Práctica 3: Programación con subrutinas

3.1 Objetivos

El objetivo de esta práctica es estudiar el soporte del ensamblador del ARM para la gestión de subrutinas, para lo que resulta necesario familiarizarse con:

- Instrucciones de llamada y retorno de subrutinas.
- Convenio de paso de parámetros entre rutinas.
- Manejo de la pila para almacenar/restaurar contexto (prólogo/epílogo).

3.2 Subrutinas en Ensamblador

Las subrutinas (funciones o procedimientos) son abstracciones que se usan en los lenguajes de alto nivel para simplificar el código y poder reusarlo. A la subrutina se la llama (invoca) desde el programa *invocante* mediante una instrucción particular. Pero antes de llamar a la subrutina tenemos que haber colocado los datos de entrada a ésta en un lugar accesible (registros reservados o pila). En general al trabajar con subrutinas se divide el trabajo a realizar entre el programa invocante y la subrutina:

1. *Programa invocante*: Poner los parámetros o argumentos de entrada a la subrutina en un lugar donde sean accesibles por ésta.
   - Registros para pasar parámetros r0 – r3
     Ej.: `mov r0, r7`

2. *Programa invocante:* Transferir el control a la subrutina
   Ej.: `bl Etiqueta  @lr=pc+4 & pc=posición de Etiqueta`

3. *Subrutina:* Ejecutar la tarea deseada usando los recursos necesarios

4. *Subrutina:* Poner el resultado en un lugar accesible al Programa
   - En el ARM la subrutina devuelve un único parámetro de salida con el resultado. En nuestro caso, vamos a trabajar siempre con números enteros con lo que el resultado siempre se devolverá únicamente en r0, pero si el resultado ocupase más de 4 bytes se usarían también los otros registros del r1 al r3.
     Ej.: `mov r0, r7`

5. *Subrutina:* Devolver el control al punto inicial, manteniendo el contexto
   Ej.: `mov pc, lr`

   También se puede hacer:
   `bx lr`

En las subrutinas en ARM se sigue el siguiente convenio:

- Registros para pasar parámetros a las subrutinas: `r0-r3`
- Registros para devolver valores al programa invocante: `r0-r3`
- Dirección de retorno al punto de origen: `lr (equivalente a r14)`.
- El procesador ARM incluye una instrucción específica para implementar subrutinas, branch&link, cuya sintaxis es: `bl Etiqueta` Esta instrucción guarda

en el registro `lr` la dirección de la instrucción siguiente a la propia `bl Etiqueta` (`r14=pc+4`) y salta a la posición de memoria indicada en la etiqueta (`pc=dirección de memoria correspondiente a Etiqueta`).

- Guardar la dirección de retorno en `lr` permite a la subrutina devolver el control a la rutina invocante mediante un salto al contenido de `lr`, lo que conseguiremos mediante el salto de retorno de subrutina: `mov pc, lr`

Nótese que los registros usados para almacenar los resultados de una subrutina son los mismos que los utilizados para el paso de parámetros y no se preservan entre llamadas (r0 – r3). En consecuencia no está garantizado que al retorno de una subrutina r0 –r 3 conserven el valor que tenían antes de la llamada. Por el contrario toda subrutina debe garantizar que el contenido de los registros r4 – r11 conserva el valor original al retorno de la subrutina.

3.2.1 Paso de Parámetros

En algunas situaciones, para ejecutar las instrucciones de una subrutina es necesario usar más registros que los argumentos y los resultados (`r0-r3`). En este caso, debemos mantener el valor de los registros que está usando el Programa (especialmente `r4-r11`). Para ello utilizamos la pila, que es la región de memoria apuntada por el registro `sp`. En el ARM este registro se inicializa al valor 0x0c200000, para ello debemos añadir al principio del código principal la instrucción

```
MOV sp,#0x0c200000
```

Es importante notar que la pila "crece hacia abajo" es decir a direcciones menores de memoria. Así si queremos reservar espacio para 3 elementos de tamaño palabra debemos restar 12 (3 elementos * 4 bytes) posiciones de memoria. El esquema general para el ARM es, por tanto:

El programa invocante necesitará la pila si:

- Necesita pasar más parámetros que los contenidos en r0 a r3.
- Desea preservar el valor original de los parámetros de entrada a la vuelta de la subrutina.

