# Central Processing Unit

By,
Er. Nabaraj Bahadur Negi

## Contents

## 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
- 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.

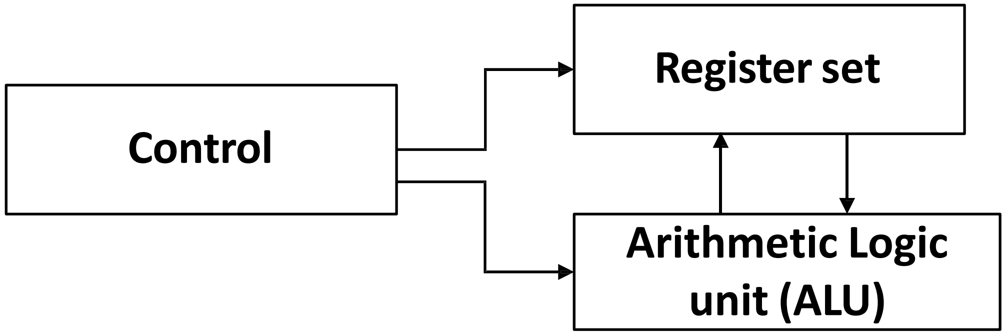
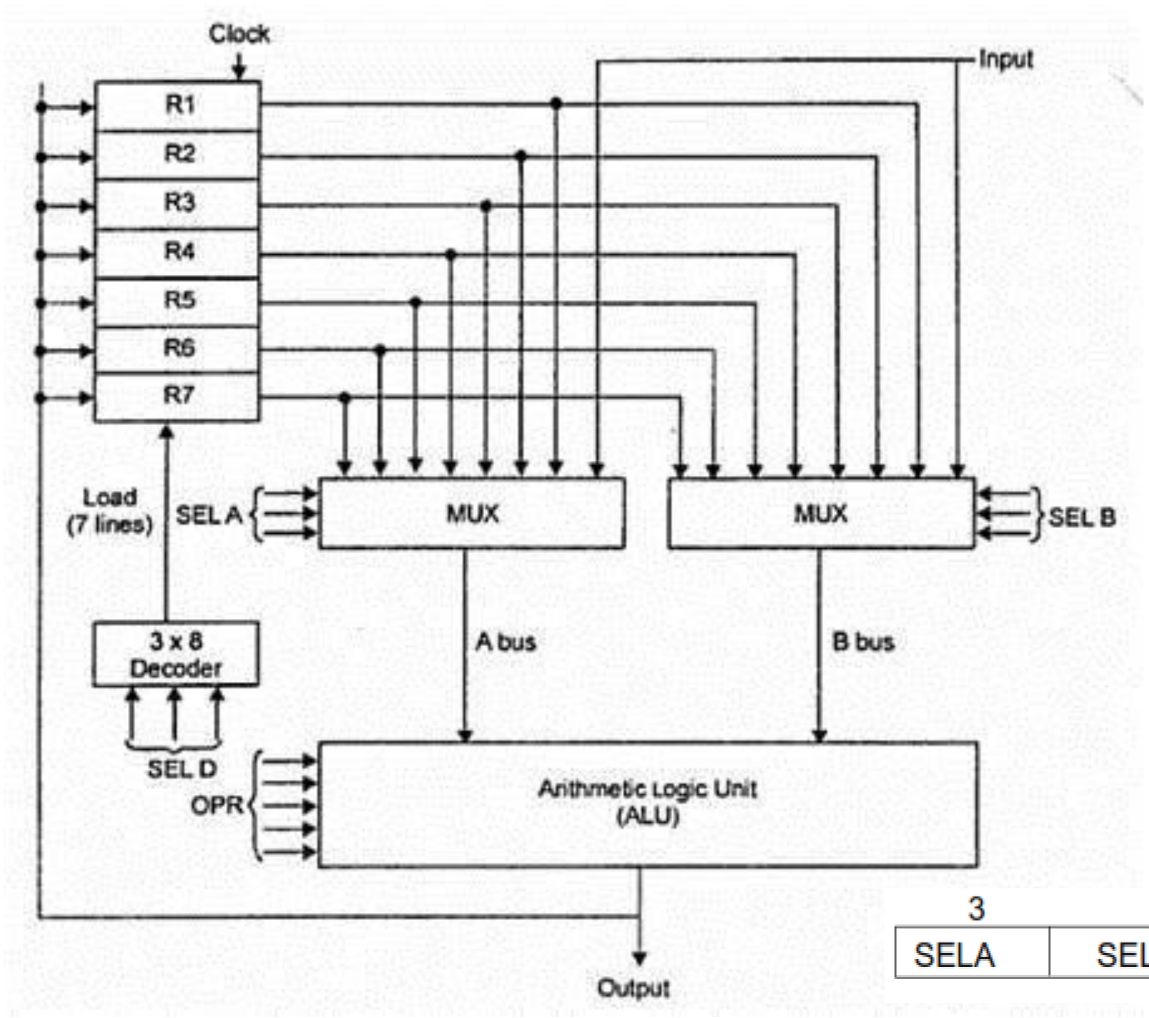


**Figure : Major components of CPU.**

# General Register Organization



| 3 | 3 | 3 | 5 |
|---|---|---|---|
| SELA | SELB | SELD | OP |

**Figure (a) : Block diagram (register Organization).** **Figure (b) : control word.**

- During instruction execution, we have shown that memory locations are needed for storing pointers, counters, return addresses, temporary results, and partial products during multiplication.
- 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 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 figure. 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.

- For example, to perform the operation:

$$R1 \leftarrow R2 + R3$$

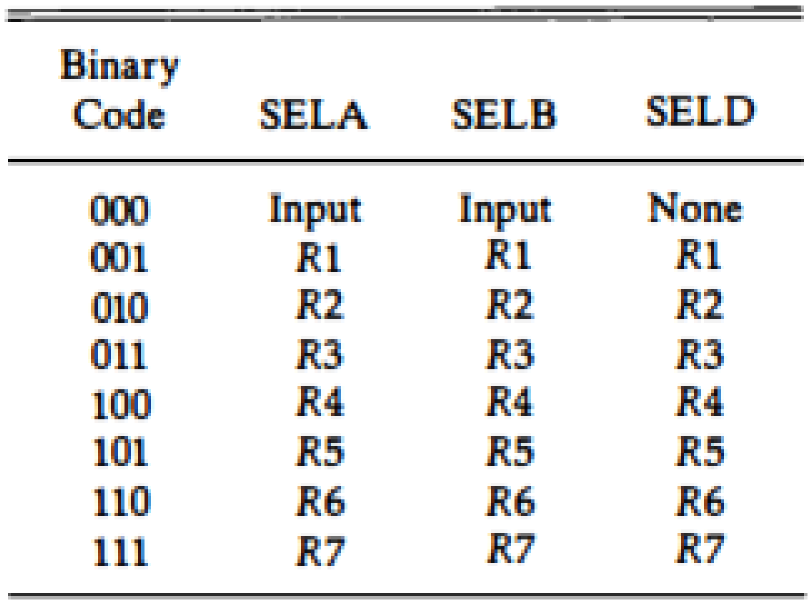
The control must provide binary selection variables to the following selector inputs:

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:**

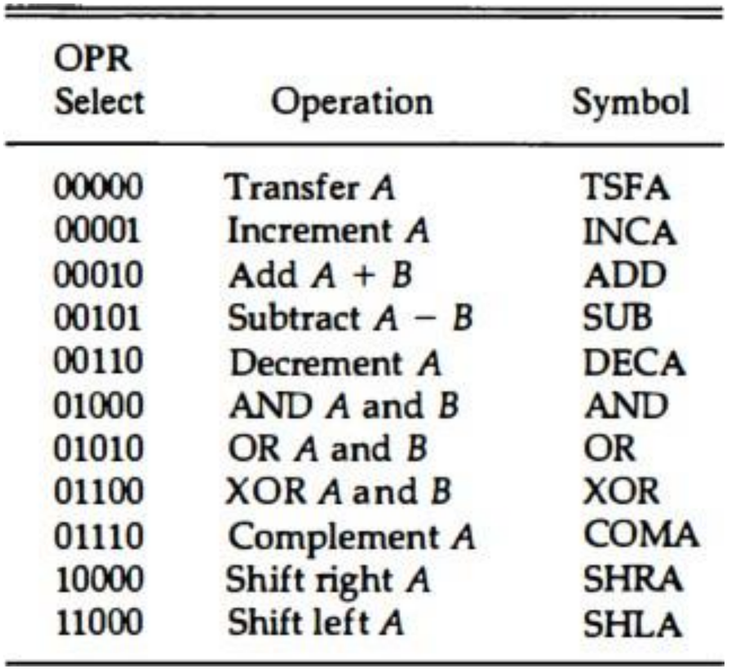
- There are 14 binary selection inputs in the unit, and their combined value specifies a control word. The 14-bit control word is defined in Fig. (b).
- It consists of four fields. Three fields contain three bits each, and one field has five bits.
- The three bits of SELA select a source register for the A input of the ALU. The three bits of SELB select a register for the B input of the ALU.

- The three bits 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 microoperation.



| 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 : Encoding of Register Selection Fields.**



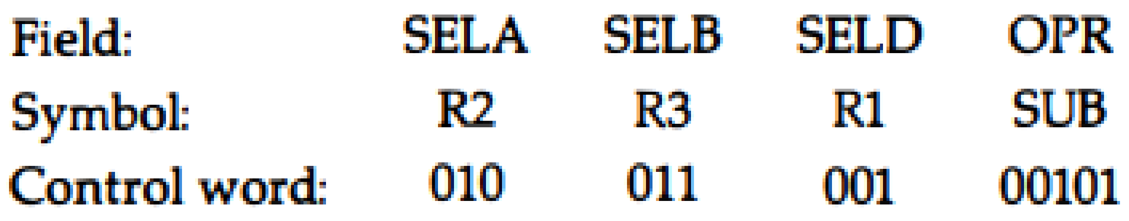
| OPR Select | Operation | Symbol |
|---|---|---|
| 00000 | Transfer *A* | TSFA |
| 00001 | Increment *A* | INCA |
| 00010 | Add *A* + *B* | ADD |
| 00101 | Subtract *A* − *B* | SUB |
| 00110 | Decrement *A* | DECA |
| 01000 | AND *A* and *B* | AND |
| 01010 | OR *A* and *B* | OR |
| 01100 | XOR *A*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

**Table : Encoding of ALU Operations.**

For example, the subtract microoperation given by the statement:

$R1 \leftarrow R2 - R3$

- Statement 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 microoperation is 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 control word for this microoperation and a few others are listed in below:

|  | 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 |

**Table : Examples of Microoperations for the CPU .**

## Stack Organization

- A useful feature that is included in the CPU of most computers is a stack or last-in, first-out (LIFO) list.
- The two operations of a stack are the insertion and deletion of items.
- The operation of insertion is called push (or push-down) and deletion is called (or pop-up) of items.
- 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.

**Register stack:**

- A stack can be placed in portion of a large memory or it can be organized as a collection of a finite number of memory words or registers.
- Following figure shows the organization of a 64-word register stack, the stack pointer contains 6 bits because $2^6 = 64$.
- 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 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.

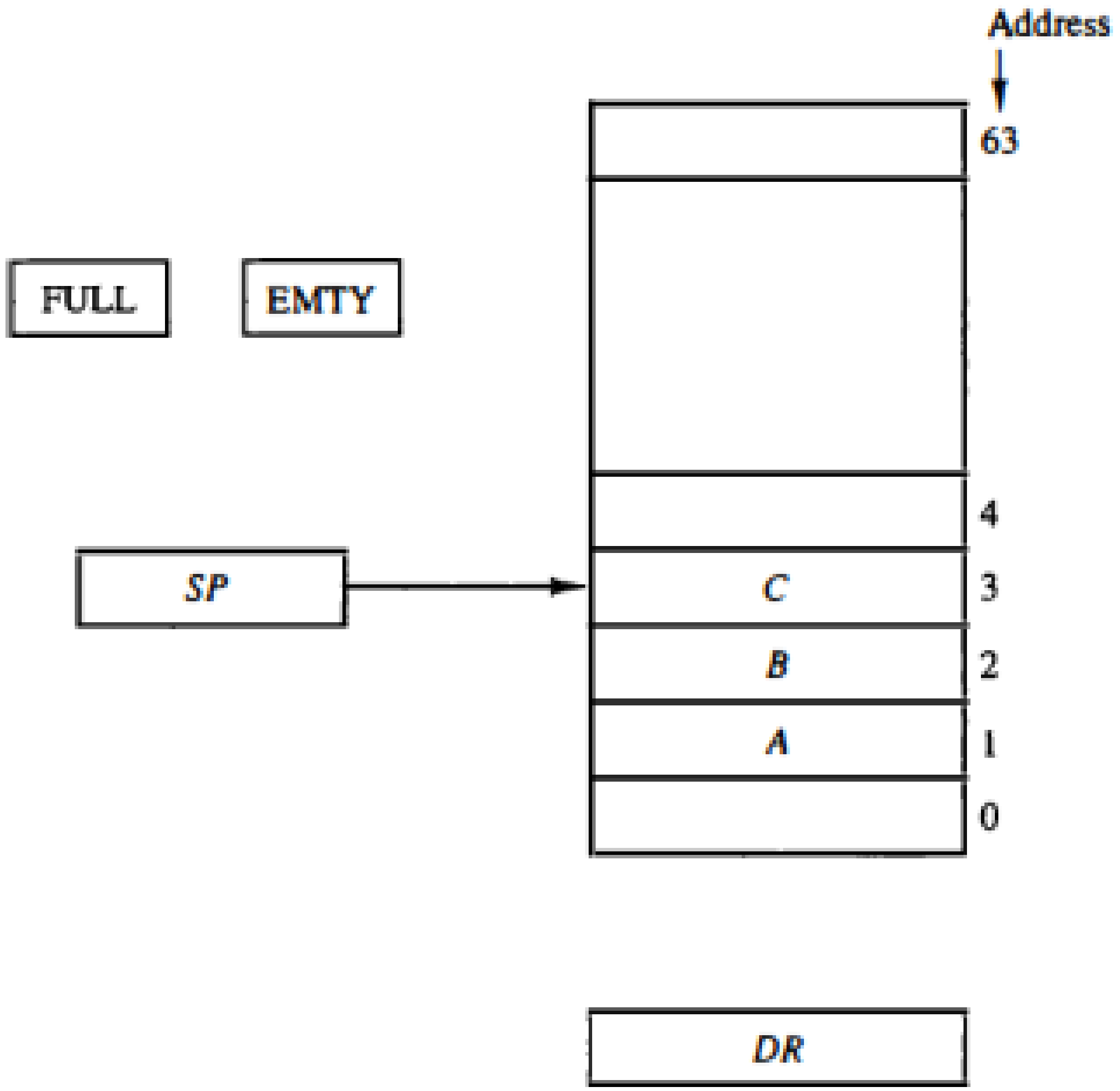


**Figure : Block diagram of a 64-word stack.**

- In a 64-word stack ,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 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. The one bits 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
```

- 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
SP ← SP – 1                    //Decrement stack pointer
If (SP = 0) then (EMTY ← 1)    //Check if stack is empty
FULL ← 0                       //Mark the stack not full
```

