CHAPTER 5

CENTRAL PROCESSING UNIT

Rolisha Sthapit

CONTENTS

- **General Register Organization:** Control Word, Stack Organization, Instruction formats, addressing modes.
- **Data Transfer and Manipulation:** Data transfer instructions, Data manipulation instructions, Arithmetic instructions, Logical and Bit Manipulation Instructions, Shift Instructions.
- **Program Control**: Status Bit Conditions, Conditional Branch Instructions, Subroutine Call and Return, Program Interrupt, Types of Interrupt.

INTRODUCTION

- The part of the computer that performs the bulk of data processing operation is called central processing unit (CPU) which consists of ALU, control unit and register array.
- CPU performs a variety of functions dictated by the type of instructions that are incorporated in the computer.
- 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 (CU) unit supervises the transfer of information among the registers and instructs the ALU as to which operation to be performed.

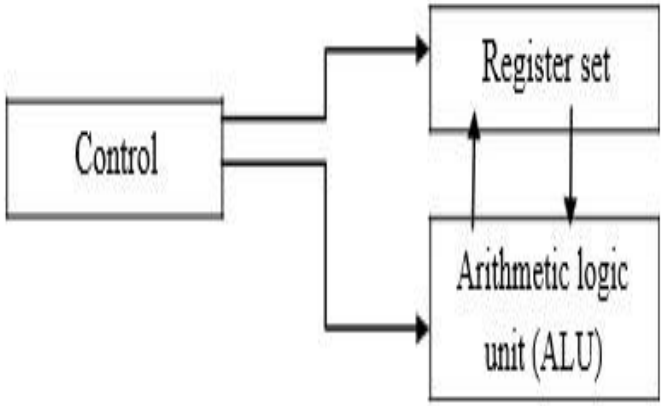
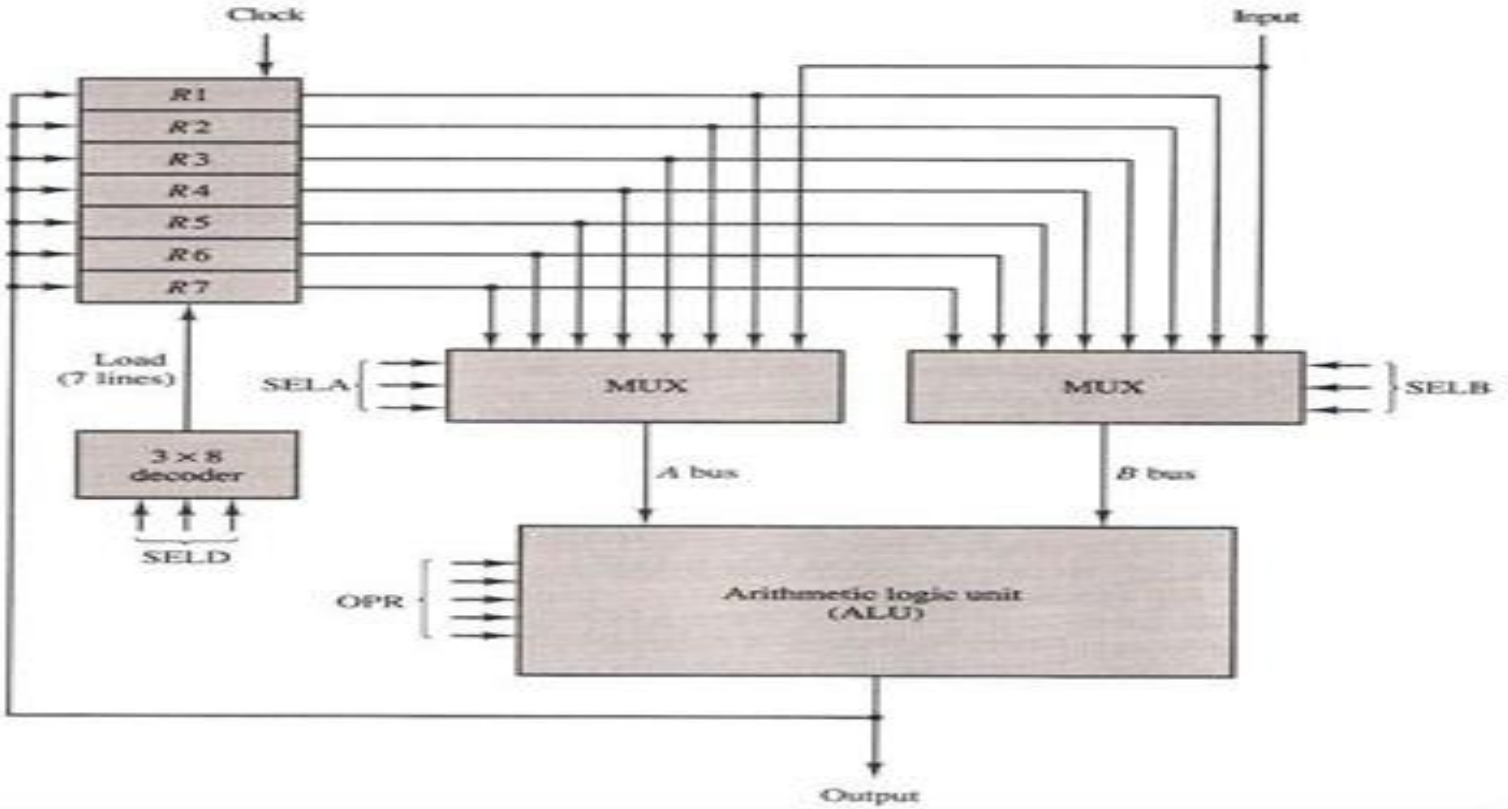


Fig: Major components of CPU

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.

