Unit 3
Central Processing Unit

By- Yadvir Kaur

Contents

- General Register Organization
- Stack Organization
- Instruction Formats
- Addressing Modes
- Data Transfer and Manipulation
- Program Control
- Reduced Instruction Set Computer & Complex Instruction Set Computer

Major components of CPU

The CPU is made up of three major parts:

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

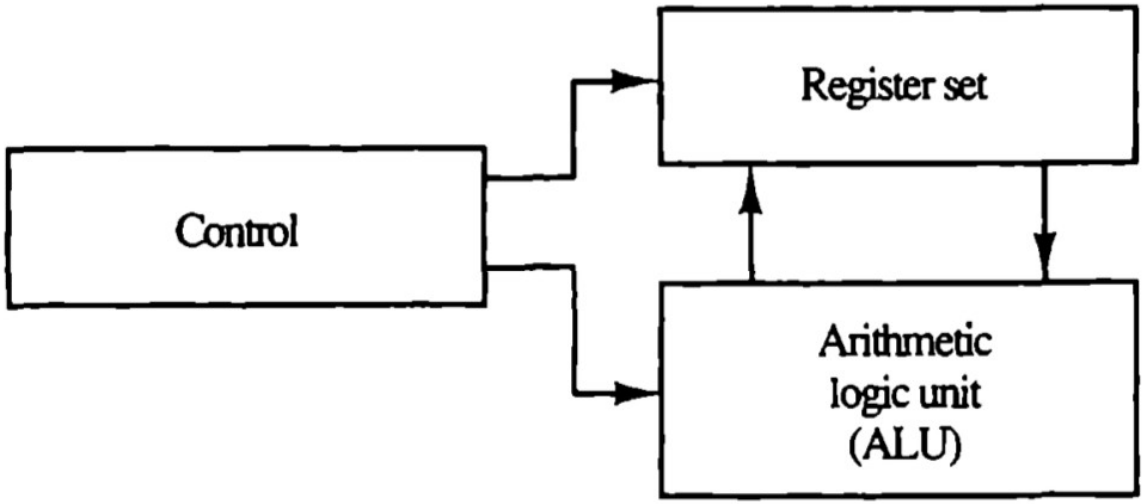


Figure 8·1 Major components of CPU.

- On basis of structure and behaviour (instruction formats, addressing modes, instruction sets), computer architecture is of two types:

  RISC

  CISC

- On basis of memory usage, there are two architectures:

  Embedded: This is basically Harward architecture in which programs and data resides in different memory systems. Eg: Micocontroller-based systems and digital signal processor based systems.

  Non-embedded: In this programs and data resides in same memory system. Eg:all desktop systems such as personal computers,

General Register Organization

Registers:

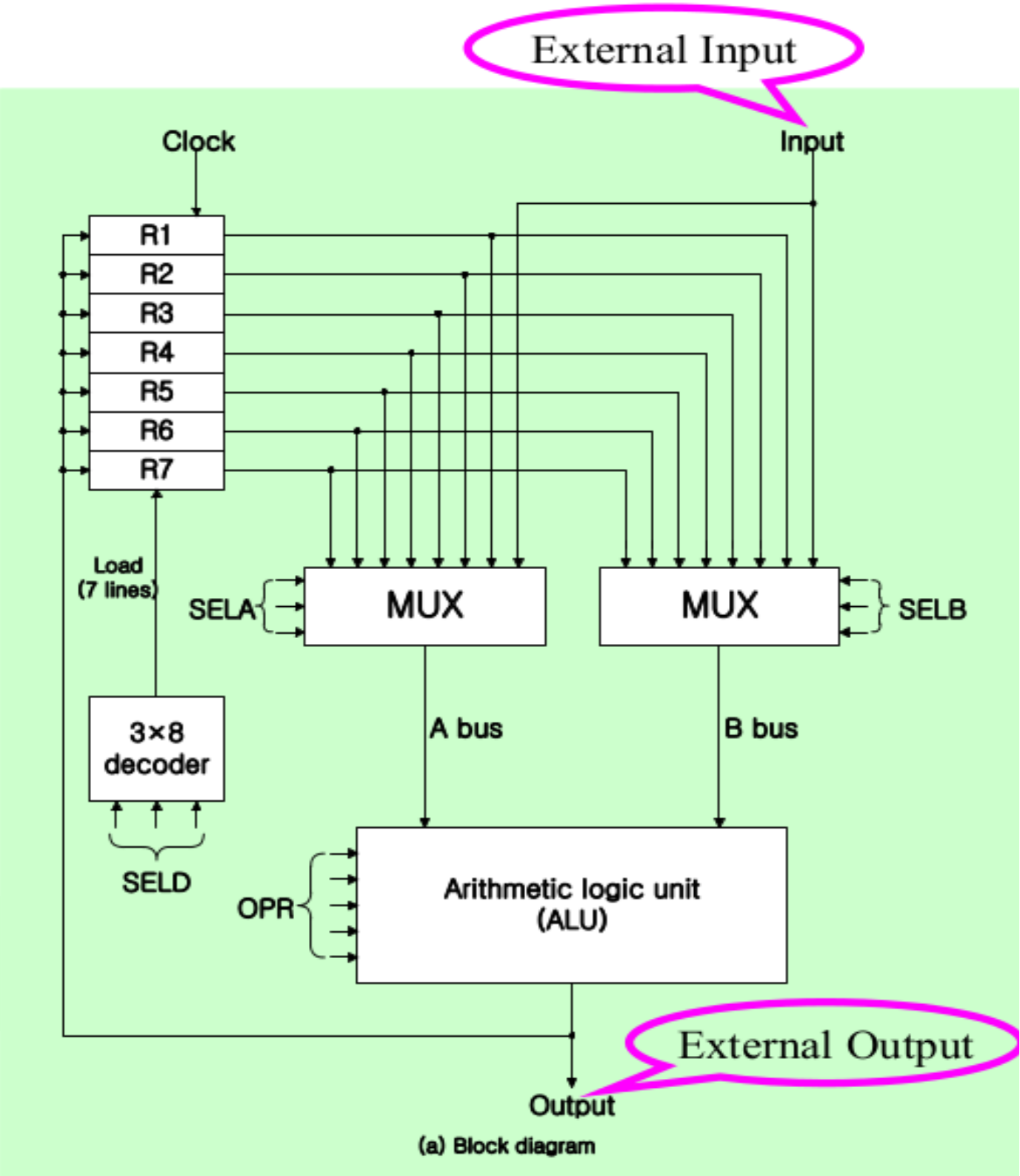
- Memory locations are needed for storing pointers, counters, return address, temporary results, and partial products during multiplication.
- Memory access is the most time-consuming operation in a computer.
- More convenient and efficient way is to store intermediate values in processor registers.

Bus organization for 7 CPU registers is shown in Fig. 8-2.

- **2 multiplexers (MUX):** The selection lines in each multiplexer select one of 7 register or the externa input data by SELA and SELB for the particular bus.
- **BUS A and BUS B:** form the inputs to a common arithmetic logic unit (ALU).
- **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.
- **3 X 8 Decoder:** select the register (by SELD) that receives the information from ALU

Figure 8.2:
Register set with
common ALU



3	3	3	5
SELA	SELB	SELD	OPR

(b) Control word

General Register Organization

The control unit that operates the CPU bus system directs the information flow through the registers and ALU by selecting the various components in the system. For example, to perform the operation:

R 1 <-- 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 o f 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 .

