Apunte Orga 2

June 2019

Este apunte esta hecho en base al primer cuatrimestre de 2019. Puede haber varios errores y simplificaciones. Úselo bajo su propio riesgo.
La materia esta dividida en 2 partes, programacion usando Assembler y programacion de OS.

Temas

Part I
Programacion usando Assembler

1 Funcionamiento basico y Esqueleto de un codigo en ASM

Queremos hacer un codigo en asm. Para esto tenemos que hablar de registros y nombres impuestos por Intel.

1.1 Registros y nombres

Primero, que es un registro? La CPU tiene unidades que guardan una pequeña cantidad de informacion con las que puede trabajar. El operar con estas es mucho mas rapido que operar con memoria. Pedir un dato a la memoria es unidades de tiempo mas lento (el impacto tambien dependera si se realiza *cache miss* pero en este momento eso no es lo relevante).
Los registros principales se muestran en la figura 1.

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

Figure 1: Registros Generales. Notar que, por ejemplo, eax, ax, etc no son registros distintos sino que son parte de los bits de rax.

Solo hay uno de los que se encuentran en la tabla que no es realmtente general. El **rsp**. Este se usa para indicar el tope del stack.
Cuando uno hace un `push`, el valor de **rsp** disminuye en el tamaño de lo que se pushea en bytes y se escribe en memoria a partir de esta posicion.
Ejemplo: pushear un numero x de 4 bytes (como son los `int`) hace

$$\textbf{rsp} \leftarrow \textbf{rsp} - 4$$
$$[\textbf{rsp}] \leftarrow x$$

De manera que cuando se pushea, **rsp** apunta a lo que se acaba de pushear. Habiendo dicho esto, se puede realizar con el **rsp** cualquiera de las instrucciones que se realizan con el resto, aunque hay que tener cuidado ya que esta conectado con el stack.
Algunos casos en los que uno quiere jugar con el **rsp** son: Alinear la pila (en cuyo caso restamos para alinear), Terminamos de usar una parte del stack y no nos interesa recuperarlo (restamos el espacio en bytes)
Ademas de estos registros, la CPU tiene muchos otros, aunque suelen cumplir propositos especificos. El registro **rip** es el *instruction pointer*, contiene la direccion de memoria donde comienza la proxima instruccion a ejecutar. Una vez que se sabe la instruccion a ejecutar actualmente, **rip** aumenta (solo) de manera que apunte a la instruccion inmediatamente despues a esta. Mas precisamente, luego de que se lee la instruccion y los operandos a utilizar, el **rip** aumenta en base a esto. Notar que al ejecutar la instruccion actual tambien puede modificarse el **rip**, como es en el caso de la instruccion `jmp`. Como se modifico el **rip**, la instruccion que se ejecutara despues puede ser otra que la siguiente en memoria, esto es el "salto".
Otro registro de la CPU es el registro de los flags. Para una arquitectura de 64-bits, este es un registro de 64 bits llamado **rflags** y para uno de 32-bits es de 32 y se llama **eflags**. Para el sistema de 64-bits, los 32 bits mas significativos estan reservados por lo que este es basicamente identico a **eflags**. Este es el registro donde se encuentran los resultados de las distintas operaciones como el *carry flag* (aca se encuentra la lista). Entre otras funciones, estos son los que se usan al momento de los saltos condicionales. Algunas instrucciones pueden modificar los flags.

En 1999 se introdujeron los registros `xmm` (de 128 bits). En 2011, con *AVX*, los `ymm` (de 256 bits). En la materia se usan exclusivamente los `xmm`, principalmente para SIMD.

1.2 Sections

Ahora, un programa escrito en asm tiene varias secciones.

- `section.text`. Donde se escribe el codigo
- `section.data`. Donde se escribe datos que pueden ser leidos y escritos.
- `section.rodata`. Donde se escribe datos que solo pueden ser leidos.

Puesto en palabras simples, se escribira distintas cosas en cada parte. No hay un orden especial para escribirlos en el archivo de texto ni se esta obligado a agregar todos estos. Además, existen otras secciones que se pueden declarar aunque no se usan en la materia, como `section.bss`.

-¿Si uno puede leer la parte de codigo, no puedo poner mis datos ahi tambien?
-Depende, se puede pero CUIDADO, uno no puede escribir en `section.text`, es *read-only*. Si estos datos se van a modificar en ejecucion, debe ir en `section.data`.

Un ejemplo que imprime Hola Mundo en la consola. No es necesario entender como funciona en detalle.

```
section .data
   msg:  db 'Hola Mundo', 10
   largo equ $ - msg
global start
section .text
   start:
   mov rax, 4      ; funcion 4
   mov rbx, 1      ; stdout
   mov rcx, msg    ; mensaje
   mov rdx, largo  ; longitud
   int 0x80
   mov rax, 1
   mov rbx, 0
   int 0x80
```

2 Programas simples en Assembler

Hay un par de instrucciones que componen la parte basica de basica de programar en Assembler. `add, jmp, cmp...` son simples y faciles de entender por lo que no se desperdicia tiempo en ellas

2.1 Pseudo-Instrucciones y similares

Además de instrucciones para ejecutar, existen otras cosas utiles que se pueden escribir usando nasm (ver aca)

	Descripcion	Ejemplos

db	Coloca en la posicion de memoria lo indicado como byte. Puede tomar tanto enteros como caracteres (los escribe en ASCII). Si se le pasa una string se usa esta posicion de memoria y las siguientes	`db 5`
		`db 5,"a", 0xf`
		`db "a","b"`
		`db "ab"`
		`db "ab", 10`

dw/dd/dq	Coloca en la posicion de memoria lo indicado como 2 bytes, 4 bytes y 8 bytes respectivamente. Toman enteros y `dd` y `dq` pueden tomar tambien punto flotante correspondiente a `float` y `double`*	`dw 5`
		`dd 5`
		`dd 5.0`
		`dd 5, 5.0`
		`dq 5.0`

resb/resw/resd/resq	Reserva la cantidad indicada la unidad correspondiente	`resb 8`
		`resw 4`
		`resd 2`
		`resq 1`

$	Direccion de memoria de la linea en la que se escribe este	`jmp $` (esta cuelga el programa en el lugar)
times	Repite la proxima instruccion la cantidad de veces indicada. Puede pensarse como un *copy-paste* n veces de la instruccion, uno abajo del otro	`times 100 db 0` (genera 100 bytes de 0)

%define	Hacer defines. Cuando aparezca el texto se cambia por el significado	`%define OFFSET_ARREGLO 125`
		`%define SELECTOR_CODIGO 0X45`
		`%define LETRA_A "a"`
		`%define registro_rdi rdi`

equ	Similar a %define. Este sin embargo evalua significados. Es decir, puede usarse con operaciones	`equ PERSONAS_EXTRA 125+7` (se define a 132)

* En realidad uno puede utilizar punto flotante con `db` y `dw` con sus respectivas precisiones pero en la materia no se usa.

