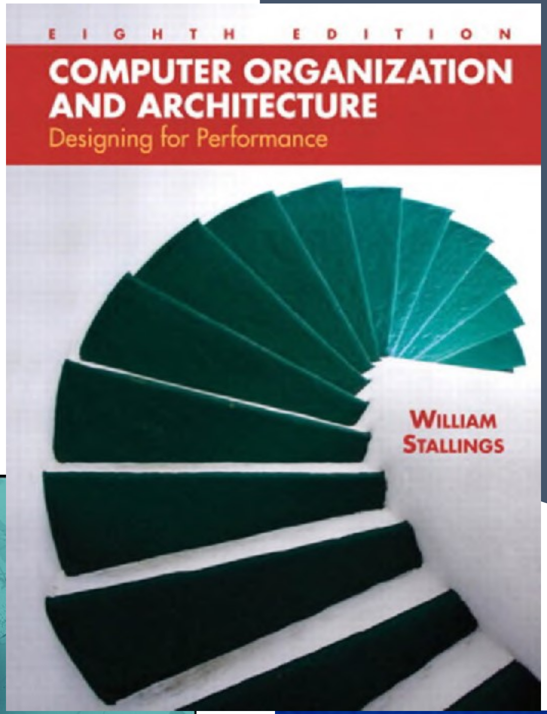
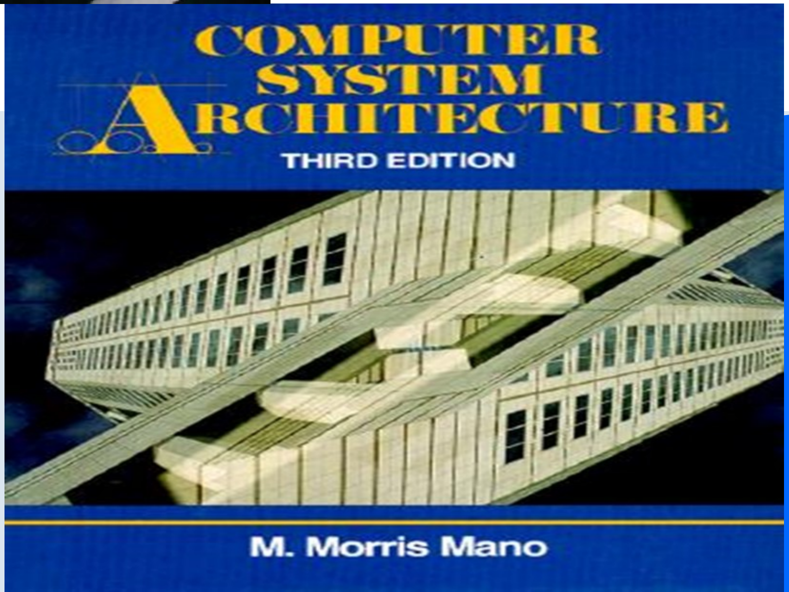
Computer Architecture

2nd Class, Computer Science Dept.

