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.

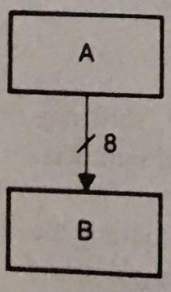


Figure 4.1 A Simple Register Transfer from A to B

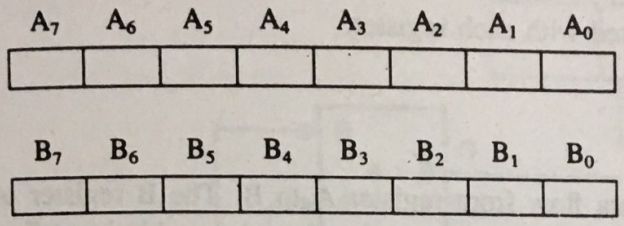


Figure 4.2 Convention Used to Mark Individual Bits of a Register

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] ← 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.

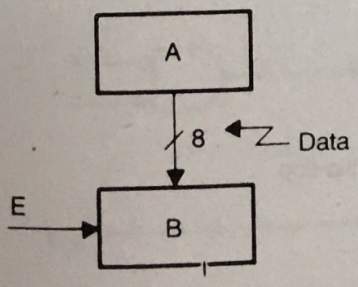


Figure 4.3 A Register Transfer Controlled by an Enable Input

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:

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

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

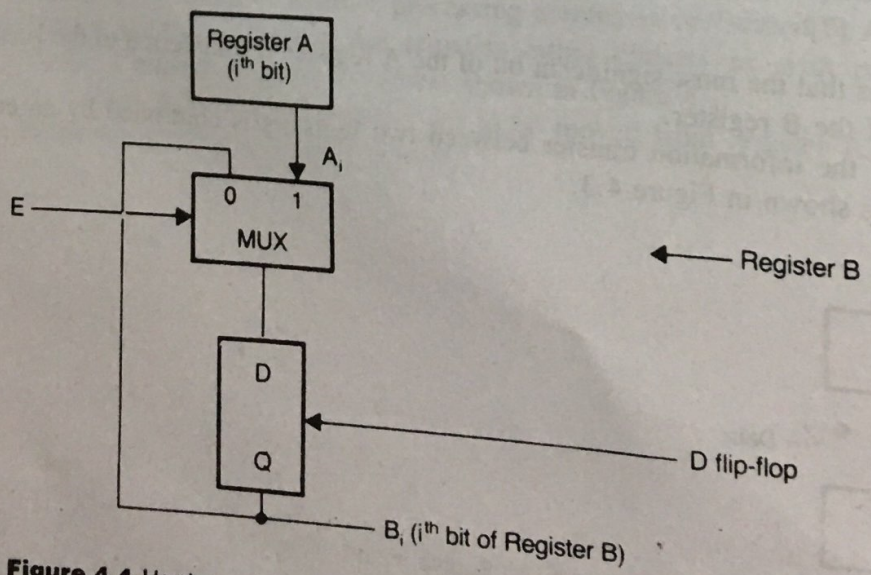


Figure 4.4 Hardware Implementation of a Register with Enable Input

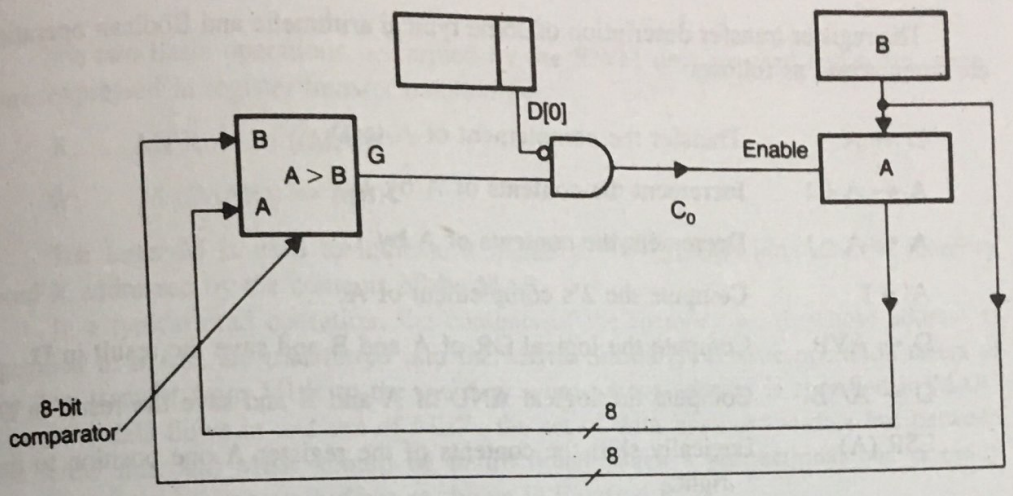


Figure 4.5 Hardware Implementation of C_0: A ← B where C_0 = G D [0]′

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:

$C_0 : A \leftarrow B$

$C_0' : A \leftarrow D$

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.

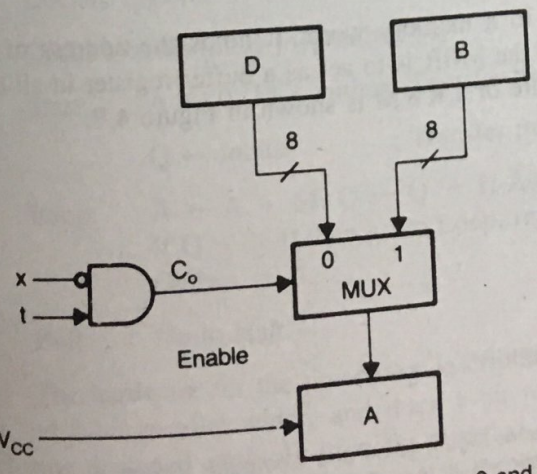


Figure 4.6 Hardware Implementation if x = 0 and t = 1, then A ← B else A ← D

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.
$D \leftarrow AVB$	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.

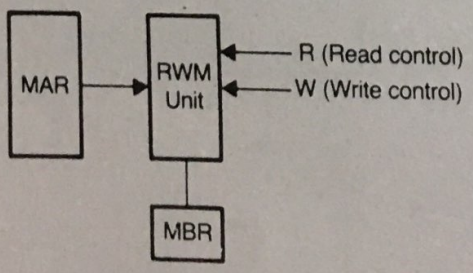


Figure 4.7 A Basic RWM Unit (M. Morris Mano, *Digital Logic and Computer Design*, © 1979, p. 314. Reprinted by permission of Prentice-Hall, Inc., Englewood Cliffs, New Jersey)

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

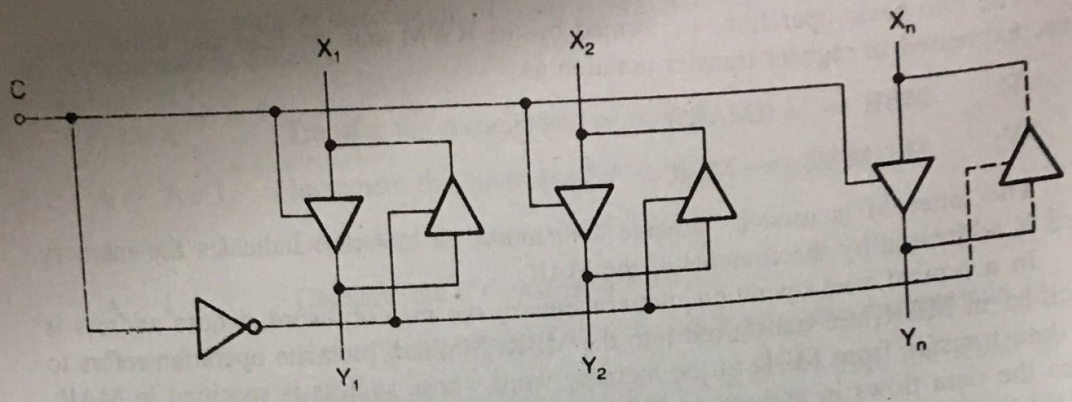


Figure 4.8 Bidirectional Data Bus

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.

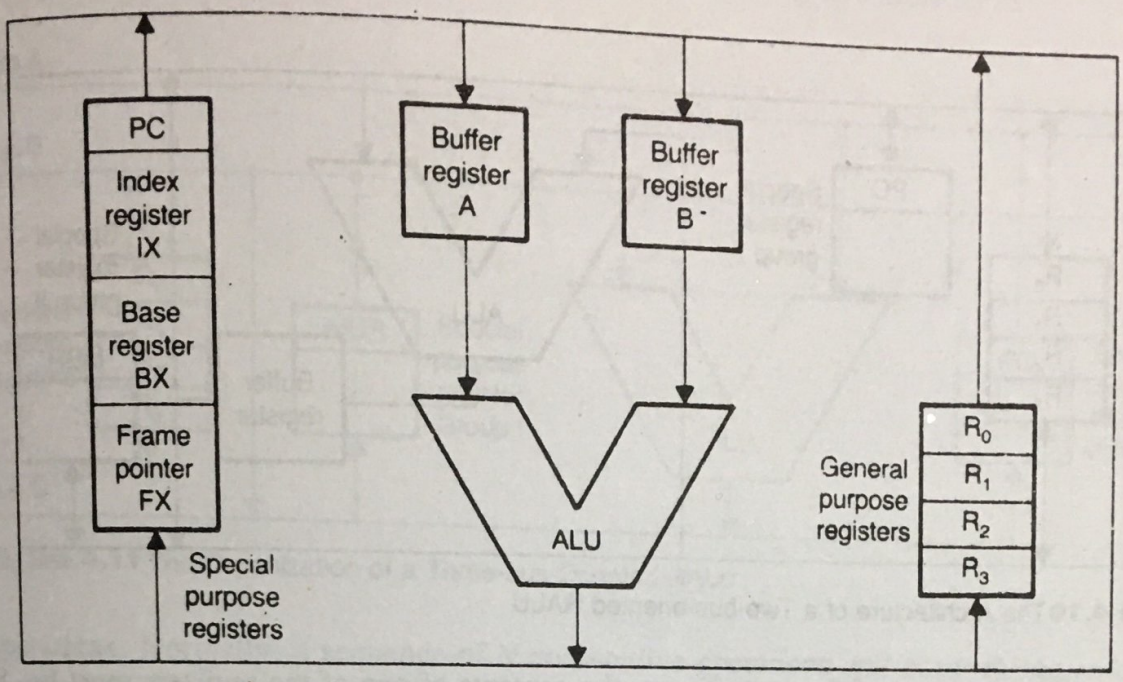


Figure 4.9 Organization of a Single Bus-oriented RALU

- 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

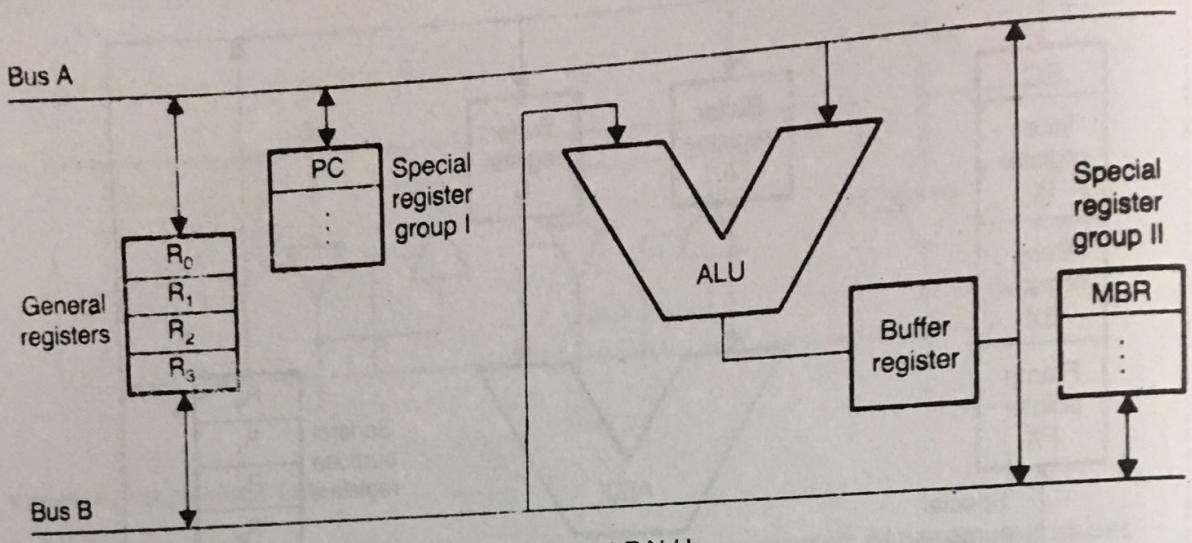


Figure 4.10 The Architecture of a Two-bus-oriented RALU

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

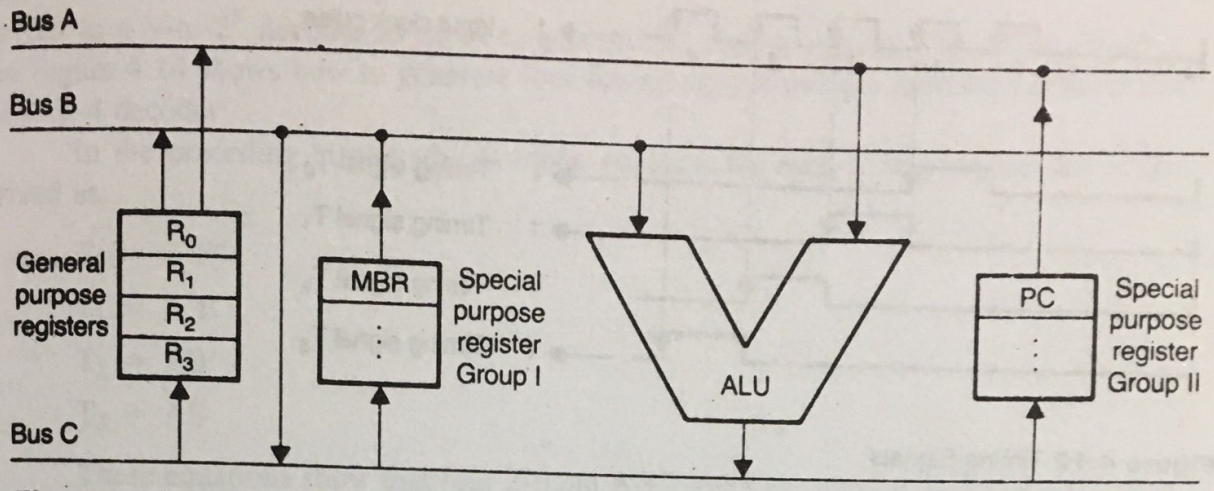


Figure 4.11 The Organization of a Three-bus-Oriented RALU

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 ith 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 ith 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 ith 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

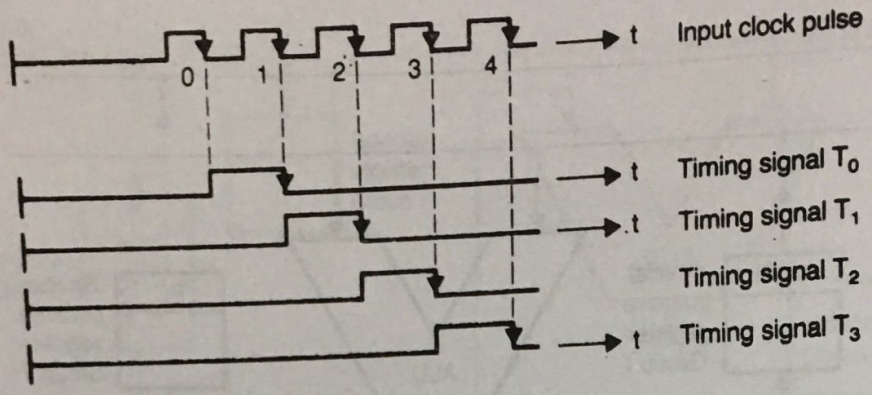


Figure 4.12 Timing Signals

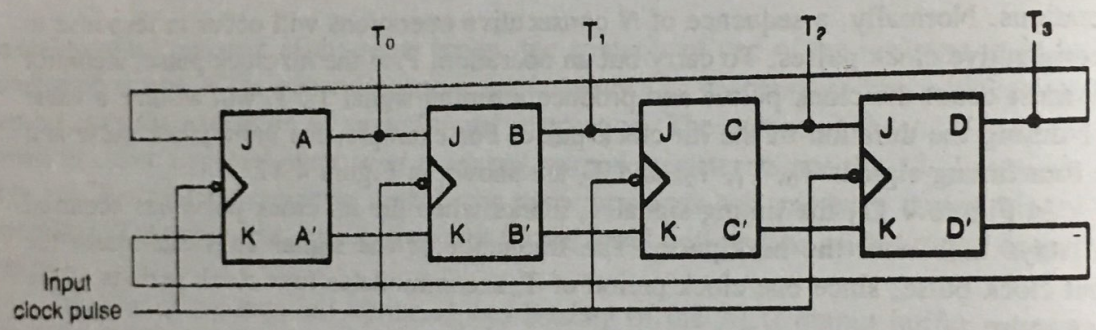


Figure 4.13 Ring Counter

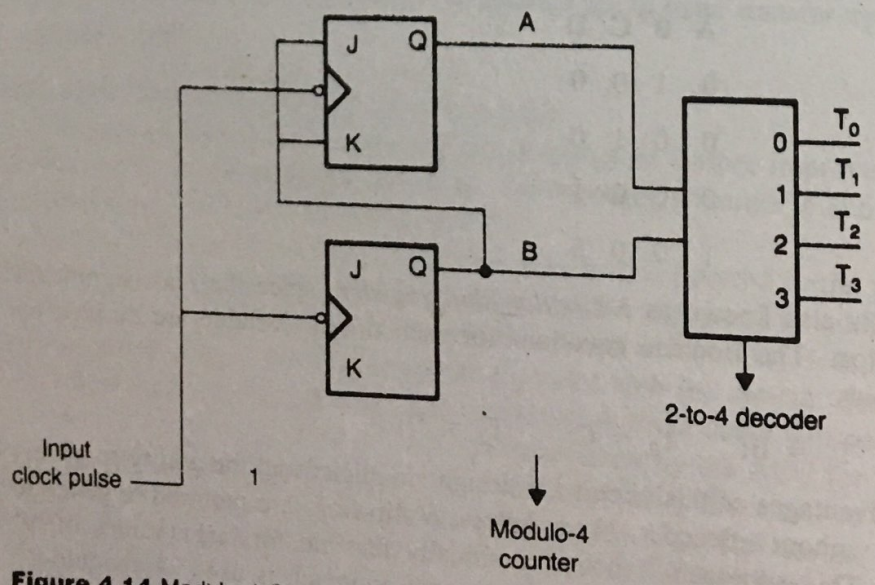


Figure 4.14 Modulo-4 Counter with a Decoder

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.

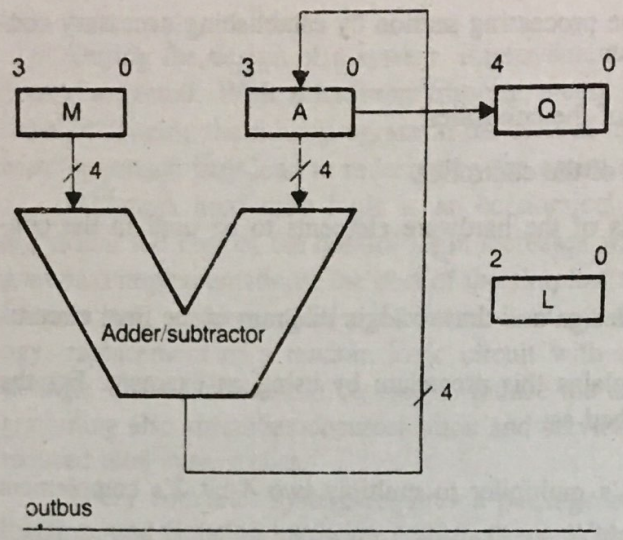


Figure 4.15 Organization of the Processing Section for Performing 4 × 4 Multiplication Using Booth's Algorithm

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 $M = 0111_2 = 7_{10}$

	M	A	MULTIPLIER BITS INSPECTED Q	L
Initialization	1100	0000	011[10]	100
Iteration 1				
$A \leftarrow A - M$	1100	0100	01110	100
ASR (A$Q), $L \leftarrow L-1$	1100	0010	001[11]	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).

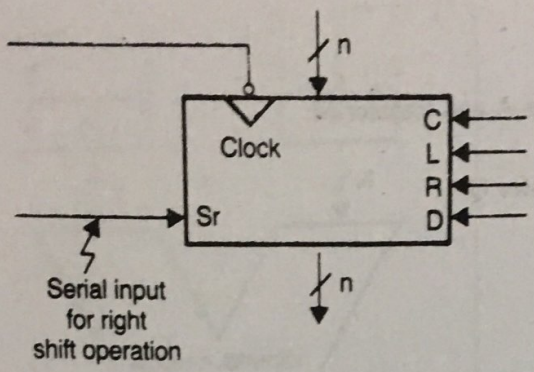
Notice that the general register is basically a trailing-edge-triggered device. Typically, four operations (clear, parallel load, right shift, and decrement) can be performed by introducing the proper values to control inputs C, L, R, and D. All these operations are synchronized with the trailing edge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

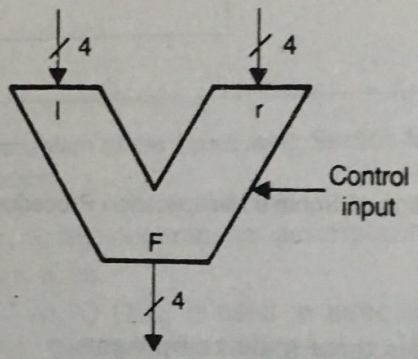
From this figure, we observe that there are 10 control points, C_0, C_1, C_2, . . ., C_9.

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.



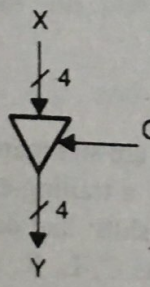
C	L	R	D	Clock	Action
1	0	0	0	↓	Clear
0	1	0	0	↓	Load external data
0	0	1	0	↓	Right shift
0	0	0	1	↓	Decrement by one
0	0	0	0	↓	No change

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.

