Computer Organization and Architecture

Carl Hamacher, Zvonko Vranesic, Safwat Zaky,
Computer Organization, 5th Edition,
Tata McGraw Hill, 2002.

Machine Instructions and Programs – Part 2

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 way in which the location of an operand is specified in an instruction is called *addressing mode.*

Addressing Modes..

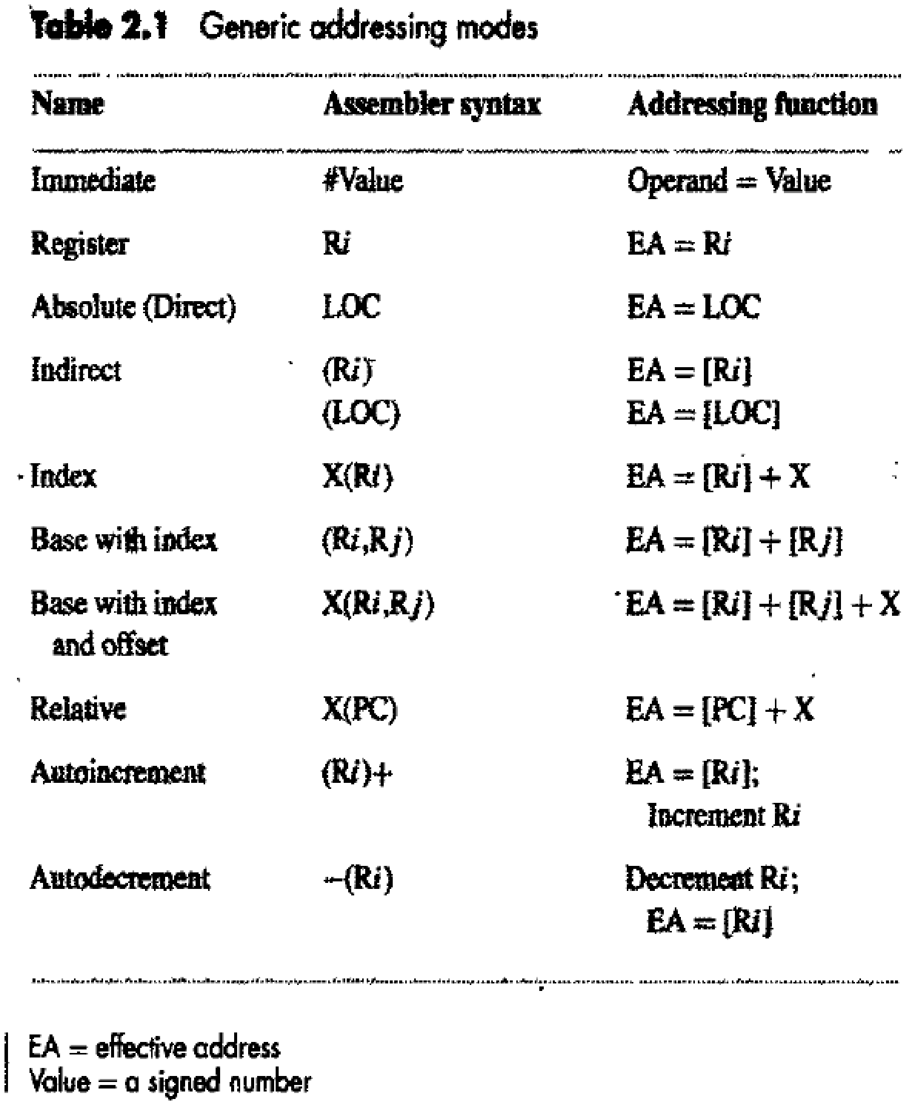


Table 2.1 Generic addressing modes

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (Direct)	LOC	EA = LOC

Indirect	(R*i*)
	(LOC)
		EA = [R*i*]
		EA = [LOC]

Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Base with index and offset	X(R*i*,R*j*)	EA = [R*i*] + [R*j*] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R*i*)+	EA = [R*i*]; Increment R*i*
Autodecrement	−(R*i*)	Decrement R*i*; EA = [R*i*]

EA = effective address
Value = a signed number

Addressing Modes..

- Immediate mode
  - The operand is given explicitly in the instruction.
    - Move #200,R0
    - Add #10,R1
- Register mode
  - The operand is the contents of a processor register; the name (address) of the register is given in the instruction.
    - Move R1,R2
    - Add R0,R1

Addressing Modes..

- Absolute mode (Direct mode)
  - The operand is in a memory location; the address of this location is given explicitly in the instruction.
    - Move LOC,R1
    - Add A,R0
- Indirect mode
  - The effective address of the operand is the contents of a register or memory location whose address appears in the instruction.
    - Move (R0),R1
    - Add (LOC),R2

Addressing Modes..

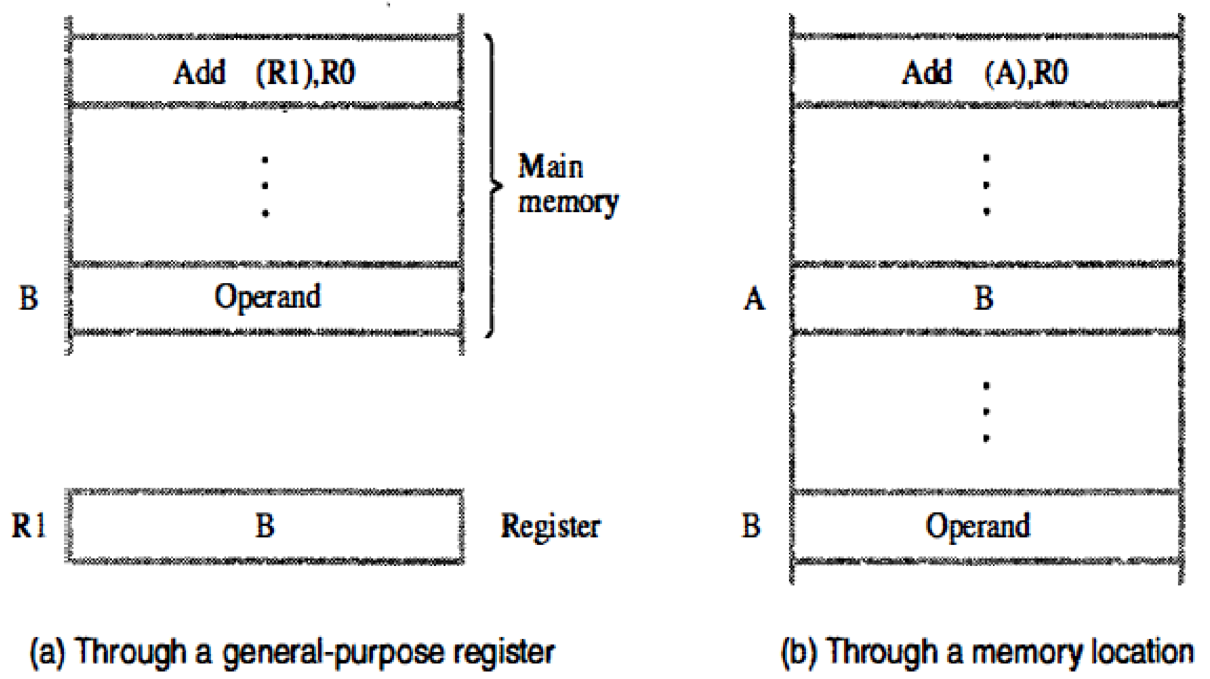
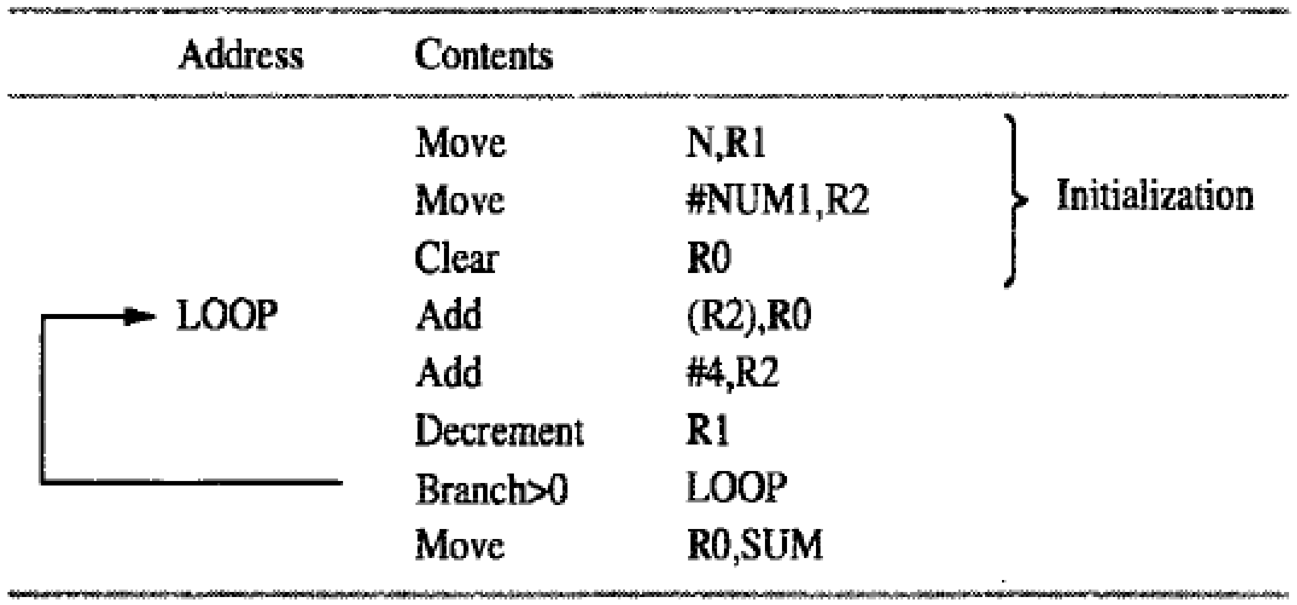

Add (R1),R0
Main memory
B
Operand
R1
B
Register
(a) Through a general-purpose register
Add (A),R0
A
B
B
Operand
(b) Through a memory location

Figure 2.11 Indirect addressing.

