Chapter 2

Machine Instructions and Programs

❖ Number, Arithmetic Operations, and Characters

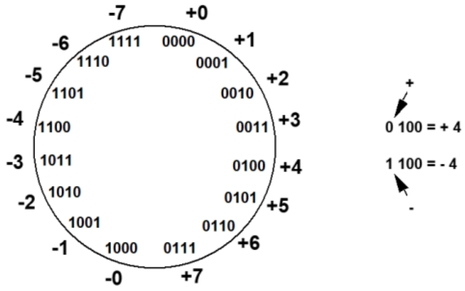
❑ Signed Integer Representations:

- ➢ 3 major representations:
  - Sign and magnitude
  - One’s complement
  - Two’s complement

★ Assumptions **for further examples:**

- ✓ The computer works with 4-bit words, meaning every number is stored using only 4 bits (0s and 1s).
- ✓ With 4 bits, 2^4 = 16 different values can be represented.
- ✓ These 16 representations are roughly divided into half positive and half negative integers.

- **Sign and Magnitude Representation**



Concept: This method uses the most significant bit (MSB) as the sign bit (0 for positive or zero, 1 for negative). The remaining bits represent the magnitude (absolute value) of the number.

Example: Using the circular diagram, a number like 0100 represents +4, while 1100 is -4.

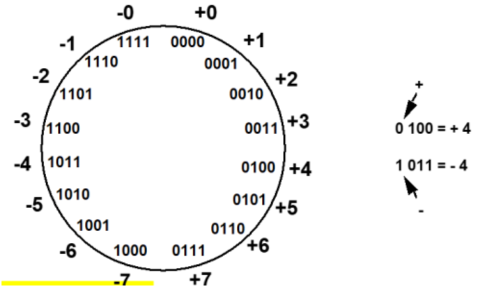
The number of values that can be represented with n bits= +/-(2^{n-1} - 1), resulting in a range of approximately half positive and half negative numbers.

Problems:

(i) **Dual zero:** Two representations for zero +0(0000) and –0(1000)(See the number wheel).

(ii) **Complex Arithmetic:** Complexities performing Addition, Subtraction.

- **One’s Complement Representation**



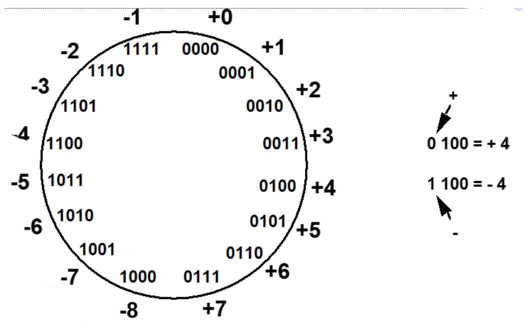
- ▪ 1’s complement is invert 0 to 1 and 1 to 0
- ▪ - x = 1’s complement of x. If we have +7 as 0111, it’s One’s complement would be 1000.

- Like Sign and Magnitude, One's Complement has two representations for zero: 0000 (positive zero or +0) and 1111 (negative zero or -0). This is a key weakness of one's complement.
- Subtraction in one's complement is done by adding the one's complement of the number to be subtracted. That is, X - Y is implemented as X + (one's complement of Y), or, X + Y'.

Problems:

i) **Two Zeros**: The two representations of zero are problematic for logical operations.
ii) **Complexities in Arithmetic**: While better than sign-magnitude, addition and subtraction still involve extra steps like the end-around carry, making it more complex than two's complement.

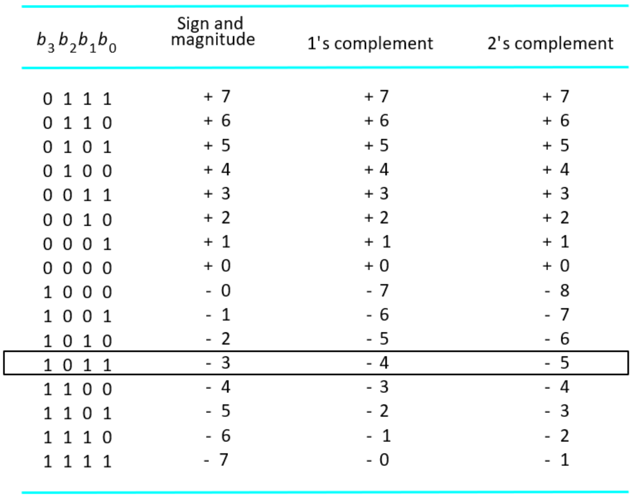
Two's Complement Representation



- l- x = 2's complement of x
- l2's complement is just 1's complement + 1
- It has only one representation for zero, 0000. If you take 0 (0000), invert its bits (1111), and add 1, you get 10000. However, since we are working with 4-bit numbers, we discard the carry bit, resulting in a single zero (0000).
- Like 1's complement except negative numbers shifted one position clockwise.
- Addition, Subtraction very simple.

Binary, Signed-Integer Representations

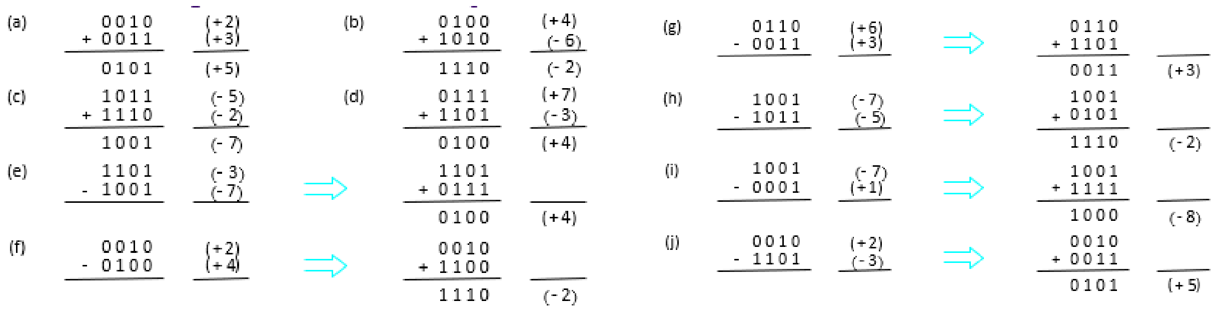
Values represented



$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+ 7	+ 7	+ 7
0 1 1 0	+ 6	+ 6	+ 6
0 1 0 1	+ 5	+ 5	+ 5
0 1 0 0	+ 4	+ 4	+ 4
0 0 1 1	+ 3	+ 3	+ 3
0 0 1 0	+ 2	+ 2	+ 2
0 0 0 1	+ 1	+ 1	+ 1
0 0 0 0	+ 0	+ 0	+ 0
1 0 0 0	- 0	- 7	- 8
1 0 0 1	- 1	- 6	- 7
1 0 1 0	- 2	- 5	- 6
1 0 1 1	- 3	- 4	- 5
1 1 0 0	- 4	- 3	- 4
1 1 0 1	- 5	- 2	- 3
1 1 1 0	- 6	- 1	- 2
1 1 1 1	- 7	- 0	- 1

Figure 2.1. Binary, signed-integer representations.

2's-Complement Add and Subtract Operations

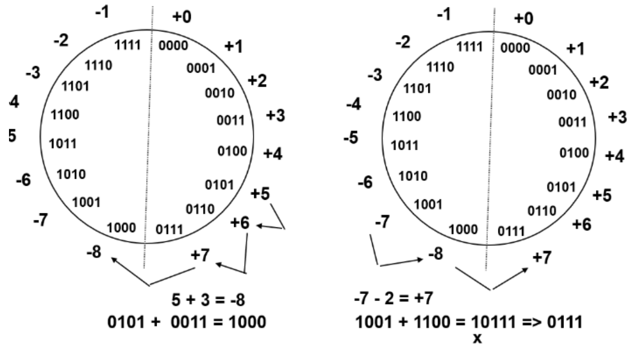


```
(a)    0010   (+2)        (b)    0100   (+4)        (g)    0110   (+6)          0110
     + 0011   (+3)             + 1010   (- 6)            - 0011   (+3)    ⇒   + 1101
     ------   ----             ------   ----             ------   ----        ------   ----
       0101   (+5)               1110   (- 2)                                   0011   (+3)

(c)    1011   (- 5)       (d)    0111   (+7)        (h)    1001   (- 7)          1001
     + 1110   (- 2)            + 1101   (- 3)            - 1011   (- 5)   ⇒   + 0101
     ------   ----             ------   ----             ------   ----        ------   ----
       1001   (- 7)              0100   (+4)                                    1110   (- 2)

(e)    1101   (- 3)              1101               (i)    1001   (- 7)          1001
     - 1001   (- 7)   ⇒        + 0111                    - 0001   (+1)    ⇒   + 1111
     ------   ----             ------   ----             ------   ----        ------   ----
                                 0100   (+4)                                    1000   (- 8)

(f)    0010   (+2)               0010               (j)    0010   (+2)           0010
     - 0100   (+4)    ⇒        + 1100                    - 1101   (- 3)   ⇒   + 0011
     ------   ----             ------   ----             ------   ----        ------   ----
                                 1110   (- 2)                                   0101   (+ 5)
```

Figure. 2’s-complement Add and Subtract operations.

Overflow happens when the result of an addition goes beyond the maximum positive or minimum negative value that can be represented within the available bits. This can lead to incorrect results if not detected and handled.

- ❑ **Overflow Condition:**
  - ✓ Adding two positive numbers gives a negative result.
  - ✓ Adding two negative numbers gives a positive result.



- ➢ **Overflow Condition – Carry in to MSB ≠ Carry out from MSB**
  - ♦ **Carry Bits in Addition**
    - ▪ Carry-in to the MSB: When adding two binary numbers, there might be a carry generated from the addition of the bit before the MSB (the second most significant bit).
    - ▪ Carry-out from the MSB: Similarly, there might be a carry generated out of the MSB when the bits at the MSB position are added, and this carry is usually discarded.
    - ▪ Overflow Detection: An overflow happens when the carry-in to the MSB is different from the carry-out from the MSB.

```
                  0 1 1 1←carry-in
Overflow     5     0 1 1 0 1          -7      1 0 0 0
                                                1 0 0 1     Overflow
             3     0 0 1 1            -2        1 1 0 0
            --    --------            --      ---------
            -8     1 0 0 0             7      1 0 1 1 1

                  0 0 0 0                     1 1 1 1
No           5     0 1 0 1            -3        1 1 0 1     No overflow
overflow     2     0 0 1 0            -5        1 0 1 1
            --    --------            --      ---------
             7     0 1 1 1            -8      1 1 0 0 0
```

★ **Two Ways to detect Overflow:**

1) when carry-in to the MSB (most significant bit) does not equal carry out from MSB
2) Add two positive numbers to get a negative number or, add two negative numbers to get a positive number.

❖ Memory Locations, Addresses, and Operations

Memory locations

- Memory in a computer system consists of millions of tiny storage cells, and each of these cells can store 1 bit (a binary digit).
- Data is usually accessed in groups of n bits, known as a "word."
- n is the “word length”.
- Commonly, n is 32 or 64 bits. Computer systems with these word lengths are referred to as 32-bit systems or 64-bit systems. For example, a 32-bit CPU or a 64-bit OS.

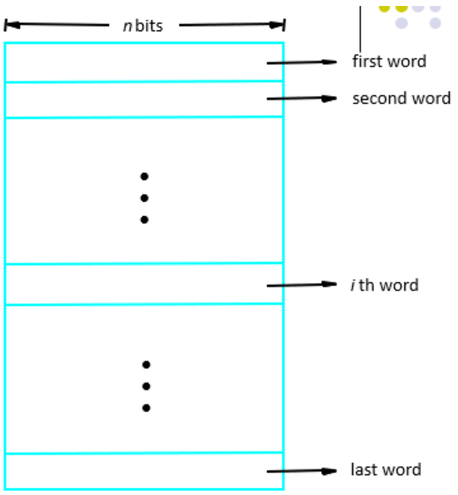
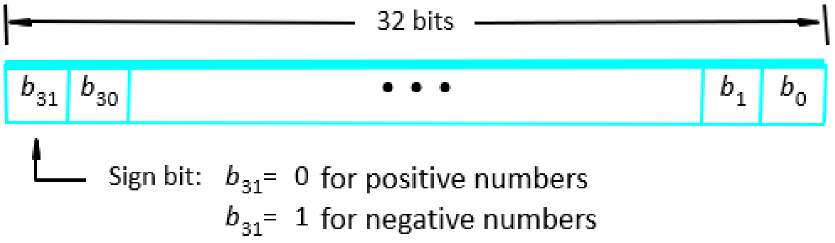
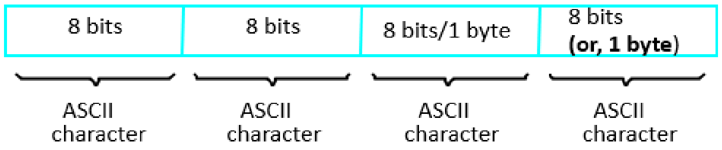


Figure 2.5. Memory words.

