Técnicas Digitales II

Llamado a Funciones

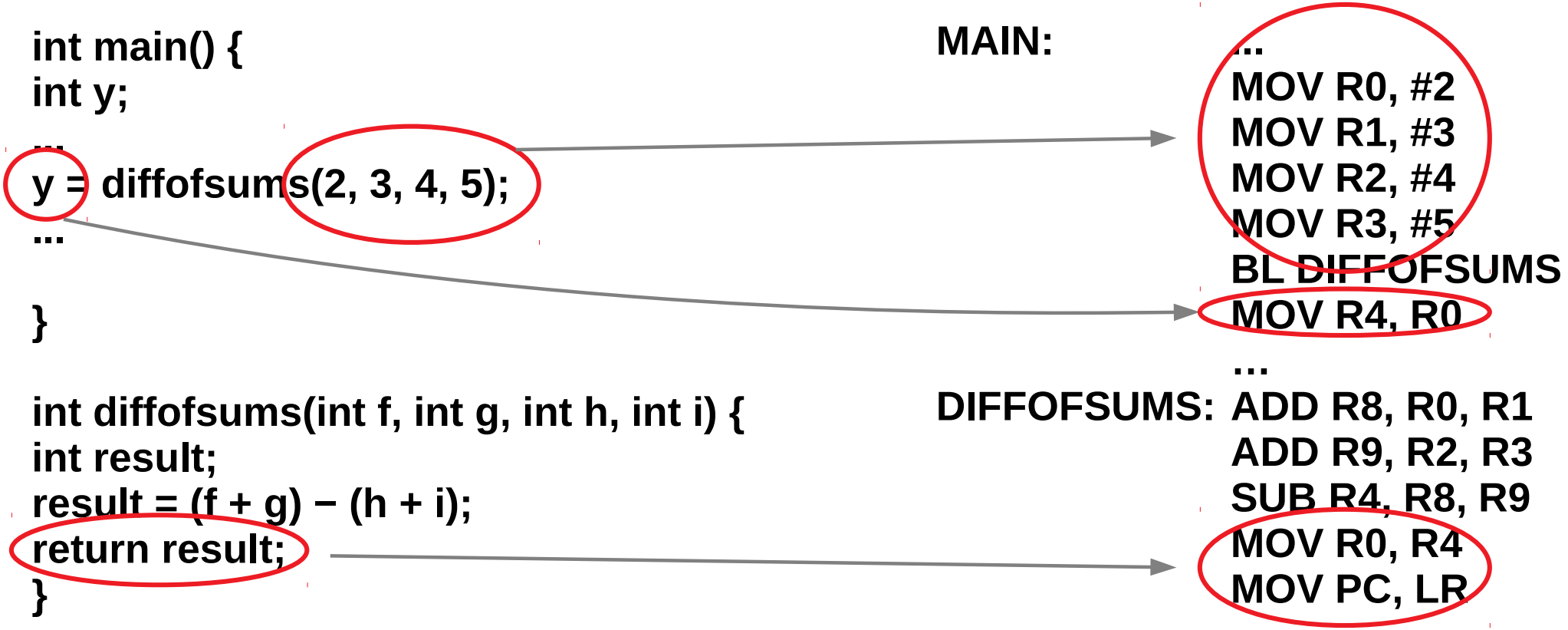
Dr.Ing.Steiner Guillermo

Funciones

- Una función, es un segmento de código en general encargado de resolver una tarea especifica.
- Se encuentra separado del bloque principal y puede ser llamado por este, por si mismo o por otra función.
- Útil para reutilizar código y hacer el programa mas modular y legible.
- Recibe el nombre de procedimiento, subrutina o subprograma.
- Elementos de una función
  - Nombre único.
  - Un valor de retorno (Salida).
  - Una lista de parámetros (Entradas).
  - El propio código.

