# YESHWA TAO CHAVAN COLLEGE OF ENGINEERING, NAGPUR
# DEPARTMENT OF COMPUTER TECHNOLOGY
## III SEM
## SESSION 2020-21
## SECTION: A, B
## SUB: COMPUTER ARCHITECTURE AND ORGANIZATION

*UNIT-I:* *[6 hrs]*

Basic Structure of Computer Hardware and Software: Functional Units, Basic Operational Concepts, Bus Structures, Software, addressing methods and machine program sequencing: Memory Locations, addressing and encoding of information, Instructions and Instruction sequencing,

*UNIT- II:* *[7 hrs]*

Addressing modes, Assembly language, Stacks, Subroutine. Instruction set : Simple RISC Processing Unit: Some fundamental concepts, Execution of a complete instruction, Single, two, three bus organization, Sequencing of control Signals.

*UNIT-III:* *[7 hrs*

Processor Design, hard wired control, Microprogrammed Control: Microinstructions, Grouping of control signals, Microprogram sequencing, Micro Instructions with next Address field, perfecting microinstruction.

*UNIT-IV:* *[7 hrs]*

Arithmetic (Fixed and Floating point): Number Representation, Addition of Positive numbers, Logic Design for fast adders, Addition and Subtraction, Arithmetic and Branching conditions, Multiplications of positive numbers, Signed- Operand multiplication, fast Multiplication, Booth's Algorithm.

*UNIT-V:* *[7 hrs]*

Integer Division, Floating point numbers and operations. The Main Memory: Basic concepts, Memory Hierarchy, semiconductor RAM memories, Memory system consideration, semiconductor ROM memories, Speed Size and Cost, Cache Memory, Performance Considerations.

*UNIT-VI :* *[6 hrs]*

Mapping techniques, Pipelining: Basic Concepts, Data Hazards, Instruction Hazards Computer Peripherals: I/O Devices, I/O transfers – program controlled, interrupt driven and DMA, Interrupt handling.

*Text Books:*

| SN | Title | Edition | Authors | Publisher |
|---|---|---|---|---|
| 1 | Computer Organization | 5th edition | V.Carl Hamacher, Zvonko Vranesic, | McGraw Hill Publications. |

**Reference Books:**

| SN | Title | Edition | Authors | Publisher |
|---|---|---|---|---|
| 1 | Computer Organization and Architecture | 6th edition | Willaiam Staliing | Pearson Education |
| 2 | Computer Architecture & Organization | 3rd edition | J.P. Hayes | McGraw Hill Publications. |

# UNIT 1

## BASIC CONCEPTS

- **Computer Architecture (CA)** is concerned with the structure and behaviour of the computer.
- CA includes the information formats, the instruction set and techniques for addressing memory.
- In general covers, CA covers 3 aspects of computer-design namely: 1) Computer Hardware, 2) Instruction set Architecture and 3) Computer Organization.

**1. Computer Hardware**

➢ It consists of electronic circuits, displays, magnetic and optical storage media and communication facilities.

**2. Instruction Set Architecture**

➢ It is programmer visible machine interface such as instruction set, registers, memory organization and exception handling.

➢ Two main approaches are 1) CISC and 2) RISC.

(CISC□Complex Instruction Set Computer, RISC□Reduced Instruction Set Computer)

**3. Computer Organization**

➢ It includes the high level aspects of a design, such as

→ memory-system<br>
→ bus-structure &<br>
→ design of the internal CPU.

➢ It refers to the operational units and their interconnections that realize the architectural specifications.

➢ It describes the function of and design of the various units of digital computer that store and process information.

## FUNCTIONAL UNITS

- A computer consists of 5 functionally independent main parts:

1) Input
2) Memory
3) ALU
4) Output &
5) Control units.

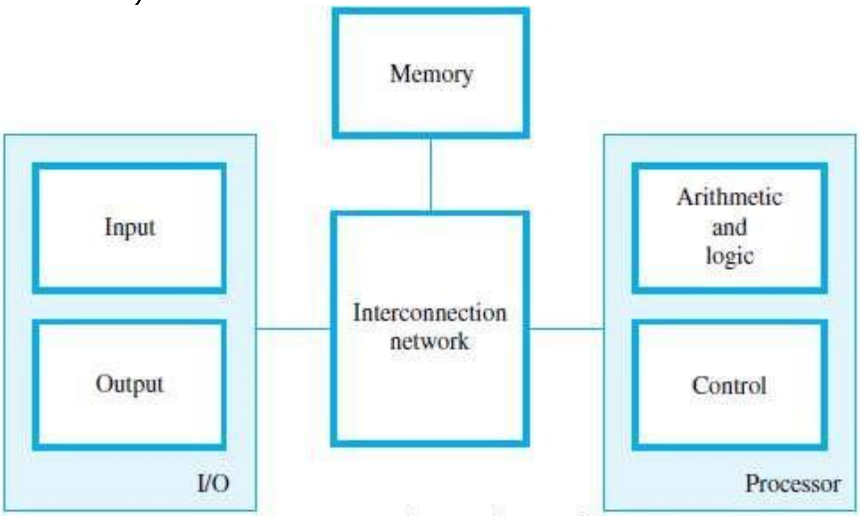


**Figure 1.1** Basic functional units of a computer.

## BASIC OPERATIONAL CONCEPTS

- An Instruction consists of 2 parts, 1) Operation code (Opcode) and 2) Operands.

| *OPCODE* | *OPERANDS* |
|---|---|

- The data/operands are stored in memory.
- The individual instruction are brought from the memory to the processor.
- Then, the processor performs the specified operation.
- Let us see a typical instruction

*ADD LOCA, R0*

- This instruction is an addition operation. The following are the steps to execute the instruction: Step 1: Fetch the instruction from main-memory into the processor.
  Step 2: Fetch the operand at location LOCA from main-memory into the processor.
  Step 3: Add the memory operand (i.e. fetched contents of LOCA) to the contents of register R0. Step 4: Store the result (sum) in R0.
- The same instruction can be realized using 2 instructions as:

*Load LOCA, R1 Add R1, R0*

- The following are the steps to execute the instruction:
  Step 1: Fetch the instruction from main-memory into the processor.
  Step 2: Fetch the operand at location LOCA from main-memory into the register R1. Step 3: Add the content of Register R1 and the contents of register R0.
  Step 4: Store the result (sum) in R0.

## MAIN PARTS OF PROCESSOR

- The **processor** contains ALU, control-circuitry and many registers.
- The processor contains „n‟ general-purpose registers **$R_0$** through **$R_{n-1}$**.
- The **IR** holds the instruction that is currently being executed.
- The **control-unit** generates the timing-signals that determine when a given action is to take place.
- The **PC** contains the memory-address of the next-instruction to be fetched & executed.
- During the execution of an instruction, the contents of PC are updated to point to next instruction.
- The **MAR** holds the address of the memory-location to be accessed.
- The **MDR** contains the data to be written into or read out of the addressed location.
- MAR and MDR facilitates the communication with memory. (IR □ Instruction-Register, PC □ Program Counter)
  (MAR □ Memory Address Register, MDR□ Memory Data Register)

## STEPS TO EXECUTE AN INSTRUCTION

1) The address of first instruction (to be executed) gets loaded into PC.
2) The contents of PC (i.e. address) are transferred to the MAR & control-unit issues Read signal to memory.
3) After certain amount of elapsed time, the first instruction is read out of memory and placed into MDR.
4) Next, the contents of MDR are transferred to IR. At this point, the instruction can be decoded & executed.
5) To fetch an operand, it's address is placed into MAR & control-unit issues Read signal. As a result, the operand is transferred from memory into MDR, and then it is transferred from MDR to ALU.
6) Likewise required number of operands is fetched into processor.
7) Finally, ALU performs the desired operation.
8) If the result of this operation is to be stored in the memory, then the result is sent to the MDR.
9) The address of the location where the result is to be stored is sent to the MAR and a Write cycle is initiated.
10) At some point during execution, contents of PC are incremented to point to next instruction in the program.

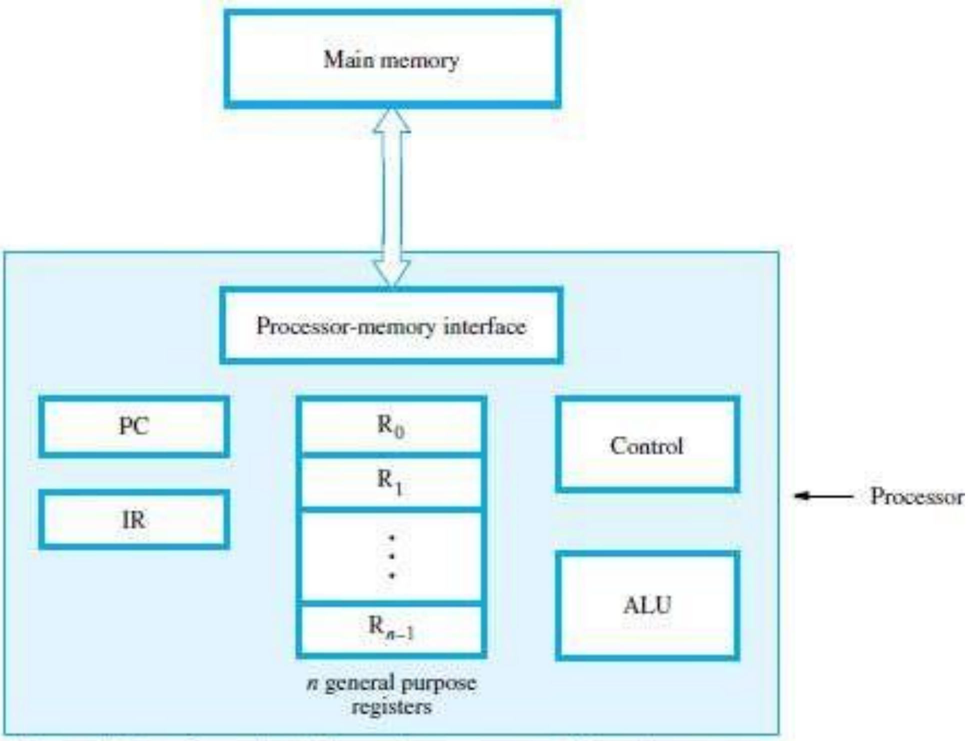


**Figure 1.2** Connection between the processor and the main memory.

## MACHINE INSTRUCTIONS & PROGRAMS:

### MEMORY-LOCATIONS & ADDRESSES

- **Memory** consists of many millions of storage cells (flip-flops).
- Each cell can store a bit of information i.e. 0 or 1 (Figure 2.1).
- Each group of n bits is referred to as a **word** of information, and n is called the **word length**.
- The word length can vary from 8 to 64 bits.
- A unit of 8 bits is called a **byte**.
- Accessing the memory to store or retrieve a single item of information (word/byte) requires distinct addresses for each item location. (It is customary to use numbers from 0 through $2^k$-1 as the addresses of successive-locations in the memory).
- If $2^k$ = no. of addressable locations;

  then $2^k$ addresses constitute the address-space of the computer.

  For example, a 24-bit address generates an address-space of $2^{24}$ locations (16 MB).

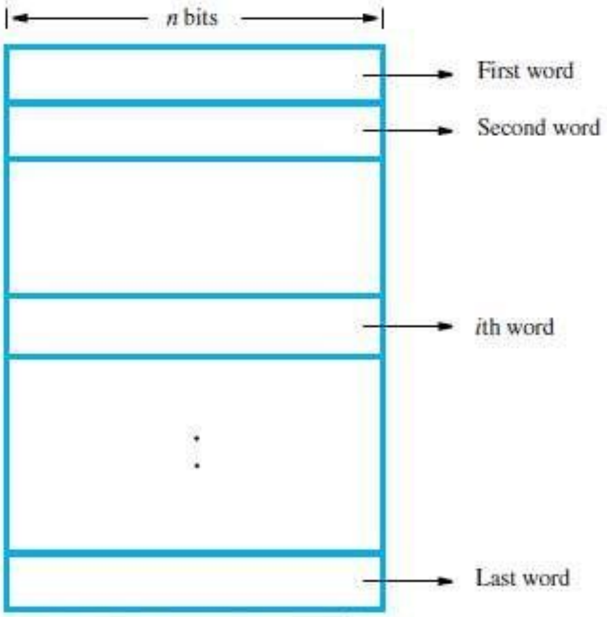
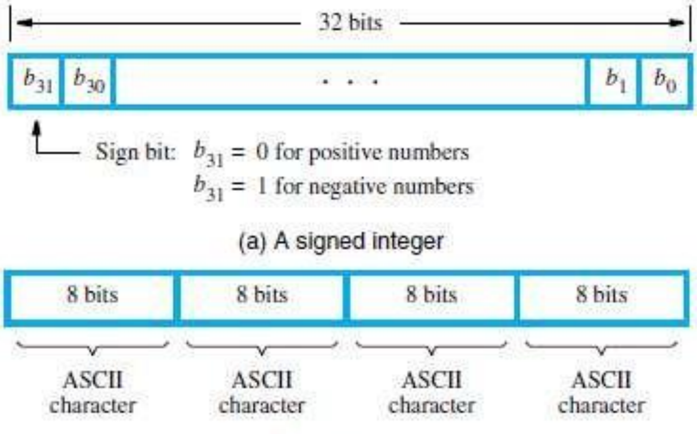


**Figure 2.1** Memory words.



(a) A signed integer

(b) Four characters

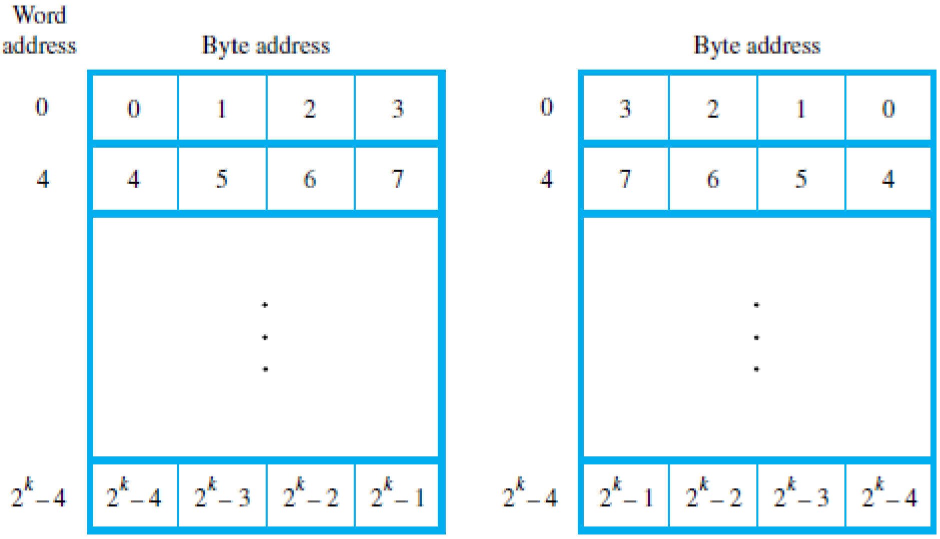
**Figure 2.2** Examples of encoded information in a 32-bit word.

## BYTE-ADDRESSABILITY

- In byte-addressable memory, successive addresses refer to successive byte locations in the memory.
- Byte locations have addresses 0, 1, 2. . . . .
- If the word-length is 32 bits, successive words are located at addresses 0, 4, 8. . with each word having 4 bytes.

## BIG-ENDIAN & LITTLE-ENDIAN ASSIGNMENTS

- There are two ways in which byte-addresses are arranged (Figure 2.3).
  1) **Big-Endian:** Lower byte-addresses are used for the more significant bytes of the word.
  2) **Little-Endian:** Lower byte-addresses are used for the less significant bytes of the word
- In both cases, byte-addresses 0, 4, 8. . . . . are taken as the addresses of successive words in the memory.
- Consider a 32-bit integer (in hex): 0**x**12345678 which consists of 4 bytes: 12, 34, 56, and 78.



(a) Big-endian assignment (b) Little-endian assignment

**Figure 2.3** Byte and word addressing.

- ➢ Hence this integer will occupy 4 bytes in memory.
- ➢ Assume, we store it at memory address starting 1000.
- ➢ On little-endian, memory will look like

| Address | Value |
|---|---|
| 1000 | 78 |
| 1001 | 56 |
| 1002 | 34 |
| 1003 | 12 |

- ➢ On big-endian, memory will look like

| Address | Value |
|---|---|
| 1000 | 12 |
| 1001 | 34 |
| 1002 | 56 |
| 1003 | 78 |

## WORD ALIGNMENT

- Words are said to be **Aligned** in memory if they begin at a byte-address that is a multiple of the number of bytes in a word.
- For example,
  - ➢ If the word length is 16(2 bytes), aligned words begin at byte-addresses 0, 2, 4 . . . . .
  - ➢ If the word length is 64(2 bytes), aligned words begin at byte-addresses 0, 8, 16 . . . . .
- Words are said to have **Unaligned Addresses**, if they begin at an arbitrary byte-address.

## ACCESSING NUMBERS, CHARACTERS & CHARACTERS STRINGS

- A number usually occupies one word. It can be accessed in the memory by specifying its word address. Similarly, individual characters can be accessed by their byte-address.
- There are two ways to indicate the length of the string:
    1) A special control character with the meaning "end of string" can be used as the last character in the string.
    2) A separate memory word location or register can contain a number indicating the length of the string in bytes.

## MEMORY OPERATIONS

- Two memory operations are:
    1) Load (Read/Fetch) &
    2) Store (Write).

- The **Load** operation transfers a copy of the contents of a specific memory-location to the processor. The memory contents remain unchanged.
- Steps for Load operation:
    1) Processor sends the address of the desired location to the memory.
    2) Processor issues „read‟ signal to memory to fetch the data.
    3) Memory reads the data stored at that address.
    4) Memory sends the read data to the processor.
- The **Store** operation transfers the information from the register to the specified memory-location. This will destroy the original contents of that memory-location.
- Steps for Store operation are:
    1) Processor sends the address of the memory-location where it wants to store data.
    2) Processor issues „write‟ signal to memory to store the data.
    3) Content of register(MDR) is written into the specified memory-location.

## INSTRUCTIONS & INSTRUCTION SEQUENCING

- A computer must have instructions capable of performing 4 types of operations:
    1) Data transfers between the memory and the registers (MOV, PUSH, POP, XCHG).
    2) Arithmetic and logic operations on data (ADD, SUB, MUL, DIV, AND, OR, NOT).
    3) Program sequencing and control (CALL.RET, LOOP, INT).
    4) I/0 transfers (IN, OUT).

## REGISTER TRANSFER NOTATION (RTN)

- The possible locations in which transfer of information occurs are: 1) Memory-location 2) Processor register & 3) Registers in I/O device.

| **Location** | **Hardware Binary Address** | **Example** | **Description** |
|---|---|---|---|
| Memory | LOC, PLACE, NUM | R1 □ [LOC] | Contents of memory-location LOC are transferred into register R1. |
| Processor | R0, R1 ,R2 | [R3] □ [R1]+[R2] | Add the contents of register R1 &R2 and places their sum into R3. |
| I/O Registers | DATAIN, DATAOUT | R1 □ DATAIN | Contents of I/O register DATAIN are transferred into register R1. |

## ASSEMBLY LANGUAGE NOTATION

- To represent machine instructions and programs, assembly language format is used.

| **Assembly Language Format** | **Description** |
|---|---|
| Move LOC, R1 | Transfer data from memory-location LOC to register R1. The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R1 are overwritten. |
| Add R1, R2, R3 | Add the contents of registers R1 and R2, and places their sum into register R3. |

## BASIC INSTRUCTION TYPES

| **Instruction Type** | **Syntax** | **Example** | **Description** | **Instructions for Operation C<-[A]+[B]** |
|---|---|---|---|---|
| Three Address | Opcode Source1,Source2,Destination | Add A,B,C | Add the contents of memory-locations A & B.<br>Then, place the result into location C. | |
| Two Address | Opcode Source, Destination | Add A,B | Add the contents of memory-locations A & B. Then, place the result into location B, replacing the original contents of this location.<br>Operand B is both a source and a destination. | Move B, C<br>Add A, C |
| One Address | Opcode Source/Destination | Load A | Copy contents of memory- location A into accumulator. | Load A<br>Add B<br>Store C |
| | | Add B | Add contents of memory- location B to contents of accumulator register & place sum back into accumulator. | |
| | | Store C | Copy the contents of the accumulator into location C. | |
| Zero Address | Opcode [no Source/Destination] | Push | Locations of all operands are defined implicitly.<br>The operands are stored in a pushdown stack. | Not possible |

- Access to data in the registers is much faster than to data stored in memory-locations.
- Let Ri represent a general-purpose register. The instructions:

*Load A,Ri*
*Store Ri,A*
*Add A,Ri*

are generalizations of the Load, Store and Add Instructions for the single-accumulator case, in which register Ri performs the function of the accumulator.

- In processors, where arithmetic operations as allowed only on operands that are in registers, the task C<-[A]+[B] can be performed by the instruction sequence:

*Move A,Ri*
*Move B,Rj*
*Add Ri,Rj*
*Move Rj,C*

## INSTRUCTION EXECUTION & STRAIGHT LINE SEQUENCING

- The program is executed as follows:
  1) Initially, the address of the first instruction is loaded into PC (Figure 2.8).
  2) Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called *Straight-Line sequencing*.
  3) During the execution of each instruction, PC is incremented by 4 to point to next instruction.
- There are 2 phases for Instruction Execution:
  1) **Fetch Phase:** The instruction is fetched from the memory-location and placed in the IR.
  2) **Execute Phase:** The contents of IR is examined to determine which operation is to be performed. The specified-operation is then performed by the processor.

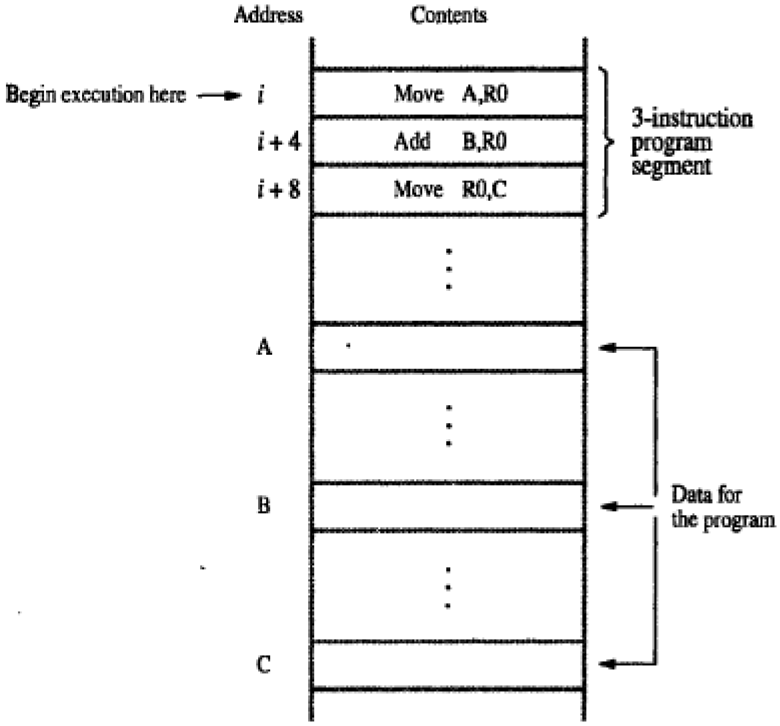


Figure 2.8 A program for C ← [A] + [B].

