Programación en Lenguaje Ensamblador

Sistemas con Microprocesadores

Ing. Esteban Volentini (evolentini@herrera.unt.edu.ar)

http://microprocesadores.unt.edu.ar/procesadores

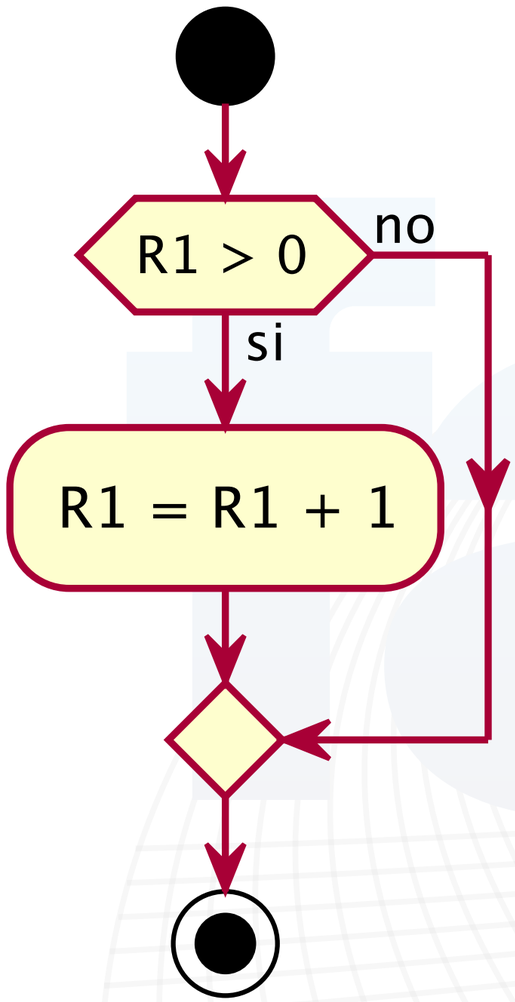
Cronograma

Actividad	Inicio	Descripción	Fin
Presentación	19/08	Reglamento de la Materia	✔
Tema 1	19/08	Estructura de las computadoras	✔
Tema 2	26/08	Proyecto con un microcontrolador	✔
Tema 3	30/08	Descripción funcional de microprocesador	✔
Tema 4	13/09	Programación en lenguaje ensablador	←
Tema 5	25/09	Descripción general de un microcontrolador	
Tema 6	27/09	Estructura general de microcontrolador	
Parcial	09/10	Primer examen parcial	
Tema 7	14/10	Sistema de Interrupciones	
Tema 8	21/10	Entradas y salidas digitales	
Tema 9	28/10	Entrada/salida con perifericos	
Tema 10	06/11	Temporizadores	
Proyectos	25/11	Seminarios de Proyectos	
Parcial	04/12	Segundo examen parcial	

Temas a Tratar

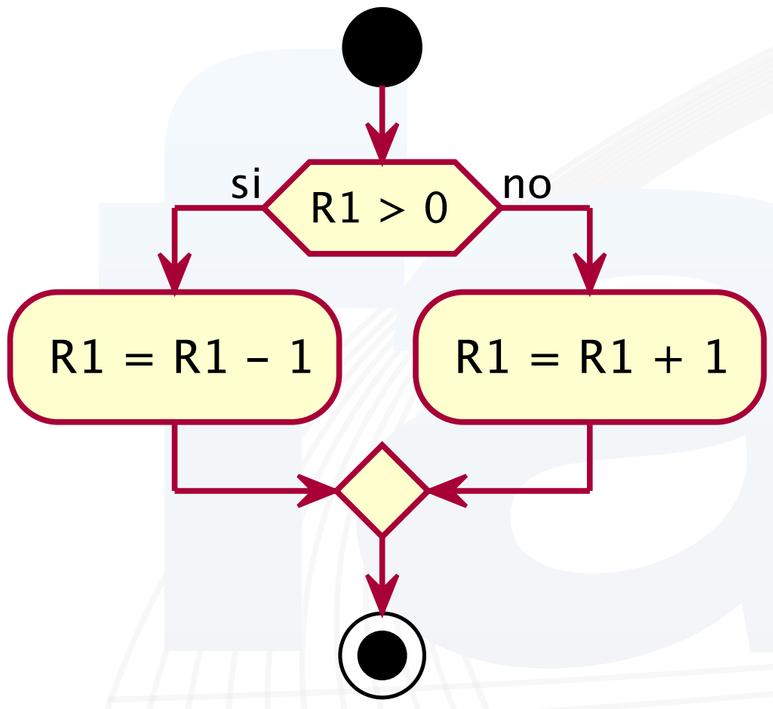
- Patrones de programación
- Pilas.
  - Implementación en ARM-Cortex-M4.
  - Ejemplos de uso.
- Variables globales y locales
- Subrutinas.
  - Implementación en ARM-Cortex-M4.
  - Pasaje de Parámetros.
  - Ejemplos.

Estructuras típicas: if-then



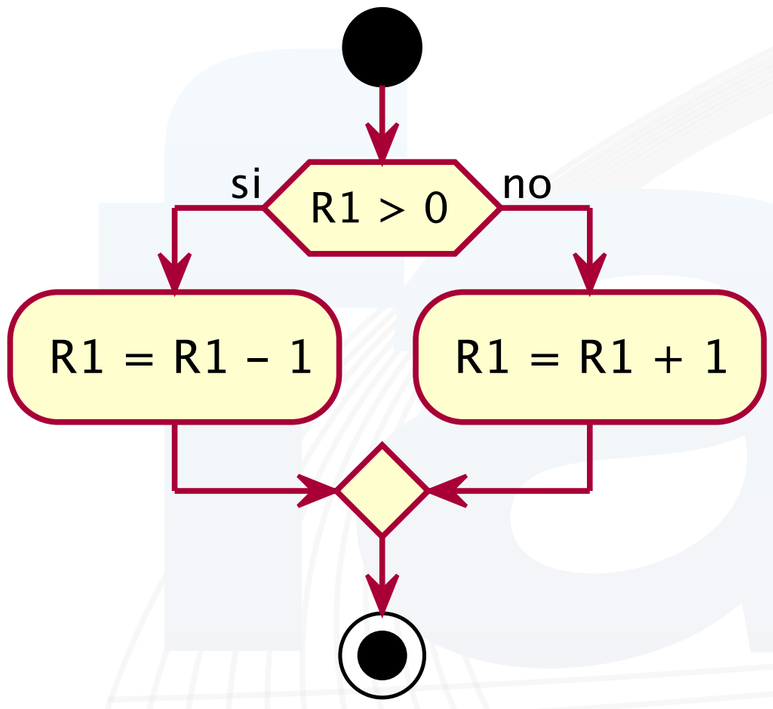

```
        ...
        // Compara R1 con cero
        CMP R1,#0
        // Salta por menor o igual
        BLE label
        // Bloque then
        ADD R1,#1
label:  ...
```

Estructuras típicas: if-then-else



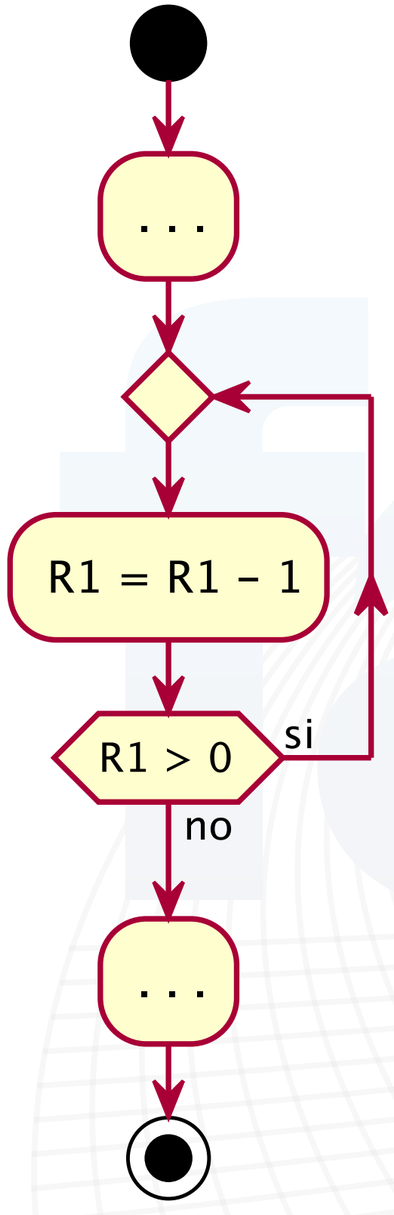

```
...
// Compara R1 con cero
CMP R1,#0
// Salta por menor o igual
BLE else
// Bloque then
SUB R1,#1
// Salta al final del if
B label
// Bloque else
else:   ADD R1,#1
label:  ...
```

