UNIT – 4

Syllabus : Central Processing Unit : General Register Organization, STACK Organization, Instruction Formats, Addressing Modes, Data Transfer and Manipulation, Program Control, Reduced Instruction Set Computer.
Micro programmed Control : Control Memory, Address Sequencing, Micro Program Example, Design of Control Unit.

Central Processing Unit : The part of the computer that performs the bulk of data processing operations is called the central processing unit and is referred to as the CPU. The CPU is made up of three major parts, as shown below.

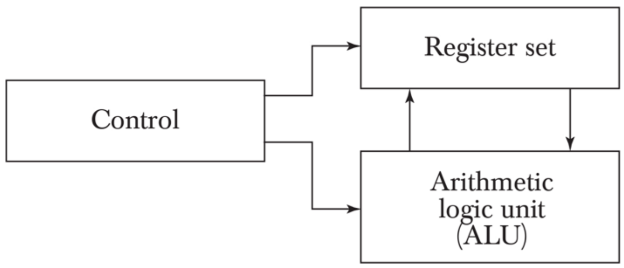


Figure 8-1 Major components of CPU.

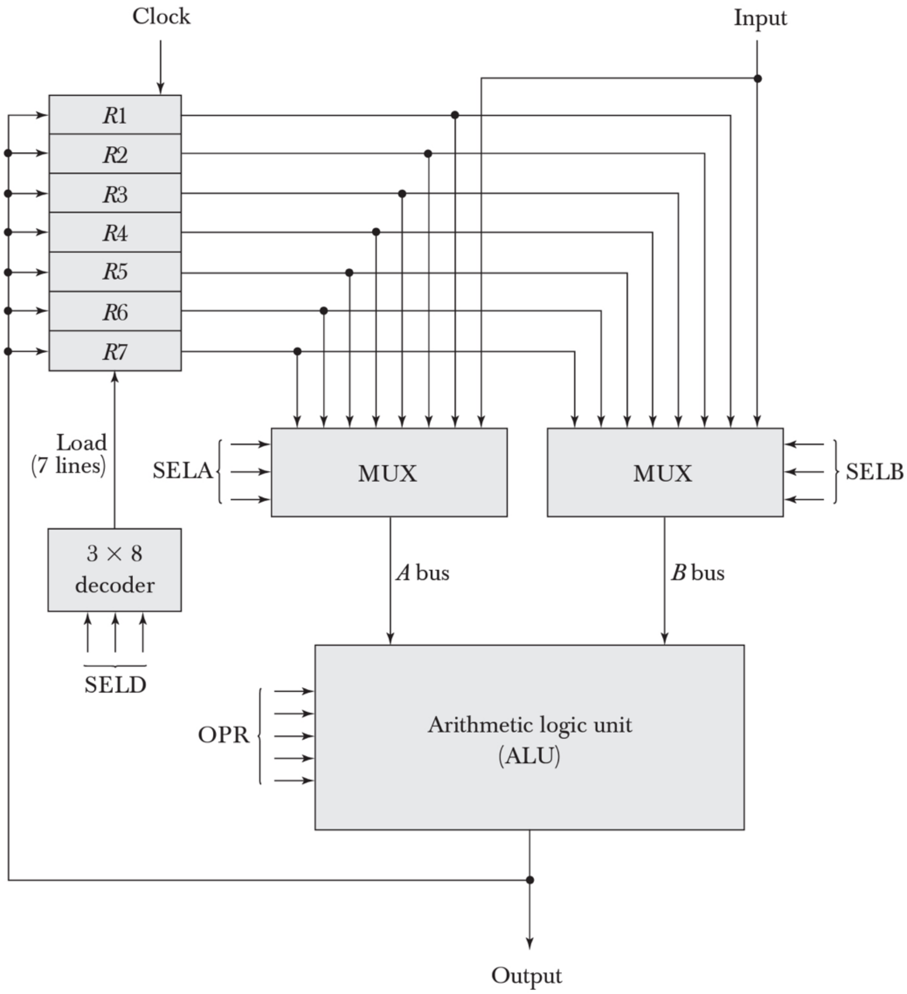
The register set stores intermediate data used during the execution of the instructions. The arithmetic logic unit (ALU) performs the required microoperations for executing the instructions. The control unit supervises the transfer of information among the registers and instructs the ALU as to which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

General Register Organization : Referring the memory locations for data is time consuming rather than referring processor registers. 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.

A bus organization for seven CPU registers is shown below. The output of each register is connected to two multiplexers (MUX) to form the two buses A and B. The selection lines in each multiplexer select one register or the input data for the particular bus. The A and B buses form the inputs to a common arithmetic logic unit (ALU). The operation selected in the ALU determines the arithmetic or logic microoperation that is to be performed. The result of the microoperation is available for output data and also goes into the inputs of all the registers. The register that receives the information from the output bus is selected by a decoder. The decoder activates one of the register load inputs, thus providing a transfer path between the data in the output bus and the inputs of the selected destination register.

The control unit that operates the CPU bus system directs the information flow through the registers and ALU by selecting the various components in the system. 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.



(a) Block diagram

3	3	3	5
SELA	SELB	SELD	OPR

(b) Control word

Figure 8-2 Register set with common ALU.

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 shown above. It consists of four fields. Three fields contain three bits each, and one field has five bits. The three bits of SELA select a source register for the A input of the ALU. The three bits of SELB select a register for the B input of the ALU. The three bits of SELD select a destination register using the decoder and its seven load outputs. The five bits of OPR select one of the operations in the ALU. The 14-bit control word when applied to the selection inputs specify a particular microoperation.

The encoding of the register selections is specified in the following table. The 3-bit binary code listed in the first column of the table specifies the binary code foreach of the three fields. The register selected by fields SELA, SELB, and SELD is the one whose decimal number is equivalent to the binary number in the code. 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.

TABLE 8-1 Encoding of Register Selection Fields

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	$R1$	$R1$	$R1$
010	$R2$	$R2$	$R2$
011	$R3$	$R3$	$R3$
100	$R4$	$R4$	$R4$
101	$R5$	$R5$	$R5$
110	$R6$	$R6$	$R6$
111	$R7$	$R7$	$R7$

TABLE 8-2 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 ALU provides arithmetic and logic operations. In addition, the CPU must provide shift operations. The shifter may be placed in the input of theALU to provide a pre-shift capability, or at the output of the ALU to provide post-shifting capability. In some cases, the shift operations are included with theALU. Different types of operations performed by ALU are listed in above table.

Examples of Micro-Operations : 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. The binary control word for the subtract 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

Some examples of micro operations are given below.

TABLE 8-3 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 ← Input	Input	–	None	TSFA	000 000 000 00000
$R4 \leftarrow$ sh1 $R4$	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 stack in digital computers is essentially a memory unit with an address register that can count only. 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.

The two operations of a stack are the insertion and deletion of items. The operation of insertion is called push (or push-down) because it can be thought of as the result of pushing a new item on top. The operation of deletion is called pop (or pop-up) because it can be thought of as the result of removing one item so that the stack pops up. However, nothing is pushed or popped in a computer stack'. 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. The following diagram shows the organization of a 64-word register stack.