2.2 System Calls

Hay ciertas operaciones que no puede realizar nuestro programa ya que no tiene los privilegios necesarios para hacerlo. Un ejemplo de eso es imprimir texto en la terminal. Para hacerlo entonces, lo que realizamos es un *syscall*. Basicamente, llamamos al sistema operativo para que realize esta accion.
Que significa esto en la practica? Se pone en los registros la informacion relevante y se llama a una interrupcion para la accion a realizar. (`int numero_interrupcion`).
La realidad es que en la primer parte de la materia no se usan mayormente las *syscall* para realizar cosas sino que se llaman a funciones de C que se encarga de hacer las syscalls necesarias. (Ejemplo de esto es hacer `printf`)
En la segunda parte de la materia se ve un poco la contra cara de esto y se ve las *syscall* desde la perspectiva del sistema. No se onda en detalles.
Veamos ahora ejemplos tomando el hola mundo.

```
1    mov rax, 4      ; funcion 4
2    mov rbx, 1      ; stdout
3    mov rcx, msg    ; mensaje
4    mov rdx, largo  ; longitud
5    int 0x80
```

Escribir texto en la terminal de Linux usando *syscall*

En `rax` va 4, `rbx` va 1, en `rcx` va puntero al comienzo de la string, `rdx` va en largo en bytes de nuestro texto

```
1    mov rax, 1
2    mov rbx, 0
3    int 0x80
```

Indicar la finalizacion del programa (exit) usando *syscall*

Esto le indica a Linux que la ejecucion del programa termino correctamente. Notar que el programa en Assembler correra hasta que se realize una operacion invalida o el mismo programa indique su finalizacion. Nuevamente, esto se ve desde la persepectiva del sistema operativo en la segunda parte de la materia.
Una ultima nota sobre las *syscalls*. Son especificas al sistema operativo. Es decir, la forma en la que se imprimiese texto en la terminal dependera del sistema operativo. Quizá nuestro sistema use otra interrupcion u otro estado de los registros.

2.3 Macros

Hacer macros en asm (nasm mas concretamente) es simple.

```
1 %macro mi_macro <numero_de_parametros>
2 ; ...
3 %endmacro
```

Se accede a cada parametro con su indice i de la forma %i
Hagamos un ejemplo, creando una macro que mueva el contenido de un lugar a otro.

```
1 %macro macro_mover 2
2     mov %1, %2
3 %endmacro
```

De esta forma, al escribir `macro_mover rax 75` (no hay comas) se convierte en `mov rax, 75`. Notar que si se invoca de la forma `macro_mover 75 rax` habra un error ya que `mov 75, rax` no es ninguna instruccion valida.
Si creamos una macro que utiliza etiquetas hay que tener cierto cuidado. Si se invoca la macro mas de una vez significara que habra multiples apariciones de la misma etiqueta. Para arreglar esto existe `%%`. Este sera remplazado por un identificador distinto para cada invocacion a la macro.

```
%macro  retz 0
    jnz %%skip
    ret
    %%skip:
%endmacro
```

Hay muchas otras cosas que pueden realizarse con el preprocesador. *Conditionals* y *loops* por ejemplo. Mas informacion sobre otras cosas que se pueden hacer aca

3 Convencion C. (Combinar ASM y C)

Supongamos que estamos escribiendo codigo en asm y queremos llamar a una funcion realizada en C que toma ciertos parametros. Como hacer esto?

Desde asm llamamos rutinas con `call`, por lo que hacer algo como `call funcion` permitiria empezar a ejecutar desde el comienzo de la funcion pero sigue estando el problema de como sabe la funcion de C los parametros. Deberia de alguna forma saber donde buscar los parametros que necesita.
Podria ser que cada funcion tenga su propio set de reglas para donde quiere que se encuentren los parametros, entonces el llamador a esta funcion deberia "saber" este al momento de necesitar cierta funcion. Colocaria todos los parametros en su sitio y luego ejecuta `call funcion`. Podriamos simplificar cosas y establecer una convencion comun y que todas nuestras funciones la respeten. Esto es lo que hace C.

Convencion C 64 bits

- Los parametros se pasan de izquierda a derecha usando `rdi, rsi, rdx, rcx, r8 y r9`. Si no alcanzan, el resto se va pusheando en la pila de izquierda a derecha. Para numeros de punto flotante se utiliza los registros `xmm`. (son registros de 128 bits).
- Algunos registros no cambian antes y despues de llamar a una funcion. `rbx, r12, r13, r14, r15`
- El `rbp` es utilizado como la base de la pila actual. Tambien debe preservarse.
- Alineacion de la pila al comienzo de funcion a 16 bytes
- Retorna el resultado en `rax` (y `rdx` si ocupa 128 bits) o `xmm0` (si es un numero de punto flotante)

Convencion C 32 bits

- Los parametros se pasan usando la pila. Pusheandose de derecha a izquierda.
- Algunos registros no cambian antes y despues de llamar a una funcion. `ebx, esi, edi`
- Alineacion de la pila al comienzo de funcion a 4 bytes
- Retorna el resultado en `eax` (y `edx` si ocupa 64 bits)

Cuando escribimos un programa en C, el compilador se encarga de hacer valer la comvencion. Por eso, si se quiere mezclar asm y C, necesitamos nosotros en asm respetar la convencion "manualmente" cuando sea nesesario.

Un exelente recurso para entender esto es godbolt. Ahi se puede comparar el programa en C con su compilado en asm y ver como se pone en practica las convenciones.

Veamos ejemplos de como quedan los parametros para ambas convenciones.

Para 64 bits usemos `int g64(int x, float y1, int* puntero, double y2)`
Para la funcion `g64` queda, `x` en `edi` (parte baja de `rdi`), `y1` en `xmm0`, `puntero` en `rsi` y `y2` en `xmm1`. La funcion devolvera el resultado en `eax` (parte baja de `rax`).

Para 32 bits usemos `int h32(int x, int* puntero, float y)`
Para la funcion `h32` queda, la direccion de retorno en el tope del stack. (Ya que al realizar un `call` se pushea la dreccion) luego `x`, `puntero` y `y` en ese orden. Esto conlleva a que al llamar a funciones que respeten la convencion C 32 bits desde asm debemos pushear los parametros de derecha a izquierda.

Otra utilidad para ver la convencion C en funcionamiento y analizar lo que hace el compilador es el desensamblado (convertir binario en assembler). Para esto, dado un binario `programa`, se llama en la terminal.

```
objdump -d programa
```

3.1 Usar estructuras de datos

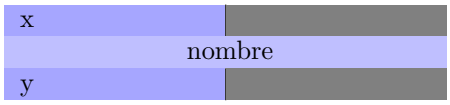
Supongamos ahora que creamos la siguiente estructura de datos en C con las respectivas definiciones de funciones.

```
1 typedef struct str_barco {
2     int  x;
3     char* nombre;
4     int  y;
5 } barco;
6
7 int x_barco(barco* b);
8 char* nombre_barco(barco* b);
9 int y_barco(barco* b);
```

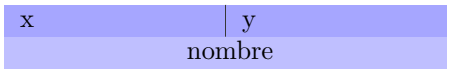
Ahora queremos implementar estas funciones en asm. Una estructura no es mas que un dato puesto en memoria uno luego de otro. Notar que esto significa que el orden en que escribamos los campos

de la estructura importa. No es lo mismo escribir los campos en el orden `int, char*, int` que escribirlos como `int, int, char*`
En C, los campos de la estructura estan alineados a sus tamaños y las estructura en si esta alineada a su campo de mayor tamaño (a menos que diga explicitamente la propiedad `packed`).
Veamos como estan colocados en memoria y como cambia dependiendo de como escribamos la estructura
Un `int` ocupa 4 bytes. Un puntero ocupa 8 bytes (ya que es una direccion de memoria de 64 bits).