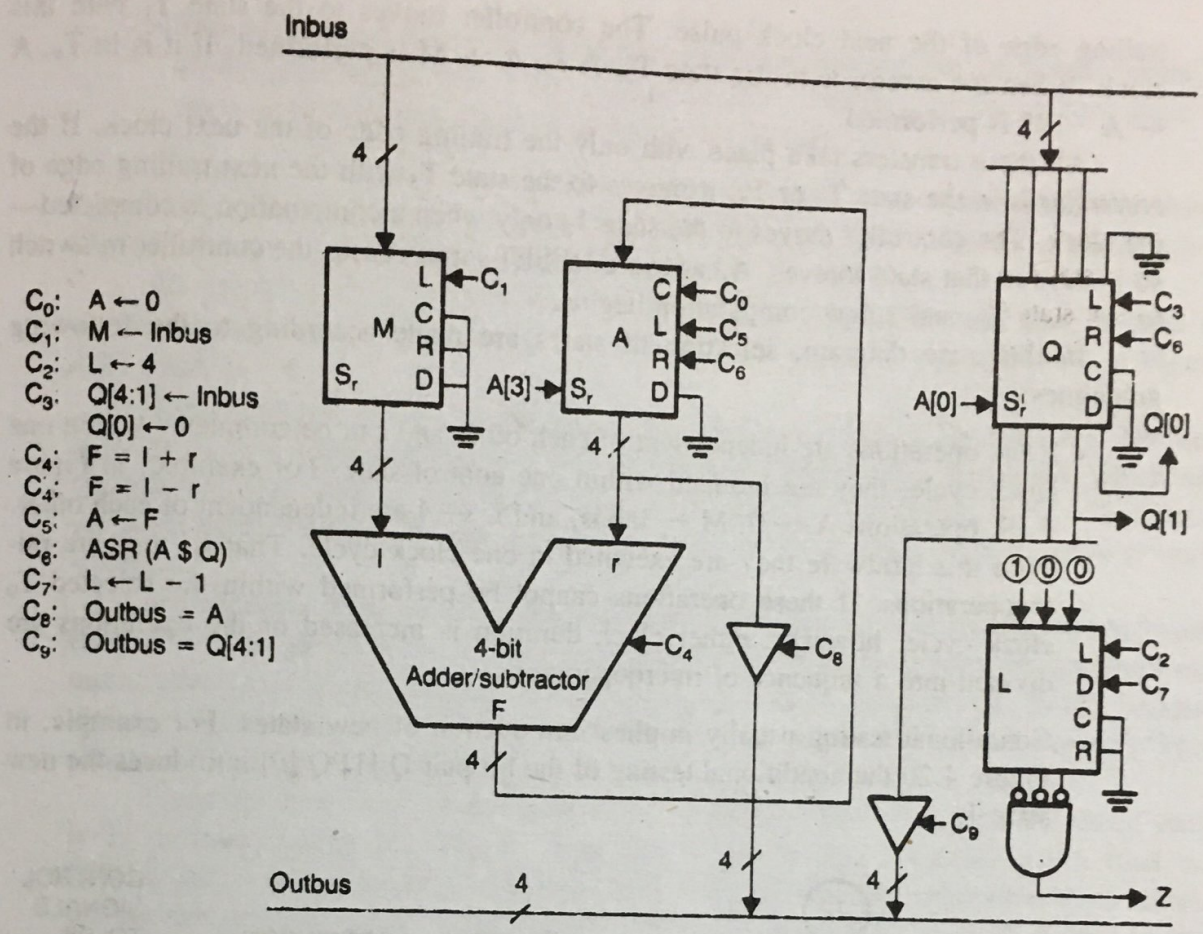


Figure 4.18 Processing Section of the Booth's Multiplier

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 ← 0, L ← 4, and M ← Inbus are carried out with the

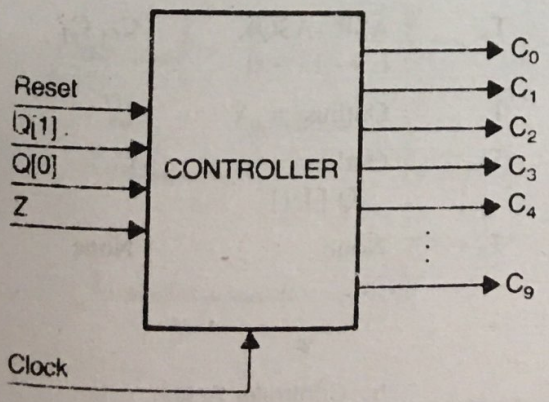


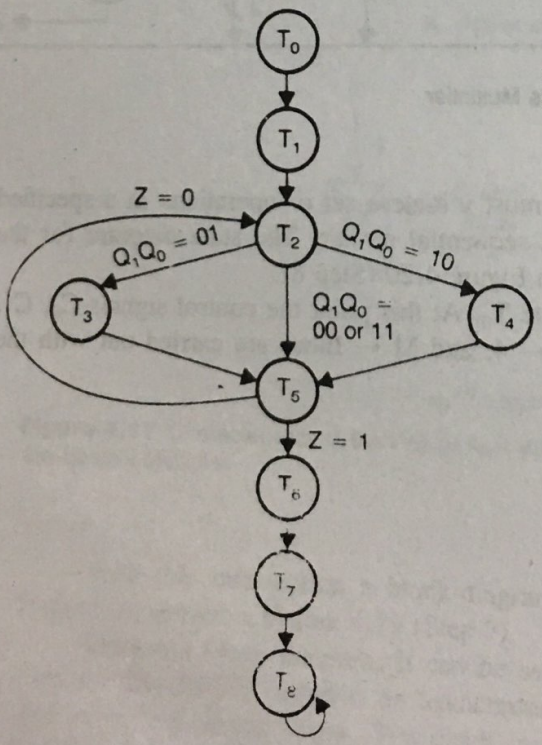
Figure 4.19 Block-diagram of the Booth's Multiplier Controller

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 tc 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 $d_3 d_2 d_1 d_0 = 0010$ 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

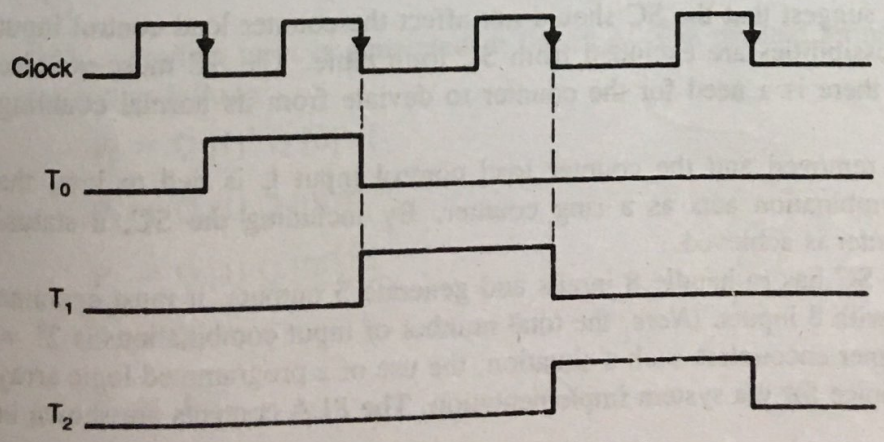
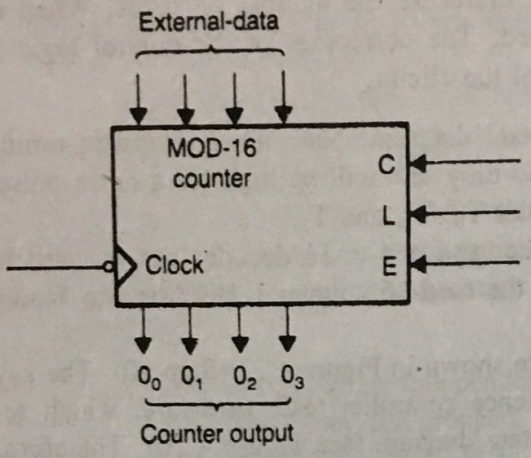


Figure 4.21 Timing Signals Generated by the Controller



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

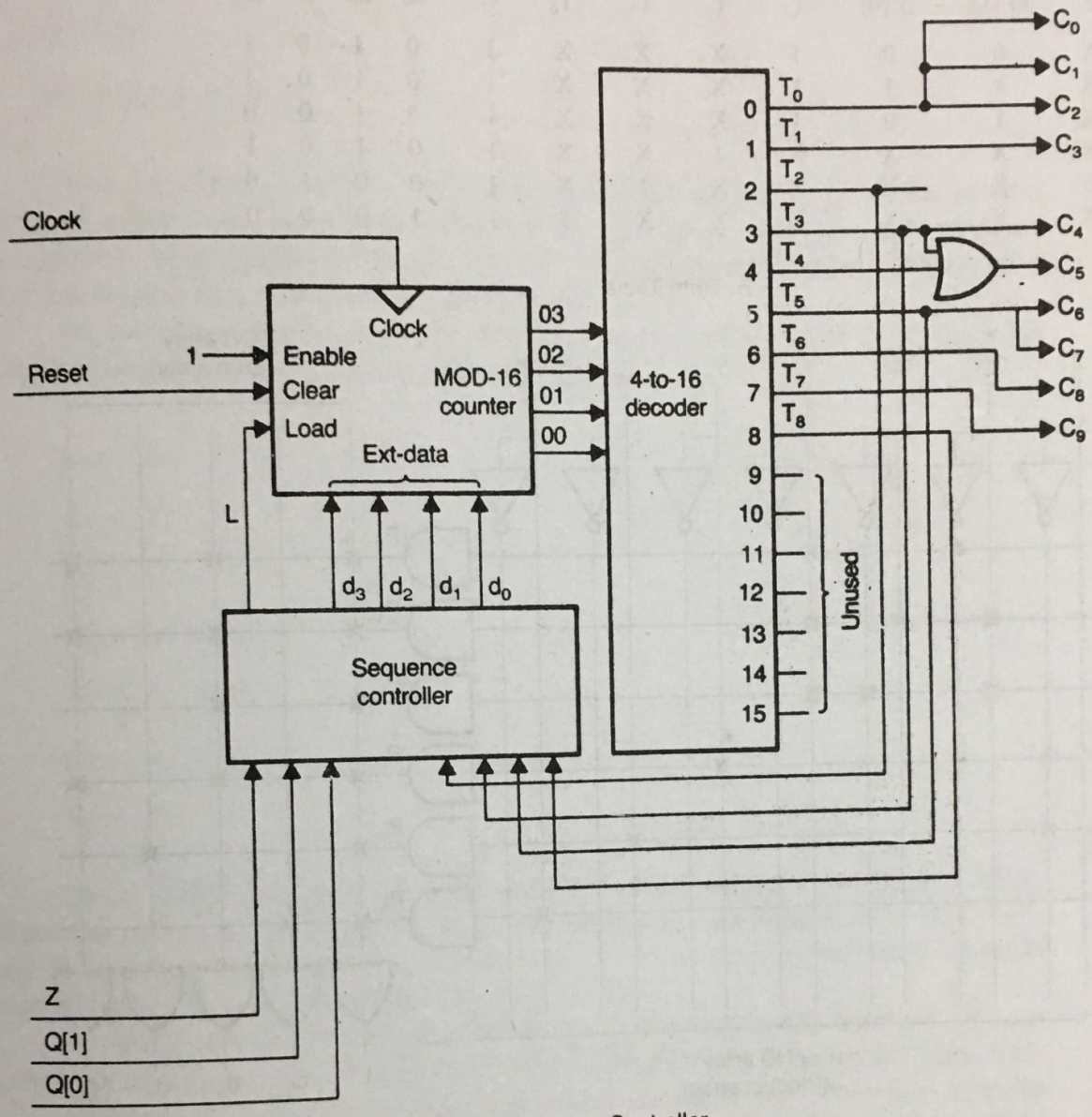


Figure 4.23 Logic Diagram of the Booth's Multiplier Controller

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

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

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

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

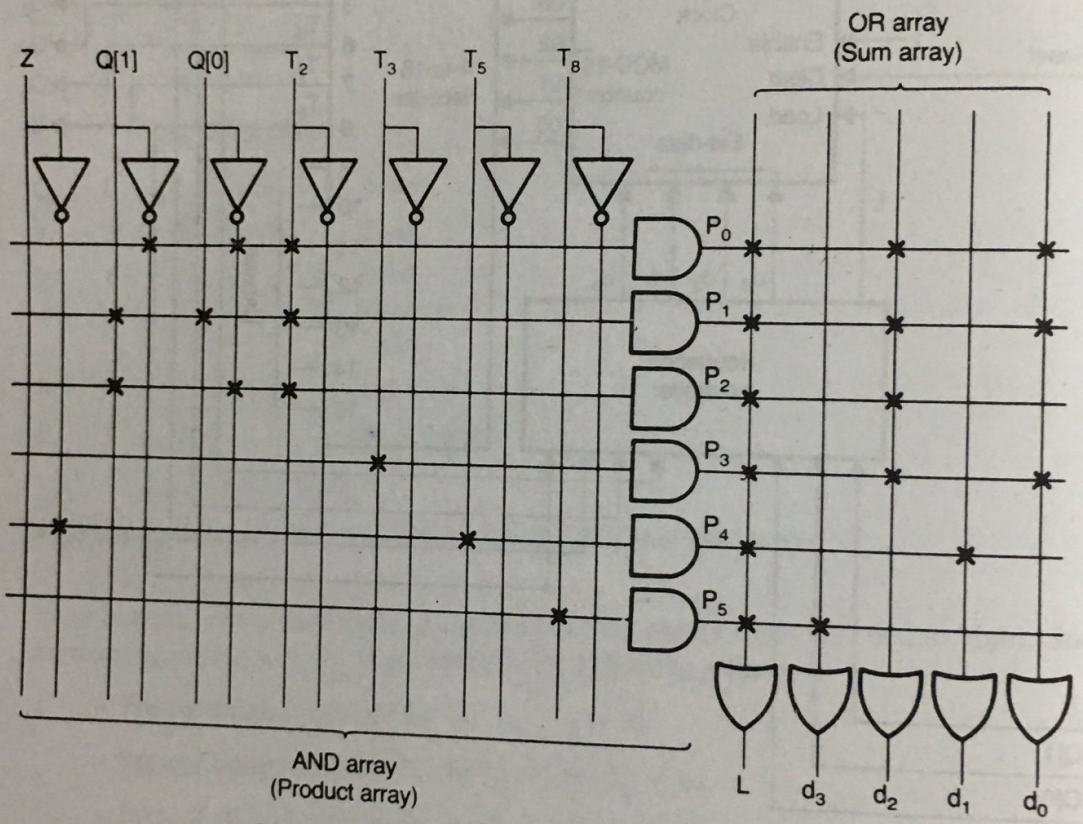
$$P_3 = T_3$$

$$P_4 = Z'T_5$$

$$P_5 = T_8$$

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

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: $A \leftarrow 0$, $L \leftarrow 4$, and $M \leftarrow$ Inbus, implying that $C_0 = C_1 = C_2 = T_0$. $A \leftarrow F$ 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:

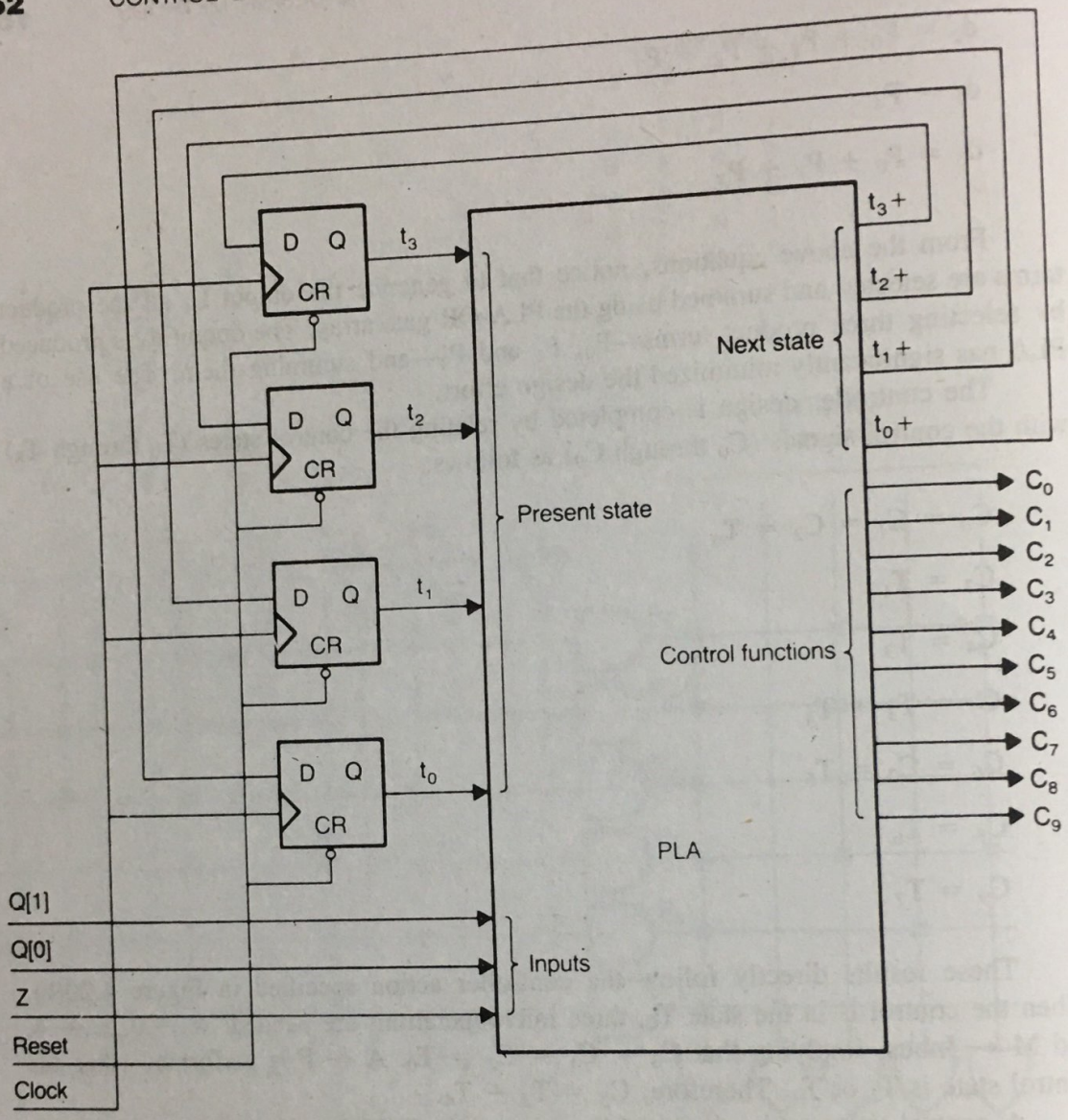


Figure 4.25 Organization of the PLA Control Unit for the Booth's Multiplier

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.

Present state is	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 4.27 PLA Contents

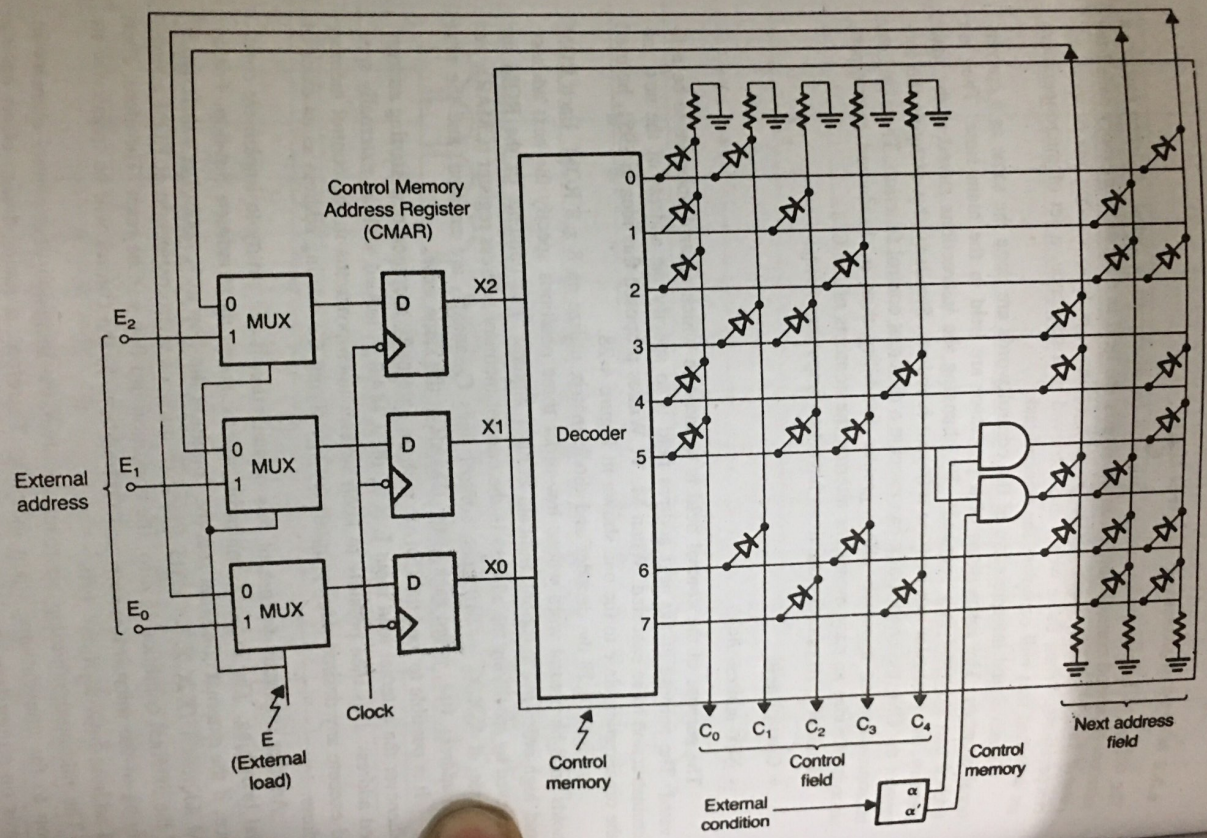


Figure 4.28 Wilke's Design

- 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:

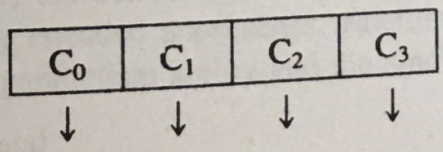
C_0: Outbus = A

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*.