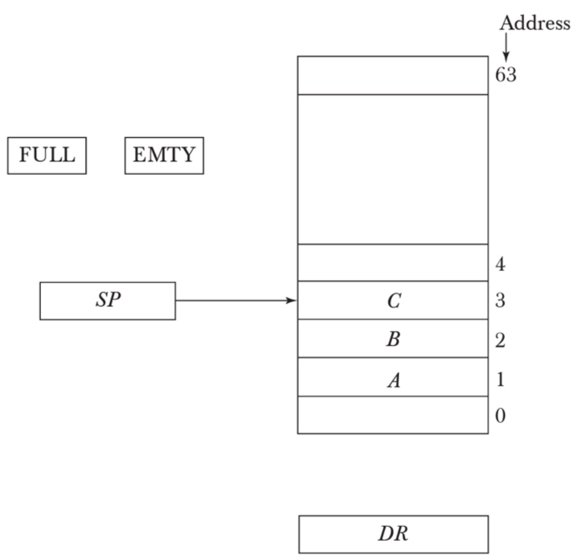


Figure 8-3 Block diagram of a 64-word stack.

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 = 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:

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

A new item is deleted from the stack if the stack is not empty (if EMTY = 0). The pop operation consists of the following sequence of microoperations:

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

Memory Stack : Stack operation can also be implemented in computer memories. This can be done by assigning few segments of the main memory to store the stack, data and program instructions. 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. The following diagram shows a portion of computer memory partitioned into three segments: program, data, and stack.

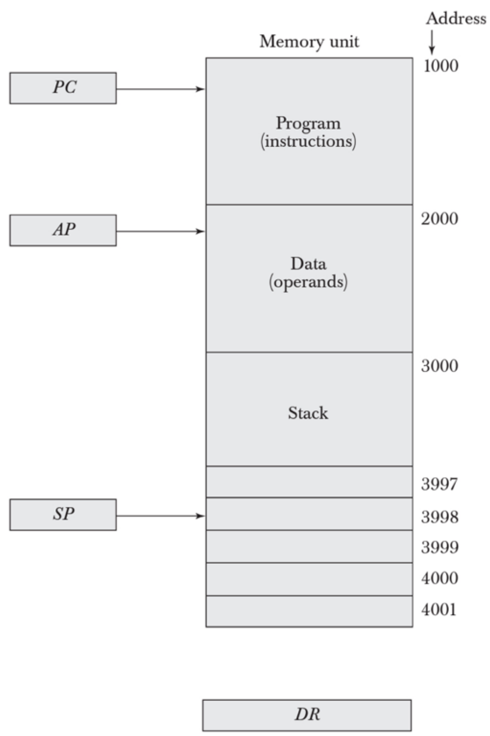


Figure 8-4 Computer memory with program, data, and stack segments.

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.

The initial value of Stack Pointer 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 stored for the stack is 3000. A new item is inserted with the PUSH operation as follows.

$$SP \leftarrow SP - 1$$

$$M[SP] \leftarrow DR$$

The stack pointer is decremented so that it points at the address of next word. A memory write operation inserts the word from DR into the top of the stack. A new item is deleted with a POP operation as follows:

$$DR \leftarrow M[SP]$$

$$SP \leftarrow SP + 1$$

Here the top item is read from the stack into DR. the stack pointer is then incremented to point at the next item in the 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.

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

Reverse Polish Notation : A stack organization is very effective for evaluating arithmetic expressions. The common arithmetic expressions are written in infix notation, with each operator written between the operands. Consider the simple arithmetic expression A + B can be represented in the following three notations.

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

The Reverse Polish notation is 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.

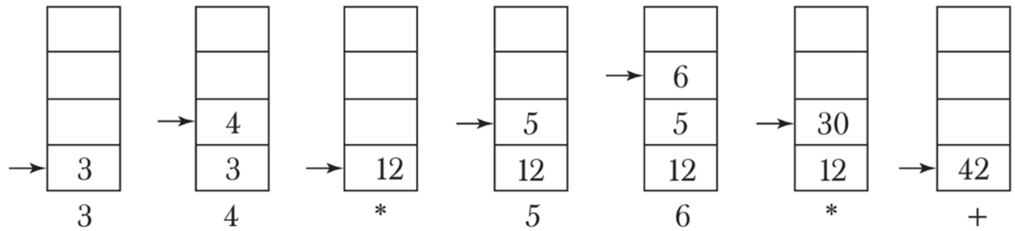
Evaluating Arithmetic Expressions : To evaluate an arithmetic expression follow the below steps.

1. Scan Reverse Polish Notation from left to right.
2. While scanning, if operand is occurred push it into the stack.
3. If operator is occurred pop top two elements from the stack and apply the operator. Again the result is pushed into the stack
4. Continue this process until end of the arithmetic expression

Consider the following arithmetic expression. : (3 * 4) + (5 * 6).

In reverse Polish notation, it is expressed as 3 4 * 5 6 * +. The following represents stack operations to evaluate (3 * 4) + (5 * 6).

Figure 8-5 Stack operations to evaluate 3 • 4 + 5 • 6.



Instruction Formats : The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register. 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.

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 R0 through R15 will have a register address field of four bits. The binary number 0101, will designate register R5.

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.
2. General register organization.
3. Stack organization.

1. **Single Accumulator Organization :** In this organization, All operations are performed with an implied accumulator register. The instruction format in this type of computer uses one address field. For example, the instruction that specifies an arithmetic addition is defined by an assembly language instruction as

   ADD X

   Where X is the address of the operand. The ADD instruction in this case results in the operation AC ← AC + M[X]. AC is the accumulator register and M[X] symbolizes the memory word located at address X.

2. **General Register Organization :** In this Organization, computer will make use of two or three address fields within the instruction format. For example,

   ADD R1, R2, R3 /* R1 ← R2 + R3 */

   This instruction will add the data present in register R1 with the data present in register R3 and the result is stored in register R1.

   ADD R1, R2 /* R1 ← R1 + R2 */

   This instruction will add the data present in register R2 with the data present in register R1 and the result is stored in register R1.

   ADD R1, X /* R1 ← R1 + M[X] */

   This instruction will add the data present in register R1 with the data present at address X in memory location and the result is stored in register R1.

3. **Stack Organization :** In this organization, computer uses only one address filed to performs all its operations. Computers with stack organization would have PUSH and POP instructions which require an address field. For example,

   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 in stack-organized computers. This is because the operation is performed on the two items that are on top of the stack. For example, the instruction ADD in a stack computer consists of an operation code only with no address field. This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack.

Types of Address Instructions : The following are different types of address instructions.

