# Computer Organization (18EC35)

Dr. M. C. Hanumantharaju
Professor
Department of Electronics and Communication Engineering
BMS Institute of Technology and Management
Bengaluru - 560064
(mchanumantharaju@bmsit.in)

# Module-1.
# Basic Structure of Computers, Machine Instructions and Programs

# Text and Reference Books

- **Text Books:**
  - Carl Hamacher, Zvonko Vranesic, Safwat Zaky: Computer Organization, 5th Edition, Tata McGraw Hill, 2002.
  - Carl Hamacher, Zvonko Vranesic, Safwat Zaky, Naraig Manjikian : Computer Organization and Embedded Systems, 6th Edition, Tata McGraw Hill, 2012.
- **Reference Books:**
  - William Stallings: Computer Organization & Architecture, 9th Edition, Pearson, 2015.

# The Computer Revolution

- Progress in computer technology
  - Underpinned by Moore’s Law
- Makes novel applications feasible
  - Computers in automobiles
  - Cell phones
  - Human genome project
  - World Wide Web
  - Search Engines
- Computers are universal

# Classes of Computers

- Desktop/laptop computers
  - General purpose, variety of software
  - Subject to cost/performance tradeoff
- Workstations
  - More computing power used in engg. applications, graphics etc.
- Enterprise System/ Mainframes
  - Used for business data processing
- Server computers (Low End Range)
  - Network based
  - High capacity, performance, reliability
  - Range from small servers to building sized
- Supercomputer (High End Range)
  - Large scale numerical calculation such as weather forecasting, aircraft design
- Embedded computers
  - Hidden as components of systems
  - Stringent power/performance/cost constraints

# What You Will Learn

- How programs are translated into the machine language
  - And how the hardware executes them
- The hardware/software interface
- What determines program performance
  - And how it can be improved
- How hardware designers improve performance

# Understanding Performance

- Algorithm
  - Determines number of operations executed
- Programming language, compiler, architecture
  - Determine number of machine instructions executed per operation
- Processor and memory system
  - Determine how fast instructions are executed
- I/O system (including OS)
  - Determines how fast I/O operations are executed

# Functional Units

# Functional Units

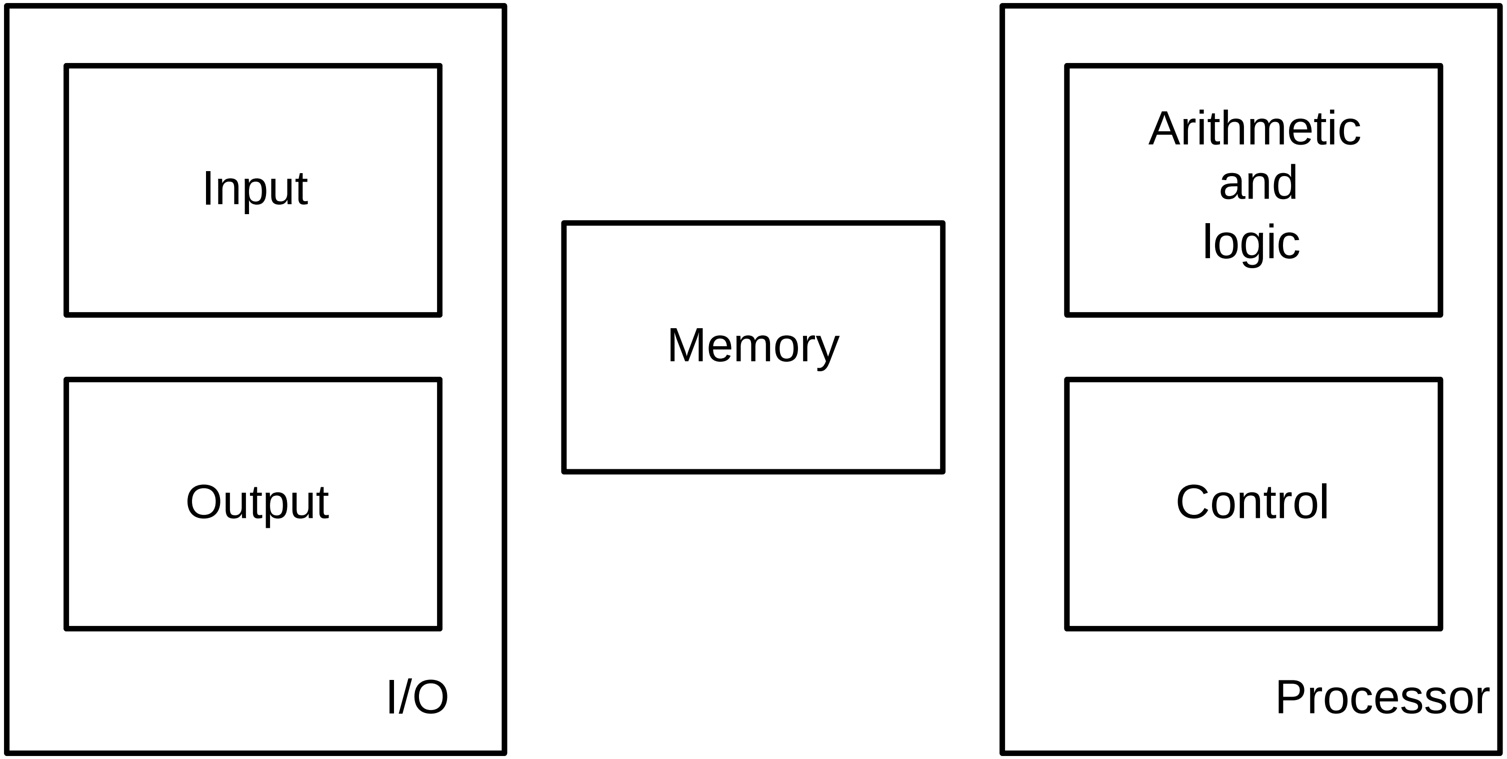


Figure 1.1.  Basic functional units of a computer.

# Information Handled by a Computer

- Instructions/machine instructions
  - Govern the transfer of information within a computer as well as between the computer and its I/O devices
  - Specify the arithmetic and logic operations to be performed
  - Program
- Data
  - Used as operands by the instructions
  - Source program
- Encoded in binary code – 0 and 1

# Memory Unit

- Store programs and data
- Two classes of storage
  - Primary storage
    - Fast
    - Programs must be stored in memory while they are being executed
    - Large number of semiconductor storage cells
    - Processed in words
    - Address
    - RAM and memory access time
    - Memory hierarchy – cache, main memory
  - Secondary storage – larger and cheaper

# Arithmetic and Logic Unit (ALU)

- Most computer operations are executed in ALU of the processor.
  - – Load the operands into memory
  - – bring them to the processor
  - – perform operation in ALU
  - – store the result back to memory or retain in the processor.
- Registers
- Fast control of ALU

# Control Unit

- All computer operations are controlled by the control unit.
- The timing signals that govern the I/O transfers are also generated by the control unit.
- Control unit is usually distributed throughout the machine instead of standing alone.
- Operations of a computer:
  - Accept information in the form of programs and data through an input unit and store it in the memory
  - Fetch the information stored in the memory, under program control, into an ALU, where the information is processed
  - Output the processed information through an output unit
  - Control all activities inside the machine through a control unit

# The operations of a computer

- The computer accepts information in the form of programs and data through an input unit and stores it in the memory.
- Information stored in the memory is fetched under program control into an arithmetic and logic unit, where it is processed.
- Processed information leaves the computer through an output unit.
- All activities in the computer are directed by the control unit.

# Basic Operational Concepts

# Review

- Activity in a computer is governed by instructions.
- To perform a task, an appropriate program consisting of a list of instructions is stored in the memory.
- Individual instructions are brought from the memory into the processor, which executes the specified operations.
- Data to be used as operands are also stored in the memory.

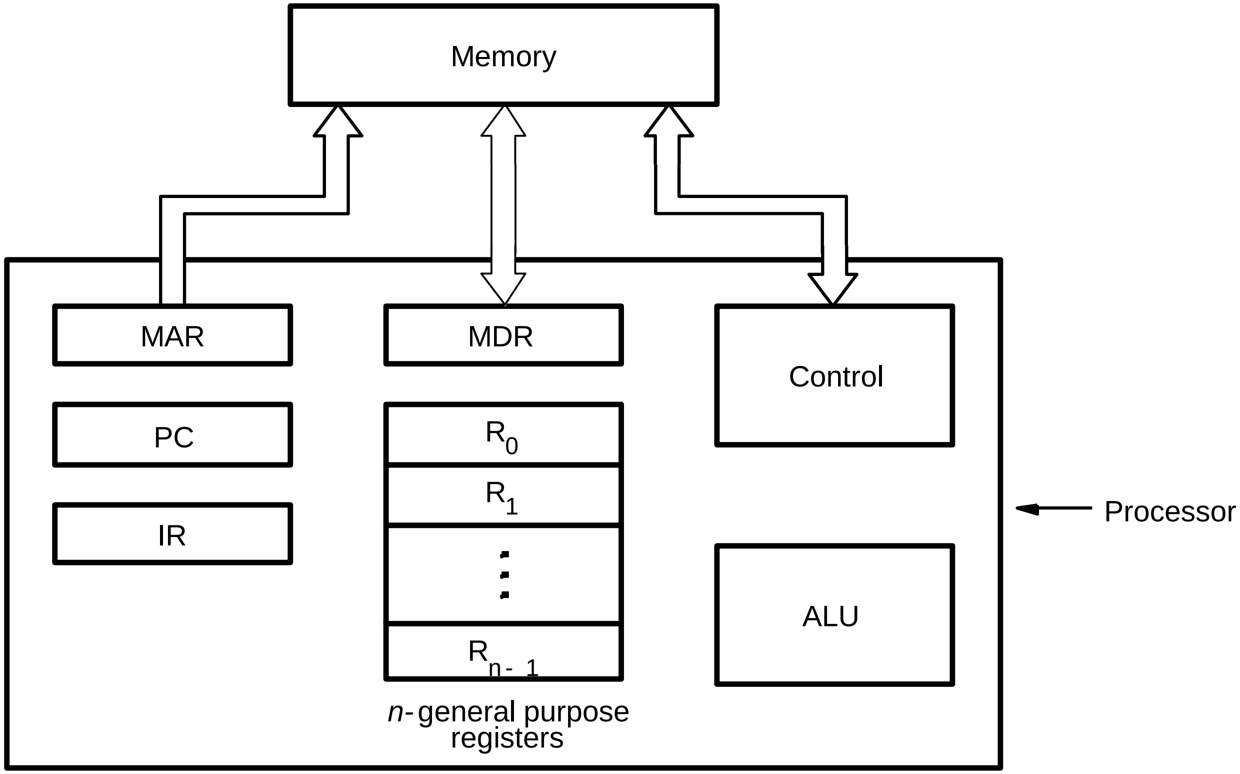
# A Typical Instruction

- 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



Connections between the processor and the memory.

# Registers

- Instruction register (IR)
- Program counter (PC)
- General-purpose register ($R_0$ – $R_{n-1}$)
- Memory address register (MAR)
- Memory data register (MDR)

# 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

# Typical Operating Steps (Cont’)

- Get operands for ALU
  - General-purpose register
  - Memory (address to MAR – Read – MDR to ALU)
- Perform operation in ALU
- Store the result back
  - To general-purpose register
  - To memory (address to MAR, result to MDR – Write)
- During the execution, PC is incremented to the next instruction

# Interrupt

- Normal execution of programs may be preempted if some device requires urgent servicing.
- The normal execution of the current program must be interrupted – the device raises an *interrupt* signal.
- Interrupt-service routine
- Current system information backup and restore (PC, general-purpose registers, control information, specific information)

# 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

# Bus Structure

- Single-bus

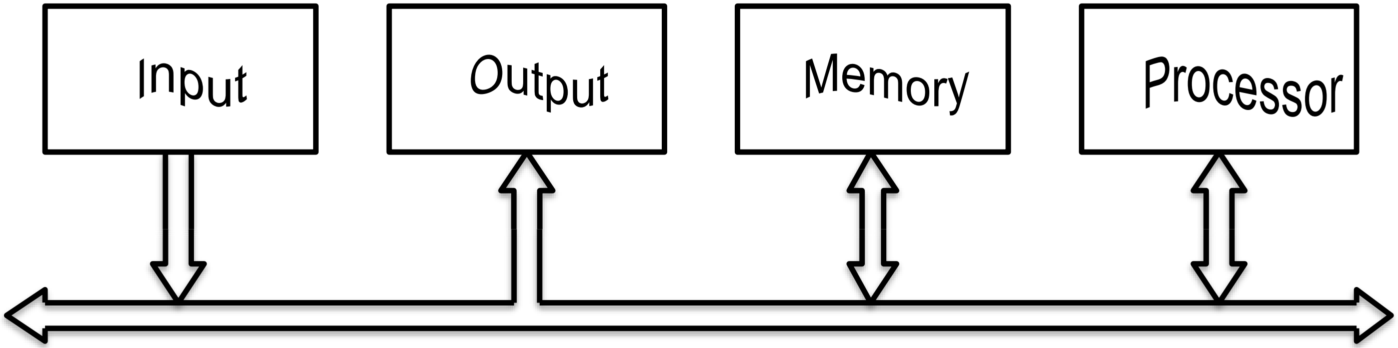


Figure 1.3. Single-bus structure.

- Multiple Buses

# Speed Issue

- Different devices have different transfer/operate speed.
- If the speed of bus is bounded by the slowest device connected to it, the efficiency will be very low.
- How to solve this?
- A common approach – use buffers.

  e.g.- Printing the characters

# Performance

# Performance

- The most important measure of a computer is how quickly it can execute programs.
- Three factors affect performance:
  - Hardware design
  - Instruction set
  - Compiler

# Performance

- Processor time to execute a program depends on the hardware involved in the execution of individual machine instructions.

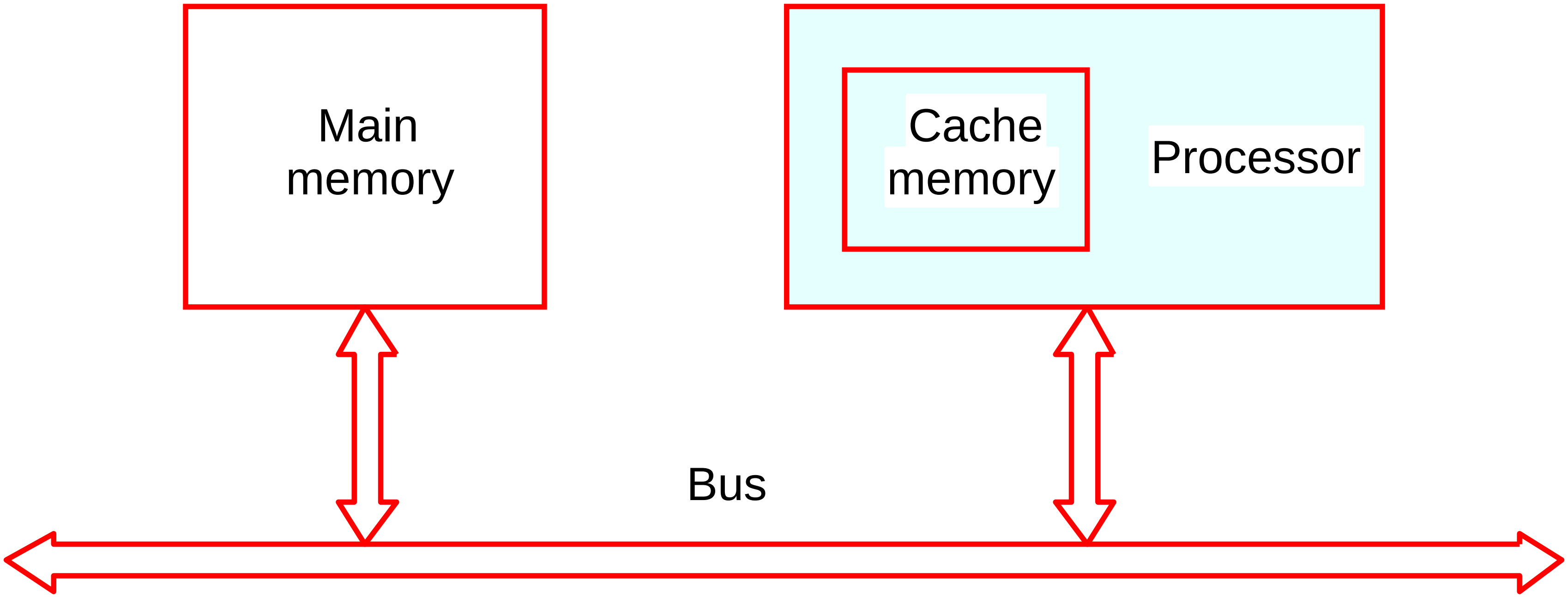


Figure 1.5. The processor cache.

