

SISTEMAS BASADOS EN MICROPROCESADORES

Grado en Ingeniería Informática Escuela Politécnica Superior – UAM

COLECCIÓN DE PROBLEMAS DE LOS TEMAS 2.7 A 5.4

P1. Si **SP=0006h** y **FLAGS=0210h** al inicio de la ejecución del código que se adjunta, indicar los valores contenidos en las **primeras seis posiciones de la pila** al ejecutar la primera instrucción del procedimiento **Leer_Datos**, tanto cuando todos los procedimientos del programa son cercanos (**NEAR**), como cuando son lejanos (**FAR**). La pila está inicializada a ceros.

```
2100:2250 E8A8FD    call Leer_Datos
2100:2253 89161000    mov Datos[0], dx
```

0	1	2	3	4	5	0	1	2	3	4	5
0	0	0	0	53h	22h	0	0	53h	22h	0	21h

Caso NEAR

Caso FAR

P2. Al inicio de la ejecución de una función invocada desde lenguaje C, se tiene que **SP=2** y que las 16 primeras posiciones de la pila contienen los siguientes valores:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11h	A0h	25h	00h	32h	00h	A2h	E9h	00h	C1h	24h	F1h	00h	63h	41h	12h

La signatura de dicha función es: `int fun (char* p, long n);`

Indicar el valor de los tres parámetros con que esa función fue invocada desde C, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**).

Caso NEAR: p = 0032h n = C100E9A2h

Caso FAR: p = C100:E9A2h n = 6300F124h

P3. Escribir en ensamblador el código necesario para **poner a 1 los bits 5, 10 y 14 del registro AX**, dejando todos los demás bits de ese registro intactos, y **poner a 0 los bits 5, 10 y 14 del registro BX**, dejando intactos los demás bits. Se valorará la eficiencia del código.

```
or ax, 0100010000100000b                      ; 4420h
and bx, 1011101111011111b                      ; BBDFh
```

P4. Usando los procedimientos lejanos `enviar0` y `enviar1`, escribir en ensamblador un procedimiento eficiente que envíe secuencialmente los bits del **registro AL**, desde el más significativo al menos significativo. Se valorará la eficiencia del código.

```

enviarAL PROC FAR

    push cx

    mov cx, 8          ; Itera los ocho bits de AL

bucle:    rcl al, 1     ; Pasa el bit más alto de AL al acarreo
          jc envial     ; Si hay acarreo envía 1, si no envía 0

          call enviar0
          jmp finbucle

envial:    call enviar1

finbucle:  dec cx
          jnz bucle

          rcl al, 1     ; Deja AL igual que al principio

    pop cx
    ret

enviarAL ENDP

```

P5. Al inicio de la ejecución de una función invocada desde lenguaje C, se tiene que **SP=0** y que las 16 primeras posiciones de la pila contienen los siguientes valores:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11h	A0h	25h	00h	32h	00h	A2h	E9h	00h	C1h	24h	F1h	00h	63h	41h	12h

La signatura de dicha función es: `int fun (char c, int n, char* p);`

Indicar el valor de los tres parámetros con que esa función fue invocada desde C, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**).

Caso NEAR: c = 25h n = 0032h p = E9A2h

Caso FAR: c = 32h n = E9A2h p = F124:C100h

P6. La función de lenguaje C cuya signatura se indica en el recuadro de la derecha es invocada desde el programa de código máquina que se muestra en el recuadro de la izquierda. En el momento anterior de la llamada, se suponen los siguientes valores del puntero de pila y de los parámetros de la función: **SP = 14**, **n = 1234h**, **c = ABh**, **p = 4253h:5678h**.

```
4253:0007 E8F6FF call _fun
4253:000A B8004C mov ax, 4C00h
```

```
fun ( int n, char c, char* p );
```

Indicar el valor de las 16 posiciones iniciales de la pila en el momento de ejecutarse la primera instrucción de código máquina de la función `fun`, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**).

Caso NEAR:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
						0Ah	00h	34h	12h	ABh	00h	78h	56h		

Caso FAR:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		0Ah	00h	53h	42h	34h	12h	ABh	00h	78h	56h	53h	42h		

P7. Escribir en ensamblador la función `_Multiply8U`, que **multiplica dos enteros sin signo de 8 bits mediante sumas sucesivas**. El primer operando ha de estar almacenado en **BH** y el segundo en **BL**. El **resultado ha de ser un entero sin signo de 16 bits** que se retornará en **AX**. La multiplicación se realizará sumando el primer operando tantas veces como indique el segundo. Se valorará la eficiencia del código.

```
_Multiply8U PROC NEAR

    mov ax, 0

    cmp bh, 0
    je fin      ; Primer operando es cero
    cmp bl, 0
    je fin      ; Segundo operando es cero

    push bx dx

    ; Pasa primer operando a dx
    mov dl, bh
    mov dh, 0

    ; Suma primer operando (dx) tantas veces como
    ; indica el segundo (bl)
seguir: add ax, dx

    dec bl
    jnz seguir

    pop dx bx

fin:    ret     ; Devuelve resultado en ax

_Multiply8U ENDP
```

P8. Llamando a la función de multiplicar desarrollada en el problema anterior, escribir en ensamblador la función de C que se reproduce en el siguiente recuadro, que calcula el producto escalar de dos vectores de **n** dimensiones cuyos elementos son enteros sin signo de 8 bits. Las variables locales están almacenadas en registros. Se supone que el programa de C está compilado en **modelo compacto**. Se valorará la eficiencia del código.

```
_DotProd8U PROC NEAR
```

```
    push bp
    mov bp, sp
    push bx cx dx si di ds es

    mov dx, 0    ; dx = res
```

```
    ; Salta @retorno (2 bytes por ser código NEAR) y bp (2 bytes)
    mov cx, [bp+4]    ; cx := n
    cmp cx, 0
    je fin            ; n es cero
```

```
    ; Punteros v1 y v2 ocupan 4 bytes por ser datos FAR
```

```
    lds si, [bp+6]    ; ds:si := v1
    les di, [bp+10]   ; es:di := v2
```

```
seguir: mov bh, [si]    ; bh := v1[i]
        mov bl, es:[di] ; bl := v2[i]
        call _Multiply8U ; ax := v1[i] * v2[i]
        add dx, ax      ; dx := dx + v1[i] * v2[i]
```

```
    ; i := i+1
    inc si
    inc di
```

```
    dec cx
    jnz seguir
```

```
fin:    mov ax, dx
        pop es ds di si dx cx bx
        pop bp
```

```
    ret                ; Devuelve resultado en ax
```

```
_DotProd8U ENDP
```

```
int DotProd8U (int n, char *v1, char *v2)
{
    register int i;
    register int res=0;

    for (i=0; i<n; i++)
        res=res + Multiply8U( v1[i], v2[i] );

    return res;
}
```

P9. La función de lenguaje C cuya signatura se indica en el recuadro de la derecha es invocada desde el programa de código máquina que se muestra en el recuadro de la izquierda. En el momento anterior de la llamada, se suponen los siguientes valores del puntero de pila y de los parámetros de la función: **SP = 16, n = 4321h, c = 12h, p = 1234h:8765h**.

```
5342:FF0A E8F6FF call _fun
5342:FF0D B8004C mov ax, 4C00h
```

```
fun ( char* p, int n, char c );
```

Indicar el valor de las 16 posiciones iniciales de la pila en el momento de ejecutarse la primera instrucción de código máquina de la función `fun`, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**).

Caso NEAR:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
								0Dh	FFh	65h	87h	21h	43h	12h	00h

Caso FAR:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
				0Dh	FFh	42h	53h	65h	87h	34h	12h	21h	43h	12h	00h

P10. Indicar el vector de la interrupción de **impresión por pantalla** dado el siguiente volcado de memoria.

```
0000:0000 54 02 CF 15 CE 01 CF 15 04 00 70 00 D7 01 CF 15
0000:0010 04 00 70 00 30 00 00 C8 30 00 00 C8 30 00 00 C8
```

Segmento = **C800h** Offset = **0030h**

P11. Escribir en ensamblador el código necesario para **poner a 1 los bits 0, 7 y 14 del registro AX**, dejando todos los demás bits de ese registro intactos, y **poner a 0 los bits 2, 10 y 15 del registro BX**, dejando intactos los demás bits. Se valorará la eficiencia del código.

```
or ax, 0100000010000001b      ; 4081h
and bx, 0111101111111011b     ; 7BFBh
```

P12. Escribir en ensamblador utilizando instrucciones básicas (sin instrucciones de manipulación de cadenas ni de bucles) la función `strlen` de C, cuya signature se reproduce a continuación. Esta función retorna la longitud de la cadena de caracteres que recibe como argumento. Dicha cadena acaba con un byte a cero. Se supone que el programa de C está compilado en **modelo largo**. Se valorará la eficiencia del código.

```
int strlen (char *s);
```

```
_strlen PROC FAR
    push bp
    mov bp, sp

    push ds bx

    lds bx, 6[bp]    ; Lee offset y segmento
    mov ax, 0        ; Inicializa contador
```

```

itera:  cmp BYTE PTR [bx], 0 ; Test de final de cadena
        je fin
        inc ax                ; Incrementa contador de caracteres
        inc bx                ; Apunta al siguiente carácter
        jmp itera

fin:     pop bx ds
        pop bp

        ret

_strlen ENDP

```

P13. Si **SP=0004h** y **FLAGS=0200h** al inicio de la ejecución del código que se adjunta, indicar los valores contenidos en las **primeras seis posiciones de la pila** al ejecutar la primera instrucción del procedimiento **Leer_Datos**, tanto cuando todos los procedimientos del programa son cercanos (**NEAR**), como cuando son lejanos (**FAR**). Se considera que la pila está inicializada a ceros.

```

4250:025E E8A8FD    call Leer_Datos
4250:0261 89161000    mov Datos[0], dx

```

0	1	2	3	4	5
00h	00h	61h	02h	00h	00h

Caso NEAR

0	1	2	3	4	5
61h	02h	50h	42h	00h	00h

Caso FAR

P14. Si **SP=0006h** y **FLAGS=0200h** al inicio de la ejecución del código que se adjunta, indicar los valores contenidos en las **primeras ocho posiciones de la pila** en el momento de ejecutar la primera instrucción de la rutina de servicio de la interrupción 61h. Se considera la pila inicializada a ceros.

```

4250:025E CD61      int 61h
4250:0260 89161000    mov Datos[0], dx

```

0	1	2	3	4	5	6	7
60h	02h	50h	42h	00h	02h	00h	00h

P15. Escribir en ensamblador un programa residente asociado a la interrupción 65h, que ejecute un retardo igual a **65536*N iteraciones**, con **N** siendo el valor recibido en el **registro AX**.

```

codigo SEGMENT
    ASSUME cs : codigo

    ORG 256

    inicio:  jmp instalar

    retardo PROC FAR        ; Procedimiento residente de retardo

