Práctica 1 - Enteros

Organización del Computador 2

1er Cuatrimestre 2019

Los ejercicios marcados con ★ constituyen un subconjunto recomendado de ejercitación. Algunos ejercicios poseen una serie de preguntas bajo el indicador **Para pensar** cuyo objetivo es fomentar la discusión, análisis y profundización de los temas. Se aconseja leerlas a modo de repaso y consolidación.

1. Instrucciones básicas y modos de direccionamiento

Notas:
Se definen los tipos de datos `superlong` y `unsignedsuperlong` como:

```
typedef struct superlong_t {
    long x1;
    long x2;
} superlong;

typedef struct unsignedsuperlong_t {
    unsigned long x1;
    unsigned long x2;
} unsignedsuperlong;
```

Ejercicio 1

a) ¿Cuál forma de almacenamiento utilizan los procesadores `x86-64`: Little-endian o Big-endian? ¿Qué ventajas aporta una arquitectura Big-endian frente a una Little-endian? ¿Y al revés?
b) Definir el concepto de "visible al programador". ¿Cuáles son los registros de la arquitectura `x86-64` visibles para el programador? ¿Qué tamaño tienen? Indique para cada uno si tienen alguna función específica.
c) ¿En qué registro se almacenan los *flags* del procesador en la arquitectura

Para pensar: ¿En que contextos vale la pena extender la cantidad de registros ocultos o las funcionalidades de cada uno? ¿Qué condiciones es capaz de verificar el procesador a través de los flags? ¿Hay algún escenario en el que haría falta extenderlas?

Ejercicio 2 ★

Determine si son correctas las siguientes instrucciones. En caso afirmativo, indicar qué modo de direccionamiento se está utilizando y la acción que realiza. En caso negativo, justificar:

a) `MOV RAX, 4`
b) `MOV 4, RAX`
c) `MOV RAX, RBX`
d) `MOV RAX, EBX`
e) `MOV RAX, RBX+2`
f) `MOV RAX, [variable]`
g) `MOV [RAX], 4`
h) `MOV [RAX], EAX`

i) `MOV [EAX], EAX`
j) `MOV [RAX],[RCX]`
k) `MOV [variable], RAX`
l) `MOV 4, [RAX]`
m) `MOV [variable_1], [variable_2]`
n) `MOV BYTE [RAX], 4`
ñ) `MOV DWORD [RAX], 4`
o) `MOV RBX, [RAX+2]`
p) `MOV RBX, [RAX+RCX]`
q) `MOV RBX, [RAX+RCX*2+3]`
r) `MOV RBX, [RAX+RCX*3+2]`
s) `MOV RBX, [[RAX]]`
t) `MOV RBX, [RAX]+2`

Opcional: Dar una posible corrección de las instrucciones incorrectas en base a cómo fueron interpretadas. Notar la importancia de definir reglas de semántica y sintáctica claras en una arquitectura.

Para pensar: En una instrucción con dos operandos, ¿cuál es el fuente y cuál el destino? ¿Y en una de 3? ¿Cuáles tamaños de operador se encuentran disponibles en la arquitectura `x86-64`?

Ejercicio 3

¿Cuántos bytes ocupan los siguientes tipos de datos de C? Especificar en 32 y 64 bits.

a) `char`
b) `short`
c) `int`
d) `long`
e) `long long`
f) `superlong`
g) `float`
h) `double`

Opcional: Realice un programa en C que muestre los valores pedidos.

