Appendix E
Answers to Review Questions

updates: 1/2/06: 6.6, #8

1 Basic Concepts

1.1 Welcome to Assembly Language

1. An assembler converts source-code programs from assembly language into machine language. A linker combines individual files created by an assembler into a single executable program.
2. Assembly language is a good tool for learning how application programs communicate with the computer's operating system, via interrupt handlers. system calls, and common memory areas. Assembly language programming also helps when learning how the operating system loads and executes application programs.
3. In a *one-to-many* relationship, a single statement expands into multiple assembly language or machine instructions.
4. A language whose source programs can be compiled and run on a wide variety of computer systems are said to be *portable*.
5. No. Each assembly language is based on either a processor family or specific computer.
6. Some examples of embedded systems applications are automobile fuel and ignition systems, air-conditioning control systems, security systems, flight control systems, hand-held computers, modems, printers, and other intelligent computer peripherals.
7. *Device drivers* are programs that translate general operating system commands into specific references to hardware details that only the manufacturer would know.
8. C++ does not allow a pointer of one type to be assigned to a pointer of another type. Assembly language has no such restriction regarding pointers.
9. Applications suited to assembly language: Hardware device driver, and embedded systems and computer games requiring direct hardware access.
10. A high-level language may not provide for direct hardware access. Even if it does, awkward coding techniques must often be used, resulting in possible maintenance problems.
11. Assembly language has minimal formal structure, so structure must be imposed by programmers who have varying levels of experience. This leads to difficulties maintaining existing code.
12. Code for the expression: X = (Y * 4) + 3:

```
mov  eax,Y              ; move Y to EAX
mov  ebx,4              ; move 4 to EBX
imul ebx                ; EAX = EAX * EBX
add  eax,3              ; add 3 to EAX
```

Copyright 2003, Pearson Education Inc. All rights reserved.

```
mov  X,eax              ; move EAX to X
```

1.2 Virtual Machine Concept

1. Computers are constructed in layers, so that each layer represents a translation layer from a higher-level instruction set to a lower level instruction set.
2. It is enormously detailed and consists purely of numbers. Hard for humans to understand.
3. True
4. An entire L1 program is converted into an L0 program, by an L0 program specifically designed for this purpose. Then the resulting L0 program is executed directly on the computer hardware.
5. The IA-32's virtual-86 operating mode emulates the architecture of the Intel 8086/8088 processor, used in the original IBM Personal Computer.
6. Java byte code is a low-level language that is quickly executed at run time by a program known as a Java virtual machine (JVM).
7. Digital logic, microarchitecture, instruction set architecture, operating system, assembly language, high-level language.
8. The specific microarchitecture commands are often a proprietary secret. Also, microcode programming is impractical because it often requires 3 or 4 microinstructions to carry out a single primitive operation.
9. Instruction-set architecture
10. Levels 2 and 3.

1.3 Data Representation

1. Least significant bit (bit 0).
2. Most significant bit (the highest numbered bit).
3. (a) 248 (b) 202 (c) 240
4. (a) 53 (b) 150 (c) 204
5. (a) 00010001 (b) 101000000 (c) 00011110
6. (a) 110001010 (b) 110010110 (c) 100100001
7. (a) 2 (b) 4 (c) 8
8. (a) 16 (b) 32 (c) 64
9. (a) 7 (b) 9 (c) 16
10. (a) 12 (b) 16 (c) 22
11. (a) CF57 (b) 5CAD (c) 93EB
12. (a) 35DA (b) CEA3 (c) FEDB
13. (a) 1110 0101 1011 0110 1010 1110 1101 0111

(b) 1011 0110 1001 0111 1100 0111 1010 0001

(c) 0010 0011 0100 1011 0110 1101 1001 0010

14. (a) 0000 0001 0010 0110 1111 1001 1101 0100

    (b) 0110 1010 1100 1101 1111 1010 1001 0101

    (c) 1111 0110 1001 1011 1101 1100 0010 1010

15. (a) 58 (b) 447 (c) 16534
16. (a) 98 (b) 457 (c) 27227
17. (a) FFE6 (b) FE3C
18. (a) FFE0 (b) FFC2
19. (a) +31915 (b) –16093
20. (a) +32667 (b) –32208
21. (a) –75 (b) +42 (c) –16
22. (a) –128 (b) –52 (c) –73
23. (a) 11111011 (b) 11011100 (c) 11110000
24. (a) 10111000 (b) 10011110 (c) 11100110
25. 58h and 88d
26. 4Dh and 77d
27. To handle international character sets that require more than 256 codes.
28. $2^{256} - 1$
29. $+2^{255} - 1$

1.4 Boolean Operations

1. (NOT X) OR Y
2. X AND Y
3. T
4. F
5. T
6. Truth table:

A	B	$A \vee B$	$\neg(A \vee B)$
F	F	F	T
F	T	T	F

A	B	$A \vee B$	$\neg(A \vee B)$
T	F	T	F
T	T	T	F

7. Truth table:

A	B	$\neg A$	$\neg B$	$\neg A \wedge \neg B$
F	F	T	T	T
F	T	T	F	F
T	F	F	T	F
T	T	F	F	F

8. 16, or (2^4)
9. 2 bits, producing the following values: 00, 01, 10, 11

2 IA-32 Processor Architecture

2.1 General Concepts

1. Control Unit, Arithmetic Logic Unit, and the clock.
2. Data, Address, and Control buses.
3. Conventional memory is outside the CPU, and it responds more slowly to access requests. Registers are hard-wired inside the CPU.
4. Fetch, decode, execute.
5. Fetch memory operands, store memory operands
6. During the fetch step
7. Executing processor stages in parallel, making possible the overlapped execution of machine instructions.
8. 10 clock cycles
9. 12 cycles (5 + (8 – 1)
10. A superscalar processor contains two or more execution pipelines
11. 15 clock cycles (5 + 10)
12. Section 2.1.4.1 mentions the file name, file size, starting location on the disk. (Most directories also store the file’s last modification date and time.)

13. The OS executes a branch (like a GOTO) to the first machine instruction in the program.
14. The CPU executes multiple tasks (programs) by rapidly switching from one program to the next. This gives the impression that all programs are executing at the same time.
15. The OS scheduler determines how much time to allot to each task, and it switches between tasks.
16. The program counter, the task's variables, and the CPU registers (including the status flags).
17. 3.33×10^{-10} , which is $1.0 / 3.0 \times 10^{9}$

2.2 IA-32 Processor Architecture

1. Real-address mode, Protected mode, and System Managment mode
2. EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
3. CS, DS, SS, ES, FS, GS
4. Loop counter
5. EBP
6. Most common: Carry, Sign, Zero, Overflow. Less common: Auxiliary Carry, Parity.
7. Carry
8. Overflow
9. Sign
10. Floating-Point Unit
11. 80 bits
12. The Intel 80386
13. The Pentium
14. The Pentium (see the Chapter corrections on the book's Web site)
15. CISC means *complex instruction set*. A large collection of instructions, some of which perform sophisticated operations that might be typical of a high-level language.
16. The term RISC stands for *reduced instruction set*. A small set of very simple (atomic) instructions that may be combined into more complex operations.

2.3 IA-32 Memory Management

1. 4 GB (0 to FFFFFFFFh)
2. 1 MB (0 to FFFFFh)
3. linear (absolute)
4. 09600h
5. 0CFF0h

6. 32 bits
7. SS register
8. Local descriptor table
9. Global descriptor table
10. The total size of all programs loaded into memory can exceed the amount of physical memory installed in the computer.
11. This is an open-ended question, of course. It is a fact that MS-DOS first had to run on the 8086/8088 processors, which only supported Real-address mode. When later processors came out which supported Protected mode, my guess is that Microsoft wanted MS-DOS to continue to run on the older processors. Otherwise, customers with older computers would refuse to upgrade to new versions of MS-DOS.
12. The following segment-offset addresses point to the same linear address: 0640:0100, and 0630:0200.

2.4 Components of an IA-32 Microcomputer

1. SRAM stands for Static RAM, used in CPU cache memory.
2. Pentium
3. The 8259 is the interrupt controller chip, sometimes called PIC, that schedules hardware interrupts and interrupts the CPU.
4. On either the video board, or on the motherboard (special memory area).
5. A beam of electrons illuminates phosphorus dots on the screen called pixels. Starting at the top of the screen, the gun fires electrons from the left side to the right in a horizontal row, briefly turns off, and returns to the left side of the screen to begin a new row. Horizontal retrace refers to the time period when the gun is off between rows. When the last row is drawn, the gun turns off (called the vertical retrace) and moves to the upper left corner of the screen to start all over.
6. Dynamic RAM, Static RAM, Video RAM, and CMOS RAM.
7. Static RAM
8. The computer can query a device connected via USB to find out its name, device type, and the type of driver it supports. The computer can also suspend power to individual devices. None of these capabilities are possible with serial and parallel ports.
9. Upstream and downstream
10. 16550 UART (universal asynchronous receiver transmitter)

2.5 Input-Output System

1. The application program level
2. BIOS functions communicate directly with the system hardware. They are independent of the operating system.

3. New devices are invented all the time, with capabilities that were often not anticipated when the BIOS was written.
4. The BIOS level.
5. The operating system, BIOS, and hardware levels.
6. Game programs often try to take advantage of the latest features in specialized sound cards. It should be noted that MS-DOS game applications were more prone to do this than games running under MS-Windows. In fact, Windows-NT, 2000 and XP all prevent applications from directly accessing system hardware.
7. No. The same BIOS would work for both operating systems. Many computer owners install two or three operating systems on the same computer. They would certainly not want to change the system BIOS every time they rebooted the computer!

