PROCEDURES

CHAPTER 6

The 80x86 architecture enables implementation of procedures that are similar to those in a high-level language. In fact, 80x86 procedures can be called from high-level language programs or can call high-level language procedures. There are three main concepts involved: (1) how to transfer control from a calling program to a procedure and back, (2) how to pass parameter values to a procedure and results back from the procedure, and (3) how to write procedure code that is independent of the calling program. In addition, sometimes a procedure must allocate local variable space. The hardware stack is used to accomplish each of these jobs. This chapter begins with a discussion of the 80x86 stack. Sections 6.1 to 6.3 cover operations in 32-bit mode only, while Section 6.4 describes differences for 64-bit mode. The final section discusses macros, sometimes used to substitute for procedure calls, and used by the *io.h* file in *windows32* and *windows64* projects to call procedures.

6.1 The 80x86 Stack

32-bit programs in this text have allocated stacks with the code

```
.STACK  4096
```

This `.STACK` directive tells the assembler to reserve 4096 bytes of uninitialized storage. The operating system initializes the stack pointer register ESP to the address of the first byte above the 4096 bytes in the stack. A larger or smaller stack could be allocated, depending on the anticipated usage in the program.

64-bit programs do not use the `.STACK` directive. Stack size is changed by specifying new values for the *Stack Reserve Size* and *Stack Commit Size* properties found in PROJECT/project Properties/configuration properties/Linker/System. The default values of 1MB and 4KB, respectively, are ample for our programs.

The stack is most often used by pushing doublewords on it, or by popping them off it. This is done automatically as part of the execution of `call` and `return` instructions (see Section 6.2). It is also done manually with `push` and `pop` instructions. This section covers the mechanics of `push` and `pop` instructions, describing how they use the stack.

Source code for a push instruction has the syntax

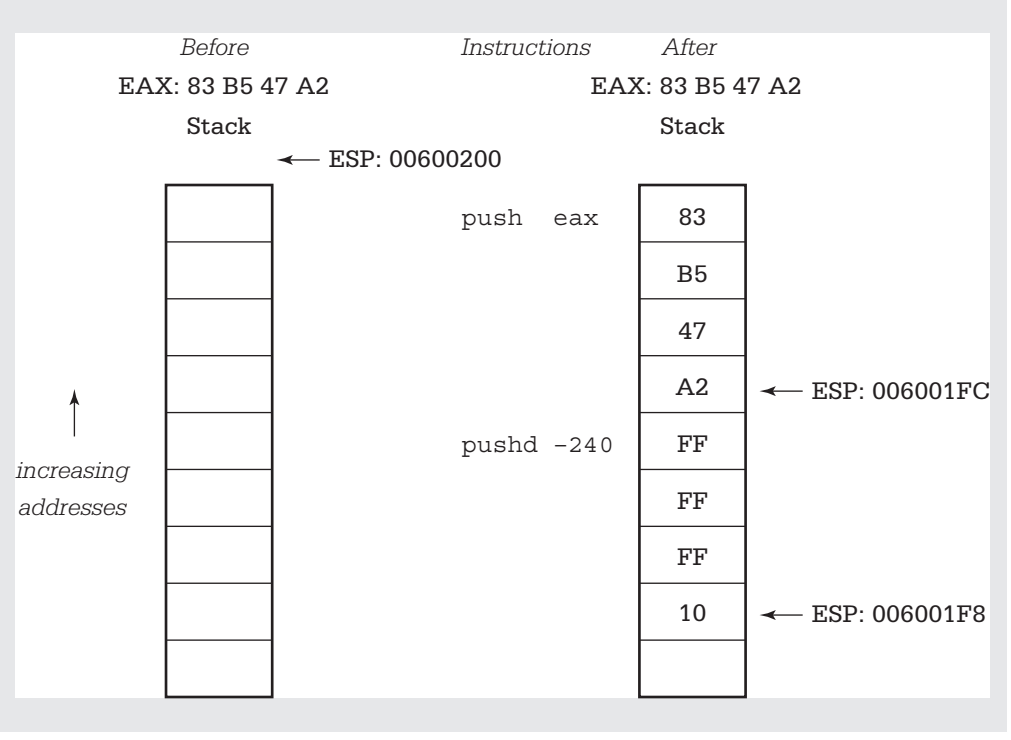
```
push source
```

The source operand can be a register 16, a register 32, a segment register, a word in memory, a doubleword in memory, an immediate byte, an immediate word, or an immediate doubleword. The only byte-size operand is immediate, and is sign-extended to a word or doubleword to get the value actually pushed on the stack. Figure 6.1 lists some allowable operand types, omitting segment registers that we will not use. The usual mnemonic for a push instruction is just `push`. However, if there is ambiguity about the size of the operand (as there would be with a small immediate value) then you can use `pushw` or `pushd` mnemonics to specify word-size or doubleword-size operands, respectively. The `WORD PTR` and `DWORD PTR` operators are used with memory operands when needed.

When a `push` instruction is executed for a doubleword-size operand, the stack pointer ESP is decremented by 4. Recall that initially ESP contains the address of the byte just above the allocated space. Subtracting 4 makes ESP point to the top doubleword in the stack. The operand is then stored at the address in ESP, that is, at the high-memory end of the stack space. Execution is similar for a word-size operand, except that ESP is decremented by 2 before the operand is stored.

EXAMPLE

We now show an example of execution of two `push` instructions. It assumes that ESP initially contains 00600200. The first `push` decrements ESP to 006001FC and then stores the contents of EAX at that address. Notice that the low-order and high-order bytes are reversed in memory. The second `push` decrements ESP to 006001F8 and stores FFFFFF10 (-240_{10}) at that address.



Operand	Opcode	Bytes of Object Code
EAX or AX	50	1
ECX or CX	51	1
EDX or DX	52	1
EBX or BX	53	1
ESP or SP	54	1
EBP or BP	55	1
ESI or SI	56	1
EDI or DI	57	1
memory word	FF	2+
memory doubleword	FF	2+
immediate byte	6A	2
immediate word	68	3
immediate doubleword	68	5

Figure 6.1

`push` instructions

You can use the debugger to watch these instructions actually execute. After you assemble a program starting with

```
mov  eax, 83b547a2h
push eax
pushd -240
```

the assembly listing displays

```
00000000 B8 83B547A2 mov  eax, 83b547a2h
00000005 50      push eax
00000006 68 FFFFFF10 pushd -240
```

This is expected from the opcodes listed in Figure 4.1 for `mov` and Figure 6.1 for `push`. Figure 6.2 gives the Visual Studio display after the EAX register has been initialized with 83B547A2. We just want to see the top few bytes of the stack, so we note that ESP contains 0041FAAC. (It might have another value at another time or on another computer.) To display the top 16 bytes, we open a memory view starting at address 0x0041FA9C. These bytes are shown on the top two lines of the memory window. Notice that the stack contains "junk" values, zeros in this case.

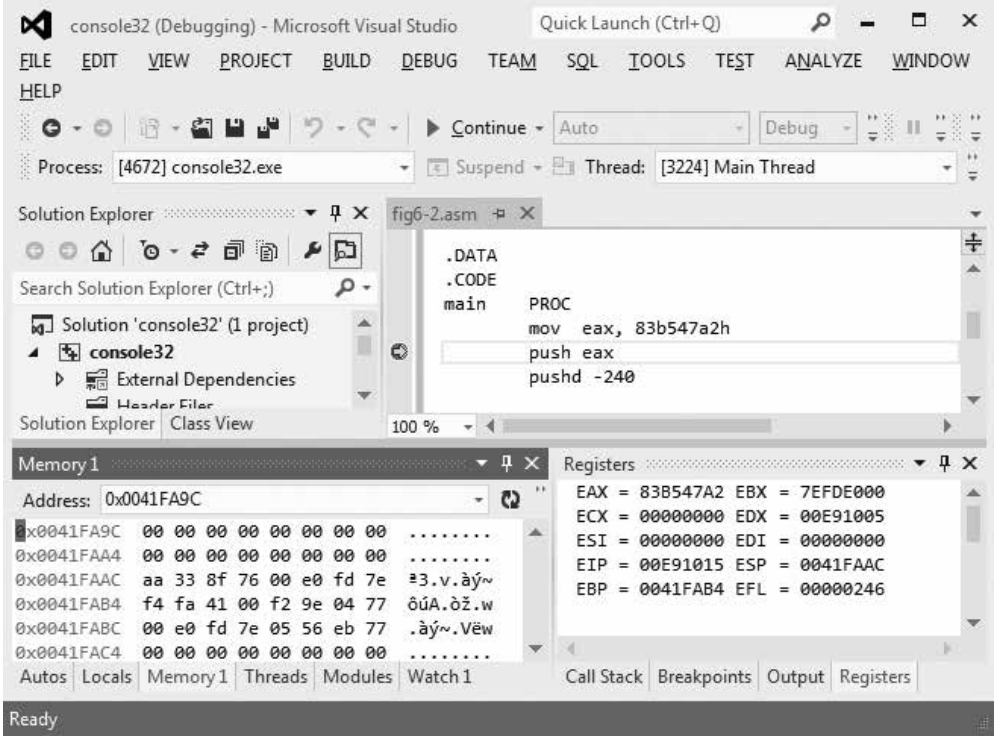


Figure 6.2

Stack test prior to push operation

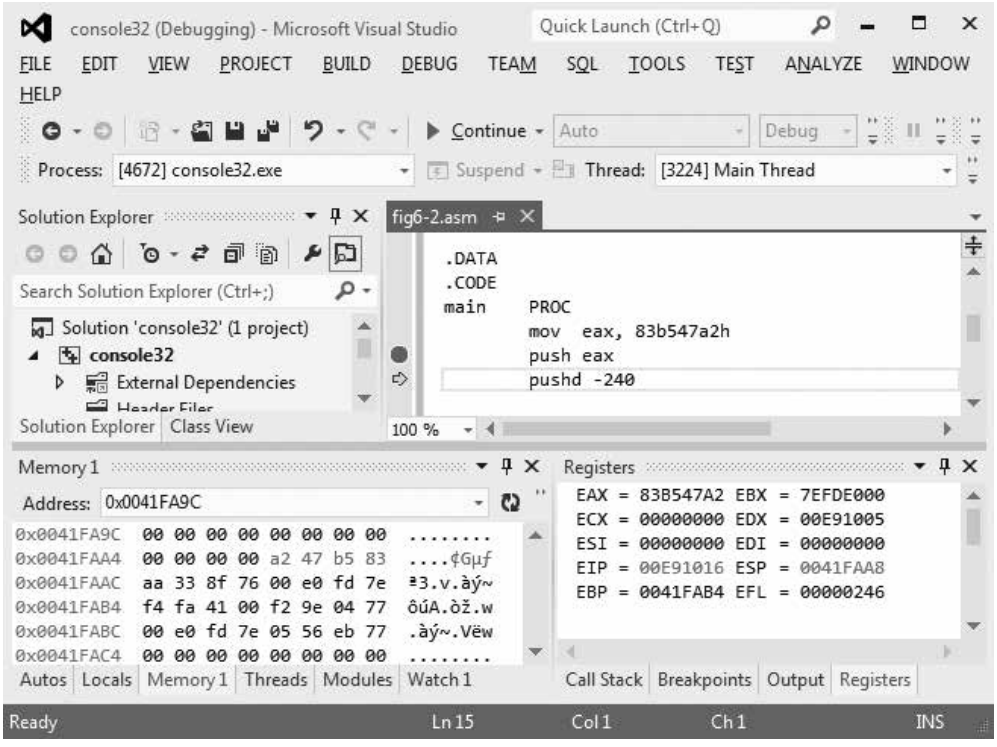


Figure 6.3

EAX has been pushed onto the stack

Now execute the `push` instruction. The resulting display is shown in Figure 6.3. Notice that ESP now contains 0041FAA8, that is, it has been decremented by 4. The last 4 bytes on the second memory line (in gray) show the doubleword stored at the new stack pointer address. The bytes from EAX have been stored backward in memory. Finally execute the `pushd` instruction. The resulting display is shown in Figure 6.4. ESP now contains 0041FAA4, again decremented by 4. The first 4 bytes of the second memory line show the value of –240, again with the bytes of FFFFFF10 stored in reverse order.

