FASM

Consola
Windows

Nombre: Vargas Cruz Jose Manuel **Carrera:** Ingeniería Informática

Introducción al ensamblador:

La mejor forma de desarrollar programas complejos es mediante el uso de los llamados "ensambladores"; estos son programas que traducen un texto (programa fuente) a un programa ejecutable.
De esta manera, se pueden aprovechar las características de los editores de texto ASCII. Se denomina así al formato más simple donde no se introducen códigos propios de los editores que mejoran el aspecto de los textos como negrillas, subrayados y otros.
"Lenguaje de máquina": se denomina conjunto de las instrucciones de un procesador.
El programa ensamblador acepta un lenguaje que incluye todo el lenguaje de máquina ("instrucciones") con ligeras y necesarias variaciones y añade una serie de "directivas",sentencias que tienen sentido en el ámbito del ensamblador y que pierden su identidad en el programa ejecutable.
Al lenguaje total, compuesto de instrucciones y directivas, se llaman "lenguaje ensamblador" ("assembly language") o simplemente "ensamblador" ("assembler").

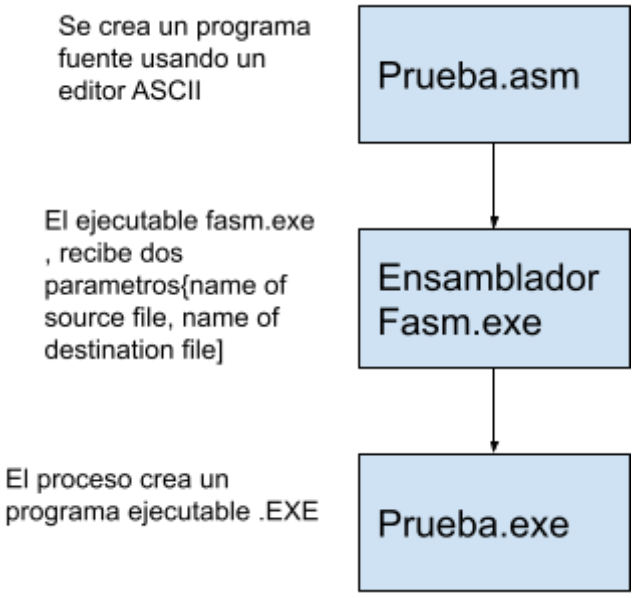
Usualmente el archivo que contiene un programa ensamblador tiene la extensión ASM; por ejemplo PRUEBA.ASM. Las líneas fuente que contiene deben tener el siguiente formato:
nombre acción expresión ;comentario
El "nombre" es una definición del usuario que crea un símbolo que puede ser utilizado luego en el programa, pueden representar código, datos y constantes; solo debe existir si la "acción"
lo requiere y bien puede no existir.
La "acción" es una directiva o una instrucción. Si no existe, la línea debe ser solo un comentario.
La "expresión" son el o los "parámetros" de la acción; por ejemplo operandos de instrucciones.
El "comentario" es opcional y debe ir precedido de punto y coma.
Ejemplos:
P0: MOV AX,BX

> Crea el nombre P0; seguido de : es una etiqueta representa código en el punto donde esta la instrucción MOV AX,BX. Antes o después de esta instrucción se puede bifurcar a ella como por ejemplo con JMP P0.

A DB 100

> Crea el nombre A; es una variable y representa un dato. Se reserva espacio de un byte con la directiva DB (Define Byte), se le da el valor inicial 100 (decimal) y el nombre A puede ser usado como operando memoria de instrucciones; por ejemplo se puede mover a AL con MOV AL,A o incrementar con INC A.

El proceso de creación de programas es el siguiente:



Tamaño de Operadores:

Operator	Bits	Bytes
byte	8	1
word	16	2
dword	32	4
fword	48	6
pword	48	6
qword	64	8
tbyte	80	10
tword	80	10
dqword	128	16
xword	128	16
qqword	256	32
yword	256	32

Registros:

Type	Bits								
General	8	al	cl	dl	bl	ah	ch	dh	bh
	16	ax	cx	dx	bx	sp	bp	si	di
	32	eax	ecx	edx	ebx	esi	ebp	esi	edi
Segment	16	es	cs	ss	ds	fs	gs		
Control	32	cr0		cr2	cr3	cr4			
Debug	32	dr0	dr1	dr2	dr3			dr6	dr7
FPU	80	st0	st1	st2	st3	st4	st5	st6	st7
MMX	64	mm0	mm1	mm2	mm3	mm4	mm5	mm6	mm7
SSE	128	xmm0	xmm1	xmm2	xmm3	xmm4	xmm5	xmm6	xmm7
AVX	256	ymm0	ymm1	ymm2	ymm3	ymm4	ymm5	ymm6	ymm7

Definiciones de datos:

Para definir datos o reservar un espacio para ellos, use una de las directivas enumeradas en la tabla:

Size (bytes)	Define data	Reserve data
1	db file	rb
2	dw du	rw
4	dd	rd
6	dp df	rp rf
8	dq	rq
10	dt	rt

La directiva de definición de datos debe ir seguida de una o más expresiones numéricas, separadas por comas. Estas expresiones definen los valores para las celdas de datos de tamaño según la directiva que se utilice. Por ejemplo, db 1,2,3 definirá los tres bytes de los valores 1, 2 y 3 respectivamente.