Addressing Modes..



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	

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

Addressing Modes..

- Index mode
  - The effective address of the operand is generated by adding a constant value to the contents of a register.
  - We indicate the Index mode symbolically as X(Ri)
  - X(Ri): EA = [Ri]+X

Addressing Modes..

- The value X defines an offset (also called a displacement) from this address to the location where the operand is found.
  - Move 20(R0),R1
  - Add 1000(R1),R2
- 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..

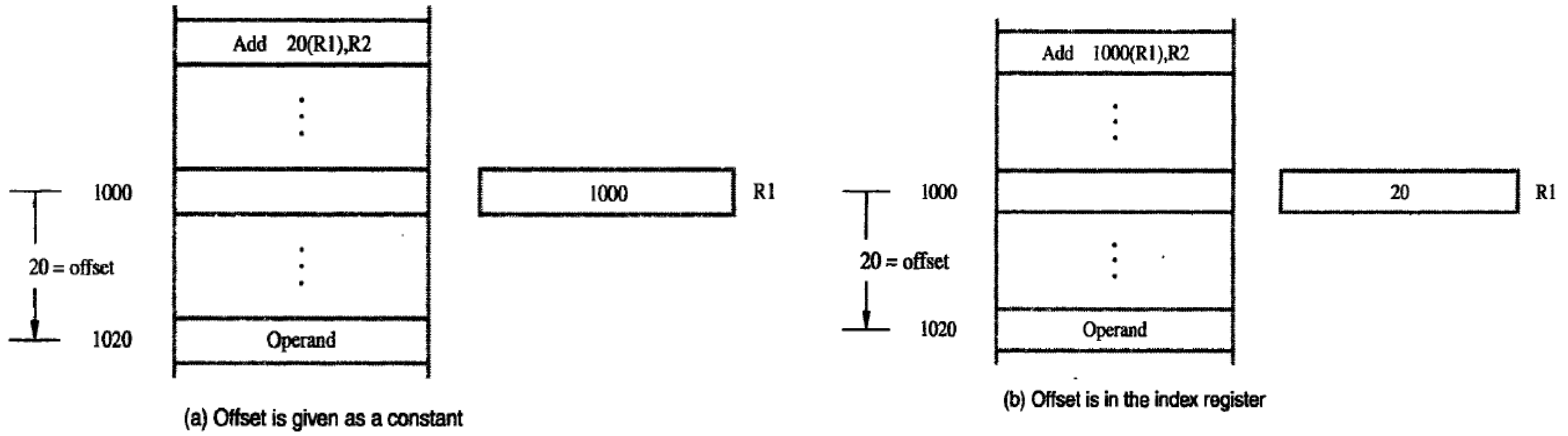


Figure 2.13 Indexed addressing.

Addressing Modes..

- Consider a simple example involving a list of test scores for students taking a given course.
- A four-word memory block comprises a record that stores the relevant information for each student.
- Each record consists of the student's identification number (ID), followed by the scores the student earned on three tests.

Addressing Modes..

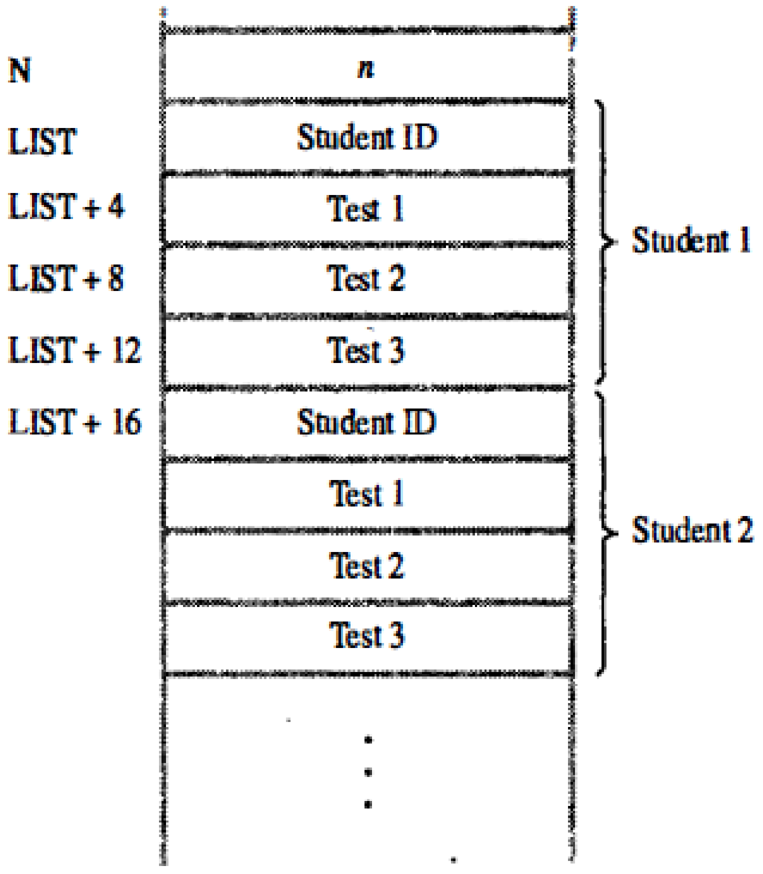
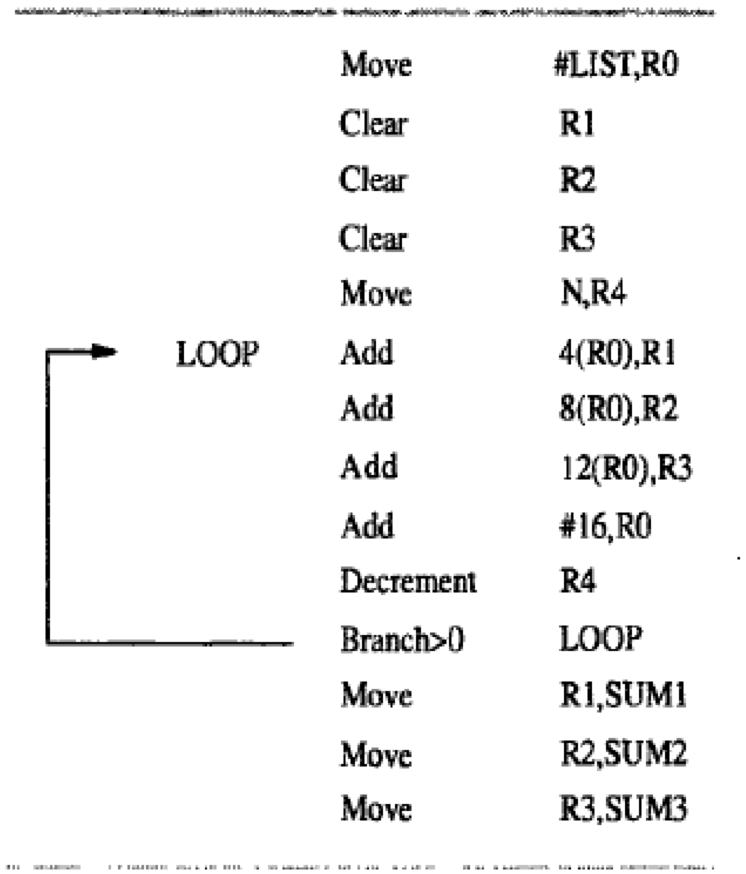


Figure 2.14 A list of students' marks.

Addressing Modes..

- Suppose that we wish to compute the sum of all scores obtained on each of the tests and store these three sums in memory locations SUMI, SUM2, and SUM3.

Addressing Modes..



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

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

Addressing Modes..

- Base with Index mode
  - The effective address is the sum of the contents of registers specified in the instruction.
  - (Ri,Rj): EA = [Ri]+[Rj]
    - Move (R0,R1),R2
    - Add (R1,R2),R0

Addressing Modes..

