# PROCESSOR ORGANIZATION

**Register Transfer Language**

- A digital computer system exhibits an interconnection of digital modules such as registers, decoders, arithmetic elements, and Control logic.
- These digital modules are interconnected with some common data and control paths to form a complete digital system.
- Moreover, digital modules are best defined by the registers and the operations that are performed on the data stored in them.
- The operations performed on the data stored in registers are called **Micro-operations**.

The **Register Transfer Language** is the symbolic representation of notations used to specify the sequence of micro-operations.

- In a computer system, data transfer takes place between processor registers and memory and between processor registers and input-output systems. These data transfer can be represented by standard notations given below:
- Notations R0, R1, R2..., and so on represent processor registers.
- The addresses of memory locations are represented by names such as LOC, PLACE, MEM, etc.
- Input-output registers are represented by names such as DATA IN, DATA OUT and so on.
- The content of register or memory location is denoted by placing square brackets around the name of the register or memory location.

## Register Transfer

The term Register Transfer refers to the availability of hardware logic circuits that can perform a given micro-operation and transfer the result of the operation to the same or another register.

Most of the standard notations used for specifying operations on various registers are stated below.

- The memory address register is designated by **MAR**.
- Program Counter **PC** holds the next instruction's address.
- Instruction Register **IR** holds the instruction being executed.
- **R1** (Processor Register).
- We can also indicate individual bits by placing them in parenthesis. For instance, PC (8-15), R2 (5), etc.
- Data Transfer from one register to another register is represented in symbolic form by means of replacement operator. For instance, the following statement denotes a transfer of the data of register R1 into register R2.

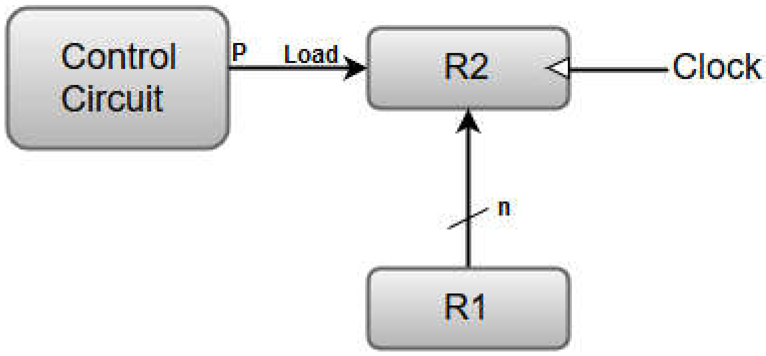
1. R2 ← R1

- Typically, most of the users want the transfer to occur only in a predetermined control condition. This can be shown by following if-then statement:
If (P=1) then (R2 ← R1); Here P is a control signal generated in the control section.
- It is more convenient to specify a control function (P) by separating the control variables from the register transfer operation. For instance, the following statement defines the data transfer operation under a specific control function (P).

1. P: R2 ← R1

The following image shows the block diagram that depicts the transfer of data from R1 to R2.

