COMPUTER ARCHITECTURE

UNIT II

Central Processing Unit : General Register Organization – Stack Organization – Instruction Formats – Addressing Modes – Data transfer and manipulation – Program Control.

COMPUTER ARCHITECTURE
UNIT – II
CENTRAL PROCESSING UNIT

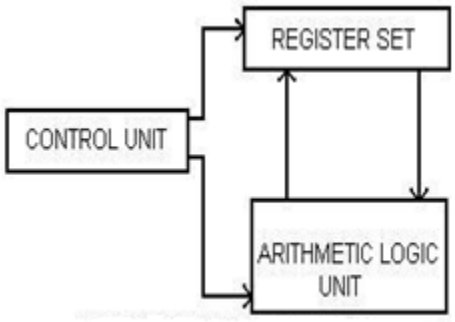
1. Introduction

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. 1-1.

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.

The CPU performs a variety of functions dictated by the type of instructions that are incorporated in the computer.

Fig 1-1: Major components of CPU



2. 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 microoperations. Hence it is necessary to provide a common unit that can perform all the Arithmetic, Logic, and shift microoperations in the processor.

A bus organization for seven CPU registers is shown in Fig. 2-1(a). The output of each register is connected to two multiplexers (MUX) to form the two buses A and B. The selection lines in each multiplexer select one register or the input data for the particular bus. The A and B buses form the inputs to a common arithmetic logic unit (ALU). The operation selected in the ALU determines the arithmetic or logic microoperation that is to be performed.

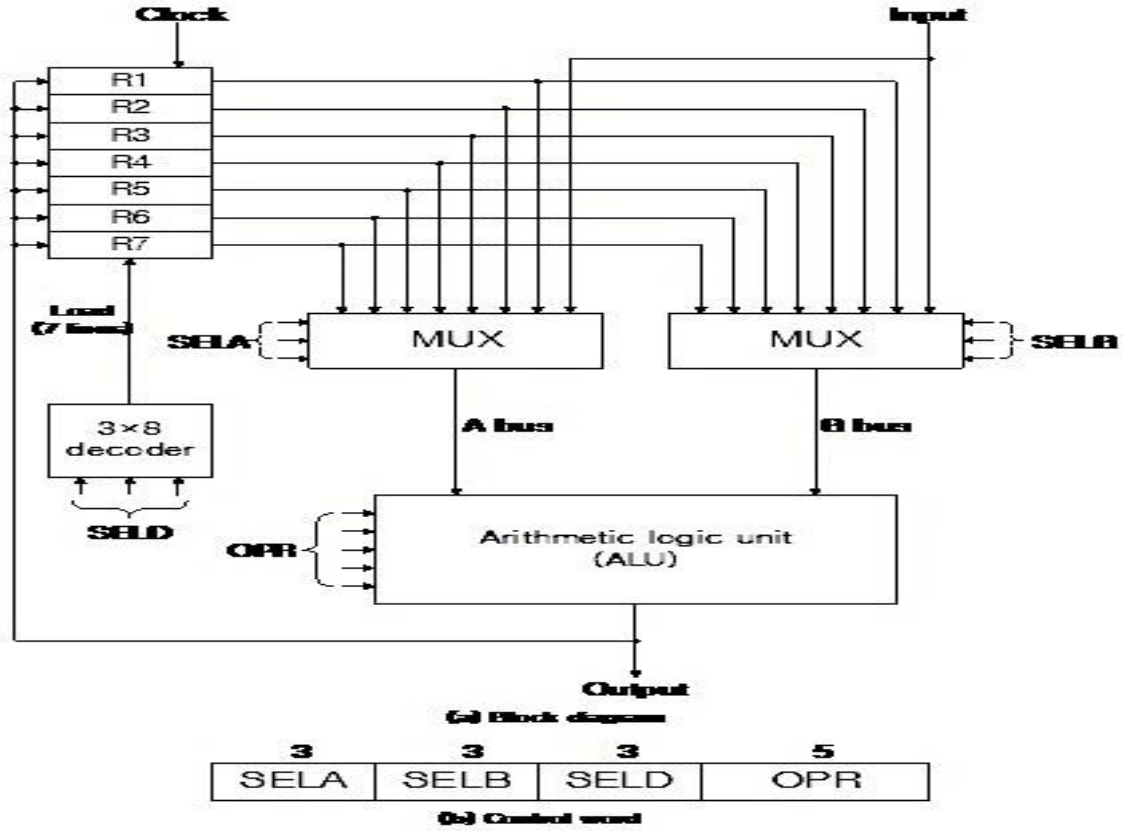
The result of the microoperation is available for output data and also goes into the inputs of all the registers. The register that receives the information from the output bus is selected by a decoder. The decoder activates one of the register load inputs, thus providing a transfer path between the data in the output bus and the inputs of the selected destination register.

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

To perform the operation

$$R1 \leftarrow R2 + R3$$

Fig 2-1: Register set with common ALU



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

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

Control word

There are 14 binary selection inputs in the unit and their combined value specifies a control word. The 14-bit control word is defined in Fig. 2-1(b). It consists of four fields. Three fields contain three bits each, and one field has five bits. The three bits of SELA select a source register for the A input of the ALU. The three bits of SELB select a register for the B input of the ALU. The three bits SELD select a destination register using the decoder and its seven load outputs. The five bits of OPR select one of the operations in the ALU. The 14- bit control word when applied to the selection inputs specify a particular microoperation

The encoding of the register selections is specified in Table 2-1. The 3-bit binary code listed in the first column of the table specifies the binary code for each of the three fields. The register selected by fields SELA, SELB, and SELD is the one whose decimal number is equivalent to the binary number in the code. When SELA or SELB is 000, the corresponding multiplexer selects the external input data. When SELD = 000, no destination register is selected but the contents of the output bus are available in the external output.