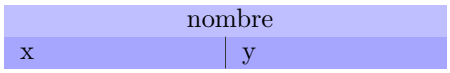
```c
typedef struct str_barco {
    int  x;
    char* nombre;
    int  y;
} barco;
```




```c
typedef struct str_barco {
    int  x;
    int  y;
    char* nombre;
} barco;
```




```c
typedef struct str_barco {
    char* nombre;
    int  x;
    int  y;
} barco;
```



Escribamos el codigo para las funciones previas en asm, ateniendonos al orden original para la estructura.

```asm
global x_barco
global nombre_barco
global y_barco


%define OFFSET_X 0
%define OFFSET_NOMBRE 8
%define OFFSET_Y 16

    x_barco:
    ; nuestro parametro es un puntero, se encuentra en rdi. Por lo que debemos
     calcular manualmente donde se encuentra el campo que queremos
    mov eax, [rdi + OFFSET_X]
    ret

    nombre_barco:
    mov rax, [rdi + OFFSET_NOMBRE]
    ret

    y_barco:
    mov eax, [rdi + OFFSET_Y]
    ret
```

3.2 Nota sobre char, strings y punteros

En C las strings son simplemente arreglos de `char`. La finalizacion de un string se indica con un 0. Si se quiere manipular o generar strings en asm debe tenerse en cuenta. Ej: `"ahora"` se escribe en asm como `db "ahora", 0`
La razon de que en la estructura de `barco` se uso `char*` para `nombre` es que si queremos que el nombre este en la estructura en lugar de un puntero a ella, se necesitaria "generar el espacio en la estructura". Por ejemplo, definiendo `char nombre[50]`, eso asignaría un espacio de 50 bytes (uno por cada caracter) para el nombre. Para que el tamaño de la string sea variable, usamos un puntero al comienzo del arreglo de caracteres.

3.3 Hola mundo usando printf

Queremos implementar la funcion `void imprimir_hola()` a ser usada desde C que imprima `"Hola mundo"` en la terminal
Ademas, hagamos una funcion `void imprimir_teorema (int x)`. Esta imprimira `"El numero <x> no es igual a <x+1>`

```
section .rodata
formato_hola_msg db "%s"
hola_msg: db "Hola mundo", 10 , 0 ; 10 es el caracter de nueva linea

section .text
extern printf


;printf(formato_hola_msg, hola_msg)

imprimir_hola:
    ; como es llamada desde C, se esta alineado a 16 bytes. Si llamamos a printf de
    esta forma, entraria desalineado (ya que el call pushea la direccion de retorno,
    8 bytes)

    ;por esto movemos manualmente el stack para que entre alineado
    sub rsp, 8

    mov rdi, formato_hola_msg ; char*
    mov rsi, hola_msg ; char *
    call printf

    add rsp, 8
    ret

imprimir_teorema:
    sub rsp, 8

    mov rsi, rdi ; int
    mov rdx, rdi
    inc rdx ; int
    mov rdi, formato_teorema_msg  ; char*
    call printf

    add rsp, 8
    ret

    ;como esto es solo lectura, tambien puede ponerse en la seccion de codigo
    ;siempre y cuando no se ejecute
```

```
38      formato_teorema_msg: db "El numero %d no es igual a %d", 10 ,0
```

4 Ejercicios de Estructura

4.1 Cifrado cesar

Queremos escribir en asm la funcion `void cifrar_cesar(char* mensaje, int n)`.
Esta funcion toma un puntero a una string y hace lo siguiente: por cada caracter, si es una letra, la remplaza por el que se encuentra n lugares mas adelante en el alfabeto.
Por ejemplo: si `n = 1` todas las $a \to b$, todas las $b \to c$ etc. Es ciclico por lo que todas las $z \to a$.
Los caracteres que no sean letras como ! o # no son modificados.
Escribamos primero el codigo en C.

```c
#define INICIO_LETRAS 97
#define TAM_ALFABETO 26

void cifrar_cesar(char* mensaje, int n) {
    int i = 0;

    /*mientras no sea el caracter de finalizacion del mensaje*/
    while (mensaje[i] != 0) {
        char letra = mensaje[i];
        /*si el caracter es una letra*/
        if (INICIO_LETRAS <= letra && letra < INICIO_LETRAS + TAM_ALFABETO) {
            /*realizamos los calculos*/
            int offset = (int) letra - INICIO_LETRAS;
            offset = (offset + n) % TAM_ALFABETO;
            letra = (char) INICIO_LETRAS + offset;
            /*reescribimos la letra*/
            mensaje[i] = letra;
        }
        ++i;
    }
```

```asm
global cifrar_cesar
%define INICIO_LETRAS 97
%define TAM_ALFABETO 26

    cifrar_cesar:
    ; rdi contriene el puntero al mensaje y esi contiene n

    .ciclo:
        mov al, [rdi] ; leemos la letra
        cmp al, 0 ; es el caracter de finalizacion?
        je .fin

        ; es letra?
        cmp al, INICIO_LETRAS
        jb .seguir
        mov bl, INICIO_LETRAS
        add bl, TAM_ALFABETO
        cmp al, bl
        jae .seguir
```

```
20
21          ;limpiamos la parte alta del registro. de esta manera "casteamos" el char a
        int
22          shl eax, 31
23          shr eax, 31
24
25
26          ; eax = offset + n
27          sub eax, INICIO_LETRAS
28          add eax, esi
29
30
31          ; eax = eax % TAM_ALFABETO = nuevo offset
32          xor rdx, rdx ; seteamos rdx en 0 para que no afecte la division
33          mov ecx, TAM_ALFABETO
34          idiv ecx
35          mov eax, edx
36
37          ; eax = eax + INICIO_LETRAS = nueva letra (como int)
38          add eax, INICIO_LETRAS
39
40          ;reescribimos en memoria
41          mov [rdi], al
42
43          .seguir:
44          ;avanzamos el puntero 1 byte
45          inc rdi
46          jmp .ciclo
47
48      .fin:
49      ret
```

4.2 Ejercicio parcial 10/5/18. Arboles binarios

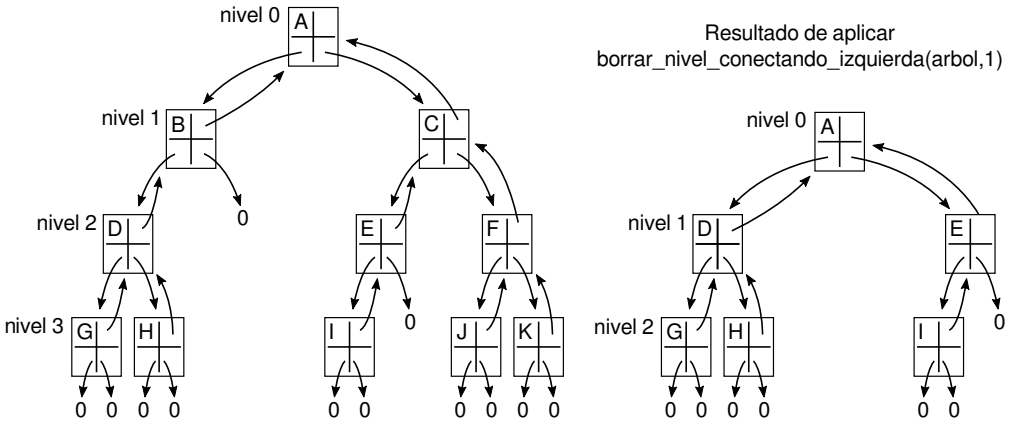
Ej. 1. (40 puntos)

Sea un árbol binario doblemente enlazado que respeta la siguiente estructura:

```
struct nodo {
            struct nodo* derecho,
            struct nodo* izquierdo,
            struct nodo* padre,
            void* data,
            void (*borrar)(void*)
            }
```

(15p) a. Programar en ASM la función `ContarPorNivel` que dado un puntero a nodo y un número de nivel, cuenta la cantidad de nodos que hay en el nivel indicado. El resultado es retornado en un puntero a entero denominado `cantidad`. Su aridad es: `void contar_por_nivel(struct nodo* arbol, unsigned int nivel, unsigned int* cantidad)`.