- Base with Index and Offset mode
  - The effective address is the sum of the constant X and the contents of registers specified in the instruction.
  - X(Ri,Rj): EA = [Ri]+[Rj]+X
    - Move 20(R0,R1),R2
    - Add 1000(R1,R2),R0

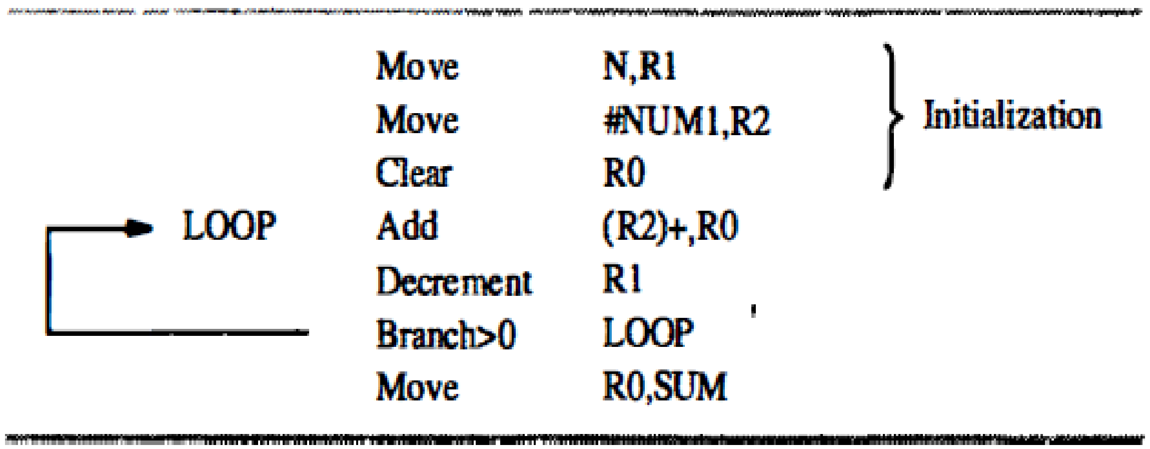
Addressing Modes..

- Relative mode
  - The effective address is determined by the Index mode using the program counter in place of the general-purpose register Ri.
    - Branch>0 LOOP
  - X(PC) : EA = [PC] + X
  - – note that X is a signed number
  - 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..

- 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.
    - Add (R2)+,R0
  - The increment is 1 for byte-sized operands, 2 for 16-bit operands, and 4 for 32-bit operands.

Addressing Modes..



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

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

Go to Figure 2.25

Addressing Modes..

- Autodecrement mode
  - The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand.
    - Add -(R2),R0
  - The decrement is 1 for byte-sized operands, 2 for 16-bit operands, and 4 for 32-bit operands.

Assembly Language

Assembly Language

- Machine instructions are represented by patterns of 0s and 1s.
  - Awkward to deal with when discussing or preparing programs.
  - We use symbolic names
- Assembly language – A complete set of symbolic names and rules for their use
- Syntax - The set of rules for using the mnemonics in the specification of complete instructions and programs
- Mnemonics

Assembly Language..

- Assembler – A program which translates programs written in an assembly language into a sequence of machine instructions
- Source program - The user program in its original alphanumeric text format
- Object program - The assembled machine language program

Assembly Language..

- MOVE R0,SUM
- The mnemonic MOVE represents the binary pattern, or OP code (Operation Code)
- R0 is Source operand and SUM is destination operand

Assembly Language..

- The assembly language must indicate which mode is being used.
- For example
  - A numerical value or a name used by itself, such as SUM in the preceding instruction, may be used to denote the Absolute mode.
  - The sharp sign usually denotes an immediate operand.
    - ADD #5,R3

Assembly Language..

- In some assembly languages, the intended addressing mode is indicated in the OP-code mnemonic.
- In this case, a given instruction has different OP-code mnemonics for different addressing modes.
  - ADDI 5,R3
- Indirect addressing is usually specified by putting parentheses around the name or symbol denoting the pointer to the operand.
  - MOVE #5,(R2)
  - MOVEI 5,(R2)

Assembler Directives

- These are commands used by the assembler while it translates a source program into an object program.
  - SUM EQU 200

Assembler Directives..

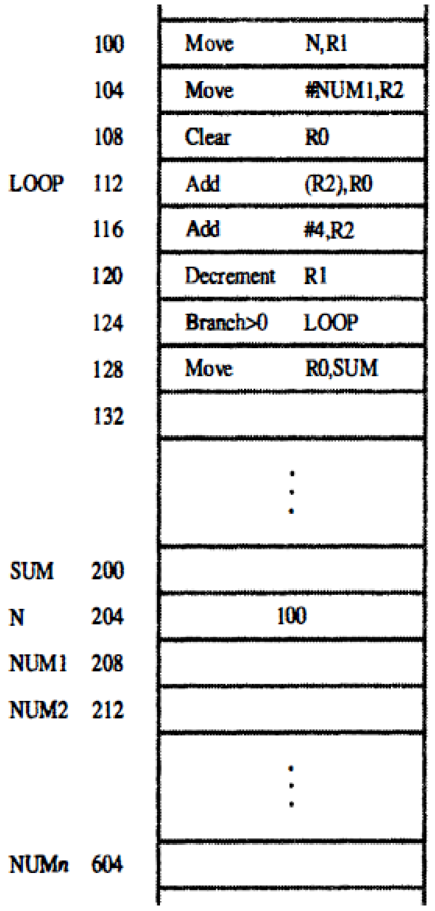
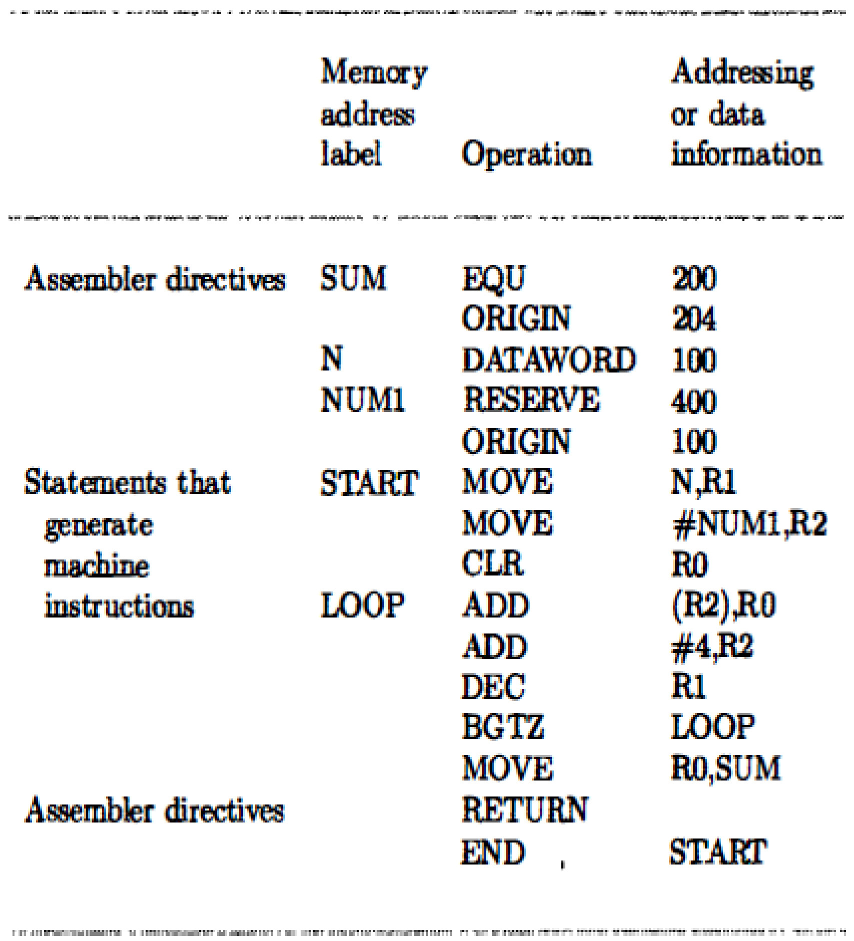


Figure 2.17 Memory arrangement for the program in Figure 2.12.

Assembler Directives..

- To produce an object program, an assembler 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

Assembler Directives..



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

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

Assembler Directives..

- EQU (*equate*) - informs the assembler that the variable name should be replaced by the value specified.
    - In this case, SUM should be replaced by the value 200 wherever it appears in the program
- ORIGIN - tells the assembler program where in the memory to place the data block that follows.
    - In this case, the location specified has the address 204

Assembler Directives..

- DATAWORD - informs the assembler that the data value specified is to be placed in the memory word
  - In this case, the data value 100 is to be placed in the memory word at address 204.

Assembler Directives..

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

Assembler Directives..

- RESERVE - declares that a memory block of specified bytes is to be reserved for data
  - In this case, a memory block of 400 bytes is to be reserved for data, and that the name NUM1 is to be associated with address 208.
- END - tells the assembler that this is the end of the source program text.

Assembler Directives..

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

  Label Operation Operand(s) Comment

- The Label is an optional name associated with the memory address
  - Address of machine instructions
  - Addresses of data items
- The Operation field contains the OP-code mnemonic of the desired instruction or assembler directive.
- The Operand field contains addressing information for accessing one or more operands, depending on the type of instruction.
- The Comment field is ignored by the assembler program.
  - Used for documentation purposes
  - Makes the program easier to understand.