Para pensar: ¿Qué razones pudieron haber llevado a cambiar el tamaño de ciertos tipos de datos? ¿Qué ventajas y limitaciones poseen estos tamaños? (pensar en representación y rendimiento)

Ejercicio 4 ★

En Linux, las llamadas al sistema operativo se realizan por la interrupción 0x80. El número de *syscall* elegida se pasa por `RAX` y los primeros parámetros se pasan por los registros, en orden: `RBX`, `RCX`, `RDX`, `RSI`, `RDI`. Cada *syscall* tiene un número único. Por ejemplo, para imprimir en pantalla se utiliza la *syscall* write, `RAX` = 4.

a) Utilizando dicha *syscall* escriba un programa en lenguaje ensamblador que imprima por pantalla "`Hola Mundo!`".
b) ¿Cuáles son las distintas secciones que puede tener un programa en lenguaje ensamblador? ¿Cuáles están presentes en este programa?
c) ¿Qué son las pseudoinstrucciones? ¿Cuáles son las pseudoinstrucciones de `nasm`? ¿Cuáles están presentes en este programa?

Opcional: Escriba en lenguaje ensamblador un programa que imprima por pantalla 20 veces "`Hola Mundo!`" (atender a los ciclos).

Para pensar: ¿Qué pudo haber motivado a separar un programa en secciones en vez de tomarlo como un bloque entero? ¿Para qué funcionalidades consideraría necesarias realizar un llamado a sistema? ¿Perderíamos expresibilidad en los programas si no tuviésemos pseudoinstrucciones?

2. Operaciones Lógicas y Aritméticas

Ejercicio 5 ★

Denote las diferencias en comportamiento (*flags* y resultados) al ejecutar los siguientes pares de instrucciones. Como guía puede escribir un programa en lenguaje ensamblador y hacer uso del *debugger* o referirse al manual:

a) `INC/ADD`
b) `DEC/SUB`
c) `CMP/SUB`
d) `TEST/AND`
e) `NOT` y `NEG`

Opcional: Pensar bajo qué escenarios sería maś adecuado usar una instrucción en lugar de la otra, y bajo qué criterio.

Ejercicio 6

a) Suponer que el registro *RFLAGS* se encuentra con `OF=1`, pero nuestro programa necesita que este *Flag* esté en 0. Escriba una secuencia corta de instrucciones que cambie el valor de `OF` al deseado.
b) Escriba una secuencia corta de instrucciones que tome el dato almacenado en `AX`, ponga en 1 los 4 bits menos significativos, ponga en cero los 3 bits más significativos, invierta los bits 7, 8 y 9; y almacene el resultado en `BX`.
c) Sume un número con signo de 32 bits almacenado en `EAX` con otro también con signo de 64 bits, almacenado en memoria apuntado por `RSI`; y almacene el resultado en esa misma posición de memoria.
d) Sume un número con signo de 128 bits almacenado en `RDX:RAX` con otro también con signo de 128 bits almacenado en memoria, apuntado por `RSI`; y almacene el resultado en esa misma posición de memoria. Pensar cómo cambiaría su programa para las diferentes operaciones aritméticas.
e) Multiplique un número de 192 bits almacenado en `RDX:RBX:RAX` por 2^n; donde n es un entero sin signo menor que 95, almacenado en `RCX`.
f) Divida un número de 192 bits con signo almacenado en `RDX:RBX:RAX` por 2^n; donde n es un entero sin signo menor que 95, almacenado en `RCX`.

Ejercicio 7

Suponer que recibimos en AX el estado de la máscara de interrupción de los controladores de interrupciones (PIC 1 y 2). Esta máscara indica con un 1 en el bit i que la interrupción IRQ_i está deshabilitada.
Escriba un programa en lenguaje ensamblador que imprima en pantalla cuáles son las interrupciones habilitadas. Utilice dos estrategias distintas: máscaras y *shifts*.

3. Stack e interacción C-Assembler

Ejercicio 8 ★

a) Explique qué diferencia existe entre ejecutar la instrucción `PUSH RBX` y el siguiente conjunto de instrucciones:

- ```
  SUB RSP, 8
  MOV [RSP], RBX
  ```

- ```
  MOV [RSP], RBX
  DEC RSP
  DEC RSP
  DEC RSP
  DEC RSP
  DEC RSP
  DEC RSP
  DEC RSP
  DEC RSP
  ```

b) ¿Cuál es el propósito de la convención C?