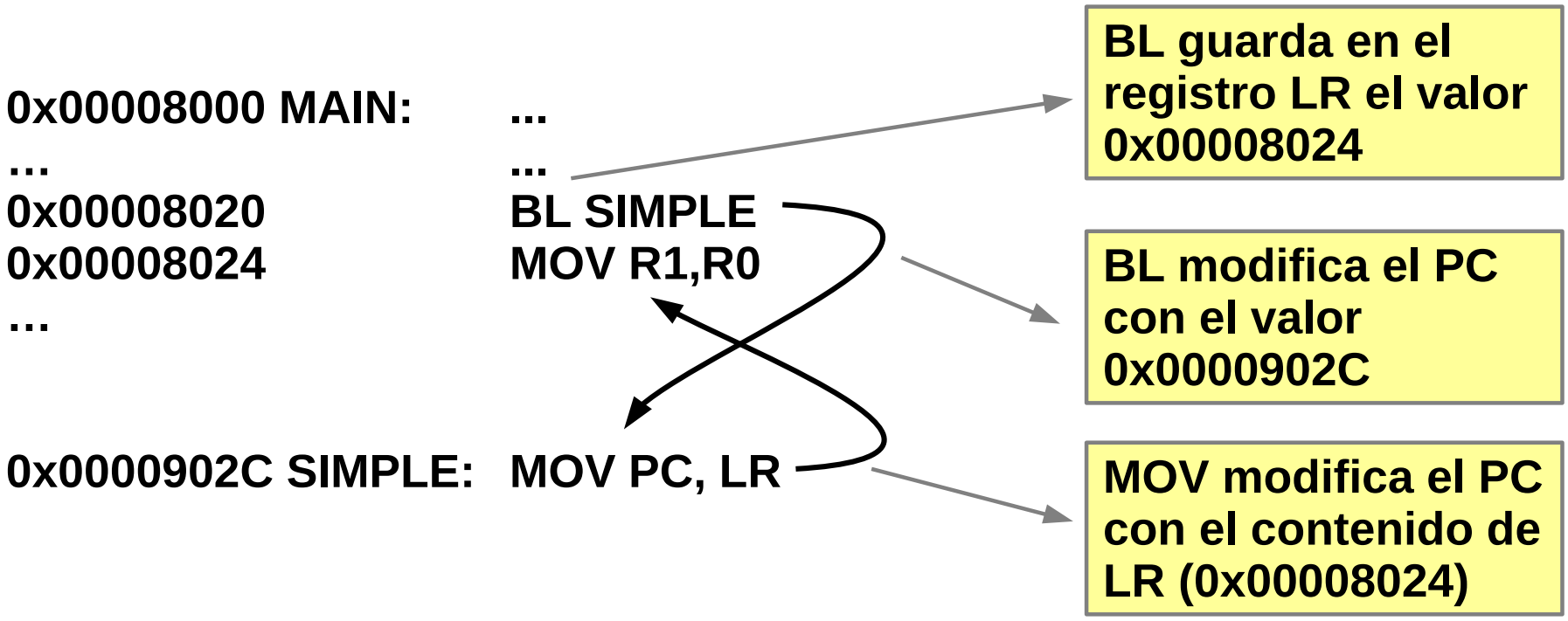
Funciones en ARM

- En ARM las funciones:
  - Devuelven el valor en R0
  - Utilizan R0-R3 para enviar los parámetros de entrada.



Funciones en ARM

- ARM utiliza el **BL** como instrucción para el llamado a funciones.
- **BL** guarda el valor de retorno en **LR** permitiendo a la función una vez finalizada regresar desde donde fue llamada

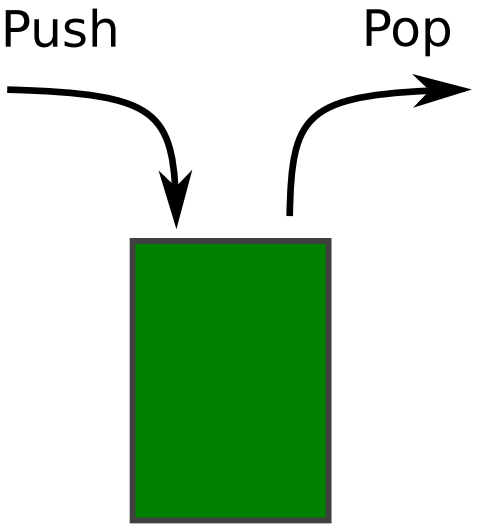


STACK

- Los parámetros son enviados de izquierda a derecha en los registros R0-R3, si se requiere mayor cantidad de parámetros, utilizaremos el STACK
- El STACK, es una porción de memoria que se utiliza para guardar información temporal dentro de una función o programa.
- Esta porción de memoria se expande cuando se requiere guardar información y se libera en el momento que esa información no es mas necesaria.
- Las funciones pueden por ejemplo utilizar el stack para las variables locales, desocupando la memoria al salir de la función.

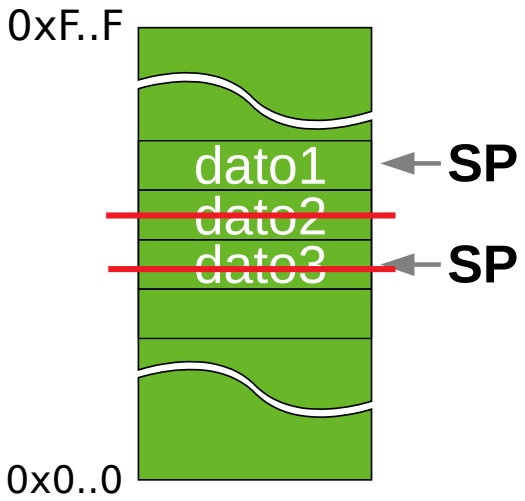
STACK

- El STACK es una lista ordenada del tipo *last-in-first-out* (LIFO),
- Como en una pila, el nuevo dato se apila sobre el anterior y cuando se extrae un dato es el último que se agregó.
- En ARM se utiliza el registro R13 o SP para el manejo del STACK, este registro contendrá la dirección del último dato guardado.
- Además por cada dato agregado, el stack se expandirá a posiciones de memoria mas bajas.