General Register Organization

Control Word:

- 14 bit control word consists of 4 fields:
  - » SELA (3 bits) : select a source register for the A input of the ALU
  - » SELB (3 bits) : select a source register for the B input of the ALU
  - » SELD (3 bits) : select a destination register using the 3 X 8 decoder
  - » OPR (5 bits) : select one of the operations in the ALU
- Encoding of Register Selection Fields : (Tab. 8-1) The register selected by fields SELA, SELB, and SELD is the one whose decimal number is equivalent to the binary number in the code.
  - » SELA or SELB = 000 (Input) : MUX selects the external input data
  - » SELD = 000 (None) : no destination register is selected but the contents of the output bus are available in the external output

Encoding of Register Selection Fields(SELA, SELB, and SELD): Tab. 8-1

Encoding of ALU Operation (OPR) : Tab. 8-2

TABLE 8-1 Encoding of Register Selection Fields

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	*R*1	*R*1	*R*1
010	*R*2	*R*2	*R*2
011	*R*3	*R*3	*R*3
100	*R*4	*R*4	*R*4
101	*R*5	*R*5	*R*5
110	*R*6	*R*6	*R*6
111	*R*7	*R*7	*R*7

TABLE 8-2 Encoding of ALU Operations

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

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 rnicrooperation given by the statement

R1 <-- R2 - R3

specifies R2 for the A input of the ALU, R3 for the B input of the ALU, R1 for the destination register, and an ALU operation to subtract A - B.

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

- The increment and transfer rnicrooperations 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
- The direct transfer from input to output is accomplished with a control word of all 0' s. A register can be cleared to 0 with an exclusive-OR operation. This is because $x \oplus x = 0$

TABLE 8-3 Examples of Microoperations for the CPU

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

Stack Organization

Stack or last-in, first-out (LIFO) list:

- A stack is a storage device that stores information.
- The item stored last is the first item retrieved = a stack of trays.

Stack Pointer (SP):

- The register that holds the address for the stack.
- SP always points at the top item in the stack.

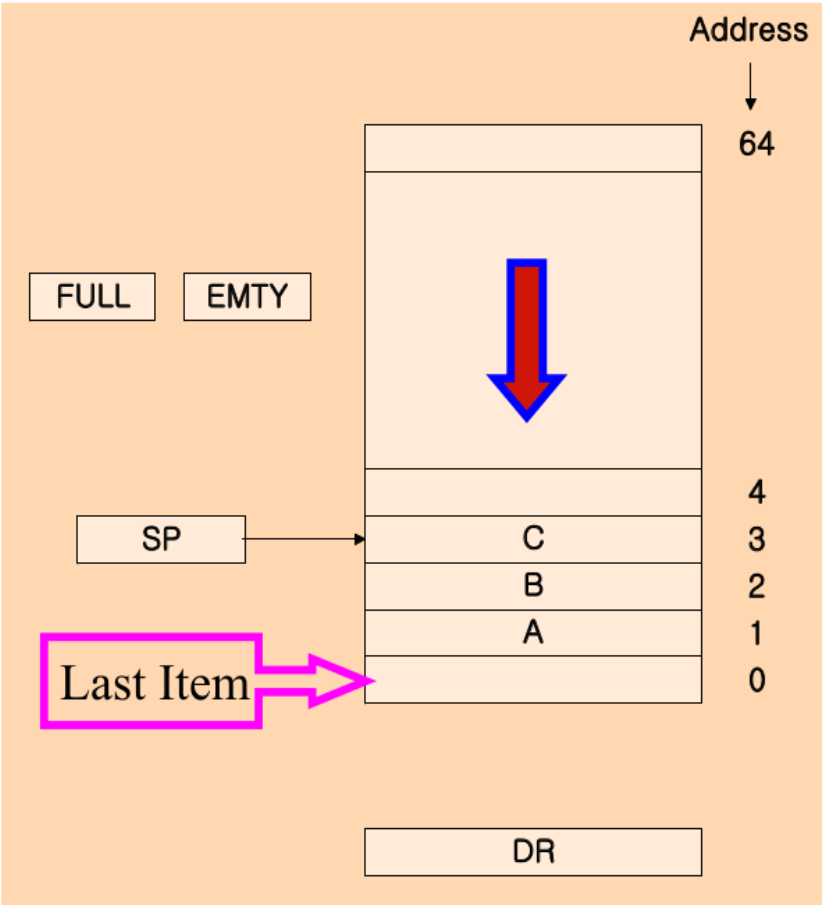
Two Operations of a stack : Insertion and Deletion of Items

- PUSH : Push-Down = Insertion
- POP : Pop-Up = Deletion

Stack

- 1) Register Stack: a collection of a finite number of memory words or register.
- 2) Memory Stack : a portion of a large memory

Register Stack: Figure shows the organization of a 64-word register stack.



- The SP contains a binary number whose value is equal to the address of the word that is currently on top of the stack. Item C is on top of the stack so that the content of SP is now 3.
- The one-bit register FULL is set to 1 when the stack is full
- 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.
- To remove the top item, the stack is popped by reading the memory word on top of the stack(at address 3) and decrementing the content of SP.
- To insert a new item, the stack is pushed by incrementing SP and writing a word in the next-higher location in the stack.
- In a 64-word stack, the stack pointer contains 6 bits because 2^6 = 64. So it cannot exceed a number greater than 63 (111111 in binary).
- When 63 is incremented by 1, the result is 0 since 111111 + 1 = 1000000 in binary, but SP can accommodate only the six least significant bits.
- Similarly, when 000000 is decremented by 1, the result is 111111 .

- 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 .

Push operation

- If the stack is not full (if FULL = 0), a new item is inserted with a push operation. 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

- Note that SP holds the address of the top of the stack and that M[SP] denotes the memory word specified by the address presently available in SP.
- The first item stored in the stack is at 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. (This condition is reached if the top item prior to the last push was in location 63 and, after incrementing SP, the last item is stored in location 0. Once an item is stored in location 0, there are no more empty registers in the stack.)

Pop operation

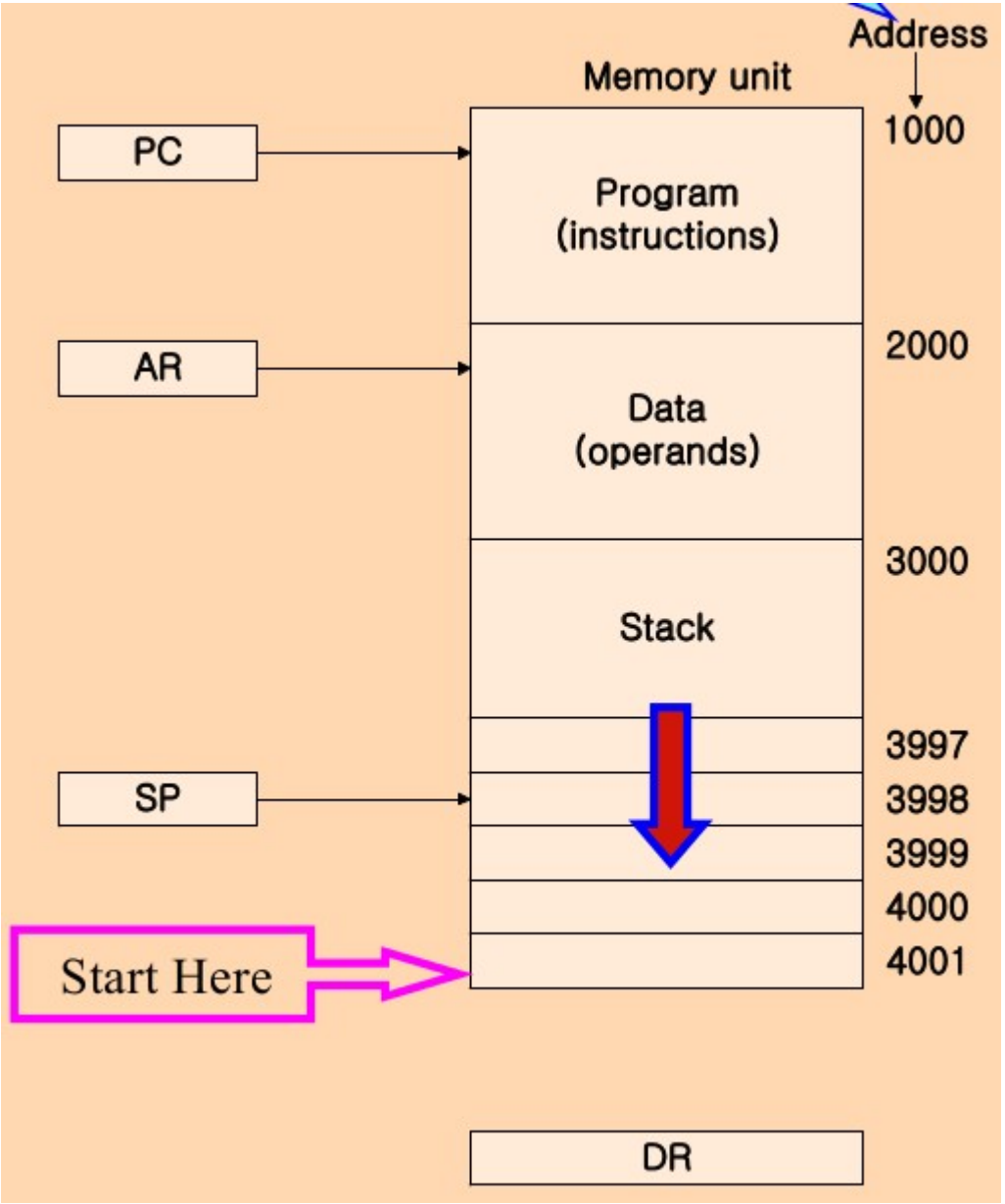
- A new item is deleted from the stack if the stack is not empty (if EMTY = 0). The pop operation consists of the following sequence of microoperations:

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

- The stack pointer is 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.
- Note that if a pop operation reads the item from location 0 and then SP is decremented, SP changes to 111111, which is equivalent to decimal 63 . In this configuration, the word in address 0 receives the last item in the stack.

Memory Stack

- 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.
- Figure shows a portion of computer memory partitioned into three segments: program, data, and stack.
- The program counter PC points at the address of the next instruction in the program.
- The address register AR points at an array of data.
- The stack pointer SP points at the top of the stack.



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

- In Fig, the initial value of SP is 4001 and the stack grows with decreasing addresses. Thus the 1st item stored in the stack is at address 4000, the 2nd item is stored at address 3999.
- **Push operation:** 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.
- **Pop operation:** 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.
- **Stack limits** : checked by using two processor registers: upper limit and lower limit register.

  » After a push operation: SP is compared with the upper-limit register

  » After a pop operation: SP is compared with the lower-limit register.
- SP is automatically decremented or incremented with every push or pop operation. The advantage of a memory stack is that the CPU can refer to it without having to specify an address, since the address is always available and automatically updated in the stack pointer.

Reverse Polish Notation

- The common mathematical method of writing arithmetic expressions imposes dif ficulties when evaluated by a computer. The common arithmetic expressions are written in infix notation, with each operator written between the operands:

  A * B + C * D

- **Prefix notation**: Arithmetic expressions can be represented in prefix notation, referred to as Polish notation, places the operator before the operands.
- **Postfix notation:** referred to as reverse Polish notation (RPN), places the operator after the operands.
- Three representations:

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

- A stack organization is very effective for evaluating arithmetic expressions. 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 * +

Evaluation of Arithmetic Expressions

- Reverse Polish notation, combined with a stack arrangement of registers, is the most efficient way known for evaluating arithmetic expressions.
- This procedure is employed in some electronic calculators and also in some computers.
- The procedure consists of first converting the arithmetic expression into its equivalent reverse Polish notation.
- The operands are pushed into the stack in the order in which they appear.
- The following microoperations are executed with the stack when an operation is entered:

  (1) the two topmost operands in the stack are used for the operation

  (2) the stack is popped and the result of the operation replaces the lower operand.

- Consider the arithmetic expression:

  (3 * 4) + (5 * 6)

- In reverse Polish notation, it is expressed as:

  34 * 56 * +

- Now consider the stack operations shown in Fig. Each box represents one stack operation and the arrow always points to the top of the stack.

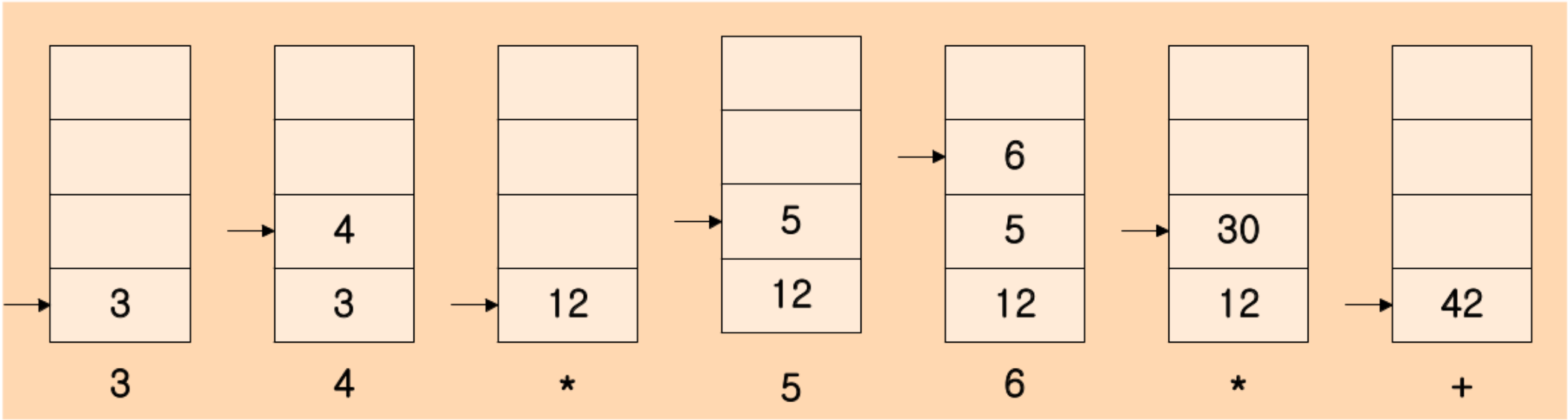


Fig: Scack operations to evaluate 3 * 4 + 5 * 6

Instruction Formats

Fields in Instruction Formats:

1) **Operation Code Field**: specify the operation to be performed
2) **Address Field**: designate a memory address or a processor register
3) **Mode Field**: specifies the way the operand or the effective address is determined (Addressing Mode)

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 (one address instructions)

2.General register organization (two or three address instructions)

3.Stack organization (zero-address operation instructions)

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

$$X = (A + B) \bullet (C + D)$$

We will use the symbols for:

» 4 arithmetic operations : ADD, SUB, MUL, DIV

» 1 transfer operation to and from memory and processor register: MOV

» 2 transfer operation to and from memory and AC register : STORE, LOAD

» Operand memory addresses : A, B, C, D

» Result memory address : X

Three-Address Instructions

Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand.

The program to evaluate X = (A + B) * (C + D) is as follows:

ADD	R1 , A, B	R 1 <-- M [A] + M [B]
ADD	R2 , C , D	R 2 <-- M [C] + M [D]
MUL	X , R1 , R2	M [X] <-- R 1 • R 2

It is assumed that the computer has two processor registers, R 1 and R2.

Advantage of the three-address format is that it results in short programs when evaluating arithmetic expressions.

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.

Eeach address field can specify either a processor register or a memory word.

The program to evaluate X = (A + B) * (C + D) is as follows:

MOV	R1, A	R 1 <-- M [A]
ADD	R1, B	R 1 <-- R 1 + M [B]
MOV	R2, C	R 2 <-- M [C]
ADD	R2, D	R 2 <-- R 2 + M [D]
MOL	R1, R2	R 1 <-- R 1 * R 2
MOV	X, R1	M [X] <-- R 1

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

One-Address Instructions

- One-address instructions use an implied accumulator (AC) register for all data manipulation.

