Introduction to CPU

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 in Fig.

- The register set stores intermediate data used during the execution of the instructions.

- The arithmetic logic unit (ALU) performs the required micro operations 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.

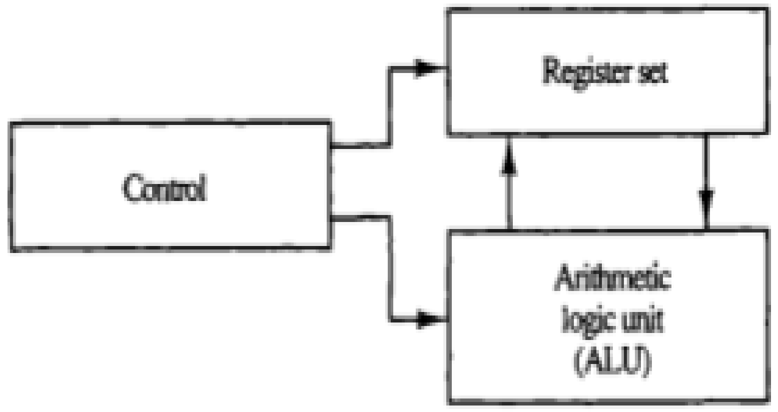


Figure 8-1 Major components of CPU.

General Register Organization

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 micro operations.

Hence it is necessary to provide a common unit that can perform all the arithmetic, logic, and shift micro operations in the processor.

Generally CPU has seven general registers. Register organization show how registers are selected and how data flow between register and ALU.

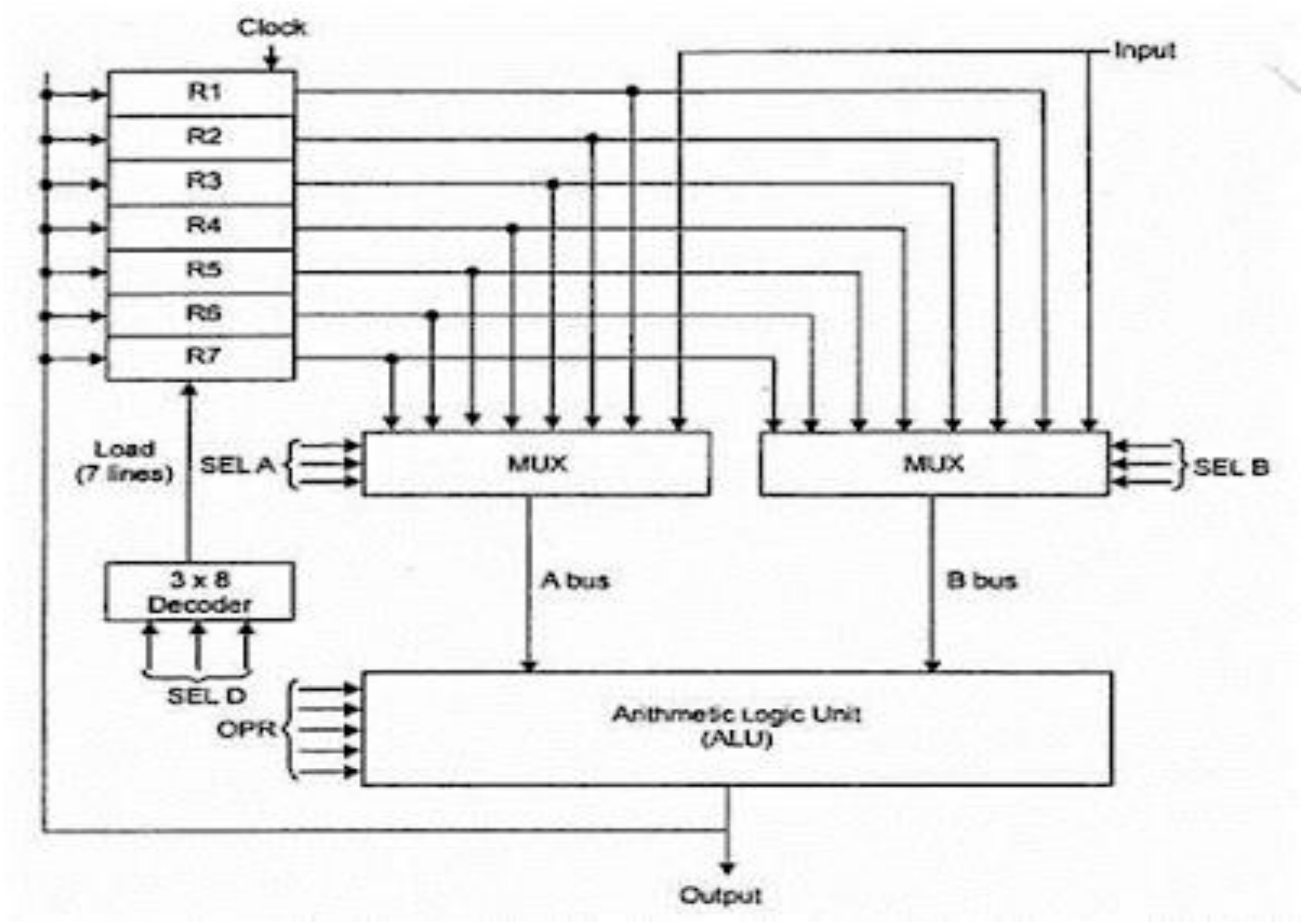
General Register Organization

A decoder is used to select a particular register. The output of each register is connected to two multiplexers to form the two buses A and B. The selection lines in each multiplexer select the input data for the particular bus.

The A and B buses form the two inputs of an ALU. The operation select lines decide the micro operation to be performed by ALU.

The result of the micro operation is available at the output bus. The output bus connected to the inputs of all registers, thus by selecting a destination register it is possible to store the result in it.

General Register Organization



General Register Organization

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<--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 R 1.

Control Word

There are 14 binary selection inputs in the unit, and their combined value specifies a control word.

SEL A	SELB	SELREG OR SELD	SELOPR

FORMATE OF CONTROL WORD

1. The three bit of SELA select a source registers of the **a** input of the ALU.
2. The three bits of SELB select a source registers of the **b** input of the ALU.
3. The three bits of SELED or SELREG select a destination register using the decoder.
4. The four bits of SELOPR select the operation to be performed by ALU.