La directiva de reserva de datos debe ir seguida de una sola expresión numérica, y este valor define cuántas celdas del tamaño especificado deben reservarse. Todas las definiciones de directivas de datos también aceptan el valor ?, lo que significa que esta celda no debe ser inicializada a cualquier valor y el efecto es el mismo que al utilizar la reserva de datos directiva.

Constantes y etiquetas

En las expresiones numéricas también puede usar constantes o etiquetas en lugar de números.

Para definir la constante o etiqueta, debe usar las directivas específicas. Cada etiqueta puede ser definida sólo una vez y es accesible desde cualquier lugar de fuente (incluso antes de que fuera definido). La constante se puede redefinir muchas veces, pero en este caso sólo es accesible después de que se definió, y siempre es igual al valor de la última definición antes del lugar dónde se usa. Cuando una constante se define sólo una vez en la fuente, es accesible desde cualquier lugar.

La definición de constante consiste en el nombre de la constante seguida del carácter = y expresión numérica, que después del cálculo se convertirá en el valor de constante.

Este valor siempre se calcula en el momento en que se define la constante. Por ejemplo tu puedes definir la constante de recuento usando la directiva count = 17, y luego usarla en el instrucciones de ensamblaje, como mov cx, count {que se convertirá en mov cx, 17 durante el proceso de compilación.

Hay diferentes formas de definir etiquetas. Lo más simple es seguir el nombre de la etiqueta. por los dos puntos, esta directiva puede incluso ser seguida por la otra instrucción en el mismo línea. Define la etiqueta cuyo valor es igual al desplazamiento del punto donde está definida. Este método se usa generalmente para etiquetar los lugares en código. La otra forma es seguir el nombre de la etiqueta (sin dos puntos) por alguna directiva de datos. Define la etiqueta con valor igual al desplazamiento del comienzo de los datos definidos, y se recuerda como una etiqueta para los datos con tamaño de celda como se especifica para esa directiva de datos en la tabla.

La etiqueta se puede tratar como una constante de valor igual al offset del código o datos etiquetados.

Por ejemplo, cuando define datos usando la directiva etiquetada **char db 224**, para poner el desplazamiento de estos datos en el registro **bx**, debe usar la instrucción **mov bx,char** y poner el valor del byte direccionado por la etiqueta **char** al registro **dl**, debe usar **mov dl, [char]** (o mov dl, ptr char). Pero cuando intentas ensamblar **mov ax, [char],** causará un error, porque el fasm compara los tamaños de los operandos, que deberían ser iguales. Tú puedes forzar el ensamblaje de esa instrucción usando la anulación de tamaño: **mov ax, word [char]**, pero recuerde que esta instrucción leerá los dos bytes comenzando en la dirección **char**, mientras se definió como un byte.

Un formato para un programa .EXE con las llamadas directivas puede ser:

```
format PE CONSOLE 4.0
entry start

include 'win32a.inc'        ; Incluímos definiciones de estructuras y constantes

section '.text' code readable executable
  start:

    ;líneas de código

section '.data' data readable writeable
```

```
msg        DB "Hola mundo", 0


section '.idata' import data readable writeable

  library kernel32,'KERNEL32.DLL'
  include 'api\kernel32.inc'
```

Por ejemplo, un cálculo del factorial de 5 en un archivo llamado FACTOR.ASM puede ser:

```
INCLUDE 'win32ax.inc' ; _

.code
  start:
    xor ecx,ecx       ;es necesario limpiar los registros
    xor eax,eax
    mov cx,5
    mov ax,1
    jcxz fin
    ciclo: mul cx
    loop ciclo
    fin:
    mov [factorial],ax
    invoke  ExitProcess,0

.end start

SECTION '.DATA' DATA readable writeable
 factorial dw ?
```

Ejercicios resueltos:

1.- Promediar 5 bytes. Dejar el resultado en AL

```
INCLUDE 'win32ax.inc'

.code
  start:
    xor ecx,ecx
    MOV cl,5 ; CX es el contador
    mov esi, a ; SI apuntara al área a con los bytes a promediar
    MOV al,0 ; El resultado quedara en AL
    P0: add ax,[esi]
    inc esi ; Avance al siguiente byte
    loop P0
    mov ah,0 ; Debe ponerse en 0 porque la división
             ; considera AH como parte alta del dividendo
    mov dl,5 ; Se debe usar un registro por que la división
    div dl; no acepta un operando inmediato.

     mov byte[num],al       ;El resultado esta en AL
                            ; se usa byte para la conversion
    invoke  ExitProcess,0
.end start

SECTION '.DATA' DATA readable writeable
  num dw ?
  a DB 3,7,9,1Ah,0BCh
```