## Memory Stack:

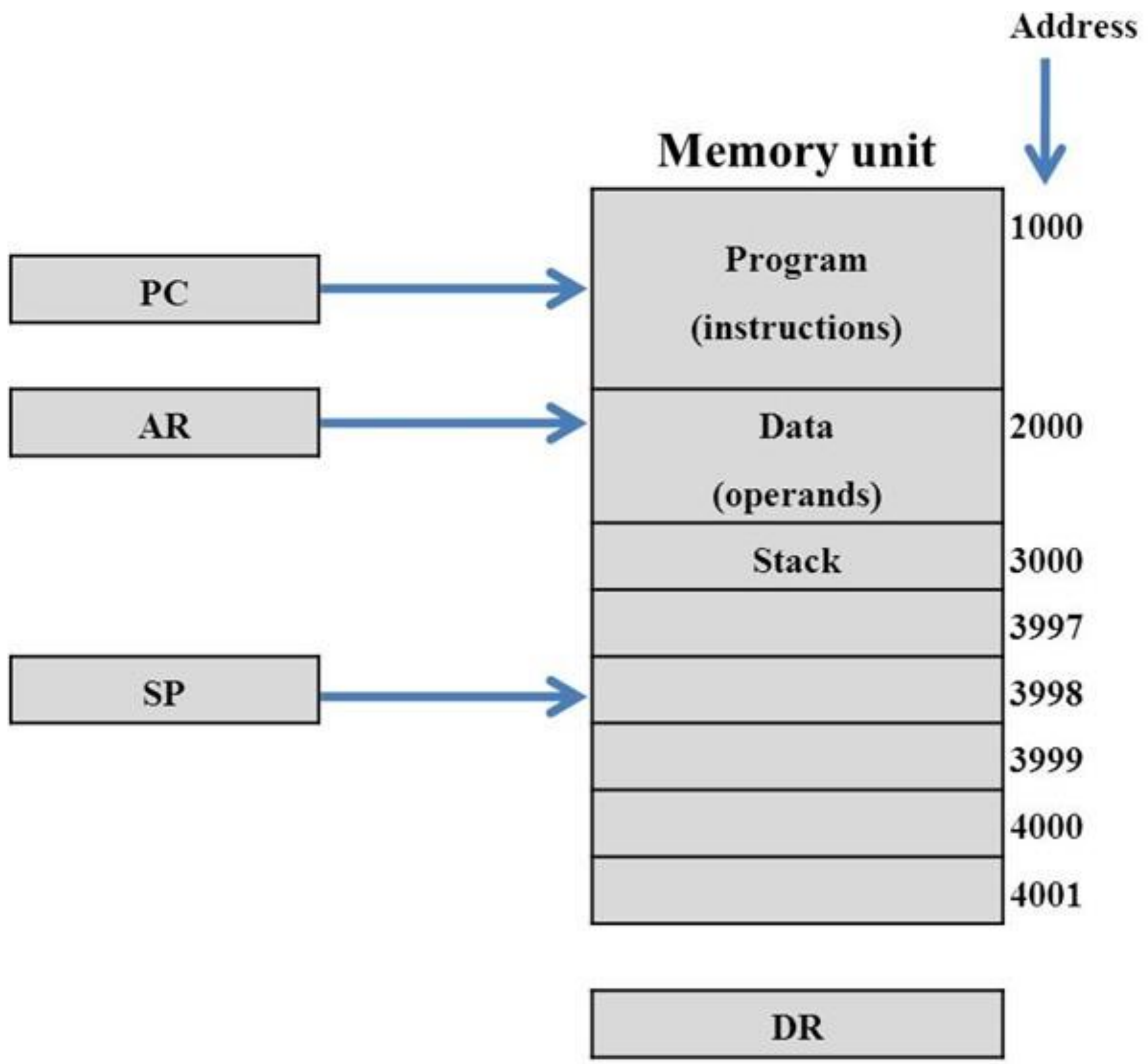


**Figure : Computer memory with program, data and stack segments.**

- 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. 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
- 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.

**PUSH :**

$SP \leftarrow SP - 1$

$M[SP] \leftarrow DR$

- The stack pointer is decremented so that it points at the address of the next word. A memory write operation inserts the word form DR into the top of the stack.

**POP :**

$DR \leftarrow M[SP]$

$SP \leftarrow SP + 1$

- 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.

Types of CPU organizations:

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

**Single Accumulator Organization :**

- The first ALU operand is always stored into the Accumulator and the second operand is present either in Registers or in the Memory.
- Accumulator is the default address thus after data manipulation the results are stored into the accumulator.
- One address instruction is used in this type of organization.

The format of instruction is: Opcode + Address

Examples:

ADD X //AC $\leftarrow$AC+M[X]

LDAY // AC $\leftarrow$M[Y]

**General register organization:**

- The computer uses two or three address fields in their instruction format. Each address field may specify a general register or a memory word.
- Any of the registers can be used as the source or destination for computer operations

Examples:

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

**Stack organization:**

- The stack is a list of data words.
- two operations that are performed on the operators of the stack are Push and Pop**.** (TOS =Top of stack.)

Examples:

PUSH X // TOS $\leftarrow$ M[X]

ADD // TOS=TOP(S) + TOP(S)

## Types of Instruction

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.

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

**Three-Address Instructions:**

- Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand.
- The program in assembly language that evaluates X = (A + B) * (C + D) is shown below, together with comments that explain the register transfer operation of each instruction.

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

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

## Two-address instructions:

- Two address instructions are the most common in commercial computers. Here again each address field can specify either a processor register or a memory word.
- The program to evaluate X = (A + B) * (C + D) is as follows:

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

- The MOV instruction moves or transfers the operands to and from memory and processor registers.
- Tries to minimize the size of instruction
- Size of program is relative larger.

## 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.
- The program to evaluate 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
```

- Memory access is only limited to load and store
- Large program size

**Zero-address instructions:**

- A stack-organized computer does not use an address field for the instructions ADD and MUL .
- Stack organization uses this type of instructions.
- The following program shows how X = (A + B) * (C + D) will be written for a stack organized computer.

```
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
```

- 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

- The operation field of an instruction specifies the operation to be performed.
- This operation 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.
- 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.

**Implied Mode :**

- In this mode the operands are specified implicitly in the definition of the instruction.
- Zero address instructions in a stack organized computer are implied mode instructions since the operands are implied to be on top of the stack.

E.g. CMA (complement accumulator)
CRC(clear carry flag)
INC A(increment accumulator)
Add

**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.