```

```

        push ax cx

itera1:  xor cx, cx

itera2:  dec cx
        jnz itera2

        dec ax
        jnz itera1

        pop cx ax

        iret

retardo ENDP

instalar: xor ax, ax
        mov es, ax
        mov ax, offset retardo
        mov bx, cs

        cli
        mov es:[65h*4], ax
        mov es:[65h*4+2], bx
        sti

        ; Deja residente el procedimiento de retardo
        mov dx, offset instalar
        int 27h

codigo ENDS
END inicio

```

P16. Escribir en ensamblador un procedimiento lejano, **suma32**, que sume las variables globales de 32 bits **op1** y **op2**, dejando el resultado en la variable global de 32 bits **res**.

```

op1 dd ?      suma32 PROC FAR
op2 dd ?
res dd ?

        push ax
        push si

        mov si, 0

        ; Suma las dos palabras de menor peso

        mov ax, WORD PTR op1[si]
        add ax, WORD PTR op2[si]
        mov WORD PTR res[si], ax

        ; Pasa a apuntar a las palabras de mayor peso

        inc si
        inc si

        ; Suma con acarreo las dos palabras de mayor peso

        mov ax, WORD PTR op1[si]
        adc ax, WORD PTR op2[si]
        mov WORD PTR res[si], ax

        pop si
        pop ax

```

```

ret
suma32 ENDP

```

P17. Si **SP=0006h** y **FLAGS=0210h** al inicio de la ejecución del código que se adjunta, indicar los valores contenidos en las **primeras seis posiciones de la pila** al ejecutar la primera instrucción del procedimiento **Leer_Datos**, tanto cuando todos los procedimientos del programa son cercanos (**NEAR**), como cuando son lejanos (**FAR**). La pila está inicializada a ceros.

```

2100:2250 E8A8FD    call Leer_Datos
2100:2253 89161000    mov Datos[0], dx

```

0	1	2	3	4	5
0	0	0	0	53h	22h

Caso NEAR

0	1	2	3	4	5
0	0	53h	22h	0	21h

Caso FAR

P18. Indicar el vector de la interrupción de **punto de ruptura** (*breakpoint*) dado el siguiente volcado de memoria.

```

0000:0000 54 02 CF 15 CE 01 CF 15 04 00 70 00 D7 01 CF 15
0000:0010 04 00 70 00 30 00 00 C8 30 00 00 C8 30 00 00 C8

```

Segmento = 15CFh Offset = 01D7h

P19. Se tiene una matriz bidimensional de tamaño (**FILAS** x **COLUMNAS**) almacenada por filas en la variable **Matriz2D**. Escribir en ensamblador un procedimiento lejano, **escribelcol**, que reciba la **dirección de la matriz en el registro BX** y ponga a **uno** todos los elementos de la **columna indicada en el registro AX**. Se valorará la eficiencia del código.

```

FILAS = 10
COLUMNAS = 20
Matriz2D db FILAS*COLUMNAS dup (?)

mov bx, offset Matriz2D
mov ax, 4
call escribelcol      ; Pone a 1 los elementos de la columna 4
                      ; de Matriz2D

escribelcol PROC FAR

    push cx, si

    mov cx, FILAS      ; Itera el número de filas dado
    mov si, ax         ; Índice a primer elemento de columna dada

buclecol:  mov BYTE PTR [bx][si], 1
           add si, COLUMNAS      ; Índice pasa a siguiente fila

```



```

        dec cx
        jnz buclecol

    pop si, cx
    ret

escribelcol ENDP

```

P20. Suponiendo que **SS=424Dh**, **SP=14**, **AX=3412h** y **BX=5678h**, indicar el **valor hexadecimal de los 16 primeros bytes del segmento SS** una vez ejecutado el siguiente programa.

```

    push AX
    pop BX
    push SS
    push BX

```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
										12h	34h	4Dh	42h		

P21. Declarar mediante directivas de ensamblador de 8086 las mismas variables que aparecen en el siguiente extracto en lenguaje C, teniendo en cuenta que las cadenas de caracteres en C acaban con el byte 0.

```

char nombre[20];           // Cadena de caracteres de 20 bytes
short edad = 17;           // Entero de 2 bytes inicializado
char tabla2D[10][2];       // Tabla de bytes de 10 filas por 2 columnas
short valores[5] = { 1, 2, 3, 4, 5 };
char despedida[20] = "Hasta luego";

```

```

_nombre db 20 dup (?)
_edad dw 17
_tabla2D db 10*2 dup (?)
_valores dw 1, 2, 3, 4, 5
_despedida db "Hasta luego", 0, 20-($-despedida) dup (?)

```

P22. El siguiente programa en lenguaje ensamblador de 8086, que debe **invertir el orden de los caracteres de una cadena dada de 512 bytes como máximo**, tiene varios errores. Proponer una versión correcta del mismo programa haciendo el **menor número de cambios**. Sólo es necesario reescribir las líneas erróneas.

```

datos segment
    cadena      dw  "Hola"
    longitud    db  cadena-$
datos ends

resultados segment
    resultado   db  200 dup (?)
resultados ends

codigo segment
    assume cs:codigo, ds:datos
    invertir proc far
        mov ax, datos
        mov ds, ax
        mov ax, resultado
        mov es, ax
        mov si, longitud
        mov di, 0
seguir:    mov al, cadena[si-1]
        mov resultado[di], al
        dec si
        inc di
        jz seguir
        mov ax, 4C00h
        int 21h
    invertir endp
codigo ends
end codigo

```

```

datos segment
    cadena      db  "Hola"
    longitud     dw  $-cadena
datos ends

resultados segment
    resultado   db  200h dup (?)
resultados ends

codigo segment
    assume cs:codigo, ds:datos, es:resultados
    invertir proc far
        mov ax, datos
        mov ds, ax
        mov ax, resultados
        mov es, ax
        mov si, longitud
        mov di, 0
seguir:    mov al, cadena[si-1]
        mov resultado[di], al
        inc di
        dec si
        jnz seguir
        mov ax, 4C00h
        int 21h
    invertir endp
codigo ends
end invertir

```

P23. Escribir en ensamblador un procedimiento lejano (descontar2_32) que **decremente en dos unidades** el valor de la variable de 32 bits cuya dirección recibe mediante los registros AX y BX tal como se indica en el código adjunto. Tras su ejecución, este procedimiento no deberá alterar los valores previos de ningún registro del banco general ni de segmento. Se valorará la eficiencia del código.

```

datos segment
    contador      dd  0FFFFFFFFh
datos ends

...

mov ax, OFFSET contador
mov bx, SEG contador

call descontar2_32

```

descontar2_32 PROC FAR

```

    push bx es

    mov es, bx
    mov bx, ax

    sub WORD PTR es:[bx], 2
    sbb WORD PTR es:[bx+2], 0

    pop es bx

    ret

```

descontar2_32 ENDP

P24. Al inicio de la ejecución de una función invocada desde lenguaje C, se tiene que **SP=2** y que las 16 primeras posiciones de la pila contienen los siguientes valores:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11h	A0h	25h	00h	32h	00h	A2h	E9h	00h	C1h	24h	F1h	00h	63h	41h	12h

La signatura de dicha función es: `int fun (char* p, long n);`

Indicar el valor de los tres parámetros con que esa función fue invocada desde C, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**).

Caso NEAR: p = 0032h n = C100E9A2h

Caso FAR: p = C100:E9A2h n = 6300F124h

P25. Si **SP=0004h** y **FLAGS=0210h** al inicio de la ejecución del código que se adjunta, indicar los valores contenidos en las **primeras seis posiciones de la pila** al ejecutar la primera instrucción del procedimiento `Leer_Datos`, tanto cuando todos los procedimientos del programa son cercanos (**NEAR**), como cuando son lejanos (**FAR**). La pila está inicializada a ceros.

```
20FF:4000 E8A8FD    call Leer_Datos
20FF:4003 89161000    mov Datos[0], dx
```

0	1	2	3	4	5	0	1	2	3	4	5
0	0	03h	40h	0	0	03h	40h	FFh	20h	0	0

Caso NEAR

Caso FAR

P26. Declarar mediante directivas de ensamblador de 8086 las mismas variables que aparecen en el siguiente extracto en lenguaje C, teniendo en cuenta que las cadenas de caracteres en C acaban con el byte 0 y que el tipo `short` ocupa 2 bytes.

```
char caracter = 'A';
short edad;
short tabla[100];
char valores[5] = { 1, 2, 3, 4, 5 };
char despedida[12] = "Hasta luego";

_caracter db 'A'
_edad dw ?
_tabla dw 100 dup (?)
_valores db 1, 2, 3, 4, 5
_despedida db "Hasta luego", 0
```

P27. El siguiente programa en lenguaje ensamblador de 8086, que debe **contar el número de caracteres de una cadena dada de 512 bytes como máximo**, tiene varios errores. Proponer una versión correcta del mismo programa haciendo el **menor número de cambios**. Sólo es necesario reescribir las líneas erróneas.

```

datos segment
    cadena    db  "Hola", 0
datos ends

resultados segment
    resultado db  2 dup(?)
resultados ends

codigo segment
    assume cs:codigo, ds:datos,
es:resultados

    contar proc far
        mov ax, cadena
        mov ds, ax
        mov ax, resultados
        mov es, ax
        mov si, 4
seguir: mov cadena[si], 0
        jz fin
        dec si
        jmp fin
fin:    mov resultado, si
        mov ax, 4C00h
        int 21h
        contar endp
codigo ends
end contar

```

```

datos segment
    cadena    db  "Hola", 0
datos ends

resultados segment
    resultado db  2 dup(?)
resultados ends

codigo segment
    assume cs:codigo, ds:datos, es:resultados

    contar proc far
        mov ax, datos
        mov ds, ax
        mov ax, resultados
        mov es, ax
        mov si, 0
seguir: cmp cadena[si], 0
        jz fin
        inc si
        jmp seguir
fin:    mov WORD PTR resultado, si
        mov ax, 4C00h
        int 21h
        contar endp
codigo ends
end contar

```

P28. Escribir en ensamblador un procedimiento lejano (contar4_48) que **incremente en cuatro unidades** el valor de la variable de 48 bits cuya dirección recibe mediante los registros AX y BX tal como se indica en el código adjunto. Tras su ejecución, este procedimiento no deberá alterar los valores previos de ningún registro del banco general ni de segmento. Se valorará la eficiencia del código.

```

datos segment
    contador db  6  dup(0)
datos ends

...

mov ax, OFFSET contador
mov bx, SEG contador

call contar4_48

```

contar4_48 PROC FAR

push bx es

mov es, bx

contar4_48 ENDP

dejarse en blanco.

```
549A:0260 89161000  mov  Datos[0], dx
```

				60h	02h	9Ah	54h	34h	12h						
--	--	--	--	-----	-----	-----	-----	-----	-----	--	--	--	--	--	--

las 16 primeras posiciones de la pila contienen los siguientes valores:

11h	A0h	25h	00h	32h	00h	A2h	E9h	00h	C1h	24h	F1h	00h	63h	41h	12h
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

La signatura de dicha función es: `int fun (int *p, char c, int n);`

Indicar el valor de los tres parámetros con que esa función fue invocada desde C, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**).

