Desarrollo Práctica 7 - 8

Abad L. Freddy L., Aguilar Y. Bryan A.

Escuela de Informática, Facultad de Ingeniería, Universidad de Cuenca, Ecuador
ORGANIZACIÓN Y ARQUITECTURA DE COMPUTADORES
freddy.abadl@ucuenca.edu.ec, bryan.aguilar@ucuenca.edu.ec

Práctica 7: Otros bucles condicionales

Objetivos:

- Cómo implementar en ensamblador las funciones C strlen y strcpy (ejemplos de cómo hacer en ensamblador los bucles condicionales while y do-while, respectivamente).

Desarrollo:

- Sea el siguiente código C:

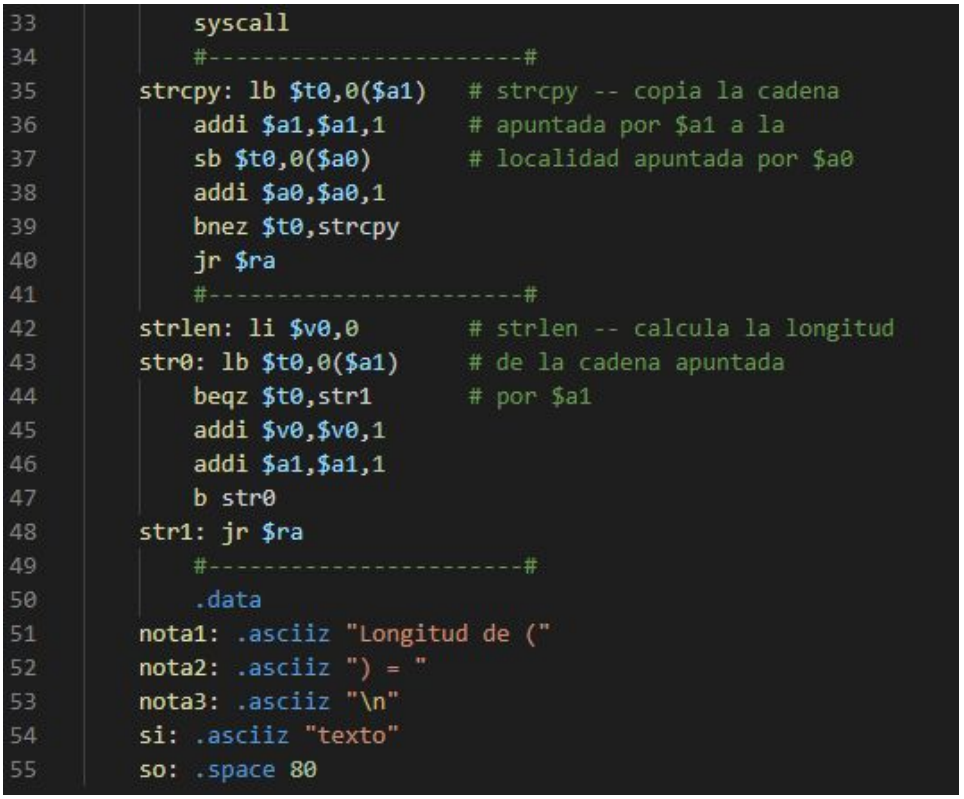
```
1   #include <iostream>
2   using namespace std;
3
4   void strcpy(char *dest, char *src){
5       do
6           *dest++ = *src++;
7       while (*src);
8   }
9   int strlen(char *s){
10      int n = 0;
11      while (*s){
12          n++;
13          s++;
14      }
15      return n;
16  }
17
18  int main(){
19      char si[] = "texto";
20      char so[80];
21      cout << strlen(si);
22      strcpy(so,si);
23      cout << so;
24      return 0;
25  }
```

- La función strcpy copia una cadena en otra posición de memoria (que es lo mismo que crear una cadena idéntica). La función strlen cuenta el número de caracteres que tiene una cadena.

