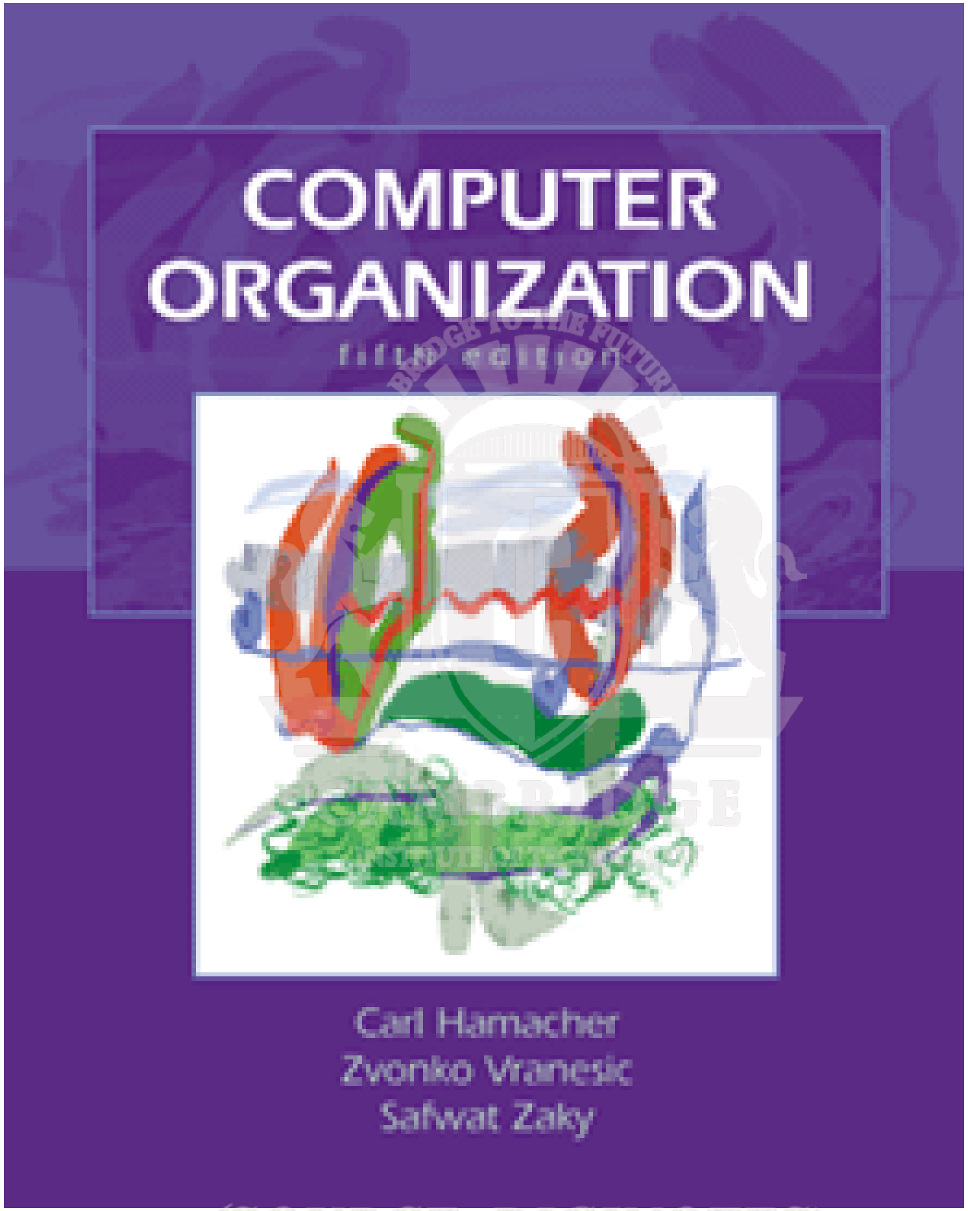
Subject code
15CS34



COMPUTER ORGANIZATION

fifth edition

Carl Hamacher
Zvonko Vranesic
Safwat Zaky

Sowmya K P
Asst. Professor
Dept of ISE
CITech

MODULE 1: Basic structure of computer & Machine instructions and programs

Courtesy: Text book: Carl Hamacher 5th Edition

Functional Units

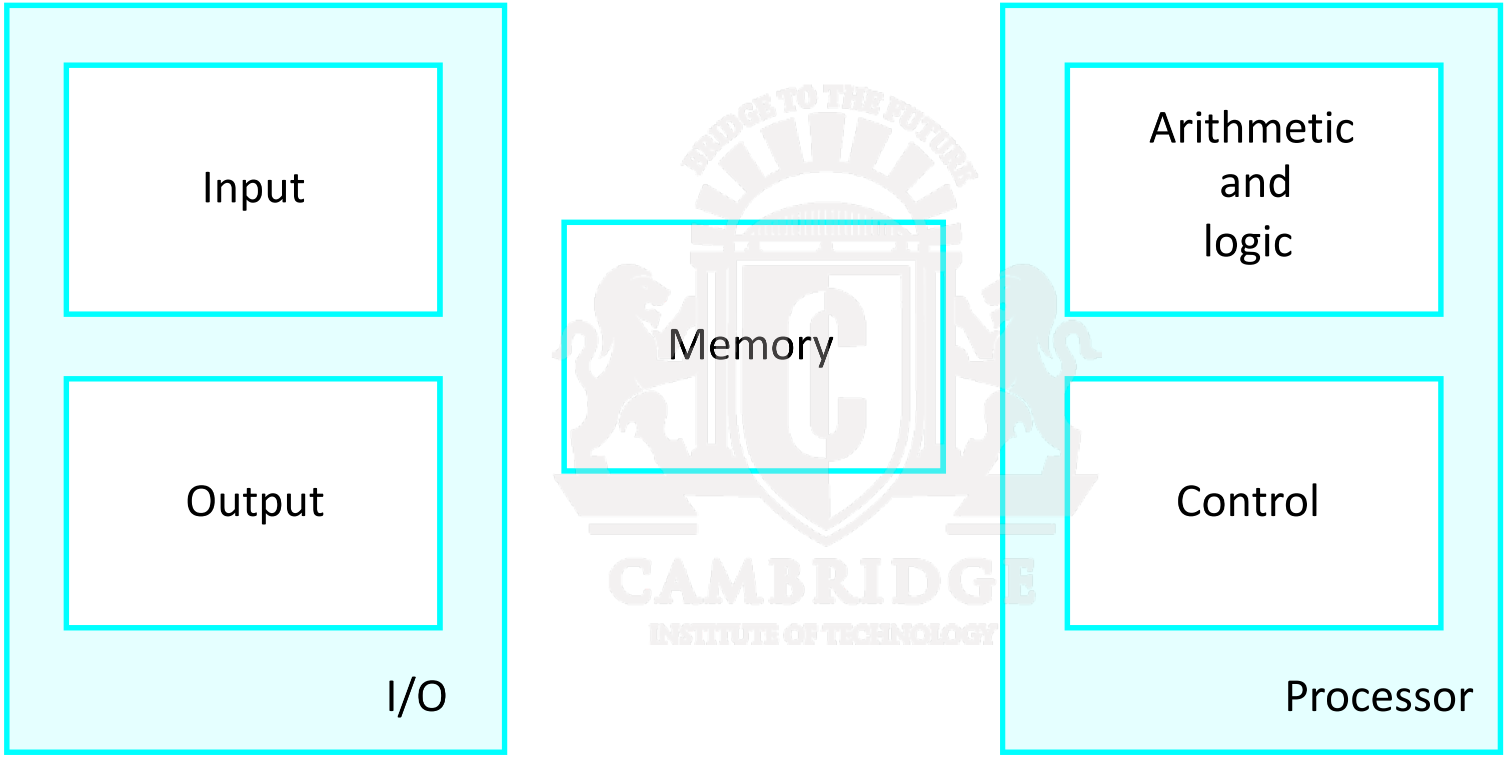


Figure 1.1. Basic functional units of a computer.

Basic operational concepts : A Typical Instruction

- Add LOCA, R0
- Add the operand at memory location LOCA to the operand in a register R0 in the processor.
- Place the sum into register R0.
- The original contents of LOCA are preserved.
- The original contents of R0 is overwritten.
- Instruction is fetched from the memory into the processor – the operand at LOCA is fetched and added to the contents of R0 – the resulting sum is stored in register R0.

Separate Memory Access and ALU Operation

- Load LOCA, R1
- Add R1, R0
- Whose contents will be overwritten?

Connection Between the Processor and the Memory

Registers

- **The instruction register (IR):- Holds the instructions that is currently being executed.**

- Its output is available for the control circuits which generates the timing signals that control the various processing elements in one execution of instruction.

- **The program counter PC:-**
- This is another specialized register that keeps track of execution of a program. It contains the memory address of the next instruction to be fetched and executed

- Besides IR and PC, there are n-general purpose registers R0 through Rn-1.

Registers

- The other two registers which facilitate communication with memory are: -
- **1. MAR – (Memory Address Register)**:- It holds the address of the location to be accessed.
- **2. MDR – (Memory Data Register):-** It contains the data to be written into or read out of the address location.

Typical Operating Steps

Programs reside in the memory through input devices

PC is set to point to the first instruction

The contents of PC are transferred to MAR

A Read signal is sent to the memory

The first instruction is read out and loaded into MDR

The contents of MDR are transferred to IR

Decode and execute the instruction

If the instruction involves an operation by the ALU, it is necessary to obtain the required operands.

Typical Operating Steps (Cont')

An operand in the memory is fetched by sending its address to MAR & Initiating a read cycle.

When the operand has been read from the memory to the MDR, it is transferred from MDR to the ALU

After one or two such repeated cycles, the ALU can perform the desired operation.

If the result of this operation is to be stored in the memory, the result is sent to MDR.

Address of location where the result is stored is sent to MAR & a write cycle is initiated.

The contents of PC are incremented so that PC points to the next instruction that is to be executed.

Interrupt

- Normal execution of programs may be preempted if some device requires urgent servicing.
- An interrupt is a request signal from an I/O device for service by the processor.
- The processor provides the requested service by executing an appropriate interrupt service routine

Bus Structures

- There are many ways to connect different parts inside a computer together.
- A group of lines that serves as a connecting path for several devices is called a *bus*.
- Address/data/control
- Since the bus can be used for only one transfer at a time, only two units can actively use the bus at any given time.
- Bus control lines are used to arbitrate multiple requests for use of one bus.
- Single bus structure is
- Low cost
- Very flexible for attaching peripheral devices
- Multiple bus structure certainly increases, the performance but also increases the cost significantly.

Bus Structure

- Single-bus

Performance

- The most important measure of a computer is how quickly it can execute programs.
- Three factors affect performance:
- ***Hardware design***
- ***Instruction set***
- ***Compiler***
- The total time required to execute the program **is elapsed time** is a measure of the performance of the entire computer system. It is affected by the speed of the processor, the disk and the printer.
- The time needed to execute a instruction is called the processor time.
- Processor time to execute a program depends on the hardware involved in the execution of individual machine instructions

Performance

- ***The figure which includes the cache memory as part of the processor unit***

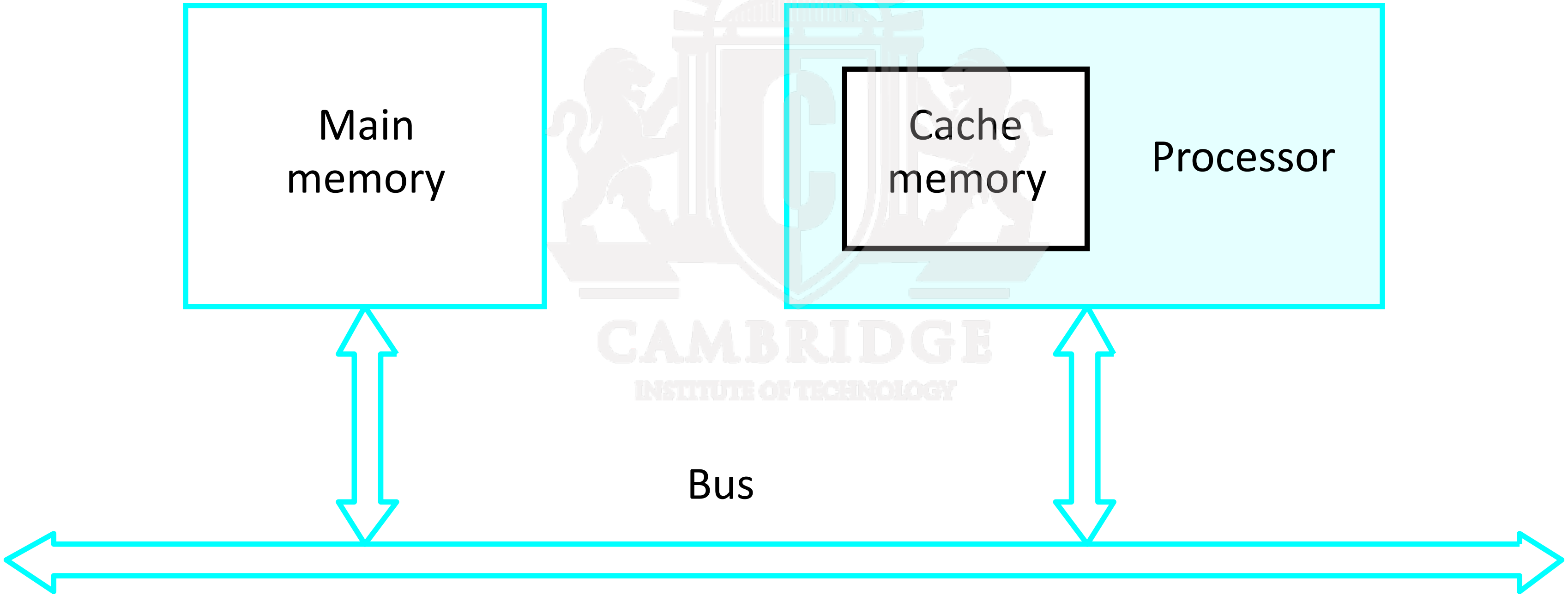


Figure 1.5. The processor cache.

Performance

- The processor and a relatively small cache memory can be fabricated on a single integrated circuit chip.
- Speed
- Cost
- Memory management

Performance

- Let us examine the flow of program instructions and data between the memory and the processor.
- At the start of execution, all **program instructions** and the **required data** are stored in the main memory.
- As the execution proceeds, instructions are fetched one by one over the bus into the processor, and **a copy is placed in the cache** later if the same instruction or data item is needed a second time, it is read directly from the cache

Processor Clock

- Processor circuits are controlled by a timing signal called ***clock***
- The clock defines the regular time intervals called **clock cycles**.
- To execute a machine instruction the processor divides the action to be performed into a sequence of basic steps that each step can be completed in one clock cycle.
- The length P of one clock cycle is an important parameter that affects the processor performance.
- Processor used in today's personal computer and work station have a clock rates that range from a few hundred million to over a billion cycles per second

Processor Clock

- The length P of one clock cycle is an important parameter that affects processor performance.
- Its inverse is the clock rate R=1/P.