➢ 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



(b) Four characters

❑ Retrieving information from Memory:

- To access information from memory, each location (whether it's for one word or one byte) needs an address.
- Each byte (8 bits) in memory can be individually addressed, a feature known as **byte addressability**.

❑ Addressable Memory Locations:

- A memory chip with k-bit addressing has 2^k memory locations.
- These locations are numbered from 0 to 2^k- 1, which is referred to as the **memory space**.
- **Example:** For a 4-bit address, the possible addresses range from 0000 to 1111 in binary, or 0 to 15 in decimal, which translates to 0 to $2^4 – 1$.

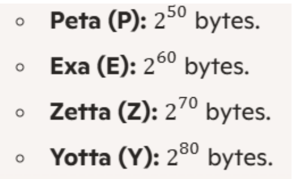
❑ Memory Units and Conversion:

- 1 Kilobyte (KB): 1K=2^{10}=1024 bytes

- 1 Megabyte (MB): 1MB = 1024KB = 1024 x 1024 = 2^{20} bytes
- 1 Gigabyte (GB): 1GB = 1024MB = 1024 x 2^{20} = 2^{30} bytes
- 1 Terabyte (TB) = 1024GB = 1024 x 2^{30} = 2^{40} bytes

❑ **Addressable Memory Examples:**

- 24-bit Memory: With 24 bits, the total addressable locations are 2^{24}. This is 2^{24} = 16 x 2^{20} = 16MB (since 1MB= 2^{20}).
- 32-bit Memory: With 32 bits, the total addressable locations are 2^{32}. This is 2^{32} = 4GB (since 1GB = 2^{30}, and 4G= 4 x 1G = 2^2 x 2^{30} = 2^{32}).
- Beyond terabytes, larger units are:



- **Peta (P):** 2^{50} bytes.
- **Exa (E):** 2^{60} bytes.
- **Zetta (Z):** 2^{70} bytes.
- **Yotta (Y):** 2^{80} bytes.

Memory Addressing

❑ **Practical Addressing:**

- It is impractical to give each individual bit in memory its own unique address.
- Instead, the most practical approach is to have successive addresses. Refers to successive byte locations in the memory **byte-addressable memory.** In this system, each byte (8 bits) has its own address.

❑ **Byte Addresses:**

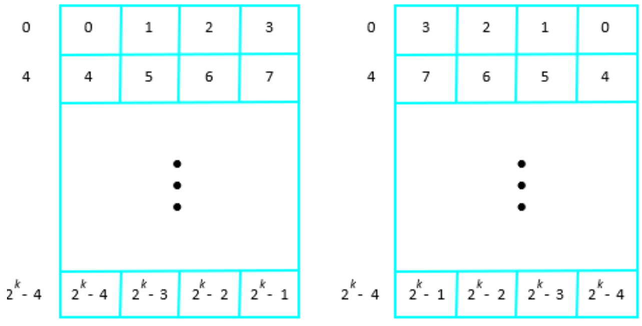
Byte locations have addresses 0,1,2,…

If word length is 32 bits (4 bytes), then successive words are located at addresses 0, 4, 8, ….

❑ **Big-Endian and Little-Endian Assignment of Memory Addresses:**

➢ Big-Endian Assignment: In Big-Endian memory addressing, the higher byte addresses are used for the least significant bytes of a word. This means that the most significant byte (MSB) is stored at the lowest memory address, and the least significant byte (LSB) is stored at the highest memory address.

➢ Little-Endian Assignment: In Little-Endian memory addressing, the lower byte addresses are used for the less significant bytes of a word. This means that the least significant byte (LSB) is stored at the lowest memory address, and the most significant byte (MSB) is stored at the highest memory address.



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

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

❑ **Ordering of bytes:** Little endian and Big endian schemes.

❑ **Word Alignment:** Words are said to be aligned in memory if they begin at a byte address that is a multiple of the number of bytes in a word.

- For different word lengths:
  - **16-bit word:** Word addresses are 0, 2, 4, 6, 8, etc. (each word is 2 bytes).

- ○ **32-bit word:** Word addresses are 0, 4, 8, 12, 16, etc. (each word is 4 bytes).
- ○ **64-bit word:** Word addresses are 0, 8, 16, 24, 32, etc. (each word is 8 bytes).

✓ Memory addresses are used to access different types of data, including numbers, characters, and strings.

Memory Operation

❑ **LOAD (Read or Fetch) Operation:**

- ♦ Purpose: Copy content from memory to a register without changing the memory content.
- ♦ Process:
  1) **Address Placement:** The CPU places the address of the desired data into the Memory Address Register (MAR).
  2) **Read Signal:** The CPU sends a read (RD) control signal to the memory chip.
  3) **Data Retrieval:** The CPU waits until the memory chip places the desired data into the Memory Data Register (MDR).

❑ **STORE (Write) Operation:**

- ♦ Purpose: Overwrite the content in memory with new data.
- ♦ Process:
  1) **Address and Data Placement:** The CPU places the address in the MAR and the data to be written in the MDR.
  2) **Write Signal:** The CPU sends a write (WR) control signal to the memory chip.
  3) **Memory Update:** Upon completing the write operation, the memory chip sends back a Memory Function Complete (MFC) signal to the CPU.

❖ Instruction and Instruction Sequencing

❑ **Must-Perform Computer Operations:**

- Data Transfers Between Memory and Processor Registers.
- Arithmetic and Logic Operations on Data.
- Program Sequencing and Control.
- I/O (Input/Output) Transfers.

❑ Computer instructions must be capable of performing 4 types of operations:

i. Data Transfer/Movement: To move data between memory and processor registers. E.g., memory read, memory write.

ii. Arithmetic and Logic Operations: To perform mathematical calculations and logical comparisons. E.g., addition, subtraction, comparison between two numbers.

iii. Program Sequencing and Flow of Control: To manage the order in which instructions are executed. E.g., Branch Instructions (Change the flow of execution based on conditions (e.g., if statements, loops).

iv. Input/Output Transfers: To transfer data between the computer and external devices.

❑ Examples of different types of instructions in assembly language notation:

- ♦ Data Transfers Between Processor and Memory:
  i) **Move A, B:** Transfers the content of A to B. (B = A)

ii) **Move A, R1:** Transfers the content of A to register R1. (R1 = A)

- Arithmetic and Logic Operations:
  Add A, B, C: Adds the contents of A and B, then stores the result in C(C = A + B).
- Sequencing:
  Jump Label (Jump to the subroutine which starts at Label).
- Input/Output Data Transfer:
  Input PORT, R5: Reads data from an input/output port labeled "PORT" and transfers it to register R5.

❑ **Register Transfer Notation(RTN):**

- Symbolic Names for Locations:
  - Denoted by symbols such as R0, R1, R2, etc. These always indicate register locations.
  - Denoted by any other symbols such as X, Y, Z, A, B, M, LOC, LOCA, LOCB. These indicate memory locations.
- To denote the contents of a specific location, square brackets are placed around the name of the location. Examples:
  - **R1 ← [LOC]:** This means that the contents of the memory location labeled LOC are transferred to register R1.
  - **R3 ← [R2]:** This means that the contents of register R2 are transferred to register R3.

❑ **Assembly Language Notation:**

- Represent machine instructions and programs.
- Instructions in assembly language consist of an opcode (operation code), a source operand, and a destination operand. The general format is: OP src_op dest_op.
- Examples:
  1) **MOV LOC, R1:** This instruction means moving the contents of memory location LOC to register R1. Equivalent in Register Transfer Notation (RTN): R1 ← [LOC].
  2) **ADD R1, R2, R3:** This instruction means adding the contents of registers R1 and R2, and storing the result in register R3. Equivalent in RTN: R3 ← [R1] + [R2].

❑ **CPU Organization: Internal Storage Architecture:**

- Controls how its instructions use the operand(s).
- The type of internal storage in a processor is the most basic differentiation.
- **Single Accumulator (AC) CPU Organization:**
  - In this organization, one of the operands for most instructions is implicitly the Accumulator Register.
  - The Accumulator serves as both an input operand and a storage location for the result of arithmetic and logic operations.
  - The Accumulator (AC) acts as an implicit input operand, meaning it is automatically used by the CPU without needing to specify it in the instruction.
  - The Accumulator holds the result of the operation, eliminating the need for an explicit result operand.
  - Since the Accumulator is used so frequently, it needs to be saved to memory often to preserve intermediate results.

- **Register-Memory CPU Organization:**
  - One operand comes from a register. The other operand is fetched directly from memory. The result is stored back into a register.
  - Memory can be accessed as part of any instruction.
- **Register-Register (Load-Store) CPU Organization:**
  - All operands for arithmetic and logical operations are stored in registers.
  - Operations do not directly access memory.
  - Memory is accessed only through dedicated load and store instructions.
- **Stack CPU Organization:**
  - **No Registers:** Unlike other CPU organizations, the Stack CPU does not use general-purpose registers. Instead, it relies on an internal stack memory to hold operands and results.
  - The TOS (Top of Stack) register points to the top of the stack, which contains the most recent operand. Operations in a Stack CPU combine the operand at the top of the stack with the one just below it. After an operation, the first operand is removed from the stack. The result of the operation takes the place of the second operand. The TOS is updated to point to the new result.
  - In a Stack CPU, all operands are implicit. This means that the operands for operations are always assumed to be at the top of the stack, and there is no need to specify them explicitly in instructions.

❑ **Instruction Formats:** In computer architecture, instruction formats are categorized based on the number of addresses or operands they use.

- **Three-Address Instructions:**
  - Format: ADD R2, R3, R1; Operation: R1 ← R2 + R3
- **Two-Address Instructions:**
  - Format: ADD R2, R1; Operation: R1 ← R1 + R2
- **One-Address Instructions** (typically used in Single Accumulator CPU organization):
  - The Accumulator Register (AC) is always an implicit operand.
  - Format: ADD M; Operation: AC ← AC + M[AR]
- **Zero-Address Instructions** (commonly used in Stack CPU organization):
  - No explicit operands are required; both operands are implicit.
  - **Format**: ADD; **Operation**: TOS ← TOS + (TOS – 1) (where TOS stands for Top of Stack)
- **RISC Instructions**:
  - Typically, an instruction can involve 3-4 registers.
  - Memory access operations are limited to LOAD and STORE instructions.

Each instruction consists of an opcode and operand(s) or address(es), depending on the specific format being used. This structure ensures efficient processing and execution of instructions within a CPU.

Example: Evaluate X ← (A+B) ∗ (C+D)

- **Three-Address Format**

```
ADD  A, B, R1     ; R1 ← M[A] + M[B] (M[A] = the contents in the memory address A)
ADD  C, D, R2     ; R2 ← M[C] + M[D]
MUL  R1, R2, X    ; M[X] ← R1 * R2
```

Example: Evaluate X←(A+B) ∗ (C+D)

- **Two-Address instruction format**

```
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
```

Why not instructions like ADD A,B to make like B ← A+B?

Answer: Because both operands can’t be memory locations, at least one must be register. Besides the content of B should not be overwritten (the programmer knows nothing about this side effect!)