(25p) b. Programar en ASM la función `borrar_nivel_conectando_izquierda` que dado un doble puntero a nodo y un número de nivel, borra todos los nodos del nivel indicado, conectando al padre solamente los hijos izquierdos. El subarbol derecho debe ser eliminado. Su aridad es: `void borrar_nivel_conectando_izquierda(struct nodo** arbol, unsigned int nivel)`. Considerar que el puntero al primer nodo puede cambiar y debe ser retornado en el parámetro `arbol`.



Nota: Para borrar `data` se debe llamar a la función almacenada en el nodo en `borrar`. La misma tienen la misma aridad que la función `free`.

a) Implementacion en C.

```c
void contar_por_nivel(struct nodo* arbol, unsigned int nivel, unsigned int* cantidad) {
    /* setea el acumulador en 0. Por si este no lo estaba antes*/
    *cantidad = 0;
    contar_por_nivel_aux(arbol,nivel,cantidad);
}

void contar_por_nivel_aux(struct nodo* arbol, unsigned int nivel, unsigned int* cantidad) {

    if (arbol != NULL) {
        /*si este es el nivel buscado */
        if (nivel == 0) {
            /* se ira incrementando como acumulador en el caso recursivo*/
            *cantidad += 1;
        }

        else {
            contar_por_nivel_aux(arbol->izquierdo, nivel-1, cantidad);
            contar_por_nivel_aux(arbol->derecho, nivel-1, cantidad);
        }
    }
}
```

Implementacion en asm

```asm
%define OFFSET_DERECHO 0
%define OFFSET_IZQUIERDO 8
%define OFFSET_PADRE 16
%define OFFSET_DATA 24
%define OFFSET_BORRAR 32

%define NULL 0

contar_por_nivel:
    mov dword [rdx], 0
    call contar_por_nivel_aux

contar_por_nivel_aux:
    ; movemos datos a registros seguros
    push r12
    push r13

    ; en r12 esta arbol, r13d nivel y en rdx cantidad
    mov r12, rdi
    mov r13d, esi

    ; es arbol vacio?
    cmp r12, NULL
    je .fin

    ; nivel == 0 ?
    cmp r13d, 0
    je .raiz

    ; llamado para el hijo derecho
    mov rdi, [r12 + OFFSET_DERECHO]
    mov esi, r13d
```

```
33      dec esi
34      call contar_por_nivel_aux
35
36      ; llamado para el hijo derecho
37      mov rdi, [r12 + OFFSET_IZQUIERDO]
38      mov esi, r13d
39      dec esi
40      call contar_por_nivel_aux
41
42      jmp .fin
43
44      .raiz:
45      ; *cantidad += 1
46      mov eax, [rdx]
47      inc eax
48      mov [rdx], eax
49
50      .fin:
51      pop r13
52      pop r12
53      ret
```

Notar que en esta implementacion no se preocupa por guardar `cantidad` nunca. Esto se puede hacer porque esta implementacion mantiene intacto el registro `rdx`, de esta forma, el llamado no le afecta. `arbol` y `nivel`, en cambio, van alterandose para cada llamado. Necesitamos establecer registros que nuestra funcion mantenga en su estado original posterior al llamado. Podia usarse cualquiera (siempre y cuando se mantenga la convencion C)

b) Implementacion en C

```
1  void borrar_arbol(struct nodo* raiz) {
2
3      if (raiz != NULL) {
4          nodo* izquierdo = raiz->izquierdo;
5          borrar_arbol(izquierdo);
6
7          nodo* derecho = raiz->derecho;
8          borrar_arbol(derecho);
9
10         (raiz->borrar)(raiz->data);
11         free(raiz);
12     }
13
14 }
15
16 void borrar_nivel_conectando_izquierda(struct nodo** arbol, unsigned int nivel) {
17     nodo* raiz = *arbol;
18     if (raiz != NULL) {
19         if (nivel == 0) {
20             *arbol = raiz->izquierdo;
21
22             if (raiz->izquierdo != NULL) {
23                 raiz->izquierdo->padre = raiz->padre;
24             }
25
26             // se desconecta el hijo izquierdo para que no sea borrado
27             raiz->izquierdo = NULL;
28
```

```
29              borrar_arbol(raiz);
30          }
31
32          else {
33              nivel -= 1;
34
35              nodo** izquierdo = &(raiz->izquierdo);
36              borrar_nivel_conectando_izquierda(izquierdo,nivel);
37
38              nodo** derecho = &(raiz->derecho);
39              borrar_nivel_conectando_izquierda(derecho,nivel);
40
41          }
42      }
43  }
```

Implementacion en asm

```
1  borrar_arbol:
2      push r12
3
4      ;r12 contiene raiz
5      mov r12, rdi
6
7      cmp r12, NULL
8      je .fin
9
10     ; llamado para subarbol derecho
11     mov rdi, [r12 + OFFSET_DERECHO]
12     call borrar_arbol
13
14     ; llamado para subarbol izquierdo
15     mov rdi, [r12 + OFFSET_IZQUIERDO]
16     call borrar_arbol
17
18     ; llamado a funcion borradora con el puntero al dato en rdi
19     mov rdi, [r12 + OFFSET_DATA]
20     mov rsi, [r12 + OFFSET_BORRAR]
21     call rsi
22
23     ; llamado a free
24     mov rdi, r12
25     call free
26
27     .fin:
28     pop r12
29     ret
30
31
32 borrar_nivel_conectando_izquierda:
33     push r12
34     push r13
35     push r14
36
37     mov r12, rdi ; arbol esta en  r12
38     mov r13, [rdi] ; raiz esta en r13
39     mov r14d, esi ; nivel esta en r14d
40
41     cmp r13, NULL
```

```
42      je .fin
43
44      cmp r14d, 0
45      jne .caso_recursivo
46
47          .caso_base:
48
49          ; *arbol = raiz->izquierdo
50          mov rdi, [r13 + OFFSET_IZQUIERDO]
51          mov [r12], rdi
52
53          ; raiz->izquierdo == NULL ?
54          cmp rdi, NULL
55          je .borrar_derecha
56
57          ; raiz->izquierdo->padre = raiz->padre
58          mov rsi, [r13 + OFFSET_PADRE]
59          mov [rdi + OFFSET_PADRE], rsi
60
61          .borrar_derecho:
62
63          ; raiz->izquierdo = NULL;
64          mov qword [r13 + OFFSET_IZQUIERDO], NULL
65
66          ; borrar_arbol(raiz)
67          mov rdi, r13
68          call borrar_arbol
69
70
71      jmp .fin
72
73          .caso_recursivo:
74
75          ;nivel -= 1
76          dec r14d
77
78          ; llamado recursivo izquierdo
79          lea rdi, [r13 + OFFSET_IZQUIERDO]
80          mov esi, r14d
81          call borrar_nivel_conectando_izquierda
82
83          ; llamado recursivo derecho
84          lea rdi, [r13 + OFFSET_DERECHO]
85          mov esi, r14d
86          call borrar_nivel_conectando_izquierda
87
88
89      .fin:
90      pop r14
91      pop r13
92      pop r12
93      ret
```

