requires excessive work to obtain the simplified input equations for the flip-flops. Here, the design can be simplified if we take into consideration the fact that the decoder outputs are available for use in the design. Instead of using flip-flop outputs as the present state conditions, we might as well use the outputs of the decoder to obtain this information. These outputs supply a single signal representing each of the possible present states of the circuit. Moreover, instead of using maps to simplify the flip-flop equations, we can obtain them directly by inspection of the state table. For example, from the next-state conditions in the table, we find that the next state of M_0 is equal to 1 when the present state is IDLE and input G is equal to 1 or when the present state is MUL1 and input Z is equal to 0. These conditions give

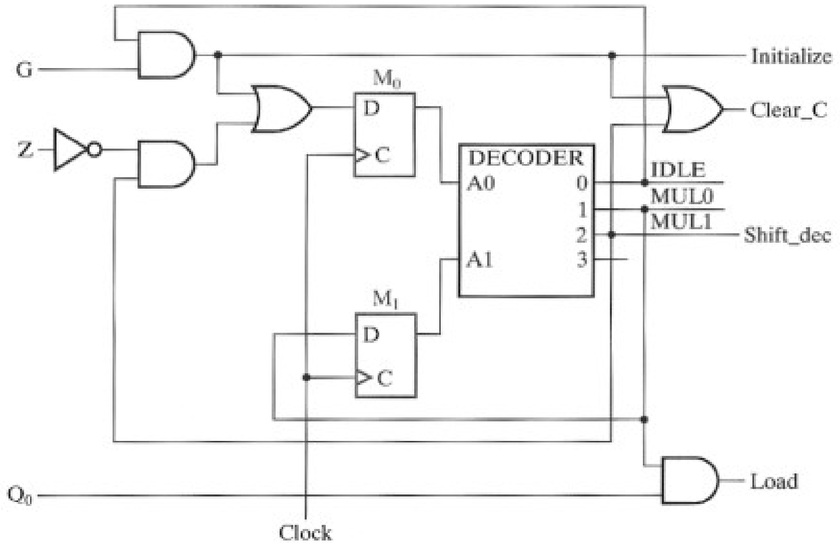
$$D_{M_0} = \text{IDLE} \cdot G + \text{MUL1} \cdot \overline{Z}$$

for the D input of the M_0 flip-flop. Similarly, the D input of the M_1 flip-flop is

$$D_{M_1} = \text{MUL0}$$

Note that these equations derived by inspection from the state table use the state names rather than the state variable names, since the decoder producing the state symbols is present. In some cases, it may be possible to find simpler D flip-flop input equations by using the state variables directly instead of the states. We can remove redundancy and reduce cost by writing the Boolean equations for the decoder and applying a simplification program to the set of control equations.

The logic diagram for the control appears in Figure 8-10. It consists of a two-bit register with flip-flops M_1 and M_0 and a 2-to-4-line decoder. The three outputs



□ **FIGURE 8-10**
Control Unit for Binary Multiplier Using a Sequence Register and a Decoder

of the decoder are used to generate the control outputs, as well as inputs to the next-state logic. The outputs Initialize, Clear_C, Shift_dec, and Load are determined from Table 8-1. Initialize and Shift_dec are already available as signals, so that only labeled output lines are added. However, as shown in the figure, we must add logic gates for Clear_C and Load. We complete the binary multiplier design by connecting the outputs of the control unit to the control inputs of the datapath.

One Flip-Flop per State

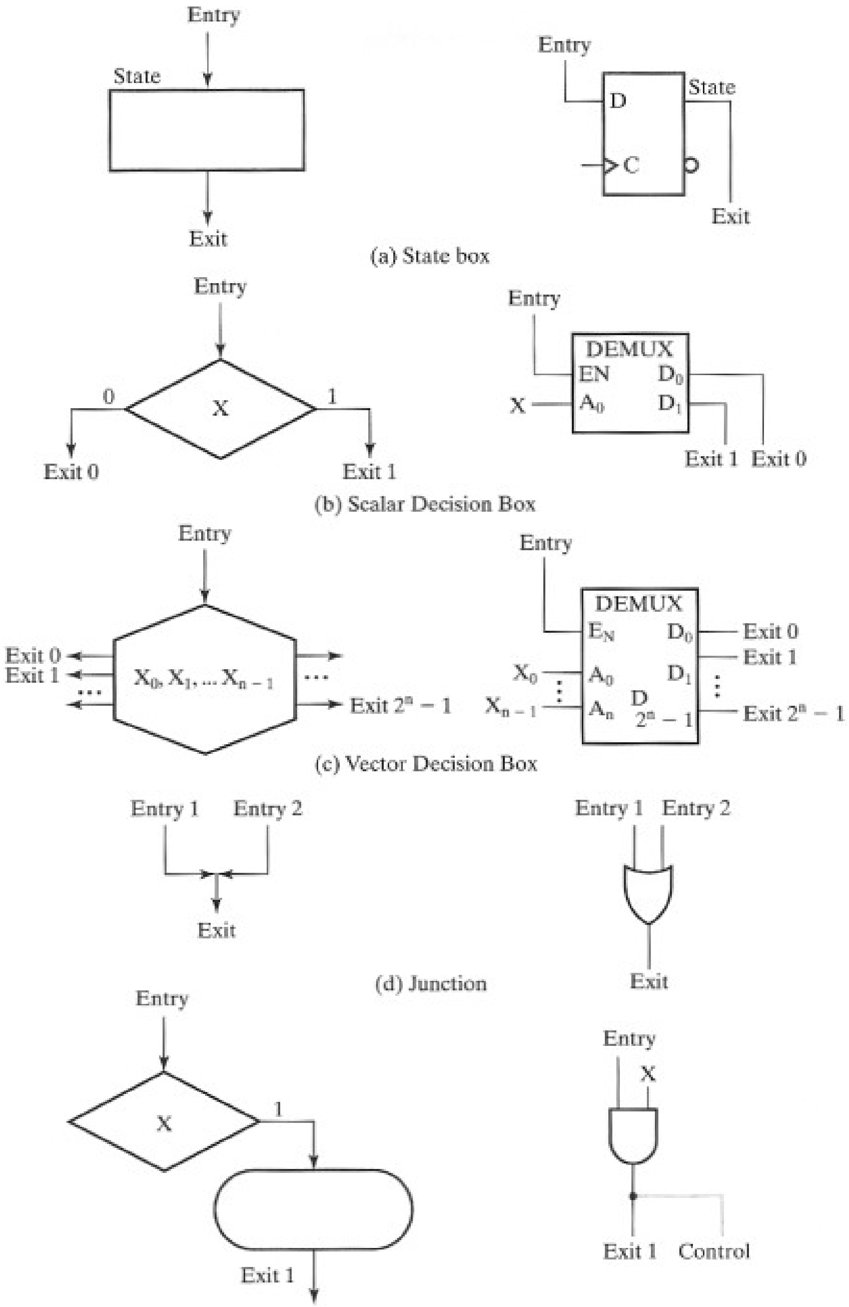
Another possible method of control logic design is the use of one flip-flop per state. A flip-flop is assigned to each of the states, and at any time, only one of the flip-flops contains a 1, with all the rest containing 0. When the 1 is in the flip-flop assigned to a particular state, the sequential circuit is in that same state. The single 1 propagates from one flip-flop to another under the control of decision logic. In such a configuration, each flip-flop represents a state that is present only when the single 1 is stored in the flip-flop.

It is obvious that, short of some error detection or correction techniques, this method uses the maximum number of flip-flops for the sequential circuit. For example, a sequential circuit with 12 states using minimum variable encoding needs four flip-flops. With one flip-flop per state, the circuit requires 12 flip-flops, one for each state. At first glance, it may seem that this method would increase the cost of the system, since more flip-flops are used. But the method offers some cost advantages that may not be apparent. One advantage is the simplicity with which the logic can be designed—merely by inspection of the ASM chart or state diagram. No state or excitation tables are needed if *D* flip-flops are employed. This offers a savings in design effort.

Figure 8-11 shows the symbol replacement rules for transforming an ASM chart into a sequential circuit with one flip-flop per state. These rules are most easily applied to an ASM chart representing only sequencing information, such as that of Figure 8-9. Each rule specifies the replacement of a component of an ASM chart with a logic circuit. As shown in Figure 8-11(a), the state box is replaced by a *D* flip-flop labeled with the name of the state. The entry to the state box corresponds to the *D* input to the flip-flop. The exit of the state box corresponds to the output of the flip-flop.

In Figure 8-11(b), the scalar decision box is replaced by a 2-way demultiplexer. The signal corresponding to the entry to the decision box is sent to one of two exit lines, depending on the value of signal X. If X is 0, the signal is sent to the exit 0 line; if X is 1, the signal is sent to the exit 1 line. So, for example, if the single 1 in the circuit is on the entry to the decision box, and X is 0, the 1 is passed to the exit 0 line. The demultiplexer acts like a switch that directs the 1 through the paths in the circuit corresponding to paths in the ASM chart.

In Figure 8-11(c), the vector decision box is replaced by an n-way demultiplexer. The signal corresponding to the entry to the decision box is sent to one of the $2^n - 1$ lines, depending on the value of the signal vector $X = X_0, ..., X_{n-1}$. If X is 0, the signal is sent to the exit 0 line; if X is 9, the signal is sent to the exit 9 line. So, for example, if the single 1 in the circuit is on the entry to the decision box, and X is 9,



(a) State box

(b) Scalar Decision Box

(c) Vector Decision Box

(d) Junction

(e) Conditional Output Box

□ **FIGURE 8-11**
Transformation Rules for Control Unit with One Flip-Flop per State

the 1 is passed to the exit 9 line. The demultiplexer acts like a switch that directs the 1 through the paths in the circuit corresponding to paths in the ASM chart.

The junction in Figure 8-11(d) is any point at which two or more directed lines in the ASM chart join together. If a 1 is present in the circuit on any line corresponding to one of the entry paths, then it must appear on the line corresponding to the exit path, giving that line the value 1. If none of the lines corresponding to entry paths into the junction have the value 1, then the exit line must have the value 0. Thus, the junction is replaced by an OR gate.

With these four transformations, the sequencing part of the ASM chart can be replaced by a circuit with one flip-flop per state, just by inspection. In order to handle outputs, it is merely a matter of attaching control lines to the proper locations in the circuit or adding output logic. The outputs are based on the original ASM chart or the control signal table derived from the chart. Attaching a control line based on an ASM chart is illustrated by the conditional output box shown in Figure 8-11(e). The conditional output box in the ASM chart is just replaced by a connection in the circuit. But to cause the output actions to happen, a control line is tapped from the connection and labeled with the output variable. The transformation is shown in blue for clarity.

We now use these transformations to find the control unit with one flip-flop per state for the binary multiplier.

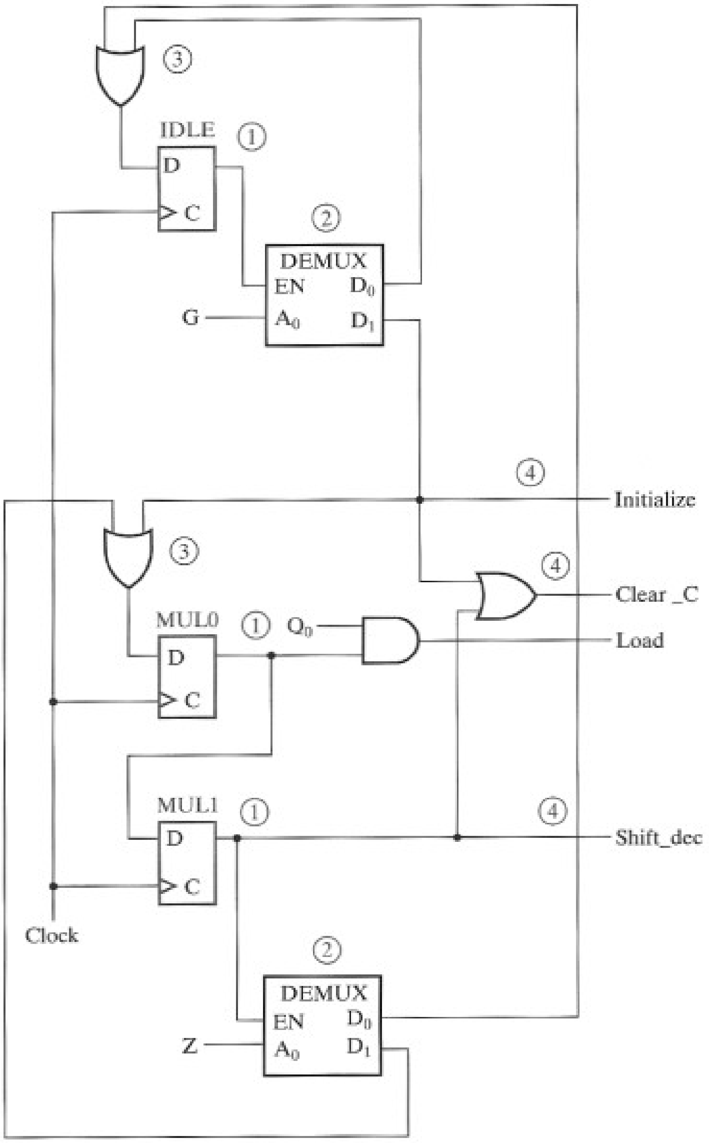
EXAMPLE 8-1 Binary Multiplier

The ASM chart in Figure 8-9 will be used for the sequencing part of the design. Note that the binary codes given are ignored, since they were for the former design approach. The resulting logic diagram is shown in Figure 8-12.

First, we replace each of the three state boxes by a *D* flip-flop labeled with the name of the state, as indicated by the circled 1's in the figure. Second, each of the decision boxes is replaced by a demultiplexer with the decision variable as its selection input, as indicated by the circled 2's in the figure. Third, each junction is replaced by an OR gate, as indicated by the circled 3's. Finally, the connections represented by the directed lines in the ASM chart are added from the outputs to the inputs of the corresponding components.

To handle the control outputs, we can use either Table 8-1 or the original ASM chart in Figure 8-7. From the table, we see that the Boolean function for Initialize is already available in the logic diagram, so we simply add the output labeled Initialize. Likewise, the output for Shift_dec can be added. For Clear_C and Load, however, logic gates are added. All of the output connections and logic added are designated by the circled 4's in Figure 8-12.

One final issue in the design of the control logic with one flip-flop per state is initialization to the state having a 1 in the IDLE flip-flop and a 0 in all of the others. This can be done by using an asynchronous PRESET input on the IDLE flip-flop and an asynchronous CLEAR on the other flip-flops. If only an asynchronous CLEAR is available, rather than both PRESET and CLEAR, a NOT gate can be placed just before the *D* input and another NOT gate just after the



□ **FIGURE 8-12**
Control Unit with One Flip-Flop per State for the Binary Multiplier

output of the IDLE flip-flop. Then the IDLE flip-flop will actually contain a 0 when in state IDLE and a 1 at all other times. This permits the asynchronous CLEAR to be used to initialize all three flip-flops in the circuit. It should be noted that, other than for resetting the circuit, the use of asynchronous flip-flop inputs for implementing ASMs or other sequential circuits is generally poor design practice. ■

Once the basic design of the control logic with one flip-flop per state is completed, it may be desirable to refine the design. For example, if there are a number of junctions connected together by lines, the OR gates that resulted from the transformation may be combined. Also, demultiplexers cascaded with each other may be combined. Other logic reduction or and technology mapping may also be applied to the design.

8-5 HDL Representation of the Binary Multiplier—VHDL

The binary multiplier just studied can be represented during the design process as a behavioral VHDL description. Such a description for a 4-bit version of the multiplier appears in Figures 8-13 and 8-14. This VHDL code represents the block diagram in Figure 8-6 and the ASM chart in Figure 8-7. The VHDL code consists of entity `binary_multiplier` and an architecture `behavior_4`. The architecture contains two assignment statements and three processes. The processes are similar to those used for the sequence recognizer in Chapter 6. The primary difference is that the output function process has been replaced by a process describing the datapath register transfers. Due to this change, the VHDL representation corresponds more closely to the description in Table 8-1 and the ASM chart in Figure 8-9 than to the ASM chart in Figure 8-7.

In the entity, multiplier inputs and outputs are defined. At the beginning of the architecture, a type declaration defines the three states. Internal signals, some of which will generate registers are declared next. Among these are `state` and `next_state` for the control, registers `A`, `B`, `P` and `Q`, and flip-flop `C`. Also, intermediate signal `Z` is declared for convenience. Next, an assignment is made which forces `Z` to be 1 whenever `P` contains value 0. Following this, the outputs of concatenated registers `A` and `Q` are assigned to the multiplier output `MULT_OUT`. This is necessary, rather than making `A` and `Q` circuit outputs, to permit `A` and `Q` to be used within the circuit.

The remainder of the description consists of the three processes. The first process describes the state register and includes a `RESET` as well as the clocking. The second process describes the next state function from Figure 8-8. Note that, since clocking and `RESET` are included in the state register, they do not appear here. In the sensitivity list, all signals that can affect the next state, `G`, `Z`, and `state` are included. Otherwise, this process resembles that for the `next_state` process in the sequence recognizer.

The final process in Figure 8-14 describes the datapath function. Since the conditions for performing an operation are defined in terms of the states and inputs, this process also implicitly defines the control signals given in Table 8-1. These control signals do not appear explicitly, however. Since the datapath function has registers as all assignment destinations, all transfers are controlled by `CLK`. Since contents will be loaded into these registers before the multiply operation is ever performed, it is unnecessary to provide a reset for these registers. The first **`if`** statement controls the loading of the multiplicand in register `B` and the second **`if`** statement controls the loading of the multiplier into register `Q`.

```
-- Binary Multiplier with n = 4: VHDL Description
-- See Figures 8-6 and 8-7 for block diagram and ASM Chart
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity binary_multiplier is
  port(CLK, RESET, G, LOADB, LOADQ: in std_logic;
    MULT_IN: in std_logic_vector(3 downto 0);
    MULT_OUT: out std_logic_vector(7 downto 0));
end binary_multiplier;

architecture behavior_4 of binary_multiplier is
  type state_type is (IDLE, MUL0, MUL1);
  signal state, next_state : state_type;
  signal A, B, Q: std_logic_vector(3 downto 0);
  signal P: std_logic_vector(1 downto 0);
  signal C, Z: std_logic;
begin
  Z <= P(1) NOR P(0);
  MULT_OUT <= A & Q;

  state_register: process (CLK, RESET)
  begin
    if (RESET = '1') then
      state <= IDLE;
    elsif (CLK'event and CLK = '1') then
      state <= next_state;
    end if;
  end process;

  next_state_func: process (G, Z, state)
  begin
    case state is
      when IDLE =>
        if G = '1' then
          next_state <= MUL0;
        else
          next_state <= IDLE;
        end if;
      when MUL0 =>
        next_state <= MUL1;
      when MUL1 =>
        if Z = '1' then
          next_state <= IDLE;
        else
          next_state <= MUL0;
        end if;
```

□ **FIGURE 8-13**
VHDL Description of a Binary Multiplier

```
    end case;
  end process;

  datapath_func: process (CLK)
  variable CA: std_logic_vector(4 downto 0);
  begin
    if (CLK’event and CLK = '1') then
      if LOADB = '1' then
        B <= MULT_IN;
      end if;
      if LOADQ = '1' then
        Q <= MULT_IN;
      end if;
      case state is
        when IDLE =>
          if G = '1' then
            C <= '0';
            A <= "0000";
            P <= "11";
          end if;
        when MUL0 =>
          if Q(0) = '1' then
            CA := ('0' & A) + ('0' & B);
          else
            CA := C & A;
          end if;
            C <= CA(4);
            A <= CA(3 downto 0);
        when MUL1 =>
          C <= '0';
          A <= C & A(3 downto 1);
          Q <= A(0) & Q(3 downto 1);
          P <= P - "01";
      end case;
    end if;
  end process;
end behavior_4;
```

□ **FIGURE 8-14**
VHDL Description of a Binary Multiplier (Continued)

The register transfers directly involved in the multiplication are controlled by a **case** statement dependent upon the control state, input `G`, and internal signals `Q(0)` and `Z`. These transfers are outlined in Figure 8-7 and Table 8-1. Representation of the addition in state `MUL0` requires some effort. First of all, to perform addition on std_logic vectors, a **use** statement appears just before the entity declaration for the package `ieee.std_logic_unsigned.`**`all`**. In addition to the sum from the addition, we also need to transfer the carry out, C_{out}, from the addition into C. To achieve this, we perform a 5-bit addition with 0's appended to the left of A and B and the result assigned to a 5-bit variable CA. The alternative would be to write C & A as the transfer destination, but use of concatenation & in destinations is not permitted in VHDL. Since CA is a variable, its value is assigned immediately and is available for assignment to C and A after the **if** statement. In state MUL1, the shift is performed by using concatenation, as was done in the example in Chapter 5. *P* is decremented by subtracting a 2-bit constant with value 1.

This description can be simulated to validate its correctness and synthesized to automatically produce the logic if desired.

8-6 HDL Representation of the Binary Multiplier—Verilog

The binary multiplier just studied can be represented during the design process as a behavioral Verilog description. Such a description for a 4-bit version of the multiplier appears in Figures 8-15 and 8-16. This Verilog code represents the block diagram in Figure 8-6 and the ASM chart in Figure 8-7. The Verilog code is contained in a module binary_multiplier_v. The description contains two assignment statements and three processes. The processes are similar to those used for the sequence recognizer in Chapter 4. The primary difference is that the output function process has been replaced by a process describing the datapath register transfer. Due to this change, the Verilog representation corresponds more closely to the description in Table 8-1 and the ASM chart in Figure 8-9 than to the ASM chart in Figure 8-7.

At the beginning of the description, multiplier inputs and outputs are defined. A parameter declaration defines the three states and their binary codes. Internal signals of type register are defined. Among these are the state and next_state for the control, registers A, B, P and Q, and flip-flop C. Based on clocking specifications, most of these will become actual positive-edge-triggered registers. The notable exception is next_state. Also, intermediate signal Z of type wire is declared for convenience. Next, an assignment is made which forces Z to be 1 whenever P contains value 0. This assignment uses the operation OR (|)as a *reduction operator*. Reduction is the application of an operator to a wire or register that combines the individual bits. In this case, the application of OR to P causes all bits of P to be ORed together. Since the OR is preceded by a ~, that overall operation performed is a NOR. Other operators may also be applied as reduction operators. The second assignment statement assigns the outputs of concatenated registers A and Q to the multiplier output MULT_OUT. This is done for convenience to make that output a single structure.

The remainder of the description consists of the three processes. The first process describes the state register and includes a RESET as well as the clocking. The second process describes the next state function from Figure 8-9. Note that since clocking and RESET are included in the state register, they do not appear here. In the event control statement, all signals that can affect the next state, G, Z, and state are included. Otherwise, this process resembles that for the next state process in the sequence recognizer.

```
// Binary Multiplier with n = 4: Verilog Description
// See Figures 8-6 and 8-7 for block diagram and ASM Chart

module binary_multiplier_v (CLK, RESET, G, LOADB, LOADQ,
       MULT_IN, MULT_OUT);
input CLK, RESET, G, LOADB, LOADQ;
input [3:0] MULT_IN;
output [7:0] MULT_OUT;
reg [1:0] state, next_state, P;
parameter IDLE = 2’b00, MUL0 = 2’b01, MUL1 = 2’b10;
reg [3:0] A, B, Q;
reg C;
wire Z;

assign Z = ~| P;
assign MULT_OUT = {A,Q};

//state register
always@(posedge CLK or posedge RESET)
begin
  if (RESET == 1)
    state <= IDLE;
  else
  state <= next_state;
end

//next state function
always@(G or Z or state)
begin
  case (state)
    IDLE:
      if (G == 1)
        next_state <= MUL0;
      else
        next_state <= IDLE;
    MUL0:
      next_state <= MUL1;
    MUL1:
      if (Z == 1)
        next_state <= IDLE;
      else
        next_state <= MUL0;
  endcase
end

//datapath function
always@(posedge CLK)
```

□ **FIGURE 8-15**
Verilog Description of a Binary Multiplier

```
begin
  if (LOADB == 1)
    B <= MULT_IN;
  if (LOADQ == 1)
    Q <= MULT_IN;
  case (state)
    IDLE:
      if (G == 1)
        begin
          C <= 0;
          A <= 4'b0000;
          P <= 2'b11;
        end
    MUL0:
      if (Q[0] == 1)
        {C, A} = A + B;
    MUL1:
      begin
        C <= 1'b0;
        A <= {C, A[3:1]};
        Q <= {A[0], Q[3:1]};
        P <= P - 2'b01;
      end
  endcase
end
endmodule
```

□ **FIGURE 8-16**
Verilog Description of a Binary Multiplier (Continued)

The final process describes the datapath function. Since the conditions for performing an operation are defined in terms of the states and inputs, this process also implicitly defines the control signals given in Table 8-1. These control signals do not appear explicitly, however. Since the datapath function has registers as all assignment destinations, all transfers are controlled by `CLK`. Since contents will be loaded into these registers before the multiply operation is ever performed, it is unnecessary to provide a reset for these registers. The first `if` statement controls the loading of the multiplicand into register `B` and the second `if` statement controls the loading of the multiplier into register `Q`.

The register transfers directly involved in the multiplication are controlled by a **case** statement dependent upon the control state, input `G`, and internal signals `Q(0)` and `Z`. These transfers are outlined in Figure 8-7 and Table 8-1. Representation of the addition in state `MUL0` uses concatenation of `C` and `A` to obtain the carry out, C_{out}, for loading into C. Verilog does permit the used of two 4-bit operands with a 5-bit result for the addition. In state `MUL1`, the shift is performed by using concatenation as was done in the example in Chapter 5. `P` is decremented by subtracting a 2-bit constant with value 1.

This description can be simulated to validate its correctness and synthesized to automatically produce the logic if desired.

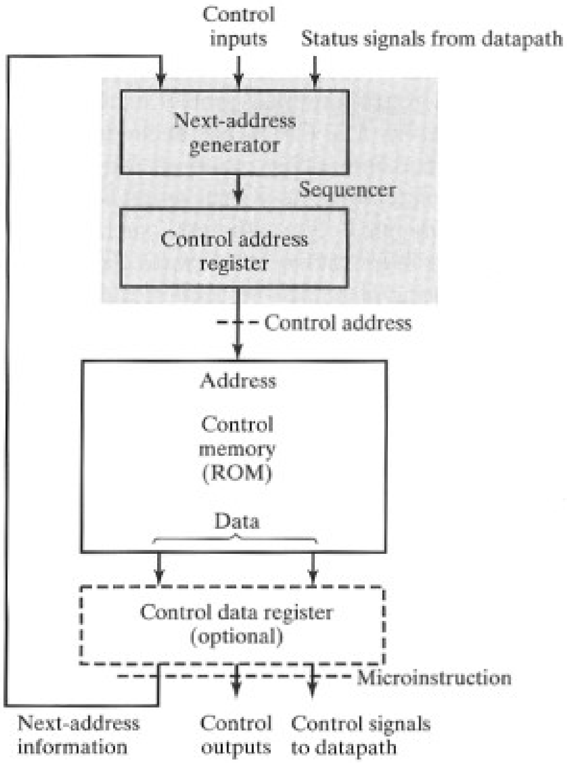
8-7 Microprogrammed Control

A control unit with its binary control values stored as words in memory is called a *microprogrammed control*. Each word in the control memory contains a *microinstruction* that specifies one or more microoperations for the system. A sequence of microinstructions constitutes a *microprogram*. The microprogram is usually fixed at the system design time and so is stored in ROM. Microprogramming involves placing representations for combinations of values of control variables in words of ROM. These representations are accessed via successive read operations for use by the rest of the control logic. The contents of a word in ROM at a given address specify the microoperations to be performed for both the datapath and the control unit. A microprogram can also be stored in RAM. In this case, it is loaded at system startup from some form of nonvolatile storage, such as a magnetic disk. With either ROM or RAM, the memory in the control unit is called *control memory*. If RAM is used, the memory is referred to as *writable control memory*.

Figure 8-17 shows the general configuration of a microprogrammed control. The control memory is assumed to be a ROM within which all control microprograms are permanently stored. The *control address register* (*CAR*) specifies the address of the microinstruction. The *control data register* (*CDR*), which is optional, may hold the microinstruction currently being executed by the datapath and the control unit. One function of the control word is to determine the address of the next microinstruction to be executed. This microinstruction may be the next one in sequence, or it may be located somewhere else in the control memory. Therefore, one or more bits that specify the method for determining the address of the next microinstruction are present in the current microinstruction. The next address may also be a function of status and external control inputs. When a microinstruction is executed, the *next-address generator* produces the next address. This address is transferred to the *CAR* on the next clock pulse and is used to read the next microinstruction to be executed from ROM. Thus, the microinstructions contain bits for activating microoperations in the datapath and bits that specify the sequence of microinstructions executed.

The next-address generator, in combination with the *CAR*, is sometimes called a microprogram *sequencer*, since it determines the sequence of instructions read from control memory. The address of the next microinstruction can be specified in several ways, depending on the sequencer inputs. Typical functions of a microprogram sequencer are incrementing the *CAR* by one and loading the *CAR*. Possible sources for the load operation include an address from control memory, an externally provided address, and an initial address to start control unit operation.

The *CDR* holds the present microinstruction while the next address is computed and the next microinstruction is read from memory. The *CDR* breaks up the long combinational delay paths through the control memory followed by the datapath. Its presence allows the system to use a higher clock frequency and process information faster. The inclusion of a *CDR* in a system, however, complicates the



□ **FIGURE 8-17**
Microprogrammed Control Unit Organization

sequencing of microinstructions, particularly when decisions are made based on status bits. For simplicity in our brief discussion, we omit the *CDR* and take the microinstructions directly from the ROM outputs. The ROM operates as a combinational circuit, with the address as the input and the corresponding microinstruction as the output. The contents of the specified word in ROM remain on the output lines as long as the address value is applied to the inputs. No read/write signal is needed, as it is with RAM. Each clock pulse executes the microoperations specified by the microinstruction and also transfers a new address to the *CAR*. In this case, the *CAR* is the only component in the control that receives clock pulses and stores state information. The next-address generator and the control memory are combinational circuits. Thus, the state of the control unit is given by the contents of the *CAR*.

Microprogrammed control has been a very popular alternative implementation technique for control units for both programmable and nonprogrammable systems. However, as systems have become more complex and performance specifications have increased the need for concurrent parallel sequences of activities, the lockstep nature of microprogramming has become less attractive for control unit implementation. Further, a large ROM or RAM tends to be much slower than the corresponding combinational logic. Finally, HDLs and synthesis tools facilitate the design of complex control units without the need for a lockstep programmable design approach. Overall, microprogrammed control for the design of control units, particularly direct datapath control in CPUs, has declined significantly. However, a new flavor of microprogrammed control has emerged, for implementing legacy computer architectures. These architectures have instruction sets that do not follow contemporary architecture principles. Nevertheless, such architectures must be implemented due to massive investments in software that uses them. Further, the contemporary architecture principles must be used in the implementations to meet performance goals. The control for these systems is hierarchical with microprogrammed control selectively used at the top level for complex instruction implementation and hardwired control at the lower level for implementing simple instructions and steps of complex instructions at a very rapid rate. This flavor of microprogramming is covered for a Complex Instruction Set Computer (CISC) in Chapter 12.

Information on the more traditional flavor of microprogrammed control, derived from past editions of this text, is available in a supplement, Microprogrammed Control, on the Companion Website for the text.

8-8 Chapter Summary

This chapter has examined the interaction between datapaths and control units and the difference between programmed and nonprogrammed systems. The algorithmic state machine (ASM) is a means for representing and specifying control functions. A binary multiplier was used to illustrate ASM chart formulation. Two implementation approaches to sequential circuit design, sequence register plus decoder and one flip-flop per state, were provided, in addition to the basic design procedure in Chapter 4. VHDL and Verilog alternatives for describing combinations of datapath and control were also illustrated. Finally, microprogrammed control was briefly discussed.

References

1. Mano, M. M. *Computer Engineering: Hardware Design:* Englewood Cliffs, NJ: Prentice Hall, 1988.
2. Mano, M. M. *Digital Design,* 3rd Ed. Englewood Cliffs, NJ: Prentice Hall, 2002.
3. *IEEE Standard VHDL Language Reference Manual.* (ANSI/IEEE Std 1076-1993; revision of IEEE Std 1076-1987). New York: The Institute of Electrical and Electronics Engineers, 1994.
4. Smith, D. J. *HDL Chip Design*. Madison, AL: Doone Publications, 1996.
5. *IEEE Standard Description Language Based on the Verilog(TM) Hardware Description Language* (IEEE Std 1364-1995). New York: The Institute of Electrical and Electronics Engineers, 1995.

6. Palnitkar, S. *Verilog HDL: A Guide to Digital Design and Synthesis.* SunSoft Press (A Prentice Hall Title), 1996.

7. Thomas, D. E., and P. R. Moorby. *The Verilog Hardware Description Language* 4th ed. Boston: Kluwer Academic Publishers, 1998.

Problems

The plus (+) indicates a more advanced problem and the asterisk (*) indicates a solution is available on the Companion Website for the text.

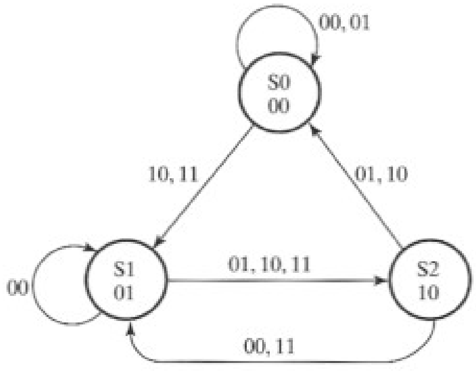
8–1. *A state diagram of a sequential circuit is given in Figure 8-18. Find the corresponding ASM chart. Minimize the chart complexity by using both vector and scalar decision boxes. The inputs to the circuit are X_1 and X_2, and the outputs are Z_1 and Z_2.

8–2. *Find the response for the ASM chart in Figure 8-19 to the following sequence of inputs (assume that the initial state is ST1):

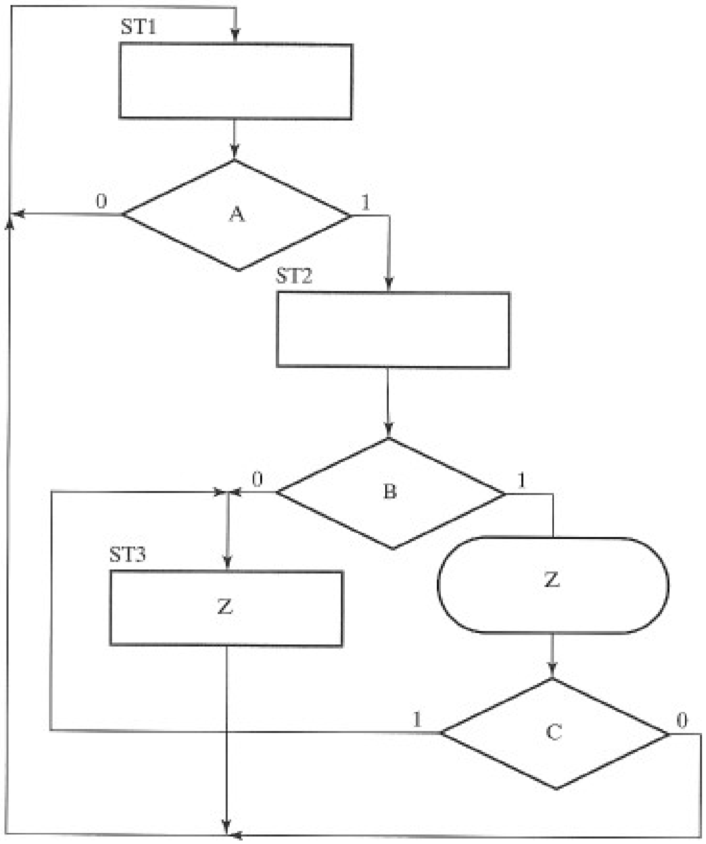
A:	0	1	1	0	1	1	0	1
B:	1	1	0	1	0	1	0	1
C:	0	1	0	1	0	1	0	1
State: ST1								
Z:								

8–3. An ASM chart is given in Figure 8-19. Find the state table for the corresponding sequential circuit.

8–4. Find the ASM chart corresponding to the following description: There are two states, *A* and *B*. If in state *A* and input *X* is 1, then the next state is *A*. If in state *A* and input *X* is 0, then the next state is *B*. If in state *B* and input *Y* is 0, then the next state is *B*. If in state *B* and input *Y* is 1, then the next state is *A*. Output *Z* is equal to 1 while the circuit is in state *B*.



□ **FIGURE 8-18**
State Diagram for Problem 8-1



□ **FIGURE 8-19**
ASM Chart for Problem 8-2 and Problem 8-3

8–5. *Find the ASM for a circuit that detects a difference in value in an input signal X at two successive positive clock edges. If X has different values at two successive positive clock edges, then output Z is equal to 1 for the next clock cycle. Otherwise, output Z is 0.

8–6. +The ASM chart for a synchronous circuit with clock CK for a washing machine is to be developed. The circuit has three external inputs, START, FULL, and EMPTY (which are 1 for at most a single clock cycle and are mutually exclusive), and external outputs, HOT, COLD, DRAIN, and TURN. The datapath for the control consists of a down-counter, which has three inputs, RESET, DEC, and LOAD. This counter synchronously decrements once each minute for DEC = 1, but can be loaded or synchronously reset on any cycle of clock CK. It has a single output, ZERO, which is 1 whenever the counter contains value zero and is 0 otherwise.

In its operation, the circuit goes through four distinct cycles, WASH, SPIN, RINSE, and SPIN, which are detailed as follows:

WASH: Assume that the circuit is in its power-up state IDLE. If START is 1 for a clock cycle, HOT becomes 1 and remains 1 until FULL = 1, filling the washer with hot water. Next, using LOAD, the down-counter is loaded with a value from a panel dial which indicates how many minutes the wash cycle is to last. DEC and TURN then become 1 and the washer washes its contents. When ZERO becomes 1, the wash is complete, and TURN and DEC become 0.

SPIN: Next, DRAIN becomes 1, draining the wash water. When EMPTY becomes 1, the down-counter is loaded with 7. DEC and TURN then become 1 and the remaining wash water is wrung from the contents. When ZERO becomes 1, DRAIN, DEC, and TURN return to 0.

RINSE: Next, COLD becomes 1 and remains 1 until FULL = 1, filling the washer with cold rinse water. Next, using LOAD, the down-counter is loaded with value 10. DEC and TURN then become 1 and the washer rinses its contents. When ZERO becomes 1, the rinse is complete, and TURN and DEC become 0.

SPIN: Next, DRAIN becomes 1, draining the rinse water. When EMPTY becomes 1, the down-counter is loaded with 8. DEC and TURN then become 1 and the remaining rinse water is wrung from the contents. When ZERO becomes 1, DRAIN, DEC, and TURN return to 0 and the circuit returns to state IDLE.

(a) Find the ASM chart for the washer circuit.

(b) Modify your design in part (a) assuming that there are two more inputs, PAUSE and STOP. PAUSE causes the circuit, including the counter, to halt and all outputs to go to 0. When START is pushed, the washer resumes operation at the point it paused. When STOP is pushed, all outputs are reset to 0 except for DRAIN which is set to 1. When EMPTY becomes 1, the state returns to IDLE.

8–7. Find an ASM chart for a traffic light controller that works as follows: A timing signal T is the input to the controller. T defines the yellow light interval, as well as the changes of the red and green lights. The outputs to the signals are defined by the following table:

Output	Light Controlled
GN	Green Light, North/South Signal
YN	Yellow Light, North/South Signal
RN	Red Light, North/South Signal
GE	Green Light, East/West Signal
YE	Yellow Light, East/West Signal
RE	Red Light, East/West Signal

While $T = 0$, the green light is on for one signal and the red light for the other. With $T = 1$, the yellow light is on for the signal that was previously green, and the signal that was previously red remains red. When T becomes 0, the signal that was previously yellow becomes red, and the signal that was previously red becomes green. This pattern of alternating changes in color continues. Assume that the controller is synchronous with a clock that changes much more frequently than input T.

8–8. *Implement the ASM chart in Figure 8-19 by using one flip-flop per state.

8–9. *Implement the ASM chart in Figure 8-19 by using a sequence register and decoder.

8–10. +Implement the ASM chart derived in Problem 8–6(a) by using one flip-flop per state.

8–11. *Multiply the two unsigned binary numbers 100110 (multiplicand) and 110101 (multiplier) by using both the hand method and the hardware method.

8–12. Manually simulate the process of multiplying the two unsigned binary numbers 1010 (multiplicand) and 1011 (multiplier). List the contents of registers A, Q, P, and C and the control state, using the system in Figure 8-6 with n equal to 4 and with the hardwired control in Figure 8-12.

8–13. Determine the time it takes to process the multiplication operation in the digital system described in Figure 8-6 and Figure 8-9. Assume that the Q register has n bits and the interval for a clock cycle is f nanoseconds.

8–14. Prove that the multiplication of two n-bit numbers gives a product of no more than $2n$ bits. Show that this condition implies that no overflow can occur in the final result in the multiplier circuit defined in Figure 8-6.

8–15. Consider the block diagram of the multiplier shown in Figure 8-6. Assume that the multiplier and multiplicand consist of 16 bits each.

(a) How many bits can be expected in the product, and where is it available?

(b) How many bits are in the P counter, and what is the binary number that must be loaded into it initially?

(c) Design the combinational circuit that checks for zero in the P counter.

8–16. *Design a digital system with three 16-bit registers AR, BR, and CR and 16-bit data input IN to perform the following operations, assuming a two's complement representation and ignoring overflow:

(a) Transfer two 16-bit signed numbers to AR and BR on successive clock cycles after a go signal G becomes 1.

(b) If the number in AR is positive but nonzero, multiply the contents of BR by two and transfer the result to register CR.

(c) If the number in AR is negative, multiply the contents of AR by two and transfer the result to register CR.

(d) If the number in AR is zero, reset register CR to 0.

8–17. +Modify the multiplier design in Figure 8-6 and the ASM chart in Figure 8-7 to perform 2's complement signed-number multiplication using Booth's algorithm, which employs an adder–subtractor. The decision to add, to subtract, or to do nothing is made on the basis of the current least significant bit (LSB) in the *Q* register and on the previous LSB bit from the *Q* register before *Q* was shifted right. Thus, a flip-flop must be provided to store the previous LSB from the *Q* register. The initial value of the previous least significant bit is to be 0. The following table defines the decisions:

LSB of *Q*	Previous LSB of *Q*	Action
0	0	Leave partial product unchanged
0	1	Add multiplicand to partial product
1	0	Subtract multiplicand from partial product
1	1	Leave partial product unchanged

8–18. +Design a digital system that multiplies two unsigned binary numbers by the repeated addition method. For example, to multiply 5 by 4, the digital system adds the multiplicand four times: 5 + 5 + 5 + 5 = 20. Let the multiplicand be in register *BR*, the multiplier in register *AR*, and the product in register *PR*. An adder circuit adds the contents of *BR* to *PR*, and AR is a down-counter. A zero-detection circuit *Z* checks when *AR* becomes zero after each time that it is decremented. Design the control by the flip-flop per state method.

8–19. *Write, compile, and simulate a VHDL description for the ASM shown in Figure 8-19. Use a simulation input that passes through all paths in the ASM chart, and include both the state and output *Z* as simulation outputs. Correct and resimulate your design if necessary.

8–20. *Write, compile, and simulate a Verilog description for the ASM in Figure 8-19. Use code 00 for state ST1, 01 for state ST2, and 10 for state ST3. Use a simulation input that passes through all paths in the ASM chart, and include both the state and *Z* as simulation outputs. Correct and resimulate your design if necessary.

8–21. Perform the design in Problem 8-5 using Verilog instead of an ASM chart. Use state names S0, S1, S2, ... , and codes that are the binary equivalent of the integer in the state name. Compile and simulate your design using a simulation input that thoroughly validates the design and that provides both state and Z as simulation outputs. Correct and resimulate your design if necessary.

8–22. +Perform the design in Problem 8-7 using VHDL instead of an ASM chart. Compile and simulate your design by running the traffic light through two full cycles. Use realistic intervals for T and a slow clock. Adjust the clock intervals if necessary to avoid long simulation times.

8–23. +Perform the design in Problem 8-7 using Verilog instead of an ASM chart. Compile and simulate your design by running the traffic light through two full cycles. Use the state assignment method in Problem 8-21. Use realistic intervals for T and a slow clock. Adjust the clock intervals if necessary to avoid long simulation times.

CHAPTER 9

MEMORY BASICS

Memory is a major component of a digital computer and is present in a large proportion of all digital systems. Random-access memory (RAM) stores data temporarily, and read-only memory (ROM) stores data permanently. ROM is one form of a variety of components called programmable logic devices (PLDs) that use stored information to define logic circuits.

Our study of RAM begins by looking at it in terms of a model with inputs, outputs, and signal timing. We then use equivalent logical models to understand the internal workings of RAM chips. Both static RAM and dynamic RAM are considered. The various types of dynamic RAM used for movement of data at high speeds between the CPU and memory are surveyed. Finally, we put RAM chips together to build simple RAM systems.

In many of the previous chapters, the concepts presented were broad, pertaining to much of the generic computer at the beginning of Chapter 1. In this chapter, for the first time, we can be more precise and point to specific uses of memory and related components. Beginning with the processor, the internal cache is largely very fast RAM. Outside the CPU, the external cache is largely fast RAM. The RAM subsystem, by its very name, is a type of memory. In the I/O area, we find substantial memory for storing information about the screen image in the video adapter. RAM appears in disk cache in the disk controller, to speed up disk access. Aside from the highly central role of the RAM subsystem in storing data and programs, we find memory in various forms applied in most subsystems of the generic computer.

9-1 MEMORY DEFINITIONS

In digital systems, memory is a collection of cells capable of storing binary information. In addition to these cells, memory contains electronic circuits for storing and retrieving the information. As indicated in the discussion of the generic computer, memory is used in many different parts of a modern computer, providing temporary or permanent storage for substantial amounts of binary information. In order for this information to be processed, it is sent from the memory to processing hardware consisting of registers and combinational logic. The processed information is then returned to the same or to a different memory. Input and output devices also interact with memory. Information from an input device is placed in memory so that it can be used in processing. Output information from processing is placed in memory, and from there it is sent to an output device.

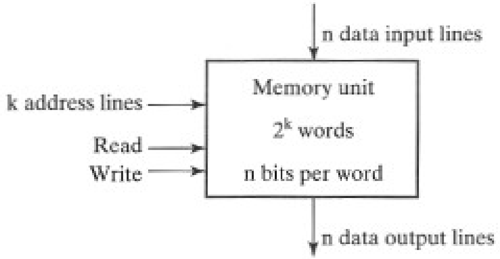
Two types of memories are used in various parts of a computer: *random-access memory* (RAM) and *read-only memory* (ROM). RAM accepts new information for storage to be available later for use. The process of storing new information in memory is referred to as a memory *write* operation. The process of transferring the stored information out of memory is referred to as a memory *read* operation. RAM can perform both the write and the read operations, whereas ROM as introduced in Chapter 3, performs only read operations. RAM sizes may range from hundreds to billions of bits.

9-2 Random-Access Memory

Memory is a collection of binary storage cells together with associated circuits needed to transfer information into and out of the cells. Memory cells can be accessed to transfer information to or from any desired location, with the access taking the same time regardless of the location, hence, the name *random-access memory.* In contrast, *serial memory,* such as is exhibited by a magnetic disk or tape unit, takes different lengths of time to access information, depending on where the desired location is relative to the current physical position of the disk or tape.

Binary information is stored in memory in groups of bits, each group of which is called a *word*. A word is an entity of bits that moves in and out of memory as a unit—a group of 1's and 0's that represents a number, an instruction, one or more alphanumeric characters, or other binary-coded information. A group of eight bits is called a *byte.* Most computer memories use words that are multiples of eight bits in length. Thus, a 16-bit word contains two bytes, and a 32-bit word is made up of four bytes. The capacity of a memory unit is usually stated as the total number of bytes that it can store. Communication between a memory and its environment is achieved through data input and output lines, address selection lines, and control lines that specify the direction of transfer of information. A block diagram of a memory is shown in Figure 9-1. The n data input lines provide the information to be stored in memory, and the n data output lines supply the information coming out of memory. The k address lines specify the particular word chosen among the many available. The two control inputs specify the direction of transfer desired: the Write input causes binary data to be transferred into memory, and the Read input causes binary data to be transferred out of memory.

The memory unit is specified by the number of words it contains and the number of bits in each word. The address lines select one particular word. Each word in memory is assigned an identification number called an *address.* Addresses



□ **FIGURE 9-1**
Block Diagram of Memory

range from 0 to $2^k - 1$, where k is the number of address lines. The selection of a specific word inside memory is done by applying the k-bit binary address to the address lines. A decoder accepts this address and opens the paths needed to select the word specified. Computer memory varies greatly in size. It is customary to refer to the number of words (or bytes) in memory with one of the letters K (kilo), M (mega), or G (giga). K is equal to 2^{10}, M is equal to 2^{20}, and G is equal to 2^{30}. Thus, 64K = 2^{16}, 2M = 2^{21}, and 4G = 2^{32}.

Consider, for example, a memory with a capacity of 1K words of 16 bits each. Since 1K = 1024 = 2^{10}, and 16 bits constitute two bytes, we can say that the memory can accommodate 2048, or 2K, bytes. Figure 9-2 shows the possible contents of the first three and the last three words of this size of memory. Each word contains 16 bits that can be divided into two bytes. The words are recognized by their decimal addresses from 0 to 1023. An equivalent binary address consists of 10 bits. The first address is specified using ten 0's, and the last address is specified with ten 1's. This is because 1023 in binary is equal to 1111111111. A word in memory is selected by its binary address. When a word is read or written, the memory operates on all 16 bits as a single unit.

Memory address		
Binary	Decimal	Memory contents
0000000000	0	10110101 01011100
0000000001	1	10101011 10001001
0000000010	2	00001101 01000110
	⋮	⋮
1111111101	1021	10011101 00010101
1111111110	1022	00001101 00011110
1111111111	1023	11011110 00100100

□ **FIGURE 9-2**
Contents of a 1024 × 16 Memory

The 1K × 16 memory of the figure has 10 bits in the address and 16 bits in each word. If, instead, we have a 64K × 10 memory, it is necessary to include 16 bits in the address, and each word will consist of 10 bits. The number of address bits needed in memory is dependent on the total number of words that can be stored there and is independent of the number of bits in each word. The number of bits in the address for a word is determined from the relationship $2^k \geq m$, where m is the total number of words and k is the minimum number of address bits satisfying the relationship.

Write and Read Operations

The two operations that a random-access memory can perform are write and read. A *write* is a transfer into memory of a new word to be stored. A *read* is a transfer of a copy of a stored word out of memory. A Write signal specifies the transfer-in operation, and a Read signal specifies the transfer-out operation. On accepting one of these control signals, the internal circuits inside memory provide the desired function.

The steps that must be taken for a write are as follows:

1. Apply the binary address of the desired word to the address lines.
2. Apply the data bits that must be stored in memory to the data input lines.
3. Activate the Write input.

The memory unit will then take the bits from the data input lines and store them in the word specified by the address lines.

The steps that must be taken for a read are as follows:

1. Apply the binary address of the desired word to the address lines.
2. Activate the Read input.

The memory will then take the bits from the word that has been selected by the address and apply them to the data output lines. The contents of the selected word are not changed by reading them.

Memory is made up of RAM integrated circuits (chips), plus additional logic circuits. RAM chips usually provide the two control inputs for the read and write operations in a somewhat different configuration from that just described. Instead of having separate Read and Write inputs to control the two operations, most integrated circuits provide at least a Chip Select that selects the chip to be read from or written to and a Read/$\overline{\text{Write}}$ that determines the particular operation. The memory operations that result from these control inputs are shown in Table 9-1.

The Chip Select is used to enable the particular RAM chip or chips containing the word to be accessed. When Chip Select is inactive, the memory chip or chips are not selected, and no operation is performed. When Chip Select is active, the Read/$\overline{\text{Write}}$ input determines the operation to be performed. While Chip Select accesses chips, a signal is also provided that accesses the entire memory. We will call this signal the Memory Enable.

Timing Waveforms

The operation of the memory unit is controlled by an external device, such as a CPU. The CPU is synchronized by its own clock pulses. The memory, however,

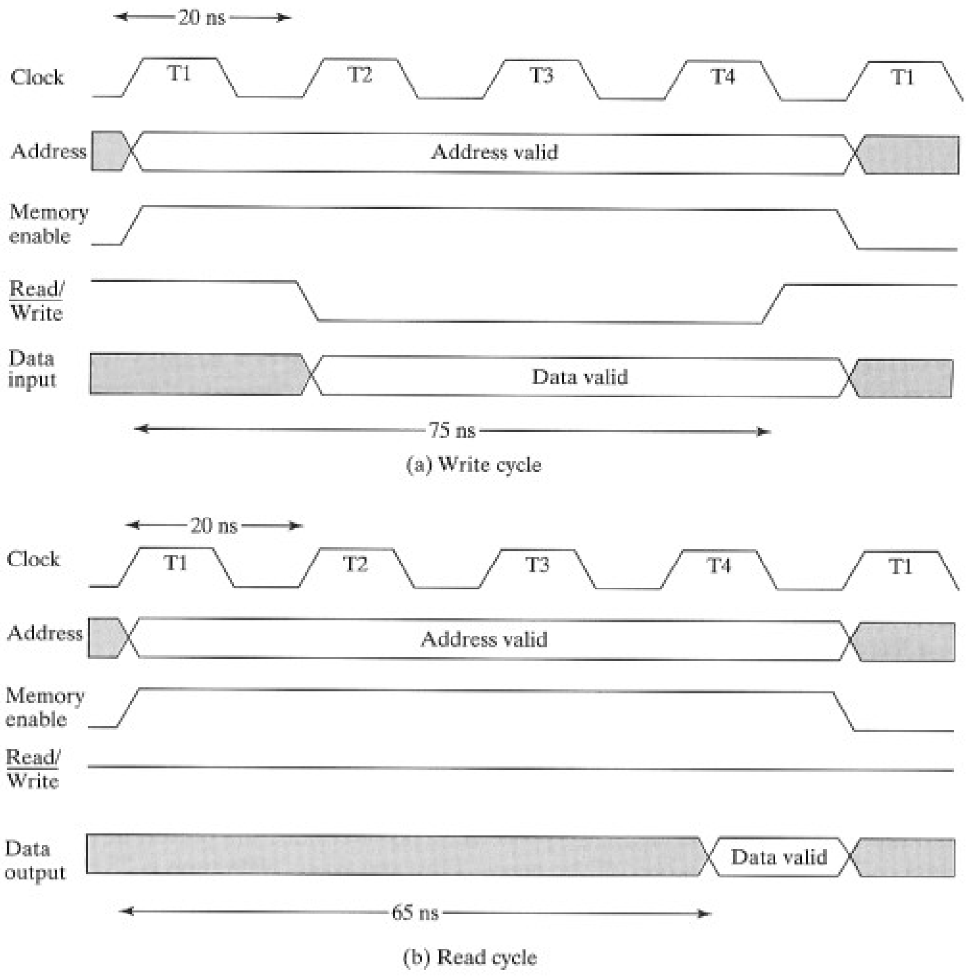
□ **TABLE 9-1**
Control Inputs to a Memory Chip

Chip select CS	Read/$\overline{\text{Write}}$ R/$\overline{\text{W}}$	Memory operation
0	×	None
1	0	Write to selected word
1	1	Read from selected word

does not employ the CPU clock. Instead, its read and write operations are timed by changes in values on the control inputs. The *access time* of a memory read operation is the maximum time from the application of the address to the appearance of the data at the Data Output. Similarly, the *write cycle time* is the maximum time from the application of the address to the completion of all internal memory operations required to store a word. Memory writes may be performed one after the other at the intervals of the cycle time. The CPU must provide the memory control signals in such a way as to synchronize its own internal clocked operations with the read and write operations of memory. This means that the access time and the write cycle time of the memory must be related within the CPU to a period equal to a fixed number of CPU clock pulse periods.

Assume, as an example, that a CPU operates with a clock frequency of 50 MHz, giving a period of 20 ns (1 ns = 10^{-9} s) for one clock pulse. Suppose now that the CPU communicates with a memory with an access time of 65 ns and a write cycle time of 75 ns. The number of clock pulses required for a memory request is the integer value greater than or equal to the larger of the access time and the write cycle time, divided by the clock period. Since the period of the CPU clock is 20 ns, and the larger of the access time and write cycle time is 75 ns, it will be necessary to devote at least four clock pulses to each memory request.

The memory cycle timing shown in Figure 9-3 is for a CPU with a 50 MHz clock and memory with a 75-ns write cycle time and a 65-ns access time. The write cycle in part (a) shows four pulses *T*1, *T*2, *T*3, and *T*4 with a cycle of 20 ns. For a write operation, the CPU must provide the address and input data to the memory. The address is applied, and Memory Enable is set to the high level at the positive edge of the *T*1 pulse. The data, needed somewhat later in the write cycle, is applied at the positive edge of *T*2. The two lines that cross each other in the address and data waveforms designate a possible change in value of the multiple lines. The shaded areas represent unspecified values. A change of the Read/$\overline{\text{Write}}$ signal to 0 to designate the write operation is also at the positive edge of *T*2. To avoid destroying data in other memory words, it is important that this change occur after the signals on the address lines have become fixed at the desired values. Otherwise, one or more other words might be momentarily addressed and accidentally written over with different data. The Read/$\overline{\text{Write}}$ signal must stay at 0 long enough after application of the address and Memory Enable to allow the write operation to complete. Finally, the address and data signals must remain stable for a short time after the Read/$\overline{\text{Write}}$ goes to 1, again to avoid destroying data in other memory



□ **FIGURE 9-3**
Memory Cycle Timing Waveforms

words. At the completion of the fourth clock pulse, the memory write operation has ended with 5 ns to spare, and the CPU can apply the address and control signals for another memory request with the next *T*1 pulse.

The read cycle shown in Figure 9-3(b) has an address for the memory that is provided by the CPU. The CPU applies the address, sets the Memory Enable to 1, and sets Read/$\overline{\text{Write}}$ to 1 to designate a read operation, all at the positive edge of *T*1. The memory places the data of the word selected by the address onto the data output lines within 65 ns from the time that the address is applied and the memory enable is activated. Then, the CPU transfers the data into one of its internal registers during the positive transition of the next *T*1 pulse, which can also change the address and controls for the next memory request.

Properties of Memory

Integrated circuit RAM may be either static or dynamic. *Static* RAM (SRAM) consists of internal latches that store the binary information. The stored information remains valid as long as power is applied to the RAM. *Dynamic* RAM (DRAM) stores the binary information in the form of electric charges on capacitors. The capacitors are accessed inside the chip by n-channel MOS transistors. The stored charge on the capacitors tends to discharge with time, and the capacitors must be periodically recharged by *refreshing* the DRAM. This is done by cycling through the words every few milliseconds, reading and rewriting them to restore the decaying charge. DRAM offers reduced power consumption and larger storage capacity in a single memory chip, but SRAM is easier to use and has shorter read and write cycles. Also, no refresh is required for SRAM.

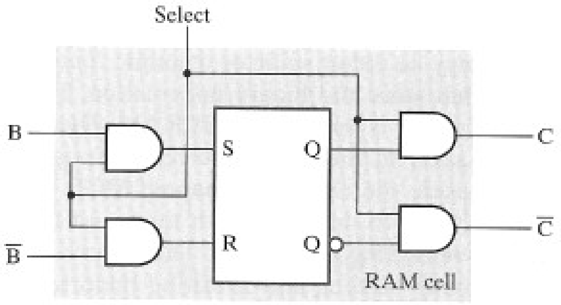
Memory units that lose stored information when power is turned off are said to be *volatile*. Integrated circuit RAMs, both static and dynamic, are of this category, since the binary cells need external power to maintain the stored information. In contrast, a *nonvolatile memory*, such as magnetic disk, retains its stored information after the removal of power. This is because the data stored on magnetic components is represented by the direction of magnetization, which is retained after power is turned off. Another nonvolatile memory is ROM, discussed in Section 3-9.

9-3 SRAM Integrated Circuits

As indicated earlier, memory consists of RAM chips plus additional logic. We will consider the internal structure of the RAM chip first. Then we will study combinations of RAM chips and additional logic used to construct memory. The internal structure of a RAM chip of m words with n bits per word consists of an array of mn binary storage cells and associated circuitry. The circuity is made up of decoders to select the word to be read or written, read circuits, write circuits, and output logic. The *RAM cell* is the basic binary storage cell used in the RAM chip, which is typically designed as an electronic circuit rather than a logic circuit. Nevertheless, it is possible and convenient to model the RAM chip using a logic model.

A static RAM chip serves as the basis for our discussion. We first present RAM cell logic for storing a single bit and then use the cell in a hierarchy to describe the RAM chip. Figure 9-4 shows the logic model of the RAM cell. The storage part of the cell is modeled by an SR latch. The inputs to the latch are enabled by a Select signal. For Select equal to 0, the stored content is held. For Select equal to 1, the stored content is determined by the values on B and $\overline{B}$. The outputs from the latch are gated by Select to produce cell outputs C and $\overline{C}$. For Select equal to 0, both C and $\overline{C}$ are 0, and for Select equal to 1, C is the stored value and $\overline{C}$ is its complement.

To obtain simplified static RAM diagrams, we interconnect a set of RAM cells and read and write circuits to form a *RAM bit slice* that contains all of the circuitry associated with a single bit position of a set of RAM words. The logic diagram for a RAM bit slice is shown in Figure 9-5(a). The portion of the model representing each RAM cell is highlighted in blue. The loading of a cell latch is



□ **FIGURE 9-4**
Static RAM Cell

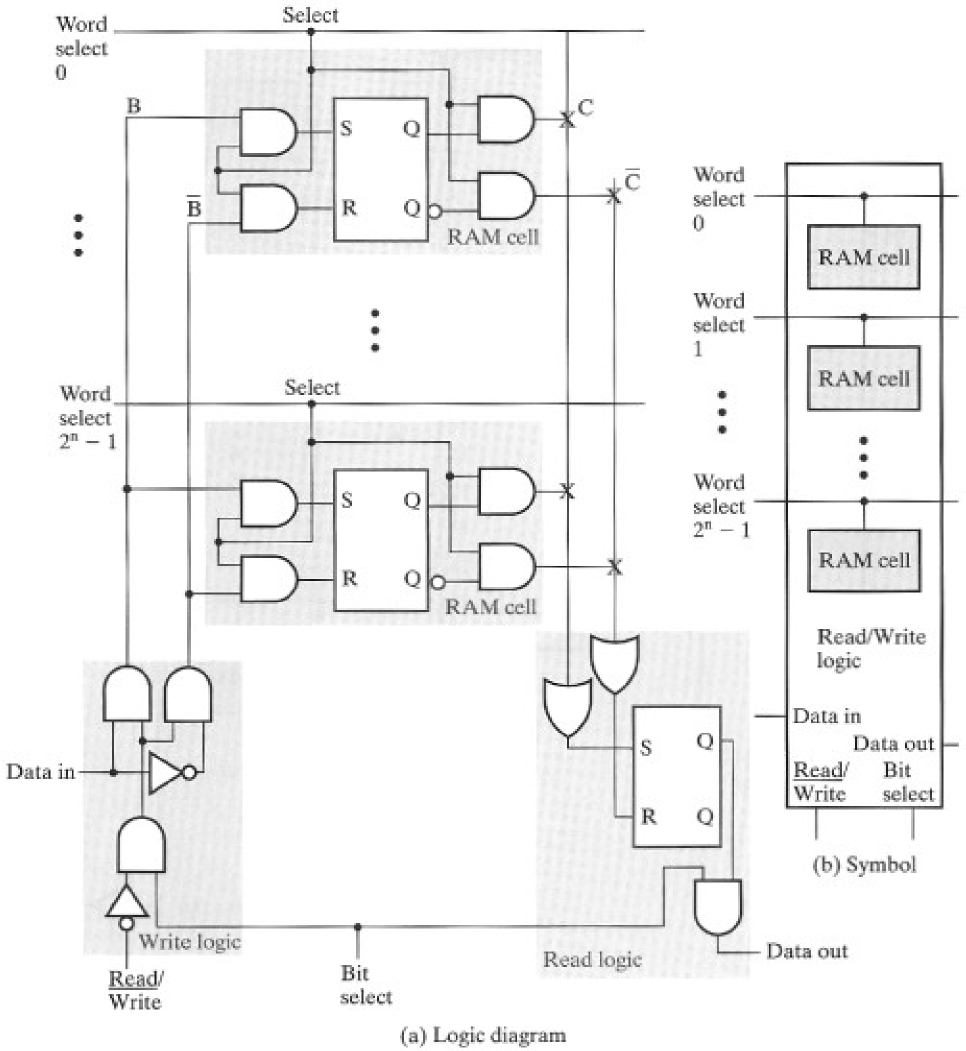
now controlled by a Word Select input. If this is 0, then both S and R are 0, and the cell latch contents remain unchanged. If the Word Select input is 1, then the value to be loaded into the latch is controlled by two signals B and $\overline{B}$ from the Write Logic. In order for either of these signals to be 1 and potentially change the stored value, Read/$\overline{\text{Write}}$ must be 0 and Bit Select must be 1. Then the Data In value and its complement are applied to B and $\overline{B}$, respectively, to set or reset the latch in the RAM cell selected. If Data In is 1 the latch is set to 1, and if Data In is 0 the latch is reset to 0, completing the write operation.

Only one word is written at a time. That is, only one Word Select line is 1, and all other Word Select lines are 0. Thus, only one RAM cell attached to B and $\overline{B}$ is written. The Word Select also controls the reading of the RAM cells, using shared Read Logic. If Word Select is 0, then the stored value in the SR latch is prevented by the AND gates from reaching the pair of OR gates in the Read Logic. But if Word Select is 1, the stored value passes through to the OR gates and is captured in the Read Logic SR latch. If Bit Select is also 1, the captured value appears on the Data Out line of the RAM bit slice. Note that for this particular Read Logic design, the read occurs regardless of the value of Read/$\overline{\text{Write}}$.

The symbol for the RAM bit slice given in Figure 9-5(b) is used to represent the internal structure of RAM chips. Each Word Select line extends beyond the bit slice, so that when multiple RAM bit slices are placed side by side, corresponding Word Select lines connect. The other signals in the lower portion of the symbol may be connected in various ways, depending on the structure of the RAM chip.

The symbol and block diagram for a 16 × 1 RAM chip are shown in Figure 9-6. Both have four address inputs for the 16 one-bit words stored in RAM. There are also Data Input, Data Output, and Read/$\overline{\text{Write}}$ signals. The Chip Select at the chip level corresponds to the Memory Enable at the level of a RAM consisting of multiple chips. The internal structure of the RAM chip consists of a RAM bit slice having 16 RAM cells. Since there are 16 Word Select lines to be controlled such that one and only one has the value logic 1 at a given time, a 4-to-16-line decoder is used to decode the four address bits into 16 Word Select bits.

The only additional logic in the figure is a triangular symbol with one normal input, one normal output, and a second input on the bottom of the symbol. This symbol is a three-state buffer that allows construction of a multiplexer with an

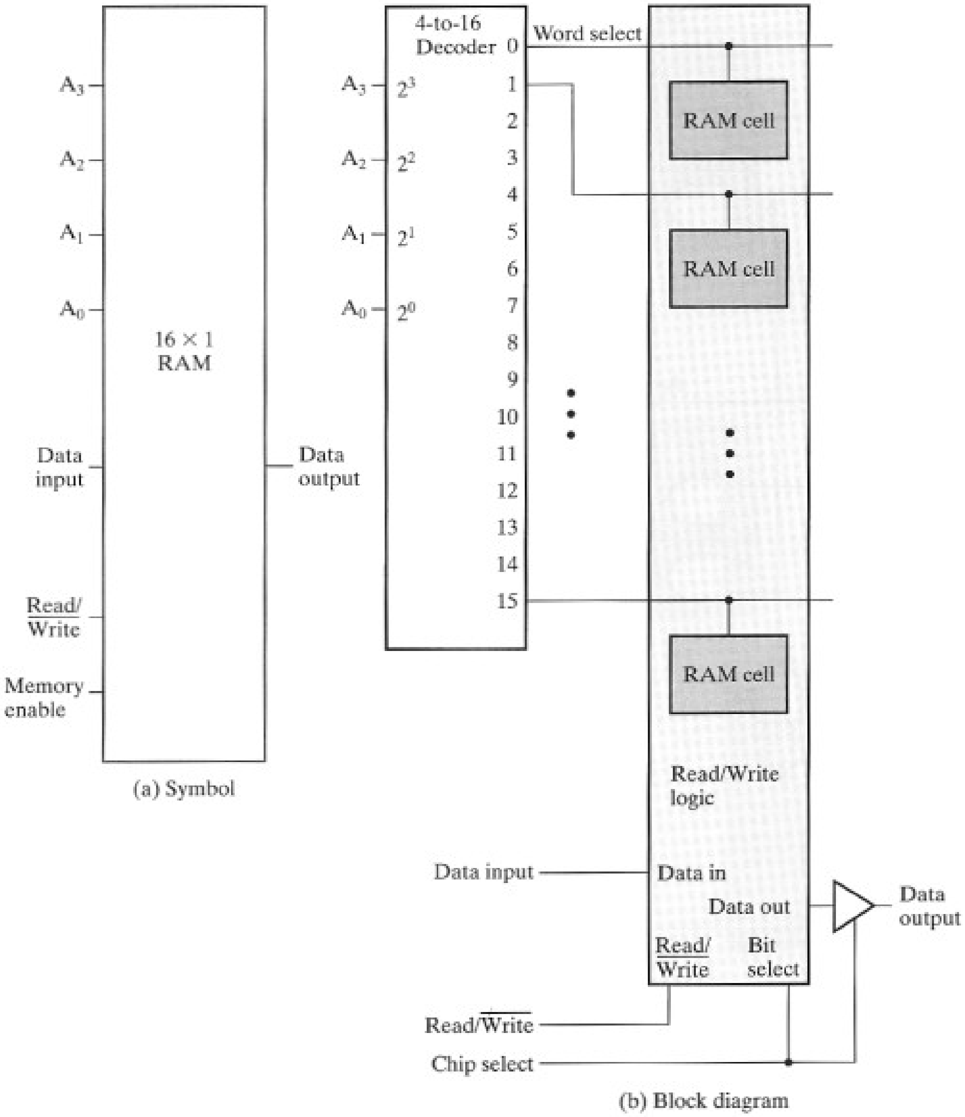


(a) Logic diagram

(b) Symbol

□ **FIGURE 9-5**
RAM Bit Slice Model

arbitrary number of inputs. Three-state outputs are connected together and properly controlled using the Chip Select inputs. By using three-state buffers on the outputs of RAM chips, these outputs can be connected together to provide the word from the chip being read on the bit lines attached to the RAM outputs. The enable signals in the preceding discussion correspond to the Chip Select inputs on the RAM chips. To read a word from a particular RAM chip, the Chip Select value for that chip must be 1, and for all other chips attached to the same output bit lines, the Chip Select must be 0. These combinations containing a single 1 can be obtained from a decoder.



□ **FIGURE 9-6**
16-Word by 1-Bit RAM Chip

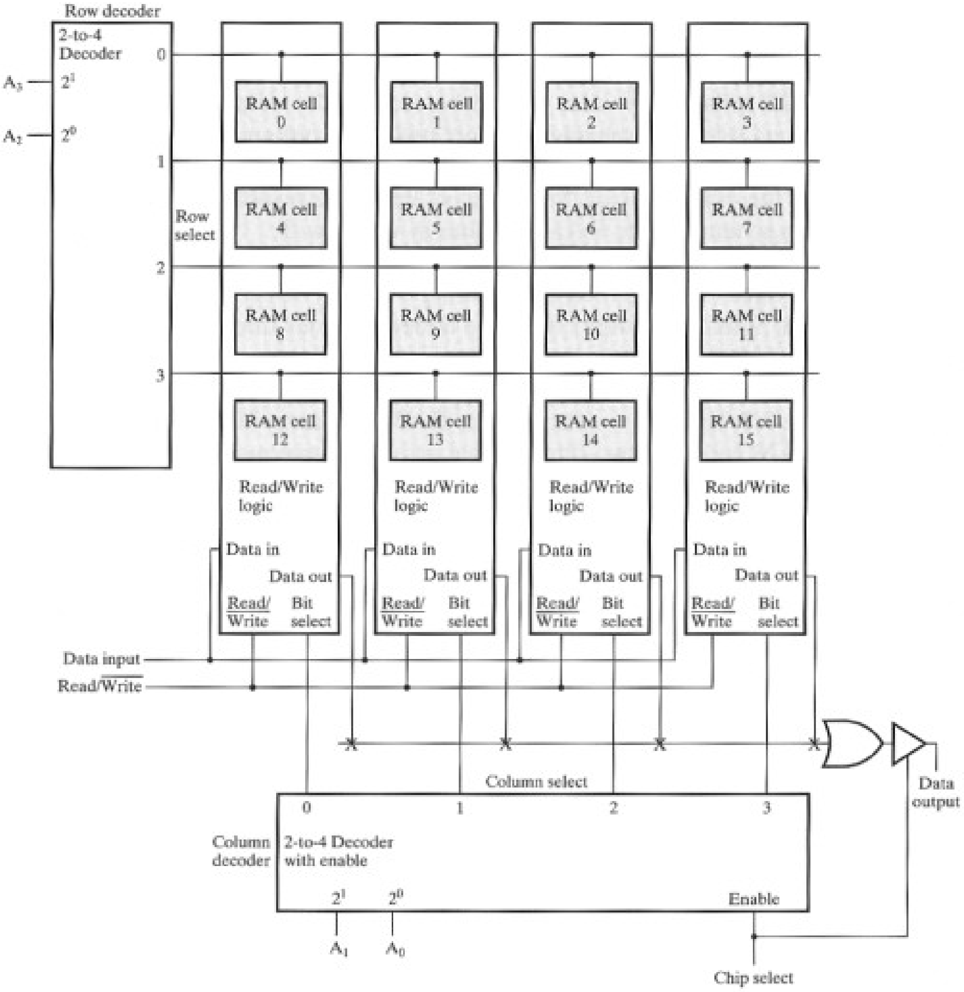
Coincident Selection

Inside a RAM chip, the decoder with k inputs and 2^k outputs requires 2^k AND gates with k inputs per gate if a straightforward design approach is used. In addition, if the number of words is large, and all bits for one bit position in the word are contained in a single RAM bit slice, the number of RAM cells sharing the read and write circuits is also large. The electrical properties resulting from both of these situations cause the access and write cycle times of the RAM to become long, which is undesirable.

The total number of decoder gates, the number of inputs per gate, and the number of RAM cells per bit slice can all be reduced by employing two decoders with a *coincident selection* scheme. In one possible configuration, two $k/2$-input decoders are used instead of one k-input decoder. One decoder controls the word select lines and the other controls the bit select lines. The result is a two-dimensional matrix selection scheme. If the RAM chip has m words with 1 bit per word, then the scheme selects the RAM cell at the intersection of the Word Select row and the Bit Select column. Since the Word Select is no longer strictly selecting words, its name is changed to *Row Select*. An output from the added decoder that selects one or more bit slices is referred to as a *Column Select*.

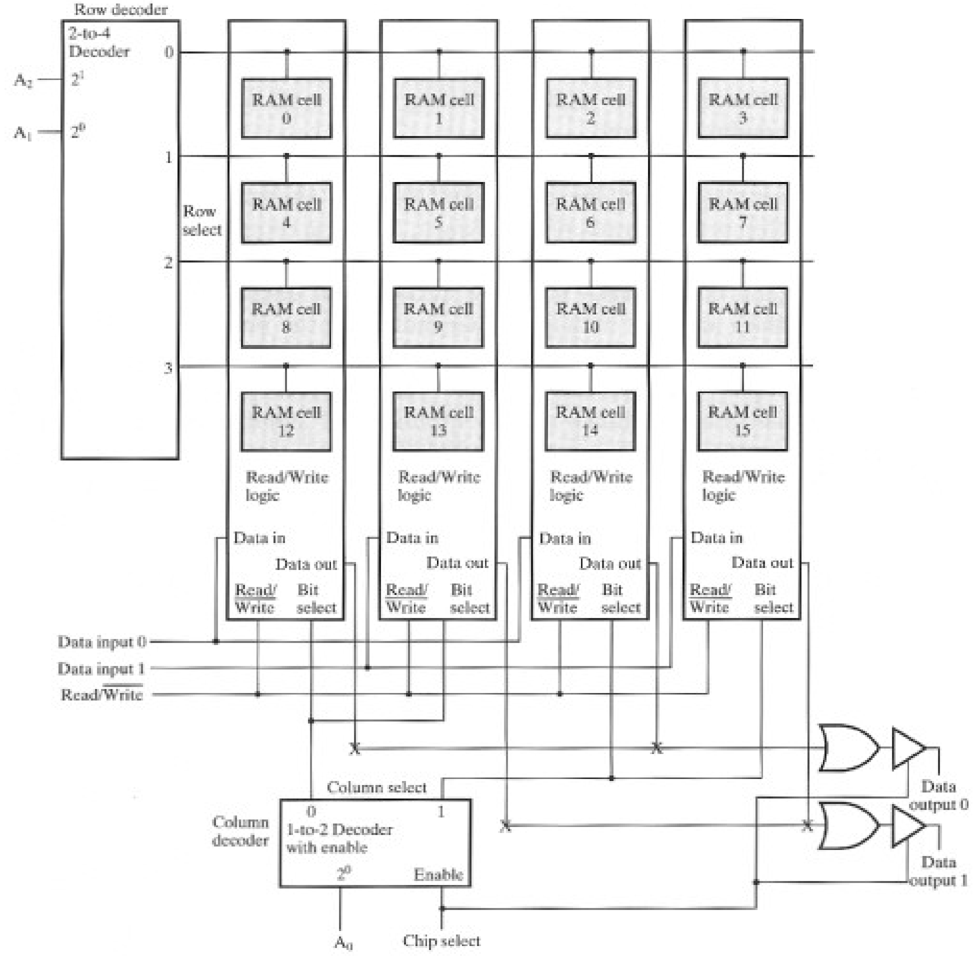
Coincident selection is illustrated for the 16×1 RAM chip with the structure shown in Figure 9-7. The chip consists of four RAM bit slices of four bits each and has a total of 16 RAM cells in a two-dimensional array. The two most significant address inputs go through the 2-to-4-line row decoder to select one of the four rows of the array. The two least significant address inputs go through the 2-to-4-line column decoder to select one of the four columns (RAM bit slices) of the array. The column decoder is enabled with the Chip Select input. When the Chip Select is 0, all outputs of the decoder are 0 and none of the cells is selected. This prevents writing into any RAM cell in the array. With Chip Select at 1, a single bit in the RAM is accessed. For example, for the address 1001, the first two address bits are decoded to select row 10 (2_{10}) of the RAM cell array. The second two address bits are decoded to select column $01(1_{10})$ of the array. The RAM cell accessed, in row 2 and column 1 of the array, is cell 9 $(10_2\ 01_2)$. With a row and column selected, the $\text{Read}/\overline{\text{Write}}$ input determines the operation. During the read operation ($\text{Read}/\overline{\text{Write}} = 1$), the selected bit of the selected row goes through the OR gate to the three-state buffer. Note that the gate is drawn according to the array logic established in Figure 3-22. Since the buffer is enabled by Chip Select, the value read appears at the Data Output. During the write operation ($\text{Read}/\overline{\text{Write}} = 0$), the bit available on the Data Input line is transferred into the selected RAM cell. Those RAM cells not selected are disabled, and their previous binary values remain unchanged.

The same RAM cell array is used in Figure 9-8 to produce an 8×2 RAM chip (eight words of two bits each). The row decoding is unchanged from that in Figure 9-7; the only changes are in the column and output logic. Since there are just three address bits, and two are handled by the row decoder, the column decoder has only one address bit and Chip Select as inputs and produces just two Column Select lines. Since two bits at a time are to be written or read, the Column Select lines go to adjacent pairs of RAM bit slices. Two input lines, Data Input 0 and Data Input 1, each go to a different bit in all of the pairs. Finally, corresponding bits of the pairs share output OR gates and three-state buffers, giving output lines Data Output 0 and Data Output 1. The operation of this structure can be illustrated by the application of the address 3 (011_2). The first two bits of the address, 01, access row 1 of the array. The final bit, 1, accesses column 1, which consists of bit slices 2 (10_2) and 3 (11_2). So the word to be written or read lies in RAM cells 6 and 7 $(011\ 0_2$ and $011\ 1_2)$, which contain bits 0 and 1, respectively, of word 3.



□ **FIGURE 9-7**
Diagram of a 16 × 1 RAM Using a 4 × 4 RAM Cell Array

We can demonstrate the savings of the coincident selection scheme by considering a more realistic static RAM size, 32K × 8. This RAM chip contains a total of 256K bits. To make the number of rows and columns in the array equal, we take the square root of 256K, giving $512 = 2^9$. So the first nine bits of the address are fed to the row decoder and the remaining six bits to the column decoder. Without coincident selection, the single decoder would have 15 inputs and 32,768 outputs. With coincident selection, there is one 9-to-512-line decoder and one 6-to-64-line decoder. The number of gates for a straightforward design of the single decoder would be 32,800. For the two coincident decoders, the number of gates is 608, reducing the gate count by a factor of more than 50. In addition, although it appears that there are 64 times as many Read/Write circuits, the column selection



□ **FIGURE 9-8**
Block Diagram of an 8 × 2 RAM Using a 4 × 4 RAM Cell Array

can be done between the RAM cells and the Read/Write circuits, so that only the original eight circuits are required. Because of the reduced number of RAM cells attached to each Read/Write circuit at any time, the access time of the chip is also improved.

9-4 Array of SRAM ICs

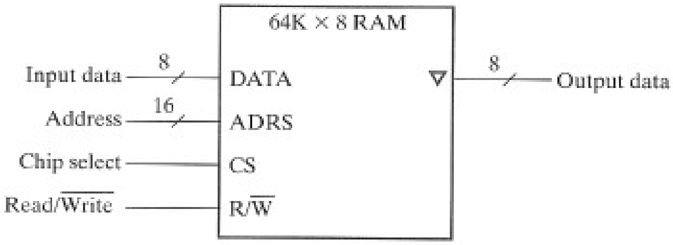
Integrated circuit RAM chips are available in a variety of sizes. If the memory unit needed for an application is larger than the capacity of one chip, it is necessary to combine a number of chips in an array to form the required size of memory. The capacity of the memory depends on two parameters: the number of words and the number of bits per word. An increase in the number of words requires that we increase the address length. Every bit added to the length of the address doubles the number of words in memory. An increase in the number of bits per word requires that we increase the number of data input and output lines, but the address length remains the same.

To illustrate an array of RAM ICs, let us first introduce a RAM chip using the condensed representation for inputs and outputs shown in Figure 9-9. The capacity of this chip is 64K words of 8 bits each. The chip requires a 16-bit address and 8 input and output lines. Instead of 16 lines for the address and 8 lines each for data input and data output, each is shown in the block diagram by a single line. Each line has a slash across it with a number indicating the number of lines represented. The *CS* (Chip Select) input selects the particular RAM chip, and the $R/\overline{W}$ (Read/$\overline{\text{Write}}$) input specifies the read or write operation when the chip is selected. The small triangle shown at the outputs is the standard graphics symbol for three-state outputs. The *CS* input of the RAM controls the behavior of the data output lines. When $CS = 0$, the chip is not selected, and all its data outputs are in the high-impedance state. With $CS = 1$, the data output lines carry the eight bits of the selected word.