Estructuras típicas: if-then-else



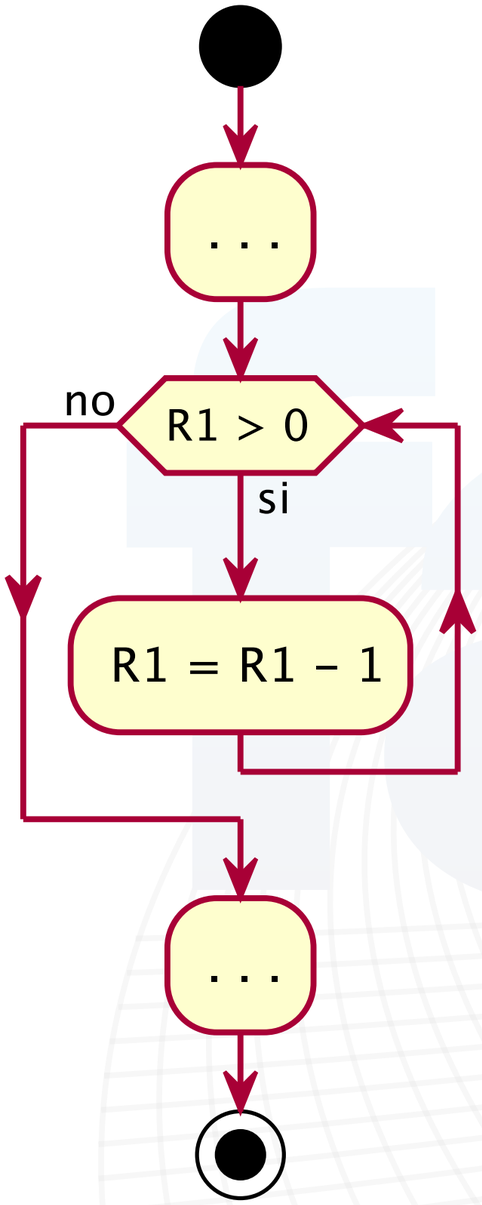

```
        ...
        // Compara R1 con cero
        CMP R1,#0
        // Salta por mayor
        BGT then
        // Bloque else
        ADD R1,#1
        // Salta al final del if
        B label
        // Bloque then
then:   SUB R1,#1
label:  ...
```

Estructuras típicas: do-while




```
lazo:   ...
        // Cuerpo del lazo
        SUBS R1,#1
        // Salta por mayor
        BGT lazo
        ...
```

Estructuras típicas: while-do

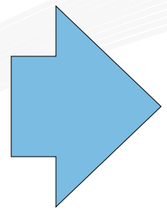



```
                // Compara antes del salto
                CMP R1,#0
                // Salta por menor o igual
lazo:   BLE label
                // Cuerpo del lazo
                SUBS R1,#1
                // Cierra el lazo
                B lazo
label:  ...
```

Bloque If-Then-Else (IT)

- If – Then ⇨ instrucción **`IT`** (16 bit)
  - Se puede crear un bloque de hasta 4 instrucciones condicionales

```
ITTET EQ
InstEQ  1
InstEQ  2
InstNE  3
instEQ  4
```



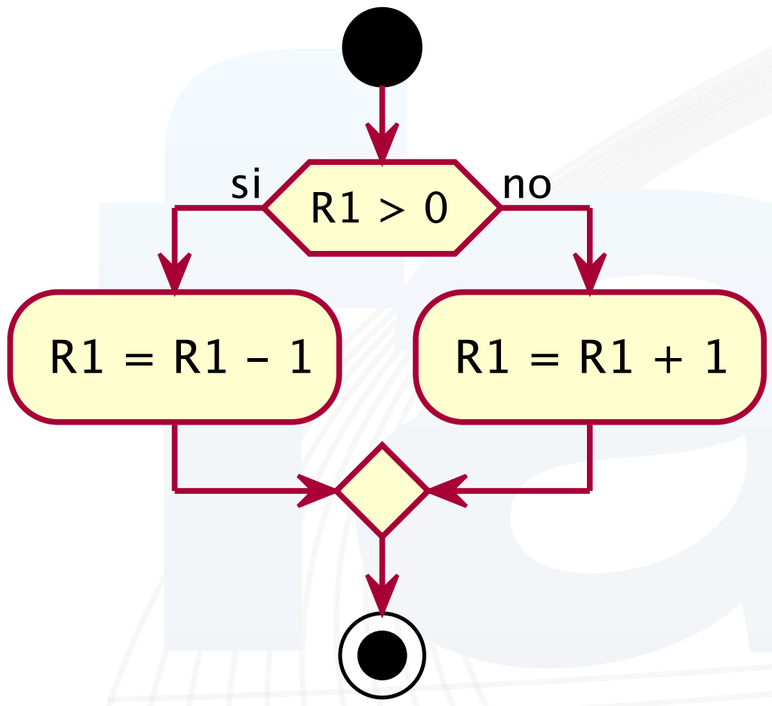
```
ITTET EQ
MOVEEQ
ADDEQ
SUBNE
ORREQ
```

- Cualquier código de condición del ARM se puede usar.
- Instrucciones de 16-bit en bloque no afectan los flags CCR
  - Excepto una instrucción de comparación.
  - Instrucciones de 32 bit pueden afectar flags
- Se informa que se ejecuta un bloque **`IT`** en CPSR (Status)
  - Permite que un bloque condicional pueda ser interrumpido sin problemas.
  - NO se puede saltar hacia dentro o afuera de un bloque 'IT' ¿por qué?

Bloque If-Then-Else

- Formato: **`ITxyz`** condición
- Máximo 4 instrucciones. La condición vale para la primera instrucción.
- **`x`** es una condición para la segunda, **`y`** para la tercera y **`z`** para la cuarta (deben ser **`T`** o **`E`**).
- Se usan tantas condiciones como instrucciones haya en el bloque (**`T`** más 3 máximo).
- Las instrucciones en el bloque tienen la condición como sufijo.

Bloque If-Then-Else




```
...
// Compara R1 con cero
CMP R1,#0
// Inicia bloque por mayor
ITE GT
// Bloque then
SUBGT R1,#1
// Bloque else
ADDLE R1,#1
...
```

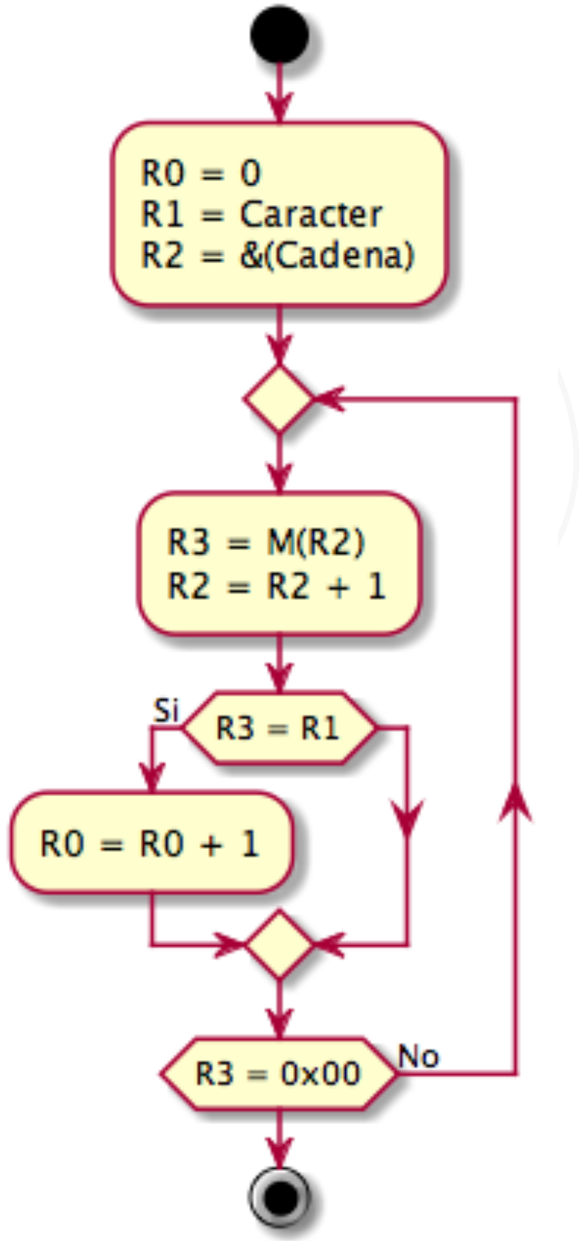
Ejemplo Bloque If-Then-Else

```
CMP     R0,R1       // para fijar los flags
ITEE    NE          // Próximas 3 inst. condicionales
ANDNE   R0,R0,R1
ADDSEQ  R2,R2,#1    // Cambia flags (S), 32 bits
MOVEQ   R2,R3       // MOVE condicional
```

- La cantidad de condiciones es igual a cantidad de instrucciones del bloque **`IT`**
- La siguiente instrucción ya no puede ser condicional.
- Como los saltos retrasan el pipeline, estas instrucciones sirven *para que no haya saltos.*

Ocurrencias de un caracter

- Se requiere contar la cantidad de veces que un carácter aparece en una cadena.
- La cadena sigue el estándar “C”, es decir que el final está marcado por el carácter nulo **0x00**.
- La cadena esta almacenada a partir de la dirección de memoria **cadena**.
- El carácter a buscar esta almacenado en la dirección de memoria **caracter**.
- El resultado de debe colocar en el registro **R0**.



