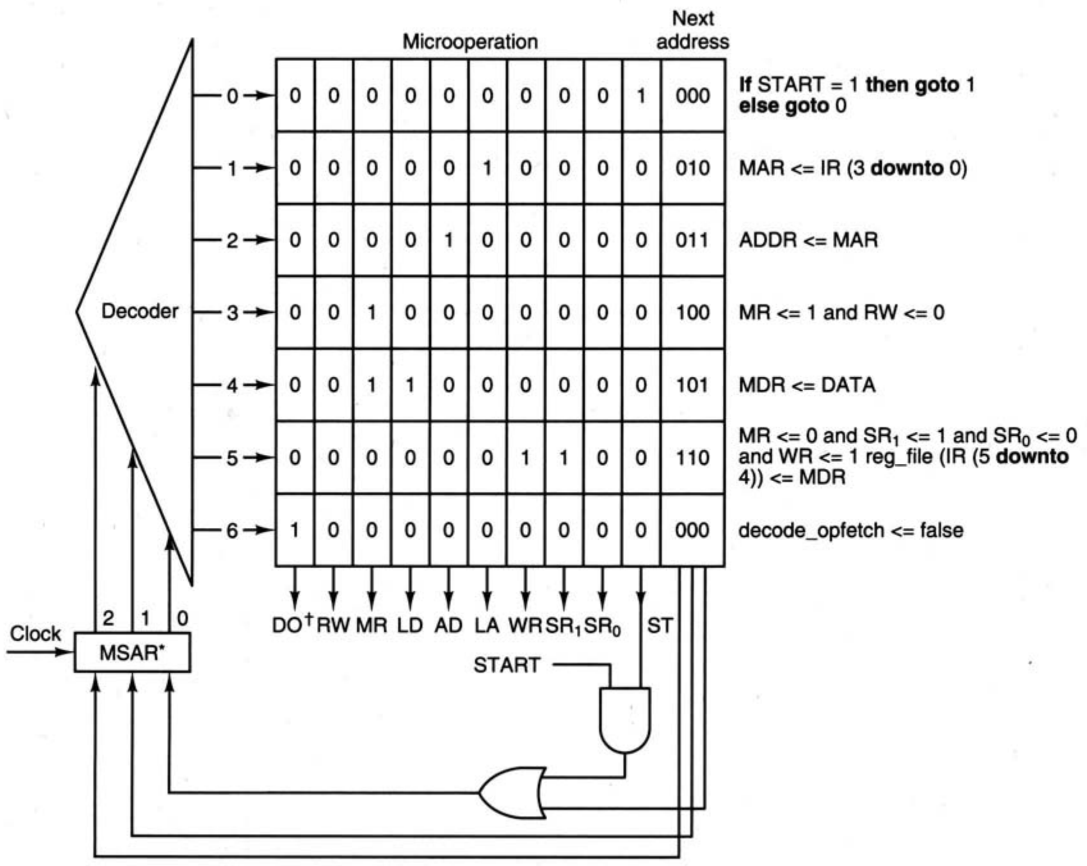
In general, a microword contains two fields, the microoperation field and the next address field. One extreme design for the microoperation field is to assign a bit to each control signal; this is called *horizontal* design, because many bits are usually needed, resulting in a wide, or horizontal, microoperation field. In this type of design, whenever a microword is addressed the stored 1's in the microoperation field will cause the invocation of corresponding control signals.

Another extreme design for the microoperation field is to have a few highly encoded subfields; this is called *vertical* design, because relatively few bits are needed, resulting in a narrower microoperation field. Each subfield may invoke only one control signal within a certain group of control signals. This is done by assigning a unique code to each control signal. For example, to a subfield with 2 bits, a unique code of 00, 01, 10, or 11 can be assigned to each control signal in a group with a maximum of three control signals. Note that one of the codes in each subfield is reserved for no operation (i.e., for when you do not want to invoke any signals within a group). The distinction between the concept of horizontal and vertical designs becomes more clear upon investigating the design of a microprogram for the load instruction of our simple microcomputer. This instruction has a format such as

LOAD R_d Address

This instruction loads the contents of a memory word into register R_d. Figure 2.13 shows the main registers and the control signals involved in this operation. The operation starts by loading the 4 least significant bits of the instruction register (IR) into the memory address register (MAR). Then the desired data are fetched from memory and loaded into the memory data register (MDR). Next the destination register R_d is selected by setting S_0 and S_1 (select lines of the multiplexor) to 0 and 1, respectively. The operation is completed by transferring the contents of the MDR to the selected register.

Figure 2.17 shows the preceding steps as a series of microinstructions that is stored in a microcode storage based on a horizontal design (i.e., a bit is assigned to each control signal). In addition to the eight control signals of Figure 2.13, there are two other control signals, ST and DO. The ST signal causes the load operation to start whenever signal START becomes 1. The DO signal indicates the completion of the load operation and causes the control unit to bring to an end the decode-opfetch phase. Initially, the contents of the MSAR register (microcode storage address register) are set to 0. Hence the contents of microword 0 appear on the control signals. That is, ST is set to 1 and the other signals are set to 0. When a load instruction is detected, START is set to 1. At this point, both signals ST and START are 1, and as a result the rightmost bit of MSAR is set to 1. Therefore, microword 1 is selected and, as a result, control signal LA is set to 1. This process continues until the last microcode, 6, is selected. At that time DO is set to 1, which completes the load operation.



	DO†	RW	MR	LD	AD	LA	WR	SR$_1$	SR$_0$	ST	Next address	
0	0	0	0	0	0	0	0	0	0	1	000	**If** START = 1 **then goto** 1 **else goto** 0
1	0	0	0	0	0	1	0	0	0	0	010	MAR <= IR (3 **downto** 0)
2	0	0	0	0	1	0	0	0	0	0	011	ADDR <= MAR
3	0	0	1	0	0	0	0	0	0	0	100	MR <= 1 and RW <= 0
4	0	0	1	1	0	0	0	0	0	0	101	MDR <= DATA
5	0	0	0	0	0	0	1	1	0	0	110	MR <= 0 and SR$_1$ <= 1 and SR$_0$ <= 0 and WR <= 1 reg_file (IR (5 **downto** 4)) <= MDR
6	1	0	0	0	0	0	0	0	0	0	000	decode_opfetch <= false

```
+ If (IR (7 downto 6) = 00 and decode_opfetch = true) then
    START <= 1;
  else
    START <= 0;
  end if;
```

* MSAR stands for microcode storage address register.
† DO stands for decode_opfetch; DO = 1 is equivalent to decode_opfetch <= false.

Figure 2.17 Horizontal microprogram for the load operation.

An alternative to the preceding design is shown in Figure 2.18. This figure represents a vertical microprogram for the same load operation. Note that in this design each microword has 8 bits, in comparison to the former horizontal design in which each microword had 13 bits. The microoperation of each microword consists of two subfields: F_1 and F_2. F_1 has 2 bits that are used to invoke one of the control signals SR_0, WR, and MR, at any given time. In F_1, the codes 00, 01, 10, and 11 are assigned to no operation, SR_0, WR, and MR, respectively. F_2 has 3 bits that are used to invoke one of the control signals SR_1, AD, LD, RW, DO, LA, and ST, at any given time. In F_2, the codes 000, 001, 010, 011, 100, 101, 110, and 111 are assigned to no operation, SR_1, AD, LD, RW, DO, LA, and ST, respectively.

An advantage of vertical microprogramming (VM) over horizontal microprogramming (HM) is that VM uses relatively shorter microinstructions. Nevertheless, horizontal microprogramming is more powerful because it does not enforce any constraint for modifying the set of microinstructions of an instruction.

Some of the advantages of HM over VM are:

1. Simultaneous execution of control signals within the same microinstruction (i.e., any combination of control signals can be triggered at the same time)
2. Relatively short microinstruction executing time. VM requires a longer execution time because of the delays associated with decoding the encoded microinstruction subfields.

The characteristics of VM and HM have a direct impact on the structures of computer systems. Therefore, when adopting one of the two techniques, computer architects must be aware of the attributes of each method. For instance, if many components of the CPU are required to be operated simultaneously, HM is more appropriate. HM always allows full use of parallelism. On the other hand, if the emphasis is on less hardware cost and a smaller set of microinstructions, VM would be more suitable. In practice, usually a mix of both approaches is chosen.

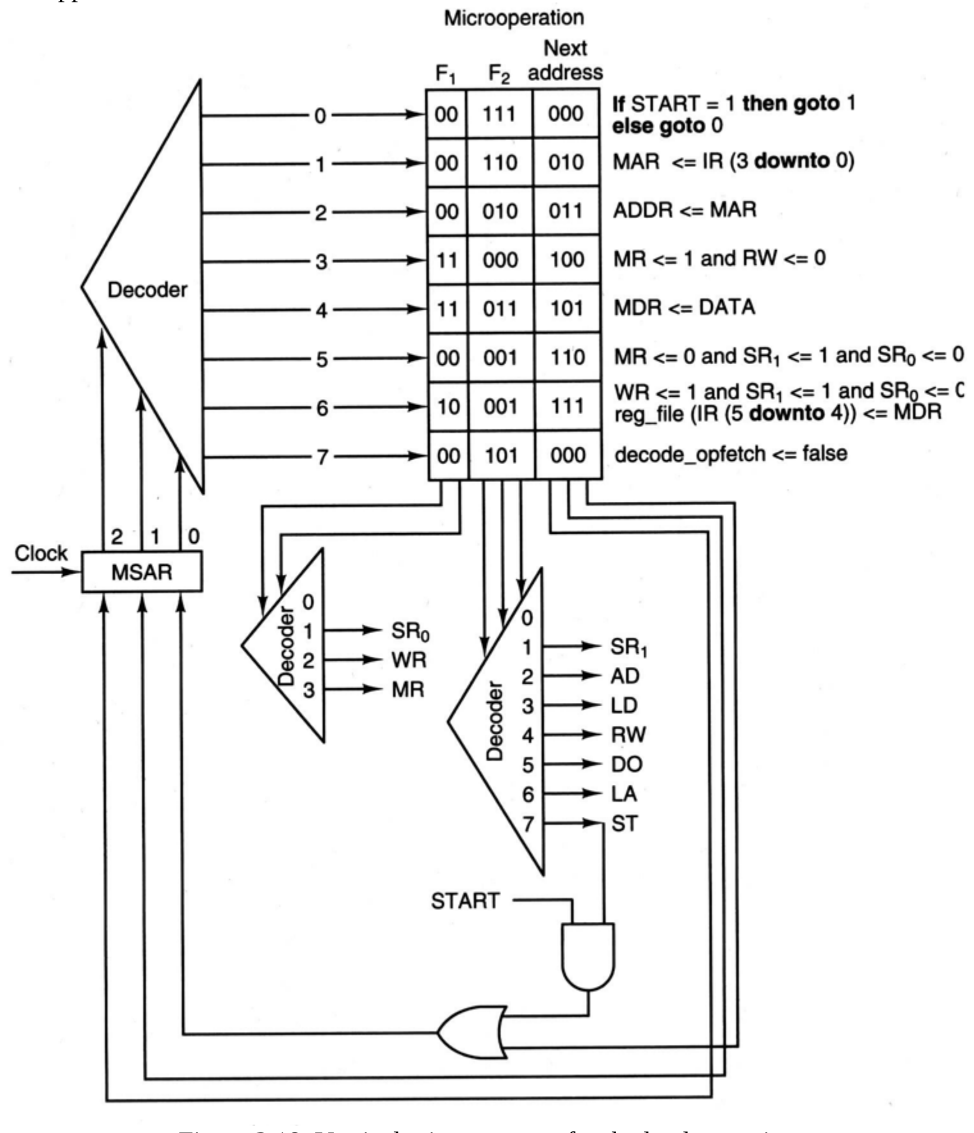
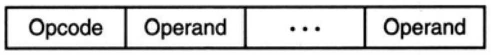

Microoperation
F1
F2
Next address
00 111 000 If START = 1 then goto 1 else goto 0
00 110 010 MAR <= IR (3 downto 0)
00 010 011 ADDR <= MAR
11 000 100 MR <= 1 and RW <= 0
11 011 101 MDR <= DATA
00 001 110 MR <= 0 and SR1 <= 1 and SR0 <= 0
10 001 111 WR <= 1 and SR1 <= 1 and SR0 <= 0 reg_file (IR (5 downto 4)) <= MDR
00 101 000 decode_opfetch <= false
Decoder
Clock
MSAR
2 1 0
Decoder 0 1 2 3
SR0
WR
MR
Decoder 0 1 2 3 4 5 6 7
SR1
AD
LD
RW
DO
LA
ST
START

Figure 2.18 Vertical microprogram for the load operation.

2.4 INSTRUCTION SET DESIGN

The design of an instruction set is one of the most important aspects of processor design. The design of the instruction set is highly complex because it defines many of the functions performed by the CPU and therefore affects most aspects of the entire system. Each instruction must contain the information required by the CPU for execution.

With most instruction sets, more than one instruction format is used. Each instruction format consists of an opcode field and 0 to 3 operand fields, as follows:



The opcode (stands for operation code) field determines the function of the instruction, and the operand fields provide the addresses of data items (or sometimes the data items themselves). Designing this type of instruction format requires answers to the following questions:

How many instructions are provided?
What type of operations are provided?
How many operand fields and what type of operands are allowed in each instruction?

A number of conflicting factors complicate the task of instruction set design. As a result, there are no simple answers to these questions, as the following discussion demonstrates.

Size of opcode. The question of how many instructions are provided is directly related to the size of the opcode. The opcode size reflects the number of instructions that can be provided by an architecture; as the number of bits in the opcode increases, the number of instructions will also increase. Having more instructions reduces the size of a program. Smaller programs tend to reduce the storage space and execution time. This is because a sequence of basic instructions can be reinterpreted as equivalent to one advanced instruction. For example, if an instruction set (all the instructions provided by an architecture) includes a multiplication operation, only one instruction is needed in the program to multiply instead of a sequence of add and shift instructions. To summarize, if there are only a few simple instructions to choose from, then many are required, making longer programs to perform a task. If many instructions are available, then fewer are needed, because each instruction will accomplish a longer part of the task. Fewer program instructions means a shorter program.

Although increasing the number of bits in the opcode reduces the program size, ultimately a price must be paid for such an increase. Eventually, the addition of an extra bit to the opcode field will result in increased storage space for a program, despite the initial program size reduction.

Furthermore, increasing the number of instructions will add more complexity to the processor design, which increases the cost. A larger set of instructions requires a more extensive control unit circuit and complicates the process of microprogramming design if microprogramming is used. Additionally, it is ideal to have the whole CPU design on a single chip, since a chip is much faster and less expensive than a board of chips. If design complexity increases, more gates will be needed to successfully implement the design, which could make it impossible to fit the whole design on a single chip.

From this discussion, it can be concluded that the size of an instruction set directly affects even the most fundamental issues involved in processor design. A small and simple instruction set offers the advantage of uncomplicated hardware designs, but also increases program size. A large and complex instruction set decreases program storage needs, but also increases hardware complexity. One trend in computer design is to increase the complexity of the instruction set by providing special instructions that are able to perform complex operations. Recent machines falling within such trends are termed *complex instruction set computers* (**CISC**s). The CISC design approach emphasizes reducing the number of instructions in the program and, as a result, increases overall performance.

Another trend in computer design is to simplify the instruction set rather than make it more complex. As a result, the terminology *reduced instruction set computer* (**RISC**) has been introduced for this type of design. The basic concept of the RISC design approach is based on the observation that in a large number of programs many complex instructions are seldom used. It has been shown that programmers or compiler writers often use only a subset of the instruction set. Therefore, in contrast to CISC, the RISC design approach employs a simpler instruction set to improve the rate at which instructions can be executed. It emphasizes reducing the average number of clock cycles required to execute an instruction, rather than reducing the number of instructions in the program. In RISC design, most instructions are executed within a single cycle. This innovative approach to computer architecture is covered in more detail in chapters 3 and 4.

Type of operation. It is important to have an instruction set that is compatible with previous processors in the same series or even with different brands of processors. Compatibility allows the user to run the existing software on the new machine. Therefore, there is a tendency to support the existing instruction set and add more instructions to it in order to increase performance (i.e., instruction set size is increasing). However, the designer should be very careful when adding an instruction to the set because, once programmers decide to use the instruction and critical programs include the instruction, it may become hard to remove it in the future.

Although instruction sets vary among machines, most of them include the same general types of operations. These operations can be classified as follows:

> **Data transfer**. For transferring (or copying) data from one location (memory or register) to another location (register or memory), such as load, store, and move instructions.
>
> **Arithmetic**. For performing basic arithmetic operations, such as increment, decrement, add, subtract, multiply, and divide.
>
> **Logical**. For performing Boolean operations such as NOT, AND, OR, and exclusive-OR. These operations operate on bits of a word as bits rather than as numbers.
>
> **Control**. For controlling the sequence of instruction execution there exist instructions such as jump, branch, skip, procedure call, return, and halt.
>
> **System**. These types of operations are generally privileged instructions and are often reserved for the use of the operating system, as in system calls and memory management instructions.
>
> **Input/output (I/O)**. For transferring data between the memory and external I/O devices. The commonly used I/O instructions are input (read) and output (write).

Type of operand fields. An instruction format may consist of one or more operand fields, which provide the address of data or data themselves. In a typical instruction format the size of each operand is quite limited. With this limited size, it is necessary to refer to a large range of locations in main memory. To achieve this goal, a variety of addressing modes have been employed. The most commonly used addressing modes are *immediate*, *direct*, *indirect*, *displacement*, and *stack*. Each of these techniques is explained next.

Immediate Addressing. In immediate addressing the operand field actually contains the operand itself, rather than an address or other information describing where the operand is. The immediate addressing is the simplest way for an instruction to specify an operand. This is because, upon execution of an instruction, the operand is immediately available for use, and hence it does not require an extra memory reference to fetch the operand. However, it has the disadvantage of restricting the range of the operand to numbers that can fit in the limited size of the operand field.

Direct Addressing. In direct addressing, also referred to as absolute addressing, the operand field contains the address of the memory location or the address of the register in which the operand is stored. When the operand field refers to a register, this mode is also referred to as *register direct addressing*. This type of addressing requires only one memory (or register) reference to fetch the operand and does not require any special calculation for obtaining the operand's address. However, it provides a limited address space depending on the size of operand field.

Indirect Addressing. In indirect addressing the operand field specifies which memory location or register contains the address of the operand. When the operand field refers to a register, this mode is also referred to as *register indirect addressing*. In this addressing mode the operand's address space depends on the word length of the memory or the register length. That is, in contrast to direct addressing, the operand's address space is not limited to the size of the operand field. The main drawback of this addressing mode is that it requires two references to fetch the operand, one memory (or register) reference to get the operand's address and a second memory reference to get the operand itself.

Displacement Addressing. Displacement addressing combines the capabilities of direct addressing and register indirect addressing. It requires that the operand field consists of two subfields and that at least one of them is explicit. One of the subfields (which may be implicit in the opcode) refers to a register whose content is added to the value of the other subfield to produce the operand's address. The value of one of the subfields is called the *displacement* from the memory address which is denoted by the other subfield. The term displacement is used because the value of the subfield is too small to reference all the memory locations. Three of the most common uses of this addressing mode are *relative addressing, base register addressing*, and *indexed addressing*.

For relative addressing, one of the subfields is implicit and refers to the program counter (PC). That is, the program counter (current instruction address) is added to the operand field to produce the operand's address. The operand field contains an integer called the displacement from the memory address indicated by the program counter.

For base register addressing, the referenced register, referred to as the *base register*, contains a memory address, and the operand field contains an integer called the displacement from that memory address. The register reference may be implicit or explicit. The content of the base register is added to the displacement to produce the operand's address. Usually, processors contain special base registers; if they do not, general-purpose registers are used as base registers.

For indexed addressing (or simply indexing), the operand field references a memory address, and the referenced register, referred to as the *index register*, contains a positive displacement from that address. The register reference may be implicit or explicit. The memory address is added to the index register to produce the operand's address. Usually, processors contain special index registers; if they do not, general-purpose registers are used as index registers.

Stack Addressing. A stack can be considered to be a linear array of memory locations. Data items are added (removed) to (from) the top of the stack. Stack addressing is in fact an implied form of register indirect addressing. This is so that, at any given time, the address of the top of the stack is kept in a specific register, referred to as the *stack register*. Thus the machine instructions do not need to include the operand field because, implicitly, the stack register provides the operand's address. That is, the instructions operate on the top of the stack. For example, to execute an add instruction, two operands are popped off the top of the stack, one after another, the addition is performed, and the result is pushed back onto the stack.

Number of operands per operation. A question that a designer must answer is how many operands (operand fields) might be needed in an instruction. The number of operands ranges from none to two or more. The relative advantages and disadvantages of each case are discussed next.

1. When there is no operand for most of the operations, the machine is called a *stack machine*. A stack machine uses implicit specification of operands; therefore, the instruction does not need a field to specify the operand. The operands are stored on a stack. Most instructions implicitly use the top operand in the stack. Only the instructions PUSH and POP access memory. For example, to perform the expression $Z = X + Y$, the following sequence of operations may be used:

```
PUSH X      -- load top of stack X
PUSH Y      -- load top of stack Y
ADD         -- add top most two operands of stack
```

```
POP Z             -- store top of stack in Z
```

In general, the evaluation of an expression is based on a simple model in this type of machine. Also, instructions have a short format (the number of bits in opcode field determines the size of most instructions). However, these good points are tempered by the following drawbacks:

a. It does not allow the use of registers (beside top of stack), which is needed for generating an efficient code and reducing the execution time of a program.

b. It makes fast implementations of programs difficult.

2. When there is one operand, the machine usually has a register called the *accumulator*, which contains the second operand. In this type of machine, most operations are performed on the operand and the accumulator with the result stored in the accumulator. There are some instructions for loading and storing data in accumulators. For example, for the expression $Z = X + Y$, the code is

```
LOAD_ACC X        -- load accumulator X
ADD Y             -- adds the contents of accumulator to Y and store result in
                  -- accumulator
STORE Z           -- store accumulator in Z
```

Similar to a stack machine, the advantages are a simple model for expression evaluation and short instruction formats. The drawbacks are also like those of the stack machine, that is, a lack of registers besides the accumulator, which affects good code performance. Since there is only one register, the performance degrades by accessing memory most of the time.

