COMPUTER ORGANIZATION & ARCHITECTURE – BEC306C

MODULE 2

Addressing Modes, Assembly Language, Basic Input and Output Operations, Stacks and queues, Subroutines, Additional Instructions.

ADDRESSING MODES

The different ways for specifying the locations of instruction operands are known as *addressing modes*. A summary is provided in Table 2.1.

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

Implementation of Variables and Constants: In assembly language, a variable is represented by allocating a register or a memory location to hold its value. This value can be changed as needed using appropriate instructions. There are two addressing modes to access variables.

Register mode - The operand is the contents of a processor register; the name of the register is given in the instruction.

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

Register Mode: The operand is the contents of a register. The name (or address) of the register is given in the instruction. Registers are used as temporary storage locations where the data in a register are accessed.

For example: *Move R1, R2*; Copy content of register R1 into register R2.

Absolute (Direct) Mode: The operand is in a memory-location. The address of memory-location is given explicitly in the instruction. The absolute mode can represent global variables in the program.

For example: *Move LOC, R2*; Copy content of memory-location LOC into register R2.

Immediate Mode: The operand is given explicitly in the instruction.

For example: *Move #200, R0*; Place the value 200 in register R0.

Clearly, the immediate mode is only used to specify the value of a source-operand.

Constant values are used frequently in high level language programs. For example: A=B + 6; contains the constant 6. This statement can be compiled as:

Move B, R1
Add #6, R1
Move R1, A

Constants are also used in assembly language to increment a counter, test for some bit pattern and so on.

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

INDIRECTION AND POINTERS

Indirect Mode: The effective address of the operand is the contents of a register or memory-location that is specified in the instruction. The register (or memory-location) that contains the address of an operand is called a Pointer. Indirection is denoted by placing the name of the register given in the instruction in parentheses as illustrated in Figure 2.1 and Table 2.1.

To execute the Add instruction in fig 2.1(a), the processor uses the value which is in register R1, as the EA of the operand. It requests a read operation from the memory to read the contents of location B. The value read is the desired operand, which the processor adds to the contents of register R0. Indirect addressing through a memory-location is also possible as shown in fig 2.1(b). In this case, the processor

first reads the contents of memory-location A, then requests a second read operation using the value B as an address to obtain the operand.

