Fundamentals of Digital Logic and Microcomputer Design. M. Rafiquzzaman

7

DESIGN OF COMPUTER INSTRUCTION SET AND THE CPU

This chapter describes the design of the instruction set and the central processor unit (CPU). Topics include op-code encoding, design of typical microprocessor registers, the arithmetic logic unit (ALU), and the control unit.

7.1 Design of the Computer Instructions

A program consists of a sequence of instructions. An instruction performs operations on stored data. There are two components in an instruction: an op-code field and an address field. The op-code field defines the type of operation to be performed on data, which may be stored in a microprocessor register or in the main memory. The address field may contain one or more addresses of data. When data are read from or stored into two or more addresses by the instruction, the address field may contain more than one address. For example, consider the following instruction:

```
MOVE        D0,D1
```

Op-code field Address field

Assume that this computer uses D0 as the source register and D1 as the destination register. This instruction moves the contents of the microprocessor register D0 to register D1. The number and types of instructions supported by a microprocessor vary from one microprocessor to another and primarily depend on the microprocessor architecture. The number of instructions supported by a typical microprocessor depends on the size of the op-code field. For example, an 8-bit op-code can specify a maximum of 256 unique instructions.

As mentioned before, a computer only understands 1's and 0's. This means that the computer can execute an instruction only if it is in binary. A unique binary pattern must be assigned to each op-code by a process called "op-code encoding."

The Block code method is one of the simplest techniques of designing instructions. In this approach, a fixed length of binary pattern is assigned to each op-code. For example, an n-bit binary number can represent 2^n unique op-codes. Consider for example, a hypothetical instruction set shown in Figure 7.1. In this figure, there are 8 different instructions that can be encoded using three bits i_2, i_1, i_0 as shown in Figure 7.2. A 3-to-8 decoder can be used to encode the 8 hypothetical instructions as shown in Figure 7.3.

An n-to-2^n decoder is required for an n-bit op-code. As n increases, the cost of the decoder and decoding time will also increase. In some op-code encoding techniques such as

the "expanding op-code" method, the length of the instruction is a function of the number of addresses used by the instruction. For example, consider a 16-bit instruction in which the lengths of the op-code and address fields are 5 bits and 11 bits respectively. Using such an instruction format, 32 (2^5) operations allowing access to 2048 (2^{11}) memory locations can be specified. Now, if the size of the instruction is kept at 16 bits but the address field is increased to 12 bits, the op-code length will then be decreased to 4 bits. This change will specify 16 (2^4) operations with access to 4096 (2^{12}) memory locations. Thus, the number of

Instruction	Operation Performed
MOVE reg_1, reg_2	$reg_2 \leftarrow reg_1$
CLR reg	reg ← 0
ADD reg_1, reg_2	$reg_2 \leftarrow reg_1 + reg_2$
SUB reg_1, reg_2	$reg_2 \leftarrow reg - reg_1$
AND reg_1, reg_2	$reg_2 \leftarrow reg_1$ AND reg_2
OR reg_1, reg_2	$reg_2 \leftarrow reg_1$ OR reg_2
INC reg	reg ← reg + 1
JMP addr	PC ← addr; Unconditionally Jump to addr

FIGURE 7.1 A hypothetical instruction set

Instruction	3-Bit Op-Code		
	i_2	i_1	i_0
MOVE	0	0	0
CLR	0	0	1
ADD	0	1	0
SUB	0	1	1
AND	1	0	0
OR	1	0	1
INC	1	1	0
JMP	1	1	1

FIGURE 7.2 Op-code encoding using block code

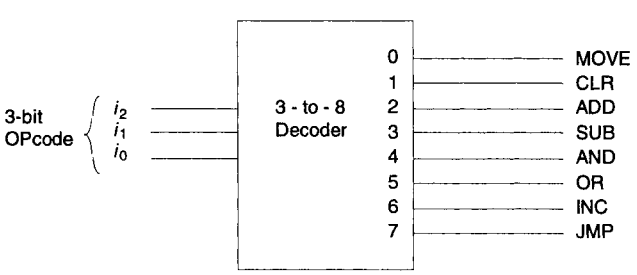


FIGURE 7.3 Instruction decoder

operations is reduced by 50% and the number of memory locations is increased by 100%. This concept is used in designing instructions with expanding op-code technique.

Consider an instruction format with 8-bit instruction length and a 2-bit op-code field. Four unique two-address (3 bits for each address) instructions can be specified. This is depicted in Figure 7.4. If three rather than four two-address instructions are used, eight one-address instructions can be specified. This is shown in Figure 7.5. The length of the op-code field for each one-address instruction is 5 bits. Thus, the length of the op-code field increases as the number of address field is decreased. Now, if the total number of one-address instructions is reduced from 8 to 7, then eight 0-address instructions can also be specified. This is shown in Figure 7.6.

7.2 Reduced Instruction Set Computer (RISC)

RISC, which stands for reduced instruction set computer, is a generation of faster and inexpensive machines. The initial application of RISC principles has been in desktop workstations. Note that the PowerPC is a RISC microprocessor. The basic idea behind

OP- Code (2-bits)	Address 1 (3-bits)	Address 2 (3-bits)
$i_1\ i_0$		
0 0	$x_2\ x_1\ x_0$	$y_2\ y_1\ y_0$
0 1	$x_2\ x_1\ x_0$	$y_2\ y_1\ y_0$
1 0	$x_2\ x_1\ x_0$	$y_2\ y_1\ y_0$
1 1	$x_2\ x_1\ x_0$	$y_2\ y_1\ y_0$

FIGURE 7.4 Four two-address instructions

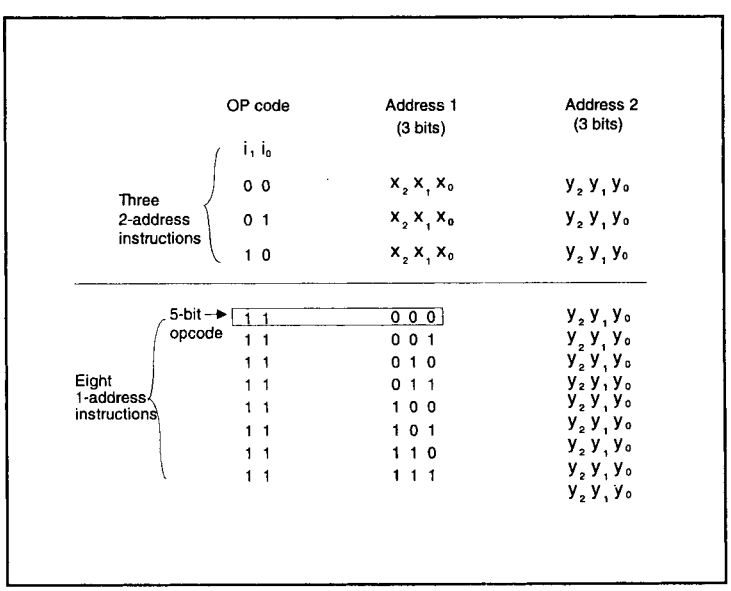


FIGURE 7.5 Three 2-address and eight 1-address instructions

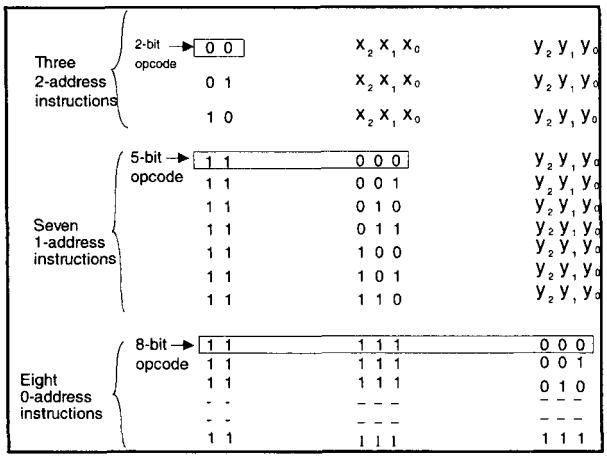


FIGURE 7.6 3 two-address, 7 one-address, and 8 zero-address instructions

RISC is for machines to cost less yet run faster, by using a small set of simple instructions for their operations. Also, RISC allows a balance between hardware and software based on functions to be achieved to make a program run faster and more efficiently. The philosophy of RISC is based on six principles: reliance on optimizing compilers, few instructions and addressing modes, fixed instruction format, instructions executed in one machine cycle, only call/return instructions accessing memory, and hardwired control.

The trend has always been to build CISCs (complex instruction set computers), which use many detailed instructions. However, because of their complexity, more hardware would have to be used. The more instructions, the more hardware logic is needed to implement and support them. For example, in a RISC machine, an ADD instruction takes its data from registers. On a CISC, each operand can be stored in any of many different forms, so the compiler must check several possibilities. Thus, both RISC and CISC have advantages and disadvantages. However, the principles of understanding optimizing compilers and what actually happens when a program is executed lead to RISC.

Case Study: RISC I (University of California, Berkeley)

The RISC machine presented in this section is the one investigated at the University of California, Berkeley. The RISC I is designed with the following design constraints:

1. Only one instruction is executed per cycle.
2. All instructions have the same size.
3. Only load and store instructions can access memory.
4. High-level languages (HLL) are supported.

Two high level Languages (C and Pascal) were supported by RISC I. A simple architecture implies a fewer transistors, and this leads to the fact that most pieces of a RISC HLL system are in software. Hardware is utilized for time-consuming operations. Using C and Pascal, a comparison study was made to determine the frequency of occurrence of particular variable and statement types. Studies revealed that integer constants appeared most frequently, and a study of the code produced revealed that the procedure calls are the most time-consuming operations.

i) Basic RISC Architecture

The RISC I instruction set contains a few simple operations (arithmetic, logical, and shift). These instructions operate on registers. Instruction, data, addresses and registers are all 32 bits long. RISC instructions fall in four categories: ALU, memory access, branch, and miscellaneous. The execution time is given by the time taken to read a register, perform an ALU operation, and store the result in a register. Register 0 always contains a 0. Load and store instructions move data between registers and memory. These instructions use two CPU cycles. Variations of memory-access instructions exist in order to accommodate sign-extended or zero-extended 8-bit, 16-bit and 32-bit data. Though absolute and register indirect addressing are not directly available, they may be synthesized using register 0. Branch instructions include CALL, RETURN, and conditional and unconditional jumps. The following instruction format is used:

opcode(7)	scc(1)	dest(5)	source1(5)	imm(1)	source2(13)

For register-to-register instructions, dest selects one of the 32 registers as destination of the result of the operation that is itself performed on registers source 1 and source2. If imm equals 0, the low-order 5 bits of source2 specify another register. If imm equals 1, then source2 is regarded as a sign-extended 13-bit constant. Since the frequency of integer constants is high, the immediate field has been made an option in every instruction. Also, Scc determines whether the condition codes are set. Memory-access instructions use source 1 to specify the index register and source2 to specify offset.

ii) Register Windows

The procedure-call statements take the maximum execution time. A RISC program has more call statements, since the complex instructions available in CISC are subroutines in RISC. The RISC register window scheme strives to make the call operation as fast as possible and also to reduce the number of accesses to data memory. The scheme works as follows.

Using procedures involve two groups of time-consuming operations, namely, saving or restoring registers on each call/return and passing parameters and results to and from the procedure. Statistics indicate that local variables are the most frequent operands.

This creates a need to support the allocation of locals in the registers. One available scheme is to provide multiple banks of registers on the chip to avoid saving and restoring of registers. Thus each procedure call results in a new set of registers being allocated for use by that procedure. The return alters a pointer that restores the old set. A similar scheme is adopted by RISC. However, there are some registers that are not saved or restored; these are called global registers. In addition, the sets of registers used by different processes are overlapped in order to allow parameters to be passed. In other machines, parameters are usually passed on the stack with the calling procedure using a register to point to the beginning of the parameters (and also to the end of the locals). Thus all references to parameters are indexed references to memory. In RISC I the set of window registers (r10 to r31) is divided into three parts. Registers r26 to r31 (HIGH) contain parameters passed from the calling procedure. Registers rl6 to r25 (LOCAL) are for local storage. Registers r10 to rl5 (LOW) are for local storage and for parameters to be passed to the called procedure. On each call, a new set of r10 to r31 registers is allocated. The LOW registers of the caller are required to become the HIGH registers of the called procedure. This is accomplished by having the hardware overlap the LOW registers of the calling frame with the HIGH registers of the called frame. Thus without actually moving the information, parameters are

transferred.

Multiple register banks require a mechanism to handle the case in which there are no free register banks available. RISC handles this problem with a separate register-overflow stack in memory and a stack pointer to it. Overflow and underflow are handled with a trap to a software routine that adjusts the stack. The final step in allocating variables in registers is handling the problem of pointers. RISC resolves this by giving addresses to the window registers. If a portion of the address space is reserved, we can determine with one comparison whether an address points to a register or to memory. Load and store are the only instructions that access memory and they take an extra cycle already. Hence this feature may be added without reducing the performance of the load and store instructions. This permits the use of straightforward computer technology and still leaves a large fraction of the variables in registers.

iii) Delayed Jump

A normal RISC I instruction cycle is long enough to execute the following sequence of operations:

1. Read a register.
2. Perform an ALU operation.
3. Store the result back into a register.

Performance is increased by prefetching the next instruction during the current instruction. To facilitate this, jumps are redefined such that they do not occur until after the following instruction. This is called delayed jump.

7.3 Design of the CPU

The CPU contains three elements: registers, the ALU (Arithmetic Logic Unit), and the control unit. These topics are discussed next. Verilog and VHDL descriptions along with simulation results of a typical CPU are provided in Appendices I and J respectively.

7.3.1 Register Design

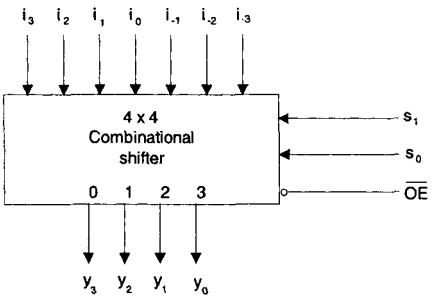
The concept of general-purpose and flag registers is provided in Chapters 5 and 6. The main purpose of a general-purpose register is to store address or data for an indefinite period of time. The computer can execute an instruction to retrieve the contents of this register when needed. A computer can also execute instructions to perform shift operations on the contents of a general-purpose register. This section includes combinational shifter design and the concepts associated with barrel shifters.

A high-speed shifter can be designed using combinational circuit components such as a multiplexer. The block diagram, internal organization, and truth table of a typical combinational shifter are shown in Figure 7.7. From the truth table, the following equations can be obtained:

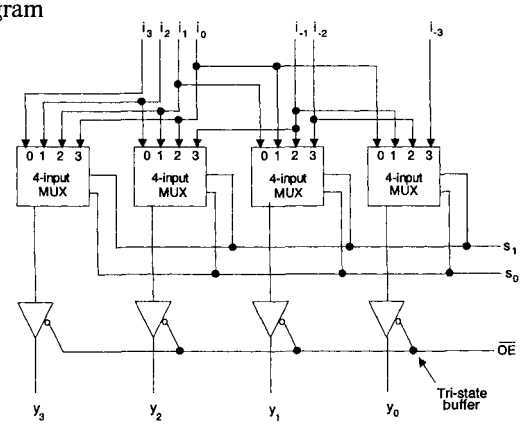
$$y_3 = \overline{s_1}\,\overline{s_0} i_3 + \overline{s_1} s_0 i_2 + s_1 \overline{s_0} i_1 + s_1 s_0 i_0$$
$$y_2 = \overline{s_1}\,\overline{s_0} i_2 + \overline{s_1} s_0 i_1 + s_1 \overline{s_0} i_0 + s_1 s_0 i_{-1}$$
$$y_1 = \overline{s_1}\,\overline{s_0} i_1 + \overline{s_1} s_0 i_0 + s_1 \overline{s_0} i_{-1} + s_1 s_0 i_{-2}$$
$$y_0 = \overline{s_1}\,\overline{s_0} i_0 + \overline{s_1} s_0 i_{-1} + s_1 \overline{s_0} i_{-2} + s_1 s_0 i_{-3}$$

The 4 × 4 shifter of Figure 7.7 can be expanded to obtain a system capable of rotating 16-bit data to the left by 0, 1, 2, or 3 positions, which is shown in Figure 7.8.