```
E.g. LD #NBR      // AC ←NBR
LD R1, #1000      //R1 ← 1000
ADD R2, #3        //R2 ← R2 + 3
```

**Register Mode:**

- In this mode the operands are in registers that reside within the CPU. The particular register is selected from a register field in the instruction. A k-bit field can specify any one of $2^k$ registers.

| | |
|---|---|
| E.g. LD R1 | // AC $\leftarrow$ R1 |
| LD R1, R2 | // R1 $\leftarrow$ R2 |
| ADD R1, R2 | //R1 $\leftarrow$ R1 + R2 |

**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.

| | |
|---|---|
| E.g. LD (R1) | // AC ← M[R1] |
| LD R1, (R2) | // R1 ← M[R2] |
| ADD R1, (R2) | // R1 ← R1 + M[R2] |
| ADD R1, (R2) | //R1 ← R1 + M[R2] |

**Auto increment or Auto decrement Mode:**

- This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory.
- In auto increment mode, the content of CPU register is incremented by 1, which gives the effective address of the operand in memory.

| | |
|---|---|
| E.g. LD (R1)+ | // AC ← M[R1], R1 ← R1 + 1 |

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

  E.g. LD (R1)- // AC ← M[R1 - 1]

**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.

| | |
|---|---|
| E.g. LD ADR | // AC ← M[ADR] |
| LD R1, 1000 | //R1 ← M[1000] |
| ADD R2, 3 | //R2 ← R2 + M[3] |

**Indirect Address Mode:**

- In the 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.

| | |
|---|---|
| E.g. LD @ADR | // AC ← M[M[ADR]] |
| LD R1, @1000 | //R1 ← M[M[1000]] |
| ADD R1, @(1000) | //R1 ← R1 + M[M[1000]] |
| ADD R1, (1000) | //R1 ← R1 + M[M[1000]] |

**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.

- Assume that the PC is 500 and the address part of instruction is 50. The instruction at location 500 is read from memory during fetch phase and PC is then incremented by 1. Hence, PC is 501, then effective address is 501+50=551.

  E.g. LD $ADR // AC ← M[PC + ADR]

**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.
- This type of addressing mode is useful to access the data array, where the address field of an instruction gives the start address of data array and content of index register gives how far the operand from the start address is.

  E.g. LD ADR(X) // AC ← M[ADR + XR]

**Based 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.

  E.g. LD ADR(BR) // AC $\leftarrow$ M[ADR + BR]

## Numerical Example

- 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 specifies 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.

- **Direct address:** EA is the address part of the instruction 500 and the operand to be loaded into AC is 800.

- **Immediate mode:** the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into AC. (The EA in this case is 201.)
- **Indirect mode:** EA is stored in memory at address 500. Therefore, the EA is 800 and the operand is 300.
- **Relative mode :** EA is 500 + 202 = 702 and the operand is 325. (Note that the value in PC after the fetch phase and during the execute phase is 202).

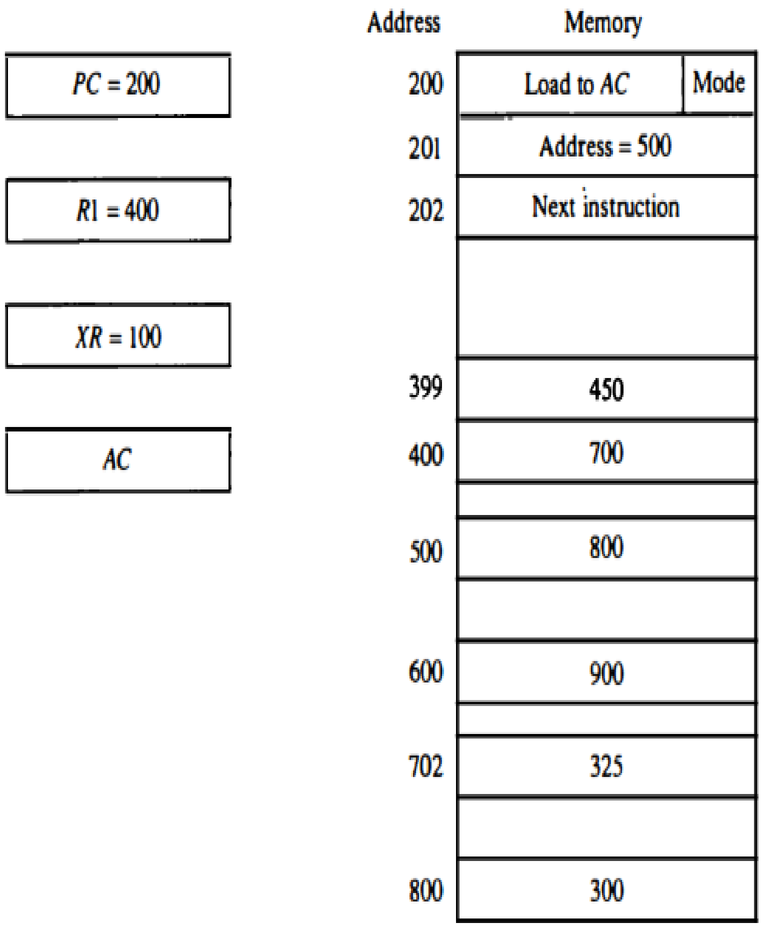
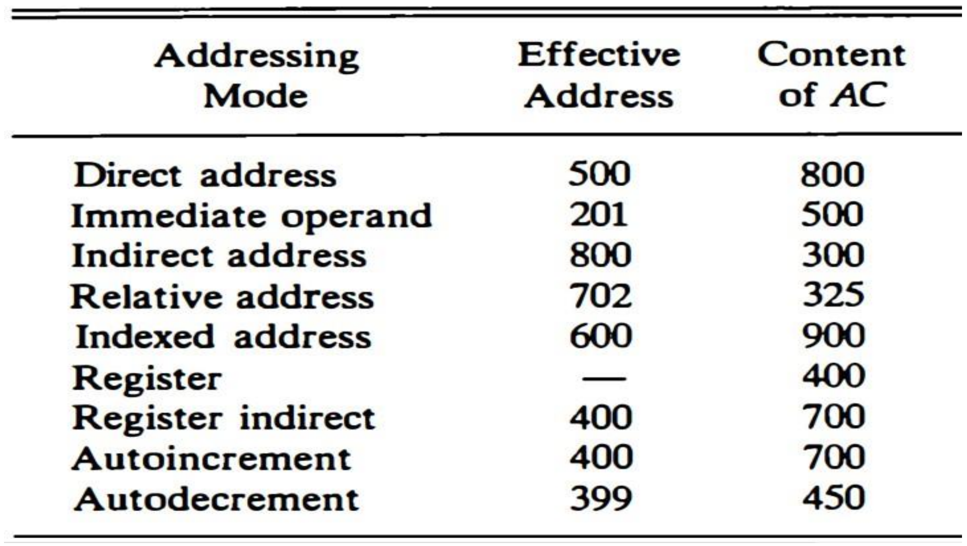


**Figure : Numerical example for addressing modes.**

- **Index mode :** EA is XR + 500 = 100 + 500 = 600 and the operand is 900.
- **Register mode:** the operand is in R1 and 400 is loaded into AC . (There is no EA in this case.)
- **Register indirect:** EA is 400, equal to the content of R1 and the operand loaded into AC is 700.
- **Autoincrement mode :** is the same as the register indirect mode except that R 1 is incremented to 401 after the execution of the instruction.
- **Autodecrement mode:** decrements R1 to 399 prior to the execution of the instruction. The operand loaded into AC is now 450.



| 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 |

**Table :Tabular List of Numerical Example.**

## Data Transfer and Manipulation

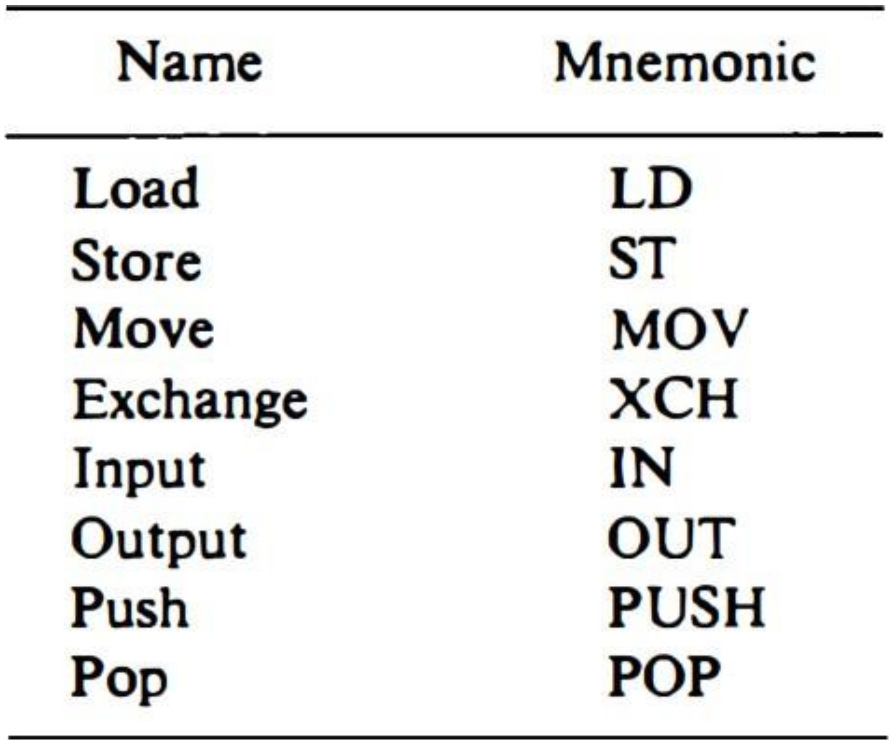
- Computers provide an extensive set of instructions to give the user the flexi- bility to carry out various computational tasks.