  The program to evaluate X = (A + B) • (C + D) is

LOAD	A	A C <-- M [A]
ADD	B	A C <-- A C + M [B]
STORE	T	M [T] <-- A C
LOAD	C	A C <-- M [C]
ADD	D	A C <-- A C + M [D]
MOL	T	A C <-- A C • M [T]
STORE	X	M [X] <-- A C

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

Zero-Address Instructions

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

PUSH	A	T O S <-- A
PUSH	B	T O S <-- B
ADD		TO S <-- (A + B)
PUSH	C	T O S <-- C
PUSH	D	T O S <-- D
ADD		TO S <-- (C + D)
MOL		T O S <-- (C + D) * (A + B)
POP	X	M [X] <- T O S

- To evaluate arithmetic expressions in a stack computer, it is necessary to convert the expression into reverse Polish notation.
- The name "zero-address" is given to this type of computer because of the absence of an address field in the computational instructions.

Addressing Modes

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 mode field is used to locate the operands needed for the operation. There may or may not be an address field in the instruction or may have more than one address field, and each address field may be associated with its own particular addressing mode.

Implied Mode

In this mode the operands are specified implicitly in definition of the instruction.

Examples:

» **COM** (Complement Accumulator): Operand in AC is implied in the definition of the instruction

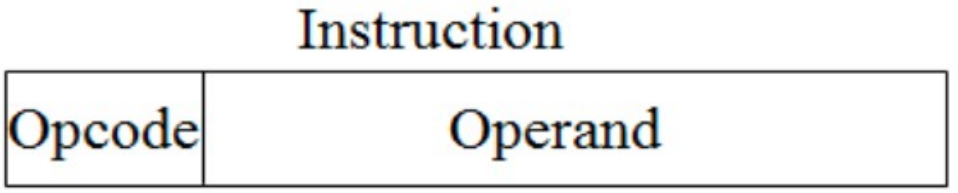
» **PUSH**: (Zero-address instructions in a stack-organisation): Operand is implied to be on top of the stack

Immediate Mode

- In this mode the operand is specified in the instruction itself.
- Useful for initializing registers to a constant value.

  Example : Add R4, #3 meaning: R4 <- R4 + 3

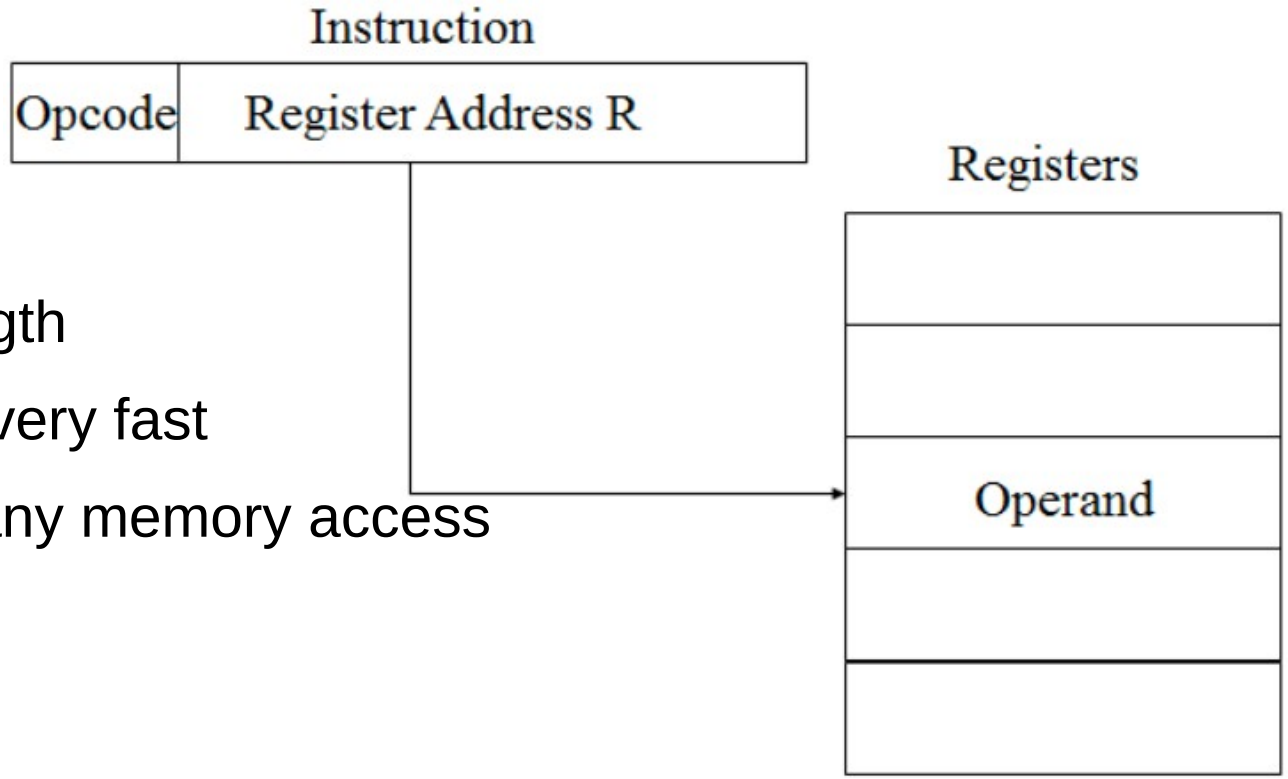
- No memory reference is required to fetch data.
- This results in a faster instruction.



Register Mode

- Operand is held in register(R) named in address field.
- The particular register is selected from a register field in the instruction. A k-bit field can specify any one of 2^k registers.
- **Example : Add R4,R3** meaning: R4 <-- R4 + R3

  LD R1 meaning: AC ← R 1



- A shorter instruction length
- Instruction execution is very fast
- This does not requires any memory access

Register Indirect Mode

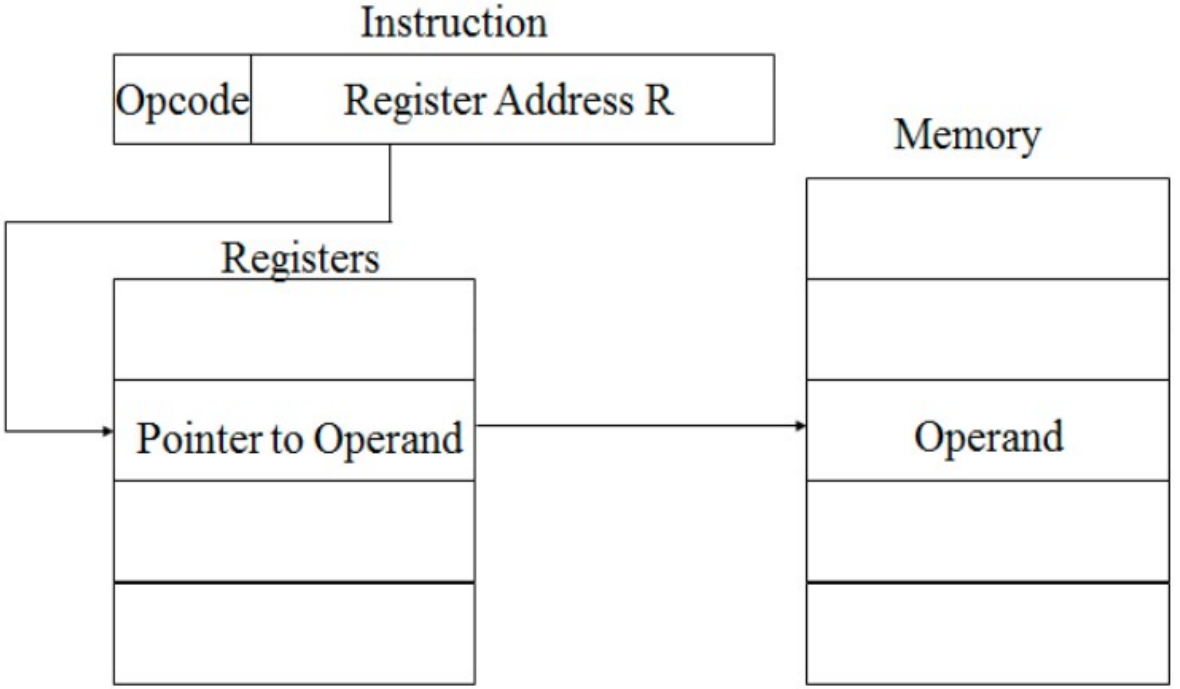
In this mode, instruction specifies a register which contains the memory address(effective address) of the operand.