Búsqueda de un caracter

```
            .section .data
cadena:     .asciz "SISTEMAS CON MICROPROCESADORES"
caracter:   .ascii "S”

            .section .text
            .global reset
reset:      MOV   R0,#0x00        // Resultado es cero ocurrencias
            LDR   R1,=caracter    // Apunta al carácter
            LDRB  R1,[R1]         // Carga el carácter a buscar
            LDR   R2,=cadena      // Apunta a la cadena
lazo:       LDRB  R3,[R2],#1      // Carga el carácter actual
            CMP   R3,R1           // Compara los caracteres
            IT    EQ              // Si los registros son iguales
            ADDEQ R0,#1           // Entonces incrementa resultado
            CMP   R3,#0x00        // Si el carácter no es igual a 0
            BNE   lazo            // Entonces repite el lazo
stop:       B     stop
```

Sentencia Switch en C

switch (test) { case 0: ... break; case 1: ... break}

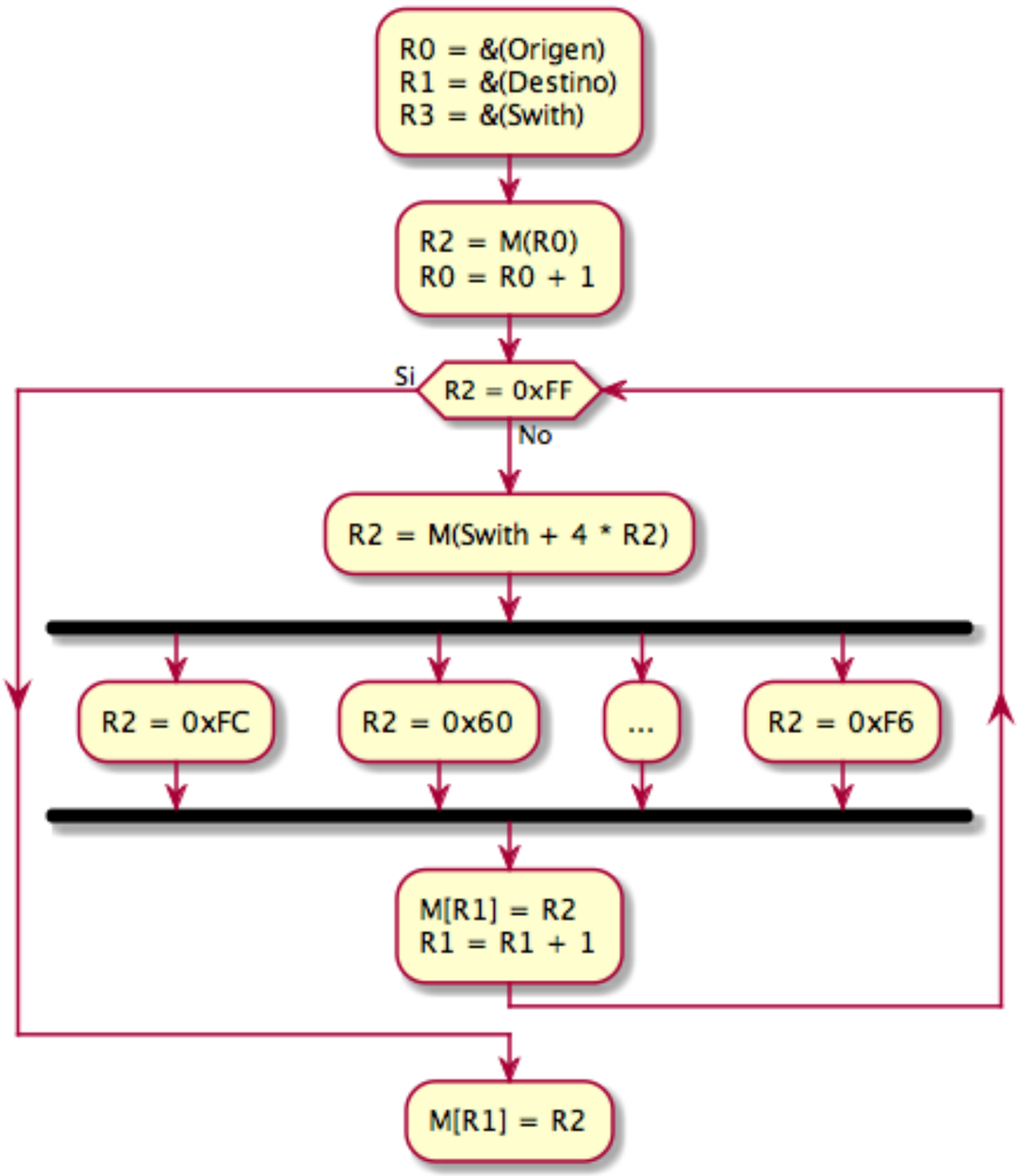
```
                LDR R2,=test              // puntero al valor de test
                LDR R0,[R2]               // cargar valor para test
                ADR R1,switchtab          // dir de la tabla de switch
                LDR R2,[R1,R0,LSL #2] // indexar tabla de switch
                BX  R2

switchtab:      .word case0                 // address of case0
                .word case1                 // address of case1
                . . .
```

- La tabla contiene palabras, por eso se multiplica por 4 el valor de test.
- No olvidar, cada dirección debe ser impar por compatibilidad Thumb.
- Hay una instrucción para armar una tabla de desplazamientos de PC, empleando direccionamiento relativo. Leer en la casa.

Ejemplo: Conversión de Códigos

- Se plantea resolver el mismo problema del conversor de código BCD a siete segmentos.
- No se quiere usar una tabla de conversión sino una estructura switch.
- Los datos están almacenados en los mismos lugares que antes.



Ejemplo: conversión códigos

```
reset:      LDR  R0,=origen          // Apunta R1 al bloque de origen
            LDR  R1,=destino         // Apunta R2 al bloque de destino
            ADR  R3,switch           // Apunta R2 al bloque con la tabla
lazo:       LDRB R2,[R0],#1          // Carga en R2 el elemento a convertir
            CMP  R2,0xFF             // Determina si es el fin de conversión
            BEQ  final               // Terminar si es fin de conversión

            LDR  R2,[R3,R2,LSL #2]   // Cargar en R2 la dirección del caso switch
            ORR  R2,0x01             // Fija el MSB para indicar instrucciones THUMB
            BX   R2                  // Saltar al caso correspondiente
            .align                   // Asegura que la tabla de saltos este alineada
switch:     .word case0, case1, case2, case3, case4
            .word case5, case6, case7, case8, case9
case0:      MOV  R2,#0xFC                // Cargar R2 con el valor correspondiente al cero
            B    break                   // Saltar al final del bloque switch
case1:      MOV  R2,#0x60                // Cargar R2 con el valor correspondiente al uno
            B    break                   // Saltar al final del bloque switch
            ...
case9:      MOV  R2,#0xF6                // Cargar R2 con el valor correspondiente al nueve
            B    break                   // Saltar al final del bloque switch

break:      STRB R2,[R1],#1              // Guardar el elemento convertido
            B    lazo                    // Repetir el lazo de conversión
final:      STRB R2,[R1]                 // Guardar el fin de conversión en destino
stop:       BRA  stop
```

Uso de la memoria

- El compilador divide la memoria de datos en tres fragmentos:
  - Static: área de datos estática utilizada para variables globales
  - Heap: área de datos dinámica utilizada con las primitivas **`malloc`** y **`free`**.
  - Stack: área de datos dinámica utilizada para variables locales y llamadas a procedimientos.

Pila / Stack (repaso)

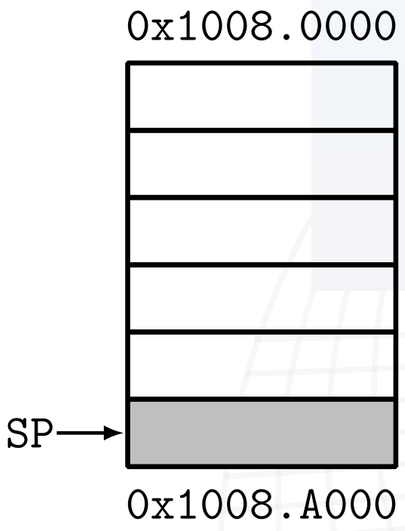
- Estructura de datos muy utilizada en general.
- ¿Ejemplos de pilas en el mundo real?
- Uso importante – sirve para explorar y saber cómo retornar (ejemplo: recorridos de árboles).
- ¿Cómo se implementa en Sw?

Implementación en ARM

- La memoria RAM hace de tabla.
- Emplea un puntero en CPU (SP=R13) para señalar ***el último lugar ocupado en la pila.***
- Instrucciones (PUSH y POP) para ingresar y retirar elementos de la pila.
- Cuando ingresamos un nuevo elemento en la pila el valor de SP se decrementa en cuatro (stack contiene palabras).
  - **Crece en el sentido de las direcciones decrecientes**.
- El puntero a la pila, SP, se inicializa cuando se hace RESET.
- Se puede inicilizar en otra parte con una instrucción LDR.
  - LDR R13,=direcc_pila