Una sutileza. Cuando se elimina un nodo `n`, se conecta el hijo izquierdo de `n` con el padre de `n`. Para ello tenemos que setear 2 campos: El de `padre` para el hijo y el de `derecho` o `izquierdo` para el padre. Ahora, se puede setear este ultimo campo sin apelar a casos. Para esto se usa doble punteros. Viendo la version en C, se encuentra que en el llamado recursivo (lineas 35-40) el nuevo doble puntero es un puntero al campo `derecho` o `izquierdo` de `n` (NO es el contenido del campo sino que

un puntero al campo en si!). De esta forma, `arbol` es un puntero al campo del que "proviene". Es decir, `derecho` o `izquierdo` de su padre.
Entonces, el paso que da la conexion del padre de `n` con el hijo izquierdo de `n` es `*arbol = raiz->izquierdo`

4.3 Ejercicio parcial 28/6/18. Listas circulares

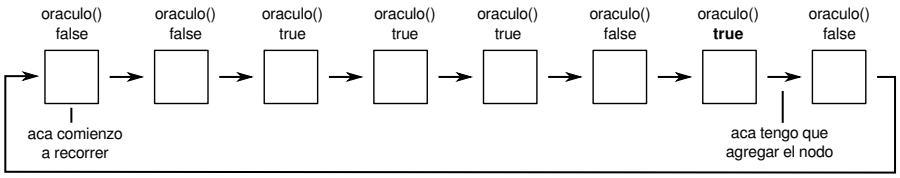
Ej. 1. (40 puntos)

Sea una lista circular que respeta la siguiente estructura:

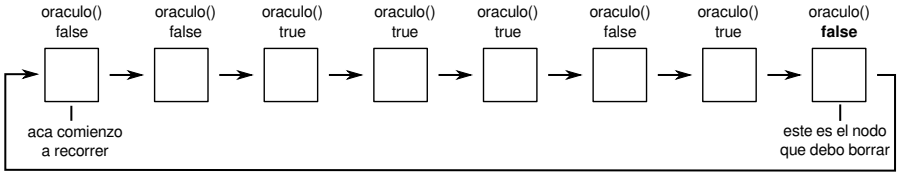
```
struct nodo {
  struct nodo* siguiente,
  int dato,
  bool (*oraculo)()
}
```

Donde `siguiente` es un puntero al siguiente nodo, `dato` es un valor entero almacenado y `oraculo` es un puntero a función que no recibe nada y devuelve un valor de tipo `bool`. `bool` es un entero de 4 bytes, que se interpreta como `false` si vale `0` y `true` en caso contrario.
Se pide escribir 2 funciones:

- `void insertarDespuesDelUltimoTrue(nodo* listaCircular, int nuevoDato, bool (*nuevoOraculo)())`:
  Inserta un nuevo nodo después del último nodo para el cual el llamado a su `oraculo` devuelva `true`. El nuevo nodo debe contener los campos `dato` y `oraculo` pasados por parámetro. Si el orácuo de ningún nodo devuelve `true`, no se debe insertar nada.



- `void borrarUltimoFalse(nodo** listaCircular)`:
  Borra el último nodo cuya llamada a `oraculo` devuelva `false`. De borrar el primero debe modificar el puntero a la lista circular. Si el orácuo de ningún nodo devuelve `false`, no se debe borrar nada.



(8p) a. Implementar la función `insertarDespuesDelUltimoTrue` en C.

(15p) b. Implementar la función `insertarDespuesDelUltimoTrue` en ASM.

(17p) c. Implementar la función `borrarUltimoFalse` en ASM.

a) Implementacion en C

```c
#define FALSE 0
#define TAMANO_NODO 24

void insertarDespuesDelUltimoTrue(nodo* listaCircular, int nuevoDato, bool
(*nuevoOraculo)()) {

    nodo* inicial = listaCircular;
    nodo* ultimo_true = inicial;
    listaCircular = inicial->siguiente;


    /* se recorre la lista hasta que se vuelva al principio*/
    while(listaCircular != inicial) {
        if(listaCircular->oraculo() != FALSE) {
            ultimo_true = listaCircular;
        }
        listaCircular = listaCircular->siguiente;
    }

    /* si habia alguno true*/
    if(ultimo_true->oraculo() != FALSE) {
        nodo* nuevo = (nodo*) malloc(TAMANO_NODO);

        nuevo->siguiente = ultimo_true->siguiente;
        ultimo_true->siguiente = nuevo;

        nuevo->dato = nuevoDato;
        nuevo->oraculo = nuevoOraculo;
    }
}
```

Notar que al finzalizar el `while` todavia no se chequeo el oraculo de `inicial`. Cuando termina el cuerpo del ciclo pueden darse 2 casos:

- `ultimo_true` se actualizo alguna vez. En este caso significa que hay algun `true` posterior a `inicial`. De esta forma el oraculo de `ultimo_true` es `true`.
- `ultimo_true` no se actualiza nunca. En este caso, significa que no hay `true` posterior a `inicial`. Resta ver si el nodo debe agregarse luego de `inicial`. Como `inicial = ultimo_true`, puede chequearse este ultimo.

b) Implementacion en asm

```asm
%define OFFSET_SIGUIENTE 0
%define OFFSET_DATO 8
%define OFFSET_ORACULO 16

%define TAMANO_NODO 24

%define FALSE


insertarDespuesDelUltimoTrue:
    push r12
    push r13
    push r14
    push r15
    push rbx ; la pila ya queda alineada para el malloc
```

```
16
17      mov r12, rdi ; r12 sera inicial
18      mov r13, rdi ; r13 sera ultimo_true
19      mov r14, [rdi + OFFSET_SIGUIENTE]   ; r14 sera el nodo actual
20      mov r15, rdx ; r15 sera nuevo oraculo
21      mov ebx, esi ; ebx sera nuevo dato
22
23          .ciclo:
24          ; actual == inicial ?
25          cmp r14, r12
26          je .lista_recorrida
27
28          ; el oraculo  de actual indica false ?
29          mov rsi, [r14 + OFFSET_ORACULO]
30          call rsi
31          cmp eax, FALSE
32          je .actualizar_nodo
33
34          ; ultimo_true  = actual
35          mov r13, r14
36
37          .actualizar_nodo:
38          ; actual = actual->siguiente
39          mov r14, [r14 + OFFSET_SIGUIENTE]
40
41          jmp .ciclo
42
43      .lista_recorrida:
44      ; el oraculo  de ultimo_true indica false ?
45      mov rsi, [r13 + OFFSET_ORACULO]
46      call rsi
47      cmp eax, FALSE
48      je .fin
49
50      ; nuevo = malloc(TAMANO_NODO)
51      mov rdi, TAMANO_NODO
52      call malloc
53
54      ; nuevo->siguiente = ultimo_true->siguiente
55      mov rsi, [r13 + OFFSET_SIGUIENTE]
56      mov [rax + OFFSET_SIGUIENTE], rsi
57
58      ; ultimo_true->siguiente = nuevo
59      mov [r13 + OFFSET_SIGUIENTE], rax
60
61      ; nuevo->dato = nuevoDato
62      mov [rax + OFFSET_DATO], ebx
63      ; nuevo->oraculo = nuevoOraculo
64      mov [rax + OFFSET_ORACULO], r15
65
66      .fin:
67      pop rbx
68      pop r15
69      pop r14
70      pop r13
71      pop r12
72      ret
```