STACK

- Ejemplo simple de utilización del STACK



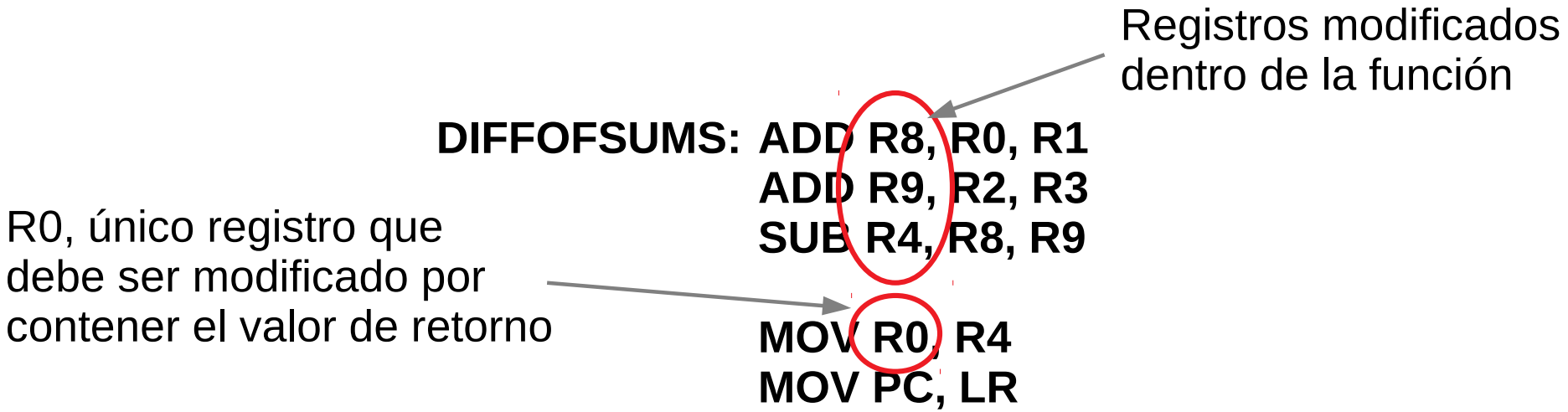
Al entrar a una función, se crean dos variables locales dato2 y dato3

Dentro de la función se utilizan las variables

Al salir de la función Se recupera el espacio

Uso del Stack

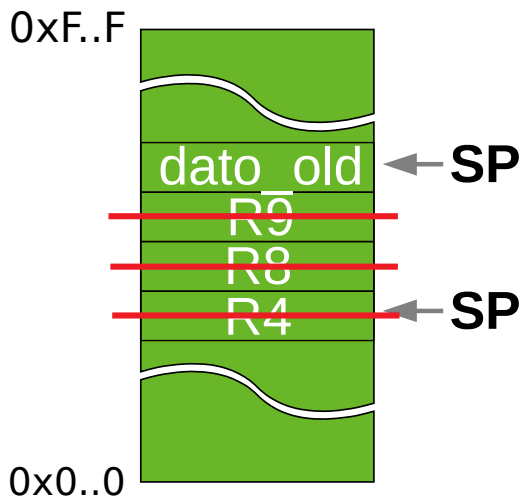
- Una función no debe afectar ningún registro o zona de memoria que el programa que llamó a la función utilice.
- Para evitar problemas, toda función guardará en el stack los registros utilizados dentro de la misma para recuperar su valor antes de retornar el programa que la llamó.



Uso del Stack

- Los pasos para guardar los registros en el stack serán los siguientes:

1) Hacer espacio en el stack	**DIFFOFSUMS:**	**SUB SP, SP, #12**
2) Guardar los registros		**STR R9, [SP, #8]** **STR R8, [SP, #4]** **STR R4, [SP]**
3) Ejecutar la función utilizando los registros		**ADD R8, R0, R1** **ADD R9, R2, R3** **SUB R4, R8, R9** **MOV R0, R4**
4) Restaurar los valores originales del stack		**LDR R4, [SP]** **LDR R8, [SP, #4]** **LDR R9, [SP, #8]**
5) Desocupar el espacio en el stack		**ADD SP, SP, #12**
		MOV PC, LR



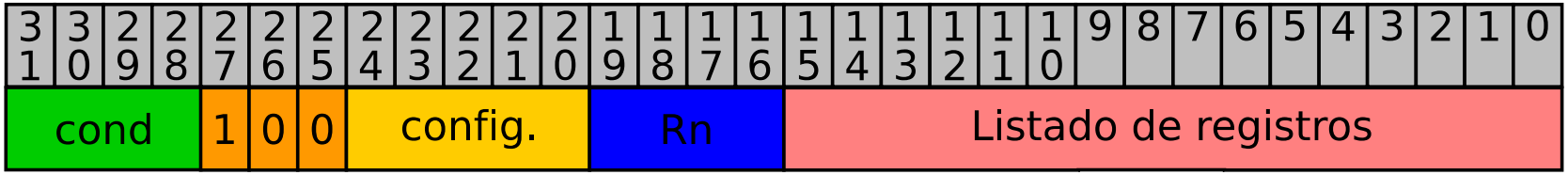
Carga y Almacenamiento de Registros Múltiples

- Para facilitar el almacenamiento y carga de los registros al stack, ARM provee funciones de carga y almacenamiento de múltiples registros.

Full Descending

```
Almacenamiento  DIFFOFSUMS: STMFD SP!, {R4, R8, R9}
                            ADD R8, R0, R1
                            ADD R9, R2, R3
                            SUB R4, R8, R9
                            MOV R0, R4
Carga                       LDMFD SP!, {R4, R8, R9}
                            MOV PC, LR
```

LDM (b20=1) / STM (b20=0)



Carga y Almacenamiento de Registros Múltiples

- Las Instrucciones LDM y STM permite ser configuradas para diferentes modos de funcionamiento.
- Como cualquier instrucción de load / store, requiere de un registro como base para acceder a la posiciones de memoria que interactúan con los registros.
- Además de esto, y como pueden ser varios los registros leídos o a guardar, se requiere que la propia instrucción calcule las posiciones donde se encuentran los restantes.