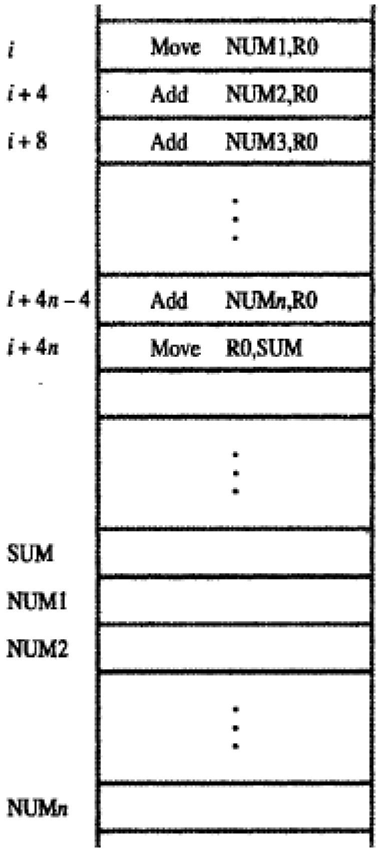


Figure 2.9 A straight-line program for adding n numbers.

**Program Explanation**

- Consider the program for adding a list of n numbers (Figure 2.9).
- The Address of the memory-locations containing the n numbers are symbolically given as NUM1, NUM2…..NUMn.
- Separate Add instruction is used to add each number to the contents of register R0.
- After all the numbers have been added, the result is placed in memory-location SUM.

## BRANCHING

- Consider the task of adding a list of „n‟ numbers (Figure 2.10).
- Number of entries in the list „n‟ is stored in memory-location **N**.
- Register **R1** is used as a counter to determine the number of times the loop is executed.
- Content-location N is loaded into register R1 at the beginning of the program.
- The **Loop** is a straight line sequence of instructions executed as many times as needed. The loop starts at location LOOP and ends at the instruction Branch>0.
- During each pass,

  → address of the next list entry is determined and

→ that entry is fetched and added to R0.

- The instruction *Decrement R1* reduces the contents of R1 by 1 each time through the loop.
- Then **Branch Instruction** loads a new value into the program counter. As a result, the processor fetches and executes the instruction at this new address called the **Branch Target**.
- A **Conditional Branch Instruction** causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in the normal way, and the next instruction in sequential address order is fetched and executed.

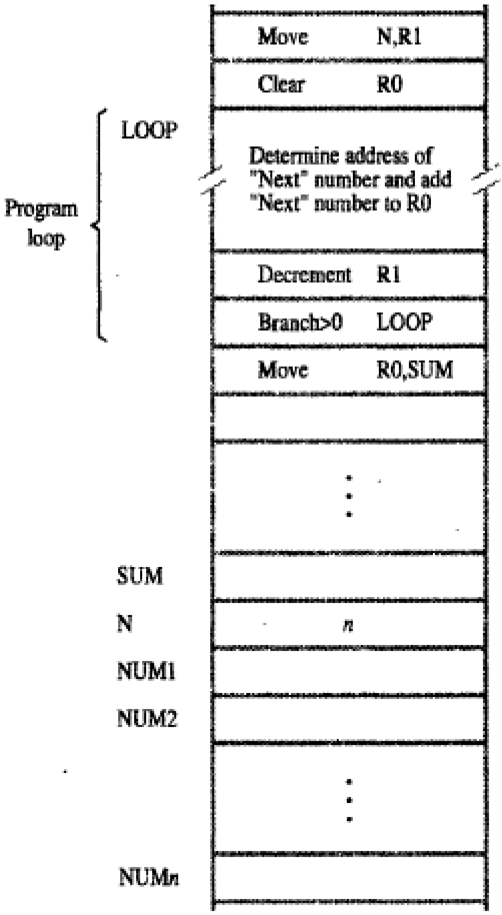


**Figure 2.10** Using a loop to add *n* numbers.

**CONDITION CODES**

- The processor keeps track of information about the results of various operations. This is accomplished by recording the required information in individual bits, called **Condition Code Flags**.
- These flags are grouped together in a special processor-register called the condition code register (or statue register).
- Four commonly used flags are:
    1) N (negative) set to 1 if the result is negative, otherwise cleared to 0.
    2) Z (zero) set to 1 if the result is 0; otherwise, cleared to 0.
    3) V (overflow) set to 1 if arithmetic overflow occurs; otherwise, cleared to 0.
    4) C (carry) set to 1 if a carry-out results from the operation; otherwise cleared to 0.

# UNIT II

**ADDRESSING MODES**

- The different ways in which the location of an operand is specified in an instruction are referred to as **Addressing Modes** (Table 2.1).

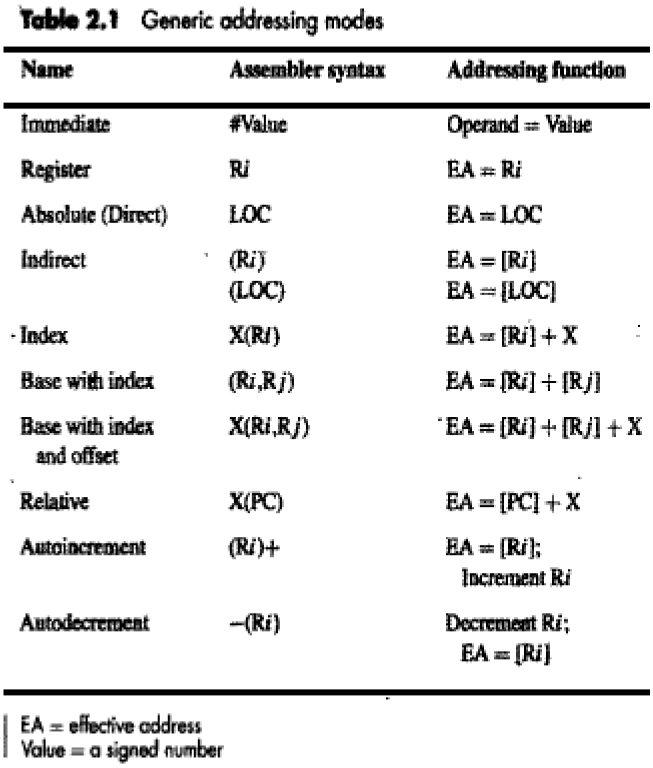


**Table 2.1** Generic addressing modes

| Name | Assembler syntax | Addressing function |
|---|---|---|
| Immediate | #Value | Operand = Value |
| Register | R*i* | EA = R*i* |
| Absolute (Direct) | LOC | EA = LOC |
| Indirect | (R*i*)<br>(LOC) | EA = [R*i*]<br>EA = [LOC] |
| Index | X(R*i*) | EA = [R*i*] + X |
| Base with index | (R*i*,R*j*) | EA = [R*i*] + [R*j*] |
| Base with index and offset | X(R*i*,R*j*) | EA = [R*i*] + [R*j*] + X |
| Relative | X(PC) | EA = [PC] + X |
| Autoincrement | (R*i*)+ | EA = [R*i*];<br>Increment R*i* |
| Autodecrement | −(R*i*) | Decrement R*i*;<br>EA = [R*i*] |

EA = effective address
Value = a signed number

**IMPLEMENTATION OF VARIABLE AND CONSTANTS**

- **Variable** is represented by allocating a memory-location to hold its value.
- Thus, the value can be changed as needed using appropriate instructions.
- There are 2 accessing modes to access the variables:
  1) Register Mode
  2) Absolute Mode

**Register Mode**

- The operand is the contents of a register.
- The name (or address) of the register is given in the instruction.
- Registers are used as temporary storage locations where the data in a register are accessed.
- For example, the instruction

  *Move R1, R2* ;Copy content of register R1 into register R2.

**Absolute (Direct) Mode**

- The operand is in a memory-location.
- The address of memory-location is given explicitly in the instruction.
- The absolute mode can represent global variables in the program.
- For example, the instruction

  *Move LOC, R2* ;Copy content of memory-location LOC into register R2.

**Immediate Mode**

- The operand is given explicitly in the instruction.
- For example, the instruction

  *Move #200, R0* ;Place the value 200 in register R0.

- Clearly, the immediate mode is only used to specify the value of a source-operand.

## INDIRECTION AND POINTERS

- Instruction does not give the operand or its address explicitly.
- Instead, the instruction provides information from which the new address of the operand can be determined.
- This address is called **Effective Address (EA)** of the operand.

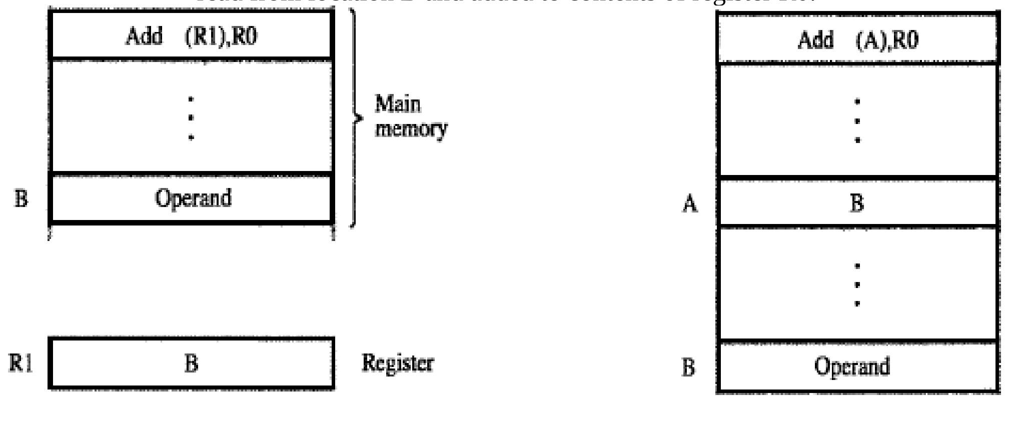
### Indirect Mode

- The EA of the operand is the contents of a register(or memory-location).
- The register (or memory-location) that contains the address of an operand is called a **Pointer**.
- We denote the indirection by

  → name of the register or

  → new address given in the instruction.

  E.g: *Add (R1),R0* ;The operand is in memory. Register R1 gives the effective-address (B) of the operand. The data is read from location B and added to contents of register R0.

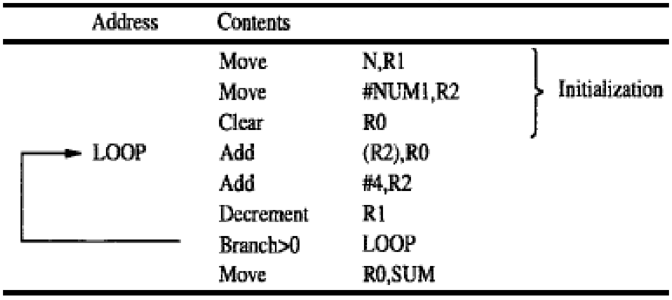


(a) Through a general-purpose register

(b) Through a memory location

**Figure 2.11** Indirect addressing.

- To execute the Add instruction in fig 2.11 (a), the processor uses the value which is in register R1, as the EA of the operand.
- It requests a read operation from the memory to read the contents of location B. The value read is the desired operand, which the processor adds to the contents of register R0.
- Indirect addressing through a memory-location is also possible as shown in fig 2.11(b). In this case, the processor first reads the contents of memory-location A, then requests a second read operation using the value B as an address to obtain the operand.



```
Address      Contents

             Move        N,R1           }
             Move        #NUM1,R2       } Initialization
             Clear       R0             }
LOOP         Add         (R2),R0
             Add         #4,R2
             Decrement   R1
             Branch>0    LOOP
             Move        R0,SUM
```

**Figure 2.12** Use of indirect addressing in the program of Figure 2.10.

### Program Explanation

- In above program, Register R2 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R2.
- The initialization-section of the program loads the counter-value n from memory-location N into R1 and uses the immediate addressing-mode to place the address value NUM1, which is the address of the first number in the list, into R2. Then it clears R0 to 0.
- The first two instructions in the loop implement the unspecified instruction block starting at LOOP.
- The first time through the loop, the instruction Add (R2), R0 fetches the operand at location NUM1 and adds it to R0.
- The second Add instruction adds 4 to the contents of the pointer R2, so that it will contain the address value NUM2 when the above instruction is executed in the second pass through the loop.

## INDEXING AND ARRAYS

- A different kind of flexibility for accessing operands is useful in dealing with lists and arrays.

### Index mode

- The operation is indicated as X(Ri)

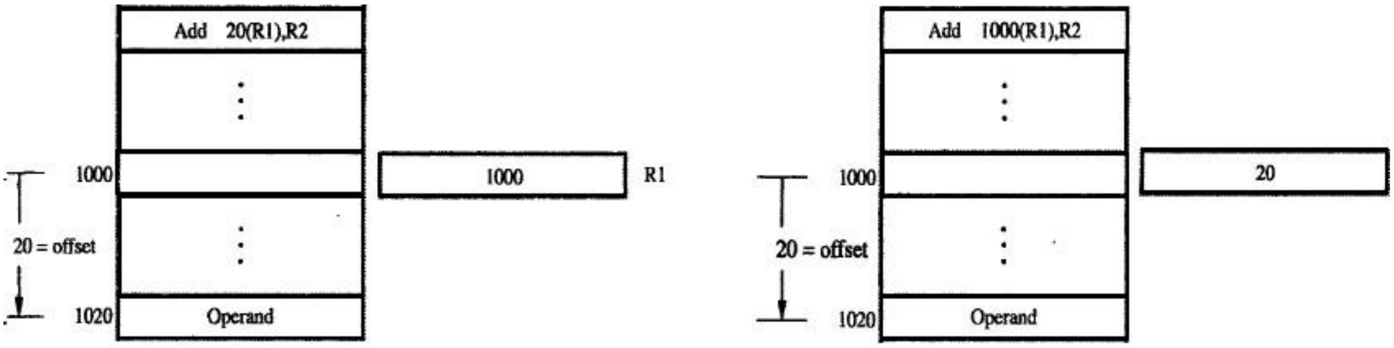
where X=the constant value which defines an offset(also called a displacement).

Ri=the name of the index register which contains address of a new location.

- The effective-address of the operand is given by EA=X+[Ri]
- The contents of the index-register are not changed in the process of generating the effective- address.
- The constant X may be given either

→ as an explicit number or

→ as a symbolic-name representing a numerical value.



(a) Offset is given as a constant

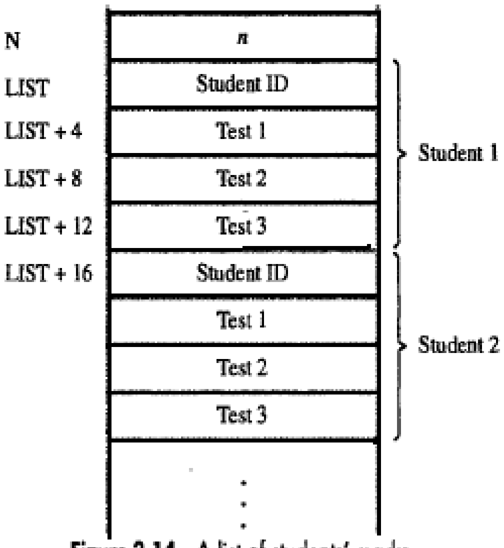
(b) Offset is in the index register

**Figure 2.13** Indexed addressing.

- Fig(a) illustrates two ways of using the Index mode. In fig(a), the index register, R1, contains the address of a memory-location, and the value X defines an offset(also called a displacement) from this address to the location where the operand is found.
- To find EA of operand: Eg: Add 20(R1), R2

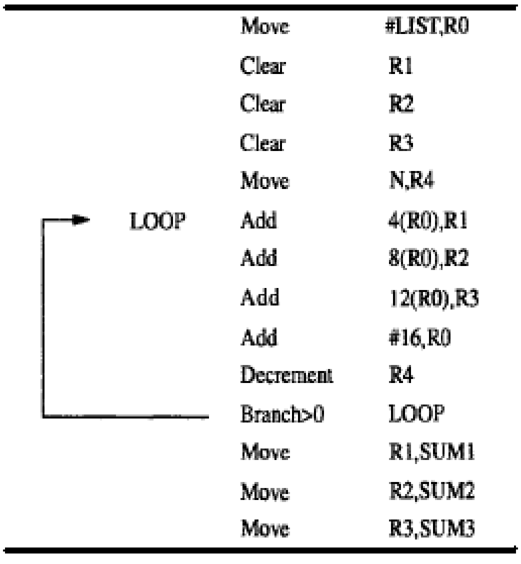
EA=>1000+20=1020

- An alternative use is illustrated in fig(b). Here, the constant X corresponds to a memory address, and the contents of the index register define the offset to the operand. In either case, the effective-address is the sum of two values; one is given explicitly in the instruction, and the other is stored in aregister.



| | | |
|---|---|---|
| N | n | |
| LIST | Student ID | Student 1 |
| LIST + 4 | Test 1 | |
| LIST + 8 | Test 2 | |
| LIST + 12 | Test 3 | |
| LIST + 16 | Student ID | Student 2 |
| | Test 1 | |
| | Test 2 | |
| | Test 3 | |

**Figure 2.14** A list of students' marks.



```
        Move        #LIST,R0
        Clear       R1
        Clear       R2
        Clear       R3
        Move        N,R4
LOOP    Add         4(R0),R1
        Add         8(R0),R2
        Add         12(R0),R3
        Add         #16,R0
        Decrement   R4
        Branch>0    LOOP
        Move        R1,SUM1
        Move        R2,SUM2
        Move        R3,SUM3
```

**Figure 2.15** Indexed addressing used in accessing test scores in the list in Figure 2.14.

**Base with Index Mode**

- Another version of the Index mode uses 2 registers which can be denoted as (Ri, Rj)
- Here, a second register may be used to contain the offset X.
- The second register is usually called the *base register*.
- The effective-address of the operand is given by EA=[Ri]+[Rj]
- This form of indexed addressing provides more flexibility in accessing operands because both components of

the effective-address can be changed.

**Base with Index & Offset Mode**

- Another version of the Index mode uses 2 registers plus a constant, which can be denoted as X(Ri, Rj)
- The effective-address of the operand is given by EA=X+[Ri]+[Rj]
- This added flexibility is useful in accessing multiple components inside each item in a record, where the beginning of an item is specified by the (Ri, Rj) part of the addressing-mode. In other words, this mode implements a 3-dimensional array.

**RELATIVE MODE**

- This is similar to index-mode with one difference:

  The effective-address is determined using the PC in place of the general purpose register Ri.
- The operation is indicated as X(PC).
- X(PC) denotes an effective-address of the operand which is X locations above or below the current contents of PC.
- Since the addressed-location is identified "relative" to the PC, the name Relative mode is associated with this type of addressing.
- This mode is used commonly in conditional branch instructions.
- An instruction such as

  *Branch > 0 LOOP* ;Causes program execution to go to the branch target location identified by name LOOP if branch condition is satisfied.

**ADDITIONAL ADDRESSING MODES**

**1) Auto Increment Mode**

➢ Effective-address of operand is contents of a register specified in the instruction (Fig: 2.16).

➢ After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list.

➢ Implicitly, the increment amount is 1.

➢ This mode is denoted as

(Ri)+ ;where Ri=pointer-register.

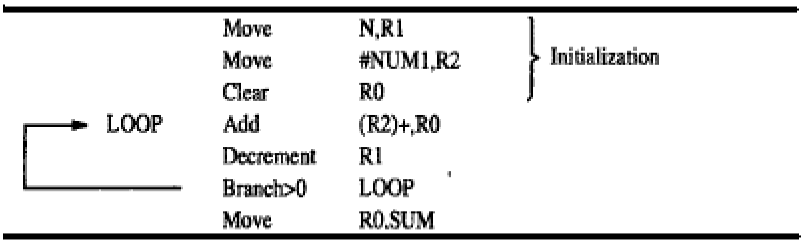
**2) Auto Decrement Mode**

➢ The contents of a register specified in the instruction are first automatically decremented and are then used as the effective-address of the operand.

➢ This mode is denoted as

-(Ri) ;where Ri=pointer-register.

➢ These 2 modes can be used together to implement an important data structure called a stack.



```
              Move        N,R1        }
              Move        #NUM1,R2    } Initialization
              Clear       R0          }
LOOP          Add         (R2)+,R0
              Decrement   R1
              Branch>0    LOOP
              Move        R0,SUM
```

Figure 2.16 The Autoincrement addressing mode used in the program of Figure 2.12.

**ASSEMBLY LANGUAGE**

- We generally use symbolic-names to write a program.
- A complete set of symbolic-names and rules for their use constitute an **Assembly Language**.
- The set of rules for using the mnemonics in the specification of complete instructions and programs is called the **Syntax** of the language.
- Programs written in an assembly language can be automatically translated into a sequence of machine instructions by a program called an **Assembler.**
- The user program in its original alphanumeric text formal is called a **Source Program**, and the assembled machine language program is called an **Object Program**.

For example:

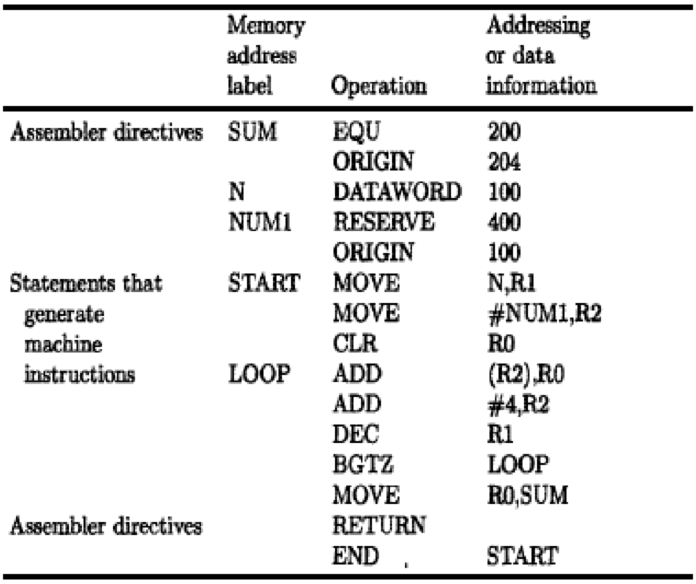
*MOVE R0,SUM* ;The term MOVE represents OP code for operation performed by instruction.

*ADD #5,R3* ;Adds number 5 to contents of register R3 & puts the result back into registerR3.

**ASSEMBLER DIRECTIVES**

- **Directives** are the assembler commands to the assembler concerning the program being assembled.

- These commands are not translated into machine opcode in the object-program.



| | Memory address label | Operation | Addressing or data information |
|---|---|---|---|
| Assembler directives | SUM | EQU | 200 |
| | | ORIGIN | 204 |
| | N | DATAWORD | 100 |
| | NUM1 | RESERVE | 400 |
| | | ORIGIN | 100 |
| Statements that generate machine instructions | START | MOVE | N,R1 |
| | | MOVE | #NUM1,R2 |
| | | CLR | R0 |
| | LOOP | ADD | (R2),R0 |
| | | ADD | #4,R2 |
| | | DEC | R1 |
| | | BGTZ | LOOP |
| | | MOVE | R0,SUM |
| Assembler directives | | RETURN | |
| | | END | START |

**Figure 2.18** Assembly language representation for the program in Figure 2.17.

- **EQU** informs the assembler about the value of an identifier (Figure: 2.18).

