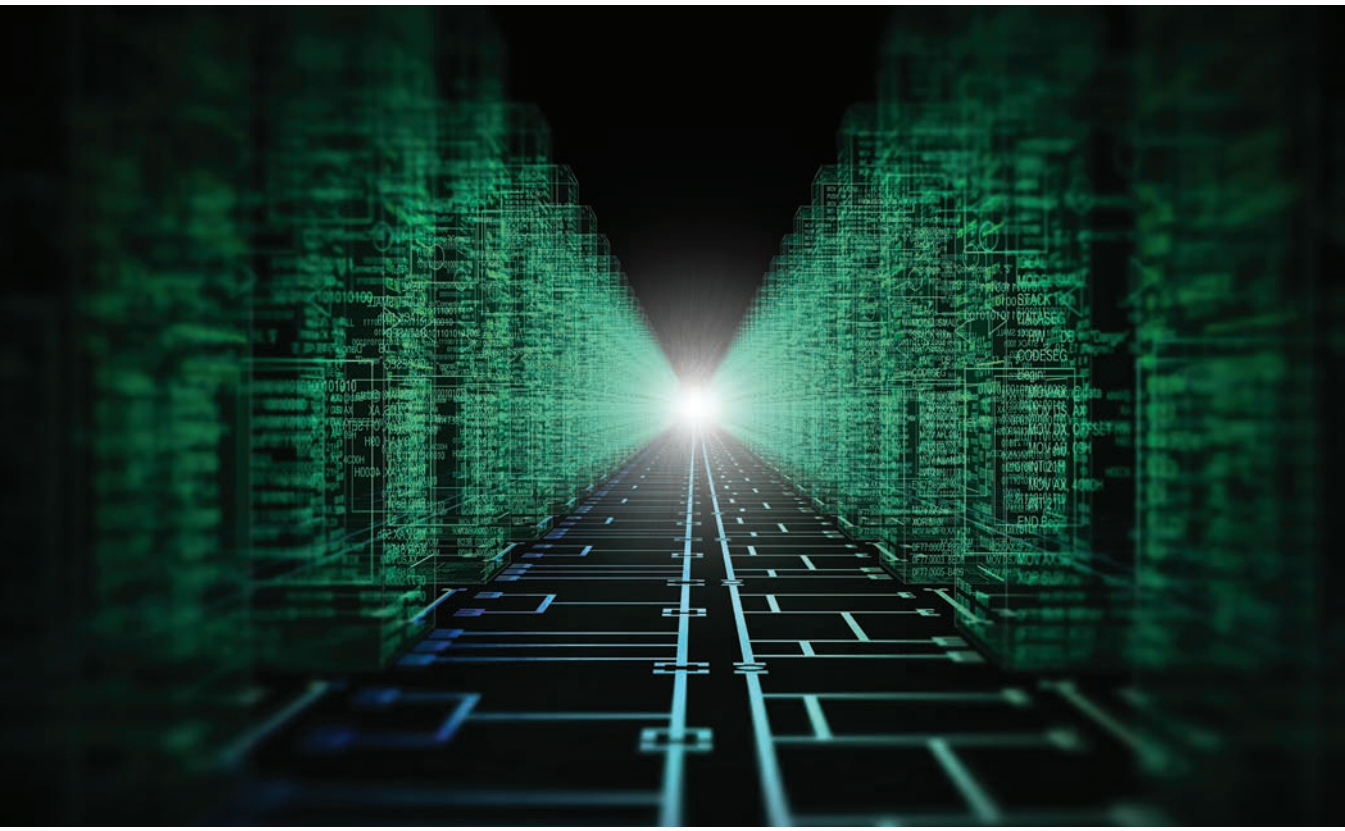
GLOBAL EDITION

Assembly Language

for x86 Processors

SEVENTH EDITION



Kip R. Irvine

ALWAYS LEARNING

PEARSON

Assembly Language for x86 Processors

Seventh Edition
Global Edition

KIP R. IRVINE

Florida International University
School of Computing and Information Sciences

Global Edition contributions by

LYLA B. DAS

National Institute of Technology Calicut
Department of Electronics and Communication Engineering

PEARSON

Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

5.7 Key Terms

5.7.1 Terms

arguments
console window
file handle
global label
input parameter
label
last-in, first-out (LIFO)
link library
nested procedure call
precondition
pop operation
push operation
runtime stack
stack abstract data type
stack data structure
stack pointer register