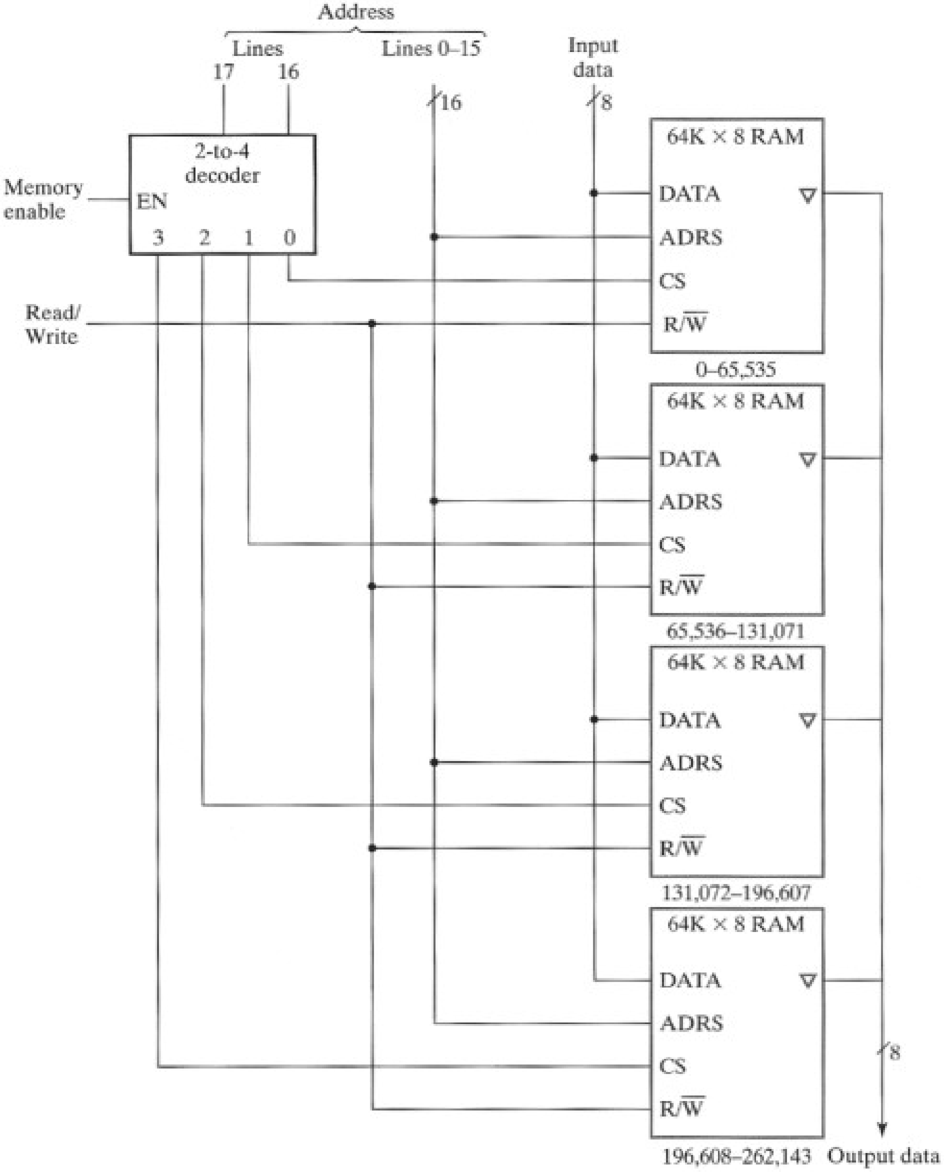
Suppose that we want to increase the number of words in the memory by using two or more RAM chips. Since every bit added to the address doubles the binary number that can be formed, it is natural to increase the number of words in factors of two. For example, two RAM chips will double the number of words and add one bit to the composite address. Four RAM chips multiply the number of words by four and add two bits to the composite address.

Consider the possibility of constructing a 256K × 8 RAM with four 64K × 8 RAM chips, as shown in Figure 9-10. The eight data input lines go to all the chips. The three-state outputs can be connected together to form the eight common data output lines. This type of output connection is possible only with three-state outputs. Just one chip select input will be active at any time, while the other three chips will be disabled. The eight outputs of the selected chip will contain 1's and 0's, and the other three will be in a high-impedance state, presenting only open circuits to the binary output signals of the selected chip.

The 256K-word memory requires an 18-bit address. The 16 least significant bits of the address are applied to the address inputs of all four chips. The two most

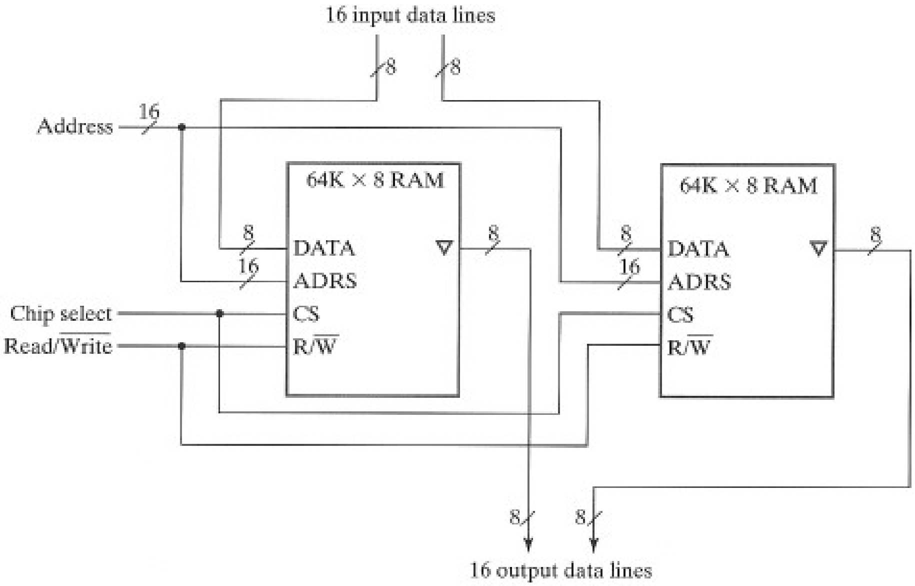


□ **FIGURE 9-9**
Symbol for a 64K × 8 RAM Chip



□ **FIGURE 9-10**
Block Diagram of a 256K × 8 RAM

significant bits are applied to a 2 × 4 decoder. The four outputs of the decoder are applied to the *CS* inputs of the four chips. The memory is disabled when the *EN* input of the decoder, Memory Enable, is equal to 0. All four outputs of the decoder are then 0, and none of the chips is selected. When the decoder is enabled, address bits 17 and 16 determine the particular chip that is selected. If these bits



□ **FIGURE 9-11**
Block Diagram of a 64K × 16 RAM

are equal to 00, the first RAM chip is selected. The remaining 16 address bits then select a word within the chip in the range from 0 to 65,535. The next 65,536 words are selected from the second RAM chip with an 18-bit address that starts with 01 followed by the 16 bits from the common address lines. The address range for each chip is listed in decimal under its symbol in the figure.

It is also possible to combine two chips to form a composite memory containing the same number of words, but with twice as many bits in each word. Figure 9-11 shows the interconnection of two 64K × 8 chips to form a 64K × 16 memory. The 16 data input and data output lines are split between the two chips. Both receive the same 16-bit address and the common *CS* and $R/\overline{W}$ control inputs.

The two techniques just described may be combined to assemble an array of identical chips into a large-capacity memory. The composite memory will have a number of bits per word that is a multiple of that for one chip. The total number of words will increase in factors of two times the word capacity of one chip. An external decoder is needed to select the individual chips based on the additional address bits of the composite memory.

To reduce the number of pins on the chip package, many RAM ICs provide common terminals for the data input and data output. The common terminals are said to be *bidirectional*, which means that for the read operation they act as outputs, and for the write operation they act as inputs. Bidirectional lines are constructed with three-state buffers and are discussed further in Section 2-8. The use of bidirectional signals requires control of the three-state buffers by both Chip Select and Read/$\overline{\text{Write}}$.

9-5 DRAM ICs

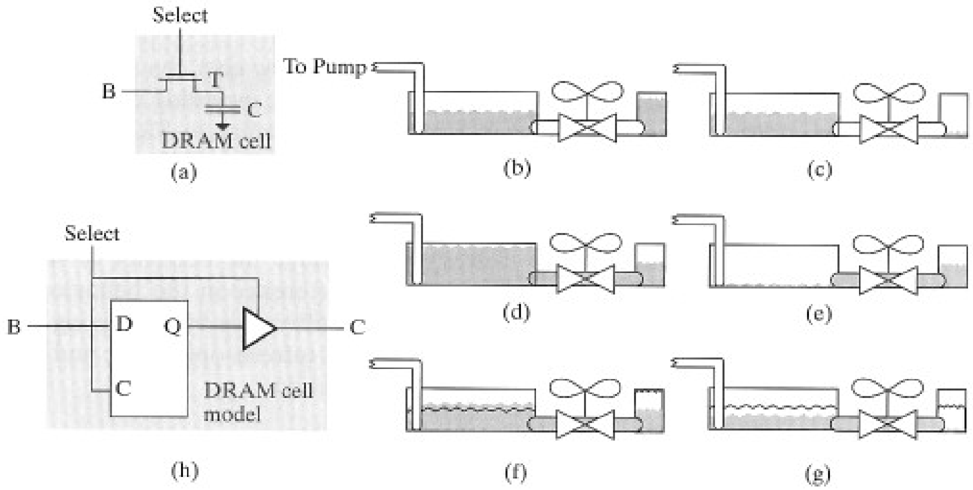
Because of its ability to provide high storage capacity at low cost, dynamic RAM (DRAM) dominates the high-capacity memory applications, including the primary RAM in computers. Logically, DRAM in many ways is similar to SRAM. However, because of the electronic circuit used to implement the storage cell, its electronic design is considerably more challenging. Further, as the name "dynamic" implies, the storage of information is inherently only temporary. As a consequence, the information must be periodically "refreshed" to mimic the behavior of static storage. This need for refresh is the primary logical difference in the behavior of DRAM compared to SRAM. We explore this logical difference by examining the dynamic RAM cell, the logic required to perform the refresh operation, and the impact of the need for refresh on memory system operation.

DRAM Cell

The dynamic RAM cell circuit is shown in Figure 9-12(a). It consists of a capacitor C and a transistor T. The capacitor is used to store electrical charge. If there is sufficient charge stored on the capacitor, it can be viewed as storing a logical 1. If there is insufficient charge stored on the capacitor, it can be viewed as storing a logical 0. The transistor acts much like a switch, in the same manner as the transmission gate introduced in Chapter 2. When the switch is "open," the charge on the capacitor roughly remains fixed, in other words, is stored. But when the switch is "closed," charge can flow into and out of the capacitor from the external Bit (B) line. This charge flow allows the cell to be written with a 1 or 0 and to be read.

In order to understand the read and write operations for the cell, we will use a hydraulic analogy with charge replaced by water, the capacitor by a small storage tank, and the transistor by a valve. Since the bit line has a large capacitance, it is represented by a large tank and pumps which can fill and empty this tank rapidly. This analogy is given in Figures 9-12(b) and 9-12(c) with the valve closed. Note that in one case the small storage tank is full representing a stored 1 and in the other case, it is empty representing a stored 0. Suppose that a 1 is to be written into the cell. The valve is opened and the pumps fill up the large tank. Water flows through the valve, filling the small storage tank as shown in Figure 9-12(d). Then the valve is closed, leaving the small tank full which represents a 1. A 0 can be written using the same sort of operations, except that the pumps empty the large tank as shown in Figure 9-12(e).

Now, suppose we want to read a stored value and that the value is a 1 corresponding to a full storage tank. With the large tank at a known intermediate level, the valve is opened. Since the small storage tank is full, water flows from the small tank to the large tank increasing the level of the water surface in the large tank slightly as shown in Figure 9-12(f). This increase in level is observed as the reading of 1 from the storage tank. Correspondingly, if the storage tank is initially empty, there will be a slight decrease in the level in the large tank in Figure 9-12(g), which is observed as the reading of a 0 from the storage tank.



□ **FIGURE 9-12**
Dynamic RAM cell, hydraulic analogy of cell operation, and cell model

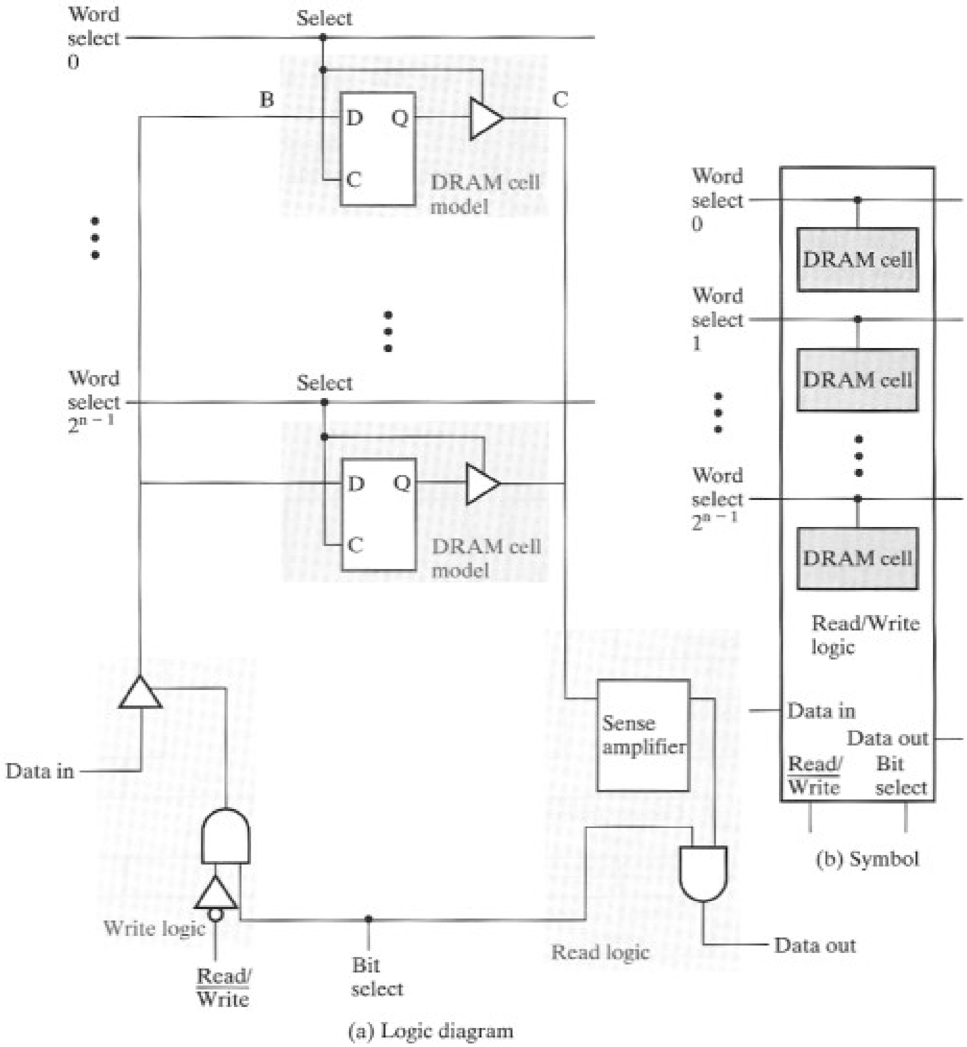
In the read operation just described, Figures 9-12(f) and 9-12(g) show that, regardless of the initial stored value in the storage tank, it now contains an intermediate value which will not cause enough of a change in the level of the external tank to permit a 0 or 1 to be observed. So the read operation has destroyed the stored value; this is referred to as a *destructive read*. To be able to read the original stored value in the future, we must *restore* it (i.e., return the storage tank to its original level). To perform the restore for a stored 1 observed, the large tank is filled by the pumps and the small tank fills through the open valve. To perform the restore for a stored 0 observed, the large tank is emptied by the pumps and the small tank drains through the open valve.

In the actual storage cell, there are other paths present for charge flow. These paths are analogous to small leaks in the storage tank. Due to these leaks, a full small storage tank will eventually drain to a point at which the increase in the level of the large tank on a read cannot be observed as an increase. In fact, if the small tank is less than half full when read, it is possible that a decrease in the level of the large tank may be observed. To compensate for these leaks, the small storage tank storing a 1 must be periodically refilled. This is referred to as a refresh of the cell contents. Every storage cell must be refreshed before its level has declined to a point at which the stored value can no longer be properly observed.

Through the hydraulic analogy, the DRAM operation has been explained. Just as for the SRAM, we employ a logic model for the cell. The model shown in Figure 9-12(h) is a D latch. The *C* input to the D latch is Select and the *D* input to the D latch is *B*. In order to model the output of the DRAM cell, we use a three-state buffer with Select as its control input and *C* as its output. In the original electronic circuit for the DRAM cell in Figure 9-12(a), *B* and *C* are the same signal, but in the logical model they are separate. This is necessary in the modeling process to avoid connecting gate outputs together.

DRAM Bit Slice

Using the logic model for the DRAM cell, we can construct the DRAM bit-slice model shown in Figure 9-13. This model is similar to that for the SRAM bit-slice in Figure 9-5. It is apparent that, aside from the cell structure, the two RAM bit slices are logically similar. However, from the standpoint of cost per bit, they are quite different. The DRAM cell consists of a capacitor plus one transistor. The SRAM cell typically contains six transistors, giving a cell complexity roughly three times that of the DRAM. Therefore, the number of SRAM cells in a chip of a given size



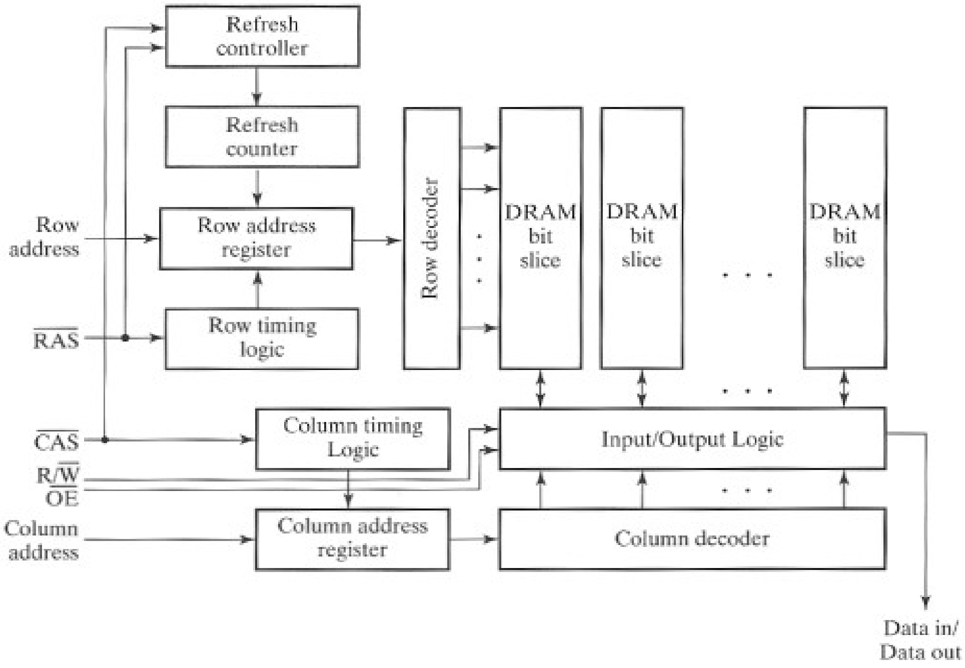
(a) Logic diagram

□ **FIGURE 9-13**
DRAM Bit Slice Model

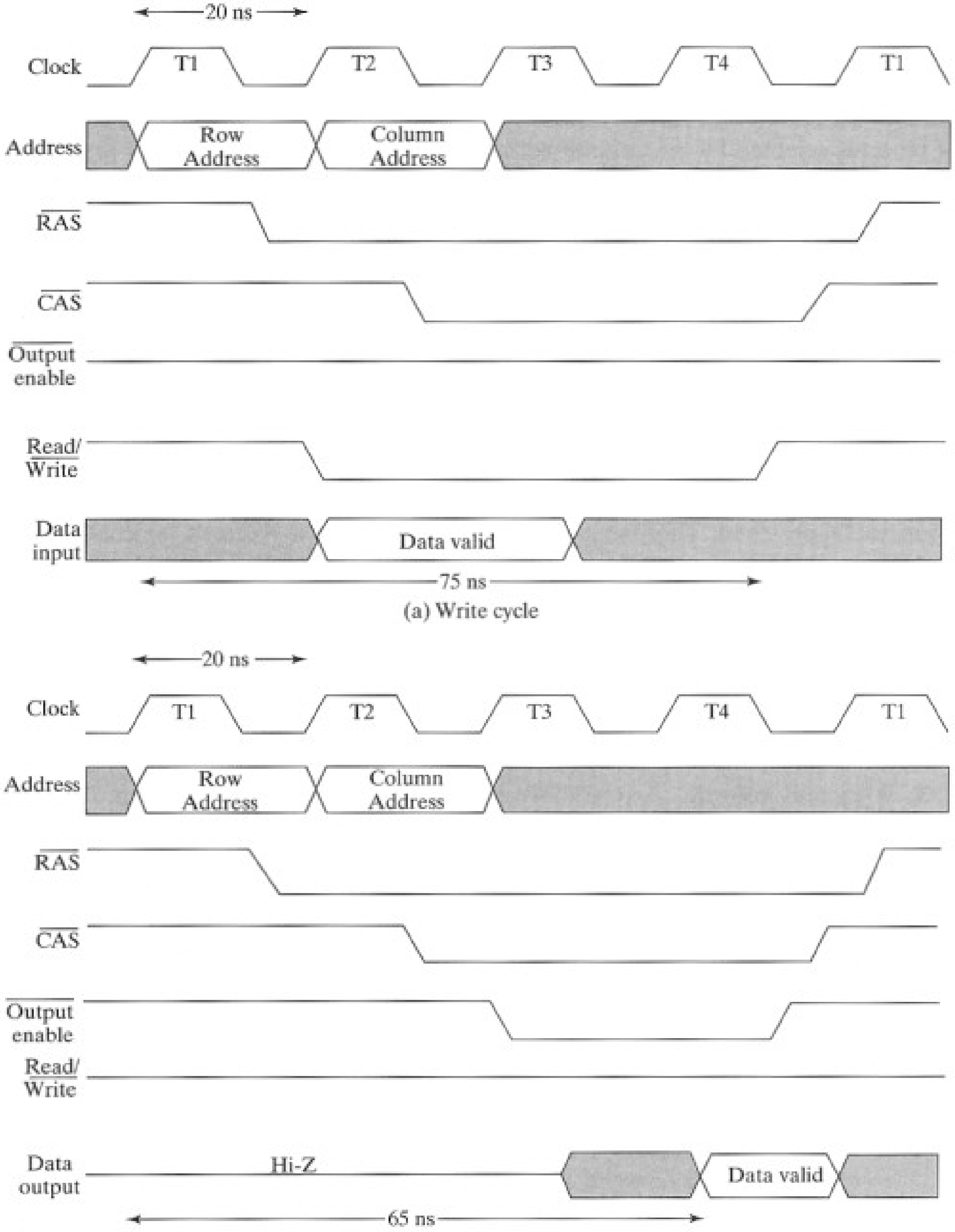
is less than one-third of those in the DRAM. The DRAM cost per bit is less than 1/3 the SRAM cost per bit, which justifies the use of DRAM in large memories.

Refresh of the DRAM contents remains to be discussed. But first, we need to develop the typical structure used to handle addressing in DRAMs. Since many DRAM chips are used in a DRAM, we want to reduce the physical size of the DRAM chips. Large DRAMs require 20 or more address bits, which would require 20 address pins on each DRAM chip. To reduce the number of pins, the DRAM address is applied serially in two parts with the row address first and the column address second. This can be done since the row address, which performs the row selection, is actually needed before the column address, which reads out the data from the row selected. In order to hold the row address throughout the read or write cycle, it is stored in a register as shown in Figure 9-14. The column address is also stored in a register. The load signal for the row address register is $\overline{\text{RAS}}$ ($\overline{\text{Row Address Strobe}}$) and for the column addresses is $\overline{\text{CAS}}$ ($\overline{\text{Column Address Strobe}}$). Note that in addition to $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$, control signals for the DRAM chip include R/$\overline{\text{W}}$ (Read/$\overline{\text{Write}}$), and $\overline{\text{OE}}$ ($\overline{\text{Output enable}}$). Note that this design uses signals active at the LOW (0) level.

The timing for DRAM write and read operation appears in Figure 9-15(a). The row address is applied to the address inputs, and then RAS changes from 1 to 0, loading the row address into the row address register. This address is applied to the row address decoder and selects a row of DRAM cells. Meanwhile, the column



□ **FIGURE 9-14**
Block Diagram of a DRAM Including Refresh Logic



(a) Write cycle

(b) Read cycle

□ **FIGURE 9-15**
Timing for DRAM Write and Read Operations

address is applied, and then CAS changes from 1 to 0, loading the column address into the column address register. This address is applied to the column address decoder, which selects a set of columns of the RAM array of size equal to the number of RAM data bits. The input data with Read/$\overline{\text{Write}}$ = 0 is applied over a time interval similar to that for the column address. The data bits are applied to the set of bit lines selected by the column address decoder which, in turn apply the values to the DRAM cells in the selected row, writing the new data into the cells. When CAS and RAS return to 1, the write cycle is complete and the DRAM cells store the newly written data. Note that the stored data in all of the other cells in the addressed row has been restored.

The read operation timing shown in Figure 9-15(b) is similar. Timing of the address operations is the same. However, no data is applied and Read/$\overline{\text{Write}}$ is 1 instead of 0. Data values in the DRAM cells in the selected row are applied to the bit lines and sensed by the sense amplifiers. The column address decoder selects the values to be sent to the Data output, which is enabled by $\overline{\text{Output enable}}$. During the read operation, all values in the addressed row are restored.

To support refresh, additional logic shown in color is present in the block diagram in Figure 9-14. There is a Refresh counter and a Refresh controller. The Refresh counter is used to provide the address of the row of DRAM cells to be refreshed. It is essential for the refresh modes that require the address to be provided from within the DRAM chip. The refresh counter advances on each refresh cycle. Due to the number of bits in the counter, when it reaches $2^n - 1$, where n is the number of rows in the DRAM array, it advances to 0 on the next refresh. The standard ways in which a refresh cycle can be triggered and the corresponding refresh types are as follows:

1. **RAS only refresh.** A row address is placed on the address lines and RAS is changed to 0. In this case, the refresh addresses must be applied from outside the DRAM chip, typically by an IC called a DRAM controller.
2. **CAS before RAS refresh.** The CAS is changed from 1 to 0 followed by a change from 1 to 0 on RAS. Additional refresh cycles can be performed by changing RAS without changing CAS. The refresh addresses for this case come from the refresh counter, which is incremented after the refresh for each cycle.
3. **Hidden refresh.** Following a normal read or write, CAS is left at 0 and RAS is cycled, effectively performing a CAS before RAS refresh. During a hidden refresh, the output data from the prior read remains valid. Thus, the refresh is hidden. Unfortunately, the time taken by the hidden refresh is significant, so a subsequent read or write operation is delayed.

In all cases, note that the initiation of a refresh is controlled externally by using the $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ signals. Each row of a DRAM chip requires refreshing within a specified maximum refresh time, typically ranging from 16 to 64 milliseconds (ms). Refreshes may be performed at evenly spaced points in the refresh time, an approach called distributed refresh. Alternatively, all refreshes may be performed one after the other, an approach called burst refresh. For example, a 4M × 4 DRAM has a refresh time of 64 ms and has 4096 rows to be refreshed. The length of time to perform a single refresh is 60 ns, and the refresh interval for distributed refresh is 64 ms/4096 = 15.6 microseconds (μs). A total time out for refresh of 0.25 ms is used out of the 64 ms refresh interval. For the same DRAM, a burst refresh also takes 0.25 ms. The DRAM controller must initiate a refresh every 15.6 μs for distributed refresh and must initiate 4,096 refreshes sequentially every 64 ms for burst refresh. During any refresh cycle, no DRAM reads or writes can occur. Since use of burst refresh would halt computer operation for a fairly long period, distributed refresh is more commonly used.

9-6 DRAM Types

Over the last two decades, both the capacity and speed of DRAM has increased significantly. The quest for speed has resulted in the evolution of many types of DRAM. Several of the DRAM types are listed with brief descriptions in Table 9-2. Of the memory types listed, the first two have largely been replaced in the marketplace by the more advanced SDRAM and RDRAM approaches. Since we have chosen to provide a discussion of error-correcting codes (ECC) for memories on the text website, our discussion of memory types here will omit the ECC feature and focus on synchronous DRAM, double data rate synchronous DRAM, and Rambus® DRAM. Before considering these three types of DRAM, some of the underlying concepts are covered briefly.

First of all, all three of these DRAM types work well because of the particular environment in which they operate. In modern high-speed computer systems, the processor interacts with the DRAM within a memory hierarchy. Most of the instructions and data for the processor are fetched from two lower levels of the hierarchy, the L1 and L2 caches. These are comparatively smaller SRAM-based memory structures that are covered in detail in Chapter 14. For our purposes, the key issue is that most of the reads from the DRAM are not directly from the CPU, but instead are reads initiated to bring data and instructions into these caches. The reads are in the form of a *line* (i.e., some number of bytes in contiguous addresses in memory) that is brought into the cache. For example, in a given read, the 16 bytes in hexadecimal addresses 000000 through 00000F would be read. This is referred to as a *burst read*. For burst reads, the effective *rate* of reading bytes, which is dependent upon reading bursts from contiguous addresses, rather than the access time is the important measure. With this measure, the three DRAM types we are discussing provide very fast performance.

Second, the effectiveness of these three DRAM types depends upon a very fundamental principle involved in DRAM operation, the reading out of all of the bits in a row for each read operation. The implication of this principle is that all of the bits in a row are available after a read using that row if only they can be accessed. With these two concepts in mind, the synchronous DRAM can be introduced.

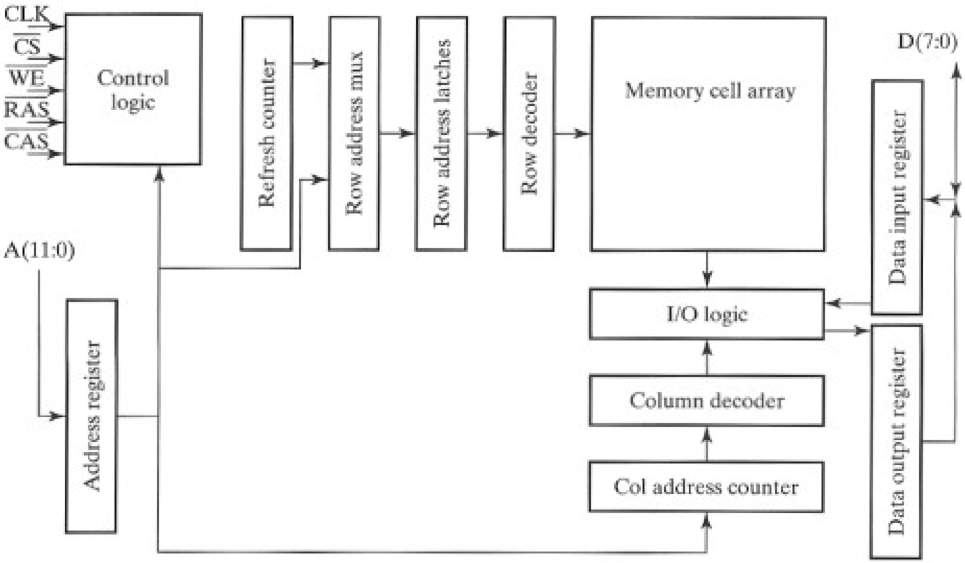
Synchronous DRAM (SDRAM)

The use of clocked transfers differentiates SDRAM from conventional DRAM. A block diagram of a 16-megabyte SDRAM IC appears in Figure 9-16. The inputs

TABLE 9-2
DRAM Types

Type	Abbreviation	Description
Fast Page Mode DRAM	FPM DRAM	Takes advantage of the fact that, when a row is accessed, all of the row values are available to be read out. By changing the column address, data from different addresses can be read out without reapplying the row address and waiting for the delay associated with reading out the row cells to pass if the row portion of the addresses match.
Extended Data Output DRAM	EDO DRAM	Extends the length of time that the DRAM holds the data values on its output, permitting the CPU to perform other tasks during the access since it knows the data will still be available.
Synchronous DRAM	SDRAM	Operates with a clock rather than being asynchronous. This permits a tighter interaction between memory and CPU, since the CPU knows exactly when the data will be available. SDRAM also takes advantage of the row value availability and divides memory into distinct banks, permitting overlapped accesses.
Double Data Rate Synchronous DRAM	DDR SDRAM	The same as SDRAM except that data output is provided on both the negative and the positive clock edges.
Rambus® DRAM	RDRAM	A proprietary technology that provides very high memory access rates using a relatively narrow bus.
Error-Correcting Code	ECC	May be applied to most of the DRAM types above to correct single bit data errors and often detect double errors.

and outputs differ little from those for the DRAM block diagram in Figure 9-14 with the exception of the presence of the clock for synchronous operation. Internally, there are a number of differences. Since the SDRAM appears synchronous from the outside, there are synchronous registers on the address inputs and the data inputs and outputs. In addition, a column address counter has been added, which is key to the operation of the SDRAM. While the control logic may appear to be similar, the control in this case is much more complex since the SDRAM has a mode control word that can be loaded from the address bus. Considering a 16 MB memory, the memory array contains 134,217,728 bits and is almost square, with 8,192 rows and 16,384 columns. There are 13 row address bits. Since there are

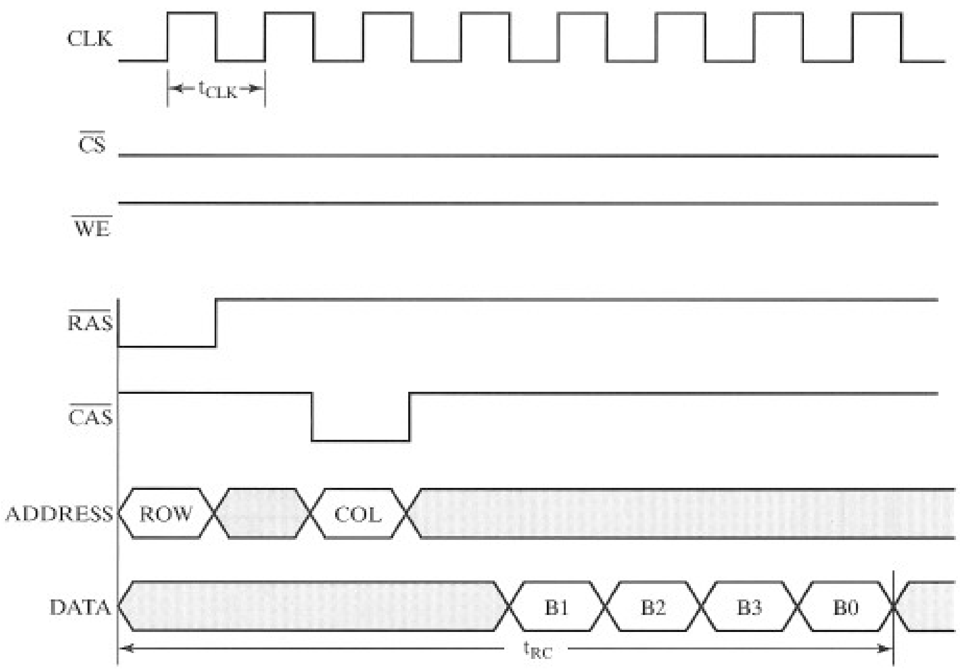


□ **FIGURE 9-16**
Block Diagram of a 16MB SDRAM

8 bits per byte, the number of column addresses is 16,384 divided by 8, which equals 2048. This requires 11 column address bits. Note that 13 plus 11 equals 24 giving the correct number of bits to address 16MB.

As with the regular DRAM, the SDRAM applies the row address first followed by the column address. The timing, however, is somewhat different, and some new terminology is used. Before performing an actual read operation from a specified column address, the entire row of 2048 bytes specified by the applied row address is read out internally and stored in the I/O logic. Internally, this step takes a few clock cycles. Next, the actual read step is performed with the column address applied. After an additional delay of a few clock cycles, the data bytes begin appearing on the output, one per clock period. The number of bytes that appear, the burst length, has been set by loading a mode control word into the control logic from the address input.

The timing of a burst read cycle with burst length equal to four is shown in Figure 9-17. The read begins with the application of the row address and the row address strobe (RAS), which causes the row address to be captured in the address register and the reading of the row to be initiated. During the next two clock periods, the reading of the row is taking place. During the third clock period, the column address and the column address strobe are applied, with the column address captured in the address register and the reading of the first data byte initiated. The data byte is then available to be read from the SDRAM at the positive clock edge occurring two cycles later. The second, third, and fourth bytes are available for reading on subsequent clock edges. In Figure 9-17, note that the bytes are presented in the order 1, 2, 3, 0. This is because, in the column address identifying the



□ **FIGURE 9-17**
Timing Diagram for an SDRAM

byte immediately needed by the CPU, the last two bits are 01. The subsequent bytes appear in the order of these two bits counted up modulo (burst length) by the column address counter, giving addresses ending in 01, 10, 11, and 00, with all other address bits fixed.

It is interesting to compare the byte rate for reading bytes from SDRAM to that of the basic DRAM. We assume that the read cycle time t_{RC} for the basic DRAM is 60 ns and that the clock period t_{CLK} for the SDRAM is 7.5 ns. The byte rate for the basic DRAM is one byte per 60 ns, or 16.67 MB/sec. For the SDRAM, from Figure 9-17, it requires 8.0 clock cycles, or 60 ns, to read four bytes, giving a byte rate of 66.67 MB/sec. If the burst is eight instead of four bytes, a read cycle time of 90 ns is required, giving a byte rate of 88.89 MB/sec. Finally, if the burst is the entire 2048-byte row of the SDRAM, the read cycle time becomes $60 + (2048 - 4) \times 7.5 = 15{,}390$ ns, giving a byte rate of 133.07 MB, which approaches the limit of one byte per 7.5 ns clock period.

Double Data Rate SDRAM (DDR SDRAM)

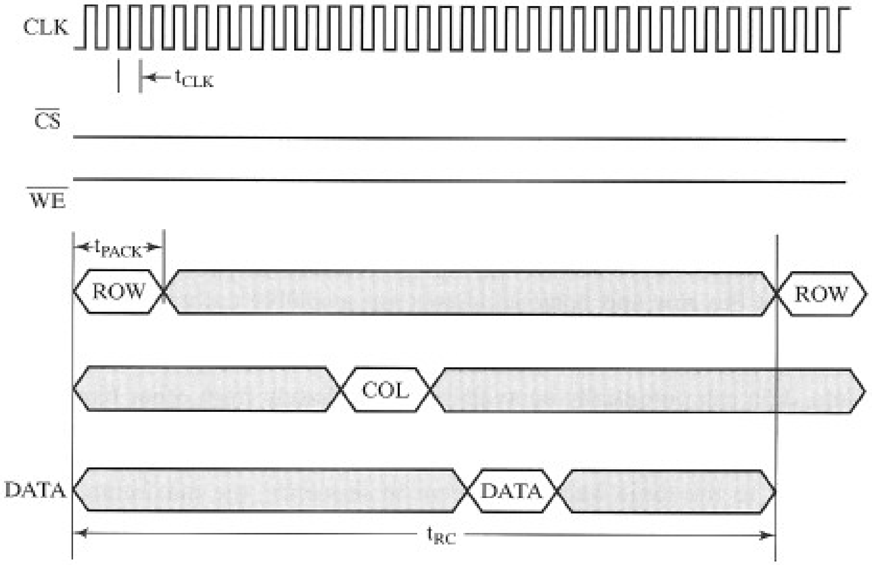
The second DRAM type, double data rate SDRAM (DDR SDRAM) overcomes the preceding limit without decreasing the clock period. Instead, it provides two bytes of data per clock period by using both the positive and negative clock edges. In Figure 9-17, four bytes are read, one per positive clock edge. By using both clock edges, eight bytes can be transferred in the same read cycle time t_{RC}. For a 7.5 ns clock period, the byte rate limit doubles in the example to 266.14 MB/sec.

Additional basic techniques can be applied to further increase the byte rate. For example, instead of having single byte data, an SDRAM IC can have the data I/O length of four bytes (32 bits). This gives a byte rate limit of 1.065 GB/sec with a 7.5 ns clock period. Eight bytes gives a byte rate limit of 2.130 GB/sec.

The byte rates achieved in the examples are upper limits. If the actual accesses needed are to different rows of the RAM, the delay from the application of the RAS pulse to read out the first byte of data is significant and leads to performance well below the limit. This can be partially offset by breaking up the memory into multiple banks where each of the banks performs the row read independently. Provided that the row and bank addresses are available early enough, row reads can be performed on one or more banks while data is still being transferred from the currently active row. When the column reads from the currently active row are complete, data can potentially be available immediately from other banks, permitting an uninterrupted flow of data from the memory. This permits the actual read rate to more closely approach the limit. Nevertheless, due to the fact that multiple row accesses to the same bank may occur in sequence, the maximum rate is not reached.

RAMBUS® DRAM (RDRAM)

The final DRAM type to be discussed is RAMBUS DRAM (RDRAM). RDRAM ICs are designed to be integrated into a memory system that uses a packet-based bus for the interaction between the RDRAM ICs and the memory bus to the processor. The primary components of the bus are a 3-bit path for the row address, a 5-bit path for the column address, and a 16-bit or an 18- bit path for data. The bus is synchronous and performs transfers on both clock edges, the same property possessed by the DDR SDRAM. Information on the three paths mentioned above is transferred in packets that are four clock cycles long, which means that there are eight transfers/packet. The number of bits per packet for each of the paths is 24 bits for the row address packet, 40 bits for the column address packet, and 128 bits or 144 bits for the data packet. The larger data packet includes 16 parity bits for implementing an error-correcting code. The RDRAM IC employs the concept of multiple memory banks mentioned earlier to provide capability for concurrent memory accesses with different row addresses. RDRAM uses the usual row activate technique in which the addressed row data of the memory is read. From this row data, the column address is used to select byte pairs in the order in which they are to be transmitted in the packet. A typical timing picture for an RDRAM read access is shown in Figure 9-18. Due to the sophisticated electronic design of the RAMBUS system, we can consider a clock period of 1.875 ns. Thus, the time for transmission of a packet is $t_{PACK} = 4 \times 1.875 = 7.5$ ns. The cycle time for accessing a single data packet of 8 byte pairs or 16 bytes is 32 clock cycles or 60 ns as shown in Figure 9-18. The corresponding byte rate is 266.67 MB/sec. If four of the byte packets are accessed from the same row, the rate increases to 1.067 GB/sec. By reading an entire RDRAM row of 2048 bytes, the cycle time increases to $60 + (2048 - 64) \times 1.875/4 = 990$ ns or a byte rate limit of $2048/(990 \times 10^{-9}) = 2.069$ MB/sec, approaching the ideal limit of 4/1.875 ns or 2.133 GB/sec.



□ **FIGURE 9-18**
Timing of a 16MB RDRAM

9-7 Arrays of Dynamic RAM ICs

Many of the same design principles used for SRAM arrays in Section 9-4 apply to DRAM arrays. There are, however, a number of different requirements for the control and addressing of DRAM arrays. These requirements are typically handled by a *DRAM controller*. The functions performed by a DRAM controller include the following:

1. controlling separation of the address into a row address and a column address and providing these addresses at the required times,
2. providing the $\overline{RAS}$ and $\overline{CAS}$ signals at the required times for read, write, and refresh operations,
3. performing refresh operations at the necessary intervals, and
4. providing status signals to the rest of the system (e.g., indicating whether the memory is busy performing refresh).

The DRAM controller is a complex synchronous sequential circuit with the external CPU clock providing synchronization of its operation.

9-8 Chapter Summary

Memory is of two types: random-access memory (RAM) and read-only memory (ROM). For both types, we apply an address to read from or write into a data word. Read and write operations have specific steps and associated timing parameters, including access time and write cycle time. Memory can be static or dynamic and volatile or nonvolatile. Internally, a RAM chip consists of an array of RAM cells, decoders, write circuits, read circuits, and output circuits. A combination of a write circuit, read circuit, and the associated RAM cells can be logically modeled as a RAM bit slice. RAM bit slices, in turn, can be combined to form two-dimensional RAM cell arrays, which, with decoders and output circuits added, form the basis for a RAM chip. Output circuits use three-state buffers in order to facilitate connecting together an array of RAM chips without significant additional logic. Due to the need for refresh, additional circuitry is required within DRAMs, as well as in arrays of DRAM chips. In a quest for faster memory access, a number of new DRAM types have been developed. The most recent forms of these high-speed DRAMs employ a synchronous interface that uses a clock to control memory accesses.

Error detection and correction codes, often based on Hamming codes, are used to detect or correct errors in stored RAM data. Material from Edition 1 covering these codes is available on the Companion Website for the text.

Material covering VHDL and Verilog for memories is available on the Companion Website for the text.

References

1. Weste, N. H. E., and Eshraghian, K. *Principles of CMOS VLSI Design: A Systems Perspective*, 2nd ed. Reading, MA: Addison-Wesley, 1993.
2. Micron Technology, Inc. *Micron 256Mb: x4, x8, x16 SDRAM*. www.micron.com, 2002.
3. Micron Technology, Inc. *Micron 64Mb: x32 DDR SDRAM*. www.micron.com, 2001.
4. Sobelman, M., "Rambus Technology Basics," *Rambus Developer Forum*. Rambus, Inc., October 2001.
5. Rambus, Inc. *Rambus Direct RDRAM 128/144-Mbit (256x16/18x32s) - Preliminary Information*, Document DL0059 Version 1.11.

Problems

The plus (+) indicates a more advanced problem and the asterisk (*) indicates a solution is available on the on the Companion Website for the text.

9–1. *The following memories are specified by the number of words times the number of bits per word. How many address lines and input-output data lines are needed in each case? (a) 16K × 8, (b) 256K × 16, (c) 64M × 32, and (d) 2G × 8.

9–2. Give the number of bytes stored in the memories listed in Problem 9–1.

9–3. *Word number $(835)_{10}$ in the memory shown in Figure 9-2 contains the binary equivalent of $(15{,}103)_{10}$. List the 10-bit address and the 16-bit memory contents of the word.

9–4. A 64K × 16 RAM chip uses coincident decoding by splitting the internal decoder into row select and column select. (a) Assuming that the RAM cell array is square, what is the size of each decoder, and how many AND gates are required for decoding an address? (b) Determine the row and column selection lines that are enabled when the input address is the binary equivalent of $(32000)_{10}$.

9–5. Assume that the largest decoder that can be used in an $m \times 1$ RAM chip has 13 address inputs and that coincident decoding is employed. In order to construct RAM chips that contain more 1-bit words than m, multiple RAM cell arrays, each with decoders and read/write circuits, are included in the chip.

(a) With the decoder restrictions given, how many RAM cell arrays are required to construct a 512 M × 1 RAM chip?

(b) Show the decoder required to select from among the different RAM arrays in the chip and its connections to address bits and column decoders.

9–6. A DRAM has 14 address pins and its row address is 1 bit longer than its column address. How many addresses, total, does the DRAM have?

9–7. A 256Mb DRAM uses 4-bit data and has equal length row and column addresses. How many address pins does the DRAM have?

9–8. A DRAM has a refresh interval of 128 ms and has 4096 rows. What is the interval between refreshes for distributed refresh? What is the minimum number of address pins on the DRAM?

9–9. ***(a)** How many 128K × 16 RAM chips are needed to provide a memory capacity of 1M bytes?

(b) How many address lines are required to access 1M bytes? How many of these lines are connected to the address inputs of all chips?

(c) How many lines must be decoded to produce the chip select inputs? Specify the size of the decoder.

9–10. Using the 64K × 8 RAM chip in Figure 9-9 plus a decoder, construct the block diagram for a 512K × 16 RAM.

9–11. Explain how SDRAM takes advantage of the two-dimensional storage array to provide a high data access rate.

9–12. Explain how a DDRAM achieves a data rate that is a factor of two higher than a comparable SDRAM.

CHAPTER 10

COMPUTER DESIGN BASICS

In Chapter 7, the separation of a design into a datapath that implements microoperations and a control unit that determines the sequence of microoperations was introduced. In this chapter, we define a generic computer datapath that implements register transfer microoperations and serves as a framework for the design of detailed processing logic. The concept of a control word provides a tie between the datapath and the control unit associated with it.

The generic datapath combined with a control unit and memory forms a programmable system, in this case, a simple computer. The concept of an instruction set architecture (ISA) is introduced as a means of specifying the computer. In order to implement the ISA, a control unit and the generic datapath are combined to form a CPU (Central Processing Unit). In addition, since this is a programmable system, memories are also present for storage of programs and data. Two different computers with two different control units are considered. The first computer has two memories, one for instructions and one for data, and performs all of its operations in a single clock cycle. The second computer has a single memory for both instructions and data and a more complex architecture requiring multiple clock cycles to perform its operations.

In the generic computer at the beginning of Chapter 1, register transfers, micro-operations, buses, datapaths, datapath components, and control words are used quite broadly. Likewise, control units appear in most of the digital parts of the generic computer. The design of processing units consisting of control units interacting with datapaths has its greatest impact within the generic computer in the CPU and FPU in the processor chip. These two components contain major datapaths that perform processing. The CPU and the FPU perform additions, subtractions, and most of the other operations specified by the instruction set.

10-1 Introduction

Computers and their design are introduced in this chapter. The specification for a computer consists of a description of its appearance to a programmer at the lowest level, its *instruction set architecture* (*ISA*). From the ISA, a high-level description of the hardware to implement the computer, called the *computer architecture,* is formulated. This architecture, for a simple computer, is typically divided into a datapath and a control. The datapath is defined by three basic components:

1. a set of registers,
2. the microoperations that are performed on data stored in the registers, and
3. the control interface.

The control unit provides signals that control the microoperations performed in the datapath and in other components of the system, such as memories. In addition, the control unit controls its own operation, determining the sequence of events that occur. This sequence may depend upon the results of current and past microoperations executed. In a more complex computer, typically multiple control units and datapaths are present.

To build a foundation for considering computer designs, initially, we extend the ideas in Chapter 7 to the implementation of datapaths. Specifically, we consider a generic datapath, one that can be used, in some cases in modified form, in all of the computer designs considered in the remainder of the text. These future designs show how a given datapath can be used to implement different instruction set architectures by simply combining the datapath with different control units.

10-2 Datapaths

Instead of having each individual register perform its microoperations directly, computer systems often employ a number of storage registers in conjunction with a shared operation unit called an *arithmetic/logic unit*, abbreviated ALU. To perform a microoperation, the contents of specified source registers are applied to the inputs of the shared ALU. The ALU performs an operation, and the result of this operation is transferred to a destination register. With the ALU as a combinational circuit, the entire register transfer operation from the source registers, through the ALU, and into the destination register is performed during one clock cycle. The shift operations are often performed in a separate unit, but sometimes these operations are also implemented within the ALU.

Recall that the combination of a set of registers with a shared ALU and interconnecting paths is the datapath for the system. The rest of this chapter is concerned with the organization and design of datapaths and associated control units used to implement simple computers. The design of a particular ALU is undertaken to show the process involved in implementing a complex combinational circuit. We also design a shifter, combine control signals into control words, and then add control units to implement two different computers.

The datapath and the control unit are the two parts of the processor, or CPU, of a computer. In addition to the registers, the datapath contains the digital logic that implements the various microoperations. This digital logic consists of buses, multiplexers, decoders, and processing circuits. When a large number of registers is included in a datapath, the registers are most conveniently connected through one or more buses. Registers in a datapath interact by the direct transfer of data, as well as in the performance of the various types of microoperations. A simple bus-based datapath with four registers, an ALU, and a shifter is shown in Figure 10-1. The shading and blue signal names relate to Figure 10-10 and will be discussed in Section 10-5. The black signal names are used here to describe the details in Figure 10-1. Each register is connected to two multiplexers to form ALU and shifter input buses *A* and *B*. The select inputs on each multiplexer select one register for the corresponding bus. For Bus *B*, there is an additional multiplexer, MUX *B*, so that constants can be brought into the datapath from outside using Constant in. Bus *B* also connects to Data out, to send data outside the datapath to other components of the system, such as memory or input-output. Likewise, Bus *A* connects to Address out, to send address information outside of the datapath for memory or input-output.

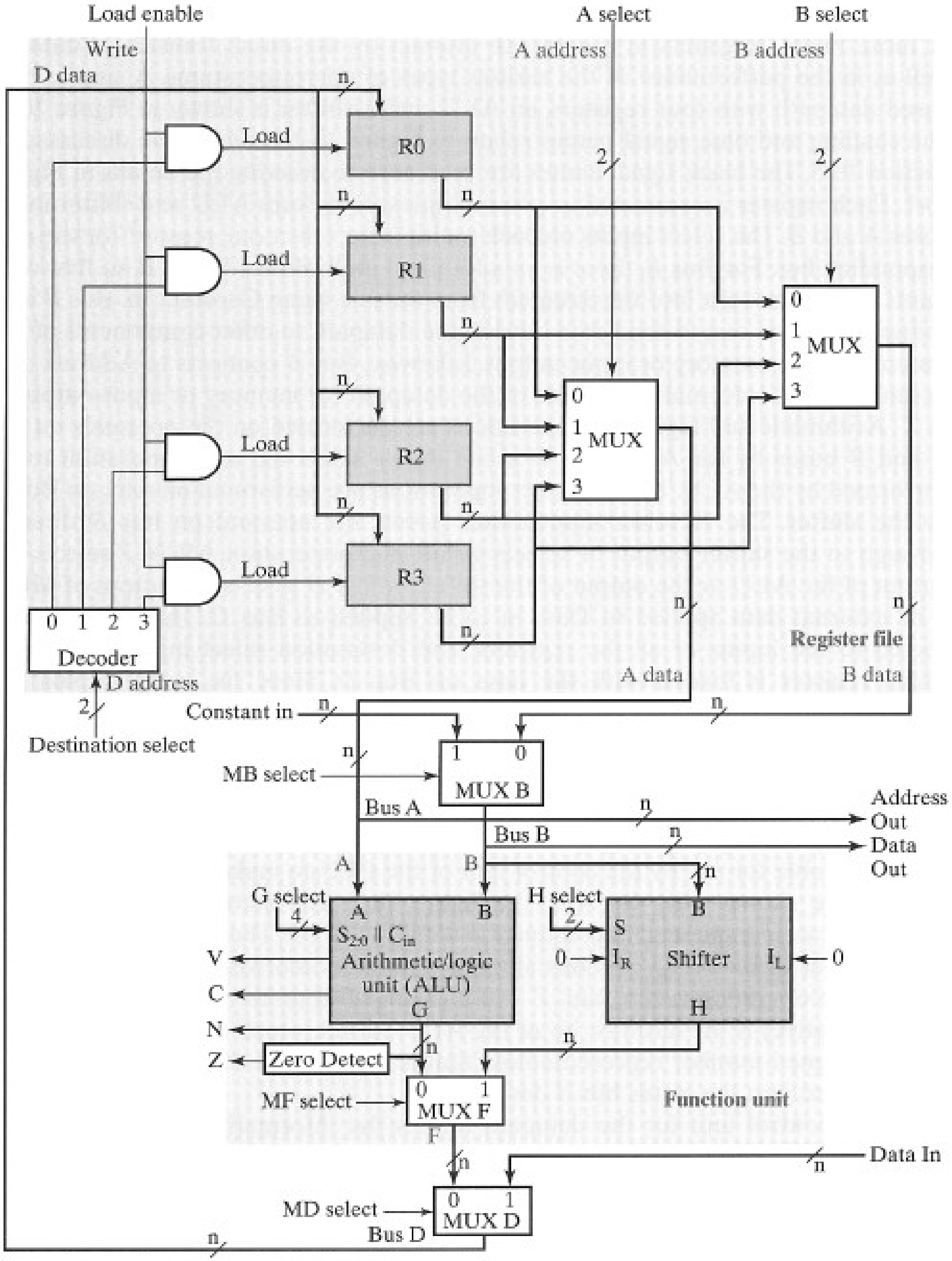
Arithmetic and logic microoperations are performed on the operands on the *A* and *B* buses by the ALU. The *G* select inputs select the microoperation to be performed by the ALU. The shift microoperations are performed on data on Bus *B* by the shifter. The *H* select input either passes the operand on bus *B* directly through to the shifter output or selects a shift microoperation. MUX *F* selects the output of the ALU or the output of the shifter. MUX *D* selects the output of MUX *F* or external data applied to Data in to be applied to Bus *D*. The latter is connected to the inputs of all the registers. The destination select inputs determine which register is loaded with the data on Bus *D*. Since the select inputs are decoded, only one register Load signal is active for any transfer of data into a register from Bus *D*. A Load enable signal that can force all Load signals to 0 using AND gates is present for transfers that are not to change the contents of any of the four registers.

It is useful to have certain information, based on the results of an ALU operation, available for use by the control unit of the CPU to make decisions. Four status bits are shown with the ALU in Figure 10-1. The status bits, carry *C* and overflow *V*, were explained in conjunction with Figure 5-9. The zero status bit *Z* is 1 if the output of the ALU contains all zeros and is 0 otherwise. Thus, $Z = 1$ if the result of an operation is zero, and $Z = 0$ if the result is nonzero. The sign status bit *N* (for negative) is the leftmost bit of the ALU output, which is the sign bit for the result in signed-number representations. Status values from the shifter can also be incorporated into the status bits if desired.

The control unit for the datapath directs the information flow through the buses, the ALU, the shifter, and the registers by applying signals to the select inputs. For example, to perform the microoperation

$$R1 \leftarrow R2 + R3$$

the control unit must provide binary selection values to the following sets of control inputs:



□ **FIGURE 10-1**
Block Diagram of a Generic Datapath

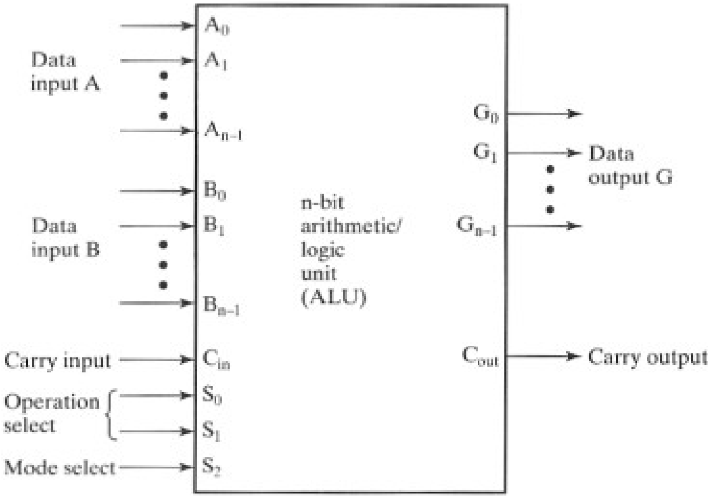
1. *A* select, to place the contents of *R*2 onto *A* data and, hence, Bus *A*.
2. *B* select, to place the contents of *R*3 onto the 0 input of MUX *B*; and *MB* select, to put the 0 input of MUX *B* onto Bus *B*.
3. *G* select, to provide the arithmetic operation $A + B$.
4. *MF* select, to place the ALU output on the MUX *F* output.
5. *MD* select, to place the MUX *F* output onto Bus *D*.
6. Destination select, to select *R*1 as the destination of the data on Bus *D*.
7. Load enable, to enable a register—in this case, *R*1—to be loaded.

The sets of values must be generated and must become available on the corresponding control lines early in the clock cycle. The binary data from the two source registers must propagate through the multiplexers and the ALU and on into the inputs of the destination register, all during the remainder of the same clock cycle. Then, when the next positive clock edge arrives, the binary data on Bus *D* is loaded into the destination register. To achieve fast operation, the ALU and shifter are constructed with combinational logic having a limited number of levels, such as a carry-lookahead adder.

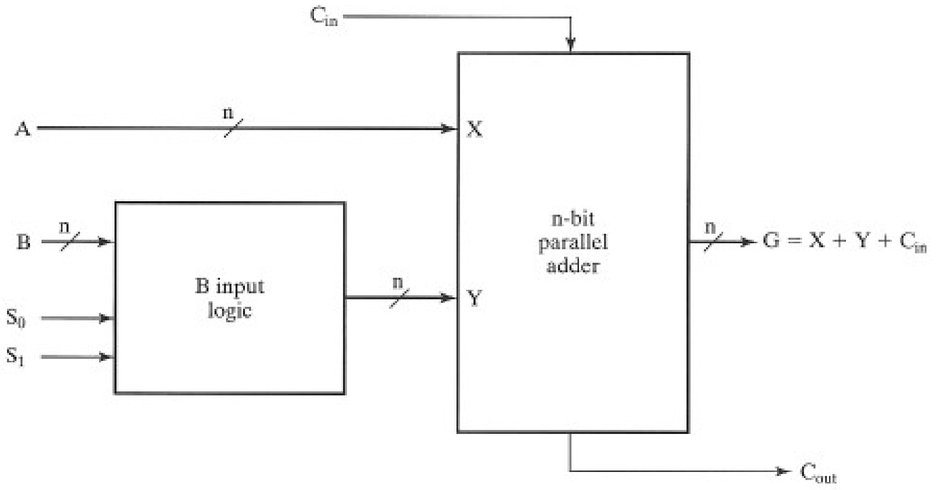
10-3 The Arithmetic/Logic Unit

The ALU is a combinational circuit that performs a set of basic arithmetic and logic microoperations. The ALU has a number of selection lines used to determine the operation to be performed. The selection lines are decoded within the ALU, so that *k* selection lines can specify up to 2^k distinct operations.

Figure 10-2 shows the symbol for a typical *n*-bit ALU. The *n* data inputs from *A* are combined with the *n* data inputs from *B* to generate the result of an operation



□ **FIGURE 10-2**
Symbol for an *n*-Bit ALU



□ **FIGURE 10-3**
Block Diagram of an Arithmetic Circuit

at the G outputs. The mode-select input S_2 distinguishes between arithmetic and logic operations. The two Operation select inputs S_1 and S_0 and the Carry input C_{in} specify the eight arithmetic operations with S_2 at 0. Operand select input S_0 and C_{in} specify the four logic operations with S_2 at 1.

We perform the design of this ALU in three stages. First, we design the arithmetic section. Then we design the logic section, and finally, we combine the two sections to form the ALU.

Arithmetic Circuit

The basic component of an arithmetic circuit is a parallel adder, which is constructed with a number of full-adder circuits connected in cascade, as shown in Figure 5-5. By controlling the data inputs to the parallel adder, it is possible to obtain different types of arithmetic operations. The block diagram in Figure 10-3 demonstrates a configuration in which one set of inputs to the parallel adder is controlled by the select lines S_1 and S_0. There are n bits in the arithmetic circuit, with two inputs A and B and output G. The n inputs from B go through the B input logic to the Y inputs of the parallel adder. The input carry C_{in} goes in the carry input of the full adder in the least-significant-bit position. The output carry C_{out} is from the full adder in the most-significant-bit position. The output of the parallel adder is calculated from the arithmetic sum as

$$G = X + Y + C_{in}$$

where X is the n-bit binary number from the inputs and Y is the n-bit binary number from the B input logic. C_{in} is the input carry, which equals 0 or 1. Note that the symbol + in the equation denotes arithmetic addition.

□ **TABLE 10-1**
Function Table for Arithmetic Circuit

Select		Input	$G = A + Y + C_{in}$	
S_1	S_0	Y	$C_{in} = 0$	$C_{in} = 1$
0	0	all 0's	$G = A$ (transfer)	$G = A + 1$ (increment)
0	1	B	$G = A + B$ (add)	$G = A + B + 1$
1	0	$\overline{B}$	$G = A + \overline{B}$	$G = A + \overline{B} + 1$ (subtract)
1	1	all 1's	$G = A - 1$ (decrement)	$G = A$ (transfer)

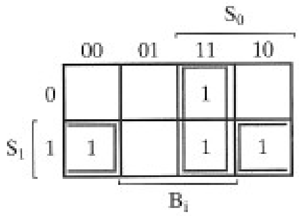
Table 10-1 shows the arithmetic operations that are obtainable by controlling the value of Y with the two selection inputs S_1 and S_0. If the inputs from B are ignored and we insert all 0's at the Y inputs, the output sum becomes $G = A + 0 + C_{in}$. This gives $G = A$ when $C_{in} = 0$ and $G = A + 1$ when $C_{in} = 1$. In the first case, we have a direct transfer from input A to output G. In the second case, the value of A is incremented by 1. For a straight arithmetic addition, it is necessary to apply the B inputs to the Y inputs of the parallel adder. This gives $G = A + B$ when $C_{in} = 0$. Arithmetic subtraction is achieved by applying the complement of inputs B to the Y inputs of the parallel adder, to obtain $G = A + \overline{B} + 1$ when $C_{in} = 1$. This gives A plus the 2's complement of B, which is equivalent to 2's complement subtraction. All 1's is the 2's complement representation for –1. Thus, applying all 1's to the Y inputs with $C_{in} = 0$ produces the decrement operation $G = A - 1$.

The B input logic in Figure 10-3 can be implemented with n multiplexers. The data inputs to each multiplexer in stage i for $i = 0, 1, \ldots, n - 1$ are $0, B_i, \overline{B}_i$, and 1, corresponding to selection values S_1S_0: 00, 01, 10, and 11, respectively. Thus, the arithmetic circuit can be constructed with n full adders and n 4-to-1 multiplexers.

The number of gates in the B input logic can be reduced if, instead of using 4-to-1 multiplexers, we go through the logic design of one stage (one bit) of the B input logic. This can be done as shown in Figure 10-4. The truth table for one typical

Inputs			Output	
S_1	S_0	B_i	Y_i	
0	0	0	0	$Y_i = 0$
0	0	1	0	
0	1	0	0	$Y_i = B_i$
0	1	1	1	
1	0	0	1	$Y_i = \overline{B}_i$
1	0	1	0	
1	1	0	1	$Y_i = 1$
1	1	1	1	

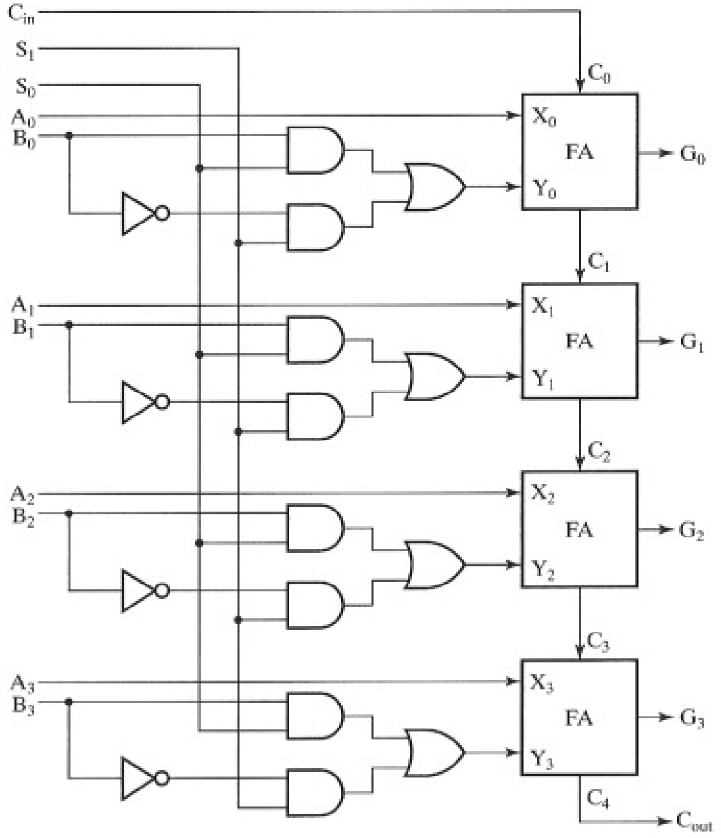
(a) Truth table



	00	01	11	10
0			1	
1	1		1	1

(b) Map Simplification:
$Y_i = B_iS_0 + \overline{B}_iS_1$

□ **FIGURE 10-4**
B Input Logic for One Stage of Arithmetic Circuit



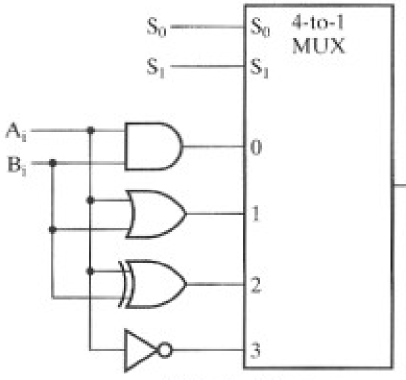
□ **FIGURE 10-5**
Logic Diagram of a 4-bit Arithmetic Circuit

stage i of the logic is given in Figure 10-4(a). The inputs are S_1, S_0, and B_i, and the output is Y_i. Following the requirements specified in Table 10-1, we let $Y_i = 0$ when $S_1S_0 = 00$, and similarly assign the other three values of Y_i for each of the combinations of the selection variables. Output Y_i is simplified in the map in Figure 10-4(b), to give

$$Y_i = B_iS_0 + \overline{B_i}S_1$$

where S_1 and S_0 are common to all n stages. Each stage i is associated with input B_i and output Y_i for $i = 0, 1, 2, \ldots, n-1$. This logic corresponds to a 2-to-1 multiplexer with B_i on the select input and S_1 and S_0 on the data inputs.

Figure 10-5 shows the logic diagram of an arithmetic circuit for $n = 4$. The four full-adder (FA) circuits constitute the parallel adder. The carry into the first stage is the input carry C_{in}. All other carries are connected internally from one stage to the next. The selection variables are S_1, S_0, and C_{in}. Variables S_1 and S_0



(a) Logic Diagram

S_1	S_0	Output	Operation
0	0	$G = A \wedge B$	AND
0	1	$G = A \vee B$	OR
1	0	$G = A \oplus B$	XOR
1	1	$G = \overline{A}$	NOT

(b) Function Table

□ **FIGURE 10-6**
One Stage of Logic Circuit

control all Y inputs of the full adders according to the Boolean function derived in Figure 10-4(b). Whenever C_{in} is 1, $A + Y$ has 1 added. The eight arithmetic operations for the circuit as a function of S_1, S_0, and C_{in} are listed in Table 10-2. It is interesting to note that the operation $G = A$ appears twice in the table. This is a harmless by-product of using C_{in} as one of the control variables while implementing both increment and decrement instructions.

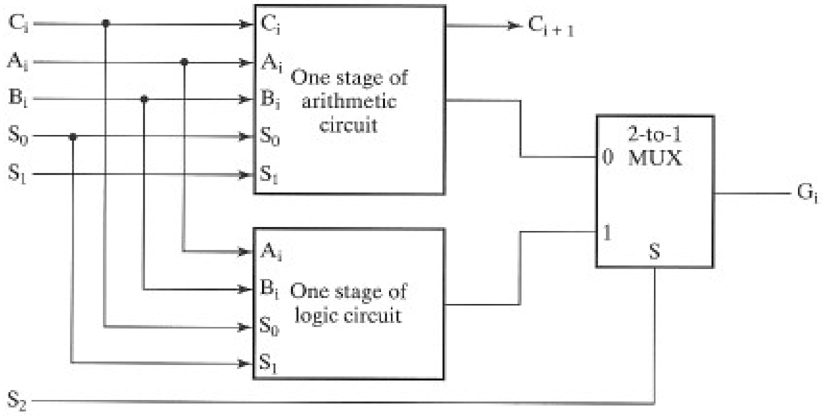
Logic Circuit

The logic microoperations manipulate the bits of the operands by treating each bit in a register as a binary variable, giving bitwise operations. There are four commonly used logic operations—AND, OR, XOR (exclusive-OR), and NOT—from which others can be conveniently derived.

Figure 10-6(a) shows one stage of the logic circuit. It consists of four gates and a 4-to-1 multiplexer, although simplification could yield less complex logic. Each of the four logic operations is generated through a gate that performs the required logic. The outputs of the gates are applied to the inputs of the multiplexer with two selection variables S_1 and S_0. These choose one of the data inputs of the multiplexer and direct its value to the output. The diagram shows a typical stage with subscript i. For the logic circuit with n bits, the diagram must be repeated n times, for $i = 0, 1, 2, \ldots, n - 1$. The selection variables are applied to all stages. The function table in Figure 10-6(b) lists the logic operations obtained for each combination of the selection values.

Arithmetic/Logic Unit

The logic circuit can be combined with the arithmetic circuit to produce an ALU. Selection variables S_1 and S_0 can be common to both circuits, provided that we use a third selection variable to differentiate between the two. The configuration for one stage of the ALU is illustrated in Figure 10-7. The outputs of the arithmetic



□ **FIGURE 10-7**
One Stage of ALU

and logic circuits in each stage are applied to a 2-to-1 multiplexer with selection variable S_2. When $S_2 = 0$, the arithmetic output is selected, and when $S_2 = 1$, the logic output is selected. Note that the diagram shows just one typical stage of the ALU; the circuit must be repeated n times for an n-bit ALU. The output carry C_{i+1} of a given arithmetic stage must be connected to the input carry C_i of the next stage in sequence. The input carry to the first stage is the input carry C_{in}, which also acts as a selection variable for the arithmetic operations.

The ALU specified in Figure 10-7 provides eight arithmetic and four logic operations. Each operation is selected through the variables S_2, S_1, S_0, and C_{in}. Table 10-2 lists the 12 ALU operations. The first eight are arithmetic operations

□ **TABLE 10-2**
Function Table for ALU

Operation Select					
S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	$G = A$	Transfer A
0	0	0	1	$G = A + 1$	Increment A
0	0	1	0	$G = A + B$	Addition
0	0	1	1	$G = A + B + 1$	Add with carry input of 1
0	1	0	0	$G = A + \overline{B}$	A plus 1's complement of B
0	1	0	1	$G = A + \overline{B} + 1$	Subtraction
0	1	1	0	$G = A - 1$	Decrement A
0	1	1	1	$G = A$	Transfer A
1	X	0	0	$G = A \wedge B$	AND
1	X	0	1	$G = A \vee B$	OR
1	X	1	0	$G = A \oplus B$	XOR
1	X	1	1	$G = \overline{A}$	NOT (1's complement)

and are selected with $S_2 = 0$. The next four are logic operations and are selected with $S_2 = 1$. To provide selection codes using as few bits as possible, S_0 and C_i are used to control the selection of the logic operations instead of S_2 and S_1. Selection input S_1 has no effect during the logic operations and is marked with X to indicate that its value may be either 0 or 1. Later in the design, it is assigned value 0 for logic operations.

The ALU logic we have designed is not as simple as it could be and has a fairly high number of logic levels, contributing to propagation delay in the circuit. With the use of logic simplification software, we can simplify this logic and reduce the delay. For example, it is quite easy to simplify the logic for a single stage of the ALU. For realistic n, a means of further reducing the carry propagation delay in the ALU, such as the carry lookahead adder from Section 5-2, is usually necessary.

10-4 The Shifter

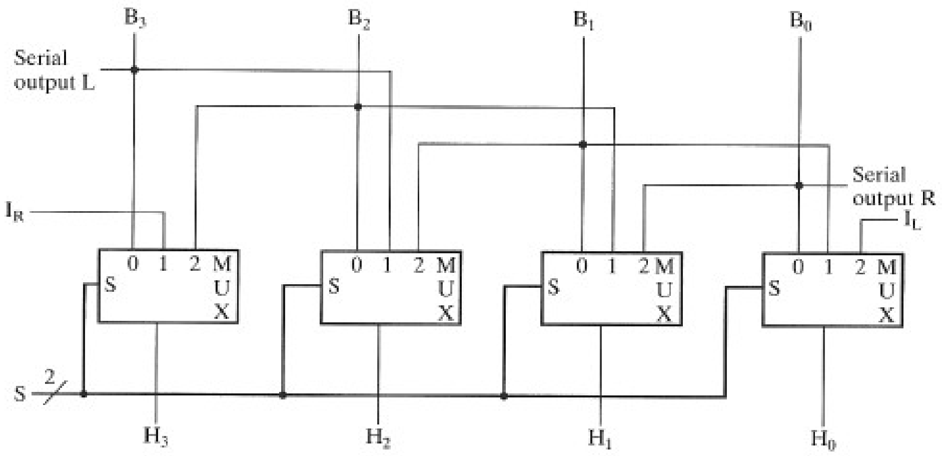
The shifter shifts the value on Bus *B*, placing the result on an input of MUX *F*. The basic shifter performs one of two main types of transformations on the data: right shift and left shift.

A seemingly obvious choice for a shifter would be a bidirectional shift register with parallel load. Data from Bus *B* can be transferred to the register in parallel and then shifted to the right, the left, or not at all. A clock pulse loads the output of Bus *A* into the shift register, and a second clock pulse performs the shift. Finally, a third clock pulse transfers the data from the shift register to the selected destination register.

Alternatively, the transfer from a source register to a destination register can be done using only one clock pulse if the shifter is implemented as a combinational circuit as done in Chapter 5. Because of the faster operation that results from the use of one clock pulse instead of three, this is the preferred method. In a combinational shifter, the signals propagate through the gates without the need for a clock pulse. Hence, the only clock needed for a shift in the datapath is for loading the data from Bus *H* into the selected destination register.

A combinational shifter can be constructed with multiplexers as shown in Figure 10-8. The selection variable *S* is applied to all four multiplexers to select the type of operation within the shifter. $S = 00$ causes B to be passed through the shifter unchanged. S = 01 causes a right-shift operation and S = 10 causes a left-shift operation. The right shift fills the position on the left with the value on serial input I_R. The left shift fills the position on the right with the value on serial input I_L. Serial outputs are available from serial output *R* and serial output *L* for right and left shifts, respectively.

The diagram of Figure 10-8 shows only four stages of the shifter, which has n stages in a system with n-bit operands. Additional selection variables may be employed to specify what goes into I_R and I_L during a single bit-position shift. Note that to shift an operand by $m > 1$ bit positions, this shifter must perform a series of m 1-bit position shifts, taking m clock cycles.



□ **FIGURE 10-8**
4-Bit Basic Shifter

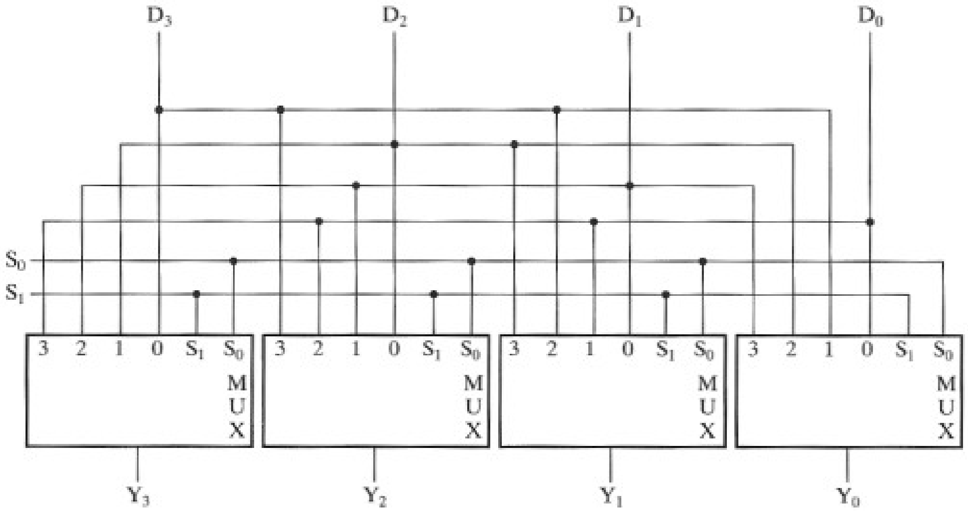
Barrel Shifter

In datapath applications, often the data must be shifted more than one bit position in a single clock cycle. A *barrel shifter* is one form of combinational circuit that shifts or rotates the input data bits by the number of bit positions specified by a binary value on a set of selection lines. The shift we consider here is a rotation to the left, which means that the binary data is shifted to the left, with the bits coming from the most significant part of the register rotated back into the least significant part of the register.

A 4-bit version of this kind of barrel shifter is shown in Figure 10-9. It consists of four multiplexers with common select lines S_1 and S_0. The selection variables determine the number of positions that the input data will be shifted to the left by rotation. When $S_1S_0 = 00$, no shift occurs, and the input data has a direct path to the outputs. When $S_1S_0 = 01$, the input data are rotated one position, with D_0 going to Y_1, D_1 going to Y_2, D_2 going to Y_3, and D_3 going to Y_0. When $S_1S_0 = 10$, the input is rotated two positions, and when $S_1S_0 = 11$, the rotation is by three bit positions. Table 10-3 gives the function table for the 4-bit barrel shifter. For each

□ **TABLE 10-3**
Function Table for 4-Bit Barrel Shifter

Select		Output				
S_1	S_0	Y_3	Y_2	Y_1	Y_0	Operation
0	0	D_3	D_2	D_1	D_0	No rotation
0	1	D_2	D_1	D_0	D_3	Rotate one position
1	0	D_1	D_0	D_3	D_2	Rotate two positions
1	1	D_0	D_3	D_2	D_1	Rotate three positions



□ **FIGURE 10-9**
4-Bit Barrel Shifter

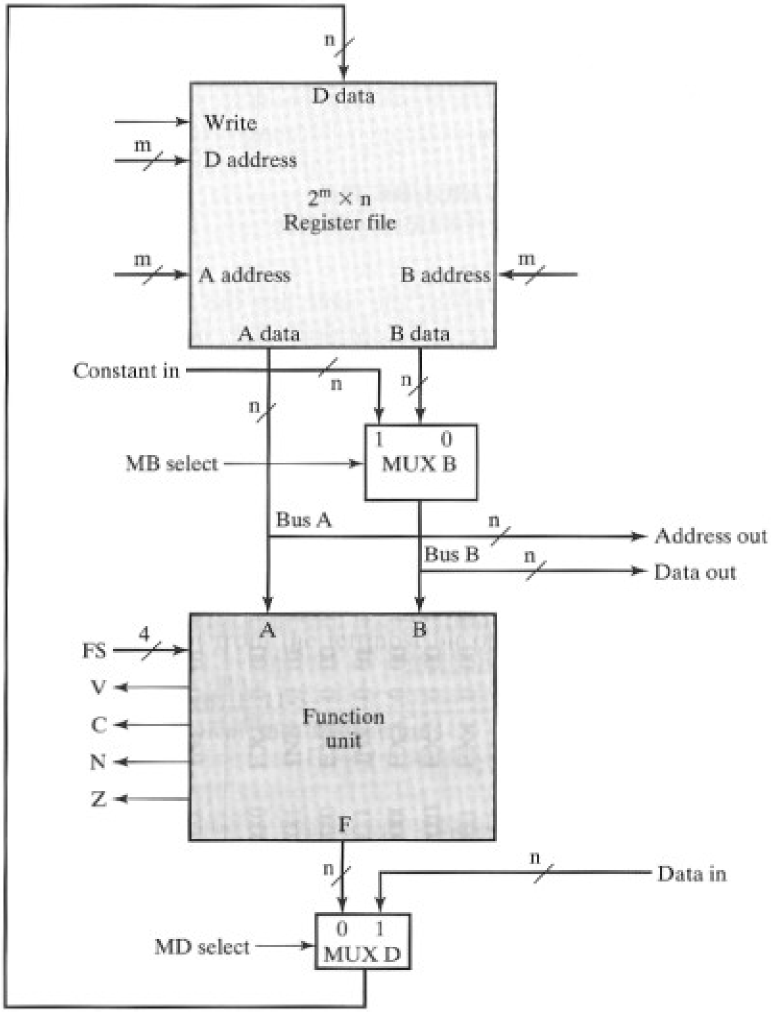
binary value of the selection variables, the table lists the inputs that go to the corresponding output. Thus, to rotate three positions, S_1S_0 must be equal to 11, causing D_0 to go to Y_3, D_1 to go to Y_0, D_2 to go to Y_1, and D_3 to go to Y_2. Note that, by using this left-rotation barrel shifter, one can generate all desired right rotations as well. For example, a left rotation by three positions is the same as a right rotation by one position in this 4-bit barrel shifter. In general, in a 2^n-bit barrel shifter, i positions of left rotation is the same as $2^n - i$ bits of right rotation.

A barrel shifter with 2^n input and output lines requires 2^n multiplexers, each having 2^n data inputs and n selection inputs. The number of positions for the data to be rotated is specified by the selection variables and can be from 0 to $2^n - 1$ positions. For a large n, the fan-in to gates is too large, so larger barrel shifters consist of layers of multiplexers, as shown in Section 12-2, or of special structures designed at the transistor level.

10-5 Datapath Representation

The datapath in Figure 10-1 includes the registers, selection logic for the registers, the ALU, the shifter, and three additional multiplexers. With a hierarchical structure, we can reduce the apparent complexity of the datapath. This reduction is important, since we frequently use this datapath. Also, as illustrated by the register file to be discussed next, the use of a hierarchy allows one implementation of a module to be replaced with another, so that we are not tied to specific logic implementations.