La subrutina necesitará la pila si:

- Necesita leer más parámetros de entrada que los contenidos en r0 a r3.
- Va a utilizar los registros de r4 a r11.

En esta práctica vamos a asumir que el número de parámetros de entrada a la subrutina es menor o igual que cuatro.

Como resultado a la hora de programar en ensamblador se deben seguir los siguientes pasos:

1. El Programa **invocante** mueve los registros (los comprendidos entre r0 y r3) a la pila con `str`. Sólo se moverán aquellos registros de los que se necesite mantener su valor a la vuelta de la subrutina ya que los registros de r0 a r3 se utilizan para pasar y devolver parámetros. En caso de que esto no sea necesario el paso 1 puede suprimirse.

El siguiente código permitiría salvar en la pila los registros r0-r3. En cada caso el código debe adaptarse en función de los registros que realmente se deseen salvar.

```
sub      sp, sp, #16      @ reservamos espacio para 16 bytes
str      r0, [sp,#12]     @ salvamos r0 en la pila
str      r1, [sp,#8]      @ salvamos r1 en la pila
str      r2, [sp,#4]      @ salvamos r2 en la pila
str      r3, [sp,#0]      @ salvamos r3 en la pila
```

2. El programa invocante mueve los parámetros de entrada a la subrutina a los registros r0 – r3.

3. El Programa **invocante** pasa el control a la subrutina mediante la instrucción `bl`:

```
bl Etiqueta
```

4. La subrutina (programa **invocado**) tendrá que guardar en la pila los registros R4-R11 que utilice para asegurar que estos recuperan su valor cuando se retorna el control al Programa.

```
sub      sp, sp, #4*M     @ reservamos espacio para 4*M bytes
str      r4, [sp,#(M-1)*4]@ salvamos r4 en la pila
              …
str      r10, [sp,#4]     @ salvamos r10 en la pila
str      r11, [sp,#0]     @ salvamos r11 en la pila
```

5. Ejecución de la subrutina.
6. La subrutina recupera (`ldr`) el valor de los registros R4-R11 previamente guardados en la pila (se recuperan en orden inverso). Se restaura el puntero de pila `sp`

```
ldr      r11, [sp,#0]      @ recuperamos r11 de la pila
              …
ldr      r4, [sp,#(M-1)*4]@ recuperamos r4 de la pila

add      sp, sp, #M*4     @ devolvemos el puntero a la posición
                          @ que le corresponde después de haber
                          @ recuperado M elementos
```

7. La subrutina retorna el control al Programa

```
mov pc, lr
```

8. El programa **invocante** mueve los valores retornados en r0-r3 a otros registros o memoria.

9. El programa **invocante** recupera los registros r0 - r3 (si fueron guardados en la pila antes de llamar a la subrutina) con `ldr` y restaura la pila sumándole a `sp` la

cantidad que se haya restado en el punto 1. Evidentemente, si se omitió el paso 1 también se deberá omitir este paso.

A la sección de código de una subrutina encargada de reservar espacio en la pila y almacenar información de contexto se la conoce habitualmente como **prólogo** y se encuentra al principio de las subrutinas. A la parte de una subrutina encargada de restaurar el contenido del contexto y liberar el espacio de la pila asociado se le llama **epílogo** y se encuentra al final de las subrutinas, justo antes del retorno al programa invocante.

3.2.2 Rutinas tipo 'hoja' y 'no hoja'

Cuando una subrutina no llama a ninguna otra subrutina, la consideraremos de tipo 'hoja' para referirnos a ella. En cambio, si una subrutina invoca a otras, usaremos el término 'no hoja' para referirnos a ella.