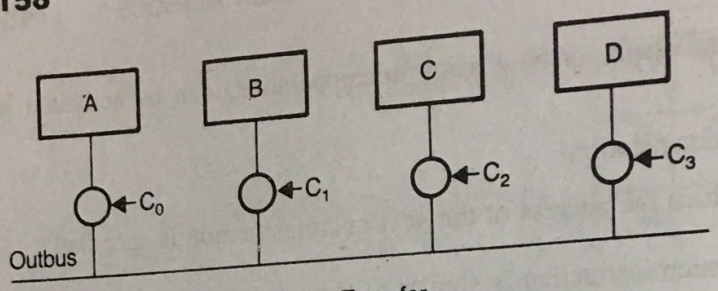


Figure 4.29 A Simple Register Transfer

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:

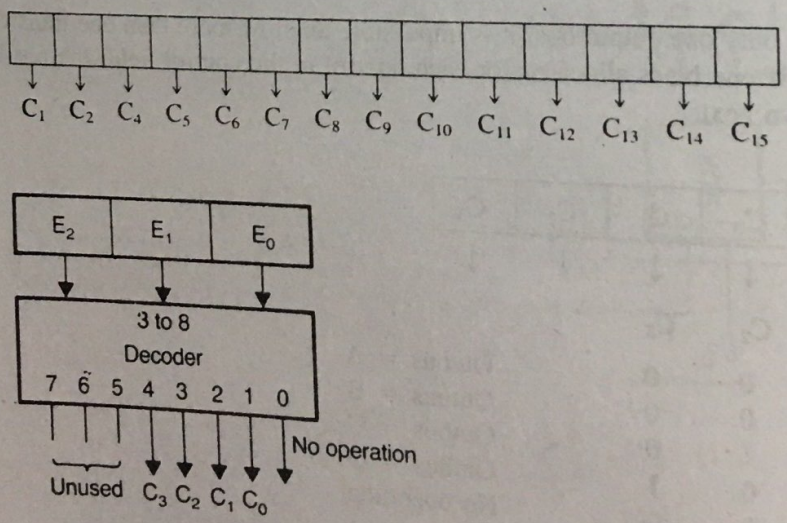
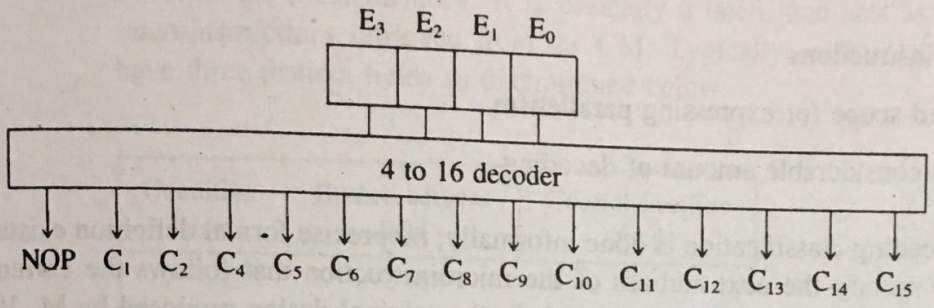
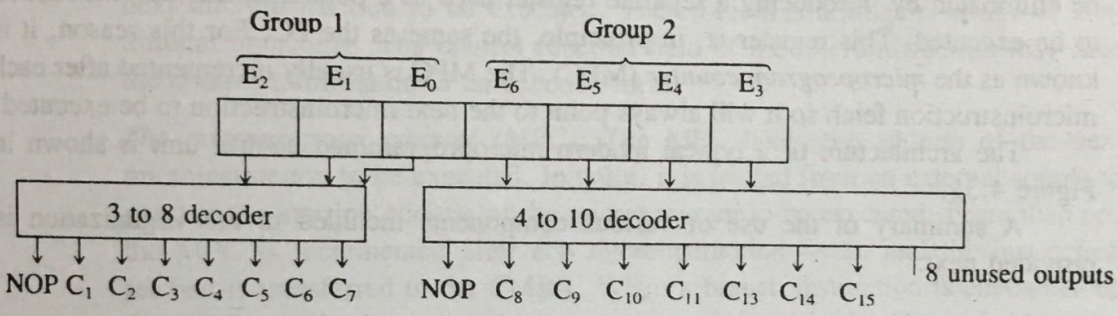


Figure 4.30 Encoded Control Arrangement

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:

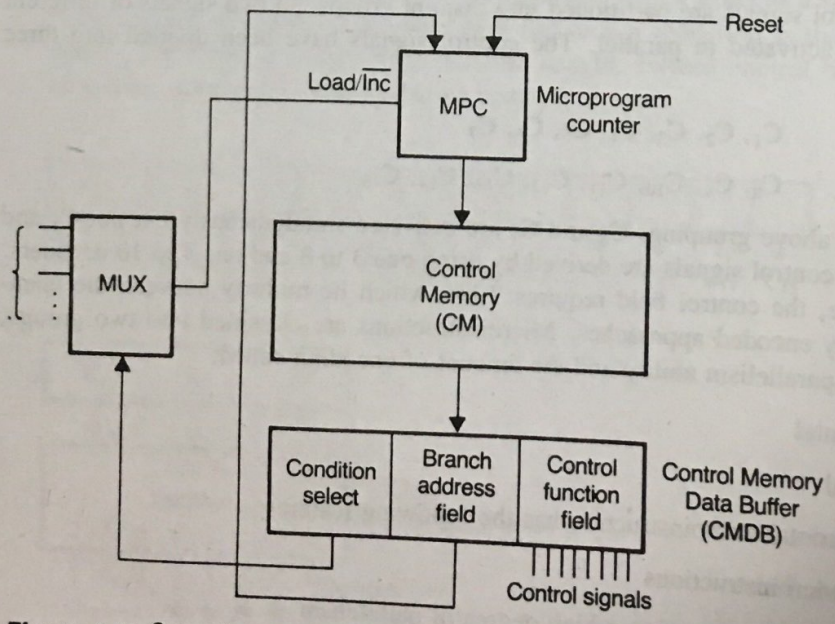


Figure 4.31 General-purpose Microprogrammed Control Organization (From J. P. Hayes, *Computer Architecture and Organization*, McGraw-Hill, 1978, p. 280. Adapted by permission of McGraw-Hill)

- *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

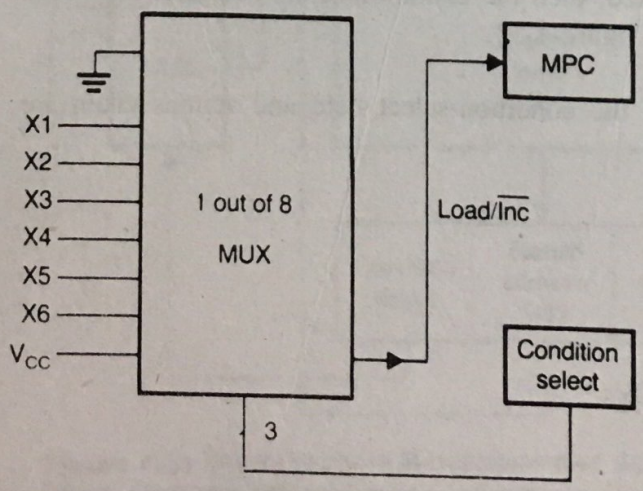


Figure 4.32 External Condition Select Multiplexer

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

General Format	Instruction Length in Bytes	Object Code		Instruction Type	Operation	Comment
		In binary	In hex			
LDA ⟨addr⟩	2	0000 1000 ⟨addr8⟩	08 ⟨addrH⟩	MRI	A ← M (⟨addr⟩)	Load accumulator direct
STA ⟨addr⟩	2	0000 1001 ⟨addr8⟩	09 ⟨addrH⟩	MRI	M (⟨addr⟩) ← A	Store accumulator direct
ADD ⟨addr⟩	2	0000 1010 ⟨addr8⟩	0A ⟨addrH⟩	MRI	A ← A + M (⟨addr⟩)	Add accumulator direct
SUB ⟨addr⟩	2	0000 1011 ⟨addr8⟩	0B ⟨addrH⟩	MRI	A ← A − M (⟨addr⟩)	Subtract accumulator direct
JZ ⟨addr⟩	2	0000 1100 ⟨addr8⟩	OC ⟨addrH⟩	MRI	If Z = 1 then PC ← ⟨addr⟩ else PC ← PC + 1	Jump on zero flag set
JC ⟨addr⟩	2	0000 1101 ⟨addr8⟩	0D ⟨addrH⟩	MRI	If C = 1 then PC ← ⟨addr⟩ else PC ← PC + 1	Jump on carry flag set
AND ⟨addr⟩	2	0000 1110	0E ⟨addrH⟩	MRI	A ← A ∧ M (⟨addr⟩)	And accumulator direct
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												
In decimal	In binary				Cond. Sel			Branch Addr.				
					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.

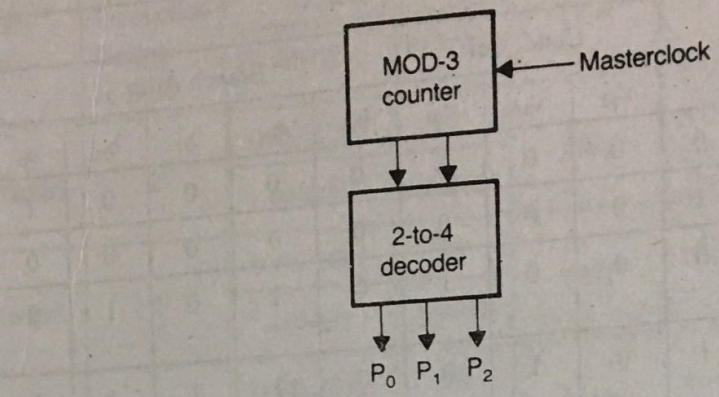
Next, the design of a microprogrammed processor is illustrated. The programming model of this processor is shown in Figure 4.37.

The CPU entails two registers:

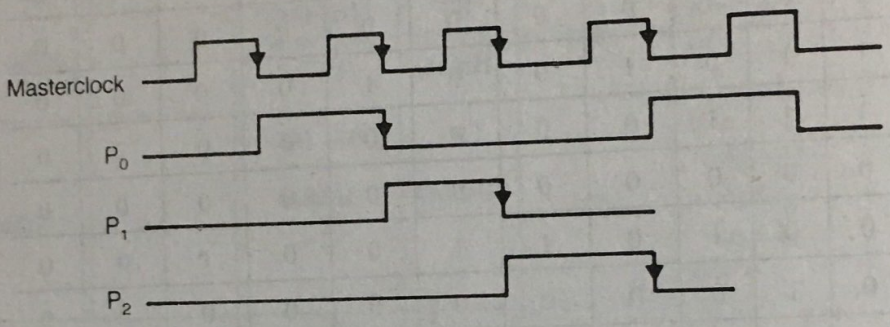
1. An 8-bit accumulator register A
2. A 2-bit flag register F

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.



a. Hardware



b. Timing Diagram

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.

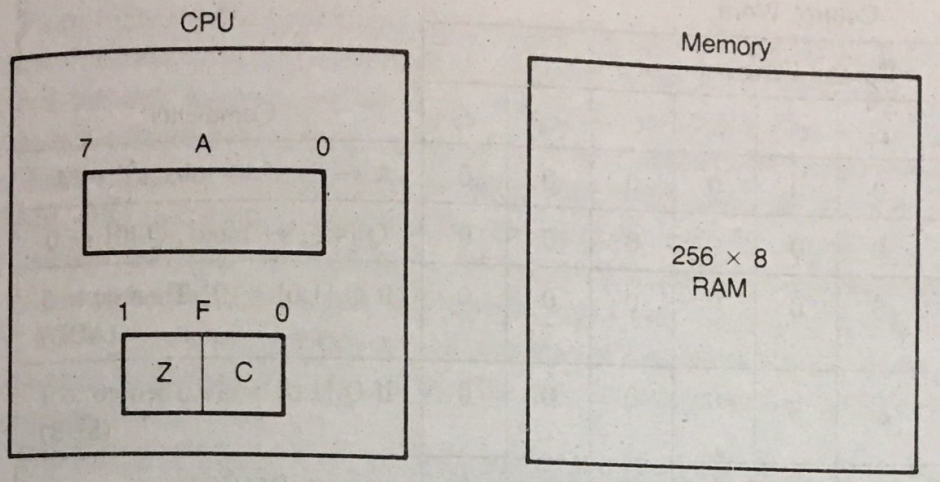


Figure 4.37 Programming Model of a Simple Processor

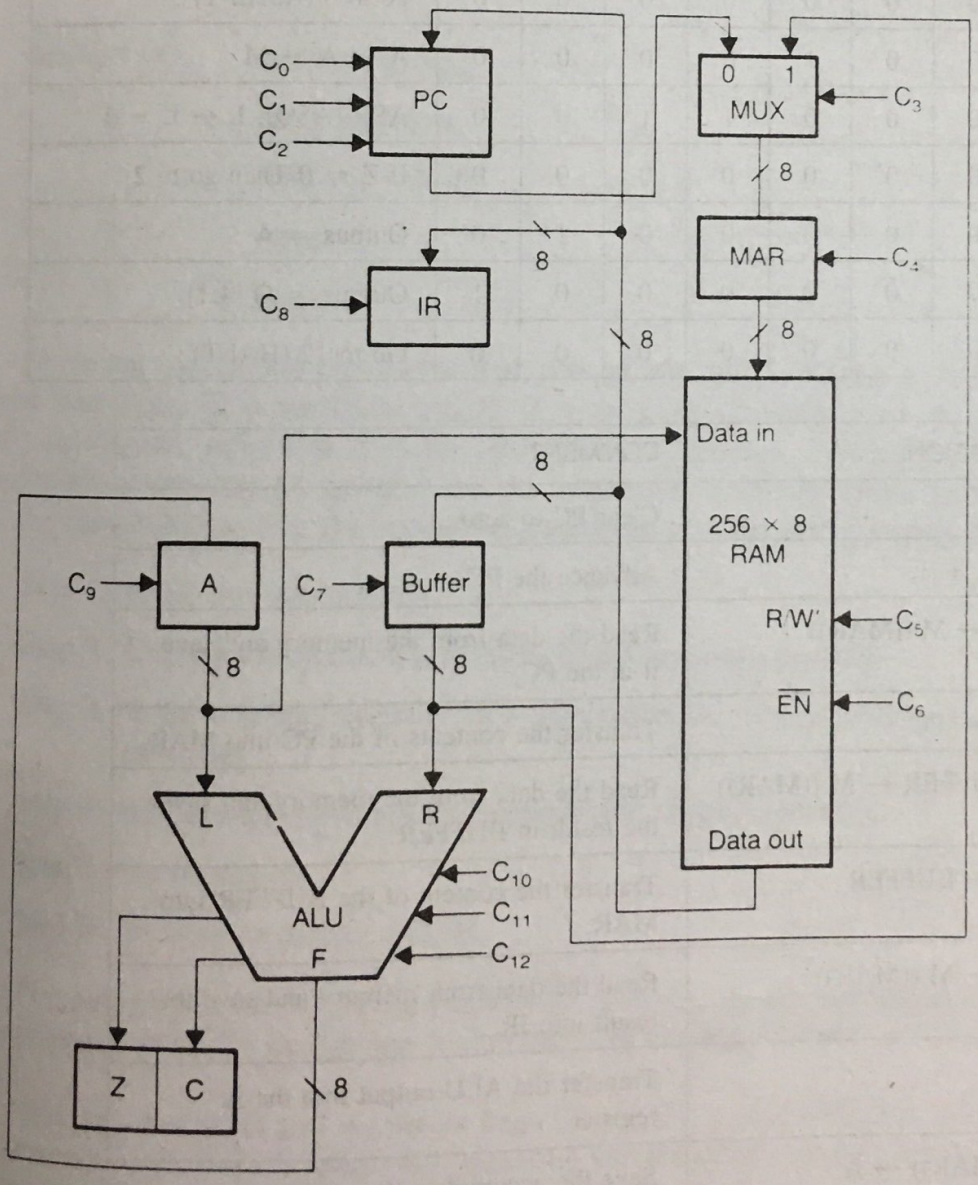


Figure 4.38 Hardware Schematic of the Simple Processor

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.
$C_5'C_6'$: 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.

Mnemonic	Op-code Bit of Their Interpretations							
					TC	GN		SC
	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
LDA	0	0	0	0	1	0	0	0
STA	0	0	0	0	1	0	0	1
ADD	0	0	0	0	1	0	1	0
SUB	0	0	0	0	1	0	1	1
JZ	0	0	0	0	1	1	0	0
JC	0	0	0	0	1	1	0	1
AND	0	0	0	0	1	1	1	0
CMA	0	0	0	0	0	0	0	0
INCA	0	0	0	0	0	0	1	0
DCRA	0	0	0	0	0	1	0	0
HLT	0	0	0	0	0	1	1	0

Note:

TC: Type classifier (if $I_3 = 1$, then it is a MRI; otherwise it is a NMRI)

GN: Group number within a type

(I_2 I_1	Group no.
0 0	0
0 1	1
1 0	2
1 1	3)

SC: Subcategory within a group

Figure 4.40 Op-code Encoding Logic

The hardware organization of the microprogrammed control unit for this situation shown in Figure 4.43 directly follows the symbolic listing shown in Figure 4.42. No attempt has been made toward arriving at a minimal microprogram. Rather, the concept was presented. The task of translating the symbolic microprogram of Figure 4.42 into a binary microprogram is left as an exercise.

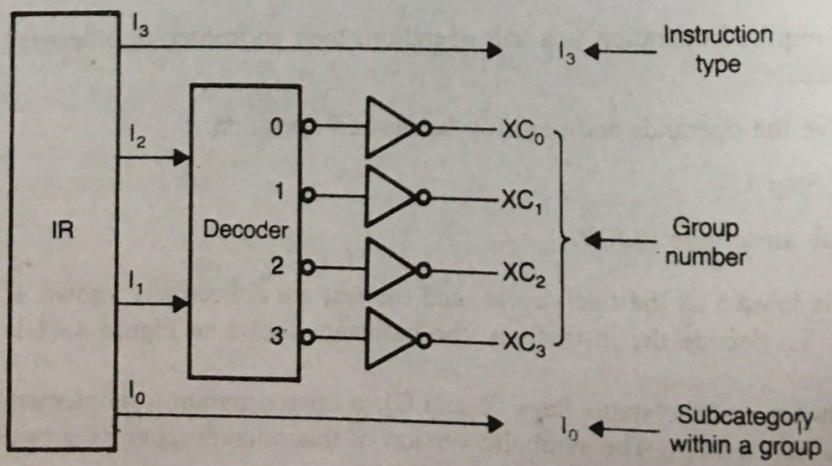


Figure 4.41 Instruction-decoding Hardware

Symbolic Microprogram:

ROM Address			
0		PC ← 0;	These operations
1	FETCH	MAR ← PC;	constitute the
2		IR ← M ((MAR)), PC ← PC + 1;	fetch cycle.
3		IF I_3 = 1 then go to MEMREF;	
4		IF XC0 = 1 then go to CMA;	Here we decode the
5		IF XC1 = 1 then go to INCA;	instructions.
6		IF XC2 = 1 then go to DCRA;	
7		Go to HALT;	
8	CMA	ACC ← ACC′	Execute CMA instructions.
9		Go to FETCH;	
10	INCA	ACC ← ACC + 1	Execute INCA instruction.
11		Go to FETCH;	
12	DCRA	ACC ← ACC − 1; then go to LDSTO;	Execute DCRA instruction.
13		Go to FETCH;	
14	MEMREF	IF XC0 = 1 then go to LDSTO;	Here we branch to
15		IF XC1 = 1 then go to ADSUB;	the various groups of
16		IF XC2 = 1 then go to JMPS;	the memory reference instruction.
17	AND	MAR ← PC;	
18		BUFFER ← M ((MAR)), PC ← PC + 1;	Execute AND instruction.
19		MAR ← BUFFER	
20		BUFFER ← M ((MAR));	
21		ACC ← ACC ∧ BUFFER;	
22		Go to FETCH;	
23	LDSTO	MAR ← PC;	
24		BUFFER ← M ((MAR)), PC ← PC + 1;	
25		MAR ← BUFFER;	
26		IF I0 = 1 then go to STO	
27	LOAD	BUFFER ← M ((MAR));	
28		ACC ← BUFFER	
29		Go to FETCH:	
30	STO	M ((MAR)) ← ACC;	
31		Go to FETCH;	

Figure 4.42 Symbolic Microprogram that Implements the Instruction Set Shown in Figure 4.39 on the Hardware Shown in Figure 4.38 *(Continues)*

```
32  ADSUB   MAR ← PC;
33          BUFFER ← M ((MAR)), PC ← PC + 1
34          MAR ← BUFFER
35          BUFFER ← M ((MAR));
36          IF I0 = 1 then go to SUB;                  Execute ADD instruction
37  ADD     ACC ← ACC + BUFFER;
38          Go to FETCH;                               Execute SUB instruction
39  SUB     ACC ← ACC − BUFFER;
40          Go to FETCH;
41  JMPS    MAR ← PC;
42
43          IF I0 = 1 then go to JOC;
44  JOZ     IF Z = 1 then go to LOADPC;                Execute JZ instruction
45          Go to FETCH;
46  JOC     IF C = 1 then go to LOADPC;                Execute JC instruction
47          Go to FETCH:
48          LOADPC                                     PC ← M((MAR))
49          Go to FETCH;
50  HALT    Go to HALT;                                Execute HALT instruction
```

Figure 4.42 *Continued*

The microprogramming approach is systematic, flexible, and less error-prone, as is evident from these examples. Advances in IC technology have made LSI designers think of a general solution for implementing a microprogrammed CPU. A microprogrammed CPU has two major activities to be performed:

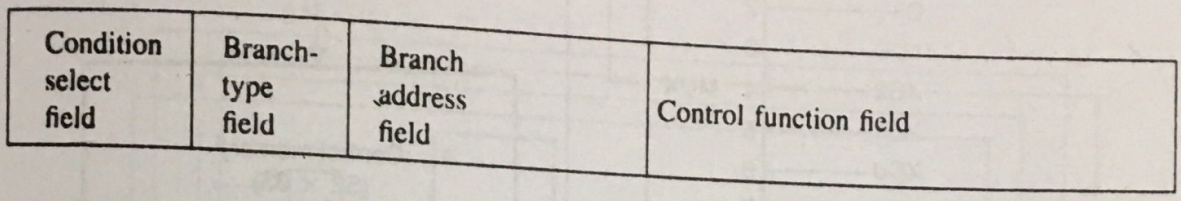
1. Fetching and interpreting microinstructions.
2. Generating the next address of the microinstruction to be retrieved.