- The term "cycles per second" is called *hertz*

- *The term "million" is denoted by the prefix Mega(M)*
- *and "billion" is denoted by the prefix Giga(G)*

Basic Performance Equation

- T – processor time required to execute a program that has been prepared in high-level language
- N – number of actual machine language instructions needed to complete the execution (note: loop)
- S – average number of basic steps needed to execute one machine instruction. Each step completes in one clock cycle
- R – clock rate
- Note: these are not independent to each other

$$T = \frac{N \times S}{R}$$

How to improve T?

Clock Rate

- These are two possibilities for increasing the clock rate 'R'.

  1. Improving the IC technology makes logical circuit faster, which reduces the time of execution of basic steps. This allows the clock period P, to be reduced and the clock rate R to be increased.

  2. Reducing the amount of processing done in one basic step also makes it possible to reduce the clock period P. however if the actions that have to be performed by an instructions remain the same, the number of basic steps needed may increase.

Performance Measurement

- The performance measure is the time taken by the computer to execute a given bench mark. Initially some attempts were made to create artificial programs that could be used as bench mark programs

- A non profit organization called SPEC- system performance evaluation corporation selects and publishes bench marks
- The program selected range from game playing, compiler, and data base applications to numerically intensive programs in astrophysics and quantum chemistry. In each case, the program is compiled under test, and the running time on a real computer is measured.
- The same program is also compiled and run on one computer selected as reference.
- The 'SPEC' rating is computed as follows.

Performance Measurement

$$SPEC\ rating = \frac{\text{Running time on the reference computer}}{\text{Running time on the computer under test}}$$

$$SPEC\ rating = \left(\prod_{i=1}^{n} SPEC_i\right)^{\frac{1}{n}}$$

If the SPEC rating = 50 Means that the computer under test is 50 times as fast as the ultra sparc 10.

This is repeated for all the programs in the SPEC suit, and the geometric mean of the result is computed.

Machine instructions and programs : Objectives

- Machine instructions and program execution, including branching and subroutine call and return operations.
- Number representation and addition/subtraction in the 2’s-complement system.
- Addressing methods for accessing register and memory operands.
- Assembly language for representing machine instructions, data, and programs.
- Program-controlled Input/Output operations.

Memory Location, Addresses, and Operation

- Memory consists of many millions of storage cells, each of which can store 1 bit.
- Data is usually accessed in *n*-bit groups. *n* is called word length.

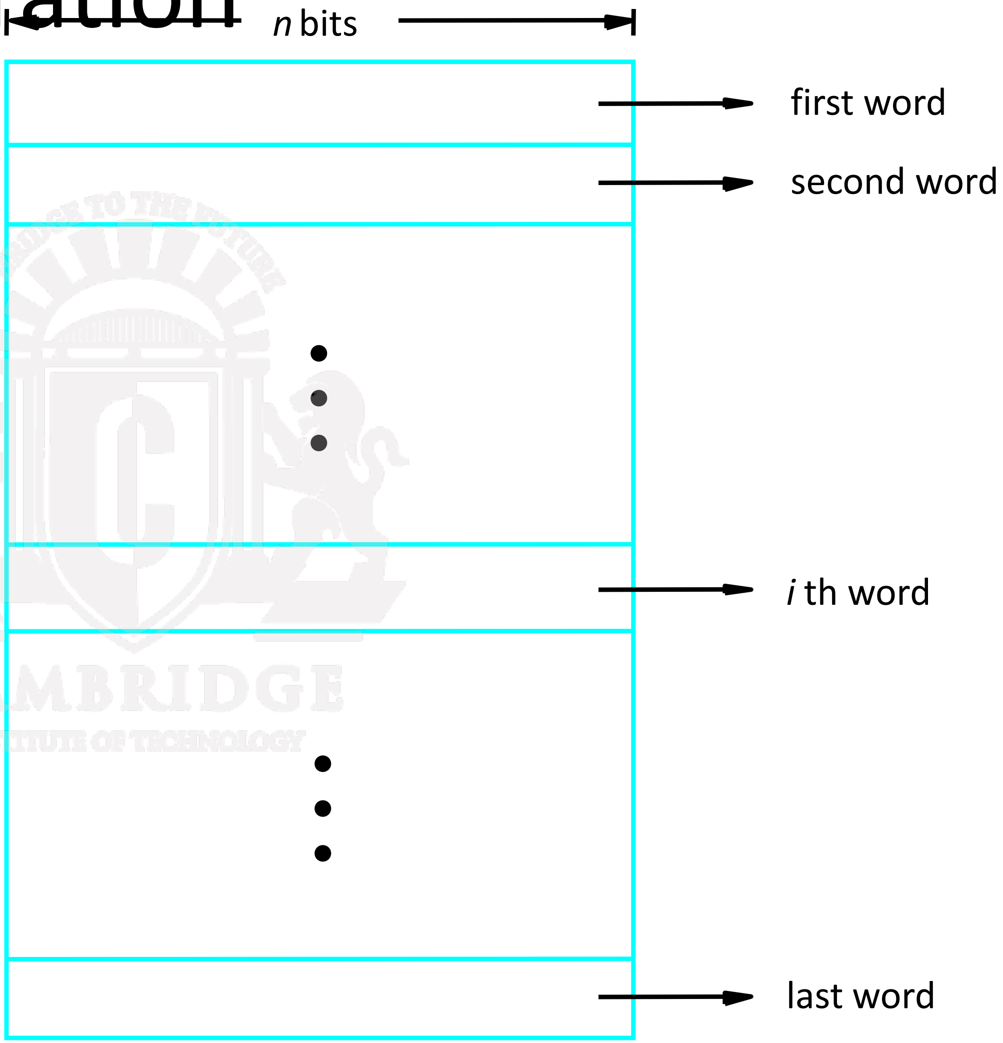
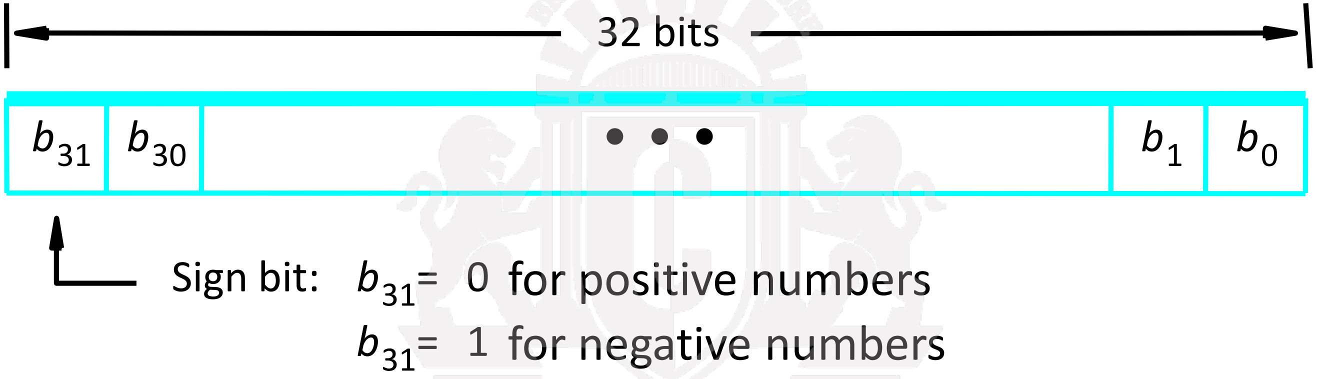


Figure 2.5. Memory words.

Memory Location, Addresses, and Operation

- 32-bit word length example



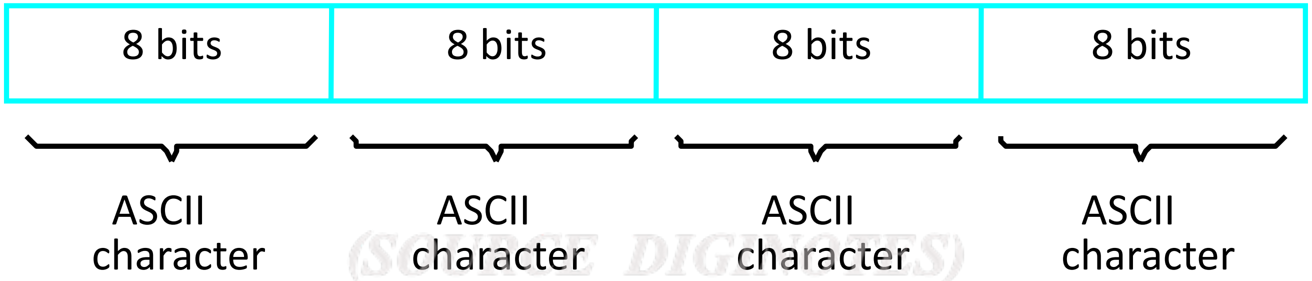
32 bits

b_{31} b_{30} • • • b_1 b_0

Sign bit: b_{31} = 0 for positive numbers

b_{31} = 1 for negative numbers

(a) A signed integer



8 bits | 8 bits | 8 bits | 8 bits

ASCII character | ASCII character | ASCII character | ASCII character

(b) Four characters

Memory Location, Addresses, and Operation

- To retrieve information from memory, either for one word or one byte (8-bit), addresses for each location are needed.
- A *k*-bit address memory has 2^k memory locations, namely $0 – 2^k-1$, called memory space.
- 24-bit memory: 2^{24} = 16,777,216 = 16M (1M=2^{20})
- 32-bit memory: 2^{32} = 4G (1G=2^{30})
- 1K(kilo)=2^{10}
- 1T(tera)=2^{40}

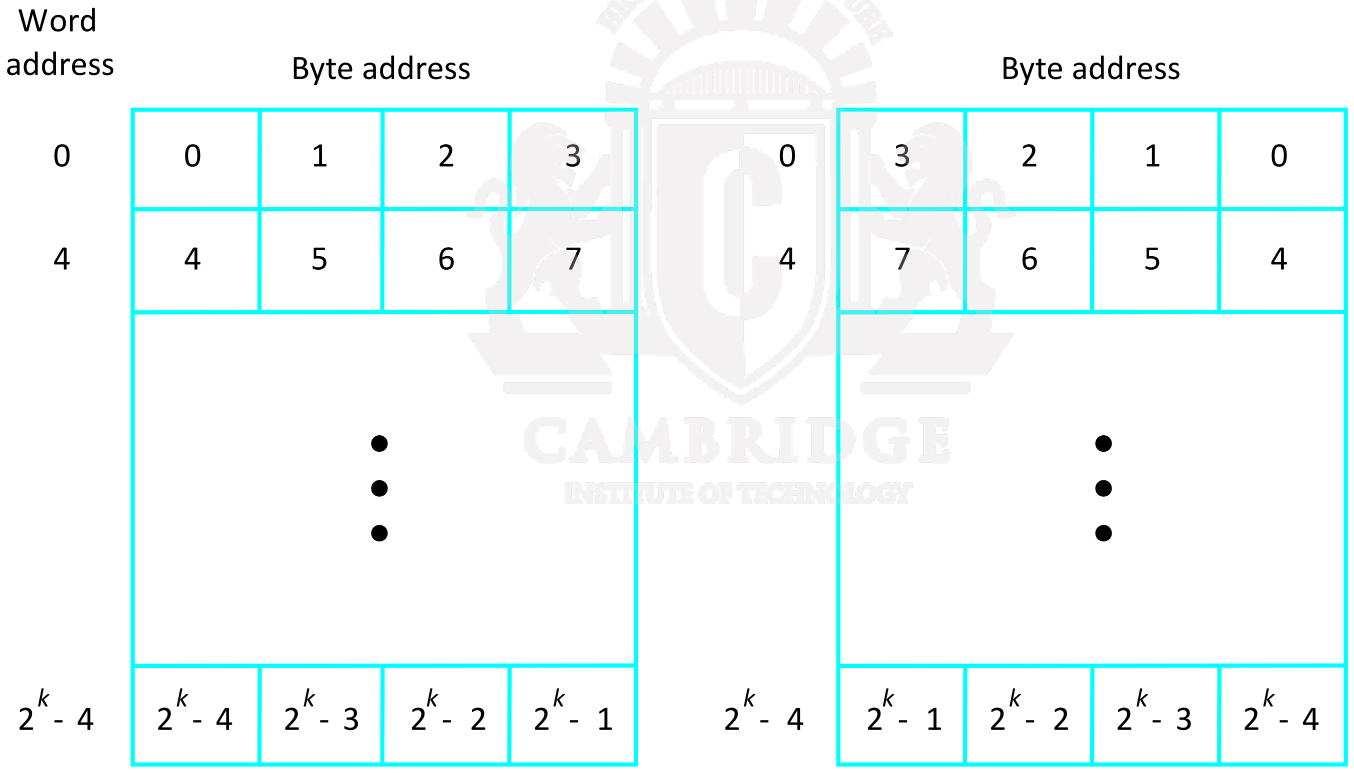
Memory Location, Addresses, and Operation

- It is impractical to assign distinct addresses to individual bit locations in the memory.
- The most practical assignment is to have successive addresses refer to successive byte locations in the memory – byte-addressable memory.
- Byte locations have addresses 0, 1, 2, … If word length is 32 bits, they successive words are located at addresses 0, 4, 8,…

Big-Endian and Little-Endian Assignments

Big-Endian: lower byte addresses are used for the most significant bytes of the word

Little-Endian: opposite ordering. lower byte addresses are used for the less significant bytes of the word



Word address	Byte address			
0	0	1	2	3
4	4	5	6	7
	⋮			
$2^k - 4$	$2^k - 4$	$2^k - 3$	$2^k - 2$	$2^k - 1$

(a) Big-endian assignment

	Byte address			
0	3	2	1	0
4	7	6	5	4
	⋮			
$2^k - 4$	$2^k - 1$	$2^k - 2$	$2^k - 3$	$2^k - 4$

(b) Little-endian assignment

Figure 2.7. Byte and word addressing.