Example: Evaluate X ← (A+B) ∗ (C+D)

- **One-Address Instruction Format**

```
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
```

Example: Evaluate X = (A+B) ∗ (C+D)

- **Zero-Address Instruction Format**

(Must use stack processor organization. TOS means Top of Stack)

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

Example: Evaluate X = (A+B) ∗ (C+D)

- **RISC Instruction Format**

(i) RISC can use 3 registers in a single instruction;
(ii) Only the LOAD and STORE instructions can access memory

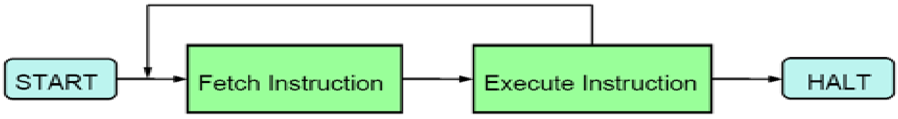
```
1.  LOAD A, R1          ; R1 ← M[A]
2.  LOAD B, R2          ; R2 ← M[B]
3.  LOAD C, R3          ; R3 ← M[C]
4.  LOAD D, R4          ; R4 ← M[D]
5.  ADD   R1, R2, R1    ; R1 ← R1 + R2
6.  ADD   R3, R4, R3    ; R3 ← R3 + R4
7.  MUL   R1, R3, R1    ; R1 ← R1 ∗ R3
8.  STORE R1, X         ; M[X] ← R1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

❑ **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:**

- Basic instruction cycle



i) **Start:** This is the beginning of the instruction cycle where the process is initiated.

ii) **Fetch Instruction**: In this stage, the CPU retrieves an instruction from memory. The Program Counter (PC) holds the address of the next instruction to be fetched. This instruction is then stored in the Instruction Register (IR).

iii) **Execute Instruction**: Once the instruction is fetched, the CPU decodes and executes it. This involves carrying out the operation specified by the instruction, such as arithmetic operations, data movement, or control operations.

iv) **HALT**: After executing the instruction, the cycle can either end if there are no more instructions to execute, or it can loop back to the Fetch Instruction stage if there are additional instructions.

The cycle continues until the CPU reaches a HALT instruction or an instruction that ends the program.

❑ **Straight-line sequencing:**

In straight-line sequencing, the processor control circuits use the information in the program counter (PC) to fetch and execute instructions, one at a time, in the increasing order of memory addresses.

❑ Instruction Execution and Straight-Line Sequencing:

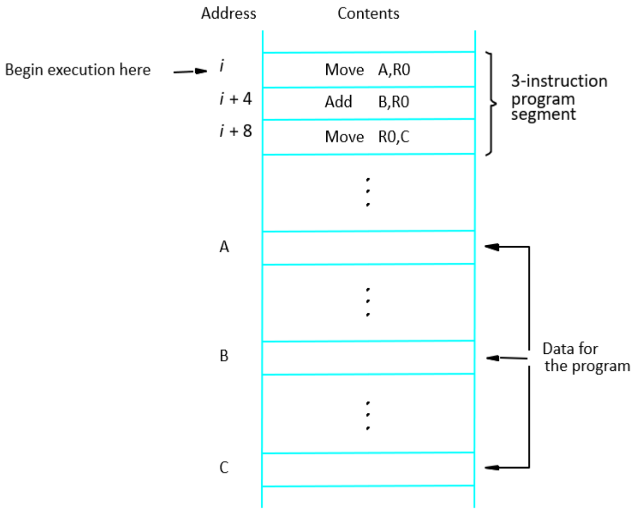


Figure 2.8. A program for C ← [A] + [B].

➢ **Assumptions**:

- **One Memory Operand per Instruction**: Each instruction accesses only one memory location.
- **32-bit Word Length**
- **Memory is Byte Addressable**
- **Each Instruction Fits in One Word**: Each instruction can be directly specified within a single 32-bit word (although this may not always be realistic).

➢ **Two-Phase Procedure**:

- **Instruction Fetch**: The CPU fetches the instruction from memory.
- **Instruction Execute**: The CPU executes the fetched instruction.

❑ Fetch/Execute Cycle:

- The execution of an instruction takes place in two distinct phases:
  - a) Instruction Fetch
  - b) Instruction Execute
- **Instruction Fetch**:
  - ✓ The CPU fetches the instruction from the memory location whose address is stored in the Program Counter (PC).
  - ✓ The fetched instruction is then placed in the Instruction Register (IR).
- **Instruction Execute**:
  - ✓ The instruction in the IR is examined and decoded to determine which operation is to be performed.
  - ✓ The necessary operands are fetched from memory or registers.
  - ✓ The specified operation is executed by the CPU.
  - ✓ The results of the operation are stored in the designated destination location (either a register or a memory location).
- The basic fetch/execute cycle repeats indefinitely, allowing the CPU to continuously fetch and execute instructions in sequence.

❑ Branching

Branch instructions are used to change the flow of execution in a program. When a branch instruction is executed, it loads a new value into the program counter (PC). This new value is known as the branch target address.

As a result, the processor fetches and executes the instruction located at the branch target address instead of the instruction that follows the branch instruction in the sequential address order. This allows for conditional and unconditional jumps within the program, facilitating loops, conditional execution, and function calls.

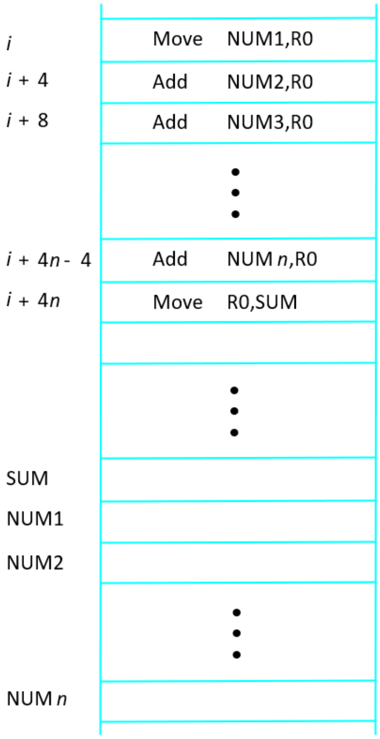
Branching is essential for creating dynamic and flexible programs that can respond to different inputs and conditions during execution.

❑ Conditional Branching

Conditional branching in computer science refers to a type of branch instruction that changes the flow of execution based on a specified condition. If the condition is met, the program counter (PC) is updated with the branch target address, causing the execution to jump to a different part of the program. If the condition is not met, the PC increments as usual, and the next sequential instruction is executed.

This mechanism allows for decision-making in programs, enabling the implementation of loops, conditional statements (like if-else), and function calls. Conditional branching is essential for creating dynamic programs that can respond to different situations during runtime.

★ Branching
Adding an array of numbers without using any Loop (straight line program)



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. A straight-line program for adding n numbers.

❑

Branching

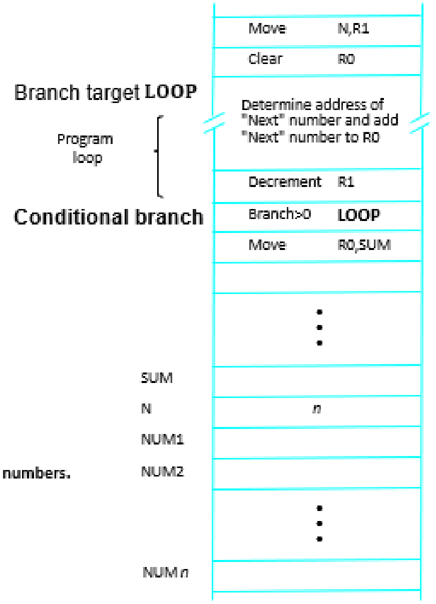


Figure 2.10. Using a loop to add *n* numbers.

1) Initialize Registers: Move the value of n (the number of elements) to register R1. Clear register R0 to prepare it for summation.
2) Loop Start (Branch Target): This is the point where the program begins looping.
3) Add Next Number: Determine the address of the next number in the sequence and add this number to R0.
4) Decrement Counter: Decrease the value in R1 by 1.
5) Conditional Branch: If R1 is greater than 0, branch back to the start of the loop (Branch Target). This ensures the loop continues until all numbers are added.
6) End of Loop: Once all numbers are added (R1 equals 0), move the total sum from R0 to the memory location labeled SUM.

This method ensures that the program effectively adds all numbers in the array using a loop and a conditional branch.

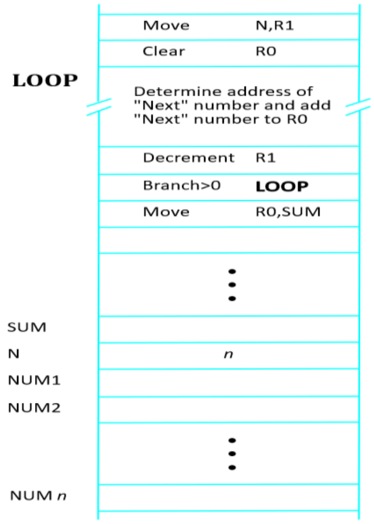
❑ Indirect Addressing to Compute the Sum of an Array

Indirect addressing allows the use of a register to hold the address of the operand, rather than the operand itself. This technique is useful when working with arrays or lists of data in memory. The program example demonstrates how to sum the elements of an array using indirect addressing in assembly language.

Book Examples (Fig. 2.12): Indirect Addressing to Compute the Array Sum

```
Address     Contents

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

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



```
Move    N,R1       ; Load the number of elements into R1
Move    #NUM1,R2   ; Load the address of the first element into R2
Clear   R0         ; Clear R0 to hold the sum

LOOP:
Add     (R2),R0    ; Add the value at the address in R2 to R0
Add     #4,R2      ; Move to the next element in the array
Decrement R1       ; Decrease the counter R1
Branch>0 LOOP      ; If R1 > 0, repeat the loop

Move    R0,SUM     ; Store the result in SUM
```

❑ Condition Code Flags (Status Flags)

Condition code flags, also known as status flags, are special bits in a status register within the CPU. These flags provide information about the result of the most recent arithmetic or logical operation performed by the Arithmetic Logic Unit (ALU). They are crucial for decision-making processes in programs, such as branching and conditional execution.

i) N (Negative) or S (Sign) Flag:

- This flag is set to 1 if the result of the most recent arithmetic operation is negative.
- It is used by instructions to determine if the result is less than zero, for example, **Branch<0** LOOP.

ii) Z (Zero) Flag:

- This flag is set if the result of the most recent arithmetic or logical operation is zero.
- It is used by instructions to check if the result is zero, for instance, **Branch==0** LABEL.

iii) C (Carry) Flag:

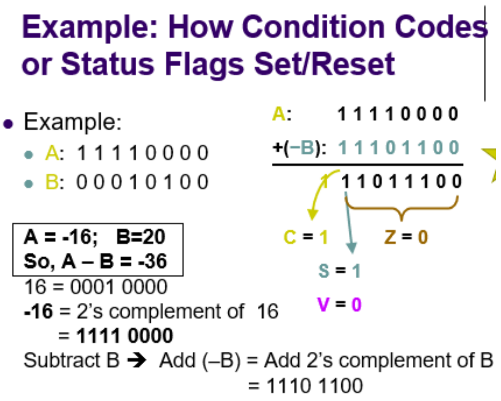
- This flag is set if there is a carry out from the most recent arithmetic operation (e.g., addition).
- It indicates that the result of an addition operation exceeds the maximum value that can be stored in the register.

iv) V (Overflow flag):

- This flag is set if Overflow occurs in most recent op.

Different instructions affect different flags.

❑



Example: How Condition Codes or Status Flags Set/Reset

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

A = -16;   B=20
So, A – B = -36

16 = 0001 0000
-16 = 2's complement of 16
= **1111 0000**
Subtract B ➔ Add (–B) = Add 2's complement of B
= 1110 1100

Analyzing the Flags from the picture:

Carry Flag (C): Set if there was a carry out of the most significant bit (for unsigned numbers). In this case, C = 1.

Zero Flag (Z): Set if the result is zero. Here, Z = 0 because the result is not zero.