This design can be extended to obtain a more powerful shifter called the *barrel*



(a) Block Diagram



(b) Internal Schematic

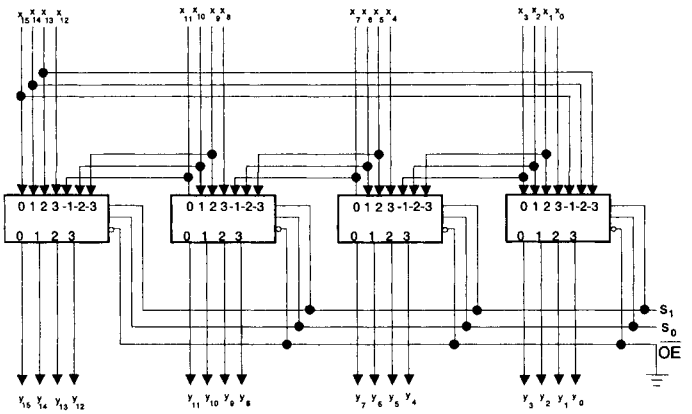
	Shift Count		Output				Comment
$\overline{OE}$	s_1	s_0	y_3	y_2	y_1	y_0	
1	X	X	Z	Z	Z	Z	Outputs are tristated
0	0	0	i_3	i_2	i_1	i_0	Pass (no shift)
0	0	1	i_2	i_1	i_0	i_{-1}	Left Shift once
0	1	0	i_1	i_0	i_{-1}	i_{-2}	Left shift twice
0	1	1	i_0	i_{-1}	i_{-2}	i_{-3}	Left shift three times

(c) Truth Table (X is don't care in the above)

FIGURE 7.7 4 × 4 combinational shifter

shifter. The shift is a cycle rotation, which means that the input binary information is shifted in one direction; the most significant bit is moved to the least significant position.

The block-diagram representation of a 16 × 16 barrel shifter is shown in Figure 7.9. This shifter is capable of rotating the given 16-bit data to the left by n positions, where $0 \leq n \leq 15$. Figure 7.9 shows the truth table representing the operation of the shifter. The barrel shifter is an on-chip component for typical 32-bit and 64-bit microprocessors.



(a) Logic Diagram

Shift Count		Output															
S_1	S_0	y_{15}	y_{14}	y_{13}	y_{12}	y_{11}	y_{10}	y_9	y_8	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
0	0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
0	1	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}
1	0	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}
1	1	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}

(b) Truth Table

FIGURE 7.8 Combinational shifter capable of rotating 16-bit data to the left by 0, 1, 2, or 3 positions

7.3.2 Adders

Addition is the basic arithmetic operation performed by an ALU. Other operations such as subtraction and multiplication can be obtained via addition. Thus, the time required to add two numbers plays an important role in determining the speed of the ALU.

The basic concepts of half-adder, full adder, and binary adder are discussed in Section 4.5.1. The following equations for the full-adder were obtained. Assume $x_i = x$, $y_i = y$, $c_i = z$, and $C_{i+1} = C$ in Table 4.6.

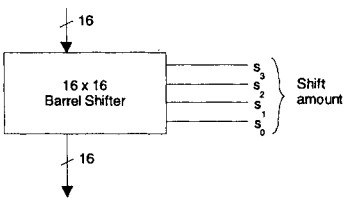
$$\text{Sum, } S_i = \bar{x}_i\bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i$$
$$= x_i \oplus y_i \oplus c_i$$

From Table 4.6,

$$\text{Carry, } C_{i+1} = \bar{x}_i y_i c_i + x_i \bar{y}_i c_i + x_i y_i \bar{c}_i + x_i y_i c_i$$
$$= (\bar{x}_i y_i c_i + x_i y_i c_i) + (x_i \bar{y}_i c_i + x_i y_i c_i) + (x_i y_i \bar{c}_i + x_i y_i c_i)$$
$$= y_i c_i + x_i c_i + x_i y_i$$

The logic diagrams for implementing these equations are given in Figure 7.10.

As has been made apparent by Figure 7.10, for generating C_{i+1} from c_i, two gate delays are required. To generate S_i from c_i, three gate delays are required because c_i must be inverted to obtain $\bar{c}_i$. Note that no inverters are required to get $\bar{x}_i$ or $\bar{y}_i$ from x_i or y_i, respectively, because the numbers to be added are usually stored in a register that is a collection of flip-flops. The flip-flop generates both normal and complemented outputs.



(a) Block Diagram of a 16 × 16 Barrel Shifter

Shift Count				Output															
S_3	S_2	S_1	S_0	y_{15}	y_{14}	y_{13}	y_{12}	y_{11}	y_{10}	y_9	y_8	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
0	0	0	0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
0	0	0	1	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}
0	0	1	0	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}
0	0	1	1	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}
0	1	0	0	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}
0	1	0	1	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}
0	1	1	0	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}
0	1	1	1	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9
1	0	0	0	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8
1	0	0	1	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7
1	0	1	0	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6
1	0	1	1	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5
1	1	0	0	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4
1	1	0	1	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3
1	1	1	0	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2
1	1	1	1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1

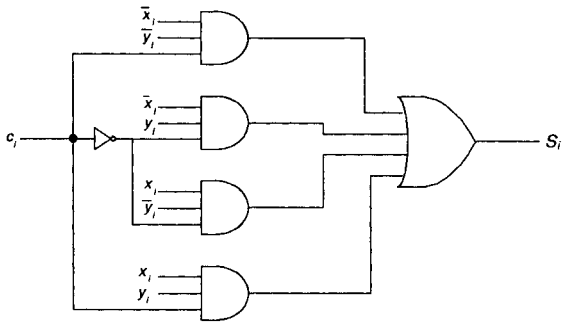
(b) Truth Table of the 16 × 16 Barrel Shifter

FIGURE 7.9 Barrel shifter

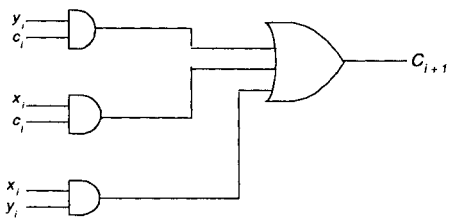
For the purpose of discussion, assume that the gate delay is Δ time units, and the actual value of Δ is decided by the technology. For example, if transistor translator logic (TTL) circuits are used, the value of Δ will be 10 ns.

By cascading n full adders, an n-bit binary adder capable of handling two n-bit operands (X and Y) can be designed. The implementation of a 4-bit ripple-carry or binary adder is shown in Figure 7.11. When two unsigned integers are added, the input carry, c_0, is always zero. The 4-bit adder is also called a "carry-propagate adder" (CPA), because the carry is propagated serially through each full adder. This hardware can be cascaded to obtain a 16-bit CPA, as shown in Figure 7.12; $c_0 = 0$ or 1 for multiprecision addition.

Although the design of an n-bit CPA is straightforward, the carry propagation time limits the speed of operation. For example, in the 16-bit CPA (see Figure 7.12), the

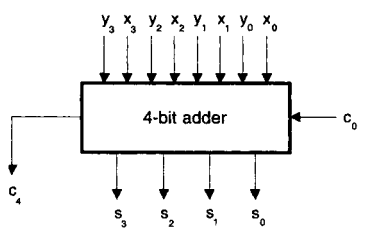


(a) Full adder

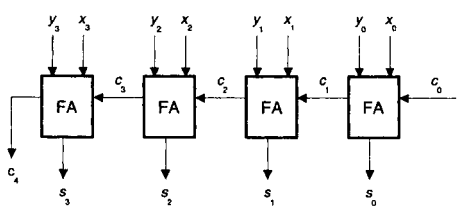


(b) Carry

FIGURE 7.10 Logic circuit of full adder



(a) Block Diagram of a 4-bit Ripple-Carry Adder



(b) Four 4-bit Full Adders are Cascaded to implement a 4-Bit Ripple-Carry Adder

FIGURE 7.11 Implementation of a 4-bit Ripple-Carry Adder

addition operation is completed only when the sum bits s_0 through s_{15} are available.

To generate s_{15}, c_{15} must be available. The generation of c_{15} depends on the availability of c_{14}, which must wait for c_{13} to become available. In the worst case, the carry process propagates through 15 full adders. Therefore, the worst-case add-time of the 16-bit CPA can be estimated as follows:

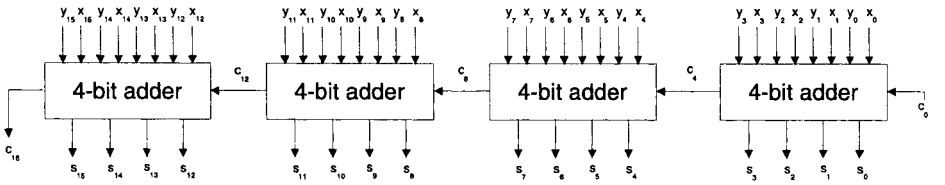


FIGURE 7.12 Implementation of a 16-bit adder using 4-Bit Adders as Building Blocks

Time taken for carry to propagate through 15 full adders (the delay involved in the path from c_0 to c_{15})	$= 15 * 2\ \Delta$
Time taken to generate s_{15} from c_{15}	$= 3\ \Delta$
Total	$= 33\ \Delta$

If $\Delta = 10$ ns, then the worst-case add-time of a 16-bit CPA is 330 ns. This delay is prohibitive for high-speed systems, in which the expected add-time is typically less than 100 ns, which makes it necessary to devise a new technique to increase the speed of operation by a factor of 3. One such technique is known as the "carry look-ahead." In this approach the extra hardware is used to generate each carry (c_i, $i > 0$) directly from c_0. To be more practical, consider the design of a 4-bit carry look-ahead adder (CLA). Let us see how this may be used to obtain a 16-bit adder that operates at a speed higher than the 16-bit CPA.

Recall that in a full adder for adding X_i, Y_i, and C_i, the output carry C_{i+1} is related to its carry input C_i, as follows:

$$C_{i+1} = X_iY_i + X_iC_i + Y_iC_i$$

The result can be rewritten as

$$C_{i+1} = G_i + P_iC_i$$

where $G_i = X_iY_i$ and $P_i = X_i + Y_i$

The function G_i is called the carry-generate function, because a carry is generated when $X_i = Y_i = 1$. If X_i or Y_i is a 1, then the input carry C_i is propagated to the next stage. For this reason, the function P_i is often referred to as the "carry-propagate" function. Using G_i and P_i, C_1, C_2, C_3, and C_4 can be expressed as follows:

$$C_1 = G_0 + P_0C_0$$
$$C_2 = G_1 + P_1C_1$$
$$C_3 = G_2 + P_2C_2$$
$$C_4 = G_3 + P_3C_3$$

All high-order carries can be generated in terms of C_0 as follows:

$$C_1 = G_0 + P_0C_0$$
$$C_2 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0)$$
$$C_3 = G_2 + P_2C_2 = G_2 + P_2(G_1 + P_1G_0 + P_1P_0C_0)$$
$$= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$$
$$C_4 = G_3 + P_3C_3 = G_3 + P_3(G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0)$$
$$= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$$

$g_0 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$

$p_0 = P_3P_2P_1P_0$

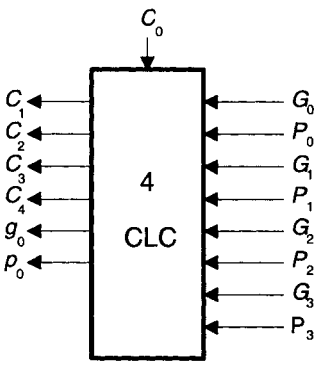


FIGURE 7.13 A Four-Stage Carry Look-ahead Circuit

Therefore C_1, C_2, C_3, and C_4 can be generated directly from C_0. For this reason, these equations are called "carry look-ahead equations," and the hardware that implements these equations is called a "4-stage look-ahead circuit" (4-CLC). The block diagram of such circuit is shown in Figure 7.13.

The following are some important points about this system:

- A 4-CLC can be implemented as a two-level AND-OR logic circuit (The first level consists of AND gates, whereas the second level includes OR gates).
- The outputs g_0 and p_0 are useful to obtain a higher-order look-ahead system.

To construct a 4-bit CLA, assume the existence of the basic adder cell shown in Figure 7.14. Using this basic cell and 4-bit CLC, the design of a 4-bit CLA can be completed as shown in Figure 7.15. Using this cell as a building block, a 16-bit adder can be designed as shown in Figure 7.16.

The worst-case add-time of this adder can be calculated as follows:

		Delay
For P_i, G_i generation		
from X_i, Y_i $(0 \le i \le 15)$	...	Δ
To generate C_4 from C_0	...	2Δ
To generate C_8 from C_4	...	2Δ
To generate C_{12} from C_8	...	2Δ
To generate C_{15} from C_{12}	...	2Δ
To generate S_{15} from C_{15}	...	3Δ
Total delay	...	12Δ

A graphical illustration of this calculation can be shown as follows:

$$\text{Data available} \xrightarrow{\Delta} G_iP_i \xrightarrow{2\Delta} C_4 \xrightarrow{2\Delta} C_8 \xrightarrow{2\Delta} C_{12} \xrightarrow{2\Delta} C_{15} \xrightarrow{3\Delta} S_{15}$$

From this calculation, it is apparent that the new 16-bit adder is faster than the 16-bit CPA by a factor of 3. In fact, this system can be speeded up further by employing another 4-bit CLC and eliminating the carry propagation between the 4-bit CLA blocks. For this purpose, the g_i and p_i outputs generated by the 4-bit CLA are used. This design task is left as an exercise to the reader.

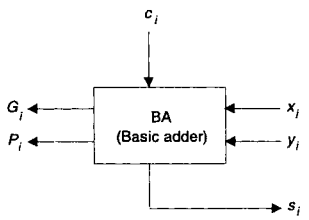


FIGURE 7.14 Basic CLA cell

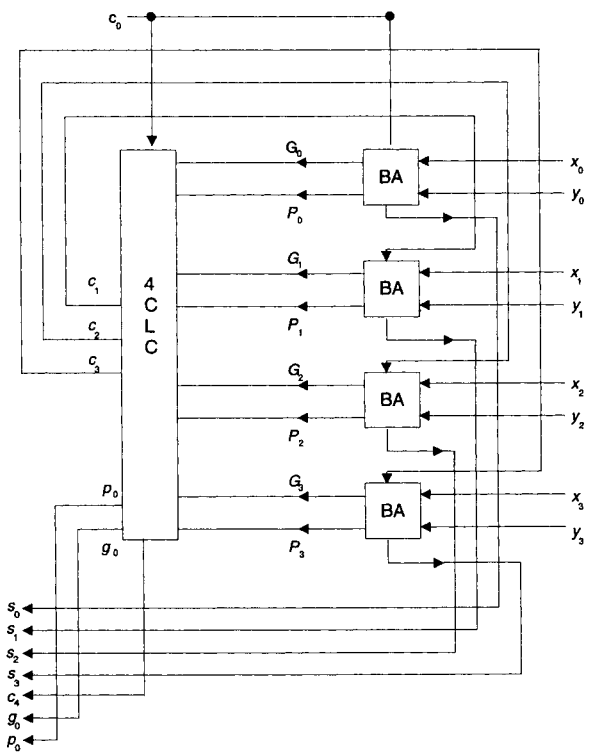


FIGURE 7.15 A 4-bit CLA

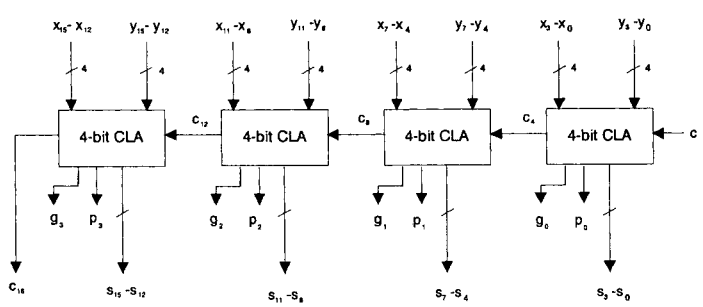


FIGURE 7.16 Design of a 16-bit adder using 4-bit CLAs

If there is a need to add more than 3 operands, a technique known as "carry-save addition" is used. To see its effectiveness, consider the following example:

44
28
32
79
63←Sum vector
12 ←Carry vector
183←Final answer

In this example, four decimal numbers are added. First, the unit digits are added, producing a sum of 3 and a carry digit of 2. Similarly, the tens digits are added, producing a sum digit of 6 and a carry digit of 1. Because there is no carry propagation from the unit digit to the tenth digit, these summations can be carried out in parallel to produce a sum vector of 63 and a carry vector of 12. When all operands are exhausted, the sum and the shifted carry vector are added in the conventional manner, which produces the final answer. Note that the carry is propagated only in the last step, which generates the final answer no matter how many operands are added. The concept is also referred to as "addition by deferred carry assimilation."

7.3.3 Addition, Subtraction, Multiplication and Division of unsigned and signed numbers

The procedure for addition and subtraction of two's complement signed binary numbers is straightforward. The procedure for adding unsigned numbers is discussed in Chapter 2. Also, addition of two 2's complement signed numbers was included in Chapter 2. Note that binary numbers represented in two's complement form contain both unsigned numbers (Most Significant Bit = 0) and signed numbers (Most Significant Bit = 1). The procedure for adding two 2's complement signed numbers using pencil and paper is provided below:

Add the two numbers along with the sign bits. Check the overflow bit (V) using $V = C_f \oplus C_p$ where C_f is the final carry and C_p is the previous carry. If V = 0, then the result of addition is correct. On the other hand, if V = 1, then the result is incorrect; one needs to increase the number of bits for each number, and repeat the addition operation until V = 0 to obtain the correct result.