Memory Location, Addresses, and Operation

- Address ordering of bytes
- Word alignment
  - Words are said to be aligned in memory if they begin at a byte addr. that is a multiple of the num of bytes in a word.
    - 16-bit word: word addresses: 0, 2, 4,….
    - 32-bit word: word addresses: 0, 4, 8,….
    - 64-bit word: word addresses: 0, 8,16,….
- Access numbers, characters, and character strings

Memory Operation

- Load (or Read or Fetch)
  - Copy the content. The memory content doesn’t change.
  - Address – Load
  - Registers can be used
- Store (or Write)
  - Overwrite the content in memory
  - Address and Data – Store
  - Registers can be used

“Must-Perform” Operations

- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

Register Transfer Notation

- Identify a location by a symbolic name standing for its hardware binary address (LOC, R0,…)
- Contents of a location are denoted by placing square brackets around the name of the location (R1←[LOC], R3 ←[R1]+[R2])
- Register Transfer Notation (RTN)

Assembly Language Notation

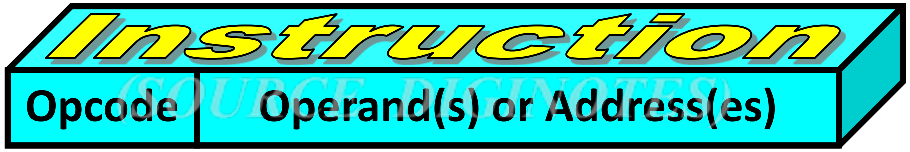
- Represent machine instructions and programs.
- Move LOC, R1 = R1←[LOC]
- Add R1, R2, R3 = R3 ←[R1]+[R2]

CPU Organization

- Single Accumulator
  - Result usually goes to the Accumulator
  - Accumulator has to be saved to memory quite often
- General Register
  - Registers hold operands thus reduce memory traffic
  - Register bookkeeping
- Stack
  - Operands and result are always in the stack

Instruction Formats

- Three-Address Instructions
  - ADD R1, R2, R3 R1 ← R2 + R3
- Two-Address Instructions
  - ADD R1, R2 R1 ← R1 + R2
- One-Address Instructions
  - ADD M AC ← AC + M[AR]
- Zero-Address Instructions
  - ADD TOS ← TOS + (TOS – 1)
- RISC Instructions
  - Lots of registers. Memory is restricted to Load & Store


Instruction

Opcode

Operand(s) or Address(es)

Instruction Formats

Example: Evaluate (A+B) ∗ (C+D)

- Three-Address

```
1. ADD   R1, A, B      ; R1 ← M[A] + M[B]
2. ADD   R2, C, D      ; R2 ← M[C] + M[D]
3. MUL   X, R1, R2     ; M[X] ← R1 ∗ R2
```

Instruction Formats

Example: Evaluate (A+B) ∗ (C+D)

- Two-Address

```
1.  MOV     R1, A                ; R1 ← M[A]
2.  ADD     R1, B                ; R1 ← R1 + M[B]
3.  MOV     R2, C                ; R2 ← M[C]
4.  ADD     R2, D                ; R2 ← R2 + M[D]
5.  MUL     R1, R2               ; R1 ← R1 ∗ R2
6.  MOV     X, R1                ; M[X] ← R1
```

Instruction Formats

Example: Evaluate (A+B) ∗ (C+D)

- One-Address

```
1. LOAD   A        ; AC ← M[A]
2. ADD    B        ; AC ← AC + M[B]
3. STORE  T        ; M[T] ← AC
4. LOAD   C        ; AC ← M[C]
5. ADD    D        ; AC ← AC + M[D]
6. MUL    T        ; AC ← AC ∗ M[T]
7. STORE  X        ; M[X] ← AC
```

Instruction Formats

Example: Evaluate (A+B) ∗ (C+D)

- Zero-Address

1. PUSH A ; TOS ← A
2. PUSH B ; TOS ← B
3. ADD ; TOS ← (A + B)
4. PUSH C ; TOS ← C
5. PUSH D ; TOS ← D
6. ADD ; TOS ← (C + D)
7. MUL ; TOS ← (C+D)∗(A+B)
8. POP X ; M[X] ← TOS

Instruction Formats

Example: Evaluate (A+B) * (C+D)

- RISC

```
1. LOAD   R1, A                  ; R1 ← M[A]
2. LOAD   R2, B                  ; R2 ← M[B]
3. LOAD   R3, C                  ; R3 ← M[C]
4. LOAD   R4, D                  ; R4 ← M[D]
5. ADD    R1, R1, R2             ; R1 ← R1 + R2
6. ADD    R3, R3, R4             ; R3 ← R3 + R4
7. MUL    R1, R1, R3             ; R1 ← R1 * R3
8. STORE  X, R1                  ; M[X] ← R1
```

Using Registers

- Registers are faster
- Shorter instructions
  - The number of registers is smaller (e.g. 32 registers need 5 bits)
- Potential speedup
- Minimize the frequency with which data is moved back and forth between the memory and processor registers.

Instruction Execution and Straight-Line Sequencing

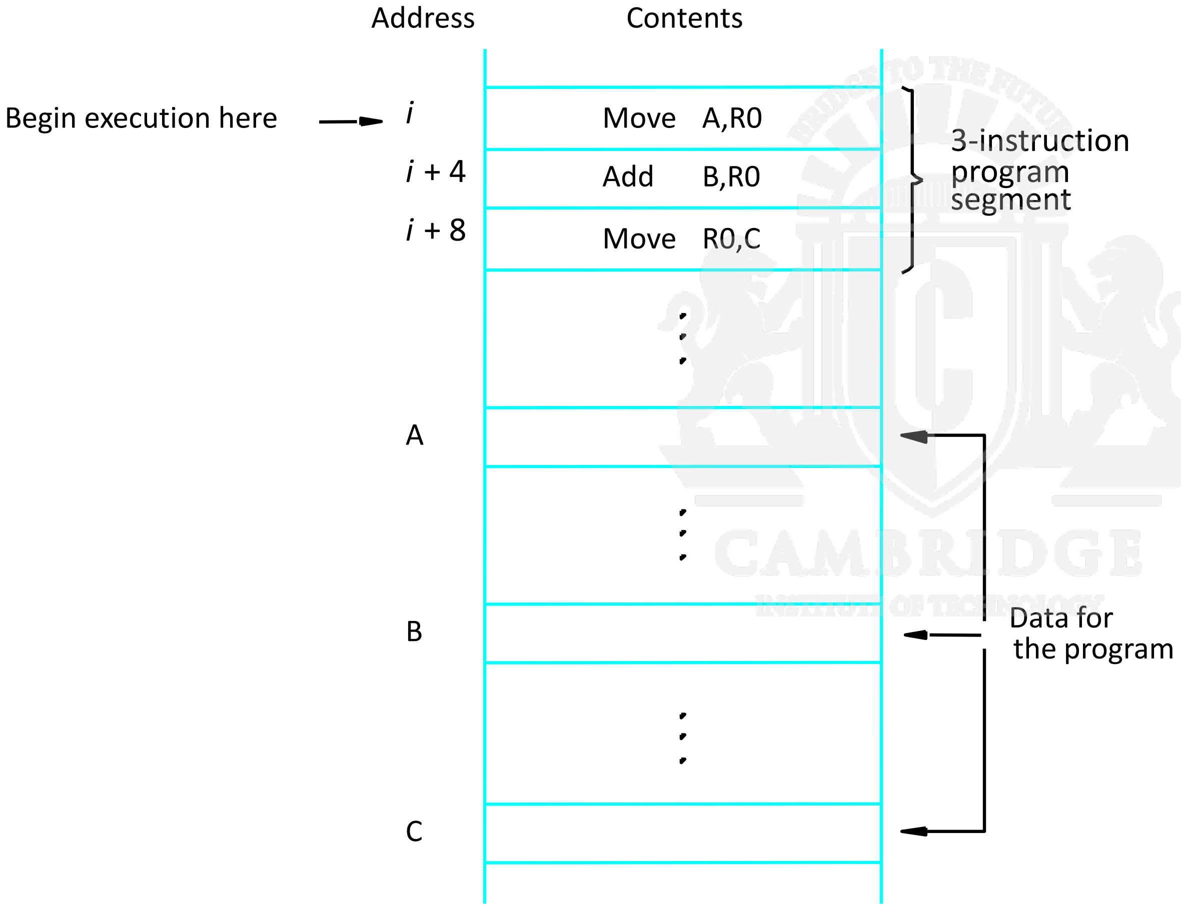


Figure 2.8. A program for C ← [A] + [B].

Assumptions:

- One memory operand per instruction
- 32-bit word length
- Memory is byte addressable
- Full memory address can be directly specified in a single-word instruction

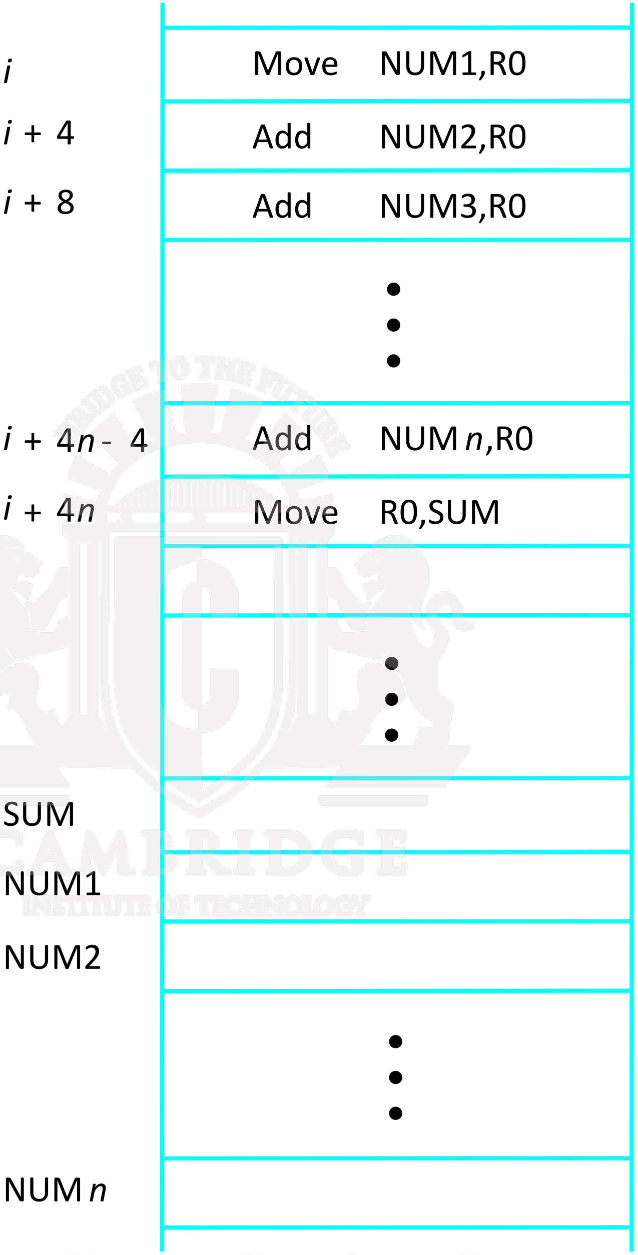
Two-phase procedure

-Instruction fetch

-Instruction execute

Page 43

Branching



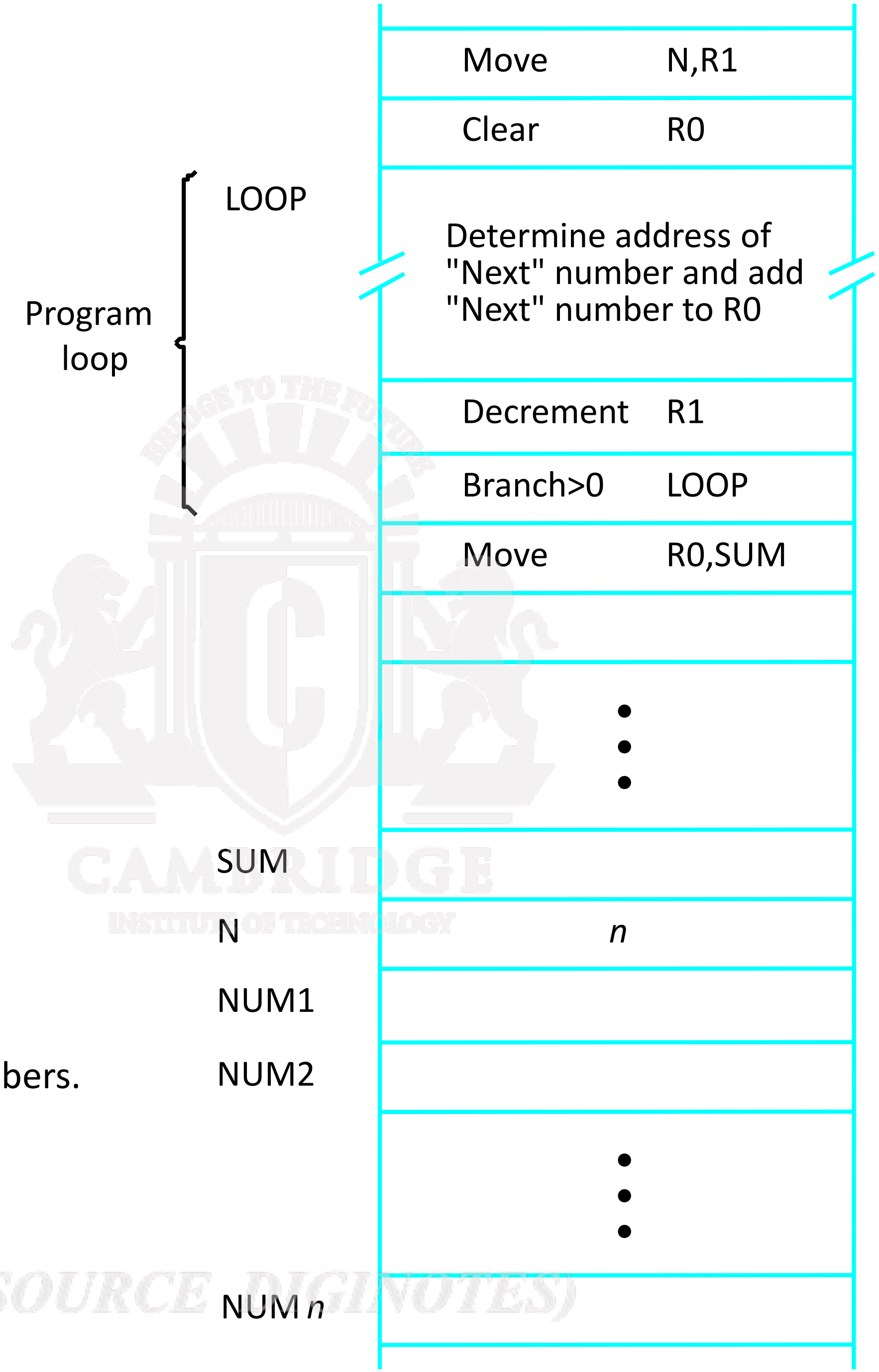
i	Move	NUM1,R0
$i + 4$	Add	NUM2,R0
$i + 8$	Add	NUM3,R0
	• • •	
$i + 4n - 4$	Add	NUMn,R0
$i + 4n$	Move	R0,SUM
	• • •	
SUM		
NUM1		
NUM2		
	• • •	
NUMn		

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

Branching

Branch target

Conditional branch



```
              Move          N,R1
              Clear         R0
Program  LOOP
loop          Determine address of
              "Next" number and add
              "Next" number to R0
              Decrement     R1
              Branch>0      LOOP
              Move          R0,SUM

                    ⋮

         SUM
         N          n
         NUM1
         NUM2
                    ⋮
         NUM n
```

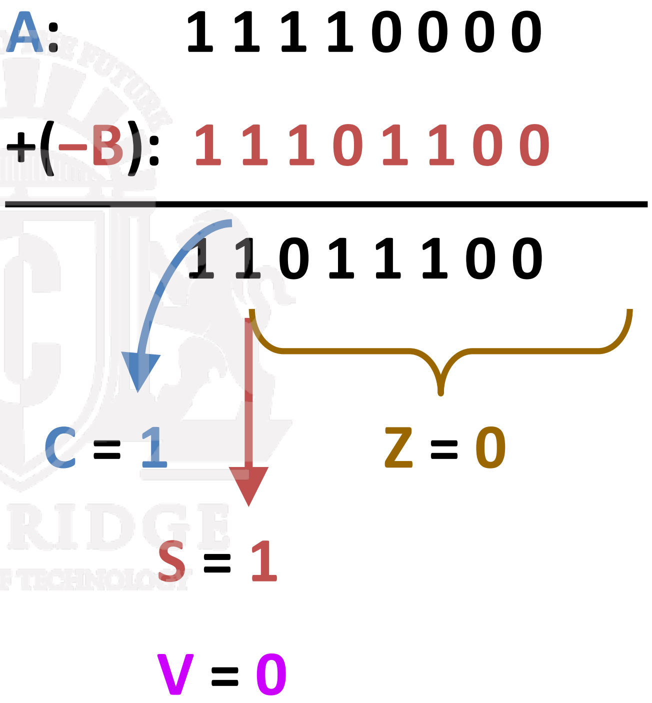
Figure 2.10. Using a loop to add *n* numbers.