3.   When there are two or three operands, usually the machine has two or more registers. Typically, arithmetic and logical operations require one to three operands. Two- and three-operand instructions are common because they require a relatively short instruction format. In the two-operand type of instruction, one operand is often used both as a source and a destination. In three-operand instructions, one operand is used as a destination and the other two as sources.

In general, the machines with two- or three-operand instructions are of two types: those with a load-store design, and those with a memory-register design. In a load-store machine, only load and store instructions access memory, and the rest of the instructions usually perform an operation on registers. For example, for $Z = X + Y$, the code may be

LOAD R_1, *X*
LOAD R_2, *Y*
ADD R_1, R_2
STORE R_1, *Z*

The advantages of this type of design are as follows:

a. Few memory accesses; that is, performance (speed) increases by accessing registers most of the time.

b. Short instruction formats, because fewer bits are required for addressing a register than they are for addressing memory (on the grounds that the register address space is much smaller than memory address space).

c. Instructions may have a fixed length.

d. Instructions often take the same number of clock cycles for execution.

e. Registers provide a simple model for compilers to generate good object code.

However, because most of the instructions access registers, fewer addressing modes can be encoded in a single instruction. In other words, instruction encoding is inefficient, and hence sometimes a sequence of instructions is needed to perform a simple operation. For example, to add the contents of a memory location to a register may involve the execution of two instructions: a LOAD and an ADD. This type of machine usually provides more instructions than the memory-register type.

In a memory-register type of machine, usually one operand addresses a memory location while the other operands (one or two) address registers. For example, for $Z = X + Y$, the code may be:

LOAD R_1, X
ADD R_1, Y
STORE R_1, Z

In comparison with the load-store machine, the memory-register machine has the advantage that data can be accessed from memory without first loading them into a register. Another benefit is that the number of addressing modes that can be encoded in a single instruction increases; as a result, on average, fewer instructions are needed to encode a program. The drawbacks are the following:

a. The instructions have variable lengths, which increases the complexity of design.

b. Most of the time the operands are fetched from memory.

c. There are fewer registers than in load-store machines due to the increase in the number of addressing modes encoded in an instruction.

4. Finally, there may be two or more memory addresses for an instruction. An advantage of this scheme is that it does not waste registers for temporary storage. However, this scheme results in too many memory accesses, causing the instructions to become more complex in size and work (utilizing zero to three memory operands and zero to three memory accesses).

2.5 ARITHMETIC LOGIC UNIT

The arithmetic logic unit (ALU) is arguably the most important part of the central processing unit (CPU). The ALU performs the decision-making (logical) and arithmetic operations. It works in combination with a number of registers that hold the data on which the logical or mathematical operations are to be performed.

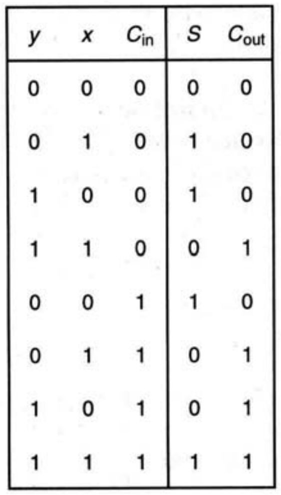
For decision-making operations, the ALU can determine if a number equals zero, is positive or negative, or which of two numbers is larger or smaller. These operations are most likely to be used as criteria to control the flow of a program. It is also standard for the ALU to perform basic logic functions such as AND, OR, and Exclusive-OR.

For arithmetic operations, the ALU often performs functions such as addition, subtraction, multiplication, and division. There are a variety of techniques for designing these functions. In this section, some well-known basic and advanced techniques for some of these functions are discussed. These same techniques can be applied to the more complex mathematical operations, such as raising numbers to powers or extracting roots. These operations are customarily handled by a program that repetitively uses the simpler functions of the ALU; however, some of the newer ALUs have been experimenting with more proficient ways of hardwiring more complex mathematical operations directly into the circuit. The trade-off is a more complex ALU circuit for much faster operation results.

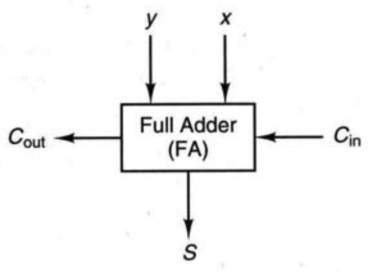
2.5.1 Addition

Many of the ALU's functions reduce to a simple addition or series of additions [DEE 74]. As such, by increasing the speed of addition, we might increase the speed of the ALU and, similarly, the speed of the overall machine. Adder designs range from very simple to very complex. Speed and cost are directly proportional to the complexity. The discussions that follow will explore full adders, ripple carry adders, carry lookahead adders, carry select adders, carry save adders, binary-coded decimal adders, and serial adders.

Full adder. A full adder adds three 1-bit inputs. Two of the inputs are the two significant bits to be added (denoted *x*, *y*), and the other input is the carry from the lower significant bit position (denoted C_{in}), producing a sum (denoted *S*) bit and a carry (denoted C_{out}) bit as outputs. Figure 2.19 presents the truth table, the Boolean expressions, and a block diagram for a full adder. The full adder will be one of the basic building blocks for the more complex adders to follow.



y	x	C_{in}	S	C_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1



$$S = x \oplus y \oplus C_{in}$$
$$C_{out} = xy + xC_{in} + yC_{in}$$

Figure 2.19 Truth table, Boolean expressions, and block diagram for a full adder.

Ripple carry adder. One of the most basic addition algorithms is the ripple carry addition algorithm. This algorithm is also easily implemented with full adders. The principle of this algorithm is similar to that of paper and pencil addition. Let $x_3\, x_2\, x_1\, x_0$ and $y_3\, y_2\, y_1\, y_0$ represent two 4-bit binary numbers. To add these numbers on paper, we would add x_0 and y_0 to determine the first digit of the sum. The resulting carry, if any, is added with x_1 and y_1 to determine the next digit of the sum, and similarly for the ensuing carry with x_2 and y_2. This process continues until x_3 and y_3 are added. The final carry, again if any, will become the most significant digit. One way to implement this process is to connect several full adders in series, one full adder for each bit of the numbers to be added. Figure 2.20 presents a ripple carry adder for adding two 4-bit binary numbers ($x_3\, x_2\, x_1\, x_0$ and $y_3\, y_2\, y_1\, y_0$). As shown in Figure 2.20, to ensure that the correct sum is calculated, the output carry of a full adder is connected to the input carry of the next full adder, and the rightmost carry in (C_0) is wired to a constant 0.

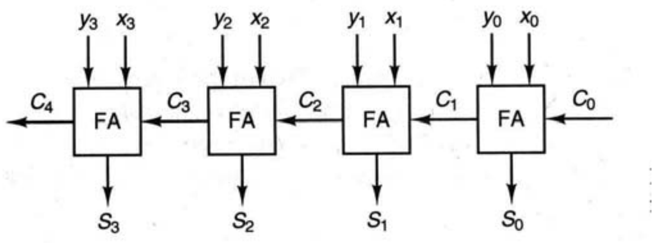


Figure 2.20 Block diagram of a 4-bit ripple carry adder.

The ripple carry adder is a parallel adder because all operands are presented at the same time. The adder gets its name because the carry must ripple through all the full adders before the final value is known. Although this type of adder is very easy and inexpensive to design, it is the slowest adder. This is due to the fact that the carry has to propagate from the least significant bit (LSB) position to the most significant bit

(MSB) position. However, since the ripple carry adder has a simple design, it is sometimes used as small adder cell in building larger adders.

The ripple carry adder can also be used as a subtractor. Subtraction of two binary numbers can be performed by taking the 2's complement of the subtrahend and adding it to the minuend [MAN 91]. (The 2's complement of an *n*-bit binary number *N* is defined as 2^n-*N* [MAN 91].) For example, given two binary numbers *X*=01001 and *Y*=00011, the subtraction *X*-*Y* can be performed as follows:

	$X =$		01001
2's complement of	$Y =$	+	11101
		1	00110
discard the carry;	$X\text{-}Y =$		00110

It is also possible to design a subtractor in a direct manner. In this way, each bit of the subtrahend is subtracted from its corresponding significant minuend bit to form a difference bit. When the minuend bit is 0 and the subtrahend bit is 1, a 1 is borrowed from the next significant position. Just as there are full adders for designing adders, there are full subtractors for designing subtractors. The design of such full subtractors is left as an exercise for the reader (see Problems section).

Carry lookahead adder. This technique increases the speed of the carry propagation in a ripple carry adder. It produces the input carry bit directly, rather than allowing the carries to ripple from full adder to full adder. Figure 2.21 presents a block diagram for a carry lookahead adder that adds two 4-bit integers. In this figure, the carry blocks generate the carry inputs for the full adders. Note that the inputs to each carry block are only the input numbers and the initial carry input (C_0). The Boolean expression for each carry block can be defined by using the carry-out expression of a full adder. For example,

$$C_{i+1} = x_i\, y_i + C_i\,(\, x_i + y_i\,). \qquad (2.1)$$

Thus, C_1 can be generated as

$$C_1 = x_0 y_0 + C_0(\, x_0 + y_0\,).$$

In a similar way, C_2 can be generated as

$$\begin{aligned} C_2 &= x_1 y_1 + C_1(x_1 + y_1) \\ &= x_1\, y_1 + [x_0 y_0 + C_0(\, x_0 + y_0\,)]\,(\, x_1 + y_1\,). \end{aligned}$$

To simplify the expression for each C_i, often two notations *g* and *p* are used. These notations are defined as

$$g_i = x_i\, y_i$$
$$p_i = x_i + y_i$$

Therefore, expression (2.1) can be written as:

$$C_{i+1} = g_i + p_i\, C_i \qquad (2.2)$$

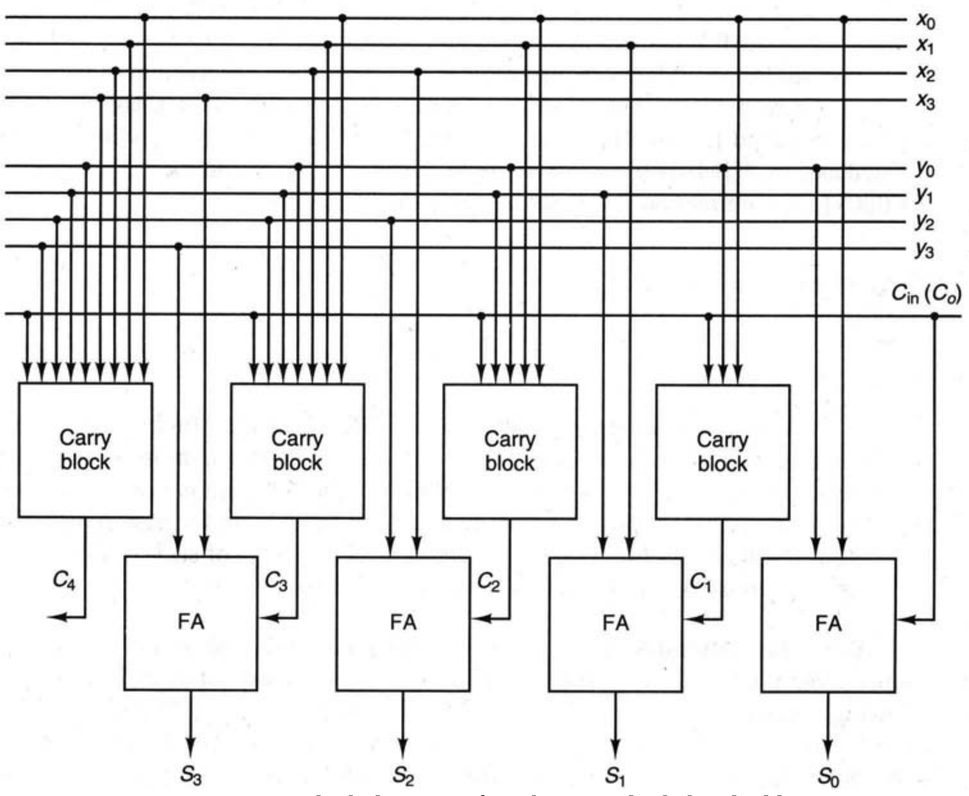


Figure 2.21 Block diagram of a 4-bit carry lookahead adder.

The notation g stands for generating a carry; that is, C_{i+1} is 1 whenever g_i is 1. The notation p stands for propagating the input carry to output carry; that is, when C_i and p_i are 1's, C_{i+1} becomes 1.

Using these notations, we get

$$C_1 = g_0 + C_0\, p_0$$
$$C_2 = g_1 + p_1\, g_0 + p_1\, p_0\, C_0$$
$$C_3 = g_2 + p_2\, g_1 + p_2\, p_1\, g_0 + p_2\, p_1\, p_0\, C_0$$
$$C_4 = g_3 + p_3\, g_2 + p_3\, p_2\, g_1 + p_3\, p_2\, p_1\, g_0 + p_3\, p_2\, p_1\, p_0\, C_0$$

Now we can draw the logic diagram for each carry block. As an example, Figure 2.22 presents the logic diagram for generating C_4. Note there is an AND gate and an OR gate with fan-in of 5 (i.e., they have five inputs). Also, the signal p_3 needs to drive four AND gates (i.e., the OR gate that generates p_3 needs to have at least fan-out of 4). In general, adding two n-bit integers requires an AND gate and an OR gate with fan-in of n+1. It also requires the signal p_{n-1} to drive n AND gates. In practice, these requirements might not be feasible for n>5. In addition to these requirements, Figures 2.21 and 2.22 do not support a modular design for a large n. A modular design requires a structure in which similar parts can be used to build adders of any size.

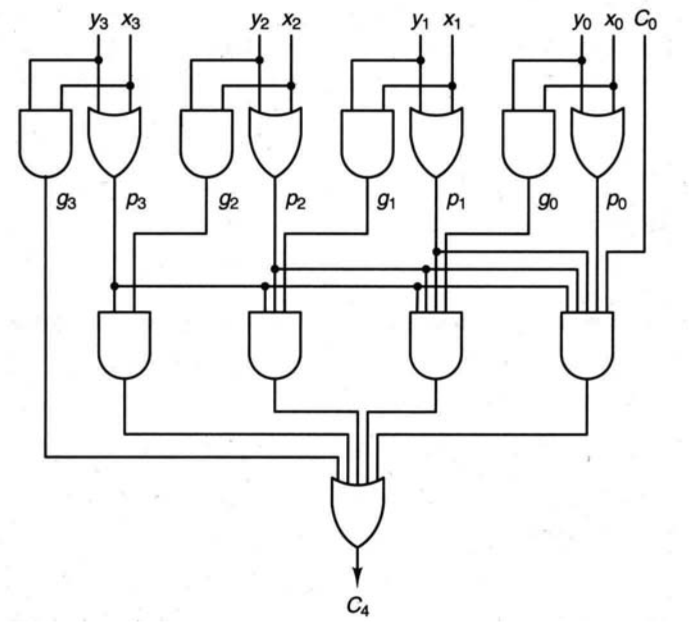


Figure 2.22 Logic diagram for generating carry C_4.

To solve the preceding problems, we limit the fan-in and fan-out to a certain number depending on technology. This requires more logic levels to be added to the lookahead circuitry. For example, if we limit the fan-in to 4 in the preceding example, more gate delay will be needed in order to compute C_4. To do this, we define two new terms, denoted as group generate G_0 and group propagate P_0, where

$$G_0 = g_3 + p_3\, g_2 + p_3\, p_2\, g_1 + p_3\, p_2\, p_1\, g_0$$

$$P_0 = p_3\, p_2\, p_1\, p_0$$

Thus we get

$$C_4 = G_0 + P_0\, C_0.$$

Figure 2.23 presents a block diagram for a carry lookahead adder that adds two 8-bit numbers. In this figure, C_8 is computed similarly to C_4:

$$C_8 = G_1 + P_1\, G_0 + P_1\, P_0\, C_0,$$

where

$$G_1 = g_7 + p_7\, g_6 + p_7\, p_6\, g_5 + p_7\, p_6\, p_5\, g_4, \text{ and } P_1 = p_7 p_6\, p_5 p_4.$$

Although the carry lookahead adder is faster than the ripple carry adder, it requires more space on the chip due to more circuitry.

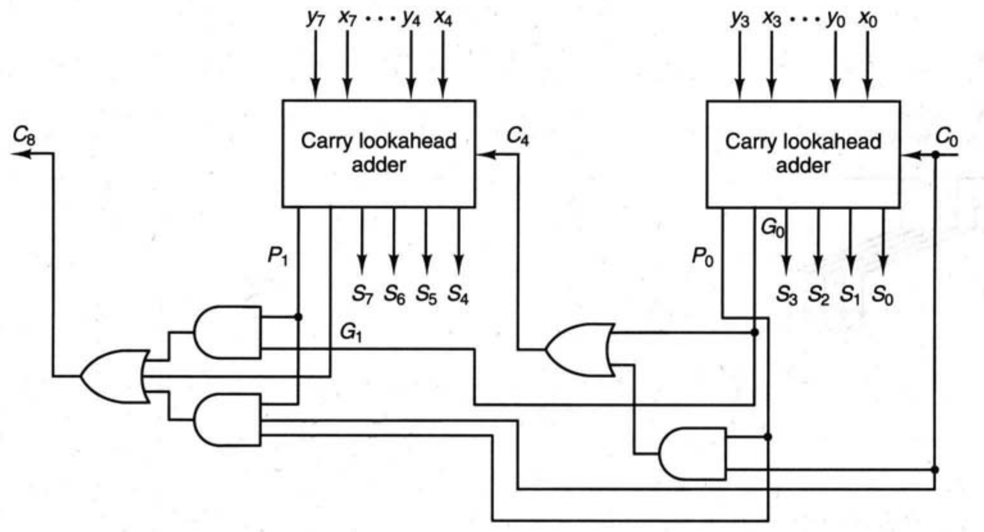


Figure 2.23 Block diagram for an 8-bit carry lookahead adder.

Carry select adder. The carry select adder uses redundant hardware to speed up addition. The increase in speed is created by calculating the high-order half of the sum for both possible input carries, once assuming an input carry of 1 and once assuming an input of 0. When the calculation of the low-order half of the sum is complete and the carry is known, the proper high-order half can be selected.

Assume that two 8-bit numbers are to be added. The low-order 4 bits might be added using a ripple carry adder or a carry lookahead adder (see Figure 2.24). Simultaneously, the high-order bits will be added once, assuming an input carry of 1 and once assuming an input carry of zero. When the output carry of the low-order bits is known, it can be used to select the proper bit pattern for the sum. Obviously, more adders are needed; therefore, more space is required on the chip [HEN 90]. By limiting the number of bits added at one time, the carry select adder overcomes the carry lookahead adder's complexity in high-order carry calculations.

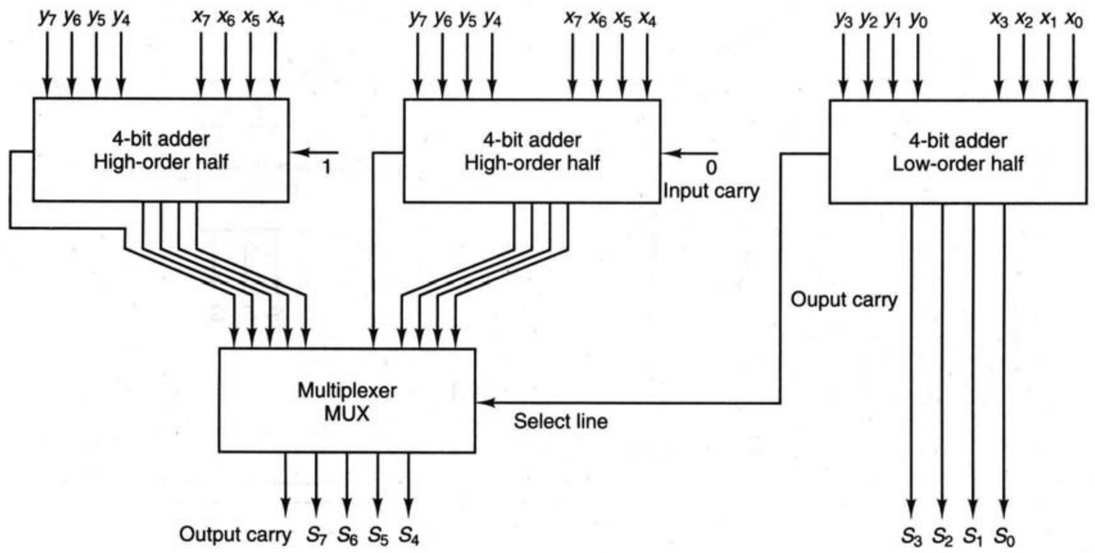


Figure 2.24 Block diagram of an 8-bit carry select adder.

Carry save adder. This type of adder is useful when more than two numbers are added. For example, when there are four numbers (*X*, *Y*, *Z*, and *W*) to be added, the carry save adder first produces a sum and a saved carry for the first 3 numbers. Assuming that *X* = 0101, *Y* = 0011, and *Z* = 0100, the produced sum and saved carry are

	0101	*X*
	0011	*Y*
+	0100	*Z*
	0010	sum
	1010	saved carry

In the next step, the sum, the saved carry, and the fourth number (*W*) are added in order to produce a new sum and a new saved carry. Assuming that *W* = 0001,

	0010	sum
	1010	saved carry
+	0001	*W*
	1001	new sum
	0100	new saved carry

In the last step, a carry lookahead adder is used to add the new sum and the new saved carry. Putting all these steps together, a multioperand adder (often used in multiplier circuits to accumulate partial products) can be designed. For example, Figure 2.25 presents a block diagram for adding four numbers. Notice that this design includes two carry save adders, each having a series of full adders.

