# 11

# Writing Procedures

*The last two chapters introduced the basics of the assembly language. Here we discuss how procedures are written in the assembly language. Procedure is an important programming construct that facilitates modular programming. In the IA-32 architecture, the stack plays an important role in procedure invocation and execution. We start this chapter by giving details on the stack, its uses, and how it is implemented. We also describe the assembly language instructions to manipulate the stack.*

*After this introduction to the stack, we look at the assembly language instructions for procedure invocation and return. Unlike high-level languages, there is not much support in the assembly language. For example, we cannot include the arguments in the procedure call. Thus parameter passing is more involved than in high-level languages. There are two parameter passing methods—one uses the registers and the other the stack. We discuss these two parameter passing methods in detail. The last section provides a summary of the chapter.*

## Introduction

A procedure is a logically self-contained unit of code designed to perform a particular task. These are sometimes referred to as *subprograms* and play an important role in modular program development. In high-level languages, there are two types of subprograms: *procedures* and *functions*. A function receives a list of arguments and performs a computation based on the arguments passed onto it and returns a single value. In this sense, these functions are very similar to the mathematical functions.

Procedures also receive a list of arguments just as the functions do. However, procedures, after performing their computation, may return zero or more results back to the calling procedure. In the C language, both these subprogram types are combined into a single function construct.

In the C function

```
int sum (int x, int y)
{
    return (x + y);
}
```

the parameters `x` and `y` are called formal parameters or simply parameters and the function body is defined based on these parameters. When this function is called (or invoked) by a statement like

```
total = sum(number1, number2);
```

the actual parameters or arguments—`number1` and `number2`—are used in the computation of the `sum` function.

There are two types of parameter passing mechanisms: *call-by-value* and *call-by-reference*. In the call-by-value mechanism, the called function (`sum` in our example) is provided only the current values of the arguments for its use. Thus, in this case, the values of these arguments are not changed in the called function; these values can only be used as in a mathematical function. In our example, the `sum` function is invoked by using the call-by-value mechanism, as we simply pass the values of `number1` and `number2` to the called `sum` function.

In the call-by-reference mechanism, the called function actually receives the addresses (i.e., pointers) of the parameters from the calling function. The function can change the contents of these parameters—and these changes will be seen by the calling function—by directly manipulating the argument storage space. For instance, the following `swap` function

```
void swap (int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

assumes that it receives the addresses of the two parameters from the calling function. Thus, we are using the call-by-reference mechanism for parameter passing. Such a function can be invoked by

```
swap (&data1, &data2);
```

Often both types of parameter passing mechanisms are used in the same function. As an example, consider finding the roots of the quadratic equation

$$ax^2 + bx + c = 0\,.$$

The two roots are defined as

$$\text{root1} = \frac{-b + \sqrt{b^2 - 4ac}}{2a}\,,$$

$$\text{root2} = \frac{-b - \sqrt{b^2 - 4ac}}{2a}\,.$$

The roots are real if $b^2 \geq 4ac$, and imaginary otherwise.

Suppose that we want to write a function that receives $a$, $b$, and $c$ and returns the values of the two roots (if real) and indicates whether the roots are real or imaginary (see Figure 11.1). The `roots` function receives parameters `a`, `b`, and `c` using the call-by-value mechanism, and `root1` and `root2` parameters are passed using the call-by-reference mechanism. A typical invocation of `roots` is