If additional operands were pushed onto the stack, ESP would be decremented further and the new values stored. No `push` instruction affects any flag bit.

Notice that a stack "grows downward," contrary to the image that you may have of a typical software stack.[1] Also notice that the only value on the stack that is readily available is the last one pushed; it is at the address in ESP. Furthermore, ESP changes frequently as you push values and as procedure calls are made. In the next section you will learn a way to establish a fixed reference point in the middle of the stack using the

[1] Of course, if you draw the picture so that lower memory addresses are at the top, then it "grows upward." The author's preference is to draw the pictures so that when ESP is decremented, its "pointer" moves down.

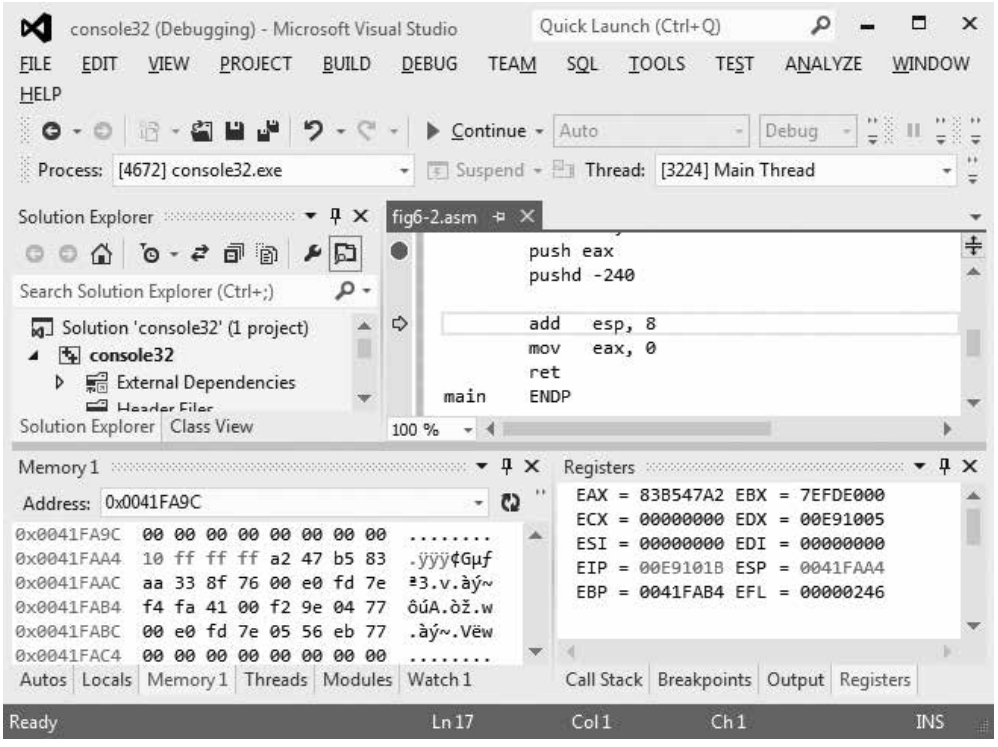


Figure 6.4

–240 has been pushed onto the stack

EBP register, so that values near that point can be accessed without having to pop off all the intermediate values.

Notice that the instruction `add esp, 8` precedes the usual exit code. This effectively removes the two values from the stack, allowing a normal exit from the program. You will see more why this is necessary later in this chapter.

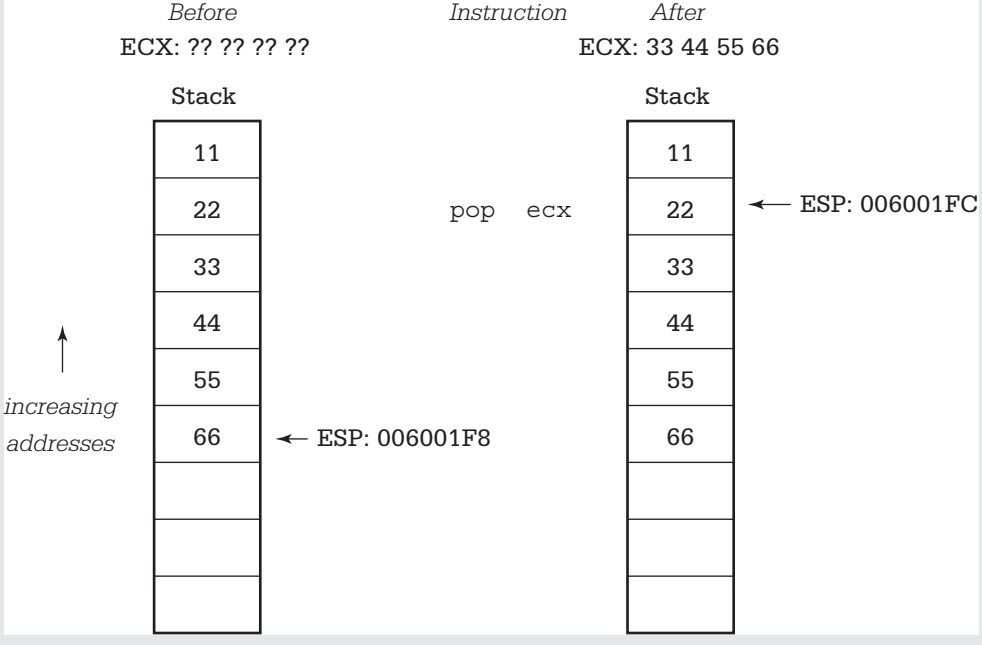
Pop instructions do the opposite job of push instructions. Each pop instruction has the format

```
pop destination
```

where *destination* can reference a word or doubleword in memory, any register 16, any register 32, or any segment register except CS. (The `push` instruction does not exclude CS.) The `pop` instruction gets a doubleword-size value from the stack by copying the doubleword at the address in ESP to the destination, then incrementing ESP by 4. The operation for a word-size value is similar, except that ESP is incremented by 2. Figure 6.5 gives information about `pop` instructions for different destination operands. Segment registers are again omitted. Pop instructions do not affect flags.

EXAMPLE

Here is an example to show how pop instructions work. The doubleword at the address in ESP is copied to ECX before ESP is incremented by 4. The values popped from the stack are physically still there even though they logically have been removed. Note again that the bytes of a doubleword are stored backward in memory in the 80x86 architecture.



Operand	Opcode	Bytes of Object Code
EAX or AX	58	1
ECX or CX	59	1
EDX or DX	5A	1
EBX or BX	5B	1
ESP or SP	5C	1
EBP or BP	5D	1
ESI or SI	5E	1
EDI or DI	5F	1
memory word	8F	2+
memory doubleword	8F	2+

Figure 6.5

pop instructions

Instruction	Opcode	Bytes of Object Code
`pushf/pushfd`	9C	1
`popf/popfd`	9D	1

Figure 6.6

`pushf` and `popf` instructions

We have noted previously that registers are a scarce resource when programming. One use of push and pop instructions is to temporarily save the contents of a register on the stack. Suppose, for example, that you are using EDX to store some program variable, but need to do a division that requires you to extend a dividend into EDX:EAX prior to the operation. One way to avoid losing the value in EDX is to push it on the stack.

```
push edx   ; save variable
cdq      ; extend dividend to quadword
idiv divisor ; divide
pop  edx   ; restore variable
```

This example assumes that you don't need the remainder the division operation puts in EDX. If you do need the remainder, it could be copied somewhere else before popping the saved value back to EDX.

As the above example shows, push and pop instructions are often used in pairs. When we examine how the stack is used to pass parameters to procedures, you will see a way to logically discard values from the stack without popping them to a destination location.

In a 32-bit environment the stack is created on a doubleword boundary, that is, the address in ESP will be a multiple of 4. It is important to keep the stack top on a doubleword boundary for certain system calls. Therefore, with few exceptions, you should always push doubleword values on the stack, even though the 80x86 architecture allows words to be used.

In addition to the ordinary push and pop instructions, there are special mnemonics to push and pop flag registers. These are `pushf` (`pushfd` for the extended flag register) and `popf` (`popfd` for the extended flag register). These are summarized in Figure 6.6. They are sometimes used in procedure code. Obviously, `popf` and `popfd` instructions change flag values; these are the only push or pop instructions that change flags.

The 80x86 architecture has `pushad` and `popad` instructions that push or pop all general-purpose registers with a single instruction. These are rarely useful and do not work in 64-bit mode, so they are not used in this text.

Exercises 6.1

1. For each instruction, give the opcode and the number of bytes of object code including prefix bytes. Assume that *double* references a doubleword in memory.

   (a) `push ax`
   (b) `pushd 10`
   *(c) `push ebp`
   (d) `pop ebx`

(e) `pop double` (f) `pop dx`
(g) `pushfd`

2. For each part of this problem, assume the "before" values when the given instructions are executed. Give the requested "after" values. Trace execution of the instructions by drawing pictures of the stack.

	Before	*Instructions*	*After*
*(a)	ESP: 06 00 10 00 ECX: 01 A2 5B 74	`push ecx` `pushd 10`	ESP, ECX
(b)	ESP: 02 00 0B 7C EBX: 12 34 56 78	`pushd 20` `push ebx`	ESP, EBX

(c)	ESP: 00 10 F8 3A
	EAX: 12 34 56 78
		`push eax`
		`pushd 30`
		`pop ebx`
		`pop ecx`
			ESP, EAX, EBX, ECX

(d)	ESP: 00 63 FB 60 EBX: 22 33 44 55 ECX: 66 77 88 99	`push ebx` `push ecx`	ESP, EBX, ECX

(e)	ESP: 00 63 FB 60
	EAX: BB CC DD EE
		`pushw 10`
		`pushw 20`
		`pop eax`
			ESP, EAX

3. Many microprocessors do not have an instruction equivalent to `xchg`. With such systems, a sequence of instructions like the following can be used to exchange the contents of two registers:

```
push eax
push ebx
pop  eax
pop  ebx
```

Explain why this sequence works to exchange the contents of the EAX and EBX registers. Compare the number of bytes of code required to execute this sequence with those required for the instruction `xchg eax,ebx`.

4. Another alternative to the `xchg` instruction is to use

```
push eax
mov  eax, ebx
pop  ebx
```

Explain why this sequence works to exchange the contents of the EAX and EBX registers. Compare the number of bytes of code required to execute this sequence with those required for the instruction `xchg eax,ebx`.

6.2 32-Bit Procedures with Value Parameters

The word **procedure** is used in high-level languages to describe a subprogram that is almost a self-contained unit. The main program or another subprogram can call a procedure by including a statement that consists of the procedure name followed by a parenthesized list of arguments to be associated with the procedure's formal parameters.

Many high-level languages distinguish between a procedure that performs an action and a function that returns a value. A **function** is similar to a procedure except that it is

called by using its name and argument list in an expression. It returns a value associated with its name; this value is then used in the expression. All subprograms in C/C++ are technically functions in this sense, but these languages allow for functions that return no value.

In assembly language and in some high-level languages the term *procedure* is used to describe both types of subprograms: those that return values and those that do not. The word *procedure* is used in both senses in this text.

Procedures are valuable in assembly language for the same reasons as in high-level languages—they help divide programs into manageable tasks and they isolate code that can be used multiple times within a single program, or that can be saved and reused in other programs. Sometimes assembly language can be used to write more efficient code than is produced by a high-level language compiler, and this code can be put in a procedure called by a high-level program that does tasks that don't need to be as efficient.