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
- Assembler 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.

Assembly and Execution of Programs..

- A key part of the assembly process is determining the values that replace the names.
  - Easy in some cases
  - Branch offset
- As the assembler scans through a source program, it keeps track of all names and the numerical values that correspond to them in a *symbol table*.
  - When a name appears a second time, it is replaced with its value from the table.

Assembly and Execution of Programs..

- Two-pass assembler
  - During the first pass, it creates a complete symbol table. At the end of this pass, all names will have been assigned numerical values.
  - The assembler then goes through the source program a second time and substitutes values for all names from the symbol table.

Assembly and Execution of Programs..

- The assembler stores the object program on a magnetic disk.
- The object program must be loaded into the memory of the computer before it is executed.
- A utility program called a *loader* is used to load the object program into memory.

Assembly and Execution of Programs..

- The assembler can detect and report syntax errors.
- *Debugger* program helps the user find other programming errors, the system software
  - 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

- It is often convenient to use the familiar decimal notation.
  - ADD #93,R1
  - These values are stored in the computer as binary numbers.
- In some situations, it is more convenient to specify the binary patterns directly.
  - ADD #%01011101,R1
- Binary numbers can be written more compactly as *hexadecimal*, or *hex*, numbers
  - ADD #$5D,R1

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.

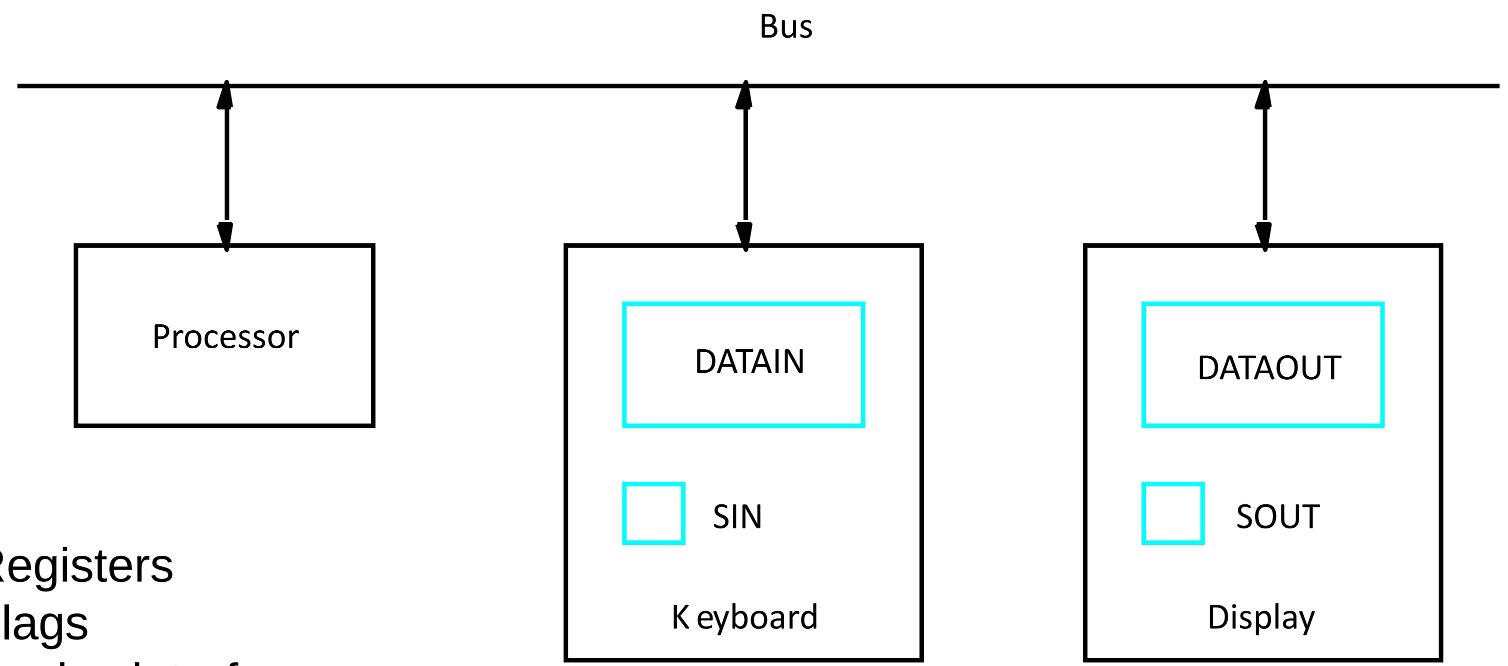
Program-Controlled I/O Example

- Consider a task that reads in character input from a keyboard and produces character output on a display screen.
  - Rate of data transfer (keyboard, display, processor)
    - Limited by the typing speed of the user
  - The rate of output transfers from the computer to the display is much higher.
    - Determined by the rate at which characters can be transmitted over the link between the computer and the display device
  - Difference in speed between processor and I/O device creates the need for mechanisms to synchronize the transfer of data.

Program-Controlled I/O Example..

- 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.
  - The processor waits for a signal indicating that a character key has been struck and that its code is available in some buffer register associated with the keyboard.
  - Then the processor proceeds to read that code.

Program-Controlled I/O Example..



- Registers
- Flags
- Device interface

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

Program-Controlled I/O Example..

- The keyboard and the display are separate devices as shown in Figure 2.19.
- The action of striking a key on the keyboard does not automatically cause the corresponding character to be displayed on the screen.
- One block of instructions in the I/O program transfers the character into the processor, and another associated block of instructions causes the character to be displayed.

Moving a character code from the keyboard to the processor

- Striking a key stores the corresponding character code in an 8-bit buffer register DATAIN.
- To inform the processor that a valid character is in DATAIN, a status control flag, SIN, is set to 1.
- A program monitors SIN, and when SIN is set to 1, the processor reads the contents of DATAIN.
- When the character is transferred to the processor, SIN is automatically cleared to 0.
- If a second character is entered at the keyboard, SIN is again set to 1 and the process repeats.

Transferring a character from the processor to the display

- A buffer register, DATAOUT, and a status control flag, SOUT, are used for this transfer.
- When SOUT equals 1, the display is ready to receive a character.
- Under program control, the processor monitors SOUT, and when SOUT is set to 1, the processor transfers a character code to DATAOUT.
- The transfer of a character to DATAOUT clears SOUT to 0.
- When the display device is ready to receive a second character, SOUT is again set to 1.

Program-Controlled I/O Example..

- The buffer registers DATAIN and DATAOUT and the status flags SIN and SOUT are part of circuitry commonly known as a *device interface*.

Program-Controlled I/O Example..

- Machine instructions that can check the state of the status flags and transfer data between the processor and the I/O device. :

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

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

Program-Controlled I/O Example..

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

```
MoveByte DATAIN,R1
MoveByte R1,DATAOUT
```

Program-Controlled I/O Example..

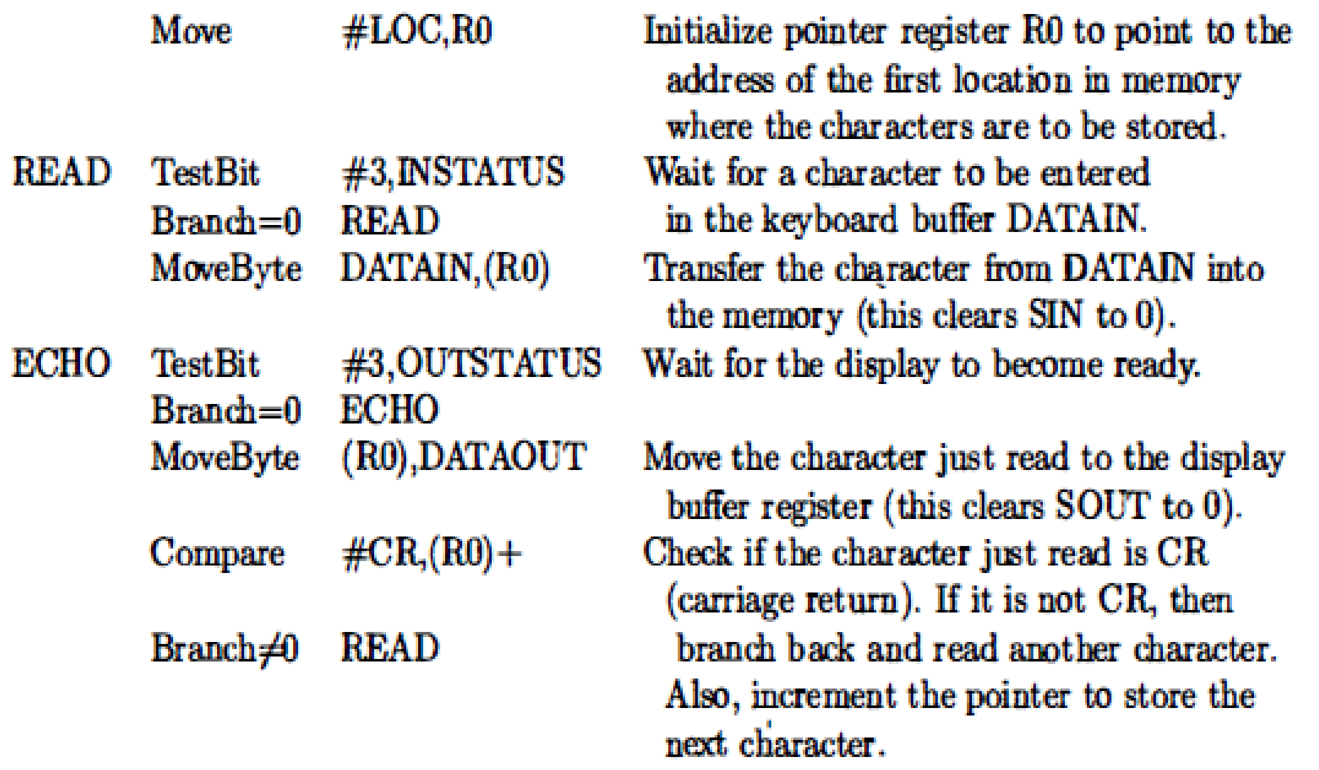
- Let us assume that bit b_3 in registers INSTATUS and OUTSTATUS corresponds to SIN and SOUT, respectively.
- The read operation may be implemented by the machine instruction sequence