Control Word

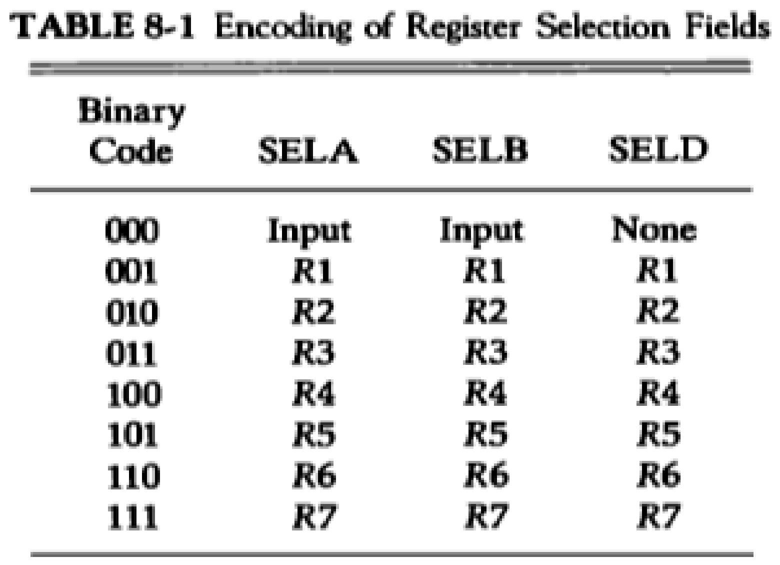


TABLE 8-1 Encoding of Register Selection Fields

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	$R7$	$R7$	$R7$

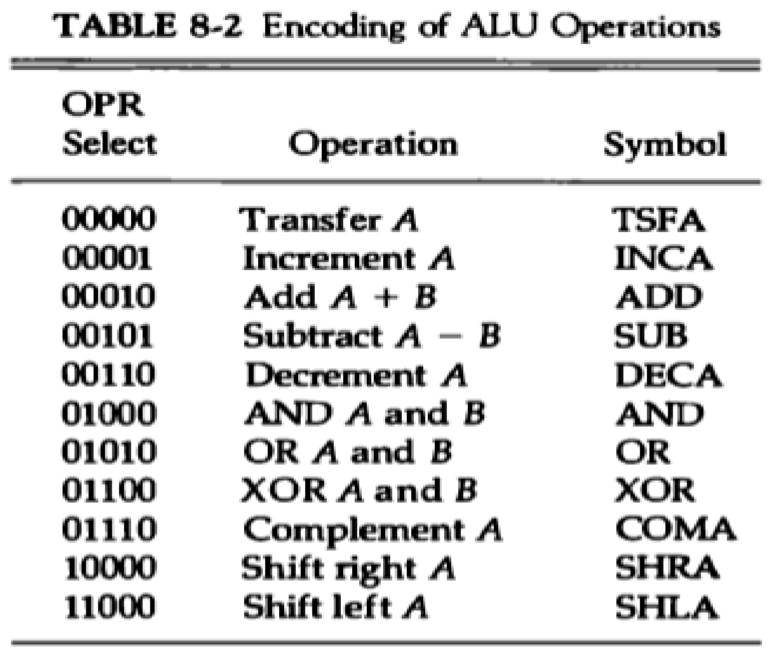


TABLE 8-2 Encoding of ALU Operations

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

A control word of 14 bits is needed to specify a micro operation in the CPU. The control word for a given micro operation can be derived from the selection variables.

Control Word

For example, the subtract micro operation 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.

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

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 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.

Stack Organization

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 8-3 shows the organization of a 64-word register stack.

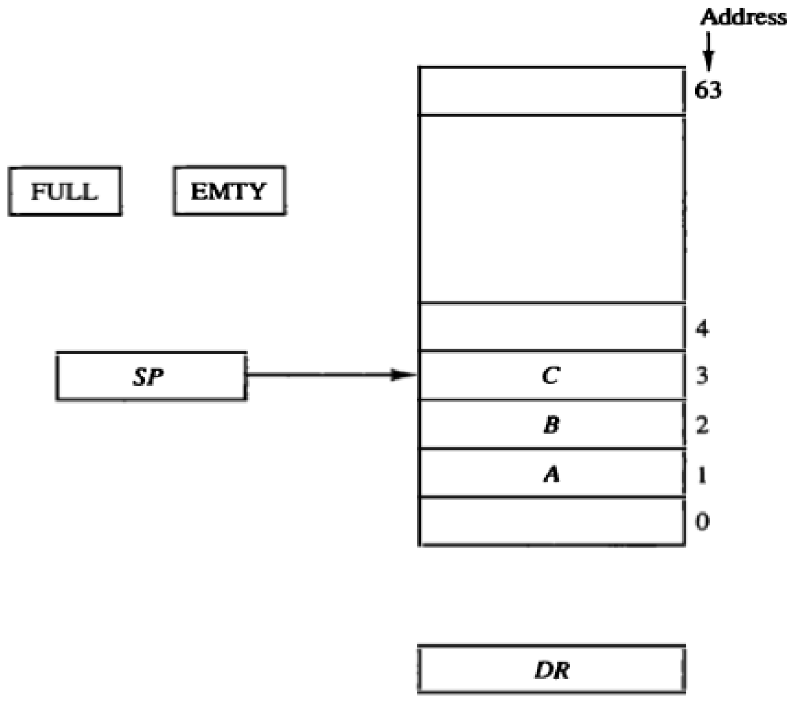


Figure 8-3 Block diagram of a 64-word stack.

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.

Register 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 micro operations;

SP <- SP + 1	Increment stack pointer
M[SP]<-DR	Write item on top of the stack

Register Stack

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 micro operations:

DR <--M[SP]	Read item from the top of stack
SP<--SP - 1	Decrement stack pointer

Memory Stack

A stack can be implemented in a random-access memory attached to a CPU.

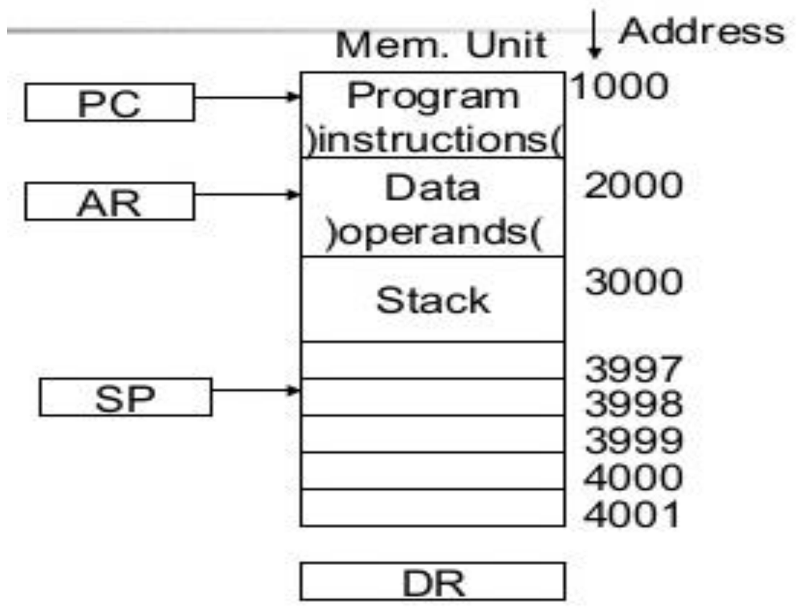
The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer.

The portion of computer memory is 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.

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 &om the stack.

Memory Stack



Computer memory with program, data & stack segments

As shown in Fig., 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.

Memory Stack

We 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<-SP - 1

M[SP]<-DR

The stack pointer is decremented so that it points at the address of the 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 <-M[SP]

SP<-SP + 1

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.

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 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);

(2) the address of the interrupt service program is determined by the hardware rather than from the address field of an instruction; and

(3) an interrupt procedure usually stores all the information necessary to define the state of the CPU rather than storing only the program counter.

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

External interrupts come from input/output (I/0) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source.

External Interrupts

Examples that cause external interrupts are I/0 device requesting transfer of data, I/0 device finished transfer of data, elapsed time of an event, or power failure.

Timeout interrupt may result from a program that is in an endless loop and thus exceeded its time allocation.

Power failure interrupt may have as its service routine a program that transfers the complete state of the CPU into a nondestructive memory in the few milliseconds before power ceases.

Internal Interrupts

Internal interrupts arise from illegal or erroneous use of an instruction or data. 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.