**Transfer from R1 to R2 when P = 1:**

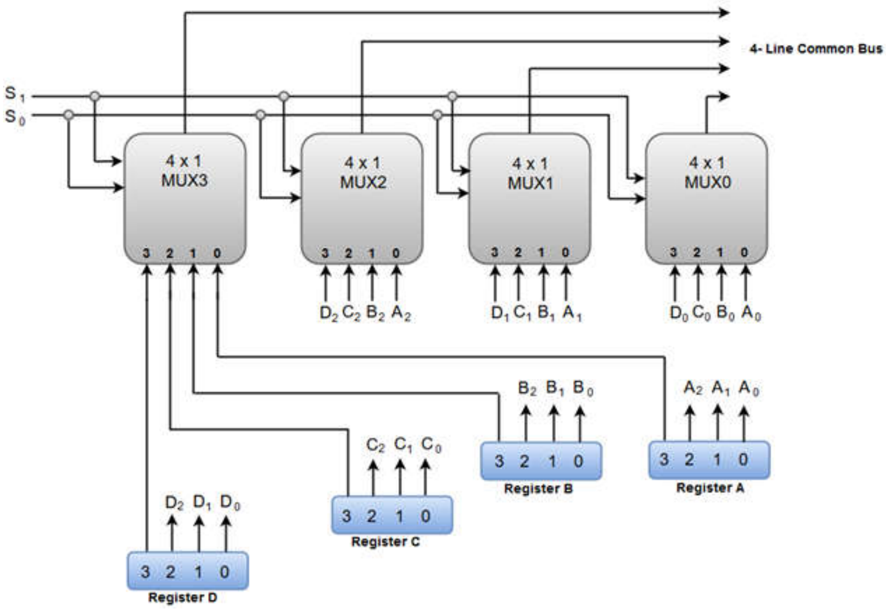



Here, the letter 'n' indicates the number of bits for the register. The 'n' outputs of the register R1 are connected to the 'n' inputs of register R2.

A load input is activated by the control variable 'P' which is transferred to the register R2.

## Bus And Memory Transfer

**Bus System for 4 Registers:**




A digital system composed of many registers, and paths must be provided to transfer information from one register to another. The number of wires connecting all of the registers will be excessive if separate lines are used between each register and all other registers in the system.
A bus structure, on the other hand, is more efficient for transferring information between registers in a multi-register configuration system.
A bus consists of a set of common lines, one for each bit of register, through which binary information is transferred one at a time. Control signals determine which register is selected by the bus during a particular register transfer.
The following block diagram shows a Bus system for four registers. It is constructed with the help of four 4 * 1 Multiplexers each having four data inputs (0 through 3) and two selection inputs (S1 and S2).

- ✓ Stack Organization
- ✓ Instruction Formats
- ✓ Addressing Modes

- ✓ Data Transfer And Manipulation
- ✓ Program Control
- ✓ Reduced Instruction Set Computer (RISC)

**Introduction:**

- The main part of the computer that performs the bulk of data-processing operations is called the central processing unit and is referred to as the CPU.
- The CPU is made up of three major parts, as shown in Fig. 8-1.

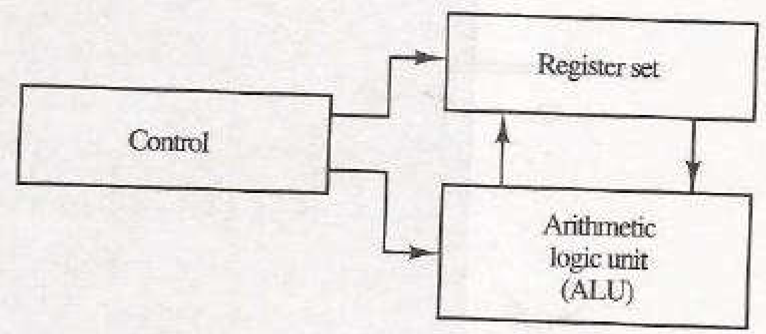



Figure 8-1 Major components of CPU.

- 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.

**<u>1. Stack Organization:</u>**

- A stack or last-in first-out (LIFO) is useful feature that is included in the CPU of most computers.
- <u>Stack:</u>
    - A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved.
- The operation of a stack can be compared to a stack of trays. The last tray placed on top of the stack is the first to be taken off.
- In the computer stack is a memory unit with an address register that can count the address only.
- The register that holds the address for the stack is called a stack pointer (SP). It always points at the top item in the stack.
- The two operations that are performed on stack are the insertion and deletion.
- The operation of insertion is called *PUSH.*
- The operation of deletion is called *POP.*
- These operations are simulated by incrementing and decrementing the stack pointer register (SP).

**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 below figure shows the organization of a 64-word register stack.

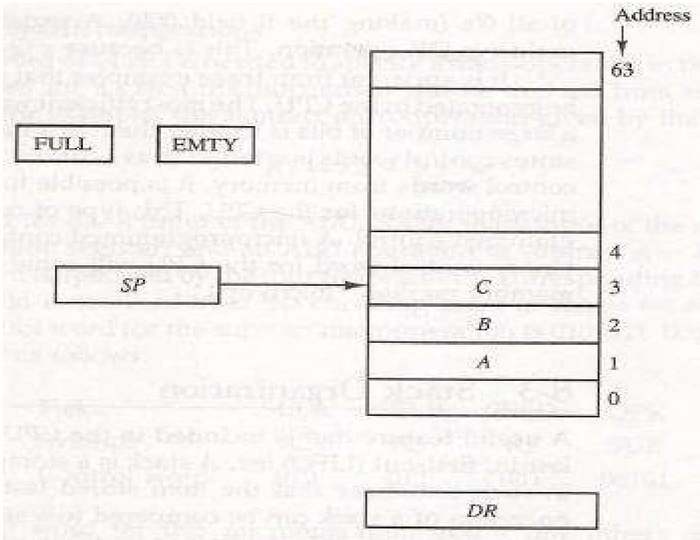



Figure 8-3 Block diagram of a 64-word stack.

