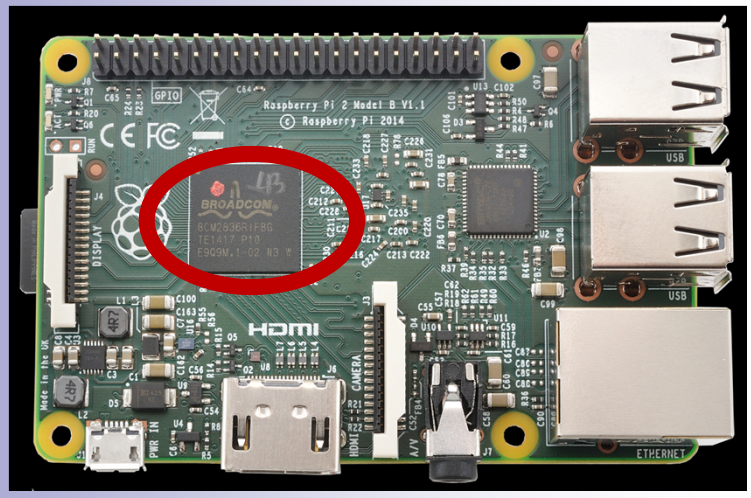
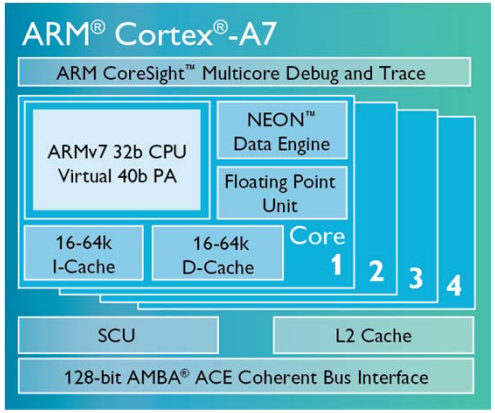
LENGUAJE ENSAMBLADOR ARM

PROGRAMACION EN ENSAMBLADOR

Introducción arquitectura ARM

- Desarrollado en la década de los 80 por Advanced RISC Machines (ahora ARM Holdings)
- Se venden cerca de 10 billones de procesadores ARM al año
- Prácticamente todos los móviles y tablets los usan
- Se usa en servidores, móviles, cámaras, robots, coches, etc…
- Se utiliza en la Raspberry Pi 2, con el SOC BCM2836





Introducción arquitectura ARM

- Se basa en una arquitectura RISC
  - **Reduced Instruction Set Computer (RISC),** con un pequeño número de instrucciones simples
  - Otras arquitecturas, como la familia de Intel x86, son CISC **(Complex Instruction Set Computers)**
- ARM incluye solo instrucciones simples y comunes (que se usan frecuentemente)
- Esto implica un HARDWARE simple y rápido a la hora de decodificar y ejecutar instrucciones
- Instrucciones más complejas (y menos frecuentes) se realizan usando múltiples instrucciones simples

Lenguaje ensamblador ARM

- Instrucción ensamblador ARM

Código C	Código ensamblador ARM
`a = b + c;`	`ADD a, b, c`

- **ADD:** mnemonico – indica la operación a realizar (instrucción).
- **b, c:** operandos fuente
- **a:** operando destino

Operandos de las instrucciones

- ¿Dónde se encuentran los operandos de las instrucciones ensamblador?
  - Tipos de operandos:
    - Operandos constantes (también llamados inmediatos):
      - En la propia instrucción
    - Operandos variables
      - En registros del procesador
      - En la memoria

Operandos en Registros (ARM 32 bits)

- Los registros son más rápidos que la memoria
- ARM tiene 16 registros (+1 registro de estado)
- Cada registro es de 32 bits (almacena 32 bits)
- Nos vamos a centrar en la arquitectura ARM de 32 bits → va a operar con datos de 32 bits
- Tipos de datos según su tamaño:
  - **Byte**: 8 bits
  - **Halfword**: 16 bits (2 bytes)
  - **Word**: 32 bits (4 bytes)

Operandos en Registros (ARM 32 bits)

Registro	Sinónimo	Especial	Preservado?	Uso
r16		CPSR	No	Current Program Status Register
r15		PC	No	Program Counter
r14		LR	Si	Link Register
r13		SP	Si	Stack Pointer
r12		IP	No	Intra-Procedure-call scratch register
r11	v8		Si	Variable register 8
r10	v7		Si	Variable register 7
r9	v6		Si	Platform register (meaning defined by platform)
r8	v5		Si	Variable register 5
r7	v4		Si	Variable register 4
r6	v3		Si	Variable register 3
r5	v2		Si	Variable register 2
r4	v1		Si	Variable register 1
r3	a4		No	Argument / scratch register 4
r2	a3		No	Argument / scratch register 3
r1	a2		No	Argument / result / scratch register 2
r0	a1		No	Argument / result / scratch register 1

Ejemplos instrucciones con registros

Código C	Código ensambaldor ARM
	`@ R0 = a, R1 = b, R2 = c`
`a = b + c`	`ADD R0, R1, R2  @ R0←R1+R2`
`a = c – b`	`SUB R0, R2, R1  @ R0←R2-R1`
`a = a & b`	`AND R0, R0, R1  @ R0←R0&R1`
`b = b | a`	`ORR R1, R1, R0  @ R0←R2|R1`
`a = b`	`MOV R0, R1  @ R0←R1`
`a = ~b`	`MVN R0, R1  @ R0←NOT R1`

Operandos constantes/inmediatos

- El valor del operando se especifica en la propia instrucción (modo de direccionamiento inmediato)
- Ejemplos de instrucciones con operando inmediato:

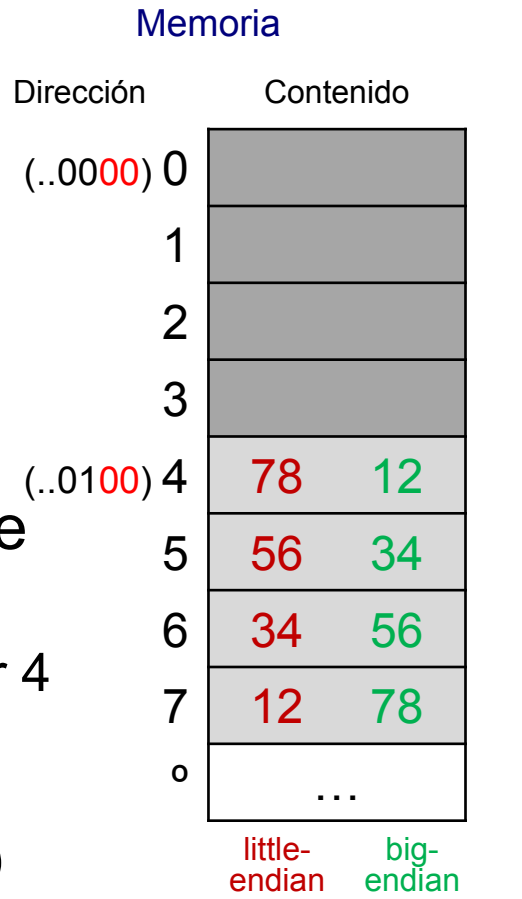
Código C	Código ensambaldor ARM
	`@ R0 = a, R1 = b`
`int a = 23;`	`MOV R0, #23 @ R0←23`
`int b = 0x45;`	`MOV R1, #0x45 @ R1←0x45`
`a = a + 4;`	`ADD R0, R0, #4 @ R0←R0+4`
`b = a – 12;`	`SUB R1, R0, #12 @ R1←R0-12`

Precisión de la constante <= 8 bits (ver formato instrucción)

Operandos en memoria

- Todos los datos no caben en registros
- Almacenar datos en memoria:
  - Mayor capacidad pero más lento el acceso
- Se organiza en palabras de 4 bytes
- Direccionamiento a nivel de byte (cada byte una dirección):
  - Dos palabras consecutivas están separadas por 4 posiciones
  - Las instrucciones están en direcciones múltiplo de 4 (los dos bits menos significativos serán 00)
    - PC (32 bits) se incrementará de 4 en 4
- ¿Dirección de un dato de 32 bits (word)?
  - **Little-endian**: dirección del byte menos significativo (Least Significant Byte, LSB)
  - **Big-endian**: dirección del byte más significativo (Most Significant Byte, MSB)

Por defecto



MEM(4) = 0x12345678

Ejemplo instrucciones con memoria

- LOAD (LDR): Cargar en un registro el valor almacenado en una posición de memoria (R←M)
- Direccionamiento relativo:
  - Registro explícito
  - Desplazamiento inmediato o en registro
- Auto-indexado del registro:
  - Pre-incrementado/decrementado
  - Post-incrementado/decrementado

Código C **Código ensambaldor ARM**

```
a = b;      @ R0 = a, b en memoria
            LDR R0, [R1]       @ R0←MEM(R1)
            LDR R0, [R1,#12]   @ R0←MEM(R1+12)
Auto-Post-Incr. → LDR R0, [R1],R2    @ R0←MEM(R1); R1←R1+R2
Auto-Pre-Decr. → LDR R0, [R1,#-4]!   @ R0←MEM(R1-4); R1←R1-4
```

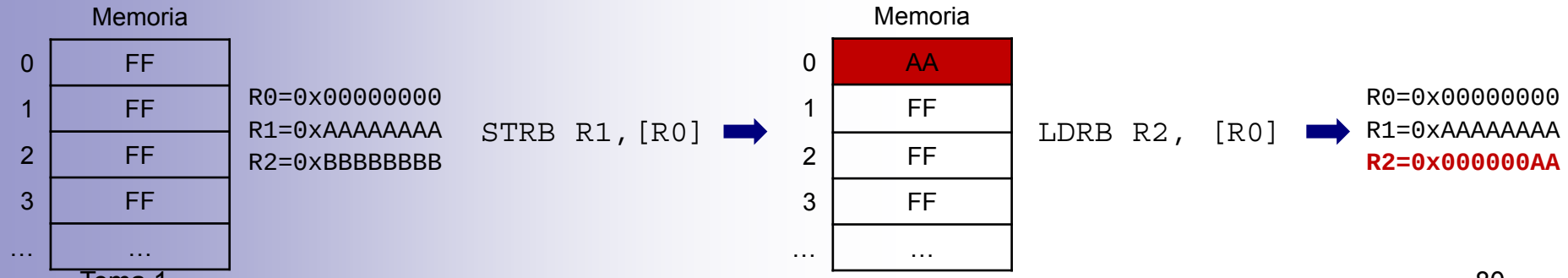
Registro destino — Registro base explícito — Desplaz.

Ejemplo instrucciones con memoria

- STORE (STR): Cargar en una posición de memoria el valor almacenado en un registro (M←R)
- Mismos modos de direccionamiento que LDR

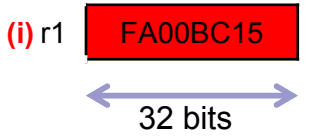
```
STR R0, [R1]        @ MEM(R1)←R0
```

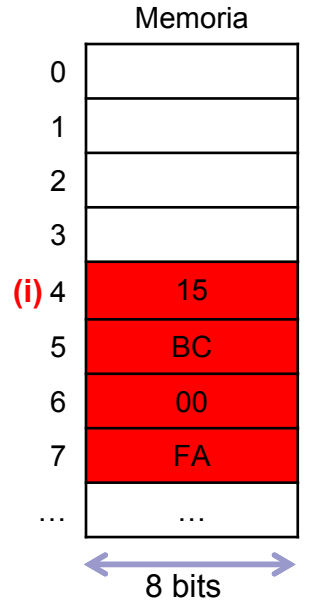
- LDR y STR mueven palabras de 32 bits (Word)
- Existen versiones para Bytes
  - LDRB: Lee solo un byte de la dir especificada y lo almacena en el LSB del registro destino, poniendo a 0 el resto (más significativos)
  - STRB: Almacena el LSB del registro en la posición de memoria especificada.



Tipos de datos C

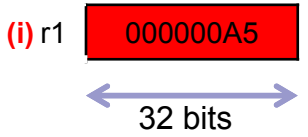
- Escalares tamaño palabra
  - int i = 0xFA00BC15
    - Almacenado en registro: ejemplo r1
      - El valor de la variable "i" se mantiene en r1
      - Operaciones directas: add r1, r1, #4
    - Almacenado en memoria: i es la dirección de memoria donde se almacena la variable
      - Cargar dirección de la variable: `ldr r0,=i`
      - Cargar el valor de la variable: `ldr r1,[r0]`
      - Operar con el registro usado (r1)
      - Almacenar el valor en memoria: `str r1,[r0]`

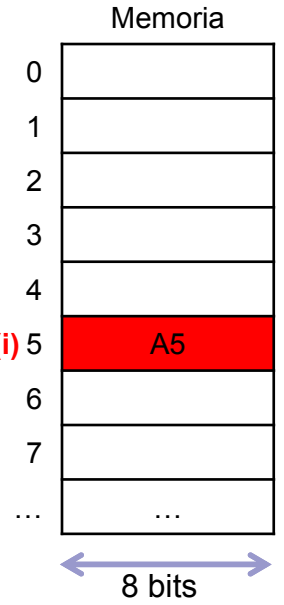




Tipos de datos C

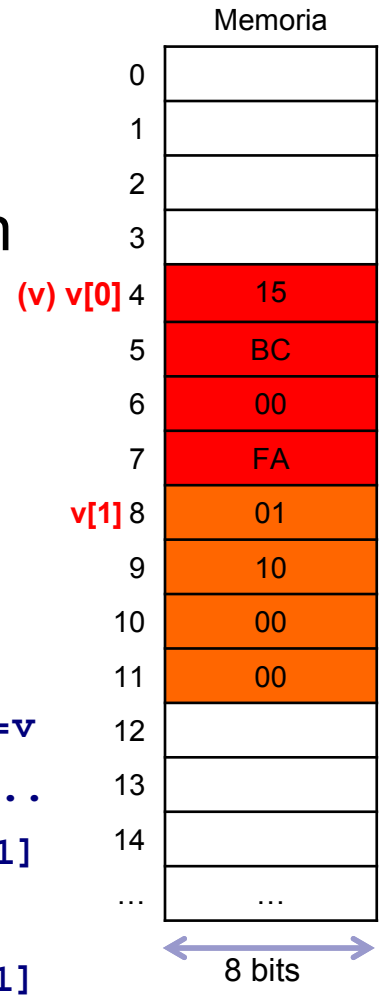
- Escalares tamaño byte
  - char i = 0xA5
    - Almacenado en registro: ejemplo r1
      - El valor de la variable "i" se mantiene en r1
      - Operaciones directas: add r1, r1, r2
    - Almacenado en memoria: i es la dirección de memoria donde se almacena la variable
      - Cargar dirección de la variable: `ldr r0,=i`
      - Cargar el valor de la variable: `ldrb r1,[r0]`
      - Operar con el registro usado (r1)
      - Almacenar el valor en memoria: `strb r1,[r0]`