These error conditions usually occur as a result of a premature termination of the instruction execution. The service program that processes the internal interrupt determines the corrective measure to be taken.

Internal vs External Interrupts

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.

If the program is rerun, the internal interrupts will occur in the same place each time. External interrupts depend on external conditions that are independent of the program being executed at the time.

Software Interrupts

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.

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. Certain operations in the computer may be assigned to the supervisor mode only, as for example, a complex input or output transfer procedure.

A program written by a user must run in the user mode. When an input or output transfer is required, the supervisor mode is requested by means of a supervisor call instruction. This instruction causes a software interrupt

Instruction Format

Computer perform task on the basis of instruction provided. **An instruction format** defines layout of bits of an instruction.

A instruction in computer comprises of groups called fields. These field contains different information as for computers every thing is in 0 and 1 so each field has different significance on the basis of which a CPU decide what so perform.

The most common fields are:

- Operation field which specifies the operation to be performed like addition.
- Address field which contain the location of operand, i.e., register or memory location.
- Mode field which specifies how operand is to be founded.

Instruction Format

A instruction is of various length depending upon the number of addresses it contain. Generally CPU organization are of three types on the basis of number of address fields:

- Single Accumulator organization
- General register organization
- Stack organization

In **first** organization operation is done involving a special register called accumulator. In **second** on multiple registers are used for the computation purpose. In **third** organization the work on stack basis operation due to which it does not contain any address field.

1. Single Accumulator organization

All operations are performed with an 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

The instruction format in this type of computer needs three register address fields. Thus the instruction for an arithmetic addition may be written in an assembly language as:

ADD R1, R2, R3

to denote the operation R 1 <---R2 + R3.

The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers. Thus the instruction

ADD R1, R2

would denote the operation R 1 <---R 1 + R2.

3. Stack organization

Computers with stack organization would have PUSH and POP instructions which require an address field. Thus the instruction

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. 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. There is no need to specify operands with an address field since all operands are implied to be in the stack.

Instruction Format

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

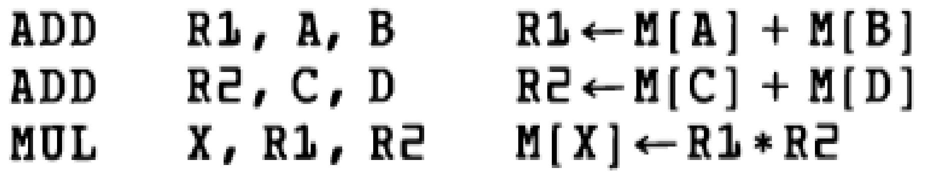
$$X = (A + B) * (C + D)$$

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

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)** is shown below, together with comments that explain the register transfer operation of each instruction.



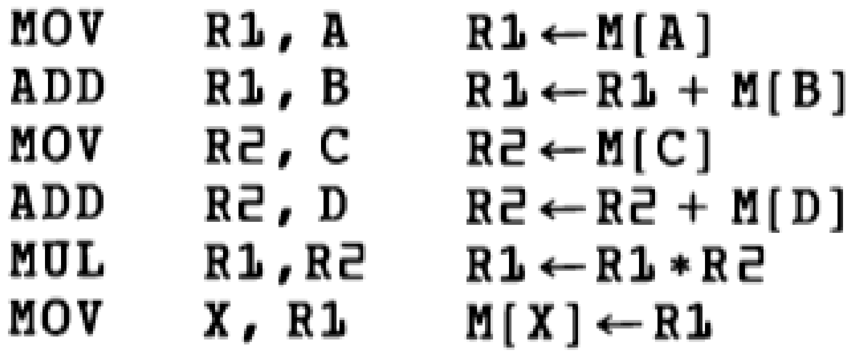
```
ADD   R1, A, B     R1←M[A] + M[B]
ADD   R2, C, D     R2←M[C] + M[D]
MUL   X, R1, R2    M[X]←R1*R2
```

It is assumed that the computer has two processor registers, R1 and R2. The symbol M[A] denotes the operand at memory address symbolized by A.

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

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)** 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. The first symbol listed in an instruction is assumed to be both a source and the destination where the result of the operation is transferred.

One-Address Instructions

One-address instructions use an accumulator (AC) register for all data manipulation. For multiplication and division there is a need for a second register. However, here we will neglect the second register and assume that the AC contains the result of all operations. The program to evaluate **X = (A + B) * (C + D)** is

```
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. T is the address of a temporary memory location required for storing the intermediate result.

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.

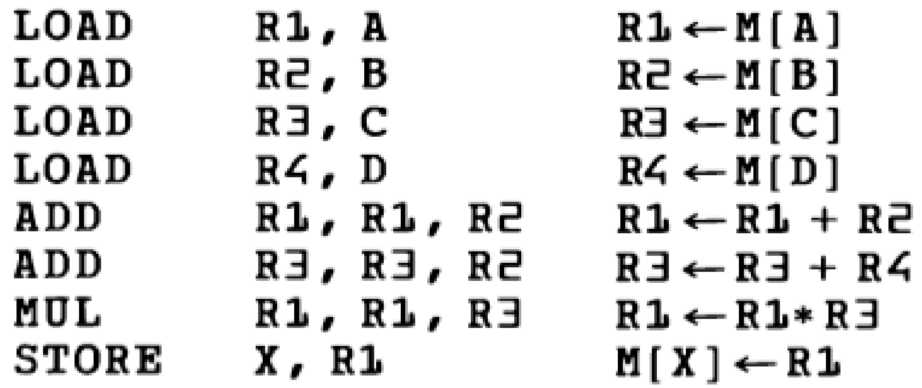
The following program shows how **X = (A + B) * (C + D)** will be written for a stack organized computer. (**TOS** stands for top of stack.)

