# CHAPTER 4

# Control Unit

In this chapter, the process of designing a control unit is discussed. Topics covered include: hardwired control, microprogrammed control, and control store–minimization methods.

## 4.1 INTRODUCTION

A CPU can be viewed as a collection of two major components:

- Processing section
- Control unit

The processing section typically includes the hardware elements (ALU, shift registers, comparators, and multipliers) to operate on data such as integer or real numbers, character codes, operation codes, and offset values.

The purpose of the control unit is to control system operations by routing the selected data items to the selected processing hardware at the right time. A control unit's responsibility is to drive the associated processing hardware by generating a set of signals that are synchronized with a master clock. This synchronization establishes a time-reference for system analysis and design.

The signals generated by a control unit actually initiate an operation within the system. In order to carry out a task such as ADD, the control unit must generate a set of control signals in a predefined sequence governed by the hardware structure of the processing section. Sometimes, the control unit decides the operation sequence based on the status information provided by the status register (for example, Z and N condition codes) or the command signals generated by an external agent (such as RESET and ABORT signals issued from the operator's console). The inputs to the control unit are the master clock, status information from the processing section, and command signals from the external agent. The outputs produced by the typical control unit are the signals that drive the processing section and responses to an external environment (operation complete and operation aborted) due to exceptions (integer overflow or underflow).

A control unit undertakes the following responsibilities:

- Instruction interpretation
- Instruction sequencing

In the interpretation phase, the control unit reads instructions from the memory (using the PC as a pointer). It then recognizes the instruction type, gets the necessary operands, and routes them to the appropriate functional units of the execution unit. Necessary signals are then issued to the execution unit to perform the desired operation, and the results are routed to the specified destination.

In the sequencing phase, the control unit determines the address of the next instruction to be executed and loads it into the PC. To design a control unit, one must know the basic concepts addressed in the next section.

## 4.2 BASIC CONCEPTS

The fundamental concepts forming the basis for control unit design are the register transfer operations and their analytical descriptions. A digital processing system is a collection of registers where a processing activity is performed by a sequence of data-transfer operations among the registers either directly or with processing hardware (ALU). Consider the simple transfer shown in Figure 4.1

Here, 8 bits of information are to be moved from register A to register B. This means transferring a copy of A to B.

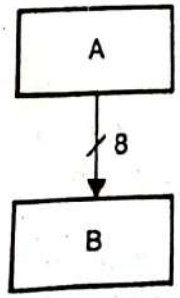



**Figure 4.1** A Simple Register Transfer from A to B

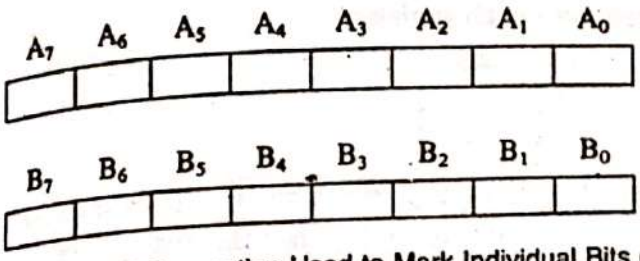



**Figure 4.2** Convention Used to Mark Individual Bits of a Register

Such an operation can be described by the following notation:

$B \leftarrow A$

The symbol ← is called the transfer operator. The statement B ← A does not indicate how many bits are transferred. To indicate this, a declaration statement is used, which specifies the size of each register. For example, the registers shown in Figure 4.2 can be declared as

Declare registers A [8], B [8]

In this declaration A and B are 8-bit registers.

Similarly, a register can be defined as a portion of some other register. For example, if the high-order byte of a 16-bit program counter is considered as one 8-bit register, the following declarations are made:

Declare register PC [16]

Declare subregisters PCHI [8] = PC [15-8]

The convention used to mark individual bits of a register is shown in Figure 4.2. In this figure, the bit positions are numbered as bits 0, 1, 2, . . . , 7. The binary weighting for bit 0 is $2^0$, bit 1 is $2^1$, and so on.

Consider the following statement:

$B[0] \leftarrow A[7]$

This means that the most-significant bit of the A register is transferred to the least-significant bit of the B register.

Normally, the information transfer between two registers is controlled by an enable signal E, as shown in Figure 4.3.

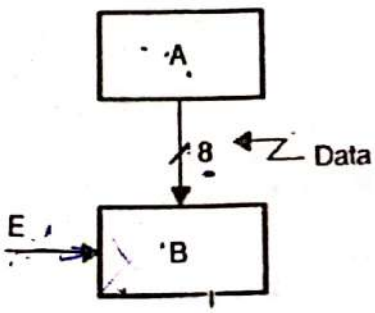



**Figure 4.3** A Register Transfer Controlled by an Enable Input

Generally two inputs are associated with each register:

- Enable input or control input
- Data input

The enable input controls the data flow from register A to B. The B register of Figure 4.3 is loaded with the contents of the A register only when the enable input E is held high; otherwise the contents of the register remain the same. Such a conditional transfer can be expressed as

$$E: B \leftarrow A$$

A possible hardware implementation of a register with an enable input is shown in Figure 4.4.

The bit $B_i$ in this figure is only loaded with bit $A_i$ when the enable input E is high. The purpose of the enable input is to make sure that data transfer between the A and B registers takes place only under a predetermined condition and not after every clock pulse. For this reason, this input is called the *control input* and is driven by the control unit.

Sometimes, the control input may be a function of more than one variable. Consider the following register transfer involving three 8-bit registers A, B, and D:

$$\text{IF } A > B \text{ and } D[0] = 0 \text{ then } A \leftarrow B$$

The condition $A > B$ can be determined by using an 8-bit comparator. If we assume that the comparator output G goes high when $A > B$, then this conditional transfer can be described as follows:

$$C_0: A \leftarrow B \quad \text{where} \quad C_0 \equiv G \wedge D[0]'$$

The hardware setup corresponding to this situation is shown in Figure 4.5.

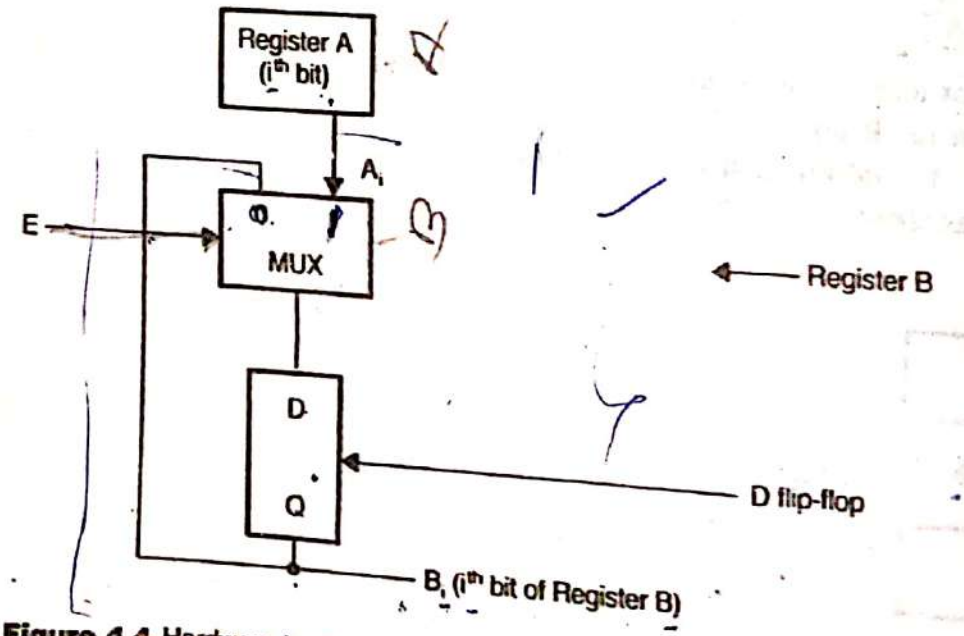



**Figure 4.4** Hardware Implementation of a Register with Enable Input

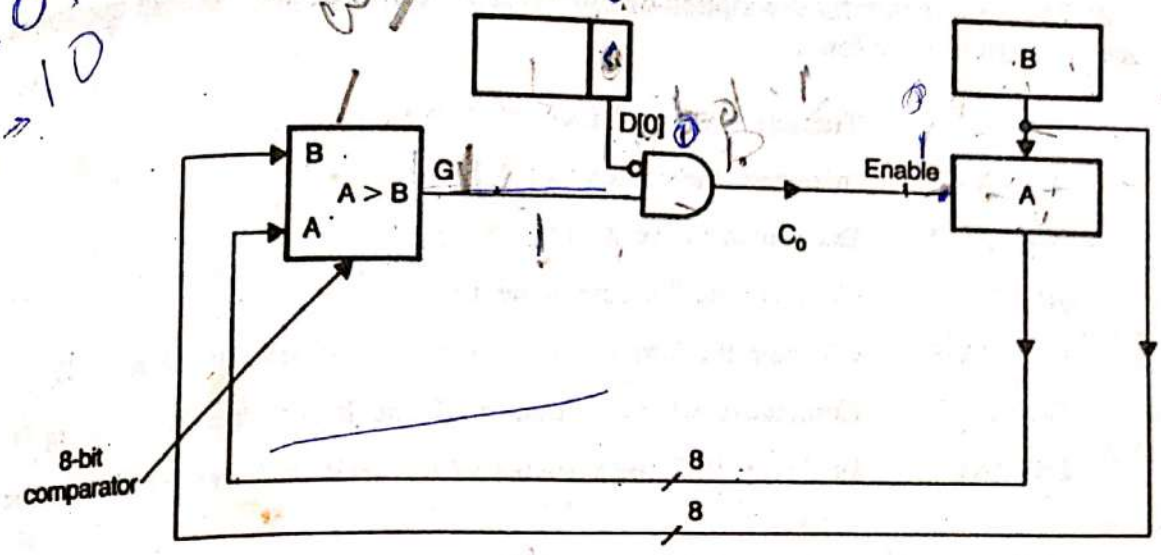



**Figure 4.5** Hardware Implementation of $C_0$: A ← B where $C_0$ = G D [0]'

It may often be necessary to perform some register transfer operation that involves selection. For example:

if x = 0 and t = 1 then A ← B
else A ← D

Where A, B, and D are 8-bit registers, such a selective register transfer can be expressed as follows:

$C_0$ : A ← B

$C_0'$ : A ← D

where $C_0 = x't$ and $C_0' = (x't)' = x + t'$.

A hardware implementation for the preceding situation is shown in Figure 4.6.

The MUX of Figure 4.6 selects register B if $C_0$ = 1; otherwise register D is selected.

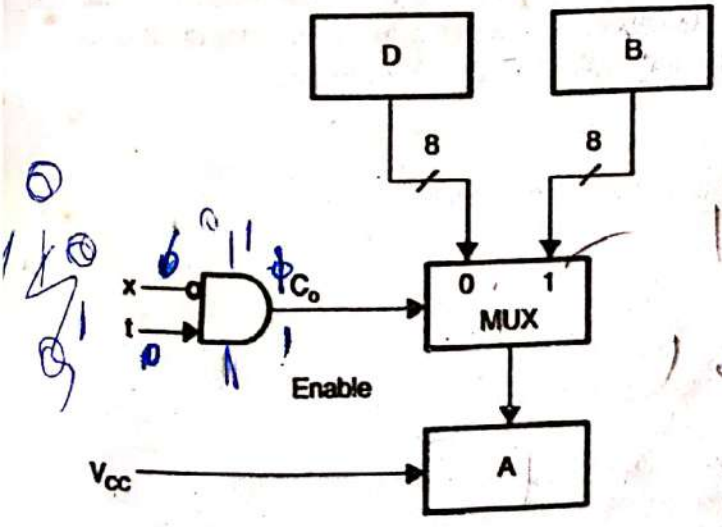



**Figure 4.6** Hardware Implementation if x = 0 and t = 1, then A ← B else A ← D

The register transfer description of some typical arithmetic and Boolean operations are summarized as follows:

| | |
|---|---|
| $D \leftarrow A'$ | Transfer the complement of A to D. |
| $A \leftarrow A+1$ | Increment the contents of A by 1. |
| $A \leftarrow A-1$ | Decrement the contents of A by 1. |
| $A'+1$ | Compute the 2's complement of A. |
| $D \leftarrow A \vee B$ | Compute the logical OR of A and B and save the result in D. |
| $D \leftarrow A \wedge B$ | Compute the logical AND of A and B and save the result in D. |
| LSR (A) | Logically shift the contents of the register A one position to the right. |
| ASR (A) | Arithmetically shift the contents of the register A one position to the right. |

The mnemonics LSL, ASL, ROR, and ROL are used to indicate logical left shift, arithmetic left shift, rotate right and rotate left operations respectively.

For this notation the $ symbol is used as the concatenation operator. For example, if A and Q are two 8-bit registers and ASR (A$Q) is written, the contents of AQ are arithmetically shifted one position to the right. This operation is a 16-bit operation with high- and low-order bytes held in the A and Q registers, respectively.

Whenever a read/write memory (RWM) is a part of the processing section, the data transfers are described with respect to the RWM unit. As mentioned before, each word in the RWM is considered as an addressable register; thus, the whole memory unit can be viewed as a collection of such addressable registers.

Associated with each RWM unit are two registers:

- Memory address register (MAR)
- Memory buffer register (MBR)

The MAR is used as a pointer to a memory word. It holds the address of the desired memory word. The purpose of the MBR is to act as a buffer register in all data transfer operations. The general structure of a RWM is shown in Figure 4.7.

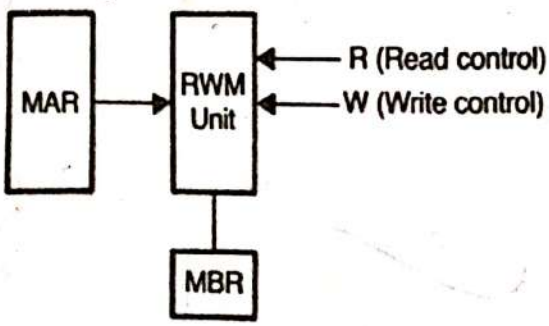



**Figure 4.7** A Basic RWM Unit (M. Morris Mano, *Digital Logic and Computer Design*, © 1979, p. 314. Reprinted by permission of Prentice-Hall, Inc., Englewood Cliffs, New Jersey)

The two basic operations performed by the RWM unit are read and write operations, expressed in register transfer notation as

R: MBR ← M ((MAR))

W: M ((MAR)) ← MBR

The letter M is used to indicate a memory. M ((MAR)) indicates the memory word X addressed by the contents of the MAR.

In a typical read operation, the contents of the memory word, whose address is specified in MAR, are transferred into the MBR. Similarly, a write operation refers to the data transfer from MBR to the memory word whose address is specified in MAR. Since the data flows in and out of MBR, the set of data lines or the data bus between the RAM unit and MBR should be bidirectional. Such a bidirectional bus is easily implemented using tristate buffers as shown in Figure 4.8.

When C = 1 in Figure 4.8, data transfer takes place from X to Y, and vice versa when C = 0.

Buses are also used to route data in and out of a digital processing system. For example, in a typical digital system, there will be a pair of buses ("inbus" and "outbus") to transfer data from the external environment into the processing section, and vice versa. As in the case of registers, the existence of such buses is shown in a register transfer description by using declaration statements. For example, declare Inbus [4] Outbus [4] indicates that the system under description includes two 4-bit-wide data buses (inbus and outbus). A ← inbus means the data on the inbus is transferred into the A register when the next clock arrives. The equate symbol specifies a transfer from a register into the bus. For example, outbus = Q [7:4] means the high-order 4 bits of an 8-bit register Q are made available on the outbus for one clock period.

By sequencing a set of register transfer operations and including typical control structures such as if-then and go-to, an algorithm implemented by a digital system can be described. Consider the following description:

Declare registers A [8,] M [8], Q [8]:

Declare buses inbus [8], outbus [8]:

Start: A ← 0, M ← inbus; Transfer the multiplicand and clear accumulator
Q ← inbus ; Transfer the multiplier

Loop: A ← A + M, Q ← Q − 1; Add multiplicand
If Q < > 0 then go to Loop; repeat if Q ≠ 0
Outbus = A;

Halt: Go to Halt

The hardware for the preceding description includes an 8-bit inbus, an 8-bit outbus, an 8-bit parallel adder, and three 8-bit registers—A, M, and Q. This hardware performs unsigned multiplication by repeated addition, much like a program written using a combination of high-level and assembly languages.

Therefore, a distinguishing feature of this description is the ability to describe concurrent operations. For example, the operations A ← 0, and M ← Inbus can be

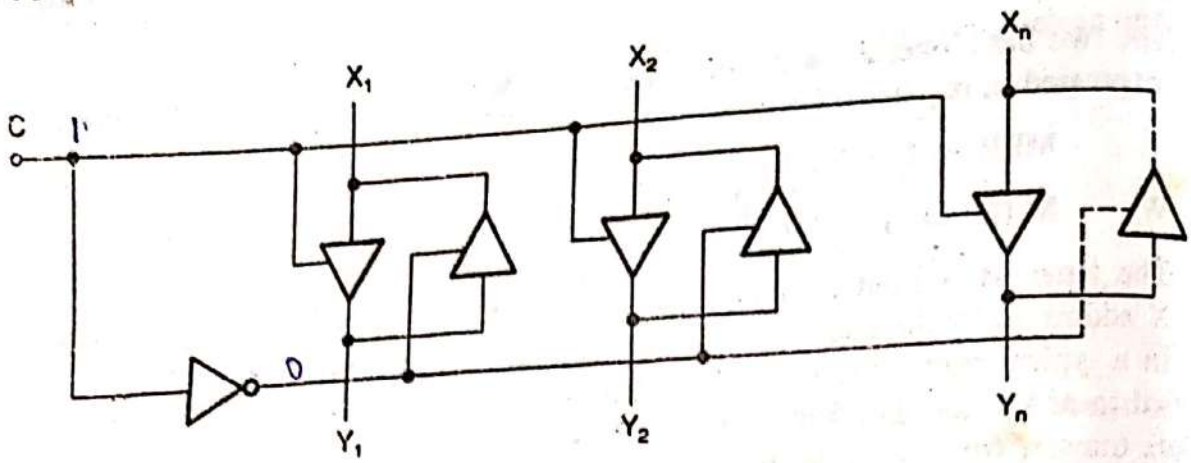



**Figure 4.8** Bidirectional Data Bus

executed at the same time. As a general rule, a comma is inserted between operations that can be executed simultaneously.

