Computer Architecture

Lectures from:

Computer System Architecture, third edition, M. Morris Mano

Computer Architecture

Digital computer

The digital computer is a digital system that performs various computational tasks. The word digital implies that the information in the computer is represented by variables that take a limited number of discrete values. These values are processed internally by components that can maintain a limited number of discrete states.

Digital computers use the binary number system, which has two digits: 0 and 1. A binary digit is called a bit.

A computer system is sometimes subdivided into two functional entities: hardware and software.

- The hardware of the computer consists of all the electronic components and electromechanical devices that comprise the physical entity of the device.
- Computer software consists of the instructions and data that the computer manipulates to perform various data-processing tasks.

Program:

A sequence of instructions for the computer is called a program. The data that are manipulated by the program constitute the data base.

A computer system is composed of its hardware and the system software available for its use. The system software of a computer consists of a collection of programs whose purpose is to make more effective use of the computer. The programs included in a systems software package are referred to as the operating system. They are distinguished from application programs written by the user for the purpose of solving particular problems. For example, a high-level language program written by a user to solve particular data-processing needs is an application program, but the compiler that translates the high-level language program to machine language is a system program. The customer who buys a computer system would need, in addition to the hardware, any available software needed for effective operation of the computer. The system software is an indispensable part of a total computer system. Its function is to compensate for the differences that exist between user needs and the capability of the hardware.

Computer hardware:

The hardware of the computer is usually divided into three major parts, as shown in Figure 1.

1. Central Processing Unit (CPU) which contains an arithmetic and logic unit for manipulating data, a number of registers for storing data, and control circuits for fetching and executing instructions.
2. The Memory of the computer which is called Random-access memory (RAM)
3. The input and output processor (IOP) contains electronic circuits for communicating and controlling the transfer of information between the computer and the outside world. The input and output devices connected to the computer include keyboards, printers, terminals, magnetic disk drives, and other communication devices.

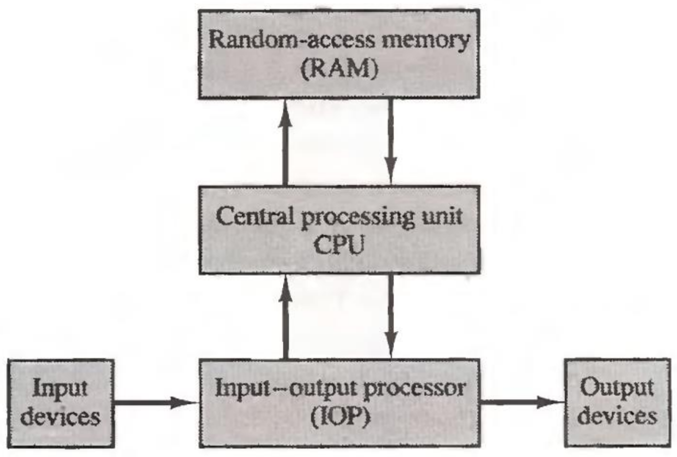


Figure 1 Block diagram of a digital computer

A digital system is an interconnection of digital hardware modules that accomplish a specific information-processing task. Digital systems vary in size and complexity from a few integrated circuits to a complex of interconnected and interacting digital computers. Digital system design invariably uses a modular approach. The modules are constructed from such digital components as registers, decoders, arithmetic elements, and control logic. The various modules are interconnected with common data and control paths to form a digital computer system.

Digital modules are best defined by the registers they contain and the operations that are performed on the data stored in them.

Micro Operation:

The operations executed on data stored in registers are called micro operations. A micro operation is an elementary operation performed on the information stored in one or more registers. The result of the operation may replace the previous binary information of a regis0ter or may be transferred to another register. Examples of micro operations are shift, count, clear, and load.

The internal hardware organization of a digital computer is best defined by specifying:

1. The set of registers it contains and their function.

2. The sequence of micro operations performed on the binary information stored in the registers.

3. The control that initiates the sequence of micro operations.

Register Transfer Language

The symbolic notation used to describe the micro operation transfers among registers is called a register transfer language. The term "register transfer" implies the availability of hardware logic circuits that can perform a stated micro operation and transfer the result of the operation to the same or another register.

The individual flip-flops in an n-bit register are numbered in sequence from 0 through n-1, starting from 0 in the rightmost position and increasing the numbers toward the left. Figure 2 shows the representation of registers in block diagram form.

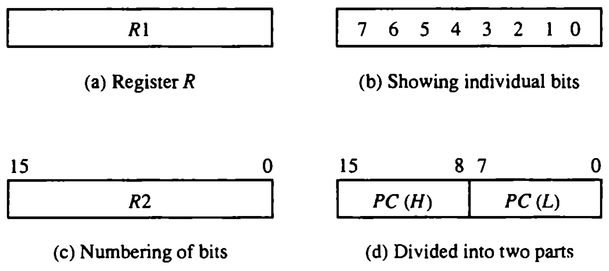


Figure 2 Block diagram of register.

Register transfer:

Information transfer from one register to another is designated in symbolic form by means of a replacement operator. The statement

$$R2 \leftarrow R1$$

denotes a transfer of the content of register R1 into register R2. It designates a replacement of the content of R2 by the content of R1. By definition, the content of the source register R1 does not change after the transfer.

A statement that specifies a register transfer implies that circuits are available from the outputs of the source register to the inputs of the destination register and that the destination register has a parallel load capability.

Normally, we want the transfer to occur only under a predetermined control condition. This can be shown by means of an if-then statement.

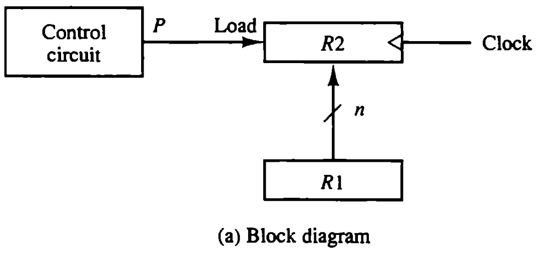
$$\text{If } (P = 1) \text{ then } (R2 \leftarrow R1)$$

where P is a control signal generated in the control section. It is sometimes convenient to separate the control variables from the register transfer operation by specifying a control function. A control function is a Boolean variable that is equal to 1 or 0. The control function is included in the statement as follows:

$$P\text{: } R2 \leftarrow R1$$

The control condition is terminated with a colon. It symbolizes the requirement that the transfer operation be executed by the hardware only if P = 1.

Every statement written in a register transfer notation implies a hardware construction for implementing the transfer. Figure 3 shows the block diagram that depicts the transfer from R1 to R2. The n outputs of register R1 are connected to the n inputs of register R2. The letter n will be used to indicate any number of bits for the register.



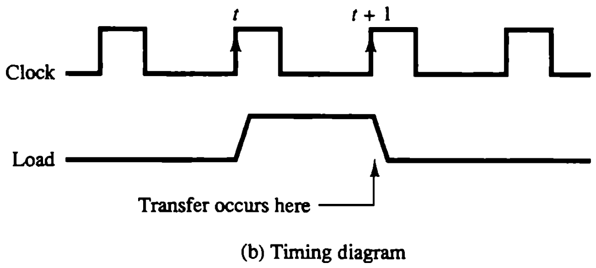


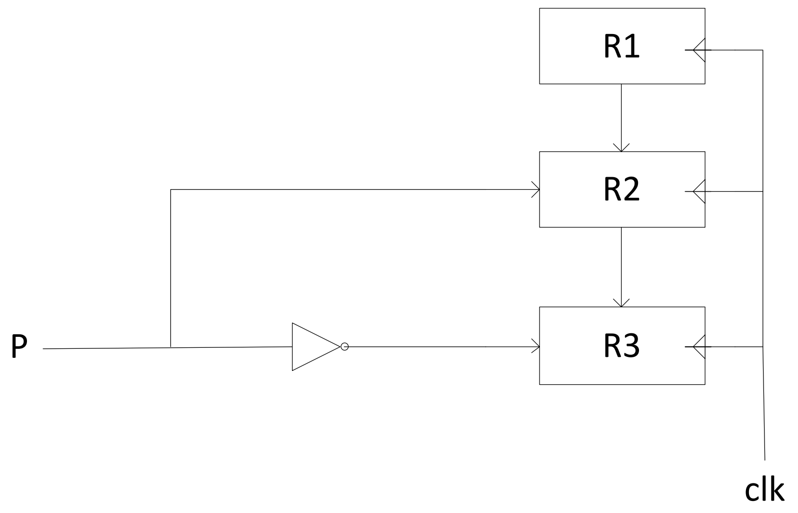
Figure 3 Transfer from R1 to R2 when P = 1.

The statement

$$T: R2 \leftarrow R1, R1 \leftarrow R2$$

denotes an operation that exchanges the contents of two registers during one common clock pulse provided that T = 1. This simultaneous operation is possible with registers that have edge-triggered flip-flops.

Example: Write the register transfer statements that represent the following block diagram



Solution:

$P: R2 \leftarrow R1$

$P': R3 \leftarrow R2$

H.W. Draw the block diagram for hardware to implement the following register transfer statements

$T: R1 \leftarrow R2, R4 \leftarrow R5$

$T': R2 \leftarrow R1$

The bus:

A typical digital computer has many registers, and paths must be provided to transfer information from one register to another. The number of wires will be excessive if separate lines are used between each register and all other registers in the system. A more efficient scheme for transferring information between registers in a multiple-register configuration is a common bus system. A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals determine which register is selected by the bus during each particular register transfer.

One way of constructing a common bus system is with multiplexers. The multiplexers select the source register whose binary information is then placed on the bus. The construction of a bus system for four registers is shown in Figure 4.

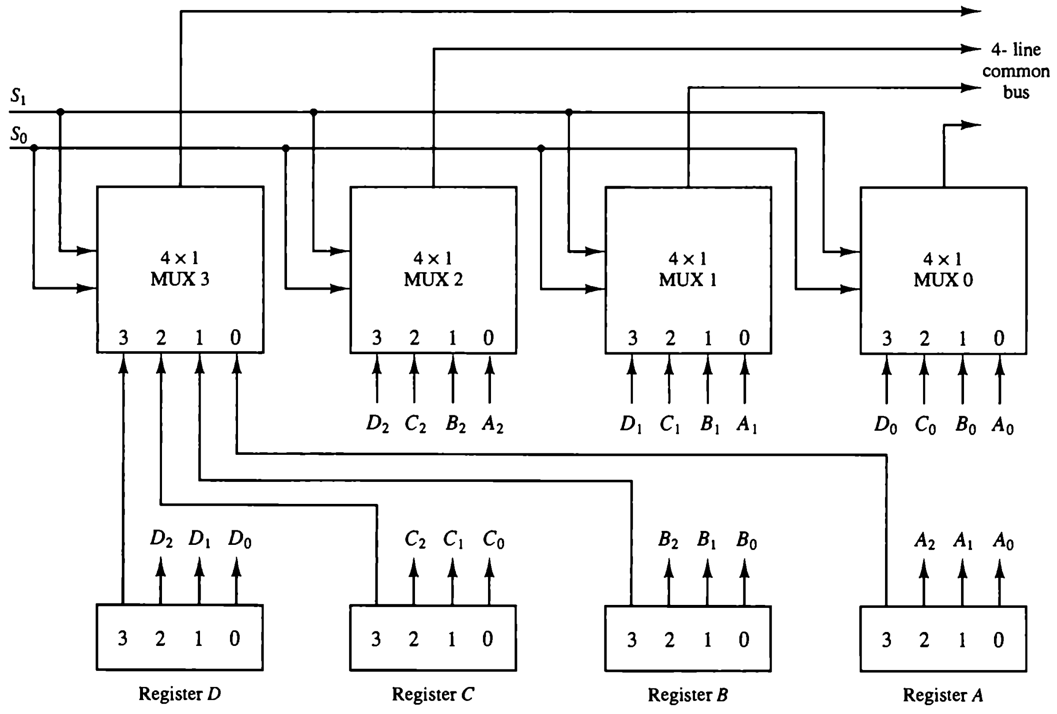


Figure 4 Bus system for four registers.

For more details of Figure 4, shows M. Morris Mano 3rd ed. p98.

In general, a bus system will multiplex k registers of n bits each to produce an n-line common bus. The number of multiplexers needed to construct the bus is equal to n, the number of bits in each register. The size of each multiplexer must be A: x 1 since it multiplexes k data lines. For example, a common bus for eight registers of 16 bits each requires 16 multiplexers, one for each line in the bus. Each multiplexer must have eight data input lines and three selection lines to multiplex one significant bit in the eight registers.

The transfer of information from a bus into one of many destination registers can be accomplished by connecting the bus lines to the inputs of all destination registers and activating the load control of the particular destination register selected. The symbolic statement for a bus transfer may mention the bus or its presence may be implied in the statement. When the bus is includes in the statement, the register transfer is symbolized as follows:

$$\text{BUS} \leftarrow \text{C}, \text{R1} \leftarrow \text{BUS}$$

The content of register C is placed on the bus, and the content of the bus is loaded into register Rl by activating its load control input. If the bus is known to exist in the system, it may be convenient just to show the direct transfer.

$$R1 \leftarrow C$$

From this statement the designer knows which control signals must be activated to produce the transfer through the bus.

Three-State Bus Buffers

A bus system can be constructed with three-state gates instead of multiplexers. A three-state gate is a digital circuit that exhibits three states. Two of the states are signals equivalent to logic 1 and 0 as in a conventional gate. The third state is a high-impedance state. The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have a logic significance.

The one most commonly used in the design of a bus system is the buffer gate. The graphic symbol of a three-state buffer gate is shown in Figure 5.

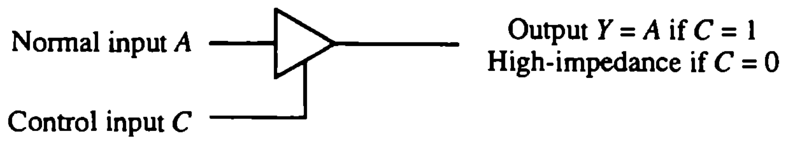


Figure 5 Graphic symbols for three-state buffer

The construction of a bus system with three-state buffers is demonstrated in Fig. (6). The outputs of four buffers are connected together to form a single bus line. (It must be realized that this type of connection cannot be done with gates that do not have three-state outputs.) The control inputs to the buffers determine which of the four normal inputs will communicate with the bus line.

Careful investigation will reveal that Figure 6 is another way of constructing a 4 x 1 multiplexer since the circuit can replace the multiplexer in Figure 4.

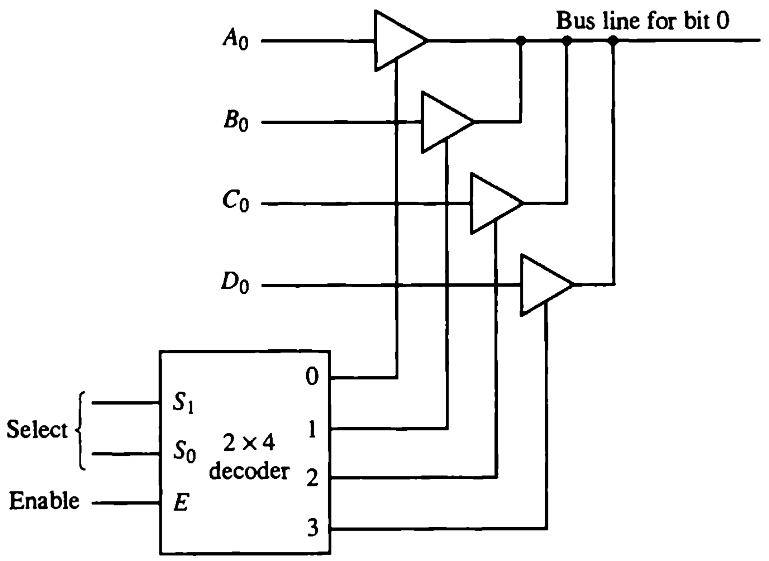


Figure 6 Bus line with three state-buffers.

H.W. Draw the block diagram for hardware that implements the following statements. (Each has four bits)

if (x=1) then R2 ← R1

else if (y=1) then R2 ← R3

Memory Transfer

The transfer of information from a memory word to the outside environment is called a read operation. The transfer of new information to be stored into the memory is called a write operation.

The read operation can be stated as follows:

Read: DR ← M[AR]

Where DR is called the Data register that the data transferred to it. AR is called the address register that the memory word M receives the address from it.

The write operation can be stated symbolically as follows

Write: M[AR] ← R1

This causes a transfer of information from R1 into the memory word M selected by the address in AR.

Arithmetic Micro operations

A micro operation is an elementary operation performed with the data stored in registers. The micro operations most often encountered in digital computers are classified into four categories:

1- Register transfer micro operations transfers binary information from one register to another.
2- Arithmetic micro operations perform arithmetic operations on numeric data stored in registers.
3- Logic micro operations perform bit manipulation operations on non-numeric data stored in registers.
4- Shift micro operations perform shift operations on data stored in registers.

The basic arithmetic micro operations are addition, subtraction, increment, decrement, and shift. Arithmetic shift are explained later in conjunction with the shift micro operations. The arithmetic micro operation defined by the statement

$$R3 \leftarrow R1 + R2$$

Specified an add micro operation. It states that the contents of register R1 are added to the contents of register R2 and the sum transferred to register R3. To implement this statement with hardware we need three registers and the digital component that performs the addition operation. The other basic micro operations are listed in table (1). Subtraction is most often implemented through complementation and addition. Instead of using the minus operator, we can specify the subtraction by the following statement:

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

R2 is the symbol for the 1's complement of R2. Adding 1 to the 1's complement produces the 2's complement. Adding the contents of R1 to the 2's complement of R2 is equivalent to R1 - R2.

Table 1 Arithmetic micro operations

Symbolic designation	Description
$R3 \leftarrow R1 + R2$	Contents of R1 plus R2 transferred to R3
$R3 \leftarrow R1 - R2$	Contents of R1 minus R2 transferred to R3
$R2 \leftarrow \overline{R2}$	Complement the contents of R2 (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement the contents of R2 (negate)
$R3 \leftarrow R1 + \overline{R2} + 1$	R1 plus the 2's complement of R2 (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of R1 by one
$R1 \leftarrow R1 – 1$	Decrement the contents of R1 by one

In such a case, the signals that perform these operations propagate through gates, and the result of the operation can be transferred into a destination register by a clock pulse as soon as the output signal propagates through the combinational circuit. In most computers, the multiplication operation is implemented with a sequence of add and shift micro operations. Division is implemented with a sequence of subtract and shift micro operations.

Binary Adder

The binary adder is constructed with full-adder circuits connected in cascade, with the output carry from one full-adder connected to the input carry of the next full-adder. Figure 7 shows the interconnections of four full-adders (FA) to provide a 4-bit binary adder. The augend bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the low-order bit. The carries are connected in a chain through the full-adders.

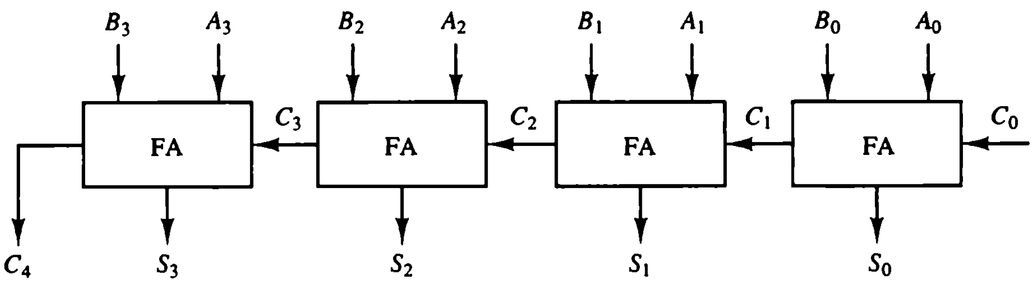


Figure 7 : 4-bit binary adder.

H.W. By using Two ICs of 4-bit binary adder, design a 8-bit binary adder.

Binary Adder-Subtractor

The addition and subtraction operations can be combined into one common circuit by including an exclusive-OR gate with each full-adder. A 4-bit adder-subtractor circuit is shown in Figure 8. The mode input M controls the operation. When M = 0 the circuit is an adder and when M = 1 the circuit becomes a subtractor.

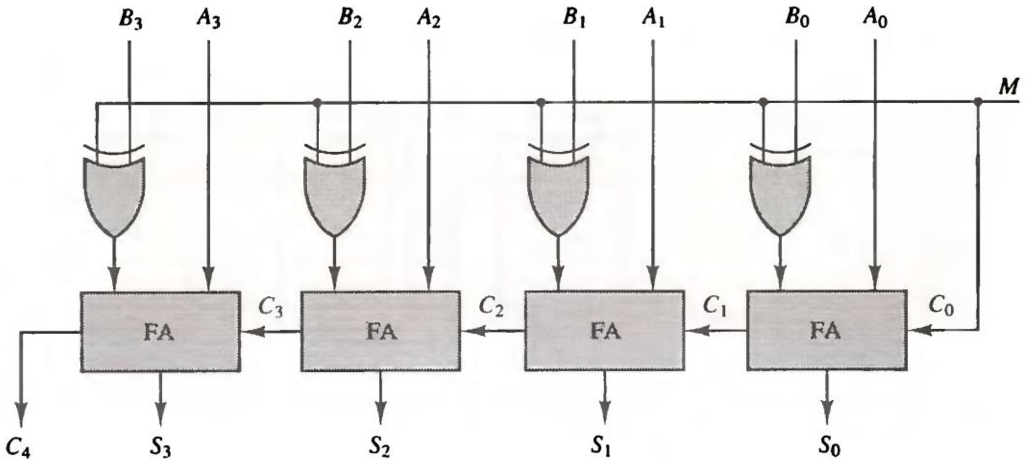


Figure 8 : 4-bit adder-subtractor.

Binary Incrementer

The diagram of a 4-bit combinational circuit incrementer is shown in Figure 9. One of the inputs to the least significant half-adder (HA) is connected to logic-1 and the other input is connected to the least significant bit of the number to be incremented. The output carry from one half-adder is connected to one of the inputs of the next-higher-order half-adder, the circuit receives the four bits from Aq through A3, adds one to it, and generates the incremented output in So through S3. The output carry C4 will be 1 only after incrementing binary 1111. This also causes outputs S0 through S3 to go to 0.

The circuit of Figure 9 can be extended to an n-bit binary incrementer by extending the diagram to include n half-adders. The least significant bit must have one input connected to logic-1. The other inputs receive the number to be incremented or the carry from the previous stage.

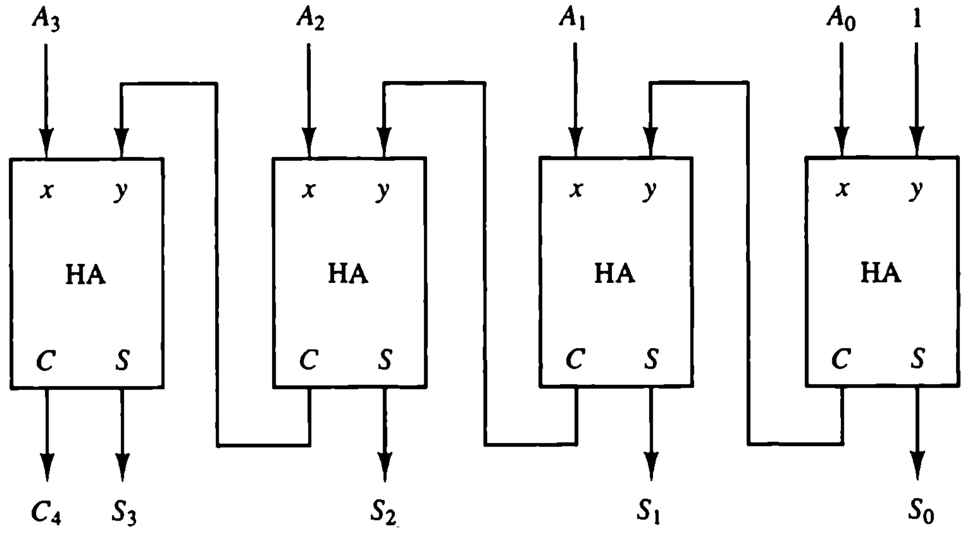


Figure 9 : 4-bit binary incrementer.

Arithmetic Circuit

The arithmetic micro operations listed in Table 1 can be implemented in one composite arithmetic circuit. The basic component of an arithmetic circuit is the parallel adder. By controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations. The diagram of a 4-bit arithmetic circuit is shown in Figure 10.

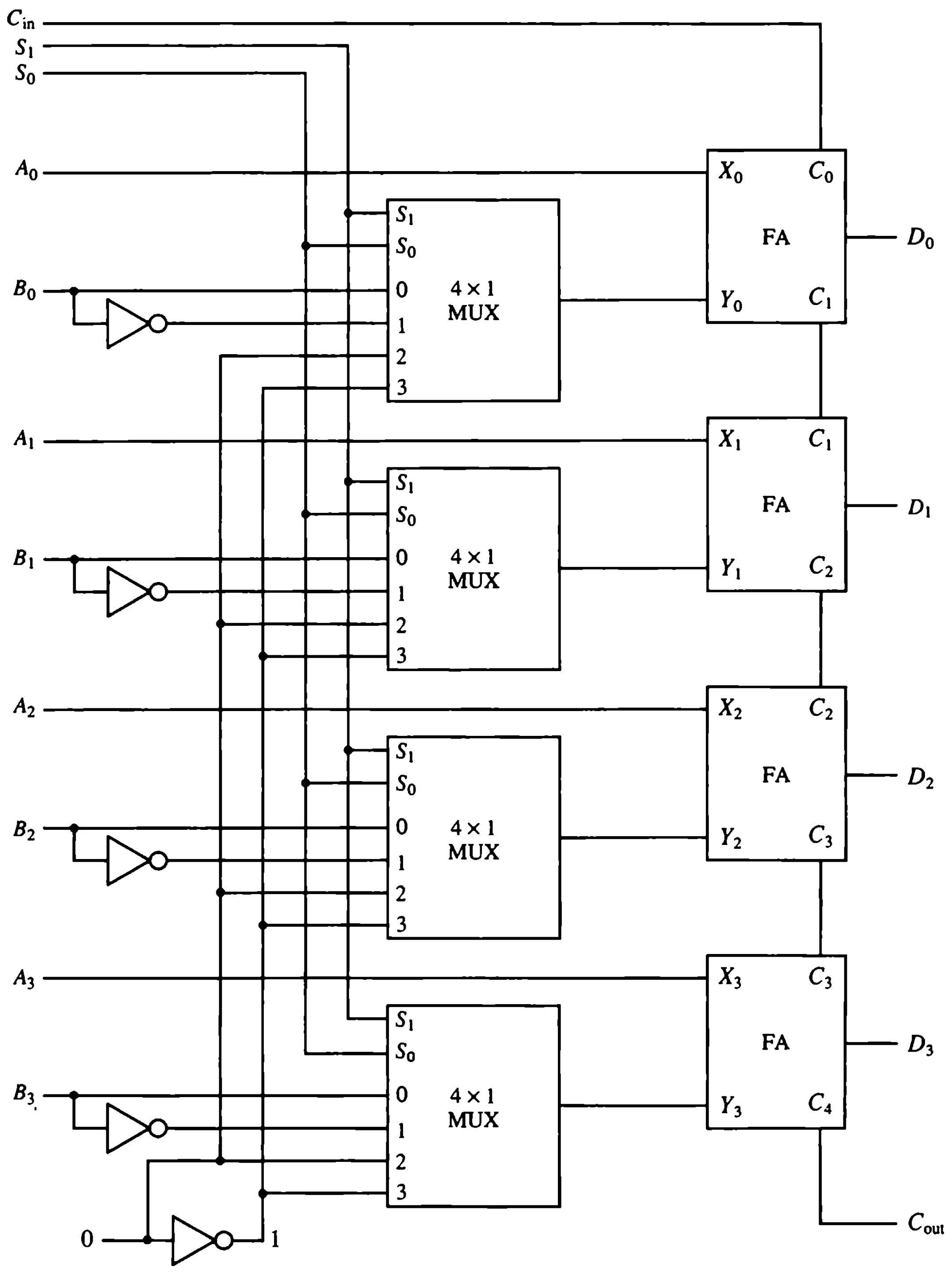


Figure 10 : 4-bit arithmetic circuit.

For more details are given about Figure 10, see M. Morris Mano 3rd ed. p106-108.

Logic Micro operations

Logic micro operations specify binary operations for strings of bits stored in registers. These operations consider each bit of the register separately and treat them as binary variables. For example, the exclusive-OR micro operation with the contents of two registers R1 and R2 is symbolized by the statement

$$P:\quad R1 \leftarrow R1 \oplus R2$$

The content of R1, after the execution of the micro operation, is equal to the bit-by-bit exclusive-OR operation on pairs of bits in R2 and previous values of R1.

Special symbols will be adopted for the logic micro operations OR, AND, and complement, to distinguish them from the corresponding symbols used to express Boolean functions. The symbol V will be used to denote an OR micro operation and the symbol Λ to denote an AND micro operation. The complement micro operation is the same as the 1's complement and uses a bar on top of the symbol that denotes the register name. By using different symbols, it will be possible to differentiate between a logic micro operation and a control (or Boolean) function. Another reason for adopting two sets of symbols is to be able to distinguish the symbol + , when used to symbolize an arithmetic plus, from a logic OR operation. Although the + symbol has two meanings, it will be possible to distinguish between them by noting where the symbol occurs. When the symbol + occurs in a micro operation, it will denote an arithmetic plus. When it occurs in a control (or Boolean) function, it will denote an OR operation. We will never use it to symbolize an OR micro operation. For example, in the statement

$$P + Q: R1 \leftarrow R2 + R3, R4 \leftarrow R5 \text{ V } K6$$

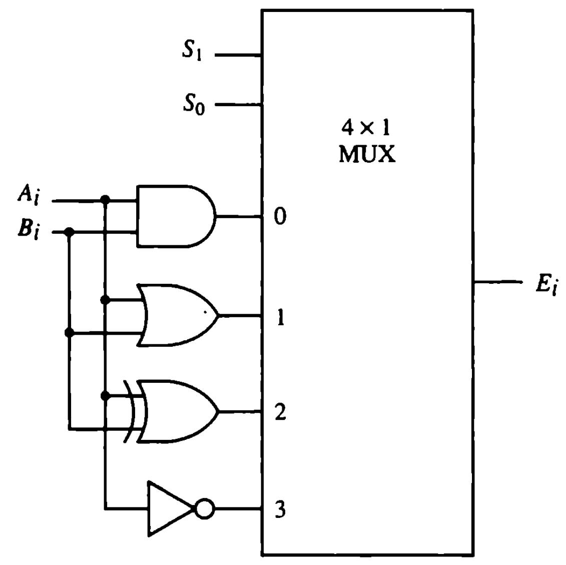
the + between P and Q is an OR operation between two binary variables of a control function. The + between R2 and R3 specifies an add micro operation. The OR micro operation is designated by the symbol V between registers R5 and R6. There are sixteen logic micro operations listed in Table 2.

Table 2 Sixteen Logic Micro operations

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR
$F_8 = (x + y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \overline{B}$	Complement B
$F_{11} = x + y'$	$F \leftarrow A \vee \overline{B}$	
$F_{12} = x'$	$F \leftarrow \overline{A}$	Complement A
$F_{13} = x' + y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow$ all 1's	Set to all 1's

Hardware Implementation

The hardware implementation of logic micro operations requires that logic gates be inserted for each bit or pair of bits in the registers to perform the required logic function. Although there are 16 logic micro operations, most computers use only four—AND, OR, XOR (exclusive-OR), and complement— from which all others can be derived. Figure 11 shows one stage of a circuit that generates the four basic logic micro operations. The diagram shows one typical stage with subscript i. For a logic circuit with n bits, the diagram must be repeated n times for i = 0, 1, 2, ..., n - 1. The selection variables are applied to all stages. The function table in Figure 11 (b) lists the logic micro operations obtained for each combination of the selection variables.



(a) Logic diagram

S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \bar{A}$	Complement

(b) Function table

Figure 11 One stage of logic circuit.

Some Applications

Logic micro operations are very useful for manipulating individual bits or a portion of a word stored in a register. They can be used to change bit values, delete a group of bits, or insert new bit values into a register. The following examples show how the bits of one register (designated by A) are manipulatedby logic micro operations as a function of the bits of another register (designated by B). In a typical application, register A is a processor register and the bits of register B constitute a logic operand extracted from memory and placed in register B.

1- Selective-set

The selective-set operation sets to 1 the bits in register A where there are corresponding l's in register B. It does not affect bit positions that have 0's in B. The following numerical example clarifies this operation:

```
1010    A before
1100    B (logic operand)
-----
1110    A after
```

The two leftmost bits of B are 1's, so the corresponding bits of A are set to 1. One of these two bits was already set and the other has been changed from 0 to 1. The two bits of A with corresponding 0's in B remain unchanged. The example above serves as a truth table since it has all four possible combinations of two binary variables. From the truth table we note that the bits of A after the operation are obtained from the logic-OR operation of bits in

B and previous values of A. Therefore, the OR micro operation can be used to selectively set bits of a register.

2- Selective-complement

The selective-complement operation complements bits in A where there are corresponding 1's in B. It does not affect bit positions that have 0's in B. For example:

```
1010  A before
1100 B (logic operand)
-----
0110 A after
```

Again the two leftmost bits of B are 1's, so the corresponding bits of A are complemented. This example again can serve as a truth table from which one can deduce that the selective-complement operation is just an exclusive-OR micro operation. Therefore, the exclusive-OR micro operation can be used to selectively complement bits of a register.