```
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
```

RISC (Reduced Instruction Set Computer) 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.

The following is a program to evaluate **X = (A + B) * (C + D).**



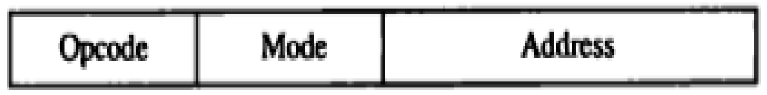
```
LOAD     R1, A          R1 ←M[A]
LOAD     R2, B          R2 ←M[B]
LOAD     R3, C          R3 ←M[C]
LOAD     R4, D          R4 ←M[D]
ADD      R1, R1, R2     R1 ←R1 + R2
ADD      R3, R3, R2     R3 ←R3 + R4
MUL      R1, R1, R3     R1 ←R1*R3
STORE    X, R1          M[X]←R1
```

The load instructions transfer the operands from memory to CPU registers. The add and multiply operations are executed with data in the registers without accessing memory. The result of the computations is then stored in memory with a store instruction.

Addressing Modes

An example of an instruction format with a distinct addressing mode field is shown in Fig. 8-6.

Figure 8-6 Instruction format with mode field.



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, the instruction may have more than one address field, and each address field may be associated with its own particular addressing mode.

Addressing Modes

1. Implied Mode

Implied Addressing Mode also known as "Implicit" or "Inherent" addressing mode is the addressing mode in which, no operand(register or memory location or data) is specified in the instruction. As in this mode the operand are specified implicit in the definition of instruction.

As an example: The instruction: “Complement Accumulator” is an Implied Mode instruction because the operand in the accumulator register is implied in the definition of instruction. In assembly language it is written as:

CMA: Take complement of content of AC

Similarly, the instruction,

RLC: Rotate the content of Accumulator is an implied mode instruction.

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.

Addressing Modes

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.

The address field of an instruction may specify either a memory word or a processor register. When the address field specifies a processor register, the instruction is said to be in the register mode.