Para pensar: ¿Qué es y para qué se usa la pila? ¿Es reemplazable por otro tipo de estructura? ¿Valdría la pena modificar la convención C para preservar una mayor cantidad de información? ¿Qué desventajas podría generar?

Ejercicio 9 ★

Muestre el contenido de la pila y los registros cuando se llama a las siguientes funciones en la convención C en 32 y 64 bits:

a) `int func_a(long a, long b);`
b) `int* func_b(long a, long* b);`
c) `void func_c(short int a, long b);`
d) `void func_d(long int a, char b, int c);`
e) `void func_e(char b, superlong* a, int c);`

Para pensar: ¿Qué registros preservan? ¿En qué registros se retornan resultados?

Ejercicio 10

Dado el siguiente programa en lenguaje C:

```
long globalNoInit;
long globalInit = 31416;
const long globalInitConst = 14142;

int main(int argc, char* argv[]){
    long localNoIni;
    long localIni = 27182;
    return 0;
}
```

a) ¿Dónde deberían estar definidas cada una de las variables y constantes del programa? (`.data`, `.rodata`, `.bss`, en la pila, etc).
b) Compile el programa con la opción `-S` y observe el archivo `.s` obtenido. ¿Se cumplen sus predicciones?

Ejercicio 11

Escriba una función en lenguaje ensamblador `x86-64` que:

a) sume dos números de 128 bits con signo, cuyo prototipo sea:
`void suma(superlong* a, superlong* b, superlong* resultado)`
¿Cambia el código en lenguaje ensamblador si el prototipo de la suma es:
`void suma(unsignedsuperlong* a, unsignedsuperlong* b, unsignedsuperlong* resultado)`?
b) Compute x^y, cuyo prototipo sea:
`void power(long x, unsigned long y, superlong* resultado)`.
Se puede asumir que el resultado de la operación entra en 128 bits.
c) Calcule la cantidad de factores primos de un número de 64 bits sin signo pasado como parámetro. Se pide respetar el siguiente código C:

```
unsigned cantFactoresPrimos(unsigned long n){
    int k = 1, d = 2;
    while(d*d <= n) {
        if (n mod d == 0){
            k++;
            n = n/d;
        }
        else d++;
    }
    return k;
}
```

Ayuda: utilizar que $d_n \times d_n = (d_{n-1} \times d_{n-1}) + 2d_{n-1} + 1$, donde $d_n = d_{n-1} + 1$.

4. Estructuras estáticas: Vectores y Matrices

Ejercicio 12

a) Para cada uno de los siguientes ítems, indicar el valor en bytes del desplazamiento en memoria (*offset*) requerido para direccionar sucesivamente los elementos de la estructura en cuestión. Indique para cada caso al menos 3 formas de acceder a cada elemento.

- vector de enteros de 8 bits sin signo (pensar también los casos de 16, 32 y 64 bits)
- vector de enteros de 12 bits empaquetado (considerar lo que sucede en el caso empaquetado)
- Matriz de $n \times m$ números de 64 bits sin signo almacenada por filas (considerar también el caso en que está almacenada por columnas)

b) Escribir una función en lenguaje ensamblador que, dado un vector v de números de 64 bits sin signo y de dimensión n, calcule la suma de todos los elementos de v. Puede asumirse que esta suma no supera los 64 bits. El prototipo de la función es el siguiente: `long sumarTodos(long *v, unsigned short n);`
¿En qué cambiaría el código si el vector contuviese números de 32 bits? ¿Y hubiese que sumar dos vectores a la vez?