The first task is assumed by the control memory and the associated circuit elements.

Designers have replaced the next address generation of a microprogrammed control unit with a single LSI component called a *microprogram sequencer*, which checks certain bits in the microinstruction and finds the next address for the control memory. The sequencer contains a microprogrammed counter (MPC) and circuit elements necessary to perform functions such as address incrementing, address sequencing for subroutine calls, returns, and conditional branching. At the present time, many microprogrammed control units manufactured use the following components:

- A control memory (ROM or RAM)
- A microprogram sequencer

The general organization of a microprogrammed control unit constructed using a microprogram sequencer is shown in Figure 4.44. The microinstruction in this figure is assumed to have the following format:



An additional field called *branch type field* (BT) has been included. The BT generates the next address, a 3-bit field (B2 B1 B0). Its interpretation is summarized as follows:

B2 B1 B0

000 No branching and, therefore, the next address is MPC + 1.

001 Branch to the address specified in the branch address field if the condition selected by the condition select field is true. The MPC is loaded with the contents of the branch address field.

010 Branch to the subroutine if the condition is met. Here, the return address is saved in a subroutine-return address register (SRAR). MPC is loaded with the subroutine's address, or MPC is loaded with MPC + 1. The single register SRAR is replaced with a set of registers to form a stack, if it is desired to implement nested or recursive subroutine calls.

011 Unconditional branch to subroutine loads the MPC in the subroutine's address, and the return address is saved in SRAR.

100 Unconditional branch loads the MPC with the contents of the branch address field.

101 Conditional return from subroutine loads the MPC with SRAR, if the condition is met; otherwise, it is loaded with MPC + 1.

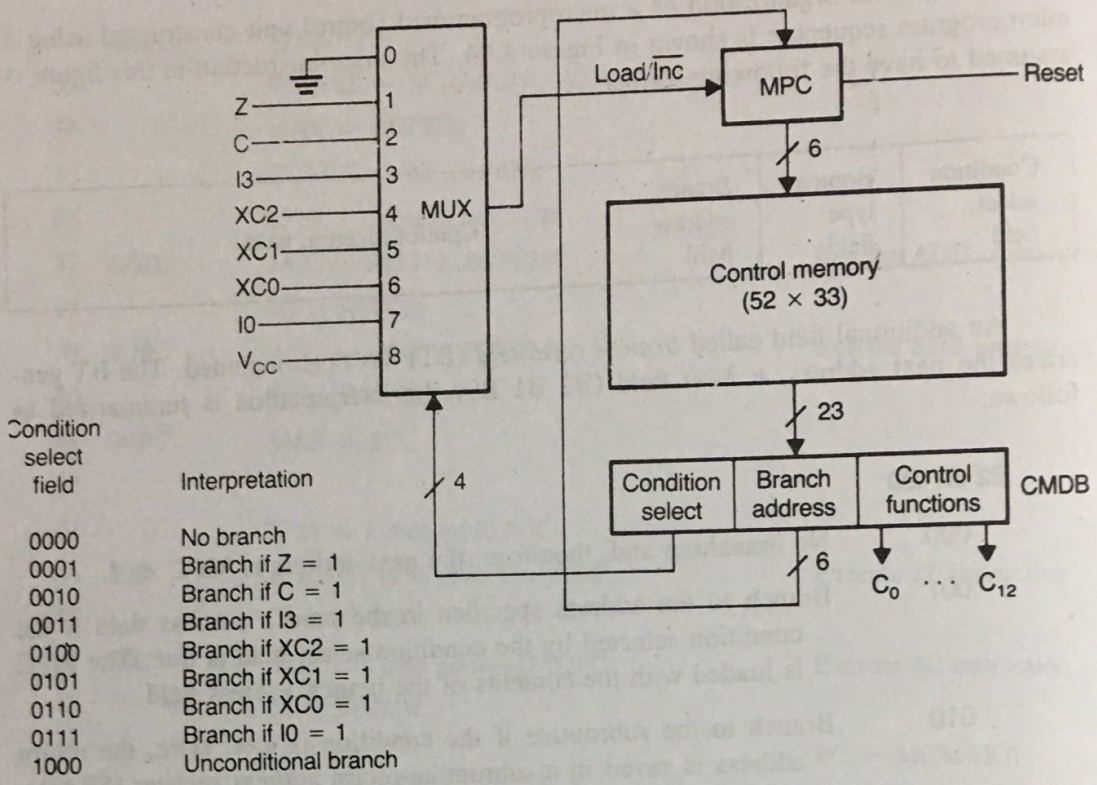
110 Unconditional return from subroutine. Always loads the MPC with the return address saved in SRAR.

The MPC is loaded with one of these sources:

- MPC + 1 computed by the parallel adder
- Branch address field A_1A_0 of the microinstruction
- SRAR

As shown from the branch-type field B2 B1 B0 listed before, the SRAR is loaded only when

$$B_2' B_1 B_0' Z_1 + B_2' B_1 B_0 \text{ is } 1$$



Condition select field	Interpretation
0000	No branch
0001	Branch if Z = 1
0010	Branch if C = 1
0011	Branch if I3 = 1
0100	Branch if XC2 = 1
0101	Branch if XC1 = 1
0110	Branch if XC0 = 1
0111	Branch if I0 = 1
1000	Unconditional branch

Figure 4.43 Microprogrammed Controller for the CPU

This Boolean equation is true when there is an unconditional branch to the subroutine or when a branch to a subroutine with the selected condition is true.

In the preceding equation, Z_1 refers to the output of the MUX1. This equation can also be simplified as follows:

$$B_2' B_1 B_0' Z_1 + B_2' B_1 B_0$$

$$= B_2' B_1 (B_0' Z_1 + B_0)$$

$$= B_2 B_1 [(B_0 + B_0') (B_0 + Z_1)] \quad \text{Since } yz + x = (x + y)(x + Z)$$

$$= B_2' B_1 (B_0 + Z_1) \quad \text{Since } B_0 + B_0' = 1$$

This hardware implementation has a 2-bit microprogram counter and three multiplexers, MUX1, MUX2, and MUX3. The MUX1 selects the desired condition, and multiplexers MUX2 and MUX3 select the low- and high-order address bits, respectively. First, the control word is fetched to the CMDB. Depending on the value of the BT, the next address is computed and loaded into the MPC. Consider some specific numeric examples:

IF $B_2B_1B_0 = 000$, line 0 of MUX2 and MUX3 are selected. Since these lines are outputs of the 2-bit parallel adder, outputs of the multiplexer MUX2 and MUX3 transfer the value MPC. + 1 into the MPC.

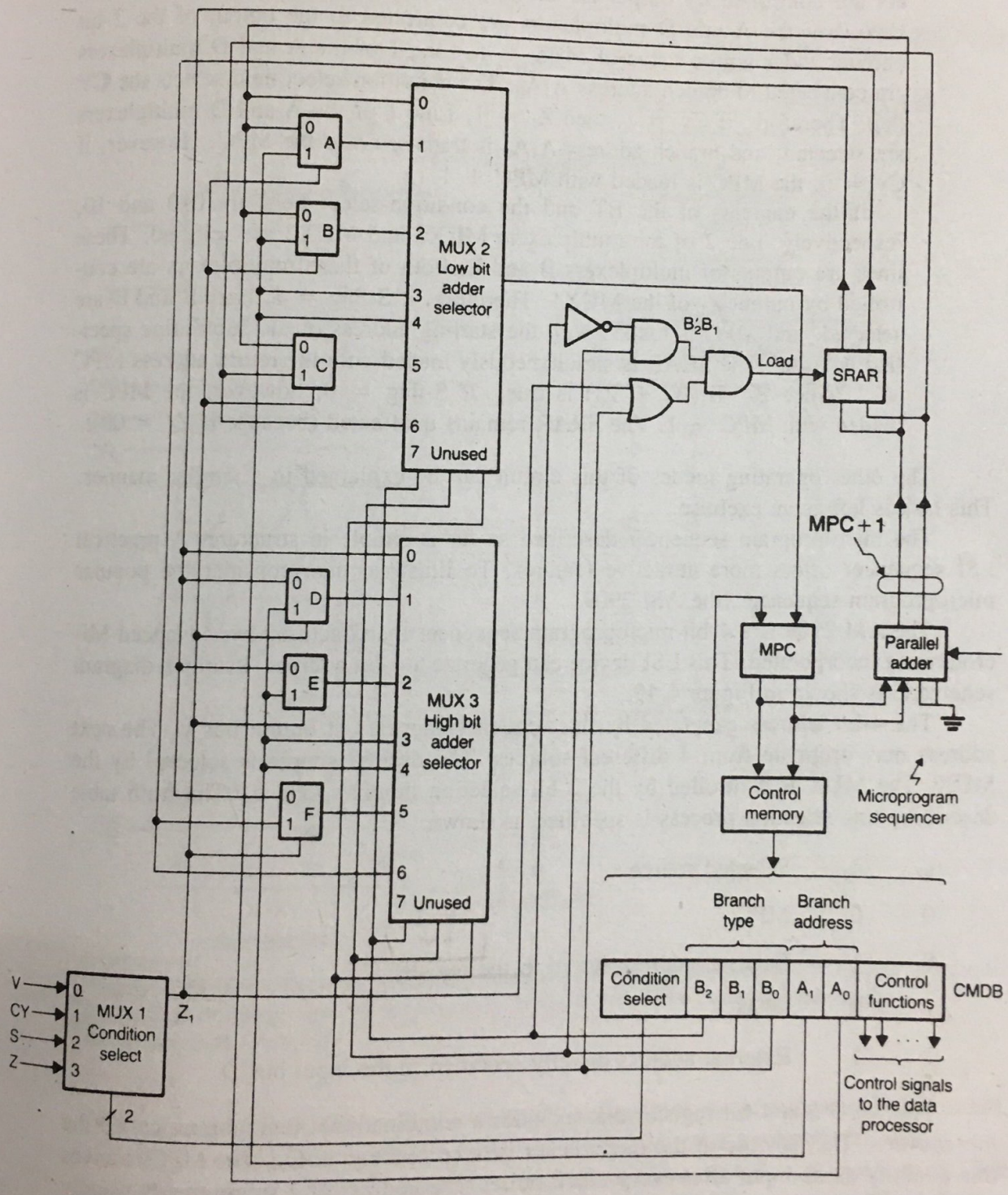


Figure 4.44 Structure of a Microprogrammed CPU Using Microprogram Sequencer

Assume $B_2B_1B_0 = 001$ and the contents of the condition select field are 01. The binary pattern of the BT selects line 1 of the multiplexers MUX2 and MUX3. These lines are outputs of multiplexers A and D. These two multiplexers are controlled by output Z1 of the condition-select multiplexer (MUX1). Line 0 of the A and D multiplexers are connected to the output of the 2-bit parallel adder with a value of MPC + 1. Line 1 of the A and D multiplexers are connected to branch address A_1A_0. The condition select field selects the CY flag. Therefore, if Cy = 1, then $Z_1 = 1$. Line 1 of the A and D multiplexers are selected, and branch address A_1A_0 is transferred to the MPC. However, if Cy = 0, the MPC is loaded with MPC + 1.

If the contents of the BT and the condition-select field are 010 and 10, respectively, line 2 of the multiplexers MUX2 and MUX3 are selected. These lines are outputs of multiplexers B and D; both of these multiplexers are controlled by output Z_1 of the MUX1. Therefore, if S-flag = 1, lines B and D are selected, and MPC is loaded with the starting address of the subroutine specified in A_1A_0. The SRAR is simultaneously loaded with the return address MPC + 1 (since $B_2' B_1(B_0 + Z_1)$ is true). If S-flag = 0, however, the MPC is loaded with MPC + 1. The SRAR remains unaffected (because $B_0Z_1 = 00$).

The other operating modes of this circuit can be explained in a similar manner. This task is left as an exercise.

The microprogram sequencer described so far is simple in structure. A practical LSI sequencer offers more attractive features. To illustrate this, consider the popular microprogram sequencer, the AM 2909.

The AM 2909 is a 4-bit microprogram sequencer manufactured by Advanced Microdevices Incorporated. This LSI device can generate a 4-bit address. Its block diagram schematic is shown in Figure 4.45.

The 4-bit address generated by this sequencer appears at output bus y. The next address may originate from 4 different sources. The desired source is selected by the MUX. The MUX is controlled by the 2-bit selection inputs s_1 and s_0. The truth table describing this selection process is specified as shown:

s_1	s_0	Selected source
0	0	MPC
0	1	External address stored in the register R
1	0	STACK
1	1	External address directly specified in the input bus D

The MPC is a 4-bit register that includes a combinational logic circuit called the *incrementer*. The purpose of the incrementer is to compute $y + C_{in}$. The MPC receives this quantity as its input after every clock pulse. The address of a subroutine is usually specified in the external 4-bit register R or on the 4-bit external input bus D. The STACK contains a 4 x 4 register file and an up/down counter (used as the SP). The purpose of the stack is to implement subroutine calls. Typically, when $\overline{SE} = 0$ and $\overline{POP}$/PUSH = 1, the contents of the MPC are pushed into the stack. Similarly, when

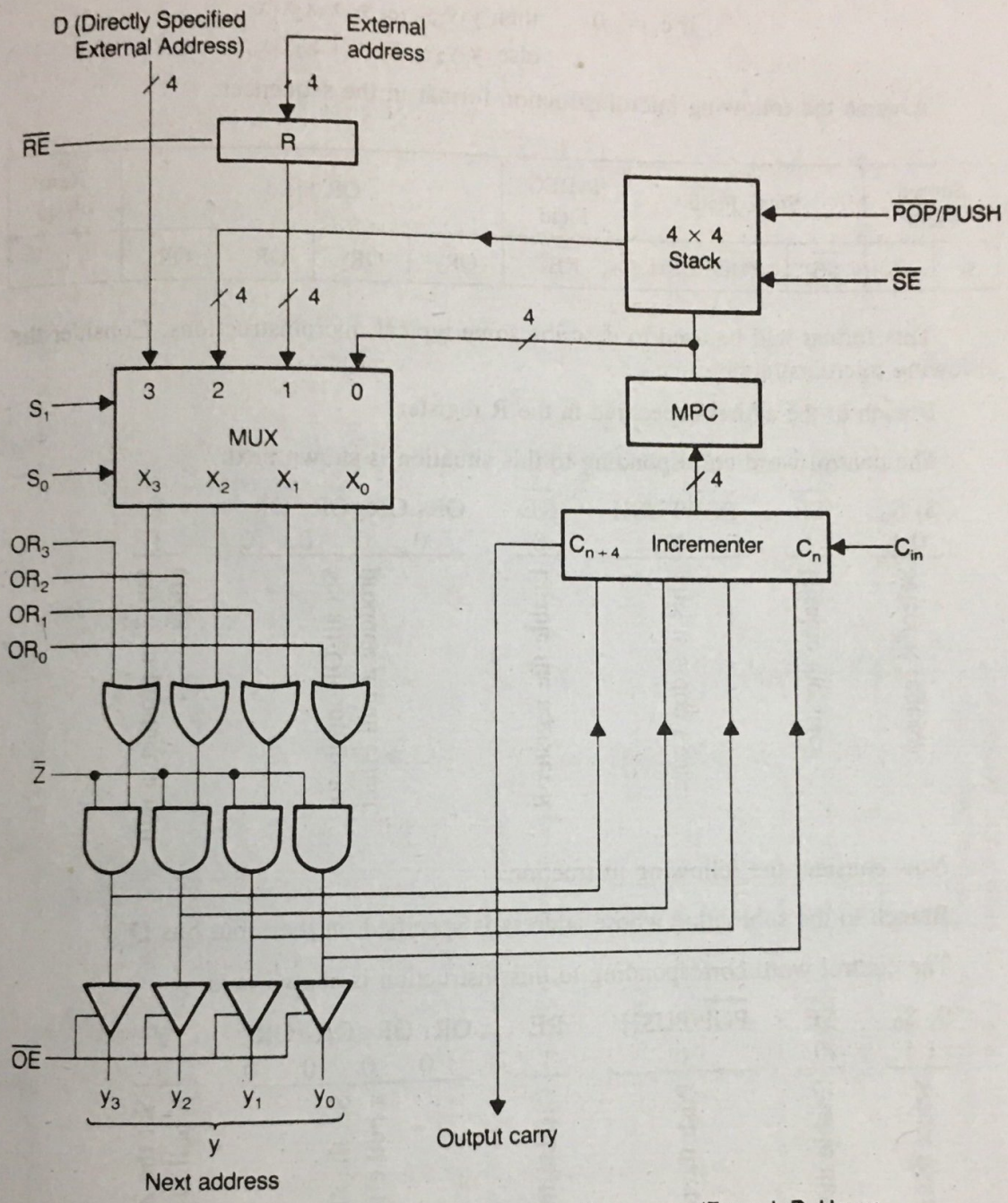


Figure 4.45 Block diagram of Am 2909 Microprogram Sequencer (From J. P. Hayes, *Digital System Design and Microcomputers*, McGraw-Hill, 1984, p. 709. Reprinted by permission of McGraw-Hill)

$\overline{SE} = 0$ and $\overline{POP}/PUSH = 0$, the contents of the stack word appear at MUX input 2. The MUX control inputs s_1 and s_0 are part of the computer's microinstruction.

The purpose of the $\overline{Z}$ input is to implement the external reset mechanism. When an external circuit drives this input low, the generated address is a 0. The four inputs (OR_3 through OR_0) along with the four 2-inputs are used to implement multiway conditional branching. Assume $OR_2 = OR_1 = OR_0 = 0$ and $OR_3 = c$ is some external condition. Two-way branching is formed as shown next:

if c = 0 then $y_3y_2y_1y_0 = x_3x_2x_1x_0$
else $y_3y_2y_1y_0 = 1\ x_2x_1x_0$

Assume the following microinstruction format in the sequencer:

Source Field		Stack Field		R-REG Field	OR Field				Zero Field
S_1	S_0	$\overline{SE}$	$\overline{POP/PUSH}$	$\overline{RE}$	OR_3	OR_2	OR_1	OR_0	Z

This format will be used to describe some typical microinstructions. Consider the following microinstruction:

Branch to the address specified in the R register

The control word corresponding to this situation is shown next:

S_1 S_0	$\overline{SE}$	$\overline{POP/PUSH}$	$\overline{RE}$	OR_3 OR_2 OR_1 OR_0	$\overline{Z}$
0 1	1	X	0	0 0 0 0	1
Select R register.	Disable the stack.	This is a don't care.	Enable the register R.	Set all OR-inputs to produce a null effect.	Set $\overline{Z}$ to produce a null effect.

Now consider the following instruction:

Branch to the subroutine whose address is specified on the input bus D

The control word corresponding to this instruction is shown next:

S_1 S_0	$\overline{SE}$	$\overline{POP/PUSH}$	RE	OR_3 OR_2 OR_1 OR_0	$\overline{Z}$
1 1	0	1	1	0 0 0 0	1
Select the input bus.	Enable the stack.	Push the return address.	Disable the register R.	Set the OR inputs to produce a null effect.	Set the $\overline{Z}$ input to produce a null effect.

Figure 4.46 shows the interconnection of three AM 2909s to form a 12-bit sequencer, which will expand the sequencer. The only interconnection between packages other than the control lines is the ripple carry between the MPC and the incrementers.

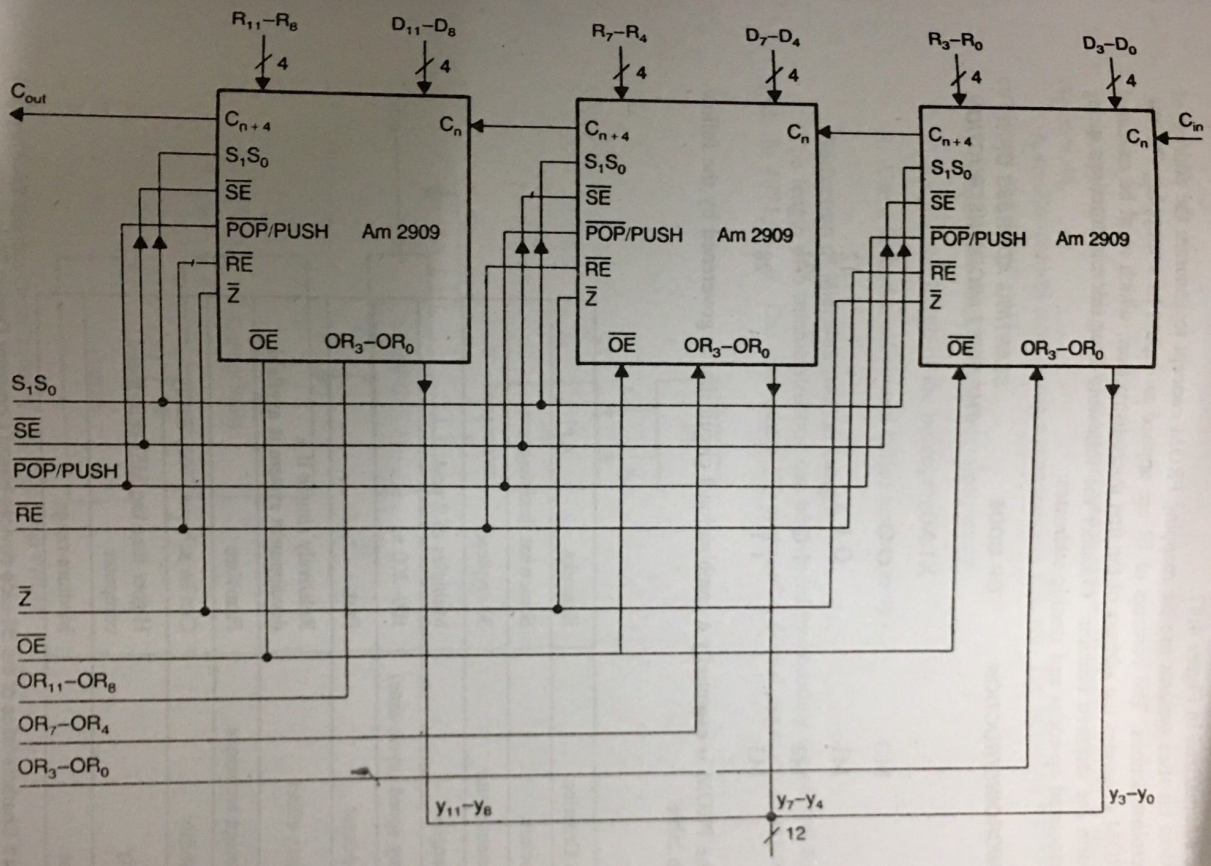


Figure 4.46 A 12-bit Microprogram Sequencer (From J. P. Hayes, *Digital System Design and Microprocessors*, McGraw-Hill, 1984, p. 711. Reprinted by permission of McGraw-Hill)

A complete computer system can be designed using bit-slice devices such as the AM 2901 and AM 2909. Bit-slice components offer flexibility for a microprogrammed implementation. The salient characteristics of bit-slice-device-based computer philosophy are summarized in Figure 4.47.