# Performance

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

# Processor Clock

- Clock, clock cycle (P), and clock rate (R=1/P)
- The execution of each instruction is divided into several steps (Basic Steps), each of which completes in one clock cycle.
- Hertz – cycles per second

# Basic Performance Equation

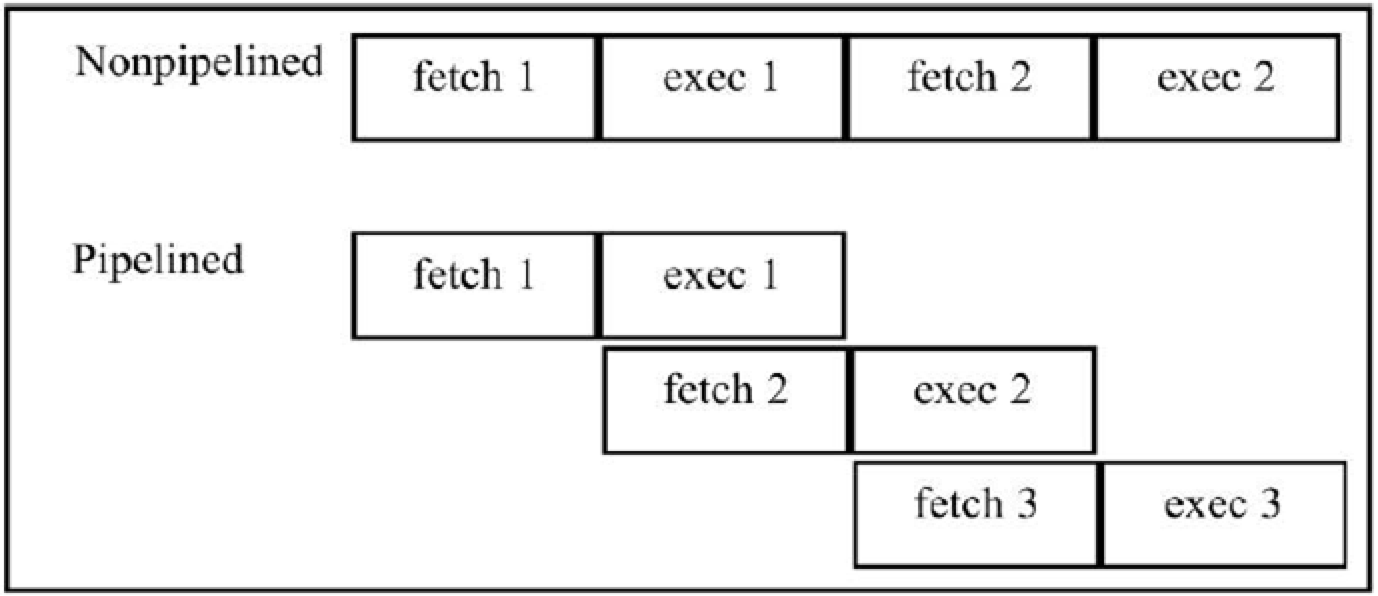
- T – processor time required to execute a program that has been prepared in high-level language
- N – number of actual machine language instructions needed to complete the execution (note: loop)
- S – average number of basic steps needed to execute one machine instruction. Each step completes in one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

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

- How to improve T?
- Reduce N and S, Increase R, but these affect one another

# Pipeline and Superscalar Operation

- Instructions are not necessarily executed one after another.
- The value of S doesn't have to be the number of clock cycles to execute one instruction.
- Pipelining – overlapping the execution of successive instructions.
- Add R1, R2, R3 at the same time processor reads next instruction in memory.



# Pipeline and Superscalar Operation

- Superscalar operation – multiple instruction pipelines are implemented in the processor.
- Goal – reduce S (could become <1!)

# Clock Rate

- Increase clock rate
  - Improve the integrated-circuit (IC) technology to make the circuits faster
  - Reduce the amount of processing done in one basic step (however, this may increase the number of basic steps needed)
- Increases in R that are entirely caused by improvements in IC technology affect all aspects of the processor’s operation equally except the time to access the main memory.

# CISC and RISC

- Tradeoff between N and S
- A key consideration is the use of pipelining
  - S is close to 1 even though the number of basic steps per instruction may be considerably larger
  - It is much easier to implement efficient pipelining in processor with simple instruction sets
- Reduced Instruction Set Computers (RISC) (Large value N , Small Value of S)
- Complex Instruction Set Computers (CISC) (Small value N , Large Value of S)

# Compiler

- A compiler translates a high-level language program into a sequence of machine instructions.
- To reduce N, we need a suitable machine instruction set and a compiler that makes good use of it.
- Goal – reduce N×S
- A compiler may not be designed for a specific processor; however, a high-quality compiler is usually designed for, and with, a specific processor.

# Performance Measurement

- T is difficult to compute.
- Measure computer performance using benchmark programs.
- System Performance Evaluation Corporation (SPEC) selects and publishes representative application programs for different application domains, together with test results for many commercially available computers.
- Compile and run (no simulation)
- Reference computer

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

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

- n is the number of program in the suite

# Multiprocessors and Multicomputers

- Multiprocessor computer
  - Execute a number of different application tasks in parallel
  - Execute subtasks of a single large task in parallel
  - All processors have access to all of the memory – shared-memory multiprocessor
  - Cost – processors, memory units, complex interconnection networks
- Multicomputers
  - Each computer only have access to its own memory
  - Exchange message via a communication network – message-passing multicomputers

# Machine Instructions and Programs

# Objectives

- Machine instructions and program execution, including branching and subroutine call and return operations.
- Addressing methods for accessing register and memory operands.
- Assembly language for representing machine instructions, data, and programs.
- Program-controlled Input/Output operations.

# Memory Locations, Addresses, and 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.

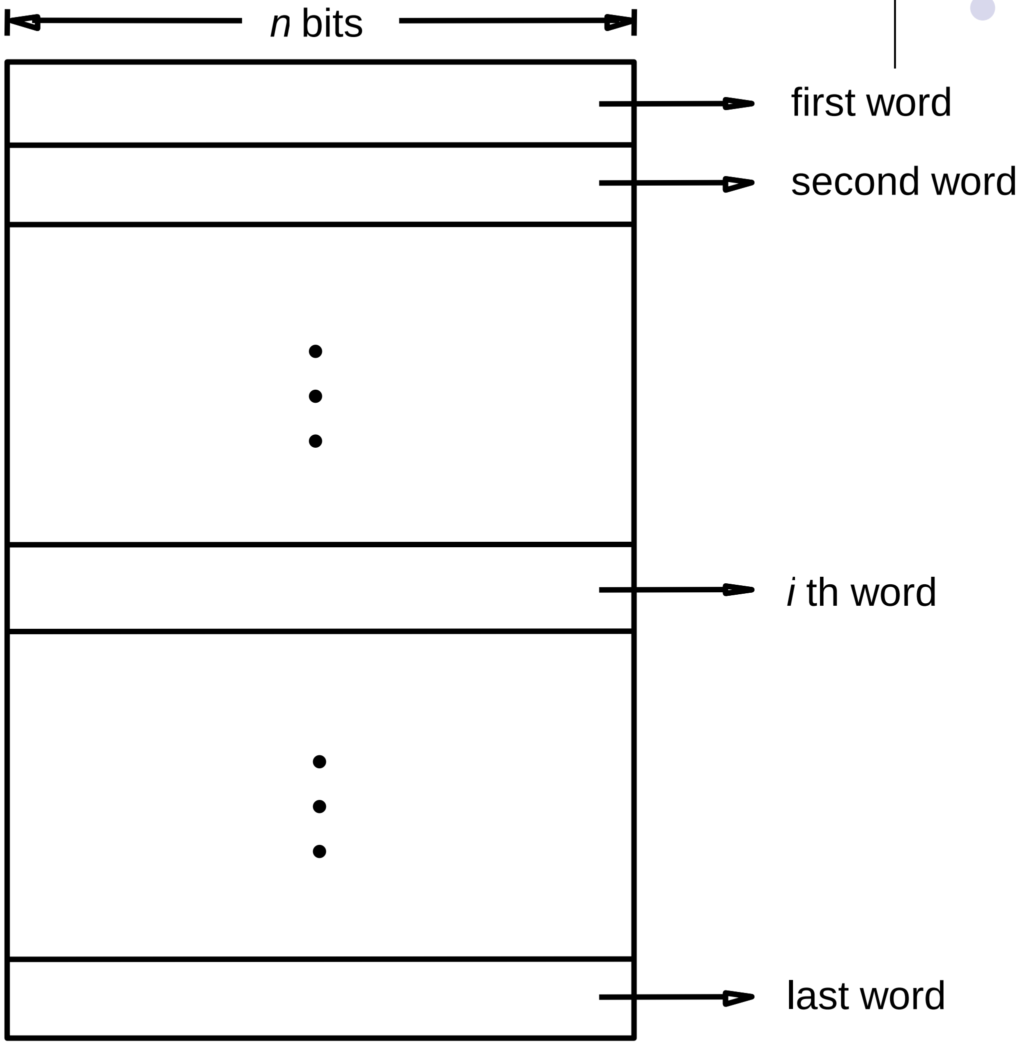
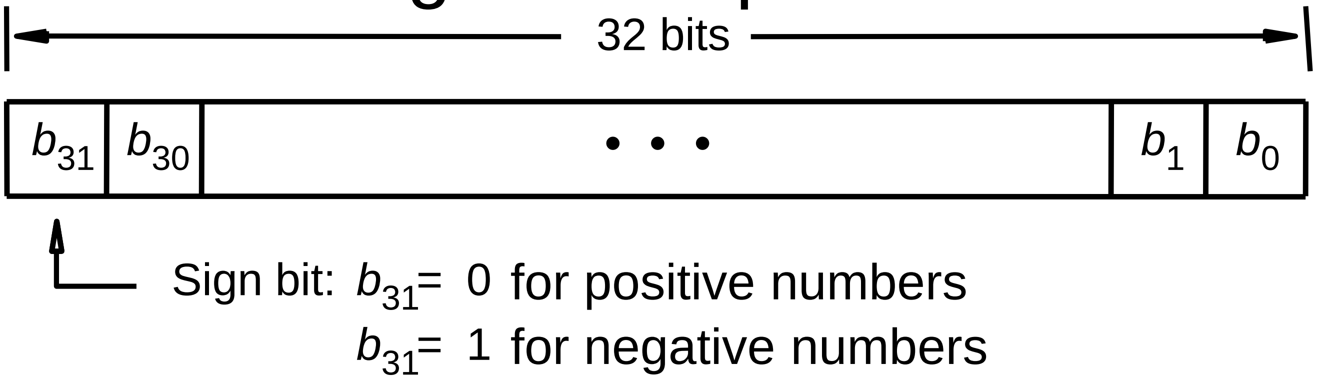


Fig: Memory words.

# Memory Location, Addresses, and Operation

- 32-bit word length example

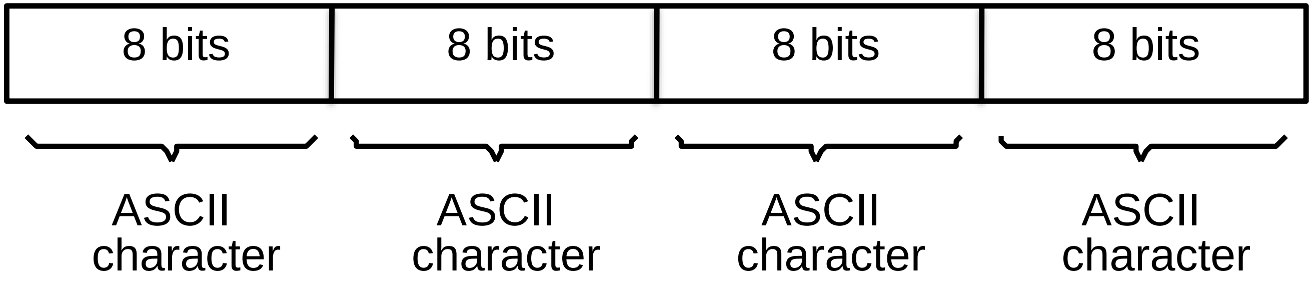


32 bits

$b_{31}$ $b_{30}$ • • • $b_1$ $b_0$

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)=$2^{40}$

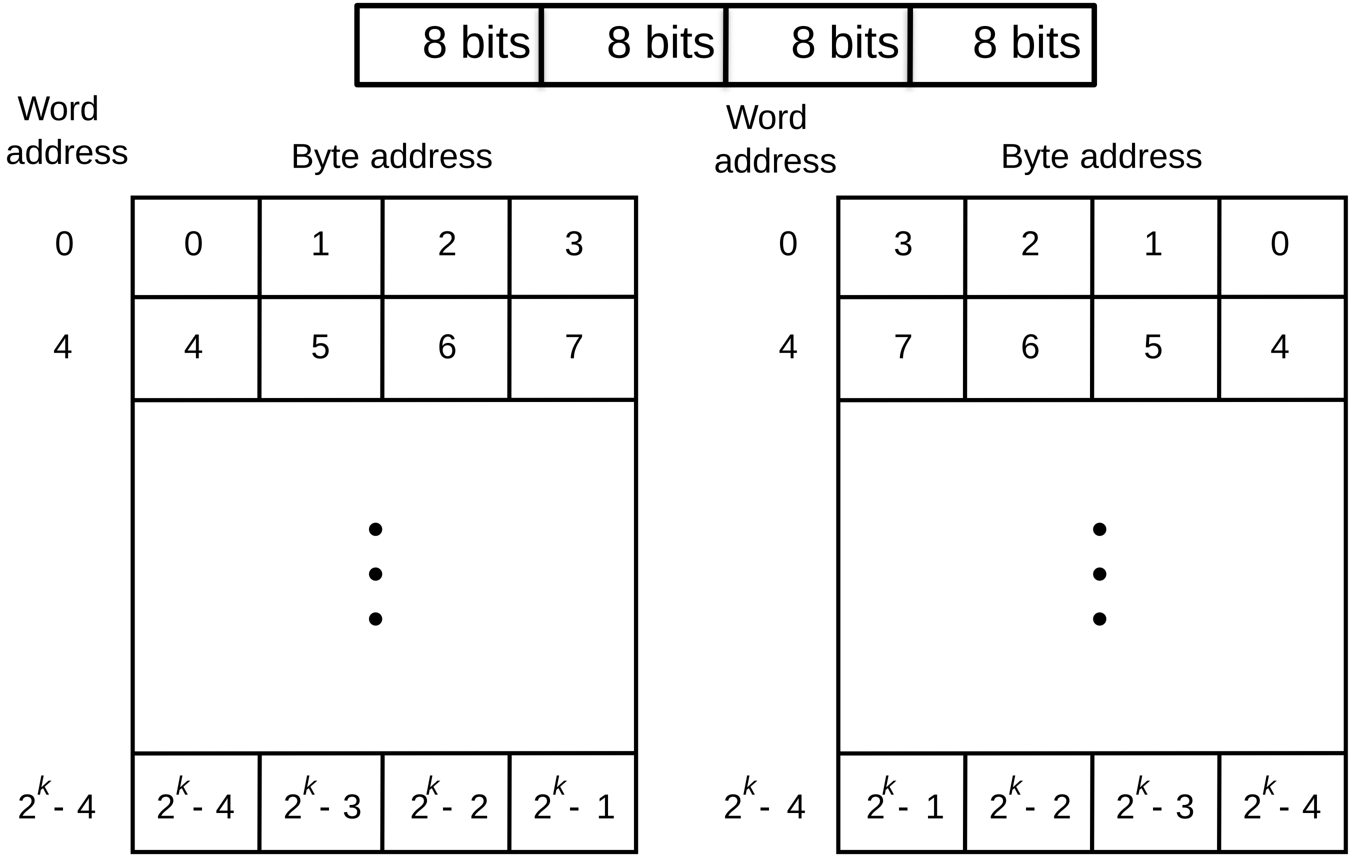
# Memory Location, Addresses, and Operation

- 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



| 8 bits | 8 bits | 8 bits | 8 bits |
|---|---|---|---|

| 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

| Word address | 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

# Instruction and Instruction Sequencing

# “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 R2, R3, R1 R1 ← [R2] + [R]3
- Two-Address Instructions
  - ADD R2, R1 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 Formats

Example: Evaluate (A+B) ∗ (C+D)

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

# Instruction Formats

Example: Evaluate (A+B) ∗ (C+D)

