Unit 3

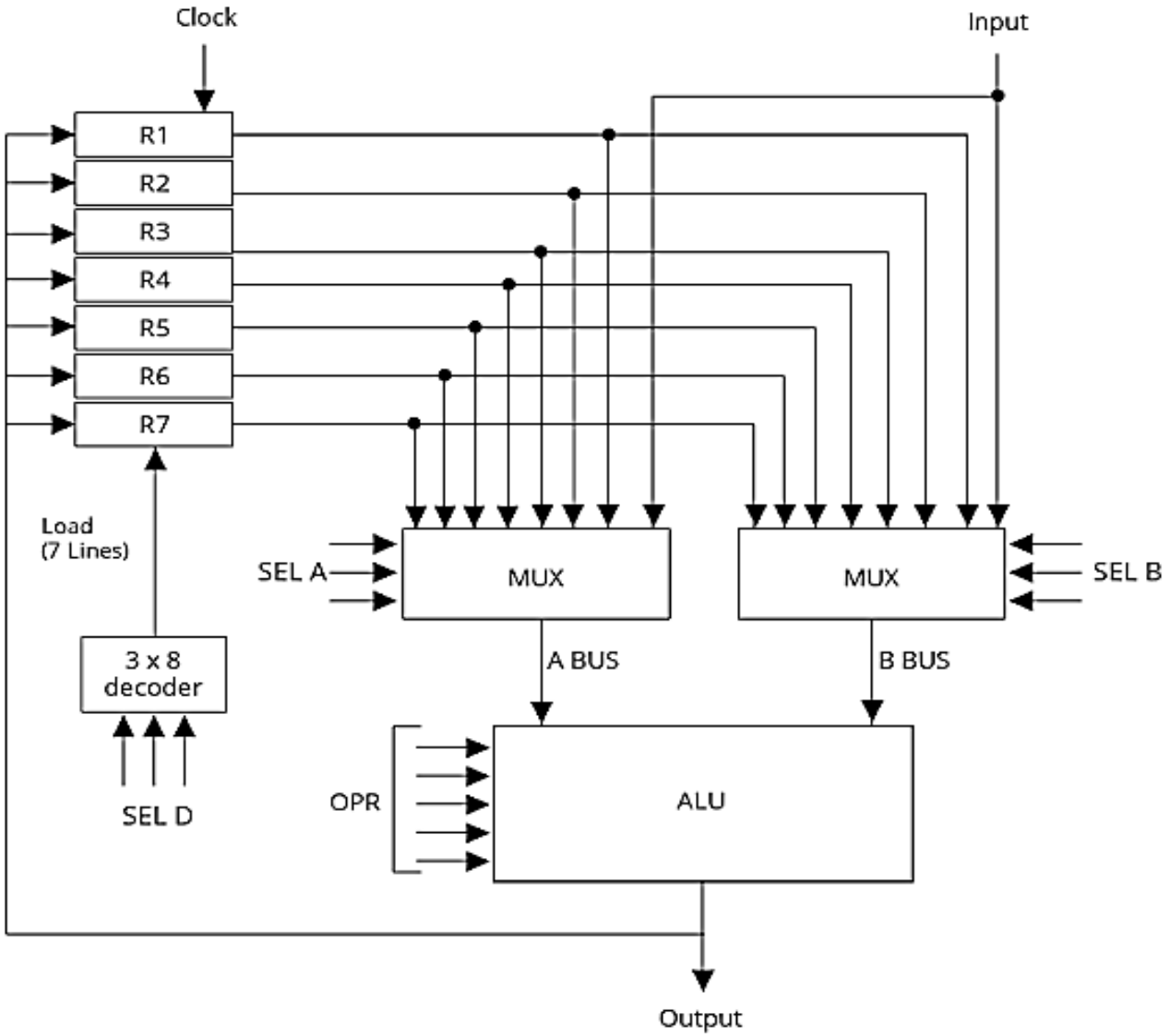
Central Processing Unit

SANTOSH MEENA

General Register Organization

- The memory locations are needed for storing pointers, counters, return addresses, temporary results, and partial products during multiplication.
- Having to refer to memory locations for such applications is time consuming because memory access is the most time-consuming operation in a computer.
- It is more convenient and more efficient to store these intermediate values in processor registers.
- When a large number of registers are included in the CPU, it is most efficient to connect them through a common bus system.
- The registers communicate with each other not only for direct data transfer, 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.

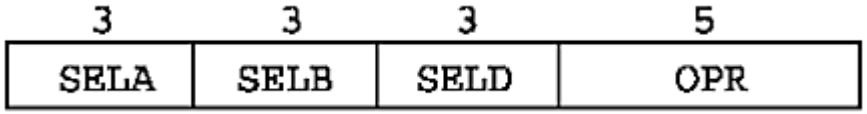
Continue…



Register set with common ALU

Continue…

- For example, to perform the operation- R1 ← R2 + R3, the control must provide binary selection variables to the following selector inputs:
- 1. MUX A selector (SELA): to place the content of R2 into bus A.
- 2. MUX B selector (SELB): to place the content of R3 into bus B.
- 3. ALU operation selector (OPR): to provide the arithmetic addition A + B.
- 4. Decoder destination selector (SELD): to transfer the content of the output bus into R1.
- **Control Word:**



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

Continue…

TABLE 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*

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

TABLE Encoding of ALU Operations

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

- The ALU provides arithmetic and logic operations. In addition, the CPU must provide shift operations.