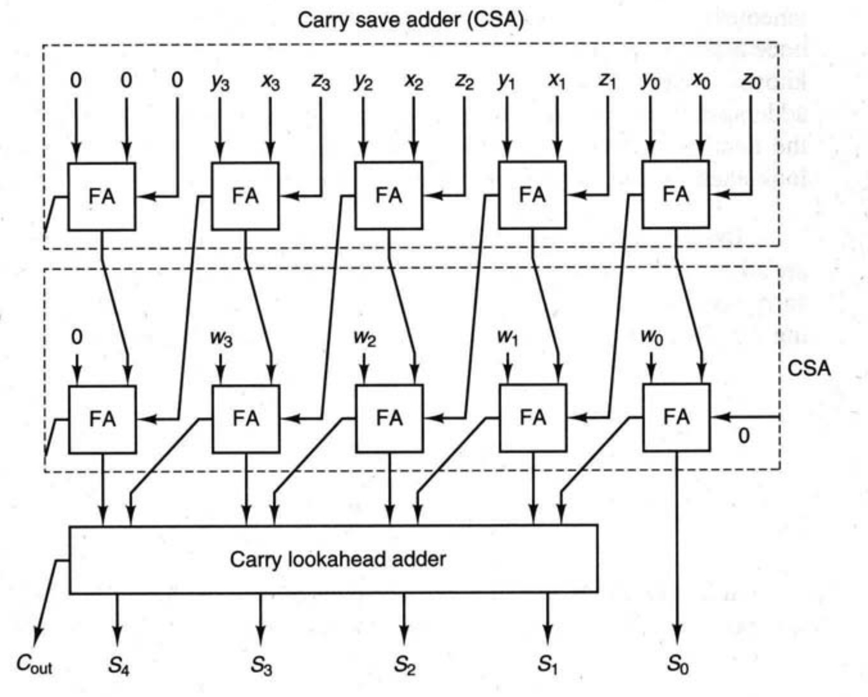


Figure 2.25 Block diagram of an adder for adding four 4-bit numbers.

Figure 2.26 shows how to use only one carry save adder (one level) to add a set of numbers. First, three numbers are applied to the inputs *X*, *Y*, and *Z*. The sum and carry generated by these three numbers are fed back to the inputs *X* and *Y*, and then added to the fourth number, which is applied to *Z*. This process continues until all the numbers are added.

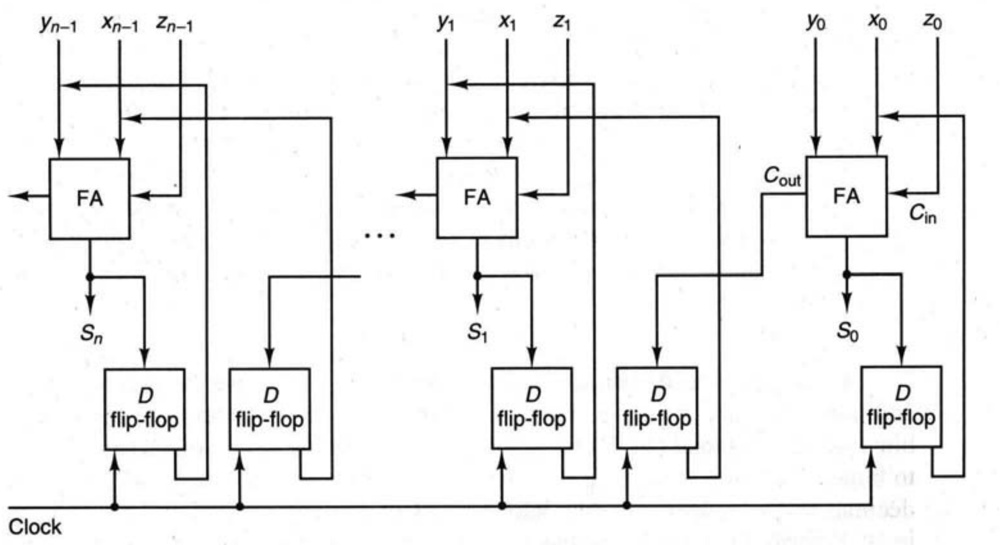


Figure 2.26 Block diagram of an *n*-bit carry save adder for adding a set of numbers.

Binary-coded decimal number addition. So far we have only considered addition of binary numbers. However, sometimes the numbers are represented in binary-coded decimal (BCD) form. In such cases, rather than converting the numbers to binary form and then using a binary adder to add them, it is more efficient to use a decimal adder. A decimal adder adds two BCD digits in parallel and produces a sum in BCD form. Given the fact that a BCD digit is between 0 and 9, whenever the sum exceeds 9 the result is corrected by adding value 6 to it. For example, in the following addition, the sum of 4 (0100) and 7 (0111) is greater than 9; in this case the corresponding intermediate sum digit is corrected by adding 6 to it:

```
      2   4   5
  +   4   7   4
  -------------
      6   B   9   <------  intermediate sum
      ▲   ▲   ▲                       digits
      |   |   └----------  not greater than 9
      |   └--------------      greater than 9
      └------------------  not greater than 9
  +   0   6   0
  -------------
      7   1   9   <------        decimal sum
```

Figure 2.27 shows a decimal adder based on 4-bit adder. In addition to two 4-bit adders, the adder includes some gates to perform logic correction for intermediate sum digits that are equal to or greater than 10 (1010); we add value 6 to these digits.

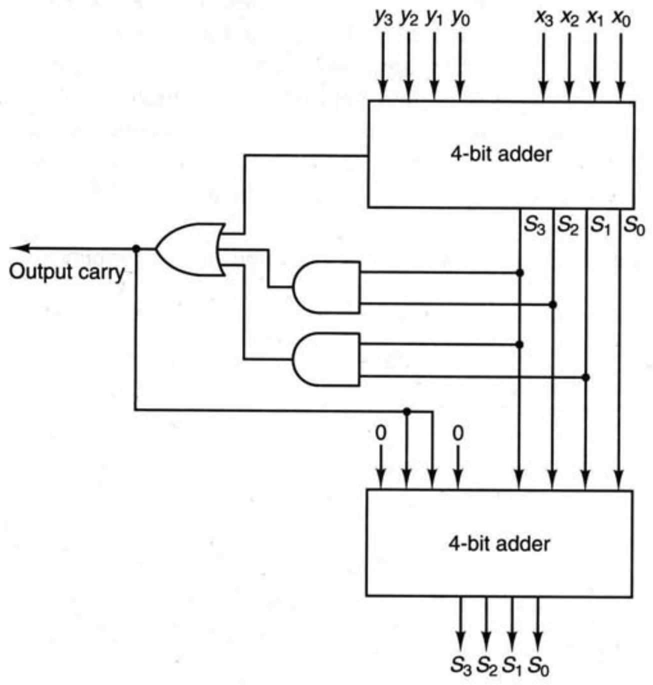


Figure 2.27 Block diagram of a 4-bit binary-coded decimal adder.

Serial adder. The serial adder performs the addition step by step from the least significant bit to the most significant bit. The output will be produced bit by bit. As is shown in Figure 2.28, a serial adder consists of only a full adder and a *D* flip-flop. The *D* flip-flop is used to propagate the carry of sum of *i*th input bits to the sum of (*i*+1)th input bits. Compared to the ripple carry adder, the serial adder is even slower, but it is simpler and inexpensive.

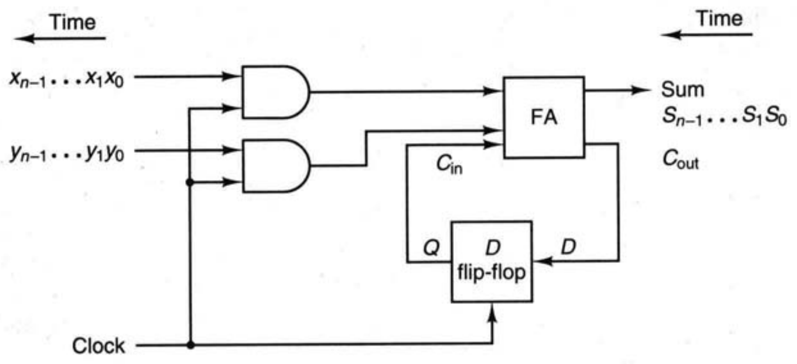


Figure 2.28 Block diagram of a serial adder.

2.5.2 Multiplication

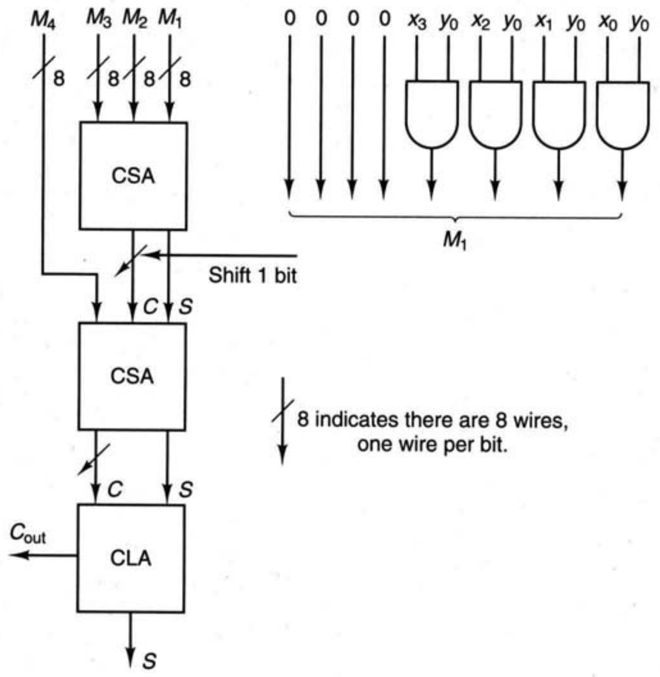
After addition and subtraction functions, multiplication is one of the most important arithmetic functions. Statistics suggest that in some large scientific programs multiplication occurs as frequently as addition and subtraction. Multiplication of two *n*-bit numbers (or binary fractions) in its simplest form can be done by addition of *n* partial products.

Wallace showed that the partial products can be added in a fast and economical way using an architecture called a *Wallace tree* [WAL 64]. The Wallace tree adds the partial products by using multilevel of carry save adders. Assuming that all partial products are produced simultaneously, in the first level the Wallace tree groups the numbers into threes and uses a carry save adder to add the numbers in each group. Thus the problem of adding n numbers reduces to the problem of adding $2n/3$ numbers. In the second level, the $2n/3$ resulting numbers are again grouped into three and added by carry save adders. This process continues until there are only two numbers left to be added. (Often the carry lookahead adder is used to add the last two numbers.) Since each level reduces the number of terms to be added by a factor of 1.5 (or a little less when the number of terms is not a multiple of three), the multiplication can be completed in a time proportional to $\log_{1.5} n$.

For example let's consider multiplication of two unsigned 4-bit numbers as shown next:

				x_3	x_2	x_1	x_0	
			*	y_3	y_2	y_1	y_0	
0	0	0	0	x_3y_0	x_2y_0	x_1y_0	x_0y_0	-----> M_1
0	0	0	x_3y_1	x_2y_1	x_1y_1	x_0y_1	0	-----> M_2
0	0	x_3y_2	x_2y_2	x_1y_2	x_0y_2	0	0	-----> M_3
0	x_3y_3	x_2y_3	x_1y_3	x_0y_3	0	0	0	-----> M_4

As is shown in Figure 2.29, the inputs to the carry save adders are M_1, M_2, M_3, and M_4. Also, Figure 2.29 shows how to generate M_1. M_1 consists of 8 bits where the leftmost 4 bits are 0 and the rightmost 4 bits are x_0y_0, x_1y_0, x_2y_0, and x_3y_0. M_2, M_3, and M_4 can be generated in a similar way.

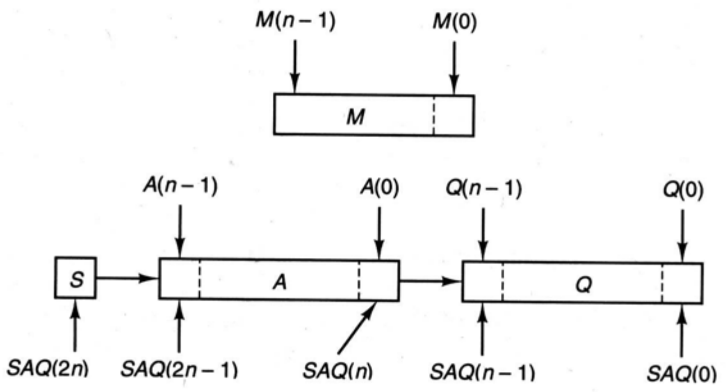


M_4
M_3 M_2 M_1
8
8
8
8
0 0 0 0 x_3 y_0 x_2 y_0 x_1 y_0 x_0 y_0
CSA
M_1
Shift 1 bit
C
S
CSA
8 indicates there are 8 wires, one wire per bit.
C
S
C_{out}
CLA
S

Figure 2.29 Block diagram of a 4-bit multiplier using carry save adders.

The discussion that follows will explore different techniques for multiplication, such as *shift-and-add*, *booth's technique*, and *array multiplier*.

Shift-and-add multiplication. Shift-and-add multiplication, also called the pencil-and-paper method, is a simple but slow method. This method adds the multiplicand Y to itself X times, where X denotes the multiplier. Assume that X and Y are n-bit 2's complement numbers that are stored in n-bit registers Q and M, respectively. Also, assume that there are an n-bit register A and a 1-bit register S. Furthermore, let registers S, A, and Q be connected to each other as shown:



By putting three registers *S*, *A*, and *Q* together, a larger register, *SAQ*, is constructed for storing the partial products and final result. The following code represents the steps of shift-and-add multiplication.

```
A=0;  S=0;                          -- Initialize the registers A and S.
M=Y;                                -- M contains multiplicand.
Q=X;                                -- Q contains multiplier.
if(Q(n-1) = '1') then               -- If the multiplier is negative, then
        M = -M;                     -- replace the contents of M and Q
        Q  = -Q;                    -- with their 2's complement.
end if;                             -- In this way the contents of Q
                                    -- will always be positive.
for i in 1 to n loop                -- Add the contents of M to A,
        if(Q(0) = '1') then         -- Q times.
                A=M + A;
                S= M(n-1);          -- Set S equal to the sign of M.
        end if;
        for j in 1 to 2n loop       -- Shift SAQ register 1 to the
                SAQ(j-1) = SAQ(j);  -- right, and don't change S.
        end loop;                   -- (Register SAQ has 2n+1 bits
                                    -- and is concatenation of the
end loop;                           -- three registers S, A, and Q.)
```

For example, let *X*=01101 and *Y*=10111 be two 2's complement numbers; that is, *X* and *Y* represent the decimal values 13 and -9, respectively. (Note that if we want to represent the same values of 13 and -9 in more than 5 bits we need to pad the extra bits, depending on what the sign of the value is. Positive values are padded with 0, and negative values with 1. So 13 and -9 in 8 bits are 00001101 and 11110111, respectively.) The contents of the registers *S*, *A*, and *Q* at each cycle are as follows:

Cycle	S	A	Q	operation
	0	00000	0110**1**	Initialization
1	1	10111	01101	Add *M* to *A*
	1	11011	1011**0**	Shift right
2	1	11101	1101**1**	Shift right
3	1	10100	11011	Add *M* to *A*
	1	11010	0110**1**	Shift right
4	1	10001	01101	Add *M* to *A*
	1	11000	1011**0**	Shift right

5	1	11100	01011	Shift right (result)

Booth's technique. In the shift-and-add form of a multiplier, the multiplicand is added to the partial product at each bit position where a 1 occurs in the multiplier. Therefore, the number of times that the multiplicand is added to the partial product is equal to the number of 1's in the multiplier. Also, when the multiplier is negative, the shift-and-add method requires an additional step to replace the multiplier and multiplicand with their 2's complement.

To increase the speed of multiplication, Booth discovered a technique that reduces the addition steps and eliminates the conversion of the multiplier to positive form [BOO 51]. The main point of Booth's multiplication is that the string of 0's in the multiplier requires no addition, but just shifting, and the string of 1's can be treated as a number with value *L*-*R*, where *L* is the weight of the zero before the leftmost 1 and *R* is the weight of the rightmost 1. So, if the number is 01100, then $L=2^4=16$ and $R=2^2=4$. That is, the value of 01100 can be represented as 16 - 4. Let's take a look at an example; let multiplier *X*=10011 and multiplicand *Y*=10111 in 2's complement representation. The multiplier *X* can be represented as

$$X= -2^4 + (2^2 - 2^0).$$

Thus

$$X*Y = [-2^4 + (2^2 - 2^0)]*Y = -2^4Y + 2^2Y -2^0Y.$$

Based on this string manipulation, the Booth's multiplier considers every two adjacent bits of the multiplier to determine which operation to perform. The possible operations are

X_{i+1}	X_i	Operation
0	0	Shift right
1	0	Subtract multiplicand and shift right
1	1	Shift right
0	1	Add multiplicand and shift right

For example, let *X*=*Q*=10011, *Y*=*M*=10111, and -*M*= 2's complement of *Y* = 01001. Initially, a 0 is placed in front of the rightmost bit of *Q*. At each cycle of operation, based on the rightmost 2 bits of *Q*, one of the preceding operations is performed. The contents of the registers *S*, *A*, and *Q* at each cycle are shown next:

Cycle	*S*	*A*	*Q*	Operation
	0	00000	10011**0**	Initialization Add an extra bit to *Q*.
1	0	01001	100110	Subtract *M* from *A*, in other words add -*M* to *A*
	0	00100	1100**11**	Shift right
2	0	00010	0110**01**	Shift right
3	1	11001	011001	Add *M* to *A*
	1	11100	1011**00**	Shift right
4	1	11110	0101**10**	Shift right
5	0	00111	010110	Subtract *M* from *A*
	0	00011	101011	Shift right (result)

Often, in practice, both Booth's technique and the Wallace tree method are used for producing fast multipliers. Booth's technique is used to produce the partial products, and the Wallace tree is used to add them. For example, assume that we want to multiply two 16-bit numbers. Let $X=x_{15}...x_1x_0$ and $Y=y_{15}...y_1y_0$

denote the multiplier and the multiplicand, respectively. Figure 2.30 represents a possible architecture for multiplying such numbers. In the first level of the tree, the partial products are produced by using a scheme similar to Booth's technique, except that 3 bits are examined at each step instead of 2 bits. (Three-bit scanning is an extension to Booth's technique. It is left as an exercise for the reader.) With a 16-bit multiplier, the 3 examined bits for producing each partial product are

x_1x_00
$x_3x_2x_1$
$x_5x_4x_3$
$x_7x_6x_5$
$x_9x_8x_7$
$x_{11}x_{10}x_9$
$x_{13}x_{12}x_{11}$
$x_{15}x_{14}x_{13}$

The partial products are then added by a set of CSAs arranged in the form a Wallace tree.

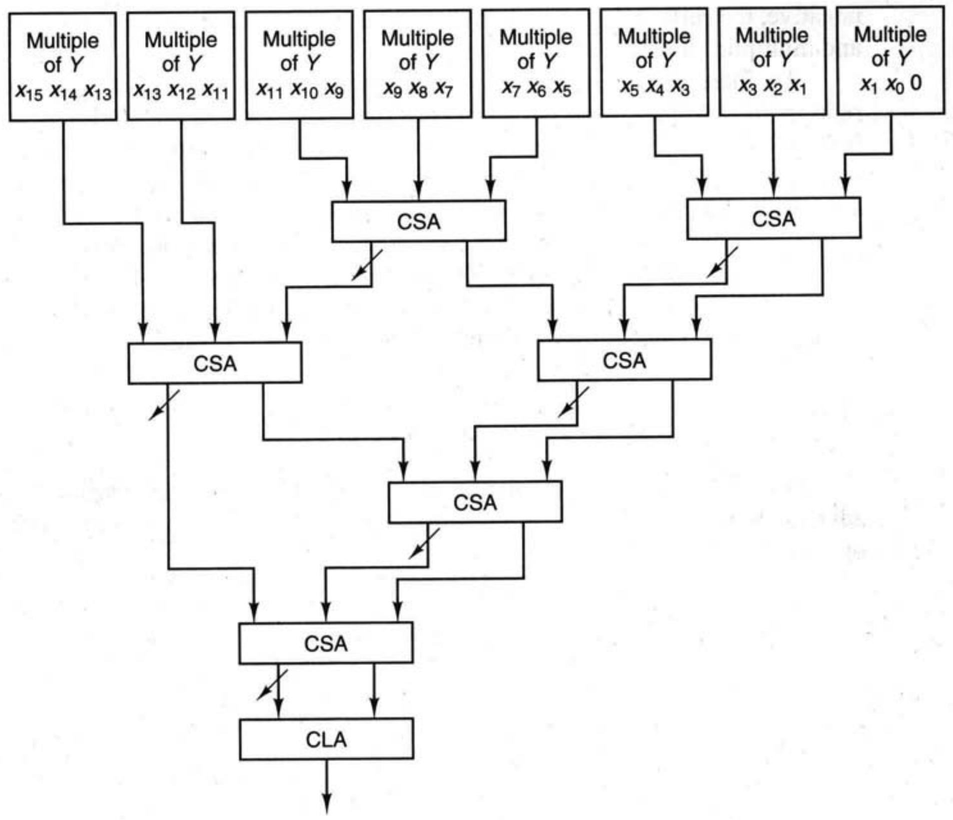
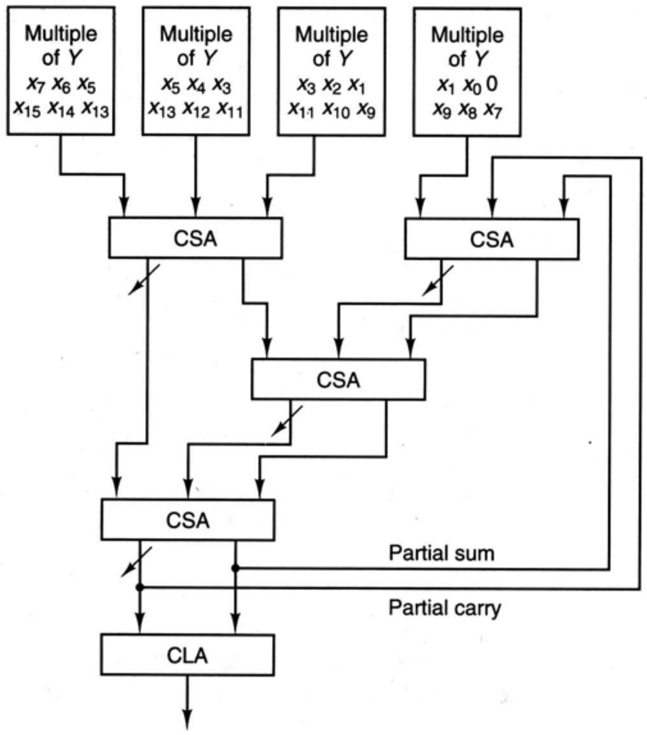

Multiple of Y $x_{15}\ x_{14}\ x_{13}$
Multiple of Y $x_{13}\ x_{12}\ x_{11}$
Multiple of Y $x_{11}\ x_{10}\ x_9$
Multiple of Y $x_9\ x_8\ x_7$
Multiple of Y $x_7\ x_6\ x_5$
Multiple of Y $x_5\ x_4\ x_3$
Multiple of Y $x_3\ x_2\ x_1$
Multiple of Y $x_1\ x_0\ 0$
CSA
CSA
CSA
CSA
CSA
CSA
CLA

Figure 2.30 A multiplier combining a Wallace tree and Booth's technique.