- The stack pointer register SP contains a binary number whose value is equal to the address of the word is currently on top of the stack. Three items are placed in the stack: A, *B, C, in* that order.
- In above figure C is on top of the stack so that the content of *SP* is 3.
- For removing the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of stack SP.
- Now the top of the stack is B, so that the content of SP is 2.
- Similarly for inserting the 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 $2^6 = 64$.
- 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.
- Then the one-bit register FULL is set to 1, when the stack is full.
- Similarly when 000000 is decremented by 1, the result is 111111, and then the one-bit register EMTY is set 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.

**PUSH:**

- 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:

| | |
|---|---|
| $SP \leftarrow SP + 1$ | Increment stack pointer |
| $M[SP] \leftarrow DR$ | Write item on top of the stack |

| | |
|---|---|
| If $(SP = 0)$ then $(FULL \leftarrow 1)$ | Check if stack is full |
| $EMTY \leftarrow 0$ | Mark the stack not empty |

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

- 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 to 1.
- This condition is reached if the top item prior to the last push way 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, so the EMTY is cleared to 0.

**POP:**

- 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 min operations:

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

- The top item is read from the stack into DR.
- The stack pointer is then decremented. If its value reaches zero, the stack is empty, so EMTY is set 1.
- This condition is reached if the item read was in location 1. Once this it is read out, SP is decremented and reaches the value 0, which is the initial value of SP.
- If a pop operation reads the item from location 0 and then is decremented, *SP* changes to 111111, which is equivalent to decimal 63 in above configuration, the word in address 0 receives the last item in the stack.

**Memory Stack:**

- In the above discussion a stack can exist as a stand-alone unit. But in the CPU implementation of a stack is done by assigning a portion of memory to a stack operation and using a processor register as stack pointer.
- The below figure shows a portion computer memory partitioned into three segments: program, data, and stack.

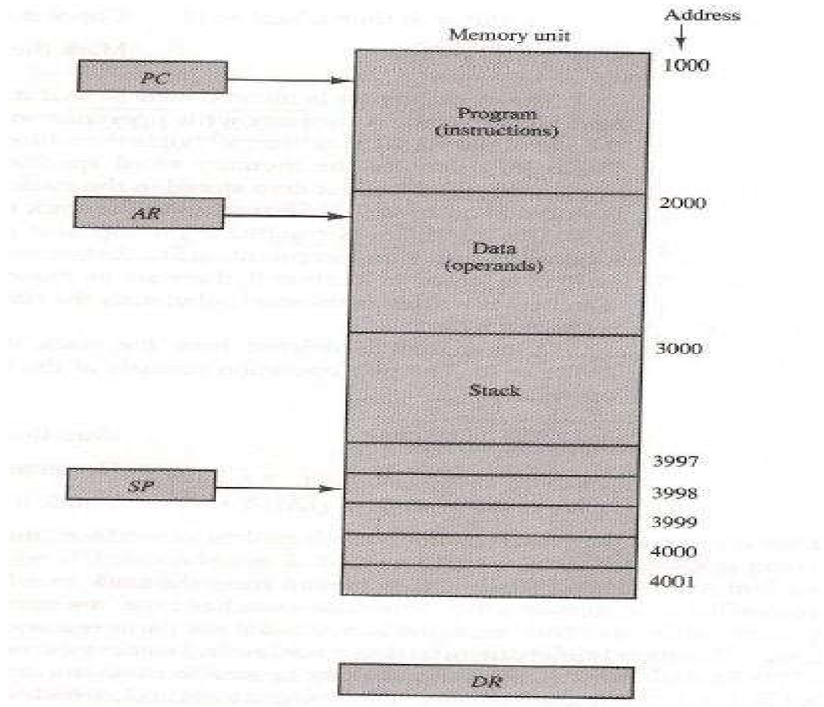



**Figure 8-4** Computer memory with program, data, and stack segments.

- The program counter *PC* points at the address of the next instruction in 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 exec phase to read an operand.
    - *SP* is used to push or pop items into or from stack.
- As shown in Fig. 8-4, 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.
- The items in the stack communicate with a data register *DR.* A new item is inserted with the push operation as follows:

**SP← SP-1**

**M [SP] ← DR**

- The stack pointer is decremented so that it points at the address of the next word.
- A memory write operation inserts the word from DR into the top of stack. A new item is deleted with a pop operation as follows:

**DR ← M [SP]**
**SP← SP+1**

- The top item is read from the stack into DR. The stack pointer is then decremented to point at the next item in the stack.
- Most computers do not provide hardware to check for stack overflow (full stack) or underflow (empty stack).
- The stack limits can be checked by using processor registers:
    - one to hold the upper limit (3000 in this case)
    - Other to hold the lower limit (4001 in this case).
- After a push operation, *SP* compared with the upper-limit register and after a pop operation, *SP* is a compared with the lower-limit register.
- The two microoperations needed for either the push or pop are

(1) An access to memory through SP

(2) Updating SP.

- The advantage of a memory stack is that the CPU can refer to it without having specify an address, since the address is always available and automatically updated in the stack pointer.

**Reverse Polish Notation:**

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

**A*B+C*D**

- For evaluating the above expression it is necessary to compute the product A*B, store this product result while computing C*D, and then sum the two products.
- For doing this type of infix notation, it is necessary to scan back and forth along the expression to determine the next operation to be performed.
- The Polish mathematician Lukasiewicz showed that arithmetic expression can be represented in *prefix notation.*
- This representation, often referred to as *Polish notation,* places the operator before the operands. So it is also called as *prefix notation*.
- The *Postfix notation,* referred to as *reverse Polish notation (RPN),* places the operator after the operands.
- The following examples demonstrate the three representations

Eg: A+B -------- > Infix notation

+AB----------> Prefix or Polish notation

AB+-----------> Post or reverse Polish notation

- The reverse Polish notation is in a form suitable for stack manipulation. The expression

  **A*B+C*D**

  Is written in reverse polish notation as

  **AB* CD* +**

  And it is evaluated as follows

  - ✓ Scan the expression from left to right.
  - ✓ When operator is reached, perform the operation with the two operands found on the left side of the operator.
  - ✓ Remove the two operands and the operator and replace them by the number obtained from the result of the operation.
  - ✓ Continue to scan the expression and repeat the procedure for every operation encountered until there are no more operators.
- For the expression above it find the operator * after A and B. So it perform the operation A*B and replace A, B and * with the result.
- The next operator is a * and it previous two operands are C and D, so it perform the operation C*D and places the result in places C, D and *.
- The next operator is + and the two operands to be added are the two products, so we add the two quantities to obtain the result.
- The conversion from infix notation to reverse Polish notation must take into consideration the operational hierarchy adopted for infix notation.
- This hierarchy dictates that we first perform all arithmetic inside inner parentheses, then inside outer parentheses, and do multiplication and division operations before addition and subtraction operations.

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

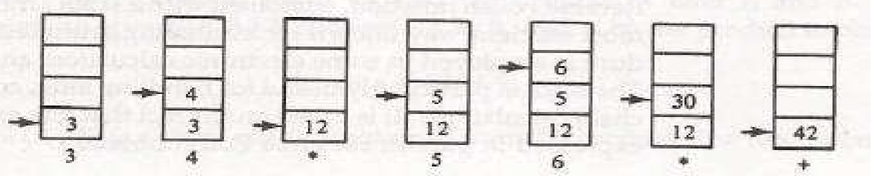
  **(3*4) + (5*6)**

  In reverse polish notation, it is expressed as

  34 * 56* +

- Now consider the stack operations shown in Fig. 8-5.

Figure 8-5 Stack operations to evaluate 3 • 4 + 5 • 6.




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

## 2. Instruction Formats:

- The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register.
- The bits of the instruction are divided into groups called fields.
- The most common fields found in instruction formats are:
    1. An operation code field that specifies the operation to be perform
    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.
- Computers may have instructions of several different lengths containing varying number of addresses.
- The number of address fields in the instruct 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.