Most bit-slice systems use the mapping PROM concept to generate the address of the microinstructions. The contents of IR are applied as inputs to a mapping PROM. The PROM generates the address of the first microinstruction, which must be executed to perform the required function. Consider the following four microinstructions along with associated op-codes and starting addresses:

MICROINSTRUCTION	*OP-CODE*	*STARTING ADDRESS OF THE FIRST MICROINSTRUCTION*
	X1 X0	
MO	O O	8
M1	O 1	12
M2	1 O	24
M3	1 1	28

The PROM is essentially a combinational circuit that is goverened by the following truth table:

Design Criterion	Remarks
Architecture	Somewhat flexible
Component count	50 (typical)
Word length	Multiples of 2 or 4
Operating speed (cycle time)	100–200 ns
Design pocess	Fast
Debugging effort	Relatively simple if a development system is available
Major design technique	Firmware
Expandability	Can be achieved very easily
Reliability	Higher than the MSI-chip-based computer
Total cost	Medium range

Figure 4.47 Characteristics of the Bit-slice-devices-based Computer Design Philosophy

X1	X0	MPC4	MPC3	MPC2	MPC1	MPC0
0	0	0	1	0	0	0
0	1	0	1	1	0	0
1	0	1	1	0	0	0
1	1	1	1	1	0	0

A typical block diagram of a system and a 256 × 40 control PROM is shown in Figure 4.48.

A smaller control store is more economically efficient. The following methods are employed to optimize size:

1. Reduce the length of the microinstruction.
2. Reduce the length of the microprogram.
3. Use a second-level control called *nanomemory*.

Reduction of Microinstruction Length

The length of a microinstruction can be reduced by encoding the control function field. In 1971, S. R. Das and others [5] proposed a formal approach to encode the

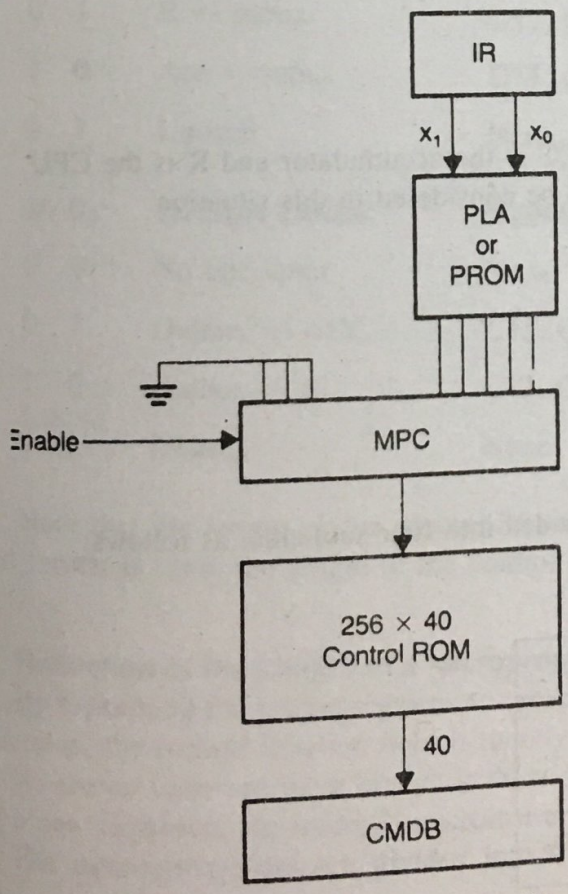


Figure 4.48 Control Organization Using a Mapping PROM

control function field so the storage requirements are minimized. The primary advantage of this method is that is can be computerized. Nevertheless, this algorithm arbitrarily groups the control functions to be encoded, which increases the effort required to write microprograms.

To overcome this difficulty, a new approach is presented to encode the control functions in a logical manner. The control functions confined to a particular functional unit are first grouped together. They are then encoded. This approach is referred to as the *encoding by function* technique. To illustrate this idea, the following example is provided. Consider the following control functions:

$$C0: ACC \leftarrow ACC + R$$

$$C1 \wedge C2: ACC \leftarrow ACC - R$$

$$C3 \wedge C4: ACC \leftarrow 0$$

$$C5: R \leftarrow 0$$

$$C6 \wedge C7 \wedge C8: R \leftarrow R + 1$$

$$C9 \wedge C10: R \leftarrow R - 1$$

$$C11 \wedge C12: R \leftarrow INBUS$$

$$C13 \wedge C14: ACC \leftarrow INBUS$$

$$C15 \wedge C16: Outbus \leftarrow ACC$$

$$C17 \wedge C18: Outbus \leftarrow R$$

In the above description, assume that ACC is the accumulator and R is the CPU register. The following functional units need to be considered in this situation:

- Accumulator unit
- Count unit (register R)
- Input bus
- Output bus

Therefore, the control function field is divided into four subfields, as follows:

Control Function Field							
ACC Field		Count Field		Input Bus Field		Output Bus Field	
A_1	A_0	B_1	B_0	I_1	I_0	D_1	D_0

Each subfield is encoded as shown next:

A_1 A_0	ACTION TAKEN	CONTROL SIGNAL ACTIVATED
0 0	No operation	None
0 1	ACC ← ACC + R	C0
1 0	ACC ← ACC − R	C1, C2
1 1	ACC ← 0	C3, C4

B_1 B_0	ACTION TAKEN	CONTROL SIGNAL ACTIVATED
0 0	No operation	None
0 1	R ← 0	C5
1 0	R ← R + 1	C6, C7, C8
1 1	R ← R − 1	C9, C10

I_1 I_0	ACTION TAKEN	CONTROL SIGNAL ACTIVATED
0 0	No operation	None
0 1	R ← inbus	C11, C12
1 0	Acc ← inbus	C13, C14
1 1	Unused	None

D_1 D_0	ACTION TAKEN	CONTROL SIGNAL ACTIVATED
0 0	No operation	None
0 1	Outbus ← ACC	C15, C16
1 0	Outbus ← R	C17, C18
1 1	Unused	None

Note that the length of the control function field is only 8 bits. If a fully unencoded format is used, the length of the control function field is 19 bits.

Reduction of the Length of a Microprogram

By examining the microprograms so far, it can be seen that in the event of branch instructions, the control function field is merely filled with 0s.

Whenever there are more branch instructions, the microinstruction is formatted by using a new approach: the multiple microinstruction format.

The microinstructions are divided into two groups: operate and branch instructions.

An operate instruction initiates one or more microoperations. For example, after the execution of an operate instruction, the MPC will be incremented by 1. In the case of a branch instruction, no microoperation will usually be initiated, and the MPC may be loaded with a new value.

This suggests the removal of the branch address field from the microinstruction format, so the control function field is used to specify the branch address itself. Typically, each microinstruction will have two fields, as shown next:

CONDITION-SELECT FIELD		CONTROL FUNCTION FIELD					
S_1	S_0	C_5	C_4	C_3	C_2	C_1	C_0

If S_1 S_0 = 00, the microinstruction is considered as an operate instruction, and the contents of the control function field are treated as the control signals. Assume the Condition Select Field is encoded as follows:

S_1 S_0	
0 0	No branch
0 1	Branch if cond-1 = 1
1 0	Branch if cond-2 = 1
1 1	Unconditional branch

If S_1 S_0 = 01, the instruction is regarded as a branch instruction, and the contents of the control field are assumed to be a 6-bit branch address. When S_1 S_0 = 01, the MPC will be loaded with (C_5 C_4 C_3 C_2 C_1 C_0) if Cond-1 is met.

To illustrate this, Booth's multiplier example is reconsidered. The binary microprogram will be rewritten using the multiple microinstruction instruction format, as shown in Figure 4.49.

From Figure 4.49 it can be seen that the total size of the control store is 169 bits (13 × 13 = 169). For the first two microinstructions, the contents of the condition select field are 0 and are considered as operation instructions. In this case, the contents of the control function field are directed to the processing hardware.

The third microinstruction is a branch instruction, since the contents of the condition select field are not 0s. Here, the 13-bit control function field directly specifies the desired branch address ($00000\ 0000\ 0101_2 = 5_{10}$).

The hardware organization that implements the microcode of Figure 4.49 is shown in Figure 4.50.

Nanomemory

Nanomemory is another approach for reducing the size of the control memory. This technique involves a two-level memory: control memory and nanomemory. At the

ROM address	Control Word										Comments	
	Condition Selection Field	Control Function/Branch Address Field									Instruction Type	Operation Performed
		c_9	c_8	c_7	c_6	c_5	c_4	c_3 / b_{r3}	c_2 / b_{r2}	c_1 / b_{r1}	c_0 / b_{r0}	
0000	000	0	0	0	0	0	0	0	1	1	1	Operate	A ← 0, M ← Inbus L ← 4
0001	000	0	0	0	0	0	0	1	0	0	0	Operate	Q [4:1] ← Inbus, Q [0] ← 0
0010	001	0	0	0	0	0	0	0	1	0	1	Branch	If Q [1:0] = 0 Then go to 5 (ADD)
0011	010	0	0	0	0	0	0	0	1	1	1	Branch	If Q [1:0] = 10 Then go to 7 (SUB)
0100	100	0	0	0	0	0	0	1	0	0	0	Branch	Go to 8 (RSHIFT)
0101	000	0	0	0	0	1	1	0	0	0	0	Operate	A ← A + M
0110	100	0	0	0	0	0	0	1	0	0	0	Branch	Go to 8 (RSHIFT)
0111	000	0	0	0	0	1	0	0	0	0	0	Operate	A ← A − M
1000	000	0	0	1	1	0	0	0	0	0	0	Operate	ASR (A$Q), L ← L − 1
1001	011	0	0	0	0	0	0	0	0	1	0	Branch	If Z = 0 Then go to 2 (LOOP)
1010	000	0	1	0	0	0	0	0	0	0	0	Operate	Outbus = A
1011	000	1	0	0	0	0	0	0	0	0	0	Operate	Outbus = Q [4:1]
1100	100	0	0	0	0	0	0	1	1	0	0	Branch	Go to 12 (HALT)

Figure 4.49 Binary Listing of the 4 x 4 Booth's Multiplication Microprogram (in Multiple Microinstruction Format)

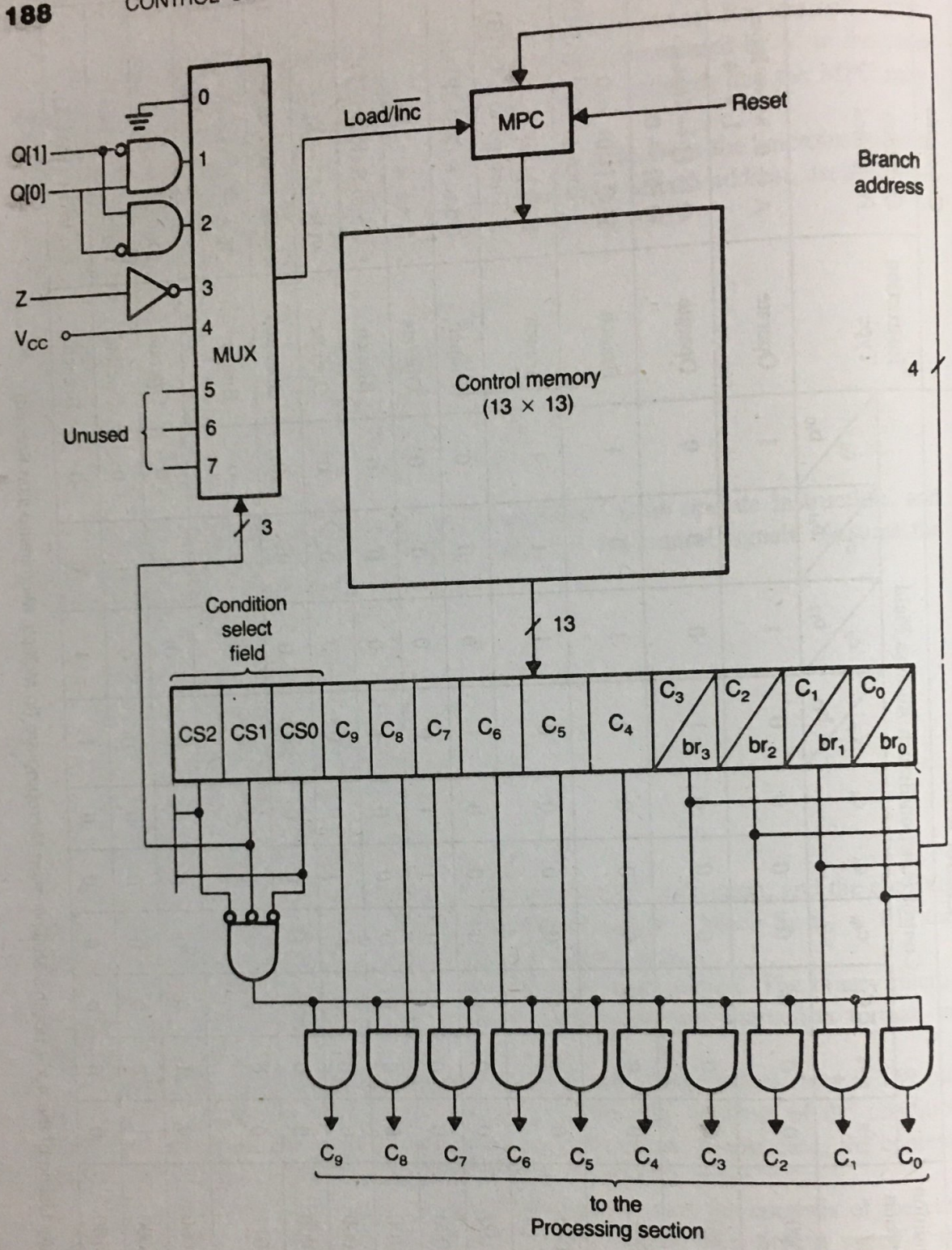


Figure 4.50 Organization of th Microprogrammed Controller for Implementing the Microcode Shown in Figure 5.49

outset, one may feel that two-level structure will increase the overall cost, but it actually improves the economy of the system by reducing the total space required.

The concept of nanomemory is derived from a combination of horizontal and vertical microinstructions. This method, in fact, provides useful trade-offs between these two disciplines. Nanomemory offers significant savings in space when a group of microoperations occur many times in a microprogram.

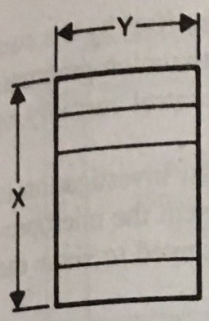


Figure 4.51 A Microprogram of Size $X \times Y$

Consider the microprogram of Figure 4.51. The microprogram in this figure consists of X microinstructions Y bits wide.

The size of the control memory to store this microprogram is XY bits. Assume this microprogram has p ($p < X$) unique microinstructions. These p microinstructions can be held in a separate memory called the *nanomemory* of size pY bits. These p instructions occur once with the nanomemory. Each microinstruction in the original microprogram is replaced with an address that specifies the location of the nanomemory in which the original Y-bit-wide microinstruction is held. Since the nanomemory has p addresses, only $\lceil \log_2 p \rceil$ (rounded up to the next integer) bits are needed to specify one nanomemory address. Therefore, the size of each microinstruction in a two-level control store is only $\lceil \log_2 p \rceil$ bits. This is illustrated in Figure 4.52.

The operation of a control unit employing a two-level store can be explained as follows. The first word from the microprogram is read. This word is actually the first address of the nanomemory. The contents of this location in the nanomemory is the control word, which is transferred to the CMDB. The bits in this register are then used to control the appropriate gates for one cycle. Upon completion of the cycle, the second word is read from the control memory and the process continues. Note that a control unit using a nanomemory control store is slower than one using only a conventional control memory, since the nanomemory concept consists of two-level memory. Therefore, two memory fetches (one for the control memory and the other for the nanomemory) are required. The conventional control memory consists of a single memory; thus,

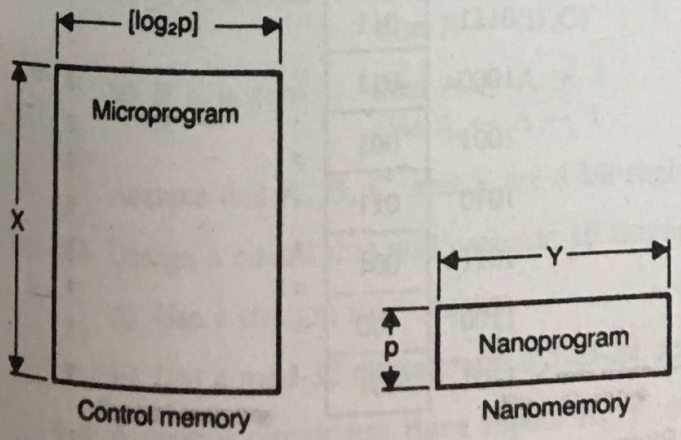


Figure 4.52 Nanomemory Concept

one memory fetch is necessary. This reduction in control unit speed is offset by the cost of the memory when the same microinstructions occur many times in the microprogram.

Consider the 14 × 8 bit microprogram stored in the single control memory of Figure 4.53.

This microprogram is implemented using a nanomemory, and an investigation is then carried out to determine which technique is appropriate to implement the microprogram. The program may not serve any practical purpose, but it is presented to show the rationale behind the nanomemory concept.

In this program, 5 out of 14 microinstructions are unique. Therefore, the size of the microcontrol store is $14 \times \lceil \log_2 5 \rceil$, or 14×3, bits, and the size of the nanomemory is 5×8 bits. This is illustrated in Figure 4.54.

memory requirements for single-control store $= 14 \times 8 = 112$ bits

memory requirements for nanomemory $= (14 \times 3 + 5 \times 8)$ bits

$= 42 + 40 = 82$ bits

therefore, savings using nanomemory $= 112 - 82 = 30$ bits

Hence, the use of two-level control store may be recommended.

0000	0000 0010
0001	0011 0111
0010	0010 0100
0011	0100 1010
0100	0000 0011
0101	0000 0010
0110	0011 0111
0111	0100 1010
1000	0100 1010
1001	0011 0111
1010	0100 1010
1011	0011 0111
1100	0000 0011
1101	0011 0111
1110	.....
1111	.....

Figure 4.53 14 × 8-bit Single-control Store

0000	000
0001	001
0010	010
0011	011
0100	100
0101	000
0110	001
0111	011
1000	011
1001	001
1010	011
1011	001
1100	100
1101	001

000	0000 0010
001	0011 0111
010	0010 0100
011	0100 1010
100	0000 0011

5 × 8 nanocontrol store

Figure 4.54 14 × 3 Microcontrol Store

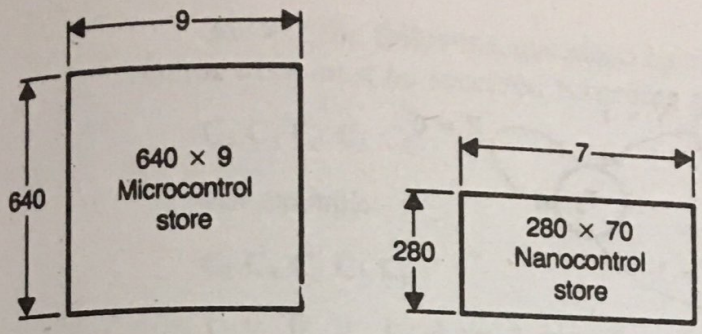


Figure 4.55 MC68000 Control Unit Structure

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 $A \leftarrow A - 1$

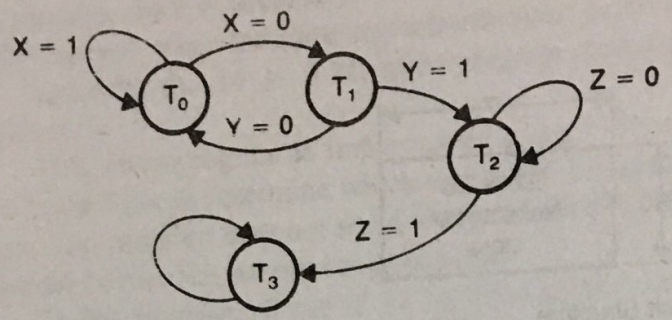
Assume that A, B, C, and X are 4-bit registers.

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.