Ejercicio 13 ★

Se tiene una matriz M que almacena números de 64 bits con signo, cuya dimensión es $n \times n$ (n de 16 bits sin signo). Escriba en lenguaje ensamblador:

a) `long esSimetrica(long *M, unsigned short n)`: Una función que indique si la matriz M es simétrica.
b) `long diagonalesIguales(long *M, unsigned short n)`: Una función que indique si la suma de los elementos de la diagonal principal es igual a la suma de elementos de la diagonal traspuesta.
c) `long diagonalDominante(long *M, unsigned short n)`: Una función que indique si la matriz M es diagonal dominante. Formalmente, se dice que la matriz M de dimensión $n \times n$ es diagonal dominante cuando se satisface,

$$|m_{i,i}| \geq \sum_{\substack{j=1 \\ j \neq i}}^{n} |m_{i,j}| \quad \forall i = 1 \dots n$$

Para pensar: Pensar en otras propiedades que se puedan comprobar sobre una matriz, como ser tridiagonal o triangular superior. ¿Cómo afecta conocer estas propiedades de antemano al querer operar sobre la estructura? Pensar en las operaciones de suma, promedio y producto de matrices.

5. Estructuras dinámicas

Ejercicio 14

Supongamos que contamos con tres vectores distintos tales que cada uno de ellos contiene respectivamente instancias de las siguientes estructuras:

```
struct A {
    int n;
    int m;
    struct A *s;
} __attribute__((packed));
```

```
struct B {
    unsigned int n;
    unsigned short v[3];
    struct B *s;
} __attribute__((packed));
```

```
struct C {
    unsigned char c;
    struct C *v[10];
} __attribute__((packed));
```

a) Indicar en cada caso el offset requerido para direccionar sucesivamente los elementos del vector.
b) Indicar, para cada `struct`, los offsets de cada uno de sus miembros.
c) Pensar cómo afectaría las respuestas anteriores tener matrices que contengan dichas estructuras.

Para pensar: ¿Cuáles son las ventajas y desventajas de empaquetar los datos de una estructura? ¿Cuánto influye la alineación de datos con la pila en el funcionamiento de un programa?

Ejercicio 15 ★

Dado un nuevo tipo de datos llamado `lista` para manejar enteros con signo, cuya definición en C es la siguiente:

```
typedef struct nodo_t {
    long    dato;          /* dato del nodo (un entero) */
    struct  nodo_t *prox;  /* puntero al siguiente */
} nodo;

typedef struct lista_t {
    nodo*   primero;       /* puntero al primer nodo */
} lista;
```

a) `int iesimo(lista* l, unsigned long i)`: Escriba una función que dada una lista y un entero i devuelva el i-ésimo elemento de la lista.
b) `void agregarAd(lista* l, long n)`: Escriba una función que dada una lista y un entero n lo agregue al comienzo de la lista.
c) Escribir las funciones en lenguaje ensamblador que permitan buscar un elemento en una lista, agregarlo en una posición arbitraria y quitarlo respectivamente. Pensar en cómo debe ser el prototipo de cada una (atender al uso de punteros y las funciones malloc y free).

Opcional: Consideremos ahora las listas que cada uno de sus nodos almacenan una lista de strings de la forma:

```
typedef struct palabra_t {
    char *str;
    struct palabra_t *prox;
} palabra;

typedef struct oracion_t {
    palabra *primero;
} oracion;
```

```
typedef struct parrafo_t {
    oracion *dato;
    struct parrafo_t *prox;
} parrafo;

typedef struct texto_t {
    parrafo *primero;
} texto;
```

Si interpretamos a cada string como una palabra, podemos imaginar que una lista de strings representa una oración y, a su vez, una secuencia de oraciones representa un párrafo. Con esto en mente, escribir las funciones en lenguaje ensamblador que permitan imprimir por pantalla respectivamente una palabra, oración y párrafo. Tener en cuenta que se permite utilizar la función `printf`.

Para pensar: ¿Qué es un puntero doble? ¿Cómo cambiarían las funciones anteriores si la lista fuese referenciada a través de este? Pensar en escenarios en que sea necesario usar punteros dobles frente a los simples.