1. **Three Address Instructions :** Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand. The program in assembly language that evaluates X = (A + B) * (C + D) using three address instructions is shown below.

```
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.

2. **Two Address Instructions :** Two-address instructions are the most common in commercial computers. Here again each address field can specify either a processor register or a memory word. The program to evaluate X = (A + B) * (C + D) using two address instructions is as follows:

```
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
```

   The MOV instruction moves or transfers the operands to and from memory and processor registers.

3. **One Address Instructions :** One-address instructions use an implied accumulator (AC) register for all data manipulation. The program to evaluate X = (A + B) * (C + D) using one address instruction is shown below.

```
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
```

   All operations are done between the AC register and a memory operand. Here, T is the address of a temporary memory location required for storing the intermediate result.

4. **Zero Address Instructions :** A stack organized computer does not use an address field for the instructions ADD and MUL. The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack. The following program shows how X = (A + B) * (C + D) will be written for a stack organized computer. (TOS stands for top of stack).

```
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 ← (C + D) * (A + B)
POP     X     M[X] ← TOS
```

Addressing Modes : The operation field of an instruction specifies the operation to be performed. This operation must be executed on some data stored in computer registers or memory words. The way the operands are chosen during program execution is dependent on the addressing mode of the instruction. The following are different types of addressing modes:

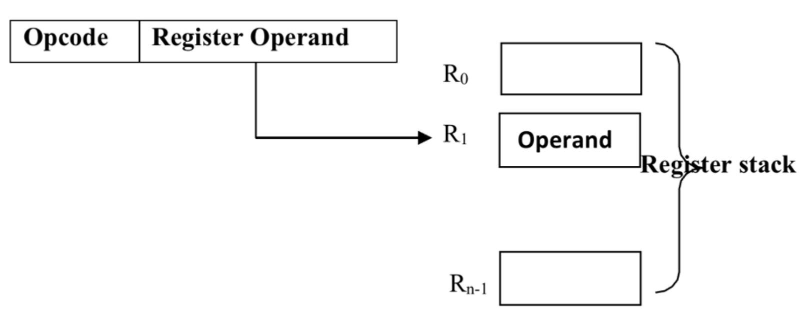
1. **Implied Mode :** In this mode the operands are specified implicitly in the definition of the instruction. For example, the instruction "complement accumulator" is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction. In fact, all register reference instructions that use an accumulator are implied-mode instructions. Zero-address instructions in a stack organized computer are implied mode instructions since the operands are implied to be on top of the stack. Instruction format with implied address mode is as shown below.

Opcode	Mode	Address

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

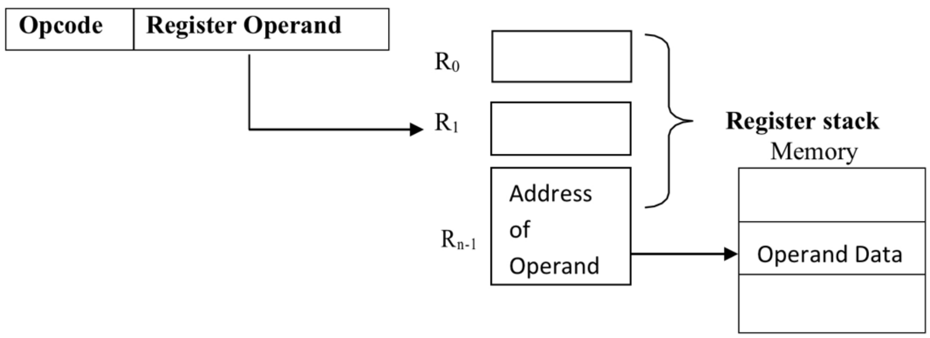
Opcode	Immediate Operand

3. **Register Mode :** In this mode the operands are in registers that reside within the CPU. The particular register is selected from a register field in the instruction. A k-bit field can specify any one of 2^k registers.

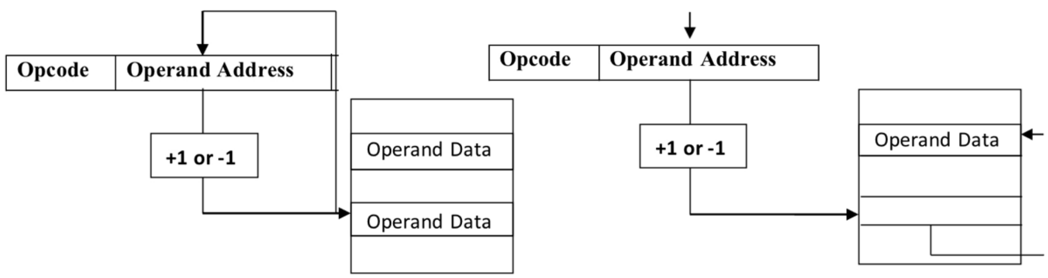


4. **Register Indirect Mode :** In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory. In other words, the selected register contains the address of the operand rather than the operand itself. A reference to the register is then equivalent to specifying a memory address.

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.

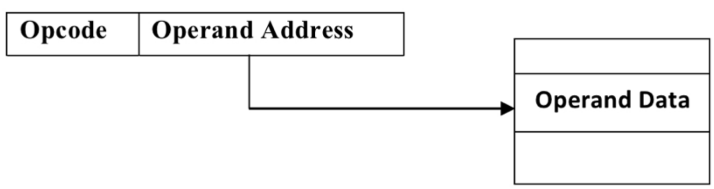


5. **Auto-increment or Auto-decrement Mode :** In this addressing mode, when the address in the register is used to access memory, the value in the register is incremented or decremented by 1 automatically. It can be implemented as memory direct or memory indirect register.



Memory direct Register Memory Indirect Register

6. **Direct Address Mode :** In this mode the effective address is equal to the address part of the instruction. The operand resides in memory and its address is given directly by the address field of the instruction. In a branch-type instruction the address field specifies the actual branch address.

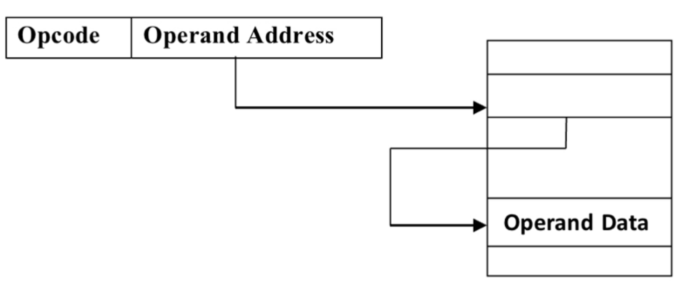


7. **Indirect Address Mode :** In this mode the address field of the instruction gives the address where the effective address is stored in memory. Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.

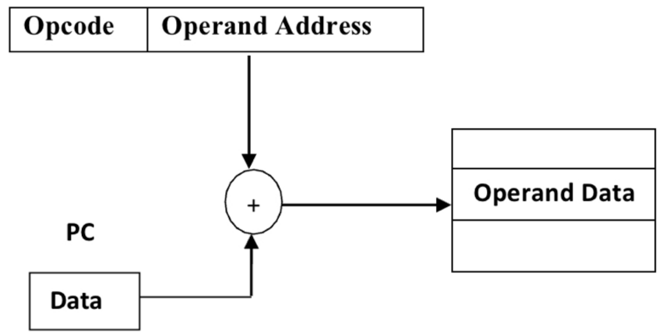
A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU. The effective address in these modes is obtained from the following computation:

effective address = address part of instruction + content of CPU register

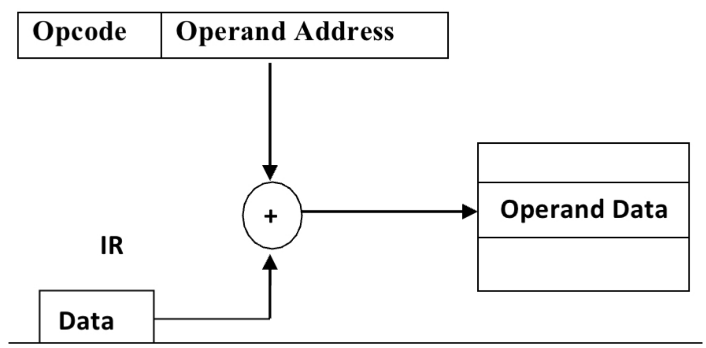
The CPU register used in the computation may be the program counter, an index register, or a base register.



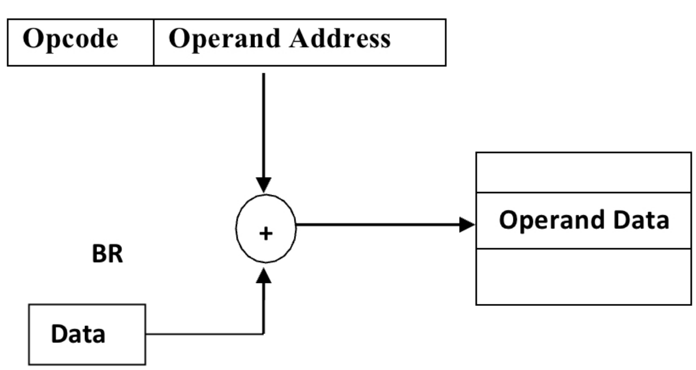
8. **Relative Addressing Mode :** In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address. The address part of the instruction is usually a signed number (in 2's complement representation) which can be either positive or negative. When this number is added to the content of the program counter, the result produces an effective address whose position in memory is relative to the address of the next instruction.