**Single Accumulator Organization:**

- ✓ In an accumulator type organization all the operations are performed with an implied accumulator register.
- ✓ The instruction format in this type of computer uses one address field.
- ✓ For example, the instruction that specifies an arithmetic addition defined by an assembly language instruction as
    - **ADD X**
- ✓ Where X is the address of the operand. The ADD instruction in this case results in the operation *AC ←AC +M[X]. AC* is the accumulator register and *M[X]* symbolizes the memory word located at address X.

**General register organization:**

- ✓ The instruction format in this type of computer needs three register address fields.
- ✓ Thus the instruction for an arithmetic addition may be written in an assembly language as

  **ADD R1, R2, R3**

  to denote the operation R1←R2 + R3. The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers.
- ✓ Thus the instruction **ADD R1, R2** would denote the operation R1 ← *R1 +* R2. Only register addresses for R1 and R2 need be specified in this instruction.
- ✓ General register-type computers employ two or three address fields in their instruction format.
- ✓ Each address field may specify a processor register or a memory word.
- ✓ An instruction symbolized by **ADD R1, X** would specify the operation R1← R1 + M[X].
- ✓ It has two address fields, one for register R1 and the other for the memory address X**.**

**Stack organization:**

- ✓ The stack-organized CPU has PUSH and POP instructions which require an address field.
- ✓ Thus the instruction **PUSH X** will push the word at address X to the top of the stack.
- ✓ The stack pointer 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.
- ✓ The instruction **ADD** 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.
- ✓ There is no need to specify operands with an address field since all operands are implied to be in the stack.

- Most computers fall into one of the three types of organizations.
- Some computers combine features from more than one organizational structure.

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

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

- Using zero, one, two, or three address instructions and using the symbols ADD, SUB, MUL and DIV for four arithmetic operations; MOV for the transfer type operations; and LOAD and STORE for transfer to and from memory and AC register.
- Assuming that the operands are in memory addresses A, B, C, and D and the result must be stored in memory ar address X and also the CPU has general purpose registers R1, R2, R3 and R4.

**Three Address Instructions:**

- ✓ Three-address instruction formats can use each address field to specify either a processor register or a memory operand.
- ✓ The program 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
```

- ✓ 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 formats use each address field can specify either a processor register or 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.
- ✓ The first symbol listed in an instruction is assumed be both a source and the destination where the result of the operation transferred.

**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. But for the basic discussion we will neglect the second register and assume that the *AC* contains the result of all operations.
- ✓ The program to evaluate X=(A+B) * *(C+*D) is

```
LOAD    A    AC←M[A]
ADD     B    AC←AC + M[B]
STORE   T    M[T]←AC
LOAD    C    AC←M[C]
ADD     D    AC←AC + M[D]
MUL     T    AC←AC*M[T]
STORE   X    M[X]←AC
```

- ✓ All operations are done between the AC register and a memory operand.
- ✓ T is the address of 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     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
```

- ✓ 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.

**RISC Instructions:**

- ✓ The instruction set of a typical RISC processor is use only load and store instructions for communicating between memory and CPU.
- ✓ All other instructions are executed within the registers of CPU without referring to memory.
- ✓ 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.
- ✓ The following is a program to evaluate X=(A+B)*(C+D)

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

- ✓ The load instructions transfer the operands from memory to CPU register.
- ✓ The add and multiply operations are executed with data in the register without accessing memory.
- ✓ The result of the computations is then stored memory with a store in instruction.

## 3. <u>Addressing Modes</u>

- ➢ The way the operands are chosen during program execution is dependent on the addressing mode of the instruction.
- ➢ Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:
  - o 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.
  - o To reduce the number of bits in the addressing field of the instruction
- ➢ Most addressing modes modify the address field of the instruction; there are two modes that need no address field at all. These are *implied* and *immediate* modes.

**Implied Mode:**

- ✓ In this mode the operands are specified implicitly in the definition of the instruction.
- ✓ For example, the instruction "complement accumulator" is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction.
- ✓ All register reference instructions that use an accumulator are implied mode instructions.
- ✓ Zero address in a stack organization computer is implied mode instructions.