- The actual operations available in the instruction set are not very different from one computer to another. It so happens that the binary code assignments in the operation code field is different in different computers, even for the same operation.
- 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.

**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 gives a list of eight data transfer instructions used in many computers.
- Load: instruction designates a transfer from memory to a processor register, usually an accumulator.

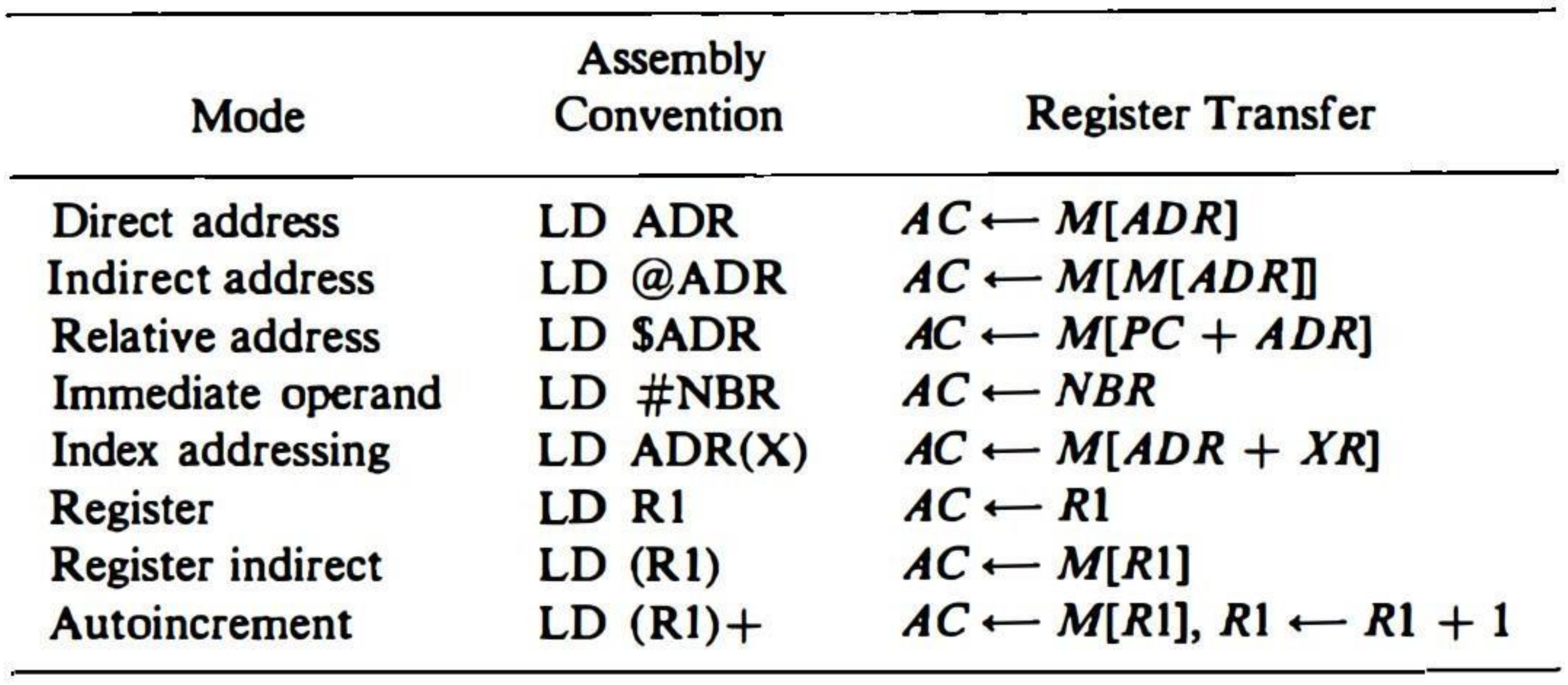
- Store: instruction designates a transfer from a processor register into memory.
- Move: transfer from one register to another, registers and memory or between two memory words.
- Exchange: instruction swaps information between two registers or a register and a memory word.
- Input and output : instructions transfer data among processor registers and input or output terminals.
- Push and pop: instructions transfer data between processor registers and a memory stack.



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

**Table :Typical Data Transfer Instructions.**

- 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.



| 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 : Eight Addressing Modes for the Load Instruction.**

- Table 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, R1 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.

## 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

| 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 |

**Table : Typical Arithmetic Instructions.**

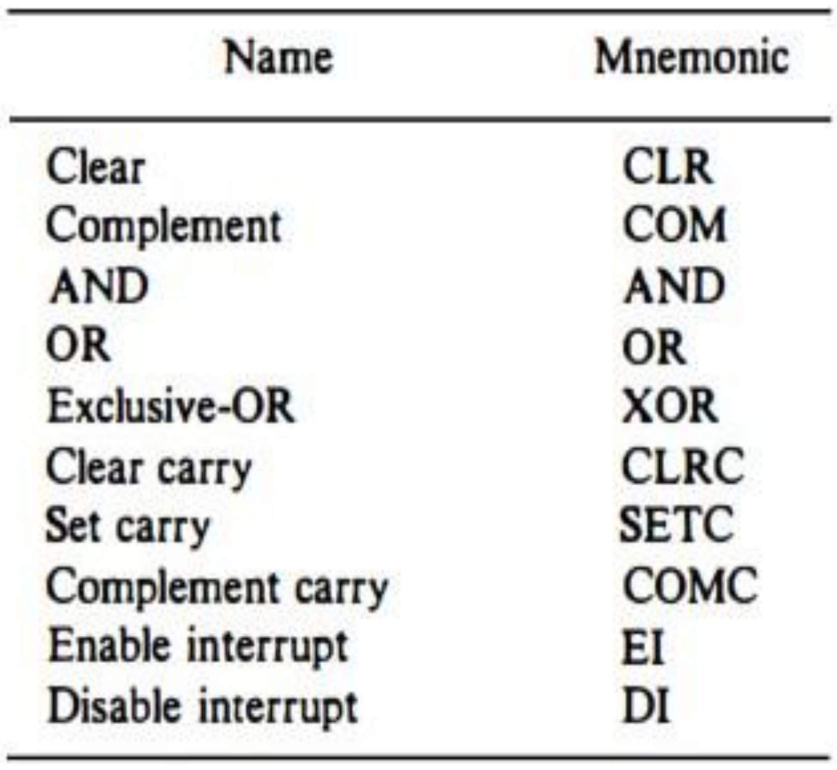
## Arithmetic instructions

- list of typical arithmetic instructions is given in Table.
- The increment instruction adds 1 to the value stored in a register or memory word.
- The add, subtract, multiply, and divide instructions may be available for different types of data.

## Logical and Bit Manipulation Instructions

- Logical instructions perform binary operations on strings of bits stored in registers and 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.

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



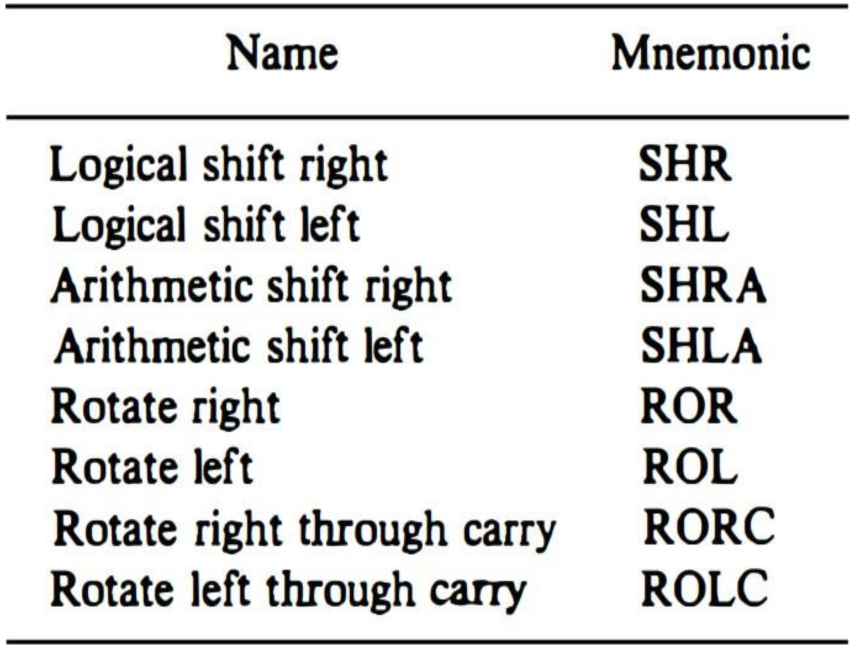
| Name | Mnemonic |
|---|---|
| Clear | CLR |
| Complement | COM |
| AND | AND |
| OR | OR |
| Exclusive-OR | XOR |
| Clear carry | CLRC |
| Set carry | SETC |
| Complement carry | COMC |
| Enable interrupt | EI |
| Disable interrupt | DI |

**Table : Typical Logical and 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
  - rotate-type operations
- Table shows lists four types of shift instructions.
- The logical shift inserts 0 to the end bit position(end position is the leftmost bit for shift right and the rightmost bit position for the shift left).

- The arithmetic shift-left instruction inserts 0 to the end position and is identical to the logical shift-left instruction and arithmetic shift-right bit unchanged (should preserve the sign).
- The rotate instructions produce a circular shift.
- The rotate-left through carry instruction transfers the carry bit into the rightmost bit of position the register, the transfers leftmost bit position into the carry, and at the same time, shifts the entire register to the left.



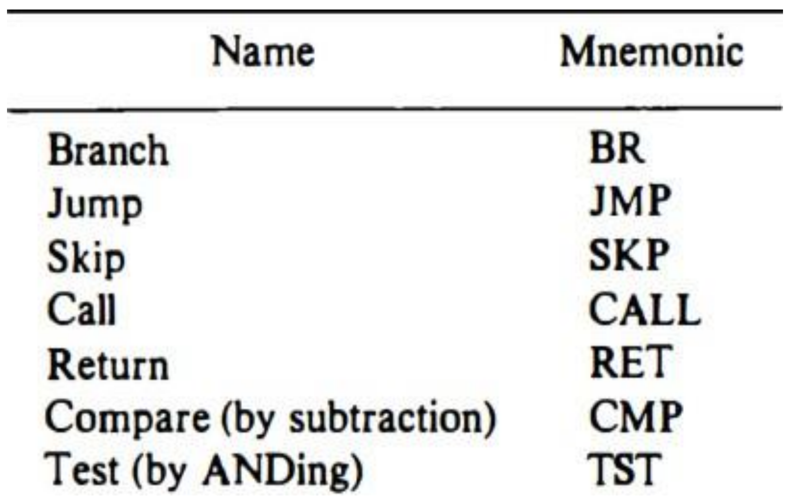
| Name | Mnemonic |
|---|---|
| Logical shift right | SHR |
| Logical shift left | SHL |
| Arithmetic shift right | SHRA |
| Arithmetic shift left | SHLA |
| Rotate right | ROR |
| Rotate left | ROL |
| Rotate right through carry | RORC |
| Rotate left through carry | ROLC |

**Table : Typical Shift Instructions.**

## Program Control Instructions/Transfer of control Instruction:

- Instructions are always stored in successive memory locations.
- When processed in the CPU, the instructions are fetched from consecutive memory locations and executed.