9. **Indexed Addressing Mode :** In this mode the content of an index register is added to the address part of the instruction to obtain the effective address. The index register is a special CPU register that contains an index value. The address field of the instruction defines the beginning address of a data array in memory. The distance between the beginning address and the address of the operand is the index value stored in the index register.



10. **Base Register Addressing Mode :** In this mode the content of a base register is added to the address part of the instruction to obtain the effective address. This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register. The difference between the two modes is in the way they are used rather than in the way that they are computed. An index register is assumed to hold an index number that is relative to the address part of the instruction. A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address.



Example : Consider a two-word instruction at address 200 and 201 (load to AC) with an address field equal to 500. The first word of the instruction specifies the operation code and mode, and the second word specifies the address part. PC has the value 200 for fetching this instruction. The content of processor register R1is 400, and the content of an index register XR is 100. AC receives the operand after the instruction is executed.

The mode field of the instruction can specify any one of a number of modes. For each possible mod, calculate the effective address and the operand that must be loaded into AC.

- In the direct address mode the effective address is the address part of the instruction 500 and the operand to be loaded into AC is 800.
- In the immediate mode the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into AC. (The effective address in this case is 201.)
- In the indirect mode the effective address is stored in memory at address 500. Therefore, the effective address is 800 and the operand is 300.
- In the relative mode the effective address is 500 + 202 = 702 and the operand is 325.
- In the index mode the effective address is XR + 500 = 100 + 500 = 600 and the operand is 900.
- In the register mode the operand is in R1 and 400 is loaded into AC.
- In the register indirect mode the effective address is 400, equal to the content of R1 and the operand loaded into AC is 700.
- The auto increment mode is the same as the register indirect mode except that R1 is incremented to 401 after the execution of the instruction.
- The auto decrement mode decrements R1 to 399 prior to the execution of the instruction. The operand loaded into AC is now 450.

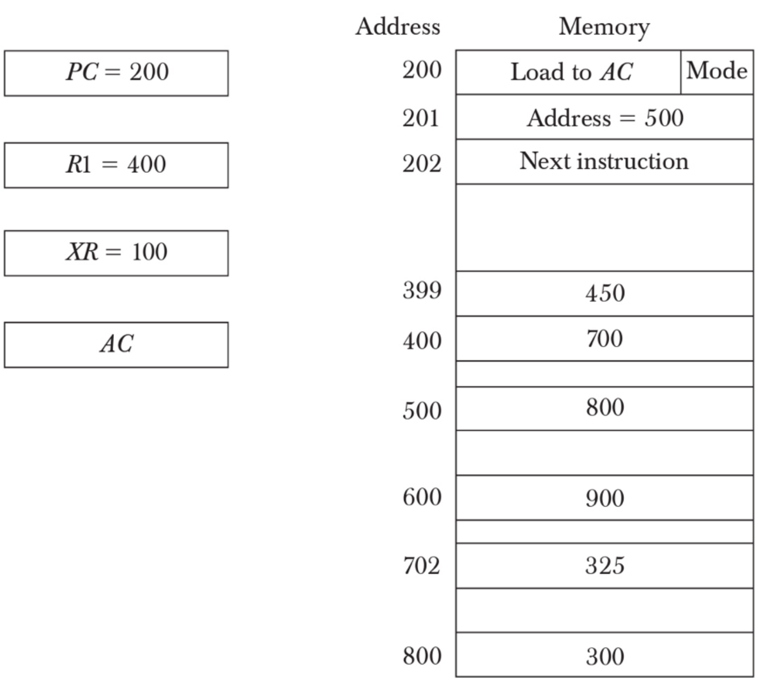


Figure 8-7 Numerical example for addressing modes.

TABLE 8-4 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

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. The actual operations available in the instruction set are not very different from one computer to another. Most computer instructions can be classified into three categories:

1. Data transfer instructions
2. Data manipulation instructions
3. Program control instructions

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

Data Transfer Instructions : Data transfer instructions move data from one place in the computer to another without changing the data content. The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves. The following table gives a list of eight data transfer instructions used in many computers.

TABLE 8-5 Typical Data Transfer Instructions

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

- 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.

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

1. **Arithmetic Instructions :** The four basic arithmetic operations are addition, subtraction, multiplication, and division. A list of typical arithmetic instructions is given in the following table.

   The Increment instruction adds 1 to the value stored in a register or memory word. One common characteristic of the increment operations when executed in processor registers is that a binary number of all 1's when incremented produces a result of all 0's. The decrement instruction subtracts 1 from a value stored in a register or memory word. A number with all 0's, when decremented, produces a number with all 1's.

TABLE 8-7 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

TABLE 8-8 Typical Logical and Bit Manipulation Instructions

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

2. **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 value, 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 the above table.
   - The Clear instruction causes the specified operand to be replaced by 0's.
   - The Complement instruction produces the l's complement by inverting all the bits of the operand.