3 Assembly Language Fundamentals

3.1 Basic Elements of Assembly Language

1. h,q,o,d,b,r,t,y
2. No (a leading zero is required)
3. No (they have the same precedence)
4. Expression: 10 MOD 3
5. Real number constant: +3.5E-02
6. No, they can also be enclosed in double quotes
7. directives
8. 247 characters
9. True
10. True
11. False
12. True
13. label, mnemonic, operand(s), comment
14. True
15. True
16. Code example:

```
Comment !
      This is a comment
      This is also a comment
!
```

17. Because the addresses coded in the instructions would have to be updated whenever new variables were inserted before existing ones.

3.2 Example: Adding Three Integers

1. The INCLUDE directive copies necessary definitions and setup information from the *Irvine32.inc* text file. The data from this file is inserted into the data stream read by the assembler.
2. The .CODE directove marks the beginning of the code segment.
3. code, data, and stack.
4. By calling the **DumpRegs** procedure
5. The **exit** statement
6. The PROC directive
7. The ENDP directive
8. It marks the last line of the program to be assembled, and the label next to END identifies the program's entry point (where execution begins).
9. PROTO declares the name of a procedure that is called by the current program.

3.3 Assembling, Linking, and Running Programs

1. Object (.OBJ) and listing (.LST) files.
2. True
3. True
4. Loader
5. Executable (.EXE) and map (.MAP).
6. The /Fl option
7. The /Zi option
8. It tells the linker to produce a *Win32 Console* application.
9. There are two many to mention here, but you can view their names by opening Kernel32.lib using the TextPad editor supplied on the book's CD-ROM. The file will display in hexadecimal. Scroll down to offset 1840h, and look at the various function names listed from that point on.
10. /ENTRY sets the program's starting address (the entry point). For example, suppose you wanted your program to begining execution at the Startup procedre. The link command line would be:

    ```
    link32 /ENTRY:Startup
    ```

    This is a challenging question because you cannot find the answer in Appendix A. Instead, you can read about the Microsoft 32-bit linker command-line options by visiting the MSDN Web site and searching for *linker reference*.)

3.4 Defining Data

1. var1 SWORD ?
2. var2 BYTE ?
3. var3 SBYTE ?
4. var4 QWORD ?
5. SDWORD
6. var5 SDWORD -2147483648
7. wArray WORD 10,20,30
8. myColor BYTE "blue",0
9. dArray DWORD 50 DUP(?)
10. myTestString BYTE 500 DUP("TEST")
11. bArray BYTE 20 DUP(0)
12. 21h,43h,65h,87h

3.5 Symbolic Constants

1. BACKSPACE = 08h
2. SecondsInDay = 24 * 60 * 60
3. ArraySize = ($ – myArray)
4. ArraySize = ($ – myArray) / TYPE DWORD
5. PROCEDURE TEXTEQU <PROC>
6. Code example:

```
Sample TEXTEQU <"This is a string">
MyString BYTE Sample
```

7. SetupESI TEXTEQU <mov esi,OFFSET myArray>

4 Data Transfers, Addressing, and Arithmetic

4.1 Data Transfer Instructions

1. Register, immediate, and memory
2. False
3. False
4. True

5. A 32-bit register or memory operand
6. A 16-bit immediate (constant) operand
7. (a) not valid (b) valid (c) not valid (d) not valid (e) not valid (f) not valid (g) valid (h) not valid
8. (a) FCh (b) 01h
9. (a) 1000h (b) 3000h (c) FFF0h (d) 4000h
10. (a) 00000001h (b) 00001000h (c) 00000002h (d) FFFFFFFCh

4.2 Addition and Subtraction

1. inc val2
2. sub eax,val3
3. Code:
```
mov ax,val4
sub val2,ax
```
4. CF = 0, SF = 1
5. CF = 1, SF = 1
6. Write down the following flag values:

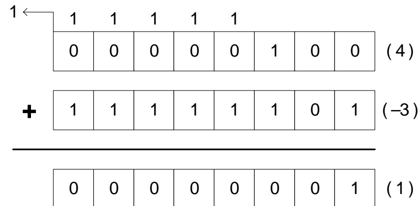
   (a) CF = 1, SF = 0, ZF = 1, OF = 0

   (b) CF = 0, SF = 1, ZF = 0, OF = 1

   (c) CF = 0, SF = 1, ZF = 0, OF = 0
7. Code example:
```
mov ax,val2
neg ax
add ax,bx
sub ax,val4
```
8. No
9. Yes
10. Yes (for example, mov al,−128...followed by... neg al)
11. No
12. Setting the Carry and Overflow flags at the same time:
```
mov al,80h
add al,80h
```
13. Setting the Zero flag after INC and DEC to indicate unsigned overflow:
```
mov al,0FFh
inc al
jz overflow_occurred
```

```
mov bl,1
dec bl
jz overflow_occurred
```

14. Subtracting 3 from 4 (unsigned). Carry out of MSB is inverted and placed in the Carry flag:




```
mov al,4
sub al,3     ; CF = 0
```

4.3 Data-Related Operators and Directives

1. False
2. False
3. True
4. False
5. True
6. Data directive:

```
.data
ALIGN 2
myBytes BYTE 10h,20h,30h,40h
etc.
```

7. (a) 1 (b) 4 (c) 4 (d) 2 (e) 4 (f) 8 (g) 5
8. mov dx, WORD PTR myBytes
9. mov al, BYTE PTR myWords+1
10. mov eax,DWORD PTR myBytes
11. Data directive:

```
myWordsD LABEL DWORD
myWords WORD 3 DUP(?),2000h
.data
mov eax,myWordsD
```

12. Data directive:

```
myBytesW LABEL WORD
myBytes BYTE 10h,20h,30h,40h
.code
mov ax,myBytesW
```

4.4 Indirect Addressing

1. False
2. True
3. False
4. False
5. True - (the PTR operator is required)
6. True
7. (a) 10h (b) 40h (c) 003Bh (d) 3 (e) 3 (f) 2
8. (a) 2010h (b) 003B008Ah (c) 0 (d) 0 (e) 0044h

4.5 JMP and LOOP Instructions

1. True
2. False
3. 4,294,967,296 times
4. False
5. True
6. CX
7. ECX
8. False (–128 to +127 bytes from the current location)
9. This is a trick! The program does not stop, because the first LOOP instruction decrements ECX to zero. The second LOOP instruction decrements ECX to FFFFFFFFh, causing the outer loop to repeat.
10. Insert the following instruction at label L1: `push ecx`. Also, insert the following instruction before the second LOOP instruction: `pop ecx`. (Once you have added these instructions, the final value of eax is 1Ch.)

5 Procedures

5.1 Introduction

(no review questions)

5.2 Linking to an External Library

1. False - (it contains object code)
2. Code example:

```
MyProc PROTO
```

3. Code example:

```
call MyProc
```

4. Irvine32.lib
5. Kernel32.lib
6. Kernel32.dll is a dynamic link library that is a fundamental part of the MS-Windows operating system.
7. %1

5.3 The Book's Link Library

1. RandomRange procedure
2. WaitMsg procedure
3. Code example:

```
      mov  eax,700
      call Delay
```

4. WriteDec procedure
5. Gotoxy procedure
6. INCLUDE Irvine32.inc
7. PROTO statements (procedure prototypes) and constant definitions. (There are also text macros, but they are not mentioned in this chapter.)
8. ESI contains the data's starting address, ECX contains the number of data units, and EBX contains the data unit size (byte, word, or doubleword).
9. EDX contains the offset of an array of bytes, and ECX contains the maximum number of characters to read.
10. Carry, Sign, Zero, and Overflow, and EFL displays the flag bits in hexadecimal.
11. Code example:

```
.data
str1 BYTE "Enter identification number: ",0
idStr BYTE 15 DUP(?)
.code
      mov  edx,OFFSET str1
      call WriteString
      mov  edx,OFFSET idStr
```

```
mov  ecx,(SIZEOF idStr) - 1
call ReadString
```

5.4 Stack Operations

1. SS and ESP
2. The runtime stack is only type of stack that is managed directly by the CPU. For example, it holds the return addresses of called procedures.
3. LIFO stands for "last in, first out". The last value pushed into the stack is the first value popped out from the stack.
4. ESP is decremented by 4.
5. True
6. False - (you can push both 16-bit and 32-bit values)
7. True
8. False (yes, it can, from the 80186 processor onwards).
9. PUSHAD
10. PUSHFD
11. POPFD
12. NASM's approach permits the programmer to be specific about which registers are to be pushed. PUSHAD, on the other hand, does not have that flexibility. This becomes important when a procedure needs to save several registers, and at the same time return a value to its caller in the EAX register. In this type of situation, EAX cannot be pushed and popped because the return value would be lost.
13. Equivalent to PUSH EAX:

```
sub esp,4
mov [esp],eax
```

5.5 Defining and Using Procedures

1. True
2. False
3. Execution would contine beyond the end of the procedure, possibly into the beginning of another procedure. This type of programming bug is often difficult to detect!
4. *Receives* indicates the input parameters given to the procedure when it is called. *Returns* indicates what value, if any, the procedure produces when it returns it its caller.
5. False - (it pushes the offset of the instruction *following* the call)
6. True
7. True

8. False - (there is no NESTED operator)
9. True
10. False
11. True - (it also receives a count of the number of array elements)
12. True
13. False
14. False
15. The following statements would have to be modified:

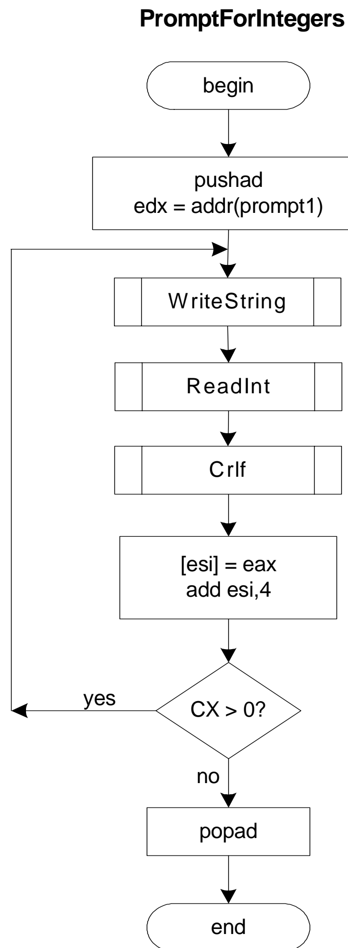
```
add eax,[esi]    becomes -->  add ax,[esi]
add esi,4        becomes -->  add esi,2
```

5.6 Program Design Using procedures

1. functional decomposition, or top-down design
2. Clrscr, WriteString, ReadInt, and WriteInt
3. A stub program contains all of its important procedures, but the procedures are either empty or nearly empty.
4. False - (it receives a pointer to an array)
5. The following statements would have to be modified:

```
mov [esi],eax    becomes -->   mov [esi],ax
add esi,4        becomes -->   add esi,2
```

6. Flowchart of the PromptForIntegers procedure:



6 Conditional Processing

6.1 Introduction

(no review questions)

6.2 Boolean and Comparison Instructions

1. (a) 00101101 (b) 01001000 (c) 01101111 (d) 10100011
2. (a) 85h (b) 34h (c) BFh (d) AEh

3. (a) CF=0, ZF=0, SF=0

   (b) CF=0, ZF=0, SF=0

   (c) CF=1, ZF=0, SF=1

4. `and ax,00FFh`

5. `or ax,0FF00h`

6. `xor eax,0FFFFFFFFh`

7. `test eax,1        ; (low bit set if eax is odd)`

8. `or al,00100000b`

9. `and al,00001111b`

10. Code example:

```
.data
memVal DWORD ?
.code
mov al,BYTE PTR memVal
xor al,BYTE PTR memVal+1
xor al,BYTE PTR memVal+2
xor al,BYTE PTR memVal+3
```

6.3 Conditional Loops

1. JA, JNBE, JAE, JNB, JB, JNAE, JBE, JNA

2. JG, JNLE, JGE, JNL, JL, JNGE, JLE, JNG

3. JECXZ

4. Yes

5. No (JB uses unsigned operands, whereas JL uses signed operands.)

6. JBE

7. JL

8. No (8109h is negative, and 26h is positive.)

9. Yes

10. Yes (The unsigned representation of –42 is compared to 26.)

11. Code:

```
cmp dx,cx
jbe L1
```

12. Code:

```
cmp ax,cx
jg  L2
```

13. Code:

```
and al,11111100b
jz  L3
jmp L4
```

14. The XOR instruction in the three-instruction sequence will always clear the Carry flag. The BTC instruction may or may not clear the Carry flag, depending on the value in sempahore.

6.4 Conditional Loop Instructions

1. False
2. True
3. True
4. Code example:

```
.data
array SWORD 3,5,14,-3,-6,-1,-10,10,30,40,4
sentinel SWORD 0
.code
main PROC
      mov esi,OFFSET array
      mov ecx,LENGTHOF array
next:
      test WORD PTR [esi],8000h              ; test sign bit
      pushfd                                 ; push flags on stack
      add esi,TYPE array
      popfd                                  ; pop flags from stack
      loopz next                             ; continue loop while
ZF=1
      jz quit                                ; none found
      sub esi,TYPE array                     ; ESI points to value
```

5. If a matching value were not found, ESI would end up pointing beyond the end of the array. This could cause data to be corrupted if ESI were dereferenced and used to modify memory.

6.5 Conditional Structures

(We will assume that all values are unsigned in this section).

1. Code example:

```
      cmp bx,cx
      jna next
      mov X,1
next:
```

2. Code example:

```
      cmp  dx,cx
```

```
        jnbe L1
        mov  X,1
        jmp  next
L1:     mov  X,2
next:
```

3. Code example:

```
        cmp val1,cx
        jna L1
        cmp cx,dx
        jna L1
        mov X,1
        jmp next
L1:     mov X,2
next:
```

4. Code example:

```
        cmp bx,cx
        ja  L1
        cmp bx,val1
        ja  L1
        mov X,2
        jmp next
L1:     mov X,1
next:
```

5. Code example:

```
        cmp bx,cx                 ; bx > cx?
        jna L1                    ; no: try condition after OR
        cmp bx,dx                 ; yes: is bx > dx?
        jna L1                    ; no: try condition after OR
        jmp L2                    ; yes: set X to 1
;-----------------OR(dx > ax) ------------------------
L1:     cmp dx,ax                 ; dx > ax?
        jna L3                    ; no: set X to 2
L2:     mov X,1                   ; yes:set X to 1
        jmp next                  ; and quit
L3:     mov X,2                   ; set X to 2
next:
```

6. Future changes to the table will alter the value of NumberOfEntries. We might forget to update the constant manually, but the assembler can correctly adjust a calculated value.

7. Code example: (pending)

6.6 Application: Finite-State Machines

1. A directed graph (also known as a *diagraph*).
2. Each node is a state.
3. Each edge is a transition from one state to another, caused by some input.
4. State C
5. An infinite number of digits.
6. The FSM enters an error state.
7. No. The proposed FSM would permit a signed integer to consist of only a plus (+) or minus (–) sign. The FSM in Section 6.6.2 would not permit that.
8. FSM that recognizes real numbers without exponents:

6.7 Using the .IF Directive (Optional)

(no review questions)

7 Integer Arithmetic

7.1 Introduction

(no review questions)

7.2 Shift and Rotate Instructions

1. ROL
2. RCR
3. SAR
4. RCL
5. Code example:

```
        shr al,1                    ; shift AL into Carry flag
        jnc next                    ; Carry flag set?
        or  al,80h                  ; yes: set highest bit
next:                               ; no: do nothing
```

6. The Carry flag receives the lowest bit of AX (before the shift)
7. `shl eax,4`
8. `shr ebx,2`
9. `ror dl,4` (or: `rol dl,4`)
10. `shld dx,ax,1`
11. (a) 6Ah (b) EAh (c) FDh (d) A9h
12. (a) 9Ah (b) 6Ah (c) 0A9h (d) 3Ah
13. Code example:

```
      shr ax,1                  ; shift AX into Carry flag
      rcr bx,1                  ; shift Carry flag into BX
; Using SHRD:
      shrd bx,ax,1
```

14. Code example: *(revised and tested 10/24/2002)*

```
      mov ecx,32                ; loop counter
      mov bl,0                  ; counts the '1' bits
L1:   shr eax,1                 ; shift into Carry flag
      jnc L2                    ; Carry flag set?
      inc bl                    ; yes: add to bit count
L2:   loop L1                   ; continue loop

; if BL is odd, clear the parity flag
; if BL is even, set the parity flag
      shr bl,1
      jc  odd
      mov bh,0
      or  bh,0                  ; PF = 1
      jmp next
odd:
      mov bh,1
      or  bh,1                  ; PF = 0
next:
```

7.3 Shift and Rotate Applications

1. This problem requires us to start with the high-order byte and work our way down to the lowest byte:

```
byteArray BYTE 81h,20h,33h
.code
shr byteArray+2,1
rcr byteArray+1,1
rcr byteArray,1
```

2. This problem requires us to start with the low-order word and work our way up to the highest word:

```
wordArray WORD 810Dh,0C064h,93ABh
.code
shl wordArray,1
rcl wordArray+2,1
rcl wordArray+4,1
```

3. The multiplier (24) can be factored into 16 * 8:

```
mov ebx,eax                ; save a copy of eax
shl eax,4                  ; multiply by 16
shl ebx,3                  ; multiply by 8
add eax,ebx                ; add the products
```

4. As the hint explains, the multiplier (21) can be factored into 16 * 4 + 1:

```
mov ebx,eax                ; save a copy of eax
mov ecx,eax                ; save another copy of eax
shl eax,4                  ; multiply by 16
shl ebx,2                  ; multiply by 4
add eax,ebx                ; add the products
add eax,ecx                ; add original value of eax
```

5. Change the instruction at label L1 to: `shr eax,1`
6. We will assume that the time stamp word is in the DX register:

```
shr dx,5
and dl,00111111b           ; (leading zeros optional)
mov bMinutes,dl            ; save in variable
```

7.4 Multiplication and Division Instructions

1. The product is stored in registers that are twice the size of the multiplier and multiplicand. If you multiply 0FFh by 0FFh, for example, the product (FE01h) easily fits within 16 bits.
2. When the product fits completely within the lower register of the product, IMUL sign-extends the product into the upper product register. MUL, on the other hand, zero-extends the product.
3. With IMUL, the Carry and Overflow flags are set when the upper half of the product is not a sign extension of the lower half of the product.
4. EAX
5. AX
6. AX
7. Code example:

```
mov ax,dividendLow
cwd                        ; sign-extend dividend
mov bx,divisor
idiv bx
```

8. DX = 0002h, AX = 2200h

9. AX = 0306h

10. EDX = 0, EAX = 00012340h

11. The DIV will cause a divide overflow, so the values of AX and DX cannot be determined.

12. Code example:

```
mov ax,3
mov bx,-5
imul bx
mov val1,ax                    ; product

// alternative solution:
mov al,3
mov bl,-5
imul bl
mov val1,ax                    ; product
```

13. Code example:

```
mov ax,-276
cwd                            ; sign-extend AX into DX
mov bx,10
idiv bx
mov val1,ax                    ; quotient
```

14. Implement the unsigned expression: val1 = (val2 * val3) / (val4 – 3).

```
mov eax,val2
mul val3
mov ebx,val4
sub ebx,3
div ebx
mov val1,eax
```

(You can substitute any 32-bit general-purpose register for EBX in this example.)