- Two-Address

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

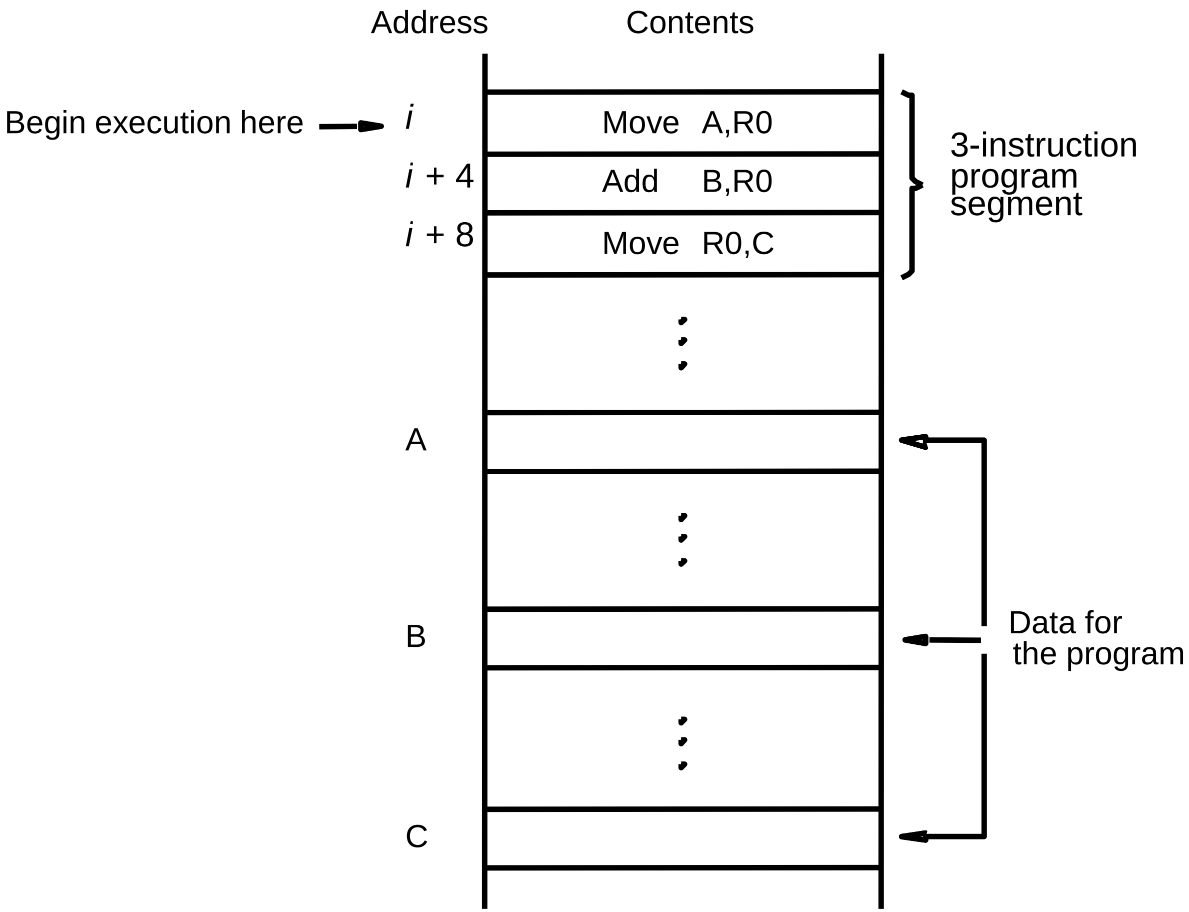


Figure 2.8. A program for C ← [A] + [B].

Assumptions:

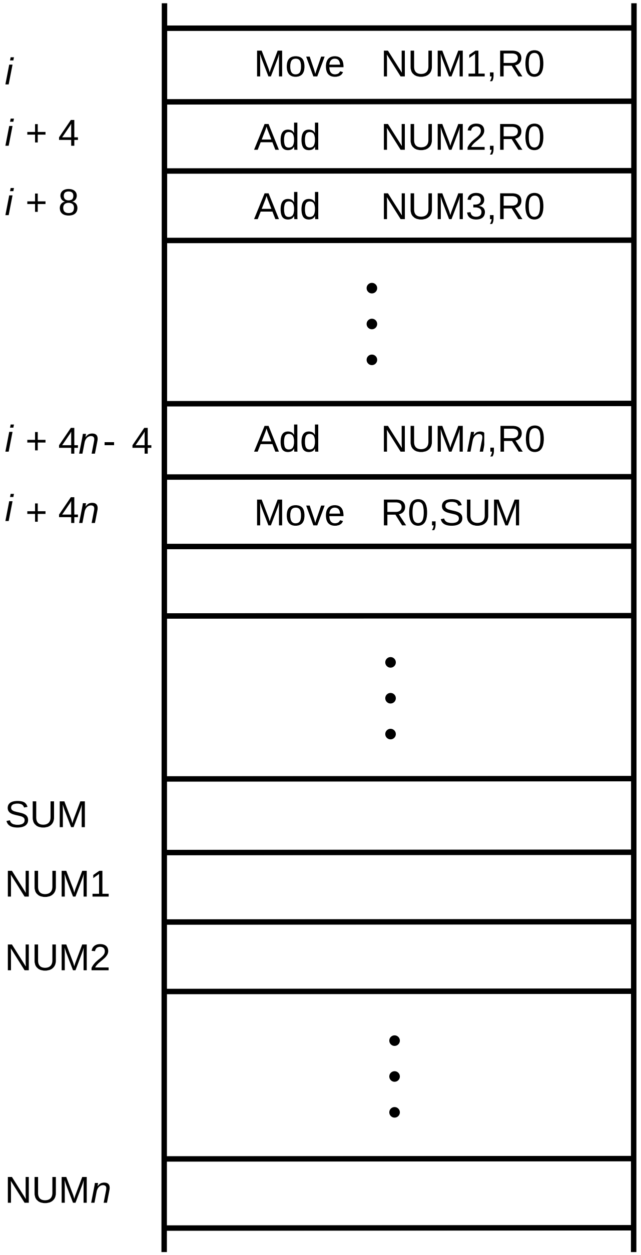
- One memory operand per instruction
- 32-bit word length
- Memory is byte addressable
- Full memory address can be directly specified in a single-word instruction

Two-phase procedure

-Instruction fetch

-Instruction execute

# Branching

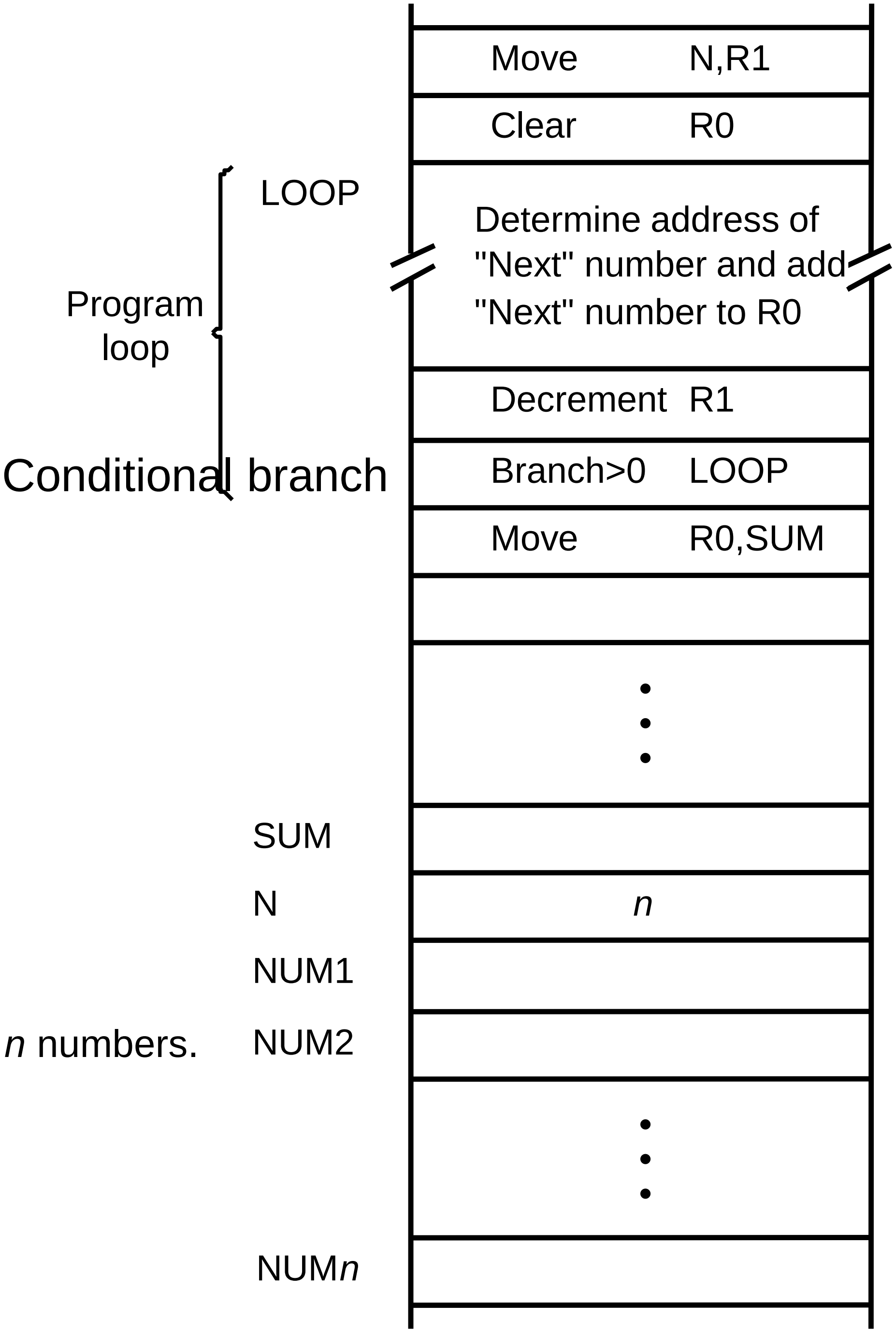


| | |
|---|---|
| *i* | Move NUM1,R0 |
| *i* + 4 | Add NUM2,R0 |
| *i* + 8 | Add NUM3,R0 |
| | ⋮ |
| *i* + 4*n* - 4 | Add NUM*n*,R0 |
| *i* + 4*n* | Move R0,SUM |
| | |
| | ⋮ |
| SUM | |
| NUM1 | |
| NUM2 | |
| | ⋮ |
| NUM*n* | |

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

# Branching

Branch target



Move N,R1

Clear R0

LOOP

Program loop

Determine address of "Next" number and add "Next" number to R0

Decrement R1

Conditional branch

Branch>0 LOOP

Move R0,SUM

SUM

N *n*

NUM1

NUM2

NUM*n*

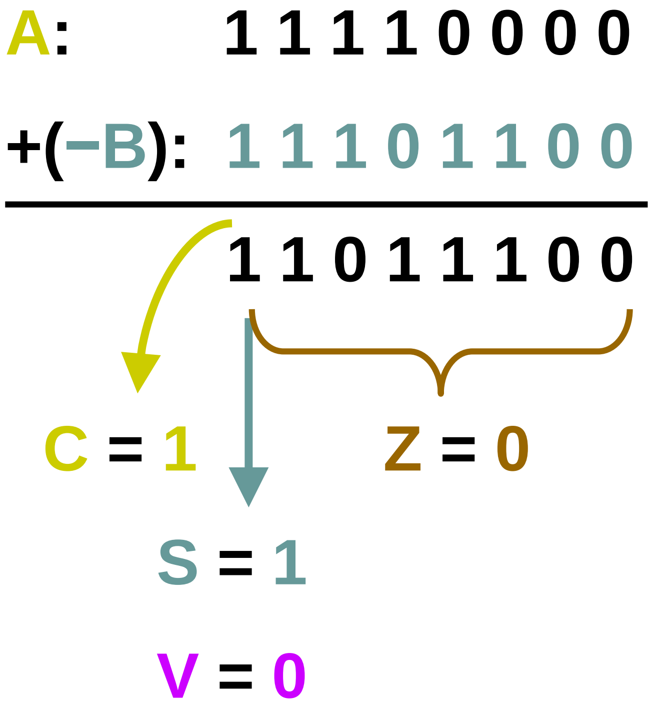
Figure 2.10. Using a loop to add *n* numbers.

# Condition Codes

- Condition code flags (bits)
- 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

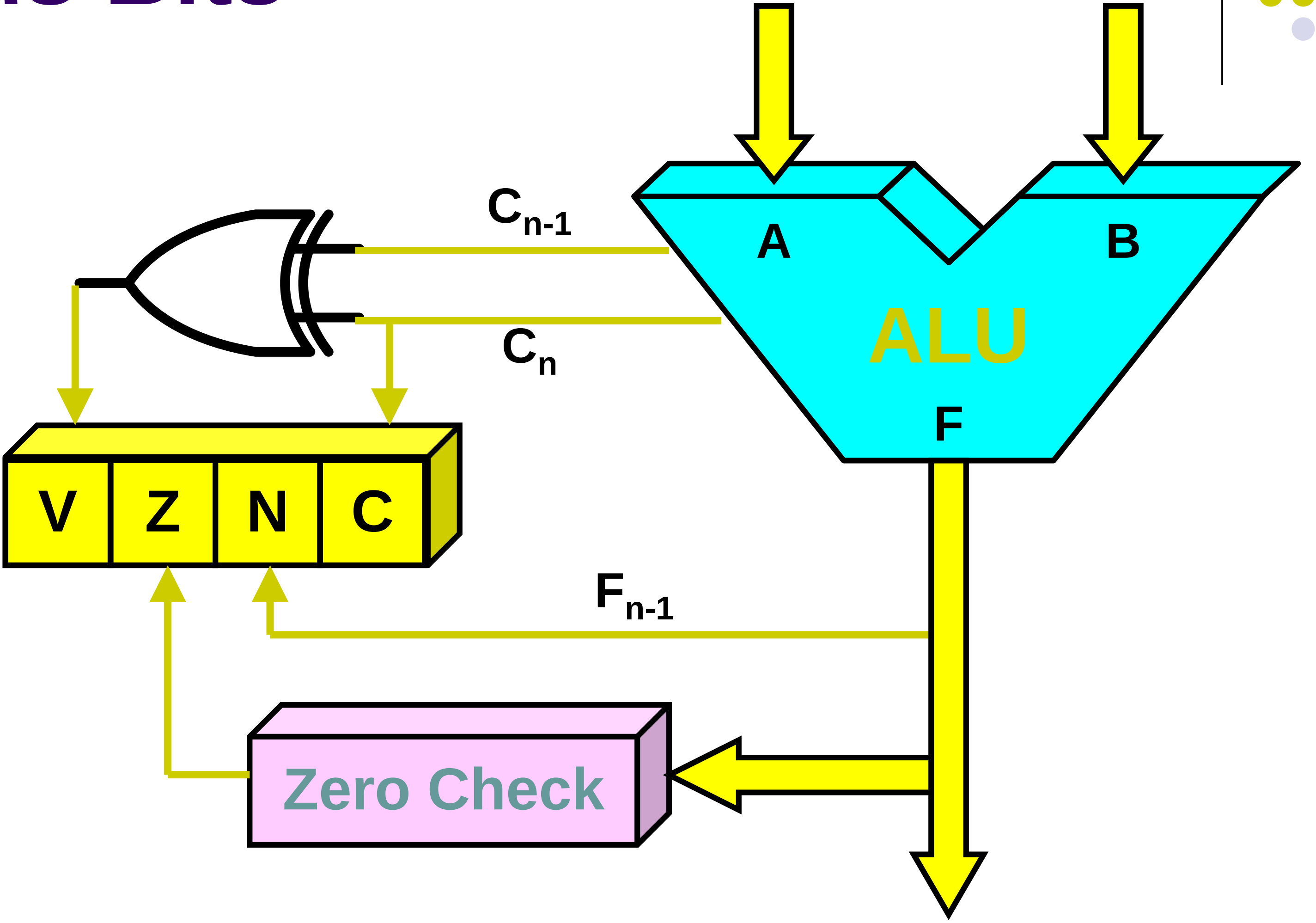
C = 1

Z = 0

S = 1

V = 0

# Status Bits



Module - 2

# Addressing Modes

# Generating Memory Addresses

- How to specify the address of branch target?
- Can we give the memory operand address directly in a single Add instruction in the loop?
- Use a register to hold the address of NUM1; then increment by 4 on each pass through the loop.

# Addressing Modes

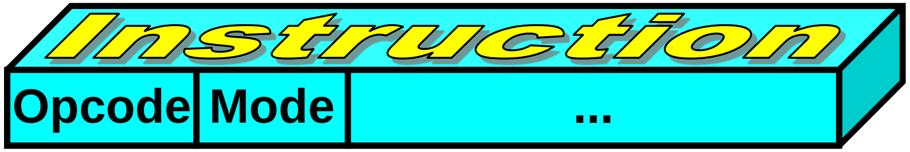
- The different ways in which the location of an operand is specified in an instruction are referred to as addressing modes.

| Name | Assembler syntax | Addressingfunction |
|---|---|---|
| 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 |
| Basewith index | (R*i*,R*j*) | EA = [R*i*] + [R*j*] |
| Basewith 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*] |

# Effective Address (EA)

- In the addressing modes that follow, the instruction does not give the operand or its address explicitly. Instead, it provides information from which an *effective address* (EA) can be derived by the processor when the instruction is executed.
- The effective address is then used to access the operand.