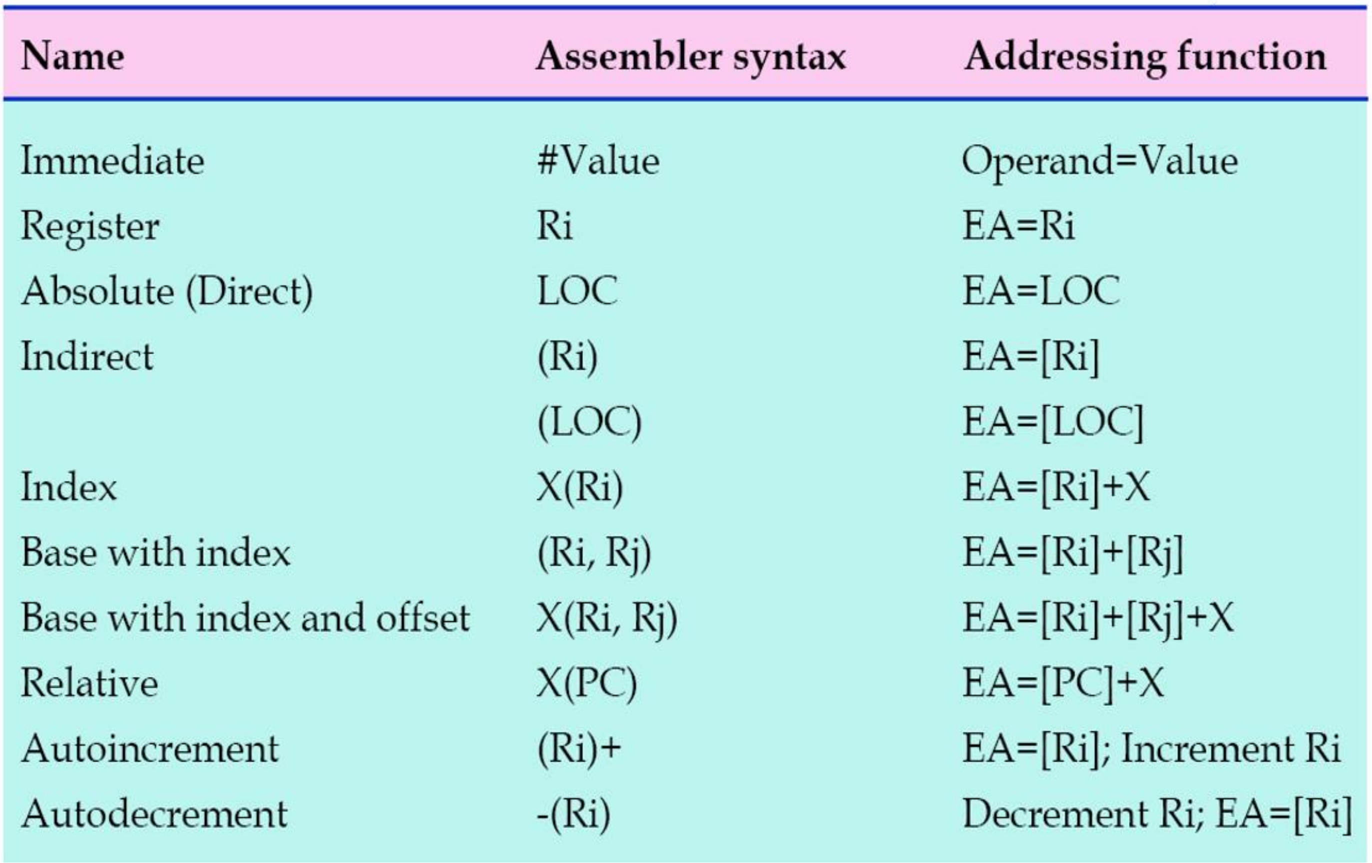
Sign Flag (S): Set if the result is negative (most significant bit is 1). Here, S = 1 because the result is negative.

Overflow Flag (V): Set if there was an overflow, meaning the sign of the result is incorrect. In this case, V = 0 because the sign of the result is correct.

Addressing Modes

- Data Structures: Programmers often use various data structures such as lists, linked lists, arrays, and queues to represent data that will be utilized in computations. These structures help in organizing and managing data efficiently.
- High-Level Language: In high-level programming languages, constructs like constants, local and global variables, pointers, and arrays are used to handle data. These languages provide an abstraction that makes programming easier and more intuitive compared to machine-level programming.
- Translation to Assembly Language: When a program written in a high-level language is compiled into assembly language, the compiler must implement these high-level constructs using the computer's instruction set. This translation process is essential for executing the program on a physical machine.
- Addressing modes define the different ways in which the location of an operand is specified in an instruction. These modes are crucial for the execution of instructions as they determine how the operand is accessed.

Generic Addressing Modes



Name	Assembler syntax	Addressing function
Immediate	#Value	Operand=Value
Register	Ri	EA=Ri
Absolute (Direct)	LOC	EA=LOC
Indirect	(Ri)	EA=[Ri]
	(LOC)	EA=[LOC]
Index	X(Ri)	EA=[Ri]+X
Base with index	(Ri, Rj)	EA=[Ri]+[Rj]
Base with index and offset	X(Ri, Rj)	EA=[Ri]+[Rj]+X
Relative	X(PC)	EA=[PC]+X
Autoincrement	(Ri)+	EA=[Ri]; Increment Ri
Autodecrement	-(Ri)	Decrement Ri; EA=[Ri]

Addressing Modes

Different ways in which the address of an operand is specified in an instruction is referred to as addressing modes.

1) **Register mode:**
    - In this mode, the operand is the contents of a processor register.
    - The address of the register (its name) is specified within the instruction.
    - E.g. *Clear R1* *or* Move R1, R2
2) **Absolute mode:**
    - In this mode, the operand resides in a memory location.
    - The address of the memory location is explicitly provided in the instruction.
    - E.g. *Clear A* *or* Move LOC, R2
    - This mode is also referred to as "Direct Mode" in some assembly languages.

Register and absolute modes can be used to represent variables.

3) **Immediate Mode:**
    - The operand is specified explicitly within the instruction.
    - E.g. Move #200, R0
        - ○ Here, the operand 200 is directly given, and it is moved to register R0.
    - Used to represent constants.

★ Register, Absolute and Immediate modes contained either the address of the operand or the operand itself.

★ Some instructions provide information from which the memory address of the operand can be determined

- ○ That is, they provide the “Effective Address” of the operand
- ○ They do not provide the operand or the address of the operand explicitly.

★ Different ways in which “Effective Address” of the operand can be generated.

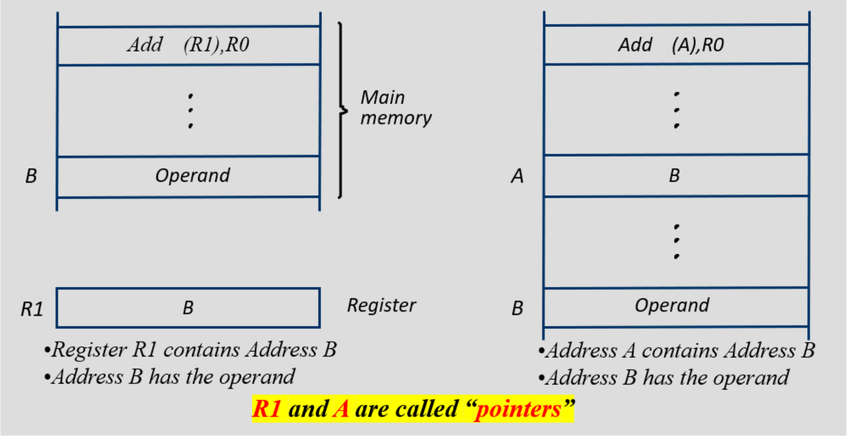
❑ Addressing modes define how the operand of an instruction is selected. Some instructions do not provide the operand or its address explicitly; instead, they provide information to calculate the "Effective Address" of the operand.

- **Indirection and Pointers**
    - ♦ **Indirect Addressing Mode:**
        - ▪ The effective address of the operand is the value found in a register or memory location. The address of this register or memory location is given in the instruction.
        - ▪ Indirection is shown by putting the register name or memory address in parentheses in the instruction.
        - ▪ A register or memory location holding the address of an operand is called a **pointer**.

    Example: Move (5000), R0

    - ○ Here, the instruction points to memory location 5000. The effective address is obtained by fetching the address stored in memory location 5000, which then points to the actual operand.

Effective Address of the operand is the contents of a register or a memory location whose address appears in the instruction.



•Register R1 contains Address B
•Address B has the operand

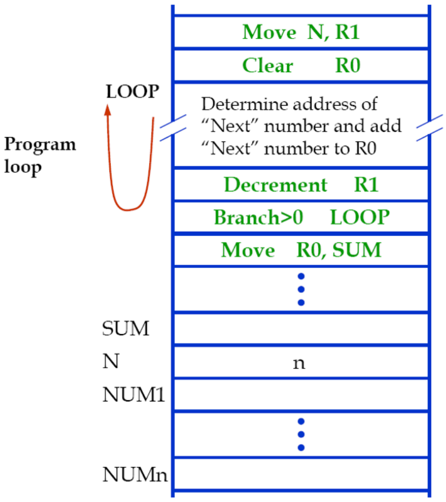
•Address A contains Address B
•Address B has the operand

R1 and A are called “pointers”

This is called as “Indirect Mode”

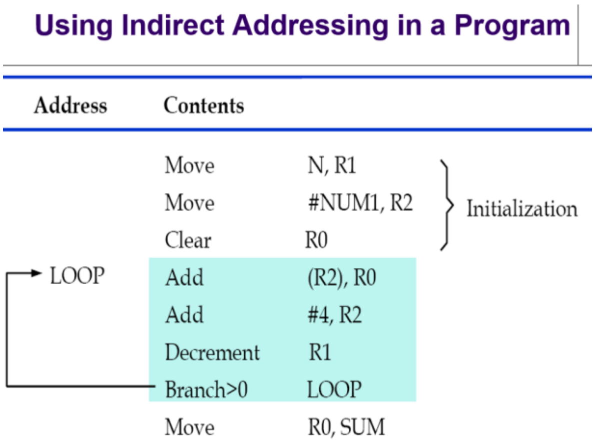
❑

Using Indirect Addressing in a Program



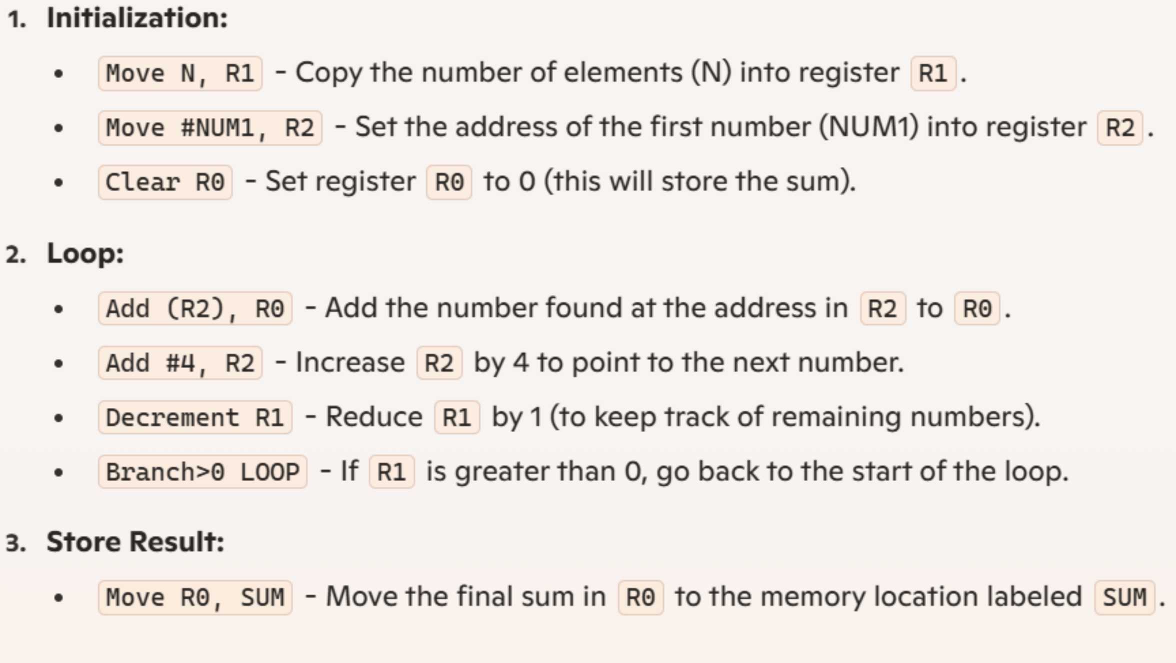
Initialize: Move the value of N (number of elements) into register R1. Clear register R0 (this will store the sum). **Loop**: Determine the address of the "Next" number using indirect addressing. Add the "Next" number to R0 (sum). Decrease the value in R1. **Repeat**: If R1 is greater than 0, go back to the start of the loop (Branch condition). **Store Result**: Move the final sum from R0 to the memory location labeled SUM.

❑



Using Indirect Addressing in a Program

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



1. **Initialization:**
   - `Move N, R1` - Copy the number of elements (N) into register `R1`.
   - `Move #NUM1, R2` - Set the address of the first number (NUM1) into register `R2`.
   - `Clear R0` - Set register `R0` to 0 (this will store the sum).
2. **Loop:**
   - `Add (R2), R0` - Add the number found at the address in `R2` to `R0`.
   - `Add #4, R2` - Increase `R2` by 4 to point to the next number.
   - `Decrement R1` - Reduce `R1` by 1 (to keep track of remaining numbers).
   - `Branch>0 LOOP` - If `R1` is greater than 0, go back to the start of the loop.
3. **Store Result:**
   - `Move R0, SUM` - Move the final sum in `R0` to the memory location labeled `SUM`.