Condition Codes

- Condition code flags
- Condition code register / status register
- N (negative)
- Z (zero)
- V (overflow)
- C (carry)
- Different instructions affect different flags

Conditional Branch Instructions

- Example:
  - A: 1 1 1 1 0 0 0 0
  - B: 0 0 0 1 0 1 0 0



A: 1 1 1 1 0 0 0 0

+(–B): 1 1 1 0 1 1 0 0

1 1 0 1 1 1 0 0

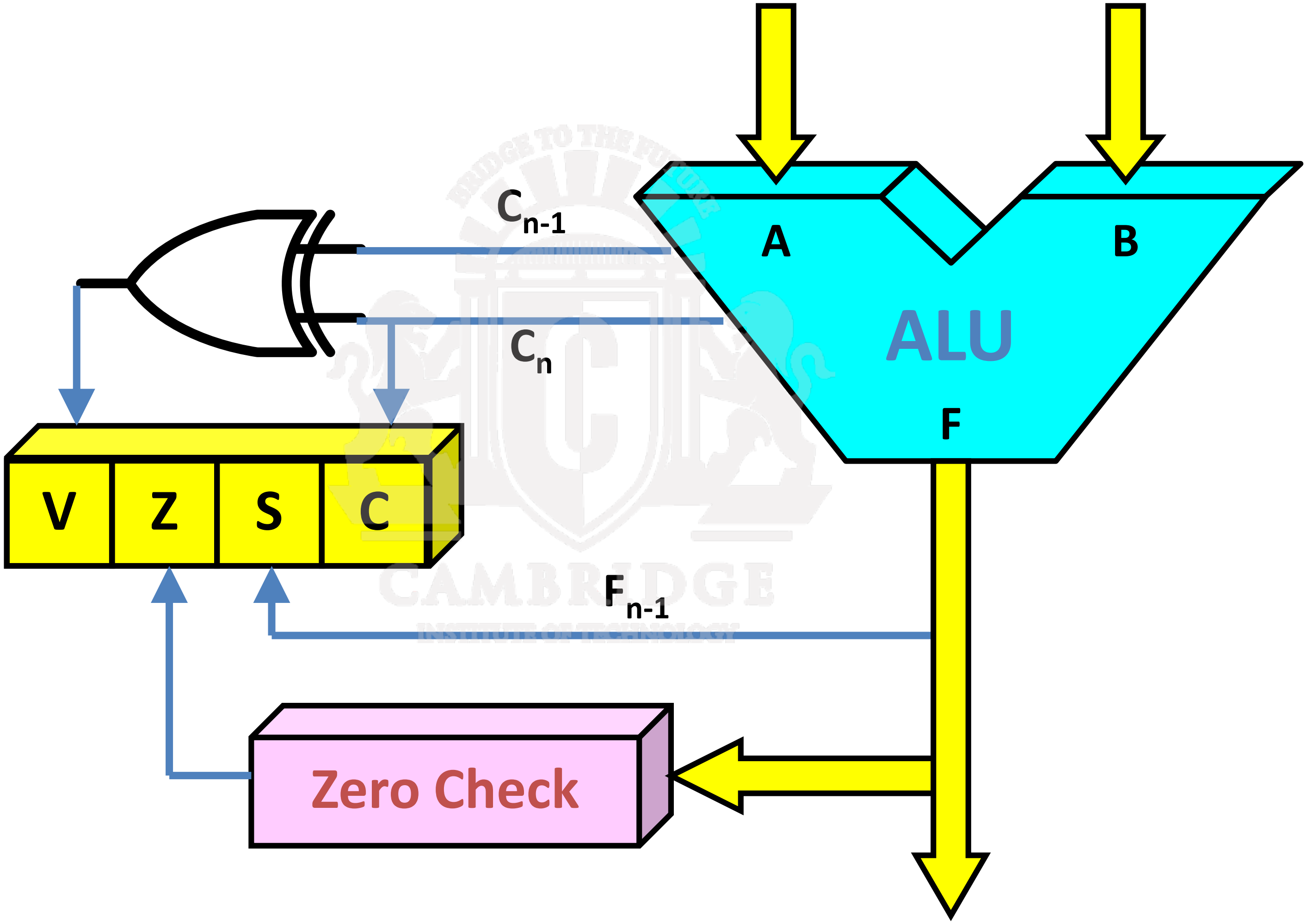
C = 1

Z = 0

S = 1

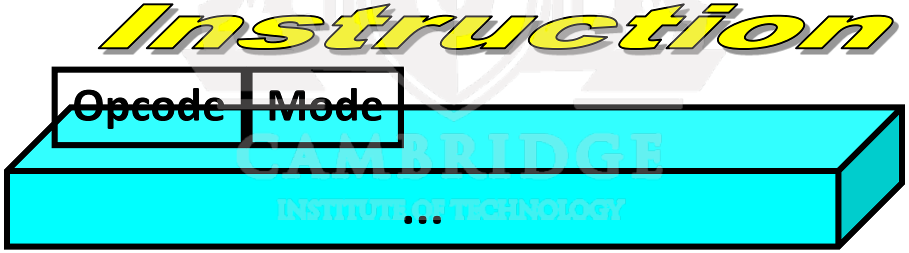
V = 0

Status Bits



Addressing Modes

Addressing modes:-The different ways in which the location of an operand is specified in an instruction are referred to as addressing modes.



CHAPTER 2: Machine instructions and programs

Courtesy: Text book: Carl Hamacher 5th Edition

Addressing Modes

- **Register mode** :-The operand is the content of a processor register; the name(address) is given in the processor.

  Ex:- Move R1,R2

- **Absolute mode**:-The operand is in a memory location; the address of this location is given explicitly in the instruction.

  Ex:- Move LOC,R2

Immediate mode:- The operand is given explicitly in the instruction(clearly immediate mode is to specify the value of a source operand)

Ex:- Move #200,R0

For ex:- A=B+6;

Addressing Modes

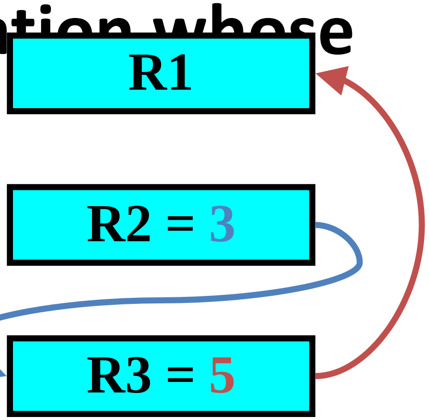
- **Indirect Address:**

  The effective address of the operand is the content of a register or memory whose address appears in the instruction

Ex:- 1) Add (R1) R0 2) Add (AR),R0

Addressing Modes

- **Indirect mode**
  - **The effective address of the operand is the contents of a register or memory location whose address appears in the instructions**

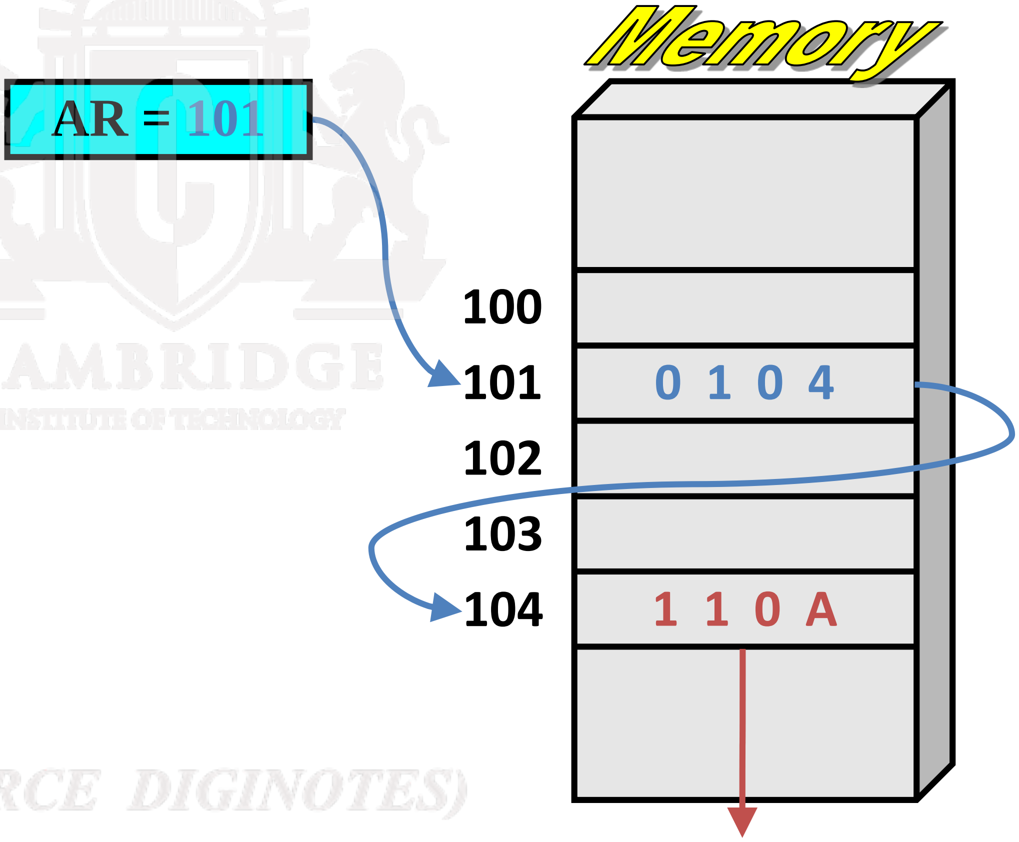


Ex:- Add R1, (R2);

Add (AR),R0;

Addressing Modes

- Indirect Address
  - – Indicate the memory location that holds the address of the memory location that holds the data



(SOURCE DIGINOTES)

Addressing Modes

- **Index mode:-** The effective address of the operand is generated by adding a constant value to the content of a register

Ex:- 1) Add 20(R1),R2 2) Add 1000(R1),R2

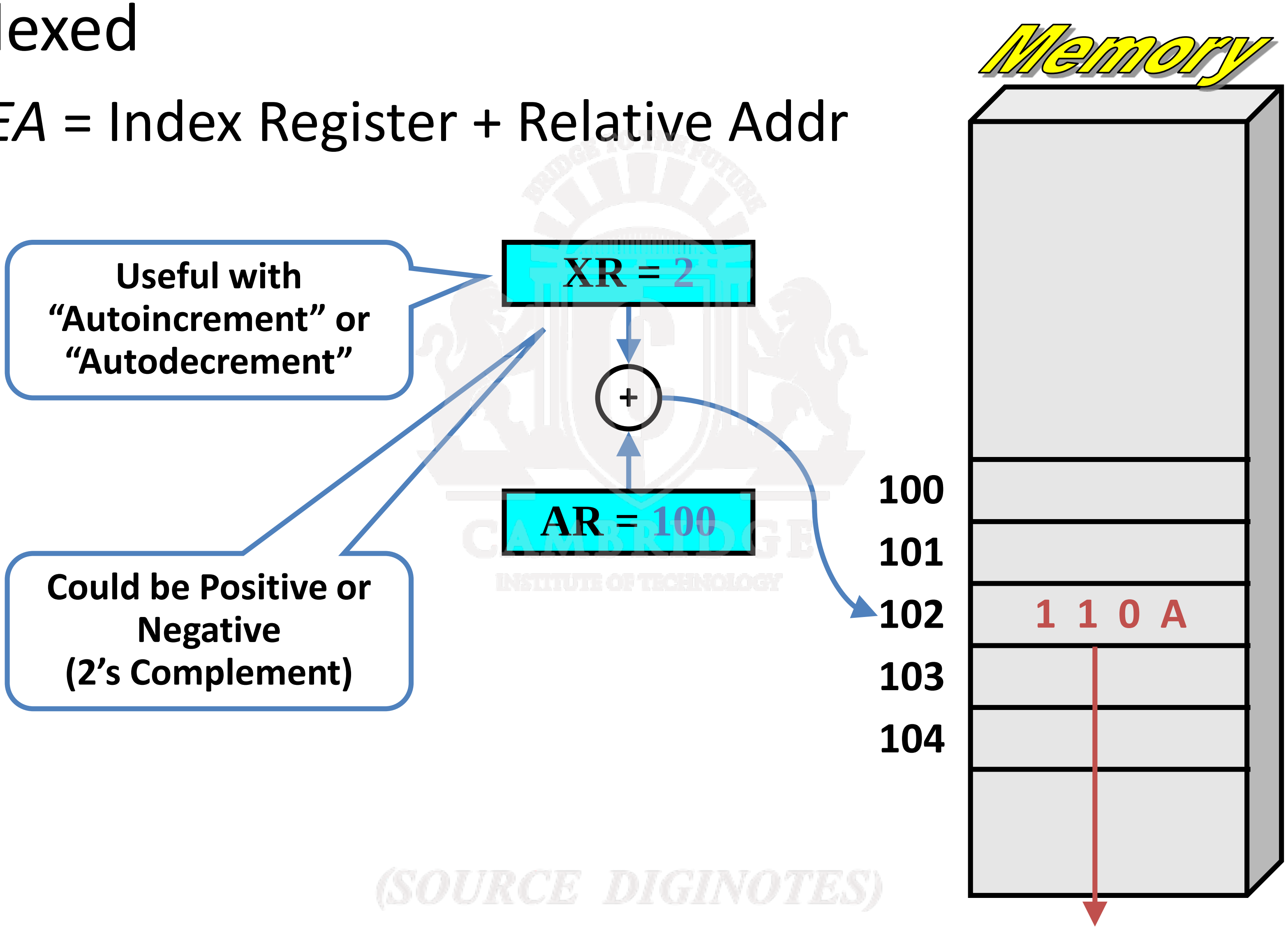
Index register

$X(R_i)$: $EA = X + [R_i]$

- The constant X may be given either as an explicit number or as a symbolic name representing a numerical value.
- If X is shorter than a word, sign-extension is needed.