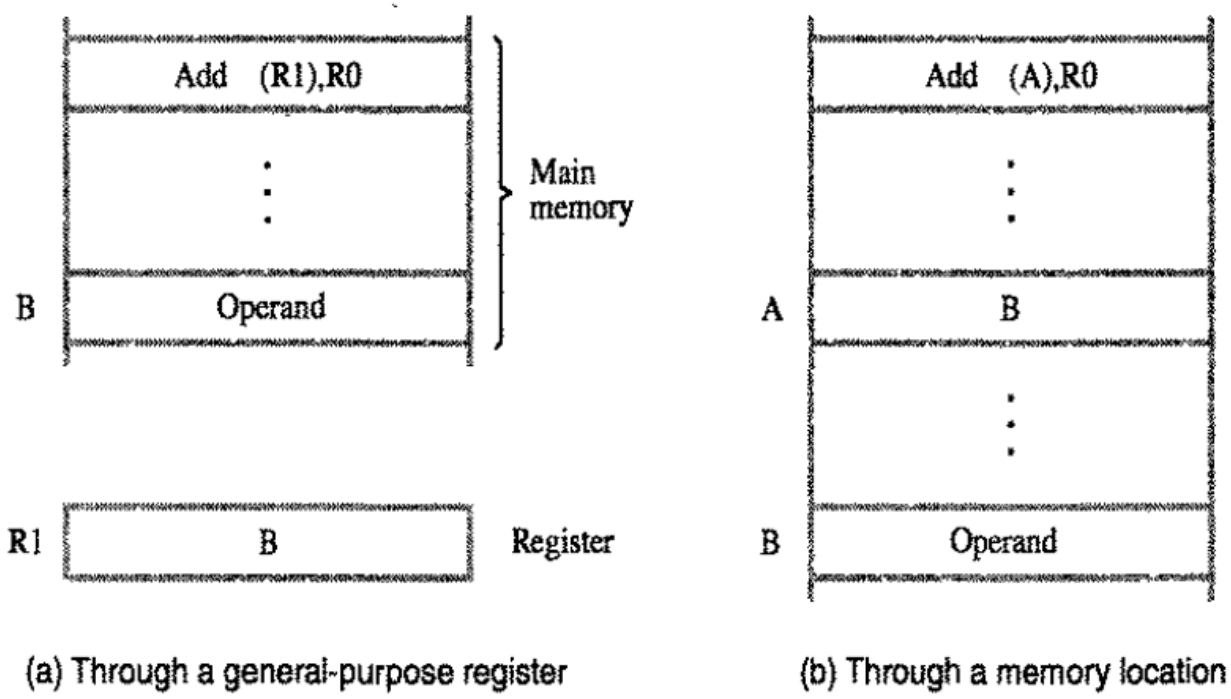


Figure 2.1 Indirect addressing.

Example: Program to add n numbers using indirect addressing mode

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	

INDEXING AND ARRAYS

Index mode: The effective address of the operand is generated by adding a constant value to the contents of a register. The register used in this mode is referred as the index register. The index mode symbolically indicated as X (R_i), where X denotes a constant signed integer value contained in the instruction and R*i* is the name of the register involved. The effective address of the operand is given by

EA = X + [R*i*]

Figure 2.13 illustrates two ways of using the Index mode. In Figure 2.13*a*, the index register, R1, contains the address of a memory location, and the value X defines an *offset* (also called a *displacement*) from this address to the location where the operand is found. In Figure 2.13*b*, the constant X corresponds to a memory address, and the contents of the index register define the offset to the operand.

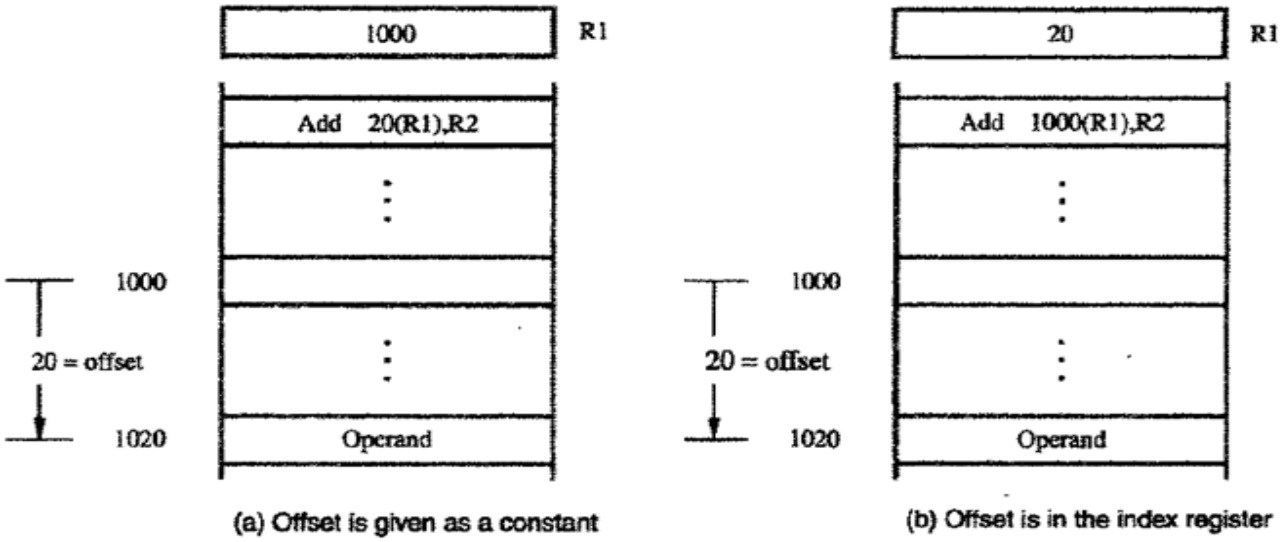


Fig. 2.13: Indexed addressing

BASIC INPUT/OUTPUT OPERATIONS

Program Controlled I/O: Consider the problem of moving a character code from the keyboard to the processor. Striking a key stores corresponding character code in an 8-bit buffer register DATAIN, as shown in Fig. 2.19. To inform the processor that a valid character is in DATAIN, a status control flag, SIN is set to 1. A program monitors SIN, & 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, SIN is again set to 1 and the process repeats.

