Arquitectura de Computadores – IIC2343

I2

13 octubre 2017

1. Considera la siguiente función **`factorial`**, que para un número entero *n* dado calcula *n*!, definida recursivamente en un (pseudo)lenguaje de alto nivel:

```
int factorial(int n):
   if n == 0:
      return 1
   else:
      return n*factorial(n-1)
```

a) [**2**] Escribe el código correspondiente en el lenguaje *assembly* x86 (286, 386 o Core i7).

b) [**1**] Escribe en el mismo (pseudo)lenguaje de alto nivel una versión iterativa (en vez de recursiva) de la función **`factorial`**.

c) [**1**] Escribe el código en lenguaje *assembly* x86 correspondiente a tu función iterativa de b).

d) [**2**] Se suele decir que las versión iterativa de un algoritmo es más eficiente que la correspondiente versión recursiva. Explica claramente por qué esto es cierto en el caso de la función **`factorial`**; o bien explica claramente por qué no lo es. Haz referencia a los códigos de tus respuestas a a) y c).

Respuesta.

a) Aparece en la p. 12 de los apuntes "7-Arquitectura x86".

b)

```
int factorial(int n):
   f = 1
   for i = 2 ... n:     —si n es 0 o 1, este loop simplemente no se ejecuta
      f = f*i:
   return f
```

c) [falta]

d) Es cierto, ya que tanto el código para a) como el código para c) incluyen las *n* multiplicaciones, pero el código para c) no incluye las instrucciones para la llamada recursiva (que se ejecuta *n* veces), incluyendo el manejo del stack: PUSH, CALL, POP.

2. Considera el siguiente programa escrito en un (pseudo)lenguaje de alto nivel, consistente en una parte principal, **`main`**, y dos funciones (o subrutinas), **`f`** y **`g`**:

```
10 main:
14     int a = 9
18     int b = 6
22     int c = f(a, b)
26     print(c)
```

```
30 int f(int p, int q):
34     if q == 0:
38         return p
42     int r = p%q
46     return g(q,r)
```

```
50 int g(int x, int y):
54     if x == y:
58         return x
62     if x > y:
66         int z = x-y
70         return f(z, y)
74     int v = y-x
78     return f(x, v)
```

Muestra el contenido del stack de *frames* (o registros de activación) a medida que se ejecuta el programa; en particular, el *frame pointer* (o *base pointer*), el *stack pointer*, la dirección de retorno, los parámetros y las variables locales, en cada uno de los siguientes momentos:

a) Cuando **`f`** es llamada por primera vez desde **`main`** y está a punto de ejecutar **`if q == 0`**.

b) Cuando **`g`** es llamada por primera vez desde **`f`** y está a punto de ejecutar **`if x == y`**.

c) Cuando **`f`** es llamada por primera vez desde **`g`** y está a punto de ejecutar **`if q == 0`**.

Respuesta.

Esta es la versión "exhaustiva". Sin embargo, la respuesta correcta es más simple. Interesa principalmente saber si tienen claro el concepto del uso de un stack para implementar llamadas de funciones. Está basada en el ejemplo de las torres de Hanoi incluido en el archivo ISA.pptx (diaps. 48-50). Por lo tanto, no es necesario pasarse a código x86.

a) Antes de llamar a f, el se usaron 2 PUSHs de 2 bytes cada uno y además se llamó a una subrutina por lo tanto esto equivale a que el SP haya quedado en -6, luego se hace otro PUSH (-8 queda el SP con esto) al BP y luego se le pone el valor de SP a BP y luego al SP se le resta -2 para ponerle una variable local. En resumen, el Stack queda así:

SP->	246	Variable Local r	Basura
	247	Variable Local r	
BP->	248	Valor de BP	Basura
	249	Valor de BP	
	250	Dirección de Retorno	26
	251	Dirección de retorno	
	252	Parámetro q	6
	253	Parámetro q	
	254	Parámetro p	9
	255	Parámetro p	

La dirección de retorno quedaría justo después del llamado a f, esta corresponde a la ubicación de MOV varC,AX , que en nuestro código sería justo cuando llamamos a print(c), si nos guiamos por los números del código original, la dirección de retorno debiera ser 26.
Los parámetros de la función f serian los números a y b, por lo tanto serian 9 y 6 y estarían posicionados en el stack entre la posición 252 y 255.

La única variable local utilizada es r y variable estaría almacenada en el stack en las ubicaciones 246 y 247. No se conoce su valor ya que aún no es inicializada por lo que tiene basura.

En resumen, sería lo siguiente:

Dato	Valor	Ubicación en el stack
BP	248	Almacena su valor anterior en 248 y 249
SP	246	---
Dirección de retorno	26	250 y 251
Parámetro p	9	254 y 255
Parámetro q	6	252 y 253
Variable local r	Basura	246 y 247

b) Cuando g es llamado por primera vez, como vimos en a), el SP está posicionado en 246 y luego se le hacen 2 PUSH de 2 bytes cada uno para introducir los parámetros por lo que baja hasta 242, luego se llama a g por lo que el SP baja hasta 240 para darle espacio a la dirección de retorno. Luego cuando se llama g lo primero que se hace es un PUSH de 2 bytes (por lo que queda en 238) y luego BP pasa a tener el valor de SP y luego se le resta 4 a SP para darle espacio a las variables locales z y v por lo que SP queda en la posición 234.