To reduce the required hardware in the preceding structure, especially for large numbers, often the final product is obtained by passing through the Wallace tree several times. For example, Figure 2.31 represents an alternative design for our example. This design requires two passes. The first pass adds the four partial products that result from the 8 least significant bits of the multiplier. In the second pass, the resulting partial sum and carry are fed back into the top of the tree to be added to the next four partial products.



Multiple of Y $x_7\ x_6\ x_5$ $x_{15}\ x_{14}\ x_{13}$
Multiple of Y $x_5\ x_4\ x_3$ $x_{13}\ x_{12}\ x_{11}$
Multiple of Y $x_3\ x_2\ x_1$ $x_{11}\ x_{10}\ x_9$
Multiple of Y $x_1\ x_0\ 0$ $x_9\ x_8\ x_7$
CSA
CSA
CSA
CSA
Partial sum
Partial carry
CLA

Figure 2.31 A two pass multiplier combining Wallace tree and Booth's technique.

Array multiplier. Baugh and Wooley have proposed a notable method for multiplying 2's complement numbers [BAU 73]. They have converted the 2's complement multiplication to an equivalent parallel array addition problem. In contrast to the conventional two's-complement multiplication, which has negative and positive partial products, Baugh-Wooley's algorithm generates only positive partial products. Using this algorithm, we can add the partial products with an array of full adders in a modular way; this is an important feature in VLSI design. With the advent of VLSI design, the implementation of an array of similar cells is very easy and economical. (In general, the placement of a set of same-sized cells, such as full adders, requires an ideal area on a chip.)

Before we study the algorithm, we should know about some of the equivalent representations of a 2's complement number. Given an n-bit 2's complement $X = (x_{n-1}, \ldots, x_0)$, the value of X, X_v, can be represented as

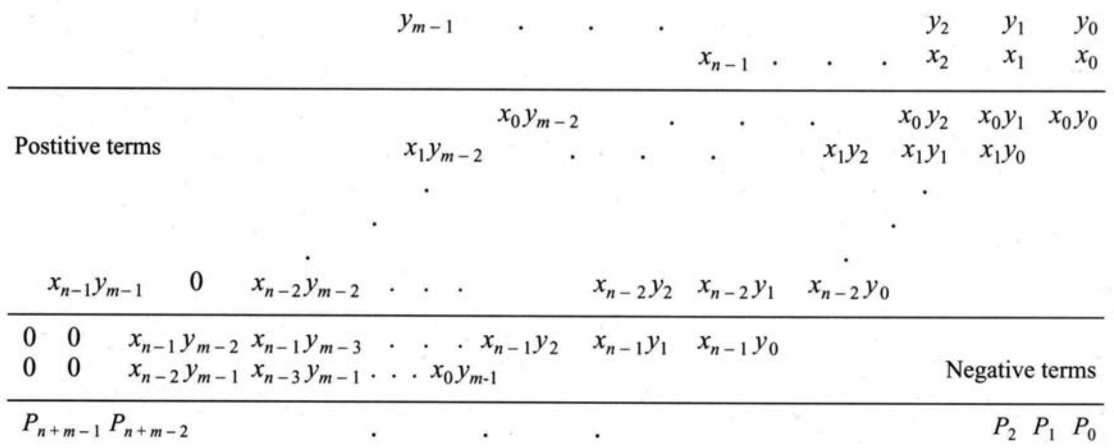
$$X_v = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$$

$$\begin{aligned} X_v &= -x_{n-1}2^{n-1} + (2^{n-1} - 2^{n-1}) + \sum_{i=0}^{n-2} x_i 2^i \\ &= -(x_{n-1}-1)\,2^{n-1} - 2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i \\ &= (1-x_{n-1})\,2^{n-1} - (1 + \sum_{i=0}^{n-2} 2^i) + \sum_{i=0}^{n-2} x_i 2^i \\ &= (1-x_{n-1})\,2^{n-1} - 1 - \sum_{i=0}^{n-2} (1-x_i)\,2^i \\ &= \bar{x}_{n-1}2^{n-1} - 1 - \sum_{i=0}^{n-2} \bar{x}_i 2^i \end{aligned} \tag{2.3}$$

Now, let the n-bit $X = (x_{n-1}, \ldots, x_0)$ be the multiplier, and the m-bit $Y = (y_{m-1}, \ldots, y_0)$ be the multiplicand. The value of the product $P = (p_{n+m-1}, \ldots, p_0)$ can be represented as:

$$
\begin{aligned}
P_v &= Y_v\, X_v \\
&= (-\,y_{m-1}2^{m-1} + \sum_{i=0}^{m-2} y_i\, 2^i)(-\,x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i\, 2^i) \\
&= y_{m-1}x_{n-1}2^{m+n-2} + \sum_{i=0}^{m-2}\sum_{j=0}^{n-2} y_i\, x_j\, 2^{i+j} - \sum_{i=0}^{m-2} x_{n-1}\, y_i\, 2^{n-1+i} - \sum_{i=0}^{n-2} y_{m-1}\, x_i\, 2^{m-1+i}.
\end{aligned}
$$

This equation shows that P_v is formed by adding the positive partial products and subtracting the negative partial products. As shown in Figure 2.32, by placing all the negative terms in the last two rows, P_v can be computed by adding the first n-2 rows and subtracting the last two rows.



y_{m-1} . . . y_2 y_1 y_0
x_{n-1} . . . x_2 x_1 x_0
Postitive terms
x_0y_{m-2} . . . x_0y_2 x_0y_1 x_0y_0
x_1y_{m-2} . . . x_1y_2 x_1y_1 x_1y_0
$x_{n-1}y_{m-1}$ 0 $x_{n-2}y_{m-2}$ . . . $x_{n-2}y_2$ $x_{n-2}y_1$ $x_{n-2}y_0$
0 0 $x_{n-1}y_{m-2}$ $x_{n-1}y_{m-3}$ . . . $x_{n-1}y_2$ $x_{n-1}y_1$ $x_{n-1}y_0$
0 0 $x_{n-2}y_{m-1}$ $x_{n-3}y_{m-1}$ . . . x_0y_{m-1}
Negative terms
P_{n+m-1} P_{n+m-2} . . . P_2 P_1 P_0

Figure 2.32 Positive and negative partial products for multiplying an n-bit number with an m-bit number.

We would like to change the negative terms to positive in order to do simple addition instead of subtraction. The negative term

$$-\sum_{i=0}^{m-2} x_{n-1}\, y_i\, 2^{n-1+i}$$

can be rewritten as

$$-\,2^{n-1}(-0 * 2^{m-1} + \sum_{i=0}^{m-2} x_{n-1}\, y_i\, 2^i).$$

Using (2.3), this term can be replaced with

$$2^{n-1}(-1 * 2^{m-1} + 1 + \sum_{i=0}^{m-2} \overline{x_{n-1}\, y_i}\, 2^i),$$

or

$$2^{n-1}(-1 * 2^m + 1 * 2^{m-1} + 1 + \sum_{i=0}^{m-2} \overline{x_{n-1}\, y_i}\, 2^i), \tag{2.4}$$

which has the following values:

$$
\begin{array}{ll}
0, & \text{when } x_{n-1}=0, \\
2^{n-1}(-\,2^m + 2^{m-1} + 1 + \sum_{i=0}^{m-2} \overline{y_i}\, 2^i), & \text{when } x_{n-1}=1.
\end{array}
$$

Therefore, (2.4) can be rewritten as

$$2^{n-1}(-\,2^m + 2^{m-1} + \overline{x_{n-1}}\, 2^{m-1} + x_{n-1} + \sum_{i=0}^{m-2} x_{n-1}\, \overline{y_i}\, 2^i).$$

So the second to last row in Figure 2.32 can be replaced by

$$
\begin{array}{cccccccc}
0 & \overline{x_{n-1}} & x_{n-1}\overline{y_{m-2}} & x_{n-1}\overline{y_{m-3}} & \cdots & x_{n-1}\overline{y_2} & x_{n-1}\overline{y_1} & x_{n-1}\overline{y_0} \\
0 & 1 & 0 & 0 & \cdots & 0 & 0 & x_{n-1}
\end{array}
$$

Similarly, the last row in Figure 2.32 can be replaced by

$$\begin{array}{cccccc} 0 & \bar{y}_{m-1} & \bar{x}_{n-2}\,y_{m-1} & \bar{x}_{n-3}\,y_{m-1} & \cdots & \bar{x}_0\,y_{m-1} \\ 1 & 1 & 0 & 0 & \cdots & y_{m-1} \end{array}$$

These replacements are shown in Figure 2.33. Notice that in this figure all partial products are positive. Therefore, the product *P* can be obtained by adding the rows; this is shown in Figure 2.34 for *m*=6 and *n*=4.

$$\begin{array}{llllllllll}
 & & y_{m-1} & \cdot & \cdot & \cdot & & y_2 & y_1 & y_0 \\
 & & & & x_{n-1} & \cdot\ \cdot\ \cdot & & x_2 & x_1 & x_0 \\
\hline
 & & & x_0 y_{m-2} & & \cdot\ \cdot\ \cdot & & x_0 y_2 & x_0 y_1 & x_0 y_0 \\
 & & x_1 y_{m-2} & & \cdot\ \cdot\ \cdot & & x_1 y_2 & x_1 y_1 & x_1 y_0 & \\
 & & \cdot & & & & & \cdot & & \\
 & & \cdot & & & & \cdot & & & \\
 & & \cdot & & & & \cdot & & & \\
x_{n-1} y_{m-1} & 0 & x_{n-2} y_{m-2} & \cdot\ \cdot\ \cdot & & x_{n-2} y_2 & x_{n-2} y_1 & x_{n-2} y_0 & & \\
\bar{x}_{n-1} & x_{n-1}\bar{y}_{m-2} & x_{n-1}\bar{y}_{m-3} & \cdot\ \cdot\ \cdot & x_{n-1}\bar{y}_2 & x_{n-1}\bar{y}_1 & x_{n-1}\bar{y}_0 & & & \\
\bar{y}_{m-1} & \bar{x}_{n-2}\, y_{m-1} & \bar{x}_{n-3} y_{m-1} & \ldots\ \bar{x}_0\, y_{m-1} & & & & & & \\
1 & & & y_{m-1} & & & x_{n-1} & & & \\
\hline
P_{n+m-1} & P_{n+m-2} & & \cdot & \cdot & \cdot & & P_2 & P_1 & P_0
\end{array}$$

Figure 2.33 Using only positive partial products for multiplying for an *n*-bit number with an *m*-bit number.

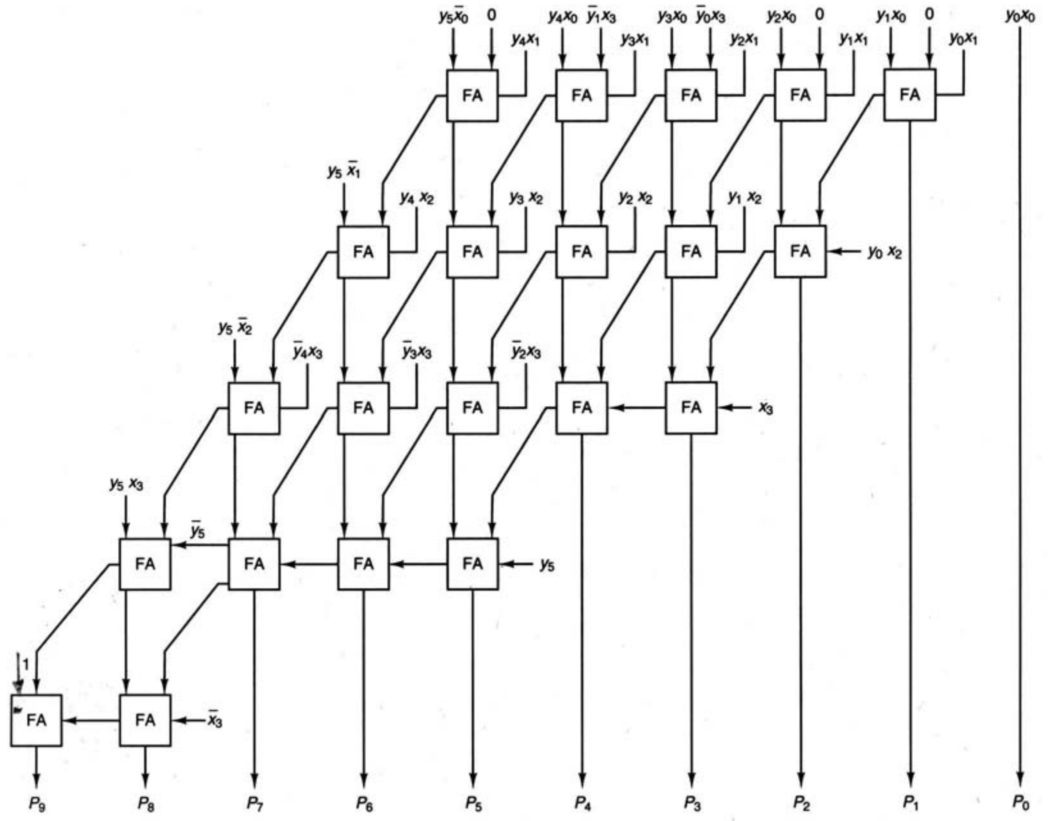


Figure 2.34 Block diagram of an array multiplier for multiplying a 4-bit number with a 6-bit number.

2.5.3 Floating-point Representation

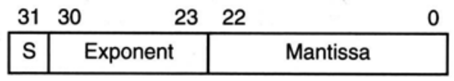
The range of numbers available for a word is strictly limited by its size. A 32-bit word has a range of 2^{32} different numbers. If the numbers are considered as integers, it is necessary to scale the numbers of many problems in order to represent the fractions. One solution can be to increase the size of the word to get a better range. However, this solution increases the storage space and computing time. A better solution is to use an automatic scaling technique, known as the floating-point representation (also referred to as scientific notation).

In general, a floating-point number can be represented in the following form:

$$\pm m * b^{e}$$

where *m*, called the *mantissa*, represents the fraction part of the number and is normally represented as a signed binary fraction. The *e* represents the exponent, and the *b* represents the base (radix) of the exponent.

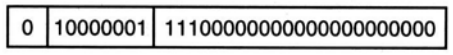
This representation can be stored in a binary word with three fields: sign, mantissa, and exponent. For example, assuming that the word has 32 bits, a possible assignment of bits to each field could be



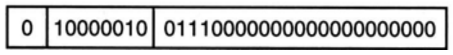
Notice that no field is assigned to the exponent base *b*. This is because the base *b* is the same for all the numbers, and often it is assumed to be 2. Therefore, there is no need to store the base. The sign field consists of 1 bit and indicates the sign of the number, 0 for positive and 1 for negative. The exponent consists of 8 bits, which can represent numbers 0 through 255. To represent positive and negative exponents, a fixed value, called *bias*, is subtracted from the exponent field to obtain the true exponent. In our example, assuming the bias value to be 128, the true exponents are in the range from -128 to +127. (The exponent -128 is stored as 0, and the exponent +127 is stored as 255 in the exponent field.) In this way, before storing an exponent in the exponent field, the value 128 should be added to the exponent. For example, to represent exponent +4, the value 132 (128+4) is stored in the exponent field, and the exponent -12 is stored as 116 (128-12).

The mantissa consists of 23 bits. Although the radix point (or binary point) is not represented, it is assumed to be at the left side of the most significant bit of the mantissa. For example, when *b* = 2, the floating-point number 1.75 can be represented in any of the following forms:

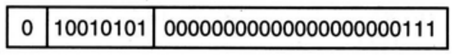
$$+0.111_{*}2^{1} \qquad (2.5)$$



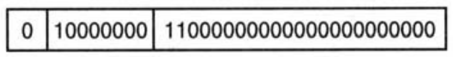
$$+0.0111_{*}2^{2} \qquad (2.6)$$



$$+0.00000000000000000000111_{*}2^{21} \quad (2.7)$$



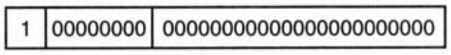
To simplify the operation on floating-point numbers and increase their precision, floating-point numbers are always represented in normalized form. A floating-point number is said to be *normalized* if the leftmost bit (most significant bit) of the mantissa is 1. Therefore, in the three representations for 1.75, the first representation, which is normalized, is used. Since the leftmost bit of the mantissa of a normalized floating-point number is always 1, this bit is often not stored and is assumed to be a hidden bit to the left of the radix point. This allows the mantissa to have one more significant bit. That is, the stored mantissa *m* will actually represents the value 1.*m*. In this case, the normalized 1.75 will have the following form: $+1.11_{*}2^{0}$



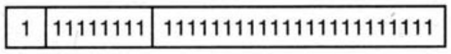
Assuming a hidden bit to the left of the radix point in our floating-point format, a nonzero normalized number represents the following value:

$(-1)^s * (1.m) * 2^{e-128}$, where s denotes the sign bit.

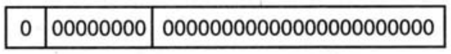
Furthermore, the format can represent the following range of numbers.
Smallest negative number: $-1.0 * 2^{-128}$



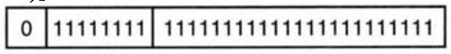
Largest negative number: $-[1+(1 - 2^{-23})] * 2^{127}$



Smallest positive number: $1.0 * 2^{-128}$



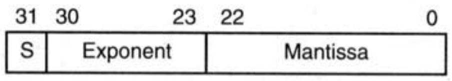
Largest positive number: $[1+(1-2^{-23})] * 2^{127}$



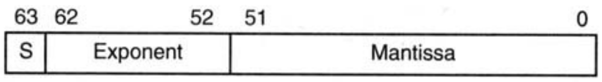
Computation on floating-point numbers may produce results that are either larger than the largest representable value or smaller than the smallest representable value. When the result is larger than the allowable representation, an *overflow* is said to have occurred, and when it is smaller than the allowable representation, an *underflow* is said to have occurred. Processors have certain mechanisms for detecting, handling, and signaling overflow and underflow.

The problem with the preceding format is that there is no representation for the value 0. This is because a zero cannot be normalized since it does not contain a nonzero digit. However, in practice the floating-point representations reserve a special bit pattern for 0. Often a zero is represented by all 0's in the mantissa and exponent. A good example of such bit pattern assignment for 0 is the standard formats defined by the IEEE Computer Society [IEE 85].

The *IEEE 754 floating-point standard* defines both a single-precision (32-bit) and a double-precision (64-bit) format for representing floating-point numbers.



Single-precision floating point



Double-precision floating point

In both formats the implied exponent base (b) is assumed to be 2. The single-precision format allocates 8 bits for exponent, 23 bits for mantissa, and 1 bit for sign. The exponent values 0 and 255 are used for representing special values, including 0 and infinity. The value 0 is represented as all 0's in the mantissa and exponent. Depending on the sign bit the value 0 can be represented as +0 or -0. Infinity is represented by storing all 0's in the mantissa and 255 in the exponent. Again, depending on sign bit, $+\infty$ and $-\infty$ are possible. When the exponent is 255 and the mantissa is nonzero, a not-a-number (NaN) is represented. The NaN is a symbol for the result of invalid operations, such as taking the square root of a negative number, subtracting infinity from infinity, or dividing a zero by a zero. For exponent values from 1 through 254, a bias value 127 is used to determine the true exponent. Such biasing allows a true exponent from -126 through +127. There is a hidden bit on the left of the radix point for normalized numbers, allowing an effective 24-bit mantissa. Thus the value of a normalized number represented by a single-precision format is

$$(-1)^s * (1.m) * 2^{e-127}$$

In addition to representation of positive and negative exponents, the bias notation also allows floating-point numbers to be sorted based on their bit patterns. Sorting can occur because the largest negative exponent is represented as 00000001, and the largest positive exponent is represented as 11111110. Given the fact that the exponent field is placed before the mantissa and the mantissa is normalized, numbers with bigger exponents are larger than numbers with smaller exponents. In other words, any bigger number has a larger bit pattern than a smaller number.

The double-precision format can be defined similarly. This format allocates 11 bits for exponent and 52 bits for mantissa. The exponent bias is 1023. The value of a normalized number represented by a double-precision format is

$$(-1)^s * (1.m) * 2^{e-1023}$$

Floating-point addition. The difficulty involved in adding two floating-point numbers stems from the fact that they may have different exponents. Therefore, before the two numbers can be properly added together, their exponents must be equalized. This involves comparing the magnitude of two exponents and then *aligning* the mantissa of the number that has smaller magnitude of exponent. The alignment is accomplished by shifting the mantissa a number of positions to the right. The number of positions to shift is determined by the magnitude of difference in the exponents. For example, to add $1.1100 * 2^4$ and $1.1000 * 2^2$, we proceed as follows. The number $1.1000 * 2^2$ has a smaller exponent, so it is rewritten as $0.0110 * 2^4$ by aligning the mantissa. Now the addition can be performed as follows:

$$\begin{array}{rrl} & 1.1100 & *2^4 \\ + & 0.0110 & *2^4 \\ & \text{---------} & \\ & 10.0010 & *2^4 \end{array}$$

Notice that the resulting number is not normalized. In general, the addition of any pair of floating-point numbers may result in an unnormalized number. In this case, the resulting number should be normalized by shifting the resulting mantissa and adjusting the resulting exponent. Whenever the exponent is increased or decreased, we should check to determine whether an overflow or underflow has occurred in this field. If the exponent cannot fit in its field, an exception is issued. (Exceptions are defined later in this chapter.) We also need to check the size of the resulting mantissa. If the resulting mantissa requires more bits than its field, we must round it to the appropriate number of bits. In the example, where 4 bits after radix point are kept, the final result will be $1.0001 * 2^5$.

Figure 2.35 represents a block diagram for floating-point addition. The result of comparison of the magnitude of two exponents directs the align unit to shift the proper mantissa. The aligned mantissa and the other mantissa are then fed to the add/subtract unit for the actual computation. (If one number has a negative sign, what should actually be performed is a subtraction operation.) The resulting number is then sent to the result normalization-round unit. The result is normalized and rounded when needed. The detail of the function of this unit is shown in Figure 2.36. (A good explanation on rounding procedure can be found in [FEL 94].)