3- selective-clear

The selective-clear operation clears to 0 the bits in A only where there are corresponding 1's in B. For example:

```
1010      A before
1100      B (logic operand)
-----
0010      A after
```

Again the two leftmost bits of B are 1's, so the corresponding bits of A are cleared to 0. One can deduce that the Boolean operation performed on the individual bits is AB'. The corresponding logic micro operation is

$$A \leftarrow A \wedge \bar{B}$$

The mask operation is similar to the selective-clear operation except that the bits of A are cleared only where there are corresponding 0's in B. The mask operation is an AND micro operation as seen from the following numerical example:

```
1010      A before
1100      B (logic operand)
-----
1000      A after masking
```

The two rightmost bits of A are cleared because the corresponding bits of B are 0's. The two leftmost bits are left unchanged because the corresponding bits of B are 1's. The mask operation is more convenient to use than the selective-clear operation because most computers provide an AND instruction, and few provide an instruction that executes the micro operation for selective-clear.

The insert operation inserts a new value into a group of bits. This is done by first masking the bits and then ORing them with the required value. For example, suppose that an A register contains eight bits, 0110 1010. To replace the four leftmost bits by the value 1001 we first mask the four unwanted bits:

```
0110 1010           A before
0000 1111           B (mask)
------------
0000 1010           A after masking
```

and then insert the new value:

```
0000 1010           A before
1001 0000           B (insert)
------------
1001 1010           A after insertion
```

The mask operation is an AND micro operation and the insert operation is an OR micro operation.

The clear operation compares the words in A and B and produces an all 0's result if the two numbers are equal. This operation is achieved by an exclusive-OR micro operation as shown by the following example:

1010 A
1010 B

0000 $A \leftarrow A \oplus B$

When A and B are equal, the two corresponding bits are either both 0 or both 1. In either case the exclusive-OR operation produces a 0. The all-0's result is then checked to determine if the two numbers were equal.

Shift Micro operations

There are three types of shifts: logical, circular, and arithmetic.

1- A logical shift is one that transfers 0 through the serial input. We will adopt

the symbols shl and shr for logical shift-left and shift-right micro operations. For example:

$$R1 \leftarrow \text{shl } R1$$

$$R2 \leftarrow \text{shr } R2$$

2- The circular shift (also known as a rotate operation) circulates the bits of the register around the two ends without loss of information. This is accomplished by connecting the serial output of the shift register to its serial input. We will use the symbols cil and cir for the circular shift left and right, respectively.

3- An arithmetic shift is a micro operation that shifts a signed binary number to the left or right. An arithmetic shift-left multiplies a signed binary number by 2. An arithmetic shift-right divides the number by 2. Arithmetic shifts must leave the sign bit unchanged because the sign of the number remains the same when it is multiplied or divided by 2. The leftmost bit in a register holds the sign bit, and the remaining bits hold the number. The sign bit is 0 for positive and 1 for negative. Negative numbers are in 2's complement form. Figure 12 shows a typical register of n bits. Bit R_{n-1} in the leftmost position holds the sign bit. R_{n-2} is the most significant bit of the number and R_0 is the least significant bit. The arithmetic shift-right leaves the sign bit unchanged and shifts the number (including the sign bit) to the right. Thus R_{n-1} remains the same, R_{n-2} receives the bit from R_{n-1} and so on for the other bits in the register. The bit in R_0 is lost.

   The arithmetic shift-left inserts a 0 into R_0, and shifts all other bits to the left. The initial bit of R_{n-1} is lost and replaced by the bit from R_{n-2}. A sign reversal occurs if the bit in R_{n-1} changes in value after the shift. This happens if the multiplication by 2 causes an overflow. An overflow occurs after an arithmetic shift left if initially, before the shift, R_{n-1} is not equal to R_{n-2}. An overflow flip-flop V_s can be used to detect an arithmetic shift-left overflow.

$$V_s = R_{n-1} \oplus R_{n-2}$$

   If $V_s = 0$, there is no overflow, but if $V_s = 1$, there is an overflow and a sign reversal after the shift. V_s must be transferred into the overflow flip-flop with the same clock pulse that shifts the register.

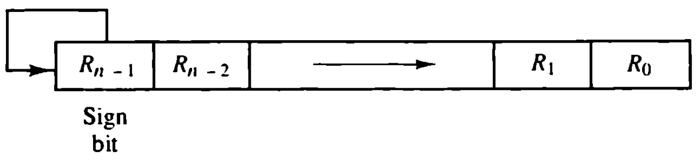


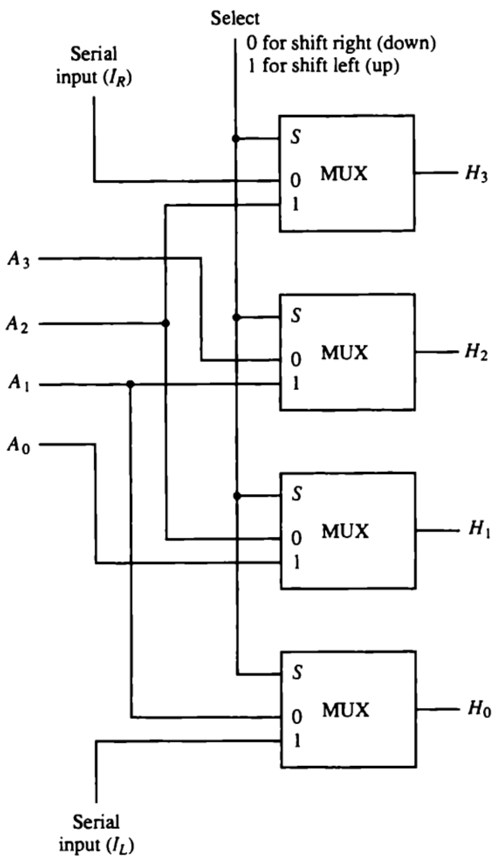
Figure 12 Arithmetic shift right.

The symbolic notation for the shift micro operations is shown in Table 3.

Table 3 Shift Micro operations

Symbolic designation	Description
R ← shl R	Shift-left register R
R ← shr R	Shift-right register R
R ← cil R	Circular shift-left register R
R ← cir R	Circular shift-right register R
R ← ashl R	Arithmetic shift-left R
R ← ashr R	Arithmetic shift-right R

A combinational circuit shifter can be constructed with multiplexers as shown in Figure 13.



Function table

Select	Output			
S	H_3	H_2	H_1	H_0
0	I_R	A_3	A_2	A_1
1	A_2	A_1	A_0	I_L

Figure 13 : 4-bit combinational circuit shifter.

Arithmetic Logic Shift Unit

The arithmetic, logic, and shift circuits introduced in previous sections can be combined into one ALU with common selection variables. One stage of an arithmetic logic shift unit is shown in Figure 14. The subscript i designates a typical stage.

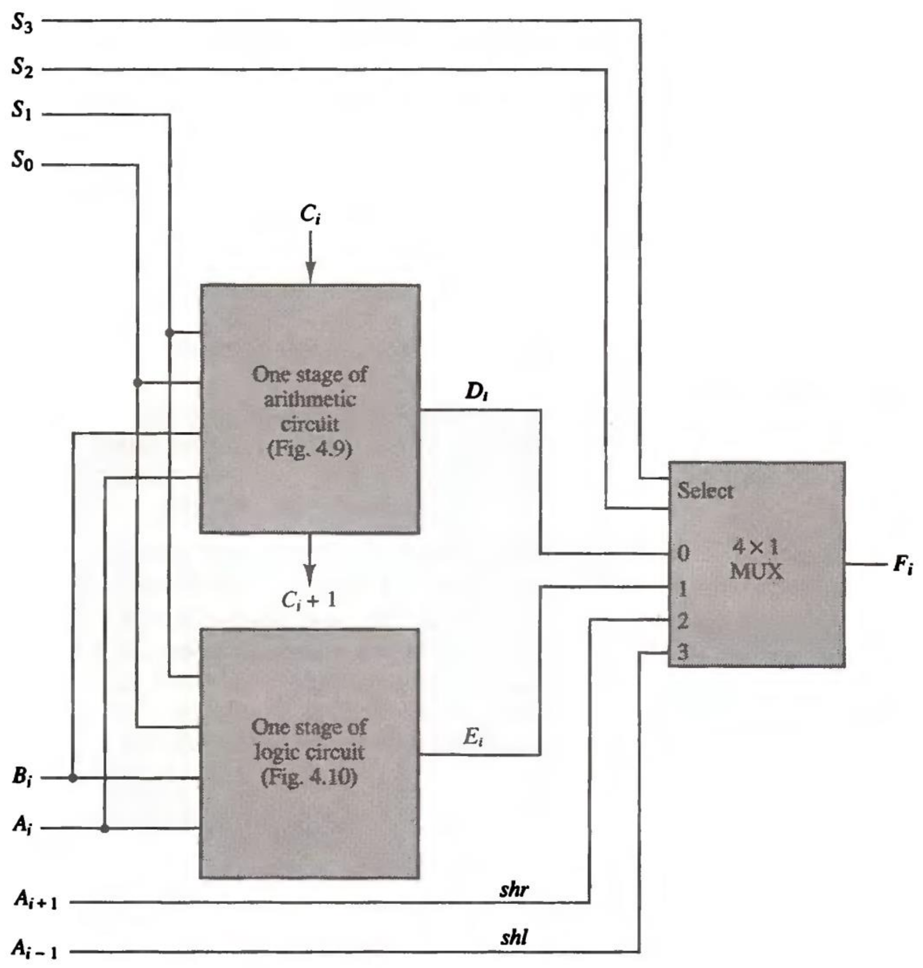


Figure 14 One stage of arithmetic logic shift unit.

The circuit whose one stage is specified in Figure 14 provides eight arithmetic operation, four logic operations, and two shift operations. Each operation is selected with the five variables S_3, S_2, S_1, S_0, and C_{in}. The input carry C_{in} is used for selecting an arithmetic operation only. These 14 operations of ALU are listed in Table 5.

Table 4 Function Table for Arithmetic Logic Shift Unit

Operation select						
S_3	S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + \overline{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \overline{B} + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	×	$F = A \wedge B$	AND
0	1	0	1	×	$F = A \vee B$	OR
0	1	1	0	×	$F = A \oplus B$	XOR
0	1	1	1	×	$F = \overline{A}$	Complement A
1	0	×	×	×	$F = \text{shr } A$	Shift right A into F
1	1	×	×	×	$F = \text{shl } A$	Shift left A into F

H.W. Draw the block diagram with one adder unit to implements the following statements:

X: R1 ← R2+R3

Y: R2 ← R1+R3

Z: R3 ← R1+R2

Note that the variables X, Y and Z are mutually exclusive, which means that only one variable is equal to 1 at any given time, while the others are equal to 0.

H.W. What is the wrong if it is found for each statement below:

X: R1 ← R2, R2 ← R3, R3 ← R1

X: R1 ← R2, R3 ← R2, R2 ← R3

X: R1 ← R2, R2 ← R3, R1 ← R3

Basic Computer Organization and Design

This lecture introduces a basic computer and shows how its operation can be specified with register transfer statements. The organization of the computer is defined by its internal registers, the timing and control structure, and the set of instructions that it uses. The design of the computer is then carried out in detail. The basic computer presented in this chapter is very small compared to commercial computers.

The general purpose digital computer is capable of executing various microoperations. An instruction code is a group of bits that instruct the computer to perform a specific operation.

The most basic part of an instruction code is its operation part. The operation code of an instruction is a group of bits that define such operations as add, subtract, multiply, shift, and complement.

The control unit receives the instruction from memory and interprets the operation code bits. It then issues a sequence of control signals to initiate microoperations in internal computer registers.

Stored Program Organization

The simplest way to organize a computer is to have one processor register and an instruction code format with two parts. The first part specifies the operation to be performed and the second specifies an address.

Figure 1 depicts this type of organization. Instructions are stored in one section of memory and data in another. For a memory unit with 4096 words we need 12 bits to specify an address since $2^{12} = 4096$.

Indirect Address

It is sometimes convenient to use the address bits of an instruction code not as an address but as the actual operand. When the second part of an instruction code specifies an operand, the instruction is said to have an immediate operand. When the second part specifies the address of an operand, the instruction is said to have a direct address. This is in contrast to a third possibility called indirect address, where the bits in the second part of the instruction designate an address of a memory word in which the address of the operand is found. One bit of the instruction code can be used to distinguish between a direct and an indirect address. Figure 2 shows direct and indirect address. Where the mode bit (I) in the instruction format refers to direct address if it is 0 and indirect address if it is 1.

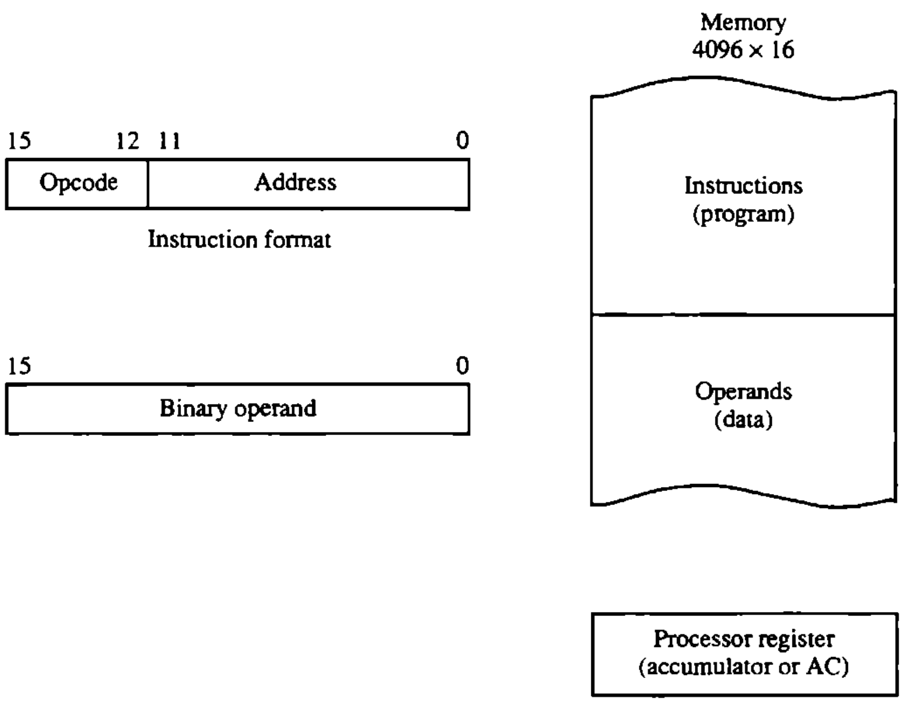


Figure 15 Stored program organization.

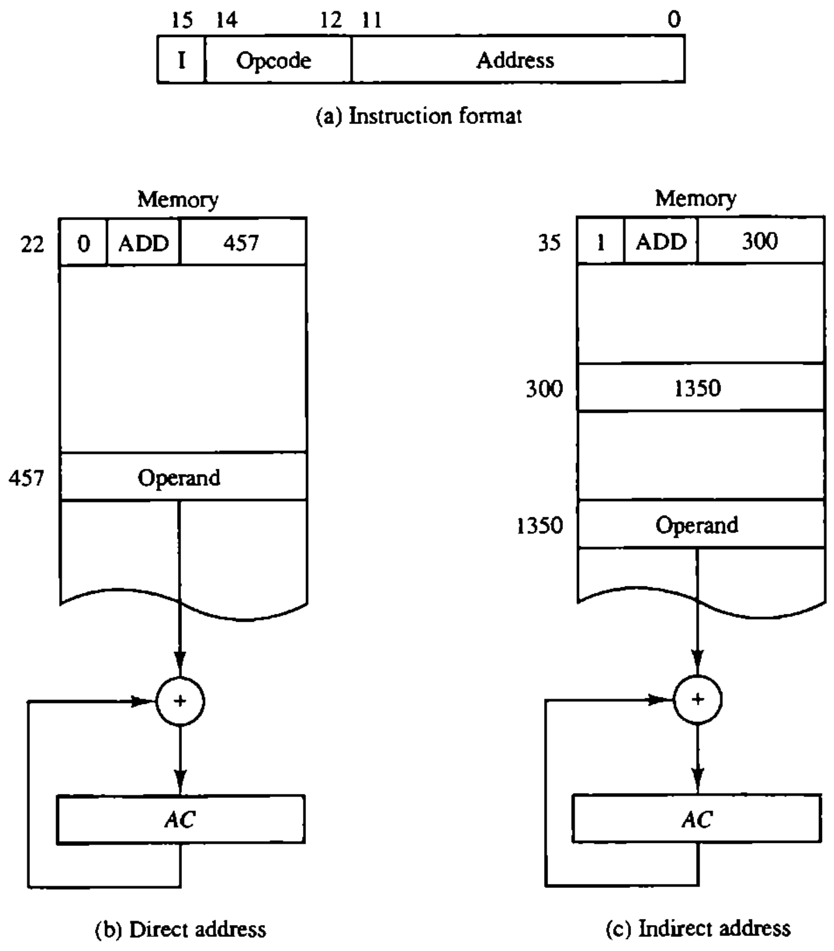


Figure 16 Demonstration of direct and indirect address.

Computer Registers

The registers are needed in the basic computer are listed in Table 1 together with a brief description of their function and the number of bits that they contain.

Table 5 List of Registers for the Basic Computer

Register symbol	Number of bits	Register name	Function
DR	16	Data register	Holds memory operand
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
PC	12	Program counter	Holds address of instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds input character
OUTR	8	Output register	Holds output character

The size of registers and memory are shown in figure 3.

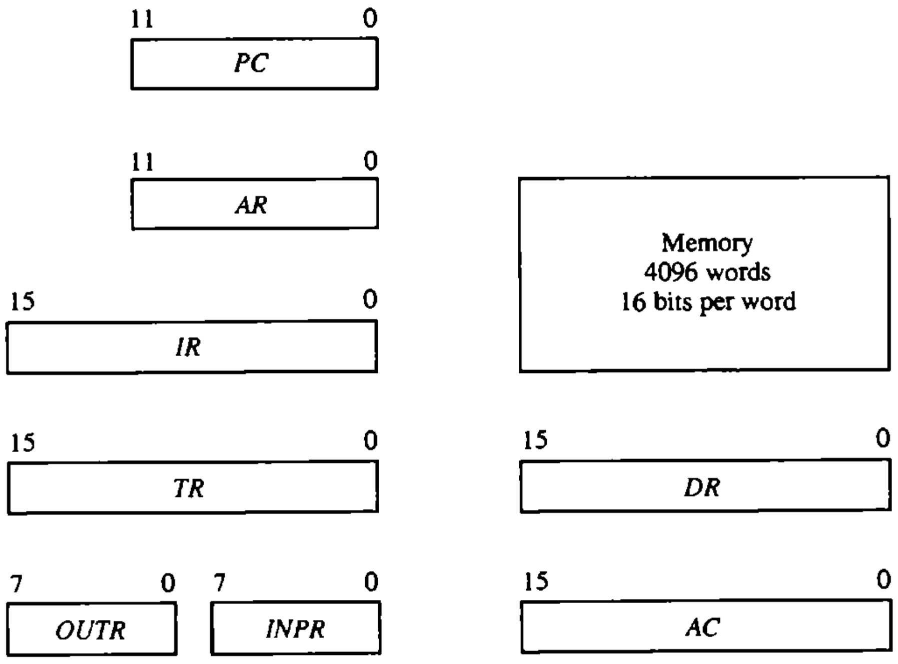


Figure 17 Basic computer registers and memory.

Common Bus System

The basic computer has eight registers, a memory unit, and a control unit. Paths must be provided to transfer information from one register to another and between memory and registers. The number of wires will be excessive if connections are made between the outputs of each register and the inputs of the other registers. A more efficient scheme for transferring information in a system with many registers is to use a common bus. The connection of the registers and memory of the basic computer to a common bus system is shown in Figure 4.

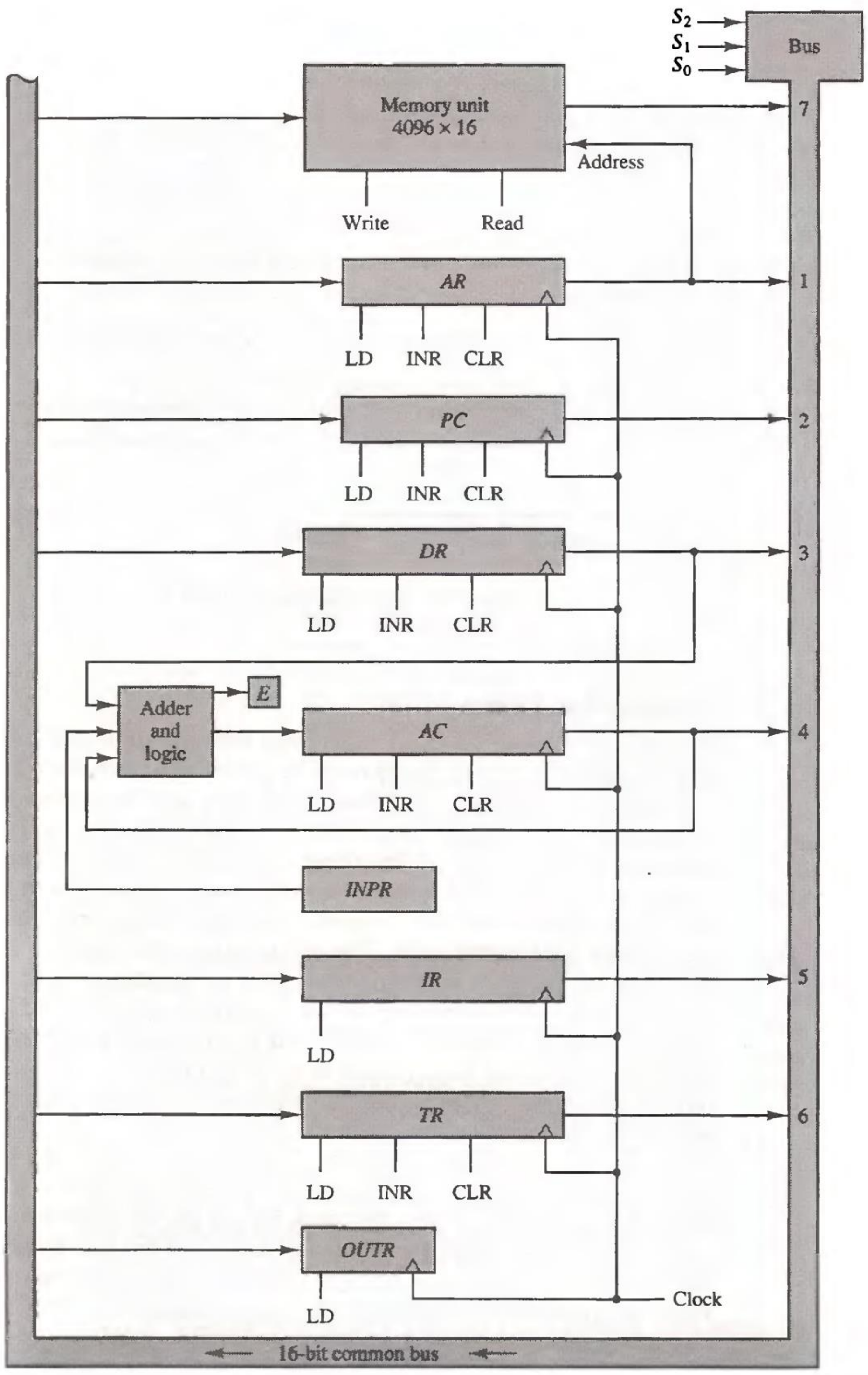


Figure 18 Basic computer registers connected to a common bus.

The content of a register can be applied onto the bus and an operation can be performed in the adder and logic circuit during the same clock cycle. The clock transition at the end of the cycle transfers the content of the bus into the designated destination register and the output of the adder and logic circuit into AC. For example, the two microoperations

$$DR \leftarrow AC \text{ and } AC \leftarrow DR$$

can be executed at the same time. This can be done by placing the content of AC on the bus (with $S_2S_1S_0 = 100$), enabling the LD (load) input of DR, transferring the content of DR through the adder and logic circuit into AC, and enabling the LD (load) input of AC, all during the same clock cycle. The two transfers occur upon the arrival of the clock pulse transition at the end of the clock cycle.

H.W. Can the two microoperations DR ← DR+1 and AC ← DR executed at the same time.

Computer Instructions

The basic computer has three instruction code formats, as shown in Figure 5. The type of instruction is recognized by the computer control from the four bits in positions 12 through 15 of the instruction. If the three opcode bits in positions 12 though 14 are not equal to 111, the instruction is a memory-reference type and the bit in position 15 is taken as the addressing mode J. If the 3-bit opcode is equal to 111, control then inspects the bit in position 15. If this bit is 0, the instruction is a register-reference type. If the bit is 1, the instruction is an input-output type. Note that the bit in position 15 of the instruction code is designated by the symbol J but is not used as a mode bit when the operation code is equal to 111.

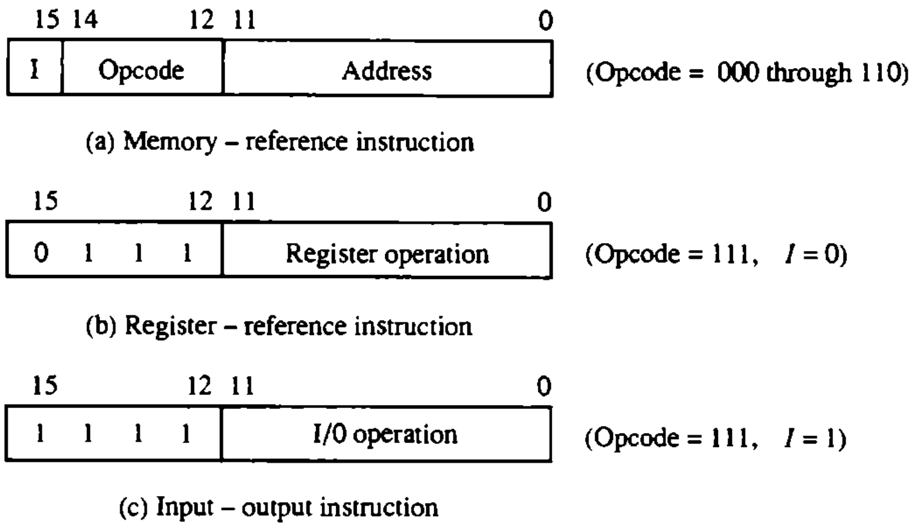


Figure 19 Basic computer instruction formats.

The instructions for the computer are listed in Table 2. The symbol designation is a three-letter word and represents an abbreviation intended for programming and users.

Table 6 Basic Computer Instructions

	Hexadecimal code		
Symbol	$I = 0$	$I = 1$	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load memory word to AC
STA	3xxx	Bxxx	Store content of AC in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instruction if AC positive
SNA	7008		Skip next instruction if AC negative
SZA	7004		Skip next instruction if AC zero
SZE	7002		Skip next instruction if E is 0
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

Instruction Set Completeness

The set of instructions are said to be complete if the computer includes a sufficient number of instructions in each of the following categories:

1. Arithmetic, logical, and shift instructions
2. Instructions for moving information to and from memory and processor registers
3. Program control instructions together with instructions that check status conditions
4. Input and output instructions

The set of instructions for the basic computer listed in table 2. It is not efficient because frequently used operations are not performed rapidly. An efficient set of instructions will include such instructions as subtract, multiply, OR, and exclusive-OR. These operations must be programmed in the basic computer. By using a limited number of instructions it is possible to show the detailed logic design of the computer. A more complete set of instructions would have made the design too complex. In this way it can demonstrate the basic principles of computer organization and design without going into excessive complex details.

Timing and Control

The timing for all registers in the basic computer is controlled by a master clock generator. The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit. The clock pulses do not change the state of a register unless the register is enabled by a control signal. The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and microoperations for the accumulator.

There are two major types of control organization: hardwired control and microprogrammed control. In the hardwired organization, the control logic is implemented with gates, flip-flops, decoders, and other digital circuits. It has the advantage that it can be optimized to produce a fast mode of operation. In the microprogrammed organization, the control information is stored in a control memory. The control memory is programmed to initiate the required sequence of microoperations. A hardwired control, as the name implies, requires changes in the wiring among the various components if the design has to be modified or changed. In the microprogrammed control, any required control changes or modifications can be done by updating the microprogram in control memory. A hardwired control for the basic computer is presented in this lecture. A microprogrammed control unit for a similar computer is presented in another lecture.

The block diagram of the control unit is shown in Figure 6. It consists of two decoders, a sequence counter, and a number of control logic gates. An instruction read from memory is placed in the instruction register (IR). The position of this register in the common bus system is indicated in Figure 4. The instruction register is shown again in Figure 6, where it is divided into three parts: the I bit, the operation code, and bits 0 through 11. The operation code in bits 12 through 14 are decoded with a 3 x 8 decoder. The eight outputs of the decoder are designated by the symbols D_0 through D_7. The subscripted decimal number is equivalent to the binary value of the corresponding operation code. Bit 15 of the instruction is transferred to a flip-flop designated by the symbol I. Bits 0 through 11 are applied to the control logic gates. The 4-bit sequence counter can count in binary from 0 through 15. The outputs of the counter are decoded into 16 timing signals T_0 through T_{15}. The internal logic of the control gates will be derived later when it considers the design of the computer in detail.

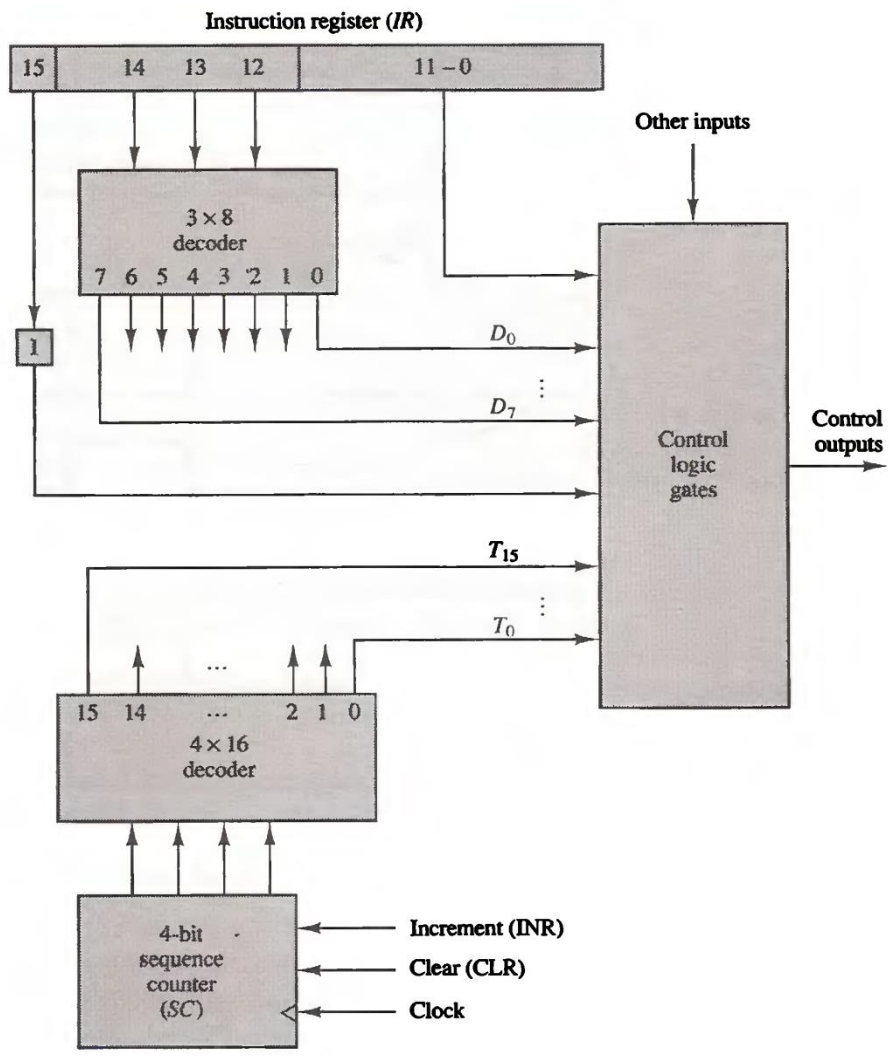


Figure 20 Control unit of basic computer.

The sequence counter SC can be incremented or cleared synchronously. Most of the time, the counter is incremented to provide the sequence of timing signals out of the 4 x 16 decoder. Once in awhile, the counter is cleared to 0, causing the next active timing signal to be T_0. As an example, consider the case where SC is incremented to provide timing signals T_0, T_1, T_2, T_3, and T_4 in sequence. At time T_4, SC is cleared to 0 if decoder output D_3 is active. This is expressed symbolically by the statement

$$D_3T_4: SC \leftarrow 0$$

The timing diagram of Figure 7 shows the time relationship of the control signals.