Caso NEAR: **p = E9A2h** **c = 00h** **n = F124h**

Caso FAR: **p = F124h:C100h c = 00h n = 1241h**

_Instalar_61h. Se valorará la eficiencia del código.

```
_Instalar_6lh PROC NEAR
    jmp inicio
```

```
rsi61 dw ?, ?
```

; Dirección larga anterior

```

inicio: push ax es
        mov ax, 0
        mov es, ax
        mov ax, es:[61h*4]          ; Guarda Offset de rsi anterior
        mov cs:rsi61, ax
        mov ax, es:[61h*4 + 2]      ; Guarda Segmento de rsi anterior
        mov cs:rsi61[2], ax

        ; Cambia rsi de 61h
        cli
        mov word ptr es:[61h*4], offset _RSI_61h
        mov word ptr es:[61h*4 + 2], seg _RSI_61h
        sti

        pop es ax
        ret

 Instalar_61h ENDP

```

P32. Escribir en ensamblador de 8086 utilizando instrucciones básicas (sin instrucciones de manipulación de cadenas ni de bucles) la función `strcmp` de C, cuya signatura se reproduce a continuación. Esta función retorna un entero que indica la relación entre las dos cadenas que recibe como argumentos: Un valor de cero indica que ambas cadenas son iguales. Un valor mayor que cero indica que el primer carácter que no coincide tiene un valor mayor en `str1` que en `str2`. Un valor menor que cero indica lo contrario. Se considera que el programa en C está compilado en **modelo pequeño** (*small*). Las cadenas de caracteres en C acaban con un cero. Se valorará la eficiencia del código.

```

int strcmp (const char *str1, const char* str2);

_strcmp PROC NEAR
    push bp
    mov bp, sp
    push bx si di

    mov si, bp[4]      ; si <= str1
    mov di, bp[6]      ; di <= str2

    mov ax, 0          ; Por defecto son iguales

continuar: mov bl, [si]   ; bl <= str1[i]
           cmp bl, [di]   ; str1[i] - str2[i]
           je iguales
           ja str1_mayor
           mov ax, -1      ; str1 es menor que str2
           jmp final
str1_mayor: mov ax, 1      ; str1 es mayor que str2
           jmp final
iguales:   cmp bl, 0       ; str1[i] = 0?
           je final        ; Acaban ambas cadenas

           ; Continúa con siguiente carácter
           inc si
           inc di
           jmp continuar

```

```

final:      pop di si bx bp
           ret
_strcmp ENDP

```

P33. Si **SP=0004h** y **FLAGS=1234h** al inicio de la ejecución del código que se adjunta, indicar los valores contenidos en las **primeras seis posiciones de la pila** al ejecutar la primera instrucción del procedimiento **Leer_Datos**, tanto cuando todos los procedimientos del programa son cercanos (**NEAR**), como cuando son lejanos (**FAR**). Los valores desconocidos de la pila deben dejarse en blanco.

```

43FF:25E3 E8A8FD    call Leer_Datos
43FF:25E6 89161000   mov Datos[0], dx

```

0	1	2	3	4	5
		E6h	25h		

Caso NEAR

0	1	2	3	4	5
E6h	25h	FFh	43h		

Caso FAR

P34. La función de lenguaje C cuya signatura se indica en el recuadro de la derecha es invocada desde el programa de código máquina que se muestra en el recuadro de la izquierda. En el momento anterior de la llamada, se suponen los siguientes valores del puntero de pila y de los parámetros de la función: **FLAGS=1234h**, **SP = 12**, **n = ABCDh**, **p = 1234h:8765h**, **c = EFh**,

```

43FF:25E3 E8F6FF    call _fun
43FF:25E6 B8004C    mov ax, 4C00h

```

```
fun ( int n, int* p, char c );
```

Indicar el valor de las 16 posiciones iniciales de la pila en el momento de ejecutarse la primera instrucción de código máquina de la función **fun**, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**). Los valores desconocidos de la pila deben dejarse en blanco.

Caso NEAR:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
				E6h	25h	CDh	ABh	65h	87h	EFh	00h				

Caso FAR:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
E6h	25h	FFh	43h	CDh	ABh	65h	87h	34h	12h	EFh	00h				

P35. Escribir en ensamblador de 8086 un procedimiento lejano `Abrir_Lectura` que invoque a la función `fopen` de la librería de C, cuya signature se reproduce a continuación. Se considera que la librería de C está compilada en **modelo largo** (*large*). El **nombre del fichero y el modo de apertura están almacenados como cadenas ASCIIZ en las variables globales `fichero` y `modo`** respectivamente. El procedimiento ha de **almacenar en la variable global `descriptor` el descriptor de fichero retornado por `fopen`**. Las tres variables globales son accesibles a través del registro **DS**. Se valorará la eficiencia del código.

```

FILE * fopen ( const char * filename, const char * mode );

fichero db "datos.csv", 0, 256 dup (?)
modo db "r", 0
descriptor dw ?, ?

extrn _fopen:FAR

Abrir_Lectura PROC FAR

    push ax dx

    ; Apila dirección larga de modo
    push ds                      ; Apila segmento de modo
    mov ax, offset modo
    push ax

    ; Apila dirección larga de fichero
    push ds                      ; Apila segmento de modo
    mov ax, offset fichero
    push ax

    call _fopen

    add sp, 8                    ; Elimina parámetros de la pila

    ; fopen retorna en dx:ax la dirección larga del descriptor
    mov ds:descriptor, ax
    mov ds:descriptor[2], dx

    pop dx ax
    ret
Abrir_Lectura ENDP

```

P36. Escribir en ensamblador de 8086 un procedimiento lejano `Nombre_Fichero_C` que **almacene en la variable `fichero` del problema anterior el nombre de un fichero pasado como primer argumento** en una invocación del programa tal como la mostrada a continuación. El nombre del fichero ha de almacenarse en la variable como cadena ASCIIZ. La variable global es accesible a través del registro **DS**, mientras que el PSP es accesible a través del registro **ES**. El procedimiento ha de **retornar en AX un 1 si el fichero indicado tiene extensión `.c` y un 0 si no se ha indicado ningún fichero o el fichero indicado no tiene extensión `.c`**. Se valorará la eficiencia del código.

```
> programa codigo.c
```



```

Nombre_Fichero_C PROC FAR
    push bx si di

    ; 82h = Offset de 1er carácter de nombre fichero en PSP
    mov si, 82h
    mov ax, 0    ; Por defecto no tiene extensión .c
    mov di, 0    ; Índice a la cadena fichero

busca_punto: mov bl, es:[si]
             cmp bl, 13
             je final          ; Encuentra final de línea (13)
             mov ds:fichero[di], bl    ; Copia carácter a fichero
             inc si
             inc di
             cmp bl, '.'
             jne busca_punto

             ; registro SI apunta después del punto
hay_punto:  cmp byte ptr es:[si], 'c'
             jne final          ; No es ".c"
             mov bl, es:[si+1]    ; Lee carácter después de .c
             cmp bl, ' '
             je hay_punto_c      ; Espacio en blanco después de .c
             cmp bl, 13
             jne final          ; Otro carácter tras .c (no es .c)

hay_punto_c: mov ds:fichero[di], 'c'
             mov ds:fichero[di+1], 0 ; Guarda final de cadena
             mov ax, 1

final:      pop di si bx
             ret

Nombre_Fichero_C ENDP

```

P37. Al inicio de la ejecución de una función invocada desde lenguaje C, se tiene que **SP=8** y que las 16 primeras posiciones de la pila contienen los siguientes valores:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11h	A0h	25h	00h	32h	00h	A2h	E9h	00h	C1h	24h	00h	63h	00h	41h	12h

La signature de dicha función es: `int fun (char p, int n);`

Indicar el valor de los dos parámetros con que esa función fue invocada desde C, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**).

Caso NEAR: p = 24h n = 0063h

Caso FAR: p = 63h n = 1241h

P38. Escribir en ensamblador de 80x86 utilizando instrucciones básicas (sin instrucciones de manipulación de cadenas ni de bucles) la función `sumatorio` de C cuyo código se reproduce a continuación. Esta función calcula de forma recursiva el sumatorio de los `n` primeros números naturales, con `n` siendo el entero de 32 bits que recibe como argumento. Se supone que la función no detecta desbordamiento del resultado y que el programa en C está compilado en **modelo largo**. Se valorará la eficiencia del código.

```

long sumatorio( long n )
{
    if (n == 1) return 1;
    else return n + sumatorio( n-1 );
}

_sumatorio PROC FAR

    push bp
    mov bp, sp

    ; Accede a parámetro de entrada de 32 bits (n)
    mov ax, [bp+6]    ; AX <= Parte baja de n
    mov dx, [bp+8]    ; DX <= Parte alta de n

    cmp dx, 0
    jne noes1         ; n != 1
    cmp ax, 1
    je final          ; n == 1 => Retorna 1 en DX:AX

    ; n != 1

noes1:                ; Decrementa n

    dec ax            ; Decrementa parte baja
    sbb dx, 0         ; Resta acarreo (borrow) a parte alta

    ; Apila n-1 (parte alta primero) y llama recursivamente

    push dx ax
    call _sumatorio   ; Llamada recursiva
    add sp, 4         ; Reequilibra la pila

    ; sumatorio( n-1 ) retornado en DX:AX

    ; DX:AX := DX:AX + n

    add ax, [bp+6]     ; Suma parte baja
    adc dx, [bp+8]     ; Suma parte alta y acarreo

final:               pop bp
                    ret

_sumatorio ENDP

```

P39. Al inicio de la ejecución de una función invocada desde lenguaje C, se tiene que **SP=0** y que las 16 primeras posiciones de la pila contienen los siguientes valores:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11h	A0h	25h	00h	32h	00h	A2h	E9h	00h	C1h	24h	00h	63h	00h	41h	12h

La signatura de dicha función es: `int fun (long p, int n);`

Indicar el valor de los tres parámetros con que esa función fue invocada desde C, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**).

Caso NEAR: p = 00320025h n = E9A2h

Caso FAR: p = E9A20032h n = C100h

P40. Escribir en ensamblador de 80x86 utilizando instrucciones básicas (sin instrucciones de manipulación de cadenas ni de bucles) la función `maximo` de C cuyo código se reproduce a continuación. Esta función determina el valor máximo de una tabla de enteros con signo que recibe como argumento. Se supone que el programa en C está compilado en **modelo pequeño** (*small*). Se valorará la eficiencia del código.