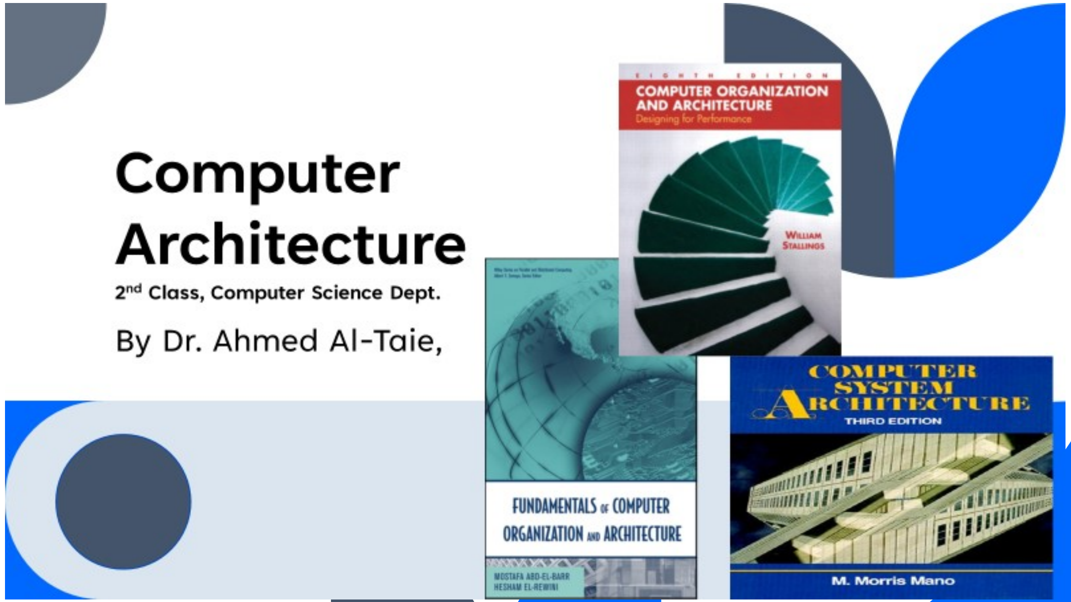
By Dr. Ahmed Al-Taie,





Chapter 2
Instruction Set Architecture and Design
Outline

- Memory Locations and Operations
- Addressing Modes
- Instruction Types
- Programming Examples



Instruction Types

The **type of instructions** **forming** the **instruction set** of a **machine** is an **indication** of the **power of** the **underlying architecture** of **the machine**.

- Instructions can in general be classified as in the following Subsections

1-Data Movement Instructions

Data movement instructions are **used to move data among the different units** of the machine.

- Most notably among these are **instructions** that are **used to move data among the different registers in the CPU**.
- A simple **register to register movement** of **data** can be made through the instruction
- **MOVE Ri,Rj .**
- This instruction moves the content of register Ri to register Rj. The effect of the instruction is to override the contents of the **(destination) register Rj** without changing the contents of the (**source**) **register Ri.**

Instruction Types

- **Data movement instructions** include those used to **move** data to (from) **registers** from (to) **memory.**
- These instructions are usually referred to as the **load and store** instructions, respectively. **Examples of the two instructions** are

LOAD 25838, Rj
STORE Ri, 1024

TABLE 2.3 Some Common Data Movement Operations

Data movement operation	Meaning
MOVE	Move data (a word or a block) from a given source (a register or a memory) to a given destination
LOAD	Load data from memory to a register
STORE	Store data into memory from a register
PUSH	Store data from a register to stack
POP	Retrieve data from stack into a register

- The **first** instruction **loads** the **content** of the **memory location** whose **address is 25838** into the destination **register Rj.** The **content** of the **memory location** is **unchanged** by executing the **LOAD** instruction.

The STORE instruction stores the content of the source register Ri into the memory location 1024.

- The **content** of the **source register** is **unchanged** by executing the **STORE** instruction.

Table 2.3 shows some common data transfer operations and their meanings

Instruction Types

Arithmetic and Logical Instructions.

- **Arithmetic and logical instructions** are those **used** to **perform arithmetic and logical manipulation of registers and memory contents**.
- **Examples of arithmetic instructions** include the **ADD and SUBTRACT** instructions.
- These are

  ADD R1,R2,R0

  SUBTRACT R1,R2,R0

- The **first instruction adds** the **contents** of **source registers R1 and R2** and **stores** the **result** in **destination register R0**.
- The **second instruction subtracts** the **contents** of the **source registers R1 and R2** and **stores** the **result** in the **destination register R0**.
- The **contents** of the **source registers** are unchanged by the **ADD** and the **SUBTRACT** instructions.
- In addition to the **ADD and SUBTRACT** instructions, some machines have **MULTIPLY** and **DIVIDE** instructions.

Instruction Types

Arithmetic and Logical Instructions.

- In addition to the **ADD and SUBTRACT** instructions, some machines have **MULTIPLY** and **DIVIDE** instructions.
- These two instructions are **expensive to implement** and **could** be **substituted by** the use of **repeated addition or repeated subtraction**.
- Therefore, most modern architectures do not have **MULTIPLY or DIVIDE** instructions on their instruction set.

- **Table 2.4 shows some common arithmetic operations and their meanings.**

TABLE 2.4 Some Common Arithmetic Operations

Arithmetic operations	Meaning
ADD	Perform the arithmetic sum of two operands
SUBTRACT	Perform the arithmetic difference of two operands
MULTIPLY	Perform the product of two operands
DIVIDE	Perform the division of two operands
INCREMENT	Add one to the contents of a register
DECREMENT	Subtract one from the contents of a register

Instruction Types

Arithmetic and Logical Instructions

- **Logical instructions** are used to **perform** logical operations such as **AND, OR, SHIFT, COMPARE**, and **ROTATE.**
- As the **names indicate**, these instructions **perform**, respectively, **and, or, shift, compare, and rotate operations** on **register or memory contents**.