  Ex*: SUM EQU 200* ;Informs assembler that the name SUM should be replaced by the value 200.
- **ORIGIN** tells the assembler about the starting-address of memory-area to place the data block.

  Ex*:* ORIGIN 204 ;Instructs assembler to initiate data-block at memory-locations starting from 204.
- **DATAWORD** directive tells the assembler to load a value into the location.

  Ex: *N DATAWORD 100 ;*Informs the assembler to load data 100 into the memory-location N(204).
- **RESERVE** directive is used to reserve a block of memory.

  Ex: *NUM1 RESERVE 400 ;*declares a memory-block of 400 bytes is to be reserved for data.

- **END** directive tells the assembler that this is the end of the source-program text.
- **RETURN** directive identifies the point at which execution of the program should be terminated.
- Any statement that makes instructions or data being placed in a memory-location may be given a **label**. The label(say N or NUM1) is assigned a value equal to the address of that location.

**GENERAL FORMAT OF A STATEMENT**

- Most assembly languages require statements in a source program to be written in the form:

| *Label* | *Operation* | *Operands* | *Comment* |
|---|---|---|---|

1) **Label** is an optional name associated with the memory-address where the machine language instruction produced from the statement will be loaded.
2) **Operation Field** contains the OP-code mnemonic of the desired instruction or assembler.
3) **Operand Field** contains addressing information for accessing one or more operands, depending on the type of instruction.
4) **Comment Field** is used for documentation purposes to make program easier to understand.

**ASSEMBLY AND EXECUTION OF PRGRAMS**

- Programs written in an assembly language are automatically translated into a sequence of machine instructions by the **Assembler**.
- **Assembler Program**

  → replaces all symbols denoting operations & addressing-modes with binary-codes used in machine instructions.

  → replaces all names and labels with their actual values.

  → assigns addresses to instructions & data blocks, starting at address given in ORIGIN directive

  → inserts constants that may be given in DATAWORD directives.

  → reserves memory-space as requested by RESERVE directives.
- **Two Pass Assembler** has 2 passes:

  1) **First Pass:** Work out all the addresses of labels.

  ➢ As the assembler scans through a source-program, it keeps track of all names of numerical- values that correspond to them in a symbol-table.

2) **Second Pass:** Generate machine code, substituting values for the labels.

➢ When a name appears a second time in the source-program, it is replaced with its value from the table.

- The assembler stores the object-program on a magnetic-disk. The object-program must be loaded into the memory of the computer before it is executed. For this, a **Loader Program** is used.
- **Debugger Program** is used to help the user find the programming errors.
- Debugger program enables the user

→ to stop execution of the object-program at some points of interest &

→ to examine the contents of various processor-registers and memory-location.

**STACKS**

- A **stack** is a special type of data structure where elements are inserted from one end and elements are deleted from the same end. This end is called the **top** of the stack (Figure: 2.14).
- The various operations performed on stack:

1) Insert: An element is inserted from top end. Insertion operation is called **push** operation.

2) Delete: An element is deleted from top end. Deletion operation is called **pop** operation.

- A processor-register is used to keep track of the address of the element of the stack that is at the top at any given time. This register is called the **Stack Pointer (SP)**.
- If we assume a byte-addressable memory with a 32-bit word length,

1) The push operation can be implemented as

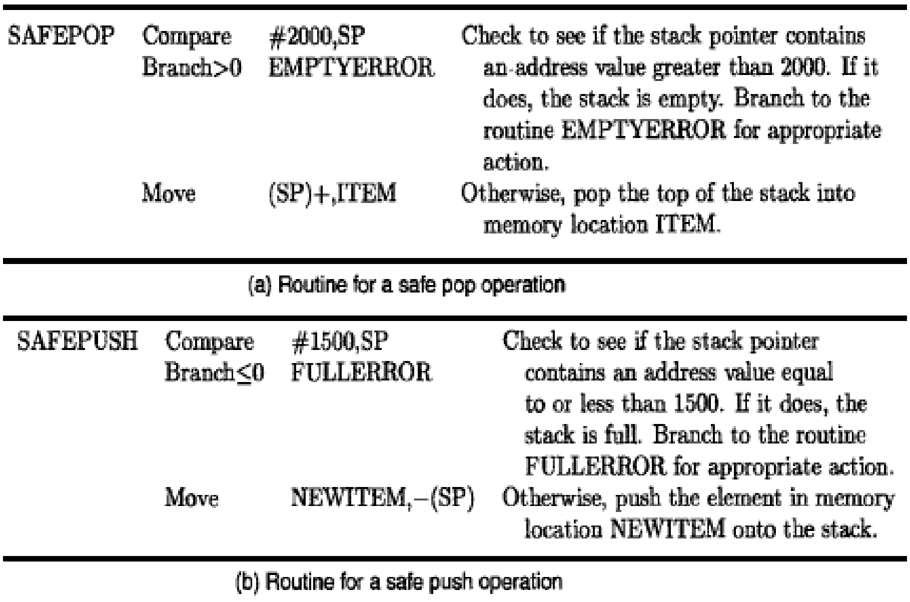
*Subtract #4, SP*

*Move NEWITEM, (SP)*

2) The pop operation can be implemented as

*Move (SP), ITEM Add #4, SP*

- Routine for a safe pop and push operation as follows:



```
SAFEPOP   Compare    #2000,SP       Check to see if the stack pointer contains
          Branch>0   EMPTYERROR       an address value greater than 2000. If it
                                      does, the stack is empty. Branch to the
                                      routine EMPTYERROR for appropriate
                                      action.
          Move       (SP)+,ITEM     Otherwise, pop the top of the stack into
                                      memory location ITEM.
```

(a) Routine for a safe pop operation

```
SAFEPUSH  Compare    #1500,SP         Check to see if the stack pointer
          Branch≤0   FULLERROR          contains an address value equal
                                        to or less than 1500. If it does, the
                                        stack is full. Branch to the routine
                                        FULLERROR for appropriate action.
          Move       NEWITEM,−(SP)    Otherwise, push the element in memory
                                        location NEWITEM onto the stack.
```

(b) Routine for a safe push operation

**Figure 2.23** Checking for empty and full errors in pop and push operations.

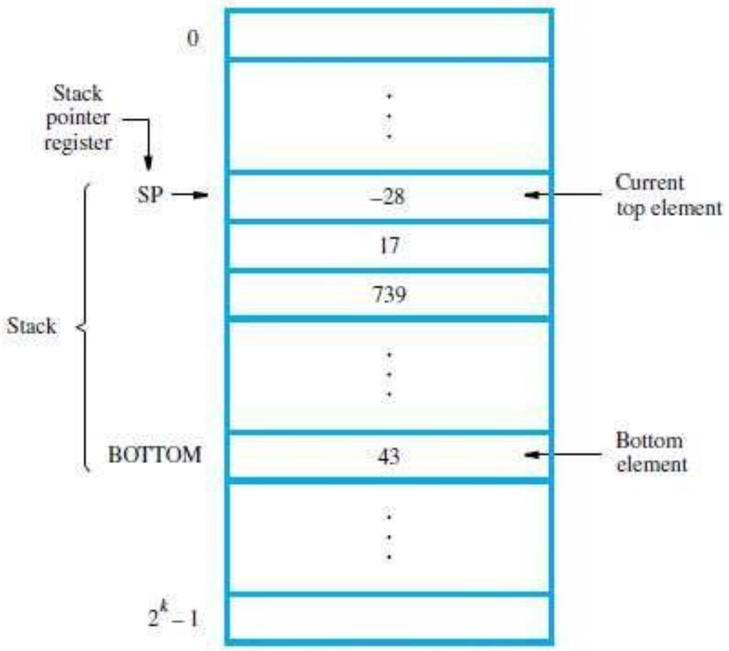


**Figure 2.14** A stack of words in the memory.

## SUBROUTINES

- A subtask consisting of a set of instructions which is executed many times is called a **Subroutine**.
- A Call instruction causes a branch to the subroutine (Figure: 2.16).
- At the end of the subroutine, a return instruction is executed
- Program resumes execution at the instruction immediately following the subroutine call
- The way in which a computer makes it possible to call and return from subroutines is referred to as its **Subroutine Linkage** method.
- The simplest subroutine linkage method is to save the return-address in a specific location, which may be a register dedicated to this function. Such a register is called the **Link Register**.
- When the subroutine completes its task, the Return instruction returns to the calling-program by branching indirectly through the link-register.
- The **Call Instruction** is a special branch instruction that performs the following operations:

→ Store the contents of PC into link-register.

→ Branch to the target-address specified by the instruction.

- The **Return Instruction** is a special branch instruction that performs the operation:

→ Branch to the address contained in the link-register.

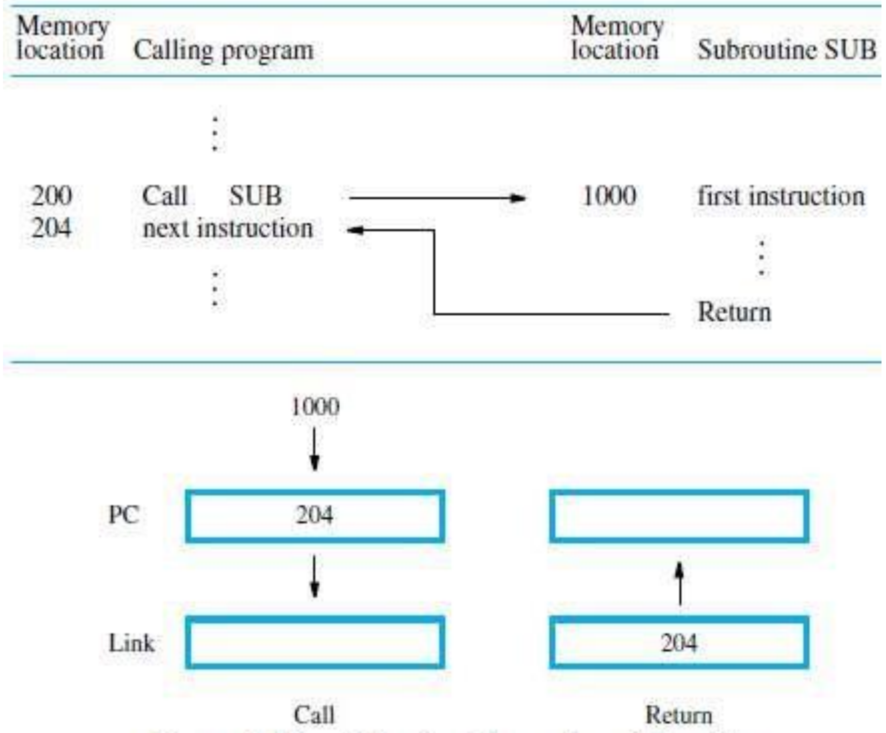


**Figure 2.16** Subroutine linkage using a link register.

**SUBROUTINE NESTING AND THE PROCESSOR STACK**

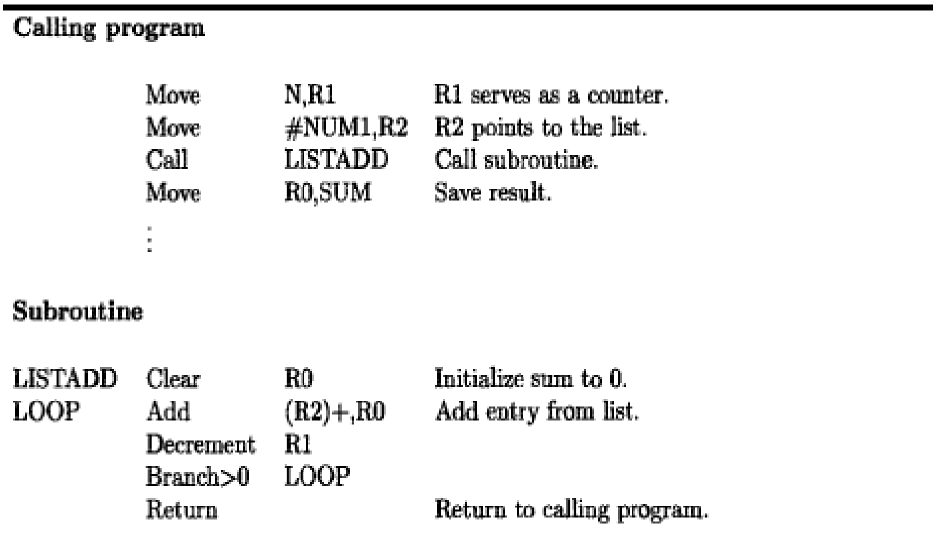
- **Subroutine Nesting** means one subroutine calls another subroutine.
- In this case, the return-address of the second call is also stored in the link-register, destroying its previous contents.
- Hence, it is essential to save the contents of the link-register in some other location before calling another subroutine. Otherwise, the return-address of the first subroutine will be lost.
- Subroutine nesting can be carried out to any depth. Eventually, the last subroutine called completes its computations and returns to the subroutine that called it.
- The return-address needed for this first return is the last one generated in the nested call sequence. That is, return-addresses are generated and used in a LIFO order.
- This suggests that the return-addresses associated with subroutine calls should be pushed onto a stack. A particular register is designated as the SP(Stack Pointer) to be used in this operation.
- SP is used to point to the processor-stack.
- Call instruction pushes the contents of the PC onto the processor-stack.

Return instruction pops the return-address from the processor-stack into the PC.

**PARAMETER PASSING**

- The exchange of information between a calling-program and a subroutine is referred to as **Parameter Passing** (Figure: 2.25).

- The parameters may be placed in registers or in memory-location, where they can be accessed by the subroutine.
- Alternatively, parameters may be placed on the processor-stack used for saving the return-address.
- Following is a program for adding a list of numbers using subroutine with the parameters passed through registers.



**Calling program**

```
          Move        N,R1        R1 serves as a counter.
          Move        #NUM1,R2    R2 points to the list.
          Call        LISTADD     Call subroutine.
          Move        R0,SUM      Save result.
          :
```

**Subroutine**

```
LISTADD   Clear       R0          Initialize sum to 0.
LOOP      Add         (R2)+,R0    Add entry from list.
          Decrement   R1
          Branch>0    LOOP
          Return                  Return to calling program.
```

**Figure 2.25** Program of Figure 2.16 written as a subroutine; parameters passed through registers.

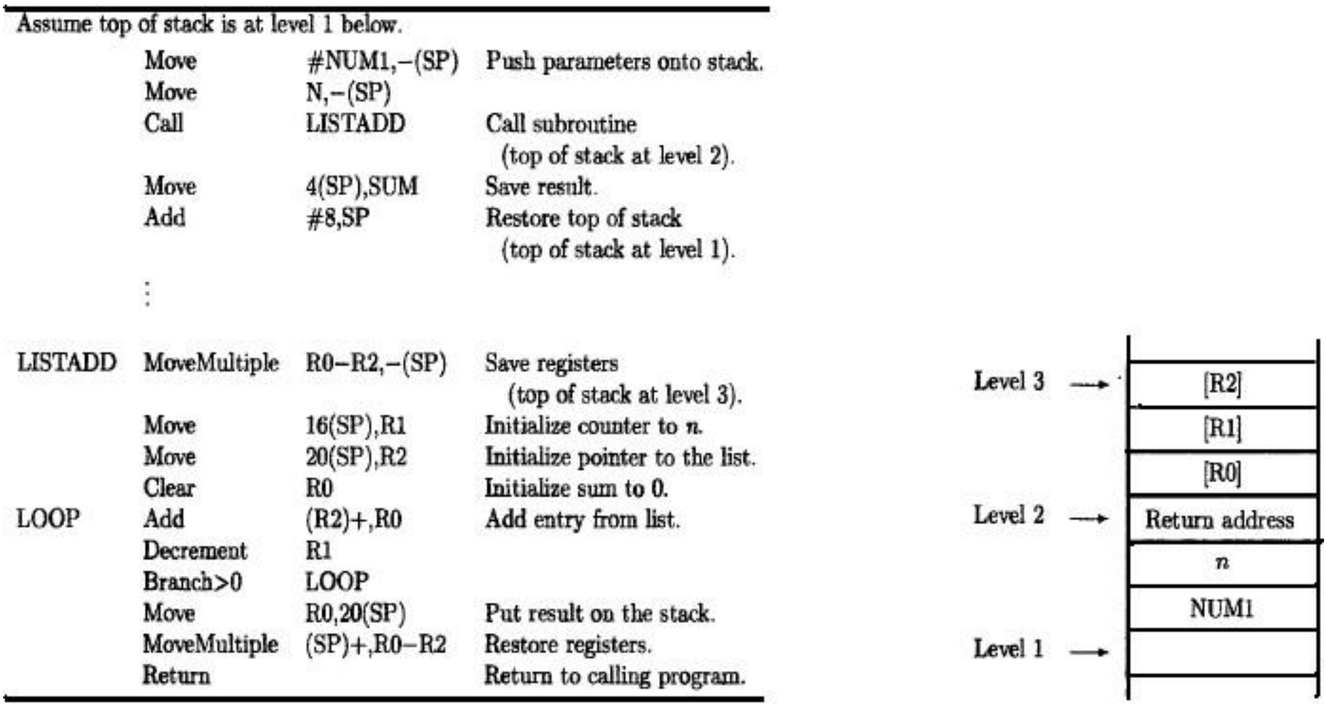
## STACK FRAME

- **Stack Frame** refers to locations that constitute a private work-space for the subroutine.
- The work-space is

  → created at the time the subroutine is entered &

  → freed up when the subroutine returns control to the calling-program (Figure: 2.26).

**Program for adding a list of numbers using subroutine with the parameters passed to stack.**



```
Assume top of stack is at level 1 below.
          Move          #NUM1,-(SP)   Push parameters onto stack.
          Move          N,-(SP)
          Call          LISTADD       Call subroutine
                                        (top of stack at level 2).
          Move          4(SP),SUM     Save result.
          Add           #8,SP         Restore top of stack
                                        (top of stack at level 1).
          ⋮

LISTADD   MoveMultiple  R0-R2,-(SP)   Save registers
                                        (top of stack at level 3).
          Move          16(SP),R1     Initialize counter to n.
          Move          20(SP),R2     Initialize pointer to the list.
          Clear         R0            Initialize sum to 0.
LOOP      Add           (R2)+,R0      Add entry from list.
          Decrement     R1
          Branch>0      LOOP
          Move          R0,20(SP)     Put result on the stack.
          MoveMultiple  (SP)+,R0-R2   Restore registers.
          Return                      Return to calling program.
```

(a) Calling program and subroutine

(b) Top of stack at various times

**Figure 2.26** Program of Figure 2.16 written as a subroutine; parameters passed on the stack.

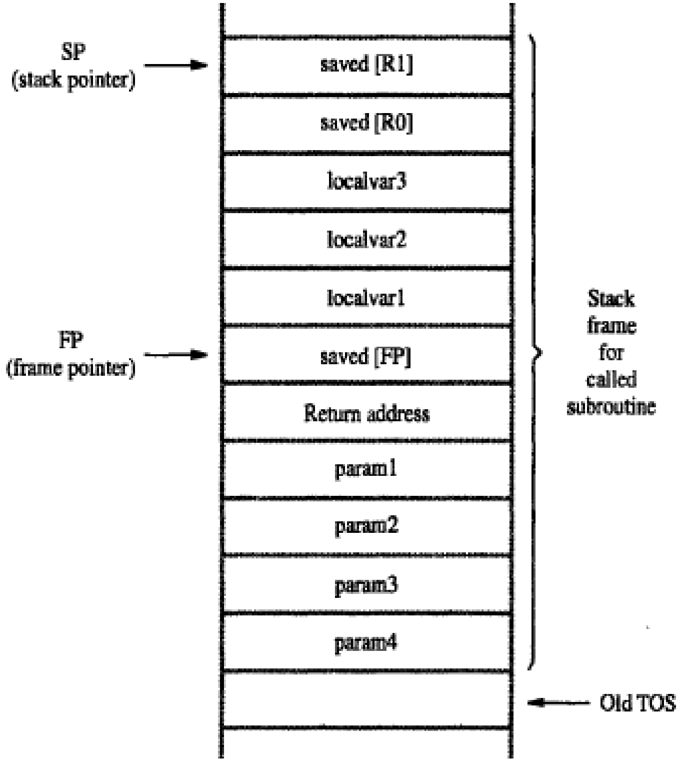


**Figure 2.27** A subroutine stack frame example.

- Fig: 2.27 show an example of a commonly used layout for information in a stack-frame.

- **Frame Pointer (FP)** is used to access the parameters passed

  → to the subroutine &

  → to the local memory-variables.

- The contents of FP remains fixed throughout the execution of the subroutine, unlike stack-pointer SP, which must always point to the current top element in the stack.

**Operation on Stack Frame**

- Initially SP is pointing to the address of oldTOS.
- The calling-program saves 4 parameters on the stack (Figure 2.27).
- The Call instruction is now executed, pushing the return-address onto the stack.
- Now, SP points to this return-address, and the first instruction of the subroutine is executed.
- Now, FP is to be initialized and its old contents have to be stored. Hence, the first 2 instructions in the subroutine are:

  *Move FP,-(SP)*

  *Move SP,FP*

- The FP is initialized to the value of SP i.e. both FP and SP point to the saved FP address.
- The 3 local variables may now be pushed onto the stack. Space for local variables is allocated by executing the instruction

  *Subtract #12,SP*

- Finally, the contents of processor-registers R0 and R1 are saved in the stack. At this point, the stack- frame has been set up as shown in the fig 2.27.
- The subroutine now executes its task. When the task is completed, the subroutine pops the saved values of R1 and R0 back into those registers, removes the local variables from the stack frame by executing the instruction.