Recall the major main concepts listed in the introduction to this chapter: (1) how to transfer control from a calling program to a procedure and back, (2) how to pass parameter values to a procedure and results back from the procedure, and (3) how to write procedure code that is independent of the calling program. These can be handled in many ways in assembly language, and this section describes one particular protocol, called *cdecl* in Microsoft documentation. It is the default convention used in C programs in the Visual Studio environment. Figure 6.7 gives a complete *windows32* program that is used to illustrate aspects of this protocol.

```
; Input x and y, call procedure to evaluate 3*x+7*y, display result
; Author: R. Detmer
; Date:  6/2013
.586
.MODEL FLAT
INCLUDE io.h
.STACK 4096

.DATA
number1 DWORD  ?
number2 DWORD  ?
prompt1 BYTE  "Enter first number x", 0
prompt2 BYTE  "Enter second number y", 0
string BYTE  20 DUP (?)
resultLbl BYTE "3*x+7*y", 0
result BYTE  11 DUP (?), 0

.CODE
_MainProc PROC
    input  prompt1, string, 20   ; read ASCII characters
    atod  string     ; convert to integer
    mov   number1, eax  ; store in memory
```

Figure 6.7

Procedure example (*continues*)

```
    input  prompt2, string, 20 ; repeat for second number
    atod  string
    mov   number2, eax

    push  number2     ; 2nd parameter
    push  number1     ; 1st parameter
    call  fctn1      ; fctn1(number1, number2)
    add   esp, 8     ; remove parameters from stack

    dtoa  result, eax   ; convert to ASCII characters
    output resultLbl, result ; output label and result

    mov   eax, 0 ; exit with return code 0
    ret
_MainProc ENDP

; int fctn1(int x, int y)
; returns 3*x+7*y
fctn1  PROC
    push  ebp       ; save base pointer
    mov   ebp, esp    ; establish stack frame
    push  ebx       ; save EBX

    mov   eax, [ebp+8]   ; x
    imul  eax, 3     ; 3*x
    mov   ebx, [ebp+12]   ; y
    imul  ebx, 7     ; 7*y
    add   eax, ebx    ; 3*x + 7*y

    pop   ebx       ; restore EBX
    pop   ebp       ; restore EBP
    ret           ; return
fctn1  ENDP

END
```

Figure 6.7

Procedure example (*continued*)

The code for a procedure always follows a `.CODE` directive. The body of a procedure is bracketed by `PROC` and `ENDP` directives. Each of these directives has a label that gives the name of the procedure. With *windows32* programs *_MainProc* is a procedure. Additional assembly language procedures can go in the same code segment before or after *_MainProc*. They can even be in separate files; information for how to do this is in the next section.

Let's first look at how to transfer control from *_MainProc* to the procedure *fctn1*. This is done by the instruction

```
call fctn1
```

In general, a `call` instruction saves the address of the next instruction (the one immediately following the call), then transfers control to the procedure code. It does this by pushing EIP onto the stack and then changing EIP to contain the address of the first instruction of the procedure.

Transferring control back from a procedure is accomplished by reversing the above steps. A `ret` (return) instruction pops the stack into EIP, so that the next instruction to be executed is the one at the address that was pushed on the stack by the call. There is almost always at least one `ret` instruction in a procedure and there can be more than one. If there is only one `ret`, it is ordinarily the last instruction in the procedure since subsequent instructions would be unreachable without "spaghetti code." Although a `call` instruction must identify its destination, the `ret` does not—control will transfer to the instruction following the most recent call. The address of that instruction is stored on the 80x86 stack.

The syntax of the 80x86 call statement is

```
call destination
```

Figure 6.8 lists some of the 80x86 `call` instructions. No `call` instruction modifies any flag. All of the procedure calls used in this text will be the first type, near relative. For a near relative call, the 5 bytes of the instruction consist of the E8 opcode plus the displacement from the next instruction to the first instruction of the procedure. The transfer of control when a procedure is called is similar to the transfer of a relative jump, except that the old contents of EIP are pushed.

Near indirect calls encode a register 32 or a reference to a doubleword in memory. When the call is executed, the contents of that register or doubleword are used as the address of the procedure. This makes it possible for a `call` instruction to go to different procedures different times.

All far calls must provide both new CS contents and new EIP contents. With far direct calls, both of these are coded in the instruction, and these 6 bytes plus the 1 for the opcode make the 7 seen in Figure 6.8. With far indirect calls, these are located at a 6-byte block in memory, and the address of that block is coded in the instruction. The extra byte is a *ModR/M* byte. Far calls were very important when the segmented memory model was used.

The return instruction `ret` is used to transfer control from a procedure body back to the calling point. Its basic operation is simple: it simply pops the address previously stored on the stack and loads it into the instruction pointer EIP. Since the stack contains the address of the instruction following the call, execution will continue at that point.

Operand	Opcode	Bytes of Object Code
near relative	E8	5
near indirect using register	FF	2
near indirect using memory	FF	2+
far direct	9A	7
far indirect	FF	6+

Figure 6.8

`call` instructions

A near return just has to restore EIP. A far return instruction reverses the steps of a far call, restoring both EIP and CS; both of these values are popped from the stack. No `ret` instruction changes any flag.

There are two formats for the `ret` instruction. The more common form has no operand, and is simply coded

```
ret
```

The other version has a single operand, and is coded

```
ret  count
```

The operand *count* is added to the contents of ESP after completion of the other steps of the return process (popping EIP and, for a far procedure, CS). This can be useful if other values (parameters in particular) have been saved on the stack just for the procedure call; this is not used with the *cdecl* protocol, however. Figure 6.9 lists the various formats of `ret` instructions.

Using a high-level language, a procedure definition often includes **parameters** (sometimes called *formal parameters*) that are associated with **arguments** (also called *actual parameters*) when the procedure is called. For the procedure's pass-by-value (in) parameters, values of the arguments (which may be expressions) are copied to the parameters when the procedure is called, and these values are then referenced in the procedure using their local names (the identifiers used to define the parameters). Reference (pass-by-location or in-out) parameters associate a parameter identifier with an argument that is a single variable, and can be used to pass a value either to the procedure from the caller or from the procedure back to the caller. Reference parameters are covered in the next section.

Our example code in Figure 6.7 has two arguments (*number1* and *number2*) in *_MainProc* that are passed by value to two parameters (*x* and *y*) in *fctn1*. We now look at how to pass parameter values to a procedure and results back from the procedure. The second part of this is simple—if the procedure returns a single doubleword value, then it puts that value in EAX to be used by the calling program. Notice that this is exactly what *fctn1* does in the program in Figure 6.7; after some preliminaries (explained next), it computes the desired value in EAX where it is available back in *_MainProc*. With the *cdecl* protocol, only the EAX register may be used for this purpose.

Doubleword parameters are passed to the procedure by pushing them on the stack. In the *cdecl* protocol, the parameters are pushed on the stack in the opposite order in which

Type	Operand	Opcode	Bytes of Object Code
near	none	C3	1
near	immediate	C2	3
far	none	CB	1
far	immediate	CA	3

Figure 6.9

`ret` instructions

they appear in the parameter list—the last parameter value is pushed first and the first parameter value is pushed last. The code that calls *fctn1* in *_MainProc* is

```
push  number2     ; 2nd parameter
push  number1     ; 1st parameter
call  fctn1       ; fctn1(number1, number2)
add   esp, 8      ; remove parameters from stack
```

The first two statements obviously push the argument values on the stack prior to the procedure call. The purpose of the last statement is to remove the values from the stack following return from the procedure. If the stack is not cleaned up and a program repeatedly calls a procedure, eventually the stack will fill up causing a run-time error with modern operating systems. Arguments could be removed using the alternative form of the `ret` statement that specifies an operand, but the *cdecl* protocol specifically leaves the stack cleanup task to the calling program. The arguments could be removed by popping the values off the stack, but it is more efficient to simply add the number of bytes of parameters to ESP, moving the stack pointer above the values.

Now, we look at how a procedure retrieves parameter values from the stack. Upon entry to the procedure, the stack looks like the left illustration in Figure 6.10. The two arguments—now the parameter values—have been pushed on the stack by the calling program and the return address has been pushed on the stack by the `call` instruction. The first instructions executed by the procedure are

```
push  ebp         ; save base pointer
mov   ebp, esp    ; establish stack frame
push  ebx         ; save EBX
```

This is known as **entry code**. The first two instructions will always be the pair shown. They preserve EBP so that it can be restored before returning, and set EBP to point at a fixed place in the stack that can be used to locate parameters. The third instruction is needed in this procedure so that EBX can be used for computations within the procedure and then restored before return; this makes its use in the procedure transparent to the calling program. After these three instructions are executed, the stack looks like the right illustration in Figure 6.10.

There are 8 bytes stored between the address stored in EBP and the first parameter (*x*) value. Parameter 1 can be referenced using based addressing by [ebp+8]. The second parameter (*y*) value is 4 bytes higher on the stack; its reference is [ebp+12]. The code

```
mov   eax, [ebp+8]   ; x
imul  eax, 3         ; 3*x
mov   ebx, [ebp+12]   ; y
imul  ebx, 7         ; 7*y
add   eax, ebx       ; 3*x + 7*y
```

copies the value of the first parameter from the stack into EAX and the value of the second parameter from the stack into EBX in order to compute the desired promised result.

You may wonder why EBP is used at all. Why not just use ESP as a base register? The principal reason is that ESP is likely to change, but the instruction `mov ebp,esp`

