

Diseño Basado en Microprocesadores

Ejemplos de programación en ensamblador x86

Índice

1. Indicaciones	2
2. Programación de 32 bits	2
2.1. Buscar un carácter en una cadena	2
2.2. Convertir una cadena de caracteres hexadecimales en un valor de 32 bits . .	3
2.3. Comprobar si la representación binaria de un valor de 32 bits es capicúa . .	5
2.4. Buscar el mayor de los elementos contenidos en un array de valores de 32 bits con signo	6
2.5. Encontrar la longitud de una cadena ASCIIZ	7
2.6. Rellenar un bloque de memoria	8
2.7. Encontrar la aparición de una cadena de caracteres dentro de otra	9
3. Programación de la FPU	11
3.1. Conversión entre coordenadas cartesianas y esféricas	11
3.2. Raíces de un polinomio de segundo grado	15
4. Programación de 64 bits	18
4.1. Sumar char, short, int y long	18
4.2. Sumar ocho datos de tipo int	19
4.3. Añadir a una cadena con el DNI la letra del NIF	20
4.4. Comparar dos cadenas ASCIIZ sin diferenciar mayúsculas de minúsculas . .	22
4.5. Sumar filas y columnas de una matriz de datos de tipo int	23
5. Programación SSE con escalares	25
5.1. Calcular el área y el volumen de una esfera	25
5.2. Raíces de un polinomio de segundo grado	27
6. Programación SSE SIMD	29
6.1. Media y desviación típica	29
6.2. Multiplicar matriz 4×4 por vectores 4×1	32
6.3. Multiplicar dos matrices 4×4	34
6.4. Aumentar el brillo de una imagen de grises	37
6.5. Fundir dos imágenes RGB	39

1. Indicaciones

- Como ha sido la norma en la asignatura, los ejercicios están planteados como funciones en ensamblador para ser llamadas desde programas en lenguaje C bajo Linux.
- Entre los ejercicios incluidos hay algunos que ya han sido resueltos en clase. Las razones para repetirlos aquí es proporcionar comentarios adicionales. Puede haber diferencias entre las soluciones presentadas en clase y las recogidas aquí.
- Para ayudar a situar el tipo de programación en la que se encuadra un determinado ejercicio, los nombres de las funciones en ensamblador llevan uno de los siguientes prefijos:
 - **b32_**: la función usa sólo recursos de la programación en ensamblador de 32 bits y usa el convenio de llamada de los compiladores de 32 bits.
 - **b64_**: la función usa recursos de la programación en ensamblador de 64 bits y usa el convenio de llamada de los compiladores de 64 bits para Linux.
 - **fpu_**: la función usa instrucciones de la FPU. La FPU puede emplearse en funciones de 32 o 64 bits pero, como la tendencia en la programación de 64 bits es abandonar el uso de la FPU, en esta colección de ejercicios, los ejemplos que usan instrucciones de la FPU serán todos de 32 bits.
 - **sse_**: la función usa instrucciones SSE/SSE2/SSE3/SSSE3/SSE4. Estas instrucciones pueden usarse en funciones de 32 o 64 bits pero los ejemplos de esta colección usan estas instrucciones sólo en funciones de 64 bits.
- Si se detecta algún error o para cualquier comentario o sugerencia, por favor, dirigirse a victor.corbacho@uca.es.

2. Programación de 32 bits

2.1. Buscar un carácter en una cadena

Listado 2.1: b32_buscar_caracter_en_cadena.S

```
; Fichero: b32_buscar_caracter_en_cadena.S
;
; Escribe una función que permita encontrar la primera aparición de un carácter dado dentro de una
; cadena de caracteres. El prototipo de la función es:
;
; char* b32_buscar_caracter_en_cadena(const char* cadena, char caracter);
;
; donde
;
; cadena      es un puntero al comienzo de la cadena donde buscar. La cadena está terminada con un
;              byte a 0.
;
; caracter    es el código ASCII del carácter a buscar.
;
; La función retorna un puntero al carácter coincidente dentro de la cadena si éste se encuentra en
; la misma o un puntero nulo si no se encuentra o si el puntero "cadena" es nulo.

global b32_buscar_caracter_en_cadena

section .text
```

```

b32_buscar_caracter_en_cadena:
    push    ebp
    mov     ebp,esp

%define cadena    [ebp+8]
%define caracter  [ebp+12]

    xor     eax,eax        ; Si error, salir retornando NULL.

    mov     edx,cadena     ; EDX = puntero a la cadena.
    test    edx,edx
    jz      salida        ; Si puntero a cadena es NULL, salir retornando NULL.

    mov     cl,caracter    ; CL = carácter a buscar.

bucle:  mov     ch,[edx]    ; Leer un carácter de la cadena.
        inc     edx        ; Incrementar el puntero a la cadena para acceder al sig. carácter.
        cmp     ch,0       ; Si el carácter que se ha leído es el terminador 0...
        je      salida    ; salir retornando NULL => carácter no encontrado.
        cmp     ch,cl      ; Comparar el carácter leído con el buscado.
        jne     bucle      ; Si no son iguales, repetir.

        ; Si la instrucción JNE anterior no salta significa que se ha encontrado el carácter buscado.
        ; Debido al INC EDX dentro del bucle, EDX no sale apuntando a la posición dentro de la cadena
        ; donde se encontró el carácter sino a la posición siguiente. Por ello, EDX se decrementa en
        ; una unidad antes de cargarlo en EAX.

        dec     edx
        mov     eax,edx    ; EAX retorna un puntero a la posición donde se encontró el carácter.

salida: pop     ebp
        ret

```

2.2. Convertir una cadena de caracteres hexadecimales en un valor de 32 bits

Listado 2.2: b32_cadena_hex_a_32_bits.S

```

; Fichero: b32_cadena_hex_a_32_bits.S
;
; Escribe una función que permita obtener el valor representado en hexadecimal por una cadena de
; caracteres. El valor representado es de 32 bits, es decir, la cadena puede tener un máximo de ocho
; caracteres hexadecimales significativos. Los caracteres válidos son de '0' a '9', de 'A' a 'F' y de
; 'a' a 'f'. El prototipo de la función es:
;
; int b32_cadena_hex_a_32_bits(const char* cadena, unsigned int *resultado);
;
; donde
;
; cadena    es un puntero a la cadena a procesar. La cadena está terminada con un byte a 0.
;
; resultado es un puntero a la variable donde debe guardarse el resultado de la conversión.
;
; La función retorna 1 si los argumentos de entrada son válidos y 0 si alguno de ellos es un puntero
; nulo.
;
; La conversión finaliza si se encuentra el final de la cadena, si se encuentra un carácter no
; válido o si el valor excede de 32 bits. En todos estos casos, la función guarda en la posición
; apuntada por "resultado" el valor convertido hasta ese momento y retorna 1.

```

```

global b32_cadena_hex_a_32_bits

section .text

b32_cadena_hex_a_32_bits:

    push    ebp
    mov     ebp,esp

%define cadena    [ebp+8]
%define resultado [ebp+12]

    ; Salvar en la pila ESI y EDI ya que podrían estar usándose en el código de la función que
    ; nos ha llamado. EAX, ECX y EDX pueden usarse sin salvarlos en la pila porque el compilador
    ; cuenta con que puedan cambiar su valor.

    push    esi
    push    edi

    xor     eax,eax        ; Si los argumentos no son correctos, salir indicando error.

    mov     esi,cadena
    test    esi,esi
    jz      error          ; Si puntero a cadena es NULL, salir retornando 0 => error.

    mov     edi,resultado
    test    edi,esi
    jz      error          ; Si puntero a resultado es NULL, salir retornando 0 => error.

    xor     edx,edx        ; El resultado se irá construyendo en EDX.

    ; Bucle de tratamiento de cada carácter.

siguiente_digito:

    mov     al,[esi]        ; Leer el siguiente carácter en AL. Llamaremos C a este carácter.

    test    al,al
    jz      salida          ; Si es 0, fin de cadena => salir del bucle.

    ; Los caracteres válidos son: '0' a '9', 'A' a 'F' y 'a' a 'f'

    ; Primero, determinar si el carácter, C, está entre '0' y '9'.

    sub     al,'0'          ; AL = C - '0'
    jc      salida          ; Si C es < '0', CF = 1.
    cmp     al,9
    jbe     nuevo_nibble    ; Si C - '0' <= 9, C es el carácter de un número y AL ya tiene el
    ; valor correspondiente entre 0 y 9.

    ; Si C - '0' > 9, C quizás sea una letra mayúscula o minúscula.

    sub     al,'A'-'0'-10    ; AL = C - '0' - 'A' + '0' + 10 = C - 'A' + 10
    cmp     al,10
    jb      salida          ; Si C - 'A' + 10 < 10, C no es un carácter válido.
    cmp     al,15
    jbe     nuevo_nibble    ; Si C - 'A' + 10 <= 15, C es una letra mayúscula y AL ya tiene el
    ; valor correspondiente entre 10 y 15.

```

```

; Si C - 'A' + 10 > 15, C quizás sea una letra minúscula.

sub    al,'a'-'A'      ; AL = C - 'A' + 10 + 'a' - 'A' = C - 'a' + 10
cmp    al,10
jb     salida          ; Si C - 'a' + 10 < 10, C no es un carácter válido.
cmp    al,15
ja     salida          ; Si AL - 'a' + 10 <= 15, C es el carácter es una letra minúscula
                        ; y AL ya tiene el valor correspondiente entre 10 y 15.

nuevo_nibble:
shl    edx,4           ; Desplazar el resultado calculado hasta el momento 4 bits a la izquierda.
or     dl,al           ; Introducir el valor de AL en los cuatro bits bajos de EDX.

inc    esi             ; Incrementar el puntero a la cadena.

; Cuando los 4 bits más significativos de EDX tomen un valor distinto de 0000 ya no podremos
; procesar más caracteres de la cadena. Si seguimos, el valor excederá de 32 bits y el
; enunciado especifica que debemos terminar la conversión si esto ocurre.

test   edx,0xF0000000 ; Comprobar los 4 bits más significativos de EDX.
jz     siguiente_digito; Si son 0000, procesar otro dígito.

salida: mov    [edi],edx
        mov    eax,1      ; Salir indicando éxito.

error:  pop    edi
        pop    esi
        pop    ebp
        ret

```

2.3. Comprobar si la representación binaria de un valor de 32 bits es capicúa

Listado 2.3: b32_es_binario_capicua.S

```

; Fichero: b32_es_binario_capicua.S
;
; Escribe una función que retorne 1 si la representación en binario de un valor de tipo unsigned int
; que se recibe como argumento es capicúa y 0 en caso contrario. El prototipo de la función es
;
; int b32_es_binario_capicua(unsigned int n);

global    b32_es_binario_capicua

section   .text

b32_es_binario_capicua:
    push    ebp
    mov     ebp,esp

    mov     edx,[ebp+8]    ; EDX = n.
    xor     eax,eax        ; EAX = 0.
    mov     ecx,16         ; ECX se usará como contador de bucle.

bucle:  shr     edx,1        ; Desplazar EDX a la derecha. El LSB cae en el acarreo y por la
                        ; izquierda entra un bit a cero.
        rcl     eax,1        ; Rotar EAX hacia la izquierda introduciendo el acarreo por su derecha.
        loop   bucle        ; Repetir 16 veces.

    cmp     eax,edx        ; Comparar EAX con EDX. Si el valor de EDX antes de entrar en el bucle

```

```

mov     eax,1          ; era capicúa, ahora EAX = EDX.
cmovne  eax,ecx        ; Preparar un 1 en EAX => retornar "es capicua".
                        ; Si el CMP anterior detectó EDX != EAX, cambiar el valor de EAX por 0.
                        ; Se aprovecha que ECX sale a 0 del bucle. Puede comprobarse el estado
                        ; en el que CMP dejó los indicadores aún teniendo entre CMP y CMOVNE
                        ; la instrucción MOV EAX,1 porque MOV no afecta a los indicadores.

pop     ebp
ret

```

2.4. Buscar el mayor de los elementos contenidos en un array de valores de 32 bits con signo

Listado 2.4: b32_buscar_mayor.S

```

; Fichero: buscar_mayor.S
;
; Escribe una función que obtenga el mayor de los valores contenidos en un array de valores de
; tipo int. Recuerda que el tipo int es un valor de 32 bits con signo. El prototipo de la función es
;
; int b32_buscar_mayor(const int *ptr_datos, int num_datos, int* ptr_resultado);
;
; donde
;
; ptr_datos   es un puntero al primer elemento del array.
;
; num_datos   es el número de datos contenidos en el array.
;
; resultado   es un puntero a la variable donde debe almacenarse el mayor valor del array.
;
; La función retorna 1 si la búsqueda se realizó con éxito y 0 si alguno de los argumentos
; no es válido (alguno de los punteros es nulo o el número de datos indicado es <= 0).

global b32_buscar_mayor

section .text

b32_buscar_mayor:
    push    ebp
    mov     ebp,esp

#define datos      [ebp+8]
#define num_datos  [ebp+12]
#define resultado  [ebp+16]

    push    edi          ; Preservar EDI. El compilador espera que EDI no cambie.

    xor     eax,eax       ; Si error, retornar 0.

    mov     edx,datos     ; EDX = puntero a los datos.
    test    edx,edx
    jz      error         ; Si el puntero a los datos es nulo, salir indicando error.

    mov     ecx,num_datos ; ECX = número de datos.
    cmp     ecx,0
    jbe     error         ; Si número de datos es <= 0, salir indicando error.

    mov     edi,resultado ; EDI = puntero al resultado.
    test    edi,edi
    jz      error         ; Si el puntero al resultado es nulo, salir indicando error.