Continue…

Example of μOperations

- $R_1 \leftarrow R_2 - R_3$

Field :	SELA	SELB	SELD	OPR
Symbol :	R_2	R_3	R_1	SUB
Control word :	010	011	001	00101

Examples of Microoperations for the CPU

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
$R4 \leftarrow$ shl $R4$	R4	--	R4	SHLA	100 000 100 11000
$R5 \leftarrow 0$	R5	R5	R5	XOR	101 101 101 01100

- We assign 000 to any unused field when formulating the binary control word.

Stack Organization

- A **stack** is a storage device that stores information in such a manner that the item stored last is the first item retrieved (LIFO).
- The stack in digital computer is essentially a memory unit with an address register.
- 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 operation of a stack are the insertion and deletion of items.
- The operation of insertion is called **push** (incrementing the SP register). The operation of deletion is called **pop** (decrementing the SP register).

Continue…

- **Register Stack:** A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers.
- The SP register 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 (C), the stack is popped by reading the memory word at address 3 and decrementing the content of SP. Item (B) is now on top of the stack since SP holds address 2.
- To insert a new item, the stack is pushed by incrementing SP and writing a word in the next-higher location in the stack.
- Note that item (C) has been read out but not physically removed. This does not matter because when the stack is pushed, a new item is written in its place.
- In a 64-word stack, the SP contains 6 bits because $2^6 = 64$. Since SP has only six bits, it cannot exceed a number greater than 63 (111111) in binary.

Continue…

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.

If the stack is not full (if FULL = 0), a new item is inserted with a push operation implemented with the following sequence of micro-operations:

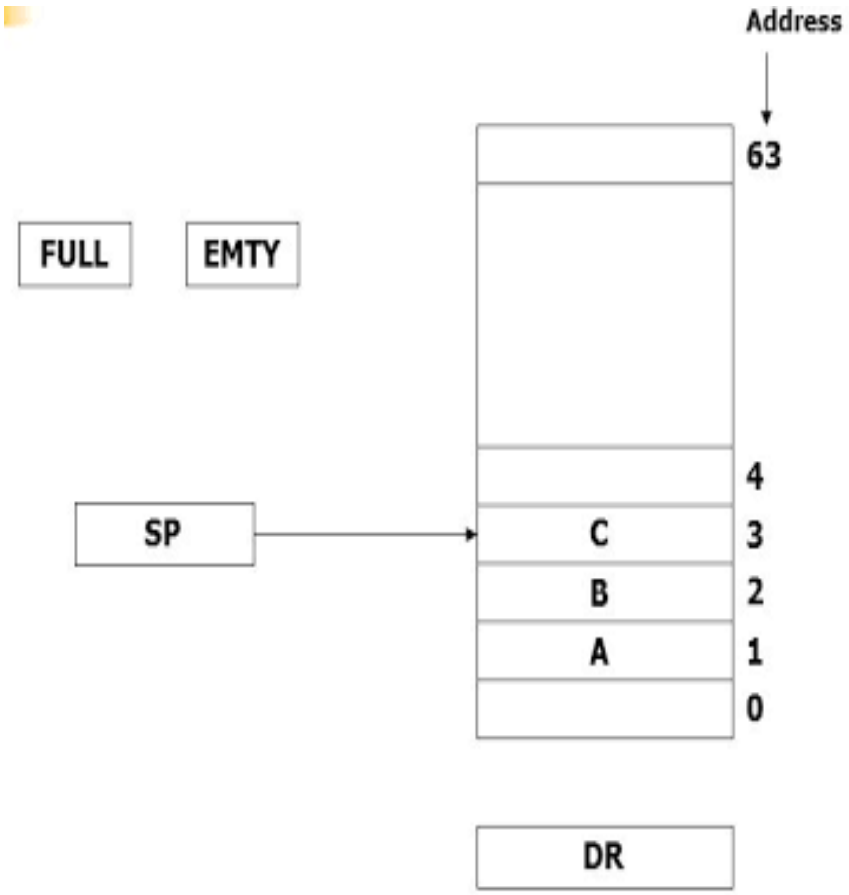
$$SP \leftarrow SP + 1$$

$$M[SP] \leftarrow DR$$

If the stack is not empty (if EMTY = 0), a new item is deleted with a pop operation implemented with the following sequence of micro-operations:

$$DR \leftarrow M[SP]$$

$$SP \leftarrow SP - 1$$



- **Block diagram of a 64-word stack**

Continue…

- **Memory Stack:** A stack can exist as a stand-alone unit (register stack) or can be implemented in RAM 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.
- A SP is loaded with an initial value. This initial value must be the bottom address of an assigned stack in memory.
- Henceforth, SP is automatically decremented or incremented with every push or pop operation.
- The initial value of SP is 4001 (the bottom address of an assigned stack in memory) 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.
- **Stack Limits:** The stack limits can be checked by using two processor registers: one to hold the upper limit (3000 in this case), and the other to hold the lower limit (4001 in this case).
- After a push operation, SP is compared with the upper-limit register and after a pop operation, SP is compared with the lower-limit register.