```
root_type = roots (a, b, c, &root1, &root2);
```

```
int roots (double a, double b, double c,
           double *root1, double *root2)
{
    int root_type = 1;
    if (4 * a * c <= b * b){  /* roots are real */
       *root1 = (-b + sqrt(b*b - 4*a*c))/(2*a);
       *root2 = (-b - sqrt(b*b - 4*a*c))/(2*a);
    }
    else    /* roots are imaginary */
       root_type = 0;
    return (root_type);
 }
```

**Figure 11.1** C function for the quadratic equation

In summary, procedures receive a list of arguments, which may be passed either by the call-by-value or by the call-by-reference mechanism. If more than one result is to be returned by a called procedure, the call-by-reference mechanism should be used.

In the assembly language we do not get as much help as we do in high-level languages. The instruction set provides only the basic support to invoke a procedure. However, there is no support to pass arguments in the procedure call. If we want to pass arguments to the called procedure, we have to use some shared space between the callee and caller. Typically, we use either registers or the stack for this purpose. This leads to the two basic parameter passing mechanisms: register-based or stack-based. Later we give more details on these mechanisms along with some examples.

Our goal in this chapter is to introduce assembly language procedures. We continue our discussion of procedures in the next chapter, which discusses passing a variable number of arguments, local variables, and multimodule programs.

## What Is a Stack?

Conceptually, a stack is a last-in-first-out (LIFO) data structure. The operation of a stack is analogous to the stack of trays you find in cafeterias. The first tray removed from the stack of trays would be the last tray that had been placed on the stack. There are two operations associated with a stack: insertion and deletion. If we view the stack as a linear array of elements, stack insertion and deletion operations are restricted to one end of the array. Thus, the only element that is directly accessible is the element at the top-of-stack (TOS). In stack terminology, insert and delete operations are referred to as *push* and *pop* operations, respectively.

There is another related data structure, the *queue*. A queue can be considered as a linear array with insertions done at one end of the array and deletions at the other end. Thus, a queue is a first-in-first-out (FIFO) data structure.

As an example of a stack, let us assume that we are inserting numbers 1000 through 1003 into a stack in ascending order. The state of the stack can be visualized as shown in Figure 11.2. The arrow points to the top-of-stack. When the numbers are deleted from the stack, the numbers will come out in the reverse order of insertion. That is, 1003 is removed first, then 1002, and so on. After the deletion of the last number, the stack is said to be in the empty state (see Figure 11.3).

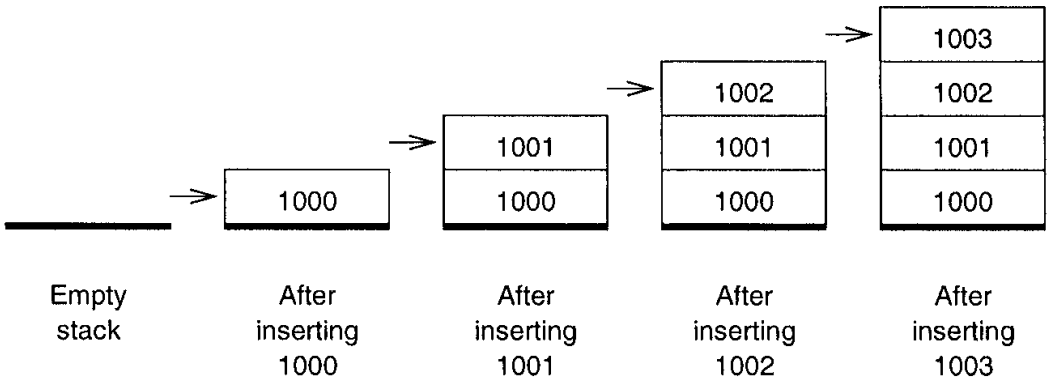



**Figure 11.2** An example showing stack growth: Numbers 1000 through 1003 are inserted in ascending order. The arrow points to the top-of-stack.

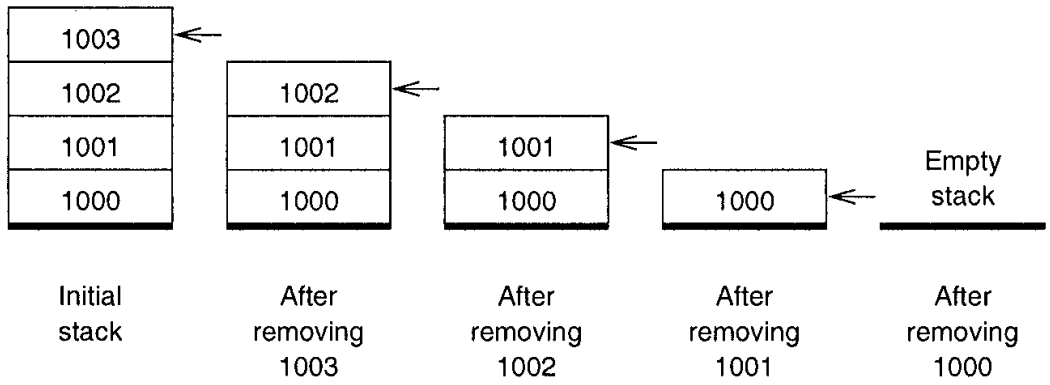



**Figure 11.3** Deletion of data items from the stack: The arrow points to the top-of-stack.

In contrast, a queue maintains the order. Suppose that the numbers 1000 through 1003 are inserted into a queue as in the stack example. When removing the numbers from the queue, the first number to enter the queue would be the one to come out first. Thus, the numbers deleted from the queue would maintain their insertion order.

# Implementation of the Stack

The memory space reserved in the stack segment is used to implement the stack. The registers SS and ESP are used to implement the stack. The top-of-stack, which points to the last item inserted into the stack, is indicated by SS:ESP, with the SS register pointing to the beginning of the stack segment, and the ESP register giving the offset value of the last item inserted.

The key stack implementation characteristics are as follows:

- Only words (i.e., 16-bit data) or doublewords (i.e., 32-bit data) are saved on the stack, never a single byte.
- The stack grows toward lower memory addresses. Since we graphically represent memory with addresses increasing from the bottom of a page to the top, we say that the stack grows *downward*.

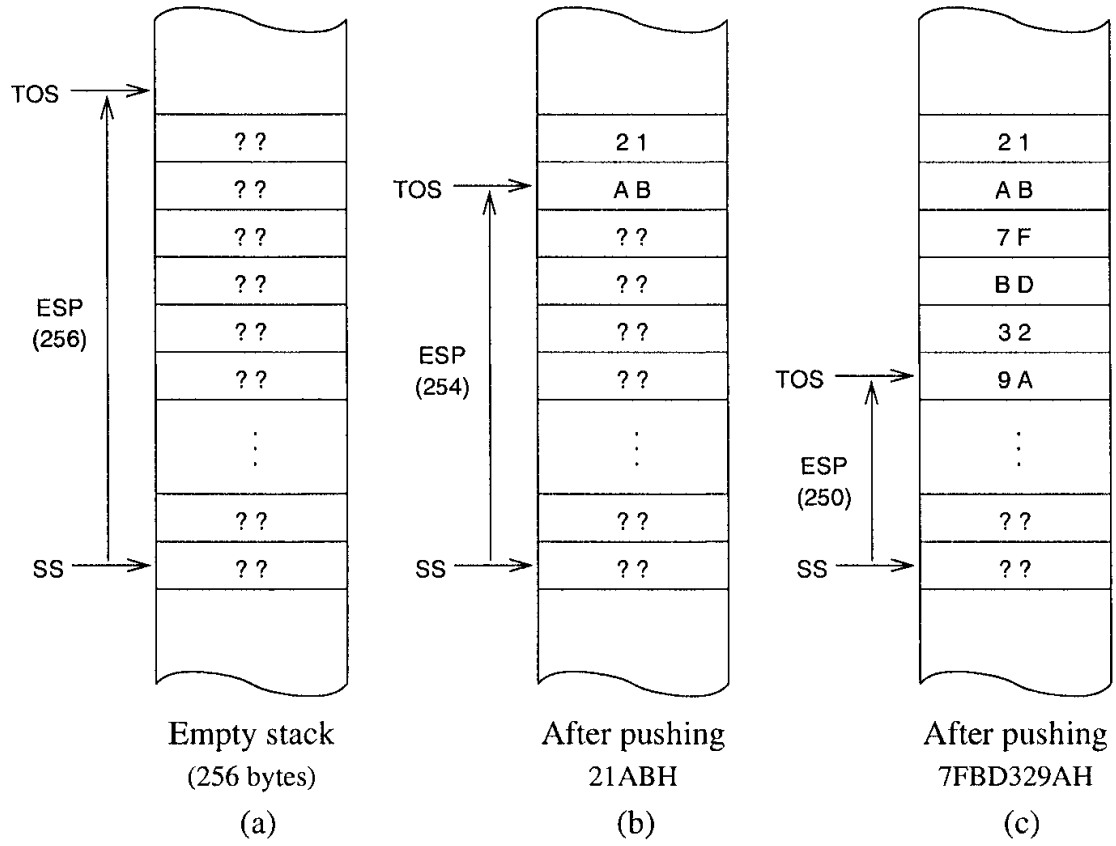



**Figure 11.4** Stack implementation in the IA-32 architecture: SS:ESP points to the top-of-stack.

- Top-of-stack (TOS) always points to the last data item placed on the stack. The TOS always points to the lower byte of the last word pushed onto the stack. For example, when we push 21ABH onto the stack, the TOS points to ABH byte as shown in Figure 11.4.

Figure 11.4*a* shows an empty stack with 256 bytes of memory for stack operations. When the stack is initialized, TOS points to a byte just outside the reserved stack area. It is an error to read from an empty stack as this causes a *stack underflow*.

When a word is pushed onto the stack, ESP is first decremented by two, and then the word is stored at SS:ESP. Since the IA-32 processors use the little-endian byte order, the higher-order byte is stored in the higher memory address. For instance, when we push 21ABH, the stack expands by two bytes, and ESP is decremented by two to point to the last data item, as shown in Figure 11.4*b*. The stack shown in Figure 11.4*c* results when we expand the stack further by four more bytes by pushing the doubleword 7FBD329AH onto the stack.

The stack full condition is indicated by the zero offset value (i.e., ESP = 0). If we try to insert a data item into a full stack, *stack overflow* occurs. Both stack underflow and overflow are programming errors and should be handled with care.

Retrieving a 32-bit data item from the stack causes the offset value to increase by four to point to the next data item on the stack. For example, if we retrieve a doubleword from the stack shown in Figure 11.5*a*, we get 7FBD329AH from the stack and ESP is updated, as shown in Figure 11.5*b*. Notice that the four memory locations retain their values. However, since TOS is updated, these four locations will be used to store the next data value pushed onto the stack, as shown in Figure 11.5*c*.

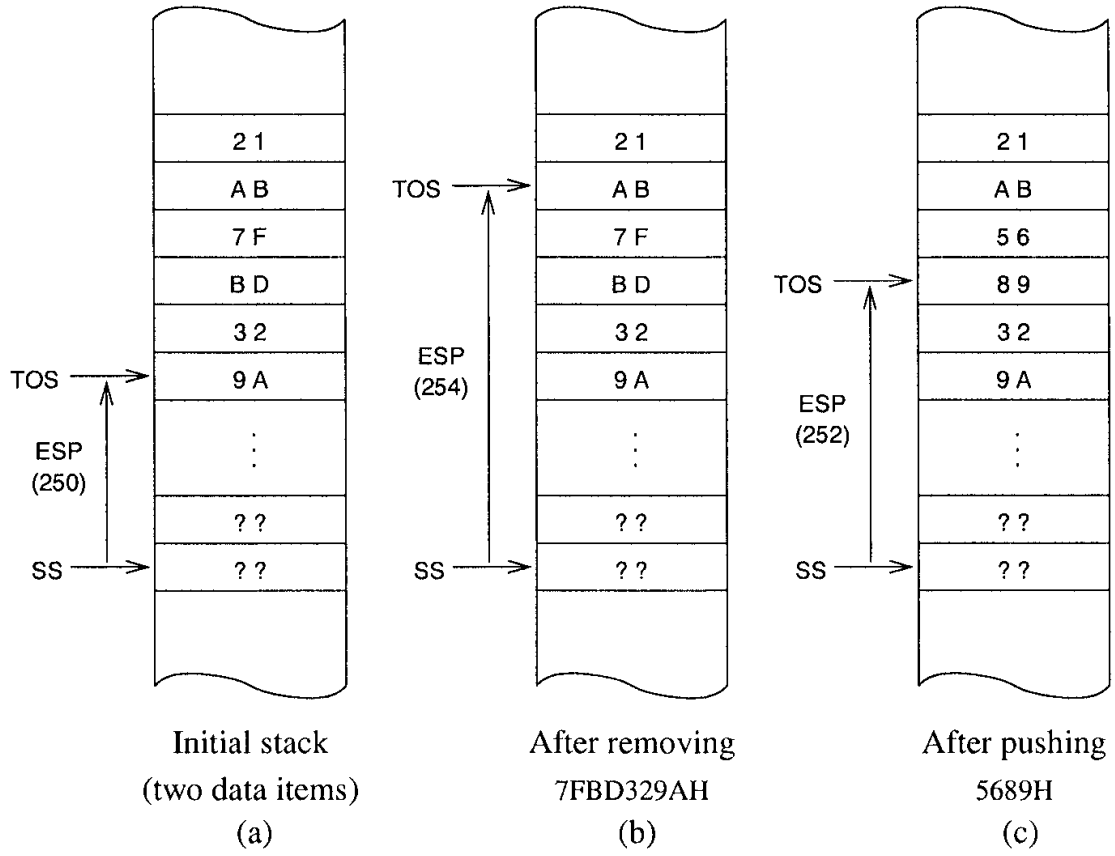



**Figure 11.5** An example showing stack insert and delete operations.

# Stack Operations

## Basic Instructions

The stack data structure allows two basic operations: insertion of a data item into the stack (called the *push* operation) and deletion of a data item from the stack (called the *pop* operation). These two operations are allowed on word or doubleword data items. The syntax is

