# Arquitectura de Computadores /Interfaces y Arquitectura Hardware
## SEGUNDO EXÁMEN PARCIAL

| Nombre | Fecha 12 de abril de 2019 |
|---|---|
| Código: | Duración: 110 minutos |

## 1. [40% -compiler 32- multiplicación 64 bits]

Considere el siguiente programa en lenguaje C. El código define una función que realiza la multiplicación de dos enteros con signo de 64 bits (variable long long).

```
1
2  long long mul64(long long , long long );
3
4  int main(){
5
6  mul64(-9,5);
7  return 0;
8  }
9  long long mul64(long long x, long long y)
10 {
11 return x*y;
12 }
13
```

El siguiente código ensamblador es el resultado de la compilación del programa anterior, pero el código está incompleto y con uno o dos instrucciones erróneas. Además, cómo el compilador es X86, supone entonces que no se puede implementar el producto de 64 bits directamente, si no que se debe realizar por medio de varias multiplicaciones de 32 bits. Como verá, además, los dos parámetros de entrada son pasados por la pila, y el resultado de la multiplicación queda en los registros EDX: EAX (EDX la parte alta y en EAX la parte baja). En el programa ensamblador se ha agregado unos comentarios para ayudarle en su análisis, donde:

A_HI, B_HI : hace referencia a los 32 bits más significativos de A y B

A_LO, B_LO : hace referencia a los 32 bits menos significativos de A y B

Se pide entonces:

a) Analice el programa ensamblador y luego explique cómo lograría el programa implementar una multiplicación de dos números enteros de 64 en una plataforma o compilador de 32 bits.(10%)

b) Complete los 7 espacios en blanco de la subrutina _mul64, y los 2 espacios de las líneas 7 y 8, con el fin que el programa lea los parámetros correctamente de la pila teniendo en cuenta el formato little-endian (18%)

c) Identifique y corrija las instrucciones erróneas (puede haber uno o dos errores) (12%)

```
 6    start:
 7       push _____          ;1st pushed: multiplicador B =-9
 8       push _____
 9       push 0              ;2nd pushed: multiplicando A =5
10       push 5
11       call __mul64    ;resultado 64 bits: A*B = EDX:EAX
12       add    esp,8
13       xor eax, eax
14       ret
15    __mul64:
16          mov    eax,[esp  ___  ] ;A_HI
17          mov    ecx,[esp  ___  ]  ;B_HI
18          or     ecx,eax
19          mov    ecx,[esp  ___  ]  ;B_LO
20          jnz    hard
21          mov    eax,[ebp  ___  ]  ;A_LO
22          mul    ecx                ;
23          ret
24    hard: push   ebx
25          mul    ecx                ;A_HI * B_LO ;
26          mov    ebx,eax
27          mov    eax,[esp  ___ ]   ;
28          mul    dword ptr [esp  ___] ;A_LO * B_HI
29          add    ebx,ecx          ;
30          mov    eax,[esp  ___ ]  ;
31          mul    ecx                ;
32          add    edx,ecx          ;
33          pop    ebx
34          ret
35    end start
```

## 2. [60% -Pila y funciones recursivas]

El siguiente programa escrito en lenguaje de alto nivel, calcula la potencia de un número base elevado a un número exponente, esto lo hace aprovechando el principio de recursividad. El programa calcula la potencia de 43, mediante el llamado de la función ***getPower***(int base,int power) que se llama así misma luego de ir disminuyendo la potencia en uno y hasta llegar a 0, y luego debe multiplicar las bases tantas veces fue llamada la función. Entonces 43
=4*4*4=64.

```cpp
#include <stdio.h>

long long int getPower(int base,int power)
{
    long long int result=1;
    if(power==0) return result;
    result=base*(getPower(base,power-1));
}
int main()
{
    long long int result;
    result=getPower(4,3);

    return 0;
}
```