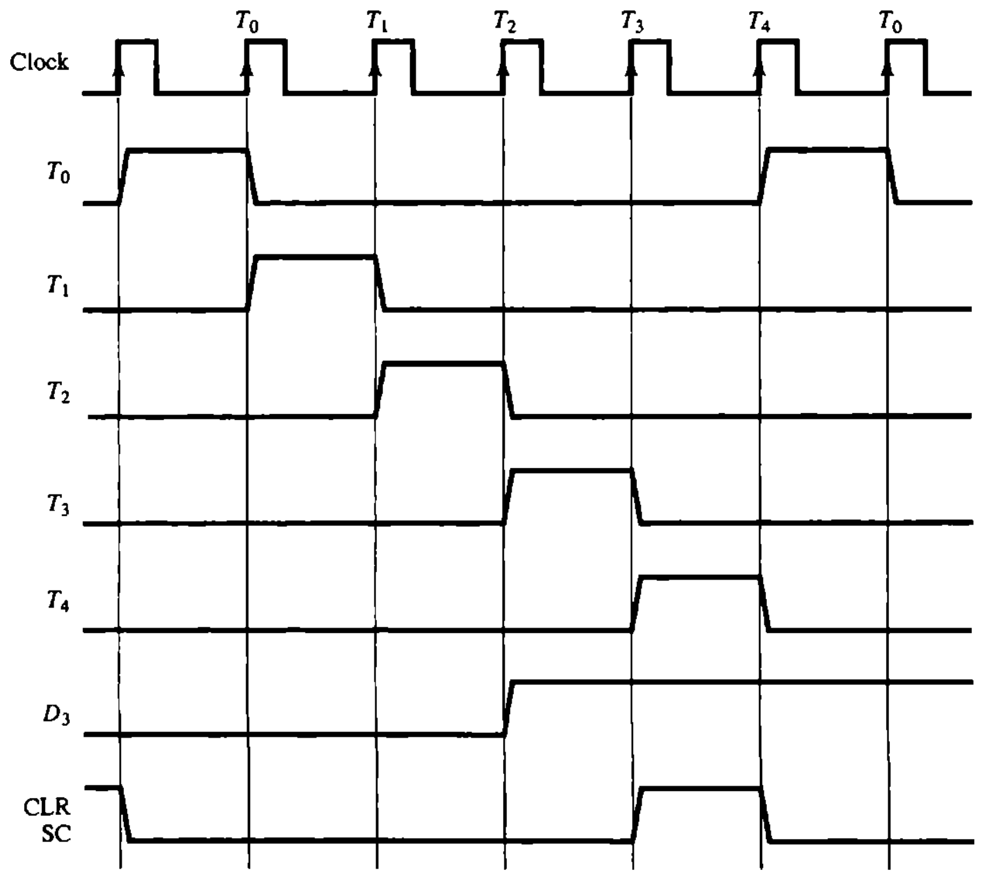


Figure 21 Example of control timing signals.

A memory read or write cycle will be initiated with the rising edge of a timing signal. It will be assumed that a memory cycle time is less than the clock cycle time. According to this assumption, a memory read or write cycle initiated by a timing signal will be completed by the time the next clock goes through its positive transition. The clock transition will then be used to load the memory word into a register. This timing relationship is not valid in many computers because the memory cycle time is usually longer than the processor clock cycle. In such a case it is necessary to provide wait cycles in the processor until the memory word is available. To facilitate the presentation, we will assume that a wait period is not necessary in the basic computer.

To fully comprehend the operation of the computer, it is crucial that one understands the timing relationship between the clock transition and the timing signals. For example, the register transfer statement

$$T_0: AR \leftarrow PC$$

specifies a transfer of the content of PC into AR if timing signal T_0 is active. T_0 is active during an entire clock cycle interval. During this time the content of PC is placed onto the bus (with $S_2S_1S_0 = 010$) and the LD (load) input of AR is enabled. The actual transfer does not occur until the end of the dock cycle when the clock goes through a positive transition. This same positive clock transition increments the sequence counter SC from 0000 to 0001. The next clock cycle has T_1 active and T_0 inactive.

Instruction Cycle

In the basic computer each instruction cycle consists of the following phases:

1. Fetch an instruction from memory.
2. Decode the instruction.
3. Read the effective address from memory if the instruction has an indirect address.

4. Execute the instruction.

Upon the completion of step 4, the control goes back to step 1 to fetch, decode, and execute the next instruction. This process continues indefinitely unless a HALT instruction is encountered.

Fetch and Decode

Initially, the program counter PC is loaded with the address of the first instruction in the program. The sequence counter SC is cleared to 0, providing a decoded timing signal T_0. After each clock pulse, SC is incremented by one, so that the timing signals go through a sequence T_0, T_1, T_2, and so on. The microoperations for the fetch and decode phases can be specified by the following register transfer statements.

T_0: AR←PC

T_1: IR←M[AR], PC←PC + 1

T_2: D_0, ..., D_7←Decode IR(12-14), AR←IR(0-11), I←IR(15)

Figure 8 shows how the first two register transfer statements are implemented in the bus system.

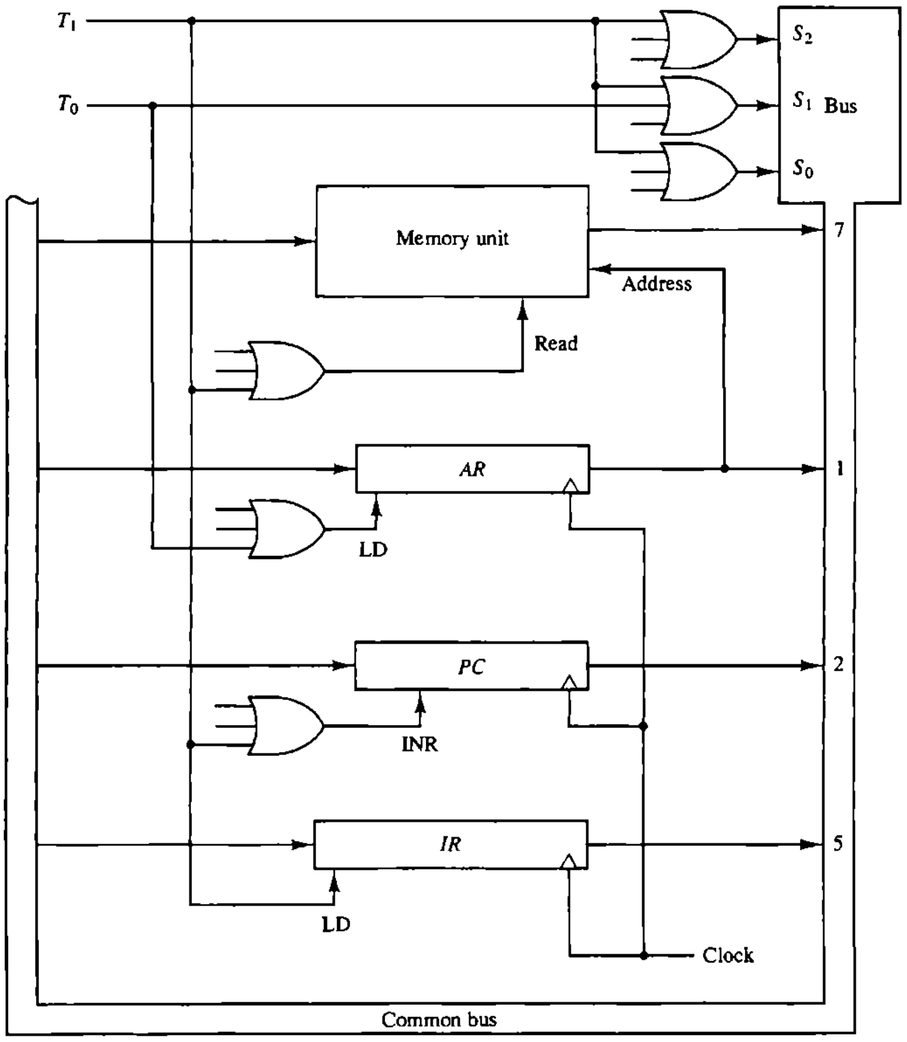


Figure 22 Register transfers for the fetch phase.

For more details about figure 8, shows M. Morris Mano p141.

Determine the Type of Instruction

The timing signal that is active after the decoding is T3. During time T3, the control unit determines the type of instruction that was just read from memory.

The flowchart of Figure 9 presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding.

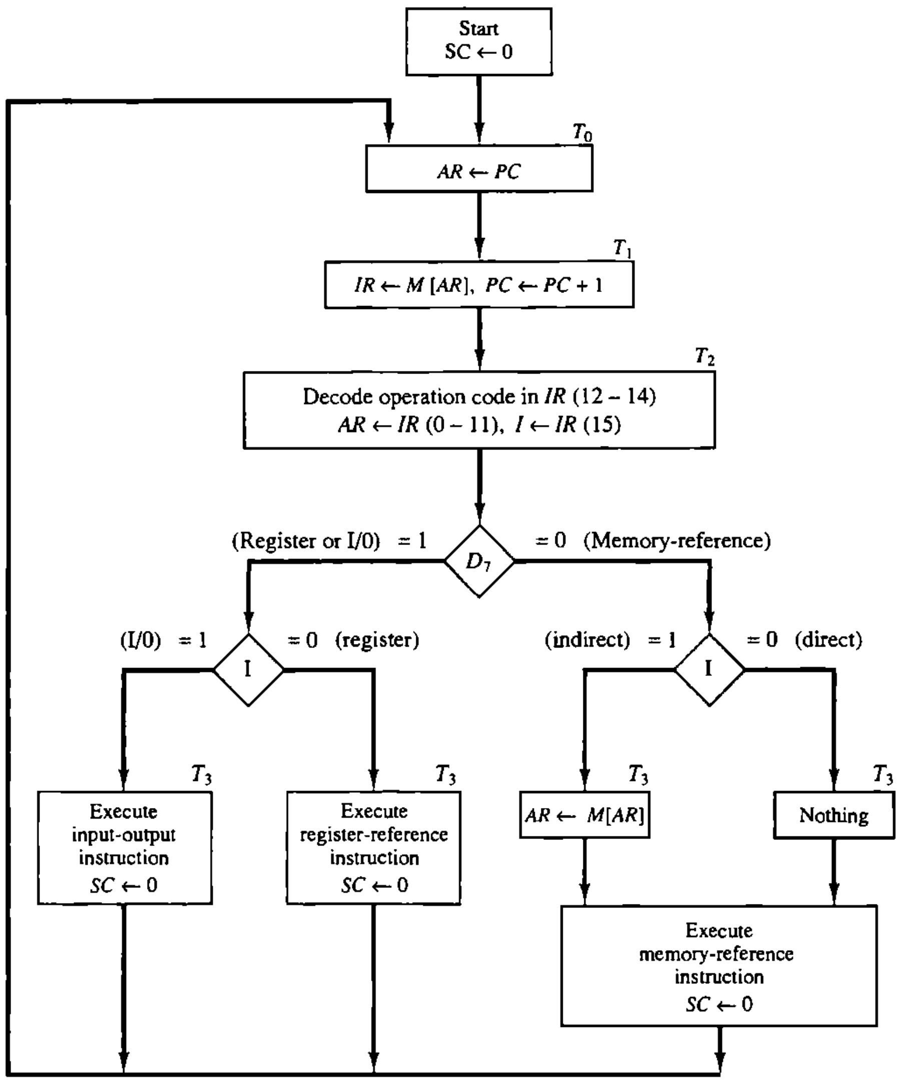

Start
SC ← 0
T_0
AR ← PC
T_1
IR ← M [AR], PC ← PC + 1
T_2
Decode operation code in IR (12 – 14)
AR ← IR (0 – 11), I ← IR (15)
(Register or I/0) = 1
D_7
= 0 (Memory-reference)
(I/0) = 1
I
= 0 (register)
(indirect) = 1
I
= 0 (direct)
T_3
Execute input-output instruction SC ← 0
T_3
Execute register-reference instruction SC ← 0
T_3
AR ← M[AR]
T_3
Nothing
Execute memory-reference instruction SC ← 0

Figure 23 Flowchart for instruction cycle (initial configuration).

The three possible instruction types available in the basic computer are specified in Figure 5.

The three instruction types are subdivided into four separate paths. The selected operation is activated with the clock transition associated with timing signal T3. This can be symbolized as follows:

D_7'IT3: AR←M[AR]

D_7'I'T3: Nothing

D_7I'T3: Execute a register-reference instruction

D_7IT3: Execute an input-output instruction

When a memory-reference instruction with I = 0 is encountered, it is not necessary to do anything since the effective address is already in AR. However, the sequence counter SC must be incremented when $D_7'T_3 = 1$, so that the execution of the memory-reference instruction can be continued with timing variable T_4. A register-reference or input-output instruction can be executed with the clock associated with timing signal T_3. After the instruction is executed, SC is cleared to 0 and control returns to the fetch phase with $T_0 = 1$.

Note that the sequence counter SC is either incremented or cleared to 0 with every positive clock transition. We will adopt the convention that if SC is incremented, we will not write the statement SC$\leftarrow$SC + 1, but it will be implied that the control goes to the next timing signal in sequence. When SC is to be cleared, we will include the statement SC$\leftarrow$0.

Register-Reference Instructions

Register-reference instructions are recognized by the control when D_7=1 and I = 0. These instructions use bits 0 through 11 of the instruction code to specify one of 12 instructions. These 12 bits are available in IR(0-11). They were also transferred to AR during time T_2.

The control functions and microoperations for the register-reference instructions are listed in Table 3. These instructions are executed with the clock transition associated with timing variable T3. Each control function needs the Boolean relation $D_7I'T_3$, which we designate for convenience by the symbol r. The control function is distinguished by one of the bits in IR(0-11). By assigning the symbol B_i to bit i of IR, all control functions can be simply denoted by rB_i.

Table 7 Execution of Register-Reference Instructions

$D_7I'T_3 = r$ (common to all register-reference instructions)
$IR(i) = B_i$ [bit in $IR(0\text{–}11)$ that specifies the operation]

	r:	$SC \leftarrow 0$	Clear SC
CLA	rB_{11}:	$AC \leftarrow 0$	Clear AC
CLE	rB_{10}:	$E \leftarrow 0$	Clear E
CMA	rB_9:	$AC \leftarrow \overline{AC}$	Complement AC
CME	rB_8:	$E \leftarrow \overline{E}$	Complement E
CIR	rB_7:	$AC \leftarrow \text{shr } AC,\ AC(15) \leftarrow E,\ E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow \text{shl } AC,\ AC(0) \leftarrow E,\ E \leftarrow AC(15)$	Circulate left
INC	rB_5:	$AC \leftarrow AC + 1$	Increment AC
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$	Skip if positive
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$	Skip if negative
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1)$	Skip if AC zero
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$	Skip if E zero
HLT	rB_0:	$S \leftarrow 0$ (S is a start–stop flip-flop)	Halt computer

Memory-Reference Instructions

Table 4 lists the seven memory-reference instructions. The decoded output D, for i = 0, 1, 2, 3, 4, 5, and 6 from the operation decoder that belongs to each instruction is included in the table. The effective address of the instruction is in the address register AR and was placed there during timing signal T_2 when I = 0, or during timing signal T_3 when I=1. The execution of the memory-reference instructions starts with timing signal T_4.

Table 8 Memory-Reference Instructions

Symbol	Operation decoder	Symbolic description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR], \quad E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC, \quad PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1$, If $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$

AND to AC

D_0T_4: DR ← M[AR]

D_0T_5: AC ← AC ∧ DR, SC ← 0

ADD to AC

D_1T_4: DR ← M[AR]

D_1T_5: AC ← AC + DR, E ← C_{out} , SC ← 0

LDA: Load to AC

D_2T_4: DR ← M[AR]

D_2T_5: AC ← DR, SC ← 0

STA: Store AC

D_3T_4: M[AR] ← AC, SC ← 0

BUN: Branch Unconditionally

D_4T_4: PC ← AR, SC ← 0

BSA: Branch and Save Return Address

D_5T_4: M[AR] ← PC, AR ← AR+1

D_5T_5: PC ← AR, SC ← 0

A numerical example that demonstrates how this instruction is used with a subroutine is shown in Figure 10.

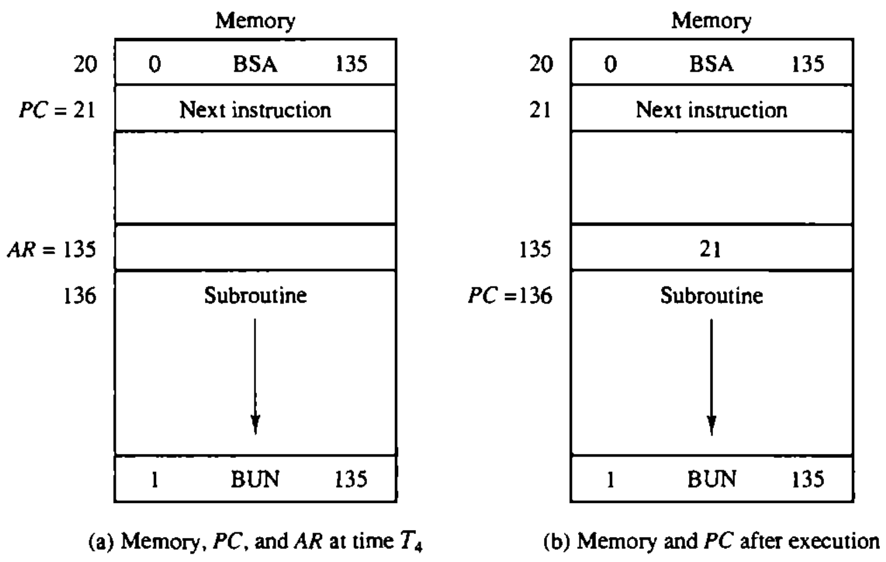


Figure 24 Example of BSA instruction execution.

The BSA instruction performs the function usually referred to as a subroutine call. The indirect BUN instruction at the end of the subroutine performs the function referred to as a subroutine return. In most commercial computers, the return address associated with a subroutine is stored in either a processor register or in a portion of memory called a stack.

H.W. Why BSA instruction dose not implemented as follows:

D_5T_4: M[AR] ← PC, PC ← AR+1, SC ← 0

ISZ: Increment and Skip if Zero

D_6T_4: DR ← M[AR]

D_6T_5: DR ← DR+1

D_6T_6: M[AR] ← DR, if (DR = 0) then (PC ← PC+1), SC ← 0

Control Flowchart

A flowchart showing all microoperations for the execution of the seven memory-reference instructions is shown in Figure 11.

Note that we need only seven timing signals to execute the longest instruction (ISZ). The computer can be designed with a 3-bit sequence counter. The reason for using a 4-bit counter for SC is to provide additional timing signals for other instructions that are presented in the problems section.

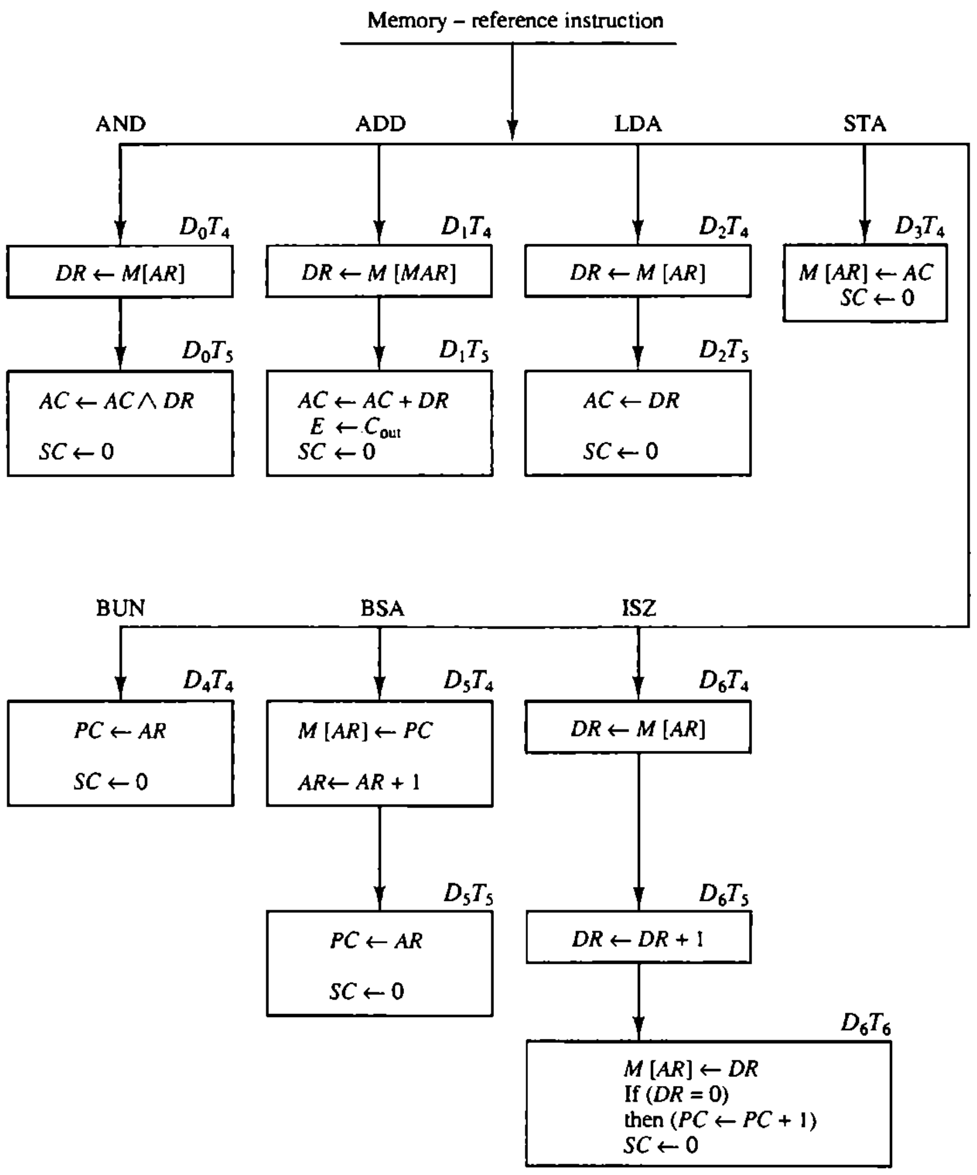


Figure 25 Flowchart for memory-reference instructions.

Input—Output and Interrupt

A computer can serve no useful purpose unless it communicates with the external environment. Instructions and data stored in memory must come from some input device. Computational results must be transmitted to the user through some output device. Commercial computers include many types of input and output devices. To demonstrate the most basic requirements for input and output communication, it will use as an illustration a terminal unit with a keyboard and printer.

Input-Output Configuration

The terminal sends and receives serial information. Each quantity of information has eight bits of an alphanumeric code. The serial information from the keyboard is shifted into the input register INPR. The serial information for the printer is stored in the output register OUTR. These two registers communicate with a communication interface serially and with the AC in parallel. The input-output configuration is shown in Figure 12. The transmitter interface receives serial information from the keyboard and transmits it to INPR. The receiver interface receives information from OUTR and sends it to the printer serially.

The input register INPR consists of eight bits and holds an alphanumeric input information. The 1-bit input flag FGI is a control flip-flop. The flag bit is set to 1 when new information is available in the input device and is cleared to 0 when the information is accepted by the computer. The flag is needed to synchronize the timing rate difference between the input device and the computer. The process of information transfer is as follows. Initially, the input flag FGI is cleared to 0. When a key is struck in the keyboard, an 8-bit alphanumeric code is shifted into INPR and the input flag FGI is set to 1. As long as the flag is set, the information in INPR cannot be changed by striking another key. The computer checks the flag bit; if it is 1, the information from INPR is transferred in parallel into AC and FGI is cleared to 0. Once the flag is cleared, new information can be shifted into INPR by striking another key.

The output register OUTR works similarly but the direction of information flow is reversed. Initially, the output flag FGO is set to 1. The computer checks the flag bit; if it is 1, the information from AC is transferred in parallel to OUTR and FGO is cleared to 0. The output device accepts the coded information, prints the corresponding character, and when the operation is completed, it sets FGO to 1. The computer does not load a new character into OUTR when FGO is 0 because this condition indicates that the output device is in the process of printing the character.

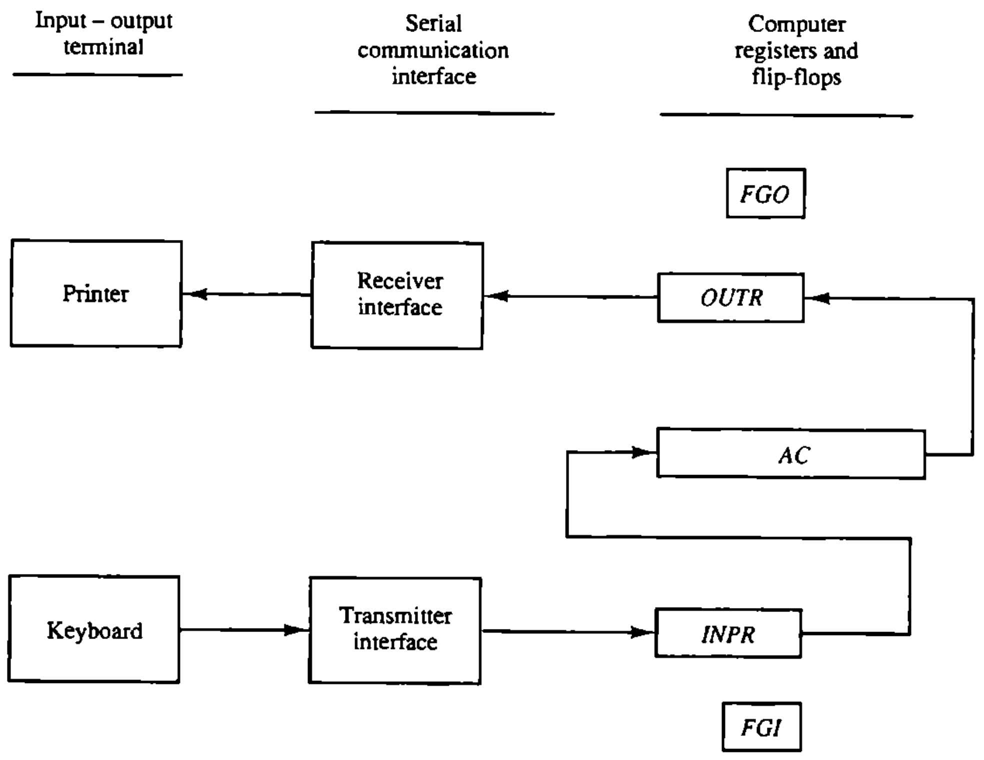


Figure 26 Input-output configuration.

Input-Output Instructions

Input and output instructions are needed for transferring information to and from AC register, for checking the flag bits, and for controlling the interrupt facility. Input-output instructions have an operation code 1111 and are recognized by the control when $D_7 = 1$ and $I = 1$. The remaining bits of the instruction specify the particular operation. The control functions and microoperations for the input-output instructions are listed in Table 5. These instructions are executed with the clock transition associated with timing signal T_3. Each control function needs a Boolean relation D_7IT_3, which we designate for convenience by the symbol p. The control function is distinguished by one of the bits in IR(6-11). By assigning the symbol B_i to bit i of IR, all control functions can be denoted by pB_i for i = 6 through 11. The sequence counter SC is cleared to 0 when $p = D_7IT_3 = 1$.

Table 9 Input-Output Instructions

$D_7IT_3 = p$ (common to all input–output instructions)
$IR(i) = B_i$ [bit in $IR(6–11)$ that specifies the instruction]

	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0–7) \leftarrow INPR, \quad FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC(0–7), \quad FGO \leftarrow 0$	Output character
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$	Skip on input flag
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$	Skip on output flag
ION	pB_7:	$IEN \leftarrow 1$	Interrupt enable on
IOF	pB_6:	$IEN \leftarrow 0$	Interrupt enable off

Program Interrupt

The process of communication just described is referred to as programmed control transfer. The computer keeps checking the flag bit, and when it finds it set, it initiates an information transfer. The difference of information flow rate between the computer and that of the input-output device makes this type of transfer inefficient. To see why this is inefficient, consider a computer that can go through an instruction cycle in 1 (is. Assume that the input-output device can transfer information at a maximum rate of 10 characters per second. This is equivalent to one character every 100,000 (is. Two instructions are executed when the computer checks the flag bit and decides not to transfer the information. This means that at the maximum rate, the computer will check the flag 50,000 times between each transfer. The computer is wasting time while checking the flag instead of doing some other useful processing task.

An alternative to the programmed controlled procedure is to let the external device inform the computer when it is ready for the transfer. In the meantime the computer can be busy with other tasks. This type of transfer uses the interrupt facility. While the computer is running a program, it does not check the flags. However, when a flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that a flag has been set. The computer deviates momentarily from what it is doing to take care of the input or output transfer. It then returns to the current program to continue what it was doing before the interrupt.

The interrupt enable flip-flop IEN can be set and cleared with two instructions. When IEN is cleared to 0 (with the IOF instruction), the flags cannot interrupt the computer. When IEN is set to 1 (with the ION instruction), the computer can be interrupted. These two instructions provide the programmer with the capability of making a decision as to whether or not to use the interrupt facility.

The way that the interrupt is handled by the computer can be explained by means of the flowchart of Figure 13. An interrupt flip-flop R is included in the computer.

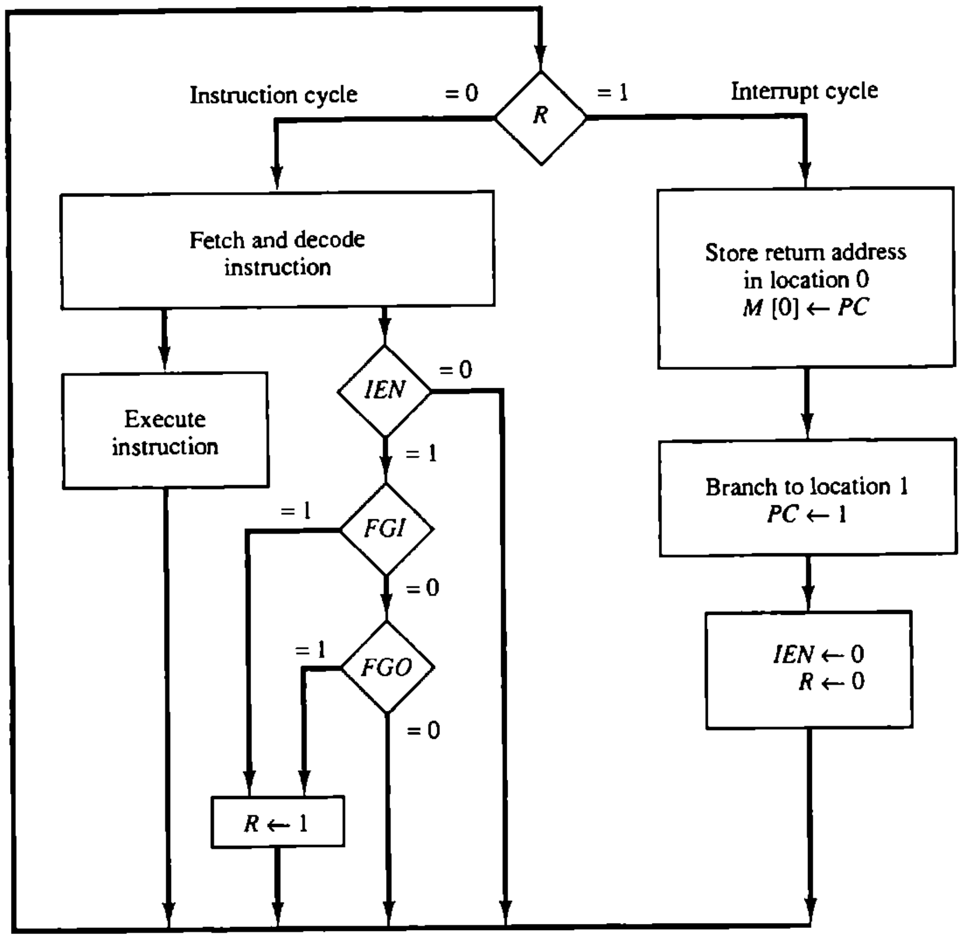


Figure 27 Flowchart for interrupt cycle.

The interrupt cycle is a hardware implementation of a branch and save return address operation. The return address available in PC is stored in a specific location where it can be found later when the program returns to the instruction at which it was interrupted. This location may be a processor register, a memory stack, or a specific memory location. Here we choose the memory location at address 0 as the place for storing the return address. Control then inserts address 1 into PC and clears IEN and R so that no more interruptions can occur until the interrupt request from the flag has been serviced.

An example that shows what happens during the interrupt cycle is shown in Figure 14. Suppose that an interrupt occurs and R is set to 1 while the control is executing the instruction at address 255. At this time, the return address 256 is in PC. The programmer has previously placed an input-output service program in memory starting from address 1120 and a BUN 1120 instruction at address 1. This is shown in Figure 14(a).

When control reaches timing signal T_0 and finds that R = 1, it proceeds with the interrupt cycle. The content of PC (256) is stored in memory location 0, PC is set to 1, and R is cleared to 0. At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1 since this is the content of PC. The branch instruction at address 1 causes the program to transfer to the input-output service program at address 1120. This program checks the flags, determines which flag is set, and then transfers the required input or output information. Once this is done, the instruction ION is executed to set IEN to 1 (to enable further interrupts), and the program returns to the location where it was interrupted. This is shown in Figure 14(b).

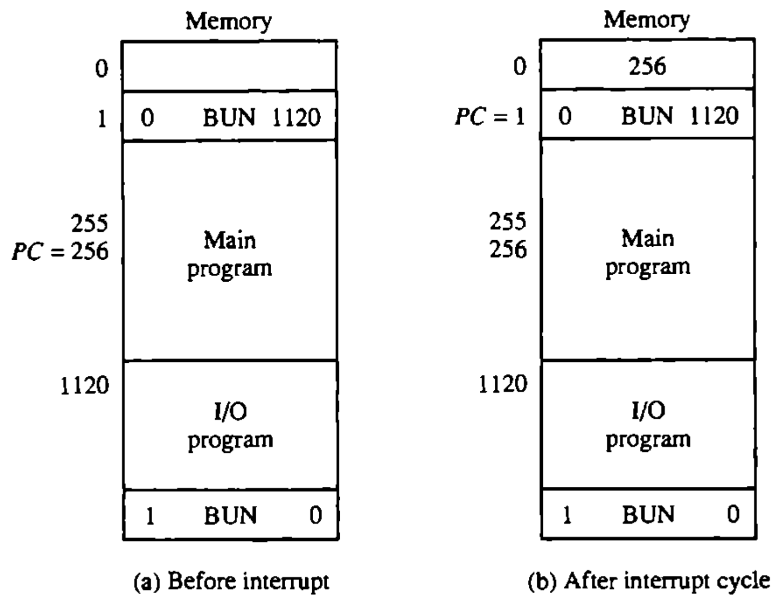


Figure 28 Demonstration of the interrupt cycle.