- A bus organization of seven CPU registers is shown below:



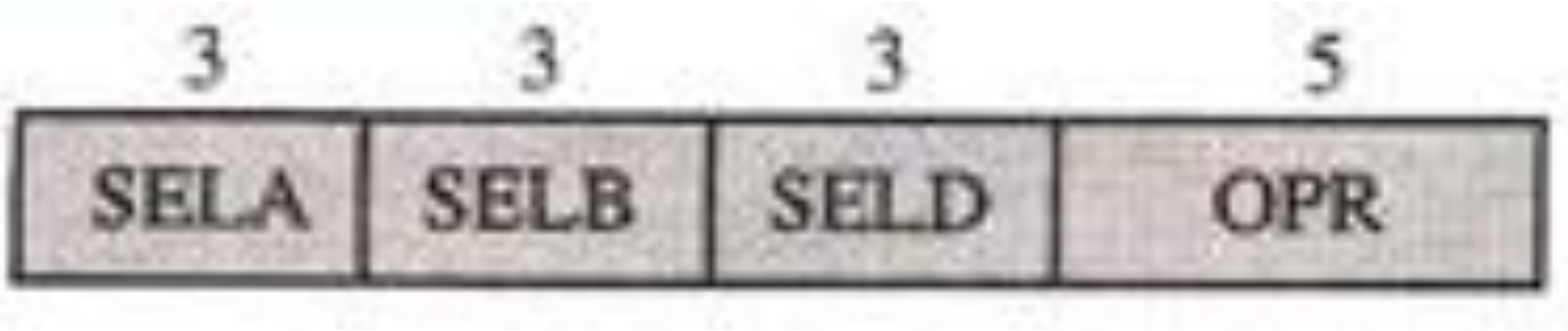
(a) Block diagram (register organization)

- 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.

- All registers are connected to two multiplexers (MUX) that select the registers for bus A and bus B. Registers selected by multiplexers are sent to ALU. Another selector (OPR) connected to ALU selects the operation for the ALU. Output produced by ALU is stored in some register and this destination register for storing the result is activated by the destination decoder (SELD).
- Example: R1 ← R2 + R3
  - MUX selector (SELA): BUS A ← R2
  - MUX selector (SELB): BUS B ← R3
  - ALU operation selector (OPR): ALU to ADD
  - Decoder destination selector (SELD): R1 ← Out Bus

Control word:

Combination of all selection bits of a processing unit is called control word. Control Word for above CPU is as below:



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

- The 14 bit control word when applied to the selection inputs specify a particular microoperation. Encoding of the register selection fields and ALU operations is given below:

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

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

Example: R1 ← R2 - R3

This microoperation specifies R2 for A input of the ALU, R3 for the B input of the ALU, R1 for the destination register and ALU operation to subtract A-B. Binary control word for this microoperation statement is:

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

Examples of different microoperations are shown below:

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

Register stack

- It is the collection of finite number of registers. Stack pointer (SP) points to the register that is currently at the top of 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.

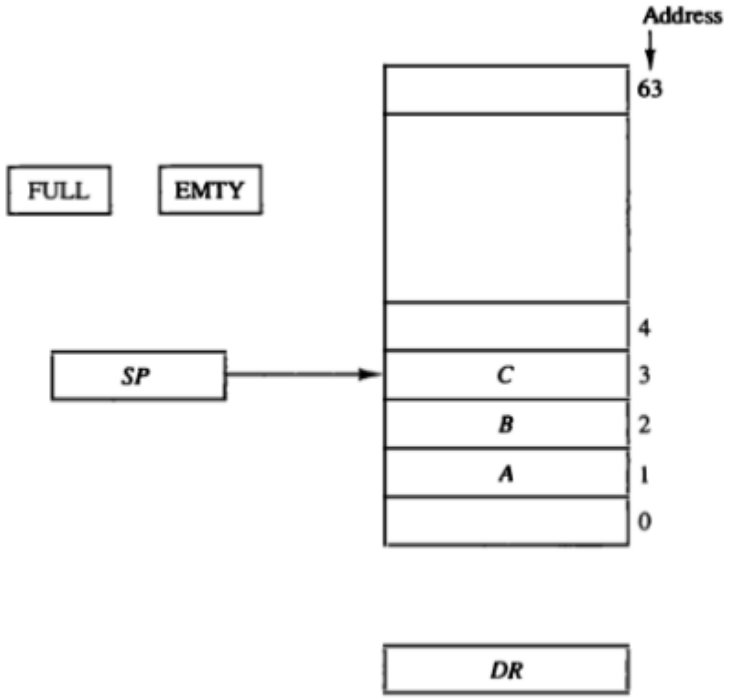


Figure 8-3 Block diagram of a 64-word stack.

- 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
- 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

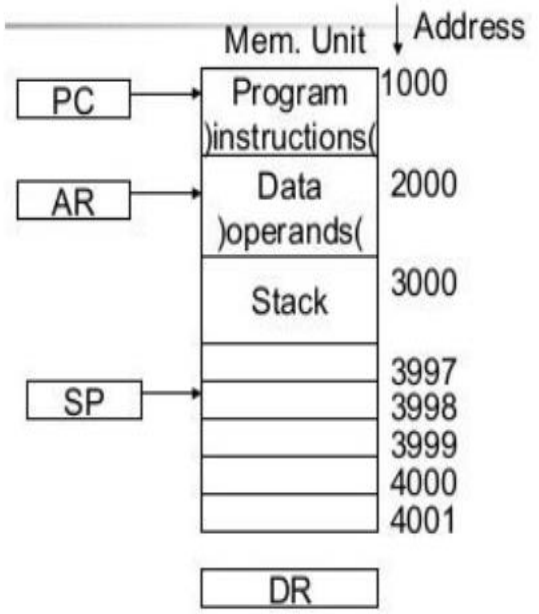
Diagram shows 64-word register stack. 6-bit address SP points stack top. Currently 3 items are placed in the stack: A, B and C do that content of SP is now 3 (actually 000011). 1-bit registers FULL and EMTY are set to 1 when the stack is full and empty respectively. DR is data register that holds the binary data to be written into or read out of the stack.

/* Initially, SP = 0, EMPTY = 1(true), FULL = 0(false) */

Push operation	**Pop operation**
SP ← SP + 1	DR ← M [SP]
M [SP] ← DR	SP ← SP – 1
If (SP = 0) then (FULL ← 1)	If (SP = 0) then (EMPTY ← 1)
EMPTY ← 0	FULL ← 0

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 the stack.
- 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.



Computer memory with program, data & stack segments

- 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.

PC: used during fetch phase to read an instruction.
AR: used during execute phase to read an operand.
SP: used to push or pop items into or from the stack.

Here, initial value of SP is 4001 and stack grows with *decreasing addresses*. First item is stored at 4000, second at 3999 and last address that can be used is 3000. No provisions are available for stack limit checks.