- **Table 2.5 presents a number of logical operations.**

TABLE 2.5 Some Common Logical Operations

Logical operation	Meaning
AND	Perform the logical ANDing of two operands
OR	Perform the logical ORing of two operands
EXOR	Perform the XORing of two operands
NOT	Perform the complement of an operand
COMPARE	Perform logical comparison of two operands and set flag accordingly
SHIFT	Perform logical shift (right or left) of the content of a register
ROTATE	Perform logical shift (right or left) with wraparound of the content of a register

Instruction Types

Sequencing Instructions

- Control (sequencing) instructions are used to change the sequence in which instructions are executed.
- They take the form of CONDITIONAL BRANCHING (CONDITIONAL JUMP), UNCONDITIONAL BRANCHING (JUMP), or CALL instructions.

A common characteristic among these instructions is that their execution changes the program counter (PC) value.

- The change made in the PC value can be unconditional, for example, in the unconditional branching or the jump instructions.
- In this case, the earlier value of the PC is lost and execution of the program starts at a new value specified by the instruction.
- Consider, for example, the instruction JUMP NEW-ADDRESS. Execution of this instruction will cause the PC to be loaded with the memory location represented by NEW-ADDRESS whereby the instruction stored at this new address is executed. On the other hand

Instruction Types

Sequencing Instructions

- On the other hand, the change made in the PC by the branching instruction can be conditional based on the value of a specific flag.
- Examples of these flags include the Negative (N), Zero (Z), Overflow (V), and Carry (C).
- These flags represent the individual bits of a specific register, called the CONDITION CODE (CC) REGISTER.

The values of flags are set based on the results of executing different instructions.

- The meaning of each of these flags is shown in Table 2.6.

TABLE 2.6 Examples of Condition Flags

Flag name	Meaning
Negative (N)	Set to 1 if the result of the most recent operation is negative, it is 0 otherwise
Zero (Z)	Set to 1 if the result of the most recent operation is 0, it is 0 otherwise
Overflow (V)	Set to 1 if the result of the most recent operation causes an overflow, it is 0 otherwise
Carry (C)	Set to 1 if the most recent operation results in a carry, it is 0 otherwise

Instruction Types

Sequencing Instructions

- Consider, for example, the following group of instructions.
- LOAD #100, R1

Loop: ADD (R2) + , R0

DECREMENT R1

BRANCH-IF-GREATER-THAN Loop

- The fourth instruction is a conditional branch instruction, which indicates that if the result of decrementing the contents of register R1 is greater than zero, that is, if the Z flag is not set, then the next instruction to be executed is that labeled by Loop. It should be noted that conditional branch instructions could be used to execute program loops (as shown above).
- The CALL instructions are used to cause execution of the program to transfer to a subroutine. A CALL instruction has the same effect as that of the JUMP in terms of loading the PC with a new value from which the next instruction is to be executed. However, with the CALL instruction the incremented value of the PC (to point to the next instruction in sequence) is pushed onto the stack.

Instruction Types

Sequencing Instructions

- Execution of a RETURN instruction in the subroutine will load the PC with the popped value from the stack.
- This has the effect of resuming program execution from the point where branching to the subroutine has occurred.

- Figure 2.12 shows a program segment that uses the CALL instruction.