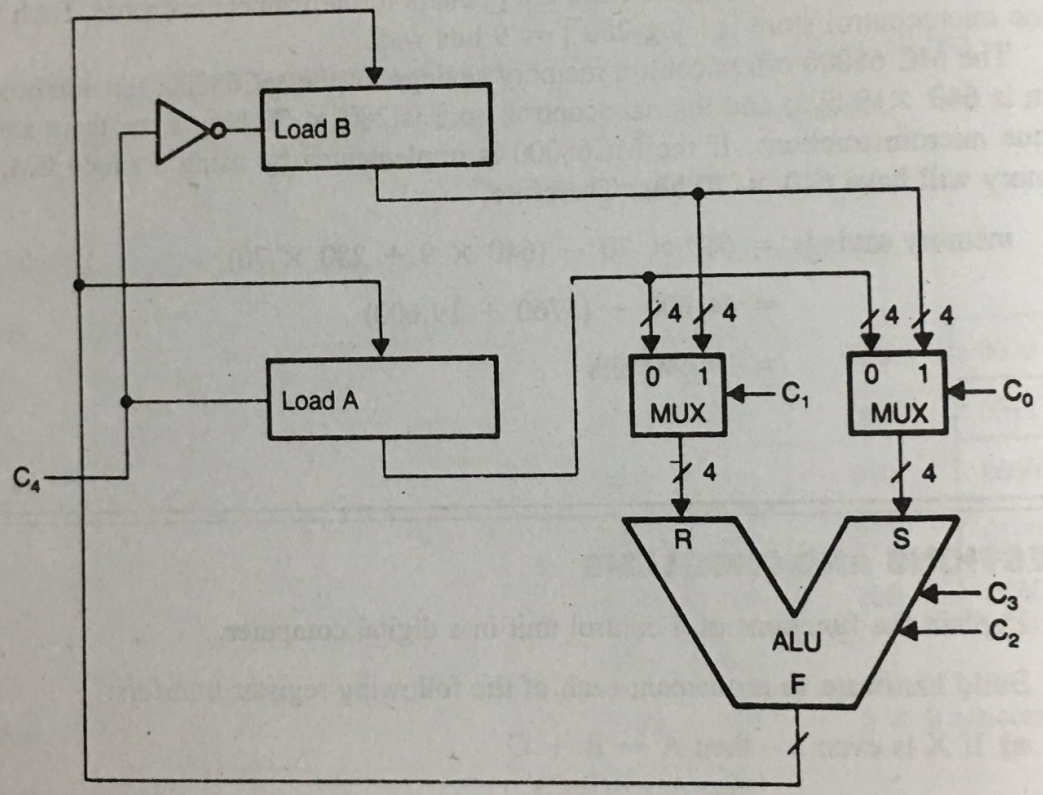
4.4 A control circuit has three inputs X, Y, and Z, and its state diagram is shown next:



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.

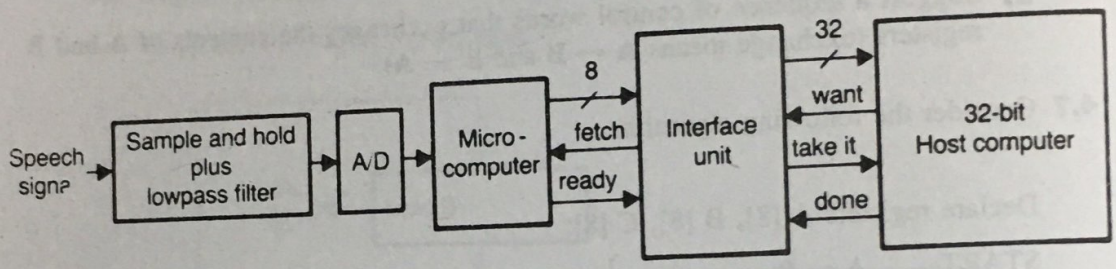
Step 8: The interface unit maintains a valid 32-bit data on the 32-bit output bus until the host processor says that it is done (the host puts a high signal on the line "done" for one clock period). In this case, the interface proceeds to step 1 and looks for a high on the "want" line.

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_ks) and the filter coefficients. This is formally expressed as follows:

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

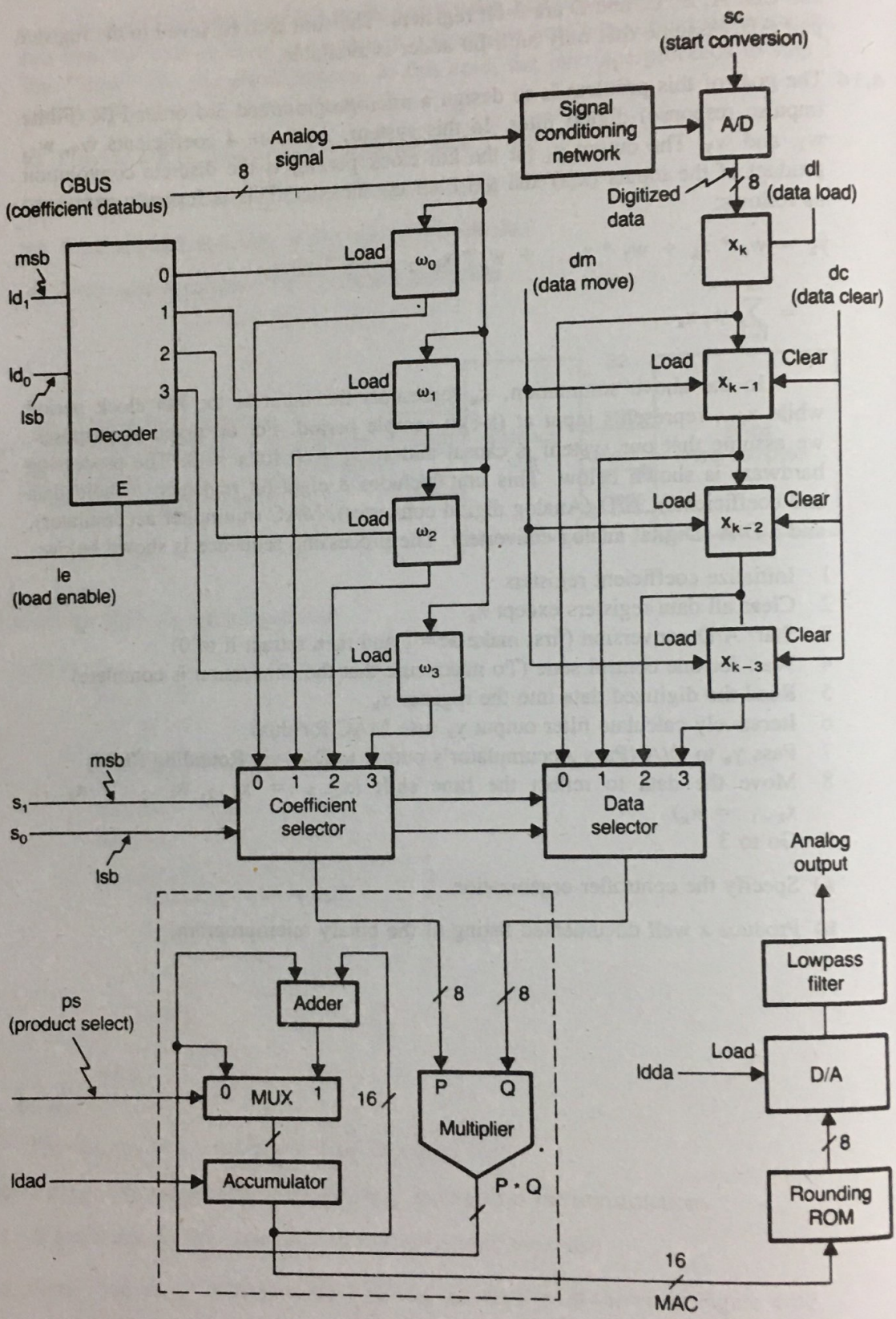
$$= \sum_{i=0}^{3} w_i \, x_{k-i}$$

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:

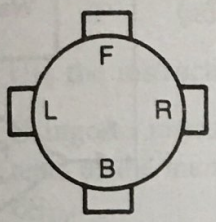
1 Initialize coefficient registers
2 Clear all data registers except x_k
3 Start A/D conversion (first make sc = 1 and then retract it to 0)
4 Wait for one control state (To make sure that the conversion is complete)
5 Read the digitized data into the register x_k
6 Iteratively calculate filter output 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



4.15 Your task is to design a microprogrammed controller for a simple robot with 4 sensors (see figure A). The sensor output will go high only if there is a wall or an obstruction within a certain distance. For example, if F = 1, there is an obstruction or wall in the forward direction. In particular, your controller is supposed to communicate with a motor controller unit shown in figure B. The flow chart that describes the control algorithm is shown in figure C. The outputs such as MFTS, MRT, MLT, MUT, and STP, andd the status signals such as FMC, and TC will be high for one clock period. Assume that a power on reset causes the controller to go the WAIT STATE 0.



F: forward direction sensor
R: right direction sensor
L: left direction sensor
B: backward direction sensor

Figure A

a) Specify the controller organization.

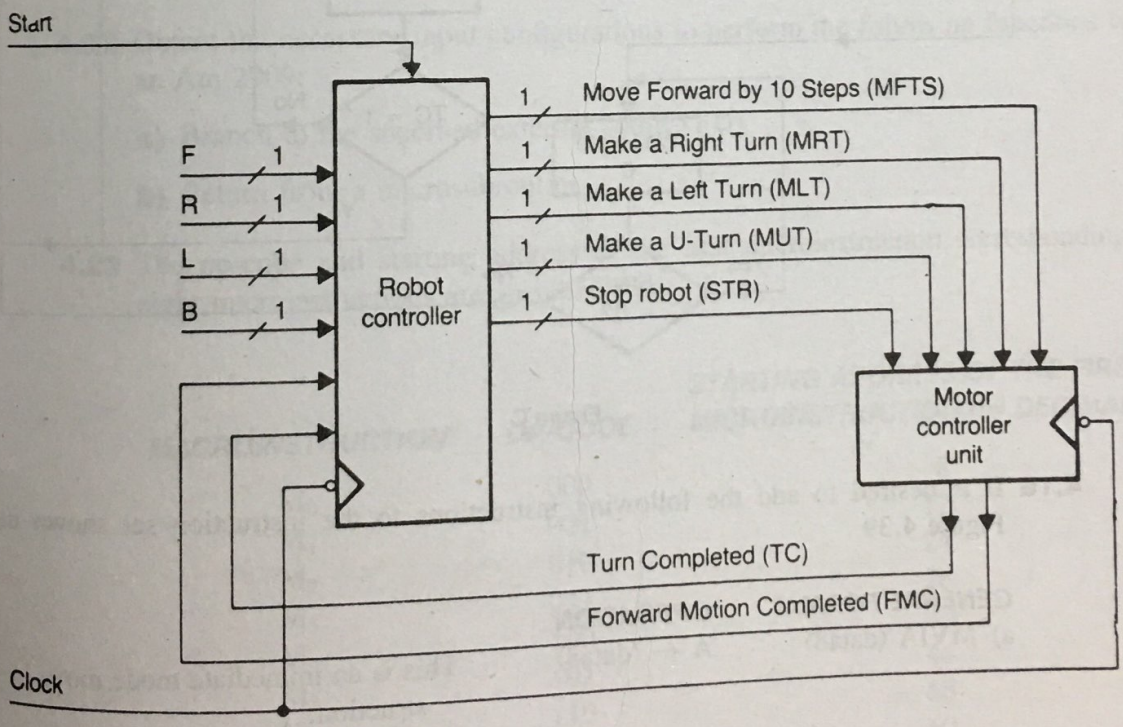


Figure B

b) Provide a well documented listing of the binary microprogram.

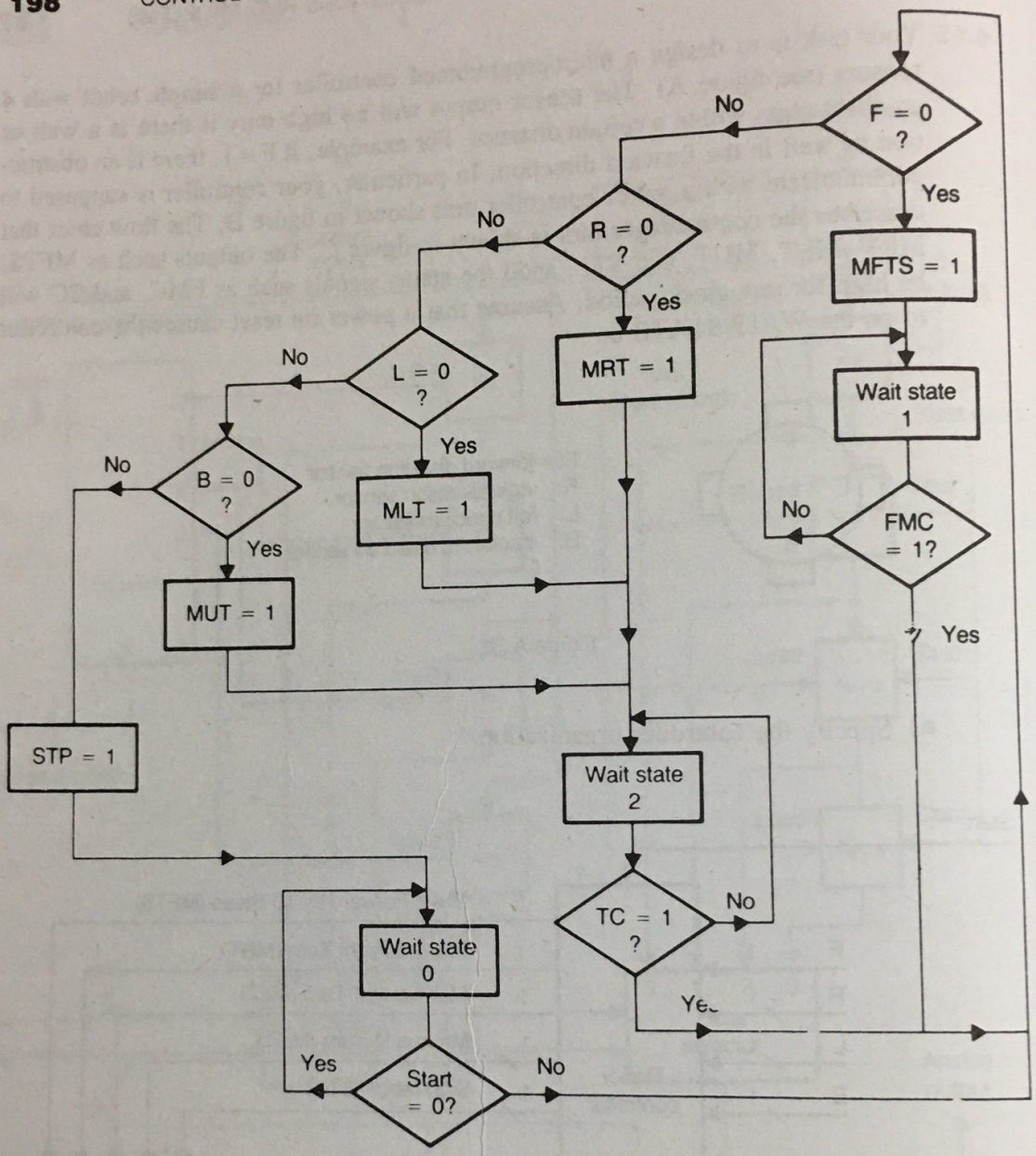


Figure C

4.16 It is desired to add the following instructions to the instruction set shown in Figure 4.39.

GENERAL FORMAT	***OPERATION***	
a) MVIA ⟨data8⟩	A ← ⟨data8⟩	This is an immediate mode move instruction.
		The first byte contains the op-code while the second byte contains the 8-bit data.

b) CLRA	$A \leftarrow 0$	This is a clear instruction.
c) PRSA	$A \leftarrow 11111111$	This is a preset instruction
d) NEGA	$A \leftarrow -A$	This instruction negates the contents of A

Write a symbolic microprogram that describes the execution of each instruction.

4.17 Explain how the effect of an unconditional branch instruction of the following form is simulated:

JP ⟨addr⟩

Use the instruction set shown in Figure 4.39.

4.18 Using the instruction set shown in Figure 4.39, write a program to add the contents of the memory locations 64_{16} through $6D_{16}$ and save the result in the address $6E_{16}$.

4.19 Mention four salient features of the Am 2909 microprogram sequencer.

4.20 Make a 16-bit microprogram sequencer using four AM 2909s.

4.21 What is the significance of the OR output in the Am 2909 microprogram sequencer?

4.22 Obtain the necessary input configurations to perform the following functions with an Am 2909:

a) Branch to the specified external address (D).

b) Return from a microsubroutine.

4.23 The op-code and starting address of the first microinstruction corresponding to eight microinstructions are specified next:

MACROINSTRUCTION	*OP-CODE*	*STARTING ADDRESS OF THE FIRST MICROINSTRUCTION (IN DECIMAL)*
M_0	000	8
M_1	001	12
M_2	010	24
M_3	011	28
M_4	100	40
M_5	101	44
M_6	110	56
M_7	111	60

a) Design a suitable mapping PROM that will achieve this mapping. Assume that the control memory has 4k words.

b) Is it possible to achieve the same result without using the mapping PROM? Assume that the control memory has 4K words.

c) Compare the advantages and disadvantages of the approaches used in parts (a) and (b) of this exercise.

4.24 What are the advantages and disadvantages of the microprogrammed approach over the hardwired approach?

4.25 What are the advantages and disadvantages of a two-level control structure?

4.26 A conventional microprogrammed control unit includes 2048 words by 117 bits. Each of 512 microinstructions is unique. Calculate the savings achieved by having a Nanomemory. Calculate the sizes of micro-control store and nanostore.

4.27 Repeat Question 4.8 using microprogramming.

4.28 Show that it is possible to specify 675 microoperations using a 10 bit control function field.

4.29 Consider the following 10 × 8 microprogram implemented in a conventional control memory:

0000	0001 0000
0001	0010 1001
0010	0001 1111
0011	0001 0000
0100	0001 1111
0101	0001 0000
0110	0001 1111
0111	0010 1001
1000	0010 1001
1001	0010 1001
1010	
1011	

Implement the above microprogram in nanomemory. Justify the use of either a single-control store or a two-level store for the program.

4.30 A microprogram occupies 100 words and each word typically emits 70 CONTROL SIGNALS. The architect claims that by using a $2^l \times 70$ nanomemory (for some $l > 0$), it is possible to save 4260 bits. IF this were true, DETERMINE THE NUMBER OF DISTINCT CONTROL STATES IN THE ORIGINAL MI-

CROPROGRAM (Note that here when we say a control state we refer only to the control function field).

HINT: You may have to employ a trial and error approach to solve this problem

REFERENCES

1. Advanced Micro Devices Inc. *The* AM 2900 *Family Data Book*. 1976.
2. Advanced Micro Devices Inc. *Microprogramming Handbook*. 1977.
3. Agarwala, A. K. and T. G. Rausher. *Foundations of Microprogramming: Architecture, Software, and Applications*. Orlando: Academic Press, 1976.
4. Chu, Y. *Computer Organization and Microprogramming*. Englewood Cliffs, N.J.: Prentice-Hall, 1972.
5. Das, S. R., D. K. Banerji, and A. Chattopadhyay. "On Control Memory Minimization in Microprogrammed Computers." *IEEE Transaction on Computers* C-23 (September 1973).
6. Frieder, G. and J. Miller. "An Analysis of Code Density for the Two-Level Programmable Control of the Nanodata QM-1": Tenth Annual Workshop on Microprogramming, October 1975, pp. 26–32.
7. Hamacher, V. C., Z. G. Vrasenic, and S. G. Taky. *Computer Organization*. New York: McGraw-Hill, 1978.
8. Hayes, J. P. *Computer Architecture and Organization*. New York: McGraw-Hill, 1978.
9. Husson, S. S. Microprogramming: Principles and Practice. Englewood Cliffs, N.J.: Prentice-Hall, 1970.
10. Katzan, H., Jr.. Microprogramming Primer. New York: McGraw-Hill, 1977.
11. Mano, M. M.. *Digital Logic and Computer Design*. Englewood Cliffs, N.J.: Prentice Hall, 1979.
12. ———. *Computer Systems Architecture*. Englewood Cliffs, N.J.: Prentice-Hall, 1982.
13. Mead, C., and L. Conway. *Introduction to VLSI Systems*. Reading, Mass.: Addison-Wesley, 1980.
14. Mick, J., J. Brick. *Bit-Slice Microprocessor Design*. New York: McGraw-Hill, 1980.
15. Rauscher, T G., and P. M. Adams. Microprogramming: A Tutorial and Survey of Recent Development. *IEEE Transaction on Computers* C-29, no. 1, (January 1980): 2–49.
16. Stritter, S., and N. Tredernick. "Microprogrammed Implementation of a Single Chip Microprocessor." *Sigmicro Newsletter* 9, no. 4 (December 1978).
17. Tanenbaum, A. S. *Structured Computer Organization*. Englewood Cliffs, N.J.: Prentice-Hall, 1981.

the first available

The tag register offers an additional ... determined by resetting the corresponding tag bit to 0. The status of the word can be determined by inspecting the tag bit. By examining the tag bit, the inactive words can be eliminated from the associative search

Associative memories are desirable in high-speed applications. This concept is used to speed up dynamic address-translation schemes. This is explained in a later section of this chapter

5.8 VIRTUAL MEMORY AND MEMORY-MANAGEMENT CONCEPTS

This section describes basic memory-management concepts. Topics include virtual memory, paging, segmentation, and address mapping schemes.

Virtual memory is the most fundamental concept implemented by a system that performs memory-management functions such as space allocation, program relocation, code sharing and protection.

The key idea behind this concept is to allow a user program to address more locations than those available in a physical memory. An address generated by a user program is called a *virtual address*. The set of virtual addresses constitutes the virtual address space. Similarly, the main memory of a computer contains a fixed number of addressable locations and a set of these locations forms the physical address space.

In the early days, when a programmer used to write a large program that could not fit into the main memory, it was necessary to divide the program into small portions so each one could fit into the primary memory. These small portions are called *overlays*. A programmer has to design overlays so they are independent of each other. Under these circumstances, one can successively bring each overlay into the main memory and execute them in a sequence. Although this idea appears to be simple, it increases the program-development time considerably.