PUSH:	**POP:**
SP ← SP - 1	DR ← M[SP]
M[SP] ← DR	SP ← SP + 1

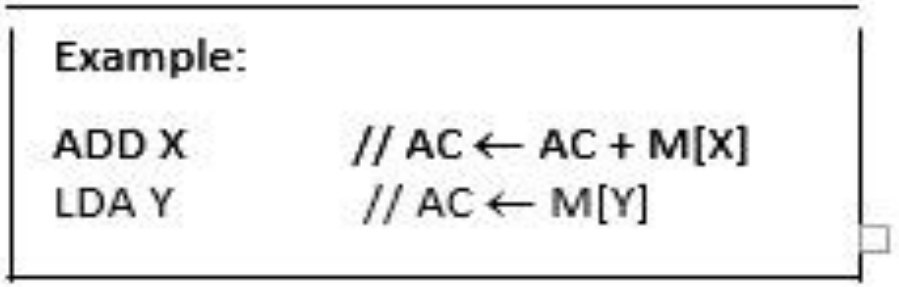
CPU Organizations/Processor Organization

There are three types of CPU organization based on the instruction format:

1. Single accumulator organization
2. General register organization
3. Stack Organization

1.Single accumulator organization:

- In this type of organization all the operations are performed with an implied accumulator register.
- Basic computer is the good example of single accumulator organization.
- The instruction of this type of organization has an address field



where X and Y is the address of the operand

2. General register organization:

- When a large number of processor 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 transfer, but also while performing various microoperations. Hence, it is necessary to provide a common unit that can perform all the arithmetic, logic and shift microoperations in the processor.
- In this type of organization the instruction has two or three address field

Example:

```
ADD R1, R2, R3      // R1 ← R2 + R3
ADD R1, R2          // R1 ← R1 + R2
MOV R1, R2          // R1 ← R2
ADD R1, X           // R1 ← R1 + M[X]
```

3. Stack organization:

- Last-in, first-out (LIFO) mechanism.
- A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved.
- In this type of organization of CPU, all the operations are performed with stack.
- The PUSH and POP instruction only need address field. The operation-type instructions do not need address field.
- This ADD instruction in the stack organization performs addition of two top of the stack element and stores the result in the top of the stack. First pops two operands from the top of the stack; adds them and stores the result in the top of the stack.

```
Example:
PUSH X          // TOS ← M[X]
ADD             // TOS = TOP(S) + TOP(S)
```

Instruction Formats

Most common field found in register are:

a) **Mode bit:** It specifies the way the operand or the effective address is determined.

b) **Op-code field:** It specifies the operation to be performed.

c) **Address field:** It designates a memory address or a processor register.

The number of address fields in the instruction format depends on the internal organization of CPU. On the basis of no. of address field we can categorize the instruction as below:

1. Three-Address Instruction:

- Computer with three address instruction can use each address field to specify either processor register or memory operand.
- Advantage –it minimize the size of program
- Disadvantage –binary coded instruction requires too many bits to specify three address fields

E.g. ADD R1, A, B / R1 ← M[A]+M[B]

Program to evaluate the following arithmetic statement **X = (A+B) * (C+D)** using three address fields 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

2.Two-Address Instruction:

- Computer with two address instruction can use each address field to specify either processer register or memory operand
- Advantage –it minimize the size of instruction
- Disadvantage –the size of program is relatively larger

Program to evaluate the following arithmetic statement **X = (A+B)*(C+D)** using two address field instruction

```
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.

3.One-Address Instruction:

- Execution of one address field instruction use an implied accumulator register for all data manipulation
- Advantage –relatively small instruction size
- Disadvantage –relatively large program size

Program to evaluate the following arithmetic statement **X = (A+B)*(C+D)** using one address field instruction

```
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.

4.Zero-Address Instruction:

- This type of instruction is used in stack organization computer. There is no address field in this type of instruction except PUSH and POP.
- Advantage –small instruction size
- Disadvantages –large the program size

Program to evaluate the following arithmetic statement **X = (A+B)*(C+D**) using zero address field instruction

```
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
```

TOS- Top of Stack

X=A-B+C+(D/E)

Three Address	Two Address	One Address	Zero Address

DIV R1,D,E
ADD R2,R1,C
ADD R3,R2,A
SUB X,R3,B
	MOV R1,D
	DIV R1,E
	ADD R1,C
	ADD R1,A
	SUB R1,B
	MOV X,R1
		LOAD D
		DIV E
		ADD C
		ADD A
		SUB B
		STORE X
			PUSH D
			PUSH E
			DIV
			PUSH C
			ADD
			PUSH A
			ADD
			PUSH B
			SUB
			POP X

Y=A+B(CD+EF-G/H)

Three address	Two address	One address	Zero address

DIV R1,G,H
MUL R2,E,F
MUL R3,C,D
ADD R4,R2,R3
SUB Y,R4,R1
MUL Y,Y,B
ADD Y,Y,A
	MOV R1,G
	DIV R1,H
	MOV R2,E
	MUL R2,F
	MOV R3,C
	MUL R3,D
	ADD R2,R3
	SUB R2,R1
	MUL R2,B
	ADD R2,A
	MOV Y,R2
		LOAD G
		DIV H
		STORE T
		LOAD E
		MUL F
		SUB T
		STORE T
		LOAD C
		MUL D
		ADD T
		MUL B
		ADD A
		STORE Y
			PUSH G
			PUSH H
			DIV
			PUSH E
			PUSH F
			MUL
			SUB
			PUSH C
			PUSH D
			MUL
			ADD
			PUSH B
			MUL
			PUSH A
			ADD
			POP Y

Addressing Modes

The method of calculating or finding the effective address of the operand in the instruction is called addressing mode. The way operands (data) are chosen during program execution depends on the addressing mode of the instruction. So, *addressing mode* specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.

Why Addressing modes?