Subtraction of two 2's complement signed binary numbers using pencil and paper can be performed as follows:

Take the 2's complement of subtrahend along with the sign bit and add it to the minuend . The result is correct if there is no overflow. The result is wrong if there is an overflow. In case of overflow, increase the number of bits for each number, repeat the subtraction operation until the overflow is zero to obtain the correct result. Note that if there is a final carry after performing the 2's complement subtraction, the result is positive. On the other hand, if there is no final carry after 2's complement subtraction, the result is negative.

Computers utilize common hardware to perform addition and subtraction operations for both unsigned and signed numbers. The instruction set of computers typically include the same ADD and SUBTRACT instructions for both unsigned and signed numbers. The interpretations of unsigned and signed ADD and SUBTRACT operations are performed by the programmer. For example, consider adding two 8-bit numbers, A and B ($A = FF_{16}$ and $B = FF_{16}$) using the ADD instruction by a computer as follows:

```
                1111 111← Intermediate carries
    FF16 = 1111 1111
+   FF16 = 1111 1111
    ---------------------------
Final carry →1  1111 1110 = FE16
```

When the above addition is interpreted as an unsigned operation by the programmer, the result will be
$A + B = FF_{16} + FF_{16} = 255_{10} + 255_{10} = 510_{10}$ which is FE_{16} with a carry as shown above. However, if the addition is interpreted as a signed operation, then, $A + B = FF_{16} + FF_{16} = (-1_{10}) + (-1_{10}) = -2_{10}$ which is FE_{16} as shown above, and the final carry must be discarded by the programmer. Similarly, the unsigned and signed subtraction can be interpreted by the programmer.

Typical 8-bit microprocessors, such as the Intel 8085 and Motorola 6809, do not include multiplication and division instructions due to limitations in the circuit densities that can be placed on the chip. Due to advances in semiconductor technology, 16-, 32-, and 64-bit microprocessors usually include multiplication and division algorithms in a ROM inside the chip. These algorithms typically utilize an ALU to carry out the operations. one can write a program that multiplies two numbers. Although this solution seems viable, the operational speed is unsatisfactory.

For application environments such as real-time digital filtering, in which the processor is expected to perform 32 to 64 eight-bit multiplication operations within 100 μsec (sampling frequency = 10 kHz), speed is an important factor. New device technologies such as BICMOS and HCMOS, allow manufacturers to pack millions of transistors in a chip. Consequently, state-of-the-art 32-bit microprocessors such as the Motorola 68060 (HCMOS) and Intel Pentium (BICMOS) designed using these technologies, have a larger instruction set than their predecessors, which includes multiplication and division instructions. In this section, multiplier design principles are discussed. Two unsigned integers can be multiplied using repeated addition as mentioned in Chapter 2. Also, they can be multiplied in the same way as two decimal numbers are multiplied by paper and pencil method. Consider the multiplication of two unsigned integers, where the multiplier Q = 15 and the multiplicand is M = 14, as illustrated:

```
M ——————→ 1110 ——— Multiplicand (14₁₀)
Q ——————→ 1111 ——— Multiplier (15₁₀)

          ————————————
           1110 ←——
           1110 ←——
           1110 ←——— Partial products
          1110 ←———
          ————————————
P ——————→ 11010010 ←——— Final product
```

In the paper and pencil algorithm, shifted versions of multiplicands are added.

$$
\begin{array}{cccc}
 & & m_3 & m_2 & m_1 & m_0 \\
 & & q_3 & q_2 & q_1 & q_0 \\
\hline
 & & m_3q_0 & m_2q_0 & m_1q_0 & m_0q_0 \leftarrow \text{Partial product } PR_0 \\
 & m_3q_1 & m_2q_1 & m_1q_1 & m_0q_1 \leftarrow \text{Partial product } PR_1 \\
 & m_3q_2 & m_2q_2 & m_1q_2 & m_0q_2 \leftarrow \text{Partial product } PR_2 \\
m_3q_3 & m_2q_3 & m_1q_3 & m_0q_3 \leftarrow \text{Partial product } PR_3 \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$

FIGURE 7.17 Generalized Version of the Multiplication of Two 4-bit Numbers Using the Paper and Pencil Algorithm

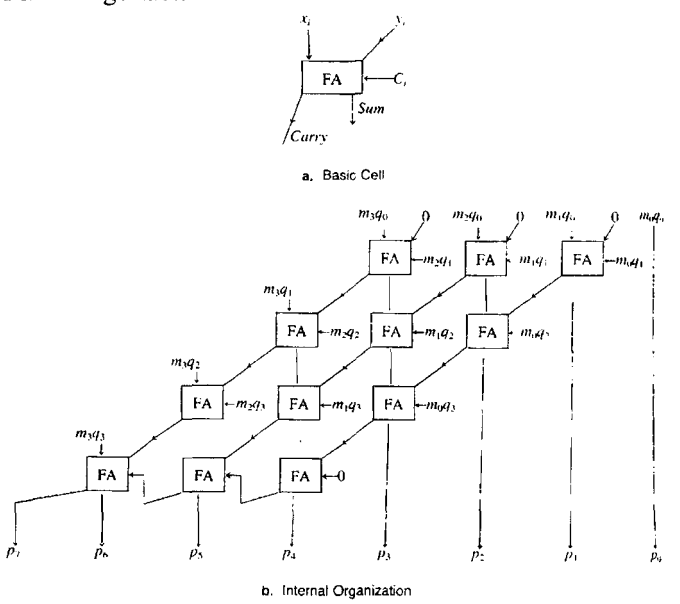


FIGURE 7.18 4 ×4 Array Multiplier

This procedure can be implemented by using combinational circuit elements such as AND gates and FULL adders. Generally, a 4-bit unsigned multiplier Q and a 4-bit unsigned multiplicand M can be written as M: $m_3\ m_2\ m_1\ m_0$ and Q: $q_3\ q_2\ q_1\ q_0$.The process of generating the partial products and the final product can also be generalized as shown in

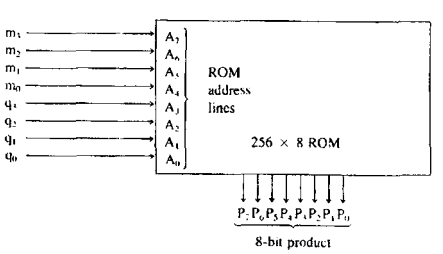


FIGURE 7.19 ROM-based 4x4 Multiplier

Figure 7.17. Each cross-product term ($m_i\, q_j$) in this figure can be generated using an AND gate. This requires 16 AND gates to generate all cross-product terms that are summed by full adder arrays, as shown in Figure 7.18.

Consider the generation of p_2 in Figure 7.18(b). From Figure 7.17, p_2 is the sum of m_2q_0, m_1q_1 and m_0q_2. The sum of these three elements is obtained by using two full adders. (See column for p_2 in Figure 7.18). The top full-adder in this column generates the sum $m_2q_0 + m_1q_1$. This sum is then added to m_0q_2 by the bottom full-adder along with any carry from the previous full-adder for p_1.

The time required to complete the multiplication can be estimated by considering the longest carry propagation path comprising of the rightmost diagonal (which includes the full-adder for p_1 and the bottom full-adders for p_2 and p_3), and the last row (which includes the full-adder for p_6 and the bottom full-adders for p_4 and p_5). The time taken to multiply two n-bit numbers can be expressed as follows:

$$T(n) + \Delta_{AND\ gate} + (n - 1)\,\Delta_{carry\ propagation} + (n - 1)\,\Delta_{carry\ propagation}$$

In this equation, all cross-product terms m_iq_i can be generated simultaneously by an array of AND gates. Therefore, only one AND gate delay is included in the equation. Also, the rightmost diagonal and the bottom row contain (n - 1) full-adders each for the n × n multiplier.

Assuming that $\Delta_{AND\ gate} = \Delta_{carry\ propagation} = 2gate\ delays = 2\Delta$, the preceding expression can be simplified as shown:

$T(n) = 2\Delta + (2n - 2)2\Delta = (4n - 2)\Delta$.

The array multiplier that has been considered so far is known as Braun's multiplier. The hardware is often called a nonadditive multiplier (NM), since it does not include any additive inputs. An additive multiplier (AM) includes an extra input R; it computes products of the form

P = M * Q + R

This type of multiplier is useful in computing the sum of products of the form $\sum XiYi$.

Both an NM and an AM are available as standard 1C blocks. Since these systems require more components, they are available only to handle 4- or 8-bit operands.

Alternatively, the same 4x4 NM discussed earlier can be obtained using a 256 × 8 ROM as shown in Figure 7.19.

It can be seen that a given MQ pair defines a ROM address, where the corresponding 8-bit product is held. The ROM approach can be used for small-scale multipliers because:

- The technological advancements allow the manufacturers to produce low-cost ROMs.
- The design effort is minimum.

In case of large multipliers, ROM implementation is unfeasible, since large-size ROMs are required. For example, in order to implement an 8 × 8 multiplier, a 2^{16} × 16 ROM is required. If the required 8 × 8 product is decomposed into a linear combination of four 4x4 products, an 8 × 8 multiplier can be implemented using four 256 × 8 ROMs and a few 4-bit parallel adders. However, PLDs can be used to accomplish this.

Signed multiplication can be performed using various algorithms. A simple algorithm follows.

In the case of signed numbers, there are three possibilities:

1. M and Q are in sign-magnitude form.
2. M and Q are in ones complement form.
3. M and Q are in twos complement form.

For the first case, perform unsigned multiplication of the magnitudes without the sign

bits. The sign bit of the product is determined as $M_n \oplus Q_n$, where M_n and Q_n are the most significant bits (sign bits) of the multiplicand (M) and the multiplier (Q), respectively. For the second case, proceed as follows:

Step 1: If $M_n = 1$, then compute the ones complement of M.
Step 2: If $Q_n = 1$, then compute the ones complement of Q.
Step 3: Multiply the $n - 1$ bits of the multiplier and the multiplicand.
Step 4: $S_n = M_n \oplus Q_n$
Step 5: If $S_n = 1$, then compute the ones complement of the result obtained in Step 3.

Whenever the ones complement of a negative number (sign bit = 1) is taken, the sign is reversed. Hence, with respect to the multiplier, the inputs are always a positive quantity. When the sign of the bit is negative, however ($M_n \oplus Q_n = 1$), the result must be presented in the ones complement form. This is why the ones complement of the product found by the unsigned multiplier is computed. When M and Q are in twos complement form, the same procedure is repeated, with the exception that the twos complement must be determined when $Q_n = 1$, $M_n = 1$, or $M_n \oplus Q_n = 1$. Consider M and Q as twos complement numbers. Suppose $M = 1100_2$ and $Q = 0111_2$. Because $M_n = 1$, take the twos complement of $M = 0100_2$; because $Q_n = 0$, do not change Q. Multiply 0111_2 and 0100_2 using the unsigned multiplication method discussed before. The product is 00011100_2. The sign of the product $S_n = M_n \oplus Q_n = 1 \oplus 0 = 1$. Hence, take the twos complement of the product 00011100_2 to obtain 11100100_2, which is the final answer: -28_{10}.

As mentioned in Chapter 2, unsigned division can be performed using repeated subtraction. However, the general equation for division can be used for signed division. Note that the general equation for division is *Dividend = Quotient * Divisor + Remainder*. For example, consider dividend = – 9, divisor = 2. Three possible solutions are shown below:

(a) $-9 = -4 * 2 - 1$, Quotient = – 4, Remainder = – 1.
(b) $-9 = -5 * 2 + 1$, Quotient = – 5, Remainder = +1.
(c) $-9 = -6 * 2 + 3$, Quotient = – 6, Remainder = +3.

However, the correct answer is shown in (a) in which, Quotient = – 4 and Remainder = – 1. Hence, for signed division, the sign of the remainder is the same as the sign of the dividend, unless the remainder is zero. Typical microprocessors such as Motorola 68XXX follow this convention.

7.3.4 ALU Design

Functionally, an ALU can be divided up into two segments: the arithmetic unit and the logic unit. The arithmetic unit performs typical arithmetic operations such as addition, subtraction, and increment or decrement by 1. Usually, the operands involved may be signed or unsigned integers. In some cases, however, an arithmetic unit must handle 4-bit binary-coded decimal (BCD) numbers and floating-point numbers. Therefore, this unit must include the circuitry necessary to manipulate these data types. As the name implies, the logic unit contains hardware elements that perform typical operations such as Boolean NOT and OR. In this section, the design of a simple ALU using typical combinational elements such as gates, multiplexers, and a 4-bit parallel adder is discussed. For this approach, an arithmetic unit and a logic unit are first designed separately; then they are combined to obtain an ALU.

For the first step, a two-function arithmetic unit, as shown in Figure 7.20 is designed. The key element of this system is a 4-bit parallel adder. The multiplexers select

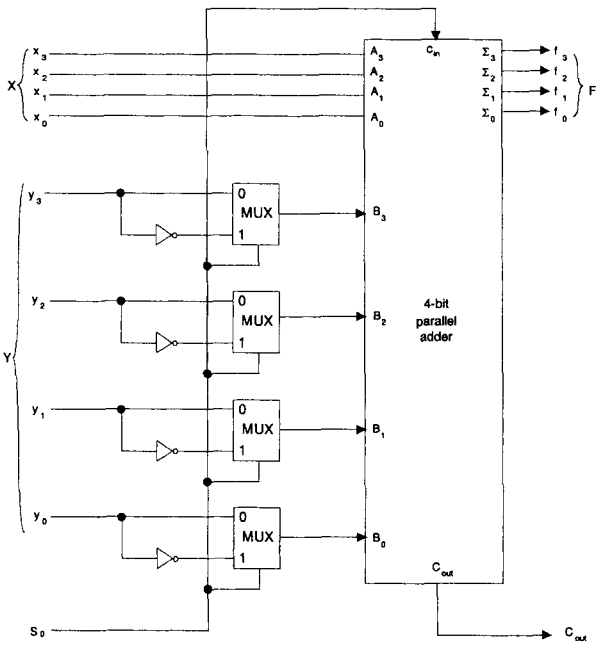


FIGURE 7.20 Organization of an arithmetic unit

either Y or $\overline{Y}$ for the 3-input of the parallel adder. In particular, if $s_0 = 0$, then $B = Y$; otherwise $B = \overline{Y}$. Because the selection input (s_0) also controls the input carry (c_{in}), the following results:

$$\begin{aligned} \text{If } s_0 = 0 \text{ then } F &= X \text{ plus } Y \\ \text{else } F &= X \text{ plus } \overline{Y} \text{ plus } 1 \\ &= X \text{ minus } Y \end{aligned}$$

This arithmetic unit generates addition and subtraction operations. For the second step, let us design a two-function logic unit; this is shown in Figure 7.21. From Figure 7.21 it can be seen that when $s_0 = 0$, the output $G = X$ AND Y; otherwise the output $G = X \oplus Y$. Note that from these two Boolean operations, other operations such as NOT and OR can be derived by the following Boolean identities:

$1 \oplus x = \overline{x}$

x OR $y = x \oplus y \oplus xy$

Therefore, NOT and OR operations can be obtained by using additional hardware and the circuit of Figure 7.21. The outputs generated by the arithmetic and logic units can be combined by using a set of multiplexers, as shown in Figure 7.22. From this organization it can be seen that when the select line $s_1 = 1$, the multiplexers select outputs generated by the logic unit; otherwise, the outputs of the arithmetic unit are selected.

More commonly, the select line, s_1, is referred to as the *mode input* because it selects the desired mode of operation (arithmetic or logic). A complete block diagram schematic of this ALU is shown in Figure 7.23. The truth table illustrating the operation of this ALU is shown in Figure 7.24. This table shows that this ALU is capable of performing 2 arithmetic and 2 logic operations on the 4-bit operands X and Y.