   - 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 AND 0 = 0 and x AND 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 OR 1 = l and x OR 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 XOR 1 = x` and x XOR 0 = x. Thus a binary variable is complemented when XORed with a 1 but does not change in value when XORed with a 0.

- 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.

3. **Shift Instructions :** 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 either logical shifts, arithmetic shifts, or rotate-type operations. In either case the shift may be to the left or to the right. The following table lists different shift instructions.

TABLE 8-9 Typical Shift Instructions

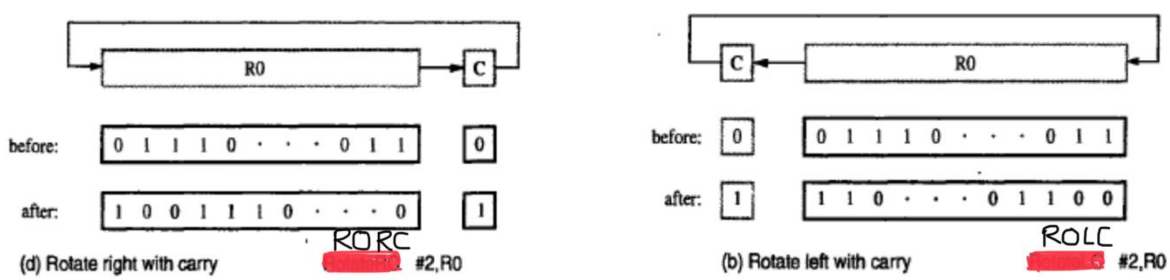
Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

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.

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.

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 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 to the left.



(d) Rotate right with carry

(b) Rotate left with carry

Program Control Instructions : 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 causes a break in the sequence of instruction execution. Program Control instructions are used to switch the control to different locations. The following table lists different types of program control instructions.

TABLE 8-10 Typical Program Control Instructions

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

The branch is usually a one-address instruction. It is written in assembly language as BR ADR, where ADR is a symbolic 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. If the condition is not met, control proceeds with the next instruction in sequence.

The call and return instructions are used in conjunction with subroutines.

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 are the carry bit, the sign bit, a zero indication, and an overflow condition.

Stratus Bit Conditions : Status bits are also called condition-code bits or flag bits. The following diagram shows the block diagram of an 8-bit ALU with a 4-bit status register.

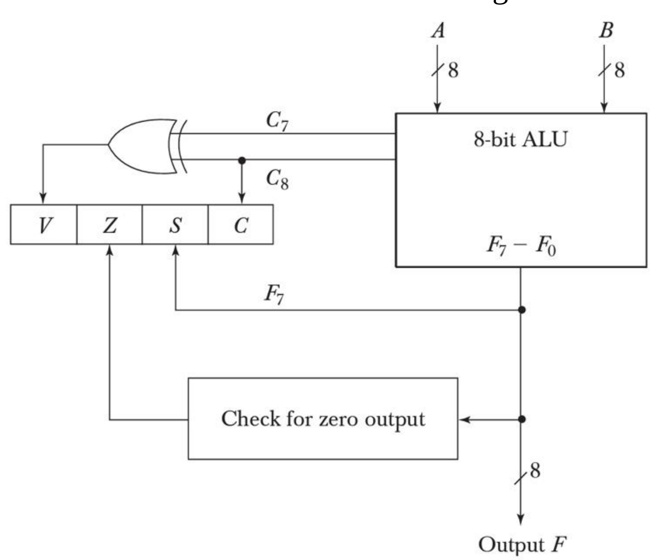


Figure 8-8 Status register bits.

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 F7 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.

Conditional Branch Instructions : The following table gives a list of the most common branch instructions.

TABLE 8-11 Conditional Branch Instructions

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	$A \neq B$

Program Interrupt : 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 is usually initiated by an internal or external signal rather than from the execution of an instruction. The address of the interrupt service program is determined by the hardware rather than from the address field of an instruction; and an interrupt procedure usually stores all the information necessary to define the state of the CPU rather than storing only the program counter.

The collection of all status bit conditions in the CPU is sometimes called a program status word or PSW. 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.

The hardware procedure for processing an interrupt is very similar to the execution of a subroutine call instruction. The state of the CPU is pushed into a memory stack and the beginning address of the service routine is transferred to the program counter. The beginning address of the service routine is determined by the hardware rather than the address field of an instruction. Some computers assign one memory location where interrupts are always transferred. The service routine must then determine what caused the interrupt and proceed to service it.

The CPU does not respond to an interrupt until the end of an instruction execution. Just before going to the next fetch phase, control checks for any interrupt signals. If an interrupt is pending, control goes to a 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:

- External interrupts
- Internal interrupts
- Software interrupts

External 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. 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.

Internal interrupts arise from illegal or erroneous use of an instruction or date. Internal interrupts are also called traps. 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.

The difference between internal and external interrupts is that the internal interrupt is initiated by some exceptional condition caused by the program itself rather than by an external event. Internal interrupts are synchronous with the program while external interrupts are asynchronous.

External and internal interrupts are initiated from signals that occur in the hardware of the CPU. A software interrupt is initiated by executing an instruction. Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program. The most common use of software interrupt is associated with a supervisor call instruction. This instruction provides means for switching from a CPU user mode to the supervisor mode.