- To give programming versatility to the user (by providing facilities as: pointers to memory, counters for loop control, indexing of data and program relocation)
- To use the bits in the address field of the instruction efficiently

Types of Addressing Modes

The various addressing modes are:

i. Implied Mode
ii. Immediate Mode
iii. Register Mode
iv. Register Indirect Mode
v. Auto increment or Auto decrement Mode
vi. Direct Address Mode
vii. Indirect Address Mode
viii. Relative Address Mode
ix. Indexed Addressing Mode
x. Base Register Addressing Mode

i. Implied Mode:

- In this type of addressing mode, operands specified implicitly in the definition of instruction.
- All the register reference instructions that use an accumulator and zero-address instruction in a stack organized computer are implied mode instruction.
- No need to specify the address in the instruction.
- E.g. CMA (complement accumulator), CLA, CME, etc.

ii. **Immediate Mode:**

- In this addressing mode, the operand is specified in the instruction itself i.e. there is no any address field to represent the operand
- Immediate mode instructions are useful for initializing register to a constant value.
- Instead of specifying the address of the operand, operand itself is specified in the instruction.

E.g. LDA #NBR / AC← NBR

iii. Register Mode:

- In this type of addressing mode, the operands are in the register which is within the CPU .
- Faster to acquire an operand than the memory addressing

AC ← R1

iv. Register Indirect Mode:

- In this addressing mode, the content of register present in the instruction specifies the effective address of operand.
- The advantage of this addressing mode is that the address field of the instruction uses fewer bits to select a register.
- EA = content of R

  AC ← M[R1]

v. Auto Increment or Auto decrement mode:

- In auto increment mode, the content of CPU register is incremented by 1, which gives the effective address of the operand in memory.

  AC ← M[R1], R1 ← R1 + 1

- In auto decrement mode, the content of CPU register is decremented by 1, which gives the effective address of the operand in memory.

  AC ← M[R1 - 1]

vi. **Direct Address Mode**

In this addressing mode, the address field of an instruction gives the effective address of operand.

AC ← M[ADR]

vii. Indirect Address Mode

In this addressing mode, the address field of the instruction gives the address of effective address.

AC ← M[M[ADR]]

viii. Relative Address Mode:

In this addressing mode, the content of program counter is added to the address part of the instruction which gives the effective address of the operand.

AC← M[PC + ADR]

ix. Indexed Addressing Mode:

In this addressing mode, the content of index register is added to the address field of the instruction which gives the effective address of operand.

AC← M[ADR + XR]

x. Base Register Addressing Mode:

In this addressing mode, the content of the base register is added to the address part of the instruction which gives the effective address of the operand.

$AC \leftarrow M[ADR + BR]$

Numerical Example

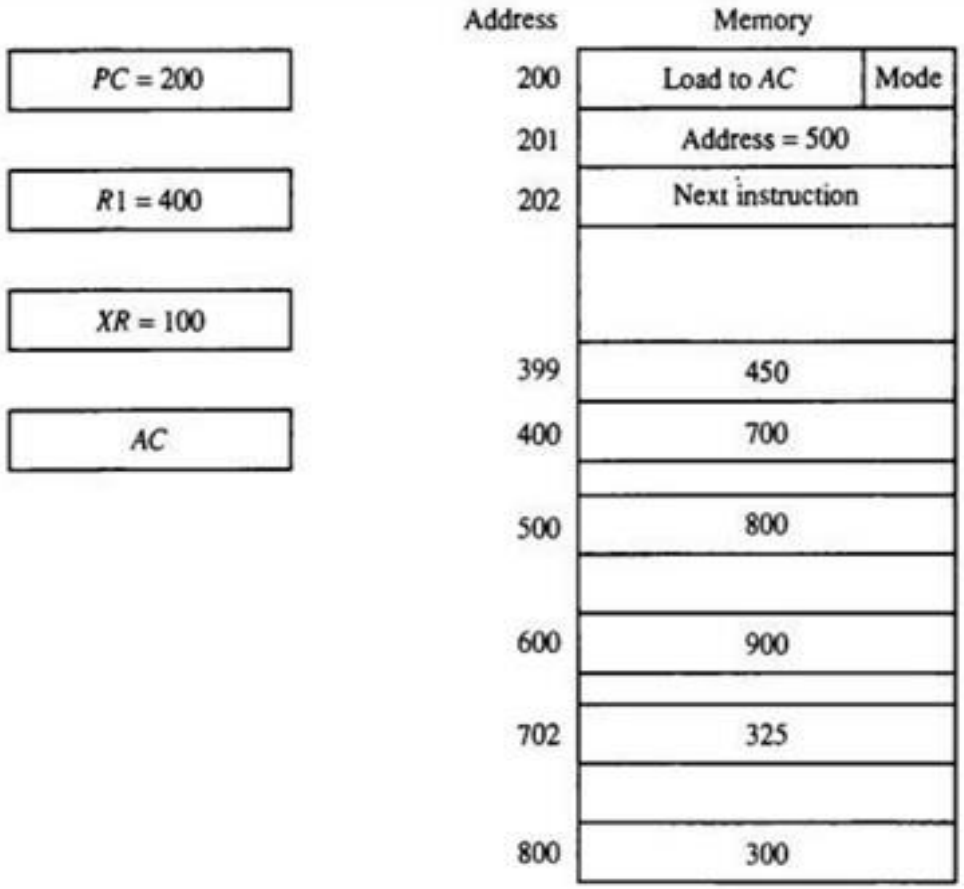


Fig: Numerical example for addressing modes

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

Fig: Content of AC after each addressing modes

PC = 150

R1 = 500

XR=107

BR=106

Address	Memory
400	Load to AC
401	Address= 600
407	Next Instruction
501	101
550	400
600	800
706	900
708	350
1007	850

Addressing Modes	Effective Address	Content of AC
Immediate	401	600
Register	-	500
Register Indirect	500	-
Auto Increment	500	-,500+1=501
Auto Decrement	499	-
Direct	600	800
Indirect	800	-
Relative Addressing	1007	850
Indexed Register	707	-
Base Register	706	900