It saves instruction bits since register address is shorter than the memory address.

Example: Add R4,(R1) meaning: R4 <- R4 + M[R1]

LD (R1) meaning: AC ← M [R 1]

It requires one memory refrence. Used for implementing pointers.



Autoincrement Mode

Similar to the register indirect mode except that

» the register is incremented after its value is used to access memory

Example: Add R1, (R2)+ meaning: R1 <- R1 +Mem[R2], R2 <-- R2 + d

LD (R1)+ meaning: AC ← M [R 1], R 1 ← R 1 + 1

- It requires one memory refrence.
- Post increment : (R2)+
- Used to implement stack

Autodecrement Mode

» the register is decrement before its value is used to access memory

Example: Add R1,-(R2) meaning: R2 <-R2-d, R1 <-- R1 + Mem[R2]

- It requires one memory refrence
- Pre decrement -(R2)
- Used to implement stack

Direct Address Mode

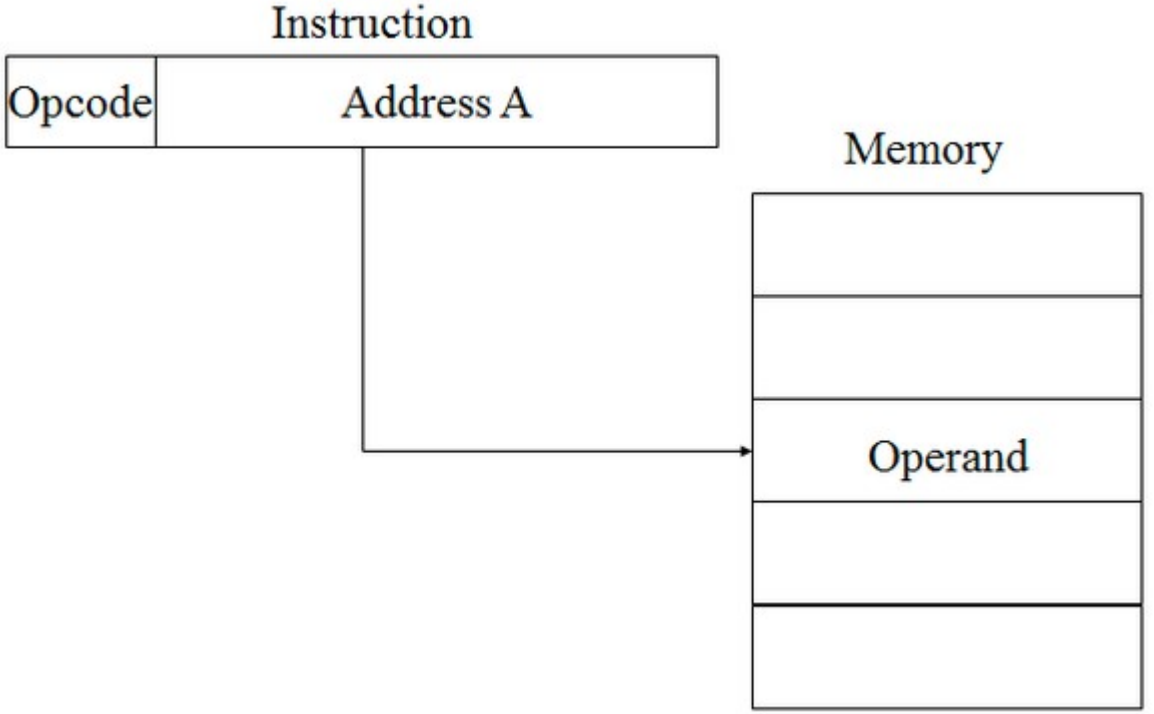
Address field contains effective address of the operand.

Example: Add R1, ADR meaning: R1 <- R1 + Mem[ADR]

LD ADR meaning: C ← M [ADR]

ADR: Address part of instruction

- Single memory reference is required to access data.
- No additional calculations are required to work out effective address.



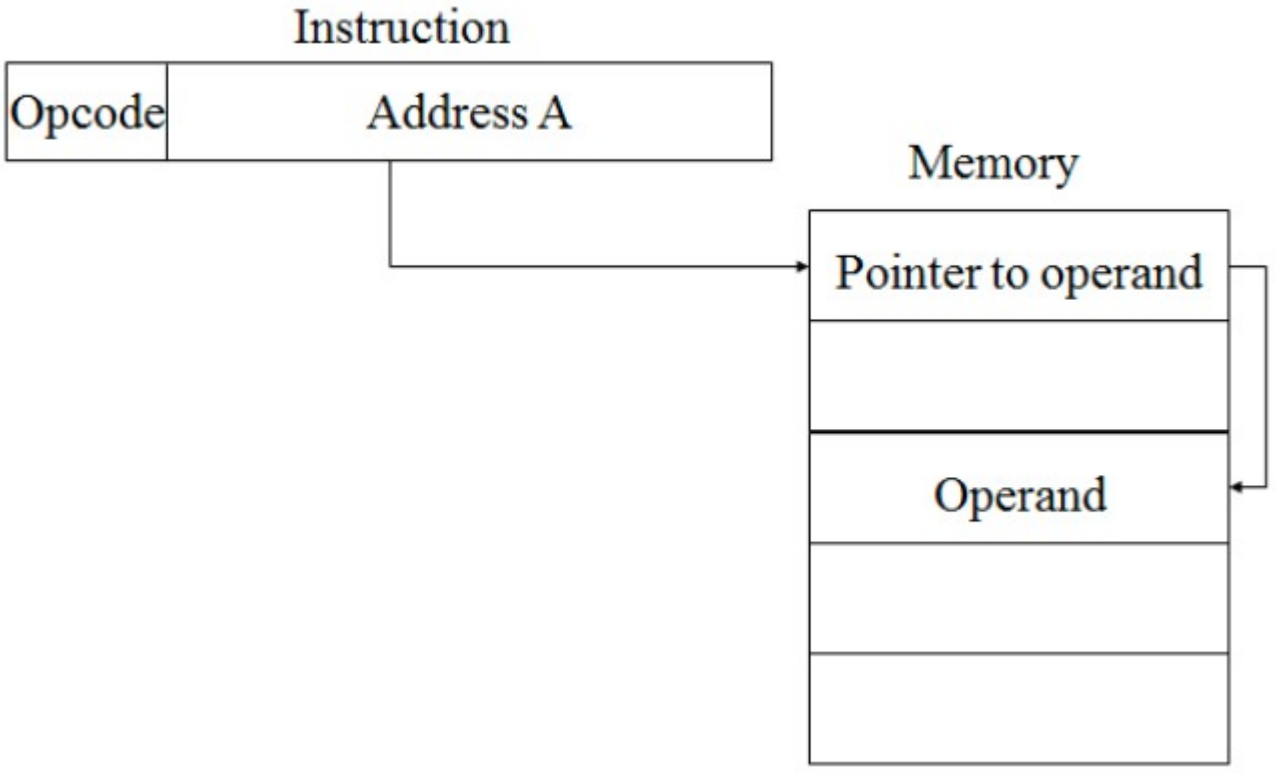
Indirect Address Mode

The address field of an instruction specifies the address of a memory location say A, that contains the address of the operand.

Example: Add R1, @ADR meaning: R1 <- R1 + Mem[Mem[ADR]]

LD @ADR meaning: AC ← M [M [ADR]]

- Two memory refrences are required to access the operand.
- Used for implementing the pointers.



Relative Address Mode

PC is added to the address part of the instruction to obtain the effective address.