Similarly, a semicolon between two register transfer operations dictates that they must be performed serially. This restriction is primarily due to the data paths provided in the hardware. For example, in the description just presented, there is only one input bus. The operations M ← inbus and Q ← inbus must be performed serially. However, either one may be overlapped with the operation A ← 0, since the operation does not use the input bus. The description also includes features such as labels and comments to increase the readability of the task description.

Operations such as A ← 0 and A ← A + M are called *microoperations*, since they are completed in one clock cycle. In principle, any task can be expressed as a sequence of microoperations. This topic is addressed later in this chapter.

The rate which a computer processes operations such as A ← A + B, and A ← A ∧ B is determined by its bus structure. Similarly, bus organization determines the cost of the system. Therefore, a less-complex bus organization is less expensive.

The simplest of all bus structures is the single-bus organization shown in Figure 4.9. All CPU registers in Figure 4.9 are connected to the same bus. At any given time, data may be transferred between any two CPU registers or between a CPU register and the ALU.

Whenever the ALU in Figure 4.9 requires the two operands (such as in the case of addition), the operands can be transferred only one at a time. Single-bus architecture should have the following features:

- The bus must be multiplexed among various operands.
- The ALU must have buffer registers to hold the transferred operand.

In Figure 4.9 an operation such as $R_2 \leftarrow R_0 + R_1$ is completed in three clock cycles because of the following:

- In the first clock cycle, the contents of register $R_0$ are transferred to buffer register A of the ALU.
- During the second clock cycle the contents of register $R_1$ are transferred to buffer register B of the ALU.

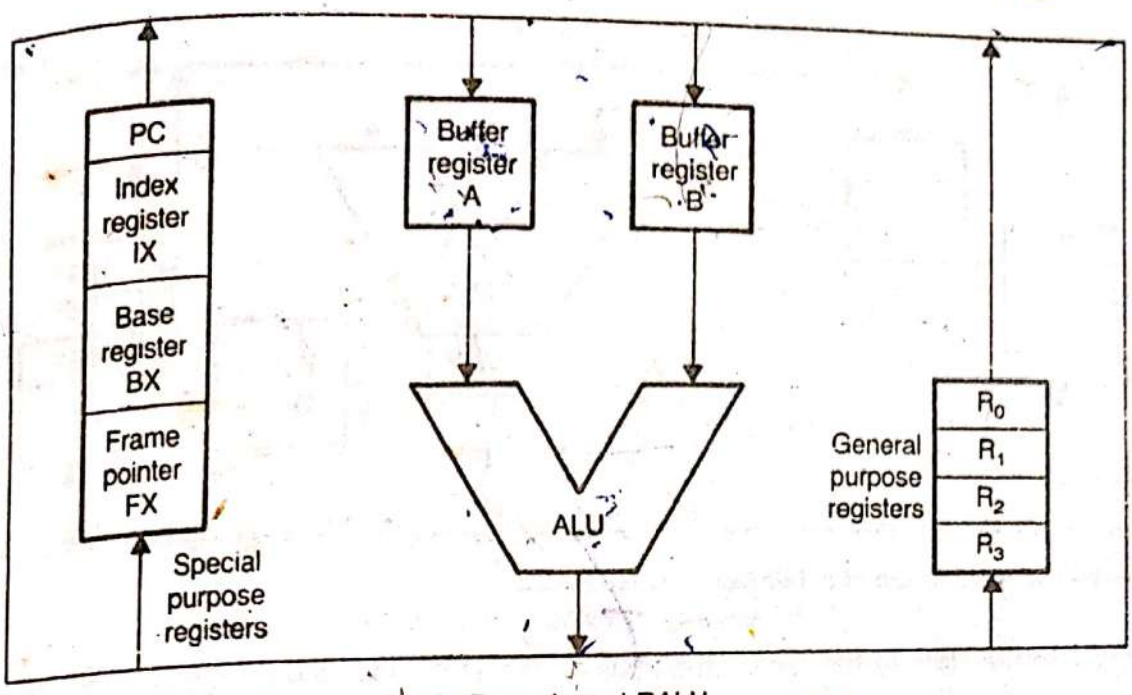



**Figure 4.9** Organization of a Single Bus-oriented RALU

- The sum produced by the ALU is loaded into register $R_2$ when the third clock pulse arrives.

Single-bus organization reduces the speed of the addition operation when the operands are already in the CPU registers. If the operands are in memory, two clock cycles are required to retrieve the required operands. Therefore, this organization affects the speed of execution of a typical two-operand memory-reference instruction.

To carry out a binary operation (such as addition), the control logic must follow a three-step sequence. Each step represents a control state. Therefore, a single-bus architecture increases the number of states in the control logic. More hardware may be required to design the control unit. Since all data transfers take place through the same bus, one at a time, the design effort required to build the control logic is greatly reduced.

A single-bus organization's speed of operation is not as high as that of a two-bus organization. A typical two-bus scheme is shown in Figure 4.10.

All general-purpose registers are connected to both buses (bus A and bus B) to form a two-bus organization. The two operands required by the ALU are, therefore, routed in one clock cycle. Instruction execution is faster, since the ALU does not wait for the second operand, as in the case with a single-bus organization. The information flowing on a bus may be from a general-purpose register or a special-purpose register. In this arrangement, the special-purpose registers are often divided into two groups. Each group is connected to one of the buses. The data from two special-purpose registers of the same group cannot be transferred to the ALU at the same time.

In Figure 4.10, the contents of the PC are always transferred to the right input of the ALU, since it is connected to bus A. Similarly, the contents of the special-purpose register MBR are always transferred to the left input of the ALU, since it is connected to bus B. Whenever there is a need to process simultaneously the contents of two

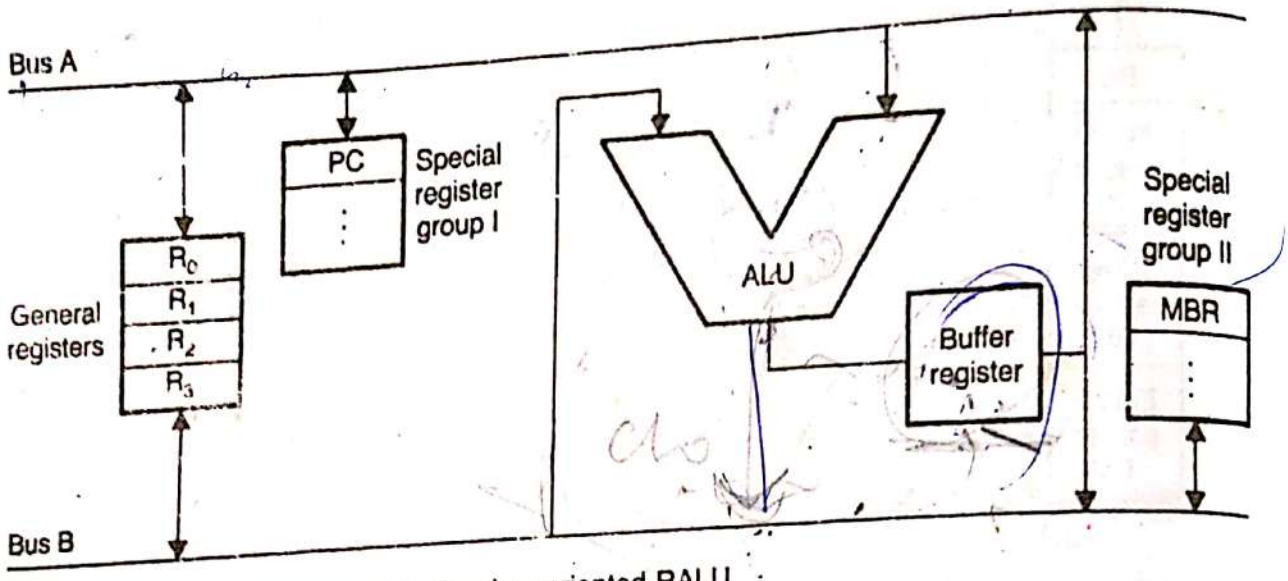



**Figure 4.10** The Architecture of a Two-bus-oriented RALU

special-purpose registers of the same group, the contents of one of the registers must be transferred to one of the general-purpose registers prior to processing. The initialization step requires the execution of an additional instruction. The output of the ALU may be routed to either a special-purpose or a general-purpose register. Since the ALU does not have any input buffer registers, both buses carry operands and are busy during binary operations. Therefore, the output produced by the ALU will first be routed to the output register.

Transfer of the required operands and loading of the ALU output buffer register takes place in one cycle. The result held in the ALU output buffer register is then routed to the required destination in the second clock cycle. The contents of the ALU output buffer register can be gated to either bus A or bus B.

Besides the data paths shown in Figure 4.10, there may be additional dedicated data paths betweens special-purpose registers. For example, a dedicated data path is often provided between the MAR and PC because the register transfer operation

$$MAR \leftarrow PC$$

appears at the beginning of each instruction cycle.

The performance of a two-bus organization can be further improved by adding a third bus, bus C, at the output of the ALU. The resulting structure is known as a *three-bus organization* and is shown in Figure 4.11.

From this figure notice that a three-bus structure is reduced to the two-bus structure if bus C is replaced with the ALU output buffer register. The addition of bus C allows the system to perform all ALU operations, such as $R_2 \leftarrow R_0 + R_1$, in one cycle, since three separate data paths or buses are provided with the system. These data paths include routing for the two operands and the result. A finite delay will occur due to the transferring of operands to the ALU and the time taken by the ALU for producing the result. This delay must be taken care of when transferring the result to the third bus. The addition of the third bus will increase the system cost. The design of the control logic will be complicated because of the critical time delays involved.

Another important concept closely related to the control unit design is the generation of timing signals. One task of a control unit is properly to sequence a set of

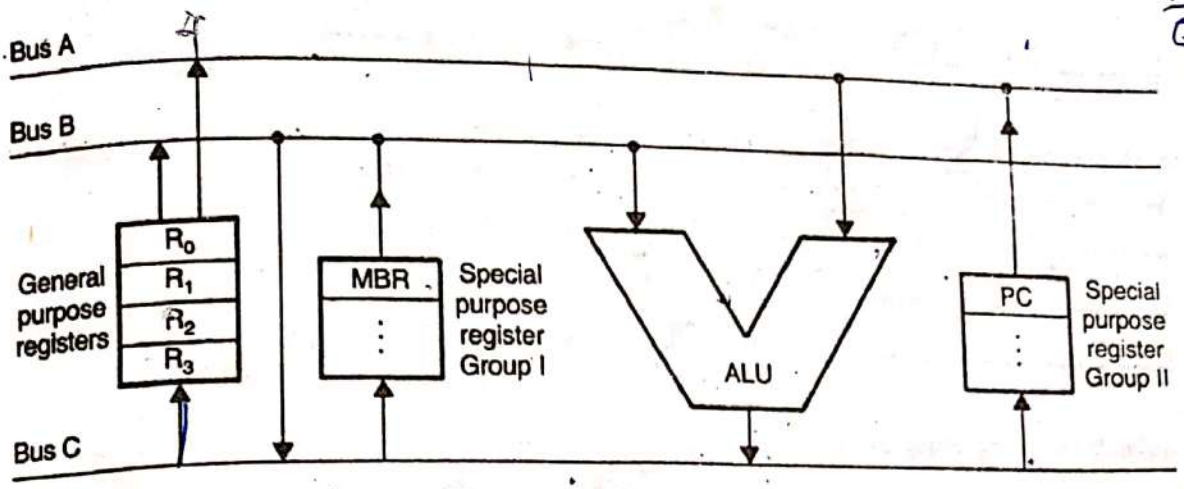



**Figure 4.11** The Organization of a Three-bus-Oriented RALU

operations. Normally, a sequence of $N$ consecutive operations will occur in response to $N$ consecutive clock pulses. To carry out an operation, $P_i$ at the $i$th clock pulse, a control unit must count the clock pulses and produce a timing signal $T_i$. $T_i$ will assume a value of 1 during the duration of the $i$th clock pulse. For example, the input clock pulse and the four timing signals $T_0$, $T_1$, $T_2$, and $T_3$ are shown in Figure 4.12.

In Figure 4.12, the timing signal $T_i$ marks when the $i$th clock pulse has occurred and stays high until the next pulse. The frequency of the signal $T_i$ is one-fourth the input clock pulse, since one clock period of $T_i$ accommodates four clock periods of the input pulse.

An easy way to generate this type of timing signals is by designing a ring counter, as shown in Figure 4.13.

This system will sequence the following manner:

| ***PRESENT STATE*** | ***NEXT STATE*** |
|---|---|
| $A\,B\,C\,D$ | $A^+B^+C^+D^+$ |
| 1 0 0 0 | 0 1 0 0 |
| 0 1 0 0 | 0 0 1 0 |
| 0 0 1 0 | 0 0 0 1 |
| 0 0 0 1 | 1 0 0 0 |

This circuit is also known as a *circular shift register*, since the least-significant bit shifted is not lost. The Boolean equations for each timing variable are derived by inspection as follows:

$$T_0 = A; \quad T_1 = B; \quad T_2 = C; \quad T_3 = D;$$

The chief advantages of this circuit are design simplicity and the ability to generate timing signals without a decoder. Nevertheless, $N$ flip-flops are required to generate $N$ timing signals. This approach is not economically feasible for large values of $N$.

To generate timing signals economically, a new approach is used. A modulo-$2^N$ counter is first designed using $N$ flip-flops. The $N$ outputs from this counter are then

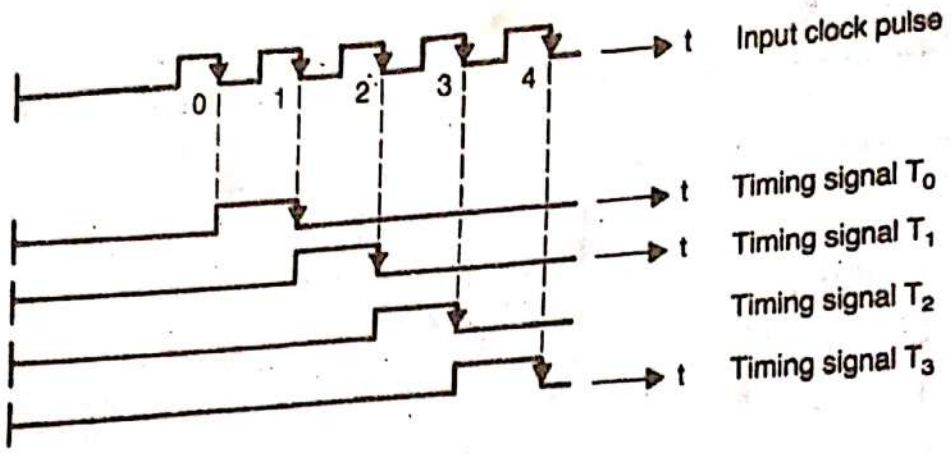



**Figure 4.12** Timing Signals

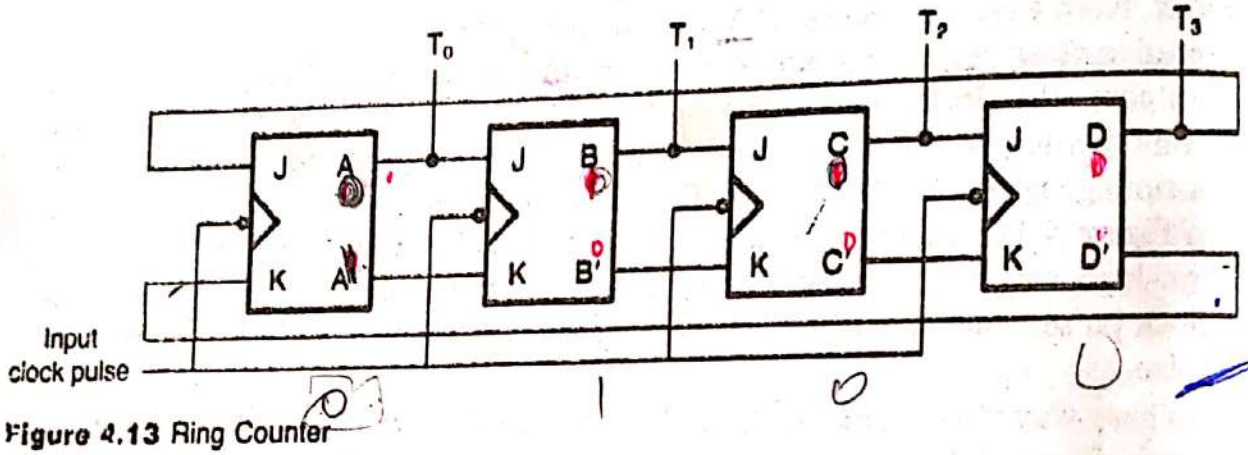



**Figure 4.13** Ring Counter

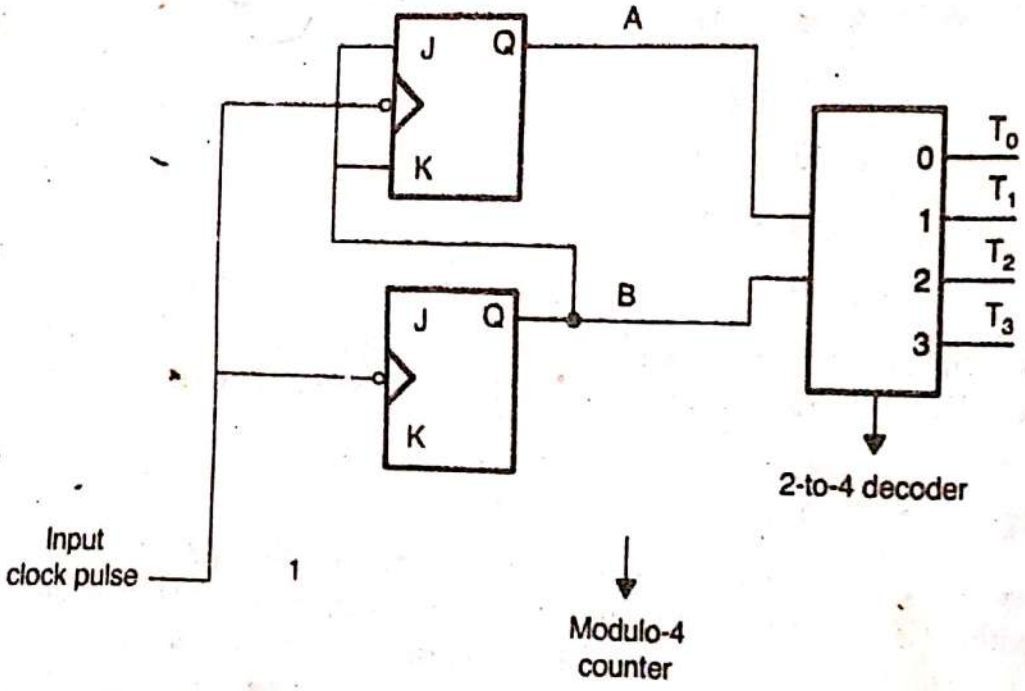



**Figure 4.14** Modulo-4 Counter with a Decoder

given to a $N$-to-$2^N$ decoder as input to generate $2^N$ timing signals. The circuit depicted in Figure 4.14 shows how to generate four timing signals using a modulo-4 counter and a 2-to-4 decoder:

In the preceding circuit, the Boolean equation for each timing signal can be derived as:

$$T_0 = A'B'$$
$$T_1 = A'B$$
$$T_2 = AB'$$
$$T_3 = AB$$

These equations show that four 2-input AND gates are needed to derive the timing signals (assuming single-level decoding). The main advantage of this approach is that $2^n$ timing signals using only $n$ flip-flops are generated. In this method, though, $2^n$ ($n$-input) AND gates are required to decode the $n$-bit output from the flip-flops into $2^n$ different timing signals. Yet the ring counter approach requires $2^n$ flip-flops to accomplish the same task.