Como hemos visto en apartados anteriores, al llamar a una subrutina utilizando la instrucción `bl Etiqueta`, implícitamente se está modificando el contenido del registro `lr` de manera que pasará a contener la dirección de retorno. El siguiente ejemplo muestra cómo el programa principal llama a la rutina `mayor` (de tipo 'hoja') y guarda el resultado devuelto en el registro `r7`:

```
          .global start

          start:
0x0C00    mov  r0, #4        @ primer parámetro de la función mayor
0x0C04    mov  r1, #5        @ segundo parámetro de la función mayor
0x0C08    bl   mayor         @ llamada:  lr ← pc+4 = 0x0C0C
                             @           pc ← pos.de mayor = 0x0E04
0x0C0C    mov  r7, r0        @ guardamos el resultado en r7

…         …

          mayor:
0x0E04    cmp  r0,r1         @ comparamos los parámetros
0x0E08    bge  FIN_CMP
0x0E0C    mov  r0,r1         @ guardamos el mayor en r0
          FIN_CMP:
0x0E10    mov pc, lr         @ Salta a 0x0C0C (contenido de lr)
…         …
```

Gracias a que la instrucción `bl` almacenó la dirección de retorno en el registro `lr`, vemos como la subrutina es capaz de devolver el control al programa invocante simplemente llamando a *mov pc, lr* .

Pero ¿qué ocurre con el registro `lr` cuando una subrutina invoca a otra? El siguiente código ilustra un ejemplo de llamada a subrutina de tipo 'no hoja' (ver figura 3.1). El programa principal llamará a esta subrutina (etiquetada como `recorre`) que a su vez invocará a la rutina `mayor` pasándole como argumento el contenido de dos posiciones consecutivas del vector a recorrer:

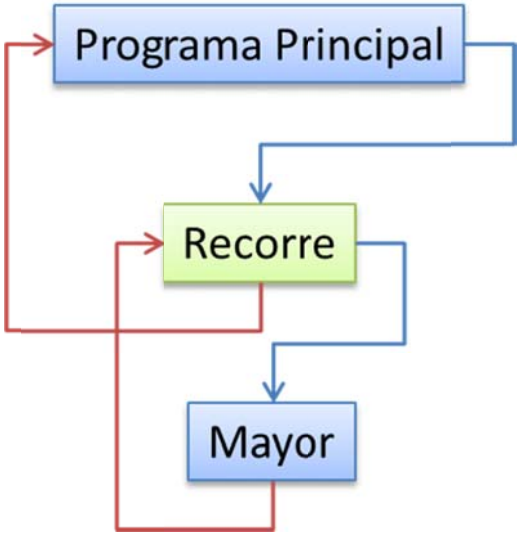

Figura 3.1 Ejemplo de llamada a una subrutina desde otra subrutina

```
#define N 4

int A[N]={7,3,25,4};
int B[N];

void Recorre();
int Mayor();

void main(){
Recorre (A, B, N);
}

int valor, j;
void Recorre (int A[], int B[], int M){
  for(j=0; j<M-1; j++){
    valor = Mayor(A[j],A[j+1]);
    B[j]=valor;
  }
}

int Mayor(int X, int Y){
    if(X>Y)
      return X;
    else
      return Y;
}
```