A continuación, aparece el código ensamblador resultado de compilar el programa C++ usando un compilador de 32 bits. Suponga que se establece un punto de interrupción en el retorno de la función _getPower, que detendrá la ejecución justo antes del primer return (línea 57)

```
4   .data
5   _result QWORD ?
6   .code
7   start:
8       push   ebp
9       mov    ebp, esp
10      sub    esp, 8
11      push   3          ;push power
12      push   4          ;push base
13      call   _getPower
14      add    esp, 8
15      pop    DWORD PTR [_result]
16      pop    DWORD PTR [_result+4]
17      mov    esp, ebp
18      pop    ebp
19      xor    eax, eax
20      ret
```

```
22  _getPower PROC
23          push   ebp
24          mov    ebp, esp
25          sub    esp, 8
26          push   esi
27          push   edi
28          mov    DWORD PTR [ebp -8 ], 1      ;result
29          mov    DWORD PTR [ebp -4], 0       ;result
30          cmp    DWORD PTR [ebp +12  ], 0
31          jne    $LN2
32          mov    eax, DWORD PTR [ebp -8]
33          mov    edx, DWORD PTR [ebp -4]
34          jmp    $LN1
35  $LN2:   mov    eax, DWORD PTR [ebp +8]     ;base
36          cdq                                ; // EDX:EAX = sign-extend of EAX
37          mov    esi, eax
38          mov    edi, edx
39          mov    eax, DWORD PTR [ebp +12]    ;power
40          sub    eax, 1
41          push   eax
42          mov    ecx, DWORD PTR [ebp+8]      ;base
43          push   ecx
44          call   _getPower
45          add    esp, 8
46          push   edx
47          push   eax
48          push   edi
49          push   esi
50          call   __mul64
51          mov    DWORD PTR [ebp -8], eax
52          mov    DWORD PTR [ebp -4], edx
53  $LN1:   pop    edi
54          pop    esi
55          mov    esp, ebp
56          pop    ebp
57          ret            ;//_getPower ENDP
```

**2.a** Realice un análisis del comportamiento de la pila hasta el primer breakpoint, para ello diligencie la tabla 1 que tiene una plantilla de las direcciones del segmento de pila. En dicha tabla vaya consignando los valores que se van almacenando en la pila. En la columna de descripción escriba el número de línea o las líneas del código que accede a dicha posición de memoria y la instrucción que se ejecutó y cambio o accedió a la pila, para las direcciones de retorno solo indique el número

de la línea de código, *ret L10* por ejemplo. Además, debe indicar en la tabla 2 el contenido de los registros que allí de piden en el momento del primer breakpoint. Asuma que los valores de esp y ebp cuando apenas inicia el programa es 19FF80H paraambos registrosd.

**2.b** Al parecer el programa no funciona bien, se pide entonces que indique donde esta la falla y realice las correcciones.

Tabla 1

| Dirección | Contenido | Descripción |
|---|---|---|
| 19FF74 | | |
| 19FF78 | | |
| 19FF7C | | |
| 19FF80 | | |
| 19FF84 | | |
| 19FF88 | | |
| 19FF8C | | |
| 19FF90 | | |
| 19FF94 | | |
| 19FF98 | | |
| 19FF9C | | |
| 19FFA0 | | |
| 19FFA4 | | |
| 19FFA8 | | |
| 19FFAC | | |
| 19FFB0 | | |
| 19FFB4 | | |
| 19FFB8 | | |
| 19FFBC | | |
| 19FFC0 | | |
| 19FFC4 | | |
| 19FFC8 | | |
| 19FFCC | | |
| 19FFD0 | | |
| 19FFD4 | | |
| 19FFD8 | | |
| 19FFDC | | |
| 19FFE0 | | |
| 19FFE4 | | |
| 19FFE8 | | |
| 19FFEC | | |
| 19FFF0 | | |
| 19FFF4 | | |
| 19FFF8 | | |
| 19FFFC | | |
| 1A0000 | | |
| 1A0004 | | |
| 1A0008 | | |

| Registro | Valor Inicial | Valor en el breakpoint |
|---|---|---|
| ESP | 19FF80 | |
| EPB | 19FF80 | |
| EAX | ABCDEF | |
| EDX | C23456 | |

*Tabla 2*

## Anexo 1.

*Lo mismo es válido para la instrucción IMUL

| IMUL: Signed Multiply | | | |
|---|---|---|---|
| **Instruction** | **64-Bit Mode** | **32-Bit Mode** | **Description** |
| MUL *r/m8* | Valid | Valid | AX← AL ∗ *r/m* byte. |
| MUL *r/m16* | Valid | Valid | DX:AX ← AX ∗ *r/m* word. |
| MUL *r/m32* | Valid | Valid | EDX:EAX ← EAX ∗ *r/m*32. |
| MUL *r/m64* | Valid | N.E. | RDX:RAX ← RAX ∗ *r/m*64. |
| MUL *r16, r/m16* | Valid | Valid | word register ← word register ∗ *r/m*16. |
| MUL *r32, r/m32* | Valid | Valid | doubleword register ← doubleword register ∗ *r/m32.* |
| MUL *r64, r/m64* | Valid | N.E. | Quadword register ← Quadword register ∗ *r/m64*. |
| CDQ | Valid | Valid | Convert Doubleword to Quadword EDX:EAX = sign-extend of EAX |