Addressing Modes

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.

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.

The instruction specifies the name of the register in which the address of the data is available. Here the data will be in memory and the address will be in the register pair.

Example: **MOV A, M** - The memory data addressed by H L pair is moved to A register.

Addressing Modes

5. Auto increment or Auto decrement Mode:

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.

After accessing the operand, the contents of this register are automatically incremented to the next value.

Before accessing the operand, the contents of this register are automatically decremented and then the value is accessed.

Addressing Modes

6. Direct Address Mode:

Here, 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

Addressing Modes

8. Relative Address 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.

To clarify with an example, assume that the program counter contains the number 825 and the address part of the instruction contains the number 24. The instruction at location 825 is read from memory during the fetch phase and the program counter is then incremented by one to 826. The effective address computation for the relative address mode is 826 + 24 = 850.

Addressing Modes

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.

Each operand in the array is stored in memory relative to the beginning address. The distance between the beginning address and the address of the operand is the index value stored in the index register.

Any operand in the array can be accessed with the same instruction provided that the index register contains the correct index value.

Addressing Modes

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.

Data Transfer & Manipulation

Computers provide an extensive set of instructions to give the user the flexibility to carry out various computational tasks.

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. It may also happen that the symbolic name given to instructions in the assembly language notation is different in different computers, even for the same instruction.