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

- The write operation may be implemented as

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

Program-Controlled I/O Example..



```
         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.
```

Figure 2.20 A program that reads a line of characters and displays it.

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 and Queues

Stacks

- A *stack* is a list of data elements, usually words or bytes, 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* - the last data item placed on the stack is the first one removed when retrieval begins.
- 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.

Stacks..

- Assume that the first element is placed in location BOTTOM, and when new elements are pushed onto the stack, they are placed in successively lower address locations.
- Figure 2.21 shows a stack of word data items in the memory of a computer.
- It contains numerical values, with 43 at the bottom and -28 at the top.
- *Stack pointer (SP)* - A register used to keep track of the address of the element of the stack that is at the top at any given time.

Stacks..

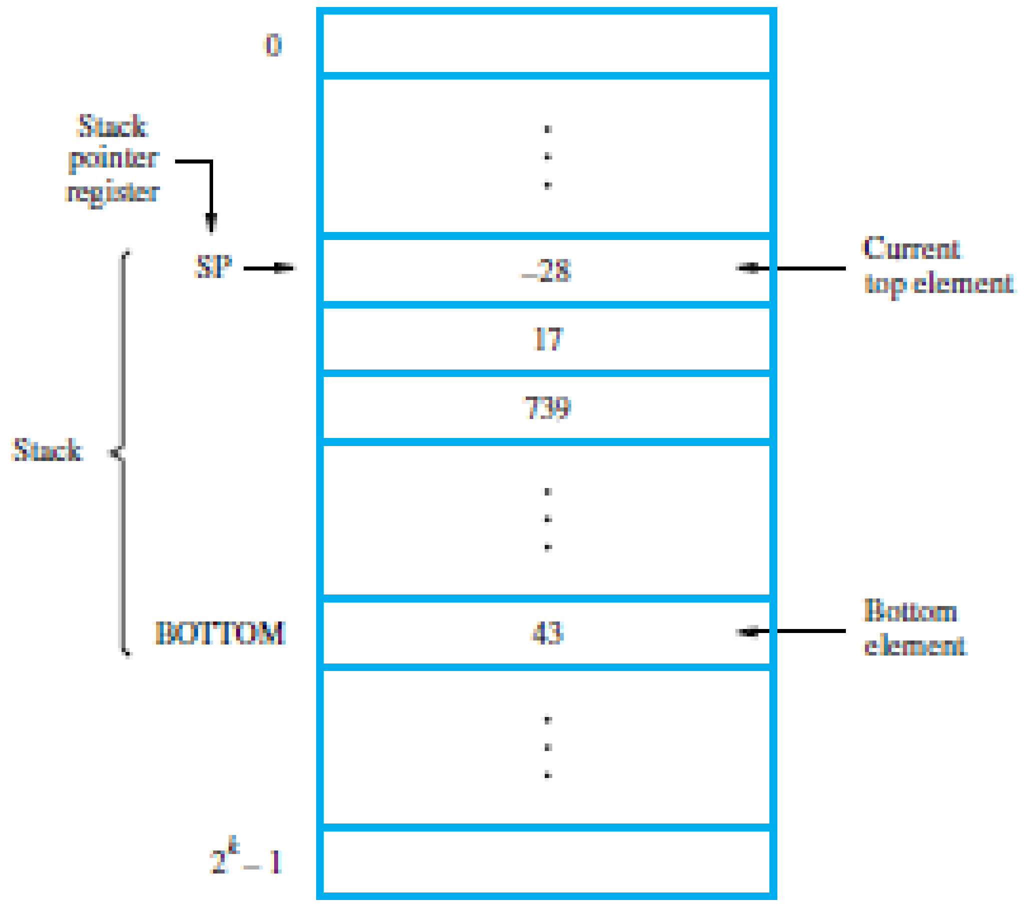


Figure 2.21 A stack of words in the memory.

Stacks..

- Assuming 32-bit word length, the PUSH operation will decrement the stack pointer by 4 and copies the source into the stack.
- Can be implemented as

```
Subtract #4,SP
Move NEWITEM,(SP)
```

- Or using autodecrement mode as

```
Move NEWITEM,-(SP)
```

Stacks..

- Assuming 32-bit word length, the POP operation will copy the content from the stack to the destination and increment the stack pointer by 4.
- Can be implemented as

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

- Or using autoincrement mode as

```
Move (SP)+,ITEM
```

Stacks..

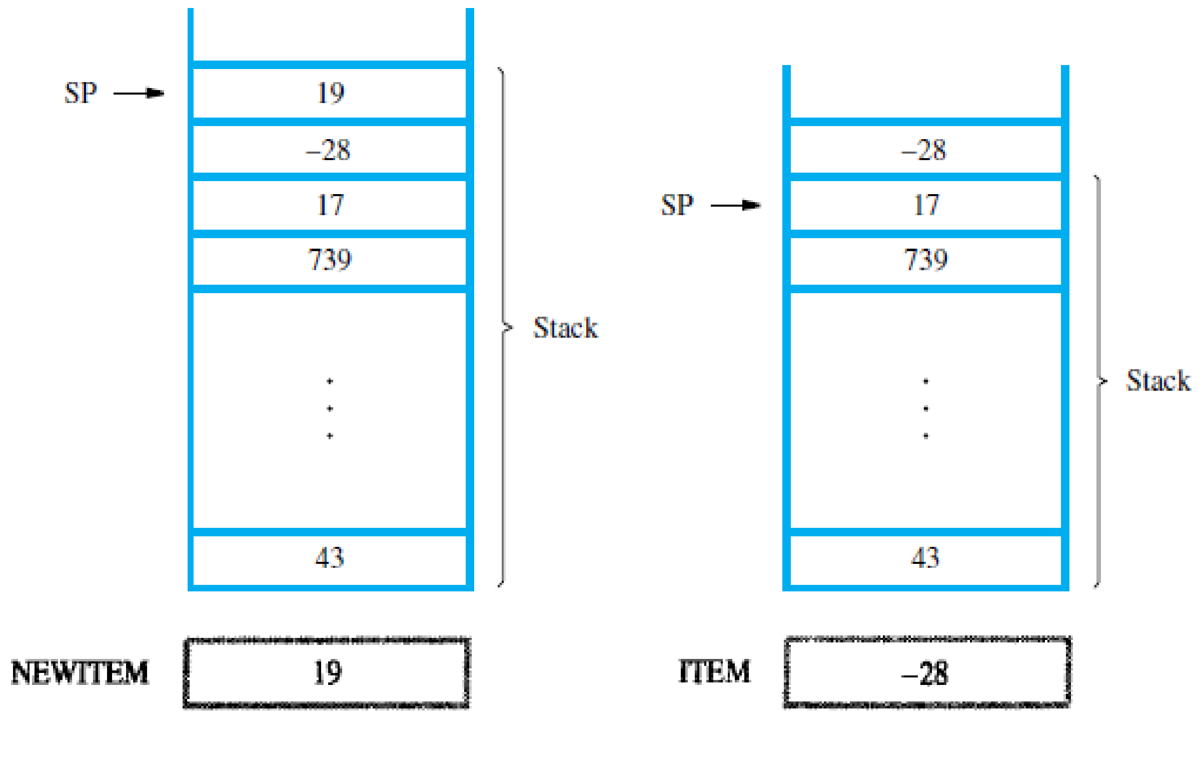
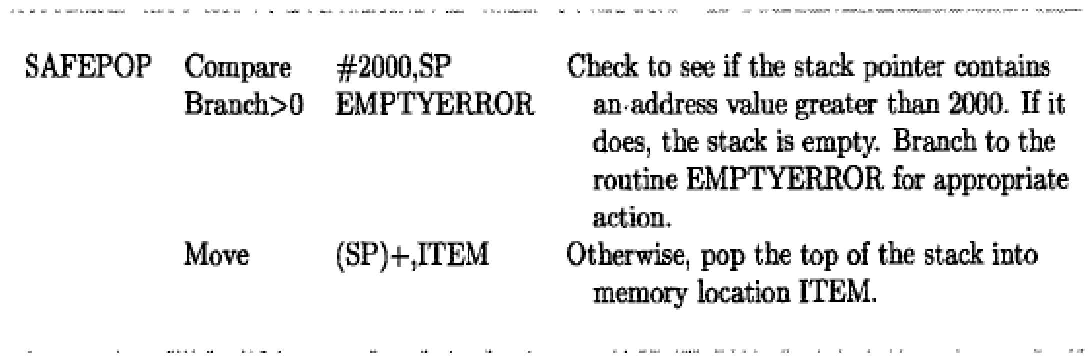


Figure 2.22 Effect of stack operations on the stack in Figure 2.21.