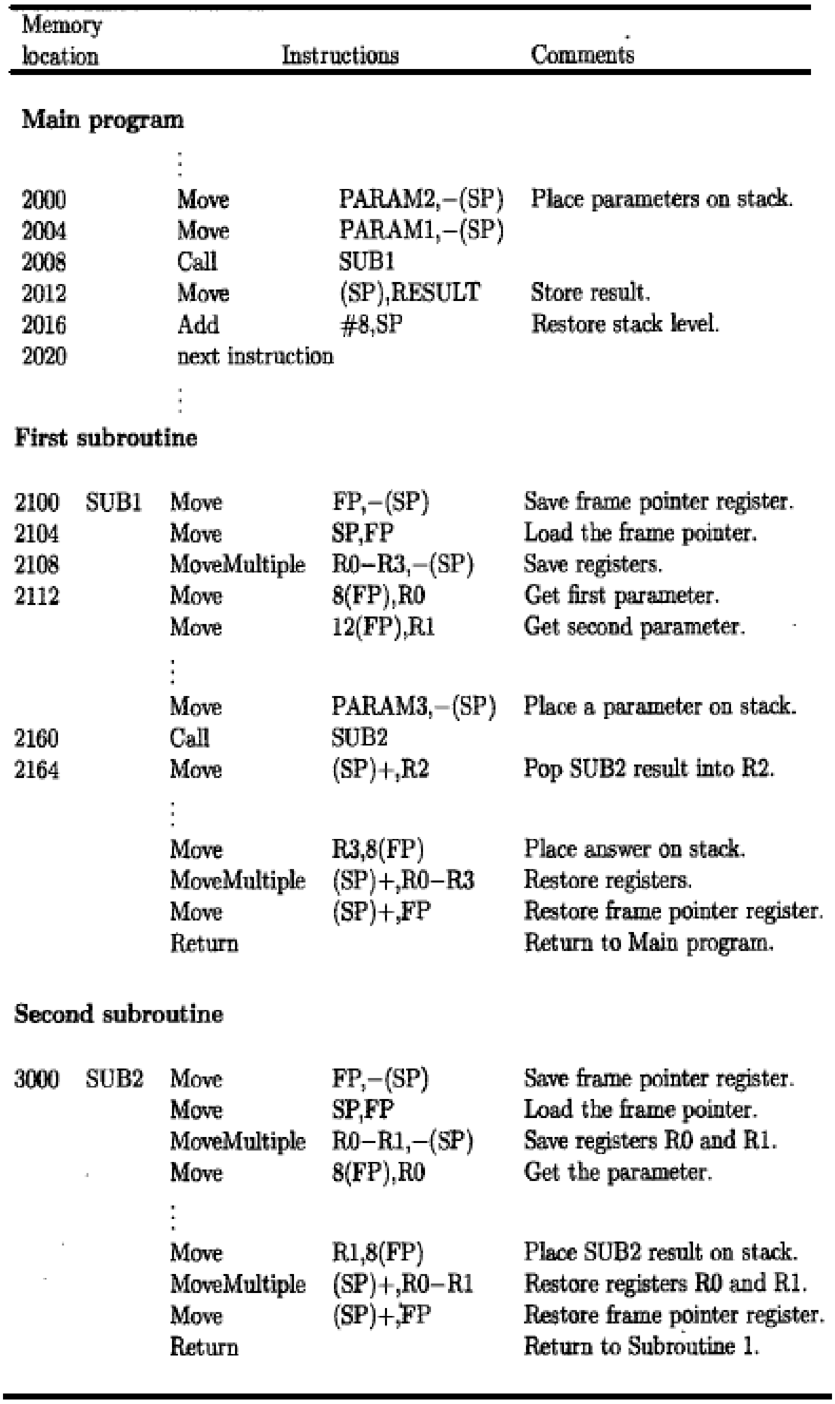
  *Add #12, SP*

- And subroutine pops saved old value of FP back into FP. At this point, SP points to return-address, so the Return instruction can be executed, transferring control back to the calling-program.

**STACK FRAMES FOR NESTED SUBROUTINES**

- Stack is very useful data structure for holding return-addresses when subroutines are nested.
- When nested subroutines are used; the stack-frames are built up in the processor-stack.

**Program to illustrate stack frames for nested subroutines**



```
Memory
location          Instructions                   Comments

Main program
                  :
2000              Move          PARAM2,-(SP)     Place parameters on stack.
2004              Move          PARAM1,-(SP)
2008              Call          SUB1
2012              Move          (SP),RESULT      Store result.
2016              Add           #8,SP            Restore stack level.
2020              next instruction
                  :
First subroutine

2100   SUB1       Move          FP,-(SP)         Save frame pointer register.
2104              Move          SP,FP            Load the frame pointer.
2108              MoveMultiple  R0-R3,-(SP)      Save registers.
2112              Move          8(FP),R0         Get first parameter.
                  Move          12(FP),R1        Get second parameter.
                  :
                  Move          PARAM3,-(SP)     Place a parameter on stack.
2160              Call          SUB2
2164              Move          (SP)+,R2         Pop SUB2 result into R2.
                  :
                  Move          R3,8(FP)         Place answer on stack.
                  MoveMultiple  (SP)+,R0-R3      Restore registers.
                  Move          (SP)+,FP         Restore frame pointer register.
                  Return                         Return to Main program.

Second subroutine

3000   SUB2       Move          FP,-(SP)         Save frame pointer register.
                  Move          SP,FP            Load the frame pointer.
                  MoveMultiple  R0-R1,-(SP)      Save registers R0 and R1.
                  Move          8(FP),R0         Get the parameter.
                  :
                  Move          R1,8(FP)         Place SUB2 result on stack.
                  MoveMultiple  (SP)+,R0-R1      Restore registers R0 and R1.
                  Move          (SP)+,FP         Restore frame pointer register.
                  Return                         Return to Subroutine 1.
```

**Figure 2.28** Nested subroutines.

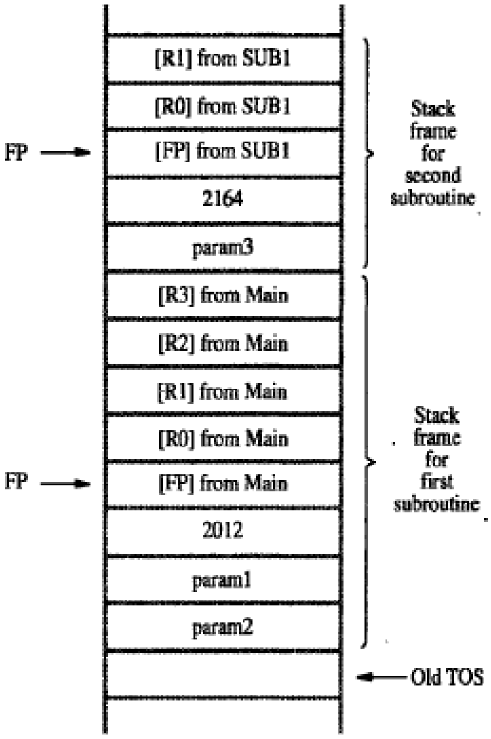


**Figure 2.29** Stack frames for Figure 2.28.

**The Flow of Execution is as follows:**

- Main program pushes the 2 parameters param2 and param1 onto the stack and then calls SUB1.
- SUB1 has to perform an operation & send result to the main-program on the stack (Fig:2.28 & 29).
- During the process, SUB1 calls the second subroutine SUB2 (in order to perform some subtask).
- After SUB2 executes its Return instruction; the result is stored in register R2 by SUB1.
- SUB1 then continues its computations & eventually passes required answer back to main-program on thestack.
- When SUB1 executes return statement, the main-program stores this answers in memory-location RESULT and continues its execution.

# BASIC PROCESSING UNIT

## SOME FUNDAMENTAL CONCEPTS

- To execute an instruction, processor has to perform following 3 steps:
  1) Fetch contents of memory-location pointed to by PC. Content of this location is an instruction to be executed. The instructions are loaded into IR, Symbolically, this operation is written as:
     IR□ [[PC]]
  2) Increment PC by 4.
     PC□ [PC] +4
  3) Carry out the actions specified by instruction (in the IR).
- The first 2 steps are referred to as **Fetch Phase**.
- Step 3 is referred to as **Execution Phase**.
- The operation specified by an instruction can be carried out by performing one or more of the following actions:
  1) Read the contents of a given memory-location and load them into a register.
  2) Read data from one or more registers.
  3) Perform an arithmetic or logic operation and place the result into a register.
  4) Store data from a register into a given memory-location.

## SINGLE BUS ORGANIZATION

- ALU and all the registers are interconnected via a **Single Common Bus** (Figure 7.1).
- Data & address lines of the external memory-bus is connected to the internal processor-bus via MDR & MAR respectively. (MDR□ Memory Data Register, MAR □ Memory Address Register).
- **MDR** has 2 inputs and 2 outputs. Data may be loaded
  → into MDR either from memory-bus (external) or
  → from processor-bus (internal).
- **MAR**'s input is connected to internal-bus; MAR's output is connected to external-bus.
- **Instruction Decoder & Control Unit** is responsible for
  → issuing the control-signals to all the units inside the processor.
  → implementing the actions specified by the instruction (loaded in the IR).
- Register R0 through R(n-1) are the **Processor Registers**.
  The programmer can access these registers for general-purpose use.
- Only processor can access 3 registers **Y**, **Z** & **Temp** for temporary storage during program-execution. The programmer cannot access these 3 registers.
- In **ALU**, 1) „A‟ input gets the operand from the output of the multiplexer (MUX).
  2) „B‟ input gets the operand directly from the processor-bus.
- There are 2 options provided for „A‟ input of the ALU.
- MUX is used to select one of the 2 inputs.
- **MUX** selects either
  → output of Y or
  → constant-value 4( which is used to increment PC content).

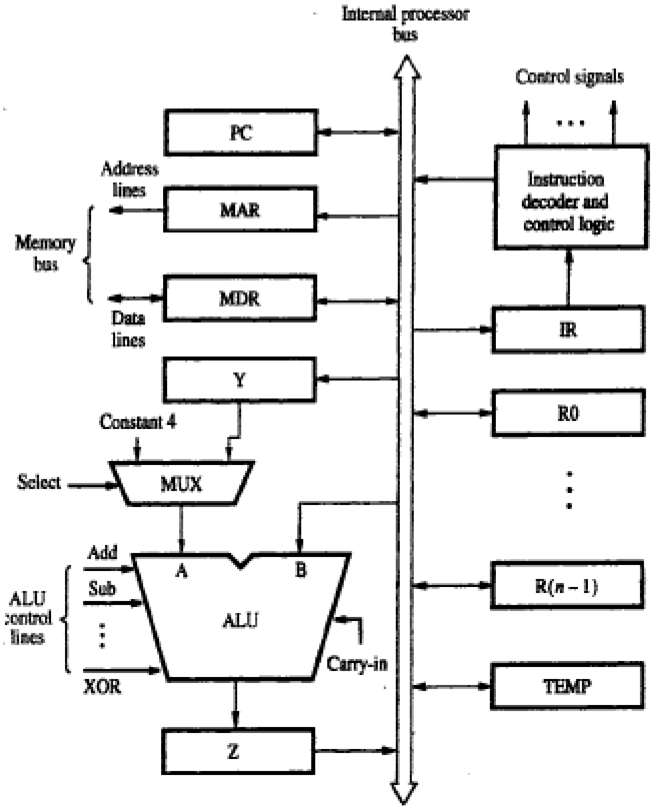


**Figure 7.1** Single-bus organization of the datapath inside a processor.

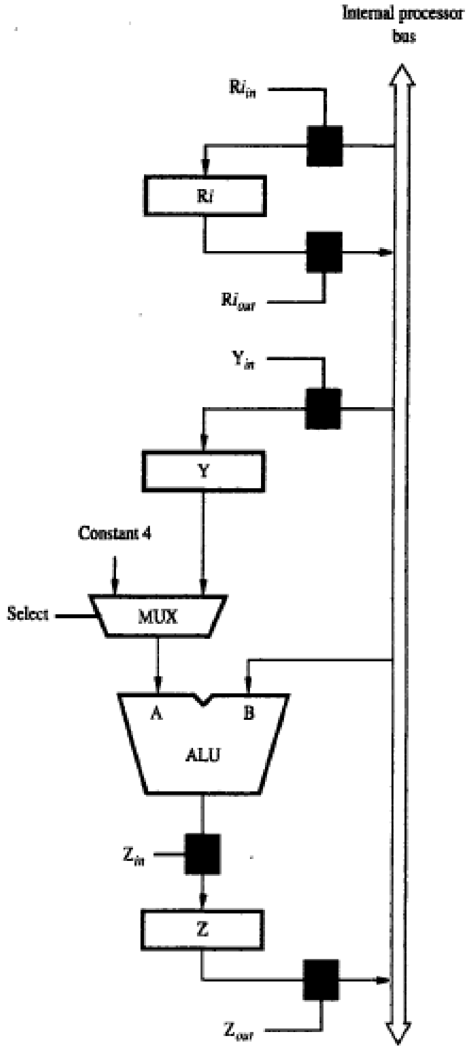
- An instruction is executed by performing one or more of the following operations:
  1) Transfer a word of data from one register to another or to the ALU.
  2) Perform arithmetic or a logic operation and store the result in a register.
  3) Fetch the contents of a given memory-location and load them into a register.
  4) Store a word of data from a register into a given memory-location.
- **Disadvantage:** Only one data-word can be transferred over the bus in a clock cycle.

**Solution:** Provide multiple internal-paths. Multiple paths allow several data-transfers to take place in parallel.

## REGISTER TRANSFERS

- Instruction execution involves a sequence of steps in which data are transferred from one register to another.
- For each register, two control-signals are used: $Ri_{in}$ & $Ri_{out}$. These are called **Gating Signals.**
- $Ri_{in}$=1  data on bus is loaded into Ri. $Ri_{out}$=1  content of Ri is placed on bus.

  $Ri_{out}$=0,  bus can be used for transferring data from other registers.
- For example, *Move R1, R2;* This transfers the contents of register R1 to register R2. This can be accomplished as follows:
  1) Enable the output of registers R1 by setting $R1_{out}$ to 1 (Figure 7.2). This places the contents of R1 on processor-bus.
  2) Enable the input of register R2 by setting $R2_{out}$ to 1. This loads data from processor-bus into register R4.
- All operations and data transfers within the processor take place within time-periods defined by the **processor-clock**.
- The control-signals that govern a particular transfer are asserted at the start of the clock cycle.



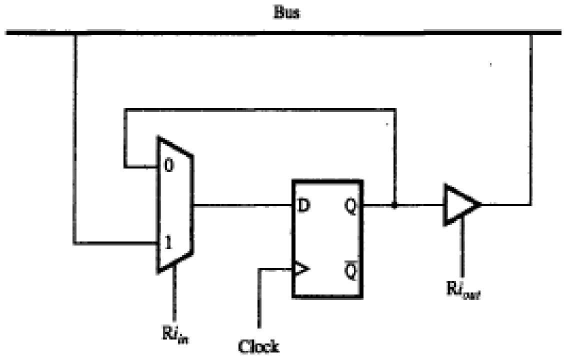


**Figure 7.2** Input and output gating for the registers in Figure 7.1.

**Figure 7.3** Input and output gating for one register bit.

**Input & Output Gating for one Register Bit**

- A 2-input multiplexer is used to select the data applied to the input of an edge-triggered D flip-flop.
- $Ri_{in}$=1 □ mux selects data on bus. This data will be loaded into flip-flop at rising-edge of clock. $Ri_{in}$=0 □ mux feeds back the value currently stored in flip-flop (Figure7.3).
- Q output of flip-flop is connected to bus via a tri-state gate. $Ri_{out}$=0 □ gate's output is in the high-impedance state.
  $Ri_{out}$=1 □ the gate drives the bus to 0 or 1, depending on the value of Q.

**PERFORMING AN ARITHMETIC OR LOGIC OPERATION**

- The ALU performs arithmetic operations on the 2 operands applied to its A and B inputs.
- One of the operands is output of MUX;
  And, the other operand is obtained directly from processor-bus.
- The result (produced by the ALU) is stored temporarily in register Z.
- The sequence of operations for [R3]□[R1]+[R2] is as follows:
  1) $R1_{out}$, $Y_{in}$
  2) $R2_{out}$, SelectY, Add, $Z_{in}$
  3) $Z_{out}$, $R3_{in}$
- Instruction execution proceeds as follows:
  Step 1 --> Contents from register R1 are loaded into register Y.
  Step2 --> Contents from Y and from register R2 are applied to the A and B inputs of ALU; Addition is performed &
  Result is stored in the Z register.
  Step 3 --> The contents of Z register is stored in the R3 register.
- The signals are activated for the duration of the clock cycle corresponding to that step. All other signals are inactive.

**CONTROL-SIGNALS OF MDR**

- The MDR register has 4 control-signals (Figure 7.4):
  1) $MDR_{in}$ & $MDR_{out}$ control the connection to the internal processor data bus &
  2) $MDR_{inE}$ & $MDR_{outE}$ control the connection to the memory Data bus.
- MAR register has 2 control-signals.
  1) $MAR_{in}$ controls the connection to the internal processor address bus &
  2) $MAR_{out}$ controls the connection to the memory address bus.

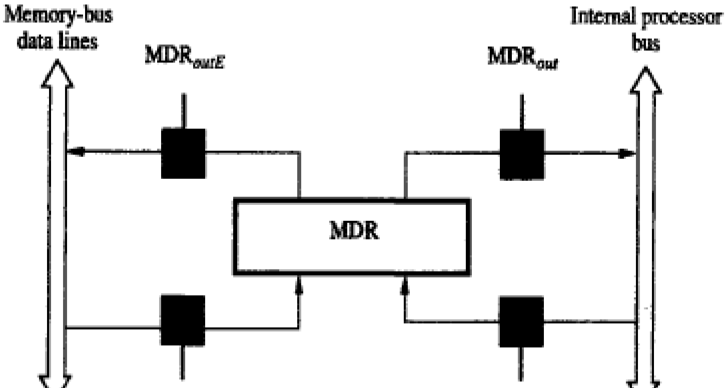


**Figure 7.4** Connection and control signals for register MDR.

**FETCHING A WORD FROM MEMORY**

- To fetch instruction/data from memory, processor transfers required address to MAR. At the same time, processor issues Read signal on control-lines of memory-bus.
- When requested-data are received from memory, they are stored in MDR. From MDR, they are transferred to other registers.
- The response time of each memory access varies (based on cache miss, memory-mapped I/O). To accommodate this, MFC is used. (MFC □ Memory Function Completed).
- MFC is a signal sent from addressed-device to the processor. MFC informs the processor that the requested operation has been completed by addressed-device.
- Consider the instruction Move (R1),R2. The sequence of steps is (Figure 7.5):
  1) $R1_{out}$, $MAR_{in}$, Read ;desired address is loaded into MAR & Read command is issued.

2) $MDR_{inE}$, WMFC ;load MDR from memory-bus & Wait for MFC response from memory.
3) $MDR_{out}$, $R2_{in}$ ;load R2 from MDR.

where WMFC=control-signal that causes processor's control. circuitry to wait for arrival of MFC signal.

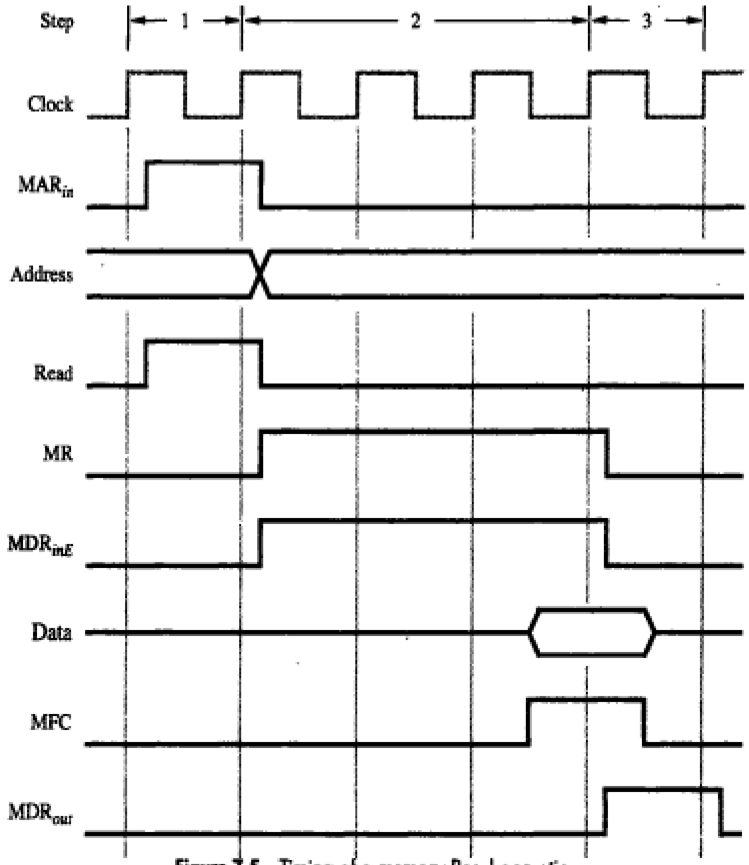


Figure 7.5 Timing of a memory Read operation.

**Storing a Word in Memory**

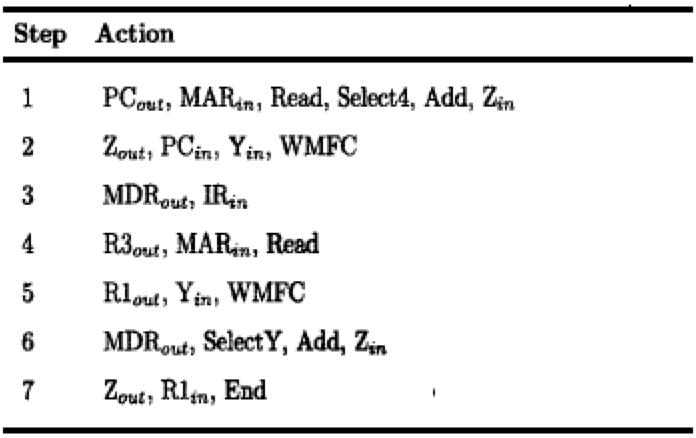
- Consider the instruction *Move R2,(R1).* This requires the following sequence:

1) $R1_{out}$, $MAR_{in}$ ;desired address is loaded into MAR.
2) $R2_{out}$, $MDR_{in}$, Write ;data to be written are loaded into MDR & Write command is issued.
3) $MDR_{outE}$, WMFC ;load data into memory-location pointed by R1 from MDR.

**EXECUTION OF A COMPLETE INSTRUCTION**

- Consider the instruction *Add (R3),R1* which adds the contents of a memory-location pointed by R3 to register R1. Executing this instruction requires the following actions:

1) Fetch the instruction.
2) Fetch the first operand.
3) Perform the addition &
4) Load the result into R1.



| Step | Action |
|---|---|
| 1 | $PC_{out}$, $MAR_{in}$, Read, Select4, Add, $Z_{in}$ |
| 2 | $Z_{out}$, $PC_{in}$, $Y_{in}$, WMFC |
| 3 | $MDR_{out}$, $IR_{in}$ |
| 4 | $R3_{out}$, $MAR_{in}$, Read |
| 5 | $R1_{out}$, $Y_{in}$, WMFC |
| 6 | $MDR_{out}$, SelectY, Add, $Z_{in}$ |
| 7 | $Z_{out}$, $R1_{in}$, End |

**Figure 7.6** Control sequence for execution of the instruction Add (R3),R1

- Instruction execution proceeds as follows:

Step1--> The instruction-fetch operation is initiated by

→ loading contents of PC into MAR &

→ sending a Read request to memory.

The Select signal is set to Select4, which causes the Mux to select constant 4. This value is added to operand at input B (PC‟s content), and the result is stored in Z.

Step2--> Updated value in Z is moved to PC. This completes the PC increment operation and PC will now point to next instruction.

Step3--> Fetched instruction is moved into MDR and then to IR. The step 1 through 3 constitutes the **Fetch Phase**.

At the beginning of step 4, the instruction decoder interprets the contents of the IR. This enables the control circuitry to activate the control-signals for steps 4 through 7.