Que se traduciría en el siguiente código ensamblador:

```
.global start

        .equ            N, 0x04

.data
A:      .word           3,7,25,4

.bss
B:      .space          16

.text   @ Programa principal
        start:
0x0B0C  mov sp,#0x0c200000
0x0C00  ldr r0, =A      @ primer parámetro de la función recorre
0x0C04  ldr r1, =B      @  segundo  parámetro  de  la  función
recorre
0x0C08  mov r2, #N      @ tercer parámetro de la función recorre
0x0C0C  bl  recorre     @ llamada:  lr ← pc+4 = 0x0C10
                        @           pc← pos.de recorre = 0x0D0
0x0C10  …
…       …
@ Fin programa principal
…       …
@ Subrutina recorre
        recorre :
0x0D00  …
…       …
0x0D24  ldr r1, [r0,#4] @ segundo parámetro de la función mayor
0x0D28  ldr r0, [r0,#0] @ primer parámetro de la función mayor
0x0D2C  bl  mayor       @ llamada:  lr ← pc+4 = 0x0D30
                        @           pc ← pos.de mayor = 0x0E04
0x0D30  …
…       …
0x0DA0  mov pc, lr      @ debería saltar a 0x0C10, pero no lo
                                hace porque lr contiene
                                0x0D30 !!!
@ Fin subrutina recorre
…       …
@ Subrutina mayor
        mayor:
0x0E04  cmp r0,r1       @ comparamos los parámetros
0x0E08  bge FIN_CMP
0x0E0C  mov r0,r1       @ guardamos el mayor en r0
        FIN_CMP:
0x0E10  mov pc, lr      @ Salta a 0x0D30 (contenido de lr)
@ Fin subrutina mayor
…       …
```

Como podemos ver, al invocar a la rutina `mayor` desde `recorre`, el contenido de `lr` cambia (pasa a valer `0x0D30`), sin embargo la dirección de retorno al programa principal (`0x0C10`) sigue siendo necesaria para devolver el control una vez procesado el vector. Por otro lado, en la rutina `recorre` también hemos tenido que modificar el contenido de los registros `r0` y `r1` para que contuviesen los valores a comparar por la función `mayor` y al igual que la dirección de retorno original, es información que seguimos necesitando para poder acceder a más datos de `A` y almacenar en `B`, resulta por tanto imprescindible guardar el contenido de `lr`, `r0` y `r1`. Para ello usaremos la pila.

Lo primero que deberemos hacer en la subrutina `recorre` será reservar hueco en la pila para los elementos que queramos guardar como programa invocado (prólogo). En este caso el **prólogo** consiste únicamente en guardar el registro `lr` en la pila.

```
sub sp, sp, #4   @ reservamos espacio para 4 bytes
str lr, [sp,#0]  @ salvamos lr en la pila
```

Antes de llamar a `mayor`, la subrutina `recorre` debe preservar, como programa invocante, el contenido de dos registros (r0, r1), restando la cantidad adecuada al registro `sp` (que apunta a la última posición ocupada de la pila y que crece hacia direcciones de memoria menores):

```
sub sp, sp, #8   @ reservamos espacio para 4*2 bytes
```

Seguidamente podremos salvar el contenido de los registros en la pila (lo normal es comenzar por la parte más profunda e ir subiendo, como haríamos con una pila de platos):

```
str r0, [sp,#4]  @ salvamos r0 en la pila
str r1, [sp,#0]  @ salvamos r1 en la pila
```

Una vez guardado el contenido de los registros en lugar seguro, podremos recuperarlos cuando los necesitemos, (como por ejemplo después de la llamada a `mayor)` y liberar el espacio ocupado en la pila, de la siguiente forma:

```
ldr r1, [sp,#0]  @ recuperamos r1 de la pila
ldr r0, [sp,#4]  @ recuperamos r0 de la pila
add sp, sp, #8
```

Por último, en el **epílogo**, al final de la función `recorre`, deberemos recuperar el contenido de los registros que se deban preservar y liberar el espacio de pila:

```
ldr lr, [sp,#0]  @ recuperamos lr de la pila
add sp, sp, #4   @ recuperamos espacio de 4 bytes
mov pc, lr       @ Salta a 0x0C10 (contenido de lr)
```