```

```

        mov     eax,[edx]      ; Leer el primer dato del array en EAX.
        dec     ecx           ; Decrementar el número de datos por procesar.
        jz      exito         ; Si hay un solo dato, éste es el mayor.

        ; EAX mantendrá el mayor dato encontrado hasta ahora.

bucle:  add     edx,4          ; Avanzar el puntero al siguiente dato.
        cmp     eax,[edx]     ; Comparar el mayor hasta ahora con el siguiente dato del array.
        cmovl   eax,[edx]     ; Si el mayor hasta ahora es menor que el dato en el array,
                               ; el mayor hasta ahora pasa a ser el dato en el array.
        loop    bucle         ; Repetir tantas veces como datos hay en el array.

exito:  mov     [edi],eax      ; Guardar el mayor en la posición apuntada por resultado.
        mov     eax,1         ; Salir indicando éxito.

error:  pop     edi
        pop     ebp
        ret

```

2.5. Encontrar la longitud de una cadena ASCII

Listado 2.5: b32_longitud_cadena.S

```

; Fichero: b32_longitud_cadena.S
;
; Escribe una función que retorne la longitud de una cadena de caracteres. El prototipo de la función
; es
;
; unsigned int b32_longitud_cadena(const char* cadena);
;
; donde
;
; cadena es un puntero al primero de los caracteres de la cadena cuya longitud se desea hallar.
; La cadena termina con un byte a 0.
;
; Si el puntero cadena es nulo la función retorna 0 como resultado.

global b32_longitud_cadena

section .text

b32_longitud_cadena:
    push     ebp
    mov     ebp,esp

    push     edi              ; Preservar el registro EDI ya que el compilador cuenta con que su valor
                               ; sea preservado por las funciones.

    xor     eax,eax           ; Poner EAX a 0.

    mov     edi,[ebp+8]       ; EDI = puntero a la cadena.
    test    edi,edi
    jz      salida           ; Si el puntero a la cadena es nulo salir indicando longitud 0.

    ; Calcular la longitud de la cadena.

    mov     ecx,0xFFFFFFFF    ; Cargar ECX con el valor de 32 bits máximo.
    cld                                           ; Poner a cero el indicador de dirección.
    repne   scasb              ; Buscar el carácter en AL (= 0 ahora) en el bloque apuntado por EDI.

```

```

; Como REPNE SCASB decreuenta ECX una vez más antes de terminar, ahora
;
; ECX = 0xFFFFFFFF - longitud - 1 => longitud = 0xFFFFFFFF - 1 - ECX = 0xFFFFFEE - ECX

mov    eax,0xFFFFFEE
sub    eax,ecx      ; EAX = 0xFFFFFEE - ECX = longitud.

salida: pop    edi
       pop    ebp
       ret

```

2.6. Rellenar un bloque de memoria

Listado 2.6: b32_rellenar_bloque.S

```

; Fichero: b32_rellenar_bloque.S
;
; Escribe una función que rellene todos los bytes de un bloque de memoria con un mismo valor.
; El prototipo de la función es
;
; int b32_rellenar_bloque(unsigned char byte_relleno,
;                          void* ptr_bloque,
;                          unsigned int longitud_bloque);
;
; donde
;
; byte_relleno es el valor con el que deben rellenarse todos los bytes del bloque.
;
; ptr_bloque es un puntero al primer byte del bloque a rellenar.
;
; longitud_bloque es el número de bytes que componen el bloque a rellenar.
;
; La función retorna 1 si la operación se realizó con éxito y 0 si ptr_bloque es nulo.

global b32_rellenar_bloque

section .text

b32_rellenar_bloque:

    push    ebp
    mov     ebp,esp

#define byte_relleno    [ebp+8]
#define ptr_bloque      [ebp+12]
#define longitud_bloque [ebp+16]

    push    edi      ; Preservar el valor de EDI en la pila ya que el compilador espera que
                     ; las funciones no alteren su valor.

    xor     eax,eax   ; Si error, retornar 0.

    mov     edi,ptr_bloque ; EDI = puntero al bloque a rellenar.
    test    edi,edi
    jz      salida    ; Si ptr_bloque es NULL, salir indicando error.

    cld                          ; Poner a 0 el indicador de dirección.

    mov     al,byte_relleno

```



```

; En lugar de rellenar todo el bloque byte a byte, primero el relleno se realiza una doble
; palabra (4 bytes) cada vez. Es de esperar que una escritura en memoria de una doble palabra
; se realice más rápidamente que cuatro escrituras de un sólo byte.
;
; Por tanto, primero se calcula cuántas dobles palabras completas hay en el bloque.

mov     ecx,longitud_bloque
shr     ecx,2           ; ECX = longitud/4.
jz      longitud_menor_4; Si el resultado es 0, la longitud es menor que 4 y el bloque no
                        ; contiene ni una sola doble palabra.

; Rellenar dobles palabras completas. Para ello, primero hay que construir una doble palabra
; con el byte de relleno repetido 4 veces. Llamando Xx al byte de relleno:

mov     ah,al           ; EAX = ???XxXx
mov     edx,eax         ; EDX = ???XxXx
shl     edx,16          ; EDX = XxXx0000
or      eax,edx         ; EAX = XxXxXxXx

rep     stosd           ; Rellenar de 4 en 4 bytes.

; Después de rellenar de 4 en 4 (o si no se rellenó ninguna doble palabra por ser la longitud
; menor de 4) pueden quedar 0, 1, 2 o 3 bytes por rellenar. Éstos se rellenan de 1 en 1.

longitud_menor_4:
mov     ecx,longitud_bloque
and     ecx,3           ; ECX = resto de dividir longitud entre 4.
jz      salida         ; Si el resultado es 0, no queda nada que rellenar.
rep     stosb           ; Rellenar de 1 en 1.

mov     eax,1           ; Salir indicando éxito.

salida: pop     edi
        pop     ebp
        ret

```

2.7. Encontrar la aparición de una cadena de caracteres dentro de otra

Listado 2.7: b32_buscar_cadena_en_cadena.S

```

; Fichero: b32_buscar_cadena_en_cadena.S
;
; Escribe una función que permita conocer si una cadena de caracteres contiene dentro de sí
; otra cadena de caracteres. El prototipo de la función es
;
; char* b32_buscar_cadena_en_cadena(const char* cadena_donde_buscar,
;                                   const char* cadena_a_encontrar);
; donde
;
; cadena_donde_buscar es un puntero al primero de los caracteres de la cadena dentro de la cual se
; desea encontrar la aparición de otra.
;
; cadena_a_encontrar es un puntero al primero de los caracteres de la cadena que se desea encontrar
; dentro de la otra.
;
; La función retorna un puntero al caracter de la cadena apuntada por cadena_donde_buscar a partir
; del cual se ha encontrado la primera aparición de la cadena apuntada por cadena_a_encontrar.
; Si la cadena apuntada por cadena_a_encontrar no se encuentra dentro de la apuntada por
; cadena_donde_buscar o si alguno de los punteros es nulo, la función retorna un puntero nulo.

```

```

global b32_buscar_cadena_en_cadena

section .text

b32_buscar_cadena_en_cadena:

    push    ebp
    mov     ebp,esp

%define cadena_donde_buscar [ebp+8]
%define cadena_a_encontrar [ebp+12]

    push    ebx                ; Preservar EBX, EDI y ESI en la pila ya que los vamos a necesitar pero
    push    edi                ; el compilador espera que las funciones no modifiquen sus valores.
    push    esi

    xor     eax,eax            ; Si error, retornar puntero nulo.

    mov     edi,cadena_donde_buscar
    test    edi,edi
    jz      salida            ; Si cadena_donde_buscar es nulo, retornar puntero nulo.

    mov     edi,cadena_a_encontrar
    test    edi,edi
    jz      salida            ; Si cadena_a_encontrar es nulo, retornar puntero nulo.

    cld                          ; Poner a 0 el flag de dirección.

    ; Calcular la longitud de la cadena a encontrar.

    mov     ecx,0xFFFFFFFF
    cld
    repne   scasb
    mov     edx,0xFFFFFFFF
    sub     edx,ecx            ; EDX = longitud de la cadena a encontrar.

    ; Si la cadena a encontrar está vacía => encontrada, ya que cualquier cadena contiene a la
    ; cadena vacía, incluso si es una cadena vacía. Se retornará un puntero al comienzo de la
    ; cadena apuntada por cadena_donde_buscar.

    cmovz   eax,cadena_donde_buscar
    jz      salida

    ; Calcular la longitud de la cadena donde buscar.

    mov     edi,cadena_donde_buscar
    mov     ecx,0xFFFFFFFF
    repne   scasb
    mov     ebx,0xFFFFFFFF
    sub     ebx,ecx            ; EBX = longitud de la cadena donde buscar.

    ; Si la cadena a encontrar no es la cadena vacía pero la cadena donde buscar sí lo es,
    ; entonces no se encontró. Se retorna EAX = 0.

    jz      salida

    ; Llegados a este punto, comenzamos la búsqueda sistemática de la cadena a encontrar dentro de
    ; la cadena donde buscar. La búsqueda se realiza dentro del siguiente bucle, en el que los
    ; registros EDX, EBX y EAX se usan para las siguientes funciones
    ;

```

```

; EDX: longitud de la cadena a encontrar.
;
; EAX: puntero a la posición actual dentro de la cadena donde buscar a partir de la cual se
; realiza la búsqueda. Al principio apunta al primer carácter de la cadena donde buscar.
; Si la búsqueda a partir de esta primera posición falla, EAX se incrementa para repetir
; la búsqueda a partir de la segunda posición y así sucesivamente hasta que la búsqueda
; tenga éxito o hasta que longitud restante de la cadena donde buscar sea menor que la
; longitud de la cadena a encontrar.
;
;
; EBX: longitud de lo que queda de cadena donde buscar. Antes de entrar en el bucle contiene
; la longitud total de la cadena donde buscar. Cada vez que la búsqueda a partir
; de la posición indicada por EAX falla, EBX se decrementa en una unidad.

mov     eax,cadena_donde_buscar

buscar:
; Antes de proceder a una búsqueda, se comprueba si la longitud restante de la cadena donde
; buscar es menor que la cadena a encontrar. Si es así, la cadena donde buscar no contiene a
; la cadena a encontrar.

cmp     ebx,edx
jae     seguir           ; Si EBX >= EDX, aún podemos intentar encontrar una dentro de otra.
xor     eax,eax          ; Si EBX < EDX, no hubo éxito en la búsqueda. Poner EAX a 0
jmp     short salida     ; y salir.

seguir: mov     ecx,edx      ; ECX = longitud de la cadena a encontrar.
mov     esi,eax            ; ESI apunta a la posición actual dentro de la cadena donde buscar.
mov     edi,cadena_a_encontrar
repe    cmpsb             ; Comparar ECX bytes a partir de [ESI] y [EDI] mientras sean iguales.
je      salida            ; Si la instrucción REPE CMPSB termina con ZF = 1 significa que
; ECX llegó a 0 y todos los caracteres fueron iguales => encontrada.
; Entonces se salta a salida con EAX apuntando a la posición dentro
; de la cadena donde buscar a partir de la cual se encontró la cadena
; a encontrar.

; Sin embargo, si REPE CMPSB terminó con ZF = 0 significa que hubo una discrepancia entre
; caracteres y que la cadena a encontrar no se encuentra a partir de la posición actual
; dentro de la cadena donde buscar. Entonces, avanzar a la siguiente posición dentro de la
; cadena donde buscar, decrementar la longitud que queda de la misma y volver a intentar.

inc     eax               ; Avanzar a la siguiente posición dentro de la cadena.
dec     ebx               ; Decrementar en 1 la longitud de la cadena en la que buscar.

jmp     short buscar

salida: pop     esi
pop     edi
pop     ebx
pop     ebp
ret