14. Consider the following figure :

PC =300

R1= 500

What is the value in AC if the instruction is

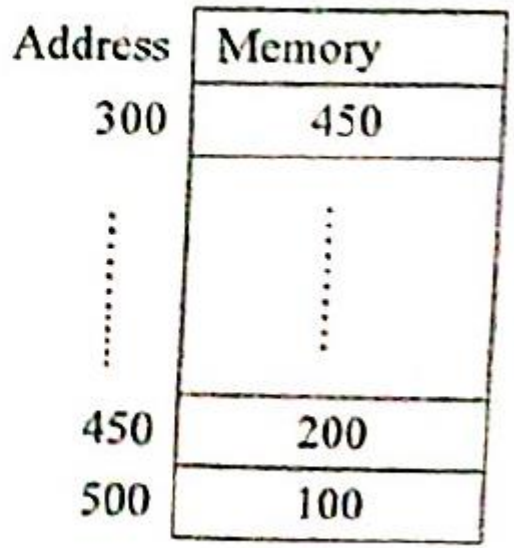
LDA 300 if the modes are:

i. Direct addressing 450

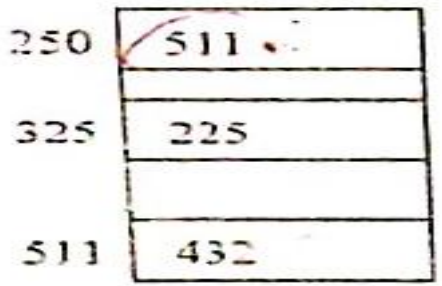
ii. immediate 300

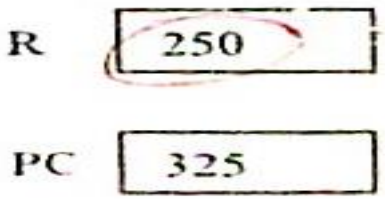
iii. Indirect addressing 200

iv. Register indirect if LDA (R1) is used 100



14. Consider the following memory and the instruction LDA 250:





Write the value loaded into AC when the addressing mode is

a) Indirect b) Register Indirect c) Immediate d) Direct

Solution:

a) Indirect : 432

b)Register Indirect: 511

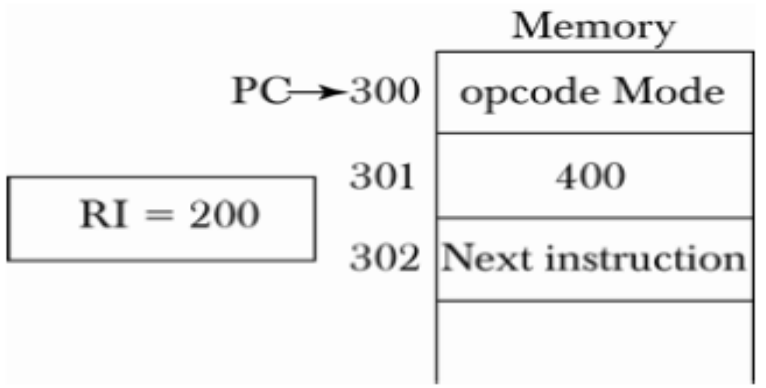
c) Immediate : 250

d) Direct : 511

8-18. An instruction is stored at location 300 with its address field at location 301. The address field has the value 400. A processor register R1 contains the number 200. Evaluate the effective address if the addressing mode of the instruction is (a) direct; (b) immediate; (c) relative; (d) register indirect; (e) index with R1 as the index register.

Effective address

(a) Direct: 400
(b) Immediate: 301
(c) Relative: 302 + 400 = 702
(d) Reg. Indirect: 200
(e) Indexed: 200 + 400 = 600

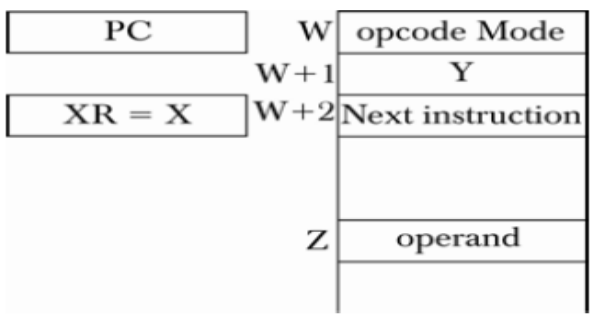


8-14. A two-word instruction is stored in memory at an address designated by the symbol W. The address field of the instruction (stored at W + 1) is designated by the symbol Y. The operand used during the execution of the instruction is stored at an address symbolized by Z. An index register contains the value X. State how Z is calculated from the other addresses if the addressing mode of the instruction is

a. direct
b. indirect
c. relative
d. indexed

Z = Effective address

(a) Direct: Z = Y
(b) Indirect: Z = M[Y]
(c) Relative: Z = Y + W + 2
(d) Indexed: Z = Y + X



Data Transfer and Manipulation

- Computers give extensive set of instructions to give the user the flexibility to carryout 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 in the instruction set are not very different from one computer to another although binary encodings and symbol name (operation) may vary. So, most computer instructions can be classified into 3 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.
- 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 causes transfer of data from one location to another without modifying the binary information content. The most common transfers are:

- between memory and processor registers
- between processor registers and I/O
- between processor register themselves

Example: Load, store, exchange, move, push, pop, etc

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

Load: denotes transfer from memory to registers (usually AC)
Store: denotes transfer from a processor registers into memory
Move: denotes transfer between registers, between memory words or memory & registers.
Exchange: swaps information between two registers or register and a memory word.
Input & Output: transfer data among registers and I/O terminals.
Push & Pop: transfer data among registers and memory stack.

- Instructions described above are often associated with the variety of addressing modes. Assembly language uses special character to designate the addressing mode. E.g. # sign placed before the operand to recognize the immediate mode. (Some other assembly languages modify the mnemonics symbol to denote various addressing modes, e.g. for load immediate: LDI). Example: consider load to accumulator instruction when used with 8 different addressing modes:

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

Table: Recommended assembly language conventions for load instruction in different addressing modes

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

Example: increment, decrement, add, subtract, add with carry, subtract with borrow, 2's complement.

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.

- Typical arithmetic instructions are listed below:

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

- Increment (decrement) instr. adds 1 to (subtracts 1 from) the register or memory word value.
- Add, subtract, multiply and divide instructions may operate on different data types (fixed-point or floating-point, binary or decimal).

Logical and bit manipulation instructions :

- Logical instructions perform binary operations on strings of bits stored in registers and are useful for manipulating individual or group of bits representing binary coded information. Logical instructions each bit of the operand separately and treat it as a Boolean variable.

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

- Clear instr. causes specified operand to be replaced by 0's.
- Complement instr. produces the 1's complement.
- AND, OR and XOR instructions produce the corresponding logical operations on individual bits of the operands.