Reduced Instruction Set Computer : A computer with a large number of instructions is classified as a complex instruction set computer, abbreviated CISC. Examples of CISC architectures are the Digital Equipment Corporation VAX computer and the IBM 370 computer.

A computer with a few number of instructions is classified as a Reduced instruction set computer, abbreviated RISC.

CISC Characteristics :

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

Micro programmed Control : Control Memory, Address Sequencing, Micro Program Example, Design of Control Unit.

Control Memory : The major functional parts in a digital computer are Central Processing Unit (CPU), Memory, and Input–output. The main digital hardware functional units of CPU are control unit, arithmetic and logic unit, and registers. The function of the control unit in a digital computer is to initiate sequences of microoperations. Two methods of implementing control unit are hardwired control and micro-programmed control.

The design of hardwired control involves the use of fixed instructions, fixed logic blocks of and/or arrays, encoders, decoders, etc. The key characteristics of hardwired control logic are high speed operation, expensive, relatively complex, and no flexibility of adding new instructions. Example CPU's with hardwired logic control are Intel 8085, Motorola 6802, Zilog 80, and any RISC (Reduced Instruction Set Computer) CPUs.

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. For example,CPUs with microprogrammed control unit are Intel 8080, Motorola 68000, and any CISC (Complex Instruction Set Computer) CPUs.

The control function that specifies a microoperation is a binary variable. When it is in one binary state, the corresponding microoperation is executed. The active state of a control variable may be either the 1 state or the 0 state, depending on the application.

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 1'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. The microinstruction specifies one or more microoperations for the system. A sequence of microinstructions constitutes a microprogram. 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.

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 shown in below diagram. 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.

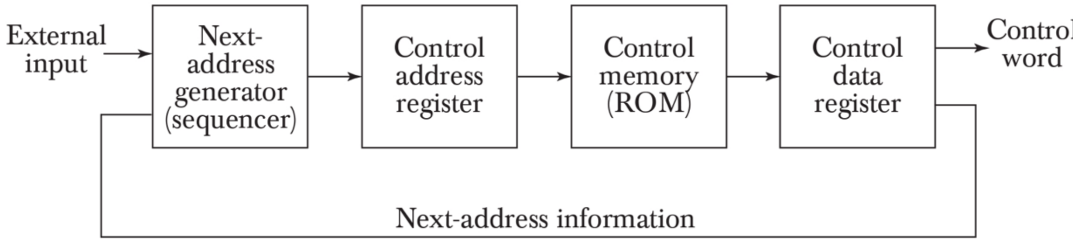


Figure 7-1 Microprogrammed control organization.

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 location of the next microinstruction may be the one next in sequence, or it may be located somewhere else in the control memory. For this reason it is necessary to use some bits of the present microinstruction to control the generation of the address of the next microinstruction. The next address may also be a function of external input conditions.

While the microoperations are being executed, the next address is computed in the next address generator circuit and then transferred into the control address register to read the next microinstruction. Thus a microinstruction contains bits for initiating microoperations in the data processor part and bits that determine the address sequence for the control memory.

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 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. The hardware configuration should not be changed for different operations; the only thing that must be changed is the microprogram residing in control memory.

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.

An initial address is loaded into the control address register when power is turned on in the computer. This address is usually the 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. The effective address computation routine in control memory can be reached through a branch microinstruction, which is conditioned on the status of the mode bits of the instruction. 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. Microprograms that employ subroutines will require an external register for storing the return address.

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:

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

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

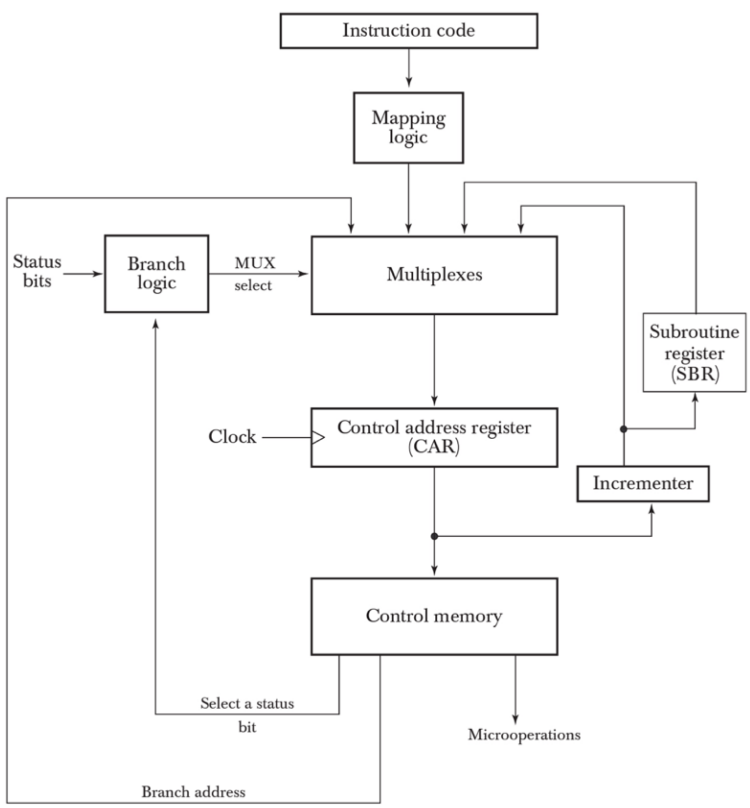


Figure 7-2 Selection of address for control memory.

The microinstruction in control memory contains a set of bits to initiate microoperations in computer registers and other bits to specify the method by which the next address is obtained. The diagram shows four different paths from which the control address register (CAR) receives the address. The incrementer increments the content of the control address register by one, to select the next microinstruction in sequence. Branching is achieved by specifying the branch address in one of the fields of the microinstruction. Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition. An external address is transferred into control memory via a mapping logic circuit. The return address for a subroutine is stored in a special register whose value is then used when the microprogram wishes to return from the subroutine.