Example: LD $ADR meaning: AC ← M [PC + ADR]

- It is a version of displacement addressing.
- Used for program relocation at runtime.
- Used in branch type instruction.

Indexed Addressing Mode

XR (Index register) is added to the address part of the instruction to obtain the effective address

Index register hold an index number that is relative to the address part of the instruction

Example : LD ADR(XR) meaning: AC ← M [ADR + XR]

- Useful in array addressing.

Base Register Addressing Mode

The content of a base register is added to the address part of the instruction to obtain the effective address.

Similar to the indexed addressing mode except that the register is now called a base register instead of an index register.

» base register hold a base address

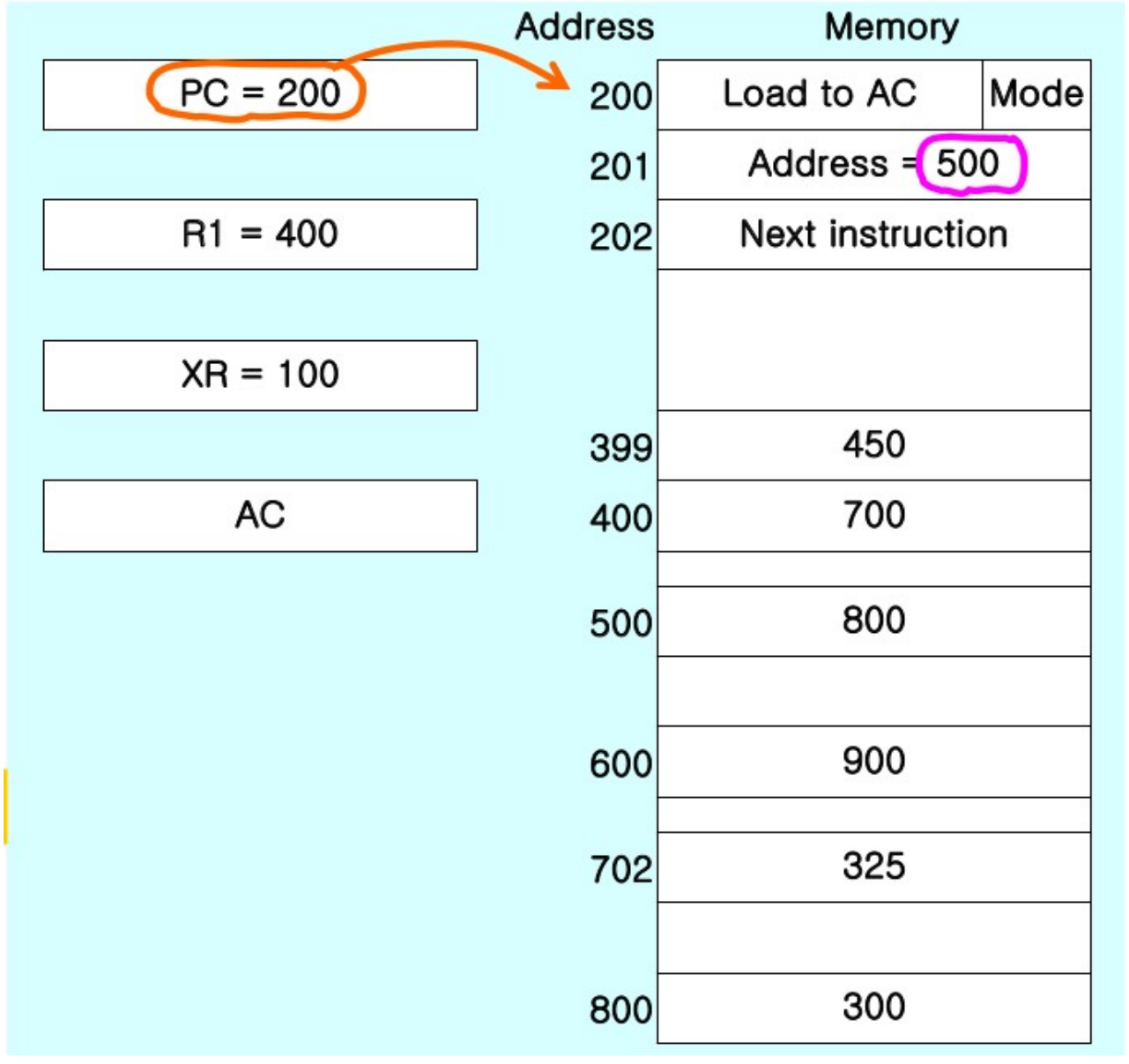
» the address field of the instruction gives a displacement relative to this base address

base register (BR) : LD ADR(BR) meaning: AC ← M [BR + ADR]

- This mode is used in computers to facilitate the relocation of programs in memory.

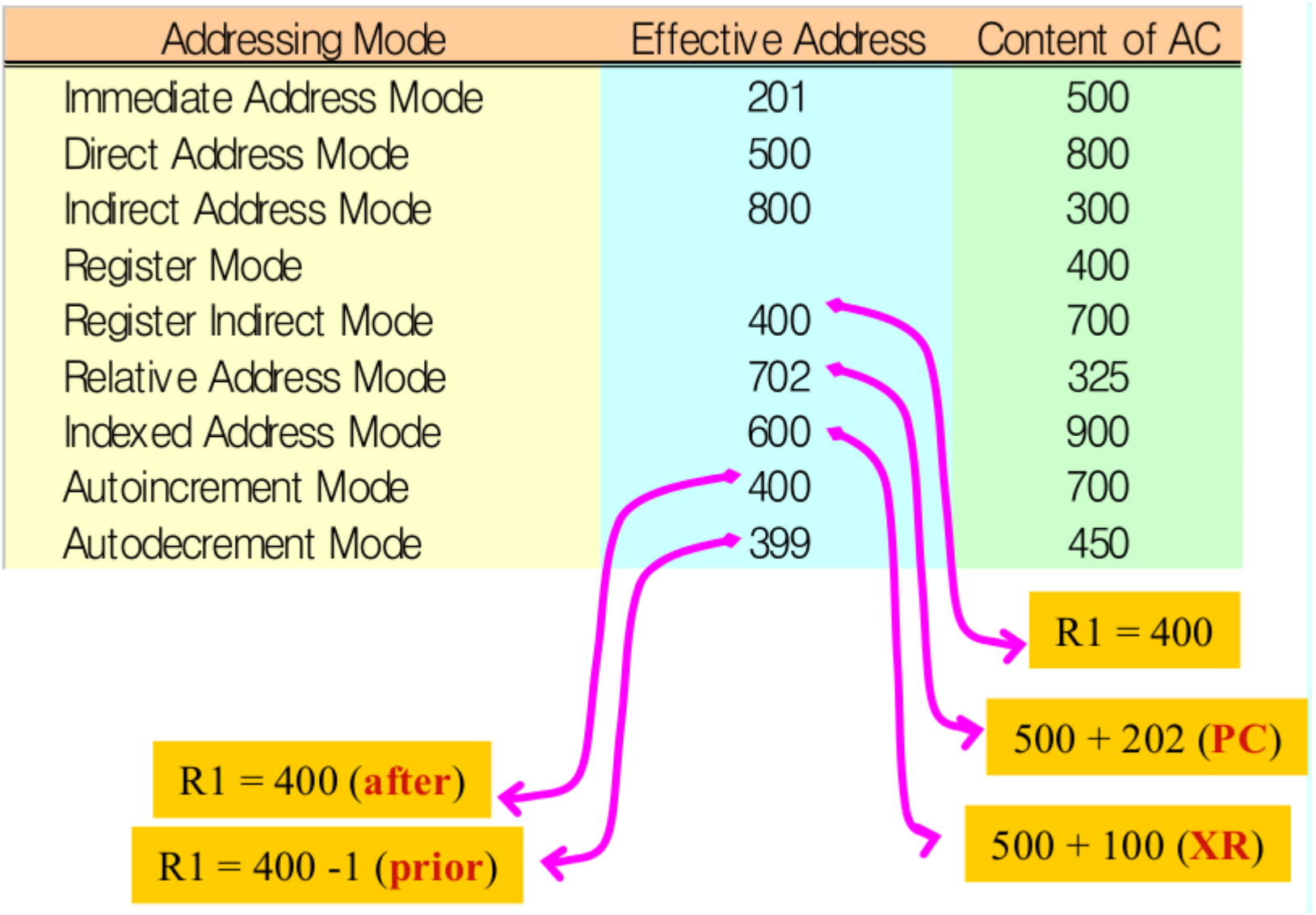
Numerical Example

The two-word instruction at address 200 and 201 is a "load to AC" instruction with an address field equal to 500.



