

ELEC3710 Microprocessor Systems: Tutorial Problems

1. Give the integer value of the following expressions:

(a) $6 \&\& 2$

Solution:

non zero $\&\&$ non zero = T & T = 1

(b) $6 \& -2$

Solution: $(0\dots0110)_2 \& (1\dots0010)_2 = (0\dots0010)_2 = 2_{10}$

(c) $3 \|\| 6$

Solution:

non-zero $\|\|$ non-zero = T $\|\|$ T = T = 1

(d) $3 | 6$

Solution:

$(0\dots0011)_2 \text{---} (0\dots0110)_2 = (0\dots0111)_2 = 7_{10}$

(e) $!(-3)$

Solution: $!(\text{non-zero}) = !(T) = F = 0$

(f) $\sim(-3)$

Solution:

$\sim(11\dots101)_2 = (00\dots010)_2 = 2_{10}$

2. Write an expression in C that is true if and only if bit 5 of the integer variable “x” is a 1.

Solution:

$(x \& (1 \ll 5)) != 0$

3. Assume that “n” and “x” are declared as `int`. Write a line of C code that regardless of the current value of x, and without modifying any other bits of x will:

- (a) Set bit n of x to 1

Solution:

$$x \mid= (1 \ll n);$$

- (b) Clear bit n of x to 0

Solution:

$$x \&= \sim (1 \ll n);$$

- (c) Invert bit n of x.

Solution:

$$x \hat{=} (1 \ll n);$$

4. Give the 8-bit binary value that remains in the unsigned char after executing the indicated line of code, with the indicated initial value of x.

- (a) $x = (11100101)_2, x \mid= (1 \ll 4)$

Solution:

$$(11110101)_2$$

- (b) $x = (11011001)_2, x \&= \sim (1 \ll 6)$

Solution:

$$(10011001)_2$$

- (c) $x = (01111010)_2, x \hat{=} (1 \ll 5)$

Solution:

$$(01011010)_2$$

- (d) $x = (10101010)_2, x = (x \gg 3) \& 0x0F$

Solution:

$$(00000101)_2$$

- (e) $x = (00001111)_2, x = \sim x$

Solution:

$$(11110000)_2$$

(f) $x = (00001111)_2, x = !x$

Solution:

$(00000000)_2$

(g) $x = (00000000)_2, x \|\ = 0x20$

Solution:

$(00000001)_2$

(h) $x = (11111111)_2, x \&\&=0xF0$

Solution:

$(00000001)_2$

5. Given the macro defined below, show how each of the following usages would be expanded.

```
#define REM(a,b) a % b;
```

(a) $\text{rem}(5, 2);$

Solution: $5 \% 2$

(b) $\text{rem}(5+2, x);$

Solution: $5 + 2 \% 2$

(c) $\text{rem}(5, x-2);$

Solution: $5 \% x - 2$

6. Define a macro called $\text{bit}(x, n)$ that expands into an expression whose value is exactly 0 or 1 corresponding to the value of the n 'th bit of x .

Solution:

```
#define bit(x,n) ((x) & (1<<(n))) !=0 )
```

7. Suppose a packed operand of 16 bits is to use bits 0-7 for a variable called `age`, bits 9-11 for a variable called `hair_color` and bits 13-14 for `gender`. Implement this packed operand using structure bit-fields.

Solution:

```
struct{
unsigned age : 8,
          : 1,
    hair_color : 3,
          : 1,
    gender    : 2;
}
```

8. The first `printf` below prints `1C3F`. What is printed by the second `printf`?

```
union{
short w;
char b[2];
}x;

x.w = 0x1C3F;
printf("%04x\n",x.w);

x.b[1] = 0xA6;
printf("%04x\n",x.w);
```

Solution:A63F

9. Write code to modify the 3rd byte (bits 16-23) of a 32-bit operand called "x" using:

(a) And'ing, or'ing and shifting;

Solution:

```
X &= 0xFF00FFFF
X |= (new_value << 16) & 0x00FF0000;
```

(b) Casts and subscripting;

Solution:

```
((char *) &X)[2] = new_value;
```

(c) A union.

Solution:

```
union{
char b[4];
long l;
}X;

X.b[2] = new_value;
```

10. Write a C function that sets a specified bit within an array of bits that are packed eight per byte into an array of bytes. The function must have exactly two parameters: the first is the name of the array of bytes, the second is the index (starting at zero) of the bit to be set. The function prototype should be

```
void SetBit(unsigned char bits[], int index);
```

Solution:

```
void SetBit(unsigned char bits[], int index);
{
bits[index/8] |= (1<<(index % 8));
}
```

11. Write a sequence of C statements that swaps the contents of the two bytes stored within a 16-bit unsigned short int named "pair".

Solution:

```
unsigned short int pair;
unsigned char temp = pair & 0xFF;

((unsigned char *)&pair)[0] = pair >> 8;
((unsigned char *)&pair)[1] = temp;
```

12. What sequence of digits with the following code print when running on a CPU that uses *little* endian byte numbering?

```
unsigned long int x = 0x01030204;
unsigned char *p = (char *)&x;
int k;
for (k=0;k<4;k++) printf("%d",p[k]);
```

Solution: 1st = 4, 2nd = 2, 3rd = 3, last = 1.

13. Use C's bitwise and shift operators to write a sequence of C assignment statements that increment the middle 6 bits of a 16 bit unsigned short int called "x" without modifying any of its other 10 bits (be careful of overflow).

Solution:

```
temp = (x>>5) & 0x3F;          /* Extract integer in middle 6 bits */
x &= ~(0x3F<<5);              /* Clear middle 6 bits */
x |= ((temp+1) & 0x3F) <<5;   /* restrict increment to 6 bits */
```

14. Define a macro called `BYTE(a,n)` where "a" is to be replaced by the name of a 32-bit integer variable, and "n" is a number between 0 and 3 that selects one of the four 8-bit bytes within "a" so that the macro can be used as follows.

```
long int x;
...
BYTE(x,2)=100;
...
if (BYTE(x,1) == 0) ...
```

Solution:

```
#define BYTE(a,n) ((unsigned char *) &a)[n];
```

15. What changes when register EBP is used within the source or destination memory address field of an instruction in the Intel processor?
- (a) No segment register is used to compute the physical address.
 - (b) Segment register DS is used to compute the physical address.
 - (c) Segment register SS is used to compute the physical address.
 - (d) Segment register CS is used to compute the physical address.