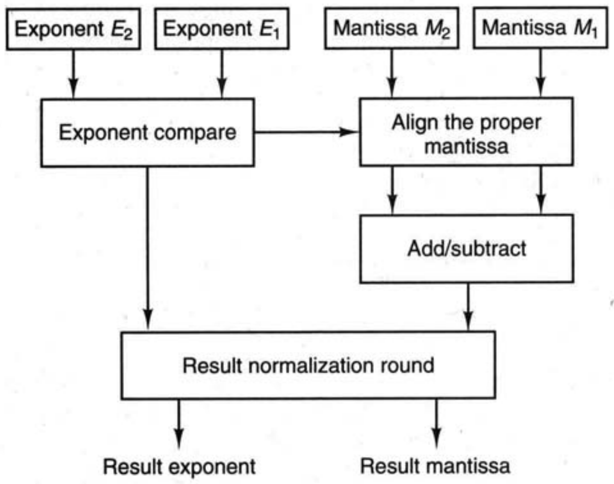


Figure 2.35 Block diagram of a floating-point adder.

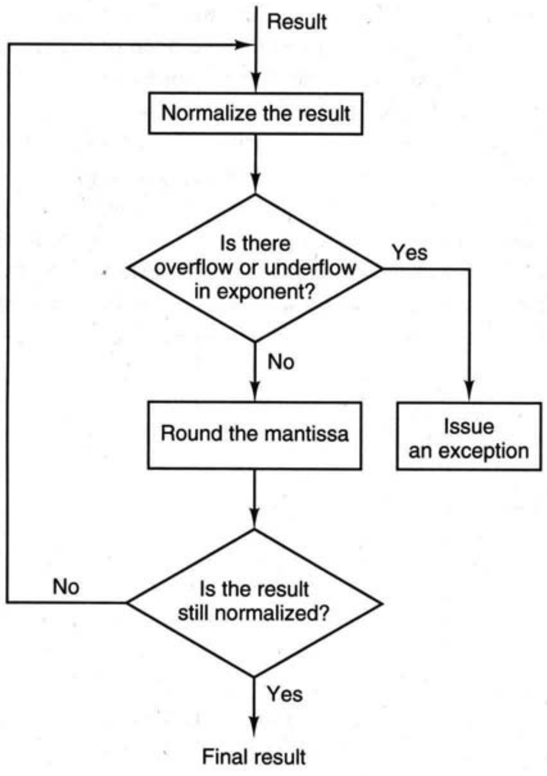


Figure 2.36 The function of the normalization-round unit in a floating-point adder.

Floating-point multiplication. Considering a pair of floating-point numbers represented by $X=m_x*2^{ex}$ and $Y=m_y*2^{ey}$, the multiplication operation can be defined as:

$$X*Y=(m_x*m_y)*2^{ex+ey}$$

For example, when $X=1.000*2^{-2}$ and $Y=-1.010*2^{-1}$, the product $X*Y$ can be computed as follows:

Add the exponents:

$$-2 + (-1) = -3$$

Multiply the mantissas:

```
              1.000
           *  -1.010
              _____
              0000
             1000
            0000
           1000
         __________
          -1.010000
```

Thus the product is $-1.0100 * 2^{-3}$. In general, an algorithm for floating-point multiplication consists of three major steps: (1) computing the exponent of the product by adding the exponents together, (2) multiplying the mantissas, and (3) normalizing and rounding the final product.

Figure 2.37 represents a block diagram for floating-point multiplication. This design consists of three units: add, multiply, and result normalization-round units. The add and multiply units can be designed similarly to the ones that are used for binary integer numbers (different methods were discussed in previous sections). The result normalization-round unit can be implemented in a similar way as the normalization-round unit in a floating-point adder, as shown in Figure 2.36.

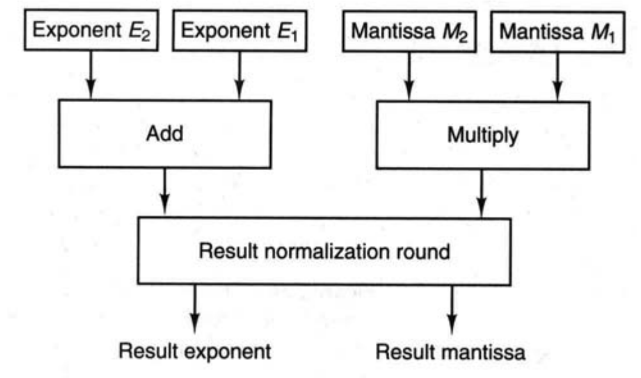


Figure 2.37 Block diagram of a floating-point multiplier.

2.6 MEMORY SYSTEM DESIGN

This section discusses the general principles and terms associated with a memory system. It also presents a review of the design of memory systems, including the use of memory hierarchy, the design of associative memories, and caches.

2.6.1 Memory Hierarchy

With increasing CPU speeds, the bottleneck in the speedup of any computer system can be associated to the memory *access time*. (The time taken to gain access to a data item assumed to exist in memory is known as the access time.) To avoid wasting precious CPU cycles in access time delay, faster memories have been developed. But the cost of such memories prohibits their exclusive use in a computer system. This leads to the need of having a memory hierarchy that supports a combination of faster (expensive) as well as slower (relatively inexpensive) memories. The fastest memories are usually smaller in capacity than slower memories. At present, the general trend of large as well as small personal computers has been toward increasing the use of memory hierarchies. The major reason for this increase is due to the way that programs operate. By statistical analysis of typical programs, it has been established that at any given interval of time the references to memory tend to be confined within local areas of memory [MAN 86]. This phenomenon is known as the property of *locality of reference*. Three concepts are associated with locality of reference: *temporal, spatial,* and *sequential*. Each of these concepts is defined next.

Temporal locality. Items (data or instructions) recently referenced have a good chance to be referenced in the near future. For example, a set of instructions in an iterative loop or in a subroutine may be referenced repeatedly many times.

Spatial locality. A program often references items whose addresses are close to each other in the address space. For example, references to the elements of an array always occur within a certain bounded area in the address space.

Sequential locality. Most of the instructions in a program are executed in a sequential order. The instructions that might cause out-of-order execution are branches. However, branches construct about 20% to 30% of all instructions; therefore, about 70% to 80% of instructions are often accessed in the same order as they are stored in memory.

Usually, the various memory units in a typical memory system can be viewed as forming a hierarchy of memories (M_1, M_2, ..., M_n), as shown in Figure 2.38. As can be seen, M_1 (the highest level) is the smallest, fastest, most expensive memory unit, and is located the closest to the processor. M_2 (the second highest level) is slightly larger, slower, and less expensive and is not as close to the processor as is M_1. The same is true for M_3 to M_n. In general, as the level increases, the speed and thus the cost per byte increases proportionally, which tends to decrease memory capacity. Because of the property of locality of reference, generally data transfers take place in fixed-sized segments of words called *blocks* (sometimes called *pages* or *lines*). These transfers are between adjacent levels and are entirely controlled by the activity in the first level of the memory.

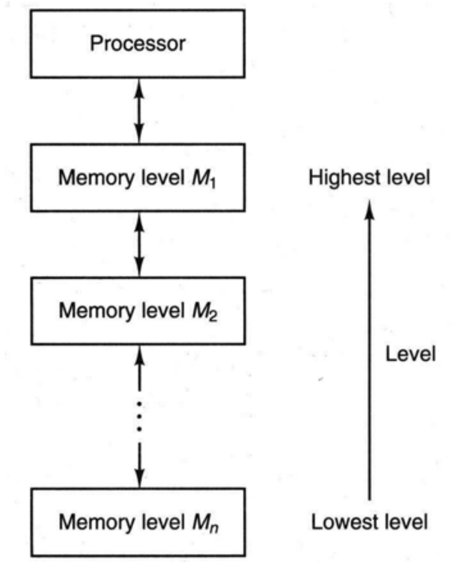


Figure 2.38 Memory hierarchy.

In general, whenever there is a block in level M_i, there is a copy of it in each of the lower levels M_{i+1}, ..., M_n [CHO 74]. Also, whenever a block is not found in M_1, a request for it is sent to successively lower levels until it is located in, say, level M_i. Since each level has limited storage capability, whenever M_i is full and a new block is to be brought in from M_{i+1}, a currently stored block in M_i is replaced using a predetermined replacement policy. These policies vary and are governed by the available hardware and the operating system. (Some replacement policies are explained later in this chapter.)

The main advantage of having a hierarchical memory system is that the information is retrieved most of the time from the fastest level M_1. Hence the average memory access time is nearly equal to the speed of the highest level, whereas the average unit cost of the memory system approaches the cost of the lowest level.

The highest two or three levels of common memory hierarchies are shown in Figure 2.39. Figure 2.39a presents an architecture in which main memory is used as the first level of memory hierarchy, and secondary memory (such as a disk) is used as the second level. Figures 2.39b and c present two alternative architectures for designing a system with three levels of memories. Here a relatively large and slower main memory is coupled together with a smaller, faster memory called *cache*. The cache memory acts as a go-between for the main memory and the processor. (The organization of a cache memory is explained later in this chapter.) Figure 2.40 shows a sample curve of the cost function versus access time for different memory units.

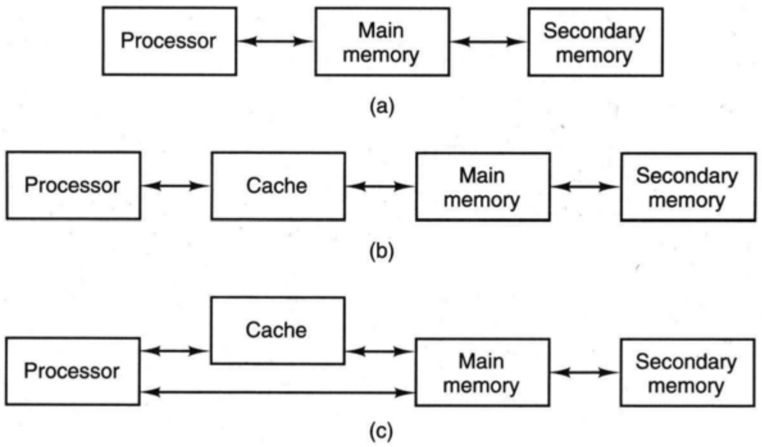


Figure 2.39 (a) Two level memory hierarchy; (b) and (c) Three level memory hierarchy.

With the advancement of technology, the cost of semiconductor memories has decreased considerably, making it possible to have large amounts of semiconductor memories installed in computer systems as main memory. As a result, most of the required data can be brought into the semiconductor memories in advance and can satisfy a major part of the memory references. Thus the impact of the speed of mass storage devices like hard disks and tapes on the overall computer system speed is lessening.

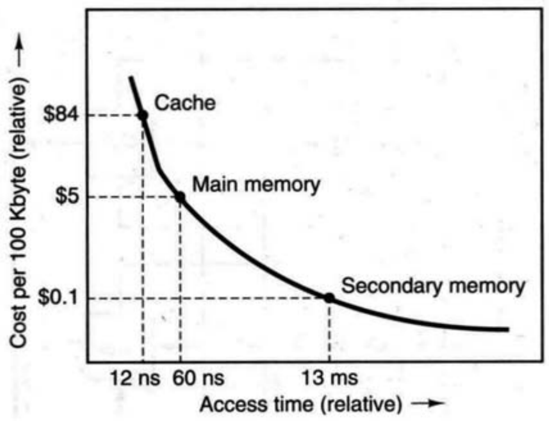


Figure 2.40 Cost function versus access time for various memory devices.

2.6.2 Memory Cell and Memory Unit

A memory cell is the building block of a memory unit. The internal construction of a random-access memory of *m* words with *n* bits per word consists of $m * n$ binary storage cells. A block diagram of a binary cell that stores 1 bit of information is shown in Figure 2.41. The select line determines whether the cell is selected (enabled). The cell is selected whenever the select line is 1. The R/W (read/write) line determines whether a read or a write operation must be performed on a selected cell. When the R/W line is 1, a write operation is performed, which causes the data on the input data line to be stored in the cell. In a similar manner, when the R/W line is 0, a read operation is performed, which causes the stored data in the cell to be sent out on the output data line.

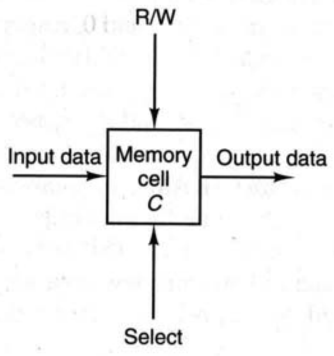


Figure 2.41 Block diagram of a binary memory cell.

A memory cell may be constructed using a few transistors (one to six transistors). The main constraint on the design of a binary cell is its size. The objective is to make it as small as possible so that more cells can be packed into the semiconductor area available on a chip.

A memory unit is an array of memory cells. In a memory unit, each cell may be individually addressed or a group of cells may be addressed simultaneously. Usually, a computer system has a fixed word size. If a word has *n* bits, then *n* memory cells will have to be addressed simultaneously, which enables the cells to have a common select line. A memory unit having four words with a word size of 2 is shown in Figure 2.42. Any particular word can be selected by means of the address lines A_1 and A_0. The operation to be performed on the selected word is determined by the R/W line. For example, if the address lines A_1 and A_0 are set to 1 and 0, respectively, word 2 is selected. If the R/W line is set to 1, then the data on the input data lines X_1 and X_0 are stored in cells $C_{2,1}$ and $C_{2,0}$ respectively. Similarly, if the R/W line is set to 0, the data in cells $C_{2,1}$ and $C_{2,0}$ are sent out on the output data lines Z_1 and Z_0, respectively.

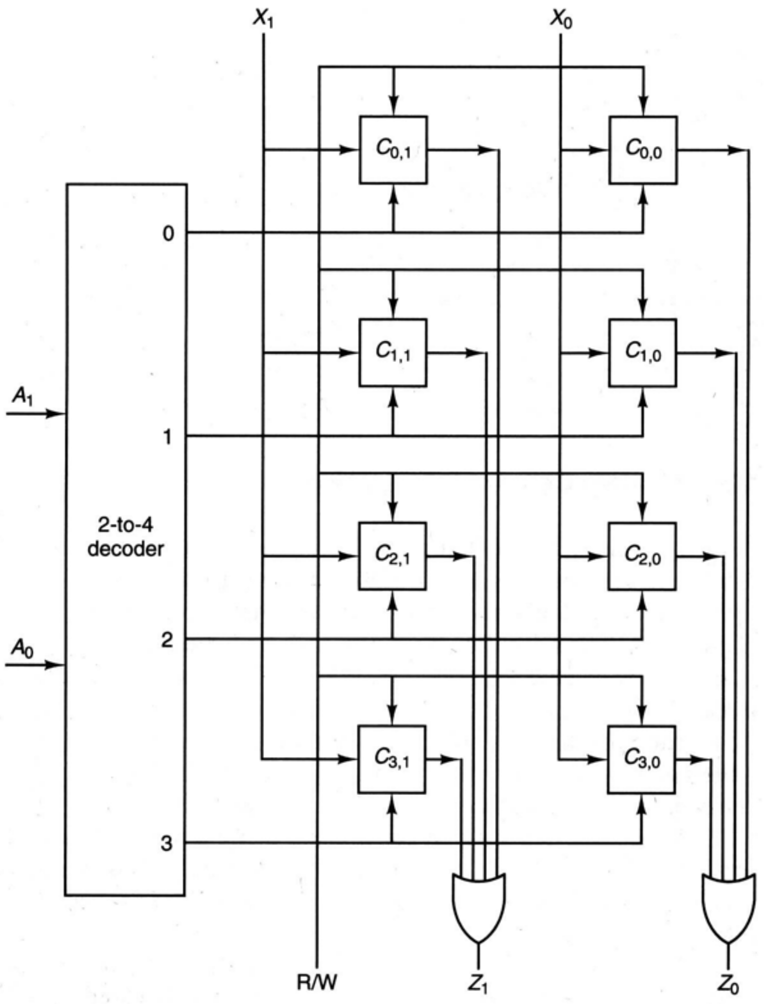


Figure 2.42 Block diagram of a 4-by-2 memory unit.

A memory unit, in which any particular word can be accessed independently, is known as a *random-access memory* (RAM). In a random-access memory the time required for accessing a word is the same for all words.

There are two types of memories, *static* (SRAM) and *dynamic* (DRAM). Static memories hold the stored data either until new data are stored in them or until the power supply is discontinued. Static memories retain the data when a word is read from it. Hence this type of memory is said to have a *nondestructive-read property*. In contrast, the data contained in dynamic memories need to be written back into the corresponding memory location after every read operation. Thus dynamic memories are characterized by a *destructive-read property*. Furthermore, dynamic memories need to be periodically refreshed. (In a DRAM, often each bit of the data is stored as a charge in a leaky capacitor, and this requires the capacitor to be recharged within certain time intervals. When the capacitors are recharged, they are said to have been *refreshed*.) The implementation of the refreshing circuit may appear as a disadvantage for DRAM design; but, in general, a cell of a static memory requires more transistors than a cell of a dynamic memory. So the refreshing circuit is considered an acceptable and unavoidable cost of DRAM.

2.6.3 Interleaved Memory

To be able to overlap read or write accesses of several data, multiple memory units can be connected to the CPU. Figure 2.43 represents an architecture for connecting 2^m memory units (memory modules, or banks) in parallel. In this design, the memory address from the CPU, which consists of *n* bits, is partitioned into two sections, S_m and S_{n-m}. The section S_m is the least significant *m* bits of the memory address and selects one of the memory units. The section S_{n-m} is the most significant *n-m* bits of the memory address and addresses a particular word in a memory unit. Thus a sequence of consecutive addresses is assigned to consecutive memory units. In other words, address *i* points to a word in the memory unit M_j, where $j=i$ (modulo 2^m). For example, addresses 0, 1, 2, . . ., 2^m-1 and 2^m are assigned to memory units M_0, M_1, M_2, . . ., M_{2m-1}, and M_0, respectively. This technique of distributing addresses among memory units is called *interleaving*. The interleaving of addresses among *m* memory units is called *m-way interleaving*. The accesses are said to overlap because the memory units can be accessed simultaneously.

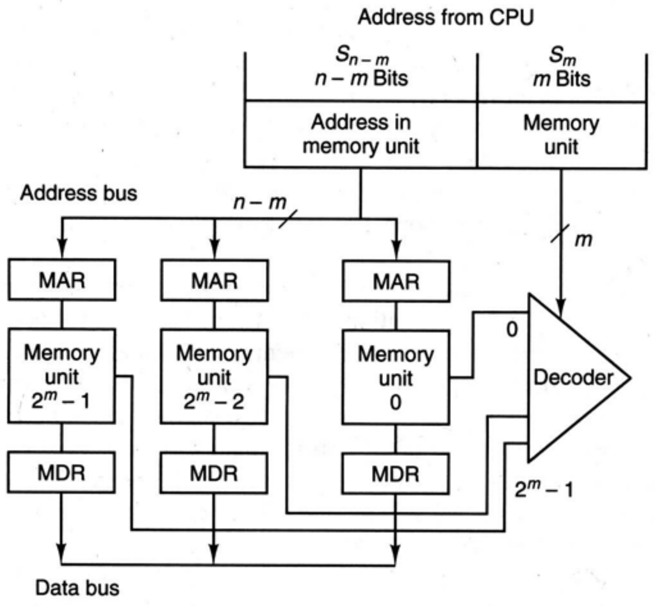


Figure 2.43 Block diagram of an interleaved memory.

2.6.4 Associative Memory

Unlike RAMs, in which the stored data are identified by means of a unique address assigned to each data item, the data stored in an associative memory are identified, or accessed, by the content of the data themselves. Because of the nature of data access in associative memories, such memories are also known as *content addressable memories* (CAM).

In general, a search time for a particular piece of data in a RAM having *n* words will take $t_*f(n)$, where *t* is the time taken to fetch and compare one word of the memory and *f* is an increasing function on *n*. [*f* is an increasing function on *n* if and only if $f(n_1)<f(n_2)$ for all $n_1<n_2$.] Hence, with an increase in *n*, the search time increases too. But in the case of a CAM having *n* words, the search time is almost independent of *n* because all the words may be searched in parallel. Only one cycle time is required to determine if the desired word is in memory, and, if present, one more cycle time is required to retrieve it.

To be able to do a parallel search by data association, each word needs to have a dedicated circuit for itself. The additional hardware associated with each word adds significantly to the cost of associative memories, and, as a result, the hardware is used only in applications for which the search time is vital to the proper implementation of the task under execution. For example, associative memories may be used, as the highest level of memory, in real-time systems or in military applications requiring a large number of memory accesses, as in the case of pattern recognition.

CAMs can be categorized into two basic types of functions. The first type describes CAMs in terms of an exact match (that is, they address data based on equality with certain key data). This type is simply referred to as an *exact match CAM*. The second type, called a *comparison CAM*, is an enhancement of the exact match. Rather than basing a search on equality alone, the search is based on a general comparison (that is, the CAM supports various relational operators, such as greater than and less than).

All associative memories are organized by words, but they may differ depending on whether their logical organizations are fixed or variable [HAN 66]. In fixed organization, a word is divided into fixed segments in which any segment may be used for interrogation. Depending on the application, a segment of a word may further be defined as the *key segment*, and each bit of this segment *must* be used for the purpose of interrogation. In variable organization, the word may be divided into *fixed segments*, but any part of a segment may be used for the purpose of interrogation. In its most general form, the CAM will be fully interrogatable (that is, it will allow any combination of bits in the word to be searched for).

In this section, we shall focus on fully interrogatable CAMs with exact match capacity. This will allow us to illustrate topics such as cache memory (in the following section) in which associative memories play a vital role more concretely.

Figure 2.44 shows a block diagram of an associative memory having *m* words and *n* bits per word. The argument register *A* and the key register *K* each have *n* bits, one for each bit of a word. Before the search process is started, the word to be searched is loaded into the argument register A. The segment of interest to the word is specified by the bit positions having 1's in the key register *K*. Once the argument and key registers are set, each word in memory is compared in parallel with the content of *A*. If a word matches the argument, the corresponding bit in the match register *M* is set. Thus, *M* has *m* bits, one for each word in memory. If more than one bit is set to 1 in *M*, the select circuit determines which word is to be read. For example, all the matching entries may be read out in some predetermined order [HAY 78].