```

3. Programación de la FPU

3.1. Conversión entre coordenadas cartesianas y esféricas

- Conversión de coordenadas cartesianas a esféricas


```

;
;  sentido_conversion  indica si la conversión solicitada es de cartesianas a esféricas o viceversa.
;                      Si es 0 la conversión se realiza de cartesianas a esféricas.
;                      Si es distinto de 0 la conversión se realiza de esféricas a cartesianas.
;
;  in_1, in_2, in_3   son las coordenadas de partida, cartesianas si sentido_conversion == 0 o
;                      esféricas si sentido_conversion es != 0.
;
;
;  out_1, out_2, out_3 son punteros a las variables donde almacenar las coordenadas calculadas.

        global      fpu_conversion_cartesianas_esfericas

        section     .text

fpu_conversion_cartesianas_esfericas:

        push        ebp
        mov         ebp,esp

#define sentido      dword [ebp+8]
#define in_1         qword [ebp+12]
#define in_2         qword [ebp+20]
#define in_3         qword [ebp+28]
#define ptr_out_1    dword [ebp+36]
#define ptr_out_2    dword [ebp+40]
#define ptr_out_3    dword [ebp+44]
#define CARTESIANAS_A_ESFERICAS 0

        xor         eax,eax          ; Si error, salir con EAX = 0.

        mov         ecx,ptr_out_2
        test        ecx,ecx
        jz          error           ; Si out_2 es nulo, salir indicando error.

        mov         edx,ptr_out_3
        test        edx,edx
        jz          error           ; Si out_3 es nulo, salir indicando error.

        mov         eax,ptr_out_1
        test        eax,eax
        jz          error           ; Si out_1 es nulo, salir indicando error.

        cmp         sentido,CARTESIANAS_A_ESFERICAS
        jne         esfericas_a_cartesianas

; ----- Conversión de cartesianas a esféricas -----
;
;  r = sqrt(x^2 + y^2 + z^2)
;  theta = FPATAN(x, y)
;  phi = FPATAN(sqrt(x^2 + y^2), z)
;
;  La instrucción FPATAN de la FPU calcula el ángulo en radianes formado entre el semieje
;  positivo X y la línea trazada desde el origen hasta el punto (ST0, ST1) y deja el resultado
;  en ST1. Después retira el valor en la cima de la pila. Por tanto, al final, el resultado queda
;  en ST0. . El ángulo depende así de los signos iniciales de ST0 y ST1 y no del signo del
;  cociente ST1/ST0. El resultado tiene siempre el mismo signo que el valor inicial de ST0 y está
;  comprendido entre -pi y pi. La instrucción trata correctamente el caso ST0 = 0.
;
;  in_1 = x, in_2 = y, in_3 = z.
;  EAX apunta a r, ECX apunta a theta, EDX apunta a phi.

```

```

;
; Abreviamos: R = sqrt(x^2 + y^2), S = sqrt(x^2 + y^2 + z^2).
;
;
; ----- Imagen de la pila de la FPU -----
;
;          ST0          ST1          ST2          ST3          ST4          ST5
;
fld     in_2          ; y
fld     in_1          ; x          y
fld     st1           ; y          x          y
fmul    st0,st0       ; y^2       x          y
fld     st1           ; x          y^2       x          y
fmul    st0,st0       ; x^2       y^2       x          y
faddp   st1,st0       ; x^2+y^2   x          y
fld     in_3          ; z          x^2+y^2   x          y
fld     st1           ; x^2+y^2   z          x^2+y^2   x          y
fsqrt   ; R           z          x^2+y^2   x          y
fld     st1           ; z          R          z          x^2+y^2   x          y
fpatan  ; FPATAN(R,z) z          x^2+y^2   x          y
fstp    qword [edx]   ; z          x^2 + y^2   x          y
fmul    st0,st0       ; z^2       x^2 + y^2   x          y
faddp   st1,st0       ; x^2+y^2+z^2 x          y
fsqrt   ; S           x          y
fstp    qword [eax]   ; x          y
fpatan  ; FPATAN(y,x)
fstp    qword [ecx]

mov     eax,1
jmp     exito

```

esfericas_a_cartesianas:

```

; ----- Conversión de esféricas a cartesianas -----
;
; x = r*sin(theta)*cos(phi)
; y = r*sin(theta)*sin(phi)
; z = r*cos(theta)
;
; in_1 = r, in_2 = theta, in_3 = phi.
; EAX apunta a x, ECX apunta a y, EDX apunta a z.
;
; Abreviamos: ct = cos(theta), st = sin(theta), cp = cos(phi), sp = sin(phi)
;
;
; ----- Imagen de la pila de la FPU -----
;
;          ST0          ST1          ST2          ST3          ST4          ST5          ST6
;
fld     in_1          ; r
fld     in_2          ; theta    r
fsincos ; ct          st          r
fld     in_1          ; r          ct          st          r
fld     in_3          ; phi      r          ct          st          r
fsincos ; cp          sp          r          ct          st
fld     st2           ; r          cp          sp          r          ct          st          r
fmul    st0,st5       ; r*st      cp          sp          r          ct          st          r
fmulp   st1,st0       ; r*st*cp   sp          r          ct          st          r
fstp    qword [eax]   ; sp          r          ct          st          r
fmulp   st1,st0       ; r*sp      ct          st          r
fmul    st0,st2       ; r*st*sp   ct          st          r
fstp    qword [ecx]   ; ct          st          r
fmulp   st2,st0       ; st          r*ct

```

```

        fstp    st0        ; r*ct
        fstp    qword [edx]

exito:  mov     eax,1        ; Salir indicando éxito.
error:  pop     ebp
        ret

```

3.2. Raíces de un polinomio de segundo grado

Listado 3.2: fpu_raices_polinomio_segundo_grado.S

```

; Fichero: fpu_raices_polinomio_segundo_grado.S
;
; Escribe una función que obtenga las raíces de un polinomio de segundo grado  $ax^2 + bx + c$ 
; tanto si éstas son reales como si son complejas. El prototipo de la función es:
;
; int fpu_raices_polinomio_segundo_grado(double coef_a,
;                                     double coef_b,
;                                     double coef_c,
;                                     double *r1_real,
;                                     double *r1_imag,
;                                     double *r2_real,
;                                     double *r2_imag);
;
; donde
;
; coef_a es el coeficiente a del polinomio.
;
; coef_b es el coeficiente b del polinomio.
;
; coef_c es el coeficiente c del polinomio.
;
; r1_real es un puntero a la variable donde debe almacenarse la parte real de la primera raíz.
;
; r1_imag es un puntero a la variable donde debe almacenarse la parte imaginaria de la primera raíz.
; Si las raíces son reales, debe almacenarse 0.
;
; r2_real es un puntero a la variable donde debe almacenarse la parte real de la segunda raíz.
;
; r2_imag es un puntero a la variable donde debe almacenarse la parte imaginaria de la segunda raíz.
; Si las raíces son reales, debe almacenarse 0.
;
; La función retorna 1 si las raíces son reales, -1 si son complejas y 0 si alguno de los argumentos
; no es válido (a, b o c es NaN, a = b = 0 o alguno de los punteros es nulo).
;
; Casos posibles:
;
; Si a == 0 y b == 0: tenemos c = 0 y no calculamos ninguna raíz.
; Si a == 0 y b != 0: *r1_real = *r2_real = -c/b, *r1_imag = *r2_imag = 0.
; Si a != 0 aplicamos la fórmula general  $r_{1,2} = (-b \pm \sqrt{b^2 - 4ac}) / (2a)$  distinguiendo
; entonces los casos  $b^2 - 4ac < 0 \Rightarrow$  complejas y  $b^2 - 4ac \geq 0 \Rightarrow$  reales.

section     .text

global     fpu_raices_polinomio_segundo_grado

fpu_raices_polinomio_segundo_grado:
    push    ebp
    mov     ebp,esp

```

```

#define coef_a      qword [ebp+8]
#define coef_b      qword [ebp+16]
#define coef_c      qword [ebp+24]
#define r1_real     dword [ebp+32]
#define r1_imag     dword [ebp+36]
#define r2_real     dword [ebp+40]
#define r2_imag     dword [ebp+44]

    push    esi                ; Preservar en la pila ESI y EDI, ya que la función los usa pero
    push    edi                ; el compilador confía en que sus valores no cambien.

    xor     eax,eax            ; Si error, salir con EAX = 0.

    mov     ecx,r1_real        ; ECX = puntero a la parte real de la 1ª solución.
    test    ecx,ecx
    jz      salida            ; Si el puntero es nulo, salir retornando 0.

    mov     edx,r1_imag        ; EDX = puntero a la parte imaginaria de la 1ª solución.
    test    edx,edx
    jz      salida            ; Si el puntero es nulo, salir retornando 0.

    mov     esi,r2_real        ; ESI = puntero a la parte real de la 2ª solución.
    test    esi,esi
    jz      salida            ; Si el puntero es nulo, salir retornando 0.

    mov     edi,r2_imag        ; EDI = puntero a la parte imaginaria de la 2ª solución.
    test    edi,edi
    jz      salida            ; Si el puntero es nulo, salir retornando 0.

    ; Primero, comprobar si a es válido y distinto de 0.

                                ;
                                ; Imagen de la pila de la FPU
                                ; -----
                                ;          ST0          ST1          ST2          ST3
    fldz                                ;          0.0
    fld     coef_a                      ;          a          0.0
    fcomi   st0,st1                     ;          a          0.0

    jp      salida                      ; Si a es NaN, salir retornando 0.
    jne     a_no_es_0                  ; Si a != 0, aplicar la fórmula general.

    ; Si a == 0, comprobar si b es válido y b != 0.

    fstp    st0                        ;          0.0
    fld     coef_b                      ;          b          0.0
    fcomi   st0,st1                     ;          b          0.0
    jp      salida                      ; Si b es NaN, salir retornando 0.
    je      salida                      ; Si b == 0, salir retornando 0.

    ; Si a == 0 y b != 0, x1 = x2 = -c/b.

    fchs                                ;          -b          0.0
    fdivr   coef_c                      ;          -b/c        0.0
    fst     qword [ecx]                 ;          -b/c        0.0
    fstp    qword [esi]                 ;          0.0
    fst     qword [edx]                 ;          0.0
    fstp    qword [edi]                 ;

    mov     eax,1                      ; Salir indicando raíces reales.

    jmp     salida

```


a_no_es_0:

; Si a != 0, aplicar la fórmula general.

		Imagen de la pila de la FPU			

		ST0	ST1	ST2	ST3
fadd	st0,st0	2a			
fld	st0	2a	2a		
fadd	st0,st0	4a	2a		
fmul	coef_c	4ac	2a		
fld	coef_b	b	4ac	2a	
fmul	st0,st0	b^2	4ac	2a	
fsubrp	st1,st0	b^2-4ac	2a		
fldz		0.0	b^2-4ac	2a	
fcomip	st0,st1	b^2-4ac	2a		
jp	salida				
ja	complejas				

; Caso de raíces reales: $r1_real = (-b + \sqrt{b^2 - 4ac})/2a$; $r1_imag = 0.0$; $r2_real = (-b - \sqrt{b^2 - 4ac})/2a$; $r2_imag = 0.0$

		Imagen de la pila de la FPU			

		ST0	ST1	ST2	ST3
		D	2a		
fsqrt		R(D)	2a		
fld	coef_b	b	R(DIS)	2a	
fchs		-b	R(DIS)	2a	
fld	st0	-b	-b	R(DIS)	2a
fadd	st0,st2	-b+R(D)	-b	R(DIS)	2a
fdiv	st0,st3	(-b-R(D))/2a	-b	R(DIS)	2a
fstp	qword [ecx]	-b	R(DIS)	2a	
fsubrp	st1,st0	-b-R(D)	2a		
fdivrp	st1,st0	(-b-R(D))/2a			
fstp	qword [esi]				
fldz		0.0			
fst	qword [edx]	0.0			
fstp	qword [edi]				
mov	eax,1	; Salir indicando raíces reales.			
jmp	salida				

complejas:

; Caso de raíces complejas: $r1_real = -b/2a$; $r1_imag = \sqrt{-(b^2 - 4ac)}/2a$; $r2_real = -b/2a$; $r1_imag = -\sqrt{-(b^2 - 4ac)}/2a$

; Imagen de la pila de la FPU

```

;-----
;          ST0          ST1          ST2          ST3
fchs      ;   $-(b^2-4ac)$           2a
fsqrt     ;   $R(-D)$           2a
fdiv      st0,st1      ;   $R(-D)/2a$           2a
fst       qword [edx]   ;   $R(-D)/2a$           2a
fchs      ;   $-R(-D)/2a$           2a
fstp      qword [edi]   ;          2a
fld       coef_b        ;          b          2a
fchs      ;          -b          2a
fdivrp    st1,st0      ;   $-b/2a$ 
fst       qword [ecx]   ;   $-b/2a$ 
fstp      qword [esi]   ;

mov       eax,-1        ; Salir indicando raíces complejas.

salida: pop     edi
         pop     esi
         pop     ebp
         ret