15. Implement the signed expression: val1 = (val2 / val3) * (val1 + val2).

```
mov eax,val2
cdq                            ; extend EAX into EDX
idiv val3                      ; EAX = quotient
mov ebx,val1
add ebx,val2
imul ebx
mov val1,eax                   ; lower 32 bits of product
```

(You can substitute any 32-bit general-purpose register for EBX in this example.)

7.5 Extended Addition and Subtraction

1. The ADC instruction adds both a source operand and the Carry flag to a destination operand.
2. The SBB instruction subtracts both a source operand and the Carry flag from a destination operand.
3. EAX = C0000000h, EDX = 00000010h
4. EAX = F0000000h, EDX = 000000FFh
5. DX = 0016h
6. In correcting this example, it is easiest to reduce the number of instructions. You can use a single register (ESI) to index into all three variables. ESI should be set to zero before the loop because the integers are stored in little endian order with their low-order bytes occurring first:

```
      mov ecx,8                              ; loop counter
      mov esi,0                              ; use the same index
reg
      clc                                    ; clear Carry flag
top:
      mov al,byte ptr val1[esi]              ; get first number
      sbb al,byte ptr val2[esi]              ; subtract second
      mov byte ptr result[esi],al            ; store the result
      inc esi                                ; move to next pair
      loop top
```

Of course, you could easily reduce the number of loop iterations by adding doublewords rather than bytes.

7.6 ASCII and Unpacked Decimal Arithmetic

1. Code example:

```
      or ax,3030h
```

2. Code example:

```
      and ax,0F0Fh
```

3. Code example:

```
      and ax,0F0Fh               ; convert to unpacked
      aad
```

4. Code example:

```
      aam
```

5. Code example (displays binary value in AX):

```
out16 PROC
      aam
      or               ax,3030h
      push             eax
      mov              al,ah
```

```
        call          WriteChar
        pop           eax
        call          WriteChar
        ret
out16 ENDP
```

6. After AAA, AX would equal 0108h. Intel says: First, if the lower digit of AL is greater than 9 or the AuxCarry flag is set, add 6 to AL and add 1 to AH. Then in all cases, AND AL with 0Fh. Pseudocode:

```
IF ((AL AND 0FH) > 9) OR (AuxCarry = 1) THEN
      add 6 to AL
      add 1 to AH
END IF
AND AL with 0FH;
```

7.7 Packed Decimal Arithmetic

1. When the sum of a packed decimal addition is greater than 99, DAA sets the Carry flag. For example:

```
mov al,56h
add al,92h                ; AL = E8h
daa                       ; AL = 48h, CF=1
```

2. When a larger packed decimal integer is subtracted from a small one, DAS sets the Carry flag. For example:

```
mov                       al,56h
sub                       al,92h        ; AL = C4h
das                                     ; AL = 64h, CF=1
```

3. $n + 1$ bytes.

4. Suppose AL = 3Dh, AF = 0, and CF = 0. Because the lower digit (D) is > 9, we subtract 6 from D. AL now equals 37h. Because the upper digit (3) is <= 9 and CF = 0, no other adjustments are necessary. DAS produces AL = 37h.

8 Advanced Procedures

8.1 Introduction

(no review questions)

8.2 Local Variables

1. (a) automatically restricts access to statements within a single procedure; (b) local variables make efficient use of memory; (c) you can use the same variable name in multiple procedures.
2. False
3. False; you can define many more than three.
4. True
5. Declaration: `LOCAL pArray:PTR DWORD`
6. Declaration: `LOCAL buffer[20]:BYTE`
7. Declaration: `LOCAL pwArray:PTR WORD`
8. Declaration: `LOCAL myByte:SBYTE`
9. Declaration: `LOCAL myArray[20]:DWORD`

8.3 Stack Parameters

1. True
2. False - (it may include many more arguments than three)
3. False
4. True
5. False
6. True
7. True
8. True - (the assembler will not catch the error)
9. True - when the immediate value is dereferenced, it will probably point to an invalid memory location.
10. no - the values are the same
11. Declaration:

```
MultArray PROC ptr1:PTR DWORD,
  ptr2:PTR DWORD,
  count:DWORD             ; (may be byte, word, or dword)
```

12. Declaration:

```
MultArray PROTO ptr1:PTR DWORD,
        ptr2:PTR DWORD,
        count:DWORD             ; (may be byte, word, or dword)
```

13. It uses input-output parameters.
14. It is an output parameter.

15. The following code is shown in the listing file, when the assembler's /Sg option is used. It shows that count, the second argument, was pushed on the stack first before the offset of myArray:

```
INVOKE SumArray, ADDR myArray, count
  push   +0000000Ah
  push   OFFSET myArray
  call   SumArray
```

(For more information about the assembler's command-line options, see Appendix D.)

8.4 Stack Frames

1. True
2. False - (a positive value is *subtracted* from the stack pointer)
3. True - (each stack position in Protected mode uses 4 bytes)
4. False
5. One code segment, and one data segment. All code and data are near, which means they can be reached using only 16-bit offsets.
6. Used in Protected mode. All offsets are 32 bits, and both code and data belong to the same segment.
7. The C option preserves the case of identifiers and prepends a leading underscore to external names. The PASCAL option converts all identifiers to upper case.
8. It passes an integer constant to the RET instruction. This constant is added to the stack pointer right after the RET instruction has popped the procedure's return address off the stack.
9. Stack frame diagram:

Value	Location
10h	[EBP + 16]
20h	[EBP + 12]
30h	[EBP + 8]
(return addr)	[EBP + 4]
EBP	<-- ESP

10. Code example:

```
AddThree PROC
; modeled after the AddTwo procedure in Section 8.4.3:
      push ebp
      mov  ebp,esp
      mov  eax,[ebp + 16]       ; 10h
      add  eax,[ebp + 12]       ; 20h
      add  eax,[ebp + 8]        ; 30h
      pop  ebp
      ret  12
AddThree ENDP
```

11. LEA can return the offset of an indirect operand; it is particularly useful for obtaining the offset of a stack parameter.
12. 4 bytes
13. The C calling convention, because it specifies that arguments must be pushed on the stack in reverse order, makes it possible to create a procedure/function with a variable number of parameters. The last parameter pushed on the stack can be a count specifying the number of parameters already pushed on the stack. In the following diagram, for example, the count value is located at [EBP + 8]:

10h	[EBP + 20]
20h	[EBP + 16]
30h	[EBP + 12]
3	[EBP + 8]
(return addr)	[EBP + 4]
EBP	<-- ESP

8.5 Recursion

1. False
2. When n equals 0
3. The following code executes after the recursive call:

```
ReturnFact:
      mov ebx,[ebp+8]
      mul ebx
L2:   pop ebp
      ret 4
```

4. The calculated value would exceed the range of an unsigned doubleword, and would roll past zero. The output would appear to be smaller than 12 factorial.
5. 12! uses 156 bytes of stack space. *Rationale:* From Figure 8-1, we see that when $n = 0$, 12 stack bytes are used (3 entries). When $n = 1$, 24 bytes are used. When $n = 2$, 36 bytes are used. Therefore, the amount of stack space required for n! is $(n+1)*12$.
6. A recursive Fibonacci algorithm uses system resources inefficiently because each call to the fibonacci function with a value of *n* generates function calls for all fibonacci numbers between 1 and *n*-1. Here is the pseudocode to generate the first 20 values:

```
for(int i = 1; i <= 20; i++)
  print( fibonacci(i) );

int fibonacci(int n)
{
      if( n == 1 )
       return 1;
      elseif( n == 2 )
```

```
            return 2;
          else
            return fibonacci(n-1) + fibonacci(n-2);
     }
```

8.6 Creating Multimodule Programs

1. True
2. False
3. True
4. False

9 Strings and Arrays

9.1 Introduction

(no review questions)

9.2 String Primitive Instructions

1. EAX
2. CMPSD
3. (E)DI
4. LODSW
5. Repeat while ZF = 1
6. 1 (set)
7. 2
8. Regardless of which operands are used, CMPS still compares the contents of memory pointed to by ESI to the memory pointed to by EDI.
9. one byte beyond the matching character
10. REPNE (REPNZ)

9.3 Selected String Procedures

1. False (it stops when the null terminator of the shorter string is reached)
2. True
3. False
4. False

5. 1 (set)

6. Check for string containing only the character to be trimmed.

7. The digit is unchanged.

8. REPNE (REPNZ)

9. The length would be: $(EDI_{final} - EDI_{initial}) – 1$

9.4 Two-Dimensional Arrays

1. Any general-purpose 32-bit registers.

2. [ebx+esi]

3. array[ebx + esi]

4. 16

5. Code example:

```
mov esi,2                        ; row
mov edi,3                        ; column
mov eax,[esi*16 + edi*4]
```