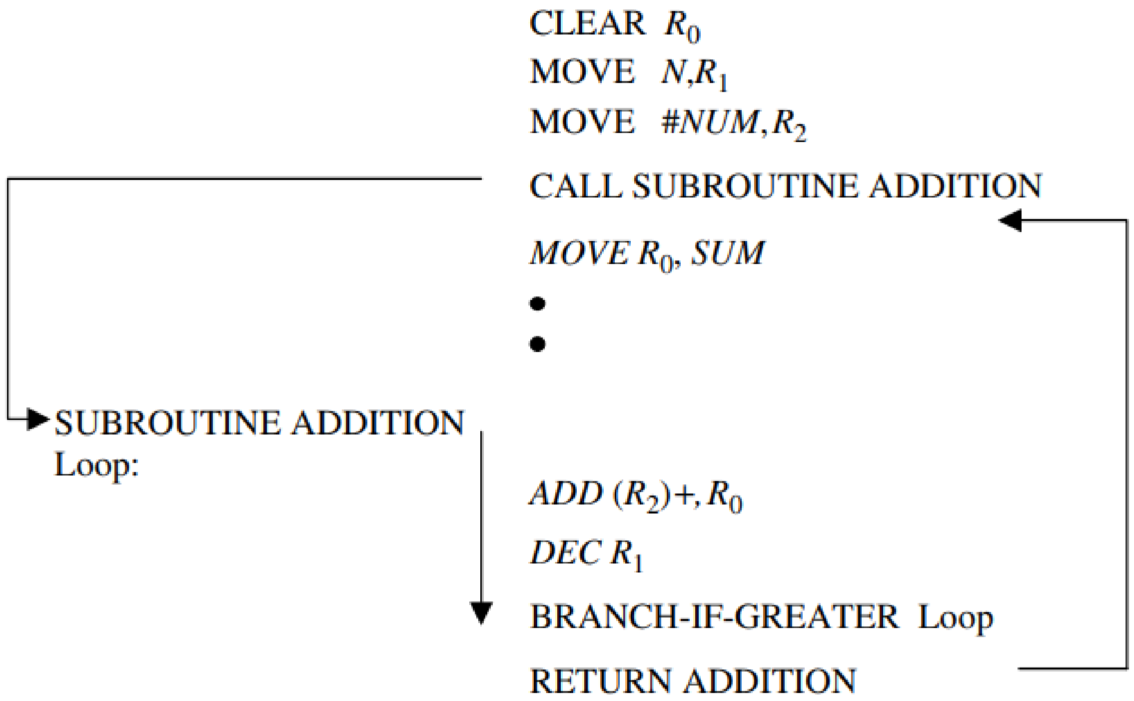


Figure 2.12 A program segment using a subroutine

Instruction Types

- This program segment sums up a number of values, N, and stores the result into memory location SUM.
- The values to be added are stored in N consecutive memory locations starting at NUM.
- The subroutine, called ADDITION, is used to perform the actual addition of values while the main program stores the results in SUM.

Table 2.7 presents some common transfer of control operations.

TABLE 2.7 Some Transfer of Control Operations

Transfer of control operation	Meaning
BRANCH-IF-CONDITION	Transfer of control to a new address if condition is true
JUMP	Unconditional transfer of control
CALL	Transfer of control to a subroutine
RETURN	Transfer of control to the caller routine

Instruction Types

Input/Output Instructions

- Input and output instructions (I/O instructions) are used to transfer data between the computer and peripheral devices. The two basic I/O instructions used are the INPUT and OUTPUT instructions.
- The INPUT instruction is used to transfer data from an input device to the processor.
- Examples of input devices include a keyboard or a mouse.
- Input devices are interfaced with a computer through dedicated input ports. Computers can use dedicated addresses to address these ports.
- Suppose that the input port through which a keyboard is connected to a computer carries the unique address 1000. Therefore, execution of the instruction INPUT 1000 will cause the data stored in a specific register in the interface between the keyboard and the computer, call it the input data register, to be moved into a specific register (called the accumulator) in the computer.

Instruction Types

Input/Output Instructions

- Similarly, the execution of the instruction OUTPUT 2000 causes the data stored in the accumulator to be moved to the data output register in the output device whose address is 2000.
- Alternatively, the computer can address these ports in the usual way of addressing memory locations. In this case, the computer can input data from an input device by executing an instruction such as MOVE Rin, R0. This instruction moves the content of the register Rin into the register R0.
- Similarly, the instruction MOVE R0, Rin moves the contents of register R0 into the register Rin, that is, performs an output operation.
- This latter scheme is called memory-mapped Input/Output. Among the advantages of memory-mapped I/O is the ability to execute a number of memory-dedicated instructions on the registers in the I/O devices in addition to the elimination of the need for dedicated I/O instructions. Its main disadvantage is the need to dedicate part of the memory address space for I/O devices.

PROGRAMMING EXAMPLES

- Having introduced addressing modes and instruction types, we now move on to illustrate the use of these concepts through a number of programming examples.
- In presenting these examples, generic mnemonics will be used.
- This is done in order to emphasize the understanding of how to use different addressing modes in performing different operations independent of the machine used.
- Applications of similar principles using real-life machine examples are presented in Chapter 3.

PROGRAMMING EXAMPLES

- **Example 1:** In this example, we would like to show a program segment that can be used to **perform** the task of adding **100 numbers** stored at consecutive memory locations starting at location 1000. The results should be stored in memory location 2000.