- The program control instructions specify conditions for altering the content of the program counter, while data transfer and manipulation in-structions 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.

- Branch and jump instructions may be conditional or unconditional.
- The conditional branch instruction specifies a condition such as branch if positive or branch if zero. If the condition is met, the program counter is loaded with the branch address and the next instruction is taken from this address.
- If the condition is not met, the program counter is not changed and the next instruction is taken from the next location in sequence.
- 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.



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

**Table : Typical Program Control 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. 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 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 overflow when negative numbers are In 2's complement. For the 8-bit ALU, V = 1 if the output is greater than +127 or less than -128.

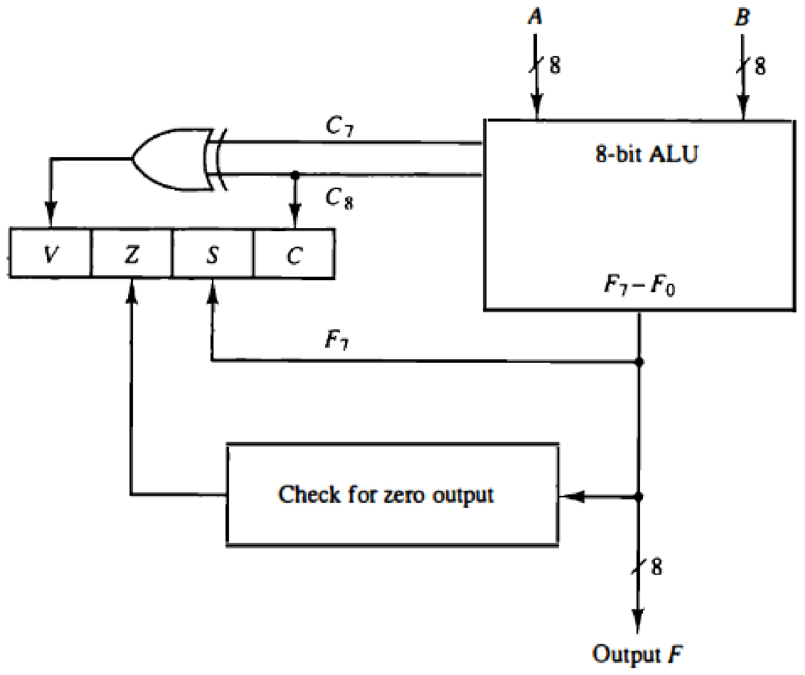
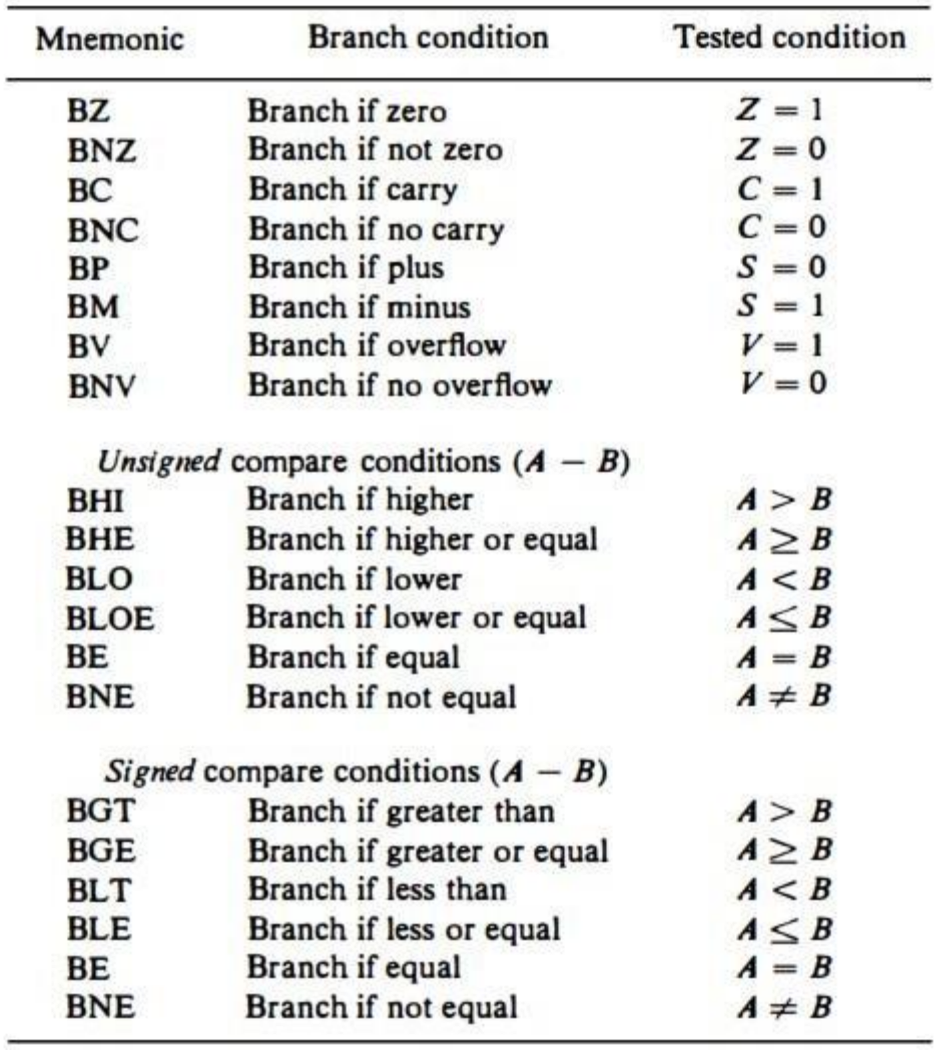


**Figure : Status register bits.**

## 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 | $A \neq B$ |

**Table : Conditional Branch Instructions**

**Subroutine Call and Return**

- A subroutine is a self-contained sequence of instructions that performs a given computational task.

- 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.

- 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.

- 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

- A subroutine call is implemented with the following micro operations:

  SP ←SP - 1 Decrement stack pointer

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

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

  SP ←SP + 1 Increment stack pointer

## 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.
- CPU does not responds to an interrupt until the executions of currently running instruction ends.
- When an interrupt initiated the state of CPU containing following information saved.
  - Content of Program counter(PC)
  - Contents of all CPU registers
  - Contents of all status bit conditions

- Before going to next fetch cycle ,it checks for an interrupt signal, if there is interrupt
    - Save CPU state (PC,CPU registers ,PSW) in memory stack.
    - PC->interrupt branch address, PSW->status bits of interrupt service program
    - Last instruction of service program is return from interrupt.
    - PSW-> old PSW, PC->old PC, CPU register-> old CPU register

**Types of Interrupts**

There are three major types of interrupts that cause a break in the normal execution of a program.

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.
- External interrupts depend on external conditions that are independent of the program being executed at the time.
- **Internal interrupts** arise from illegal or erroneous use of an instruction or data.
- Internal interrupts are also called traps .
- Internal interrupt is initiated by some exceptional condition caused by the program.
- 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.

- Internal interrupts are synchronous with the program while external interrupts are asynchronous.
- A **software interrupt** is initiated by executing an instruction.
- Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call.
- The most common use of software interrupt is associated with a supervisor call instruction ,switching from a CPU user mode to the supervisor mode.
- 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.

## RISC and CISC

- An important aspect of computer architecture is the design of the instruction set for the processor. The instruction set chosen for a particular computer determines the way that machine language programs are constructed.