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

Table 2-1: 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 2-2: 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

Examples of Microoperations

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

$R1 \leftarrow R2 - R3$

Specifies R2 for the A input of the ALU, R3 for the B input of the ALU, R1 for the destination register, and an ALU operation to subtract A – B. The binary control word for the subtract microoperationis 010 011 001 00101 and is obtained as follows:

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

The increment and transfer microoperations do not use the B input of the ALU. For these cases, the B field is marked with a dash. We assign 000 to any unused field when formulating the binary control word, although any other binary number may be used. To place the content of a register into the output terminals we place the content of the register into the A input of the ALU, but none of the registers are selected to accept the data. The ALU operation TSFA places the data from the register, through the ALU, into the output terminals. The direct transfer from input to output is accomplished with a control word of all 0' s (making the B field 000). A register can be cleared to 0 with an exclusive-OR operation. This is because $x \oplus x = 0$.

TABLE 2-3: Examples of Microoperations for the CPU

Symbolic Designation					
Microoperation	**SELA**	**SELB**	**SELD**	**OPR**	**Control Word**
R 1 <--R2 - R3	R2	R3	R1	SUB	010 011 001 00101
R 1 <--R2 + R3	R2	R3	R1	ADD	010 011 001 00010
R4 <--R4 V R5	R4	R5	R4	OR	100 101 100 01010
R6 <--R6 + 1	R6	-	R6	INCA	110 000 110 00001
R7 <--R1	R1	-	R7	TSFA	001 000 111 00000
Output <--R2	R2	-	None	TSFA	010 000 000 00000
Output <--lnput	Input	-	None	TSFA	000 000 000 00000
R4 <--sh1 R4	R4	-	R4	SHLA	100 000 100 11000
R5 <--0 [R5←R5 ⊕R5]	R5	R5	R5	XOR	101 101 101 01100

3. Stack Organization

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

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

The stack in digital computers is essentially a memory unit with an address register that can count only (after initial value is loaded into it). The register that holds the address for the stack is called a stack pointer (SP) because its value always points at the top item in the stack.

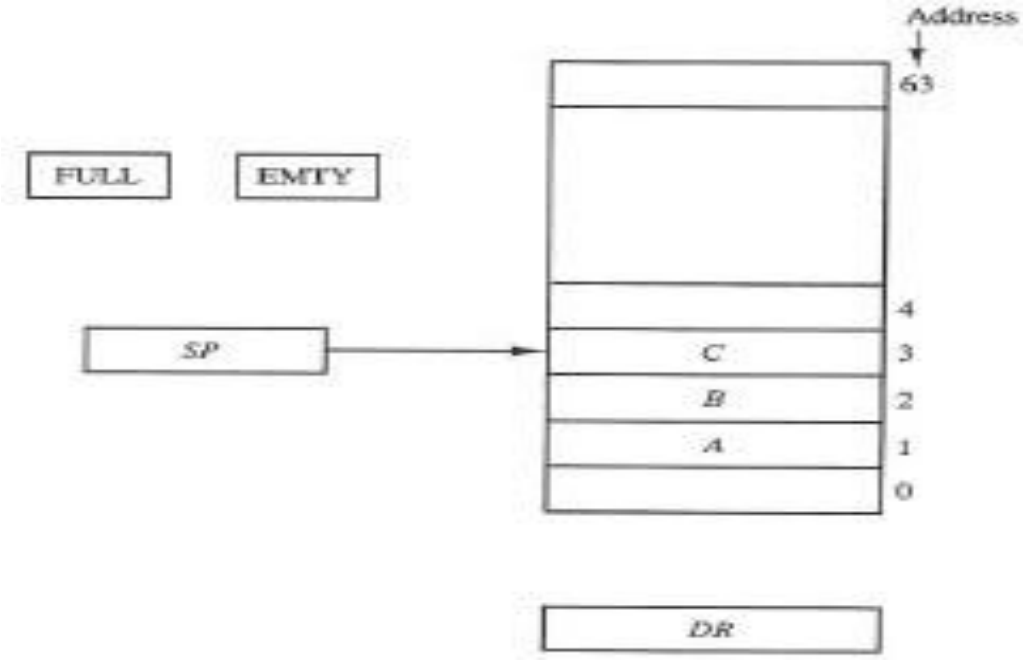
The two operations of a stack are the insertion and deletion of items. The operation of insertion is called push (or push-down) because it can be thought of as the result of pushing a new item on top. The operation of deletion is called pop (or pop-up) because it can be thought of as the result of removing one item so that the stack popsup.

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 pushed by incrementing SP and writing a word in the next- higher location in the stack. Note that item C has been read out but not physically removed. This does not matter because when the stack is pushed, a new item is written in its place.

Fig 3-1: Block diagram of 64-word stack



In a 64-word stack the stack pointer contains 6 bits because 2^6= 64. Since SP has only six bits, it cannot exceed a number greater than 63 (111111 in binary). When 63 is incremented by 1, the result 0 since 111111 + 1 = 1000000 in binary, but SP can accommodate only the six least significant bits.

Similarly, when 000000 is decremented by 1, the result is 111111. The one-bit register FULL is set to 1 when the stack is full, and the one-bit register EMTY is set to 1 when the stack is empty of items. DR is the data register that holds the binary data to be written into or read out of the stack.

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

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

The stack pointer is incremented so that it points to the address of the next-higher word. A memory write operation inserts the word from DR into the top of the stack.