Continue…

A new item is inserted with the push operation as follows:

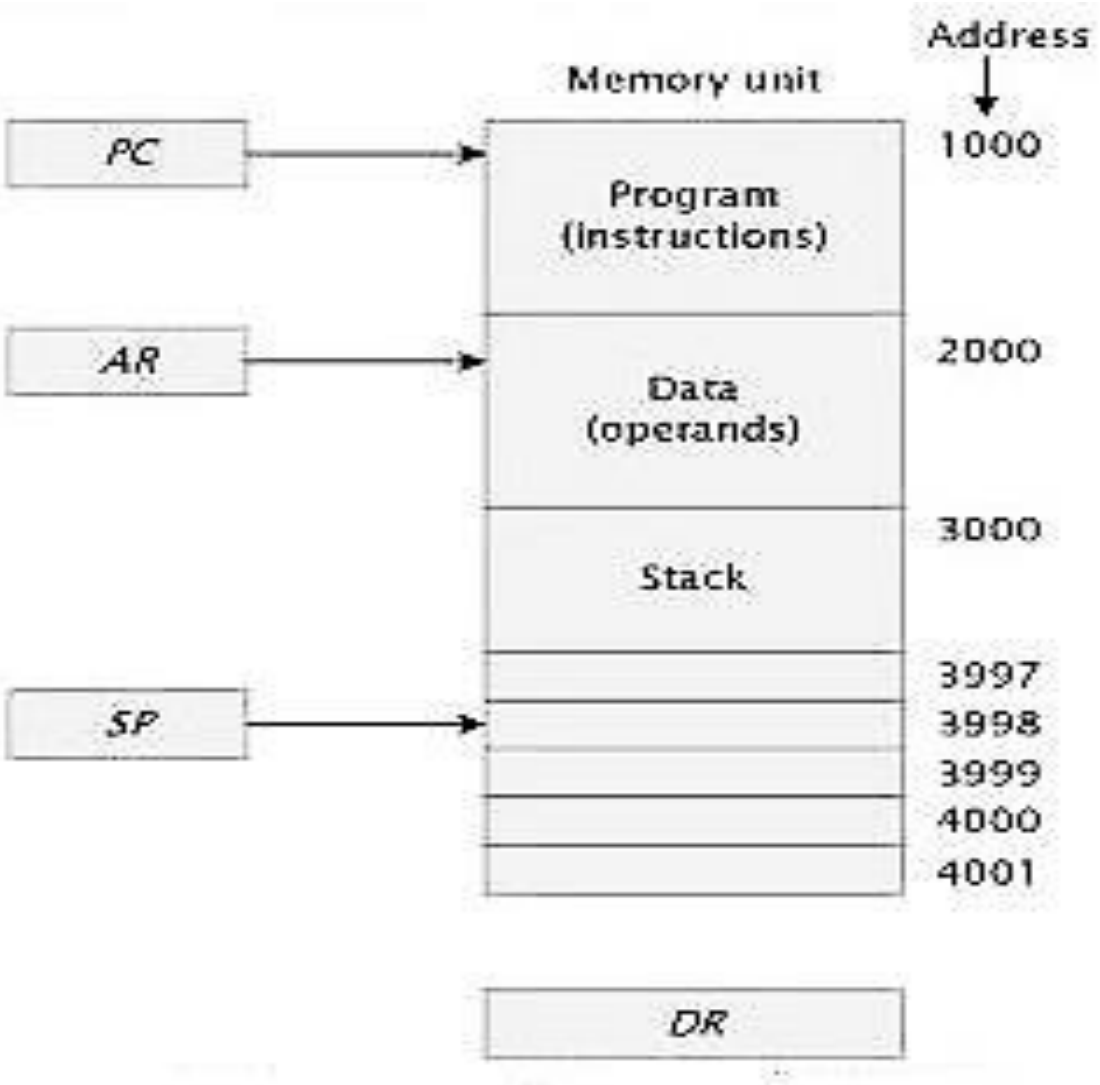
$SP \leftarrow SP - 1$

$M[SP] \leftarrow DR$

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

$DR \leftarrow M[SP]$

$SP \leftarrow SP + 1$



- **Computer memory with program, data, and stack segments**

Instruction Formats

- The bits of instruction are divided into groups called fields. The most common fields found in instruction formats are:
- 1. An operation code field that specifies the operation to be performed.
- 2. An address field that designates a memory address or a processor register.
- 3. A mode field that specifies the way the operand or the effective address is determined.
- Operations specified by computer instructions are executed on some data stored in memory or processor registers.
- Operand residing in memory are specified by their memory address. Operand residing in processor registers are specified with a register address.
- A register address is a binary number of k bits that defines one of 2^k registers in the CPU. For example k = 4, CPU with 16 processor registers R0 through R15, the 0101 will designate register R5.
- The number of address fields in the instruction format of a computer depends on the internal organization of its registers. Most computers fall into one of three types of CPU organizations:
    1. Single accumulator organization
    2. General register organization
    3. Stack organization

Continue…

- **Single accumulator organization:** All operations are performed with an implied accumulator register. The instruction format in this type of computer uses one address field.