6. BP points to the stack segment in Real-address mode.

7. No (the flat memory model uses the same segment for stack and data).

9.5 Searching and Sorting Integer Arrays

1. n – 1 times

2. n – 1 times

3. No: it decreases by 1 each

4. $T(5000) = 0.5 * 10^2$

5. $(\log_2 128) + 1 = 8$

6. $(\log_2 n) + 1$

7. EDX and EDI were already compared

8. change each JMP L4 instruction to JMP L1

10 Structures and Macros

10.1 Structures

1. Structures are essential whenever you need to pass a large amount of data between procedures. One variable can be used to hold all the data.
2. Structure Definition:

```
MyStruct STRUCT
      field1 WORD ?
      field2 DWORD 20 DUP(?)
MyStruct ENDS
```

3. temp1 MyStruct <>
4. temp2 MyStruct <0>
5. temp3 MyStruct <, 20 DUP(0)>
6. array MyStruct 20 DUP(<>)
7. mov ax,array.field1
8. Code example:

```
mov esi,OFFSET array
add esi,3 * (TYPE myStruct)
mov (MyStruct PTR[esi]).field1.ax
```

9. 82
10. 82
11. TYPE MyStruct.field2 (or: SIZEOF Mystruct.field2)
12. Multiple answers:

    a. yes

    b. no

    c. yes

    d. yes

    e. no
13. Code example:

```
.data
time SYSTEMTIME <>
.code
mov ax,time.wHour
```

14. Code example:

```
myShape Triangle < <0,0>, <5,0>, <7,6> >
```

15. Code example (initializes an array of Triangle structures):

```
.data
ARRAY_SIZE = 5
triangles Triangle ARRAY_SIZE DUP(<>)
.code
      mov  ecx,ARRAY_SIZE
      mov  esi,0
L1:   mov  eax,11
      call RandomRange
      mov triangles[esi].Vertex1.X, ax
      mov  eax,11
      call RandomRange
      mov triangles[esi].Vertex1.Y, ax
      add esi,TYPE Triangle
      loop L1
```

10.2 Macros

1. False
2. True
3. Macros can have parameters
4. False
5. True
6. False
7. To permit the use of labels in a macro that is invoked more than once by the same program.
8. ECHO (also, the %OUT operator, which is shown later in the chapter)
9. Code example:

```
mPrintChar MACRO char,count
LOCAL temp
.data
temp BYTE count DUP(&char),0
.code
    push  edx
    mov   edx,OFFSET temp
    call  WriteString
    pop   edx
ENDM
```

10. Code example:

```
mGenRandom MACRO n
```

```
        mov  eax,n
        call RandomRange
ENDM
```

11. mPromptInteger:

```
mPromptInteger MACRO prompt,returnVal
     mWriteprompt
     call  ReadInt
     mov   returnVal,eax
ENDM
```

12. Code example:

```
mWriteAt MACRO X,Y,literal
        mGotoxy X,Y
        mWrite literal
ENDM
```

13. Code example:

```
mWriteStr namePrompt
1       push edx
1       mov edx,OFFSET namePrompt
1       call WriteString
1       pop edx
```

14. Code example:

```
mReadStr customerName
1       push ecx
1       push edx
1       mov edx,OFFSET customerName
1       mov ecx,(SIZEOF customerName) - 1
1       call ReadString
1       pop edx
1       pop ecx
```

15. Code example:

```
;------------------------------------------------
mDumpMemx MACRO varName
;
; Displays a variable in hexadecimal, using the
; variable's attributes to determine the number
; of units and unit size.
;------------------------------------------------
        push ebx
        push ecx
        push esi
        mov  esi,OFFSET varName
        mov  ecx,LENGTHOF varName
        mov  ebx,TYPE varName
```

```
        call DumpMem
        pop  esi
        pop  ecx
        pop  ebx
ENDM
; Sample calls:

.data
array1 BYTE  10h,20h,30h,40h,50h
array2 WORD  10h,20h,30h,40h,50h
array3 DWORD 10h,20h,30h,40h,50h
.code
mDumpMemx array1
mDumpMemx array2
mDumpMemx array3
```

10.3 Conditional-Assembly Directives

1. The IFB directive is used to check for blank macro parameters.
2. The IFIDN directive compares two text values and returns true if they are identical. It performs a case-sensitive comparison.
3. EXITM
4. IFIDNI is the case-insensitive version of IFIDN.
5. The IFDEF returns true if a symbol has already been defined.
6. ENDIF
7. Code example:

```
mWriteLn MACRO text:=<" ">
        mWrite text
        call Crlf
ENDM
```

8. List of relational operators:

   LT Less than

   GT Greater than

   EQ Equal to

   NE Not equal to

   LE Less than or equal to

   GE Greater than or equal to

9. Code example:

```
mCopyWord MACRO intVal
```

```
        IF (TYPE intVal) EQ 2
          mov ax,intVal
        ELSE
          ECHO Invalid operand size
        ENDIF
ENDM
```

10. Code example:

```
mCheck MACRO Z
        IF Z LT 0
          ECHO **** Operand Z is invalid ****
        ENDIF
ENDM
```

11. The substitution (&) operator resolves ambiguous references to parameter names within a macro.
12. The literal-character operator (!) forces the preprocessor to treat a predefined operator as an ordinary character.
13. The expansion operator (%) expands text macros or converts constant expressions into their text representations.
14. Code example:

```
CreateString MACRO strVal
.data
temp BYTE "Var&strVal",0
.code
ENDM
```

15. Code example:

```
mLocate -2,20
;(no code generated because xval < 0)

mLocate 10,20
1  mov bx,0
1  mov ah,2
1  mov dh,20
1  mov dl,10
1  int 10h

mLocate col,row
1  mov bx,0
1  mov ah,2
1  mov dh,row
1  mov dl,col
1  int 10h
```

10.4 Defining Repeat Blocks

1. The WHILE directive repeats a statement block based on a boolean expression.
2. The REPEAT directive repeats a statement block based on the value of a counter.
3. The FOR directive repeats a statement block by iterating over a list of symbols.
4. The FORC directive repeats a statement block by iterating over a string of characters.
5. FORC
6. Code example:

```
      BYTE 0,0,0,100
      BYTE 0,0,0,20
      BYTE 0,0,0,30
```

7. Code example:

```
mRepeat MACRO 'X',50
      mov cx,50
??0000: mov ah,2
      mov dl,'X'
      int 21h
      loop ??0000

mRepeat MACRO AL,20
      mov cx,20
??0001: mov ah,2
      mov dl,AL
      int 21h
      loop ??0001

mRepeat MACRO byteVal,countVal
      mov cx,countVal
??0002: mov ah,2
      mov dl,byteVal
      int 21h
      loop ??0002
```

8. If we examine the linked list data (in the listing file), it is apparent that the **NextPtr** field of each **ListNode** always equals 00000008 (the address of the second node):

```
Offset    ListNode
-----------------------------
00000000  00000001  NodeData
          00000008  NextPtr

00000008  00000002  NodeData
          00000008  NextPtr
```

```
00000010  00000003  NodeData
          00000008  NextPtr

00000018  00000004  NodeData
          00000008  NextPtr

00000020  00000005  NodeData
          00000008  NextPtr

00000028  00000006  NodeData
          00000008  NextPtr
```

We hinted at this in the text when we said: "the location counter's value ($) remains fixed at the first node of the list."

11 32-Bit Windows Programming

11.1 Win32 Console Programming

1. /SUBSYSTEM:CONSOLE
2. True
3. False
4. False
5. True
6. BOOL = byte, COLORREF = DWORD, HANDLE = DWORD, LPSTR = PTR BYTE, WPARAM = DWORD
7. GetStdHandle
8. ReadConsole
9. Example from the *ReadConsole.asm* program in Section 11.1.3.1:

```
INVOKE ReadConsole, stdInHandle, ADDR buffer,
      BufSize - 2, ADDR bytesRead, 0
```

10. The COORD structure contains X and Y screen coordinates in character measurements.
11. Example from the *Console1.asm* program in Section 11.1.4.3:

```
INVOKE WriteConsole,
      consoleHandle,            ; console output handle
      ADDR message,             ; string pointer
      messageSize,              ; string length
      ADDR bytesWritten,        ; returns num bytes written
      0                         ; not used
```

12. Calling **CreateFile** when reading an input file:

```
INVOKE CreateFile,
      ADDR filename,            ; ptr to filename
      GENERIC_READ,             ; access mode
      DO_NOT_SHARE,             ; share mode
      NULL,                     ; ptr to security attributes
      OPEN_EXISTING,            ; file creation options
      FILE_ATTRIBUTE_NORMAL,    ; file attributes
      0                         ; handle to template file
```

13. Calling **CreateFile** to create a new file:

```
INVOKE CreateFile,
      ADDR filename,
      GENERIC_WRITE,
      DO_NOT_SHARE,
      NULL,
      CREATE_ALWAYS,
      FILE_ATTRIBUTE_NORMAL,
      0
```

14. Calling **ReadFile**:

```
INVOKE ReadFile,                ; read file into buffer
      fileHandle,
      ADDR buffer,
      bufSize,
      ADDR byteCount,
      0
```

15. Calling **WriteFile**:

```
INVOKE WriteFile,               ; write text to file
      fileHandle,               ; file handle
      ADDR buffer,              ; buffer pointer
      bufSize,                  ; number of bytes to write
      ADDR bytesWritten,        ; number of bytes written
      0                         ; overlapped execution flag
```

16. SetFilePointer
17. SetConsoleTitle
18. SetConsoleScreenBufferSize
19. SetConsoleCursorInfo
20. SetConsoleTextAttribute
21. WriteConsoleOutputAttribute
22. Sleep

11.2 Writing a Graphical Windows Application

Note: most of these questions can be answered by looking in *GraphWin.inc*, the include file supplied

with MASM in the INCLUDE subdirectory.