The rapid growth in IC technology permitted the manufacturers to produce an ALU as an MSI block. Such systems implement many operations, and their use as a system

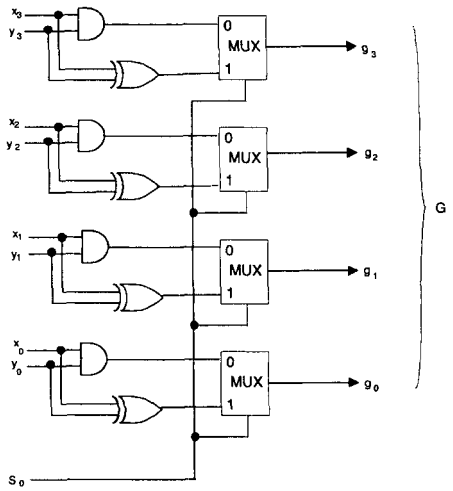


FIGURE 7.21 Organization of a 4-bit two-function logic unit

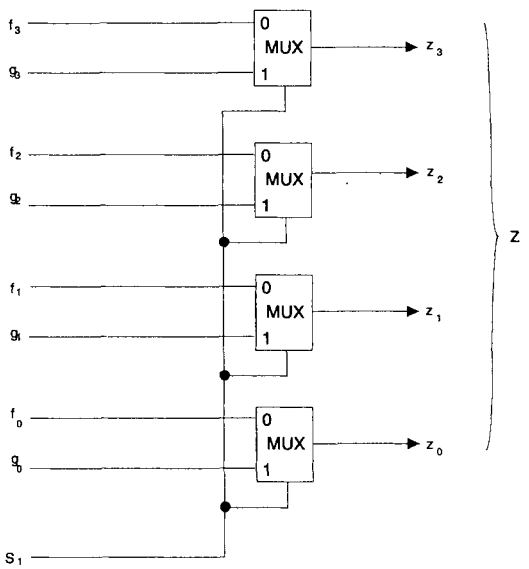


FIGURE 7.22 Combining the outputs generated by the arithmetic and logic units

component reduces the hardware cost, board space, debugging effort, and failure rate. Usually, each MSI ALU chip is designed as a 4-bit slice. However, a designer can easily interconnect n such chips to get a $4n$-bit ALU. Some popular 4-bit ALU chips are the 74381 and 74181. The 74381 ALU performs 3 arithmetic and 2 miscellaneous operations on 4-bit operands. The 74181 ALU performs 16 arithmetic and 16 Boolean operations on two 4-bit operands, using either active high or active low data. A complete description and operational characteristics of these devices may be found in the data books.

Typical 8-bit microprocessors, such as the Intel 8085 and Motorola 6809, do not include multiplication and division instructions due to limitations in the circuit densities that can be placed on the chip. Due to advanced semiconductor technology, 16-, 32-, and 64-bit

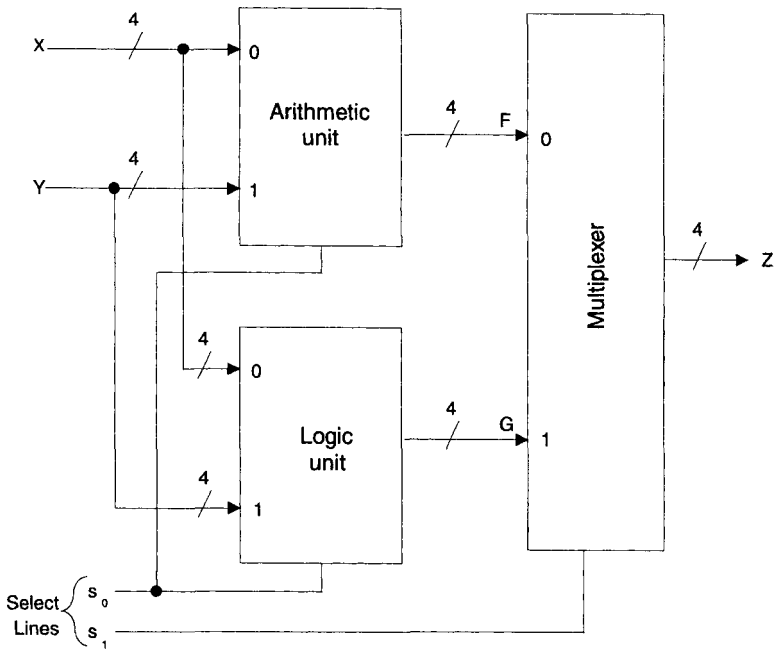


FIGURE 7.23 Schematic representation of the four functions

Select Lines		Output Z	Comment
s_1	s_0		
0	0	X plus Y	Addition
0	1	X plus $\overline{Y}$ Plus 1	2's Complement subtraction
1	0	X ∧ Y	Boolean AND
1	1	X ⊕ Y	Exclusive-OR

FIGURE 7.24 Truth table controlling the operations of the ALU of Figure 7.23

microprocessors usually include multiplication and division algorithms in a ROM inside the chip. These algorithms typically utilize an ALU to carry out the operations. Verilog and VHDL descriptions along with simulation results of typical ALU's are included in Appendices I and J respectively.

7.3.5 Design of the Control Unit

The main purpose of the control unit is to translate or decode instructions and generate appropriate enable signals to accomplish the desired operation. Based on the contents of the instruction register, the control unit sends the selected data items to the appropriate processing hardware at the right time. The control unit drives the associated processing hardware by generating a set of signals that are synchronized with a master clock.

The control unit performs two basic operations: instruction interpretation and instruction sequencing. In the interpretation phase, the control unit reads (fetches) an instruction from the memory addressed by the contents of the program counter into

the instruction register. The control unit inputs the contents of the instruction register. It recognizes the instruction type, obtains the necessary operands, and routes them to the appropriate functional units of the execution unit (registers and ALU). The control unit then issues the necessary signals to the execution unit to perform the desired operation and routes the results to the specified destination.

In the sequencing phase, the control unit generates the address of the next instruction to be executed and loads it into the program counter. To design a control unit, one must be familiar with some basic concepts such as register transfer operations, types of bus structures inside the control unit, and generation of timing signals. These are described in the next section.

There are two methods for designing a control unit: hardwired control and microprogrammed control. In the hardwired approach, synchronous sequential circuit design procedures are used in designing the control unit. Note that a control unit is a clocked sequential circuit. The name "hardwired control" evolved from the fact that the final circuit is built by physically connecting the components such as gates and flip-flops. In the microprogrammed approach, on the other hand, all control functions are stored in a ROM inside the control unit. This memory is called the "control memory." RAMs and PALs are also used to implement the control memory. The words in this memory are called "control words," and they specify the control functions to be performed by the control unit. The control words are fetched from the control memory and the bits are routed to appropriate functional units to enable various gates. An instruction is thus executed. Design of control units using microprogramming (sometimes called *firmware* to distinguish it from hardwired control) is more expensive than using hardwired controls. To execute an instruction, the contents of the control memory in microprogrammed control must be read, which reduces the overall speed of the control unit.The most important advantage of microprogramming is its flexibility; 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.

There are two types of microprocessor architectures: CISC (Complex Instruction Set Computer) and RISC (Reduced Instruction Set Computer). CISC microprocessors contain a large number of instructions and many addressing modes while RISC microprocessors include a simple instruction set with a few addressing modes. Almost all computations can be obtained from a few simple operations. RISC basically supports a small set of commonly used instructions which are executed at a fast clock rate compared to CISC which contains a large instruction set (some of which are rarely used) executed at a slower clock rate. In order to implement fetch /execute cycle for supporting a large instruction set for CISC, the clock is typically slower. In CISC, most instructions can access memory while RISC contains mostly load/store instructions. The complex instruction set of CISC requires a complex control unit, thus requiring microprogrammed implementation. RISC utilizes hardwired control which is faster. CISC is more difficult to pipeline while RISC provides more efficient pipelining. An advantage of CISC over RISC is that complex programs require fewer instructions in CISC with a fewer fetch cycles while the RISC requires a large number of instructions to accomplish the same task with several fetch cycles. However, RISC can significantly improve its performance with a faster clock, more efficient pipelining and compiler optimization. PowerPC and Intel 80XXX utilize RISC and CISC architectures respectively. Intel Pentium family, on the other hand, utilizes a combination of RISC and CISC architectures for providing high performance. The Pentium uses RISC (hardwired control) to implement efficient pipelining for simple

$$R_1 \leftarrow R_0$$

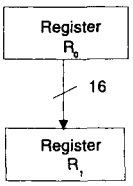


FIGURE 7.25 16-Bit register transfer from R_0 to R_1

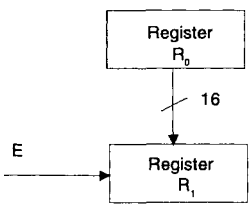


FIGURE 7.26 An enable input controlling register transfer

instructions. CISC (microprogrammed control) for complex instructions is utilized by the Pentium to provide upward compatibility with the Intel 8086/80X86 family.

Basic Concepts

Register transfer notation is the fundamental concept associated with the control unit design. For example, consider the register transfer operation of Figure 7.25. The contents of 16-bit register R_0 are transferred to 16-bit register R_1 as described by the following notation:

$$R_1 \leftarrow R_0$$

The symbol $\leftarrow$ is called the transfer operator. However, this notation does not indicate the number of bits to be transferred. A declaration statement specifying the size of each register is used for the purpose:

```
Declare registers R0 [16], R1 [16]
```

The register transfer notation can also be used to move a specific bit from one register to a particular bit position in another. For example, the statement

$$R_1[1] \leftarrow R_0[14]$$

means that bit 14 of register R_0 is moved to bit 1 of register R_1.

An enable signal usually controls transfer of data from one register to another. For example, consider Figure 7.26. In the figure, the 16-bit contents of register R_0 are transferred to register R_1 if the enable input E is HIGH; otherwise the contents of R_0 and R_1 remain the same. Such a conditional transfer can be represented as

$$E\colon R_1 \leftarrow R_0$$

Figure 7.27 shows a hardware implementation of transfer of each bit of R_0 and R_1. The enable input may sometimes be a function of more than one variable. For example, consider the following statement involving three 16-bit registers: If $R_0 < R_1$ and $R_2[1] = 1$ then $R_1 \leftarrow R_0$.

The condition $R_0 < R_1$ can be determined by an 8-bit comparator such that the output y of the comparator goes to 0 if $R_0 < R_1$. The conditional transfer can then be

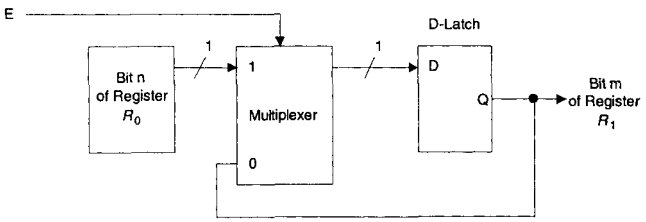


FIGURE 7.27 Hardware for each bit transfer from R_0 to R_1

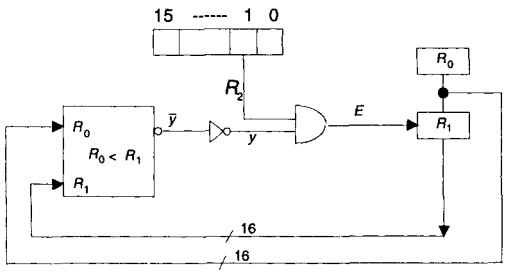


FIGURE 7.28 Hardware implementation E: $R_1 \leftarrow R_0$ where $E = y \cdot R_2$ [1]

```
Declare registers R[8],M[8],Q[8];
Declare buses inbus[8],outbus[8];
Start:    R ← 0, M ← inbus;          Clear register R to 0 and move
                                     multiplicand
          Q ← inbus;                 Transfer multiplier
Loop:     R ← R + M, Q ← Q-1;        Add multiplicand
          If Q < > 0 then go to loop; repeat if Q≠ 0
          Outbus ← R;
  Halt:   Go to Halt;
```

FIGURE 7.29 Register transfer description of 8 × 8 unsigned multiplication (Assume 8-bit result)

expressed as follows: E: $R_1 \leftarrow R_0$ where $E = y \cdot R_2$ [1]. Figure 7.28 depicts the hardware implementation.

A number of wires called "buses" are normally used to transfer data in and out of a digital processing system. Typically, there will be a pair of buses ("inbuses" and "outbuses") inside the CPU to transfer data from the external devises into the processing section and vice versa. Like the registers, these buses are also represented using register transfer notations and declaration statements. For example, "Declare inbus [16] and outbus [16]" indicate that the digital system contains two 16-bit wide data buses (inbus and outbus). $R_0 \leftarrow$ inbus means that the data on the inbus is transferred into register R_0 when the next clock arrives. An equate (=) symbol can also be used in place of $\leftarrow$. For example, "outbus = R_1 [15:8]" means that the high-order 8 bits of the 16-bit register R_1 are made available on the outbus for one clock period. An algorithm implemented by a digital system can be described by using a set of register transfer notations and typical control structures such as if-then and go to. For example, consider the description shown in Figure 7.29 for

multiplying two 8-bit unsigned numbers (Multiplication of an 8-bit unsigned multiplier by an 8-bit multiplicand) using repeated addition.

The hardware components for the preceding description include an 8-bit inbus, an 8-bit outbus, an 8-bit parallel adder, and three 8-bit registers, R, M, and Q. This hardware performs unsigned multiplication by repeated addition. This is equivalent to unsigned multiplication performed by assembly language instruction.

A distinguishing feature of this description is to describe concurrent operations. For example, the operations $R \leftarrow 0$ and $M \leftarrow$ inbus can be performed simultaneously. As a general rule, a comma is inserted between operations that can be executed concurrently. On the other hand, a semicolon between two transfer operations indicates that they must be performed serially. This restriction is primarily due to the data path provided in the hardware. For example, in the description, because there is only one input bus, the operations $M \leftarrow$ inbus and $Q \leftarrow$ inbus cannot be performed simultaneously. Rather, these two operations must be carried out serially. However, one of these operations may be overlapped with the operation $R \leftarrow 0$ because the operation does not use the inbus. The description also includes labels and comments to improve readability of the task description. Operations such as $R \leftarrow 0$ and M $\leftarrow$ inbus are called "micro-operations", because they can be completed in one clock cycle. In general, a computer instruction can be expressed as a sequence of micro-operations.

The rate at which a microprocessor completes operations such as $R \leftarrow R + M$ is determined by its bus structure inside the microprocessor chip. The cost of the microprocessor increases with the complexity of the bus structure. Three types of bus structures are typically used: single-bus, two-bus, and three-bus architectures.

The simplest of all bus structures is the single-bus organization shown in Figure 7.30. At any time, data may be transferred between any two registers or between a register and the ALU. If the ALU requires two operands such as in response to an ADD instruction, the operands can only be transferred one at a time. In single-bus architecture, the bus must be multiplexed among various operands. Also, the ALU must have buffer registers to hold the transferred operand.

In Figure 7.30, an add operation such as $R_0 \leftarrow R_1 + R_2$ is completed in three clock cycles as follows:

First clock cycle: The contents of R_1 are moved to buffer register B_1 of the ALU.
Second clock cycle: The contents of R_2 are moved to buffer register B_2 of the ALU.
Third clock cycle: The sum generated by the ALU is loaded into R_0.

A single-bus structure slows down the speed of instruction execution even though data may already be in the microprocessor registers. The instruction's execution time is longer if the operands are in memory; two clock cycles may be required to retrieve the operands into the microprocessor registers from external memory.

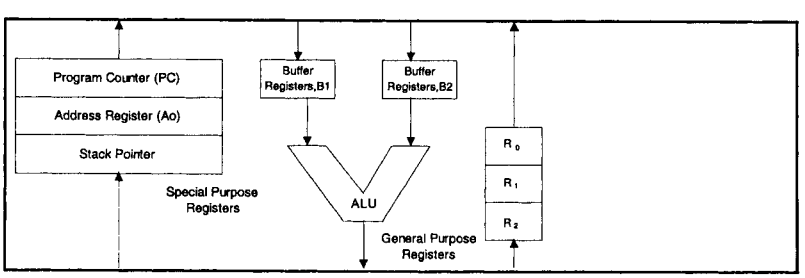


FIGURE 7.30 Single-bus architecture

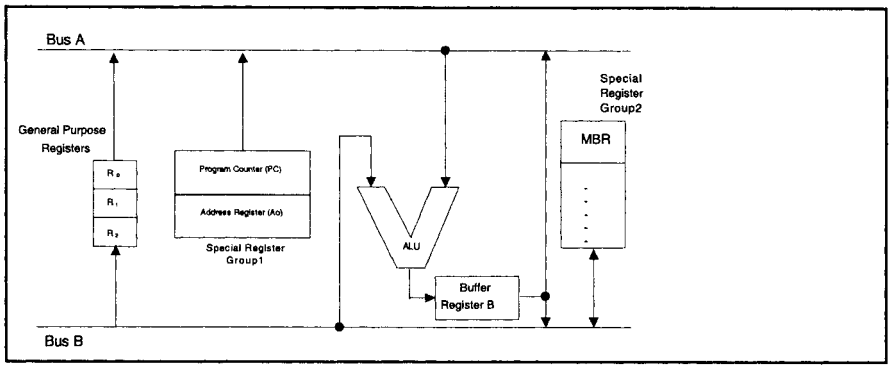


FIGURE 7.31 Two-bus architecture

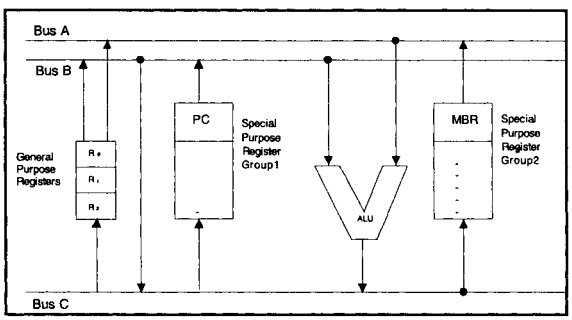


FIGURE 7.32 Three-bus architecture

To execute an instruction such as ADD between two operands already in register, the control logic in a single-bus structure must follow a three-step sequence. Each step represents a control state. Therefore, a single-bus architecture requires a large number of states in the control logic, so more hardware may be needed to design the control unit. Because all data transfers take place through the same bus one at a time, the design effort to build the control logic is greatly reduced.