- 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.
- 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.

Shift instructions

- Instructions to shift the content of an operand are quite useful and are often provided in several variations (bit shifted at the end of word determine the variation of shift). Shift instructions may specify 3 different shifts:

❑Logical shifts

❑Arithmetic shifts

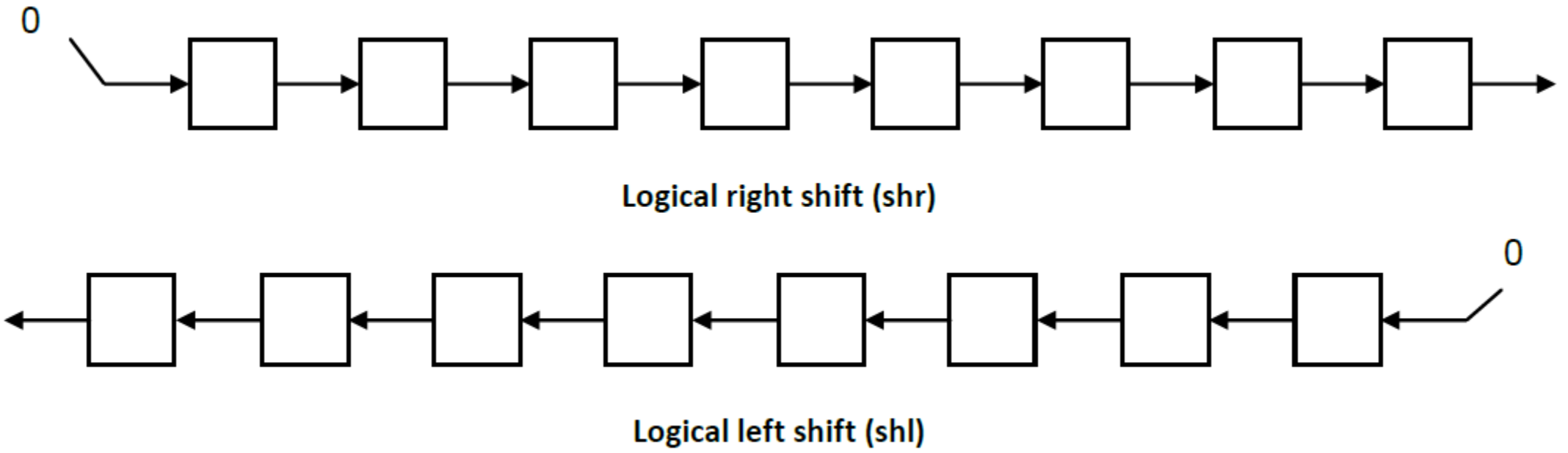
❑Rotate-type operations

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

- Table lists 4 types of shift instructions.
- Logical shift inserts 0 at the end position
- Arithmetic shift left inserts 0 at the end (identical to logical left shift) and arithmetic shift right leave the sign bit unchanged (should preserve the sign).
- Rotate instructions produce a circular shift.
- Rotate left through carry instruction transfers carry bit to right and so is for rotate shift right.

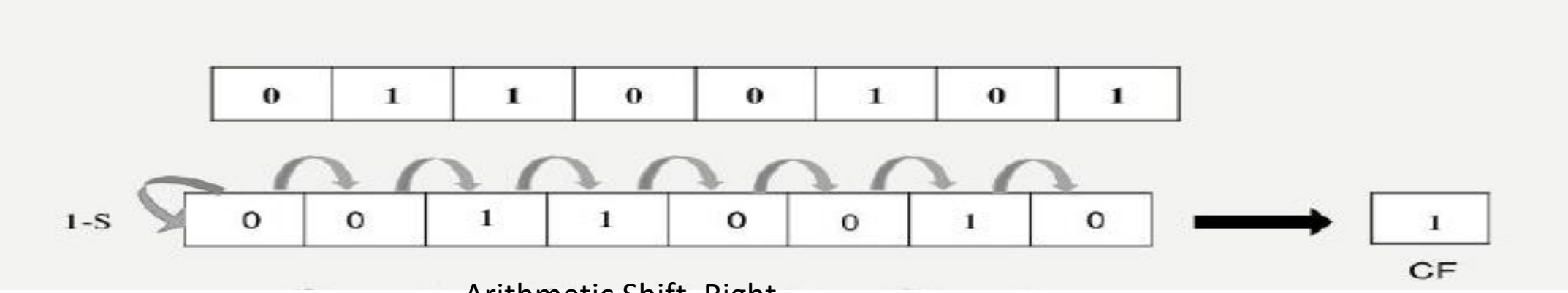
Logical Shift

A logical shift is the one that transfers 0 through the serial input.

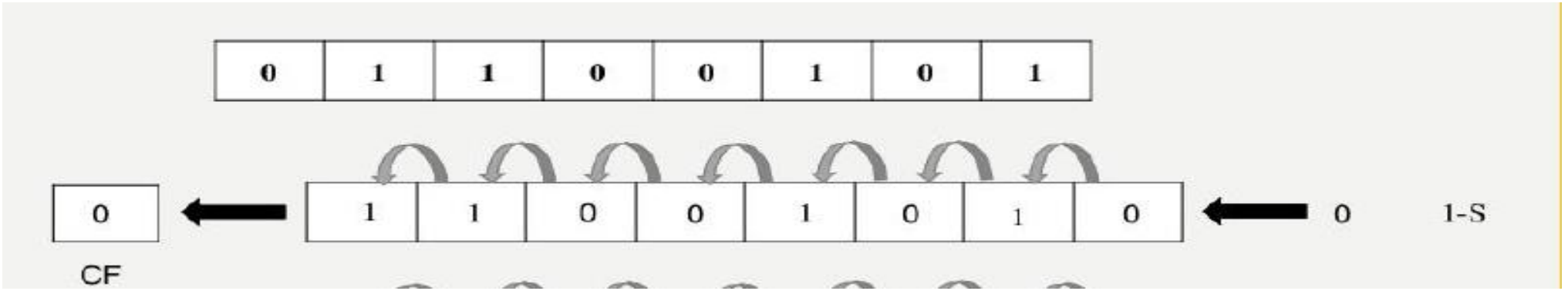


Arithmetic Shift

An arithmetic shift is meant for signed binary numbers (integer). An arithmetic left shift multiplies a signed number by two and an arithmetic right shift divides a signed number by two.