SP holds the address of the top of the stack and the M[SP] donates the memory word specified by the address 1. The last item is stored at address 0. If SP reaches 0, the stack is full of items, so FULL is set to 1. If an item is written in the stack. obviously the stack cannot be empty, so EMTY is cleared to 0.

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

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

The top item is read from the stack into DR. The stack pointer is then decremented. If its value reaches zero, the stack is empty, so EMTY is set to 1. This condition is reached if the item read was in location 1. Once this item is read out, SP is decremented and reaches the value 0, which is the initial value of SP. An erroneous operation will result if the stack is pushed when FULL = 1 or popped when EMPTY =1.

Memory Stack

The implementation of a stack is the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer. Fig 3-2 shows a portion of computer memory partitioned into three segments program, data, and stack. The program counter PC points at the address of the nest instruction in the program. The address register AR points at an array of data. The stack pointer SP points at the top of the stack. The three registers are connected to a common address bus, and either one can provide an address for memory. PC is used during the fetch phase to read an instruction. AR is used during the execute phase to read an operand. SP is used to push or pop items into or from the stack.

As shown in Fig 3-2, the initial value of SP is 4001 and the stack grows with decreasing addresses. Thus the first item stored in the stack is at address 4000, the second item is stored at address 3999, and the last address that can be used for the stack is 3000. No provisions are available for stack limit checks.

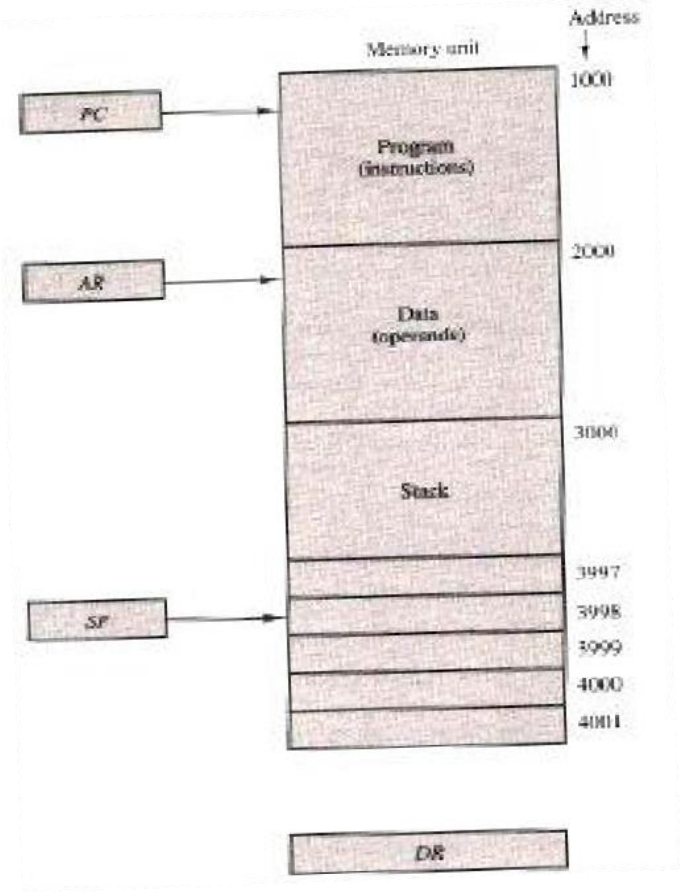
We assume that the items in the stack communicate with a data register DR. A new item is inserted with the push operation as follows.

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.

Fig 3-2: Computer memory with program, data and stack segments.



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.

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

Reverse Polish Notation

A stack organization is very effective for evaluating arithmetic expressions. The common arithmetic expressions are written in infix notation with each operator written between the operands. Consider the simple arithmetic expression.

A*B + C*D

The star (denoting multiplication) is placed between two operands A and B or C and D. The plus is between the two products. To evaluate this arithmetic expression it is necessary to compute the product A*B, store this product while computing C*D, and the sum the two products.

The Polish mathematician Lukasiewicz showed that arithmetic expressions can be represented in prefix notation. This representation often referred to as Polish notation. places the operator before the operands. The postfix notation referred to as reverse polish notation (RPN), places the operator after the operands. The following examples demonstrate the three representations:

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

The reverse Polish notation is in a form suitable for stack manipulation. The expression A*B + C* D is written in reverse Polish notation as

AB*CD * +

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

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

(A*B) CD* +

The next operator is a * and its previous two operands are C and D, so we perform C*D and obtain an expression with two operands and one operator.

(A*B) (C*D) +

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

Conversion to RPN

The conversion from infix notation to reverse Polish notation must take into consideration the operational hierarchy adopted for infix notation.

Consider the expression (A+B)*[C*(D+E)+F]

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

AB + DE + C* F+*

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

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

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

Evaluation of Arithmetic Expressions

Reverse Polish notation, combined with a stack arrangement of registers, is the most efficient way known for evaluating arithmetic expressions.