A buffer register, DATAOUT, and a status control flag, SOUT, are used for transferring a character from the processor to the display. 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.

The buffer registers DATAIN and DATAOUT and the status flags SIN and SOUT are part of circuitry commonly known as a device interface. The circuitry for each device is connected to the processor via a bus, as indicated in Figure 2.19.

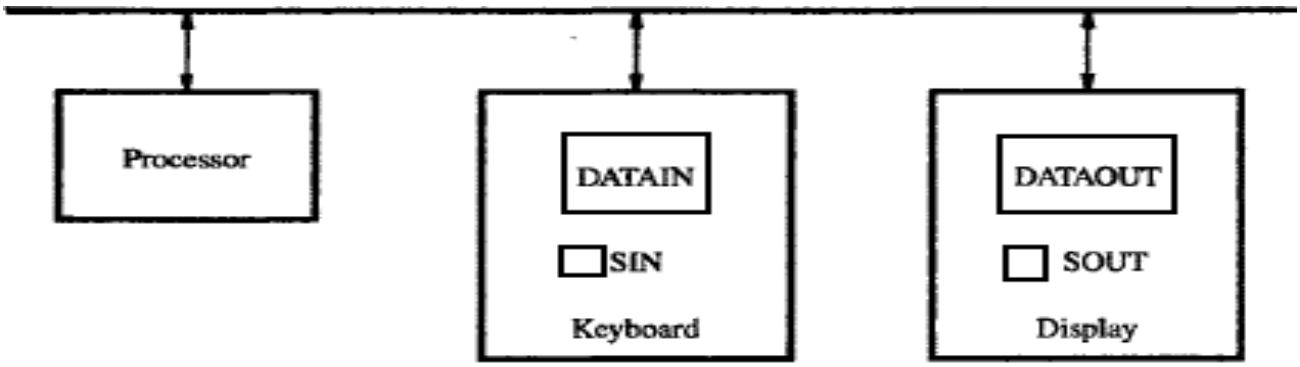


Figure 2.19 Bus connection for processor, keyboard, and display.

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

```
READWAIT        Branch to READWAIT if SIN = 0
                Input from DATAIN to R1
WRITEWAIT       Branch to WRITEWAIT if SOUT = 0
                Output from R1 to DATAOUT
```

Memory mapped I/O: Many computers use an arrangement called memory-mapped I/O in which some memory address values are used to refer to peripheral device buffer registers, such as DATAIN and DATAOUT. Thus, no special instructions are needed to access the contents of these registers; data can be transferred between these registers and the processor using instructions such as Move, Load, or Store.

For example, the contents of the keyboard character buffer DATAIN can be transferred to register R1 in the processor by instruction: MoveByte DATAIN, R1. Similarly, the contents of register R1 can be transferred to DATAOUT by the instruction: MoveByte R1, DATAOUT.

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
                MoveByteR1,        DATAOUT
```

The Testbit instruction tests the state of one bit in the destination location. If the bit tested is equal to 0, then the condition of the branch instruction is true, and a branch is made to the beginning of the wait loop. When the bit tested becomes equal to 1, the data are read from the input buffer or written into the output buffer.

```
        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.

The program shown in Figure 2.20 uses these two operations to read a line of characters typed at a keyboard and send them out to a display device. As the characters are read in, one by one, they are stored in a data area in the memory and then echoed back out to the display. The program finishes when the carriage return character, CR, is read, stored, and sent to the display.

STACKS AND QUEUES

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.

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. A processor register is used to keep track of the address of the element of the stack that is at the top at any given time. This register is called the *stack pointer* (SP).