Addressing Modes

- Indexed
  - *EA* = Index Register + Relative Addr



Useful with “Autoincrement” or “Autodecrement”

Could be Positive or Negative (2’s Complement)

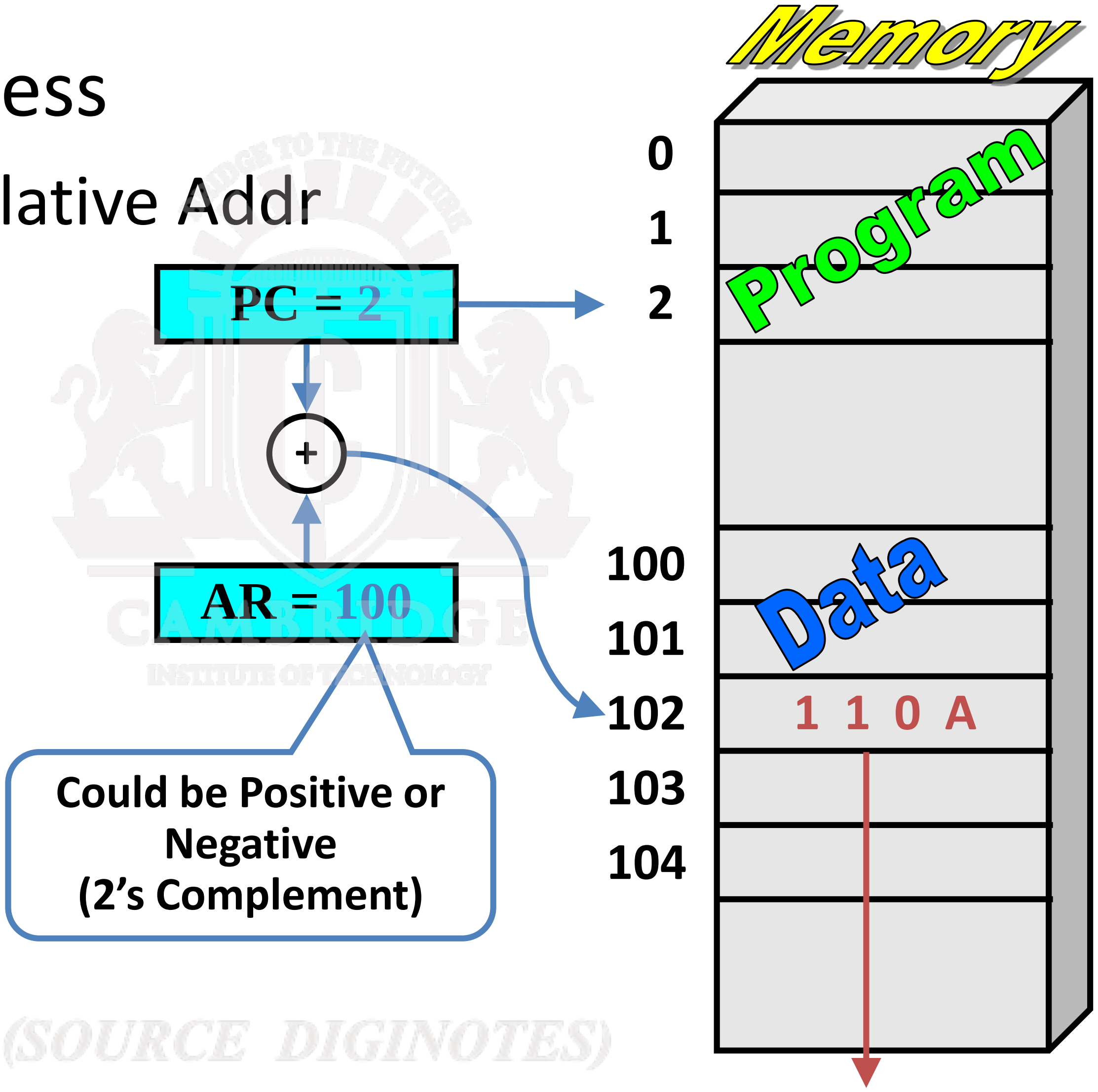
Addressing Modes

- **Relative mode** :-The effective address is determined by the Index mode using the program counter in place of the general purpose register R*i.*

- **Ex:- Add 20(PC),R0**

Addressing Modes

- Relative Address
  - *EA* = PC + Relative Addr



Additional Modes

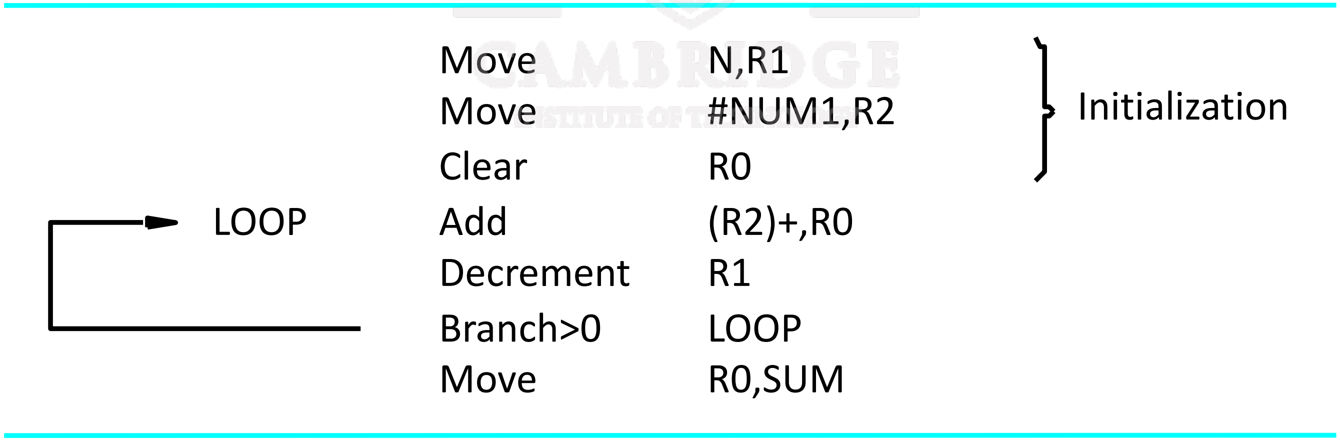
- **Auto increment mode** :-The effective address of the operand is the content of a register specified in the instruction determined. After accessing the operand, the contents of this registers are automatically incremented to point to the next item in a list.
- **Ex:- (R*i*)+**

- **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.
- **Ex:- -(R*i*)**

Additional Modes

- Autoincrement mode – the effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list.

- Autodecrement mode: -(R_i) – decrement first



```
                  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.

Addressing Modes

Name	**Assem bler syn tax**	**Addressing function**
Immediate	#V alue	Op erand = Value
Register	R *i*	EA = R *i*
Absolute (Direct)	LOC	EA = LOC

Indirect	(R *i*)
	(LOC)
		EA = [R *i*]
		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
Autoincremen t	(R *i*)+	EA = [R *i*] ; Incremen t R *i*
Autodecrement	–(R *i*)	Decremen t R *i* ; EA = [R *i*]

Assembly Language

- Machine instructions are represented by patterns of 0's and 1's –awkward to deal
- So we use symbolic names to represent the patterns. so far we have used normal words, such as **Move, Add, and Branch,** for the instruction operation to represent the corresponding binary code patterns.
- When writing programs for a specific computer, such words are normally replaced by acronyms called ***mnemonics,*** such as **MOV,ADD,BR** .similarly we use the notation **R3** to refer to **register 3** and **LOC** to refer to a **memory location.**
- A complete set of names and rules for their use constitutes a programming language, generally referred to as an **assembly language**

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.

Assembly Language : Types of Instructions

- Data Transfer Instructions

Data value is not modified

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

Assembler DIrectives

- In addition to provide a mechanism for representing instructions in a program.
- The assembly language allows the programmer to specify other information needed to translate the source program into object program.
- Suppose that the name SUM is used to represent the value 200. This can be conveyed to the assembler through a statement as

SUM EQU 200

This statement simply informs the assembler that the SUM should be replaced by the value 200 where ever it appears in the program.

Such statements are **assembler directives**

Assembler Directives

	Address Label	Operation	Addressing or Data operation
Ex:-	**SUM**	**EQU**	**200**
		ORIGIN	**204**
	N	**DATAWORD**	**100**
	NUM1	**RESERVE**	**400**

This **ORIGIN** directive tells the assembler where in the memory to place the data block that follows. In this case the location specified has the address 204.since this location is to be loaded with value 100.

- A **DATAWORD** directive is used to inform the assembler of this requirement .It states that the data value is to be placed in the memory word at address 204

Assembler Directives

- Any statement that results in instructions of data being placed in a memory location may be given a **memory address label.** This label is assigned a value equal to the address of that location.

- Because the **DATAWORD** statement is given the label N, the Name N is assigned the value 204.Whenever N is encountered in the rest of the program it will be replaced with this value.

- The **RESERVE** directive declares that a memory block of 400 bytes to be reserved for data and that the name NUM1 is to be associated with address 208

(SOURCE DIGINOTES)

Assembler DIrectives

```
START     MOVE   N,R1
          MOVE  #NUM1,R2
          CLR R0
LOOP      ADD (R2),R0
           ADD #4,R2
           DEC R1
           BGTZ  LOOP
           MOVE R0,SUM
           RETURN
           END START
```

Assembler DIrectives

- The Last Statement in the source program is the assembler directive **END**, which tells that this is the end of source program text. The END directive includes the label **START**, which is the address of the location at which execution of the program is to begin.

- The **RETURN** assembler directive identifies the point at which execution of the program should be terminated.
- Most assembly languages require statements in a source program to be written in the form

Label Operation Operand(s) Comment;

These four fields are separated by an appropriate delimiter, typically one or more blank characters.

Number Notations

- Numbers can be specified as an operand in an instruction using different represent representations.

- **Ex:- ADD #93,R1 (Decimal).**
-
- **ADD #%01011101,R1 (Binary)**

- **ADD #$5D,R1 (Hexadecimal)**

Basic Input/Output Operations:Program-Controlled I/O

- Read in character input from a keyboard and produce character output on a display screen.

Rate of data transfer (keyboard, display, processor)

Difference in speed between processor and I/O device creates the need for mechanisms to synchronize the transfer of data.

A solution: on output, the processor sends the first character and then waits for a signal from the display that the character has been received. It then sends the second character. Input is sent from the keyboard in a similar way.

Program-Controlled I/O Example

- Registers
- Flags
- Device interface

Program-Controlled I/O Example

The keyboard and the display are separate device as shown in fig. the action of striking a key on the keyboard does not automatically cause the corresponding character to be displayed on the screen.

One block of instructions in the I/O program transfers the character into the processor, and another associated block of instructions causes the character to be displayed

Striking a key stores the corresponding character code in an 8-bit buffer register associated with the keyboard. Let us call this register **DATAIN,** as shown in fig

Program-Controlled I/O Example

To inform the processor that a valid character is in **DATAIN**, a status control flag, **SIN**, is set to 1. A program monitors **SIN**, and when **SIN** is set to 1, the processor reads the contents of **DATAIN**. When the character is transferred to the processor, **SIN** is automatically cleared to 0. If a second character is entered at the keyboard, **SIN** is again set to 1, and the processor repeat

An analogous process takes place when characters are transferred from the processor to the display. A buffer register, **DATAOUT**, and a status control flag, **SOUT**, are used for this transfer. When **SOUT** equals 1, the display is ready to receive a character.

Program-Controlled I/O Example

- Machine instructions that can check the state of the status flags and transfer data:

- **READWAIT Branch to READWAIT if SIN = 0**
  Input from DATAIN to R1

  WRITEWAIT Branch to WRITEWAIT if SOUT = 0
  Output from R1 to DATAOUT

Program-Controlled I/O Example

- Memory-Mapped I/O – some memory address values are used to refer to peripheral device buffer registers. No special instructions are needed. Also use device status registers.

```
READWAIT  Testbit   #3, INSTATUS
          Branch=0  READWAIT
          MoveByte  DATAIN, R1
```

- **The write may be implemented as**