Stacks..

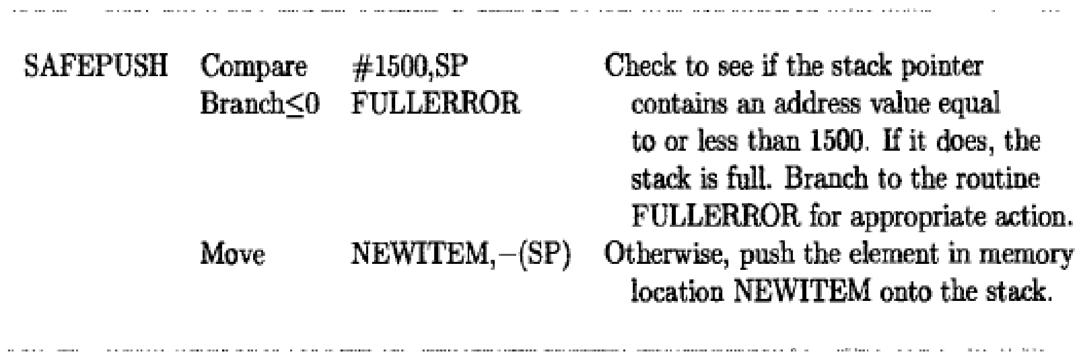
- Stack is usually allocated a fixed amount of space in the memory.
- We must avoid pushing an item on a full stack and popping an item from an empty stack.
  - Could result from a programming error.
- Suppose that a stack runs from location 2000 (BOTTOM) down to 1500.
- The stack pointer is loaded initially with address value 2004.

Stacks..



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

(a) Routine for a safe pop operation



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

(b) Routine for a safe push operation

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

Stacks..

- The Compare instruction

  Compare src,dst

  performs the operation

  [dst] - [src]

  and sets the condition code flags according to the result.
- It does not change the value of either operand.

Queues

- A queue is a list of data elements that works on first-in-first-out (FIFO) basis.
- Data are stored in and retrieved from a queue on a first-in-first-out (FIFO) basis.
- 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 Stacks and Queues

Stacks	Queues
One end of the stack is fixed (the bottom), while the other end rises and falls as data are pushed and popped.	Both ends of a queue move to higher addresses as data are added at the back and removed from the front.
A single pointer is needed to point to the top of the stack at any given time.	Two pointers are needed to keep track of the two ends of the queue.
A stack is limited by the top and bottom of the stack in the memory.	A queue would continuously move through the memory of a computer in the direction of higher addresses.

Queues..

- One way to limit the queue to a fixed region in memory is to use a *circular buffer*.
- Let us assume that memory addresses from BEGINNING to END are assigned to the queue.
  - The first entry in the queue is entered into location BEGINNING, and successive entries are appended to the queue by entering them at successively higher addresses.
  - By the time the back of the queue reaches END, space will have been created at the beginning if some items have been removed from the queue.
  - Hence, the back pointer is reset to the value BEGINNING and the process continues.

Subroutines

- In a given program, it is often necessary to perform a particular subtask many times on different data values.
- Such a subtask is usually called a *subroutine*.
- For example, a subroutine may evaluate the sine function or sort a list of values into increasing or decreasing order.

Subroutines..

- To save space, only one copy of the instructions of the subroutine 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.

Subroutines..

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

Subroutines..

- While a subroutine is called, 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 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.

Subroutines..

- Subroutine linkage method - The way in which a computer makes it possible to call and return from subroutines.
- Simplest method - 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 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

Subroutines..

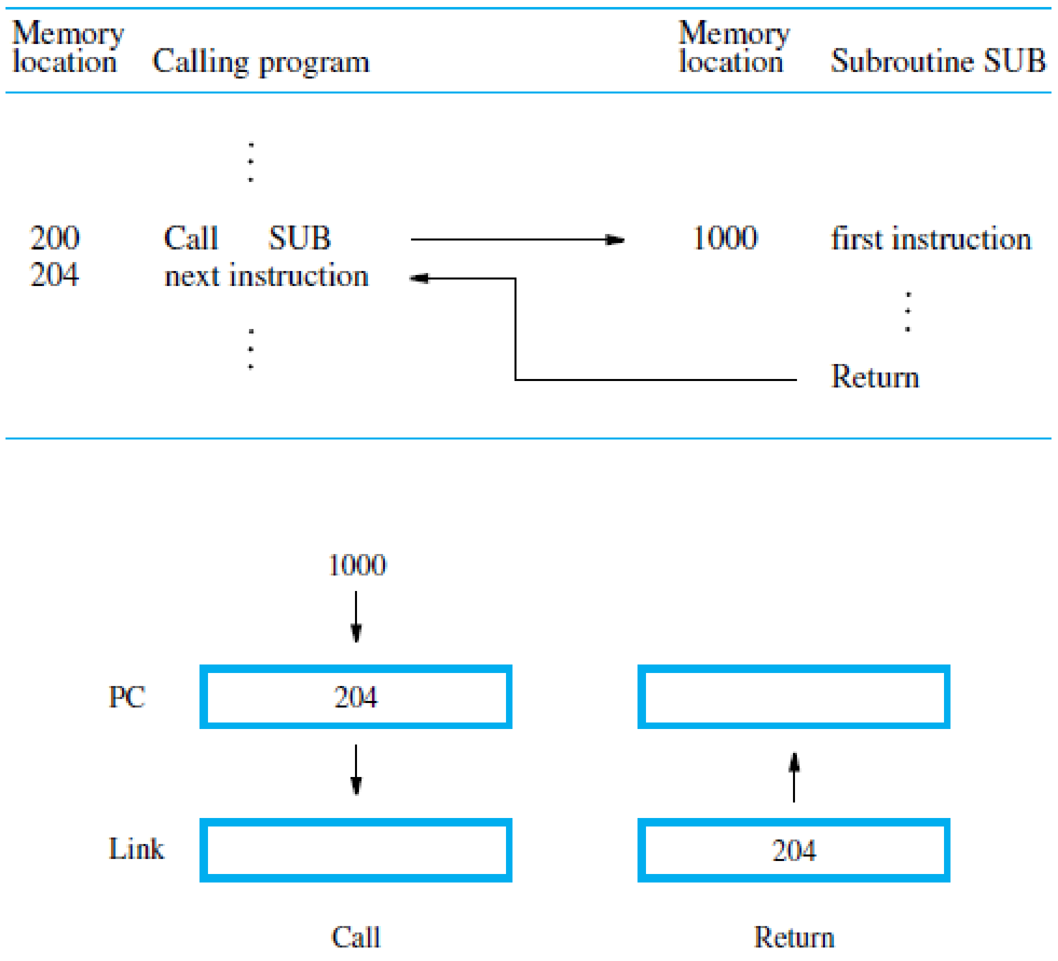


Figure 2.24 Subroutine linkage using a link register.

Subroutine Nesting and the Processor Stack

- Subroutine nesting – subroutine calling another subroutine
- In this case, the return address of the second call is also stored in the link register, destroying its previous contents.
  - The contents of the link register has to be saved in some other location before calling another subroutine.
  - Otherwise, the return address of the first subroutine will be lost.

Subroutine Nesting and the Processor Stack..

- Subroutine nesting can be carried out to any depth.
- Eventually, the last subroutine called, completes its computations and returns to the subroutine that called it.
- The return address needed for this first return is the last one generated in the nested call sequence.
- That is, return addresses are generated and used in a last-in-first-out order.

Subroutine Nesting and the Processor Stack..

- The return addresses associated with subroutine calls should be pushed onto a stack.
  - Many processors do this automatically as one of the operations performed by the Call instruction.
- A particular register is designated as the stack pointer, SP, to be used in this operation.
  - The stack pointer points to a stack called the *processor stack*.

Subroutine Nesting and the Processor Stack..

- The Call instruction pushes the contents of the PC onto the processor stack and loads the subroutine address into the PC.
- The Return instruction pops the return address from the processor stack into the PC.