A typical datapath has more than four registers. Indeed, computers with 32 or more registers are common. The construction of a bus system with a large number of registers requires different techniques. A set of registers having common microoperations performed on them may be organized into a *register file*. The typical register file is a special type of fast memory that permits one or more words to be read and one or more words to be written, all simultaneously. Functionally, a simple register file contains the equivalent of the logic shaded in blue in Figure 10-1. Due to the memory-like nature of register files, the *A* select, *B* select, and Destination select inputs in the figure, become three addresses. As shown in Figure 10-1 in blue and on the register file symbol in Figure 10-10, the *A* address accesses a word to be read onto *A* data, the *B* address accesses a second word to be read onto *B* data, and the *D* address accesses a word to be written into from *D* data. All of these accesses



□ **FIGURE 10-10**
Block Diagram of Datapath Using the Register File and Function Unit

□ **TABLE 10-4**
***G* Select, *H* Select, and *MF* Select Codes Defined in Terms of *FS* Codes**

FS(3:0)	MF Select	G Select(3:0)	H Select(3:0)	Microoperation
0000	0	0000	XX	$F = A$
0001	0	0001	XX	$F = A + 1$
0010	0	0010	XX	$F = A + B$
0011	0	0011	XX	$F = A + B + 1$
0100	0	0100	XX	$F = A + \overline{B}$
0101	0	0101	XX	$F = A + \overline{B} + 1$
0110	0	0110	XX	$F = A - 1$
0111	0	0111	XX	$F = A$
1000	0	1X00	XX	$F = A \wedge B$
1001	0	1X01	XX	$F = A \vee B$
1010	0	1X10	XX	$F = A \oplus B$
1011	0	1X11	XX	$F = \overline{A}$
1100	1	XXXX	00	$F = B$
1101	1	XXXX	01	$F = \text{sr } B$
1110	1	XXXX	10	$F = \text{sl } B$

occur in the same clock cycle. A Write input corresponding to the Load Enable signal is also provided. When at 1, the Write signal permits registers to be loaded, during the current clock cycle, and, when at 0, prevents register loading. The size of the register file is $2^m \times n$, where m is the number of register address bits and n is the number of bits per register. For the datapath in Figure 10-1, $m = 2$, giving four registers, and n is unspecified.

Since the ALU and the shifter are shared processing units with outputs that are selected by MUX *F*, it is convenient to group the two units and the MUX together to form a shared function unit. Gray shading in Figure 10-1 highlights the function unit, which can be represented by the symbol given in Figure 10-10. The inputs to the function unit are from Bus *A* and Bus *B*, and the output of the function unit goes to MUX *D*. The function unit also has the four status bits *V*, *C*, *N*, and *Z* as added outputs.

In Figure 10-1, there are three sets of select inputs: the *G* select, *H* select, and *MF* select. In Figure 10-10, there is a single set of select inputs labeled *FS*, for "function select." To fully specify the function unit symbol in the figure, all of the codes for *MF* select, *G* select, and *H* select must be defined in terms of the codes for *FS*. Table 10-4 defines these code transformations. The codes for *FS* are given in the left column. From Table 10-4, it is apparent that *MF* is 1 for the leftmost two bits of *FS* both equal to 1. If *MF* select = 0, then the *G* select codes determine the function on the output of the function unit. If *MF* select = 1, then the *H* select codes determine the function on the output of the function unit. To show this dependency, the codes that determine the function unit outputs are highlighted in blue in the table. From Table 10-4, the code transformations can be implemented using the Boolean equations: $MF = F_3 \cdot F_2$, $G_3 = F_3$, $G_2 = F_2$, $G_1 = F_1$, $G_0 = F_0$, $H_1 = F_1$, and $H_0 = F_0$.

The status bits are assumed to be meaningless when the shifter is selected, although in a more complex system, shifter status bits can be designed to replace those for the ALU whenever a shifter microoperation is specified. Note that the status bit implementation depends on the specific implementation that has been used for the arithmetic circuit. Alternative implementations may not produce the same results.

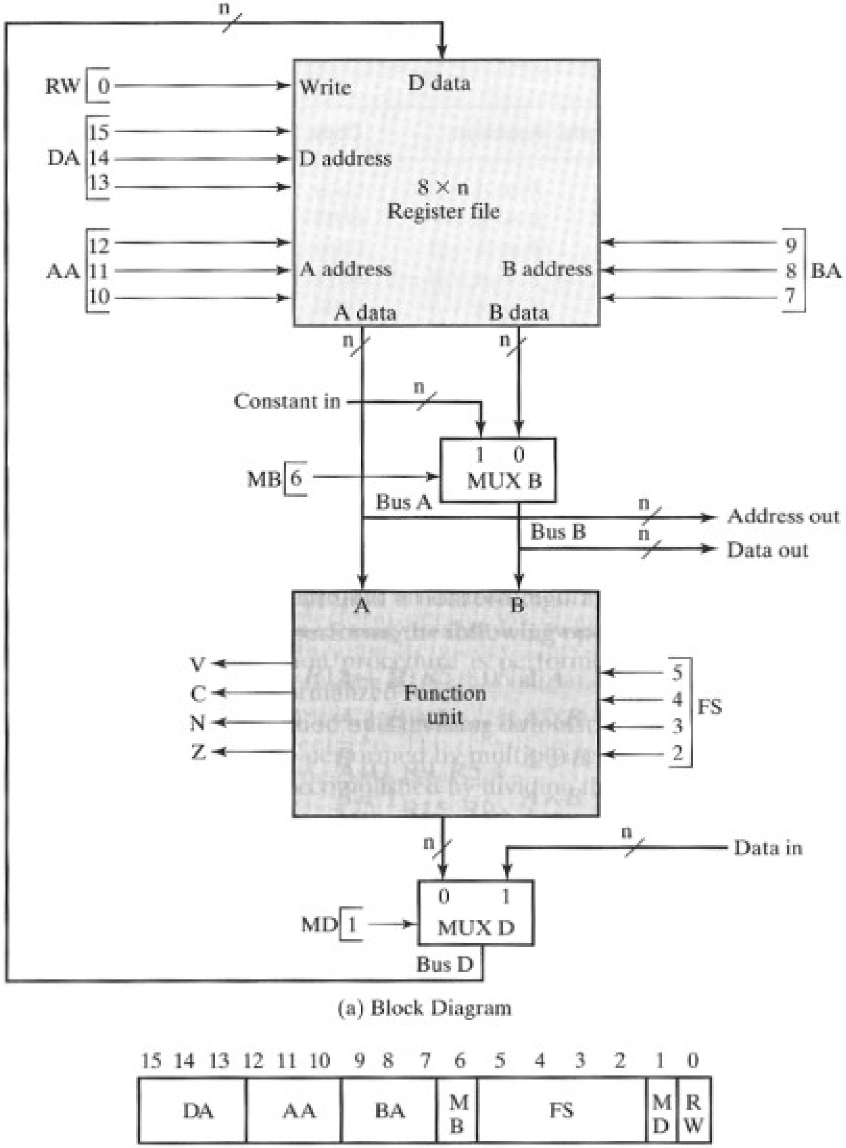
10-6 The Control Word

The selection variables for the datapath control the microoperations executed within the datapath for any given clock pulse. For the datapath in Section 10-5, the selection variables control the addresses for the data read from the register file, the function performed by the function unit, and the data loaded into the register file, as well as the selection of external data. We will now demonstrate how these control variables select the microoperations for the datapath. The choice of control variable values for typical microoperations will be discussed, and a simulation of the datapath will be illustrated.

A block diagram of a datapath that is a specific version of the datapath in Figure 10-10 is shown in Figure 10-11(a). It has a register file with eight registers, *R*0 through *R*7. The register file provides the inputs to the function unit through Bus *A* and Bus *B*. MUX *B* selects between constant values on Constant in and register values on *B* data. The ALU and zero-detection logic within the function unit generate the binary data for the four status bits: *V* (overflow), *C* (carry), *N* (sign), and *Z* (zero). MUX *D* selects the function unit output or the data on Data in as input for the register file.

There are 16 binary control inputs. Their combined values specify a *control word*. The 16-bit control word is defined in Figure 10-11(b). It consists of seven parts called *fields*, each designated by a pair of letters. The three register fields are three bits each. The remaining fields have one or four bits. The three bits of DA select one of eight destination registers for the result of the microoperation. The three bits of AA select one of eight source registers for the Bus *A* input to the ALU. The three bits of BA select a source register for the 0 input of the MUX *B*. The single MB bit determines whether Bus *B* carries the contents of the selected source register or a constant value. The 4-bit FS field controls the operation of the function unit. The FS field contains one of the 15 codes from Table 10-4. The single bit of MD selects the function unit output or the data on Data in as the input to Bus *D*. The final field, RW, determines whether a register is written or not. When applied to the control inputs, the 16-bit control word specifies a particular microoperation.

The functions of all meaningful control codes are specified in Table 10-5. For each of the fields, a binary code for each of the functions is given. The register selected by each of the fields DA, AA, and BA is the one with the decimal equivalent equal to the binary number for the code. MB selects either the register selected by the BA field or a constant from outside of the datapath on Constant in. The ALU operations, the shifter operations, and the selection of the ALU or



(a) Block Diagram

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DA			AA			BA			MB	FS				MD	RW

(b) Control word

□ **FIGURE 10-11**
Datapath with Control Variables

shifter outputs are all specified by the FS field. The field MD controls the information to be loaded into the register file. The final field, RW, has the functions No Write, to prevent writing to any registers, and Write, to signify writing to a register.

□ **TABLE 10-5**
Encoding of Control Word for the Datapath

DA, AA, BA		MB		FS		MD		RW	
Function	**Code**	**Function**	**Code**	**Function**	**Code**	**Function**	**Code**	**Function**	**Code**
$R0$	000	Register	0	$F = A$	0000	Function	0	No write	0
$R1$	001	Constant	1	$F = A + 1$	0001	Data In	1	Write	1
$R2$	010			$F = A + B$	0010				
$R3$	011			$F = A + B + 1$	0011				
$R4$	100			$F = A + \overline{B}$	0100				
$R5$	101			$F = A + \overline{B} + 1$	0101				
$R6$	110			$F = A - 1$	0110				
$R7$	111			$F = A$	0111				
				$F = A \wedge B$	1000				
				$F = A \vee B$	1001				
				$F = A \oplus B$	1010				
				$F = \overline{A}$	1011				
				$F = B$	1100				
				$F = \mathrm{sr}\ B$	1101				
				$F = \mathrm{sl}\ B$	1110				

The control word for a given microoperation can be derived by specifying the value of each of the control fields. For example, a subtraction given by the statement

$$R1 \leftarrow R2 + \overline{R3} + 1$$

specifies $R2$ for the A input of the ALU and $R3$ for the B input of the ALU. It also specifies function unit operation $F = A + \overline{B} + 1$ and selection of the function unit output for input into the register file. Finally, the microoperation selects $R1$ as the destination register and sets RW to 1 to cause $R1$ to be written. The control word for this microinstruction is specified by its seven fields, with the binary value for each field obtained from the encoding listed in Table 10-5. The binary control word for this subtraction microoperation, 001_010_011_0_0101_0_1, (with underline "_" used for convenience to separate the fields) is obtained as follows:

Field:	DA	AA	BA	MB	FS	MD	RW
Symbolic:	$R1$	$R2$	$R3$	Register	$F = A + \overline{B} + 1$	Function	Write
Binary:	001	010	011	0	0101	0	1

The control word for the microoperation and those for several other microoperations are given in Table 10-6 using symbolic notation and in Table 10-7 using binary codes.

The second example in Table 10-6 is a shift microoperation given by the statement

$$R4 \leftarrow \mathrm{sl}\ R6$$

□ **TABLE 10-6**
Examples of Microoperations for the Datapath, Using Symbolic Notation

Micro- operation	DA	AA	BA	MB	FS	MD	RW
$R1 \leftarrow R2 - R3$	$R1$	$R2$	$R3$	Register	$F = A + \overline{B} + 1$	Function	Write
$R4 \leftarrow$ sl R6	$R4$	—	$R6$	Register	$F = \text{sl } B$	Function	Write
$R7 \leftarrow R7 + 1$	$R7$	$R7$	—	Register	$F = A + 1$	Function	Write
$R1 \leftarrow R0 + 2$	$R1$	$R0$	—	Constant	$F = A + B$	Function	Write
Data out $\leftarrow R3$	—	—	$R3$	Register	—	—	No Write
$R4 \leftarrow$ Data in	$R4$	—	—	—	—	Data in	Write
$R5 \leftarrow 0$	$R5$	$R0$	$R0$	Register	$F = A \oplus B$	Function	Write

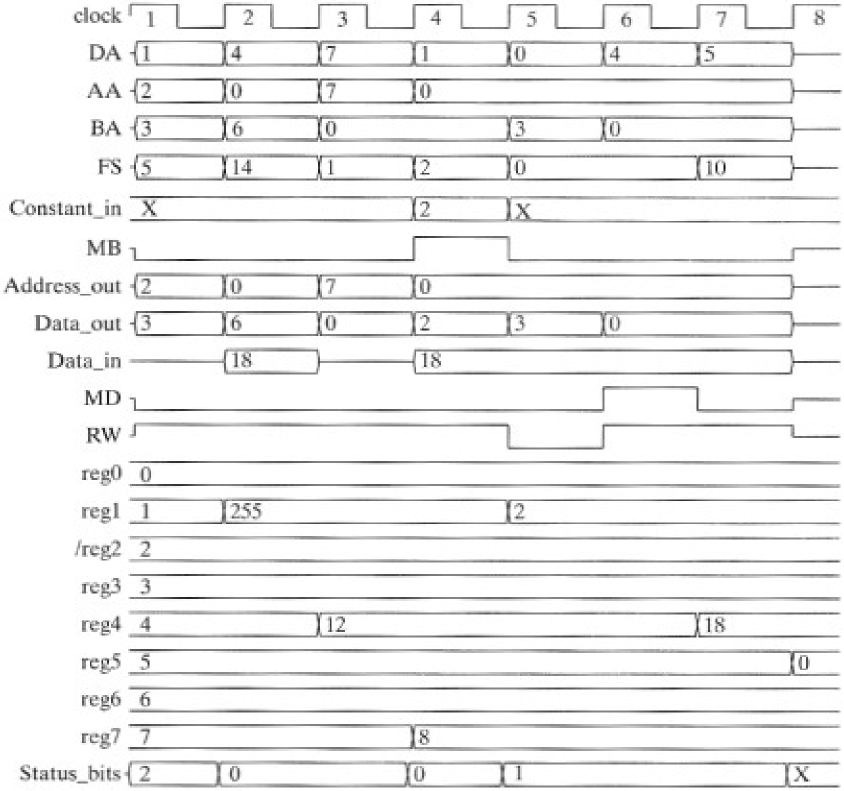
This statement specifies a shift left for the shifter. The content of register $R6$, shifted to the left, is transferred to $R4$. Note that because the shifter is driven by the B bus, the source for the shift is specified in the BA field rather than the AA field. From the knowledge of the symbols in each field, the control word in binary is derived as shown in Table 10-7. For many microoperations, neither the A data nor the B data from the register file is used. In these cases, the respective symbolic field is marked with a dash. Since these values are unspecified, the corresponding binary values in Table 10-7 are Xs. Continuing with the last three examples in Table 10-6, to make the contents of a register available to an external destination only, we place the contents of the register on the B data output of the register file, with RW = No Write (0) to prevent the register file from being written. To place a small constant in a register or use a small constant as one of the operands, we place the constant on Constant in, set MB to Constant, and pass the value from Bus B through the ALU and Bus D to the destination register. To clear a register to 0, Bus D is set to all 0's by using the same register for both A data and B data with an XOR operation specified (FS = 1010) and MD = 0. The DA field is set to the code for the destination register, and RW is Write (1).

□ **TABLE 10-7**
Examples of Microoperations from Table 10-6, Using Binary Control Words

Micro- operation	DA	AA	BA	MB	FS	MD	RW
$R1 \leftarrow R2 - R3$	001	010	011	0	0101	0	1
$R4 \leftarrow$ sl R6	100	XXX	110	0	1110	0	1
$R7 \leftarrow R7 + 1$	111	111	XXX	0	0001	0	1
$R1 \leftarrow R0 + 2$	001	000	XXX	1	0010	0	1
Data out $\leftarrow R3$	XXX	XXX	011	0	XXXX	X	0
$R4 \leftarrow$ Data in	100	XXX	XXX	X	XXXX	1	1
$R5 \leftarrow 0$	101	000	000	0	1010	0	1

It is apparent from these examples that many microoperations can be performed by the same datapath. Sequences of such microoperations can be realized by providing a control unit that produces the appropriate sequences of control words.

To complete this section, we perform a simulation of the datapath in Figure 10-11. The number of bits in each register, *n*, is equal to 8. An unsigned decimal representation, which is most convenient for reading the simulation output, is used for all multiple bit signals. We assume that the microoperations in Table 10-7, executed in sequence, provide the inputs to the datapath and that the initial content of each register is its number in decimal (e.g., *R*5 contains $(0000\ 0101)_2 = (5)_{10}$). Figure 10-12 gives the result of this simulation. The first value displayed is the Clock with the clock cycles numbered for reference. The inputs, outputs, and state for the datapath are given roughly in the order of the flow of information through the path. The first four inputs are the primary control word fields, which specify the register addresses that determine the register file outputs and the function selection. Next are inputs Constant in and MB, which control the input to Bus B.



□ **FIGURE 10-12**
Simulation of the Microoperation Sequence in Table 10-7

Following are the outputs Address out and Data out, which are the outputs from Bus A and Bus B, respectively. The next three variables—Data in, MD, and RW—are the final three inputs to the datapath. They are followed by the content of the eight registers and the Status bits, which are given as a vector (V, C, N, Z). The initial content of each register is its number in decimal. The value 2 is applied to Constant only in cycle 4 where MB equals 1. Otherwise, the value on Constant in is unknown as indicated by X. Finally, Data in has value 18. In the simulation, this value comes from a memory that is addressed by Address out and that has value 18 in location 0 with unknown values in all other locations. The resulting value, except when Address out is 0, is represented by a line midway between 0 and 1 indicating the value is unknown.

Of note in the simulation results is that changes in registers as a result of a particular microoperation appear in the clock cycle *after* that in which the microoperation is specified. For example, the result of the subtraction in clock cycle 1 appears in register *R*1 in clock cycle 2. This is because the result is loaded into flip-flops on the positive edge of the clock at the end of the clock cycle 1. On the other hand, the values on the Status bits, Address out, and Data out appear in the same clock cycle as the microoperation controlling them, since they do not depend on a positive clock edge occurring. Since there is no combinational delay specified in the simulation, these values change at the same time as the register values. Finally, note that eight clock cycles of simulation are used for seven microoperations so that the values in the registers that result from the last microoperation executed can be observed. Although Status bits appear for all microoperations, they are not always meaningful. For example, for the microoperations, R3 = Data out and R4 ← Data in, in clock cycles 5 and 6, respectively, the value of the status bits does not relate to the result since the Function unit is not used in these operations. Finally, for R5 ← R0 ⊕ R0 in clock cycle 7, the arithmetic unit is not used, so the values of V and C from that unit are irrelevant, but the values for N and Z do represent the status of the result as a signed 2's complement integer.

10-7 A Simple Computer Architecture

We introduce a simple computer architecture to obtain a beginning understanding of computer design and to illustrate control designs for programmable systems. In a programmable system, a portion of the input to the processor consists of a sequence of *instructions*. Each instruction specifies the operation the system is to perform, which operands to use for the operation, where to place the results of the operation and, in some cases, which instruction to execute next. For the programmable system, the instructions are usually stored in memory, which is either RAM or ROM. To execute the instructions in sequence, it is necessary to provide the address in memory of the instruction to be executed. In a computer, this address comes from a register called the *program counter* (*PC*). As the name implies, the *PC* has logic that permits it to count. In addition, to change the sequence of operations using decisions based on status information, the *PC* needs parallel load capability. So, in the case of a programmable system, the control unit contains a *PC* and associated decision logic, as well as the necessary logic to interpret the instruction in order to execute it. *Executing* an instruction means activating the necessary sequence of microoperations in the datapath (and elsewhere) required to perform the operation specified by the instruction. In contrast to the preceding, note that for a nonprogrammable system, the control unit is not responsible for obtaining instructions from memory, nor is it responsible for sequencing the execution of those instructions. There is no *PC* or similar register in such a system. Instead, the control unit determines the operations to be performed and the sequence of those operations, based on only its inputs and the status bits.

We show how the operations specified by instructions for the simple computer can be implemented by microoperations in the datapath, plus movement of information between the datapath and memory. We also show two different control structures for implementing the sequences of operations necessary for controlling program execution. The purpose here is to illustrate two different approaches to control design and the effects that such approaches have on datapath design and system performance. A more extensive study of the concepts associated with instruction sets for digital computers is presented in detail in the next chapter, and more complete CPU designs are undertaken in Chapter 12.

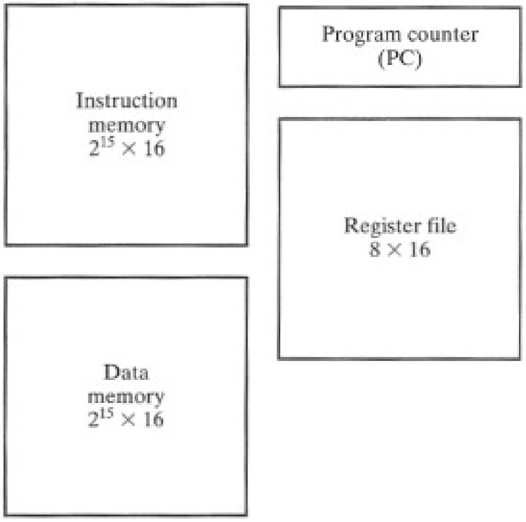
Instruction Set Architecture

The user specifies the operations to be performed and their sequence by the use of a *program*, which is a list of instructions that specifies the operations, the operands, and the sequence in which processing is to occur. The data processing performed by a computer can be altered by specifying a new program with different instructions or by specifying the same instructions with different data. Instructions and data are usually stored together in the same memory. By means of the techniques discussed in Chapter 12, however, they may appear to be coming from different memories. The control unit reads an instruction from memory and decodes and executes the instruction by issuing a sequence of one or more microoperations. The ability to execute a program from memory is the most important single property of a general-purpose computer. Execution of a program from memory is in sharp contrast to the nonprogrammable multiplier control unit considered earlier, which executes only a single, fixed operation.

An *instruction* is a collection of bits that instructs the computer to perform a specific operation. We call the collection of instructions for a computer its *instruction set* and a thorough description of the instruction set its *instruction set architecture* (*ISA*). Simple instruction set architectures have three major components: the storage resources, the instruction formats, and the instruction specifications.

Storage Resources

The storage resources for the simple computer are represented by the diagram in Figure 10-13. The diagram depicts the computer structure as viewed by a user programming it in a language that directly specifies the instructions to be executed. It


Instruction memory $2^{15} \times 16$
Program counter (PC)
Register file 8×16
Data memory $2^{15} \times 16$

□ **FIGURE 10-13**
Storage Resource Diagram for a Simple Computer

gives the resources the user sees available for storing information. Note that the architecture includes two memories, one for storage of instructions and the other for storage of data. These may actually be different memories, or they may be the same memory, but viewed as different from the standpoint of the CPU as discussed in Chapter 12. Also visible to the programmer in the diagram is a register file with eight 16-bit registers and the 16-bit program counter.

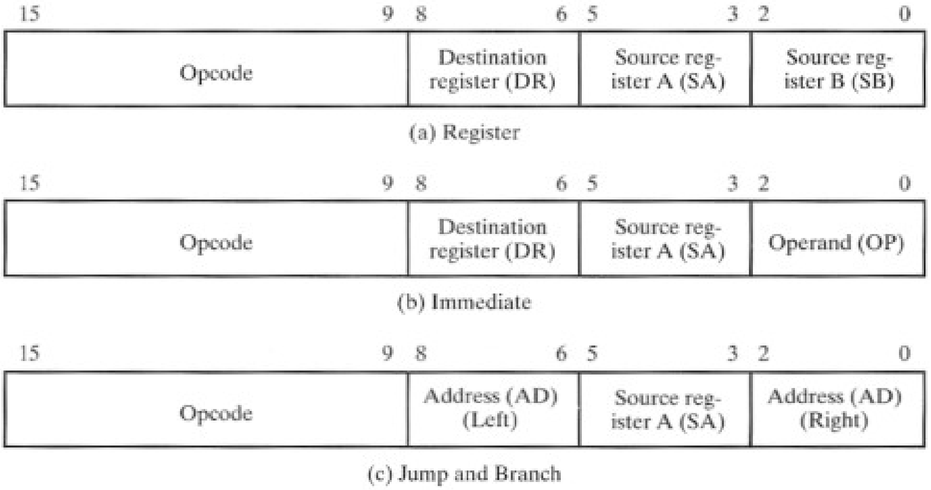
Instruction Formats

The format of an instruction is usually depicted by a rectangular box symbolizing the bits of the instruction, as they appear in memory words or in a control register. The bits are divided into groups or parts called *fields*. Each field is assigned a specific item, such as the operation code, a constant value, or a register file address. The various fields specify different functions for the instruction and, when shown together, constitute an instruction format.

The *operation code* of an instruction, often shortened to "opcode," is a group of bits in the instruction that specifies an operation, such as add, subtract, shift, or complement. The number of bits required for the opcode of an instruction is a function of the total number of operations in the instruction set. It must consist of at least m bits for up to 2^m distinct operations. The designer assigns a bit combination (a code) to each operation. The computer is designed to accept this bit configuration at the proper time in the sequence of activities and to supply the proper control word sequence to execute the specified operation. As a specific example, consider a computer with a maximum of 128 distinct operations, one of them an addition operation. The opcode assigned to this operation consists of seven bits 0000010. When the opcode 0000010 is detected by the control unit, a sequence of control words is applied to the datapath to perform the intended addition.

The opcode of an instruction specifies the operation to be performed. The operation must be performed using data stored in computer registers or in memory (i.e., on the contents of the storage resources). An instruction, therefore, must specify not only the operation, but also the registers or memory words in which the operands are to be found and the result is to be placed. The operands may be specified by an instruction in two ways. An operand is said to be specified *explicitly* if the instruction contains special bits for its identification. For example, the instruction performing an addition may contain three binary numbers specifying the registers containing the two operands and the register that receives the result. An operand is said to be defined *implicitly* if it is included as a part of the definition of the operation itself, as represented by the opcode, rather than being given in the instruction. For example, in an Increment Register operation, one of the operands is implicitly +1.

The three instruction formats for the simple computer are illustrated in Figure 10-14. Suppose that the computer has a register file consisting of eight registers, $R0$ through $R7$. The instruction format in Figure 10-14(a) consists of an opcode that specifies the use of three or fewer registers, as needed. One of the registers is designated a destination for the result and two of the registers sources for operands. For convenience, the field names are abbreviated DR, for "Destination Register," SA for "Source Register A," and SB for "Source Register B." The number of register fields and registers actually used are determined by the specific opcode. The opcode also specifies the use of the registers. For example, for a subtraction operation, suppose that the three bits in SA are 010, specifying $R2$, the three bits in SB



(a) Register

(b) Immediate

(c) Jump and Branch

□ **FIGURE 10-14**
Three Instruction Formats

are 011, specifying *R*3, and the three bits in DR are 001, specifying *R*1. Then the contents of *R*3 will be subtracted from the contents of *R*2, and the result will be placed in *R*1. As an additional example, suppose that the operation is a store (to memory). Suppose further, that the three bits in SA specify *R*4 and the three bits in SB specify *R*5. For this particular operation, it is assumed that the register specified in SA contains the address and the register specified in SB contains the operand to be stored. So the value in *R*5 is stored in the memory location given by the value in *R*4. The DR field has no effect, since the store operation prevents the register file from being written.

The instruction format in Figure 10-14(b), has an opcode, two register fields, and an operand. The operand is a constant called an *immediate operand*, since it is immediately available in the instruction. For example, for an add immediate operation with SA specified as *R*7, DR specified as *R*2, and operand OP equal to 011, the value 3 is added to the contents of *R*7, and the result of the addition is placed in *R*2. Since the operand is only three bits rather than a full 16 bits, the remaining 13 bits must be filled by using either zero fill or sign extension as discussed in Chapter 5. In this ISA, zero-fill is specified for the operand.

The instruction format in Figure 10-14(c), in contrast to the other two formats, does not change any register file or memory contents. Instead, it affects the order in which the instructions are fetched from memory. The location of an instruction to be fetched is determined by the program counter denoted by *PC*. Ordinarily, the program counter fetches the instructions from sequential addresses in memory as the program is executed. But much of the power of a processor comes from its ability to change the order of execution of the instructions based on results of the processing performed. These changes in the order of instruction execution are based on the use of instructions referred to as jumps and branches.

The example format given in Figure 10-14(c) for jump and branch instructions has an operation code, one register field SA, and a split address field AD. If a branch (possibly based on the contents of the register specified) is to occur, the new address is formed by adding the current PC contents and the contents of the 6-bit address field. This addressing method is called PC relative and the 6-bit address field, which is referred to as an *address offset* is treated as a signed two's complement number. To preserve the two's complement representation, *sign extension* is applied to the 6-bit address to form a 16-bit offset before the addition. If the leftmost bit of the address field AD is a 1, then the 10 bits to its left are filled with 1's to give a negative two's complement offset. If the leftmost bit of the address field is 0, then the 10 bits to its left are filled with 0's to give a positive two's complement offset. The offset is added to the contents of the *PC* to form the location from which the next instruction is to be fetched. For example, with the *PC* value equal to 55, suppose that a branch is to occur to location 35 if the contents of *R*6 is equal to zero. The opcode would specify a branch on zero instruction, SA would be specified as *R*6, and AD would be the 6-bit, two's complement representation of – 20. If *R*6 is zero, then *PC* contents becomes 55 + (– 20) = 35 and the next instruction would be fetched from address 35. Otherwise, if *R*6 is nonzero, the *PC* will count up to 56 and the next instruction will be fetched from address 56. This addressing method alone provides only branch addresses within a small range below and above the PC value. The jump provides a broader range of addresses by using the unsigned contents of a 16-bit register as the jump target.

The three formats in Figure 10-14 are used for the simple computer to be discussed in this chapter. In Chapter 11, we present and discuss more generally other instruction types and formats.

Instruction Specifications

Instruction specifications describe each of the distinct instructions that can be executed by the system. For each instruction, the opcode is given along with a shorthand name called a *mnemonic,* that can be used as a symbolic representation for the opcode. This mnemonic, along with a representation for each of the additional instruction fields in the format for the instruction, represents the notation to be used in specifying all of the fields of the instruction symbolically. This symbolic representation is then converted to the binary representation of the instruction by a program called an *assembler*. A description of the operation performed by the instruction execution is given, including the status bits that are affected by the instruction. This description may be text or may use a register transfer-like notation.

The instruction specifications for the simple computer are given in Table 10-8. The register transfer notation introduced in previous chapters is used to describe

□ **TABLE 10-8**
Instruction Specifications for the Simple Computer

Instruction	Opcode	Mnemonic	Format	Description	Status Bits
Move A	0000000	MOVA	RD,RA	$R[DR] \leftarrow R[SA]$	N, Z
Increment	0000001	INC	RD,RA	$R[DR] \leftarrow R[SA] + 1$	N, Z
Add	0000010	ADD	RD,RA,RB	$R[DR] \leftarrow R[SA] + R[SB]$	N, Z
Subtract	0000101	SUB	RD,RA,RB	$R[DR] \leftarrow R[SA] - R[SB]$	N, Z
Decrement	0000110	DEC	RD,RA	$R[DR] \leftarrow R[SA] - 1$	N, Z
AND	0001000	AND	RD,RA,RB	$R[DR] \leftarrow R[SA] \wedge R[SB]$	N, Z
OR	0001001	OR	RD,RA,RB	$R[DR] \leftarrow R[SA] \vee R[SB]$	N, Z
Exclusive OR	0001010	XOR	RD,RA,RB	$R[DR] \leftarrow R[SA] \oplus R[SB]$	N, Z
NOT	0001011	NOT	RD,RA	$R[DR] \leftarrow \overline{R[SA]}$	N, Z
Move B	0001100	MOVB	RD,RB	$R[DR] \leftarrow R[SB]$	
Shift Right	0001101	SHR	RD,RB	$R[DR] \leftarrow \text{sr } R[SB]$	
Shift Left	0001110	SHL	RD,RB	$R[DR] \leftarrow \text{sl } R[SB]$	
Load Immediate	1001100	LDI	RD, OP	$R[DR] \leftarrow \text{zf OP}$	
Add Immediate	1000010	ADI	RD,RA,OP	$R[DR] \leftarrow R[SA] + \text{zf OP}$	
Load	0010000	LD	RD,RA	$R[DR] \leftarrow M[SA]$	
Store	0100000	ST	RA,RB	$M[SA] \leftarrow R[SB]$	
Branch on Zero	1100000	BRZ	RA,AD	if $(R[SA] = 0)$ $PC \leftarrow PC + \text{se AD}$	
Branch on Negative	1100001	BRN	RA,AD	if $(R[SA] < 0)$ $PC \leftarrow PC + \text{se AD}$	
Jump	1110000	JMP	RA	$PC \leftarrow R[SA]$	

the operation performed, and the status bits that are valid for each instruction are indicated. In order to illustrate the instructions, suppose that we have a memory with 16 bits per word with instructions having one of the formats in Figure 10-14. Instructions and data, in binary, are placed in memory as shown in Table 10-9. This stored information represents the four instructions illustrating the distinct formats. At address 25, we have a register format instruction that specifies an operation to subtract *R*3 from *R*2 and load the difference into *R*1. This operation is represented symbolically in the rightmost column of Table 10-9. Note that the 7-bit opcode for subtraction is 0000101, or decimal 5. The remaining bits of the instruction specify the three registers: 001 specifies the destination register as *R*1, 010 specifies the source register *A* as *R*2, and 011 specifies the source register *B* as *R*3.

In memory location 35 is a register format instruction to store the contents of *R*5 in the memory location specified by *R*4. The opcode is 0100000, or decimal 32, and the operation is given symbolically, again, in the rightmost column of the figure. Suppose *R*4 contains 70 and *R*5 contains 80. Then the execution of this instruction will store the value 80 in memory location 70, replacing the original value of 192 stored there.

At address 45, an immediate format instruction appears that adds 3 to the contents of *R*7 and loads the result into *R*2. The opcode for this instruction is 66, and the operand to be added is the value 3 (011) in the OP field, the last three bits of the instruction.

In location 55, the branch instruction previously described appears. The opcode for this instruction is 96, and source register A is specified as *R*6. Note that AD (Left) contains 101 and AD (Right) contains 100. Putting these two together

□ **TABLE 10-9**
Memory Representation of Instructions and Data

Decimal Address	Memory Contents	Decimal Opcode	Other Fields	Operation
25	0000101 001 010 011	5 (Subtract)	DR:1, SA:2, SB:3	R1 ← R2 – R3
35	0100000 000 100 101	32 (Store)	SA:4, SB:5	M[R4] ← R5
45	1000010 010 111 011	66 (Add Immediate)	DR:2, SA:7, OP:3	R2 ← R7 + 3
55	1100000 101 110 100	96 (Branch on Zero)	AD: 44, SA:6	If R6 = 0, PC ← PC – 20
70	0000000011000000	Data = 192. After execution of instruction in 35, Data = 80.		

and applying sign extension, we obtain 1111111111101100, which represents – 20 in two's complement. If register *R*6 is zero, then – 20 is added to the *PC* to give 35. If register *R*6 is nonzero, the new *PC* value will be 56. It should be noted that we have assumed that the addition to the *PC* content occurs before the *PC* has been incremented which would be the case in the simple computer. In real systems, however, the *PC* has sometimes been incremented to point to the next instruction in memory. In such a case, the value stored in AD needs to be adjusted accordingly to obtain the right branch address.

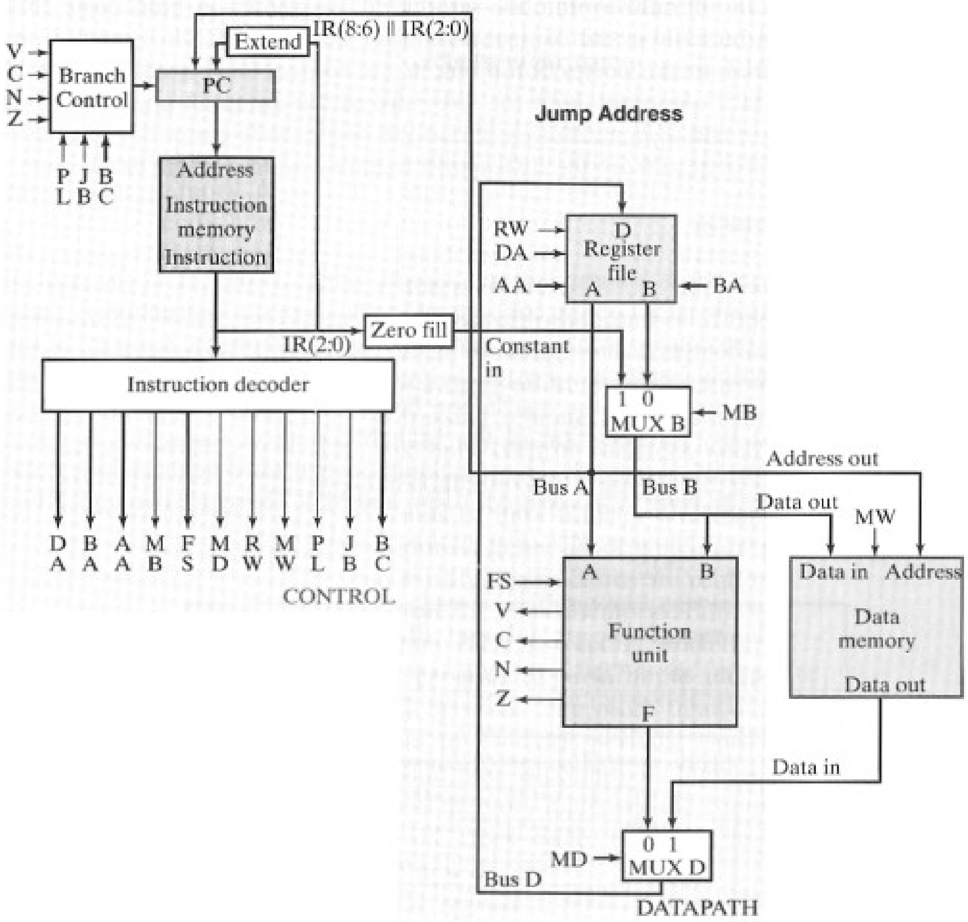
The placement of instructions in memory as shown in Table 10-9 is quite arbitrary. In many computers, the word length is from 32 to 64 bits, so the instruction formats can hold much larger immediate operands and addresses than those we have given. Depending on the computer architecture, some of the instruction formats may occupy two or more consecutive memory words. Also, the number of registers is often larger, so the register fields in the instructions must contain more bits.

At this point, it is vital to recognize the difference between a computer *operation* and a hardware *microoperation*. An operation is specified by an instruction stored in binary, in the computer's memory. The control unit in the computer uses the address or addresses provided by the program counter to retrieve the instruction from memory. It then decodes the opcode bits and other information in the instruction to perform the required microoperations for the execution of the instruction. In contrast, a microoperation is specified by the bits in a control word in the hardware which is decoded by the computer hardware to execute the microoperation. The execution of a computer operation often requires a sequence or program of microoperations, rather than a single microoperation.

10-8 Single-Cycle Hardwired Control

The block diagram for a computer that has a hardwired control unit and that fetches and executes an instruction in a single clock cycle is shown in Figure 10-15. We refer to this computer as the single-cycle computer. The storage resources, instruction formats, and instruction specifications for this computer are given in the previous section. The datapath shown is the same as that in Figure 10-11 with $m = 3$ and $n = 16$. The data memory *M* is attached to the Address out, Data out, and Data in by connections to the datapath. It has a single control signal MW which is 1 to write the memory, and 0 otherwise.

The Control unit appears on the left in Figure 10-15. Although not usually thought of as part of the control unit, the instruction memory, together with its address inputs and instruction outputs, is shown for convenience with the control unit. We do not write to the instruction memory, in theory, making it a combinational rather than a sequential component. As previously discussed, the *PC* provides the instruction address to the instruction memory, and the instruction output from the instruction memory goes to the control logic, which, in this case, is the instruction decoder. The output from the instruction memory also goes to Extend and Zero fill, which provide the address offset to the *PC* and the constant input, Constant in, to the datapath, respectively. Extension appends the leftmost bit of the 6-bit address offset field AD to the left of AD, preserving its two's complement



□ **FIGURE 10-15**
Block Diagram for a Single-Cycle Computer

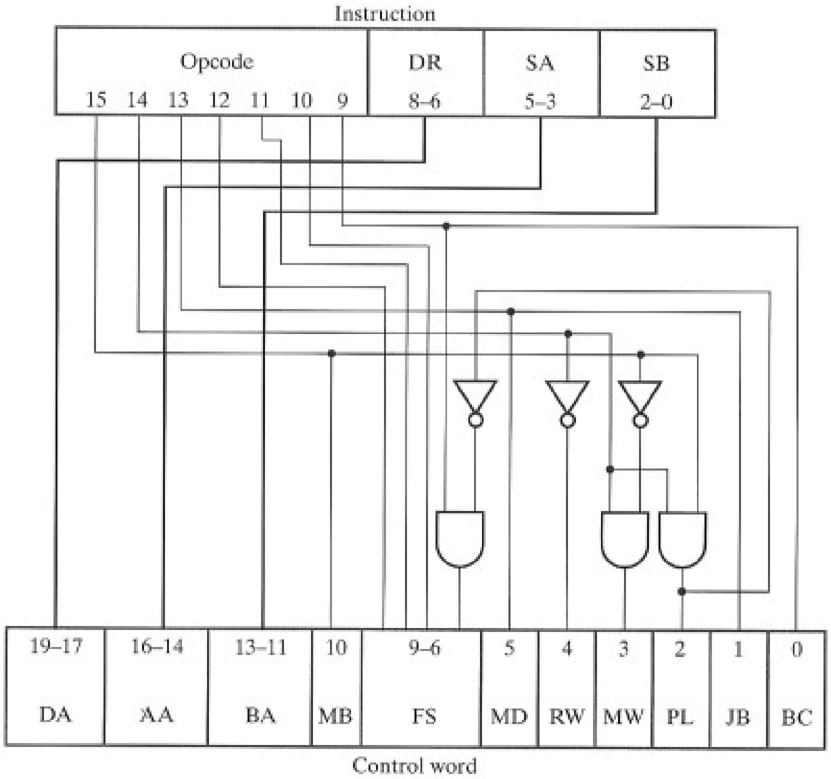
representation. Zero fill appends 13 zeros to the left of the operand (OP) field of the instruction to form a 16-bit unsigned operand for use in the datapath. For example, operand value 110 becomes 0000000000000110 or +6.

The *PC* is updated in each clock cycle. The behavior of the *PC*, which is a complex register, is determined by the opcode, N, and Z, since C and V are not used in this control unit design. If a jump occurs, the new *PC* value becomes the value on Bus *A*. If a branch is taken, then the new *PC* value is the sum of the previous *PC* value and the sign-extended address offset, which in two's complement can be either positive or negative. Otherwise, the *PC* is incremented by 1. A jump occurs for bit 13 in the instruction equal to 1. For bit 13 equal to 0, a conditional branch occurs. The status bit that is the condition for the branch is selected by bit 9 of the instruction. For bit 9 equal to 1, N is selected and, for bit 9 equal to 0, Z is selected.

All parts of the computer that are sequential are shown in blue. Note that there is no sequential logic in the control part other than the *PC*. Thus, aside from providing the address to the instruction memory, the control logic is combinational in this case. That fact, combined with the structure of the datapath and the use of separate instruction and data memories, allows the single-cycle computer to obtain and execute an instruction from the instruction memory, all in a single clock cycle.

Instruction Decoder

The instruction decoder is a combinational circuit that provides all of the control words for the datapath, based on the contents of the fields of the instruction. A number of the fields of the control word can be obtained directly from the contents of the fields in the instruction. Looking at Figure 10-16, we see that the control word fields DA, AA, and BA are equal to the instruction fields DR, SA, and SB, respectively. Also, control field BC for selection of the branch condition status bits is taken directly from the last bit of Opcode. The remaining control word fields include datapath and data memory control bits MB, MD, RW, and MW. There are two added bits



□ **FIGURE 10-16**
Diagram of Instruction Decoder

for the control of the PC, PL, and JB. If there is to be a jump or branch, PL = 1, loading the *PC*. For PL = 0, the *PC* is incremented. With PL = 1, JB = 1 calls for a jump, and JB = 0 calls for a conditional branch. Some of the single bit control word fields require logic for their implementation. In order to design this logic, we divide the various instructions possible for the simple computer into different function types and then assign the first three bits of the opcode to the various types. The instruction function types shown in Table 10-10 are based on the use of specific hardware resources in the computer, such as MUX *B*, the Function unit, the Register file, Data memory, and the *PC*. For example, the first function type uses the ALU, sets MUX *B* to use the Register file source, sets MUX *D* to use the Function unit output, and writes to the Register file. Other instruction function types are defined as various combinations of use of a constant input instead of a register, Data memory reads and writes, and manipulation of the *PC* for jumps and branches.

By looking at the relationship between the instruction function types and the necessary control word values needed for their implementation, bits 15 through 13 and bit 9 were assigned as shown in Table 10-10. This assignment attempted to minimize the logic required to implement the decoder. To perform the design of the decoder, the values for all of the single bit fields in the control word were determined from the function types and entered into Table 10-10. Note that there are a number of don't care (X) entries. Treating Table 10-10 as a truth-table and optimizing the logic functions, the logic for the single bit outputs of the instruction decoder in Figure 10-16 results. In the optimization, the four unused codes for bits 15, 14, 13, and 9 were assumed to have X values for all of the single bit fields. This implies that if one of these codes occurs in a program, the effect is unknown. A more conservative design specifies RW, MW, and PL all zero for these four codes to insure that the storage resource state is unchanged for these

□ TABLE 10-10
Truth Table for Instruction Decoder Logic

	Instruction Bits				Control Word Bits						
Instruction Function Type	**15**	**14**	**13**	**9**	**MB**	**MD**	**RW**	**MW**	**PL**	**JB**	**BC**
Function unit operations using registers	0	0	0	X	0	0	1	0	0	X	X
Memory read	0	0	1	X	0	1	1	0	0	X	X
Memory write	0	1	0	X	0	X	0	1	0	X	X
Function unit operations using register and constant	1	0	0	X	1	0	1	0	0	X	X
Conditional branch on zero (Z)	1	1	0	0	X	X	0	0	1	0	0
Conditional branch on negative (N)	1	1	0	1	X	X	0	0	1	0	1
Unconditional Jump	1	1	1	X	X	X	0	0	1	1	X

unused codes. The optimization results in the logic in Figure 10-16 for implementing MB, MD, RW, MW, PL, and JB.

The remaining logic in the decoder deals with the FS field. For all but the conditional branch and unconditional jump instructions, bits 9 through 12 are fed directly through to form the FS field. During conditional branch operations, such as Branch on Zero, the value in source register A must be passed through the ALU so that the status bits N and Z can be evaluated. This requires FS = 0000. The use of bit 9, however, for status bit selection for conditional branches, requires at times that bit 9, which controls the rightmost bit of FS, be a 1. The contradiction in values between bit 9 and FS is resolved by adding an enable on bit 9 that forces FS_0 to zero whenever PL = 1 as shown in Figure 10-16.

Sample Instructions and Program

Six instructions for the single-cycle computer are listed in Table 10-11. The symbolic names associated with the instructions are useful for listing programs in symbolic form rather than in binary code. Because of the importance of instruction decoding, the rightmost six columns of the table show critical control signal values for each instruction, based on the values obtained using the logic in Figure 10-16.

Now suppose that the first instruction, "Add Immediate" (ADI), is present on the output of the instruction memory shown in Figure 10-15. Then, on the basis of the first three bits of the opcode, 100, the outputs of the instruction decoder have the values MB = 1, MD = 0, RW = 1, and MW = 0. The last three bits of the instruction, $OP_{2\text{-}0}$, are extended to 16 bits by zero fill. We denote this in a register transfer statement by zf. Since MB is 1, this zero-filled value is placed on Bus *B*. With MD equal to 0, the function unit output is selected, and since the last four bits of the opcode, 0010, specify field FS, the operation is $A + B$. So the zero-filled value on Bus *B* is added to the contents of register SA, with the result presented on Bus *D*. Since RW = 1, the value on Bus *D* is written into register DR. Finally, with MW = 0, no write into memory occurs. This entire operation takes place in a single clock cycle. At the beginning of the next cycle, the destination register is written and, since PL = 0, the *PC* is incremented to point to the next instruction.

The second instruction, LD, is a load from memory with opcode 0010000. The first three bits of this opcode, 001, give control values MD = 1, RW = 1, and MW = 0. These values, plus the register source field SA and register destination field DR, fully specify this instruction, which loads the contents of the memory address specified by register SA into register DR. Again, since PL = 0, the *PC* is incremented. Note that the values of JB and BC are ignored, since this is neither a jump nor a branch instruction.

The third instruction, ST, stores the contents of a register in memory. The first three bits of the opcode, 010, give control signal values MB = 0, RW = 0, and MW = 1. MW = 1 causes a memory write operation, with the address and data from the register file. RW = 0 prevents the register file from being written. The address for the memory write comes from the register selected by field SA, and the data for the memory write come from the register selected by SB, since MB = 0. The DR field, although present, is not used, since no write occurs to a register.

□ **TABLE 10-11**
Six Instructions for the Single-Cycle Computer

Operation code	Symbolic name	Format	Description	Function	MB	MD	RW	MW	PL	JB	BC
1000010	ADI	Immediate	Add immediate operand	$R[\mathrm{DR}] \leftarrow R[\mathrm{SA}] + \mathrm{zf}\ I(2{:}0)$	1	0	1	0	0	0	0
0010000	LD	Register	Load memory content into register	$R[\mathrm{DR}] \leftarrow M[R[\mathrm{SA}]]$	0	1	1	0	0	1	0
0100000	ST	Register	Store register content in memory	$M[R[\mathrm{SA}]] \leftarrow R[\mathrm{SB}]$	0	1	0	1	0	0	0
0001110	SL	Register	Shift left	$R[\mathrm{DR}] \leftarrow \mathrm{sl}R[\mathrm{SB}]$	0	0	1	0	0	1	0
0001011	NOT	Register	Complement register	$R[\mathrm{DR}] \leftarrow \overline{R[\mathrm{SA}]}$	0	0	1	0	0	0	1
1100000	BRZ	Jump/Branch	If $R[\mathrm{SA}] = 0$, branch to PC + se AD	If $\mathrm{R[SA]} = 0$, $PC \leftarrow PC + \mathrm{se}AD$, If $\mathrm{R[SA]} \neq 0, PC \leftarrow PC + 1$	1	0	0	0	1	0	0

Because this computer has load and store instructions and does not combine loading and storing of data operands with other operations, it is referred to as having a *load/store* architecture. The use of such an architecture simplifies the execution of instructions.

The next two instructions use the Function unit and write to the Register file without immediate operands. The last four bits of the opcode, the value for the FS field of the control word, specify Function unit operation. For these two instructions, only one source register, R[SA] for the NOT and R[SB] for the shift left, and a destination register are involved.

The final instruction is a conditional branch and manipulates the *PC* value. It has PL = 1, causing the program counter to be loaded instead of incremented, and JB = 0, causing a conditional branch rather than a jump. Since BC = 0, register R[SA] is tested for a value of zero. If R[SA] equals zero, the *PC* value becomes *PC* + se AD, where se stands for sign extend. Otherwise, *PC* is incremented. For this instruction, the DR and SB fields become the 6-bit address field AD, which is sign extended and added to the *PC*.

To demonstrate how instructions such as these can be used in a simple program, consider the arithmetic expression 83 − (2 + 3). The following program performs this computation, assuming that register *R*3 contains 248, location 248 in data memory contains 2, location 249 contains 83, and the result is to be placed in location 250:

```
LD    R1, R3        Load R1 with contents of location 248 in memory (R1 = 2)
ADI   R1, R1, 3     Add 3 to R1 (R1 = 5)
NOT   R1, R1        Complement R1
INC   R1, R1        Increment R1 (R1 = −5)
INC   R3, R3        Increment the contents of R3 (R3 = 249)
LD    R2, R3        Load R2 with contents of location 249 in memory (R2 = 83)
ADD   R2, R2, R1    Add contents of R1 to contents of R2 (R2 = 78)
INC   R3, R3        Increment the contents of R3 (R3 = 250)
ST    R3, R2        Store R2 in memory location 250 (M[250] = 78)
```

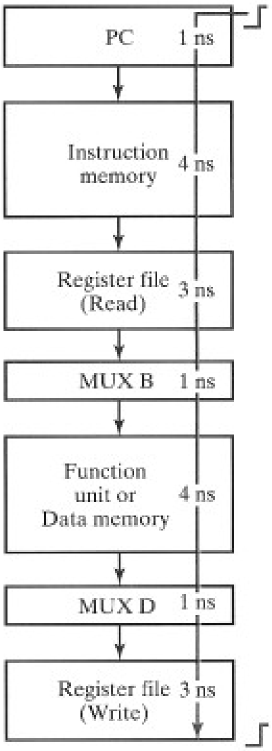
The subtraction in this case is done by taking the 2's complement of (2 + 3) and adding it to 83; the subtraction operation SUB could have been used as well. If a register field is not used in executing an instruction, its symbolic value is omitted. The symbolic values for the register-type instructions, when the latter are present, are in the order DR, SA, and SB. For immediate types, the fields are in the order DR, SA, and OP. To store this program in the instruction memory, it is necessary to convert all of the symbolic names and decimal numbers used to their corresponding binary codes.

Single-Cycle Computer Issues

Although there may be instances in which single-cycle computer timing and control strategy is useful, it has a number of shortcomings. One shortcoming is in the area of performing complex operations. For example, suppose that an instruction is desired that executes unsigned binary multiplication using an add-and-shift algorithm. With the given datapath, this cannot be accomplished by a microoperation that can be executed in a single clock cycle. Thus, a control organization that provides multiple clock cycles for the execution of instructions is needed.

Also, the single-cycle computer has two distinct 16-bit memories, one for instructions and one for data. For a simple computer with instructions and data in the same 16-bit memory, two read accesses of memory are required to execute an instruction that loads a data word from memory into a register. The first access obtains the instruction, and the second access, if required, reads or writes the data word. Since two different addresses must be applied to the memory address inputs, at least two clock cycles, one for each address, are required for obtaining and executing the instruction. This can also be accomplished easily with multiple-cycle control.

Finally, the single-cycle computer has a lower limit on the clock period based on a long worst case delay path. This path is shown in blue in the simplified diagram of Figure 10-17. The total delay along the path is 17 ns. This limits the clock frequency to 58.8 MHz, which, although it may be adequate for some applications,



□ **FIGURE 10-17**
Worst Case Delay Path in Single-Cycle Computer

is too slow for a modern computer CPU. In order to have a higher clock frequency, either the delays of the components on the path or the number of components in the path must be reduced. If the delays of the components cannot be reduced, reducing the number of components in the path is the only alternative. In Chapter 12, pipelining of the datapath reduces the number of components in the longest combinational delay path and permits the clock frequency to be increased. A pipelined datapath and control given in Chapter 12, demonstrates the improved CPU performance that can be obtained.

10-9 Multiple-Cycle Hardwired Control

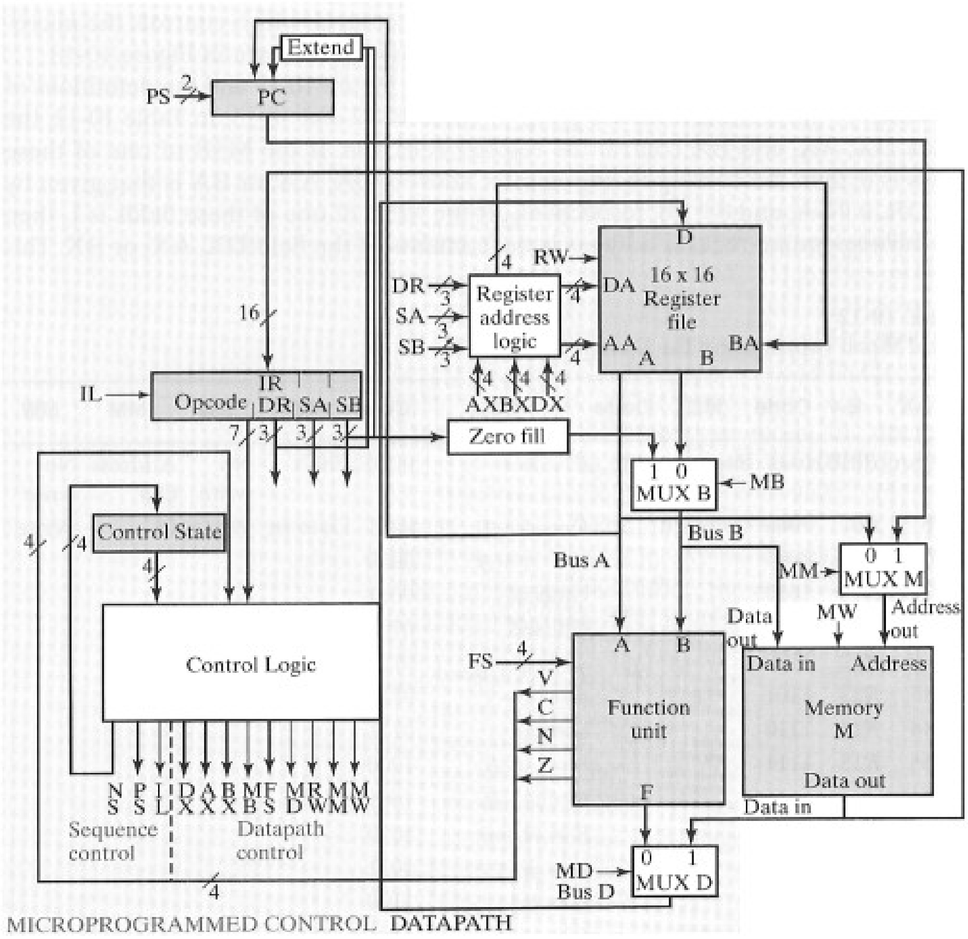
To demonstrate multiple-cycle control, we use the architecture of the simple computer, but modify its datapath, memory, and control. The goal of the modifications is to demonstrate the use of a single memory for both data and instructions and to demonstrate how more complex instructions can be implemented by using multiple clock cycles per instruction. The block diagram in Figure 10-18 shows the modifications to the datapath, memory, and control.

The changes to the single-cycle computer can be observed by comparing Figures 10-15 and 10-18. The first modification, which is possible with, but not essential to, multiple-cycle operation, replaces the separate instruction memory and data memory in Figure 10-15 with the single Memory *M* in Figure 10-18. To fetch instructions, the *PC* is the address source for the memory, and to fetch data, Bus *A* is the address source. At the address input to memory, multiplexer MUX *M* selects between these two address sources. MUX *M* requires an additional control signal, MM, which is added to the control word format. Since instructions from Memory M are needed in the control unit, a path is added from its output to the instruction register IR in the control unit.

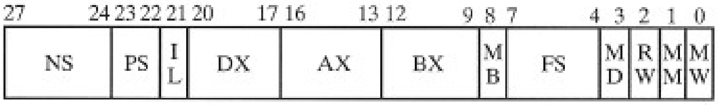
In executing an instruction across multiple clock cycles, data generated during the current cycle is often needed in a later cycle. This data can be temporarily stored in a register from the time it is generated until the time it is used. Registers used for such temporary storage during the execution of the instruction are usually not visible to the user (i.e., are not part of the storage resources). The second modification provides these temporary storage registers by doubling the number of registers in the register file. Registers 0 through 7 are storage resources and registers 8 through 15 are used only for temporary storage during instruction execution, so are not part of the storage resources visible to the user. The addressing of 16 registers requires 4 bits, and becomes more complex, since addressing of the first eight registers must be controlled from the instruction and the control unit, and the second eight registers are controlled from the control unit. This is handled by the Register address logic in Figure 10-18 and by modified DX, AX, and BX fields in the control word. The details of this change will be discussed later when the control is defined.

The *PC* is the only control unit component retained and it must also be modified. During the execution of a multiple-cycle instruction, the *PC* must be held at its current value for all but one of the cycles. To provide this hold capability, as well as an increment and two load operations, the *PC* is modified to be controlled by a 2-bit control word field, PS. Since the *PC* is controlled completely by the control word, the Branch control logic previously represented by BC is absorbed into the Control Logic block in Figure 10-18.

Because of the multiple cycles of the modified computer, the instruction needs to be held in a register for use during its execution since its values are likely to be needed for more than just the first cycle. The register used for this purpose is the *instruction register IR* in Figure 10-18. Since the *IR* loads only when an instruction is being read from memory, it has a load-enable signal IL that is added to the control word. Because of the multiple-cycle operation, a sequential control circuit, which can provide a sequence of control words for microoperations used



□ **FIGURE 10-18**
Block Diagram for a Multiple-Cycle Computer



□ **FIGURE 10-19**
Control Word Format for Multiple-Cycle Computer

to interpret the instruction is required and replaces the Instruction decoder. The sequential control unit consists of the Control state register and the combinational Control logic. The Control logic has the state, the opcode, and the status bits as its inputs and produces the control word as its output. Conceptually, the control word is divided into two parts, one for Sequence control, which determines the next state of the overall control unit, and one for Datapath control, which controls the microoperations executed by the Datapath and Memory *M* as shown in Figure 10-18.

The 28-bit modified control word is given in Figure 10-19 and the definitions of the fields of the control word are given in Table 10-12 and 10-13. In Table 10-12, the fields DX, AX, and BX control the register selection. If the MSB of one of these fields is 0, then the corresponding register address DA, AA, or BA is that given by 0 || DR, 0 || SA, and 0 || SB, respectively. If the MSB of one of these fields is 1, then the corresponding register address is the contents of the field DX, AX, or BX. This

□ **TABLE 10-12**
Control Word Information for Datapath

DX	AX	BX	Code	MB	Code	FS	Code	MD	RW	MM	MW	Code
R[DR]	*R*[SA]	*R*[SB]	0XXX	Register	0	$F = A$	0000	FnUt	No write	Address Out	No write	0
*R*8	*R*8	*R*8	1000	Constant	1	$F = A + 1$	0001	Data In	Write	PC	Write	1
*R*9	*R*9	*R*9	1001			$F = A + B$	0010					
*R*10	*R*10	*R*10	1010			Unused	0011					
*R*11	*R*11	*R*11	1011			Unused	0100					
*R*12	*R*12	*R*12	1100			$F = A + \overline{B} + 1$	0101					
*R*13	*R*13	*R*13	1101			$F = A - 1$	0110					
*R*14	*R*14	*R*14	1110			Unused	0111					
*R*15	*R*15	*R*15	1111			$F = A \wedge B$	1000					
						$F = A \vee B$	1001					
						$F = A \oplus B$	1010					
						$F = \overline{A}$	1011					
						$F = B$	1100					
						$F = \text{sr } B$	1101					
						$F = \text{sl } B$	1110					
						Unused	1111					

selection process is performed by the Register address logic, which contains three multiplexers, one for each of DA, AA, and BA, controlled by the MSB of DX, AX, and BX, respectively. Table 10-12 also gives the code values for the MM field, which determines whether Address out or PC serves as the Memory *M* address. The remaining fields in Table 10-12, MB, MD, RW, and MW, have the same functions as for the single-cycle computer.

In the sequential control circuit, the State control register has a set of states, just as a set of flip-flops in any other sequential circuit, has. At the level of our discussion, we assume that each state has an abstract name which can be used as both the state and the next state value. In the design process, a state assignment needs to be made to these abstract states. Referring to Table 10-13, the field NS in the control word provides the next state for the Control State register. We have assigned four bits for the state code, but this can be modified as necessary depending on the number of states needed and the state assignment used in the design. This particular field could be considered as integral to the control and sequential circuit and not part of the control word, but it will appear in the state table of the control in any case. The 2-bit PS field controls the program counter, *PC*. On a given clock cycle the *PC* holds its state (00), increments its state by 1 (01), conditionally loads *PC* plus sign-extended AD (10), or unconditionally loads the contents of *R*[SA] (11). Finally, the instruction register is loaded only once during the execution of an instruction. Thus, on any given cycle, either a new instruction is loaded (IL = 1) or the instruction remains unchanged (IL = 0).

Sequential Control Design

The design of the sequential control circuit can be done using techniques from Chapter 6 and Chapter 8. However, compared to the examples there, even for this comparatively simple computer, the control is quite complex. Assuming there are four state variables, the combinational Control logic has 15 input variables and 28 output variables. It turns out that a condensed state table for the circuit is not too difficult to develop, but manual design of the detailed logic is very complex, making the use of a PLA or logic synthesis more viable options. As a consequence, we focus on state table development rather than detailed logic implementation

We begin by developing an ASM chart that represents the instructions that can be implemented with the minimum number of clock cycles. Extensions of this

□ **TABLE 10-13**
Control Information for Sequence Control

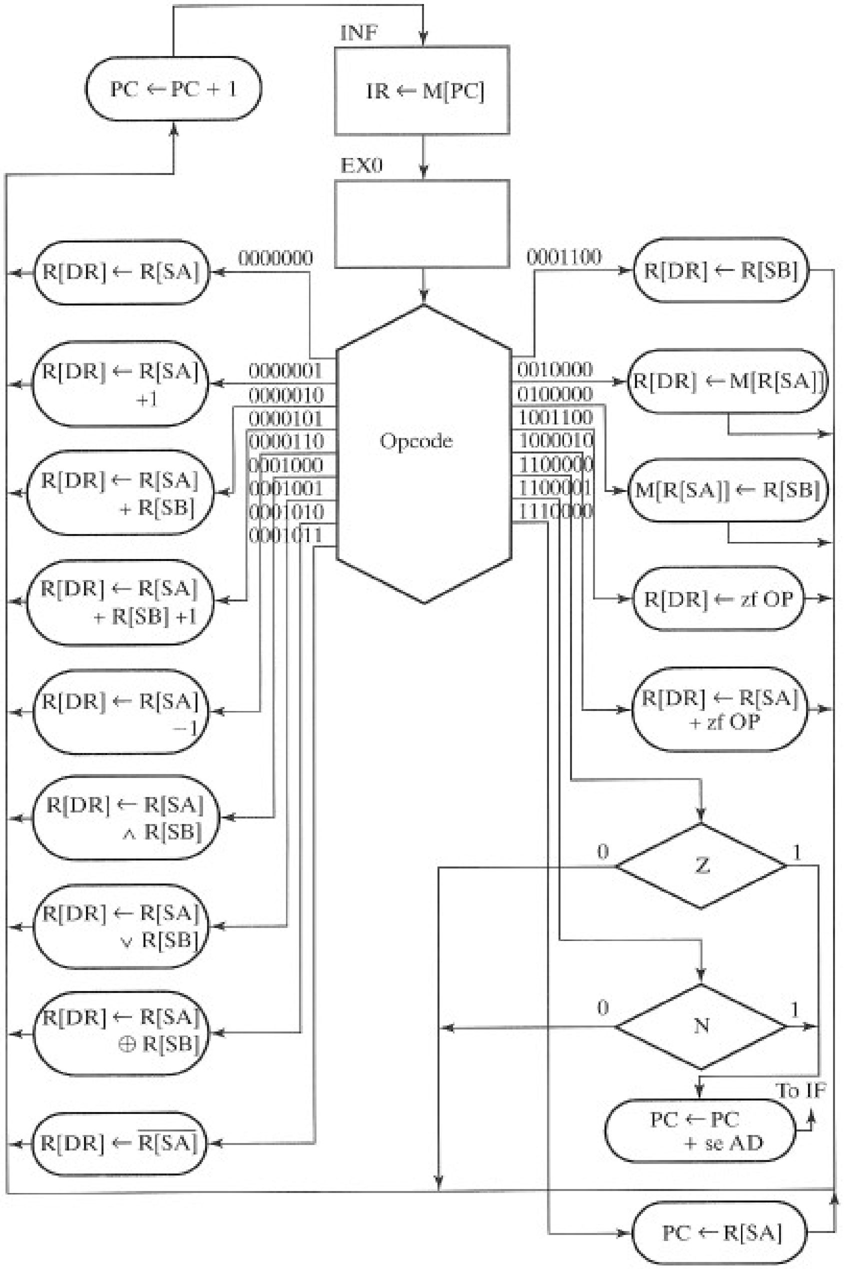
NS	PS		IL	
Next State	Action	Code	Action	Code
Gives next state of Control State Register	Hold PC	00	No load	0
	Inc PC	01	Load instr.	1
	Branch	10		
	Jump	11		

chart can then be developed for implementation of instructions requiring more than the minimum number of clock cycles. The ASM charts provide the information needed to develop the state table entries for implementing the instruction set. For instructions requiring a memory access for data as well as for the instruction itself, at least two cycles are required. It is convenient to separate the cycles into two processing steps: *instruction fetch* and *instruction execution*. On the basis of this division, the ASM chart for the two-cycle instructions is given in Figure 10-20. The instruction fetch occurs in state INF at the top of the chart. The *PC* contains the address of the instruction in Memory *M*. This address is applied to the memory, and the word read from memory is loaded into the *IR* on the clock pulse that ends state INF. The same clock pulse causes the new state to become EX0. In state EX0, the instruction is decoded by use of a large vector decision box and the microoperations executing all or part of the instruction appears in a conditional output box. If the instruction can be completed in state EX0, the next state is INF in preparation for fetching of the next instruction. Further, for instructions that do not change *PC* contents during their execution, the *PC* is incremented. If additional states are required for instruction execution, the next state is EX1. In each of the execution states, there are 128 different input combinations possible, based on the opcode. When the status bits are used, typically one at a time, the output of the vector decision box feeds one or more scalar decision boxes as illustrated for the branch instructions on the lower right of Figure 10-20.

Next, we describe a sampling of the instruction executions specified by the ASM chart in Figure 10-20. The first opcode is 0000000 for the move A, (MOVA) instruction. This instruction involves a simple transfer from the source A register to the destination register, as specified by the register transfer shown in state EX0 for the instruction opcode. Although the status bits *N* and *Z* are valid, they are not used in the execution of this instruction. The *PC* is incremented on the clock edge ending state EX0, an action that occurs for all but branch and jump instructions in the ASM chart.

The third opcode is 0000010 for the ADD instruction with the register transfer for addition shown. In this case, status bits *V*, *C*, *N*, and *Z* are valid, although not used. The eleventh opcode, 0010000, is the load (LD) instruction, which uses the value in the register specified by SA for the address and loads the data word from Memory *M* into the register specified by DR. The twelfth opcode, 0100000, is for the store (ST) instruction, which stores the value in register SB into the location in Memory *M* specified by the address from register SA. The fourteenth opcode, 1001100, is add immediate (ADI), which adds the zero-filled value of the OP field, the rightmost three bits of the instruction, to the contents of register SA and places the result in the register DR.

The sixteenth opcode, 1100001, is the branch on negative (BRN) instruction. The decoding of this instruction causes the value in the register specified by SA to be passed through the Function unit in order to evaluate status bits N and Z. The values N and Z then propagate back to the Control logic. Based on the value of N, the branch is taken or not taken by adding the extended address AD from the instruction to the value in the *PC* or incrementing the *PC*, respectively. This is represented by the scalar decision box for N shown in Figure 10-20.



□ **FIGURE 10-20**
Basic ASM Chart for Multiple-Cycle Computer

□ TABLE 10-14
State Table for Two-Cycle Instructions

State	Inputs		Next state	Outputs												
	Opcode	VCNZ		IL	PS	DX	AX	BX	MB	FS	MD	RW	MM	MW		Comments
INF	XXXXXXX	XXXX	EX0	1	00	XXXX	XXXX	XXXX	X	XXXX	X	0	1	0		$IR \leftarrow M[PC]$
EX0	0000000	XXXX	INF	0	01	0XXX	0XXX	XXXX	X	0000	0	1	X	0	MOVA	$R[\mathrm{DR}] \leftarrow R[\mathrm{SA}]$*
EX0	0000001	XXXX	INF	0	01	0XXX	0XXX	XXXX	X	0001	0	1	X	0	INC	$R[\mathrm{DR}] \leftarrow R[\mathrm{SA}] + 1$*
EX0	0000010	XXXX	INF	0	01	0XXX	0XXX	0XXX	0	0010	0	1	X	0	ADD	$R[\mathrm{DR}] \leftarrow R[\mathrm{SA}] + R[\mathrm{SB}]$*
EX0	0000101	XXXX	INF	0	01	0XXX	0XXX	0XXX	0	0101	0	1	X	0	SUB	$R[\mathrm{DR}\} \leftarrow R[\mathrm{SA}] + \overline{R[\mathrm{SB}]} + 1$*
EX0	0000110	XXXX	INF	0	01	0XXX	0XXX	XXXX	X	0110	0	1	X	0	DEC	$R[\mathrm{DR}] \leftarrow R[\mathrm{SA}] + (-1)$*
EX0	0001000	XXXX	INF	0	01	0XXX	0XXX	0XXX	0	1000	0	1	X	0	AND	$R[\mathrm{DR}] \leftarrow R[\mathrm{SA}] \wedge R[\mathrm{SB}]$*
EX0	0001001	XXXX	INF	0	01	0XXX	0XXX	0XXX	0	1001	0	1	X	0	OR	$R[\mathrm{DR}] \leftarrow R[\mathrm{SA}] \vee R[\mathrm{SB}]$*
EX0	0001010	XXXX	INF	0	01	0XXX	0XXX	0XXX	0	1010	0	1	X	0	XOR	$R[\mathrm{DR}] \leftarrow R[\mathrm{SA}] \oplus R[\mathrm{SB}]$*
EX0	0001011	XXXX	INF	0	01	0XXX	0XXX	XXXX	X	1011	0	1	X	0	NOT	$R[\mathrm{DR}] \leftarrow \overline{\mathrm{R[SA]}}$ *
EX0	0001100	XXXX	INF	0	01	0XXX	XXXX	0XXX	0	1100	0	1	X	0	MOVB	$R[\mathrm{DR}] \leftarrow \mathrm{R[SB]}$*
EX0	0010000	XXXX	INF	0	01	0XXX	0XXX	XXXX	X	XXXX	1	1	0	0	LD	$R[\mathrm{DR}] \leftarrow M[R[\mathrm{SA}]]$*
EX0	0100000	XXXX	INF	0	01	XXXX	0XXX	0XXX	0	XXXX	X	0	0	1	ST	$M[R[\mathrm{SA}]] \leftarrow R[\mathrm{SB}]$*
EX0	1001100	XXXX	INF	0	01	0XXX	XXXX	XXXX	1	1100	0	1	0	0	LDI	$R[\mathrm{DR}] \leftarrow$ zf OP*
EX0	1000010	XXXX	INF	0	01	0XXX	0XXX	XXXX	1	0010	0	1	0	0	ADI	$R[\mathrm{DR}] \leftarrow R[\mathrm{SA}] +$ zf OP*
EX0	1100000	XXX1	INF	0	10	XXXX	0XXX	XXXX	X	0000	X	0	0	0	BRZ	$PC \leftarrow PC +$ se AD
EX0	1100000	XXX0	INF	0	01	XXXX	0XXX	XXXX	X	0000	X	0	0	0	BRZ	$PC \leftarrow PC + 1$
EX0	1100001	XX1X	INF	0	10	XXXX	0XXX	XXXX	X	0000	X	0	0	0	BRN	$PC \leftarrow PC +$ se AD
EX0	1100001	XX0X	INF	0	01	XXXX	0XXX	XXXX	X	0000	X	0	0	0	BRN	$PC \leftarrow PC + 1$
EX0	1110000	XXXX	INF	0	11	XXXX	0XXX	XXXX	X	0000	X	0	0	0	JMP	$PC \leftarrow R[\mathrm{SA}]$

* For this state and input combination, PC ← PC + 1 also occurs.

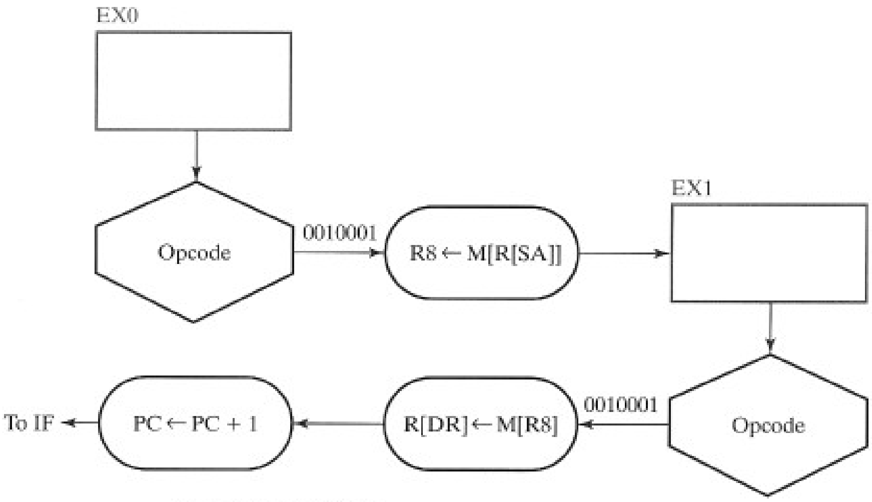
From this ASM chart, the state table for the sequential control circuit can be developed as shown in Table 10-14. The present states are given as abstract state names, and the opcodes and status bits serve as inputs. In the case of the status bits, only those bits that are used in the instruction are specified. By using combinations of bits and multiple status bit patterns, it is possible to specify functions of status bits. Note that many of the entries in Table 10-14 contain Xs, symbolizing "don't cares." For these entries, the input or resource is not used in the given microoperation or the specific bits of the code that are X are not used for controlling it. It is a useful exercise to determine how each of the entries in Table 10-14 is obtained, based on Table 10-12, Table 10-13, and Figure 10-20.

It is interesting to briefly compare the timing of the execution of instructions in this organization with that for the single-cycle computer. Each instruction requires two clock cycles to fetch and execute, compared with one clock cycle for the single-cycle computer. Because the very long delay path from the *PC* through the Instruction memory, Instruction decoder, datapath, and branch control is broken up by the instruction register, the clock periods are somewhat shorter. Nevertheless, due to setup time requirements for the added flip-flops in the *IR* and a potential imbalance in delays for the various paths through the circuit, the overall time taken to execute an instruction could be just as long as or longer than in the single-cycle computer. So what is the benefit of this organization other than ability to use a single memory? The next two instructions give the answer.

The first instruction to be added is a "load register indirect" (LRI), with opcode 0010001. In this instruction, the contents of register SA address a word in memory. The word, which is known as an *indirect address*, is then used to address the word in memory that is loaded into register DR. This can be represented symbolically as

$$R[DR] \leftarrow M[M[R[SA]]]$$

The ASM chart for the execution of this instruction is given in Figure 10-21. Following the instruction fetch, the state becomes EX0. In this state, R[SA] addresses the memory to obtain the indirect address, which is then placed in temporary register *R*8. In the next state, EX1, the next memory access occurs with the address from *R*8. The operand obtained is placed in R[DR] to complete the operation, and the *PC* is incremented. The ASM then returns to state INF to fetch the next instruction. The vector decision box for opcode is required for all states, since these same states are used by other instructions for their execution. Clearly, with two accesses to Memory *M*, this instruction could not be executed by the single-clock-cycle computer or using two clock cycles in the multiple-cycle computer. Also, to avoid disturbing the contents of registers *R*0 through *R*7 (except for R[SA]), the use of register *R8* for temporary storage is essential. The LRI instruction requires three clock cycles for its execution. To accomplish the same operation in the single-cycle computer requires two LD instructions, taking two clock cycles. In the multiple-cycle computer, due to two instruction fetches and two data accesses, it would require two LD instructions, but would take four clock cycles. So the LRI instruction gives an improvement in execution time in the latter case.

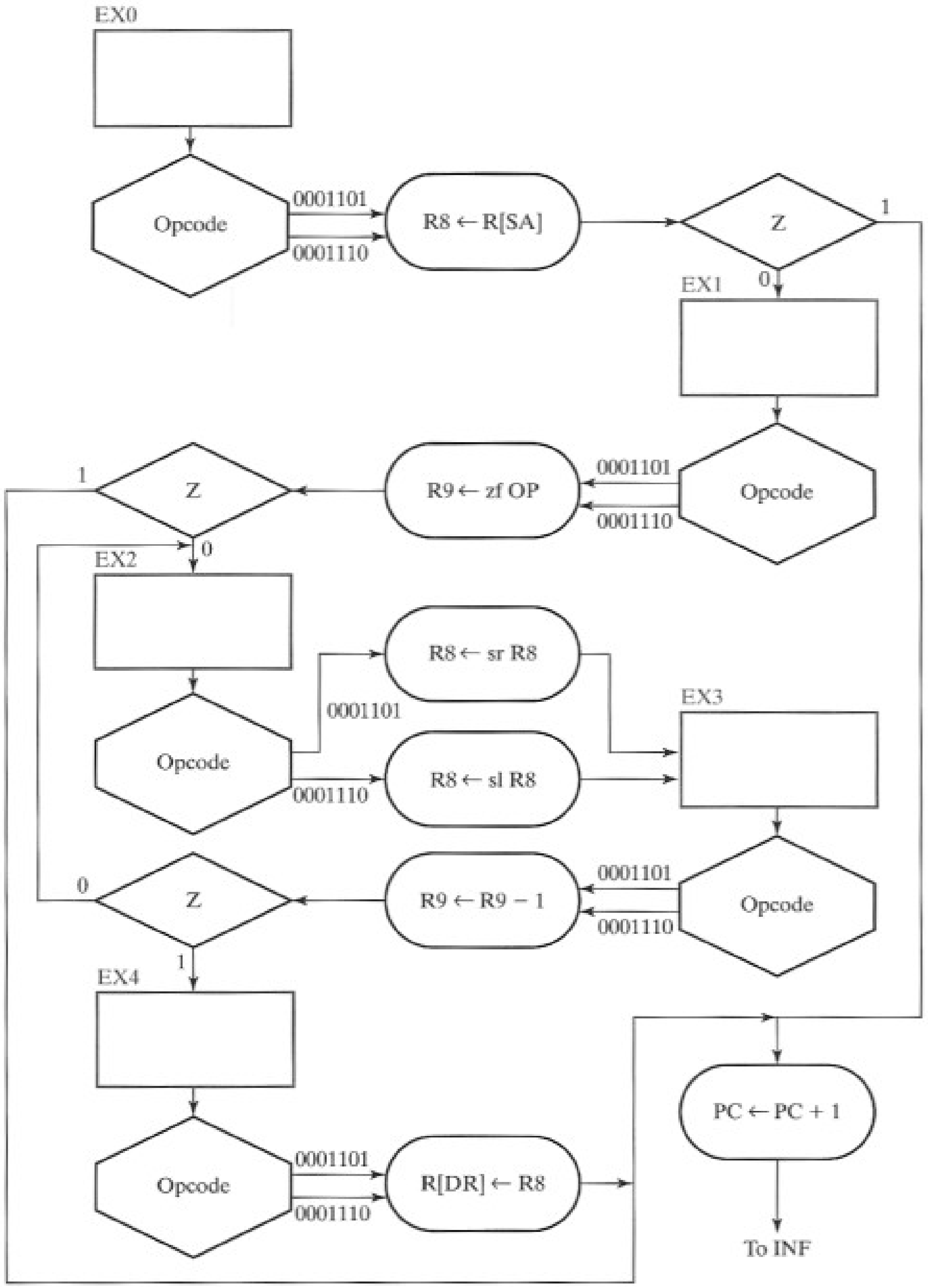


□ **FIGURE 10-21**
ASM Chart for Register Indirect Instruction

The final two instructions to be added are "shift right multiple" (SRM) and "shift left multiple" (SLM), with opcodes 0001101 and 0001110, respectively. These two instructions can share most of the microinstruction sequence to be used. SRM specifies that the contents of register SA are to be shifted to the right by the number of positions given by the three bits of the OP field, with the result placed in register DR. The ASM chart for this operation (and for SLM) is given in Figure 10-22. Register *R*9 stores the number of bit positions remaining to be shifted, and the shifting is performed in register *R*8.

Initially, the contents of R[SA] to be shifted is placed in *R*8. As it is loaded into *R*8, it is checked to see if it is 0 and shifting is not needed. Likewise, the shift amount being loaded into *R*9 is checked to see whether it is 0, meaning that shifting is not needed. If either case is satisfied, the instruction execution is complete, and the ASM flow returns to state INF. Otherwise, a right-shift operation is performed on the contents of register *R*8. *R*9 is decremented and tested to see whether it will be 0. If $R9 \neq 0$, then the shift and decrement are repeated. If $R9 = 0$, then the contents of *R*8 have been shifted by the number of bit positions specified by OP, so the result is transferred to R[DR] to complete the instruction execution, and the ASM flow returns to state INF.

If both the operand and the shift amount are nonzero, SRM, including fetch, requires $2s + 4$ clock cycles, where s is the number of positions shifted. The range of clock cycles required, including the instruction fetch, is from 6 to 18. If the same operation were implemented by a program using the right-shift instruction plus increment and branching, then $3s + 3$ instructions would be required giving $6s + 6$ cycles. The improvement in the required number of clock cycles is $4s + 2$, so 6 to 30 clock cycles are saved in the multiple-cycle computer for a nonzero operand and shift amount. Also, five fewer memory locations are required for storage of the SRM instruction, in contrast to that for the program.