<pre> _maximo PROC NEAR push bp mov bp, sp push bx cx mov bx, [bp+6] ; bx == &tabla mov ax, [bx] ; ax == max mov cx, 1 ; cx == i bucle: cmp cx, [bp+4]; ; ¿i == n? je final add bx, 2 cmp [bx], ax ; ¿tabla[i] > max? jle no_maximo ; tabla[i] <= max mov ax, [bx] ; max = tabla[i] no_maximo: inc cx ; i++ jmp bucle final: pop cx bx ret _maximo ENDP </pre>	<pre> int maximo(int n, int *tabla) { int i, max; max = tabla[0]; for (i=1; i<n; i++) if (tabla[i] > max) max = tabla[i]; return max; } </pre>
---	---

P41. Suponiendo que **SP=8** y que las primeras 16 posiciones del segmento de pila contienen los siguientes valores:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
FFh	A0h	25h	00h	32h	00h	A2h	E9h	00h	C1h	24h	F1h	00h	63h	41h	12h

Indicar el valor de los cuatro registros después de la ejecución del siguiente programa.

```

pop AX      AX = C100h  BX = F124h  CX = C100h  DX = 6300h
pop BX
push AX
pop CX
pop DX

```

P42. Al inicio de la ejecución de una función invocada desde lenguaje C, se tiene que **SP=0** y que las 16 primeras posiciones de la pila contienen los siguientes valores:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11h	A0h	25h	00h	32h	00h	A2h	E9h	00h	C1h	24h	F1h	00h	63h	41h	12h

La signatura de dicha función es: `int fun(char c, int n, char *s);`

Indicar el valor de los tres parámetros con que esa función fue invocada desde C, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**).

Caso NEAR: `c = 25h` `n = 0032h` `s = E9A2h`

Caso FAR: `c = 32h` `n = E9A2h` `s = F124h:C100h`

P43. Escribir en ensamblador de 80x86 el código necesario para **invocar a la función Maximo** de C, cuya signatura se reproduce a continuación, así como para **reequilibrar la pila** después de dicha invocación. Los parámetros de la invocación son una tabla de enteros y su tamaño. Se supone que el programa de C está compilado en **modelo largo (large)**. Se valorará la eficiencia del código.

```

int Maximo( int n, int *tabla );

int Tabla[10] = {2, 4, 5, 7, 1, 20, -2, 8, -100, 9};

Maximo( 10, Tabla );    // Implementar en ensamblador.

mov ax, SEG _Tabla
push ax
mov ax, OFFSET _Tabla
push ax
mov ax, 10
push ax
call _Maximo
add sp, 6

```

P44. Escribir en ensamblador de 80x86 la función de C que se reproduce en el siguiente recuadro, que calcula el **valor máximo de una tabla de n enteros con signo de 16 bits**. Las variables locales están almacenadas en registros. Se supone que el programa de C está compilado en **modelo largo (large)**. Se valorará la eficiencia del código.

`_Maximo PROC FAR`

```
push bp
mov bp, sp
push bx cx dx si es

mov dx, [bp+6] ; dx == n
les bx, [bp+8] ; es:bx == tabla
mov ax, -32768 ; ax == max
mov cx, 0      ; cx == i
```

`for:`

```
cmp cx, dx      ; i < n?
jge fin_for     ; i >= n

; i < n
mov si, cx
shl si, 1       ; si == i * sizeof(int)
cmp es:[bx][si], ax ; tabla[i] > max?
jle fin_if      ; tabla[i] <= max
```

```
; tabla[i] > max
mov ax, es:[bx][si] ; max = tabla[i]
```

`fin_if:`

```
inc cx      ; i++
jmp for
```

`fin_for:`

```
pop es si dx cx bx
pop bp
ret
```

`_Maximo ENDP`

```
int Maximo( int n, int *tabla )
{
    register int i;
    register int max = -32768;

    for (i=0; i<n; i++)
        if (tabla[i] > max) max = tabla[i];

    return max;
}
```

P45. Suponiendo que **SP=0** y que las primeras 16 posiciones del segmento de pila contienen los siguientes valores:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
FFh	A0h	25h	00h	32h	00h	A2h	E9h	00h	C1h	24h	F1h	00h	63h	41h	12h

Indicar el valor de los cuatro registros después de la ejecución del siguiente programa.

```
pop AX      AX = A0FFh  BX = 0032h  CX = 0025h  DX = A0FFh
pop CX
pop BX
push AX
pop DX
```

P46. Al inicio de la ejecución de una función invocada desde lenguaje C, se tiene que **SP=0** y que las 16 primeras posiciones de la pila contienen los siguientes valores:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11h	A0h	25h	00h	32h	00h	A2h	00h	00h	C1h	24h	F1h	00h	63h	41h	12h

La signatura de dicha función es: `int fun(int n, char c, char *s);`

Indicar el valor de los tres parámetros con que esa función fue invocada desde C, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**).

Caso NEAR: `n = 0025h c = 32h s = 00A2h`

Caso FAR: `n = 0032h c = A2h s = F124h:C100h`

P47. Escribir en ensamblador de 80x86 el código necesario para **invocar a la función Mínimo** de C, cuya signatura se reproduce a continuación, así como para **reequilibrar la pila** después de dicha invocación. Los parámetros de la invocación son una tabla de enteros y su tamaño. Se supone que el programa de C está compilado en **modelo compacto**. Se valorará la eficiencia del código.

```
int Minimo( int n, int *tabla );

int Tabla[10] = {2, 4, 5, 7, 1, 20, -2, 8, -100, 9};

Minimo( 10, Tabla );    // Implementar en ensamblador.

mov ax, SEG _Tabla
push ax
mov ax, OFFSET _Tabla
push ax
mov ax, 10
push ax
call _Minimo
add sp, 6
```

P48. Escribir en ensamblador de 80x86 la función de C que se reproduce en el siguiente recuadro, que calcula el **valor mínimo de una tabla de n enteros con signo de 16 bits**. Las variables locales están almacenadas en registros. Se supone que el programa de C está compilado en **modelo compacto**. Se valorará la eficiencia del código.

_Minimo PROC NEAR

```
push bp
mov bp, sp
push bx cx dx si es

mov dx, [bp+4]    ; dx == n
les bx, [bp+6]    ; es:bx == tabla
mov ax, 32767     ; ax == min
mov cx, 0        ; cx == i
```

```
int Minimo( int n, int *tabla )
{
    register int i;
    register int min = 32767;

    for (i=0; i<n; i++)
        if (tabla[i] < min) min = tabla[i];

    return min;
}
```

```

for:
    cmp cx, dx      ; i < n?
    jge fin_for     ; i >= n

    ; i < n
    mov si, cx
    shl si, 1       ; si == i * sizeof(int)
    cmp es:[bx][si], ax ; tabla[i] < min?
    jge fin_if      ; tabla[i] >= min

    ; tabla[i] < min
    mov ax, es:[bx][si] ; min = tabla[i]
fin_if:
    inc cx          ; i++
    jmp for

fin_for:
    pop es si dx cx bx
    pop bp
    ret

_Minimo ENDP

```

P49. La función de lenguaje C cuya signature se indica en el recuadro de la derecha es invocada desde el programa de código máquina que se muestra en el recuadro de la izquierda. En el momento anterior de la llamada, se suponen los siguientes valores del puntero de pila y de los parámetros de la función: **SP = 12**, **n = 1234h**, **c = ABh**, **p = 4253h:5678h**.

```

4253:0007 E8F6FF call _fun
4253:000A B8004C mov ax, 4C00h

```

```
fun ( int n, char c, char* p );
```

Indicar el valor de las 16 posiciones iniciales de la pila en el momento de ejecutarse la primera instrucción de código máquina de la función `fun`, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**).

Caso NEAR:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
				0Ah	00h	34h	12h	ABh	00h	78h	56h				

Caso FAR:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0Ah	00h	53h	42h	34h	12h	ABh	00h	78h	56h	53h	42h				

P50. Escribir en ensamblador de 80x86 la función `_Sum8U`, que calcula el **sumatorio de enteros sin signo de 8 bits**. El **límite inferior** es un operando que ha de estar previamente almacenado en **BL**. El **límite superior** es un operando que ha de estar previamente almacenado en **BH**. El

resultado ha de ser un entero sin signo de 16 bits que se retornará en **AX**. La función ha de sumar todos los enteros comprendidos entre el límite inferior y el superior, ambos incluidos. Se valorará la eficiencia del código.

```

_Sum8U PROC FAR

    push cx
    mov ax, 0

    cmp bl, bh
    ja final ; Acaba si inferior > superior

    mov ch, 0
    mov cl, bl
    dec cx

seguir: inc cx ; Incrementa índice del sumatorio
        add ax, cx
        cmp cl, bh
        jne seguir

final: pop cx
      ret

_Sum8U ENDP

```

P51. Llamando a la función de sumatorio desarrollada en el problema anterior, escribir en ensamblador de 80x86 la función de C que se reproduce en el siguiente recuadro, que calcula el sumatorio entre dos enteros de 8 bits pasados por referencia. Las variables locales están almacenadas en registros. Se supone que el programa de C está compilado en **modelo largo**. Se valorará la eficiencia del código.

```

_Sumatorio2 PROC FAR

```

```

    push bp
    mov bp, sp
    push bx cx dx si di ds es

    mov ax, 0
    mov bx, 0
    mov cx, 0 ; cx = res
    mov dx, 0 ; dx = tmp

```

```

; Salta @retorno (4 bytes por ser código FAR) y bp (2 bytes)
lds si, [bp+6] ; ds:si := dirección de límite inferior

; Salta @retorno, bp y ler argumento (4 bytes por ser datos FAR)
les di, [bp+10] ; es:di := dirección de límite superior

mov bl, ds:[si] ; bl := *inf
mov dl, es:[di] ; dl := *sup
add dx, bx ; dx := *inf + *sup
sar dx, 1 ; dx := (*inf + *sup) / 2

mov bh, dl ; bh := (*inf + *sup) / 2

```

```

int Sumatorio2 ( char *inf, char *sup )
{
    register int res=0;
    register int tmp;

    tmp = (*inf + *sup) / 2;
    res = Sum8U( *inf, tmp );
    res = res + Sum8U( tmp + 1, *sup );

    return res;
}

```



```

call _Sum8U          ; ax := Sum8U( *inf, (*inf + *sup) /2 )
mov cx, ax           ; cx := Sum8U( *inf, (*inf + *sup) /2 )

mov bl, bh           ; bl := (*inf + *sup) /2;
inc bl              ; bl := (*inf + *sup) /2 + 1
mov bh, es:[di]      ; bh := *sup

call _Sum8U          ; ax := Sum8U((*inf + *sup)/2+1, *sup )
add ax, cx           ; ax := Sum8U( *inf, (*inf + *sup)/2 ) +
                    ; Sum8U((*inf + *sup)/2+1, *sup )

pop es ds di si dx cx bx
pop bp

ret                  ; Devuelve resultado en ax

```

`_Sumatorio2 ENDP`

P52. Al inicio de la ejecución de una función invocada desde lenguaje C, se tiene que **SP=4** y que las 16 primeras posiciones de la pila contienen los siguientes valores:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11h	A0h	25h	00h	E0h	00h	FFh	22h	00h	1Ch	22h	F1h	00h	63h	41h	12h

La signatura de dicha función es: `int fun (int n, long *p);`

Indicar el valor de los tres parámetros con que esa función fue invocada desde C, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**).