- Early computers had small and simple instruction sets, forced mainly by the need to minimize the hardware used to implement them. As digital hardware became cheaper with the advent of integrated circuits, computer instructions tended to increase both in number and complexity. Many computers have instruction sets that include more than 100 and sometimes even more than 200 instructions**.**
- A computer with a large number of instructions is classified as a **complex instruction set computer (CISC).**
- A number of computer designers recommended that computers use fewer instructions with simple constructs so they can be executed much faster within the CPU without having to use memory as often. This type of computer is classified as a **reduced instruction set computer (RISC)**.

**RISC Characteristics :**

- The concept of RISC architecture involves an attempt to reduce execution time by simplifying the instruction set of the computer.
    - Relatively few instructions
    - Relatively few addressing modes
    - Memory access limited to load and store instructions
    - All operations done within the registers of the CPU
    - Fixed-length, easily decoded instruction format
    - Single-cycle instruction execution
    - Hardwired rather than microprogrammed control
    - A relatively large number of registers in the processor unit
    - Use of overlapped register windows to speed-up procedure call and return
    - Efficient instruction pipeline
    - Compiler support for efficient translation of high-level language pro grams into machine language programs

- Example of RISC: ARM, PA-RISC, Power Architecture, Alpha, AVR, ARC and the SPARC.
- The essential goal of a CISC architecture is to attempt to provide a single machine instruction for each statement that is written in a high-level language.
- Examples of CISC architectures are the Digital Equipment Corporation VAX, Motorola 68000 family, System/360, AMD ,the Intel x86 CPUs and the IBM 370 computer.

**CISC characteristics:**

- A large number of instructions-typically from 100 to 250 instructions
- Some instructions that perform specialized tasks and are used infrequently
- A large variety of addressing modes-typically from 5 to 20 different modes
- Variable-length instruction formats
- Instructions that manipulate operands in memory

## Advantages of RISC Architecture

- The performance of RISC processors is often two to four times than that of CISC processors because of simplified instruction set.
- This architecture uses less chip space due to reduced instruction set.
- The per-chip cost is reduced by this architecture that uses smaller chips consisting of more components on a single silicon wafer.
- RISC processors can be designed more quickly than CISC processors due to its simple architecture.
- The execution of instructions in RISC processors is high due to the use of many registers for holding and passing the instructions.

## Disadvantages of RISC Architecture

- The RISC processor's performance may vary according to the code executed because subsequent instructions may depend on the previous instruction for their execution in a cycle.
- Programmers and compilers often use complex instructions.

- RISC processors require very fast memory systems to feed various instructions. Typically, a large memory cache is provided on the chip in most RISC based systems.

**Advantages of CISC Architecture**

- Microprogramming is easy to implement and much less expensive than hard wiring a control unit.
- It is easy to add new commands into the chip without changing the structure of the instruction set as the architecture uses general-purpose hardware to carry out commands.
- This architecture makes the efficient use of main memory since the complexity (or more capability) of instruction allows to use less number of instructions to achieve a given task.
- The compiler need not be very complicated, as the micro program instruction sets can be written to match the constructs of high level languages.