	CLEAR R_0;	$R_0 \leftarrow 0$
	MOVE # *100*, R_1;	$R_1 \leftarrow 100$
	CLEAR R_2;	$R_2 \leftarrow 0$
LOOP:	*ADD 1000*(R_2), R_0;	$R_0 \leftarrow R_0 + M\ (1000 + R_2)$
	INCREMENT R_2;	$R_2 \leftarrow R_2 + 1$
	DECREMENT R_1;	$R_1 \leftarrow R_1 - 1$
	BRANCH-IF > 0 *LOOP*;	*GO TO LOOP if contents of* $R_1 > 0$
	STORE R_0, *2000*;	$M(2000) \leftarrow R_0$

In this example, use has been made of immediate (MOVE #100, R1) and indexed (ADD 1000 (R2), R0) addressing.

PROGRAMMING EXAMPLES

- **Example 2: In this example autoincrement addressing will be used to perform the same task performed in Example 1**

	$CLEAR\ R_0;$	$R_0 \leftarrow 0$
	$MOVE\ \#100, R_1;$	$R_1 \leftarrow 100$
	$CLEAR\ R_2;$	$R_2 \leftarrow 0$
$LOOP:$	$ADD\ 1000(R_2)+, R_0;$	$R_0 \leftarrow R_0 + M\ (1000 + R_2)\ \&\ R_2 \leftarrow R_2 + 1$
	$DECREMENT\ R_1;$	$R_1 \leftarrow R_1 - 1$
	$BRANCH\text{-}IF > 0\ LOOP;$	$GO\ TO\ LOOP\ if\ contents\ of\ R_1 > 0$
	$STORE\ R_0, 2000;$	$M(2000) \leftarrow R_0$

- **As can be seen, a given task can be performed using more than one programming methodology.**
- **The method used by the programmer depends on his/her experience as well as the richness of the instruction set of the machine used.**
- **Note also that the use of the autoincrement addressing in Example 2 has led to a decrease in the number of instructions used to perform the same task.**

PROGRAMMING EXAMPLES

- **Example 3: This example illustrates the use of a subroutine, SORT, to sort N values in ascending order (Fig. 2.13).**
  - The numbers are originally stored in a list starting at location 1000.
  - The sorted values are also stored in the same list and again starting at location 1000.
  - The subroutine sorts the data using the wellknown “Bubble Sort” technique.
  - The content of register R3 is checked at the end of every loop to find out whether the list is sorted or not.

```
MAIN PROGRAM    MOVE #999, R1                  ; R1 ← 999
                MOVE N, R0                     ; R0 ← N
                CALL SUBROUTINE SORT

SUBROUTINE SORT

Again:          CLEAR R3                       ; R3 ← 0

Next:           COMPARE (R1), (R1+1)           ; Compare current and next values
                BRANCH_IF_LOWER Skip           ; Branch if R1 lower than R1 + 1
                MOVE (R1), R2                  ; If not, Swap the contents of
                MOVE (R1 + 1), (R1)            ; the current location with the
                MOVE R2 , (R1 + 1)             ; next one.
                MOVE #1, R3                    ; Indicate the swap
Skip:           INCEMENT R1                    ; R1 ← R1 + 1
                DECREMENT R0                   ; R0 ← R0 – 1
                BRANCH_IF_GREATER Next         ; Go to next value

                COMPARE R3, 1                  ; Was there any swap?
                BRANCH_IF_EQUAL Again          ; Repeat process
                RETURN SORT
```

Figure 2.13 *SORT* subroutine

PROGRAMMING EXAMPLES

- **Example 3: This example illustrates the use of a subroutine, SORT, to sort N values in ascending order (Fig. 2.13).**
  - The numbers are originally stored in a list starting at location 1000.
  - The sorted values are also stored in the same list and again starting at location 1000.
  - The subroutine sorts the data using the wellknown “Bubble Sort” technique.
  - The content of register R3 is checked at the end of every loop to find out whether the list is sorted or not.

```
MAIN PROGRAM    MOVE #999, R1               ; R1 <- 999
                MOVE N, R0                  ; R0 <- N

                CALL SUBROUTINE SORT

     SUBROUTINE SORT

     Again:     CLEAR R3                    ; R3 <- 0

     Next:      COMPARE (R1), (R1+1)        ; Compare current and next values
                BRANCH_IF_LOWER Skip        ; Branch if R1 lower than R1 + 1
                MOVE (R1), R2               ; If not, Swap the contents of
                MOVE (R1 + 1), (R1)         ; the current location with the
                MOVE R2 , (R1 + 1)          ; next one.
                MOVE #1, R3                 ; Indicate the swap
     Skip:      INCEMENT R1                 ; R1 <- R1 + 1
                DECREMENT R0                ; R0 <- R0 – 1
                BRANCH_IF_GREATER Next      ; Go to next value

                COMPARE R3, 1               ; Was there any swap?
                BRANCH_IF_EQUAL Again       ; Repeat process
                RETURN SORT
```