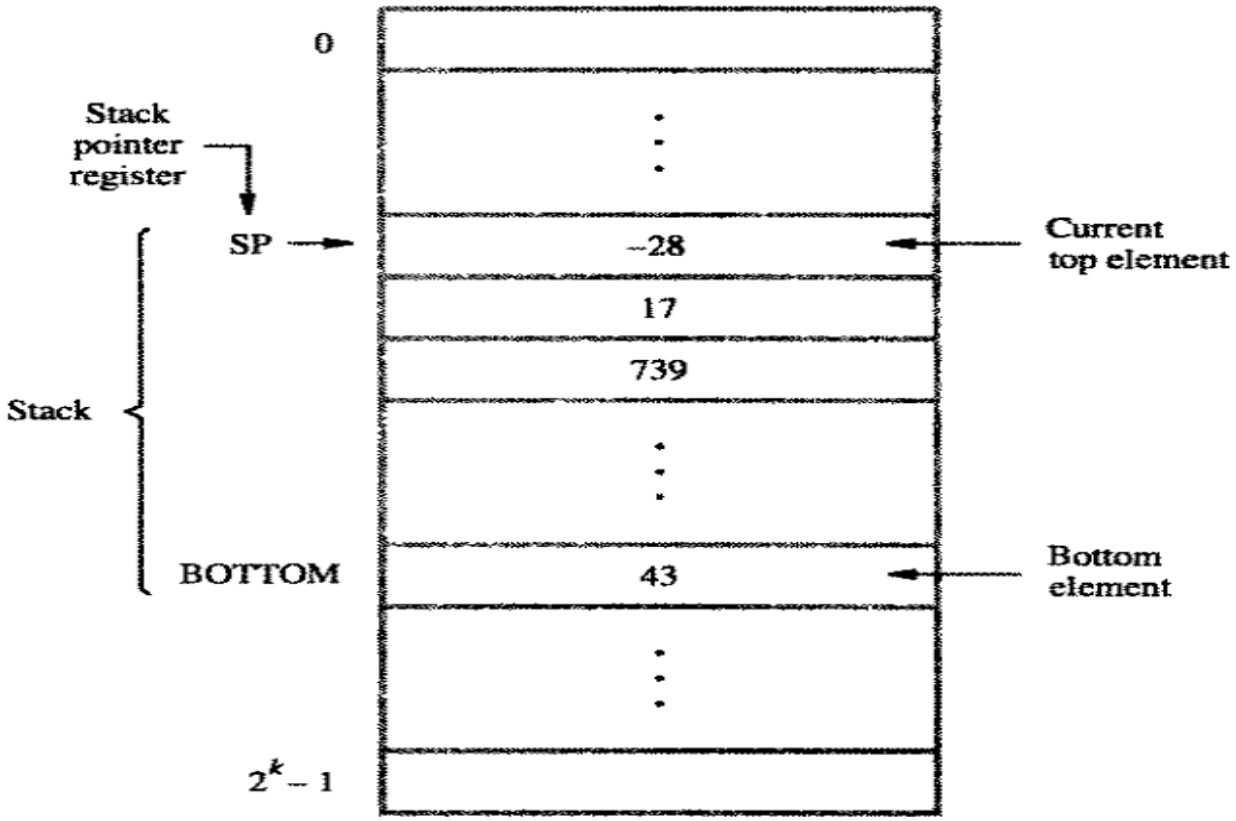


Figure 2.21 A stack of words in the memory.

Assuming a byte-addressable memory with a 32-bit word length, the push operation can be implemented as

Subtract #4, SP

Move NEWITEM, (SP)

These two instructions move the word from location NEWITEM onto the top of the stack, decrementing the stack pointer by 4 before the move.