Solution:

Answer = 15c

16. Suppose that register EDI and ESI already hold a single 64-bit quantity with the most significant bits in EDI. Give a minimum length sequence of Intel instructions that would be required to implement:

- (a) A 64-bit logical right shift by one bit position;

Solution:

```
SHR EDI,1
RCR ESI,1
```

- (b) A 64-bit arithmetic right shift by one bit position;

Solution:

```
SAR EDI,1
RCR ESI,1
```

- (c) A 64-bit right rotate by one bit position;

Solution:

```

    SHR EDI,1
    RCR ESI,1
    JNC L1
    OR EDI,80000000h
L1:

```

17. Write a sequence of Intel protected mode instructions that will form the sum of the numbers 1 through 100.

Solution:

```

    MOV DWORD [_sum],0
    MOV ECX,1
L1: ADD [_sum],ECX
    INC ECX
    CMP ECX,100
    JLE L1

```

18. Write a sequence of Intel protected mode instructions that corresponds to the following C code statements where x, y and z are labels on 32-bit memory locations whose contents are unsigned integers.

- (a) if (x<y) z= 6; else z=x;

Solution:

```

    MOV EAX,[_x]
    CMP EAX,[_y]
    JNB L1
    MOV DWORD [_z],6
    JMP L2
L1: MOV EAX,[_x]
    MOV [_z], EAX
L2:

```

- (b) x = (13*y)/3;

Solution:

```

    MOV EAX,13
    MUL DWORD [_y]
    MOV ECX,3
    DIV ECX
    MOV [_x],EAX

```

- (c) x=0; for (y=0;y<10000;y=y<<1) x+=y;

Solution:

```

MOV DWORD [_x],0
MOV DWORD [_y],0
L1: CMP DWORD [_y],1000
    JNB L2
    MOV EAX,[_y]
    ADD [_x],EAX
    SHL DWORD [_y],1
    JMP L1
L2:

```

(d) if (x>10) if (x<20) y=1; else z=0;

Solution:

```

CMP DWORD [_x],10
JNA L2
CMP DWORD [_x],20
JNB L1
MOV DWORD [_y],1
JMP L2
L1: MOV DWORD [_z],0
L2:

```

19. Write a sequence of Intel protected mode instructions that corresponds to the following C code statements where x, y and z are labels on 32-bit memory locations whose contents are signed integers.

(a) if (x<y) && y<z) z=6; else z=x;

Solution:

```

MOV EAX,[_y]
CMP [_x],EAX
JNL L1
CMP EAX,[_z]
JNL L1
MOV DWORD [_z],6
JMP L2
L1: MOV EAX,[_x]
    MOV [_z],EAX
L2:

```

(b) if (-10<x&& x<10) goto L1;

Solution:

```

CMP DWORD [_x],-10
JNG L2

```

```

    CMP DWORD [_x],+10
    JL L1

```

L2:

(c) if (x<10 || x>20) y=0; else y=1;

Solution:

```

    CMP DWORD [_x],10
    JL L1
    CMP DWORD [_x],20
    JNG L2
L1: MOV DWORD [_y],0
    JMP L3
L2: MOV DWORD [_y],1
L3:

```

(d) if 'a' <= ch && ch='z') ch=ch-'a'+'A'

Solution:

```

    CMP [_ch], 'a'
    JNGE L1
    CMP [_ch], 'z'
    JNLE L1
    SUB [_ch], 'a'
    ADD [_ch], 'A'
L1:

```

(e) x=y/5;

Solution:

```

    MOV EAX,[_y]
    CDQ
    MOV ECX,5
    IDIV ECX
    MOV [_x],EAX

```

20. Write a sequence of Intel protected mode instructions required to call the following function as indicated.

(a) int f1(long,long); /* function prototype */
x=f1(a,125); /* function call */

Solution:

```

MOV EAX,125
PUSH EAX
PUSH DWORD [_a]

```

```
CALL _f1
ADD ESP,8
MOV [_x],EAX
```

(b) void f2(char); /* function prototype */
f2(c); /* function call */

Solution:

```
MOVZX EAX,BYTE [_c]
PUSH EAX
CALL _f2
ADD ESP,4
```

(c) void f3(char *); /* function prototype */
f3(&c); /* function call */

Solution:

```
MOV EAX,_c
PUSH EAX
CALL _f3
ADD ESP,4
```

21. Write a sequence of Intel protected mode instructions to implement the following function in assembly. 1

```
int GetAndClear(int *p)
{
int temp *p;
*p = 0;
return temp;
}
```

Solution:

```
_GetAndClear: MOV EDX,[ESP+4]
MOV EAX,[EDX]
MOV DWORD [EDX],0
RET
```

22. Write a sequence of Intel protected mode instructions required to implement the following function in assembly language.

```

typedef unsigned long long int DWORD64

DWORD64 Add(DWORD64 a,DWORD64 b)
{
return a+b;
}

```

Solution:

```

_Add:  MOV  EAX,[ESP+4]
        MOV  EDX,[ESP+8]
        ADD  EAX,[ESP+12]
        ADC  EDX,[ESP+16]
        RET

```

23. Consider the following function prototype:

```

unsigned CountOnes(unsigned long int);

```

Write a sequence of Intel protected mode instructions to implement this function so that it counts and returns the number of bits that are 1 in its argument.