The Reverse Polish Notation is : A B + C D E + * F + *

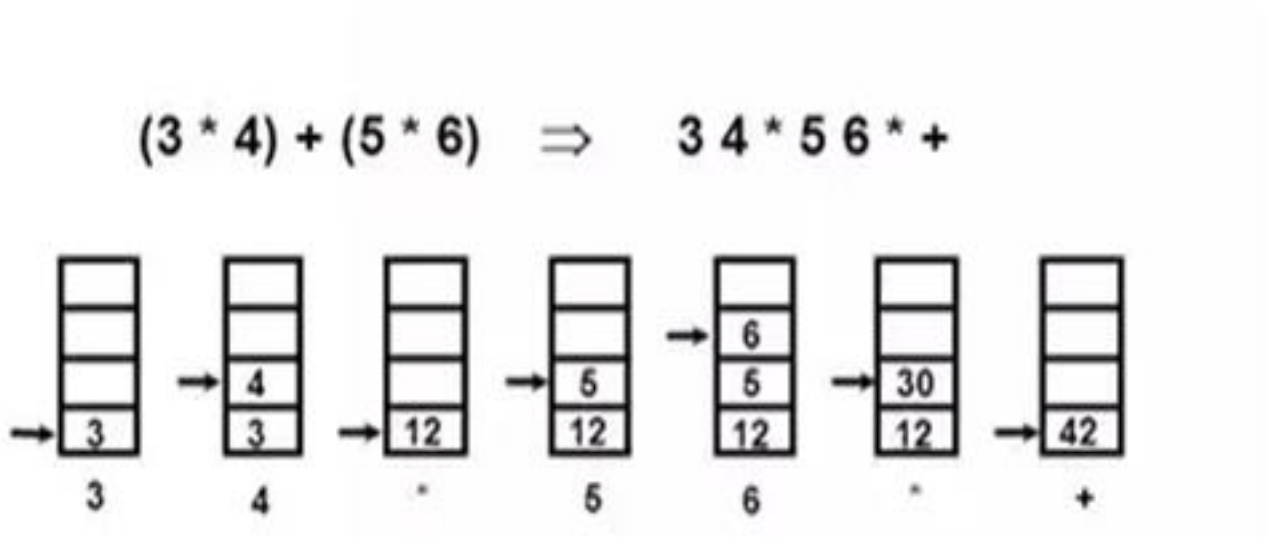
The Polish notation is : * + A B + * C + D E F

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

(3*4) + (5*6)

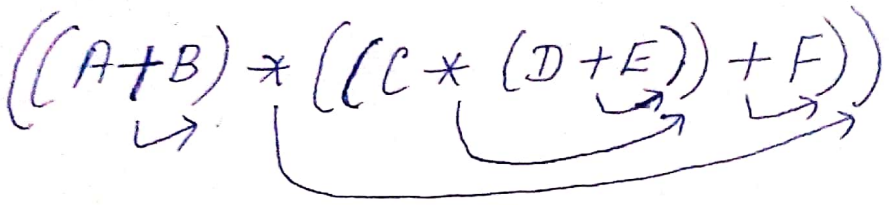
In reverse Polish notation, it is expressed as 34 * 56 * +

First the number 3 is pushed into the stack, then the number 4. The next symbol is the multiplication operator*. This causes a multiplication of the two topmost items in the stack. The stack is then popped and the product is placed on top of the stack, replacing the two original operands. Next we encounter the two operands 5 and 6, so they are pushed into the stack. The stack operation that results from the next* replaces these two numbers by their product. The last operation causes an arithmetic addition of the two topmost numbers in the stack to produce the final result of 42.



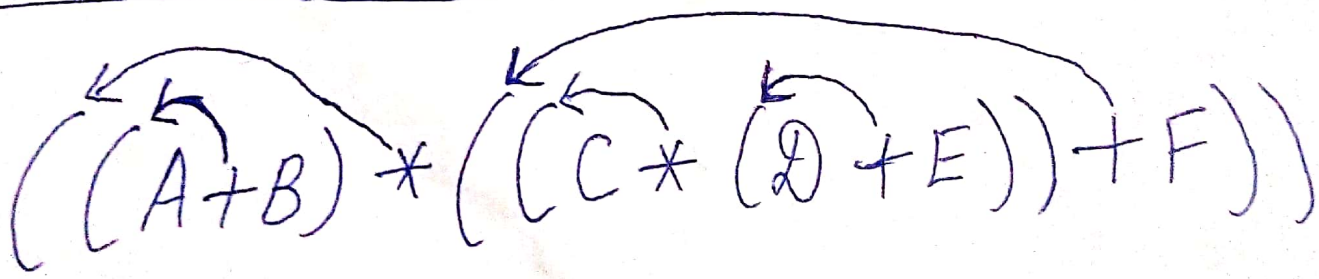
Infix :→ (A+B)*(C*(D+E)+F)

Infix to RPN (Postfix)



RPN: AB+CDE+*F+*

Infix to PN (Prefix)



PN: *+AB+*C+DEF

4. Instruction Formats

The physical and logical structure of computers is normally described in reference manuals provided with the system. Such manuals explain the internal construction of the CPU, including the processor registers available and their logical capabilities.

The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register. The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are;

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

The operation code field of an instruction is a group of bits that define various processor operations such as add, subtract, complement, and shift. The bits that define the mode field of an instruction code specify a variety of alternatives for choosing the operands from the given address.

A register address is a binary number of k bits that defines one of 2^k registers in the CPU. Thus a CPU with 16 processor registers R0 through R15 will have a register address field of four bits. The binary number 0101, for example, will designate register R5.

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

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

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

ADD X

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

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 R1 ← 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 R1 ← R1 + R2. Only register addresses for R1 and R2 need be specified in this instruction.

Computers with multiple processor registers use the move instruction with a mnemonic MOV to symbolize a transfer instruction. Thus the instruction

MOV R1, R2

denotes the transfer R1 ← R2 (or R2 ← R1, depending on the particular computer). Thus transfer – type instructions need two address fields to specify the source and the destination.

General register – type computers employ two or three address fields in their instruction format. Each address field may specify a processor register or a memory word. An instruction symbolized by

ADD R1, X

would specify the operation R1 ← R1 + M [X]. It has two address fields, one for register R1 and the other for the memory address X.