Caso NEAR: `n = 22FFh` `p = 1C00h`

Caso FAR: `n = 1C00h` `p = 6300h:F122h`

P53. Escribir en ensamblador de 80x86 la función de C que se reproduce en el siguiente recuadro, que calcula el **sumatorio de los *n* primeros enteros sin signo de 8 bits**. Se supone que el programa de C está compilado en **modelo largo**. Se valorará la eficiencia del código.

```

_Sumatorio8 PROC FAR
push bp
mov bp, sp
mov ax, [bp+6] ; ax = n
cmp ax, 2
jnb fin        ; n < 2

; n >= 2
dec ax         ; ax = n-1
push ax

```

```

int Sumatorio8( unsigned char n )
{
    if ( n < 2) return n;
    else return ( n + Sumatorio8( n-1 ) );
}

```

```

call _Sumatorio8
; ax = Sumatorio8( n-1 )

add sp, 2      ; reequilibra pila
add ax, [bp+6] ; ax = n + Sumatorio8( n-1 )

fin:
pop bp
ret
_Sumatorio8 ENDP

```

P54. Suponiendo que **SP=6** y que las primeras 16 posiciones del segmento de pila contienen los siguientes valores:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12h	AFh	76h	00h	A2h	FDh	A2h	11h	00h	00h	42h	F0h	07h	62h	49h	22h

Indicar el valor de los cuatro registros después de la ejecución del siguiente programa

```

pop AX      AX = 11A2h  BX = 11A2h  CX = 0000h  DX = F042h
push AX
pop BX
pop CX
pop DX

```

P55. Al inicio de la ejecución de una función invocada desde lenguaje C, se tiene que **SP=2** y que las 16 primeras posiciones de la pila contienen los siguientes valores:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12h	AFh	76h	00h	AAh	FDh	A2h	11h	00h	00h	42h	F0h	07h	62h	49h	22h

La signatura de dicha función es: `int fun (int *p, int n, char *c);`

Indicar el valor de los tres parámetros con que esa función fue invocada desde C, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**).

Caso NEAR: p = FDAAh n = 11A2h c = 0000h

Caso FAR: p = 0000h:11A2h n = F042h c = 2249h:6207h

P56. Escribir en ensamblador de 80x86 el código necesario para medir el **tiempo de ejecución en milisegundos** de una subrutina utilizando la **interrupción 1Ah** de la BIOS (se adjunta un extracto de la documentación de Ralph Brown). El código solicitado consta de **dos partes**: unas líneas de código que se ejecutan antes de la subrutina (A) y leen el tiempo inicial, y unas líneas de código que se ejecutan detrás de la subrutina (B) y leen el tiempo final y calculan en **DX:AX los milisegundos transcurridos** entre ambos tiempos. Se supone que el **tiempo final es siempre superior al inicial**, y que la duración de la subrutina es inferior a una hora (**menos de 65535 ticks de reloj**, con **1 tick cada 55 milisegundos**). Se valorará la eficiencia del código.

```

; (A) Lee tiempo inicial.

mov ah, 0
int 1Ah
mov di, dx      ; di := dx_inicial

call Subrutina

; (B) Lee tiempo final y calcula
;      milisegundos en DX:AX.

int 1Ah
sub dx, di      ; Solo resta
mov ax, 55      ; palabra baja porque diferencia es menor de 65535.
mul dx          ; dx:ax := (dx_final - dx_inicial) * 55

```

TIME - GET SYSTEM TIME

AH = 00h

Return:

CX:DX = number of clock ticks since midnight

AL = midnight flag, nonzero if midnight passed since time last read

Category: [Bios](#) - [Int 1Ah](#) - [T](#)

P57. Escribir en ensamblador de 80x86 utilizando instrucciones básicas (sin instrucciones de manipulación de cadenas ni de bucles) y sin variables auxiliares la función `Transpose` de C cuyo código se reproduce a continuación. Esta función transpone una matriz de enteros bidimensional de *rows* filas y *cols* columnas. Se supone que el programa en C está compilado en **modelo pequeño**. Se valorará la eficiencia del código.

```

void Transpose( int** A, int rows, int cols, int** Res )
{
    register int row, col;

    for (row=0; row<rows; row++)
        for (col=0; col<cols; col++)
            Res[col][row] = A[row][col];
}

```

`_Transpose PROC NEAR`

```

    push bp
    mov bp, sp
    push ax bx dx si di
    mov si, 0      ; si == row := 0
for1: cmp si, [bp+6] ; row < rows?
    jge finfor1    ; row >= rows
    shl si, 1      ; si := row * 2
    mov bx, [bp+4] ; bx := A
    mov dx, [bx][si] ; dx := A[row]

    mov di, 0      ; di == col := 0
for2: cmp di, [bp+8] ; col < cols?
    jge finfor2    ; col >= cols
    shl di, 1      ; di := col * 2
    mov bx, dx     ; bx := A[row]
    mov ax, [bx][di] ; ax := A[row][col]

    mov bx, [bp+10] ; bx := Res
    mov bx, [bx][di] ; bx := Res[col]
    mov [bx][si], ax ; Res[col][row] := ax
    sar di, 1      ; di := col
    inc di         ; col++
    jmp for2

```

`finfor2:`

```

    sar si, 1      ; si := row
    inc si         ; row++
    jmp for1

```

`finfor1:`

```

    pop di si dx bx ax
    pop bp
    ret

```

`_Transpose ENDP`

P58. La función de lenguaje C cuya signatura se indica en el recuadro de la derecha es invocada desde el programa de código máquina que se muestra en el recuadro de la izquierda. En el momento anterior de la llamada, se suponen los siguientes valores del puntero de pila y de los parámetros de la función: **SP = 14**, **n = 1234h**, **c = ABh**, **p = 3524h:5678h**.

```
3524:0000 E8F6FF call _fun
3524:0003 B8004C mov ax, 4C00h
```

```
fun ( int n, char** p, char c );
```

Indicar el valor de las 16 posiciones iniciales de la pila en el momento de ejecutarse la primera instrucción de código máquina de la función **fun**, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**).

Caso NEAR:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
						03h	00h	34h	12h	78h	56h	ABh	00h		

Caso FAR:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		03h	00h	24h	35h	34h	12h	78h	56h	24h	35h	ABh	00h		

P59. Llamando a la función **malloc** de C, escribir en ensamblador de 80x86 la función de C **Insert**, que se reproduce en el siguiente recuadro, que inserta un elemento en un árbol binario de búsqueda (BST) dinámico. Las variables locales están almacenadas en registros. Se supone que el programa de C está compilado en **modelo corto**. Se valorará la eficiencia del código.

_Insert PROC NEAR

```
push bp
mov bp, sp
push bx cx
```

```
; bx == tmp := t
mov bx, [bp+4]
mov cx, [bp+6]      ; cx == k
cmp bx, 0           ; t==0?
jne nocero         ; no malloc
```

```
mov ax, 6           ; ax := sizeof(tBST)
push ax
call _malloc
```

```
typedef struct tBST {
    int k; struct tBST *l, *r;
} tBST;

void* malloc( unsigned int );

tBST* Insert( tBST* t, int k ){
    tBST* tmp = t;

    if (t == 0){
        tmp = (tBST*) malloc( sizeof(tBST) );
        tmp->k = k;
        tmp->l = tmp->r = 0;
    }
    else if (k < t->k)
        t->l = Insert( t->l, k );
    else t->r = Insert( t->r, k );
    return tmp;
}
```

```

        add sp, 2          ; Equilibrar pila
        mov bx, ax         ; tmp := malloc()
        mov [bx], cx       ; tmp->k := k
        mov WORD PTR [bx+2], 0 ; tmp->l := 0
        mov WORD PTR [bx+4], 0 ; tmp->r := 0
        jmp fin

nocero:  push cx           ; Apilar k
        cmp cx, [bx]      ; k < t->k
        jge elsif        ; k >= t->k

        push [bx+2]       ; Apilar t->l
        call _Insert
        add sp, 4         ; Equilibrar pila
        mov [bx+2], ax    ; t->l := Insert()
        jmp fin

elsif:  push [bx+4]       ; Apilar t->r
        call _Insert
        add sp, 4         ; Equilibrar pila
        mov [bx+4], ax    ; t->r := Insert()

fin:    mov ax, bx        ; retornar tmp

        pop cx bx
        pop bp

        ret

_Insert ENDP

```

P60. Al inicio de la ejecución de una función invocada desde lenguaje C, se tiene que **SP=2** y que las 16 primeras posiciones de la pila contienen los siguientes valores:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12h	FAh	67h	00h	AAh	FDh	A1h	11h	00h	00h	42h	0Fh	74h	62h	94h	33h

La signatura de dicha función es: `int fun (char *c, int *p, int n);`

Indicar el valor de los tres parámetros con que esa función fue invocada desde C, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**).

Caso NEAR: c = FDAAh p = 11A1h n = 0000h

Caso FAR: c = 0000h:11A1h p = 6274h:0F42h n = 3394h

P61. Escribir en ensamblador de 80x86 la función de C `Inorder`, que se reproduce en el siguiente recuadro, que realiza un recorrido en orden de un árbol binario de búsqueda (BST) dinámico. Se supone que el programa de C está compilado en **modelo largo**. Se valorará la eficiencia del código.

```

_Inorder PROC FAR
    push bp
    mov bp, sp
    push bx
    les bx, [bp+6]
    ; bx = OFFSET t, es = SEG t
    mov bp, es
    cmp bp, 0
    jnz seguir      ; es != 0
    cmp bx, 0
    je final        ; es = bx = 0
    ; t != 0
seguir: push es:[bx+4] ; apilar SEG t->l
        push es:[bx+2] ; apilar OFFSET t->l
        call _Inorder
        add sp, 4
        push es:[bx]   ; apilar t->k
        call _Print
        add sp, 2
        push es:[bx+8] ; apilar SEG t->r
        push es:[bx+6] ; apilar OFFSET t->r
        call _Inorder
        add sp, 4
final:  pop es bx
        pop bp
        ret
_Inorder ENDP

```

```

typedef struct tBST {
    int k;
    struct tBST *l, *r;
} tBST;

void Print( int );

void Inorder( tBST* t ) {
    if (t == 0) return;
    else {
        Inorder( t->l );
        Print( t->k );
        Inorder( t->r );
    }
}

```

P62. Suponiendo que **CS=3000h**, **DS=324Ah**, **ES=324Bh**, **SS=324Ah**, **BP=0006h**, **SI=0004h** y **DI=24A0h**, Indicar el valor del **registro AX** tras ejecutar cada una de las instrucciones siguientes (**independientes entre sí**), dado el volcado de memoria adjunto. Expresar los **dígitos hexadecimales desconocidos de AX con un ‘?’**.

```

324A:0000 23 4E 21 AA FF DD 1A 6E 21 A0 01 33 12 00 98 7E
324A:0010 1B 22 00 00 1F C5 4F 24 02 FF 4D E5 11 AA 23 00

```

mov AL, DS: [SI][BP]	AX = ??01h
mov AX, CS: 20[DI]	AX = C51Fh
mov AH, SS: [BP][SI]	AX = 01??h
mov AX, ES: [0005h]	AX = 4FC5h
mov AL, CS: [024Fh]	AX = ????h

P63. Indicar el vector de la **interrupción no enmascarable (NMI)** dado el siguiente volcado de memoria.

```

0000:0000 54 02 CF 15 CE 01 CF 15 04 00 70 00 D7 01 CF 15
0000:0010 04 00 70 00 30 00 00 C8 30 00 00 C8 30 00 00 C8

```

Segmento = 0070h Offset = 0004h

P64. Al inicio de la ejecución de una función invocada desde lenguaje C, se tiene que **SP=0** y que las 16 primeras posiciones de la pila contienen los siguientes valores:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A1h	B0h	7Fh	11h	AAh	00h	A2h	11h	F1h	00h	F0h	43h	34h	12h	33h	56h

La signatura de dicha función es: `int fun (long *p, char c, int n);`

Indicar el valor de los tres parámetros con que esa función fue invocada desde C, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**).