2.- Promediar 5 words. Dejar el resultado en BX

```
INCLUDE 'win32ax.inc'

.code
  start:
  xor ecx,ecx
   MOV CX,5
   MOV esi,a
   MOV AX,0
   P0:
        ADD AX,[esi]
        INC SI ; Como son words, hay que incrementar dos veces SI
        INC SI
   LOOP P0
   MOV DX,0 ; Es necesario poner DX en 0 porque la división
   MOV CX,5 ; lo considera parte alta del dividendo.
   DIV CX
   MOV BX,AX

    invoke  ExitProcess,0
.end start

SECTION '.DATA' DATA readable writeable

  a DW 322,721,911,13Ah,10BCh
```

3.- En X se encuentra una lista de words. No se conoce la cantidad exacta de words pero se sabe que el ultima es anterior a la marca FFFF. Promediarlas y dejar el resultado en DX.

```
INCLUDE 'win32ax.inc'

.code
  start:
        MOV esi, X
        xor eax,eax  ; es mas eficiente limpiar con XOR
        xor ecx,ecx
        mov bx,0FFFFh
        P0: CMP [esi],bx
        JZ FIN
        ADD AX,[esi]
        INC CX
        ADD SI,2 ; Otra forma de incrementar SI en 2.
        JMP P0
        FIN: xor edx,edx
        JCXZ NO_HAY
        DIV CX
        NO_HAY:

    invoke  ExitProcess,0
.end start

SECTION '.DATA' DATA readable writeable

  X DW 12,111,223,223,11H,123,22,0FFFFH
```

Instrucciones lógicas

'not' invierte los bits en el operando especificado para formar un complemento a uno del operando.

No tiene ningún efecto sobre las banderas. Las reglas para el operando son las mismas que para la instrucción inc.

Las instrucciones **and**, **or** y **xor** realizan las operaciones lógicas estándar. Ellos actualizan las banderas SF, ZF y PF. Las reglas para los operandos son las mismas que para la instrucción **add**.

Las instrucciones bt, bts, btr y btc operan en un solo bit que puede estar en la memoria o en un registro general.

La ubicación del bit se especifica como un desplazamiento del orden inferior final del operando. El valor del desplazamiento se toma del segundo operando, puede ser un byte inmediato o un registro general. Estas instrucciones primero asignan el valor del bit seleccionado a CF. La instrucción **bt** no hace nada más, **bts** establece el seleccionado bit a 1, **btr** restablece el bit seleccionado a 0, **btc** cambia el bit a su complemento. los El primer operando puede ser word o double word.

```
bt  ax,15        ; test bit in register
bts word [bx],15 ; test and set bit in memory
btr ax,cx        ; test and reset bit in register
btc word [bx],cx ; test and complement bit in memory
```

Las instrucciones **bsf** y **bsr** escanean una palabra o palabra doble para el primer bit establecido y almacenan el índice de este bit en el operando de destino, que debe ser un registro general. El bit cadena siendo escaneado está especificada por el operando de origen, puede ser un registro general o memoria. La bandera ZF se establece si toda la cadena es cero (no se encuentran bits establecidos); de lo contrario está despejado. Si no se encuentra ningún bit establecido, el valor del registro de destino no está definido.
bsf de orden inferior a orden superior (comenzando desde el índice de bits cero). **bsr** escanea desde lo alto orden a orden inferior (a partir del índice de bits 15 de una word o el índice 31 de una double word)

```
bsf ax,bx        ; scan register forward
bsr ax,[si]      ; scan memory reverse
```

shl desplaza el operando de destino a la izquierda el número de bits especificado en el segundo operando. El operando de destino puede ser un registro general de byte, word o doble word o memoria. El segundo operando puede ser un valor inmediato o el registro cl. Los procesadores desplazan ceros desde el lado derecho (orden inferior) del operando a medida que los bits salen de el lado izquierdo. El último bit que salió se almacena en CF. **sal** es sinónimo de **shl.**

```
shl al,1         ; shift register left by one bit
shl byte [bx],1  ; shift memory left by one bit
shl ax,cl        ; shift register left by count from cl
shl word [bx],cl ; shift memory left by count from cl
```

shr y **sar** desplazan el operando de destino a la derecha el número de bits especificado en el segundo operando. Las reglas para los operandos son las mismas que para la instrucción **shl**. **Shr** desplaza ceros desde el lado izquierdo del operando a medida que los bits salen del lado derecho. El último bit que salió se almacena en **CF**. **sar** conserva el signo del operando cambiando a ceros en el lado izquierdo si el valor es positivo o cambiando a unos si el valor es negativo. **shld** desplaza los bits del operando de destino a la izquierda el número de bits especificado en el tercer operando, mientras se desplazan los bits de orden superior del operando de origen al operando de destino a la derecha. El operando fuente permanece sin modificar. El destino El operando puede ser un registro general o memoria de word o double word, el operando fuente debe ser un registro general, el tercer operando puede ser un valor inmediato o el registro cl.

```
shld ax,bx,1     ; shift register left by one bit
shld [di],bx,1   ; shift memory left by one bit
shld ax,bx,cl    ; shift register left by count from cl
shld [di],bx,cl  ; shift memory left by count from cl
```

hrd desplaza bits del operando de destino hacia la derecha, mientras desplaza bits de orden inferior desde el operando de origen al operando de destino a la izquierda. El operando fuente permanece sin modificar. Las reglas para los operandos son las mismas que para la instrucción **shld**. **rol** y **rcl** rotan el operando de destino de byte, word o double word que deja el número de bits especificados en el segundo operando. Para cada rotación