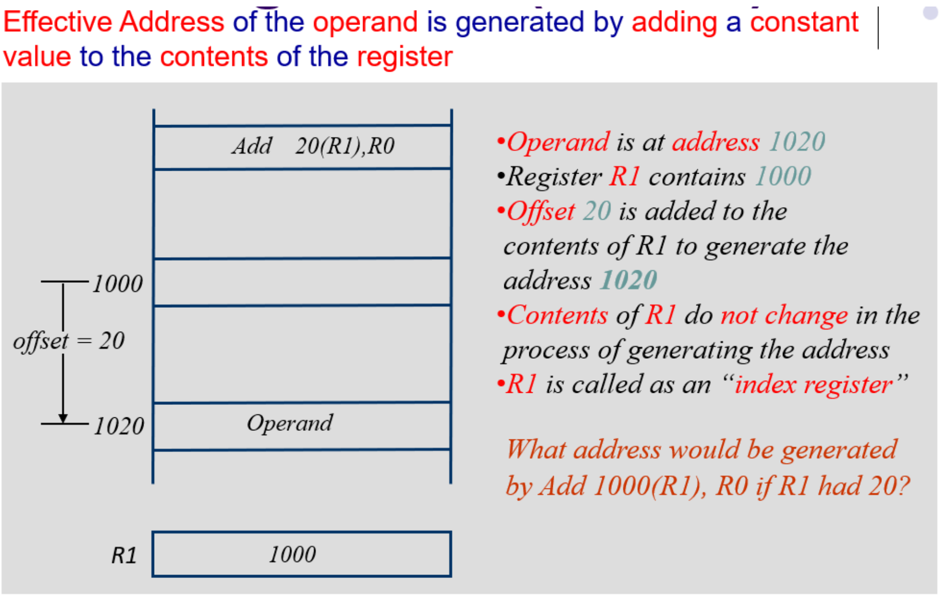
- ❑ **Indexing and Arrays**

Index mode: Adds a constant value to a register's contents to generate the operand's effective address.

Register used: Can be a special register or a general-purpose register in the processor.

Index register: The register used for this purpose.

- ❑ **Indexing and Arrays**
  - ▪ The index mode is useful in dealing with lists and arrays
  - ▪ We denote the Index mode symbolically as X(Ri), where X denotes the constant value contained in the instruction and Ri is the name of the register involved.
  - ▪ The effective address of the operand is given by EA=X+(Ri).
  - ▪ The contents of the index register are not changed in the process of generating the effective address



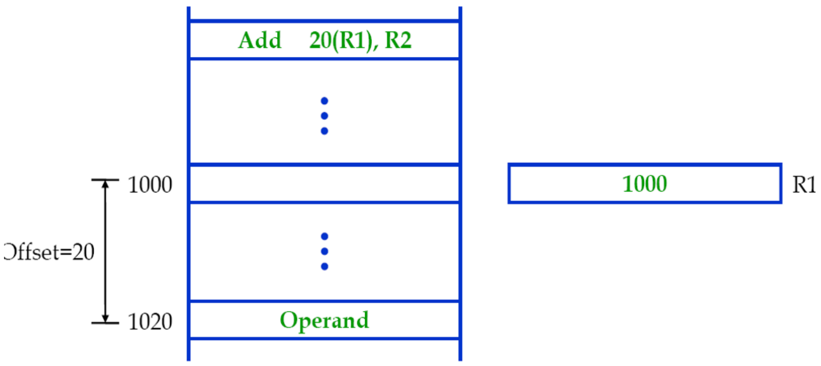
Effective Address of the operand is generated by adding a constant value to the contents of the register

•*Operand is at address 1020*
•*Register R1 contains 1000*
•*Offset 20 is added to the contents of R1 to generate the address 1020*
•*Contents of R1 do not change in the process of generating the address*
•*R1 is called as an "index register"*

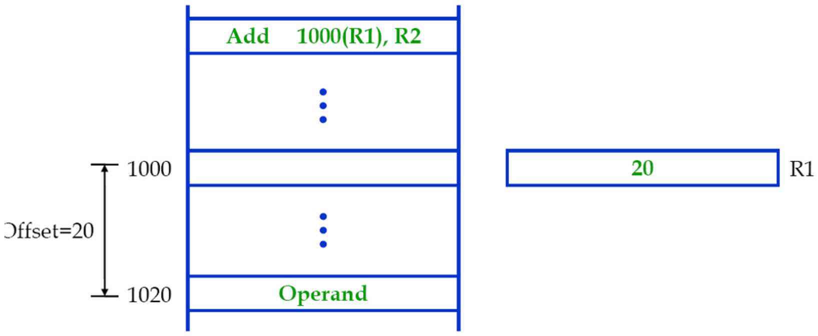
What address would be generated by Add 1000(R1), R0 if R1 had 20?

This is the "Indexing Mode"

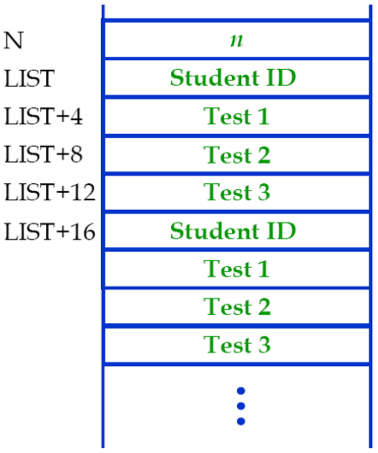
Offset is given as a constant



Offset is in the index register



An Example for Indexed Addressing



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

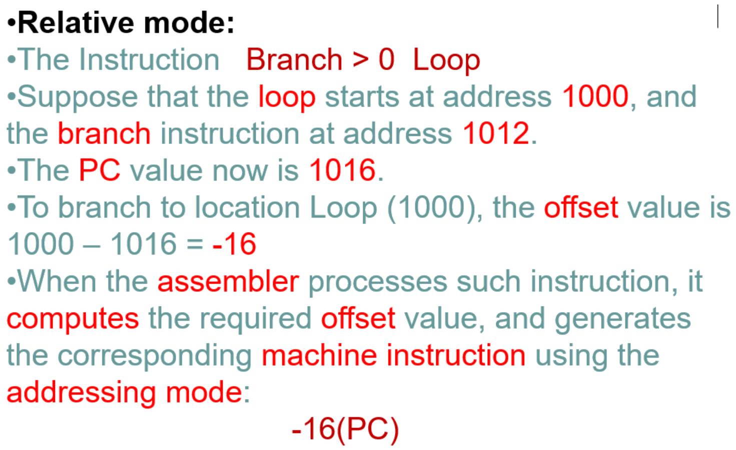
- ❑ **Variations of Indexed Addressing Mode**
  - ➢ **Using a Second Register for Offset:**
    - ✓ A second register may be used to contain the offset XX, in which case the Index mode can be written as (Ri,Rj)(Ri, Rj).
    - ✓ The effective address is the sum of the contents of registers RiRi and RjRj.
    - ✓ The second register is usually called the base register.
    - ✓ This mode implements a two-dimensional array.

- **Using Two Registers Plus a Constant:**
  - ✓ Another version of the Index mode uses two registers plus a constant, which can be denoted as X(Ri,Rj)X(Ri, Rj).
  - ✓ The effective address is the sum of the constant XX and the contents of registers RiRi and RjRj.
  - ✓ This mode implements a three-dimensional array.

❑ Relative Mode

Relative Addressing Mode is a method in computer architecture where the effective address of an operand is determined by adding a constant value to the contents of the Program Counter (PC). The Program Counter holds the address of the next instruction to be executed in the program. This mode is a variation of the Indexing Mode, where the index register is the PC instead of a general-purpose register.

The Relative Addressing Mode is particularly useful for specifying target addresses in branch instructions. In this mode, the addressed location is "relative" to the current value of the PC, hence the name "Relative Mode."



•Relative mode:

•The Instruction   Branch > 0 Loop

•Suppose that the loop starts at address 1000, and the branch instruction at address 1012.

•The PC value now is 1016.

•To branch to location Loop (1000), the offset value is 1000 – 1016 = -16

•When the assembler processes such instruction, it computes the required offset value, and generates the corresponding machine instruction using the addressing mode:

-16(PC)

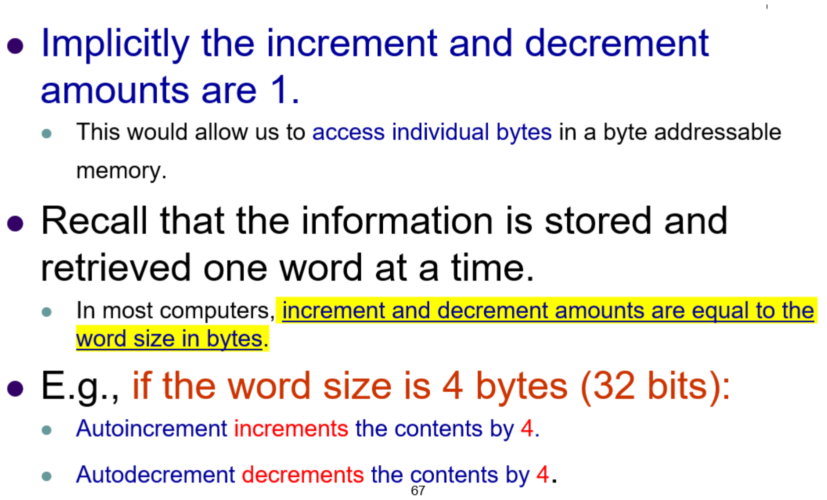
In summary, the effective address in Relative Addressing Mode is determined by adding an offset to the current value of the Program Counter. This mode is particularly useful for specifying target addresses in branch instructions.

❑ **Autoincrement Mode:** Autoincrement Mode is an addressing mode in computer architecture where the effective address of the operand is determined by 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 consecutive memory location. This mode is useful for iterating over arrays or other data structures.
Notation: (R1)+
Example: If register R1R1 contains the address 1000, the operand at address 1000 will be accessed first. After accessing the operand, the register R1R1 is incremented to 1001.

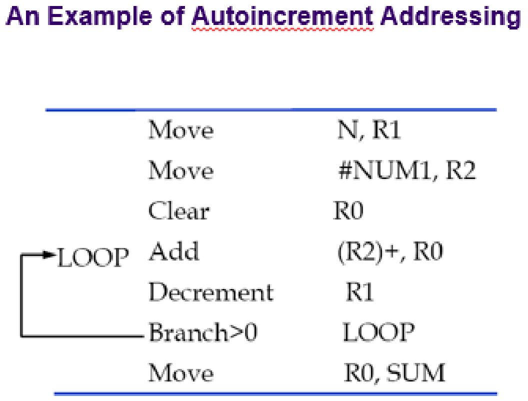
❑ **Autodecrement Mode:** Autodecrement Mode is an addressing mode where the effective address of the operand is determined by the contents of a register specified in the instruction. Before accessing the operand, the contents of this register are automatically decremented to point to the previous consecutive memory location. This mode is useful for implementing Last-In-First-Out (LIFO) data structures like stacks.
Notation: -(R1)

Example: If register R1R1 contains the address 1001, the register R1R1 is decremented to 1000 first, and then the operand at address 1000 will be accessed.

Autoincrement and Autodecrement modes are useful for implementing "Last-In-First-Out" data structures.