## Disadvantages of CISC Architecture

- A new or succeeding versions of CISC processors consists early generation processors in their subsets (succeeding version). Therefore, chip hardware and instruction set became complex with each generation of the processor.
- The overall performance of the machine is reduced because of slower clock speed.
- The complexity of hardware and on-chip software included in CISC design to perform many functions.

**Overlapped Register Windows**

- Procedure call and return occurs quite often in high-level programming languages.
- When translated into machine language, a procedure call produces a sequence of instructions that save register values, pass parameters needed for the procedure, and then calls a subroutine to execute the body of the procedure.
- After a procedure return, the program restores the old register values, passes results to the calling program, and returns from the subroutine. Saving and restoring registers and passing of parameters and results involve time consuming operation.
- A characteristic of some RISC processors is their use of overlapped register windows to provide the passing of parameters and avoid the need for saving and restoring register values.
- Each procedure call results in the allocation of a new window consisting of a set of registers from the register file for use by the new procedure.

- Each procedure call activates a new register window by incrementing a pointer, while the return statement decrements the pointer and causes the activation of the previous window. Windows for adjacent procedures have overlapping registers that are shared to provide the passing of parameters and results.
- Figure , The system has a total of 74 registers.
- Registers R0-R9 are global registers that hold parameters shared by all procedures.
- The other 64 registers are divided into four windows to accommodate procedures A, B, C, and D.
- Each register window consists of 10 local registers and two sets of six registers common to adjacent windows.
- The common overlapped registers permit parameters to be passed without the actual movement of data.

- Only one register window is activated at any given time with a pointer indicating the active window.
- The high registers of the calling procedure overlap the low registers of the called procedure, and therefore the parameters automatically transfer from calling to called procedure.

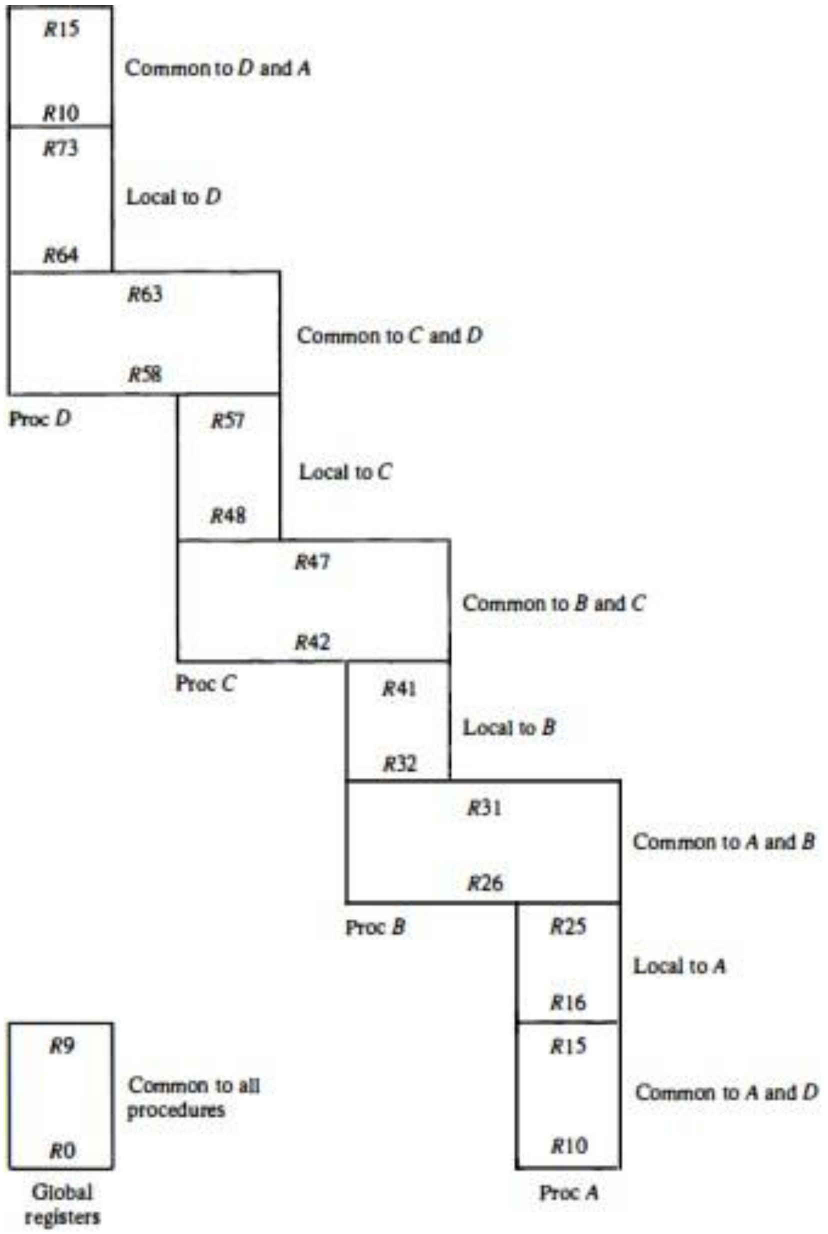


**Figure : Overlapped register windows.**

Example: Suppose that procedure A calls procedure B.

- Registers R26 through R31 are common to both procedures, and therefore procedure A stores the parameters for procedure B in these registers.
- Procedure B uses local registers R32 through R41 for local variable storage.
- If procedure B calls procedure C, it will pass the parameters through registers R42 through R47.
- When procedure B is ready to return at the end of its computation, the program stores results of the computation in registers R26 through R31 and transfers back to the register window of procedure A.
- Note that registers R10 through R15 are common to procedures A and D because the four windows have a circular organization with A being adjacent to D.

- In general, the organization of register windows will have the following relationships:

  Number of global registers = G

  Number of local registers in each window = L

  Number of registers common to two windows = C

  Number of windows = W

The number of registers available for each window is calculated as follows:

window size = L + 2C + G

The total number of registers needed in the processor is register file = (L + C) W + G

In the example of Figure we have G = 10, L = 10, C = 6, and W = 4.

The window size is 10 + 12 + 10 = 32 registers,

The register file consists of (10 + 6) x 4 +10 = 74 registers.