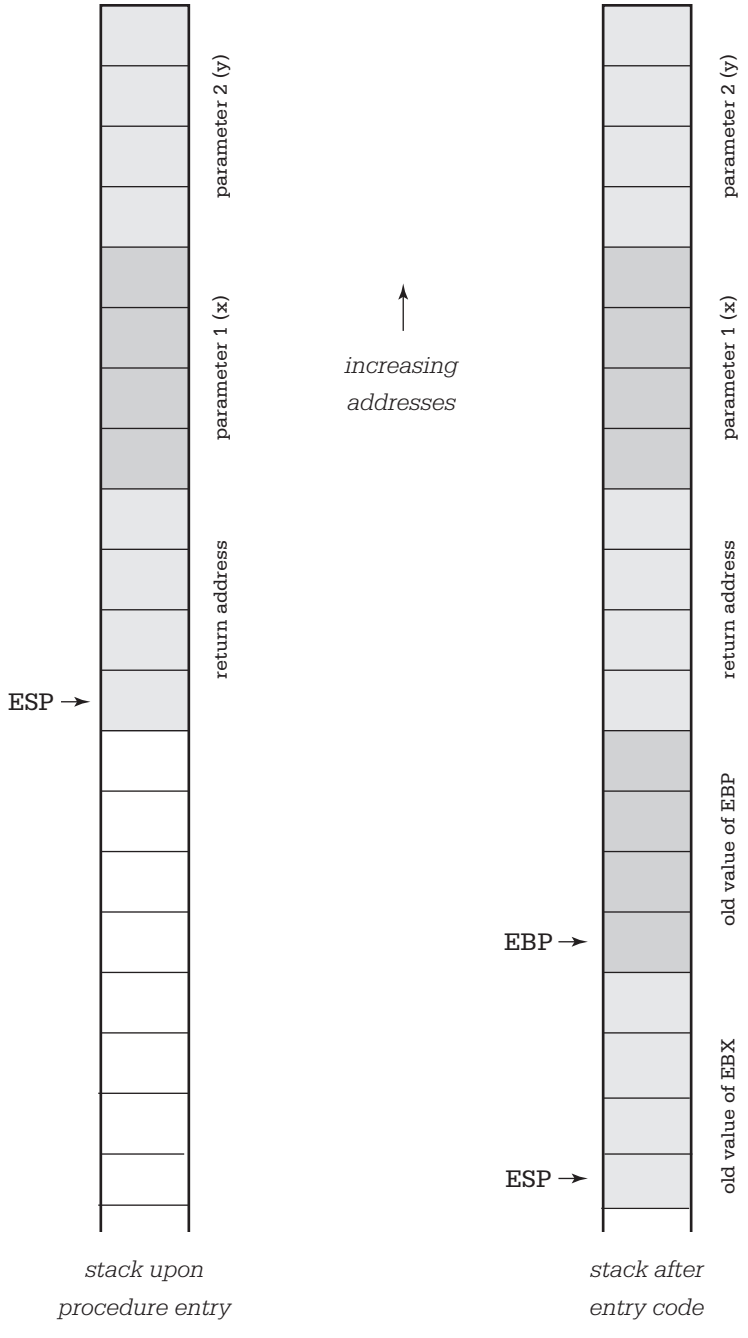


Figure 6.10

Establishing base pointer in procedure entry code

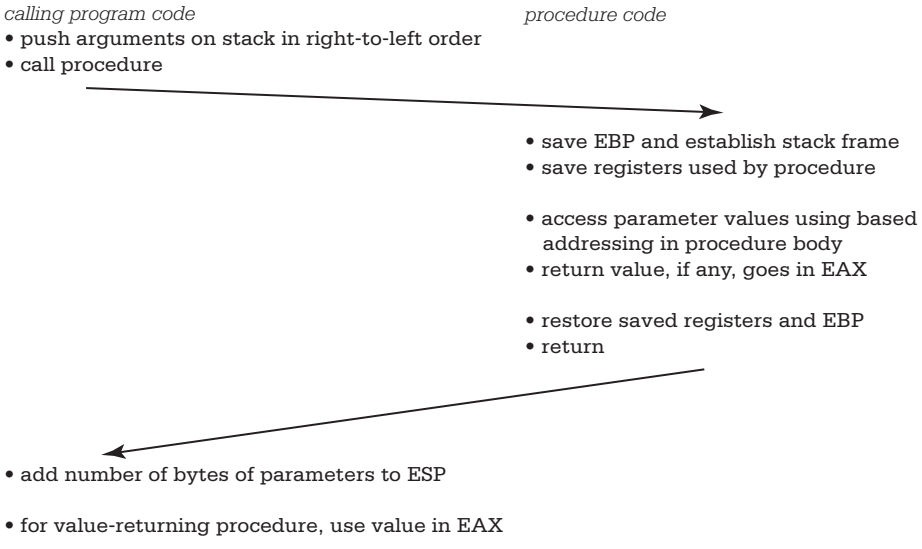


Figure 6.11
cdecl protocol

loads EBP with a fixed reference point in the stack. This fixed reference point will not change as other instructions in the procedure are executed, even if the stack is used for other purposes, for example, to push additional registers or to call other procedures.

We now come to the third major concept, how to write procedure code that is independent of and preserves the environment for the calling program. You have already seen most of the code for this. Basically, the entry code pushes each register that will be used by the procedure, and the **exit code** pops them in the opposite order. Obviously, you must not save and restore EAX when a value is being returned in EAX. The exit code for our example consists of

```
pop   ebx        ; restore EBX
pop   ebp        ; restore EBP
ret            ; return
```

EBP is always restored last since it is always saved first. This example only used EBX for computations, but it is not unusual to save and restore several registers. Figure 6.11 summarizes the *cdecl* protocol.

Exercises 6.2

*1. Suppose that the procedure *exercise1* is called by the instruction

```
call exercise1
```

If this call statement is at address 00402000 and ESP contains 00406000 before the call, what return address will be on the stack when the first instruction of procedure *exercise1* is executed? What will be the value in ESP?

2. Suppose that a procedure begins with this entry code

```
push ebp      ; save EBP
mov  ebp,esp   ; new base pointer
push ecx      ; save registers
push esi
...
```

Assume that this procedure has three doubleword parameters whose formal order is first *x*, then *y*, and last *z*. Draw a picture of the stack following execution of the above code. Include parameters, return address, and show the bytes to which EBP and ESP point. Give the based address with which each parameter can be referenced.

Programming Exercises 6.2

For each of these exercises follow the *cdecl* protocol for the specified procedure and write a short *console32* or *windows32* test-driver program to test the procedure.

1. Write a procedure *discr* that could be described in C/C++ by

```
int discr(int a, int b, int c)
// return the discriminant b*b-4*a*c
```

that is, its name is *discr*, it has three doubleword integer parameters, and it is a value-returning procedure.
2. Write a value-returning procedure *min2* to find the smaller of two doubleword integer parameters.
3. Write a value-returning procedure *max3* to find the largest of three doubleword integer parameters.
4. Programming Exercise 5.3.6 has an algorithm for finding the greatest common divisor of two positive integers. Write a procedure *gcd* to implement this algorithm. It might be described in C/C++ by `int gcd(int number1, int number2)`, that is, its name is *gcd*, it has two doubleword integer parameters, and it is a value-returning procedure.
5. The volume of a pyramid with a rectangular base is given by the formula *h*x*y*/3 where *h* is the height of the pyramid, *x* is the length, and *y* is the width of the base. Write a procedure *pVolume* that implements the function described by the following C/C++ function header:

```
int pVolume(int height, int length, int width);
// return volume of pyramid with rectangular base
```

6.3 Additional 32-Bit Procedure Options

The previous section's main example showed how to pass arguments to parameters by value. With a reference parameter, the address of the argument instead of its value is passed to the procedure. Reference parameters are used for several purposes, two of which are to send a large argument (for example, an array or a structure) to a procedure, or to send results back to the calling program as argument values. This section begins with an example that illustrates both of these uses of reference parameters.

Consider the procedure for which the C++ function prototype could be written

```
void minMax(int arr[], int count, int& min, int& max);
// Set min to smallest value in arr[0],..., arr[count-1]
// Set max to largest value in arr[0],..., arr[count-1]
```

In C++, the notation `int arr[]` indicates that the address of the integer array *arr* will be passed, and `int&` instead of `int` says that the addresses of integer variables *min* and *max* will be passed. Figure 6.12 shows an implementation of this procedure in a *console32* program. It also includes a simple test driver that establishes locations for an array and the smallest and largest numbers to be stored, and calls *minMax*, pushing the four parameters, three of which are addresses. Note that there are 16 bytes of parameters to remove after the call.

```
; procedure minMax to find smallest and largest elements in an
; array and test driver for minMax
; author: R. Detmer
; date: 6/2013

.586
.MODEL FLAT
.STACK 4096

.DATA
minimum   DWORD  ?
maximum   DWORD  ?
nbrArray  DWORD  25, 47, 95, 50, 16, 95 DUP (?)

.CODE
main  PROC
    lea  eax, maximum ; 4th parameter
    push eax
    lea  eax, minimum ; 3rd parameter
    push eax
    pushd 5       ; 2nd parameter (number of elements)
    lea  eax, nbrArray ; 1st parameter
    push eax
    call minMax    ; minMax(nbrArray, 5, minimum, maximum)
    add  esp, 16    ; remove parameters from stack
quit:  mov  eax, 0    ; exit with return code 0
    ret
main  ENDP
```

Figure 6.12

Procedure using address parameters (*continues*)

```
; void minMax(int arr[], int count, int& min, int& max);
; Set min to smallest value in arr[0],..., arr[count-1]
; Set max to largest value in arr[0],..., arr[count-1]
minMax PROC
    push ebp      ; save base pointer
    mov  ebp,esp    ; establish stack frame
    push eax      ; save registers
    push ebx
    push ecx
    push edx
    push esi
    mov  esi,[ebp+8]  ; get address of array arr
    mov  ecx,[ebp+12] ; get value of count
    mov  ebx, [ebp+16] ; get address of min
    mov  edx, [ebp+20] ; get address of max

    mov  DWORD PTR [ebx], 7fffffffh ; largest possible integer
    mov  DWORD PTR [edx], 80000000h ; smallest possible integer
    jecxz exitCode   ; exit if there are no elements

forLoop:
    mov  eax, [esi]  ; a[i]
    cmp  eax, [ebx]  ; a[i] < min?
    jnl  endIfSmaller ; skip if not
    mov  [ebx], eax  ; min := a[i]
endIfSmaller:
    cmp  eax, [edx]  ; a[i] > max?
    jng  endIfLarger  ; skip if not
    mov  [edx], eax  ; max := a[i]
endIfLarger:
    add  esi, 4    ; point at next array element
    loop forLoop    ; repeat for each element of array

exitCode:
    pop  esi     ; restore registers
    pop  edx
    pop  ecx
    pop  ebx
    pop  eax
    pop  ebp
    ret         ; return
minMax ENDP
END
```

Figure 6.12

Procedure using address parameters (*continued*)

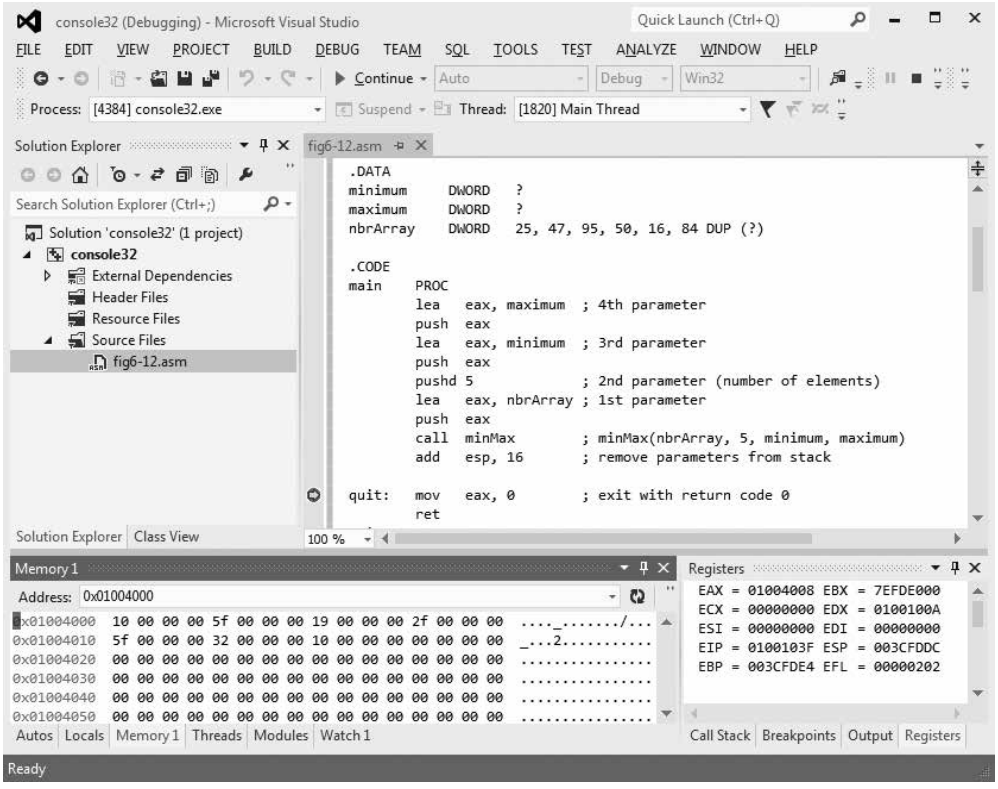


Figure 6.13

Procedure using address parameters

The *minMax* procedure follows a straightforward design that is in the comments of the procedure. Notice that several registers are used and their contents are saved in the entry code and restored in the exit code. The reader should draw the stack picture to see where the parameters are placed on the stack. Immediately after the entry code, the various parameters are copied into registers. The *minMax* procedure uses indirect addressing extensively, based addressing to retrieve the parameters, and register indirect addressing to access the array sequentially. Register indirect addressing is also used as EBX and EDX point at *min* and *max*, in this case the doublewords allocated for *minimum* and *maximum*, respectively, in the test driver. As an alternative to starting *min* at the largest possible integer and *max* at the smallest possible integer, each could have been initialized to the first array element's value. This takes slightly more code. Figure 6.13 shows a debugger window with the program paused following the call; the memory window has been set to start at the address of *minimum*.