## 4.3 DESIGN METHODS

Control units are designed in two different ways:

- Hardwired approach
- Microprogramming

The first approach exists because control logic is a clocked sequential circuit and, therefore, conventional sequential circuit design procedures can be applied to build a control unit. This technique earns the name *hardwired approach*, because the final circuit is obtained by physically connecting typical components such as gates and flip-flops. In the microprogrammed approach, all control functions that can be simultaneously activated are grouped to form control words stored in a separate memory called the control memory. The control words are fetched from the control memory, and the individual control fields are routed to various functional units to enable appropriate gates. When these gates are activated sequentially, the desired task is performed.

The design of a control unit with a control memory is more expensive than hardwired. This is typical when the control system to be designed does not require many microoperations or many control decisions. The inclusion of the control ROM carries overhead such as hardware production cost for masking the bits in the control ROM. A control memory may reduce the overall speed of the system, since the microinstruction retrieval process takes a significant amount of time.

The most important advantage of microprogramming is it provides a well-structured control organization. Control functions are systematically transformed into programming discipline, and a regular memory replaces most of the random combinational circuit elements.

During the design of a system, improvements are thought of and accidental deletions discovered. With microprogramming, many additions and changes are made by simply changing the microprogram in the control memory. A small change in the hardwired approach may lead to redesigning the entire system.

Although hardwired logic is an economical way of obtaining a simple control algorithm, the cost of the control logic increases with system complexity. In microprogrammed implementations, the cost of the simplest system is higher, though adding new features only requires additional control memory. Because of advances in IC technology, replacement of a random logic circuit with a ROM offers substantial hardware savings. CAD methods can be used to reduce the design cost of the system. Microprogramming also simplifies documentation and service training of the system, resulting in reduced total system cost.

Every computer system requires a package of diagnostic routines for checking, locating, and isolating hardware malfunctions. With microprogramming, these routines can be made available in the control memory. An EAROM (electrically alterable ROM) may load these microdiagnostic routines in small segments, with each checking a different portion of hardware.

By changing the control program, it is possible to interpret a new instruction set. The simulation of a processor's instruction set in another processor is called emulation. Emulation provides compatibility of instruction sets between smaller and larger machines of a series.

Implementation of commonly used routines, such as matrix inversion and table look-up procedures in microcode, can result in significant performance improvement. Savings in speed are achieved if a special microprogram is written to carry out repetitive system tasks such as program interpretation and job-queue manipulation (operating system routine).

In summary, microprogramming is accepted as a standard tool for designing the control unit of a computer. For example, processors such as the IBM 370, VAX-11, POP-11, MC68000, and Intel 8086 have a microprogrammed control unit. Nevertheless, Zilog's 16-bit microprocessor Z8000 still employs a hardwired control unit.

### 4.3.1 Hardwired Control Design

In this section, the hardwired approach to control logic design is discussed. The steps involved in this method are summarized as follows:

1. Define the task to be performed.
2. Propose a trial processing section.
3. Provide a register-transfer description of the algorithm based on the processing section outlined in the previous step.
4. Validate the algorithm by using trial data.
5. Describe the basic characteristics of the hardware elements to be used in the processing section.

6. Complete the design of the processing section by establishing necessary control points.
7. Propose a block diagram of the controller.
8. Specify the state diagram of the controller.
9. Specify the characteristics of the hardware elements to be used in the controller.
10. Complete the controller design and draw a logic diagram of the final circuit.

The following discussion explains this procedure by using an example. For the first step, the control task is described as:

***Task definition:*** Design a Booth's multiplier to multiply two 4-bit 2's complement numbers.

In previous chapter, it was shown that Booth's procedure inspects a pair of multiplier bits and initiates one of the following actions:

| ***MULTIPLIER BITS EXAMINED*** | ***ACTION*** |
|---|---|
| $q_j\ q_{j-1}$ | (In position $j$) |
| 0 0 | None |
| 0 1 | Add M |
| 1 0 | Sub M |
| 1 1 | None |

To implement Booth's procedure, the processing section (Step 2) shown in Figure 4.15 is proposed. This setup is the same as the one explained in the previous chapter.

As mentioned earlier, the 4-bit register M will hold the multiplicand. The Q register is 5 bits wide. Initially, the high-order 4-bits of this register will hold the 4-bit multiplier. The least-significant bit of this register is initialized with the fictitious zero discussed in the previous chapter. The 4-bit adder/subtracter is used to perform the operations A + M or A − M. The result produced by this hardware is always routed to the 4-bit accumulator register A. In this implementation the multiplicand is not shifted to the left; rather the accumulated partial product is shifted right. When the computation terminates, the high- and low-order 4 bits of the final product are found in registers A and Q, respectively.

The L register is used to keep track of the iteration count. In this particular case, this register is initialized with $4_{10}$ $(100_2)$ and is decremented by 1 at the completion of each iteration. Therefore, the algorithm terminates when 000 is in the L register. The 4-bit data buses, inbus and outbus, are used to transfer data into and out of the processing section, respectively.

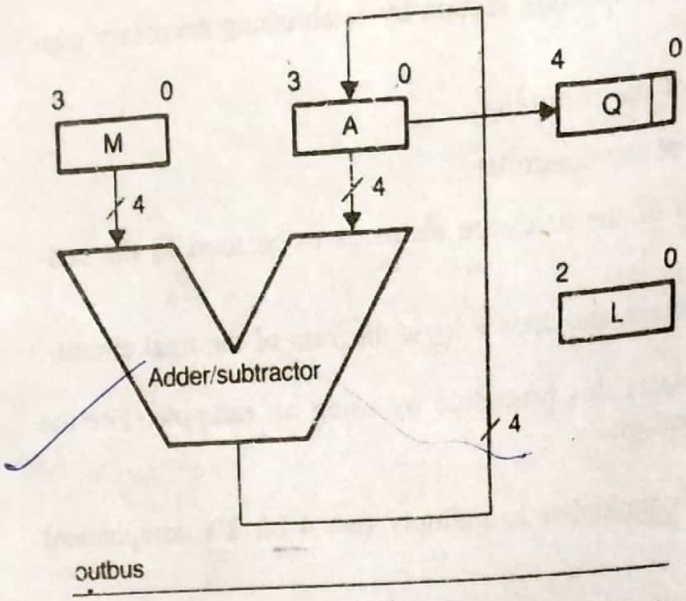



**Figure 4.15** Organization of the Processing Section for Performing 4 × 4 Multiplication Using Booth's Algorithm

For Step 3, a register transfer description of Booth's multiplication algorithm is provided in Figure 4.16.

In Figure 4.16, Q [1:0] is used to indicate the low-order 2 bits of the Q register (Q [1] and Q [0]). Similarly, Q [4:1] indicates the high-order 4 bits of the Q register. The last step, go to HALT, introduces an infinite loop after the computation is completed.

The correctness of this procedure is validated by a complete trace (Step 4) as shown next:

$M = 1100_2 = -4_{10}$ and $Q = 0111_2 = 7_{10}$

| | M | A | MULTIPLIER BITS INSPECTED Q | L |
|---|---|---|---|---|
| Initialization | 1100 | 0000 | 0111[10] | 100 |
| Iteration 1 | | | | |
| $A \leftarrow A - M$ | 1100 | 0100 | 01110 | 100 |
| ASR (A$Q), $L \leftarrow L-1$ | 1100 | 0010 | 00[11]1 | 011 |
| Iteration 2 | | | | |
| ASR (A$Q), $L \leftarrow L-1$ | 1100 | 0001 | 000[11] | 010 |
| Iteration 3 | | | | |
| ASR (A$Q), $L \leftarrow L-1$ | 1100 | 0000 | 100[01] | 001 |
| Iteration 4 | | | | |
| $A \leftarrow A + M$ | 1100 | 1100 | 10001 | 001 |
| ASR (A$Q), $L \leftarrow L-1$ | 1100 | 1110 | 01000 | 000 |

Product = $-28$

```
Declare registers A [4], M [4], Q [5], L [3];
Declare buses   Inbus [4], Outbus [4];
Start:  A ←— 0, M ←— Inbus, L ←— 4; clear A and transfer M

        Q [4:1] ←— Inbus, Q [0] ←— 0; Transfer Q
Loop:   If Q [1:0] = 01 then go to Add;
        If Q [1:0] = 10 then go to Sub;
        Go to RShift;
Add:    A ←— A + M;
        Go to RShift
Sub:    A ←— A – M;
RShift: ASR (AQ), L ←— L – 1;
        If L ≠ 0 then go to Loop
        Outbus = A;
        Outbus = Q [4:1];
Halt:   Go to Halt
```

**Figure 4.16** Register Transfer Description of Booth's Multiplication Procedure

The processing section includes three main components:

- General-purpose storage registers
- 4-bit adder/subtracter
- Tristate buffers

The functional characteristics of these parts are summarized in Figure 4.17 (Step 5).

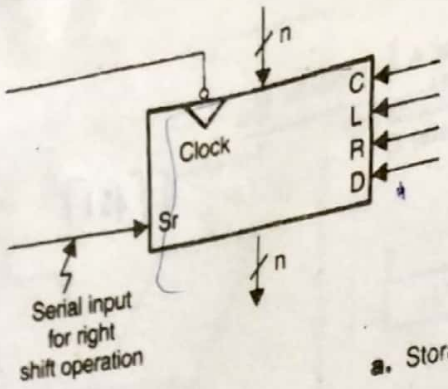
Notice that the general register is basically a trailing-edge-triggered device. Typically, four operations (clear, parallel load, right shift, and decrement) can be performed by introducing the proper values to control inputs C, L, R, and D. All these operations are synchronized with the trailing edge (high to low) of the clock pulse.

Such a general-purpose register can be built using standard MSI parts and gates. The 4-bit adder/subtracter is implemented by using a 4-bit parallel adder chip (7483) and four 2-input exclusive-OR gates. The tristate buffers are used to control the data transfer to the outbus.

A detailed logic diagram of the processing section, along with interpretation of various control points, is shown in Figure 4.18 (Step 6).

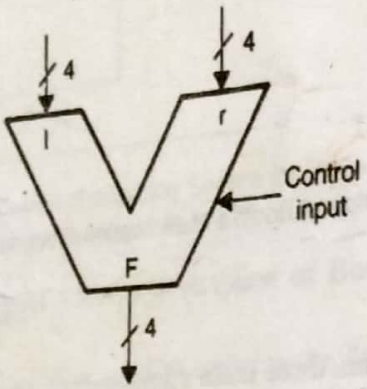
From this figure, we observe that there are 10 control points, $C_0, C_1, C_2, \ldots, C_9$.

If $C_0$ is held high, the A register will be cleared with the trailing edge of the next clock. The other control points are interpreted similarly. One control point is introduced for each microoperation specified in the register-transfer description (see Figure 4.16). The processing section extends three outputs Q [1], Q [0], and Z; Z = 1 only when the contents of the L register become zero. These outputs are the status outputs and are used as inputs to the controller to allow the controller to decide the future course of action.



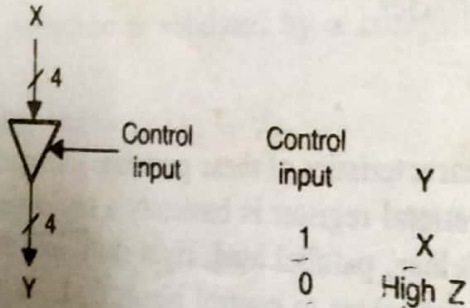

| C | L | R | D | Clock | Action |
|---|---|---|---|---|---|
| ↓ | | | | ↓ | Clear |
| 1 | 0 | 0 | 0 | ↓ | Load external data |
| 0 | 1 | 0 | 0 | ↓ | Right shift |
| 0 | 0 | 1 | 0 | ↓ | Decrement by one |
| 0 | 0 | 0 | 1 | ↓ | No change |
| 0 | 0 | 0 | 0 | | |

a. Storage Register




| Control input | F |
|---|---|
| 1 | l + r |
| 0 | l − r |

b. Adder subtractor




| Control input | Y |
|---|---|
| 1 | X |
| 0 | High Z |

c. Tri-state Buffer

**Figure 4.17** Characteristics of the Component Parts Used in the Processing Section of the Booth's Multiplier

With this information a block-diagram representation for the controller can be formed, as shown in Figure 4.19 (Step 7).

From the block diagram, it can be seen that the controller has 5 inputs and 10 outputs. The RESET input is an asynchronous input used to reset the controller so a new computation can begin. The clock input is used to synchronize the controller's action. All activities are assumed to be synchronized with the trailing edge of the clock pulse.

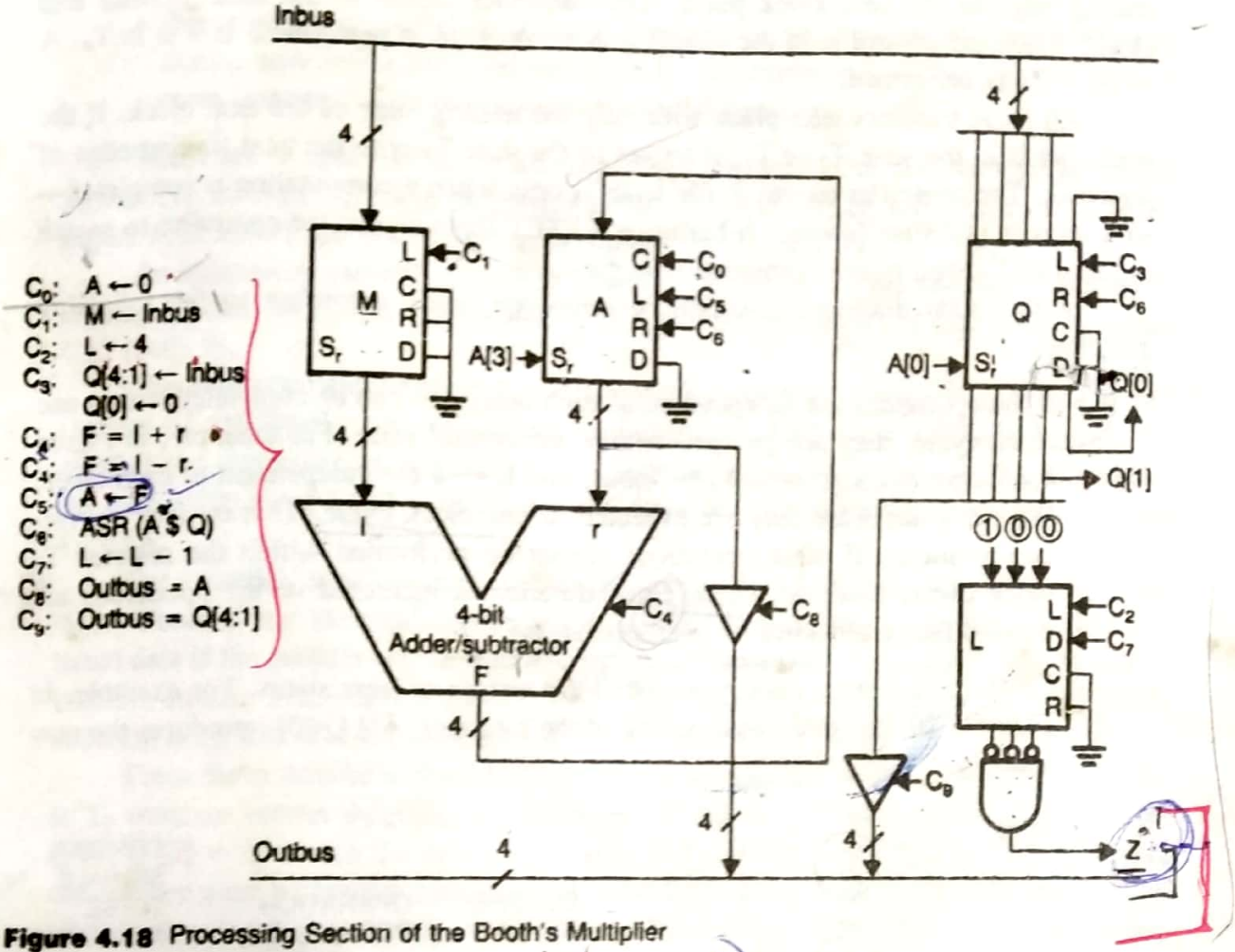



**Figure 4.18** Processing Section of the Booth's Multiplier

We mentioned earlier a controller must initiate a set of operations in a specified sequence. Therefore, it is modeled as a sequential circuit. The state diagram for the Booth's Multiplier Controller is shown in Figure 4.20 (Step 8).