especificada, el bit de orden alto que sale de la izquierda del operando vuelve a la derecha para convertirse en el nuevo poco de orden bajo. Además, **rcl** pone en CF cada bit de orden superior que sale por la izquierda lado del operando antes de que vuelva al operando como el bit de orden inferior en el siguiente ciclo de rotación. Las reglas para los operandos son las mismas que para la instrucción shl. ror y rcr rotan el operando de destino de byte, palabra o palabra doble a la derecha del número de bits especificados en el segundo operando. Para cada rotación especificada, el bit de orden bajo que sale por la derecha del operando regresa a la izquierda para convertirse en el nuevo bit de orden superior. rcr adicionalmente pone en CF cada bit de orden bajo que sale por la derecha lado del operando antes de que vuelva al operando como el bit de orden superior en el siguiente ciclo de rotación. Las reglas para los operandos son las mismas que para la instrucción shl. prueba realiza la misma acción que la instrucción y, pero no altera la operando de destino, solo actualiza las banderas. Las reglas para los operandos son las mismas que para la instrucción **and**.

Funciones para Consola:

- **GetStdHandle** devuelve un identificador para la entrada estándar, la salida estándar o el dispositivo de error estándar.
  HANDLE GetStdHandle(

    DWORD nStdHandle // input, output, or error device
  );

  Parametros

  nStdHandle:

  Especifica el dispositivo para el que devolver el identificador. Este parámetro puede tener uno de los siguientes valores:

Value	Meaning
STD_INPUT_HANDLE	Standard input handle
STD_OUTPUT_HANDLE	Standard output handle
STD_ERROR_HANDLE	Standard error handle

- **WriteFile** escribe datos en un archivo y está diseñado para operaciones sincrónicas y asincrónicas. La función comienza a escribir datos en el archivo en la posición indicada por el puntero del archivo. Una vez completada la operación de escritura, el puntero del archivo se ajusta por el número de bytes realmente escritos, excepto cuando el archivo se abre con FILE_FLAG_OVERLAPPED. Si el identificador de archivo se creó para entrada y salida (E / S) superpuestas, la aplicación debe ajustar la posición del puntero del archivo una vez finalizada la operación de escritura.
  BOOL WriteFile(

    HANDLE hFile, // handle to file to write to
    LPCVOID lpBuffer, // pointer to data to write to file
    DWORD nNumberOfBytesToWrite, // number of bytes to write

LPDWORD lpNumberOfBytesWritten, // pointer to number of bytes written
LPOVERLAPPED lpOverlapped // pointer to structure needed for overlapped I/O
);

- **ReadFile** lee datos de un archivo, comenzando en la posición indicada por el puntero del archivo. Una vez completada la operación de lectura, el puntero de archivo se ajusta por el número de bytes realmente leídos, a menos que el identificador de archivo se cree con el atributo superpuesto. Si el identificador de archivo se crea para entrada y salida (E / S) superpuestas, la aplicación debe ajustar la posición del puntero del archivo después de la operación de lectura.

BOOL ReadFile(

HANDLE hFile, // handle of file to read
LPVOID lpBuffer, // address of buffer that receives data
DWORD nNumberOfBytesToRead, // number of bytes to read
LPDWORD lpNumberOfBytesRead, // address of number of bytes read
LPOVERLAPPED lpOverlapped // address of structure for data
);

Ejercicios resueltos

1.- Leer una cadena de longitud fija

```
format PE CONSOLE 4.0
entry start

include 'win32a.inc'

section '.text' code readable executable
  start:
    invoke     GetStdHandle, STD_INPUT_HANDLE
    invoke     ReadFile, eax,  msg, msg.len, hNum,0  ;leemos desde consola
    xor        eax, eax

    invoke    ExitProcess,0

section '.data' data readable writeable
    hNum        RQ 1
    msg         DB ?
    msg.len     = 10 ;Cadena de Longitud Fija

section '.idata' import data readable writeable

  library kernel32,'KERNEL32.DLL'
  include 'api\kernel32.inc'
```

2.- Desplegar una cadena Asciiz

```
format PE CONSOLE 4.0
entry start

include 'win32a.inc'

section '.text' code readable executable
  start:
    invoke     GetStdHandle, STD_OUTPUT_HANDLE
    invoke     WriteFile, eax,  msg, msg.len, hNum,0  ;leemos desde consola
    xor        eax, eax

    invoke    ExitProcess,0

section '.data' data readable writeable
    hNum        RQ 1
    msg         DB "roberto va a casa",0 ;Cadena ASCIIZ termina en cero
    msg.len     = $-msg

section '.idata' import data readable writeable

  library kernel32,'KERNEL32.DLL'
  include 'api\kernel32.inc'
```

Ejercicios Usando FASM (Consola)