Solution:

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



Program Control

Program control instructions specify conditions for altering the content of the program counter.

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.

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

- **Branch** and **jump** instructions may be conditional or unconditional. An unconditional branch instruction causes a branch to the specified address with out any conditions. The conditional branch instruction specifies a condition such as branch if positive or branch if zero.
- The **skip** instruction does not need an address field and is therefore a zero-address instruction. A conditional skip instruction will skip the next instruction if the condition is met.
- The **cal**l and **return** instructions are used in conjunction with subroutines.
- The **compare** instruction performs a subtraction between two operands, but the result of the operation is not retained. However, certain status bit conditions are set as a result of the operation.(numerical example on slide 41)
- 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

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.

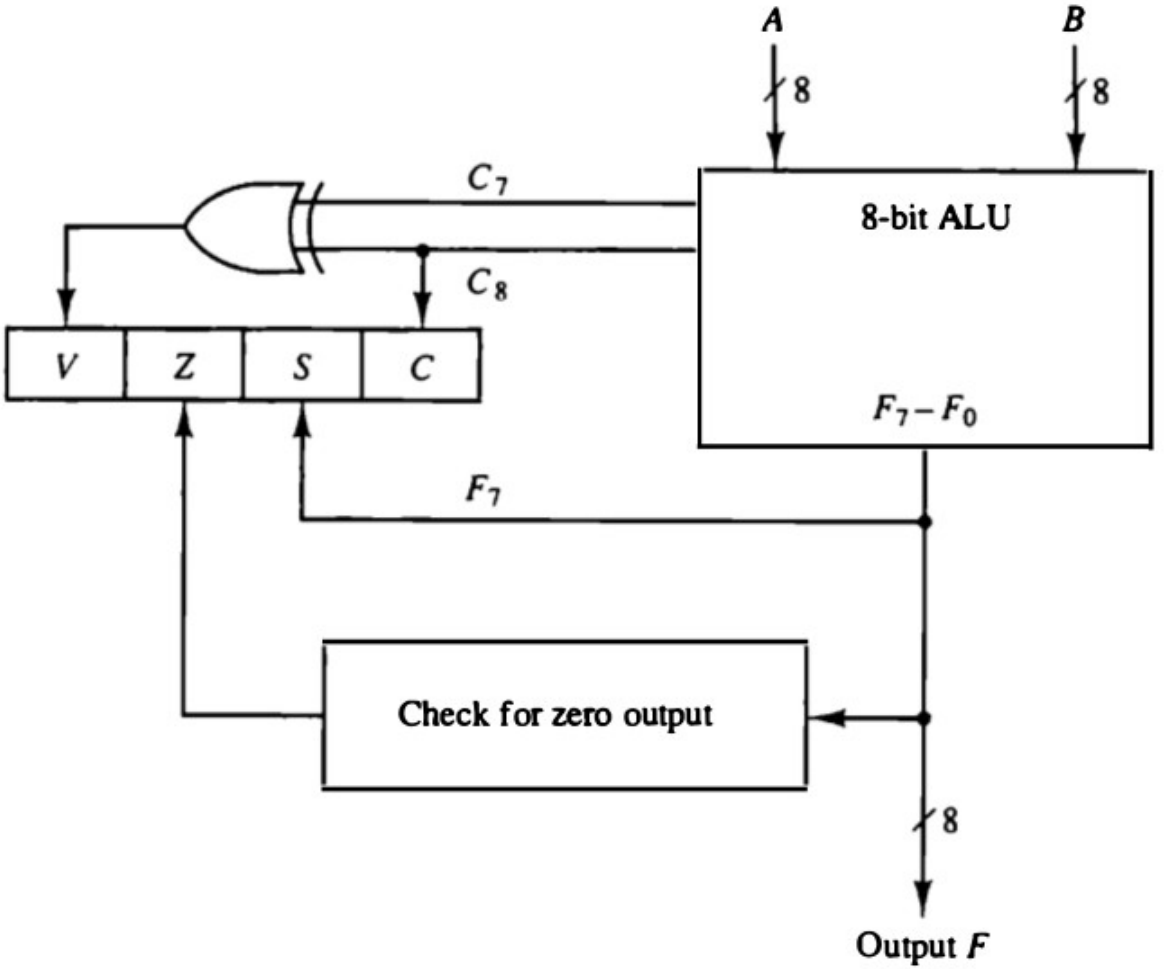


Figure 8-8 Status register bits.

Status Bit Conditions

- Bit C (carry) : set to 1 if the end carry C 8 is 1
- Bit S (sign) : set to 1 if F 7 is 1
- Bit Z (zero) : set to 1 if the output of the ALU contains all 0's
- Bit V (overflow) : set to 1 if the exclusive-OR of the last two carries (C 8 and C 7) is equal to 1

Flag Example : A - B =A + (2's Comp. of B) : A =11110000, B = 00010100

11110000

+ 11101100 (2's comp. of B)

1 11011100 C = 1, S = 1, V = 0, Z = 0

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.

Conditional Branch Instructions

TABLE 8-11 Conditional Branch Instructions

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	$Z = 1$
BNZ	Branch if not zero	$Z = 0$
BC	Branch if carry	$C = 1$
BNC	Branch if no carry	$C = 0$
BP	Branch if plus	$S = 0$
BM	Branch if minus	$S = 1$
BV	Branch if overflow	$V = 1$
BNV	Branch if no overflow	$V = 0$
Unsigned compare conditions ($A - B$)		
BHI	Branch if higher	$A > B$
BHE	Branch if higher or equal	$A \geq B$
BLO	Branch if lower	$A < B$
BLOE	Branch if lower or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$
Signed compare conditions ($A - B$)		
BGT	Branch if greater than	$A > B$
BGE	Branch if greater or equal	$A \geq B$
BLT	Branch if less than	$A < B$
BLE	Branch if less or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$

Subroutine Call and Return

A call subroutine instruction consists of an operation code together with an address that specifies the beginning of the subroutine.

The instruction is executed by performing two operations:

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

CALL :

SP ← SP − 1 : Decrement stack point

M [SP] ← PC : Push content of PC onto the stack

PC ← Effective Address : Transfer control to the subroutine

RETURN :

PC ← M [SP] : Pop stack and transfer to PC

SP ← SP + 1 : Increment stack pointer

- 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

  » some store it in a **memory stack**. --> **most efficient way**

- The most efficient way is to store the return address in a memory stack, as explained below:

  » If only one register or memory location is used to store return address and recursive subroutines calls itself, it destroys the previous return address.

  » But if stack is used return each return address can be pushed into the stack without destroying any previous values.

Program Interrupt

» Transfer 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) An interrupt is initiated by an internal or external signal (except for software interrupt)

   A subroutine call is initiated from the execution of an instruction (CALL)

2) The address of the interrupt service program is determined by the hardware

   The address of the subroutine call is determined from the address field of an instruction

3) An interrupt procedure stores all the information necessary to define the state of the CPU

   A subroutine call stores only the program counter (Return address)

Types of Interrupts

1) **External Interrupts**: come from

» I/O device: I/0 device requesting transfer of data, I/0 device finished transfer of data

» from a timing device: elapsed time of an event

» from a circuit monitoring the power supply, or from any other external source

2) **Internal Interrupts or TRAP:** Occurs due to premature termination of the instruction execution. caused by

»register overflow ,attempt to divide by zero, stack overflow, protection violation

3) **Software Interrupts:** initiated by executing an instruction

» 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.

supervisor call instruction :provides means for switching from a CPU user mode to the supervisor mode.