Initially, the controller is in the state $T_0$. At this point the control signals $C_0$, $C_1$, and $C_2$ are high. Operations $A \leftarrow 0$, $L \leftarrow 4$, and $M \leftarrow$ Inbus are carried out with the

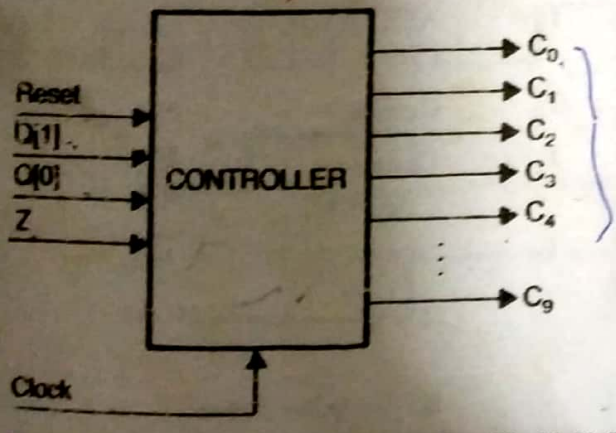


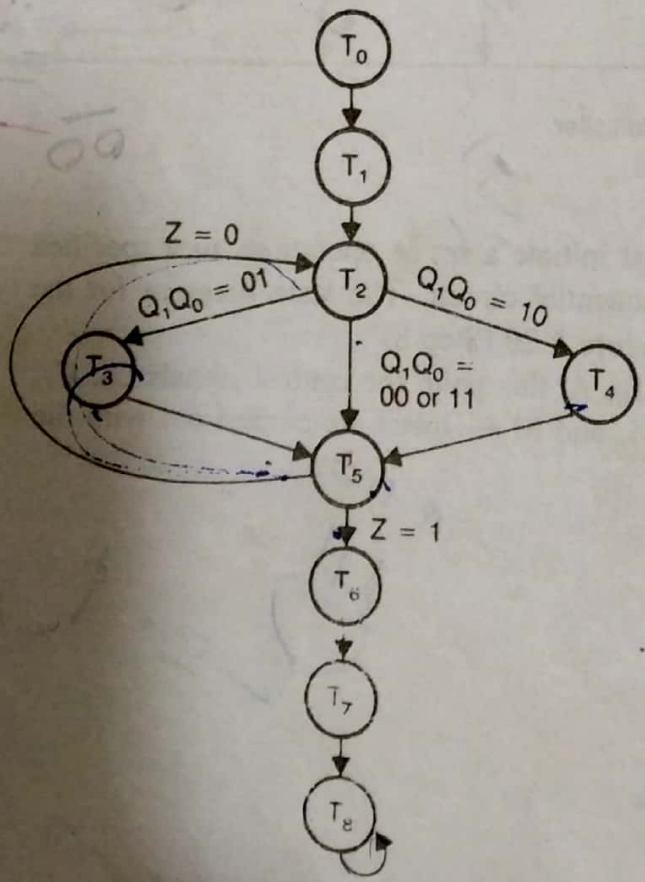

**Figure 4.19** Block-diagram of the Booth's Multiplier Controller

trailing edge of the next clock pulse. The controller moves to the state $T_1$ with this clock. When the control is in the state $T_3$, $A \leftarrow A + M$ is performed. If it is in $T_4$, $A \leftarrow A - M$ is performed.

All these transfers take place with only the trailing edge of the next clock. If the controller is in the state $T_3$ or $T_4$, it moves to the state $T_5$ with the next trailing edge of the clock. The controller moves to the state $T_8$ only when a computation is completed—so it stays in that state forever. A hardware RESET input causes the controller to switch to the state $T_0$, and a new computation begins.

In this state diagram, selection of states are made according to the following guidelines:

- If the operations are independent of each other and can be completed within one clock cycle, they are grouped within one control state. For example, in Figure 4.19, operations $A \leftarrow 0$, $M \leftarrow$ Inbus, and $L \leftarrow 4$ are independent of each other. With this hardware they are executed in one clock cycle. That is, they are microoperations. If these operations cannot be performed within the selected $T_0$ clock cycle, however, either clock duration is increased or the operations are divided into a sequence of microoperations.
- Conditional testing usually implies introduction of new states. For example, in Figure 4.20 the conditional testing of the bit pair Q [1] Q [0] introduces the new state $T_2$.




a. State Diagram

| CONTROL STATE | OPERATION PERFORMED | CONTROL SIGNALS TO BE ACTIVATED |
|---|---|---|
| $T_0$ | $A \leftarrow 0$, $L \leftarrow 4$, $M \leftarrow$ Inbus | $C_0$, $C_1$, $C_2$ |
| $T_1$ | $Q[4:1] \leftarrow$ Inbus, $Q[0] \leftarrow 0$ | $C_3$ |
| $T_2$ | None | None |
| $T_3$ | $A \leftarrow A + M$ | $C_4$, $C_5$ |
| $T_4$ | $A \leftarrow A - M$ | $C_5$ ($C_4 = 0$) |
| $T_5$ | ASR (A$Q), $L \leftarrow L - 1$ | $C_6$, $C_7$ |
| $T_6$ | Outbus = A | $C_8$ |
| $T_7$ | Outbus = Q [4:1] | $C_9$ |
| $T_8$ | None | None |

b. Controller Action

**Figure 4.20** Controller Description

- One should not make an attempt to minimize the number of states. When in doubt, new states must be introduced. The correctnes of the control logic is more important than the economics of the circuit.

There are 9 states in the controller state diagram. Nine nonoverlapping timing signals ($T_0$ through $T_8$) must be generated so only one will be high for a clock pulse. Figure 4.21 shows the first three timing signals $T_0$, $T_1$, and $T_2$.

As mentioned earlier, a mod-16 counter and a 4-to-16 decoder can be used to accomplish this task. The characteristics of the mod-16 counter is discussed in Figure 4.22 (Step 9).

The controller and its logic diagram are shown in Figure 4.23 (Step 10). The key element of this implementation is the sequence controller (SC) hardware, which sequences the controller as indicated in the state diagram (see Figure 4.20). Therefore, the truth table for the SC must be derived from the controller's state diagram (Figure 4.24(a)).

For example, consider the logic involved in deriving the first entry of the SC truth table. Observe that the mod-16 counter is loaded (or initialized) with the specified external data if the counter control inputs C and L are 0 and 1, respectively. In this counter module, counter load control, input L, overrides the count enable control, input E. This situation is typical of 4-bit MSI counter chips, such as the 74161.

From the controller's state diagram, it can be seen that if the present control state is $T_2$ (counter output $0_3 0_2 0_1 0_0 = 0010$) and if the bit pair inspected is 00, (that is, Q [1] Q [0] = 00), then the next control state is $T_5$. When these input conditions occur, the counter must be loaded with external value 0101 along with the trailing edge of the next clock pulse ($T_5 = 1$ only when $0_3,0_2,0_1,0_0 = 0101$). Therefore, the SC generates $L = 1$ and $d_3,d_2,d_1,d_0 = 0101$.

Using the same reasoning, the fifth entry of the SC truth table is obtained as follows. From the controller's state diagram, it can be seen that if the present control state is $T_5$ and if $Z = 0$, the next control state is $T_2$. The SC must generate the outputs $L = 1$ and $d_3d_2d_1d_0 = 0010$ to achieve the desired state sequence. Other entries of the SC truth table are derived in the same manner.

When the counter load control input $L = 0$, the counter will automatically count

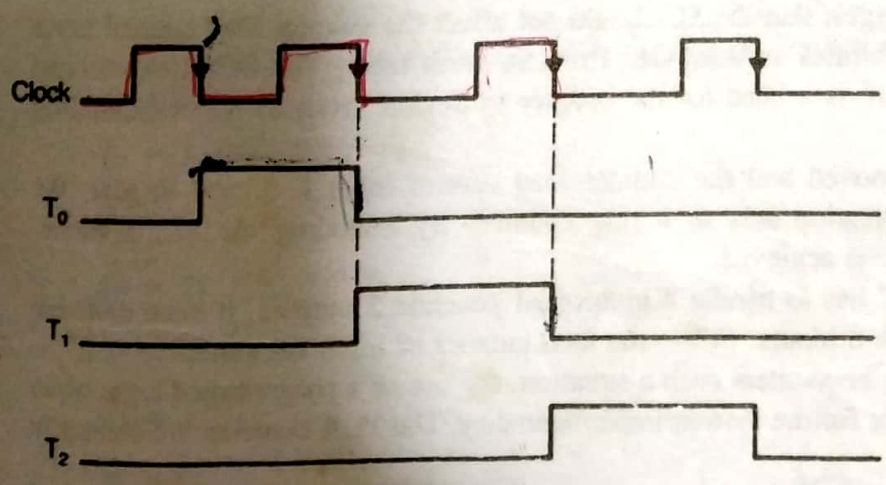
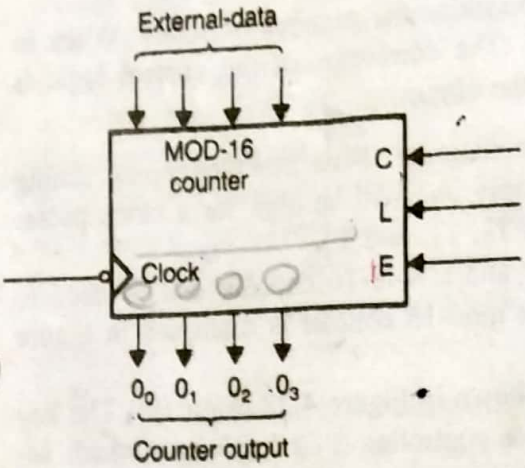



**Figure 4.21** Timing Signals Generated by the Controller




**a.** Block Diagram

| C | L | E | Clock | Action |
|---|---|---|---|---|
| 1 | X | X | X | Clear |
| 0 | 1 | X | ↓ | Load external data |
| 0 | 0 | 1 | ↓ | Count up |
| 0 | 0 | 0 | ↓ | No operation |

**b.** Function Table

**Figure 4.22** Characteristics of the Counter Used in the Controller Design

up in response to the next clock pulse (because the enable input E is tied to high). Such normal sequencing activity is desirable in the following situations:

- Present control state is $T_0$, $T_1$, $T_4$, $T_6$, or $T_7$.
- Present control state is $T_2$ and Q [1] Q [0] = 01.
- Present control state is $T_5$ and Z = 1.

These results suggest that the SC should not affect the counter load control input L. Hence, these possibilities are excluded from SC truth table. The SC must exercise control only when there is a need for the counter to deviate from its normal counting sequence.

If the SC is removed and the counter load control input L is tied to low, the counter-decoder combination acts as a ring counter. By including the SC, a status-dependent ring counter is achieved.

Although the SC has to handle 8 inputs and generate 5 outputs, it must examine a few possibilities with 8 inputs. (*Note:* the total number of input combinations is $2^8$ = 256). When a designer encounters such a situation, the use of a programmed logic array (PLA) is a good choice for the system implementation. The PLA contents are shown in Figure 4.24(b).

This implementation follows the SC truth table. For each row of the SC truth

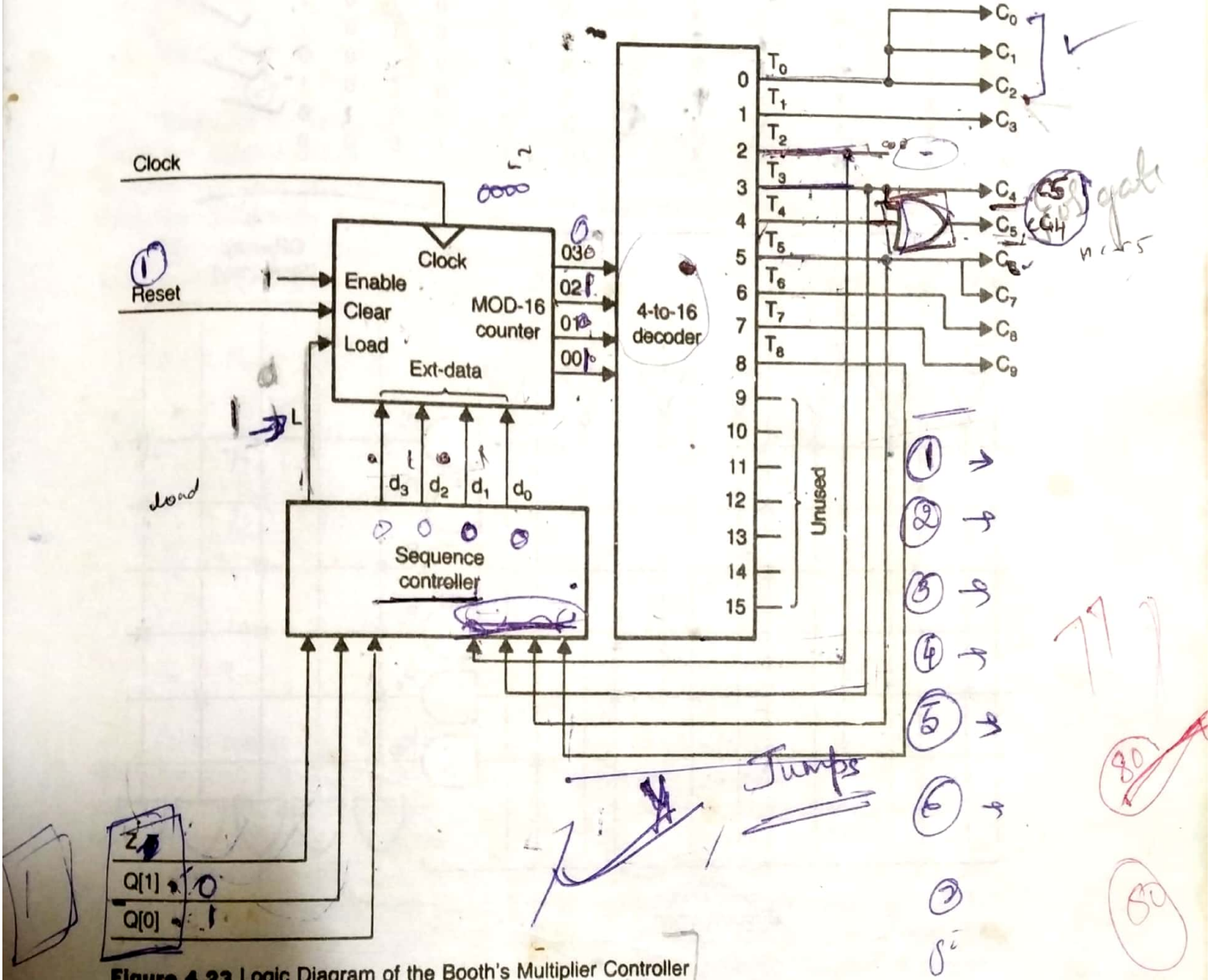



**Figure 4.23** Logic Diagram of the Booth's Multiplier Controller

table, a product term is generated in the PLA. The product terms generated are summarized as follows:

$$P_0 = Q[1]' Q[0]' T_2$$

$$P_2 = Q[1] Q[0] T_2$$

$$P_2 = Q[1] Q[0]' T_2$$

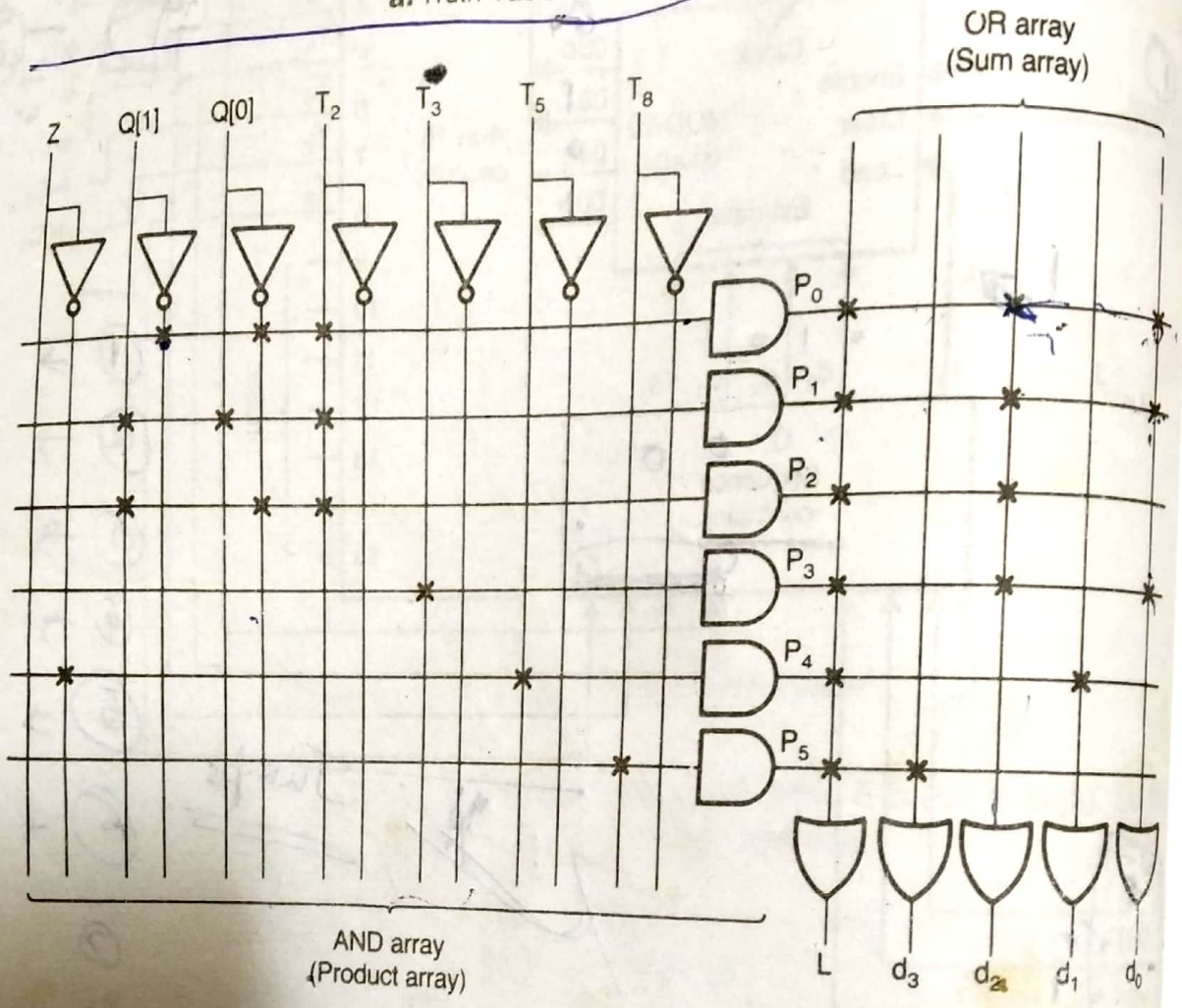
$$P_3 = T_3$$

$$P_4 = Z'T_5$$

$$P_5 = T_8$$

| Z | Q[1] | Q[0] | $T_2$ | $T_3$ | $T_5$ | $T_8$ | L | External-data d3 | d2 | d1 | d0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| X | 0 | 0 | 1 | X | X | X | 1 | 0 | 1 | 0 | 1 |
| X | 1 | 0 | 1 | X | X | X | 1 | 0 | 1 | 0 | 1 |
| X | 1 | 0 | 1 | X | X | X | 1 | 0 | 1 | 0 | 0 |
| X | 1 | 0 | 1 | X | X | X | 1 | 0 | 1 | 0 | 1 |
| X | 1 | 0 | 1 | X | X | X | 1 | 0 | 0 | 1 | 0 |
| X | 1 | 0 | 1 | X | X | X | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | X | X | X | 1 | 1 | 0 | 0 | 0 |
| X | 1 | 0 | 1 | X | X | X | 1 | | | | |

a. Truth Table




b. PLA Implementation

**Figure 4.24** Sequence Controller Design

The PLA generates five outputs: L, $d_3$, $d_2$, $d_1$, and $d_0$. Each output is directly generated by using the SC truth table and the product terms. The Boolean equations that govern the relationship between the product terms and the PLA outputs are summarized below

$$L = P_0 + P_1 + P_2 + P_3 + P_4 + P_5$$

$$d_3 = P_5$$

$$d_2 = P_0 + P_1 + P_2 + P_3$$
$$d_1 = P_4$$
$$d_0 = P_0 + P_1 + P_3$$

From the above equations, notice that to generate the output L, all the product terms are selected and summed using the PLA-OR gate array. The output $d_0$ is produced by selecting three product terms—$P_0$, $P_1$ and $P_3$—and summing them. The use of a PLA has significantly minimized the design effort.