Procedure *minMax* required the use of several registers in its implementation. Using registers is almost always preferable to using memory but there simply aren't enough of them to implement complex algorithms. Some procedures need to have local variables in memory. The standard way to do this is to allocate space for them on the stack.

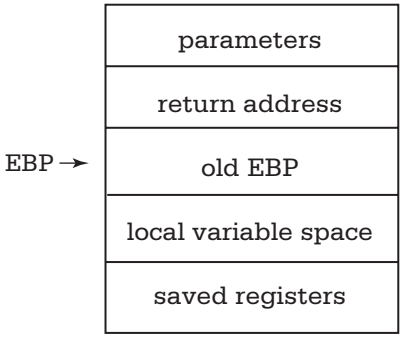
- save EBP and establish stack frame
- **subtract number of bytes of local space from ESP**
- save registers used by procedure
- access parameter values using based addressing in procedure body
- return value, if any, goes in EAX
- restore saved registers
- **copy EBP to ESP**
- restore EBP
- return

Figure 6.14

Procedure code with local variable space

Figure 6.14 outlines procedure code to do this. It is a minor modification of the right side of Figure 6.11 with the new steps shown in bold.

Here is a simplified, not-to-scale picture of the stack with local storage reserved.



Just as doubleword parameters above the reference point can be referenced by `[ebp+8]`, `[ebp+12]`, and so on, the first doubleword in the local variable space below the reference point can be accessed by `[ebp-4]`, the next below by `[ebp-8]`, and so on. C and C++ compilers normally allocate all local variables on the stack. There are several difficulties with doing this in assembly language, not the least of which is remembering where a particular local variable is stored in the stack.

The two new steps are obviously implemented by `sub esp,`*n* and `mov esp,ebp`, where *n* is the number of bytes of local storage you want to reserve. You may be wondering why the "deallocation" step isn't `add esp,`*n*. The answer is that it could be, but the `mov` instruction is both simpler and safer. It is safer because it will still restore the correct value to ESP even if the saved registers were not properly popped off the stack.

A **recursive** procedure or function is one that calls itself, either directly or indirectly. The best algorithms for manipulating many data structures are recursive. It is frequently very difficult to code certain algorithms in a programming language that does not support recursion.

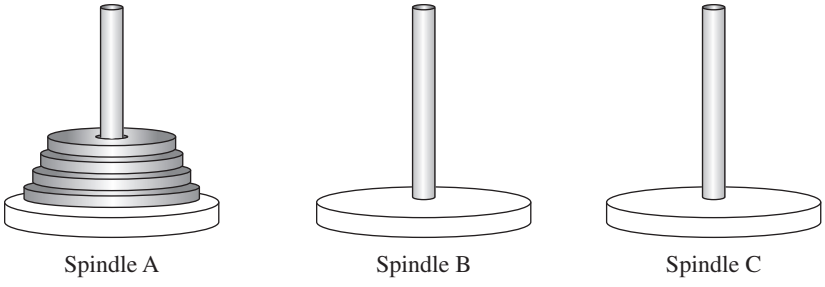


Figure 6.15
Towers of Hanoi puzzle

It is almost as easy to code a recursive procedure in 80x86 assembly language as it is to code a non-recursive procedure. If parameters are passed on the stack and local variables are stored on the stack, then each call of the procedure gets new storage allocated for its parameters and local variables. There is no danger of the arguments passed to one call of a procedure being confused with those for another call because each call has its own stack frame. If registers are properly saved and restored, then the same registers can be used by each call of the procedure.

This section gives one example of a recursive procedure in 80x86 assembly language. It solves the Towers of Hanoi puzzle, pictured in Figure 6.15 with four disks. The object of the puzzle is to move all disks from source spindle A to destination spindle B, one at a time, never placing a larger disk on top of a smaller disk. Disks can be moved to spindle C, a spare spindle. For instance, if there are only two disks, the small disk can be moved from spindle A to C, the large one can be moved from A to B, and finally the small one can be moved from C to B.

In general, the Towers of Hanoi puzzle is solved by looking at two cases. If there is only one disk, then the single disk is simply moved from the source spindle to the destination. If the number of disks *nbrDisks* is greater than one, then the top (*nbrDisks*-1) disks are moved to the spare spindle, the largest one is moved to the destination, and finally the (*nbrDisks*-1) smaller disks are moved from the spare spindle to the destination. Each time (*nbrDisks*-1) disks are moved, exactly the same procedure is followed, except that different spindles have the roles of source, destination, and spare. Figure 6.16 expresses the algorithm in pseudocode.

Figure 6.17 lists 80x86 code that implements the design as a *windows32* program. The code is a fairly straightforward translation of the pseudocode design, with each recursive procedure call in *move* implemented just like the call in the main program. Note that although the spindles are designated by single characters, these characters are passed in doublewords to ensure that the stack stays on a doubleword boundary. A high-level language compiler would probably calculate *nbrDisks-1* twice—once for each recursive call where it is used—but we can be a little more efficient and calculate it just one time. This value is computed in EAX and will be there after the intervening code because subsequent calls save and restore EAX. The last thing to note is that variables in the data section are used by procedure *move*. In general, use of global variables is discouraged, but here it is simpler and more efficient than allocating local variables on the stack. They are only being used for display of a single instruction and do not need to be preserved between calls. Figure 6.18 shows a sample run of this program.

```
procedure move(nbrDisks, source, destination, spare);
begin
  if NbrDisks = 1
  then
    display "Move disk from ", source, " to ", destination
  else
    move(nbrDisks-1, source, spare, destination);
    move(1, source, destination, spare);
    move(nbrDisks-1, spare, destination, source);
  end if;
end procedure move;

begin {main program}
  prompt for and input number;
  move(number, 'A', 'B', 'C');
end;
```

Figure 6.16

Pseudocode for Towers of Hanoi Solution

```
; program to display instructions for "Towers of Hanoi" puzzle
; author: R. Detmer
; revised: 6/2013
.586
.MODEL FLAT
INCLUDE io.h
.STACK 4096

.DATA
prompt BYTE  "How many disks?",0
number BYTE  16 DUP (?)
outLbl BYTE  "Move disk", 0
outMsg BYTE  "from spindle "
source BYTE  ?, 0ah, 0dh
       BYTE  'to spindle '
dest   BYTE  ?, 0

.CODE
_MainProc PROC
    mov  al,'C'    ; argument 4: 'C'
    push  eax
    mov  al,'B'    ; argument 3: 'B'
    push  eax
    mov  al,'A'    ; argument 2: 'A'
```

Figure 6.17

Towers of Hanoi solution (*continues*)

```
    push  eax
    input prompt, number,16  ; read ASCII characters
    atod  number     ; convert to integer
    push  eax      ; argument 1: number
    call  move      ; Move(number,Source,Dest,Spare)
    add  esp,16     ; remove parameters from stack

    mov  eax, 0 ; exit with return code 0
    ret
_MainProc ENDP

move  PROC
; procedure move(nbrDisks : integer; { number of disks to move }
;        source, dest, spare : character { spindles to use } )
; all parameters are passed in doublewords on the stack

    push  ebp        ; save base pointer
    mov  ebp,esp     ; establish stack frame
    push  eax        ; save registers
    push  ebx

    cmp  DWORD PTR [ebp+8],1 ; nbrDisks = 1?
    jne  elseMore     ; skip if more than 1
    mov  ebx,[ebp+12]  ; source
    mov  source,bl     ; copy character to output
    mov  ebx,[ebp+16]  ; destination
    mov  dest,bl      ; copy character to output
    output outLbl, outMsg ; display move instruction
    jmp  endIfOne     ; return
elseMore:
    mov  eax,[ebp+8]    ; get nbrDisks
    dec  eax        ; nbrDisks – 1
    push  DWORD PTR [ebp+16] ; par 4: old destination is new spare
    push  DWORD PTR [ebp+20] ; par 3: old spare is new destination
    push  DWORD PTR [ebp+12] ; par 2: source does not change
    push  eax        ; par 1: nbrDisks-1
    call  move       ; move(nbrDisks-1,source,spare,destination)
    add  esp,16      ; remove parameters from stack

    push  DWORD PTR [ebp+20] ; par 4: spare unchanged
    push  DWORD PTR [ebp+16] ; par 3: destination unchanged
    push  DWORD PTR [ebp+12] ; par 2: source does not change
    pushd 1          ; par 1: 1
    call  move       ; move(1,source,destination,spare)
    add  esp,16      ; remove parameters from stack
    push  DWORD PTR [ebp+12] ; par 4: original source is spare
    push  DWORD PTR [ebp+16] ; par 3: original destination
```

Figure 6.17

Towers of Hanoi solution (*continues*)

```
    push  DWORD PTR [ebp+20] ; par 2: source is original spare
    push  eax        ; parameter 1: nbrDisks-1
    call  move       ; move(nbrDisks-1,spare,destination,source)
    add   esp,16     ; remove parameters from stack
endIfOne:
    pop   ebx        ; restore registers
    pop   eax
    pop   ebp        ; restore base pointer
    ret            ; return
move  ENDP
END
```

Figure 6.17

Towers of Hanoi solution (*continued*)