Computers with stack organized 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 instruction ADD in a stack computer consists of an operation code only with no address field. This operation has the effects of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack. There is no need to specify operands with an address field since all operands are implied to be in the stack.

Some computers combine features from more than one organizational structure. For example, the Intel 8080 microprocessor has seven CPU registers, one of which is an accumulator type. All arithmetic and logic instructions, as well as the load and store instructions, use the accumulator register, so these instructions have only one address field. On the other hand, instructions that transfer data among the seven processor registers have a format that contains two register address fields. Moreover, the Intel 8080 processor has a stack pointer and instructions to push and pop from a memory stack. The processor, however, does not have the zero-address-type instructions which are characteristic of a stack – organized CPU.

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 operand are in memory address 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 whether a processor register or a memory operand. The program in assembly 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 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 result 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 procedure 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.

One–Address Instructions

One-address instructions use an implied 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 the 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 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 Instructions

The instruction set of a typical RISC (Reduced Instruction Set Computer) processor is restricted to the use of load and store instructions when communicating between memory and CPU. All other instructions are executed within the registers of the CPU without referring to memory.

A program for a RISC–type CPU consists of LOAD and STORE instructions that have one memory and one memory and one register address, and computational–type instructions that have three addresses with all three specifying processor registers. The following is a program to evaluate X = (A+B) * (C +D)

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

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.

5. Addressing Modes

The addressing mode specifies a rule for interpreting or modifying the address field of the instructions before the operand is actually referenced. Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions

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

The control unit of a computer is designed to go through an instruction cycle that is divided into three major phases.

1. Fetch the instruction from memory
2. Decode the instruction
3. Execute the instruction.

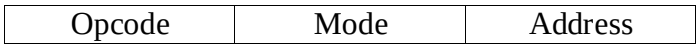
There is one register in the computer called the program counter or PC that keeps track of the instructions in the program stored in memory. PC holds the address of the instruction to be executed next and is incremented each time an instruction is fetched from memory.

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 filed, it may designate a memory address or a processor register. Most addressing modes modify the address field of the instruction there are two modes that need no address field at all. There are the implied and immediate modes.

Implied Mode

In this mode the operands are specified implicitly in the definition of the instruction. For example the instruction complement accumulator is an implied mode instruction, 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.

Fig 5-1: Instruction format with mode field



Immediate Mode

In this mode the operand is specified in the instruction itself, In order 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.

Ex:

In the immediate mode the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into AC. (The effective address in this case is 201.)

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.

Ex:

In the register mode the operand is in R1 and 400 is loaded into AC.

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 advantage of a register indirect mode instruction is that the address field of the instruction uses fewer bits to select a register than would have been required to specify the memory address directly.

Ex:

In the register indirect mode the effective address is 400, equal to the content of R1 and the operand loaded into AC is 700.

Autoincrement or Autodecrement 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. When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table. This can be achieved by using the increment or decrement instruction.

The effective address is defined to be the memory address obtained from the computation by the given addressing mode. The effective address is the address of the operand in a computational type instruction.

Ex:

The auto increment mode is the same as the register indirect mode except that R1 is incremented to 401 after the execution of the instruction. The auto decrement mode decrements R1 to 399 prior to the execution of the instruction. The operand loaded into AC is now 450.

Direct Address Mode

In this mode the effective address is equal to the address part of the instruction.

Ex:

In the direct address mode the effective address is the address part of the instruction 500 and the operand to be loaded into AC is 800.

Indirect Address Mode

In the mode the address field of the instruction gives the address where the effective address is stored in memory.

Effective Address = Address part of the instruction + Content of CPU register

Ex:

In the indirect mode the effective address is stored in memory at address 500. Therefore, the effective address is 800 and the operand is 300.

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.

Ex:

In the relative mode that effective address is 500 + 202= 702 and the operand is 325.

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.

Ex:

In the index mode the effective address is XR + 500 = 100+500 = 600 and the operand is 900.

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.

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.

Numerical Example

In Fig.5-2 The two– word instruction at address 200 and 201 is a –load to AC instruction with an address field equal to 500.

The first word of the instruction specifics the operation code and mode, and the second word specifies the address part. PC has the value 200 for fetching this instruction.

The content of processor register R1 is 400 and the content of an index register XR is 100.

AC receives the operand after the instruction is executed.