Ejercicio 16

Sea el siguiente tipo de datos `dicc` (diccionario sobre árbol binario) para manejar claves y significados, se piden definir en lenguaje ensamblador las funciones que permitan obtener el significado de una palabra, agregar una nueva definición, modificarla y eliminarla respectivamente. Pensar en el prototipo de las funciones y el manejo de punteros:

```
typedef struct _nodo_t {
    char *clave;
    char *significado;
    struct nodo_t *izq; /* puntero al arbol izquierdo */
    struct nodo_t *der; /* puntero al arbol derecho */
} nodo;

typedef struct _dicc_arb_bin_t {
    nodo *raiz;
} dicc_arb_bin;
```

Invariante de representación: El árbol binario siempre está ordenado. Para todo nodo vale que las claves en el árbol izquierdo son menores a la clave del nodo actual; y las claves en el árbol derecho son mayores a la clave del nodo actual.
Se tienen las funciones:
`int esMayor(char* clave1, char* clave2)` que devuelve 1 si clave1 $\geq$ clave2 y 0 si no (si clave2 $>$ clave1); utilizando el orden lexicográfico de las palabras.
`esIgual(char *clave1, char *clave2)` que devuelve 1 cuando clave1 = clave2 y 0 en caso contrario.

Para pensar: ¿Bajo qué circumstancias resultaría necesario implementar un diccionario? Busque ejemplos de sistemas actuales.

Ejercicio 17

Sea el siguiente tipo de datos `conj` (conjunto sobre listas) para manejar conjuntos de números enteros con signo, se piden definir en lenguaje ensamblador las funciones que permitan verificar si un elemento está en el conjunto, agregar uno nuevo y eliminarlo respectivamente:

```
typedef struct nodo_t{
    long elem;                  /* elemento (un entero con signo) */
    struct nodo_t *siguiente;   /* puntero al siguiente */
} dato;

typedef struct conj_t {
    nodo* primero; /*puntero al primer elemento del conjunto */
} conj;
```

Invariante de representación: el conjunto no tiene elementos repetidos y esta representado sobre una lista ordenada en forma ascendente.

Para pensar: ¿Qué otras operaciones podrían programarse sobre un conjunto? ¿Cómo transformaría una lista en un conjunto teniendo en cuenta el invariante de representación?

Ejercicio 18

Dado el tipo de datos `hash` para manejar datos de números enteros con signo, se piden definir en lenguaje ensamblador las funciones que permitan verificar que un elemento está en el hash, agregar uno nuevo y eliminarlo respectivamente:

```
typedef struct_nodo_t {
    long valor;
    struct _nodo _t *prox;
} nodo;
```

```
typedef struct _hash_t {
    nodo *arreglo[20] ;
} hash;
```

Nótese como la estructura del hash contiene un arreglo de 20 posiciones con punteros a nodos. De estos cuelga una lista de números enteros con signo. Este hash en particular maneja números enteros entre -1000 y 1000 ordenándolos por centena sin repetidos. Dado un número entero con signo x, si -1000 $\leq x <$ -900 entonces x se va a almacenar en la primera lista, si -900 $\leq x <$ -800 se va a almacenar en la segunda y así sucesivamente.

Para pensar: ¿Bajo qué escenarios resultaría útil implementar una estructura de hash? ¿Qué pasaría si quisiésemos extender el arreglo?

Ejercicio 19

Dado un *File System* sencillo cuya definición viene dada por las siguientes estructuras:

```
enum entry_type { TYPE_FILE, TYPE_DIR };

typedef struct dir_entry {
    enum entry_type this_entry_type;
    char name[8+3];
    void *first_entry;
    void *next_entry;
} __attribute__((packed)) *ptr_dir_entry;

typedef struct file_entry {
    enum entry_type this_entry_type;
    char name[8+3];
    unsigned long attr;
    unsigned long size;
    void *data;
    void *next_entry;
} __attribute__((packed)) *ptr_file_entry;
```

donde el árbol de directorios se arma de la siguiente manera:

- Se tiene un `dir_entry` inicial.
- `first_entry` apunta al primer elemento (archivo o directorio) incluido en él. `NULL` si no corresponde.
- `next_entry` apunta al siguiente elemento dentro del directorio actual. `NULL` si no corresponde.
- `data` apunta a los datos del archivo, guardados en binario.
- `size` indica el tamaño del archivo en bytes

Escriba una función en lenguaje ensamblador que recorra un *File System* pasado como parámetro y calcule el tamaño promedio de los archivos. El prototipo de la función es el siguiente: `unsigned long calcularPromedioSize(ptr_dir_entry root_folder)`

Nota: tener en cuenta que la sumatoria de los tamaños de todos los archivos no necesariamente entra en 64 bits.

Para pensar: ¿Qué es conceptualmente un file system? ¿Se le ocurren ejemplos reales en los que esta estructura resulte aplicable?

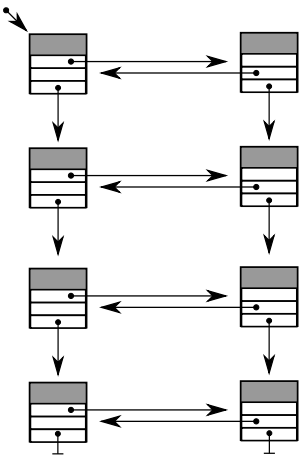


Figura 1: Ejemplo de cadena de ADN con 4 eslabones

6. Ejercicios de Parcial

Ejercicio 20

El fin de Buenos Aires se avecina; una plaga de mosquitos se acerca desde la selva húmeda del Congo con mucha sed de sangre latina. Para combatirla, el *Ministerio de control de plagas*, propuso esperar la llegada de la nube asesina de mosquitos con un gran conjunto de sapos. Por esta razón se le encargó a un grupo de estudiantes de biología la clonación a mansalva de los anfibios salvadores.

Los estudiantes de biología comenzaron a trabajar de inmediato en el proyecto pero confundieron el orden de la cadena de ADN del sapo y como resultado, obtuvieron ovejas verdes saltarinas.

Una cadena de ADN está formada por eslabones. Cada eslabón contiene una base nitrogenada, un puntero al eslabón pareja y otro puntero al eslabón inferior.

Ayudemos al grupo de biología a reordenar la cadena implementando una función que dada una cadena de ADN la reordene. El ordenamiento deberá respetar el resultado de la función `cmpBase()` (ya dada por los biólogos) de forma tal que **duplique** o **destruya** un par de eslabones, según corresponda.

La estructura es:

```
typedef enum action_e { borrar=0, duplicar=1 } action;

typedef struct eslabon_t{
    char base;
    struct eslabon_t* eslabon_pareja;
    struct eslabon_t* eslabon_inferior;
} __attribute__((__packed__)) eslabon;
```

La función a implementar es:

```
eslabon* ordenarCadena(eslabon* primer_eslabon,enum action_e (*cmpBase)(char* base1,
char* base2));
```

La función pasada por parámetro tiene la aridad `enum action_e cmpBase(char* eslabon1, char* eslabon2)` y retorna un valor de tipo enumerado, representando la acción a seguir. La función `ordenarCadena` debe retornar el primer eslabon, ya que este puede cambiar.

a) Escribir el pseudocódigo de la función `ordenarCadena`.

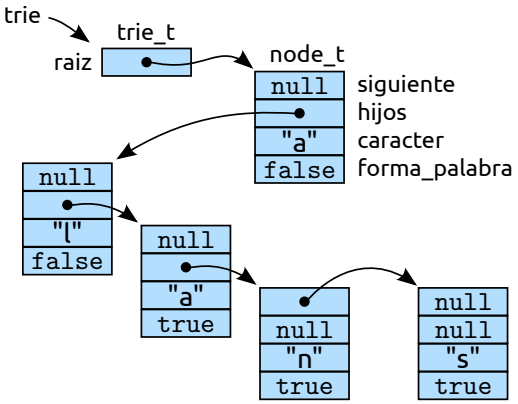
b) Implementar en ASM de 64 bits la función `ordenarCadena`.