Most computer instructions can be classified into three categories:

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

Data Transfer & Manipulation

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.

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

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. Table 8-5 gives a list of eight data transfer instructions used in many computers.

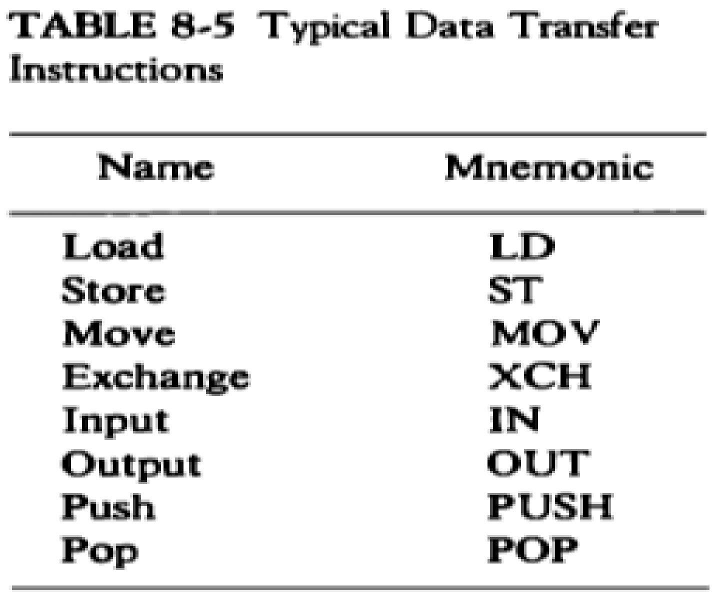


TABLE 8-5 Typical Data Transfer Instructions

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

Data Transfer Instructions

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. Most computers provide instructions for all four operations. Some small computers have only addition and possibly subtraction instructions.

Data Manipulation Instructions

1. Arithmetic Instructions (contd…)

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.

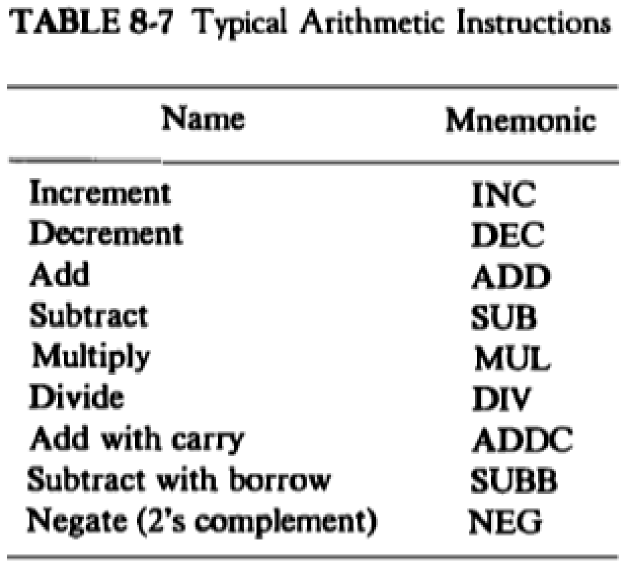


TABLE 8-7 Typical Arithmetic Instructions

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

Data Manipulation Instructions

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.

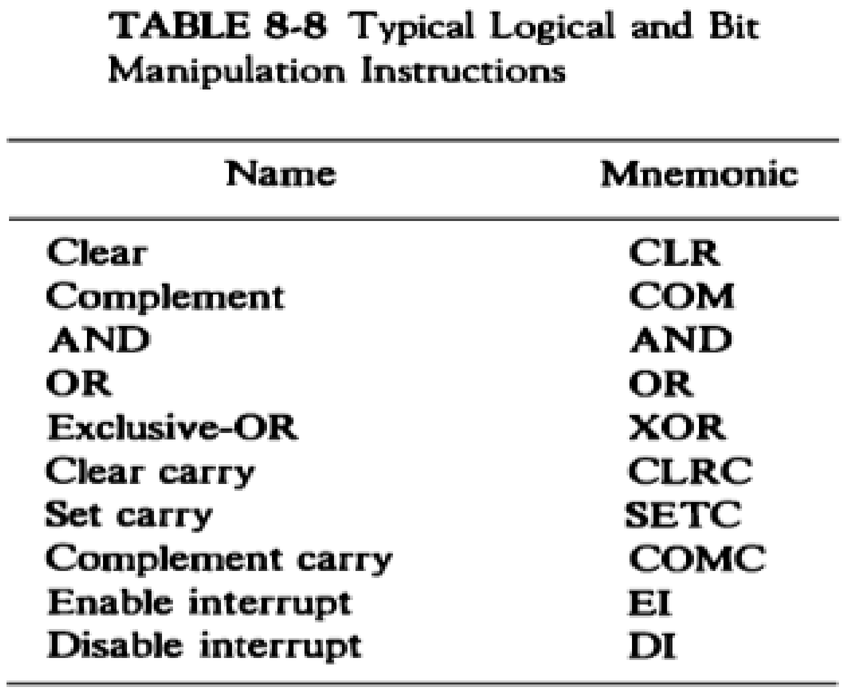


TABLE 8-8 Typical Logical and Bit Manipulation Instructions

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

Data Manipulation Instructions

2. Logical and Bit Manipulation Instructions

The **clear** instruction causes the specified operand to be replaced by 0's.

The **complement** instruction produces the 1's complement by inverting all the bits of the operand.

The **AND**, **OR**, and **XOR** instructions produce the corresponding logical operations on individual bits of the operands. Although they perform Boolean operations, when used in computer instructions, the logical instructions should be considered as performing bit manipulation operations.

A few other bit manipulation instructions are included in Table 8-8. 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.

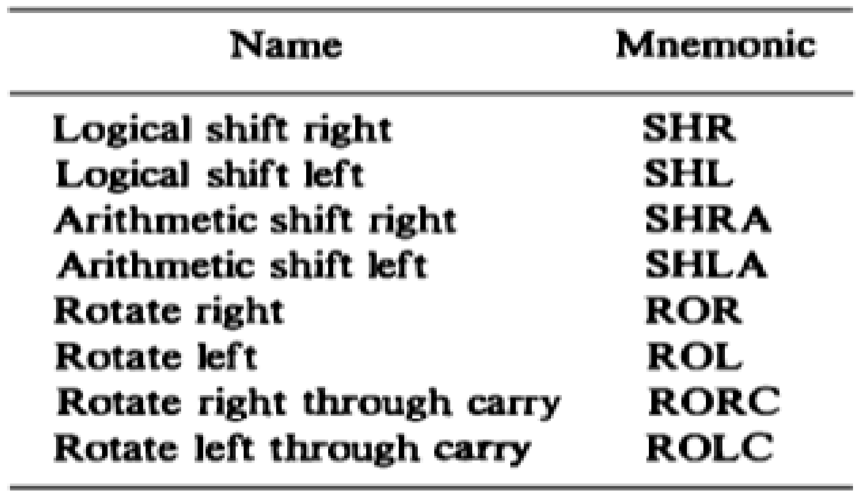
Data Manipulation Instructions