Parameter Passing

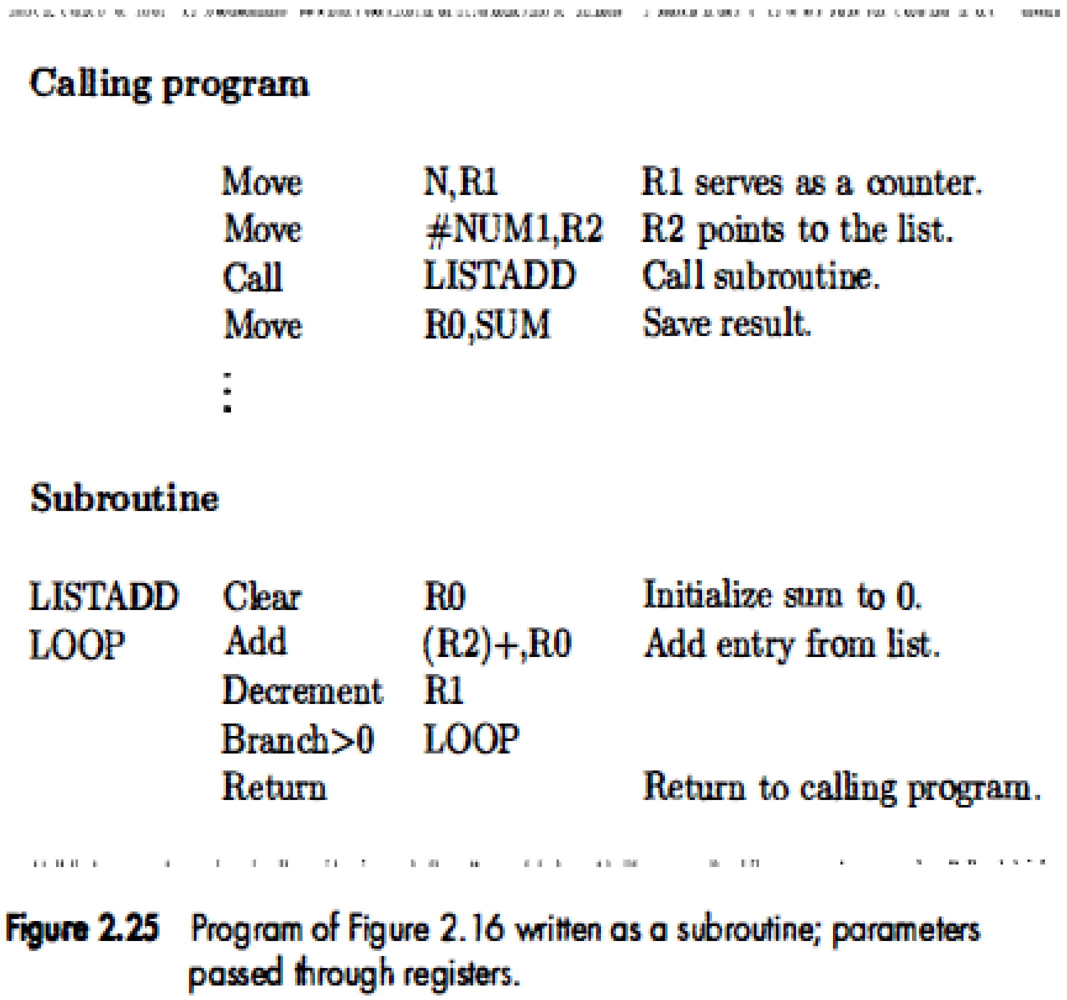
- *Parameter passing* - The exchange of information between a calling program and a subroutine.
- When calling a subroutine, a program must provide to the subroutine the parameters, that is, the operands or their addresses, to be used in the computation.
- Later, the subroutine returns other parameters, in this case, the results of the computation.

Parameter Passing..

- The parameters may be placed in registers or in memory locations, where they can be accessed by the subroutine.
- Alternatively, the parameters may be placed on the processor stack used for saving the return address.

Parameter Passing..

Refer Figure 2.16



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.

Parameter Passing..

- Figure 2.25 shows how the program in Figure 2.16 for adding a list of numbers can be implemented as a subroutine, with the parameters passed through registers.
- The size of the list, n, contained in memory location N, and the address, NUM1, of the first number, are passed through registers R1 and R2.
- The sum computed by the subroutine is passed back to the calling program through register R0.

Parameter Passing..

- If many parameters are involved, there may not be enough general-purpose registers available for passing them to the subroutine.
- Using a stack, on the other hand, is highly flexible
  - A stack can handle a large number of parameters.

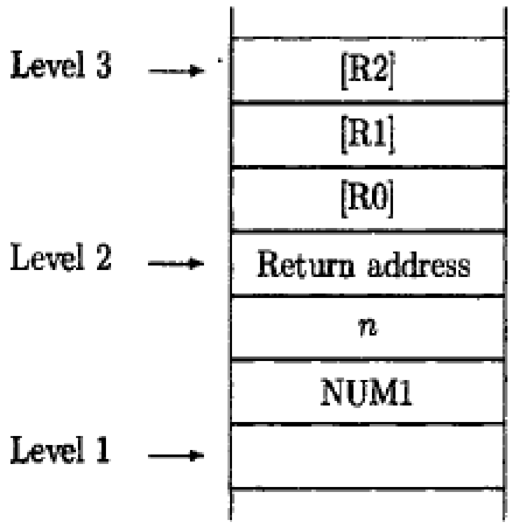
Parameter Passing..

Assume top of stack is at level 1 below.

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

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

(a) Calling program and subroutine



(b) Top of stack at various times

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

Parameter Passing by Value and by Reference

- Note the nature of the two parameters, NUM1 and n, passed to the subroutines in previous examples.
- Instead of passing the actual list entries, the calling program passes the address of the first number in the list.
  - This technique is called *passing by reference*.
- In case of the second parameter, the actual number of entries, n, is passed to the subroutine.
  - This technique is called *passing by value*.

Additional Instructions

Logic Instructions

- Logic operations such as AND, OR, and NOT, applied to individual bits, are the basic building blocks of digital circuits.
- Bitwise logic instructions
  - Not dst
  - And src,dst
  - Or src,dst

Logic Instructions..

- To find negative (2’s complement) of a number

```
Not R0
Add #1,R0
```

- Many computers have a single instruction for this

```
Negate R0
```

Logic Instructions..

- Suppose that four ASCII characters are contained in the 32-bit register R0.
- In some task, we wish to determine if the leftmost character is Z.
- If it is, a conditional branch to YES is to be made.
- The ASCII code for Z is 01011010, which is expressed in hexadecimal notation as 5A.

Logic Instructions..

- The three-instruction sequence implements the desired action.

```
And #$FF000000,R0
Compare #$5A000000,R0
Branch=0 YES
```

- The And instruction is often used in practical programming tasks where all bits of an operand except for some specified field are to be cleared to 0.

Shift and Rotate Instructions

- Shift Instructions
  - Logical Shift
  - Arithmetic Shift
- Rotate Instructions

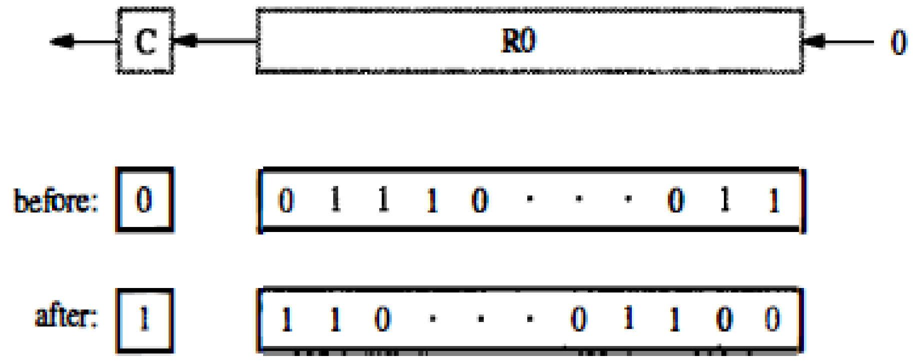
Logical Shifts

- Logical shift
  - Logical shift left (LShiftL)
  - Logical shift right (LShiftR)
- These instructions shift an operand over a number of bit positions specified in a count operand contained in the instruction.
- General form

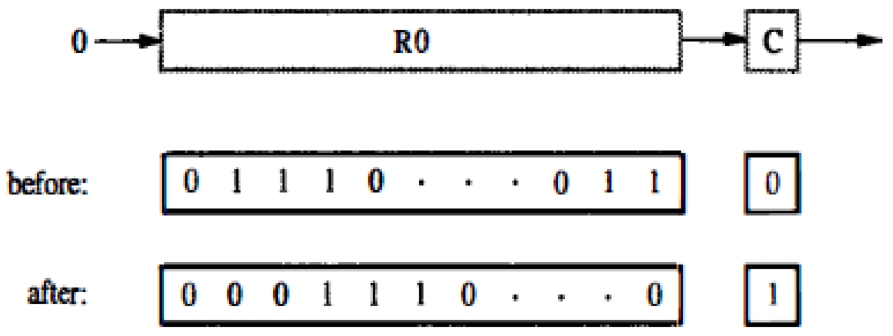
```
LShiftL count,dst
LShiftR count,dst
```

- The count operand may be an immediate operand, or it may be contained in a processor register.