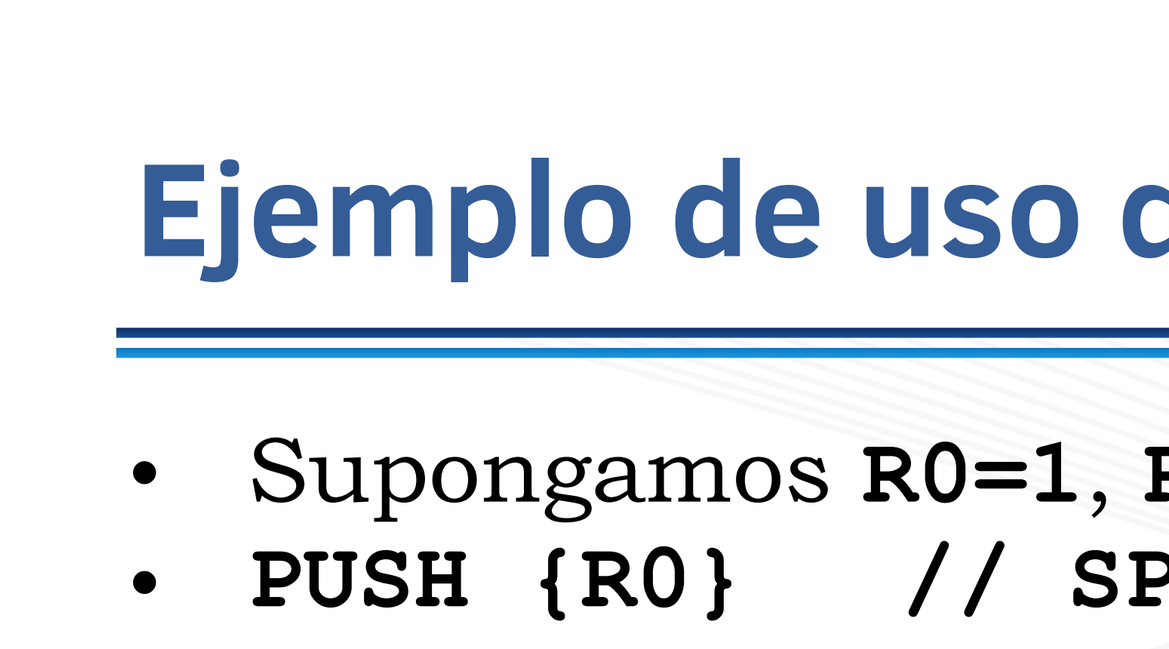
Ejemplo de uso del stack

- Supongamos **`R0=1`**, **`R1=2`**, **`R2=3`**



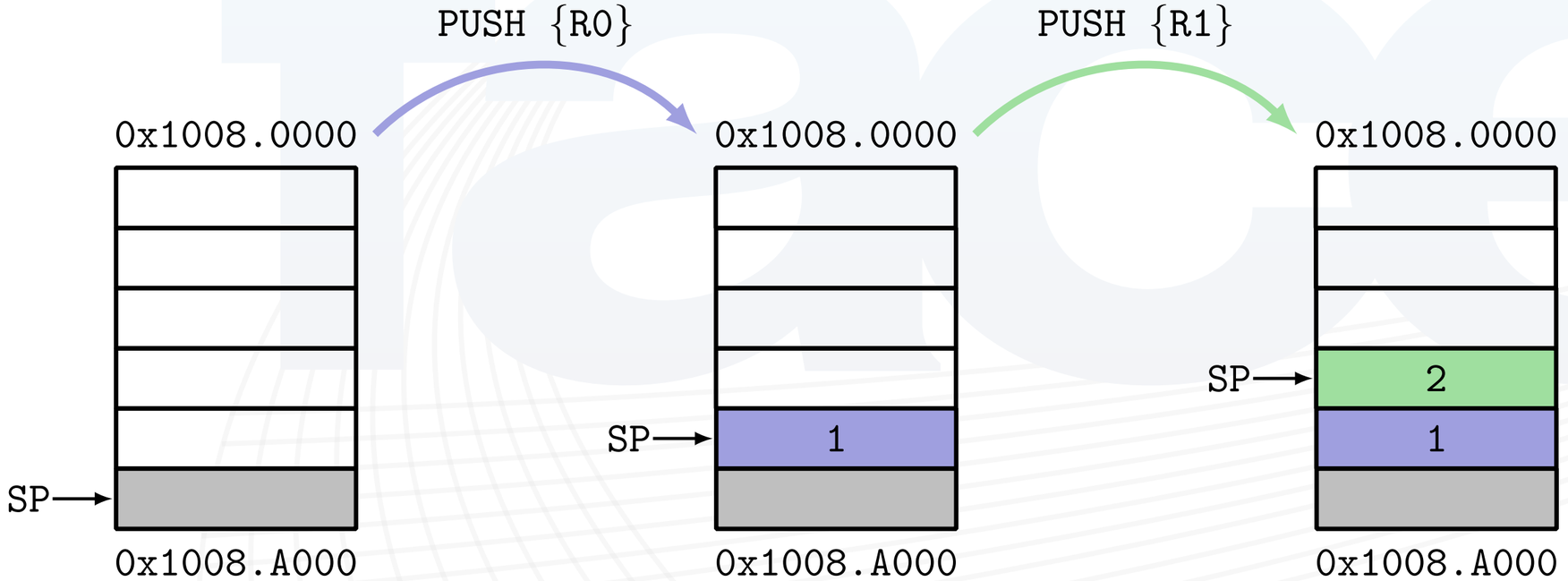
Ejemplo de uso del stack

- Supongamos **R0=1**, **R1=2**, **R2=3**
- **PUSH {R0} // SP=SP-4, M(SP)←R0**



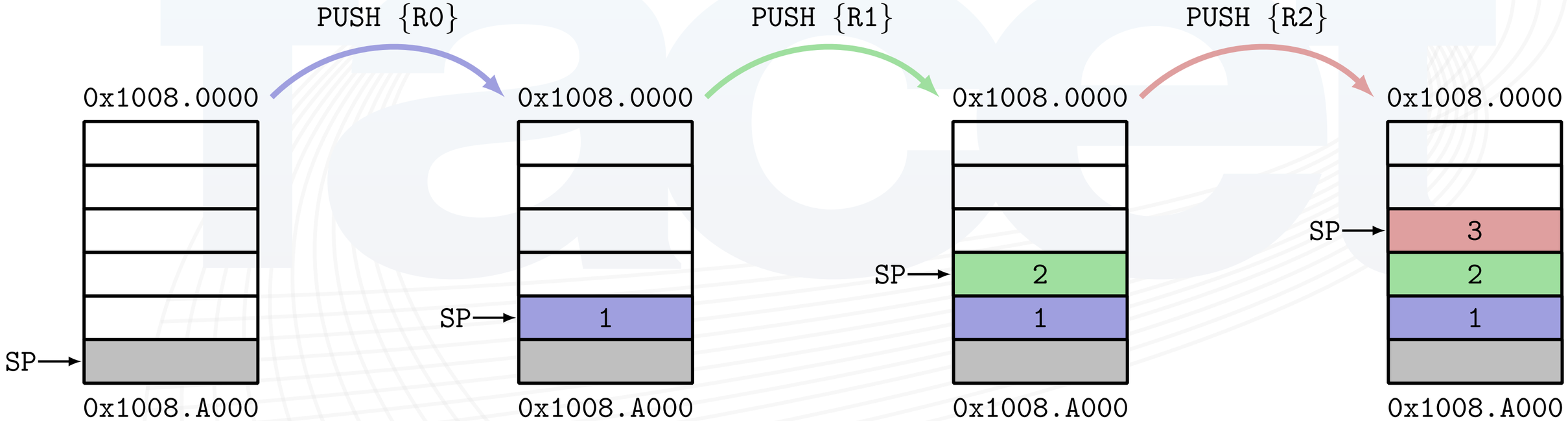
Ejemplo de uso del stack

- Supongamos **R0=1**, **R1=2**, **R2=3**
- **PUSH {R0} // SP=SP-4, M(SP)←R0**
- **PUSH {R1} // SP=SP-4, M(SP)←R1**



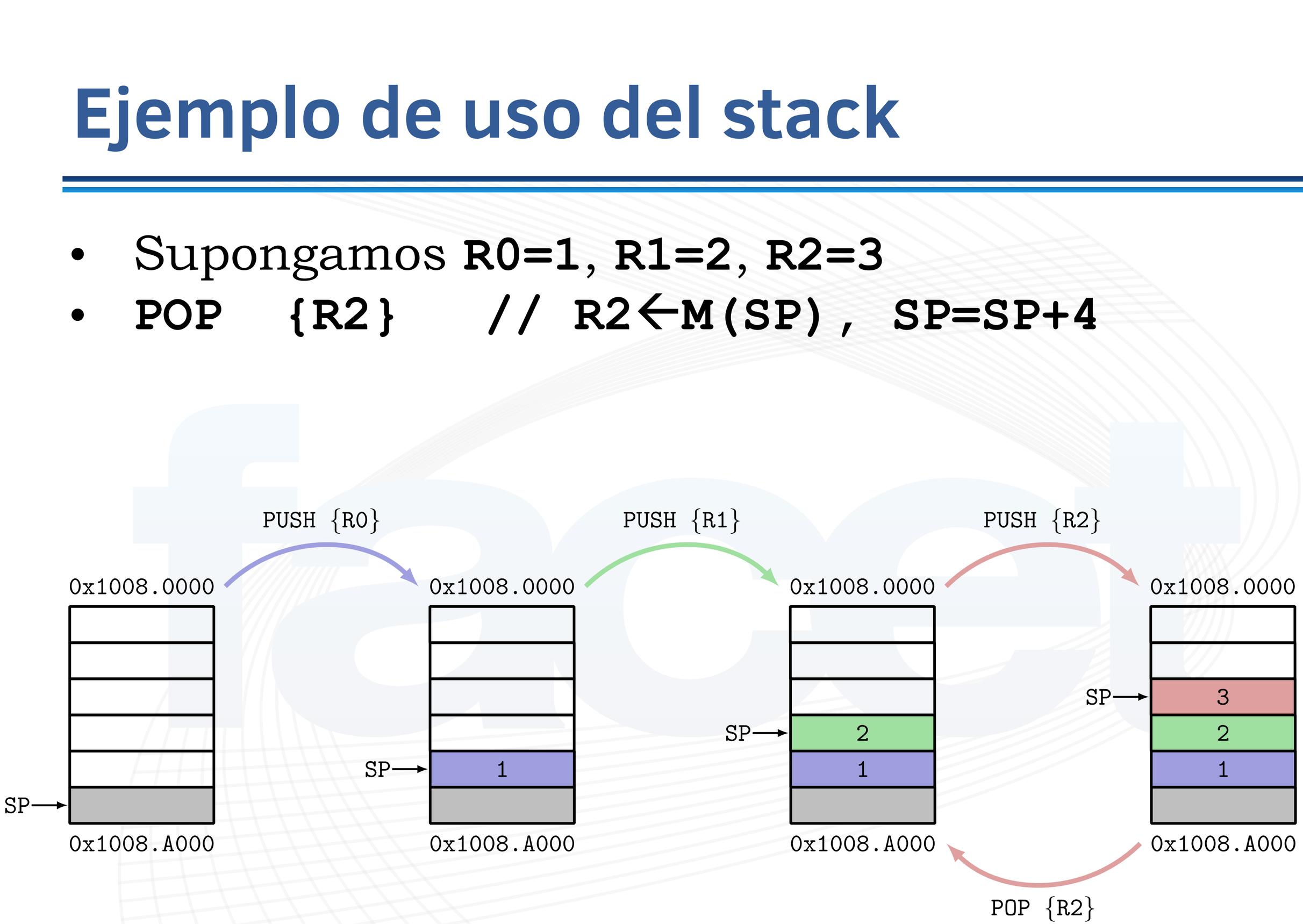
Ejemplo de uso del stack

- Supongamos **R0=1**, **R1=2**, **R2=3**
- **PUSH {R0} // SP=SP-4, M(SP)←R0**
- **PUSH {R1} // SP=SP-4, M(SP)←R1**
- **PUSH {R2} // SP=SP-4, M(SP)←R2**