The controller design is completed by relating the control states ($T_0$ through $T_8$) with the control signals ($C_0$ through $C_9$) as follows:

$$C_0 = C_1 = C_2 = T_0$$
$$C_3 = T_1$$
$$C_4 = T_3$$
$$C_5 = T_3 + T_4$$
$$C_6 = C_7 = T_5$$
$$C_8 = T_6$$
$$C_9 = T_7$$

These results directly follow the controller action specified in Figure 4.20(b). When the control is in the state $T_0$, three microoperations are needed: A ← 0, L ← 4, and M ← Inbus, implying that $C_0 = C_1 = C_2 = T_0$. A ← F is performed when the control state is $T_3$ or $T_4$. Therefore, $C_5 = T_3 + T_4$.

A control unit can be designed by using a single PLA and D flip-flops. The organization of a PLA control unit for the Booth's multiplier unit is shown in Figure 4.25.

In Figure 4.25, four D flip-flops and a PLA are used. The D flip-flops hold the present control state, and the PLA uses this information to generate the required control signals and the next control state. The flip-flops are updated with the next state information when the next clock arrives. The controller's state diagram (see Figure 4.20) contains 9 states. The binary encoding of these states calls for 4 bits and four D flip-flops are required to hold them.

The PLA of Figure 4.25 implements the state table shown in Figure 4.26. This state table is the tabular representation of the controller's state diagram. The table also includes the control functions to be activated at any particular state. This information is obtained from Figure 4.20(b). The PLA contents are shown in Figure 4.27. Verification of the correctness of this figure is left as an exercise.

The PLA, a programmable LSI component, is effective in replacing a random logic circuit that typically contains 6 to 8 MSI chips. Such replacement is desirable for the following reasons:

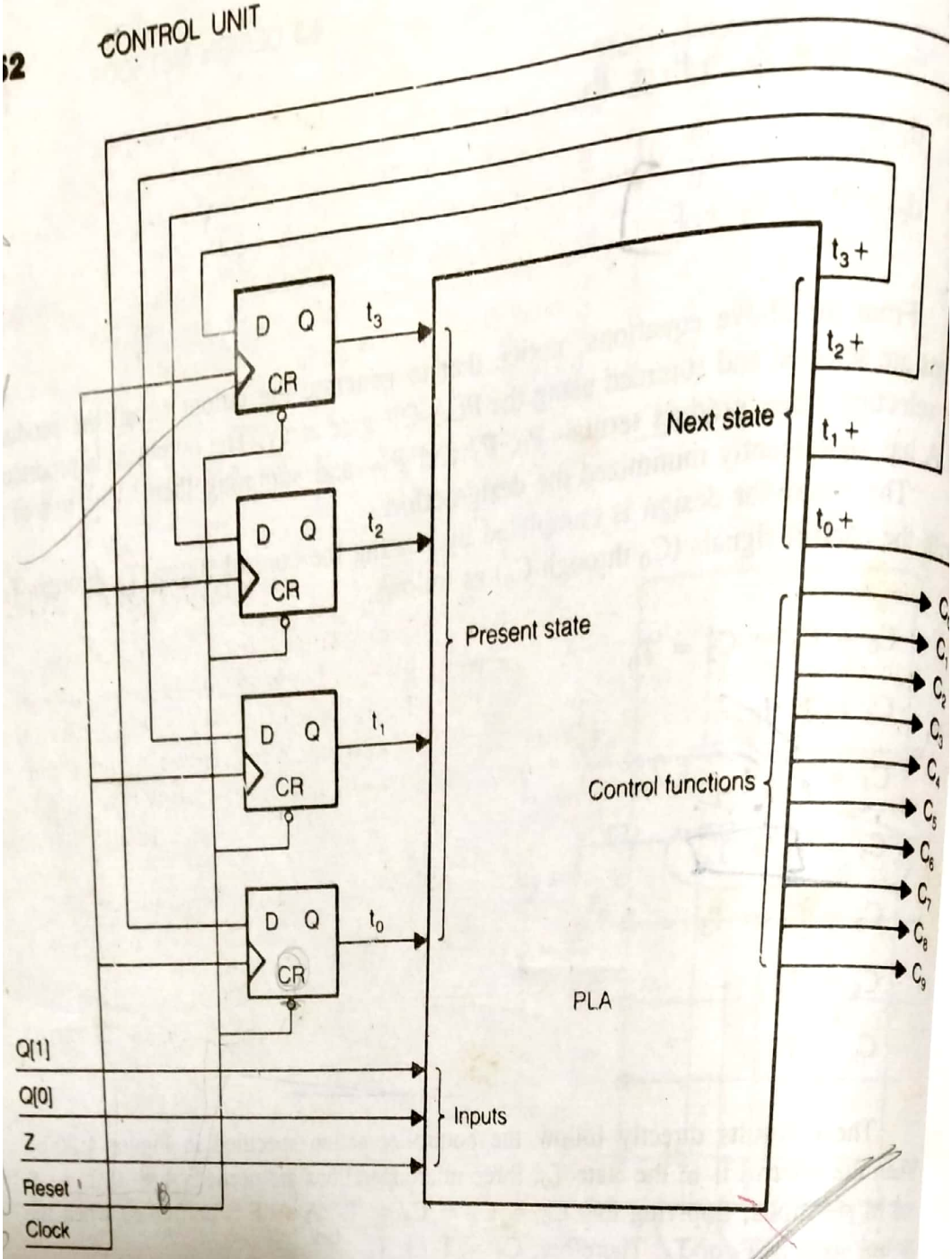



**Figure 4.25** Organization of the PLA Control Unit for the Booth's Multiplier

1. Increased flexibility and reliability
2. Minimum design effort
3. Uniform component structure
4. Space saving

Uniform hardware component structures result in efficient mass production giving lower hardware cost. The size of the control memory (to be explained later) of the Intel 8086 processor is 504 words with 21 bits per word. To save chip area, this component is made by using a PLA, as opposed to a ROM. With the advent of integrated circuit technology, manufacturers are able to produce erasable programmable PLAs.

Such a product is produced by ALTERA Corporation. Such PLAs offer great flexibility since their contents can be altered on line. It is speculated that in the future the cost of the erasable PLA will drop to a level allowing wider usage.

| | PLA Input | | | | | | | PLA Output | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Present state in binary | | | | Inputs | | | Next state (in binary) | | | | Control functions | | | | | | | | | |
| | $t_3$ | $t_2$ | $t_1$ | $t_0$ | Q [1] | Q [0] | Z | $t_3^+$ | $t_2^+$ | $t_1^+$ | $t_0^+$ | $C_0$ | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ | $C_8$ | $C_9$ |
| $T_0$ | 0 | 0 | 0 | 0 | X | X | X | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $T_1$ | 0 | 0 | 0 | 1 | X | X | X | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| $T_2$ | 0 | 0 | 1 | 0 | 0 | 1 | X | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $T_2$ | 0 | 0 | 1 | 0 | 0 | 0 | X | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $T_2$ | 0 | 0 | 1 | 0 | 1 | 1 | X | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $T_2$ | 0 | 0 | 1 | 0 | 1 | 0 | X | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $T_3$ | 0 | 0 | 1 | 1 | X | X | X | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| $T_4$ | 0 | 1 | 0 | 0 | X | X | X | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| $T_5$ | 0 | 1 | 0 | 1 | X | X | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| $T_5$ | 0 | 1 | 0 | 1 | X | X | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| $T_6$ | 0 | 1 | 1 | 0 | X | X | X | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| $T_7$ | 0 | 1 | 1 | 1 | X | X | X | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| $T_8$ | 1 | 0 | 0 | 0 | X | X | X | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Controller's state table

**Figure 4.26** PLA Table

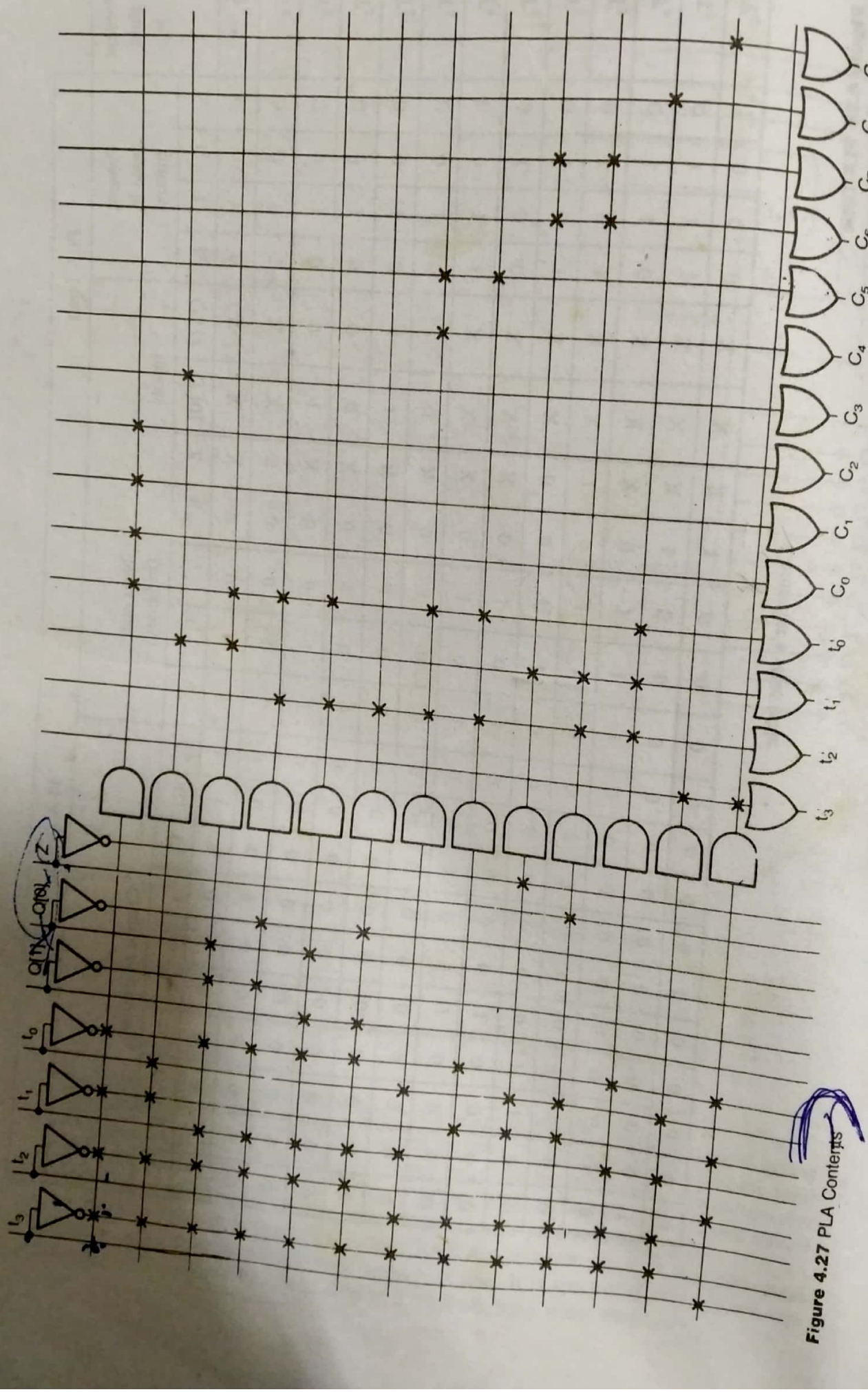



**Figure 4.27** PLA Contents

### 4.3.2 Microprogrammed Control Unit

The design of a microprogrammed unit will be discussed in detail. As seen earlier, a microprogrammed control unit's control words are held in a separate memory called the *control memory* (CM). Each control word contains signals to activate one or more microoperations. When these words are retrieved in a sequence, a set of microoperations are activated that will complete the desired task.

Retrieval and interpretation of the control words are done the same as a conventional program. The instructions of a processor are held in the main store. They are fetched and executed in a sequence. By changing the instructions stored in the main memory, the processor can perform different functions. Similarly, by changing the contents of the CM, the control unit can execute a different control function. Therefore, the microprogrammed approach offers greater flexibility than its hardwired counterpart, since it provides an easy means for altering the contents of the CM.

Generally, all microinstructions have two important fields:

- Control field
- Next-address field

The purpose of the control field is to indicate which control lines are to be activated. The purpose of the next address field is to specify the address of the next microinstruction to be executed. When M. V. Wilkes proposed this idea in 1951, he used the organization close to the one shown in Figure 4.28.

In Figure 4.28 the decoder and diode matrix utilize an 8 x 8 ROM. Each ROM holds an 8-bit control word whose low-order three positions specify the next address, and high-order five positions indicate control signals. The contents of the ROM are retrieved by specifying the address in the control memory address register (CMAR). For example, if $X_2X_1X_0 = 010$, then control lines $C_0$ and $C_3$ are enabled, and the next ROM address, 011, is fed back to the CMAR at the same time.

It is possible to load the CMAR with an externally specified new starting address. Whenever the external load input E is 1, the CMAR is loaded with an externally specified address. It is then possible to keep several microprograms in the control memory and execute any desired microprogram by specifying the starting address as an external address.

Another important feature of this organization is the ability to implement conditional branching. The external condition sets or resets the condition flip-flop. For instance, if the external condition sets the condition flip-flop to 1, control is transferred to ROM address 1 ($X_2X_1X_0 = 001$) after execution of microinstruction at ROM address 5. If the external condition is zero, the condition flip-flop will be reset. Therefore, after execution of the microinstruction at ROM address 5, the control will be transferred to ROM address 6 ($X_2X_1X_0 = 110$).

The microprogramming approach involves the inclusion of a control memory in addition to the conventional main memory. Therefore, a major design effort has its emphasis on minimizing the length of the microinstruction. The length of the microinstruction decides the size of the control store, as well as the cost involved with this approach. The length of a microinstruction is directly related to the following factors:

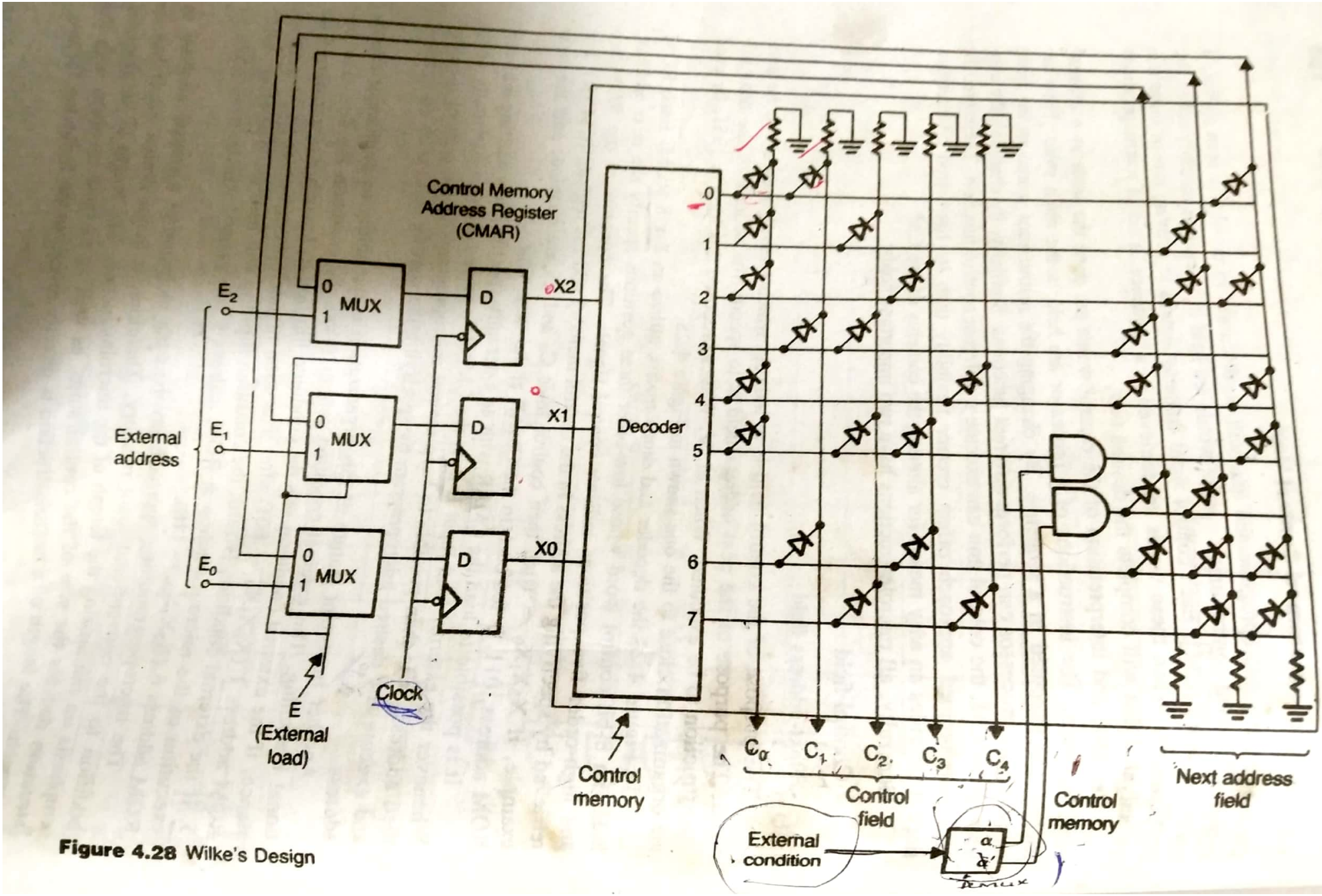



**Figure 4.28** Wilke's Design

- The degree of parallelism, or how many microoperations can be activated simultaneously.
- The control field organization.
- The method by which the address of the next microinstruction is specified.