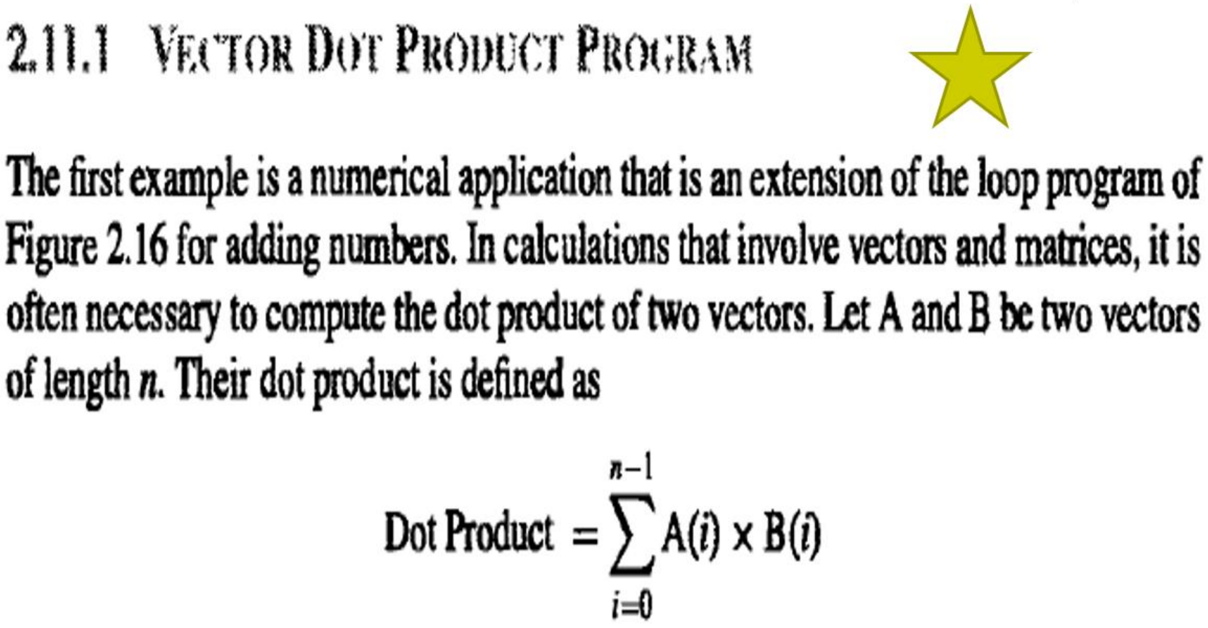
- Implicitly the increment and decrement amounts are 1.
  - This would allow us to access individual bytes in a byte addressable memory.
- Recall that the information is stored and retrieved one word at a time.
  - In most computers, increment and decrement amounts are equal to the word size in bytes.
- E.g., if the word size is 4 bytes (32 bits):
  - Autoincrement increments the contents by 4.
  - Autodecrement decrements the contents by 4.



An Example of Autoincrement Addressing

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

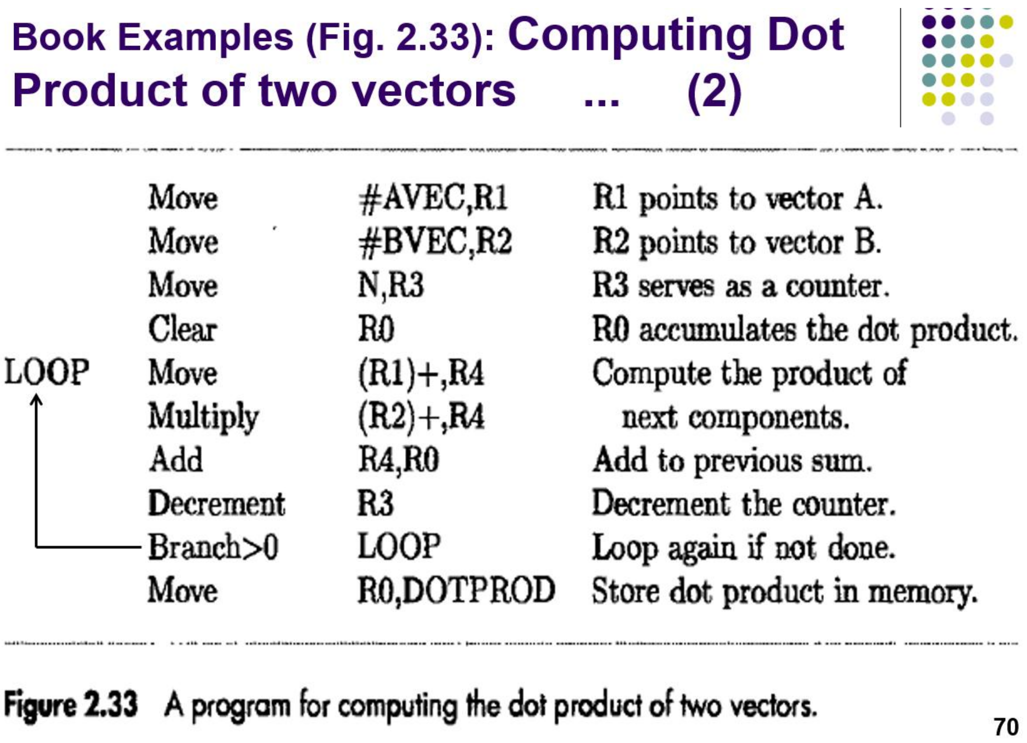
Book Examples (Fig. 2.33): Computing Dot Product of two vectors ... (1)



2.11.1 Vector Dot Product Program

The first example is a numerical application that is an extension of the loop program of Figure 2.16 for adding numbers. In calculations that involve vectors and matrices, it is often necessary to compute the dot product of two vectors. Let A and B be two vectors of length *n*. Their dot product is defined as

$$\text{Dot Product} = \sum_{i=0}^{n-1} A(i) \times B(i)$$



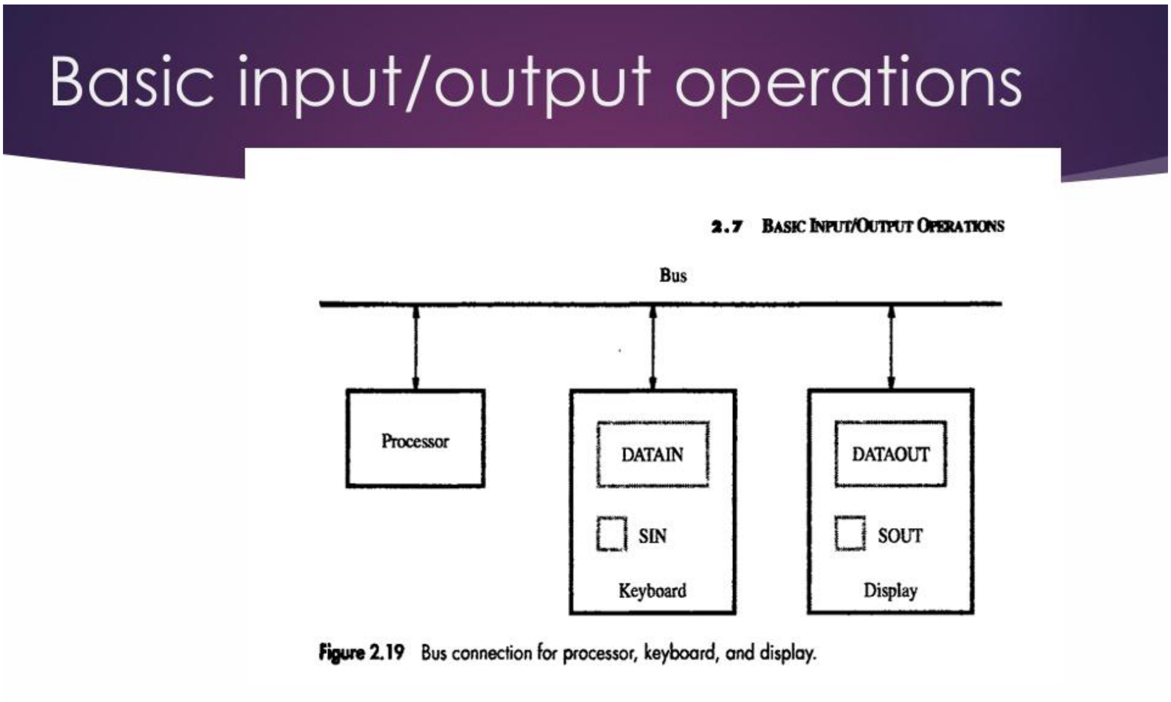
Book Examples (Fig. 2.33): Computing Dot Product of two vectors ... (2)

```
        Move        #AVEC,R1      R1 points to vector A.
        Move        #BVEC,R2      R2 points to vector B.
        Move        N,R3          R3 serves as a counter.
        Clear       R0            R0 accumulates the dot product.
LOOP    Move        (R1)+,R4      Compute the product of
        Multiply    (R2)+,R4        next components.
        Add         R4,R0         Add to previous sum.
        Decrement   R3            Decrement the counter.
        Branch>0    LOOP          Loop again if not done.
        Move        R0,DOTPROD    Store dot product in memory.
```

Figure 2.33 A program for computing the dot product of two vectors.

70

Chapter 2 (part 2)



Basic input/output operations

2.7 Basic Input/Output Operations

Figure 2.19 Bus connection for processor, keyboard, and display.

- For reading input from a keyboard or other devices and giving output to display, a method known as program-controlled I/O is used

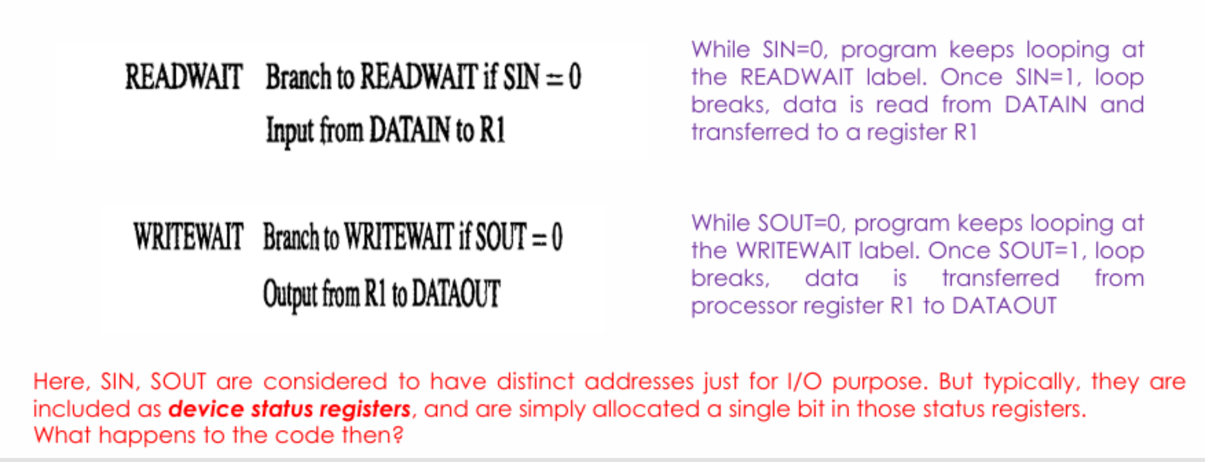
- I/O devices are much slower than processors, so synchronization is done using registers and signals

- Simply striking a key does not mean it will automatically show up on the display screen

- Keyboard and display has 8-bit buffer registers, called DATAIN and DATAOUT respectively

- When a key is struck on the keyboard, its corresponding character code is stored in the DATAIN register, and a status control flag called SIN is set to 1(one). An internal program constantly monitors SIN, and when it sees that SIN=1, it reads from DATAIN and transfers it to the processor. After reading is completed, SIN is reset to 0.

- An analogous process occurs while writing to display. DATAOUT and SOUT are used. SOUT = 1 means display is ready to receive a character. When an internal program notices that SOUT=1, it transfers a character code to DATAOUT, and SOUT is set to 0. When display device is again ready to receive a character, SOUT=1 is set again.



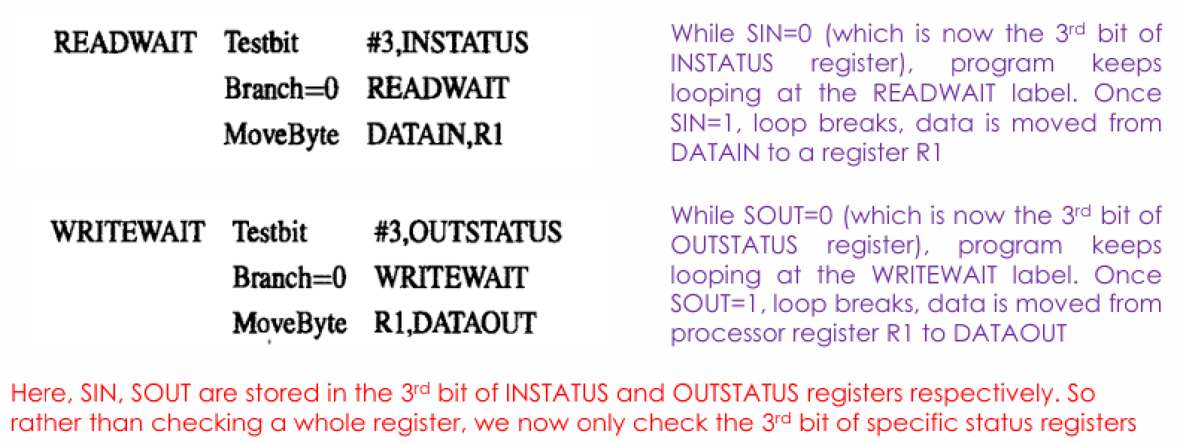
```
READWAIT   Branch to READWAIT if SIN = 0
           Input from DATAIN to R1
```

While SIN=0, program keeps looping at the READWAIT label. Once SIN=1, loop breaks, data is read from DATAIN and transferred to a register R1

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

While SOUT=0, program keeps looping at the WRITEWAIT label. Once SOUT=1, loop breaks, data is transferred from processor register R1 to DATAOUT

Here, SIN, SOUT are considered to have distinct addresses just for I/O purpose. But typically, they are included as ***device status registers***, and are simply allocated a single bit in those status registers. What happens to the code then?