Ejercicio 21

Considerando una estructura de trie como la siguiente:

```
typedef struct trie_t {
        nodo *raiz;
} __attribute__((packed)) trie;

typedef struct nodo_t {
        struct nodo_t *sig;
        struct nodo_t *hijos;
        char c;
        bool fin;
} __attribute__((packed)) nodo;
```



donde `bool` equivale a 1 `byte`. La estructura almacena letras, las mismas están ordenadas lexicográficamente por *nivel* del trie. Un *nivel* está compuesto por los nodos que se encuentran enlazados a través del campo `sig`. Lamentablemente, la implementación que está en producción tiene un error muy difícil de encontrar, en el cual resulta ser que las letras no quedan ordenadas correctamente por nivel.

Se requiere desarrollar una función que dado un trie compruebe que las letras en todos los niveles están ordenadas lexicográficamente. En caso de que encuentre un nivel no ordenado, debe retornar el **doble puntero** al primer nodo del nivel desordenado, en caso contrario devolverá `null`. La aridad de la función será: `nodo** check_trie(trie* t)`

(a) Implementar en `lenguaje C` la función `check_trie`.

(b) Implementar en `ASM` la función `check_trie` utilizando como pseudocódigo la implementación en `lenguaje C` del punto anterior.

Ejercicio 22

Un obstinado desarollador de software está utilizando una compleja biblioteca de funciones desarrollada en un extraño y desconocido lenguaje de programación. Esta biblioteca tiene errores, por lo que el astuto desarrollador planteó un experimento para analizar todo el árbol de llamadas a funciones.

El experimento consta de reemplazar en el código compilado todas las instrucciones `call` x, por la instrucción `call log_call`. La función `log_call` tiene como objetivo loggear datos del llamado antes de hacerlo propiamente. Para ello se encargará de realizar los tres pasos siguientes:

1. Obtener la dirección de la función original (es decir la dirección de x)
2. Almacenar en el log de llamadas datos para el seguimiento
3. Llamar a la función x, como si nunca se hubiera llamado a `log_call`

Para los pasos 1 y 2 se proveen las siguiente funciones que deberán utilizar:

1. `void* get_func(void* addr_call)`
   Retorna el puntero a la función original dada la posición de memoria de la instrucción `call`.
2. `void store_data(void* rsp, void* rbp, void* func)`
   Almacena en el log de llamadas los valores que tenían `rsp` y `rbp` justo antes de la llamada a la función x, y el puntero `func` obtenido mediante la función anterior.

a- Programar en ASM la función `log_call`. Recordar que se debe respetar el estado del programa original en todo momento.

Nota: Considerar que la instrucción `call` ocupa exactamente 10 bytes. Además las funciónes dadas respetan convención C, con la salvedad que no afectan los registos `xmm1` a `xmm15`.

Ejercicio 23

Considerando que al llamar a una función se almacena en la pila la dirección de retorno de la misma y en esta función siempre se almacena en `RBP` la base de la pila, se pide armar una función denominada `superRET` la cual toma un solo parámetro entero. Este parámetro indica la cantidad de llamados a función sobre los que se debe retornar.
Por ejemplo, si se llama a `funcionA`, luego a `funcionB`, luego a `funcionC`, y desde esta última se ejecuta `superRET(2)`, la próxima instrucción a ejecutar será el código perteneciente a `funcionA` que continuaba en su ejecución.

(a) Dibuje un ejemplo de la pila y cómo se almacenan las direcciones de retorno.

(b) Escriba el código ASM de la función `superRET`.
La aridad de la función será: `void superRET(unsigned int n)`