# Addressing Modes



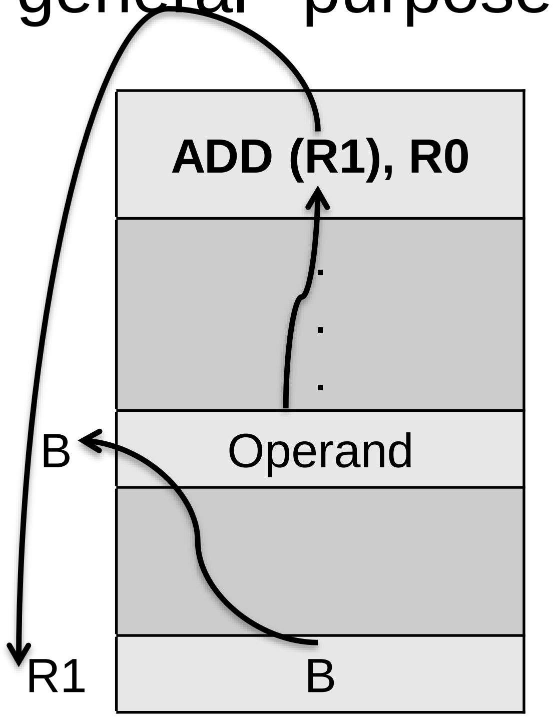
- Implied
  - AC is implied in “ADD   M[AR]” in “One-Address” instr.
  - TOS is implied in “ADD” in “Zero-Address” instr.
- Immediate
  - The use of a constant in “MOV   5, R1”
    or “MOV  #5, R1”             i.e. R1 ← 5

  MOV #NUM1, R2 ; to copy the variable memory address
- Register
  - Indicate which register holds the operand
- Direct Address
  - Use the given address to access a memory location
  - E.g. Move NUM1, R1
  -          Move R0, SUM

# Addressing Modes
# Indirect Addressing

- Indirect Addressing
  - Indirection and Pointer
  - Indirect addressing through a general purpose register.
  - Indicate the register (e.g. R1) that holds the address of the variable (e.g. B) that holds the operand

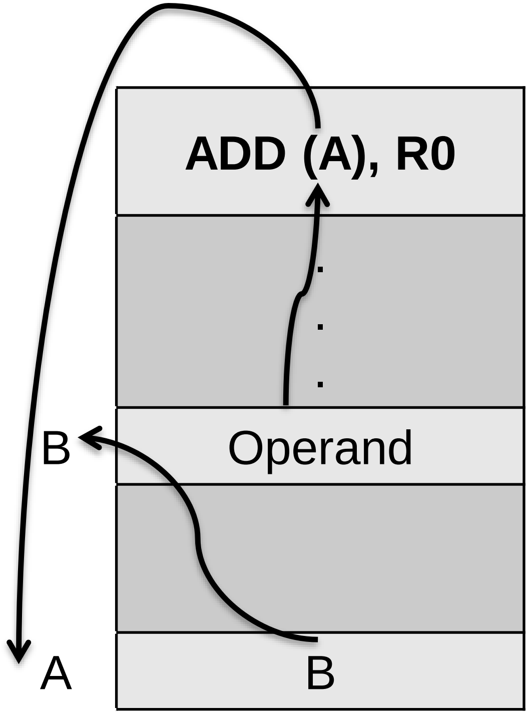
    ADD (R1), R0
  - The register or memory location that contain the address of an operand is called a pointer



# Addressing Modes
# Indirect Addressing

- Indirect Addressing
  - Indirect addressing through a memory addressing.
  - Indicate the memory variable (e.g. A )that holds the address of the variable (e.g. B) that holds the operand

    ADD (A), R0



# Indirect Addressing Example

- Addition of N numbers

```
              Move N,R1            ; N = Numbers to add
              Move #NUM1,R2        ; R2= Address of 1st no.
              Clear R0             ; R0 = 00
       Loop : Add (R2), R0         ; R0 = [NUM1] + [R0]
              Add #4, R2      ; R2= To point to the  next
                              ; number
              Decrement R1         ; R1 = [R1] -1
              Branch>0 Loop        ; Check if R1>0 or not if
                                   ; yes go to Loop
               Move R0, SUM        ; SUM= Sum of all no.
```

# Example

- Addition of N numbers

```
1.             Move N,R1          ; N = 5
2.             Move #NUM1,R2      ; R2= 10000H
3.             Clear R0           ; R0 = 00
4.     Loop : Add (R2), R0        ; R0 = 10 + 00 = 10
5.             Add #4, R2         ; R2 = 10004H
6.             Decrement R1       ; R1 = 4
7.             Branch>0 Loop      ; Check if R1>0 if
                                  ; yes go to Loop
8.              Move R0, SUM      ; SUM=
```

# Example

- Addition of N numbers

```
        Move N,R1          ; N = 5
        Move #NUM1,R2      ; R2= 10000H
        Clear R0           ; R0 = 00
Loop :  Add (R2), R0       ; R0 = 20 + 10 = 30
        Add #4, R2         ; R2 = 10008H
        Decrement R1       ; R1 = 3
        Branch>0 Loop      ; Check if R1>0 if
                           ; yes go to Loop
         Move R0, SUM      ; SUM=
```

# Example

- Addition of N numbers

```
1.          Move N,R1          ; N = 5
2.          Move #NUM1,R2      ; R2= 10000H
3.          Clear R0           ; R0 = 00
4.   Loop : Add (R2), R0       ; R0 = 30 + 30 = 60
5.          Add #4, R2         ; R2 = 1000CH
6.          Decrement R1       ; R1 = 2
7.          Branch>0 Loop      ; Check if R1>0 if
                               ; yes go to Loop
8.           Move R0, SUM      ; SUM=
```

# Example

- Addition of N numbers

```
             Move N,R1            ; N = 5
             Move #NUM1,R2        ; R2= 10000H
             Clear R0             ; R0 = 00
     Loop :  Add (R2), R0         ; R0 = 40 + 60 = 100
             Add #4, R2           ; R2 = 10010H
             Decrement R1         ; R1 = 1
             Branch>0 Loop        ; Check if R1>0 if
                                  ; yes go to Loop
              Move R0, SUM        ; SUM=
```

# Example

- Addition of N numbers

```
              Move N,R1          ; N = 5
              Move #NUM1,R2      ; R2= 10000H
              Clear R0           ; R0 = 00
       Loop : Add (R2), R0       ; R0 = 50 + 100 = 150
              Add #4, R2         ; R2 = 10014H
              Decrement R1       ; R1 = 0
              Branch>0 Loop      ; Check if R1>0 if
                                 ; yes go to Loop
               Move R0, SUM      ; SUM =
```

# Example

- Addition of N numbers

```
        Move N,R1          ; N = 5
        Move #NUM1,R2      ; R2= 10000H
        Clear R0           ; R0 = 00
Loop :  Add (R2), R0       ; R0 = 50 + 100 = 150
        Add #4, R2         ; R2 = 10014H
        Decrement R1       ; R1 = 0
        Branch>0 Loop      ; Check if R1>0 if
                           ; yes go to Loop
         Move R0, SUM      ; SUM = 150
```

# Addressing Modes
# Indexing and Arrays

- Indexing and Array
- The EA of the operand is generated by adding a constant value to the contents of a register.
- X(Ri)     ; EA= X + (Ri)   X= Signed number
- X defined as offset or displacement

# Addressing Modes Indexing and Arrays

- Index mode – the effective address of the operand is generated by adding a constant value to the contents of a register.
- 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 Indexing and Arrays

- In general, the Index mode facilitates access to an operand whose location is defined relative to a reference point within the data structure in which the operand appears.
- 2D Array
  - (Ri, Rj)            so EA =  [Ri] + [Rj]
  - Rj is called the base register
- 3D Array
  - X(Ri, Rj)         so EA =  X + [Ri] + [Rj]

# Addressing Modes Indexing and Arrays

| Address | Memory |
|---|---|
| | Add 20(R1), R2 |
| | .<br>.<br>.<br>. |
| 10000H | |
| Offset=20 | .<br>.<br>.<br>. |
| 10020H | Operand |

| | |
|---|---|
| R1 | 10000H |

Offset is given as a Constant

| Address | Memory |
|---|---|
| | Add 10000H(R1), R2 |
| | .<br>.<br>.<br>. |
| 10000H | |
| Offset=20 | .<br>.<br>.<br>. |
| 10020H | Operand |

| | |
|---|---|
| R1 | 20H |

Offset is in the  index register

# Addressing Modes Indexing and Arrays

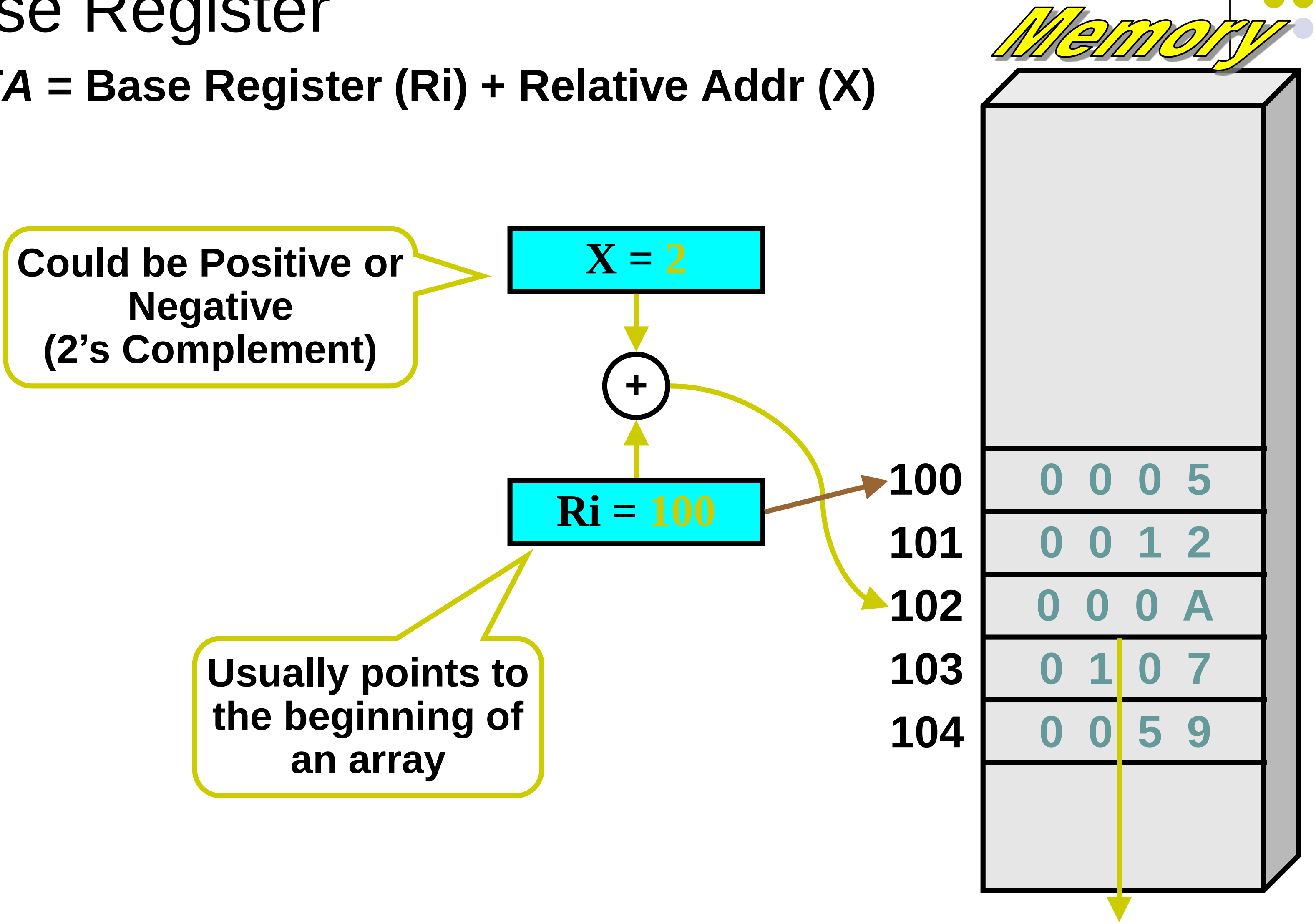
- Array
- E.g. List of students marks

| Address | Memory | Comments |
|---|---|---|
| N | n | No. of students |
| LIST | Student ID1 | Student 1 |
| LIST+4 | Test 1 | |
| LIST+8 | Test 2 | |
| LIST+12 | Test 3 | |
| LIST+16 | Student ID2 | Student 2 |
| LIST+20 | Test 1 | |
| LIST+24 | Test 2 | |
| LIST+28 | Test 3 | |

- Indexed addressing used in accessing test marks from the list

# Addressing Modes

- Base Register
  - *EA* = Base Register (Ri) + Relative Addr (X)



Could be Positive or Negative (2’s Complement)

Usually points to the beginning of an array

| | |
|---|---|
| 100 | 0 0 0 5 |
| 101 | 0 0 1 2 |
| 102 | 0 0 0 A |
| 103 | 0 1 0 7 |
| 104 | 0 0 5 9 |

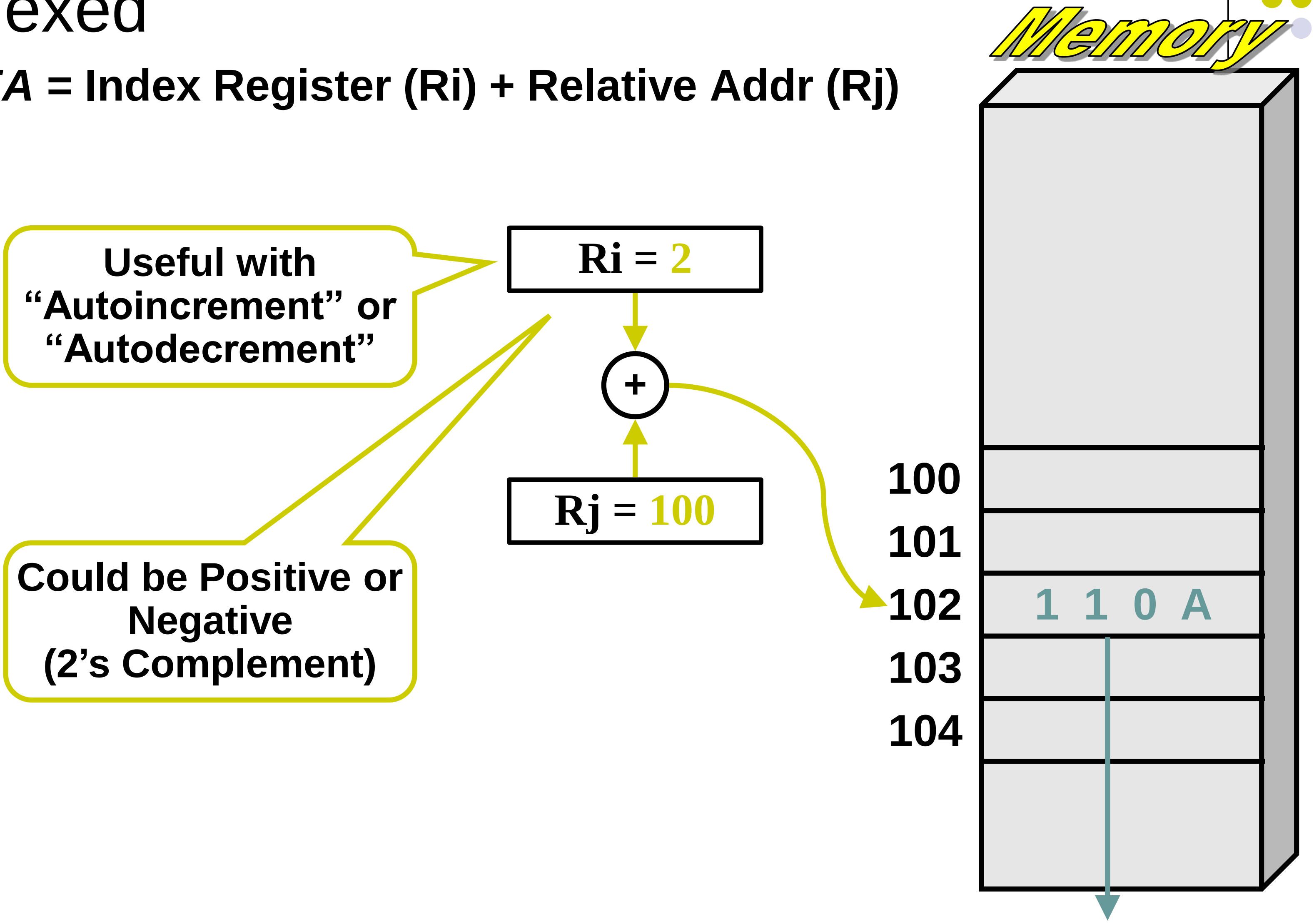
# Addressing Modes Indexing and Arrays

- Program to find the sum of marks of all subjects of reach students and store it in memory.

```
                Move #LIST, R0
                Clear R1
                Clear R2
                Move #SUM, R2
                Move N, R4
Loop :          Add 4(R0), R1
                Add 8(R0), R1
                Add 12(R0),R1
                Move R1, (R2)
                Clear R1
                Add #16, R0
                Add #4, R2
                Decrement R4
                Branch>0 Loop
```