```
WRITEWAIT    Testbit   #3, OUTSTATUS
             Branch=0  WRITEWAIT
             MoveByte  R1,DATAOUT
```

Stacks

A stack is a small area in the memory of a computer used to store data elements. The main feature of the stack is that, the elements can be added or removed to one end only and the other is fixed.

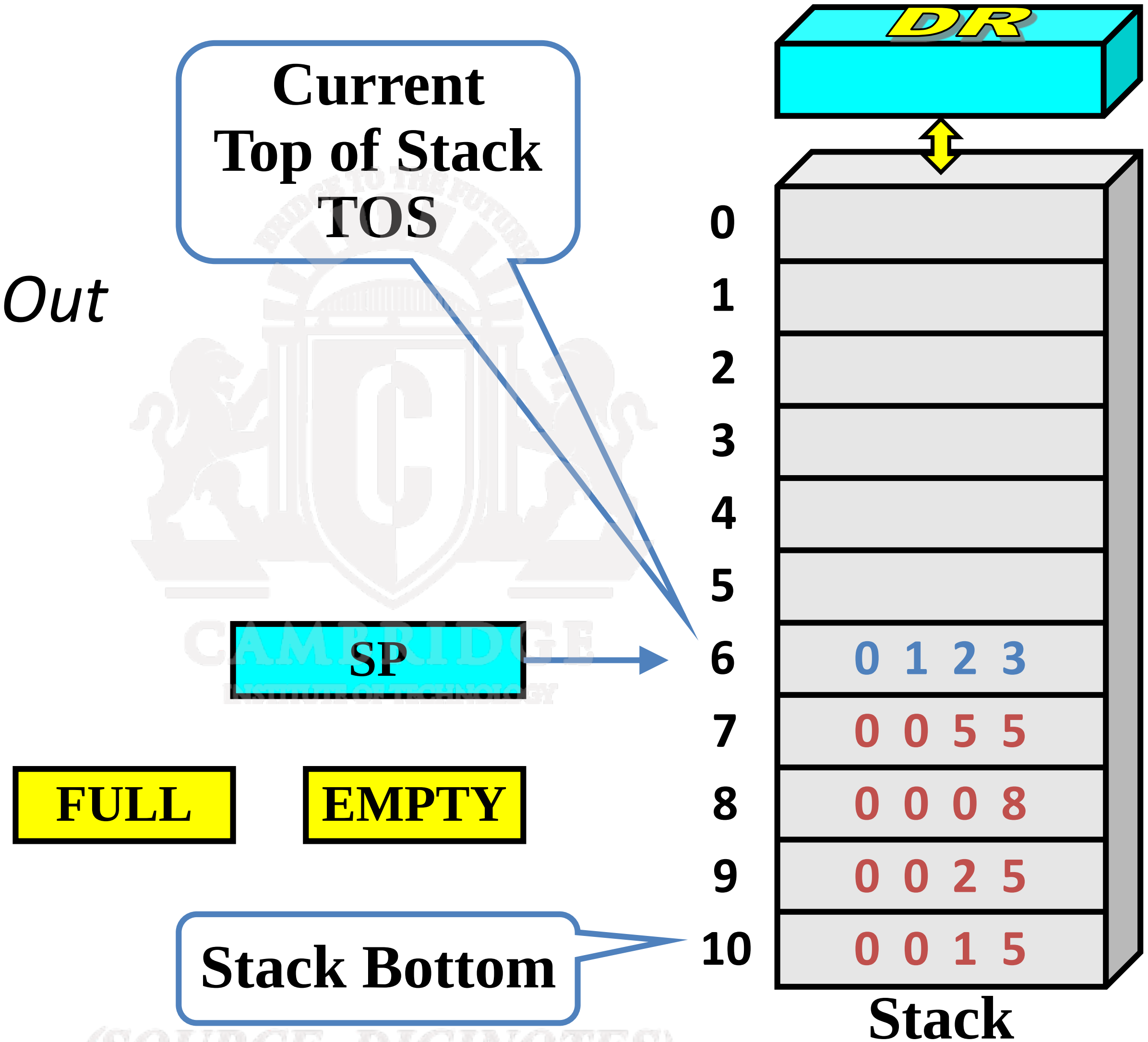
The open end is called top of the stack**(TOS),**and the fixed end is **bottom.** We use the term **Push** and **POP** to denote the operations on the stack.

When elements are pushed into the stack they are placed in successively **lower address locations**. Thus the stack **grows** in the direction of **decreasing** memory address

Stack : Stack Organization

- LIFO

  Last In First Out



Stack Organization

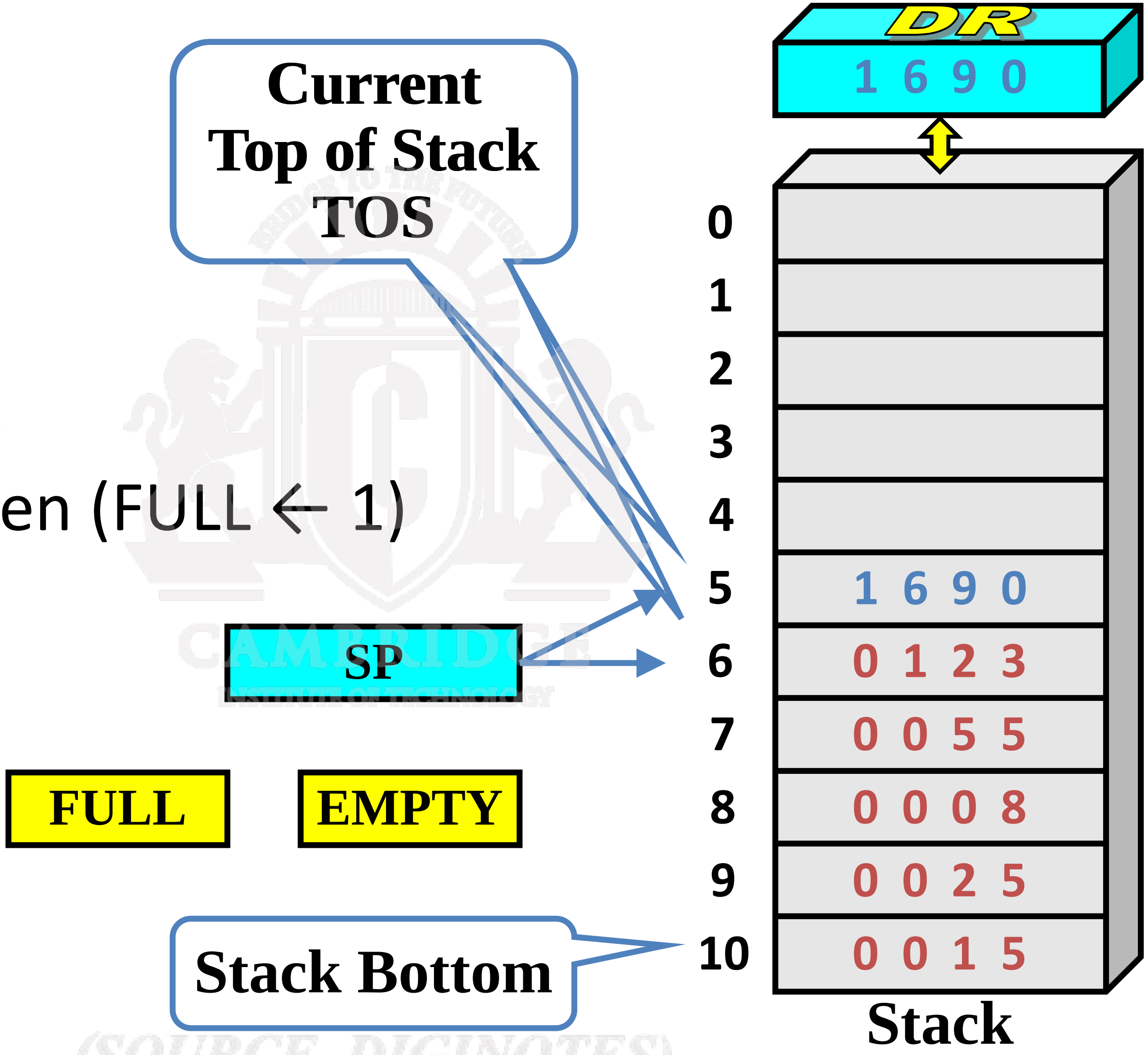
- PUSH

  SP ← SP – 1

  M[SP] ← DR

  If (SP = 0) then (FULL ← 1)

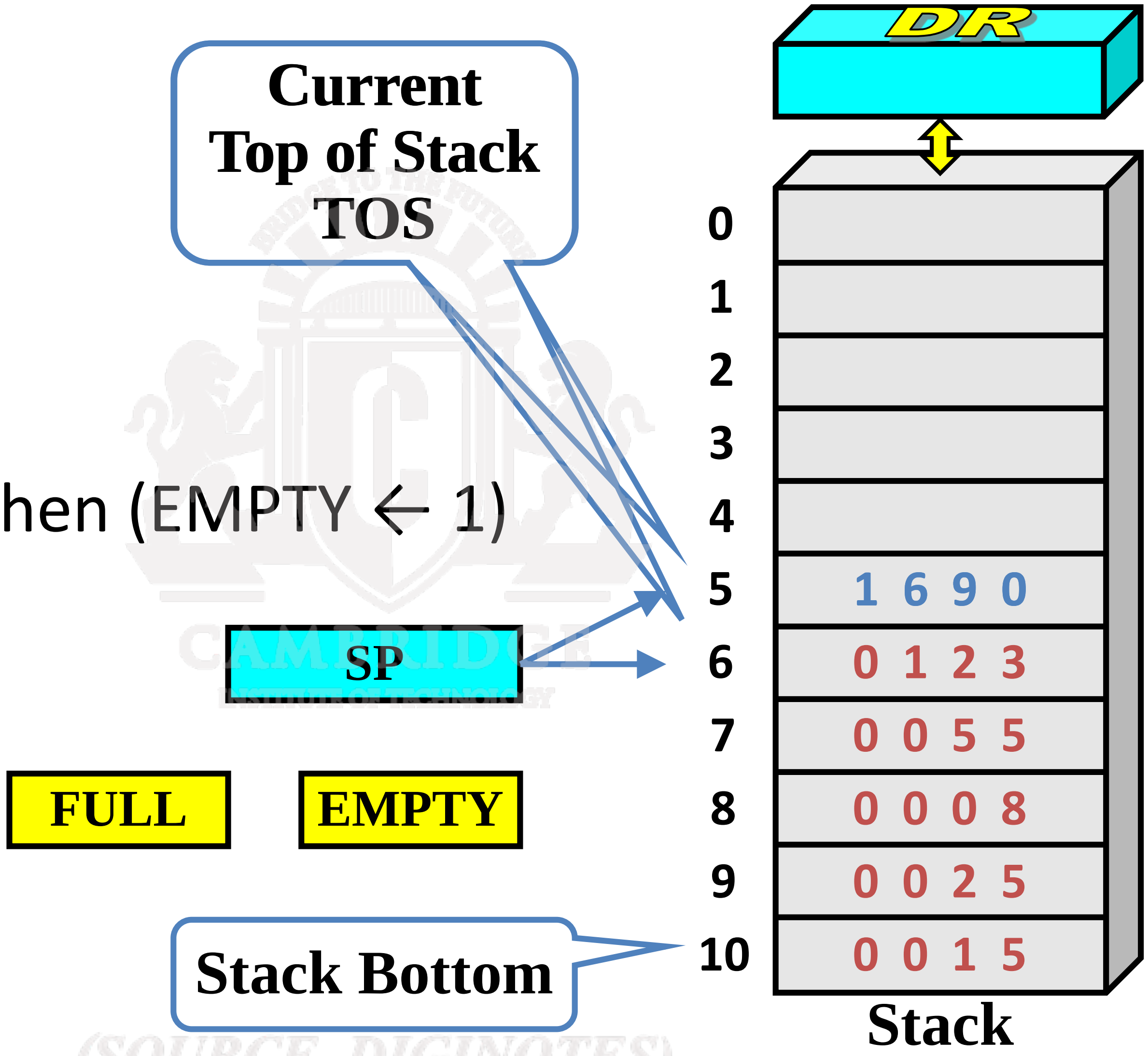
  EMPTY ← 0



Stack Organization

- POP