The instruction that returns the computer to the original place in the main program is a branch indirect instruction with an address part of 0. This instruction is placed at the end of the I/O service program. After this instruction is read from memory during the fetch phase, control goes to the indirect phase (because I = 1) to read the effective address. The effective address is in location 0 and is the return address that was stored there during the previous interrupt cycle. The execution of the indirect BUN instruction results in placing into PC the return address from location 0.

Interrupt Cycle

The interrupt cycle is initiated after the last execute phase if the interrupt flip-flop R is equal to 1. This flip-flop is set to 1 if IEN = 1 and either FGI or FGO are equal to 1. This can happen with any clock transition except when timing signals T_0, T_1, or T_2 are active. The condition for setting flip-flop R to 1 can be expressed with the following register transfer statement:

$T_0'T_1'T_2'(IEN)(FGI + FGO)$: $R \leftarrow 1$

The symbol + between FGI and FGO in the control function designates a logic OR operation. This is ANDed with IEN and $T_0'T_1'T_2'$.

The fetch and decode phases of the instruction cycle is now modified. Instead of using only timing signals T_0, T_1, and T_2 (as shown in Figure 9) we will AND the three timing signals with R' so that the fetch and decode phases will be recognized from the three control functions $R'T_0$, $R'T_1$, and $R'T_2$. The reason for this is that after the instruction is executed and SC is cleared to 0, the control will go through a fetch phase only if R = 0. Otherwise, if R = 1, the control will go through an interrupt cycle. The interrupt cycle stores the return address (available in PC) into memory location 0, branches to memory location 1, and clears IEN, R, and SC to 0. This can be done with the following sequence of microoperations:

RT_0: $AR \leftarrow 0$, $TR \leftarrow PC$
RT_1: $M[AR] \leftarrow TR$, $PC \leftarrow 0$
RT_2: $PC \leftarrow PC + 1$, $IEN \leftarrow 0$, $R \leftarrow 0$, $SC \leftarrow 0$

Complete Computer Description

The final flowchart of the instruction cycle, including the interrupt cycle for the basic computer, is shown in Figure 15.

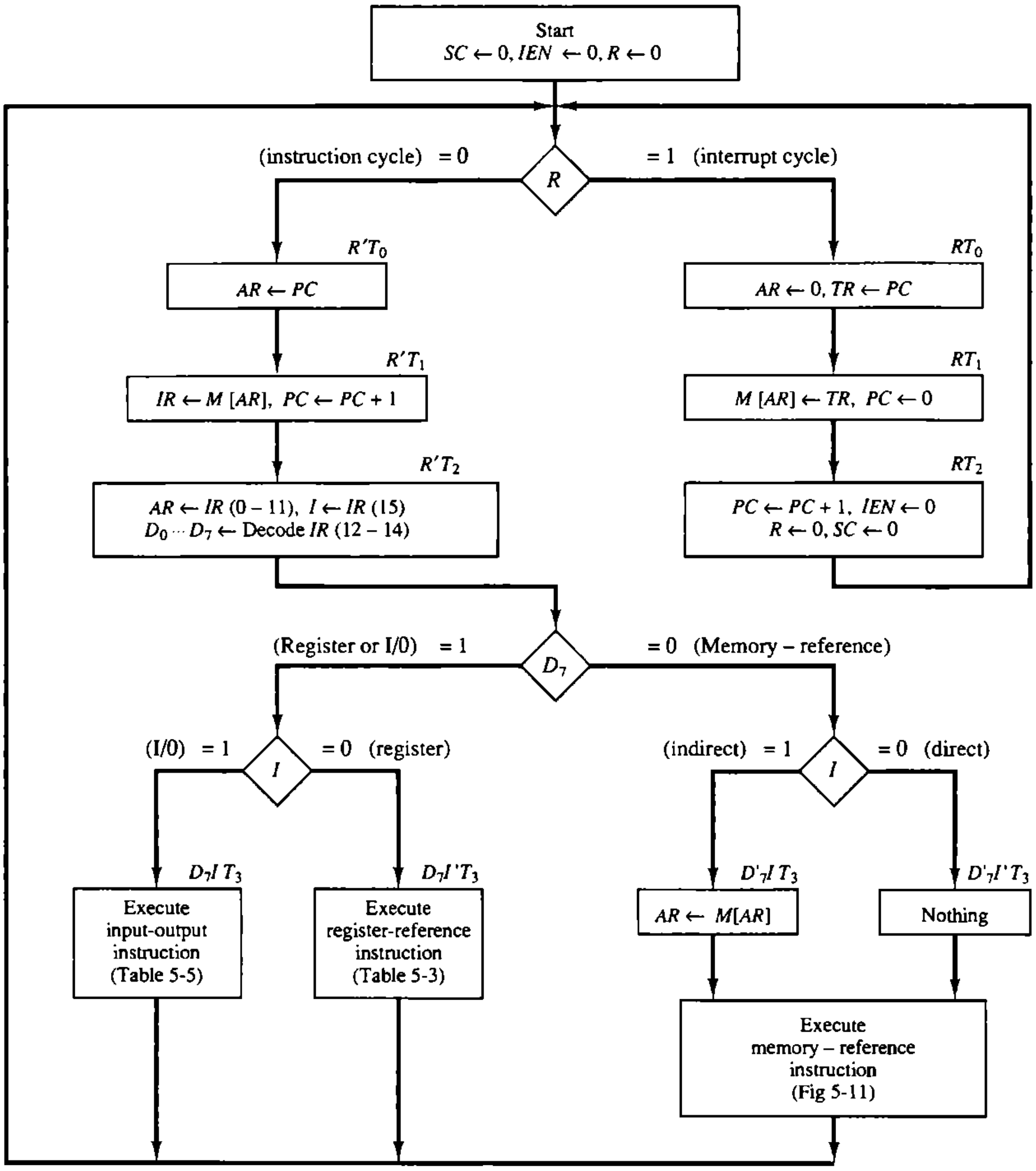

Start
$SC \leftarrow 0, IEN \leftarrow 0, R \leftarrow 0$
(instruction cycle) = 0
R
= 1 (interrupt cycle)
$R'T_0$
$AR \leftarrow PC$
$R'T_1$
$IR \leftarrow M[AR], PC \leftarrow PC + 1$
$R'T_2$
$AR \leftarrow IR(0-11), I \leftarrow IR(15)$
$D_0 \cdots D_7 \leftarrow$ Decode $IR(12-14)$
RT_0
$AR \leftarrow 0, TR \leftarrow PC$
RT_1
$M[AR] \leftarrow TR, PC \leftarrow 0$
RT_2
$PC \leftarrow PC + 1, IEN \leftarrow 0$
$R \leftarrow 0, SC \leftarrow 0$
(Register or I/0) = 1
D_7
= 0 (Memory – reference)
(I/0) = 1
I
= 0 (register)
(indirect) = 1
I
= 0 (direct)
D_7IT_3
$D_7I'T_3$
D'_7IT_3
$D'_7I'T_3$
Execute input-output instruction (Table 5-5)
Execute register-reference instruction (Table 5-3)
$AR \leftarrow M[AR]$
Nothing
Execute memory – reference instruction (Fig 5-11)

Figure 29 Flowchart for computer operation.

The control functions and microoperations for the entire computer are summarized in Table 6. The register transfer statements in this table describe in a concise form the internal organization of the basic computer. They also give all the information necessary for the design of the logic circuits of the computer. The control functions and conditional control statements listed in the table formulate the Boolean functions for the gates in the control unit. The list of microoperations specifies the type of control inputs needed for the registers and memory. A register transfer language is useful not only for describing the internal organization of a digital system but also for specifying the logic circuits needed for its design.

Table 10 Control Functions and Microoperations for the Basic Computer

Fetch	$R'T_0$:	$AR \leftarrow PC$
	$R'T_1$:	$IR \leftarrow M[AR]$, $PC \leftarrow PC + 1$
Decode	$R'T_2$:	$D_0, \ldots, D_7 \leftarrow$ Decode $IR(12\text{–}14)$,
		$AR \leftarrow IR(0\text{–}11)$, $I \leftarrow IR(15)$
Indirect	$D_7'IT_3$:	$AR \leftarrow M[AR]$
Interrupt:		
$T_0'T_1'T_2'(IEN)(FGI + FGO)$:		$R \leftarrow 1$
	RT_0:	$AR \leftarrow 0$, $TR \leftarrow PC$
	RT_1:	$M[AR] \leftarrow TR$, $PC \leftarrow 0$
	RT_2:	$PC \leftarrow PC + 1$, $IEN \leftarrow 0$, $R \leftarrow 0$, $SC \leftarrow 0$
Memory-reference:		
AND	D_0T_4:	$DR \leftarrow M[AR]$
	D_0T_5:	$AC \leftarrow AC \wedge DR$, $SC \leftarrow 0$
ADD	D_1T_4:	$DR \leftarrow M[AR]$
	D_1T_5:	$AC \leftarrow AC + DR$, $E \leftarrow C_{out}$, $SC \leftarrow 0$
LDA	D_2T_4:	$DR \leftarrow M[AR]$
	D_2T_5:	$AC \leftarrow DR$, $SC \leftarrow 0$
STA	D_3T_4:	$M[AR] \leftarrow AC$, $SC \leftarrow 0$
BUN	D_4T_4:	$PC \leftarrow AR$, $SC \leftarrow 0$
BSA	D_5T_4:	$M[AR] \leftarrow PC$, $AR \leftarrow AR + 1$
	D_5T_5:	$PC \leftarrow AR$, $SC \leftarrow 0$
ISZ	D_6T_4:	$DR \leftarrow M[AR]$
	D_6T_5:	$DR \leftarrow DR + 1$
	D_6T_6:	$M[AR] \leftarrow DR$, if $(DR = 0)$ then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$
Register-reference:		
		$D_7I'T_3 = r$ (common to all register-reference instructions)
		$IR(i) = B_i$ $(i = 0, 1, 2, \ldots, 11)$
	r:	$SC \leftarrow 0$
CLA	rB_{11}:	$AC \leftarrow 0$
CLE	rB_{10}:	$E \leftarrow 0$
CMA	rB_9:	$AC \leftarrow \overline{AC}$
CME	rB_8:	$E \leftarrow \overline{E}$
CIR	rB_7:	$AC \leftarrow \text{shr } AC$, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$
CIL	rB_6:	$AC \leftarrow \text{shl } AC$, $AC(0) \leftarrow E$, $E \leftarrow AC(15)$
INC	rB_5:	$AC \leftarrow AC + 1$
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1)$
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$
HLT	rB_0:	$S \leftarrow 0$
Input-output:		
		$D_7IT_3 = p$ (common to all input–output instructions)
		$IR(i) = B_i$ $(i = 6, 7, 8, 9, 10, 11)$
	p:	$SC \leftarrow 0$
INP	pB_{11}:	$AC(0\text{–}7) \leftarrow INPR$, $FGI \leftarrow 0$
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{–}7)$, $FGO \leftarrow 0$
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$
ION	pB_7:	$IEN \leftarrow 1$
IOF	pB_6:	$IEN \leftarrow 0$

Design of Basic Computer

The basic computer consists of the following hardware components:

1. A memory unit with 4096 words of 16 bits each
2. Nine registers: AR, PC, DR, AC, IR, TR, OUTR, INPR, and SC
3. Seven flip-flops: I, S, E, R, IEN, FGI, and FGO
4. Two decoders: a 3 x 8 operation decoder and a 4 x 16 timing decoder
5. A 16-bit common bus
6. Control logic gates
7. Adder and logic circuit connected to the input of AC

Control Logic Gates

The block diagram of the control logic gates is shown in Figure 6. The inputs to this circuit come from the two decoders, the I flip-flop, and bits 0 through 11 of IR. The other inputs to the control logic are: AC bits 0 through 15 to check if AC = 0 and to detect the sign bit in AC(15); DR bits 0 through 15 to check if DR = 0; and the values of the seven flip-flops.

The outputs of the control logic circuit are:

1. Signals to control the inputs of the nine registers
2. Signals to control the read and write inputs of memory
3. Signals to set, clear, or complement the flip-flops
4. Signals for S2, Si, and S0 to select a register for the bus
5. Signals to control the AC adder and logic circuit

The specifications for the various control signals can be obtained directly from the list of register transfer statements in Table 6.

Control of Registers and Memory

The registers of the computer connected to a common bus system are shown in Figure 4. The control inputs of the registers are LD (load), INR (increment), and CLR (clear). Suppose that we want to derive the gate structure associated with the control inputs of AR. We scan Table 6 to find all the statements that change the content of AR:

$R'T_0$:	$AR \leftarrow PC$
$R'T_2$:	$AR \leftarrow IR(0\text{-}11)$
$D_7'IT_3$:	$AR \leftarrow M[AR]$
RT_0:	$AR \leftarrow 0$
D_5T_4:	$AR \leftarrow AR+1$

The first three statements specify transfer of information from a register or memory to AR. The content of the source register or memory is placed on the bus and the content of the bus is transferred into AR by enabling its LD control input. The fourth statement clears AR to 0. The last statement increments AR by 1. The control functions can be combined into three Boolean expressions as follows:

$LD(AR) = R'T_0 + R'T_2 + D_7'IT_3$

$CLR(AR) = RT_0$

$INR(AR) = D_5T_4$

where LD(AR) is the load input of AR, CLR(AR) is the clear input of AR, and INR(AR) is the increment input of AR. The control gate logic associated with AR is shown in Figure 16.

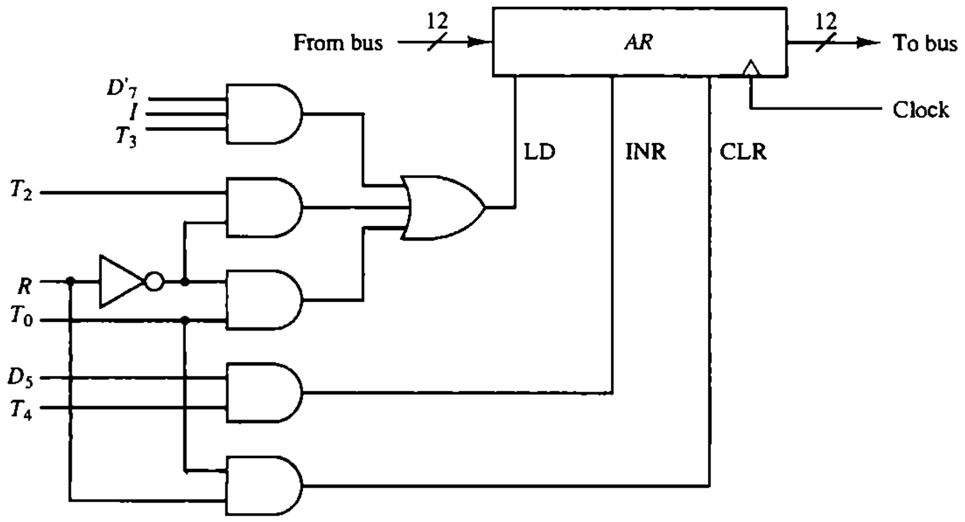


Figure 30 Control gates associated with AR.

In a similar fashion we can derive the control gates for the other registers as well as the logic needed to control the read and write inputs of memory. The logic gates associated with the read input of memory is derived by scanning Table 6 to find the statements that specify a read operation. The read operation is recognized from the symbol $\leftarrow$ M[AR].

Read = $R'T_1 + D_7'IT_3 + (D_0 + D_1 + D_2 + D_6)T_4$

The output of the logic gates that implement the Boolean expression above must be connected to the read input of memory.

H.W. Derive the control gates of write, LD(PC), INR(PC) and CLR(PC)

Control of Single Flip-flops

The control gates for the seven flip-flops can be determined in a similar manner. For example, Table 6 shows that IEN may change as a result of the two instructions ION and IOF.

pB_7: IEN $\leftarrow$ 1

pB_6: IEN $\leftarrow$ 0

where p = D_7IT_3 and B_7 and B_6 are bits 7 and 6 of IR, respectively. Moreover, at the end of the interrupt cycle IEN is cleared to 0.

RT_2: IEN $\leftarrow$ 0

If we use a JK flip-flip for IEN, the control gate logic will be as shown in Figure 17.

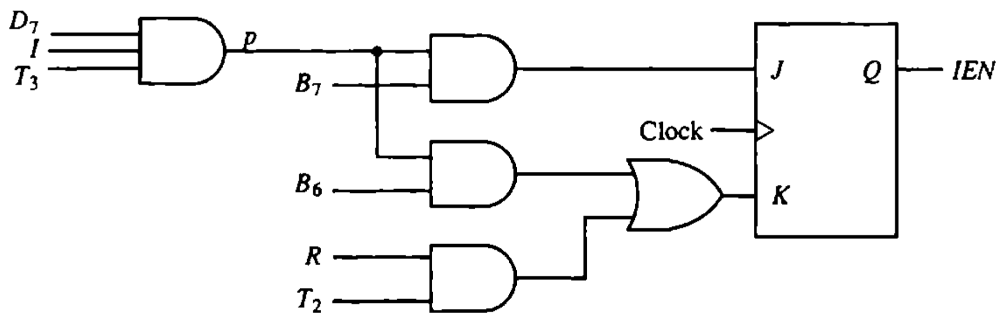


Figure 31 Control inputs for IEN.

Control of Common Bus

The 16-bit common bus shown in Fig. 5-4 is controlled by the selection inputs S_2, S_1, and S_0. The decimal number shown with each bus input specifies the equivalent binary number that must be applied to the selection inputs in order to select the corresponding register. Table 5-7 specifies the binary numbers for $S_2S_1S_0$ that select each register. Each binary number is associated with a Boolean variable x_1 through x_7, corresponding to the gate structure that must be active in order to select the register or memory for the bus. For example, when $x_1 = 1$, the value of $S_2S_1S_0$ must be 001 and the output of AR will be selected for the bus. Table 7 is recognized as the truth table of a binary encoder. The placement of the encoder at the inputs of the bus selection logic is shown in Figure 18. The Boolean functions for the encoder are

$S_0 = x_1 + x_3 + x_5 + x_7$

$S_1 = x_2 + x_3 + x_6 + x_7$

$S_2 = x_1 + x_5 + x_6 + x_7$

Table 11 Encoder for Bus Selection Circuit

Inputs							Outputs			Register selected for bus
x_1	x_2	x_3	x_4	x_5	x_6	x_7	S_2	S_1	S_0	
0	0	0	0	0	0	0	0	0	0	None
1	0	0	0	0	0	0	0	0	1	*AR*
0	1	0	0	0	0	0	0	1	0	*PC*
0	0	1	0	0	0	0	0	1	1	*DR*
0	0	0	1	0	0	0	1	0	0	*AC*
0	0	0	0	1	0	0	1	0	1	*IR*
0	0	0	0	0	1	0	1	1	0	*TR*
0	0	0	0	0	0	1	1	1	1	Memory

To determine the logic for each encoder input, it is necessary to find the control functions that place the corresponding register onto the bus. For example, to find the logic that makes $x_1 = 1$, we scan all register transfer statements in Table 6 and extract those statements that have AR as a source.

D_4T_4: PC $\leftarrow$ AR

D_5T_5: PC $\leftarrow$ AR

Therefore, the Boolean function for x_1 is

$x_1 = D_4T_4 + D_5T_5$

The data output from memory are selected for the bus when $x_7 = 1$ and $S_2S_1S_0 = 111$. The gate logic that generates x7 must also be applied to the read input of memory. Therefore, the Boolean function for x_7 is the same as the one derived previously for the read operation.

$x_7 = R'T_1 + D_7IT_3 + (D_0 + D_1 + D_2 + D_6)T_4$

In a similar manner we can determine the gate logic for the other registers.

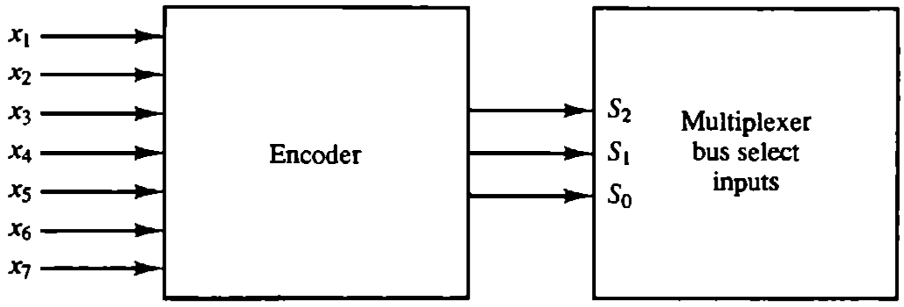


Figure 32 Encoder for bus selection inputs.

Design of Accumulator Logic

The circuits associated with the AC register are shown in Figure 19. The adder and logic circuit has three sets of inputs. One set of 16 inputs comes from the outputs of AC. Another set of 16 inputs comes from the data register DR. A third set of eight inputs comes from the input register INPR. The outputs of the adder and logic circuit provide the data inputs for the register. In addition, it is necessary to include logic gates for controlling the LD, INR, and CLR in the register and for controlling the operation of the adder and logic circuit.

In order to design the logic associated with AC, it is necessary to go over the register transfer statements in Table 6 and extract all the statements that change the content of AC.

D_0T_5:	$AC \leftarrow AC \wedge DR$	AND with DR
D_1T_5:	$AC \leftarrow AC + DR$	Add with DR
D_2T_5:	$AC \leftarrow DR$	Transfer from DR
pB_{11}:	$AC(0\text{–}7) \leftarrow INPR$	Transfer from $INPR$
rB_9:	$AC \leftarrow \overline{AC}$	Complement
rB_7:	$AC \leftarrow \text{shr } AC, \quad AC(15) \leftarrow E$	Shift right
rB_6:	$AC \leftarrow \text{shl } AC, \quad AC(0) \leftarrow E$	Shift left
rB_{11}:	$AC \leftarrow 0$	Clear
rB_5:	$AC \leftarrow AC + 1$	Increment

From this list we can derive the control logic gates and the adder and logic circuit.

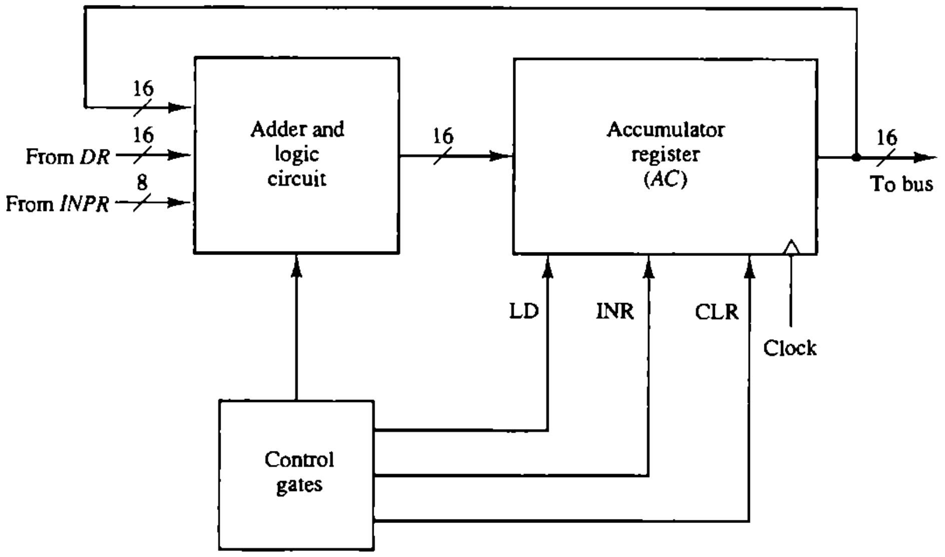


Figure 33 Circuits associated with AC.

Control of AC Register

The gate structure that controls the LD, INR, and CLR inputs of AC is shown in Figure 20. The gate configuration is derived from the control functions in the list above. The control function for the dear microoperation is rB_{11}, where r = D_7IT_3 and B_{11} = IR(11). The output of the AND gate that generates this control function is connected to the CLR input of the register. Similarly, the output of the gate that implements the increment microoperation is connected to the INR input of the register. The other seven microoperations are generated in the adder and logic circuit and are loaded into AC at the proper time. The outputs of the gates for each control function is marked with a symbolic name. These outputs are used in the design of the adder and logic circuit.

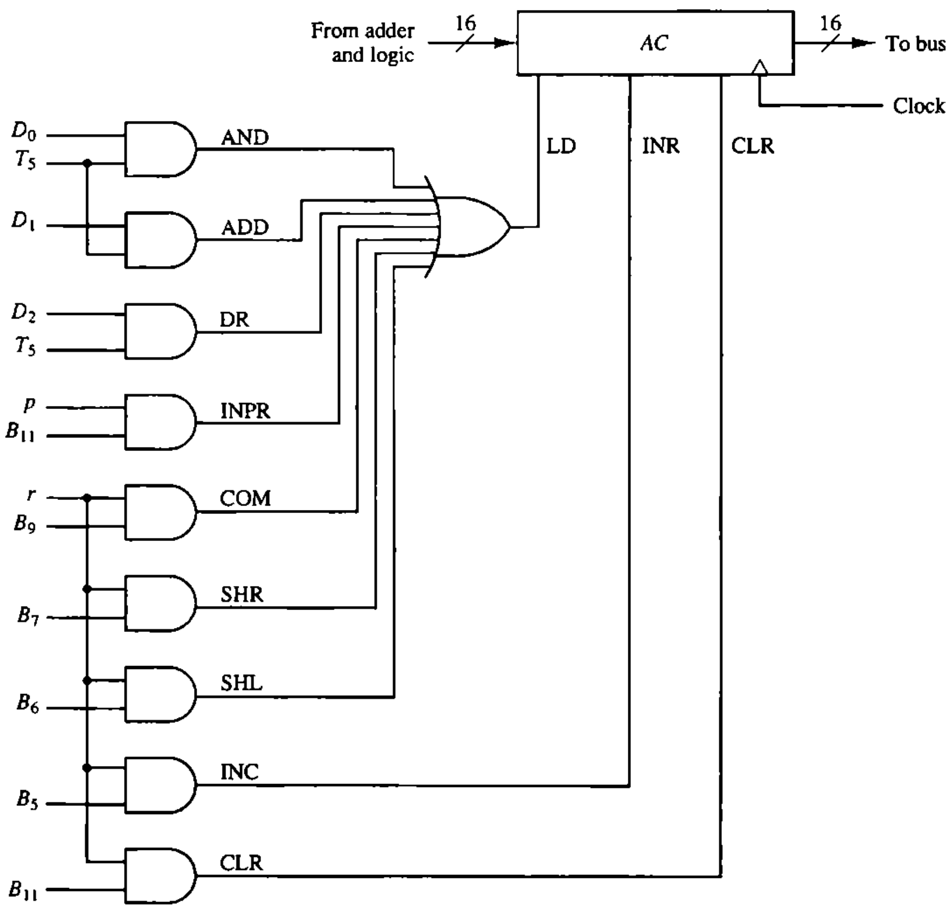


Figure 34 Gate structure for controlling the LD, INR, and CLR of AC.

Adder and Logic Circuit

The adder and logic circuit can be subdivided into 16 stages, with each stage corresponding to one bit of AC.

Figure 21 shows one such AC register stage (with the OR gates removed). The input is labeled I, and the output AC(i). When the LD input is enabled, the 16 inputs I_i for i = 0, 1, 2,..., 15 are transferred to AC(0-15).

One stage of the adder and logic circuit consists of seven AND gates, one OR gate and a full-adder (FA), as shown in Figure 21. The inputs of the gates with symbolic names come from the outputs of gates marked with the same symbolic name in Figure 20. For example, the input marked ADD in Figure 21 is connected to the output marked ADD in Figure 20.

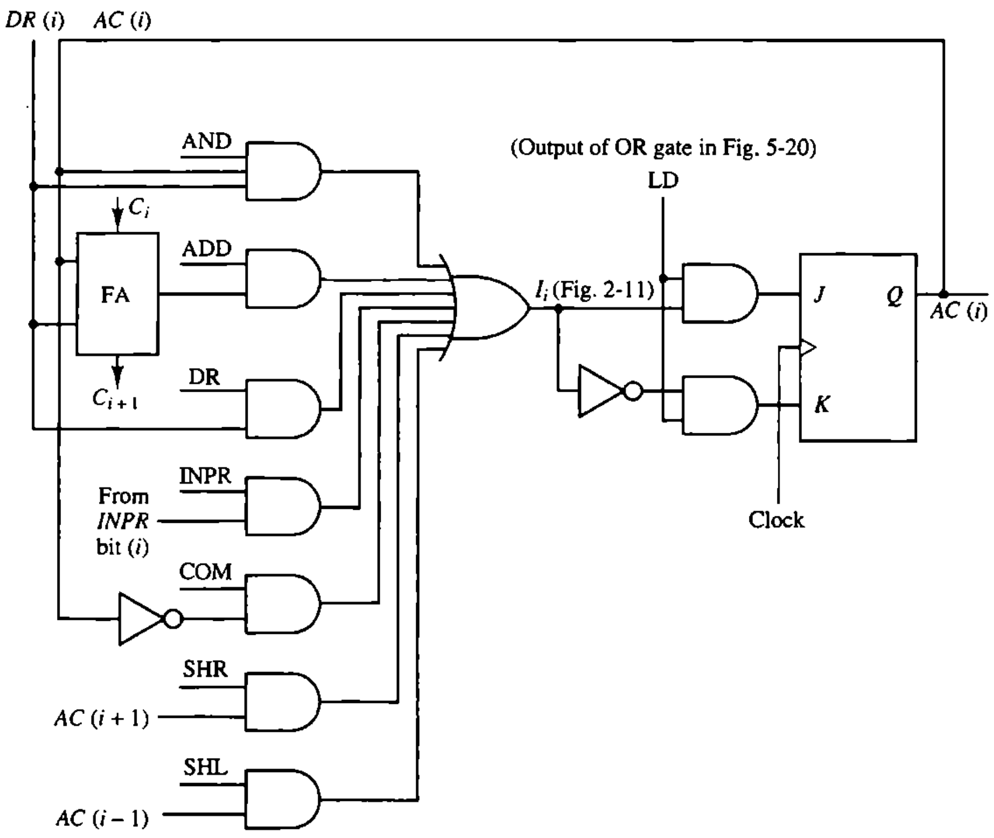


Figure 35 One stage of adder and logic circuit.

The complete adder and logic circuit consists of 16 stages connected together.

The Assembler

- An assembler is a program that accepts a symbolic language program and produces its binary machine language equivalent.
- The input symbolic program is called the source program and the resulting binary program is called the object program.
- The assembler is a program that operates on character strings and produces an equivalent binary interpretation.

Representation of Symbolic Program in Memory

- Prior to starting the assembly process, the symbolic program must be stored in memory.
- A loader program is used to input the characters of the symbolic program into memory.
- Since the program consists of symbols, its representation in memory must use an alphanumeric character code.
- In the basic computer, each character is represented by an 8-bit code. The high-order bit is always 0 and the other seven bits are as specified by ASCII.
- The hexadecimal equivalent of the character set is listed in Table 1.
- The last entry in the table does not print a character but is associated with the physical movement of the cursor in the terminal.
- The code for CR is produced when the return key is depressed. This causes the "carriage" to return to its initial position to start typing a new line. The assembler recognizes a CR code as the end of a line of code.

Table 12 Hexadecimal Character Code

Character	Code	Character	Code	Character	Code	
A	41	Q	51	6	36	
B	42	R	52	7	37	
C	43	S	53	8	38	
D	44	T	54	9	39	
E	45	U	55	space	20	
F	46	V	56	(	28	
G	47	W	57	)	29	
H	48	X	58	*	2A	
I	49	Y	59	+	2B	
J	4A	Z	5A	,	2C	
K	4B	0	30	−	2D	
L	4C	1	31	.	2E	
M	4D	2	32	/	2F	
N	4E	3	33	=	3D	
O	4F	4	34	CR	0D	(carriage
P	50	5	35			return)

A line of code is stored in consecutive memory locations with two characters in each location. Two characters can be stored in each word since a memory word has a capacity of 16 bits.

A label symbol is terminated with a comma. Operation and address symbols are terminated with a space and the end of the line is recognized by the CR code. For example, the following line of code:

PL3, LDA SUB I

is stored in seven consecutive memory locations, as shown in Table 2. The label PL3 occupies two words and is terminated by the code for comma (2C). The instruction field in the line of code may have one or more symbols. Each symbol is terminated by the code for space (20) except for the last symbol, which is terminated by the code of carriage return (0D). If the line of code has a comment, the assembler recognizes it by the code for a slash (2F). The assembler neglects all characters in the comment field and keeps checking for a CR code. When this code is encountered, it replaces the space code after the last symbol in the line of code.

The input for the assembler program is the user's symbolic language program in ASCII. This input is scanned by the assembler twice to produce the equivalent binary program. The binary program constitutes the output generated by the assembler. The next section will describes briefly the major tasks that must be performed by the assembler during the translation process.

First Pass

- A two-pass assembler scans the entire symbolic program twice.
- During the first pass, it generates a table that correlates all user-defined address symbols with their binary equivalent value.
- The binary translation is done during the second pass.
- To keep track of the location of instructions, the assembler uses a memory word called a location counter (abbreviated LC).
- The content of LC stores the value of the memory location assigned to the instruction or operand presently being processed.
- The ORG pseudoinstruction initializes the location counter to the value of the first location.
- Since instructions are stored in sequential locations, the content of LC is incremented by 1 after processing each line of code.
- To avoid ambiguity in case ORG is missing, the assembler sets the location counter to 0 initially.

The tasks performed by the assembler during the first pass are described in the flowchart of Figure 1.