ADD X

AC ← AC + M[X]

- Where X is the address of the operand (located at address X in memory).
- **General register organization:** The instruction format in this type of computer needs three or two register address fields. Each address field may specify a processor register or a memory word.

ADD R1, R2, R3 (three address: R1, R2, R3)

R1 ← R2 + R3

ADD R1, R2 (two address: R1, R2)

R1 ← R1 + R2

MOV R1, R2 (two address: R1, R2)

R1 ← R2

ADD R1, X (two address: one for R1 and other for memory address X)

R1 ← R1 + M[X]

Continue…

- **Stack organization:** Computers with stack organization would have PUSH and POP instructions which require one address field.

PUSH X

- This instruction will push the word at address X to the top of the stack. The SP is updated automatically.
- Operation-type instructions do not need an address field in stack-organized computers. This is because the operation is performed on the two items that are on top of the stack.

ADD

- This instruction in a stack computer consists of an operation code only with no address field. This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack.

Continue…

- **Reverse Polish Notation:** A stack organization is very effective for evaluating arithmetic expressions.

  A + B Infix Notation

  + AB Prefix or Polish Notation

  AB + Postfix or Reverse Polish Notation

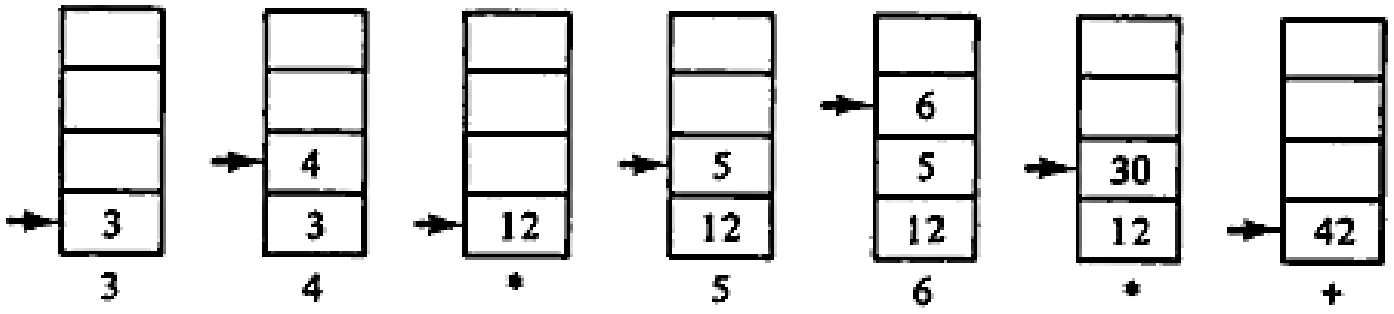
- The reverse polish notation (RPN) is in a form suitable for stack manipulation. The expression is written in reverse polish notation as:

  A * B + C * D

  AB * CD * + Reverse Polish Notation

- **Stack operations:**

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



Continue…

- Computers may have instructions of several different lengths containing varying number of addresses: Three-address instructions, Two-address instructions, One-address instructions, Zero-address instructions.
- **Three-Address Instructions:** Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand. For example, the program to evaluate is as follows:

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

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

- The computer has two processor registers, R1 and R2. The symbol M[A] denotes the operand at memory address symbolized by A.
- The advantage of the three-address format is that it results in short programs when evaluating arithmetic expressions.
- The disadvantage is that the binary-coded instructions require too many bits to specify three addresses.

Continue…

- **Two-Address Instructions:** Two-address instructions are the most common in commercial computers.
- Each address field can specify either a processor register or a memory word. For example, the program to evaluate is as follows:

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

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

Continue…

- **One-Address Instructions:** These instructions use an implied accumulator (AC) register for all data manipulation. For example, the program to evaluate is as follows:

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

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.

Continue…

- **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.
- To evaluate arithmetic expressions in a stack computer, it is necessary to convert the expression into reverse polish notation (RPN).

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

RPN: AB + CD + *

(A + B) CD + *

(A + B) (C + D) *

(A + B) * (C + D)

Continue…

- For example, the program to evaluate is as follows: (TOS stand for top of stack)

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

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

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

Continue…

- **RISC Instructions:** A program for a RISC-type (Reduced Instruction Set Computer) CPU consists of LOAD and STORE instructions that have one memory and one register address, and computational-type instructions that have three addresses with all three specifying processor registers. For example, the program to evaluate is as follows:

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

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

Addressing Modes

- The operation to be performed specified by operation field of instruction must be executed on some data stored in computer registers or memory words.
- The way the operands are chosen during program execution is dependent on the addressing mode of the instruction.
- The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.
- Although most addressing modes modify the address field of the instruction, there are two modes that need no address field at all. These are the implied and immediate modes.

Figure Instruction format with mode field.