The pop operation can be implemented as

Move (SP), ITEM

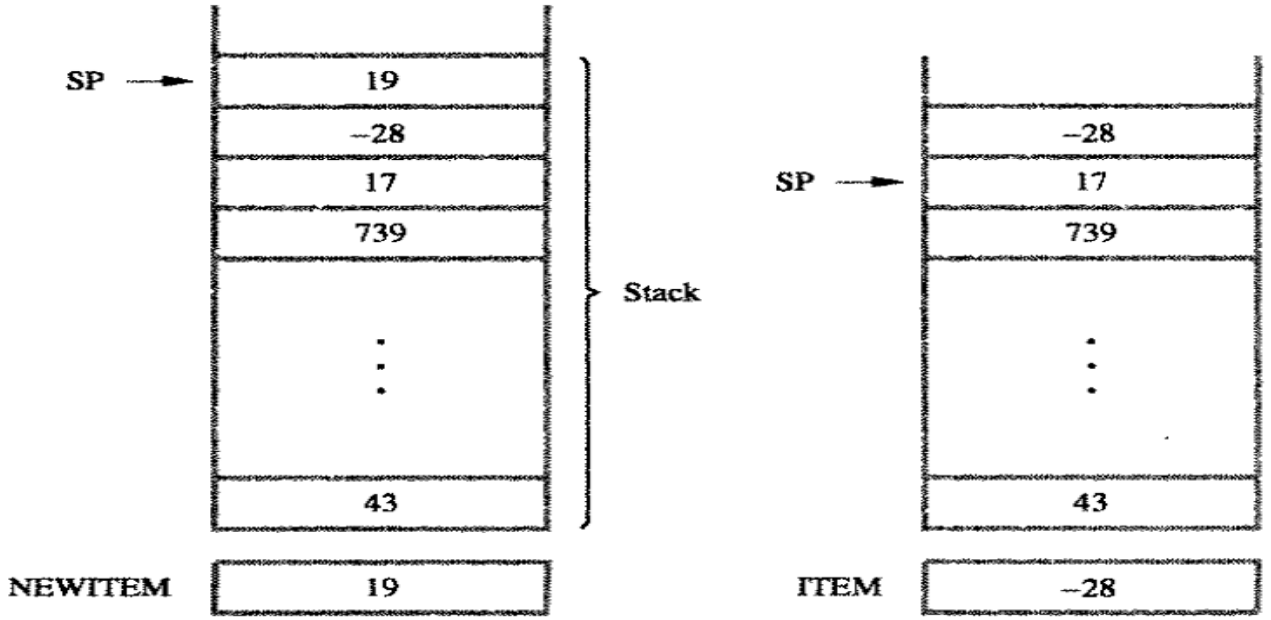
Add #4, SP

These two instructions move the top value from the stack into location ITEM and then increment the stack pointer by 4 so that it points to the new top element.

Fig. 2.22 shows the effect of each of these operations on the stack in Fig. 2.21. Using autoincrement and autodecrement addressing mode the push and pop operation can be implemented as:

Move NEWITEM,-(SP)

Move (SP)+, ITEM



(a) After push from NEWITEM (b) After pop into ITEM

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

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.

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*. To save space, only one copy of the instructions that constitute the subroutine is placed in

the memory, and any program that requires the use of the subroutine simply branches to its starting location. When a program branches to a subroutine we say that it is *calling* the subroutine. The instruction that performs this branch operation is named a *Call* instruction.

After a subroutine has been executed, the calling program must resume execution, continuing immediately after the instruction that called the subroutine. The subroutine is said to return to the program that called it by executing a *Return* instruction. Since the subroutine may be called from different places in a calling program, 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.

The way in which a computer makes it possible to call and return from subroutines is referred to as its *subroutine linkage* method. The simplest subroutine linkage method is to save the return address in a specific location, such a register 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.