1.- Leer cadenas estructuradas y leer cadenas de longitud fija.

```
format PE CONSOLE 4.0
entry start

include 'win32a.inc'

section '.text' code readable executable
  start:
    invoke     GetStdHandle, STD_INPUT_HANDLE
    invoke     ReadFile, eax,  msg, msg.len, hNum,0  ;leemos desde consola
    xor        eax, eax

    invoke    ExitProcess,0

section '.data' data readable writeable
    hNum        RQ 1
    msg         DB ?
    msg.len     = 10 ;Cadena de Longitud Fija

section '.idata' import data readable writeable

  library kernel32,'KERNEL32.DLL'
  include 'api\kernel32.inc'
```

2.- Concatenar cadenas

a) longitud fija con longitud fija

```
format PE CONSOLE 4.0
entry start

include 'win32a.inc'

section '.text' code readable executable
  start:

    mov edi,cad1
    mov esi,cad3
    xor ecx,ecx
    mov cl,cad1.len
    stdcall concatenar
    mov edi,cad2
    xor ecx,ecx
    mov cl, cad2.len
    stdcall concatenar

    invoke     GetStdHandle, STD_OUTPUT_HANDLE    ;muestra en consola
    invoke WriteFile, eax,cad3,cad3.len,hNum,0
    xor        eax, eax

    invoke    ExitProcess,0

proc concatenar         ;concatena edi con esi

     xor bx,bx

     ciclo:
         mov bx,[edi]
         mov [esi],bx
         inc edi
         inc esi
     loop ciclo

ret
endp
section '.data' data readable writeable
    hNum        RQ 1
    cad1         DB "murcielago"
    cad1.len     = $-cad1 ;Cadena de Longitud Fija
    cad2         DB "bonito"
    cad2.len     =  $-cad2;Cadena de Longitud Fija
    cad3         DB ?
    cad3.len = cad1.len+cad2.len


section '.idata' import data readable writeable

  library kernel32,'KERNEL32.DLL'
  include 'api\kernel32.inc'
```

b) longitud fija con longitud fija con supresión de espacios en blanco

```
format PE CONSOLE 4.0
entry start

include 'win32a.inc'

section '.text' code readable executable
  start:
    xor edx,edx     ;contamos los caract. que se agregan (no espacios)
    mov edi,cad1
    mov esi,cad3
    xor ecx,ecx
    mov cl,cad1.len
    stdcall concatenar
    mov edi,cad2
    xor ecx,ecx
    mov cl, cad2.len
    stdcall concatenar

    invoke     GetStdHandle, STD_OUTPUT_HANDLE    ;muestra en consola
    invoke WriteFile, eax,cad3,edx,hNum,0
    xor        eax, eax

    invoke    ExitProcess,0

proc concatenar         ;concatena edi con esi

     xor ebx,ebx


     ciclo:

         cmp byte[edi],20h   ;comparamos si es espacio
         je saltar
         mov bx,[edi]
         mov [esi],bx
         inc esi
         inc edx
        saltar:
         inc edi
     loop ciclo

ret
endp
section '.data' data readable writeable
    hNum        RQ 1
    cad1         DB "mur c i el ago neg ro"
    cad1.len     = $-cad1 ;Cadena de Longitud Fija
    cad2         DB "b o n i t o"
    cad2.len     =  $-cad2;Cadena de Longitud Fija
    cad3         DB ?



section '.idata' import data readable writeable

  library kernel32,'KERNEL32.DLL'
  include 'api\kernel32.inc'
```

c) estructurada con ASCIIZ

```
format PE CONSOLE 4.0
entry start

include 'win32a.inc'

section '.text' code readable executable
  start:

    mov edi,cadl
    mov esi,cad3
    xor ecx,ecx
    inc edi
    mov cl,byte [edi] ; 2do caract. contiene longitud de la cadena
    push cx
    inc edi
    stdcall concatenar
    mov edi,cad2
    xor ecx,ecx
    xor ebx,ebx
    xor edx,edx
    ciclo2:
        cmp byte[edi],0
        je salir
         mov bx,[edi]
         mov [esi],bx
         inc edi
         inc esi
         inc dl
    jmp ciclo2

    salir:
        pop cx
        add dl,cl  ;longitud cadena3= 2da pos de estructurada+log asciiz

    invoke     GetStdHandle, STD_OUTPUT_HANDLE    ;muestra en consola
    invoke WriteFile, eax,cad3,edx,hNum,0
    xor        eax, eax
```

```
    invoke    ExitProcess,0

proc concatenar         ;concatena edi con esi

     xor bx,bx

     ciclo:
         mov bx,[edi]
         mov [esi],bx
         inc edi
         inc esi
     loop ciclo

ret
endp
section '.data' data readable writeable
    hNum        RQ 1
    cad1         DB 20,10,"murcielago"; Cadena estructurada
    cad2         DB "bonito",0 ;Cadena ASCIIZ

    cad3         DB ?


section '.idata' import data readable writeable

  library kernel32,'KERNEL32.DLL'
  include 'api\kernel32.inc'
```