The structure of a microinstruction is similar to that of the processor instruction discussed in Chapter 2. It is possible to execute several microoperations simultaneously. All microoperations executed in parallel can be specified in a single microinstruction with a common op-code. This allows short microprograms to be written. Nevertheless, whenever there is an overabundance of parallelism, the length of a microinstruction increases. Similarly, short microinstructions have limited capability in expressing parallelism. Since short microinstructions are not able to express a massive parallelism, the overall length of a microprogram written using these instructions will increase.

The control information can be organized in various ways. A trivial way to organize the control field would be to have 1 bit for each control line that controls the data processor, allowing full parallelism, and there is no need for decoding the control field. This method, however, leads to inefficient use of the control memory space when it is impossible to invoke all control operations simultaneously.

Consider the following situation shown in Figure 4.29. Assume there are four registers, A, B, C, and D, and each communicates with the outbus when the appropriate control line is activated:

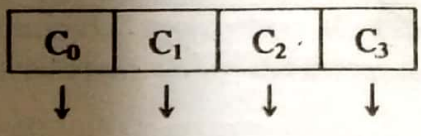
$C_0$: Outbus = A

$C_1$: Outbus = B

$C_2$: Outbus = C

$C_3$: Outbus = D

Since there is only one output bus, it is impossible to allow more than one transfer at any given time. If one bit is allocated for each control in the control field, the result will appear as shown next:




| $C_0$ | $C_1$ | $C_2$ | $C_3$ | |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | Outbus = A |
| 0 | 1 | 0 | 0 | Outbus = B |
| 0 | 0 | 1 | 0 | Outbus = C |
| 0 | 0 | 0 | 1 | Outbus = D |
| 0 | 0 | 0 | 0 | No operation |

This method is also known as *unencoded format*.

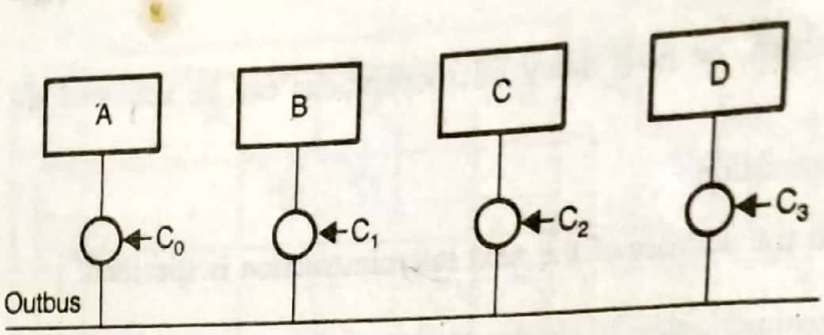



**Figure 4.29** A Simple Register Transfer

There are only five valid binary patterns in the preceding format; but from the basic switching theory, five distinct binary patterns can be represented by using only 3 bits. Such an arrangement is shown in Figure 4.30.

In Figure 4.30, the control information is encoded into a 3-bit field, and a decoder is needed to extract the actual control information. The relationship between the encoded and the actual control information is specified as follows:

| $E_2$ | $E_1$ | $E_0$ | |
|---|---|---|---|
| 0 | 0 | 0 | No operation |
| 0 | 0 | 1 | Outbus = A |
| 0 | 1 | 0 | Outbus = B |
| 0 | 1 | 1 | Outbus = C |
| 1 | 0 | 0 | Outbus = D |

This method is known as *encoded format*, which leads to a short control field and short microinstructions. The price paid for such a reduction is the need to have a decoder. Therefore, a compromise must be sought. Fifteen control lines can be specified in a fully unencoded form as shown next:

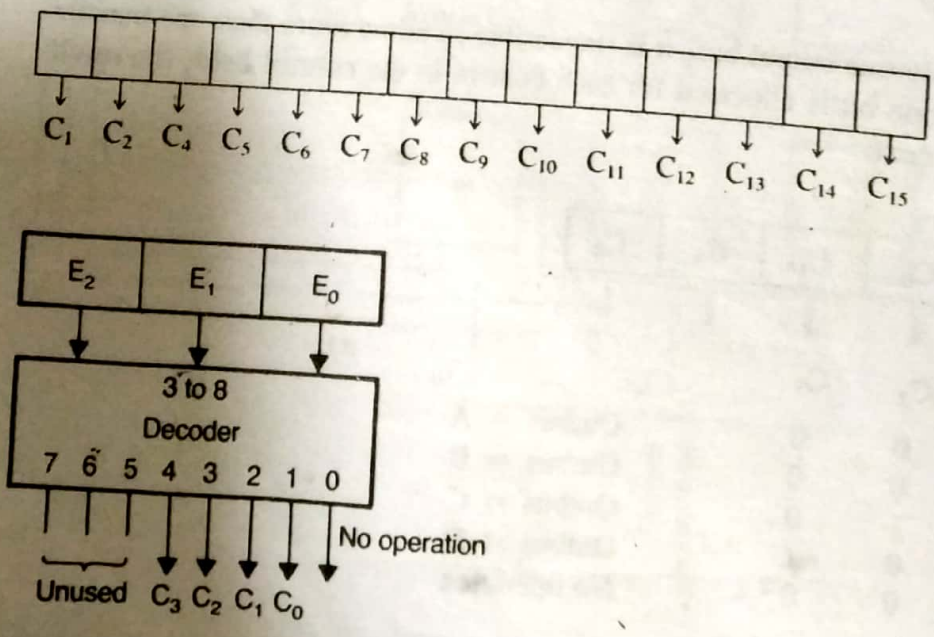
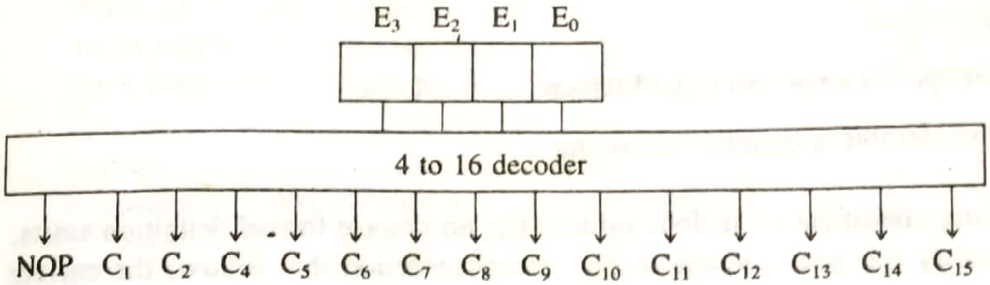
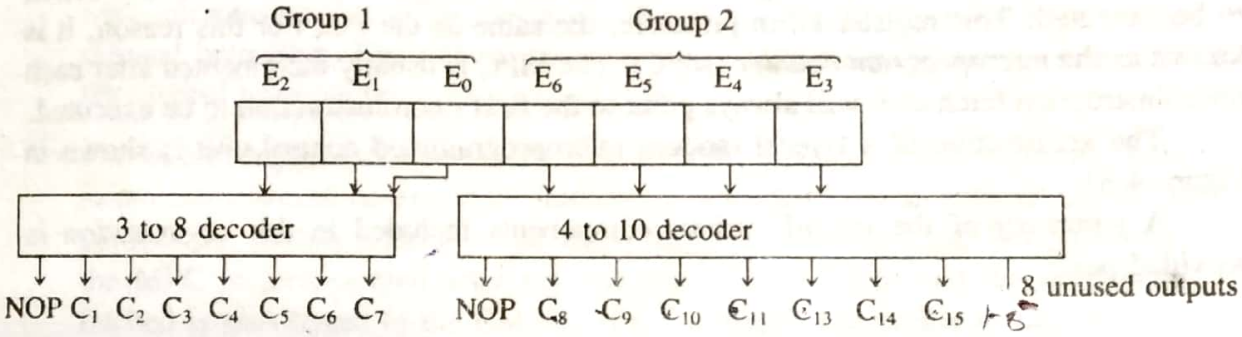



**Figure 4.30** Encoded Control Arrangement

The same information can be specified in the encoded form as shown next:




In the first and second cases, the sizes of the control field are 16 and 4 bits, respectively. The latter approach, however, needs a 4-to-16 decoder to derive the actual control signals. As a measure of compromise, the control information is given by a partial encoding as shown next:




The control signals are partitioned into disjoint groups, so two signals of different groups can be activated in parallel. The control signals have been divided into three groups:

Group 1: $C_1, C_2, C_3, C_4, C_5, C_6, C_7$

Group 2: $C_8, C_9, C_{10}, C_{11}, C_{12}, C_{13}, C_{14}, C_{15}$

With the above grouping, $C_8$ and $C_1$ are activated simultaneously but not $C_1$ and $C_2$. The actual control signals are derived by using one 3 to 8 and one 4 to 10 decoders. In the last case, the control field requires 7 bits which lie midway between the unencoded and fully encoded approaches. Microinstructions are classified into two groups, which express parallelism ability and the amount of encoding called:

- Horizontal
- Vertical

The horizontal microinstruction has the following features:

- Long microinstructions
- Capability of expressing a high degree of parallelism
- Very little encoding

The vertical instruction possesses the following basic attributes:

- Short instructions
- Limited scope for expressing parallelism
- Needs considerable amount of decoding

The preceding classification is done informally; no precise formal definition exists.

How to specify the next address of the microinstruction that follows the current instruction being executed will be discussed. In the original design proposed by M. V. Wilkes, the next address is specified in each microinstruction. A close examination reveals that except in the case of a branch instruction, the address of the next microinstruction to be executed is the address of the memory word that follows the current microinstruction word. Therefore, the next address field from the microinstruction can be eliminated by introducing a separate register used as a pointer to the next instruction to be executed. This register is, in principle, the same as the PC. For this reason, it is known as the *microprogram counter* (MPC). The MPC is usually incremented after each microinstruction fetch so it will always point to the next microinstruction to be executed.

The architecture of a typical modern microprogrammed control unit is shown in Figure 4.31.

A summary of the use of various components included in this organization is provided next:

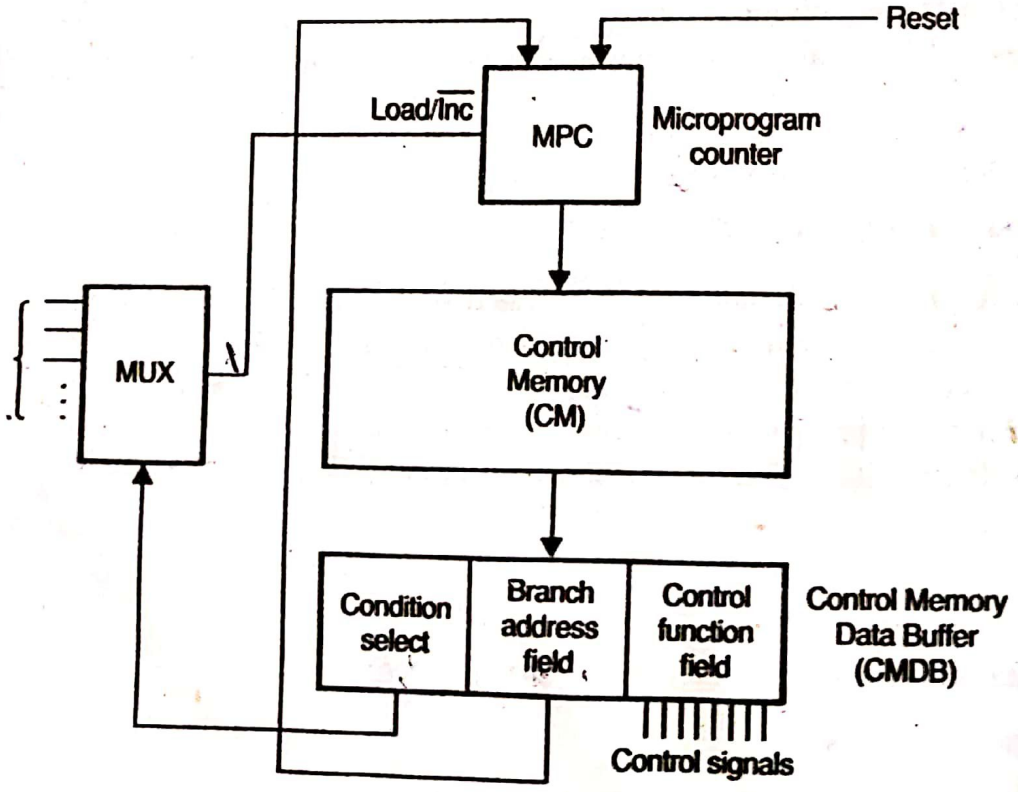



**Figure 4.31** General-purpose Microprogrammed Control Organization (From J. P. Hayes, *Computer Architecture and Organization*, McGraw-Hill, 1978, p. 280. Adapted by permission of McGraw-Hill)

- *Control memory buffer register (CMBR):* The CMBR functions the same as the MBR of the main memory. It is basically a latch, and acts as a buffer for the microinstructions retrieved from the CM. Typically, each microinstruction will have three distinct fields as diagrammed below:

| Condition select | Branch address field | Control function field |
|---|---|---|

The condition *select field* selects the external condition to be tested. If the selected condition is true, the output of the MUX will be 1. Since the output of the MUX is connected to the load input of the MPC, the MPC will be loaded with the address specified in the branch address field of the microinstruction. However, if the selected external condition is false, the MPC will point to the next microinstruction to be executed. Therefore, this arrangement allows conditional branching. The control function field of the microinstruction may hold the control information in an encoded form.

- *The microprogram counter (MPC):* The MPC holds the address of the next microinstruction to be executed. Initially, it is loaded from an external source to point to the starting address of the microprogram to be executed. From then on, the MPC is incremented after ech microinstruction fetch, and the instruction fetched is transferred to the CMBR. When a branch instruction is encountered, the MPC will be loaded with the contents of the branch address field of the microinstruction that is held in the CMBR.

- *External condition select MUX:* This MUX selects one of the external conditions according to the contents of the condition select field of the microinstruction. Therefore, the condition to be selected must be specified in an encoded form. Any encoding leads to a short microinstruction, which implies a small control memory; hence, the cost is reduced. Suppose six external conditions, $X_1$, $X_2$, $X_3$, $X_4$, $X_5$, $X_6$, are to be tested; then the condition-select field and the MUX can be organized as shown in Figure 4.32.

In Figure 4.32, the contents of the condition-select field and actions taken are summarized next:

| CONDITION SELECT | ACTION TAKEN |
|---|---|
| 000 | No branching |
| 001 | Branch if $X_1 = 1$ |
| 010 | Branch if $X_2 = 1$ |
| 011 | Branch if $X_3 = 1$ |

| | |
|---|---|
| 100 | Branch if $X_4 = 1$ |
| 101 | Branch if $X_5 = 1$ |
| 110 | Branch if $X_6 = 1$ |
| 111 | Branch always (unconditional branch) |

If the condition-select field contains 000, the output of the MUX is 0. The MPC will then be incremented to point the next address. Therefore, no branching will take place. When the condition-select field is 111, the output of the MUX is 1, causing the MPC to down-load a branch address. Since this happens regardless of any external condition, an unconditional branch is formed. When the condition-select field is 010, the output of the MUX is the same as the value of the external condition variable $X_2$. Therefore, the MPC will be loaded with a branch address only when $X_2 = 1$; otherwise it is incremented. Conditional branching is achieved. The structure of the control memory will now be addressed.

In the early days, the control memory was organized as a ROM. ROM's were constructed by using a diode matrix, since it accessed faster than ferrite-core read-write memories. Since the control program had little likelihood to change, it was economical to use a ROM rather than a read-write memory.

Present technology allows the use of control memories whose contents may be rewritten. As mentioned earlier, a CPU designed with such a writable control memory can be made to interpret different instruction set, by inserting the appropriate microprogram. In effect, it is possible to produce different virtual machines with the same hardware. When the microcode of CPU A interprets the instruction set of CPU B, machine A emulates machine B. This philosophy is adopted by many leading manufacturers, such as the Burroughs Corporation and IBM.

Microprogramming is the activity of writing microprograms for a microprogrammable processor. Writing microprograms is similar to writing programs in an assembly

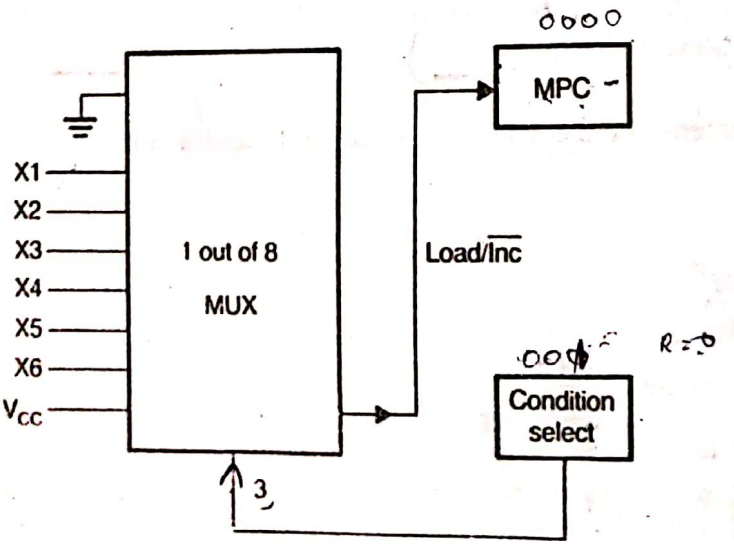



**Figure 4.32** External Condition Select Multiplexer

language. However, a microprogrammer must have a more thorough knowledge about the system's architecture than an assembly language programmer. To speed up the microcode-development process, microprograms are written in a symbolic language called *microassembly language*. These are then translated by a microassembler to produce microcodes that can be held in the CM.

The design of a typical microprogrammed system is now discussed. Consider the design of a microprogrammed control unit for the 4 x 4 Booth's multiplier presented earlier. For the first step, write the microprogram in a symbolic form, as shown in Figure 4.33.