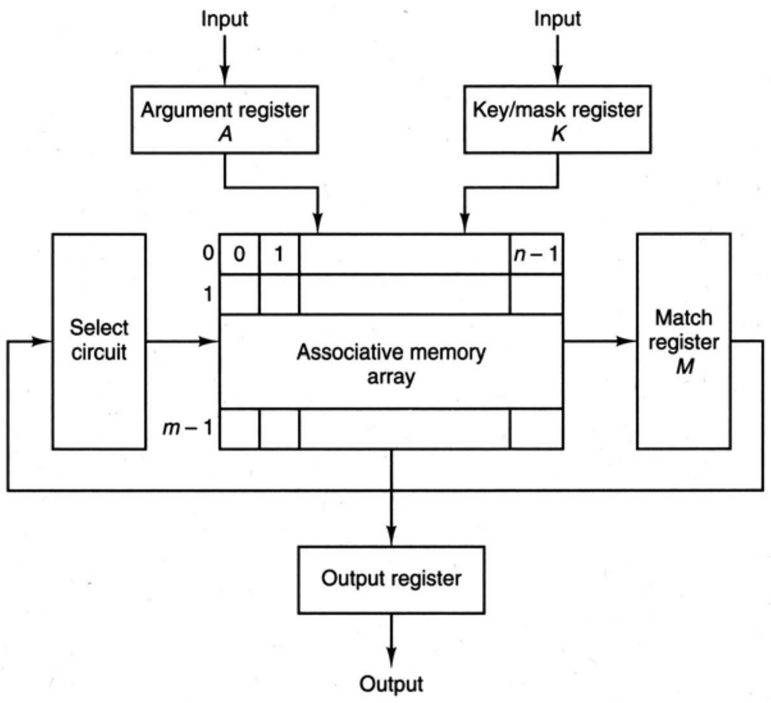


Figure 2.44 Block diagram of an associative memory.

The following example shows the bit settings in *M* for three words if the contents of *A* and *K* are as shown (assuming a word size of 8 bits).

A	01 101010	
K	11 000000	
Word 1	00 101010	Match bit = 0
Word 2	01 010001	Match bit = 1
Word 3	01 000000	Match bit = 1

Because only the two leftmost bits of *K* are set, words 2 and 3 are declared as a match with the given argument, whereas word 1 is not a match, although it is closer to the contents of *A*. A subsequent sequential search of words 2 and 3 will be required to access the required word.

An associative memory is made of an array of similar cells. A block diagram of one of these cells is given in Figure 2.45. It has four inputs: the argument bit *a*, the corresponding key bit *k*, the R/W line to specify the operation to be performed, and the select line *S* to select the particular cell for read or write. The cell has two outputs: the match bit *m*, which shows if the data stored in the cell match with the argument bit *a*, and the data output *q*. The match bit is set to 1 if the key bit *k* is 0.

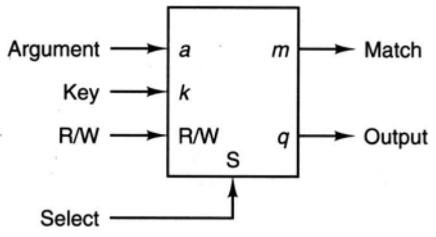


Figure 2.45 Block diagram of an associative memory cell.

Figure 2.46 shows a 4-by-2 associative memory array. When the read operation is selected, the argument is matched with the words whose select lines are set to 1. The match output of each cell is then ANDed together to generate the match bit for that particular word. When the write operation is selected, the argument bits are stored in the selected word.

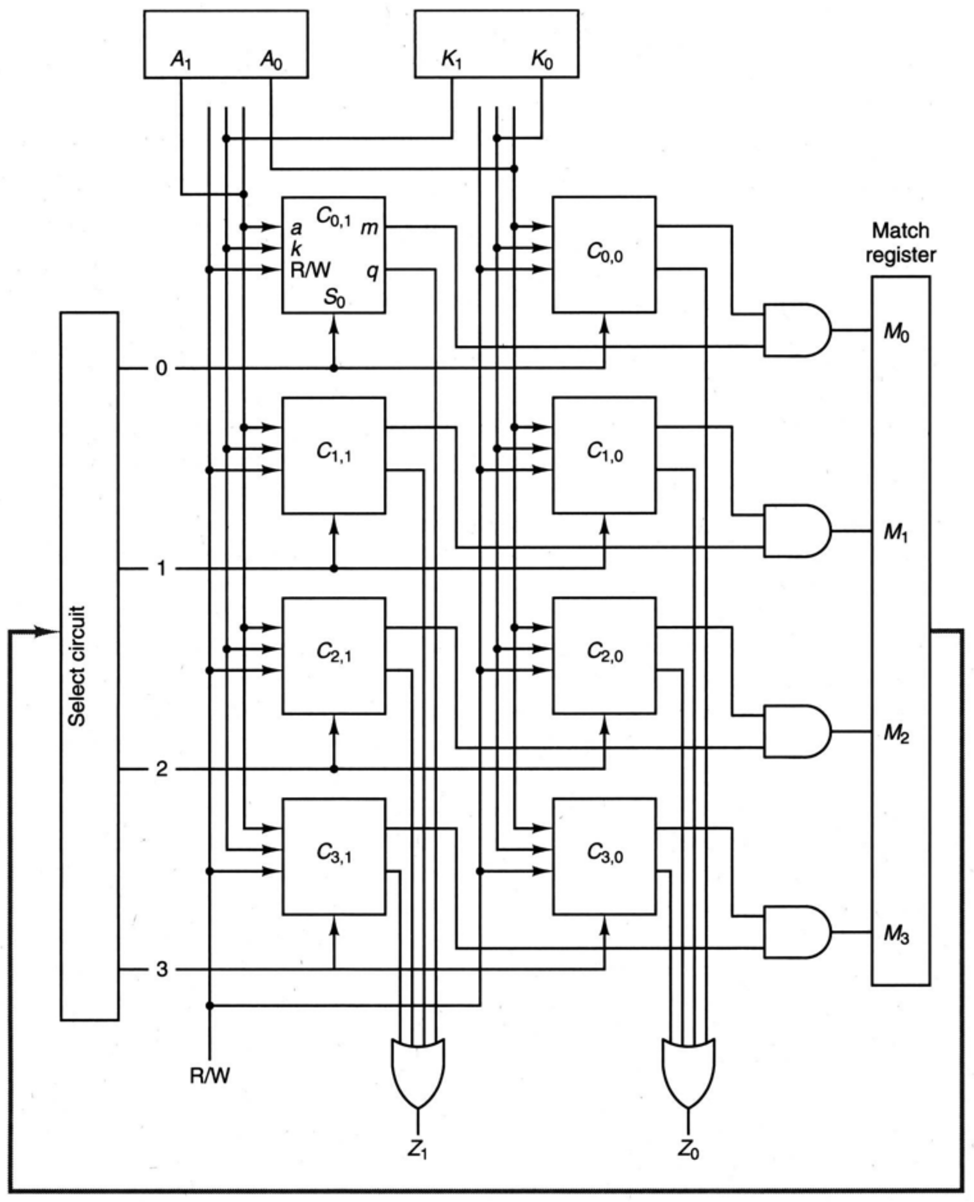


Figure 2.46 Block diagram of a 4-by-2 associative memory array.

The Boolean expression for each bit M_i of the match register can be derived as follows: Let W_{ij} denote the jth bit of the ith word in the memory, A_j and K_j denote the jth bit of A and K respectively, and M_i denote the ith bit of M.

Suppose that x_j is defined as $x_j = A_j W_{ij} + \overline{A}_j \overline{W}_{ij}$ for j=0, 1, 2, …, n-1. That is, x_j is equal to 1 if w_{ij} matches A_j; otherwise x_j is 0. Then word i matches the argument if M_i is equal to 1, where

$$\begin{aligned} M_i &= (x_0 + \overline{K}_0)(x_1 + \overline{K}_1)(x_2 + \overline{K}_2)\cdots(x_{n-1} + \overline{K}_{n-1}) \\ &= \prod_{j=0}^{n-1} (x_j + \overline{K}_j) \\ &= \prod_{j=0}^{n-1} (A_j W_{ij} \overline{A}_j \overline{W}_{ij} + \overline{K}_j) \end{aligned}$$

Compared to RAMs, associative memories are very expensive (require more transistors), but have much faster response time in searching. (Response time refers to the time interval between the start and finish time of the search.)

2.6.5 Cache Memory

One problem designers face in constructing a processor is the bottleneck associated with memory speeds. Because fetches from main memory require considerably more time when compared to the overall speeds in the processor, designers spend a lot of time and effort making memory speeds as fast as possible.

One way to make the memory appear faster is to reduce the number of times main memory has to be accessed. If a small amount of fast memory is installed and at any point in time part of a program is loaded in this fast memory, then, due to the property of locality of reference, the number of references to the main memory will be drastically reduced. Such a fast memory unit, used temporarily to store a portion of the data and instructions (from the main memory) for immediate use, is known as *cache memory*.

Because cache memory is expensive, a computer system can have only a limited amount of it installed. Therefore, in a computer system there is a relatively large and slower main memory coupled together with a smaller, faster cache memory. The cache acts as a "go-between" for the main memory and the CPU. The cache contains copies of some blocks of the main memory. Therefore, when the CPU requests a word (if the word is in the fast cache), there will be no need to go to the larger, slower main memory.

Although the cache size is only a small fraction of the main memory, a large part of the memory reference requests will be fulfilled by the cache due to the nonrandomness of consecutive memory reference addresses. As stated previously, if the average memory access time per datum approaches the access time for the cache, while the average cost per bit approaches that of the main memory, the goals of the memory hierarchy design are realized.

The performance of a system can be greatly improved if the cache is placed on the same chip as the processor. In this case, the outputs of the cache can be connected to the ALU and registers through short wires, significantly reducing access time. For example, the Intel 80486 microprocessor has an on-chip cache [CRW 90]. Although the clock speed for the 80486 is not much faster than that for the 80386, the overall system speed is much faster.

Cache operation. When the CPU generates an address for memory reference, the generated address is first sent to the cache. Based on the contents of the cache, a *hit* or a *miss* occurs. A hit occurs when the requested word is already present in the cache. In contrast, a miss happens when the requested word is not in the cache.

Two types of operations can be requested by the CPU: a read request and a write request. When the CPU generates a read request for a word in memory, the generated request is first sent to the cache to check if the word currently resides in the cache. If the word is not found in the cache (i.e., a read miss), the requested word is supplied by the main memory. A copy of the word is stored in the cache for future reference by the CPU. If the cache is full, a predetermined replacement policy is used to swap out a word from the cache in order to accommodate the new word. (A detailed explanation of cache replacement policies is given later in this chapter.) If the requested word is found in the cache (i.e., a read hit), the word is supplied by the cache. Thus no fetch from main memory is required. This speeds up the system considerably.

When the CPU generates a write request for a word in memory, the generated request is first sent to the cache to check if the word currently resides in the cache. If the word is not found in the cache (i.e., a write miss), a copy of the word is brought from the memory into the cache. Next, a write operation is performed. Also, a write operation is performed when the word is found in the cache (i.e., a write hit). To perform a write operation, there are two main approaches that the hardware may employ: *write through*, and *write back*. In the write-through method, the word is modified in both the cache and the main memory. The advantage of the write-through method is that the main memory always has consistent data with the cache. However, it has the disadvantage of slowing down the CPU because all write operations require subsequent accesses to the main memory, which are time consuming.

In the write-back method, every word in the cache has a bit associated with it, called a *dirty bit* (also called an *inconsistent bit*), which tells if it has been changed while in the cache. In this case, the word in the cache may be modified during the write operation, and the dirty bit is set. All changes to a word are performed in the cache. When it is time for a word to be swapped out of the cache, it checks to see if the word's dirty bit is set: if it is, it is written back to the main memory in its updated form.

The advantage of the write-back method is that as long as a word stays in the cache it may be modified several times and, for the CPU, it does not matter if the word in the main memory has not been updated. The disadvantage of the write-back method is that, although only one extra bit has to be associated with each word, it makes the design of the system slightly more complex.

Basic cache organization. The basic motivation behind using cache memories in computer systems is their speed. Most of the time, the presence of the cache is not apparent to the user. Since it is desirable that very little time be wasted when searching for words in a cache, usually the cache is managed through hardware-based algorithms. The translation of the memory address, specified by the CPU, into the possible location of the corresponding word in the cache is referred to as a *mapping* process.

Based on the mapping process used, cache organization can be classified into three types:

1. Associative-mapping cache
2. Direct-mapping cache
3. Set-associative mapping cache

The following sections explain these cache organizations. To illustrate these three different cache organizations, the memory organization shown in Figure 2.39c is used. In this figure, the CPU communicates with the cache as well as the main memory. The main memory stores 64K words (16-bit address) of 16 bits each. The cache is capable of storing 256 of these words at any given time. Also, in the following discussion it is assumed that the CPU generates a read request and not a write request. (The write request would be handled in a similar way.)

Associative Mapping. In an associative-mapping cache (also referred to as fully associative cache), both the address and the contents are stored as one word in the cache. As a result, a memory word is allowed to be stored at any location in the cache, making it the most flexible cache organization. Figure 2.47 shows the organization of an associative-mapping cache for a system with 16-bit addressing and 16-bit data. Note that the words are stored at arbitrary locations regardless of their absolute addresses in the main memory.

The organization of an associative-mapping cache can be viewed as a combination of an associative memory and a RAM, as shown in Figure 2.47 (all numbers are in hexadecimal). Since each associative memory cell is many times more expensive than a RAM cell, only the addresses of the words are stored in the associative part, while the data can be stored in the RAM part of the cache because only the address is used for associative search. This will not increase the access time of the cache significantly, but will result in a significant drop in cost. In this organization, when the CPU generates an address for memory reference, it is passed into the argument register and is compared, in parallel, with the address fields of all words currently stored in the cache for a matching address. Once the location has been determined, the corresponding data can be accessed from the RAM.

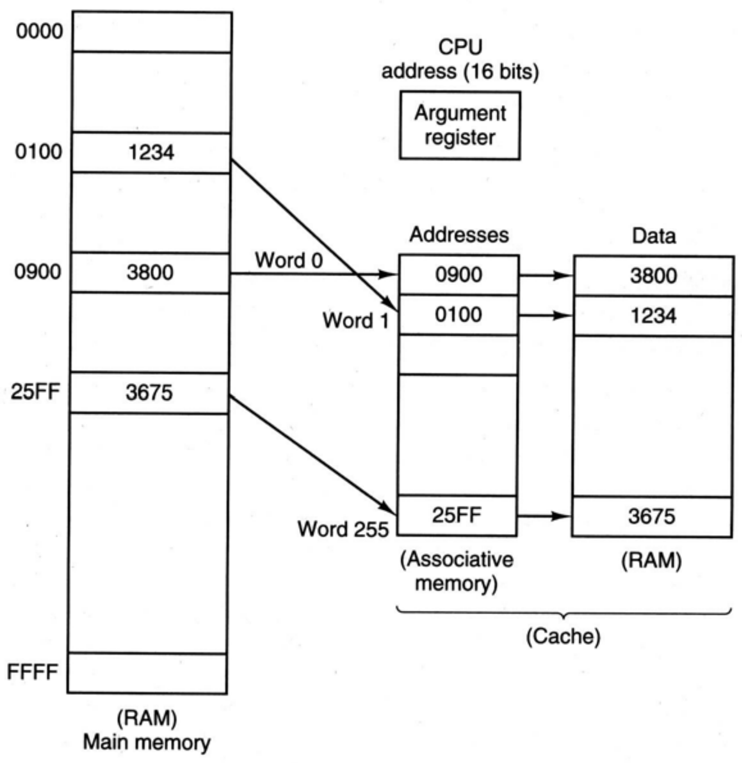


Figure 2.47 Associative-mapping cache (all numbers are in hexadecimal).

The major disadvantage of this method is its need for a large associative memory, which is very expensive and increases the access time of the cache.

Direct Mapping. Associative-mapping caches require associative memory for some part of their organization. Since associative memories are very expensive, an alternative cache organization, known as a *direct-mapping cache*, may be used. In this organization, RAM memories are used as the storage mechanism for the cache. This reduces the cost of the cache, but imposes limitations on its use. In a direct-mapping cache, the requested memory address is divided into two parts, an *index* field, which refers to the lower part of the address, and a *tag* field, which refers to the upper part. The index is used as an address to a location in the cache where the data are located. At this index, a tag and a data value are stored in the cache. If the tag of the requested memory address matches the tag of cache, the data value is sent to the CPU. Otherwise, the main memory is accessed, and the corresponding data value is fetched and sent to the CPU. The data value, along with the tag part of its address, also replaces any word currently occupying the corresponding index location in the cache.

Figure 2.48 represents an architecture for the direct-mapping cache. The design consists of three main components: data memory, tag memory, and match circuit. The data memory holds the cached data. The tag memory holds the tag associated with each cached datum and has an entry for each word of the data memory. The match circuit sets the match line to 1, indicating that the referenced word is in the cache.

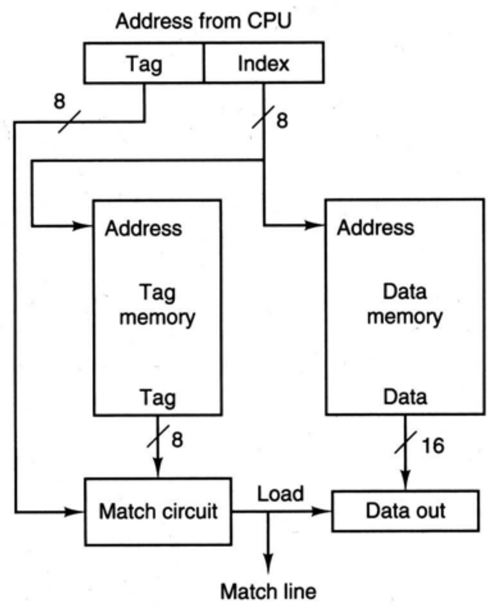


Figure 2.48 Architecture of a direct-mapping cache.

An example illustrating the direct-mapping cache operation is shown in Figure 2.49 (all numbers are in hexadecimal). In this example, the memory address consists of 16 bits and the cache has 256 words. The eight least significant bits of the address constitute the index field, and the remaining eight bits constitute the tag field. The 8 index bits determine the address of a word in the tag and data memories. Each word in the tag memory has 8 bits, and each word in the data memory has 16 bits. Initially, the content of address 0900 is stored in the cache. Now, if the CPU wants to read the contents of address 0100, the index (00) matches, but the tag (01) is now different. So the content of main memory is accessed, and the data word 1234 is transferred to the CPU. The tag memory and the data memory words at index address 00 are then replaced with 01 and 1234, respectively.

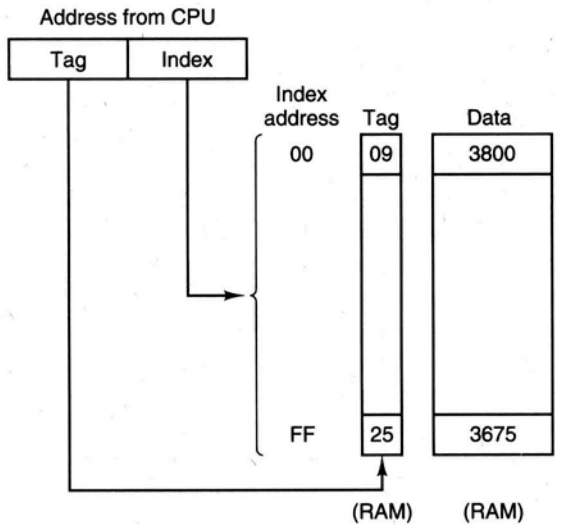


Figure 2.49 Direct-mapping cache (all numbers in hexadecimal).

The advantage of direct mapping over associative mapping is that it requires less overhead in terms of the number of bits per word in the cache. The major disadvantage is that the performance can drop considerably if two or more words having the same index but different tags are accessed frequently. For example, memory addresses 0100 and 0200 both have to be put in the cache at position 00, so a great deal of time is spent swapping them back and forth. This slows the system down, thus defeating the purpose of the cache in the first place. However, considering the property of locality of reference, the probability of having two words with the same index is low. Such words are located 2^k bits apart in the main memory, where k denotes the number of bits in the index field. In our example, such a situation will only occur if the CPU requests reference to words that are 256 (2^8) words apart. To further reduce the effects of such situations often an expanded version of the direct-mapping cache, called a *set-associative cache*, is used. The following section describes the basic structure of a set-associative mapped cache.

Set-Associative Mapping. The set-associative mapping cache organization (also referred to as set-associative cache) is an extension of the direct-mapping cache. It solves the problem of direct mapping by providing storage for more than one data value with the same index. For example, a set-associative cache with *m* memory blocks, called *m*-way set associative, can store *m* data values having the same index, along with their tags. Figure 2.50 represents an architecture for a set-associative mapping cache with *m* memory blocks. Each memory block has the same structure as a direct-mapping cache. To determine that a referenced word is in the cache, its tag is compared with the tag of cached data in all memory blocks in parallel. A match in any of the memory blocks will enable (set to 1) the signal match line to indicate that the data are in the cache. If a match occurs, the corresponding data value is passed on to the CPU. Otherwise, the data value is brought in from the main memory and sent to the CPU. The data value, along with its tag, is then stored in one of the memory blocks.

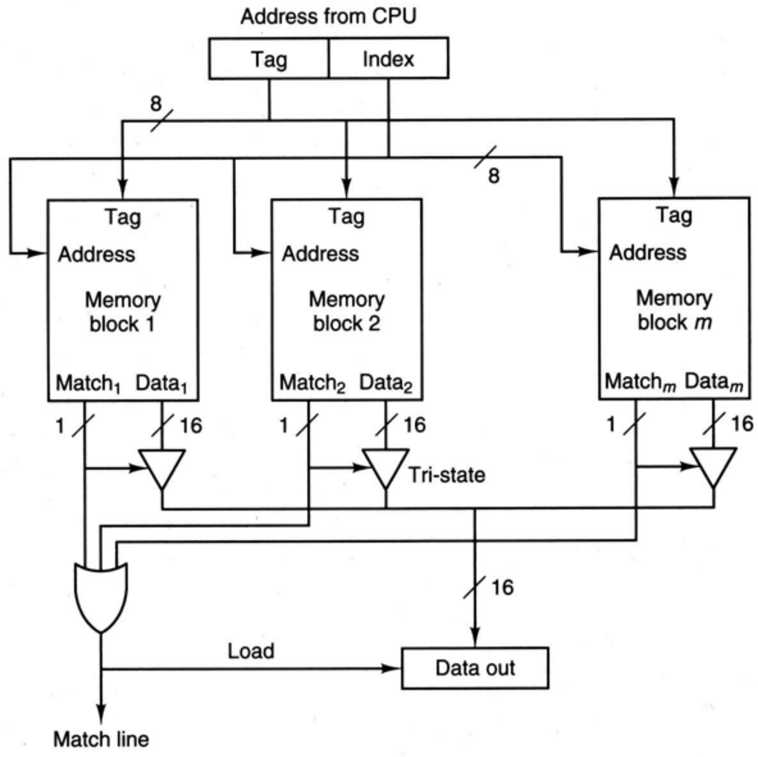


Figure 2.50 Architecture of an *m*-way set-associative mapping cache.