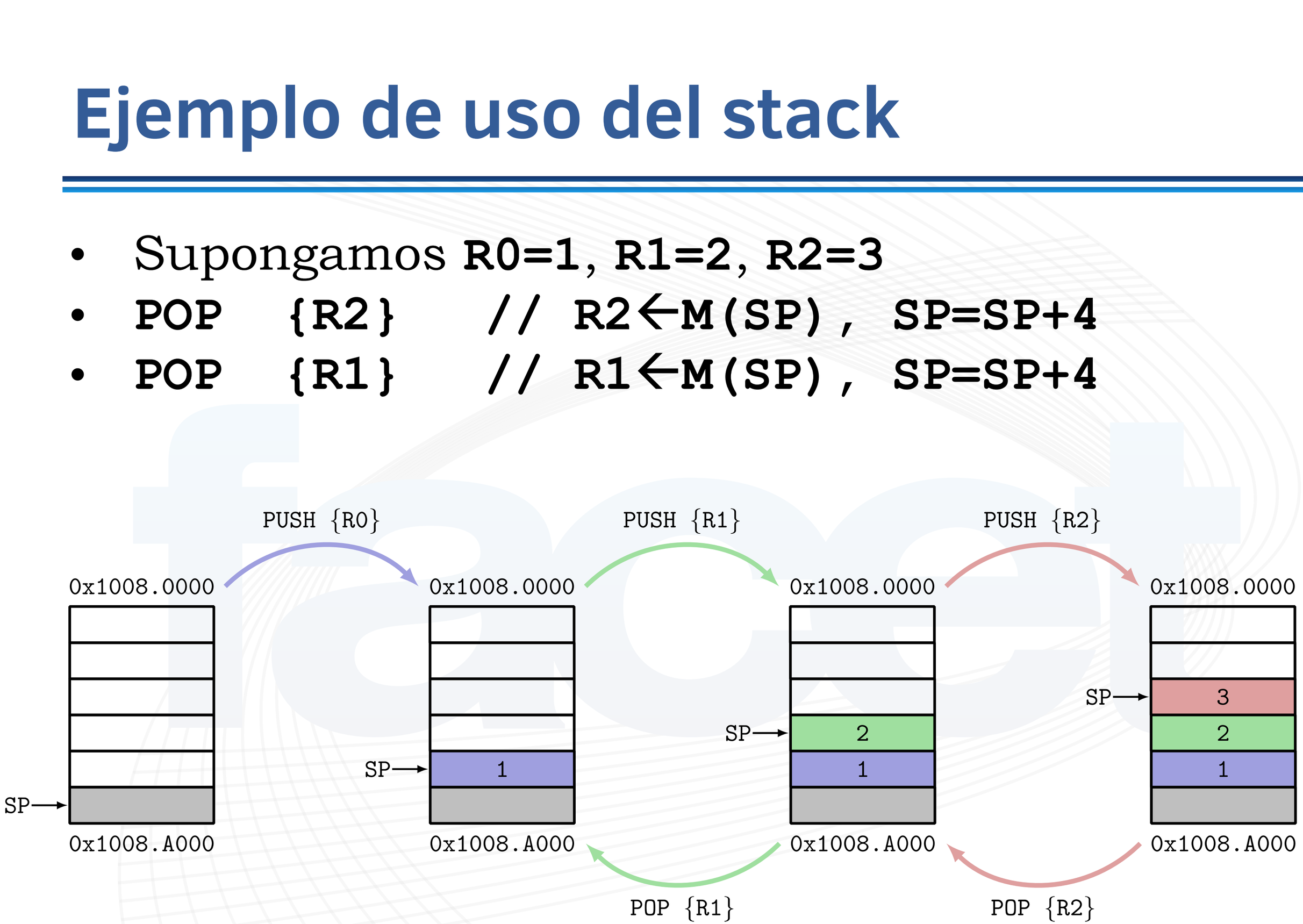
Ejemplo de uso del stack

- Supongamos **R0=1**, **R1=2**, **R2=3**
- **POP {R2} // R2←M(SP), SP=SP+4**



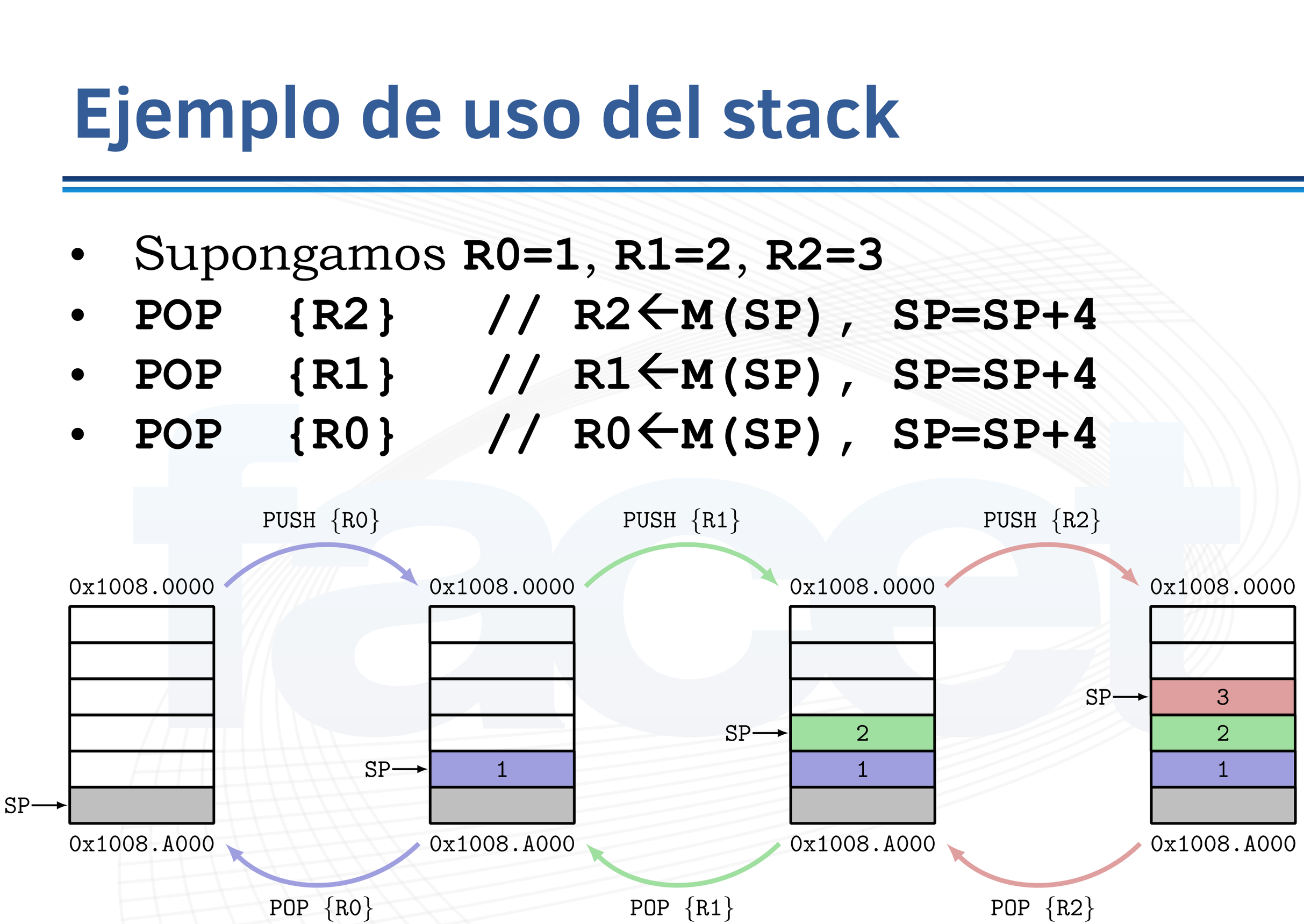
Ejemplo de uso del stack

- Supongamos **R0=1**, **R1=2**, **R2=3**
- **POP {R2} // R2←M(SP), SP=SP+4**
- **POP {R1} // R1←M(SP), SP=SP+4**



Ejemplo de uso del stack

- Supongamos **R0=1**, **R1=2**, **R2=3**
- **POP {R2} // R2←M(SP), SP=SP+4**
- **POP {R1} // R1←M(SP), SP=SP+4**
- **POP {R0} // R0←M(SP), SP=SP+4**



Implementaciones en otros CPUs

- El stack puede crecer para abajo o para arriba.
- El SP apunta al primer lugar vacío o al lleno.
  - ¿Inicialización?
- Que crezca en direcciones decrecientes permite usar memoria hacia abajo.
  - Poca probabilidad de que se superpongan con aplicaciones que almacenan hacia arriba.

Instrucciones PUSH y POP

- Se puede escribir una lista de registros luego de PUSH o POP
  - La instrucción se ejecuta para cada uno de ellos.
- PUSH múltiple guarda regs en orden decreciente de num registros. **Independientemente de cómo se escriban.**
- POP múltiple recupera en orden creciente de núm registros
- Ventaja: no hacer varios fetchs de la instrucción: programa más corto.
- Ej.

```
PUSH   {R0,R4-R7}
PUSH   {R0,R4,R5,R6,R7}
POP    {R0,R10,PC}
```

- **¿por qué orden creciente y decreciente?**

Variables globales

- Son comunes a todas las subrutinas o procedimientos.
- Antes de comentar la ejecución del programa se asigna un espacio suficiente en memoria.
- Este espacio de memoria permanece ocupado durante toda la ejecución del programa.
- Cualquier parte del programa puede acceder para leer o modificar el valor almacenado en la variable.