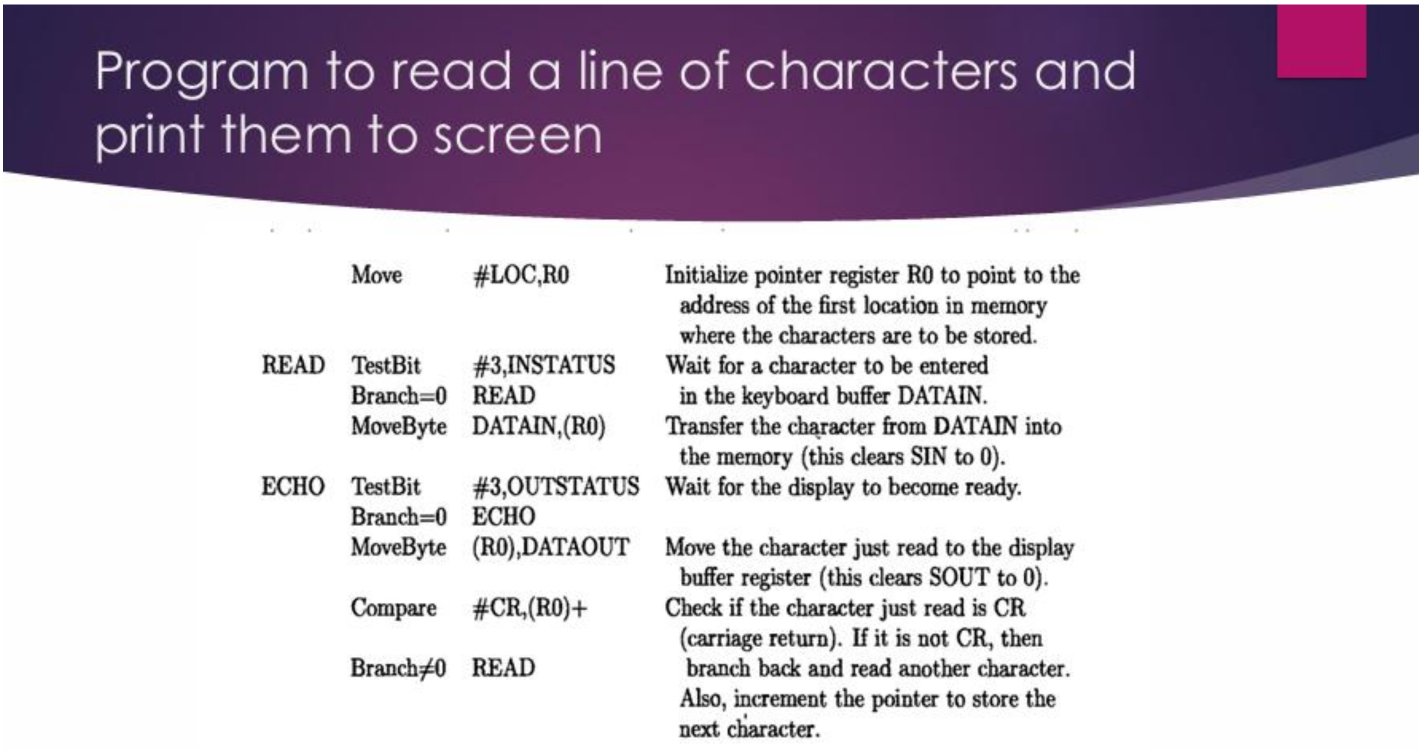
```
READWAIT   Testbit    #3,INSTATUS
           Branch=0   READWAIT
           MoveByte   DATAIN,R1
```

While SIN=0 (which is now the 3rd bit of INSTATUS register), program keeps looping at the READWAIT label. Once SIN=1, loop breaks, data is moved from DATAIN to a register R1

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

While SOUT=0 (which is now the 3rd bit of OUTSTATUS register), program keeps looping at the WRITEWAIT label. Once SOUT=1, loop breaks, data is moved from processor register R1 to DATAOUT

Here, SIN, SOUT are stored in the 3rd bit of INSTATUS and OUTSTATUS registers respectively. So rather than checking a whole register, we now only check the 3rd bit of specific status registers

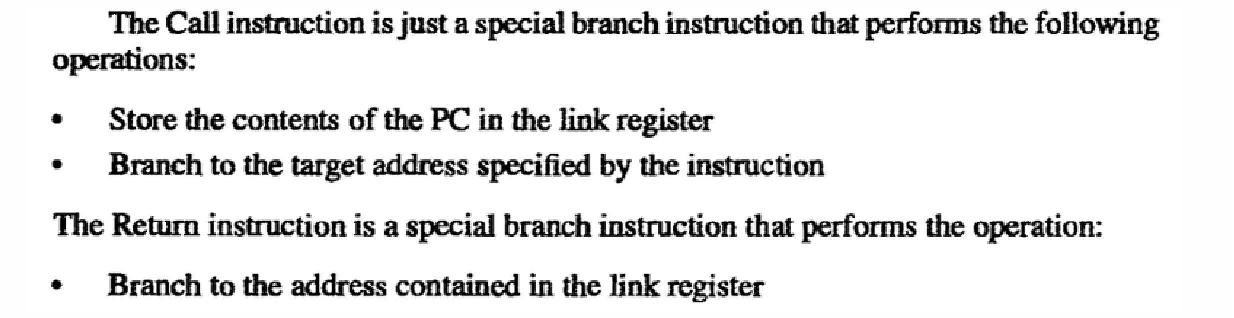


Program to read a line of characters and print them to screen

```
       Move       #LOC,R0         Initialize pointer register R0 to point to the
                                    address of the first location in memory
                                    where the characters are to be stored.
READ   TestBit    #3,INSTATUS     Wait for a character to be entered
       Branch=0   READ              in the keyboard buffer DATAIN.
       MoveByte   DATAIN,(R0)     Transfer the character from DATAIN into
                                    the memory (this clears SIN to 0).
ECHO   TestBit    #3,OUTSTATUS    Wait for the display to become ready.
       Branch=0   ECHO
       MoveByte   (R0),DATAOUT    Move the character just read to the display
                                    buffer register (this clears SOUT to 0).
       Compare    #CR,(R0)+       Check if the character just read is CR
                                    (carriage return). If it is not CR, then
       Branch≠0   READ              branch back and read another character.
                                    Also, increment the pointer to store the
                                    next character.
```

❑ Subroutine

- If it is necessary to perform a particular subtask many times for different data, this can be done using a subroutine.
- For example – sorting different data values, calculating factorial for different values etc.
- When a program branches to a subroutine, it is known as calling the subroutine
- The instruction that performs this branching operation is named as Call instruction
- After the subroutine execution is completed, the calling program needs to resume execution and continues from its paused state. So, the subroutine needs to return to the program that called it using Return Instruction.
- The way in which computers make it possible to call and return from subroutines is called "Subroutine Linkage" method
- Simplest "Linkage" method is to just save return address to a register which is referred to as the Link register. This could be a special register just for this purpose or simply a general register. When the subroutine is completed, 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 that performs the operation:

- Branch to the address contained in the link register

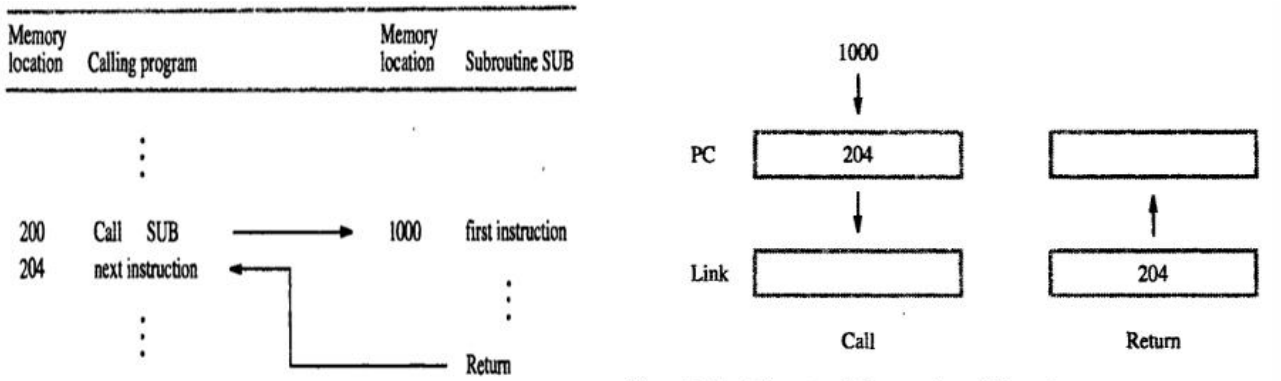


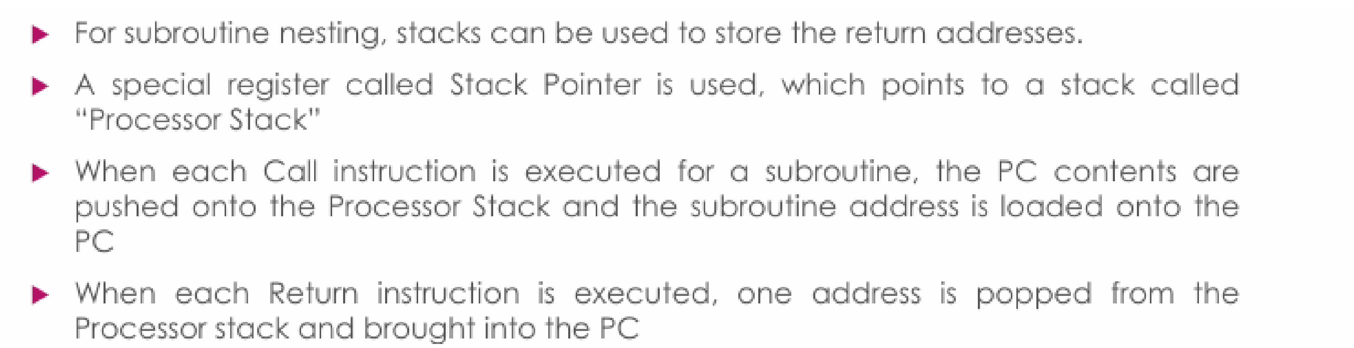
Figure 2.24 Subroutine linkage using a link register.