- La función main declara la cadena “texto”, y reserva un buffer (memoria) de 80 bytes (80 caracteres) para que podamos copiar a él la cadena “texto”. Después, cuenta los caracteres de la cadena “texto” para, a continuación, copiarla sobre el buffer.
- El código ensamblador que hace las tareas del código C es:

Users ▸ Usuario ▸ Desktop ▸ Practica7S.asm

```
1   # PRAC_7.S
2           .text
3           .globl main
4   main:   la $a1,si           # $a1 tiene si = dirección base
5                               # de la cadena "texto"
6           jal strlen          # Devuelve en $v0 la longitud de
7                               # la cadena que hay en $a1
8           move $t0,$v0        # Imprimo información de la longitud
9           la $a0,nota1
10          li $v0,4
11          syscall
12          la $a0,si
13          li $v0,4
14          syscall
15          la $a0,nota2
16          li $v0,4
17          syscall
18          move $a0,$t0
19          li $v0,1
20          syscall
21          la $a0,nota3
22          li $v0,4
23          syscall
24          la $a1, si          # $a1 tiene si = dirección base
25                              # de la cadena "texto"
26          la $a0, so          # $a0 tiene so = dirección base
27                              # para la futura cadena
28          jal strcpy          # Copio en so la cadena que hay en si
29          la $a0, so          # Consola muestra la cadena
30          li $v0, 4           # que hay en so
31          syscall
32          li $v0, 10          # Fin del programa
```



```
33          syscall
34          #----------------------#
35      strcpy: lb $t0,0($a1)   # strcpy -- copia la cadena
36          addi $a1,$a1,1      # apuntada por $a1 a la
37          sb $t0,0($a0)       # localidad apuntada por $a0
38          addi $a0,$a0,1
39          bnez $t0,strcpy
40          jr $ra
41          #----------------------#
42      strlen: li $v0,0        # strlen -- calcula la longitud
43      str0: lb $t0,0($a1)     # de la cadena apuntada
44          beqz $t0,str1       # por $a1
45          addi $v0,$v0,1
46          addi $a1,$a1,1
47          b str0
48      str1: jr $ra
49          #----------------------#
50          .data
51      nota1: .asciiz "Longitud de ("
52      nota2: .asciiz ") = "
53      nota3: .asciiz "\n"
54      si: .asciiz "texto"
55      so: .space 80
```

- Estudiar el uso de la directiva .space
- Estudiar en profundidad cómo se implementan strcpy y strlen en ensamblador, observando cómo se construyen los bucles condicionales while y do-while.
- Observar que, para imprimir los resultados en consola, y dado que la cadena “texto” no termina con un retorno de carro, hemos puesto unas cadenas adicionales para que la presentación por consola sea más ordenada.

Práctica 8: Pilas y rutinas recursivas.

Objetivos:

- Llamada a una rutina recursiva.
- Creación y uso de marcos de pila.
- Cómo guardar y restaurar registros.
- Cómo usar el convenio guardar-invocador.

Desarrollo:

Fundamentos teóricos:

Estudiar detalladamente las transparencias nº 14, 15, 16, 17, 18, 19 , 20 y 21 del tutorial. Parte de la información ahí descrita ya la conocemos, tanto por la teoría de la asignatura como por el desarrollo de anteriores prácticas (por ejemplo, palabras de memoria, segmentos de datos y texto, etc.).

Ahora vemos aspectos más detallados como el uso de pilas de datos, formas de introducir y sacar datos de pila, los convenios de llamadas a procedimientos, etc. Sea el siguiente código ensamblador:

```
# PRAC_8.S
        .text
        .globl main
main:   #-----------------------#
        # (0)
        #-----------------------#
        subu $sp,$sp,32 # La rutina main crea su marco de pila
        sw $ra,20($sp) # Mem[$sp+20]=$ra. Guardo direcc. de vuelta
        sw $fp,16($sp) # Mem[$sp+16]=$fp. Guardo $fp antiguo.
        # Estas posiciones de memoria van separadas
        # por 4 Bytes (estamos almacenando palabras)
        addu $fp,$sp,32 # $fp=$sp+32. $fp apunta al comienzo del
        # marco de pila (donde estaba $sp antes)
        #-----------------------#
        # (1)
        #-----------------------#
        li $a0,3 # Pongo argumento (n=3) en $a0
        jal fact # Llama fact, almacena en $ra dir. sig. inst.
        #-----------------------#
        move $a0,$v0 # En $v0 está el resultado. Lo imprimo
        li $v0,1 # en la consola.
        syscall
        #-----------------------#
        lw $ra,20($sp) # Restauro registros
        lw $fp,16($sp)
        addu $sp,$sp,32
        #-----------------------#
        # (10)
        #-----------------------#
        j $ra
```

```
31          fact:   #======================# # Rutina 'fact'
32                  subu $sp,$sp,32 # Crea marco de pila
33                  sw $ra,20($sp)
34                  sw $fp,16($sp)
35                  addu $fp,$sp,32
36                  sw $a0,0($fp) # Guardo argumento $a0 en marco de pila (n)
37                  #----------------------#
38                  # (2),(3),(4),(5)
39                  #----------------------#
40                  lw $v0,0($fp)
41                  bgtz $v0,$L2
42                  li $v0,1
43                  j $L1
44          $L2:    lw $v1,0($fp)
45                  subu $v0,$v1,1
46                  move $a0,$v0
47                  jal fact
48                  lw $v1,0($fp)
49                  mul $v0,$v0,$v1
50          $L1:    lw $ra,20($sp)
51                  lw $fp,16($sp)
52                  addu $sp,$sp,32
53                  #----------------------#
54                  # (6),(7),(8),(9)
55                  #----------------------#
56                  j $ra
57                  #----------------------#
```