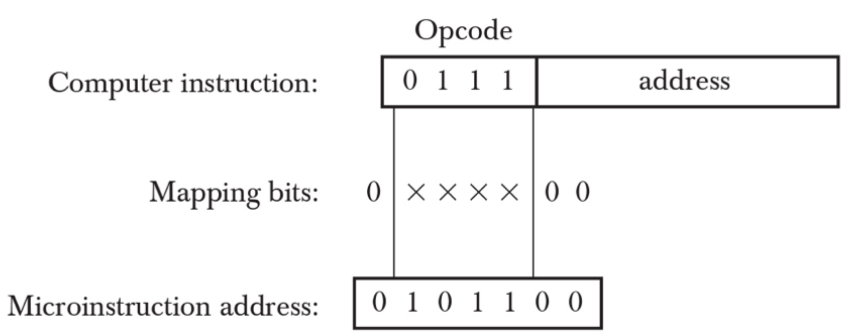
Conditional Branching : The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and input or output status conditions. Information in these bits can be tested and actions initiated based on their condition: whether their value is 1 or 0.

The 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. This can be implemented with a multiplexer.

An unconditional branch microinstruction can be implemented by loading the branch address from control memory into the control address register. This can be accomplished by fixing the value of one status bit at the input of the multiplexer, so it is always equal to 1. A reference to this bit by the status bit select lines from control memory causes the branch address to be loaded into the control address register unconditionally.

Mapping of Instruction : A special type of branch exists when a microinstruction specifies a branch to the first word in control memory. 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 below diagram has an operation code of four bits which can specify up to 16 distinct instructions.

Figure 7-3 Mapping from instruction code to microinstruction address.



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 above diagram. 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. If the routine needs more than four microinstructions, it can use addresses 1000000 through 1111111.

<u>**Micro Program Example :**</u>

Computer Configuration : The block diagram of the computer is shown below.

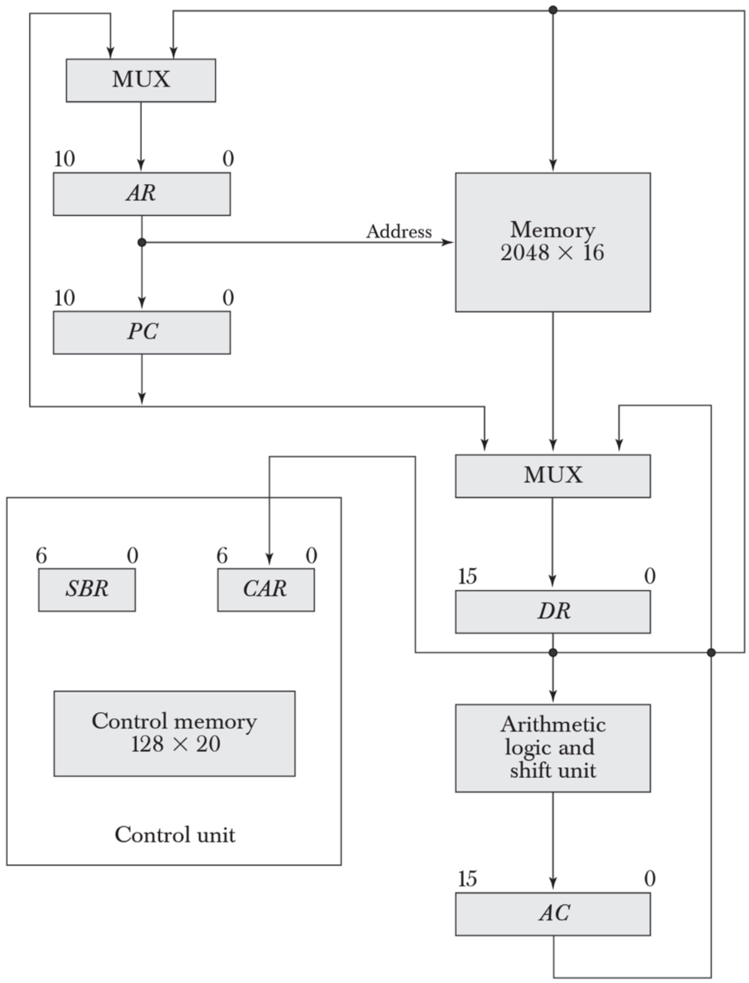
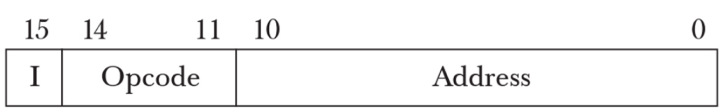


Figure 7-4 Computer hardware fonfiguration.

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 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 shown below.



(a) Instruction format

It consists of three fields: a 1-bit field for indirect addressing symbolized by I, a 4-bit operation code (opcode), and an 11-bit address field.

The following figure lists four of the 16 possible memory-reference instructions. The ADD instruction adds the content of the operand found in the effective address to the content of AC. The BRANCH instruction causes a branch to the effective address if the operand in AC is negative. The program proceeds with the next consecutive instruction if AC is not negative. The AC is negative if its sign bit (the bit in the leftmost position of the register) is a 1. The STORE instruction transfers the content of AC into the memory word specified by the effective address. The EXCHANGE instruction swaps the data between AC and the memory word specified by the effective address.

Symbol	Opcode	Description
ADD	0000	$AC \rightarrow AC + M[EA]$
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

Microinstruction Format : The microinstruction format for the control memory is shown in the following figure. The 20 bits of the microinstruction 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 or 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.

3	3	3	2	2	7
F1	F2	F3	CD	BR	AD

F1, F2, F3: Microoperation fields

CD: Condition for branching

BR: Branch field

AD: Address field

Figure 7-6 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 the following tables.

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	$AR \leftarrow DR(0-10)$	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	$DR(0-10) \leftarrow PC$	PCTDR

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

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. For example,

$$DR \leftarrow M[AR] \qquad \text{with F2} = 100$$

and $$PC \leftarrow PC + 1 \qquad \text{with F3} = 101$$

The nine bits of the microoperation fields will then be 000 100 101.

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.

The CD (condition) field consists of two bits which are encoded to specify four status bit conditions as listed below. The first condition is always a 1, so that a reference to CD = 00 (or the symbol U) will always find the condition to be true. When this condition is used in conjunction with the BR (branch) field, it provides an unconditional branch operation. The indirect bit I is available from bit 15 of DR after an instruction is read from memory. The sign bit of AC provides the next status bit. The zero value, symbolized by Z, is a binary variable whose value is equal to 1 if all the bits in AC are equal to zero.

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.

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	$CAR(2-5) \leftarrow DR(11-14)$, $CAR(0,1,6) \leftarrow 0$

When BR = 00, the control performs a jump (JMP) operation (which is similar to a branch), and when BR = 01, it performs a call to subroutine (CALL) operation. The two operations are identical except that a call microinstruction stores the return address in the subroutine register SBR. The jump and call operations depend on the value of the CD field. It the status bit condition specified in the CD field is equal to 1, the next address in the AD field is transferred to the control address register CAR. Otherwise, CAR is incremented by 1.