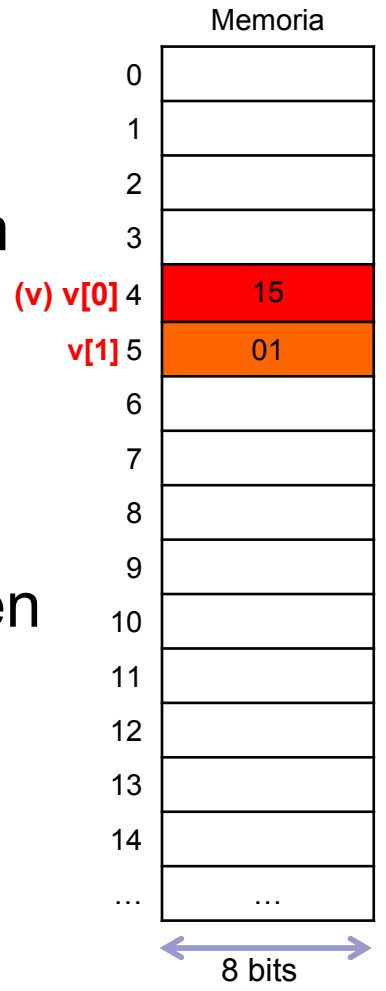
Tipos de datos C

- Vectores tamaño palabra (array)
  - int v[2] = {0xFA00BC15,0x00001001}
    - Almacenado en memoria: v es la dirección de memoria donde se almacena el primer elemento del vector (v[0])
    - Para acceder a posición i del vector, desplazar i*tamaño_element(4) sobre v
    - Ej: cargar en r0 dirección del vector (v) y en r1 el desplazamiento del índice del elemento (i)
      - Cargar dirección del vector v: `ldr r0,=v`
      - Calcular el desplazamiento del índice (r1): `... r1...`
      - Cargar el valor de la posición: `ldr r2,[r0,r1]`
      - Operar con el registro usado (r2)
      - Almacenarlo en memoria: `str r2,[r0,r1]`



Tipos de datos

- Vectores tamaño byte(array)
  - char v[2] = {0x15,0x01}
    - Almacenado en memoria: v es la dirección de memoria donde se almacena el primer elemento del vector (v[0])
    - Para acceder a posición i del vector, desplazar i*tamaño_element(1) sobre v
    - Ej: cargar en r0 dirección del vector (v) y en r1 el índice del elemento (i)
      - Cargar dirección del vector v: `ldr r0,=v`
      - Cargar el valor del índice:
        ```
        ldr r2,=i
        ldr r1,[r2]
        ```
      - Cargar el valor de la posición: `ldrb r2,[r0,r1]`
      - Operar con el registro usado (r2)
      - Almacenarlo en memoria: `strb r2,[r0,r1]`



Instrucciones de procesado de datos

- Estas instrucciones sólo trabajan con registros (en el caso de **`Operand2`** tben. con inmediatos pequeños)
- Son:
  - Aritméticas: **`ADD` `ADC` `SUB` `SBC` `RSB` `RSC`**
  - Comparaciones: **`CMP` `CMN` `TST` `TEQ`**
  - Lógicas: **`AND` `ORR` `EOR` `BIC`**
  - Movimiento de datos: **`MOV` `MVN`**

Sintaxis:

`<Operation>{<cond>}{S} Rd, Rn, Operand2`

- Las comparaciones sólo actualizan los flags (no hay un operando destino Rd)
- En los movimientos de datos no hay Rn

- El segundo operando pasa por un barrel shifter antes de llegar a su entrada de la ALU.

Instrucciones de procesado de datos

Categoría	Instrucción	Ejemplo
Aritméticas	`ADD r1, r2, Operand2`	r1 = r2 + Operand2
	`ADC r1, r2, Operand2`	r1 = r2 + Operand2 + C
	`SUB r1, r2, Operand2`	r1 = r2 – Operand2
	`SBC r1, r2, rOperand2`	r1 = r2 – Operand2 – C
	`RSB r1, r2, Operand2`	r1 = Operand2 – r2
	`RSC r1, r2, Operand2`	r1 = Operand2 – r2 – C
Lógicas	`AND r1, r2, Operand2`	r1 = r2 AND Operand2
	`ORR r1, r2, Operand2`	r1 = r2 OR Operand2
	`EOR r1, r2, Operand2`	r1 = r2 XOR Operand2
	`BIC r1, r2, Operand2`	r1 = r2 AND NOT(Operand2)
Comparaciones	`CMP r1, Operand2`	Flag.cond.= r1 – Operand2
	`CMN r1, Operand2`	Flag.cond.= r1 + Operand2
	`TST r1, Operand2`	Flag.cond.= r1 AND Operand2
	`TEQ r1, Operand2`	Flag.cond.= r1 EOR Operand2
Movimiento de datos	`MOV r1,Operand2`	r1 = Operand2
	`MVN r1,Operand2`	r1 = !Operand2

Operaciones con variables escalares en registro

- Código C:
  - **f = (g+h)-(i+j);**
- Variables almacenadas en registros:
  - f →r4, g→r5, h→r6, i→r7,
  - j→r8, tmp1→r9, tmp2→r10
- Código ensamblador:

Operaciones con variables escalares en registro

- Código C:
  - **f = (g+h)-(i+j);**
- Variables almacenadas en registros:
  - f →r4, g→r5, h→r6, i→r7,
  - j→r8, tmp1→r9, tmp2→r10
- Código ensamblador:

```
add r9, r5, r6     @ tmp1←g+h
add r10, r7, r8    @ tmp2←i+j
sub r4, r9, r10    @ f←tmp1-tmp2
```

Carga de constantes de 32 bits

- Código C:
  - **g=123456789;**
- Variables escalares en registros
  - g→r4
- La instrucción MOV solo puede usar inmediatos con precisión 8 bits (no vale para ese número)
- Código ensamblador:

```
ldr r4, =123456789      @ g←123456789
```

- También para cargar dirección de una etiqueta:

```
ldr r4, =label     @ g←dir mem de label
```

Acceso a vectores

- Código C:
  - **g=h+A[i];**
- Variables escalares en registros
  - g→r4, h→r5, i→r6
- Vector A de elementos de tamaño palabra
  - Etiqueta A →dirección comienzo de A en memoria
- Código ensamblador:

Instrucciones lógicas

- `AND`
- `ORR`
- `EOR` **(XOR)**
- `BIC` **(Bit Clear)**
- `MVN` **(MoVe and NOT)**

Instrucciones lógicas: ejemplos

	Source registers			
R1	0100 0110	1010 0001	1111 0001	1011 0111
R2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly code		Result			
AND R3, R1, R2	R3	0100 0110	1010 0001	0000 0000	0000 0000
ORR R4, R1, R2	R4	1111 1111	1111 1111	1111 0001	1011 0111
EOR R5, R1, R2	R5	1011 1001	0101 1110	1111 0001	1011 0111
BIC R6, R1, R2	R6	0000 0000	0000 0000	1111 0001	1011 0111
MVN R7, R2	R7	0000 0000	0000 0000	1111 1111	1111 1111

Instrucciones lógicas: aplicación

- AND, BIC: forzar a 0 bits específicos

  Ejemplo: **Enmascarar** todos los bits de un registro, excepto el byte menos significativo

Instrucciones lógicas: aplicación

- AND, BIC: forzar a 0 bits específicos

  Ejemplo: **Enmascarar** todos los bits de un registro, excepto el byte menos significativo

  ```
  0xF234012F AND 0x000000FF  = 0x0000002F
  0xF234012F BIC 0xFFFFFF00 = 0x0000002F
  ```

Instrucciones lógicas: aplicación

- AND, BIC: forzar a 0 bits específicos

  Ejemplo: **Enmascarar** todos los bits de un registro, excepto el byte menos significativo

  ```
  0xF234012F AND 0x000000FF  = 0x0000002F
  0xF234012F BIC 0xFFFFFF00 = 0x0000002F
  ```

- ORR: forzar a 1 bits bits específicos

  Ejemplo: forzar 1 el byte menos significativo de un registro

Instrucciones lógicas: aplicación

- AND, BIC: forzar a 0 bits específicos

  Ejemplo: **Enmascarar** todos los bits de un registro, excepto el byte menos significativo

  0xF234012F AND 0x000000FF = 0x0000002F

  0xF234012F BIC 0xFFFFFF00 = 0x0000002F

- ORR: forzar a 1 bits bits específicos

  Ejemplo: forzar 1 el byte menos significativo de un registro

  0xF2340000 ORR 0x000000FF = 0xF23400FF

Más inst. de procesado de datos

- Desplazamientos / rotaciones
- Multiplicación

Inst. desplazamiento/rotación

- LSL: logical shift left
- LSR: logical shift right
- ASR: arithmetic shift right
- ROR: rotate right

Inst. desplazamiento: ejemplo 1

- **Immediato** para el *shift amount* (5-bits)
  - *Shift amount*: 0-31

	Source register			
R5	1111 1111	0001 1100	0001 0000	1110 0111

Assembly Code		Result			
`LSL R0, R5, #7`	R0	1000 1110	0000 1000	0111 0011	1000 0000
`LSR R1, R5, #17`	R1	0000 0000	0000 0000	0111 1111	1000 1110
`ASR R2, R5, #3`	R2	1111 1111	1110 0011	1000 0010	0001 1100
`ROR R3, R5, #21`	R3	1110 0000	1000 0111	0011 1111	1111 1000

Inst. desplazamiento: ejemplo 2

- **Registro** para el *shift amount* (se utilizan los 8 bits menos significativos)
- *Shift amount*: 0-255

Source registers

R8	0000 1000	0001 1100	0001 0110	1110 0111
R6	0000 0000	0000 0000	0000 0000	0001 0100

Assembly code	Result				
`LSL R4, R8, R6`	R4	0110 1110	0111 0000	0000 0000	0000 0000
`ROR R5, R8, R6`	R5	1100 0001	0110 1110	0111 0000	1000 0001

Multiplicación y división

- Hay dos clases de producto:
  - Resultado de 32-bits (en un ARM7TDMI ejecuta en 2 - 5 ciclos)
    - `MUL r0, r1, r2` `@ r0 = r1 * r2`
    - `MLA r0, r1, r2, r3` `@ r0 = (r1 * r2) + r3`
  - Resultado de 64-bits, con versión para enteros y naturales (con y sin signo)
    - Estas instrucciones utilizan dos registros destino:
    - `[U|S]MULL r4, r5, r2, r3` `@ r5:r4 = r2 * r3`
    - `[U|S]MLAL r4, r5, r2, r3` `@ r5:r4 = (r2 * r3) + r5:r4`
- La mayoría de los núcleos ARM no ofrecen instrucciones para la división de enteros
  - Las operaciones de división las realizan rutinas de librerías, o se utilizan desplazamientos.

Ejecución condicional

- No siempre se quiere ejecución secuencial
  - If/else, for loops, while loops, llamada función …
- ARM posee **flags de condición** que:
  - Pueden ser modificados por las instrucciones
  - Pueden ser usados para ejecutar condicionalmente una instrucción
  - Se almacenan en el registro de estado (CPSR)

Flag	Name	Description
N	**N**egative	Instruction result is negative
Z	**Z**ero	Instruction results in zero
C	**C**arry	Instruction causes an unsigned carry out
V	o**V**erflow	Instruction causes an overflow

Ejecución condicional

- Instrucciones que modifican los flags
  - Instrucciones de comparación
    - `CMP r1, Operand2` (carga N,Z,C,V resultantes de operación r1-Operand2)
    - `CMN r1, Operand2` (carga N,Z,C,V resultantes de operación r1+Operand2)
    - `TST r1, Operand2` (carga N,Z,C,V resultantes de operación r1 AND Operand2)
    - `TEQ r1, Operand2` (carga N,Z,C,V resultantes de operación r1 EOR Operand2)
- Añadiendo sufijo "s" al mnemónico de la instrucción (no todas las instrucciones lo soportan):
  - `ADDS r1,r2,r3` (r1←r2+r3 y además carga N,Z,C,V resultantes de dicha operación)
  - `SUBS r1,r2,r3` (r1←r2-r3 y además carga N,Z,C,V resultantes de dicha operación)

Ejecución condicional

- Ejecución de instrucciones condicional al estado de los flags
  - Añadir mnemotécnico de condición al mnemotécnico de la instrucción

    Ejemplo:
    ```
    CMP    R1, R2
    SUBNE  R3, R5, R8
    ADDEQ  R3, R8, R5
    ```
  - **NE** y **EQ**: mnemotécnicos de condición
  - SUB solo se ejecuta si R1 ≠ R2
  - ADD solo se ejecuta si R1 = R2

Mnemotécnicos de condición

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\bar{Z}$
0010	CS / HS	Carry set / Unsigned higher or same	C
0011	CC / LO	Carry clear / Unsigned lower	$\bar{C}$
0100	MI	Minus / Negative	N
0101	PL	Plus / Positive of zero	$\bar{N}$
0110	VS	Overflow / Overflow set	V
0111	VC	No overflow / Overflow clear	$\bar{V}$
1000	HI	Unsigned higher	$\bar{Z}C$
1001	LS	Unsigned lower or same	$Z\ OR\ \bar{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\bar{Z}(\overline{N \oplus V})$
1101	LE	Signed less than or equal	$Z\ OR\ (N \oplus V)$
1110	AL (or none)	Always / unconditional	ignored

Instrucciones de salto

- Permiten la ejecución NO secuencial de instrucciones
- Tipos de instrucciones de salto:
  - **Branch (`B label`)**
    - Salta a la instrucción de la etiqueta "label"
  - **Branch and link (`BL label`)**
    - Salta y enlaza (se verá después)
- Los dos tipos pueden ser incondicionales o condicionales (mnemotécnicos de condición)

Aplicación de ejecución condicional y saltos

- Sentencia “if” en código C:

```
if (a < b) { c = 10; }
else { c = 20; }
```

- Variables almacenadas en registros:
  - a→r4, b→r5, c→r6
- Código ensamblador:

TRADUCCIÓN DE ESTRUCTURAS DE ALTO NIVEL A ARM

PROGRAMACION EN ENSAMBLADOR

Estructuras tipo IF

- Código C

```
if (i==j)
   f=g+h;
f=f-i;
```

- Ensamblador

```
    cmp r1,r2
    addeq r3,r4,r5
    sub r3,r3,r1
```

```
if (i==j)
     f=g+h;
else
     f=g-h;
endif
```

```
     cmp r1,r2
     bne ELS
THN: add r3,r4,r5
     b END
ELS: sub r3,r4,r5
END: ...
```

Bucle WHILE

- Código C

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x   = 0;


while (pow != 128) {

  pow = pow * 2;
  x = x + 1;
}
```

- Ensamblador

Bucle FOR

- Código C

```
// adds numbers from 0-9
int i;
int sum = 0;


for (i=0; i<10; i=i+1){
   sum = sum + i;
}
```

- Ensamblador

Acceso a Array en bucle

- Código C

```
int i;
int scores[200];
…
for (i=0; i<200; i=i+1)
  scores[i]=scores[i]+10;
```

- Ensamblador

```
@ R0 = array base address, R1=i
```

Llamadas a función: actores

- En una llamada existen dos actores:
  - **Caller:** Función que llama (`main` en ejemplo)
  - **Callee:** Función llamada (`sum` en ejemplo)

C Code