Continue…

- **Implied Mode:** In this mode the operands are specified implicitly in the definition of the instruction.
- For example, the instruction “complement accumulator”, the operand in the accumulator register is implied in the instruction.
- All register-reference instructions that use 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.
- **Immediate Mode:** In this mode the operand is specified in the instruction itself. In other words, an immediate-mode instruction has an operand field rather than an address field.
- Immediate-mode instructions are useful for initializing registers to a constant value.

Continue…

- **Register Mode:** The address field of an instruction may specify either a memory word or a processor register. When the address field specifies a processor register, the instruction is said to be in the register mode.
- In this mode the operands are in registers that reside within the CPU. The particular register is selected from a register field in the instruction. A k-bit can specify any one of 2^k registers.
- **Register Indirect Mode:** In this mode the instruction specifies a register in the CPU, who contains the address of the operand in memory rather than the operand itself.
- Before using this mode, the programmer must ensure that the memory address of the operand is placed in the processor register with a previous instruction.
- The advantage of a register indirect mode instruction is that the address field of the instruction uses fewer bits to select a register.

Continue…

- **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.
- However, because it is such a common requirement, some computers incorporate a special mode that automatically increments or decrements the content of the register after data access.
- **Direct Address Mode:** In this mode, the effective address is equal to the address part of the instruction.
- The operand resides in memory and its address is given directly by the address field of the instruction.
- In branch-type instruction the address field specifies the actual branch address.

Continue…

- **Indirect Address Mode:** In this mode the address field of the instruction gives the address where the effective address is stored in memory.
- Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.
- **Relative Address Mode:** In this mode the content of PC is added to the address part of the instruction in order to obtain the effective address whose position in memory is relative to the address of the next instruction. Relative addressing is often used with branch-type instructions.
- For example PC = 825 and address part of the instruction = 24. The instruction at location 825 is read from memory during fetch phase and the PC is then incremented by one to 826 (address of the next instruction).
- The effective address computation for the relative address mode is 826 + 24 = 850. This is 24 memory locations forward from the address of the next instruction (826).

Continue…

- **Indexed Addressing Mode:** In this mode the content of index register is added to the address part of the instruction to obtain the effective address. The index register is a special CPU register that contains an index value.
- The address field of the instruction defines the beginning address of a data array in memory. Each operand in the array is stored in memory relative to the beginning address of array.
- The distance between the beginning address and the address of the operand is the index value stored in the index register.
- An index register is assumed to hold an index number that is relative to the address part of the instruction.
- The index register can be incremented to facilitate access to consecutive operands.

Continue…

- **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 indexed addressing mode except that the register is now called a base register instead of an index register.
- When programs and data are moved from one segment of memory to another, as required in multiprogramming systems, the address values of instructions must reflect this change of position.
- With a base register, the displacement values of instruction do not have to change. Only the value of the base register requires updating to reflect the beginning of a new memory segment.