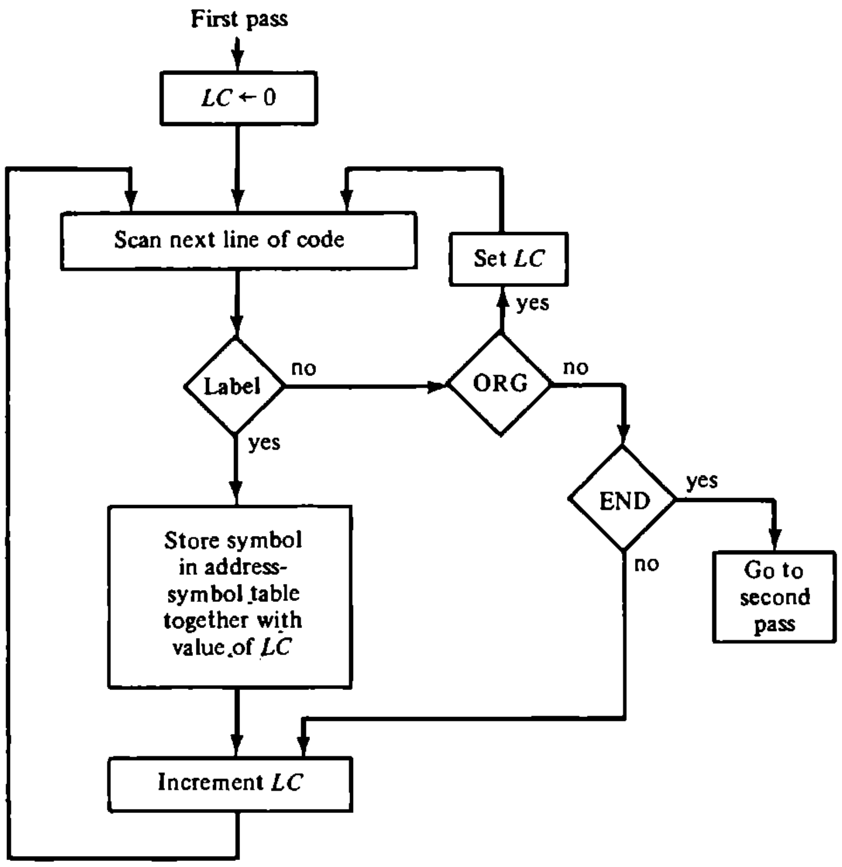


Figure 36 Flowchart for first pass of assembler.

- For the program of Table 2, the assembler generates the address symbol table listed in Table 3.
- Each label symbol is stored in two memory locations and is terminated by a comma.
- If the label contains less than three characters, the memory locations are filled with the code for space.
- The value found in LC while the line was processed is stored in the next sequential memory location.
- The program has three symbolic addresses: MIN, SUB, and DEF. These symbols represent 12-bit addresses equivalent to hexadecimal 106, 107, and 108, respectively.
- The address symbol table occupies three words for each label symbol encountered and constitutes the output data that the assembler generates during the first pass.

Table 13 Assembly Language Program to Subtract Two Numbers

```
        ORG 100      /Origin of program is location 100
        LDA SUB      /Load subtrahend to AC
        CMA          /Complement AC
        INC          /Increment AC
        ADD MIN      /Add minuend to AC
        STA DIF      /Store difference
        HLT          /Halt computer
MIN,    DEC 83       /Minuend
SUB,    DEC −23      /Subtrahend
DIF,    HEX 0        /Difference stored here
        END          /End of symbolic program
```

Table 14 Address Symbol Table for Program in Table 2

Memory word	Symbol or (LC)*	Hexadecimal code	Binary representation
1	M I	4D 49	0100 1101 0100 1001
2	N ,	4E 2C	0100 1110 0010 1100
3	(LC)	01 06	0000 0001 0000 0110
4	S U	53 55	0101 0011 0101 0101
5	B ,	42 2C	0100 0010 0010 1100
6	(LC)	01 07	0000 0001 0000 0111
7	D I	44 49	0100 0100 0100 1001
8	F ,	46 2C	0100 0110 0010 1100
9	(LC)	01 08	0000 0001 0000 1000

* (LC) designates content of location counter.

Second Pass

Machine instructions are translated during the second pass by means of table-lookup procedures.

A table-lookup procedure is a search of table entries to determine whether a specific item matches one of the items stored in the table.

The assembler uses four tables. Any symbol that is encountered in the program must be available as an entry in one of these tables; otherwise, the symbol cannot be interpreted. We assign the following names to the four tables:

1. Pseudoinstruction table.

2. MRI table.

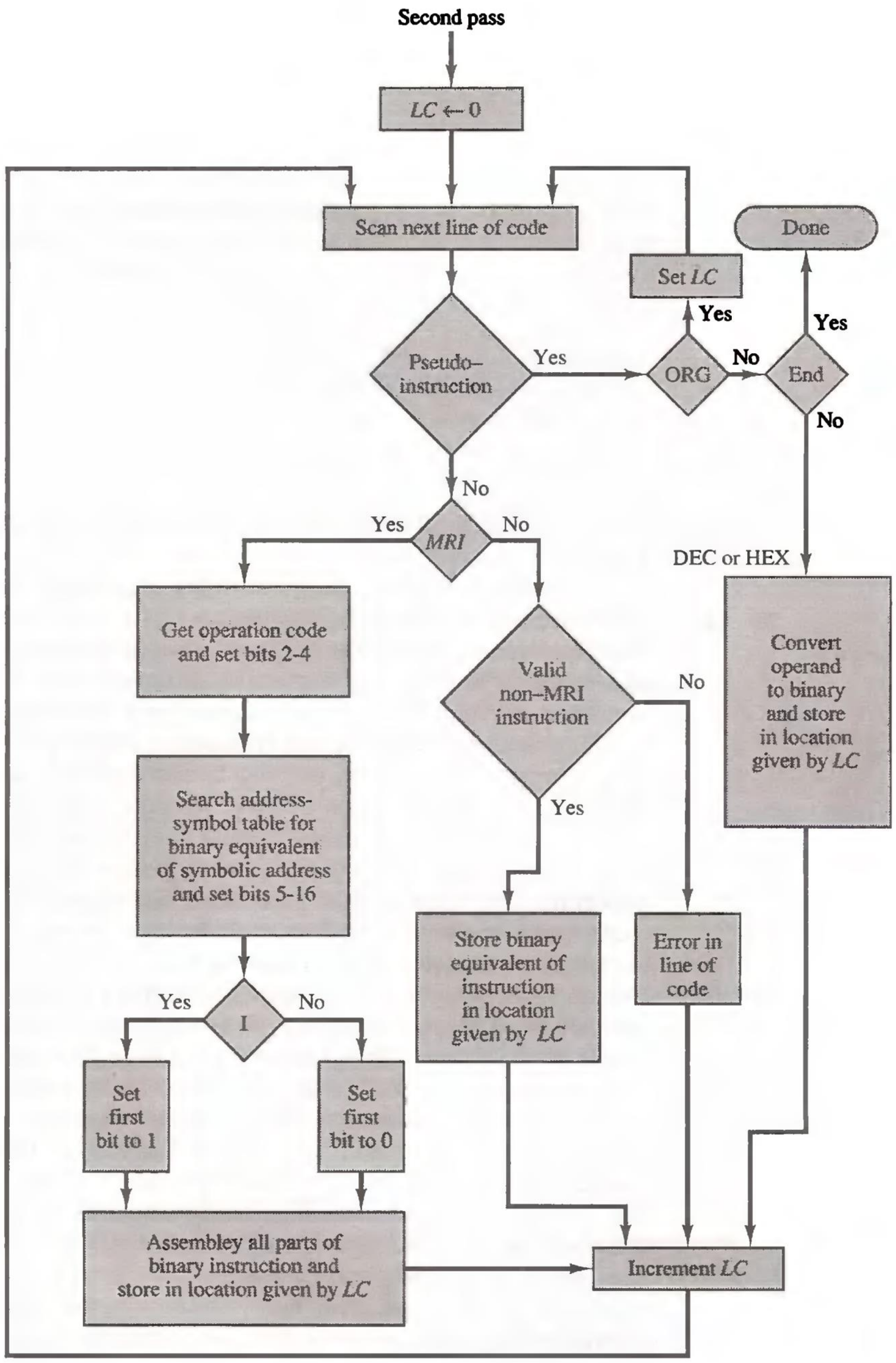
3. Non-MRI table.

4. Address symbol table.

- The entries of the pseudoinstruction table are the four symbols ORG, END, DEC, and HEX.

- Each entry refers the assembler to a subroutine that processes the pseudoinstruction when encountered in the program.
- The MRI table contains the seven symbols of the memory-reference instructions and their 3-bit operation code equivalent.
- The non-MRI table contains the symbols for the 18 register-reference and input-output instructions and their 16-bit binary code equivalent.
- The address symbol table is generated during the first pass of the assembly process.
- The assembler searches these tables to find the symbol that it is currently processing in order to determine its binary value.

The tasks performed by the assembler during the second pass are described in the flowchart of Figure 2.



Second pass
LC ← 0
Scan next line of code
Pseudo–instruction
Yes
ORG
No
End
Yes
Set LC
Yes
Done
No
No
DEC or HEX
MRI
Yes
No
Get operation code and set bits 2-4
Search address-symbol table for binary equivalent of symbolic address and set bits 5-16
I
Yes
No
Set first bit to 1
Set first bit to 0
Assembley all parts of binary instruction and store in location given by LC
Valid non–MRI instruction
No
Yes
Store binary equivalent of instruction in location given by LC
Error in line of code
Convert operand to binary and store in location given by LC
Increment LC

Figure 37 Flowchart for second pass of assembler.

One important task of an assembler is to check for possible errors in the symbolic program. This is called error diagnostics.

- One such error may be an invalid machine code symbol which is detected by its being absent in the MRI and non-MRI tables. The assembler cannot translate such a symbol because it does not know its binary equivalent value. In such a case, the assembler prints an error message to inform the programmer that his symbolic program has an error at a specific line of code.
- Another possible error may occur if the program has a symbolic address that did not appear also as a label. The assembler cannot translate the line of code properly because the binary equivalent of the symbol will not be found in the address symbol table generated during the first pass.
- Other errors may occur and a practical assembler should detect all such errors and print an error message for each.

It should be emphasized that a practical assembler is much more complicated than the one explained here. Most computers give the programmer more flexibility in writing assembly language programs. For example, the user may be allowed to use either a number or a symbol to specify an address. Many assemblers allow the user to specify an address by an arithmetic expression. Many more pseudoinstructions may be specified to facilitate the programming task. As the assembly language becomes more sophisticated, the assembler becomes more complicated.

Microprogrammed Control

Control Memory

The function of the control unit in a digital computer is to initiate sequences of microoperations. The number of different types of microoperations that are available in a given system is finite. The complexity of the digital system is derived from the number of sequences of microoperations that are performed.

- When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be hardwired.
- Microprogramming is a second alternative for designing the control unit of a digital computer. The principle of microprogramming is an elegant and systematic method for controlling the microoperation sequences in a digital computer.

The control unit initiates a series of sequential steps of microoperations. During any given time, certain microoperations are to be initiated, while others remain idle. The control variables at any given time can be represented by a string of l's and 0's called a control word.

- As such, control words can be programmed to perform various operations on the components of the system.
- A control unit whose binary control variables are stored in memory is called a microprogrammed control unit.
- Each word in control memory contains within it a microinstruction. See figure 1.
- The microinstruction specifies one or more microoperations for the system.
- A sequence of microinstructions constitutes a microprogram.
- Since alterations of the microprogram are not needed once the control unit is in operation, the control memory can be a read-only memory (ROM).
- The content of the words in ROM are fixed and cannot be altered by simple programming since no writing capability is available in the ROM.

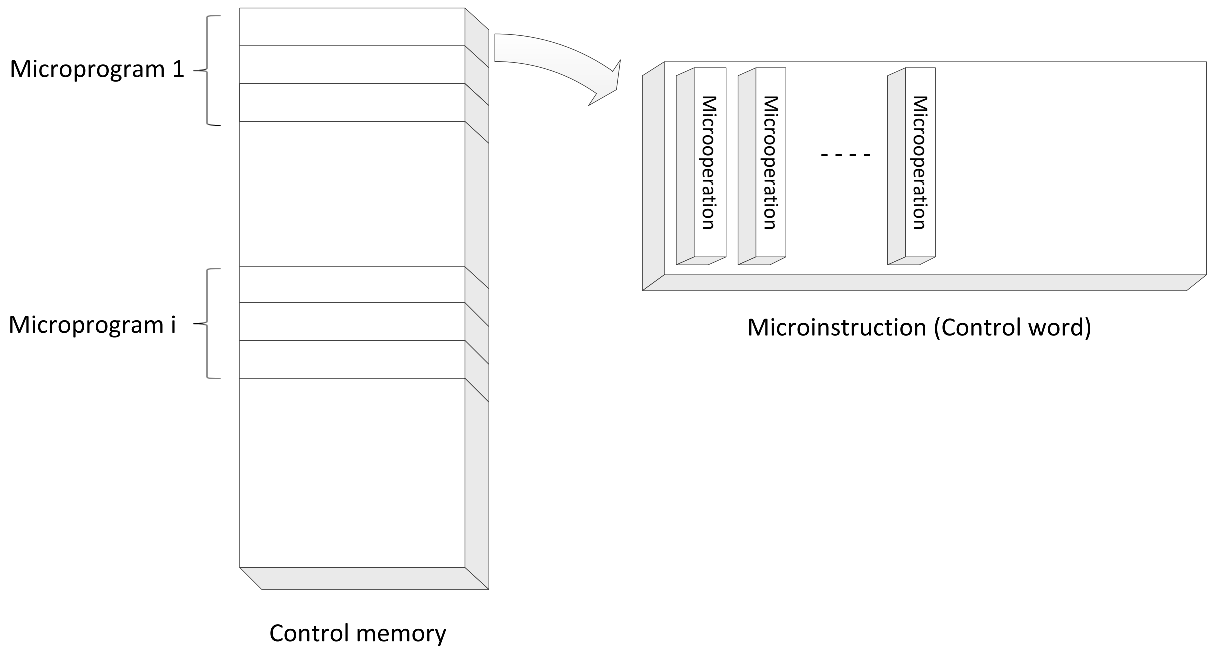


Figure 38 Control memory

ROM words are made permanent during the hardware production of the unit. The use of a microprogram involves placing all control variables in words of ROM for use by the control unit through successive read operations. The content of the word in ROM at a given address specifies a microinstruction.

A more advanced development known as dynamic microprogramming permits a microprogram to be loaded initially from an auxiliary memory such as a magnetic disk. Control units that use dynamic microprogramming employ a writable control memory. This type of memory can be used for writing (to change the microprogram) but is used mostly for reading. A memory that is part of a control unit is referred to as a control memory.

A computer that employs a microprogrammed control unit will have two separate memories: a main memory and a control memory.

- The main memory is available to the user for storing the programs. The contents of main memory may alter when the data are manipulated and every time that the program is changed. The user's program in main memory consists of machine instructions and data.
- In contrast, the control memory holds a fixed microprogram that cannot be altered by the occasional user. The microprogram consists of microinstructions that specify various internal control signals for execution of register microoperations.

Each machine instruction initiates a series of microinstructions in control memory. These microinstructions generate the microoperations to fetch the instruction from main memory; to evaluate the effective address, to execute the operation specified by the instruction, and to return control to the fetch phase in order to repeat the cycle for the next instruction.

The general configuration of a microprogrammed control unit is demonstrated in the block diagram of Figure 2. The control memory is assumed to be a ROM, within which all control information is permanently stored. The control memory address register specifies the address of the microinstruction, and the control data register holds the microinstruction read from memory.

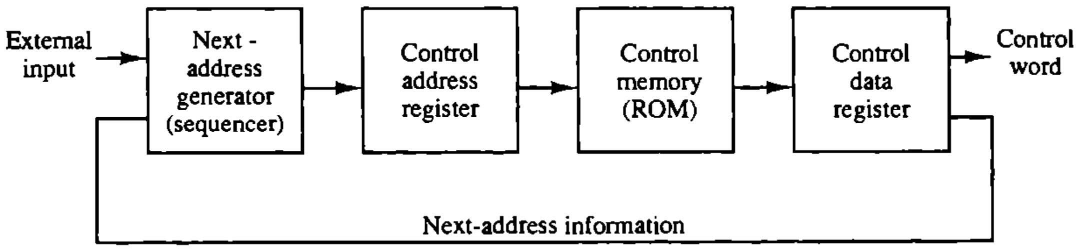


Figure 39 Microprogrammed control organization.

The microinstruction contains a control word that specifies one or more microoperations for the data processor. Once these operations are executed, the control must determine the next address.

The next address generator is sometimes called a microprogram sequencer, as it determines the address sequence that is read from control memory. The address of the next microinstruction can be specified in several ways, depending on the sequencer inputs. Typical functions of a microprogram sequencer are incrementing the control address register by one, loading into the control address register an address from control memory, transferring an external address, or loading an initial address to start the control operations.

The control data register holds the present microinstruction while the next address is computed and read from memory. The data register is sometimes called a pipeline register. It allows the execution of the microoperations specified by the control word simultaneously with the generation of the next microinstruction. This configuration requires a two-phase clock, with one clock applied to the address register and the other to the data register.

The system can operate without the control data register by applying a single-phase clock to the address register. The control word and next-address information are taken directly from the control memory. It must be realized that a ROM operates as a combinational circuit, with the address value as the input and the corresponding word as the output. The content of the specified word in ROM remains in the output wires as long as its address value remains in the address register. No read signal is needed as in a random-access memory. Each clock pulse will execute the microoperations specified by the control word and also transfer a new address to the control address register. In the example that follows we assume a single-phase clock and therefore we do not use a control data register. In this way the address register is the only component in the control system that receives clock pulses. The other two components: the sequencer and the control memory are combinational circuits and do not need a clock.

H.W. Problem 7-4 in M. Morris Mano.

The main advantage of the microprogrammed control is the fact that once the hardware configuration is established, there should be no need for further hardware or wiring changes. If we want to establish a different control sequence for the system, all we need to do is specify a different set of microinstructions for control memory. The hardware configuration should not be changed for different operations; the only thing that must be changed is the microprogram residing in control memory.

It should be mentioned that most computers based on the reduced instruction set computer (RISC) architecture concept use hardwired control rather than a control memory with a microprogram.

Address Sequencing

- Microinstructions are stored in control memory in groups, with each group specifying a routine.
- Each computer instruction has its own microprogram routine in control memory to generate the microoperations that execute the instruction.
- The hardware that controls the address sequencing of the control memory must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another.

The steps that the control must undergo during the execution of a single computer instruction are:

- The control address register is loaded by address of the first microinstruction that activates the instruction fetch routine. At the end of the fetch routine, the instruction is in the instruction register of the computer.
- The control memory next must go through the routine that determines the effective address of the operand. A machine instruction may have bits that specify various addressing modes, such as indirect address and index registers. When the effective address computation routine is completed, the address of the operand is available in the memory address register.
- The next step is to generate the microoperations that execute the instruction fetched from memory. The microoperation steps to be generated in processor registers depend on the operation code part of the instruction. Each instruction has its own microprogram routine stored in a given location of control memory.

The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a mapping process. A mapping procedure is a rule that transforms the instruction code into a control memory address. Once the required routine is reached, the microinstructions that execute the instruction may be sequenced by incrementing the control address register, but sometimes the sequence of microoperations will depend on values of certain status bits in processor registers. Microprograms that employ subroutines will require an external register for storing the return address. Return addresses cannot be stored in ROM because the unit has no writing capability.

When the execution of the instruction is completed, control must return to the fetch routine. This is accomplished by executing an unconditional branch microinstruction to the first address of the fetch routine. In summary, the address sequencing capabilities required in a control memory are:

1. Incrementing of the control address register.
2. Unconditional branch or conditional branch, depending on status bit conditions.
3. A mapping process from the bits of the instruction to an address for control memory.
4. A facility for subroutine call and return.

Figure 3 shows a block diagram of a control memory and the associated hardware needed for selecting the next microinstruction address.

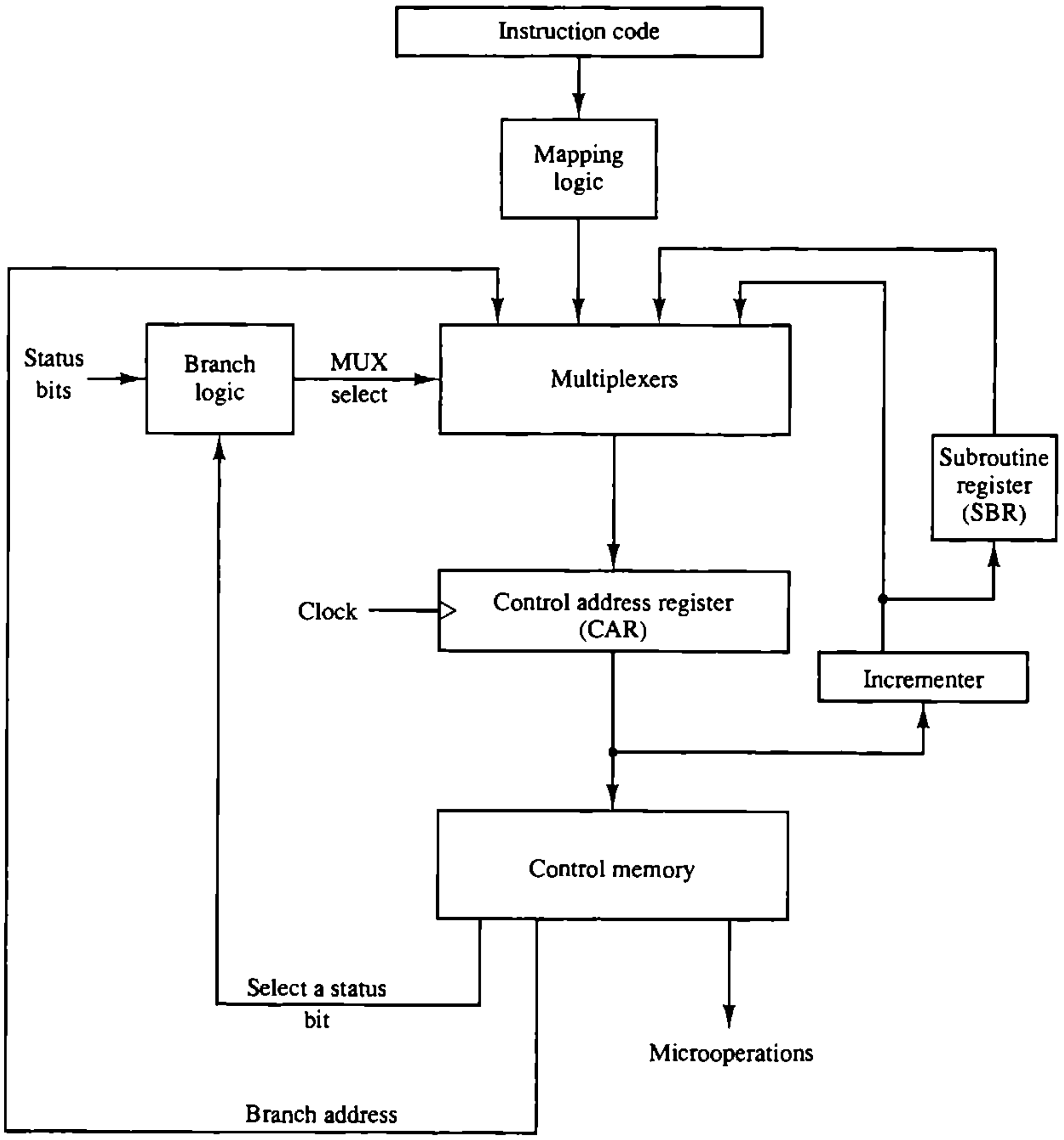


Figure 40 Selection of address for control memory.

Conditional Branching

The branch logic of Figure 3 provides decision-making capabilities in the control unit. The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and input or output status conditions. Information in these bits can be tested and actions initiated based on their condition: whether their value is 1 or 0. The status bits, together with the field in the microinstruction that specifies a branch address, control the conditional branch decisions generated in the branch logic.

The branch logic hardware may be implemented in a variety of ways. The simplest way is to test the specified condition and branch to the indicated address if the condition is met; otherwise, the address register is incremented.

An unconditional branch microinstruction can be implemented by loading the branch address from control memory into the control address register.

Mapping of Instruction

A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine for an instruction is located. The status bits for this type of branch are the bits in the operation code part of the instruction. For example, a computer with a simple instruction format as shown in Figure 4 has an operation code of four bits which can specify up to 16 distinct instructions. Assume further that the control memory has 128 words, requiring an address of seven bits. For each operation code there exists a microprogram routine in control memory that executes the instruction. One simple mapping process that converts the 4-bit operation code to a 7-bit address for control memory is shown in Figure 4. This mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the two least significant bits of the control address register. This provides for each computer instruction a microprogram routine with a capacity of four microinstructions. If the routine needs more than four microinstructions, it can use addresses 1000000 through 1111111. If it uses fewer than four microinstructions, the unused memory locations would be available for other routines.

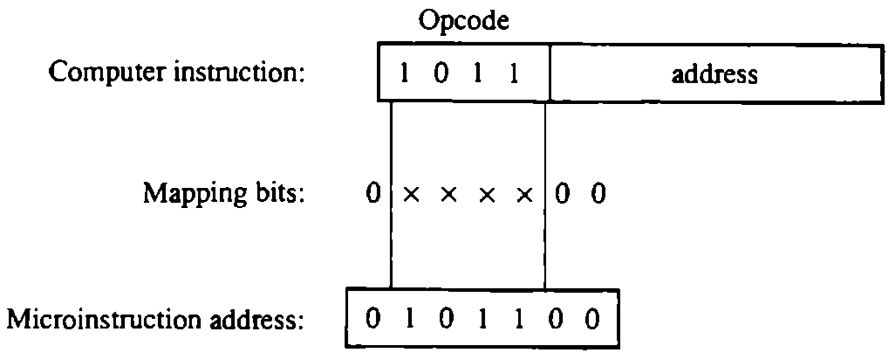


Figure 41 Mapping from instruction code to microinstruction address.

One can extend this concept to a more general mapping rule by using a ROM to specify the mapping function. In this configuration, the bits of the instruction specify the address of a mapping ROM. The contents of the mapping ROM give the bits for the control address register. In this way the microprogram routine that executes the instruction can be placed in any desired location in control memory. (How it is?). The mapping concept provides flexibility for adding instructions for control memory as the need arises.

The mapping function is sometimes implemented by means of an integrated circuit called programmable logic device or PLD. A PLD is similar to ROM in concept except that it uses AND and OR gates with internal electronic fuses. The interconnection between inputs, AND gates, OR gates, and outputs can be programmed as in ROM. A mapping function that can be expressed in terms of Boolean expressions can be implemented conveniently with a PLD.

H.W. Problem 7-8 in M. Morris Mano.

Subroutines

Subroutines are programs that are used by other routines to accomplish a particular task. A subroutine can be called from any point within the main body of the microprogram. Frequently, many microprograms contain identical sections of code. Microinstructions can be saved by employing subroutines that use common sections of microcode. For example, the sequence of microoperations needed to generate the effective address of the operand for an instruction is common to all memory reference instructions. This sequence could be a subroutine that is called from within many other routines to execute the effective address computation.

Microprograms that use subroutines must have a provision for storing the return address during a subroutine call and restoring the address during a subroutine return. This may be accomplished by placing the incremented output from the control address register into a subroutine register and branching to the beginning of the subroutine. The subroutine register can then become the source for transferring the address for the return to the main routine. The best way to structure a register file that stores addresses for subroutines is to organize the registers in a last-in, first-out (LIFO) stack. (Why?).

Microprogram Example

Once the configuration of a computer and its microprogrammed control unit is established, the designer's task is to generate the microcode for the control memory. This code generation is called microprogramming and is a process similar to conventional machine language programming. To appreciate this process, we present here a simple digital computer and show how it is microprogrammed. The computer used here is similar but not identical to the basic computer introduced in the previous lecture.

Computer Configuration

The block diagram of the computer is shown in Figure 5. It consists of two memory units: a main memory for storing instructions and data, and a control memory for storing the microprogram. Four registers are associated with the processor unit and two with the control unit. The processor registers are program counter PC, address register AR, data register DR, and accumulator register AC. The control unit has a control address register CAR and a subroutine register SBR. The control memory and its registers are organized as a microprogrammed control unit, as shown in Figure 3.

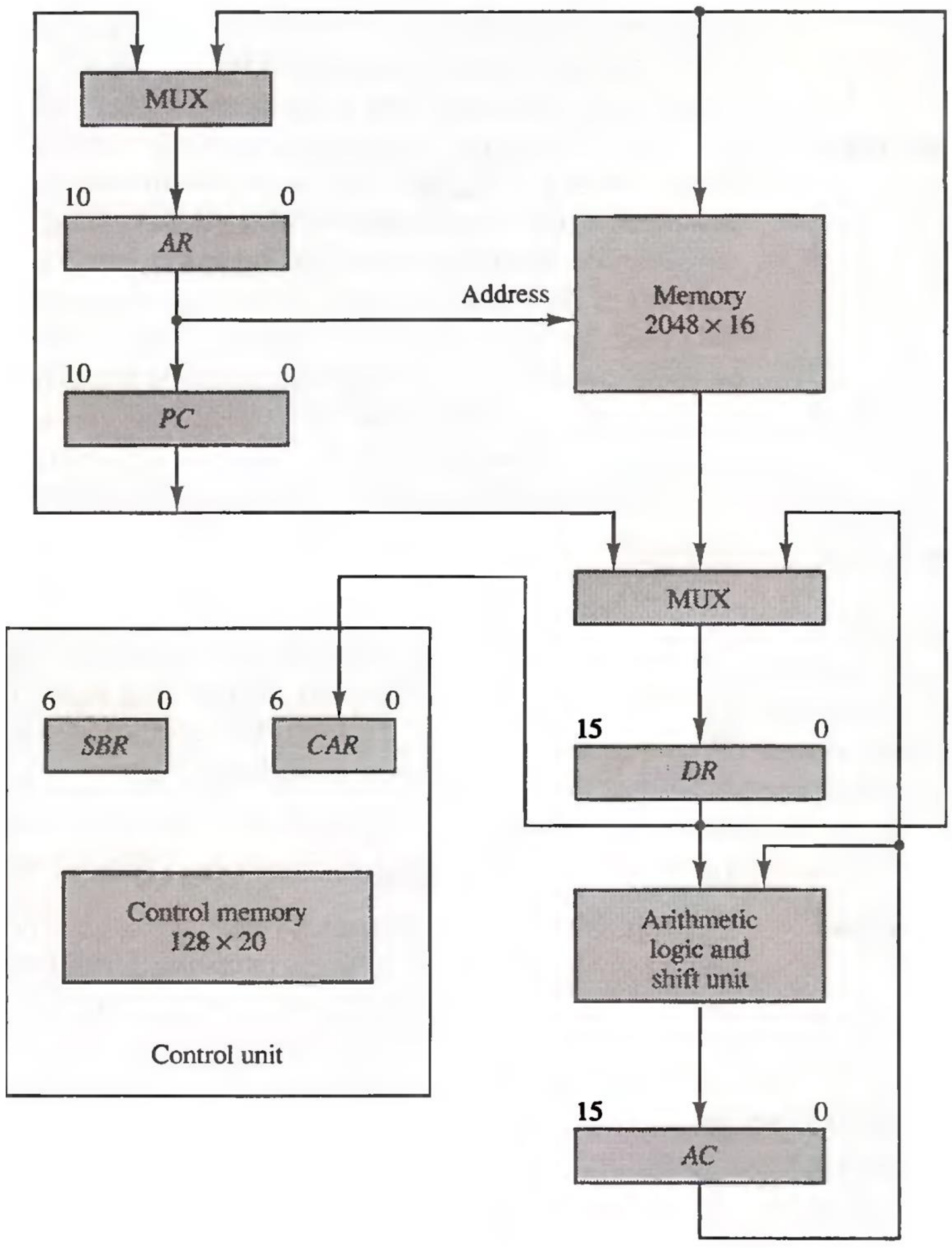
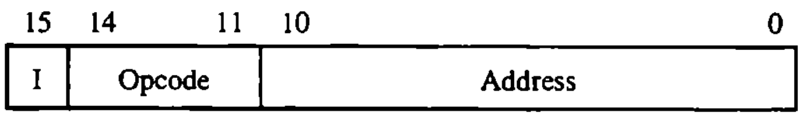


Figure 42 Computer hardware configuration.

The transfer of information among the registers in the processor is done through multiplexers rather than a common bus. DR can receive information from AC, PC, or memory. AR can receive information from PC or DR. PC can receive information only from AR. The arithmetic, logic, and shift unit performs microoperations with data from AC and DR and places the result in AC. Note that memory receives its address from AR. Input data written to memory come from DR, and data read from memory can go only to DR.

The computer instruction format is depicted in Figure 6(a). It consists of three fields: a 1-bit held for indirect addressing symbolized by I, a 4-bit operation code (opcode), and an 11-bit address field. Figure 6(b) lists four of the 16 possible memory-reference instructions.



(a) Instruction format

Symbol	Opcode	Description
ADD	0000	$AC \leftarrow AC + M[EA]$
BRANCH	0001	If $(AC < 0)$ then $(PC \leftarrow EA)$
STORE	0010	$M[EA] \leftarrow AC$
EXCHANGE	0011	$AC \leftarrow M[EA], M[EA] \leftarrow AC$

EA is the effective address

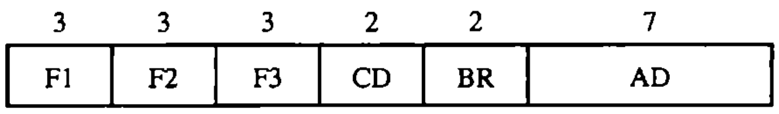
(b) Four computer instructions

Figure 43 Computer instructions.