Carga y Almacenamiento de Registros Múltiples

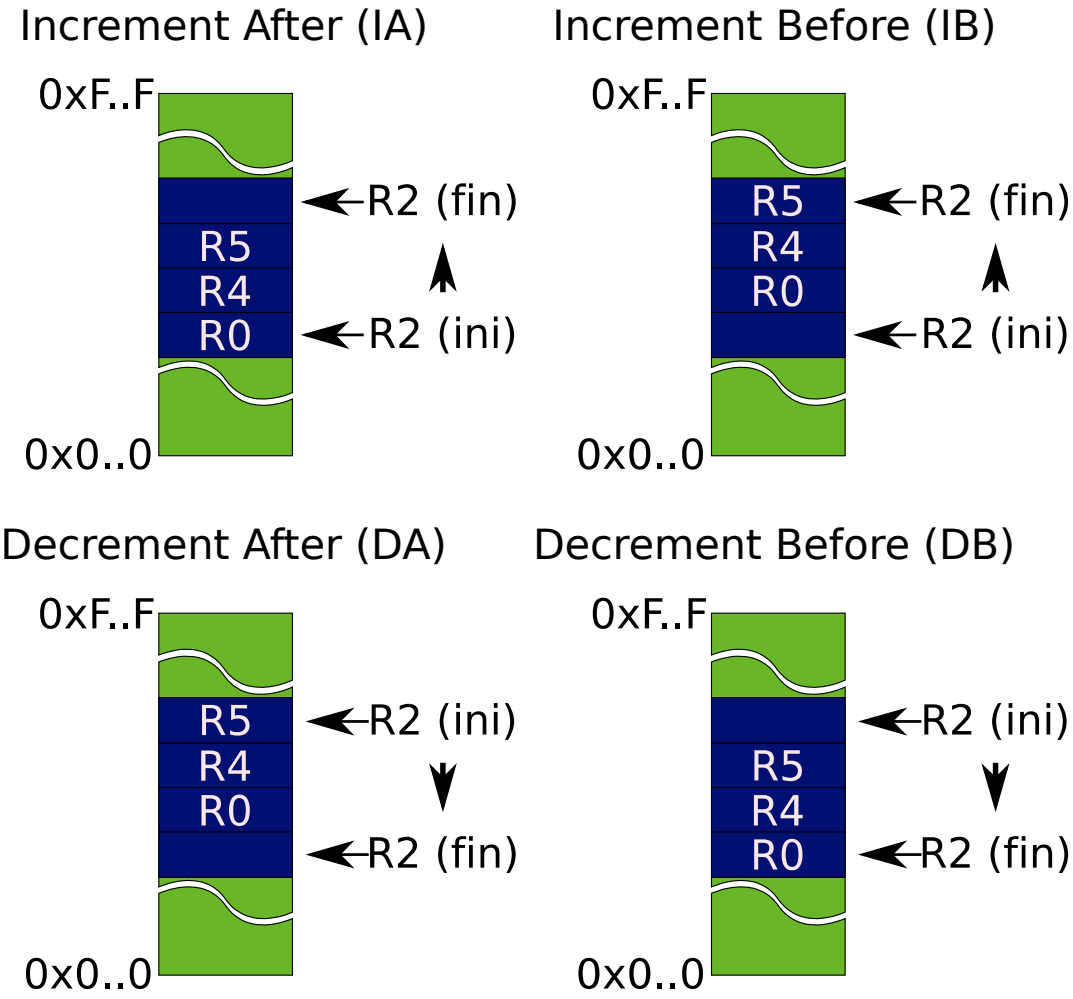
- Según el sufijo elegido, se obtiene los siguientes modo de direccionamiento.

Nemónico según el modo de direccionamiento (IA,IB,DA,DB)

STMxx R2!,{R0,R4,R5}

Indica que el registro base se actualizará

No importa el orden, siempre se comienza por el registro mas bajo y se continua en orden ascendente



Carga y Almacenamiento de Registros Múltiples

- Los modos indicados, permiten usar las instrucciones de CyA múltiples para una diversidad de aplicaciones.
- La utilización específica para el STACK, será una combinación de las indicada anteriormente.
- Combinando entonces estos modo de direcciones, se pueden implementar 4 tipos de stack.

Full/Empty: indica si el puntero apunta al último dato guardado (full) o no (empty)