# Addressing Modes

- Indexed
  - *EA* = Index Register (Ri) + Relative Addr (Rj)

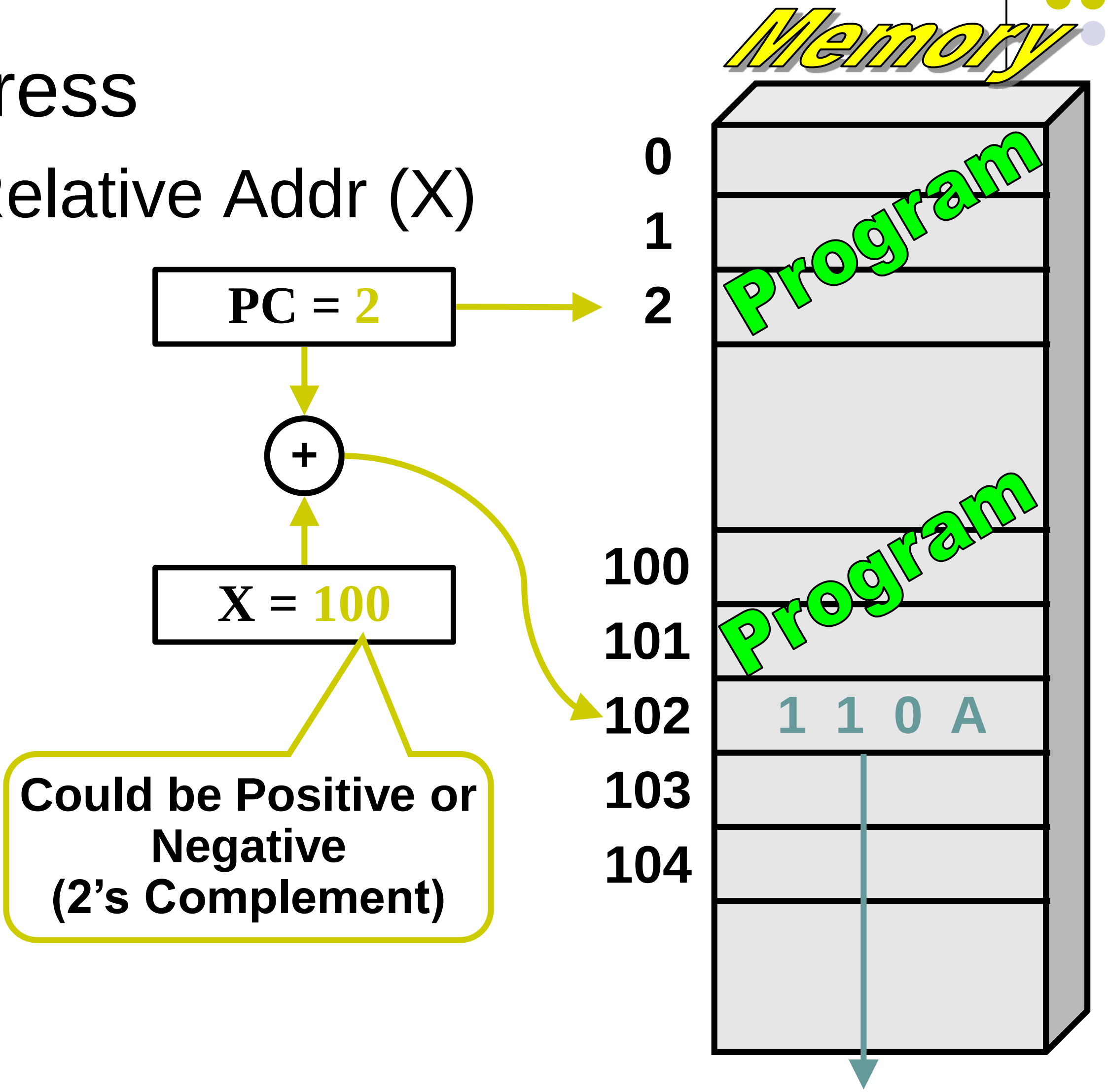


# Addressing Modes
# Relative Addressing

- Relative mode – the effective address is determined by the Index mode using the program counter in place of the general-purpose register.
- X(PC) – note that X is a signed number
- Branch>0        LOOP
- This location is computed by specifying it as an offset from the current value of PC.
- Branch target may be either before or after the branch instruction, the offset is given as a singed num.

# Addressing Modes
# Relative Addressing

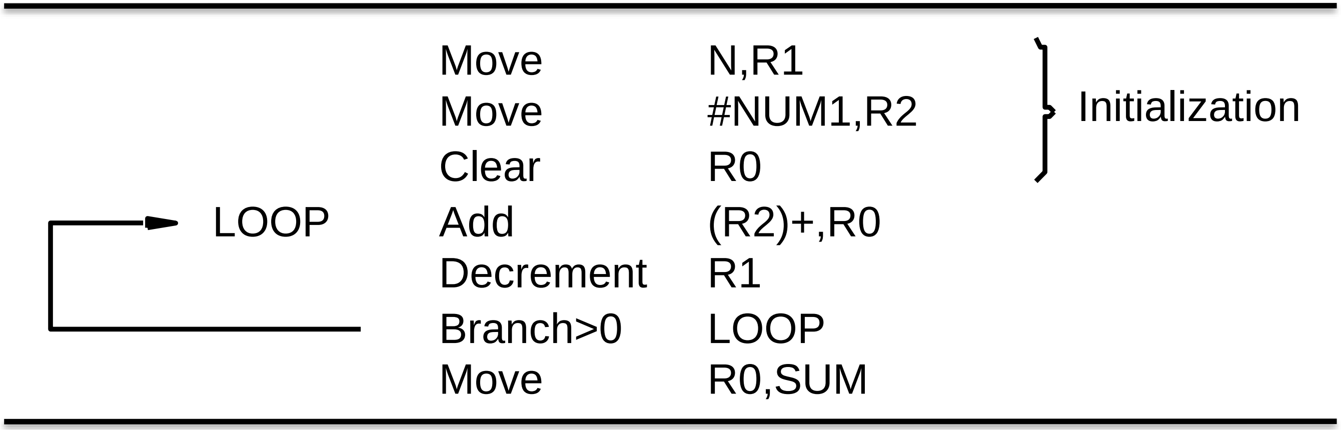
- Relative Address
  - *EA* = PC + Relative Addr (X)



Could be Positive or Negative (2's Complement)

# Addressing Modes
# 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.
- $(R_i)+$. The increment is 1 for byte-sized operands, 2 for 16-bit operands, and 4 for 32-bit operands.
- Autodecrement mode: $-(R_i)$ – decrement first



```
            Move        N,R1        } Initialization
            Move        #NUM1,R2    }
            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.

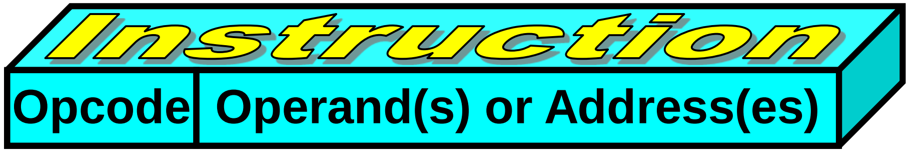
# Assembly Language

# Assembly Language

- Machine instructions are represented by patterns of 0s and 1s. So these patterns represented by symbolic names called "*mnemonics*"
- E.g. Load, Store, Add, Move, BR, BGTZ
- A complete set of such symbolic names and rules for their use constitutes a programming language, referred to as an *assembly language*.
- The set of rules for using the mnemonics and for 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 assembler program is one of a collection of utility programs that are a part of the system software of a computer.

# Assembly Language

- The user program in its original alphanumeric text format is called a *source program*, and the assembled machine-language program is called an *object program*.
- The assembly language for a given computer is not case sensitive.
- E.g. MOVE R1, SUM



Instruction

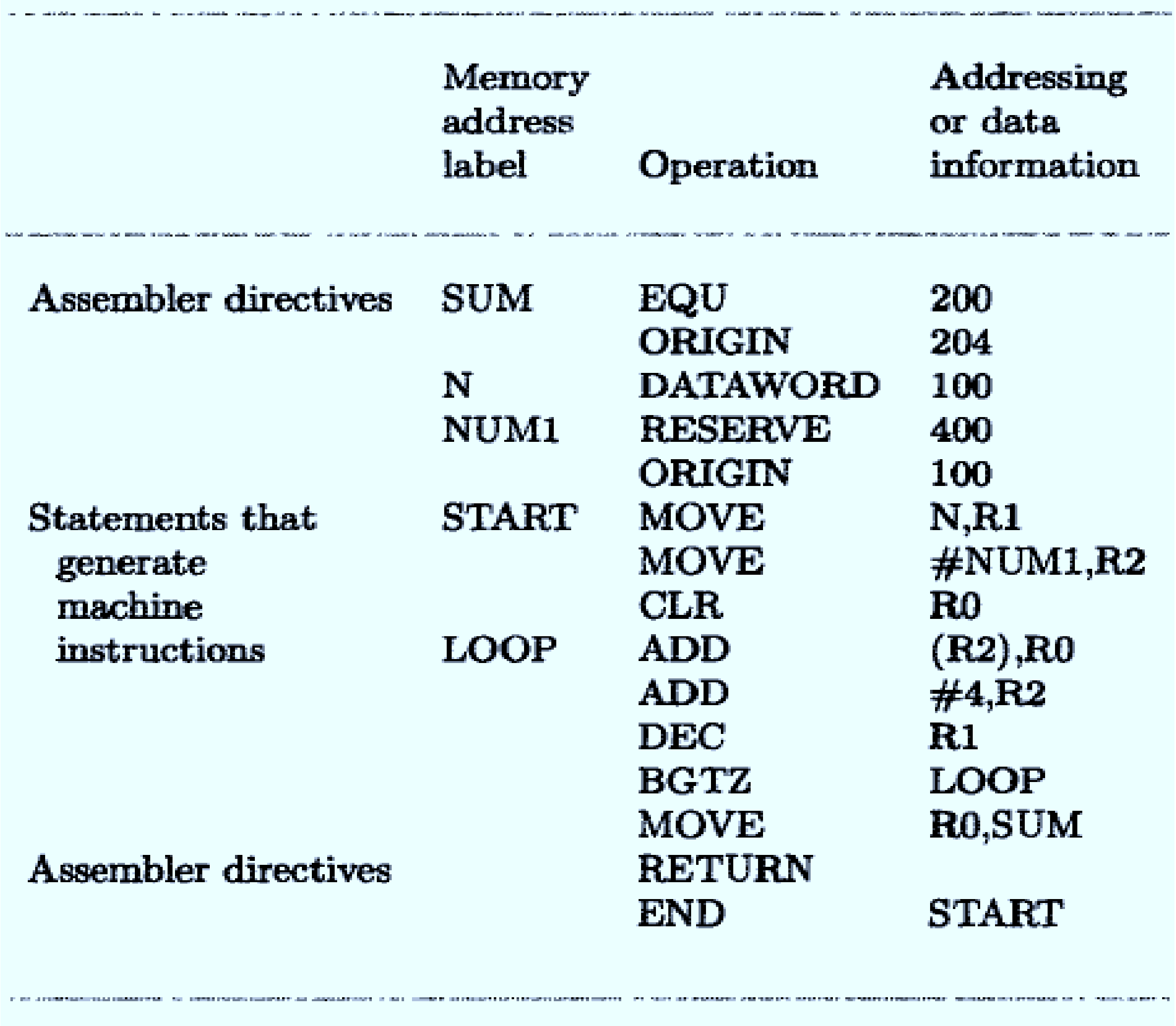
Opcode | Operand(s) or Address(es)

# Assembler Directives

- In addition to providing a mechanism for representing instructions in a program, assembly language allows the programmer to specify other information needed to translate the source program into the object program.
- Assign numerical values to any names used in a program.
  - For e,g, name TWENTY is used to represent the value 20. This fact may be conveyed to the assembler program through an *equate* statement such as TWENTY EQU 20
- If the assembler is to produce an object program according to this arrangement, it has to know
  - How to interpret the names
  - Where to place the instructions in the memory
  - Where to place the data operands in the memory

# Assembly language representation for the program

- Label: Operation Operand(s) Comment



| | 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 |

Assembly language representation for the program

# Assembly and Execution of Programs

- A source program written in an assembly language must be assembled into a machine language object program before it can be executed. This is done by the assembler program, which replaces all symbols denoting operations and addressing modes with the binary codes used in machine instructions, and replaces all names and labels with their actual values.
- A key part of the assembly process is determining the values that replace the names. Assembler keep track of Symbolic name and Label name, create table called symbol table.
- The symbol table created by scan the source program twice.
- A branch instruction is usually implemented in machine code by specifying the branch target as the distance (in bytes) from the present address in the Program Counter to the target instruction. The assembler computes this branch offset, which can be positive or negative, and puts it into the machine instruction.

# Assembly and Execution of Programs

- The assembler stores the object program on the secondary storage device available in the computer, usually a magnetic disk. The object program must be loaded into the main memory before it is executed. For this to happen, another utility program called a *loader* must already be in the memory.
- Executing the loader performs a sequence of input operations needed to transfer the machine-language program from the disk into a specified place in the memory. The loader must know the length of the program and the address in the memory where it will be stored.
- The assembler usually places this information in a header preceding the object code (Like start/end offset address).
- When the object program begins executing, it proceeds to completion unless there are logical errors in the program. The user must be able to find errors easily.
- The assembler can only detect and report syntax errors. To help the user find other programming errors, the system software usually includes a *debugger* program.
- This program enables the user to stop execution of the object program at some points of interest and to examine the contents of various processor registers and memory locations.

# Number Notation

- Decimal Number
  - ADD #93,R1
- Binary Number
  - ADD #%0101110,R1
- Hexadecimal Number
  - ADD #$5D,R1

# Types of Instructions

- Data Transfer Instructions

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

**Data value is not modified**

# Data Transfer Instructions

| Mode | Assembly | Register Transfer |
|---|---|---|
| Direct address | LD ADR | $AC \leftarrow M[ADR]$ |
| Indirect address | LD @ADR | $AC \leftarrow M[M[ADR]]$ |
| Relative address | LD $ADR | $AC \leftarrow M[PC+ADR]$ |
| Immediate operand | LD #NBR | $AC \leftarrow \mathrm{NBR}$ |
| Index addressing | LD ADR(X) | $AC \leftarrow M[ADR+XR]$ |
| Register | LD R1 | $AC \leftarrow R1$ |
| Register indirect | LD (R1) | $AC \leftarrow M[R1]$ |
| Autoincrement | LD (R1)+ | $AC \leftarrow M[R1]$, $R1 \leftarrow R1+1$ |

# Data Manipulation Instructions

- Arithmetic
- Logical & Bit Manipulation
- Shift

| Name | Mnemonic |
|---|---|
| Increment | INC |
| Decrement | DEC |
| Add | ADD |
| Subtract | SUB |
| Multiply | MUL |
| Divide | DIV |
| Add with carry | ADDC |
| Subtract with borrow | SUBB |
| Negate | NEG |

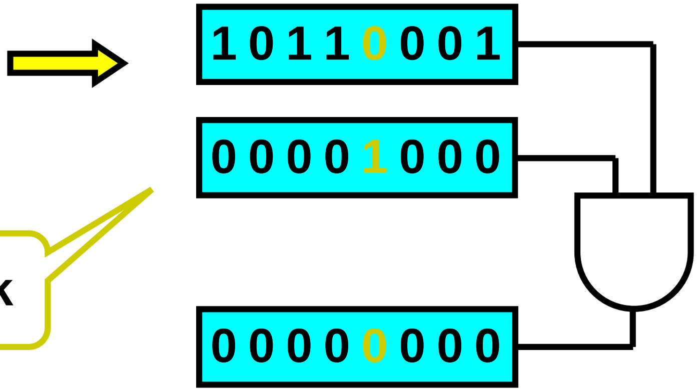
| Name | Mnemonic |
|---|---|
| Clear | CLR |
| Complement | COM |
| AND | AND |
| OR | OR |
| Exclusive-OR | XOR |
| Clear carry | CLRC |
| Set carry | SETC |
| Complement carry | COMC |
| Enable interrupt | EI |
| Disable interrupt | DI |

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

# Program Control Instructions

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

Subtract A – B but don't store the result



1 0 1 1 0 0 0 1

0 0 0 0 1 0 0 0

0 0 0 0 0 0 0 0

Mask

# Conditional Branch Instructions

| Mnemonic | Branch Condition | Tested Condition |
|---|---|---|
| BZ | Branch if zero | Z = 1 |
| BNZ | Branch if not zero | Z = 0 |
| BC | Branch if carry | C = 1 |
| BNC | Branch if no carry | C = 0 |
| BP | Branch if plus | S = 0 |
| BM | Branch if minus | S = 1 |
| BV | Branch if overflow | V = 1 |
| BNV | Branch if no overflow | V = 0 |

# Basic Input/Output Operations

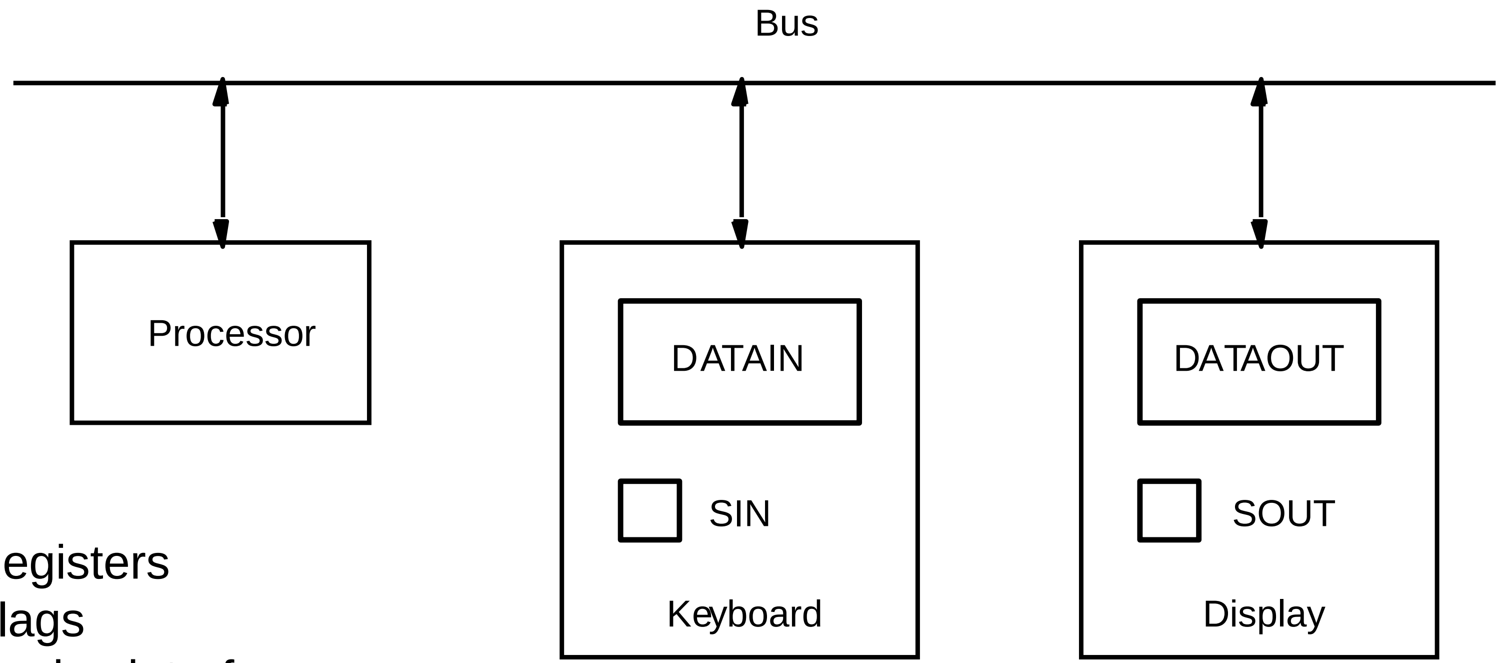
# I/O

- The data on which the instructions operate are not necessarily already stored in memory.
- Data need to be transferred between processor and outside world (disk, keyboard, etc.)
- I/O operations are essential, the way they are performed can have a significant effect on the performance of the computer.

# Program-Controlled I/O Example

- 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

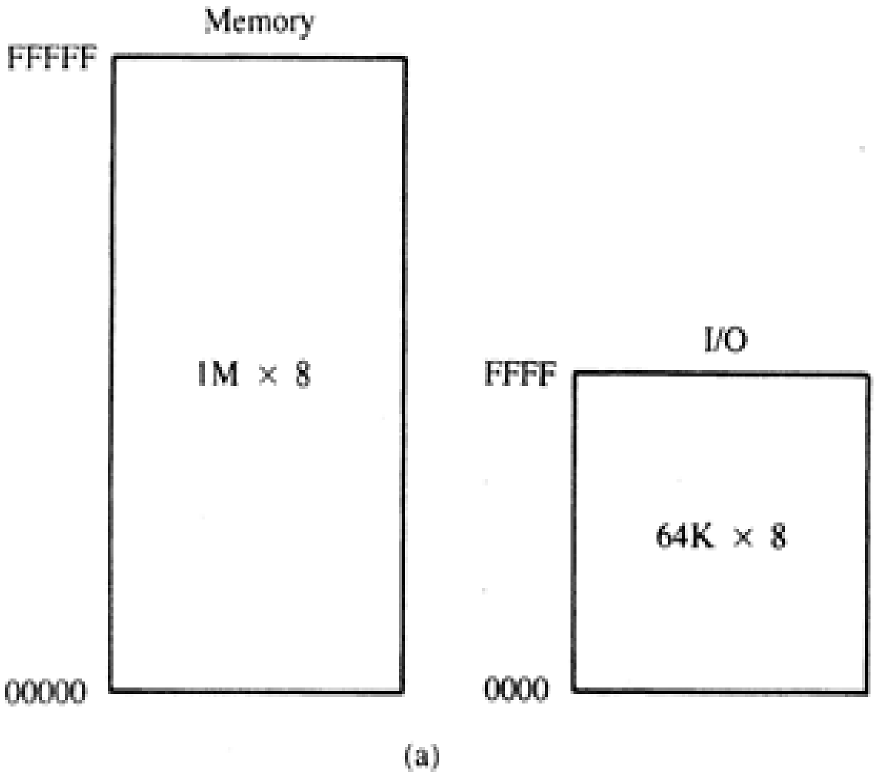
Figure 2.19 Bus connection for processor , keyboard, and display

# Program-Controlled I/O Example

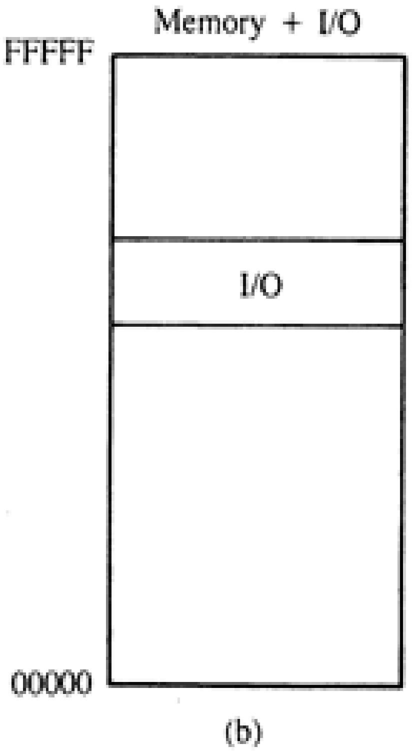
- Machine instructions that can check the state of the status flags and transfer data:

```
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



Memory Mapped I/O



I/O Mapped I/O

# 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.
  - E.g.     Movebyte DATAIN,R1
  -          Movebyte R1,DTATOUT

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

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

# Program-Controlled I/O Example

- Assumption – the initial state of SIN is 0 and the initial state of SOUT is 1.
- Any drawback of this mechanism in terms of efficiency?
  - Two wait loops→processor execution time is wasted
- Alternate solution?
  - Interrupt

# Stacks

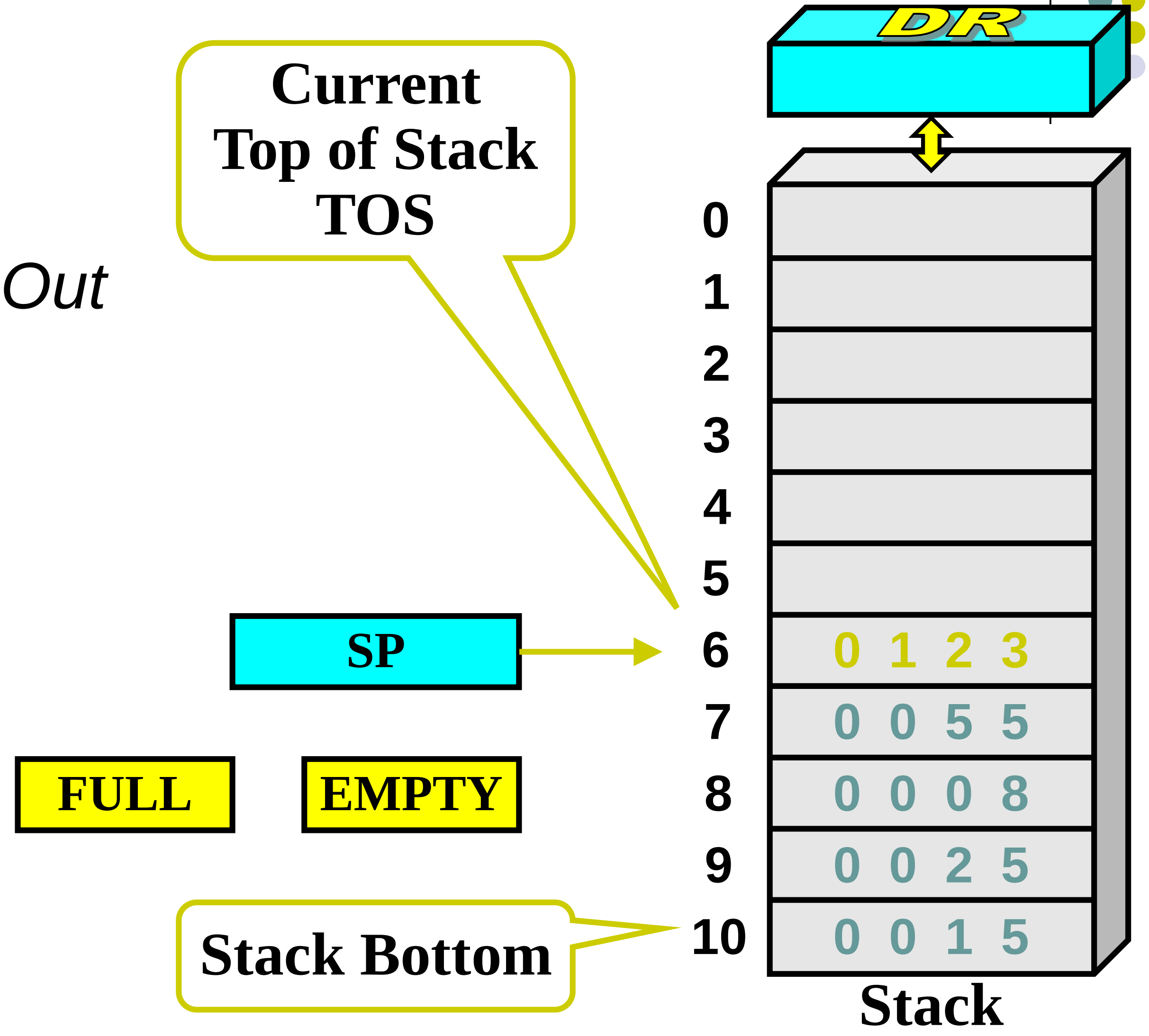
# Stacks

- A *stack* is a list of data elements, usually words, with the accessing restriction that elements can be added or removed at one end of the list only. This end is called the top of the stack, and the other end is called the bottom. The structure is sometimes referred to as a *pushdown* stack.
- *last-in–first-out* (LIFO) stack working.
- The terms *push* and *pop* are used to describe placing a new item on the stack and removing the top item from the stack, respectively.
- The stack pointer, SP, is used to keep track of the address of the element of the stack that is at the top at any given time.

# Stack Organization

- LIFO

  *Last In First Out*



| | |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 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

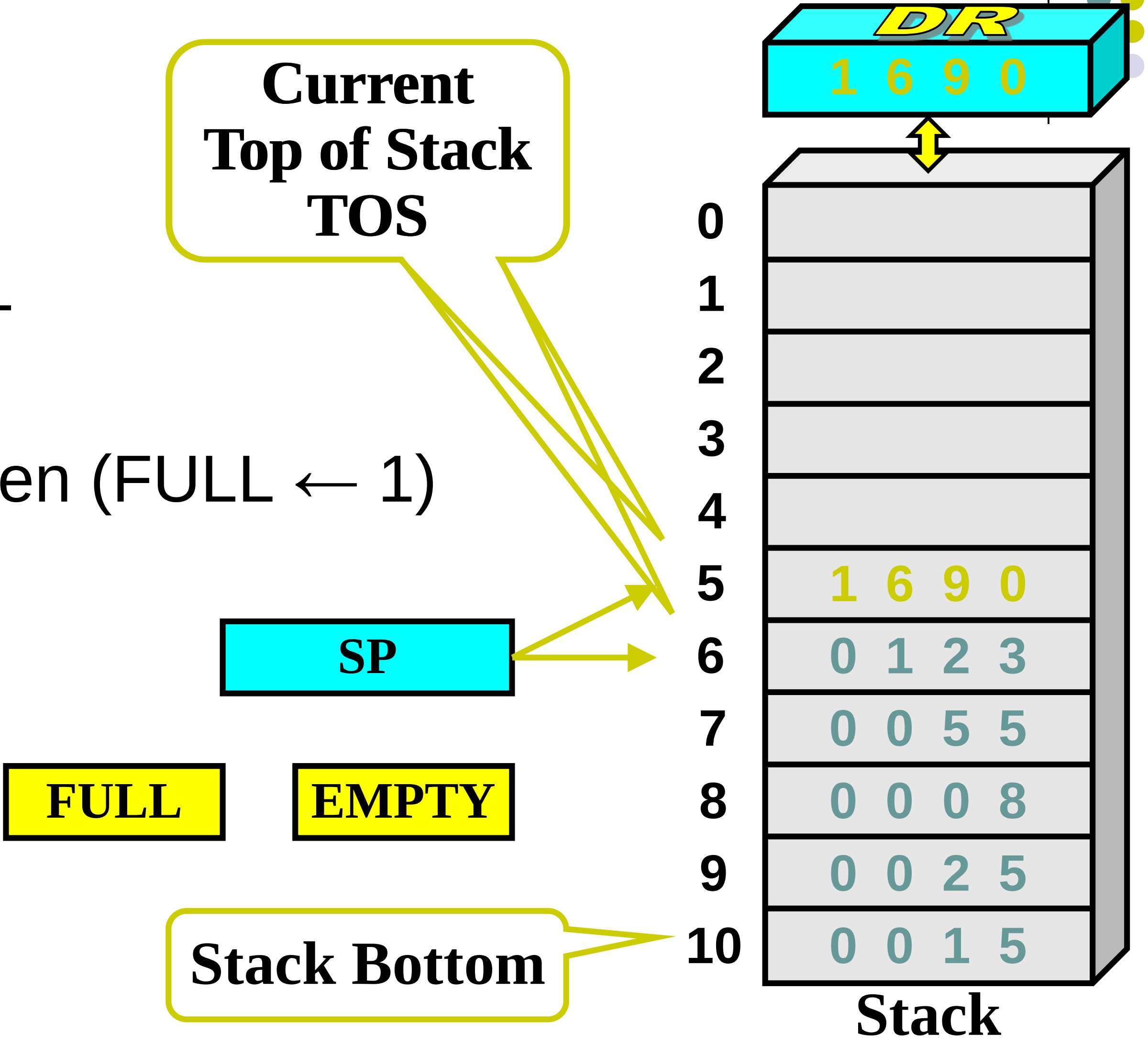
- PUSH

  SP ← SP – 1

  M[SP] ← DR

  If (SP = 0) then (FULL ← 1)