1. A POINT structure contains two fields, ptX and ptY, that describe the X and Y coordinates (in pixels) of a point on the screen.
2. The WNDCLASS structure defines a window class. Each window in a program must belong to a class, and each program must define a window class for its main window. This class is registered with the operating system before the main window can be shown.
3. *lpfnWndProc* is a pointer to a function in an application program that receives and processes event messages triggered by the user.
4. The *style* field is a combination of different style options, such as WS_CAPTION and WS_BORDER, that control a window's appearance and behavior.
5. *hInstance* holds a handle to the current program instance. Each programming running under MS-Windows is automatically assigned a handle by the operating system when the program is loaded into memory.
6. *(This should be a challenge question, because it was not explained in Section 11.2.)*

   The prototype for **CreateWindowEx** is located in the *GraphWin.inc* file:

```
CreateWindowEx PROTO,
      classexWinStyle:DWORD,
      className:PTR BYTE,
      winName:PTR BYTE,
      winStyle:DWORD,
      X:DWORD,
      Y:DWORD,
      rWidth:DWORD,
      rHeight:DWORD,
      hWndParent:DWORD,
      hMenu:DWORD,
      hInstance:DWORD,
      lpParam:DWORD
```

The fourth parameter, *winStyle*, determines the window's style characteristics. In the WinApp.asm program in Section 11.2.6, when we call CreateWindowEx, we pass it a combination of predefined style constants:

```
MAIN_WINDOW_STYLE = WS_VISIBLE + WS_DLGFRAME + WS_CAPTION
      + WS_BORDER + WS_SYSMENU + WS_MAXIMIZEBOX + WS_MINIMIZEBOX
      + WS_THICKFRAME
```

The window described here will be visible, it will have a dialog box frame, a caption bar, a border, a system menu, a maximize icon, a minimize icon, and a thick surrounding frame.

7. Calling MessageBox:

```
INVOKE MessageBox, hMainWnd, ADDR GreetText,
      ADDR GreetTitle, MB_OK
```

8. Choose any two of the following (from *GraphWin.inc*):

```
MB_OK, MB_OKCANCEL, MB_ABORTRETRYIGNORE, MB_YESNOCANCEL, MB_YESNO,
MB_RETRYCANCEL, MB_CANCELTRYCONTINUE
```

9. Icon constants (choose any two):

```
MB_ICONHAND, MB_ICONQUESTION, MB_ICONEXCLAMATION, MB_ICONASTERISK
```

10. Tasks performed by **WinMain** (choose any three):

- Get a handle to the current program.
- Load the program's icon and mouse cursor.
- Register the program's main window class and identify the procedure that will process event messages for the window.
- Create the main window.
- Show and update the main window.
- Begin a loop that receives and dispatches messages.

11. The **WinProc** procedure receives and processes all event messages relating to a window. It decodes each message, and if the message is recognized, carries out application-oriented (or application-specific) tasks relating to the message.

12. The following messages are processed:

- WM_LBUTTONDOWN, generated when the user presses the left mouse button
- WM_CREATE, indicates that the main window was just created
- WM_CLOSE, indicates that the application's main window is about to close

13. The ErrorHandler procedure, which is optional, is called if the system reports an error during the registration and creation of the program's main window.

14. The message box is shown before the application's main window appears.

15. The message box appears before the main window closes.

11.3 Dynamic Memory Allocation

1. dynamic memory allocation

2. Returns a 32-bit integer handle to the program's existing heap area in EAX.

3. Allocates a block of memory from a heap.

4. HeapCreate example:

```
HEAP_START =   2000000          ;   2 MB
HEAP_MAX  =  400000000          ; 400 MB
.data
hHeap HANDLE ?; handle to heap
.code
INVOKE HeapCreate, 0, HEAP_START, HEAP_MAX
```

5. Pass a pointer to the memory block (along with the heap handle).

11.4 IA-32 Memory Management

1. (a) Multitasking permits multiple programs (or tasks) to run at the same time. The processor divides up its time between all of the running programs.

   (b) Segmentation provides a way to isolate memory segments from each other. This permits multiple programs to run simultaneously without interfering with each other.

2. (a) A segment selector is a 16-bit value stored in a segment register (CS, DS, SS, ES, FS, or GS).

   (b) A logical address is a combination of a segment selector and a 32-bit offset.

3. True
4. True
5. False
6. False
7. A linear address is a 32-bit integer ranging between 0 and FFFFFFFFh, which refers to a memory location. The linear address may also be the physical address of the target data, if a feature called paging is disabled.
8. When paging is enabled, the processor translates each 32-bit linear address into a 32-bit physical address. A linear address is divided into three fields: a pointer to a page directory entry, a pointer to a page table entry, and an offset into a page frame.
9. The linear address is automatically a 32-bit physical memory address.
10. Paging makes it possible for a computer to run a combination of programs that would not otherwise fit into memory. The processor does this by initially loading only part of a program in memory, while the remaining parts are kept on disk.
11. The LDTR register
12. The GDTR register
13. One
14. Many (each task or program has its own local descriptor table)
15. Choose any four from the following list: Base address, privilege level, segment type, segment present flag, granularity flag, segment limit.
16. Page Directory, Page Table, and Page (page frame).
17. The Table field of a linear address (see Figure 11-4).
18. The Offset field of a linear address (see Figure 11-4).

12 High-Level Language Interface

12.1 Introduction

1. The naming convention used by a language refers to the rules or characteristics regarding the naming of variables and procedures.
2. Tiny, small, compact, medium, large, huge
3. No, because the procedure name will not be found by the linker.
4. The memory model determines whether near or far calls are made. A near call pushes only the 16-bit offset of the return address on the stack. A far call pushes a 32-bit segment/offset address on the stack.
5. C and C++ are case-sensitive, so they will only execute calls to procedures that are named in the same fashion.
6. Yes, many languages specify that EBP (BP), ESI (SI), and EDI (DI) must be preserved across procedure calls.

12.2 Inline Assembly Code

1. Inline assembly code is assembly language source code that is inserted directly into high-level language programs. The inline qualifier in C++, on the other hand, asks the C++ compiler to insert the body of a function directly into the program's compiled code, to avoid the extra execution time it would take to call and return from the function. (Note: answsering this question requires some knowledge of the C++ language, that is not found in the current book.)
2. The primary advantage to writing inline code is simplicity, because there are no external linking issues, naming problems, and parameter passing protocols to worry about. Secondarily, inline code can execute more quickly because it avoids the extra execution time typically required by calling and returning from an assembly language procedure.
3. Examples of comments (select any two):

```
mov esi,buf              ; initialize index register
mov esi,buf              // initialize index register
mov esi,buf              /* initialize index register */
```

4. Yes
5. Yes
6. No
7. No
8. A program bug might result, because the __fastcall convention allows the compiler to use general-purpose registers as temporary variables.
9. Use the LEA instruction.

10. The LENGTH operator returns the number of elements in the array, specified by the DUP operator. For example, the value placed in EAX by the LENGTH operator is 20:

```
myArray DWORD 20 DUP(?), 10, 20, 30
.code
mov eax,LENGTH myArray                  ; 20
```

(Note that the LENGTHOF operator, introduced in Chapter 4, would return 23 when applied to myArray.)

11. The SIZE operator returns the product of TYPE (4) * LENGTH.

12.3 Linking to C++ Programs

1. X will be pushed last.
2. To prevent the the decoration (altering) of external procedure names by the C++ compiler. *Name decoration* (also called *name mangling*) is done by programming languages that permit function overloading, which permits multiple functions to have the same name.
3. If name decoration is in effect, an external function name generated by the C++ compiler will not be the same as the name of the called procedure written in assembly language. Understandably, the assembler does not have any knowledge of the name decoration rules used by C++ compilers.
4. Assembly procedures called by Borland C++ must preserve the values of BP, DS, SS, SI, DI, and the Direction flag.
5. INT = 2, enum = 1, float = 4, double = 8.
6. mov eax,[bp + 6]
7. What SHLD instruction? Actually, there was a line in the LongRandom code originally, that read:

```
shld edx,eax,16
```

So using that instruction as the basis for the question, we can say that the equivalent statements would be:

```
mov ecx,16
L1:   shl eax,1
      rcl edx,1
      Loop L1
```

The current version of this procedure uses the follwoign statement to rotate out the lowest digit of EAX, which prevents a recurring pattern when generating sequences of small random numbers:

```
      ror eax,8
```

8. Virtually no changes at all, showing that array subscripts can be just as efficient as pointers when manipulating arrays.