Ascending/Descending: Se refiere a la manera que se guardan los datos en memoria

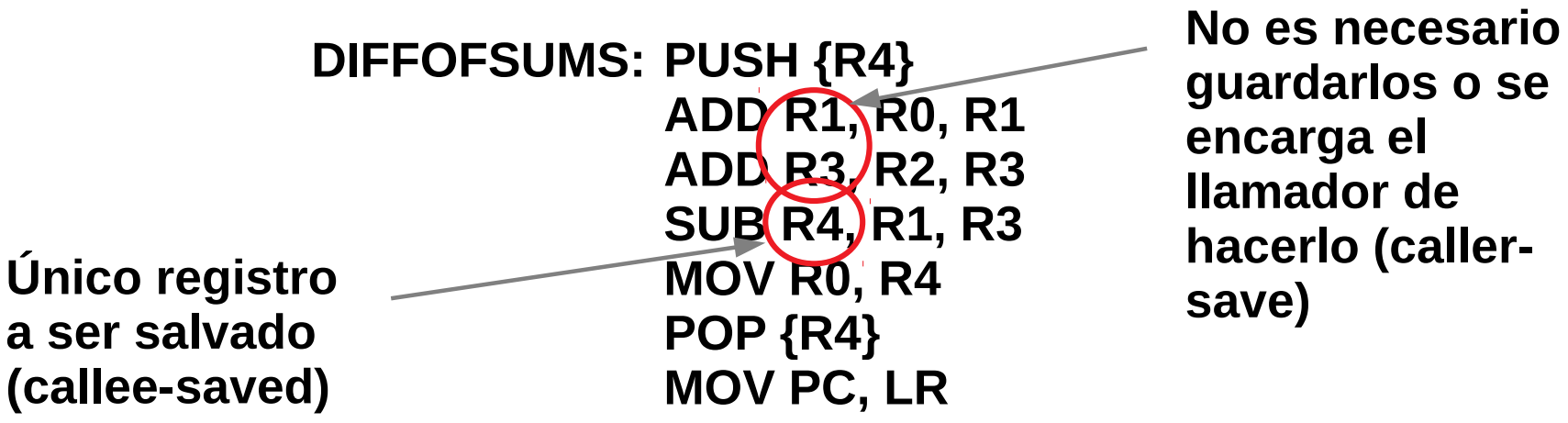
Nombre	Nemónico	Nemónico LDM	Nemónico STM
Full Ascending	FA	LDMDA	STMIB
Empty Ascending	EA	LDMDB	STMIA
Full Descending	FD	LDMIA	STMDB
Empty Descending	ED	LDMIB	STMDA

STACK Full Descending

- **PUSH** y **POP** son sinónimos para las instrucciones **STM** y **LDM** respectivamente configuradas en el modo FD y utilizando el stack como registro base.
  - **`PUSH {regs}`** → **`STMFD SP!,{regs}`**
  - **`POP {regs}`** → **`LDMFD SP!,{regs}`**
- El Full Descending queda entonces como el modelo estándar utilizado por ARM, y es utilizado en otras arquitecturas como modo único de manejo del STACK (ejemplo x86)

Reduciendo los registros salvados

- ARM divide a los registros, en registros preservados y no preservados.
  - Preservados: R4-R11,SP y LR, una función debe guardar estos registros antes de ser modificados y restaurado su valor antes de salir de la función El código que llama a una función supone que esta no va a modificar estos registros.
  - No preservados; R0-R3, R12 pueden ser cambiados libremente, no se puede asumir que no serán modificados luego de llamar a una función.



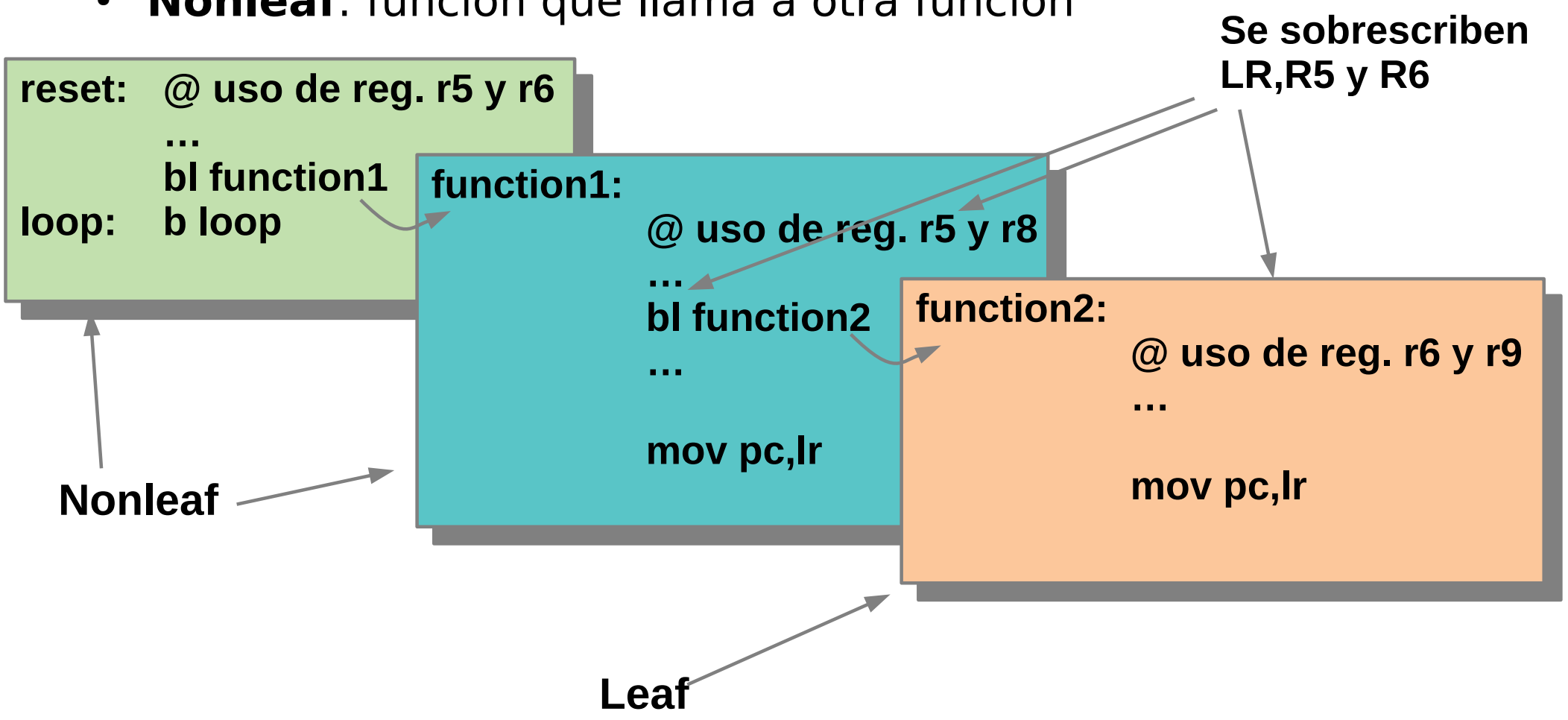
Reduciendo los registros salvados

- Tabla con resumen de los registros Preservados y No Preservados.