This program is identical to the register transfer description presented earlier. Thus, there is a one-to-one correspondence between the register transfer description of a task and its microprogram implementation. Each line of the above symbolic program is stored as a word in the CM whose address is specified as an integer number. For example, the word stored in the CM address 0 activates operations A ← 0, M ← Inbus, and L ← 4. Similarly, the word stored in the CM address 4 implements the unconditional branch instruction go to RSHIFT. The control memory is able to hold 13 words, requiring a 4-bit branch address field.

In this task, three conditions, Q [1] Q [0] = 01, Q [1] Q [0] = 10, and Z = 0, are checked. These conditions are applied as inputs to the condition select MUX. Additionally, a logic 0 and a logic 1 are applied as data inputs to this MUX to take care of no-branch and unconditional-branch situations, respectively. Therefore, the MUX is able to handle five data inputs and must be at least a 1-out-of-8 data selector. The size of the condition-select field must be 3 bits wide ($2^3 = 8$).

A 3-bit condition-select field guess gives eight distinct 3-bit patterns. However, only the first five 3-bit patterns are used to encode the five different conditions encountered in this problem. With this design, the condition select field may be interpreted as follows on the top of page 164:

| Control Memory Address | | Control Word |
|---|---|---|
| 0 | START | A ← 0, M ← Inbus, L ← 4 |
| 1 | | Q [4:1] ← Inbus, Q [0] ← 0; |
| 2 | LOOP | If Q [1:0] = 01 Then go to ADD |
| 3 | | If Q [1:0] = 10 Then go to SUB |
| 4 | | Go to RSHIFT; |
| 5 | ADD | A ← A + M; |
| 6 | | Go to RSHIFT; |
| 7 | SUB | A ← A − M; |
| 8 | RSHIFT | ASR (A$Q), L ← L − 1; |
| 9 | | If Z = 0 then go to LOOP |
| 10 | | Outbus ≐ A; |
| 11 | | Outbus = Q [4:1]; |
| 12 | HALT | Go to HALT |

**Figure 4.33** Symbolic Microprogram for 4 x 4 Booth's Multiplier

| CONDITION-SELECT FIELD | INTERPRETATION |
|---|---|
| 000 | No branching |
| 001 | Branch if Q [1] = 0 and Q [0] = 1 |
| 010 | Branch if Q [1] = 1 and Q [0] = 0 |
| 011 | Branch if Z = 0 |
| 100 | Unconditional branching |

With these details, the size of the control word is found as follows:

size of a control word = size of the cond.-select field + size of the branch address field + number of control functions

= 3 + 4 + 10

= 17 bits

Hence, the sizes of the CMDB and CM are 17 and 221 (13 x 17) bits, respectively. The complete hardware organization of the control unit is shown in Figure 4.34.

Finally, the production of the binary microprogram stored in the CM will be discussed. As mentioned before, for each line of the symbolic program listing, there exists a control word. For example, consider the first line of the symbolic listing shown (see Figure 4.33). This instruction introduces no branching. Therefore, the condition-select

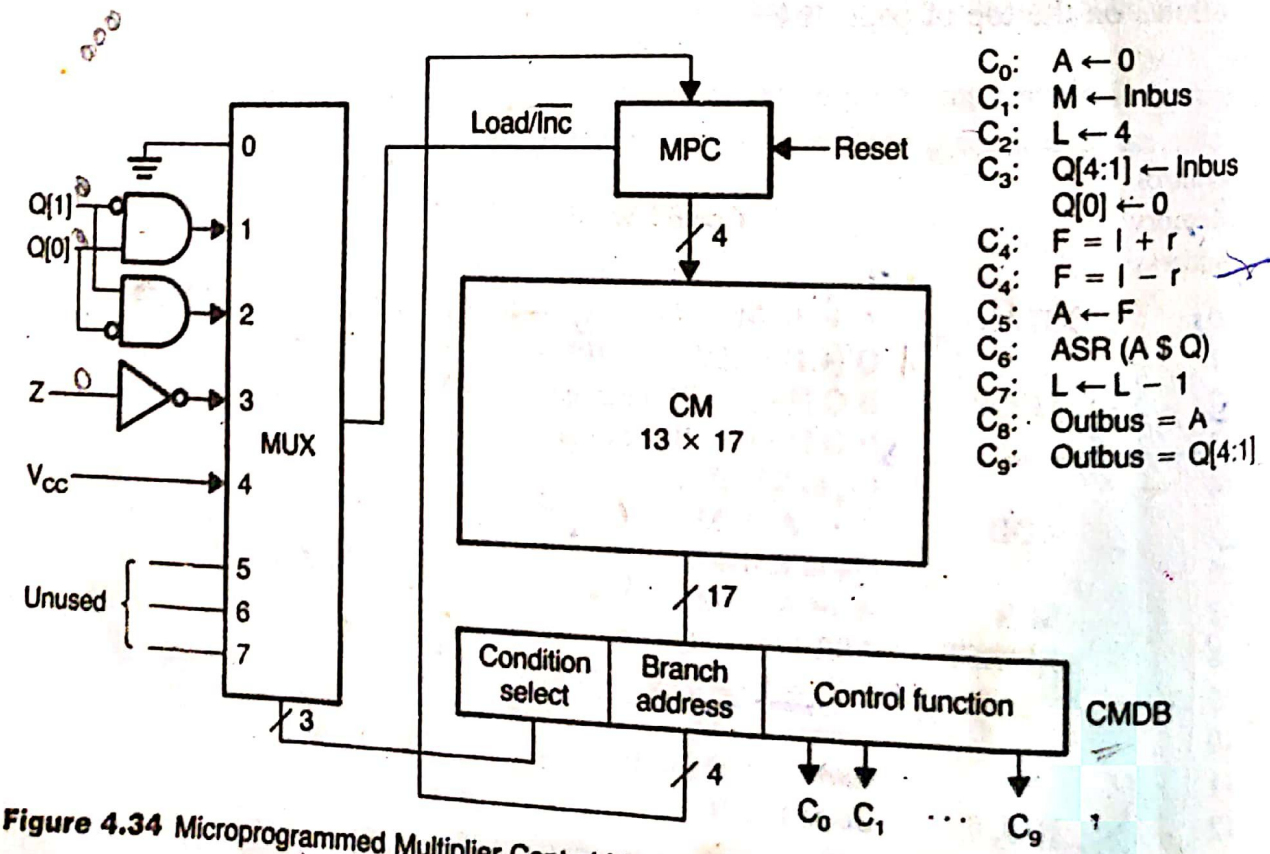



**Figure 4.34** Microprogrammed Multiplier Control Unit

field should be 000. Thus the contents of the branch address field are irrelevant. However, the contents of this field can be reset to 0000 without any loss of generality. In this instruction, three microoperations $C_0$, $C_1$, and $C_2$ are activated. Therefore, only the corresponding bit positions in the control function field are set to 1. This results in the following binary microinstruction:

| Condition Select | Branch Address | Control Function |
|---|---|---|
| 000 | 0000 | 1110000000 |

The binary microinstructions corresponding to the tenth line of the symbolic microprogram does not activate any microoperations. Therefore, all bits of the control function field must be reset to 0. Since the branching is based on the value of the external condition variable Z, the condition select field must be set to 011. If the condition is met, the program control must branch to execute the microinstruction whose symbolic address is LOOP. Since this label appears in line number 2 of the symbolic listing, the branch address is 2 (0010). The value of the branch address field must be 0010. The following binary microinstruction is obtained:

| Condition Select | Branch Address | Control Function |
|---|---|---|
| 011 | 0010 | 0000 0000 0000 |

Continuing in this manner, the complete binary-microprogram can be produced (Figure 4.35).

The microprogrammed Booth's multiplier is timed by using a three-phase clock signal generated as shown in Figure 4.36.

The contents of the CM word whose address is specified in the MPC are transferred into the CMBR with the trailing edge of the timing pulse $P_0$. The processing section responds to the trailing edge of the timing pulse $P_1$. The MPC is then loaded with a branch address or incremented by 1 with the trailing edge of the timing pulse $P_2$. This process continues until the last instruction of the microprogram is reached. Close examination reveals that the microinstruction execution and the next address generation may be overlapped by making both the processing section and the MPC respond to the same timing pulse. Branching and microoperation initiation can thus be combined in the same control word.

The control words stored in CM addresses 5 and 6 (see Figure 4.35) can be combined as one single word as shown next:

| *CONDITION SELECT* | *BRANCH ADDER* | *CONTROL FUNCTION* |
|---|---|---|
| | | $C_0C_1C_2C_3C_4C_5C_6C_7C_8C_9C_{10}$ |
| 100 | 1000 | 0 0 0 0 1 1 0 0 0 0 0; $A \leftarrow A + M$ and go to 8 |

| General Format | Instruction Length in Bytes | Object Code | | Instruction Type | Operation | Comment |
|---|---|---|---|---|---|---|
| | | In binary | In hex | | | |
| LDA ⟨addr⟩ | 2 | 0000 1000 | 08 | MRI | A ← M (⟨addr⟩) | Load accumulator direct |
| | | ⟨addr8⟩ | ⟨addrH⟩ | | | |
| STA ⟨addr⟩ | 2 | 0000 1001 | 09 | MRI | M (⟨addr⟩) ← A | Store accumulator direct |
| | | ⟨addr8⟩ | ⟨addrH⟩ | | | |
| ADD ⟨addr⟩ | 2 | 0000 1010 | 0A | MRI | A ← A + M (⟨addr⟩) | Add accumulator direct |
| | | ⟨addr8⟩ | ⟨addrH⟩ | | | |
| SUB ⟨addr⟩ | 2 | 0000 1011 | 0B | MRI | A ← A − M (⟨addr⟩) | Subtract accumulator direct |
| | | ⟨addr8⟩ | ⟨addrH⟩ | | | |
| JZ ⟨addr⟩ | 2 | 0000 1100 | OC | MRI | If Z = 1 then PC ← ⟨addr⟩ else PC ← PC + 1 | Jump on zero flag set |
| | | ⟨addr8⟩ | ⟨addrH⟩ | | | |
| JC ⟨addr⟩ | 2 | 0000 1101 | 0D | MRI | If C = 1 then PC ← ⟨addr⟩ else PC ← PC + 1 | Jump on carry flag set |
| | | ⟨addr8⟩ | ⟨addrH⟩ | | | |
| AND ⟨addr⟩ | 2 | 0000 1110 | 0E | MRI | A ← A ∧ M (⟨addr⟩) | And accumulator direct |
| | | | ⟨addrH⟩ | | | |
| CMA | 1 | 0000 0000 | 00 | NMRI | A ← A′ | Complement accumulator |
| INCA | 1 | 0000 0010 | 02 | NMRI | A ← A + 1 | Increment accumulator |
| DCRA | 1 | 0000 0100 | 04 | NMRI | A ← A − 1 | Decrement accumulator |
| HLT | 1 | 0000 0110 | 06 | NMRI | Halt | Halt CPU. |

*Note:*

⟨addr8⟩: 8-bit memory address in binary
⟨addrH⟩: 8-bit memory address in hex
MRI: memory reference instruction
NMRI: nonmemory reference instruction.

**Figure 4.39** Instruction Set to be Implemented.

| ROM Address | | | | | Cond. Sel | | | Branch Addr. | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| In decimal | In binary | | | | $c_{s2}$ | $c_{s1}$ | $c_{s0}$ | $b_{r3}$ | $b_{r2}$ | $b_{r1}$ | $b_{r0}$ | $c_0$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 3 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| 4 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 7 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 10 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 11 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 12 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |

**Figure 4.35** Binary Listing of the Microprogram For the 4 x 4 Booth's Multiplier

When doing so, however, one must make sure that the system operation produces the correct result. By separately handling branch instructions (as in Figure 4.35), a new strategy for minimizing the size of the control store can be divised. This method is referred to as *multiple microinstruction format* and is covered later.

Next, the design of a microprogrammed processor is illustrated. The programming model of this processor is shown in Figure 4.37.

The CPU entails two registers:

1. An 8-bit accumulator register A
2. A 2-bit flag register F

The flag register holds only zero (Z) and carry (C) flags. All programs and data are stored in the 256 x 8 RAM. The detailed hardware schematic of the data-flow part of this processor is shown in Figure 4.38.

From Figure 4.38, it can be seen that the hardware organization includes four more 8-bit registers, PC, IR, MAR, and BUFFER. These registers are transparent to a programmer. The 8-bit register BUFFER is used to hold the data that is retrieved from memory. In this system, only a restricted number of data paths are available. These paths are controlled by the control inputs $C_0$ through $C_9$, as described on page 167 bottom.

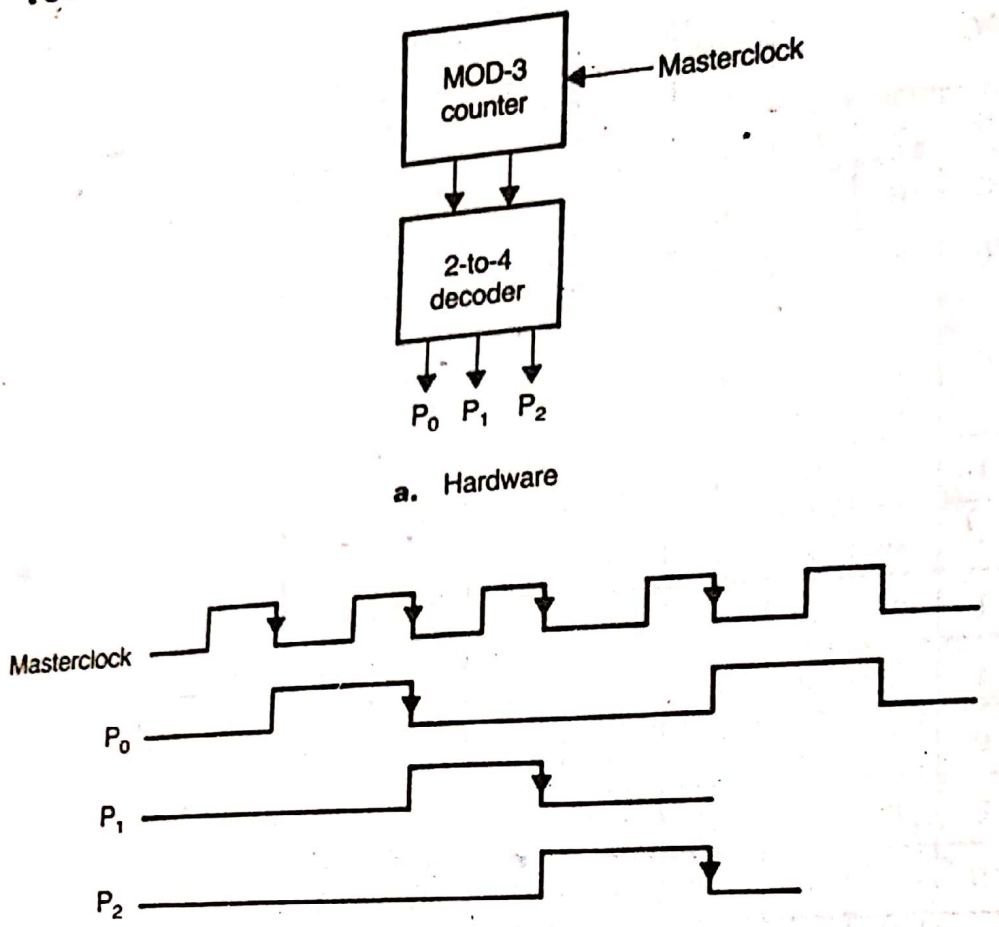



**Figure 4.36** Microinstruction Timing

The ALU in this hardware can perform eight distinct operations. They are controlled by the select inputs $C_{10}$, $C_{11}$, and $C_{12}$ as follows:

| $C_{10}$ | $C_{11}$ | $C_{12}$ | F |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | R |
| 0 | 1 | 0 | L + R |
| 0 | 1 | 1 | L − R |
| 1 | 0 | 0 | L + 1 |
| 1 | 0 | 1 | L − 1 |
| 1 | 1 | 0 | L ∧ R |
| 1 | 1 | 1 | L′ |

Implementation of the instruction set described in Figure 4.39 is now shown by writing a suitable microprogram.