The step 4 through 7 constitutes **the Execution Phase.**

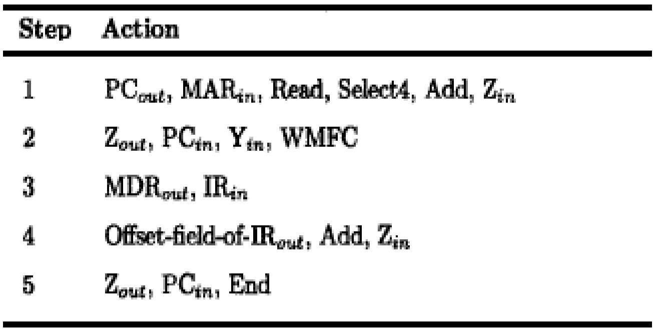
Step4--> Contents of R3 are loaded into MAR & a memory read signal is issued. Step5--> Contents of R1 are transferred to Y to prepare for addition.

Step6--> When Read operation is completed, memory-operand is available in MDR, and the addition is performed.

Step7--> Sum is stored in Z, then transferred to R1.The End signal causes a new instruction fetch cycle to begin by returning to step1.

## BRANCHING INSTRUCTIONS

- Control sequence for an **unconditional branch instruction** is as follows:



| Step | Action |
|---|---|
| 1 | $PC_{out}$, $MAR_{in}$, Read, Select4, Add, $Z_{in}$ |
| 2 | $Z_{out}$, $PC_{in}$, $Y_{in}$, WMFC |
| 3 | $MDR_{out}$, $IR_{in}$ |
| 4 | Offset-field-of-$IR_{out}$, Add, $Z_{in}$ |
| 5 | $Z_{out}$, $PC_{in}$, End |

**Figure 7.7** Control sequence for an unconditional Branch instruction.

- Instruction execution proceeds as follows:

Step 1-3--> The processing starts & the fetch phase ends in step3.

Step 4--> The offset-value is extracted from IR by instruction-decoding circuit.

Since the updated value of PC is already available in register Y, the offset X is gated onto the bus, and an addition operation is performed.

Step 5--> the result, which is the branch-address, is loaded into the PC.

- The branch instruction loads the branch target address in PC so that PC will fetch the next instruction from the branch target address.
- The branch target address is usually obtained by adding the offset in the contents of PC.
- The offset X is usually the difference between the branch target-address and the address immediately following the branch instruction.
- In case of **conditional branch**,

we have to check the status of the condition-codes before loading a new value into the PC. e.g.: Offset-field-of-$IR_{out}$, Add, $Z_{in,}$ If N=0 then End

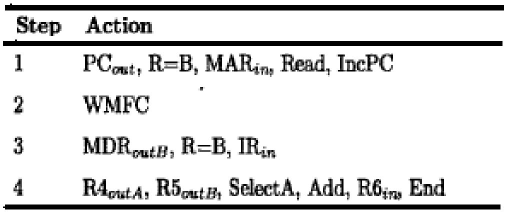
If N=0, processor returns to step 1 immediately after step 4. If N=1, step 5 is performed to load a new value into PC.

## MULTIPLE BUS ORGANIZATION

- **Disadvantage of Single-bus organization:** Only one data-word can be transferred over the bus in a clock cycle. This increases the steps required to complete the execution of the instruction

**Solution:** To reduce the number of steps, most processors provide multiple internal-paths. Multiple paths enable several transfers to take place in parallel.

- As shown in fig 7.8, three buses can be used to connect registers and the ALU of the processor.
- All general-purpose registers are grouped into a single block called the **Register File**.
- Register-file has 3 ports:
    1) Two output-ports allow the contents of 2 different registers to be simultaneously placed on buses A & B.
    2) Third input-port allows data on bus C to be loaded into a third register during the same clock-cycle.
- Buses A and B are used to transfer source-operands to A & B inputs of ALU.
- The result is transferred to destination over bus C.
- **Incrementer Unit** is used to increment PC by 4.



| Step | Action |
|---|---|
| 1 | $PC_{out}$, R=B, $MAR_{in}$, Read, IncPC |
| 2 | WMFC |
| 3 | $MDR_{outB}$, R=B, $IR_{in}$ |
| 4 | $R4_{outA}$, $R5_{outB}$, SelectA, Add, $R6_{in}$, End |

**Figure 7.9** Control sequence for the instruction Add R4,R5,R6

- Instruction execution proceeds as follows:

Step 1--> Contents of PC are

→ passed through ALU using R=B control-signal &

→ loaded into MAR to start memory Read operation. At the same time, PC is incremented by 4.

Step2--> Processor waits for MFC signal from memory.

Step3--> Processor loads requested-data into MDR, and then transfers them to IR.

Step4--> The instruction is decoded and add operation takes place in a single STEP

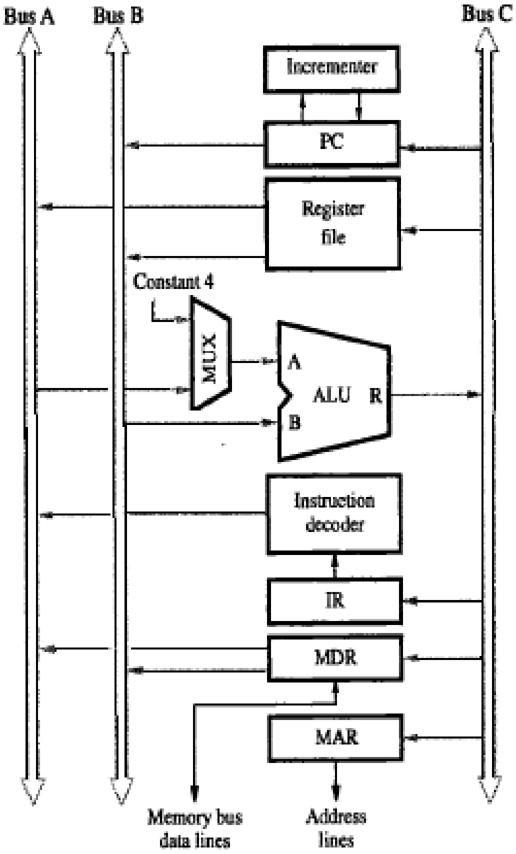


**Figure 7.8** Three-bus organization of the datapath.

# UNIT III

To execute instructions, the processor must have some means of generating the control-signals. There are two approaches for this purpose:

1) Hardwired control and 2) Microprogrammed control.

**HARDWIRED CONTROL**

- Hardwired control is a method of control unit design (Figure 7.11).
- The control-signals are generated by using logic circuits such as gates, flip-flops, decoders etc.
- **Decoder/Encoder Block** is a combinational-circuit that generates required control-outputs depending on state of all its inputs.
- **Instruction Decoder**
  - It decodes the instruction loaded in the IR.
  - If IR is an 8 bit register, then instruction decoder generates $2^8$(256 lines); one for each instruction.
  - It consists of a separate output-lines $INS_1$ through $INS_m$ for each machine instruction.
  - According to code in the IR, one of the output-lines $INS_1$ through $INS_m$ is set to 1, and all other lines are set to 0.
- **Step-Decoder** provides a separate signal line for each step in the control sequence.
- **Encoder**
  - It gets the input from instruction decoder, step decoder, external inputs and condition codes.
  - It uses all these inputs to generate individual control-signals: $Y_{in}$, $PC_{out}$, Add, End and so on.
  - For example (Figure 7.12), $Z_{in}=T_1+T_6.ADD+T_4.BR$

    ;This signal is asserted during time-slot $T_1$ for all instructions.<br>
    during $T_6$ for an Add instruction.<br>
    during $T_4$ for unconditional branch instruction
- When **RUN**=1, counter is incremented by 1 at the end of every clock cycle. When RUN=0, counter stops counting.
- After execution of each instruction, **end** signal is generated. End signal resets step counter.
- Sequence of operations carried out by this machine is determined by wiring of logic circuits, hence the name "**hardwired**".

- **Advantage**: Can operate at high speed.
- **Disadvantages:**
  1) Since no. of instructions/control-lines is often in hundreds, the complexity of control unit is very high.
  2) It is costly and difficult to design.
  3) The control unit is inflexible because it is difficult to change the design.

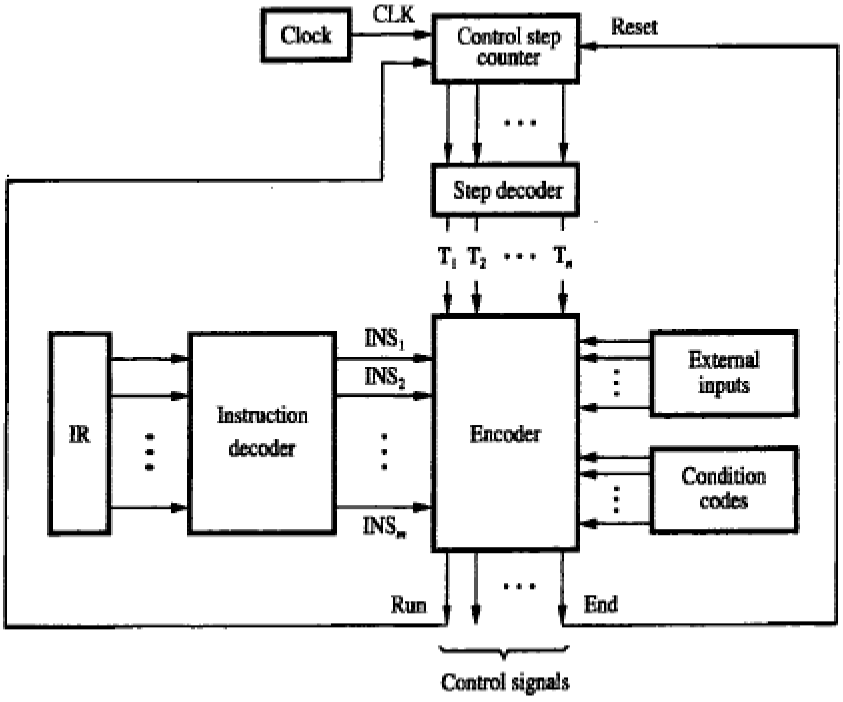


**Figure 7.11** Separation of the decoding and encoding functions.

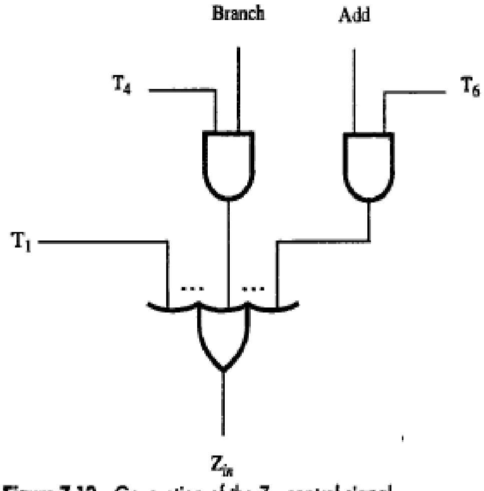
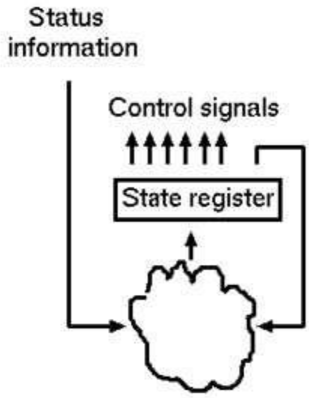
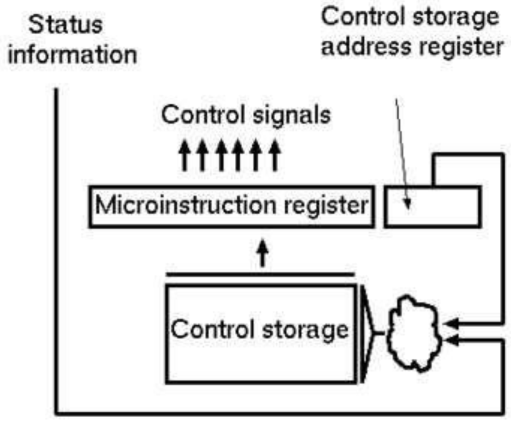


**Figure 7.12** Generation of the $Z_{in}$ control signal

## HARDWIRED CONTROL VS MICROPROGRAMMED CONTROL

| Attribute | Hardwired Control | Microprogrammed Control |
|---|---|---|
| **Definition** | Hardwired control is a control mechanism to generate control- signals by using gates, flip- flops, decoders, and other digital circuits. | Micro programmed control is a control mechanism to generate control-signals by using a memory called control store (CS), which contains the control- signals. |
| **Speed** | Fast | Slow |
| **Control functions** | Implemented in hardware. | Implemented in software. |
| **Flexibility** | Not flexible to accommodate new system specifications or new instructions. | More flexible, to accommodate new system specification or new instructions redesign is required. |
| **Ability to handle large or complex instruction sets** | Difficult. | Easier. |
| **Ability to support operating systems & diagnostic features** | Very difficult. | Easy. |
| **Design process** | Complicated. | Orderly and systematic. |
| **Applications** | Mostly RISC microprocessors. | Mainframes, some microprocessors. |
| **Instructionset size** | Usually under 100 instructions. | Usually over 100 instructions. |
| **ROM size** | - | 2K to 10K by 20-400 bit microinstructions. |
| **Chip area efficiency** | Uses least area. | Uses more area. |
| **Diagram** |  |  Status information<br>Control storage address register<br>Control signals<br>Microinstruction register<br>Control storage |

## MICROPROGRAMMED CONTROL

- Microprogramming is a method of control unit design (Figure 7.16).
- Control-signals are generated by a program similar to machine language programs.
- **Control Word(CW*)*** is a word whose individual bits represent various control-signals (like Add, $PC_{in}$).
- Each of the control-steps in control sequence of an instruction defines a unique combination of 1s & 0s in CW.
- Individual control-words in microroutine are referred to as **microinstructions** (Figure 7.15).
- A sequence of CWs corresponding to control-sequence of a machine instruction constitutes the **microroutine.**
- The microroutines for all instructions in the instruction-set of a computer are stored in a special memory called the **Control Store (CS)**.
- Control-unit generates control-signals for any instruction by sequentially reading CWs of corresponding microroutine from CS.

- **µPC** is used to read CWs sequentially from CS. (µPC□ Microprogram Counter).
- Every time new instruction is loaded into IR, o/p of **Starting Address Generator** is loaded into µPC.
- Then, µPC is automatically incremented by clock;

  causing successive microinstructions to be read from CS.

  Hence, control-signals are delivered to various parts of processor in correct sequence.

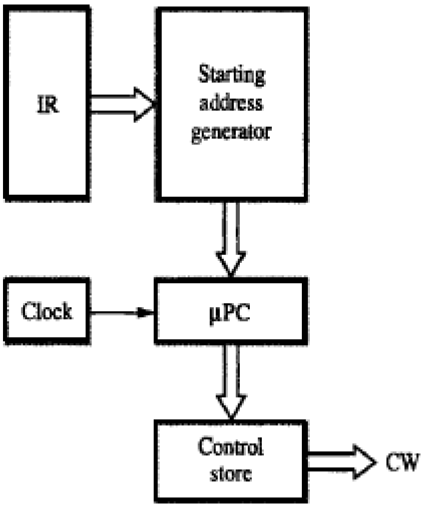
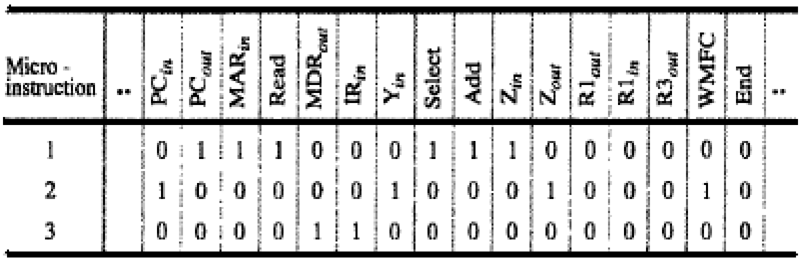


Figure 7.16 Basic organization of a microprogrammed control unit.



| Micro - instruction | •• | $PC_{in}$ | $PC_{out}$ | $MAR_{in}$ | Read | $MDR_{out}$ | $IR_{in}$ | $Y_{in}$ | Select | Add | $Z_{in}$ | $Z_{out}$ | $R1_{out}$ | $R1_{in}$ | $R3_{out}$ | WMFC | End | •• |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 2 | | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | |
| 3 | | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Figure 7.15 An example of microinstructions for Figure 7.6.

**Advantages**

- It simplifies the design of control unit. Thus it is both, cheaper and less error prone implement.
- Control functions are implemented in software rather than hardware.
- The design process is orderly and systematic.
- More flexible, can be changed to accommodate new system specifications or to correct the design errors quickly and cheaply.
- Complex function such as floating point arithmetic can be realized efficiently.

**Disadvantages**

- A microprogrammed control unit is somewhat slower than the hardwired control unit, because time is required to access the microinstructions from CM.
- The flexibility is achieved at some extra hardware cost due to the control memory and its access circuitry.

## ORGANIZATION OF MICROPROGRAMMED CONTROL UNIT TO SUPPORT CONDITIONAL BRANCHING

- **Drawback of previous Microprogram control:**
  - ➢ It cannot handle the situation when the control unit is required to check the status of the condition codes or external inputs to choose between alternative courses of action.

**Solution:**

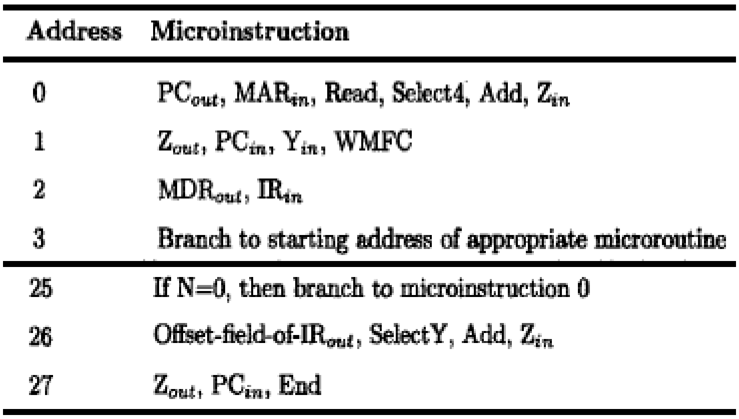
  - ➢ Use conditional branch microinstruction.

- In case of conditional branching, microinstructions specify which of the external inputs, condition- codes should be checked as a condition for branching to take place.
- **Starting and Branch Address Generator Block** loads a new address into µPC when a microinstruction instructs it to do so (Figure 7.18).
- To allow implementation of a conditional branch, inputs to this block consist of

  → external inputs and condition-codes &

  → contents of IR.

- µPC is incremented every time a new microinstruction is fetched from microprogram memory except in following situations:
  1) When a new instruction is loaded into IR, µPC is loaded with starting-address of microroutine for that instruction.
  2) When a Branch microinstruction is encountered and branch condition is satisfied, µPC is loaded with branch-address.
  3) When an End microinstruction is encountered, µPC is loaded with address of first CW in microroutine for instruction fetch cycle.



| Address | Microinstruction |
|---|---|
| 0 | $PC_{out}$, $MAR_{in}$, Read, Select4, Add, $Z_{in}$ |
| 1 | $Z_{out}$, $PC_{in}$, $Y_{in}$, WMFC |
| 2 | $MDR_{out}$, $IR_{in}$ |
| 3 | Branch to starting address of appropriate microroutine |
| 25 | If N=0, then branch to microinstruction 0 |
| 26 | Offset-field-of-$IR_{out}$, SelectY, Add, $Z_{in}$ |
| 27 | $Z_{out}$, $PC_{in}$, End |

**Figure 7.17** Microroutine for the instruction Branch < 0.

**MICROINSTRUCTIONS**

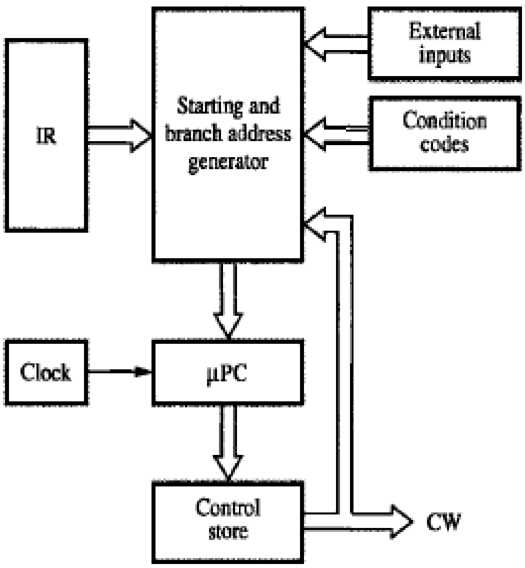
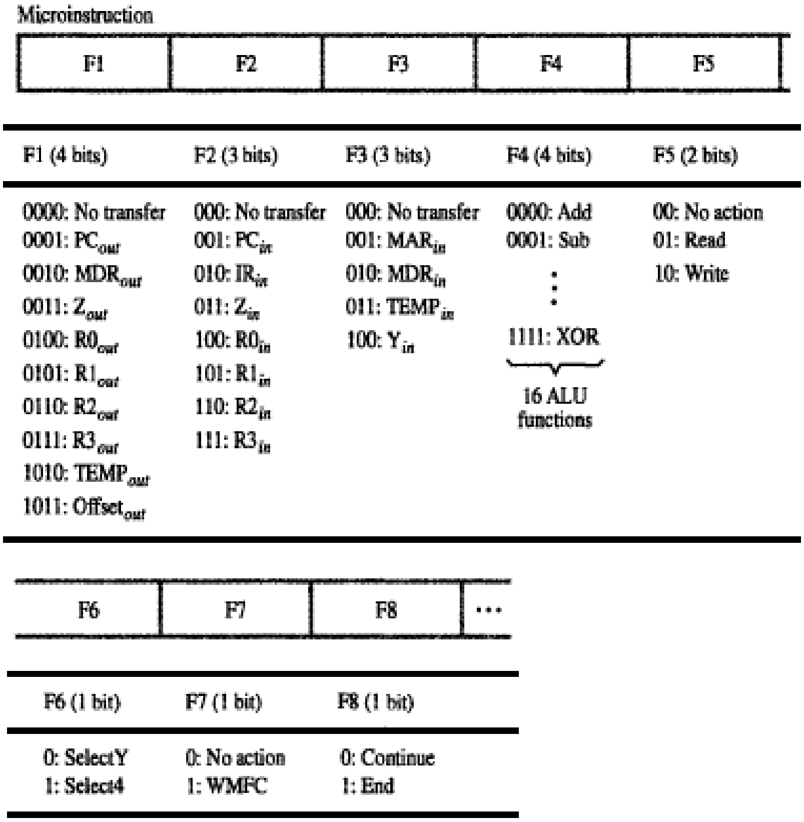


**Figure 7.18** Organization of the control unit to allow conditional branching in the microprogram.

- A simple way to structure microinstructions is to assign one bit position to each control-signal required in the CPU.
- There are 42 signals and hence each microinstruction will have 42 bits.
- **Drawbacks of microprogrammed control:**
  1) Assigning individual bits to each control-signal results in long microinstructions because the number of required signals is usually large.
  2) Available bit-space is poorly used because only a few bits are set to 1 in any given microinstruction.
- **Solution:** Signals can be grouped because
  1) Most signals are not needed simultaneously.
  2) Many signals are mutually exclusive. E.g. only 1 function of ALU can be activated at a time. For ex: Gating signals: IN and OUT signals (Figure 7.19).
     Control-signals: Read, Write.
     ALU signals: Add, Sub, Mul, Div, Mod.
- Grouping control-signals into fields requires a little more hardware because decoding-circuits must be used to decode bit patterns of each field into individual control-signals.