Next, consider a two-bus architecture, shown in Figure 7.31. All general-purpose registers are connected to both buses (bus *A* and bus *B*) to form a two-bus architecture. The two operands required by the ALU are, therefore, routed in one clock cycle. Instruction execution is faster because the ALU does not have to wait for the second operand, unlike the single-bus architecture. The information on a bus may be from a general-purpose register or a special-purpose register. In this arrangement, special-purpose registers are often divided into two groups. Each group is connected to one of the buses. Data from two special-purpose registers of the same group cannot be transferred to the ALU at the same time.

In the two-bus architecture, the contents of the program counter are always transferred to the right input of the ALU because it is connected to bus *A*. Similarly, the contents of the special register MBR (memory buffer register, to hold up data retrieved from external memory) are always transferred to the left input of the ALU because it is connected to bus *B*.

In Figure 7.31, an add operation such as $R_0 \leftarrow R_1 + R_2$ is completed in two clock cycles as follows:

First clock cycle: The contents of R_1 and R_2 are moved to the inputs of ALU. The ALU then generates the sum in the output register.

Second clock cycle: The sum from the output register is routed to R_0.

The performance of a two-bus architecture can be improved by adding a third bus (bus *C*), at the output of the ALU. Figure 7.32 depicts a typical three-bus architecture. The three-bus architecture perform the addition operation $R_0 \leftarrow R_1 + R_2$ in one cycle as follows:

First cycle: The contents of R_1 and R_2 are moved to the inputs of the ALU via bus A and bus *B* respectively. The sum generated by the ALU is then transferred to R_0 via bus *C*.

The addition of the third bus will increase the system cost and also the complexity of the control unit design.

Note that the bus architectures described so far are inside the microprocessor chip. On the other hand, the system bus connecting the microprocessor, memory, and I/O are external to the microprocessor.

Another important concept required in the design of a control unit is the generation of timing signals. One of the main tasks of a control unit is to properly sequence a set of operations such as a sequence of *n* consecutive clock pulses. To carry out an operation, timing signals are generated from a master clock. Figure 7.33 shows the input clock pulse and the four timing signals T_0, T_1, T_2, and T_3. A ring counter (described in Chapter 5) can be used to generate these timing signals. To carry out an operation P_i at the *i*th clock pulse, a control unit must count the clock pulses and produce a timing signal T_i.

Hardwired Control Design

The steps involved in hardwired control design are summarized as follows:

1. Derive a flowchart from the problem definition and validate the algorithm by using trial data.
2. Obtain a register transfer description of the algorithm from the flowchart.
3. Specify a processing hardware along with various components.
4. Complete the design of the processing section by establishing the necessary control inputs.
5. Determine a block diagram of the controller.

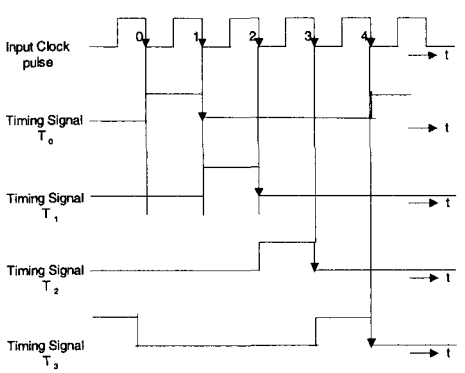


FIGURE 7.33 Timing signals

6. Obtain the state diagram of the controller.
7. Specify the characteristic of the hardware for generating the required timing signals used in the controller.
8. Draw the logic circuit of the controller.

The following example is provided to illustrate the concepts associated with implementation of a typical instruction in a control unit using hardwired control. The unsigned multiplication by repeated addition discussed earlier is used for this purpose. A 4-

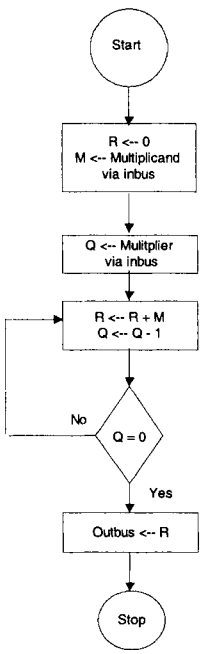


FIGURE 7.34 Flowchart for 4-bit × 4-bit multiplication

	R	M	Q
Initialization	0 0 0 0	0 1 0 0	0 0 1 1
Iteration 1 R <-- R + M Q <-- Q - 1	0 1 0 0	0 1 0 0	0 0 1 0
Iteration 2 R <-- R + M Q <-- Q - 1	1 0 0 0	0 1 0 0	0 0 0 1
Iteration 3 R <-- R + M Q <-- Q - 1	1 1 0 0	0 1 0 0	0 0 0 0

Product $=12_{10}$

FIGURE 7.35 Verification of the unsigned multiplication algorithm

bit by 4-bit unsigned multiplication will be considered. Assume the result of multiplication is 4 bits.

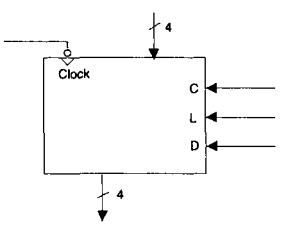
Step 1: Derive a flowchart from the problem definition and then validate the algorithm using trial data.

Figure 7.34 shows the flowchart. In the figure, M and Q are two 4-bit registers containing the unsigned multiplicand and unsigned multiplier respectively. Assume that the result of multiplication is 4-bit wide. The 4-bit result of the multiplication called the "product" will be stored in the 4-bit register, R. The contents of R are then output to the outbus.

The flowchart in Figure 7.34 is similar to an ASM chart and provides a hardware description of the algorithm. The sequence of events and their timing relationships are described in the flowchart. For example, the operations, $R \leftarrow 0$ and $M \leftarrow$ multiplicand shown in the same block are executed simultaneously. Note that $M \leftarrow$ multiplicand via inbus and $Q \leftarrow$ multiplier via inbus must be performed serially because both operations use a single input bus for loading data. These operations are, therefore, shown in different

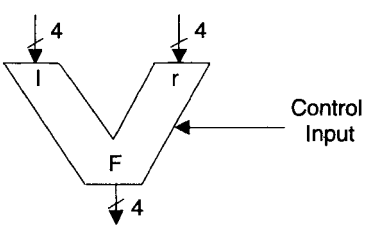
```
Start: R ← 0, M ← inbus;          Clear Register to 0 and move multiplicand
       Q ← inbus;                 Transfer Multiplier
Loop:  R ← R + M, Q ← Q -1;       Perform addition, decrement counter
       If Q < > 0 then goto Loop; Repeat if Q ≠ 0
       outbus ← R;
Halt:  Go to Halt;
```

FIGURE 7.36 Register transfer description 4-bit × 4-bit unsigned multiplication



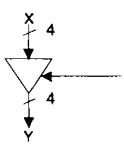
C	L	D	Clock	Action
1	0	0	↓	Clear
0	1	0	↓	Load External data
0	0	1	↓	Decrement by one
0	0	0	↓	No Change

(a) General Purpose Register



Control Input	F
1	I + r
0	No operation

(b) 4-bit Adder



Control Input	Y
1	X
0	High Impedence State

(c) Tristate Buffer

FIGURE 7.37 Components of the processing section of 4-bit by 4-bit unsigned multiplication

blocks. Because $R \leftarrow 0$ does not use the inbus, this operation is overlapped, in our case, with initializing of M via the inbus. This simultaneous operation is indicated by placing them in the same block.

The algorithm will now be verified by means of a numerical example as shown in Figure 7.35. Suppose $M = 0100_2 = 4_{10}$ and $Q = 0011_2 = 3_{10}$; then R = product = $1100_2 = 12_{10}$

Step 2: Obtain a register transfer description of the algorithm from the flowchart. Figure 7.36 shows the description of the algorithm.

Step 3: Specify a processing hardware along with various components.

The processing section contains three main components:

- General-purpose registers
- 4-bit adder
- Tristate buffer

Figure 7.37 shows these components. The general-purpose register is a trailing edge-triggered device.

Three operations (clear, parallel load, and decrement) can be performed by applying the appropriate inputs at C, L, and D. All these operations are synchronized at the trailing (high to low) edge of the clock pulse.

The 4-bit adder can be implemented using 4-bit adder circuits. The tristate buffer is used to control data transfer to the outbus.

Step 4: Complete the design of the processing section by establishing the necessary control inputs.

Figure 7.38 shows the detailed logic diagram of the processing section, along with the control inputs.

Step 5: Determine a block diagram of the controller. Figure 7.39 shows the block diagram.

The controller has three inputs and seven outputs. The Reset input is an asynchronous input used to reset the controller so that 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.

Step 6: Obtain the state diagram of the controller.

The controller must initiate a set of operations in a specified sequence. Therefore, it is modeled as a sequential circuit. The state diagram of the unsigned multiplier controller is shown in Figure 7.40.

Initially, the controller is in state T_0. At this point, the control signals C_0 and C_1 are HIGH. Operations $R \leftarrow 0$ and $M \leftarrow$ inbus are carried out with the trailing edge of the next clock pulse. The controller moves to state T_1 with this clock pulse. When the controller is in T_2, $R \leftarrow R + M$ and $Q \leftarrow Q - 1$ are performed.

All these operations take place at the trailing edge of the next clock pulse. The controller moves to state T_5 only when the unsigned multiplication is completed. The controller then stays in this state forever. A hardware reset input causes the controller to move to state T_0, and a new computation will start.

In this state diagram, selection of states is 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 7.40, operations $R \leftarrow 0$ and $M \leftarrow$ inbus are independent of each other. With this hardware, they can be executed in one clock cycle. That is, they are

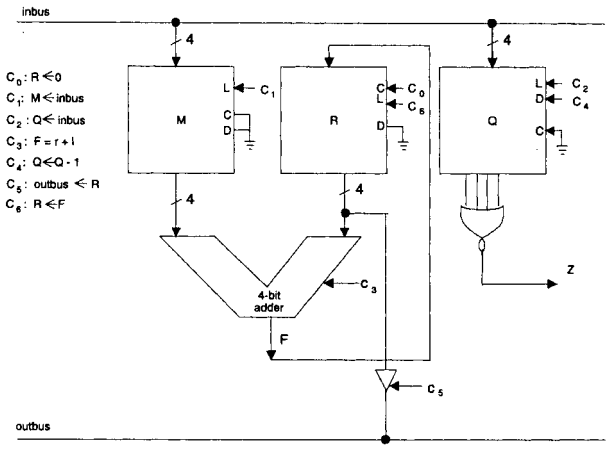


FIGURE 7.38 Detailed logic diagram of the processing section

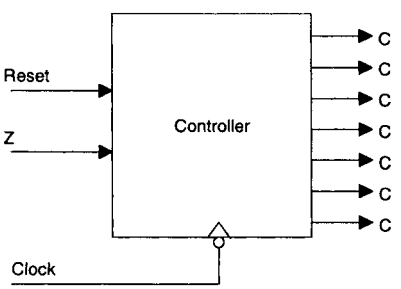
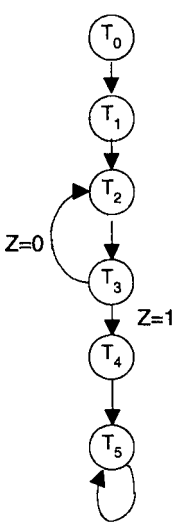


FIGURE 7.39 Block diagram of the unsigned multiplier controller



Control State	Operation Performed	Control Signal to be activated
T_0	R ← 0, M ← inbus	C_0, C_1
T_1	Q ← inbus	C_2
T_2	R ← R + M, Q ← Q - 1	C_3, C_4, C_6
T_3	None	None
T_4	outbus ← R	C_5
T_5	None	None

(a) State Diagram (b) Controller action

FIGURE 7.40 Controller description

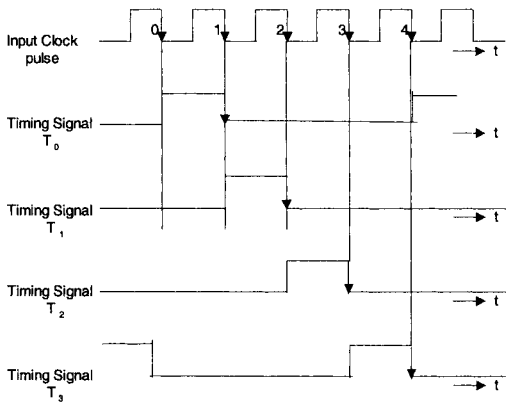


FIGURE 7.41 Timing signals generated by the controller

microoperations. However, if they cannot be completed within the T_0 clock cycle, either clock duration must be increased or the operations should be divided into a sequence of microoperations.

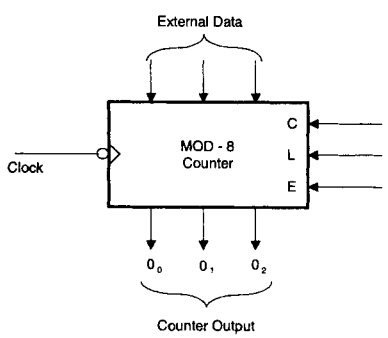
- Conditional testing normally implies the introduction of new states. For example, in the figure, conditional testing of Z introduces the new state T_3.
- One should not attempt to minimize the number of states. When in doubt, new states must be introduced. The correctness of the control logic is more important than the cost of the circuit.

Step 7: Specify the characteristics of the hardware for generating the required timing signals.

There are six states in the controller state diagram. Six nonoverlapping timing signals (T_0 through T_5) must be generated so that only one will be high for a clock pulse. For example, Figure 7.41 shows the four timing signals T_0, T_1, T_2, and T_3. A mod-8 counter and a 3-to-8 decoder can be used to accomplish this task. Figure 7.42 shows the mod-8 counter.

Step 8: Draw the logic circuit of the controller.

Figure 7.43 shows the logic circuit of the controller. The key element of the implementation in Figure 7.43 is the sequence controller (SC) hardware, which sequences



(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

Note: X = don't care

(b) Function Table

FIGURE 7.42 Characteristics of the counter used in the controller design

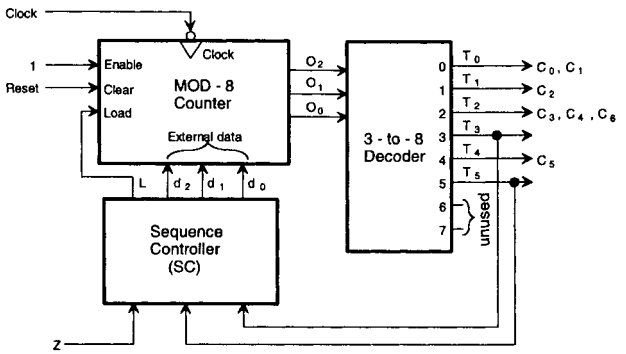
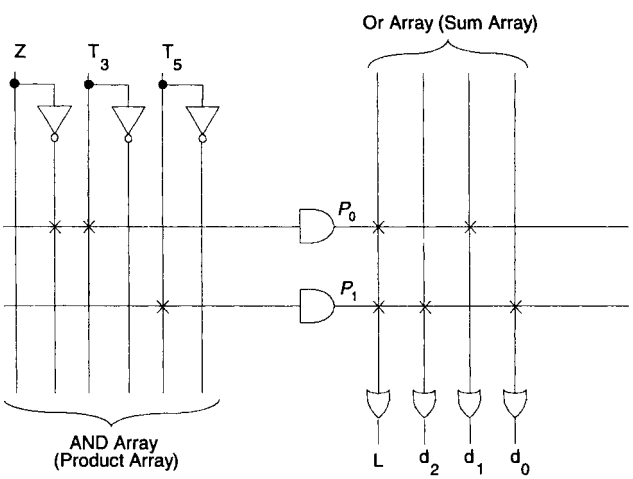


FIGURE 7.43 Logic diagram of the unsigned multiplier controller

Inputs			Outputs			
Z	T_3	T_5	L	d_2	d_1	d_0
0	1	x	1	0	1	0
x	x	1	1	1	0	1

Note: x = don't care

(a) Truth Table



(b) PLA Implementation

FIGURE 7.44 Sequence controller design

the controller according to the state diagram of Figure 7.40. Figure 7.44(a) shows the truth table for the SC controller.

Consider the logic involved in deriving the entries of the SC truth table. The mod-8 counter is loaded (or initialized) with the specified external data if the counter control inputs *C* and *L* are 0 and 1 respectively from Figure 7.42. In this counter, the counter load

control input L overrides the counter enable control input E.