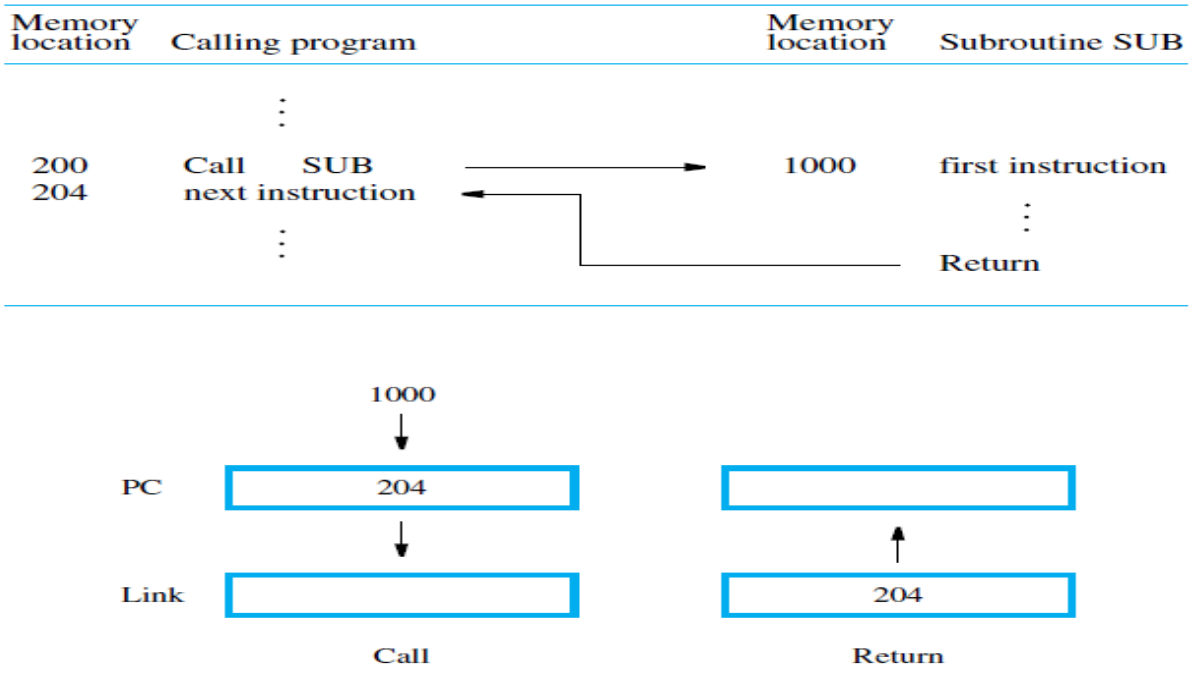


Figure 2.16 Subroutine linkage using a link register.

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 2.16 illustrates this procedure.

Subroutine Nesting and the Processor Stack

Subroutine Nesting means one subroutine calls another subroutine. In this case, the return-address of the second call is also stored in the link-register, destroying its previous contents. Hence, it is essential to save the contents of the link-register in some other location before calling another subroutine. Otherwise, the return-address of the first subroutine will be lost.

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. This suggests that the return addresses associated with subroutine calls should be pushed onto a stack. The register stack pointer (SP), is used in this operation. The SP points to a stack called the processor stack. The Call instruction pushes the contents of the PC onto the processor stack and loads the subroutine address into the PC. The Return instruction pops the return address from the processor stack into the PC.

Parameter Passing

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

Figure 2.25 shows how the program 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.

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.
```

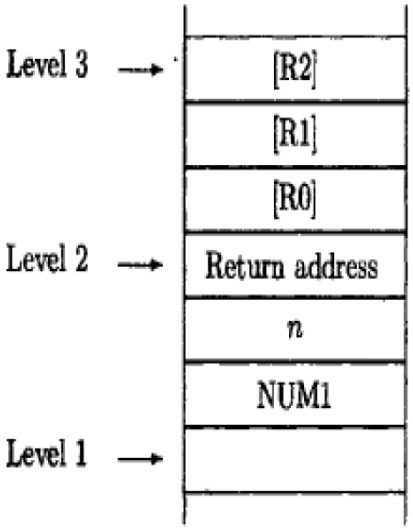
Figure 2.25 Program of Figure 2.16 written as a subroutine; parameters passed through registers.

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. Figure 2.26a shows the program of rewritten as a subroutine, LISTADD, which can be called by any other program to add a list of numbers.

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

Note the nature of the two parameters, NUM1 and *n*, passed to the subroutines in Figures 2.25 and 2.26. The purpose of the subroutines is to add a list of numbers. Instead of passing the actual list entries, the calling program passes the address of the first number in the list. This technique is called *passing by reference*.

The second parameter is *passed by value*, that is, the actual number of entries, *n*, is passed to the subroutine.