12.4 Calling C++ Functions

1. The extern and "C" keywords must be used.
2. The Irvine32 library uses STDCALL, which is not the same as the C calling convention used by C and C++. The important difference is in how the stack is cleaned up after a function call.
3. Floating-point values are usually pushed on the processor's floating-point stack before returning from the function.
4. A short int is returned in the AX register.
5. printf PROTO C, pString:PTR BYTE, args:VARARG

13 16-Bit MS-DOS Programming

13.1 MS-DOS and the IBM-PC

1. 9FFFFh
2. Interrupt vector table
3. 00400h
4. The BIOS
5. Suppose a program was named myProg.exe. The following would redirect its output to the default printer:

```
myProg > prn
```

6. LPT1
7. An interrupt service routine (also called an *interrupt handler*) is an operating system procedure that (1) provides basic services to application programs, and (2) handles hardware events. For more details, see Section 16.4.
8. Push the flags on the stack.
9. See the 4 steps in Section 13.1.4.1.
10. The interrupt handler executes an IRET instruction.
11. 10h
12. 1Ah
13. 21h * 4 = 0084h

13.2 MS-DOS Function Calls (INT 21h)

1. AH
2. Function 4Ch

3. Functions 2 and 6 both write a single character.
4. Function 9
5. Function 40h
6. Functions 1 and 6
7. Function 3Fh
8. Functions 2Ah and 2Bh. To display the time, you would call the **WriteDec** procedure from the book's library. That procedure uses Function 2 to output digits to the console. (Look in the *Irvine16.asm* file for details, located in the \Examples\Lib16 directory.)
9. Functions 2Bh (set system date) and 2Dh (set system time).
10. Function 6

13.3 Standard MS-DOS File I/O Services

1. Device Handles: 0 = Keyboard (standard input), 1 = Console (standard output), 2 = Error output, 3 = Auxiliary device (asynchronous), 4 = Printer
2. Carry flag
3. Parameters for function 716Ch

```
AX = 716Ch
BX = access mode (0 = read, 1 = write, 2 = read/write)
CX = attributes (0 = normal, 1 = read only, 2 = hidden,
 3 = system, 8 = volume ID, 20h = archive)
DX = action (1 = open, 2 = truncate, 10h = create)
DS:SI = segment/offset of filename
DI = alias hint (optional)
```

4. Opening an existing file for input:

```
.data
infile BYTE "myfile.txt",0
inHandle WORD ?
.code
      mov ax,716Ch             ; extended create or open
      mov bx,0                 ; mode = read-only
      mov cx,0                 ; normal attribute
      mov dx,1                 ; action: open
      mov si,OFFSET infile
      int 21h                  ; call MS-DOS
      jc quit                  ; quit if error
      mov inHandle,ax
```

5. Reading a binary array from a file is best done with INT 21h Function 3Fh. The following parameters are required

```
AH = 3Fh
BX = open file handle
```

```
CX = maximum bytes to read
DS:DX = address of input buffer
```

6. After calling INT 21h, compare the return value in AX to the value that was placed in CX before the function call. If AX is smaller, the end of the file must have been reached.

7. The only difference is the value in BX. When reading from the keyboard, BX is set to the keyboard handle (0). When reading from a file, BX is set to the handle of the open file.

8. Function 42h

9. Code example (BX already contains the file handle):

```
mov ah,42h                      ; move file pointer
mov al,0                        ; method: offset from beginning
mov cx,0                        ; offsetHi
mov dx,50                       ; offsetLo
int 21h
```

14 Disk Fundamentals

14.1 Disk Storage Systems

1. True
2. False
3. Cylinder
4. True
5. 512
6. For faster access, because the closer gother the cylinders, the smaller distance the read/write heads must travel.
7. The read/write heads must jump over other cylinders, wasting time and increasing the probablility that errors will occur.
8. Volume
9. The average amount of time required to move the read/write heads between tracks.
10. The marking of physical sectors on the disk surfaces.
11. The disk partition table, and a program that locates a single partition's boot sector and runs another program that loads the operating system.
12. One
13. System

14.2 File Systems

1. True
2. No, it is in the disk directory
3. False - all systems, including NTFS, require at least one cluster to store a file
4. False
5. False
6. 4 GB (shown in Table 14-1)
7. FAT32 and NTFS
8. NTFS
9. NTFS
10. NTFS
11. NTFS
12. 8 GB
13. Boot record, file allocation table, root directory, and the data area.
14. This information is at offset 0Dh in the boot record.
15. Two 8 KB clusters would be required, for a total of 16,384 bytes. The number of wasted bytes would be (16,384 – 8,200), or 8,184 bytes.
16. (This one's up to you!)

14.3 Disk Directory

1. True
2. False - (it is called the root directory)
3. False - (it contains the starting *cluster* number)
4. True
5. 32
6. Filename, extension, attribute, time stamp, date stamp, starting cluster number, file size.
7. The status bytes and their descriptions are listed in Table 14-5.
8. Bits 0-4 = seconds; bits 5-10 = minutes; bits 11-15 = hours.
9. The first byte of the entry is 4*x*h, where x indicates the number of long filename entries to be used for the file.
10. Two
11. Actually, there are three new fields: Last access date, create date, and create time

12. File allocation table links:

	3	7	8		4	6	eof								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

14.4 Reading and Writing Disk Sectors (7305h)

1. True
2. False - the function runs in Real-address mode.
3. Parameters:

```
AX: 7305h
DS:BX: Segment/offset of a DISKIO structure variable
CX: 0FFFFh
DL: Drive number (0 = default, 1 = A, 2 = B, 3 = C, etc.)
SI: Read/write flag
```

4. INT 10h displays special ASCII graphics characters without trying to interpret them as control codes (such as Tab and Carriage return).
5. The Carry flag is set if function 7305h cannot read the requested sector, and the program displays an error message. (Remember that you cannot test this program under Windows NT, 2000, or XP.)

14.5 System-Level File Functions

1. Function 7303h
2. Function 7303h
3. Function 39h (create subdirectory) and Function 3Bh (set current directory).
4. Function 7143h (get and set file attributes).

15 BIOS-Level Programming

15.1 Introduction

(no review questions)

15.2 Keyboard Input with INT 16h

1. INT 16h is best
2. In the keyboard typeahead buffer, at location 0040:001E.
3. INT 9h reads the keyboard input port, retrieves the keyboard scan code and produces the corresponding ASCII code. It inserts both in the keyboard typeahead buffer.

4. Function 05h.
5. Function 10h
6. Function 11h examines the buffer and lets you know which key, if any, is waiting.
7. No
8. Function 12h
9. Bit 4 (see Table 15-2)
10. Code example:

```
L1:    mov ah,12h                ; get keyboard flags
       int 16h
       test all,100h             ; Ctrl key down?
       jz L1                     ; no: repeat the loop
; At this point, the Ctrl key has been pressed
```

11. To check for other keyboard keys, add more CMP and JE instructions after the existing ones currently in the loop. Suppose we wanted to check for the ESC, F1, and Home keys:

```
L1:    .
       .
       cmp ah,1                  ; ESC key's scan code?
       je  quit                  ; yes: quit
       cmp ah,3Bh                ; F1 function key?
       je  quit                  ; yes: quit
       cmp ah,47h                ; Home key?
       je  quit                  ; yes: quit
       jmp L1                    ; no: check buffer again
```

15.3 Video Programming with INT 10h

1. MS-DOS level, BIOS level, and Direct video level.
2. Direct video
3. In MS-Windows, there are two ways to switch into full-screen mode:

- Create a shortcut to the program's EXE file. Then open the Properties dialog for the shortcut, select the Screen properties, and select Full-screen mode.
- Open a Command window from the Start menu, then press Alt-Enter to switch to full screen mode. Using the CD (change directory) command, navigate to your EXE file's directory, and run the program by typing its name. Alt-Enter is a toggle, so if you press it again, it will return the program to Window mode.

4. Mode 3 (color, 80 x 25)
5. ASCII code and attribute (2 bytes)
6. Red, green, blue, and intensity
7. Background: bits 4-7. Foreground: bits 0-3.

8. Function 02h
9. Function 06h
10. Function 09h
11. Function 01h
12. Function 00h
13. AH = 2, DH = row, DL = column, and BH = video page
14. There are two ways: (1) use INT 10h Function 01h to set the cursor's top line to an illegal value, or (2) use INT 10h Function 02h to position the cursor outside the displayable range of rows and columns.
15. AH = 6, AL = number of lines to scroll, BH = attribute of scrolled lines, CH & CL = upper-left window corner, and DH & DL = lower right window corner.
16. AH = 9, AL = ASCII code of character, BH = video page, BL = attribute, and CX = repetition count.
17. Function 10h, Subfunction 03h (set AH to 10h, and AL to 03h)
18. AH = 06h, and AL = 0
19. Every pixel on the screen is made of three colors: red, green, and blue. Dogs are color blind, so they cannot see pixels made from colors. I've tried displaying a picture of a cat on the screen, but my own dog seems not to notice.

15.4 Drawing Graphics Using INT 10h

1. Function 0Ch
2. AH = 0Ch, AL = pixel value, BH = video page, CX = x-coordinate, and DX = y-coordinate.
3. It's very slow.
4. Code example:

```
mov ah,0                ; set video mode
mov al,11h              ; to mode 11h
int 10h                 ; call the BIOS
```

5. Mode 6Ah
6. Formula: sx = (sOrigX + X)
7. a. (350,150) b. (375,225) c. (150,400)

15.5 Memory-Mapped Graphics

1. False - (each byte corresponds to 1 pixel)
2. True
3. Mode 13h maps each pixel's integer value into a table of colors called a palette.