From the controller's state diagram of Figure 7.40, the controller counts up automatically in response to the next clock pulse when the counter load control input L = 0 because the enable input E is tied to HIGH. Such normal sequencing activity is desirable for the following situations:

- Present control state is T_0, T_1, T_2, T_4.
- Present control state is T_3 and $Z = 1$; the next state is T_4.

The SC must load the counter with the appropriate count when the counter is required to load the count out of its normal sequence.

For example, from the controller's state diagram of Figure 7.40, if the present control state is T_3 (counter output $O_2O_1O_0$= 011) and if $Z = 0$, the next state is T_2. When these input conditions occur, the counter must be loaded with external value 010 at the trailing edge of the next clock pulse ($T_2 = 1$ only when $O_2O_1O_0$= 010. Therefore, the SC generates $L = 1$ and $d_2d_1d_0 = 010$.

Similarly, from the controller's state diagram of Figure 7.40, if the present state is T_5, the next control state is also T_5. The SC must generate the outputs $L = 1$ and $d_2d_1d_0 =$ 101. The SC truth table of Figure 7.41 shows these out-of-sequence counts. For each row of the SC truth table of Figure 7.44(a), a product term is generated in the PLA:

$$P_0 + \overline{Z}T_3 \text{ and } P_1 = T_5.$$

The PLA (Figure 7.44b) generates four outputs: L, d_2, d_1, and d_0. Each output is directly generated by the SC truth table and the product terms. The PLA outputs are as follows:

$$\begin{aligned} L &= P_0 + P_1 \\ d_2 &= P_1 \\ d_1 &= P_0 \\ d_0 &= P_1 \end{aligned}$$

The controller design is completed by relating the control states (T_0 through T_5) to the control signals (C_0 though C_6) as follows:

$$\begin{aligned} C_0 &= C_1 = T_0 \\ C_2 &= T_1 \\ C_3 &= C_4 = C_6 = T_2 \\ C_5 &= T_4 \end{aligned}$$

From these equations, when the control is in state T_0 or T_2, multiple micro-operations are performed. Otherwise,when the control is in state T_1 or T_4, a single micro-operation is performed.

The unsigned multiplication algorithm just implemented using hardwired control can be considered as an unsigned multiplication instruction with a microprocessor. To execute this instruction, the microcomputer will read (fetch) this multiplication instruction from external memory into the instruction register located inside the microprocessor. The contents of this instruction register will be input to the control unit for execution. The control unit will generate the control signals C_0 through C_6 as shown in Figure 7.43. These control signals will then be applied to the appropriate components of the processing section in Figure 7.38 at the proper instants of time shown in Figure 7.40. Note that the control signals are physically connected to the hardware elements of Figure 7.38. Thus, the execution of the unsigned multiplication instruction will be completed by the microprocessor.

Microprogrammed Control Unit Design

As mentioned earlier, a microprogrammed control unit contains programs written

using microinstructions. These programs are stored in a control memory normally in a ROM inside the CPU. To execute instructions, the microprocessor reads (fetches) each instruction into the instruction register from external memory. The control unit translates the instruction for the microprocessor. Each control word contains signals to activate one or more microoperations. A program consisting of a set of microinstructions is executed in a sequence of micro-operations to complete the instruction execution. Generally, all microinstructions have two important fields:

- Control word
- Next address

The control field indicates which control lines are to be activated. The next address field specifies the address of the next microinstruction to be executed. The concept of microprogramming was first proposed by W. V. Wilkes in 1951 utilizing a decoder and an 8 × 8 ROM with a diode matrix. This concept is extended further to include a control memory inside the CPU. The cost of designing a CPU primarily depends on the size of the control memory. The length of a microinstruction, on the other hand, affects the size of the control memory. Therefore, a major design effort is to minimize the cost of implementing a microprogrammed CPU by reducing the length of the microinstruction.

The length of a microinstruction is directly related to the following factors:

- The number of micro-operations that can be activated simultaneously. This is called the "degree of parallelism."
- The method by which the address of the next microinstruction is determined.

All microinstructions executed in parallel can be included in a single microinstruction with a common op-code. The result is a short microprogram. However, the length of the microinstruction increases as parallelism grows.

The control bits in a microinstruction can be organized in several ways. One obvious way is to assign a single bit for each control line. This will provide full parallelism. No decoding of the control field is necessary. For example, consider Figure 7.45 with two registers, X and Y with one outbus.

In figure 7.45, the contents of each register are transferred to the outbus when the

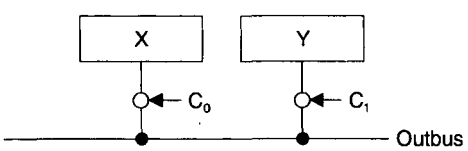


FIGURE 7.45 An example of a register transfer

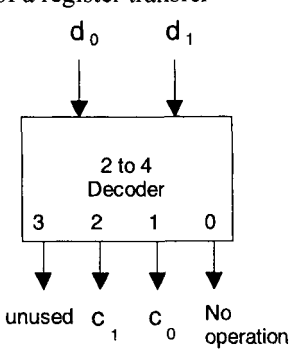


FIGURE 7.46 Encoded format

appropriate control line is activated:

C_0: outbus ← X

C_1: outbus ← Y

Here, each operation can be performed one at a time because there is only one outbus. A single bit can be assigned to perform each transfer as follows:

Control Bits		Operation Performed
C_0	C_1	
1	0	Outbus ← X
0	1	Outbus ← Y
0	0	No operation

This method is called "unencoded format."

The three operations can be implemented using two bits and a 2-to-4 decoder as shown in Figure 7.46. This is called "encoded format." The relationship between the encoded and actual control information is as follows:

Encoded Bits		Operation Performed
d_1	d_0	
0	0	No operation
0	1	Outbus ← x
1	0	Outbus ← y

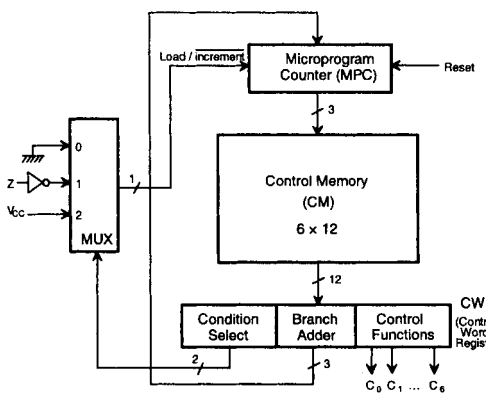
Note that a 5-bit control field is required for five operations. However, three encoded bits are required for five operations using a 3 to 8 decoder. Hence, the encoded format typically provides a short control field and thus results in short microinstructions. However, the need for a decoder will increase the cost. Therefore, there is a trade-off between the degree of parallelism and the cost. Microinstructions can be classified into two groups: horizontal and vertical. The horizontal microinstruction mechanism provides long microinstructions, a high degree of parallelism, and little or no encoding. The vertical microinstruction method, on the other hand, offers short microinstructions, limited parallelism, and considerable decoding.

Microprogramming is the technique of writing microprograms in a microprogrammed control unit. Writing microprograms is similar to writing assembly language programs. Microprograms are basically written in a symbolic language called microassembly language. These programs are translated by a microassembler to generate microcodes, which are then stored in the control memory.

In the early days, the control memory was implemented using ROMs. However, these days control memories are realized in writeable memories. This provides the flexibility of interpreting different instruction set by rewriting the original microprogram, which allows implementation of different control units with the same hardware. Using this approach, one CPU can interpret the instruction set of another CPU. The design of a microprogrammed control unit is considered next. The 4-bit × 4-bit unsigned multiplication

Control Memory Address		*Control Word*
0	START	`R ← 0, M ← inbus;`
1		`Q ← inbus;`
2	LOOP	`R ← R + M, Q ← Q - 1;`
3		`If Z = 0 then goto Loop;`
4		`outbus ← R;`
5	HALT	`Go to HALT`

FIGURE 7.47 Symbolic microprogram for 4-bit x 4-bit unsigned multiplication using repeated addition



C_0 : R ← 0
C_1 : M ← inbus
C_2 : Q ← inbus
C_3 : F ← 1 + r
C_4 : Q ← Q − 1
C_5 : outbus ← R
C_6 : R ← F

FIGURE 7.48 Microprogrammed unsigned multiplier control unit

using hardwired control (presented earlier) is implemented by microprogramming. The register transfer description shown in Figure 7.36 is rewritten in symbolic microprogram language as shown in Figure 7.47. Note that the unsigned 4-bit x 4-bit multiplication uses repeated addition. The result (product) is assumed to be 4 bits wide.

To implement the microprogram, the hardware organization of the control unit shown in Figure 7.48 can be used. The various components of the hardware of Figure 7.48 are described in the following:

1. **Microprogram Counter (MPC).** The MPC holds the address of the next microinstruction to be executed. It is initially loaded from an external source to point to the starting address of the microprogram. The MPC is similar to the program counter (PC). The MPC is incremented after each microinstruction fetch. If a branch instruction is encountered, the MPC is loaded with the contents of the branch address field of the microinstruction.
2. **Control Word Register (CWR).** Each control word in the control memory in this example is assumed to contain three fields: condition select, branch address, and control function. Each microinstruction fetched from the Control Memory is loaded into the CWR. The organization of the CWR is same for each control word

and contains the three fields just mentioned. In the case of a conditional branch microinstruction, if the condition specified by the condition select field is true, the MPC is loaded with the branch address field of the CWR; otherwise, the MPC is incremented to point to the next microinstruction. The control function field contains the control signals.

3. **MUX (Multiplexer).** The MUX is a condition select multiplexer. It selects one of the external conditions based on the contents of the condition select field of the microinstruction fetched into the CWR.

In Figure 7.48, a 2-bit condition select field is required as follows:

Condition Select Field		Interpretation
0	0	No branching (no condition)
0	1	Branch if $Z = 0$
1	0	Unconditional branching

From Figure 7.47 six control memory address (addresses 0 through 5) are required for the control memory to store the microprogram. Therefore, a 3-bit address is necessary for each microinstruction. Hence, three bits for the branch address field are required. From Figure 7.48 seven control signals (C_0 through C_6) are required. Therefore, the size of the control function field is 7 bits wide. Thus, the size of each control word can be determined as follows:

size of *a* control word	=	size of the condition select field	+	size of the branch address field	+	number of control signals
	=	2	+	3	+	7
	=	12 bits				

Therefore, the size of the control memory is 6 bits × 12 bits because the microprogram requires six addresses (0 through 5) and each control word is 12 bits wide. The size of the CWR is 12 bits. The complete binary listing of the microprogram is shown in Figure 7.49.

ROM Address			Control Word											Comments
In decimal	In binary		Condition Select		Branch Address			Control Function						
								C_0	C_1	C_2	C_3	C_4	C_5	C_6	
0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	R← 0, M ← inbus
1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	Q ← inbus
2	0	1	0	0	0	0	0	0	0	0	1	1	0	1	R ← R + M, Q ← Q –1, R ← F
3	0	1	1	0	1	0	1	0	0	0	0	0	0	0	If Z = 0 then go to address 2 (loop)
4	1	0	0	0	0	0	0	0	0	0	0	0	1	0	outbus ← R
5	1	0	1	1	0	1	0	1	0	0	0	0	0	0	Go to address 5 (HALT)

FIGURE 7.49 Binary listing of the microprogram for 4-bit × 4-bit unsigned multiplication

Let us now explain the binary program. Consider the first line of the program. The instruction contains no branching. Therefore, the condition select field is 00. The contents of the branch in this case filled with 000. In the control function field, two micro-operations, C_0 and C_1, are activated. Therefore, both C_0 and C_1 are set to 1; C_2 through C_6 are set to 0.

This results in the following binary microinstruction shown in the first line (address 0) of Figure 7.49:

Condition Select	Branch Address	Control Function
00	000	1100000

Next, consider the conditional branch instruction of Figure 7.49. This microinstruction implements the conditional instruction "If $Z = 0$ then go to address 2." In this case, the microinstruction does not have to activate any control signal of the control function field. Therefore, C_0 through C_6 are zero. The condition select field is 01 because the condition is based on $Z = 0$. Also, if the condition is true ($Z = 0$), the program branches to address *2*. Therefore, the branch address field contains 010_2. Thus, the following binary microinstruction is obtained:

Condition Select	Branch Address	Control Function
01	010	000000

The other lines in the binary representation of the microprogram can be explained similarly. To execute an unsigned multiplication instruction implemented using the repeated addition just described, a microprogrammed microprocessor will fetch the instruction from external memory into the instruction register. To execute this instruction, the microprocessor uses the control unit of Figure 7.48 to generate the control word based on the microprogram of Figure 7.49 stored in the control memory. The control signals C_0 through C_6 of the control function field of the CWR will be connected to appropriate components of Figure 7.38 The instruction will thus be executed by the microprocessor.

By examining the microprogram in Figure 7.49, it is obvious that the control function field contains all zeros in case of branch instructions. In a typical microprogram, there may be several conditional and unconditional branch instructions. Therefore, a lot of valuable memory space inside the control unit will be wasted if the control field is filled with zeros. In practice, the format of the control word is organized in a different manner to minimize its size. This reduces the implementation cost of the control unit. Whenever there are several branch instructions, the microinstructions, can be formatted by using a method called multiple microinstruction format. In this approach, 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 means that the branch address field can be removed from the microinstruction format. Therefore, the control function field is used to specify the branch address itself. Typically,

ROM Address		Control Word													Comments
In decimal	In binary	Condition Select		Branch Address			Control Function								
							C_0	C_1	C_2	C_3	C_4	C_5	C_6		
0	0 0 0	0	0	0	0	0	1	1	0	0	0	0	0		R← 0, M ← inbus
1	0 0 1	0	0	0	0	0	0	0	1	0	0	0	0		Q ← inbus
2	0 1 0	0	0	0	0	0	0	0	0	1	1	0	1		R ← R + M, Q ← Q −1, R ← F
3	0 1 1	0	1	0	1	0	0	0	0	0	0	0	0		If Z = 0 then go to address 2 (loop)
4	1 0 0	0	0	0	0	0	0	0	0	0	0	1	0		outbus ← R
5	1 0 1	1	0	1	0	1	0	0	0	0	0	0	0		Go to address 5 (HALT)

FIGURE 7.50 Reduction of the length of microinstruction of Figure 7.49

each microinstruction will have two fields, as shown next:

CONDITION-SELECT FIELD		CONTROL FUNCTION FIELD						
S_1	S_0	C_6	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	1	Branch if cond-2 = 1
1	0	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 7-bit branch address. In this example, it is assumed that when S_1 S_0 = 01, the MPC will be loaded with the appropriate address specified by C_6 C_5 C_4 C_3 C_2 C_1 C_0 if the condition Z = 0 is satisfied; on the other hand, if S_1 S_0 = 10, an unconditional branch to the address specified by the Control Function / Branch Address Field occurs.

In order to illustrate this concept, the microprogram for 4-bit by 4-bit unsigned multiplication of Figure 7.49 is rewritten using the multiple instruction format as shown in Figure 7.50.

It can be seen from the figure 7.50 that the total size of the control store is 54 bits (6 × 9 = 54). In contrast, the control store of figure 7.49 contains 72 bits. For large microprograms with many branch instructions, tremendous memory savings can be accomplished using the multiple microinstructon format. Addresses 0, 1, 2, and 4 contain microinstructions with the contents of the conditional select field as 00, and are considered as operate instructions. In this case, the contents of the control function field are directed to the processing hardware.

Address 3 contains a conditional branch instruction since the contents of the condition select field are 01; while address 5 contains an unconditional branch instruction

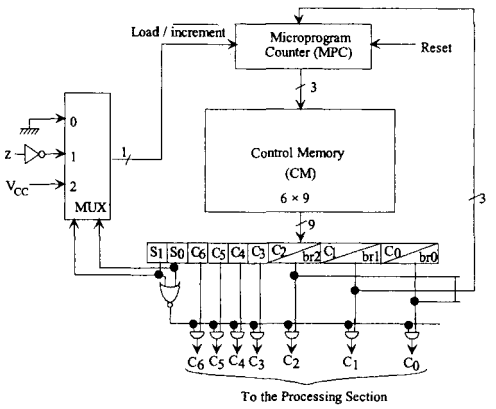


FIGURE 7.51 Microprogrammed Controller for the Microprogram of Figure 7.50.

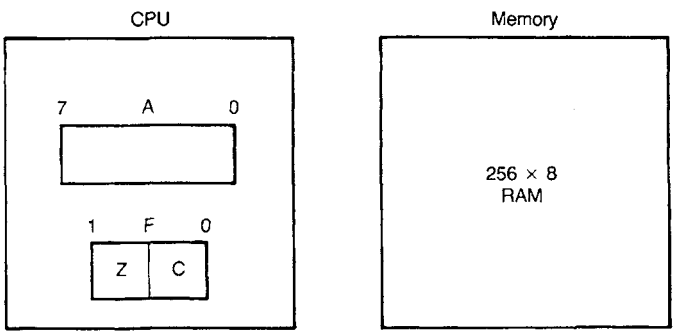


FIGURE 7.52 Programming Model of a Simple Processor

(halt instruction; that is, jump to the same address) since the condition select field is 10. Hence, the 7-bit control function field directly specifies the desired branch addresses 2 and 5, respectively. Figure 7.51 shows the hardware schematic.