Preservados	No Preservados
R4-R11	**Registro temporal R12**
Stack Pointer R13 (SP) Esta implícito en el propio uso. Al entrar a una función los datos guardados en el stack son descargados al salir, recuperando su valor inicial.	**Registros de Argumentos R0-R3**
Dirección de Retorno R14 (LR) Si una función llama a otra, se deberá guardar la dirección de retorno de la primera.	**Current Program Status Register (CPSR)** Las Banderas no son preservadas entre llamado de funciones
Stack sobre el SP	**Stack por debajo del SP**

Funciones *leaf* y *nonleaf*

- **Leaf:** función que no llama a otra función.
- **Nonleaf**: función que llama a otra función



Funciones *leaf* y *nonleaf* uso del STACK

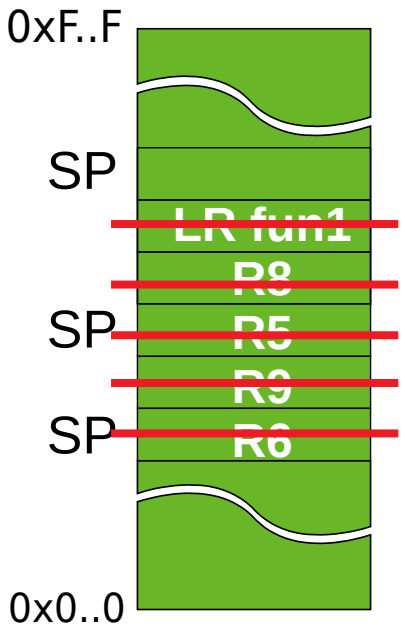
```
reset:   @ uso de reg. r5 y r6
         ...
         bl function1
loop:    b loop
```

```
LR= ret call function2
R5 = val main fun1
R6 = val main fun2
R8 = val fun1
R9 = val fun2
```

```
function1: push {r5,r8,lr}
           @ uso de reg. r5 y r8
           ...
           bl function2
           ...
           pop {r5,r8,lr}
           mov pc,lr
```

```
function2: push {r6,r9}
           @ uso de reg. r6 y r9
           ...
           pop {r6,r9}
           mov pc,lr
```

Llamar function1
Copiar al STACK
Actualiza var en fun1
Llamar a fun2
Copiar al STACK
Actualiza var en fun2
Recuperar del STACK
Retorno a function1
Recupera del STACK



Uso del STACK

```
int f1(int a, int b) {
        int i, x;
        x = (a + b)*(a − b);
        for (i=0; i<a; i++)
                x = x + f2(b+i);
        return x;
}
```

```
int f2(int p) {
        int r;
        r = p + 5;
        return r + p;
}
```

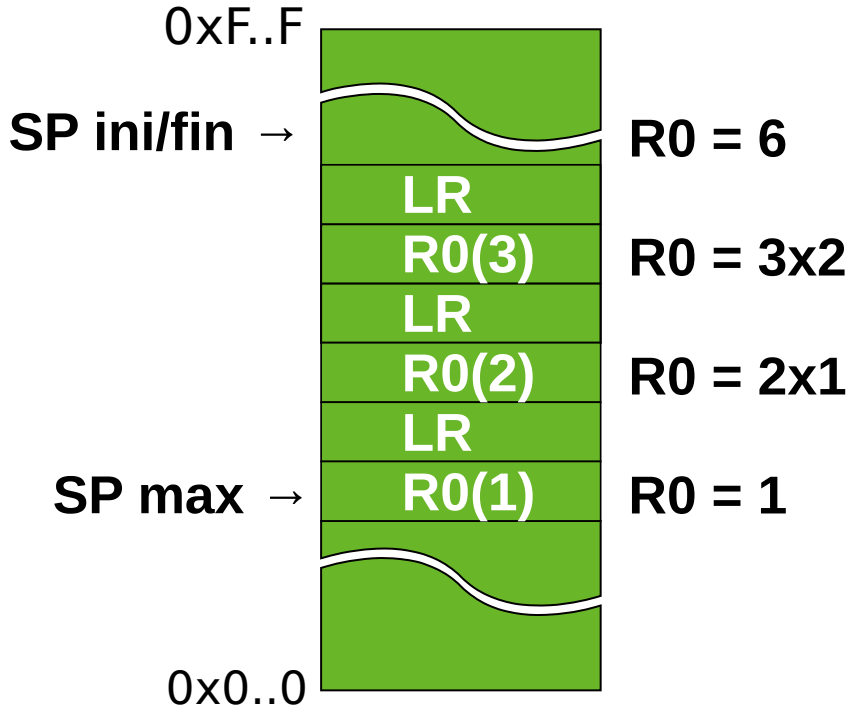
```
F1:     PUSH {R4, R5, LR}
        ADD R5, R0, R1
        SUB R12, R0, R1
        MUL R5, R5, R12
        MOV R4, #0
FOR:    CMP R4, R0
        BGE RETU
        PUSH {R0, R1}
        ADD R0, R1, R4
        BL F2
        ADD R5, R5, R0
        POP {R0, R1}
        ADD R4, R4, #1
        B FOR
RETU:   MOV R0, R5
        POP {R4, R5, LR}
        MOV PC, LR
```

```
F2: PUSH {R4}
    ADD R4, R0, 5
    ADD R0, R4, R0
    POP {R4}
    MOV PC, LR
```