□ **FIGURE 10-22**
ASM Chart for Right-Shift Multiple Instruction

□ **TABLE 10-15**
State Table for Illustration of Instructions Having Three or More Cycles

State	Inputs		Next state	Outputs												Comments
	Opcode	VCNZ		IL	PS	DX	AX	BX	MB	FS	MD	RW	MM	MW		
EX0	0010001	XXXX	EX1	0	00	1000	0XXX	XXXX	X	0000	1	1	X	0	LRI	*R*8←*M*[*R*[SA],→EX1
EX1	0010001	XXXX	INF	0	01	0XXX	1000	XXXX	X	0000	1	1	X	0	LRI	*R*[DR] ← *M*[*R*8],→INF*
EX0	0001101	XXX0	EX1	0	00	1000	0XXX	XXXX	X	0000	0	1	X	0	SRM	*R*8←*R*[SA],$\overline{Z}$:→EX1
EX0	0001101	XXX1	INF	0	01	1000	0XXX	XXXX	X	0000	0	1	X	0	SRM	*R*8←*R*[SA],Z:→INF*
EX1	0001101	XXX0	EX2	0	00	1001	XXXX	XXXX	1	1100	0	1	X	0	SRM	*R*9← zf OP,$\overline{Z}$:→EX2
EX1	0001101	XXX1	INF	0	01	1001	XXXX	XXXX	1	1100	0	1	X	0	SRM	*R*9← zf OP,Z:→INF*
EX2	0001101	XXXX	EX3	0	00	1000	XXXX	1000	0	1101	0	1	X	0	SRM	*R*8 ← sr *R*8,→EX3
EX3	0001101	XXX0	EX2	0	00	1001	1001	XXXX	X	0110	0	1	X	0	SRM	*R*9 ← *R*9 − 1,$\overline{Z}$:→EX2
EX3	0001101	XXX1	EX4	0	00	1001	1001	XXXX	X	0110	0	1	X	0	SRM	*R*9 ← *R*9 − 1,Z:→EX4
EX4	0001101	XXXX	INF	0	01	0XXX	1000	XXXX	X	0000	0	1	X	0	SRM	*R*[DR] ← *R*8,→ INF*
EX0	0001110	XXX0	EX1	0	00	1000	0XXX	XXXX	X	0000	0	1	X	0	SLM	*R*8←*R*[SA],$\overline{Z}$:→EX1
EX0	0001110	XXX1	INF	0	00	1000	0XXX	XXXX	X	0000	0	1	X	0	SLM	*R*8←*R*[SA],Z:→INF*
EX1	0001110	XXX0	EX2	0	01	1001	XXXX	XXXX	1	1100	0	1	X	0	SLM	*R*9← zf OP,$\overline{Z}$:→EX2
EX1	0001110	XXX1	INF	0	01	1001	XXXX	XXXX	1	1100	0	1	X	0	SLM	*R*9← zf OP,Z:→INF*
EX2	0001110	XXXX	EX3	0	00	1000	XXXX	1000	0	1110	0	1	X	0	SLM	*R*8 ← sl *R*8,→EX3
EX3	0001110	XXX0	EX2	0	00	1001	1001	XXXX	X	0110	0	1	X	0	SLM	*R*9 ← *R*9 − 1,$\overline{Z}$:→EX2
EX3	0001110	XXX1	EX4	0	00	1001	1001	XXXX	X	0110	0	1	X	0	SLM	*R*9 ← *R*9 − 1,Z:→EX4
EX4	0001110	XXXX	INF	0	01	0XXX	1000	XXXX	X	0000	0	1	X	0	SLM	*R*[DR] ←*R*8,→ IF*

*For this state and input combination, PC ← PC + 1 also occurs.

In the ASM chart in Figure 10-22, the states INF and EX0 (and EX1) are the same as those used for the two-cycle instructions in the ASM chart in Figure 10-20 and for the LRI instruction in Figure 10-21. Also, implementation of the left shift multiple operation is shown in Figure 10-22 in which, based on the opcode, the left shift of *R*8 replaces the right shift of *R*8. As a consequence, the logic implementing the states used for implementation of these two instructions can be shared. Further, the logic used for the sequencing of the states can be shared between the SRM and SLM instruction implementations.

The state table specification in Table 10-15 is derived by using the information from the ASM chart in Figure 10-22, and Tables 10-12 and 10-13. The codes are derived from the register transfer and sequencing action described in the comments on the right in the same way that Table 10-15 was derived.

Implementation of the LRI and SRM instructions illustrates the flexibility achieved using multiple-cycle control. Implementation of additional instructions is explored in the problems at the end of the chapter.

10-10 Chapter Summary

In the first part of the chapter, the concept of datapaths for information processing in digital systems was introduced. Among the major components of datapaths are register files, buses, arithmetic/logic units (ALUs), and shifters. The control word provides a means of organizing the control of the microoperations performed by the datapath. These concepts were combined into the concept of a datapath, which serves as a basis for exploring computers in the remainder of the text.

In the second part of the chapter, control design for programmed systems was introduced by examining two different implementations of basic control units for a simple computer architecture. We introduced the concept of instruction set architectures and defined instruction formats and operations for the simple computer. The first implementation of this computer is capable of executing any instruction in a single clock cycle. Aside from having a program counter and its logic, the control unit of this computer consists of a combinational decoder circuit.

Among the shortcomings of the single-cycle computer are limitations on the complexity of the instructions that can be executed on it, problems with the interface to a single memory, and the relatively low clock frequencies attained. To deal with the first two of these shortcomings, we examined a multiple-cycle version of the simple computer in which a single memory is used and instructions are implemented using two distinct phases: instruction fetch and instruction execution. The remaining issue of long clock cycles is dealt with in Chapter 12 by introducing pipelined datapaths and control.

References

1. Mano, M. M. *Computer Engineering: Hardware Design:* Englewood Cliffs, NJ: Prentice Hall, 1988.
2. Mano, M. M. *Computer System Architecture,* 3rd Ed. Englewood Cliffs, NY: Prentice Hall, 1993.

3. Patterson, D. A., and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface,* 2nd ed. San Francisco, CA: Morgan Kaufmann, 1998.

4. Hennessy, J. L., and D. A. Patterson. *Computer Architecture: A Quantitative Approach,* 2nd ed. San Francisco, CA: Morgan Kaufmann, 1996.

Problems

The plus (+) indicates a more advanced problem and the asterisk (*) indicates a solution is available on the Companion Website for the text.

10–1. A datapath similar to the one in Figure 10-1 has 128 registers. How many selection lines are needed for each set of multiplexers and for the decoder?

10–2. *Given an 8-bit ALU with outputs F_7 through F_0 and available carries C_8 and C_7, show the logic circuit for generating the signals for the four status bits N (sign), Z (zero), V (overflow), and C (carry).

10–3. *Design an arithmetic circuit with two selection variables S_1 and S_0 and two n-bit data inputs A and B. The circuit generates the following eight arithmetic operations in conjunction with carry C_{in}:

S_1	S_0	$C_{in} = 0$	$C_{in} = 1$
0	0	$F = A + B$ (add)	$F = A + \overline{B} + 1$ (subtract A – B)
0	1	$F = \overline{A} + B$	$F = \overline{A} + B + 1$ (subtract B – A)
1	0	$F = A - 1$ (decrement)	$F = A + 1$ (increment)
1	1	$F = \overline{A}$ (1's Complement)	$F = \overline{A} + 1$ (2's Complement)

Draw the logic diagram for the two least significant bits of the arithmetic circuit.

10–4. *Design a 4-bit arithmetic circuit, with two selection variables S_1 and S_0, that generates the following arithmetic operations:

$S_1 S_0$	$C_{in} = 0$	$C_{in} = 1$
0 0	$F = A + B$ (add)	$F = A + B + 1$
0 1	$F = A$ (transfer)	$F = A + 1$ (increment)
1 0	$F = \overline{B}$ (complement)	$F = \overline{B} + 1$ (negate)
1 1	$F = A + \overline{B}$	$F = A + \overline{B} + 1$ (subtract)

Draw the logic diagram for a single bit stage.

10–5. Inputs X_i and Y_i of each full adder in an arithmetic circuit have digital logic specified by the Boolean functions

$$X_i = A_i \qquad Y_i = \overline{B}_i S + B_i \overline{C}_{\text{in}}$$

where S is a selection variable, C_{in} is the input carry, and A_i and B_i are input data for stage i.

(a) Draw the logic diagram for the 4-bit circuit, using full adders and multiplexers.

(b) Determine the arithmetic operation performed for each of the four combinations of S and C_{in}: 00, 01, 10, and 11.

10–6. *Design one bit of a digital circuit that performs the four logic operations of exclusive-OR, exclusive-NOR, NOR, and NAND on register operands A and B with the result to be loaded into register A. Use two selection variables.

(a) Using a Karnaugh map, design minimum logic for one typical stage, and show the logic diagram.

(b) Repeat (a), trying different assignments of the selection codes to the four operations to see whether the logic for the stage can be simplified further.

10–7. +Design an ALU that performs the following operations:

$A + B$	sl A
$A + \overline{B} + 1$	$A \vee B$
$\overline{B}$	$A \oplus B$
$\overline{B} + 1$	$A \wedge B$

Give the result of your design as the logic diagram for a single stage of the ALU. Your design should have only a single carry line between stages and three selection bits. If you have access to logic simplification software, apply it to the design to obtain reduced logic.

10–8. *Find the output Y of the 4-bit barrel shifter in Figure 10-9 for each of the following bit patterns applied to S_1, S_0, D_3, D_2, D_1, and D_0:

(a) 000101 **(b)** 010011

(c) 101010 **(d)** 111100

10–9. Specify the 16-bit control word that must be applied to the datapath of Figure 10-11 to implement each of the following microoperations:

(a) $R0 \leftarrow R1 + R7$ **(b)** $R7 \leftarrow 0$

(c) $R6 \leftarrow$ sl $R6$ **(d)** $R3 \leftarrow$ sr$R4$

(e) $R1 \leftarrow R7 + 1$ **(f)** $R2 \leftarrow R4 -$ Constant in

(g) $R1 \leftarrow R2 \oplus R3$ **(h)** $R5 \leftarrow$ Data in

10–10. *Given the following 16-bit control words for the datapath of Figure 10-11, determine (a) the microoperation that is executed and (b) the change in the contents of the register for each control word (assume that the registers are 8-bit registers and that, before the execution of a control word, they contain the value of their number (e.g., register *R*5 contains 05 in hexadecimal)). Assume that Constant has value 6 and Data in has value 1B, both in hexadecimal.

(a)101 100 101 0 1000 0 1

(b)110 010 100 0 0101 0 1

(c)101 110 000 0 1100 0 1

(d)101 000 000 0 0000 0 1

(e)100 100 000 1 1101 0 1

(f)011 000 000 0 0000 1 1

10–11. Given the sequence of 16-bit control words below for the datapath in Figure 10-11 and the initial ASCII character codes in 8-bit registers, simulate the datapath to determine the alphanumeric characters in the registers after the execution of the sequence. The result is a scrambled word: what is it?

011 011 001 0 0010 0 1	*R*0	00000000
100 100 001 0 1001 0 1	*R*1	00100000
101 101 001 0 1010 0 1	*R*2	01000100
001 001 000 0 1011 0 1	*R*3	01000111
001 001 000 0 0001 0 1	*R*4	01010100
110 110 001 0 0101 0 1	*R*5	01001100
111 111 001 0 0101 0 1	*R*6	01000001
001 111 000 0 0000 0 1	*R*7	01001001

10–12. A datapath has five major components, *A* through *E*, attached in a loop from register file to register file similar to that in Figure 10-17. The maximum delay of each of the components is *A*, 2 ns; *B*, 1 ns; *C*, 3 ns; *D*, 4 ns; and *E*, 4 ns.

(a) What is the maximum clock frequency that can be used for the datapath?

(b) The datapath is to be changed to one that is pipelined using three stages. How should the components be combined into stages, and what is the maximum clock frequency that can be achieved?

(c) Repeat (b) for four pipeline stages.

10–13. A computer has a 32-bit instruction word broken into fields as follows: opcode, 6 bits; two register fields, 6 bits each; and one immediate operand/register field, 14 bits.

(a) What is the maximum number of operations that can be specified?

(b) How many registers can be addressed?

(c) What is the range of unsigned immediate operands that can be provided?

(d) What is the range of signed immediate operands that can be provided, assuming that bit 13 is the sign bit?

10–14. *A digital computer has a memory unit with a 32-bit instruction and a register file with 32 registers. The instruction set consists of 110 different operations. There is only one type of instruction format, with an opcode part, a register file address, and an immediate operand part. Each instruction is stored in one word of memory.

(a) How many bits are needed for the opcode part of the instruction?

(b) How many bits are left for the immediate part of the instruction?

(c) If the immediate operand is used as an unsigned address to memory, what is the maximum number of words that can be addressed in memory?

(d) What are the largest and the smallest algebraic values of signed 2's complement binary numbers that can be accommodated as an immediate operand?

10–15. A digital computer has 32-bit instructions. There are a number of different instruction formats and the number of bits in each format used for opcodes varies depending on the bits needed for other fields. If the first bit of the opcode is 0, then there are 4 opcode bits. If the first bit of the opcode is 1 and the second bit of the opcode is 0, then there are 6 opcode bits. If the first bit of the opcode is 1 and the second bit of the opcode is 1, then there are 8 opcode bits. How many distinct opcodes are available for this computer?

10–16. The single-cycle computer in Figure 10-15 executes the five instructions described by the register transfers in the table that follows.

(a) Complete the following table, giving the binary instruction decoder outputs from Figure 10-16 during execution of each of the instructions:

Instruction—Register Transfer	DA	AA	BA	MB	FS	MD	RW	MW	PL	JB
$R[0] = R[7] \oplus R[3]$										
$R[1] \leftarrow M[R[4]]$										
$R[2] \leftarrow R[5] + 2$										
$R[3] \leftarrow \mathrm{sl}\ R[6]$										

if (R([4] = 0)
$PC \leftarrow PC + \mathrm{se}\ PC$
else $PC \leftarrow PC + 1$										

(b) Complete the following table, giving the instruction in binary for the single-cycle computer that executes the register transfer (if any field is not used, give it the value 0):

Instruction—Register Transfer	Opcode	DR	SA	SB or Operand
$R[0] = \mathrm{sr}R[7]$				
$R[1] \leftarrow M[R[6]]$				
$R[2] \leftarrow R[5] + 4$				
$R[3] \leftarrow R[4] \oplus R[3]$				
$R[4] \leftarrow R[2] - R[1]$				

10–17. Using the information in the truth table in Table 10-10, verify that the design for the single-bit outputs in the decoder in Figure 10-16 is correct.

10–18. Manually simulate the single-cycle computer in Figure 10-15 for the following sequence of instructions, assuming that each register initially contains contents equal to its index (i.e., $R0$ contains 0, $R1$ contains 1, etc.):

SUB R0, R1, R2
SUB R3, R4, R5
SUB R6, R7, R0
SUB R0, R0, R3
SUB R0, R0, R6
ST R7, R0
LD R7, R6
ADI R0, R6, 0
ADI R3, R6, 3

Give (a) the binary value of the instruction on the current line of the results and (b) the contents of any register changed by the instruction, or the location and contents of any memory location changed by the instruction on the next line of the results. The results are positioned in this fashion because the new values do not appear in a register or memory, due to the execution of an instruction, until after a positive clock edge has occurred.

10–19. Give an instruction for the single-cycle computer that resets register $R4$ to 0 and updates the Z and N status bits based on the value 0 transferred to $R4$. (*Hint*: Try the exclusive-OR.) By examining the detailed ALU logic, determine the values of the V and C status bits.

10–20. List the control logic state table entries for the multiple-cycle computer (see Table 10-15) that implement the following register transfer statements. Assume that in all cases the present state is EX0 and the opcode is 0010001.

(a) $R3 \leftarrow R1 - R2, \ \rightarrow EX1$ Assume DR = 3, SA = 1, SB = 2.

(b) $R8 \leftarrow \mathrm{sr}\ R8, \ \rightarrow INF$ Assume DR = 5, SB = 5

(c) if $(N = 0)$ then $(PC \rightarrow PC + \text{se}, \ \rightarrow INF)$ else $(PC \rightarrow PC + 1, \ \rightarrow INF)$

(d) $R6 \leftarrow R6, C \leftarrow 0, \ \rightarrow INF$ Assume DR = SA = 6.

10–21. Manually simulate the SRM instruction in the multiple-cycle computer for operand 0001001101111000 for OP = 6.

10–22. A new instruction is to be defined for the multiple-cycle computer with opcode 0010001. The instruction implements the register transfer

$$R[DR] \leftarrow R[SB] + M[R[SA]]$$

Find the ASM chart for implementing the instruction, assuming that 0010001 is the opcode. Form the part of the control state table that implements this instruction.

10–23. Repeat Problem 10-22 for the two instructions: Add and check OV (AOV), described by the register transfer

$$R[DR] \leftarrow R[SA] + R[SB], V{:}R8 \leftarrow 1, \overline{V}{:}R8 \leftarrow 0$$

and BRanch on oVerflow (BRV), described by the register transfer

$$R8 \leftarrow R8, V{:}PC \leftarrow PC + \text{se AD}, \overline{V}{:}PC \leftarrow PC + 1$$

The opcode for AOV is 1000101 and, for BRV, is 1000110. Note that register R8 is used as a "status" register that stores the overflow result V for the previous operation. All of the values N, Z, C and V could be stored in R8 to give a complete status on the prior arithmetic or logic operation.

10–24. +A new instruction is to be defined for the multiple-cycle computer. The instruction compares two unsigned integers stored in register R[SA] and R[SB]. If the integers are equal, then bit 0 of R[DR] is set to 1. If R[SA] is greater than R[SB], then bit 1 of R[DR] is set to 1. Otherwise, bits 0 and 1 are both 0. All other bits of R[DR] have value 0. Find the ASM chart for implementing the instruction, assuming that 0010001 is the opcode. Form the part of the control state table that implements this instruction.

CHAPTER

11

Instruction Set Architecture

Up to this point, much of what we have studied has focused on digital system design, with computer components used as examples. In this chapter, the material studied becomes decidedly more specialized, dealing with instruction set architecture for general-purpose computers. We will examine the operations that the instructions perform and focus particularly on how the operands are obtained and where the results are stored. In our studies, we will contrast two distinct classes of architectures: reduced instruction set computers (RISCs) and complex instruction set computers (CISCs). We will classify elementary instructions into three categories: data transfer, data manipulation, and program control. In each of these categories, we will elaborate on typical elementary instructions.

In light of this change in focus, the general-purpose parts of the generic computer at the beginning of Chapter 1, including the central processing unit (CPU) and the accompanying floating-point unit (FPU), are heavily shaded. In addition, since a small general-purpose microprocessor may be present for controlling keyboard and monitor functions, we have lightly shaded these components. Aside from addressing used to access memory and I/O components, the concepts studied apply less to other areas of the computer. Increasingly, however, small CPUs are appearing more and more in the I/O components, giving a changing picture of the role of general-purpose instruction set architectures in the generic computer.

11-1 Computer Architecture Concepts

The binary language in which instructions are defined and stored in memory is referred to as *machine language*. A symbolic language that replaces binary opcodes and addresses with symbolic names and that provides other features helpful to the programmer is referred to as *assembly language*. The logical structure of computers is normally described in assembly language reference manuals. Such manuals explain various internal elements of the computer that are of interest to the programmer, such as processor registers. The manuals list all hardware-implemented instructions, specify the symbolic names and binary code format of the instructions, and provide a precise definition of each instruction. In the past, this information represented the *architecture* of the computer. A computer was composed of its architecture, plus a specific *implementation* of that architecture. The implementation was separated into two parts: the organization and the hardware. The *organization* consists of structures such as datapaths, control units, memories, and the buses that interconnect them. *Hardware* refers to the logic, the electronic technologies employed, and the various physical design aspects of the computer. As computer designers pushed for higher and higher performance, and as increasingly more of the computer resided within a single IC, the relationships among architecture, organization, and hardware became so intertwined that a more integrated viewpoint became necessary. According to this new viewpoint, architecture as previously defined is more restrictively called *instruction set architecture* (*ISA*), and the term *architecture* is used to encompass the whole of the computer, including instruction set architecture, organization, and hardware. This unified view enables intelligent design trade-offs to be made that are apparent only in a tightly coupled design process. These trade-offs have the potential for producing better computer designs. In this chapter, we focus on instruction set architecture. In the next, we will look at two distinct instruction set architectures, with a focus on implementation using two very different organizations.

A computer usually has a variety of instructions and multiple instruction formats. It is the function of the control unit to decode each instruction and provide the control signals needed to process it. Simple examples of instructions and instruction formats were presented in Section 10-7. We will now expand this presentation by introducing typical instructions found in commercial general-purpose computers. We will also investigate the various instruction formats that may be encountered in a typical computer, with an emphasis on the addressing of operands. The format of an instruction is depicted in a rectangular box symbolizing the bits of the binary instruction. The bits are divided into groups called *fields*. The following are typical fields found in instruction formats:

1. An *opcode field*, which specifies the operation to be performed.
2. An *address field*, which provides either a memory address or an address for selecting a processor register.
3. A *mode field*, which specifies the way the address field is to be interpreted.

Other special fields are sometimes employed under certain circumstances—for example, a field that gives the number of positions to shift in a shift-type instruction or an operand field in an immediate operand instruction.

Basic Computer Operation Cycle

In order to comprehend the various addressing concepts to be presented in the next two sections, we need to understand the basic operation cycle of the computer. The control unit of a computer is designed to execute each instruction of a program in the following sequence of steps:

1. Fetch the instruction from memory into a control register.
2. Decode the instruction.
3. Locate the operands used by the instruction.
4. Fetch operands from memory (if necessary).
5. Execute the operation in processor registers.
6. Store the results in the proper place.
7. Go back to step 1 to fetch the next instruction.

As explained in Section 10-7, there is a register in the computer called the program counter (*PC*) that keeps track of the instructions in the program stored in memory. The *PC* holds the address of the instruction to be executed next and is incremented by one each time a word is read from the program in memory. The decoding done in Step 2 determines the operation to be performed and the addressing mode of the instruction. The operands in Step 3 are located from the addressing mode and the address field of the instruction. The computer executes the instruction, storing the result, and returns to Step 1 to fetch the next instruction in sequence.

Register Set

The *register set* consists of all registers in the CPU that are accessible to the programmer. These registers are typically those mentioned in assembly language programming reference manuals. In the simple CPUs we have dealt with so far, the register set has consisted of the programmer-accessible portion of the register file and the *PC*. The CPUs can also contain other registers, such as the instruction register, registers in the register file that are accessible only to microprograms, and pipeline registers. These registers, however, are not directly accessible to the programmer and, as a consequence, are not a part of the register set, which represents the stored information in the CPU that instructions can access. Thus, the register set has a considerable influence on instruction set architecture.

The register set for a realistic CPU can become quite complex. For the discussion in this chapter, we add two registers to the set we have used thus far: the *processor status register* (*PSR*) and the *stack pointer* (*SP*). The processor status register contains flip-flops that are selectively set by status values *C*, *N*, *V*, and *Z* from the ALU. These stored status bits are used to make decisions that determine the program flow, based on ALU results or the contents of registers. The stored status bits in the processor status register are also referred to as the *condition codes* or the *flags*. Additional bits in the *PSR* will be discussed when we cover associated concepts in this chapter.

11-2 Operand Addressing

Consider an instruction such as ADD, which specifies the addition of two operands to produce a result. Suppose that the result of the addition is treated as just another operand. Then the ADD instruction has three operands: the addend, the augend, and the result. An operand residing in memory is specified by its address. An operand residing in a processor register is specified by a register address, a binary code of n bits that specifies one of 2^n registers in the register file. Thus, a computer with 16 processor registers, say, $R0$ through $R15$, has in its instructions one or more register address fields of four bits. The binary code 0101, for example, designates register $R5$.

Some operands, however, are not explicitly addressed, because their location is specified either by the opcode of the instruction or by an address assigned to one of the other operands. In such a case, we say that the operand has an *implied address*. If the address is implied, then there is no need for a memory or register address field for the operand in the instruction. On the other hand, if an operand has an address in the instruction, then we say that the operand is explicitly addressed or has an *explicit address*.

The number of operands explicitly addressed for a data manipulation operation such as ADD is an important factor in defining the instruction set architecture for a computer. An additional factor is the number of such operands that can be explicitly addressed in memory by the instruction. These two factors are so important in defining the nature of instructions that they act a means of distinguishing different instruction set architectures. They also govern the length of computer instructions.

We begin by illustrating simple programs with different numbers of explicitly addressed operands per instruction. Since each explicitly addressed operand has up to three memory or register addresses per instruction, we label the instructions as having three, two, one, or zero addresses. Note that, of the three operands needed for an instruction such as ADD, the addresses of all operands not having an address in the instruction are implied.

To illustrate the influence of the number of operands on computer programs, we will evaluate the arithmetic statement

$$X = (A + B)(C + D)$$

using three, two, one, and zero address instructions. We will assume that the operands are in memory addresses symbolized by the letters A, B, C, and D and must not be changed by the program. The result is to be stored in memory at a location with address X. The arithmetic operations to be used in the instructions are addition, subtraction, and multiplication, denoted by ADD, SUB, and MUL, respectively. Further, three operations needed to transfer data during the evaluation are move, load, and store, denoted by MOVE, LD, and ST, respectively. LD moves an operand from memory to a register and ST from a register to memory. Depending on the addresses permitted, MOVE can transfer data between registers, between memory locations, or from memory to register or register to memory.

Three-Address Instructions

A program that evaluates $X = (A + B)(C + D)$ using three-address instructions is as follows (a register transfer statement is shown for each instruction):

ADD T1, A, B	$M[T1] \leftarrow M[A] + M[B]$
ADD T2, C, D	$M[T2] \leftarrow M[C] + M[D]$
MUL X, T1, T2	$M[X] \leftarrow M[T1] \times M[T2]$

The symbol $M[A]$ denotes the operand stored in memory at the address symbolized by A. The symbol × designates multiplication. T1 and T2 are temporary storage locations in memory.

This same program can use registers as the temporary storage locations:

ADD R1, A, B	$R1 \leftarrow M[A] + M[B]$
ADD R2, C, D	$R2 \leftarrow M[C] + M[D]$
MUL X, R1, R2	$M[X] \leftarrow R1 \times R2$

Use of registers reduces the memory accesses required from nine to five. An advantage of the three-address format is that it results in short programs for evaluating expressions. A disadvantage is that the binary coded instructions require more bits to specify three addresses, particularly if they are memory addresses.

Two-Address Instructions

For two-address instructions, each address field can again specify either a possible register or a memory address. The first operand address listed in the symbolic instruction also serves as the implied address to which the result of the operation is transferred. The program is as follows:

MOVE T1, A	$M[T1] \leftarrow M[A]$
ADD T1, B	$M[T1] \leftarrow M[T1] + M[B]$
MOVE X, C	$M[X] \leftarrow M[C]$
ADD X, D	$M[X] \leftarrow M[X] + M[D]$
MUL X, T1	$M[X] \leftarrow M[X] \times M[T1]$

If a temporary storage register R1 is available, it can replace T1. Note that this program takes five instructions instead of the three used by the three-address instruction program.

One-Address Instructions

To perform instructions such as ADD, a computer with one-address instructions uses an implied address—such as a register called an *accumulator ACC*—for obtaining one of the operands and as the location of the result. The program to evaluate the arithmetic statement is as follows:

LD	A	$ACC \leftarrow M[A]$
ADD	B	$ACC \leftarrow ACC + M[B]$
ST	X	$M[X] \leftarrow ACC$
LD	C	$ACC \leftarrow M[C]$
ADD	D	$ACC \leftarrow ACC + M[D]$
MUL	X	$ACC \leftarrow ACC \times M[X]$
ST	X	$M[X] \leftarrow ACC$

All operations are done between the *ACC* register and a memory operand. In this case, the number of instructions in the program has increased to seven and the memory accesses is also seven.

Zero-Address Instructions

To perform an ADD instruction with zero addresses, all three addresses in the instruction must be implied. A conventional way of achieving this goal is to use a structure referred to as a *stack*, which is a mechanism or structure that stores information such that the item stored last is the first retrieved. Because of its "last in, first out" nature, a stack is also called a *last in, first out* (*LIFO*) queue. The operation of a computer stack is analogous to that of a stack of trays or plates in which the last tray placed on top of the stack is the first to be taken off. Data manipulation operations such as ADD are performed on the stack. The word at the top of the stack is referred to as TOS. The word below it in the stack is referred to as TOS$_{-1}$. When one or more words are used as operands for an operation, they are removed from the stack. The word below them then becomes the new TOS. When a resulting word is produced, it is placed on the stack and becomes the new TOS. Thus, TOS and a few locations below it are the implied addresses for operands, and TOS is the implied address for the result. For example, the instruction that specifies an addition is simply

ADD

The resulting register transfer action is $TOS \leftarrow TOS + TOS_{-1}$. Thus, there are no registers or register addresses used for data manipulation instructions in a stack architecture. Memory addressing, however, is used in such architectures for data transfers. For instance, the instruction

PUSH X

results in $TOS \leftarrow M[X]$, a transfer of the word in address X in memory to the top of the stack. A corresponding operation,

POP X

results in $M[X] \leftarrow TOS$, a transfer of the entry at the top of the stack to address X in memory.

The program for evaluating the sample arithmetic statement for the zero-address situation is as follows:

```
PUSH  A   $TOS \leftarrow M[A]$
PUSH  B   $TOS \leftarrow M[B]$
ADD       $TOS \leftarrow TOS + TOS_{-1}$
PUSH  C   $TOS \leftarrow M[C]$
PUSH  D   $TOS \leftarrow M[D]$
ADD       $TOS \leftarrow TOS + TOS_{-1}$
MUL       $TOS \leftarrow TOS \times TOS_{-1}$
POP   X   $M[X] \leftarrow TOS$
```

This program requires eight instructions—one more than the number required by the previous one-address program. However, it uses addressed memory locations or registers only for PUSH and POP and not to execute data manipulation instructions involving ADD and MUL.

Addressing Architectures

The programs just presented change if the number of addresses to the memory in the instructions is restricted or if the memory addresses are restricted to specific instructions. These restrictions, combined with the number of operands addressed, define addressing architectures. We can illustrate such architectures with the evaluation of an arithmetic statement in a three-address architecture that has all of the accesses to memory. Such an addressing scheme is called a *memory-to-memory architecture*. This architecture has only control registers, such as the program counter in the CPU. All operands come directly from memory, and all results are sent directly to memory. The formats of data transfer and manipulation instructions contain from one to three address fields, all of which are used for memory addresses. For the previous example, three instructions are required, but if an extra word must appear in the instruction for each memory address, then up to four memory reads are required to fetch each instruction. Including the fetching of operands and storing of results, the program to perform the addition would require 21 accesses to memory. If memory accesses take more than one clock cycle, the execution time would be in excess of 21 clock periods. Thus, even though the instruction count is low, the execution time is potentially high. Also, providing the capability for all operations to access memory increases the complexity of the control structures and may lengthen the clock cycle. Thus, this memory-to-memory architecture is typically not used in new designs.

In contrast, the three-address *register-to-register* or *load/store architecture*, which allows only one memory address and restricts its use to load and store types of instructions, is typical in modern processors. Such an architecture requires a sizeable register file, since all data manipulation instructions use register operands. With this architecture, the program to evaluate the sample arithmetic statement is as follows:

```
LD     R1, A         R1←M[A]
LD     R2, B         R2←M[B]
ADD    R3, R1, R2    R3←R1 + R2
LD     R1, C         R1←M[C]
LD     R2, D         R2←M[D]
ADD    R1, R1, R2    R1←R1 + R2
MUL    R1, R1, R3    R1←R1 × R3
ST     X, R1         M[X]←R1
```

Note that the instruction count increases to eight compared to three for the three-address, memory-to-memory case. Note also that the operations are the same as those for the stack case, except for the need for register addresses. By using registers, the number of accesses to memory for instructions, addresses, and operands is reduced from 21 to 18. If addresses can be obtained from registers instead of memory, as discussed in the next section, this number can be further reduced.

Variations on the previous two addressing architectures include three-address instructions and two-address instructions with one or two of the addresses to memory. The program lengths and number of memory accesses tend to be intermediate between the previous two architectures. An example of a two-address instruction with a single memory address allowed is

$$\text{ADD} \qquad \text{R1, A} \qquad R1 \leftarrow R1 + M[A]$$

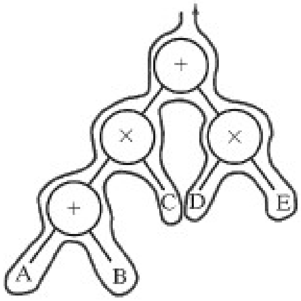
This type of architecture is a *register-memory* architecture and remains prevalent among the current instruction set architectures, primarily to provide compatibility with older software using a specific architecture.

The program with one-address instructions illustrated previously gives the *single-accumulator architecture*. Since this architecture has no register file, its single address is for accessing memory. It requires 21 accesses to memory to evaluate the sample arithmetic statement. In more complex programs, significant additional memory accesses would be needed for temporary storage locations in memory. Because of its large number of memory accesses, this architecture is inefficient and, as a consequence, is restricted to use in CPUs for simple, low-cost applications that do not require high performance.

The zero-address instruction case using a stack supports the concept of a *stack architecture*. Data manipulation instructions such as ADD use no address, since they are performed on the top few elements of the stack. Single memory-address load and store operations, as shown in the program to evaluate the sample arithmetic statement, are used for data transfer. Since most of the stack is located in memory, one or more hidden memory accesses may be required for each stack operation. As register-register and load/store architectures have made strong performance advances, the high frequency of memory accesses in stack architectures has made them unattractive. However, recent stack architectures have begun to borrow technological advances from these other architectures. These new architectures store substantial numbers of stack locations in the processor chip and handle transfers between these locations and the memory transparently. Stack architectures are particularly useful for rapid interpretation of high-level language programs in which the intermediate code representation uses stack operations.

Stack architectures are compatible with a very efficient approach to expression processing which uses postfix notation rather than the traditional infix notation to which we are accustomed. The infix expression

$$(\mathrm{A} + \mathrm{B}) \times \mathrm{C} + (\mathrm{D} \times \mathrm{E})$$



□ **FIGURE 11-1**
Graph for Example of Conversion from Infix to RPN

with the operators between the operands can be written as postfix expression

$$A\ B + C \times D\ E \times +$$

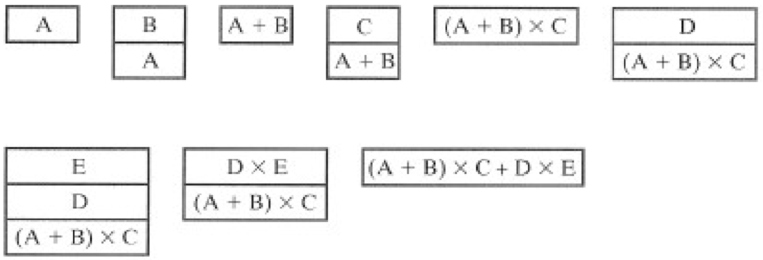
Postfix notation is called reverse Polish notation (RPN), named for Polish mathematician Jan Lukasiewicz, who proposed prefix (the reverse of postfix) notation.

Conversion of (A + B) × C + (D × E) to RPN can be achieved graphically as shown in Figure 11-1. When the path shown traversing the graph passes a variable, that variable is entered into the RPN expression. When the path passes an operation for the final time, the operation is entered into the RPN expression.

It is very easy to develop a program for an RPN expression. Whenever a variable is encountered, it is pushed onto the stack. Whenever an operation is encountered, it is executed on the implicit address TOS, or addresses TOS and TOS_{-1}, with the result placed in the new TOS. The program for the example RPN expression is

```
PUSH A
PUSH B
ADD
PUSH C
MUL
PUSH D
PUSH E
MUL
ADD
```

The execution of the program is illustrated by the successive stack states shown in Figure 11-2. As an operand is pushed on the stack, the stack contents are pushed down one stack location. When an operation is performed, the operand in the TOS is popped off and temporarily stored in a register. The operation is applied to the stored operand and the new TOS operand, and the result replaces the TOS operand.



□ **FIGURE 11-2**
Stack Activity for Execution of Example Stack Program

11-3 Addressing Modes

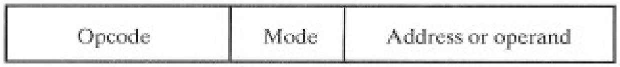
The operation field of an instruction specifies the operation to be performed. This operation must be executed on data stored in computer registers or memory words. How the operands are selected during program execution is dependent on the addressing mode of the instruction. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced. The address of the operand produced by the application of such a rule is called the *effective address*. Computers use addressing-mode techniques to accommodate one or both of the following provisions:

1. To give programming flexibility to the user via pointers to memory, counters for loop control, indexing of data, and relocation of programs.
2. To reduce the number of bits in the address fields of the instruction.

The availability of various addressing modes gives the experienced programmer the ability to write programs that require fewer instructions. The effect, however, on throughput and execution time must be carefully weighed. For example, the presence of more complex addressing modes may actually result in lower throughput and longer execution time. Also, most machine-executable programs are produced by compilers that often do not use complex addressing modes effectively.

In some computers, the addressing mode of the instruction is specified by a distinct binary code. Other computers use a common binary code that designates both the operation and the addressing mode of the instruction. Instructions may be defined with a variety of addressing modes, and sometimes two or more addressing modes are combined in one instruction.

An example of an instruction format with a distinct addressing-mode field is shown in Figure 11-3. The opcode specifies the operation to be performed. The



□ **FIGURE 11-3**
Instruction Format with Mode Field

mode field is used to locate the operands needed for the operation. There may or may not be an address field in the instruction. If there is an address field, it may designate a memory address or a processor register. Moreover, as discussed in the previous section, the instruction may have more than one address field. In that case, each address field is associated with its own particular addressing mode.

Implied Mode

Although most addressing modes modify the address field of the instruction, there is one mode that needs no address field at all: the implied mode. In this mode, the operand is specified implicitly in the definition of the opcode. It is the implied mode that provides the location for the two-operand-plus-result operations when fewer than three addresses are contained in the instruction. For example, the instruction "complement accumulator" is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction. In fact, any instruction that uses an accumulator without a second operand is an implied-mode instruction. For example, data manipulation instructions in a stack computer, such as ADD, are implied-mode instructions, since the operands are implied to be on top of stack.

Immediate Mode

In the immediate mode, the operand is specified in the instruction itself. In other words, an immediate-mode instruction has an operand field rather than an address field. The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction. Immediate-mode instructions are useful, for example, for initializing registers to a constant value.

Register and Register-Indirect Modes

Earlier, we mentioned that the address field of the instruction may specify either a memory location or a processor register. When the address field specifies a processor register, the instruction is said to be in the register mode. In this mode, the operands are in registers that reside within the processor of the computer. The particular register is selected from a register address field in the instruction format.

In the register-indirect mode, the instruction specifies a register in the processor whose content gives the address of the operand in memory. In other words, the selected register contains the memory address of the operand, rather than the operand itself. Before using a register-indirect mode instruction, the programmer must ensure that the memory address is available in the processor register. A reference to the register is then equivalent to specifying a memory address. The advantage of register-indirect mode is that the address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly.

An autoincrement or autodecrement mode is similar to the register-indirect mode, except that the register is incremented or decremented after (or before) its address value is used to access memory. When the address stored in the register refers to an array of data in memory, it is convenient to increment the register after each access to the array. This can be achieved by using a separate register-increment instruction. However, because it is such a common requirement, some computers incorporate an autoincrement mode that increments the content of the register containing the address after the memory data are accessed.

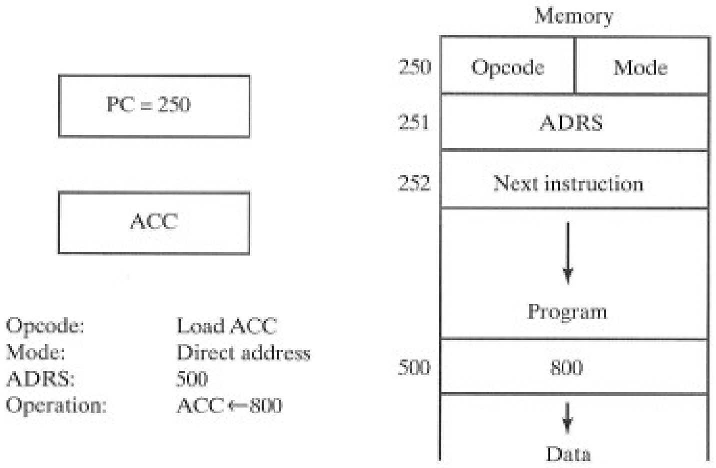
In the following instruction, an autoincrement mode is used to add the constant value 3 to the elements of an array addressed by register $R1$:

$$\text{ADD} \quad \text{(R1)+, 3} \qquad M[R1] \leftarrow M[R1] + 3, R1 \leftarrow R1 + 1$$

$R1$ is initialized to the address of the first element in the array. Then the ADD instruction is repeatedly executed until the addition of 3 to all elements of the array has occurred. The register transfer statement accompanying the instruction shows the addition of 3 to the memory location addressed by $R1$ and the incrementing of $R1$ in preparation for the next execution of the ADD on the next element in the array.

Direct Addressing Mode

In the direct addressing mode, the address field of the instruction gives the address of the operand in memory in a data transfer or data manipulation instruction. An example of a data transfer instruction is shown in Figure 11-4. The instruction in memory consists of two words. The first, at address 250, has the opcode for "load to *ACC*" and a mode field specifying a direct address. The second word of the



□ **FIGURE 11-4**
Example Demonstrating Direct Addressing for a Data Transfer Instruction

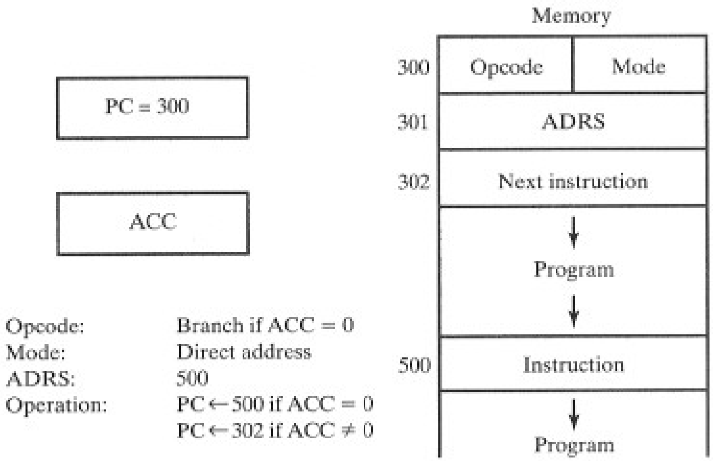
instruction, at address 251, contains the address field, symbolized by ADRS, and is equal to 500. The *PC* holds the address of the instruction, which is brought from memory using two memory accesses. Simultaneously with or after the completion of the first access, the *PC* is incremented to 251. Then the second access for ADRS occurs and the *PC* is again incremented. The execution of the instruction results in the operation

$$ACC \leftarrow M[\mathrm{ADRS}]$$

Since ADRS = 500 and M[500] = 800, the *ACC* receives the number 800. After the instruction is executed, the *PC* holds the number 252, which is the address of the next instruction in the program.

Now consider a branch-type instruction, as shown in Figure 11-5. If the contents of *ACC* equal 0, control branches to ADRS; otherwise, the program continues with the next instruction in sequence. When *ACC* = 0, the branch to address 500 is accomplished by loading the value of the address field ADRS into the *PC*. Control then continues with the instruction at address 500. When *ACC* ≠ 0, no branch occurs, and the *PC*, which was incremented twice during the fetch of the instruction, holds the address 302, the address of the next instruction in sequence.

Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated. To differentiate among the various addressing modes, it is useful to distinguish between the address part of the instruction, as given in the address field, and the address used by the control when executing the instruction. Recall that we refer to the latter as the effective address.



Opcode:	Branch if ACC = 0
Mode:	Direct address
ADRS:	500

Operation:	PC←500 if ACC = 0
	PC←302 if ACC ≠ 0

□ **FIGURE 11-5**
Example Demonstrating Direct Addressing in a Branch Instruction

Indirect Addressing Mode

In the indirect addressing mode, the address field of the instruction gives the address at which the effective address is stored in memory. The control unit fetches the instruction from memory and uses the address part to access memory again in order to read the effective address. Consider the instruction "load to *ACC*" given in Figure 11-4. If the mode specifies an indirect address, the effective address is stored in M[ADRS]. Since ADRS = 500 and M[ADRS] = 800, the effective address is 800. This means that the operand loaded into the *ACC* is the one found in memory at address 800 (not shown in the figure).

Relative Addressing Mode

Some addressing modes require that the address field of the instruction be added to the content of a specified register in the CPU in order to evaluate the effective address. Often, the register used is the *PC*. In the relative addressing mode, the effective address is calculated as follows:

$$\text{Effective address} = \text{Address part of the instruction} + \text{Contents of } PC$$

The address part of the instruction is considered to be a signed number that can be either positive or negative. When this number is added to the contents of the *PC*, the result produces an effective address whose position in memory is relative to the address of the next instruction in the program.

To clarify this with an example, let us assume that the *PC* contains the number 250 and the address part of the instruction contains the number 500, as in Figure 11-5, with the mode field specifying a relative address. The instruction at location 250 is read from memory during the fetch phase of the operation cycle, and the *PC* is incremented by 1 to 251. Since the instruction has a second word, the control unit reads the address field into a control register, and the *PC* is incremented to 252. The computation of the effective address for the relative addressing mode is 252 + 500 = 752. The result is that the operand associated with the instruction is 500 locations away, relative to the location of the next instruction.

Relative addressing is often used in branch-type instructions when the branch address is in a location close to the instruction word. Relative addressing produces more compact instructions, since the relative address can be specified with fewer bits than are required to designate the entire memory address.

Indexed Addressing Mode

In the indexed addressing mode, the content of an index register is added to the address part of the instruction to obtain the effective address. The index register may be a special CPU register or simply a register in a register file. We illustrate the use of indexed addressing by considering an array of data in memory. The address field of the instruction defines the beginning address of the array. Each operand in the array is stored in memory relative to the beginning address. The distance between the beginning address and the address of the operand is the index value stored in the register. Any operand in the array can be accessed with the same instruction, provided that the index register contains the correct index value. The index register can be incremented to facilitate access to consecutive operands.

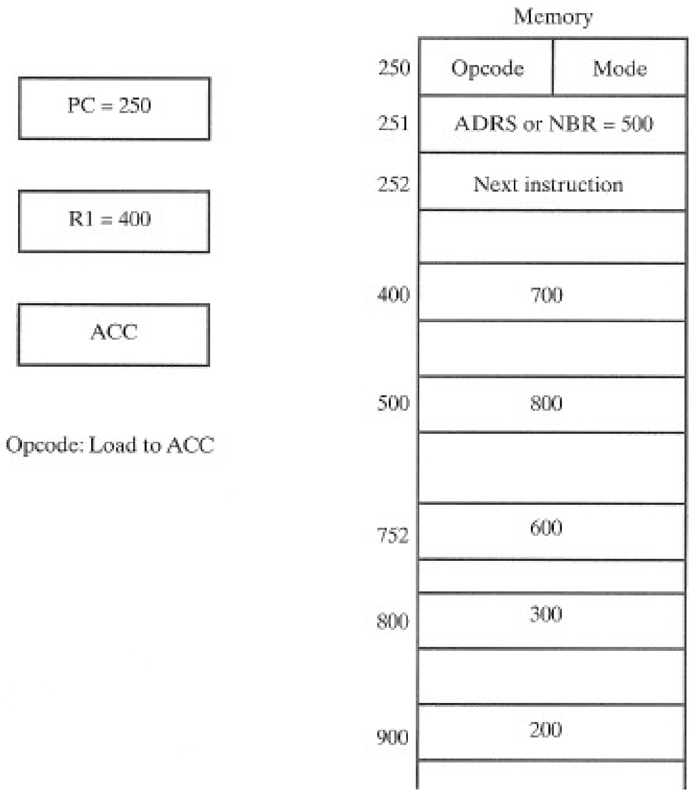
Some computers dedicate one CPU register to function solely as an index register. This register is addressed implicitly when an index-mode instruction is used. In computers with many processor registers, any CPU register can be used as an index register. In such a case, the index register to be used must be specified with a register field within the instruction format.

A specialized variation of the index mode is the base-register mode. In this mode, the contents of a base register are added to the address part of the instruction to obtain the effective address. This is similar to indexed addressing, except that the register is called a base register instead of an index register. The difference between the two modes is in the way they are used rather than in the way addresses are computed: an index register is assumed to hold an index number that is relative to the address field of the instruction; a base register is assumed to hold a base address, and the address field of the instruction gives a displacement relative to the base address.

Summary of Addressing Modes

In order to show the differences among the various modes, we will investigate the effect of the addressing mode on the instruction shown in Figure 11-6. The instruction in addresses 250 and 251 is "load to *ACC*," with the address field ADRS (or an operand NBR) equal to 500. The *PC* has the number 250 for fetching this instruction. The contents of a processor register *R*1 are 400, and the *ACC* receives the result after the instruction is executed. In the direct mode, the effective address is 500, and the operand to be loaded into the *ACC* is 800. In the immediate mode, the operand 500 is loaded into the *ACC*. In the indirect mode, the effective address is 800, and the operand is 300. In the relative mode, the effective address is 500 + 252 = 752, and the operand is 600. In the index mode, the effective address is 500 + 400 = 900, assuming that *R*1 is the index register. In the register mode, the operand is in *R*1, and 400 is loaded into the *ACC*. In the register-indirect mode, the effective address is the contents of *R*1, and the operand loaded into the *ACC* is 700.

Table 11-1 lists the value of the effective address and the operand loaded into the *ACC* for seven addressing modes. The table also shows the operation with a register transfer statement and a symbolic convention for each addressing mode. LDA is the symbol for the load-to-accumulator opcode. In the direct mode, we use the symbol ADRS for the address part of the instruction. The # symbol precedes the operand NBR in the immediate mode. The symbol ADRS enclosed in square brackets symbolizes an indirect address, which some compilers or assemblers designate with the symbol @. The symbol $ before the address makes the effective address relative to the *PC*. An index-mode instruction is recognized by the symbol of a register placed in parentheses after the address symbol. The register mode is indicated by giving the name of the processor register following LDA. In the register-indirect mode, the name of the register that holds the effective address is enclosed in parentheses.



□ **FIGURE 11-6**
Numerical Example for Addressing Modes

□ **TABLE 11-1**
Symbolic Convention for Addressing Modes

			Refers to Figure 11-6	
Addressing mode	**Symbolic convention**	**Register transfer**	**Effective address**	**Contents of *ACC***
Direct	LDA ADRS	$ACC \leftarrow M[ADRS]$	500	800
Immediate	LDA #NBR	$ACC \leftarrow NBR$	251	500
Indirect	LDA [ADRS]	$ACC \leftarrow M[M[ADRS]]$	800	300
Relative	LDA $ADRS	$ACC \leftarrow M[ADRS + PC]$	752	600
Index	LDA ADRS (R1)	$ACC \leftarrow M[ADRS + R1]$	900	200
Register	LDA R1	$ACC \leftarrow R1$	—	400
Register-indirect	LDA (R1)	$ACC \leftarrow M[R1]$	400	700

11-4 Instruction Set Architectures

Computers provide a set of instructions to permit computational tasks to be carried out. The instruction sets of different computers differ in several ways from each other. For example, the binary code assigned to the opcode field varies widely for different computers. Likewise, although a standard exists (see Reference 7), the symbolic name given to instructions varies for different computers. In comparison to these minor differences, however, there are two major types of instruction set architectures that differ markedly in the relationship of hardware to software: *Complex instruction set computers* (CISCs) provide hardware support for high-level language operations and have compact programs; *Reduced instruction set computers* (RISCs) emphasize simple instructions and flexibility that, when combined, provide higher throughput and faster execution. These two architectures can be distinguished by considering the properties that characterize their instruction sets.

A *RISC architecture* has the following properties:

1. Memory accesses are restricted to load and store instructions, and data manipulation instructions are register-to-register.
2. Addressing modes are limited in number.
3. Instruction formats are all of the same length.
4. Instructions perform elementary operations.

The goal of a RISC architecture is high throughput and fast execution. To achieve these goals, accesses to memory, which typically take longer than other elementary operations, are to be avoided, except for fetching instructions. A result of this view is the need for a relatively large register file. Because of the fixed instruction length, limited addressing modes, and elementary operations, the control unit of a RISC is comparatively simple and is typically hardwired. In addition, the underlying organization is universally a pipelined design as covered in Chapter 12.

A purely *CISC architecture* has the following properties:

1. Memory access is directly available to most types of instructions.
2. Addressing modes are substantial in number.
3. Instruction formats are of different lengths.
4. Instructions perform both elementary and complex operations.

The goal of the CISC architecture is to match more closely the operations used in programming languages and to provide instructions that facilitate compact programs and conserve memory. In addition, efficiencies in performance may result through a reduction in the number of instruction fetches from memory, compared with the number of elementary operations performed. Because of the high memory accessibility, the register files in a CISC are smaller than in a RISC. Also, because of the complexity of the instructions and the variability of the instruction formats, microprogrammed control is often used. In the quest for speed, the microprogrammed control in newer designs is likely to be controlling a pipelined datapath. CISC instructions are converted to a sequence of RISC-like operations that are processed by the RISC-like pipeline as discussed in detail in Chapter 12.

Actual instruction set architectures range between those which are purely RISC and those which are purely CISC. Nevertheless, there is a basic set of elementary operations that most computers include among their instructions. In this chapter, we will focus primarily on elementary instructions that are included in both CISC and RISC instruction sets. Most elementary computer instructions can be classified into three major categories: (1) data transfer instructions, (2) data manipulation instructions, and (3) program control instructions.

Data transfer instructions cause transfer of data from one location to another without changing the binary information content. Data manipulation instructions perform arithmetic, logic, and shift operations. Program control instructions provide decision-making capabilities and change the path taken by the program when executed in the computer. In addition to the basic instruction set, a computer may have other instructions that provide special operations for particular applications.

11-5 Data Transfer Instructions

Data transfer instructions move data from one place in the computer to another without changing the data. Typical transfers are between memory and processor registers, between processor registers and input and output registers, and among the processor registers themselves.

Table 11-2 gives a list of eight typical data transfer instructions used in many computers. Accompanying each instruction is a mnemonic symbol, the assembly language abbreviation recommended by an IEEE standard (Reference 6). Different computers, however, may use different mnemonics for the same instruction name. The load instruction is used to designate a transfer from memory to a processor register. The store instruction designates a transfer from a processor register into a memory word. The move instruction is used in computers with multiple processor registers to designate a transfer from one register to another. It is also used for data transfer between registers and memory and between two memory words.

□ **TABLE 11-2**
Typical Data Transfer Instructions

Name	Mnemonic
Load	LD
Store	ST
Move	MOVE
Exchange	XCH
Push	PUSH
Pop	POP
Input	IN
Output	OUT

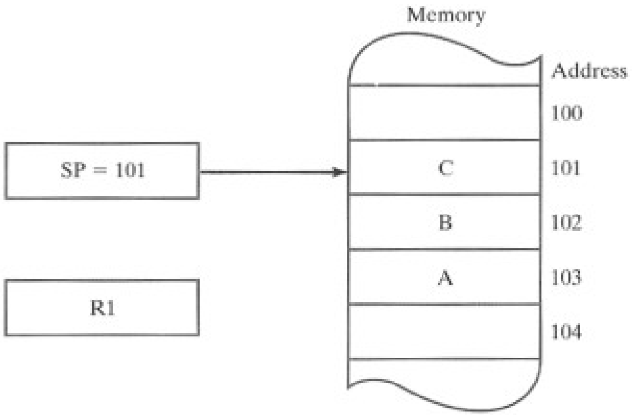
The exchange instruction exchanges information between two registers, between a register and a memory word, or between two memory words. The push and pop instructions are for stack operations described next.

Stack Instructions

The stack architecture introduced earlier possesses features that facilitate a number of data-processing and control tasks. A stack is used in some electronic calculators and computers for the evaluation of arithmetic expressions. Unfortunately, because of the negative effects on performance of having the stack reside primarily in memory, a stack in a computer typically handles only state information related to procedure calls and returns and interrupts, as explained in Section 11-8 and Section 11-9.

The stack instructions push and pop transfer data between a memory stack and a processor register or memory. The *push* operation places a new item onto the top of the stack. The *pop* operation removes one item from the stack so that the stack pops up. However, nothing is really physically pushed or popped in the stack. Rather, the memory stack is essentially a portion of a memory address space accessed by an address that is always incremented or decremented before or after the memory access. The register that holds the address for the stack is called a *stack pointer* (*SP*) because its value always points to TOS, the item at the top of the stack. Push and pop operations are implemented by decrementing or incrementing the stack pointer.

Figure 11-7 shows a portion of a memory organized as a stack that grows from higher to lower addresses. The stack pointer, *SP*, holds the binary address of the item that is currently on top of the stack. Three items are presently stored in the stack: *A*, *B*, and *C*, in consecutive addresses 103, 102, and 101, respectively. Item *C* is on top of the stack, so *SP* contains 101. To remove the top item, the stack is popped by reading the item at address 101 and incrementing *SP*. Item *B* is now on top of the stack, since



□ **FIGURE 11-7**
Memory Stack

SP contains address 102. To insert a new item, the stack is pushed by first decrementing *SP* and then writing the new item on top of the stack. Note that item *C* has been read out of the stack, but is not physically removed from it. This does not matter as far as the stack operation is concerned, because when the stack is pushed, a new item is written over it regardless of what was there before.

We assume that the items in the stack communicate with a data register *R*1 or a memory location X. A new item is placed on to the stack with the push operation as follows:

$$SP \leftarrow SP - 1$$

$$M[SP] \leftarrow R1$$

The stack pointer is decremented so that it points at the address of the next word. A memory write microoperation inserts the word from *R*1 onto the top of the stack. Note that *SP* holds the address of the top of the stack and that *M*[*SP*] denotes the memory word specified by the address presently in *SP*. An item is deleted from the stack with a pop operation as follows:

$$R1 \leftarrow M[SP]$$

$$SP \leftarrow SP + 1$$

The top item is read from the stack into *R*1. The stack pointer is then incremented to point at the next item in the stack, which is the new top of the stack.

The two microoperations needed for either the push or the pop operation are an access to memory through *SP* and an update of *SP*. Which microoperation is done first, and whether *SP* is updated by incrementing or decrementing it, depends on the organization of the stack. In Figure 11-7, the stack grows by decreasing the memory address. By contrast, a stack may be constructed to grow by increasing the memory address. In such a case, *SP* is incremented for the push operation and decremented for the pop operation. A stack may also be constructed so that *SP* points to the next empty location above the top of the stack. In that case, the sequence of microoperations must be interchanged.

A stack pointer is loaded with an initial value, which must be the bottom address of an assigned stack in memory. From then on, *SP* is automatically decremented or incremented with every push or pop operation. The advantage of a memory stack is that the processor can refer to it without having to specify an address, since the address is always available and automatically updated in the stack pointer.

The final pair of data transfer instructions, input and output, depend on the type of input-output used, as described next.

Independent versus Memory-Mapped I/O

Input and output (I/O) instructions transfer data between processor registers and input and output devices. These instructions are similar to load and store instructions, except that the transfers are to and from external registers instead of memory words. The computer is considered to have a certain number of input and output ports, with one or more ports dedicated to communication with a specific input or output device. A *port* is typically a register with input and/or output lines attached to the device. The particular port is chosen by an address, in a manner similar to the way an address selects a word in memory. Input and output instructions include an address field in their format, for specifying the particular port selected for the transfer of data.

Port addresses are assigned in two ways. In the *independent I/O system*, the address ranges assigned to memory and I/O ports are independent from each other. The computer has distinct input and output instructions, as listed in Table 11-2, containing a separate address field that is interpreted by the control and used to select a particular I/O port. Independent I/O addressing isolates memory and I/O selection, so that the memory address range is not affected by the port address assignment. For this reason, the method is also referred to as an *isolated I/O configuration*.

In contrast to independent I/O, *memory-mapped I/O*, assigns a subrange of the memory addresses for addressing I/O ports. There are no separate addresses for handling input and output transfers, since I/O ports are treated as memory locations in one common address range. Each I/O port is regarded as a memory location, similar to a memory word. Computers that adopt the memory-mapped scheme have no distinct input or output instructions, because the same instructions are used for manipulating both memory and I/O data. For example, the load and store instructions used for memory transfer are also used for I/O transfer, provided that the address associated with the instruction is assigned to an I/O port and not to a memory word. The advantage of this scheme is the simplicity that results with the same set of instructions serving for both memory and I/O access.

11-6 Data Manipulation Instructions

Data manipulation instructions perform operations on data and provide the computational capabilities of the computer. In a typical computer, data manipulation instructions are usually divided into three basic types:

1. Arithmetic instructions.
2. Logical and bit manipulation instructions.
3. Shift instructions.

A list of elementary data manipulation instructions looks very much like the list of microoperations given in Chapter 10. However, an instruction is typically processed by executing a *sequence* of one or more microinstructions. A microoperation is an elementary operation executed by the hardware of the computer under the control of the control unit. In contrast, an instruction may involve several elementary operations that fetch the instruction, bring the operands from appropriate processor registers, and store the result in the specified location.

Arithmetic Instructions

The four basic arithmetic instructions are addition, subtraction, multiplication, and division. Most computers provide instructions for all four operations. Some small computers, however, have only addition and subtraction instructions; on such computers, multiplication and division must be carried out by means of programs. The four basic arithmetic operations are sufficient for formulating solutions to any numerical problem when they are used with numerical analysis methods.

A list of typical arithmetic instructions is given in Table 11-3. The increment instruction adds one to the value stored in a register or memory word. A common characteristic of the increment operation, when executed on a computer word, is that a binary number of all 1's produces a result of all 0's when incremented. The decrement instruction subtracts one from a value stored in a register or memory word. When decremented, a number of all 0's produces a number of all 1's.

The add, subtract, multiply, and divide instructions may be available for different types of data. The data type assumed to be in processor registers during the execution of these arithmetic operations is included in the definition of the opcode. An arithmetic instruction may specify unsigned or signed integers, binary or decimal numbers, or floating-point data. The arithmetic operations with binary integers were presented in Chapter 1 and Chapter 5. The floating-point representation is used for scientific calculations and is presented in the next section.

The number of bits in any register is finite; therefore, the results of arithmetic operations are of finite precision. Most computers provide special instructions to facilitate double-precision arithmetic. A carry flip-flop is used to store the carry from an operation. The instruction "add with carry" performs the addition with two operands plus the value of the carry from the previous computation. Similarly, the "subtract with borrow" instruction subtracts two operands and a borrow that may have resulted from a previous operation. The subtract reverse instruction reverses the order of the operands, performing $B - A$ instead of $A - B$. The negate instruction performs the 2's complement of a signed number, which is equivalent to multiplying the number by -1.

□ **TABLE 11-3**
Typical Arithmetic Instructions

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Subtract reverse	SUBR
Negate	NEG

Logical and Bit Manipulation Instructions

Logical instructions perform binary operations on words stored in registers or memory words. They are useful for manipulating individual bits or a group of bits that represent binary-coded information. Logical instructions consider each bit of the operand separately and treat it as a Boolean variable. By proper application of the logical instructions, it is possible to change bit values, to clear a group of bits, or to insert new bit values into operands stored in registers or memory.

Some typical logical and bit manipulation instructions are listed in Table 11-4. The clear instruction causes the specific operand to be replaced by 0's. The set instruction causes the operand to be replaced by 1's. The complement instruction inverts all the bits of the operand. The AND, OR, and XOR instructions produce the corresponding logical operations on individual bits of the operand. Although logical instructions perform Boolean operations, when used on words they often are viewed as performing bit manipulation operations. There are three bit manipulation operations possible: A selected bit can be cleared to 0, set to 1, or complemented. The three logical instructions are usually applied to do just that.

The AND instruction is used to clear a bit or a selected group of bits of an operand to 0. For any Boolean variable X, the relationship $X \cdot 0 = 0$ dictates that a binary variable ANDed with a 0 produces a 0; and similarly, the relationship $X \cdot 1 = X$ dictates that the variable does not change when ANDed with a 1. Therefore, the AND instruction is used to selectively clear bits of an operand by ANDing the operand with a word that has 0's in the bit positions that must be cleared and 1's in the bit positions that must remain the same. The AND instruction is also called a *mask* because, by inserting 0's, it masks a selected portion of an operand. AND is also sometimes referred to as a *bit clear* instruction.

The OR instruction is used to set a bit or a selected group of bits of an operand to 1. For any Boolean variable X, the relationship $X + 1 = 1$ dictates that a binary variable ORed with a 1 produces a l; similarly, the relationship $X + 0 = X$ dictates that the variable does not change when ORed with a 0. Therefore, the OR instruction can be used to selectively set bits of an operand by ORing the operand with a word with 1's in the bit positions that must be set to 1. The OR instruction is sometimes called a *bit set* instruction.

□ **TABLE 11-4**
Typical Logical and Bit Manipulation Instructions

Name	Mnemonic
Clear	CLR
Set	SET
Complement	NOT
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC

The XOR instruction is used to selectively complement bits of an operand. This is because of the Boolean relationships $X \oplus 1 = \overline{X}$ and $X \oplus 0 = X$. A binary variable is complemented when XORed with a 1, but does not change value when XORed with a 0. The XOR instruction is sometimes called a *bit complement* instruction.

Other bit manipulation instructions included in Table 11-4 clear, set, or complement the carry bit. Additional instructions clear, set, or complement other status bits or flag bits in a similar manner.

Shift Instructions

Instructions to shift the content of an operand are provided in several varieties. Shifts are operations in which the bits of the operand are moved to the left or to the right. The incoming bit shifted in at the end of the word determines the type of shift. Instead of using just a 0, as for sl and sr in Chapter 10, here we add further possibilities. The shift instructions may specify either logical shifts, arithmetic shifts, or rotate-type operations.

Table 11-5 lists four types of shift instructions. The logical shift inserts 0 into the incoming bit position after the shift. Arithmetic shifts conform to the rules for shifting two's complement signed numbers. The arithmetic shift right instruction preserves the sign bit in the leftmost position. The value of the sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unchanged. The arithmetic shift left instruction inserts 0 into the incoming bit in the rightmost position and is identical to the logical shift left instruction. The two instructions may differ, however, in that an arithmetic shift left may set the overflow status bit *V*, while a logical shift left does not affect *V*.

The rotate instructions produce a circular shift: the values shifted out of the outgoing bit of the word are not lost, as in a logical shift, but are rotated back into the incoming bit. The rotate-with-carry instructions treat the carry bit as an extension of the register whose word is being rotated. Thus, a rotate left with carry transfers the carry bit into the incoming bit in the rightmost bit position of the register, transfers the outgoing bit from the leftmost bit of the register into the carry, and

□ **TABLE 11-5**
Typical Shift Instructions

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right with carry	RORC
Rotate left with carry	ROLC

shifts the entire register to the left. Some computers have a multiple-field format for the shift instruction. One field contains the opcode, and the others specify the type of shift and the number of positions that an operand is to be shifted. A shift instruction may include the following five fields:

OP REG TYPE RL COUNT

OP is the opcode field for specifying a shift, and REG is a register address that specifies the location of the operand. TYPE is a 2-bit field that specifies one of the four types of shifts (logical, arithmetic, rotate, and rotate with carry), while RL is a 1-bit field that specifies whether a shift is to the right or the left. COUNT is a k-bit field that specifies shifts of up to $2^k - 1$ positions. With such a format, it is possible to specify the type of shift, the direction of the shift, and the number of positions to be shifted, all in one instruction.

11-7 Floating-Point Computations

In many scientific calculations, the range of numbers is very large. In a computer, the way to express such numbers is in floating-point notation. The floating-point number has two parts, one containing the sign of the number and a *fraction* (sometimes called a *mantissa*) and the other designating the position of the radix point in the number and called the *exponent*. For example, the decimal number +6132.789 is represented in floating-point notation as

Fraction	Exponent
+.6132789	+04

The value of the exponent indicates that the actual position of the decimal point is four positions to the right of the indicated decimal point in the fraction. This representation is equivalent to the scientific notation $+.6132789 \times 10^{+4}$. Decimal floating-point numbers are interpreted as representing a number in the form

$$F \times 10^E$$

where F is the fraction and E the exponent. Only the fraction and the exponent are physically represented in computer registers; radix 10 and the decimal point of the fraction are assumed and are not shown explicitly. A floating-point binary number is represented in a similar manner, except that it uses radix 2 for the exponent. For example, the binary number +1001.11 is represented with an 8-bit fraction and 6-bit exponent as

Fraction	Exponent
01001110	000100

The fraction has a 0 in the leftmost position to denote a plus. The binary point of the fraction follows the sign bit, but is not shown in the register. The exponent has the equivalent binary number +4. The floating-point number is equivalent to

$$F \times 2^{E} = +(0.1001110)_2 \times 2^{+4}$$

A floating-point number is said to be *normalized* if the most significant digit of the fraction is nonzero. For example, the decimal fraction 0.350 is normalized, but 0.0035 is not. Normalized numbers provide the maximum possible precision for the floating-point number. A zero cannot be normalized because it does not have a nonzero digit; it is usually represented in floating-point by all 0's in both the fraction and the exponent.

Floating-point representation increases the range of numbers that can be accommodated in a given register. Consider a computer with 48-bit registers. Since one bit must be reserved for the sign, the range of signed integers will be $\pm(2^{47} - 1)$, which is approximately $\pm 10^{14}$. The 48 bits can be used to represent a floating-point number, with one bit for the sign, 35 bits for the fraction, and 12 bits for the exponent. The largest positive or negative number that can be accommodated is thus

$$\pm(1 - 2^{-35}) \times 2^{+2047}$$

This number is derived from a fraction that contains 35 1's, and an exponent with a sign bit and 11 1's. The maximum exponent is $2^{11} - 1$, or 2047. The largest number that can be accommodated is approximately equivalent to decimal 10^{615}. Although a much larger range is represented, there are still only 48 bits in the representation. As a consequence, exactly the same number of numbers are represented. Hence, the range is traded for the precision of the numbers, which is reduced from 48 bits to 35 bits.

Arithmetic Operations

Arithmetic operations with floating-point numbers are more complicated than with integer numbers, and their execution takes longer and requires more complex hardware. Adding and subtracting two numbers requires that the radix points be aligned, since the exponent parts must be equal before adding or subtracting the fractions. The alignment is done by shifting one fraction and correspondingly adjusting its exponent until it is equal to the other exponent. Consider the sum of the following floating-point numbers:

$$\begin{aligned} &.5372400 \times 10^{2} \\ +\ &.1580000 \times 10^{-1} \end{aligned}$$

It is necessary that the two exponents be equal before the fractions can be added. We can either shift the first number three positions to the left or shift the second number three positions to the right. When the fractions are stored in registers, shifting to the left causes a loss of the most significant digits. Shifting to the right causes a loss of the least significant digits. The second method is preferable because it only reduces the precision, whereas the first method may cause an error. The usual alignment procedure is to shift the fraction with the smaller exponent to the right by a number of places equal to the difference between the exponents. After this is done, the fractions can be added:

$$\begin{array}{r} .5372400 \times 10^2 \\ +\ .0001580 \times 10^2 \\ \hline .5373980 \times 10^2 \end{array}$$

When two normalized fractions are added, the sum may contain an overflow digit. An overflow can be corrected by shifting the sum once to the right and incrementing the exponent. When two numbers are subtracted, the result may contain most significant zeros in the fraction, as shown in the following example:

$$\begin{array}{r} .56780 \times 10^5 \\ -\ .56430 \times 10^5 \\ \hline .00350 \times 10^5 \end{array}$$

A floating-point number that has a 0 in the most significant position of the fraction is not normalized. To normalize the number, it is necessary to shift the fraction to the left and decrement the exponent until a nonzero digit appears in the first position. In the preceding example, it is necessary to shift left twice to obtain $.35000 \times 10^3$. In most computers, a normalization procedure is performed after each operation to ensure that all results are in normalized form.

Floating-point multiplication and division do not require an alignment of the fractions. Multiplication can be performed by multiplying the two fractions and adding the exponents. Division is accomplished by dividing the fractions and subtracting the exponents. In the examples shown, we used decimal numbers to demonstrate arithmetic operations on floating-point numbers. The same procedure applies to binary numbers, except that the base of the exponent is 2 instead of 10.

Biased Exponent

The sign and fraction part of a floating-point number is usually a signed-magnitude representation. The exponent representation employed in most computers is known as a *biased exponent*. The bias is an excess number added to the exponent so that, internally, all exponents become positive. As a consequence, the sign of the exponent is removed from being a separate entity.

Consider, for example, the range of decimal exponents from -99 to $+99$. This is represented by two digits and a sign. If we use an excess 99 bias, then the biased exponent e will be equal to $e = E + 99$, where E is the actual exponent. For $E = -99$, we have $e = -99 + 99 = 0$; and for $E = +99$, we have $e = 99 + 99 = 198$. In this way, the biased exponent is represented in a register as a positive number in the range from 000 to 198. Positive-biased exponents have a range of numbers from 099 to 198. Subtraction of the bias, 99, gives the positive values from 0 to $+99$. Negative-biased exponents have a range from 098 to 000. Subtraction of 99 gives the negative values from -1 to -99.

The advantage of biased exponents is that the resulting floating-point numbers contain only positive exponents. It is then simpler to compare the relative magnitude between two numbers without being concerned with the signs of their exponents. Another advantage is that the most negative exponent converts to a biased exponent with all 0's. The floating-point representation of zero is then a zero fraction and a zero biased exponent, which is the smallest possible exponent.

Standard Operand Format

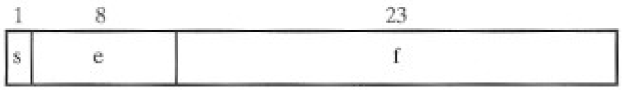
Arithmetic instructions that perform operations with floating-point data often use the suffix F. Thus, ADDF is an add instruction with floating-point numbers. There are two standard formats for representing a floating-point operand: the single-precision data type, consisting of 32 bits, and the double-precision data type, consisting of 64 bits. When both types of data are available, the single-precision instruction mnemonic uses an FS suffix, and the double precision uses FL (for "floating-point long").

The format of the IEEE standard (see Reference 7) single-precision floating-point operand is shown in Figure 11-8. It consists of 32 bits. The sign bit *s* designates the sign for the fraction. The biased exponent *e* contains 8 bits and uses an excess 127 number. The fraction *f* consists of 23 bits. The binary point is assumed to be immediately to the left of the most significant bit of the *f* field. In addition, an implied 1 bit is inserted to the left of the binary point, which, in effect, expands the number to 24 bits representing a value from 1.0_2 to $1.11...1_2$. The component of the binary floating-point number that consists of a leading bit to the left of the implied binary point, together with the fraction in the field, is called the *significand*. Following are some examples of field values and the corresponding significands:

f Field	Significand	Decimal Equivalent
100 . . . 0	1.100 . . . 0	1.50
010 . . . 0	1.010 . . . 0	1.25
000 . . . 0	1.000 . . . 0*	1.00*

*Assuming the exponent is not equal to 00 . . . 0.

Even though the *f* field by itself may not be normalized, the significant is always normalized because it has a nonzero bit in the most significant position. Since normalized numbers must have a nonzero most significant bit, this 1 bit is not included explicitly in the format, but must be inserted by the hardware during arithmetic computations. The exponent field uses an excess 127 bias value for normalized numbers. The range of valid exponents is from −126 (represented as



□ **FIGURE 11-8**
IEEE Floating-Point Operand Format

00000001) through +127 (represented as 11111110). The maximum (11111111) and minimum (00000000) values for the e field are reserved to indicate exceptional conditions. Table 11-6 shows the biased and actual values of some exponents.

Normalized numbers are numbers that can be expressed as floating-point operands in which the e field is neither all 0's nor all 1's. The value of the number is derived from the three fields in the format of Figure 11-8 using the formula

$$(-1)^s 2^{e-127} \times (1.f)$$

The most positive normalized number that can be obtained has a 0 for the sign bit for a positive sign, a biased exponent equal to 254, and an f field with 23 1's. This gives an exponent $E = 254 - 127 = 127$. The significant is equal to $1 + 1 - 2^{-23} = 2 - 2^{-23}$. The maximum positive number that can be accommodated is

$$+2^{127} \times (2\,2\,2^{-23})$$

The smallest positive normalized number has a biased exponent equal to 00000001 and a fraction of all 0's. The exponent is $E = 1 - 127 = -126$, and the significant is equal to 1.0. The smallest positive number that can be accommodated is $+2^{-126}$. The corresponding negative numbers are the same, except that the sign bit is negative. As mentioned before, exponents with all 0's or all 1's (decimal 255) are reserved for the following special conditions:

1. When $e = 255$ and $f = 0$, the number represents plus or minus infinity. The sign is determined from the sign bit s.
2. When $e = 255$ and $f \neq 0$, the representation is considered to be *not a number*, or NaN, regardless of the sign value. NaNs are used to signify invalid operations, such as the multiplication of zero by infinity.
3. When $e = 0$ and $f = 0$, the number denotes plus or minus zero.
4. When $e = 0$, and $f \neq 0$, the number is said to be denormalized. This is the name given to numbers with a magnitude less than the minimum value that is represented in the normalized format.

□ **TABLE 11-6**
Evaluating Biased Exponents

Exponent E in decimal	Biased exponent $e = E + 127$	
	Decimal	Binary
−126	−126 + 127 = 1	00000001
−001	−001 + 127 = 126	01111110
000	000 + 127 = 127	01111111
+001	001 + 127 = 128	10000000
+126	126 + 127 = 253	11111101
+127	127 + 127 = 254	11111110

11-8 Program Control Instructions

The instructions of a program are stored in successive memory locations. When processed by the control, the instructions are read from consecutive memory locations and executed one by one. Each time an instruction is fetched from memory, the *PC* is incremented so that it contains the address of the next instruction in sequence. In contrast, a program control instruction, when executed, may change the address value in the *PC* and cause the flow of control to be altered. The change in the *PC* as a result of the execution of a program control instruction causes a break in the sequence of execution of instructions. This is an important feature of digital computers, since it provides control over the flow of program execution and a capability of branching to different program segments, depending on previous computations.

Some typical program control instructions are listed in Table 11-7. The branch and jump instructions are often used interchangeably to mean the same thing, although sometimes they are used to denote different addressing modes. For example, the jump may use direct or indirect addressing, whereas the branch uses relative addressing. The branch (or jump) is usually a one-address instruction. When executed, the branch instruction causes a transfer of the effective address into the *PC*. Since the *PC* contains the address of the instruction to be executed next, the next instruction will be fetched from the location specified by the effective address.

Branch and jump instructions may be conditional or unconditional. An unconditional branch instruction causes a branch to the specified effective address without any conditions. The conditional branch instruction specifies a condition that must be met in order for the branch to occur, such as the value in a specified register being negative. If the condition is met, the *PC* is loaded with the effective address, and the next instruction is taken from this address. If the condition is not met, the *PC* is not changed, and the next instruction is taken from the next location in sequence.

The skip instruction does not need an address field. A conditional skip instruction will skip the next instruction if the specified condition is met. This is

□ TABLE 11-7
Typical Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Skip next instruction	SKP
Call procedure	CALL
Return from procedure	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TEST

accomplished by incrementing the *PC* during the execute phase of the instruction, in addition to incrementing it during the fetch phase. If the condition is not met, control proceeds to the next instruction in sequence, at which point the programmer may insert an unconditional branch instruction. Thus, a conditional skip instruction followed by an unconditional branch instruction causes a branch if the condition is not met. This contrasts with a single conditional branch instruction, which causes a branch if the condition *is* met. Since the skip involves the execution of two instructions, it is slower and uses more instruction memory.

The call and return instructions are used in conjunction with procedures. Their performance and implementation are discussed later in this section.

The compare instruction performs a comparison via a subtraction, with the difference not retained. Instead, the comparison causes a conditional branch, changes the contents of a register, or sets or resets stored status bits. Similarly, the test instruction performs the logical AND of two operands without retaining the result and executes one of the actions listed for the compare instruction.

Based on their three possible actions, compare and test instructions are viewed to be of three distinct types, depending upon the way in which conditional decisions are handled. The first type executes the entire decision as a single instruction. For example, the contents of two registers can be compared and a branch or jump taken if the contents are equal. Since there are two register addresses and a memory address involved, such an instruction requires three addresses. The second type of compare and test instruction also uses three addresses, all of which are register addresses. Considering the same example, if the contents of the first two registers are equal, a 1 is placed in the third register. If the contents are not equal, then a 0 is placed in the third register. These two types of instruction avoid the use of stored status bits. In the first case, no such bit is required, and in the second case, a register is used to simulate the presence of a status bit. The third type of compare and test, with the most complex structure, has compare and test operations that set or reset stored status bits. Branch or jump instructions are then used to conditionally change the program sequence. This third type of compare and test instruction is the focus of discussion in the next subsection.

Conditional Branch Instructions

A conditional branch instruction is a branch instruction that may or may not cause a transfer of control, depending on the value of stored bits in the *PSR*. Each conditional branch instruction tests a different combination of status bits for a condition. If the condition is true, control is transferred to the effective address. If the condition is false, the program continues with the next instruction.

Table 11-8 gives a list of conditional branch instructions that depend directly on the bits in the *PSR*. In most cases, the instruction mnemonic is constructed with the letter B (for "branch") and a letter for the name of the status bit. The letter N (for "not") is included if the status bit is tested for a 0 condition. Thus, BC is a branch if carry = 1, and BNC is branch if carry = 0.

The zero status bit Z is used to check whether the result of an ALU operation is equal to zero. The carry bit C is used to check the carry after the addition

□ TABLE 11-8
Conditional Branch Instructions Relating to Status Bits in the PSR

Branch condition	Mnemonic	Test condition
Branch if zero	BZ	$Z = 1$
Branch if not zero	BNZ	$Z = 0$
Branch if carry	BC	$C = 1$
Branch if no carry	BNC	$C = 0$
Branch if minus	BN	$N = 1$
Branch if plus	BNN	$N = 0$
Branch if overflow	BV	$V = 1$
Branch if no overflow	BNV	$V = 0$

or the borrow after the subtraction of two operands in the ALU. It is also used in conjunction with shift instructions to check the value of the outgoing bit. The sign bit N reflects the state of the leftmost bit of the output from the ALU. $N = 0$ denotes a positive sign and $N = 1$ a negative sign. These instructions can be used to check the value of the leftmost bit, whether it represents a sign or not. The overflow bit V is used in conjunction with arithmetic operations with signed numbers.

As stated previously, the compare instruction performs a subtraction of two operands, say, $A - B$. The result of the operation is not transferred into a destination register, but the status bits are affected. The status bits provide information about the relative magnitude between A and B. Some computers provide special branch instructions that can be applied after the execution of a compare instruction. The specific conditions to be tested depend on whether the two numbers are considered to be unsigned or signed.

The relative magnitude between two unsigned binary numbers A and B can be determined by subtracting $A - B$ and checking the C and Z status bits. Most commercial computers consider the C status bit as a carry after addition and a borrow after subtraction. A borrow occurs when $A < B$ because the most significant position must borrow a bit to complete the subtraction. A borrow does not occur if $A \geq B$, because the difference $A - B$ is positive. The condition for borrowing is the inverse of the condition for carrying when the subtraction is done by taking the 2's complement of B. Computers that use the C status bit as a borrow after a subtraction complement the output carry after adding the 2's complement of the subtrahend and call this bit a borrow. The technique is typically applied to all instructions that use subtraction within the functional unit, not just the subtract instruction. For example, it applies to compare instructions.

The conditional branch instructions for unsigned numbers are listed in Table 11-9. It is assumed that a previous instruction has updated status bits C and Z after a subtraction $A - B$ or some other similar instruction. The words "higher," "lower," and "equal" are used to denote the relative magnitude between two unsigned numbers. The two numbers are equal if $A = B$. This is determined from

□ **TABLE 11-9**
Conditional Branch Instructions for Unsigned Numbers

Branch condition	Mnemonic	Condition	Status bits*
Branch if higher	BH	$A > B$	$C + Z = 0$
Branch if higher or equal	BHE	$A \geq B$	$C = 0$
Branch if lower	BL	$A < B$	$C = 1$
Branch if lower or equal	BLE	$A \leq B$	$C + Z = 1$
Branch if equal	BE	$A = B$	$Z = 1$
Branch if not equal	BNE	$A \neq B$	$Z = 0$

*Note that C here is a borrow bit.

the zero status bit Z, which is equal to 1 because $A - B = 0$. A is lower than B and the borrow $C = 1$ when $A < B$. For A to be lower than or equal to B $(A \leq B)$, we must have $C = 1$ or $Z = 1$. The relationship $A > B$, is the inverse of $A \leq B$ and is detected from the complemented condition of the status bits. Similarly, $A \geq B$ is the inverse of $A < B$, and $A \neq B$ is the inverse of $A = B$.