```
DR ← M[SP]
SP ← SP + 1
If (SP = 11) then (EMPTY ← 1)
FULL ← 0
```



0	
1	
2	
3	
4	
5	1 6 9 0
6	0 1 2 3
7	0 0 5 5
8	0 0 0 8
9	0 0 2 5
10	0 0 1 5

Stack Organization

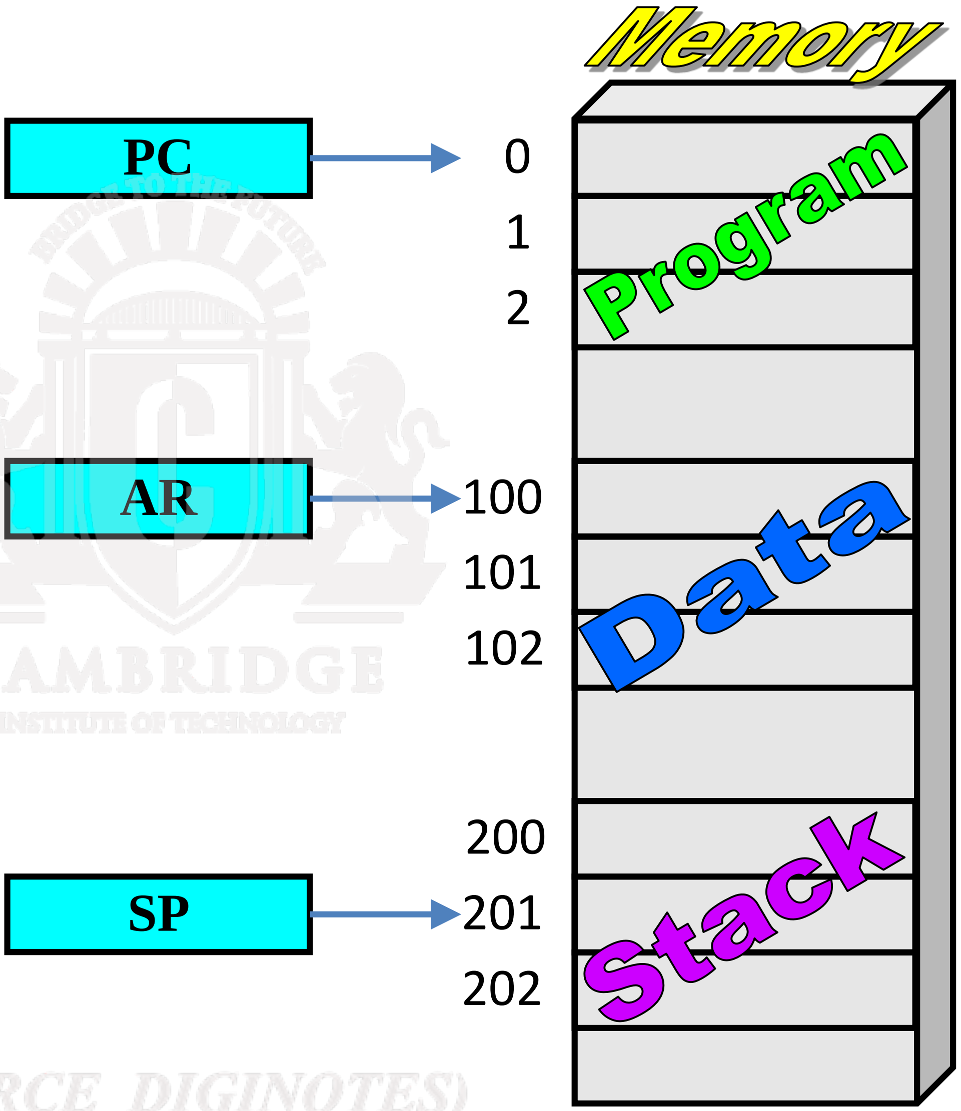
- Memory Stack
  - PUSH

    SP ← SP – 1

    M[SP] ← DR
  - POP

    DR ← M[SP]

    SP ← SP + 1



Stack organization

- Assume a byte addressable memory and a word length of a data to be **32-bits**
- The push operation places a data item above the current top of the stack. i.e. the **SP** is to be decremented before data can be placed.
- So to transfer a data from a memory location NUM to the have the following instructions.
  - **Subtract #4,SP**
  - **Move NUM,(SP)**
- **To remove a data**
  - **Move (SP),NUM**
  - **Add #4,SP**

(SOURCE DIGINOTES)

Reverse Polish Notation

- Infix Notation

  $A + B$

- Prefix or Polish Notation

  $+ A\ B$

- Postfix or Reverse Polish Notation (RPN)

  $A\ B +$

$$A * B + C * D \xrightarrow{\text{RPN}} A\ B * C\ D * +$$

(2) (4) * (3) (3) * +

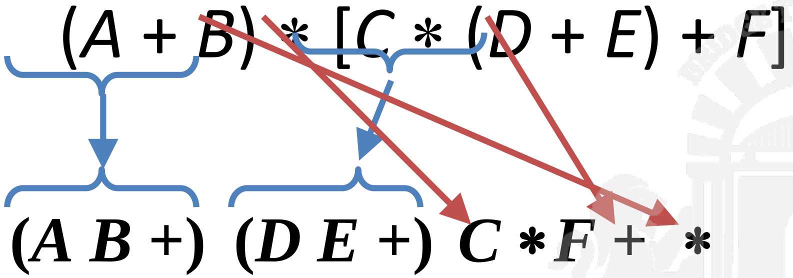
(8) (3) (3) * +

(8) (9) +

17

Reverse Polish Notation

- Example



$(A + B) * [C * (D + E) + F]$

$(A\ B\ +)\ (D\ E\ +)\ C * F + \ *$

Reverse Polish Notation

- Stack Operation

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

PUSH 3

PUSH 4

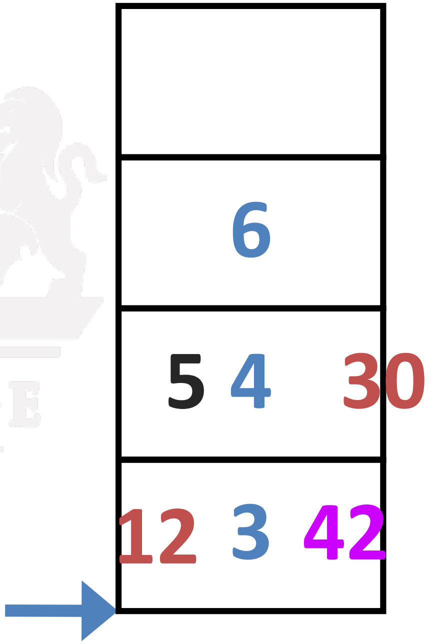
MULT

PUSH 5

PUSH 6

MULT

ADD



STACK:-POP and PUSH

- Suppose that a stack runs from location 2000(BOTTOM) down no further location 1500. The stack pointer is loaded initially with the address value 2004.
- SP is decremented by 4 before new data are stored onto the stack, hence a initial value of 2004 means that the first item pushed onto the stack will be at location 2000.
- To prevent either pushing an item on full stack or popping an item off an empty stack, the single-instruction push and pop operations can be replaced by the instruction sequences
- The compare instruction

Compare src, dst **performs** **[dst]-[src]**

Doesn’t change the operand value

Stack: POP and PUSH

- SAFEPOP Compare #2000,SP
- Branch>0 EMPTYERROR
- Move (SP)+,ITEM.
- SAFEPUSH Compare #1500,SP
- Branch<=0 FULLERROR
- Move NEWITEM,-(SP).

Subroutines

- In a given program, it is often necessary to perform a particular subtask many times on different data-values. Such a subtask is usually called a subroutine.

For example,

- a subroutine may evaluate the sine function or sort a list of values into increasing or decreasing order.
- It is possible to include the block of instructions that constitute a subroutine at every place where it is needed in the program. However, to save space, only one copy of the instructions that constitute the subroutine is placed in the memory, and any program that requires the use of the subroutine simply branches to its starting location..

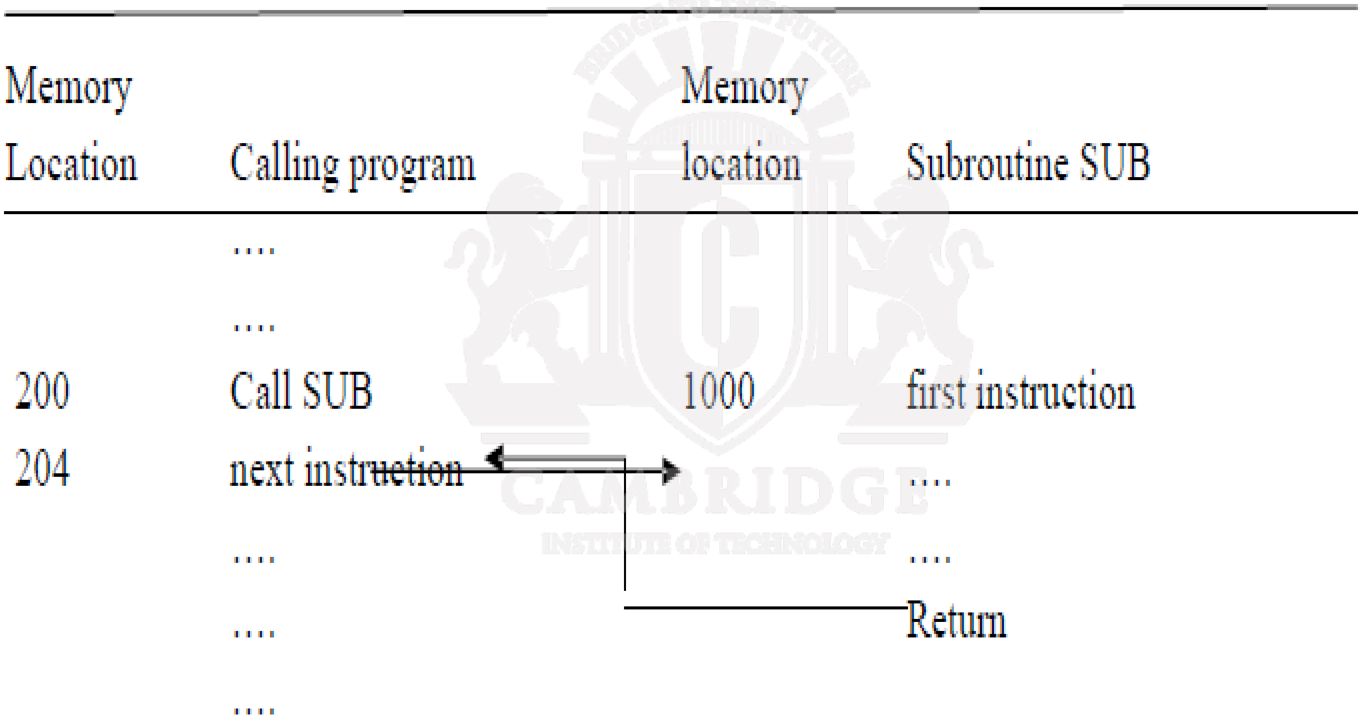
Subroutines

- When a program branches to a subroutine we say that it is calling the subroutine. The instruction that performs this branch operation is named a **Call instruction**

- After a subroutine has been executed, the calling program must resume execution, continuing immediately after the instruction that called the subroutine. The subroutine is said to return to the program that called it by executing a **Return instruction**

- The way in which a computer makes it possible to call and return from subroutines is referred to as its **subroutine linkage method**

Subroutines

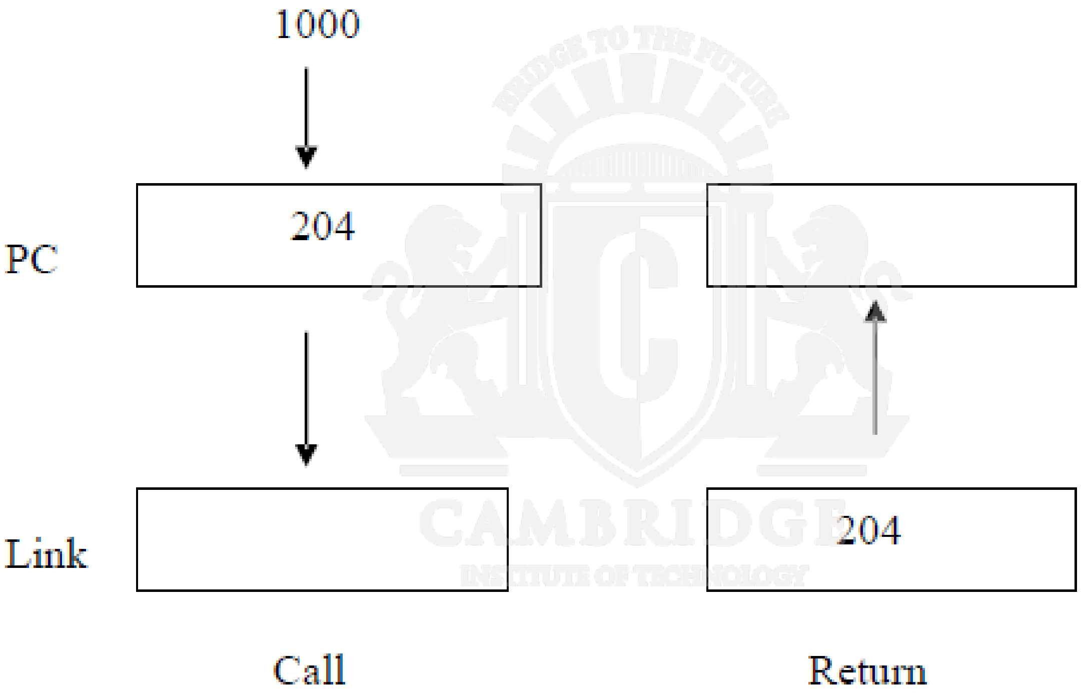


Memory Location	Calling program	Memory location	Subroutine SUB
	....		
	....		
200	Call SUB	1000	first instruction
204	next instruction		....
	....		....
	....		Return
	....		

Subroutines

- 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 just a special branch instruction that performs the following operations
- • *Store the contents of the PC in the link register*
- Branch to the target address specified by the instruction
- The Return instruction is a special branch instruction the performs the operation
- • *Branch to the address contained in the link register*

Subroutines



Subroutine Nesting

- A common programming practice, called **subroutine nesting**, is to have one subroutine call another. 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**.
- This suggests that the return addresses associated with subroutine calls should be pushed onto a **stack**. A particular register is designated as the **stack pointer**, SP, to be used in this operation. The stack pointer points to a stack called the **processor stack**. The Call instruction pushes the contents of the PC onto the processor stack and loads the subroutine address into the PC. The 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**. Parameter passing may be accomplished in several ways. The parameters may be placed in **registers** or in **memory locations**, where they can be accessed by the subroutine. Alternatively, the parameters may be placed on the **processor stack** used for saving the return address.

- The purpose of the subroutines is to add a list of numbers. Instead of passing the actual list entries, the calling program passes the address of the first number in the list.
- This technique is called **passing by reference.** The second parameter is **passed by value**, that is, the actual number of entries, n, is passed to the subroutine.

Parameter Passing

- The exchange of **information** between a **calling program** and a **subroutine** is referred to as **parameter passing**. Parameter passing may be accomplished in several ways. The parameters may be placed in **registers** or in **memory locations**, where they can be accessed by the subroutine. Alternatively, the parameters may be placed on the **processor stack** used for saving the return address.

- The purpose of the subroutines is to add a list of numbers. Instead of passing the actual list entries, the calling program passes the **address** of the first number in the list.
- This technique is called **passing by reference.** The second parameter is **passed by value**, that is, the actual number of entries, n, is passed to the subroutine.

Parameter Passing

```
                Move  N,R1
                Move  #NUM1,R2
                Clear  R0
                Call  ARRAYADD
                Move R0,SUM
                ……..
                END
ARRAYADD   Add  (R2)+,R0
                Decrement  R1
                Branch>0  ARRAYADD
                Return
```

Calling (main) Program

Subroutine

Stack Frame

During execution of the subroutine, **six locations** at the top of the stack contain entries that are needed by the subroutine.

These locations constitute a ***private workspace*** for the subroutine, created at the time the subroutine is entered and freed up when the subroutine returns control to the calling program. Such space is called a **stack frame** .

Stack Frame

-
- Move #NUM1,-(SP)
- Move N,-(SP)
- **Call ARRAYADD**
- Move 4(SP),SUM
- Add #8,SP
- ……..
- END

Calling (main) Program

Stack Frame

```
ARRAYADD   Move Multiple R0-R2,-(SP)
           Move 16(SP),R1
           Move 20(SP),R2