7.4 Design of a Microprogrammed CPU

Next, the design of a microprogrammed processor is illustrated. The programming model of this processor is shown in Figure 7.52.

The CPU contains two registers:

1. An 8-bit 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 7.53.

From Figure 7.53, 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 defined in Table 7.1.

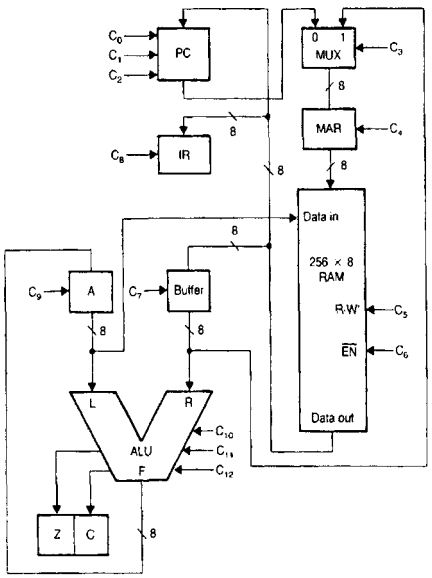


FIGURE 7.53 Hardware Schematic of the Simple Processor (Note: 8-bit PC is connected to eight 2 to 1 MUXs-- Not shown above)

From Figure 7.54, 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 number 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 7.55.
The bit I3 of Figure 7.55 decides the instruction type. If I3 = 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 Io 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:

TABLE 7.1 Definitions of the Control Inputs C_0-C_9

MICROOPERATION	COMMENT
C_0: PC ← 0	Clear PC to zero.
C_1: PC ← PC + 1	Advance the PC.
$C_2C_5\overline{C_6}$: PC ← M ((MAR))	Read the data from the memory and save it in the PC.
$\overline{C_3}C_4$: MAR ← PC	Transfer the contents of the PC into MAR.
$C_5\overline{C_6}C_7$: 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.
$C_5\overline{C_6}C_8$: 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.
$\overline{C_5}\,\overline{C_6}$: M ((MAR)) ← A	Save contents of register A into memory.

The eight ALU operations performed by the CPU are defined by $C_{10}C_{11}C_{12}$ as follows:

C_{11}	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 AND R
1	1	1	NOT L

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 7.56 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 microprogram is shown in

General Format	Instruction Length in Bytes	Object Code		Instruction Type	Operation	Comment
		In binary	In hex			
LDA ‹addr›	2	0000 1000	08	MRI	A ← M (‹addr›)	Load register A direct
		‹addr8›	‹addrH)			
STA ‹addr›	2	0000 1001	09	MRI	M (‹addr›) ← A	Store register A direct
		‹addr8›	‹addrH›			
ADD ‹addr›	2	0000 1010	0A	MRI	A ← A + M (‹addr›)	Add register A direct
		‹addr8›	‹addrH›			
SUB ‹addr›	2	0000 1011	0B	MRI	A ← A – M (‹addr›)	Subtract register A direct
		‹addr8›	‹addrH›			
JZ ‹addr›	2	0000 1100	0C	MRI	If Z=l then PC ← ‹addr› else PC ← PC + 1	Jump on zero flag set
		‹addr8›	‹addrH›			
JC ‹addr›	2	0000 1101	0D	MRI	If C = 1 then PC ← ‹addr› else PC ← PC + 1	Jump on carry flag set
		‹addr8›	‹addrH›			
AND ‹addr›	2	0000 1110	0E	MRI	A ← A ^ M (‹addr›)	And register A direct
		‹addr8›	‹addrH›			
CMA	1	0000 0000	00	NMRI	A ← $\overline{A}$	Comple-ment register A
INCA	1	0000 0010	02	NMRI	A ← A + 1	Increment register A
DCRA	1	0000 0100	04	NMRI	A ← A - 1	Decrement register A
HLT	1	0000 0110	06	NMRI	Halt	Halt CPU.

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

FIGURE 7.54 Instruction Set to be Implemented

Figure 7.57.

The hardware organization of the microprogrammed control unit for this situation shown in Figure 7.58 directly follows the symbolic listing shown in Figure 7.57. 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 7.57 into a binary microprogram is left as an exercise.

Mnemonic	Op-code Bit and Their Interpretations							
					TC	GN		SC
	I7	I6	I5	I4	I3	I2	I1	I0
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 I3 = 1, then it is a MRI; otherwise it is a NMRI)

GN: Group number within a type

(I2	I1	Group no.	
0	0	0	
0	1	1	
1	0	2	
1	1	3	)

SC: Subcategory within a group

FIGURE 7.55 Op-code Encoding Logic

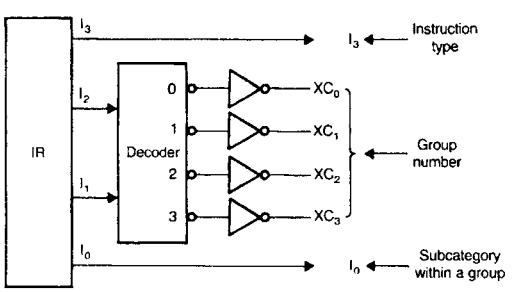


FIGURE 7.56 Instruction-decoding Hardware

Symbolic Microprogram:

ROM Address			
0		PC ← 0;	These operations constitute the fetch cycle.
1	FETCH	MAR ← PC;	
2		IR ← M ((MAR)), PC ← PC + 1;	
3		IF I_3 = 1 then go to MEMREF;	
4		IF XC_0 = 1 then go to CMA;	Here we decode the instructions.
5		IF XC_1 = 1 then go to INCA;	
6		IF XC_2 = 1 then go to DCRA;	
7		Go to HALT;	
8	CMA	A ← $\overline{A}$;	Execute CMA instructions.
9		Go to FETCH;	
10	INCA	A ← A + 1;	Execute INCA instruction.
11		Go to FETCH;	
12	DCRA	A ← A - 1;	Execute DCRA instruction.
13		Go to FETCH;	
14	MEMREF	IF XC_0 = 1 then go to LDSTO;	Here we branch to the various groups of the memory reference instruction.
15		IF XC_1 = 1 then go to ADSUB;	
16		IF XC_2 = 1 then go to JMPS;	
17	AND	MAR ← PC;	
18		BUFFER ← M ((MAR)), PC ← PC + 1;	Execute AND instruction.
19		MAR ← BUFFER;	
20		BUFFER ← M ((MAR));	
21		A ← A ∧ BUFFER;	
22		Go to FETCH;	
23	LDSTO	MAR ← PC;	
24		BUFFER ← M ((MAR)), PC ← PC + 1;	
25		MAR ← BUFFER;	
26		IF I_0 = 1 then go to STO;	
27	LOAD	BUFFER ← M ((MAR));	
28		A ← BUFFER;	
29		Go to FETCH;	
30	STO	M ((MAR)) ← A;	
31		Go to FETCH;	

FIGURE 7.57 Symbolic Microprogram that implements the instruction set of figure 7.54

```
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;
37  ADD      A ← A + BUFFER;                    Execute ADD instruction
38           Go to FETCH;
39  SUB      A ← A – BUFFER;                    Execute SUB instruction
40           Go to FETCH;
41  JMPS     MAR ← PC;
42           IF I0 = 1 then go to JOC;
43           IF I0 = 1 then go to JOC;
44  JOZ      IF Z = 1 then go to LOADPC;        Execute JZ instruction
45           PC ← PC + 1;
46           Go to FETCH;
47  JOC      IF C = 1 then go to LOADPC;        Execute JC instruction
48           PC ← PC + 1;
49           Go to FETCH;
50  LOADPC   PC ← M((MAR));
51           Go to FETCH;
52  HALT     Go to HALT;                        Execute HALT instruction
```

FIGURE 7.57 *Continued*

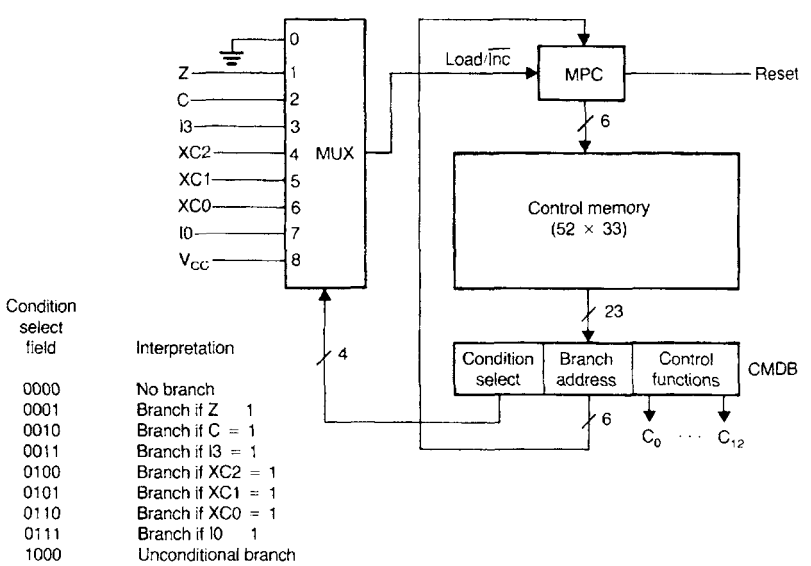


FIGURE 7.58 Microprogrammed Controller for the CPU

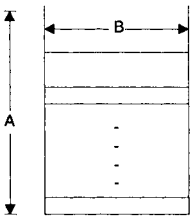


FIGURE 7.59 A microprogram of size A × B

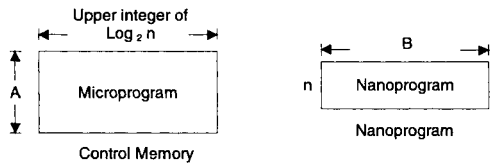


FIGURE 7.60 Nanomemory

000	0100
001	0000
010	0100
011	0100
100	0000
101	1010
110	1010

FIGURE 7.61 7 × 4-bit single control memory

000	00
001	01
010	00
011	00
100	01
101	10
110	10

7 × 2-bit microcontrol store

00	0100
01	0000
10	1010

3 × 4 nanocontrol store

FIGURE 7.62 Two-level store (nanomemory)

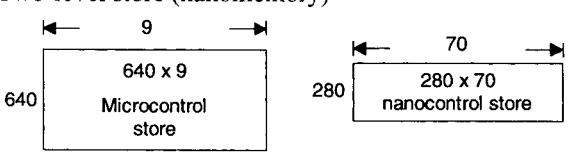


FIGURE 7.63 68000 nanomemory

Example 7.1

If the following two instructions are to be added to the instruction set of Figure 7.54, write a symbolic microprogram for the CPU of section 7.3 that describes the execution of each instruction:

	GENERAL FORMAT	OPERATION	DESCRIPTION
(a)	CLRA	A←0	Clear register A
(b)	PRSA	A← 11111111	Set register A to all ones

Solution:

```
(a)  CLRA:   A←0              ; Use ALU's zero output (C10C11C12=000)
             go to FETCH      ;
(b)  PRSA:   A←0              ; Use ALU's zero output (C10C11C12=000)
             A←Ā              ;
             go to FETCH      ;
```

Nanomemory is another approach for reducing the size of the control memory. This technique contains a two-level memory: control memory and nanomemory. At the outset, are may feel that the two-level memory will increase the overall cost. In fact, it reduces the cost of the system by minimizing the memory size.

The concept of nanomemory is derived from a combination of horizontal and vertical instructions. However, this method provides trade-offs between them.

Motorola uses nanomemory to design the control units of their popular 16-bit and 32-bit microprocessors, including the 68000, 68020, 68030, and 68040. The nanomemory method provides significant savings in memory when a group of micro-operations occur several times in a microprogram. Consider the microprogram of Figure 7.59, which contains A microinstructions B bits wide. The size of the control memory to store this microprogram is AB bits. Assume that the microprogram has n ($n < A$) unique microinstructions. These n microinstructions can be held in a separate memory called the "nanomemory" of size nB bits. Each of these n instructions occurs once in the nanomemory. Each microinstruction in the original microprogram is replaced with the address that specifies the location of the nanomemory in which the original B-bit-wide microinstructions are held.

Because the nanomemory has n addresses, only the upper integer of $\log_2 n$ bits is required to specify a nanomemory address. This is illustrated in Figure 7.60. The operation of microprocessor employing a nanomemory can be explained as follows: The microprocessor's control unit reads an address from the microprogram. The content of this address in the nanomemory is the desired control word. The bits in the control word are used by the control unit to accomplish the desired operation. Note that a control unit employing nanomemory (two-level memory) is slower than the one using a conventional control memory (single memory). This is because the nanomemory requires two memory reads (one for the control memory and the other for the nanomemory). For a single conventional control memory, only 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 7 x 4-bit microprogram stored in the single control memory of Figure 7.61. This simplified example is chosen to illustrate the nanomemory concept even though this is not a practical example. In this program, 3 out of 7 microinstructions are unique.

Therefore, the size of the microcontrol store is 7 × 2 bits and the size of the nanomemory is 3 × 4 bits. This is shown in Figure 7.62.

Memory requirements for the single control memory = 7 × 4 = 28 bits. Memory requirements for nanomemory = (7 × 2 + 3 × 4) bits = 26 bits. Therefore, the saving using nanomemory = 28 - 26 = 2 bits. For a simple example like this, 2 bits are saved. The HMOS 68000 control unit nanomemory includes a 640 × 9-bit microcontrol store and a 280 × 70-bit nanocontrol store as shown in Figure 7.63. In Figure 7.63, out of 640 microinstructions, 280 are unique. If the 68000 were implemented using a single control memory, the requirements would have been 640 × 70 bits. Therefore,

Memory savings = (640 × 70) – (640 × 9 + 280 × 70) bits
= 44,800 – 25,360
= 19,440 bits

This is a tremendous memory savings for the 68000 control unit.

QUESTIONS AND PROBLEMS

7.1 It is desired to implement the following instructions using block code: ADD, SUB, XOR, MOVE, HALT. Draw a block diagram.

7.2 The instruction length and the size of an address field are 9 bits and 3 bits respectively. Is it possible to have

6 two-address instructions
15 one-address instructions
8 zero-address instructions

using expanding op-code technique? Justify your answer.

7.3 Using the instruction format of Problem 7.2, is it possible to have

7 two-address instructions
7 one-address instructions
8 zero-address instructions

using expanding opcode technique? Justify your answer.

7.4 Assume that it is desired to have 2 two-address, 7 one-address, and 25 zero-address instructions in a computer instruction set. Using expanding op-code technique with a 2-bit op-code and 3-bit address field, is it possible to accomplish the above? If so, justify your answer and determine the instruction length.

7.5 Assume that using an instruction length of 9 bits and the address field size of 3 bits, 5 two-address and 10 one-address instructions have already been designed, using expanding op-code technique. Is it possible to have at least 48 zero-address instructions that can be added to the instruction set?

7.6 Design a combinational logic shifter with 4-bit input and 4-bit output as follows:

$\overline{OE}$	Shift Count S_1	Shift Count S_0	4 - bit output
1	X	X	High Impedance output lines
0	0	0	No Shift
0	0	1	Right Shift once
0	1	0	Right Shift twice
0	1	1	Right Shift three times

where X means don't care. Using multiplexers and tristate buffers, draw a logic diagram.

7.7 Draw a logic diagram for a 4 × 4 barrel shifter.

7.8 Using a minimum number of full adders and multiplexers, design an incrementer/ decrementer circuit as follows: If $S = 0$, output $y = x + 1$; otherwise, $y = x - 1$. Assume x and y are 4-bit signed numbers and the result is 4 bits wide.

7.9 Design a combinational circuit to compute the absolute value of an 8-bit twos complement number. Use 8-bit binary adder and exclusive-OR gates. Draw a logic circuit.

7.10 Using a 4-bit CLA as the building block, design an 8-bit adder.

7.11 Design:

(a) a 16-bit adder whose worst-case add-time is 10Δ using a 4-bit CLA as a building block.

(b) the fastest 64-bit adder using a 4-bit CLA as the building block. Estimate the worst-case add-time of your design.

(c) a combinational circuit to compute the function $f(x) = (3/8) * x$ where x is a 4-bit 2's complement number.

7.12 Design an arithmetic logic unit to perform the following functions:

S_1	S_0	F
0	0	A plus B
0	1	A minus B
1	0	A AND B
1	1	A OR B

Use multiplexers, binary adders, and gates as needed. Assume that A and B are 4-bit numbers. Draw a logic circuit.

7.13 Design a combinational circuit that will perform the following operations:

S_1	S_0	Y
0	0	0
0	1	A
1	0	B
1	1	15_{10}

Assume that A is a 4-bit number and $B = \overline{a_3}\,\overline{a_2}\,\overline{a_1}\,\overline{a_0}$. Draw a logic diagram.