Continue…

- **Example:**

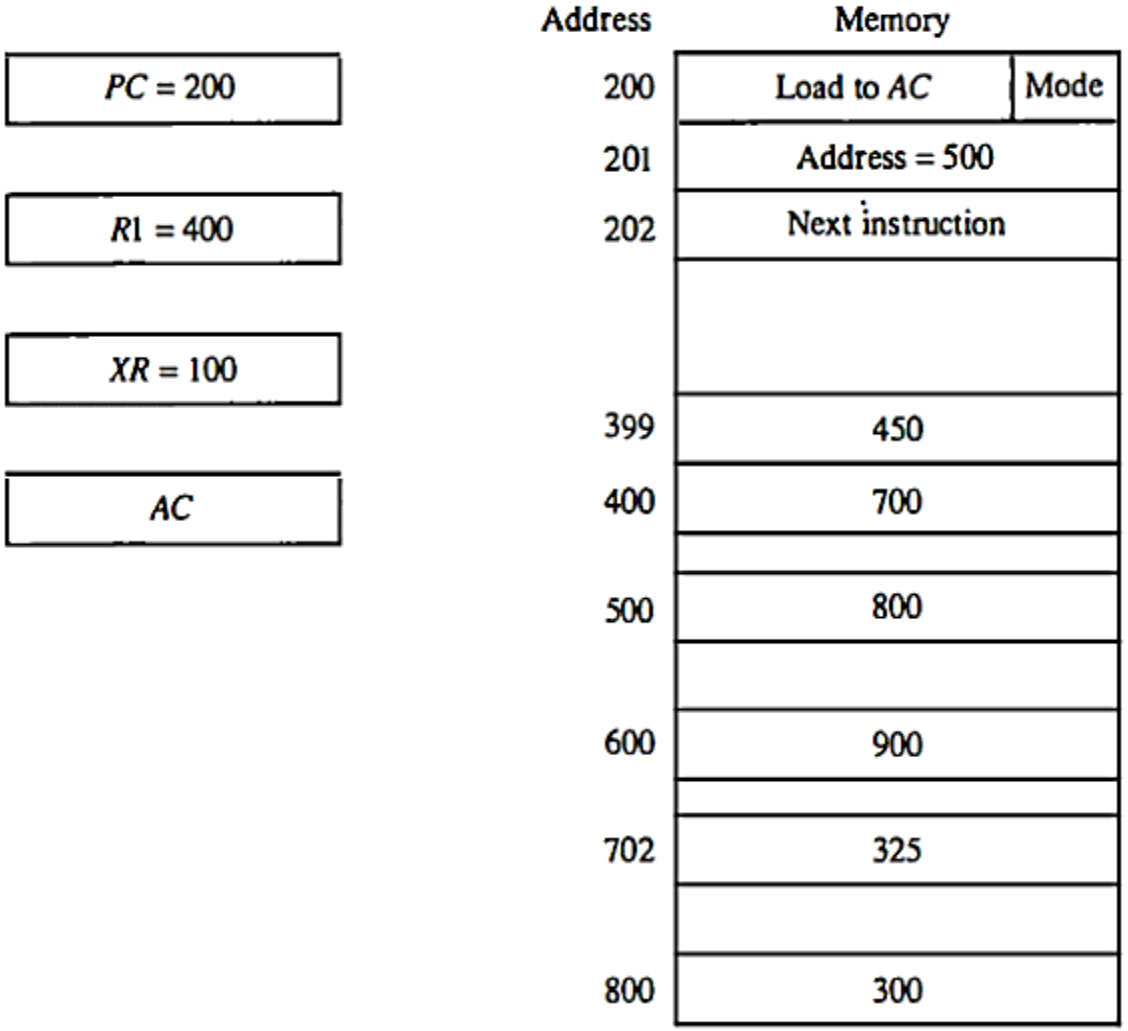


Figure Numerical example for addressing modes.

Continue…

- In the **relative mode** the effective address (address part of the instruction + next instruction address) is 500 + 202 = 702 and the operand is 325, AC is loaded with 325.
- In the **index mode** the effective address (index register content + address part of the instruction) is XR + 500 = 100 + 500 = 600 and the operand is 900, AC is loaded with 900.
- In the **register mode** the operand is in R1 and 400 is loaded into AC. There is no effective address in this case.
- The **autoincrement mode** is the same as the register indirect mode except that R1 is incremented to 401 after the execution of the instruction.
- The **autodecrement mode** decrements R1 to 399 prior to the execution of the instruction. The operand loaded into AC is now 450.

Continue…

TABLE Tabular List of Numerical Example

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

Data Transfer and Manipulation

- Most computer instructions can be classified into three categories:
    1. Data Transfer instructions
    2. Data Manipulation instructions
    3. Program Control instructions
- **Data Transfer Instructions:** Data transfer instructions move data from one place in the computer to another without changing the data content.
- The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves.
- The **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.

Continue…

- 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.

Typical Data Transfer Instructions

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

Continue…

Data Transfer Instructions with Different Addressing Modes

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

Continue…

- **Data Manipulation Instructions:** 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
- **Arithmetic Instructions:** The four basic arithmetic operations are addition, subtraction, multiplication, and division.
- 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 operations when executed in processor registers is that a binary number of all 0's when decremented produces a result of all 1's.
- The add, subtract, multiply, and divide instructions may be available for different types of data.
- An arithmetic instruction may specify fixed-point or floating- point data, binary or decimal data, single-precision (single word) or double-precision data (two words).

Continue…

- Three or more add instructions: one for binary integers, one for floating-point operands, and one for decimal operands.

  ADDI Add two binary integer numbers

  ADDF Add two floating-point numbers

  ADDD Add two decimal numbers in BCD

TABLE 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

Continue…

- **Logical and Bit Manipulation Instructions:** Logical instructions perform binary operations on strings of bits stored in registers. These instructions consider each bit of the operand separately and treat it as a Boolean variable.
- They are very useful for manipulating individual bits or a group of bits that represent binary-coded information.
- They can be used to change bit value, to clear a group of bits, or insert new bit values into operands stored in registers or memory words.
- **Clear carry (CLRC):** The AND instruction is used to clear a bit or a selected group of bits of an operand. The AND instruction is also used called a mask because it masks or inserts 0's in a selected portion of an operand.
- **Set Carry (SETC):** The OR instruction is used to set a bit or a selected group of bits of an operand.
- **Complement Carry (COMC):** The XOR instruction is used to selectively complement bits of an operand.

Continue…

- Another example is a flip-flop that controls the interrupt facility and is either enabled or disabled by means of bit manipulation instructions.

TABLE 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

Continue…

- **Shift Instructions:** Shift are operations in which the bits of a word are moved to the left or right. They may specify either arithmetic shifts, logical shifts, or rotate-type operations.

TABLE 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

Continue…

- Some computers have a multiple-field format for the shift instructions. 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 shift.
- RL is a 1-bit field specifying a shift right or left.
- 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.

Program Control

- After the execution of a data transfer or data manipulation instruction, control returns to the fetch cycle with the PC 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 PC and causes a break in the sequence of instruction execution, and cause the flow of control to be altered.
- Branch and jump instructions may be conditional or unconditional.
- An unconditional branch instruction causes a break to the specified address without any conditions.
- The conditional branch instruction specifies a condition, if the condition is met, the PC is loaded with the branch address and the next instruction is taken from this address. If the condition is not met, PC is not changed and the next instruction is taken from the next location, continuation in sequence.