```
void main()
{
  int y;
  y = sum(42, 7);
  ...
}
int sum(int a, int b)
{
  return (a + b);
}
```

Llamadas a función: actores

- **Caller:**
  - Pasa los **argumentos** al “callee”
  - Salta a la primera instrucción del “callee”
- **Callee:**
  - **Realiza** el trabajo de la función
  - **Devuelve** un resultado al “caller”
  - **Vuelve** a la instrucción siguiente del punto de llamada en el “caller”
  - **No debe sobreescribir** registros o memoria del “caller”

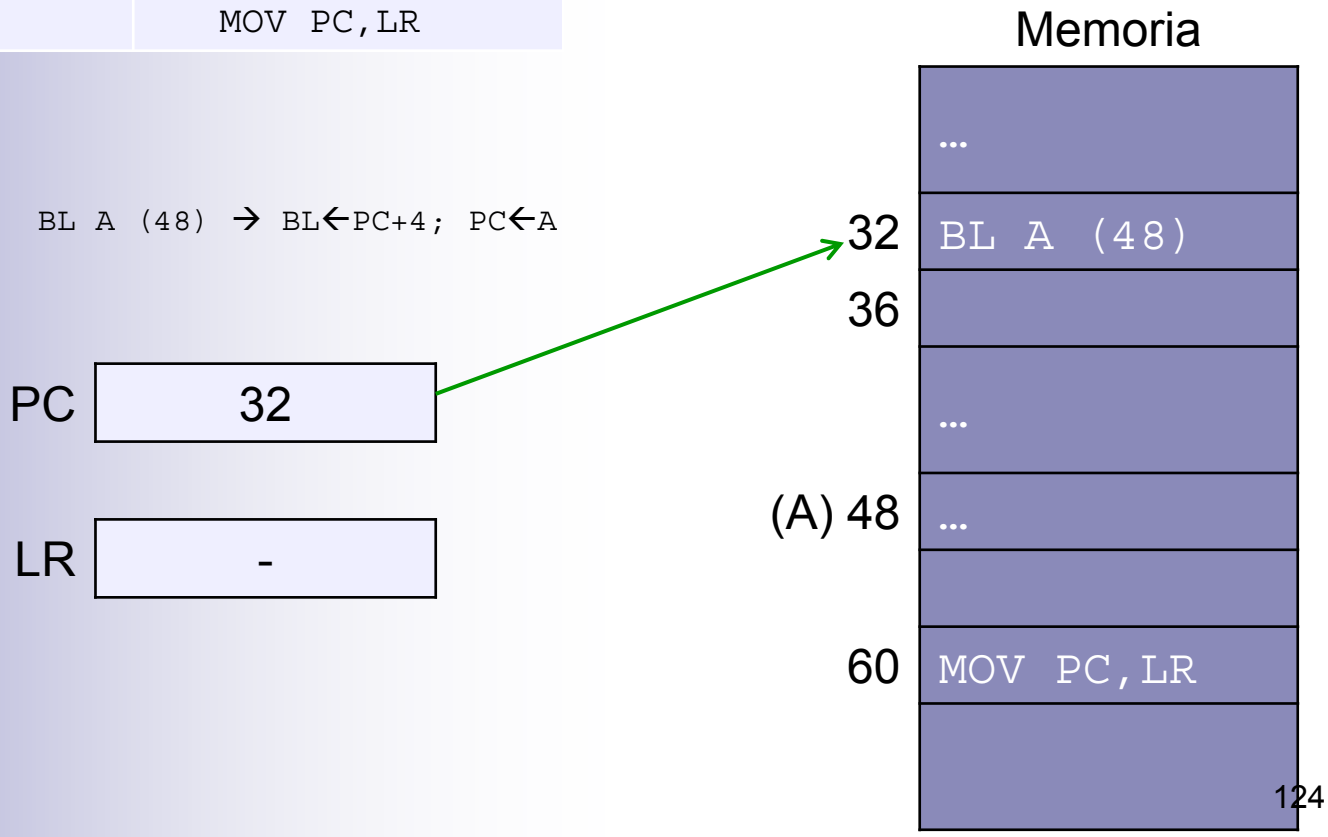
Llamadas a función: instrucciones

- El repertorio de instrucciones ARM permite:
  - Llamada a función:
    - Instrucción: **`BL`** (Branch and Link).
    - La posición de retorno se guarda en el registro LR (Link Register, r14).
  - Regreso del procedimiento:
    - Continuar por la instrucción siguiente a la de llamada (mover a PC el valor de LR)
    - Instrucción: **`MOV PC, LR`**

Llamadas a función: ejemplo

- Llamada a función A y retorno.

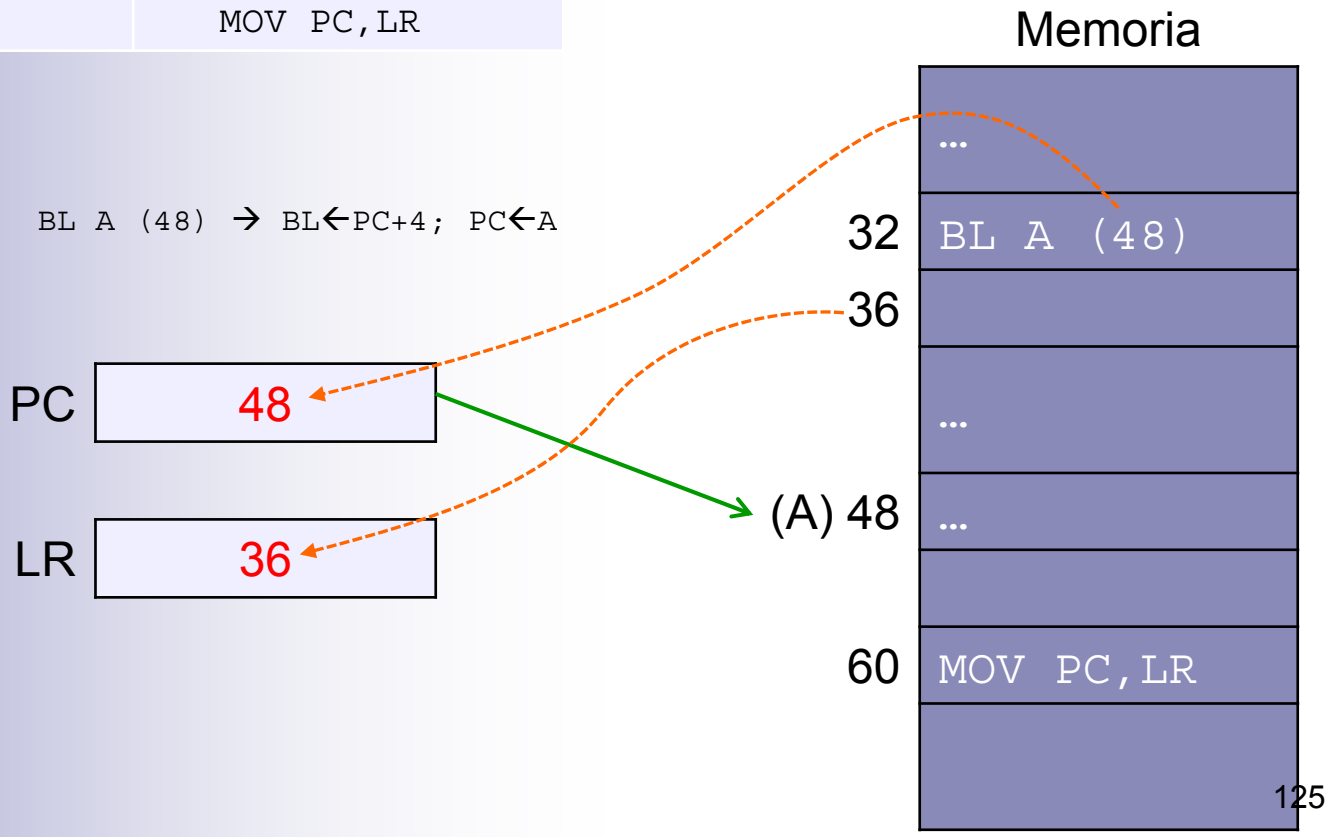
Bloque de código main	Bloque de código A
… BL A …	A: … … MOV PC,LR



Llamadas a función: ejemplo

- Llamada a función A y retorno.

Bloque de código main	Bloque de código A
… BL A …	A: … … MOV PC,LR

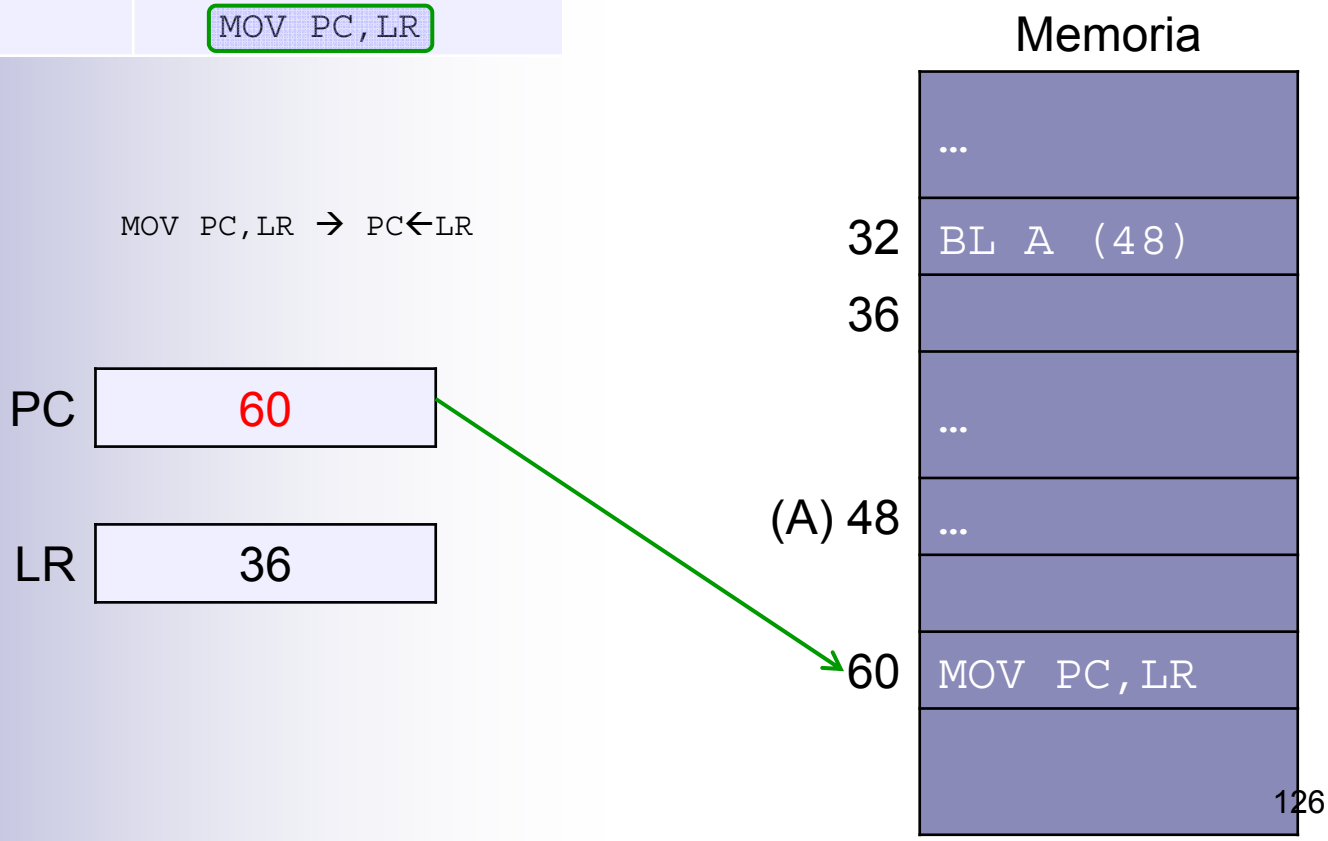


Llamadas a función: ejemplo

- Llamada a función A y retorno.

Bloque de código main	Bloque de código A
… `BL A` …	`A:` … … `MOV PC,LR`

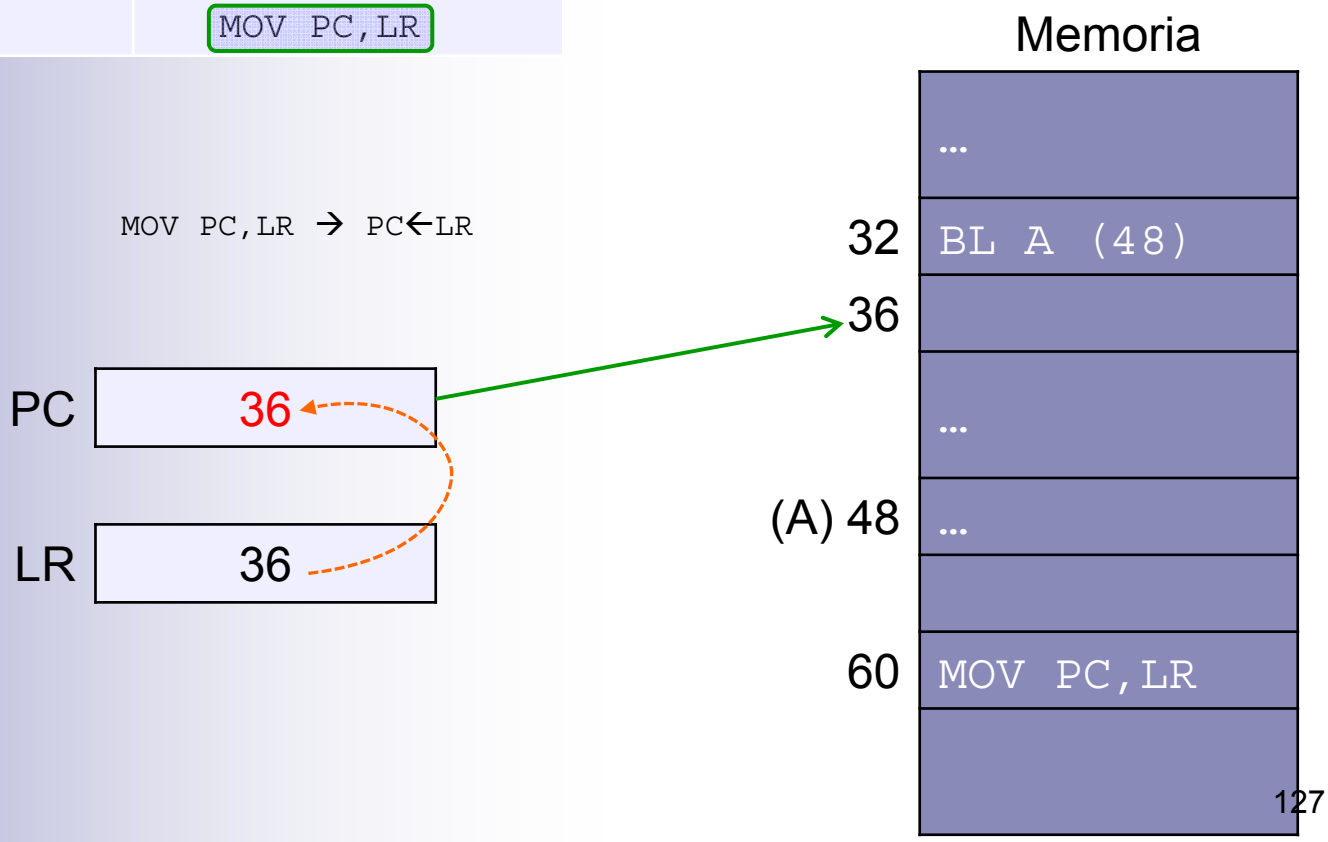
`MOV PC,LR` → `PC←LR`



Llamadas a función: ejemplo

- Llamada a función A y retorno.

Bloque de código main	Bloque de código A
… `BL A` …	`A:` … … `MOV PC,LR`



Convenios: paso de parámetros

- Pasar parámetros a las funciones:
  - Se pasan por registro (r0-r3, a1-a4)
    - Si hay más de 4 argumentos se pasan por la pila.
  - El resultado se devuelve el los registros (r0-r1, a1-a2).
    - r0 para resultados de 32 bits
    - r0-r1 para resultados de 64 bits

Código C

```
int main(){
  int y;
  y += sum(2, 3); // 2 arguments
}
int sum(int a, int b){
  int result;
  result = a + b;
  return result; // return value
}
```

ARM Assembly Code