Solution:

```

_CountOnes:  MOV    EAX,0
              MOV    EDX,[ESP+4]
              MOV    ECX,32
L1:          TEST   EDX,1
              JZ     L2
              INC    EAX
L2:          SHR    EDX,1
              DEC    ECX
              JNZ   L1
              RET

```

24. Suppose that you have a 64-bit long long int named x. Give an appropriate sequence of Intel assembly language instructions required to implement the simple C statement x++.

Solution:

```

ADD DWORD [x],1
ADC DWORD [x+4],0

```

25. Given the declaration statements

```
unsigned long int x,y;
unsigned char int z;
```

Write a sequence of Intel protected mode instructions that corresponds to the following statement:

```
z = (y*z)/15;
```

Solution:

```
MOVZX  EAX, BYTE [_z]
MUL    DWORD [_y]
XOR    EDX, EDX
MOV    ECX, 15
DIV    ECX
MOV    DWORD [_x], EAX
```

26. Write a sequence of Intel protected mode instructions that correspond to the following statements where x,y and z are labels on 32-bit memory locations whose contents are signed integers.

(a) if (x>0) z=y; else y=z;

Solution:

```
                CMP  DWORD, [_x], 0
                JNG  else
then: MOV  EAX, [_y]
      MOV  [_z], EAX
      JMP  endif
else: MOV  EAX, [_z]
      MOV  [_y], EAX
endif:
```

(b) if (x>0 && x<100) z=1;

Solution:

```
                CMP  DWORD [_x], 0
                JNG  endif
                CMP  DWORD [_x], 100
                JNL  endif
                MOV  DWORD [_z], 1
endif:
```

(c) if (x<y || x<0) z=2;

Solution:

```

        MOV EAX,[_x]
        CMP EAX,[_y]
        JL  then
        CMP EAX,0
        JNL endif
then:   MOV DWORD [_z],2
endif:

```

27. Write a sequence of Intel protected mode instructions required to call the following function as indicated.

```

void foo(long int*); /* function prototypes */
long int bar(void);

foo(&x); /* function calls */
x=bar();

```

Solution:

```

PUSH  DWORD x
CALL  foo
ADD   ESP,4
CALL  bar
MOV   [x],EAX

```

28. Write a sequence of Intel protected mode instructions to implement the following function in assembly.

```

void Invert(unsigned char *p)
{
*p=~*p;
}

```

Solution:Two possibilities. Either

```

Invert:  PUSH  EBP
         MOV   EBP,ESP
         MOV   EAX,[EBP+8]
         NOT  BYTE [EAX]
         POP  EBP
         RET

```

or (shorter)

```
Invert: MOV  EAX, [ESP+4]
        NOT  BYTE [EAX]
        RET
```

29. Given the declaration statement

```
unsigned short int x,k,*a;
```

Provide a sequence of instructions to implement the following C statement in Intel protected mode assembly language.

```
x = a[k];
```

Solution:

```
MOVZX  EAX, [_k]
MOV     EDX, [_a]
MOV     AX, [EDX+4*EAX]
MOV     [_x], AX
```

30. Suppose that you have a 64-bit fixed point real named x in 32.32 format. Give an appropriate sequence of Intel assembly language instructions required to add 0.5 to x.

Solution:

```
ADD  DWORD [_x], 8000000H
ADC  DWORD [_x4], 0
```

31. Given the declaration statement

```
signed long int x,y;
signed short int z;
```

write a sequence of Intel protected mode instructions that correspond to the following statement

```
x = (y*z)/15;
```

Solution:

```

MOVSBX  EAX,WORD  [_z]
IMUL    DWORD  [_y]
CDQ
MOV     ECX,15
IDIV   ECX
MOV     [_z],AX

```

32. Write a sequence of Intel protected mode instructions that corresponds to the following statements, where x,y and z are labels on 32-bit memory locations whose contents are unsigned long integers.

(a) if (x>0) z=x; else y=x;

Solution:

```

                CMP  DWORD  [_x],0
                JNA  Else
                MOV  EAX,[_x]
                MOV  [_z],EAX
                JMP  Endif
Else:          MOV  EAX,[_x]
                MOV  [_y],EAX
Endif:

```

(b) if (x<0 || x>100) z=1;

Solution:

```

                CMP  DWORD  [_x],0
                JB   Then
                CMP  DWORD  [x],100
                JNA  Endif
Then:          MOV  DWORD  [_z],1
Endif:

```

(c) if (x<y && x<0) z=2;

Solution:

```

                MOV  EAX,[_x]
                CMP  EAX,[_y]
                JNB  Endif
                CMP  DWORD  [_x],0
                JNB  Endif
                MOV  DWORD  [_z],2
Endif:

```

33. Write a sequence of Intel protected mode instructions required to call the following functions

```
void foo(short int*); /* function prototypes */
short int bar(void);
```

```
foo(&x); /* function calls */
x=bar();
```

Solution:

```
PUSH  DWORD x
CALL  foo
ADD   ESP,4
CALL  bar
MOV   [x],AX
```

Now assume that you have executed the first three instructions of function foo:

```
foo:  PUSH  EBP
      MOV   EBP,ESP
      SUB   ESP,8
```

- (a) What is the EBP-relative address expression used to reference the parameter x?

Solution:

[EBP+12]

- (b) How many bytes of temporary storage have been allocated?

Solution:

8

34. Translate the executable statement of each of the following C source code fragments into Intel protected mode assembly:

- (a) long int k, *a;

```
a=&k;
```

Solution:

```
MOV  DWORD [a],k
```

- (b) long int c[10], *a;

```
a=&c[1];
```

Solution:

```
MOV  DWORD [a],c+4
```

(c) signed long int k, *a;

```
k=k/*a;
```

Solution:

```
MOV    EAX,[_k]
CDQ
MOV    ECX,[_a]
IDIV  DWORD [ECX]
MOV    [_k],EAX
```

(d) long long int big64;

```
big64 -= 5;
```

Solution:

```
SUB DWORD [_big64],5
SBB DWORD [_big64+4],0
```

(e)

```
char ch,b[10];
int k;
```

```
ch=b[k];
```

Solution:

```
MOV EAX,[_k]
MOV AL,[_b+EAX]
MOV [_ch],AL
```

35. Given the declaration statement

```
signed long int x,y,z;
```

Translate each of the following C statements into a sequence of Intel protected mode instructions.