The conditional branch instructions for signed numbers are listed in Table 11-10. Again, it is assumed that a previous instruction has updated the status bits N, V, and Z after a subtraction $A - B$. The words "greater," "less," and "equal" are used to denote the relative magnitude between two signed numbers. If $N = 0$, the sign of the difference is positive, and A must be greater than or equal to B, provided that $V = 0$, indicating that no overflow occurred. An overflow causes a sign reversal, as discussed in Section 5-4. This means that if $N = 1$ and $V = 1$, there was a sign reversal, and the result should have been positive, which makes A greater than or equal to B. Therefore, the condition $A \geq B$ is true if both N and V are equal to 0 or both are equal to 1. This is the complement of the exclusive-OR operation.

For A to be greater than but not equal to B $(A > B)$, the result must be positive and nonzero. Since a zero result gives a positive sign, we must ensure that the Z bit is 0 to exclude the possibility that $A = B$. Note that the condition $(N \oplus V) + Z = 0$ means that both the exclusive-OR operation and the Z bit must be equal to 0. The other two conditions in the table can be derived in a similar manner. The conditions BE (branch on equal) and BNE (branch on not equal) given for unsigned numbers apply to signed numbers as well and can be determined from $Z = 1$ and $Z = 0$, respectively.

□ **TABLE 11-10**
Conditional Branch Instructions for Signed Numbers

Branch condition	Mnemonic	Condition	Status bits
Branch if greater	BG	$A > B$	$(N \oplus V) + Z = 0$
Branch if greater or equal	BGE	$A \geq B$	$N \oplus V = 0$
Branch if less	BL	$A < B$	$N \oplus V = 1$
Branch if less or equal	BLE	$A \leq B$	$(N \oplus V) + Z = 1$

Procedure Call and Return Instructions

A *procedure* is a self-contained sequence of instructions that performs a given computational task. During the execution of a program, a procedure may be called to perform its function many times at various points in the program. Each time the procedure is called, a branch is made to the beginning of the procedure to start executing its set of instructions. After the procedure has been executed, a branch is made again to return to the main program. A procedure is also called a *subroutine.*

The instruction that transfers control to a procedure is known by different names, including call procedure, call subroutine, jump to subroutine, branch to subroutine, and branch and link. We will refer to the routine containing the procedure call as the calling procedure. The call procedure instruction has a one-address field and performs two operations. First, it stores the value of the *PC*, which is the address following the call procedure instruction, in a temporary location. This address is called the *return address*, and the corresponding instruction is the *continuation point* in the calling procedure. Second, the address in the call procedure instruction—the address of the first instruction in the procedure—is loaded into the *PC*. When the next instruction is fetched, it comes from the called procedure.

The final instruction in every procedure must be a return to the calling procedure. The return instruction takes the address that was stored by the call procedure instruction and places it in the *PC*. This results in a transfer of program execution back to the continuation point in the calling procedure.

Different computers use different temporary locations for storing the return address. Some computers store it in a fixed location in memory, some store it in a processor register, and some store it in a memory stack. The advantage of using a stack for the return address is that, when a succession of procedures are called, the sequential return address can be pushed onto the stack. The return instruction causes the stack to pop, and the contents of the top of the stack are then transferred to the *PC*. In this way, a return is always to the program that last called the procedure. A procedure call instruction using a stack is implemented with the following microoperations:

$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PC$	Store return address on stack
$PC \leftarrow$ Effective address	Transfer control to procedure

The return instruction is implemented by popping the stack and transferring the return address to the *PC*:

$PC \leftarrow M[SP]$	Transfer return address to *PC*
$SP \leftarrow SP + 1$	Increment stack pointer

By using a procedure stack, all return addresses are automatically stored by the hardware in the memory stack. Thus, the programmer does not have to be concerned about managing the return addresses for procedures called from within procedures.

11-9 Program Interrupt

A program interrupt is used to handle a variety of situations that require a departure from the normal program sequence. A program interrupt transfers control from a program that is currently running to another service program as a result of an externally or internally generated request. Control returns to the original program after the service program is executed. In principle, the interrupt procedure is similar to a call procedure, except in three respects:

1. The interrupt is usually initiated at an unpredictable point in the program by an external or internal signal, rather than the execution of an instruction.
2. The address of the service program that processes the interrupt request is determined by a hardware procedure, rather than the address field of an instruction.
3. In response to an interrupt, it is necessary to store information that defines all or part of the contents of the register set, rather than storing only the program counter.

After the computer has been interrupted and the appropriate service program executed, the computer must return to exactly the same state that it was in before the interrupt occurred. Only if this happens will the interrupted program be able to resume exactly as if nothing happened. The state of the computer at the end of an execution of an instruction is determined from the contents of the register set. In addition to containing the condition codes, the *PSR* can specify what interrupts are allowed to occur and whether the computer is operating in user or system mode. Most computers have a resident operating system that controls and supervises all other programs. When the computer is executing a program that is part of the operating system, the computer is placed in system mode, in which certain instructions are privileged and can be executed in the system mode only. The computer is in user mode when it executes user programs, in which case it cannot execute the privileged instructions. The mode of the computer at any given time is determined from a special status bit or bits in the *PSR*.

Some computers store only the program counter when responding to an interrupt. In such computers, the program that performs the data processing for servicing the interrupt must include instructions to store the essential contents of the register set. Other computers store the entire register set automatically in response to an interrupt. Some computers have two sets of processor registers, so that when the program switches from user to system mode in response to an interrupt, it is not necessary to store the contents of processor registers because each computer mode employs its own set of registers.

The hardware procedure for processing interrupts is very similar to the execution of a procedure call instruction. The contents of the register set of the processor are temporarily stored in memory, typically by being pushed onto a memory stack, and the address of the first instruction of the interrupt service program is loaded into the *PC*. The address of the service program is chosen by the hardware. Some computers assign one memory location for the beginning address of the service program:

the service program must then determine the source of the interrupt and proceed to service it. Other computers assign a separate memory location for each possible interrupt source. Sometimes, the interrupt source hardware itself supplies the address of the service routine. In any case, the computer must possess some form of hardware procedure for selecting a branch address for servicing the interrupt.

Most computers will not respond to an interrupt until the instruction that is in the process of being executed is completed. Then, just before going to fetch the next instruction, the control checks for any interrupt signals. If an interrupt has occurred, control goes to a hardware interrupt cycle. During this cycle, the contents of some part or all of the register set are pushed onto the stack. The branch address for the particular interrupt is then transferred to the *PC*, and the control goes to fetch the next instruction, which is the beginning of the interrupt service routine. The last instruction in the service routine is a return from the interrupt instruction. When this return is executed, the stack is popped to retrieve the return address, which is transferred to the *PC* as well as any stored contents of the rest of the register set, which are transferred back to the appropriate registers.

Types of Interrupts

The three major types of interrupts that cause a break in the normal execution of a program are as follows:

1. External interrupts.
2. Internal interrupts.
3. Software interrupts.

External interrupts come from input or output devices, from timing devices, from a circuit monitoring the power supply, or from any other external source. Conditions that cause external interrupts are an input or output device requesting a transfer of data, an external device completing a transfer of data, the time-out of an event, or an impending power failure. A time-out interrupt may result from a program that is in an endless loop and thus exceeds its time allocation. A power failure interrupt may have as its service program a few instructions that transfer the complete contents of the register set of the processor into a nondestructive memory such as a disk in the few milliseconds before power ceases.

Internal interrupts arise from the invalid or erroneous use of an instruction or data. Internal interrupts are also called *traps*. Examples of interrupts caused by internal conditions are an arithmetic overflow, an attempt to divide by zero, an invalid opcode, a memory stack overflow, and a protection violation. A *protection violation* is an attempt to address an area of memory that is not supposed to be accessed by the currently executing program. The service programs that process internal interrupts determine the corrective measure to be taken in each case.

External and internal interrupts are initiated by the hardware of the computer. By contrast, a *software interrupt* is initiated by executing an instruction. The software interrupt is a special call instruction that behaves like an interrupt rather than a procedure call. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program. Typical use of the software interrupt is associated with a system call instruction. This instruction provides a means for switching from user mode to system mode. Certain operations in the computer may be performed by the operating system only in system mode. For example, a complex input or output procedure is done in system mode. In contrast, a program written by a user must run in user mode. When an input or output transfer is required, the user program causes a software interrupt, which stores the contents of the *PSR* (with the mode bit set to "user"), loads new *PSR* contents (with the mode bit set to "system"), and initiates the execution of a system program. The calling program must pass information to the operating system in order to specify the particular task that is being requested.

An alternative term for an interrupt is an *exception*, which may apply only to internal interrupts or to all interrupts, depending on the particular computer manufacturer. As an illustration of the use of the two terms, what one programmer calls interrupt-handling routines may be referred to as exception-handling routines by another programmer.

Processing External Interrupts

External interrupts may have single or multiple interrupt input lines. If there are more interrupt sources than there are interrupt inputs in the computer, two or more sources are ORed to form a common line. An interrupt signal may originate at any time during program execution. To ensure that no information is lost, the computer usually acknowledges the interrupt only after the execution of the current instruction is completed and only if the state of the processor warrants it.

Figure 11-9 shows a simplified external interrupt configuration. Four external interrupt sources are ORed to form a single interrupt input signal. Within the CPU is an enable-interrupt flip-flop (*EI*) that can be set or reset with two program instructions: enable interrupt (ENI) and disable interrupt (DSI). When *EI* is 0, the interrupt signal is neglected. When *EI* is 1 and the CPU is at the end of executing an instruction, the computer acknowledges the interrupt by enabling the interrupt acknowledge output *INTACK*. The interrupt source responds to *INTACK* by providing an interrupt vector address *IVAD* to the CPU. The program-controlled *EI* flip-flop allows the programmer to decide whether to use the interrupt facility. If a DSI instruction to reset *EI* has been inserted in the program, it means that the programmer does not want the program to be interrupted. The execution of an ENI instruction to set *EI* indicates that the interrupt facility will be active while the program is running.

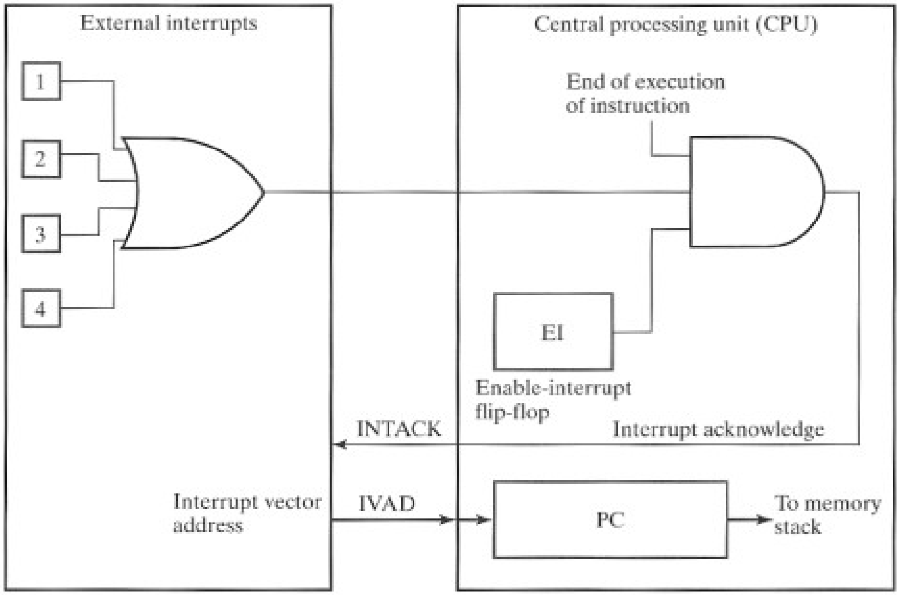
The computer responds to an interrupt request signal if $EI = 1$ and execution of the present instruction is completed. Typical microinstructions that implement the interrupt are as follows:

$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PC$	Store return address on stack
$SP \leftarrow SP - 1$	Decrement stack pointer

$M[SP] \leftarrow PSR$	Store processor status word on stack
$EI \leftarrow 0$	Reset enable-interrupt flip-flop
$INTACK \leftarrow 1$	Enable interrupt acknowledge
$PC \leftarrow IVAD$	Transfer interrupt vector address to *PC*
	Go to fetch phase.

The return address available in the *PC* is pushed onto the stack, and the *PSR* contents are pushed onto the stack. *EI* is reset to disable further interrupts. The program that services the interrupt can set *EI* with an instruction whenever it is appropriate to enable other interrupts. The CPU assumes that the external source will provide an *IVAD* in response to an *INTACK*. The *IVAD* is taken as the address of the first instruction of the program that services the interrupt. Obviously, a program must be written for that purpose and stored in memory.

The return from an interrupt is done with an instruction at the end of the service program that is similar to a return from a procedure. The stack is popped, and the return address is transferred to the *PC*. Since the *EI* flip-flop is usually included in the *PSR*, the value of *EI* for the original program is returned to *EI* when the old value of the *PSR* is returned. Thus, the interrupt system is enabled or disabled for the original program, as it was before the interrupt occurred.



□ **FIGURE 11-9**
Example of External Interrupt Configuration

11-10 CHAPTER SUMMARY

In this chapter, we defined the concepts of instruction set architecture and the components of an instruction and explored the effects on programs of the maximum address count per instruction, using both memory addresses and register addresses. This led to the definitions of four types of addressing architecture: memory-to-memory, register-to-register, single-accumulator, and stack. Addressing modes specify how the information in an instruction is interpreted in determining the effective address of an operand.

Reduced instruction set computers (RISCs) and complex instruction set computers (CISCs) are two broad categories of instruction set architecture. A RISC has as its goals high throughput and fast execution of instructions. In contrast, a CISC attempts to closely match the operations used in programming languages and facilitates compact programs.

Three categories of elementary instructions are data transfer, data manipulation, and program control. In elaborating data transfer instructions, the concept of the memory stack appears. Transfers between the CPU and I/O are addressed by two different methods: independent I/O, with a separate address space, and memory-mapped I/O, which uses part of the memory address space. Data manipulation instructions fall into three classes: arithmetic, logical, and shift. Floating-point formats and operations handle broader ranges of operand values for arithmetic operations.

Program control instructions include basic unconditional and conditional transfers of control, the latter of which may or may not use condition codes. Procedure calls and returns permit programs to be broken up into procedures that perform useful tasks. Interruption of the normal sequence of program execution is based on three types of interrupts: external, internal, and software. Also referred to as exceptions, interrupts require special processing actions upon the initiation of routines to service them and upon returns to execution of the interrupted programs.

REFERENCES

1. MANO, M. M. *Computer Engineering: Hardware Design.* Englewood Cliffs, NJ: Prentice Hall, 1988.
2. GOODMAN, J., AND K. MILLER *A Programmer's View of Computer Architecture.* Fort Worth, TX: Saunders College Publishing, 1993.
3. HENNESSY, J. L., AND D. A. PATTERSON *Computer Architecture: A Quantitative Approach,* 2nd Ed. San Francisco, CA: Morgan Kaufmann, 1996.
4. MANO, M. M. *Computer System Architecture,* 3rd Ed. Englewood Cliffs, NJ: Prentice Hall, 1993.
5. PATTERSON, D. A., AND J. L. HENNESSY *Computer Organization and Design: The Hardware/Software Interface,* 2nd Ed. San Mateo, CA: Morgan Kaufmann, 1998.
6. *IEEE Standard for Microprocessor Assembly Language.* (IEEE Std 694-1985.) New York, NY: The Institute of Electrical and Electronics Engineers.

7. *IEEE Standard for Binary Floating-Point Arithmetic.* (ANSI/IEEE Std 754-1985.) New York, NY: The Institute of Electrical and Electronics Engineers.

PROBLEMS

The plus (+) indicates a more advanced problem and the asterisk (*) indicates a solution is available on the Companion Website for the text.

11–1. Based on operations illustrated in Section 11-1, write a program to evaluate the arithmetic expression

$$X = (A - B) \times (A + C) \times (B - D)$$

Make effective use of the registers to minimize the number of MOV or LD instructions where possible.

(a) Assume a register-to-register architecture with three-address instructions.

(b) Assume a memory-to-memory architecture with two-address instructions.

(c) Assume a single-accumulator computer with one-address instructions.

11–2. *Repeat Problem 11-1 for

$$Y = (A + B) \times C \div (D - E \times F)$$

All operands are initially in memory and DIV represents divide.

11–3. *A program is to be written for a stack architecture for the arithmetic expression

$$X = (A - B) \times (A + C) \times (B - D)$$

(a) Find the corresponding RPN expression.

(b) Write the program using PUSH, POP, ADD, MUL, SUB, and DIV instructions.

(c) Show the contents of the stack after the execution of each of the instructions.

11–4. Repeat Problem 11-3 for the arithmetic expression

$$(A + B) \times C \div (D - (E \times F))$$

11–5. A two-word instruction is stored in memory at an address designated by the symbol W. The address field of the instruction (stored at $W + 1$) is designated by the symbol Y. The operand used during the execution of the instruction is stored at an address symbolized by Z. An index register contains the value X. State how Z is calculated from the other addresses if the addressing mode of the instruction is **(a)** direct; **(b)** indirect; **(c)** relative; **(d)** indexed.

11–6. *A two-word relative mode branch-type instruction is stored in memory at location 207 and 208 (decimal). The branch is made to an address equivalent to decimal 195. Let the address field of the instruction (stored at address 208) be designated by X.

(a) Determine the value of X in decimal.

(b) Determine the value of X in binary, using 16 bits. (Note that the number is negative and must be in 2's complement notation. Why?)

11–7. Repeat Problem 11-6 for a branch instruction in locations 143 and 144 and a branch address equivalent to 1000. All values are in decimal.

11–8. How many times does the control unit refer to memory when it fetches and executes a two-word indirect addressing-mode instruction if the instruction is **(a)** a computational type requiring one operand from a memory location with the return of the result to the same memory location; **(b)** a branch type?

11–9. An instruction is stored at location 300 with its address field at location 301. The address field has the value 211. A processor register $R1$ contains the number 189. Evaluate the effective address if the addressing mode of the instruction is **(a)** direct; **(b)** immediate; **(c)** relative; **(d)** register indirect; **(e)** indexed with $R1$ as the index register.

11–10. *A computer has a 32-bit word length, and all instructions are one word in length. The register file of the computer has 16 registers.

(a) For a format with no mode fields and three register addresses, what is the maximum number of opcodes possible?

(b) For a format with two register address fields, one memory field, and a maximum of 100 opcodes, what is the maximum number of memory address bits available?

11–11. A computer with a register file, but without PUSH and POP instructions, is to be used to implement a stack. The computer does have the following register indirect modes:

Register indirect + increment:

LD R2 R1	$R2 \leftarrow M[R1]$
	$R1 \leftarrow R1 + 1$
ST R2 R1	$M[R1] \leftarrow R2$
	$R1 \leftarrow R1 + 1$

Decrement + register indirect:

LD R2 R1	$R1 \leftarrow R1 - 1$
	$R2 \leftarrow M[R1]$
ST R2 R1	$R1 \leftarrow R1 - 1$
	$M[R1] \leftarrow R2$

Show how these instructions can be used to provide the equivalent of PUSH and POP by using the instructions and register *R*6 as the stack pointer.

11–12. A complex instruction, push registers (PSHR), pushes the contents of all of the registers onto the stack. There are eight registers, *R*0 through *R*7, in the CPU. A corresponding instruction, POPR, pops the saved contents of the registers back from the stack into the registers.

(a) Write a register transfer description for the execution of PSHR.

(b) Write a register transfer description for the execution of POPR.

11–13. A computer with an independent I/O system has the input and output instructions

IN R[DR] ADRS

OUT ADRS R[SB]

where ADRS is the address of an I/O register port. Give the equivalent instructions for a computer with memory-mapped I/O.

11–14. *Assume a computer with 8-bit words for the multiple-precision addition of two 32-bit unsigned numbers,

1F C6 24 7B + 00 57 ED 4B

(a) Write a program to execute the addition, using add and add with carry instructions.

(b) Execute the program for the given operands. Each byte is expressed as a 2-digit hexadecimal number.

11–15. Perform the logic AND, OR, and XOR with the two bytes 00110101 and 10111001.

11–16. Given the 16-bit value 1010 1001 0111 1100, what operation must be performed, and what operand is needed, in order to

(a) set the least significant 8 bits to 1's?

(b) complement the bits in odd positions (The leftmost bit is 15 and the rightmost bit is 0)?

(c) clear the bits in odd positions to 0's?

11–17. *An 8-bit register contains the value 01101001, and the carry bit is equal to 1. Perform the eight shift operations given by the instructions listed in Table 11-5 as a sequence of operations on this register.

11–18. Show how the following two floating-point numbers are to be added to get a normalized result:

$$(-.12345 \times 10^{+3}) + (+.71234 \times 10^{-1})$$

11–19. *A 36-bit floating-point number consists of 26 bits plus sign for the fraction and 8 bits plus sign for the exponent. What are the largest and smallest positive nonzero quantities for normalized numbers?

11–20. *A 4-bit exponent uses an excess 7 number for the bias. List all biased binary exponents from +8 through −7.

11–21. The IEEE standard double-precision floating-point operand format consists of 64 bits. The sign occupies 1 bit, the exponent has 11 bits, and the fraction occupies 52 bits. The exponent bias is 1023 and the base is 2. There is an implied bit to the left of the binary point in the fraction. Infinity is represented with a biased exponent equal to 2047 and a fraction of 0.

(a) Give the formula for finding the decimal value of a normalized number.

(b) List a few biased exponents in binary, as is done in Table 11-6.

(c) Calculate the largest and smallest positive normalized numbers that can be accommodated.

11–22. Prove that if the equality $2^x = 10^y$ holds, then $y = 0.3x$. Using this relationship, calculate the largest and smallest normalized floating-point numbers in decimal that can be accommodated in the single-precision IEEE format.

11–23. *It is necessary to branch to ADRS if the bit in the least significant position of the operand in a 16-bit register is equal to 1. Show how this can be done with the TEST (Table 11-7) and BNZ (Table 11-8) instructions.

11–24. Consider the two 8-bit numbers $A = 00101101$ and $B = 01101001$.

(a) Give the decimal equivalent of each number, assuming that (1) they are unsigned and (2) they are signed 2's complement.

(b) Add the two binary numbers and interpret the sum, assuming that the numbers are (1) unsigned and (2) signed two's complement.

(c) Determine the values of the C (carry), Z (zero), N (sign), and V (overflow) status bits after the addition.

(d) List the conditional branch instructions from Table 11-8 that will have a true condition.

11–25. *The program in a computer compares two unsigned numbers A and B by performing a subtraction $A - B$ and updating the status bits.
Let $A = 01011101$ and $B = 01011100$.

(a) Evaluate the difference and interpret the binary result.

(b) Determine the values of status bits C (borrow) and Z (zero).

(c) List the conditional branch instructions from Table 11-9 that will have a true condition.

11–26. The program in a computer compares two signed 2's complement numbers A and B by performing subtraction $A - B$ and updating the status bits.
Let $A = 11011110$ and $B = 11010110$.

(a) Evaluate the difference and interpret the binary result.

(b) Determine the value of status bits N (sign), Z (zero), and V (overflow).

(c) List the conditional branch instructions from Table 11-10 that will have a true condition.

11–27. *The top of a memory stack contains 3000. The stack pointer *SP* contains 2000. A two-word procedure call instruction is located in memory at address 2000, followed by the address field of 0301 at location 2001. What are the contents of *PC*, *SP*, and the top of the stack

(a) before the call instruction is fetched from memory?

(b) after the call instruction is executed?

(c) after the return from the procedure?

11–28. A computer has no stack, but instead uses register *R*7 as a link register (i.e., the computer stores the return address in *R*7).

(a) Show the register transfers for a branch and link instruction.

(b) Assuming that another branch and link is present in the procedure called, what action must be taken by software before the branch and link occurs?

11–29. What are the basic differences between a branch, a procedure call, and a program interrupt?

11–30. *Give five examples of external interrupts and five examples of internal interrupts. What is the difference between a software interrupt and a procedure call?

11–31. A computer responds to an interrupt request signal by pushing onto the stack the contents of the *PC* and the current *PSR*. The computer then reads new *PSR* contents from memory from the location given by the interrupt vector address (*IVAD*). The first address of the service program is taken from memory at location *IVAD* + 1.

(a) List the sequence of microoperations implementing the interrupt.

(b) List the sequence of microoperations implementing the return from interrupt.

CHAPTER 12

RISC AND CISC CENTRAL PROCESSING UNITS

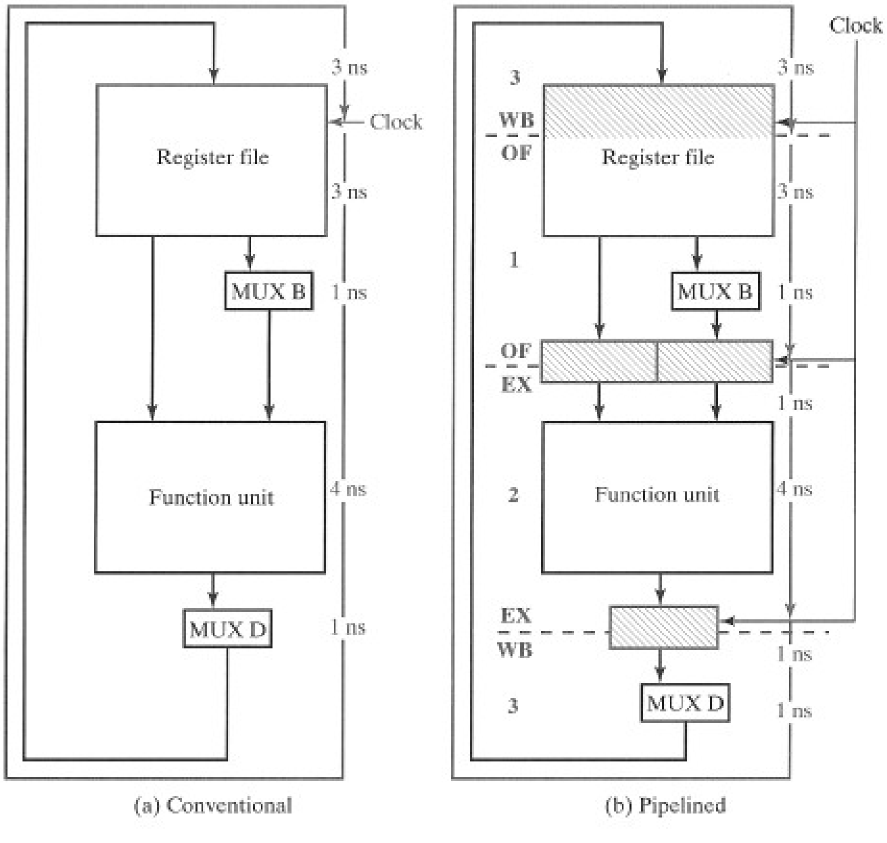
The central processing Unit (CPU) is the key component of a digital computer. Its purpose is to decode instructions received from memory and perform transfer, arithmetic, logic, and control operations with data stored in internal registers, memory, or I/O interface units. Externally, the CPU provides one or more buses for transferring instructions, data, and control information to and from components connected to it.

In the generic computer at the beginning of Chapter 1, the CPU is a part of the processor and is heavily shaded. CPUs, however, may also appear elsewhere in computers. Small, relatively simple computers called microcontrollers are used in computers and in other digital systems to perform limited or specialized tasks. For example, a microcontroller is present in the keyboard and in the monitor in the generic computer; thus, these components are also shaded. In such microcontrollers, the CPU may be quite different from those discussed in this chapter. The word lengths may be short (e. g., eight bits), the number of registers small, and the instruction sets limited. Performance, relatively speaking, is low, but adequate. Most important, the cost of these microcontrollers is very low, making their use cost effective.

The approach in this chapter builds upon and parallels that in Chapter 10. It begins by converting the datapath in Chapter 10 to a pipelined datapath. A pipelined control unit is added to form a reduced instruction set computer (RISC) that is analogous to the single-cycle computer. Problems that arise due to the use of pipelining are introduced and solutions are offered in the context of the RISC design. Next, the control unit is augmented to provide a complex instruction set computer (CISC) that is analogous to the multiple-cycle computer. A brief overview of some of the methods used to enhance pipelined processor performance is presented. Finally, we relate the design ideas discussed to general digital system design.

12-1 Pipelined Datapath

Figure 10-17 was used to illustrate the long delay path present in the single-cycle computer and the resultant clock frequency limit. With a narrower focus, Figure 12-1(a) illustrates maximum delay values for each of the components of a typical datapath. A maximum of 4 ns (3 ns + 1 ns) is required to read two operands from the register file or to read one operand from the register file and obtain a constant from MUX *B*. A maximum of 4 ns is also required to execute an operation in the functional unit. Finally, a maximum of 4 ns (1 ns + 3 ns) is required to write the result back into the register file, including the delay of MUX *D*. Adding these delays, we find that 12 ns are required to perform a single microoperation. The maximum rate at which the microoperations can be performed is the inverse of 12 ns (i.e., 83.3 MHz). This is the maximum frequency at which the clock can be operated, since 12 ns is the smallest clock period that will allow each microoperation to be completed with certainty. As illustrated in Figure 10-17, delay paths that



(a) Conventional

(b) Pipelined

□ **FIGURE 12-1**
Datapath Timing

pass through both the datapath and the control unit limit the clock frequency to an even smaller value. For the datapath alone and for the combination of the datapath and control unit in the single-cycle computer, the execution of a microoperation constitutes the execution of an instruction. Thus, the rate of execution of instructions equals the clock frequency.

Now suppose that the datapath execution rate is not adequate for a particular application, and that there are no faster components available with which to reduce the 12 ns required to complete a microoperation. Still, it may be possible to reduce the clock period and increase the clock frequency. This can be done by breaking up the 12-ns delay path with registers. The resulting datapath, sketched in Figure 12-1(b), is referred to as a *pipelined datapath*, or just a *pipeline.*

Three sets of registers break the delay of the original datapath into three parts. These registers are shown crosshatched in blue. The register file contains the first set of registers. Cross-hatching covers only the top half of the register file, since the lower half is viewed as the combinational logic that selects the two registers to be read. The two registers that store the *A* data from the register file and the output of MUX *B* constitute the second set of registers. The third set of registers stores the inputs to MUX *D*.

The term "pipeline," unfortunately, does not provide the best analogy for the corresponding datapath structure. A better analogy for the datapath pipeline is a production line. A common illustration of a production line is an automated car wash in which cars are pulled through a series of stations at which a particular step of car washing is performed:

1. Wash - Flush with hot, soapy water,
2. Rinse - Flush with plain warm water, and
3. Dry - Blow air over the surface.

In this example, the processing of a vehicle through the car wash consists of three steps and requires a certain amount of time to complete. Using this analogy, the processing of an instruction by a pipeline consists of $n > 2$ steps and requires a certain amount of time to complete. The length of time required to process an instruction is called the *latency time*. Using the car wash analogy, the latency time is the length of time it takes for a car to pass through the three stations performing the three steps of the process. This time remains the same regardless of whether there is a single car or there are three cars in the car wash at a given time.

Continuing this analogy, with the pipeline datapath corresponding to the car wash, what corresponds to the nonpipelined datapath? A car wash with all of the steps available at a single station, with the steps performed serially. We now can compare the analogies, thereby comparing the pipelined and nonpipelined datapath. For the multiple station car wash and the single station car wash, the latencies are approximately the same. So by going to the multiple station car wash, there is no decrease in the time required to wash a car. However, suppose that we consider the frequency at which a washed car emerges from the two types of car washes. For the single station car wash, this frequency is the inverse of the latency time. In contrast, for the multiple station car wash with three stages, a washed car emerges at a frequency of three times the inverse of the latency time. Thus, there is a factor of three improvement in the frequency or rate of delivery of washed cars. Based on the analogy to pipelined datapaths with n stages and nonpipelined datapaths, the former has a processing rate or *throughput* for instructions that is n times that of the latter.

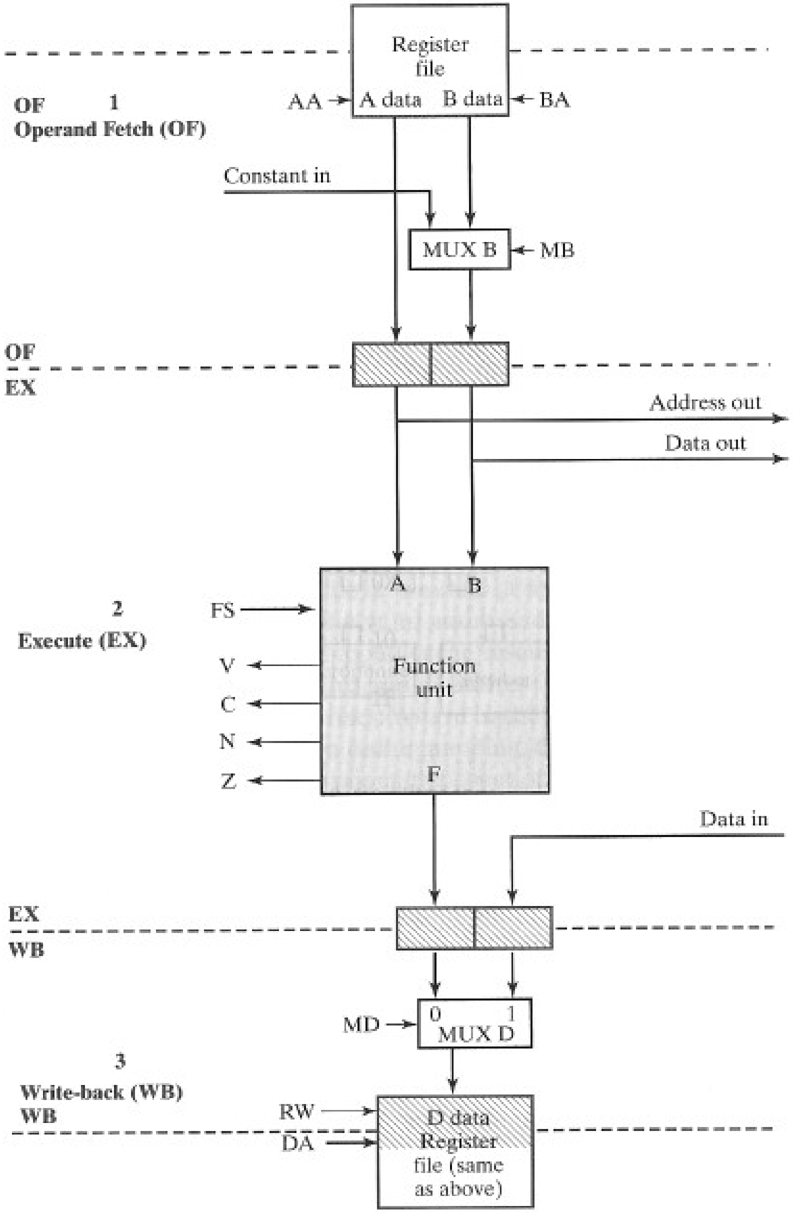
The desired structure, based on the nonpipelined, conventional datapath described in Chapter 10, is sketched in Figure 12-1(b). The operand fetch (OF) is stage 1, the execution (EX) is stage 2, and the write-back (WB) is stage 3. These stages are labeled at their boundaries with appropriate abbreviations. At this point, the analogy breaks down somewhat because the cars move smoothly through the car wash while the data within the pipeline moves synchronously with a clock controlling the movement from stage to stage. This has some interesting implications. First of all, the movement of the data through the pipeline is in discrete steps rather than continuous. Second, the length of time in each of the stages must be the clock period and is the same for all stages. To provide the mechanism separating the stages in the pipeline, registers are placed between the stages of the pipeline. These registers provide temporary storage for data passing through the pipeline and are called *pipeline platforms*.

Returning to the pipelined datapath example in Figure 12-1(b), Stage 1 of the pipeline has the delay required for reading the register file followed by selection by MUX *B*. This delay is 3 plus 1 ns, or 4 ns. Stage 2 of the pipeline has the 1 ns delay of the platform plus the 4 ns delay of the functional unit, giving 5 ns. Stage 3 has the 1 ns delay of the platform, the delay for the selection by MUX *D*, and the delay for writing back into the register file. This delay is $1 + 1 + 3$, for a total of 5 ns. Thus, all flip-flop–to–flip-flop delays are at most 5 ns, allowing a minimum clock period of 5 ns (assuming that the setup times for the flip-flops are zero) and a maximum clock frequency of 200 MHz, compared with the 83.3 MHz for the single state datapath. This clock frequency corresponds to the maximum throughput of the pipeline which is 200 million instructions per second, about 2.4 times that of the nonpipelined datapath. Even though there are three stages, the improvement factor is not three. This is due to two factors: (1) the delay contributed by the pipeline platforms and (2) the differences between the delay of the logic assigned to each stage. The clock period is governed by the longest delay, rather than the average delay assigned to any stage.

A more detailed diagram of the pipelined datapath appears in Figure 12-2. In this diagram, rather than showing the path from the output of MUX *D* to the register file input, the register file is shown *twice*—once in the OF stage, where it is read, and once in the WB stage, where it is written.

The first stage, OF, is the operand fetch stage. The operand fetch consists of reading register values to be used from the register file and, for Bus B, selecting between a register value or a constant by using MUX *B*. Following the OF stage is the first pipeline platform. The pipeline registers store the operand or operands for use in the next stage during the next clock cycle.

The second stage of the pipeline is the execute stage, denoted EX. In this stage, a function unit operation occurs for most microoperations. The results produced from this stage are captured by the second pipeline platform.



□ **FIGURE 12-2**
Block Diagram of Pipelined Datapath

The third and final stage of the pipeline is the write-back stage, denoted WB. In this stage, the result saved from the EX stage, or the value on Data in, is selected by MUX *D* and written back into the register file at the end of the stage. In this case, the write part of the register file is the pipeline platform. The WB stage completes the execution of each microoperation that requires writing to a register.

Before leaving the car wash analogy, we examine the cost of the single-stage car wash versus that of the three-stage car wash. First, even though the three-stage car wash washes vehicles three times as fast as the single-stage car wash does, it costs three times as much in terms of space. Plus, it has the overhead of the mechanism to move the cars along through the stages. So it appears that it is not very cost effective compared with having three single-stage assembly stations operating in parallel. Nevertheless, from a business standpoint, it has proven to be cost effective. In terms of the car wash, can you figure out why? In contrast, for the pipelined datapath, pipeline platforms cut a single datapath into three pieces. Thus, a first order estimate of the cost increase is mainly that of the pipeline platforms.

Execution of Pipeline Microoperations

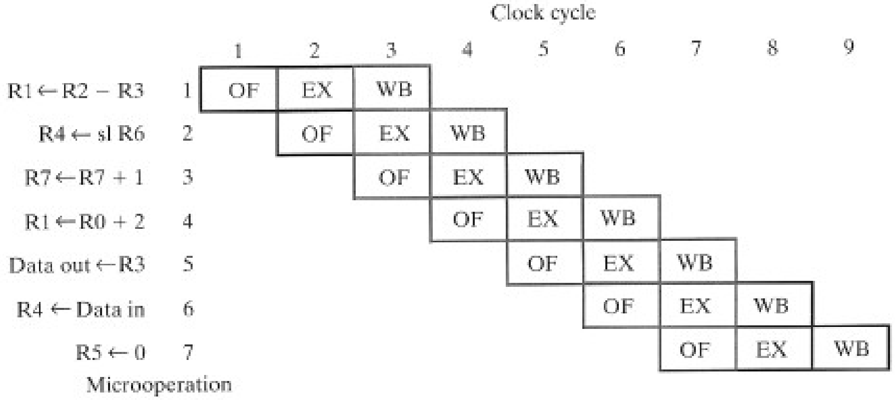
There are up to three operations at some stage of completion in the car wash at any given time. By analogy, we should be able to have three microoperations at some stage of completion in the pipelined datapath at any given time.

We now examine the execution of this sequence of microoperations with respect to the stages of the pipeline in Figure 12-2. In clock period 1, microoperation 1 is in the OF stage. In clock period 2, microoperation 1 is in the EX stage, and microoperation 2 is in the OF stage. In clock period 3, microoperation 1 is in the WB stage, microoperation 2 is in the EX stage, and microoperation 3 is in the OF stage. So at the end of the third clock period, microoperation 1 has completed execution, microoperation 2 is two-thirds finished, and microoperation 3 is one-third finished. So we have completed 1 + 2/3+ 1/3 = 2.0 microoperations in three clock periods, or 15 ns. In the conventional datapath, we would have completed execution of microoperation 1 only. So, indeed, the pipelined datapath performance is superior in this example.

The procedure we have been using to analyze the sequence of microoperations so far is somewhat tedious. So to finish the analysis of the timing of the sequence, we will use a *pipeline execution pattern* diagram, as shown in Figure 12-3. Each vertical position in this diagram represents a microoperation to be performed, and each horizontal position represents a clock cycle. An entry in the diagram represents the stage of processing of the microoperation. So, for example, the execution (EX) stage of microoperation 4, which adds the constant 2 to *R*0, occurs in clock cycle 5.

We can see from the overall diagram that the sequence of seven microoperations requires nine clock cycles to execute completely. The time required for execution is 9 × 5 = 45 ns, compared to 7 × 12 = 84 ns for the conventional datapath. Thus, the sequence of microoperations is executed about 1.9 times faster.

Now let us examine the pipeline execution pattern carefully. In the first two clock cycles, not all of the pipeline stages are active, since the pipeline is *filling*. In the next five clock cycles, all stages of the pipeline are active, as indicated in blue, and the pipeline is fully utilized. In the last two clock cycles, not all stages of the



Microoperation		1	2	3	4	5	6	7	8	9
R1 ← R2 − R3	1	OF	EX	WB						
R4 ← sl R6	2		OF	EX	WB					
R7 ← R7 + 1	3			OF	EX	WB				
R1 ← R0 + 2	4				OF	EX	WB			
Data out ← R3	5					OF	EX	WB		
R4 ← Data in	6						OF	EX	WB	
R5 ← 0	7							OF	EX	WB

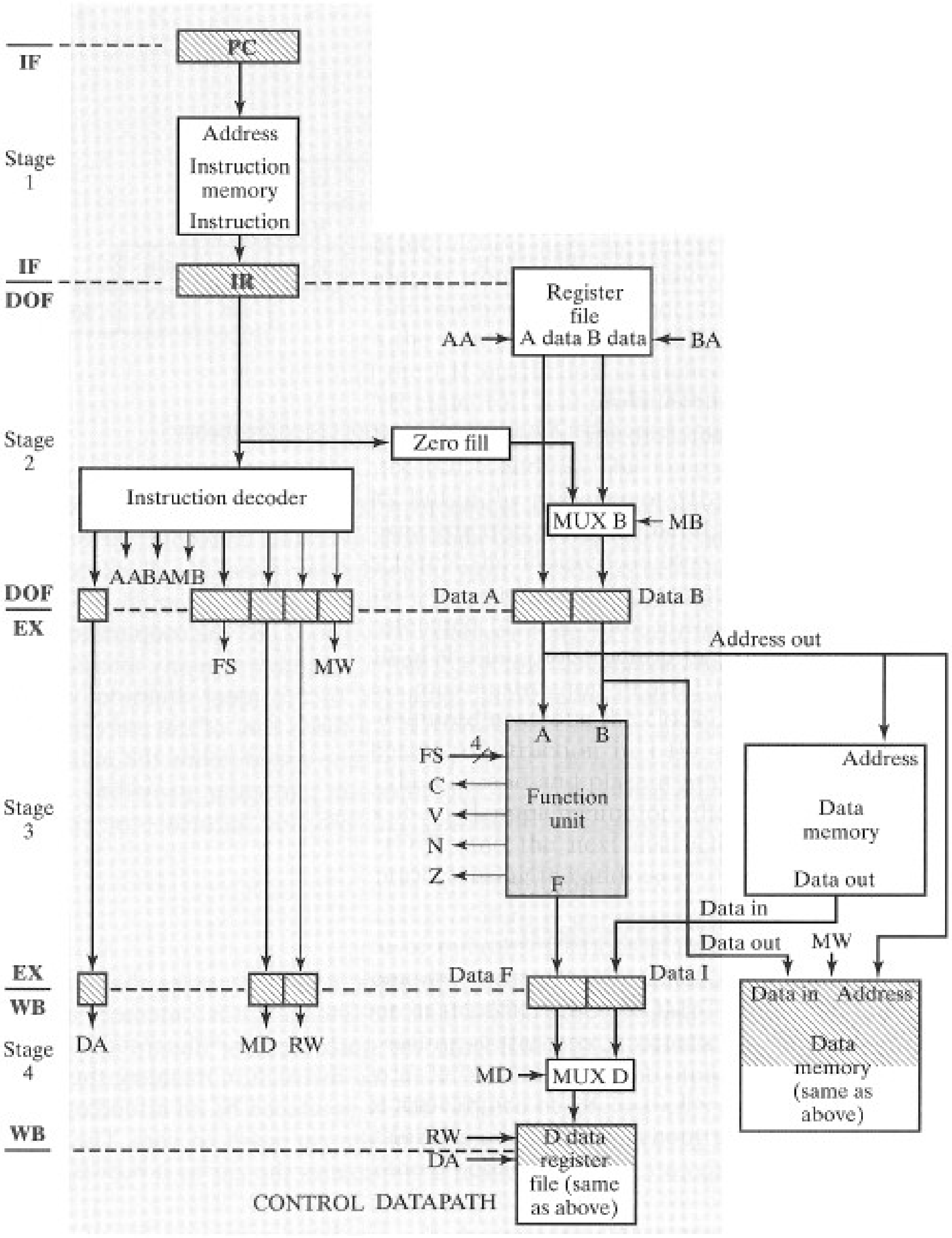
□ **FIGURE 12-3**
Pipeline Execution Pattern for Microoperation Sequence

pipeline are active, since the pipeline is *emptying*. If we want to find the maximum possible improvement of the pipelined datapath over the conventional one, we compare the two when the pipeline is fully utilized. Over these five clock cycles, 3 through 7, the pipeline executes $(5 \times 3) \div 3 = 5$ microoperations in 25 ns. In the same time, the conventional datapath executes $25 \div 12 = 2.083$ microoperations. So the pipelined datapath executes at best $5 \div 2.083 = 2.4$ times as many microoperations in a given time as the conventional datapath. In this ideal situation, we say that the throughput of the pipelined datapath is 2.4 times that of the conventional one. Note that filling and emptying reduce the pipeline speed below the maximum of 2.4. Additional topics associated with pipelines—in particular, providing a control unit for a pipelined datapath and dealing with pipeline hazards—are covered in the next two sections.

12-2 Pipelined Control

In this section, a control unit is specified to produce a CPU by using the datapath from the last section. Since the instruction must be fetched from a memory as well as executed, we add a stage to the analogous car wash used for illustration in that section. Analogous to the instruction fetch from the instruction memory, the operations in the car wash are specified by order sheets, produced by an attendant, that permit the functions performed in the stages of the car wash to vary. The order sheet, which is analogous to an instruction, accompanies the car as it moves down the line.

Figure 12-4 shows the block diagram of a pipelined computer based on the single-cycle computer. The datapath is that of Figure 12-2. The control has an added stage for instruction fetch that includes the *PC* and instruction memory. This becomes stage 1 of the combined pipeline. The instruction decoder and register file read are now in stage 2, the function unit and data memory read and write are in stage 3, and the register file write is in stage 4. These stages are labeled at their



□ **FIGURE 12-4**
Block Diagram of Pipelined Computer

boundaries with appropriate abbreviations. In the figure, we have added registers to the pipeline platforms between stages, as necessary to pass the decoded instruction information through the pipeline along with the data being processed. These additional registers serve to pass along the instruction information, just as order information was passed along in the car wash.

The added first stage is the instruction fetch stage, denoted by IF, which lies wholly in the control. In this stage, the instruction is fetched from the instruction memory, and the value in the *PC* is updated. Due to additional complexities of handling jumps and branches in a pipelined design, *PC* update is restricted here to an increment, with a more complete treatment provided in the next section. Between the first stage and the second stage is an interstage pipeline platform that plays the role of instruction register, so it has been labeled *IR*.

In the second stage, DOF for decode and operand fetch, decoding of the *IR* into control signals takes place. Among the decoded signals, the register file addresses AA and BA and the multiplexer control signal MB are used in this stage for operand fetch. All other decoded control signals are passed on to the next pipeline platform, to be used later. Following the DOF stage is the second pipeline platform, whose registers store control signals to be used later. The third stage of the pipeline is the execution stage, denoted EX. In this stage, an ALU operation, a shift operation, or a memory operation is executed for most instructions. Thus, the control signals used in this stage are FS and MW. The read part of the data memory *M* is considered a part of the stage. For a memory read, the value of the word addressed is read to Data out from the data memory. All of the results produced from this stage, plus the control signals for the last stage, are captured by the third pipeline platform. The write part of data memory *M* is considered a part of this platform, so a memory write may occur here. The control information held in the final pipeline platform consists of DA, MD, and RW, which are used in the final write-back stage, WB.

The location of the pipeline platforms has balanced the partitioning of the delays, so that the delays per stage are no more that 5 ns. This gives a potential maximum clock frequency of 200 MHz, about 3.4 times that of the single-cycle computer. Note, however, that an instruction takes 4 × 5 = 20 ns to execute. This latency of 20 ns compares to that of 17 ns for the single-cycle computer. So if only one instruction at a time is being executed, even fewer instructions are executed per second than for the single-cycle computer.

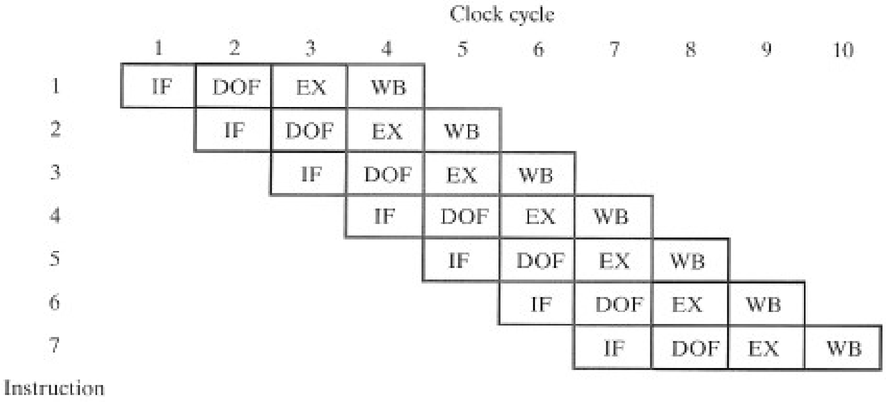
Pipeline Programming and Performance

If our hypothetical car wash is extended to four stages, there are up to four operations at some stage of completion at any given time. By analogy, then, we should be able to have four instructions at some stage of completion in the pipeline of our computer at any given time. Suppose we consider a simple calculation: Load the constants 1 through 7 into the seven registers *R*1 through *R*7, respectively. The program to do this is as follows (the number on the left is a number to identify the instruction):

```
1   LDI R1, 1
2   LDI R2, 2
3   LDI R3, 3
4   LDI R4, 4
5   LDI R5, 5
6   LDI R6, 6
7   LDI R7, 7
```

Let us examine the execution of this program with respect to the stages of the pipeline in Figure 12-4. We employ the pipeline execution pattern diagram shown in Figure 12-5. In clock period 1, instruction 1 is in the IF stage of the pipeline. In clock period 2, instruction 1 is in the DOF stage and instruction 2 is the IF stage. In clock period 3, instruction 1 is in the EX stage, instruction 2 is in the DOF stage, and instruction 3 is in the IF stage. In clock period 4, instruction 1 is in the WB stage, instruction 2 is in the EX stage, instruction 3 is in the DOF stage, and instruction 4 is in the IF stage. So at the end of the fourth clock period, instruction 1 has completed execution, instruction 2 is three-fourths finished, instruction 3 is half finished, and instruction 4 is one-fourth finished. So we have completed 1 + 3/4 + 1/2 + 1/4 = 2.5 instructions in four clock periods, or 20 ns. We can see from the overall diagram that the complete program of seven instructions requires 10 clock cycles to execute. Thus, the time required is 50 ns, compared to 119 ns for the single-cycle computer, and the program is executed about 2.4 times faster.

Now suppose that we examine the pipeline execution pattern carefully. In the first three clock cycles, not all of the pipeline stages are active, since the pipeline is *filling*. In the next four clock cycles, all stages of the pipeline are active, as indicated in blue, and the pipeline is fully utilized. In the last three clock cycles, not all stages of the pipeline are active, since the pipeline is *emptying*. If we want to find the



Instruction \ Clock cycle	1	2	3	4	5	6	7	8	9	10
1	IF	DOF	EX	WB						
2		IF	DOF	EX	WB					
3			IF	DOF	EX	WB				
4				IF	DOF	EX	WB			
5					IF	DOF	EX	WB		
6						IF	DOF	EX	WB	
7							IF	DOF	EX	WB

□ **FIGURE 12-5**
Pipeline Execution Pattern of Register Number Program

maximum possible improvement of the pipelined computer over the single-cycle computer, we compare the two in the situation in which the pipeline is fully utilized. Over these four clock cycles, or 20 ns, the pipeline executes 4 × 4 ÷ 4 = 4.0 instructions. In the same time, the single-cycle computer executes 20 ÷ 17 = 1.18 instructions. So in the best case, the pipelined computer executes 4 ÷ 1.18 = 3.4 times as many instructions in a given time as the single-cycle computer does. In this ideal situation, we say that the throughput of the pipelined computer is 3.4 times that of the single-cycle computer. Note that even though the pipeline has four stages, the pipelined computer is not four times as fast as the single-cycle computer, because the delays of the latter cannot be divided exactly into four equal pieces and the delays of the added pipeline platforms. Also, filling and emptying the pipeline reduces its speed enough that the speed of the pipelined computer is less than the ideal maximum speed of 3.4 times as fast as the single-cycle computer.

The study of the pipelined computer here, along with the single-cycle computer and multiple-cycle computer in Chapter 10, completes our examination of three computer control organizations. Both the pipelined datapaths and the controls we have studied here are simplified and missing elements. Next we present two CPU designs that illustrate combinations of architectural characteristics of the instruction set, the datapath, and the control unit. The designs are top down, but reuse prior component designs, illustrating the influence of the instruction set architecture on the datapath and control units, and the influence of the datapath on the control unit. The material makes extensive use of tables and diagrams. Although we reuse and modify component designs from Chapter 10, background information from these chapters is not repeated here. Pointers, however, are given to earlier sections of the book, where detailed information can be found.

The two CPUs presented are for a RISC using a pipelined datapath with a hardwired pipelined control unit and a CISC based on the RISC using an auxiliary microprogrammed control unit. These two designs represent two distinct instruction set architectures with architectures using a common pipelined core that contributes enhanced performance.

12-3 The Reduced Instruction Set Computer

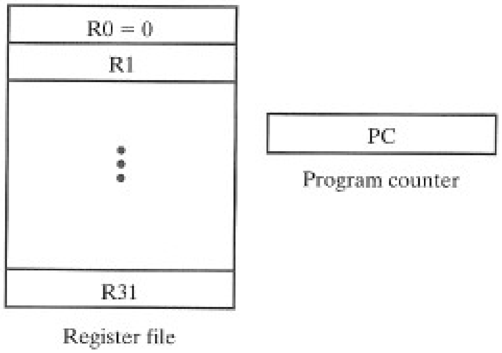
The first design we examine is for a reduced instruction set computer with a pipelined datapath and control unit. We begin by describing the RISC instruction set architecture, which is characterized by load/store memory access, four addressing modes, a single instruction format length, and instructions that require only elementary operations. The operations, resembling those that can be performed by the single-cycle computer, can be performed by a single pass through the pipeline. The datapath for implementing the ISA is based on the single-cycle datapath initially described in Figure 10-11 and converted to a pipeline in Figure 12-2. In order to implement the RISC instruction set architecture, modifications are made to the register file and the function unit. These modifications represent the effects of a longer instruction word length and the desire to include multiple position shifts among the elementary operations. The control unit is based on the pipelined control unit in Figure 12-4. Modifications include support for the 32-bit instruction word and a more extensive program counter structure for dealing with branches in the pipeline environment. In response to data and control hazards associated with pipelined designs, additional changes will be made to both the control and datapath to sustain the performance gain achieved by using a pipeline.

Instruction Set Architecture

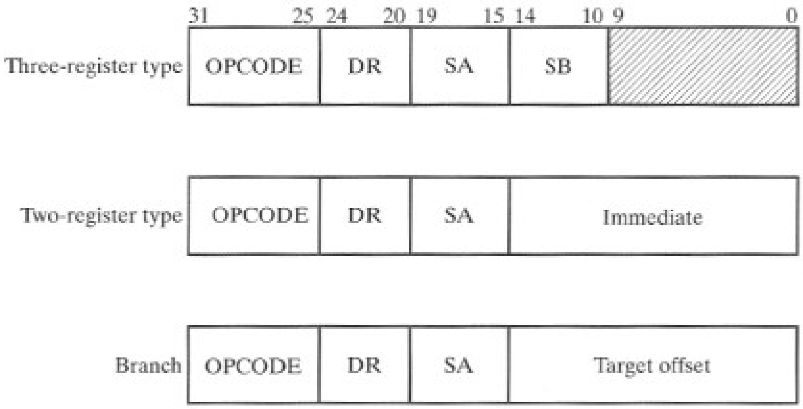
Figure 12-6 shows the CPU registers accessible to the programmer in this RISC. All registers are 32 bits. The register file has 32 registers, *R*0 through *R*31. *R*0 is a special register that supplies the value zero when used as a source and discards the result when used as a destination. The size of the programmer-accessible register file is comparatively large in the RISC because of the load/store instruction set architecture. Since the data manipulation operations can use only register operands, many active operands need to be present in the register file. Otherwise, numerous stores and loads would be needed to temporarily save operands in the data memory between data manipulation operations. In addition, in many real pipelines, these stores and loads require more than one clock cycle for their execution. To prevent these factors from degrading RISC performance, a larger register file is required.

In addition to the register file, only a program counter, *PC*, is provided. If stack pointer-based or processor status register-based operations are required, they are simply implemented by sequences of instructions using registers.

Figure 12-7 gives the three instruction formats for the RISC CPU. The formats use a single word of 32 bits. This longer word length is needed to hold realistic address values, since additional instruction words for holding addresses are difficult to accommodate in the RISC CPU. The first format specifies three registers. The two registers addressed by the 5-bit source register fields SA and SB contain the two operands. The third register, addressed by a 5-bit destination register field DR, specifies the register location for the result. A 7-bit OPCODE provides for a maximum of 128 operations.



□ **FIGURE 12-6**
CPU Register Set Diagram for RISC



□ **FIGURE 12-7**
RISC CPU Instruction Formats

The remaining two formats replace the second register with a 15-bit constant. In the two-register format, the constant acts as an immediate operand and, in the branch format, the constant is a *target offset*. The *target address* is another name for the effective address, particularly if the address is used in a branch instruction. The target address is formed by adding the target offset to the contents of the *PC*. Thus, branching uses relative addressing based on the updated value of the *PC*. In order to branch backward from the current *PC* location, the offset, regarded as a 2's complement number with sign extension is added to the *PC*. The branch instructions specify source register SA. Whether the branch or jump is taken is based on whether the source register contains zero. The DR field is used to specify the register in which to store the return address for the procedure call. Finally, the rightmost 5 bits of the 15-bit constant are also used as the shift amount SH for multiple bit shifts.

Table 12-1 contains the 27 operations to be performed by the instructions. A mnemonic, an opcode, and a register transfer description are given for each operation. All of the operations are elementary and can be described by a single register transfer statement. The only operations that can access memory are Load and Store. A significant number of immediate instructions help to reduce data memory accesses and speed up execution when constants are employed. Since the immediate field of the instruction is only 15 bits, the leftmost 17 bits must be filled to form a 32-bit operand. In addition to using zero fill for logical operations, a second method used is called *sign extension*. The most significant bit of the immediate operand, bit 14 of the instruction, is viewed as a sign bit. To form a 32-bit 2's-complement operand, this bit is copied into the 17 bits. In Table 12-1, the sign extension of the immediate field is denoted by se *IM*. The same notation, se *IM*, also represents the sign extension of the target offset field discussed previously.

The absence of stored versions of status bits is handled by the use of three instructions: Branch if Zero (BZ), Branch if Nonzero (BNZ), and Set if Less Than (SLT). BZ and BNZ are single instructions that determine whether a register operand is zero or nonzero and branch accordingly. SLT stores a value in register

□ **TABLE 12-1**
RISC Instruction Operations

Operation	Symbolic Notation	Opcode	Action
No Operation	NOP	0000000	None
Move A	MOVA	1000000	$R[DR] \leftarrow R[SA]$
Add	ADD	0000010	$R[DR] \leftarrow R[SA] + R[SB]$
Subtract	SUB	0000101	$R[DR] \leftarrow R[\mathrm{SA}] + \overline{R[SB]} + 1$
AND	AND	0001000	$R[DR] \leftarrow R[SA] \wedge R[SB]$
OR	OR	0001001	$R[DR] \leftarrow R[SA] \vee R[SB]$
Exclusive-OR	XOR	0001010	$R[DR] \leftarrow \mathrm{R[SA]} \oplus \mathrm{R[SB]}$
Complement	NOT	0001011	$R[DR] \leftarrow \overline{R[SA]}$
Add Immediate	ADI	0100010	$R[DR] \leftarrow R[SA] + \mathrm{se}\ IM$
Subtract Immediate	SBI	0100101	$R[DR] \leftarrow R[SA] + \overline{(\mathrm{se}\ IM)} + 1$
AND Immediate	ANI	0101000	$R[DR] \leftarrow R[SA] \wedge (0 \Vert IM)$
OR Immediate	ORI	0101001	$R[DR] \leftarrow R[SA] \vee (0 \Vert IM)$
Exclusive-OR Immediate	XRI	0101010	$R[DR] \leftarrow R[SA] \oplus (0 \Vert IM)$
Add Immediate Unsigned	AIU	1000010	$R[DR] \leftarrow R[SA] + (0 \Vert IM)$
Subtract Immediate Unsigned	SIU	1000101	$R[DR] \leftarrow R[SA] + \overline{(0 \Vert IM)} + 1$
Move B	MOVB	0001100	$R[DR] \leftarrow R[SB]$
Logical Right Shift by SH Bits	LSR	0001101	$R[DR] \leftarrow \mathrm{lsr}\ R[SA]$ by SH
Logical Left Shift by SH Bits	LSL	0001110	$R[DR] \leftarrow \mathrm{lsl}\ R[SA]$ by SH
Load	LD	0010000	$R[DR] \leftarrow M[R[SA]]$
Store	ST	0100000	$M[R[SA]] \leftarrow R[SB]$
Jump Register	JMR	1110000	$PC \leftarrow R[SA]$
Set if Less Than	SLT	1100101	If $R[\mathrm{SA}] < R[SB]$ then $R[DR] = 1$
Branch on Zero	BZ	1100000	If $R[SA] = 0$, then $PC \leftarrow PC + 1 + \mathrm{se}\ IM$
Branch on Nonzero	BNZ	1010000	If $R[SA] \neq 0$, then $PC \leftarrow PC + 1 + \mathrm{se}\ IM$
Jump	JMP	1101000	$PC \leftarrow PC + 1 + \mathrm{se}\ IM$
Jump and Link	JML	0110000	$PC \leftarrow PC + 1 + \mathrm{se}\ IM, R[DR] \leftarrow PC + 1$

$R[DR]$ that acts like a negative status bit. If $R[SA]$ is less than $R[SB]$, a 1 is placed in register $R[DR]$; if $R[SA]$ is greater than or equal to $R[SB]$, a 0 is placed in $R[DR]$. The register $R[DR]$ can then be examined by a subsequent instruction to see whether it is zero (0) or nonzero (1). Thus, using two instructions, the relative values of two operands or the sign of one operand (by letting $R[SB]$ equal $R0$) can be determined.

The Jump and Link (JML) instruction provides a mechanism for implementing procedures. The value in the *PC* after updating is stored in register $R[DR]$, and then the sum of the *PC* and the sign-extended target offset from the instruction is placed in the *PC*. The return from a called procedure can use the Jump Register instruction with SA equal to DR for the calling procedure. If a procedure is to be called from within a called procedure, then each successive procedure that is called will need its own register for storing the return value. A software stack that moves return addresses from $R[DR]$ to memory at the beginning of a called procedure and restores them to $R[SA]$ before the return can also be used.

Addressing Modes

The four addressing modes in the RISC are register, register indirect, immediate, and relative. The mode is specified by the operation code, rather than by a separate mode field. As a consequence, the mode for a given operation is fixed and cannot be varied. The three-operand data manipulation instructions use register mode addressing. Register indirect, however, applies only to the load and store instructions, the only instructions that access data memory. Instructions using the two-register format have an immediate value that replaces register address SB. Relative addressing applies exclusively to branch and jump instructions and so produces addresses only for the instruction memory.

When programmers want to use an addressing mode not provided by the instruction set architecture, such as indexed addressing, they must use a sequence of RISC instructions. For example, for an indexed address for a load operation, the desired transfer is

$$R15 \leftarrow M[R5 + 0 \parallel I]$$

This transfer can be accomplished by executing two instructions:

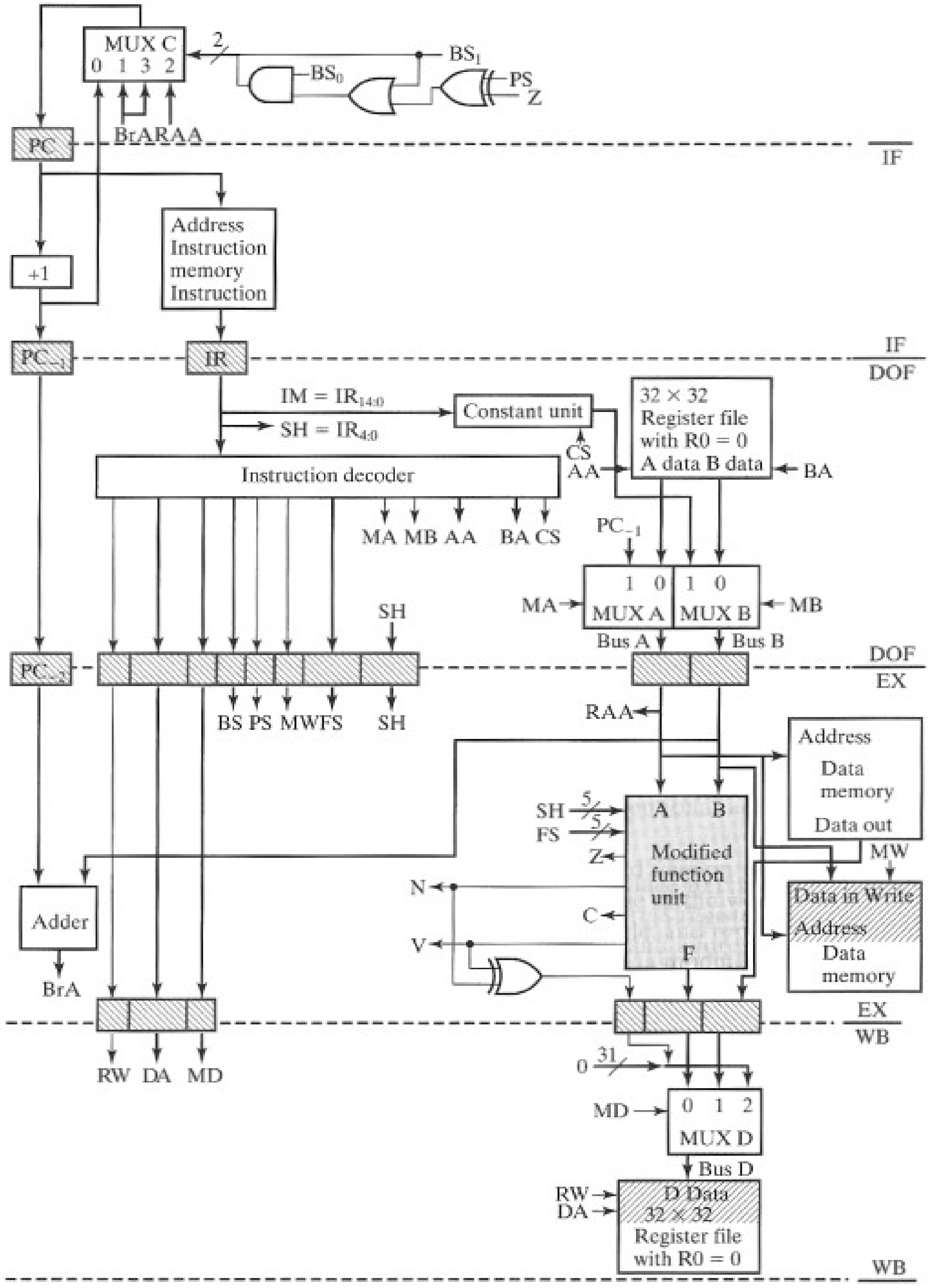
```
AIU R9, R5, I
LD  R15, R9
```

The first instruction, Add Immediate Unsigned, forms the address by appending 17 0's to the left of I and adding the result to $R5$. The resulting effective address is then temporarily stored in $R9$. Next, the Load instruction uses the contents of $R9$ as the address at which to fetch the operand and places the operand in the destination register $R15$. Since, for indexed addressing, I is regarded as a positive offset in memory, the use of unsigned addition is appropriate. Sequences of operations for implementing addressing modes is the primary justification for having unsigned immediate addition available.

Datapath Organization

The pipelined datapath in Figure 12-2 serves as the basis for the datapath here, and we deal only with modifications. These modifications affect the register file, the function unit, and the bus structure. The reader should also refer to the datapath in Figure 12-2 and the new datapath shown in Figure 12-8 in order to understand fully the discussion that follows. We treat each modification in turn, beginning with the register file.

In Figure 12-2, there are 16 16-bit registers, and all registers are identical in function. In the new datapath, there are 32 32-bit registers. Also, reading register

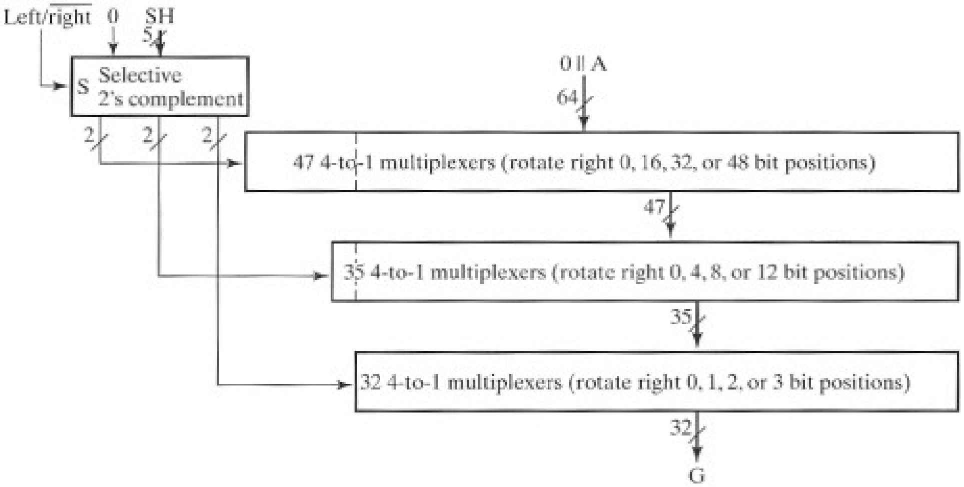


□ **FIGURE 12-8**
Pipelined RISC CPU

$R0$ gives a constant value of zero. If a write is attempted into $R0$, the data will be lost. These changes are implemented in the new register file in Figure 12-8. All data inputs and the data output are 32 bits. To correspond to the 32 registers, the address inputs are five bits. The fixed value of 0 in $R0$ is implemented by replacing the storage elements for $R0$ with open circuits on the lines that were their inputs, and with constant zero values on the lines that were their outputs.

A second major modification to the datapath is the replacement of the single-bit position shifter with a barrel shifter to permit multiple-position shifting. This barrel shifter can perform a logical right or logical left shift of from 0 to 31 positions. A block diagram for the barrel shifter appears in Figure 12-9. The data input is 32-bit operand A, and the output is 32-bit result G. Left/$\overline{\text{right}}$, a control signal decoded from OPCODE, selects a left or right shift. The shift amount field $SH = IR(4{:}0)$ specifies the number of bit positions to shift the data input and takes on values from 0 through 31. A logical shift of p bit positions involves inserting p zeros into the result. In order to provide these zeros and simplify the design of the shifter, we will perform both the left and right shift by using a right rotate. The input to this rotate will be the input data A with 32 zeros concatenated to its left. A right shift is performed by rotating the input p positions to the right; a left shift is performed by rotating $64 - p$ positions to the right. This number of positions can be obtained by taking the 2's complement of the 6-bit value of $0 \parallel SH$.

The 63 different rotates can be obtained by using three levels of 4-to-1 multiplexers, as shown in Figure 12-8. The first level shifts by 0, 16, 32, or 48 positions, the second level by 0, 4, 8, or 12 positions, and the third level by 0, 1, 2, or 3 positions. The number of positions for A to be shifted, 0 through 63, can be implemented by representing 0 || SH as a three-digit base-4 integer. From left to right, the digits have weights $4^2 = 16$, $4^1 = 4$, and $4^0 = 1$. The digit values in each of the positions are 0, 1, 2, and 3. Each digit controls a level of the 4-to-1 multiplexers, the


Left/right
0
SH
5
S
Selective
2's complement
2
2
2
0 || A
64
47 4-to-1 multiplexers (rotate right 0, 16, 32, or 48 bit positions)
47
35 4-to-1 multiplexers (rotate right 0, 4, 8, or 12 bit positions)
35
32 4-to-1 multiplexers (rotate right 0, 1, 2, or 3 bit positions)
32
G

□ **FIGURE 12-9**
32-bit Barrel Shifter

most significant digit controlling the first level, the least significant the third level. Due to the presence of 32 zeros in the 64-bit input, fewer than 64 multiplexers can be used in each level. A level requires the number of multiplexers to be 32 plus the total number of positions its output can be shifted by subsequent levels. The output of the first level can be shifted at most $12 + 3 = 15$ positions to the right. Thus, this level requires $32 + 15 = 47$ multiplexers. The output of the second level can be shifted at most 3 positions, giving $32 + 3 = 35$ multiplexers. The final level cannot be shifted further and so needs just 32 multiplexers.

In the function unit, the ALU is expanded to 32 bits, and the barrel shifter replaces the single position shifter. The resulting modified function unit uses the same function codes as in Chapter 10, except that the two codes for shifts are now labeled as logical shifts, and some codes are not used. The shift amount SH is a new 5-bit input to the modified function unit in Figure 12-8.

The remaining datapath changes are shown in Figure 12-8. Beginning at the top of the datapath, zero fill has been replaced by the constant unit. The constant unit performs zero fill for $CS = 0$ and sign extension for $CS = 1$. MUX A is added to provide a path for the updated PC, PC_{-1}, to the register file for implementation of the Jump and Link (JML) instruction.

One other change in the figure helps implement the Set if Less Than (SLT) instruction. This logic provides a 1 to be loaded into $R[DA]$ if $R[AA] - R[BA] < 0$ and a 0 to be loaded into $R[DA]$ if $R[AA] - R[BA] \geq 0$. It is implemented by adding an additional input to MUX D. The leftmost 31 bits of the input are 0; the rightmost bit is 1 if N is 1 and V is 0 (i.e., if the result of the subtraction is negative and there is no overflow). It is also 1 if N is 0 and V is 1 (i.e., if the result of the subtraction is positive and there is an overflow). These represent all cases in which $R[AA]$ is greater than $R[BA]$ and can be implemented using an exclusive-OR of N and V.

A final difference in the datapath is that the register file is no longer edge triggered and is no longer a part of a pipeline platform at the end of the write-back (WB) stage. Instead, the register file uses latches and is written much earlier than the positive clock edge. Special timing signals are provided that permit the register file to be written in the first half and to be read in the last half of the cycle. In particular, in the second half of the cycle, it is possible to read data written into the register file during the first half of the same clock cycle. This is called a *read-after-write* register file, and it both avoids added complexity in the logic used for handling hazards and reduces the cost of the register file.

Control Organization

The control organization in the RISC is modified from that in Figure 12-4. The modified instruction decoder is essential to deal with the new instruction set. In Figure 12-8, SH is added as an IR field, a 1-bit CS field is added to the instruction decoder, and MD is expanded to two bits. There is a new pipeline platform for SH, and expanded 2-bit platforms for MD.

The remaining control signals are included to handle the new control logic for the PC. This logic permits the loading of addresses into the PC for implementing branches and jumps. MUX C selects from three different sources for the next value of *PC*. The updated *PC* is used to move sequentially through a program. The branch target address *BrA* is formed from the sum of the updated *PC* value for the branch instruction and the sign-extended target offset. The value in *R*[*AA*] is used for the register jump. The selection of these values is controlled by the field BS. The effects of BS are summarized in Table 12-2. If $BS_0 = 0$, then the updated *PC* is selected by $BS_1 = 0$, and *R*[*AA*] is selected by $BS_1 = 1$. If $BS_0 = 1$ and $BS_1 = 1$, then *BrA* is selected unconditionally. If $BS_0 = 1$ and $BS_1 = 0$, then, for $PS = 0$, a branch to *BrA* occurs for $Z = 1$, and for PS = 1, a branch to *BrA* occurs for $Z = 0$. This implements the two conditional branch instructions BZ and BNZ.

In order to have the value of the updated *PC* for the branch and jump instructions when they reach the execution stage, two pipeline registers, PC_{-1} and PC_{-2}, are added. PC_{-2} and the value from the constant unit are inputs to the dedicated adder that forms *BrA* in the execution stage. Note that MUX *C* and the attached control logic are in the EX stage, although shown above the *PC*. The related clock cycle difference causes problems with instructions following branches that we will deal with in later subsections.

The heart of the control unit is the instruction decoder. This is combinational circuitry that converts the operation code in the *IR* into the control signals necessary for the datapath and control unit. In Table 12-3, each instruction is identified by its mnemonic. A register transfer statement and the opcode are given for the instruction. The opcodes are selected such that the least significant four of the seven bits match the bits in the control field FS whenever it is used. This leads to simpler decoding. The register file addresses AA, BA, and DA come directly from SA, SB, and DR, respectively, in the *IR*.

Otherwise, to determine the control codes, the CPU is viewed much as is the single-cycle CPU in Figure 10-15. The pipeline platforms can be ignored in this determination; however, it is important to examine the timing carefully to be sure that various parts of the register transfer statement for the operation take place in the right stage of the pipeline. For example, note that the adder for the *PC* is in stage EX. This adder is connected to MUX *C* and its attached control logic, and to the incrementer +1 for the *PC*. Thus, all of this logic is in the EX stage, and the loading

□ **TABLE 12-2**
Definition of Control Fields BS and PS

Register Transfer	BS Code	PS Code	Comments
$PC \leftarrow PC + 1$	00	X	Increment *PC*
$Z: PC \leftarrow BrA, \overline{Z}: PC \leftarrow PC + 1$	01	0	Branch on Zero
$\overline{Z}: PC \leftarrow BrA, Z: PC \leftarrow PC + 1$		1	Branch on Nonzero
$PC \leftarrow R[AA]$	10	X	Jump to Contents of *R*[*AA*]
$PC \leftarrow BrA$	11	X	Unconditional Branch

□ **TABLE 12-3**
Control Words for Instructions

			Control Word Values								
Symbolic Notation	Action	Op Code	RW	MD	BS	PS	MW	FS	MB	MA	CS
NOP	None	0000000	0	XX	00	X	0	XXXX	X	X	X
MOVA	$R[DR] \leftarrow R[SA]$	1000000	1	00	00	X	0	0000	X	0	X
ADD	$R[DR] \leftarrow R[SA] + R[SB]$	0000010	1	00	00	X	0	0010	0	0	X
SUB	$R[DR] \leftarrow R[SA] + \overline{R[SB]} + 1$	0000101	1	00	00	X	0	0101	0	0	X
AND	$R[DR] \leftarrow R[SA] \wedge R[SB]$	0001000	1	00	00	X	0	1000	0	0	X
OR	$R[DR] \leftarrow \mathrm{R}[SA] \vee \mathrm{R}[SB]$	0001001	1	00	00	X	0	1001	0	0	X
XOR	$R[DR] \leftarrow \mathrm{R}[SA] \oplus \mathrm{R}[SB]$	0001010	1	00	00	X	0	1010	0	0	X
NOT	$R[DR] \leftarrow \overline{R[SA]}$	0001011	1	00	00	X	0	1011	X	0	X
ADI	$R[DR] \leftarrow R[SA] + \text{se}\,IM$	0100010	1	00	00	X	0	0010	1	0	1
SBI	$R[DR] \leftarrow R[SA] + \overline{(\text{se}\,IM)} + 1$	0100101	1	00	00	X	0	0101	1	0	1
ANI	$R[DR] \leftarrow R[SA] \wedge \text{zf}\,IM$	0101000	1	00	00	X	0	1000	1	0	0
ORI	$R[DR] \leftarrow R[SA] \vee \text{zf}\,IM$	0101001	1	00	00	X	0	1001	1	0	0
XRI	$R[DR] \leftarrow R[SA] \oplus \text{zf}\,IM$	0101010	1	00	00	X	0	1010	1	0	0
AIU	$R[DR] \leftarrow R[SA] + \text{zf}\,IM$	1000010	1	00	00	X	0	0010	1	0	0
SIU	$R[DR] \leftarrow R[SA] + \overline{(\text{zf}\,IM)} + 1$	1000101	1	00	00	X	0	0101	1	0	0
MOVB	$R[DR] \leftarrow R[SB]$	0001100	1	00	00	X	0	1100	0	X	X
LSR	$R[DR] \leftarrow \text{lsr}\ R[SA]\ \text{by}\ SH$	0001101	1	00	00	X	0	1101	X	0	X
LSL	$R[DR] \leftarrow \text{lsl}\ R[SA]\ \text{by}\ SH$	0001110	1	00	00	X	0	1110	X	0	X
LD	$R[DR] \leftarrow M[R[SA]]$	0010000	1	01	00	X	0	XXXX	X	0	X
ST	$M[R[SA]] \leftarrow R[SB]$	0100000	0	XX	00	X	1	XXXX	0	0	X
JMR	$PC \leftarrow R[SA]$	1110000	0	XX	10	X	0	XXXX	X	0	X
SLT	If $R[SA] < R[SB]$ then $R[DR] = 1$	1100101	1	10	00	X	0	0101	0	0	X
BZ	If $R[SA] = 0$, then $PC \leftarrow PC + 1 + \text{se}\,IM$	1100000	0	XX	01	0	0	0000	1	0	1
BNZ	If $R[SA] \neq 0$, then $PC \leftarrow PC + 1 + \text{se}\,IM$	1010000	0	XX	01	1	0	0000	1	0	1
JMP	$PC \leftarrow PC + 1 + \text{se}\,IM$	1101000	0	XX	11	X	0	XXXX	1	X	1
JML	$PC \leftarrow PC + 1 + \text{se}\,IM, R[DR] \leftarrow PC + 1$	0110000	1	00	11	X	0	0000	1	1	1

of the *PC* that begins the IF stage is controlled from the EX stage. Likewise, the input *R*[*AA*] is in the same combinational block of logic and comes not from the *A* Data output of the register file, but from Bus *A* in the EX stage, as shown.

Table 12-3 can serve as the basis for the design of the instruction decoder. It contains the values for all control signals, except the register addresses from *IR*. In contrast to the instruction decoder in Section 10-8, the logic is complex and is most easily designed by using a computer-based logic synthesis program.