Variables locales

- Solo existen en un ámbito determinado, generalmente un subprograma.
- La asignación de memoria se realiza durante la ejecución del programa al ingresar en el ámbito de la variable.
- Solo el fragmento de programa que se encuentra en el mismo ámbito puede acceder a la variable.
- Al completar la ejecución del ámbito el espacio asignado es liberado y la variable deja de existir.

Uso de la pila para variables locales

- Al ingresar al ámbito de la variable se reserva espacio en la pila moviendo el puntero de pila tantos bytes como sea necesario.
- Al salir del ámbito se retorna el espacio reservado devolviendo al puntero de pila al valor original.
- El valor del puntero de pila antes de la reserva es un puntero a la variable local.

Uso de la pila para variables locales

- Ocupan lugar solo cuando se usan.
  - Dynamic Allocation/Release.
- Protección de datos: solo visibles mientras el procedimiento corre.
- Código reentrante.
- Código relocalizable.

Marco de Pila (Stack Frame)

- El puntero de pila puede variar y esto dificulta acceder a las variable locales.
- El puntero al marco de pila sirve para resolver ese problema:
  - Conserva el valor del puntero de pila antes de reservar el espacio para las variables locales.
  - Permite el acceso a las variables locales en forma indexada respecto al puntero de marco de pila que es constante.
- En ARM se utiliza el registro R7 como puntero de marco de pila

Subrutinas

- Generalmente los programas contienen bloques de código que se repiten.
- En estos bloques de código pueden variar algunos de los operandos (parámetros).
- Se puede ahorrar memoria (y tiempo de desarrollo) si estos bloques se escriben una vez y se ejecutan cada vez que se los requiere.
- Además estos bloques se podrían utilizar en otros programas.

Ventajas Subrutinas

- Diseño Top-Down
  - Se divide el Sistema en Módulos.
  - Cada módulo en submódulos.
  - Hasta llegar a módulos elementales.
  - Se arma (programa) de arriba hacia abajo
  - Se prueba al revés.
  - La cantidad de módulos debe ser manejable (máx<7)
- Trabajo en Equipo.
- Reutilización (Bibliotecas de Rutinas)
  - Requiere estandarización y transparencia.

Requerimientos.

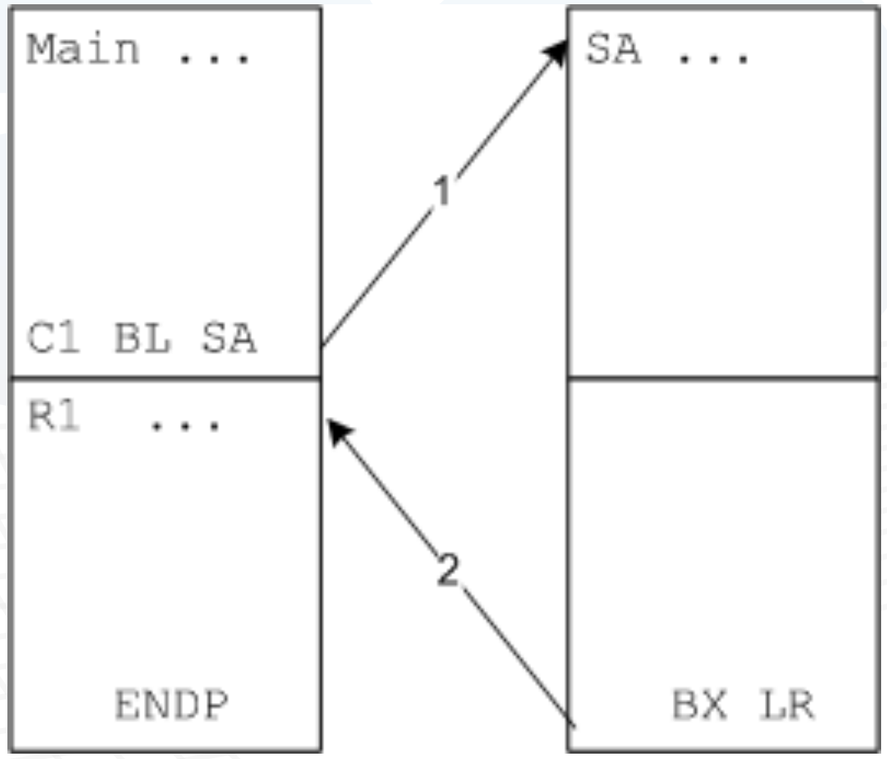
- Se las convoca de distintos lugares.
- Deben saber de dónde fueron llamadas para retornar a la instrucción siguiente al llamado.
  - Evidentemente un simple salto no es suficiente…
- Debe haber un mecanismo para pasar y devolver parámetros.

Implementación en ARM

- Instrucción de llamada: Branch and Link (BL)
  - **`BL etiqueta → LR = PC + 4, PC = etiqueta`**
  - Guarda PC+4 en registro LR (R14), “link register”
  - El salto es relativo al valor actual del PC
  - El alcance del destino es ±16MBytes
- También llamada indexada.
  - **`BLX Rn → LR = PC+4, PC = Rn`**
- OJO: Rn o etiqueta deben ser impares para indicar modo Thumb.
- Instrucción de Retorno
  - **`BX LR → PC = LR`**
  - Si al entrar en la Subrutina se hace PUSH LR, la instrucción POP PC también permite retornar.

Ejemplo

- ¿Esta implementación permite a un programa convocar a otra subrutina para hacer parte del trabajo?
- Si, en la ejecución de BL SA, PC=“R1”. A continuación LR=PC=“R1” y PC=“SA”. **La subrutina no debe modificar LR.**
- Con BX LR, PC=LR=“R1” y retorna al siguiente luar del llamado a SA

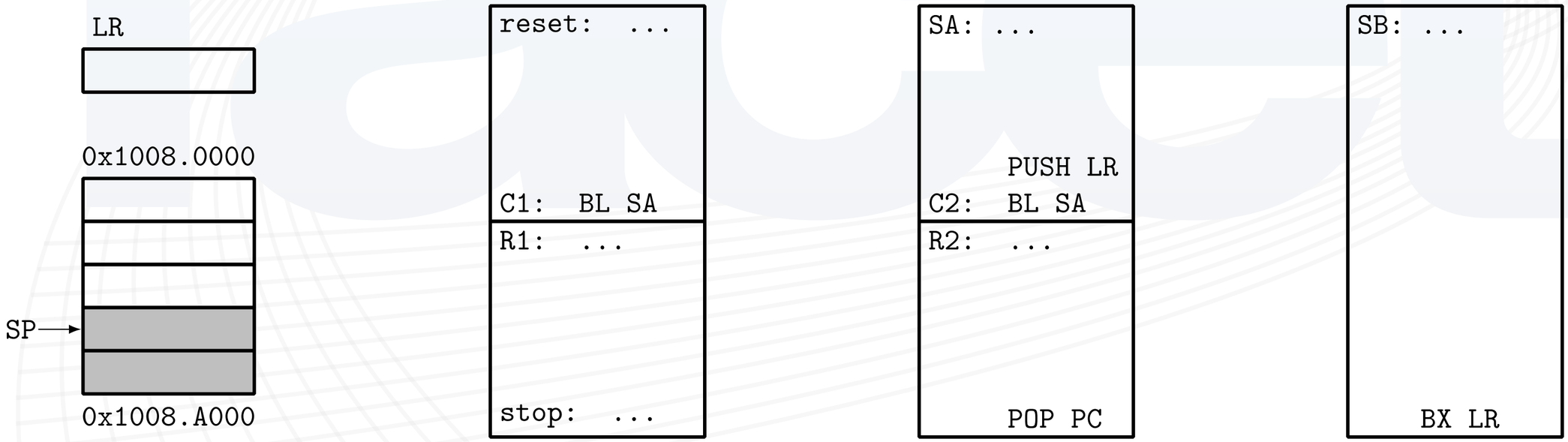


Subrutinas Anidadas

- ¿Esta implementación permite a una subrutina convocar a otra subrutina para hacer parte del trabajo?
- No, se pierde el contenido de LR en la llamada anidada.
- Para resolverlo la subrutina debe guardar LR en stack, PUSH LR.
- Cuando esta subrutina retorna a quien la llamó debe recuperar LR mediante POP LR y ejecutar luego BX LR.
- Una alternativa más corta a lo anterior es usar POP PC (entonces si LR estaba en Stack pasa directamente a PC y se retorna).
- Veamos cómo funciona:

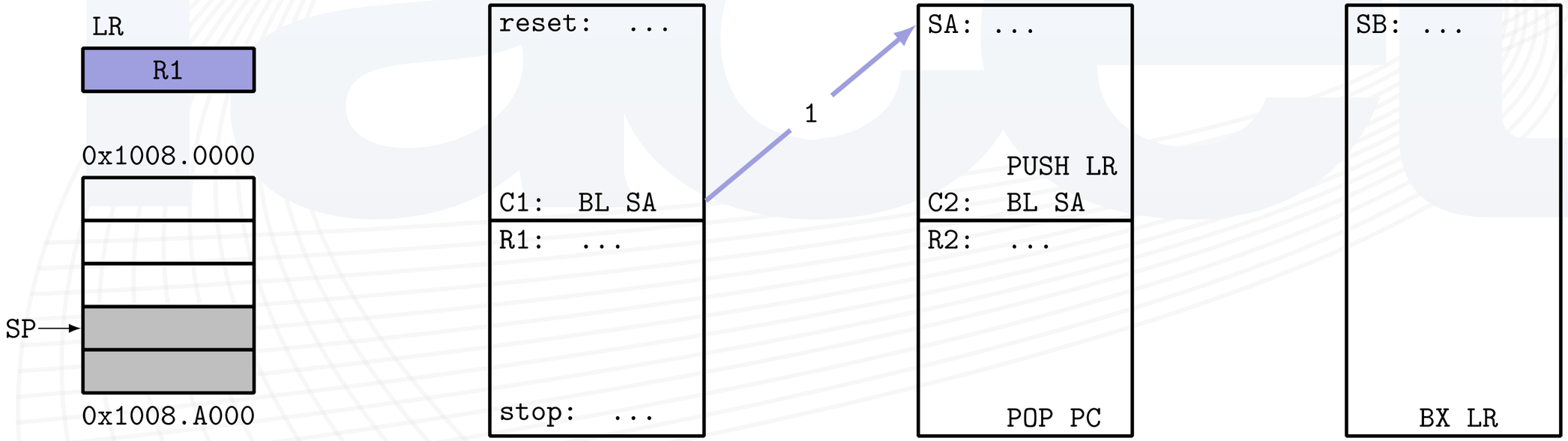
Subrutinas Anidadas

- ¿Esta implementación permite a una subrutina convocar a otra subrutina para hacer parte del trabajo?
- Si, el mecanismo de Stack lo permite sin problemas.