```

4. Programación de 64 bits

4.1. Sumar char, short, int y long

Listado 4.1: b64_sumar_char_short_int_long.S

```

; Fichero: b64_sumar_char_short_int_long.S
;
; Escribe una función que retorne la suma de cuatro datos de tipo char, short, int y long
; que se reciben como argumentos. El prototipo de la función es
;
; long b64_sumar_char_short_int_long(char a, short b, int c, long d);
;
; La finalidad de este ejercicio es aclarar el tamaño de los tipos char, short, int y long
; en un compilador de 64 bits para Linux, cómo se reciben en la función argumentos de estos
; tipos y cómo se retorna el resultado.

global b64_sumar_char_short_int_long

section .text

b64_sumar_char_short_int_long:

    ; La función retorna un long. En un compilador de 64 bits para Linux el tipo long es un tipo
    ; con signo de 64 bits (en un compilador para Windows el tipo long tiene 32 bits y es el tipo
    ; long long el tipo con signo de 64 bits).
    ; Un compilador de 64 bits espera que las funciones envíen el valor de retorno a través del
    ; registro RAX. Por tanto, el resultado de la suma debe quedar en este registro.

    ; El primer argumento, a, es de tipo char. El tipo char es un tipo con signo de 8 bits.
    ; El primer argumento de una función se recibe en el registro RDI, pero al ser en este caso un
    ; argumento de 8 bits sólo consideraremos los 8 bits bajos de RDI, a los que podemos referirnos
    ; como DIL.
    ; Entonces, lo primero que hacemos es copiar el valor de DIL en RAX pero extendiendo el signo,
    ; de forma que el valor con signo que con 8 bits está representado en DIL pase a estar
    ; representado con 64 bits en RAX.

    movsx    rax, dil

```

```

; El segundo argumento, b, es de tipo short. El tipo short es un tipo con signo de 16 bits.
; El segundo argumento de una función se recibe en el registro RSI, pero al ser en este caso un
; argumento de 16 bits sólo consideraremos los 16 bits bajos de RSI, a los que podemos
; referirnos como SI.
; Como para sumar un valor a los 64 bits de RAX necesitamos que éste sea también de 64 bits,
; extendemos en signo el valor de SI a todo RSI. De esta forma, el valor con signo que con 16
; bits está representado en SI pasa a estar representado con 64 bits en RSI.

movsx    rsi, si

; A continuación sumamos los valores de los dos primeros argumentos. El resultado queda en RAX.

add      rax, rsi

; El tercer argumento, c, es de tipo int. El tipo int es un tipo con signo de 32 bits.
; El tercer argumento de una función se recibe en el registro RDX, pero al ser en este caso
; un argumento de 32 bits sólo consideraremos los 32 bits bajos de RDX, es decir, EDX.
; Como para sumar un valor a los 64 bits de RAX necesitamos que éste sea también de 64 bits,
; extendemos en signo el valor de EDX a RDX. De esta forma, el valor con signo que con 32
; bits está representado en EDX pasa a estar representado con 64 bits en RDX.

movsxd   rdx, edx

; A continuación sumamos a RAX el valor del tercer argumento.

add      rax, rdx

; El cuarto argumento, d, es de tipo long. En un compilador de 64 bits para Linux el tipo long
; es un tipo con signo de 64 bits.
; El cuarto argumento de una función se recibe en el registro RCX y como, en este caso, es un
; argumento de 64 bits, ocupa todo el registro. Entonces podemos sumarlo directamente a RAX.

add      rax, rcx

ret

```

4.2. Sumar ocho datos de tipo int

Listado 4.2: b64_sumar_8_ints.S

```

; Fichero: b64_sumar_8_ints.S
;
; Escribe una función que retorne la suma de ocho datos de tipo int. El prototipo de la función es
;
; int b64_sumar_8_ints(int a, int b, int c, int d, int e, int f, int g, int h);
;
; La finalidad de este ejercicio es aclarar cómo se trata el caso de que una función Linux de 64 bits
; tenga más de seis argumentos.
;
; Los argumentos adicionales al sexto se pasan a través de la pila.
;
; En caso de que ninguno de los argumentos que pasan a través de la pila sea de 256 (ningún argumento
; AVX), al llegar a la primera instrucción de la función el puntero de pila RSP está alineado en una
; dirección divisible entre 16.
;
; En caso de que alguno de los argumentos que pasan a través de la pila sea de 256 (argumento AVX), al
; llegar a la primera instrucción de la función el puntero de pila RSP está alineado en una dirección
; divisible entre 32.
;

```

```

; En cualquier caso, al llegar a la primera instrucción de la función, RSP apunta a la dirección de
; retorno y el séptimo argumento de la función está en RSP + 8. Los argumentos de tipo char, short,
; int y long ocupan 8 bytes en la pila.

    global b64_sumar_8_ints

    section .text

b64_sumar_8_ints:

    push    rbp
    mov     rbp, rsp

    ; Ahora RBP = RSP.
    ; RBP apunta al anterior RBP.
    ; La dirección de retorno está en RBP + 8.
    ; El séptimo argumento está en RBP + 16.
    ; El octavo argumento está en RBP + 24.

    mov     eax, edi
    add     eax, esi
    add     eax, edx
    add     eax, ecx
    add     eax, r8d
    add     eax, r9d
    add     eax, dword [rbp+16] ; Aunque el tipo int tiene 32 bits, cada argumento ocupa ocho bytes
    add     eax, dword [rbp+24] ; en la pila. Por tanto, es segundo está en
                                ; RBP + 16 + 8 = RBP + 24.

    pop     rbp
    ret

```

4.3. Añadir a una cadena con el DNI la letra del NIF

Listado 4.3: b64_anadir_letra_nif.S

```

; Fichero: b64_anadir_letra_nif.S
;
; Escribe una función en ensamblador que añada a una cadena de caracteres con los dígitos de un DNI
; la letra del NIF. Se supone que la hay espacio para el carácter adicional de la letra.
;
; Para obtener la letra del NIF hay que calcular el resto de la división del número de DNI entre 23 y
; tomar la letra de la siguiente tabla:
;
; Resto: 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
; -----
; Letra: T  R  W  A  G  M  Y  F  P  D  X  B  N  J  Z  S  Q  V  H  L  C  K  E
;
; El prototipo de la función es
;
; int b64_anadir_letra_nif(char *cadena_dni);
;
; cadena_dni apunta a la cadena con los caracteres numéricos del DNI. La cadena debe estar terminada
; con un carácter nulo. La función añadirá la letra del NIF detrás del último dígito del DNI y colocará
; un terminador nulo a continuación de ésta.
;
; La función retorna 1 si pudo realizar su trabajo correctamente y 0 en los siguientes casos:
;
; - El puntero cadena_dni es nulo.
; - La cadena del DNI contiene caracteres distintos a los de los dígitos decimales.

```

```

; - La cadena del DNI no tiene exactamente 8 dígitos.

        section      .data
letras_nif:
        db      "TRWAGMYFPDXBNJZSQVHLCKE" ; Tabla de conversión de resto a letra.

        section      .text

        global      b64_anadir_letra_nif

b64_anadir_letra_nif:

        test     rdi,rdi
        jz       error      ; Si el puntero a la cadena es nulo, error.

        ; Primero hay que convertir la cadena de caracteres en un número, convirtiendo cada carácter en
        ; su valor numérico entre 0 y 9 correspondiente y dándole a cada uno su peso de potencia de 10
        ; correspondiente a su posición. Para ello:
        ;
        ; - El registro EAX se pondrá inicialmente a 0.
        ; - Obtendremos los sucesivos dígitos decimales del DNI recorriendo la cadena
        ; desde el principio hasta el final (esto es, leemos los dígitos de izquierda
        ; a derecha.
        ; - Cada vez que leemos un nuevo carácter el valor de EAX se multiplicará por 10
        ; y al resultado de la multiplicación se sumará el valor numérico del carácter.
        ; - Al final del proceso EAX tendrá el valor numérico del DNI.
        ; - El proceso terminará cuando se encuentre el terminador nulo de la cadena o
        ; cuando se encuentre un error.

        xor      eax,eax      ; EAX se usará para ir obteniendo el valor numérico del DNI.
        xor      ecx,ecx      ; ECX se usará como contador de caracteres.
        xor      r8d,r8d      ; R8L se usará para leer en él los sucesivos caracteres de la cadena. Se
        ; pone R8D a cero porque dentro del bucle necesitaremos que R8D valga
        ; lo mismo que R8L.
        mov      esi,10      ; ESI es el valor por el que multiplicar el resultado parcial previo.

sig_caracter:
        mov      r8b,[rdi]    ; Leer un carácter de la cadena.
        test     r8b,r8b
        jz       fin_cadena   ; Si se encontró el terminador nulo, salir del bucle.

        inc      ecx          ; Incrementar el contador de caracteres válidos encontrados.
        cmp      ecx,8
        ja       error        ; Si ya van más de 8, error.

        sub      r8b,'0'      ; Convertir el nuevo carácter a su valor numérico.
        cmp      r8b,10
        ja       error        ; Si es mayor que 10, error.

        mul      esi          ; Multiplicar por 10 el resultado parcial previo.
        add      eax,r8d      ; Sumarle el valor numérico del nuevo dígito. Como R8D se puso a 0
        ; antes de entrar en el bucle y la modificación de R8L no afecta al
        ; resto de bits de R8D, el valor de R8D es igual al de R8L.

        inc      rdi          ; Avanzar el puntero al siguiente carácter.
        jmp      sig_caracter ; Repetir con el siguiente carácter.

fin_cadena:
        cmp      ecx,8        ; Si hay menos de 8 dígitos, error.
        jb       error
        mov      esi,23      ; Cargar es ESI el divisor.

```

```

    xor     edx,edx        ; Poner a cero la parte alta del dividendo.
    div     esi            ; Dividir EDX:EAX entre 23: el cociente queda en EAX y el resto en EDX.
    mov     rsi,letras_nif ; RSI apunta a la primera letra de la cadena de conversión.
    mov     al,[rsi+rdx]   ; El resto en RDX se usa como desplazamiento para leer a la letra
                          ; adecuada. Ahora AL tiene la letra del NIF.
    mov     [rdi],al       ; A la salida del bucle, RDI apunta al terminador nulo de la cadena
                          ; del DNI. El terminador se sustituye por la letra del NIF.
    mov     byte [rdi+1],0 ; Se coloca un terminador nulo detrás de la letra del NIF.
    mov     eax,1          ; Salir indicando éxito.
    ret

error: xor     eax,eax      ; En caso de error, retornar con EAX = 0.
       ret

```

4.4. Comparar dos cadenas ASCII sin diferenciar mayúsculas de minúsculas

Listado 4.4: b64_comparar_cadenas_insensible_mayusculas_minusculas.S

```

; Fichero: b64_comparar_cadenas_insensible_mayusculas_minusculas.S
;
; Escribe una función que compare dos cadenas de caracteres sin diferenciar entre mayúsculas y
; minúsculas. El prototipo es
;
; int b64_comparar_cadenas_insensible_mayusculas_minusculas(const char* s1,
;                                                         const char* s2);
;
; donde
;
; s1 apunta a la primera cadena. La cadena debe estar terminada con un byte a 0.
;
; s2 apunta a la segunda cadena. La cadena debe estar terminada con un byte a 0.
;
; La función retorna:
;
; 1 si la primera cadena es lexicográficamente mayor que la segunda.
; 0 si las dos cadenas son iguales.
; -1 si la primera cadena es lexicográficamente menor que la segunda.
; -2 si alguno de los punteros de entrada es nulo.

global b64_comparar_cadenas_insensible_mayusculas_minusculas

section .text

b64_comparar_cadenas_insensible_mayusculas_minusculas:

    mov     eax,-2          ; Si algún puntero es nulo, retornar con EAX = -2.

    test    rdi,rdi
    jz      error           ; Si s1 es nulo, retornar -2.

    test    rsi,rsi
    jz      error           ; Si s2 es nulo, retornar -2.

    xor     eax,eax         ; EAX = 0.

bucle: mov     al,[rdi]      ; Leer un carácter de la cadena s1 en AL.
       call    convertir_a_mayuscula ; Convertir AL a mayúscula.
       mov     cl,al        ; Guardar en CL.
       mov     al,[rsi]     ; Leer un carácter de la cadena s2 en AL.

