

[Área personal](#) / [Mis cursos](#) / [Licenciatura en Ciencias de la Computación](#) / [Segundo Año](#) / [Arquitectura del Computador](#)
 / [Semana 12 - 1/11](#) / [Parcial 2 - 03/11/2021](#)



Comenzado el	Wednesday, 3 de November de 2021, 08:00
Estado	Finalizado
Finalizado en	Wednesday, 3 de November de 2021, 09:32
Tiempo empleado	1 hora 32 minutos
Puntos	12,4/16,0
Calificación	7,8 de 10,0 (78%)
Comentario -	¡Buen trabajo! El examen está aprobado.

Pregunta 1

Finalizado

Puntúa 2,0 sobre 2,0

Considerar la siguiente función en C:

```
long scale(long x, long y, long z) {
    return x + 6*y + 12*z;
}
```

Cuando se compila, las operaciones aritméticas de la función se implementan mediante una secuencia de cuatro instrucciones **leaq**,:

```
scale:
    leaq    (.....), %rax    # rax = y + 2*y
    leaq    (.....), %rax    # rax = x + 6*y
    leaq    (.....), %rdx    # rdx = z + 2*z
    leaq    (.....), %rax    # rax = x + 6*y + 12*z
    ret
```

Completar con lo que falta en las líneas de punto para que las instrucciones queden coherentes con los comentarios.

```
scale:
    leaq    (%rsi,%rsi,2), %rax    # rax = y + 2*y
    leaq    (%rdi,%rax,2), %rax    # rax = x + 6*y
    leaq    (%rdx,%rdx,2), %rdx    # rdx = z + 2*z
    leaq    (%rax,%rdx,4), %rax    # rax = x + 6*y + 12*z
    ret
```

Comentario:

Pregunta **2**

Finalizado

Puntúa 1,8 sobre 2,0

Dada la siguiente función en C:

```
void cond(long a, long *p)
{
    if (a < *p)
        *p = a;
}
```

Implementar la función anterior en Assembler, explicando el código realizado.

```
.global cond
```

```
cond:
```

```
    cmpq (%rsi), %rdi
```

```
    jz fin
```

```
    js sumar
```

```
fin:
```

```
    ret
```

```
sumar:
```

```
    movq %rdi, (%rsi)
```

```
    jmp fin
```

Comentario:

La idea está bien. Se podría haber simplificado usando jl

Pregunta **3**

Finalizado

Puntúa 1,8 sobre 2,0

Dado el siguiente código en lenguaje C:

```
#include<stdio.h>
long a=1, b=2, c=3, d=4, e=5, f=6, g=7, h=8;
long suma2(long a, long b, long c, long d, long e, long f, long g, long h);

long main(){
    return suma2(a, b, c, d, e, f, g, h);
}

long suma2(long a, long b, long c, long d, long e, long f, long g, long h){
    return b+f+h;
}
```

1. Dibujar el esquema de pila al momento de ingresar a la función **suma2**.
2. Indicar el valor de los registros involucrados. Asumir que **rbp=rsp=0x7fffffff100** al entrar a **main**. Asumir también que la función **suma2** hace uso del registro **rbp**.

Asumiendo que el main llama a suma2 y no a suma

1)

pila

0x7FFFFFFFD100 main

0x7FFFFFFFD0F8 0x00 00 00 00 00 00 00 08

0x7FFFFFFFD0F0 0x00 00 00 00 00 00 00 07

0x7FFFFFFFD0E8 Direccion de retorno (%rsp)

0x7FFFFFFFD0E0

2)

%rdi = 0x1

%rsi = 0x2

%rdx = 0x3

%rcx = 0x4

%r8 = 0x5

%r9 = 0x6

Como se usa %rbp se pushea a la pila

y se llama movq %rsp, %rbp

Comentario:

En la pila falta rbp

Pregunta **4**

Finalizado

Puntúa 1,8 sobre 2,0

Suponga que inicialmente se tienen almacenados los siguientes valores en las direcciones de memoria y registros indicados:

<u>Dirección de memoria</u>	<u>Valor</u>	<u>Registro</u>	<u>Valor</u>
0x1000	0xFF	%rax	0x1000
0x1008	0x8	%rcx	0x1
0x1010	0x16	%rdx	0x4
0x1018	0x16		

Complete la siguiente tabla, indicando dónde se ubica el operando destino y cuál es el valor del mismo luego de ejecutarse cada una las instrucciones en el orden indicado:

<u>Instrucción</u>	<u>Operando destino</u>	<u>Valor</u>
addq %rcx, (%rax)		
subq %rdx, 8(%rax)		
subq %rdx, %rcx		
incq 16(%rax)		
decq %rcx		
imulq %rdx		

<u>Instrucción</u>	<u>Operando destino</u>	<u>Valor</u>
addq %rcx, (%rax)	Direc. de mem: 0x1000	0x100
subq %rdx, 8(%rax)	Direc. de mem: 0x1008	0x4
subq %rdx, %rcx	%rcx	0xFF FF FF FF FF FF FF FD
incq 16(%rax)	Direc. de mem: 0x1010	0x17
decq %rcx	%rcx	0xFF FF FF FF FF FF FF FC
imulq %rdx	%rax	0x4000

Comentario:

En la última instrucción parte del destino está en rdx

Pregunta 5

Finalizado

Puntúa 2,0 sobre 2,0

Dado el siguiente código en Assembler:

```
mov.... %eax, (%rsp)
mov.... (%rax), %dx
mov.... $0xff, %bl
lea.... (%rsp, %rdx, 7), %edx
mov.... (%rdx), %rax
add.... %dx, (%rax)
```

- Determinar el sufijo apropiado basado en los operandos para cada una de las instrucciones. Justificar.
- Para cada una de las líneas de código anteriores indicar si se está haciendo algún acceso a memoria (exceptuando la etapa de búsqueda de la instrucción en la memoria principal (*fetch*)), de qué tipo/s (lectura y/o escritura) y por qué.
- En la instrucción **lea**, indicar si la sintaxis es correcta o no. Explicar.

movl %eax, (%rsp) #aquí se esta haciendo un acceso a memoria de la pila

```
movw (%rax), %dx #aquí se piden 16 bits desde la direccion en la que apunta %rax
movb $0xff, %bl
lea... (%rsp, %rdx, 7), %edx #es invalida la operacion ya que en la componente donde hay un 7, solo pueden ir 1, 2, 4 u 8
movq (%rdx), %rax #se toman 64 bits de donde esta apuntando %rdx y se guardan en %rax
addw %dx, (%rax) #se guarda la suma de lo existente en los 16 bits de memoria siguientes a lo que apunta %rax y %dx, en la direc. de mem
```

a la que apunta %rax

Comentario:

Pregunta **6**

Finalizado

Puntúa 1,0 sobre 2,0

Explicar por qué cada una de las siguientes instrucciones es incorrecta:

```
movb $0xffff, (%ebx)
movl %rax, (%rsp)
```

```
movb $0xffff, (%ebx) #es incorrecta porque %ebx tiene 32 bits, y todas las direcciones de memoria tienen 64 bits
movl %rax, (%rsp) #es una instruccion incorrecta, porque se trabaja con %rax, que es un registro de 64 bits,
```

```
# y movl es una operacion para registros de 32 bits
```

Comentario:

```
movb $0xffff, (%ebx) es incorrecta porque 0xffff tiene más de un byte
```