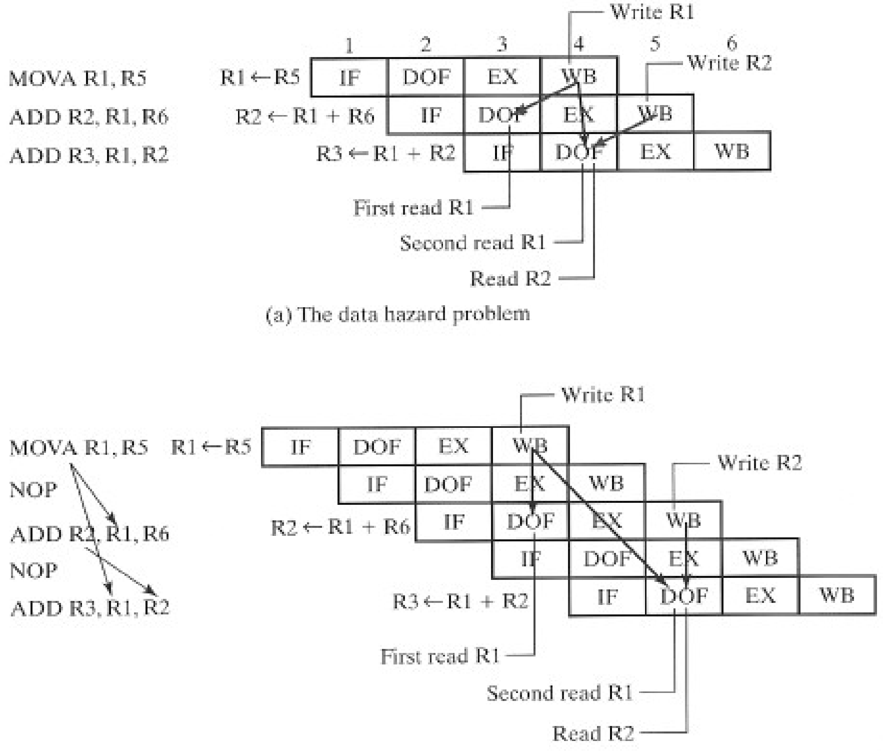
Data Hazards

In Section 12-1, we examined a pipeline execution diagram and found that filling and flushing of the pipeline reduced the throughput below the maximum level achievable. Unfortunately, there are other problems with pipeline operation that reduce throughput. In this and the next subsection, we will examine two such problems: data hazards and control hazards. Hazards are timing problems that arise because the execution of an operation in a pipeline is delayed by one or more clock cycles from the time at which the instruction containing the operation was fetched. If a subsequent instruction tries to use the result of the operation as an operand before the result is available, it uses the old or stale value, which is very likely to give a wrong result. To deal with data hazards, we present two solutions, one that uses software and another that uses hardware.

Two data hazards are illustrated by examining the execution of the following program:

```
1  MOVA  R1, R5
2  ADD   R2, R1, R6
3  ADD   R3, R1, R2
```

The execution diagram of this program appears in Figure 12-10(a). The MOVA instruction places the contents of *R*5 into *R*1 in the first half of WB in cycle 4. But,



(a) The data hazard problem

(b) A program-based solution

□ **FIGURE 12-10**
Example of Data Hazard

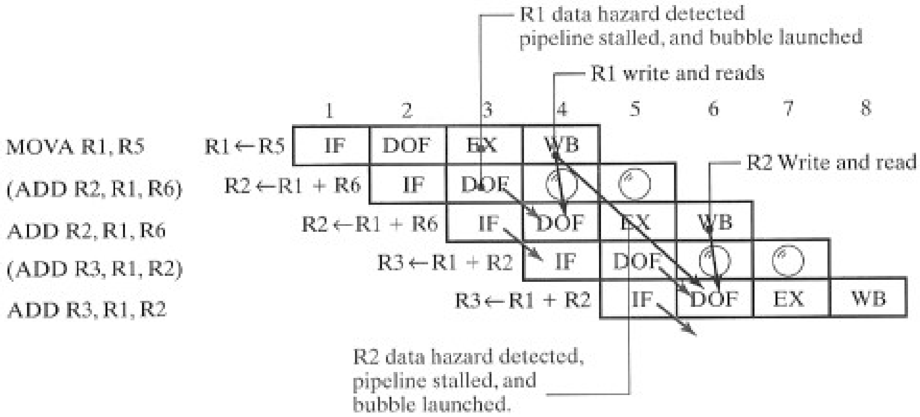
as shown by the blue arrow, the first ADD instruction reads $R1$ in the last half of DOF in cycle 3, one cycle before it is written. Thus, the ADD instruction uses the stale value in $R1$. The result of this operation is placed in $R2$ in the first half of WB in cycle 5. The second ADD instruction, however, reads both $R1$ and $R2$ in the second half of DOF in cycle 4. In the case of $R1$, the value read was written in the first half of WB in cycle 4. So the value read in the second half of cycle 4 is the new value. The write-back of $R2$, however, occurs in the first half of cycle 5, after it is read by the next instruction during cycle 4. So $R2$ has not been updated to the new value at the time it is read. This gives two data hazards, as indicated by the large blue arrows in the figure. The registers that are not properly updated to new values are highlighted in blue in the program and in the register transfer statements in the figure. In each of these cases, the read of the involved register occurs one clock cycle too soon with respect to the write of that register.

One possible remedy for data hazards is to have the compiler or programmer generate the machine code to delay instructions so that new values are available. The program is written so that any pending write to a register occurs in the same or an earlier clock cycle than a subsequent read from the register. To accomplish this, the programmer or compiler needs to have detailed information on how the pipeline operates. Figure 12-10(b) illustrates a modification of the simple three-line program that solves the problem. No-operation (NOP) instructions are inserted between the first and second instructions, and between the second and third instructions to delay the respective reads relative to the writes by one clock cycle. The execution diagram shows that, at worst, this approach has writes and subsequent reads in the same clock cycle. This is indicated by the pairs consisting of a register write and a subsequent register read connected by a black arrow in the diagram. Because of the read-after-write assumption for the register file, the timing shown permits the program to be executed on correct operands.

This approach solves the problem, but what is the cost? First of all, the program is obviously longer, although it may be possible to place other, unrelated instructions in the NOP positions instead of just wasting them. Also, the program takes two clock cycles longer and reduces the throughput from 0.5 instruction per cycle to 0.375 instruction per cycle with the NOPs in place.

Figure 12-11 illustrates an alternative solution involving added hardware. Instead of the programmer or compiler putting NOPs in the program, the hardware inserts the NOPs automatically. When an operand is found at the DOF stage that has not been written back yet, the associated execution and write-back are delayed by stalling the pipeline flow in IF and DOF for one clock cycle. Then the flow resumes with completion of the instruction when the operand becomes available, and a new instruction is fetched as usual. The delay of one cycle is enough to permit the result to be written before it is read as an operand.

When the actions associated with an instruction flowing through the pipe are prevented from happening at a given point, the pipeline is said to contain a *bubble* in subsequent clock cycles and stages for that instruction. In Figure 12-11, when the flow for the first ADD instruction is prevented beyond the DOF stage, in the next two clock cycles a bubble passes through the EX and the WB stages, respectively. The holding of the pipeline flow in the IF and DOF stages delays the



□ **FIGURE 12-11**
Example of Data Hazard Stall

microoperations taking place in these stages for one clock cycle. In the figure, this delay is represented by two diagonal blue arrows from the initial location in which the completion of the microoperation is prevented to the location one clock cycle later in which the microoperation is performed. When the pipeline flow is held in IF and DOF for an extra clock cycle, the pipeline is said to be *stalled*, and if the cause of the stall is a data hazard, then the stall is referred to as a *data hazard stall*.

An implementation of data hazard handling for the pipelined RISC that uses data hazard stalls is presented in Figure 12-12. The added or modified hardware is shown in the areas shaded in light blue. For this particular pipeline stage arrangement, a data hazard will occur for a register file read if there is a destination register at the execution stage that is to be written back in the next clock cycle and that is to be read at the current DOF stage as either of the two operands. So we have to determine whether such a register exists. This is done by evaluating the Boolean equations

$$HA = \overline{MA_{\mathrm{DOF}}}\cdot(DA_{\mathrm{EX}} = AA_{\mathrm{DOF}})\cdot RW_{\mathrm{EX}}\cdot\sum_{i=0}^{4}(DA_{\mathrm{EX}})_i$$

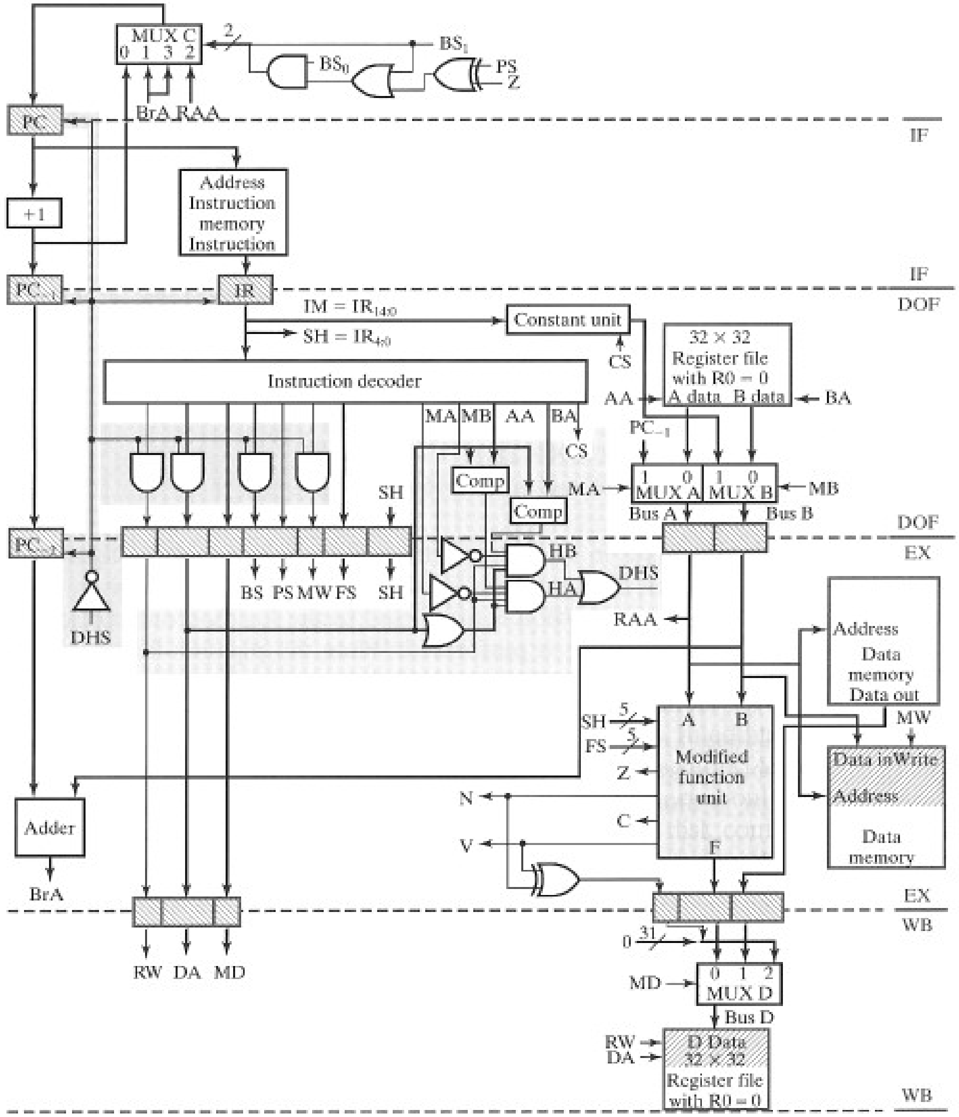
$$HB = \overline{MB_{\mathrm{DOF}}}\cdot(DA_{\mathrm{EX}} = BA_{\mathrm{DOF}})\cdot RW_{\mathrm{EX}}\cdot\sum_{i=0}^{4}(DA_{\mathrm{EX}})_i$$

and

$$DHS = HA + HB$$

The following events must all occur for HA, which represents a hazard for the A data, to equal 1:

1. MA in the DOF stage must be 0, meaning that the A operand is coming from the register file.



□ **FIGURE 12-12**
Pipelined RISC: Data Hazard Stall

2. *AA* in the DOF stage equals *DA* in the EX stage, meaning that there is potentially a register being read in the DOF stage that is to be written in the next clock cycle.
3. *RW* in the EX stage is 1, meaning that register *DA* in the EX stage will definitely be written in WB during the next clock cycle.

4. The OR (Σ) of all bits of DA is 1, meaning that the register to be written is not $R0$ and so is a register that must be written before being read. ($R0$ has the same value 0 regardless of any writes to it.)

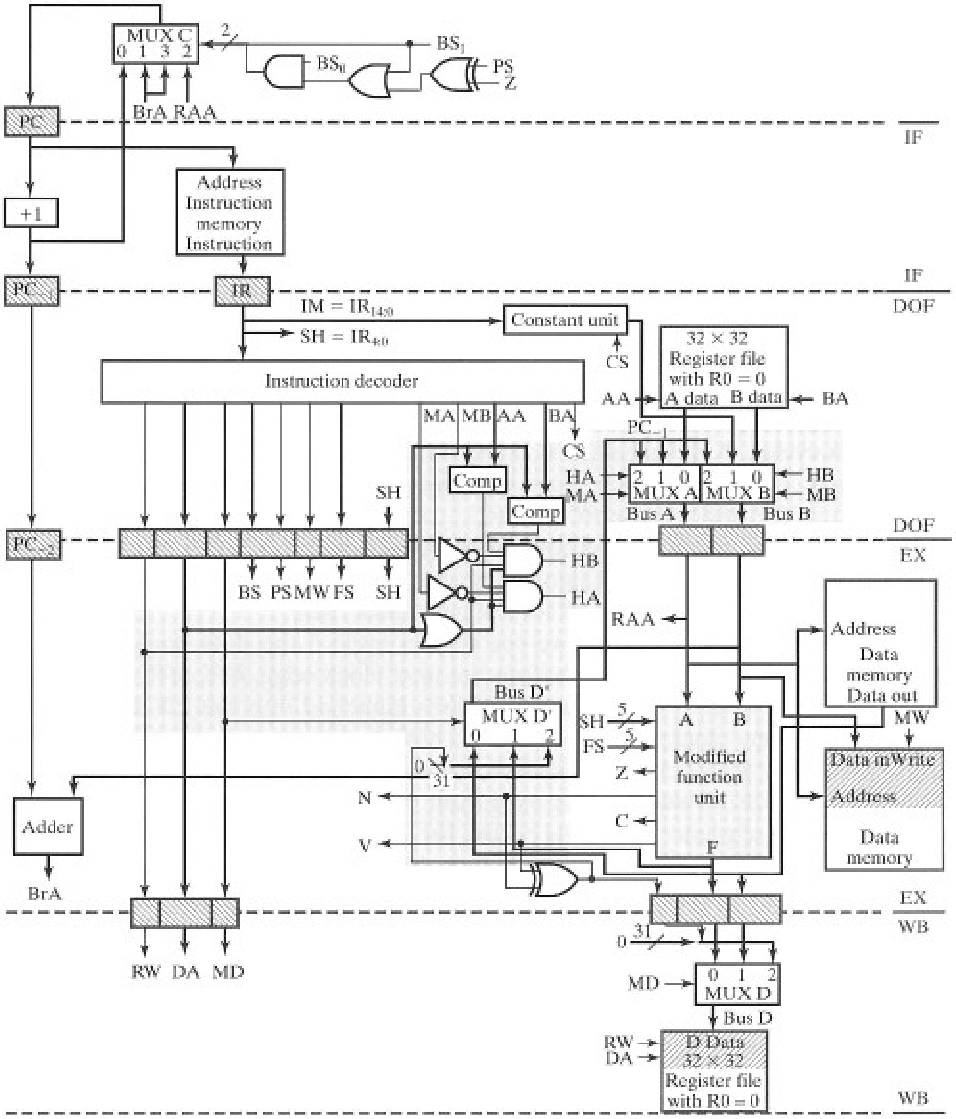
If all these conditions hold, there is a write pending for the next clock cycle to a register that is the same as one being read and used on Bus A. Thus, a data hazard exists for the A operand from the register file. HB represents the same combination of events for the B data. If either of the HA or HB terms equals 1, there is a data hazard and DHS is 1, meaning that a data hazard stall is required.

The logic implementing the preceding equations is shown in the shaded area in the center of Figure 12-12. The blocks marked "Comp" are equality comparators that have output 1 if and only if the two 5-bit inputs are equal. The OR gate with DA entering it ORs together the five bits of DA and has output 1 as long as DA is not 00000 ($R0$).

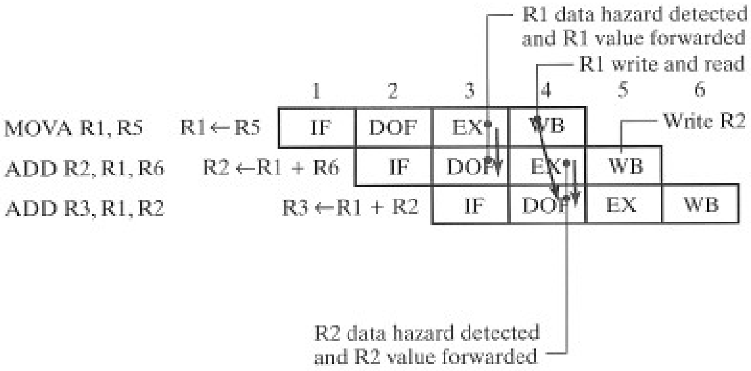
DHS is inverted and the inverted signal is used to initiate a bubble in the pipeline for the instruction currently in the IR, as well as to stop the PC and IR from changing. The bubble, which prevents actions from occurring as the instruction passes through the EX and WB stages, is produced by using AND gates to force RW and MW to 0. These 0s prevent the instruction from writing the register file and the memory. AND gates also force BS to 0 causing the PC to be incremented instead of loaded during the EX stage for a jump register or branch instruction affected by a data hazard. Finally, to prevent the data stall from continuing for the next and subsequent clock cycles, AND gates force DA to 0 so that it appears that $R0$ is being written, giving a condition which does not cause a stall. The registers to remain unchanged in the stall are the PC, the PC_{-1}, PC_{-2}, and the IR. These registers are replaced with registers with load control signals driven by $\overline{DHS}$. When $\overline{DHS}$ goes to 0, requesting a stall, the load signals become 0 and these pipeline platform registers hold their contents unchanged for the next clock cycle.

Returning to Figure 12-12, we see that in cycle 3 the data hazard for $R1$ is detected, so that $\overline{DHS}$ goes to 0 before the next clock edge. RW, MW, BS, and DA are set to 0, and at the clock edge, a bubble is launched into the EX stage for the ADD. At the same clock edge, the IF and DOF stages are stalled, so the information in them now is associated with clock cycle 4 instead of 3. In clock cycle 4, since DA_{EX} is 0, there is no stall, so the execution of the stalled ADD instruction proceeds. The same sequence of events occurs for the next ADD. Note that the execution diagram is identical to that in Figure 12-10(b), except that the NOPs are replaced by stalled instructions, shown in parentheses. Thus, although it removes the need for programming NOPs into the software, the data hazard stall solution has the same throughput penalty as the program with the NOPs.

A second hardware solution, *data forwarding*, does not have this penalty. Data forwarding is based on the answer to the following question: When a data hazard is detected, is the result available somewhere else in the pipeline, so that it can be used immediately in the operation having the data hazard? The answer is "almost." The result will be on Bus D, but it is not available until the next clock cycle. The result is to be written into the destination register during that clock cycle. The information needed to form the result, however, is available on the inputs to the pipeline platform that provides the inputs to MUX *D*. All that is needed to form the result during the current clock cycle is a multiplexer to select from the three values, just as MUX *D* does. MUX *D′* is accordingly added to produce the result on Bus *D′*. In Figure 12-13, instead of reading the operand from the register file, we use data forwarding to replace the operand with the value on Bus D′ . This replacement is implemented with an additional input to MUX *A* and to MUX *B* from Bus *D′* as shown. Essentially the same logic as before is used to detect the data hazard, except that the separate detection signals *HA* and *HB* are



□ **FIGURE 12-13**
Pipeline RISC: Data Forwarding



□ **FIGURE 12-14**
Example of Data Forwarding

used directly for *A* data and *B* data, respectively, so that the replacement occurs for the operand that has the data hazard.

The data-forwarding execution diagram for the three-instruction example appears in Figure 12-14. The data hazard for *R*1 is detected in cycle 3. This causes the value to go into *R*1 in the next cycle, to be forwarded from the EX stage of the first instruction in cycle 3. The correct value of *R*1 enters the DOF/EX platform at the next clock edge so that execution of the first ADD can proceed normally. The data hazard for *R*2 is detected in cycle 4, and the correct value is forwarded from the EX stage of the second instruction in that cycle. This gives the correct value in the DOF/EX platform needed for the second ADD to proceed normally. In contrast to the data hazard stall method, data forwarding does not increase the number of clock cycles required to execute the program and hence does not affect the throughput in terms of the number of clock cycles required. It may, however, add combinational delay, causing the clock period to be somewhat longer.

Data hazards can also occur with memory access, as well as with register access. For the ST and LD instructions, it is not likely that a data memory read can be performed after a write in a single clock cycle. Further, some memory reads may take more than one clock cycle, in contrast to what we have assumed here. Thus, the reduction in throughput for a data hazard may be increased due to a longer delay before the data is available.

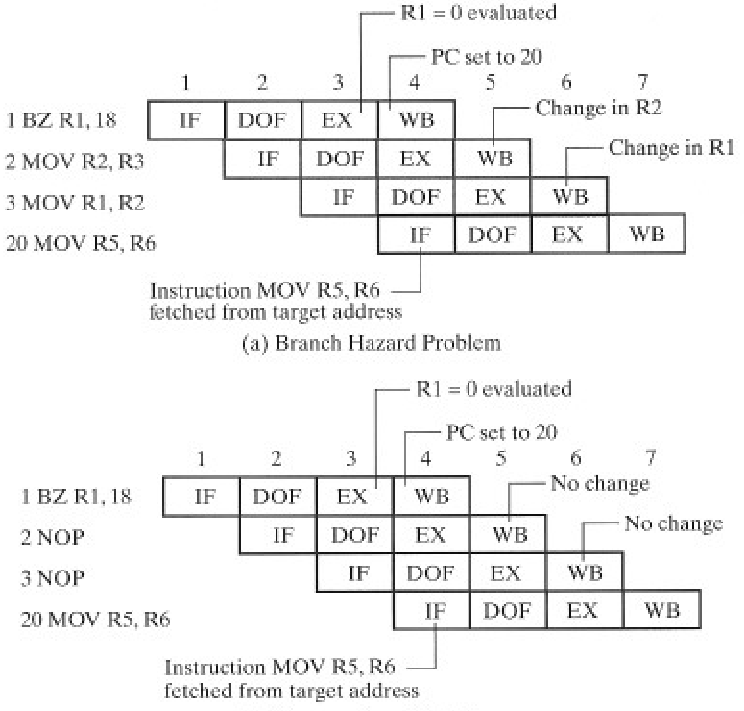
Control Hazards

Control hazards are associated with branches in the control flow of the program. The following program containing a conditional branch illustrates a control hazard:

```
 1    BZ      R1, 18
 2    MOVA    R2, R3
 3    MOVA    R1, R2
 4    MOVA    R4, R2
20    MOVA    R5, R6
```

The execution diagram for this program is given in Figure 12-15(a). If *R*1 is zero, then a branch to the instruction in location 20 (recall that addressing is PC relative) is to occur, skipping the instructions in locations 2 and 3. If *R*1 is nonzero, then the instructions in locations 2 and 3 are to be executed in sequence. Assume that the branch is taken to location 20 because *R*1 is equal to zero. The fact that *R*1 equals 0 is not detected until EX in cycle 3 of the first instruction in Figure 12-15(a). So the *PC* is set to 20 on the clock edge at the end of cycle 3. But the MOVA instructions in locations 2 and 3 are into the EX and DOF stages, respectively, after the clock edge. Thus, unless corrective action is taken, these instructions will complete execution, even though the programmer's intention was for them to be skipped. This situation is one form of a *control hazard.*

NOP instructions can be used to deal with control hazards just as they were used with data hazards. The insertion of NOPs is performed by the programmer or compiler generating the machine language program. The program must be written so that only operations intended to be performed, regardless of whether the branch is taken, are introduced into the pipeline before the branch execution actually occurs. Figure 12-15(b) illustrates a modification of the simple three-line program that satisfies this condition. Two NOPs are inserted after the branch instruction BZ. These two NOPs can be performed regardless of whether the branch is taken in the



(a) Branch Hazard Problem

(b) Program-based Solution

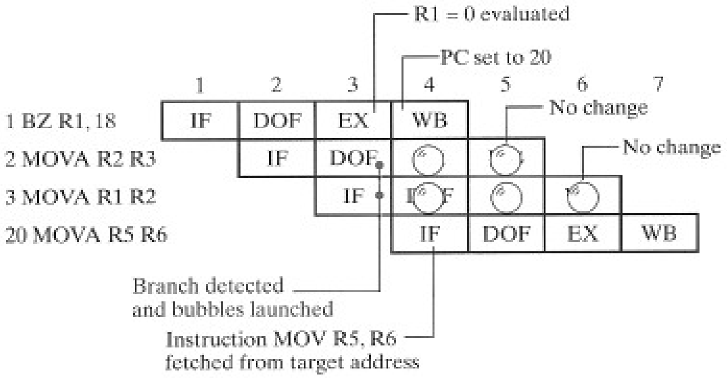
□ **FIGURE 12-15**
Example of Control Hazard

EX stage of BZ in cycle 3 with no adverse effects on the correctness of the program. When control hazards in the CPU are handled in this manner by programming, the branch hazard dealt with by the NOPs is referred to as a *delayed branch*. Branch execution is delayed by two clock cycles in this CPU.

The NOP solution in Figure 12-15(b) increases the time required to process the simple program by two clock cycles, regardless of whether the branch is taken. Note, however, that these wasted cycles can sometimes be avoided by rearranging the order of instructions. Suppose that those instructions to be executed regardless of whether the branch is taken can be placed in the two locations following the branch instruction. In this situation, the lost throughput is completely recovered.

Just as in the case of the data hazard, a stall can be used to deal with the control hazard. But, also as in the case of the data hazard, the reduction in throughput will be the same as with the insertion of NOPs. This solution is referred to as a *branch hazard stall* and will not be presented here.

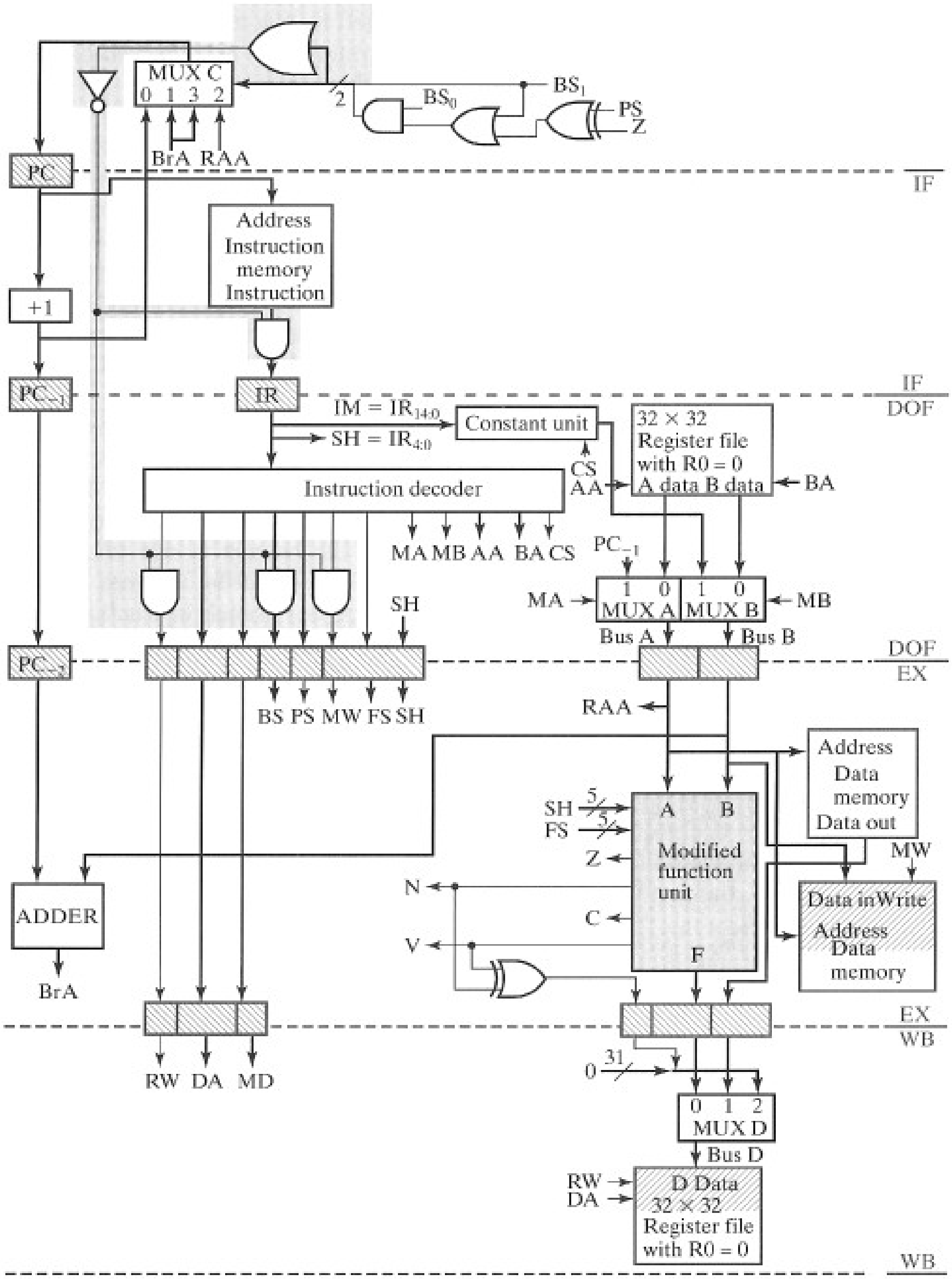
A second hardware solution is to use *branch prediction*. In its simplest form, this method predicts that branches will never be taken. Thus, instructions will be fetched and decoded and operands fetched on the basis of the addition of 1 to the value of the *PC*. These actions occur until it is known during the execution cycle whether the branch in question will be taken. If the branch is not taken, the instructions already in the pipeline due to the prediction will be allowed to proceed. If the branch is taken, the instructions following the branch instruction need to be cancelled. Usually, the cancellation is done by inserting bubbles into the execution and write-back stages for these instructions. This is illustrated for the four-instruction program in Figure 12-16. On the basis of the prediction that the branch will not be taken, the two MOVA instructions after BZ are fetched, the first one is decoded, and its operands are fetched. These actions take place in cycles 2 and 3. In cycle 3, the condition upon which the branch is based has been evaluated, and it is found that $R1 = 0$. Thus, the branch is to be taken. At the end of cycle 3, the *PC* is set to 20, and the instruction fetch in cycle 4 is performed using the new value of the *PC*. In cycle 3, the fact that the branch is taken has been detected, and bubbles



	1	2	3	4	5	6	7
1 BZ R1, 18	IF	DOF	EX	WB			
2 MOVA R2 R3		IF	DOF	○	○		
3 MOVA R1 R2			IF	I○F	○	○	
20 MOVA R5 R6				IF	DOF	EX	WB

R1 = 0 evaluated
PC set to 20
No change
No change
Branch detected and bubbles launched
Instruction MOV R5, R6 fetched from target address

□ **FIGURE 12-16**
Example of Branch Prediction with Branch Taken



□ **FIGURE 12-17**
Pipelined RISC: Branch Prediction

are inserted into the pipeline for instructions 2 and 3. Proceeding through the pipeline, these bubbles have the same effect as two NOP instructions. However, because the NOPs are not present in the program, there is no delay or performance penalty when the branch is not taken.

The branch prediction hardware is shown in Figure 12-17. Whether a branch is taken is determined by looking at the selection values on the inputs to MUX *C*. If the pair of inputs is 01, then a conditional branch is being taken. If the pair is 10, then an unconditional JMR is occurring. If the pair is 11, then an unconditional JMP or JML is taking place. On the other hand, if the pair of inputs is 00, then no branch is occurring. Thus, a branch occurs for all combinations other than 00 (i.e., for at least one 1) on the pair of lines. Logically, this corresponds to the OR of the lines, as shown in the figure. The output of the OR is inverted and then ANDed with the *RW* and *MW* fields, so that the register file and the data memory cannot be written for the instruction following the branch instruction if the branch is taken. The inverted output is also ANDed with the *BS* field, so that a branch in the next instruction is not executed. In order to cancel the second instruction following the branch, the inverted OR output is ANDed with the *IR* output. This gives an instruction of all 0's, for which the OPCODE field is defined as NOP. If the branch is not taken, however, the inverted OR output is 1, and the *IR* and the three control fields remain unchanged, giving normal execution of the two instructions following the branch.

Branch prediction can also be done on the assumption that the branch is taken. In this case, the instructions and operands must be fetched down the path of the branch target. Thus, the branch target address must be computed and used for fetching the instruction in the branch target location. In case the branch does not take place, however, the updated value of the *PC* must also be saved. As a consequence, this solution will require additional hardware to compute and store the branch target address. Nevertheless, if branches are more likely to be taken than not, the "branch taken" prediction may yield a more favorable cost–performance trade-off than the "branch not taken" prediction.

For simplicity of presentation, we have treated the hardware solutions for dealing with hazards one at a time. In an actual CPU, these solutions need to be combined. In addition, other hazards, such as those associated with writing and reading memory locations, need to be handled.

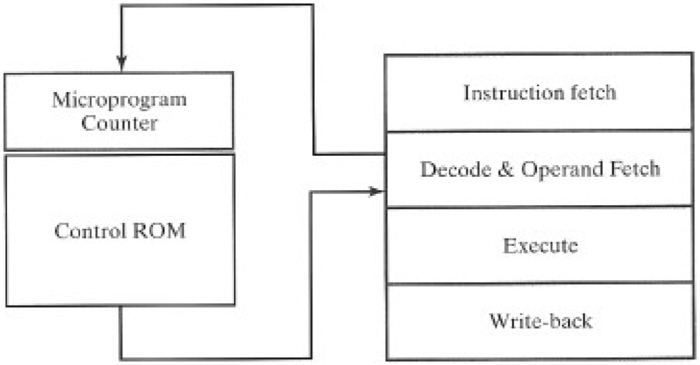
12-4 The Complex Instruction Set Computer

CISC instruction set architectures are characterized by complex instructions that are, at worst, impossible, and, at best, difficult to implement using a single cycle computer or a single pass through a pipeline. A CISC ISA often employs a sizable number of addressing modes. Further, the ISA often employs variable length instructions. The support for decision making via conditional branching is also more sophisticated than the simple concepts of branch on zero register contents and setting a register bit to 1 based on a comparison of two registers. In this section, a basic architecture for a CISC is developed with the high-performance of a RISC for simple instructions and most of the characteristics of a CISC ISA as just described.

Suppose that we are to implement a CISC architecture, but we are interested in approaching a throughput of one instruction per short RISC clock cycle for simple, frequently used instructions. To accomplish this goal, we use a pipelined datapath and a combination of pipelined and microprogrammed control as shown in Figure 12-18. An instruction is fetched into the IR and enters the Decode and Operand Fetch stage. If it is a simple instruction that executes completely in a single pass through the normal RISC pipeline, it is decoded and operand fetch occurs as usual. On the other hand, if the instruction requires multiple microoperations or multiple memory accesses in sequence, the decode stage produces a microcode address for the microcode ROM and replaces the usual decoder outputs with control values from the microcode ROM. Execution of microinstructions from the ROM, selected by the microprogram counter, continues until the execution of the instruction is completed.

Recall that to execute a sequence of microinstructions, it is often necessary to have temporary registers in which to store information. An organization of this type will frequently supply temporary registers with a convenient mechanism for switching between temporary registers and the usual programmer-accessible register resources.

The preceding organization supports an architecture that has combined CISC-RISC properties. It illustrate that pipelines and microprograms can be compatible and need not be viewed as mutually exclusive. The most frequent use of such a combined architecture allows existing software designed for a CISC to take advantage of a RISC architecture while preserving the existing ISA. The CISC-RISC architecture is a combination of concepts from the multiple-cycle computer in Chapter 10, the RISC CPU in the previous section, and the microprogramming concept introduced briefly in Chapter 10. This combination of concepts makes sense, since the CISC CPU executes instructions using multiple passes through the RISC datapath pipeline. To sequence these multiple-pass



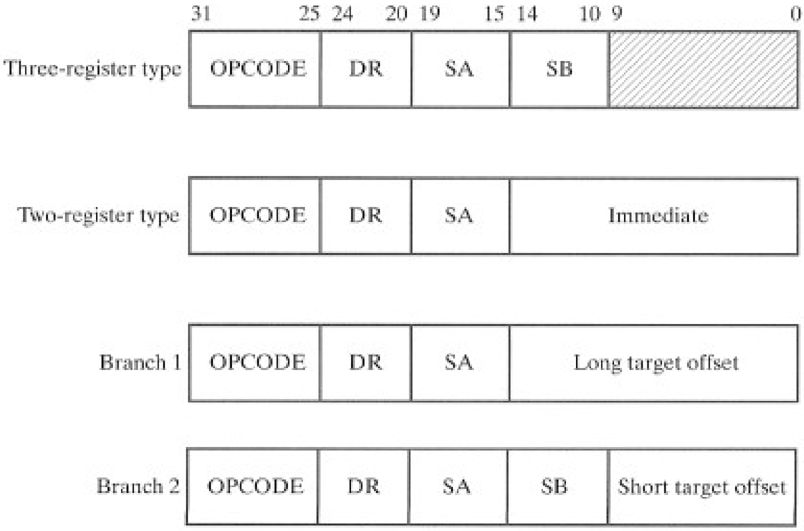
□ **FIGURE 12-18**
Combined CISC-RISC Organization

instruction implementations, a sequential control of considerable complexity is needed, so microprogrammed control is chosen.

The development of the architecture begins with some minor modifications to the RISC ISA to obtain some capabilities desirable in the CISC ISA. Next, the datapath is modified to support the ISA changes. These include modification of the Constant Unit, addition of a Condition Code register *CC*, and deletion of the hardware for supporting the SLT instruction. Further, the Register file addressing logic is modified to provide addressing for 16 temporary registers for multiple-pass use of the datapath with 16 registers remaining in the storage resources. This is in contrast to the 32 registers in the storage resources for the RISC. The next step is to adapt the RISC control to work with the microprogrammed control in implementing the multiple pass instructions. Finally, the microprogrammed control itself is developed and its operation is illustrated by the implementation of three CISC instructions that characterize a CISC ISA.

ISA Modifications

The first modification to the RISC ISA is the addition of a new format for branch instructions. In terms of the instructions provided in the CISC, it is desirable to have the capability to compare the contents of two source registers and branch, indicating the relationship between the contents of the two registers. To perform such a comparison, a format with two source register fields SA and SB and a target offset are required. Referring to Figure 12-7, addition of the SB field to the branch format reduces the length of the target offset from 15 bits to 10 bits. The resulting Branch 2 format added for the CISC instructions is shown in Figure 12-19.



□ **FIGURE 12-19**
CISC CPU Instruction Formats

The second modification is to partition the Register file to provide addressing for 16 temporary registers for multiple-pass use of the datapath. With the partition, there are only 16 registers remaining in the storage resources. Rather than modify all of the register address fields in the instruction formats, we will simply ignore the most significant bit of these fields. For example, only the rightmost four bits of the field DR will be used. DR_4 will be ignored.

The third modification to the RISC ISA is the addition of condition codes (also called flags) as discussed in Chapter 11. The condition codes provided are designed specifically to be used in combination with branch on zero and branch on nonzero in implementing instructions that will provide a wide spectrum of decisions, such as greater than, less than, less than or equal to, etc. for both signed and unsigned integers. The codes are zero (Z), negative (N), carry (C), overflow (V), and L (less than). The first four are stored versions of the status outputs of the Function Unit. The less than (L) bit is the exclusive OR of Z and V which is useful in easily implementing particular decisions. The inclusion of the L bit in the condition codes eliminates the need for the SLT instruction.

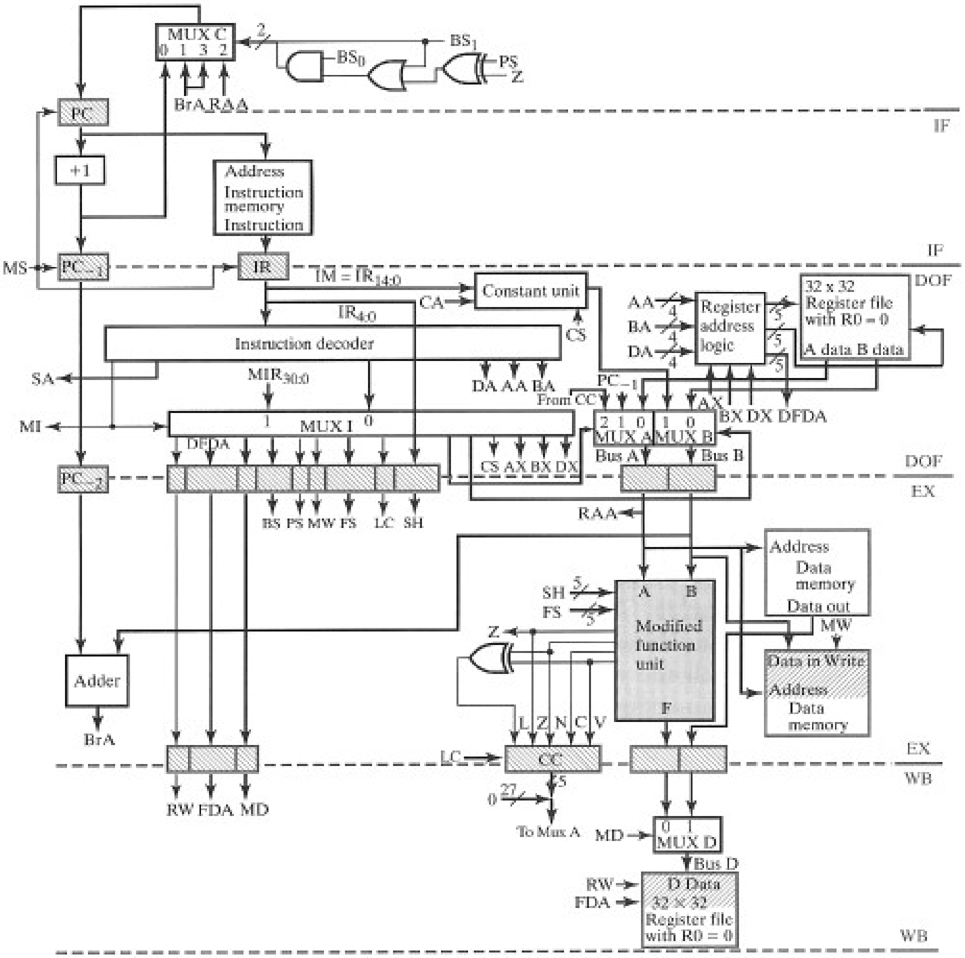
To make the most effective use of these condition codes, it is useful to control whether or not they are modified for a particular microoperation execution from the instructions. Examination of the RISC instruction codes in Table 12-1 shows that bit 4 (third from the left) of the opcode is 0 for the operations down through instruction LSL. This bit can be used for these instructions to control whether the condition codes are affected by the instruction. If the bit is 1, then the condition code values are affected by the execution of the instruction. If it is 0, then the condition codes will not be affected. This permits flexible use of the condition codes in making decisions at both the ISA level and in the microcode.

Datapath Modifications

Several changes to the datapath are required to support the ISA modifications. These changes will be covered beginning with the datapath components in the DOF stage in Figure 12-20.

First, modifications are made to the Constant unit to handle the change in the length of the target offset. Logic added to the Constant unit extracts a constant, $IM_S = IR_{9:0}$, from constant IM. Sign extension is applied to IM_S to obtain a 32-bit word. Also, for use in comparisons with condition code values, an 8-bit constant CA is provided from the microinstruction register, MIR, in the microprogrammed control. This constant is zero-filled to form a 32-bit word. The CS control field for the Constant unit is expanded to two bits to perform selection from among the four possible constant sources.

Second, the Register address logic from the multiple-cycle computer in Chapter 10 is added to the address inputs of the Register file. The purpose of this change is to support the ISA modification that provides 16 temporary registers and 16 registers that are a part of the storage resources. An additional mode supports the use of DX as a register file source address with BX as the corresponding register file destination address. This is necessary to capture the contents for R[DR] for use in destination address mode calculations.



□ **FIGURE 12-20**
Pipelined CISC CPU

Third, a number of changes are made to support the modification adding condition codes. In the DOF stage, an additional port is added on MUX *A* in order to provide access to *CC*, the stored condition codes, for storage in temporary registers or comparison to constant values. In the EX stage, the condition code bit L (less than) is implemented and the condition code register *CC* is added to the pipeline platform. The new control signal LC determines whether *CC* is loaded for the execution of a specific microoperation using a function unit operation. In the WB stage, the logic for support of the SLT instruction is replaced by a zero-filled *CC* value, which is passed to the new port on MUX *A*. Since the new condition code structure provides support for the same decision making as SLT did and more, support for SLT is no longer needed.

Control Unit Modifications

The addition of a microprogrammed control to the control unit to support instruction implementation using multiple passes through the pipeline causes significant changes to the existing control as shown in Figure 12-20. The microprogrammed control is a part of the instruction decoding hardware in the DOF stage, but it interacts with other parts of the control as well. For convenience, it will be described separately.

A quick overview of the execution of a multiple-pass instruction provides a perspective for the control unit changes. The *PC* points to the instruction in the Instruction memory. The instruction is fetched in the IF stage, and on the next clock edge, it is loaded into the *IR* and the *PC* is updated. The instruction is identified as a multiple-pass instruction from its opcode. Decoding of the opcode changes signal MI to 1 to indicate that this instruction is to use the microprogrammed control. The decoder also produces an 8-bit starting address, SA, that identifies the beginning of the microprogram in the Microcode ROM. Since multiple passes through the pipeline are needed to implement the instruction, the loading of subsequent instructions into the *IR* and further updating of the *PC* must be prevented. A signal MS produced by the microprogrammed control logic becomes 1 and stalls the *PC* and the *IR*. This prevents the *PC* from incrementing, but permits *PC* + 1 to continue down the pipeline into PC_{-1} and PC_{-2} for use in a branch. This stall remains until the multiple pass instruction has been executed or until there is branch or jump action on the *PC*. Also, when MI = 1, most of the fields of the decoded instruction are replaced with fields of the current microinstruction, which is a decoded NOP (no operation). This 31-bit field replacement, performed by MUX *I*, prevents the instruction itself from causing any direct actions. Some changes have been made to the control word to control modified datapath resources. Fields CS and MA have been expanded to two bits each, and field LC has been added. At this point, the microprogrammed control is now controlling the pipeline and supplies a series of microinstructions (control words) to implement the instruction execution. The control word format follows that for the multiple-cycle computer and includes fields such as SH, AX, BX, and DX. DX is modified to match the register address changes described for the datapath. In addition, the microprogrammed control has to interact with the datapath in order to perform decisions. This interaction includes application of the constant CA, use of the condition codes *CC*, and use of the zero detect signal *Z*.

To support the operations just discussed, the following changes are made to the control unit:

1. the addition of the stall signal MS to the *PC*, PC_{-1}, and *IR*,
2. changes in the instruction decoder to produce MI and SA,
3. expansion of the fields CS and MA to two bits,
4. addition of MUX *I*, and
5. addition of control fields AX, BX, and DX, and LC.

The definitions of new and modified control fields are given in Table 12-4.

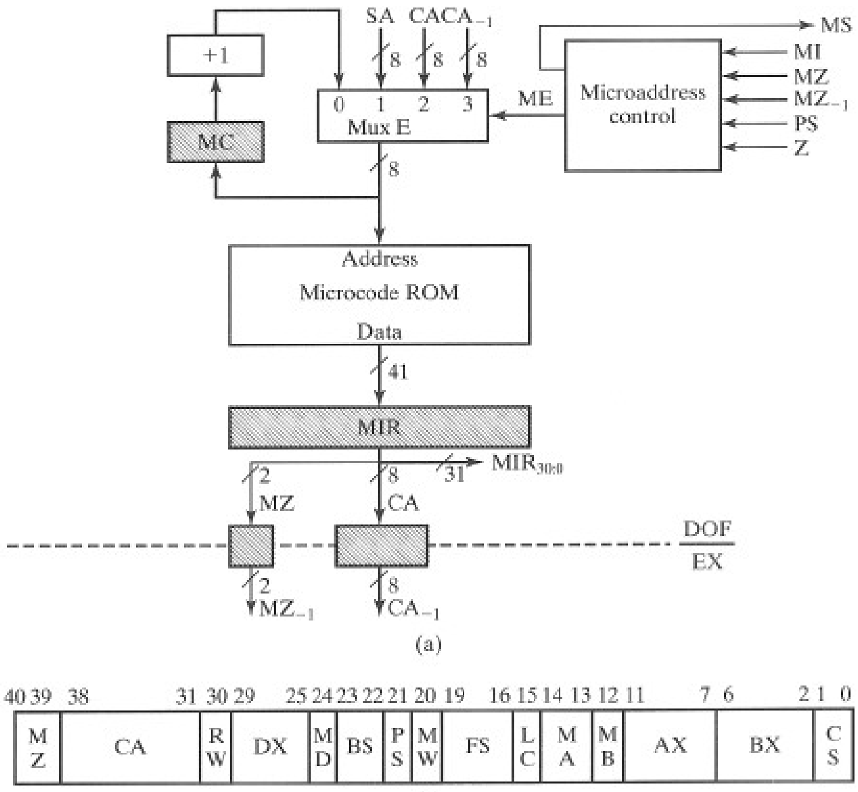
□ **TABLE 12-4**
Added or Modified Control Word (Microinstruction) Fields for CISC

Control Fields				Register Fields		CS		MA		LC	
MZ 2b	CA 8h	BS 2b	P S	Action	Code 5h	Action	Code 2b	Action	Code 2b	Action	Code
See Table 12-3	Next Address or Constant	See Table 12-2		**AX, BX**		zf *IM*	00	A Data	00	Hold *CC*	0
						se *IM*	01	PC_{-1}	01	Load *CC*	1
				R[SA], R[SB]	0X	se IM_S	10	$0 \Vert CC$	10		
				R_{16}	10	zf CA	11				
				...	...						
				R_{31}	1F						
				DX							
				Source R[DR] and Dest. R[SB]	00						
				Dest R[DR] with X ≠ 0	0X						
				R_{16}	10						
				...	...						
				R_{31}	1F						

Except for the addition of the microprogrammed control discussed in the next section, this completes the changes to the control unit.

Microprogrammed Control

A block diagram for the microprogrammed control and the format for microinstructions appear in Figure 12-21. The control is centered about the Microcode ROM, which has an 8-bit address and stores up to 256 41-bit microinstructions. The microprogram counter *MC* stores the address corresponding to the current microinstruction stored in the microninstruction register, *MIR*. The address for the ROM is provided by MUX *E*, which selects from the incremented *MC*, the jump address obtained from the microinstruction, CA, the prior value of the jump address, CA_{-1}, and the starting address from the instruction decoder in the control unit, SA. Table 12-5 defines the 2-bit select input ME for MUX *E* and stall bit, MS, in terms of the new control field MZ plus other variables. This function is implemented by the Microaddress Control logic. To set the context for the discussion, in location 0 of the ROM, the IDLE state 0 for the microprogrammed control contains a microinstruction that is a NOP consisting of all zeros. This microinstruction has MZ = 0 and CA = 0. From Table 12-5, with MI = 0, the microprogram address is CA = 0, causing the control to remain in this state until



(a)

(b)

□ **FIGURE 12-21**
Pipelined CISC CPU: Microprogrammed Control

MI = 1. With MI = 1, starting address SA is applied to fetch the first microinstruction of the microprogram for the complex instruction being held in *IR*. In the control unit, MI = 1 also switches MUX *I* from the normal control word coming from the decoder to the 31-bit *MIR* portion that is a NOP instruction. In addition, the output MS from the Microaddress control becomes 1, stalling the *PC*, *PC*$_{-1}$, and the *IR* in the main control. At the next clock edge, the microinstruction fetched from the starting address SA enters the *MIR*, and the pipeline is now controlled by the microprogram.

In Figure 12-21, two pipeline registers are required as a part of the microprogrammed control. The stored pipeline values, *MZ*$_{-1}$ and *CA*$_{-1}$, are required for the execution of a conditional microbranch since the value of *Z* to be tested occurs during the execution cycle for the microbranch instruction, one clock cycle after it enters the *MIR*.

During the execution of the microprogram, the microaddress is controlled by MZ, MZ$_{-1}$, MI, PS, and Z. For MZ$_{-1}$ = 11, MZ = 01 since the microinstruction following a conditional microbranch must be a NOP. Under these conditions, the ME values are controlled by PS and Z with $MS = 1$. For PS and Z having opposite values, a conditional branch to the microaddress value from CA_{-1} occurs. Otherwise, for $\text{MZ}_{-1} = 11$ and $\text{MZ} = 01$, the next microaddress becomes the incremented value of MC.

For $\text{MZ}_{-1} \neq 11$, MZ, MI, and PS control the microaddress. For MZ = 00, the values of ME and MS are controlled by MI. For MI = 0, the next microaddress is CA and $MS = 0$, corresponding to the idle state for the microprogrammed control. For MI = 1, the next microaddress is SA and $MS = 1$, selecting the next microinstruction from the Microcode ROM and stalling the first two pipeline platforms. For MZ = 01, the next microaddress is the incremented value of MC, advancing execution to the next microinstruction in sequence. For MZ = 10, an unconditional jump is performed in the microcode control and the value of MS is controlled by PS. PS = 1 causes $MS = 1$, continuing microprogram execution. PS = 0 forces $MS = 0$, removing the stall, and returning control to the pipeline. This causes MI to become 0 (if the new instruction is not also a complex one). If CA = 0, the microprogrammed control is locked the IDLE state until MI = 1. In order for this to happen, the final instruction in the microprogram must have MZ = 10, PS = 0, and CA = 0.

□ **TABLE 12-5**
Address Control

Inputs					Outputs			
MZ_{-1}	MZ	MI	PS	Z	ME_1	ME_0	MS	Register Transfer Due to ME
11	01	X	0	0	0	0	1	$\overline{PS}\cdot\overline{Z}: MC \leftarrow MC+1$
11	01	X	0	1	0	1	1	$\overline{PS}\cdot Z: MC \leftarrow CA_{-1}$
11	01	X	1	0	0	1	1	$PS\cdot\overline{Z}: MC \leftarrow CA_{-1}$
11	01	X	1	1	0	0	0	$\overline{PS}\cdot Z: MC \leftarrow MC+1$
0X	01	X	X	X	0	0	1	$MC \leftarrow MC+1$
X0	01	X	X	X	0	0	1	$MC \leftarrow MC+1$
XX	00	0	X	X	1	0	0	$MC \leftarrow CA$
XX	00	1	X	X	0	1	1	$MC \leftarrow SA$
XX	10	X	0	X	1	0	0	$\overline{PS}: MC \leftarrow CA$
XX	10	X	1	X	1	0	1	$PS: MC \leftarrow CA$
XX	11	X	X	X	0	0	1	$MC \leftarrow MC+1$

Microprograms for Complex Instructions

Three examples illustrate complex instructions implemented by using the CISC capabilities provided by the design just completed. The resulting microprograms are given in Table 12-6.

EXAMPLE 12-1 LD Instruction with Indirect Indexed Addressing (LII)

The LII instruction adds the target offset to the contents of a register that is being used as an index register. In the indirection step, the indexed address formed is then used to fetch the effective address from memory. Finally, the effective address is used to fetch the operand from memory. The opcode for this instruction is 0110001, and the instruction uses the Immediate format with the SA register field and a 15-bit target offset. When the LII instruction is fetched and appears in the *IR*, the instruction decoder sets MI equal to 1 and provides the microcode address symbolically represented by LII0 in Table 12-6. The first microinstruction to be executed is the one appearing in the IDLE address. This microoperation executes a NOP in the datapath and memory, but in the presence of MI = 1, the address control selects SA as the next microinstruction address, thereby leaving the IDLE state. The LII0 microinstruction forms the indexed address and increments the address in *MC* to fetch the next microinstruction LII1. This causes the NOP microinstruction in address LII1 to be fetched for execution in the pipeline. This NOP has been inserted, since the result of the microinstruction in LII0 is not placed in R_{16} until the WB stage. The next microinstruction in LII2 fetches the effective address from memory. A NOP is required next, due the clock cycle delay in writing the effective address to R_{17}. The microinstruction in LII4 applies the effective address to the memory to obtain the operand and place it in the destination register *R*[DR]. Since this completes the LII implementation, the microprogrammed control state in *MC* returns to IDLE and the next instruction following LII is fetched from the instruction memory by using the address in the *PC*. ■

In Table 12-6, this sequence of microinstructions is described in the Action column by register transfer statements, and symbolic names are provided for the addresses of the microinstructions in the Microcode ROM. The remainder of the columns in the table provide the coding of the microinstruction fields. These codes are selected from Tables 10-12, 12-2, 12-3, and 12-5, to implement the register transfers. Of particular note is the appearance of MC = 10, PS = 0, and CA = IDLE (00) in microinstruction LII4 causing the microprogram control to return to IDLE and program control to return to the pipeline control.

EXAMPLE 12-2 Branch on Less Than or Equal to (BLE)

The BLE instruction compares the contents of registers *R*[SA] and *R*[SB]. If *R*[SA] is less than or equal to *R*[SB], then the *PC* branches to *PC* + 1 plus the sign-extended Short Target Offset (IM_S). Otherwise, the incremented *PC* is used. The opcode for the instruction is 1100101.

□ **TABLE 12-6**
Example Microprograms for CISC Architecture

		Microinstructions														
				R		M		P	M		L		M			
Action	Address	MZ	CA	W	DX	D	BS	S	W	FS	C	MA	B	AX	BX	CS
Shared Microinstructions																
$MI: MC \leftarrow SA, \overline{MI}: MC \leftarrow 00$	IDLE	00	00	0	00	0	00	0	0	0	0	00	0	00	00	00
$MC \leftarrow MC + 1$ (NOP)	Arbitrary	01	XX	0	00	0	00	0	0	0	0	00	0	00	00	00
Load Indirect Indexed (LII)																
$R_{16} \leftarrow R[SA] + \text{zf } IM_L$	LII0	01	00	1	10	0	00	0	0	2	0	00	1	00	00	00
$MC \leftarrow MC + 1$ (NOP)	LII1	01	00	0	00	0	00	0	0	0	0	00	0	00	00	00
$R_{17} \leftarrow M[R_{16}]$	LII2	01	00	1	11	1	00	0	0	0	0	00	0	10	00	00
$MC \leftarrow MC + 1$ (NOP)	LII3	01	00	0	00	0	00	0	0	0	0	00	0	00	00	00
$R[DR] \leftarrow M[R_{17}]$	LII4	10	IDLE	1	01	1	00	0	0	0	0	00	0	11	00	00
Compare Less Than or Equal To (BLE)																
$R[SA] - R[SB]$, $CC \leftarrow L \Vert Z \Vert N \Vert C \Vert V$	BLE0	01	00	0	01	0	00	0	0	5	1	00	0	00	00	00
$MC \leftarrow MC + 1$ (NOP)	BLE1	01	00	0	00	0	00	0	0	0	0	00	0	00	00	00
$R_{31} \leftarrow CC \wedge 11000$	BLE2	01	18	1	1F	0	00	0	0	8	0	10	1	00	00	11
$MC \leftarrow MC + 1$ (NOP)	BLE3	01	00	0	00	0	00	0	0	0	0	00	0	00	00	00
if $(R_{31} \neq 0) MC \leftarrow BLE7$ else MC $\leftarrow MC + 1$	BLE4	11	BLE7	0	00	0	00	1	0	0	0	00	0	1F	00	00
$MC \leftarrow MC + 1$ (NOP)	BLE5	01	00	0	00	0	00	0	0	0	0	00	0	00	00	00
$MC \leftarrow IDLE$	BLE6	00	IDLE	0	00	0	00	0	0	0	0	00	0	00	00	00
$PC \leftarrow (PC_{-1}) + \text{se } IM_L$, $MC \leftarrow IDLE$	BLE7	10	IDLE	0	00	0	11	0	0	0	0	01	1	00	00	10
Move Memory Block (MMB)																
$R_{16} \leftarrow R[SB]$	MMB0	01	00	1	10	0	00	0	0	C	0	00	0	00	00	00
$MC \leftarrow MC + 1$ (NOP)	MMB1	01	00	0	00	0	00	0	0	0	0	00	0	00	00	00
$R_{16} \leftarrow R_{16} - 1$	MMB2	01	01	1	10	0	00	0	0	5	0	00	1	00	00	11
$R_{17} \leftarrow R[DR]$	MMB3	01	00	1	00	0	00	0	0	C	0	00	0	00	11	00
$R_{18} \leftarrow R[SA] + R_{16}$	MMB4	01	00	1	12	0	00	0	0	2	0	00	0	00	10	00
$R_{19} \leftarrow R_{17} + R_{16}$	MMB5	01	00	1	13	0	00	0	0	2	0	00	0	11	10	00
$R_{20} \leftarrow M[R_{18}]$	MMB6	01	00	1	14	1	00	0	0	0	0	00	0	12	00	00
$MC \leftarrow MC + 1$ (NOP)	MMB7	01	00	0	00	0	00	0	0	0	0	00	0	00	00	00
$M[R_{19}] \leftarrow R_{20}$	MMB8	01	00	0	00	0	00	0	1	0	0	00	0	13	14	00
if $(R_{16} \neq 0) MC \leftarrow MMB2$	MMB9	11	MMB2	0	00	0	00	1	0	0	1	00	0	10	00	00
$MC \leftarrow MC + 1$ (NOP)	MMB10	01	00	0	00	0	00	0	0	0	0	00	0	00	00	00
$MC \leftarrow IDLE$	MMB11	10	IDLE	0	00	0	00	0	0	0	0	00	0	00	00	00

The register transfers for the instruction are given in the Action column of Table 12-6. In microinstruction BLE0, R[SB] is subtracted from R[SA] and the condition codes L through V are captured in register CC. Due to the one-cycle delay in writing to CC, a NOP is required in microinstruction BLE1. R[SA] is less than or equal to R[SB] if (L + Z) = 1 (+ is OR in this expression). Thus, of the five condition code bits, only L and Z are of interest. So in microinstruction BLE2, the least significant three bits of CC are masked out using the mask 11000 ANDed with CC. The result is placed in register R_{31}, and, in BLE3, another NOP is required waiting for R_{31} to be written. In BLE4, a microbranch on R_{31} nonzero occurs. If R_{31} is nonzero, then L + Z = 1 giving R[SA] less than or equal to R[SB]. Otherwise, both L and Z are 0 indicating R[SA] is not less than or equal to R[SB]. Due to the microbranch, a NOP is required in BLE5. The connections to MUX E require only one NOP after a microbranch instead of the two NOPs needed for the conditional branch in the main control. If the branch is not taken, the next microinstruction BLE6 executes, returning MC to IDLE and reactivating the pipeline control to execute the next instruction. If the branch is taken, microinstruction BLE7 is executed, placing $PC + 1 + BrA$ into the PC for fetching the next instruction when the microinstruction reaches the EX stage. Note that such a branch on the PC can take place only after MS becomes 0 and the pipeline is reactivated. In this regard, a control hazard exists for this instruction in the main control, so it must be followed by a NOP. The codes for the microinstruction fields appear in Table 12-6 ■

EXAMPLE 12-3 Move Memory Block (MMB)

The MMB instruction copies a block of information from one set of contiguous locations in memory to another. It has opcode 0100011 and uses the three-register type format. Register R[SA] specifies address A, the beginning location of the source block in memory, and register R[DR] specifies address B, the beginning location of the destination block. R[SB] gives the number n of words in the block.

The register transfers for the instruction are given in the Action column of Table 12-6. In microinstruction MMB0, R[SB] is loaded into R_{16}. MMB1 contains a NOP waiting for $R16$ to be written. In MMB2, R_{16} is decremented, providing an index with n values, $n - 1$ down to 0, for use in addressing the copying of n words. Since R[DR] is a destination register, it is ordinarily not available as a source. But to do address manipulation for the destination locations, it is necessary for its value be placed in a register that can act as a source. Thus, in MMB3, the value of R[DR] is copied to register R_{17} by using the register code DX = 00000, which treats R[DR] as the source and the register specified in the BX field, R_{17}, as the destination. In microinstructions MMB4 and MMB5, R_{16} is added to R[SA] and to R[SB] to serve as pointers to the addresses in the blocks. Due to these operations, the words in the blocks are transferred from the highest location first. In MMB6, the first word is transferred from the first source address in memory to temporary register R_{20}. In MMB7, a NOP appears to permit the writing of the value in R_{20} by MMB6 before the use of the value by MMB8. In MMB8, the first word is transferred from R_{20} to the first destination address in memory. In MMB9, a branch on zero is done on the contents of R_{16} to determine if all of the words in the block have been transferred. If not, then MM2 is the next microaddress in which the next word transfer begins. If R_{16} equals zero, the next microinstruction is the NOP placed in MMB10 due to the branch. The final microinstruction in MMB11 returns the *MC* to IDLE and returns execution back to the pipeline control.

The codes for the microinstructions appear in Table 12-6. The code consists of simple register and memory transfers with a single branch to provide the looping capability and NOPs to deal with data and control hazards. ■

12-5 More on Design

The two designs considered in this chapter represent two different ISAs and two different supporting CPU organizations. The RISC architecture matches well with the pipelined control organization because of the simplicity of the instructions. Due to the need for high performance, the modern CISC architecture presented is built upon the RISC foundation. In this section, we will deal with additional features for speeding up the fundamental RISC pipeline. Finally, we relate the two organizations to more general digital systems design.

High-Performance CPU Concepts

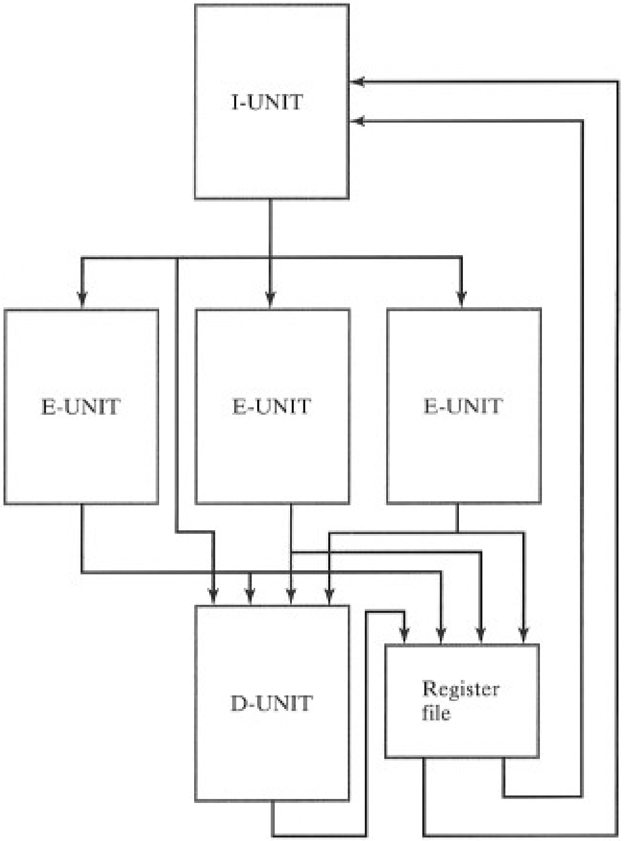
Among the various methods used to design high-speed CPUs are multiple units organized as a pipeline-parallel structure, superpipelines, and superscalar architectures.

Consider the case in which an operation takes multiple clock cycles to execute, but the instruction fetch and write-back operations can be handled in a single cycle. Then it is possible to initiate an instruction every clock cycle, but not possible to complete the execution of an instruction every cycle. In such a situation, the performance of the CPU can be substantially improved by having multiple execution units in parallel. A high-level block diagram for this kind of system is shown in Figure 12-22. The instruction fetch, decoding, and operand fetch are carried out in the I-unit pipeline. In addition, the I-unit handles branches. When decoding of a nonbranch instruction has been completed, the instruction and operands are *issued* to the appropriate E-unit. When execution of the instruction is completed by the E-unit, the write-back to the register file occurs. If a memory access is required, then the D-unit is used to execute the memory write. If the operation is a store, it goes immediately to the D-unit. Note that the actual execution units may be microprogrammed and may also have internal pipelines.

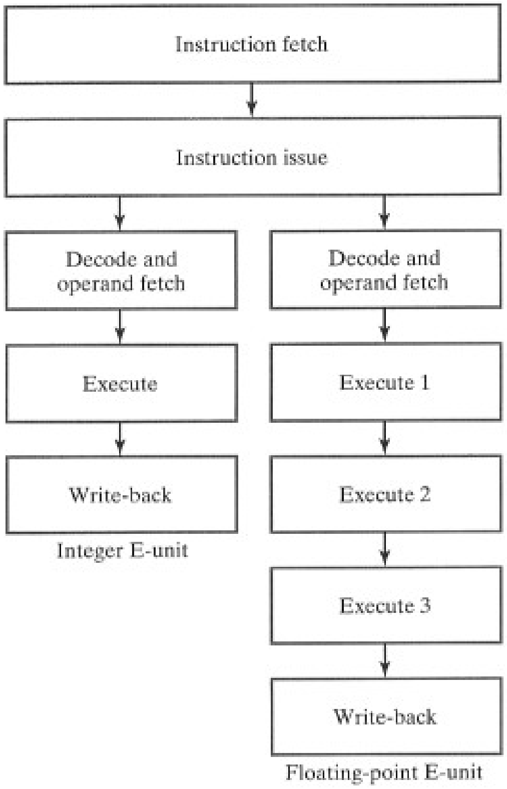
Suppose that a sequence of three instructions—say, a multiplication, a 16-bit shift, and an addition—has no data hazards. Suppose further that there is a single pipelined E-unit that performs all of these operations, which take 17, 8, and 2 clock cycles, respectively, and that both the multiplication and the shift require multiple passes through portions of the E-unit pipeline. This situation allows only one clock cycle of overlap between pairs of the three instructions. Thus, the fastest that the sequence of operations executes in the E-unit is 17 + 8 + 2 − 2 = 25 clock cycles.

But with an E-unit for each operation, these operations can be executed in max(17, 1 + 8, 2 + 2) clock cycles, which equals 17 clock cycles. The additional 1 and 2 are due to the issuing of one instruction per clock cycle to the E-unit set. The resulting execution throughput is improved by a factor of 25/17 = 1.5.

In all of the methods considered thus far, the peak throughput possible is one instruction per clock cycle. With this limitation, it is desirable to maximize the clock rate by minimizing the maximum pipeline stage delay. If, as a consequence, a large number of pipeline stages is used, the CPU is said to be *superpipelined*. A superpipelined CPU will generally have a very high clock frequency, in the range of a GHz. In such an organization, however, handling hazards effectively is critical, since any stalling or reinitialization of the pipeline will degrade the performance of the CPU significantly. Also, as more pipeline stages are added, further dividing up the combinational logic, the setup and propagation delay times of the flip-flops begin to dominate the platform-to-platform delay and the speed of the clock. The improvement



□ **FIGURE 12-22**
Multiple Execution Unit Organization



□ **FIGURE 12-23**
Superscalar Organization

achieved is less, and when hazards are taken into account, the performance may actually become worse rather than better.

For fast execution, an alternative to superpipelining is the use of a *superscalar* organization. The goal of this kind of organization is to have a peak rate of initiating instructions in excess of one instruction per clock cycle. A superscalar CPU that fetches a pair of instructions simultaneously by using a double-word wide path from instruction memory is illustrated in Figure 12-23. The processor checks for hazards among the instructions, as well as available execution units in the instruction issue stage of the pipeline. If there are hazards or busy execution units corresponding to the first instruction, then both instructions are held for later issuing. If the first instruction has no hazard and its E-unit is available, but there is a hazard or no available E-unit for the second instruction, then only the first instruction is issued. Otherwise, both instructions are issued in parallel. If a given superscalar architecture has the ability to issue up to four instructions simultaneously, then its peak execution rate is four instructions per clock cycle. If the clock cycle is 5 ns, then such a CPU has a peak execution rate of 800 MIPS. Note that the hazard checking for instructions in the execution stages and those in the issue stage become very complex as the maximum number of instructions issued simultaneously is increased. The resulting hardware complexity has the potential to increase the clock cycle length, so the trade-offs in such a design need to be examined very carefully.

We close this section with two observations. First, as the quest for better performance causes us to design increasingly complex organizations, hazards cause the order of the instructions to play a more important role in the throughput that is achievable. Also, improved performance can be achieved by reducing the number of hazard-producing instructions, such as branches. As a consequence, to fully exploit the performance capabilities of the hardware, the assembly language programmer and the compiler writer need to be very knowledgeable about the behavior of not only the instruction set architecture, but also the underlying organization of the hardware of the CPU.

When multiple execution units are involved, very often the CPU design we have been considering here actually becomes the design for the entire processor, as is shown for the generic computer. This is apparent in the superscalar organization in Figure 12-23, which contains the floating-point unit (FPU). The FPU, the MMU, and the portion of the internal cache that handles data are effectively types of E-units. The portion of the internal cache that handles the instructions can be viewed as a part of the I-Unit that fetches instructions. Thus, in the quest for higher and higher throughput, the realm of the CPU becomes that of the processor, as in the generic computer.

Recent Architectural Innovations

Beyond the concepts presented in the previous section, two general trends have become apparent in one of the most recent high-performance architectures. The first trend is the development of compilers and hardware architectures that permit the compiler to explicitly identify to the hardware instructions that can be executed in parallel. In this approach, the identification of parallelism typically done in hardware in the superscalar architecture has now been moved to a fair degree into the compiler. This releases hardware for other uses, notably more execution units and larger register files. The second trend is the use of techniques that allow the processor to avoid waiting for branches to be taken and for data values to become available. Three techniques that support this trend will be discussed in the remainder of this section.

Instead of waiting for a branch to be taken, the processor will execute both sides of the branch and produce results for both sides. When the results of the branch becomes available, the right result is selected and the computation proceeds. Thus, there is no delay waiting for a branch, significantly improving performance for long pipelines. This simple approach is referred to as *predication* and uses special 1-bit registers referred to as predicate registers that determine which result is used when the branch outcome is known.

Instead of waiting to load data from memory until it is known that the data is needed, *speculative loading* of data from memory is performed before it is known for sure whether or not the data is needed. The reason for use of this technique is to avoid the relatively long delay required to fetch an operand from memory. If the data that is speculatively fetched turns out to be the data needed, then the data will be available and the computation can proceed immediately without having to wait for a memory access to get the data.

Instead of waiting for data to become available, *data speculation* uses methods to predict data values and proceeds to compute using these values. When the actual value becomes known and matches the predicted value, then the result produced from the predicted value can be used to carry forward the computation. If the actual value and the predicted value differ, then the result based on the predicted value is discarded and the actual value is used to continue computation. An example of data speculation is permitting a value to be loaded from memory before a store into the same memory location occurring earlier in the program has been executed. In this case, it is predicted that the store will not change the value of the data in memory, so that the value loaded before the store will be valid. If, at the time the store occurs, the loaded value is not valid, the result of computation using it is discarded.

All of these techniques perform operations or sequences of operations for which results are discarded with some frequency. Thus, there is "wasted" computation. To be able to do large amounts of useful computation, as well as the wasted computation, more parallel resources, as well as specialized hardware for implementing the techniques, are required. The payoff in return for the cost of these resources is potentially higher performance.

Digital Systems

The two sizable digital system designs we have examined in this chapter are general-purpose CPUs. How does their design relate to that of other digital systems? First of all, each digital system has an architecture. Although that architecture may not in any way deal with instructions to be executed, it is likely that it still can be described by using register transfer descriptions and, possibly, one or more algorithmic state machines. On the other hand, it might have instructions, but they may be quite different from those for a CPU. The system may have no datapath at all or may have several datapaths. There is likely to be some form of control unit, and there may be multiple control units that interact. The system may or may not include memories. Thus, the total spectrum of digital systems has a very wide range of architectural possibilities.

So, what is the connection of the general digital system to the content of this chapter? Simply stated, the connection is design techniques. To illustrate, consider that we have shown in detail how a system with instructions can be implemented using a datapath and a control unit. From here, it is relatively easy to implement a simpler system without instructions. We have shown how high speeds can be achieved by using pipelines or parallel execution units. Thus, if the goal of a system is high speed, then pipelining or parallel units are techniques to consider. For example, one of the authors, in an example design of a system for implementing a portion of a USB transmitter (see section 13-4), used a pipelined datapath with a control that involved both pipeline and conventional sequential control. We have shown how microprogramming has been used to implement controls for complex functions carried out in a pipeline. If a system has one or more very complex functions, whether pipelined, programmable, or not, then a microprogrammed control is a possibility.

12-6 Chapter Summary

This chapter has covered the design of two processors—one for a reduced instruction set computer (RISC) and one for a complex instruction set computer CISC. As a prelude to the design of these processors, the chapter began with an illustration of a pipelined datapath. The pipeline concept enables operations to be performed with clock frequencies and throughput not achievable with the same processing components in a conventional datapath. The pipeline execution pattern diagram was introduced for visualizing the behavior of a pipeline and estimating its peak performance. The problem of the low clock frequency of the single-cycle computer was addressed by adding a pipelined control unit to the datapath.

Next, we examined a RISC design with a pipelined datapath and control unit. Based on the single-cycle computer in Chapter 10, the RISC ISA is characterized by a single instruction length, a limited number of instructions with only a few addressing modes, and memory access restricted to load and store operations. Most RISC operations are simple in the sense that, in a conventional architecture, they can be executed using a single microoperation.

The RISC ISA is implemented by using a modified version of the pipelined datapath in Figure 12-2. Modifications include an increase of the word length to 32 bits, doubling of the number of registers in the register file, and replacement of the shifter in the function unit with a barrel shifter. Likewise, a modified version of the control unit in Figure 12-4 is used. Control changes were performed to accommodate the datapath changes and to handle branches and jumps in a pipeline environment. After completion of the basic design, consideration was given to data hazard and control hazard problems. We examined each type of hazard, as well as software and hardware solutions for each.

The ISA of the CISC has the potential for performing many distinct operations, with memory access supported by several addressing modes. The CISC also has operations that are complex in the sense that they require many clock cycles for their execution. The CISC permits many of the instructions to perform memory accesses and is characterized by complex conditional branching supported by condition codes (status bits). Although, in general, a CISC ISA permits multiple instruction lengths, this feature is not provided by the example architecture.

To provide high throughput, the RISC architecture serves as the core of the CISC architecture. Simple instructions can be executed at the RISC throughput, with complex instructions, executed by multiple passes through the RISC pipeline, reducing overall throughput. RISC datapath modification provided registers for temporary operand storage and condition code storage. Changes to the control unit were required to support these datapath changes. The primary control unit modification, however, was the addition of the microprogram control for execution of complex instructions. Added changes to the RISC control unit were required to integrate the microprogram control into the control pipeline. Examples of microprograms for three complex instructions were provided.

After completing the CISC and RISC designs, we touched on some advanced concepts, including parallel execution units, superpipelined CPUs, superscalar CPUs, and predictive and speculative techniques for high performance. Finally, we related the design techniques in this chapter to more general digital system design.

References

1. Mano, M. M. *Computer System Architecture,* 3rd Ed. Englewood Cliffs, NY: Prentice Hall, 1993.
2. Patterson, D. A., and J. L. Hennessy *Computer Organization and Design: The Hardware/Software Interface,* 2nd ed. San Francisco, CA: Morgan Kaufmann, 1998.
3. Hennessy, J. L., and D. A. Patterson *Computer Architecture: A Quantitative Approach,* 2nd ed. San Francisco, CA: Morgan Kaufmann, 1996.
4. Dietmeyer, D. L., *Logic Design of Digital Systems,* 3rd ed. Boston, MA: Allyn-Bacon, 1988.
5. Kane, G., And J. Heinrich *MIPS RISC Architecture.* Englewood Cliffs, NJ: Prentice Hall, 1992.
6. Sparc International, Inc. *The SPARC Architecture Manual: Version 8.* Englewood Cliffs, NJ: Prentice Hall, 1992.
7. Weiss, S., And J. E. Smith *POWER and PowerPC.* San Mateo, CA: Morgan Kaufmann, 1994.
8. Wyant, G., and T. Hammerstrom *How Microprocessors Work.* Emeryville, CA: Ziff-Davis Press, 1994.
9. Heuring, V., and H. Jordan *Computer Systems Design and Architecture.* Upper Saddle River, NJ: Prentice-Hall,1997.

Problems

The plus (+) indicates a more advanced problem and the asterisk (*) indicates a solution is available on the Companion Website for the text.

12–1. A pipelined datapath is similar to that in Figure 12-1(b), but with the delays from the top to the bottom replaced by the following values: 1.0 ns, 1.0 ns, 0.1 ns, 0.2 ns, 1.3 ns, 0.2 ns, and 0.1 ns. Determine (a) the maximum clock frequency, (b) the latency time, and (c) the maximum throughput for this datapath.

12–2. *A program consisting of a sequence of 12 instructions without branch or jump instructions is to be executed in a six-stage pipelined computer with a clock period of 1.25 ns. Determine (a) the latency time for the pipeline, (b) the maximum throughput for the pipeline, and (c) the time required for executing the program.

12–3. The sequence of seven LDI instructions in the register number program with the pipeline execution pattern given below Figure 12-5 is fetched and executed. Manually simulate the execution by giving, for each clock cycle, the values in pipeline registers *PC*, *IR*, Data *A*, Data *B*, Data *F*, Data *I*, and in the register file having its value changed for each clock cycle. Assume that all file registers initially contain −1 (all 1's).

12–4. For each of the RISC operations in Table 12-1, list the addressing mode or modes used.

12–5. Simulate the operation of the barrel shifter in Figure 12-8 for each of the following shifts and $A = 7E93C2A1_{16}$. List the hexadecimal values on the 47 lines, 35 lines, and 32 lines out of the three levels of the shifter.

(a) Left, $SH = 11$

(b) Right, $SH = 13$

(c) Left, $SH = 30$

12–6. *For the RISC CPU in Figure 12-9, manually simulate, in hexadecimal, the processing of the instruction ADI R1 R16 2F01 located in $PC = 10F$. Assume that *R*16 contains 0000001F. Show the contents of each of the pipeline platforms and of the register file (the latter only when a change occurs) for each of the clock cycles.

12–7. Repeat Problem 12–6 for the instruction SLT R31 R10 R16 with *R*10 containing 0000100F and R16 containing 00001022.

12–8. Repeat Problem 12–6 for the instruction LSL R1 R16 000F.

12–9. +Use a computer-based logic minimization program to design the instruction decoder for a RISC from Table 12-3. The field FS need not be done, since it can be wired directly from OPCODE.

12–10. *For the RISC design, draw the execution diagram for the following RISC program, and indicate any data hazards that are present:

```
1 MOVA   R7, R6
2 SUB    R8, R8, R6
3 AND    R8, R8, R7
```

12–11. For the RISC design, draw the execution diagram for the following RISC program (with the contents of *R*7 nonzero after the subtraction), and indicate any data or control hazards that are present:

```
1 SUB     R7, R7, R6
2 BNZ     R7, 000F
3 AND     R8, R7, R6
4 OR      R5, R8, R5
```

12–12. *Rewrite the RISC programs in Problem 12–10 and Problem 12–11 using NOPs to avoid all data and control hazards and draw the new execution diagrams.

12–13. Draw the execution diagrams for the program in Problem 12–10, assuming

(a) the RISC CPU with data stall given in Figure 12-12.

(b) the RISC CPU with data forwarding in Figure 12-13.

12–14. Simulate the processing of the program in Problem 12–11 using the RISC CPU with data hazard stall in Figure 12-12. Give the contents of each pipeline platform and the register file (the latter only whenever a change occurs) for each clock cycle. Initially, *R*6 contains 00000010_{16}, *R*7 contains 00000020_{16}, *R*8 contains 00000030_{16}, and the *PC* contains 00000001_{16}. Is the data hazard avoided?

12–15. *Repeat Problem 12–14 using the RISC CPU with data forwarding in Figure 12-13.

12–16. Draw the execution diagram for the program in Problem 12–11, assuming the combination of the RISC CPU with branch prediction in Figure 12-17 and the RISC CPU with data forwarding in Figure 12-13.

12–17. Design the Constant Unit in the Pipelined CISC CPU by using the information given in Table 12-5 and multiple -bit multiplexers, AND gates, OR gates, and inverters.

12–18. *Design the Register Address Logic in the Pipelined CISC CPU by using information given in the register fields of Table 12-5 plus multiple-bit multiplexers, AND gates, OR gates, and inverters.

12–19. Design the Address Control logic described by Table 12-4 by using AND gates, OR gates, and inverters.

12–20. Write microcode for the execution part of each of the following CISC instructions. Give both a register transfer description and binary or hexadecimal representations similar to those shown in Table 12-6 for the binary code for each microinstruction.

(a) Compare Greater Than

(b) Branch if less than zero (*CC* bit N = 1)

(c) Branch if overflow (*CC* bit V = 1)

12–21. Repeat problem 12-20 for the following CISC instructions that are specified by register transfer statements.

(a) Push: $R[SA] \leftarrow R[SA]+1$ followed by $M[R[SA]] \leftarrow R[SB]$

(b) Pop: $R[DR] \leftarrow M[R[SA]]$ followed by $R[SA] \leftarrow R[SA]-1$

12–22. *Repeat problem 12-21 for the following CISC instructions.