However, in a system that uses a virtual memory, the size of the virtual address space is usually much larger than the available physical address space. In such a system, a programmer does not have to worry about overlay design, and thus a program can be written assuming a huge address space is available. In a virtual memory system, the programming effort can be greatly simplified. However, in reality, the actual number of physical addresses available is considerably less than the number of virtual addresses provided by the system. There should be some mechanism for dividing a large program into small overlays automatically. A virtual memory system is one that mechanizes the process of overlay generation by performing a series of mapping operations.

Sometimes the size of a virtual address space may be even less than the available physical address space. For example, in the PDP-11/45 minicomputer, the size of an address generated by a user program is 16 bits wide, allowing a user program to address only 64K different locations. However, the actual physical memory size can be up to

256K. To address 256K addresses, the length of the address field must be 18 bits. In this case, the mechanism that implements the virtual memory actually maps a 16-bit virtual address into an 18-bit physical address. This concept is known as *address extension*.

A virtual memory system may be configured in one of the following ways:

- Paging systems
- Segmentation systems

In a paging system, the virtual address space is divided into equal-size blocks called *pages*. Similarly, the physical memory is also divided into equal-size blocks called frames. The size of a page is the same as the size of a frame. The size of a page may be 512, 1024 or 2048 words.

In a paging system, each virtual address may be regarded as an ordered pair $\langle p, n \rangle$, where p is the page number and n is the word number within the page p. Sometimes the quantity n is referred to as the *displacement*, or *offset*. A user program may be regarded as a sequence of pages, and a complete copy of the program is always held in a backup store such as a drum or a disk. A page p of the user program can be placed in any available page frame p' of the main memory. A program may access a page if the page is in the main memory. In a paging scheme, pages are brought from secondary memory and are stored in main memory in a dynamic manner. All virtual addresses generated by a user program must be translated into physical memory addresses. This process is known as *dynamic address translation* and is shown in Figure 5.32.

When a running program accesses a virtual memory location $v = \langle p, n \rangle$, the mapping algorithm finds that the virtual page p is mapped to the physical frame p'. The physical address is then determined by appending p' to n.

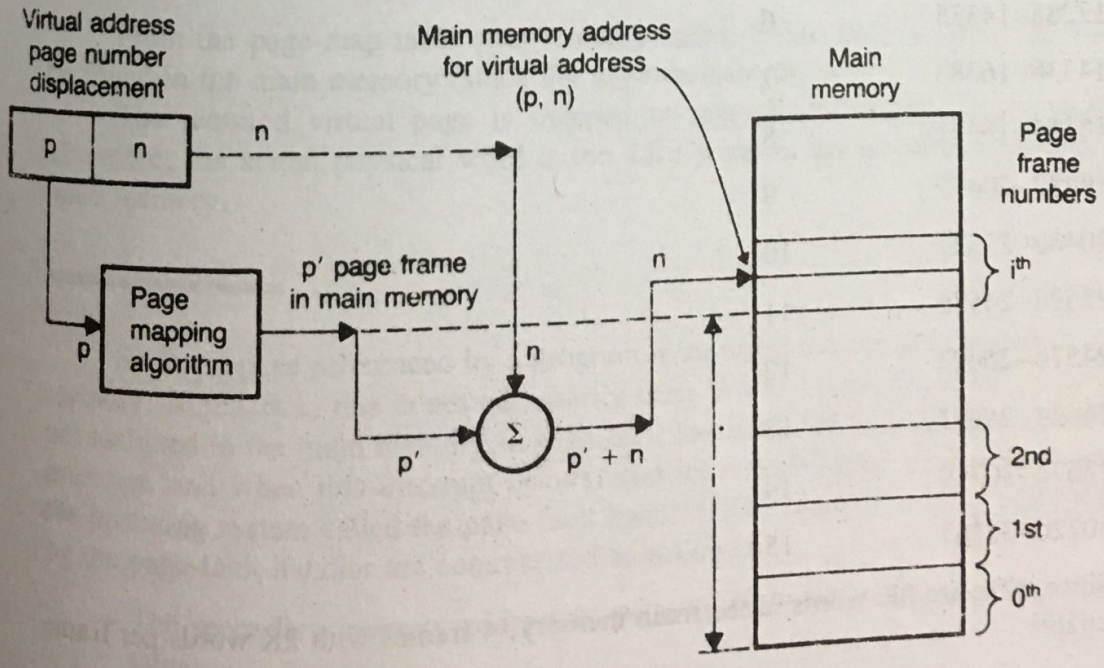


Figure 5.32 Paging Systems—Virtual versus Main Memory Mapping

This dynamic address translator can be implemented using a page table. In most systems, this table is maintained in the main memory. It will have one entry for each virtual page of the virtual address space. This is illustrated in the following example.

■

EXAMPLE 5.5

Design a mapping scheme with the following specifications:

- Virtual address space = 32K words
- Virtual memory size = 8K words
- Page size = 2K words
- Secondary memory address = 24 bits

SOLUTION

32K words can be divided into 16 virtual pages with 2K words per page, as follows:

VIRTUAL ADDRESS	PAGE NUMBER
0–2047	0
2048–4095	1
4096–6143	2
6144–8191	3
8192–10239	4
10240–12287	5
12288–14335	6
14336–16383	7
16384–18431	8
18432–20479	9
20480–22527	10
22528–24575	11
24576–26623	12
26624–28671	13
28672–30719	14
30720–32767	15

Since there are 8K words in the main memory, 4 frames with 2K words per frame are available:

PHYSICAL ADDRESS	FRAME NUMBER
0–2047	0
2048–4095	1
4096–6143	2
6144–8191	3

Since there are 32K addresses in the virtual space, 15 bits are required for the virtual address. Because there are 16 virtual pages, the page map table will contain 16 entries. The 4 most-significant bits of the virtual address can be used as an index to the page map table, and the remaining 11 bits of the virtual address can be used as the displacement to locate a word within the page frame. Each entry of the page table is 32 bits long. This obtained as follows:

1 bit for determining whether the page table is in main memory or not (residence bit).

2 bits for main memory page frame number.

24 bits for secondary memory address

5 bits for future use. (Unused at the moment)

32 bits total

The complete layout of the page table is shown in Figure 5.33.

Assume the virtual address generated is 0111 000 0010 1101. From this, compute the following:

Virtual page number = 7_{10}

Displacement = 43_{10}

From the page-map table entry corresponding to the address 0111, the page can be found in the main memory (since the page resident bit is 1).

The required virtual page is mapped to main memory page frame number 2. Therefore, the actual physical word is the 43rd word in the second page frame of the main memory.

So far, a page referenced by a program is assumed always to be found in the main memory. In practice, this is not necessarily true. When a page needed by a program is not assigned to the main memory, a page fault has occurred. A page fault is actually an interrupt, and when this interrupt occurs, control is transferred to a service routine of the operating system called the page-fault handler. The sequence of activities performed by the page-fault handler are summarized as follows:

- The secondary memory address of the required page p is located from the page table.

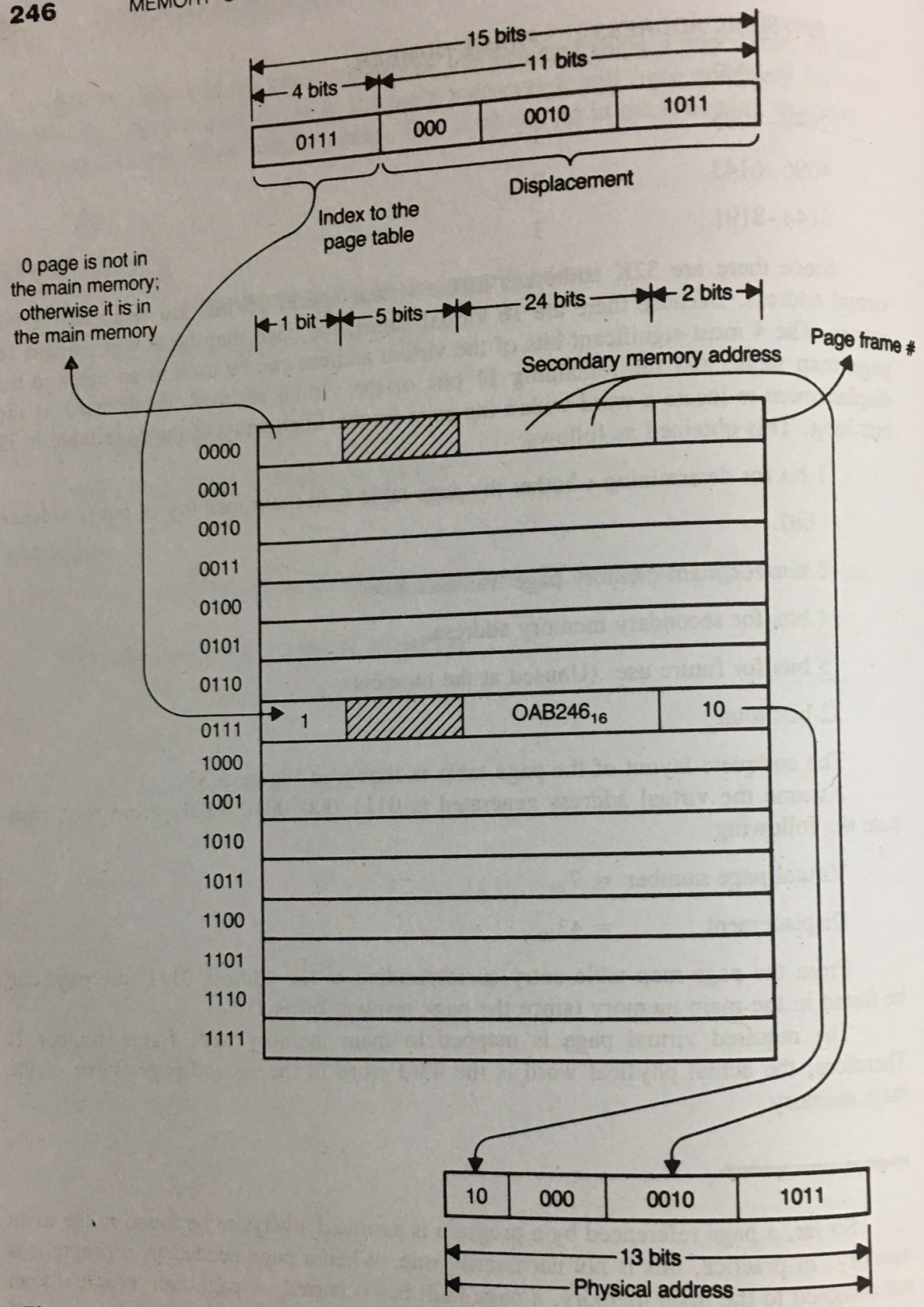


Figure 5.33 Mapping Scheme for the Pagiang System of Example 6.4

- Page p from the secondary memory is transferred into one of the available main memory frames by performing a block-move operation.
- The page table is updated by entering the frame number where page p is loaded and by setting the residence bit to 1 and the change bit to 0.

When a page-fault handler completes its task, control is transferred to the user program, and the main memory is accessed again for the required data or instruction. All these activities are kept hidden from a user. Pages are transferred to main memory only at specified times. The policy that governs this decision is known as the *fetch policy*. Similarly, when a page is to be transferred from the secondary memory to main memory, all frames may be full. At this point, one of the frames has to be removed from the main memory to provide room for an incoming page. The frame to be removed is selected using a replacement policy. The performance of a virtual memory system is dependent upon the fetch and replacement strategies. These issues are elaborated upon later.

The paging concept discussed so far is viewed as a one-dimensional technique because the virtual addresses generated by a program may linearly increase from 0 to some maximum value M. There are many situations where it is desirable to have a multidimensional virtual address space. This is the key idea behind segmentation systems.

Each logical entity such as a stack, an array, or a subroutine has a separate virtual address space in segmentation systems. Each virtual address space is called a *segment*, and each segment can grow from zero to some maximum value. Since each segment refers to a separate virtual address space, it can grow or shrink independently without affecting other segments.

In a segmentation system, the details about segments are held in a table called a *segment table*. Each entry in the segment table is called a *segment descriptor*, and it typically includes the following information:

- Segment base address b (starting address of the segment in the main memory)
- Segment length l (size of a segment)
- Segment presence bit
- Protection bits

From the structure of a segment descriptor, it is possible to create two or more segments whose sizes are different from one another. In a sense, a segmentation system degenerates to a paging system if all segments are of equal length. Because of this similarity, there is a close relationship between the paging and segmentation systems from the viewpoint of address translation.

A virtual address, V, in a segmentation system is regarded as an ordered pair $\langle s, d \rangle$, where s is the segment number and d is the displacement within segment s. The address translator for a segmentation system can be implemented using a segment table, and its organization is shown in Figure 5.34.

The mechanics of the address translation process is briefly discussed next.

Let V be the virtual address generated by the user program. First, the segment number field, s, of the virtual address V is used as an index to the segment table. The base address and length of this segment are b_s and l_s, respectively. Then, compare the displacement d of the virtual address V to the length of the segment l_s to make sure that the required address lies within the segment. Then, d must be less than or equal to l_s, and the comparator output z will be high. When $d \leq l_s$, the physical address is formed by adding b_s and d. From this physical address, data is retrieved and transferred to the

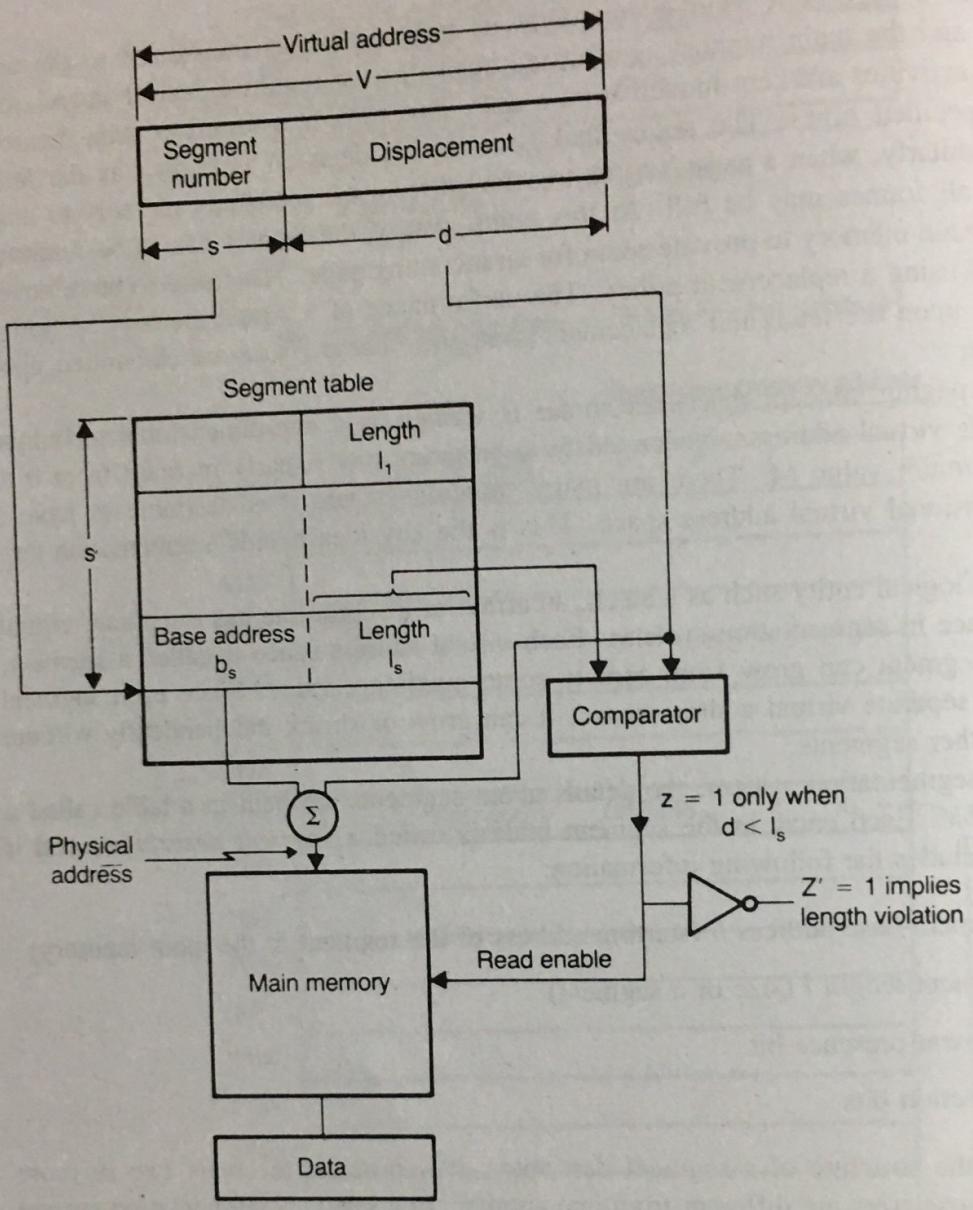


Figure 5.34 Address Translation in a Segmentation System

CPU. However, when $d > l_s$, the required address lies out of the segment range, and thus an *address out of range* trap will be generated. A *trap* is a nonmaskable interrupt with highest priority.

In a segmentation system, a segment needed by a program may not reside in main memory. This situation is indicated by a bit clled a *valid bit*. A valid bit serves the same purpose as that of a page resident bit, and thus it is regarded as a component of the segment descriptor. When the valid bit is reset to 0, it may be concluded that the required segment is not in main memory.

This means that its secondary memory address must be included in the segment descriptor. Recall that each segment represents a logical entity. This implies that we can protect segments with different protection protocols based on the logical contents of

the segment. The following are the common protection protocols used in a segmentation system:

- Read only
- Execute only
- Read and execute only
- Unlimited access
- No access

Thus it follows that these protection protocols have to be encoded into some protection codes and these codes have to be included in a segment descriptor.

In a segmented memory system, when a virtual address is translated into a physical address, one of the following traps may be generated:

- *Segment fault trap* is generated when the required segment is not in the main memory.
- *Address violation trap* occurs when $d > l_s$.
- *Protection violation trap* is generated when there is a protection violation.

When a segment fault occurs, control will be transferred to the operating system. In response, the operating system has to perform the following activities:

- First, it finds the secondary memory address of the required segment from its segment descriptor.
- Next, it transfers the required segment from the secondary to primary memory.
- Finally, it updates the segment descriptor to indicate that the required segment is in the main memory.

After performing the preceding activities, the operating system transfers control to the user program and the data or instruction retrieval or write operation is repeated.

A comparison of the paging and segmentation systems is provided next. The primary idea behind a paging system is to provide a huge virtual space to a programmer, allowing a programmer to be relieved from performing tedious memory-management tasks such as overlay design. The main goal of a segmentation system is to provide several virtual address spaces, so the programmer can efficiently manage different logical entities such as a program, data, or a stack.

The operation of a paging system can be kept hidden in the user level. However, a programmer is aware of the existence of a segmented memory system.

To run a program in a paging system, only its current page is needed in the main memory. Several programs can be held in the main memory and can be multiplexed. The paging concept improves the performance of a multiprogramming system. In contrast, a segmented memory system can be operated only if the entire program segment is held in the main memory.

In a paging system, a programmer cannot efficiently handle typical data structures such as stacks or sysmbol tables because their sizes vary in a dynamic fashion during

program execution. Typically, large pages for a symbol table or small pages for a stack cannot be created. In a segmentation system, a programmer can treat these two structures as two logical entities and define the two segments with different sizes.

The concept of segmentation encourages people to share programs efficiently. For example, assume a copy of a matrix multiplication subroutine is held in the main memory. Two or more users can use this routine if their segment tables contain copies of the segment descriptor corresponding to this routine. In a paging system, this task cannot be accomplished so elegantly because the system operation is hidden from the user. This result also implies that in a segmentation system, the user can apply protection features to each segment in any desired manner. However, a paging system does not provide such a versatile protection feature.

Since page size is a fixed parameter in a paging system, a new page can always be loaded in the space used by a page being swapped out. However, in a segmentation system with uneven segment sizes, there is no guarantee that an incoming segment can fit into the free space created by a segment being swapped out.

In a dynamic situation, several programs may request more space, whereas some other programs may be in the process of releasing the spaces used by them. When this happens in a segmented memory system, there is a possibility that uneven-sized free spaces may be sparsely distributed in the physical address space. These free spaces are so irregular in size that they cannot normally be used to satisfy any new request. This is called an *external fragmentation,* and an operating system has to merge all free spaces to form a single large useful segment by moving all active segments to one end of the memory. This activity is known as *memory compaction*. This is a time-consuming operation and is a pure overhead. Since pages are of equal size, no external fragmentation can occur in a paging system.

In a segmented memory system, a programmer defines a segment, and all segments are completely filled.

The page size is decided by the operating system, and the last page of a program may not be filled completely when a program is stored in a sequence of pages. The space not filled in the last page cannot be used for any other program. This difficulty is known as *internal fragmentation*—a potential disadvantage of a paging system.

In summary, the paging concept simplifies the memory-management tasks to be performed by an operating system and is an elegant idea from the standpoint of an operating system. The segmentation concept offers a great appeal to programmers because it allows both protection and sharing of logical entities among a group of programmers.

To take advantage of both paging and segmentation, some systems use a different approach, in which these concepts are merged. In this technique, a segment is viewed as a collection of pages. The number of pages per segment may vary. However, the number of words per page still remains fixed. In this situation, a virtual address V is an ordered triple $\langle s, p, d \rangle$, where s is the segment number and p and d are the page number and the displacement within a page, respectively.

The following tables are used to translate a virtual address into a physical address:

- *Page table:* This table holds pointers to the physical frames.

Segment table: Each entry in the segment table contains the base address of the page table that holds the details about the pages that belong to the given segment.

The address-translation scheme of such a paged-segmentation system is shown in Figure 5.35:

- First, the segment number s of the virtual address is used as an index to the segment table, which leads to the base address b_p of the page table.
- Then, the page number p of the virtual address is used as an index to the page table, and the base address of the frame number p' (to which the page p is mapped) can be found.
- Finally, the physical memory address is computed by adding the displacement d of the virtual address to the base address p' obtained before.

To illustrate this concept, the following numerical example is provided.