Continue…

- The branch is usually a one-address instruction. It is written in assembly language as BR ADR, where ADR is for an address. When executed, the branch instruction causes a transfer of the value of ADR into the PC.
- The skip instruction does not need an address field and is therefore a zero-address instruction.
- The call and return instructions are used in conjunction with subroutines.
- The compare and test instruction do not change the program sequence directly.

TABLE Typical Program Control Instructions

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

Continue…

- **Status Bit Conditions:** Status bits are also called condition-code bits or flag bits. The four status bits are 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 F7 is 1. It is set to 0 if the bit is 0.
- 3. Bit Z (zero) is set to 1 if the output of the ALU contains all 0's. It is cleared to 0 if the output is not zero.
- 4. Bit V (overflow) is set to 1 if the Ex-OR of the last two carries is equal to 1, and cleared to 0 otherwise. For the 8-bit ALU, V = 1 if the output for signed number is greater than +127 or less than -128 (range of signed numbers) and V = 1 if the output for unsigned numbers is greater than 255 (largest unsigned number in 8-bit).

Continue…

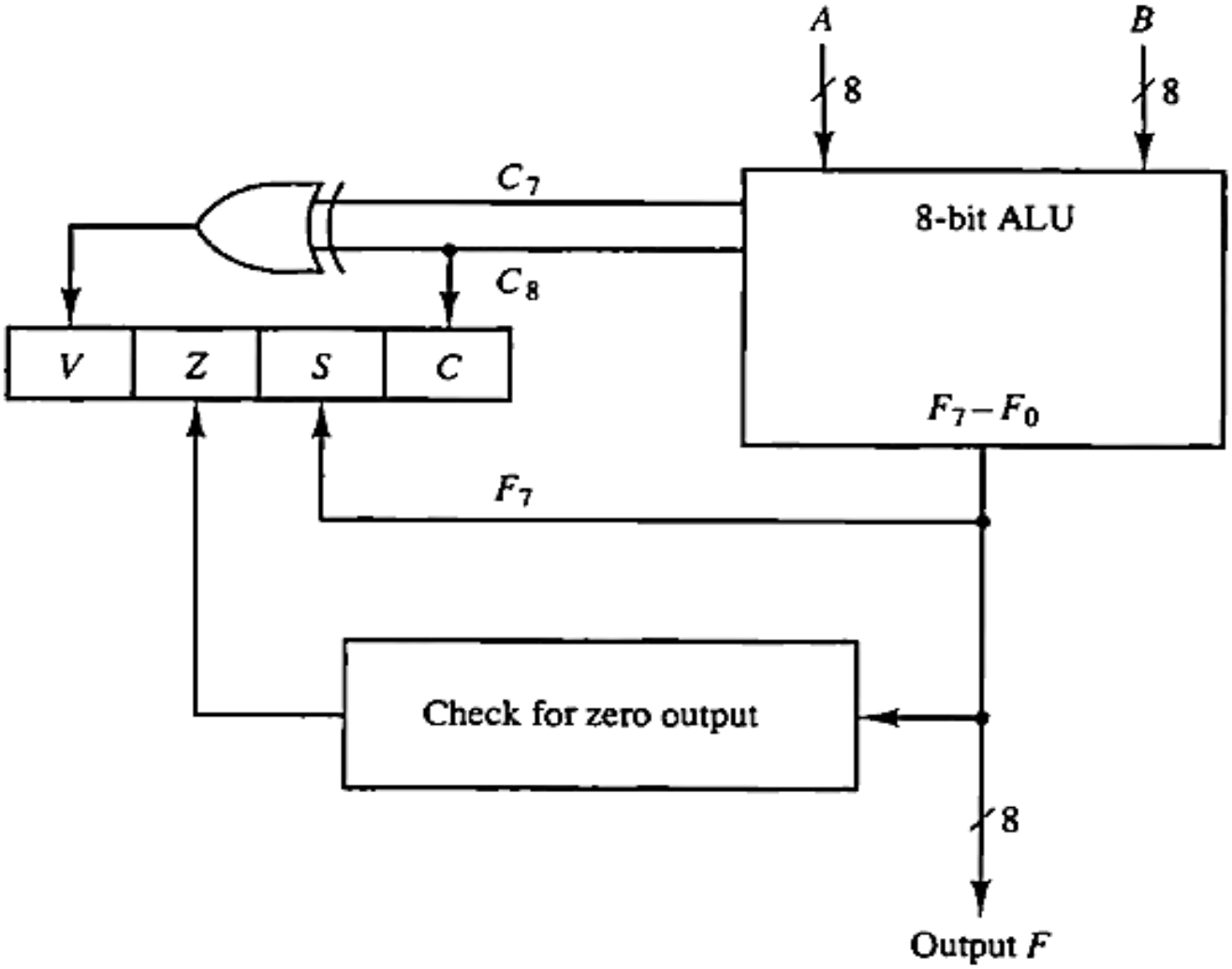


Figure Status register bits.

Continue…

- **Conditional Branch Instructions:**

TABLE 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	$A \neq B$

Continue…

- **Example:** The subtraction of two numbers is the same whether they are unsigned or in signed 2's complement representation.
- Let A = 11110000 and B = 00010100. To perform A – B, the ALU takes the 2's complement of B and adds it to A.