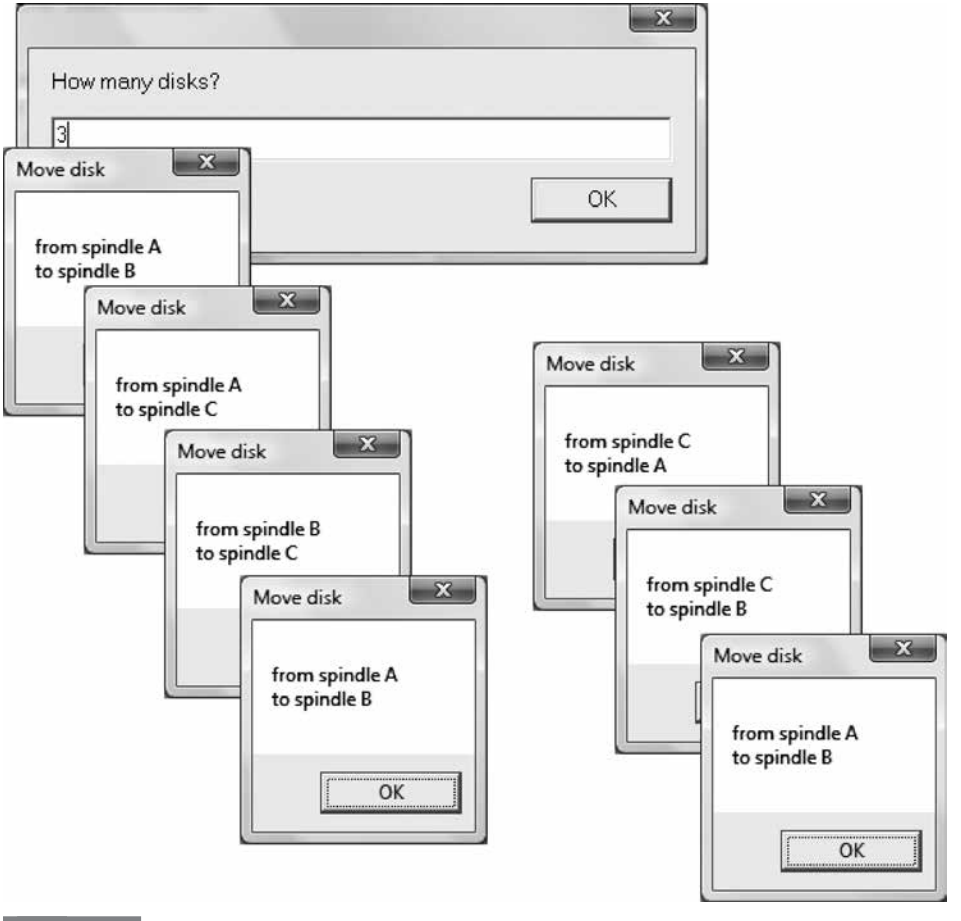


Figure 6.18

Towers of Hanoi sample run

One of the reasons for using procedures is so that code that performs a useful task can be reused in other programs. Although you can always just copy and paste code from one program to another, it is often more convenient to package a procedure in a separate file, and then simply include the file in another project. We now return to the first example of this section and show how it can be split into separate files. The test-driver code is shown in Figure 6.19 and the procedure code is in Figure 6.20. There is little new here except that the test-driver file needs an `EXTERN` directive to identify *minMax* as a procedure. The procedure file must repeat directives that are also used in the test-driver file: `.586`, `.MODEL FLAT`, `.CODE`, and `END`. It is not necessary to have another `.STACK` directive, and in a *windows32* program `INCLUDE io.h` is only needed if the procedure is using the macros defined in *io.h*. In the Visual Studio environment the two *.asm* files will be separate source files.

```
; and test driver for minMax
; author: R. Detmer
; date: 6/2013

.586
.MODEL FLAT
.STACK 4096

.DATA
minimum   DWORD  ?
maximum   DWORD  ?
nbrArray  DWORD  25, 47, 95, 50, 16, 84 DUP (?)

EXTERN minMax:PROC

.CODE
main  PROC
    lea  eax, maximum ; 4th parameter
    push eax
    lea  eax, minimum ; 3rd parameter
    push eax
    pushd 5       ; 2nd parameter (number of elements)
    lea  eax, nbrArray ; 1st parameter
    push eax
    call minMax    ; minMax(nbrArray, 5, minimum, maximum)
    add  esp, 16    ; remove parameters from stack

quit:  mov  eax, 0    ; exit with return code 0
    ret
main  ENDP
END
```

Figure 6.19

Test driver for *minMax* in separate file

```
; procedure minMax to find smallest and largest elements in an array
; author: R. Detmer   date: 6/2013
.586
.MODEL FLAT
.CODE
; void minMax(int arr[], int count, int& min, int& max);
; Set min to smallest value in arr[0],..., arr[count-1]
; Set max to largest value in arr[0],..., arr[count-1]
minMax PROC
    push ebp      ; save base pointer
    mov  ebp,esp    ; establish stack frame
    push eax      ; save registers
    push ebx
    push ecx
    push edx
    push esi

    mov  esi,[ebp+8]  ; get address of array arr
    mov  ecx,[ebp+12] ; get value of count
    mov  ebx, [ebp+16] ; get address of min
    mov  edx, [ebp+20] ; get address of max

    mov  DWORD PTR [ebx], 7fffffffh ; largest possible integer
    mov  DWORD PTR [edx], 80000000h ; smallest possible integer
    jecxz exitCode   ; exit if there are no elements

forLoop:
    mov  eax, [esi]  ; a[i]
    cmp  eax, [ebx]  ; a[i] < min?
    jnl  endIfSmaller ; skip if not
    mov  [ebx], eax  ; min := a[i]
endIfSmaller:
    cmp  eax, [edx]  ; a[i] > max?
    jng  endIfLarger  ; skip if not
    mov  [edx], eax  ; max := a[i]
endIfLarger:
    add  esi, 4    ; point at next array element
    loop forLoop    ; repeat for each element of array

exitCode:
    pop  esi     ; restore registers
    pop  edx
    pop  ecx
    pop  ebx
    pop  eax
    pop  ebp
    ret         ; return
minMax ENDP
END
```

Figure 6.20

minMax in separate file

How can you call a high-level language procedure from assembly language or an assembly language procedure from a high-level language? The answer is by carefully following the calling protocol used by the compiler for the high-level language. The Visual Studio C compiler uses the *cdecl* protocol. The *windows32* projects that you have been using for programs with input and output already do this. For example, the file *framework.c* contains the code

```
int MainProc(void);
// prototype for user's main program

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance,
  LPSTR lpCmdLine, int nCmdShow)
{
  _hInstance = hInstance;
  return MainProc();
}
```

Execution begins with *WinMain* that basically just calls your assembly language procedure *MainProc*. However, recall that the name of your procedure is not *MainProc*, but *_MainProc*. The code generated by the C compiler follows the *cdecl* **text decoration** convention of appending a leading underscore. In general, to call an assembly language procedure from a Visual Studio C program, prototype the function to describe it, name the assembly language procedure with the same name prefixed with an underscore, and follow the *cdecl* protocols in the assembly language code.

A *windows32* project also calls C functions in *framework.c* from expansions of the *output* and *input* macros. For example, the definition code for the *output* macro contains

```
lea  eax,outStr     ; string address
push  eax          ; string parameter on stack
lea  eax,outLbl     ; label address
push  eax          ; string parameter on stack
call  _showOutput     ; showOutput(outLbl, outStr)
add  esp, 8        ; remove parameters
```

This is clearly a call to procedure *_showOutput*, which is *showOutput* in *framework.c*. The assembly language code must add the underscore to the name because the assembler does not decorate the name, but the C compiler will. In general, text decoration is only a concern when you are mixing high-level and assembly language procedures, not when you are entirely writing in assembly language where no text decoration is generated or in C where the compiler takes care of text decoration automatically.

The Visual Studio programming environment uses several other procedure protocols, one of which is **stdcall**. The *stdcall* protocol is similar to *cdecl*, the biggest differences being that the procedure rather than the caller must remove parameters from

the stack (which makes the `ret` instruction with an operand very handy!) and the text decoration convention is much more complex, involving not only a leading underscore but a trailing at sign (@) followed by a decimal number that is the number of bytes of parameters. The **fastcall** protocol gives yet another set of conventions. With *fastcall*, parameters are passed in registers. The important point here is that when you are mixing assembly language and a high-level language, you must know what protocol the high-level language compiler is using and follow it carefully.

Exercises 6.3

*1. Give entry code and exit code for a procedure that reserves 8 bytes of storage on the stack for local variables. Assuming that this space is used for two doublewords, give the based address with which each local variable can be referenced.
2. Figure 6.11 gave the steps for calling code and procedure code using the *cdecl* protocol. Write down the corresponding lists for the *stdcall* protocol.

Programming Exercises 6.3

For each of these exercises follow the *cdecl* protocol for the specified procedure and write a short *console32* or *windows32* test-driver program to test the assembly language procedure.

1. Suppose that a procedure is described in C/C++ by `void toUpper(char str[])`, that is, its name is *toUpper*, and it has a single parameter that is the address of an array of characters. Assuming that the character string is null-terminated, implement *toUpper* so that it changes each lowercase letter in the string to its uppercase equivalent, leaving all other characters unchanged.
2. Suppose that a procedure is described in C/C++ by `int upperCount(char str[])`, that is, its name is *upperCount*, it has a single parameter that is the address of an array of characters, and it returns an integer. Assuming that the character string is null-terminated, implement *upperCount* so that it returns a count of how many uppercase letters appear in the string.
3. Programming Exercise 5.5.5 gave the selection sort algorithm. Implement this algorithm in a procedure whose C/C++ description could be

```
void selectionSort(int nbrArray[], int nbrElts)
; sort nbrArray[0] .. nbrArray[nbrElts-1]
; into increasing order using selection sort
```

   The first parameter will be the address of the array.
4. Write a procedure *avg* to find the average of a collection of doubleword integers in an array. Procedure *avg* will have three parameters in the following order:
   (1) The address of the array.
   (2) The number of integers in the array (passed as a doubleword).
   (3) The address of a doubleword at which to store the result.
5. Write a value-returning procedure *search* to search an array of doublewords for a specified doubleword value. Procedure *search* will have three parameters:
   (1) The value for which to search (a doubleword integer).
   (2) The address of the array.
   (3) The number *n* of doublewords in the array (passed as a doubleword).
   Return the position (1,2, . . . ,*n*) at which the value is found, or return 0 if the value does not appear in the array.

6. The factorial function is defined for a nonnegative integer argument *n* by

$$factorial\,(n) = \begin{cases} 1 \text{ if } n = 0 \\ n \times factorial\ (n-1)\ \textit{if}\ n > 0 \end{cases}$$

   Write a value-returning procedure named *factorial* that implements this recursive definition.

7. The greatest common divisor (GCD) of two positive integers *m* and *n* can be calculated recursively by the function described below in pseudocode.

```
function gcd(m, n : integer) : integer;
if n = 0
then
  return m;
else
  remainder := m mod n;
  return gcd(n, remainder);
end if;
```

   Write a value-returning procedure named *gcd* that implements this recursive definition.

8. Write a procedure *arrMax* that implements the function described by the following C/C++ function header:

```
int arrMax(int arr[], int nbrElts);
// if nbrElts <= 0 returns -99999
// otherwise returns maximum of the first nbrElts
// elements of arr
```

   The first parameter of *arrMax* is the address of the first doubleword of the array *arr*.

9. Write a procedure *hasLower* that implements the function described by the following C/C++ function header:

```
int hasLower(char str[]);
// precondition: str is a null terminated string
// postcondition: returns true (-1) if str contains at least
//   one lowercase letter; otherwise returns false (0)
```

   The parameter of *hasLower* is the address of the first byte of the string *str*.

10. Programming Exercise 5.5.7 gave an algorithm for merging two sorted integer arrays into a third sorted array. Write a procedure *arrayMerge* that implements that design in a procedure described by the following C/C++ function header:

```
void arrayMerge(int array1[], int count1,
                int array2[], int count2,
                int array3[]);
// precondition:  array1 and array2 are sorted arrays with
//                count1 and count2 elements, respectively.
//                Each has at least one unused slot.
//                array3 has at least count1+count2 slots
// postcondition: array1 and array2 have been merged into
//                the sorted array3
```

   Parameters 1, 3, and 5 are the addresses of the three arrays.

11. The inner product of two vectors is the sum of the products of corresponding terms. For instance, given [3, 6, 5] and [2, −4, 1] (each stored as an array with 3 elements), the inner product is 3*2 + 6*(−4) + 5*1 = −13. Write a procedure *innerProduct* that implements the function described by the following C/C++ function header:

```
int innerProduct(int vector1[], int vector2[], int vLength);
// returns the inner product of vector1 and vector2,
// each with vLength components
```

The first two parameters are the addresses of the two arrays.

6.4 64-Bit Procedures

This section describes the differences in the 32-bit procedure protocol and the 64-bit procedure protocol. First we look at the additional `push` and `pop` instructions available in 64-bit mode. These are shown in Figures 6.21 and 6.22. The entries in these tables are very similar to those for 32-bit mode instructions (see Figures 6.1 and 6.5 in this chapter's first section). There are entries for all 16 64-bit general registers. The REX prefix 41 is used for R8–R15.

One important difference in 32- and 64-bit modes is that you cannot use 32-bit register or memory operands with 64-bit `push` and `pop` instructions. The available immediate operand sizes for `push` remain byte, word, and doubleword—quadword is not added. Also, the `pushad` and `popad` instructions in 32-bit mode do not exist in 64-bit mode, nor are there instructions to push and pop all 16 64-bit registers.

Operand	Opcode	Bytes of Object Code
64-bit mode		
RAX or R8	50	1
RCX or R9	51	1
RDX or R10	52	1
RBX or R11	53	1
RSP or R12	54	1
RBP or R13	55	1
RSI or R14	56	1
RDI or R15	57	1
memory word	FF	2+
memory quadword	FF	2+
immediate byte	6A	2
immediate word	68	3
immediate doubleword	68	5

Figure 6.21

64-bit mode `push` instructions

Operand	Opcode	Bytes of Object Code
64-bit mode		
RAX or R8	58	1
RCX or R9	59	1
RDX or R10	5A	1
RBX or R11	5B	1
RSP or R12	5C	1
RBP or R13	5D	1
RSI or R14	5E	1
RDI or R15	5F	1
memory word	8F	2+
memory quadword	8F	2+

Figure 6.22
64-bit mode `pop` instructions

Just as it is important to keep the stack on a doubleword boundary in a 32-bit environment, it is important to keep it on a quadword boundary in a 64-bit environment. Therefore, you almost always push and pop quadwords.