Figure 2.13 *SORT* subroutine

PROGRAMMING EXAMPLES

- **Example 4: This example illustrates the use of a subroutine, SEARCH, to search for a value VAL in a list of N values (Fig. 2.14).**

  - We assume that the list is not originally sorted and therefore a brute force search is used.
  - In this search, the value VAL is compared with every element in the list from top to bottom.
  - The content of register R3 is used to indicate whether
  - VAL was found. The first element of the list is located at address 1000.

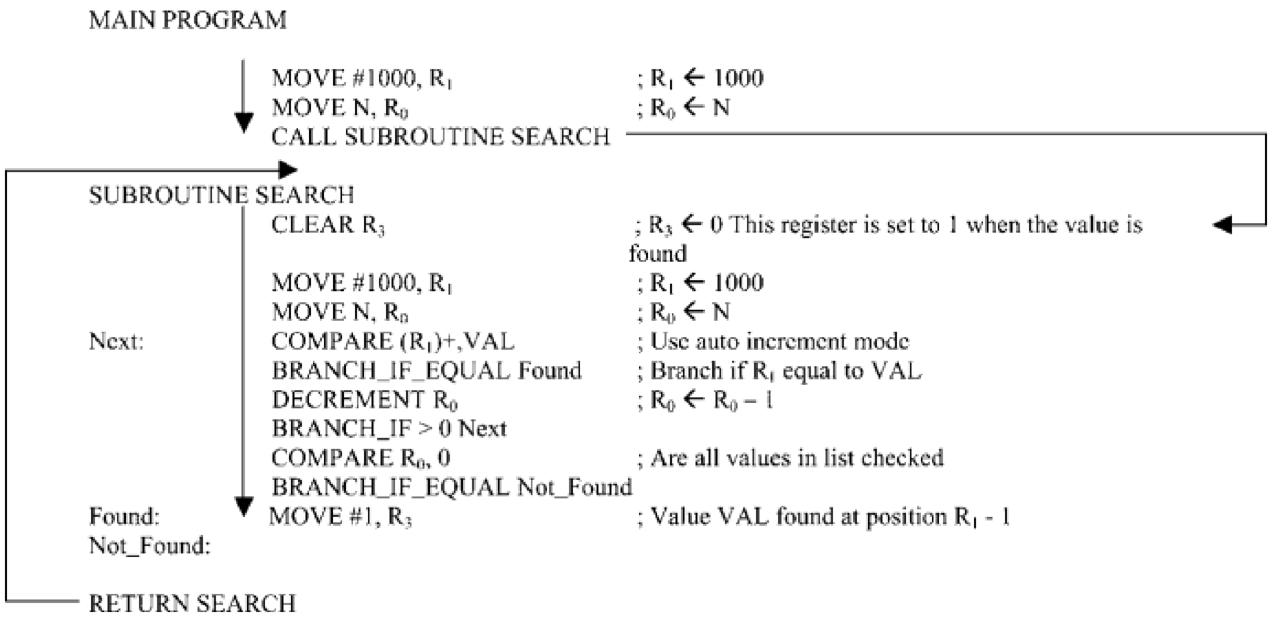


Figure 2.14 *SEARCH* subroutine

PROGRAMMING EXAMPLES

- **Example 5: This example illustrates the use of a subroutine, SEARCH, to search for a value VAL in a list of N values (as in Example 4) (Fig. 2.15).**

- **Here, we make use of the stack to send the parameters VAL and N.**

```
MAIN PROGRAM

              PUSH N
              PUSH VAL
              CALL SUBROUTINE SEARCH

SUBROUTINE SEARCH
              POP R_TEMP                    ; The top of the stack contains the value of the
                                    PC
              POP R_VAL
              POP R_N
              CLEAR R_3                     ; R_3 ← 0. This register is set to 1 checked
               ......
              BRANCH_IF_EQUAL Not_Found
Found:        MOVE #1, R_3                  ; Value VAL found at position R_1 - 1
Not_Found:    POP R_TEMP

          RETURN SEARCH
```

Figure 2.15 Subroutine *SEARCH* using stack to send parameters *VAL* and *N*

Thank you