c) Implementacion en C

```
void borrarUltimoFalse(nodo** listaCircular) {
    nodo* inicial = *listaCircular;
    nodo** ultimo_false  = listaCircular;
    listaCircular = &(*listaCircular)->siguiente;


    /* se recorre la lista hasta que se vuelva al principio*/
    while(*listaCircular != inicial) {
        if(*listaCircular->oraculo() == FALSE) {
            ultimo_false = listaCircular;
        }
        listaCircular = &(*listaCircular)->siguiente;
    }

    /* si habia alguno false*/
    if(*ultimo_false->oraculo() == FALSE) {
        nodo* a_eliminar = *ultimo_false;

        *ultimo_false = a_eliminar->siguiente;

        if( a_eliminar == inicial) {
            *listaCircular = a_eliminar->siguiente;
        }
        free(a_eliminar);
    }
}
```

Eliminar el primer nodo es un caso especial. Ya que es apuntado desde 2 lugares, (`listaCircular` pasado por parametro y el ultimo nodo de la lista). En este caso deben actualizarse ambos. Un detalle es que esta implementacion supone que la eliminacion de un nodo no destruye completamente la lista. Es decir si hay un nodo `false` entonces la lista circular tiene por lo menos 2 nodos.

```
borrarUltimoFalse:
    push r12
    push r13
    push r14 ; la pila ya queda alineada para el free


    mov r12, [rdi] ; r12 sera inicial
    mov r13, rdi  ; r13 sera ultimo_false
    mov r14, [rdi]
    lea r14, [r14 + OFFSET_SIGUIENTE] ; r14 sera actual, en este caso es doble
    puntero

        .ciclo:
        ; *actual == inicial ?
        cmp [r14], r12
        je .lista_recorrida

        ; el oraculo de actual indica true ?
        mov rsi, [r14]
        mov rsi, [rsi + OFFSET_ORACULO]
        call rsi
        cmp eax, FALSE
        jne .actualizar_nodo

        ; ultimo_false  = actual
        mov r13, r14

```

```
27          .actualizar_nodo:
28          ; actual = & (*actual->siguiente)
29          mov r14, [r14]
30          lea r14, [r14 + OFFSET_SIGUIENTE]
31          jmp .ciclo
32
33      .lista_recorrida:
34      ; el oraculo de ultimo_false indica true ?
35      mov rsi, [r13]
36      mov rax, [rsi + OFFSET_ORACULO]
37      call rsi
38      cmp eax, FALSE
39      jne .fin
40
41      mov rsi, [r13] ; rsi sera a_eliminar
42
43      ; *ultimo_false = a_eliminar->siguiente
44      mov rdi, [rsi + OFFSET_SIGUIENTE]
45      mov [r13], rdi
46
47      ; a_eliminar == inicial ?
48      cmp rsi, r12
49      jne .liberar_memoria
50
51      ; *actual = a_eliminar->siguiente
52      mov [r14], rdi
53
54      .liberar_memoria:
55      ; free(a_eliminar)
56      mov rdi, rsi
57      call free
58
59      .fin:
60      pop r14
61      pop r13
62      pop r12
63      ret
```

5 Instrucciones de FPU

La CPU puede realizar operaciones matematicas mas avanzadas como seno, coseno y tangente. Para esto se utiliza la *floating point unit* (FPU). El comportamiento de este es un poco diferente.

- Funciona con un stack de registros propios. Nombrados `st0` a `st7`. La FPU solo realiza operaciones con estos registros.
- Estos registros solo pueden ser cargados usando posiciones de memoria. Es decir, si se tiene algun valor en un registro como `rax` y quiere usarse este en la FPU, se debe copiar el contenido de `rax` en memoria y luego utilizar la instruccion para cargar la FPU tomando como parametro la direccion donde se encuentra el dato.
- Similar, para obtener los resultados de la FPU en un registro de proposito general debe escribirse en memoria.

- Dependiendo de la instruccion que usemos para cargar los registros puede tomar `float` , `double` o `int`. Opera internamente como `double`. Similar para colocar los resultados en memoria.
- Algunas operaciones solo pueden realizarse con el tope de la pila (`st0`).

Aca se encuentra una tabla de las instrucciones de FPU con una breve descripcion.

5.1 Ejemplo de uso

Supongamos que hay que implementar `double f(float x)`

$$f(x) = \sqrt{cos\left(\frac{2\pi}{x+5}\right)} \tag{1}$$

```
1  section .data
2  x : dd 0
3  y : dq 0
4  5_float : dd 5.0
5
6  global f
7  section .text
8
9  f:
10     ; x se encuentra en los primeros 32 bits de xmm0
11     movss [x], xmm0
12     finit
13     fld dword [x] ;dword ya que es un float, son 32 bits.
14     fld dword [5_float] ;dword ya que es un float, son 32 bits.
15     ;    ............
16     ;    |     5      |  ST0
17     ;    |------------|
18     ;    |     x      |  ST1
19     ;    |------------|
20     ;    |            |  ST2
21     ;    |------------|
22
23
24     faddp st1, st0    ; suma st0 a st1 y realiza pop, eliminando st0
25     ;    ............
26     ;    |   x + 5    |  ST0
27     ;    |------------|
28     ;    |            |  ST1
29     ;    |------------|
30     ;    |            |  ST2
31     ;    |------------|
32
33
34     fldpi
35     ;    ............
36     ;    |     pi     |  ST0
37     ;    |------------|
38     ;    |   x + 5    |  ST1
39     ;    |------------|
40     ;    |            |  ST2
41     ;    |------------|
42
```

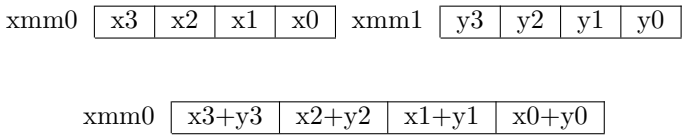
```
43      fadd st0, st0
44      ;    ............
45      ;    |    2pi     |  ST0
46      ;    |------------|
47      ;    |   x + 5    |  ST1
48      ;    |------------|
49      ;    |            |  ST2
50      ;    |------------|
51
52      fdivrp st1, st0
53      ;    ............
54      ;    |  2pi/x+5   |  ST0
55      ;    |------------|
56      ;    |            |  ST1
57      ;    |------------|
58      ;    |            |  ST2
59      ;    |------------|
60
61      fcos
62      ;    ............
63      ;    |cos(2pi/x+5)|  ST0
64      ;    |------------|
65      ;    |            |  ST1
66      ;    |------------|
67      ;    |            |  ST2
68      ;    |------------|
69
70      fsqrt
71      ;    ................
72      ;    |sqrt cos(2pi/x+5)|  ST0
73      ;    |-----------------|
74      ;    |                 |  ST1
75      ;    |-----------------|
76      ;    |                 |  ST2
77      ;    |-----------------|
78
79      fstp qword [y]
80      ;    ............
81      ;    |            |  ST0
82      ;    |------------|
83      ;    |            |  ST1
84      ;    |------------|
85      ;    |            |  ST2
86      ;    |------------|
87
88      movsd xmm0, [y]
89      ret
```