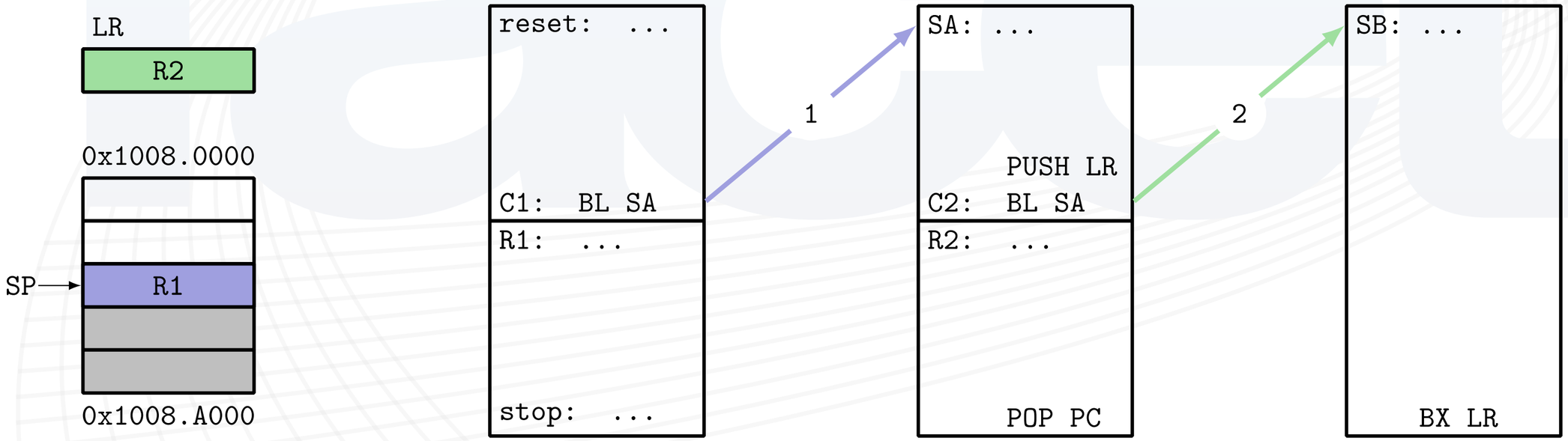
Subrutinas Anidadas

- ¿Esta implementación permite a una subrutina convocar a otra subrutina para hacer parte del trabajo?
- Si, el mecanismo de Stack lo permite sin problemas.



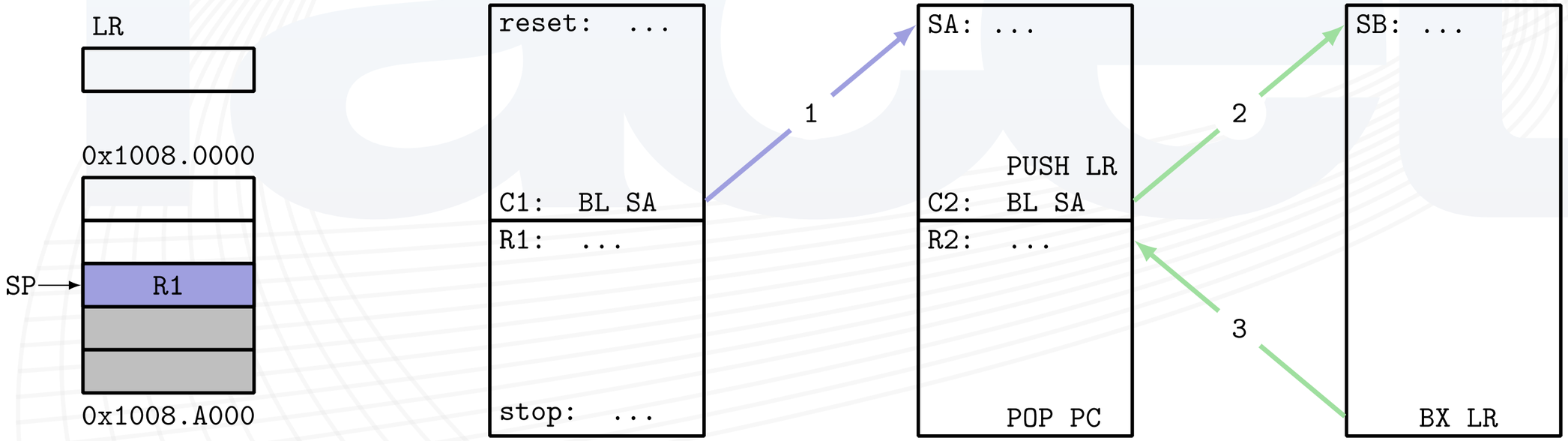
Subrutinas Anidadas

- ¿Esta implementación permite a una subrutina convocar a otra subrutina para hacer parte del trabajo?
- Si, el mecanismo de Stack lo permite sin problemas.



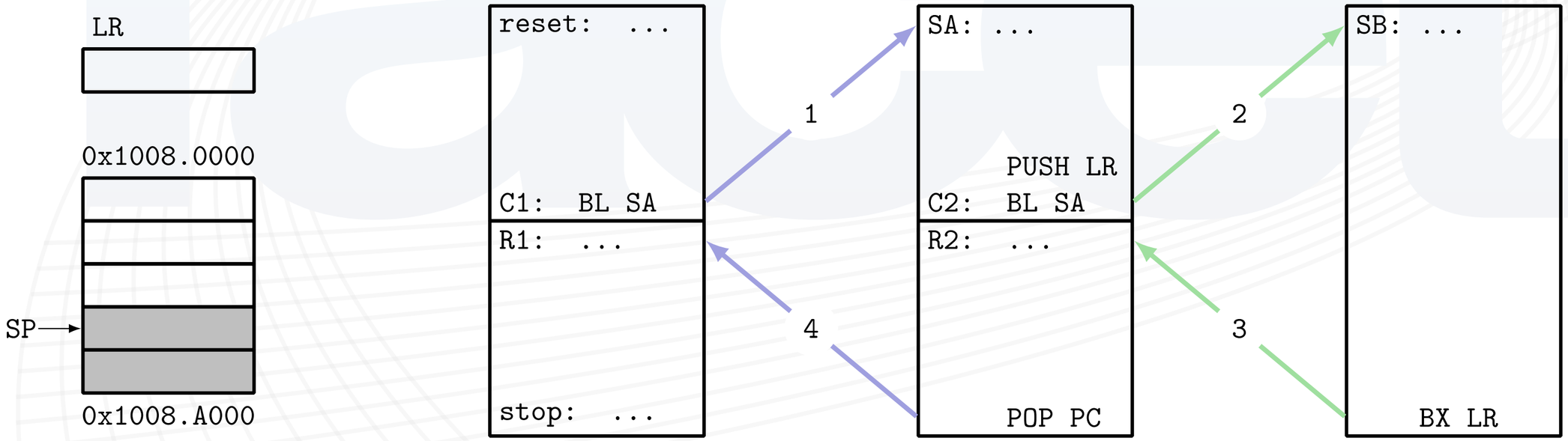
Subrutinas Anidadas

- ¿Esta implementación permite a una subrutina convocar a otra subrutina para hacer parte del trabajo?
- Si, el mecanismo de Stack lo permite sin problemas.



Subrutinas Anidadas

- ¿Esta implementación permite a una subrutina convocar a otra subrutina para hacer parte del trabajo?
- Si, el mecanismo de Stack lo permite sin problemas.
- **¿Qué pasa si en las rutinas se modifica el valor de retorno o el valor de SP?**



Reglas para el uso del stack

- Se debe hacer uso balanceado del stack, es decir debe haber el mismo número de push y pop
- Los accesos push o pop no deben realizarse fuera del área asignada de antemano por el diseñador.
- Ese área debe estar en Memoria RAM.
  - No superponerse con otros datos.
  - No superponerse con FLASH.
  - No superponerse con partes no ocupadas.

Pasaje de Parámetros

- Los parámetros son los datos de entrada y los resultados de salida de la Subrutina.
- Existen 4 formas de pasaje de parámetros:
  - Por Registros.
  - En Memoria.
  - En la Pila.
  - Por Referencias a Memoria.
- Cada caso posee ventajas y desventajas, hay que saber cuándo conviene utilizar cada uno.

Pasaje parámetros por registros

- Los parámetros son almacenados en los registros de propósito general antes de la llamada a la subrutina.
- Es la forma más sencilla y rápida para pasar parámetros.

Pasaje parámetros por registros

- Parámetro de Entrada: **R1** y **R2**.
- Rutina hace R0=R1+R2.
- Parámetro de Salida: **R0** .

```
reset:      MOV   R1,#57        // Cargo el primer sumando
            MOV   R2,#10        // Cargo el segundo sumando
            BL    suma          // Llamo a la subrutina
stop:       B     stop          // Lazo infinito para terminar

suma:       ADD   R0,R1,R2      // Realizo la suma
            BX    LR            // Retorno al programa principal
```

Pasaje parámetros por registros

- Ventaja:
  - Rapidez y comodidad, sobre todo si la variable ya está en un registro.
- Desventaja:
  - Pocos registros
  - Imposibilidad de pasar estructuras grandes como parámetro a un subrutina.