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 right or to the left.

TABLE 8-9 Typical Shift Instructions

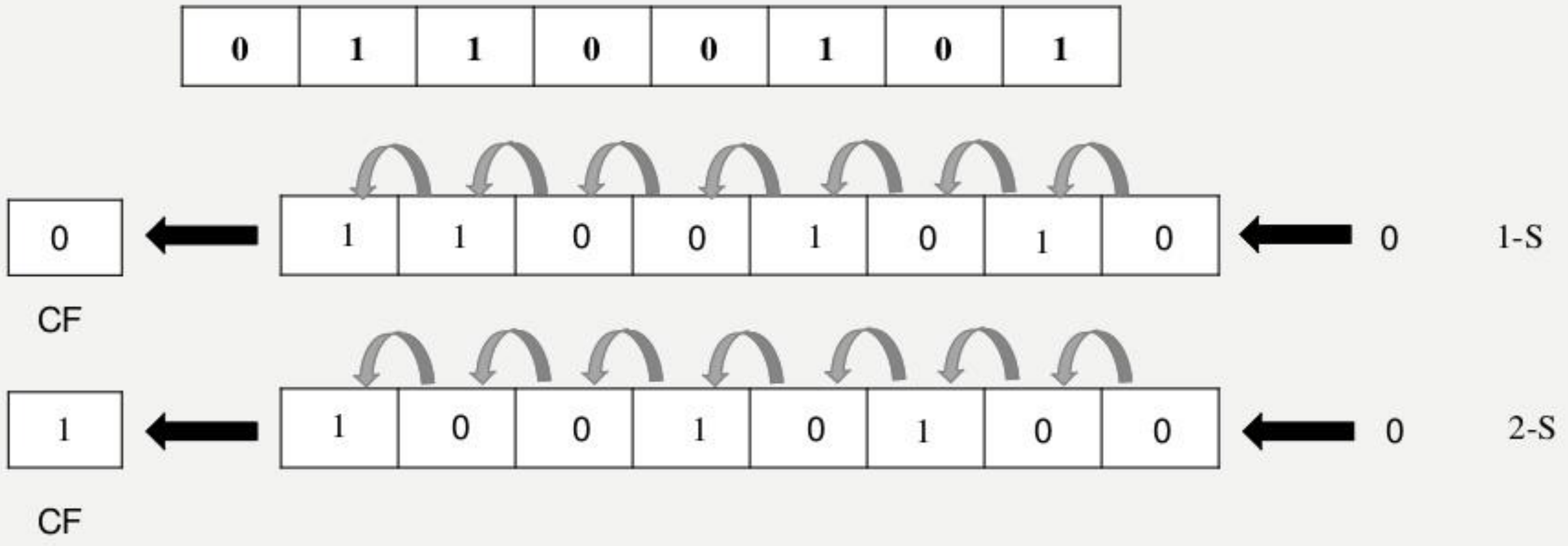


Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

Data Manipulation Instructions

SHIFT INSTRUCTION: SHL AND SAL

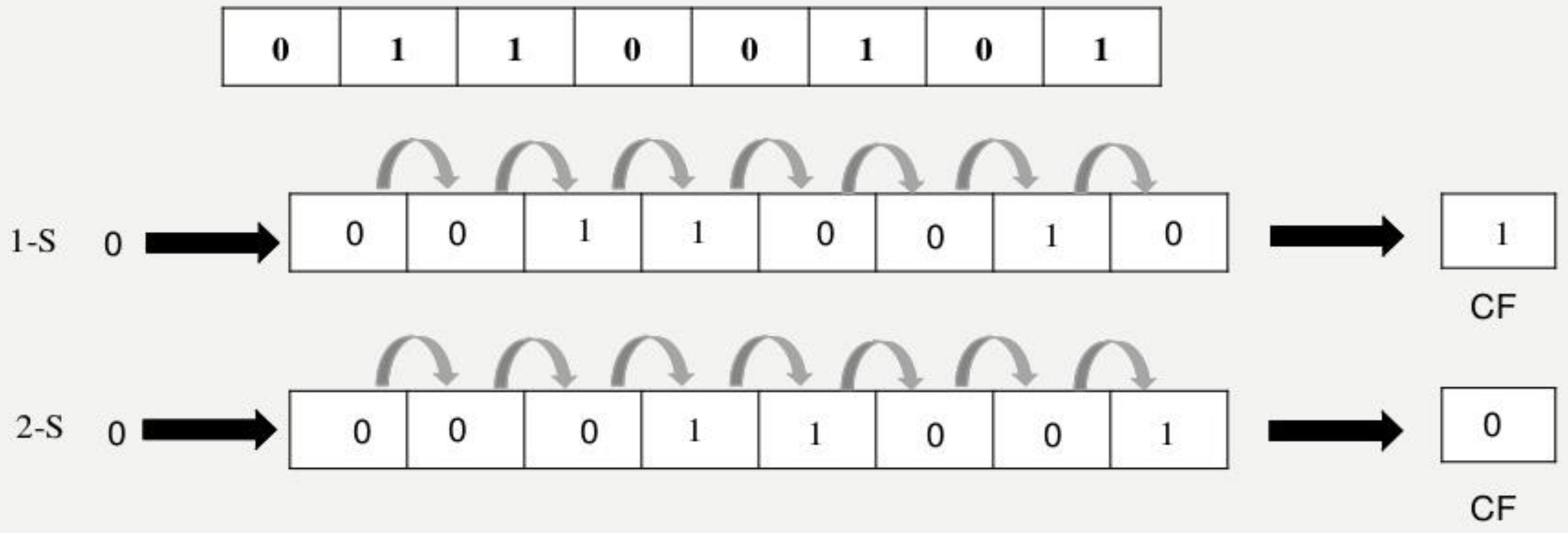
- The SHL (shift left) instruction shifts the bits in the destination to the left.
- A 0 is shifted into the right most bit position and the msb is shifted into CF.



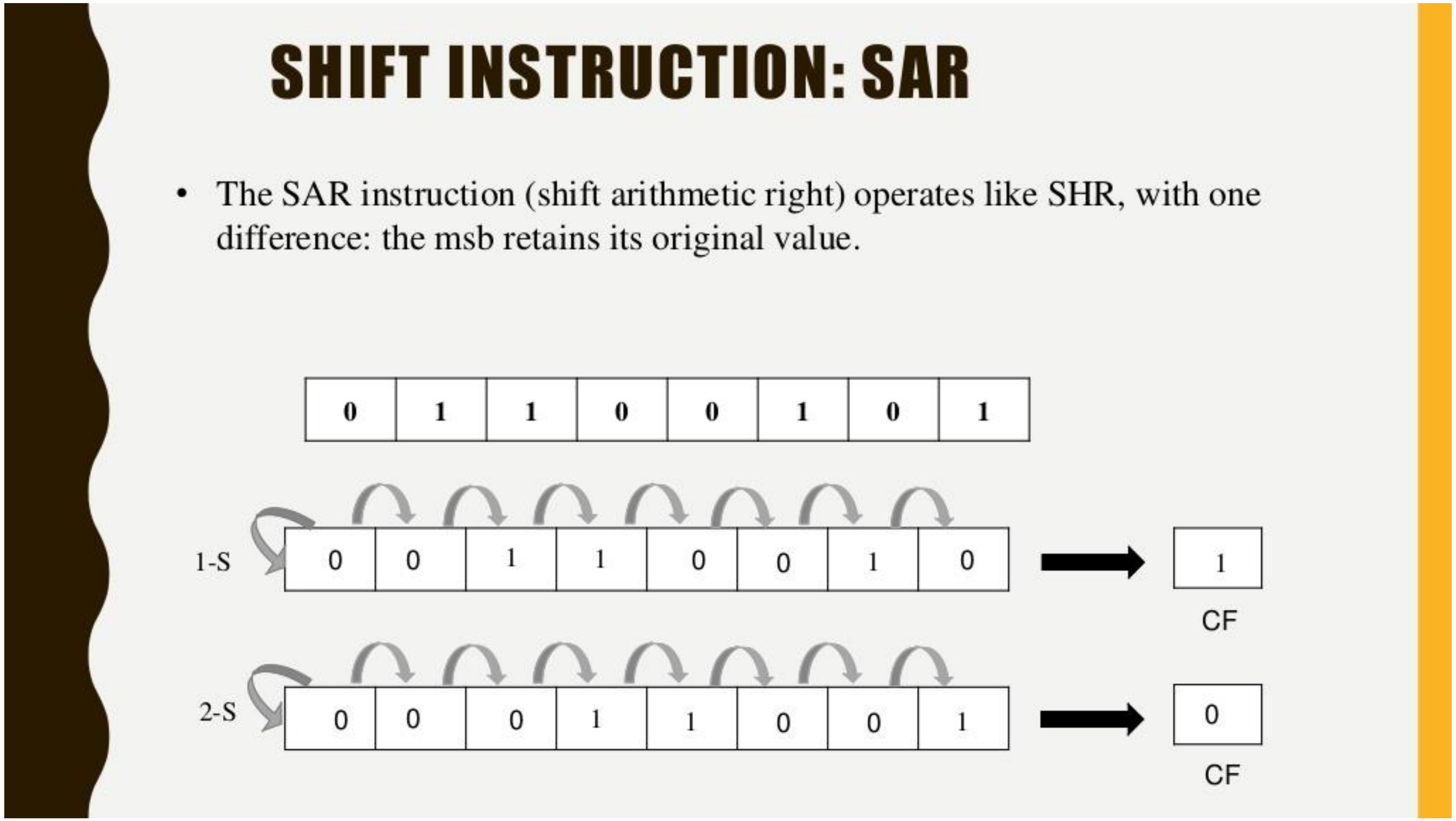
Data Manipulation Instructions

SHIFT INSTRUCTION: SHR

- The instruction SHR(shift right) performs right shift on the destination operand.
- A 0 is shifted into the msb position ,and the rightmost bit is shifted into CF.



Data Manipulation Instructions



SHIFT INSTRUCTION: SAR

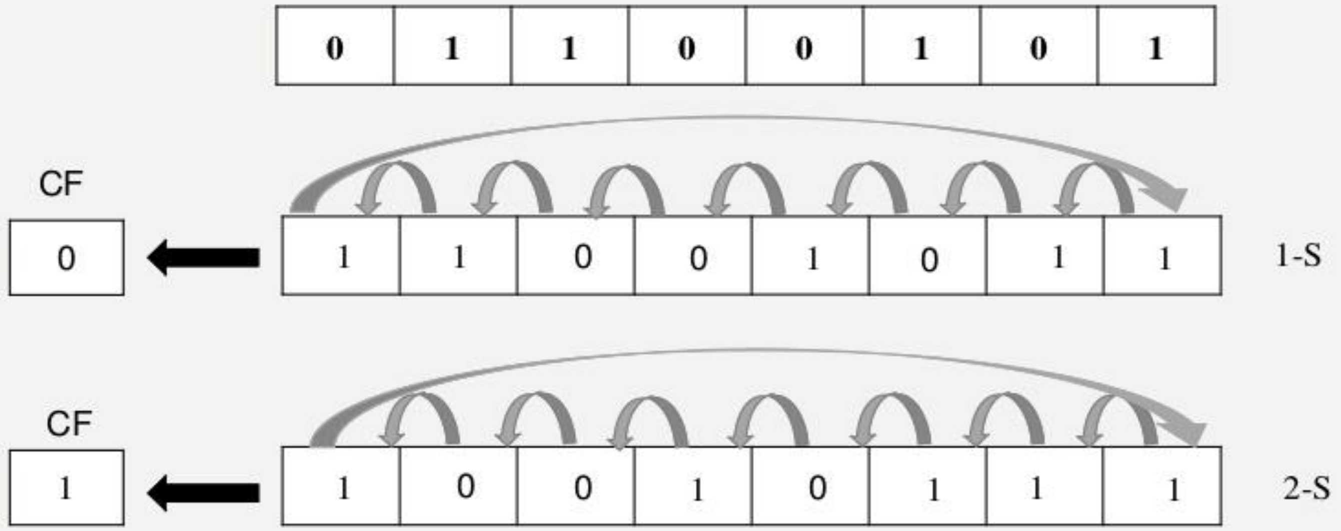
- The SAR instruction (shift arithmetic right) operates like SHR, with one difference: the msb retains its original value.

0 1 1 0 0 1 0 1
1-S
0 0 1 1 0 0 1 0
1
CF
2-S
0 0 0 1 1 0 0 1
0
CF

Data Manipulation Instructions

ROTATE INSTRUCTION: ROL

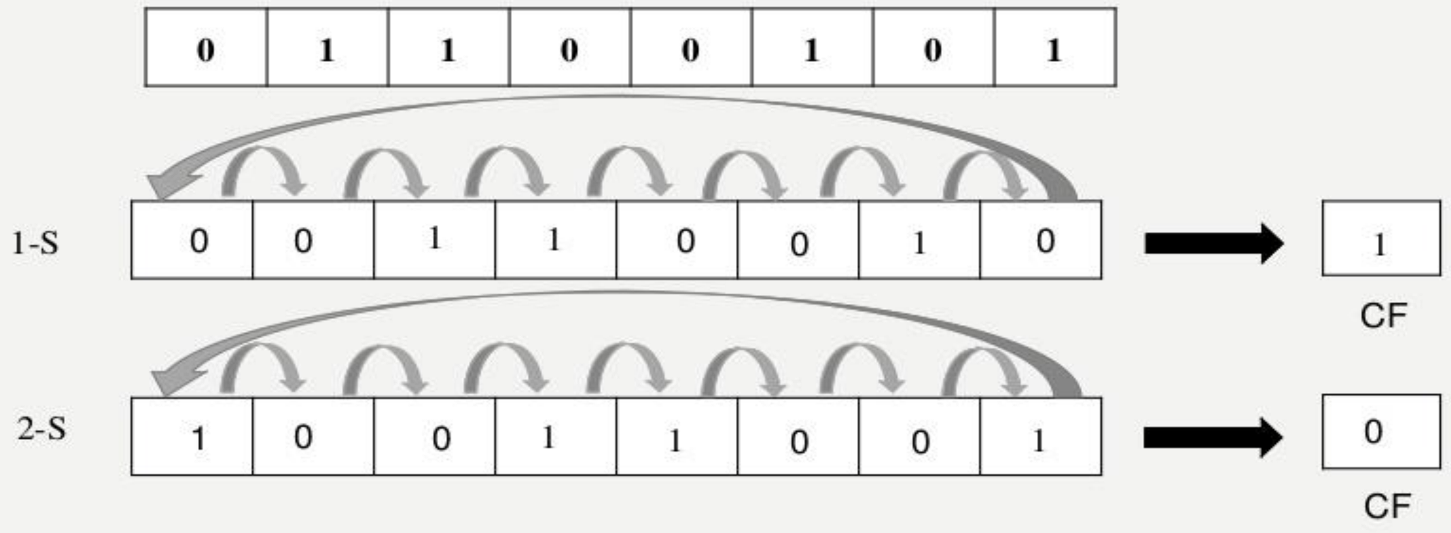
- The instruction ROL (rotate left)shifts bits to the left .The msb is shifted into the rightmost bit . The CF also gets the bit shifted out of the msb .



Data Manipulation Instructions

ROTATE INSTRUCTION: ROR

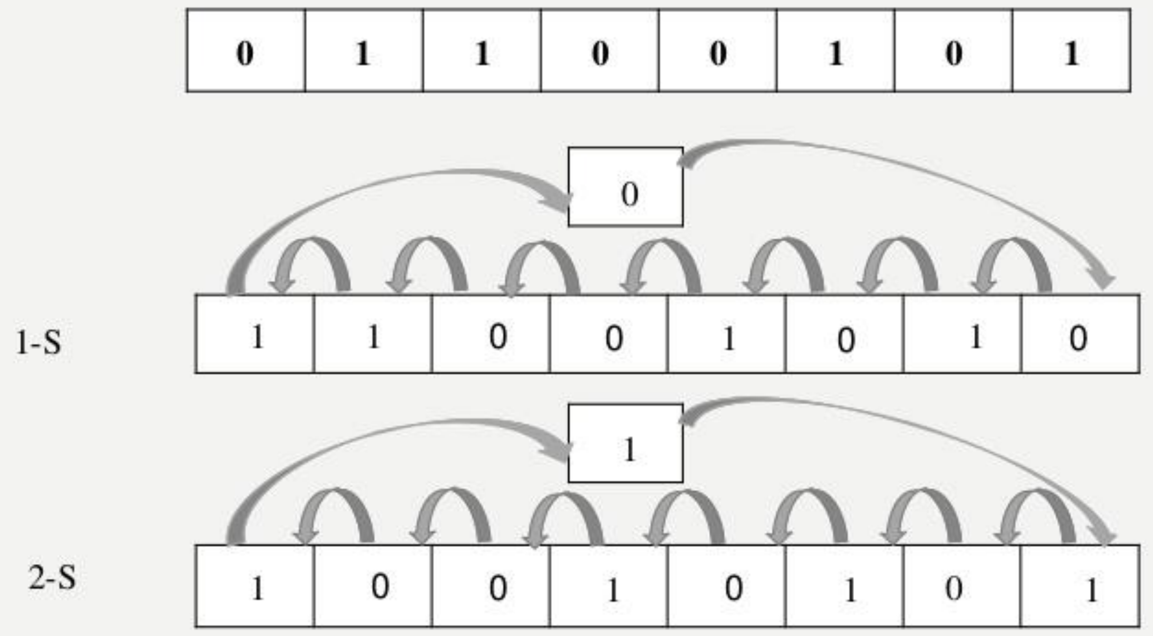
- The instruction ROR (rotate right) works just like ROL , except that the bits are rotate to the right .The rightmost bit is shifted into the msb , and also into The CF.



Data Manipulation Instructions

ROTATE INSTRUCTION: RCL

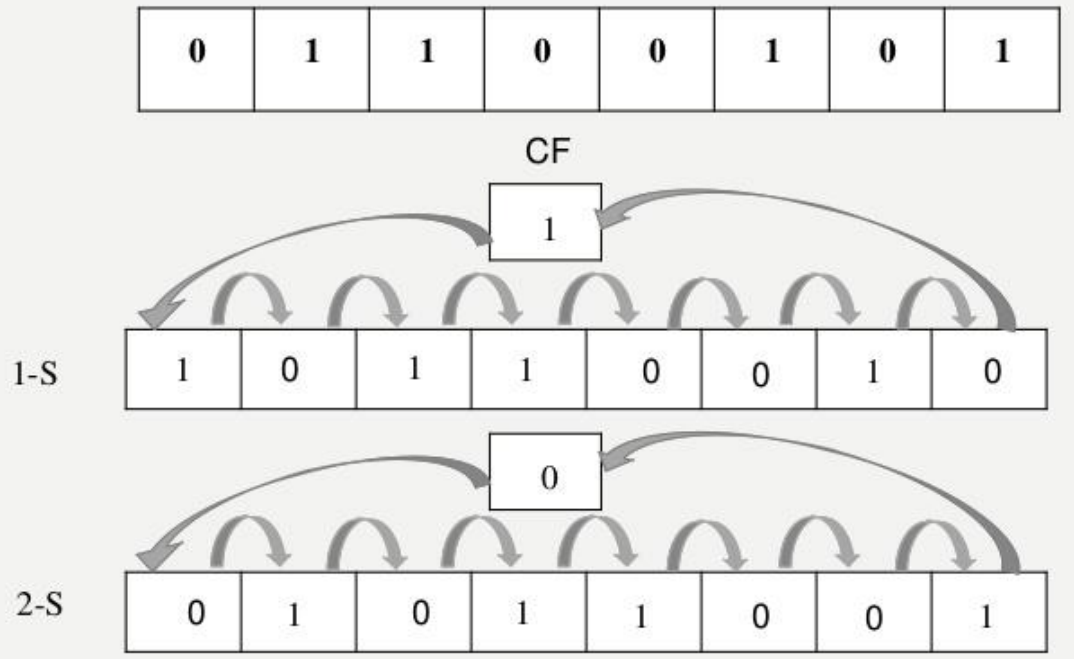
- The instruction **RCL** (Rotate through carry left) shifts the bits of the destination to the left. The msb is shifted
into the CF, and the previous value of CF is shifted into the rightmost bit.



Data Manipulation Instructions

ROTATE INSTRUCTION: RCR

The instruction RCR(Rotate through carry right) works just like RCL, except that the bits are rotated to the right.



Program Control Instructions

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.

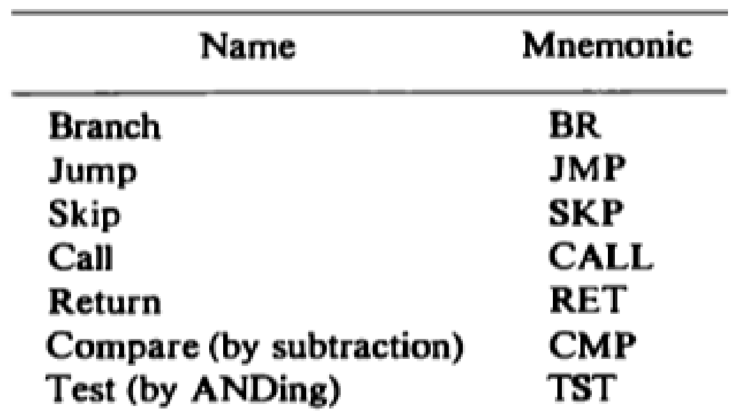