```

```

    call    convertir_a_mayuscula    ; Convertir AL a mayúscula.

    cmp     cl,al                    ; Comparar el carácter de s1 con el de s2.
    jne     cadenas_distintas        ; Si son diferentes saltar a la etiqueta cadenas_distintas.
                                        ; Si son iguales la ejecución continúa en la siguiente línea.

    cmp     al,0                     ; Si además de ser iguales entre sí son iguales a 0 significa
    je      cadenas_iguales          ; que se ha encontrado simultáneamente el final de ambas
                                        ; cadenas y que ambas cadenas son iguales y se retorna con
                                        ; EAX = 0.

    inc     rdi                      ; Si los caracteres son iguales pero no iguales a 0 hay que
    inc     rsi                      ; seguir comparando caracteres. Incrementar los punteros a
    jmp     bucle                    ; ambas cadenas y continuar con la comparación.

cadenas_distintas:
    mov     eax,1                    ; Preparar en EAX la respuesta para caso cadena 1 > cadena 2.
    mov     ecx,-1                   ; Preparar en ECX la respuesta para caso cadena 1 < cadena 2.
    cmovb   eax,ecx                  ; Si cadena 1 > cadena 2 en EAX queda el valor 1.
                                        ; Si cadena 1 < cadena 2 el valor de EAX cambia a -1.

cadenas_iguales:
error:  ret

; La función auxiliar convertir_a_mayuscula convierte a mayúscula el carácter en AL. El resultado queda
; en AL. Si el valor original de AL no es el ASCII de una minúscula, AL no cambia.

convertir_a_mayuscula:
    cmp     al,'a'                   ; Comparar el carácter de entrada con 'a'.
    jb      no_minuscula             ; Si el carácter está por debajo de 'a' salir sin cambiarlo.
    cmp     al,'z'                   ; Comparar el carácter de entrada con 'z'.
    ja      no_minuscula             ; Si el carácter está por encima de 'z' salir sin cambiarlo.
    sub     al,'a'-'A'               ; Si está entre 'a' y 'z', transformar en mayúscula.
no_minuscula:
    ret

```

4.5. Sumar filas y columnas de una matriz de datos de tipo int

Listado 4.5: b64_sumar_filas_y_columnas.S

```

; Fichero: b64_sumar_filas_y_columnas_64.S
;
; Escribir una función que obtenga la suma de los elementos que se encuentran a lo largo de cada fila
; de una matriz bidimensional compuesta por elementos de tipo int. Asimismo, la función debe obtener
; la suma de los elementos que se encuentran a lo largo de cada columna de la matriz. El prototipo
; de la función es
;
; int b64_sumar_filas_y_columnas_64(const int *matriz,
;                                   int num_filas,
;                                   int num_columnas,
;                                   int* suma_filas,
;                                   int* suma_columnas);
;
; donde
;
; matriz      es un puntero al primer elemento de la matriz. Los elementos de la matriz están
;              almacenados por filas.
;
; num_filas   es el número de filas que componen la matriz.
;
; num_columnas es el número de columnas que componen la matriz.
;

```

```

; suma_filas      apunta a la posición a partir de la cual deben quedar almacenados los resultados de
;                  sumar los elementos que se encuentran a lo largo de cada fila de la matriz.
;
; suma_columnas   apunta a la posición a partir de la cual deben quedar almacenados los resultados de
;                  sumar los elementos que se encuentran a lo largo de cada columna de la matriz.
;
; La función retorna 1 si puede realizar su trabajo con éxito y 0 si hay algún error en los argumentos.
;
; Ejemplo:
;
; int m[3][4] = {{1,  2,  3,  4},
;                {5,  6,  7,  8},
;                {9, 10, 11, 12}};
; int sf[4];
; int sc[3];
;
; b64_sumar_filas_y_columnas((int*)m, 3, 4, sf, sc);
;
; Deja en sf {10, 26, 42} y en sc {15, 18, 21, 24}.

global b64_sumar_filas_y_columnas

section .text

b64_sumar_filas_y_columnas:

    xor     eax,eax                ; Retornar EAX = 0 si hay algún argumento no válido.

    test    rdi,rdi
    jz      error                 ; Si el puntero matriz es nulo, retornar indicando error.

    cmp     rsi,0
    jle     error                 ; Si num_filas <= 0, retornar indicando error.

    cmp     rdx,0
    jle     error                 ; Si num_columnas <= 0, retornar indicando error.

    test    rcx,rcx
    jz      error                 ; Si suma_filas es nulo, retornar indicando error.

    test    r8,r8
    jz      error                 ; Si suma_columnas es nulo, retornar indicando error.

    ; Sumar filas. La matriz está almacenada por filas. Justo en la posición apuntada por RDI está
    ; el primer elemento de la primera fila, al que le siguen de forma consecutiva los demás
    ; elementos de la primera fila. Después del último elemento de cada fila está situado el
    ; primer elemento de la fila siguiente. Por tanto, para recorrer las filas de la matriz
    ; basta ir sumando 4 para pasar de un elemento al siguiente en el orden requerido.

    mov     r9,rdi                ; Copiar el puntero al comienzo de la matriz en R9.

    xor     r10,r10                ; R10 hará de contador de filas. Inicialmente a 0.

siguiente_fila:
    xor     r11,r11                ; R11 hará de contador de columnas. Inicialmente a 0.
    xor     eax,eax                ; EAX se usará para sumar en él todos los elementos de una fila.

recorrer_fila:
    add     eax,[rdi]              ; Sumar siguiente elemento.
    add     rdi,4                  ; Apuntar al siguiente elementos.
    inc     r11                    ; Incrementar contador de columnas.
    cmp     r11,rdx                ; Comparar con número de columnas de la matriz.
    jl      recorrer_fila         ; Si contador cols. < núm. de cols., seguir recorriendo esta fila.

```



```

mov    [rcx],eax        ; Almacenar resultado en suma_filas.
add    rcx,4            ; Apuntar al siguiente elemento del vector suma_columnas.
inc    r10              ; Incrementar contador de filas.
cmp    r10,rsi          ; Comparar con número de filas.
jl     siguiente_filas  ; Si contador filas < núm. filas, pasar a la siguiente fila.

; Sumar columnas. Para "descender" a lo largo de una columna hay que sumar al puntero a la
; matriz el número de elementos que hay en cada fila multiplicado por el tamaño de cada
; elemento (4 en este caso). Una vez recorrida una columna, para pasar al comienzo de la
; siguiente columna, recuperamos el puntero al comienzo de la columna recién recorrida y
; sumamos 4.

xor     r11,r11          ; R11 hará de contador de columnas. Inicialmente a 0.
siguiente_columna:
xor     r10,r10          ; R10 hará de contador de filas. Inicialmente a 0.
mov     rdi,r9           ; RDI apunta ahora al primer elemento de la columna a sumar.
xor     eax,eax          ; EAX se usará para sumar en él todos los elementos de una columna.
recorrer_columna:
add     eax,[rdi]        ; Sumar siguiente elemento de columna.
lea     rdi,[rdi+rdx*4]   ; RDI = RDI + RDX*4 = RDI + (número de columnas)*4. Con esto RDI pasa
; a apuntar al siguiente elemento de esta columna.
inc     r10              ; Incrementar contador de filas.
cmp     r10,rsi          ; Comparar con número de filas.
jl     recorrer_columna  ; Si contador filas < núm. filas seguir recorriendo esta columna.
mov     [r8],eax         ; Almacenar resultado en suma_columna.
add     r8,4             ; Apuntar al siguiente elemento del vector suma_columna.
add     r9,4             ; R9 apunta ahora al primer elemento de la siguiente columna.
inc     r11              ; Incrementar contador de columnas.
cmp     r11,rdx          ; Comparar con el número de columnas.
jl     siguiente_columna ; Si contador columnas < núm. columnas pasar a la siguiente columna.

mov     eax,1            ; Indicar éxito.

error:  ret

```

5. Programación SSE con escalares

5.1. Calcular el área y el volumen de una esfera

Listado 5.1: sse_area_volumen_esfera.S

```

; Fichero: sse_area_volumen_esfera.S
;
; Escribe una función que calcule el área y el volumen de una esfera dado su radio. El prototipo de la
; función es
;
; int sse_area_volumen_esfera(double radio,
;                               double* area,
;                               double* volumen);
;
; donde
;
; radio es el radio de la esfera.
;
; area apunta a la variable donde debe almacenarse el área.
;
; volumen apunta a la variable donde debe almacenarse el volumen.
;
; La función retorna 1 si realizó los cálculos con éxito y 0 si hay algún error en los argumentos

```

```

; (el radio es negativo o NaN o alguno de los punteros es nulo).
;
; Área = 4*PI*r^2
;
; Volumen = 4/3*PI*r^3

global sse_area_volumen_esfera

section .data

; Constantes usadas en los cálculos.
; Queremos que las constantes queden almacenadas como números en punto flotante de doble
; precisión. Entoces usamos la directiva DQ, ya que ésta incializa datos de tamaño "quad word",
; es decir, de 64 bits, que es el tamaño de un dato en punto flotante de doble precisión. El
; punto decimal presente en las constantes le dice al ensamblador que debe codificar los
; números en punto flotante. Si no aparece el punto decimal, el ensamblador codifica las
; constantes como números enteros, que no es lo que queremos en este caso. Esa es la razón por
; la que se ha incluido un punto decimal en las constantes 3 y 4 aunque a primera vista pudiera
; parecer que no es necesario.

pi:    dq    3.14159265358979
tres:  dq    3.0
cuatro: dq    4.0

section .text

sse_area_volumen_esfera:

; El argumento radio se recibe en XMM0 (64 bits bajos).
; El argumento area se recibe en RDI.
; El argumento volumen se recibe en RSI.

xor    eax,eax    ; Si error, salir con RAX = 0.

test   rdi,rdi
jz     error      ; Si puntero area es nulo, salir indicando error.

test   rsi,rsi
jz     error      ; Si puntero area es nulo, salir indicando error.

xorpd  xmm1,xmm1  ; XMM1 = 0.0.
comisd xmm0,xmm1  ; Comparar radio con 0.0.
jp     error      ; Si radio es NaN, salir indicando error.
jb     error      ; Si radio < 0, salir indicando error.

movsd  xmm1,xmm0  ; Copiar radio en XMM1.

mulsd  xmm1,xmm1  ; XMM1 = radio^2.
mulsd  xmm1,[cuatro] ; XMM1 = 4*radio^2.
mulsd  xmm1,[pi]   ; XMM1 = 4*pi*radio^2 = área.
movsd  [rdi],xmm1  ; Almacenar área.

mulsd  xmm1,xmm0  ; XMM1 = 4*pi*radio^3.
divsd  xmm1,[tres] ; XMM1 = 4/3*pi*radio^3 = volumen.
movsd  [rsi],xmm1 ; Almacenar volumen.

mov    eax,1      ; Salir indicando éxito.

error: ret