Paso de parámetros en memoria

- Los parámetros son almacenados en una zona de memoria conocida por el programa principal y la subrutina.
- Es el equivalente a utilizar variables globales.

Paso de parámetros en memoria

```
            .section .data      // Define la sección de variables (RAM)
base:       .space 12           // Reserva un espacio de tres palabras

            .section .text      // Define la sección de código (FLASH)
            .global reset       // Define el punto de entrada del código
reset:      LDR  R0,=base       // Apunto R0 al bloque de parametros
            MOV  R1,#57         // Cargo en R1 el primer operando
            STR  R1,[R0],#4     // Preparo el primer parametro del bloque
            MOV  R1,#10         // Cargo en R1 el segundo operando
            STR  R1,[R0],#4     // Preparo el segundo parametro del bloque
            BL   suma           // Llamo a la subrutina
stop:       B    stop           // Lazo infinito para terminar

suma:       LDR  R0,=base       // Apunto R0 al bloque de parametros
            LDR  R1,[R0],#4     // Cargo en R1 el primer operando
            LDR  R2,[R0],#4     // Cargo en R2 el primer operando
            ADD  R1,R2          // Realizo la suma
            STR  R1,[R0]        // Almaceno el resultado en el bloque
            BX   LR             // Retorno al programa principal
```

Paso de parámetros en memoria

- Ventaja:
  - No hay límite de parámetros a pasar.
  - Bueno para variables globales (la pueden acceder todos los procedimientos)
- Desventaja: Referencia fija a Memoria.
  - Si varias subrutinas comparten un mismo bloque de memoria, entonces esas subrutinas no se pueden llamar entre si.
  - Si hay n rutinas distintas que se puedan llamar entre sí… ¿cuántas áreas independientes debo tener? Polución de Memoria.

Paso de parámetros en la pila

- Los parámetros son almacenados en la pila antes de llamar a la subrutina.
- Los resultados producidos por la subrutina tambien se colocan en la pila, en un especio reservado por el programa principal al efectuar el llamado.

Paso de parámetros en la pila

```
reset:      MOV   R0,#57     // Cargo en R0 el primer operando
            MOV   R1,#10     // Cargo en R1 el segundo operando
            PUSH  {R0,R1}    // Preparo los parámetros en la pila
            BL    suma       // Llamo a la subrutina
            POP   {R0}       // Recupero el resultado de la pila
stop:       B     stop       // Lazo infinito para terminar

suma:       POP   {R0,R1}    // Recupero los parámetros de la pila
            ADD   R0,R1      // Realizo la suma
            PUSH  {R0}       // Almaceno el resultado en la pila
            BX    LR         // Retorno al programa principal
```

Paso de parámetros en la pila

```
reset:      MOV   R0,#57        // Cargo en R0 el primer operando
            MOV   R1,#10        // Cargo en R1 el segundo operando
            PUSH  {R0,R1}       // Preparo los parámetros en la pila
            BL    suma          // Llamo a la subrutina
            POP   {R0,R1}       // Recupero el resultado de la pila
stop:       B     stop          // Lazo infinito para terminar

suma:       PUSH  {LR}          // Almaceno la dirección de retorno
            LDR   R0,[SP,#4]    // Recupero el primer parámetro
            LDR   R1,[SP,#8]    // Recupero el primer parámetro
            ADD   R0,R1         // Realizo la suma
            STR   R0,[SP,#4]    // Almaceno el resultado
            POP   {PC}          // Retorno al programa principal
```

¿Por qué programa principal retira R0 y R1 del stack?

Paso de parámetros en la pila

- Ventajas
  - Número ilimitado de parámetros.
  - No hace falta m áreas de memoria.
  - El Programador no se preocupa por “cuidar áreas”.
  - Rutinas anidadas no presentan problemas.
- Desventajas
  - El programa principal emplea tiempo en poner los parámetros en el stack, especialmente si son muchos. Puede ser demasiado tiempo y lugar.
  - A veces se debe acceder a parámetros que no están en el tope de la pila.

Pasaje por referencia a memoria

- Los parámetros están en un bloque de memoria tal como los utiliza el programa principal.
- La dirección inicial, y la dimensión del bloque si hace falta, se ponen en registros o en la pila.
- Si la rutina entrega un bloque como resultado, también pasa la dirección incial y la dimensión del bloque en la pila o en registros.

Paso por referencia en registros

```
reset:    LDR   R0,=base      // Apunto R0 al bloque de parámetros
          MOV   R1,#57        // Cargo en R1 el primer operando
          STR   R1,[R0]       // Preparo el primer dato del bloque
          MOV   R1,#10        // Cargo en R1 el segundo operando
          STR   R1,[R0,#4]    // Preparo el segundo dato del bloque
          BL    suma          // Llamo a la subrutina
          LDR   R1,[R0,#8]    // Cargo el resultado en R1
stop:     B     stop          // Lazo infinito para terminar

suma:     LDR   R1,[R0]       // Cargo en R1 el primer operando
          LDR   R2,[R0,#4]    // Cargo en R1 el primer operando
          ADD   R1,R2         // Realizo la suma
          STR   R1,[R0,#8]    // Almaceno el resultado en el bloque
          BX    LR            // Retorno al programa principal
```

Paso por referencia en la pila

```
reset:    LDR   R0,=base     // Apunto R0 al bloque de parámetros
          MOV   R1,#57       // Cargo en R1 el primer operando
          STR   R1,[R0]      // Preparo el primer dato del bloque
          MOV   R1,#10       // Cargo en R1 el segundo operando
          STR   R1,[R0,#4]   // Preparo el segundo dato del bloque
          PUSH  {R0}         // Apilo la dirección del bloque
          BL    suma         // Llamo a la subrutina
          LDR   R1,[R0,#8]   // Cargo el resultado en R1
stop:     B     stop         // Lazo infinito para terminar

suma:     POP  {R0}          // Recupero la dirección del bloque
          LDR   R1,[R0]      // Cargo en R1 el primer operando
          LDR   R2,[R0,#4]   // Cargo en R1 el primer operando
          ADD   R1,R2        // Realizo la suma
          STR   R1,[R0,#8]   // Almaceno el resultado en el bloque
          BX    LR           // Retorno al programa principal
```

Pasaje por referencia a memoria

- Ventajas:
  - No hay límite en cantidad de parámetros.
  - No hay problemas de anidamientos si uso la pila.
  - El programa principal no gasta tiempo moviendo datos de un lugar a otro.
- Desventajas:
  - Complicado para pocos datos, y lento en ese caso.
- ¿Si la rutina usa unos pocos datos y también un bloque de n datos?
  - Nadie dice que sólo se debe emplear un único método.
  - Se lo puede combinar con otros métodos, vistos antes.

Paso de parámetros (standard)

- Varias formas de pasar parámetros.
- Tanto registros como stack tienen ventajas, pero se pueden combinar:
  - Algunas entradas podrían venir en registros
  - Otras en Stack.
  - Resultado en registro o stack.
- Convención para llamados a procedimientos.
  - Application Binary Interface (ABI) – compatible C.
  - Permite que una rutina puede llamarse desde otro lenguaje.
  - Permite poder usar bibliotecas de rutinas de terceras partes.

ARM Arch. Procedure Call Standard (AAPCS)

- Convención ABI para todas las arquitecturas ARM.
- Registros R0, R1, R2, y R3 para pasar los primeros cuatro parámetros de entrada. Resto en Stack
- ABI obliga a poner el parámetro de retorno en R0.
- Las funciones son libres para modificar R0–R3, R12 es reservado para uso del Assembler.
- Si una función requiere R4-R11, deberá guardar primero los valores de los registros a usar en el stack. Antes de retornar, debe recuperar los viejos valores del stack.
- Por la convención ABI, el primer parámetro se pasa en Registro R0, el segundo en R1 y así...

Rutinas transparentes

- Son las que cumplen con ABI.
- Lo que no se puede pasar en los registros apropiados se pasan en la pila.
- Es transparente porque el programa principal sabe qué puede cambiar y qué no.
- ¿Los flags pueden cambiar?
- Todo lo que no se documente se basa en ABI.
  - Por ejemplo se puede documentar que a R1 la rutina no la cambia, y el programa lo asume como parte de la transparencia.
  - Si no se documenta nada, vale ABI en general.

Subrutinas recursivas

- Rutinas que se llaman a sí mismas.
  - De manera directa o indirecta.
- Existen problemas de naturaleza recursiva. Hay que atacarlos siguiendo su naturaleza.
  - Ej. Torres de Hanoi.
- Dos condiciones para que funcione bien:
  - Código Puro.
  - Areas de Datos Independientes por cada llamado.
- Areas de Datos Independientes ⇛ Stack.
- Código Puro ⇛ norma mínima de higiene.

Abstracción

- Tiene por objetivo independizar las subrutinas del programa principal.
- Una persona puede emplear la rutina, sabiendo cuál es su función y sin saber cómo se la programa. Se **abstrae** del detalle.
- Cada subrutina debe respetar la convención ABI.
- Debe declarar si altera Memoria o PSR.
- El programa convocante tomará su prevención.
- Se pueden integrar las subrutinas en librerías y reutilizarlas en futuros proyectos.

Abstracción - Driver

- La abstracción nos permite modularizar nuestro código y brinda la opción de exponer al usuario lo que queremos que vea y esconder lo que no deseamos que toque.
- Un manejador de un dispositivo (device driver) es un buen ejemplo del uso de abstracción para exponer rutinas públicas que deseamos que el usuario del driver emplee.
- Se usan rutinas privadas para esconder detalles internos del driver.