6 SIMD

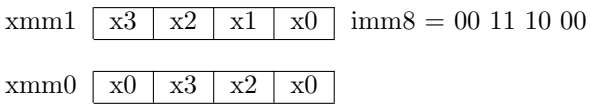
Supongamos que tenemos dos pares de numeros (x_0, x_1) (y_0, y_1) y queremos sumarlos para obtener $(x_0 + y_0, x_1 + y_1)$.
Si estos numeros son `int` podemos sumarlos realizando 2 instrucciones `add`. Sin embargo un `int` solo ocupa 4 bytes, tenemos registros de 8 bytes como `rax`. En cada uno de estos registros entran 2 `int`. Tenemos incluso registros de 16 bytes como `xmm0`, estos pueden almacenar 4 `int`.

Seria util (y mas performante) si pudiesemos realizar esta misma instruccion de forma paralela y no secuencial. *Single Instruction- Multiple Data* (SIMD)
Las CPU modernas poseen instrucciones para esta clase de problemas. El *SSE instrucion set* (para `xmm`) y su extension *AVX instruction set* (para `ymm`).
Algunos ejemplos ilustrativos de las instrucciones.

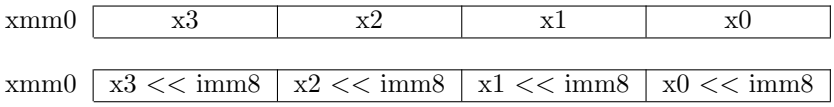
- `paddd xmm0, xmm1`. *Parallel Add Doubleword*. Suma los enteros en `xmm1` con los de `xmm0`. Guarda el resultado en `xmm0`



- `pshufd xmm0, xmm1, imm8`. *Shuffle Packed Doublewords*. Cada 2 bits de `imm8` indica que doubleword de `xmm1` selecionar, se coloca en `xmm0`



- `pslld xmm0, imm8`. *Parallel Shift Left Logical Doubleword*. Shiftea a la izquierda cada una de las double word en `xmm0` la cantidad indicada por imm8



No todas las instrucciones normales tienen su equivalente vectorizado. Por ejemplo, no existe una instruccion para division de enteros. Aca se encuentra una tabla de instrucciones entre las cuales se encuentran las SSE y las AVX (util para saber si cierta instruccion existe).

Bien, como afecta la performance el usar estas instrucciones?. Este articulo hace una comparacion de esto para C++ con un par de programas simples. Un detalle interesante es que los compiladores como *gcc* "intentan" colocar instrucciones SIMD para optimizar codigo. Puede compararse compilando algun programa en C con flag `-O0` y `-O3`, desensamblar ambos con `objdump` y observar las diferencias.

6.1 Ejemplo: Sumar 2 arreglos de enteros

La funcion `void suma_vectorial(int* x, int* y, int* res, unsigned int dimension)` realiza lo siguiente:
`res[i] = x[i] + y[i]` para $0 \leq$ `i` $<$ dimension. Para simplificar, suponer que dimension es multiplo de 4.

```
suma_vectorial:
    ;rdi contiene x
    ;rsi contiene y
    ;rdx contiene z
    ;ecx contiene dimension

    .ciclo:
    cmp ecx, 0
    je. fin

    movdqu xmm0, [rdi] ;  xmm0 = |x3|x2|x1|x0|
    movdqu xmm1, [rsi] ;  xmm1 = |y3|y2|y1|y0|

    ; parallel add dobleword
    paddd xmm0, xmm1 ; xmm0 = |x3+y3|x2+y2|x1+y1|x0+y0|

    ; guardamos los datos
    movdqu [rdx], xmm0

    ; avanzamos el puntero 4 int. 16 bytes
    add rdx, 16

    ; reducimos dimension 4
    sub ecx, 4

    jmp. ciclo

    .fin:
    ret
```

7 Ejercicios de SIMD

7.1 Invertir texto

Queremos escribir una funcion `void invertir(char* mensaje, int len)`
Esta funcion toma un puntero a una string y su largo e invierte el orden de los caracteres.
Por ejemplo *hola* → *aloh*.
Supongamos que el largo del mensaje es al menos 8.
Puede resolverse sin aprovechar SIMD recorriendo el arreglo e ir intercambiando primeros con ultimos. Mas concretamente en C

```c
void invertir(char* mensaje, int len) {
    int por_izquierda = 0;
    int por_derecha = len-1;
    /*mientras no se crucen*/
    while (por_izquierda < por_derecha) {
        /*se intercambian los caracteres*/
        char temp = mensaje[por_izquierda];
        mensaje[por_izquierda] = mensaje[por_derecha];
        mensaje[por_derecha] = temp;

        /*avanzan los indices*/
        por_izquierda++;
        por_derecha--;
    }
}
```

Usando SIMD se pueden intercambiar en paralelo 8 pares de caracteres. La idea es cargar por izquierda y por derecha, invertirlos e intercambiarlos.

```
section .data
invertidor: db 7,6,5,4,3,2,1,0

section .text
invertir:
    ;rdi es puntero a inicio
    ; esi es largo

    ;limpiamos parte alta de rsi
    sll rsi, 32
    srl rsi, 32

    ; rdi sera puntero a 8 por izquierda
    ; rsi sera puntero a 8 por derecha
    add rsi, rdi
    sub rsi, 8

    movdqu xmm0, [invertidor] ;  xmm0 = |0|1|2|3|4|5|6|7|

    .ciclo:
    cmp rdi, rsi
    ja .fin

    ; cargado de grupos de 8
    movdqu xmm1, [rdi]
    movdqu xmm2, [rsi]

    ; se invierten los grupos
```

```
29      pshufb xmm1, xmm0
30      pshufb xmm2, xmm0
31
32      ; intercambio de grupos
33      movdqu [rdi], xmm2
34      movdqu [rsi], xmm1
35
36      ; se avanzan los indices
37      add rdi, 8
38      sub rsi, 8
39
40      jmp .ciclo
41
42      .fin:
43      ret
```

Un detalle de esta implementación es lo que sucede cuando el largo del mensaje no es multiplo de 8. Si se da esto, en el ultimo ciclo sucedera que los caracteres tomados por izquierda y por derecha tienen un solapamiento parcial. Dicho de otra forma, los ultimos caracteres de los tomados por izquierda serán los primeros caracteres de los tomados por derecha. Sin embargo, si se observan como 3 partes (izquierda, solapada y derecha) puede verse que la parte solapada es invertida en el lugar y las de derecha e izquierda se invierten e intercambian lugar. Esto es una inversion, por lo que funciona. Tambien puede verse como "un giro de 180" con la parte solapada como centro.