It will be shown subsequently that each computer instruction must be microprogrammed. In order not to complicate the microprogramming example, only four instructions are considered here. It should be realized that 12 other instructions can be included and each instruction must be microprogrammed by the procedure outlined below.

Microinstruction Format

The microinstruction format for the control memory is shown in Figure 7. The 20 bits of the microiristruction are divided into four functional parts. The three fields Fl, F2, and F3 specify microoperations for the computer. The CD field selects status bit conditions. The BR field specifies the type of branch to be used. The AD field contains a branch address. The address field is seven bits wide, since the control memory has 128 = 2^7 words.



F1, F2, F3: Microoperation fields

CD: Condition for branching

BR: Branch field

AD: Address field

Figure 44 Microinstruction code format (20 bits).

The microoperations are subdivided into three fields of three bits each. The three bits in each field are encoded to specify seven distinct microoperations as listed in Table 1. This gives a total of 21 microoperations. No more than three microoperations can be chosen for a microinstruction, one from each field. If fewer than three microoperations are used, one or more of the fields will use the binary code 000 for no operation. As an illustration, a microinstruction can specify two simultaneous microoperations from F2 and F3 and none from Fl.

DR ← M[AR]   with F2 = 100

and   PC ← PC + 1   with F3 = 101

The nine bits of the microoperation fields will then be 000 100 101. It is important to realize that two or more conflicting microoperations cannot be specified simultaneously. For example, a microoperation field 010 001 000 has no meaning because it specifies the operations to clear AC to 0 and subtract DR from AC at the same time.

H.W. Problem 7-20 in M. Morris Mano.

Each microoperation in Table 1 is defined with a register transfer statement and is assigned a symbol for use in a symbolic microprogram. All transfer-type microoperations symbols use five letters. The first two letters designate the source register, the third letter is always a T, and the last two letters designate the destination register. For example, the microoperation that specifies the transfer AC←DR (Fl = 100) has the symbol DRTAC, which stands for a transfer from DR to AC.

The CD (condition) field consists of two bits which are encoded to specify four status bit conditions as listed in Table 1.

The BR (branch) field consists of two bits. It is used, in conjunction with the address field AD, to choose the address of the next microinstruction.

The return from subroutine is accomplished with a BR field equal to 10. This causes the transfer of the return address from SBR to CAR. The mapping from the operation code bits of the instruction to an address for CAR is accomplished when the BR field is equal to 11. This mapping is as depicted in Figure 4. The bits of the operation code are in DR(11-14) after an instruction is read from memory. Note that the last two conditions in the BR field are independent of the values in the CD and AD fields.

Table 15   Symbols and Binary Code for Microinstruction Fields

F1	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC + DR$	ADD
010	$AC \leftarrow 0$	CLRAC
011	$AC \leftarrow AC + 1$	INCAC
100	$AC \leftarrow DR$	DRTAC
101	$AR \leftarrow DR(0\text{–}10)$	DRTAR
110	$AR \leftarrow PC$	PCTAR
111	$M[AR] \leftarrow DR$	WRITE

F2	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC - DR$	SUB
010	$AC \leftarrow AC \vee DR$	OR
011	$AC \leftarrow AC \wedge DR$	AND
100	$DR \leftarrow M[AR]$	READ
101	$DR \leftarrow AC$	ACTDR
110	$DR \leftarrow DR + 1$	INCDR
111	$DR(0\text{-}10) \leftarrow PC$	PCTDR

F3	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC \oplus DR$	XOR
010	$AC \leftarrow \overline{AC}$	COM
011	$AC \leftarrow \text{shl } AC$	SHL
100	$AC \leftarrow \text{shr } AC$	SHR
101	$PC \leftarrow PC + 1$	INCPC
110	$PC \leftarrow AR$	ARTPC
111	Reserved	

CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	$DR(15)$	I	Indirect address bit
10	$AC(15)$	S	Sign bit of AC
11	$AC = 0$	Z	Zero value in AC

BR	Symbol	Function
00	JMP	$CAR \leftarrow AD$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
01	CALL	$CAR \leftarrow AD$, $SBR \leftarrow CAR + 1$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
10	RET	$CAR \leftarrow SBR$ (Return from subroutine)
11	MAP	$CAR(2\text{–}5) \leftarrow DR(11\text{–}14)$, $CAR(0,1,6) \leftarrow 0$

Symbolic Microinstructions

The symbols defined in Table 1 can be used to specify microinstructions in symbolic form. A symbolic microprogram can be translated into its binary equivalent by means of an assembler. A microprogram assembler is similar in concept to a conventional computer assembler. The simplest and most straightforward way to formulate an assembly language for a microprogram is to define symbols for each held of the microinstruction and to give users the capability for defining their own symbolic addresses.

Each line of the assembly language microprogram defines a symbolic microinstruction. Each symbolic microinstruction is divided into five fields: label, microoperations, CD, BR, and AD. The fields specify the following information.

1. The label held may be empty or it may specify a symbolic address. A label is terminated with a colon (:).
2. The microoperations field consists of one, two, or three symbols, separated by commas, from those defined in Table 1. There may be no more than one symbol from each F field. The NOP symbol is used when the microinstruction has no microoperations. This will be translated by the assembler to nine zeros.
3. The CD field has one of the letters U, I, S, or Z.
4. The BR field contains one of the four symbols defined in Table 1.
5. The AD field specifies a value for the address field of the microinstruction in one of three possible ways:
   a. With a symbolic address, which must also appear as a label.
   b. With the symbol NEXT to designate the next address in sequence.
   c. When the BR field contains a RET or MAP symbol, the AD field is left empty and is converted to seven zeros by the assembler.

We will use also the pseudoinstruction ORG to define the origin, or first address, of a microprogram routine. Thus the symbol ORG 64 informs the assembler to place the next microinstruction in control memory at decimal address 64, which is equivalent to the binary address 1000000.

The Fetch Routine

The control memory has 128 words, and each word contains 20 bits. To microprogram the control memory, it is necessary to determine the bit values of each of the 128 words. The first 64 words (addresses 0 to 63) are to be occupied by the routines for the 16 instructions. The last 64 words may be used for any other purpose. A convenient starting location for the fetch routine is address 64. The microinstructions needed for the fetch routine are

$AR \leftarrow PC$

$DR \leftarrow M[AR], \ PC \leftarrow PC + 1$

$AR \leftarrow DR(0\text{-}10), \ CAR(2\text{-}5) \leftarrow DR(11\text{-}14), \ CAR(0,l,6) \leftarrow 0$

The fetch routine needs three microinstructions, which are placed in control memory at addresses 64, 65, and 66. Using the assembly language conventions defined previously, we can write the symbolic microprogram for the fetch routine as follows:

```
                ORG 64
FETCH:  PCTAR           U      JMP     NEXT
                READ, INCPC     U      JMP     NEXT
                DRTAR           U      MAP
```

The translation of the symbolic microprogram to binary produces the following binary microprogram. The bit values are obtained from Table 1.

Binary Address	F1	F2	F3	CD	BR	AD
1000000	110	000	000	00	00	1000001
1000001	000	100	101	00	00	1000010
1000010	101	000	000	00	11	0000000

The three microinstructions that constitute the fetch routine have been listed in three different representations. The register transfer representation shows the internal register transfer operations that each microinstruction implements. The symbolic representation is useful for writing microprograms in an assembly language format. The binary representation is the actual internal content that must be stored in control memory. It is customary to write microprograms in symbolic form and then use an assembler program to obtain a translation to binary.

Symbolic Microprogram

The execution of the third (MAP) microinstruction in the fetch routine results in a branch to address 0xxxx00, where xxxx are the four bits of the operation code. For example, if the instruction is an ADD instruction whose operation code is 0000, the MAP microinstruction will transfer to CAR the address 0000000, which is the start address for the ADD routine in control memory. The first address for the BRANCH and STORE routines are 0 0001 00 (decimal 4) and 0 0010 00 (decimal 8), respectively. The first address for the other 13 routines are at address values 12, 16, 20,..., 60. This gives four words in control memory for each routine.

In each routine we must provide microinstructions for evaluating the effective address and for executing the instruction. The indirect address mode is associated with all memory-reference instructions. A saving in the number of control memory words may be achieved if the microinstructions for the indirect address are stored as a subroutine. This subroutine, symbolized by INDRCT, is located right after the fetch routine, as shown in Table 2. The table also shows the symbolic microprogram for the fetch routine and the microinstruction routines that execute four computer instructions.

To see how the transfer and return from the indirect subroutine occurs, assume that the MAP microinstruction at the end of the fetch routine caused a branch to address 0, where the ADD routine is stored. The first microinstruction in the ADD routine calls subroutine INDRCT, conditioned on status bit I. If I = 1, a branch to INDRCT occurs and the return address (address 1 in this case) is stored in the subroutine register SBR. The INDRCT subroutine has two microinstructions:

```
INDRCT:   READ   U       JMP      NEXT
                 DRTAR  U        RET
```

Table 16   Symbolic Microprogram (Partial)

Label	Microoperations	CD	BR	AD
	ORG 0			
ADD:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
	ORG 4			
BRANCH:	NOP	S	JMP	OVER
	NOP	U	JMP	FETCH
OVER:	NOP	I	CALL	INDRCT
	ARTPC	U	JMP	FETCH
	ORG 8			
STORE:	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 12			
EXCHANGE:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ACTDR, DRTAC	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 64			
FETCH:	PCTAR	U	JMP	NEXT
	READ, INCPC	U	JMP	NEXT
	DRTAR	U	MAP	
INDRCT:	READ	U	JMP	NEXT
	DRTAR	U	RET	

Remember that an indirect address considers the address part of the instruction as the address where the effective address is stored rather than the address of the operand. Therefore, the memory has to be accessed to get the effective address, which is then transferred to AR. The return from subroutine (RET) transfers the address from SBR to CAR, thus returning to the second microinstruction of the ADD routine.

The execution of the ADD instruction is carried out by the microinstructions at addresses 1 and 2. The first microinstruction reads the operand from memory into DR. The second microinstruction performs an add microoperation with the content of DR and AC and then jumps back to the beginning of the fetch routine.

H.W. Problem 7-13 in M. Morris Mano.

The BRANCH instruction should cause a branch to the effective address if AC < 0. The AC will be less than zero if its sign is negative, which is detected from status bit S being a 1. The BRANCH routine in Table 2 starts by checking the value of S. If S is equal to 0, no branch occurs and the next microinstruction causes a jump back to the fetch routine without altering the content of PC. If S is equal to 1, the first JMP microinstruction transfers control to location OVER. The microinstruction at this location calls the INDRCT subroutine if I = 1. The effective address is then transferred from AR to PC and the microprogram jumps back to the fetch routine.

H.W. What is the value of AD field that labeled by OVER in Table 2.

The STORE routine again uses the INDRCT subroutine if I = 1. The content of AC is transferred into DR. A memory write operation is initiated to store the content of DR in a location specified by the effective address in AR.

The EXCHANGE routine reads the operand from the effective address and places it in DR. The contents of DR and AC are interchanged in the third microinstruction. This interchange is possible when the registers are of the edge-triggered type. The original content of AC that is now in DR is stored back in memory.

Note that Table 2 contains a partial list of the microprogram. Only four out of 16 possible computer instructions have been microprogrammed. Also, control memory words at locations 69 to 127 have not been used. Instructions such as multiply, divide, and others that require a long sequence of microoperations will need more than four microinstructions for their execution. Control memory words 69 to 127 can be used for this purpose.

Binary Microprogram

The symbolic microprogram is a convenient form for writing microprograms in a way that people can read and understand. But this is not the way that the microprogram is stored in memory. The symbolic microprogram must be translated to binary either by means of an assembler program or by the user if the microprogram is simple enough as in this example.

The equivalent binary form of the microprogram is listed in Table 3. The addresses for control memory are given in both decimal and binary. The binary content of each microinstruction is derived from the symbols and their equivalent binary values as defined in Table 1.

Note that address 3 has no equivalent in the symbolic microprogram since the ADD routine has only three microinstructions at addresses 0,1, and 2. The next routine starts at address 4. Even though address 3 is not used, some binary value must be specified for each word in control memory. We could have specified all 0's in the word since this location will never be used. However, if some unforeseen error occurs, or if a noise signal sets CAR to the value of 3, it will be wise to jump to address 64, which is the beginning of the fetch routine.

The binary microprogram listed in Table 3 specifies the word content of the control memory. When a ROM is used for the control memory, the microprogram binary list provides the truth table for fabricating the unit. This fabrication is a hardware process and consists of creating a mask for the ROM so as to produce the l's and 0's for each word. The bits of ROM are fixed once the internal links are fused during the hardware production. The ROM is made of IC packages that can be removed if necessary and replaced by other packages. To modify the instruction set of the computer, it is necessary to generate a new microprogram and mask a new ROM. The old one can be removed and the new one inserted in its place.

Table 17 Binary Microprogram for Control Memory (Partial)

Micro Routine	Address		Binary Microinstruction					
	Decimal	Binary	F1	F2	F3	CD	BR	AD
ADD	0	0000000	000	000	000	01	01	1000011
	1	0000001	000	100	000	00	00	0000010
	2	0000010	001	000	000	00	00	1000000
	3	0000011	000	000	000	00	00	1000000
BRANCH	4	0000100	000	000	000	10	00	0000110
	5	0000101	000	000	000	00	00	1000000
	6	0000110	000	000	000	01	01	1000011
	7	0000111	000	000	110	00	00	1000000
STORE	8	0001000	000	000	000	01	01	1000011
	9	0001001	000	101	000	00	00	0001010
	10	0001010	111	000	000	00	00	1000000
	11	0001011	000	000	000	00	00	1000000
EXCHANGE	12	0001100	000	000	000	01	01	1000011
	13	0001101	001	000	000	00	00	0001110
	14	0001110	100	101	000	00	00	0001111
	15	0001111	111	000	000	00	00	1000000
FETCH	64	1000000	110	000	000	00	00	1000001
	65	1000001	000	100	101	00	00	1000010
	66	1000010	101	000	000	00	11	0000000
INDRCT	67	1000011	000	100	000	00	00	1000100
	68	1000100	101	000	000	00	10	0000000

If a writable control memory is employed, the ROM is replaced by a RAM. The advantage of employing a RAM for the control memory is that the microprogram can be altered simply by writing a new pattern of l's and 0's without resorting to hardware procedures. A writable control memory possesses the flexibility of choosing the instruction set of a computer dynamically by changing the microprogram under processor control. However, most microprogrammed systems use a ROM for the control memory because it is cheaper and faster than a RAM and also to prevent the occasional user from changing the architecture of the system.

Design of Control Unit

The bits of the microinstruction are usually divided into fields, with each field defining a distinct, separate function. The various fields encountered in instruction formats provide control bits to initiate microoperations in the system, special bits to specify the way that the next address is to be evaluated, and an address field for branching. The number of control bits that initiate microoperations can be reduced by grouping mutually exclusive variables into fields and encoding the k bits in each field to provide 2^k microoperations. Each field requires a decoder to produce the corresponding control signals. This method reduces the size of the microinstruction bits but requires additional hardware external to the control memory. It also increases the delay time of the control signals because they must propagate through the decoding circuits.

The encoding of control bits was demonstrated in the programming example of the preceding section. The nine bits of the microoperation field are divided into three subfields of three bits each. The control memory output of each subfield must be decoded to provide the distinct microoperations. The outputs of the decoders are connected to the appropriate inputs in the processor unit.

Figure 8 shows the three decoders and some of the connections that must be made from their outputs. Each of the three fields of the microinstruction presently available in the output of control memory are decoded with a 3x8 decoder to provide eight outputs. Each of these outputs must be connected to the proper circuit to initiate the corresponding microoperation as specified in Table 1.

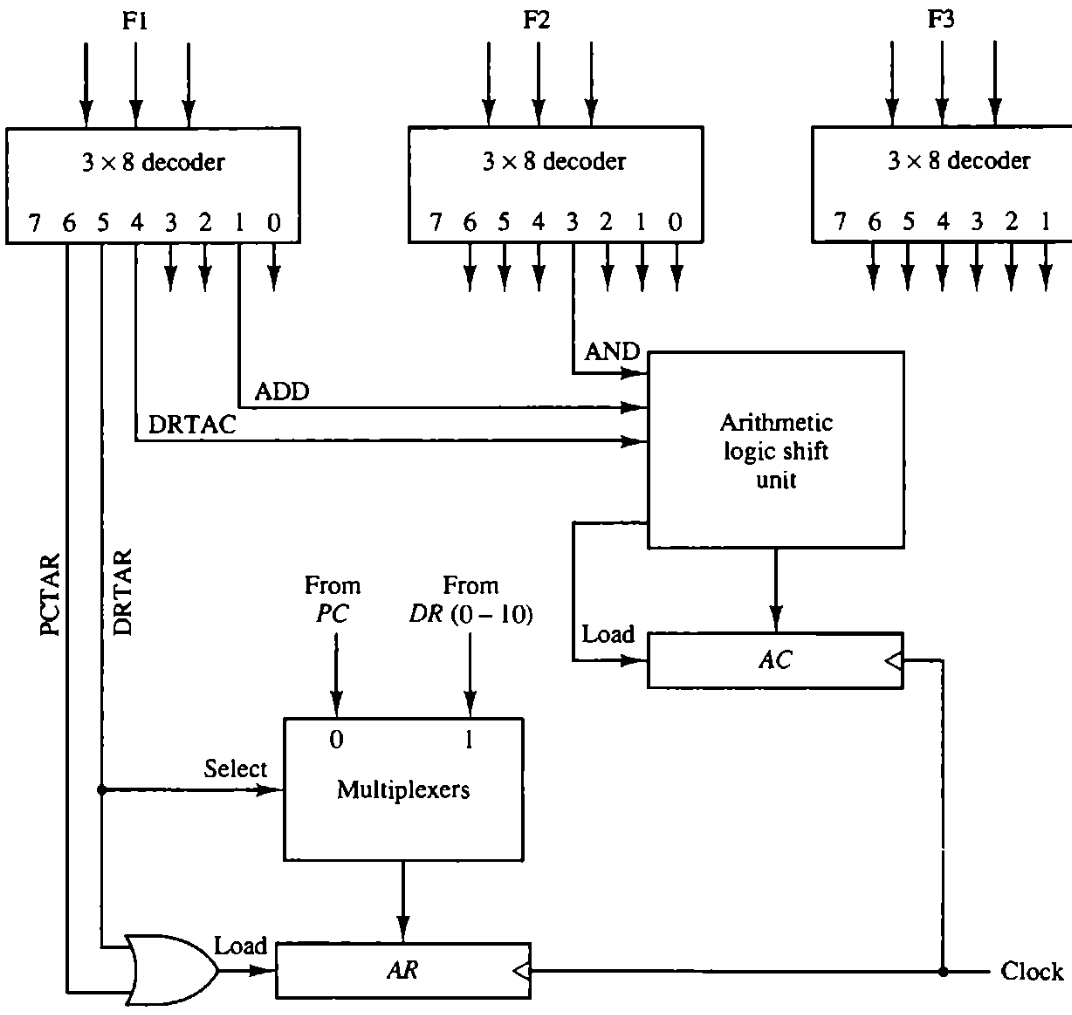


Figure 45 Decoding of microoperation fields.

H.W. Show the signals that control the DR register and.

Microprogram Sequencer

The basic components of a microprogrammed control unit are the control memory and the circuits that select the next address. The address selection part is called a microprogram sequencer. A microprogram sequencer can be constructed with digital functions to suit a particular application. However, just as there are large ROM units available in integrated circuit packages, so are general-purpose sequencers suited for the construction of microprogram control units. To guarantee a wide range of acceptability, an integrated circuit sequencer must provide an internal organization that can be adapted to a wide range of applications.

The purpose of a microprogram sequencer is to present an address to the control memory so that a microinstruction may be read and executed. The next-address logic of the sequencer determines the specific address source to be loaded into the control address register. The choice of the address source is guided by the next-address information bits that the sequencer receives from the present microinstruction. Commercial sequencers include within the unit an internal register stack used for temporary storage of addresses during microprogram looping and subroutine calls. Some sequencers provide an output register which can function as the address register for the control memory.

To illustrate the internal structure of a typical microprogram sequencer we will show a particular unit that is suitable for use in the microprogram computer example developed in the preceding lecture. The block diagram of the microprogram sequencer is shown in Figure 9. The control memory is included in the diagram to show the interaction between the sequencer and the memory attached to it.

The diagram in Figure 9 shows a single subroutine register, a typical sequencer will have a register stack about four to eight levels deep. In this way, a number of subroutines can be active at the same time. A push and pop operation, in conjunction with a stack pointer, stores and retrieves the return address during the call and return microinstructions.

The CD (condition) field of the microinstruction selectgs one of the status bits in the second multiplexer. If the bit selected is equal to 1, the T (test) variable is equal to 1; otherwise, it is equal to 0. The T value together with the two bits from the BR (branch) field go to an input logic circuit. The input logic in a particular sequencer will determine the type of operations that are available in the unit. Typical sequencer operations are: increment, branch or jump, call and return from subroutine, load an external address, push or pop the stack, and other address sequencing operations. With three inputs, the sequencer can provide up to eight address sequencing operations. Some commercial sequencers have three or four inputs in addition to the T input and thus provide a wider range of operations.

The input logic circuit in Figure 9 has three inputs, I_0, I_1, and T, and three outputs, S_0, S_1, and L. Variables S_0 and S_1 select one of the source addresses for CAR. Variable L enables the load input in SBR. The binary values of the two selection variables determine the path in the multiplexer. For example, with $S_1 S_0 = 10$, multiplexer input number 2 is selected and establishes a transfer path from SBR to CAR. Note that each of the four inputs as well as the output of MUX 1 contains a 7-bit address.

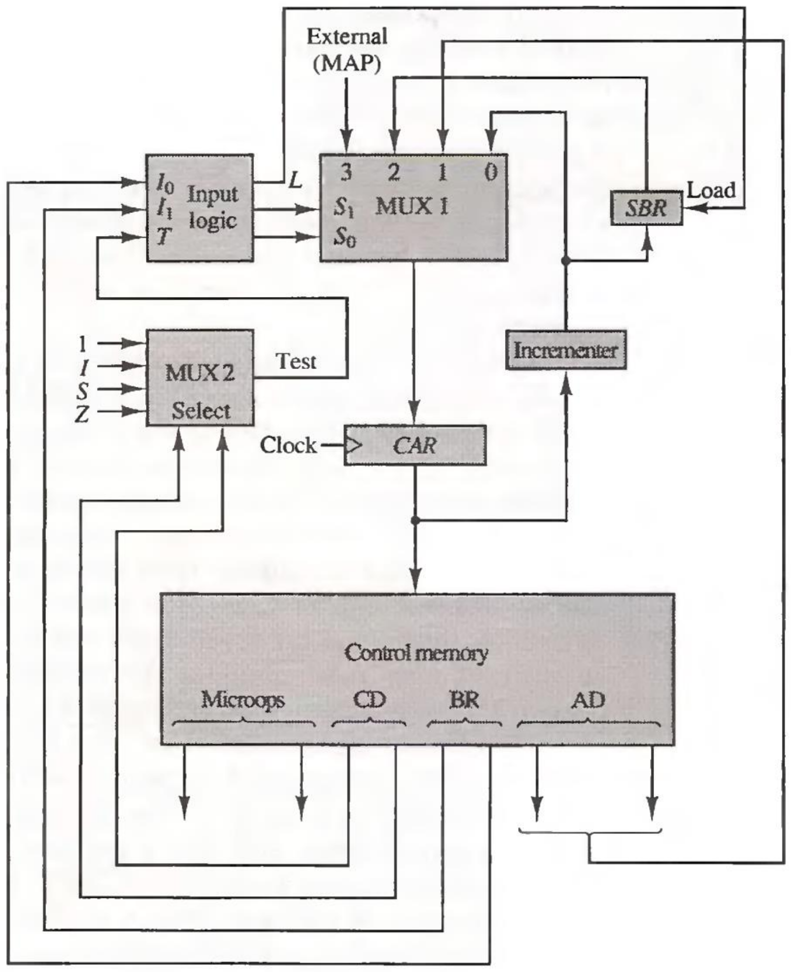


Figure 46 Microprogram sequencer for a control memory.

The truth table for the input logic circuit is shown in Table 4. Inputs I_1 and I_0 are identical to the bit values in the BR field. The function listed in each entry was defined in Table 1. The bit values for S_1 and S_0 are determined from the stated function and the path in the multiplexer that establishes the required transfer. The subroutine register is loaded with the incremented value of CAR during a call microinstruction (BR = 01) provided that the status bit condition is satisfied (T = 1). The truth table can be used to obtain the simplified Boolean functions for the input logic circuit:

$$S_1 = I_1$$

$$S_0 = I_1 I_0 + I_1' T$$

$$L = I_1' I_0 T$$

The circuit can be constructed with three AND gates, an OR gate, and an inverter.

Table 18 Input Logic Truth Table for Microprogram Sequencer

BR Field	Input I_1 I_0 T	MUX 1 S_1 S_0	Load *SBR* L
0 0	0 0 0	0 0	0
0 0	0 0 1	0 1	0
0 1	0 1 0	0 0	0
0 1	0 1 1	0 1	1
1 0	1 0 ×	1 0	0
1 1	1 1 ×	1 1	0

H.W. Problem 7-22 in M. Morris Mano.

Note that the incrementer circuit in the sequencer of Figure 9 is not a counter constructed with flip-flops but rather a combinational circuit constructed with gates. A combinational circuit incrementer can be designed by cascading a series of half-adder circuits. The output carry from one stage must be applied to the input of the next stage. One input in the first least significant stage must be equal to 1 to provide the increment-by-one operation.

H.W. Problem 7-23 in M. Morris Mano.

Central Processing Unit

Central processing unit (CPU) is the part of the computer that performs the bulk of data-processing operations

The CPU is made up of three major parts, as shown in Figure 47.

1. **The register set** stores intermediate data used during the execution of the instructions.
2. **The arithmetic logic unit (ALU)** performs the required microoperations for executing the instructions.
3. **The control unit** supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.

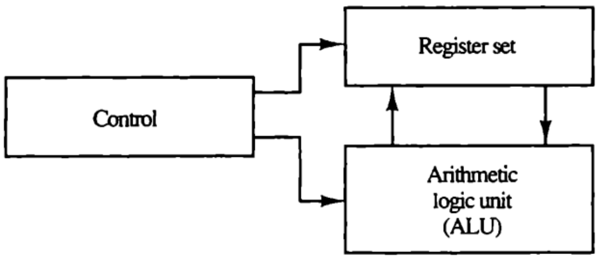


Figure 47 Major components of

Computer architecture is sometimes defined as the computer structure and behavior as seen by the programmer that uses machine language instructions.

This includes the **instruction formats**, **addressing modes**, **the instruction set**, and **the general organization of the CPU registers**.

One boundary where the computer designer and the computer programmer see the same machine is the part of the CPU associated with the instruction set.

The point of view for:

Computer designer	Computer programmer
The computer instruction set provides the specifications for the design of the CPU. The design of a CPU is a task that in large part involves choosing the hardware for implementing the machine instructions.	He must be aware of the register set, the memory structure, the type of data supported by the instructions, and the function that each instruction performs.

General Register Organization

Having to refer to memory locations for such applications is time consuming because memory access is the most time-consuming operation in a computer.

It is more convenient and more efficient to store these intermediate values in processor registers.

When a large number of registers are included in the CPU, it is most efficient to connect them through a common bus system. The registers communicate with each other not only for direct data transfers, but also while performing various microoperations. Hence it is necessary to provide a common unit that can perform all the arithmetic, logic, and shift microoperations in the processor.

Bus system:

A bus organization for seven CPU registers is shown in Figure 48. The output of each register is connected to two multiplexers (MUX) to form the two buses A and B.

For example, to perform the operation

$$R1 \leftarrow R2 + R3$$

the control must provide binary selection variables to the following selector inputs:

1. MUX A selector (SELA): to place the content of R2 into bus A.
2. MUX B selector (SELB): to place the content of R3 into bus B.
3. ALU operation selector (OPR): to provide the arithmetic addition A + B.
4. Decoder destination selector (SELD): to transfer the content of the output bus into R1.

Control Word

There are 14 binary selection inputs in the unit, and their combined value specifies a control word. The 14-bit control word is defined in Figure 48(b).

The encoding of the register selections is specified in Table 1. When SELA or SELB is 000, the corresponding multiplexer selects the external input data. When SELD = 000, no destination register is selected but the contents of the output bus are available in the external output.

The ALU provides arithmetic and logic operations. In addition, the CPU must provide shift operations. The shifter may be placed in the input of the ALU to provide a preshift capability, or at the output of the ALU to provide postshifting capability. In some cases, the shift operations are included with the ALU.

The encoding of the ALU operations for the CPU is specified in Table 2. The OPR field has five bits and each operation is designated with a symbolic name.

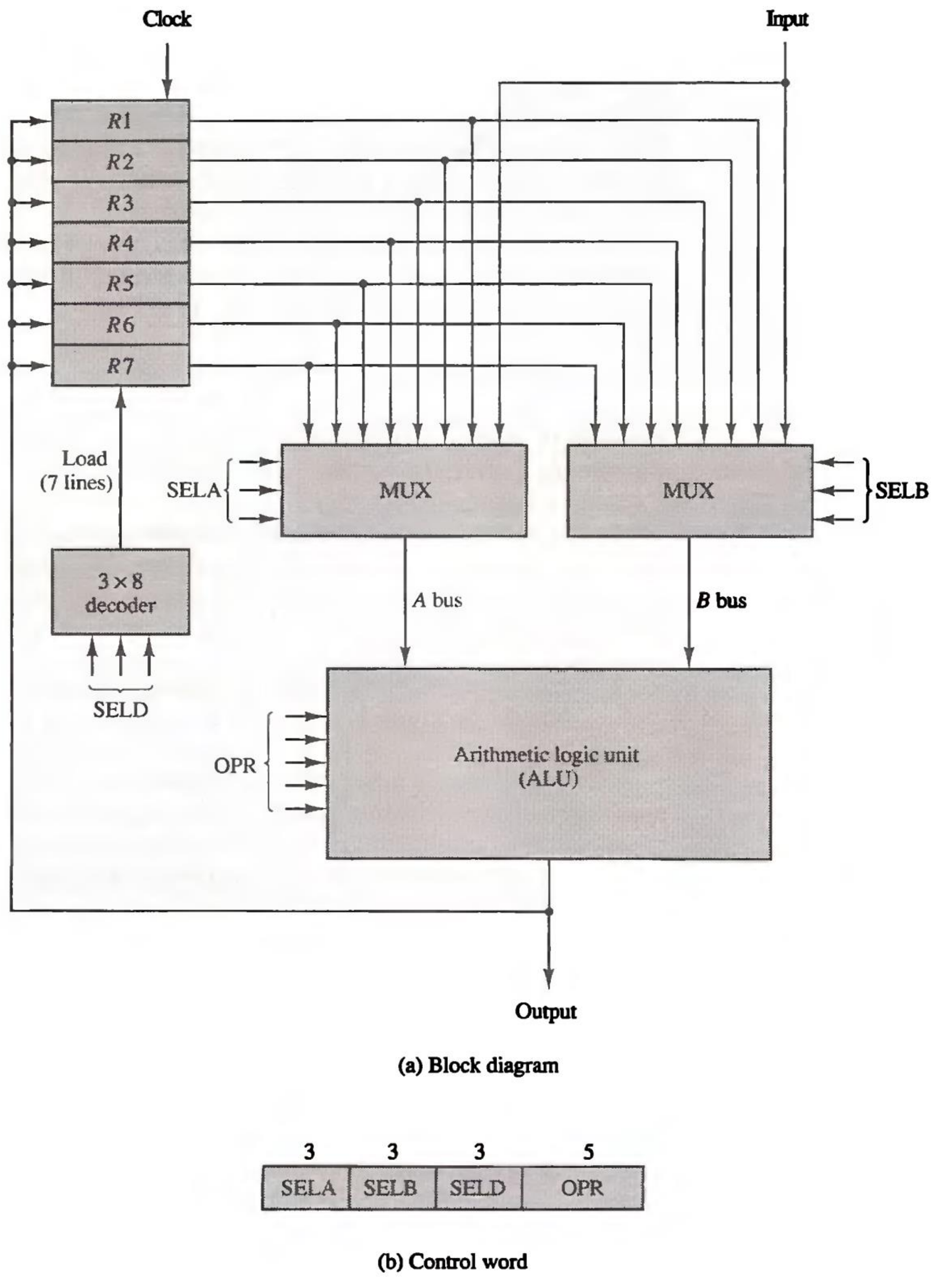


Figure 48 Register set with

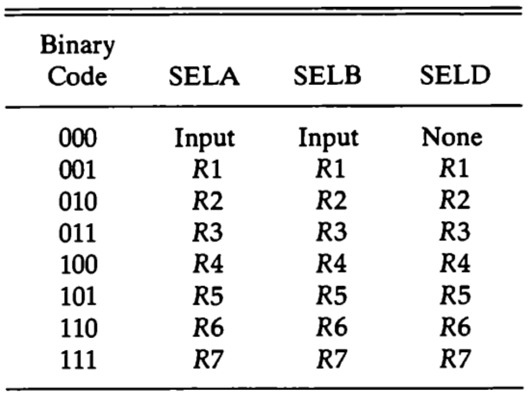
Examples of Microoperations

A control word of 14 bits is needed to specify a microoperation in the CPU. The control word for a given microoperation can be derived from the selection variables. For example, the subtract microoperation given by the statement

R1 ← R2 - R3

specifies R2 for the A input of the ALU, R3 for the B input of the ALU, R1 for the destination register, and an ALU operation to subtract A - B. Thus the control word is specified by the four fields and the corresponding binary value for each field is obtained from the encoding listed in Tables 1 and 2. The binary control word for the subtract

Table 19 Encoding of Register Selection

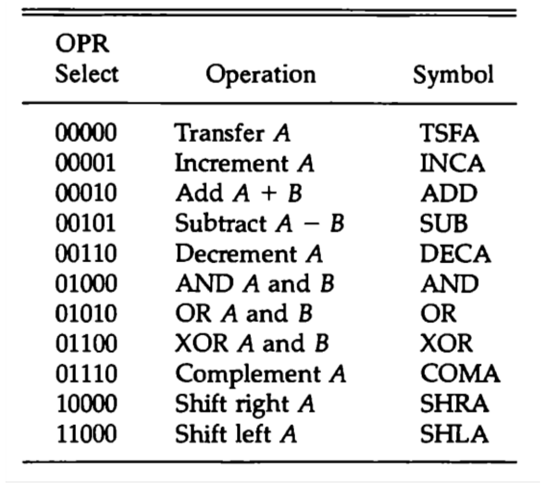


Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	*R*1	*R*1	*R*1
010	*R*2	*R*2	*R*2
011	*R*3	*R*3	*R*3
100	*R*4	*R*4	*R*4
101	*R*5	*R*5	*R*5
110	*R*6	*R*6	*R*6
111	*R*7	*R*7	*R*7

microoperation is 010 011 001 00101 and is obtained as follows:

Field:	SELA	SELB	SELD	OPR
Symbol:	R2	R3	R1	SUB
Control word:	010	011	001	00101

Table 20 Encoding of ALU Operations



OPR Select	Operation	Symbol
00000	Transfer *A*	TSFA
00001	Increment *A*	INCA
00010	Add *A* + *B*	ADD
00101	Subtract *A* − *B*	SUB
00110	Decrement *A*	DECA
01000	AND *A* and *B*	AND
01010	OR *A* and *B*	OR
01100	XOR *A* and *B*	XOR
01110	Complement *A*	COMA
10000	Shift right *A*	SHRA
11000	Shift left *A*	SHLA

The control word for this microoperation and a few others are listed in Table 3.