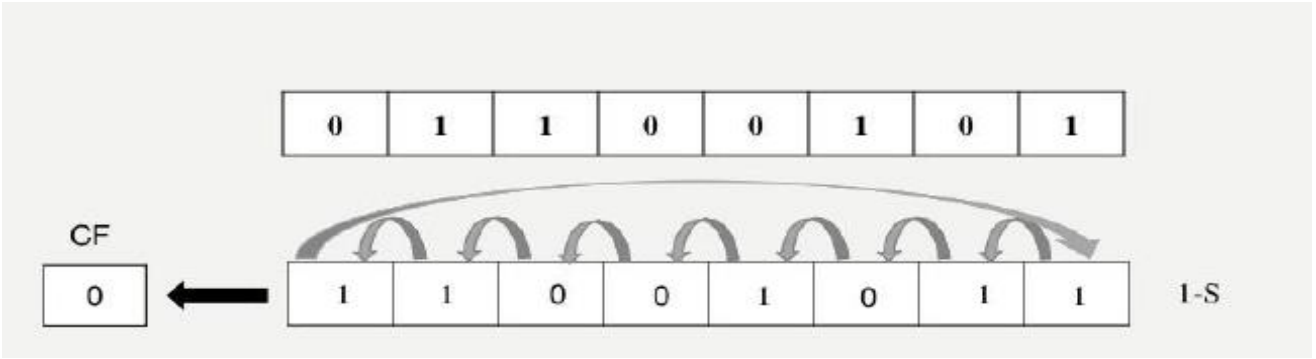
Arithmetic Shift Right



Arithmetic Shift Left

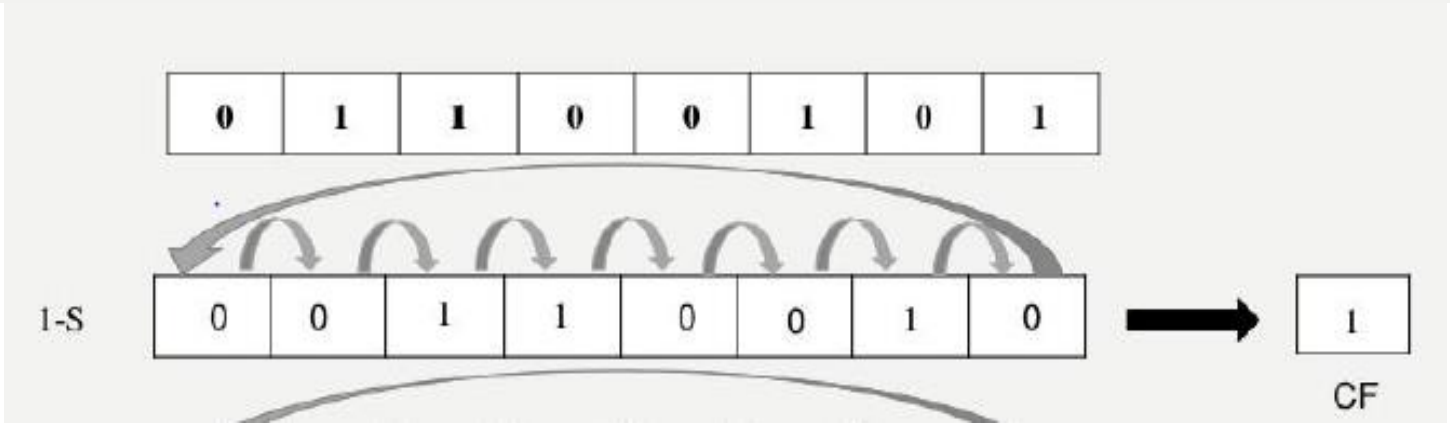
Rotate Instruction

- 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 .



Rotate Left

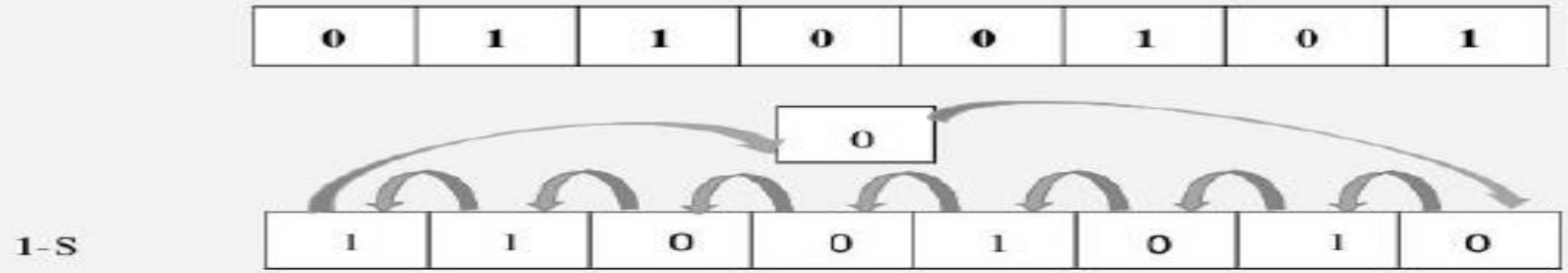
- The instruction ROR (rotate right) works just like ROL , except that the bits are rotate to the right .The rightmost bit is shifted into the msb , and also into The CF.