An example illustrating the set-associative mapping cache operation is shown in Figure 2.51 (all numbers are in hexadecimal). This figure represents a two-way set-associative mapping cache. The content of address 0900 is stored in the cache under index 00 and tag 09. If the CPU wants to access address 0100, the index (00) matches, but the tag is now different. Therefore, the content of main memory is accessed, and the data value 1234 is transferred to the CPU. This data with its tag (01) is stored in the second memory block of the cache. When there is no space for a particular index in the cache, one of the two data values stored under that index will be replaced according to some predetermined replacement policy (discussed next).

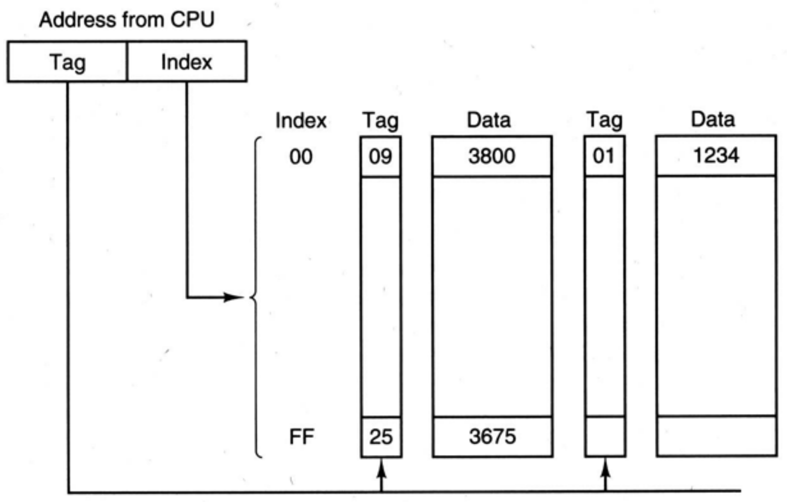


Figure 2.51 Two-way set-associative mapping cache (all numbers are in hexadecimal).

Replacement strategies. Sooner or later, cache will become full. When this occurs, a mechanism should be there to replace a word with the newly accessed data from memory. In general, there are three main strategies for determining which word should be swapped out from the cache; they are called *random, least frequently used*, and *least recently used* replacement policies.   Each is explained next.

Random Replacement. This method picks a word at random and replaces that word with the newly accessed data. This method is easy to implement in hardware, and it is faster than most other algorithms. The disadvantage is that the words most likely to be used again have as much of a chance of being swapped out as a word that is likely not to be used again. This disadvantage diminishes as the cache size increases.

Least Frequently Used. This method replaces the data that are used the least. It assumes that data that are not referenced frequently are not needed as much. For each word, a counter is kept for the total number of times the word has been used since it was brought into the cache. The word with the lowest count is the word to be swapped out. The advantage of this method is that a frequently used word is more likely to remain in cache than a word that has not been used often. One disadvantage is that words that have recently been brought into the cache have a low count total, despite the fact that they are likely to be used again. Another disadvantage is that this method is more difficult to implement in terms of hardware and is thus more expensive.

Least Recently Used. This method has the best performance per cost compared with the other techniques and is often implemented in real-world systems. The idea behind this replacement method is that a word that has not been used for a long period of time has a lesser chance of being needed in the near future according to the property of temporal locality. Thus, this method retains words in the cache that are more likely to be used again. To do this, a mechanism is used to keep track of which words have been accessed most recently. The word that will be swapped out is the word that has not been used for the longest period of time. One way to implement such a mechanism is to assign a counter to each word in the cache. Each time the cache is accessed, each word's counter is incremented, and the word's counter that was accessed is reset to zero. In this manner, the word with the highest count is the one that was least recently used.

Example: i486 Microprocessor Cache Structure

To increase overall performance, the Intel i486 microprocessor contains an 8-Kbyte on-chip cache [INT 91]. The write-through strategy is used for writing into this cache. As shown in Figure 2.52, the cache has four-way set-associative organization. Each memory block contains a data memory with 128 lines; each line is 16 bytes. Also, each memory block contains a 128 X 21 memory to keep the tags. Note that it is not necessary to store the 4 least significant bits of memory addresses because each time 16 bytes are fetched into the cache. Considering Figure 2.52, the structure of the cache memory can also be expressed by saying that the 8 Kbytes of cache are logically organized as 128 sets, each containing four lines.

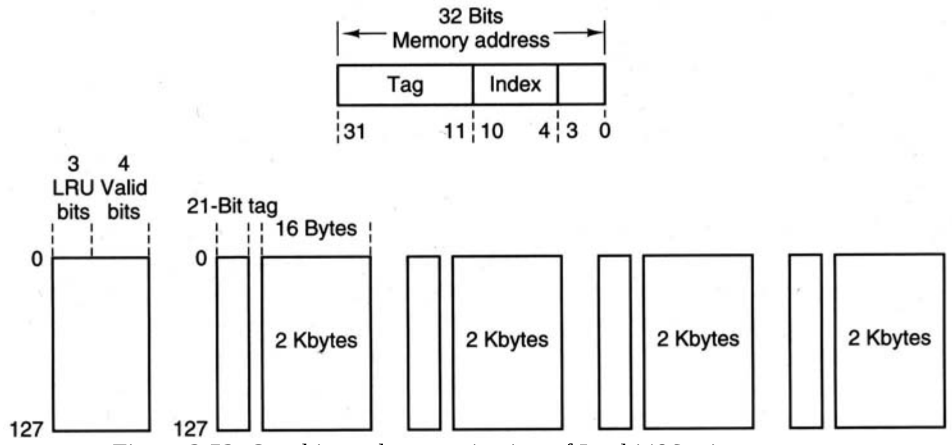


Figure 2.52 On-chip cache organization of Intel i486 microprocessor.

A valid bit is assigned to each line in the cache. Each line is either valid or non-valid. When a system is powered up, the cache is supposed to be empty. But, in practice, it has some random data that are invalid. To discard such data, when the system is powered up, all the valid bits are set to 0. The valid bit associated with each line in the cache is set to 1 the first time the line is loaded from main memory. As long as the valid bit is 0, the corresponding lines are excluded from any search performed on the cache.

When a new line needs to be placed in the cache, a pseudo least recently used mechanism (implemented in hardware) is used to determine which line should be replaced. If there is a nonvalid line among the four possible lines, that line will be replaced. Otherwise, when all four lines are valid, a least recently used line is selected for replacement based on the value of 3 bits, r_0, r_1, and r_2, which are defined for each set of four lines in the cache. In Figure 2.52, these bits are denoted as LRU bits. The LRU bits are updated for every hit or replaced in their corresponding four lines. Let these four lines be labeled l_0 , l_1 , l_2 , and l_3. If the most recent access was to l_0 or l_1, r_0 is set to 1. Otherwise, if the most recent access was to l_2 or l_3, r_0 is set to 0. Among l_0 and l_1, if the most recent access was to l_0, r_1 is set to 1, otherwise r_1 is set to 0. Among l_2 and l_3, if the most recent access was to l_2, r_2 is set to 1; otherwise, r_2 is set to 0.

This updating policy allows us to replace a valid line based on the following rules:

if $r_0 = 0$ and $r_1 = 0$, then replace l_0,
if $r_0 = 0$ and $r_1 = 1$, then replace l_1,
if $r_0 = 1$ and $r_2 = 0$, then replace l_2,
if $r_0 = 1$ and $r_2 = 1$, then replace l_3.

Whenever the cache is flushed all 128 three LRU bits are set to 0.

Cache performance. How well a cache performs is based on the number of times a given piece of data is matched, or found in the cache. In extreme cases, if no matches are found in the cache, then the cache must fetch everything from main memory, and cache performance is very poor. On the other hand, if everything could be held in the cache, it would match every time and never have to access main memory. Cache performance for this case would be extremely high. In reality, caches fall somewhere between these extremes. Nevertheless, as you can see, cache performance is still based on the number of matches or hits found in the cache. This probability of getting a match, called, *the hit ratio*, is denoted by H. During execution of a program, if N_c and N_m are the number of address references satisfied by the cache and the main memory, respectively, then H is the ratio of total addresses satisfied by the cache to the total number of addresses satisfied by both the cache and main memory. That is,

$$H = \frac{N_c}{N_c + N_m}$$

Because data in the cache can be retrieved more quickly than data in main memory, making H as close to 1 as possible is desirable. This definition of H is applicable to any two adjacent levels of a memory hierarchy; but due to the huge amount of main memory available on modern computers, it is often used in the previously described context.

In the preceding equation, if H is the probability that a hit will occur, then 1- H, called the *miss ratio*, is the probability that a hit will not occur. Given that t_c denotes the access time for the cache when a hit occurs, and t_m denotes the access time for the main memory in case of a miss, then the average access time for the cache, t_a, is equal to the probability for a hit times the access time for the cache, plus the probability for a miss times the access time for main memory (because if it is not in the cache, it must be fetched from main memory). Thus

$$t_a = H\, t_c + (1 - H)\, t_m$$

and, by substitution, we get:

$$t_a = \frac{N_c\, t_c}{N_c + N_m} + \left(1 - \frac{N_c}{N_c + N_m}\right) t_m$$
$$= \frac{N_c\, N_c + N_m\, N_m}{N_c + N_m}$$

This makes perfect sense, because this equation simply means the total amount of time spent addressing from the cache plus the total amount of time spent addressing from the main memory divided by the total number of requests.

In general, direct-mapping caches have larger miss ratios than set-associative caches. However, Hill [HIL 88] based on simulations on some data, has shown that the gap diminishes as the size of caches gets larger. For example, an 8-Kbyte two-way set-associative cache with line size of 32 bytes has a miss ratio difference of 0.013 in comparison with a direct-mapping cache of the same size. However, for the caches with the 32 Kbytes the difference is 0.005. The main reason for this phenomenon is that the miss ratio of all kinds of caches decreases as the cache size increases.

In comparing the direct-mapping caches with set-associative caches, it turns out that the direct-mapping caches have smaller average access times for sufficiently large cache sizes [HIL 88]. One reason is that a set-associative cache requires extra gates and hence has a longer hit access time (t_c) than a direct-mapping cache (this can be observed from their basic structure given in Figures 2.48 and 2.50). Another reason is that the gap between the direct-mapping caches miss ratio and set-associative caches miss ratio diminishes as the caches get larger. Therefore, set-associative organization is preferred for small caches, while direct-mapping organization is preferred for large caches.

2.6.6 Virtual Memory

Another technique used to improve system performance is called *virtual memory*. As the name implies, virtual memory is the illusion of a much larger main memory size (logical view) than what actually exists (physical view). Prior to the advent of virtual memory, if a program's address space exceeded the actual available memory, the programmer was responsible for breaking up the program into smaller pieces called *overlays*. Each overlay then could fit in main memory. The basic process was to store all these overlays in secondary memory, such as on a disk, and to load individual overlays into main memory as they were needed.

This process required knowledge of where the overlays were to be stored on disk, knowledge of input/output operations involved with accessing the overlays, and keeping track of the entire overlay process. This was a very complex and tedious process that made the complexity of programming a computer even more difficult.

The concept of virtual memory was created to relieve the programmer of this burden and to let the computer manage this process. Virtual memory allows the user to write programs that grow beyond the bounds of physical memory and still execute properly. It also allows for multiprogramming, by which main memory is shared among many users on a dynamic basis. With multiprogramming, portions of several programs are placed in the main memory at the same time, and the processor switches its time back and forth among these programs. The processor executes one program for a brief period of time (called a *quantum* or *time-slice*) and then switches to another program; this process continues until each program is completed. When virtual memory is used, the addresses used by the programmer are seen by the system as *virtual addresses*, which are so called because they are mapped onto the addresses of physical memory and therefore do not access the same physical memory address from one execution of an instruction to the next.

Virtual addresses, also called *logical addresses*, are generated by the processor during the compile time and are translated into physical addresses at run time. The two main methods for achieving a virtual memory environment are *paging* and *segmentation*. Each is explained next.

Paging. Paging is the technique of breaking a program (referred to in the following as process) into smaller blocks of identical size and storing these blocks in secondary storage in the form of *pages*. By taking advantage of the locality of reference, these pages can then be loaded into main memory, a few at a time, into blocks of the same size called *frames* and executed just as if the entire process were in memory. For this method to work properly, each process must maintain a *page table* in main memory. Figure 2.53 shows how a paging scheme works. The base register, which each process has, points to the beginning of the process's page table. Page tables have an entry for each page that the process contains. These entries usually contain a *load* field of one bit, an *address* field, and an *access* field. The load field specifies whether the page has been brought into main memory. The address field specifies the frame number of the frame into which the page is loaded. The address of the page within main memory is evaluated by multiplying the frame number and the frame size. (Since frame size is usually a power of 2, shifting is often used for multiplying frame number by frame size.) If a page has not been loaded, the address of the page within secondary memory is held in this field. The access field specifies the type of operation that can be performed on a block. It determines whether a block is read only, read/write, or executable.

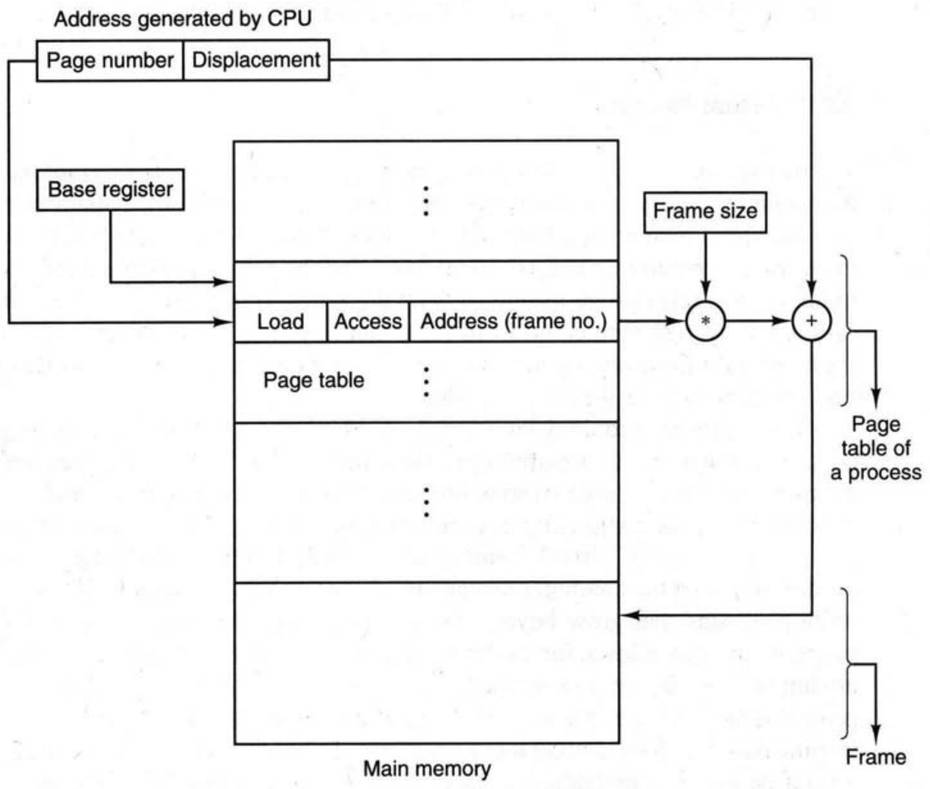


Figure 2.53 Using page table to convert a virtual address to a physical address.

When an access to a variable or an instruction that is not currently loaded into the memory is encountered, a *page fault* occurs, and the page that contains the necessary variable or instruction is brought into the memory. The page is stored in a free frame, if one exists. If a free frame does not exist, one of the process's own frames must be given up, and the new page will be stored in its place. Which frame is given up and whether the old page is written back to secondary storage depend on which of several page replacement algorithms (discussed later in this section) is used.

As an example, Figure 2.54 shows the contents of page tables for two processes, process 1 and process 2. Process 1 consists of three pages, P_0, P_1, and P_2, whereas process 2 has only two pages, P_0 and P_1. Assume that all the pages of process 1 have read access only and the pages of process 2 have read/write access; this is denoted by *R* and *W* in each page table. Because each frame has 4096 (4K) bytes, the physical address of the beginning of each frame is computed by the product of the frame number and 4096. Therefore, given that P_0 and P_2 of process 1 are loaded into frames 1 and 3, their beginning address in main memory will be 4K = (1 * 4096) and 12K = (3 * 4096), respectively.

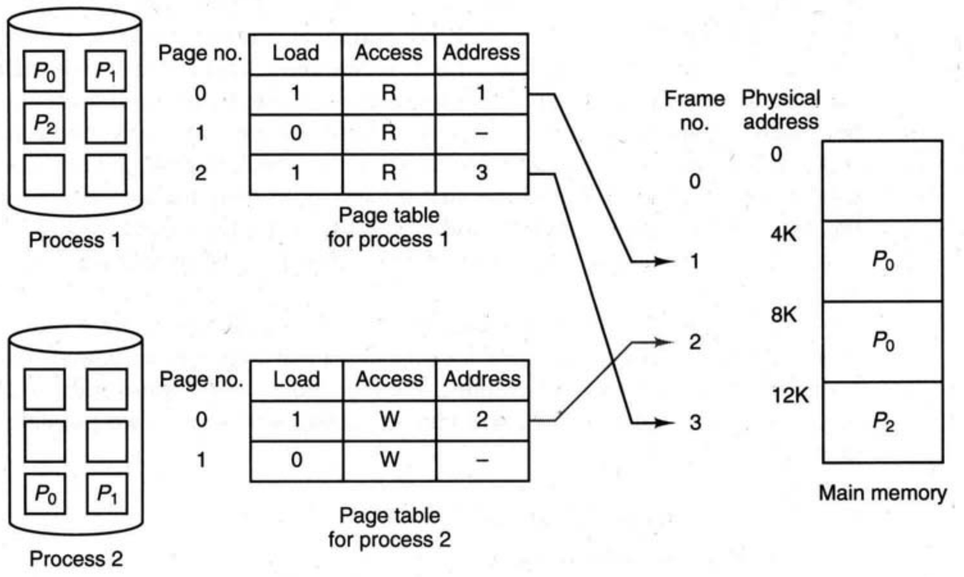


Figure 2.54 Page tables for two different processes, Process 1 and Process 2.

The process of converting a virtual address to a physical address can be sped up by using a high-speed lookup table called a *translation lookaside buffer* (TLB). The page number of the virtual address is fed to the TLB where it is translated to a frame number of the physical address. The TLB can be implemented as an associative memory that has an entry for each of the most recently or likely referenced pages. Each entry contains a page number and other relevant information similar to the page table entry, essentially the frame number and access type. For a given page number, an entry of the TLB that matches (a hit) this page number is used to provide the corresponding frame number. If a match cannot be found in the TLB (a miss), the page table of the corresponding process will be located and used to produce the frame number.

The efficiency of the virtual memory system depends on minimizing the number of page faults. Because the access time of secondary memory is much higher than the access time of main memory, an excessive number of page faults can slow the system dramatically. When a page fault occurs, a page in the main memory must be located and identified as one not needed at the present time so that it can be written back to the secondary memory. Then the requested page can be loaded into this newly freed frame of main memory.

Obviously, paging increases the processing time of a process substantially, because two disk accesses would be required along with the execution of a replacement algorithm. There is an alternative, however, which at times can reduce the number of the disk accesses to just one. This reduction is achieved by adding to the hardware an extra bit to each frame, called a *dirty* bit (also called an *inconsistent* bit). If some modification has taken place to a particular frame, the corresponding dirty bit is set to 1. If the dirty bit for frame *f* is 1, for example, and in order to create an available frame, *f* has been chosen as the frame to swap out, then two disk accesses would be required. If the dirty bit is 0 (meaning that there were no modifications on *f* since it was last loaded), there would be no need to write *f* back to disk. Because the original state of *f* is still on disk (remember that the frames in main memory contain copies of the pages in secondary memory) and no modifications have been made to *f* while in main memory, the page frame containing *f* can simply be overwritten by the newly requested page.

Most replacement algorithms consider the principle of locality when selecting a frame to replace. The principle of locality states that over a given amount of time the addresses generated will fall within a small portion of the virtual address space and that these generated addresses will change slowly with time. Two possible replacement algorithms are

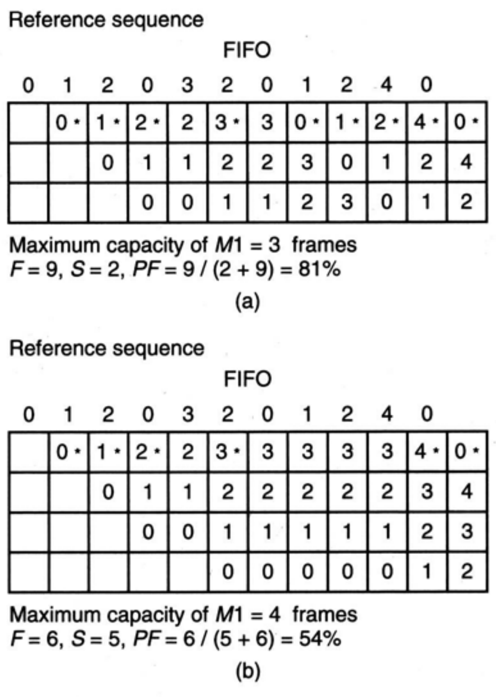
1. First in, first out (FIFO)
2. Least recently used (LRU)

Before the discussion of replacement algorithms, you should note that the efficiency of the algorithm is based on the page size (*Z*) and the number of pages (*N*) the main memory (*M*1) can contain. If $Z = 100$ bytes and $N = 3$, then $M1 = N * Z = 3 * 100 = 300$ bytes.