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

Fig 5-2: Numerical example for addressing modes

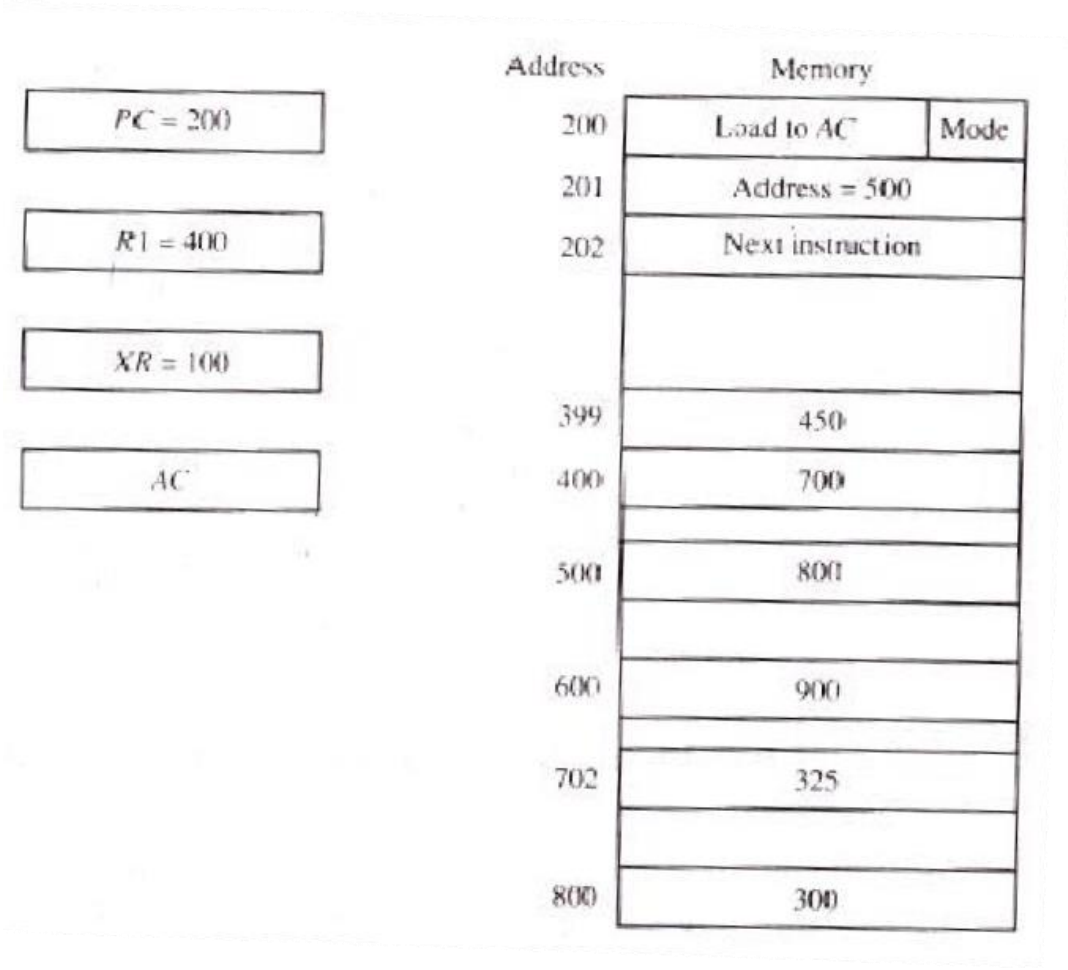


Table 5-1: Table list of numerical example

Addressing Modes	Effective Address	Content of AC
Immediate mode	201	500
Direct Address	500	800
Indirect Address	800	300
Register Mode	-	400
Register Indirect Mode	400	700
Auto Increment	400	700
Auto Decrement	399	450
Relative Address	702	325
Indexed Address	600	900

The above table lists the values of the effective address and the operand loaded into AC for the nine addressing modes.

8-6 Data Transfer and 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 so happens that the binary code assignments in the operation code field is different in different computers, even for the same operation. 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. Nevertheless, there is a set of basic operations that most, if not all, computers include in their instruction repertoire. The basic set of operations available in a typical computer is the subject covered in this and the next section.

set of basic operations

Most computer instructions can be classified into three categories:

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

Data transfer instructions cause transfer of data from one location to another without changing the binary information content. Data manipulation instructions are those that perform arithmetic, logic, and shift operations. Program control instructions provide decision making capabilities and change the path taken by the program when executed in the computer. 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. Accompanying each instruction is a mnemonic symbol. It must be realized that different computers use different mnemonics for the same instruction name.

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.

It must be realized that the instructions listed in Table 8-5, as well as in subsequent tables in this section, are often associated with a variety of addressing modes. Some assembly language conventions modify the mnemonic symbol to differentiate between the different addressing modes. For example, the mnemonic for *load immediate* becomes LDI. Other assembly language conventions use a special character to designate the addressing mode. For example, the immediate mode is recognized from a pound sign # placed before the operand. In any case, the important thing is to realize that each instruction can occur with a variety of addressing modes. As an example, consider the *load to accumulator* instruction when used with eight different addressing modes.

TABLE 8-5 Typical Data Transfer Instructions

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

TABLE 8-6 Eight Addressing Modes for the Load Instruction

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 8 6 shows the recommended assembly language convention and the actual transfer accomplished in each case. *ADR* stands for an address, *NBR* is a number or operand, *X* is an index register, *R*1 is a processor register, and *AC* is the accumulator register. The @ character symbolizes an indirect address. The $ character before an address makes the address relative to the program counter *PC*. The # character precedes the operand in an immediate-mode instruction. An indexed mode instruction is recognized by a register that is placed in parentheses after the symbolic address. The register mode is symbolized by giving the name of a processor register. In the register indirect mode, the name of the register that holds the memory address is enclosed in parentheses. The autoincrement mode is distinguished from the register indirect mode by placing a plus after the parenthesized register. The autodecrement mode would use a minus instead. To be able to write assembly language programs for a computer, it is necessary to know the type of instructions available and also to be familiar with the addressing modes used in the particular computer.

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

A list of data manipulation instructions will look very much like the list of microoperations given in Chap. 4. It must be realized, however, that each instruction when executed in the computer must go through the fetch phase

to read its binary code value from memory. The operands must also be brought into processor registers according to the rules of the instruction addressing mode. The last step is to execute the instruction in the processor. This last step is implemented by means of microoperations as explained in Chap. 4 or through an ALU and shifter as shown in Fig. 8-2. Some of the arithmetic instructions need special circuits for their implementation.