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

- **For unsigned numbers:** A = 11110000 (240), B = 00010100 (20), A – B = 240 – 20 = 220; 240 > 20; A > B, and A ≠ B.
- The instruction (for unsigned compare) 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).
- **For signed numbers:** A = 11110000 (-16), B = 00010100 (+20), A – B = (-16) - (+20) = (-36); (-16) < (+20); A < B, and A ≠ B.
- The instruction (for signed compare) 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).

Continue…

- **Subroutine Call and Return:** A subroutine is a self-contained sequence of instructions that performs a given computational task.
- 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 as 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 executed by performing two operations:
- 1. The address of the next instruction available in the PC (the return address) is stored in a temporary location so the subroutine knows where to return.
- 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 PC.

Continue…

- The return address may be stored in first memory location of the subroutine, in a fixed location in memory, in a processor register, 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 without destroying any previous values.
- The return from subroutine instruction causes the stack to pop and the contents of the top of the stack are transferred to the PC.
- A subroutine call is implemented with the following micro-operations:

$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PC$	Push content to 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 micro-operations:

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

Continue…

- **Program Interrupt:** Program interrupt refers to the transfer of program control from a currently running program to another service program as a result of an external or internal generated request. Control returns to the original program after the service program is executed.
- The interrupt procedure is quite similar to a subroutine call except for three variations:
- 1. The interrupt is usually initiated by an internal or external signal rather than from the execution of an instruction (except for software interrupt).
- 2. The address of the interrupt service program is determined by the hardware rather than from the address field of an instruction.
- 3. An interrupt procedure usually stores all the information necessary to define the state of the CPU rather than storing only the PC.
- 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.
- The state of the CPU at the end of the execute cycle is determined from:
- 1. The content of the program counter
- 2. The content of all processor registers
- 3. The content of certain status conditions

Continue…

- 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.
- 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. The CPU is normally in the user mode when executing user programs.
- The mode that the CPU is operating at any given time is determined from special status bits in the PSW.
- The state of the CPU is pushed into a memory stack and the beginning address of the service routine is transferred to the PC.
- 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 interrupts signals. If an interrupt is pending, control goes to a hardware interrupt cycle.
- During interrupt 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 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 PC. Thus the CPU state is restored and the original program can continue executing.

Continue…

- **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:** They come from I/O devices, from a time device, from a circuit monitoring the power supply, or from any other external source.
- Examples that cause externa interrupts are I/O device requesting transfer of data, I/O device finished transfer of data or power failure.
- **2. Internal Interrupts:** They arise from illegal or erroneous use of an instruction or data. They 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.
- External and internal interrupts are initiated from signals that occur in the hardware of the CPU.
- **3. Software Interrupts:** A software interrupt is initiated by executing an instruction. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program.

Complex Instruction Set Computer (CISC)

- A computer with a large number of instructions is classified as a **complex instruction set computer or CISC.**
- The essential goal of a CISC architecture is to provide a single machine instruction for each statement that is written in a high-level language.
- Another characteristic of CISC architecture is the incorporation of variable-length instruction formats.
- Instructions that require register operands may be only two bytes in length, but instructions that need two memory addresses may need five bytes to include the entire instruction code.
- The instructions in a typical CISC processor provide direct manipulation of operands residing in memory.
- The major characteristics of CISC architecture are:
- 1. A large number of instructions-typically from 100 to 250 instructions
- 2. Some instructions that perform specialized tasks and are used infrequently
- 3. A large variety of addressing modes-typically from 5 to 20 different modes
- 4. Variable-length instructions formats
- 5. Instructions that manipulate operands in memory
- As more instructions and addressing modes are incorporated into a computer, the more hardware logic is needed to implement and support them, and this may cause the computations to slow down.

Reduced Instruction Set Computer (RISC)

- 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.**
- The concept of RISC architecture involves an attempt to reduce execution time by simplifying the instruction set of the computer. The major characteristics of a RISC processor are:
- 1. Relatively few instructions
- 2. Relatively few addressing modes
- 3. Memory access limited to load and store instructions
- 4. All operations done within the registers of the CPU
- 5. Fixed-length, easily decoded instruction format
- 6. Single-cycle instruction execution
- 7. For faster operations, Hardware control is preferable rather than microprogrammed control

Continue…

- The small set of instructions of a typical RISC processor consists mostly of register-to-register operations, with only simple load and store operations for memory access.
- Each operand brought into a processor register with a load instruction. After computation results are transferred to memory by means of store instructions.
- RISC instruction format is easy to decode. Thus the operation code and register fields of the instruction code can be accessed simultaneously by the control.
- A characteristic of RISC processors is their ability to execute one instruction per clock cycle. This is done by overlapping the fetch, decode, and execute phases of two or three instructions by using a procedure referred to as pipelined.
- A load or store instruction may require two clock cycles because access to memory takes more time than register operations.
- Other characteristics attributed to RISC architecture are:
- 1. A relatively large number of registers in the processor unit
- 2. Efficient instruction pipeline
- 3. Compiler support for efficient translation of high-level language programs into machine language programs