Rotate Right

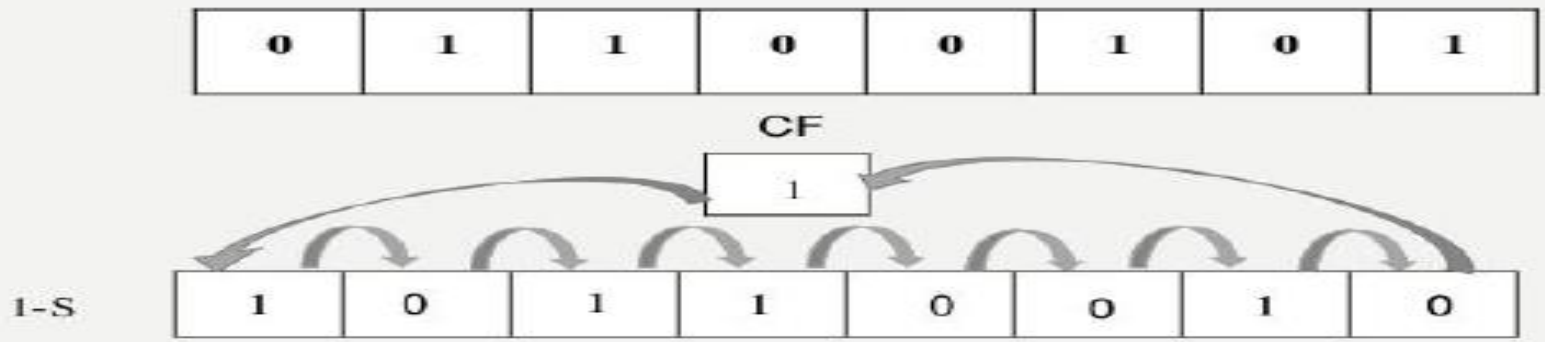
ROTATE INSTRUCTION: RCL

- 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.



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.

- 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.

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

- 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.

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. 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.

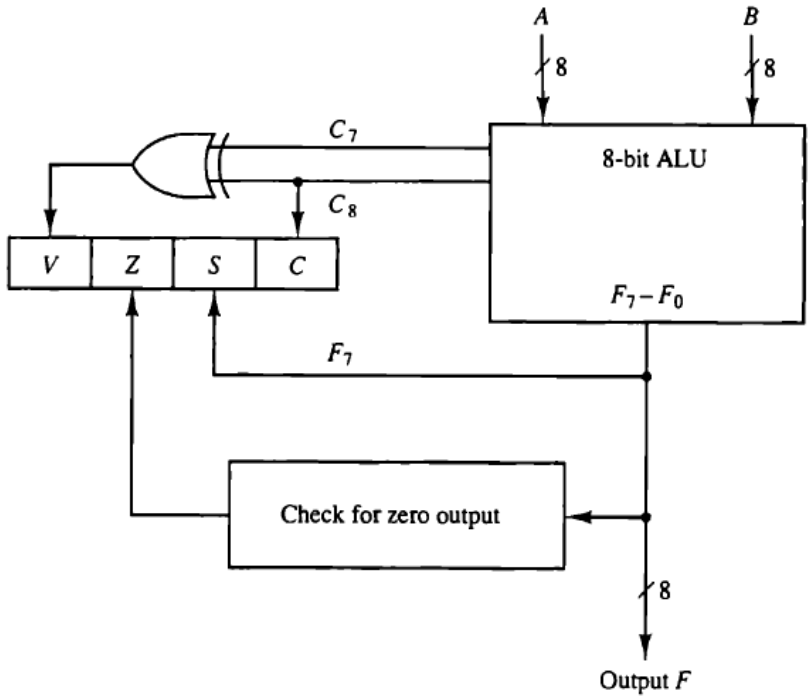


Figure 8-8 Status register bits.

Conditional 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	$A \neq B$

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.**

- A call subroutine instruction consists of an operation code together with an address that specifies the beginning of the subroutine. The instruction is executed by performing two operations:

(1) the address of the next instruction available in the program counter (the return address) is stored in a temporary location so the subroutine knows where to return, and

(2) control is transferred to the beginning of the subroutine.

- The last instruction of every subroutine, commonly called return from subroutine, transfers the return address from the temporary location into the program counter. This results in a transfer of program control to the instruction whose address was originally stored in the temporary location.

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

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

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

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

Program Interrupt

- The concept of program interrupt is to handle a variety of problems that arise out of normal program sequence.
- Program interrupt refers to the transfer of program control from a currently running program to another service program as a result of an external or internal generated request. Control returns to the original program after the service program is executed.
- After a program has been interrupted and the service routine has been executed, the CPU must return to exactly the same state that it was when the interrupt occurred.

- The state of the CPU at the end of the execute cycle (when the interrupt is recognised) is determined from:
    1. The content of program counter
    2. The content of all processor registers
    3. The content of certain status conditions

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.

Types of Interrupt

- 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

1. External Interrupt

- 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.
- 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.

2. Internal Interrupt

- 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.

3. Software Interrupt

- 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 that stores the old CPU state and brings in a new PSW that belongs to the supervisor mode. The calling program must pass information to the operating system in order to specify the particular task required.

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.

RISC and 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.

RISC and CISC characteristics

RISC (reduced instruction set computer) 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:

- Relatively few instructions
- Relatively few addressing modes
- Memory access limited to load and store instructions
- All operations done within the registers of the CPU
- Fixed-length, easily decoded instruction format
- Single-cycle instruction execution
- The control unit is hardwired rather than micro programmed
- Relatively large number of registers in the processor unit
- Efficient instruction pipeline

The main concept of RISC is to reduce execution time by simplifying the instruction set of the computer. Example: MIPS(**Microprocessor without Interlocked Pipeline Stages**), ARM (Advanced RISC Machine)

- **CISC (complex instruction set computer) characteristics**

➢A large number of instructions - typically from 100 to 250 instructions

➢Some instructions that perform specialized tasks and are used infrequently

➢A large variety of addressing modes – typically from 5 to 20 different modes

➢Variable-length instruction format

➢Uses memory to load and store instruction and operand as well

➢Instructions that manipulate operands in memory

- Principle of CISC:

To provide single machine instruction for each statement that is written in a higher level language. One reason to provide a complex instruction set is to simplify the compilation and improve the overall computer performance.

Example: IBM computer, desktop computer, Digital equipment corporation.

Assignment: Write any 8 differences between RISC and CISC.

CISC and RISC architecture Microcontrollers:

CISC Processors	***RISC Processors***
Complex Instruction Set Computer	Reduced Instruction Set Computer
When an MCU supports many addressing modes for arithmetic and logical instructions and for memory accesses and data transfer instructions, the MCU is said to of CISC architecture.	When an MCU has an instruction set that supports one or two addressing modes for arithmetic and logical instructions and few for memory accesses and data transfer instructions, the MCU is said to of RISC architecture
Large number of complex instructions	Small number of instructions
Instructions are of variable number of bytes	Instructions are of fixed number of bytes
Instructions take varying amounts of time for execution	Instructions take fixed amount of time for execution