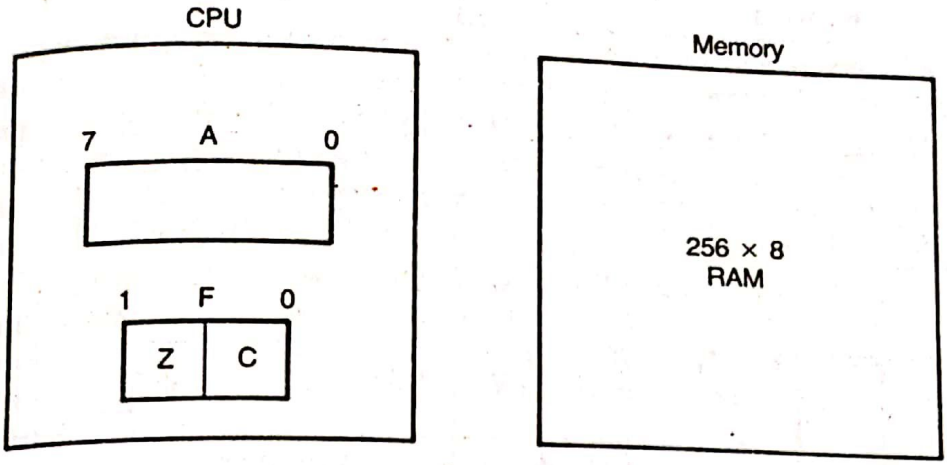



**Figure 4.37** Programming Model of a Simple Processor

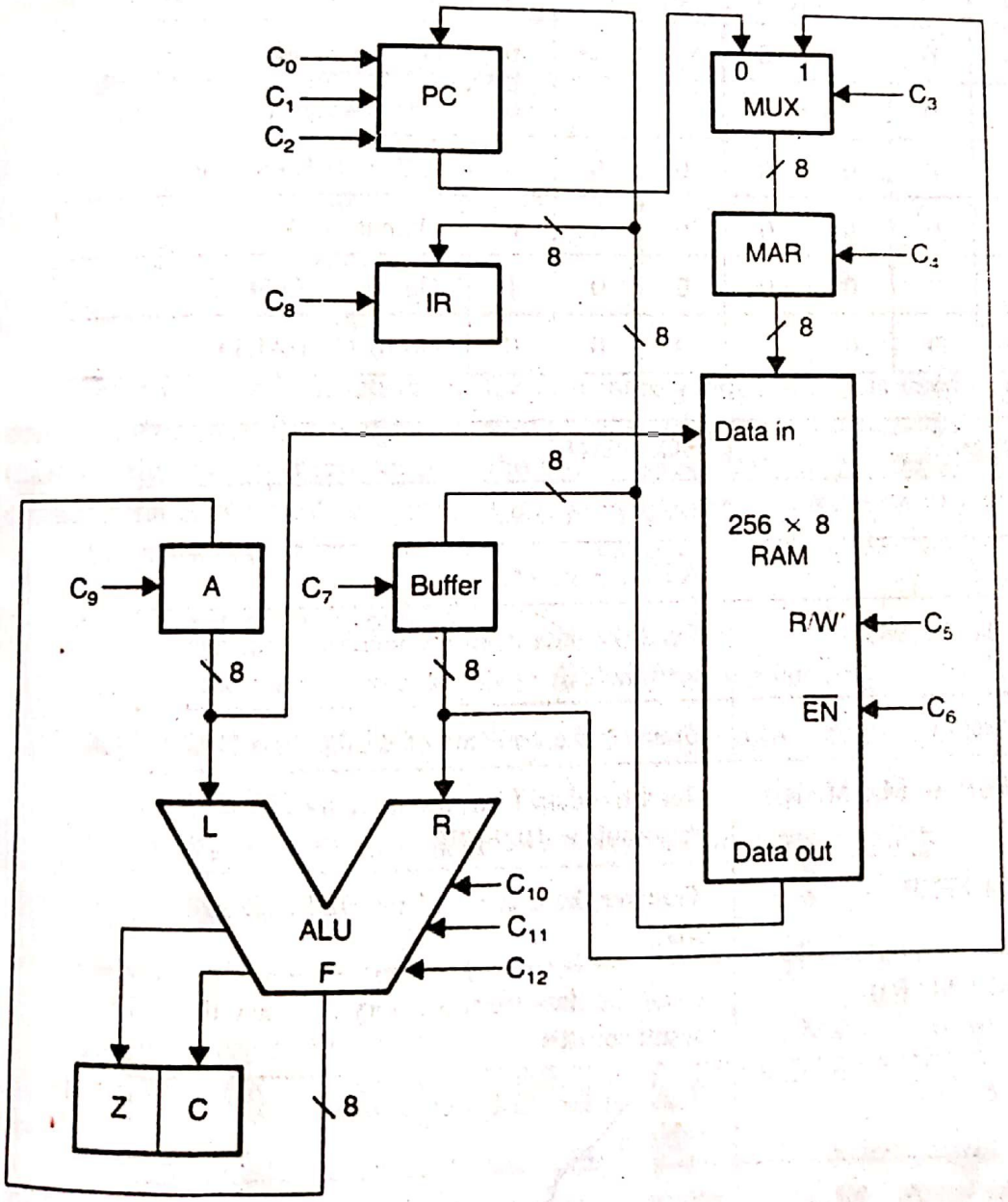



**Figure 4.38** Hardware Schematic of the Simple Processor

| Control Word | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Control Function | | | | | | | | | Comments |
| $c_1$ | $c_2$ | $c_3$ | $c_4$ | $c_5$ | $c_6$ | $c_7$ | $c_8$ | $c_9$ | |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | A ← 0, M ← Inbus, L ← 4 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | Q [4:1] ← Inbus, Q [0] ← 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | If Q [1:0] = 01 Then go to 5 (ADD) |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | If Q [1:0] = 10 Then go to 7 (SUB) |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | go to 8 (RSHIFT) |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | A ← A + M |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | go to 8 (RSHIFT) |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | A ← A − M |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | ASR (A$Q), L ← L − 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | If Z = 0 Then go to 2 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | Outbus = A |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | Outbus = Q [4:1] |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Go to 12 (HALT) |

| MICROOPERATION | COMMENT |
|---|---|
| $C_0$: PC ← 0 | Clear PC to zero. |
| $C_1$: PC ← PC + 1 | Advance the PC. |
| $C_2C_5C_6'$: PC ← M ((MAR)) | Read the data from the memory and save it in the PC. |
| $C_3'C_4$: MAR ← PC | Transfer the contents of the PC into MAR. |
| $C_5 C_6' C_7$: BUFFER ← M ((MAR)) | Read the data from the memory and save the result in BUFFER. |
| $C_3C_4$: MAR ← BUFFER | Transfer the content of the BUFFER into MAR. |
| $C_5C_6'C_8$: IR ← M ((MAR)) | Read the data from memory and save the result into IR. |
| $C_9$: A ← F | Transfer the ALU output into the A register. |
| $C_7'C_6'$: M ((MAR)) ← A | Save the accumulator contents in the memory. |

From Figure 4.39, notice that the proposed instruction set contains 11 instructions. The first 7 instructions are classified as memory reference instructions, since they all require a memory address (which is an 8-bit quantity in this case). The last 4 instructions do not require any memory address; they are called nonmemory reference instructions. Each memory reference instruction is assumed to occupy 2 consecutive bytes in the RAM. The first byte is reserved for the op-code, and the second byte indicates the 8-bit memory address. In contrast, a nonmemory reference instruction takes only one byte of storage. This instruction set supports only two addressing modes: implicit and direct. Both branch instructions are assumed to be absolute mode branch instructions. The op-code encoding for this instruction set is carried out in a logical manner, as explained in Figure 4.40.

The bit $I_3$ of Figure 4.40 decides the instruction type. If $I_3 = 1$, it is a memory reference instruction (MRI), otherwise it is a nonmemory reference instruction (NMRI).

Within the memory reference category, instructions are classified into four groups, as follows:

| *GROUP NO.* | *INSTRUCTIONS* |
|---|---|
| 0 | Load and store |
| 1 | Add and subtract |
| 2 | Jumps |
| 3 | Logical |

There are two instructions in the first three groups. Bit $I_0$ is used to determine the desired instruction of a particular group. If $I_0$ of group 0 equals zero, it is the load (LDA) instruction; otherwise it is the store (STA) instruction. Nevertheless, no such classification is required for group 3 and the nonmemory reference instructions.

As mentioned before, the instruction execution involves the following steps:

Step 1: Fetch the instruction.

Step 2: Decode the instruction to find out the required operation.

Step 3: If the required operation is a halt operation, then go to Step 6; otherwise continue.

Step 4: Retrieve the operands and perform the desired operation.

Step 5: Go to Step 1.

Step 6: Execute an infinite LOOP.

The first step is known as the fetch cycle, and the rest are collectively known as the execution cycle. To decode the instruction, the hardware shown in Figure 4.41 is used.

With this hardware and the status flags (Z and C), a microprogram to implement the instruction set can be written. The symbolic version of this micorprogram is shown in Figure 4.42.

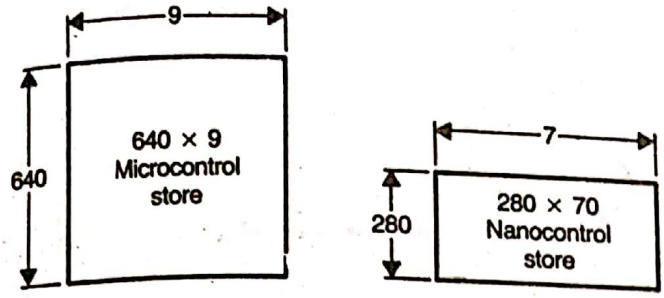



**Figure 4.55** MC68000 Control Unit Structure

As a practical example of the nanomemory concept, consider the nanomemory structure of the Motorola MC68000 16-bit microprocessor in Figure 4.55.

From Figure 4.55 it can be seen that out of 640 microinstructions, 280 are unique. The contents of the microcontrol store are pointers to the nanocontrol store. Each word of the microcontrol store is $\lceil \log_2 280 \rceil = 9$ bits wide.

The MC 68000 offers control memory savings. In the MC68000, the microcontrol store is 640 × 9 bits, and the nanocontrol store is 280 × 70 bits, since there are 280 unique microinstructions. If the MC68000 is implemented by using a single CM, this memory will have 640 × 70 bits. Therefore,

$$\begin{aligned} \text{memory savings} &= 640 \times 70 - (640 \times 9 + 280 \times 70) \\ &= 44{,}800 - (5760 + 19{,}600) \\ &= 19{,}440 \text{ bits} \end{aligned}$$

## QUESTIONS AND PROBLEMS

**4.1** Explain the functions of a control unit in a digital computer.

**4.2** Build hardware to implement each of the following register transfers:

**a)** If X is even then $A \leftarrow B + C$
else $A \leftarrow (B.C)'$

**b)** If X is zero then $A \leftarrow A + 1$
else $A \leftarrow A - 1$

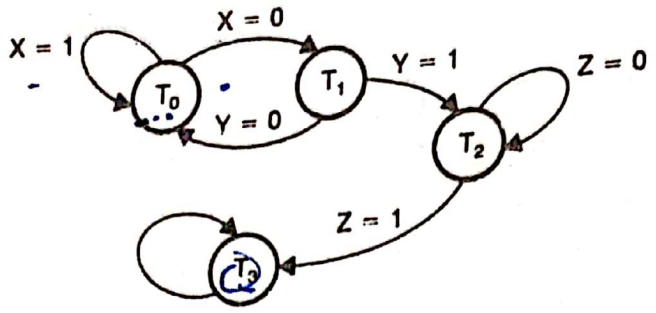
Assume that A, B, C, and X are 4-bit registers.

**4.3** Design a circuit that will generate 19 timing signals.

**a)** Use a straight ring counter.

**b)** Use a mod-32 counter and a 5-to-32 decoder.

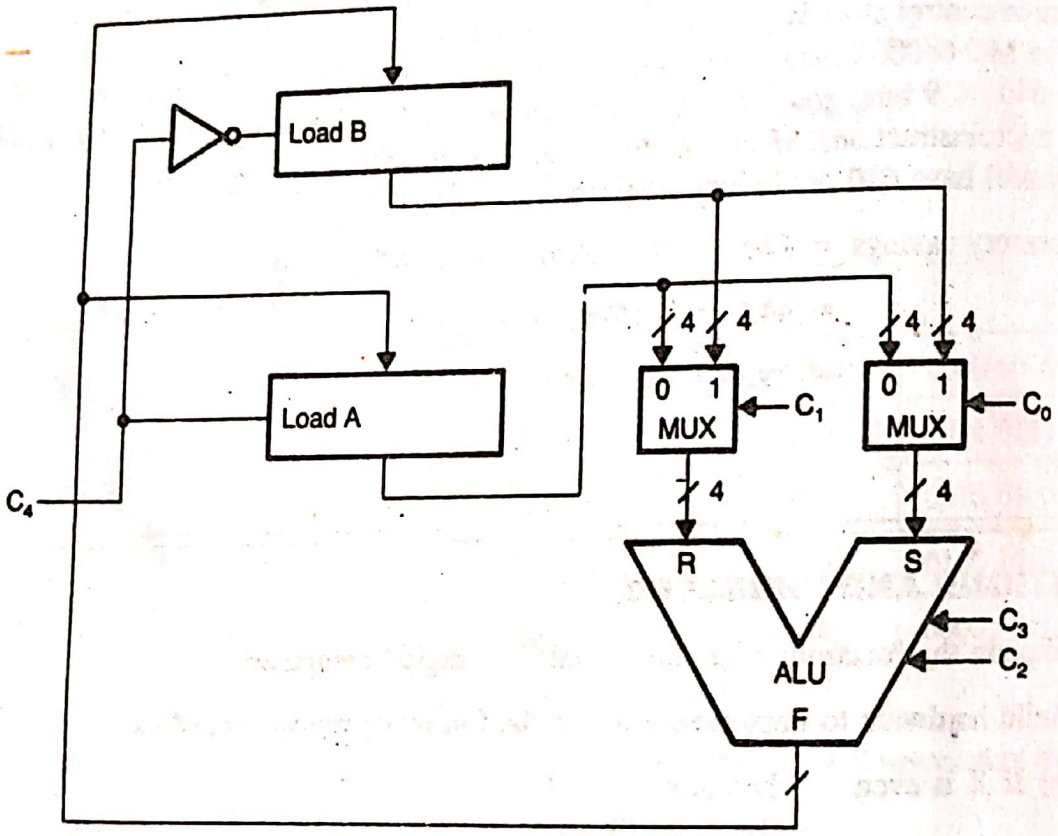
**4.4** A control circuit has three inputs X, Y, and Z, and its state diagram is shown next:




Design the control circuit using counter, decoder and a PLA.

**4.5** Repeat problem 4.4 with D flip-flops and a PLA.

**4.6** Consider the RALU shown next:




The interpretation of various control points are summarized as follows:

| $C_3C_2$ | F |
|---|---|
| 0 0 | R plus S |
| 0 1 | R minus S |
| 1 0 | R and S |
| 1 1 | R EX-OR S |

| $C_1C_0$ | R-INPUT | S-INPUT |
|---|---|---|
| 0 0 | A | A |
| 0 1 | A | B |
| 1 0 | B | A |
| 1 1 | B | B |

| $C_4$ | ACTION |
|---|---|
| 0 | B ← F |
| 1 | A ← F |

Answer the following questions by writing suitable conbtrol word(s). Each control word must be specified according to the following format:

$C_4\ C_3\ C_2\ C_1\ C_0$

For example:

$C_4\ C_3\ C_2\ C_2\ C_0$

1 0 0 0 1 ; A ← A plus B

**a)** How will the A register be cleared? (Suggest at least two possible ways.) DIRECT CLEAR input is not available.

**b)** Suggest a sequence of control words that exchanges the contents of A and B registers (exchange means A ← B and B ← A).

**4.7** Consider the following algorithm:

```
Declare registers A [8], B [8], C [8];
START:  A ← 0;
        B ← 00001010;
LOOP:   A ← A + B;
        B ← B − 1;
        If B < > 0 then go to LOOP
        C ← A;
HALT:   Go to HALT
```

Design a hardwired controller that will implement this algorithm.

**4.8** It is desired to build an interface in order establish communication between a 32-bit host computer and a front end 8-bit micro computer (See figure shown below). The operation of this system is described as follows:

Step 1: First the host processor puts a high signal on the line "want" (saying that it needs a 32-bit data) for one clock period.

Step 2: The interface recognizes this by polling the want line.

Step 3: The interface unit puts a high signal on the line "fetch" for one clock period (that is it instructs the microcomputer to fetch an 8-bit data).

Step 4: In response to this, the microcomputer samples the speech signal, converts it into an 8-bit digital data and informs the interface that the data is ready by placing a high signal on the "ready" line for one clock period.

Step 5: The interface recognizes this (by polling the ready line), and it reads the 8-bit data into its internal register.

Step 6: The interface unit repeats the steps 3 through 5 for three more times (so that it acquires 32-bit data from the microcomputer).

Step 7: The interface informs the host computer that the latter can read the 32-bit data by placing a high signal on the line "takeit" for one clock period.

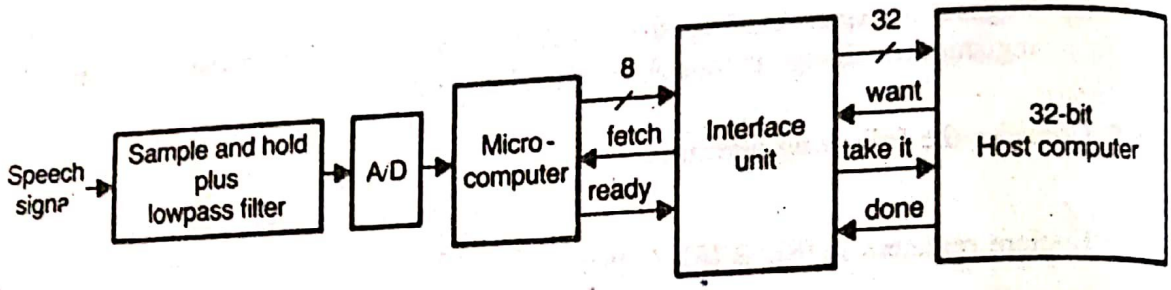
Step 8: The interface unit maintains a valid 32-bit data on the 32-bit output bus until the host processor says that it is done (the host puts a high signal on the line "done" for one clock period). In this case, the interface proceeds to step 1 and looks for a high on the "want" line.

**a)** Provide a RTL description of the interface

**b)** Design the processing section of the interface.

**c)** Draw a block diagram of the interface controller.

**d)** Draw state diagram of the interface controller.




**4.9** Consider the following algorithm:

```
begin
      m: = 14
      q: = 5;
      i: = 1;
      s: = 0;
      While   i <= m do
      begin
         j: = 1
         while j <= q do
         begin
            s: = s + 1;
            j: = j + 1
         end;
         i: = i + 1
      end
end
```

Explain the task accomplished by this algorithm.

**4.10** Explain the significance of horizontal and vertical microinstructions.

**4.11** Solve Problem 4.7 using the microprogrammed approach.

**4.12** Obtain the binary listing of the symbolic microprogram shown in Figure 4.42

**4.13** Design a microprogrammed system to add numbers stored in the register pair AB

and CD. A, B, C, and D are 8-bit registers. The sum is to be saved in the register pair AB. Assume that only an 8-bit adder is available.

**4.14** The goal of this problem is to design a microprogrammed 3rd order FIR (Finite impulse response) digital filter. In this system, there are 4 coefficients $w_0$, $w_1$, $w_2$, and $w_3$. The output $y_k$ (at the kth clock period) is the discrete convolution product of the inputs ($x_k$s) and the filter coefficients. This is formally expressed as follows:

$$y_k = w_0 * x_k + w_1 * x_{k-1} + w_2 * x_{k-2} + w_3 * x_{k-3}$$

$$= \sum_{i=0}^{3} w_i x_{k-i}$$

In the above summation, $x_k$ represents the input at the kth clock period while $x_{k-i}$ represents input at (k-i)th sample period. For all practical purposes, we assume that our system is causal and so $x_i = 0$ for $i < 0$. The processing hardware is shown below. This unit includes 8 eight-bit registers (to hold data and coefficients), A/D (Analog digital converter), MAC (multiplier accumulator), and a D/A (Digital analog converter). The processing sequence is shown below:

1 Initialize coefficient registers
2 Clear all data registers except $x_k$
3 Start A/D conversion (first make sc = 1 and then retract it to 0)
4 Wait for one control state (To make sure that the conversion is complete)
5 Read the digitized data into the register $x_k$
6 Iteratively calculate filter output $y_k$ (use MAC for this)
7 Pass $y_k$ to D/A (Pass Accumulator's output to D/A via Rounding ROM)
8 Move the data to reflect the time shift ($x_{k-3} := x_{k-2}$, $x_{k-2} := x_{k-1}$, $x_{k-1} := x_k$)
9 Go to 3

**a)** Specify the controller organization.

**b)** Produce a well documented listing of the binary microprogram