(a) Add with carry: $R[DR] \leftarrow R[SA]+R[SB]+C$

(b) Subtract with borrow: $R[DR] \leftarrow R[SA]-R[SB]-B$

Borrow B is defined as the complement of the carry out, C.

12–23. Repeat problem 12-21 for the following CISC instructions.

(a) Add Memory Indirect: $R[DR] \leftarrow R[SA]+M[M[R[SB]]]$

(b) Add to Memory: $M[R[DR]] \leftarrow M[R[SA]]+R[SB]$

12–24. *Repeat problem 12-20 for the CISC instruction, Memory Scalar Add. This instruction uses the contents of *R*[SB] as the vector length. It adds the elements of the vector with its least significant element in memory pointed to by *R*[SA] and places the result in the memory location pointed to by *R*[DR].

12–25. Repeat problem 12-20 for the CISC instruction, Memory Vector Add. This instruction uses the contents of *R*[SB] as the vector length. It adds the vector with its least significant element in memory pointed to by *R*[SA] to the vector with its least significant element in memory pointed to by *R*[DR]. The result of the addition replaces the vector with its least significant element pointed to by *R*[DR].

CHAPTER 13

INPUT-OUTPUT AND COMMUNICATION

In this chapter, we give an overview of selected aspects of computer input-output (I/O) and communication between the CPU and I/O devices, I/O interfaces, and I/O processors. Because of the wide variety of different I/O devices and the quest for faster handling of programs and data, I/O is one of the most complex areas of computer design. As a consequence, we are able to present only selected pieces of the I/O puzzle. We illustrate just three devices in detail: a keyboard, a hard disk, and a graphics display. We then introduce the I/O bus and the I/O interfaces that connect to I/O devices. We consider serial communication and use the I/O structure for the keyboard as an illustration. We then look at the Universal Serial Bus (USB), an alternative solution to the problem of accessing I/O devices. Finally, we discuss four modes for performing data transfers: program-controlled transfer, interrupt-initiated transfer, direct memory access, and the use of an I/O processor.

In terms of the generic computer at the beginning of Chapter 1, it is apparent that I/O involves a very large part of the computer. Only the processor, external cache, and RAM are not as highly involved, although they, too, are used extensively in directing and performing I/O transfers. Even the generic computer, which has fewer I/O devices than most PC systems, has a diverse set of such devices requiring significant digital electronic hardware for support.

13-1 COMPUTER I/O

The input and output subsystem of a computer provides an efficient mode of communication between the CPU and the outside environment. Programs and data must be entered into the memory for processing, and results obtained from computations must be recorded or displayed. Among the input and output devices that are commonly found in computer systems are keyboards, monitors, printers, magnetic disks, and compact disk read-only memory (CD-ROM) drives. Other input and output devices frequently encountered are modems or other communication interfaces, scanners, and sound cards with speakers and microphones. Significant numbers of computers, such as those used in automobiles, have analog-to-digital converters, digital-to-analog converters, and other data acquisition and control components.

The I/O facility of a computer is a function of its intended application. This results in a wide diversity of attached devices and corresponding differences in the needs for interacting with them. Since each device behaves differently, it would be time consuming to dwell on the detailed interconnections needed between the computer and each peripheral. We will, therefore, examine just three peripherals that appear in most computers: the keyboard, the hard disk, and the graphics display. These represent typical points in the range of data transfer rates required for peripherals. In addition, we present some of the common characteristics found in the I/O subsystem of computers, as well as the various techniques available for transferring data either in parallel, using many conducting paths, or serially, through communication lines.

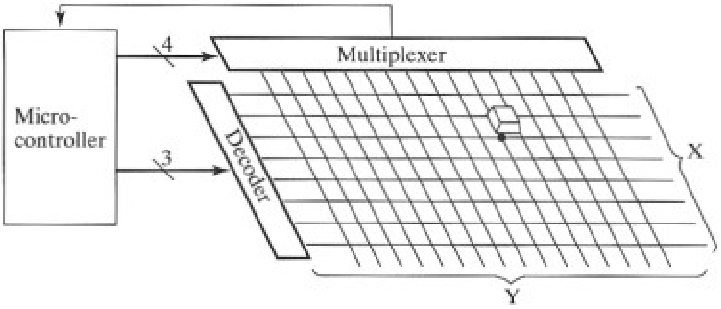
13-2 Sample Peripherals

Devices that the CPU controls directly are said to be connected *on-line.* These devices communicate directly with the CPU or transfer binary information into or out of the memory upon command from the CPU. Input or output devices attached to the computer on-line are called *peripherals.* In this section, we examine three peripheral devices: a keyboard, a hard disk, and a graphics display. We also use the keyboard as an example to illustrate I/O concepts in a later section. We introduce the hard disk both to motivate the need for direct memory access and to provide background for the role of the device in Chapter 14 as a component in a memory hierarchy. We include the graphics display to illustrate the very high potential transfer rate requirements of contemporary applications.

Keyboard

The keyboard is among the simplest of the electromechanical devices attached to the typical computer. Since it is manually controlled, it has one of the slowest data rates of any peripheral.

The keyboard consists of a collection of keys that can be depressed by the user. It is necessary to detect which of the keys have been depressed. To do this, a *scan matrix* that lies beneath the keys is used, as shown in Figure 13-1. This two-dimensional matrix is conceptually similar to the matrix used in RAM. The matrix shown in the figure is 8 × 16, giving 128 intersections, so it can handle up to 128 keys. A decoder drives the X lines of the matrix, which are analogous to the word lines of a RAM. A multiplexer is attached to the Y lines of the matrix, which are analogous to the bit lines of a RAM. The decoder and the multiplexer are controlled by a



□ **FIGURE 13-1**
Keyboard Scan Matrix

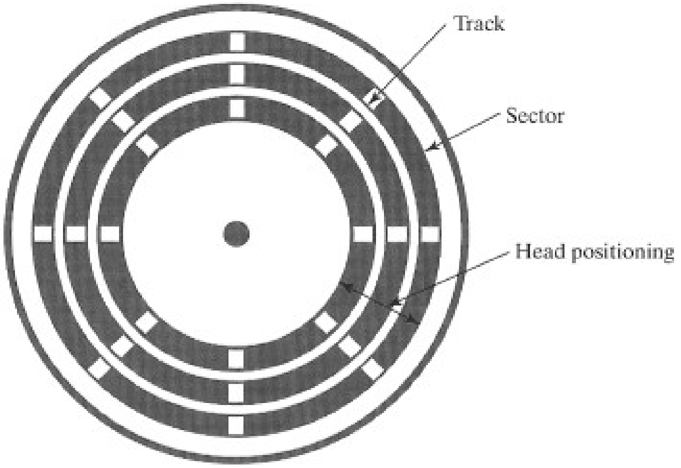
microcontroller, a tiny computer that contains RAM, ROM, a timer, and simple I/O interfaces.

The microcontroller is programmed to periodically scan all intersections in the matrix by manipulating the control inputs of the decoder and multiplexer. If the key is depressed at an intersection, a signal path is closed from an output of the *X* decoder to an input of the *Y* multiplexer. The existence of this path is sensed at an input to the microcontroller. The 7-bit control code applied to the decoder and multiplexer at the time identifies the key. To allow for "rollover" in typing, in which multiple keys are depressed before any of them is released, the microcontroller actually identifies the depressing and release of the keys. Whether a key is depressed or released, the control code at the time of the event is sensed and is translated by the microcontroller into a *K-scan code*. When a key is depressed, a *make code* is produced; when a key is released, a *break code* is produced. Thus, there are two codes for each key, one for when the key is depressed and one for when the key is released. Note that the scanning of the entire keyboard occurs hundreds of times per second, so there is no danger of missing any depression or release of a key.

After presenting a number of I/O interface concepts, we will revisit the keyboard to see what happens to the K-scan codes before they are finally translated to ASCII characters.

Hard Disk

The hard disk is the primary intermediate-speed, nonvolatile, writable storage medium for most computers. The typical hard drive stores information serially on a nonremovable disk with a few to many platters, as shown in the upper right of the generic computer at the beginning of Chapter 1. Each platter is magnetizable on one or both surfaces. There are one or more read/write *heads* per recording surface; for the remainder of our discussion, we will assume a single head per surface. Each platter is divided into concentric *tracks*, as illustrated in Figure 13-2. The set of tracks that are at the same distance from the center of the disk on all platter surfaces is referred to as a *cylinder*. Each track is divided into *sectors* containing a fixed



□ **FIGURE 13-2**
Hard Disk Format

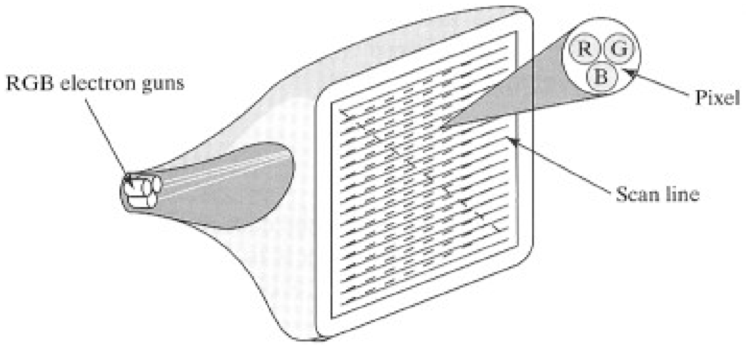
number of bytes. The number of bytes per sector typically ranges from 256 to 4K. The typical byte address includes the cylinder number, head number, sector number, and word offset within the sector. The addressing assumes that the number of sectors per track is fixed. In modern, high-capacity disks, more sectors are included in the longer outer tracks than in the shorter inner tracks. In addition, a number of spare sectors are reserved to take the place of defective sectors. As a consequence of these design choices, the actual physical address of a sector on the disk is likely to be different from the address of the sector sent to the disk controller. The mapping from this address to the physical address is typically accomplished in the disk controller or drive electronics.

To enable information to be accessed, the set of heads is mounted on an actuator that can move the heads radially over the disk, as shown in the generic computer drawing. The time required to move the heads from the current cylinder to the desired cylinder is called the *seek time*. The time required to rotate the disk from its current position to that having the desired sector under the heads is called the *rotational delay*. In addition, a certain amount of time is required by the disk controller to access and output information. This time is the *controller time*. The time required to locate a word on the disk is the *disk access time*, which is the sum of the controller time, the seek time, and the rotational delay. Average values over all possibilities are used for these four parameters. Words may be transferred singly, but as we will see in Chapter 14, they are often accessed in blocks. The transfer rate for a block of words, once the block has been located, is the *disk transfer rate*, typically specified in megabytes/second (MB/s). The transfer rate required by the CPU-memory bus to transfer a sector from disk is the number of bytes in the sector divided by the length of time taken to read a sector from the disk. The length of time required to read a sector is equal to the proportion of the cylinder occupied by the sector divided by the rotational speed of the disk. For example, with 63 sectors, 512 B per sector, a rotational speed of 5400 rpm, and allowance for the gap between sectors, this time is about 0.15 ms, giving a transfer rate of 512/0.15 ms = 3.4 MB/s. The controller will store the information read from the sector in its memory. The sum of the disk access time and the disk transfer rate times the number of bytes per sector gives an estimate of the time required to transfer the information in a sector to or from the hard disk. Typical values in the mid-1990s are a seek time of 10 ms, a rotational delay of 6 ms, a sector transfer time of 0.15 ms, and a negligible controller time, giving an access time for an isolated sector of 16.15 ms.

Graphics Display

The graphics display is the primary output device for the interactive use of computers. Displays use a number of different technologies, the most prevalent of which is currently the cathode-ray tube (CRT), illustrated in Figure 13-3. The most modern versions of the CRT display are based on analog signals, which are generated on the display adapter board. The display is defined in terms of picture elements called *pixels*. The color display has three locations associated with each pixel on the screen. These locations correspond to the primary colors red, green, and blue (RGB). At each location, there is the corresponding colored phosphor. A phosphor emits light of its color when excited by a beam of electrons. In order to excite the three phosphors simultaneously, three electron guns are used, one for red, one for green, and one for blue—hence the RGB electron guns shown in the figure. The color that results for a given pixel is determined by the intensity of the electron beams striking the phosphors within the pixel.

The electron beams are scanned across the screen to form a set of horizontal lines called *scan lines*. This set of lines is referred to as a *raster*. The lines are scanned from top to bottom, beginning at the upper left and ending at the lower right. The electron guns remain at zero intensity as they scan from right to left in preparation for drawing the next scan line. The resolution of the information displayed is given in terms of the number of pixels per scan line and the number of



□ **FIGURE 13-3**
CRT Display

scan lines in the raster. A high-resolution super video graphics array (SVGA) display may have as many as 1280 pixels per scan line and 1024 lines in the raster. The electron beams scan the entire raster in 1/60 of a second.

Each of the pixels is controlled by the display adapter. A typical adapter uses a byte to define the color of a pixel. Since the byte contains 8 bits, it can define 256 colors at any given time. The byte does not directly drive the display, but instead selects 1 out of 256 registers in the graphics adapter to define the color. Each register is 20 bits or more, so the 256 colors can be selected from over 1 million colors by defining the contents of the registers.

Typically, the display adapter has video RAM that stores all of the bytes which control the display pixels. For a high-resolution display with 1280 pixels per scan line and 1024 scan lines, the number of pixels is 1280 × 1024 = 1,310,720. So, for 256 colors, a single screen of information requires at least 1.25 MB of video RAM.

I/O Transfer Rates

An indicated earlier, the three peripheral devices discussed in this section give a sense of the range of peak I/O transfer rates. The keyboard data transfer rate is less than 10 bytes/s. For the hard disk, while the disk controller is capturing the data arriving rapidly from the disk in the sector buffer, the transfer of data from the buffer to main memory is impossible. Thus, in the case in which the next sector is to be read immediately, all of the data from the sector buffer needs to be stored in main memory during the time the gap on the disk between the sectors passes under the disk head. For 63 sectors and a rotational speed of 5400 rpm, this time is about 25 μs. Thus, the peak transfer rate required is 512B/25 ms = 20 MB/s. For a display with 256 colors, if a screen is to be changed entirely every 1/60 of a second, 1.25 MB of data must be delivered to the video RAM from the CPU in that amount of time. This requires a data rate of 1.25 MB × 60 = 75 MB/s.

Based on the preceding, we can conclude that the peak data rates required by the particular peripherals we have considered have a wide range. The rates for the hard disk and the display are high enough compared to the maximum rate of transfer on the computer buses to provide a challenge to designers. Attempts to meet this challenge use techniques in the disk controller and the graphics adapter to reduce the peak transfer rates required and use fast bus designs between the peripheral interfaces and memory.

13-3 I/O Interfaces

Peripherals connected to a computer need special communication links to interface them with the CPU. The purpose of these links is to resolve the differences in the properties of the CPU and memory and the properties of each peripheral. The major differences are as follows:

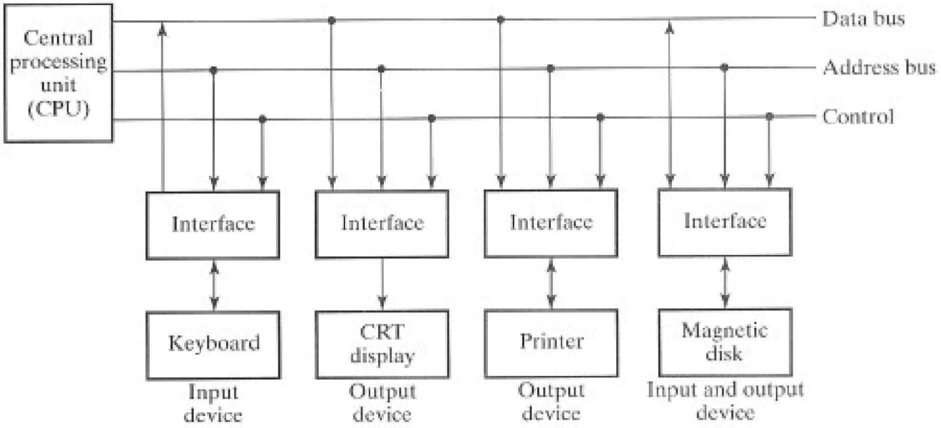
1. Peripherals are often electromechanical devices whose manner of operation is different from that of the CPU and memory, which are electronic devices. Therefore, a conversion of signal values may be required.

2. The data transfer rate of peripherals is usually different from the clock rate of the CPU. Consequently, a synchronization mechanism may be needed.
3. Data codes and formats in peripherals differ from the word format in the CPU and memory.
4. The operating modes of peripherals differ from each other, and each must be controlled in a way that does not disturb the operation of other peripherals connected to the CPU.

To resolve these differences, computer systems include special hardware components between the CPU and the peripherals to supervise and synchronize all input and output transfers. These components are called *interface units*, because they interface between the bus from the CPU and the peripheral device. In addition, each device has its own controller to supervise the operations of the particular mechanism of that peripheral. For example, the controller in a printer attached to a computer controls the motion of the paper, the timing of the printing, and the selection of the characters to be printed.

I/O Bus and Interface Unit

A typical communication structure between the CPU and several peripherals is shown in Figure 13-4. Each peripheral has an interface unit associated with it. The common bus from the CPU is attached to all peripheral interfaces. To communicate with a particular device, the CPU places a device address on the address bus. Each interface attached to the common bus contains an address decoder that monitors the address lines. When the interface detects its own address, it activates the path between the bus lines and the device that it controls. All peripherals with addresses that do not correspond to the address on the bus are disabled by their interface. At the same time that the address is made available on the address bus, the CPU provides a function



□ **FIGURE 13-4**
Connection of I/O Devices to CPU

code on the control lines. The selected interface responds to the function code and proceeds to execute it. If data must be transferred, the interface communicates with both the device and the CPU data bus to synchronize the transfer.

In addition to communicating with the I/O devices, the CPU of a computer must communicate with the memory unit through an address and data bus. There are three ways that external computer buses communicate with memory and I/O. One method uses common data, address, and control buses for both memory and I/O. We have referred to this configuration as *memory-mapped I/O*. The common address space is shared between the interface units and memory words, each having distinct addresses. Computers that adopt the memory-mapped scheme read and write from interface units as if they were assigned memory addresses by using the same instructions that read from and write to memory.

The second alternative is to share a common address bus and data bus, but use different control lines for memory and I/O. Such computers have separate read and write lines for memory and I/O. To read or write from memory, the CPU activates the memory read or memory write control. To perform input to or output from an interface, the CPU activates the read I/O or write I/O control, using special instructions. In this way, the addresses assigned to memory and I/O interface units are independent from each other and are distinguished by separate control lines. This method is referred to as the *isolated I/O configuration*.

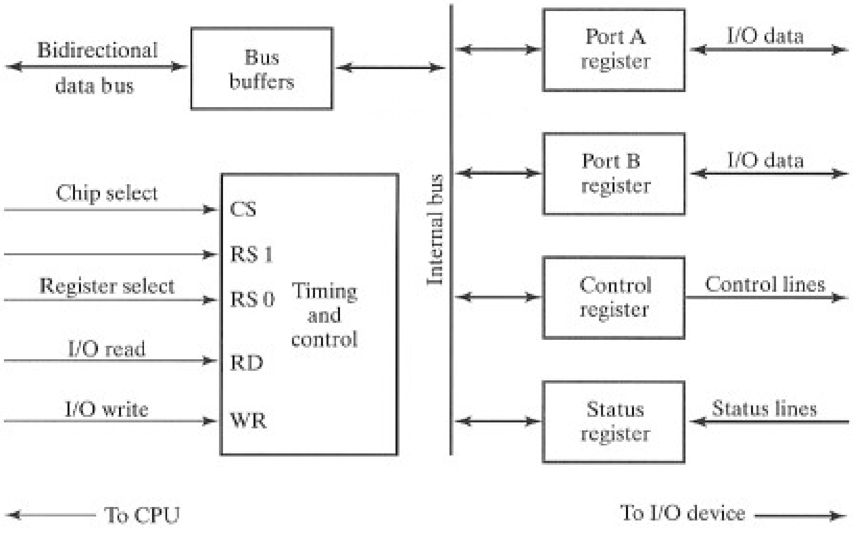
The third alternative is to have two independent sets of data, address, and control buses. This is possible in computers that include an *I/O processor* in the system in addition to the CPU. The memory communicates with both the CPU and I/O processor through a common memory bus. The I/O processor communicates with the input and output devices through separate address, data, and control lines. The purpose of the I/O processor is to provide an independent pathway for the transfer of information between external devices and internal memory. The I/O processor is sometimes called a *data channel*.

Example of I/O Interface

A typical I/O interface unit is shown in block diagram form in Figure 13-5. It consists of two data registers called *ports*, a control register, a status register, a bidirectional data bus, and timing and control circuits. The function of the interface is to translate the signals between the CPU buses and the I/O device and to provide the needed hardware to satisfy the two sets of timing constraints.

The I/O data from the device can be transferred into either port A or port B. The interface may operate with an output device, with an input device, or with a device that requires both input and output. If the interface is connected to a printer, it will only output data; if it services a scanner, it will only input data. A hard disk transfers data in both directions, but not at the same time; so the interface needs only one set of I/O bidirectional data lines.

The *control register* receives control information from the CPU. By loading appropriate bits into this register, the interface and the device can be placed in a variety of operating modes. For example, port A may be defined as an input port only. A magnetic tape unit may be instructed to rewind the tape or to start the tape moving in



CS	RS1	RS0	Register selected
0	x	x	None: data bus in high-impedance state
1	0	0	Port A register
1	0	1	Port B register
1	1	0	Control register
1	1	1	Status register

□ **FIGURE 13-5**
Example of I/O Interface Unit

the forward direction. The bits in the status register are used for status conditions and for recording errors that may occur during data transfer. For example, a status bit may indicate that port A has received a new data item from the device, while another bit in the status register may indicate that a parity error has occurred during the transfer.

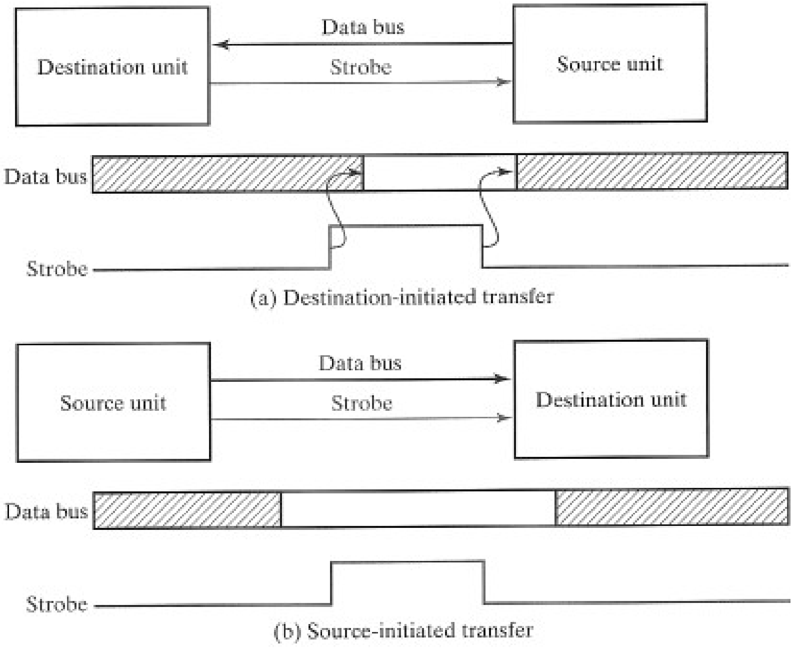
The interface registers communicate with the CPU through the bidirectional data bus. The address bus selects the interface unit through the chip select input and the two register select inputs. A circuit (usually a decoder or a gate) detects the address assigned to the interface registers. This circuit enables the chip select (CS) input when the interface is selected by the address bus. The two *register select inputs* $RS1$ and $RS0$ are usually connected to the two least significant lines of the address bus. These two inputs select one of the four registers in the interface, as specified in the table accompanying the diagram in Figure 13-5. The contents of the selected register are transferred into the CPU via the data bus when the I/O read signal is enabled. The CPU transfers binary information into the selected register via the data bus when the I/O write input is enabled.

The CPU, interface, and I/O device are likely to have different clocks that are not synchronized with each other. Thus, these units are said to be *asynchronous* with respect to each other. Asynchronous data transfer between two independent units requires that control signals be transmitted between the units to indicate the time at which data is being transmitted. In the case of CPU-to-interface communication, control signals must also indicate the time at which the address is valid. We will look at two methods for performing this timing: strobing, as it is called, and handshaking. Initially, we will consider generic cases in which no addresses are involved; subsequently, we will add addressing. The communicating units for the generic case will be referred to as the source unit and destination unit.

Strobing

Data transfers using *strobing* are shown in Figure 13-6. The data bus between the two units is assumed to be made bidirectional by the use of three-state buffers.

The transfer in Figure 13-6(a) is initiated by the destination unit. In the shaded area of the data signal, the data is invalid. Also, a change in Strobe at the tail of each arrow causes a change on the data bus at the head of the arrow. The destination unit changes the Strobe from 0 to 1. When the value 1 on Strobe reaches the source unit, the unit responds by placing the data on the data bus. The destination unit expects the data to be available, at worst, a fixed amount of time after Strobe goes to 1. At that time, the destination unit captures the data in a register and changes Strobe from 1 to 0. In response to the 0 value on Strobe, the source unit removes the data from the bus.



(a) Destination-initiated transfer

(b) Source-initiated transfer

□ **FIGURE 13-6**
Asynchronous Transfer Using Strobing

The transfer in Figure 13-6(b) is initiated by the source unit. In this case, the source unit places the data on the data bus. After the short time required for the data to settle on the bus, the source unit changes Strobe from 0 to 1. In response to Strobe equal to 1, the destination unit sets up the transfer to one of its registers. The source then changes Strobe from 1 to 0, which triggers the transfer into the register at the destination. Finally, after a short time required to ensure that the register transfer is done, the source removes the data from the data bus, completing the transfer.

Although simple, the strobe method of transferring data has several disadvantages. First, when the source unit initiates the transfer, there is no indication to it that the data was ever captured by the destination unit. It is possible, due to a hardware failure, that the destination unit did not receive the change in Strobe. Second, when the destination unit performs the transfer, there is no indication to it that the source has actually placed the data on the bus. Thus, the destination unit could be reading arbitrary values from the bus rather than actual data. Finally, the speeds at which the various units respond may vary. If there are multiple units, the unit initiating a transfer must wait for the delay of the slowest of the attached communicating units before changing Strobe to 0. Thus, the time taken for every transfer is determined by the slowest unit with which a given unit initiates transfers.

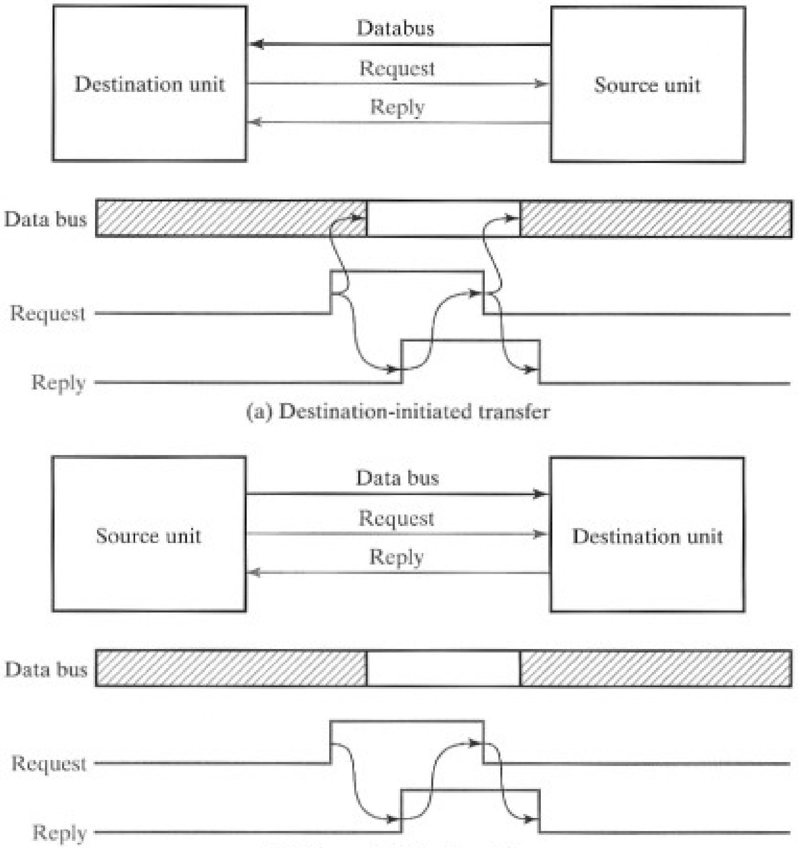
Handshaking

The *handshaking* method uses two control signals to deal with the timing of transfers. In addition to the signal from the unit initiating the transfer, there is a second control signal from the other unit involved in the transfer.

The basic principle of a two-signal handshaking procedure for data transfer is as follows. One control line from the initiating unit is used to request a response from the other unit. The second control line from the other unit is used to reply to the initiating unit that the response is occurring. In this way, each unit informs the other of its status, and the result is an orderly transfer through the bus.

Figure 13-7 shows data transfer procedures using handshaking. In Figure 13-7(a), the transfer is initiated by the destination unit. The two handshaking lines are called Request and Reply. The initial state is when both Request and Reply are disabled and in the 00 state. The subsequent states are 10, 11, and 01. The destination unit initiates the transfer by enabling Request. The source unit responds by placing the data on the bus. After a short time for settling of the data on the bus, the source unit activates Reply to signal the presence of the data. In response to Reply, the destination unit captures the data in a register and disables Request. The source unit then disables Reply and the system goes to the initial state. The destination unit may not make another request until the source unit has shown its readiness to provide new data by disabling Reply. Figure 13-7(b) represents handshaking for the source-initiated transfer. In this case, the source controls the interval between when the data is applied and when Request changes to 1 and between when Request changes to 0 and when the data is removed.

The handshaking scheme provides a high degree of flexibility and reliability, because the successful completion of a data transfer relies on active participation by both units. If one unit is faulty, the data transfer will not be completed. Such an



(a) Destination-initiated transfer

(b) Source-initiated transfer

□ **FIGURE 13-7**
Asynchronous Transfer Using Handshaking

error can be detected by means of a time-out mechanism, which produces an alarm if the data transfer is not completed within a predetermined time interval. The time-out is implemented by means of an internal clock that starts counting time when the unit enables one of its handshaking control signals. If the return handshake does not occur within a given period, the unit assumes that an error occurred. The time-out signal can be used to interrupt the CPU and execute a service routine that takes appropriate error recovery action. Also, the timing is controlled by both units, not just the initiating unit. Within the time-out limits, the response of each unit to a change in the control signal of the other unit can take an arbitrary amount of time, and the transfer will still be successful.

The examples of transfers in Figure 13-6 and Figure 13-7 represent transfers between an interface and an I/O device and between a CPU and an interface. In the latter case, however, an address will be necessary to select the interface with which the CPU wishes to communicate and a register within the interface. In order to ensure that the CPU addresses the correct interface, the address must have settled on the address bus before the Strobe or Request signal changes from 0 to 1. Further, the address must remain stable until the change in the strobe or request from 1 to 0 has settled to 0 at the interface logic. If either of these conditions is violated, another interface may be falsely activated, causing an incorrect data transfer.

13-4 Serial Communication

The transfer of data between two units may be parallel or serial. In parallel data transfer, each bit of the message has its own path, and the entire message is transmitted at one time. This means that an n-bit message is transmitted in parallel through n separate conductor paths. In serial data transmission, each bit in the message is sent in sequence, one at a time. This method requires the use of one or two signal lines. Parallel transmission is faster, but requires many wires. It is used for short distances and when speed is important. Serial transmission is slower, but less expensive, since it requires only one conductor.

One way that computers and terminals that are remote from each other are connected is via telephone lines. Since telephone lines were originally designed for voice communication, but computers communicate in terms of digital signals, some form of conversion is needed. The devices that do the conversion are called *data sets* or *modems* (modulator-demodulators). A modem converts digital signals into audio tones to be transmitted over telephone lines and also converts audio tones from the line to digital signals for use by a computer. There are various modulation schemes, as well as several different grades of communication media and transmission speeds. Serial data can be transmitted between two points in three different modes: simplex, half duplex, or full duplex. A *simplex* line carries information in one direction only. This mode is seldom used in data communication, because the receiver cannot communicate with the transmitter to indicate whether errors have occurred. Examples of simplex transmission are radio and television broadcasting.

A *half-duplex* transmission system is a system that is capable of transmitting in both directions, but in only one direction at a time. A pair of wires is needed for this mode. A common situation is for one modem to act as the transmitter and the other as the receiver. When transmission in one direction is completed, the roles of the modems are reversed to enable transmission in the opposite direction. The time required to switch a half-duplex line from one direction to the other is called the *turnaround time.*

A *full-duplex* transmission system can send and receive data in both directions simultaneously. This can be achieved by means of a two-wire plus ground link, with a different wire dedicated to each direction of transmission. Alternatively, a single-wire circuit can support full-duplex communication if the frequency spectrum is subdivided into two nonoverlapping frequency bands to create separate receiving and transmitting channels in the same physical pair of wires.

The serial transmission of data can be synchronous or asynchronous. In *synchronous transmission*, the two units share a common clock frequency, and bits are transmitted continuously at that frequency. In long-distance serial transmission, the transmitter and receiver units are each driven by separate clocks of the same frequency. Synchronization signals are transmitted periodically between the two units to keep their clock frequencies in step with each other. In *asynchronous* transmission, binary information is sent only when it is available, and the line remains idle when there is no information to be transmitted. This is in contrast to synchronous transmission, in which bits must be transmitted continuously to keep the clock frequencies in both units synchronized.

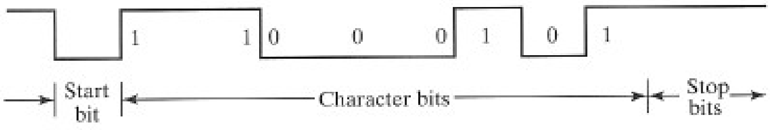
Asynchronous Transmission

One of the most common applications of serial transmission is the communication of one computer with another via modems connected through the telephone system. Each character consists of an alphanumeric code of eight bits, with additional bits inserted at both ends of the code. In asynchronous serial transmission, each character consists of three parts: the start bit, the character bits, and the stop bits. The convention is for the transmitter to rest at the 1 state when no characters are transmitted. The first bit, called the start bit, is always 0 and is used to indicate the beginning of a character. An example of this format is shown in Figure 13-8.

A transmitted character can be detected by the receiver by applying the transmission rules. When a character is not being sent, the line is kept in the 1 state. The initiation of transmission is detected from the start bit, which is always 0. The character bits always follow the start bit. After the last bit of the character is transmitted, a stop bit is detected when the line returns to the 1 state for at least the time taken to transmit one bit. By means of these rules, the receiver can detect the start bit when the line goes from 1 to 0. By using a clock, the receiver examines the line at appropriate times to determine the bit values. The receiver knows the transfer rate of the bits and the number of character bits to accept.

After the character bits are transmitted, one or two stop bits are sent. The stop bits are always in the 1 state and frame the end of character to signify the idle or wait state. These bits allow both the transmitter and the receiver to resynchronize. The length of time that the line stays in the 1 state depends on the amount of time required for the equipment to resynchronize. Some older electromechanical terminals use two stop bits, but newer equipment often uses just one. The line remains in the 1 state until another character is transmitted. The stop time ensures that a new character will not follow for the time taken to transmit one or two bits.

As an illustration, consider serial transmission with a transfer rate of 10 characters per second. Suppose that each transmitted character consists of a start bit,



□ **FIGURE 13-8**
Format of Asynchronous Serial Transfer of Data

8 character bits, and 2 stop bits, for a total of 11 bits. If the bits are transmitted at a rate of 10 bits per second, then each bit takes 0.1 second for transfer. Since there are 11 bits to be transmitted, it follows that the *bit time* is 9.09 msec. The *baud rate* is defined as the maximum number of changes per second in the signal being transmitted. This is often, but not always, equivalent to the rate of data transfer in bits per second. Ten characters per second with an 11-bit format has a transfer rate of 110 baud.

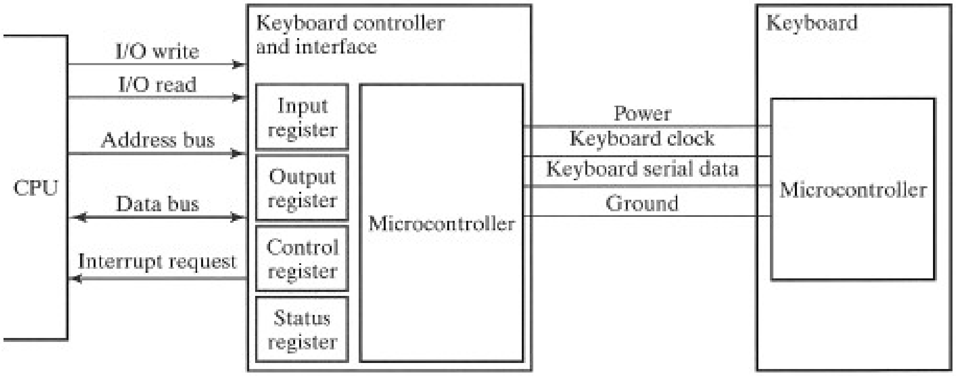
Synchronous Transmission

Synchronous transmission does not use start or stop bits to frame characters. The modems employed in synchronous transmission have internal clocks that are set to the frequency at which bits are being transmitted. For proper operation, it is required that the clocks of the transmitter and receiver modems remain synchronized at all times. The communication line, however, carries only the data bits, from which information on the clock frequency must be extracted. Frequency synchronization is achieved by the receiving modem from the signal transitions that occur in the data that is received. Any frequency shift that may occur between the transmitter and receiver clocks is continuously adjusted by maintaining the receiver clock at the frequency of the incoming bit stream. In this way, the same rate is maintained in both the transmitter and the receiver.

Contrary to asynchronous transmission, in which each character can be sent separately with its own start and stop bits, synchronous transmission must send a continuous message in order to maintain synchronism. The message consists of a group of bits that form a block of data. The entire block is transmitted with special control bits at the beginning and the end, in order to frame the block into one unit of information.

The Keyboard Revisited

To this point, we have covered the basic nature of the I/O interface and serial transmission. With these two concepts available, we are now ready to continue with the example of the keyboard and its interface, as shown in Figure 13-9. The K-scan code produced by the keyboard microcontroller is to be transferred serially from the keyboard through the keyboard cable to the keyboard controller in the computer. The serial transfer on the Keyboard serial data line uses a format just like that shown for asynchronous transfer in Figure 13-8. In this case, however, a signal Keyboard clock is also sent through the cable. Thus, the transmission is synchronous with a transmitted clock signal, rather than asynchronous. These same signals are used to transmit control commands to the keyboard. In the keyboard controller, the microcontroller converts the K-scan code to a more standard *scan code*, which it then places in the Input register, at the same time sending an interrupt signal to the CPU indicating that a key has been pressed and a code is available. The interrupt-handling routine reads the scan code from the input register into a special area in memory. This area is manipulated by software stored in the Basic



□ **FIGURE 13-9**
Keyboard Controller and Interface

Input/Output System (BIOS) that can translate the scan code into an ASCII character code for use by applications.

The Output register in the interface receives data from the CPU. The data can be passed on to control the keyboard—for example, setting the repetition rate when a key is held down. The Control register is used for commands to the keyboard controller. Finally, the Status register reports specific information on the status of the keyboard and the keyboard controller.

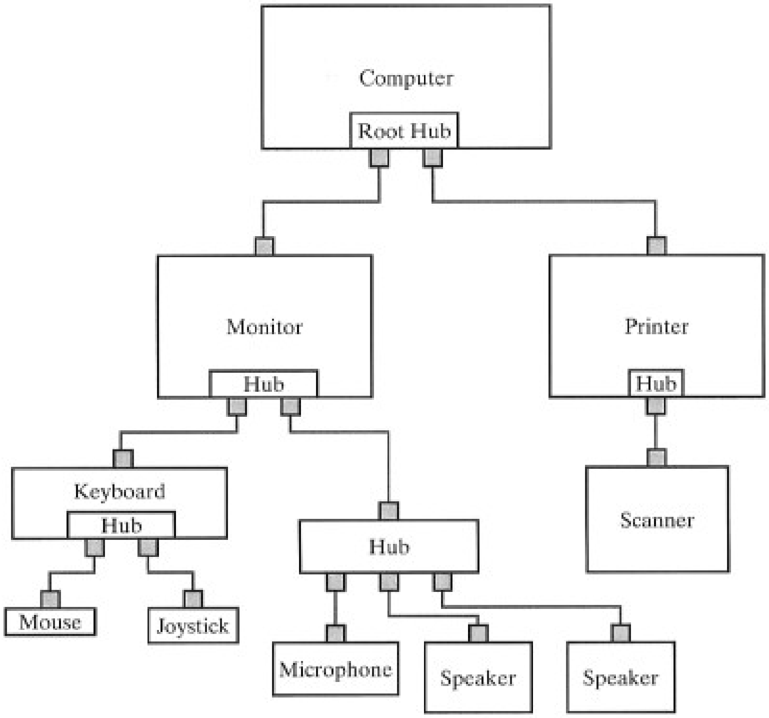
Perhaps one of the most interesting aspects of keyboard I/O is its high complexity. It involves two microcontrollers executing different programs, plus the main processor executing BIOS software (i.e., three different computers executing three distinct programs).

A Packet-Based Serial I/O Bus

Serial I/O, as described for the keyboard, uses a serial cable specifically dedicated to communicating between the computer and the keyboard. Whether parallel or serial, external I/O connections are typically dedicated. The use of these dedicated paths often requires that the computer case be opened and cards inserted with electronics and connectors specific to the particular I/O standard used for a given I/O device.

In contrast, packet-based serial I/O permits many different external I/O devices to use a shared communication structure that is attached to the computer through just one or two connectors. The types of devices supported include keyboards, mice, joysticks, printers, scanners, and speakers. The particular packet-based serial I/O we will describe here is the Universal Serial Bus (USB), which is becoming commonplace as the connection approach of choice for slow- to medium-speed I/O devices.

The interconnection of I/O devices by using USB is shown in Figure 13-10. The computer and attached devices can be classified as hubs, devices, or compound devices. A hub provides attachment points for USB devices and other hubs. A hub

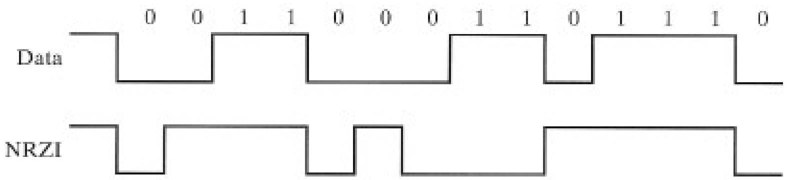


□ **FIGURE 13-10**
I/O Device Connection Using the Universal Serial Bus (USB)

contains a USB interface for control and status handling and a repeater for transferring information through the hub.

The computer contains a USB controller and the root hub. Additional hubs may be a part of the USB I/O structure. If a hub is combined with a device such as the keyboard shown in Figure 13-10, then the keyboard is referred to as a *compound device*. Aside from such compound devices, a USB device contains only one USB port to serve its function alone. The scanner is an example of a regular USB device. Without USB, the monitor, keyboard, mouse, joystick, microphone, speakers, printer, and scanner shown would all have separate I/O connections directly on the computer. The monitor, printer, scanner, microphone, and speakers might all require special cards to be inserted as discussed previously. With USB, only two connections are required.

The USB cables contains four wires: ground, power, and two data lines (D+ and D–) used for differential signaling. The power wire is used to provide small amounts of power to devices such as keyboards so that they do not need to have their own power supplies. To provide immunity to signal variation and noise, 0's and 1's are transmitted by using the difference in voltage between D+ and D–. If the voltage on D+ exceeds the voltage on D– by 200 millivolts or more, then the logic value is a 1. If the voltage on D– exceeds the voltage on D+ by 200 millivolts



□ **FIGURE 13-11**
Non-Return-to-Zero Inverted Data Representation

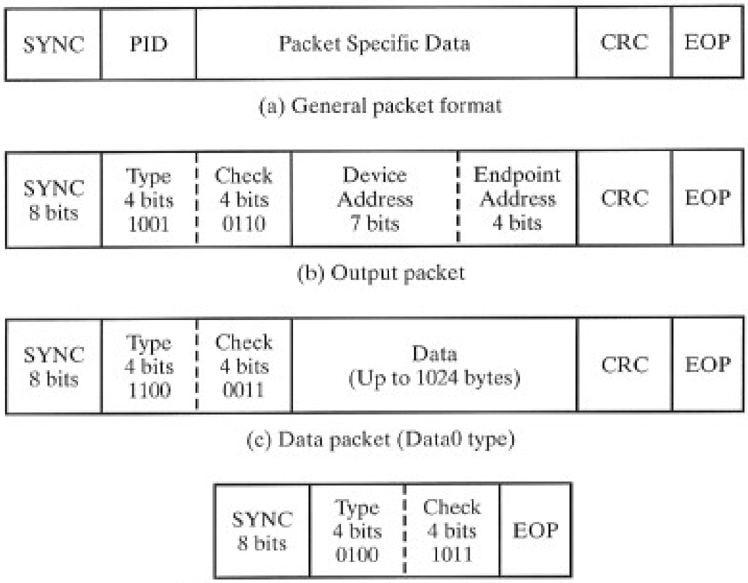
or more, the logic value is a 0. Other voltage relationships between D+ and D– are used as special signal states as well.

The logic values used for signalling are not the actual logic values of the information being transmitted. Instead, a Non-Return-to-Zero Inverted (NRZI) signalling convention is used. A zero in the data being transmitted is represented by a transition from 1 to 0 or 0 to 1 and a 1 is represented by a fixed value of 1 or 0. The relationship between the data being transmitted and the NRZI representation is illustrated in Figure 13-11. As is typical for I/O devices, there is no common clock serving both the computer and the device. NRZI encoding of the data provides edges that can be used to maintain synchronization between the arriving data and the time at which each bit is sampled at the receiver. If there are a large number of 1's in series in the data, there will be no transitions for some time in the NRZI encoding. To prevent loss of synchronization, a 0 is "stuffed" in before every seventh bit position in a string of 1's prior to NRZI encoding so that no more than six 1's appear in series. The receiver must be able to remove these extra zeros when converting NRZI data to normal data.

USB information is transmitted in packets. Each packet contains a specific set of fields depending on the packet type. Logical strings of packets are used to compose USB transactions. For example, an output transaction consists of an Out packet followed by a Data packet and a Handshake packet. The Out packet comes from the USB controller in the computer and notifies the device that it is to receive data. The computer then sends the Data packet. If the Data packet is received without error, then the device responds with the Acknowledge Handshake packet. Next, we detail the information contained in each of these packets.

Figure 13-12(a) shows a general format for USB packets and the formats for each of the three packets involved in an output transaction. Note that each packet begins with a synchronization pattern SYNC. This pattern is 00000001. Because of the sequence of zeros, the corresponding NRZI pattern contains seven edges, which provides a pattern to which the receiving clock can be synchronized. Since this pattern is preceded by a specific signal voltage state referred to as Idle, the pattern also signals the beginning of a new packet.

Following the SYNC, each of the packet formats contains 8 bits called the packet identifier (PID). In the PID, the packet type is specified by 4 bits, with an additional 4 bits that are complements of the first 4 to provide an error check on the type. A very large class of type errors will be detected by the repetition of the type as its complement. The type is optionally followed by information specific to the packet,



(a) General packet format

(b) Output packet

(c) Data packet (Data0 type)

(d) Handshake packet (Acknowledge type)

□ **FIGURE 13-12**
USB Packet Formats

which varies depending upon the packet type. Optionally, a CRC field appears next. The CRC pattern consisting of 5 or 16 bits is a Cyclic Redundancy Check pattern. This pattern is calculated at transmission of the packet from the packet-specific data. The same calculation is performed when the data is received. If the CRC pattern does not match the newly calculated pattern, then an error has been detected. In response to the error, the packet can be ignored and retransmitted. In the last field of the packet, an End of Packet (EOP) appears. This consists of D+ and D–, both low for two bit times, followed by the Idle state for a bit time. As its name indicates, this sequence of signal states identifies the end of the current packet. It should be noted that all fields are presented least significant bit first.

Referring to Figure 13-12(b), for the Output packet, the Type and Check fields are followed by a Device Address, an Endpoint Address, and a CRC pattern. The Device Address consists of seven bits and defines the device that is to input data. The Endpoint Address consists of four bits and defines which port of the device is to receive the information in the Data packet to follow. For example, there may be a port for data and one for control on a given device.

For the Data packet, the packet-specific data consists of 0 to 1024 data bytes. Due to the length of the packet, complex errors are more likely, so the CRC pattern is increased in length to 16 bits to improve its error detection capability.

In the Handshake packet, the packet-specific data is empty. The response to the receipt of the data packet is carried by the PID. PID 01001011 is an Acknowledge (ACK) indicating that the packet was received without any errors detected.

Absence of any HANDSHAKE packet when one would normally appear is an indication of an error. PID 01011010 is a No Acknowledge, indicating that the target is temporarily unable to accept or return data. PID 01111000 is a Stall (STALL), indicating that the target is unable to complete the transfer and that software intervention is required to recover from the stall condition.

The preceding concepts illustrate the general principles underlying a packet-based serial I/O bus and are specific to USB. USB supports other packet types and many different kinds of transactions. In addition, the attachment and detachment of devices is sensed and can trigger various software reactions. In general, there is substantial software in the computer to support the details of the control and operation of the Universal Serial Bus.

13-5 Modes of Transfer

Binary information received from an external device is usually stored in memory for later processing. Information transferred from the central computer into an external device originates in the memory. The CPU merely executes the I/O instructions and may accept the data temporarily, but the ultimate source or destination is the memory. Data transfer between the central computer and I/O devices may be handled in a variety of modes, some of which use the CPU as an intermediate path, while others transfer the data directly to and from the memory. Data transfer to and from peripherals may be handled in one of four possible modes:

1. Data transfer under program control.
2. Interrupt-initiated data transfer.
3. Direct memory access transfer.
4. Transfer through an I/O processor.

Program-controlled operations are the result of I/O instructions written in the computer program. Each transfer of data is initiated by an instruction in the program. Usually, the transfer is to and from a CPU register and peripheral. Other instructions are needed to transfer the data to and from the CPU and memory. Transferring data under program control requires constant monitoring of the peripheral by the CPU. Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made. It is up to the programmed instructions executed in the CPU to keep close tabs on everything that is taking place in the interface unit and the external device.

In the program-controlled transfer, the CPU stays in a program loop called a *busy-wait loop* until the I/O unit indicates that it is ready for data transfer. This is a time-consuming process, since it keeps the processor busy needlessly. The loop can be avoided by using the interrupt facility and special commands to inform the interface to issue an interrupt request signal when the data is available from the device. This allows the CPU to proceed to execute another program. The interface, meanwhile, keeps monitoring the device. When the interface determines that the device is ready for data transfer, it generates an interrupt request to the computer.

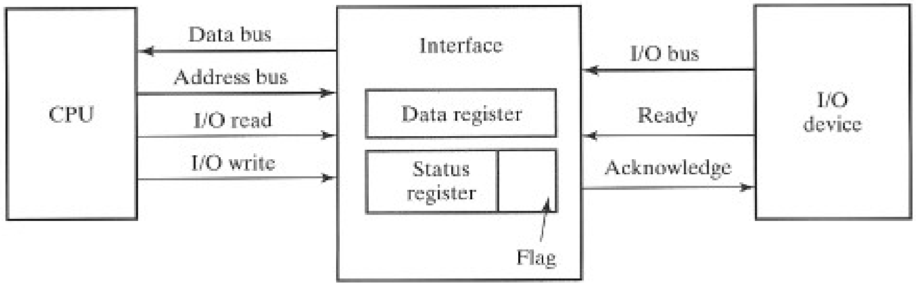
Upon detecting the external interrupt signal, the CPU momentarily stops the task it is performing, branches to a service program to process the data transfer, and then returns to the original task. This interrupt-initiated transfer is the type used for the keyboard controller shown in Figure 13-9.

Transferring of data under program control is performed through the I/O bus and between the CPU and a peripheral interface unit. In *direct memory access* (DMA), the interface unit transfers data into and out of the memory unit through the memory bus. The CPU initiates the transfer by supplying the interface with the starting address and the number of words needing to be transferred and then proceeds to execute other tasks. When the transfer is made, the interface requests memory cycles through the memory bus. When the request is granted by the memory controller, the interface transfers the data directly into memory. The CPU merely delays memory operations to allow the direct memory I/O transfer. Since the speed of a peripheral is usually slower than that of a processor, I/O memory transfers are infrequent compared with processor access to memory. DMA transfer is discussed in more detail in Section 13-7.

Many computers combine the interface logic with the requirements for DMA into one unit called an *I/O processor* (IOP). The IOP can handle many peripherals through a DMA-and-interrupt facility. In such a system, the computer is divided into three separate modules: the memory unit, the CPU, and the IOP. I/O processors are presented in Section 13-8.

Example of Program-Controlled Transfer

A simple example of data transfer from an I/O device through an interface into the CPU is shown in Figure 13-13. The device transfers bytes of data one at a time as they are available. When a byte is available, the device places it on the I/O bus and enables Ready. The interface accepts the byte into its data register and enables Acknowledge. The interface sets a bit in the status register, which we will refer to as a *flag*. The device can now disable Ready, but it will not transfer another byte until Acknowledge is disabled by the interface, according to the handshaking procedure established in Section 13-3.

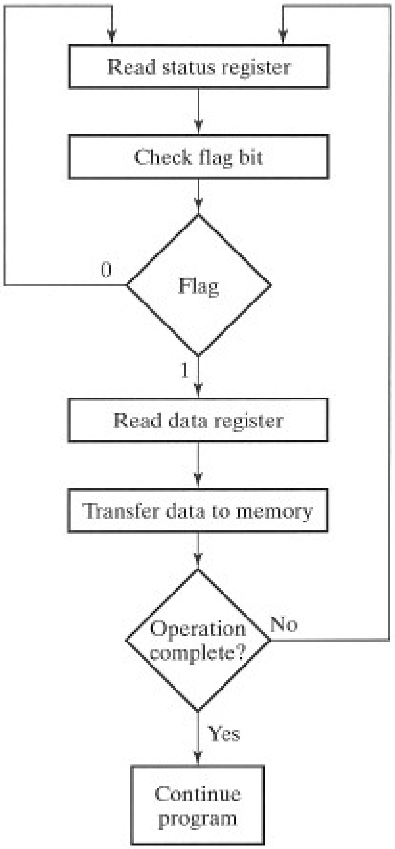


□ **FIGURE 13-13**
Data Transfer from I/O Device to CPU

Under program control, the CPU must check the flag to determine whether there is a new byte in the interface data register. This is done by reading the contents of the status register into a CPU register and checking the value of the flag. If the flag is equal to 1, the CPU reads the data from the data register. The flag is then cleared to 0 either by the CPU or the interface, depending on how the interface circuits are designed. Once the flag is cleared, the interface disables Acknowledge, and the device can transfer the next data byte.

A flowchart of the a program written for the preceding transfer is shown in Figure 13-14. The flowchart assumes that the device is sending a sequence of bytes that must be stored in memory. The program continually examines the status of the interface until the flag is set to 1. Each byte is brought into the CPU and transferred to memory until all of the data have been transferred.

The program-controlled data transfer is used only in systems that are dedicated to monitor a device continuously. The difference in information transfer rate between the CPU and the I/O device makes this type of transfer inefficient.



□ **FIGURE 13-14**
Flowchart for CPU Program to Input Data

To see why, consider a typical computer that can execute the instructions to read the status register and check the flag in 100 ns. Assume that the input device transfers its data at an average rate of 100 bytes/s. This is equivalent to one byte every 10,000 μs, meaning that the CPU will check the flag 100,000 times between each transfer. Thus, the CPU is wasting time checking the flag instead of doing a useful processing task.

Interrupt-Initiated Transfer

An alternative to the CPU constantly monitoring the flag is to let the interface inform the computer when it is ready to transfer data. This mode of transfer uses the interrupt facility. While the CPU is running a program, it does not check the flag. However, when the flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that the flag has been set. The CPU drops what it is doing to take care of the input or output transfer. After the transfer is completed, the computer returns to the previous program to continue what it was doing before the interrupt. The CPU responds to the interrupt signal by storing the return address from the program counter into a memory stack or register, and then control branches to a service routine that processes the required I/O transfer. The way that the processor chooses the branch address of the service routine varies from one unit to another. In principle, there are two methods for accomplishing this: *vectored interrupt* and *nonvectored interrupt*. In a nonvectored interrupt, the branch address is assigned to a fixed location in memory. In a vectored interrupt, the source that interrupts supplies the branch address to the computer. This information is called the *vector address*. In some computers, the vector address is the first address of the service routine; in other computers, the vector address is an address that points to a location in memory where the first address of the service routine is stored. The vectored interrupt procedure was presented in Section 13-9 in conjunction with Figure 13-9.

13-6 Priority Interrupt

A typical computer has a number of I/O devices attached to it that are able to originate an interrupt request. The first task of the interrupt system is to identify the source of the interrupt. There is also the possibility that several sources will request service simultaneously. In this case, the system must decide which device to service first.

A priority interrupt system establishes a priority over the various interrupt sources to determine which interrupt request to service first when two or more arrive simultaneously. The system may also determine which requests are permitted to interrupt the computer while another interrupt is being serviced. Higher levels of priority are assigned to requests that, if delayed or interrupted, could have serious consequences. Devices with high-speed transfers such as magnetic disks are given high priority, and slow devices such as keyboards receive the lowest priority.

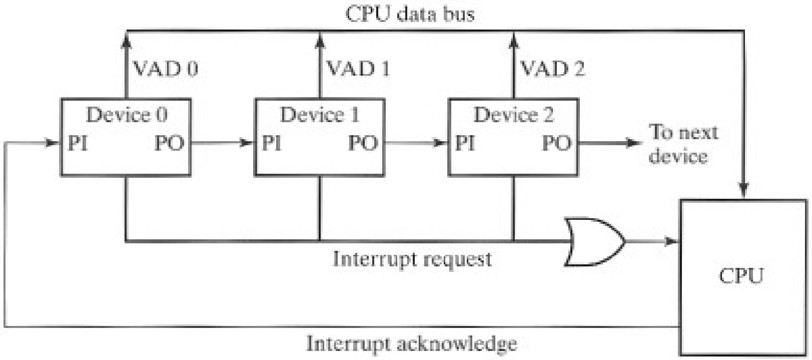
When two devices interrupt the computer at the same time, the computer services the device with the higher priority first.

Establishing the priority of simultaneous interrupts can be done by software or hardware. Software uses a polling procedure to identify the interrupt source of highest priority. In this method, there is one common branch address for all interrupts. The program at the branch address takes care of interrupts by polling the interrupt sources in sequence. The priority of each interrupt source determines the order in which it is polled. The source with the highest priority is tested first, and if its interrupt signal is on, control branches to a routine which services that source. Otherwise, the source with the next lower priority is tested, and so on. Thus, the initial service routine for all interrupts consists of a program that tests the interrupt sources in sequence and branches to one of many other possible service routines. The particular service routine that is reached belongs to the highest priority device among all devices that interrupted the computer. The disadvantage of the software method is that if there are many interrupts, the time required to poll all the sources can exceed the time available to service the I/O device. In this situation, a hardware priority interrupt unit can be used to speed up the operation of the system.

A hardware priority interrupt unit functions as an overall manager in an interrupt system environment. The unit accepts interrupt requests from many sources, determines which of the incoming requests has the highest priority, and issues an interrupt request to the computer based on this determination. To speed up the operation, each interrupt source has its own interrupt vector address to access its own service routine directly. Thus, no polling is required, because all the decisions are made by the hardware priority interrupt unit. The hardware priority function can be established either by a serial or parallel connection of interrupt lines. The serial connection is also known as the daisy chain method.

Daisy Chain Priority

The *daisy chain* method of establishing priority consists of a serial connection of all devices that request an interrupt. The device with the highest priority is placed in the first position, followed by devices of priority in descending order, down to the device with the lowest priority, which is placed last in the chain. This method of connection between three devices and the CPU is shown in Figure 13-15. Interrupt request lines from all devices are ORed to form the interrupt line to the CPU. If any device has its Interrupt request at 1, the interrupt line goes to 1 and enables the interrupt input of the CPU. When no interrupts are pending, the interrupt line stays at 0, and no interrupts are recognized by the CPU. The CPU responds to an interrupt request by enabling Interrupt acknowledge. The signal that is produced is received by device 0 at its *PI* (priority in) input. The signal then passes on to the next device through the *PO* (priority out) output only if device 0 is not requesting an interrupt. If device 0 has a pending interrupt, it blocks the acknowledge signal from the next device by placing a 0 on the *PO* output and proceeds to insert its own interrupt vector address (*VAD*) onto the data bus for the CPU to use during the interrupt cycle.



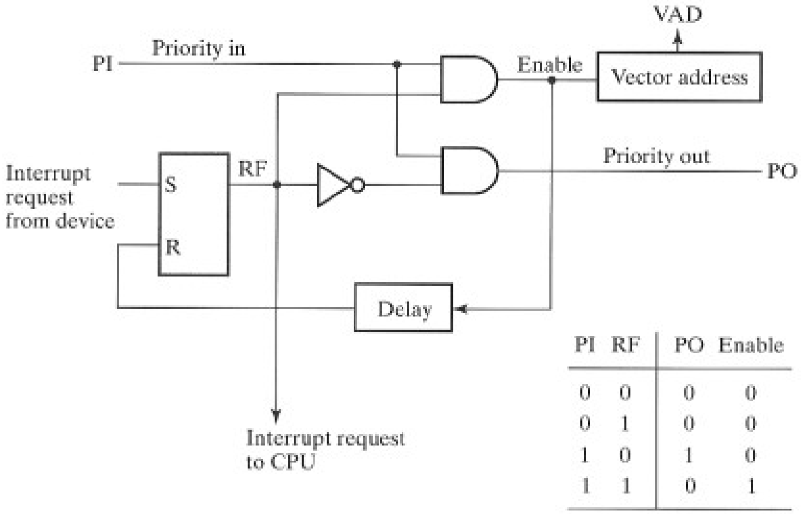
□ **FIGURE 13-15**
Daisy Chain Priority Interrupt

A device with a 0 on its *PI* input generates a 0 on its *PO* output to inform the device with next lower priority that the acknowledge signal has been blocked. A device that is requesting an interrupt and has a 1 on its *PI* input will intercept the acknowledge signal by placing a 0 on its *PO* output. If the device does not have pending interrupts, it transmits the acknowledge signal to the next device by placing a 1 on its *PO* output. Thus, the device with $PI = 1$ and $PO = 0$ is the one with the highest priority that is requesting an interrupt, and this device places its *VAD* on the data bus. The daisy chain arrangement gives the highest priority to the device that receives the Interrupt acknowledge signal from the CPU. The farther the device is from the first position, the lower is its priority.

Figure 13-16 shows the internal logic that must be included within each device connected in the daisy chain scheme. The device sets its *RF* latch when it is about to interrupt the CPU. The output of the latch functionally enters the OR that drives the interrupt line. If $PI = 0$, both *PO* and the enable line to *VAD* are equal to 0, irrespective of the value of *RF*. If $PI = 1$ and $RF = 0$, then $PO = 1$, the vector address is disabled, and the acknowledge signal passes to the next device through *PO*. The device is active when $PI = 1$ and $RF = 1$, which places a 0 on *PO* and enables the vector address onto the data bus. It is assumed that each device has its own distinct vector address. The *RF* latch is reset after a sufficient delay to ensure that the CPU has received the vector address.

Parallel Priority Hardware

The parallel priority interrupt method uses a register with bits set separately by the interrupt signal from each device. Priority is established according to the position of the bits in the register. In addition to the interrupt register, the circuit may include a mask register to control the status of each interrupt request. The mask register can be programmed to disable lower priority interrupts while a higher priority device is being serviced. It can also allow a high-priority device to interrupt the CPU while a lower priority device is being serviced.

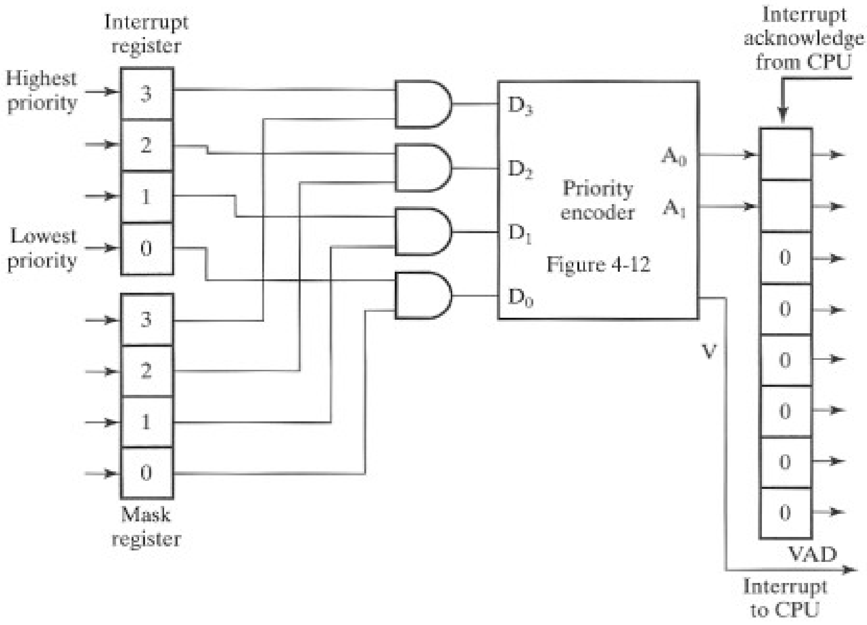


PI	RF	PO	Enable
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

□ **FIGURE 13-16**
One Stage of the Daisy Chain Priority Arrangement

The priority logic for a system with four interrupt sources is shown in Figure 13-17. The logic consists of an interrupt register with individual bits set by external conditions and cleared by program instructions. Interrupt input 3 has the highest priority, input 0 the lowest. The mask register has the same number of bits as the interrupt register. By means of program instructions, it is possible to set or reset any bit in the mask register. Each interrupt bit and its corresponding mask bit are applied to an AND gate to produce the four inputs to a priority encoder. In this way, an interrupt is recognized only if its corresponding mask bit is set to 1 by the program. The priority encoder generates two bits of the vector address, which is transferred to the CPU via the data bus. Output V of the encoder is set to 1 if an interrupt request that is not masked has occurred. This provides the interrupt signal for the CPU.

The priority encoder is a circuit that implements the priority function. The logic of the priority encoder is such that, if two or more inputs are 1 at the same time, the input having the highest priority takes precedence. The circuit of a four-input priority encoder can be found in Section 4-4, and its truth table is listed in Table 4-5. Input D_3 has the highest priority so, regardless of the values of other inputs, when this input is 1, the output is $A_1 A_0 = 11$. D_2 has the next lower priority. The output is 10 if $D_2 = 1$, provided that $D_3 = 0$, regardless of the values of the other two lower priority inputs. The output is 01 when $D_1 = 1$, provided that the two higher priority inputs are equal to 0, and so on down the priority levels. The interrupt output labeled V is equal to 1 when one or more inputs are equal to 1. If all inputs are 0, V is 0, and the other two outputs of the encoder are not used. This is because the vector address is not transferred to the CPU when $V = 0$.



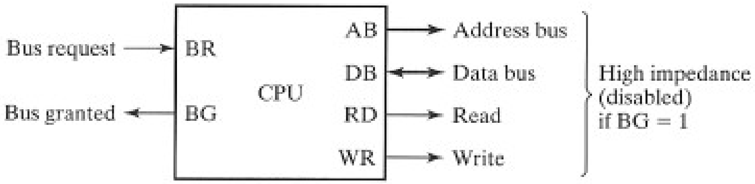
□ **FIGURE 13-17**
Parallel Priority Interrupt Hardware

The output of the priority encoder is used to form part of the vector address of the interrupt source. The other bits of the vector address can be assigned any values. For example, the vector address can be found by appending six zeros to the outputs of the encoder. With this choice, the interrupt vectors for the four I/O devices are assigned the 8-bit binary numbers equivalent to decimal 0, 1, 2, and 3.

13-7 Direct Memory Access

The transfer of blocks of information between a fast storage device such as magnetic disk and the CPU can preoccupy the CPU and permit little, if any, other processing to be accomplished. Removing the CPU from the path and letting the peripheral device manage the memory buses directly will relieve the CPU from many I/O operations and allow it to proceed with other processing. In this transfer technique, called direct memory access (DMA), the DMA controller takes over the buses to manage the transfer directly between the I/O device and memory. As a consequence, the CPU is temporarily deprived of access to memory and control of the memory buses.

DMA may capture the buses in a number of ways. One common method extensively used in microprocessors is to disable the buses through special control signals. Figure 13-18 shows two control signals in a CPU that facilitate the DMA transfer. The bus request (*BR*) input is used by the DMA controller to request the



□ **FIGURE 13-18**
CPU Bus Control Signals

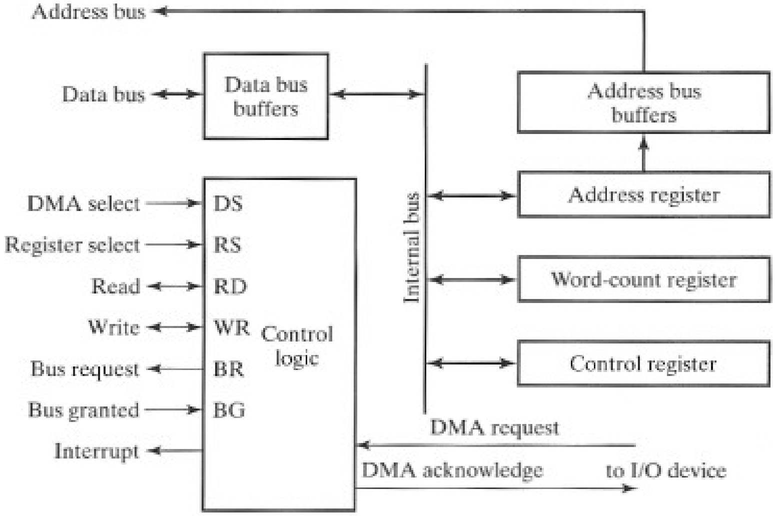
CPU to relinquish control of the buses. When *BR* input is active, the CPU places the address bus, the data bus, and the read and write lines into a high-impedance state. After this is done, the CPU activates the bus granted (*BG*) output to inform the external DMA that it can take control of the buses. As long as the *BG* line is active, the CPU is unable to proceed with any operations requiring access to the buses. When the bus request input is disabled by the DMA, the CPU returns to its normal operation, disables the *BG* output, and takes control of the buses.

When the *BG* line is enabled, the external DMA controller takes control of the bus system in order to communicate directly with memory. The transfer can be made for an entire block of memory words, suspending operation of the CPU until the entire block is transferred, a process referred to as *burst transfer*. Or the transfer can be made one word at a time between executions of CPU instructions, a process called *single-cycle transfer* or *cycle stealing*. The CPU merely delays its bus operations for one memory cycle to allow the direct memory-I/O transfer to steal one memory cycle.

DMA Controller

The DMA controller needs the usual circuits of an interface to communicate with the CPU and the I/O device. In addition, it needs an address register, a word-count register, and a set of address lines. The address register and address lines are used for direct communication with memory. The word-count register specifies the number of words that must be transferred. The data transfer may be done directly between the device and memory under control of the DMA.

Figure 13-19 shows the block diagram of a typical DMA controller. The unit communicates with the CPU via the data bus and control lines. The registers in the DMA are selected by the CPU through the address bus by enabling the *DS* (DMA select) and *RS* (register select) inputs. The *RD* (read) and *WR* (write) inputs are bidirectional. When the *BG* (bus granted) input is 0, the CPU can communicate with the DMA registers through the data bus to read from or write to those registers. When *BG* = 1, the CPU has relinquished the buses, and the DMA can communicate directly with memory by specifying an address on the address bus and activating the *RD* or *WR* control. The DMA communicates with the external peripheral through the DMA request and DMA acknowledge lines by a prescribed handshaking procedure.



□ **FIGURE 13-19**
Block Diagram of a DMA Controller

The DMA controller has three registers: an address register, a word-count register, and a control register. The address register contains an address to specify the desired location of a word in memory. The address bits go through bus buffers onto the address bus. The address register is incremented after each word is transferred to memory. The word-count register holds the number of words to be transferred. This register is decremented by one after each word transfer and internally tested for zero. The control register specifies the mode of transfer. All registers in the DMA appear to the CPU as I/O interface registers. Thus, the CPU can read from or write to the DMA registers under program control via the data bus.

After initialization by the CPU, the DMA starts and continues to transfer data between memory and the peripheral unit until an entire block is transferred. The initialization process is essentially a program consisting of I/O instructions that include the address for selecting particular DMA registers. The CPU initializes the DMA by sending the following information through the data bus:

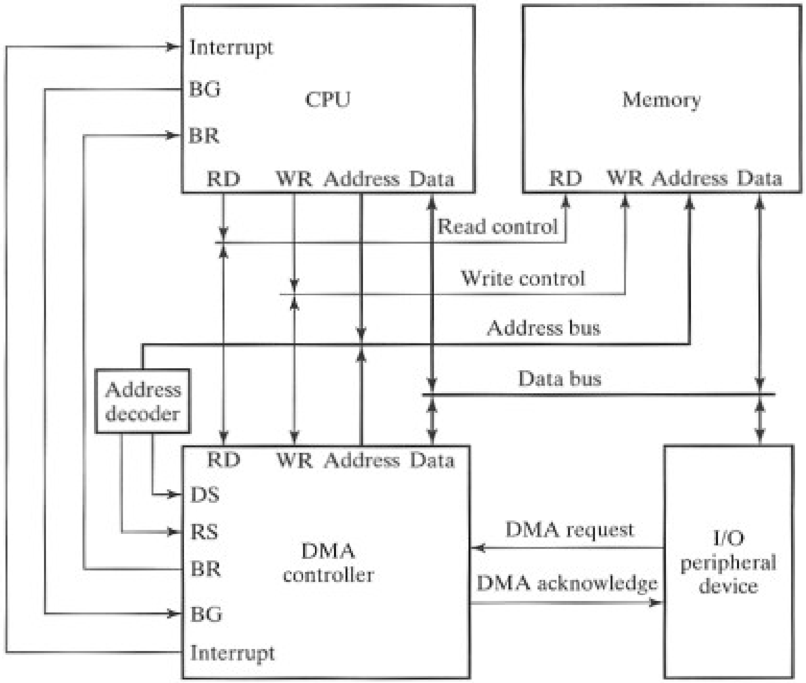
1. The starting address of the memory block in which data is available (for reading) or data is to be stored (for writing).
2. The word count, which is the number of words in the memory block.
3. A control bit to specify the mode of transfer, such as read or write.
4. A control bit to start the DMA transfer.

The starting address is stored in the address register, the word count in the word-count register, and the control information in the control register. Once the DMA is initialized, the CPU stops communicating with it unless the CPU receives an interrupt signal or needs to check how many words have been transferred.

DMA Transfer

The position of the DMA controller among the other components in a computer system is illustrated in Figure 13-20. The CPU communicates with the DMA through the address and data buses, as with any interface unit. The DMA has its own address, which activates the *DS* and *RS* lines. The CPU initializes the DMA through the data bus. Once the DMA receives the start control bit, it can begin transferring data between the peripheral device and memory. When the peripheral device sends a DMA request, the DMA controller activates the *BR* line, informing the CPU that it is to relinquish the buses. The CPU responds with its *BG* line, informing the DMA that the buses are disabled. The DMA then puts the current value of its address register onto the address bus, initiates the *RD* or *WR* signal, and sends a DMA acknowledge to the peripheral device.

When the peripheral device receives a DMA acknowledge, it puts a word on the data bus (for writing) or receives a word from the data bus (for reading). Thus, the DMA controls the read or write operation and supplies the address for memory. The peripheral unit can then communicate with memory through the data bus for a direct transfer of data between the two units while the CPU access to the data bus is momentarily disabled.



□ **FIGURE 13-20**
DMA Transfer in a Computer System

For each word that is transferred, the DMA increments its address register and decrements its word-count register. If the word count has not reached zero, the DMA checks the request line coming from the peripheral. In a high-speed device, the line will be activated as soon as the previous transfer is completed. A second transfer is then initiated, and the process continues until the entire block is transferred. If the speed of the peripheral is slower, the DMA request line may be activated somewhat later. In this case, the DMA disables the bus request line so that the CPU can continue to execute its program. When the peripheral requests a transfer, the DMA requests the buses again.

If the word count reaches zero, the DMA stops any further transfer and removes its bus request. It also informs the CPU of the termination of the transfer by means of an interrupt. When the CPU responds to the interrupt, it reads the contents of the word-count register. A value of zero indicates that all the words were successfully transferred. The CPU can read the word-count register at any time, as well, to check the number of words already transferred.

A DMA controller may have more than one channel. In this case, each channel has a request and acknowledge pair of control signals that are connected to separate peripheral devices. Each channel also has its own address register and word-count register so that channels with high priority are serviced before channels with lower priority.

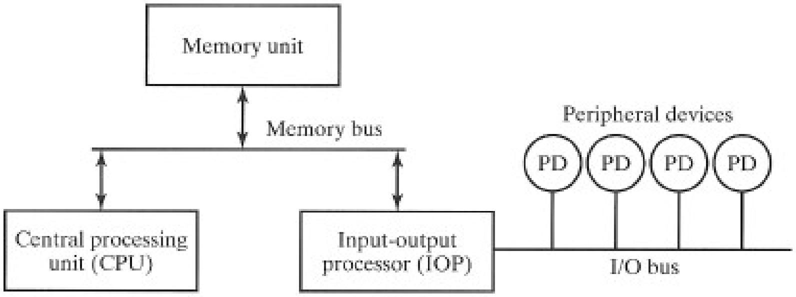
DMA transfer is very useful in many applications, including the fast transfer of information between magnetic disks and memory and between graphic displays and memory.

13-8 I/O Processors

Instead of having each interface communicate with the CPU, a computer may incorporate one or more external processors and assign them the task of communicating directly with all I/O devices. An input-output processor (IOP) may be classified as a processor with direct memory access capability that communicates with I/O devices. In this configuration, the computer system can be divided into a memory unit and a number of processors composed of the CPU and one or more IOPs. Each IOP takes care of input and output tasks, relieving the CPU of the "housekeeping" chores involved in I/O transfers. A processor that communicates with remote units over telephone and other communication media in a serial fashion is called a *data communication processor* (DCP). The benefit derived from using I/O processors is improved system performance, achieved through relieving the CPU of detailed tasks relating to I/O and assigning them to the appropriate I/O processors.

An IOP is similar to a CPU, except that it is designed to handle the details of I/O processing. Unlike the DMA controller, which must be set up entirely by the CPU, the IOP can fetch and execute its own instructions. IOP instructions are specifically designed to facilitate I/O transfers. In addition, the IOP can perform other processing tasks, such as arithmetic, logic, branching, and translation of code.

The block diagram of a computer with two processors is shown in Figure 13-21. The memory occupies a central position and can communicate with each processor by means of DMA. The CPU is responsible for processing data needed in the



□ **FIGURE 13-21**
Block Diagram of a Computer with I/O Processor

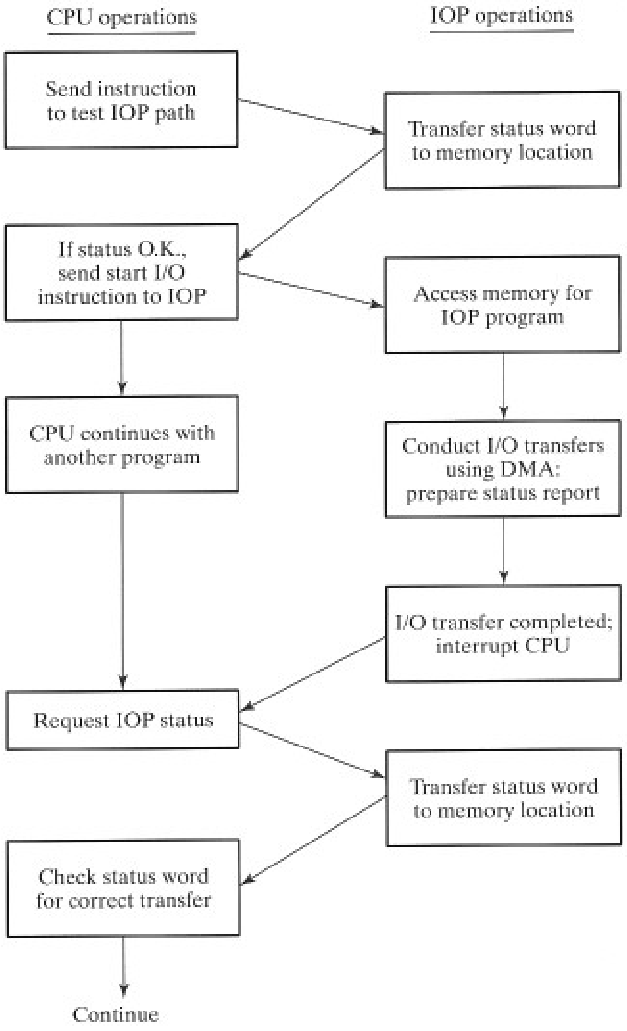
solution of computational tasks. The IOP provides a path for the transfer of data between various peripheral devices and the memory. The CPU is usually assigned the task of initiating the I/O program. From then on, the IOP operates independently of the CPU and continues to transfer data between external devices and memory. The data formats of peripheral devices often differ from those of memory and the CPU. The IOP must structure data words from many different sources. For example, it may be necessary to take four bytes from an input device and pack them into one 32-bit word before the transfer to memory. Data are gathered in the IOP at the device bit rate and bit capacity while the CPU is executing its own program. After assembly into a memory word, the data is transferred from the IOP directly into memory by stealing one memory cycle from the CPU. Similarly, an output word transferred from memory to the IOP is directed from the IOP to the output device at the device bit rate and bit capacity.

The communication between the IOP and the devices attached to it is similar to the program-controlled method of transfer. Communication with memory is similar to the DMA method. The way the CPU and IOP communicate with each other depends on the level of sophistication of the system. In very large-scale computers, each processor is independent of all the others, and any one processor can initiate an operation. In most computer systems, the CPU is the master, while the IOP is a slave processor. The CPU is assigned the task of initiating all operations, but I/O instructions are executed in the IOP. CPU instructions provide operations to start an I/O transfer and also to test I/O status conditions needed for making decisions on various I/O activities. The IOP, in turn, typically asks for attention from the CPU by means of an interrupt. It also responds to CPU requests by placing a status word in a prescribed location in memory, to be examined later by a CPU program. When an I/O operation is desired, the CPU informs the IOP where to find the I/O program and then leaves the details of the transfer to the IOP.

Instructions that are read from memory by an IOP are sometimes called *commands*, to distinguish them from instructions that are read by the CPU. An instruction and a command have similar functions. Commands are prepared by programmers and are stored in memory. The command words constitute the program for the IOP. The CPU informs the IOP where to find commands in memory when it is time to execute the I/O program.

Communication between the CPU and the IOP may take different forms, depending on the particular computer used. In most cases, the memory acts as a message center, where each processor leaves information for the other. To appreciate the operation of a typical IOP, we illustrate the method by which the CPU and IOP communicate with each other. This simplified example omits many operating details in order to provide an overview of basic concepts.

The sequence of operations may be carried out as shown in the flowchart of Figure 13-22. The CPU sends an instruction to test the IOP path. The IOP


CPU operations
IOP operations
Send instruction to test IOP path
Transfer status word to memory location
If status O.K., send start I/O instruction to IOP
Access memory for IOP program
CPU continues with another program
Conduct I/O transfers using DMA: prepare status report
I/O transfer completed; interrupt CPU
Request IOP status
Transfer status word to memory location
Check status word for correct transfer
Continue

□ **FIGURE 13-22**
CPU-IOP Communication

responds by inserting a status word in memory for the CPU to check. The bits of the status word indicate the condition of the IOP and I/O device, such as "IOP overload condition," "device busy with another transfer," or "device ready for I/O transfer." The CPU refers to the status word in memory to decide what to do next. If all is in order, the CPU sends the instruction to start the I/O transfer. The memory address received with this instruction tells the IOP where to find its program.

The CPU can now continue with another program while the IOP is busy with the I/O program. Both programs refer to memory by means of DMA transfer. When the IOP terminates the execution of its program, it sends an interrupt request to the CPU. The CPU responds by issuing an instruction to read the status from the IOP. The IOP then responds by placing the contents of its status report into a specified memory location. The status word indicates whether the transfer has been completed or whether any errors occurred during the transfer. By inspecting the bits in the status word, the CPU determines whether the I/O operation was completed satisfactorily, without errors.