After the execution of a data transfer or data manipulation instruction, control returns to the fetch cycle with the program counter containing the address of the instruction next in sequence.

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.

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** 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 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.**

Program Control Instructions

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.

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

Status Bit Conditions

Status bits are also called **condition-code bits** or **flag bits**. Figure below 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.

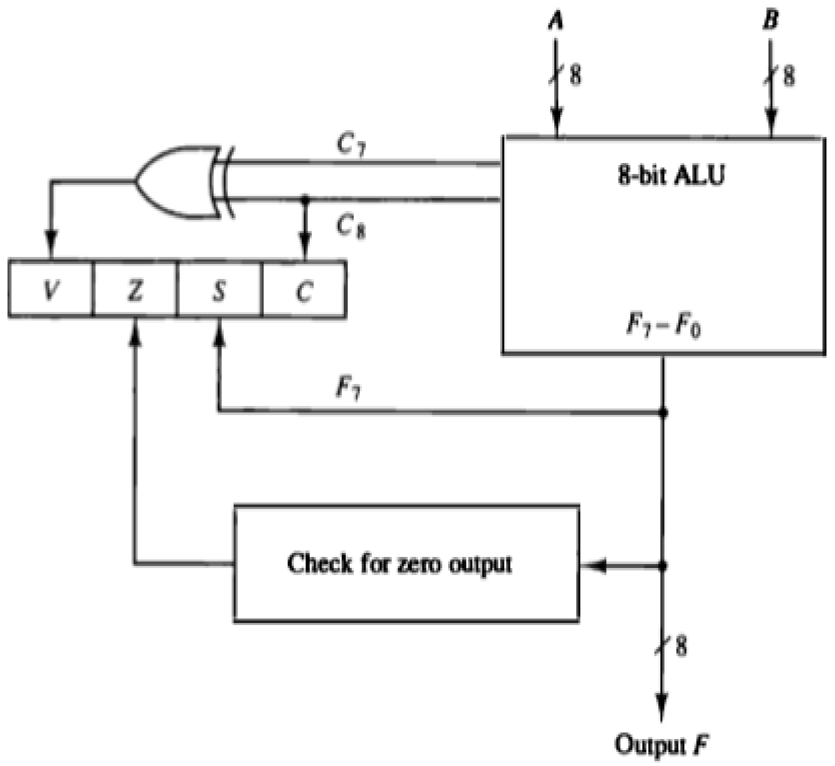
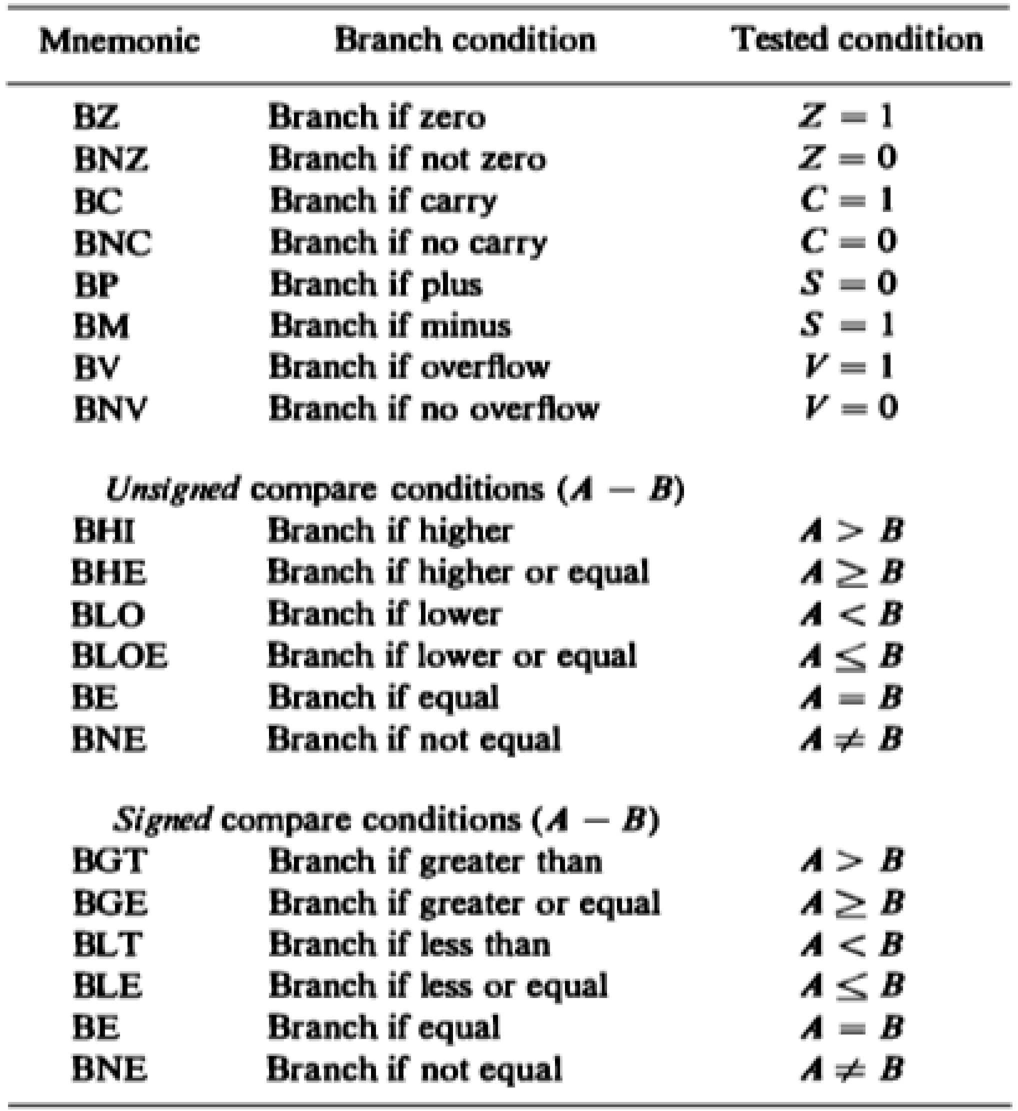


Figure 8-8 Status register bits.

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 C8 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, 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 O's. !tis 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



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	$A \neq B$

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. Thus BC is Branch on Carry, and BNC is Branch on No Carry. If the stated condition is true, program control is transferred to the address specified by the instruction. If not, control continues with the instruction that follows. The conditional instructions can be associated also with the jump, skip, call, or return type of program control instructions.

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***.

Subroutine Call and Return

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.

RISC & CISC

A computer with a large number of instructions is classified as a **complex instruction set computer**, abbreviated **CISC**.

In the early 1980s, a number of computer designers recommended that 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

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. A relatively large number of registers in the processor unit.

RISC & CISC

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