```
push    source
pop     destination
```

The operand of these two instructions can be a 16- or 32-bit general-purpose register, segment register, or a word or doubleword in memory. In addition, `source` for the push instruction can be an immediate operand of size 8, 16, or 32 bits. Table 11.1 summarizes the two stack operations.

On an empty stack shown in Figure 11.4*a* the statements

```
push    21ABH
push    7FBD329AH
```

would result in the stack shown in Figure 11.5*a*. Executing the statement

```
pop     EBX
```

on this stack would result in the stack shown in Figure 11.5*b* with the register EBX receiving 7FBD329AH.

**Table 11.1** Stack operations on 16- and 32-bit data

| | | |
|---|---|---|
| `push source16` | ESP = ESP − 2<br>SS:ESP = `source16` | ESP is first decremented by 2 to modify TOS. Then the 16-bit data from `source16` is copied onto the stack at the new TOS. The stack expands by 2 bytes. |
| `push source32` | ESP = ESP − 4<br>SS:ESP = `source32` | ESP is first decremented by 4 to modify TOS. Then the 32-bit data from `source32` is copied onto the stack at the new TOS. The stack expands by 4 bytes. |
| `pop dest16` | `dest16` = SS:ESP<br>ESP = ESP + 2 | The data item located at TOS is copied to `dest16`. Then ESP is incremented by 2 to update TOS. The stack shrinks by 2 bytes. |
| `pop dest32` | `dest32` = SS:ESP<br>ESP = ESP + 4 | The data item located at TOS is copied to `dest32`. Then ESP is incremented by 4 to update TOS. The stack shrinks by 4 bytes. |

## Additional Instructions

The instruction set supports two special instructions for stack manipulation. These instructions can be used to save or restore the flags and general-purpose registers.

**Stack Operations on Flags** The `push` and `pop` operations cannot be used to save or restore the flags register. For this, two special versions of these instructions are provided:

```
pushfd    (push 32-bit flags)
popfd     (pop 32-bit flags)
```

These instructions do not need any operands. For operating on the 16-bit flags register (FLAGS), we can use `pushfw` and `popfw` instructions. If we use `pushf` the default operand size selects either `pushfd` or `pushfw`. In our programs, since our default is 32-bit operands, `pushf` is used as an alias for `pushfd`. However, we use `pushfd` to make the operand size explicit. Similarly, `popf` can be used as an alias for either `popfd` or `popfw`.

**Stack Operations on All General-Purpose Registers** The instruction set also has special `pusha` and `popa` instructions to save and restore the eight general-purpose registers. The `pushad` saves the 32-bit general-purpose registers EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI. These registers are pushed in the order specified. The last register pushed is the EDI register. The `popad` restores these registers except that it will not copy the ESP value (i.e., the ESP value is not loaded into the ESP register as part of the `popad` instruction). The corresponding instructions for the 16-bit registers are `pushaw` and `popaw`. These instructions are useful in procedure calls, as we will show later. Like the `pushf` and `popf` instructions, we can use `pusha` and `popa` as aliases.

# Uses of the Stack

The stack is used for three main purposes: as a scratchpad to temporarily store data, for transfer of program control, and for passing parameters during a procedure call.

## Temporary Storage of Data

The stack can be used as a scratchpad to store data on a temporary basis. For example, consider exchanging the contents of two 32-bit variables that are in the memory: `value1` and `value2`. We cannot use

```
xchg    value1,value2       ; illegal
```

because both operands of `xchg` are in the memory. The code

```
mov     EAX,value1
mov     EBX,value2
mov     value1,EBX
mov     value2,EAX
```

works, but it uses two 32-bit registers. This code requires four memory operations. However, due to the limited number of general-purpose registers, finding spare registers that can be used for temporary storage is nearly impossible in almost all programs.

What if we need to preserve the contents of the EAX and EBX registers? In this case, we need to save these registers before using them and restore them later as shown below:

```
        . . .
;save EAX and EBX registers on the stack
     push   EAX
     push   EBX
     ;EAX and EBX registers can now be used
     mov    EAX,value1
     mov    EBX,value2
     mov    value1,EBX
     mov    value2,EAX
;restore EAX and EBX registers from the stack
     pop    EBX
     pop    EAX
        . . .
```

This code requires eight memory accesses. Because the stack is a LIFO data structure, the sequence of `pop` instructions is a mirror image of the `push` instruction sequence.

An elegant way of exchanging the two values is

```
push    value1
push    value2
pop     value1
pop     value2
```

Notice that the above code does not use any general-purpose registers and requires eight memory operations as in the other example. Another point to note is that `push` and `pop` instructions allow movement of data from memory to memory (i.e., between data and stack segments). This

is a special case because mov instructions do not allow memory-to-memory data transfer. Stack operations are an exception. String instructions, discussed in Chapter 17, also allow memory-to-memory data transfer.

Stack is frequently used as a scratchpad to save and restore registers. The necessity often arises when we need to free up a set of registers so they can be used by the current code. This is often the case with procedures as we will show later.

It should be clear from these examples that the stack grows and shrinks during the course of a program execution. It is important to allocate enough storage space for the stack, as stack overflow and underflow could cause unpredictable results, often causing system errors.

### Transfer of Control

The previous discussion concentrated on how we, as programmers, can use the stack to store data temporarily. The stack is also used by some instructions to store data temporarily. In particular, when a procedure is called, the return address of the instruction is stored on the stack so that the control can be transferred back to the calling program. A detailed discussion of this topic is in the next section.

### Parameter Passing

Another important use of the stack is to act as a medium to pass parameters to the called procedure. The stack is extensively used by high-level languages to pass parameters. A discussion on the use of the stack for parameter passing is deferred to a later section.

## Procedure Instructions

The instruction set provides call and ret (return) instructions to write procedures in the assembly language. The call instruction can be used to invoke a procedure, and has the format

```
call    proc-name
```

where proc-name is the name of the procedure to be called. The assembler replaces proc-name by the offset value of the first instruction of the called procedure.

### How Is Program Control Transferred?

The offset value provided in the call instruction is not the absolute value (i.e., offset is not relative to the start of the code segment pointed to by the CS register), but a relative displacement in bytes from the instruction following the call instruction. Let us look at the example in Figure 11.6.

After the call instruction of main has been fetched, the EIP register points to the next instruction to be executed (i.e., EIP = 00000007H). This is the instruction that should be executed after completing the execution of sum procedure. The processor makes a note of this by pushing the contents of the EIP register onto the stack.

Now, to transfer control to the first instruction of the sum procedure, the EIP register would have to be loaded with the offset value of the

```
push    EBP
```

instruction in sum. To do this, the processor adds the 32-bit relative displacement found in the call instruction to the contents of the EIP register. Proceeding with our example, the machine

```
 offset     machine code
(in hex)     (in hex)
                              main:
                                        . . .
00000002 E816000000            call    sum
00000007 89C3                  mov     EBX,EAX
                                        . . .
                              ; end of main procedure
;*****************************************************
                              sum:
0000001D    55                    push    EBP
                                        . . .
                              ; end of sum procedure
;*****************************************************
                              avg:
                                        . . .
00000028    E8F0FFFFFF            call    sum
0000002D    89D8                  mov     EAX,EBX
                                        . . .
                              ; end of avg procedure
;*****************************************************
```

**Figure 11.6** An example to illustrate the transfer of program control.

language encoding of the `call` instruction, which requires five bytes, is E816000000H. The first byte E8H is the opcode for the `call` and the next four bytes give the (signed) relative displacement in bytes. In this example, it is the difference between 0000001DH (offset of the `push EBP` instruction in `sum`) and 00000007H (offset of the instruction `mov EBX,EAX` in `main`). Therefore, the displacement should be 0000001DH − 00000007H = 00000016H. This is the displacement value encoded in the `call` instruction. Note that this displacement value in this instruction is shown in the little-endian order, which is equal to 00000016H. Adding this difference to the contents of the EIP register leaves the EIP register pointing to the first instruction of `sum`.

Note that the procedure call in `main` is a forward call, and therefore the relative displacement is a positive number. As an example of a backward procedure call, let us look at the `sum` procedure call in the `avg` procedure. In this case, the program control has to be transferred back. That is, the displacement is a negative value. Following the explanation given in the last paragraph, we can calculate the displacement as 0000001DH − 0000002DH = FFFFFFF0H. Since negative numbers are expressed in 2's complement notation, FFFFFFF0H corresponds to −10H (i.e., −16D), which is the displacement value in bytes.

The following is a summary of the actions taken during a procedure call:

```
ESP = ESP − 2                        ; push return address onto the stack
SS:ESP = EIP
EIP = EIP + relative displacement    ; update EIP to point to the procedure
```

The relative displacement is a signed 32-bit number to accommodate both forward and backward procedure calls.

### The ret Instruction

The `ret` (return) instruction is used to transfer control from the called procedure to the calling procedure. Return transfers control to the instruction following the `call` (the `mov EBX,EAX` instruction in our example). How will the processor know where this instruction is located? Remember that the processor made a note of this when the `call` instruction was executed. When the `ret` instruction is executed, the return address from the stack is recovered. The actions taken during the execution of the `ret` instruction are

```
EIP = SS:ESP    ; pop return address at TOS into IP
ESP = ESP + 4   ; update TOS by adding 4 to ESP
```

An optional integer may be included in the `ret` instruction, as in

```
ret 8
```

The details on this optional number are covered later.

## Our First Program

In our first procedure example, two parameter values are passed onto the called procedure via the general-purpose registers. The procedure `sum` receives two integers in the CX and DX registers and returns the sum of these two integers via AX. No check is done to detect the overflow condition. The main program, shown in Program 11.1, requests two integers from the user and displays the sum on the screen.

**Program 11.1** Parameter passing by call-by-value using registers

```
 1:  ;Parameter passing via registers                PROCEX1.ASM
 2:  ;
 3:  ;         Objective: To show parameter passing via registers.
 4:  ;             Input: Requests two integers from the user.
 5:  ;            Output: Outputs the sum of the input integers.
 6:  %include "io.mac"
 7:  .DATA
 8:  prompt_msg1  DB   "Please input the first number: ",0
 9:  prompt_msg2  DB   "Please input the second number: ",0
10:  sum_msg      DB   "The sum is ",0
11:
12:  .CODE
13:        .STARTUP
14:        PutStr  prompt_msg1     ; request first number
15:        GetInt  CX              ; CX = first number
16:
17:        PutStr  prompt_msg2     ; request second number
18:        GetInt  DX              ; DX = second number
19:
20:        call    sum             ; returns sum in AX
21:        PutStr  sum_msg         ; display sum
22:        PutInt  AX
23:        nwln
```

```
24:  done:
25:         .EXIT
26:
27:  ;-----------------------------------------------------------
28:  ;Procedure sum receives two integers in CX and DX.
29:  ;The sum of the two integers is returned in AX.
30:  ;-----------------------------------------------------------
31:  sum:
32:         mov     AX,CX           ; sum = first number
33:         add     AX,DX           ; sum = sum + second number
34:         ret
```

# Parameter Passing

Parameter passing in assembly language is different and more complicated than that used in high-level languages. In the assembly language, the calling procedure first places all the parameters needed by the called procedure in a mutually accessible storage area (usually registers or memory). Only then can the procedure be invoked. There are two common methods depending on the type of storage area used to pass parameters: *register method* or *stack method*. As their names imply, the register method uses general-purpose registers to pass parameters, and the stack is used in the other method.

## Register Method

In the register method, the calling procedure places the necessary parameters in the general-purpose registers before invoking the procedure, as we did in the last example. Next, let us look at the advantages and disadvantages of passing parameters using the register method.

**Pros and Cons of the Register Method** The register method has its advantages and disadvantages. These are summarized here.

### Advantages

1. The register method is convenient and easier for passing a small number of arguments.
2. This method is also faster because all the arguments are available in registers.

### Disadvantages

1. The main disadvantage is that only a few arguments can be passed by using registers, as there are a limited number of general-purpose registers available in the CPU.
2. Another problem is that the general-purpose registers are often used by the calling procedure for some other purpose. Thus, it is necessary to temporarily save the contents of these registers on the stack to free them for use in parameter passing before calling a procedure, and restore them after returning from the called procedure. In this case, it is difficult to realize the second advantage listed above, as the stack operations involve memory access.

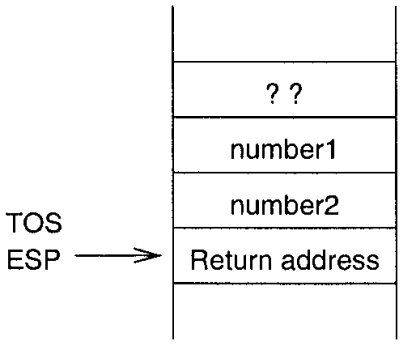



**Figure 11.7** Stack state after the `sum` procedure call: Return address is the EIP value pushed onto the stack as part of executing the call instruction.

### Stack Method

In this method of parameter passing, all arguments required by a procedure are pushed onto the stack before the procedure is called. As an example, let us consider passing the two parameters required by the `sum` procedure shown in Program 11.1. This can be done by

```
push    number1
push    number2
call    sum
```

After executing the call instruction, which automatically pushes the EIP contents onto the stack, the stack state is shown in Figure 11.7.

Reading the two arguments—`number1` and `number2`—is tricky. Since the parameter values are buried inside the stack, first we have to pop the EIP value to read the two arguments. This, for example, can be done by