Logical Shifts..



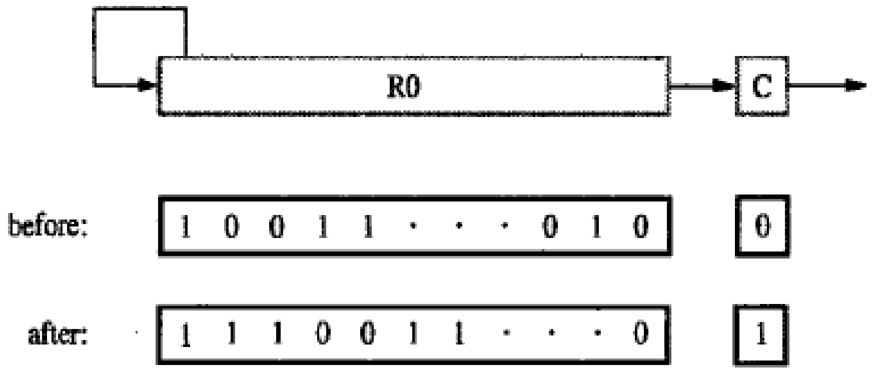
(a) Logical shift left LShiftL #2,R0



(b) Logical shift right LShiftR #2,R0

Arithmetic Shifts

- Retains the sign of the number



(c) Arithmetic shift right AShiftR #2,R0

Figure 2.30 Logical and arithmetic shift instructions.

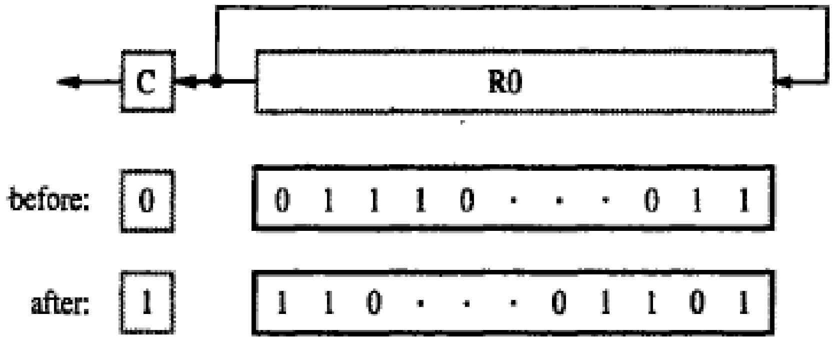
Arithmetic Shifts

- Note:
  - Shifting a number one bit position to the left is equivalent to multiplying it by 2; and shifting it to the right is equivalent to dividing it by 2.

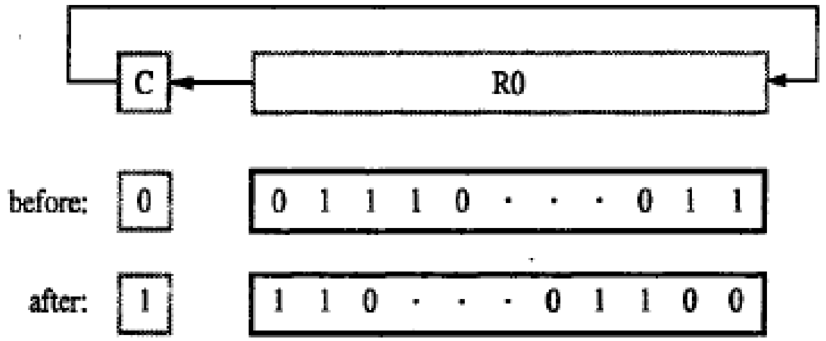
Rotate Operations

- In the shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is retained in the Carry flag C.
- To preserve all bits, a set of rotate instructions can be used.
- They move the bits that are shifted out of one end of the operand back into the other end.
- Rotate without carry
- Rotate with carry

Rotate Operations..



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



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

Rotate Operations..

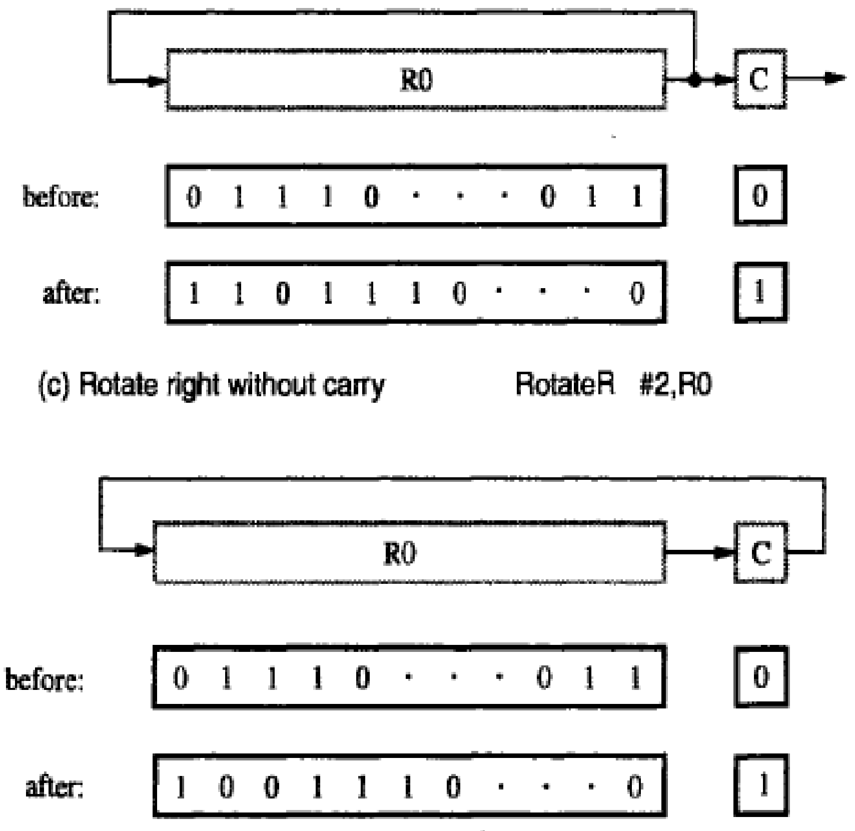
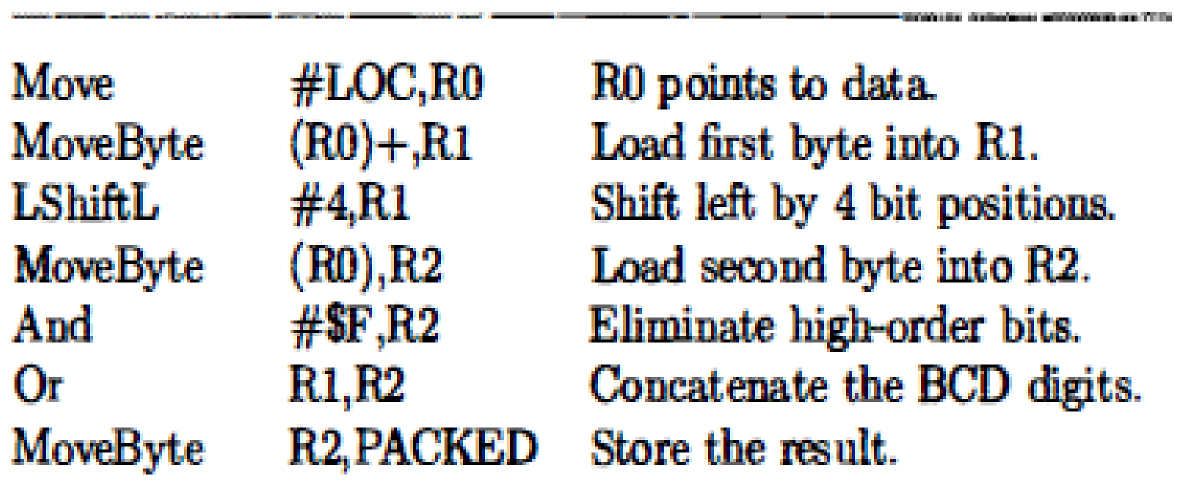


Figure 2.32 Rotate instructions.

Digit-Packing Example

- Suppose that two decimal digits represented in ASCII code are located in memory at byte locations LOC and LOC + 1.
- We wish to represent each of these digits in the 4-bit BCD code and store both of them in a single byte location PACKED.
  - The result is said to be in *packed-BCD* format.
- Hence, the required task is to extract the low-order four bits in LOC and LOC + 1 and concatenate them into the single byte at PACKED.

Digit-Packing Example



```
Move        #LOC,R0      R0 points to data.
MoveByte    (R0)+,R1     Load first byte into R1.
LShiftL     #4,R1        Shift left by 4 bit positions.
MoveByte    (R0),R2      Load second byte into R2.
And         #$F,R2       Eliminate high-order bits.
Or          R1,R2        Concatenate the BCD digits.
MoveByte    R2,PACKED    Store the result.
```

Figure 2.31 A routine that packs two BCD digits.

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)