Table 21 Examples of Microoperations for the CPU

	Symbolic Designation				
Microoperation	SELA	SELB	SELD	OPR	Control Word
$R1 \leftarrow R2 - R3$	R2	R3	R1	SUB	010 011 001 00101
$R4 \leftarrow R4 \vee R5$	R4	R5	R4	OR	100 101 100 01010
$R6 \leftarrow R6 + 1$	R6	—	R6	INCA	110 000 110 00001
$R7 \leftarrow R1$	R1	—	R7	TSFA	001 000 111 00000
Output $\leftarrow R2$	R2	—	None	TSFA	010 000 000 00000
Output $\leftarrow$ Input	Input	—	None	TSFA	000 000 000 00000
$R4 \leftarrow$ shl $R4$	R4	—	R4	SHLA	100 000 100 11000
$R5 \leftarrow 0$	R5	R5	R5	XOR	101 101 101 01100

Stack Organization

A useful feature that is included in the CPU of most computers is a stack or last-in, first-out (LIFO) list.

A **stack** is a storage device that stores information in such a manner that the item stored last is the first item retrieved. The operation of a stack can be compared to a stack of trays. The last tray placed on top of the stack is the first to be taken off.

The stack in digital computers is essentially a memory unit with an address register that can count only (after an initial value is loaded into it). The register that holds the address for the stack is called a **stack pointer** (SP) because its value always points at the top item in the stack. Contrary to a stack of trays where the tray itself may be taken out or inserted, the physical registers of a stack are always available for reading or writing. It is the content of the word that is inserted or deleted.

push (or push-down): the operation of insertion.

pop (or pop-up): the operation of deletion.

These operations are simulated by incrementing or decrementing the stack pointer register.

Register Stack

A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers. Figure 3 shows the organization of a 64-word register stack. The stack pointer register SP contains a binary number whose value is equal to the address of the word that is currently on top of the stack. Three items are placed in the stack: A, B, and C, in that order. Item C is on top of the stack so that the content of SP is now 3. To remove the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of SP. Item B is now on top of the stack since SP holds address 2. To insert a new item, the stack is pushed by incrementing SP and writing a word in the next-higher location in the stack. Note that item C has been read out but not physically removed. This does not matter because when the stack is pushed, a new item is written in its place.

In a 64-word stack, the stack pointer contains 6 bits because $2^6 = 64$. Since SP has only six bits, it cannot exceed a number greater than 63 (111111 in binary). When 63 is incremented by 1, the result is 0 since 111111 + 1 =

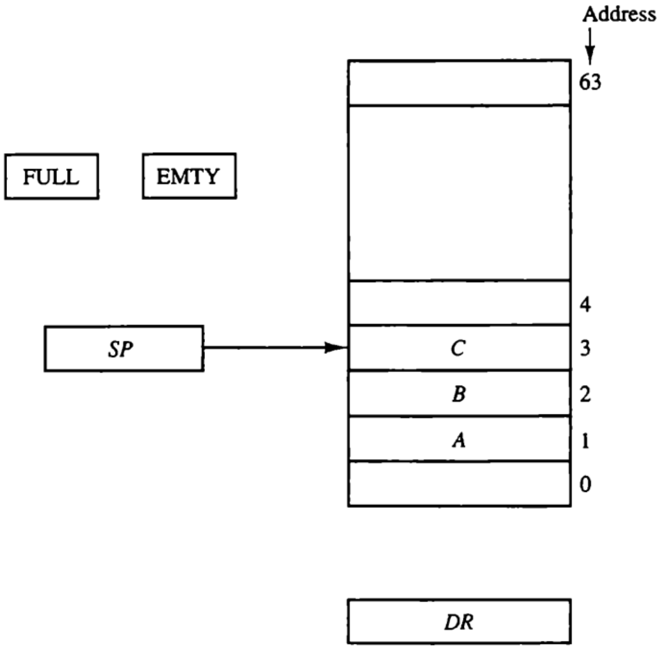


Figure 49 Block diagram of a 64-word stack.

1000000 in binary, but SP can accommodate only the six least significant bits. Similarly, when 000000 is decremented by 1, the result is 111111. The one-bit register FULL is set to 1 when the stack is full, and the one-bit register EMTY is set to 1 when the stack is empty of items. DR is the data register that holds the binary data to be written into or read out of the stack.

Initially, SP is cleared to 0, EMTY is set to 1, and FULL is cleared to 0, so that SP points to the word at address 0 and the stack is marked empty and not full. If the stack is not full (if FULL = 0), a new item is inserted with a push operation.

The push operation is implemented with the following sequence of microoperations:

SP ⟵ SP + 1	Increment stack pointer
M[SP] ⟵ DR	Write item on top of the stack
If (SP = 0) then (FULL ⟵ 1)	Check if stack is full
EMTY ⟵ 0	Mark the stack not empty

The first item stored in the stack is at address 1. The last item is stored at address 0. If SP reaches 0, the stack is full of items, so FULL is set to 1. This condition is reached if the top item prior to the last push was in location 63 and, after incrementing SP, the last item is stored in location 0. Once an item is stored in location 0, there are no more empty registers in the stack. If an item is written in the stack, obviously the stack cannot be empty, so EMTY is cleared to 0.

A new item is deleted from the stack if the stack is not empty (if pop EMTY = 0).

The pop operation consists of the following sequence of micro- operations:

DR ⟵ M[SP]	Read item from the top of stack
SP ⟵ SP - 1	Decrement stack pointer
If (SP = 0) then (EMTY ⟵ 1)	Check if stack is empty
FULL ⟵ 0	Mark the stack not full

Note also that an erroneous operation will result if the stack is pushed when FULL = 1 or popped when EMTY = 1.

Memory Stack

A stack can exist as a stand-alone unit as in Figure 3 or can be implemented in a random-access memory attached to a CPU. The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer. Figure 4 shows a portion of computer memory partitioned into three segments: program, data, and stack.

The **program counter PC** points at the address of the next instruction in the program.

The **address register AR** points at an array of data.

The **stack pointer SP** points at the top of the stack.

The three registers are connected to a common address bus, and either one can provide an address for memory.

PC is used during the fetch phase to read an instruction.

AR is used during the execute phase to read an operand.

SP is used to push or pop items into or from the stack.

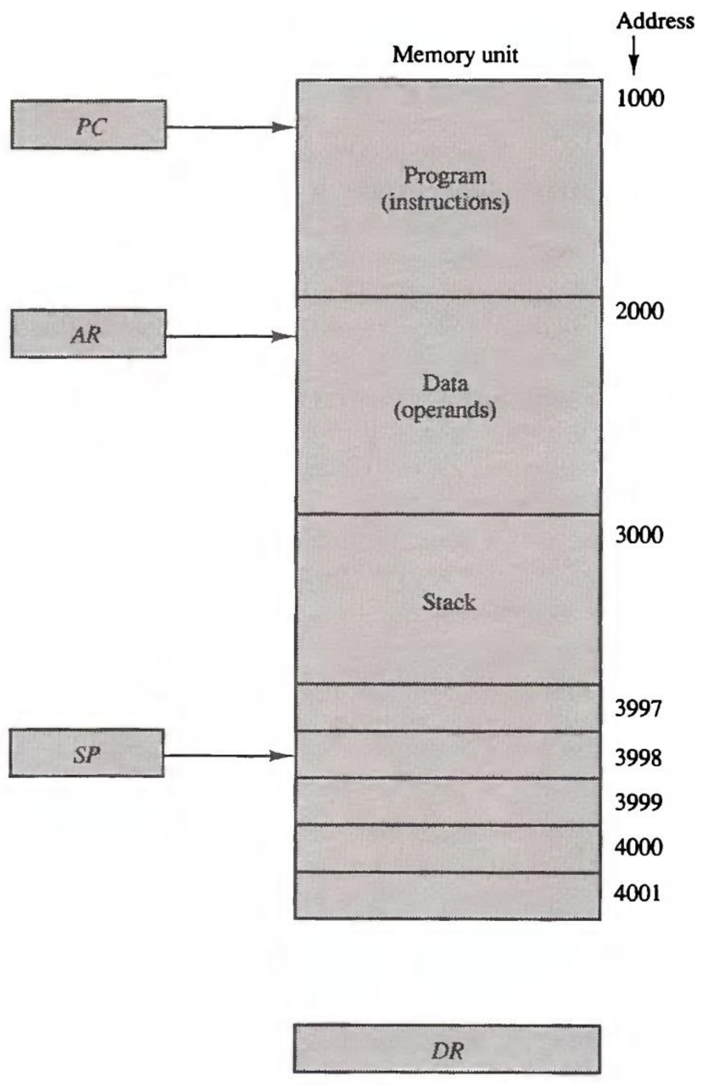


Figure 50 Computer memory with program, data, and stack segments.

As shown in Figure 4, the initial value of SP is 4001 and the stack grows with decreasing addresses. Thus the first item stored in the stack is at address 4000, the second item is stored at address 3999, and the last address that can be used for the stack is 3000. No provisions are available for stack limit checks.

Assume that the items in the stack communicate with a data register DR. A new item is inserted with the push operation as follows:

$$SP \leftarrow SP - 1$$

$$M[SP] \leftarrow DR$$

A new item is deleted with a pop operation as follows:

$$DR \leftarrow M[SP]$$

$$SP \leftarrow SP + 1$$

Most computers do not provide hardware to check for stack overflow (full stack) or underflow (empty stack). The stack limits can be checked by using two processor registers: one to hold the upper limit (3000 in this case), and the other to hold the lower limit (4001 in this case). After a push operation, SP is compared with the upper-limit register and after a pop operation, SP is compared with the lower-limit register.

The two microoperations needed for either the push or pop are (1) an access to memory through SP, and (2) updating SP. Which of the two microoperations is done first and whether SP is updated by incrementing or decrementing depends on the organization of the stack.

Reverse Polish Notation

A stack organization is very effective for evaluating arithmetic expressions.

Infix notation: each operator written between the operands.

A*B + C*D

It is necessary to scan back and forth along the expression to determine the next operation to be performed.

The Polish mathematician Lukasiewicz showed that arithmetic expressions can be represented in:

Prefix notation: this representation often referred to as **Polish notation**, places the operator before the operands.

The **postfix notation**, referred to as **reverse Polish notation (RPN)**, places the operator after the operands.

The following examples demonstrate the three representations:

A + B	Infix notation
+AB	Prefix or Polish notation
AB+	Postfix or reverse Polish notation

The reverse Polish notation is in a form suitable for stack manipulation. The expression

A*B + C*D

is written in reverse Polish notation as

AB*CD*+

and is evaluated as follows: Scan the expression from left to right. When an operator is reached, perform the operation with the two operands found on the left side of the operator. Remove the two operands and the operator and replace them by the number obtained from the result of the operation. Continue to scan the expression and repeat the procedure for every operator encountered until there are no more operators.

For the expression above we find the operator * after A and B. We perform the operation A * B and replace A, B, and * by the product to obtain

(A*B)CD* +

where (A * B) is a single quantity obtained from the product. The next operator is a * and its previous two operands are C and D, so we perform C*D and obtain an expression with two operands and one operator:

(A*B)(C*D) +

The next operator is + and the two operands to be added are the two products, so we add the two quantities to obtain the result.

The conversion from infix notation to reverse Polish notation must take into consideration the operational hierarchy adopted for infix notation. This hierarchy dictates that we first perform all arithmetic inside inner parentheses, then inside outer parentheses, and do multiplication and division operations before addition and subtraction operations. The expression

$$\underbrace{\underbrace{(A+B)}_{(1)} * \underbrace{[\underbrace{C * \underbrace{(D+E)}_{(1)}}_{(2)} + F]}_{(3)}}_{(4)}$$

can be converted to reverse Polish notation, without the use of parentheses, by taking into consideration the operation hierarchy. The converted expression is

AB + DE + C*F + *

Proceeding from left to right, we first add A and B, then add D and E. At this point we are left with

(A + B)(D + E)C*F + *

where (A + B) and (D + E) are each a single number obtained from the sum. The two operands for the next * are C and (D+E). These two numbers are multiplied and the product added to F. The final * causes the multiplication of the two terms.

Evaluation of Arithmetic Expressions

Reverse Polish notation, combined with a stack arrangement of registers, is the most efficient way known for evaluating arithmetic expressions. This procedure is employed in some electronic calculators and also in some computers. The stack is particularly useful for handling long, complex problems involving chain calculations. It is based on the fact that any arithmetic expression can be expressed in parentheses-free Polish notation.

The procedure consists of first converting the arithmetic expression into its equivalent reverse Polish notation. The operands are pushed into the stack in the order in which they appear. The initiation of an operation depends on whether we have a calculator or a computer. In a calculator, the operators are entered through the keyboard. In a computer, they must be initiated by instructions that contain an operation field (no address field is required). The following microoperations are executed with the stack when an operation is entered in a calculator or issued by the control in a computer: (1) the two topmost operands in the stack are used for the operation, and (2) the stack is popped and the result of the operation replaces the lower operand. By pushing the operands into the stack continuously and performing the operations as defined above, the expression is evaluated in the proper order and the final result remains on top of the stack.

The following numerical example may clarify this procedure. Consider the arithmetic expression

(3*4) + (5*6)

In reverse Polish notation, it is expressed as

34*56* +

Now the stack operations are shown in Figure 5.

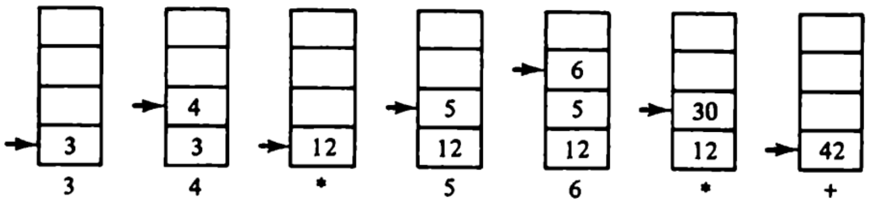


Figure 51 Stack operations to evaluate 3 • 4 + 5 •

Scientific calculators that employ an internal stack require that the user convert the arithmetic expressions into reverse Polish notation. Computers that use a stack-organized CPU provide a system program to perform the conversion for the user. Most compilers, irrespective of their CPU organization, convert all arithmetic expressions into Polish notation anyway because this is the most efficient method for translating arithmetic expressions into machine language instructions. So in essence, a stack-organized CPU may be more efficient in some applications than a CPU without a stack.

Instruction Formats

The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are:

1. An operation code field that specifies the operation to be performed.
2. An address field that designates a memory address or a processor register.
3. A mode field that specifies the way the operand or the effective address is determined.

Other special fields are sometimes employed under certain circumstances, as for example a field that gives the number of shifts in a shift-type instruction.

Operations specified by computer instructions are executed on some data stored in memory or processor registers.

- Operands residing in memory are specified by their memory address.
- Operands residing in processor registers are specified with a register address.

A **register address** is a binary number of k bits that defines one of 2^k registers in the CPU. Thus a CPU with 16 processor registers RQ through R15 will have a register address field of four bits. The binary number 0101, for example, will designate register R5.

Computers may have instructions of several different lengths containing varying number of addresses. The number of address fields in the instruction format of a computer depends on the internal organization of its registers. Most computers fall into one of three types of CPU organizations:

1. Single accumulator organization.
   - The instruction has one address field.
   - All operations are performed with an implied accumulator register.
   - e.g. ADD X it means $AC \leftarrow AC + M[X]$

2. General register organization.

- Two or three address fields in their instruction format. Each address field may specify a processor register or a memory word:
  - Three address register fields

    e.g. ADD R1, R2, R3 it means R1 ⟵ R2 + R3
  - Two address register fields

    e.g. ADD R1, R2 it means R1 ⟵ R1 + R2
  - One memory address and one register address fields

    e.g. ADD R1, X it means R1 ⟵ R1 + M[X]

3. Stack organization.
   - Computers would have PUSH and POP instructions which require an address field.

     e.g. PUSH X ,will push the word at address X to the top of the stack.

   - The stack pointer is updated automatically.
   - Operation-type instructions do not need an address field. This is because the operation is performed on the two items that are on top of the stack.

     e.g. ADD This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack. There is no need to specify operands with an address field since all operands are implied to be in the stack.

Some computers fall into one of the three types of organizations that have just been described and some computers combine features from more than one organizational structure.

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

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

using zero, one, two, or three address instructions. We will use the symbols ADD, SUB, MUL, and DIV for the four arithmetic operations; MOV for the transfer-type operation; and LOAD and STORE for transfers to and from memory and AC register. We will assume that the operands are in memory addresses A, B, C, and D, and the result must be stored in memory at address X.

❖ Three – Address Instructions

```
ADD     R1, A, B          R1 ⟵ M[A] + M[B]
ADD     R2, C, D          R2 ⟵ M[C] + M[D]
MUL     X, R1, R2         M[X] ⟵ R1 * R2
```

The advantage of the three-address format is that it results in short programs when evaluating arithmetic expressions. The disadvantage is that the binary-coded instructions require too many bits to specify three addresses.

❖ Two – Address Instructions

```
MOV     R1, A             R1 ⟵ M[A]
ADD     R1, B             R1 ⟵ R1 + M[B]
MOV     R2, C             R2 ⟵ M[C]
ADD     R2, D             R2 ⟵ R2 + M[D]
MUL     R1, R2       R1 ⟵ R1 * R2
```

MOV	X, R1	M[X] ← R1

❖ One – Address Instructions

LOAD	A	AC ← M[A]
ADD	B	AC ← AC + M[B]
STORE	T	M[T] ← AC
LOAD	C	AC ← M[C]
ADD	D	AC ← AC + M[D]
MUL	T	AC ← AC * M[T]
STORE	X	M[X] ← AC

❖ Zero – Address Instructions

PUSH	A	TOS ← A
PUSH	B	TOS ← B
ADD		TOS ← (A + B)
PUSH	C	TOS ← C
PUSH	D	TOS ← D
ADD		TOS ← (C + D)
MUL		TOS ← (A + B) * (C + D)
POP	X	M[X] ← TOS

RISC Instructions

The instruction set of a typical RISC processor is restricted to the use of load and store instructions when communicating between memory and CPU. All other instructions are executed within the registers of the CPU without referring to memory. A program for a RISC-type CPU consists of LOAD and STORE instructions that have one memory and one register address, and computational-type instructions that have three addresses with all three specifying processor registers. The following is a program to evaluate

X = (A + B)*(C + D).

LOAD	R1, A	R1 ← M[A]
LOAD	R2, B	R2 ← M[B]
LOAD	R3, C	R3 ← M[C]
LOAD	R4, D	R4 ← M[D]
ADD	R1, R1, R2	R1 ← R1 + R2
ADD	R3, R3, R4	R3 ← R3 + R4
MUL	R1, R1, R3	R1 ← R1 * R3
STORE	X, R1	M[X]← R1

Addressing Modes

Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:

1. To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
2. To reduce the number of bits in the addressing field of the instruction.

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

An example of an instruction format with a distinct addressing mode field is shown in Figure 6. The operation code specifies the operation to be performed. The mode field is used to locate the operands needed for the operation. There may or may not be an address field in the instruction. If there is an address field, it may designate a memory address or a processor register. Moreover, as discussed in the preceding section, the instruction may have more than one address field, and each address field may be associated with its own particular addressing mode.

Although most addressing modes modify the address field of the instruction, there are two modes that need no address field at all. These are the implied and immediate modes. Table 4 shows the description of the modes.

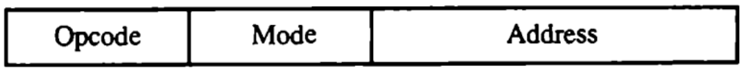


Figure 52 Instruction format with mode field.

Table 22 description of the addressing modes

Mode	Description
Implied	The operands are specified implicitly in the definition of the instruction. All register reference instructions that use an accumulator Zero-address instructions in a stack-organized computer For example, the instruction "complement accumulator"
Immediate	The operand is specified in the instruction itself. In other words, an immediate-mode instruction has an operand field rather than an address field. Useful for initializing registers to a constant value.
Register	The operands are in registers that reside within the CPU.
Register Indirect	The instruction specifies a register in the CPU whose contents give the address of the operand in memory. In other words, the selected register contains the address of the operand rather than the operand itself. The advantage of a register indirect mode instruction is that the address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly.
Autoincrement or Autodecrement	This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory.
Direct Address	The effective address is equal to the address part of the instruction.
Indirect Address	The address field of the instruction gives the address where the effective address is stored in memory.
Relative Address	The content of the program counter is added to the address part of the instruction in order to obtain the effective address. It is often used with branch-type instructions when the branch address is in the area surrounding the instruction word itself. It results in a shorter address field in the instruction format since the relative address can be specified with a smaller number of bits compared to the number of bits required to designate the entire memory address.
Indexed Addressing	The content of an index register is added to the address part of the instruction to obtain the effective address. For more details see M. Morris Mano p 264.
Base Register Addressing	The content of a base register is added to the address part of the instruction to obtain the effective address.

Numerical Example

To show the differences between the various modes, we will show the effect of the addressing modes on the instruction defined in Figure 7. The two-word instruction at address 200 and 201 is a "load to AC" instruction with an address field equal to 500. The first word of the instruction specifies the operation code and mode, and the second

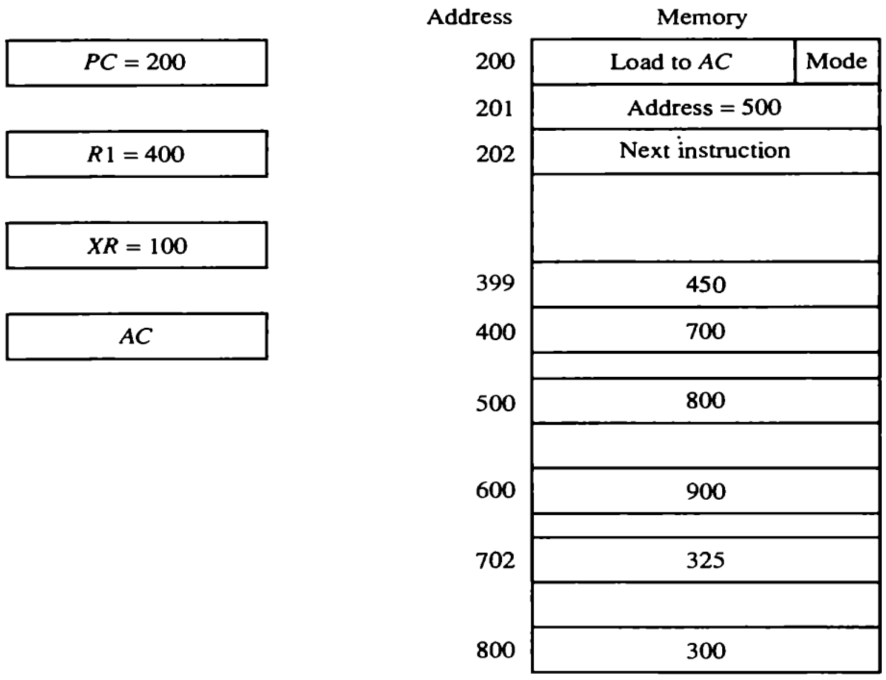


Figure 53 Numerical example for addressing modes.

word specifies the address part. Table 4 lists the values of the effective address and the operand loaded into AC for the nine addressing modes.

Table 23 Tabular List of Numerical Example

Addressing Mode	Effective Address	Content of AC
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

H.W. An instruction is stored at location 500H with its address field at locations 501H and 502H. The address field has the value 700H. A processor register R1 contains the number 400H. The index register XR contains the number 600H. The base register BR contains the number 800H. The location 701H contains the value 30H. The location 700H has the value 20H. Evaluate the effective address if the addressing mode of the instruction is (a) direct; (b) immediate; (c) relative; (d) register indirect; (e) indirect; (f) relative; (g) index register; (h) base register; (i) Auto increment; (j) Auto decrement.

Data Transfer and Manipulation

The instruction set of different computers differ from each other mostly in the way the operands are determined from the address and mode fields.

Most computer instructions can be classified into three categories:

1. **Data transfer instructions** cause transfer of data from one location to another without changing the binary information content.
2. **Data manipulation instructions** are those that perform arithmetic, logic, and shift operations.
3. **Program control instructions** provide decision-making capabilities and change the path taken by the program when executed in the computer.

The instruction set of a particular computer determines the register transfer operations and control decisions that are available to the user.

Data Transfer Instructions

The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves. Table 24 gives a list of eight data transfer instructions used in many computers.

- The **load** instruction has been used mostly to designate a transfer from memory to a processor register, usually an accumulator.
- The **store** instruction designates a transfer from a processor register into memory.
- The **move** instruction has been used in computers with multiple CPU registers to designate a transfer from one register to another. It has also been used for data transfers between CPU registers and memory or between two memory words.
- The **exchange** instruction swaps information between two registers or a register and a memory word.
- The **input** and **output** instructions transfer data among processor registers and input or output terminals.
- The **push** and **pop** instructions transfer data between processor registers and a memory stack.

Table 24 Typical Data Transfer

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

It must be realized that the instructions listed in Table 24, as well as in subsequent tables in this section, are often associated with a variety of addressing modes. Some assembly language conventions modify the mnemonic symbol to differentiate between the different addressing modes. For example, the mnemonic for load immediate becomes LDI.

Other assembly language conventions use a special character to designate the addressing mode. For example, the immediate mode is recognized from a pound sign # placed before the operand. In any case, the important thing is to realize that each instruction can occur with a variety of addressing modes. As an example, consider the load to accumulator instruction when used with eight different addressing modes.

Table 25 shows the recommended assembly language convention and the actual transfer accomplished in each case. ADR stands for an address, NBR is a number or operand, X is an index register, R1 is a processor register, and

Table 25 Eight Addressing Modes for the Load

Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	$AC \leftarrow M[ADR]$
Indirect address	LD @ADR	$AC \leftarrow M[M[ADR]]$
Relative address	LD $ADR	$AC \leftarrow M[PC + ADR]$
Immediate operand	LD #NBR	$AC \leftarrow NBR$
Index addressing	LD ADR(X)	$AC \leftarrow M[ADR + XR]$
Register	LD R1	$AC \leftarrow R1$
Register indirect	LD (R1)	$AC \leftarrow M[R1]$
Autoincrement	LD (R1)+	$AC \leftarrow M[R1],\ R1 \leftarrow R1 + 1$

AC is the accumulator register.

Data Manipulation Instructions

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

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

- Each instruction when executed in the computer must go through the fetch phase to read its binary code value from memory.
- The operands must also be brought into processor registers according to the rules of the instruction addressing mode.
- The last step is to execute the instruction in the processor. This last step is implemented by means of microoperations or through an ALU and shifter. Some of the arithmetic instructions need special circuits for their implementation.

Arithmetic Instructions

- The four basic arithmetic operations are addition, subtraction, multiplication, and division.

- Most computers provide instructions for all four operations.
- Some small computers have only addition and possibly subtraction instructions.
- The multiplication and division must then be generated by means of software subroutines.
- The four basic arithmetic operations are sufficient for formulating solutions to scientific problems when expressed in terms of numerical analysis methods.
- A list of typical arithmetic instructions is given in Table 26.

Table 26 Typical Arithmetic Instructions

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate (2's complement)	NEG

The add, subtract, multiply, and divide instructions may be available for different types of data. The data type assumed to be in processor registers during the execution of these arithmetic operations is included in the definition of the operation code. An arithmetic instruction may specify fixed-point or floating-point data, binary or decimal data, single-precision or double-precision data.

It is not uncommon to find computers with three or more add instructions: one for binary integers, one for floating-point operands, and one for decimal operands. The mnemonics for three add instructions that specify different data types are shown below.

ADDI	Add two binary integer numbers
ADDF	Add two floating-point numbers
ADDD	Add two decimal numbers in BCD

- The number of bits in any register is of finite length and therefore the results of arithmetic operations are of finite precision.
- Some computers provide hardware double-precision operations where the length of each operand is taken to be the length of two memory words.
- Most small computers provide special instructions to facilitate double-precision arithmetic.
- A special carry flip-flop is used to store the carry from an operation.
- The instruction "**add with carry**" performs the addition on two operands plus the value of the carry from the previous computation. Similarly, the "**subtract with borrow**" instruction subtracts two words and a borrow which may have resulted from a previous subtract operation.
- The negate instruction forms the 2's complement of a number, effectively reversing the sign of an integer when represented in the signed-2's complement form.

H.W. Assuming an 8-bit computer, show the multiple precision subtraction of the two 32-bit unsigned numbers listed below using the subtract with borrow instruction. Each byte is expressed as a two-digit hexadecimal number.

(48 39 B4 78) – (C4 67 DF AC)

Logical and Bit Manipulation Instructions

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

Some typical logical and bit manipulation instructions are listed in Table 27.

Table 27 Typical Logical and Bit Manipulation

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

There are three bit manipulation operations possible: a selected bit can be cleared to 0, or can be set to 1, or can be complemented.

- The **AND** instruction is used to **clear** a bit or a selected group of bits of an operand. For any Boolean variable x, the relationships x . 0 = 0 and x . 1 = x dictate that a binary variable ANDed with a 0 produces a 0; but the variable does not change in value when ANDed with a 1. Therefore, the AND instruction can be used to clear bits of an operand selectively by ANDing the operand with another operand that has 0's in the bit positions that must be cleared. The AND instruction is also called a mask because it masks or inserts 0's in a selected portion of an operand.
- The **OR** instruction is used to **set** a bit or a selected group of bits of an operand. For any Boolean variable x, the relationships x + 1 = 1 and x + 0 = x dictate that a binary variable ORed with a 1 produces a 1; but the variable does not change when ORed with a 0. Therefore, the OR instruction can be used to selectively set bits of an operand by ORing it with another operand with 1's in the bit positions that must be set to 1.
- Similarly, the **XOR** instruction is used to selectively **complement** bits of an operand. This is because of the Boolean relationships $x \oplus 1 = x'$ and $x \oplus 0 = x$. Thus a binary variable is complemented when XORed with a 1 but does not change in value when XORed with a 0.

A few other bit manipulation instructions are included in Table 27. Individual bits such as a carry can be cleared, set, or complemented with appropriate instructions. Another example is a flip-flop that controls the interrupt facility and is either enabled or disabled by means of bit manipulation instructions.

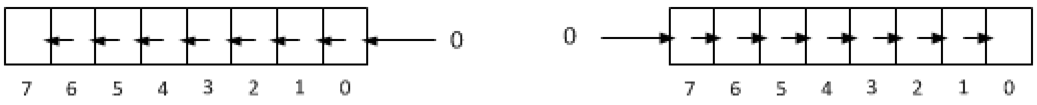
H.W. Given the 16-bit value 1001011011000101, the carry flag is low and interrupt flag is high. Apply all instructions in Table 27 with the operand 1010101010101010, show the results.

Shift Instructions

Instructions to shift the content of an operand are quite useful and are often provided in several variations. Shifts are operations in which the bits of a word are moved to the left or right. The bit shifted in at the end of the word determines the type of shift used. Shift instructions may specify logical shifts, arithmetic shifts, or rotate-type operations. In either case the shift may be to the right or to the left.

Table 10 lists four types of shift instructions:

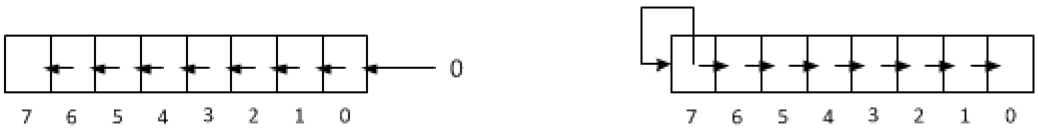
- The **logical shift** inserts 0 to the end bit position. The end position is the leftmost bit for shift right and the rightmost bit position for the shift left.