```
pop     EAX
pop     EBX
pop     ECX
```

in the `sum` procedure. Since we have removed the return address (EIP) from the stack, we will have to restore it by

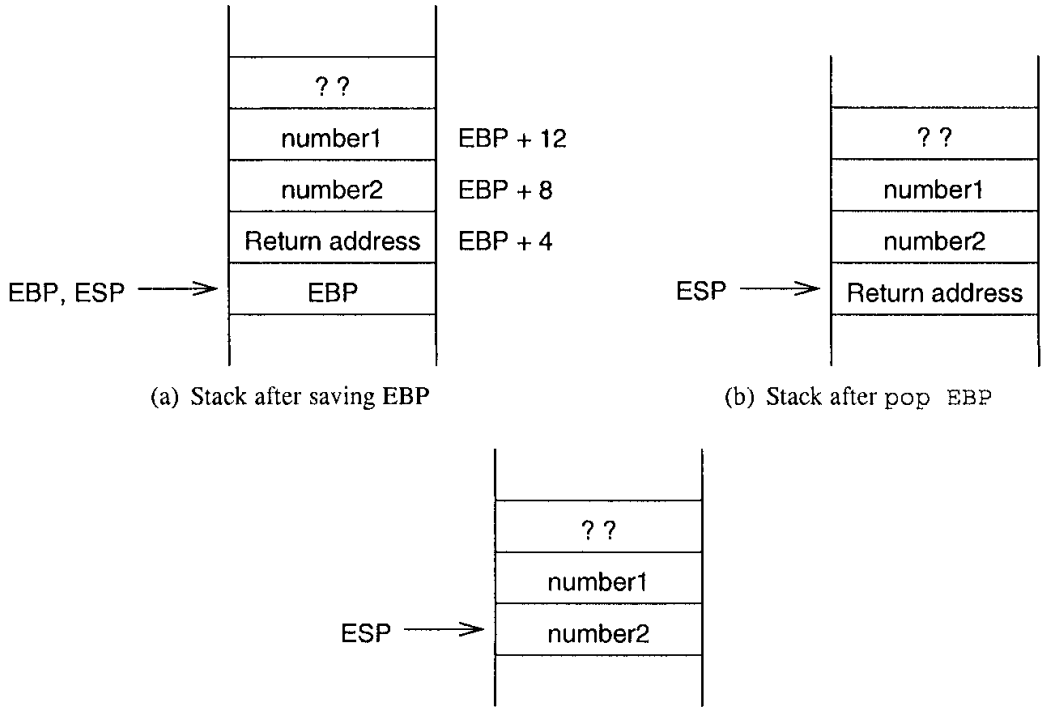
```
push    EAX
```

so that TOS is pointing to the return address.

The main problem with this code is that we need to set aside general-purpose registers to copy parameter values. This means that the calling procedure cannot use these registers for any other purpose. Worse still, what if you want to pass 10 parameters? One way to free up registers is to copy the parameters from the stack to local data variables, but this is impractical and inefficient.

The best way to get parameter values is to leave them on the stack and read them from the stack as needed. Since the stack is a sequence of memory locations, ESP + 4 points to `number2`, and ESP + 6 to `number1`. Note that both `number1` and `number2` are 16-bit values. For instance,

```
mov     EBX, [ESP+4]
```




(a) Stack after saving EBP

(b) Stack after `pop EBP`

(c) Stack after `ret`

**Figure 11.8** Changes in stack state during a procedure execution.

can be used to access `number2`, but this causes a problem. The stack pointer register is updated by the push and pop instructions. As a result, the relative offset changes with the stack operations performed in the called procedure. This is not a desirable situation.

There is a better alternative: we can use the EBP register instead of ESP to specify an offset into the stack segment. For example, we can copy the value of `number2` into the EAX register by

```
mov     EBP,ESP
mov     EAX,[EBP+4]
```

This is the usual way of pointing to the parameters on the stack. Since every procedure uses the EBP register to access parameters, the EBP register should be preserved. Therefore, we should save the contents of the EBP register before executing the

```
mov     EBP,ESP
```

statement. We, of course, use the stack for this. Note that

```
push    EBP
mov     EBP,ESP
```

causes the parameter displacement to increase by four bytes, as shown in Figure 11.8*a*.

The information stored in the stack—parameters, return address, and the old EBP value—is collectively called the *stack frame*. As we show on page 256, the stack frame also consists of local

variables if the procedure uses them. The EBP value is referred to as the *frame pointer* (FP). Once the EBP value is known, we can access all items in the stack frame.

Before returning from the procedure, we should use

```
pop    EBP
```

to restore the original value of EBP. The resulting stack state is shown in Figure 11.8*b*.

The `ret` statement causes the return address to be placed in the EIP register, and the stack state after `ret` is shown in Figure 11.8*c*.

Now the problem is that the four bytes of the stack occupied by the two arguments are no longer useful. One way to free these four bytes is to increment ESP by four after the call statement, as shown below:

```
push    number1
push    number2
call    sum
add     ESP,4
```

For example, C compilers use this method to clear parameters from the stack. The above assembly language code segment corresponds to the

```
sum(number2, number1);
```

function call in C.

Rather than adjusting the stack by the calling procedure, the called procedure can also clear the stack. Note that we cannot write

```
sum:
        . . .
    add     ESP,4
    ret
```

because when `ret` is executed, ESP should point to the return address on the stack. The solution lies in the optional operand that can be specified in the `ret` statement. The format is

```
ret     optional-value
```

which results in the following sequence of actions:

EIP = SS:ESP
ESP = ESP + 4 + `optional-value`

The `optional-value` should be a number (i.e., 16-bit immediate value). Since the purpose of the optional value is to discard the parameters pushed onto the stack, this operand takes a positive value.

### Who Should Clean Up the Stack?

We have discussed the following ways of discarding the unwanted parameters on the stack:

1. clean-up is done by the calling procedure, or
2. clean-up is done by the called procedure.

If procedures require a fixed number of parameters, the second method is preferred. In this case, we write the clean-up code only once in the called procedure independent of the number of times this procedure is called. We follow this convention in our assembly language programs. However, if a procedure receives a variable number of parameters, we have to use the first method. We discuss this topic in detail in a later section.

### Preserving Calling Procedure State

It is important to preserve the contents of the registers across a procedure call. The necessity for this is illustrated by the following code:

```
            . . .
          mov     ECX,count
repeat:
        call      compute
            . . .
            . . .
        loop      repeat
            . . .
```

The code invokes the `compute` procedure `count` times. The ECX register maintains the number of remaining iterations. Recall that, as part of the `loop` instruction execution, the ECX register is decremented by 1 and, if not 0, starts another iteration.

Suppose, now, that the `compute` procedure uses the ECX register during its computation. Then, when `compute` returns control to the calling program, ECX would have changed, and the program logic would be incorrect.

Since there are a limited number of registers and registers should be used for writing efficient code, registers should be preserved. The stack is used to save registers temporarily.

### Which Registers Should Be Saved?

The answer to this question is simple: Save those registers that are used by the calling procedure but changed by the called procedure. This leads to the following question: Which procedure, the calling or the called, should save the registers?

Usually, one or two registers are used to return a value by the called procedure. Therefore, such register(s) do not have to be saved. For example, the EAX register is often used to return integer values.

In order to avoid the selection of the registers to be saved, we could save, blindly, all registers each time a procedure is invoked. For instance, we could use the `pushad` instruction (see page 237). But such an action results in unnecessary overhead.

If the calling procedure were to save the necessary registers, it needs to know the registers used by the called procedure. This causes two serious difficulties:

1. Program maintenance would be difficult because, if the called procedure were modified later on and a different set of registers used, every procedure that calls this procedure would have to be modified.
2. Programs tend to be longer because if a procedure is called several times, we have to include the instructions to save and restore the registers each time the procedure is called.

For these reasons, we assume that the called procedure saves the registers that it uses and restores them before returning to the calling procedure. This also conforms to the modular program design principles.

### When to Use pusha

The `pusha` instruction is useful in certain instances, but not all. We identify some instances where `pusha` is not useful. First, what if some of the registers saved by `pusha` are used for returning

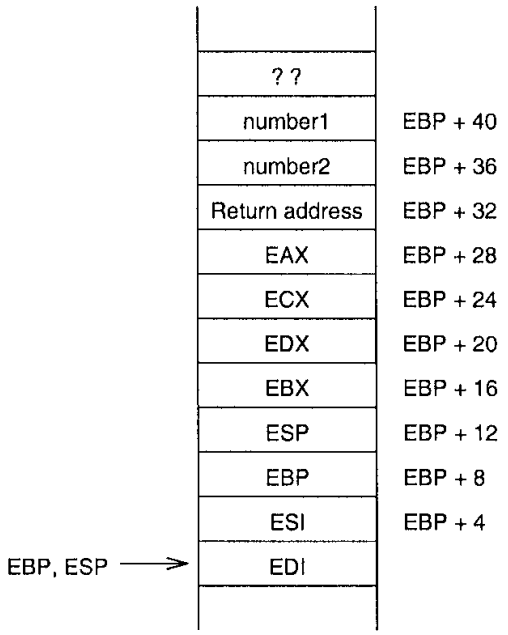



**Figure 11.9** Stack state after `pusha`.

results? For instance, EAX register is often used to return integer results. In this case `pusha` is not really useful, as `popa` destroys the result to be returned to the calling procedure. Second, since `pusha` introduces more overhead, it may be worthwhile to use the `push` instruction if we want to save only one or two registers. Of course, the other side of the coin is that `pusha` improves readability of code and reduces memory required for the instructions.

When `pusha` is used to save registers, it modifies the offset of the parameters. Note that

```
pusha
mov     EBP,ESP
```

causes the stack state, shown in Figure 11.9, to be different from that shown in Figure 11.8*a* on page 244. You can see that the offset of `number1` and `number2` increases.

## ENTER and LEAVE Instructions

The instruction set has two instructions to facilitate stack frame allocation and release on procedure entry and exit. The `enter` instruction can be used to allocate a stack frame on entering a procedure. The format is

```
enter     bytes,level
```

The first operand `bytes` specifies the number of bytes of local variable storage we want on the new stack frame. We discuss local variables in the next chapter. Until then, we set the first operand to zero. The second operand `level` gives the nesting level of the procedure. If we specify a nonzero level, it copies `level` stack frame pointers into the new frame from the preceding stack frame. In all our examples, we set the second operand to zero. Thus the statement

```
enter     XX,0
```

is equivalent to

```
push    EBP
mov     EBP,ESP
sub     ESP,XX
```

The `leave` instruction releases the stack frame allocated by the `enter` instruction. It does not take any operands. The `leave` instruction effectively performs the following:

```
mov     ESP,EBP
pop     EBP
```

We use the `leave` instruction before the `ret` instruction as shown in the following template for procedures:

```
proc-name:
    enter   XX,0
        . . .
    procedure body
        . . .
    leave
    ret     YY
```

As we show in the next chapter (page 259), the `XX` value is nonzero only if our procedure needs some local variable space on the stack frame. The value `YY` is used to clear the arguments passed on to the procedure.

# Illustrative Examples

In this section, we use several examples to illustrate register-based and stack-based parameter passing.

**Example 11.1** *Parameter passing by call-by-reference using registers.*
This example shows how parameters can be passed by call-by-reference using the register method. The program requests a character string from the user and displays the number of characters in the string (i.e., string length). The string length is computed by the `str_len` function. This function scans the input string for the NULL character while keeping track of the number of characters in the string. The pseudocode is shown below:

```
str_len (string)
    index := 0
    length := 0
    while (string[index] ≠ NULL)
        index := index + 1
        length := length + 1      { AX is used for string length}
    end while
    return (length)
end str_len
```

The `str_len` function receives a pointer to the string in EBX and returns the string length in the EAX register. The program listing is given in Program 11.2. The `main` procedure executes

```
mov     EBX,string
```

to place the address of `string` in EBX (line 22) before invoking the procedure on line 23. Note that even though the procedure modifies the EBX register during its execution, it restores the original value of EBX by saving its value initially on the stack (line 35) and restoring it (line 44) before returning to the `main` procedure.

**Program 11.2** Parameter passing by call-by-reference using registers

```
 1:  ;Parameter passing via registers                    PROCEX2.ASM
 2:  ;
 3:  ;         Objective: To show parameter passing via registers
 4:  ;             Input: Requests a character string from the user.
 5:  ;            Output: Outputs the length of the input string.
 6:
 7:  %include "io.mac"
 8:  BUF_LEN     EQU  41          ; string buffer length
 9:
10:  .DATA
11:  prompt_msg  db  "Please input a string: ",0
12:  length_msg  db  "The string length is ",0
13:
14:  .UDATA
15:  string      resb  BUF_LEN    ;input string < BUF_LEN chars.
16:
17:  .CODE
18:        .STARTUP
19:        PutStr  prompt_msg     ; request string input
20:        GetStr  string,BUF_LEN ; read string from keyboard
21:
22:        mov     EBX,string     ; EBX = string address
23:        call    str_len        ; returns string length in AX
24:        PutStr  length_msg     ; display string length
25:        PutInt  AX
26:        nwln
27:  done:
28:        .EXIT
29:
30:  ;---------------------------------------------------------
31:  ;Procedure str_len receives a pointer to a string in BX.
32:  ;String length is returned in AX.
33:  ;---------------------------------------------------------
34:  str_len:
35:        push    EBX
36:        sub     AX,AX          ; string length = 0
37:  repeat:
38:        cmp     byte [EBX],0   ; compare with NULL char.
39:        je      str_len_done   ; if NULL we are done
40:        inc     AX             ; else, increment string length
41:        inc     EBX            ; point BX to the next char.
```