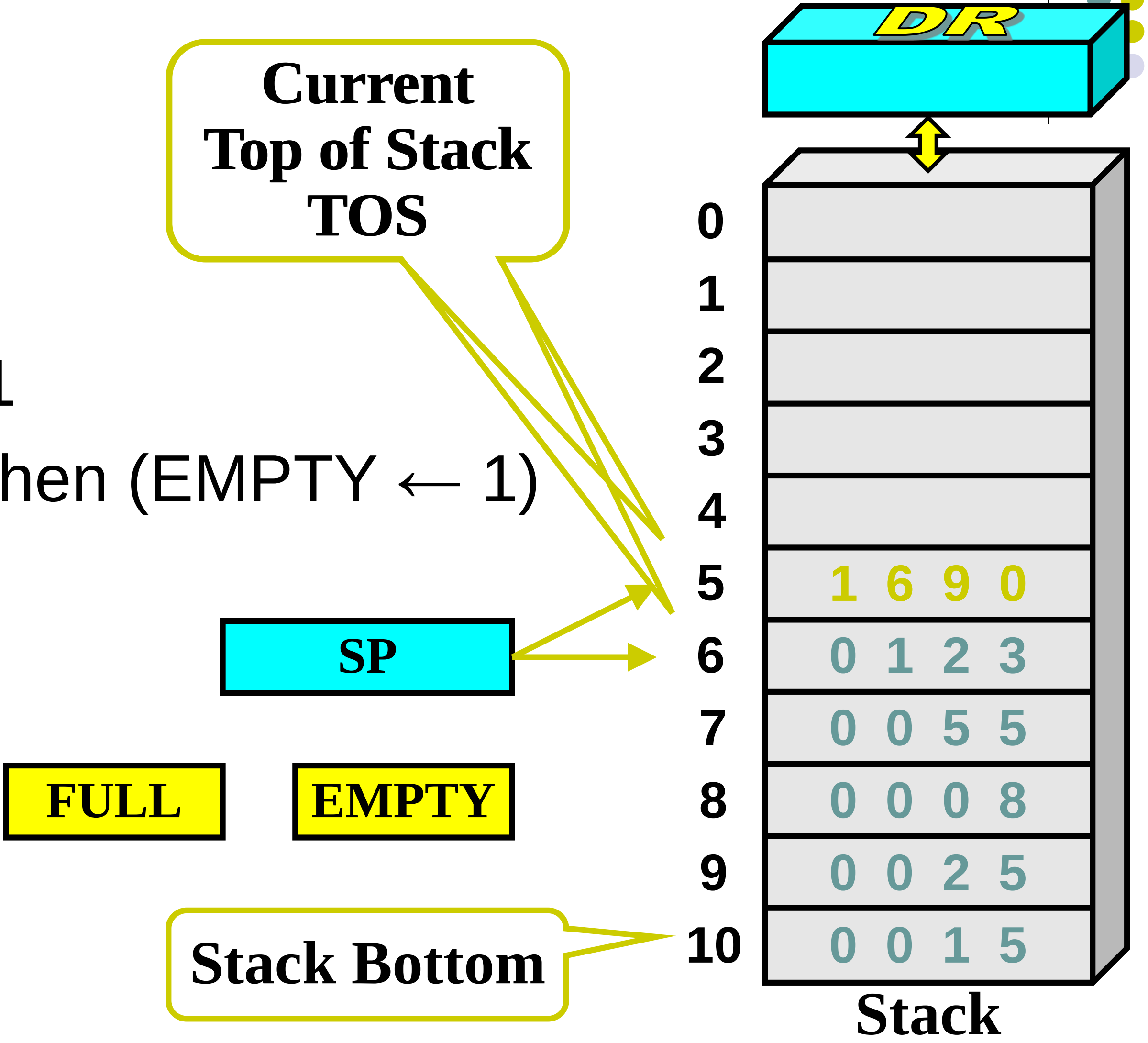
  EMPTY ← 0



# Stack Organization

- POP

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



# Stack Organization

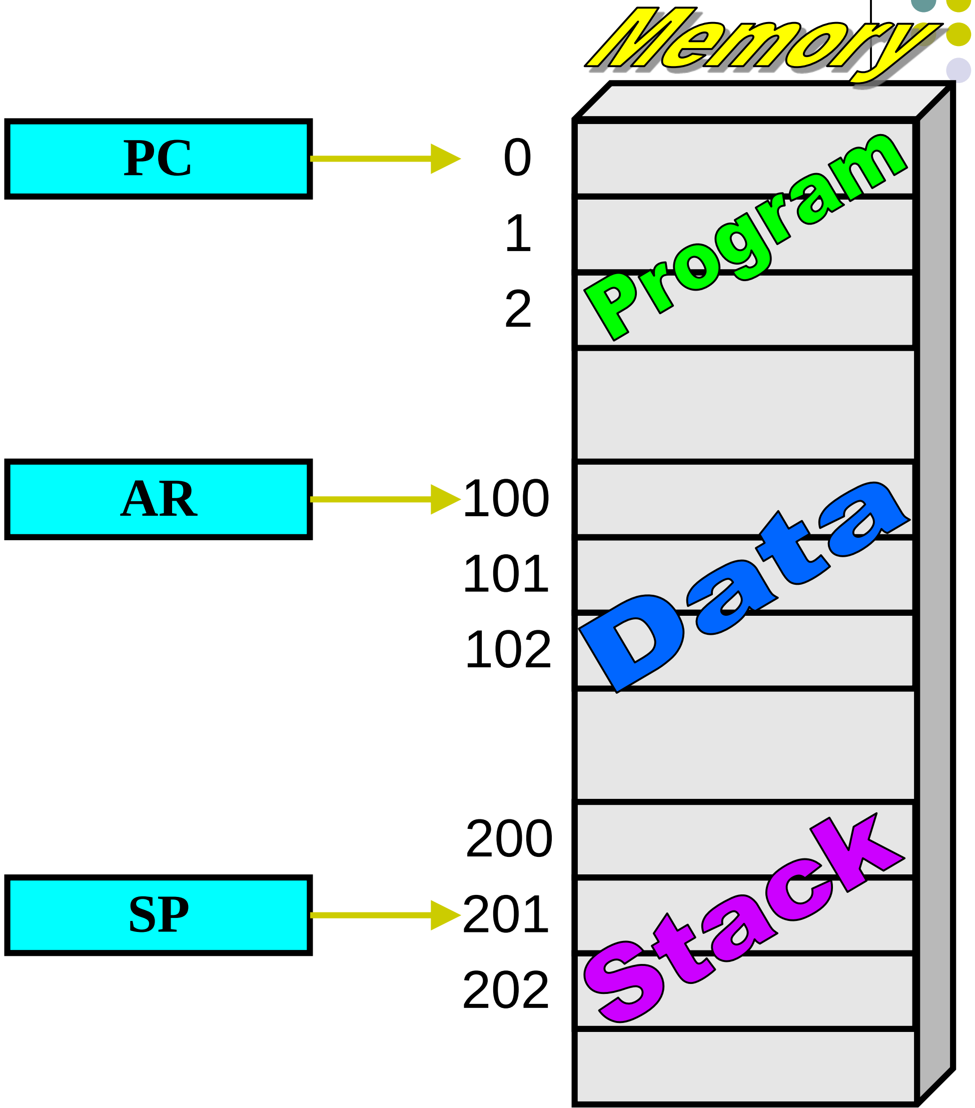
- Memory Stack
  - PUSH

    SP ← SP – 1

    M[SP] ← DR
  - POP

    DR ← M[SP]

    SP ← SP + 1



# Queue

# Queue

- FIFO basis
- Data are stored in and retrieved from a queue on a first-in–first-out (FIFO) basis. Thus, if we assume that the queue grows in the direction of increasing addresses in the memory, new data are added at the back (high-address end) and retrieved from the front (low-address end) of the queue.

# Differences between a stack and a queue

| Stack | Queue |
|---|---|
| LIFO | FIFO |
| One end is fixed other end for PUSH and POP item | One end is to add item and other is to remove item |
| One pointer used | Two Pointer is used |
| Fixed Size | Not fixed size |

# Subroutines

# Subroutines

- In a given program, it is often necessary to perform a particular task many times on different data values. It is prudent to implement this task as a block of instructions that is executed each time the task has to be performed. Such a block of instructions is usually called a *subroutine*.
- However, to save space, only one copy of this block is placed in the memory, and any program that requires the use of the subroutine simply branches to its starting location.
- 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, and it does so by executing a Return instruction.

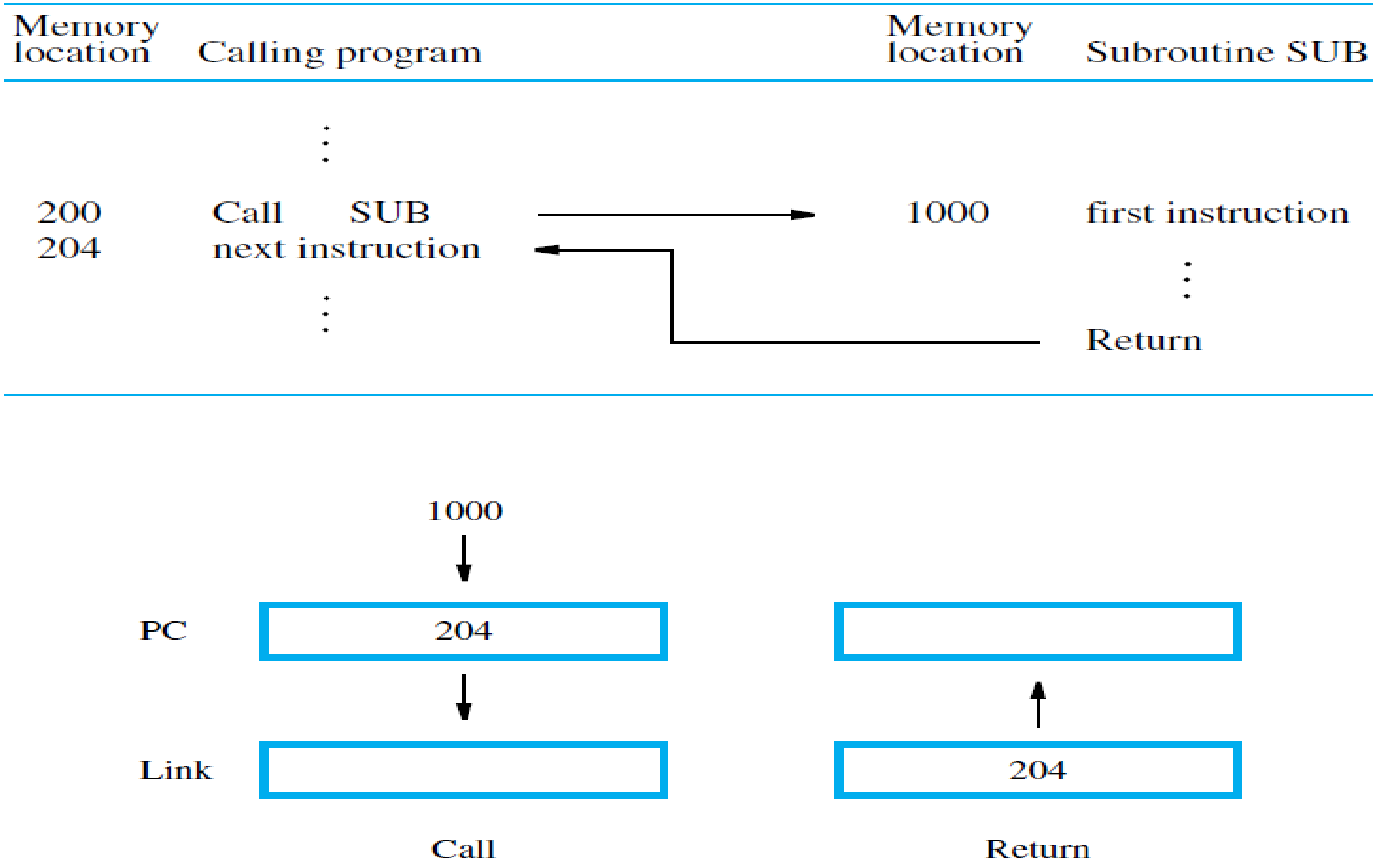
# Subroutines

- Since the subroutine may be called from different places in a calling program, provision must be made for returning to the appropriate location. The location where the calling program resumes execution is the location pointed to by the updated program counter (PC) while the Call instruction is being executed.
- Hence, the contents of the PC must be saved by the Call instruction to enable correct return to the calling program.
- 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.

# Subroutines

- 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 Call instruction
- The Return instruction is a special branch instruction that performs the operation
  - Branch to the address contained in the link register

# Subroutines



| Memory location | Calling program | Memory location | Subroutine SUB |
|---|---|---|---|
| | ⋮ | | |
| 200 | Call SUB | 1000 | first instruction |
| 204 | next instruction | | ⋮ |
| | ⋮ | | Return |

Subroutine linkage using a link register.

# Subroutine Nesting and the Processor Stack

- 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, overwriting 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.
- That is, return addresses are generated and used in a last-in–first-out order. This suggests that the return addresses associated with subroutine calls should be pushed onto the processor stack.

# Parameter Passing

- When calling a subroutine, a program must provide to the subroutine the parameters, that is, the operands or their addresses, to be used in the computation. Later, the subroutine returns other parameters, which are the results of the computation. This 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, in memory locations, or on the processor stack where they can be accessed by the subroutine.

# Program of subroutine
# Parameters passed through registers.

- Calling Program

1. Move N, R1
2. Move #NUM1,R2
3. Call LISTADD
4. Move R0,SUM

- Subroutine

```
1. LISTADD: Clear R0
2. LOOP:    Add (R2)+,R0
3.          Decrement R1
4.          Branch>0 LOOP
5.          Return
```

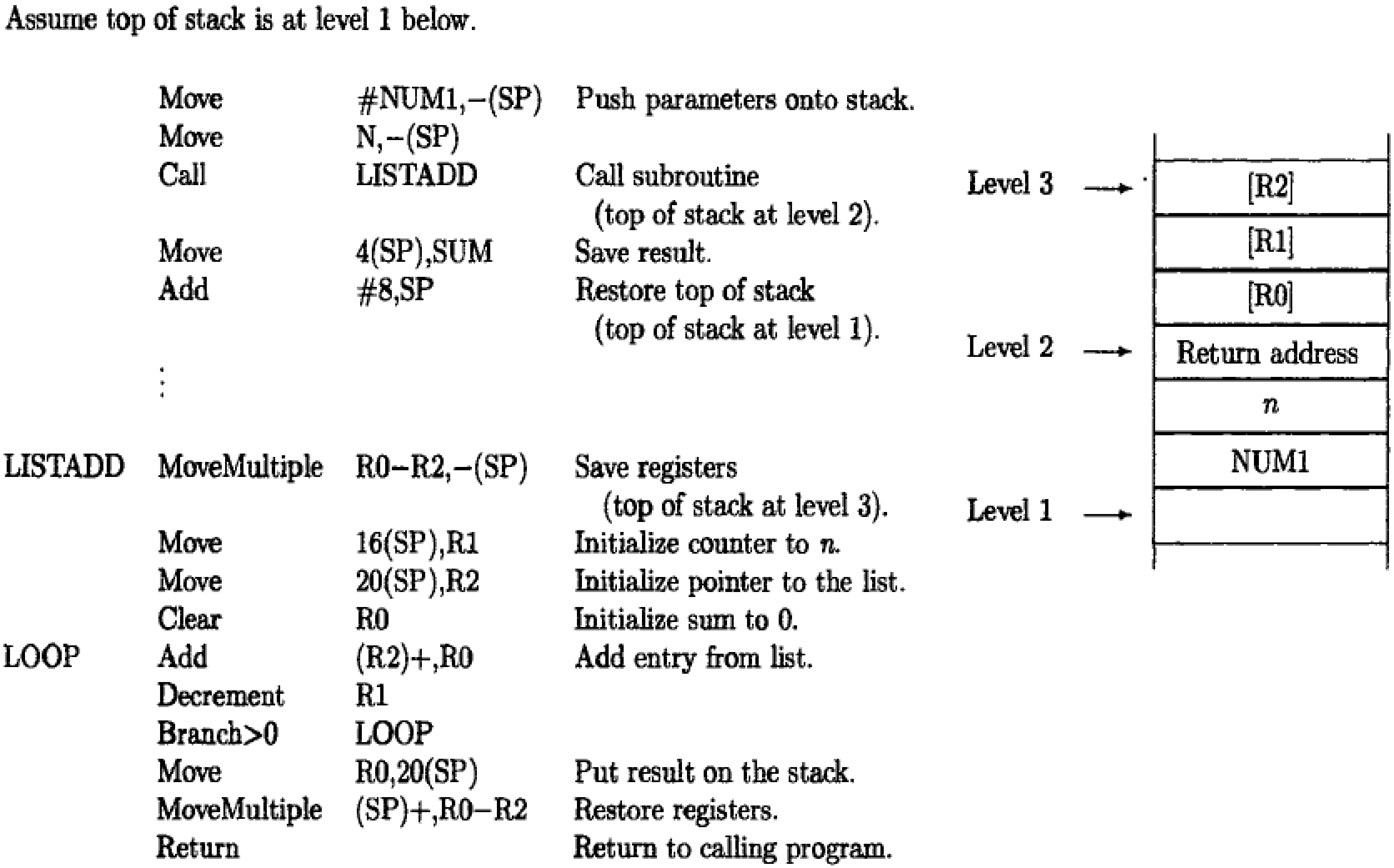
# Parameter Passing by Value and by Reference

- Instead of passing the actual Value(s), the calling program passes the address of the Value(s). This technique is called *passing by reference*.
- The second parameter is *passed by value*, that is, the actual number of entries, is passed to the subroutine.

# Program of subroutine Parameters passed on the stack.

- MoveMultiple R0-R2, -(SP)
- MoveMultiple to store contents of register R0 through R2 on he stack

# Program of subroutine Parameters passed on the stack.



Assume top of stack is at level 1 below.