```

5.2. Raíces de un polinomio de segundo grado

Listado 5.2: sse_raices_polinomio_segundo_grado.S

```
; Fichero: sse_raices_polinomio_segundo_grado.S
;
; Escribe una función que obtenga las raíces de un polinomio de segundo grado  $a*x^2 + b*x + c$ 
; tanto si éstas son reales como si son complejas. El prototipo de la función es:
;
; int sse_raices_polinomio_segundo_grado(double coef_a,
;                                     double coef_b
;                                     double coef_c,
;                                     double *r1_real,
;                                     double *r1_imag,
;                                     double *r2_real,
;                                     double *r2_imag);
;
; donde
;
; coef_a es el coeficiente a del polinomio.
;
; coef_b es el coeficiente b del polinomio.
;
; coef_c es el coeficiente c del polinomio.
;
; r1_real es un puntero a la variable donde debe almacenarse la parte real de la primera raíz.
;
; r1_imag es un puntero a la variable donde debe almacenarse la parte imaginaria de la primera raíz.
; Si las raíces son reales, debe almacenarse 0.
;
; r2_real es un puntero a la variable donde debe almacenarse la parte real de la segunda raíz.
;
; r2_imag es un puntero a la variable donde debe almacenarse la parte imaginaria de la segunda raíz.
; Si las raíces son reales, debe almacenarse 0.
;
; La función retorna 1 si las raíces son reales, -1 si son complejas y 0 si alguno de los argumentos
; no es válido (a, b o c es NaN, a = b = 0 o alguno de los punteros es nulo).
;
; Casos posibles:
;
; Si a == 0 y b == 0: tenemos c = 0 y no calculamos ninguna raíz.
; Si a == 0 y b != 0: *r1_real = *r2_real = -c/b, *r1_imag = *r2_imag = 0.
; Si a != 0 aplicamos la fórmula general  $r_{1,2} = (-b \pm \sqrt{b^2 - 4ac}) / (2a)$  distinguiendo
; entonces los casos  $b^2 - 4ac < 0 \Rightarrow$  complejas y  $b^2 - 4ac \geq 0 \Rightarrow$  reales.

global sse_raices_polinomio_segundo_grado

section .text

sse_raices_polinomio_segundo_grado:

    ; coef_a llega en XMM0.
    ; coef_b llega en XMM1.
    ; coef_c llega en XMM2.
    ; r1_real llega en RDI.
    ; r1_imag llega en RSI.
    ; r2_real llega en RDX.
    ; r2_imag llega en RCX.

    xor    eax, eax    ; Si error, salir con EAX = 0.
```

```

test    rdi,rdi
jz      salida      ; Si r1_real es nulo salir con EAX a 0 => error.

test    rsi,rsi
jz      salida      ; Si r1_imag es nulo salir con EAX a 0 => error.

test    rdx,rdx
jz      salida      ; Si r2_real es nulo salir con EAX a 0 => error.

test    rcx,rcx
jz      salida      ; Si r2_imag es nulo salir con EAX a 0 => error.

xorpd   xmm3,xmm3    ; XMM3 = (0.0, 0.0, 0.0, 0.0)
comisd  xmm0,xmm3    ; Comparar el double en la parte baja de XMM0 (coef_a) y el double en la
                    ; parte baja de XMM3 (0.0).
jp      salida      ; Si coef_a es un Nan salir indicando error.
jne     a_no_es_0    ; Si coef_a no es 0.0 resolver el caso general.

; Caso coef_a == 0.0.

comisd  xmm1,xmm3    ; Comparar el double en la parte baja de XMM1 (coef_b) y el double en la
                    ; parte baja de XMM3 (0.0).
jp      salida      ; Si coef_b es un NaN salir indicando error.
je      salida      ; Si coef_b es 0.0 la ecuación es c = 0 => salir indicando error.

; Caso coef_a == 0.0 y coef_b != 0.

comisd  xmm2,xmm3    ; Comparar el double en la parte baja de XMM2 (coef_c) y el double en la
                    ; parte baja de XMM3 (0.0).
jp      salida      ; Si coef_c es un NaN, salir indicando error.

movsd   [rsi],xmm3    ; Almacenar 0.0 en la parte imaginaria de ambas raíces.
movsd   [rcx],xmm3
subsd   xmm3,xmm2    ; XMM3 = -c
divsd   xmm1,xmm3    ; XMM3 = -c/b
movsd   [rdi],xmm3    ; Almacenar -c/b en la parte real de ambas raíces.
movsd   [rdx],xmm3

mov     eax,1        ; Salir indicando que las raices son reales.
jmp     salida

; Caso general.

a_no_es_0:
comisd  xmm1,xmm3    ; Comparar el double en la parte baja de XMM1 (coef_b) y el double en la
                    ; parte baja de XMM3 (0.0).
jp      salida      ; Si coef_b es un NaN salir indicando error.
comisd  xmm2,xmm3    ; Comparar el double en la parte baja de XMM2 (coef_c) y el double en la
                    ; parte baja de XMM3 (0.0).
jp      salida      ; Si coef_c es un NaN salir indicando error.

movsd   xmm4,xmm3    ; XMM4 = 0.0
movsd   xmm5,xmm1    ; XMM5 = b
subsd   xmm4,xmm1    ; XMM4 = -b
addsd   xmm0,xmm0    ; XMM0 = 2*a
mulsd   xmm5,xmm5    ; XMM5 = b^2
movsd   xmm6,xmm0    ; XMM6 = 2*a
addsd   xmm6,xmm6    ; XMM6 = 4*a
mulsd   xmm6,xmm2    ; XMM6 = 4*a*c
subsd   xmm5,xmm6    ; XMM5 = b^2 - 4*a*c = d.
comisd  xmm5,xmm3    ; Compara XMM5 con 0.0

```

```

    jb      complejas ; Si XMM5 es menor que 0.0 las raíces son complejas.

    ; Caso de raíces reales.

    sqrtss  xmm5,xmm5 ; XMM5 = sqrt(d)
    movss   xmm1,xmm4 ; XMM1 = -b
    addss   xmm1,xmm5 ; XMM1 = -b + sqrt(d)
    divss   xmm1,xmm0 ; XMM1 = (-b + sqrt(d))/(2*a)
    subss   xmm4,xmm5 ; XMM4 = -b - sqrt(d)
    divss   xmm4,xmm0 ; XMM4 = (-b - sqrt(d))/(2*a)
    movss   [rdi],xmm1 ; Almacenar la parte real de la primera raíz.
    movss   [rsi],xmm3 ; Almacenar 0.0 en la parte imaginaria de la primera raíz.
    movss   [rdx],xmm4 ; Almacenar la parte real de la segunda raíz.
    movss   [rcx],xmm3 ; Almacenar 0.0 en la parte imaginaria de la segunda raíz.

    mov     eax,1      ; Salir indicando raíces reales.
    jmp     salida

    ; Caso de raíces complejas.

complejas:
    movss   xmm6,xmm3 ; XMM6 = 0.0
    subss   xmm6,xmm5 ; XMM6 = -d
    sqrtss   xmm6,xmm6 ; XMM6 = sqrt(-d)
    divss   xmm4,xmm0 ; XMM4 = -b/(2*a)
    divss   xmm6,xmm0 ; XMM6 = sqrt(-d)/(2*a)
    subss   xmm3,xmm6 ; XMM3 = -sqrt(-d)/(2*a)
    movss   [rdi],xmm4 ; Almacenar la parte real de la primera raíz.
    movss   [rsi],xmm6 ; Almacenar la parte imaginaria de la primera raíz.
    movss   [rdx],xmm4 ; Almacenar la parte real de la segunda raíz.
    movss   [rcx],xmm3 ; Almacenar la parte imaginaria de la segunda raíz.

    mov     eax,-1     ; Salir indicando raíces reales.

salida: ret

```

6. Programación SSE SIMD

6.1. Media y desviación típica

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n}$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n - 1}}, \quad n > 1$$

Si $n = 1$ tomaremos $\sigma = 0$.

Listado 6.1: sse_media_y_desviacion_tipica.S

```

; Fichero: sse_media_y_desviacion_tipica.S
;
; Escribe una función que calcule la media aritmética y la desviación típica de una muestra de
; datos de tipo float. El prototipo de la función es
;
; int sse_media_y_desviacion_tipica(const float* datos,

```

```

;                                     int numero_datos,
;                                     float* media,
;                                     float* desviacion);
;
; donde
;
; datos          es un puntero al primero de los datos del array de datos.
;
; numero_datos   es el número de datos en el array.
;
; media          apunta a la variable donde debe almacenarse la media aritmética.
;
; desviacion     apunta a la variable donde debe almacenarse la desviación típica.
;
; La función retorna 1 si realizó los cálculos con éxito y 0 en caso de error.

global sse_media_y_desviacion_tipica
section .text

sse_media_y_desviacion_tipica:

    xor     eax,eax                ; Si error, retornar RAX = 0.

    test    rdi,rdi
    jz      error                 ; Si puntero datos es nulo, salir indicando error.

    test    rdi,15
    jnz     error                 ; Si puntero datos no está alineado, salir indicando error.

    cmp     esi,1
    jl      error                 ; Si numero_datos < 1, salir indicando error.

    test    rdx,rdx
    jz      error                 ; Si puntero media es nulo, salir indicando error.

    test    rcx,rcx
    jz      error                 ; Si puntero desviacion es nulo, salir indicando error.

    mov     r8,rdi                ; Hacer una copia del puntero datos en R8.

    ; ----- Calcular la media -----

    ; Para calcular la media tenemos que sumar todos los datos y luego dividir entre el número de
    ; datos.
    ; Primero, los datos se suman en grupos de cuatro mediante instrucciones SSE. Luego, si quedan
    ; datos (el número de datos no es divisible entre 4), el resto de datos se suman de uno en uno.
    ; Finalmente, la suma se divide entre el número de datos.
    ;
    ; El registro XMM0 se usará para acumular la suma de grupos de cuatro datos. Primero se
    ; inicializa a 0.

    xorps   xmm0,xmm0             ; XMM0 = (0.0, 0.0, 0.0, 0.0)

    ; Calcular el número de grupos de cuatro datos.

    mov     eax,esi                ; EAX = numero_datos.
    shr     eax,2                  ; EAX = numero_datos/4.
    mov     r9d,eax                ; Copiar en R9D para uso posterior.
    jz      resto_de_datos         ; Si hay menos de 4 datos, sumar de uno en uno.

    ; Sumar de cuatro en cuatro.

```

```

bucle_media_1:
    addps    xmm0,[rdi]      ; Sumar cuatro datos a XMM0.
    add      rdi,16          ; Avanzar a los 4 siguientes.
    dec      eax             ; Decrementar núm. bloques de 4.
    jnz      bucle_media_1   ; Si no llegó a 0, repetir.

    ; Al salir del bucle, hay que sumar entre sí los cuatro datos en XMM0.
    ; Los datos se suman en "horizontal".
    ;
    ; Llamamos (a, b, c, d) a los cuatro datos en XMM0:
    ;
    ; XMM0 = (a, b, c, d)

    haddps   xmm0,xmm0       ; XMM0 = (a+b, c+d, a+b, c+d)
    haddps   xmm0,xmm0       ; XMM0 = (a+b+c+d, a+b+c+d, a+b+c+d, a+b+c+d)

    ; Ahora en los 32 bits bajos de XMM0 tenemos la suma de los todos los datos excepto los que
    ; no han podido agruparse de cuatro en cuatro. Esos, si los hay, se suman a continuación.

resto_de_datos:
    mov      eax,esi         ; EAX = numero_datos.
    and      eax,3           ; EAX = resto numero_datos/4.
    mov      r10d,eax        ; Copiar en R10D para uso posterior.
    jz       no_mas_datos    ; Si resto numero_datos/4 = 0, no sumar más.

bucle_media_2:
    addss    xmm0,[rdi]      ; Sumar un dato más.
    add      rdi,4           ; Avanzar al siguiente.
    dec      eax             ; Decrementar número de datos restantes.
    jnz      bucle_media_2   ; Si no llegó a 0, repetir.

no_mas_datos:

    ; La suma de todos los datos está ahora en los 32 bits bajos de XMM0.

    cvtsi2ss xmm1,esi        ; Convertir numero_datos a float.
    divss    xmm0,xmm1       ; Dividir suma entre número de datos.

    movss    [rdx],xmm0      ; Almacenar la media.

    ; ----- Calcular la desviación típica -----

    xorps    xmm1,xmm1       ; XMM1 = (0.0, 0.0, 0.0, 0.0)

    test     r9d,r9d         ; R9D tiene una copia de numero_datos/4.
    jz       resto_de_datos_2 ; Si hay menos de 4 datos, procesar de uno en uno.

    shufps   xmm0,xmm0,0     ; Repetir la media cuatro veces en XMM0:
                                ; XMM0 = (media, media, media, media).

bucle_desviacion_1:
    movaps   xmm2,[r8]       ; Tomar cuatro datos.
    subps    xmm2,xmm0       ; Restar la media a cada uno de los cuatro.
    mulps    xmm2,xmm2       ; Elevar al cuadrado los resultados de las restas.
    addps    xmm1,xmm2       ; Acumular estos cuatro resultados.
    add      r8,16           ; Avanzar a los cuatro datos siguientes.
    dec      r9d             ; Decrementar número de grupos de cuatro restantes.
    jnz      bucle_desviacion_1 ; Si no llegó a 0, repetir.

    ; Al salir del bucle, hay que sumar entre sí los cuatro datos en XMM1.

```

```

haddps    xmm1,xmm1
haddps    xmm1,xmm1

resto_de_datos_2:
    test    r10d,r10d        ; R10D tiene una copia del resto de numero_datos/4.
    jz      no_mas_datos_2    ; Si resto numero_datos/4 = 0, no procesar más.

bucle_desviacion_2:
    movss   xmm2,[r8]        ; Tomar un dato.
    subss   xmm2,xmm0        ; Restarle la media.
    mulss   xmm2,xmm2        ; Elevar al cuadrado.
    addss   xmm1,xmm2        ; Acumular este nuevo resultado.
    add     r8,4              ; Avanzar al siguiente dato.
    dec     r10d              ; Decrementar número de datos restantes.
    jnz     bucle_desviacion_2 ; Si no llegó a 0, repetir.

no_mas_datos_2:

    ; La suma de los cuadrados de la diferencia entre cada dato y la media está ahora en los
    ; 32 bits bajos de XMM1.

    dec     esi              ; ESI = numero_datos - 1.
    jz      un_solo_dato     ; Si numero_datos - 1 = 0, desviación = 0.
    cvtsi2ss xmm2,esi        ; Convertir a punto flotante.
    divss   xmm1,xmm2        ; Dividir entre numero_datos - 1.
    sqrtss  xmm1,xmm1        ; Calcular raíz cuadrada.

un_solo_dato:
    movss   [rcx],xmm1       ; Almacenar desviación típica.

    mov     eax,1            ; Salir indicando éxito.

error: ret

```

6.2. Multiplicar matriz 4×4 por vectores 4×1

En aplicaciones de gráficos 3D por ordenador, para representar puntos en el espacio tridimensional se usan comúnmente coordenadas homogéneas. Las coordenadas homogéneas de un punto dado por sus coordenadas tridimensionales (x, y, z) se representan por un vector columna 4×1 , $(x, y, z, w)^T$. La coordenada adicional w es habitualmente 1. Las transformaciones de rotación, traslación, escalado, proyección de perspectiva, etc. se representan por matrices 4×4 . Para transformar un punto se multiplica la matriz representativa de la transformación por el vector de coordenadas homogéneas del punto. Las operaciones SSE se adaptan bien a este tipo de aplicaciones, ya que los registros XMM pueden usarse para almacenar las cuatro coordenadas homogéneas de los puntos o las filas de las matrices de transformación. Las operaciones en paralelo permiten acelerar las operaciones de multiplicación de matriz por vector.