```
42:         jmp     repeat          ;  and repeat the process
43:  str_len_done:
44:         pop     EBX
45:         ret
```

**Example 11.2** *Parameter passing by call-by-value using the stack.*
This is the stack counterpart of Program 11.1, which passes two integers to the procedure sum. The procedure returns the sum of these two integers in the AX register. The program listing is given in Program 11.3.

The program requests two integers from the user. It reads the two numbers into the CX and DX registers using GetInt (lines 16 and 19). Since the stack is used to pass the two numbers, we have to place them on the stack before calling the sum procedure (see lines 21 and 22). The state of the stack after the control is transferred to sum is shown in Figure 11.7 on page 243.

As discussed before, the EBP register is used to access the two parameters from the stack. Therefore, we have to save EBP itself on the stack. We do this by using the enter instruction (line 35), which changes the stack state to that in Figure 11.8*a* on page 244.

The original value of EBP is restored at the end of the procedure using the leave instruction (line 38). Accessing the two numbers follows the explanation given in Section 11. Note that the first number is at EBP + 10, and the second one at EBP + 8. As in our first example on page 241, no overflow check is done by sum. Control is returned to main by

```
ret    4
```

because sum has received two parameters requiring a total space of four bytes on the stack. This ret statement clears number1 and number2 from the stack.

**Program 11.3** Parameter passing by call-by-value using the stack

```
 1:  ;Parameter passing via the stack                    PROCEX3.ASM
 2:  ;
 3:  ;         Objective: To show parameter passing via the stack.
 4:  ;             Input: Requests two integers from the user.
 5:  ;            Output: Outputs the sum of the input integers.
 6:  %include "io.mac"
 7:
 8:  .DATA
 9:  prompt_msg1  db   "Please input the first number: ",0
10:  prompt_msg2  db   "Please input the second number: ",0
11:  sum_msg      db   "The sum is ",0
12:
13:  .CODE
14:         .STARTUP
15:         PutStr  prompt_msg1     ; request first number
16:         GetInt  CX              ; CX = first number
17:
18:         PutStr  prompt_msg2     ; request second number
19:         GetInt  DX              ; DX = second number
```

```
20:
21:         push    CX              ; place first number on stack
22:         push    DX              ; place second number on stack
23:         call    sum             ; returns sum in AX
24:         PutStr  sum_msg         ; display sum
25:         PutInt  AX
26:         nwln
27:   done:
28:         .EXIT
29:
30:   ;-----------------------------------------------------------
31:   ;Procedure sum receives two integers via the stack.
32:   ;The sum of the two integers is returned in AX.
33:   ;-----------------------------------------------------------
34:   sum:
35:         enter   0,0             ; save EBP
36:         mov     AX,[EBP+10]     ; sum = first number
37:         add     AX,[EBP+8]      ; sum = sum + second number
38:         leave                   ; restore EBP
39:         ret     4               ; return and clear parameters
```

**Example 11.3** *Parameter passing by call-by-reference using the stack.*
This example shows how the stack can be used for parameter passing using the call-by-reference mechanism. The procedure swap receives two pointers to two characters and interchanges them. The program, shown in Program 11.4, requests a string from the user and displays the input string with the first two characters interchanged.

**Program 11.4** Parameter passing by call-by-reference using the stack

```
 1:  ;Parameter passing via the stack                  PROCSWAP.ASM
 2:  ;
 3:  ;        Objective: To show parameter passing via the stack.
 4:  ;            Input: Requests a character string from the user.
 5:  ;           Output: Outputs the input string with the first
 6:  ;                   two characters swapped.
 7:
 8:  BUF_LEN     EQU  41          ; string buffer length
 9:  %include "io.mac"
10:
11:  .DATA
12:  prompt_msg  db    "Please input a string: ",0
13:  output_msg  db    "The swapped string is: ",0
14:
15:  .UDATA
16:  string      resb  BUF_LEN      ;input string < BUF_LEN chars.
17:
18:  .CODE
```

```
19:         .STARTUP
20:         PutStr  prompt_msg      ; request string input
21:         GetStr  string,BUF_LEN  ; read string from the user
22:
23:         mov     EAX,string      ; EAX = string[0] pointer
24:         push    EAX
25:         inc     EAX             ; EAX = string[1] pointer
26:         push    EAX
27:         call    swap            ; swaps the first two characters
28:         PutStr  output_msg      ; display the swapped string
29:         PutStr  string
30:         nwln
31:   done:
32:         .EXIT
33:
34:   ;-------------------------------------------------------------
35:   ;Procedure swap receives two pointers (via the stack) to
36:   ;characters of a string. It exchanges these two characters.
37:   ;-------------------------------------------------------------
38:   .CODE
39:   swap:
40:         enter   0,0
41:         push    EBX             ; save EBX - procedure uses EBX
42:         ; swap begins here. Because of xchg, AL is preserved.
43:         mov     EBX,[EBP+12]    ; EBX = first character pointer
44:         xchg    AL,[EBX]
45:         mov     EBX,[EBP+8]     ; EBX = second character pointer
46:         xchg    AL,[EBX]
47:         mov     EBX,[EBP+12]    ; EBX = first character pointer
48:         xchg    AL,[EBX]
49:         ; swap ends here
50:         pop     EBX             ; restore registers
51:         leave
52:         ret     8               ; return and clear parameters
```

In preparation for calling `swap`, the `main` procedure places the addresses of the first two characters of the input string on the stack (lines 23 to 26). The `swap` procedure, after saving the EBP register as in the last example, can access the pointers of the two characters at EBP + 8 and EBP + 12. Since the procedure uses the EBX register, we save it on the stack as well. Note that, once the EBP is pushed onto the stack and the ESP value is copied to EBP, the two parameters (i.e., the two character pointers in this example) are available at EBP + 8 and EBP + 12, irrespective of the other stack push operations in the procedure. This is important from the program maintenance point of view.

# Summary

The stack is a last-in-first-out data structure that plays an important role in procedure invocation and execution. It supports two operations: push and pop. Only the element at the top-of-stack is

directly accessible through these operations. The stack segment is used to implement the stack. The top-of-stack is represented by SS:ESP. In the implementation, the stack grows toward lower memory addresses (i.e., grows downward).

The stack serves three main purposes: temporary storage of data, transfer of control during a procedure call and return, and parameter passing.

When writing procedures in the assembly language, parameter passing has to be explicitly handled. Parameter passing can be done via registers or the stack. Although the register method is efficient, the stack-based method is more general. We have used several examples to illustrate the register-based and stack-based parameter passing.