(a) if (x>0 && y>0) z+=x; else z=0;

Solution:

```
CMP    DWORD [_x],0
JNG    else
CMP    DWORD [_y],0
JNG    else
```

```

        MOV  EAX,[_x]
        ADD  [_z],EAX
        JMP  done
else:   MOV  DWORD [_z],0
done:

```

(b) if (x>0 || y>0) z+=x; else z=0;

Solution:

```

        CMP  DWORD [_x],0
        JG   then
        CMP  DWORD [_y],0
        JNG  else
then:   MOV  EAX,[_x]
        ADD  [_z],EAX
        JMP  done
else:   MOV  DWORD [_z],0
done:

```

(c) z=0;

for (x=0; x<y,x++) z+= x;

Solution:

```

        MOV  DWORD [_z],0
        MOV  DWORD [_x],0
top:    MOV  EAX,[_x]
        CMP  EAX,[_y]
        JNL  done
        MOV  EAX,[_x]
        ADD  [_z],EAX
        INC  DWORD [_x]
        JMP  top
done:

```

36. Translate the following C function into a sequence of Intel protected mode instructions. Assume that the functions `disable_interrupts` and `enable_interrupts` are library functions, your code merely needs to call them.

```

typedef char BOOL

#define TRUE 1
#define FALSE 0

BOOL GetAndSet(BOOL *semaphore)

```

```

{
BOOL current_value;
disable_interrupts();
current_value=*semaphore;
*semaphore=TRUE;
enable_interrupts();
return current_value;
}

```

Solution:

```

GetAndSet:  PUSH  EBP                ;function prologue
            MOV   EBP,ESP
            SUB   ESP,1            ;allocate 1 byte for current value
            CALL  disable_interrupts
            MOV   EAX,[EBP+8]      ;EAX <- semaphore
            MOV   AL,[EAX]        ;AL , - *semaphore
            MOV   [EBP-1],AL      ;current_value <- *semaphore
            MOV   EAX,[EBP+8]     ;EAX<-sempahore
            MOV   BYTE [EAX],1    ;*semaphore <-true
            CALL  enable_interrupts
            MOVZX EAX,BYTE [EBP-1] ;EAX <- current_value
            MOV   ESP,EBP        ;release byte for current_value
            POP   EBP            ;function epilog
            RET

```

37. Assuming one byte for the opcode itself, what is the minimum number of bytes required to represent each of the following instructions?

(a) INC EAX

Solution:

1

(b) MOV AL,15

Solution:

2

(c) MOV DX,2F8h

Solution:

3

(d) MOV EAX,[result]

Solution:

5

(e) JZ L1

Solution:

2

38. Considering address alignment, what is the worst case time required to fetch a 24-byte data object from memory with a 60 ns cycle time and a data bus that is 64 bits wide?

Solution:

Worst-case time required = 4 cycles \times 60ns = 240 ns.

39. Write a sequence of Intel protected mode instructions (not a function) to exchange the contents of two 32-bit memory locations named X and Y..

(a) Using only PUSH and POP instructions;

Solution:

```
PUSH  DWORD [_x]
PUSH  DWORD [_y]
POP   DWORD [_x]
POP   DWORD [_y]
```

(b) Using the XCHG instruction;

Solution:

```
MOV  EAX, [_x]
XCHG EAX, [_y]
MOV  [_x], EAX
```

(c) Without using the XCHG instruction.

Solution:

```
MOV  EAX, [_x]
MOV  EDX, [_y]
MOV  [_x], EDX
MOV  [_y], EAX
```

40. Given the declaration statement

```
long int k, *a, *b, c[10];
```

translate each of the following C source code statements into Intel protected mode assembly

(a) `a = &k;`

Solution:

```
MOV  DWORD [_a], _k
```

(b) `a = &c[1];`

Solution:

```
MOV  DWORD [_a],_c+4
```

(c) `a = &b[1];`

Solution:

```
MOV  EAX,[_b]
ADD  EAX,4
MOV  [_a],EAX
```

(d) `a[1] = k;`

Solution:

```
MOV  EAX,[_k]
MOV  EDX,[_a]
MOV  [EDX+4],EAX
```

(e) `((char *) c)[k] = 0;`

Solution:

```
MOV  EAX,[_k]
MOV  BYTE [_c+EAX],0
```

41. Given the declaration statement

```
signed long int x,y,z;
signed long int foo(int, int);
```

Translate each of the following C statements into a sequence of Intel protected mode instructions.

(a) `z = (x<0 || y<0);`

Solution:

```
        CMP  DWORD [_x],0
        JL   ONE
        CMP  DWORD [_y],0
        JL   ONE
ZERO:   MOV  DWORD [_z],0
        JMP  DONE
ONE:    MOV  DWORD [_z],1
DONE:
```

(b) `if (x>0 && y>0) z = foo(x,y);`

Solution:

```

    CMP  DWORD [_x],0
    JNG  DONE
    CMP  DWORD [_y],0
    JNG  DONE
    PUSH DWORD [_y]
    PUSH DWORD [_x]
    CALL foo
    ADD  ESP,8
    MOV  [_z],EAX

```

DONE:

(c) `y=0;`
`for (x=1;x>0;x<=1) y++;`

Solution:

```

    MOV  DWORD [_y],0
    MOV  DWORD [_x],1
TOP:  CMP  DWORD [_x],0
    JNG  DONE
    INC  DWORD [_y]
    SHL  DWORD [_x],1
    JMP  TOP

```

DONE:

42. Translate the following C function into a sequence of Intel protected mode instructions

```

long int Limit(long int min, long int x, long int max)
{
if (x<min) x = min;
if (x>max) x = max;
return x;
}