Caso NEAR: `p = 117Fh` `c = AAh` `n = 11A2h`

Caso FAR: `p = 11A2h:00AAh` `c = F1h` `n = 43F0h`

P65. Escribir en ensamblador de 80x86 utilizando instrucciones básicas (sin instrucciones de manipulación de cadenas ni de bucles) y sin variables auxiliares la función `Rand1000` de C, cuyo código se reproduce a continuación. Esta función retorna un número entero pseudo-aleatorio entre 1 y 1000. Para ello invoca a la función `rand` de C. Se supone que el programa en C está compilado en **modelo pequeño** (*small*). Se valorará la eficiencia del código.

```
_Rand1000 PROC NEAR
    push bx dx
    call _rand

    ; ax = rand()

    mov dx, 0
    mov bx, 1000
    div bx          ; dx = dx:ax % 1000
    inc dx          ; dx = dx:ax % 1000 + 1
    mov ax, dx
    pop dx bx
    ret
_Rand1000 ENDP
```

```
unsigned int rand (void);

unsigned int Rand1000()
{
    return (rand() % 1000) + 1; // % (módulo)
}
```

P66. Escribir en ensamblador de 80x86 utilizando instrucciones básicas (sin instrucciones de manipulación de cadenas ni de bucles) y sin variables auxiliares la función `MulU8` de C cuyo código se reproduce a continuación. Esta función multiplica dos enteros de 8 bits sin signo. Se supone que el programa en C está compilado en **modelo medio** (*medium*). Se valorará la eficiencia del código.

```
unsigned int MulU8( unsigned char x, unsigned char y )
{
    register unsigned int res = 0;

    while (y != 0)
    {
        if (y & 1) res = res + x;
        y = y >> 1;    // Desplaza un bit a la derecha
        x = x << 1;    // Desplaza un bit a la izquierda
    }
    return res;
}
```

```

_MulU8 PROC FAR
    push bp
    mov bp, sp
    push bx cx
    mov ax, 0          ; ax == res = 0
    mov bx, [bp+6]     ; bx == x
    mov cx, [bp+8]     ; cx == y
while:
    cmp cx, 0          ; y == 0?
    je end_while       ; y == 0 (end while)
    test cx, 1         ; y & 1
    jz end_if          ; y & 1 == 0 (end if)
    add ax, bx         ; res = res + x
end_if:
    shr cx, 1         ; y = y >> 1
    shl bx, 1         ; x = x << 1
    jmp while
end_while:
    pop cx bx
    pop bp
    ret
_MulU8 ENDP

```

P67. Al inicio de la ejecución de una función invocada desde lenguaje C, se tiene que **SP=2** y que las 16 primeras posiciones de la pila contienen los siguientes valores:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11h	A0h	25h	FFh	32h	12h	A2h	00h	30h	00h	F3h	1Dh	56h	4Ah	41h	32h

La signatura de dicha función es: `int fun(int n, char c, long *s);`

Indicar el valor de los tres parámetros con que esa función fue invocada desde C, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**).

Caso NEAR: **n = 1232h c = A2h s = 0030h**

Caso FAR: **n = 00A2h c = 30h s = 4A56h:1DF3h**

P68. Escribir en ensamblador de 80x86 la función de C `Mean` que se reproduce en el siguiente recuadro, que calcula la media de los elementos de un vector de N bytes con signo en una ventana de tamaño `WSIZE` centrada en la posición `x`. Las variables locales están almacenadas en registros. Se supone que el programa en C está compilado en **modelo largo (large)**. Se valorará la eficiencia del código.


```

_Mean PROC FAR
    push bp
    mov bp, sp
    push bx cx dx si di es

    les bx, [bp+6]    ; es:bx == array
    mov di, [bp+10]   ; di == x

    mov dx, -WSIZE/2  ; dx == row
    mov ax, 0         ; ax == acum
    mov cx, 0         ; cx == count

for:  cmp dx, WSIZE/2 ; row <= WSIZE/2?
      jg end_for     ; row > WSIZE/2 => end_for

      mov si, dx      ; si == r
      add si, di      ; r = row + x

      cmp si, 0       ; r >= 0?
      jl end_if      ; r < 0 => end_if
      cmp si, N       ; r < N?
      jge end_if     ; r >= N => end_if

      add al, es:[bx][si]
      adc ah, 0       ; acum += array[r]
      inc cx         ; count++

end_if: inc dx        ; row++
      jmp for

end_for:
    cmp cx, 0        ; count == 0?
    jle elif        ; count <= 0
    mov dx, 0        ; dx:ax == acum

    ; extiende signo ax a dx (sin llamar a CWD)
    test ax, 8000h
    jz divide
    dec dx           ; dx = -1

divide: idiv cx       ; ax = acum / count
      jmp final

elif: mov ax, 0

final:  pop es di si dx cx bx
      pop bp
      ret

_Mean ENDP

```

```

char array[N];

int Mean( char *array, int x )
{
    register int row, r;
    register int count = 0;
    register int acum = 0;

    for (row = -WSIZE/2; row <= WSIZE/2; row++)
    {
        r = row+x;

        if (r >= 0 && r < N)
        {
            acum = acum + array[r];
            count++;
        }
    }

    if (count > 0) return( acum / count );
    else return 0;
}

```

P69. La función de lenguaje C cuya signature se indica en el recuadro de la derecha es invocada desde el programa de código máquina que se muestra en el recuadro de la izquierda. En el momento anterior de la llamada, se suponen los siguientes valores del puntero de pila y de los parámetros de la función: **FLAGS=5678h**, **SP = 16**, **n = ABCDh**, **p = 43FF:1234h**, **c = FFh**,

```
43FF:25E3 E8F6FF call _fun
43FF:25E6 B8004C mov ax, 4C00h
```

```
fun (char c, int n, int* p );
```

Indicar el valor de las 16 posiciones iniciales de la pila al ejecutarse la primera instrucción de código máquina de la función **fun**, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**). Los valores desconocidos de la pila han dejarse en blanco.

Caso NEAR:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
								E6h	25h	FFh	00h	CDh	ABh	34h	12h

Caso FAR:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
				E6h	25h	FFh	43h	FFh	00h	CDh	ABh	34h	12h	FFh	43h

P70. Escribir en ensamblador de 80x86 una rutina cercana que llame a la función de C `Dichotomic_Search`, cuya signature se indica en el siguiente recuadro. La rutina recibirá la dirección corta de la tabla (*array*) en el registro BX, el tamaño de la tabla (*size*) en el registro CX y el valor (*v*) en el registro AX. La rutina devolverá el resultado de la función de C en el registro AX. Se supone que la función de C está compilada en **modelo pequeño (small)**. Se valorará la eficiencia del código.

```
int Dichotomic_Search( int *array, int size, int v );
```

```
fun PROC NEAR
    push ax        ; Apila v
    push cx        ; Apila size
    push bx        ; Apila array
    call _Dichotomic_Search
    add sp, 6
    ret
fun ENDP
```

P71. Escribir en ensamblador de 80x86 la función de C `Dichotomic_Search` que se reproduce en el siguiente recuadro, que busca un valor dado dentro de una tabla de enteros con signo. Las variables locales han de almacenarse en registros. Se supone que el programa en C está compilado en **modelo pequeño (small)**. Se valorará la eficiencia del código.

_Dichotomic_Search PROC NEAR

```

    push bp
    mov bp, sp
    push bx cx dx si di

    mov bx, [bp+4]    ; bx == array
    mov ax, [bp+8]    ; ax == v
    mov cx, 0         ; cx == min
    mov dx, [bp+6]
    dec dx            ; dx == max
    mov di, 0         ; di == found

while: cmp dx, cx      ; max >= min?
      jl endwhile     ; max < min
      cmp di, 0       ; found = 0?
      jne endwhile   ; found != 0

      mov si, cx      ; si == i := min
      add si, dx      ; si := min + max
      sar si, 1       ; si := (min + max)/2;

      shl si, 1       ; si := i*2 (enteros ocupan 2 bytes)
      mov bp, [bx][si] ; bp == st := array[i]
      sar si, 1       ; si := i

      sub bp, ax      ; st == bp := array[i] - v
      jnz else_if     ; st != 0
      mov di, 1       ; di == found := 1
      jmp while

else_if: jle else1     ; bp == st <= 0
        mov dx, si
        dec dx        ; dx == max := i-1;
        jmp while

else1: mov cx, si
      inc cx          ; cx == min := i+1
      jmp while

endwhile: cmp di, 0    ; di == final = 0?
        jne return    ; final != 0
        mov si, -1

return: mov ax, si     ; ax := i
        pop di si dx cx bx
        pop bp
        ret

```

_Dichotomic_Search ENDP

```

int Dichotomic_Search( int *array, int size, int v ) {
    register int min, max, i, st;
    register char found;

    min = 0;
    max = size - 1;
    found = 0;

    while (max >= min && found == 0){
        i = (min + max) / 2;
        st = array[i] - v;

        if (st == 0) found = 1;
        else if (st > 0) max = i-1;
        else min = i+1;
    }
    if (found == 0) i = -1;
    return( i );
}

```

P72. Suponiendo que **SP=4** y que las primeras 16 posiciones del segmento de pila contienen los siguientes valores:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12h	AFh	76h	00h	A2h	FDh	A2h	11h	00h	00h	42h	F0h	07h	62h	49h	22h

Indicar el valor de los cinco registros después de la ejecución del siguiente programa

```

pop AX          AX= 0000h BX= 11A2h CX= FDA2h DX= FDA2h SI= 2249h
push AX
push AX
pop DX
pop CX
pop BX
pop AX
add sp, 4
pop SI

```

P73. Al inicio de la ejecución de una función invocada desde lenguaje C, se tiene que **SP=2** y que las 16 primeras posiciones de la pila contienen los siguientes valores:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A1h	B0h	7Fh	11h	AAh	00h	A2h	11h	F1h	00h	F0h	43h	34h	00h	33h	56h

La signatura de dicha función es: `int fun (char *p, int n, char c);`

Indicar el valor de los tres parámetros con que esa función fue invocada desde C, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**).