5.7.2 Instructions, Operators, and Directives

ENDP
POP
POPA
POPAD
POPFD
PROC
PUSH
PUSHA
PUSHAD
PUSHFD
RET
USES

5.8 Review Questions and Exercises

5.8.1 Short Answer

1. Why is the stack of the x86 processor designated as a descending stack?
2. Which instruction pushes the 32-bit EFLAGS register on the stack?
3. Which instruction pops the stack into the EFLAGS register?
4. *Challenge:* Another assembler (called NASM) permits the PUSH instruction to list multiple specific registers. Why might this approach be better than the PUSHAD instruction in MASM? Here is a NASM example:

```
PUSH EAX EBX ECX
```

5. List the steps involved in pushing data into, and popping data from, the stack of the processor.
6. *(True/False):* The RET instruction pops the top of the stack into the instruction pointer.
7. *(True/False):* Nested procedure calls are not permitted by the Microsoft assembler unless the NESTED operator is used in the procedure definition.
8. *(True/False):* A stack is just a data structure created by the action of the stack pointer.
9. *(True/False):* The ESI and EDI registers cannot be used when passing 32-bit parameters to procedures.

10. *(True/False):* The ArraySum procedure (Section 5.2.5) receives a pointer to any array of doublewords.
11. In the context of procedures, what does it mean to pass a parameter?
12. *(True/False):* The USES operator only generates PUSH instructions, so you must code POP instructions yourself.
13. *(True/False):* The register list in the USES directive must use commas to separate the register names.
14. Which statement(s) in the ArraySum procedure (Section 5.2.5) would have to be modified so it could accumulate an array of 16-bit words? Create such a version of ArraySum and test it.
15. What will be the value of the stack pointer after these instructions execute?

```
mov  sp,6800
push ax
push bx
push cx
```

16. Which statement is true about what will happen when the example code runs?

```
 1: main PROC
 2:      push 10
 3:      push 20
 4:      call Ex2Sub
 5:      pop  eax
 6:      INVOKE ExitProcess,0
 7: main ENDP
 8:
 9: Ex2Sub PROC
10:      pop eax
11:      ret
12: Ex2Sub ENDP
```

    a. EAX will equal 10 on line 6
    b. The program will halt with a runtime error on Line 10
    c. EAX will equal 20 on line 6
    d. The program will halt with a runtime error on Line 11

17. Find the contents of CX and DX after each of the following three sets of instructions executes. Assume that AX and BX have been initialized to 3600h and 0D07h respectively.

```
a.  push ax
    push bx
    pop  cx
    pop  dx
b.  push ax
    push bx
    pop  dx
    pop  cx
c.  push bx
    push ax
    pop  cx
    pop  dx
```

```
d.  push bx
    push ax
    pop  dx
    pop  cx
```

18. Which statement is true about what will happen when the example code runs?

```
 1: main PROC
 2:      mov eax,40
 3:      push offset Here
 4:      jmp  Ex4Sub
 5:    Here:
 6:      mov eax,30
 7:      INVOKE ExitProcess,0
 8: main ENDP
 9:
10: Ex4Sub PROC
11:      ret
12: Ex4Sub ENDP
```

a. EAX will equal 30 on line 7
b. The program will halt with a runtime error on Line 4
c. EAX will equal 30 on line 6
d. The program will halt with a runtime error on Line 11

19. Which statement is true about what will happen when the example code runs?

```
 1: main PROC
 2:      mov edx,0
 3:      mov eax,40
 4:      push eax
 5:      call Ex5Sub
 6:      INVOKE ExitProcess,0
 7: main ENDP
 8:
 9: Ex5Sub PROC
10:      pop  eax
11:      pop  edx
12:      push eax
13:      ret
14: Ex5Sub ENDP
```

a. EDX will equal 40 on line 6
b. The program will halt with a runtime error on Line 13
c. EDX will equal 0 on line 6
d. The program will halt with a runtime error on Line 11

20. What values will be written to the array when the following code executes?

```
.data
array DWORD 4 DUP(0)
.code
main PROC
     mov eax,10
     mov  esi,0
```

```
     call proc_1
     add  esi,4
     add  eax,10
     mov  array[esi],eax
     INVOKE ExitProcess,0
main ENDP

proc_1 PROC
     call proc_2
     add  esi,4
     add  eax,10
     mov  array[esi],eax
     ret
proc_1 ENDP

proc_2 PROC
     call proc_3
     add  esi,4
     add  eax,10
     mov  array[esi],eax
     ret
proc_2 ENDP

proc_3 PROC
     mov  array[esi],eax
     ret
proc_3 ENDP
```

5.8.2 Algorithm Workbench

The following exercises can be solved using either 32-bit or 64-bit code.