Pregunta 7

Finalizado

Puntúa 1,0 sobre 1,0

Dado el siguiente programa en C:

```
#include<setjmp.h>
#include<stdio.h>
```

```
jmp_buf buf;
int error1 = 0;
int error2 = 1;
void foo(void), bar(void);
```

```
int main()
{
    switch(setjmp(buf)) {
        case 0:
            foo();
            break;
        case 1:
            printf("Error tipo 1 en foo\n");
            break;
        case 2:
            printf("Error tipo 2 en foo\n");
            break;
        default:
            printf("Error desconocido en foo\n");
    }
    return 0;
}

void foo(void)
{
    if (error1)
        longjmp(buf, 1);
    bar();
}
```

```
void bar(void)
{
    if (error2)
        .....;
}
```

a) Completar la línea de puntos de manera tal que al ejecutarse el programa imprima en pantalla

Error tipo 2 en foo

No usar la instrucción **printf**.

b) Explicar cómo funciona el programa, indicando la secuencia de ejecución.

a)

```
#include<setjmp.h>
#include<stdio.h>
```

```
jmp_buf buf;
int error1 = 0;
int error2 = 1;
void foo(void), bar(void);
```

```
int main()
{
    switch(setjmp(buf)) {
        case 0:
            foo();
            break;
        case 1:
            printf("Error tipo 1 en foo\n");
            break;
        case 2:
            printf("Error tipo 2 en foo\n");
            break;
        default:
            printf("Error desconocido en foo\n");
    }
    return 0;
}

void foo(void)
{
    if (error1)
        longjmp(buf, 1);
    bar();
}
```

```
void bar(void)
{
    if (error2)
        longjmp(buf, 2);
}
```

b)

el programa comienza en main, al ingresar al switch, hace un setjmp para posteriormente saltar a esa línea con longjmp, al hacer la primera llamada de setjmp devuelve 0 por ser la primera llamada de la misma. Ingresa al foo() por el switch, luego no entra al if ya que error1 == 0, al ingresar a bar(), desde foo(), ingresa dentro del if, ya que error2 == 1, y posteriormente realiza un longjmp a la línea previamente seteada con el setjmp, devolviendo el valor que se pasa como segundo argumento, por lo que el switch nos lleva al printf esperado.

Comentario:

Pregunta **8**

Finalizado

Puntúa 0,5 sobre 2,0

Dado el siguiente código en Assembler:

```
.global main
main:
    movq $2, %rcx
    leaq (%rsi,%rcx,8), %rax
    movq (%rax), %rax
    movw (%rax), %ax
    ret
```

Indicar el valor de retorno si se lo ejecuta de la siguiente manera una vez compilado:

```
./prog 1 2 3
```

Explicar además cómo se puede obtener el valor de retorno y por qué se obtiene el valor indicado.

Para poder pasar parametros como linea de comando y no podriamos hacerlo usando gdb, agregue un printf de rax, lo cual devuelve siempre 50.

```
.data
fmt: .asciz "%d\n"
.text
.global main
main:
    movq $2, %rcx
    leaq (%rsi,%rcx,8), %rax
    movq (%rax), %rax
    movw (%rax), %ax
    xorq %rsi, %rsi
    movw %ax, %si
    movq $fmt, %rdi
    xorq %rax, %rax
    call printf
    ret
```

Comentario:

Se pueden pasar parámetros con gdb.

Falta explicación.

Pregunta 9

Finalizado

Puntúa 0,5 sobre 1,0

Dado el siguiente código en Assembler X86-64:

```
movq (%rdi), %rax  
movq %rax, (%rsi)
```

1. Explicar qué hace cada instrucción.
2. Explicar por qué no se puede hacer todo con una sola instrucción.
3. Realizar un esquema que ilustre lo realizado por el código anterior.

1)

```
movq (%rdi), %rax #guarda en %rax el valor al que apunta %rdi  
movq %rax, (%rsi) #guarda el valor de %rax en la direccion a la que apunta %rsi
```

2) No se puede desreferenciar 2 direcciones de memoria en una misma instruccion

3) (%rdi => memoria) => %rax

%rax => (memoria <= %rsi)

Comentario:

Falta especificar el tamaño de los datos. Falta esquema.

◀ Video Clase 29/10/2021 - parte 2

Ir a...

Entrega de ejercicios ►