- **Advantage:** This method results in a smaller control-store (only 20 bits are needed to store the patterns for the 42 signals).



Microinstruction

| F1 | F2 | F3 | F4 | F5 |
|---|---|---|---|---|

| F1 (4 bits) | F2 (3 bits) | F3 (3 bits) | F4 (4 bits) | F5 (2 bits) |
|---|---|---|---|---|
| 0000: No transfer | 000: No transfer | 000: No transfer | 0000: Add | 00: No action |
| 0001: $PC_{out}$ | 001: $PC_{in}$ | 001: $MAR_{in}$ | 0001: Sub | 01: Read |
| 0010: $MDR_{out}$ | 010: $IR_{in}$ | 010: $MDR_{in}$ | ⋮ | 10: Write |
| 0011: $Z_{out}$ | 011: $Z_{in}$ | 011: $TEMP_{in}$ |  |  |
| 0100: $R0_{out}$ | 100: $R0_{in}$ | 100: $Y_{in}$ | 1111: XOR |  |
| 0101: $R1_{out}$ | 101: $R1_{in}$ |  | 16 ALU functions |  |
| 0110: $R2_{out}$ | 110: $R2_{in}$ |  |  |  |
| 0111: $R3_{out}$ | 111: $R3_{in}$ |  |  |  |
| 1010: $TEMP_{out}$ |  |  |  |  |
| 1011: $Offset_{out}$ |  |  |  |  |

| F6 | F7 | F8 | ··· |
|---|---|---|---|

| F6 (1 bit) | F7 (1 bit) | F8 (1 bit) |
|---|---|---|
| 0: SelectY | 0: No action | 0: Continue |
| 1: Select4 | 1: WMFC | 1: End |

**Figure 7.19** An example of a partial format for field-encoded microinstructions.

## TECHNIQUES OF GROUPING OF CONTROL-SIGNALS

- The grouping of control-signal can be done either by using
  1) Vertical organization &
  2) Horizontal organisation.

| Vertical Organization | Horizontal Organization |
|---|---|
| Highly encoded schemes that use compact codes to specify only a small number of control functions in each microinstruction are referred to as a vertical organization. | The minimally encoded scheme in which many resources can be controlled with a single microinstuction is called a horizontal organization. |
| Slower operating-speeds. | Useful when higher operating-speed is desired. |
| Short formats. | Long formats. |
| Limited ability to express parallel microoperations. | Ability to express a high degree of parallelism. |
| Considerable encoding of the control information. | Little encoding of the control information. |

## MICROPROGRAM SEQUENCING

- The task of microprogram sequencing is done by microprogram sequencer.
- Two important factors must be considered while designing the microprogram sequencer:
  1) The size of the microinstruction &
  2) The address generation time.
- The size of the microinstruction should be minimum so that the size of control memory required to store microinstructions is also less.
- This reduces the cost of control memory.
- With less address generation time, microinstruction can be executed in less time resulting better throughout.
- During execution of a microprogram the address of the next microinstruction to be executed has 3 sources:
  1) Determined by instruction register.
  2) Next sequential address &

3) Branch.

- Microinstructions can be shared using microinstruction branching.
- **Disadvantage of microprogrammed branching**:
  1) Having a separate microroutine for each machine instruction results in a large total number of microinstructions and a large control-store.
  2) Execution time is longer because it takes more time to carry out the required branches.
- Consider the instruction *Add src,Rdst* ;which adds the source-operand to the contents of Rdst and places the sum in Rdst.
- Let source-operand can be specified in following addressing modes (Figure 7.20):
  a) Indexed
  b) Autoincrement
  c) Autodecrement
  d) Register indirect &
  e) Register direct
- Each box in the chart corresponds to a microinstruction that controls the transfers and operations indicated within the box.
- The microinstruction is located at the address indicated by the octal number (001,002).

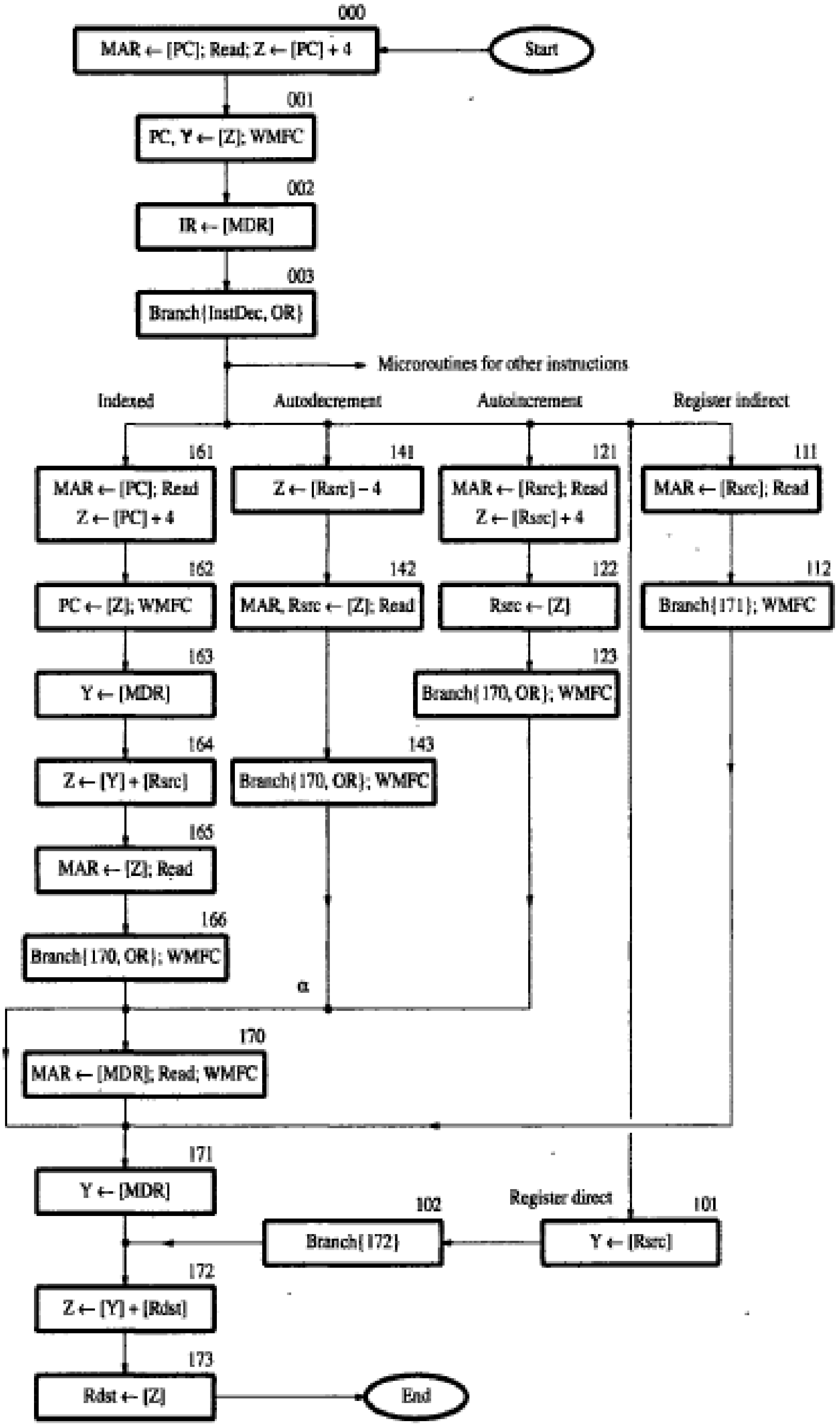


**Figure 7.20** Flowchart of a microprogram for the Add src,Rdst instruction.

**BRANCH ADDRESS MODIFICATION USING BIT-ORING**

- The branch address is determined by ORing particular bit or bits with the current address of microinstruction.
- **Eg:** If the current address is 170 and branch address is 171 then the branch address can be generated by ORing 01(bit 1), with the current address.
- Consider the point labeled $\alpha$ in the figure. At this point, it is necessary to choose between direct and indirect addressing modes.
- If indirect-mode is specified in the instruction, then the microinstruction in location 170 is performed to fetch the operand from the memory.

  If direct-mode is specified, this fetch must be bypassed by branching immediately to location 171.
- The most efficient way to bypass microinstruction 170 is to have bit-ORing of

  → current address 170 &

  → branch address 171.

**WIDE BRANCH ADDRESSING**

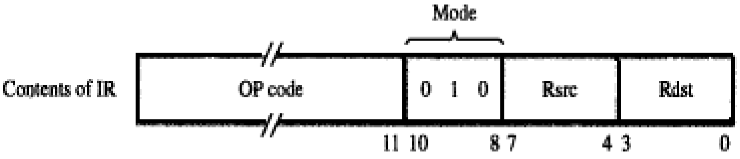
- The instruction-decoder (InstDec) generates the starting-address of the microroutine that implements the instruction that has just been loaded into the IR.
- Here, register IR contains the Add instruction, for which the instruction decoder generates the microinstruction address 101. (However, this address cannot be loaded as is into the μPC).
- The source-operand can be specified in any of several addressing-modes. The bit-ORing technique can be used to modify the starting-address generated by the instruction-decoder to reach the appropriate path.

**Use of WMFC**

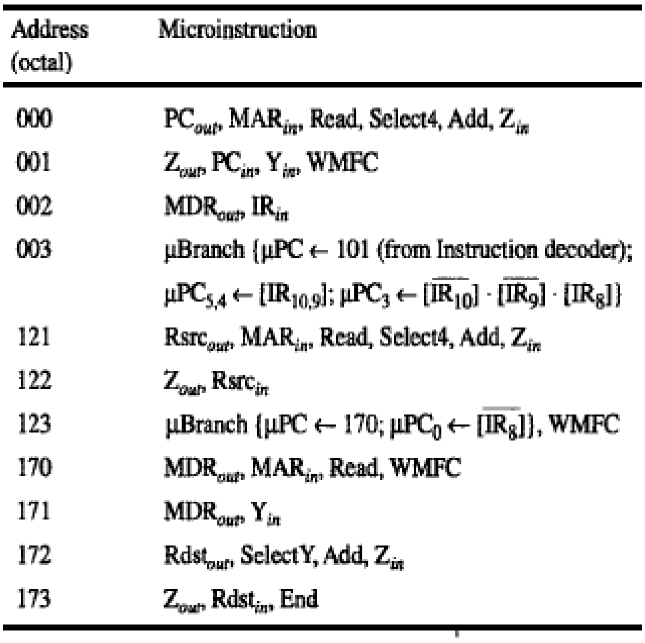
- WMFC signal is issued at location 112 which causes a branch to the microinstruction in location 171.
- WMFC signal means that the microinstruction may take several clock cycles to complete. If the branch is allowed to happen in the first clock cycle, the microinstruction at location 171 would be fetched and executed prematurely. To avoid this problem, WMFC signal must inhibit any change in the contents of the μPC during the waiting-period.

**Detailed Examination of Add (Rsrc)+,Rdst**

- Consider *Add (Rsrc)+,Rdst;* which adds Rsrc content to Rdst content, then stores the sum in Rdst and finally increments Rsrc by 4 (i.e. auto-increment mode).
- In bit 10 and 9, bit-patterns 11, 10, 01 and 00 denote indexed, auto-decrement, auto-increment and register modes respectively. For each of these modes, bit 8 is used to specify the indirect version.
- The processor has 16 registers that can be used for addressing purposes; each specified using a 4- bit-code (Figure 7.21).
- There are 2 stages of decoding:
  1) The microinstruction field must be decoded to determine that an Rsrc or Rdst register is involved.
  2) The decoded output is then used to gate the contents of the Rsrc or Rdst fields in the IR into a second decoder, which produces the gating-signals for the actual registers R0 to R15.



**MICROINSTRUCTIONS WITH NEXT-ADDRESS FIELDS**



| Address (octal) | Microinstruction |
|---|---|
| 000 | $PC_{out}$, $MAR_{in}$, Read, Select4, Add, $Z_{in}$ |
| 001 | $Z_{out}$, $PC_{in}$, $Y_{in}$, WMFC |
| 002 | $MDR_{out}$, $IR_{in}$ |
| 003 | μBranch {μPC ← 101 (from Instruction decoder); |
| | $\mu PC_{5,4} \leftarrow [IR_{10,9}]; \mu PC_3 \leftarrow [\overline{IR_{10}}] \cdot [\overline{IR_9}] \cdot [IR_8]\}$ |
| 121 | $Rsrc_{out}$, $MAR_{in}$, Read, Select4, Add, $Z_{in}$ |
| 122 | $Z_{out}$, $Rsrc_{in}$ |
| 123 | $\mu\text{Branch}\ \{\mu PC \leftarrow 170; \mu PC_0 \leftarrow [\overline{IR_8}]\}$, WMFC |
| 170 | $MDR_{out}$, $MAR_{in}$, Read, WMFC |
| 171 | $MDR_{out}$, $Y_{in}$ |
| 172 | $Rdst_{out}$, SelectY, Add, $Z_{in}$ |
| 173 | $Z_{out}$, $Rdst_{in}$, End |

**Figure 7.21** Microinstruction for Add (Rsrc)+,Rdst.

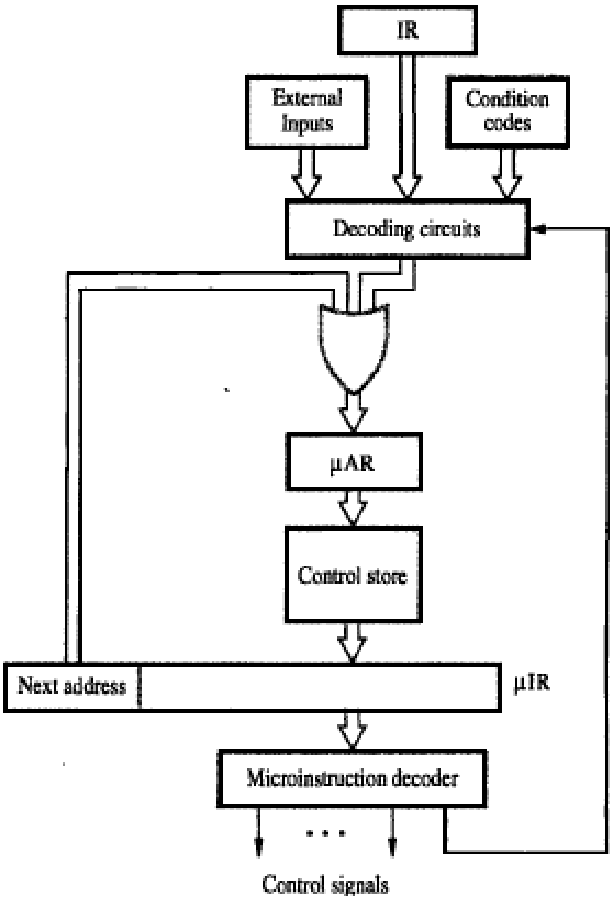


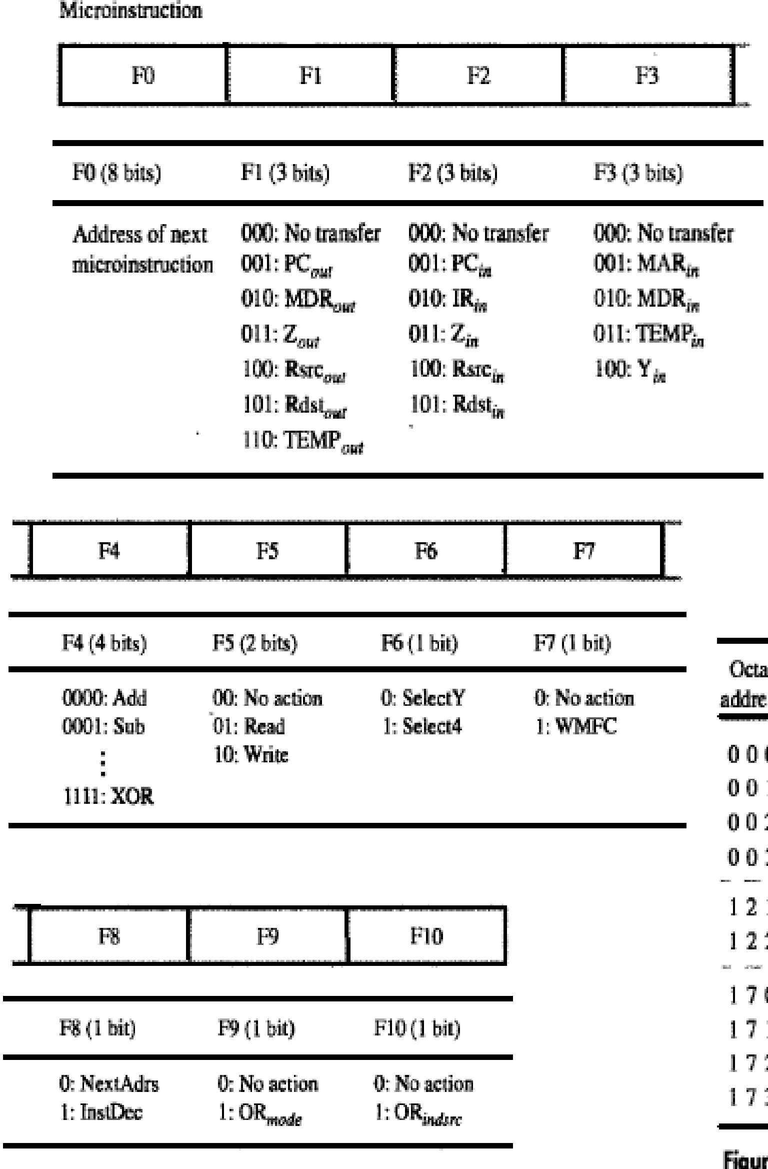
**Figure 7.22** Microinstruction-sequencing organization.

- **Drawback of previous organization:**
  - ➢ The microprogram requires several branch microinstructions which perform no useful operation. Thus, they detract from the operating-speed of the computer.

**Solution:**

- ➢ Include an address-field as a part of every microinstruction to indicate the location of the next microinstruction to be fetched. (Thus, every microinstruction becomes a branch microinstruction).

- The flexibility of this approach comes at the expense of additional bits for the address-field(Fig 7.22).
- **Advantage:** Separate branch microinstructions are virtually eliminated. (Figure 7.23-24).
- **Disadvantage:** Additional bits for the address field (around 1/6).
- There is no need for a counter to keep track of sequential address. Hence, μPC is replaced with μAR.
- The next-address bits are fed through the OR gate to the μAR, so that the address can be modified on the basis of the data in the IR, external inputs and condition-codes.
- The decoding circuits generate the starting-address of a given microroutine on the basis of the opcode in the IR. (μAR □ Microinstruction Address Register).

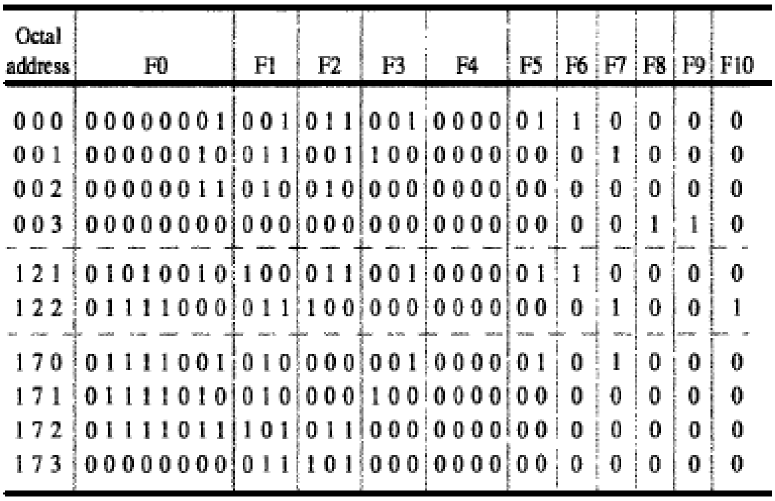

Microinstruction
F0
F1
F2
F3
F4
F5
F6
F7
F8
F9
F10

| F0 (8 bits) | F1 (3 bits) | F2 (3 bits) | F3 (3 bits) |
|---|---|---|---|
| Address of next microinstruction | 000: No transfer | 000: No transfer | 000: No transfer |
| | 001: $PC_{out}$ | 001: $PC_{in}$ | 001: $MAR_{in}$ |
| | 010: $MDR_{out}$ | 010: $IR_{in}$ | 010: $MDR_{in}$ |
| | 011: $Z_{out}$ | 011: $Z_{in}$ | 011: $TEMP_{in}$ |
| | 100: $Rsrc_{out}$ | 100: $Rsrc_{in}$ | 100: $Y_{in}$ |
| | 101: $Rdst_{out}$ | 101: $Rdst_{in}$ | |
| | 110: $TEMP_{out}$ | | |

| F4 (4 bits) | F5 (2 bits) | F6 (1 bit) | F7 (1 bit) |
|---|---|---|---|
| 0000: Add | 00: No action | 0: SelectY | 0: No action |
| 0001: Sub | 01: Read | 1: Select4 | 1: WMFC |
| ⋮ | 10: Write | | |
| 1111: XOR | | | |

| F8 (1 bit) | F9 (1 bit) | F10 (1 bit) |
|---|---|---|
| 0: NextAdrs | 0: No action | 0: No action |
| 1: InstDec | 1: $OR_{mode}$ | 1: $OR_{indsrc}$ |

**Figure 7.23** Format for microinstructions in the example of Section 7.5.3.



| Octal address | F0 | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F9 | F10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 0 0 | 0 0 0 0 0 0 0 1 | 0 0 1 | 0 1 1 | 0 0 1 | 0 0 0 0 | 0 1 | 1 | 0 | 0 | 0 | 0 |
| 0 0 1 | 0 0 0 0 0 0 1 0 | 0 1 1 | 0 0 1 | 1 0 0 | 0 0 0 0 | 0 0 | 0 | 1 | 0 | 0 | 0 |
| 0 0 2 | 0 0 0 0 0 0 1 1 | 0 1 0 | 0 1 0 | 0 0 0 | 0 0 0 0 | 0 0 | 0 | 0 | 0 | 0 | 0 |
| 0 0 3 | 0 0 0 0 0 0 0 0 | 0 0 0 | 0 0 0 | 0 0 0 | 0 0 0 0 | 0 0 | 0 | 0 | 1 | 1 | 0 |
| 1 2 1 | 0 1 0 1 0 0 1 0 | 1 0 0 | 0 1 1 | 0 0 1 | 0 0 0 0 | 0 1 | 1 | 0 | 0 | 0 | 0 |
| 1 2 2 | 0 1 1 1 1 0 0 0 | 0 1 1 | 1 0 0 | 0 0 0 | 0 0 0 0 | 0 0 | 0 | 1 | 0 | 0 | 1 |
| 1 7 0 | 0 1 1 1 1 0 0 1 | 0 1 0 | 0 0 0 | 0 0 1 | 0 0 0 0 | 0 1 | 0 | 1 | 0 | 0 | 0 |
| 1 7 1 | 0 1 1 1 1 0 1 0 | 0 1 0 | 0 0 0 | 1 0 0 | 0 0 0 0 | 0 0 | 0 | 0 | 0 | 0 | 0 |
| 1 7 2 | 0 1 1 1 1 0 1 1 | 1 0 1 | 0 1 1 | 0 0 0 | 0 0 0 0 | 0 0 | 0 | 0 | 0 | 0 | 0 |
| 1 7 3 | 0 0 0 0 0 0 0 0 | 0 1 1 | 1 0 1 | 0 0 0 | 0 0 0 0 | 0 0 | 0 | 0 | 0 | 0 | 0 |

**Figure 7.24** Implementation of the microroutine of Figure 7.21 using a next-microinstruction address field. (See Figure 7.23 for encoded signals.)