```
MAIN:   @ R4 = y
  MOV R0, #2      @ argument 0 = 2
  MOV R1, #3      @ argument 1 = 3
  BL SUM          @ call function
  ADD R4,R4,R0    @ y = y+returned value
  …
SUM:    @ R4 = result
  ADD R4, R0, R1   @ result = a + b
  MOV R0, R4       @ return value in R0
  MOV PC, LR       @ return to caller
```

Convenios: problema sobreescritura de registros

- Una función “callee” puede escribir en registros útiles de la función “caller”:
  - En el ejemplo: `R4` (var “y” en main y “result” en sum)
  - En una llamada a función dentro de otra función se sobreescribe `LR` (**Porblema de anidamiento de llamadas**)
    - ¿cómo se retorna de función `MAIN`?

Código C

```
int main(){
  int y;
  y += sum(2, 3); // 2 arguments
}
int sum(int a, int b){
  int result;
  result = a + b;
  return result; // return value
}
```

ARM Assembly Code

```
MAIN:   @ R4 = y
  MOV R0, #2     @ argument 0 = 2
  MOV R1, #3     @ argument 1 = 3
  BL SUM         @ call function
  ADD R4,R4,R0   @ y = y+returned value
  …
SUM:    @ R4 = result
  ADD R4, R0, R1   @ result = a + b
  MOV R0, R4       @ return value in R0
  MOV PC, LR       @ return to caller
```

Convenios: problema sobreescritura de registros

- Convenio de uso de registros en llamadas:

Preservados *Callee-Saved*	**No preservados** *Caller-Saved*
R4-R11	**R12**
R14 (LR)	**R0-R3**
R13 (SP)	**CPSR**
stack above SP	**stack below SP**

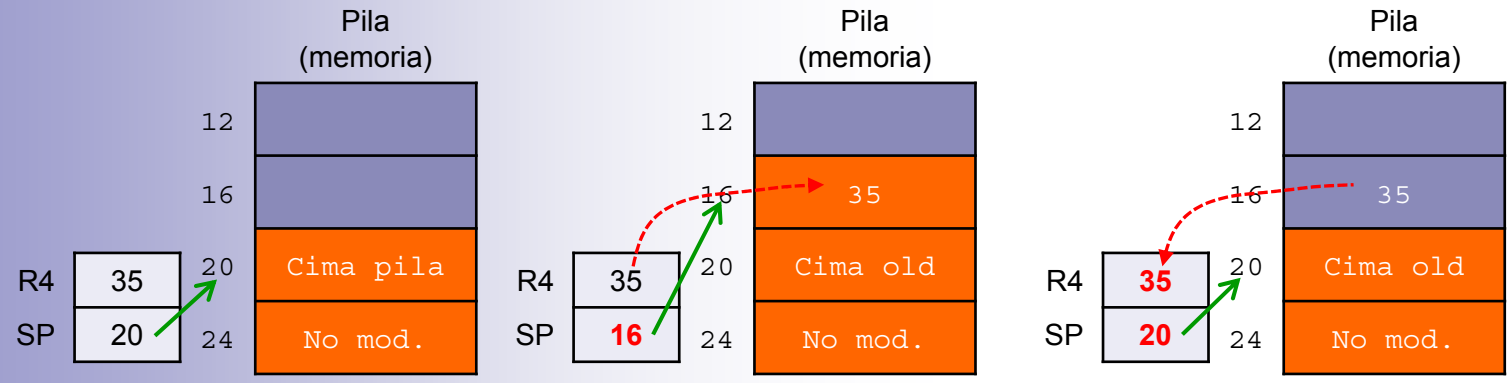
- ¿Dónde salvar valor de los resgistros?
  - Memoria temporal – PILA
  - Tipo LIFO (Last-In-First-Out)
  - Dinámica: Se expande cuando se necesita espacio, se contrae cuando ya no se necesita

La PILA

- Zona de memoria apuntada por el puntero de pila (Stack Pointer): Almacenado en registro `SP` (`R13`)
- Crece hacia abajo (direcciones más bajas de memoria)
- Salvar – **PUSH**
  - Decrementar SP
  - Almacenar relativo a SP
- Restaurar – **POP**
  - Cargar relativo a SP
  - Incrementar SP

```
@PUSH
SUB SP,SP,#4
STR R4,[SP]
```

```
@POP
LDR R4,[SP]
ADD SP,SP,#4
```



Ejemplo de función hoja (no llama)

- Código C:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - Argumentos `g,h,i y j` en `R0,R1,R2 y R3`
  - `f` en `R4` (por tanto hay que guardarlo en pila)
  - Resultado en `R0`

Ejemplo de función hoja

- ARM code:

```
leaf_example:
   SUB SP, SP, #4
   STR R4, [SP]
```

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

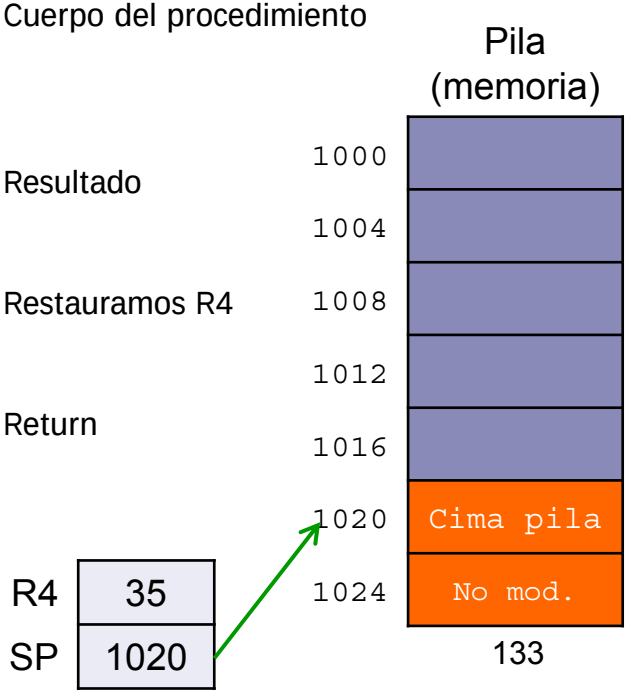
Guardamos R4 en la pila

Cuerpo del procedimiento

Resultado

Restauramos R4

Return



Ejemplo de función hoja

- ARM code:

```
leaf_example:
   SUB SP, SP, #4
   STR R4, [SP]
```

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

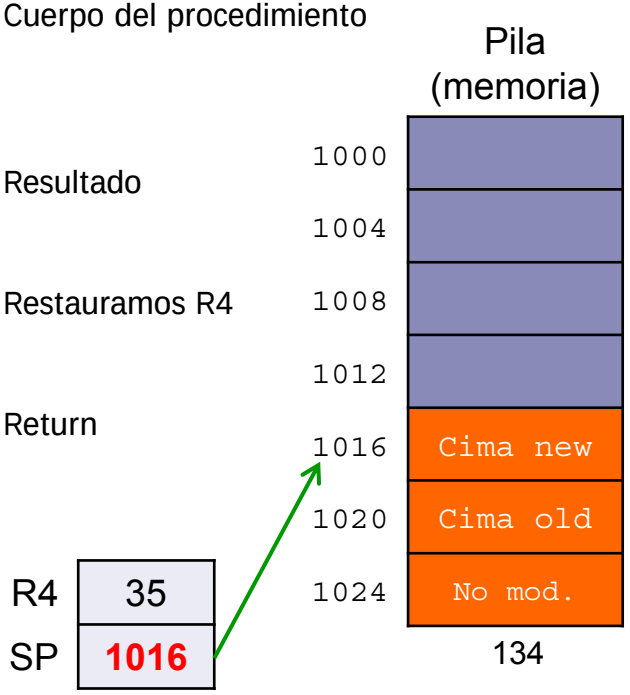
Guardamos R4 en la pila

Cuerpo del procedimiento

Resultado

Restauramos R4

Return



Ejemplo de función hoja

- ARM code:

```
leaf_example:
   SUB SP, SP, #4
   STR R4, [SP]
```

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

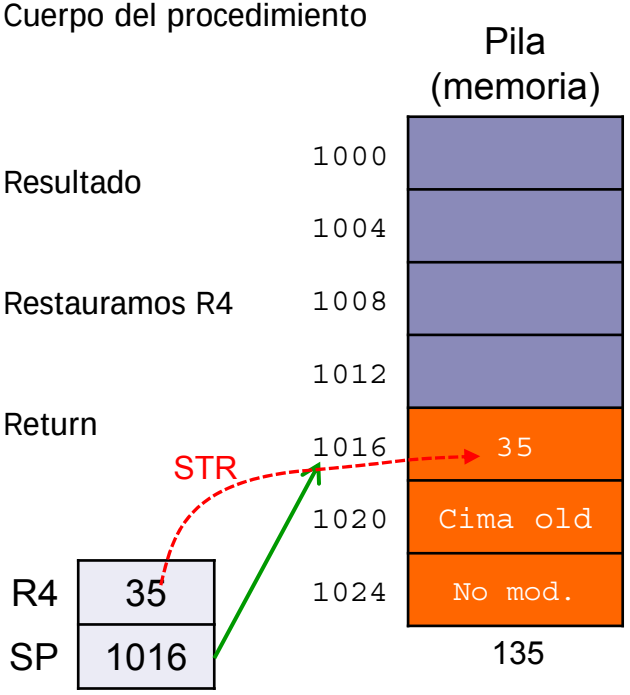
Guardamos R4 en la pila

Cuerpo del procedimiento

Resultado

Restauramos R4

Return



Ejemplo de función hoja

- ARM code:

```
leaf_example:
   SUB SP, SP, #4
   STR R4, [SP]
   ADD R0, R0, R1
   ADD R1, R2, R3
   SUB R4, R0, R1
```

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

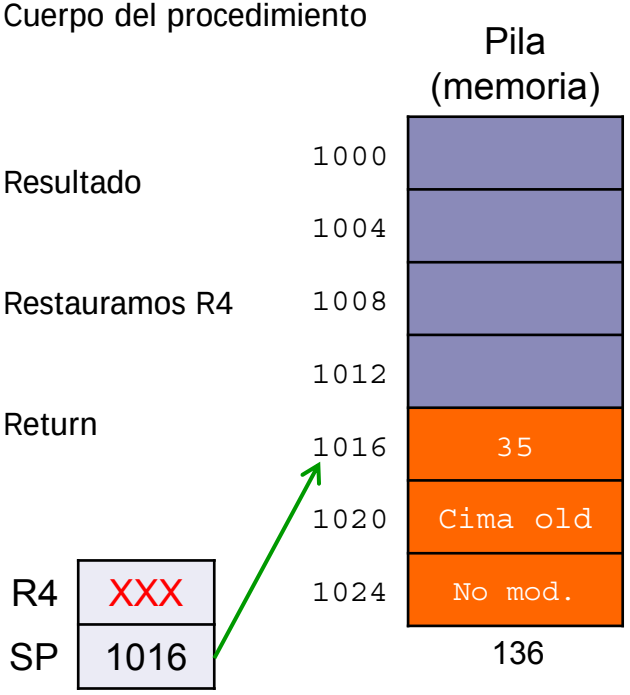
Guardamos R4 en la pila

Cuerpo del procedimiento

Resultado

Restauramos R4

Return



Ejemplo de función hoja

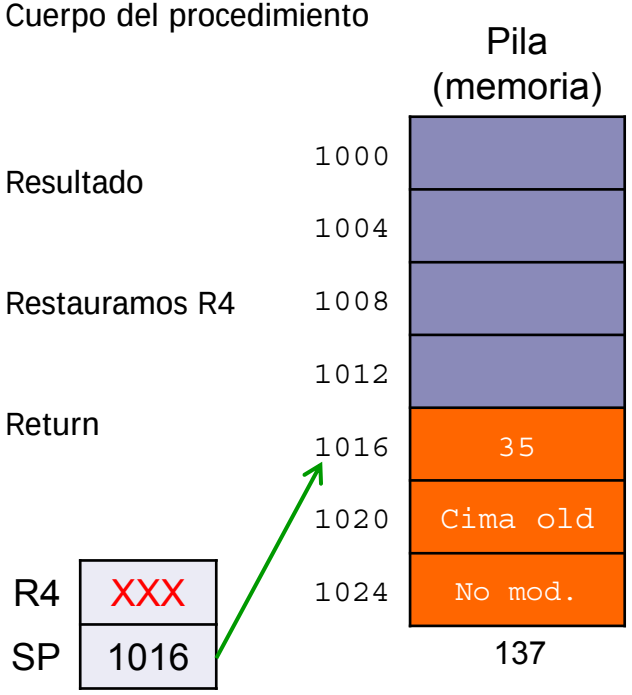
- ARM code:

```
leaf_example:
   SUB SP, SP, #4
   STR R4, [SP]
   ADD R0, R0, R1
   ADD R1, R2, R3
   SUB R4, R0, R1
   MOV R0, R4
```

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

Guardamos R4 en la pila

Cuerpo del procedimiento



Ejemplo de función hoja

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

- ARM code:

```
leaf_example:
    SUB SP, SP, #4
    STR R4, [SP]
    ADD R0, R0, R1
    ADD R1, R2, R3
    SUB R4, R0, R1
    MOV R0, R4
    LDR R4, [SP]
    ADD SP, SP, #4
```

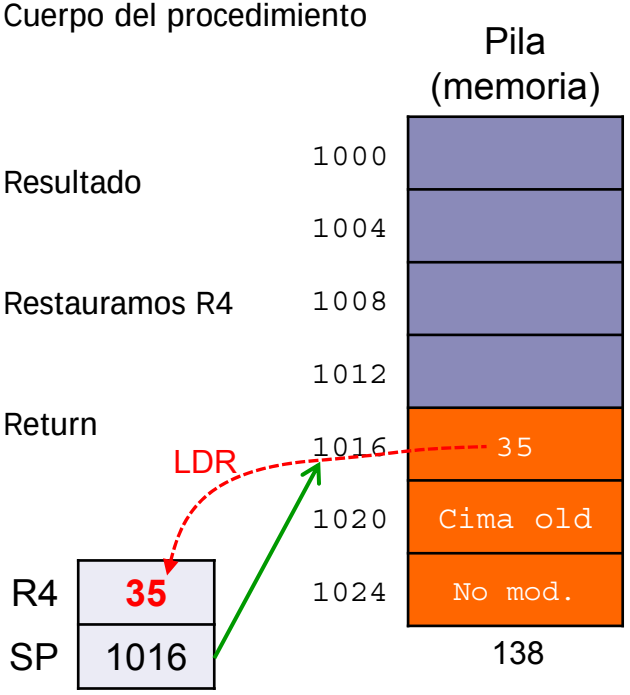
Guardamos R4 en la pila

Cuerpo del procedimiento

Resultado

Restauramos R4

Return



Ejemplo de función hoja

- ARM code:

```
leaf_example:
    SUB SP, SP, #4
    STR R4, [SP]
    ADD R0, R0, R1
    ADD R1, R2, R3
    SUB R4, R0, R1
    MOV R0, R4
    LDR R4, [SP]
    ADD SP, SP, #4
```

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

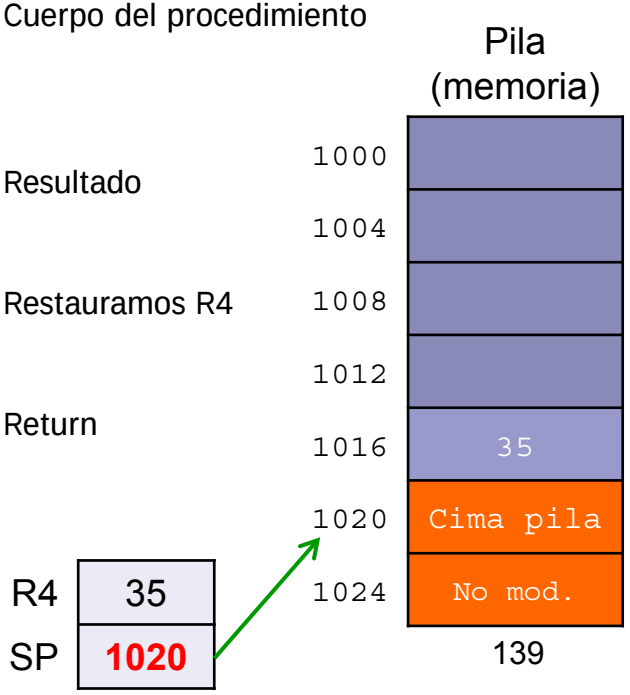
Guardamos R4 en la pila

Cuerpo del procedimiento

Resultado

Restauramos R4

Return



Ejemplo de función hoja

- ARM code:

```
leaf_example:
    SUB SP, SP, #4
    STR R4, [SP]
    ADD R0, R0, R1
    ADD R1, R2, R3
    SUB R4, R0, R1
    MOV R0, R4
    LDR R4, [SP]
    ADD SP, SP, #4
    MOV PC, LR
```

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

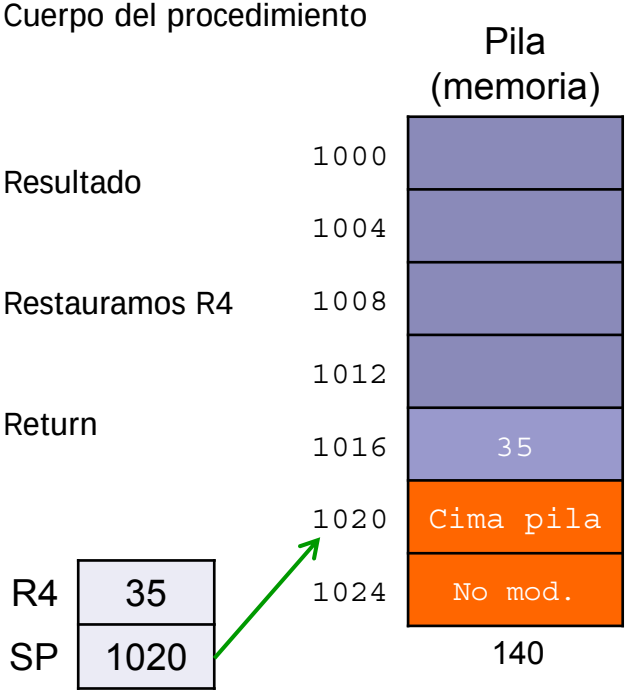
Guardamos R4 en la pila

Cuerpo del procedimiento

Resultado

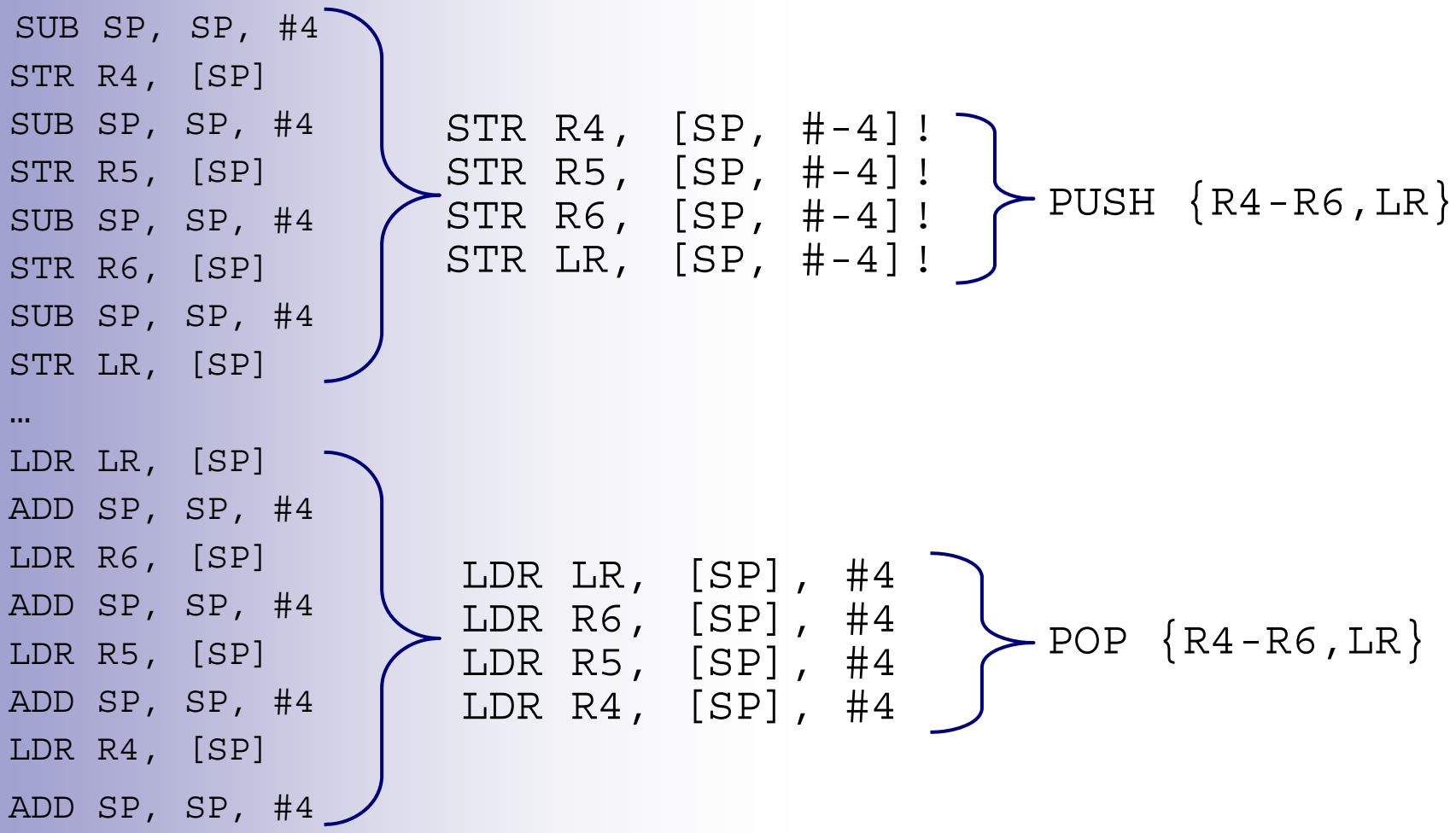
Restauramos R4

Return



Simplificación PUSH-POP

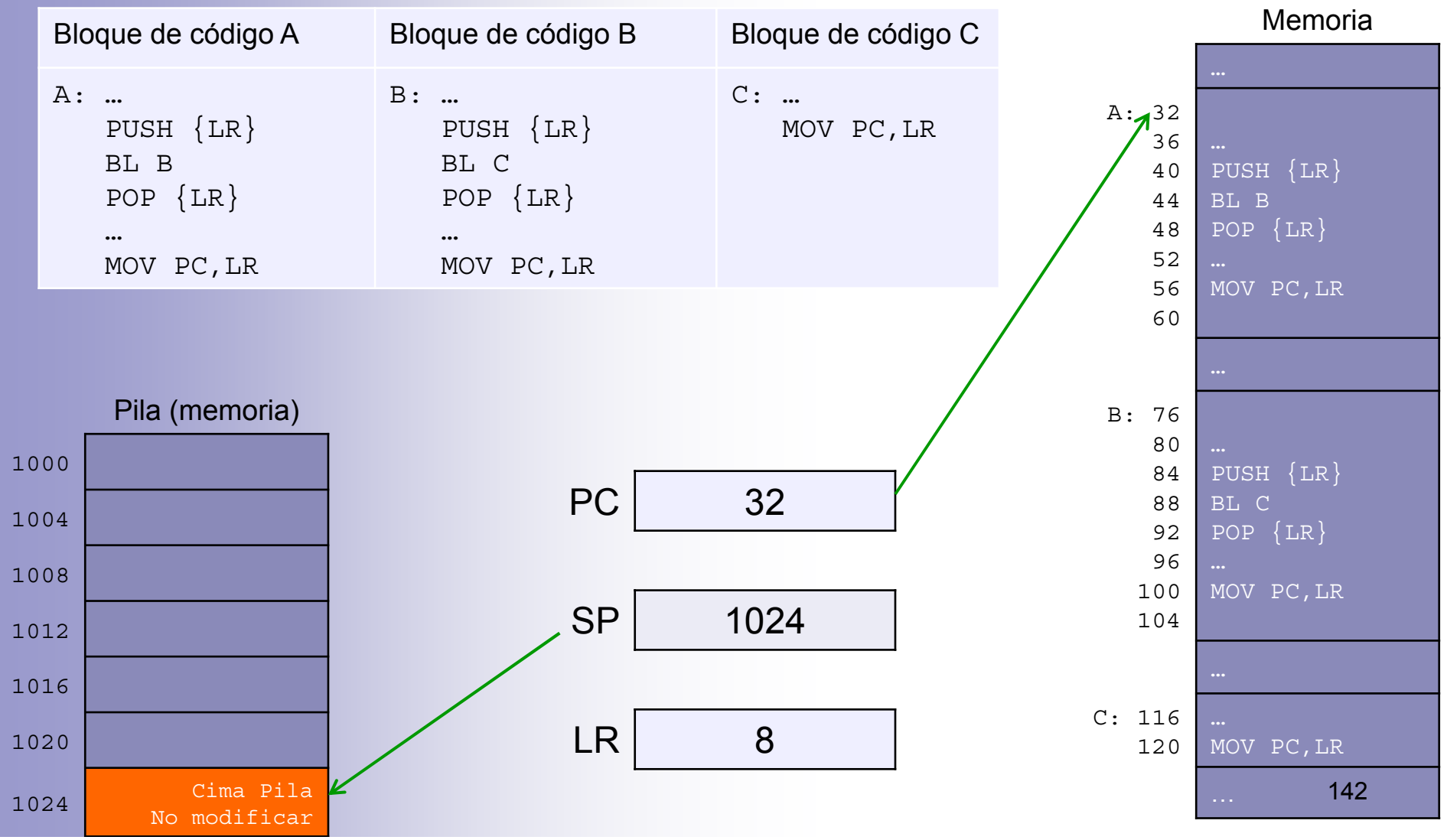
- ARM code:



Anidamiento de llamadas a función

- Llamadas 3 funciones anidadas: A→B→C

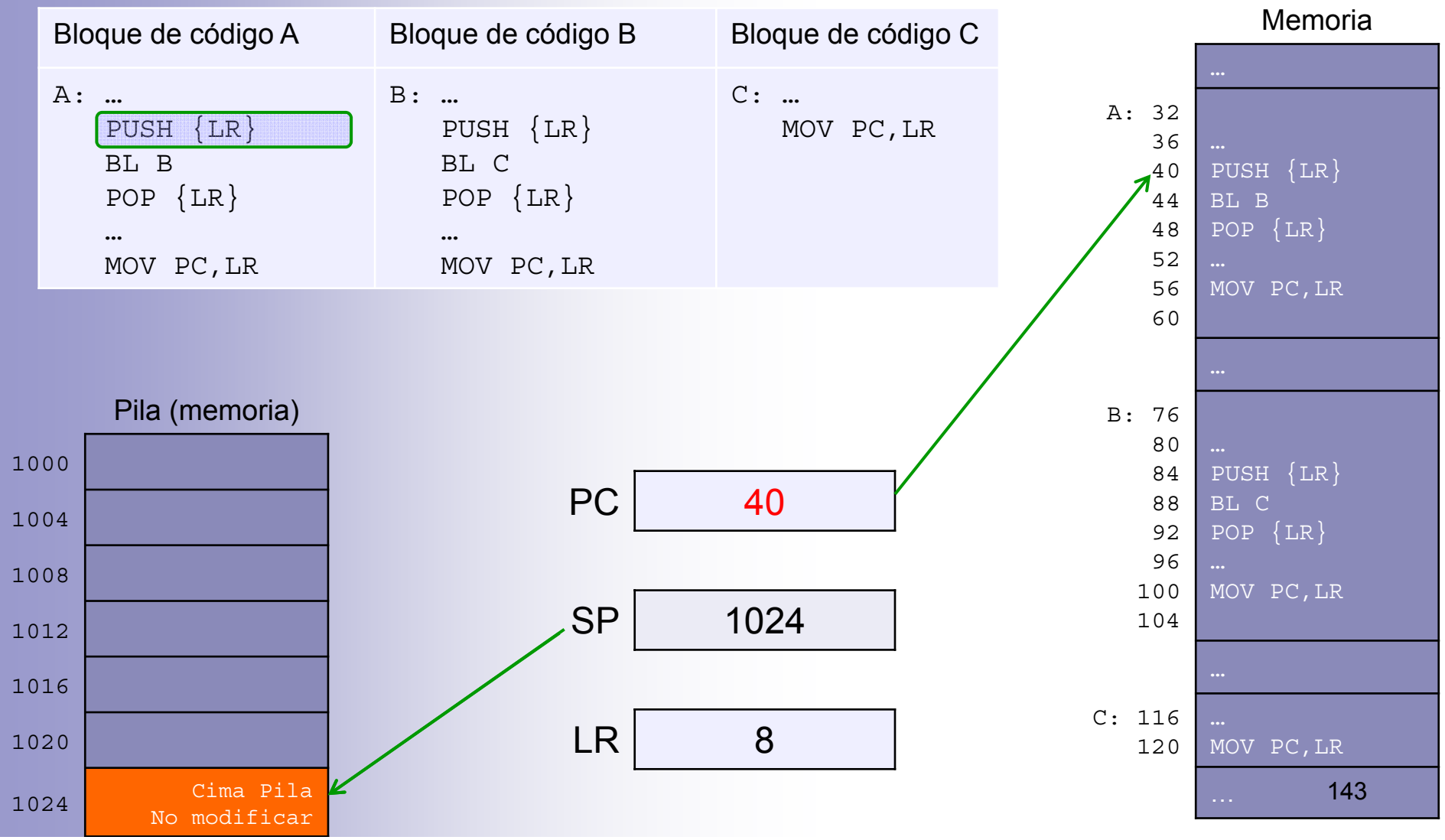
Bloque de código A	Bloque de código B	Bloque de código C
A: … PUSH {LR} BL B POP {LR} … MOV PC,LR	B: … PUSH {LR} BL C POP {LR} … MOV PC,LR	C: … MOV PC,LR



Anidamiento de llamadas a función

- Llamadas 3 funciones anidadas: A→B→C

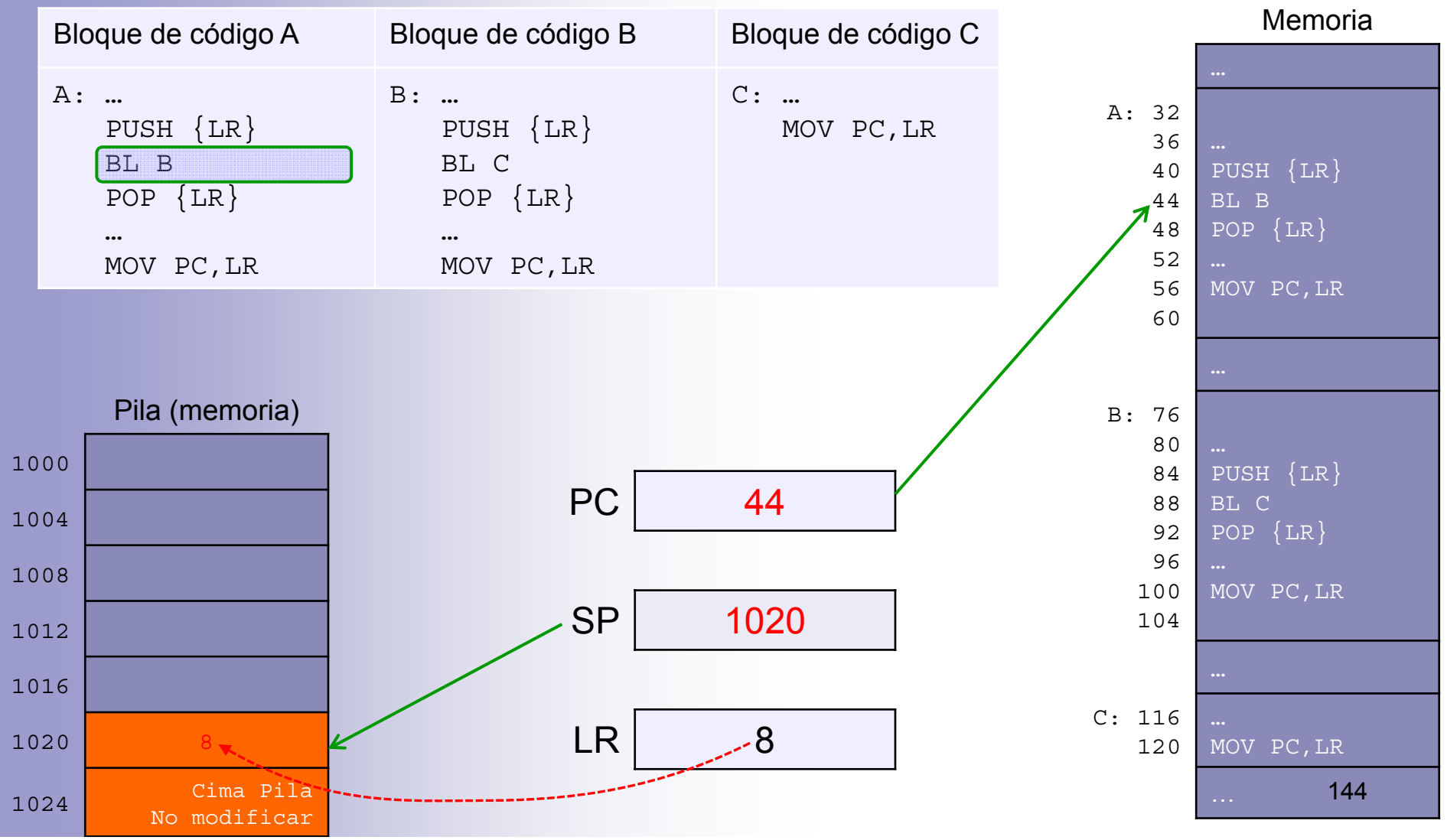
Bloque de código A	Bloque de código B	Bloque de código C
A: … PUSH {LR} BL B POP {LR} … MOV PC,LR	B: … PUSH {LR} BL C POP {LR} … MOV PC,LR	C: … MOV PC,LR



Anidamiento de llamadas a función

- Llamadas 3 funciones anidadas: A→B→C

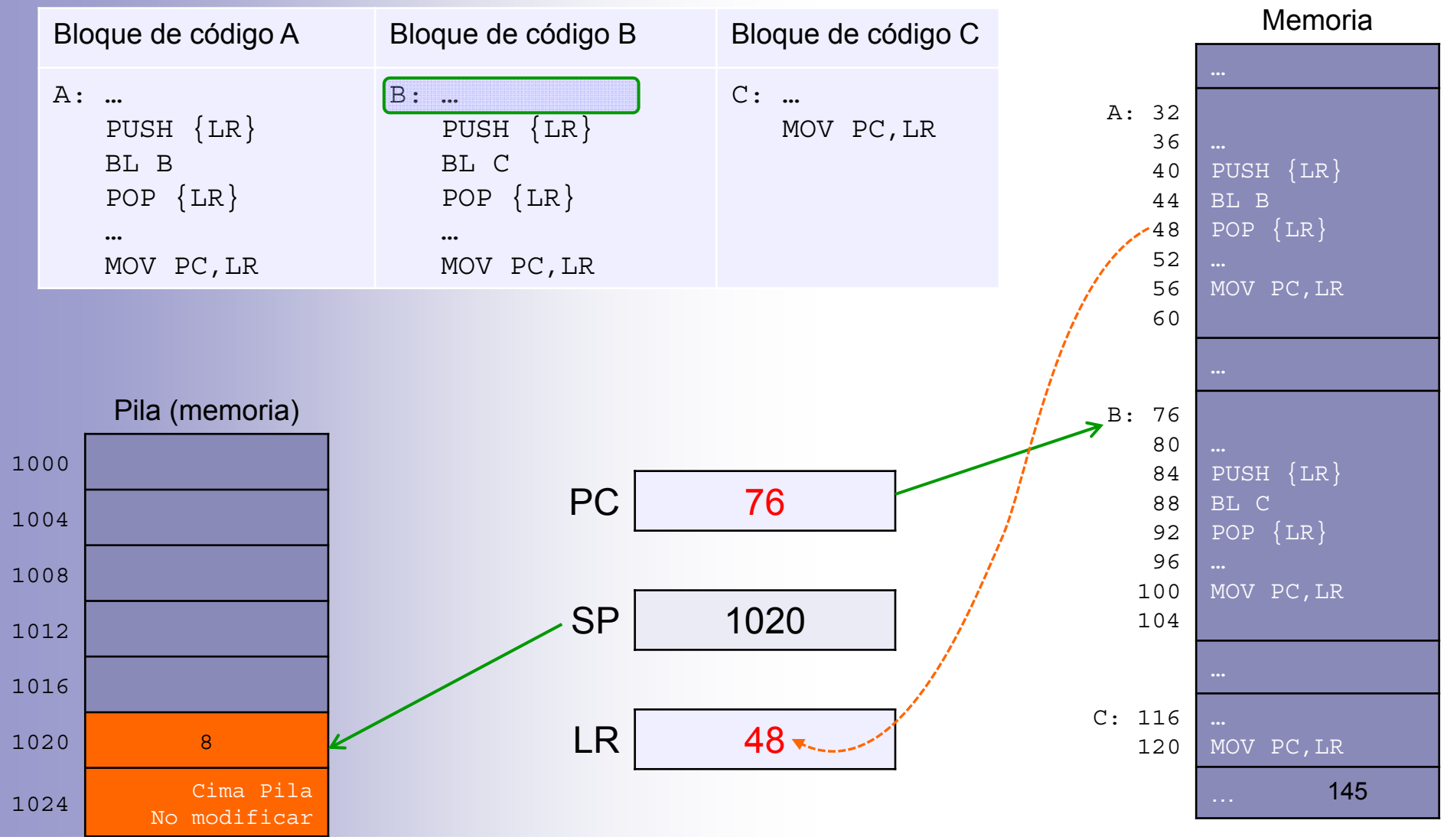
Bloque de código A	Bloque de código B	Bloque de código C
A: … PUSH {LR} BL B POP {LR} … MOV PC,LR	B: … PUSH {LR} BL C POP {LR} … MOV PC,LR	C: … MOV PC,LR



Anidamiento de llamadas a función

- Llamadas 3 funciones anidadas: A→B→C

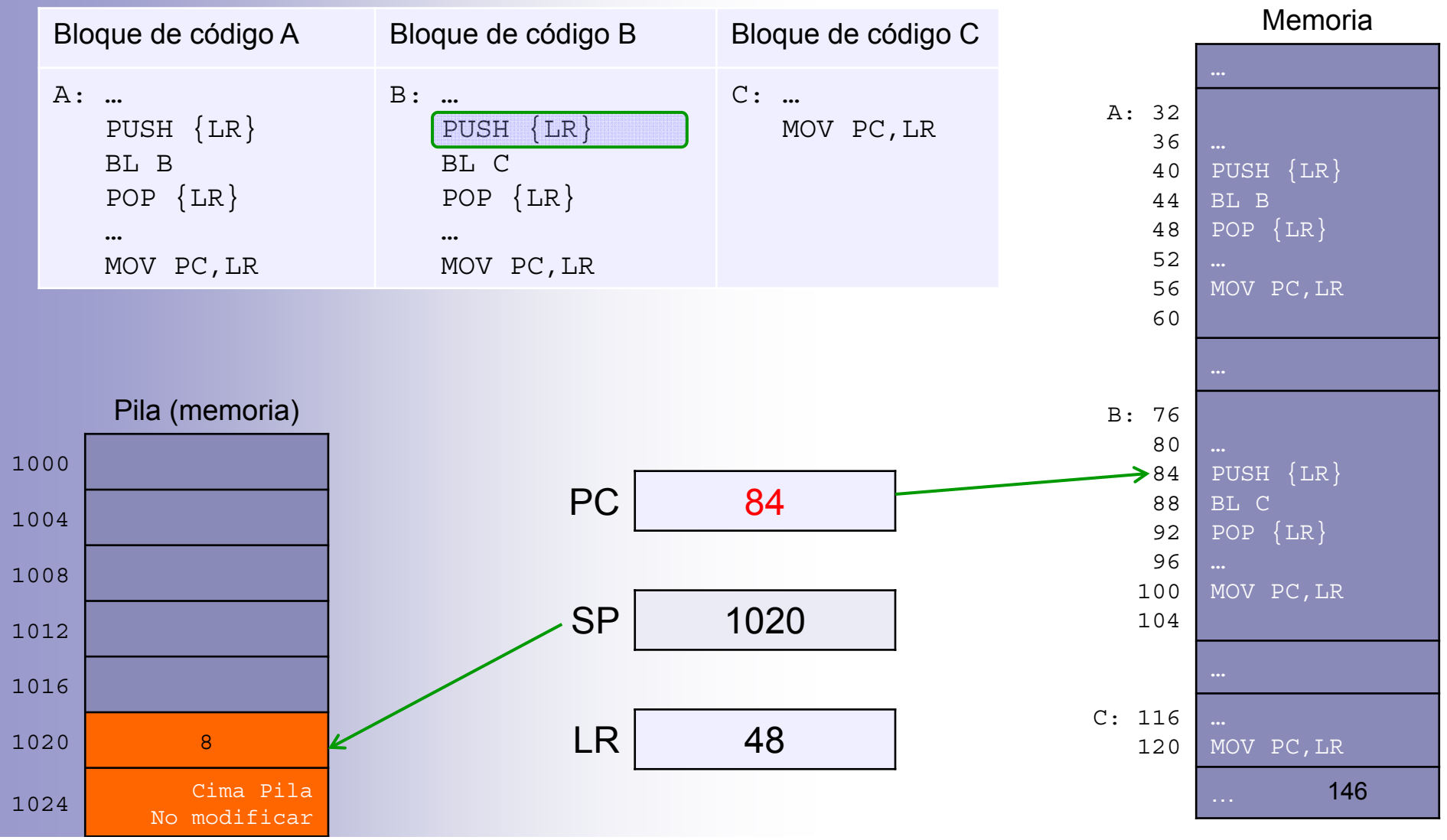
Bloque de código A	Bloque de código B	Bloque de código C
A: … PUSH {LR} BL B POP {LR} … MOV PC,LR	B: … PUSH {LR} BL C POP {LR} … MOV PC,LR	C: … MOV PC,LR



Anidamiento de llamadas a función

- Llamadas 3 funciones anidadas: A→B→C

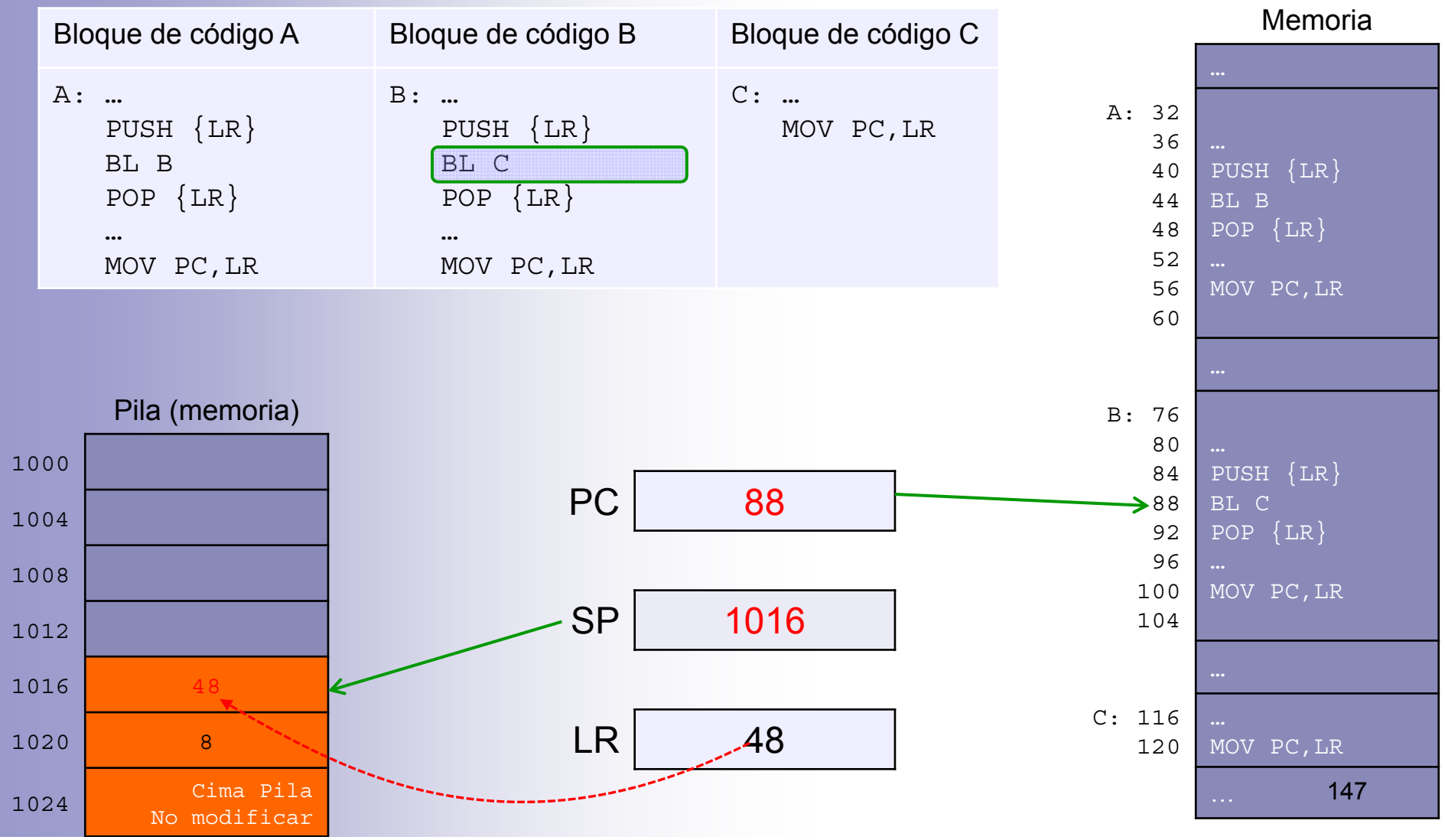