```

Solution:

```

Limit:  PUSH  EBP
        MOV  EBP,ESP
        MOV  EAX,[EBP+12] ;EAX<-x
        CMP  EAX,[EBP+8]  ;x<min?
        JNL  L1
        MOV  EAX,[EBP+8]  ;x<-min
L1:     CMP  EAX,[EBP+16] ;x>max?
        JNG  L2

```

```

                MOV    EAX,[EBP+16]    ;x<-max
L2:             POP    EBP
                RET

```

43. Translate each of the following C assignment statements into an equivalent sequence of Intel assembly language instructions:

```

(a) signed long long int; /* 64 bits */
    signed long y;        /* 32 bits */

    x=y;

```

Solution:

```

MOV  EAX,[_y]
CDQ
MOV  [_x],EAX
MOV  [_x+4],EDX

```

```

(b) unsigned short int x,y,z; /* 16 bits */
    z = x/y;

```

Solution:

```

MOV  AX,[_x]
SUB  DX,DX
DIV  WORD [_y]
MOV  [_z],AX

```

```

(c) signed long int x,y,z; /* 32 bits */
    z = x % y;

```

Solution:

```

MOV  EAX,[_x]
CDQ
IDIV DWORD [_y]
MOV  [_z],EDX

```

44. Translate the following C function into a sequence of Intel protected mode instructions.

```

void FillArray(char *baseAdrs, int items, char value)
{
while (items>0)
{
    items = items-1;
    baseAdrs[items]=value;
}
}

```

```
}  
}
```

Solution:

```
FillArray: PUSH   EBP           ;standard prologue  
           MOV    EBP,ESP  
           MOV    EAX,[EBP+8]   ;get baseAdrs  
           MOV    ECX,[EBP+12]  ;get items  
           MOV    EDX,[EBP+16]  ;get value  
L1:        CMP    ECX,0         ;items>0?  
           JNG   L2           ;done if not  
           DEC   ECX          ;items--  
           MOV   [EAX+ECX],EDX ;fill element  
           JMP  L1           ;repeat loop  
L2:        POP   EBP          ;epilogue  
           RET
```

45. Which of the following must be true in order for the CPU in an 80x86 based system with 8259 interrupt controller to respond to an interrupt request? (list all that apply)
- (a) The interrupting device must raise its corresponding IRQ line;
 - (b) The 8259 PIC mask register must have a 0 in the bit that corresponds to the device's IRQ line;
 - (c) The 8259 PIC in-service register must have 0's in the bits that correspond to the IRQ lines of all higher priority devices;
 - (d) Flag IF must be set to 1;
 - (e) The current instruction must be completed.

Solution:All of the above.

46. Which of the following are true statements? (list all that apply)
- (a) Interrupts may be serviced between two memory cycles;
 - (b) Interrupts may be serviced between the execution of two assembly language instructions;
 - (c) Interrupts may be serviced between the execution of two C statements.

Solution:Options 46b and 46c.

47. Which of the following are true statements? (list all that apply)
- (a) DMA transfer may occur between two memory cycles;

- (b) DMA transfer may occur between the execution of two assembly language instructions;
- (c) DMA transfer may occur between the execution of two C statements;

Solution:All of the above.

48. Which of the following 80x86 registers are pushed on the stack when the CPU responds to an interrupt? (list all that apply)

- (a) EAX
- (b) EIP
- (c) DS
- (d) CS
- (e) EFlags

Solution:Options 48b, 48d and 48e.

49. Which of the following are re-enabled by executing an STI instruction immediately upon entry to an ISR?

- (a) Interrupts from all devices;
- (b) Interrupts from lower priority devices;
- (c) Interrupts from higher priority devices.

Solution:Option 49c.

50. Which of the following provide the index used to retrieve an address from the Interrupt Descriptor Table (IDT) for a hardware interrupt?

- (a) The 8259 PIC;
- (b) The second byte of an INT instruction;
- (c) A hard-coded constant built into the CPU.

Solution:Option 50a

51. Which of the following provide the index used to retrieve an address from the Interrupt Descriptor Table (IDT) for a software interrupt?

- (a) The 8259 PIC;
- (b) The second byte of an INT instruction;
- (c) A hard-coded constant built into the CPU.

Solution:Option 51b

52. Which of the following provide the index used to retrieve an address from the Interrupt Descriptor Table (IDT) for an exception?
- (a) The 8259 PIC;
 - (b) The second byte of an INT instruction;
 - (c) A hard-coded constant built into the CPU.

Solution:Option 52c.

53. Which of the following are true? (list all that apply)
- (a) The starting address of the interrupt descriptor table (IDT) is always zero;
 - (b) The starting address of the IDT is stored in the IDTR register;
 - (c) The starting address of the IDT is different for hardware interrupts, software interrupts and exceptions.

Solution:Option 53b.

54. Which of the following I/O methods provides the *slowest* possible data transfer rate?
- (a) Polled waiting loop;
 - (b) Interrupt driven I/O;
 - (c) DMA.

Solution:Option 54b.

55. If interrupts are enabled and no interrupt routines are executing, the maximum delay before an interrupt request is acknowledged by the CPU is...
- (a) The time required for one memory read or write cycle;
 - (b) The maximum time required to fetch a single instruction from memory;
 - (c) The maximum time required to fetch and execute a single instruction.

Solution:Option 55c.

56. The maximum delay before a DMA request is acknowledged by the CPU is:
- (a) The time required for one memory read or write cycle;
 - (b) The maximum time required to fetch a single instruction from memory;
 - (c) The maximum time required to fetch and execute a single instruction.

Solution:Option 56a.

57. In an Intel 80x86 microprocessor system, what device limits the ultimate effect of executing an STI (enable interrupts) instruction at the beginning of an interrupt service routine so that only interrupts of *higher* priority are re-enabled?
- (a) The Control Unit inside the CPU;
 - (b) The Interrupt Enable Flag (IF) of the EFlags register inside the CPU;
 - (c) The 'In service' register inside the 8259A programmable interrupt controller (PIC);
 - (d) The 'mask' register inside the 8259A PIC.