Arithmetic Instructions

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

A list of typical arithmetic instructions is given in Table 8-7. 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.

data type

The add, subtract, multiply, and divide instructions may be available for different types of data. The data type assumed to be in processor registers during the execution of these arithmetic operations is included in the definition of the operation code. An arithmetic instruction may specify fixed point or floating point data, binary or decimal data, single precision or double precision data. The various data types are presented in Chap. 3.

It is not uncommon to find computers with three or more add instruc

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

tions: one for binary integers, one for floating-point operands, and one for decimal operands. The mnemonics for three add instructions that specify different data types are shown below.

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

Algorithms for integer, floating-point, and decimal arithmetic operations are developed in Chap. 10.

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

Logical and Bit Manipulation Instructions

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

Some typical logical and bit manipulation instructions are listed in Table 8-8. 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. There are three bit manipulation operations possible: a selected bit can be cleared to 0, or can be set to 1, or can be complemented. The three logical instructions are usually applied to do just that.

clear selected bits

The AND instruction is used to clear a bit or a selected group of bits of an operand. For any Boolean variable x, the relationships $x \cdot 0 = 0$ and $x \cdot 1 = x$ dictate that a binary variable ANDed with a 0 produces a 0; but the 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

does not change in value when ANDed with a 1. Therefore, the AND instruction can be used to clear bits of an operand selectively by ANDing the operand with another operand that has 0's in the bit positions that must be cleared. The AND instruction is also called a *mask* because it masks or inserts 0's in a selected portion of an operand.

set selected bits

The OR instruction is used to set a bit or a selected group of bits of an operand. For any Boolean variable x, the relationships $x + 1 = 1$ and $x + 0 = x$ dictate that a binary variable ORed with a 1 produces a 1; but the variable does not change when ORed with a 0. Therefore, the OR instruction can be used to selectively set bits of an operand by ORing it with another operand with 1's in the bit positions that must be set to 1.

complement selected bits

Similarly, the XOR instruction is used to selectively complement bits of an operand. This is because of the Boolean relationships $x \oplus 1 = x'$ and $x \oplus 0 = x$. Thus a binary variable is complemented when XORed with a 1 but does not change in value when XORed with a 0. Numerical examples showing the three logic operations are given in Sec. 4-5.

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.

Shift Instructions

Instructions to shift the content of an operand are quite useful and are often provided in several variations. Shifts are operations in which the bits of a word are moved to the left or right. The bit shifted in at the end of the word determines the type of shift used. Shift instructions may specify 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 lists four types of shift instructions. The logical shift inserts 0 to the end bit position. The end position is the leftmost bit for shift right and the rightmost bit position for the shift left. Arithmetic shifts usually conform with the rules for signed 2's complement numbers. These rules are given in Sec. 4-6. The arithmetic shift right instruction must preserve the sign bit in the leftmost position. The sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unchanged. This is a shift right operation with the end bit remaining the same. The arithmetic shift left instruction inserts 0 to the end position and is identical to the logical shift left instruction. For this reason many computers do not provide a distinct arithmetic shift left instruction when the logical shift left instruction is already available.

The rotate instructions produce a circular shift. Bits shifted out at one end of the word are not lost as in a logical shift but are circulated back into the other end. The rotate through carry instruction treats a carry bit as an extension of the register whose word is being rotated. Thus a rotate left through carry instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit position into the carry, and at the same time, shifts the entire register to the left.

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

```
OP    REG    TYPE    RL    COUNT
```

Here OP is the operation code field; REG is a register address that specifies the location of the operand; TYPE is a 2 bit field specifying the four different types of shifts; RL is a 1 bit field specifying a shift right or left; and COUNT is a k bit field specifying up to $2^k - 1$ shifts. With such a format, it is possible to specify the type of shift, the direction, and the number of shifts, all in one instruction.

TABLE 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

8-7 Program Control

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

Some typical program control instructions are listed in Table 8 10. 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.

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

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

from this address. If the condition is not met, the program counter is not changed and the next instruction is taken from the next location in sequence.

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

The call and return instructions are used in conjunction with subroutines. Their performance and implementation are discussed later in this section. The compare and test instructions do not change the program sequence directly. They are listed in Table 8 10 because of their application in setting conditions for subsequent conditional branch instructions. The compare instruction performs a subtraction between two operands, but the result of the operation is not retained. However, certain status bit conditions are set as a result of the operation. Similarly, the test instruction performs the logical AND of two operands and updates certain status bits without retaining the result or changing the operands. The status bits of interest are the carry bit, the sign bit, a zero indication, and an overflow condition. The generation of these status bits will be discussed first and then we will show how they are used in conditional branch instructions.

Status Bit Conditions

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

1. Bit C (carry) is set to 1 if the end carry C_8 is 1. It is cleared to 0 if the carry is 0.
2. Bit S (sign) is set to 1 if the highest order bit F_7 is 1. It is set to 0 if the bit is 0.
3. Bit Z (zero) is set to 1 if the output of the ALU contains all 0's. It is cleared to 0 otherwise. In other words, $Z = 1$ if the output is zero and $Z = 0$ if the output is not zero.
4. Bit V (overflow) is set to 1 if the exclusive OR of the last two carries is equal to 1, and cleared to 0 otherwise. This is the condition for an

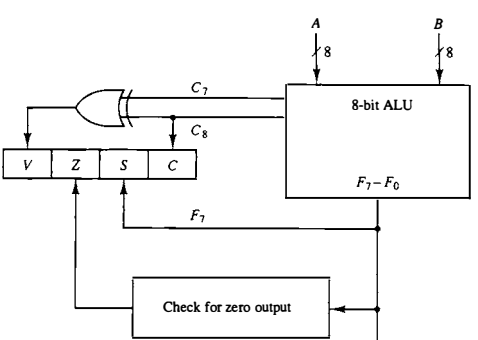


Figure 8-8 Status register bits.

overflow when negative numbers are in 2's complement (see Sec. 3-3). For the 8 bit ALU, $V = 1$ if the output is greater than +127 or less than −128.