Este programa calcula, mediante la rutina fact, el factorial de un número n. Tomaremos n=3, de forma que el resultado (que imprimimos por consola), será 6. La rutina fact está implementada de forma recursiva, o sea, es una rutina que se llama a sí misma.

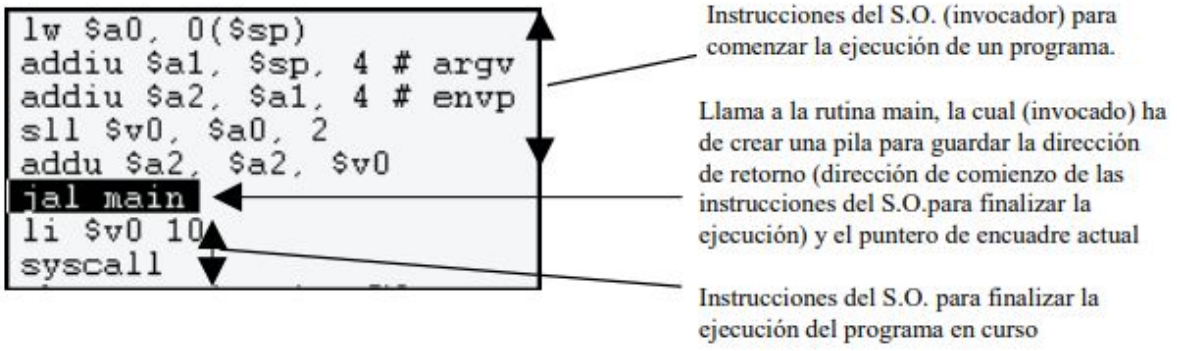
Por tanto, ha de guardar en memoria (en una pila) sus valores actuales puesto que, al llamarse a sí misma, esos valores serán alterados, y los necesitaremos a la vuelta. Para explicar las operaciones de este programa, nos ayudaremos del dibujo en el que se presenta la memoria y algunos registros.

En él (y en el código ensamblador), rotulamos con los indicadores (0), (1), ..., (10) ciertos momentos en la ejecución del programa, con el objeto de comprobar la situación de la memoria y de los registros justo cuando se ha alcanzado este punto.

La ejecución transcurre de la siguiente forma:

Justo antes de comenzar la ejecución de la función main, la situación de la memoria y los registros es la descrita en (0). Para que el programa "arranque", hay una serie de instrucciones suministradas por el sistema operativo (invocador), que se ejecutan, y que producen que en (0) la situación de los registros sea la descrita.

La rutina main (invocado) comienza creando un marco de pila para guardar la dirección de vuelta y el puntero de encuadre, que, al ser recuperados al final del programa, nos llevarán de nuevo al invocador, es decir, a las instrucciones de fin del programa:



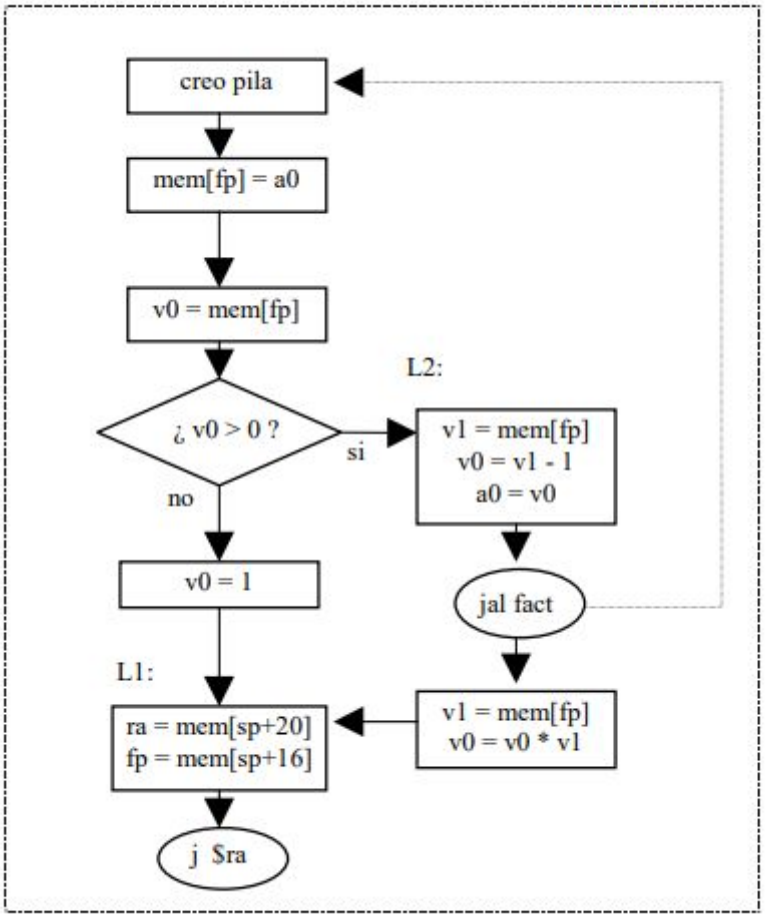
- El tamaño mínimo de una estructura de pila debe ser de 32 bytes (que es bastante más grande de lo que necesitamos para nuestros propósitos).
- Después, main llama a la rutina fact (habiendo puesto en $a0, previamente, el argumento a pasarle a fact, y que es n=3). Cuando fact termine su ejecución, devolverá (en $v0) el resultado, que main va a imprimir por consola.
- Una vez imprimido el resultado, hay que restaurar los valores que main guardó en pila (dirección de vuelta y puntero del marco de pila), y debe hacer que el puntero de pila apunte a la posición primitiva (lo que equivale a "borrar" la pila de main). Así, y mediante la instrucción j $ra, podemos volver a la posición donde se llamó a main.
- En esta posición están las instrucciones necesarias para terminar la ejecución del programa, devolviendo el control al S.O.

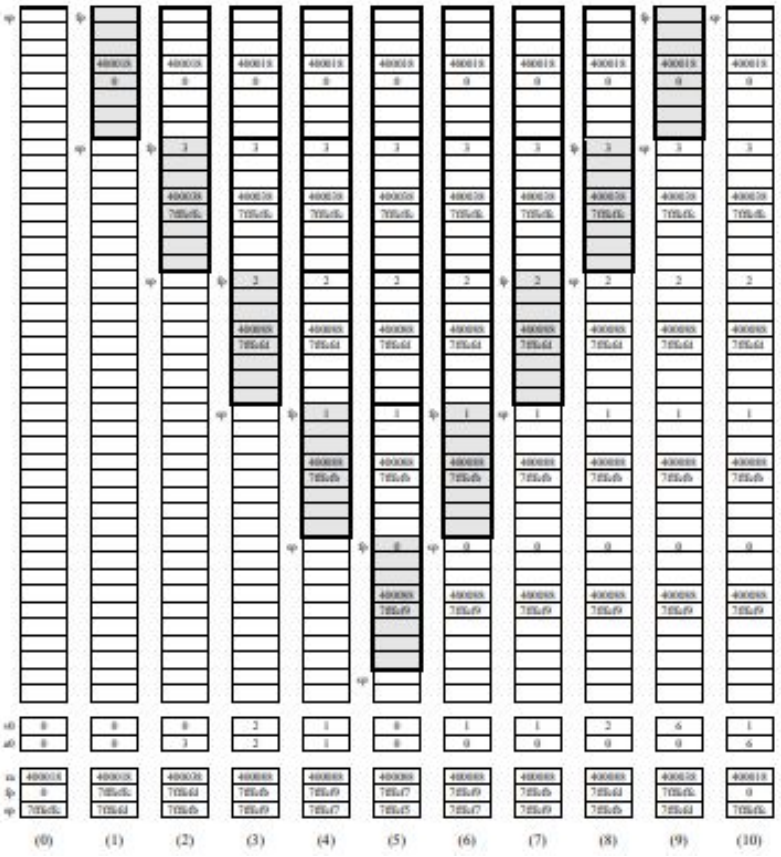
Vamos ahora a estudiar cómo funciona la rutina fact.