Solution:Option 57c.

58. At the end of an interrupt service routine on an Intel 80x86 based microprocessor system, we typically find a pair of instructions that send the value 20_{16} out to I/O port 20_{16} in order to re-enable interrupts from *lower* priority devices. Executing these two instructions changes the values held in
- (a) The Control Unit inside the CPU;
 - (b) The interrupt enable flag (IF) of the EFlags register inside the CPU;
 - (c) The 'In service' register inside the 8269A Programmable Interrupt Controller (PIC);
 - (d) The 'Mask' register inside the PIC.

Solution:Option 58c.

59. Which of the following techniques should *never* be used to protect a critical section?
- (a) Disabling interrupts;
 - (b) Disabling task switching;
 - (c) Spin locks;
 - (d) Mutex objects;
 - (e) Semaphores.

Solution:Option 59c.

60. Which of the following is only appropriate for protecting *short* critical sections? (list all that apply)
- (a) Disabling interrupts;
 - (b) Disabling task switching;
 - (c) Spin locks;

- (d) Mutex objects;
- (e) Semaphores.

Solution:Option 60a - any other method has a higher run-time overhead.

61. Which of the following causes unnecessary context switching? (list all that apply)

- (a) Disabling interrupts;
- (b) Disabling task switching;
- (c) Spin locks;
- (d) Mutex objects;
- (e) Semaphores.

Solution:Option 61c.

62. Which of the following can protect data that is shared between a thread and an interrupt service routine? (list all that apply)

- (a) Disabling interrupts;
- (b) Disabling task switching;
- (c) Spin locks;
- (d) Mutex objects;
- (e) Semaphores.

Solution:Option 62a.

63. Which of the following methods can prevent *loss* of data due to interrupt overrun?

- (a) Setting a flag while the interrupt service routine (ISR) is busy, and testing on entry;
- (b) Keeping interrupts enabled during the ISR
- (c) Masking the device's IRQ line in the PIC;
- (d) Using a faster processor.

Solution:Option 63d - the key word here is *loss*. All other techniques prevent overrun but lose data.

64. True or False : Each thread in a *non-preemptive* multi-threaded application must have its own stack.

Solution:True.

65. True or False : Each thread in a *preemptive* multi-threaded application must have its own stack.

Solution:True.

66. When a *non-preemptive* kernel is used, the context switch occurs:

- (a) Only by an explicit call to a kernel function;
- (b) Only because of a hardware interrupt;
- (c) Either of the above.

Solution:Option 66a.

67. When a *preemptive* kernel is used, the context switch occurs:

- (a) Only by an explicit call to a kernel function;
- (b) Only because of a hardware interrupt;
- (c) Either of the above.

Solution:Option 67c.

68. The following function transmits a packet of information over a serial communications line but may transmit corrupted packets when called by more than one thread in a multi-tasking application. Describe what changes (if any) are required and why.

- (a) When a *preemptive* kernel is used;
- (b) When a non-preemptive kernel is used.

Don't forget to consider how the function `Send1Byte` might be coded.

```
void SendPacket(int type, char *packet, int length)
{
    Send1Byte(0xFF);
    Send1Byte(type);
    Send1Byte(length);
    while (length--) Send1Byte(*packet++);
}
```

Solution:Preemptive Kernel: All of the code inside the `SendPacket` function comprises a critical section and must be protected against an interrupt which may trigger a context switch (and thus a possible reentry) by another thread. Immediately upon entry to the function we need a semaphore pend, and just before exit we need a semaphore post. Disabling interrupts is not acceptable since the `Send1Byte` function may be interrupt driven.

Non-Preemptive Kernel: A context switch in a non-preemptive application would require an explicit *yield* call to be coded by the application programmer. Although non appear within the SendPacket function, it is possible (and likely) that one is coded within the Send1Byte function while it waits for the serial device to finish sending the previous byte. Disabling interrupts won't provide protection since it won't prevent the yield call. Again, the only reliable solution is as before - protecting the critical section with a semaphore pend-post.

69. Which of the following statements are true?

- (a) Priority inversions can cause data corruption;
- (b) Priority inversions can cause deadlock;
- (c) The duration of a *bounded* priority inversion is no longer than the execution time of the critical section of the low priority thread;
- (d) *Unbounded* priority inversions can cause a system to miss deadlines.

Solution:Options 69c and 69d.

70. Which thread is running during an *unbounded* priority inversion?

- (a) The high priority thread;
- (b) The medium priority thread;
- (c) The low priority thread.

Solution:Option 70b.

71. A bounded priority inversion ends when the low-priority thread...

- (a) Acquires the shared resource;
- (b) Starts its critical section;
- (c) Completes its critical section;
- (d) Is preempted.

Solution:Option 71c.

72. Which threads compete for the same shared resource during an unbounded priority inversion?

- (a) The high priority thread;
- (b) The medium priority thread;
- (c) The low priority thread.

Solution:Options 72a and 72c.

73. Suppose that threads T1 and T2 compete for shared resources R1 and R2. Which resource acquisition sequences can cause deadlock.
- (a) T1 requests R1 and then R2; T2 requests R1 and then R2;
 - (b) T1 requests R1 and then R2; T2 requests R2 and then R1;
 - (c) T1 requests R2 and then R1; T2 requests R1 and then R2;
 - (d) T1 requests R2 and then R1; T2 requests R2 and then R1;

Solution:Options 73b and 73c.

74. Which solution to unbounded priority inversion has the least detrimental effect on system response time?
- (a) The priority ceiling protocol;
 - (b) The priority inheritance protocol.

Solution:Option 74b.

75. When both the main program and an interrupt service routine of a conventional (non-multithreaded) application program access the same shared data, what method *must* be used in the main program to prevent data corruption?
- (a) Disabling interrupts;
 - (b) Disabling task switching;
 - (c) Spin locks;
 - (d) Semaphores.

Solution:Option 75a.