Listado 6.2: sse_multiplicar_matriz_4x4_vectores_4x1.S

```
; Fichero: sse_multiplicar_matriz_4x4_vectores_4x1.S  
;  
; Escribe una función que permita multiplicar una matriz 4x4 por un conjunto de vectores 4x1. El  
; prototipo de la funcion es  
;  
; int sse_multiplicar_matriz_4x4_vectores_4x1(float matriz[4][4],  
; float* vectores_entrada,
```



```

;                                     float* vectores_salida,
;                                     int numero_vectores);
;
;  matriz                apunta al primero de los elementos de la matriz. Los elementos de la matriz
;                        están almacenados por filas.
;
;  vectores_entrada      apunta al primer elemento del primer vector por el que se multiplicará la
;                        matriz. Los demás elementos del primer vector están almacenados a continuación
;                        y a éstos siguen los elementos de los demás vectores.
;
;  vectores_salida       apunta a la posición a partir de la cual deben quedar almacenados los vectores
;                        producto.
;
;  numero_vectores       indica el numero de vectores a multiplicar.
;
; La función retorna 1 si puede realizar su trabajo y 0 si alguno de los argumentos es incorrecto.
;
; M = m11 m12 m13 m14
;      m21 m22 m23 m24
;      m31 m32 m33 m34
;      m41 m42 m43 m44
;
; x = x1
;      x2
;      x3
;      x4
;
; y = M*x = m11*x1+m12*x2+m13*x3+m14*x4
;           m21*x1+m22*x2+m23*x3+m24*x4
;           m31*x1+m32*x2+m33*x3+m34*x4
;           m41*x1+m42*x2+m43*x3+m44*x4

global sse_multiplicar_matriz_4x4_vectores_4x1

section .text

sse_multiplicar_matriz_4x4_vectores_4x1:

    xor    eax,eax                ; RAX = 0 => si hay error, retornar 0.

    test   rdi,rdi
    jz     error                  ; Salir con RAX = 0 si ptr_matriz es nulo.
    test   rdi,15
    jnz    error                  ; Salir con RAX = 0 si ptr_matriz no está alineado.

    test   rsi,rsi
    jz     error                  ; Salir con RAX = 0 si ptr_vectores_entrada es nulo.
    test   rsi,15
    jnz    error                  ; Salir con RAX = 0 si ptr_vectores_entrada no está alineado.

    test   rdx,rdx
    jz     error                  ; Salir con RAX = 0 si ptr_vectores_salida es nulo.
    test   rdx,15
    jnz    error                  ; Salir con RAX = 0 si ptr_vectores_salida no está alineado.

    cmp    ecx,0
    jle    error                  ; Salir con RAX = 0 si numero_vectores <= 0.

    ; Cargar las filas de la matriz en XMM1, XMM2, XMM3, XMM4.

    movaps xmm1,[rdi]            ; XMM1 = (m11, m12, m13, m14)

```

```

movaps xmm2,[rdi+16] ; XMM2 = (m21, m22, m23, m24)
movaps xmm3,[rdi+32] ; XMM3 = (m31, m32, m33, m34)
movaps xmm4,[rdi+48] ; XMM4 = (m41, m42, m43, m44)

bucle: ; Cargar un nuevo vector de entrada en XMM0.

movaps xmm0,[rsi+rax] ; XMM0 = (x1, x2, x3, x4)

; Copiar las filas de la matriz en XMM5, XMM6, XMM7 y XMM8.

movaps xmm5,xmm1
movaps xmm6,xmm2
movaps xmm7,xmm3
movaps xmm8,xmm4

; Multiplicar cada fila de la matriz por el vector.

mulps xmm5,xmm0 ; XMM5 = (m11*x1, m12*x2, m13*x3, m14*x4)
mulps xmm6,xmm0 ; XMM6 = (m21*x1, m22*x2, m23*x3, m24*x4)
mulps xmm7,xmm0 ; XMM7 = (m31*x1, m32*x2, m33*x3, m34*x4)
mulps xmm8,xmm0 ; XMM8 = (m41*x1, m42*x2, m43*x3, m44*x4)

; Sumar los productos de cada fila de la matriz por el vector.

haddps xmm5,xmm6 ; XMM5 = (m11*x1+m12*x1, m13*x3+m14*x4, m21*x1+m22*x2, m23*x3+m24*x4)
haddps xmm7,xmm8 ; XMM7 = (m31*x1+m32*x2, m33*x3+m34*x4, m41*x1+m42*x2, m43*x3+m44*x4)
haddps xmm5,xmm7 ; XMM5 = (m11*x1 + m12*x2 + m13*x3 + m14*x4,
; m21*x1 + m22*x2 + m23*x3 + m24*x4,
; m31*x1 + m32*x2 + m33*x3 + m34*x4,
; m41*x1 + m42*x2 + m43*x3 + m44*x4)

movaps [rdx+rax],xmm5 ; Almacenar en memoria el vector resultante.

add rax,16 ; En la siguiente iteración, acceder al siguiente
; vector de entrada y al siguiente vector de salida.

dec ecx ; Decrementar el número de vectores.
jnz bucle ; Si no llegó a cero, repetir.

mov eax, 1 ; Salir indicando éxito.

error: ret

```

6.3. Multiplicar dos matrices 4×4

Como se comentó en el ejercicio 6.2, en gráficos 3D las transformaciones de rotación, traslación, escalado, proyección de perspectiva, etc. se representan por matrices 4×4 . Una secuencia de transformaciones se obtiene multiplicando las matrices representativas de cada una de las transformaciones de la secuencia. De esta forma, puede aplicarse una secuencia de transformaciones a un conjunto de puntos hallando primero el producto de las matrices de la secuencia y luego multiplicando la matriz resultante por los vectores de las coordenadas homogéneas de los puntos a transformar. Por tanto, una función de multiplicación de matrices de 4×4 resulta útil para estas aplicaciones.

Listado 6.3: sse_multiplicar_matrices_4x4.S

```

; Fichero: sse_multiplicar_matrices_4x4.S
;
; Escribe una función que permita multiplicar dos matrices 4x4. El prototipo de la función es

```

```

;
; int sse_multiplicar_matrices_4x4(float matriz_fuente_1[4][4],
;                                float matriz_fuente_2[4][4],
;                                float matriz_destino[4][4]);
;
; donde
;
; matriz_fuente_1    apunta a la primera matriz a multiplicar.
;
; matriz_fuente_2    apunta a la segunda matriz a multiplicar.
;
; matriz_destino      apunta a la matriz donde hay que almacenar el producto.
;
; La función retorna 1 si pudo realizar el cálculo y 0 si alguno de los argumentos es nulo.

global    sse_multiplicar_matrices_4x4

section   .text

sse_multiplicar_matrices_4x4:

    xor     eax,eax        ; RAX = 0 => si hay error, retornar 0.

    test    rdi,rdi
    jz      error          ; Salir con RAX = 0 si matriz_fuente_1 es nulo.
    test    rdi,15
    jnz     error          ; Salir con RAX = 0 si matriz_fuente_1 no está alineado.

    test    rsi,rsi
    jz      error          ; Salir con RAX = 0 si matriz_fuente_2 es nulo.
    test    rsi,15
    jnz     error          ; Salir con RAX = 0 si matriz_fuente_2 no está alineado.

    test    rdx,rdx
    jz      error          ; Salir con RAX = 0 si matriz_destino es nulo.
    test    rdx,15
    jnz     error          ; Salir con RAX = 0 si matriz_destino no está alineado.

    ; Cargar las filas de la matriz_fuente_1 en XMM1, XMM2, XMM3, XMM4.

    movaps  xmm1,[rdi]      ; xmm1 = (a11, a12, a13, a14)
    movaps  xmm2,[rdi+16]   ; xmm1 = (a21, a22, a23, a24)
    movaps  xmm3,[rdi+32]   ; xmm3 = (a31, a32, a33, a34)
    movaps  xmm4,[rdi+48]   ; xmm4 = (a41, a42, a43, a44)

    ; Cargar las filas de la matriz_fuente_2 en XMM5, XMM6, XMM7, XMM8.

    movaps  xmm5,[rsi]      ; xmm5 = (b11, b12, b13, b14)
    movaps  xmm6,[rsi+16]   ; xmm6 = (b21, b22, b23, b24)
    movaps  xmm7,[rsi+32]   ; xmm7 = (b31, b32, b33, b34)
    movaps  xmm8,[rsi+48]   ; xmm8 = (b41, b42, b43, b44)

    ; Trasponer la matriz_fuente_2. La matriz traspuesta queda en XMM5, XMM7, XMM9 y XMM11.

    movaps  xmm9,xmm5        ; xmm9 = (b11, b12, b13, b14)
    movaps  xmm11,xmm7       ; xmm11 = (b31, b32, b33, b34)

    unpcklps xmm5,xmm6        ; xmm5 = (b11, b21, b12, b22)
    unpckhps xmm9,xmm6        ; xmm9 = (b13, b23, b14, b24)
    unpcklps xmm7,xmm8        ; xmm7 = (b31, b41, b32, b42)
    unpckhps xmm11,xmm8       ; xmm11 = (b33, b43, b34, b44)

```

```

movaps    xmm6,xmm5      ; xmm6 = (b11, b21, b12, b22)
movaps    xmm10,xmm9     ; xmm10 = (b13, b23, b14, b24)

movhlps   xmm5,xmm7      ; xmm5 = (b11, b21, b31, b41)
movhlps   xmm7,xmm6      ; xmm7 = (b12, b22, b32, b42)
movhlps   xmm9,xmm11     ; xmm9 = (b13, b23, b33, b43)
movhlps   xmm11,xmm10    ; xmm11 = (b14, b24, b34, b44)

; Copiar la matriz_fuente_2 traspuesta en XMM6, XMM8, XMM10 y XMM12.

movaps    xmm6,xmm5
movaps    xmm8,xmm7
movaps    xmm10,xmm9
movaps    xmm12,xmm11

; Multiplicar la fila 1 de matriz_fuente_1 por las columnas de matriz_fuente_2 para obtener
; la fila 1 de la matriz producto.

mulps     xmm6,xmm1
mulps     xmm8,xmm1
mulps     xmm10,xmm1
mulps     xmm12,xmm1

haddps    xmm6,xmm8
haddps    xmm10,xmm12
haddps    xmm6,xmm10

movaps     [rdx],xmm6 ; Guardar en memoria la fila 1 de la matriz producto.

; Copiar la matriz_fuente_2 traspuesta en XMM6, XMM8, XMM10 y XMM12.

movaps    xmm6,xmm5
movaps    xmm8,xmm7
movaps    xmm10,xmm9
movaps    xmm12,xmm11

; Multiplicar la fila 2 de matriz_fuente_1 por las columnas de matriz_fuente_2 para obtener
; la fila 2 de la matriz producto.

mulps     xmm6,xmm2
mulps     xmm8,xmm2
mulps     xmm10,xmm2
mulps     xmm12,xmm2

haddps    xmm6,xmm8
haddps    xmm10,xmm12
haddps    xmm6,xmm10

movaps     [rdx+16],xmm6 ; Guardar en memoria la fila 2 de la matriz producto.

; Copiar la matriz_fuente_2 traspuesta en XMM6, XMM8, XMM10 y XMM12.

movaps    xmm6,xmm5
movaps    xmm8,xmm7
movaps    xmm10,xmm9
movaps    xmm12,xmm11

; Multiplicar la fila 3 de matriz_fuente_1 por las columnas de matriz_fuente_2 para obtener
; la fila 3 de la matriz producto.