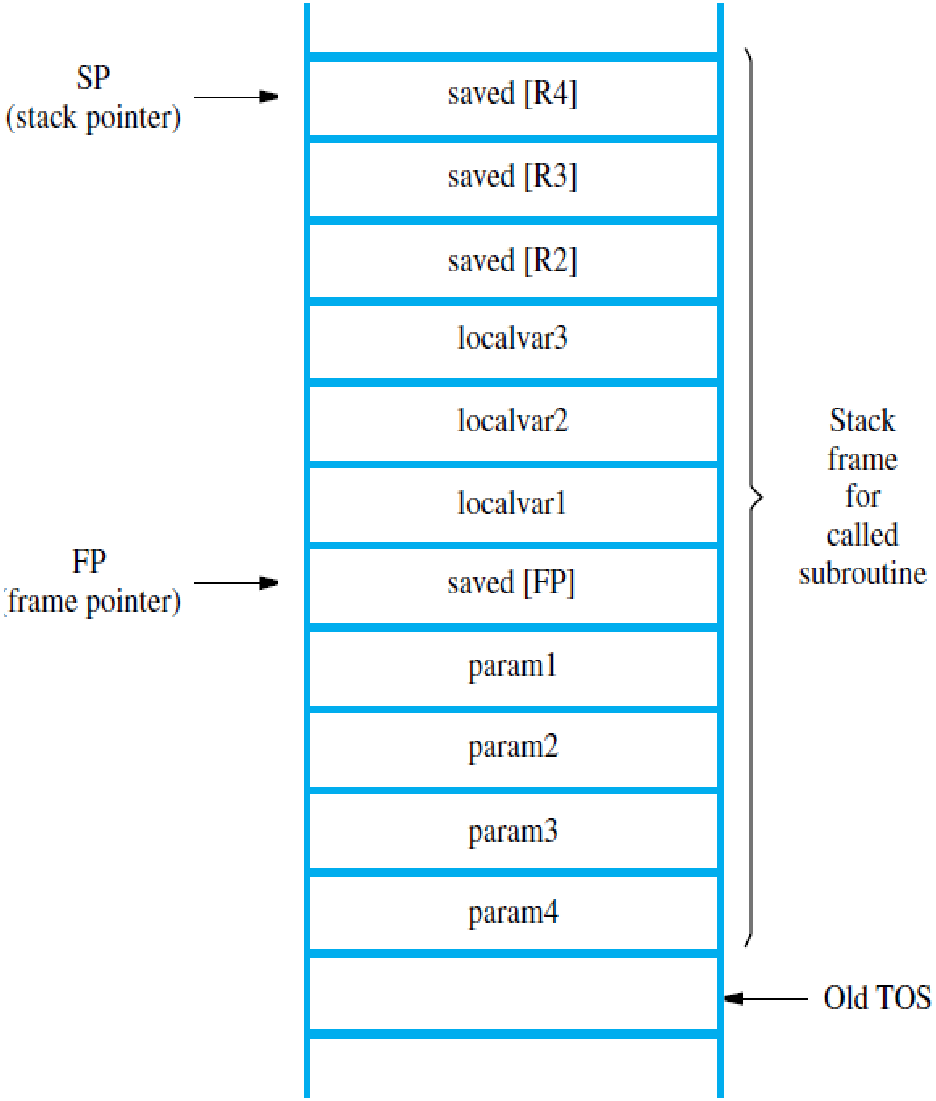
```
            Move          #NUM1,–(SP)   Push parameters onto stack.
            Move          N,–(SP)
            Call          LISTADD       Call subroutine
                                          (top of stack at level 2).
            Move          4(SP),SUM     Save result.
            Add           #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.
```

# The Stack Frame

- If the subroutine requires more space for local memory variables, the space for these variables can also be allocated on the stack this area of stack is called Stack Frame.
- For e.g. 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 work space for the subroutine, allocated at the time the subroutine is entered and deallocated when the subroutine returns control to the calling program.

# The Stack Frame

- *Frame pointer* (FP), for convenient access to the parameters passed to the subroutine and to the local memory variables used by the subroutine.
- In the figure, we assume that four parameters are passed to the subroutine, three local variables are used within the subroutine, and registers R2, R3, and R4 need to be saved because they will also be used within the subroutine.
- When nested subroutines are used, the stack frame of the calling subroutine would also include the return address, as we will see in the example that follows.



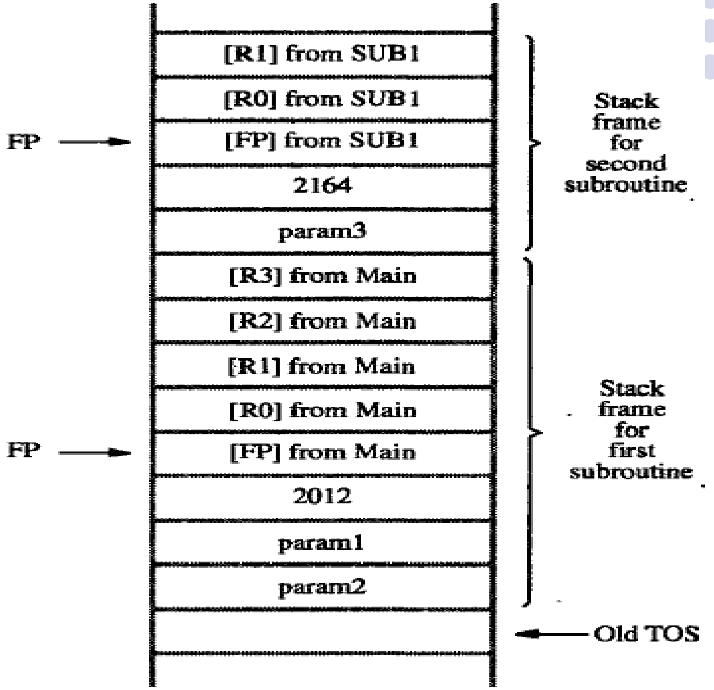
# Stack Frames for Nested Subroutines

**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.
```



[R1] from SUB1

[R0] from SUB1

FP → [FP] from SUB1

2164

param3

Stack frame for second subroutine

[R3] from Main

[R2] from Main

[R1] from Main

[R0] from Main

FP → [FP] from Main

2012

param1

param2

Stack frame for first subroutine

← Old TOS

**Second subroutine**

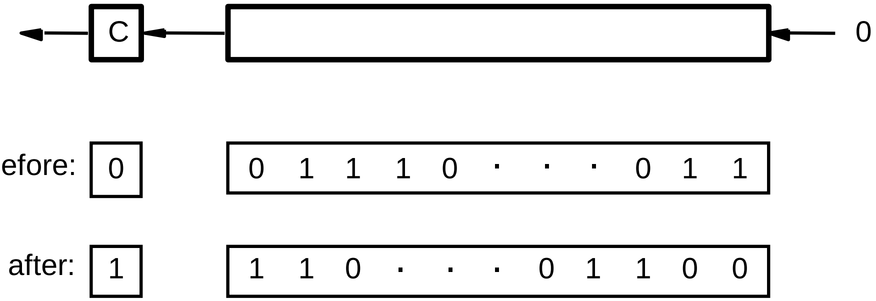
```
3000  SUB2  Move          FP,–(SP)       Save frame pointer register.
            Move          SP,FP          Load the frame pointer.
            MoveMultiple  R0–R1,–(SP)    Save registers R0 and R1.
            Move          8(FP),R0       Get the parameter.
              ⋮
            Move          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.
```

# Stack Frames for Nested Subroutines

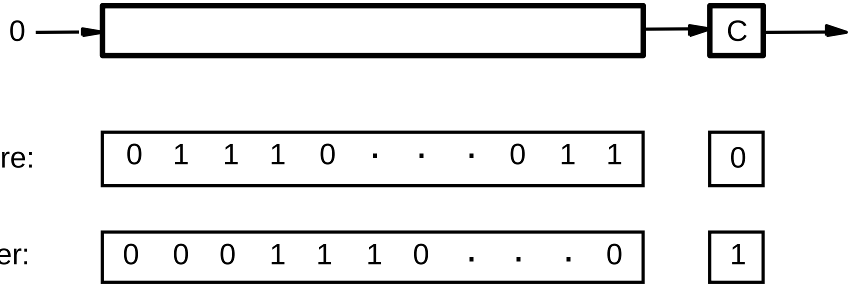
# Additional Instructions

# Logical Shifts

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

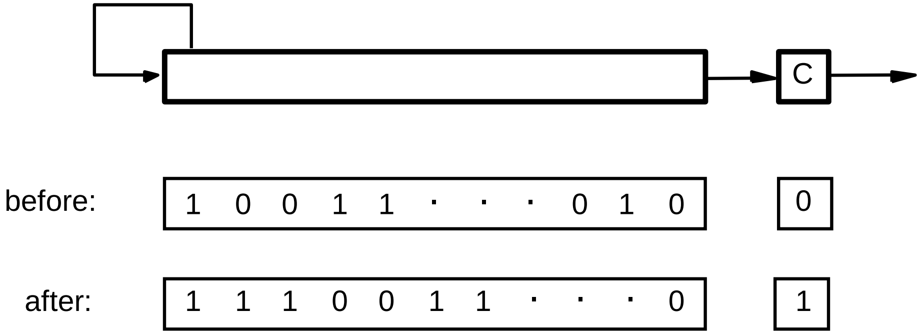


(a) Logical shift left LShiftL #2,R0



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

# Arithmetic Shifts

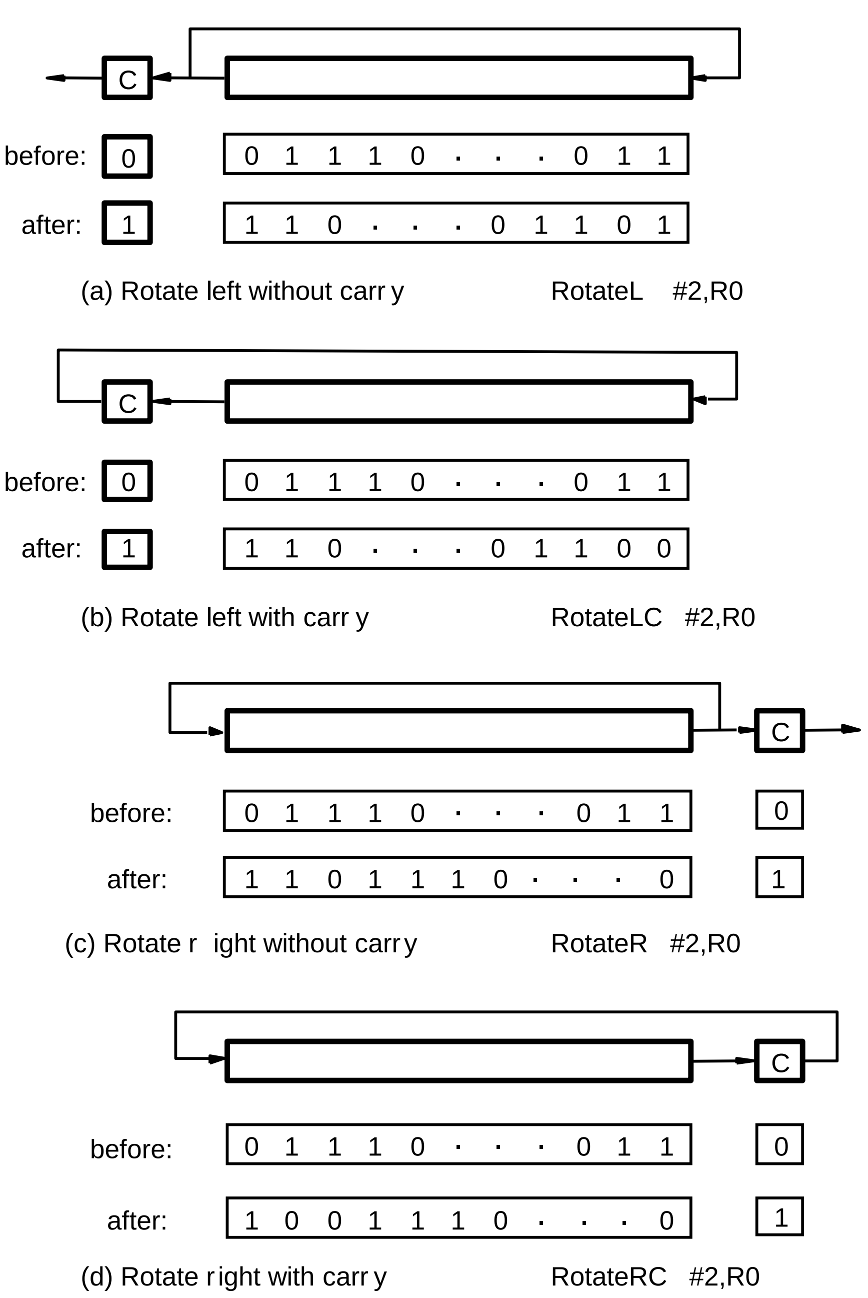


before: 1 0 0 1 1 · · · 0 1 0   0

after: 1 1 1 0 0 1 1 · · · 0   1

(c) Arithmetic shift right   AShiftR #2,R0

# Rotate



C

before: 0 | 0 1 1 1 0 . . . 0 1 1

after: 1 | 1 1 0 . . . 0 1 1 0 1

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

C

before: 0 | 0 1 1 1 0 . . . 0 1 1

after: 1 | 1 1 0 . . . 0 1 1 0 0

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

C

before: 0 1 1 1 0 . . . 0 1 1 | 0

after: 1 1 0 1 1 1 0 . . . 0 | 1

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

C

before: 0 1 1 1 0 . . . 0 1 1 | 0

after: 1 0 0 1 1 1 0 . . . 0 | 1

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

Figure 2.32. Rotate instructions.

# Multiplication and Division

- Not very popular (especially division)
- Multiply $R_i$, $R_j$
  $R_j \leftarrow [R_i] \times [R_j]$
- 2n-bit product case: high-order half in R(j+1)
- Divide $R_i$, $R_j$
  $R_j \leftarrow [R_i] / [R_j]$
  Quotient is in Rj, remainder may be placed in R(j+1)

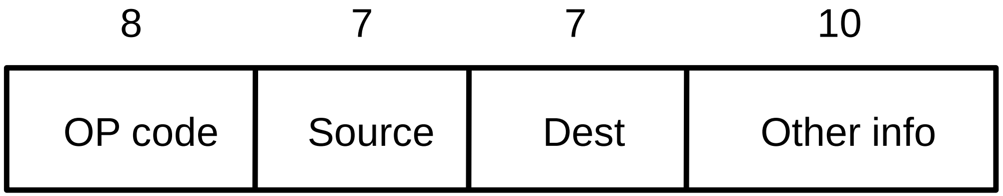
# Logic Instructions

- And R2, R3, R4
- And #Value, R4, R2
- And #$0FF, R2, R2,

# Encoding of Machine Instructions

# Encoding of Machine Instructions

- Assembly language program needs to be converted into machine instructions. (ADD = 0100 in ARM instruction set)
- In the previous section, an assumption was made that all instructions are one word in length.
- OP code: the type of operation to be performed and the type of operands used may be specified using an encoded binary pattern
- Suppose 32-bit word length, 8-bit OP code (how many instructions can we have?), 16 registers in total (how many bits?), 3-bit addressing mode indicator.
- Add  R1, R2
- Move  24(R0), R5
- LshiftR  #2, R0
- Move  #$3A, R1

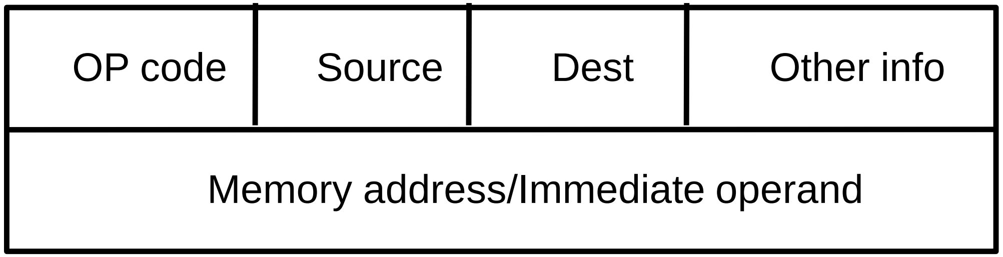


| 8 | 7 | 7 | 10 |
|---|---|---|---|
| OP code | Source | Dest | Other info |

(a) One-word instruction

# Encoding of Machine Instructions

- What happens if we want to specify a memory operand using the Absolute addressing mode?
- Move R2, LOC
- 14-bit for LOC – insufficient
- Solution – use two words



OP code | Source | Dest | Other info

Memory address/Immediate operand

(b) Two-word instruction

# Encoding of Machine Instructions

- Then what if an instruction in which two operands can be specified using the Absolute addressing mode?
- Move  LOC1, LOC2
- Solution – use two additional words
- This approach results in instructions of variable length. Complex instructions can be implemented, closely resembling operations in high-level programming languages – Complex Instruction Set Computer (CISC)

# Encoding of Machine Instructions

- If we insist that all instructions must fit into a single 32-bit word, it is not possible to provide a 32-bit address or a 32-bit immediate operand within the instruction.
- It is still possible to define a highly functional instruction set, which makes extensive use of the processor registers.
- Add  R1, R2 ----- yes
- Add  LOC, R2 ----- no
- Add  (R3), R2 ----- yes