76. A spin lock uses a flag stored in a Boolean variable to indicate whether or not a corresponding shared resource is available. Which of all of the following statements are true?
- (a) The flag can only be tested while the scheduler is running the thread that contains the spin lock;
 - (b) The flag itself is a shared memory object;
 - (c) A spin lock's test-and-set operation is a critical section;
 - (d) With spin locks, while one thread owns the shared resource, other threads waiting for the same resource can be removed from the scheduler's list of ready threads.

Solution:Options 76a, 76b and 76c.

77. When a non-preemptive kernel is used, the kernel yield function:

- (a) Never needs to be called from within the threads;
- (b) Must be called inside any waiting loop;
- (c) Should only be called outside of waiting loops.

Solution:Option 77b.

78. Any kernel call that changes the ready status of a thread:

- (a) Always invokes the scheduler, and thus perhaps a context switch;
- (b) Only invokes the scheduler in a non-preemptive kernel;
- (c) Only invokes the scheduler in a preemptive kernel.

Solution:Option 78a.

79. Which of the following statements are true? A priority inversion:

- (a) May cause deadlock;
- (b) May occur whenever two or more threads compete for a shared resource;
- (c) Can be prevented using either priority inheritance or priority ceiling protocols;
- (d) Causes data corruption in shared memory objects.

Solution:Option 79b.

80. Which thread owns the shared resource during an *unbounded* priority inversion?

- (a) The high priority thread;
- (b) The medium priority thread;
- (c) The low priority thread.

Solution:Option 80c.

81. A bounded priority inversion becomes unbounded when

- (a) The low priority thread is preempted by the medium priority thread;
- (b) The low priority thread is preempted by the high priority thread;
- (c) The medium priority thread is preempted by the high priority thread.

Solution:Option 81a.

82. Consider an application with three threads T1, T2 and T3 and three shared resources R1, R2 and R3. Fill in the blanks in a manner that corresponds to a three-way deadlock

- (a) T1 owns ?? and is pending on ??;
- (b) T2 owns ?? and is pending on ??;
- (c) T3 owns ?? and is pending on ??;

Solution:

- (a) T1 owns R1 and is pending on R2;
- (b) T2 owns R2 and is pending on R3;
- (c) T3 owns R3 and is pending on R1;

83. Which solution to unbounded priority inversion raises a thread's priority for the *least* amount of time?

- (a) The priority ceiling protocol;
- (b) The priority inheritance protocol.

Solution:Option 83b.

84. Which of the following memory allocation methods can be used in a shared function for thread-specific local objects?

- (a) Automatic;
- (b) Static;
- (c) Dynamic;
- (d) Those generated by an `alloca` call.

Solution:Options 84a, 84c and 84d.

85. Which of the following memory allocation methods can be used in a shared function for a shared local object?

- (a) Automatic;
- (b) Static;
- (c) Dynamic;
- (d) Those generated by an `alloca` call.

Solution:Option 85b.

86. Which of the following statements are true?

- (a) All shared functions are called by more than one thread;
- (b) All shared functions are re-entrant;

- (c) All shared functions contain shared objects;
- (d) All shared functions are thread-safe;

Solution:Option 86a.

87. Use casts and subscripting to set the 2nd byte (bits 8-15) of a 32-bit int called x to zero without modifying the other three bytes.

Solution:

```
((char *)&x)[1]=0;
```

88. What is the maximum number of bytes of memory that are allocated for data at any moment during the execution of the following programs?

(a)

```
void f1(void) {static char a[1000]};
void f2(void) {static char a[2000]};
int main(void){ f1(); f2(); return 0; }
```

(b)

```
void f1(void) {char a[1000]};
void f2(void) {char a[2000]};
int main(void){ f1(); f2(); return 0; }
```

(c)

```
void f1(void) {char *p=malloc(1000)};
void f2(void) {char *p=malloc(2000)};
int main(void){ f1(); f2(); return 0; }
```

(d)

```
void f1(int n) {char a[n]};
void f2(int n) {char a[n]};
int main(void){ f1(1000); f2(2000); return 0; }
```

(e)

```
int main(void)
{
  int j,n=1000;
  for (j=0; j<2; j++)
  {
    char *p=alloc(n);
  }
}
```

(f)

```
int main(void)
{
  int j,n=1000;
  for (j=0; j<2; j++)
  {
    char a[n];
  }
}
```

}

Solution:Option 88a - 3000 bytes, Option 88b - 2000 bytes, 88c - 3000 bytes, Option 88d - 2000 bytes, Option 88e - 2012 bytes, Option 88f - 2012 bytes.

89. Which of the following memory allocation methods support conservation of memory through re-use?

- (a) Automatic;
- (b) Register;
- (c) Static;
- (d) Dynamic;
- (e) Using `alloca`;
- (f) Variable size arrays;
- (g) None of the above;

Solution:All but Options 89c and 89g.

90. Which of the following memory allocation methods are possible outside of a function?

- (a) Automatic;
- (b) Register;
- (c) Static;
- (d) Dynamic;
- (e) Using `alloca`;
- (f) Variable size arrays;
- (g) None of the above;

Solution:Option 90c only.

91. Which of the following memory allocation methods cause the same object to be initialized more than once during execution?

- (a) Automatic;
- (b) Register;
- (c) Static;
- (d) Dynamic;
- (e) Using `alloca`;
- (f) Variable size arrays;

(g) None of the above;

Solution:Options 91a and 91b.

92. Which of the following memory allocation methods require calling a function to create an object?

- (a) Automatic;
- (b) Register;
- (c) Static;
- (d) Dynamic;
- (e) Using `alloca`;
- (f) Variable size arrays;
- (g) None of the above;

Solution:Option 92d.

93. Which of the following memory allocation methods destroys objects transparently during program execution?

- (a) Automatic;
- (b) Register;
- (c) Static;
- (d) Dynamic;
- (e) Using `alloca`;
- (f) Variable size arrays;
- (g) None of the above;

Solution:Options 93a, 93b, 93e and 93f.