Caso NEAR: p = 00AAh n = 11A2h c = F1h

Caso FAR: p = 00F1h:11A2h n = 43F0h c = 34h

P74. La siguiente función en lenguaje ensamblador de 80x86, que implementa la función `strlen` de C, cuya signatura se reproduce a continuación, tiene varios errores. Proponer una versión correcta de la misma función haciendo el **menor número de cambios**. Sólo es necesario reescribir las líneas erróneas. Esta función retorna la longitud de la cadena de caracteres que recibe como argumento en formato ASCII. Se supone que el programa de C está compilado en **modelo largo** (*large*).

```
int strlen (char *s);
```

```

_strlen PROC NEAR

    push bp
    mov bp, sp

    push ds bx

    lea bx, 6[bp]
    mov ax, 0

loop1:  cmp [bx], 0
        je end
        inc ax
        add bx, 2
        jmp loop1

end:    pop ds bx
        pop bp

    ret

_strlen ENDP

```

```

_strlen PROC FAR

    push bp
    mov bp, sp

    push ds bx

    lds bx, 6[bp]
    mov ax, 0

loop1:  cmp BYTE PTR [bx], 0
        je end
        inc ax
        inc bx
        jmp loop1

end:    pop bx ds
        pop bp

    ret

_strlen ENDP

```

P75. Escribir en ensamblador de 80x86 utilizando instrucciones básicas (sin instrucciones de manipulación de cadenas ni de bucles) y sin variables auxiliares la función `MulU16` de C cuyo código se reproduce a continuación. Esta función multiplica dos enteros de 16 bits sin signo. Se supone que el programa en C está compilado en **modelo pequeño** (*small*). Se valorará la eficiencia del código.

```

unsigned long MulU16( unsigned int x, unsigned int y )
{
    register unsigned long res = 0, xx = x;

    while (y != 0)
    {
        if (y & 1) res = res + xx;
        y = y >> 1;    // Desplaza un bit a la derecha
        xx = xx << 1;  // Desplaza un bit a la izquierda
    }
    return res;
}

```

```

_MulU16 PROC NEAR
    push bp
    mov bp, sp
    push bx cx si
    mov dx, 0
    mov ax, 0        ; dx:ax == res = 0
    mov bx, [bp+4]
    mov si, 0        ; si:bx == xx = x
    mov cx, [bp+6]   ; cx == y

while:
    cmp cx, 0        ; y == 0?
    je end_while    ; y == 0 (end while)
    test cx, 1       ; y & 1

```

```

        jz end_if          ; y & 1 == 0 (end if)
        add ax, bx
        adc dx, si        ; res = res + xx
end_if:
        shr cx, 1         ; y = y >> 1
        clc               ; carry = 0
        rcl bx, 1
        rcl si, 1         ; xx = xx << 1
        jmp while
end_while:
        pop si cx bx
        pop bp
        ret
_MulU16 ENDP

```

P76. Al inicio de la ejecución de una función invocada desde lenguaje C, se tiene que **SP=0** y que las 16 primeras posiciones de la pila contienen los siguientes valores:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11h	A0h	25h	FFh	32h	12h	A2h	00h	30h	00h	F3h	1Dh	56h	4Ah	41h	32h

La signatura de dicha función es: `int fun(int n, char *c, long s);`

Indicar el valor de los tres parámetros con que esa función fue invocada desde C, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**). (2 puntos)

Caso NEAR: n = **FF25h** c = **1232h** s = **003000A2h**

Caso FAR: n = **1232h** c = **0030h:00A2h** s = **4A561DF3h**

P77. Escribir en ensamblador de 80x86 la función de C `Fibonacci` que se reproduce en el siguiente recuadro, que calcula el elemento i-ésimo de la sucesión de Fibonacci. Se supone que el programa en C está compilado en **modelo pequeño (small)**. Se valorará la eficiencia del código.

```

_Fibonacci PROC NEAR
    push bp
    mov bp, sp
    push bx

    mov ax, [bp+4]      ; ax == i
    cmp ax, 1
    jbe final          ; i <= 1

    dec ax
    push ax             ; Apila parámetro i-1 (en bp-4)
    call _Fibonacci     ; ax = Fibonacci( i-1 );
    mov bx, ax          ; bx = Fibonacci( i-1 );
    dec WORD PTR [bp-4] ; Parámetro apilado == i-2 (en bp-4)
    call _Fibonacci     ; ax = Fibonacci( i-2 );

```

```

        add sp, 2                ; Desapila parámetro

        add ax, bx                ; ax == Fibonacci( i-2 ) + Fibonacci( i-1 )

final: pop bx bp
        ret

_Fibonacci ENDP

```

P78. Escribir en ensamblador de 80x86 el código necesario para **poner a 1 los bits 1, 3, 5, 7** del **registro AH**, dejando todos los demás bits de ese registro intactos, y **poner a 0 los bits 0, 2, 4 y 6** del **registro AL**, dejando intactos sus demás bits. Se valorará la eficiencia del código.

```

or ah, 10101010b ; AAh
and al, 10101010b ; AAh

```

P79. Escribir en ensamblador de 80x86 el código en C que se reproduce a continuación, incluyendo la declaración de la variable global, la cadena de caracteres en formato ASCII y la llamada a las funciones de C `is_prime` y `printf`. Se supone que el programa en C está compilado en **modelo largo (large)**. Se valorará la eficiencia del código.

```

_n DW 1000
string DB "%d %d\n", 0

```

```

unsigned int n = 1000;
char is_prime( unsigned int n );

printf( "%d %d\n", n, is_prime(n) );

```

```

mov bx, _n
push bx                ; Apila parámetro de is_prime (n)
call _is_prime
add sp, 2              ; Equilibra pila

push ax                ; Apila tercer parámetro de printf (resultado de is_prime)
push bx                ; Apila segundo parámetro de printf (n)

; Apila primer parámetro de printf (puntero largo a cadena)
mov ax, SEG string
push ax
mov ax, OFFSET string
push ax

call _printf
add sp, 8

```

P80. Escribir en ensamblador de 80x86 la función de C `is_prime` que se reproduce en el siguiente recuadro, que devuelve un booleano indicando si el entero que recibe como parámetro es un número primo o no. Se supone que el programa en C está compilado en **modelo largo (large)**. Las variables locales han de almacenarse en registros. Se valorará la eficiencia del código.

```

_is_prime PROC FAR
    push bp
    mov bp, sp
    push bx cx dx si di

    mov si, 0            ; si == res

```

```

mov di, 2      ; di == divs
mov bx, [bp+6] ; bx == n

cmp bx, 2
jbe else_if    ; n <= 2

mov cx, bx
shr cx, 1 ; cx == half := n/2

dowhile:
mov dx, 0
mov ax, bx      ; dx:ax := n
div di          ; dx := n % divs, ax := n / divs

mov si, 0        ; res := 0
cmp dx, 0        ; n % divs != 0?
je false        ; n % divs == 0 (res == 0)
inc si          ; n % divs != 0 => res := 1

false:
inc di          ; divs++

cmp si, 0        ; res != 0?
je final        ; res == 0
cmp di, cx       ; divs <= half?
jbe dowhile     ; res != 0 && divs <= half
jmp final

else_if:
jne final       ; n != 2
mov si, 1       ; n == 2 => res := 1

final:
mov ax, si      ; ax := res
pop di si dx cx bx
pop bp
ret

_is_prime ENDP

```

```

char is_prime( unsigned int n )
{
    register char res = 0;
    register unsigned int divs = 2;
    register unsigned int half;

    if (n > 2)
    {
        half = n/2;
        do
        {
            res = (n % divs) != 0; // % == resto
            divs++;
        } while (res != 0 && divs <= half);
    }
    else if (n == 2) res = 1;

    return res;
}

```

P81. Suponiendo que **SS=424Dh**, **SP=8**, **AX=CAFEh** y **BX=5678h**, indicar el **valor hexadecimal de los 16 primeros bytes del segmento SS** una vez ejecutado el siguiente programa.

```

push AX
push AX
pop BX
push BX

```


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
				FEh	CAh	FEh	CAh								

P82. Al inicio de la ejecución de una función invocada desde lenguaje C, se tiene que **SP=4** y que las 16 primeras posiciones de la pila contienen los siguientes valores:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11h	A0h	25h	00h	32h	00h	FEh	CAh	B1h	00h	F0h	A2h	63h	00h	4Fh	21h

La signatura de dicha función es: `int fun (int *p, char c, int n);`

Indicar el valor de los tres parámetros con que esa función fue invocada desde C, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**).

Caso NEAR: p = CAFEh c = B1h n = A2F0h

Caso FAR: p = A2F0h:00B1h c = 63h n =214Fh

P83. La siguiente función en lenguaje ensamblador de 80x86, que implementa la función `Sum` de C reproducida a continuación, tiene varios errores. Proponer una versión correcta de la misma función haciendo el **menor número de cambios**. Sólo se deben reescribir las líneas erróneas. Se supone que el programa de C está compilado en **modelo largo** (*large*).

```
unsigned int Sum( unsigned char n )
{
    if (n < 2) return n;
    else return ( n + Sumatorio8( n-1 ) );
}
```

```

_Sum PROC FAR
    push bp
    mov sp, bp

    mov ax, [bp+4]
    cmp ax, 2
    jle fin

    dec ax
    push ax
    call _Sum

    mov ax, [bp+6]

fin: pop bp
    ret
_Sum ENDP

```

```

_Sum PROC FAR
    push bp
    mov bp, sp

    mov ax, [bp+6]
    cmp ax, 2
    jb fin

    dec ax
    push ax
    call _Sum

    add sp, 2
    add ax, [bp+6]

fin: pop bp
    ret
_Sum ENDP

```

P84. Escribir en ensamblador de 80x86 utilizando instrucciones básicas (sin instrucciones de manipulación de cadenas ni de bucles) y sin variables auxiliares la función `Histogram_String_1KB` de C cuyo código se reproduce a continuación. Esta función obtiene el histograma de una cadena de caracteres dada. Cada carácter es un código ASCII. El tamaño máximo de la cadena es de 1024 caracteres. El final de la cadena se marca con el código 0. Se supone que el programa en C está compilado en **modelo pequeño** (*small*). Se valorará la eficiencia del código.

```

unsigned int Histogram_String_1KB( char *string, unsigned int *histo )
{
    register unsigned int i = 0;
    register unsigned int count = 0;

    while (i <= 1023 && string[i] > 0)
    {
        histo[ string[i] ]++;
        count++;
        i++;
    }
    return count;
}

```

`_Histogram_String_1KB PROC NEAR`

```

    push bp
    mov bp, sp

    push bx cx dx si di

    mov si, 0          ; si == i
    mov ax, 0          ; ax == count
    mov dx, 0          ; dx == string[i]
    mov bx, [bp+4]      ; bx == string
    mov cx, [bp+6]      ; cx == histo

```

```

while: cmp si, 1023      ; i <= 1023?
      ja end_while     ; i > 1023

      mov dl, [bx][si]   ; dx := string[i]
      cmp dl, 0         ; string[i] > 0?
      jle end_while     ; string[i] <= 0

      mov di, dx
      shl di, 1         ; di := string[i] * 2

      add di, cx         ; di := &(amp;histo[string[i]])
      inc WORD PTR [di] ; histo[string[i]]++

      inc ax            ; count++
      inc si            ; i++

      jmp while

end_while: pop di si dx cx bx
          pop bp
          ret

_Histogram_String_1KB ENDP

```

P85. Al inicio de la ejecución de una función invocada desde lenguaje C, se tiene que **SP=0** y que las 16 primeras posiciones de la pila contienen los siguientes valores:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
22h	A0h	52h	00h	A5h	00h	1Eh	00h	BFh	00h	F1h	B1h	35h	00h	42h	21h

La signatura de dicha función es: `int fun (int p, char c, char *n);`

Indicar el valor de los tres parámetros con que esa función fue invocada desde C, tanto cuando todas las direcciones son cercanas (**NEAR**), como cuando son lejanas (**FAR**).