LR debe ser guardado en toda función nonleaf

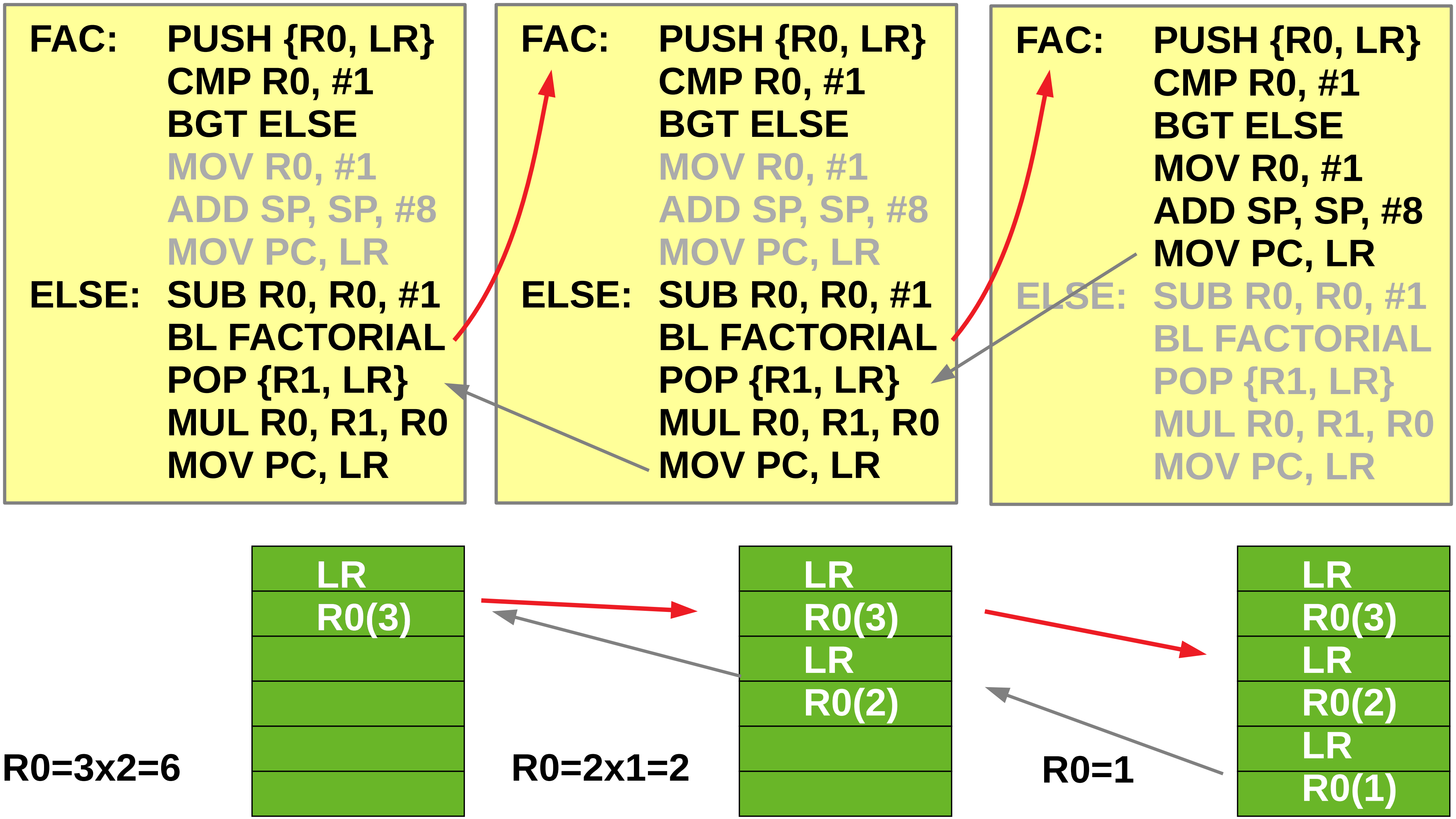
Funciones Recursivas

- Son funciones nonleaf que se llaman a si mismas
- Son ambas llamador y llamada
- Se deben guardar los registros preservados, pero también los no preservados.