**Immediate Mode:**

- ✓ In this mode the operand is specified in the instruction itself.
- ✓ In other words an immediate-mode instruction has an operand rather than an address field.
- ✓ Immediate-mode instructions are useful for initializing registers to a constant value.
- ➢ The address field of an instruction may specify either a memory word or a processor register.
- ➢ When the address specifies a processor register, the instruction is said to be in the register mode.

**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.

**Register Indirect Mode:**

- ✓ In this mode the instruction specifies a register in 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 few bits to select a register than would have been required to specify a memory address directly.

**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.

- ➢ The address field of an instruction is used by the control unit in the CPU to obtain the operand from memory.
- ➢ Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated.
- ➢ The basic two mode of addressing used in CPU are *direct* and *indirect* address mode.

**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 a branch-type instruction the address field specifies the actual branch address.

**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.

- ➢ A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU.
- ➢ The effective address in these modes is obtained from the following computation:

Effective address =address part of instruction + content of CPU register

- ➢ The CPU register used in the computation may be the program counter, an index register, or a base register.
- ➢ We have a different addressing mode which is used for a different application.

**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.

**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.
- ✓ An index register is a special CPU register that contains an index value.

**Base Register Addressing Mode:**

- ✓ In this mode the content of a base register is added to the address part of the instruction to obtain the effective address.
- ✓ This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register.

**Numerical Example:**

- To show the differences between the various modes, we will show the effect of the addressing modes on the instruction defined in Fig. 8-7.

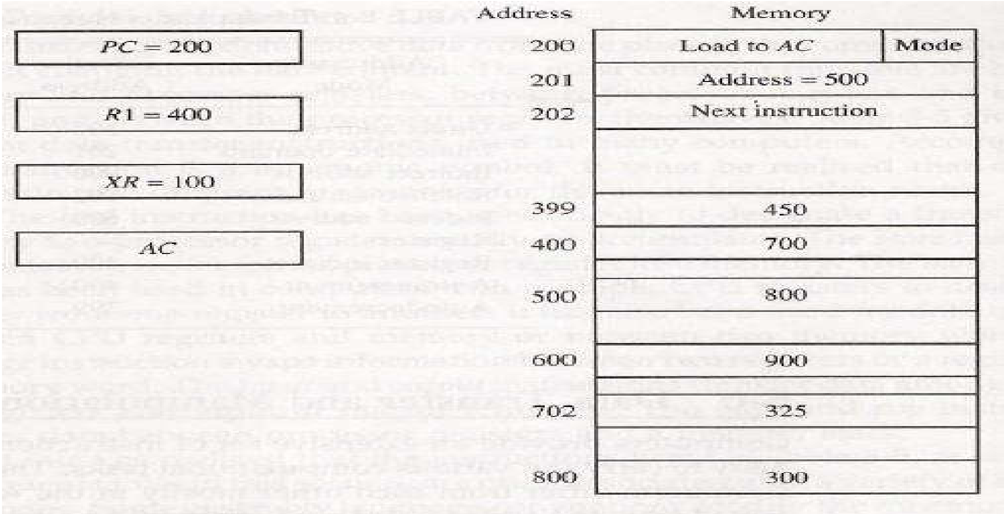



Figure 8-7 Numerical example for addressing modes.

- 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.
- In the **direct address mode** the effective address is the address part of the instruction 500 and the operand to be loaded into *AC* is 500.
- In the **immediate mode** the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into AC.
- In the **indirect mode** the effective address is stored in memory at address 500. Therefore, the effective address is 800 and the operand is 300.
- In the **relative mode** the effective address is 500 + 202 =702 and the operand is 325. (the value in PC after the fetch phase and during the execute phase is 202.)
- In the **index mode** the effective address is XR+ 500 = 100 + 500 = 600 and the operand is 900.
- In the **register mode** the operand is in R1 and 400 is loaded into *AC.*
- In the **register indirect mode** the effective address is 400, equal to the content of R1 and the operand loaded into AC is 700.
- The **auto-increment mode** is the same as the register indirect mode except that R1 is incremented to 401 after the execution of the instruction.
- The **auto-decrement mode** decrements R1 to 399 prior to the execution of the instruction. The operand loaded into AC is now 450.
- Table 8-4 lists the values of the effective address and the operand loaded into AC for the nine addressing modes.

TABLE 8-4 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 |