Bloque de código A	Bloque de código B	Bloque de código C
A: … PUSH {LR} BL B POP {LR} … MOV PC,LR	B: … PUSH {LR} BL C POP {LR} … MOV PC,LR	C: … MOV PC,LR



Anidamiento de llamadas a función

- Llamadas 3 funciones anidadas: A→B→C

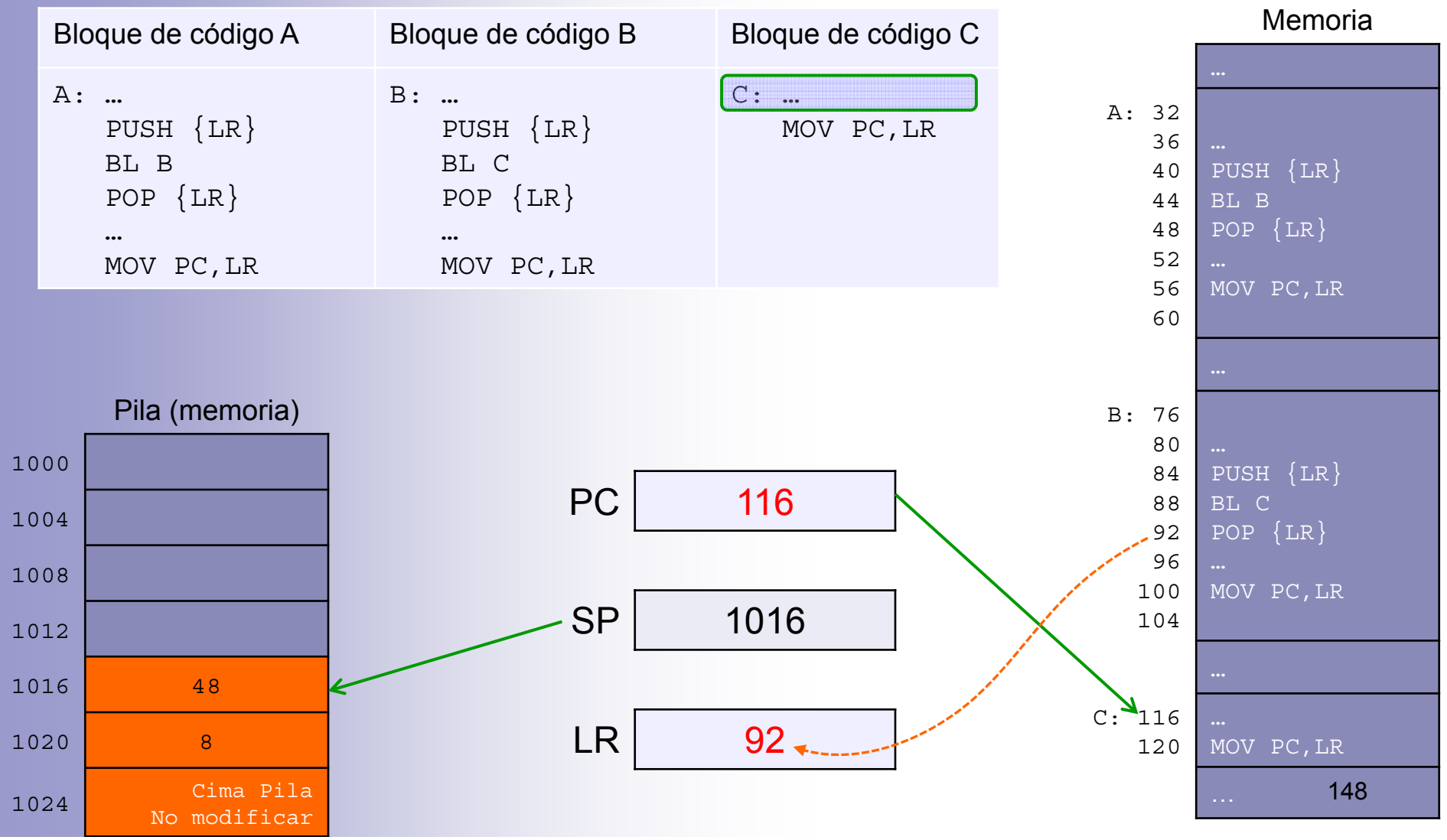
Bloque de código A	Bloque de código B	Bloque de código C
A: … PUSH {LR} BL B POP {LR} … MOV PC,LR	B: … PUSH {LR} BL C POP {LR} … MOV PC,LR	C: … MOV PC,LR



Anidamiento de llamadas a función

- Llamadas 3 funciones anidadas: A→B→C

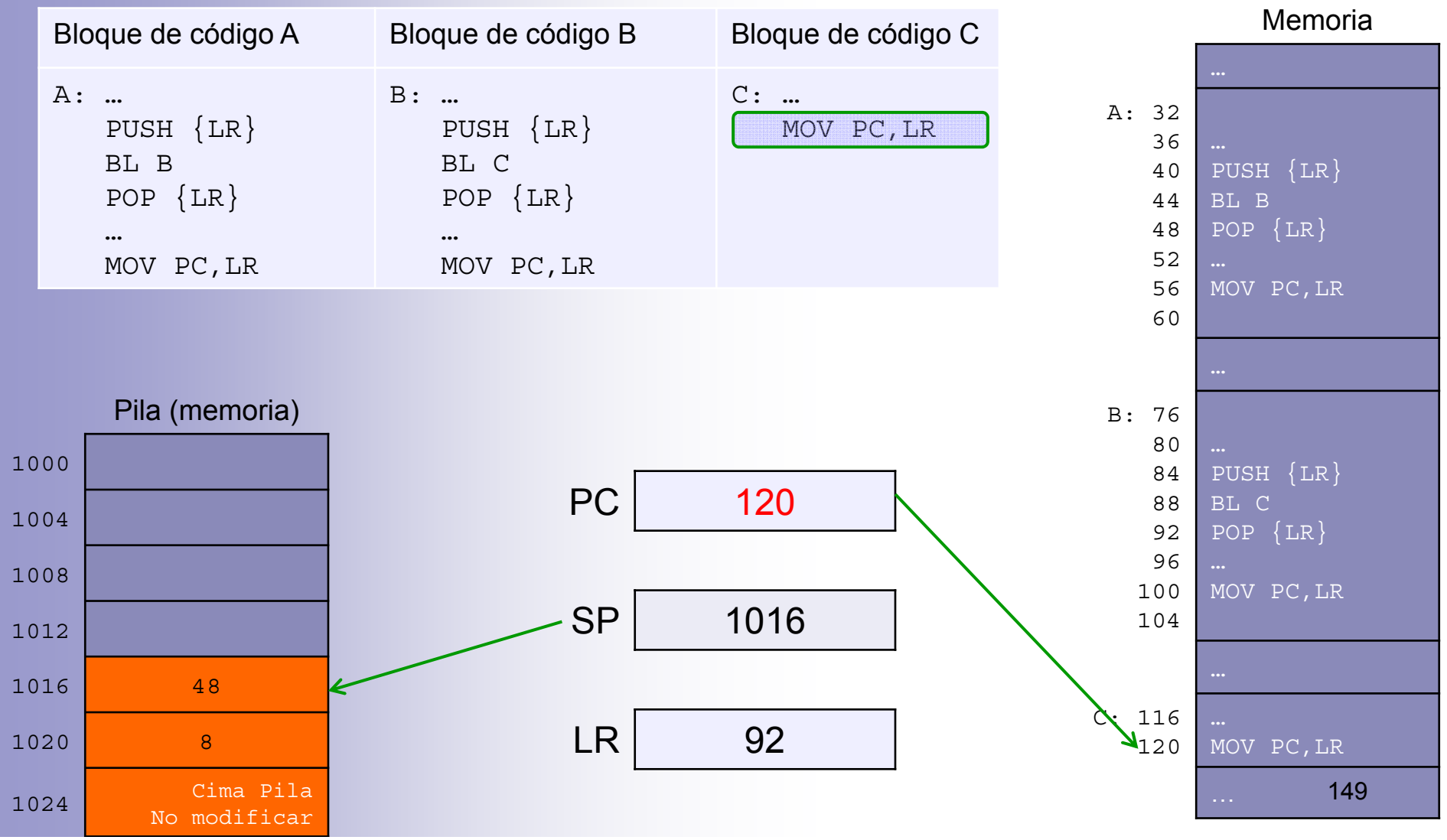
Bloque de código A	Bloque de código B	Bloque de código C
A: … PUSH {LR} BL B POP {LR} … MOV PC,LR	B: … PUSH {LR} BL C POP {LR} … MOV PC,LR	C: … MOV PC,LR



Anidamiento de llamadas a función

- Llamadas 3 funciones anidadas: A→B→C

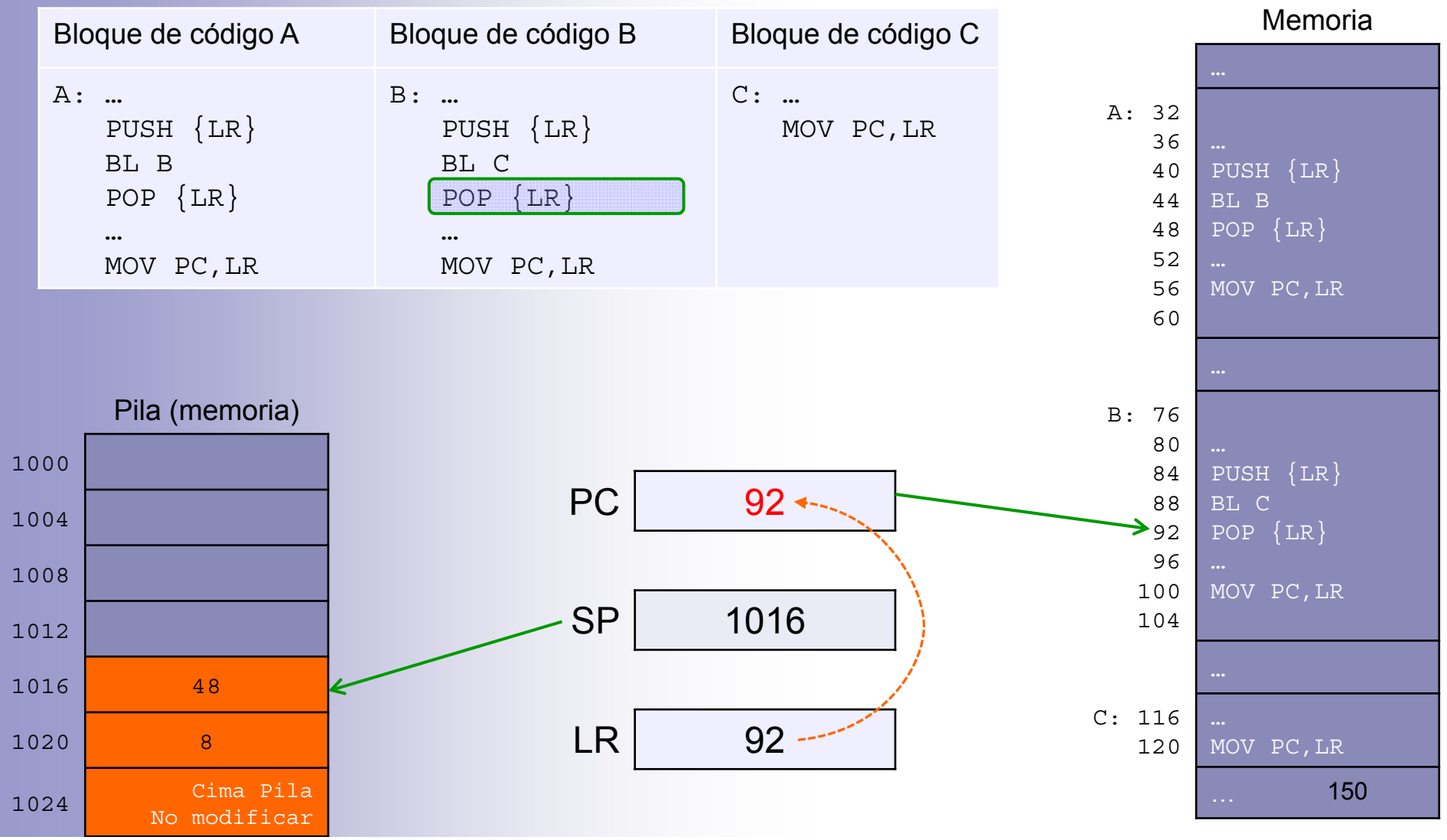
Bloque de código A	Bloque de código B	Bloque de código C
A: … PUSH {LR} BL B POP {LR} … MOV PC,LR	B: … PUSH {LR} BL C POP {LR} … MOV PC,LR	C: … MOV PC,LR



Anidamiento de llamadas a función

- Llamadas 3 funciones anidadas: A→B→C

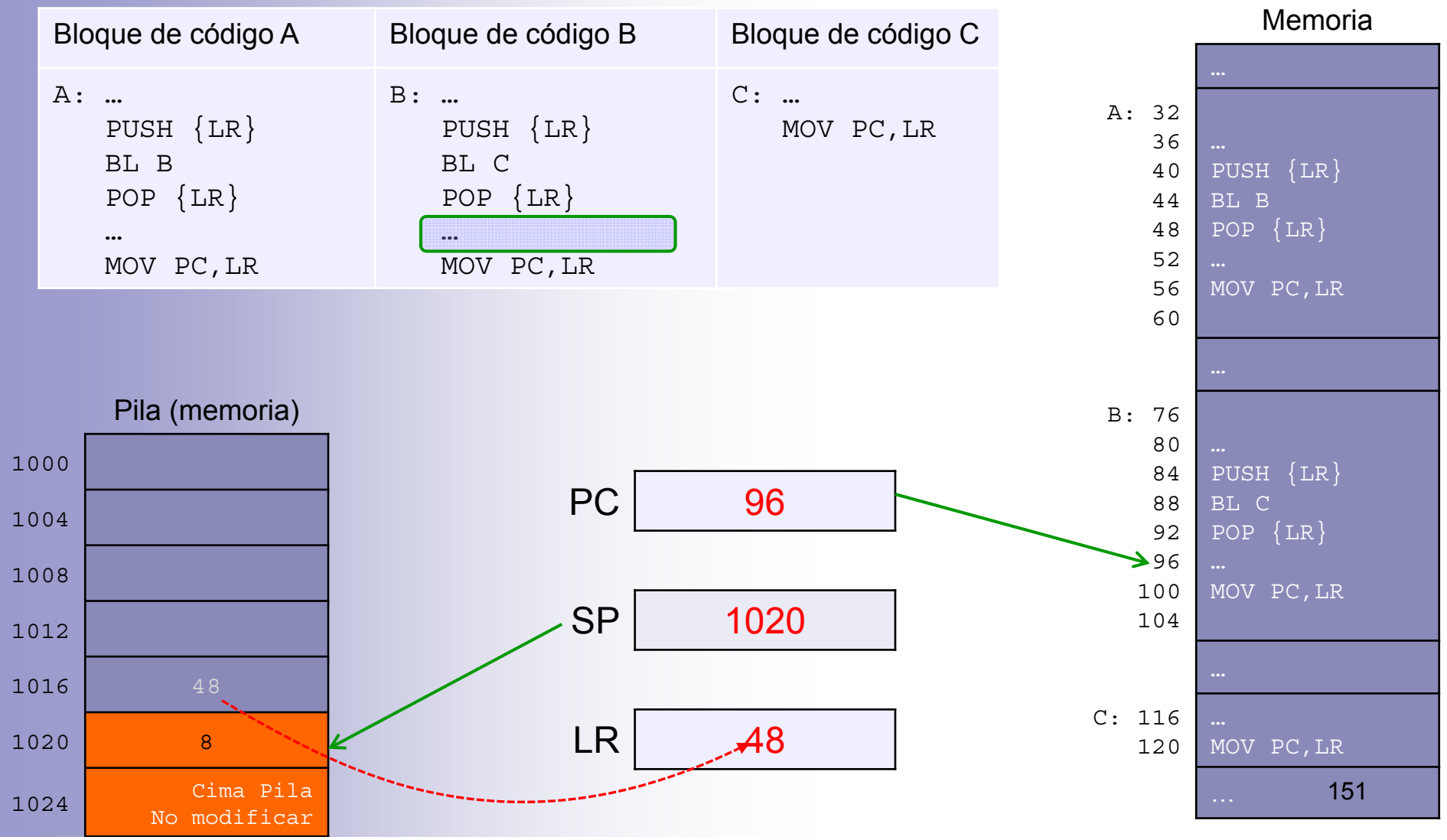
Bloque de código A	Bloque de código B	Bloque de código C
A: … PUSH {LR} BL B POP {LR} … MOV PC,LR	B: … PUSH {LR} BL C POP {LR} … MOV PC,LR	C: … MOV PC,LR



Anidamiento de llamadas a función

- Llamadas 3 funciones anidadas: A→B→C

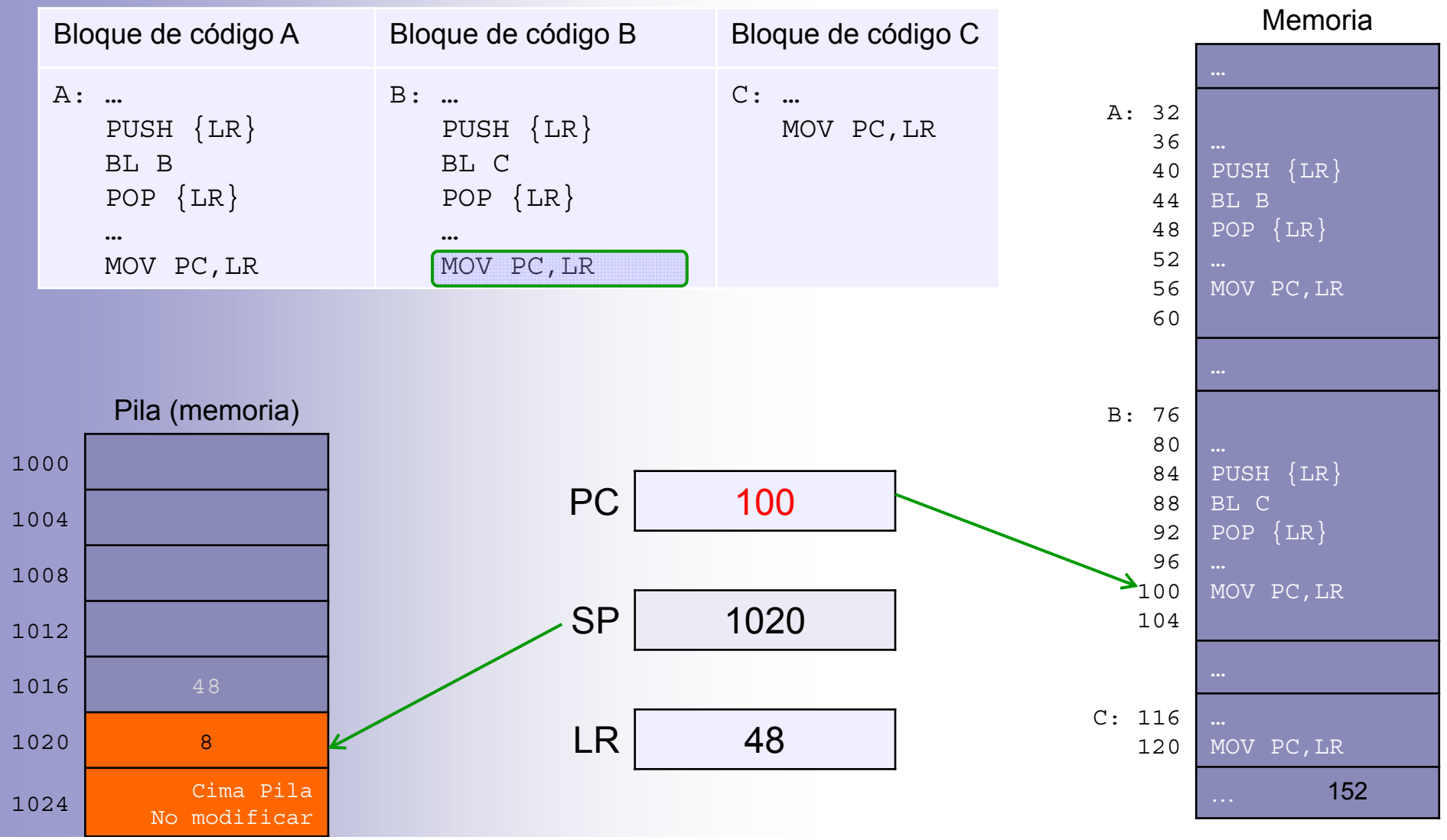
Bloque de código A	Bloque de código B	Bloque de código C
A: … PUSH {LR} BL B POP {LR} … MOV PC,LR	B: … PUSH {LR} BL C POP {LR} … MOV PC,LR	C: … MOV PC,LR



Anidamiento de llamadas a función

- Llamadas 3 funciones anidadas: A→B→C

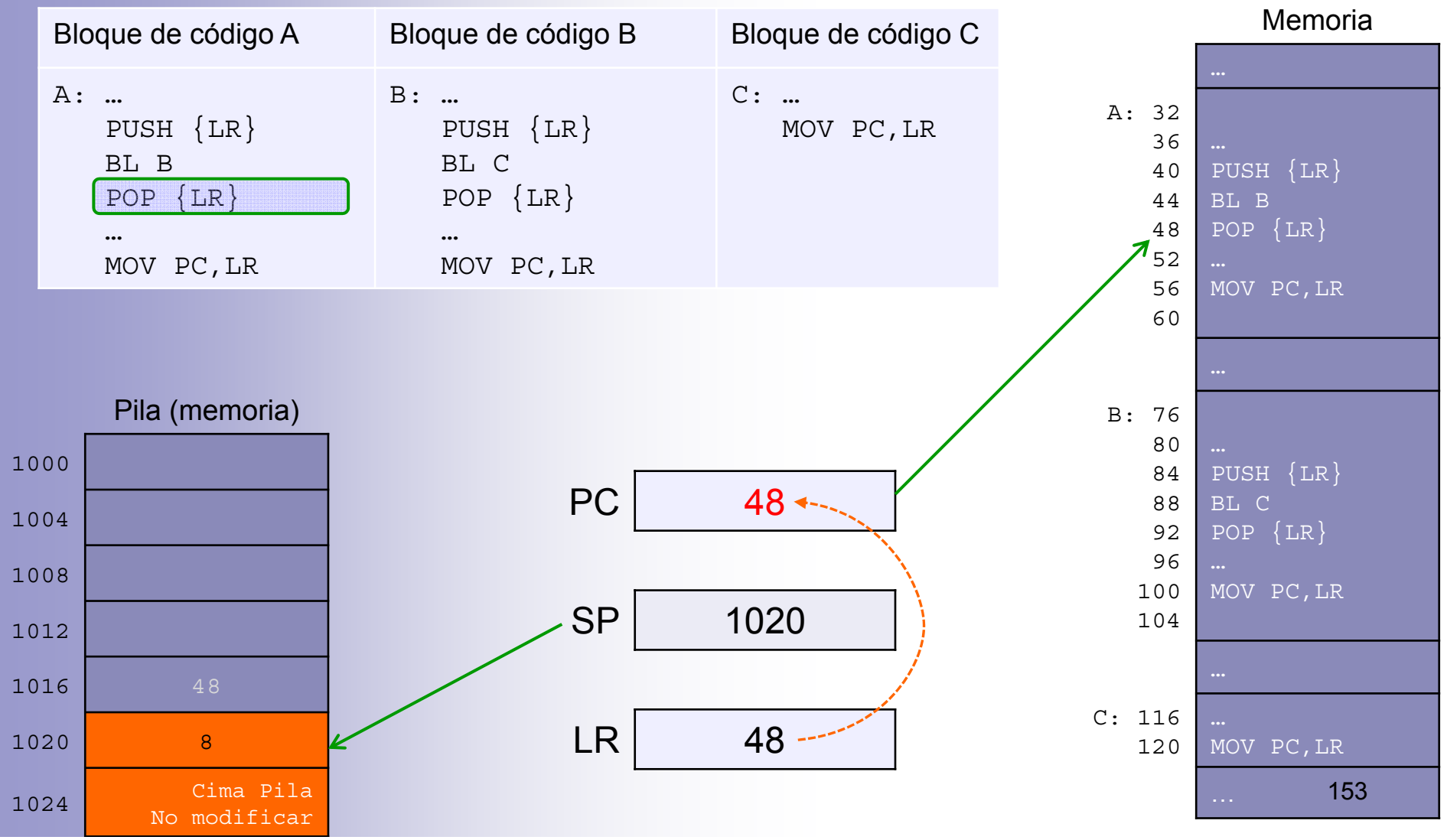
Bloque de código A	Bloque de código B	Bloque de código C