**PREFETCHING MICROINSTRUCTIONS**

- **Disadvantage of Microprogrammed Control:** Slower operating-speed because of the time it takes to fetch microinstructions from the control-store.

**Solution:** Faster operation is achieved if the next microinstruction is pre-fetched while the current one is being executed.

**Emulation**

- The main function of microprogrammed control is to provide a means for simple, flexible and relatively inexpensive execution of machine instruction.
- Its flexibility in using a machine's resources allows diverse classes of instructions to be implemented.
- Suppose we add to the instruction-repository of a given computer M1, an entirely new set of instructions that is in fact the instruction-set of a different computer M2.
- Programs written in the machine language of M2 can be then be run on computer M1 i.e. M1 emulates M2.
- Emulation allows us to replace obsolete equipment with more up-to-date machines.
- If the replacement computer fully emulates the original one, then no software changes have to be made to run existing programs.

Emulation is easiest when the machines involved have similar architectures

# UNIT IV

**NUMBERS, ARITHMETIC OPERATIONS AND CHARACTERS NUMBER REPRESENTATION**

- Numbers can be represented in 3 formats:
  1) Sign and magnitude
  2) 1's complement
  3) 2's complement
- In all three formats, MSB=0 for +ve numbers & MSB=1 for -ve numbers.
- In **sign-and-magnitude system**,

  negative value is obtained by changing the MSB from 0 to 1 of the corresponding positive value.

  For ex, +5 is represented by 0101 &

  -5 is represented by 1101.
- In **1's complement system**,

  negative values are obtained by complementing each bit of the corresponding positive number.

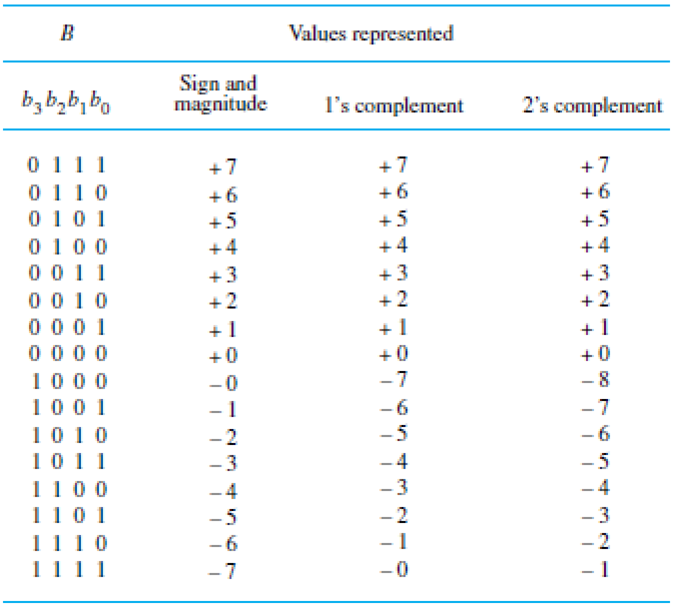
  For ex, -5 is obtained by complementing each bit in 0101 to yield 1010.

(In other words, the operation of forming the 1's complement of a given number is equivalent to subtracting that number from $2^n$-1).

- In **2's complement system**,

  forming the 2's complement of a number is done by subtracting that number from $2^n$.

  For ex, -5 is obtained by complementing each bit in 0101 & then adding 1 to yield 1011. (In other words, the 2's complement of a number is obtained by adding 1 to the 1's complement of that number).
- 2's complement system yields the most efficient way to carry out addition/subtraction operations.



| *B* | Values represented | | |
|---|---|---|---|
| $b_3 b_2 b_1 b_0$ | Sign and magnitude | 1's complement | 2's complement |
| 0 1 1 1 | +7 | +7 | +7 |
| 0 1 1 0 | +6 | +6 | +6 |
| 0 1 0 1 | +5 | +5 | +5 |
| 0 1 0 0 | +4 | +4 | +4 |
| 0 0 1 1 | +3 | +3 | +3 |
| 0 0 1 0 | +2 | +2 | +2 |
| 0 0 0 1 | +1 | +1 | +1 |
| 0 0 0 0 | +0 | +0 | +0 |
| 1 0 0 0 | −0 | −7 | −8 |
| 1 0 0 1 | −1 | −6 | −7 |
| 1 0 1 0 | −2 | −5 | −6 |
| 1 0 1 1 | −3 | −4 | −5 |
| 1 1 0 0 | −4 | −3 | −4 |
| 1 1 0 1 | −5 | −2 | −3 |
| 1 1 1 0 | −6 | −1 | −2 |
| 1 1 1 1 | −7 | −0 | −1 |

**Figure 1.3** Binary, signed-integer representations.

**ADDITION OF POSITIVE NUMBERS**

- Consider adding two 1-bit numbers.
- The sum of 1 & 1 requires the 2-bit vector 10 to represent the value 2. We say that sum is 0 and the carry-out is 1.

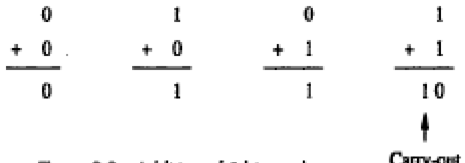


**Figure 2.2** Addition of 1-bit numbers.

**ADDITION & SUBTRACTION OF SIGNED NUMBERS**

- Following are the two rules for addition and subtraction of n-bit signed numbers using the 2's complement representation system (Figure 1.6).

**Rule 1:**

- ➢ **To Add** two numbers, add their n-bits and ignore the carry-out signal from the MSB position.
- ➢ Result will be algebraically correct, if it lies in the range $-2^{n-1}$ to $+2^{n-1}-1$.

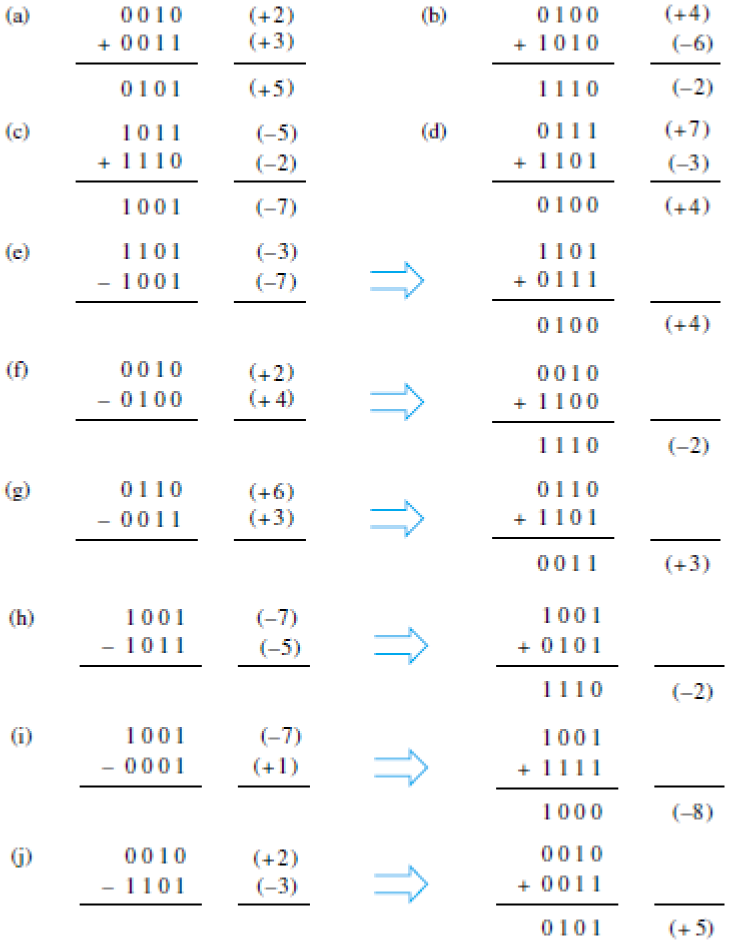
**Rule 2:**

- ➢ **To Subtract** two numbers X and Y (that is to perform X-Y), take the 2's complement of Y and then add it to X as in rule 1.
- ➢ Result will be algebraically correct, if it lies in the range $(2^{n-1})$ to $+(2^{n-1}-1)$.

- When the result of an arithmetic operation is outside the representable-range, an arithmetic overflow is said to occur.
- To represent a signed in 2's complement form using a larger number of bits, repeat the sign bit as many times as needed to the left. This operation is called **sign extension**.
- In 1's complement representation, the result obtained after an addition operation is not always correct. The carry-out($c_n$) cannot be ignored. If $c_n$=0, the result obtained is correct. If $c_n$=1, then a 1 must be added to the result to make it correct.

## OVERFLOW IN INTEGER ARITHMETIC

- When result of an arithmetic operation is outside the representable-range, an **arithmetic overflow** is said to occur.
- For example: If we add two numbers +7 and +4, then the output sum S is 1011(□0111+0100), which is the code for -5, an incorrect result.
- An overflow occurs in following 2 cases
  1) Overflow can occur only when adding two numbers that have the same sign.
  2) The carry-out signal from the sign-bit position is not a sufficient indicator of overflow when adding signed numbers.



```
(a)    0010    (+2)          (b)    0100    (+4)
     + 0011    (+3)               + 1010    (-6)
     ------    ----               ------    ----
       0101    (+5)                 1110    (-2)

(c)    1011    (-5)          (d)    0111    (+7)
     + 1110    (-2)               + 1101    (-3)
     ------    ----               ------    ----
       1001    (-7)                 0100    (+4)

(e)    1101    (-3)                 1101
     - 1001    (-7)     ⇒         + 0111
     ------    ----               ------    ----
                                    0100    (+4)

(f)    0010    (+2)                 0010
     - 0100    (+4)     ⇒         + 1100
     ------    ----               ------    ----
                                    1110    (-2)

(g)    0110    (+6)                 0110
     - 0011    (+3)     ⇒         + 1101
     ------    ----               ------    ----
                                    0011    (+3)

(h)    1001    (-7)                 1001
     - 1011    (-5)     ⇒         + 0101
     ------    ----               ------    ----
                                    1110    (-2)

(i)    1001    (-7)                 1001
     - 0001    (+1)     ⇒         + 1111
     ------    ----               ------    ----
                                    1000    (-8)

(j)    0010    (+2)                 0010
     - 1101    (-3)     ⇒         + 0011
     ------    ----               ------    ----
                                    0101    (+5)
```

**Figure 1.6** 2's-complement Add and Subtract operations.

## ADDITION & SUBTRACTION OF SIGNED NUMBERS n-BIT RIPPLE CARRY ADDER

- A cascaded connection of n full-adder blocks can be used to add 2-bit numbers.
- Since carries must propagate (or ripple) through cascade, the configuration is called an n-bit ripple carry adder

(Figure 9.1).

| $x_i$ | $y_i$ | Carry-in $c_i$ | Sum $s_i$ | Carry-out $c_{i+1}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

$$s_i = \overline{x}_i\overline{y}_i c_i + \overline{x}_i y_i \overline{c}_i + x_i \overline{y}_i \overline{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$

Example:

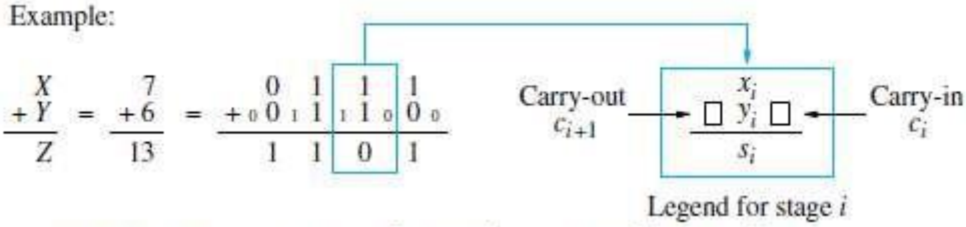
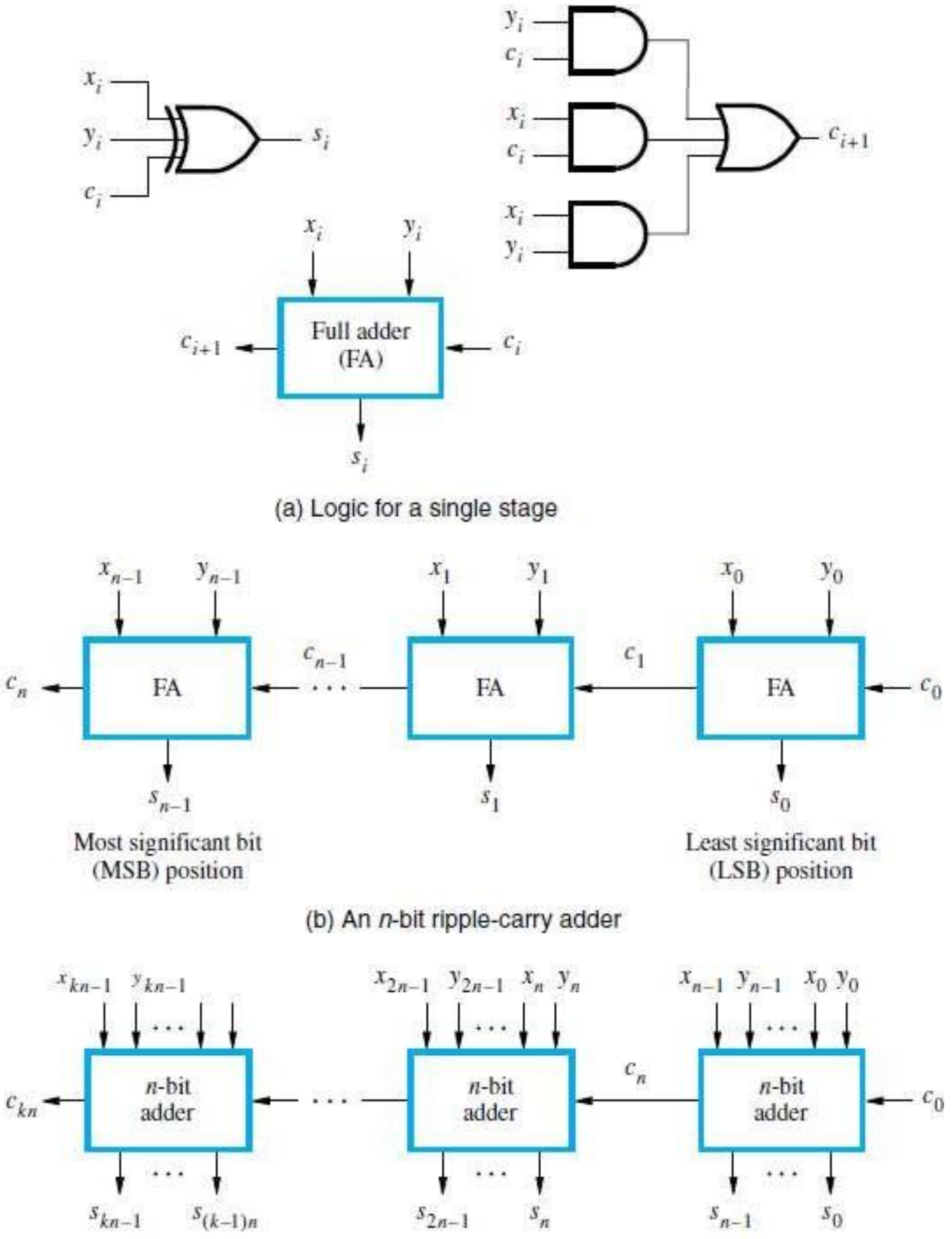


**Figure 9.1** Logic specification for a stage of binary addition.



(a) Logic for a single stage

(b) An *n*-bit ripple-carry adder

(c) Cascade of *k* *n*-bit adders

**Figure 9.2** Logic for addition of binary numbers.

**ADDITION/SUBTRACTION LOGIC UNIT**

- The n-bit adder can be used to add 2's complement numbers X and Y (Figure 9.3).
- **Overflow** can only occur when the signs of the 2 operands are the same.
- In order to perform the subtraction operation X-Y on 2's complement numbers X and Y; we form the 2's complement of Y and add it to X.
- Addition or subtraction operation is done based on value applied to the Add/Sub input control-line.
- Control-line=0 for addition, applying the Y vector unchanged to one of the adder inputs.

Control-line=1 for subtraction, the Y vector is 2's complemented.

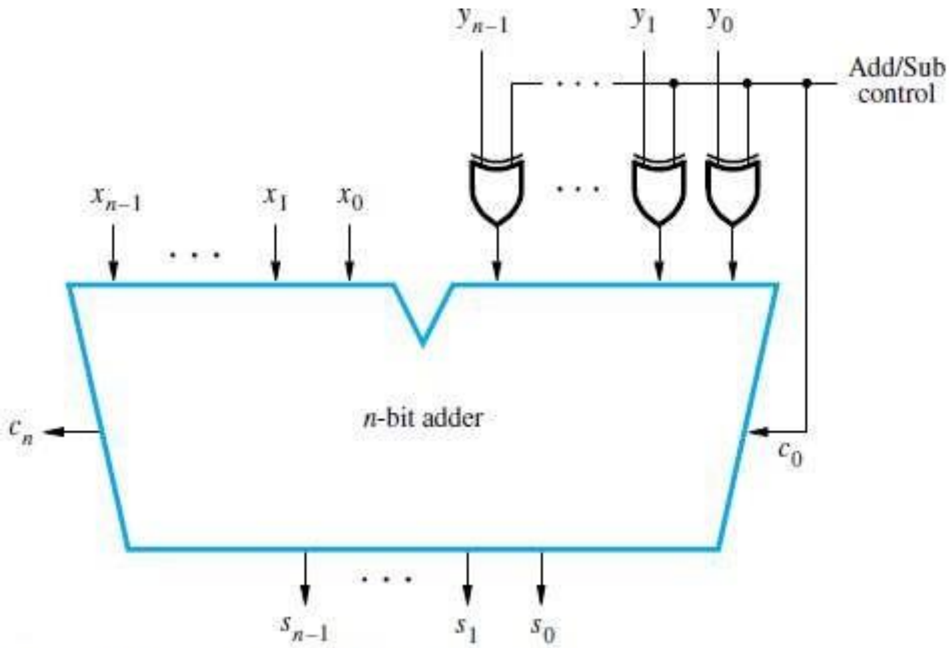


**Figure 9.3** Binary addition/subtraction logic circuit.

**DESIGN OF FAST ADDERS**

- **Drawback of ripple carry adder:** If the adder is used to implement the addition/subtraction, all sum bits are available in 2n gate delays.
- Two approaches can be used to reduce delay in adders:

1) Use the fastest possible electronic-technology in implementing the ripple-carry design.

Use an augmented logic-gate network structure

**CARRY-LOOKAHEAD ADDITIONS**

- The logic expression for $s_i$(sum) and $c_{i+1}$(carry-out) of stage i are

$s_i=x_i+y_i+c_i$ ------(1) $c_{i+1}=x_iy_i+x_ic_i+y_ic_i$ ----------------- (2)

- Factoring (2) into

$c_{i+1}=$ $x_iy_i+(x_i+y_i)c_i$ we can write

$c_{i+1}=G_i+P_iC_i$ where $G_i=x_iy_i$ and $P_i=x_i+y_i$

- The expressions $G_i$ and $P_i$ are called generate and propagate functions (Figure 9.4).
- If $G_i=1$, then $c_{i+1}=1$, independent of the input carry $c_i$. This occurs when both $x_i$ and $y_i$ are 1. Propagate function means that an input-carry will produce an output-carry when either $x_i=1$ or $y_i=1$.
- All $G_i$ and $P_i$ functions can be formed independently and in parallel in one logic-gate delay.
- Expanding $c_i$ terms of i-1 subscripted variables and substituting into the $c_{i+1}$ expression, we obtain

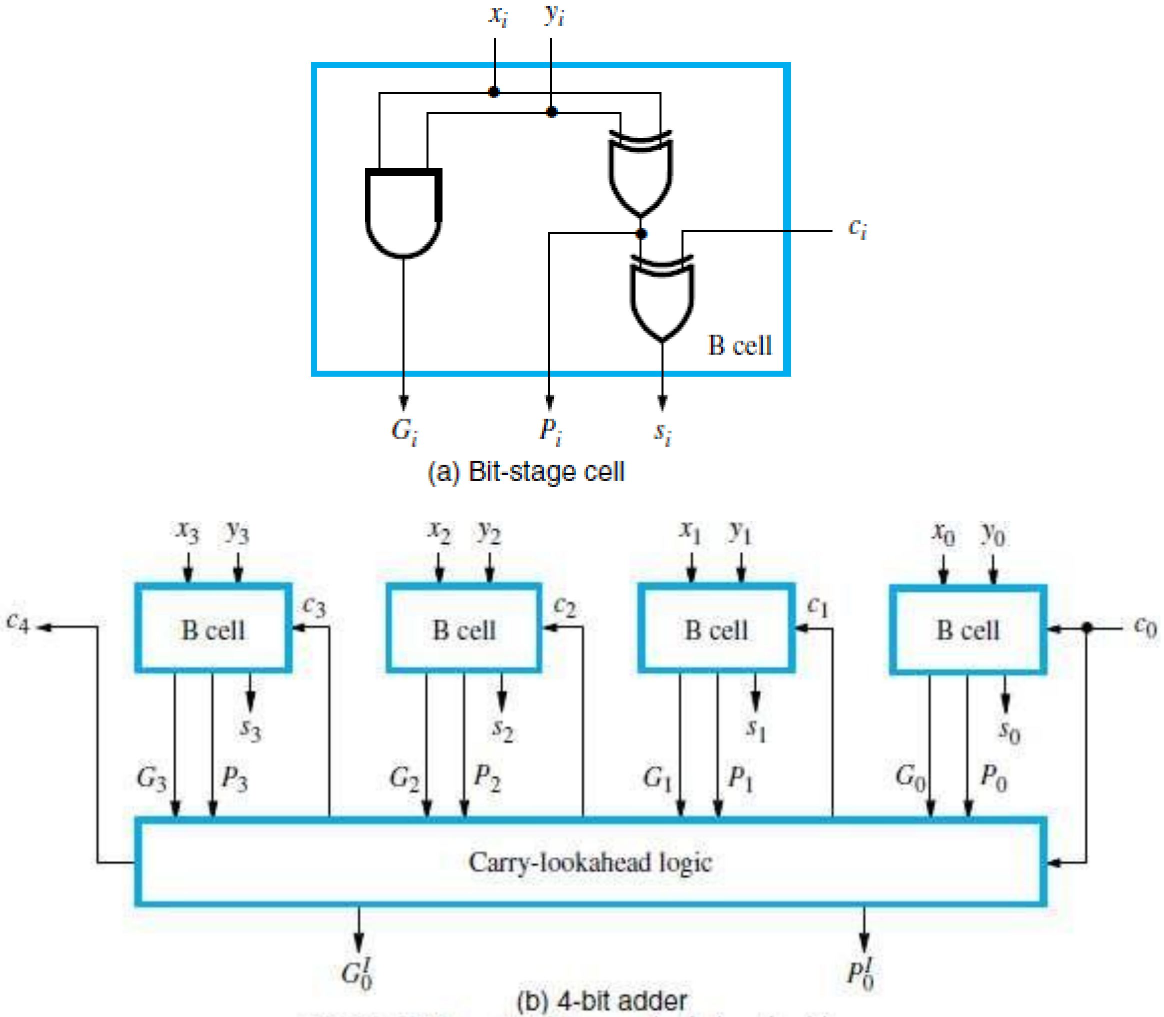
$c_{i+1}=G_i+P_iG_{i-1}+P_iP_{i-1}G_{i-2.}$ . . . . .$+P_1G_0+P_iP_{i-1}$ ........................ $P_0c_0$

- Conclusion: Delay through the adder is 3 gate delays for all carry-bits & 4 gate delays for all sum-bits.
- Consider the design of a 4-bit adder. The carries can be implemented as

$c_1=G_0+P_0c_0$
$c_2=G_1+P_1G_0+P_1P_0c_0$
$c_3=G_2+P_2G_1+P_2P_1G_0+P_2P_1P_0c_0$
$c_4=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0+P_3P_2P_1P_0$
$c_0$

- The carries are implemented in the block labeled carry-lookahead logic. An adder implemented in this form is called a **Carry-Lookahead Adder**.