❑ Subroutine Nesting

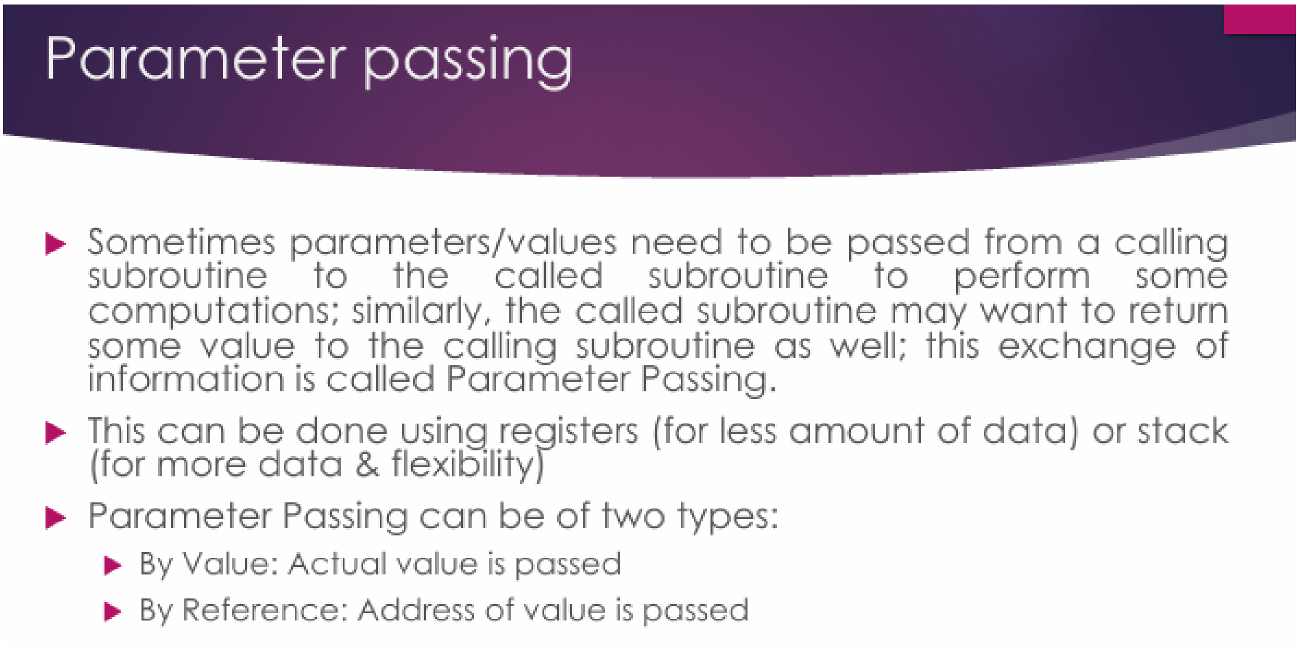
- When one subroutine calls another subroutine, it is called subroutine nesting.
- If we use the original link register to repeat the calling process, the return address of the 1st calling subroutine will be lost. Hence, we cannot use the same link register if we are to perform subroutine nesting. We must store the return address in some other location.
- Subroutine nesting may take place to any levels or depth (2, 3, 4 etc.). When the last nested subroutine finishes execution, then it starts returning to the calling subroutines.
- The return addresses are stored and used in a Last In First Out (LIFO) order

- Which data structure do we know that utilizes a LIFO order?
  The data structure that utilizes a LIFO order is a stack. Stacks are used to store return addresses in subroutine nesting to ensure that each subroutine can return to its caller in the correct sequence.

❑ **Processor Stack**

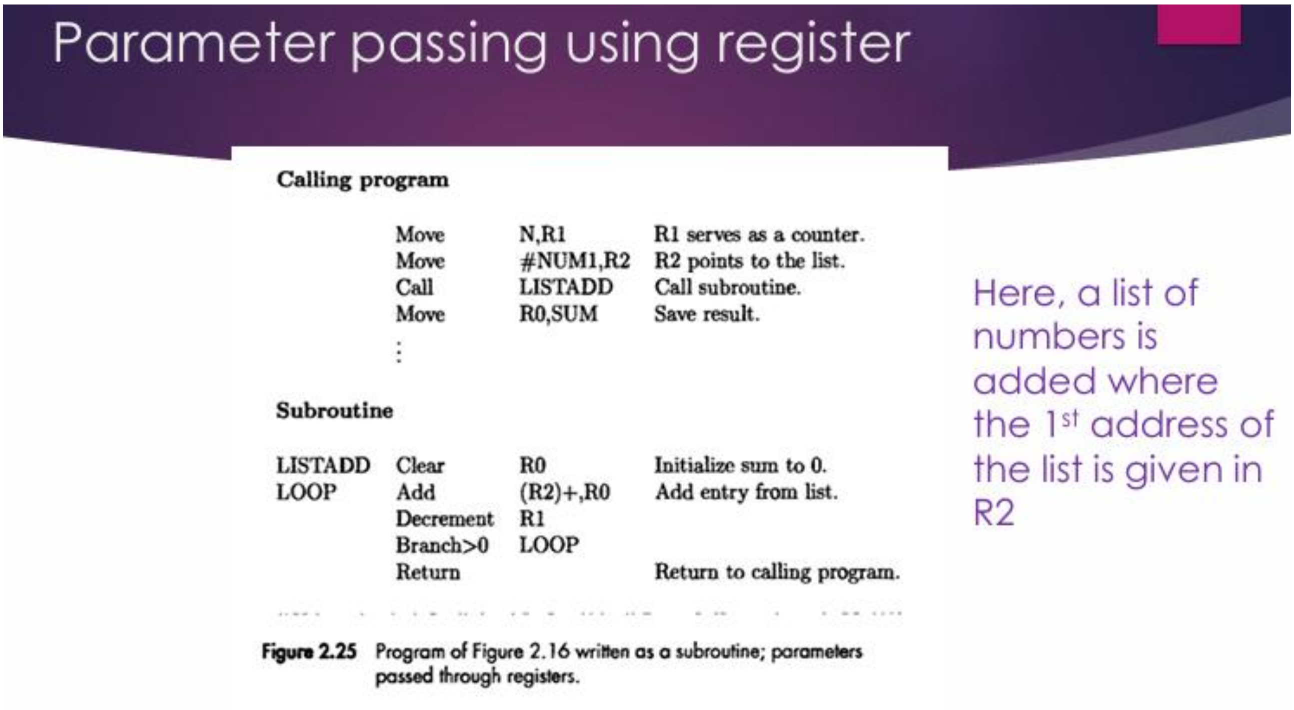


- For subroutine nesting, stacks can be used to store the return addresses.
- A special register called Stack Pointer is used, which points to a stack called "Processor Stack"
- When each Call instruction is executed for a subroutine, the PC contents are pushed onto the Processor Stack and the subroutine address is loaded onto the PC
- When each Return instruction is executed, one address is popped from the Processor stack and brought into the PC



Parameter passing

- Sometimes parameters/values need to be passed from a calling subroutine to the called subroutine to perform some computations; similarly, the called subroutine may want to return some value to the calling subroutine as well; this exchange of information is called Parameter Passing.
- This can be done using registers (for less amount of data) or stack (for more data & flexibility)
- Parameter Passing can be of two types:
  - By Value: Actual value is passed
  - By Reference: Address of value is passed



Parameter passing using register

```
Calling program

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

Subroutine

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

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

Here, a list of numbers is added where the 1st address of the list is given in R2

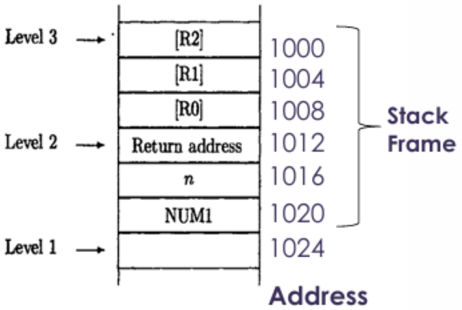
Parameter passing using Stack frame

Assume top of stack is at level 1 below.

```
            Move          #NUM1,-(SP)   Push parameters onto stack.
            Move          N,-(SP)
            Call          LISTADD       Call subroutine
                                          (top of stack at level 2).
            Move          4(SP),SUM     Save result.
            Add           #8,SP         Restore top of stack
                                          (top of stack at level 1).
            :
LISTADD     MoveMultiple  R0-R2,-(SP)   Save registers
                                          (top of stack at level 3).
            Move          16(SP),R1     Initialize counter to n.
            Move          20(SP),R2     Initialize pointer to the list.
            Clear         R0            Initialize sum to 0.
LOOP        Add           (R2)+,R0      Add entry from list.
            Decrement     R1
            Branch>0      LOOP
            Move          R0,20(SP)     Put result on the stack.
            MoveMultiple  (SP)+,R0-R2   Restore registers.
            Return                      Return to calling program.
```

(a) Calling program and subroutine

Here, a list of numbers is added as well, but stack is used to pass parameters



(b) Top of stack at various times

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

Stack frame

Now, observe how space is used in the stack in the example in Figure 2.26. 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, 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*. If the subroutine requires more space for local memory variables, they can also be allocated on the stack.

Stack Pointer, Frame pointer

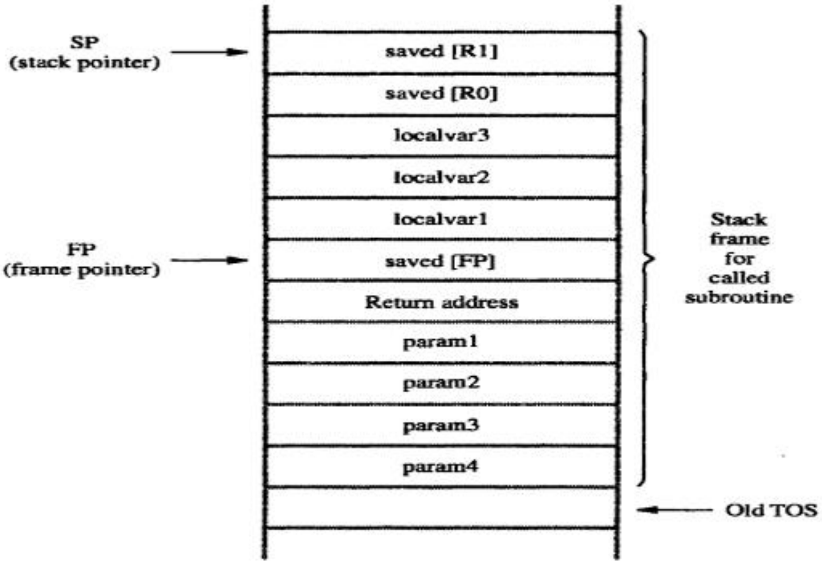


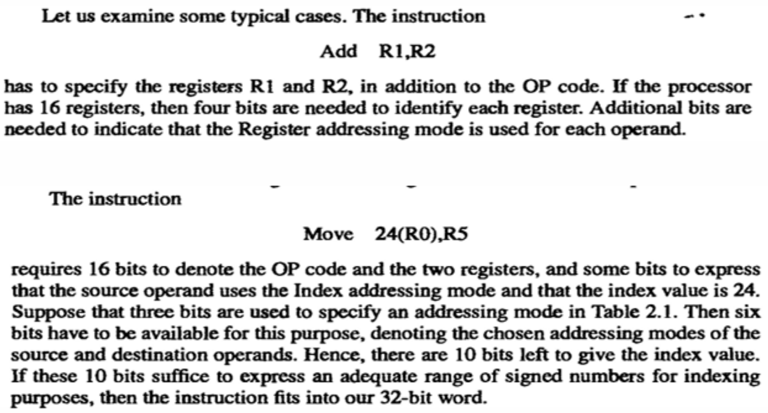
Figure 2.27 A subroutine stack frame example.

Stack Pointer, Frame pointer

- Stack Pointer register points to the top of the stack, whereas Frame Pointer register points to the location just above the stored return address.
- While SP is constantly changing to point to the current TOS, FP remains fixed. As a result, the parameters and local variables of the called subroutine can be easily accessible using Index Addressing Mode

Self-Study from pdf of CARL HAMACHER book pg. 78 to 80

ENCODING OF MACHINE INSTRUCTIONS



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.

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. Suppose that three bits are used to specify an addressing mode in Table 2.1. Then six bits have to be available for this purpose, denoting the chosen addressing modes of the source and destination operands. Hence, there are 10 bits left to give the index value. If these 10 bits suffice to express an adequate range of signed numbers for indexing purposes, then the instruction fits into our 32-bit word.

Different instructions may need different amount of storage space when they are encoded from assembly language to machine instructions. Here, the 1st instruction has 2 registers and an op-code, while the 2nd has 2 registers, a number and an op-code

ENCODING OF MACHINE INSTRUCTIONS

- If we are to enforce a certain word size (16/32 etc.) for every instruction of a computer, that might cause parts of instructions to be left out.

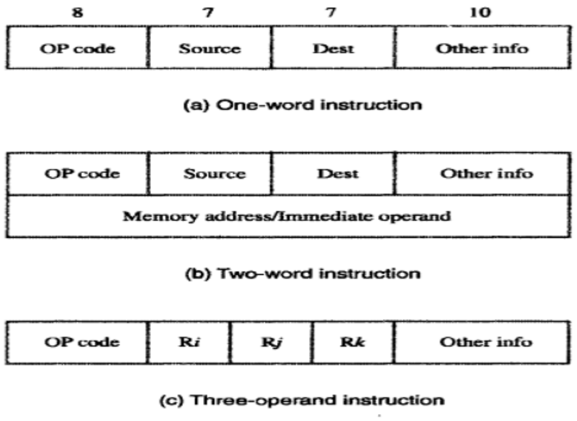


Figure 2.39 Encoding instructions into 32-bit words.

We could use variable size of instruction set, so that some instructions might use 16 bits (one word), while others use 32 bits (2 words) or as needed. This type of processors are referred to as CISC. We could have complex instructions as needed in this system.

However, if we enforce the instruction size in such a way that each instruction must occupy only 1 word length, that would be called RISC. As a result, instructions would be limited and simpler.

Oboseshe chapter ti sesh holo!^_^