d) estructurada con estructurada

```
format PE CONSOLE 4.0
entry start

include 'win32a.inc'

section '.text' code readable executable
  start:

    mov edi,cad1
    mov esi,cad3
    xor ecx,ecx
    inc edi
    mov cl,byte [edi]
    push cx
    inc edi
    stdcall concatenar
    mov edi,cad2
    inc edi
    xor ecx,ecx
    mov cl, byte [edi]
    push cx
    inc edi
    stdcall concatenar
    pop cx
    pop dx
    xor ebx,ebx
    add dx,cx
    mov bx,dx        ;sumamos las longitudes de las cad estructuradas
    invoke     GetStdHandle, STD_OUTPUT_HANDLE    ;muestra en consola
    invoke WriteFile, eax,cad3,ebx,hNum,0
    xor        eax, eax

    invoke    ExitProcess,0

proc concatenar         ;concatena edi con esi

     xor bx,bx
```

```
    ciclo:
        mov bx,[edi]
        mov [esi],bx
        inc edi
        inc esi
    loop ciclo

ret
endp
section '.data' data readable writeable
    hNum        RQ 1
    cad1         DB 20,10,"murcielago"

    cad2         DB 10,6,"bonito"
    cad3         DB ?



section '.idata' import data readable writeable

  library kernel32,'KERNEL32.DLL'
  include 'api\kernel32.inc'
```

3.- Obtener sub-cadenas; parte izquierda, derecha y central de:
a) cadena estructurada

```
format PE CONSOLE 4.0
entry start

include 'win32a.inc'

section '.text' code readable executable
  start:
    mov edi,cad
   inc di
    xor ecx,ecx
    mov cl,byte[edi]
   inc edi

    invoke     GetStdHandle, STD_OUTPUT_HANDLE
    invoke WriteFile, eax,edi,ecx,hNum,0
    xor        eax, eax

        mov esi,cad
        inc esi

        mov al,byte[esi]
        mov bl,3
        div bl
        xor ecx,ecx
        mov cl,al

        mov esi,cad
        inc esi
        inc esi

        xor dx,dx
        xor ecx,ecx
        mov cl,al
        mov edi,izq

       cicloiz:
        mov dx,[esi]
        mov [edi],dx
        add esi,1
```

```
     add edi,1
    loop cicloiz

    xor ebx,ebx
    mov bl,al
    mov bh,ah
    add bl,bh

    xor ecx,ecx
    mov cl,bl

    mov edi,medi
    xor edx,edx
   ciclom:
         mov dl,[esi]
         mov [edi],dl
         inc esi
         inc edi
         xor edx,edx
   loop ciclom

    xor edx,edx
    xor ecx,ecx
    mov cl,al
    mov edi,der

    cicloder:
         mov dl,byte[esi]
         mov byte[edi],dl
         inc edi
         inc esi
     loop cicloder
mostrar:
invoke     GetStdHandle, STD_OUTPUT_HANDLE
invoke WriteFile, eax,pizq,pizq.len,hNum,0
xor        eax, eax
invoke     GetStdHandle, STD_OUTPUT_HANDLE
invoke WriteFile, eax,izq,1,hNum,0
xor        eax, eax
```

```
    invoke      GetStdHandle, STD_OUTPUT_HANDLE
    invoke WriteFile, eax,pmed,pmed.len,hNum,0
    xor         eax, eax
    invoke      GetStdHandle, STD_OUTPUT_HANDLE
    invoke WriteFile, eax,medi,2,hNum,0
    xor         eax, eax
    invoke      GetStdHandle, STD_OUTPUT_HANDLE
    invoke WriteFile, eax,pder,pder.len,hNum,0
    xor         eax, eax
    invoke      GetStdHandle, STD_OUTPUT_HANDLE
    invoke WriteFile, eax,der,1,hNum,0
    xor         eax, eax

   invoke    ExitProcess,0


section '.data' data readable writeable
    hNum          RQ 1
    cad           DB 11,4,"hola" ; Cadena estructurada
    pizq          DB 10,13,"Parte Izquierda: ",0
    pizq.len      =$-pizq
    pmed          DB 10,13,"Parte Media: ",0
    pmed.len      =$-pmed
    pder          DB 10,13,"Parte Izquierda: ",0
    pder.len      =$-pder

    izq          DB ?
    medi         DB ?
    der          DB ?



section '.idata' import data readable writeable

  library kernel32,'KERNEL32.DLL'
  include 'api\kernel32.inc'
```