The 64-bit versions of instructions `call` and `ret` are very similar to the 32-bit versions. The tables for `call` and `ret` instructions are not repeated since they are exactly the same tables as in 32-bit instructions mode shown in Figures 6.8 and 6.9. A `push` instruction pushes a 64-bit return address onto the stack before loading RIP with the procedure's address, and a `pop` instruction pops the 64-bit return address from the stack into RIP.

Where the 64-bit protocol is most different is in parameter conventions. With the 32-bit architecture, registers are often a scarce resource. The 64-bit architecture doubles the number of available registers, making it more practical to pass arguments in registers, and the 64-bit protocol takes advantage of this. Arguments that can be passed as quadwords (including bytes, words, and doublewords) are extended to quadword length, if necessary. The first four arguments are always passed in the registers shown in Figure 6.23. Additional arguments, if any, are passed on the stack.

In a 64-bit environment, a calling procedure must reserve space on the stack for arguments. Normally, the procedure does this in entry code. The *windows64* programs in this text start with

```
sub     rsp, 120   ; reserve stack space for MainProc
```

that generates enough space for 15 quadwords. The bottom part of the reserved space is reserved for arguments. If there is a fifth argument, then it is copied to [RSP+32], a sixth to [RSP+40], and so on. After the return address (8 bytes) is pushed on the stack, the called procedure will then find these values at [RSP+40], [RSP+48], and so on. Why start

Argument	Register
1	RCX
2	RDX
3	R8
4	R9

Figure 6.23
64-bit registers used to pass arguments

32 bytes from the bottom? This is to leave space in the stack for the first four parameters, even though they are in the registers. The called procedure can use this space to copy any of the first four argument values.

Registers can be pushed by entry code and popped by exit code similar to the way they are done in the 32-bit environment. However, this is usually done before the local stack space is reserved. Once the local stack space is established, there should be no change to RSP before a subsequent procedure is called. This makes it possible to use RSP and based addressing to locate parameters and local variables. However, you can use RBP as a frame pointer if needed.

RAX is used to return a single quadword value. Microsoft documentation labels registers RAX, RCX, RDX, and R8–R11 as **volatile**, meaning that the called procedure is free to change them. Similarly, RBX, RDI, RSI, RBP, RSP, and R12–R15 are called **nonvolatile**, meaning that a called procedure has the responsibility of preserving them. In practice, sometimes it is safest to preserve any register that you don't want destroyed by a called procedure. For example, in *windows64* projects, the *atod* macro includes the code

```
mov    [rsp+32], rcx  ; save register used to pass parameters
mov    [rsp+40], rbx  ; save registers destroyed by atodproc
mov    [rsp+48], rdx
lea    rcx,source     ; source address to rcx
call   atodproc       ; call atodproc(source)
mov    rcx, [rsp+32]  ; restore register used to pass parameters
mov    rbx, [rsp+40]  ; restore registers destroyed by atodproc
mov    rdx, [rsp+48]
```

Notice that the registers are saved in the stack area above the area reserved for copying the first four parameters. This code preserves RBX even though it should be nonvolatile.

Figure 6.24 shows the listing of a *console64* version of the program whose *console32* version appeared in Figure 6.12. It is noticeably simpler than the 32-bit version. The four arguments are simply placed in registers and then used in the procedure. The procedure *minMax* itself does not call additional procedures, so it does not need to establish local

stack space. One difference is that the 32-bit *minMax* simply places the largest and smallest possible values in the caller's data at the addresses passed in the third and fourth parameters, respectively, but since there is no immediate quadword to memory `mov` in the 64-bit architecture, the immediate values are first placed in RAX and then copied to their destinations.

```
; procedure minMax to find smallest and largest elements in an
; array and test driver for minMax   —   64-bit version
; author: R. Detmer
; date: 6/2013

.DATA
minimum   QWORD  ?
maximum   QWORD  ?
nbrArray  QWORD  25, 47, 95, 50, 16, 95 DUP (?)

.CODE
main    PROC
        sub  rsp, 32       ; local stack space
        lea  rcx, nbrArray ; 1st parameter
        mov  rdx, 5        ; 2nd parameter (number of elements)
        lea  r8, minimum   ; 3rd parameter
        lea  r9, maximum   ; 4th parameter
        call minMax    ; minMax(nbrArray, 5, minimum, maximum)

quit:   add  rsp, 32    ; clean up stack
        mov  rax, 0    ; exit with return code 0
        ret
main    ENDP

; void minMax(int arr[], int count, int& min, int& max);
; Set min to smallest value in arr[0],..., arr[count-1]
; Set max to largest value in arr[0],..., arr[count-1]
minMax PROC
    push rax      ; save registers
    push rsi
    mov  rsi,rcx  ; get address of array arr (1st parameter)
    mov  rcx,rdx  ; get value of count (2nd parameter)

    mov  rax, 7fffffffffffffffh ; largest possible integer
    mov  QWORD PTR [r8], rax
```

Figure 6.24

64-bit procedure using address parameters (*continues*)

```
    mov  rax, 8000000000000000h ; smallest possible integer
    mov  QWORD PTR [r9], rax
    jrcxz exitCode    ; exit if there are no elements

forLoop:
    mov  rax, [rsi]   ; a[i]
    cmp  rax, [r8]    ; a[i] < min?
    jnl  endIfSmaller ; skip if not
    mov  [r8], rax    ; min := a[i]
endIfSmaller:
    cmp  rax, [r9]    ; a[i] > max?
    jng  endIfLarger  ; skip if not
    mov  [r9], rax    ; max := a[i]
endIfLarger:
    add  rsi, 8    ; point at next array element
    loop forLoop   ; repeat for each element of array

exitCode:
    pop  rsi     ; restore registers
    pop  rax
    ret          ; return
minMax ENDP
END
```

Figure 6.24

64-bit procedure using address parameters (*continued*)

We conclude this section with an example of an assembly language procedure called from a C main program. The C test driver is shown in Figure 6.25 and the assembly language procedure is shown in Figure 6.26. These are separate source files in a *console64* project. One of the satisfying things about this program is that if you launch it with control-F5, you can actually see the output in the console window!

You may wonder why *add5* uses EAX instead of RAX to accumulate the sum. With the Visual Studio 2012 C compiler, an `int` is a 32-bit integer. The C compiler passes the five arguments in quadwords, but the high-order half of each quadword is undefined, so *add5* just adds the low-order doublewords that contain the integers. This C compiler uses `long long` to designate a quadword integer; `long int` is still 32 bits.

Another point of this example is to show how a fifth argument is handled in a procedure—in this case it is located at [RSP+40] since no registers needed to be saved in *add5*. Finally, note that the C compiler did not use text decoration in the 64-bit environment so that the called procedure could be named simply *add5*.

```
/* C test driver for assembly language procedure add5 */
/* author: R. Detmer */
/* date: 6/2013 */

int add5(int x1, int x2, int x3, int x4, int x5);
/* returns sum of arguments */

#include <stdio.h>

int main()
{
    int a=5;
    int b=7;
    int c=9;

    int sum;

    sum = add5(a, 6, b, 8, c);
    printf("The sum is %d\n", sum);
    return 0;
}
```

Figure 6.25

C test driver for 64-bit procedure

```
; procedure add5 to add five parameters
; 64-bit version
; author: R. Detmer
; date: 6/2013

.CODE

; void add5(int x1, int x2, int x3, int x4, int x5);
; returns sum of arguments
add5    PROC
        mov  eax, ecx   ; x1
        add  eax, edx   ; x2
        add  eax, r8d   ; x3
        add  eax, r9d   ; x4
        add  eax, DWORD PTR [rsp+40]    ; x5
        ret             ; return
add5    ENDP
END
```

Figure 6.26

64-bit procedure to add five integers

Exercises 6.4

1. Suppose that the entry code for a 64-bit procedure saves no register and reserves no local stack space. How do you find each of the following quadword parameter values in the body of the procedure?
   (a) parameter 1 *(b) parameter 3
   (c) parameter 5 *(d) parameter 7
2. Suppose that the entry code for a 64-bit procedure is
   ```
   push  rsi
   push  r12
   ```
   How do you find each of the following quadword parameter values in the body of the procedure?
   (a) parameter 1 *(b) parameter 3
   (c) parameter 5 *(d) parameter 7
3. Suppose that the entry code for a 64-bit procedure is
   ```
   push  rsi
   push  r12
   sub   rsp, 48
   ```
   How do you find each of the following quadword parameter values in the body of the procedure?
   (a) parameter 1 *(b) parameter 3
   (c) parameter 5 *(d) parameter 7

Programming Exercises 6.4

For each of these exercises follow the 64-bit protocol for the specified procedure. Embed the procedure and a test-driver program in a *console64* project. The test-driver program may be written either in assembly language or C.

1. Write a value-returning procedure *min2* to find the smaller of two quadword integer parameters.
2. Write a value-returning procedure *max6* to find the largest of six quadword integer parameters.
3. Suppose that a value-returning procedure is described in C/C++ by `int alphaCount(char str[])`, that is, its name is *alphaCount*, it has a single parameter that is the address of an array of characters, and it returns a doubleword integer. Assuming that the character string is null-terminated, implement *alphaCount* so that it returns a count of how many letters (lowercase or uppercase) appear in the string.
4. Programming Exercise 5.5.5 gave the selection sort algorithm. Implement this algorithm in a procedure whose C/C++ description could be

   ```
   void selectionSort(long long nbrArray[], int nbrElts)
   // sort nbrArray[0] .. nbrArray[nbrElts-1]
   // into increasing order using selection sort
   ```

   The first parameter will be the address of the array. Notice that the array is an array of quadwords and the count of how many elements is a doubleword.
5. Write a procedure *avg* to find the average of a collection of quadword integers in an array. Procedure *avg* will have three parameters in the following order:
   (1) The address of the array.
   (2) The number of integers in the array (passed as a doubleword).
   (3) The address of a quadword at which to store the result.

6. The factorial function is defined for a nonnegative integer argument *n* by

$$factorial\,(n) = \begin{cases} 1 \text{ if } n = 0 \\ n \times factorial\ (n-1) \text{ if } n > 0 \end{cases}$$

   Write a value-returning procedure named *factorial* that implements this recursive definition. Pass the argument as a doubleword integer, but return a quadword result.
7. Programming Exercise 5.5.7 gave an algorithm for merging two sorted integer arrays into a third sorted array. Write a procedure *arrayMerge* that implements that design in a procedure described by the following C/C++ function header:

```
void arrayMerge(int array1[], int count1,
                int array2[], int count2),
                int array3[];
// precondition:  array1 and array2 are sorted arrays with
//                count1 and count2 elements, respectively.
//                Each has at least one unused slot.
//                array3 has at least count1+count2 slots
// postcondition: array1 and array2 have been merged into
//                the sorted array3
```

   Each of the arrays is an array of doublewords; *count1* and *count2* are also doublewords. Parameters 1, 3, and 5 are the addresses of the three arrays.
8. Programming Exercise 6.3.11 defined the inner product of two vectors. Write a procedure *innerProduct* that implements the function described by the following C/C++ function header:

```
int innerProduct(int vector1[], int vector2[], int vLength);
// returns the inner product of vector1 and vector2,
// each with vLength components
```

   Each of the vectors is an array of doublewords; *vLength* is also a doubleword. The first two parameters are the addresses of the two arrays.