- ➔ Cómo fact es una rutina que se llama a sí misma, actúa como invocador e invocado al mismo tiempo, por lo que puede que nos interese guardar en pila el valor de algún registro (en nuestro caso, $a0) durante las sucesivas llamadas. Usaremos, entonces, el convenio de "guardar-invocador". : Ejecutamos por primera vez fact, al ser llamada por main.
- ➔ Al principio de fact hay que crear un marco de pila, en el que guardaremos la dirección de retorno y el puntero del marco de pila del invocador.
- ➔ Enseguida, fact va a actuar como invocador de fact. Cómo usamos el convenio "guardar invocador", hemos de guardar en esta pila, antes de llamar al invocado mediante "jal fact", el valor de $a0 (argumento n). En este momento estamos en el rótulo (2).
- ➔ Tras unas operaciones condicionales, llamamos al invocado.
- ➔ El proceso se repite pasando por los rótulos (3), (4) y (5). Como ya no vamos a volver a llamar a fact, tenemos que ir retornando a los sucesivos invocadores. Para ello, primero realizaremos las operaciones con los valores que hemos ido guardando en las pilas para calcular el factorial; después, restauraremos los registros con las direcciones de vuelta y los punteros de encuadre, y "borraremos" las pilas mediante la suma del puntero de pila. Esto está reflejado en los rótulos (6), (7), (8) y (9).

En el siguiente esquema se puede ver cómo trabaja la rutina fact :

Rutina **fact**


creo pila
mem[fp] = a0
v0 = mem[fp]
L2:
¿ v0 > 0 ?
si
v1 = mem[fp]
v0 = v1 - 1
a0 = v0
no
v0 = 1
jal fact
L1:
ra = mem[sp+20]
fp = mem[sp+16]
v1 = mem[fp]
v0 = v0 * v1
j $ra


(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
(8)
(9)
(10)

Propuesta. Se ha calculado el factorial de un número de esta forma (un tanto complicada) para ilustrar el uso de pilas y, sobre todo, las funciones recursivas. Hay formas mucho más sencillas de calcular el factorial de un número sin necesidad de llamadas a funciones recursivas, pilas, etc. *Elaborar un código ensamblador sencillo que lo realice.*
La elaboración de un **factorial iterativo**, agiliza la escritura de código. El código resultante se nota más corto y entendible que un factorial recursivo.
Cabecera de documento ASM:

```
.data
    prmpt: .asciiz "Ingrese un Entero Positivo: "
    nl:     .asciiz "\n"
            .align 2
    name:   .asciiz "\t\tPractica 9\nFactorial Iterativo\n\n"
            .align 2
    lomsg:  .asciiz "! es :"
            .align 2
    space:  .asciiz "   "
            .align 2
.text
            .globl  main
```

En esta se describe los mensajes a mostrarse en el algoritmo iterativo de factorial

```
main:
        #li     $v0,4
        #la     $a0, prmpt
        #syscall
        #li     $v0,5
        #syscall
        move    $t0,$v0
        li      $a3,9
        li      $t0,1
        la      $a0,name        #las llamadas al sistema usan a0 para argumento, y v0 para que el valor devuelto pase al sistema
        li      $v0,4
        syscall
        move    $a0,$a3
        li      $v0,1
        syscall
        ble     $a3,1,print
```

En esta sección se define el main, donde se procede a integrar un número al cual se procesa para obtener su factorial, para agilizar este proceso se ingresa directamente el número 9. Además de definir las instrucciones que se utilizaran .

```
33  loop:   mult    $t0,$a3
34          mflo    $t0
35          mfhi    $t2
36
37          mult    $t1,$a3
38          mflo    $t3
39          add     $t1,$t2,$t3
40          addiu   $a3,-1
41          bge     $a3,2,loop
```

En esta sección se integra un bucle for, para multiplicar los enteros menores al entero ingresado inicialmente, se utilizan directorios temporales para no superponer las multiplicaciones. La instrucción **mflo** conserva los valores de retorno de la multiplicación antigua para calcular el desbordamiento. La instrucción **addiu** permite el decremento del número para la siguiente iteración, y la última instrucción **bge** permite validar la condición para el bucle, al ser mayor a 2 se repite la función, caso contrario sigue el flujo del program.

```
print:
        la      $a0,lomsg
        li      $v0,4
        syscall
        move    $a0,$t0
        li      $v0,1
        syscall

Exit:   li      $v0,10
        syscall
```

Finalmente se registra el número resultante del factorial

```
                                Practica 9
Factorial Iterativo

9! es :362880
```