The status bits can be checked after an ALU operation to determine certain relationships that exist between the values of A and B. If bit V is set after the addition of two signed numbers, it indicates an overflow condition. If Z is set after an exclusive OR operation, it indicates that $A = B$. This is so because $x \oplus x = 0$, and the exclusive OR of two equal operands gives an all 0's result which sets the Z bit. A single bit in A can be checked to determine if it is 0 or 1 by masking all bits except the bit in question and then checking the Z status bit. For example, let $A = 101x1100$, where x is the bit to be checked. The AND operation of A with $B = 00010000$ produces a result $000x0000$. If $x = 0$, the Z status bit is set, but if $x = 1$, the Z bit is cleared since the result is not zero. The AND operation can be generated with the TEST instruction listed in Table 8 10 if the original content of A must be preserved.

Conditional Branch Instructions

Table 8 11 gives a list of the most common branch instructions. Each mnemonic is constructed with the letter B (for branch) and an abbreviation of the condition name. When the opposite condition state is used, the letter N (for no) is

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	$A \neq B$

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.

The zero status bit is used for testing if the result of an ALU operation is equal to zero or not. The carry bit is used to check if there is a carry out of the most significant bit position of the ALU. It is also used in conjunction with the rotate instructions to check the bit shifted from the end position of a register into the carry position. The sign bit reflects the state of the most significant bit of the output from the ALU. $S = 0$ denotes a positive sign and $S = 1$, a negative sign. Therefore, a branch on plus checks for a sign bit of 0 and a branch on minus checks for a sign bit of 1. It must be realized, however, that these two conditional branch instructions can be used to check the value of the most significant bit whether it represents a sign or not. The overflow bit is used in conjunction with arithmetic operations done on signed numbers in 2's comple ment representation.

As stated previously, the compare instruction performs a subtraction of two operands, say $A - B$. The result of the operation is not transferred into a destination register, but the status bits are affected. The status register provides information about the relative magnitude of A and B. Some computers provide conditional branch instructions that can be applied right after the execution of a compare instruction. The specific conditions to be tested depend on whether the two numbers A and B are considered to be unsigned or signed numbers. Table 8 11 gives a list of such conditional branch instructions. Note that we use the words higher and lower to denote the relations between unsigned numbers, and greater and less than for signed numbers. The relative magnitude shown under the tested condition column in the table seems to be the same for unsigned and signed numbers. However, this is not the case since each must be considered separately as explained in the following numerical example.

numerical example

Consider an 8 bit ALU as shown in Fig. 8 8. The largest unsigned number that can be accommodated in 8 bits is 255. The range of signed numbers is between +127 and −128. The subtraction of two numbers is the same whether they are unsigned or in signed 2's complement representation (see Chap. 3). Let $A = 11110000$ and $B = 00010100$. To perform $A - B$, the ALU takes the 2's complement of B and adds it to A.

$$\begin{array}{rll} A: & 11110000 & \\ \overline{B} + 1: & +11101100 & \\ A - B: & 11011100 & C = 1 \quad S = 1 \quad V = 0 \quad Z = 0 \end{array}$$

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

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

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

It should be noted that the instruction BNE and BNZ (branch if not zero) are identical. Similarly, the two instructions BE (branch if equal) and BZ (branch if zero) are also identical. Each is repeated three times in Table 8 11 for the purpose of clarity and completeness.

It should be obvious from the example that the relative magnitude of two unsigned numbers can be determined (after a compare instruction) from the values of status bits C and Z (see Prob. 8 26). The relative magnitude of two signed numbers can be determined from the values of S, V, and Z (see Prob. 8 27).

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

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

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

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

Program Interrupt

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

The interrupt procedure is, in principle, quite similar to a subroutine call except for three variations: (1) The interrupt is usually initiated by an internal or external signal rather than from the execution of an instruction (except for software interrupt as explained later); (2) the address of the interrupt service program is determined by the hardware rather than from the address field of an instruction; 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. These three procedural concepts are clarified further below.

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

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

program status word

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

supervisor mode

Some computers store only the program counter when responding to an interrupt. The service program must then include instructions to store status and register content before these resources are used. Only a few computers store both program counter and all status and register content in response to an interrupt. Most computers just store the program counter and the PSW. In some cases, there exist two sets of processor registers within the computer, one for each CPU mode. In this way, when the program switches from the user to the supervisor mode (or vice versa) in response to an interrupt, it is not necessary to store the contents of processor registers as each mode uses its own set of registers.

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

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

The CPU does not respond to an interrupt until the end of an instruction execution. Just before going to the next fetch phase, control checks for any interrupt signals. If an interrupt is pending, control goes to a hardware interrupt cycle. During this cycle, the contents of PC and PSW are pushed onto the stack. The branch address for the particular interrupt is then transferred to PC and a new PSW is loaded into the status register. The service program can now be executed starting from the branch address and having a CPU mode as specified in the new PSW.

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

Types of Interrupts

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

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

External interrupts come from input–output (I/O) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source. Examples that cause external interrupts are I/O device requesting transfer of data, I/O device finished transfer of data, elapsed time of an event, or power failure. 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 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.

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

the program while external interrupts are asynchronous. 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 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. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program. The most common use of software interrupt is associated with a supervisor call instruction. This instruction provides means for switching from a CPU user mode to the supervisor mode. 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 requested.