4. The color index of a pixel identifies which color in the palette is to be used when drawing the pixel on the screen.
5. Each entry in the palette consists of three separate integer values (0–63) known as RGB (red, green, blue). Entry 0 in the color palette controls the screen's background color.
6. (20,20,20)
7. (63,63,63)
8. (63,0,0)
9. Code example:

```
; Set screen background color to bright green.
mov dx,3c8h                     ; video paletter port
mov al,0                        ; index 0 (background color)
out dx,al
mov dx,3c9h                     ; colors go to port 3C9h
mov al,0                        ; red
out dx,al
mov al,63                       ; green (intensity = 63)
out dx,al
mov al,0                        ; blue
out dx,al
```

10. Code example:

```
; Set screen background color to white
mov dx,3c8h                     ; video paletter port
mov al,0                        ; index 0 (background color)
out dx,al
mov dx,3c9h                     ; colors go to port 3C9h
mov al,63                       ; red = 63
out dx,al
mov al,63                       ; green = 63
out dx,al
mov al,63                       ; blue = 63
out dx,al
```

(The last two MOV statements can be eliminated if you want to reduce the amount of code in this example.)

15.6 Mouse Programming

1. Function 0
2. Code example:

```
      mov ax,0                  ; reset mouse
      int 33h                   ; call the BIOS
      cmp ax,0                  ; mouse not available?
      je MouseNotAvailable      ; yes: show error message
```

3. Functions 1 and 2
4. Code example:

```
mov ax,2                    ; hide mouse pointer
int 33h
```

5. Function 3
6. Code example:

```
mov ax,3                    ; get mouse position and status
int 33h
mov mouseX,cx
mov mouseY,dx
```

7. Function 4
8. Code example:

```
mov ax,4                    ; set mouse position
mov cx,100                  ; X-value
mov dx,400                  ; Y-value
int 33h
```

9. Function 5
10. Code example:

```
mov ax,5                    ; get button press information
mov bx,0                    ; button ID for left button
int 33h
test ax,1                   ; left button currently down?
jne Button1                 ; yes: jump to label
```

Implementation note: This function will tell you if a certain button is currently being pressed. But if you just want the coordinates of the last button press, there is no need for the TEST instruction used in our example.

11. Function 6
12. Code example:

```
      mov ax,6              ; get button release information
      mov bx,1              ; button ID
      int 33h
      test ax,2             ; right button released?
      jz skip               ; no - skip
      mov mouseX,cx         ; yes: save coordinates
      mov mouseY,dx
skip:
```

13. Code example:

```
mov ax,8                    ; set vertical limits
mov cx,200                  ; lower limit
```

```
mov dx,400                    ; upper limit
int 33h
```

Note: in the first printing of the book, the box that describes Function 8 had a few errors. AX must be set to 8, and INT 33h should be called *after* CX and DX have been set.

14. Code example:

```
mov ax,7                      ; set horizontal limits
mov cx,300                    ; lower limit
mov dx,600                    ; upper limit
int 33h
```

15. Assuming that character cells are 8 pixels by 8 pixels, the X, Y coordinates values would be (8 * 20), (8 * 10). The cell will be at position 160, 80.
16. The upper-left corner of the cell will be at (8 * 22), (8 * 15). If we add 4 to each of these values to bring the mouse to the center of the cell, the answer is: 180, 124.
17. The mouse was invented by Douglas Engelbart in 1963, at the Stanford Research Institute.(Source: *http://en.wikipedia.org/wiki/Computer_mouse*).

16 Expert MS-DOS Programming

16.1 Introduction

(no review questions)

16.2 Defining Segments

1. Declares the beginning of a segment.
2. Returns the segment address of a data label or code label.
3. The ASSUME directive makes it possible for the assembler to calculate the offsets of labels and variables at assembly time. A directive such as:

```
assume DS:myData
```

says to the assembler, "assume that from this point on, all references to data labels (via DS) will be located in the segment named **myData**."

4. BYTE, WORD, DWORD, PARA, and PAGE
5. PRIVATE, PUBLIC, MEMORY, STACK, COMMON, and AT
6. DWORD
7. The combine type tells the linker how to combine segments having the same name.
8. Use the AT combine type. The following defines a segment with value 0040h:

```
bios SEGMENT AT 40h
```

9. A segment's class type provides another way of combining segments, in particular, those with different names. Segments having the same class type are loaded together, although they may be listed in a different order in the program source code.
10. Code example:

```
mov al,es:[di]
```

11. The third segment will also begin at address 1A060h.

16.3 Runtime Program Structure

1. The command processor checks to see if there is filename with extension COM in the current directory. If a file is found, it is executed. If a matching file is not found, see Section 16.3 for a description of the subsequent steps.
2. No.
3. Application programs loaded into the lowest 640K of memory. They are transient because when the finish executing, they are automatically unloaded from memory.
4. Program segment prefix
5. At offset 2Ch inside the program segment prefix area.
6. A COM program is a single-segment MS-DOS program. When stored on disk as a COM file, it is simply a binary image of the program when loaded into memory.
7. Tiny
8. /T
9. 64 Kilobytes
10. Not efficient, because even the smallest COM program uses an entire 64 K memory segment.
11. One
12. All segment registers are set to offset zero within the program. The program, in turn, is loaded into memory at the first available segment location following other programs still in memory.
13. The ORG directive assigns a specific offset to the very next label or instruction following the directive. The addresses of all subsequent labels are calculated from that point onward. COM programs, for example, always have ORG 100h at the beginning of the program code, so the first executable instruction will be located at offset 100h.
14. Load module
15. DS and ES point to the program segment prefix area of the program.
16. MS-DOS automatically allocates all of available memory to a program when it is first loaded, unless the program's EXE header specificically limits its maximum memory allocation size.
17. The EXEMOD program displays statistics about a program's memory usage, and also permits many settings in the EXE header to be modified.
18. Run the EXEMOD program, passing it the name of the EXE file. The last line of the display will show the number of relocation entries.

16.4 Interrupt Handling

1. It displays a message on the screen "Abort, retry, or ignore?", and terminates the current program.
2. A 32-bit segment/offset address pointing to an interrupt handler
3. At address 0000:0040h, because 0040h equals 10h * 4
4. The 8259 Programmable Interrupt Controller chip
5. The CLI (clear interrupt flag) instruction
6. The STI (set interrupt flag) instruction
7. IRQ 0 has highest priority
8. Before the file has been created, because the keyboard (IRQ 1) is at a higher priority than the disk drive (IRQ 14).
9. INT 9h
10. An IRET instruction at the end of the interrupt handler returns control to the code that was running when the interrupt occurred.
11. Functions 25h and 35h
12. An interrupt handler is any procedure that takes over processing an interrupt. It might be loaded when an application starts, and then be unloaded when the application ends. A memory-resident program, on the other hand, remains in memory even after the program that installed it has ended. A memory-resident program does not necessarily have to be an interrupt handler.
13. A terminate and stay resident (TSR) program leaves part of itself in memory when it exits. This is accomplished by calling INT 21h function 31h.
14. The computer can be rebooted, or a special utility program can remove the TSR.
15. Rather than executing an IRET instruction when it finishes, it can instead execute a JMP to the address that was previously stored in the interrupt vector.
16. A terminate and stay resident (TSR) program.
17. Ctrl + Alt + Right shift + Del

17 Floating-Point Processing and Instruction Encoding

17.1 Floating-Point Binary Representation (ANSWERS PENDING)

1. Because the reciprocal of –127 is +127, which would generate an overflow.
2. Because adding +128 to the exponent bias (127) would generate a negative value.
3. 52 bits
4. 8 bits
5. 1101.01101 = 13/1 + 1/4 + 1/8 + 1/32

6. 0.2 generates an infinitely repeating bit pattern.
7. 11011.01011 = 1.101101011 x 2^4
8. 0000100111101.1 = 1.001111011 x 2^{-8}
9. +1110.011 = 1.110011 x 2^{-3}, so the encoding is 0 01111100 11001100000000000000000
10. Quiet NaN and Signaling NaN.
11. 5/8 = 0.101 binary.
12. 17/32 = 0.10001 binary
13. +10.75 = +1010.11 = +1.01011 x 2^3, encoded as: 0 10000010 01011000000000000000000
14. –76.0625 = –01001100.0001 = –1.0011000001 x 2^{-6}, encoded as:
    1 10000101 00110000010000000000000
15. Positive or negative infinity, depending on the sign of the numerator.

17.2 Floating-Point Unit

1. fld st(0)
2. R0
3. Choose from opcode, control, status, tag word, last instruction pointer, last data pointer.
4. Binary-coded decimal
5. None
6. REAL10 -- 80 bits
7. It pops ST(0) off the stack.
8. FCHS
9. None, m32fp, m64fp, stack register.
10. FISUB converts the source operand from integer to floating-point.
11. FCOM, or FCOMP.
12. Code example:

```
fnstsw  ax
lahf
```

13. FILD
14. RC field
15. 1.010101101 rounded to nearest even becomes 1.010101110.
16. –1.010101101 rounded to nearest even becomes –1.010101110.
17. Assembly instructions:

```
.data
B REAL8 7.8
```

```
M REAL8 3.6
N REAL8 7.1
P REAL8 ?
.code
fld  M
fchs
fld  N
fadd B
fmul
fst  P
```

18. Assembly language code:

```
.data
B DWORD 7
N REAL8 7.1
P REAL8 ?
.code
fld  N
fsqrt
fiadd B
fst  P
```

17.3 Intel Instruction Encoding

1. (a) 8E (b) 8B (c) 8A (d) 8A (e) A2 (f) A3
2. (a) 8E (b) 8A (c) 8A (d) 8B (e) A0 (f) 8B
3. (a) D8 (b) D3 (c) 1D (d) 44 (e) 84 (f) 85
4. (a) 06 (b) 56 (c) 1D (d) 55 (e) 84 (f) 81
5. Machine language bytes:

```
a.    8E D8
 b.   A0 00 00
 c.   8B 0E 01 00
 d.   BA 00 00
 e.   B2 02
 f.   BB 00 10
```