Another concern with replacement algorithms is the page fault frequency (*PF*). The PF is determined by the number of page faults (*F*) that occurs in an entire execution of a process divided by *F* plus the number of no-fault references (*S*): $PF = F / (S + F)$. The *PF* should be as low a percentage as possible in order to minimize disk accesses. The *PF* is affected by page size and the number of page frames.

First In, First Out. First in, first out (FIFO) is one of the simplest algorithms to employ. As the name implies, the first page loaded will be the first page to be removed from main memory. Figure 2.55a demonstrates how FIFO works as well as how the page fault frequency (*PF*) is determined by using a table. The number of rows in the table represents the number of available frames, and the columns represent each reference to a page. These references come from a given reference sequence 0, 1, 2, 0, 3, 2, 0, 1, 2, 4, 0, where the first reference is to page 0, the second is to page 1, the third is to page 2, and so on. When a page fault occurs, the corresponding page number is put in the top row, representing its precedence, and marked with an asterisk. The previous pages in the table are moved down. Once the page frames are filled and a page fault occurs, the page in the bottom page frame is removed or swapped out. (Keep in mind that this table is used only to visualize a page's current precedence. The movement of these pages does not imply that they are actually being shifted around in *M*1. A counter can be associated with each page to determine the oldest page.)

Once the last reference in the reference sequence is loaded, the *PF* can be calculated. The number of asterisks appearing in the top row equals page faults (*F*) and the items in the top row that do not contain an asterisk equals success (*S*). Considering Figure 2.55, when *M*1 contains three frames, *PF* is 81% for the above reference sequence. When *M*1 contains four frames, *PF* reduces to 54%. That is, *PF* is improved by increasing the number of page frames to 4.



Reference sequence

FIFO

0	1	2	0	3	2	0	1	2	4	0	
	0 •	1 •	2 •	2	3 •	3	0 •	1 •	2 •	4 •	0 •
		0	1	1	2	2	3	0	1	2	4
			0	0	1	1	2	3	0	1	2

Maximum capacity of $M1 = 3$ frames
$F = 9$, $S = 2$, $PF = 9 / (2 + 9) = 81\%$

(a)

Reference sequence

FIFO

0	1	2	0	3	2	0	1	2	4	0	
	0 •	1 •	2 •	2	3 •	3	3	3	3	4 •	0 •
		0	1	1	2	2	2	2	2	3	4
			0	0	1	1	1	1	1	2	3
					0	0	0	0	0	1	2

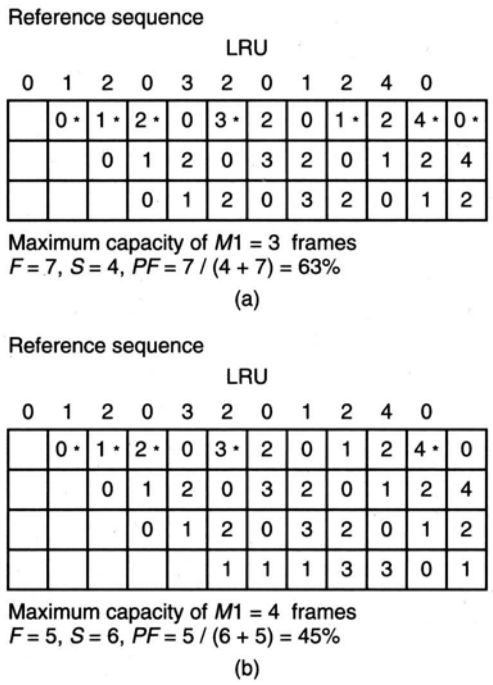
Maximum capacity of $M1 = 4$ frames
$F = 6$, $S = 5$, $PF = 6 / (5 + 6) = 54\%$

(b)

Figure 2.55 Performance of the FIFO replacement technique on two different memory configurations.

The disadvantage of FIFO is that it may significantly increase the time it takes for a process to execute because it does not take into consideration the principle of locality and consequently may replace heavily used frames as well as rarely used frames with equal probability. For example, if an early frame contains a global variable that is in constant use, this frame will be one of the first to be replaced. During the next access to the variable, another page fault will occur, and the frame will have to be reloaded, replacing yet another page.

Least Recently Used. The least recently used (LRU) method will replace the frame that has not been used for the longest time. In this method, when a page is referenced that is already in *M*1, it is placed in the top row and the other pages are shifted down. In other words, the most used pages are kept at the top. See Figure 2.56a. An improvement of *PF* is made using LRU by adding a page frame to *M*1. See Figure 2.56b. In general, LRU is more efficient than FIFO, but it requires more hardware (usually a counter or a stack) to keep track of the least and most recently used pages.


Reference sequence

LRU

0	1	2	0	3	2	0	1	2	4	0	
	0 *	1 *	2 *	0	3 *	2	0	1 *	2	4 *	0 *
		0	1	2	0	3	2	0	1	2	4
			0	1	2	0	3	2	0	1	2

Maximum capacity of $M1 = 3$ frames
$F = 7$, $S = 4$, $PF = 7 / (4 + 7) = 63\%$

(a)

Reference sequence

LRU

0	1	2	0	3	2	0	1	2	4	0	
	0 *	1 *	2 *	0	3 *	2	0	1	2	4 *	0
		0	1	2	0	3	2	0	1	2	4
			0	1	2	0	3	2	0	1	2
					1	1	1	3	3	0	1

Maximum capacity of $M1 = 4$ frames
$F = 5$, $S = 6$, $PF = 5 / (6 + 5) = 45\%$

(b)

Figure 2.56 Performance of LRU replacement technique on two different memory configurations.

Segmentation. Another method of swapping between secondary and main memory is called *segmentation*. In segmentation, a program is broken into variable-length sections known as *segments*. For example, a segment can be a data set or a function within the program. Each process keeps a segment table within main memory that contains basically the same information as the page table. However, unlike pages, segments have variable lengths, and they can start anywhere in the memory; therefore, removing one segment from main memory may not provide enough space for another segment.

There are several strategies for placing a given segment into the main memory. Among the most well-known strategies are *first fit*, *best fit*, and *worst fit*. Each of these strategies maintains a list that represents the size and position of the free storage blocks in the main memory. This list is used for finding a suitable block size for the given segment. The following is an explanation.

First Fit. This strategy puts the given segment into the first suitable free storage. It searches through the free storage list until it finds a block of free storage that is large enough for the segment; then it allocates a block of memory for the segment.

The main advantage of this strategy is that it encourages free storage areas to become available at high-memory addresses by assigning segments to the low-memory addresses whenever possible. However, this strategy will produce free areas that may be too small to hold a segment. This phenomenon is known as *fragmentation*. When fragmentation occurs, eventually some sort of compaction algorithm will have to be run to collect all the small free areas into one large one. This causes some overhead, which degrades the performance.

Best Fit. This strategy allocates the smallest available free storage block that is large enough to hold the segment. It searches through the free storage list until it finds the smallest block of storage that is large enough for the segment. To prevent searching the entire list, the free storage list is usually sorted according to the increasing block size. Unfortunately, like first fit, this strategy also causes fragmentation. In fact, it may create many small blocks that are almost useless.

Worst Fit. This strategy allocates the largest available free storage block for the segment. It searches the free storage list for the largest block. The list is usually sorted according to the decreasing block size.

Again, the worst fit, like the other two strategies, causes fragmentation. However, in contrast to first fit and best fit, worst fit reduces the number of small blocks by always allocating the largest block for the segment.

Example: i486 Microprocessor Addressing Mechanism

The i486 supports both segmentation and paging. It contains a segmentation unit and a paging unit. The i486 has three different address spaces: *virtual*, *linear*, and *physical*. Figure 2.57 represents the relationship between these address spaces. The segmentation unit translates a virtual address into a linear address. When the paging is not used, the linear address corresponds to a physical address. When paging is used, the paging unit translates the linear address into a physical address.

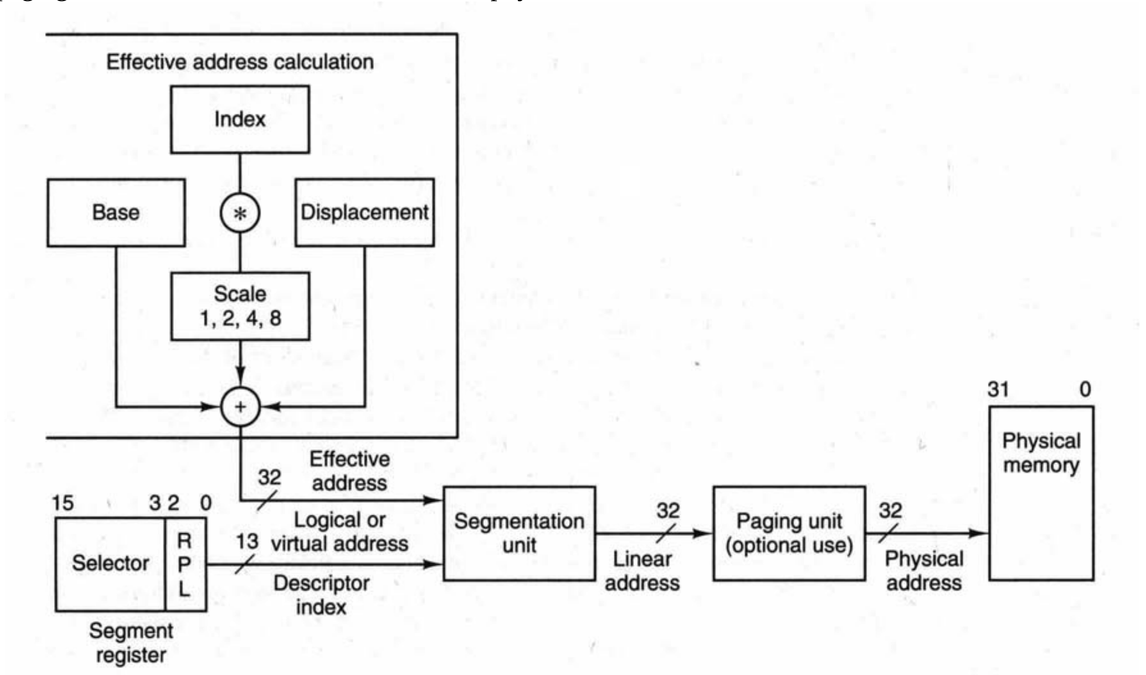


Figure 2.57 Address translation (from [INT 91]). Reprinted with permission of Intel Corporation, Copyright/Intel Corporation 1991.

The virtual address consists of a 16-bit *segment selector* and a 32-bit *effective address*. The segment selector points to a table called segment descriptor (which is the same as segment table); see Figure 2.58. The descriptor contains information about a given segment. It includes the base address of the segment, the length of the segment (limit), read, write, or execute privileges (access rights), and so on. The size of a segment can vary from 1 byte to the maximum size of the main memory, 4 gigabytes (2^{32} bytes). The effective address is computed by adding some combinations of the addressing components. There are three addressing components: *displacement*, *base*, and *index*. The displacement is an 8-, or 32-bit immediate value following the instruction. The base is the contents of any general-purpose register and often points to the beginning of the local variable area. The index is the contents of any general-purpose register and often is used to address the elements of an array or the characters of a string. The index may be multiplied by a scale factor (1, 2, 4, or 8) to facilitate certain addressing, such as addressing arrays or structures. As shown in Figure 2.57, the effective address is computed as:

effective address = *base register* + (*index register* * *scaling*) + *displacement*

The segmentation unit adds the contents of the segment base register to the effective address to form a 32-bit linear address (see Figure 2.58).

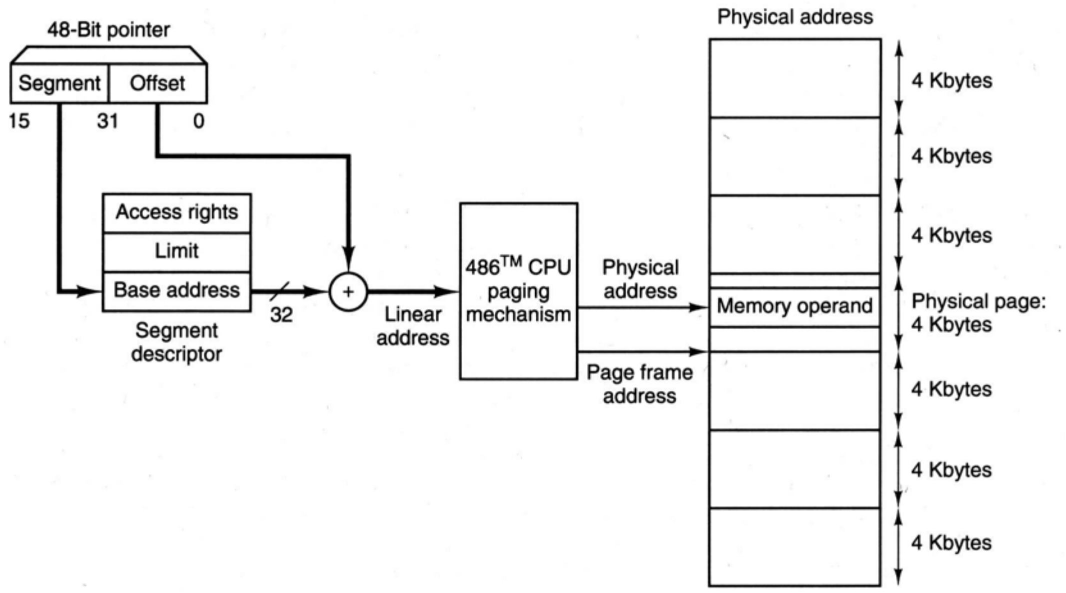


Figure 2.58 Paging and segmentation (from [INT 91]). Reprinted with permission of Intel Corporation, Copyright/Intel Corporation 1991.

The paging unit can be enabled or disabled by software control. When paging is enabled, the linear address will be translated to a physical address. The paging unit uses two levels of tables to translate a linear address into a physical address. Figure 2.59 represents these two levels of tables, a page directory that points to a page table. The page directory can have up to 1024 entries. Each page directory entry contains the address of a page table and statistical information about the table, such as read/write privilege bits, a dirty bit, and a present bit [INT 92]. (The dirty bit is set to 1 when a write to an address covered by the corresponding page occurs. The present bit indicates whether a page directory or page table entry can be used in address translation.) The starting address of the page directory is stored in a register called the page directory base address register (*CR*3, root). The upper 10 bits of the linear address are used as an index to select one of the page directory entries. Similar to the page directory, the page table allows up to 1024 entries, where each entry contains the address of a page frame and statistical information about the page. The main memory is divided into 4-Kbytes page frames. The address bits 12 to 21 of the linear address are used as an index to select one of the page table entries. The page frame address of the selected entry is concatenated with the lower 12 bits of the linear address to form the physical address.

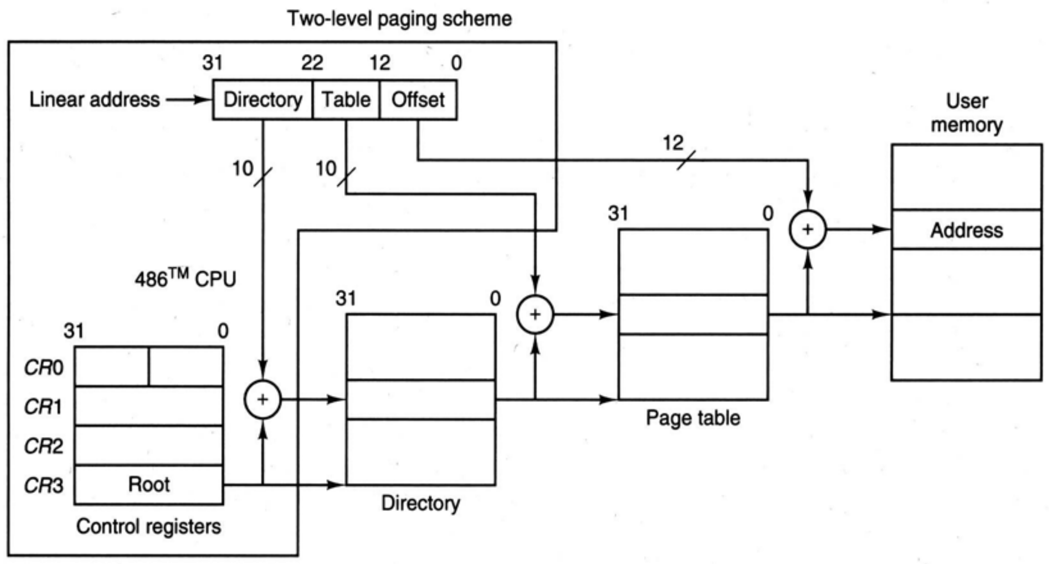


Figure 2.59 Paging mechanism (from [INT 91]). Reprinted with permission of Intel Corporation, Copyright/Intel Corporation 1991.

This paging mechanism requires access to two levels of tables for every memory reference. The access to the tables degrades the performance of the processor. To prevent this degradation, the *i*486 uses a translation lookaside buffer (TLB) to keep the most commonly used page table entries. The TLB is a four-way, set associative cache with 32 entries. Given that a page has 4 Kbytes, the 32-entry TLB can cover 128 Kbytes of main memory addresses. This size of coverage gives a hit rate of about 98% for most applications [INT 92]. That is, the processor needs to access the two tables for only 2% of all memory references. Figure 2.60 represents how the TLB is used in the paging unit.

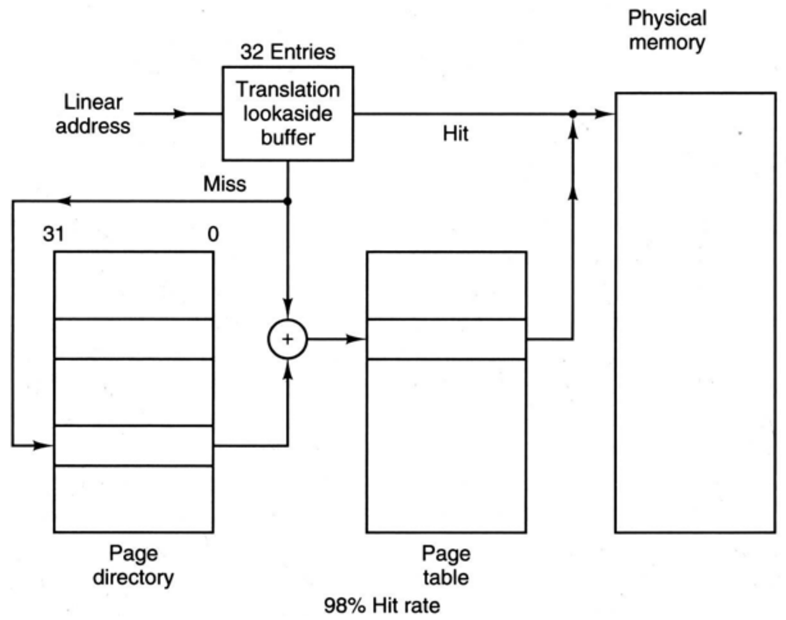


Figure 2.60 Translation lookalike buffer (from [INT 91]). Reprinted with permission of Intel Corporation, Copyright/Intel Corporation 1991.

2.7 INTERRUPTS AND EXCEPTIONS

During execution of a program, an interrupt or an exception may cause the processor to temporarily suspend the program in order to service the needs of a particular event. The event may be external to the processor or may be internal. Interrupts are used for handling asynchronous external events, such as when an external device wants to perform an I/O operation. Exceptions are used for handling synchronous internal events that occur due to instruction faults, such as divide by zero [INT 91].

Interrupts are caused by asynchronous signals applied to certain input lines, called interrupt request lines, of a processor. When an external event triggers an interrupt during execution of a program, the processor suspends the execution of the program, determines the source of the interrupt, acknowledges the interrupt, saves the state of the program (such as program counter and registers) in a stack, loads the address of the proper interrupt service routine into the program counter, and then processes the interrupt by executing the interrupt service routine. After finishing the processing of the interrupt, it resumes the interrupted program. Often, interrupts are classified into different levels of priorities. For example, disk drives are given higher priorities then keyboards. Thus, if an interrupt is being processed and a higher-priority interrupt arrives, the process of the former interrupt will be suspended and the latter interrupt will be serviced. In general, the priority levels are divided into two groups: *maskable* and *nonmaskable*. A maskable interrupt can be handled or delayed by the processor. However, nonmaskable interrupts have very high priority. Often, when a nonmaskable interrupt routine is started, it cannot be interrupted by any other interrupt. They are usually intended for catastrophic events, such as memory error detection or power failure. A typical use of a nonmaskable interrupt would be to start a power failure routine when the power goes down.

Depending on how the processor obtains the address of interrupt service routines, the maskable interrupts are further divided into three classes: *nonvectored*, *vectored*, and *autovectored*. In a nonvectored interrupt scheme, each interrupt request line is associated with a fixed interrupt service routine address. When an external device sends a request on one of the interrupt request lines, the processor loads the corresponding interrupt service routine address into the program counter. In contrast to nonvectored interrupt, in the vectored interrupt scheme the external device supplies a vector number from which the processor retrieves the address of the interrupt service routine. In the vectored interrupt scheme, an interrupt request line is shared among all the external devices. In this way, the number of interrupt sources can be increased independent of the number of available interrupt request lines. When the processor sends an acknowledgment for an interrupt request, the device that requested the interrupt places a vector number on the data bus (or system bus). The processor converts the vector number to a vector address that points to a vector in the memory. This vector contains the address of the interrupt service routine. The processor fetches the vector and loads that into the program counter. Such vectors are usually stored in a specific memory space called a *vector table*. An autovectored interrupt is similar to the vectored interrupt except that in the former case an external vector number is not supplied to the processor. Instead the processor generates a vector number depending on the specific interrupt request line that the interrupt was detected on. The vector number then is converted to a vector address.

Exceptions are used to handle instruction faults [INT 91]. They are divided into three classes: *faults*, *traps*, and *aborts*. An exception is called a *fault* if it can be detected and serviced before the execution of a faulty instruction. For example, as a result of a programming error such as divide by zero, stack overflow, or an illegal instruction, a fault is generated.

An exception is called a *trap* if it is serviced after the execution of the instruction that requested (or caused) the exception. For example, a trap could be an exception routine defined by the user. Often a programmer uses traps as a substitute for CALL instructions to frequently used procedures in order to save some execution time.

An exception is called an *abort* when the precise location of the cause cannot be determined. Often, aborts are used to report severe errors, such as a hardware failure.