» Allows to execute certain operations (in supervisor mode), which are not allowed in the user mode.

RISC v/s CISC

CISC	RISC
Complex Instruction Set Computer.	Reduced Instruction Set Computer.
A large number of instructions - typically from 100 to 250 instruction.	Relatively few instructions.
A large variety of addressing modes - typically from 5 to 20 different modes.	Relatively few addressing modes.
Variable-length instruction formats.	Fixed-length, easily decoded instruction format.
Requires multiple clock cycles to execute one instruction.	Each instruction requires only one clock cycle to execute.
Most of the instructions access memory.	Memory access limited to load and store instruction. All operations done within the registers of the CPU.
It has microprogrammed control uint.	It has Hardwired control uint.
It requires external memory for calculations.	It does not require external memory for calculations.
It has single register set.	It has multiple register set.
Less pipelined.	Highly pipelined.
Execution time is very high.	Execution time is very less.

Q: Write a program to evaluate the arithmetic statement:

$$X = \frac{A - B + C * (D * E - F)}{G + H * K}$$

a. Using a general register computer with three address instructions.

b. Using a general register computer with two address instructions.

c. Using an accumulator type computer with one address instructions.

d. Using a stack organized computer with zero-address operation instructions.

a) Three address instructions:

SUB	R1, A, B	R1 <-- M [A] - M [B]
MUL	R2, D, E	R2 <-- M [D] * M [E]
SUB	R2, R2, F	R2 <-- R2 – M [F]
MUL	R2, R2, C	R2 <-- R2*M [C]
ADD	R1, R1, R2	R1 <-- R1+R2
MUL	R3, H, K	R3 <-- M [H] + M [K]
ADD	R3, R3, G	R3 <-- R3+ M [G]
DIV	X, R1, R3	X <-- R1/R3

b) Two address instructions:

MOV	R1, A	R1 <-- M [A]
SUB	R1, B	R1 <-- R1-M [B]
MOV	R2, D	R2 <-- M [D]
MUL	R2, E	R2 <-- R2*M [E]
SUB	R2, F	R2 <-- R2 – M [F]
MUL	R2, C	R2 <-- R2*M[C]
ADD	R1, R2	R1 <-- R1+R2
MOV	R3, H	R3 <-- M [H]
ADD	R3, G	R3 <-- R3+M [G]
DIV	R1, R3	R1 <-- R1/R3
MOV	X, R1	M[X] <-- R1

c) One Address instructions:

LOAD	A	AC <-- M [A]
SUB	B	AC <-- AC-M [B]
STORE	T	M [T] <-- AC
LOAD	D	AC <-- M [D]
MUL	E	AC <-- AC*M [E]
SUB	F	AC <-- AC-M [F]
MUL	C	AC <-- AC*M[C]
ADD	T	AC <-- AC+M [T]
STORE	T	M [T] <-- AC
LOAD	H	AC <-- M [H]
MUL	K	AC <-- AC*M [K]
ADD	G	AC <-- AC+M [G]
STORE	T1	M [T1] <-- AC
LOAD	T	AC <-- M [T]
DIV	T1	AC <-- AC/M [T1]
STORE	X	M[X] <-- AC

d) Zero address instructions:

RPN: AB-CDE*F-*+GHK*+/

PUSH A	TOS <-- A
PUSH B	TOS <-- B
SUB	TOS <-- (A-B)
PUSH C	TOS <-- C
PUSH D	TOS <-- D
PUSH E	TOS <-- E
MUL	TOS <-- (D*E)
PUSH F	TOS <-- F
SUB	TOS <-- ((D*E)-F)
MUL	TOS <-- C*((D*E)-F)
ADD	TOS <-- ((A-B)+ C*((D*E)-F)
PUSH G	TOS <-- G
PUSH H	TOS <-- H
PUSH K	TOS <-- K
MUL	TOS <-- (H*K)
ADD	TOS <-- G+(H*K)
DIV	TOS <-- ((A-B)+ C*((D*E)-F)/(G+(H*K))
POP	X M[X] <-- TOS

Practice Questions

- Differentiate between RISC and CISC architectures.
- What is an instruction format?
- Define control word.
- What is the role of registers in computers?
- What do you know about Relative Address Mode?
- Give example of effective address in relation to computer architecture.
- Differentiate BUN and BSA instructions along with suitable example
- Define the following: Micro-instruction and Micro-program
- Write 1-address and zero address instruction for expression: (A+B)*(C+D) .
- Differentiate between JMP and CALL instructions.
- Explain the overflow condition in arithmetic shift microoperation?

Practice Questions

- Explain the basic computer instruction formats by giving examples.
- Elaborate the various types of data transfer operations with examples.
- Write Short Notes on: (i) RISC vs CISC (ii) Addressing Modes
- Write Short Note on:

  a) Data Transfer Operations

  b) Input/Output Interface

- Discuss the three-address, two-address, one-address and zero-address instructions in detail. Give examples of these instructions.

Numerical Questions

Q1: Specify the control word that must be applied to the processor of Fig. 8-2 to implement the following microoperations.

a. $R1 \leftarrow R2 + R3$
b. $R4 \leftarrow R4$
c. $R5 \leftarrow R5 - 1$
d. $R6 \leftarrow \text{shl } R1$
e. $R7 \leftarrow \text{input}$

Q2: Determine the microoperations that will be executed in the processor of Fig. 8-2 when the following 14-bit control words are applied.

a. 00101001100101
b. 00000000000000
c. 01001001001100
d. 00000100000010
e. 11110001110000

Numerical Questions

Q3: Let SP=000000 in the stack. How many items are there in the stack if:

a. FULL = 1 and EMTY = 0?

b. FULL = 0 and EMTY = 1?

Q4: Convert the following numerical arithmetic expression into reverse Polish notation and show the stack operations for evaluating the numerical result.

$$(3 + 4)[10(2 + 6) + 8]$$

Q5: An instruction is stored at location 300 with its address field at location 301. The address field has the value 400 . A processor register R1 contains the number 200. Evaluate the effective address if the addressing mode of the instruction is (a) direct; (b) immediate; (c) relative; (d) register indirect (e) index with R1 as the index register.

Numerical Solutions

Q1:

		SELA	SELB	SELD	OPR	Control word
(a)	R1 ← R2 + R3	R2	R3	R1	ADD	010 011 001 00010
(b)	R4 ← $\overline{R4}$	R4	—	R4	COMA	100 xxx 100 01110
(c)	R5 ← R5 –1	R5	—	R5	DECA	101 xxx 101 00110
(d)	R6 ← SH1 R1	R1	—	R6	SHLA	001 xxx 110 11000
(e)	R7 ← Input	Input	—	R7	TSFA	000 xxx 111 00000

Q2: **8.4**

	Control word	SELA	SELB	SELD	OPR	Microoperation
(a)	001 010 011 00101	R1	R2	R3	SUB	R3 ← R1 – R2
(b)	000 000 000 00000	Input	Input	None	TSFA	Output ← Input
(c)	010 010 010 01100	R2	R2	R2	XOR	R2 ← R2 ⊕ R2
(d)	000 001 000 00010	Input	R1	None	ADD	Output ← Input + R1
(e)	111 100 011 10000	R7	R4	R3	SHRA	R3 ← shrR7

Numerical Solutions

Q3: (a) Stack full with 64 items.

(b) stack empty

Q4: (3 + 4) [10 (2 + 6) + 8] = 616

RPN : 34 + 2 6 + 10 * 8 + *

				6		10		8		
	4		2	2	8	8	80	80	88	
3	3	7	7	7	7	7	7	7	7	616
3	4	+	2	6	+	10	*	8	+	*

Numerical Solutions

Q5:

Effective address

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