Caso NEAR: p = 0052h c = A5h n = 001Eh

Caso FAR: p = 00A5h c = 1Eh n = B1F1h:00BFh

P86. Escribir en ensamblador de 80x86 utilizando instrucciones básicas y sin variables auxiliares la función `Porcentaje_Vocales` de C cuyo código se reproduce a continuación. Esta función calcula el porcentaje de vocales de una cadena de caracteres. Cada carácter es un código ASCII de una letra mayúscula. El tamaño máximo de la cadena es de 64KB, incluyendo el final de la cadena. Se supone que el programa en C está compilado en **modelo pequeño** (*small*). Se valorará la eficiencia del código.

```

unsigned int Porcentaje_Vocales( char *buf )
{
    register unsigned int vocales, i;
    register char c;

    vocales = i = 0;

    c = buf[i];
    while (c != 0)
    {
        if (c=='A' || c=='E' || c=='I' || c == 'O' || c == 'U')
            vocales++;
        i++;
        c = buf[i];
    }

    return vocales * 100 / i;
}

```

```

_Porcentaje_Vocales PROC NEAR
    push bp
    mov bp, sp

    push bx cx dx si

    mov bx, [bp+4]    ; bx == buf
    mov si, 0         ; si == i
    mov ax, 0         ; ax == vocales

    mov cl, [bx][si] ; cl == c := buf[i]

while:
    cmp cl, 0        ; c != 0?
    je end_while     ; c == 0

    cmp cl, 'A'
    je vocal
    cmp cl, 'E'
    je vocal
    cmp cl, 'I'
    je vocal
    cmp cl, 'O'
    je vocal
    cmp cl, 'U'
    jne no_vocal

vocal:
    inc ax            ; vocales++

no_vocal:
    inc si            ; i++
    mov cl, [bx][si] ; c := buf[i]
    jmp while

end_while:
    mov cx, 100

```

```

        mul cx                ; dx:ax := vocales * 100
        div si                ; ax := vocales * 100 / i

        pop si dx cx bx
        pop bp
        ret

_Porcentaje_Vocales ENDP

```

P87. La siguiente función en lenguaje ensamblador de 80x86, que implementa la función Fibonacci de C reproducida a continuación, tiene varios errores. Proponer una versión correcta de la misma función haciendo el **menor número de cambios**. Sólo se deben reescribir las líneas erróneas. Se supone que el programa de C está compilado en **modelo pequeño** (*small*).

```

unsigned int Fibonacci( unsigned int i )
{
    if (i <= 1) return i;
    return Fibonacci(i - 1) + Fibonacci(i - 2);
}

```

```

_Fibonacci PROC NEAR
    push bp
    mov bp, sp
    push bx

    mov ax, [bp+4]
    cmp ax, 1
    jle final

    inc ax
    push ax
    call Fibonacci
    mov ax, bx

    dec WORD PTR [bp-4]

    call Fibonacci
    add sp, 4
    add ax, bx

final: pop bp bx
    ret

_Fibonacci ENDP

```

```

_Fibonacci PROC NEAR
    push bp
    mov bp, sp
    push bx

    mov ax, [bp+4]
    cmp ax, 1
    jbe final

    dec ax
    push ax
    call _Fibonacci
    mov bx, ax

    dec WORD PTR [bp-4]

    call _Fibonacci
    add sp, 2
    add ax, bx

final: pop bx bp
    ret

_Fibonacci ENDP

```

P88. Escribir en ensamblador de 80x86 la función de C `Primes` que se reproduce en el siguiente recuadro, que calcula la factorización en números primos de un número. Se supone que el programa en C está compilado en **modelo compacto**. Se valorará la eficiencia del código.

```

_Primes PROC NEAR
    push bp
    mov bp, sp

```

```

push ax bx cx dx si di es

mov cx, [bp+4] ; cx == n
les bx, [bp+6] ; es:bx == t

mov si, 2      ; si == i
mov di, 0      ; di == count

while: cmp si, cx ; i <= n?
      ja final   ; i > n

mov dx, 0
mov ax, cx      ; dx:ax == n
div si          ; ax := n/i dx := n%i

cmp dx, 0       ; n%i == 0?
jne else        ; n%i != 0

mov es:[bx][di], si ; t[ count ] := i
add di, 2        ; count++

mov cx, ax      ; n := n/i
jmp while

else: inc si     ; i++
     jmp while

final: mov WORD PTR es:[bx][di], 0 ; t[ count ] := 0

pop es di si dx cx bx ax
pop bp

ret
_Primes ENDP

```

```

void Primes( unsigned int n, unsigned int *t )
{
    register unsigned int i, count;
    i = 2;
    count = 0;

    while ( i <= n )
    {
        if ((n % i) == 0)
        {
            t[ count ] = i;
            count++;
            n = n / i;
        }
        else i++;
    }
    t[ count ] = 0;
}

```

P89. Escribir en ensamblador de 80x86 la función recursiva de C alreves que se reproduce en el siguiente recuadro, que calcula el número capicúa de un entero dado con el número de dígitos indicado (1 para 1 dígito, 10 para 2 dígitos, 100 para 3 dígitos, etc.). Se supone que el programa en C está compilado en **modelo medio** (*medium*). Se valorará la eficiencia del código.

```

_alreves PROC FAR

push bp
mov bp, sp

mov ax, [bp+6] ; ax := n
cmp ax, 10
jb final      ; n < 10

push bx cx dx si di

mov cx, [bp+8] ; cx := digitos

```

```

unsigned int alreves( unsigned int n,
                     unsigned int digitos )
{
    register unsigned int resto, cociente;

    if ( n < 10 ) return n;
    else
    {
        resto = n % 10;
        cociente = n / 10;
        return resto * digitos +
            alreves( cociente, digitos/10 );
    }
}

```

```

mov dx, 0
mov bx, 10
div bx      ; ax := n / 10  dx := n % 10 (== resto)

mov si, ax  ; si == cociente

mov ax, cx  ; ax := digitos
mul dx      ; ax := resto * digitos
mov di, ax  ; di := resto * digitos

mov dx, 0
mov ax, cx  ; ax := digitos
div bx      ; ax := digitos/10

push ax     ; Apila digitos/10
push si     ; Apila cociente
call _alreves ; Retorna en ax
add sp, 4   ; Equilibra pila

add ax, di  ; ax := alreves() + resto * digitos

pop di si dx cx bx

final:
pop bp
ret

_alreves ENDP

```

P90. Escribir en ensamblador de 80x86 la función de C `insertNode` que se reproduce en el siguiente recuadro, que inserta un byte sin signo en un minHeap. Se supone que el programa en C está compilado en **modelo pequeño (small)**. Las variables locales han de estar almacenadas en registros. Se valorará la eficiencia del código.

```

_insertNode PROC NEAR
    push bp
    mov bp, sp
    push ax bx dx si di

    mov bx, [bp+4] ; bx == minheap
    mov si, [bp+6] ; si == size
    mov dl, [bp+8] ; dl == data

    mov di, [si]   ; di == i := *size
    inc WORD PTR [si] ; (*size)++

    mov si, di     ; si == di == i

while:
    cmp di, 0      ; i > 0?
    jbe endwhile  ; i <= 0

    shr si, 1      ; si := i/2

    mov al, [bx][si] ; al == minheap[i/2]

```

```

void insertNode( unsigned char *minheap,
                 unsigned int *size,
                 unsigned char data )
{
    register unsigned int i;

    i = *size;
    (*size)++;

    while( i > 0 && data < minheap[ i/2 ] )
    {
        minheap[ i ] = minheap[ i/2 ];
        i = i/2;
    }
    minheap[i] = data;
}

```

```

        cmp dl, al          ; data < minheap[i/2]?
        jae endwhile       ; data >= minheap[i/2]

        mov [bx][di], al    ; minheap[i] := minheap[i/2]
        mov di, si          ; i := i/2

        jmp while

endwhile:
        mov [bx][di], dl    ; minheap[i] := data

        pop di si dx bx ax
        pop bp
        ret

_inserNode ENDP

```

P91. Escribir en ensamblador de 80x86 la función de C `crc16` que se reproduce en el siguiente recuadro, que retorna el código CRC de 16 bits de una cadena de caracteres del tamaño en bytes indicado. Se debe incluir la definición en ensamblador de las tres constantes. Se supone que el programa en C está compilado en **modelo pequeño (small)**. Las variables locales han de almacenarse en registros. Se valorará la eficiencia del código.

```

WIDTH_ EQU 16
TOPBIT EQU 1 SHL (WIDTH_-1)
POLYNOMIAL EQU 8005h

```

```
_crc16 PROC NEAR
```

```

        push bp
        mov bp, sp

        push bx cx dx si di

        mov bx, [bp+4]      ; bx == message
        mov di, [bp+6]      ; di == nBytes

        mov ax, 0           ; ax == remainder := 0
        mov si, 0           ; si == byte := 0

for1:   cmp si, di           ; byte < nBytes?
        jae finfor1

        mov dh, 0
        mov dl, [bx][si]    ; dx := message[byte]
        mov cl, WIDTH_-8
        shl dx, cl          ; dx := message[byte] << (WIDTH-8)
        xor ax, dx          ; ax := message[byte] << (WIDTH-8) ^ remainder

```

```

#define WIDTH_ 16
#define TOPBIT (1 << (WIDTH_ - 1)) // << : SHIFT LEFT
#define POLYNOMIAL 0x8005

unsigned int crc16( unsigned char *message, int nBytes )
{
    register unsigned int remainder = 0;
    register unsigned int byte;
    register unsigned char bit;

    for (byte = 0; byte < nBytes; byte++)
    {
        // ^ : XOR
        remainder = (message[byte] << (WIDTH_-8)) ^ remainder;

        for (bit = 8; bit != 0; bit--)
        {
            if (remainder & TOPBIT)
                remainder = (remainder << 1) ^ POLYNOMIAL;
            else remainder = (remainder << 1);
        }
    }
    return (remainder);
}

```



```

        mov cl, 8           ; cl == bit := 8
for2:    cmp cl, 0           ; bit != 0?
        je finfor2         ; bit == 0

        test ax, TOPBIT    ; remainder & TOPBIT
        jz else1          ; remainder & TOPBIT == 0

        shl ax, 1          ; remainder := remainder << 1
        xor ax, POLYNOMIAL ; remainder := (remainder << 1) ^ POLYNOMIAL
        jmp endif1

else1:   shl ax, 1          ; remainder := remainder << 1
endif1:  dec cl            ; bit--
        jmp for2

finfor2: inc si             ; byte++
        jmp for1

finfor1: pop di si dx cx bx
        pop bp
        ret

_crc16 ENDP

```