- Limitation: If we try to extend the carry-lookahead adder for longer operands, we run into a problem of gate fan-in constraints.



(a) Bit-stage cell

(b) 4-bit adder

**Figure 9.4** A 4-bit carry-lookahead adder.

**HIGHER-LEVEL GENERATE & PROPAGATE FUNCTIONS**

- 16-bit adder can be built from four 4-bit adder blocks (Figure 9.5).
- These blocks provide new output functions defined as $G_k$ and $P_k$, where k=0 for the first 4-bit block, k=1 for the second 4-bit block and so on.
- In the first block,

  $P_0=P_3P_2P_1P_0$

  &

  $G_0=G_3+P_3G_2+P_3P_2G_1+P_3$
  $P_2P_1G_0$

- The first-level $G_i$ and $P_i$ functions determine whether bit stage i generates or propagates a carry, and the second level $G_k$ and $P_k$ functions determine whether block k generates or propagates a carry.
- Carry $c_{16}$ is formed by one of the carry-lookahead circuits as $c_{16}=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0+P_3P_2P_1P_0c_0$
- Conclusion: All carries are available 5 gate delays after X, Y and $c_0$ are applied as inputs.

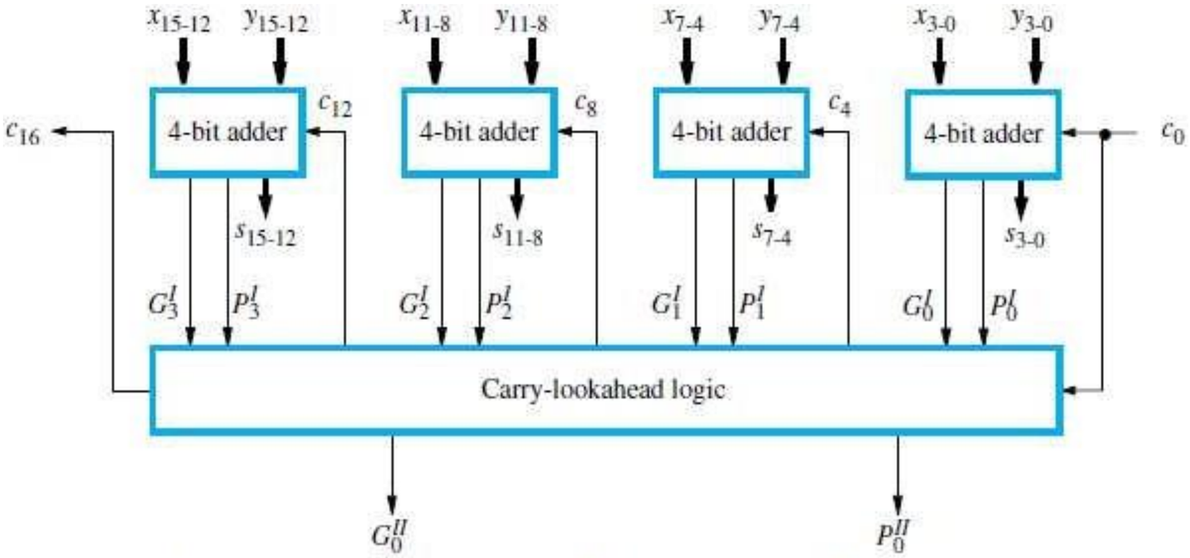
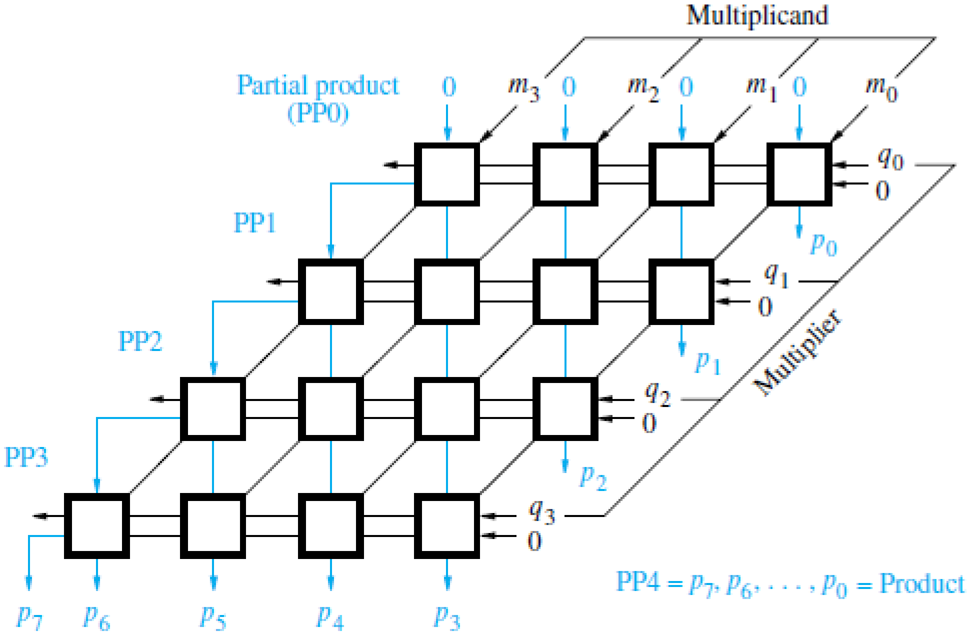


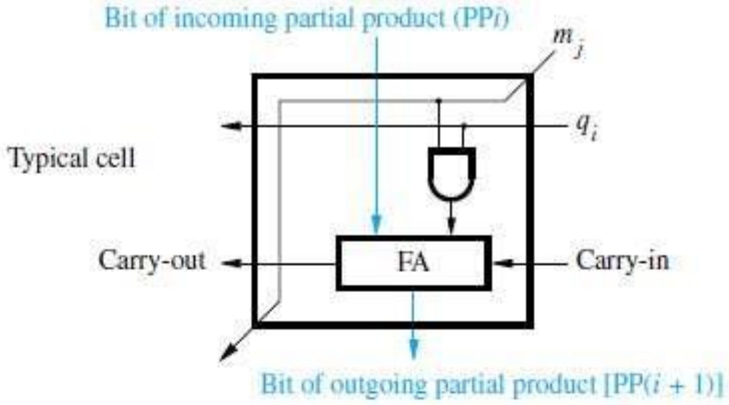
**Figure 9.5** A 16-bit carry-lookahead adder built from 4-bit adders (see Figure 9.4*b*).

## MULTIPLICATION OF POSITIVE NUMBERS

```
                1  1  0  1      (13)  Multiplicand M
             ×  1  0  1  1      (11)  Multiplier Q
             ------------
                1  1  0  1
             1  1  0  1
          0  0  0  0
       1  1  0  1
    ---------------------
    1  0  0  0  1  1  1  1      (143)  Product P
```

(a) Manual multiplication algorithm




Bit of incoming partial product (PPi)
$m_j$
$q_i$
Typical cell
Carry-out
FA
Carry-in
Bit of outgoing partial product [PP(i + 1)]

(b) Array implementation

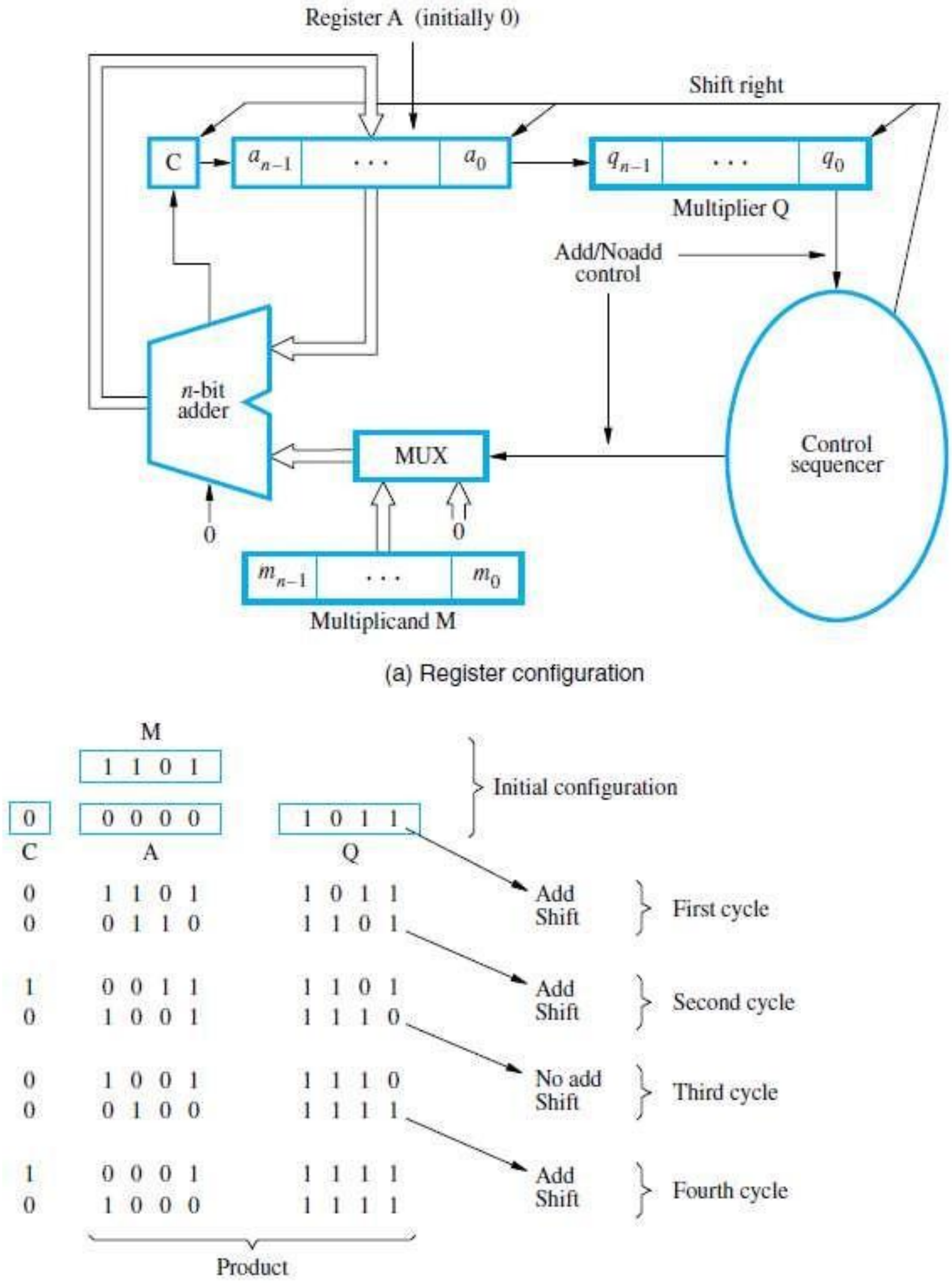
**Figure 9.6** Array multiplication of unsigned binary operands.

## ARRAY MULTIPLICATION

- The main component in each cell is a full adder(FA)..
- The AND gate in each cell determines whether a multiplicand bit $m_j$, is added to the incoming partial- product bit, based on the value of the multiplier bit $q_i$ (Figure9.6).

## SEQUENTIAL CIRCUIT BINARY MULTIPLIER

- Registers A and Q combined hold $PP_i$(partial product) while the multiplier bit $q_i$ generates the signal Add/Noadd.
- The carry-out from the adder is stored in flip-flop C (Figure 9.7).
- Procedure for multiplication:
  1) Multiplier is loaded into register Q, Multiplicand is loaded into register M and C & A are cleared to 0.
  2) If $q_0$=1, add M to A and store sum in A. Then C, A and Q are shifted right one bit-position. If $q_0$=0, no addition performed and C, A & Q are shifted right one bit-position.
  3) After n cycles, the high-order half of the product is held in register A and the low-order half is held in register Q.



(a) Register configuration

M: 1 1 0 1

| C | A | Q | | |
|---|---|---|---|---|
| 0 | 0 0 0 0 | 1 0 1 1 | | Initial configuration |
| 0 | 1 1 0 1 | 1 0 1 1 | Add | First cycle |
| 0 | 0 1 1 0 | 1 1 0 1 | Shift | |
| 1 | 0 0 1 1 | 1 1 0 1 | Add | Second cycle |
| 0 | 1 0 0 1 | 1 1 1 0 | Shift | |
| 0 | 1 0 0 1 | 1 1 1 0 | No add | Third cycle |
| 0 | 0 1 0 0 | 1 1 1 1 | Shift | |
| 1 | 0 0 0 1 | 1 1 1 1 | Add | Fourth cycle |
| 0 | 1 0 0 0 | 1 1 1 1 | Shift | |

(b) Multiplication example

**Figure 9.7** Sequential circuit binary multiplier.

**SIGNED OPERAND MULTIPLICATION BOOTH ALGORITHM**

- This algorithm

→ generates a 2n-bit product

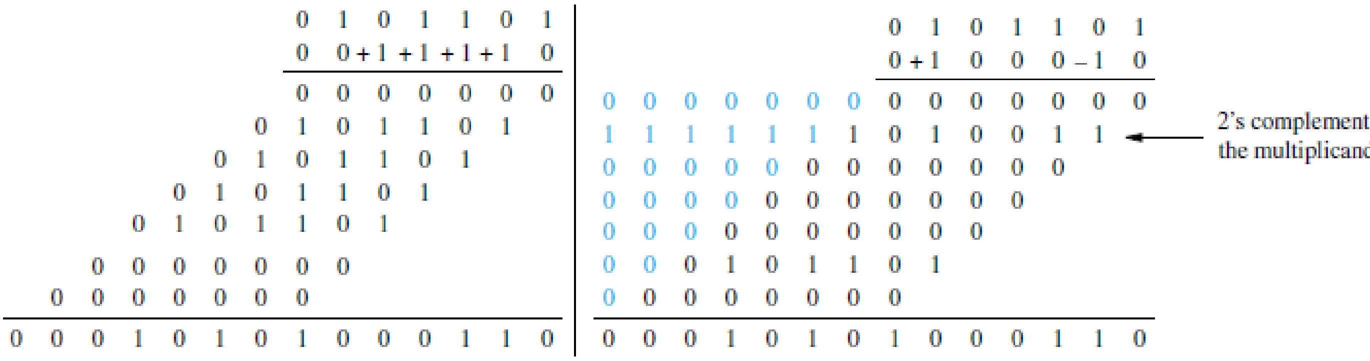
→ treats both positive & negative 2's-complement n-bit operands uniformly(Figure 9.9-9.12).

- Attractive feature: This algorithm achieves some efficiency in the number of addition required when the multiplier has a few large blocks of 1s.
- This algorithm suggests that we can reduce the number of operations required for multiplication by representing multiplier as a difference between 2 numbers.

For e.g. multiplier(Q) 14(001110) can be represented as

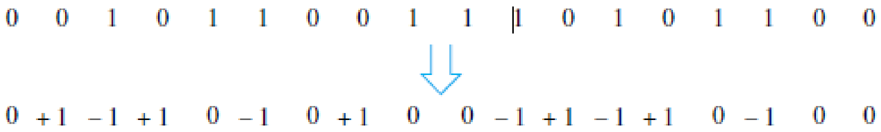
010000 (16)
<u>-000010 (2)</u>
001110 (14)

- Therefore, product P=M*Q can be computed by adding $2^4$ times the M to the 2's complement of $2^1$ times the M.



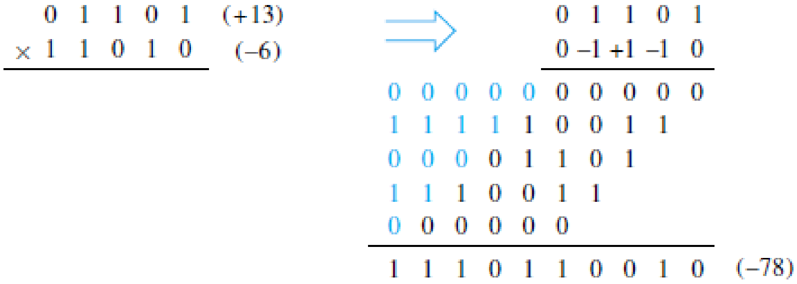
```
              0  1  0  1  1  0  1    |                        0  1  0  1  1  0  1
              0  0 +1 +1 +1 +1  0    |                        0 +1  0  0  0 -1  0
              0  0  0  0  0  0  0    |  0  0  0  0  0  0  0  0  0  0  0  0  0  0
           0  1  0  1  1  0  1       |  1  1  1  1  1  1  1  0  1  0  0  1  1     <---- 2's complement
        0  1  0  1  1  0  1          |  0  0  0  0  0  0  0  0  0  0  0  0              the multiplican
     0  1  0  1  1  0  1             |  0  0  0  0  0  0  0  0  0  0  0
  0  1  0  1  1  0  1                |  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0                |  0  0  0  1  0  1  1  0  1
0  0  0  0  0  0  0                  |  0  0  0  0  0  0  0  0
0 0 0 1 0 1 0 1 0 0 0 1 1 0          |  0  0  0  1  0  1  0  1  0  0  0  1  1  0
```

**Figure 9.9** Normal and Booth multiplication schemes.



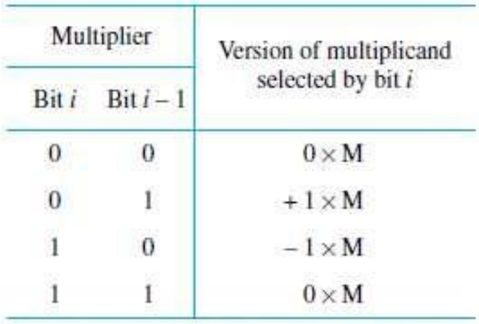
```
0   0   1   0   1   1   0   0   1   1  |1   0   1   0   1   1   0   0
                             ⇓
0  +1  -1  +1   0  -1   0  +1   0   0  -1  +1  -1  +1   0  -1   0   0
```

**Figure 9.10** Booth recoding of a multiplier.



```
  0 1 1 0 1   (+13)                   0  1  1  0  1
× 1 1 0 1 0   (−6)     ⟹              0 −1 +1 −1  0
                        0  0  0  0  0  0  0  0  0  0
                        1  1  1  1  1  0  0  1  1
                        0  0  0  0  1  1  0  1
                        1  1  1  0  0  1  1
                        0  0  0  0  0  0
                        1  1  1  0  1  1  0  0  1  0   (−78)
```

**Figure 9.11** Booth multiplication with a negative multiplier.

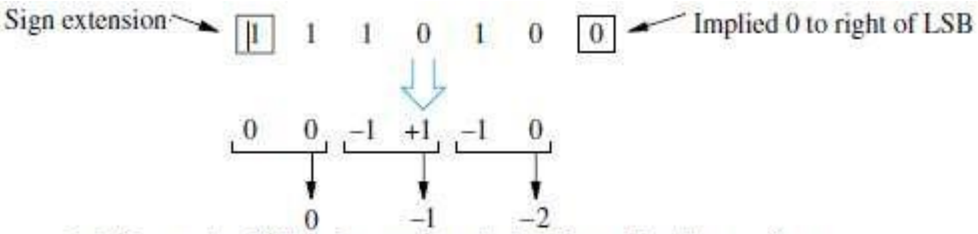


| Multiplier | | Version of multiplicand selected by bit *i* |
|---|---|---|
| Bit *i* | Bit *i* – 1 | |
| 0 | 0 | 0 × M |
| 0 | 1 | + 1 × M |
| 1 | 0 | – 1 × M |
| 1 | 1 | 0 × M |

**Figure 9.12** Booth multiplier recoding table.

## FAST MULTIPLICATION
## BIT-PAIR RECODING OF MULTIPLIERS

- This method
  → derived from the booth algorithm
  → reduces the number of summands by a factor of 2
- Group the Booth-recoded multiplier bits in pairs. (Figure 9.14 & 9.15).
- The pair (+1 -1) is equivalent to the pair (0 +1).



(a) Example of bit-pair recoding derived from Booth recoding

| Multiplier bit-pair | | Multiplier bit on the right | Multiplicand selected at position *i* |
|---|---|---|---|
| *i* + 1 | *i* | *i* − 1 | |
| 0 | 0 | 0 | 0 × M |
| 0 | 0 | 1 | + 1 × M |
| 0 | 1 | 0 | + 1 × M |
| 0 | 1 | 1 | + 2 × M |
| 1 | 0 | 0 | − 2 × M |
| 1 | 0 | 1 | − 1 × M |
| 1 | 1 | 0 | − 1 × M |
| 1 | 1 | 1 | 0 × M |

(b) Table of multiplicand selection decisions

**Figure 9.14** Multiplier bit-pair recoding.

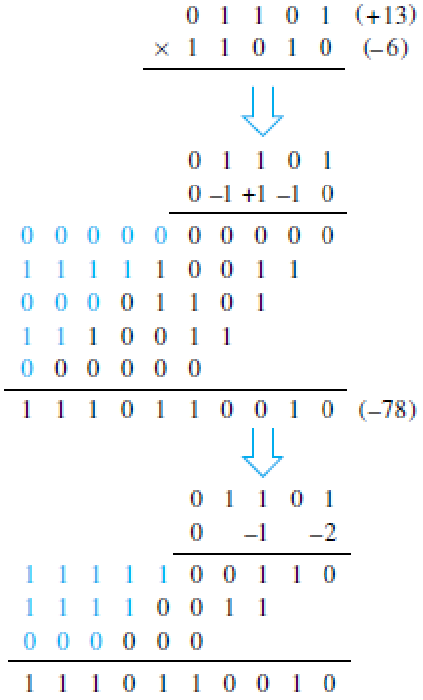


**Figure 9.15** Multiplication requiring only *n*/2 summands.

Prepared By:

Roshni S. Khedgaonkar