94. Which of the following memory allocation methods may be used to maintain a value in an object from one invocation of a function to the next invocation of the same function?

- (a) Automatic;
- (b) Register;
- (c) Static;
- (d) Dynamic;
- (e) Using `alloca`;
- (f) Variable size arrays;

(g) None of the above;

Solution:Options 94c and 94d.

95. Which of the following memory allocation methods will prevent application of the 'address of' operator to an object?

- (a) Automatic;
- (b) Register;
- (c) Static;
- (d) Dynamic;
- (e) Using `alloca`;
- (f) Variable size arrays;
- (g) None of the above;

Solution:Option 95b.

96. Which of the following memory allocation methods requires that the allocated object be referenced indirectly through a pointer?

- (a) Automatic;
- (b) Register;
- (c) Static;
- (d) Dynamic;
- (e) Using `alloca`;
- (f) Variable size arrays;
- (g) None of the above;

Solution:Options 96d and 96e.

97. Which of the following memory allocation methods is not available inside a function?

- (a) Automatic;
- (b) Register;
- (c) Static;
- (d) Dynamic;
- (e) Using `alloca`;
- (f) Variable size arrays;
- (g) None of the above;

Solution:Option 97g.

98. Which of the following memory allocation methods is used for function parameters?

- (a) Automatic;
- (b) Register;
- (c) Static;
- (d) Dynamic;
- (e) Using `alloca`;
- (f) Variable size arrays;
- (g) None of the above;

Solution:Option 98a.

99. The code shown below compiles without fatal errors, but doesn't execute as intended. Provide a corrected version.

```
char *AddressOfCopy(int value)
{
int copy=value;
return &copy;
}
```

Solution:

```
char *AddressOfCopy(int value)
{
static int copy=value;
copy = value;
return &copy;
}
```

100. Consider the program below, and then answer the following questions.

- (a) What is the maximum number of instances of the function parameter `n` that co-exist at any one time during the execution of the program?
- (b) What is the maximum number of instances of the static integer `s` that co-exist at any one time during the execution of the program?
- (c) What is the maximum number of instances of automatic integer `a` that co-exists at any one time during the execution of the program?

(d) Which two of the three objects n, s and a are allocated from the same region of memory?

```
#include <stdio.h>

void go(int n)
{
    static int s=0;
        int a=++s;

    printf("Enter: Object Address Contents\n");
    printf("      n   %08X   %d\n", &n, n);
    printf("      a   %08X   %d\n", &a, a);
    printf("      s   %08X   %d\n", &s, s);
    printf("\n");

    if (n) go(n-1);

    printf("Leave: Object Address Contents\n");
    printf("      n   %08X   %d\n", &n, n);
    printf("      a   %08X   %d\n", &a, a);
    printf("      s   %08X   %d\n", &s, s);
    printf("\n");

}

int main(void)
{
    go(3);
    return 0;
}
```

Solution: Question 100a - 4, Question 100b - 1, Question 100c - 4, Question 100d - “n” and “a”.

101. What do each of the printf statements print?

```
int x=0;

void one(void)    { int x; x=1; }
void two(int x)  { x=2; }
```

```

void three(void) { extern int x; x=2; }
void four(int *p) { *p=3; }

int main(void)
{
printf("%d",x);
x=4; one();
printf("%d",x);
x=5; two(x);
printf("%d",x);
x=6;
{
int x;
x=7; three();
printf("%d",x);
}
printf("%d",x);
x=8; four(&x);
printf("%d",x);

return 0;
}

```

Solution:In order of printf call: 0,4,5,7,2,3.

102. What do each of the printf statements print?

```

int foo(void)
{
int x=0;
x = x+1;
return x;
}

int bar(void)
{
static int x=0;
x=x+1;
return x;
}

int main(void)

```

```

{
printf("%d",foo());
printf("%d",bar());
printf("%d",foo());
printf("%d",bar());
return 0;
}

```

Solution:In order of printf call: 1,1,1,2

103. Rearrange the code shown below so that the scope of all variables is restricted as much as possible without altering the intended operation of the code.

```

#include <stdio.h>

int old_value=0;
int average;

void f(int value)
{
if (value != old_value)
{
average = (value+old_value)/2;
printf("average = %d\n",average);
}
old_value = value;
}

```

Solution:

```

#include <stdio.h>

void f(int value)
{
static int old_value=0;
if (value != old_value)
{
int average;
average = (value+old_value)/2;
printf("average = %d\n",average);
}
}

```

```

    }
old_value = value;
}

```

104. Which lines of the following C program cause the compiler to produce an *error* message (not just a warning message)?

```

int *f(int p)
{
register int r1=0;
register int r2=r1+1;
int a1=0;
int a2=a1+1;
static int s1=0;
static int s2=s1+1;

if (r1==0) return &r1;
if (a1==0) return &a1;
if (s1==0) return &s1;

return &p;
}

```

Solution:

```

static int s1=0;
static int s2=s1+1; /* Error! Initial value cannot */
/* depend on variable */

if (r1==0) return &r1; /* Error! Can't return address of */
/* register variable */

```

105. Suppose that variables *x*, *y* and *z* are all declared to be of type *int* which is a 16-bit quantity. Suppose further that they are used to represent signed fixed point quantities as follows

x = 8.8 *y* = 10.6 *z* = 12.4,

where *x* = *a.b* means that *x* has *a* bits for the whole number plus sign bit part and *b* bits for the fractional part. Translate the following floating point statements into their fixed point equivalents.

(a) $z = x+y;$

Solution:

$$z = (x \gg 4) + (y \gg 2);$$

(b) $z = x*y;$

Solution:

$$z = (x*y) \gg 10;$$

(c) $z = x/y;$

Solution:

$$z = (x \ll 2) / y;$$

(d) $y = x*z;$

Solution:

$$z = (x*z) \gg 6;$$

(e) $z = 1 + x + x^2/2;$

Solution:

$$z = (1 \ll 4) + x \gg 4 + (x*x) \gg 11;$$