7.2 Ejercicio parcial 10/5/18. Numeros de 3 bytes en big-endian

---

Ej. 2. (40 puntos)

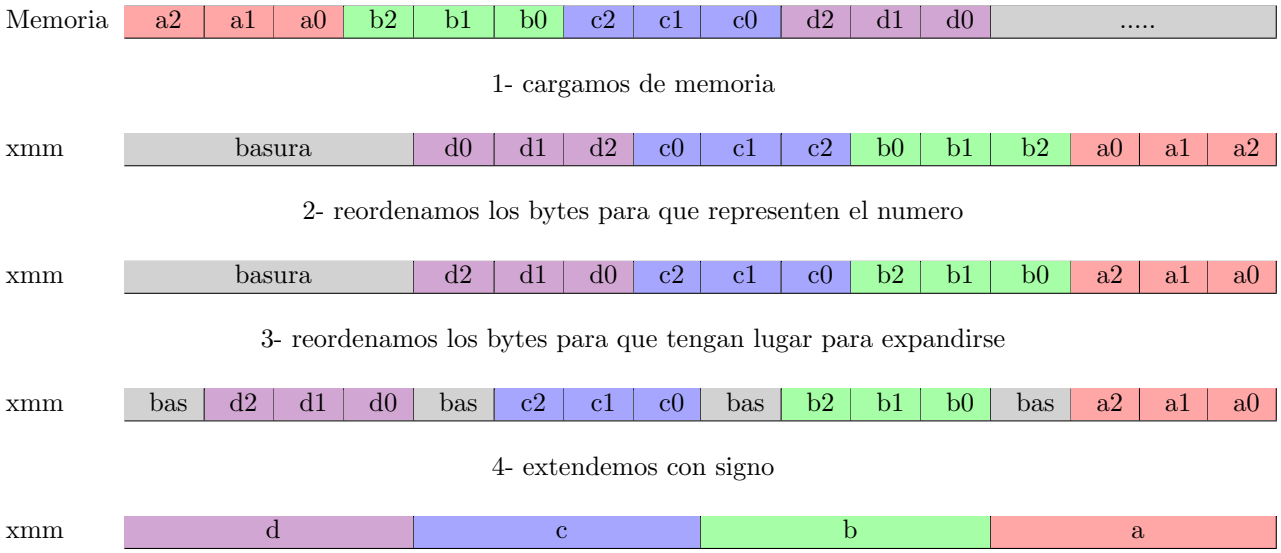
Considerar un vector de 16 números enteros con signo de 24 bits almacenados en big-endian.

(25p) a. Construir una función en ASM utilizando SIMD que dado un puntero al vector de números mencionado, retorne la suma de los números del vector como un entero de 4 bytes.

(15p) b. Modificar la función anterior para que a los números pares los multiplique por π. En este caso el resultado debe ser retornado como *double*.

---

a) 24 bits son 3 bytes. Un registro `xmm` puede almacenar 5.33 de estos numeros. Para hacerlo simple, tomemos en cada iteracion 4 numeros. Quedando en la parte alta informacion que no se va a utilizar. Como sobran exactamente 4 bytes podemos "castear" estos numeros para que se conviertan numeros de 4 bytes.
El problema es que estos numeros estan almacenados en *big-endian*. Como trabajamos con *little-endian*, cuando los numeros sean cargados de memoria tendran los bytes al reves de como los queremos.
Lo que haremos puede verse graficamente como:



Luego se suma en paralelo con un acumulador `xmm`. De esta forma suceden 4 sumatorias en paralelo. Notar que el paso 2 y 3 son reordenamientos por lo que podemos reducirlos en un solo *shuffle*.

```
1
2  section.data
3  ; al escribirlo hay que tener en cuenta que se invierten los bytes
4  reordenador:
5  db
6  2, 1, 0, 15,
7  5, 4, 3, 15,
8  8, 7, 6, 15
9  11, 10, 9, 15
10
11 section.text
12 funcion_pedida:
13     ;rdi contiene puntero a arreglo
14     mov rsi, 16
15     movdqu xmm1, [reordenador]
16
17
18     ;xmm2 sera nuestro acumulador paralelo
19     pxor xmm2, xmm2
20
21     .ciclo:
22     cmp rsi, 4
23     je .ultimos_cuatro
24
25     movdqu xmm0, [rdi]
26     call agregar_a_acumulador
27
28     ; 4 numeros de 3 bytes son 12 bytes
29     add rdi, 12
30     sub rsi, 4
31     jmp .ciclo
32
33     ; en los ultimos cuatro hay que ser cuidadosos ya que si lo hacemos como los
       demas estariamos accediendo a memoria por fuera del arreglo
```

```
34      .ultimos_cuatro:
35      sub rdi, 4
36      movdqu xmm0, [rdi]
37      pslldq xmm0, 4
38      call agregar_a_acumulador
39
40      .juntar:
41                              ;    xmm2 = |s3|s2|s1|s0|
42      movdqu xmm0, xmm2       ;    xmm0 = |s3|s2|s1|s0|
43      psrldq xmm0, 8          ;    xmm0 = |0|0|s3|s2|
44      paddd xmm0, xmm2        ;    xmm0 = |0|0|s3+s1|s2+s0|
45
46      movdqu xmm2, xmm0       ;    xmm2 = |0|0|s3+s1|s2+s0|
47      psrldq xmm2, 4          ;    xmm2 = |0|0|0|s3+s1|
48
49      paddd xmm0, xmm2        ;    xmm0 = |0|0|s3+s1|s3+s2+s1+s0|
50
51      movd eax, xmm0
52      ret
53
54  agregar_a_acumulador:
55      ; reordenamos segun el diagrama
56      pshufb xmm0, xmm1
57
58      ;movemos todo un byte a la izquierda para extender
59      pslldq xmm0, 1
60
61      ; extendemos.
62      ; psrad toma bits  y no bytes.
63      ; este es un shift aritmetico, por lo que el signo se va copiando
64      psrad xmm0, 8
65
66      ; agregamos a xmm2
67      paddd xmm2, xmm0
68      ret
```

b) Lo que se puede hacer es, una vez obtenidas las sumas parciales, separar en `|s2+s0|` y `|s3+s1|`, pasar ambos a `double` y multiplicar por π la suma de los elementos pares.

```
1       %define pi rdx
2       pi_ptr : dq 0
3       .
4       .
5       .
6       .
7       .
8       .juntar:
9                               ;    xmm2 = |s3|s2|s1|s0|
10      movdqu xmm0, xmm2       ;    xmm0 = |s3|s2|s1|s0|
11      psrldq xmm0, 8          ;    xmm0 = |0|0|s3|s2|
12      paddd xmm0, xmm2        ;    xmm0 = |0|0|s3+s1|s2+s0|
13
14      movdqu xmm2, xmm0       ;    xmm2 = |0|0|s3+s1|s2+s0|
15      psrldq xmm2, 4          ;    xmm2 = |0|0|0|s3+s1|
16
17      ; conversion a double
18      movss eax, xmm0
19      cvtsi2sd xmm0, eax
20
```

```
21      movss eax, xmm2
22      cvtsi2sd xmm2, eax
23
24      ; multiplicacion por pi de los elementos pares
25      finit
26      fldpi
27      fstp qword [pi_ptr]
28      mov pi, [pi_ptr]
29
30      movsd xmm1, pi
31      mulsd xmm0, xmm1 ;     xmm0 = |0| pi * (s2+s0)|
32
33      ; suma
34      addsd xmm0, xmm2  ;    xmm0 = |0|s3+ pi*s2 + s1 + pi*s0 |
35
36      ; el resultado ya se encuentra en xmm0
37      ret
```