■

EXAMPLE 5–6

Assume the following values for the system of Figure 5.35:

- Length of the virtual address field = 32 bits
- Length of the segment number field = 12 bits
- Length of the page number field = 8 bits
- Length of the displacement field = 12 bits

Now, determine the value of the physical address using the following information:

- Value of the virtual address field = $000FAOBA_{16}$
- Contents of the segment table address $(000)_{16} = OFF_{16}$
- Contents of the page table address $(1F9_{16}) = AC_{16}$

SOLUTION

From the given virtual address, the segment table address is 000_{16} (three high-order hexadecimal digits of the virtual address). It is given that the contents of this segment-able address is OFF_{16}. Therefore, by adding the page number p (fourth and fifth hexadecimal digits of the virtual address) with OFF_{16}, the base address of the page table can be determined as:

$$OFF_{16} + FA_{16} = 1F9_{16}$$

From the hypothesis, the contents of the page table address $IF9_{16}$ is AC_{16}. Hence, the physical address can be obtained by adding the displacement (low-order three hexadecimal digits of the virtual address) with AC_{16} as follows:

$AC000_{16}$

000BA +

ACOBA

In this addition, the displacement value OBA is sign-extended to obtain a 20-bit number that can be directly added to the base value p′. The same final answer can be obtained if p′ and d are first concatenated. Thus, the value of the physical address is $ACOBA_{16}$.

The figures quoted in Example 5.6 are typical of IBM 360/67 computers. Since the virtual space of these machines use both paging and segmentation, it is called a *linear segmented virtual memory system*. In this system, the main memory is accessed three times to retrieve data (one for accessing the page table; one for accessing the segment table; and one for accessing the data itself).

Accessing the main memory is a time-consuming operation. To speed up the retrieval operation, a new approach is used. A small associative memory is used, called the translation lookaside buffer (TLB). The TLB stores the translation information for the 8 or 16 most recent virtual addresses. The organization of a address translation scheme that includes a TLB is shown in Figure 5.36.

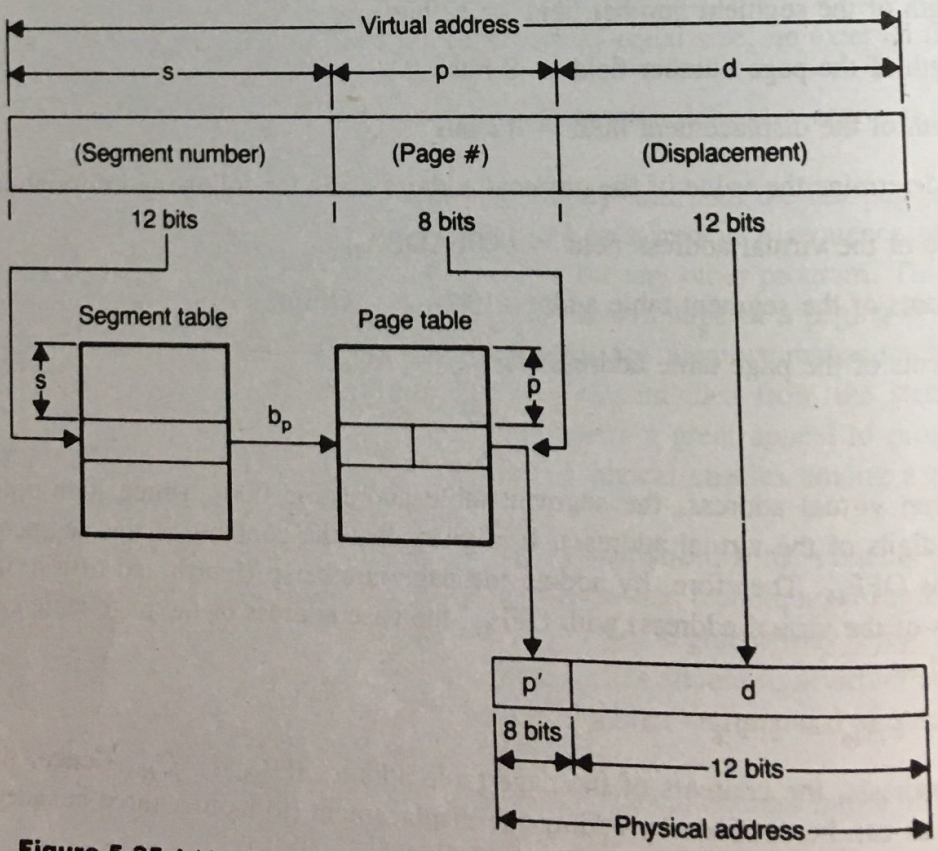


Figure 5.35 Address-translation Scheme for a Paged-segmentation System

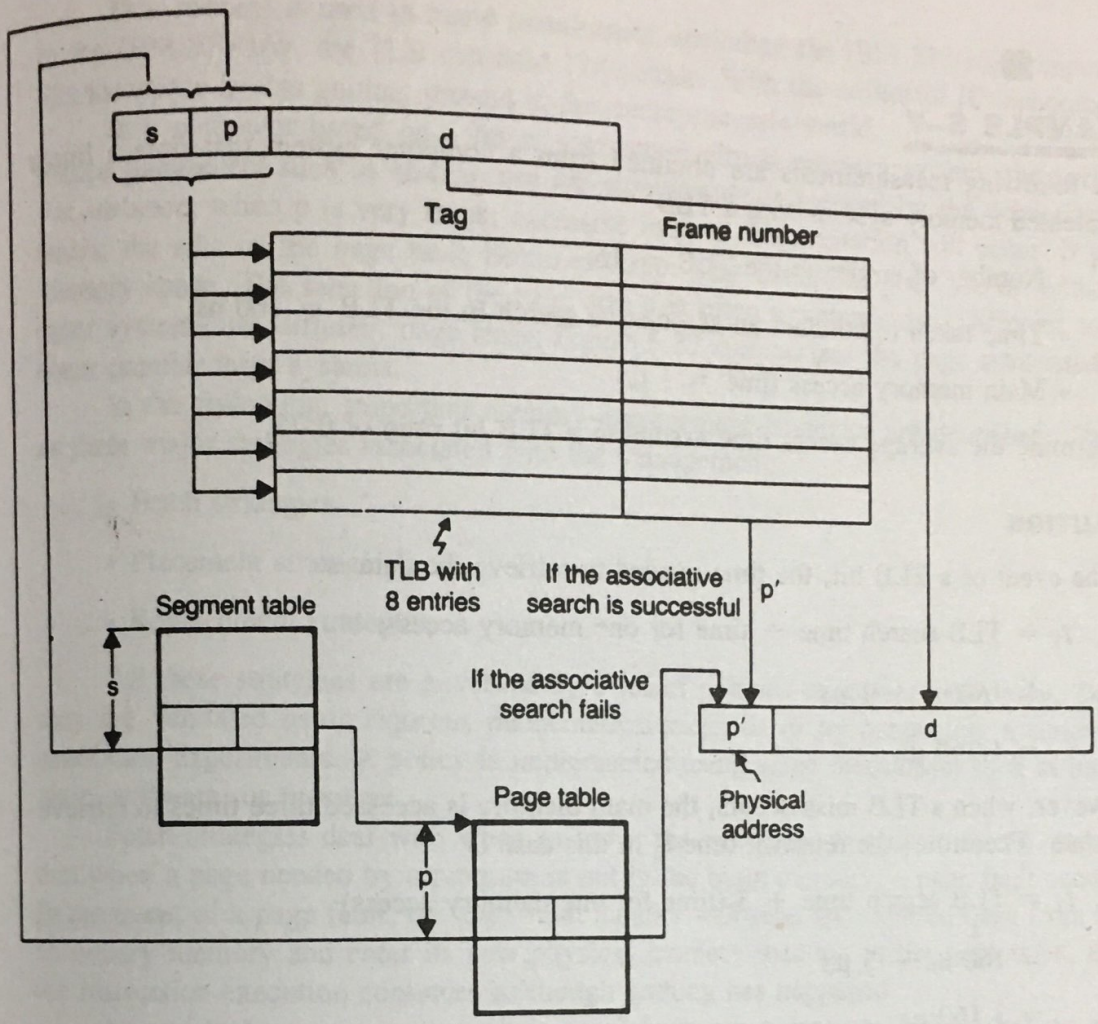


Figure 5.36 Address Translation Using a TLB

In this scheme, assume the TLB is capable of holding the translation information about the 8 most recent virtual addresses.

The pair ⟨s, p⟩ of the virtual address is known as a *tag*, and each entry in the TLB is of the form:

⟨s, p⟩ or tag	Base address of the frame p′

When a user program generates a virtual address, the ⟨s, p⟩ pair is associatively compared with all tags held in the TLB for a match. If there is a match, the physical address is formed by retrieving the base address of the frame p′ from the TLB and concatenating this with the displacement d. However, in the event of a TLB miss, the physical address is generated after accessing the segment and page tables, and this informatioan will also be loaded in the TLB. This ensures that translation information pertaining to a future reference is confined to the TLB. To illustrate the effectiveness of the TLB, the following numerical example is provided.

■

EXAMPLE 5–7

The following measurements are obtained from a computer system that uses a linear segmented memory system with a TLB:

- Number of entries in the TLB = 16
- Time taken to conduct an associative search in the TLB = 160 ns
- Main memory access time = 1 μs

Determine the average access time assuming a TLB hit ratio of 0.75.

SOLUTION

In the event of a TLB hit, the time needed to retrieve the data is:

t_1 = TLB search time + time for one memory access

= 160 ns + 1 μs

= 1.160 μs

However, when a TLB miss occurs, the main memory is accessed three times to retrieve the data. Therefore, the retrieval time t_2 in this case is

t_2 = TLB search time + 3 (time for one memory access)

= 160 ns + 3 μs

= 3.160 μs

The average access time $\hat{t}$ is

$$\hat{t} = ht_1 + (1 - h)t_2$$

where h is the TLB hit ratio.

The average access time $\hat{t}$ is:

$\hat{t}$ = 0.75 (1.6) + 0.25 (3.160) μsec

= 1.2 + 0.79 μsec

= 1.99 μsec

This example shows that the use of a small TLB significantly improves the efficiency of the retrieval operation (by 33%). There are two main reasons for this improvement. First, the TLB is designed using the associated memory. Second, the TLB hit ratio may be attributed to the locality of reference. Simulation studies indicate that it is possible to achieve a hit ratio in the range of 0.8 to 0.9 by having a TLB with 8 to 16 entries.

This concept is used in many mainframes, including the IBM 370/168 computer. In the IBM 370/168, the TLB can hold 128 entries. With the advent of IC technology, this technique is also gaining ground in the microprocessor world.

In a computer based on a linear segmented virtual memory system, the performance parameters such as storage use are significantly influenced by the page size p. For instance, when p is very large, excessive internal fragmentation will occur. If p is small, the size of the page table becomes large. This results in poor use of valuable memory space. The selection of the page size p is often a compromise. Different computer systems use different page sizes; Figure 5.37 summarizes the page sizes used in some popular large systems.

In the following, important memory-management strategies are described. There ae three major strategies associated with the management:

- Fetch strategies
- Placement strategies
- Replacement strategies

All these strategies are governed by a set of policies conceived intuitively. Then they are validated using rigorous mathematical methods or by conducting a series of simulation experiments. A policy is implemented using some mechanism such as hardware, software, or firmware.

Fetch strategies deal with when to move the next page to main memory. Recall that when a page needed by a program is not in the main memory, a page fault occurs. In the event of a page fault, the page-fault handler will read the required page from the secondary memory and enter its new physical memory location in the page table, and the instruction execution continues as though nothing has happened.

In a virtual memory system, it is possible to run a program without having any page in the primary memory. In this case, when the first instruction is attempted, there is a page fault. As a consequence, the required page is brought into the main memory, where the instruction execution process is repeated again. Similarly, the next instruction may also cause a page fault. This situation is handled exactly in the same manner as described before. This strategy is referred to as *demand paging* because a page is brought in only when it is needed. This idea is useful in a multiprogramming environment because several programs can be kept in the main memory and executed concurrently.

Model	Page Size	Word Length
IBM 360/67	4096	32 bits
IBM 370/168	1024 or 512	32 bits
DEC 10	512	36 bits
VAX 11/780	512	32 bits

Figure 5.37 Page Sizes in Some Popular Computer Systems

multiprogramming system, when the execution of a suspended program is resumed, its present working set can be reasonably estimated based on the value of its working set at the time it was suspended. If this estimated working set is loaded, page faults are less likely to occur. This anticipatory fetching further improves the system performance because the working set of a program can be loaded while another program is being executed by the CPU.

However, the accuracy of a working set model depends on the value of m. Larger values of m result in more-accurate predictions. Typical values of m lie in the range of 5000 to 10,000.

To keep track of the working set of a program, the operating system has to perform time-consuming housekeeping operations. This activity is pure overhead, and thus the system performance may be degraded.

Placement strategies are significant with segmentation systems, and they are concerned with where to place an incoming progrm or data in the main memory. The following are the three widely used placement strategies:

- First-fit technique
- Best-fit technique
- Worst-fit technique

The first-fit technique places the program in the first available free block or hole that is adequate to store it. The best-fit technique stores the program in the smallest free hole of all the available holes able to store it. The worst-fit technique stores the program in the largest free hole.

The first-fit technique is easy to implement and does not have to scan the entire space to place a program. The best-fit technique appears to be efficient because it finds an optimal hole size. However, it has the following drawbacks:

- It is very difficult to implement.
- It may have to scan the entire free space to find the smallest free hole that can hold the incoming program. Therefore, it may be time-consuming.
- It has the tendency continuously to divide the holes into smaller sizes. These smaller holes may eventually become useless.

Worst-fit strategy is sometimes used when the design goal is to avoid creating small holes. In general, the operating system maintains a list known as the available space list (ASL) to indicate the free memory space. Typically, each entry in this list includes the following information:

- Starting address of the free block
- Size of the free block

After each allocation or release, the operating updates the ASL.

In the following example, the mechanics of the various placement strategies presented earlier are explained.

EXAMPLE 5.9

The available space list of a computer memory system is specified as follows:

STARTING ADDRESS	BLOCK SIZE (IN WORDS)
100	50
200	150
450	600
1200	400

Determine the available space list after allocating the space for the stream of requests consisting of the following block sizes:

25, 100, 250, 200, 100, 150

a) Use the first-fit method.

b) Use the best-fit method.

c) Use the worst-fit method.

SOLUTION

a) *First-fit method*. Consider the first request with a block size of 25. Examination of the block sizes of the available space list reveals that this request can be satisfied by allocating from the first available block. The block size (50) is the first of the available space list and is adequate to hold the request (25 blocks). Therefore, the first request with 25 blocks will be allocated from the available space list starting at address 100 with a block size of 50. Request 1 will be allocated starting at an address of 100 ending at an address $100 + 24 = 124$ (25 locations including 100). Therefore, the first block of the available space list will start at 125 with a block size of 25.

The starting address and block size of each request can be calculated similarly.

b) *Best-fit method*. Consider request 1. Examination of the available block size reveals that this request can be satisfied by allocating from the first smallest available block capable of holding it. Request 1 will be allocated starting at address 100 and ending at 124. Therefore, the available space list will start at 125 with a block size of 25.

c) *Worst-fit method*. Consider requet 1. Examination of the available block sizes reveals that this request can be satisfied by allocating from the third block (largest) starting at 450. After this allocation the starting address of the available list will be 500 instead of 450 with a block size of $600 - 25 = 575$. Various results for all the other requests are shown in Figure 5.40.

Figure 5.40 Memory Map after Allocating Space for All Requests Given Example Using Different Placement Strategies

	Request 1 (25)		Request 2 (100)		Request 3 (250)		Request 4 (200)		Request 5 (100)		Request 6 (150)	
	Start address	Block size	Start address	Block size	Start address	Block size	Start address	Block size	Start address	Block size	Start address	Block size
First fit	125	25	125	25	125	25	125	25	125	25	125	25
	200	150	300	50	300	50	300	50	300	50	300	50
	450	600	450	600	700	350	900	150	1000	50	1000	50
	1200	400	1200	400	1200	400	1200	400	1200	400	1350	250
Best fit	125	25	125	25	125	25	125	25	125	25	125	25
	200	150	300	50	300	50	300	50	300	50	300	50
	450	600	450	600	450	600	650	400	650	400	800	250
	1200	400	1200	400	1450	150	1450	150	1550	50	1550	50
Worst fit	100	50	100	50	100	50	100	50	100	50	100	50
	200	150	200	150	200	150	200	150	200	150	200	150
	500	575	600	475	850	225	850	225	950	125	850	125
	1200	400	1200	400	1200	400	1400	200	1400	200	1550	50

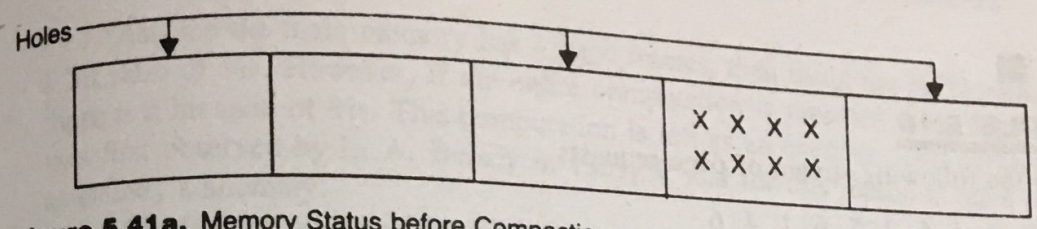
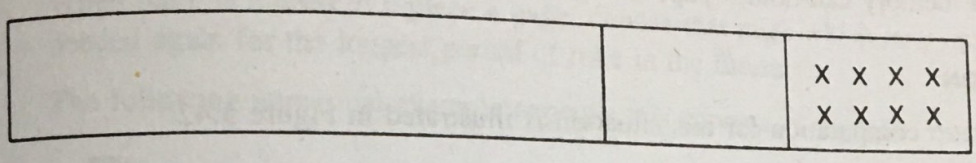


Figure 5.41a. Memory Status before Compaction



b. Memory Status after Compaction

In a multiprogramming system, programs of different sizes may reside in the main memory. As these programs are completed, the allocated memory space becomes free. It may happen that these unused free spaces, or holes, become available between two allocated blocks, or partitions. Some of these holes may not be large enough to satisfy the memory request of a program waiting to run. Thus valuable memory space may be wasted. One way to get around this problem is to combine adjacent free holes to make the hole size larger and usable by other jobs. This technique is known as *coalescing of holes*.

It is possible that the memory request made by a program may be larger than any free hole but smaller than the combined total of all available holes. If the free holes are combined into one single hole, the request can be satisfied. This technique is known as *memory compaction*. For example, the status of a computer memory before and after memory compaction is shown in Figure 5.41(a) and (b), respectively.

Placement strategies such as first-fit and best-fit are usually implemented as software procedures. These procedures are included in the operating system's software. The advent of high-level languages such as Pascal and C greatly simplify the programming effort because they support abstract data objects such as pointers. The available space list discussed in this section can easily be implemented using pointers.

The memory compaction task is performed by a special software routine of the operating system called a *garbage collector*. Normally, an operating system runs the garbage collector routine at regular intervals.

In a paged virtual memory system, when no frames are vacant, it is necessary to replace a current main memory page to provide room for a newly fetched page. The page for replacement is selected using some replacement policy. An operating system implements the chosen replacement policy. In general, a replacement policy is considered efficient if it guarantees a high hit ratio. The hit ratio h is defined as the ratio of the number of page references that did not cause a page fault to the total number of page references.

The simplest of all page replacement policies is the FIFO policy. This algorithm selects the oldest page (or the page that arrived first) in the main memory for replacement. The hit ratio h for this algorithm can be analytically determined using some arbitrary stream of page references as illustrated in the following example.

EXAMPLE 5.10

Consider the following stream of page requests.

2, 3, 2, 4, 6, 2, 5, 6, 1, 4, 6

Determine the hit ratio h for this stream using the FIFO replacement policy. Assume the main memory can hold 3 page frames and initially all of them are vacant.

SOLUTION

The hit ratio computation for this situation is illustrated in Figure 5.42.

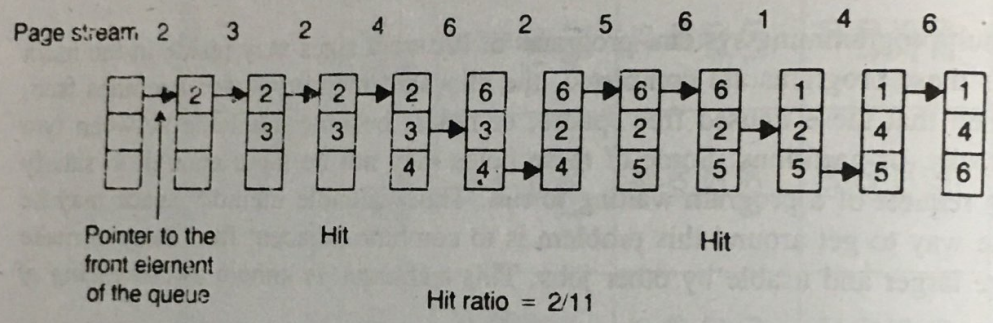


Figure 5.42 Hit Ratio Computation for Example 5.10

From Figure 5.42, it can be seen that the first two page references cause page faults. However, there is a hit with the third reference because the required page (page 2) is already in the main memory. After the first four references, all main memory frames are completely used. In the fifth reference, page 6 is required. Since this page is not in the main memory, a page fault occurs. Therefore, page 6 is fetched from the secondary memory. Since there are no vacant frames in the main memory, the oldest of the current main memory pages is selected for replacement. Page 6 is loaded in this position. All other data tabulated in this figure are obtained in the same manner. Since 9 out of 11 references generate a page fault, the hit ratio is $^2/_{11}$.

The primary advantage of the FIFO algorithm is its simplicity. This algorithm can be implemented by using a FIFO queue. FIFO policy gives the best result when page references are made in a strictly sequential order. However, this algorithm fails if a program loop needs a variable introduced at the beginning. Another difficulty with the FIFO algorithm is it may give anomalous results.

Intuitively, one may feel that an increase in the number of page frames will also increase the hit ratio. However, with FIFO, it is possible that when the page frames are increased, there is a drop in the hit ratio. Consider the following stream of requests:

1, 2, 3, 4, 5, 1, 2, 5, 1, 2, 3, 4, 5, 6, 5