A: … PUSH {LR} BL B POP {LR} … MOV PC,LR	B: … PUSH {LR} BL C POP {LR} … MOV PC,LR	C: … MOV PC,LR



Anidamiento de llamadas a función

- Llamadas 3 funciones anidadas: A→B→C

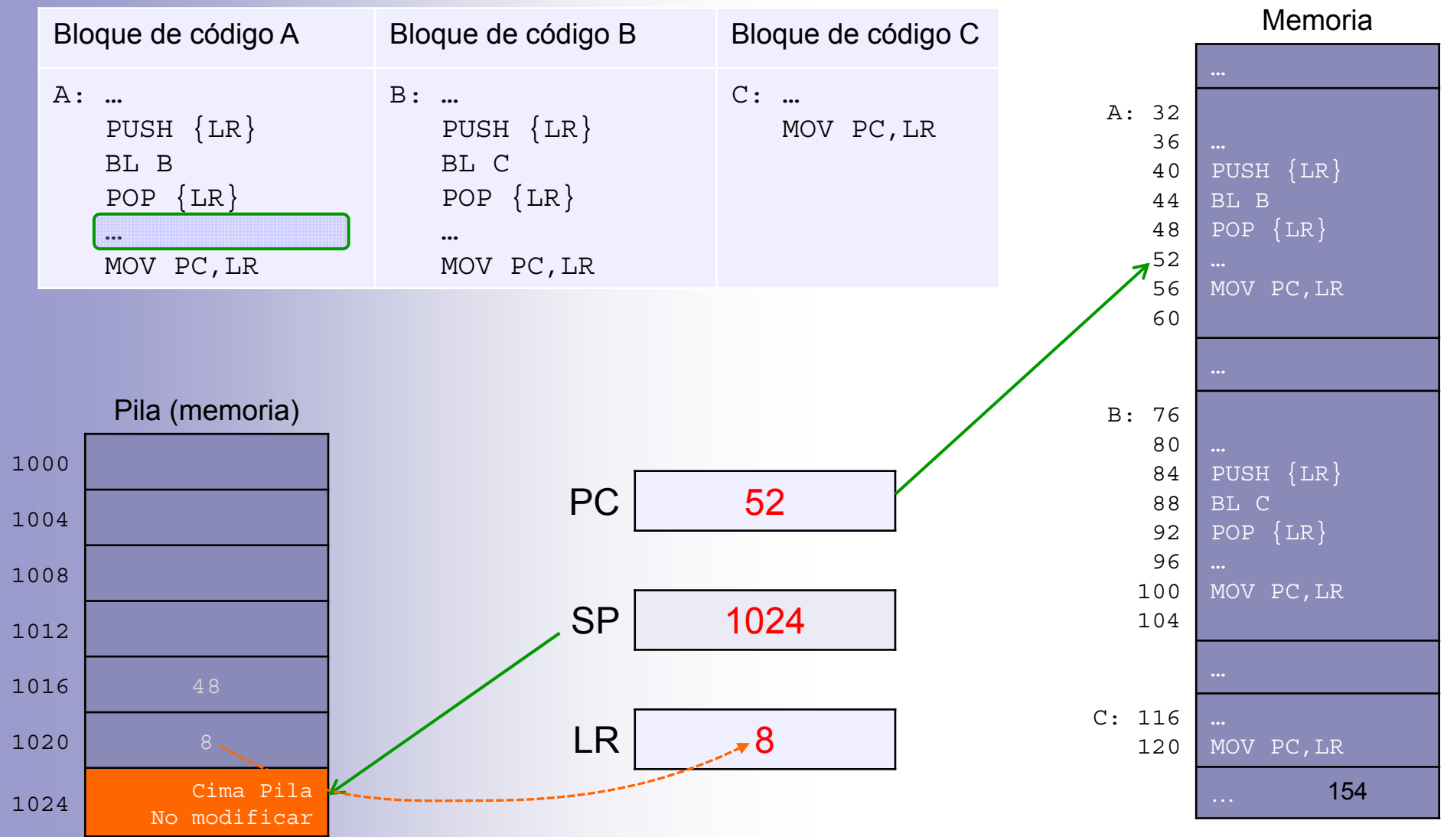
Bloque de código A	Bloque de código B	Bloque de código C
A: … PUSH {LR} BL B POP {LR} … MOV PC,LR	B: … PUSH {LR} BL C POP {LR} … MOV PC,LR	C: … MOV PC,LR



Anidamiento de llamadas a función

- Llamadas 3 funciones anidadas: A→B→C

Bloque de código A	Bloque de código B	Bloque de código C
A: … PUSH {LR} BL B POP {LR} … MOV PC,LR	B: … PUSH {LR} BL C POP {LR} … MOV PC,LR	C: … MOV PC,LR



Resumen instrucciones ARM

	Instrucción	Ejemplo
Aritméticas	`ADD{C}{S} r1,r2,Op2`	r1 = r2 + Op2
	`ADC{C}{S} r1,r2,Op2`	r1 = r2 + Op2 + C
	`SUB{C}{S} r1,r2,Op2`	r1 = r2 – Op2
	`RSB{C}{S} r1,r2,Op2`	r1 = Op2 – r2
	`MUL{C}{S} r1,r2,r3`	r1 = r2 x r3
	`MLA{C}{S} r1,r2,r3,r4`	r1 = r2 x r3 + r4
Lógi.	`AND{C}{S} r1,r2,Op2`	r1 = r2 AND Op2
	`ORR{C}{S} r1,r2,Op2`	r1 = r2 OR Op2
Comp.	`CMP{C} r1,Op2`	Flag.cond.= r1 – Op2
	`TST{C} r1,Op2`	Flag.cond.= r1 AND Op2
Movimiento de datos	`MOV{C}{S} r1,Op2`	r1 = Op2
	`MVN{C}{S} r1,Op2`	r1 = !Op2
	`LDR{C}{B} r1,[r2]`	r1 = MEM(r2)
	`STR{C}{B} r1,[r2]`	MEM(r2) = r1
Salto	`B{C} label`	PC = label
	`BL{C} label`	LR = PC + 4; PC = label

Sufijo {c}	Condición
EQ	Equal
NE	Not equal
CS / HS	Carry set / Unsigned higher or same
CC / LO	Carry clear / Unsigned lower
MI	Minus / Negative
PL	Plus / Positive of zero
VS	Overflow / Overflow set
VC	No overflow / Overflow clear
HI	Unsigned higher
LS	Unsigned lower or same
GE	Signed greater than or equal
LT	Signed less than
GT	Signed greater than
LE	Signed less than or equal

Sufijo	
{S}	Guardar flags en reg. estado
{B}	Transferencia de Byte