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. The bits of the operation code are in DR(11–14) after an instruction is read from memory.

Symbolic Microinstructions : A symbolic microprogram can be translated into its binary equivalent by means of an assembler. The simplest and most straightforward way to formulate an assembly language for a microprogram is to define symbols for each field 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 field 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. 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.
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.

Note : Symbol ORG is used 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.The microinstructions needed for the fetch routine are

$$AR \leftarrow PC$$
$$DR \leftarrow \mathrm{M}[AR], \quad PC \leftarrow PC + 1$$
$$AR \leftarrow DR(0-10), \quad CAR(2-5) \leftarrow DR(11-14), \ CAR(0,1,6) \leftarrow 0$$

The fetch routine needs three microinstructions, which are placed in control memory at addresses 64, 65, and 66. 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.

Binary Address	Fl	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

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.

Sample Microprogram :

TABLE 7-2 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	

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 micro-operation with the content of DR and AC and then jumps back to the beginning of the fetch routine.

The BRANCH instruction should cause a branch to the effective address 0. The AC will be less than zero if its sign is negative, which is detected from status bit 5 being a 1. The BRANCH routine starts by checking the value of S. IfS 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.

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. The original content of AC that is now in DR is stored back in memory.

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.

The equivalent binary form of the symbolic microprogram shown above is listed below. The addresses for control memory are given in both decimal and binary.

TABLE 7-3 Binary Microprogram for Control Memory (Partial)

Micro Routine	Address		Binary Microinstruction					
	Decimal	Binary	Fl	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

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.

The following figure shows the three decoders and some of the connections that must be made from their outputs.

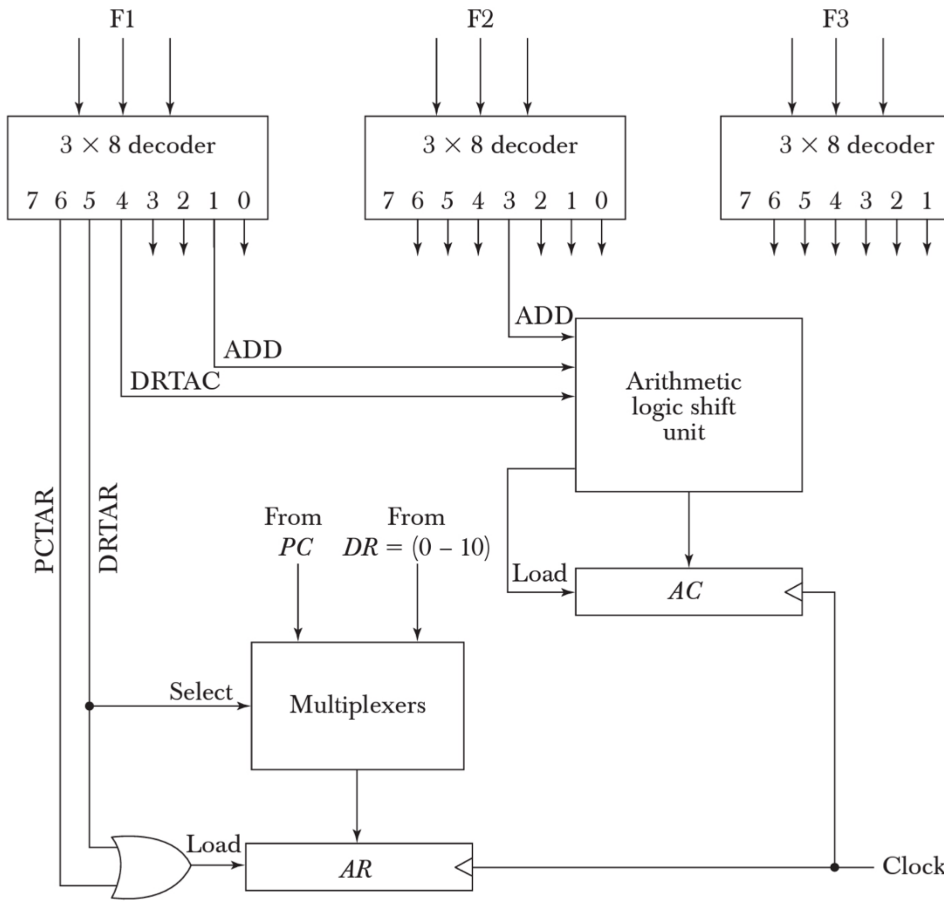


Figure 7-7 Decoding of microoperation fields.

Each of the three fields of the microinstruction presently available in the output of control memory are decoded with a 3 * 8 decoder to provide eight outputs. For example, when F1=101 (binary 5), the next clock pulse transition transfers the content of DR(0–10) to AR (symbolized by DRTAR). Similarly, when F1=110 (binary 6) there is a transfer from PC to AR (symbolized by PCTAR).

Outputs 5 and 6 of decoder F1 are connected to the load input of AR, so that when either one of these outputs is active, information from the multiplexers is transferred to AR. The multiplexers select the information from DR when output 5 is active and from PC when output 5 is inactive.

Micro Program Sequencer : 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.

The block diagram of the microprogram sequencer is shown below. There are two multiplexers in the circuit. The first multiplexer selects an address from one of four sources and routes it into a control address register CAR. The second multiplexer tests the value of a selected status bit and the result of the test is applied to an input logic circuit. The output from CAR provides the address for the control memory. The content of CAR is incremented and applied to one of the multiplexer inputs and to the subroutine register SBR. The other three inputs to multiplexer number 1 come from the address field of the present microinstruction, from the output of SBR, and from an external source that maps the instruction.

The CD (condition) field of the microinstruction selects 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 operation.

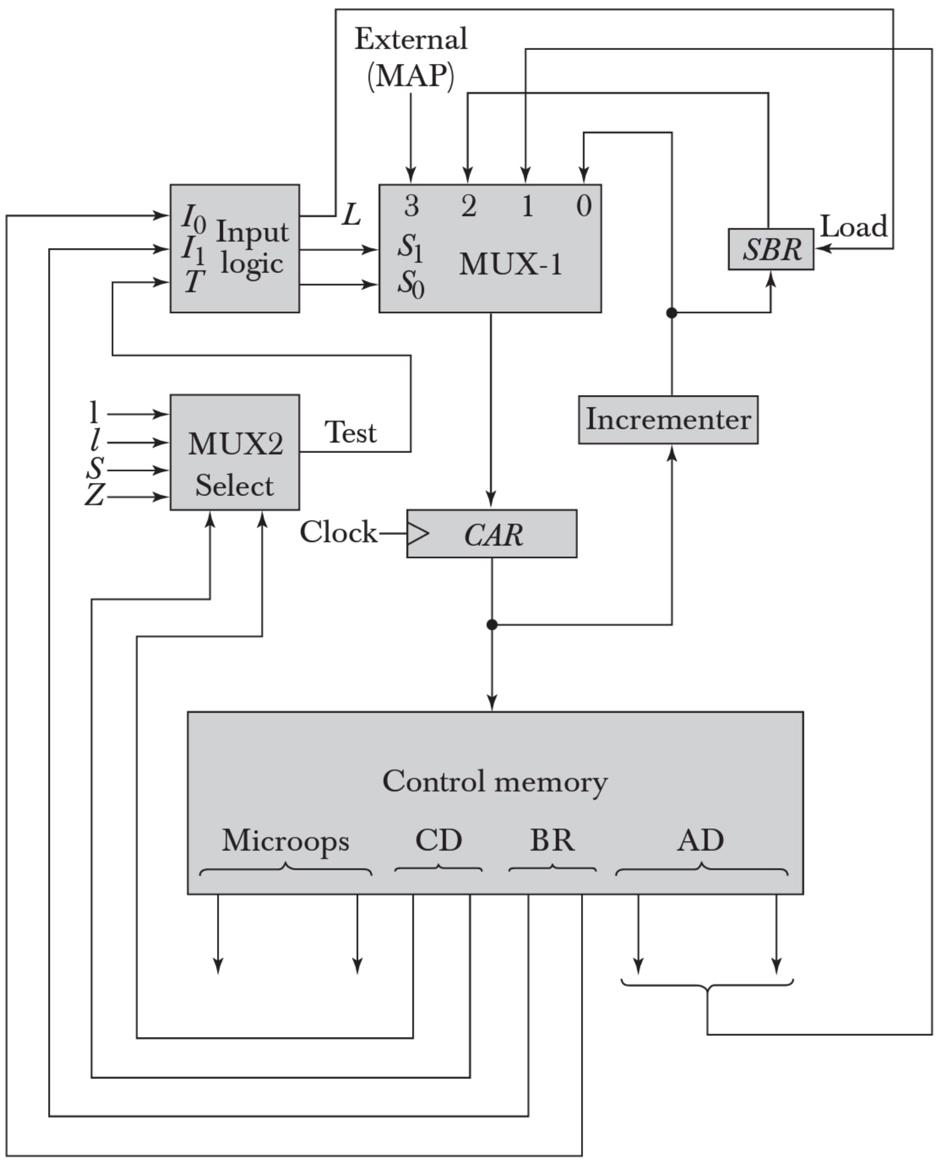


Figure 7-8 Microprogram sequencer for a control memory.

The input logic circuit 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 $S_1S_0 = 10$, multiplexer input number 2 is selected and establishes a transfer path from SBR to CAR.

The truth table for the input logic circuit is shown below. Inputs I_1 and I_0 are identical to the bit values in the BR field. 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).

TABLE 7-4 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	1	1	1
1	0	1	0	×	1	0	0
1	1	1	1	×	1	1	0