```

```

mulps    xmm6,xmm3
mulps    xmm8,xmm3
mulps    xmm10,xmm3
mulps    xmm12,xmm3

haddps   xmm6,xmm8
haddps   xmm10,xmm12
haddps   xmm6,xmm10

movaps    [rdx+32],xmm6    ; Guardar en memoria la fila 3 de la matriz producto.

; Multiplicar la fila 4 de matriz_fuente_1 por las columnas de matriz_fuente_2 para obtener
; la fila 4 de la matriz producto.

mulps    xmm5,xmm4
mulps    xmm7,xmm4
mulps    xmm9,xmm4
mulps    xmm11,xmm4

haddps   xmm5,xmm7
haddps   xmm9,xmm11
haddps   xmm5,xmm9

movaps    [rdx+48],xmm5    ; Guardar en memoria la fila 4 de la matriz producto.

mov     eax, 1             ; Salir indicando éxito.

error: ret

```

6.4. Aumentar el brillo de una imagen de grises

Listado 6.4: sse_aumentar_brillo.S

```

; Fichero: sse_aumentar_brillo.S
;
; Escribe una función que permita incrementar el brillo de una imagen de escala de grises. Cada pixel
; está representado con un byte cuyo valor entre 0 y 255 representa el nivel de gris, entre negro y
; blanco, del mismo. Para incrementar el brillo de la imagen se suma un valor fijo a los bytes que
; representan los pixels de la imagen.
;
; int sse_aumentar_brillo(unsigned char *ptr_imagen,
;                          int ancho,
;                          int alto,
;                          unsigned char incremento_brillo);
;
; donde
;
; ptr_imagen    es un puntero a la imagen. La imagen debe estar alineada en una dirección
;               divisible entre 16.
;
; ancho         es la anchura de la imagen en pixels.
;
; alto         es la altura de la imagen en pixels.
;
; incremento_brillo es la cantidad en la que hay que aumentar el brillo de cada pixel de la imagen.
;
; La función retorna 1 si puede realizar la operación de aumento de brillo correctamente y 0 si alguno
; de los argumentos no es correcto.

global     sse_aumentar_brillo

```

```

section      .text

sse_aumentar_brillo:

    xor      eax,eax      ; Salir con RAX = 0 si error.

    test     rdi,rdi
    jz       error        ; Si ptr_imagen es nulo, salir indicando error.

    test     rdi,15
    jnz      error        ; Si ptr_imagen no está alineado, salir indicando error.

    cmp      esi,0
    jle      error        ; Si ancho <= 0, salir indicando error.

    cmp      edx,0
    jle      error        ; Si alto <= 0, salir indicando error.

    imul     edx,esi      ; EDX = ancho*alto = número de pixels.
    mov      eax,edx      ; Copiar en EAX.

    ; Mediante las instrucciones SSE podemos procesar la imagen en bloques de 16 pixels
    ; (ya que la imagen es de escala de grises con 1 byte por pixel).
    ; Dividimos el número de pixels de la imagen entre 16 para ver cuantos bloques completos
    ; de 16 pixels hay. Si el resultado es 0, hay menos de 16 pixels.

    shr      eax,4
    jz       menos_de_16_pixels

    ; Si hay al menos un bloque completo de 16 pixels, procesarlo con instrucciones SSE.

    ; Primero, hay que conseguir que el aumento de brillo quede repetido 16 veces en los
    ; 16 bytes de un registro XMM.
    ; En los siguientes cinco comentarios cada dígito representa un byte. a representa el byte
    ; de aumento de brillo.

    movd     xmm0,ecx      ; XMM0 = 000000000000000a
    punpcklbw xmm0,xmm0    ; XMM0 = 000000000000000aa
    punpcklwd xmm0,xmm0    ; XMM0 = 0000000000000aaaa
    punpckldq xmm0,xmm0    ; XMM0 = 00000000aaaaaaaa
    punpcklq dq xmm0,xmm0  ; XMM0 = aaaaaaaaaaaaaa

    ; Una manera alternativa más corta
    ;
    ; movd     xmm0,ecx
    ; pxor     xmm1,xmm1
    ; pshufb   xmm0,xmm1

bucle:  movdqa  xmm1,[rdi]   ; Mover a XMM1 16 pixels de la imagen.
        paddusb xmm1,xmm0   ; Aumentar con saturación el brillo de los 16.
        movdqa  [rdi],xmm1  ; Devolver el resultado a la imagen.
        add     rdi,16      ; Avanzar el puntero a los siguientes 16.
        dec     eax         ; Decrementar contador de bloques de 16.
        jnz     bucle       ; Si no es 0, repetir.

menos_de_16_pixels:

    ; Si después de procesar bloques de 16 pixels queda algún pixel adicional
    ; procesarlos uno a uno.
    ;
    ; EDX aún tiene el número de pixels de la imagen.

```

```

; CL aún tiene el aumento de brillo.

and     edx,0b1111 ; EDX = resto de núm. pixels entre 16.
jz      salida     ; Si no queda ningún pixel, salir.

mov     esi,255     ; Valor de saturación.
bucle2: mov     al,[rdi] ; Leer un pixel de la imagen.
add     al,cl       ; Sumar el aumento de brillo.
cmovc   eax,esi     ; Si hay acarreo, saturar a 255.
mov     [rdi],al    ; Devolver el resultado a la imagen.
inc     rdi         ; Incrementar puntero.
dec     edx         ; Decrementar contador de pixels.
jnz     bucle2      ; Si no es 0, repetir.

salida: mov     eax,1 ; Salir indicando éxito.

error:  ret

```

6.5. Fundir dos imágenes RGB

Queremos escribir una función en ensamblador de 64 bits que use instrucciones SSE para obtener una imagen a partir del fundido de otras dos. Usaremos imágenes en color RGB de 24 bits por pixel en la que cada pixel está representado por tres bytes cuyos valores indican las intensidades de luz roja, verde y azul que contribuyen a su color. Los valores de las componentes roja, verde y azul de cada pixel de la imagen fundida se obtienen a partir de las correspondientes componentes de los pixels que ocupan la misma posición en las imágenes originales. Si llamamos (r_f, g_f, b_f) a las componentes roja, verde y azul de un pixel de la imagen fundida, (r_1, g_1, b_1) a las componentes del pixel que ocupa la misma posición en la imagen original 1 y (r_2, g_2, b_2) a las componentes del pixel que ocupa la misma posición en la imagen original 2, la forma de obtener las primeras a partir de las últimas es

$$\begin{aligned}
 r_f &= P \cdot r_1 + (1 - P) \cdot r_2 \\
 g_f &= P \cdot g_1 + (1 - P) \cdot g_2 \\
 b_f &= P \cdot b_1 + (1 - P) \cdot b_2
 \end{aligned}$$

donde P es un valor real entre 0 y 1 que indica la proporción relativa de la primera imagen en la imagen resultante. Por ejemplo, si $P = 0.1$, la imagen fundida estará compuesta por un 10% de la imagen 1 y un 90% de la imagen 2.

Como la operación que hay que realizar para obtener cada componente de cada pixel de la imagen fundida sólo depende de las componentes de mismo color de los pixels correspondientes de las imágenes 1 y 2, podemos expresar la operación simplemente como

$$c_f = P \cdot c_1 + (1 - P) \cdot c_2 \quad P \in [0, 1], \quad P \in \mathbb{R}$$

sin importar de qué componente se trate, roja, verde o azul. El sistema funciona igualmente con una imagen en escala de grises, donde cada pixel está representado por una única componente.

Para que todos los cálculos se puedan realizar con números enteros, volveremos a expresar la operación en función de otro parámetro, K , que podrá tomar valores enteros en el intervalo $[0, 256]$. En función de K , la operación quedará

$$c_f = \frac{K \cdot c_1 + (256 - K) \cdot c_2}{256} \quad K \in [0, 256], \quad K \in \mathbb{Z}$$

Las operaciones de división entre 256 pueden hacerse mediante desplazamientos a la derecha de 8 bits.

Los distintos valores de K permiten 257 niveles diferentes de proporción relativa de una y otra imagen en la imagen final.



Figura 2: Imágenes originales (a, b) e imagen fundida (c) para valor de $K = 120$.

Listado 6.5: sse_fundir_imagenes_rgb.S

```
; Fichero: sse_fundir_imagenes_rgb.S
;
; Escribe una función que permita realizar el fundido de dos imágenes RGB. El prototipo de la función
; es
;
;   int sse_fundir_imagenes_rgb(unsigned char *ptr_imagen_1,
;                               unsigned char *ptr_imagen_2,
;                               int ancho,
;                               int alto,
;                               unsigned int nivel_fundido);
;
; donde
;
;   ptr_imagen_1   apunta a la primera imagen a fundir. La imagen fundida queda almacenada
;                   sustituyendo a esta imagen, por tanto la primera imagen original se sobrescribe
;                   con la imagen fundida. La imagen debe estar alineada en una posición divisible
;                   entre 16.
;
;   ptr_imagen_2   apunta a la segunda imagen a fundir. La imagen debe estar alineada en una posición
;                   divisible entre 16.
;
;   ancho          ancho de las imágenes a fundir en pixels.
;
;   alto           alto de las imágenes a fundir en pixels.
;
;   nivel_fundido   valor entre 0 y 256 que indica la proporción relativa con la que las imágenes
;                   originales aparecen en la imagen final. Si nivel_fundido es 256 la imagen final
;                   será igual a la imagen 1. Si nivel_fundido es 0 la imagen final será igual a la
;                   imagen 2.
;
; La función retorna 1 si trabajó correctamente y 0 si alguno de los argumentos es erróneo.

global    sse_fundir_imagenes_rgb

section   .text

sse_fundir_imagenes_rgb:

    xor    eax,eax    ; Si hay error, retornar con RAX = 0.
```


[illegible]

```

pxor      xmm2,xmm2    ; XMM0 = 00000000000000000000000000000000

; En cada iteración del siguiente bucle se procesan 16 bytes de la imagen 1 y 16 bytes
; de la imagen 2 para dar lugar a 16 bytes de la imagen fundida que sustituyen a los
; 16 bytes de la imagen 1.

bucle:    movdqa      xmm3,[rdi]  ; XMM3 = 16 bytes de la imagen 1.
          movdqa      xmm4,xmm3   ; Copiar en XMM4.
          punpcklbw    xmm3,xmm2   ; Entrelazar los 8 primeros bytes con bytes a 0.
          punpckhbw    xmm4,xmm2   ; Entrelazar los otros 8 bytes con bytes a 0.

          ; Con las dos últimas instrucciones, 16 bytes de la imagen 1 han sido convertidas a 16 words
          ; que han quedado en XMM3 y XMM4.

          movdqa      xmm5,[rsi]  ; XMM5 = 16 bytes de la imagen 2.
          movdqa      xmm6,xmm5   ; Copiar en XMM6.
          punpcklbw    xmm5,xmm2   ; Entrelazar los 8 primeros bytes con bytes a 0.
          punpckhbw    xmm6,xmm2   ; Entrelazar los otros 8 bytes con bytes a 0.

          ; Con las dos últimas instrucciones, 16 bytes de la imagen 2 han sido convertidas a 16 words
          ; que han quedado en XMM5 y XMM6.

          ; Ahora se hacen las operaciones K*byte_1 con 16 words, primero las 8 en XMM3 y luego las 8 en
          ; XMM4. Cada operación de multiplicación de words genera en principio un producto de 32 bits
          ; pero la instrucción PMULLW sólo se queda con los 16 bajos. Estos 16 bits bajos tienen el
          ; producto de 8 bits por 8 bits que realmente nos interesa.

          pmullw      xmm3,xmm0
          pmullw      xmm4,xmm0

          ; Ahora se hacen las operaciones (256 -K)*byte_2

          pmullw      xmm5,xmm1
          pmullw      xmm6,xmm1

          ; Ahora se hacen las operaciones K*byte_1 + (256 - K)*byte_2.

          paddusw     xmm3,xmm5
          paddusw     xmm4,xmm6

          ; Ahora se hace la división entre 256 desplazando cada palabra 8 bits hacia la derecha.

          psrlw      xmm3,xmm8
          psrlw      xmm4,xmm8

          ; Los bytes situados en la parte baja de cada palabra de XMM3 y XMM4 se empaquetan en XMM3
          ; obteniendo los 16 bytes de la imagen fundida.

          packuswb    xmm3,xmm4

          movdqa      [rdi],xmm3   ; Se guardan 16 bytes de la imagen fundida.
                                   ; sustituyendo a 16 bytes de la imagen 1.

          add         rdi,16        ; Avanzar a los siguientes 16 bytes de la imagen 1.
          add         rsi,16        ; Avanzar a los siguientes 16 bytes de la imagen 2.

          dec         edx           ; Decrementar contador de bloques de 16 bytes.
          jnz         bucle         ; Si no llegó a 0, repetir.

          ; Ahora se procesan los bytes que quedaron después de procesar los bloques de 16.

```

```
menos_de_16_pixels:
    and     r9d,15      ; R9D = resto de dividir numero de bytes en la imagen entre 16.
    jz      no_mas_pixels ; Si el resto es 0 no quedan bytes que procesar.

bucle2: mov     al,[rdi] ; Tomar byte de la imagen 1.
        mul     r8b      ; Multiplicar por K.
        mov     r10w,ax   ; Guardar resultado en R10W.
        mov     al,[rsi] ; Tomar byte de la imagen 2.
        mul     cl        ; Multiplicar por (256 - K).
        add     ax,r10w   ; Sumar K*byte_1 + (256 - K)*byte_2.
        mov     [rdi],ah  ; Almacenar los 8 bits altos del resultado.
                        ; Tomar los 8 bits altos equivale a desplazar 8 bits hacia la derecha.
        inc     rdi       ; Pasar al siguiente byte de la imagen 1.
        inc     rsi       ; Pasar al siguiente byte de la imagen 2.

        dec     r9d       ; Decrementar número de bytes restantes.
        jnz     bucle2    ; Si no llegó a 0, repetir.

no_mas_pixels:
    mov     eax,1        ; Salir indicando éxito.

error:  ret
```