b) cadena ASCIIZ

```
format PE CONSOLE 4.0
entry start

include 'win32a.inc'

section '.text' code readable executable
  start:


    invoke     GetStdHandle, STD_OUTPUT_HANDLE
    invoke WriteFile, eax,cad,cad.len,hNum,0
    xor        eax, eax

        mov esi,cad

        mov al,cad.len
        mov bl,3
        div bl
        xor ecx,ecx
        mov cl,al
        mov esi,cad

        xor dx,dx
        xor ecx,ecx
        mov cl,al
        mov edi,izq

       cicloiz:
        mov dx,[esi]
        mov [edi],dx
        inc esi
        inc edi
       loop cicloiz

       xor ebx,ebx
       mov bl,al
       mov bh,ah
       add bl,bh
```

```
    xor ecx,ecx
    mov cl,bl

    mov edi,medi
    xor edx,edx
  ciclom:
        mov dl,[esi]
        mov [edi],dl
        inc esi
        inc edi
        xor edx,edx
  loop ciclom

    xor edx,edx
    xor ecx,ecx
    mov cl,al
    mov edi,der

    cicloder:
        mov dl,byte[esi]
        mov byte[edi],dl
        inc edi
        inc esi
     loop cicloder
mostrar:
invoke     GetStdHandle, STD_OUTPUT_HANDLE
invoke WriteFile, eax,pizq,pizq.len,hNum,0
xor        eax, eax
invoke     GetStdHandle, STD_OUTPUT_HANDLE
invoke WriteFile, eax,izq,1,hNum,0
xor        eax, eax
invoke     GetStdHandle, STD_OUTPUT_HANDLE
invoke WriteFile, eax,pmed,pmed.len,hNum,0
xor        eax, eax
invoke     GetStdHandle, STD_OUTPUT_HANDLE
invoke WriteFile, eax,medi,2,hNum,0
xor        eax, eax
```

```
    invoke      GetStdHandle, STD_OUTPUT_HANDLE
    invoke WriteFile, eax,medi,2,hNum,0
    xor         eax, eax
    invoke      GetStdHandle, STD_OUTPUT_HANDLE
    invoke WriteFile, eax,pder,pder.len,hNum,0
    xor         eax, eax
    invoke      GetStdHandle, STD_OUTPUT_HANDLE
    invoke WriteFile, eax,der,1,hNum,0
    xor         eax, eax

   invoke    ExitProcess,0


section '.data' data readable writeable
    hNum          RQ 1
    cad           DB "hola",0
    cad.len       =$-cad     ; Cadena estructurada
    pizq          DB 10,13,"Parte Izquierda: ",0
    pizq.len      =$-pizq
    pmed          DB 10,13,"Parte Media: ",0
    pmed.len      =$-pmed
    pder          DB 10,13,"Parte Izquierda: ",0
    pder.len      =$-pder

    izq          DB ?
    medi         DB ?
    der          DB ?



section '.idata' import data readable writeable

  library kernel32,'KERNEL32.DLL'
  include 'api\kernel32.inc'
```

4.- Asignar cadenas:

a) estructurada a estructurada

```
format PE CONSOLE 4.0
entry start

include 'win32a.inc'

section '.text' code readable executable
  start:

    mov esi,cad2
    inc esi
    xor ecx,ecx
    mov cl,byte[esi]
    push cx
    inc esi
    ciclo1:
    inc esi      ;recorrer al final
    loop ciclo1

    mov edi,cad1
    inc edi
    xor ecx,ecx
    mov cl,byte[edi]
    xor ebx,ebx
    inc edi
    ciclo2:
        mov bx,[edi]
        mov [esi],bx
        inc edi
        inc esi
    loop ciclo2
    mov edi,cad1
    mov esi,cad2
    inc edi
    inc esi
    xor ebx,ebx
    mov bl,byte[edi]
    add bl,byte[esi]    ;nueva longitud
```

```
    mov edi,cad2
    inc edi
    mov byte[edi],bl
    inc edi


     invoke     GetStdHandle, STD_OUTPUT_HANDLE    ;muestra en consola
    invoke WriteFile, eax,edi,ebx,hNum,0
    xor        eax, eax


    invoke    ExitProcess,0


section '.data' data readable writeable
    hNum        RQ 1
    cad1         DB 20,10,"murcielago" ;estructurada

    cad2         DB 10,6,"bonito"      ;estructurada



section '.idata' import data readable writeable

  library kernel32,'KERNEL32.DLL'
  include 'api\kernel32.inc'
```

b) longitud fija a estructurada

```
format PE CONSOLE 4.0
entry start

include 'win32a.inc'

section '.text' code readable executable
  start:

    mov esi,cad2
    inc esi
    xor ecx,ecx
    mov cl,byte[esi]
    push cx
    inc esi
    ciclo1:
    inc esi      ;recorrer al final
    loop ciclo1

    mov edi,cad1
    xor ecx,ecx
    mov cl,cad1.len
    xor ebx,ebx
    ciclo2:
        mov bx,[edi]
        mov [esi],bx
        inc edi
        inc esi
    loop ciclo2
    mov esi,cad2
    inc esi
    xor ebx,ebx
    mov bl,byte[esi]
    add bl,cad1.len    ;nueva longitud

    mov edi,cad2
    inc edi
    mov byte[edi],bl
    inc edi
```

```
     invoke      GetStdHandle, STD_OUTPUT_HANDLE    ;muestra en consola
    invoke WriteFile, eax,edi,ebx,hNum,0
    xor         eax, eax


    invoke    ExitProcess,0


section '.data' data readable writeable
    hNum        RQ 1
    cad1         DB "murcielago" ;longitud fija
    cad1.len     =$-cad1
    cad2         DB 10,6,"bonito"      ;estructurada



section '.idata' import data readable writeable

  library kernel32,'KERNEL32.DLL'
  include 'api\kernel32.inc'
```