7.14 Design a 4-bit ALU to perform the following operations:

S	F
0	Logical Left Shift A once
1	0

Assume that A is a 4-bit number. Draw a logic diagram using a binary adder, multiplexers, and inverters as necessary.

7.15 Design a 4-bit arithmetic unit as follows:

S	F
0	A plus B
1	A plus 1

Assume that A and B are 4-bit numbers

7.16 Design an ALU to perform the following operations:

S_1	S_0	F
0	0	x plus y
0	1	x
1	0	B
1	1	x $\oplus$ y

Assume that x and y are 4-bit numbers, and $B = \overline{y_3}\,\overline{y_2}\,\overline{y_1}\,\overline{y_0}$. Draw a logic diagram.

7.17 Assume two 2's complement signed numbers, $M = 11111111_2$ and $Q = 11111100_2$. Perform the signed multiplication using the algorithm described in Section 7.2.2.

7.18 What is the purpose of the control unit in a microprocessor?

7.19 Draw a logic diagram to implement the following register transfers:

(a) If the content of the 8-bit register R is odd, then

$$x \leftarrow x \oplus y$$

else $x \leftarrow x$ AND y

Assume x and y are 4 bits wide.

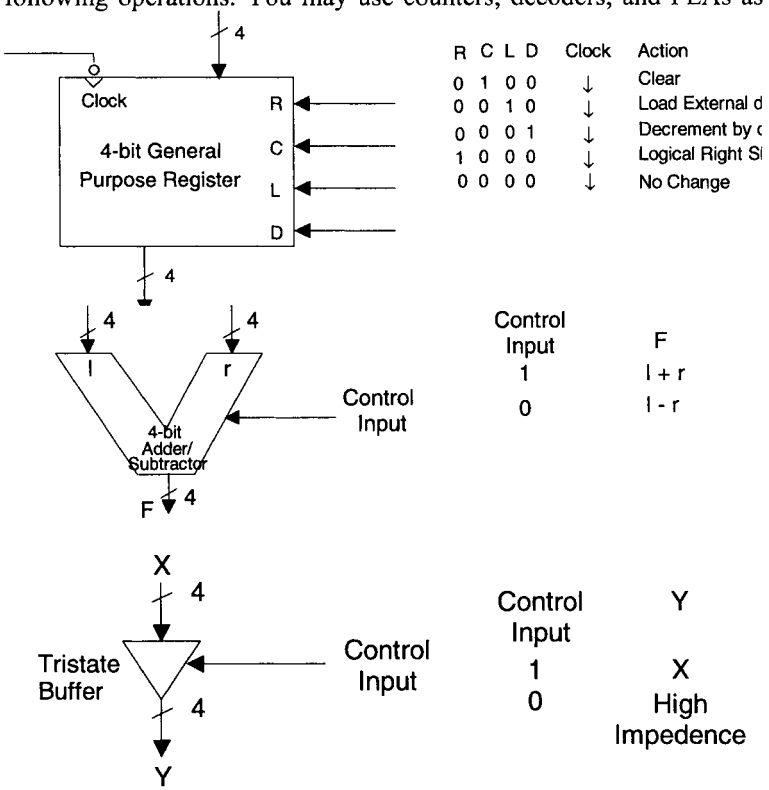
(b) If the number in the 8-bit register R is negative, then $x \leftarrow x - 1$ else $x \leftarrow x + 1$. Assume x and y are 4 bits wide.

7.20 Discuss briefly the merits and demerits of single-bus, two-bus, and three-bus architectures inside a control unit.

7.21 What is the basic difference between hardwired control, microprogramming, and nanoprogramming? Name the technique used for designing the control units of the Intel 8086, Motorola 68000, and PowerPC.

7.22 Using the following components: 4-bit general-purpose register, 4-bit adder/subtractor, and tristate buffer, and assuming the inbus and outbus are

4 bits wide, design a control unit using hardwired control to perform the following operations. You may use counters, decoders, and PLAs as required.



(a) Outbus ← 4 × *A*. Assume *A* is a 4-bit unsigned number and the result is 4 bits wide.

(b) If the 4-bit number in register *B* is odd, outbus ← 0; otherwise outbus ← *A* + (*B* / 2). Assume *A* and *B* are unsigned 4 bit numbers. Also, assume data is already loaded into *B*.

(c) If the content of a 4-bit register *Q* = 0, perform *R* ← *M* and then transfer the 4-bit result to outbus. On the other hand, if the content of the 4-bit register *Q* ≠ 0, perform *R* ← 0 and then transfer the 4-bit result to the outbus. Assume *M* and *R* are 4 bits wide.

7.23 Repeat Problem 7.22 using microprogramming.

7.24 Discuss the basic differences between microprogramming and nano-programming.

7.25 (a) A conventional microprogrammed control unit includes 1024 words by 85 bits. Each of 512 microinstructions are unique. Calculate the savings if any by having a nanomemory. Calculate the sizes of microcontrol memory and nanomemory.

(b) Consider the following 14 × 6 microprogram using a conventional control memory:

0000	000001
0001	000010
0010	000001
0011	000011
0100	000001
0101	000010
0110	000001
0111	000011
1000	000010
1001	000001
1010	000011
1011	000010
1100	000011
1101	000010
1110	000001

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

7.26 Discuss the basic differences between CISC and RISC.

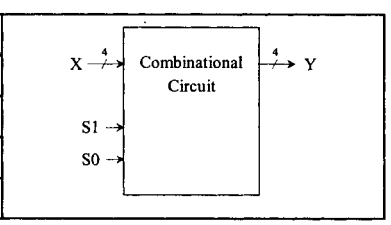
7.27 Design and implement a combinational circuit that will work as follows:

S1	S0	F
0	0	A plus B
0	1	Shift left (A)
1	0	A plus B plus 1
1	1	Shift left (A) + 1

Note that A and B are 4-bit operands

7.28 i) Design a combinational circuit that will satisfy the following specification.

S1	S0	Y_i
0	0	0
0	1	X_i
1	0	$\overline{X_i}$
1	1	1



ii) Using the results of part i), design a 4-bit, 8-function arithmetic unit that ii) will function as described next:

S2	S1	S0	F
0	0	0	A
0	0	1	A plus B
0	1	0	A plus $\overline{B}$
0	1	1	A minus 1

1	0	0	A plus 1
1	0	1	A plus B plus 1
1	1	0	A plus $\overline{B}$ plus 1
1	1	1	A

7.29 Design a 4-bit, 8-function arithmetic unit that will meet the following specifications:

S2	S1	S0	F
0	0	0	2A
0	0	1	A plus $\overline{B}$
0	1	0	A plus B
0	1	1	A minus 1
1	0	0	2A plus 1
1	0	1	A plus $\overline{B}$ plus 1
1	1	0	A plus B plus 1
1	1	1	A

7.30 (a) Using a 4-bit binary adder with inputs (A, B, and C_{in}), outputs (F and C_{out}), and one selection bit (S0), design an arithmetic circuit as follows:

S0	FUNCTION TO BE PERFORMED
0	A plus B
1	B plus 1

(b) Using another selection bit S1, modify the circuit of i) to include the arithmetic and logic functions as follows:

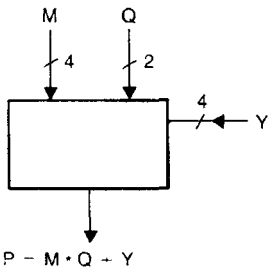
S1	S0	FUNCTION TO BE PERFORMED
0	0	F = A plus B
0	1	F = B
1	0	F = shift left (logical) A
1	1	F = $\overline{A}$

(c) Design a 4-bit logic unit that will function as follows:

S1	S0	F
0	0	A + B
0	1	A • B
1	0	$\overline{A}$
1	1	A ⊕ B

7.31 Design and implement a 6 × 6 array multiplier.

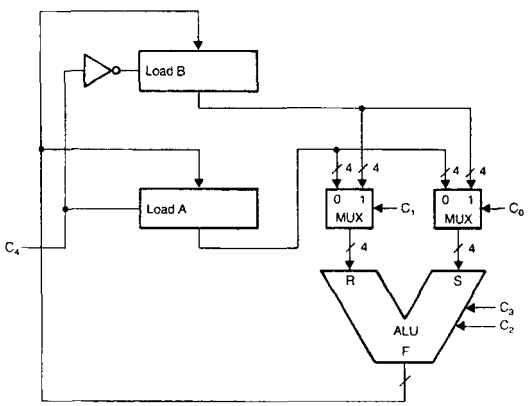
7.32 Design an unsigned 8 × 4 non-additive multiplier using additive-multiplier-module whose block diagram representation is as follows:



Assume that M, Q, and Y are unsigned integers.

7.33 Using four 256 x 8 ROMS and 4-bit parallel adders, design a 8 x 8 unsigned, nonadditive multiplier. Draw a logic diagram of your implementation.

7.34 Consider the registers and ALU shown in Figure P7.34:



The interpretation of various control points are summarized as follows:

C_3	C_2	F
0	0	R plus S
0	1	R minus S
1	0	R and S
1	1	R EX-OR S

C_1	C_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

FIGURE P7.34

Answer the following questions by writing suitable control 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:

$C_4\ C_3\ C_2\ C_1\ C_0$
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).

7.35 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.

7.36 It is desired to build an interface in order establish communication between a 32-bit host computer and a front end 8-bit microcomputer (See Figure P7.36). 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 Register Transfer Language description of the interface.

(b) Design the processing section of the interface.

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

(d) Draw a state diagram of the interface controller.

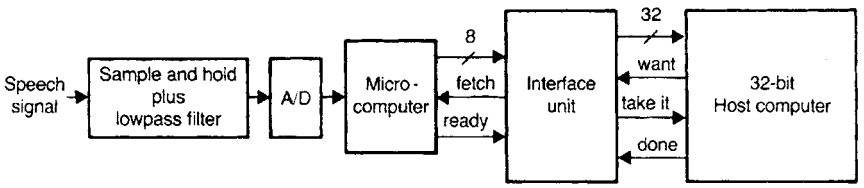


FIGURE P7.36

7.37 Solve Problem 7.35 using the microprogrammed approach.

7.38 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.

7.39 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:

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

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 in Figure P7.39. 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_i
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

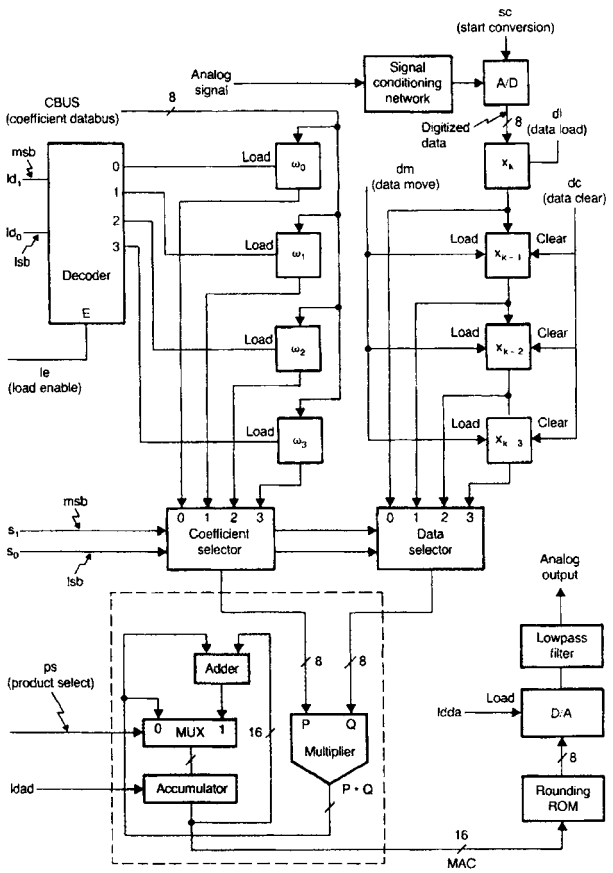
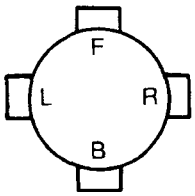


FIGURE P7.39

7.40 Your task is to design a microprogrammed controller for a simple robot with 4 sensors (see Fig. 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 Fig. B. The flow chart that describes the control algorithm is shown in Fig. 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

FIGURE P7.40a

(a) Specify the controller organization.

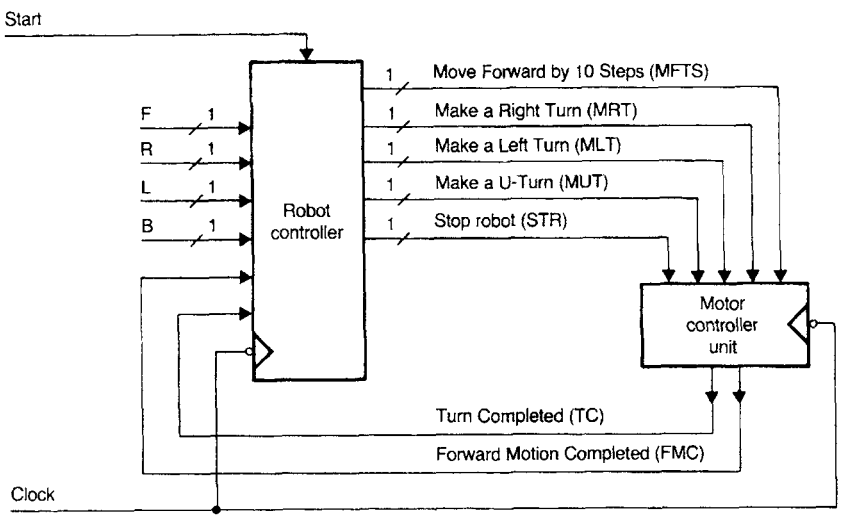


Figure B

FIGURE P7.40b

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

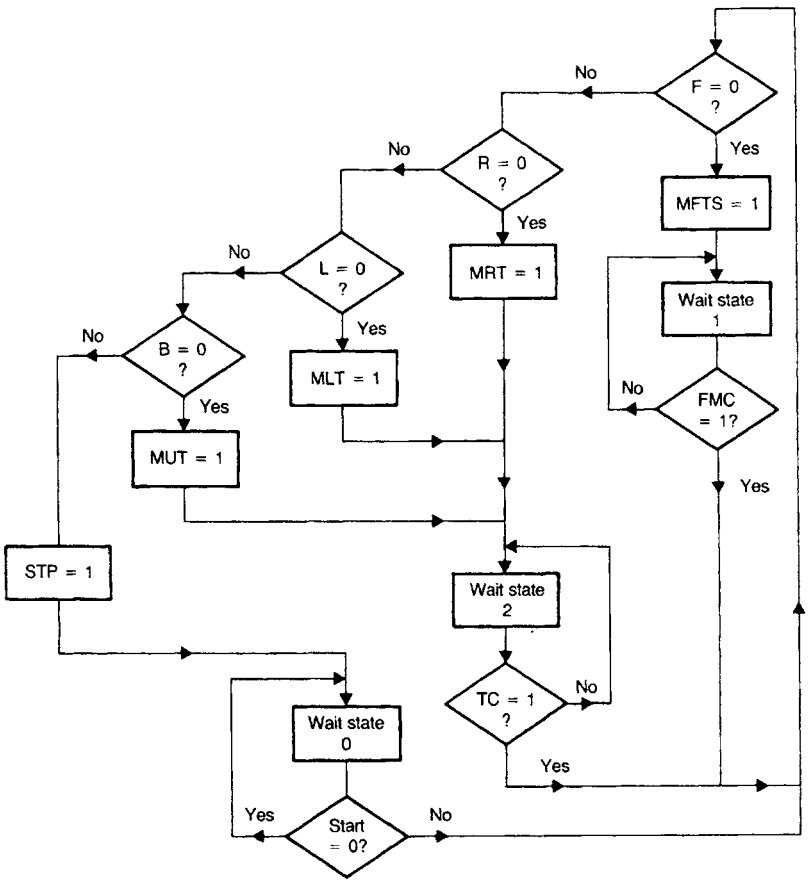


Figure C

FIGURE P7.40c

7.41 It is desired to add the following instructions to the instruction set shown in Figure 7.54.

	GENERAL FORMAT	OPERATION	DESCRIPTION
(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)	NEGA	A ← − A	This instruction negates the contents of A

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

7.42 Explain how the effect of an unconditional branch instruction of the following form is simulated:
JP ‹addr›

Use the instruction set shown in Figure 7.54.

7.43 Using the instruction set shown in Figure 7.54, write a program to add the contents of the memory locations 64_{16} through $6D_{16}$ and save the result in the address $6E_{16}$.

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

7.45 A microprogram occupies 100 words and each word typically emits 70 control signals. The architect claims that by using a $2^i \times 70$ nanomemory (for some $i > 0$), it is possible to save 4260 bits. If this were true, determine the number of distinct control states in the original microprogram (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.