1. Write a procedure to calculate the sum of the squares of the first N natural numbers, where the value of N is to be entered by the user.
2. Write a program to find the sum of the factorials of three numbers. The numbers must be passed to a procedure, which must then compute the factorials recursively.
3. Functions in high-level languages often declare local variables just below the return address on the stack. Write an instruction that you could put at the beginning of an assembly language subroutine that would reserve space for two integer doubleword variables. Then, assign the values 1000h and 2000h to the two local variables.
4. Write a sequence of statements using indexed addressing that copies an element in a doubleword array to the previous position in the same array.
5. Write a sequence of statements that display a subroutine's return address. Be sure that whatever modifications you make to the stack do not prevent the subroutine from returning to its caller.

5.9 Programming Exercises

When you write programs to solve the programming exercises, use multiple procedures when possible. Follow the style and naming conventions used in this book, unless instructed otherwise by your instructor. Use explanatory comments in your programs at the beginning of each procedure and next to nontrivial statements.

★ **1. Draw Text Colors**

Write a program that displays the same string in four different colors, using a loop. Call the **SetTextColor** procedure from the book's link library. Any colors may be chosen, but you may find it easiest to change the foreground color.

★★ **2. Linking Array Items**

Suppose you are given three data items that indicate a starting index in a list, an array of characters, and an array of link index. You are to write a program that traverses the links and locates the characters in their correct sequence. For each character you locate, copy it to a new array. Suppose you used the following sample data, and assumed the arrays use zero-based indexes:

```
start = 1
chars:     H    A    C    E    B    D    F    G
links:     0    4    5    6    2    3    7    0
```

Then the values copied (in order) to the output array would be A,B,C,D,E,F,G,H. Declare the character array as type BYTE, and to make the problem more interesting, declare the links array type DWORD.

★ **3. Simple Addition (1)**

Write a program that clears the screen, locates the cursor near the middle of the screen, prompts the user for two integers, adds the integers, and displays their sum.

★★ **4. Simple Addition (2)**

Use the solution program from the preceding exercise as a starting point. Let this new program repeat the same steps three times, using a loop. Clear the screen after each loop iteration.

★ **5. BetterRandomRange Procedure**

The RandomRange procedure from the Irvine32 library generates a pseudorandom integer between 0 and $N - 1$. Your task is to create an improved version that generates an integer between M and $N-1$. Let the caller pass M in EBX and N in EAX. If we call the procedure *BetterRandomRange*, the following code is a sample test:

```
mov  ebx,-300           ; lower bound
mov  eax,100            ; upper bound
call BetterRandomRange
```

Write a short test program that calls BetterRandomRange from a loop that repeats 50 times. Display each randomly generated value.

★★ **6. Random Strings**

Create a procedure that generates a random string of length L, containing all capital letters. When calling the procedure, pass the value of L in EAX, and pass a pointer to an array of byte that will hold the random string. Write a test program that calls your procedure 20 times and displays the strings in the console window.

★ **7. Random Screen Locations**

Write a program that displays a single character at 100 random screen locations, using a timing delay of 100 milliseconds. *Hint:* Use the GetMaxXY procedure to determine the current size of the console window.

★★ **8. Color Matrix**

Write a program that displays a single character in all possible combinations of foreground and background colors ($16 \times 16 = 256$). The colors are numbered from 0 to 15, so you can use a nested loop to generate all possible combinations.

★★★ **9. Recursive Procedure**

Direct recursion is the term we use when a procedure calls itself. Of course, you never want to let a procedure keep calling itself forever, because the runtime stack would fill up. Instead, you must limit the recursion in some way. Write a program that calls a recursive procedure. Inside this procedure, add 1 to a counter so you can verify the number of times it executes. Run your program with a debugger, and at the end of the program, check the counter's value. Put a number in ECX that specifies the number of times you want to allow the recursion to continue. Using only the LOOP instruction (and no other conditional statements from later chapters), find a way for the recursive procedure to call itself a fixed number of times.

★★★ **10. Fibonacci Generator**

Write a procedure that produces *N* values in the Fibonacci number series and stores them in an array of doubleword. Input parameters should be a pointer to an array of doubleword, a counter of the number of values to generate. Write a test program that calls your procedure, passing $N = 47$. The first value in the array will be 1, and the last value will be 2,971,215,073. Use the Visual Studio debugger to open and inspect the array contents.

★★★ **11. Finding Multiples of K**

In a byte array of size *N*, write a procedure that finds all multiples of *K* that are less than *N*. Initialize the array to all zeros at the beginning of the program, and then whenever a multiple is found, set the corresponding array element to 1. Your procedure must save and restore any registers it modifies. Call your procedure twice, with $K = 2$, and again with $K = 3$. Let *N* equal to 50. Run your program in the debugger and verify that the array values were set correctly. *Note: This procedure can be a useful tool when finding prime integers. An efficient algorithm for finding prime numbers is known as the* ***Sieve of Eratosthenes****. You will be able to implement this algorithm when conditional statements are covered in Chapter 6.*