En resumen, quedaría el Stack de esta manera:

Pertenecen a la función g	SP->	234	Variable Local v	Basura
		235	Variable Local v	
		236	Variable Local z	Basura
		237	Variable Local z	
	BP->	238	Valor de BP	248
		239	Valor de BP	
		240	Dirección de Retorno	46~50
		241	Dirección de Retorno	
		242	Parámetro y	3
		243	Parámetro y	
		244	Parámetro x	6
		245	Parámetro x	
Pertenecen a la función f		246	Variable Local r	3
		247	Variable Local r	
		248	Valor de BP	Basura
		249	Valor de BP	
		250	Dirección de Retorno	26
		251	Dirección de retorno	
		252	Parámetro q	6
		253	Parámetro q	
		254	Parámetro p	9
		255	Parámetro p	

La dirección de retorno ahora sería justo después del llamado a g y si nos guiamos por el código inicial seria la posición 46~50. Los parámetros corresponderían al valor de q y r en el llamado de f por lo que serían 6 y 9%6 = 3, el valor de BP seria 238 y las variables locales z y v aun no tendrían un valor por lo que tendrían basura.

En resumen, sería lo siguiente:

Dato	Valor	Ubicación en el stack
BP	238	Almacena su valor anterior en 238 y 239 (valor anterior 248)
SP	234	---
Dirección de retorno	46~50	240 y 241
Parámetro y	3	244 y 245
Parámetro x	6	242 y 243
Variable local z	Basura	236 y 237
Variable local v	Basura	234 y 235

c) Ya que en la función g, el parámetro x era 6 y el parámetro y era 3 y 6 es mayor a 3 entonces entra la funcion al segundo if. Por lo que f será llamado con los parámetros z = 6-3 = 3 y parámetro y que es 3.

Al momento de llamar a f, el SP esta en 234 se hacen primero 2 PUSHs por lo que queda en 230, luego se llama a f con el call por lo que SP queda en 228 y luego se hace un PUSH para el BP por lo que SP queda en 226, el BP queda con el mismo valor que SP y luego al SP se le resta 2 para darle un espacio a la variable local r quedando así en 224. Por lo que quedaría así:

Pertenecen a la función f	SP->	224	Variable Local r	Basura
		225	Variable Local r	
	BP->	226	Valor de BP	238
		227	Valor de BP	
		228	Dirección de Retorno	66~70
		229	Dirección de retorno	
		230	Parámetro q	3
		231	Parámetro q	
		232	Parámetro p	3
		233	Parámetro p	
Pertenecen a la función g		234	Variable Local v	Basura
		235	Variable Local v	
		236	Variable Local z	3
		237	Variable Local z	
		238	Valor de BP	248
		239	Valor de BP	
		240	Dirección de Retorno	46~50
		241	Dirección de Retorno	
		242	Parámetro y	3
		243	Parámetro y	
		244	Parámetro x	6
		245	Parámetro x	
Pe		246	Variable Local r	3

		247	Variable Local r	
		248	Valor de BP	Basura
		249	Valor de BP	
		250	Dirección de Retorno	26
		251	Dirección de retorno	
		252	Parámetro q	6
		253	Parámetro q	
		254	Parámetro p	9
		255	Parámetro p	

Como se dijo antes el parámetro x e y tendrían el valor 3.
La dirección de retorno no quedaría clara ya que en ASM no puedes hacer retornar una función, sino que debes hacerlo por separado, pero sería un numero entre 66 y 70. La variable local r no han sido inicializada aun por lo que contienen basura.

3. Con respecto a cada uno de los siguientes modos de direccionamiento: ***inmediato***, ***directo***, ***por registro***, ***indirecto por registro***, e ***indexado***, responde lo siguiente:

Suponiendo que hay un arreglo de n números enteros en memoria, ¿qué dificultades y/o qué facilidades te entrega cada modo de direccionamiento para poder sumar los n números? Sé preciso en tus respuestas. Considera que cuentas con las instrucciones del lenguaje assembly x86 similares a la del computador básico.

Respuesta.

D. inmediato: El argumento de la instrucción no es la dirección del dato sino el dato propiamente tal. No hay cómo hacer un código general que sume n números: es necesario escribir n instrucciones SUM, cada una acompañada por uno de los n números. Dos ejecuciones del programa suman los mismos n números.

D. directo: El argumento de la instrucción es la dirección que el dato ocupa en memoria; pero esta dirección es fija, por lo que hay que conocerla de antemano y especificarla explícitamente para cada dato. También son necesarias n instrucciones SUM, cada una acompañada por la dirección en memoria de uno de los números. Dos ejecuciones del programa pueden sumar dos conjuntos distintos de n números.

D. por registro: El argumento de la instrucción es el registro en que está el dato. Es necesario ir cargando en el registro cada uno de los n números … pero ¿cómo? Se necesita otro tipo de direccionamiento para instrucciones LOAD, en cuyo caso (tal vez) mejor usamos este otro direccionamiento para la suma.

D. indirecto por registro: El argumento de la instrucción es el registro en el que está la dirección de memoria del dato; puede hacer referencia a diferentes direcciones de memoria en cada ejecución de la instrucción (guardando una dirección diferente en el registro cada vez), lo que permite recorrer fácilmente el arreglo de números.

D. indexado: El argumento de la instrucción está compuesto de dos partes: una dirección de memoria constante (p.ej., la del primer dato del arreglo) y un registro; la dirección especificada por el argumento es la suma de la dirección constante más el valor almacenado en el registro, que puede variar en cada ejecución de la instrucción, permitiendo recorrer fácilmente el arreglo de números.