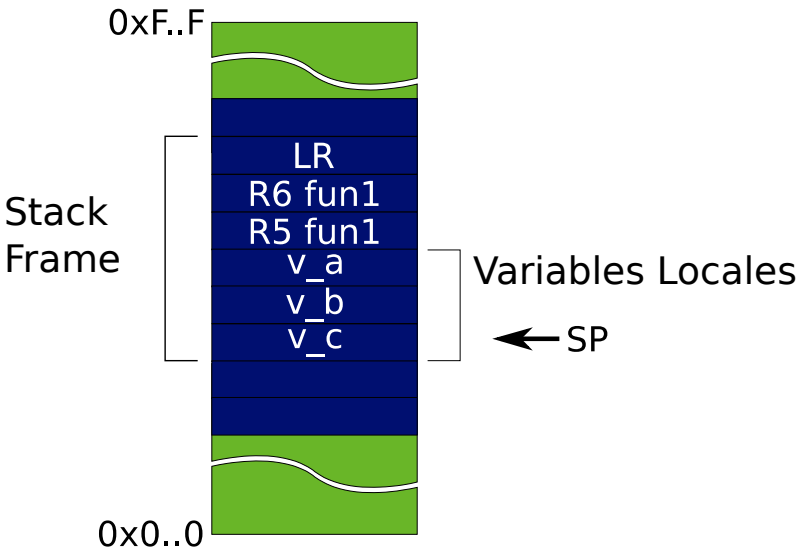

```
FACTORIAL:  PUSH {R0, LR}
            CMP R0, #1
            BGT ELSE
            MOV R0, #1
            ADD SP, SP, #8
            MOV PC, LR
ELSE:       SUB R0, R0, #1
            BL FACTORIAL
            POP {R1, LR}
            MUL R0, R1, R0
            MOV PC, LR
```

Funciones Recursivas



Variables Locales

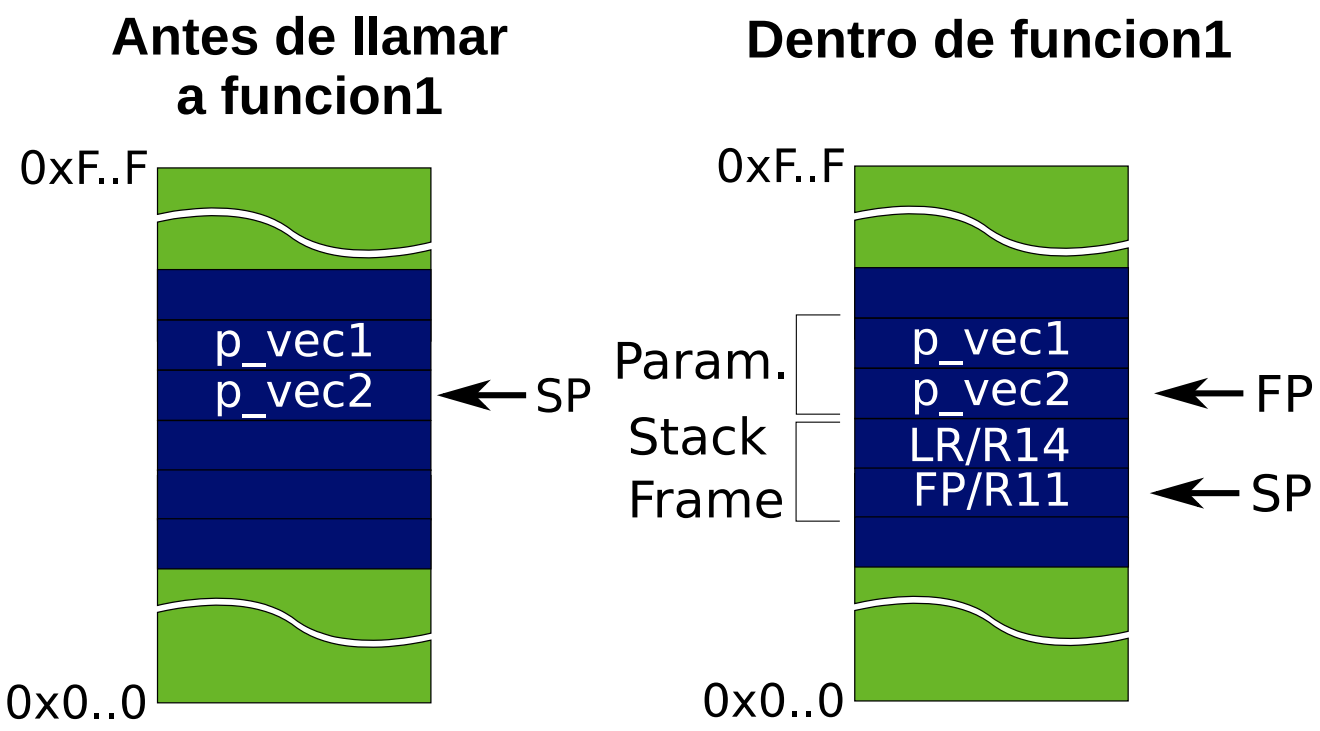
- Las variables locales solo son accedidas por la propia función.
- Se utiliza el stack para guardarlas.
- Al salir se recupera la memoria




```
Function1: push{r5,r6,lr}
           sub sp,#12
           str r0,[sp,#+8]  @v_a
           str r1,[sp,#+4]  @v_b
           str r2,[sp]      @v_c
           bl funcion2
           ....
           ....
           add sp,#12
           pop {r5,r6,lr}
           mov pc,lr
```

Parámetros por stack

- Cuando se debe enviar mas de 4 parámetros se utiliza el STACK




```
Main:   ...
        ldr r1,=vector1
        push {r1}
        ldr r1,=vector2
        push {r1}
        bl funcion1
        add sp,#8
        ...
```

```
function1:  push {fp,lr}
            add fp,sp,#8
            ...
            ldr r0,[fp,]
            ldr r1,[fp,#4]
            ...
            pop {fp,lr}
            mov pc,lr
```

- Bibliografía

Harris & Harris. Digital design and computer architecture: ARM edition. Elsevier, 2015. Capítulo 6.

William Hohl & Christopher Hinds. ARM assembly language. Fundamentals and techniques. 2nd edition. CRC press, 2015. Capítulo 13.

¿ Preguntas ?