El código quedaría como sigue (**cuidado, faltaría incluir también el salvado, en el prólogo, y restaurado, en el epílogo, de los registros r4 a r11 que cada subrutina utilice**):

```
@ Subrutina recorre
        recorre:
@ Prólogo
0x0D00   sub sp, sp, #4   @ reservamos espacio para 4 bytes
0x0D04   str lr, [sp,#0]  @ salvamos lr en la pila
@ Fin prólogo
...      ...
@ Antes de llamar a la subrutina hoja salvamos r0 y r1 porque
@ necesitamos sus valores a la vuelta de mayor para seguir
@ ejecutando recorre
0x0D38   sub sp, sp, #8   @ reservamos espacio para 8 bytes
0x0D3C   str r0, [sp,#4]  @ salvamos r0 en la pila
0x0D40   str r1, [sp,#0]  @ salvamos r1 en la pila
@ Pasamos los parámetros a mayor por r0 y r1
0x0D44   ldr r1, [r0,#4]  @ segundo parámetro de la función mayor
0x0D48   ldr r0, [r0,#0]  @ primer parámetro de la función mayor
0x0D4C   bl  mayor        @ llamada:  lr ← pc+4 = 0x0D50
                          @           pc ← pos.de mayor = 0x0E04
0x0D50   mov r5, r0       @ guardamos el resultado de la subrutina
                          @ en otro registro para recuperar r0
0x0D54   ldr r1, [sp,#0]  @ recuperamos r1 de la pila
0x0D58   ldr r0, [sp,#4]  @ recuperamos r0 de la pila
0x0D5C   add sp, sp, #8   @ liberamos espacio ocupado por r0 y r1

...      ...
@ Epílogo
0x0D98   ldr lr, [sp,#0]  @ recuperamos lr de la pila
0x0D9C   add sp, sp, #4   @ liberamos espacio ocupado por lr
0x0DA0   mov pc, lr       @ Salta a 0x0C10 (contenido de lr)
@ Fin epílogo
@ Fin subrutina recorre

...      ...

@ Subrutina mayor
        mayor:
0x0E04   cmp r0,r1        @ comparamos los parámetros
0x0E08   bge FIN_CMP
0x0E0C   mov r0,r1        @ guardamos el mayor en r0
        FIN_CMP:
0x0E10   mov pc, lr       @ Salta a 0x0D50 (contenido de lr)
@ Fin subrutina mayor
```

3.3 Desarrollo de la práctica 3

Trabajo a realizar en casa:

a. Codificar en ensamblador del ARM la siguiente función en C encargada de buscar el valor máximo de un vector **A** de enteros positivos de longitud **longA** y devolver la posición de ese máximo (el índice):

```
int i, max, ind;
int max(int A[], int longA){
  max=0;
  ind=0;
  for(i=0; i<longA; i++){
    if(A[i]>max){
      max=A[i];
      ind=i;
    }
  }
  return(ind);
}
```

NOTA: El código del apartado a se corresponde (excepto por las instrucciones de gestión de subrutina) con el bucle interno del apartado b de la práctica 2. Nótese que la subrutina es de tipo hoja, por tanto sólo necesita salvar y restaurar los registros r4 a r11 que utilice.

b. Codificar en ensamblador del ARM un algoritmo de ordenación basado en el código del apartado anterior. Supongamos un vector **A** de **N** enteros mayores de 0, queremos rellenar un vector **B** con los valores de **A** ordenados de mayor a menor. Para ello nos podemos basar en el siguiente código de alto nivel:

```
#define N 8
int A[N]={7,3,25,4,75,2,1,1};
int B[N];
int j;
void main(){
  for(j=0; j<N; j++){
    ind=max(A,N)
    B[j]=A[ind];
    A[ind]=0;
  }
}
```

NOTA: El código del apartado b se corresponde (excepto por las instrucciones de llamada a subrutina) con el bucle externo del apartado b de la práctica 2.

En el laboratorio se pedirá una modificación del trabajo realizado en casa. Igual que en las prácticas anteriores se realizará un examen online en el laboratorio para

comprobar que el estudiante ha entendido correctamente todos los conceptos asociados a la práctica desarrollada.