(a) Logical left shift (b) Logical right shift

Figure 54

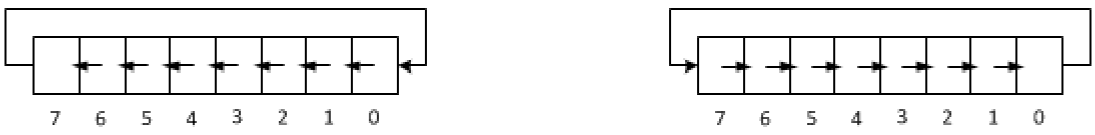
- **Arithmetic shifts** usually conform with the rules for signed-2's complement numbers. The arithmetic shift-right instruction must preserve the sign bit in the leftmost position. The sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unchanged. This is a shift-right operation with the end bit remaining the same. The arithmetic shift-left instruction inserts 0 to the end position and is identical to the logical shift-left instruction. For this reason many computers do not provide a distinct arithmetic shift-left instruction when the logical shift-left instruction is already available.



(a) Arithmetic left shift (b) Arithmetic right shift

Figure 55 Arithmetic

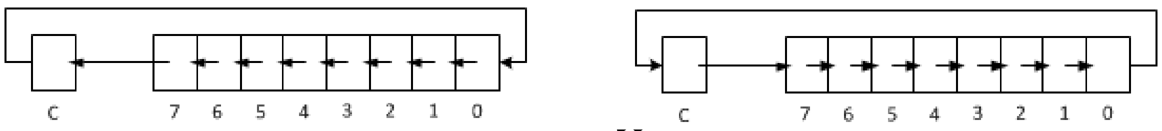
- The **rotate** instructions produce a circular shift. Bits shifted out at one end of the word are not lost as in a logical shift but are circulated back into the other end.
- The **rotate through carry** instruction treats a carry bit as an extension of the register whose word is being



(a) Rotate left (b) Rotate right

Figure 56

rotated. Thus a rotate-left through carry instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit position into the carry, and at the same time, shifts the entire register



(a) Rotate left through carry (b) Rotate right through carry

Figure 57 Rotate through

to the left.

Some computers have a multiple-held format for the shift instructions. One field contains the operation code and the others specify the type of shift and the number of times that an operand is to be shifted. A possible instruction code format of a shift instruction may include five fields as follows:

OP REG TYPE RL COUNT

OP:	Operation code field.
REG:	Register address that specifies the location of the operand.
TYPE:	A 2-bit field specifying the four different types of shifts.
RL:	A 1-bit field specifying a shift right or left.
COUNT:	A k-bit field specifying up to 2^k - 1 shifts.

With such a format, it is possible to specify the type of shift, the direction, and the number of shifts, all in one instruction.

Table 28 Typical Shift Instructions

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

Program Control

Instructions are always stored in successive memory locations. When processed in the CPU, the instructions are fetched from consecutive memory locations and executed. Each time an instruction is fetched from memory, the **program counter is incremented** so that it contains the address of the **next instruction** in sequence. After the execution of a data transfer or data manipulation instruction, control returns to the fetch cycle with the program counter containing the address of the instruction next in sequence. On the other hand, program control instructions specify conditions for altering the content of the program counter, while data transfer and manipulation instructions specify conditions for data-processing operations. The change in value of the program counter as a result of the execution of a program control instruction causes a break in the sequence of instruction execution. This is an important feature in digital computers, as it provides control over the flow of program execution and a capability for branching to different program segments.

Some typical program control instructions are listed in Table 29. The branch and jump instructions are used interchangeably to mean the same thing, but sometimes they are used to denote different addressing modes. The branch is usually a one-address instruction. It is written in assembly language as **BR ADR**, where ADR is a symbolic

Table 29 Typical Program Control

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TST

name for an address. When executed, the branch instruction causes a transfer of the value of ADR into the program counter. Since the program counter contains the address of the instruction to be executed, the next instruction will come from location ADR.

Branch and **jump** instructions may be **conditional** or **unconditional**. An unconditional branch instruction causes a branch to the specified address without any conditions. The conditional branch instruction specifies a condition such as branch if positive or branch if zero. If the condition is met, the program counter is loaded with the branch address and the next instruction is taken from this address. If the condition is not met, the program counter is not changed and the next instruction is taken from the next location in sequence.

The **skip** instruction does not need an address field and is therefore a **zero-address instruction**. A conditional skip instruction will skip the next instruction if the condition is met. This is accomplished by incrementing the program counter during the execute phase in addition to its being incremented during the fetch phase. If the condition is not met, control proceeds with the next instruction in sequence where the programmer inserts an unconditional branch instruction. Thus a skip-branch pair of instructions causes a branch if the condition is not met, while a single conditional branch instruction causes a branch if the condition is met.

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

The **compare** and test instructions do not change the program sequence directly. They are listed in Table 29 because of their application in setting conditions for subsequent conditional branch instructions. The compare instruction performs a **subtraction** between two operands, but the result of the operation is **not retained**. However, certain status bit conditions are set as a result of the operation.

Similarly, the **test** instruction performs the **logical AND** of two operands and updates certain status bits **without retaining the result or changing the operands**.

The status bits of interest are the carry bit, the sign bit, a zero indication, and an overflow condition. The generation of these status bits will be discussed first and then we will show how they are used in conditional branch instructions.

Status Bit Conditions

It is sometimes convenient to supplement the ALU circuit in the CPU with a status register where status bit conditions can be stored for further analysis. Status bits are also called condition-code bits or flag bits. Figure 58 shows the block diagram of an 8-bit ALU with a 4-bit status register. The four status bits are symbolized by C, S, Z, and V. The bits are set or cleared as a result of an operation performed in the ALU.

1. Bit C (carry) is set to 1 if the end carry C_8 is 1. It is cleared to 0 if the carry is 0.
2. Bit S (sign) is set to 1 if the highest-order bit F_7 is 1. It is set to 0 if the bit is 0.
3. Bit Z (zero) is set to 1 if the output of the ALU contains all 0's. It is cleared to 0 otherwise. In other words, Z = 1 if the output is zero and Z = 0 if the output is not zero.
4. Bit V (overflow) is set to 1 if the exclusive-OR of the last two carries is equal to 1, and cleared to 0 otherwise. This is the condition for an overflow when negative numbers are in 2's complement. For the 8-bit ALU, V = 1 if the output is greater than +127 or less than -128.

The status bits can be checked after an ALU operation to determine certain relationships that exist between the values of A and B.

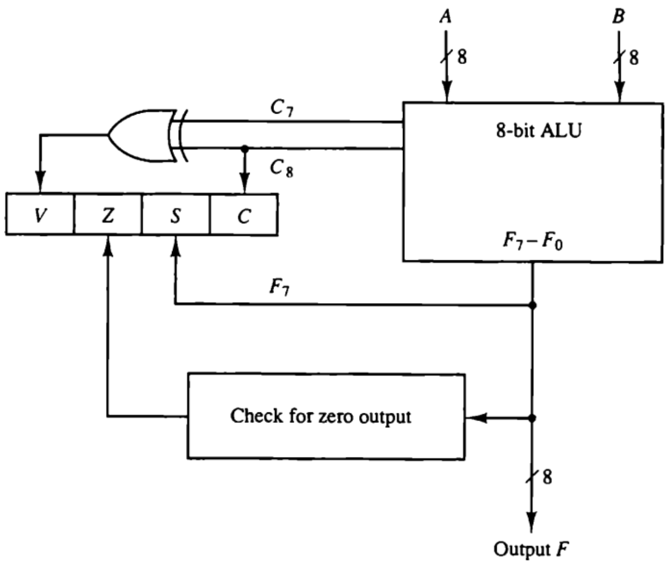


Figure 58 Status register bits.

If bit V is set after the addition of two **signed** numbers, it indicates an overflow condition. If Z is set after an exclusive-OR operation, it indicates that A = B. This is so because $x \oplus x = 0$, and the exclusive-OR of two equal operands gives an all-0's result which sets the Z bit. A single bit in A can be checked to determine if it is 0 or 1 by masking all bits except the bit in question and then checking the Z status bit. For example, let A = 101x1100, where x is the bit to be checked. The AND operation of A with B = 00010000 produces a result 000x0000. If x = 0, the Z status bit is set, but if x = 1, the Z bit is cleared since the result is not zero. The AND operation can be generated with the TEST instruction listed in Table 29 if the original content of A must be preserved.

H.W. What is the logic circuit of the block labeled "check for zero output" in Figure 58?

H.W. Show that these status cases are impossible.

1. Z=1 and S=1.
2. Z=1, C=0 and V=1.

H.W. Give a tow eight-bit numbers that the subtraction between them gives the following status bits:

1. Z=0, S=0, C=1 and V=0.
2. Z=1, S=0, C=1 and V=0.

Conditional Branch Instructions

Table 12 gives a list of the most common branch instructions. Each mnemonic is constructed with the letter B (for branch) and an abbreviation of the condition name. When the opposite condition state is used, the letter N (for no) is inserted to define the 0 state.

The compare instruction performs a subtraction of two operands, say A - B. The result of the operation is not transferred into a destination register, but the status bits are affected. The status register provides information about the relative magnitude of A and B. Some computers provide conditional branch instructions that can be applied right after the execution of a compare instruction. The specific conditions to be tested depend on whether the two numbers A and B are considered to be unsigned or signed numbers. Table 30 gives a list of such conditional branch instructions. Note that we use the words higher and lower to denote the relations between unsigned numbers, and greater and less than for signed numbers. The relative magnitude shown under the tested condition column in the table seems to be the same for unsigned and signed numbers. However, this is not the case since each must be considered separately as explained in the following numerical example.

Consider an 8-bit ALU as shown in Figure 58. The largest unsigned number that can be accommodated in 8 bits is 255. The range of signed numbers is between +127 and -128. The subtraction of two numbers is the same whether they are unsigned or in signed-2's complement representation. Let A = 11110000 and B = 00010100. To perform A - B, the ALU takes the 2's complement of B and adds it to A.

A: 11110000

Table 30 Conditional Branch

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	$Z = 1$
BNZ	Branch if not zero	$Z = 0$
BC	Branch if carry	$C = 1$
BNC	Branch if no carry	$C = 0$
BP	Branch if plus	$S = 0$
BM	Branch if minus	$S = 1$
BV	Branch if overflow	$V = 1$
BNV	Branch if no overflow	$V = 0$
Unsigned compare conditions ($A - B$)		
BHI	Branch if higher	$A > B$
BHE	Branch if higher or equal	$A \geq B$
BLO	Branch if lower	$A < B$
BLOE	Branch if lower or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$
Signed compare conditions ($A - B$)		
BGT	Branch if greater than	$A > B$
BGE	Branch if greater or equal	$A \geq B$
BLT	Branch if less than	$A < B$
BLE	Branch if less or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$

$\overline{B} + 1$: $\underline{11101100 \ +}$

$A - B$: 11011100 C=1 S=1 V=0 Z=0

The compare instruction updates the status bits as shown. C = 1 because there is a carry out of the last stage. S = 1 because the leftmost bit is 1. V = 0 because the last two carries are both equal to 1, and Z = 0 because the result is not equal to 0.

If we assume unsigned numbers, the decimal equivalent of A is 240 and that of B is 20. The subtraction in decimal is 240 - 20 = 220. The binary result 11011100 is indeed the equivalent of decimal 220. Since 240 > 20, we have that A > B and A ≠ B. These two relations can also be derived from the fact that status bit C is equal to 1 and bit Z is equal to 0. The instructions that will cause a branch after this comparison are BHI (branch if higher), BHE (branch if higher or equal), and BNE (branch if not equal).

If we assume signed numbers, the decimal equivalent of A is —16. This is because the sign of A is negative and 11110000 is the 2's complement of 00010000, which is the decimal equivalent of +16. The decimal equivalent of B is +20. The subtraction in decimal is (-16) - (+20) = -36. The binary result 11011100 (the 2's complement of 00100100) is indeed the equivalent of decimal -36. Since (-16) < (+20) we have that A < B and A ± B. These two relations can also be derived from the fact that status bits S = 1 (negative), V = 0 (no overflow), and Z = 0 (not zero). The instructions that will cause a branch after this comparison are BLT (branch if less than), BLE (branch if less or equal), and BNE (branch if not equal).

It should be noted that the instruction BNE and BNZ (branch if not zero) are identical. Similarly, the two instructions BE (branch if equal) and BZ (branch if zero) are also identical. Each is repeated three times in Table 30 for the purpose of clarity and completeness.

It should be obvious from the example that the relative magnitude of two unsigned numbers can be determined (after a compare instruction) from the values of status bits C and Z. The relative magnitude of two signed numbers can be determined from the values of S, V, and Z.

Some computers consider the C bit to be a borrow bit after a subtraction operation A — B. A borrow does not occur if A ≥ B, but a bit must be borrowed from the next most significant position if A < B. The condition for a borrow is the complement of the carry obtained when the subtraction is done by taking the 2's complement of B. For this reason, a processor that considers the C bit to be a borrow after a subtraction will complement the C bit after adding the 2's complement of the subtrahend and denote this bit a borrow.

H.W. Write a Boolean algebraic form of the tested conditions for unsigned and signed compare conditions instructions in Table 30.

Subroutine Call and Return

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

- The instruction that transfers program control to a subroutine is known by different names. The most common names used are call subroutine, jump to subroutine, branch to subroutine, or branch and save address.
- A call subroutine instruction consists of an operation code together with an address that specifies the beginning of the subroutine.
- The instruction is executed by performing two operations: (1) the address of the next instruction available in the program counter (the return address) is stored in a temporary location so the subroutine knows where to return, and (2) control is transferred to the beginning of the subroutine.
- The last instruction of every subroutine, commonly called return from subroutine, transfers the return address from the temporary location into the program counter. This results in a transfer of program control to the instruction whose address was originally stored in the temporary location.

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

$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PC$	Push content of PC onto the stack
$PC \leftarrow effctive\ address$	Transfer control to the subroutine

If another subroutine is called by the current subroutine, the new return address is pushed into the stack, and so on. The instruction that returns from the last subroutine is implemented by the microoperations:

$PC \leftarrow M[SP]$	Pop stack and transfer to PC
$SP \leftarrow SP+1$	Increment stack pointer

By using a subroutine stack, all return addresses are automatically stored by the hardware in one unit. The programmer does not have to be concerned or remember where the return address was stored.

A recursive subroutine is a subroutine that calls itself. If only one register or memory location is used to store the return address, and the recursive subroutine calls itself, it destroys the previous return address. This is undesirable because vital information is destroyed. This problem can be solved if different storage locations are employed for each use of the subroutine while another lighter-level use is still active. When a stack is used, each return address can be pushed into the stack without destroying any previous values. This solves the problem of recursive subroutines because the next subroutine to exit is always the last subroutine that was called.

Program Interrupt

The concept of program interrupt is used to handle a variety of problems that arise out of normal program sequence. Program interrupt refers to the transfer of program control from a currently running program to another service program as a result of an external or internal generated request. Control returns to the original program after the service program is executed.

The interrupt procedure is, in principle, quite similar to a subroutine call except for three variations:

1) The interrupt is usually initiated by an internal or external signal rather than from the execution of an instruction (except for software interrupt as explained later)
2) The address of the interrupt service program is determined by the hardware rather than from the address field of an instruction
3) An interrupt procedure usually stores all the information necessary to define the state of the CPU rather than storing only the program counter.

After a program has been interrupted and the service routine been executed, the CPU must return to exactly the same state that it was when the interrupt occurred. Only if this happens will the interrupted program be able to resume exactly as if nothing had happened. The state of the CPU at the end of the execute cycle (when the interrupt is recognized) is determined from:

1. The content of the program counter
2. The content of all processor registers
3. The content of certain status conditions

- Program status word (PSW): The collection of all status bit conditions in the CPU.
- The PSW is stored in a separate hardware register and contains the status information that characterizes the state of the CPU.

- Typically, it includes the status bits from the last ALU operation and it specifies the interrupts that are allowed to occur and whether the CPU is operating in a supervisor or user mode.
- Many computers have a resident operating system that controls and supervises all other programs in the computer. When the CPU is executing a program that is part of the operating system, it is said to be in the supervisor or system mode. Certain instructions are privileged and can be executed in this mode only. The CPU is normally in the user mode when executing user programs. The mode that the CPU is operating at any given time is determined from special status bits in the PSW.

The CPU must possess some form of hardware procedure for selecting a branch address for servicing the interrupt.

The CPU does not respond to an interrupt until the end of an instruction execution. (Why?)

Just before going to the next fetch phase, control checks for any interrupt signals. If an interrupt is pending, control goes to hardware interrupt cycle. During this cycle, the contents of PC and PSW are pushed onto the stack. The branch address for the particular interrupt is then transferred to PC and a new PSW is loaded into the status register. The service program can now be executed starting from the branch address and having a CPU mode as specified in the new PSW.

The last instruction in the service program is a return from interrupt instruction. When this instruction is executed, the stack is popped to retrieve the old PSW and the return address. The PSW is transferred to the status register and the return address to the program counter. Thus the CPU state is restored and the original program can continue executing.

Types of Interrupts

There are three major types of interrupts that cause a break in the normal execution of a program. They can be classified as:

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

External interrupts	Internal interrupts	Software interrupts
Come from input-output (I/O) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source.	Arise from illegal or erroneous use of an instruction or data. Internal interrupts are also called traps.	Initiated by executing an instruction. While External and internal interrupts are Initiated from signals that occur in the hardware of the CPU.
Examples that cause external interrupts are I/O device requesting transfer of data, I/O device finished transfer of data, elapsed time of an event, or power failure.	Examples of interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation.	It is a special call instruction that behaves like an interrupt rather than a subroutine call.
Initiated by an external event.	Initiated by some exceptional condition caused by the program itself	It can be used by the programmer to initiate an interrupt procedure at any desired point in the program.
They are asynchronous with the program	They are synchronous with the program	
External interrupts depend on external conditions that are independent of the program being	If the program is rerun, the internal interrupts will occur in the same place each time.	

executed at the time.		

For more information about software interrupt, see M. Morris Mano p282.

Reduced Instruction Set Computer (RISC)

- A computer with a large number of instructions is classified as **a complex instruction set computer**, abbreviated **CISC**.
- Computers use fewer instructions with simple constructs so they can be executed much faster within the CPU without having to use memory as often. This type of computer is classified as a **reduced instruction set computer** or **RISC**.

CISC Characteristics

The essential goal of a CISC architecture is to attempt to provide a single machine instruction for each statement that is written in a high-level language. Examples of CISC architectures are the Digital Equipment Corporation VAX computer and the IBM 370 computer.

Another characteristic of CISC architecture is the incorporation of variable-length instruction formats.

The major characteristics of CISC architecture are:

1. A large number of instructions—typically from 100 to 250 instructions
2. Some instructions that perform specialized tasks and are used infrequently
3. A large variety of addressing modes—typically from 5 to 20 different modes
4. Variable-length instruction formats
5. Instructions that manipulate operands in memory

RISC Characteristics

The concept of RISC architecture involves an attempt to reduce execution time by simplifying the instruction set of the computer. The major characteristics of a RISC processor are:

1. Relatively few instructions
2. Relatively few addressing modes
3. Memory access limited to load and store instructions
4. All operations done within the registers of the CPU
5. Fixed-length, easily decoded instruction format
6. Single-cycle instruction execution
7. Hardwired rather than microprogrammed control

- The small set of instructions of a typical RISC processor consists mostly of register-to-register operations, with only simple load and store operations for memory access.
- An important aspect of RISC instruction format is that it is easy to decode.
- A characteristic of RISC processors is their ability to execute one instruction per clock cycle. This is done by overlapping the fetch, decode, and execute phases of two or three instructions by using a procedure referred to as pipelining.
- A load or store instruction may require two clock cycles because access to memory takes more time than register operations.

Other characteristics attributed to RISC architecture are:

1. A relatively large number of registers in the processor unit
2. Use of overlapped register windows to speed-up procedure call and return
3. Efficient instruction pipeline
4. Compiler support for efficient translation of high-level language programs into machine language programs

Overlapped Register Windows

Procedure call and return occurs quite often in high-level programming languages. When translated into machine language, a procedure call produces a sequence of instructions that save register values, pass parameters needed for the procedure, and then calls a subroutine to execute the body of the procedure. After a procedure return, the program restores the old register values, passes results to the calling program, and returns from the subroutine. Saving and restoring registers and passing of parameters and results involve time-consuming operations. Some computers provide multiple-register banks, and each procedure is allocated its own bank of registers. This eliminates the need for saving and restoring register values. Some computers use the memory stack to store the parameters that are needed by the procedure, but this requires a memory access every time the stack is accessed.

A characteristic of some RISC processors is their use of overlapped register windows to provide the passing of parameters and avoid the need for saving and restoring register values.

The concept of overlapped register windows is illustrated in Figure 59.

- The system has a total of 74 registers.
- Registers R0 through R9 are global registers that hold parameters shared by all procedures.
- The other 64 registers are divided into four windows to accommodate procedures A,B,C, and D.
- Each register window consists of 10 local registers and two sets of six registers common to adjacent windows. Local registers are used for local variables.
- Common registers are used for exchange of parameters and results between adjacent procedures.
- The common overlapped registers permit parameters to be passed without the actual movement of data.
- Only one register window is activated at any given time with a pointer indicating the active window.
- Each procedure call activates a new register window by incrementing the pointer.
- The high registers of the calling procedure overlap the low registers of the called procedure, and therefore the parameters automatically transfer from calling to called procedure.

Example:

- Procedure A calls procedure B.
  Registers R26 through R31 are common to both procedures, and therefore procedure A stores the parameters for procedure B in these registers.
- Procedure B uses local registers R32 through R41 for local variable storage.
- If procedure B calls procedure C, it will pass the parameters through registers R42 through R47.
- When procedure B is ready to return at the end of its computation, the program stores results of the computation in registers R26 through R31 and transfers back to the register window of procedure A.

Note that registers R10 through R15 are common to procedures A and D because the four windows have a circular organization with A being adjacent to D.

As mentioned previously, the 10 global registers R0 through R9 are available to all procedures. Each procedure in Figure 59 has available a total of 32 registers while it is active. This includes 10 global registers, 10 local registers, six low overlapping registers, and six high overlapping registers.

Other fixed-size register window schemes are possible, and each may differ in the size of the register window and the size of the total register file. In general, the organization of register windows will have the following relationships:

Number of global registers = G
Number of local registers in each window = L
Number of registers common to two windows = C
Number of windows = W

The number of registers available for each window is calculated as follows:

$$\text{window size} = L + 2C + G$$

The total number of registers needed in the processor is

$$\text{register file} = (L + C)W + G$$

In the example of Figure 59 we have G = 10, L = 10, C = 6, and W = 4. The window size is 10 + 12 + 10 = 32 registers, and the register file consists of (10 + 6) x 4 + 10 = 74 registers.

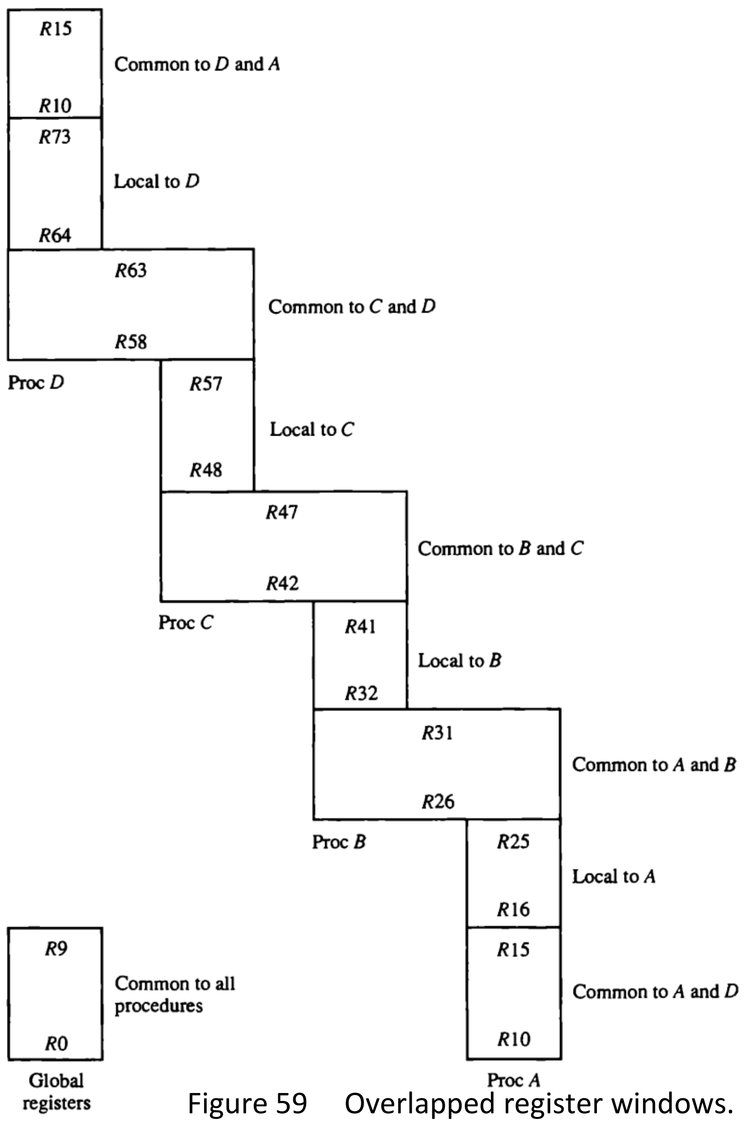


Figure 59 Overlapped register windows.

Berkeley RISC I

One of the first projects intended to show the advantages of RISC architecture was conducted at the University of California, Berkeley. The Berkeley RISC I is a 32-bit integrated circuit CPU. It supports 32-bit addresses and either 8-, 16-, or 32-bit data. It has a 32-bit instruction format and a total of 31 instructions. There are three basic addressing modes: register addressing, immediate operand, and relative to PC addressing for branch instructions. It has a register file of 138 registers arranged into 10 global registers and 8 windows of 32 registers in each. The 32 registers in each window have an organization similar to the one shown in Figure 59. Since only one set of 32 registers in a window is accessed at any given time, the instruction format can specify a processor register with a register held of five bits.

Figure 8-10 shows the 32-bit instruction formats used for register-to-register instructions and memory access

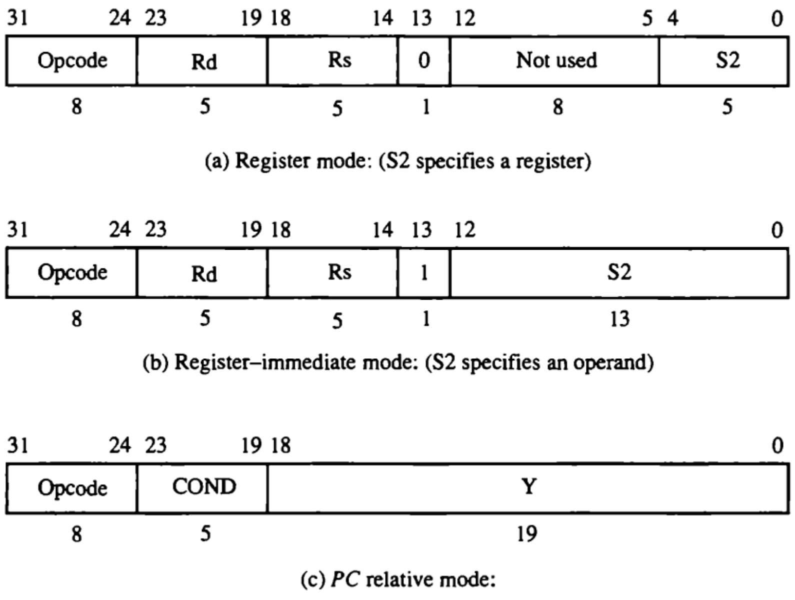


Figure 60 Berkeley RISC I instruction formats.

instructions.

- Seven of the bits in the operation code specify an operation, and the eighth bit indicates whether to update the status bits after an ALU operation.
- For register-to-register instructions:
  - 5-bit Rd field selects one of the 32 registers as a destination for the result of the operation.
  - The operation is performed with the data specified in fields Rs and S2. Rs is one of the source registers.
  - If bit 13 of the instruction is 0, the low-order 5 bits of S2 specify another source register. If bit 13 of the instruction is 1, S2 specifies a sign-extended 13-bit constant.
  - Thus the instruction has a three-address format, but the second source may be either a register or an immediate operand.
- Memory access instructions use Rs to specify a 32-bit address in a register and S2 to specify an offset.
- Register R0 contains all 0's, so it can be used in any field to specify a zero quantity.
- The third instruction format combines the last three fields to form a 19-bit relative address Y and is used primarily with the jump and call instructions. The COND field replaces the Rd field for jump instructions and is used to specify one of 16 possible branch conditions.

The 31 instructions of RISC I are listed in Table 8-12. They have been grouped into three categories. Data manipulation instructions perform arithmetic, logic, and shift operations. The symbols under the opcode and operands columns are used when writing assembly language programs. The register transfer and description columns explain the instruction in register transfer notation and in words. Note that all instructions have three operands. The second source S2 can be either a register or an immediate operand, symbolized by the number sign #. Consider, for example, the ADD instruction and how it can be used to perform a variety of operations.

ADD R22 , R21 , R23	$R23 \leftarrow R22 + R21$
ADD R22 , #150 , R23	$R23 \leftarrow R22 + 150$
ADD R0 , R21 , R22	$R22 \leftarrow R21$ (Move)
ADD R0 , #150 , R22	$R22 \leftarrow 150$ (Load immediate)
ADD R22 , #1 , R22	$R22 \leftarrow R22 + 1$ (Increment)

By using register R0, which always contains 0's, it is possible to transfer the contents of one register or a constant into another register. The increment operation is accomplished by adding a constant 1 to a register.

The load and store instructions move data between a register and memory. The load instructions accommodate signed or unsigned data of eight bits (byte) or 16 bits (short word). The long-word instructions operate on 32-bit data. Although there appears to be a register plus displacement addressing mode in data transfer instructions, register indirect addressing and direct addressing is also possible. The following are examples of load long instructions with different addressing modes.

LDL (R22) #150 , R5	$R5 \leftarrow M[R22 + 150]$
LDL (R22) #0 , R5	$R5 \leftarrow M[R22]$
LDL (R0) #500 , R5	$R5 \leftarrow M[500]$

Table 31 Instruction Set of Berkeley RISC I

Opcode	Operands	Register Transfer	Description
Data manipulation instructions			
ADD	Rs,S2,Rd	$Rd \leftarrow Rs + S2$	Integer add
ADDC	Rs,S2,Rd	$Rd \leftarrow Rs + S2 + \text{carry}$	Add with carry
SUB	Rs,S2,Rd	$Rd \leftarrow Rs - S2$	Integer subtract
SUBC	Rs,S2,Rd	$Rd \leftarrow Rs - S2 - \text{carry}$	Subtract with carry
SUBR	Rs,S2,Rd	$Rd \leftarrow S2 - Rs$	Subtract reverse
SUBCR	Rs,S2,Rd	$Rd \leftarrow S2 - Rs - \text{carry}$	Subtract with carry
AND	Rs,S2,Rd	$Rd \leftarrow Rs \wedge S2$	AND
OR	Rs,S2,Rd	$Rd \leftarrow Rs \vee S2$	OR
XOR	Rs,S2,Rd	$Rd \leftarrow Rs \oplus S2$	Exclusive-OR
SLL	Rs,S2,Rd	$Rd \leftarrow Rs$ shifted by $S2$	Shift-left
SRL	Rs,S2,Rd	$Rd \leftarrow Rs$ shifted by $S2$	Shift-right logical
SRA	Rs,S2,Rd	$Rd \leftarrow Rs$ shifted by $S2$	Shift-right arithmetic
Data transfer instructions			
LDL	(Rs)S2,Rd	$Rd \leftarrow M[Rs + S2]$	Load long
LDSU	(Rs)S2,Rd	$Rd \leftarrow M[Rs + S2]$	Load short unsigned
LDSS	(Rs)S2,Rd	$Rd \leftarrow M[Rs + S2]$	Load short signed
LDBU	(Rs)S2,Rd	$Rd \leftarrow M[Rs + S2]$	Load byte unsigned
LDBS	(Rs)S2,Rd	$Rd \leftarrow M[Rs + S2]$	Load byte signed
LDHI	Rd,Y	$Rd \leftarrow Y$	Load immediate high
STL	Rd,(Rs)S2	$M[Rs + S2] \leftarrow Rd$	Store long
STS	Rd,(Rs)S2	$M[Rs + S2] \leftarrow Rd$	Store short
STB	Rd,(Rs)S2	$M[Rs + S2] \leftarrow Rd$	Store byte
GETPSW	Rd	$Rd \leftarrow PSW$	Load status word
PUTPSW	Rd	$PSW \leftarrow Rd$	Set status word
Program control instructions			
JMP	COND, S2(Rs)	$PC \leftarrow Rs + S2$	Conditional jump
JMPR	COND,Y	$PC \leftarrow PC + Y$	Jump relative
CALL	Rd,S2(Rs)	$Rd \leftarrow PC$ $PC \leftarrow Rs + S2$ $CWP \leftarrow CWP - 1$	Call subroutine and change window
CALLR	Rd,Y	$Rd \leftarrow PC$ $PC \leftarrow PC + Y$ $CWP \leftarrow \text{CWP} - 1$	Call relative and change window
RET	Rd,S2	$PC \leftarrow Rd + S2$ $CWP - CWP + 1$	Return and change window
CALLINT	Rd	$Rd \leftarrow PC$ $CWP \leftarrow CWP - 1$	Disable interrupts
RETINT	Rd,S2	$PC \leftarrow Rd + S2$ $CWP \leftarrow CWP + 1$	Enable interrupts
GTLPC	Rd	$Rd \leftarrow PC$	Get last PC

The program control instructions operate with the program counter PC to control the program sequence. There are two jump and two call instructions. One uses an index plus displacement addressing; the second uses a relative to PC mode with the 19-bit Y value as the relative address. The call and return instructions use a 3-bit CWP (current window pointer) register which points to the currently active register window. Every time the program calls a new procedure, CWP is decremented by one to point to the next-lower register window. After a return instruction CWP is incremented by one to return to the previous register window.