6.5 Macro Definition and Expansion

A macro was defined in Chapter 3 as a statement that is shorthand for a sequence of other statements. The assembler expands a macro to the statements it represents, and then assembles these new statements. The *windows32* and *windows64* programs in previous chapters have made extensive use of macros defined in the file *io.h*. This section explains how to write macro definitions and tells how the assembler uses these definitions to expand macros into other statements.

A macro definition resembles a procedure definition in a high-level language. The first line gives the name of the macro being defined and a list of parameters; the main part of the definition consists of a collection of model statements that describe the action of the macro in terms of the parameters. A macro is called much like a high-level language procedure, too—the name of the macro is followed by a list of arguments.

These similarities are superficial. A procedure call in a high-level language is compiled into a sequence of instructions to push parameters on the stack followed by a `call`

```
add2   MACRO nbr1, nbr2
; put sum of two doubleword parameters in EAX
       mov  eax, nbr1
       add  eax, nbr2
       ENDM
```

Figure 6.27
Macro to add two integers

instruction, whereas a macro call actually expands into statements given in the macro, with the arguments substituted for the parameters used in the macro definition. Code in a macro is repeated every time a macro is called, but there is just one copy of the code for a procedure. Macros may execute more rapidly than procedure calls since there is no overhead for passing parameters or for `call` and `ret` instructions, but this is usually at the cost of more bytes of object code.

Every macro definition is bracketed by `MACRO` and `ENDM` directives. The format of a macro definition is

```
name  MACRO  list of parameters
      assembly language statements
      ENDM
```

The parameters in the `MACRO` directive are ordinary symbols, separated by commas. The assembly language statements may use the parameters as well as registers, immediate operands, or symbols defined outside the macro. The statements may even include macro calls.

A macro definition can appear anywhere in an assembly language source code file as long as the definition comes before the first statement that calls the macro. It is good programming practice to place macro definitions near the beginning of a source file or in a separate file that is included with the `INCLUDE` directive.

This section gives several examples of macro definitions and macro calls. Figure 6.27 lists the definition of a macro *add2* that finds the sum of two parameters, putting the result in the EAX register. The parameters used to define the macro are *nbr1* and *nbr2*. These labels are local to the definition. The same names could be used for other purposes in the program, although some human confusion might result.

The statements to which `add2` expands depends on the arguments used in a call. For example, the macro call

```
add2 value, 30  ; value + 30
```

expands to

```
; put sum of two doubleword parameters in EAX
mov  eax, value
add  eax, 30
```

The statement

```
add2 value1, value2  ; value1 + value2
```

expands to

```
; put sum of two doubleword parameters in EAX
mov  eax, value1
add  eax, value2
```

The macro call

```
add2 eax, ebx   ; sum of two values
```

expands to

```
; put sum of two doubleword parameters in EAX
mov  eax, eax
add  eax, ebx
```

Note that the instruction `mov eax, eax` is legal, even if it accomplishes nothing.

However, the macro call

```
add2 ebx, eax   ; sum of two values
```

expands to

```
; put sum of two doubleword parameters in EAX
mov  eax, ebx
add  eax, eax
```

that will double the value in EBX, not add the values in EBX and EAX.

In each of these examples the first argument is substituted for the first parameter *nbr1* and the second argument is substituted for the second parameter *nbr2*. Each macro results in `mov` and `add` instructions, but because the types of arguments differ, the object code will vary.

If one of the arguments is missing, the macro will still be expanded. For instance, the macro

```
add2 value
```

expands to

```
; put sum of two doubleword parameters in EAX
mov  eax, value
add  eax,
```

The argument *value* replaces *nbr1* and an empty string replaces *nbr2*. The assembler will report an error, but it will be for the illegal `add` instruction that results from the macro expansion, not directly because of the missing argument.

Similarly, the macro call

```
add2  , value
```

expands to

```
; put sum of two doubleword parameters in EAX
mov  eax,
add  eax, value
```

The comma in the macro call separates the first missing argument from the second argument, *value*. An empty argument replaces the parameter *nbr1*. The assembler will again report an error, this time for the illegal `mov` instruction.

Note again that the definition and expansion for the *add2* macro contain no `ret` instruction. Although macros look much like procedures, they generate in-line code when the macro call is expanded at assembly time.

Figure 6.28 shows the definition of a macro *swap* that will exchange the contents of two doublewords in memory. It is very similar to the 80x86 `xchg` instruction that will not work with two memory operands.

As with the *add2* macro, the code generated by calling the *swap* macro depends on the arguments used. For example, the call

```
swap [ebx], [ebx+4]  ; swap adjacent words in array
```

expands to

```
; exchange two doublewords in memory
     push  eax
     mov  eax, [ebx]
     xchg  eax, [ebx+4]
     mov  [ebx], eax
     pop  eax
```

It might not be obvious to the user that the *swap* macro uses the EAX register, so the `push` and `pop` instructions in the macro protect the user from unexpectedly losing the contents of this register.

```
swap   MACRO dword1, dword2
; exchange two doublewords in memory
     push  eax
     mov  eax, dword1
     xchg  eax, dword2
     mov  dword1, eax
     pop  eax
     ENDM
```

Figure 6.28

Macro to swap two memory words

```
min2   MACRO first, second
; put smaller of two doublewords in the EAX register
     LOCAL endIfMin
     mov  eax, first
     cmp  eax, second
     jle  endIfMin
     mov  eax, second
endIfMin:
     ENDM
```

Figure 6.29

Macro to find smaller of two memory words

Figure 6.29 gives a definition of a macro *min2* that finds the minimum of two doubleword signed integers, putting the smaller in the EAX register. The code for this macro must implement a design with an *if* statement, and such a design usually has at least one assembly language statement with a label. If an ordinary label were used, then it would appear every time a *min2* macro call was expanded and the assembler would produce error messages because of duplicate labels. The solution is to use a LOCAL directive to define a symbol *endIfMin* that is local to the *min2* macro.

The LOCAL directive is used only within a macro definition and goes at the beginning of the definition. It lists one or more symbols, separated by commas, that are used within the macro definition. Each time the macro is expanded and one of these symbols is needed, it is replaced by a symbol starting with two question marks and ending with four hexadecimal digits (??0000, ??0001, etc.). The same *??dddd* symbol replaces the local symbol each instance the local symbol is used in one particular expansion of a macro call. The same symbols may be listed in LOCAL directives in different macro definitions or may be used as regular symbols in code outside of macro definitions.

The macro call

```
min2  [ebx], ecx  ; find smaller of two values
```

might expand to the code

```
; put smaller of two doublewords in the EAX register
     mov  eax, [ebx]
     cmp  eax, ecx
     jle  ??000C
     mov  eax, ecx
??000C:
```

Here, *endIfMin* has been replaced the two instances it appears within the macro definition by *??000C* in the expansion. Another expansion of the same macro in a single file would have a different number after the question marks.

The assembler has several directives that control how macros and other statements are shown in listing files. The most useful are

- .LIST that causes statements to be included in the listing file,
- .NOLIST that completely suppresses the listing of all statements, and
- .NOLISTMACRO that selectively suppresses macro expansions while allowing the programmer's original statements to be listed.

The file *io.h* starts with a .NOLIST directive so that macro definitions do not clutter the listing of a program that includes it. Similarly *io.h* ends with .NOLISTMACRO and .LIST directives so that macro expansion listings do not obscure the programmer's code, but original statements are listed.

We conclude this section by looking at two of the macro definitions in *io.h*. Figure 6.30 shows the *atod* and *dtoa* macro definitions. Like the other macro definitions in *io.h*, these simply expand to procedure calls, and the real work is done by the procedures. The expansion of *atod* is simpler, both because it has only one parameter, and because *atodproc* returns the needed value in EAX. This means that EAX can also be used temporarily to push the necessary parameter onto the stack.

The situation is more complicated with *dtoa*. There is no safe choice of a register to use to push parameter values onto the stack. You can save and restore any register—here EBX is used—but if that register contains the source value, then its contents will be destroyed when the destination parameter is handled. To ensure that the expansion

```
atod        MACRO  source          ; convert ASCII string to integer in EAX
            lea    eax,source      ; source address to AX
            push   eax             ; source parameter on stack
            call   atodproc        ; call atodproc(source)
            add    esp, 4          ; remove parameter
            ENDM

dtoa        MACRO  dest,source     ; convert double to ASCII string
            push   ebx             ; save EBX
            lea    ebx, dest       ; destination address
            push   ebx             ; destination parameter
            mov    ebx, [esp+4]    ; in case source was EBX
            mov    ebx, source     ; source value
            push   ebx             ; source parameter
            call   dtoaproc        ; call dtoaproc(source,dest)
            add    esp, 8          ; remove parameters
            pop    ebx             ; restore EBX
            ENDM
```

Figure 6.30

atod and *dtoa* macro definitions

works even when the original source argument is EBX, the instruction `mov ebx, [esp+4]` restores the original value of EBX after handling the destination parameter and before handling the source parameter. This could have been accomplished by a pair of `pop` and `push` instructions.

Exercises 6.5

1. Using the macro definition for *add2* given in Figure 6.27, show the sequence of statements to which each of the following macro calls expands.
   *(a) `add2 25, ebx`
   (b) `add2 ecx, edx`
   (c) `add2 ; no argument`
2. Using the macro definition for *swap* given in Figure 6.28, show the sequence of statements to which each of the following macro calls expands.
   *(a) `swap value1, value2`
   (b) `swap temp, [ebx]`
   (c) `swap value`
3. Using the macro definition for *min2* given in Figure 6.29, show the sequence of statements to which each of the following macro calls expands.
   *(a) `min2 value1, value2`
   (Assume the local symbol counter is at 000A.)
   (b) `min2 ecx, value`
   (Assume the local symbol counter is at 0019.)

Programming Exercises 6.5

Assemble each macro definition below in a short *console32* or *console64* test-driver program.

1. Write a definition of a macro *add3* that has three doubleword integer parameters and puts the sum of the three numbers in the EAX register.
2. Write a definition of a macro *max2* that has two doubleword integer parameters and puts the maximum of the two numbers in the EAX register.
3. Write a definition of a macro *min3* that has three doubleword integer parameters and puts the minimum of the three numbers in the EAX register.
4. Write a definition of a macro *toUpper* with one parameter, the address of a byte in memory. The code generated by the macro will examine the byte, and if it is the ASCII code for a lowercase letter, replace it by the ASCII code for the corresponding uppercase letter.

6.6 Chapter Summary

This chapter has discussed protocols for implementing procedures in the 80x86 architecture. There are three main concepts involved: (1) how to transfer control from a calling program to a procedure and back, (2) how to pass parameter values to a procedure and results back from the procedure, and (3) how to write procedure code that is independent of the calling program. The stack serves several important purposes in procedure implementation. When a procedure is called, the address of the next instruction is stored

on the stack before control transfers to the first instruction of the procedure. A return instruction retrieves this address from the stack in order to transfer control back to the correct point in the calling program. Argument values (or their addresses) can be pushed onto the stack to pass them to a procedure; when this is done, the base pointer EBP and based addressing provide a convenient mechanism for accessing the values in the procedure. The stack can be used to provide space for a procedure's local variables. The stack is also used to "preserve the environment"—for example, register contents can be pushed onto the stack when a procedure begins and popped off before returning to the calling program so that the calling program does not need to worry about what registers might be altered by the procedure.

In the 32-bit environment there are several protocols used for procedures. This chapter emphasized the *cdecl* protocol that is also used by the Visual Studio C compiler. Following this protocol makes it possible to have a C function call an assembly language procedure, or an assembly language procedure call a C function.

There is just one standard procedure protocol in the 64-bit environment. It uses registers rather than the stack to pass the first four argument values.

A macro is a statement that is shorthand for a sequence of other statements. The assembler expands a macro to the statements it represents, and then assembles these new statements.