The IOP takes care of all data transfers between several I/O units and memory while the CPU is processing another program. The IOP and CPU compete for the use of memory, so the number of devices that can be in operation is limited by the access time of the memory. It is not possible for I/O devices to saturate the memory in most systems, as the speed of most devices is much slower than that of the CPU. However, multiple fast units, such as magnetic disks or graphics displays, can use an appreciable number of the available memory cycles. In that case, the speed of the CPU may deteriorate because the CPU often has to wait for the IOP to conduct memory transfers.

13-9 Chapter Summary

In this chapter, we introduced I/O devices, typically called peripherals, and their associated digital support structures, including I/O buses, interfaces, and controllers. We studied the structure of a keyboard, a hard disk, and a graphics display. We looked at an example of a generic I/O interface and examined the interface and I/O controller for the keyboard. We introduced USB as an alternative solution to the attachment of many I/O devices. We considered timing problems between systems with different clocks and the parallel and serial transmission of information.

We also looked at modes of transferring information and saw how the more complex modes came about, principally to relieve the CPU from extensive, performance-robbing handling of I/O transfers. Interrupt-initiated transfers with multiple I/O interfaces lead to means of establishing priority between interrupt sources. Priority can be handled by software, serial daisy chain logic, or parallel interrupt-priority logic. Direct memory access accomplishes the transfer of data directly between an I/O interface and memory, with little CPU involvement. Finally, the I/O processor provides even greater independence of the CPU in handling I/O.

References

1. Patterson, D. A., and J. L. Hennessy *Computer Organization and Design: The Hardware/Software Interface.* San Francisco, CA: Morgan Kaufmann, 1998.
2. Van Gilluwe, F. *The Undocumented PC.* Reading, MA: Addison-Wesley, 1994.
3. Messmer, H. P. *The Indispensable PC Hardware Book.* 2nd ed. Reading, MA: Addison-Wesley, 1995.
4. MindShare, Inc. (Don Anderson). *Universal Serial Bus System Architecture.* Reading, MA: Addison-Wesley Developers Press, 1997.

Problems

The plus (+) indicates a more advanced problem and the asterisk (*) indicates a solution is available on the Companion Website for the text.

13–1. *Find the formatted capacity of the hard disks described in the following table:

Disk	Heads	Cylinders	Sectors/ Track	Bytes/ Sector
A	1	1023	63	512
B	4	8191	63	512
C	16	16383	63	512

13–2. Estimate the time required to transfer a block of 1MB (2^{20} B) from disk to memory given the following disk parameters: seek time, 8.5 ms; rotational delay, 4.17 ms; controller time, negligible; transfer rate, 100 MB/s.

13–3. The addresses assigned to the four registers of the I/O interface of Figure 13-5 are equal to the binary equivalent of 240, 241, 242, and 243. Show the external circuit that must be connected between an 8-bit I/O address from the CPU and the *CS*, *RS*0, and *RS*1 inputs of the interface.

13–4. *How many I/O interface units of the type shown in Figure 13-5 can be addressed by using a 16-bit address, assuming

(a) that each of the chip select (CS) lines is attached to a different address line?

(b) that address bits are fully decoded to form the chip select inputs?

13–5. Six interface units of the type shown in Figure 13-5 are connected to a CPU that uses an I/O address of eight bits. Each one of the six chip select (*CS*) inputs is connected to a different address line. Specifically, address line 0 is connected to the *CS* input of the first interface unit, and address line 5 is connected to the *CS* input of the sixth interface unit. Address lines 7 and 6 are connected to the $RS1$ and $RS0$ inputs, respectively, of all six interface units. Determine the 8-bit address of each register in each interface (a total of 24 addresses).

13–6. *A different type of I/O interface does not have the $RS1$ and $RS0$ inputs. Up to two registers can be addressed by using a separate I/O read signal and I/O write signal for each address available. Assume that 50% of the registers at the interface with the CPU are read only, 25% of the registers are write only, and 25% of the registers are both read and write (bidirectional). How many registers can be addressed if the address contains four bits?

13–7. A commercial interface unit uses names different from those appearing in this text for the handshake lines associated with the transfer of data from the I/O device to the interface unit. The interface input handshake line is labeled STB (strobe), and the interface output handshake line is labeled IBF (input buffer full). A low-level signal on STB loads data from the I/O bus into the interface data register. A high-level signal on IBF indicates that the data has been accepted by the interface. IBF goes low after an I/O read signal from the CPU when it reads the contents of the data register.

(a) Draw a block diagram showing the CPU, the interface, and the I/O device, along with the pertinent interconnections between the three units.

(b) Draw a timing diagram for the handshaking transfer.

13–8. *Assume that the transfers with strobing shown in Figure 13-6 are between a CPU on the left and an I/O interface on the right. There is an address coming from the CPU for each of the transfers, both of which are initiated by the CPU.

(a) Draw block diagrams showing the interconnections for the transfers.

(b) Draw the timing diagrams for the two transfers, assuming that the address must be applied some time before the strobe becomes 1 and removed some time after the strobe becomes 0.

13–9. Assume that the transfers with handshaking shown in Figure 13-7 are between a CPU on the left and an I/O interface on the right. There is an address coming from the CPU for each of the transfers, both of which are initiated by the CPU.

(a) Draw block diagrams, showing that interconnections for the transfers.

(b) Draw the timing diagrams, assuming that the address must be applied some time before the request becomes 1 and removed some time after the request becomes 0.

13–10. *How many characters per second can be transmitted over a 57,600-baud line in each of the following modes? (Assume a character code of eight bits.)

(a) Asynchronous serial transmission with two stop bits.

(b) Asynchronous serial transmission with one stop bit.

(c) Repeat a and b for a 115,200-baud line.

13–11. Sketch the timing diagram of the 11 bits (similar to Figure 13-8) that are transmitted over an asynchronous serial communication line when the ASCII letter E is transmitted with even parity. Assume that the ASCII character code is transmitted least significant bit first, with the parity bit following the character code.

13–12. What is the difference between the synchronous and the asynchronous serial transfer of information?

13–13. *Sketch the waveforms for the SYNC pattern used for USB and the corresponding NRZI waveform. Explain why the pattern selected is a good choice for achieving synchronization.

13–14. The following stream of data is to be transmitted by USB:

0111111100100000111111011111101

(a) Assuming bit stuffing is not used, sketch the NRZI waveform.
(b) Modify the stream by applying bit stuffing.
(c) Sketch the NRZI waveform for the result in b.

13–15. *The 8-bit ASCII word "Bye" is to be transmitted to a device address 39 and endpoint 2. List the Output and Data 0 packets and the Handshake packet for a Stall for this transmission prior to NRZI encoding.

13–16. Repeat problem 13-15 for the word "Hlo" and a Handshake packet of type No Acknowledge.

13–17. What is the basic advantage of using interrupt-initiated data transfer over transfer under program control without an interrupt?

13–18. *What happens in the daisy chain priority interrupt shown in Figure 13-15 when device 0 requests an interrupt after device 2 has sent an interrupt request to the CPU, but before the CPU responds with the interrupt acknowledge?

13–19. Consider a computer without priority interrupt hardware. Any one of many sources can interrupt the computer, and any interrupt request results in storing the return address and branching to a common interrupt routine. Explain how a priority can be established in the interrupt service program.

13–20. *What changes are needed in Figure 13-17 to make the four VAD values equal to the binary equivalent of 024, 025, 026, and 027?

13–21. Repeat problem 13-20 for VAD values 224, 225, 226 and 227.

13–22. *Design parallel priority interrupt hardware for a system with six interrupt sources.

13–23. A priority structure is to be designed that provides vector addresses.
(a) Obtain the condensed truth table of a 16 × 4 priority encoder.
(b) The four outputs w, x, y, z from the priority encoder are used to provide an 8-bit vector address in the form $10wxyz01$. List the 16 addresses, starting from the one with the highest priority.

13–24. *Why are the read and write control lines in a DMA controller bidirectional? Under what condition and for what purpose are they used as inputs? Under what condition and for what purpose are they used as outputs?

13–25. It is necessary to transfer 1024 words from a magnetic disk to a section of memory starting from address 2048. The transfer is by means of DMA as shown in Figure 13-20.

(a) Give the initial values that the CPU must transfer to the DMA controller.

(b) Give the step-by-step account of the actions taken during the input of the first two words.

CHAPTER 14

MEMORY SYSTEMS

In Chapter 9, we discussed basic RAM technology for implementing memory systems, including SRAMs and DRAMs. In the current chapter, we probe more deeply into what really constitutes a computer memory system. We begin with the premise that a fast, large memory is desirable and demonstrate that a straightforward implementation of such a memory for the typical computer is too costly and too slow. As a consequence, we study a more elegant solution in which most accesses to memory are fast (but some are slow) and the memory appears to be large. This solution employs two concepts: cache memory and virtual memory. A cache memory is a small, fast memory with special control hardware that permits it to handle a significant proportion of all accesses required by the CPU with an access time of the order of the CPU clock period. Virtual memory, implemented in software and hardware, using an intermediate-sized main memory (typically, DRAM), gives the appearance of a large main memory with access time similar to the main memory for the vast majority of accesses. The actual storage medium for most of the code and data in the virtual memory is a hard disk. Because there is a progression of components in the memory system having larger and larger storage capability, but slower and slower access (cache, main memory, and hard disk), the term *memory hierarchy* is applied.

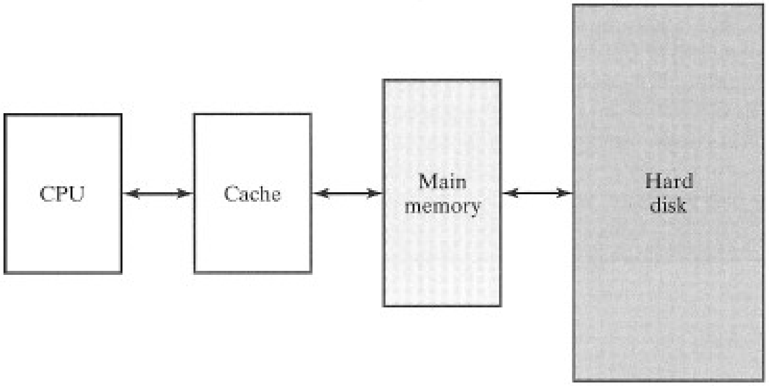
In the generic computer at the beginning of Chapter 1, a number of components are heavily involved in the memory hierarchy. Within the processor, there is the memory management unit (MMU), which is hardware provided to support virtual memory. Also in the processor, the internal cache appears. Since this cache is too small to fully support the cache function, there is also an external cache attached to the CPU bus. Of course, the RAM is involved, and due to the presence of virtual memory, the hard disk, the bus interface, and the disk controller all have a role as parts of the memory system.

14-1 Memory Hierarchy

Figure 14-1 shows a generic block diagram for a memory hierarchy. The lowest level of the hierarchy is a small, fast memory called a *cache*. For the hierarchy to function well, a very large proportion of the CPU instruction and operand fetches are expected to be from the cache. At the next level upward in the hierarchy is the *main memory*. The main memory serves directly most of the CPU instruction and operand fetches not satisfied by the cache. In addition, the cache fetches all of its data, some portion of which is passed on to the CPU, from the main memory. At the top level of the hierarchy is the *hard disk*, which is accessed only in the very infrequent cases in which a CPU instruction or operand fetch is not found in main memory.

With this memory hierarchy, since the CPU fetches most of the instructions and operands from the cache, it "sees" a fast memory, most of the time. Occasionally, when a word must come from main memory, a fetch takes somewhat longer. Very infrequently, when a word must be fetched from the hard disk, the fetch takes a very long time. In this last case, the CPU is likely to experience an interrupt that passes execution to a program which brings in a block of words from the hard disk. On balance, the situation is usually satisfactory, providing an average fetch time close to that of the cache. Moreover, the CPU sees a memory address space considerably larger than that of main memory.

With this general notion of a memory hierarchy kept in mind, we will proceed to consider an example that illustrates the potential power of such a hierarchy. However, there is one issue to be clarified first. In most instruction set architectures, the smallest of the objects that are addressed is a byte rather than a word. For a given load or store operation, whether a byte or word is affected is typically determined by the opcode. Addressing to bytes brings with it some assumptions and hardware details that are important, but, if used up to this point in the text, would have unnecessarily complicated much of the material covered. Consequently, for simplicity, we have assumed up to now that an addressed location contains a word. By contrast, in this chapter we will assume that addresses are defined



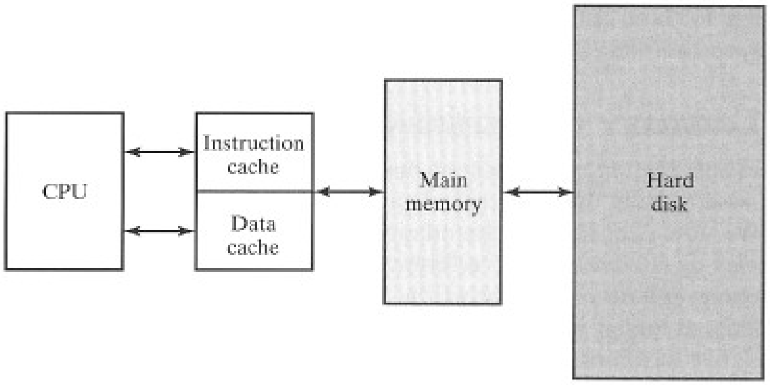
□ **FIGURE 14-1**
Memory Hierarchy

for bytes, to match current practice. Nevertheless, we will still assume that data is moved around outside of the CPU as words or sets of words, to avoid messy explanations relating to the manipulation of bytes. This assumption simply hides some hardware details that would distract from the main focus of our discussion, but nevertheless must be handled by the hardware designer. To accomplish the simplification, if there are 2^b bytes per word, we will ignore the last b bits of the address. Since these bits are not needed to address a word, we show their values as 0's. For the examples we will present, b is always equal to 2, so two 0's are shown.

In Section 12-3, the pipelined CPU had a memory address with 32 bits and was able to access an instruction and data, if necessary, in each of the 1-ns clock cycles. Also, we assumed that the instruction and the data were, in effect, fetched from two different memories. To support this assumption in this chapter, we will suppose initially that the memory is divided in half—one-half for instructions and one-half for data. Each half of the memory must have an access time of 1 ns. In addition, if we utilize all the bits in the 32-bit address, then the memory can contain up to 2^{32} bytes, or 4 gigabytes (GB), of information. So the goal is to have two 2-GB memories, each with an access time of 1 ns.

Is such a memory realistic in terms of current (2003) computer technology? The typical memory is constructed of DRAM modules ranging in size from 16 to 64 Mbytes. The typical access time is about 10 ns. Thus, our two 2-GB memories would have an access time of somewhat more than 10 ns per word. This kind of memory both is too costly and operates at only one-tenth the desired speed. So our goal must be achieved another way, leading us to explore a memory hierarchy.

We begin by assuming a hierarchy with two caches, one for instructions and one for data, as shown in Figure 14-2. The use of these two caches permits one instruction and one operand to be fetched, or one instruction to be fetched and one result to be stored, in a single clock cycle if the caches are fast enough. In terms of the generic computer, we assume that the caches are internal, so that they can operate at speeds comparable to that of the CPU. Thus, fetches from the instruction cache and fetches from and stores to the data cache can be accomplished in



□ **FIGURE 14-2**
Example of Memory Hierarchy

2 ns. Hence, most of the fetches and stores for the CPU are from or to these caches and will take 2 CPU clock cycles. Suppose, then, that we are satisfied with most—say, 95%—of the memory accesses taking 2 ns. Suppose further that most of the remaining 5% of the memory accesses take 10 ns. Then the average access time is

$$0.95 \times 2 + 0.05 \times 10 = 2.4 \text{ ns}$$

This means that, on 19 out of every 20 memory accesses, the CPU operates at full speed, while the CPU will have to wait for 10 clock cycles for 1 out of every 20 memory accesses. This wait can be accomplished by stalling the CPU pipeline. Thus, we have accomplished our goal of "most" memory accesses taking 2 ns. But there is still the problem of the cost of the large memory.

Now suppose that, in addition to infrequently accepting a wait for a word from main memory that will take more than 10 ns, we are also willing to accept a very infrequent wait for a hard disk access taking 13 ms = 1.3×10^7 ns. Suppose that we have data indicating that about 95% of the fetches will be from a cache and about 4.999995% of the fetches will be from main memory. With this information, we can estimate the average access time as

$$0.95 \times 2 + 0.04999995 \times 10 + 5 \times 10^{-8} \times 1.3 \times 10^{7} = 3.05 \text{ ns}$$

Thus, the average access time is about 3 times the 1 ns CPU clock period, but is about one-third of the 10 ns access time for main memory, again with 19 out of 20 of the accesses taking place in 2 ns. So we have achieved an average access time of about 3.05 ns for a memory structure with a capacity of 2^{32} bytes, not far from the original goal. Further, the cost of this memory hierarchy is tens of times smaller than the large, fast memory approach.

It therefore appears that the original goal of the appearance of a fast, large memory has been approached by the memory hierarchy at a reasonable cost. But along the way, we made some assumptions, namely, that 95% of the time the word desired would come from what we are now calling the cache and that 99.999995% of the time the words would come from either cache or main memory, with the remainder from hard disk. In the rest of this chapter, we will explore why assumptions similar to these usually hold, and we will examine the hardware and associated software components needed to achieve the goals of the memory hierarchy.

14-2 Locality of Reference

In the previous section, we indicated that the success of the memory hierarchy is based on assumptions that are critical to achieving the appearance of a large, fast memory. We now deal with the foundation for making these assumptions, which is called *locality of reference*. Here "reference" means reference to memory for accessing instructions and for reading or writing operands. The term "locality" refers to the relative times at which instructions and operands are accessed (*temporal locality*) and the relative locations at which they reside in main memory (*spatial locality*).

Let us consider first the nature of the typical program. A program frequently contains many loops. In a loop, a sequence of instructions is executed many times before the program exits the loop and moves on to another loop or straight-line code not in a loop. In addition, loops are often nested in a hierarchy in which loops are contained in loops, and so on. Suppose we have a loop of eight instructions that is to be executed 100 times. Then for 800 executions, all instruction fetches will occur from just eight addresses in memory. Thus, each of the eight addresses is visited 100 times during the time the loop is executed. This is an example of temporal locality in the sense that an address which is accessed is likely to be accessed many times in the near future. Also, it is likely that the addresses of the instructions will be in sequential order. Thus, if an address is accessed for an instruction, nearby addresses are going to be addressed during the execution of the loop. This is an example of spatial locality.

In terms of accessing operands, similar temporal and spatial localities also occur. For example, in a computation on an array of numbers, there are multiple visits to the locations of many of the operands, giving temporal locality. Also, as the computation proceeds, when a particular address is accessed for a number, sequential addresses near to it are likely to be accessed for other numbers in the array, giving spatial locality.

From the prior discussion, we can conjecture that there is significant locality of reference in computer programs. To verify this decisively, it is necessary to study the patterns of execution of real programs. Such studies have demonstrated the presence of significant temporal and spatial locality of reference and play an important role in the design of caches and virtual memory systems.

The next question to answer is: What is the relation of locality of reference to the memory hierarchy? To examine this issue, we consider again the instruction fetch within a loop and look at the relationship between the cache and main memory. Initially, we assume that instructions are present only in main memory and that the cache is empty. When the CPU fetches the first instruction in a loop, it obtains the instruction from main memory. But the instruction and a portion of its address called the *address tag* are also placed in the cache. What then happens for the next 99 executions of this instruction? The answer is that the instruction can be fetched from the cache, which provides a much faster access. This is temporal locality at work: The instruction that was fetched once will tend to be used again and is now present in the cache for fast access.

Additionally, when the CPU fetches the instruction from main memory, the cache fetches nearby instructions into its SRAM. Now suppose that the nearby instructions include the entire loop of eight instructions presented in our example. Then all of the instructions are in the cache. By bringing in such a block of instructions, the cache is able to exploit spatial locality: It takes advantage of the fact that the execution of the first instruction implies the execution of instructions with nearby addresses by making the latter instructions available for fast access.

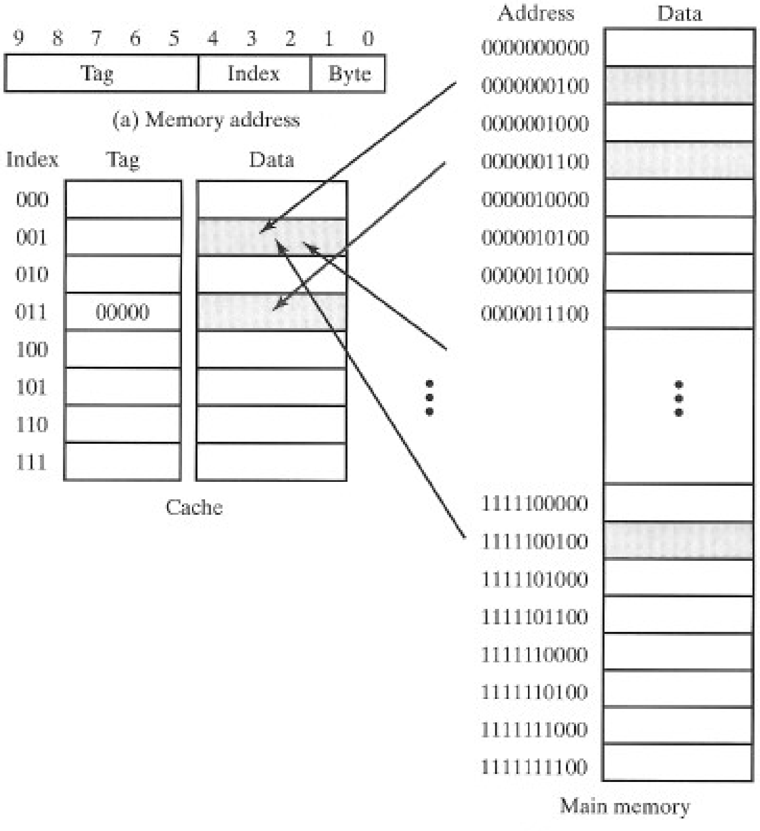
In our example, each of the instructions is fetched from main memory exactly once for the 100 executions of the loop. All other instruction fetches come from the cache. Thus, in this particular example, at least 99% of the instructions being executed are fetched from the cache, so that the rate of execution of instructions is governed almost completely by the cache access time and CPU speed, and very little by the main memory access time. Without temporal locality, many more accesses to main memory would occur, slowing down the system.

A relationship similar to that between cache and the main memory can exist between main memory and the hard disk. Again, both temporal and spatial locality of reference are of interest, except this time on a much larger scale. Programs and data are fetched from the hard disk, and data is written to the hard disk in blocks that range from kilowords to megawords. Ideally, once the code and initial data for a program reside in main memory, the hard disk need not be accessed except for storing final results of the program. But this can happen only if all of the code and data, including intermediate data used by the program, reside fully in main memory. If not, then it will be necessary to bring in code from the hard disk and to read and write data from and to the hard disk during program execution. Words are read from and written to the disk in blocks referred to as *pages*. If the movement of pages between main memory and hard disk is transparent to the programmer, then it will appear as if main memory is large enough to hold the entire program and all of the data. Hence, this automated arrangement is referred to as *virtual memory*. During the execution of the program, if an instruction to be executed is not in main memory, the CPU program flow is diverted to bring the page containing the instruction into main memory. Then the instruction can be read from main memory and executed. The details of this operation and the hardware and software actions required for it will be covered in Section 14-4.

In summary, locality of reference is absolutely key to the success of the concepts of cache memory and virtual memory. In the case of most programs, locality of reference is present to a fairly high degree. But occasionally, one does encounter a program that, for example, requires frequent access to a large body of data that cannot be accommodated in main memory. In such a case, the computer spends almost all of its time moving information between main memory and the hard disk and does little other computation. Continuous sounds emanating from the hard disk as the heads move from track to track is a telltale sign of this phenomenon, which is referred to as *thrashing*.

14-3 CACHE MEMORY

To illustrate the concept of cache memory, we assume a very small cache of eight 32-bit words and a small main memory with 1 KB (256 words), as shown in Figure 14-3. Both of these are too small to be realistic, but their size makes illustration of the concepts easier. The cache address contains 3 bits, the memory address 10. Out of the 256 words in main memory, only 8 at a time may lie in the cache. In order for the CPU to address a word in the cache, there must be information in the cache to identify the address of the word in main memory. If we consider the example of the loop in the last section, clearly, we find it desirable to contain the entire loop within the cache, so that all of the instructions can be fetched from the cache while the program is executing most of the passes through the loop. The instructions in the loop lie in consecutive word addresses.



(a) Memory address

(b) Cache mapping

□ **FIGURE 14-3**
Direct Mapped Cache

Thus, it is desirable for the cache to have words from consecutive addresses in main memory present simultaneously. A simple way to facilitate this feature is to make bits 2 through 4 of the main memory address be the cache address. We refer to these bits as the *index*, as shown in Figure 14-3. Note that the data from address 0000001100 in main memory must be stored in cache address 011. The upper 5 bits of the main memory address, called the *tag*, are stored in the cache along with the data. Continuing the example, we find that for main memory address 0000001100, the tag is 00000. The tag combined with the index (or cache address) and 00 byte field identify an address in main memory.

Suppose that the CPU is to fetch an instruction from location 000001100 in main memory. This instruction may actually come from either the cache or main memory. The cache separates the tag 00000 from the cache address 011, internally fetches the tag and the stored word from location 011 in the cache memory, and compares the tag fetched with the tag portion of the address from the CPU. If the tag fetched is 00000, then the tags match, and the stored word fetched from cache memory is the desired instruction. Thus, the cache control places this word on the bus to the CPU, completing the fetch operation. This case in which the memory word is fetched from cache is called a *cache hit*. If the tag fetched from cache memory is not 00000, then there is a tag mismatch, and the cache control notifies main memory that it must provide the memory word, which is not available in the cache. This situation is called a *cache miss*. For a cache to be effective, the slower fetches from main memory must be avoided as much as possible, making considerably more cache hits than cache misses necessary.

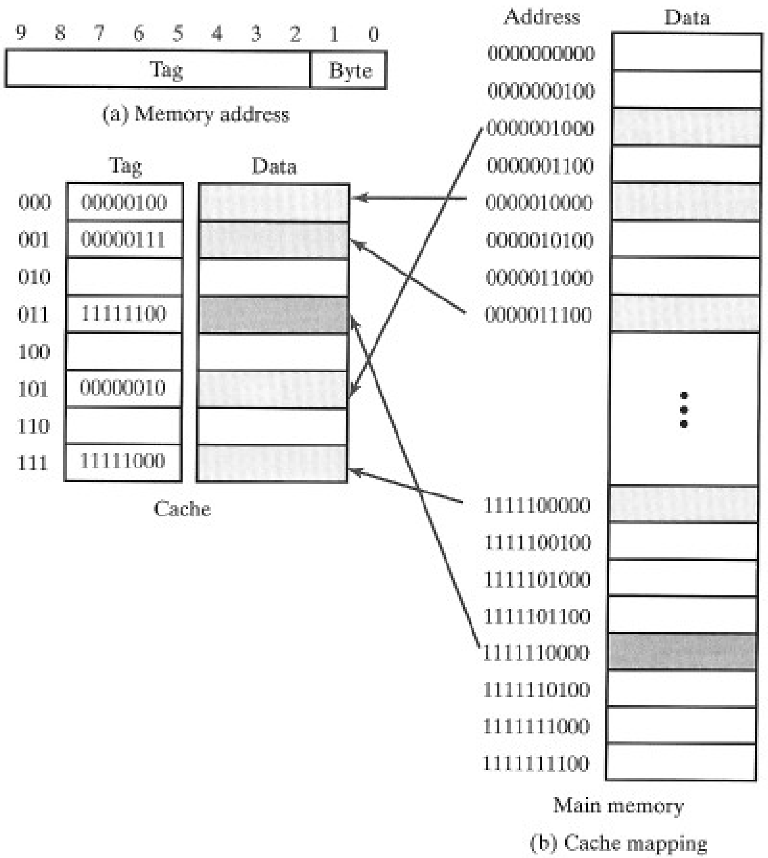
When a cache miss occurs on a fetch, the word from main memory is not placed just on the bus for the CPU. The cache also captures the word and its tag and stores them for future access. In our example, the tag 00000 and the word from memory will be written in cache location 011 in anticipation of future accesses to the same memory address. The handling of writes to memory will be dealt with later in the chapter.

Cache Mappings

The example we just considered uses a particular association or mapping between the main memory address and the cache address; namely, the last three bits of the main memory word address are the cache address. Additionally, there is only one location in the cache for the 2^5 locations in main memory that have their last three bits in common. This mapping in Figure 14-3 in which only one specific location in the cache can contain the word from a particular main memory location is called *direct mapping*.

Direct mapping for a cache, however, does not always produce the most desirable situation. In our loop instruction fetch example, suppose that instructions and data are in the same cache and that data from location 1111101100 is frequently used. Then when the instruction in 0000001100 is fetched, location 011 in the cache is likely to contain the data from 1111101100 and tag 11111. A cache miss occurs and causes tag 11111 to be replaced in the cache with tag 00000 and the data to be replaced with the instruction. But the next time the data is needed, another cache miss occurs, since the location in the cache is now occupied by the instruction. Throughout the execution of the loop, both instruction fetch and data fetch cause many cache misses, significantly slowing CPU processing. To solve this problem, we explore alternative cache mappings.

In direct mapping, 2^5 addresses in main memory map to the single address in the cache that matches their last three bits. These locations are highlighted in gray in Figure 14-3 for index 001. As is illustrated, only one of the 2^5 addresses can have its word in cache address 001 at any time. In contrast, suppose that we let locations in main memory map into an arbitrary location in the cache. Then any location in memory can be mapped to any one of the eight addresses in the cache. This means that the tag will now be the full main memory word address. We examine the operation of such a cache having a *fully associative* mapping in Figure 14-4. Note that in this case there are two main memory addresses, 0000010000 and 1111110000, with bits 2 through 4 equal to 100 among the cache

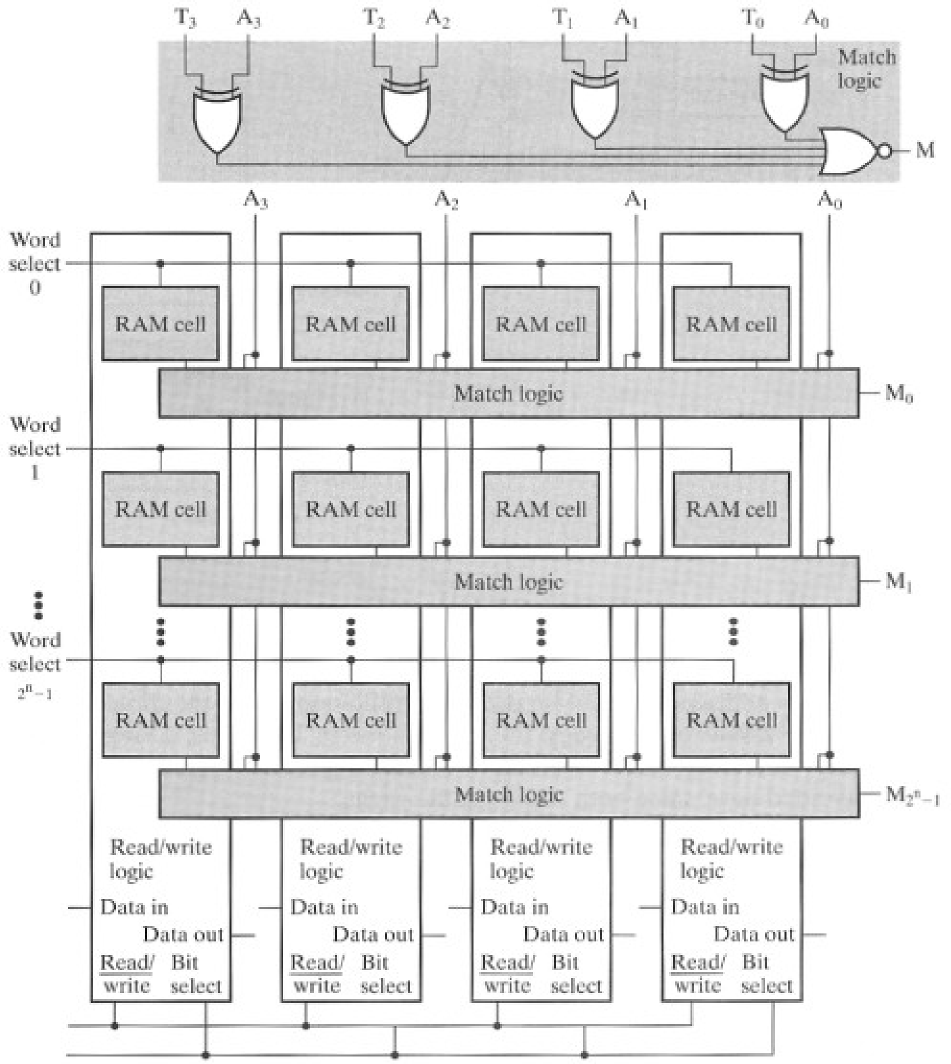


□ **FIGURE 14-4**
Fully Associative Cache

tags. These two addresses cannot be present simultaneously in the direct-mapped cache, as they would both occupy the cache address 100. Thus, a succession of cache misses due to alternate fetching of an instruction and data with the same index is avoided here, since both can be in the cache.

Now suppose that the CPU is to fetch an instruction from location 0000010000 in main memory. This instruction may actually be returned from either the cache or main memory. Since the instruction might lie in the cache, the cache must compare 00000100 to each of its eight tags. One way to do this is to successively read each tag and the associated word from the cache memory and compare the tag to 00000100. If a match occurs, as it will for the given address and cache location 000 in Figure 14-4, a cache hit occurs. The cache control then places the word on the bus to the CPU, completing the fetch operation. If the tag fetched from the cache is not 00000100, then there is a tag mismatch, and the cache control fetches the next successive tag and word. In the worst case, a match on the tag in cache address 111, eight fetches from the cache are required before the cache hit occurs. At 2 ns a fetch, this requires at least 16 ns, about half the time it would take to obtain the instruction from main memory. So successive reads of tags and words from the cache memory to find a match is not a very desirable approach. Instead, a structure called *associative memory* implements the tag portion of the cache memory.

Figure 14-5 shows an associative memory for a cache with 4-bit tags. The mechanism for writing tags into the memory uses a conventional write. Likewise, the tags can be read from the memory using the conventional memory read. Thus, the associative memory can use the bit slice model for RAM presented in



□ **FIGURE 14-5**
Associative Memory for 4-bit Tags

Chapter 9. In addition, each tag storage row has match logic. The implementation of this logic and its connection to the RAM cells are shown in the figure. The match logic does an equality comparison or match between the tag T and the applied address A from the CPU. The match logic for each tag is composed of an exclusive-OR gate for each bit and a NOR gate that combines the outputs of the exclusive-ORs. If all of the bits of the tag and the address match, then the outputs of all the exclusive-ORs are 0 and the NOR output is a 1, indicating a match. If there is a mismatch between any of the bits in the tag and the address, then at least one exclusive-OR has a 1 output, which causes the output of the NOR gate to be 0, indicating a mismatch.

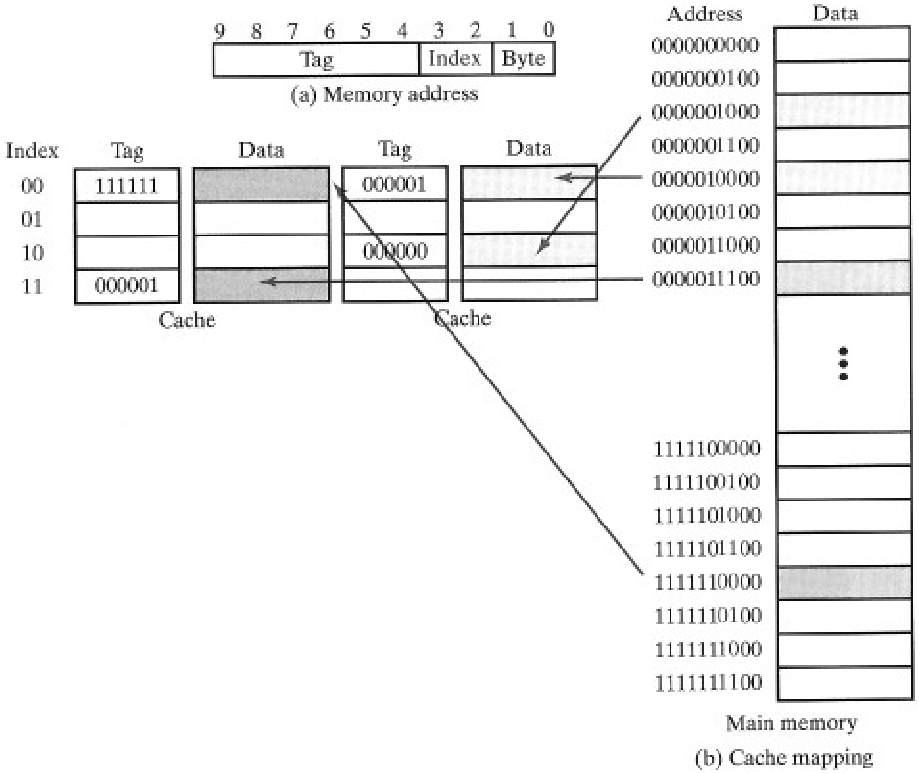
Since all tags are unique, only two situations can arise in the associative memory: there will be a match, with a 1 on the output of the match logic for one matching tag and a 0 on the remaining match logic outputs; or there will be no match, and all of the match logic outputs will be 0. With an associative memory holding the cache tags, the outputs of the match logic drive the word lines for the data memory words to be read. A signal must indicate whether a hit or a miss has occurred. If this signal is 1 for a hit and 0 for a miss, then it can be generated by using the OR of the match outputs. In the case of a hit, a 1 on Hit/$\overline{\text{miss}}$ places the word on the memory bus to the CPU; in the case of a miss, a 0 on Hit/$\overline{\text{miss}}$ tells the main memory that it is to provide the word addressed.

As in the case of the direct-mapped cache discussed earlier, the fully associative cache must capture the data word and its address tag and store them for future accesses. But now a new problem arises: Where in the cache are the tag and data to be placed? In addition to selecting a cache mapping, the cache designer must select a replacement approach that determines the location in the cache to be used for the incoming tag and data. One possibility is to select a *random replacement* location. The 3-bit address can be read from a simple hardware structure that generates a number which satisfies certain properties of random numbers. A somewhat more thoughtful approach is to use a first in, first out (*FIFO*) location. In this case, the location selected for replacement is the one that has occupied the cache for the longest time, based on the notion that the use of this oldest entry is likely to be finished. An approach that appears to attack the replacement problem even more directly is the *least recently used* (LRU) location approach. The goal of this approach is to replace the entry that has been unused for the longest time—hence the least recently used entry. The reason is that a cache entry that has not been used for the longest time is least likely to be used in the future. Thus, it can be replaced by a new cache entry. Although the LRU approach yields better results for caches, the difference between it and the other approaches is not large, and full implementation is costly. As a consequence, if used at all, the LRU approach is often only approximated.

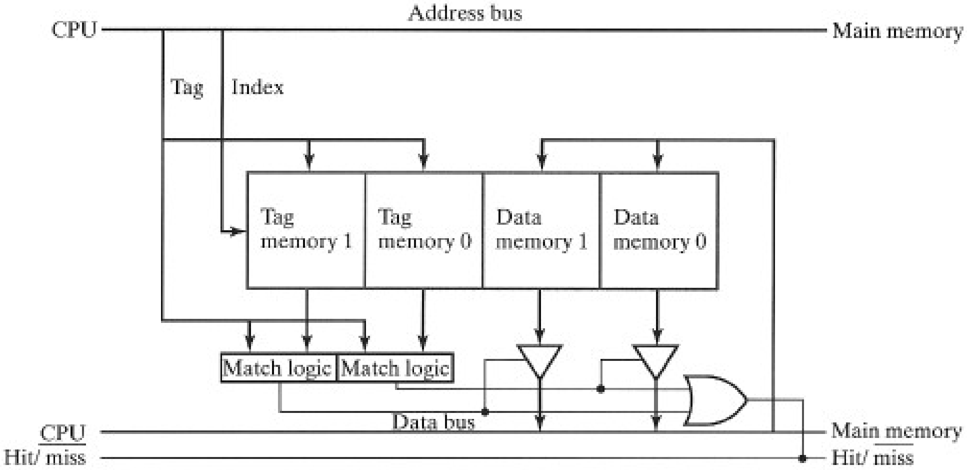
There are also performance and cost issues surrounding the fully associative cache. Although such a cache provides maximum flexibility and good performance, it is not clear that the cost is justified. In fact, an alternative mapping that has better performance and eliminates the cost of most of the matching logic is a compromise between a direct-mapped cache and a fully associative cache. For such a mapping, lower order address bits act much as they do in direct mapping;

however, for each combination of lower order address bits, instead of having one location, there is a *set* of *s* locations. As with direct mapping, the tags and words are read from the cache memory locations addressed by the lower order address bits. For example, if the *set size s* equals two, then two tags and the two accompanying data words are read simultaneously. The tags are then simultaneously compared to the CPU-supplied address using just two matching logic structures. If one of the tags matches the address, then the associated word is returned to the CPU on the memory bus. If neither tag matches the address, then the two 0 matching values are used to send a miss signal to the CPU and main memory. Since there are sets of locations and associativity is used on sets, this technique is called *set-associative mapping*. Such a mapping with a set size *s* is an *s-way* set-associative mapping.

Figure 14-6 shows a two-way set-associative cache. There are eight cache locations arranged in four rows of two locations each. The rows are addressed by a 2-bit index and contain tags made up of the remaining six bits of the main memory address. The cache entry for a main memory address must lie in a specific row of the cache, but can be in either of the two columns. In the figure, the addresses are the same as they are in the fully associative cache in Figure 14-4. Note that no



□ **FIGURE 14-6**
Two-way Set-associative Cache



□ **FIGURE 14-7**
Partial Hardware Block Diagram for Set-associative Cache

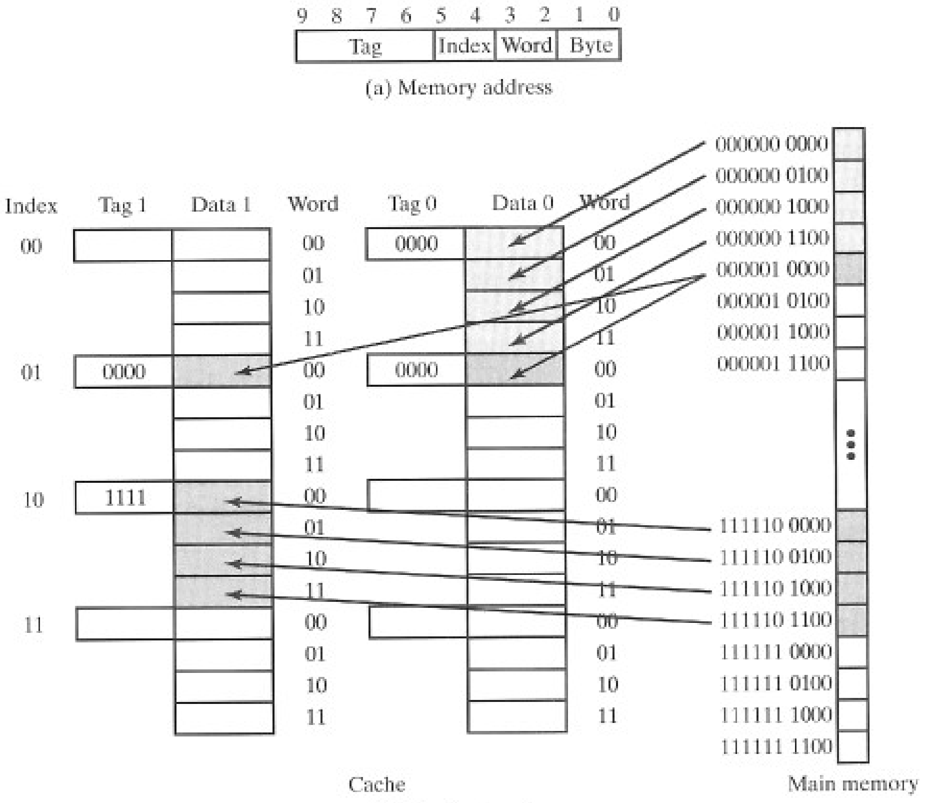
mapping is shown for main memory address 1111100000, since the two cache cells in set 00 are already occupied by addresses 0000010000 and 1111110000. In order to accommodate 1111100000, the set size would need to be at least three. This example illustrates a case in which the reduced flexibility of a set-associative cache, compared to a fully associative cache, has an impact. The impact declines as the set size increases.

Figure 14-7 is a section of a hardware block diagram for the set-associative cache of Figure 14-6. The index is used to address each row of the cache memory. The two tags read from the tag memories are compared to the tag part of the address on the address bus from the CPU. If a match occurs, then the three-state buffer on the corresponding data memory output is activated, placing the data onto the data bus to the CPU. In addition, the match signal causes the output of the Hit/$\overline{\text{miss}}$ OR gate to become 1, indicating a hit. If a match does not occur, then Hit/$\overline{\text{miss}}$ is 0, informing the main memory that it must supply the word to the CPU and informing the CPU that the word will be delayed.

Line Size

To this point, we have assumed that each cache entry consists of a tag and a single memory word. In real caches, spatial locality is to be exploited, so additional words close to the one addressed are included in the cache entry. Then, rather than a single word being fetched from main memory when a cache miss occurs, a block of *l* words called a *line* is fetched. The number of words in a line is a power of two, and the words are aligned on address boundaries. For example, if four words are included in a line, then the addresses of the words in the line differ only in bits 2 and 3. The use of a block of words changes the makeup of the fields into which the cache divides the address. The new field structure is shown in Figure 14-8(a). Bits 2 and 3, the Word field, are used to address the word within the line. In this case, two bits are used, so there are four words per line. The next field, Index, identifies the set. Here there are two bits used, so there are four sets of tags and lines. The remainder of the address word is the Tag field, which contains the remaining four bits of the 10-bit memory address.

The resulting cache structure is shown in Figure 14-8(b). The tag memory has eight entries, two in each of the four sets. Corresponding to each of the tag entries is a line of four data words. To ensure fast operation, Index is applied to the tag memory to read two tags, one for each of the set entries, simultaneously. At the same time, Index and the Word address are applied to read out two words from the cache data memory that correspond to the two tags. Matching logic provided for each of the two set elements compares each tag to the CPU-supplied address. If a match occurs, then the associated cache data word already read is placed on the memory bus to the CPU. Otherwise, a cache miss is signaled, and the word addressed is returned from main memory to the CPU. The line containing the word



(a) Memory address

(b) Cache mapping

□ **FIGURE 14-8**
Set-associative Cache with 4-word Lines

and its tag are also loaded into the cache. To facilitate loading the entire line of words, the width of the memory bus between main memory and the cache, as well as the cache load path, is made more than one word wide. Ideally, for our example the path is 4 × 32 = 128 bits wide. This allows the entire line to be placed in the cache in a single main memory read cycle. If the path is narrower, then a sequence of several reads from main memory is required.

An additional decision that the cache designer has to make is to determine the line size. A wide path to memory can affect both cost and performance, and a narrower path can slow transfer of the line to the cache. These features encourage a smaller cache line size, while spatial locality of reference encourages a larger line. In current systems, however, use of synchronous DRAM facilitates reading or writing large cache lines without the cost and performance issues associated with wide path. The rapid writing to and reading from memory of consecutive words achieved by using synchronous DRAM matches well the needs for transferring cache lines.

Cache Loading

Before any words and tags have been loaded into the cache, all locations contain invalid information. If a hit occurs on the cache at this time, then the word fetched and sent to the CPU cannot have come from main memory and is invalid. As lines are fetched from main memory into the cache, cache entries become valid, but there is no way to distinguish valid from invalid entries. To deal with this problem, in addition to the tag, a bit is added to each cache entry. This *valid bit* indicates that the associated cache line is valid (1) or invalid (0). It is read out of the cache along with the tag. If the valid bit is 0, then a cache miss occurs even if the tag matches the address from the CPU, requiring the addressed word to be taken from main memory.

Write Methods

We have focused so far on reading instructions and operands from the cache. What happens when a write occurs? Recall that, up to now, the words in a cache have been viewed simply as copies of words from main memory that are read from the cache to provide faster access. Now that we are considering writing results, this viewpoint changes somewhat. Following are three possible write actions from which we can select:

1. Write the result into main memory.
2. Write the result into the cache.
3. Write the result into both main memory and the cache.

Various realistic cache write methods employ one or more of these actions. Such methods fall into two main categories: write-through and write-back.

In *write-through*, the result is always written to main memory. This uses the main memory write time and can slow down processing. The slowdown can be partially avoided by using *write buffering*, a technique in which the address and word to be written are stored in special registers called write buffers by the CPU so that it can continue processing during the write to main memory. In most cache designs, the result is also written into the cache if the word is present there—that is, if there is a cache hit.

In the *write-back* method, also called *copy-back*, the CPU performs a write only to the cache in the case of a cache hit. If there is a miss, the CPU performs a write to main memory. There are two possible design choices for when a cache miss occurs. One is to read the line containing the word to be written from main memory into the cache, with the new word written into both the cache and main memory. This is referred to as *write-allocate*. It is done with the hope that there will be additional writes to the same block which will result in write hits and thus avoid writes to main memory. The other choice on a write miss is simply to write to main memory. In what follows, we will assume that write-allocate is used.

The goal of a write-back cache is to be able to write at the writing speed of the cache whenever there is a cache hit. This avoids having all writes performed at the slower writing speed of main memory. In addition, it reduces the number of accesses to main memory, making it more accessible to DMA, an I/O processor, or another CPU in the system. A disadvantage of write-back is that main memory entries corresponding to words in the cache that have been written are invalid. Unfortunately, this can cause a problem with respect to I/O processors or another CPU in the system accessing the same main memory, due to "stale" data in the memory.

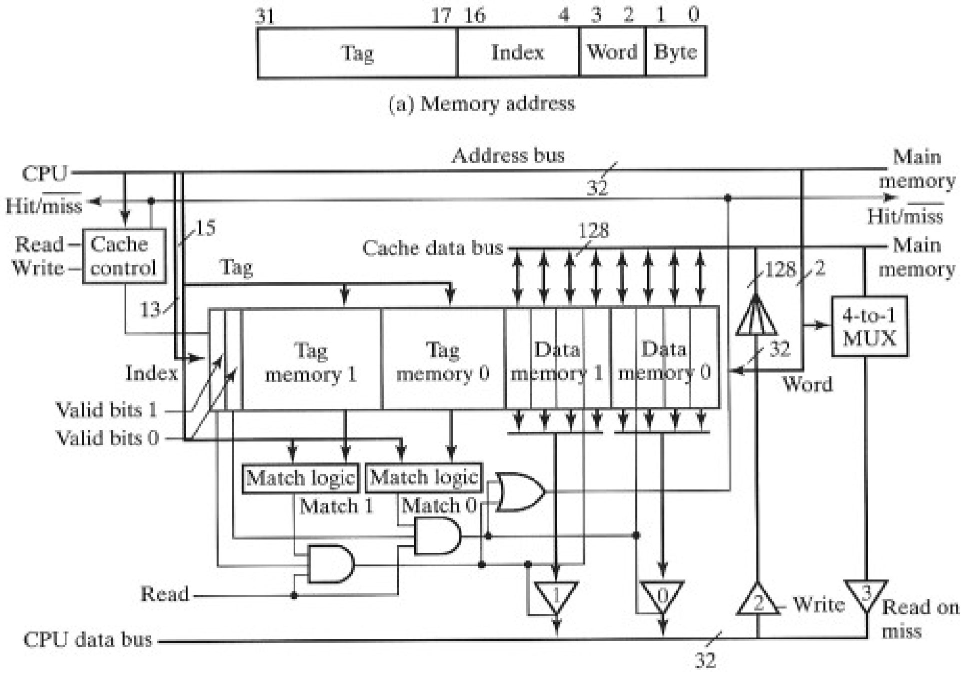
The implementation of the write-back concept requires a write-back operation from the cache location to be used to store a new line being brought from main memory on a read miss. If the location in the cache contains a word that has been written into, then the entire line from the cache must be written back into main memory in order to release the location for the new line. This write-back requires additional time whenever a read miss occurs. To avoid a write-back on every read miss, an additional bit is added to each cache entry. This bit, called the *dirty bit*, is a 1 if the line in the cache has been written and a 0 if it has not been written. Write-back must be performed only if the dirty bit is a 1. With write-allocate used in a write-back cache, a write-back operation may also be required on a write miss.

Many other issues affect the choice of cache design parameters, particularly in the case of caches in a system in which the main memory may be read or written by a device other than the CPU for which the cache is provided.

Integration of Concepts

We now put together the basic concepts we have examined to determine the block diagram for a 256KB, two-way set-associative cache with write-through. The memory address shown in Figure 14-9(a) contains 32 bits using byte addressing with line size l = 16 bytes. The index contains 13 bits. Since 4 bits are used for addressing words and bytes, and 13 bits are used for the index, the tag contains the remaining 15 bits of the 32-bit address. The cache contains 16,384 entries consisting of 2^{13} = 8192 sets. Each cache entry contains 16 bytes of data, a 15-bit tag, and a valid bit. The replacement strategy is random replacement.

Figure 14-9(b) gives the block diagram for the cache. There are two data memories and two tag memories, since the cache is two-way set associative. Each of these memories contains 2^{13} = 8192 entries. Each entry in the data memory consists of 16 bytes. Since 32-bit words are assumed, there are four words in each data memory entry. Thus, each of the data memories consists of four 8192 × 32 memories in parallel with the index as their common address. In order to read a single word from these four memories on a cache hit, a 4-to-1 selector using three-state memory outputs selects the word, based on the two bits in the Word field of the address. The two tag memories are 8192 × 15; in addition to them, a valid bit is associated with each cache entry. These bits are stored in an 8192 × 2 memory and read out during a cache access with the data and tags. Note that the path between the cache and main memory is 128 bits wide. This allows us to assume that an entire cache line can be read from main memory in a single main memory cycle, an assumption that does not necessarily hold in practice. To understand the elements



(a) Memory address

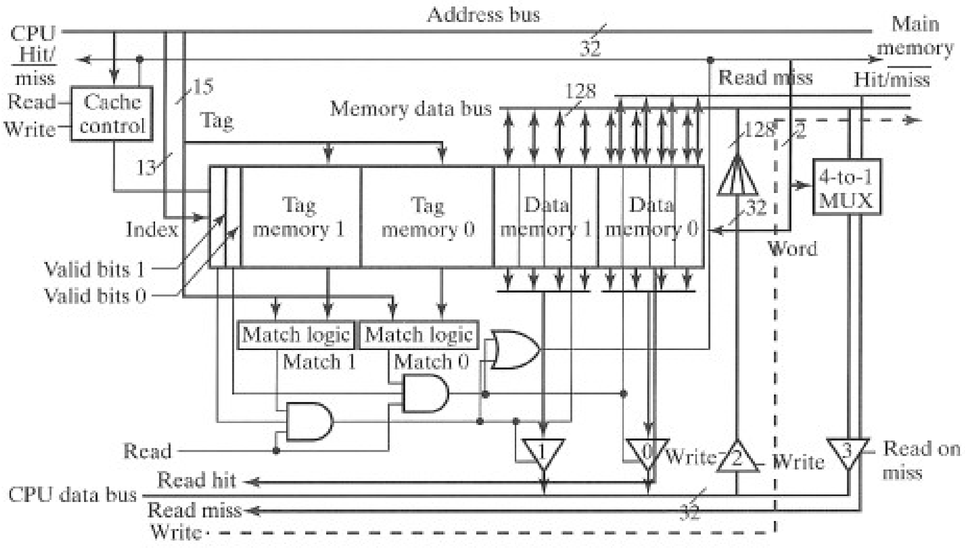
(b) Cache diagram

□ **FIGURE 14-9**
Detailed Block Diagram for 256K Cache

of the cache and how they work together, we will look at three possible cases of reading and writing. For each of these cases, we assume that the address from the CPU is $0F3F4024_{16}$. This gives Tag = 000011110011111_2 = $079F_{16}$, Index = 1010000000010_2 = 1402_{16}, and Word = 01_2.

First we assume a read hit—a read operation in which the data word lies in a cache entry, as in Figure 14-10. The cache uses the Index field to read out two tag entries from location 1402_{16} in Tag memory 1 and Tag memory 0. The match logic compares the tags of the entries, and in this case we assume that Tag 0 matches, causing Match 0 to be 1. This does not necessarily mean that we have a hit, since the cache entry may be invalid. Thus, the Valid 0 from location 1402_{16} bit is ANDed with Match 0. Also, the data can be placed on the CPU data bus only if the operation is a read. Thus, Read is ANDed with the Match 0 bit and the Valid 0 bit to form the control signal for three-state buffer 0. In this case, the control signal for the buffer 0 is 1. The data memories have used the Index field to read out eight words from location 1402_{16} at the same times the tags were read. The Word field selects the two of the eight words with word = 01_2 to place on the data buses going into the three-state buffers 1 and 0. Finally, with three-state buffer 0 turned on, the word addressed is placed on the CPU data bus. Also, the Hit/miss signal sends a 1 to the CPU and the main memory, notifying them of the hit.

In the second case, also shown in Figure 14-10, we assume a read miss—a read operation in which the data word is not in a cache entry. As before, the Index field address reads out the tag and valid entries, two tag comparisons are made, and two valid bits are checked. For both entries, a miss has occurred and is signaled



□ **FIGURE 14-10**
256K Cache: Read and Write Operations

by Hit/$\overline{\text{miss}}$ at 0. This means that the word must be fetched from main memory. Accordingly, the cache control selects the cache entry to be replaced, and four words read from main memory are applied simultaneously by the memory data bus to the cache inputs and are written into the cache entry. At the same time, the 4-to-1 multiplexer selects the word addressed by the Word field and places it on the CPU data bus using the three-state buffer 3.

In the third case in Figure 14-10, we assume a write operation. The word from the CPU is fanned out to appear in all four of the word positions of the 128-bit memory data bus. The address to which the word is to be written is provided by the address bus to main memory for the write operation into the addressed word only. If the address causes a hit on the cache, the word addressed is also written into the cache.

Instruction and Data Caches

In most of the designs in previous chapters, we assumed that it was possible to fetch an instruction and to read an operand or write a result in the same clock cycle. To do this, however, we need a cache that can provide access to two distinct addresses in a single clock cycle. In response to this need, we discussed in a prior subsection an *instruction cache* and a *data cache*. In addition to easily providing multiple accesses per clock, the use of two caches permits caches that have different design parameters. The design parameters for each cache can be selected to fit the different characteristics of access for fetching instructions or reading and writing data. Because the demands on each of these caches are typically less than those on a single cache, a simpler design can be used. For example, a single cache may require a four-way set-association structure, whereas an instruction cache needs only direct mapping, and a data cache may need only a two-way set-associative structure.

In other instances, a single cache for both instructions and data may be used. Such a *unified cache* is typically as large as the instruction and data caches combined. The unified cache allows cache entries to be shared by instructions and data freely. Thus, at one time more entries can be occupied by instructions, and at another time more entries can be occupied by data. This flexibility has the potential for increasing the number of cache hits. This higher hit rate may be misleading, however, since the unified cache supports only one access at a time, and separate caches support two simultaneous accesses as long as one is for instructions and one is for data.

Multiple-Level Caches

It is possible to extend the depth of the memory hierarchy by adding additional levels of cache. Two levels of cache, often referred to as L1 and L2, with L1 closest to the CPU, are often used. In order to satisfy the demand of the CPU for instruction and operands, a very fast L1 cache is needed. To achieve the necessary speed, the delay that occurs when crossing IC boundaries is intolerable. Thus, the L1 cache is placed in the processor IC together with the CPU and is referred to as the *internal cache*, as in the generic computer processor. But the area in the IC is limited, so the L1 cache is typically small and inadequate if it is the only cache. Thus, a larger L2 cache is added outside of the processor IC.

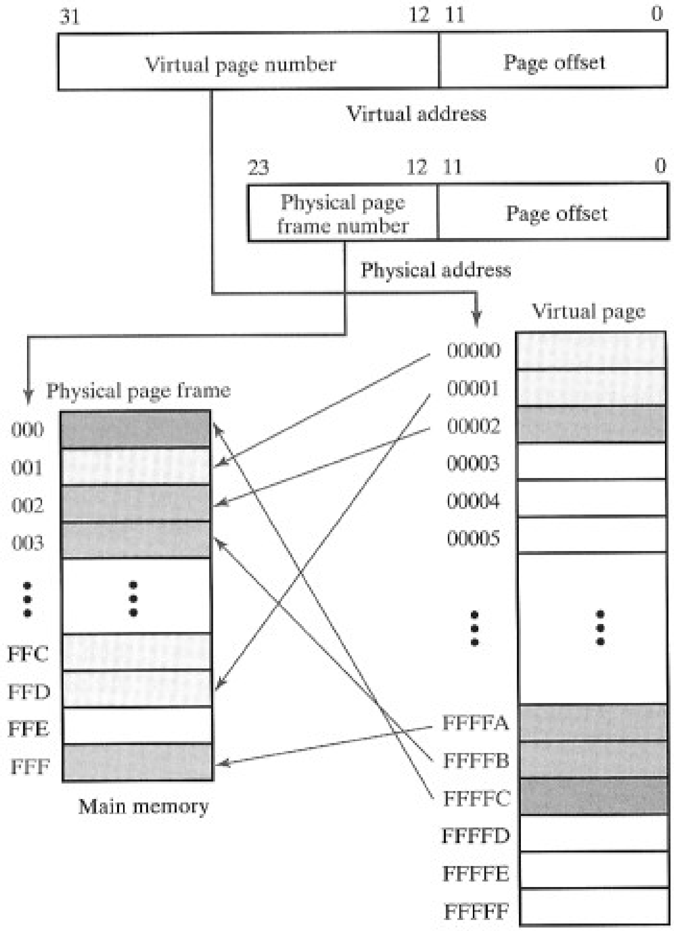
The design of a two-level cache is more complex than that of a single-level cache. Two sets of parameters are specified. The L1 cache can be designed to specific CPU access needs including the possibility of separate instruction and data caches. Also, the constraint of external pins between the CPU and L1 cache is removed. In addition to permitting faster reads, the path between the CPU and the L1 cache can be quite wide, allowing, for example, multiple instructions to be fetched simultaneously. On the other hand, the L2 cache occupies the typical external cache environment. It differs, however, from the typical external cache in that, rather than providing instructions and operands to a CPU, it primarily provides instructions and operands to the first-level cache L1. Since the L2 cache is accessed only on L1 misses, the access pattern is considerably different than that for a CPU, and the design parameters are accordingly different.

14-4 Virtual Memory

In our quest for a large, fast memory, we have achieved the appearance of a fast, medium-sized memory through the use of a cache. In order to have the appearance of a large memory, we now explore the relationship between main memory and hard disk. Because of the complexity of managing transfers between these two media, the control of such transfers involves the use of data structures and programs. Initially, we will discuss the most basic data structure used and the necessary hardware and software actions. Then we will deal with special hardware used to implement time-critical hardware actions.

With respect to large memory, not only do we want the entire virtual address space to appear to be main memory, but in most cases we would also like this complete space to appear to be available to each program that is executing. Thus, each program will "see" a memory the size of the virtual address space. Equally important to the programmer is the fact that real address space in main memory and real disk addresses are replaced by a single address space that has no restrictions on its use. With this arrangement, virtual memory can be used not only to provide the appearance of large main memory, but also to free up the programmer from having to consider the actual locations of the program and data in main memory and on the hard disk. The job of the software and hardware that implement virtual memory is to map each *virtual address* for each program into a *physical address* in the main memory. In addition, with a virtual address space for each program, it is possible for a virtual address from one program and a virtual address from another program to map to the same physical address. This allows code and data to be shared by multiple programs, thereby reducing the size of the main memory space and disk space required.

To permit the software to map virtual addresses to physical addresses, and to facilitate the transfer of information between main memory and hard disk, the virtual address space is divided into blocks of addresses, typically of a fixed size. These blocks, called *pages*, are larger than, but analogous to, lines in a cache. The physical address space in memory is divided into blocks called *page frames* that are the same size as the pages. When a page is present in the physical address space, it occupies a page frame. For purposes of illustration, we assume that a page consists of 4K bytes (1K words of 32 bits). Further, we assume that there are 32 address bits in the virtual address space. There are 2^{20} pages, maximum, in the virtual address space, and assuming a main memory of 16M bytes, there are 2^{12} page frames in main memory. Figure 14-11 shows the fields of virtual and physical addresses. The portion of the virtual address used to address words or bytes within a page is the *page offset*, which is the only part of the address that the virtual and physical addresses share. Note that words are assumed to be aligned in terms of their location with respect to their byte addresses such that each word address ends in



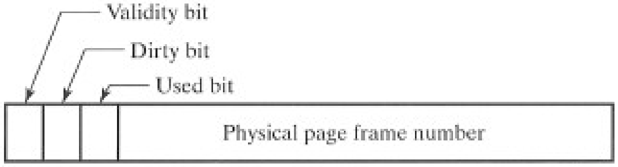
□ **FIGURE 14-11**
Virtual and Physical Address Fields and Mapping

binary 00. Likewise, pages are assumed to be aligned with respect to the byte addresses such that the page offset of the first byte in the page is 000_{16} and the page offset of the last byte in the page is FFF_{16}. The 20-bit portion of the virtual address used to select pages from the virtual address space is the *virtual page number*. The 12-bit portion of the physical address used to select pages in main memory is the *page frame number*. The figure shows a hypothetical mapping from the virtual address space into the physical address space. The virtual and physical page numbers are given in hexadecimal. A virtual page can be mapped to any physical page frame. Six mappings of pages from virtual memory to physical memory are shown. These pages constitute a total of 24K bytes. Note that there are no virtual pages mapped to physical page frames FFC and FFE. Thus, any data present in these pages is invalid.

Page Tables

In general, there may be a very large number of virtual pages, each of which must be mapped to either main memory or hard disk. The mappings are stored in a data structure called a *page table*. There are many ways to structure page tables and access them; we assume that page tables themselves are also kept in pages. Assuming that the representation of each mapping requires one word, 2^{10}, or 1K, mappings can be contained in a 4 KB page. Thus, the mappings for the entire address space for a program of 2^{22} bytes (4 MB) can be contained in one 4 KB page. A special table for each program called a *directory page* provides the mappings used to locate the 4 KB program page tables.

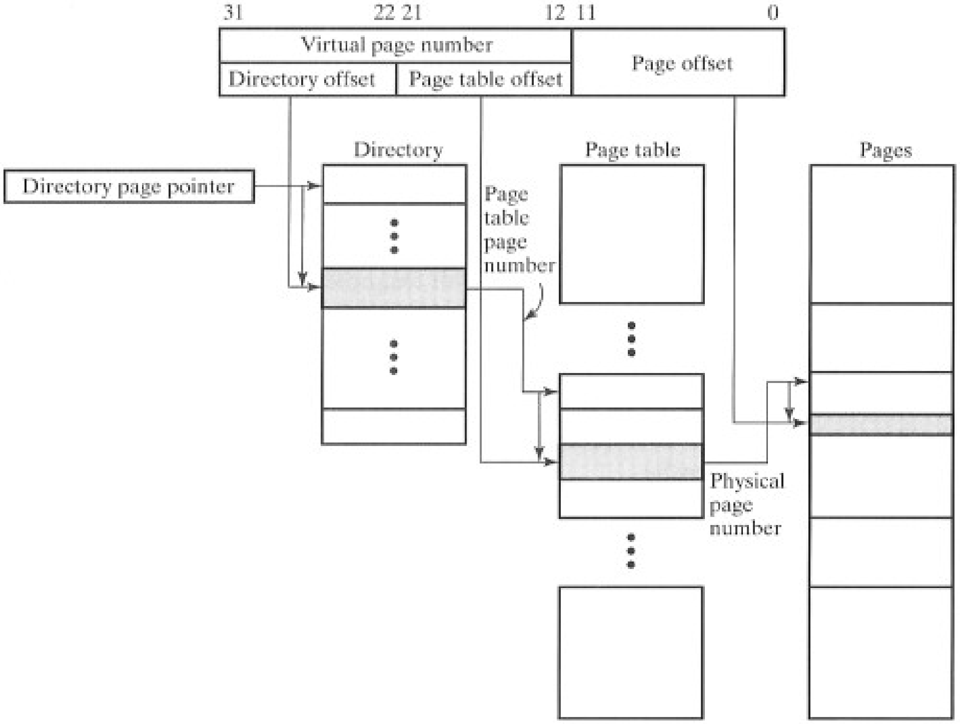
A sample format for a page table entry is given in Figure 14-12. Twelve bits are used for the page frame number in which the page is located in main memory. In addition, there are three single bit fields: Valid, Dirty, and Used. If Valid is 1, then the page frame in memory is valid; if Valid is 0, the page frame in memory is invalid, meaning that it does not correspond to correct code or data. If Dirty is 1, then there has been a write to at least one byte in the page since it was placed in main memory. If Dirty is 0, there have been no writes to the page since it entered main memory. Note that the Valid and Dirty bits correspond exactly to those in a cache which uses write-back. When it is necessary for a page to be removed from main memory and the Dirty bit is 1, then the page is copied back to the hard disk. If the Dirty bit is 0, indicating that the page in main memory has not been written into, then the page coming into the same page frame is



□ **FIGURE 14-12**
Format for Page Table Entries

simply written over the present page. This can be done because the disk version of the present page is still correct. In order to use this feature, the software keeps a record of the location of the page on the disk elsewhere when it places the page in main memory. The Used bit is a simple mechanism for implementing a crude approximation to an LRU replacement scheme. Some additional bit positions in a page entry may be reserved for flags used by the computer operating system. For example, a few flags might represent the read and write protection status of a page and whether the page can be accessed in user mode or supervisor mode.

The page table structure we have just described is shown in Figure 14-13. The *directory page pointer* is a register that points to the location of the directory page in main memory. The directory page contains the locations of up to 1K page tables associated with the program that is executing. These page tables may be in main memory or on the hard disk. The page table to be accessed is derived from the most significant 10 bits of the virtual page number, which we call the *directory offset*. Assuming that the page table selected is in main memory, it can be accessed by the *page table page number*. The least significant 10 bits of the virtual page number, which we call the *page table offset*, can be used to access the entry for the page to be accessed. If the page is in main memory, the page offset is used to



□ **FIGURE 14-13**
Example of Page Table Structure

locate the physical location of the byte or word to be accessed. If either the page table or the desired page is not in main memory, it must first be fetched by software from the hard disk to main memory before the word within it is accessed. Note that combining the offsets with register or table entries is done by simply setting the offset to the right of the page frame number, rather than adding the two together. This approach requires no delay, whereas addition would cause significant delay.

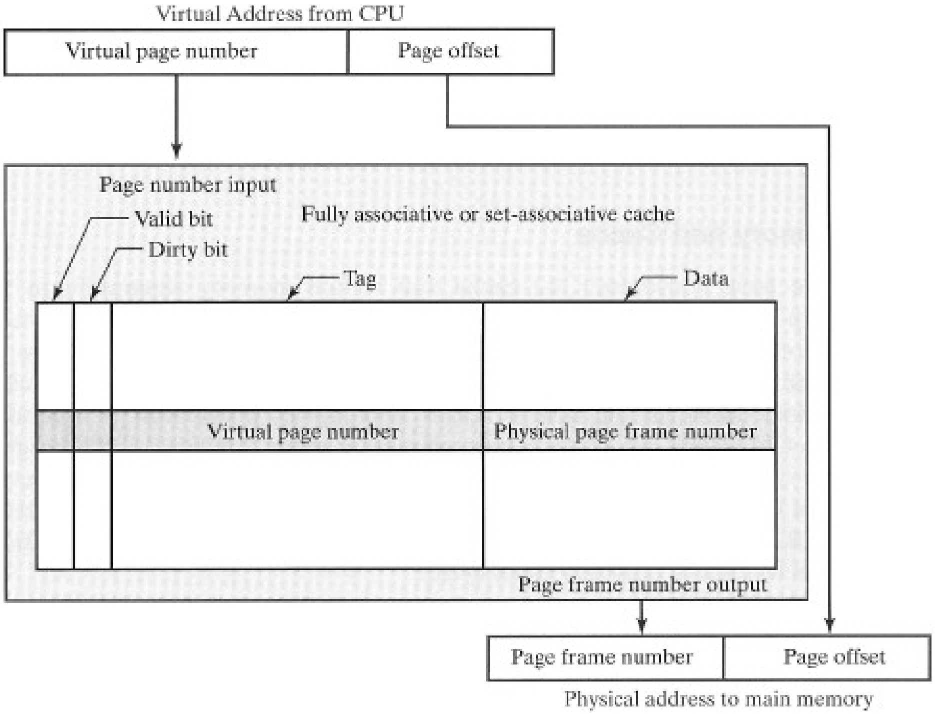
Translation Lookaside Buffer

From the preceding discussion, we note that virtual memory has a considerable performance penalty even in the best case, when the directory, the page table, and the page to be accessed are in main memory. For our assumed page table approach, three successive accesses to main memory occur in order to fetch a single operand or instruction:

1. Access for the directory entry.
2. Access for the page table entry.
3. Access for the operand or instruction.

Note that these accesses are performed automatically by hardware that is part of the MMU in the generic computer. Thus, to make virtual memory feasible, we need to drastically reduce accesses to main memory. If we have a cache, and if all of the entries are in the cache, then the time for each access is reduced. Nevertheless, three accesses are needed to the cache. To reduce the number of accesses, we will employ yet another cache for the purpose of translating the virtual address directly into a physical address. This new cache is called a *translation lookaside buffer* (TLB). It holds the locations of recently addressed pages to speed access to cache or main memory. Figure 14-14 gives an example of a TLB, which is typically fully associative or set associative, since it is necessary to compare the virtual page number from the CPU with a number of virtual page number tags. In addition to the latter, a cache entry includes the physical page number for those pages in main memory and a Valid bit. If the page is in main memory, the Dirty bit also appears. The Dirty bit serves the same function for a page in main memory as discussed previously for a line in a cache.

We now briefly look at a memory access using the TLB in Figure 14-14. The virtual page number is applied to the page number input to the cache. Within the cache, this page number is compared simultaneously with all of the virtual page number tags. If a match occurs and the Valid bit is a 1, then a TLB hit has occurred, and the physical page frame number appears on the page number output of the cache. This operation can be performed very quickly and produces the physical address required to access memory or a cache. On the other hand, if there is a TLB miss, then it is necessary to access main memory for the directory table entry and the page table entry. If there is a physical page in main memory, then the page table entry is brought into the TLB cache and replaces one of the entries there.



□ **FIGURE 14-14**
Example of Translation Lookaside Buffer

Overall, three memory accesses are required, including the one for the operand. If the physical page does not exist in main memory, then a *page fault* occurs. In this case, a software-implemented action fetches the page from its hard disk location to main memory. During the time required to complete this action, the CPU may execute a different program rather than waiting until the page has been placed in main memory.

Noting the prior hierarchy of actions based on the presentation of a virtual address, we see that the effectiveness of virtual memory depends on temporal and spatial locality. The fastest response is possible when the virtual page number is present in the TLB. If the hardware is fast enough and a hit also occurs on the cache, the operand can be available in as little as one or two CPU clock cycles. Such an event is likely to happen frequently if the same virtual pages tend to get accessed over time. Because of the size of the pages, if one operand is accessed from a page, then, due to spatial locality, it is likely that another operand will be accessed on the same page. With the limited capacity of the TLB, the next fastest action requires three accesses to main memory and slows processing considerably. In the worst of all situations, the page table and the page to be accessed are not in main memory. Then, lengthy transfers of two pages—the page table and the page from hard disk—are required.

Note that the basic hardware for implementing virtual memory, the TLB, and other optional features for memory access are included in the MMU in the generic computer. Among the other features is hardware support for an additional layer of virtual addressing called *segmentation* and for protection mechanisms to permit appropriate isolation and sharing of programs and data.

Virtual Memory and Cache

Although we have considered the cache and virtual memory separately, in an actual system they are both very likely to be present. In that case, the virtual address is converted to the physical address, and then the physical address is applied to the cache. Assuming that the TLB takes one clock cycle and the cache takes one clock cycle, in the best of cases fetching an instruction or operand requires two CPU clock cycles. As a consequence, in many pipelined CPU designs, two or more clock cycles are allowed for an operand fetch. Since instruction fetch addresses are more predictable, it is possible to modify the CPU pipeline and consider the TLB and cache to be a two-stage pipeline segment, so that an instruction fetch appears to require only one clock cycle.

14-5 Chapter Summary

In this chapter, we examined the components of a memory hierarchy. Two concepts fundamental to the hierarchy are cache memory and virtual memory.

Based on the concept of locality of reference, a cache is a small, fast memory that, holds the operands and instructions most likely to be used by the CPU. Typically, a cache gives the appearance of a memory the size of main memory with a speed close to that of the cache. A cache operates by matching the tag portion of the CPU address with the tag portions of the addresses of the data in the cache. If a match occurs and other specific conditions are satisfied, a cache hit occurs, and the data can be obtained from the cache. If a cache miss occurs, the data must be obtained from the slower main memory. The cache designer must determine the values of a number of parameters, including the mapping of main memory addresses to cache addresses, the selection of the line of the cache to be replaced when a new line is added, the size of the cache, the size of the cache line, and the method for performing memory writes. There may be more than one cache in a memory hierarchy, and instructions and data may have separate caches.

Virtual memory is used to give the appearance of a large memory—much larger than the main memory—at a speed that is, on average, close to that of the main memory. Most of the virtual address space is actually on hard disk. To facilitate the movement of information between the memory and the hard disk, both are divided up in fixed size address blocks called page frames and pages, respectively. When a page is placed in main memory, its virtual address must be translated to a physical address. The translation is done using one or more page tables. In order to perform the translation on each memory access without a severe performance penalty, special hardware is employed. This hardware, called a translation lookaside buffer (TLB), is a special cache that is a part of the memory management unit (MMU) of the computer.

Together with main memory, the cache and the TLB give the illusion of a large, fast memory that is, in fact, a hierarchy of memories of different capacities, speeds, and technologies, with hardware and software performing automatic transfers between levels.

References

1. Mano, M. M. *Computer Engineering: Hardware Design.* Englewood Cliffs, NJ: Prentice Hall, 1988.
2. Hennessy, J. L., and D. A. Patterson *Computer Architecture: A Quantitative Approach.* San Francisco, CA: Morgan Kaufmann, 1996.
3. Baron, R. J., and L. Higbie *Computer Architecture.* Reading, MA: Addison-Wesley, 1992.
4. Handy, J. *Cache Memory Book.* San Diego: Academic Press, 1993.
5. Mano, M. M. *Computer System Architecture,* 3rd Ed. Englewood Cliffs, NJ: Prentice Hall, 1993.
6. Patterson, D. A., and J. L. Hennessy *Computer Organization and Design: The Hardware/Software Interface.* San Francisco, CA: Morgan Kaufmann, 1998.
7. Wyant, G., and T. Hammerstrom *How Microprocessors Work.* Emeryville, CA: Ziff-Davis Press, 1994.
8. Messmer, H. P., *The Indispensable PC Hardware Book,* 2nd ed. Wokingham, U.K.: Addison-Wesley, 1995.

Problems

The plus (+) indicates a more advanced problem and the asterisk (*) indicates a solution is available on the Companion Website for the text.

14–1. *A CPU produces the following sequence of read addresses in hexadecimal:
54, 58, 104, 5C, 108, 60, F0, 64, 54, 58, 10C, 5C, 110, 60, F0, 64
Supposing that the cache is empty to begin with, and assuming an LRU replacement, determine whether each address produces a hit or a miss for each of the following caches: **(a)** direct mapped in Figure 14-3, **(b)** fully associative in Figure 14-4, and **(c)** two-way set associative in Figure 14-6.

14–2. Repeat Problem 14-1 for the following sequence of read addresses:
0, 4, 8, 12, 14, 1A, 1C, 26, 28, 2E, 30, 36, 38, 3E, 40, 46, 48, 4E, 50, 56, 58, 5E

14–3. Repeat problem 14-1 for the following sequence of read addresses in hexadecimal: 20, 04, 28, 60, 20, 04, 28, 4C, 10, 6C, 70, 10, 60, 70

14–4. *A computer has a 32-bit address and a direct-mapped cache. Addressing is to the byte level. The cache has a capacity of 1K bytes and uses lines that are 32 bytes. It uses write-through and so does not require a dirty bit.

(a) How many bits are in the index for the cache?

(b) How many bits are in the tag for the cache?

(c) What is the total number of bits of storage in the cache, including the valid bits, the tags, and the cache lines?

14–5. A two-way set-associative cache in a system with 24-bit addresses has two 4-byte words per line and a capacity of 512K bytes. Addressing is to the byte level.

(a) How many bits are there in the index and the tag?

(b) Indicate the value of the index in hexadecimal for cache entries from the following main memory addresses in hexadecimal: 82AF82, 14AC89, 48CF0F and3ACF01.

(c) Can all of the cache entries from part (b) be in the cache simultaneously?

14–6. *Discuss the advantages and disadvantages of:

(a) separate instruction and data caches versus a unified cache for both.

(b) a write-back cache versus a write-through cache.

14–7. Give an example of a sequence of program and data memory read addresses that will have a high hit rate for separate instruction and data caches and a low hit rate for a unified cache. Assume direct mapped caches with the parameters in Figure 14-3. Both the instructions and data are 32-bit words and the address resolution is to bytes.

14–8. *Give an example of a sequence of program and data memory read addresses that will have a high hit rate for a unified cache and a low hit rate for separate instruction and data caches. Assume that each of the instruction and data caches is two-way set associative with parameters as in Figure 14-6. Assume that the unified cache is four-way set associative with parameters as in Figure 14-6. Both the instructions and the data are 32-bit words and the address resolution is to bytes.

14–9. Explain why write-allocate is typically not used in a write-through cache.

14–10. A high-speed workstation has 64-bit words and 64-bit addresses with address resolution to the byte level.

(a) How many words can be in the address space of the workstation?

(b) Assuming a direct-mapped cache with 8192 32-byte lines, how many bits are in each of the following address fields for the cache: (1) Byte, (2) Index, and (3) Tag?

14–11. *A cache memory has an access time from the CPU of 4 ns, and the main memory has an access time from the CPU of 40 ns. What is the effective access time for the cache–main memory hierarchy if the hit ratio is: **(a)** 0.91, **(b)** 0.82, and **(c)** 0.96?

14–12. Redesign the cache in Figure 14-7 so that it is the same size, but is four-way set associative rather than two-way set associative.

14–13. +The cache in Figure 14-9 is to be redesigned to use write-back with write-allocate rather than write-through. Respond to the following requests, making sure to deal with all of the address and data issues involved in the write-back operation.

(a) Draw the new block diagram.

(b) Explain the sequence of actions you propose for a write miss and for a read miss.

14–14. *A virtual memory system uses 4K byte pages, 64-bit words, and a 48-bit virtual address. A particular program and its data require 4263 pages.

(a) What is the minimum number of page tables required?

(b) What is the minimum number of entries required in the directory page?

(c) Based on your answers to (a) and (b), how many entries are there in the last page table?

14–15. A small TLB has the following entries for a virtual page number of length 20 bits, a physical page number of 12 bits, and a page offset of 12 bits.

Valid bit	Dirty bit	Tag (Virtual Page Number)	Data (Physical Page Number)
1	1	01AF4	FFF
0	0	0E45F	E03
0	0	012FF	2F0
1	0	01A37	788
1	0	02BB4	45C
0	1	03CA0	657

The page numbers and offset are given in hexadecimal. For each of the virtual addresses listed, indicate whether a hit occurs and if it does, give the physical address: (a) 02BB4A65, (b) 0E45FB32, (c) 0D34E9DC, and (d) 03CA0777.

14–16. A computer can accommodate a maximum of 384M bytes of main memory. It has a 32-bit word and a 32-bit virtual address and uses 4K byte pages. The TLB contains only entries that include the Valid, Dirty, and Used bits, the virtual page number, and the physical page number. Assuming that the TLB is fully associative and has 32 entries, determine the following:

(a) How many bits of associative memory are required for the TLB?

(b) How many bits of SRAM are required for the TLB?

14–17. Four programs are concurrently executing in a multitasking computer with virtual memory pages having 4K bytes. Each page table entry is 32 bits.

What is the minimum numbers of bytes of main memory occupied by the directory pages and page tables for the four programs if the numbers of pages per program, in decimal, are as follows: 6321, 7777, 9602, and 3853.

14–18. *In caches, we use both write-through and write-back as potential writing approaches. But for virtual memory, only an approach that resembles write-back is used. Give a sound explanation of why this is so.

14–19. Explain clearly why both the cache memory concept and the virtual memory concept would be ineffective if locality of reference of memory-addressing patterns did not hold.

INDEX