c) estructurada a longitud fija

```
format PE CONSOLE 4.0
entry start

include 'win32a.inc'

section '.text' code readable executable
  start:

    mov esi,cad2

    xor ecx,ecx
    mov cl,cad2.len
    ciclo1:
    inc esi      ;recorrer al final
    loop ciclo1

    mov edi,cad1
    inc edi
    xor ecx,ecx
    mov cl,byte[edi]
    xor ebx,ebx
    inc edi
    ciclo2:
        mov bx,[edi]
        mov [esi],bx
        inc edi
        inc esi
    loop ciclo2

    mov edi,cad1
    inc edi
    xor ebx,ebx
    mov bl,byte[edi]
    add bl,cad2.len    ;nueva longitud

     invoke     GetStdHandle, STD_OUTPUT_HANDLE    ;muestra en consola
    invoke WriteFile, eax,cad2,ebx,hNum,0
    xor        eax, eax

    invoke    ExitProcess,0

section '.data' data readable writeable
    hNum        RQ 1
    cad1        DB 20,10,"murcielago" ;estructurada

    cad2        DB "bonito"      ;longitud fija
    cad2.len    =$-cad2



section '.idata' import data readable writeable

  library kernel32,'KERNEL32.DLL'
  include 'api\kernel32.inc'
```

d) ASCIIZ a estructurada

```
format PE CONSOLE 4.0
entry start

include 'win32a.inc'

section '.text' code readable executable
  start:

    mov esi,cad2
    inc esi
    xor ecx,ecx
    mov cl,byte[esi]
    push cx
    inc esi
    ciclo1:
    inc esi      ;recorrer al final
    loop ciclo1

    mov edi,cad1
    xor ecx,ecx
    mov cl,cad1.len
    xor ebx,ebx
    xor edx,edx
    ciclo2:
        cmp byte[edi],0
        je salir
        mov bx,[edi]
        mov [esi],bx
        inc edi
        inc esi
        inc dl
    jmp ciclo2
    salir:
    mov esi,cad2
    inc esi
    xor ebx,ebx
    mov bl,byte[esi]
    add bl,dl    ;nueva longitud
```

```
    mov edi,cad2
    inc edi
    mov byte[edi],bl
    inc edi

    invoke     GetStdHandle, STD_OUTPUT_HANDLE    ;muestra en consola
    invoke WriteFile, eax,edi,ebx,hNum,0
    xor        eax, eax

    invoke    ExitProcess,0

section '.data' data readable writeable
    hNum        RQ 1
    cad1         DB "murcielago",0 ;ASCIIz

    cad2         DB 10,6,"bonito"      ;estructurada

section '.idata' import data readable writeable

  library kernel32,'KERNEL32.DLL'
  include 'api\kernel32.inc'
```

e) ASCIIZ a longitud fija

```
format PE CONSOLE 4.0
entry start

include 'win32a.inc'

section '.text' code readable executable
  start:

    mov esi,cad2

    xor ecx,ecx
    mov cl,cad2.len
    ciclo1:
    inc esi      ;recorrer al final
    loop ciclo1

    mov edi,cad1
    xor ecx,ecx

    xor ebx,ebx
    xor edx,edx
    ciclo2:
        cmp byte[edi],0
        je salir
        mov bx,[edi]
        mov [esi],bx
        inc edi
        inc esi
        inc dl
    jmp ciclo2
   salir:
    xor ebx,ebx
    mov bl,cad2.len
    add bl,dl    ;nueva longitud

     invoke     GetStdHandle, STD_OUTPUT_HANDLE    ;muestra en consola
    invoke WriteFile, eax,cad2,ebx,hNum,0
    xor        eax, eax

    invoke    ExitProcess,0

section '.data' data readable writeable
    hNum        RQ 1
    cad1        DB "murcielago",0 ;ASCIIZ

    cad2        DB "bonito"      ;longitud fija
    cad2.len    =$-cad2



section '.idata' import data readable writeable

  library kernel32,'KERNEL32.DLL'
  include 'api\kernel32.inc'
```

f) estructurada a ASCIIZ

```
format PE CONSOLE 4.0
entry start

include 'win32a.inc'

section '.text' code readable executable
  start:

    mov esi,cad2

    xor ecx,ecx
    xor edx,edx
    ciclo1:
    cmp byte[esi],0
    je salir
    inc esi      ;recorrer al final
    inc dl
    jmp ciclo1
    salir:

    mov edi,cad1
    inc edi
    xor ecx,ecx
    mov cl,byte[edi]
    xor ebx,ebx
    inc edi
    ciclo2:
        mov bx,[edi]
        mov [esi],bx
        inc edi
        inc esi
    loop ciclo2

    mov edi,cad1
    inc edi
    xor ebx,ebx
    mov bl,byte[edi]
    add bl,dl    ;nueva longitud
```

```
     invoke      GetStdHandle, STD_OUTPUT_HANDLE     ;muestra en consola
    invoke WriteFile, eax,cad2,ebx,hNum,0
    xor         eax, eax

    invoke    ExitProcess,0

section '.data' data readable writeable
    hNum        RQ 1
    cad1        DB 20,10,"murcielago" ;estructurada

    cad2        DB "bonito",0       ;ASCIIZ



section '.idata' import data readable writeable

  library kernel32,'KERNEL32.DLL'
  include 'api\kernel32.inc'
```

Nota: archivos .ASM y .EXE adjuntados en un .ZIP

*De preferencia **compilar con Flat Assembler**, u similar y **ejecutar el .EXE desde consola** (CMD)