           Clear R0
BACK       Add  (R2)+,R0
           Decrement R1
           Branch>0  LOOP
           Move R0,20(SP)
           Move Multiple (SP)+,R0-R2
           Return
```

Subroutine

Stack Frame

- In addition to the stack pointer SP, it is useful to have another pointer register, called the **Frame pointer (FP)**, for convenient access to the parameters passed to the subroutine and to the local memory variables used by the subroutine
- These **local variables** are only used within the **subroutine**, so it is appropriate to allocate space for them in the stack frame associated with the subroutine. We assume that **four parameters** are passed to the subroutine, **three local variables** are used within the subroutine, and registers **R0** and **R1** need to be saved because they will also be used within the subroutine.
- The pointers **SP** and **FP** are manipulated as the stack frame is built, used, and dismantled for a particular of the subroutine

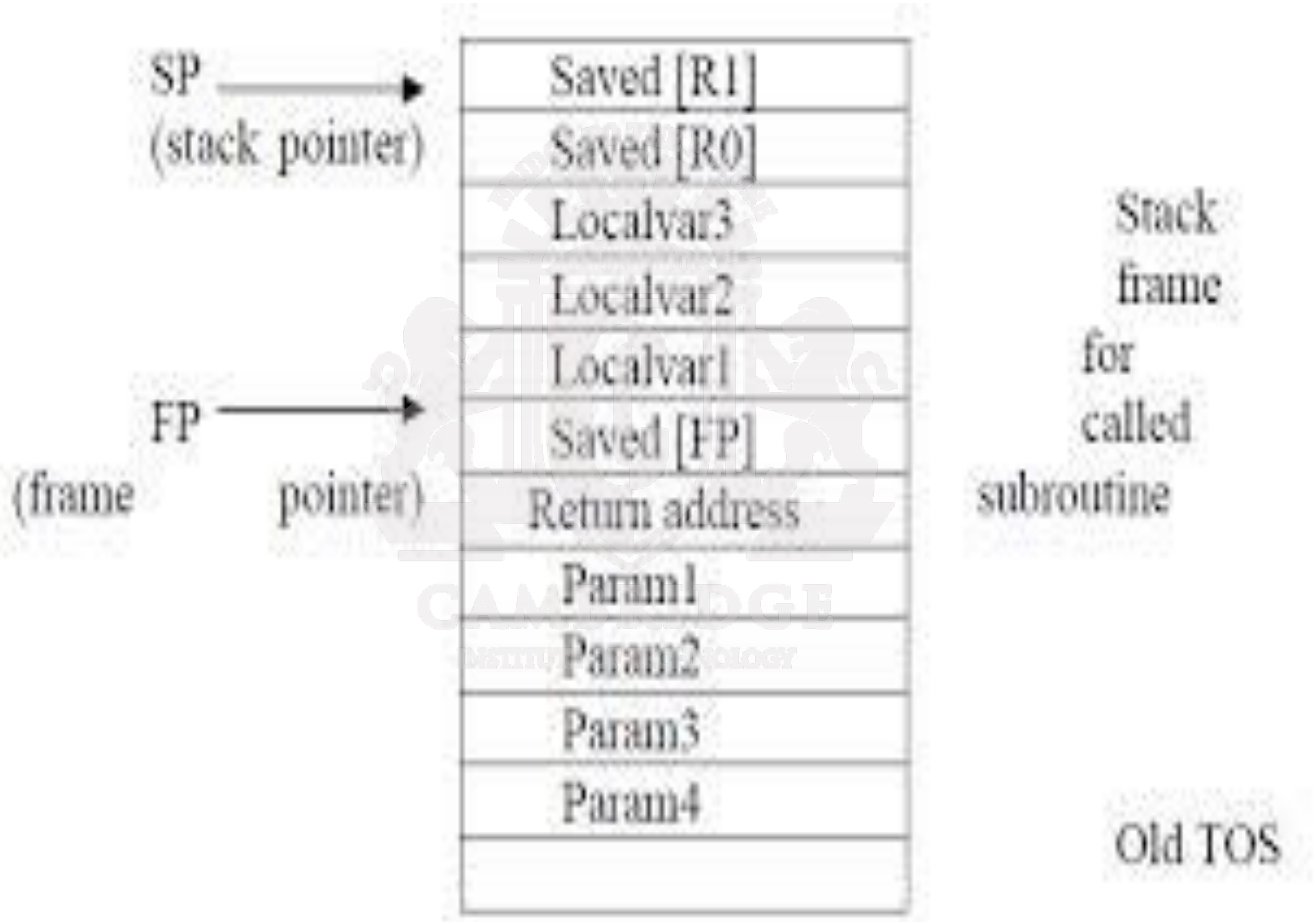
Stack Frame

- We begin by assuming that **SP** point to the old **top-of-stack (TOS)** element in fig b. Before the subroutine is called, the calling program pushes the **four parameters** onto the stack.

- The call instruction is then executed, resulting in the **return address** being pushed onto the stack. Now, **SP** points to this return address, and the first instruction of the subroutine is about to be executed. This is the point at which the frame pointer **FP** is set to contain the proper memory address.

- Since **FP** is usually a general-purpose register, it may contain **information** of use to the Calling program. Therefore, its contents are saved by pushing them onto the stack. Since the SP now points to this position, its contents are copied into FP

Stack Frame



(SOURCE DIGINOTES)

Queues

- Data structure similar to stack
- **First In First Out**
- Queue has two ends :**One entry** and **One exit.**
- Both the ends of a queue move to **higher addresses** as data are added at the back and removed from the front, so **two pointers** are needed to keep track of the queue operations.
- The queue would continuously move through the memory in the direction of higher address: may leads to a "**queue overflow**"
- Solution is to limit the queue to a fixed region in memory by using a **circular buffer.**

Queues

- Consider the memory addresses from **BEGINNING** to **END** are assigned to queue.
- The first element is entered into the location **BEGINNING** and successive entries are appended to the queue by entering them at successively higher addresses.
- By the time the back of the queue reaches **END** some items have been removed from the queue creating empty spaces.
- Hence the back pointer is reset to the value **BEGINNING** and the process continues.

Additional Instructions

LOGICAL INSTRUCTIONS:-

- The basic logic operations are: AND,OR,NOT, and XOR logic gates
- All the programming languages provide instructions to perform these operations on all bits of a byte or word independently.
- For ex:- **Not dst**
- **Complements all bits contained in the destination operand that are changed to 1's and 0's and 0's to 1's**

The **dst** may be a processor register or memory location

Additional Instructions

SHIFT AND ROTATE INSTRUCTIONS:-

- There are many applications that require the bits of an operand to be **shifted right** or **left** some specified number of bit positions.
- The details of how the shifts are performed depend on whether the operand is a signed number or some more general binary-coded information.
- For general operands, we use a **logical shift**. For a signed number, we use an **arithmetic shift**, which preserves the sign of the number

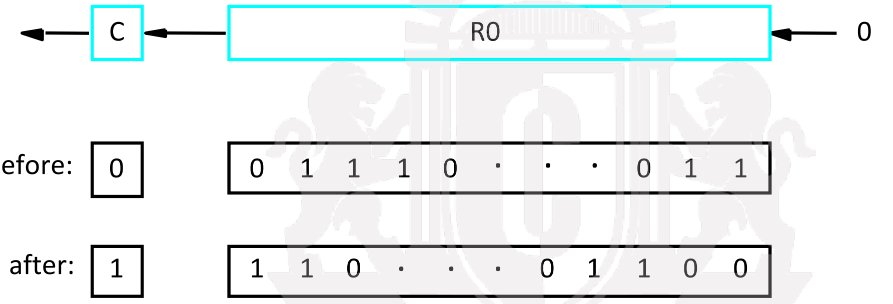
Additional Instructions

- **Logical shifts:-**
- Two logical shift instructions are needed, one for shifting left (LShiftL) and another for shifting right (LShiftR). These instructions shift an operand over a number of bit positions specified in a count operand contained in the instruction.
- The general form of a logical left shift instruction is
-
- LShiftL Count,R0

(a) Logical shift left LShiftL #2, R0

Additional Instructions : Logical Shifts

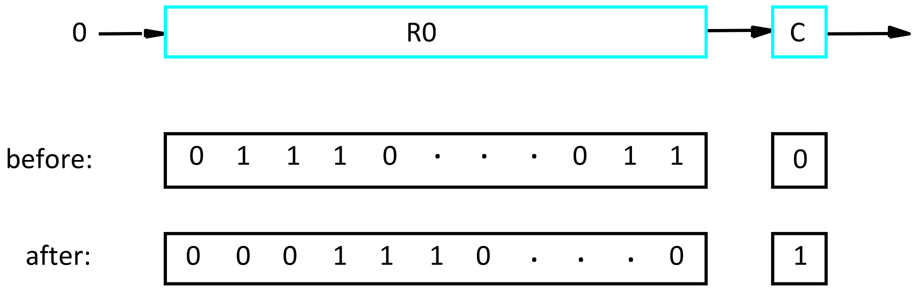
- Logical shift – shifting left (LShiftL) and shifting right (LShiftR)



before: 0 | 0 1 1 1 0 · · · 0 1 1

after: 1 | 1 1 0 · · · 0 1 1 0 0

(a) Logical shift left LShiftL #2,R0



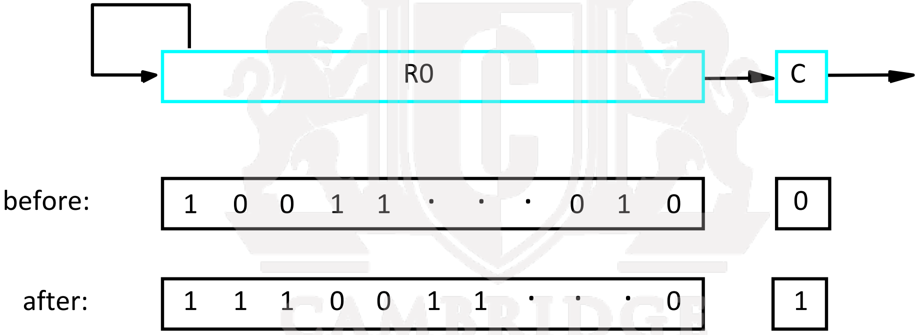
before: 0 1 1 1 0 · · · 0 1 1 | 0

after: 0 0 0 1 1 1 0 · · · 0 | 1

(b) Logical shift r ight LShiftR #2,R0

Arithmetic Shifts

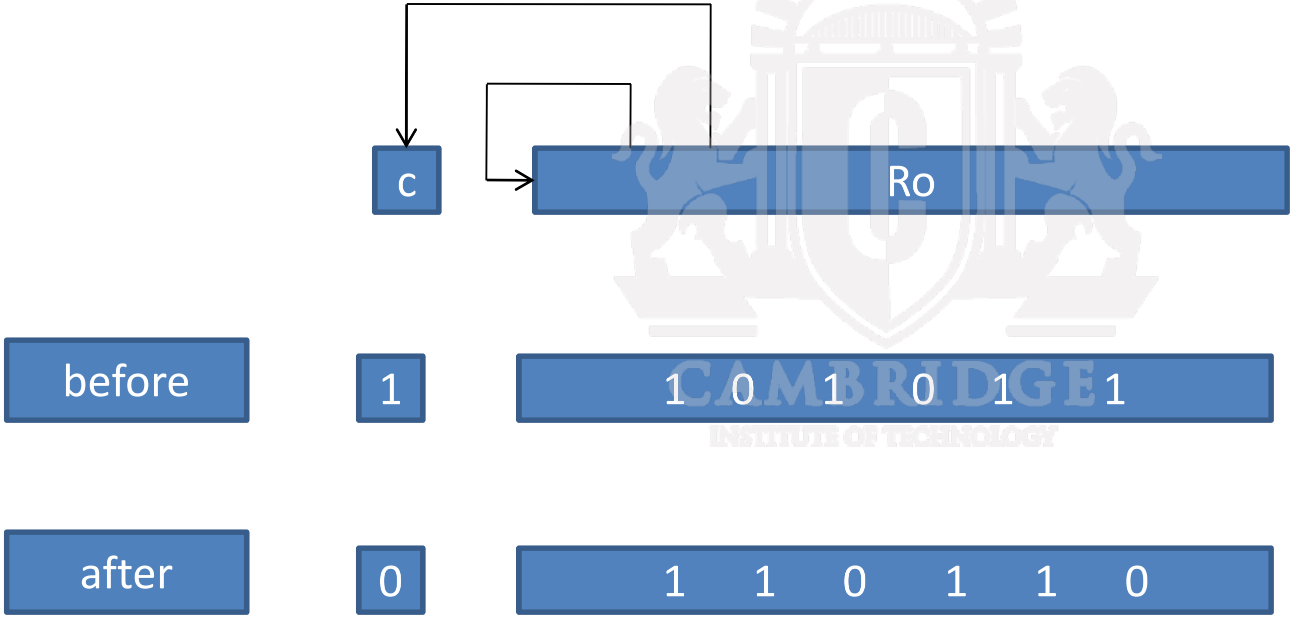
Arithmetic Shift Right :-In this case sign bit is repeated into the bit towards its right and it is also put back in the same position



(c) Ar ithmetic shift right AShiftR #2,R0

Arithmetic Shifts

Arithmetic Shift Left **:-In this case the bit next to the sign bit on the right side is shifted out into the carry bit**



c

Ro

before 1 1 0 1 0 1 1

after 0 1 1 0 1 1 0

AshiftL #1,R0

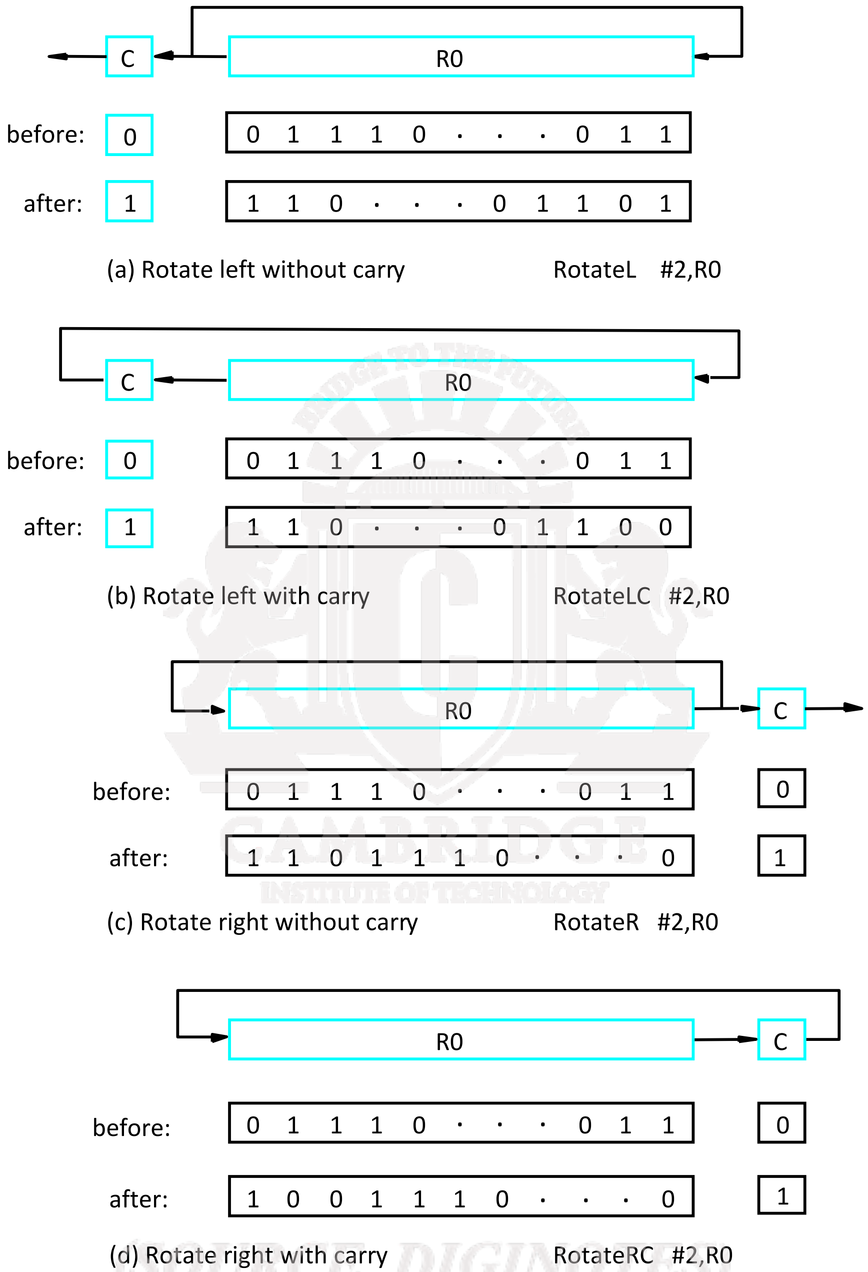
Rotate Instructions

- In the shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is retained in the **Carry flag C.**

- To **preserve all bits**, a set of **rotate instructions** can be used. They **move the bits** that are shifted out of one end of the operand **back into the other end**.

- Two versions of both the left and right rotate are usually provided. In one version, the bits of the operand are **simply rotated**. In the other version, the rotation includes the **C flag** instructions.

- When carry flag is not involved in the rotation it contains the **last bit shifted out of the register.**
- **All four possibilities of rotate instructions are as shown**



(a) Rotate left without carry RotateL #2,R0

(b) Rotate left with carry RotateLC #2,R0

(c) Rotate right without carry RotateR #2,R0

(d) Rotate right with carry RotateRC #2,R0

Figure 2.32. Rotate instructions.

Encoding of Machine Instructions

- To be executed in a processor, an instruction must be encoded in a **compact binary pattern.** Such encoded instructions are properly referred to as machine instructions.
- The instructions that use symbolic names and acronyms are called **assembly language instructions**, which are converted into the **machine instructions** using the **assembler program**.
- We have seen instructions that perform operations such as **add, subtract, move, shift, rotate, and branch.**
- These instructions may use operands of different sizes, such as **32-bit and 8-bit numbers or 8-bit ASCII-encoded characters**.
- The type of operation that is to be performed and the type of operands used may be specified using an encoded binary pattern referred to as the **OP code** for the given instruction.

Encoding of Machine Instructions

- Suppose that **8** bits are allocated for this purpose, giving **256** possibilities for specifying different instructions. This leaves **24** bits to specify the rest of the required information.
- Let us examine some typical cases. The instruction
- **Add R1, R2**
- Has to specify the registers R1 and R2, in addition to the OP code. If the processor has **16 registers**, then **four bits** are needed to identify each register. **Additional bits** are needed to indicate that the **Register addressing mode** is used for each operand.

Encoding of Machine Instructions

- The instruction **Move 24(R0), R5**
- Requires **16** bits to denote the **OP code** and the **two registers**, and some bits to express that the source operand uses the **Index addressing mode** and that the **index value is 24**.
- In all these examples, the instructions can be encoded in a **32-bit** word. Depicts a possible format.
- There is an 8-bit Op-code field and two 7-bit fields for specifying the source and destination operands. The 7-bit field identifies the addressing mode and the register involved (if any). The "Other info" field allows us to specify the additional information that may be needed, such as an index value or an immediate operand.

(a) One-word instruction

Opcode	Source	Dest	Other info

(b) Two-Word instruction

Opcode	Source	Dest	Other info
Memory address/Immediate operand			

(c) Three-operand instruction

Op code	Ri	Rj	Rk	Other info

Encoding of Machine Instructions

- But, what happens if we want to specify a memory operand using the Absolute addressing mode? The instruction
  - **Move R2, LOC**
- Requires 18 bits to denote the OP code, the addressing modes, and the register.
- This leaves 14 bits to express the address that corresponds to LOC, which is clearly insufficient.
  - **And #$FF000000. R2**
- In which case the **second word** gives a full 32-bit immediate operand. If we want to allow an instruction in which two operands can be specified using the Absolute addressing mode, for example

(a) One-word instruction

Opcode	Source	Dest	Other info

(b) Two-Word instruction

Opcode	Source	Dest	Other info
Memory address/Immediate operand			

(c) Three-operand instruction

Op code	Ri	Rj	Rk	Other info

Encoding of Machine Instructions

- If we want to allow an instruction in which two operands can be specified using the Absolute addressing mode, for example
  - **Move LOC1, LOC2**
- Then it becomes necessary to use two additional words for the **32-bit addresses of the operands**.