Estructura de Computadores

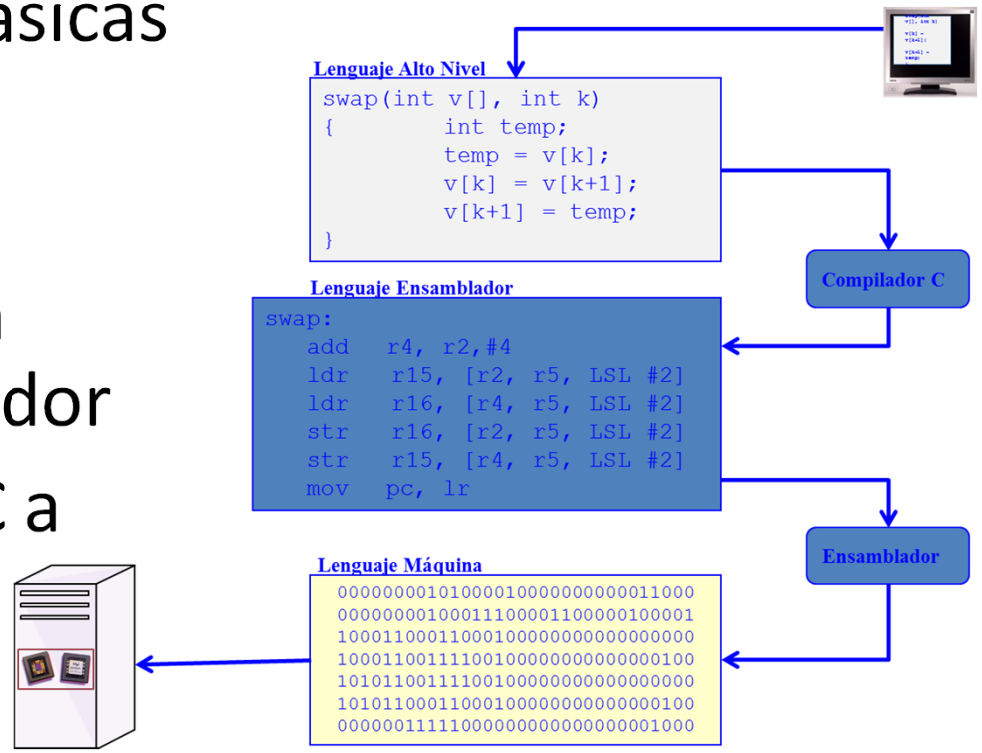
2º Cuatrimestre
2013-2014
Grupo C

Marcos Sánchez-Élez

ec

Módulo 1. Repaso de ARM

- Repertorio de instrucciones del ARM
- Acceso a estructuras básicas de datos
- Subrutinas
- Estructura básica de un programa en ensamblador
- Compilación, paso de C a ensamblador



Módulo 1. Repaso de ARM

Repertorio de instrucciones

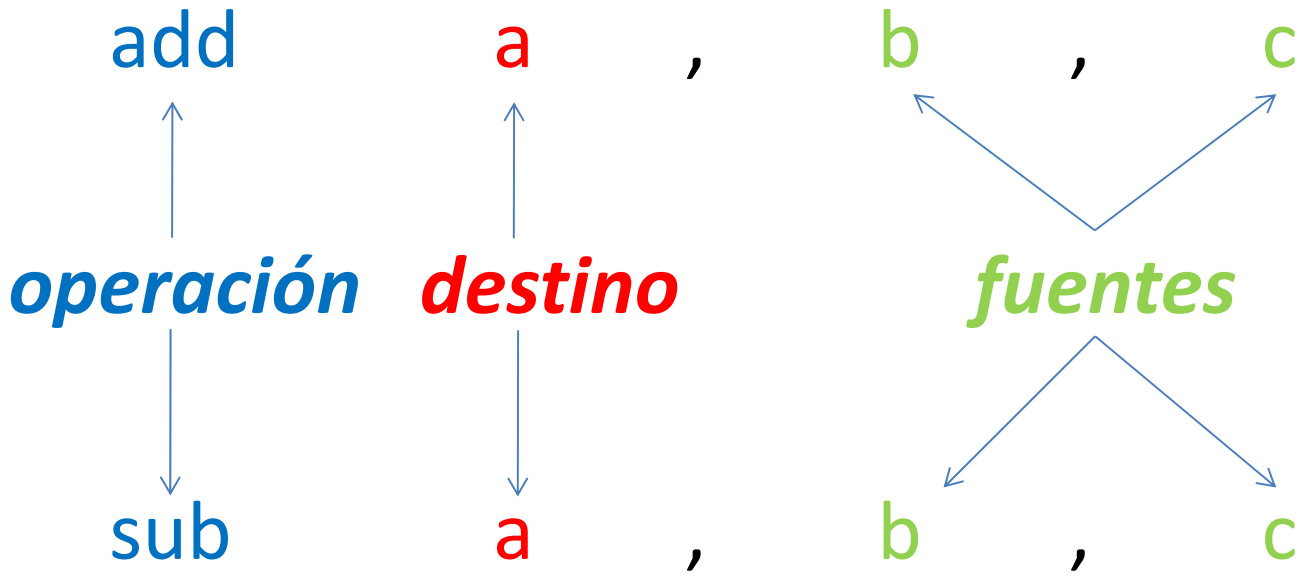
ec

Repertorio de instrucciones

- Un computador debe ser capaz de:
  - Realizar las operaciones matemáticas elementales: **Instrucciones aritmetico-lógicas**
  - Obtener y almacenar los datos que utiliza la instrucción: **Instrucciones de acceso a memoria**
  - Modificar el flujo secuencial del programa: **Instrucciones de salto**
  - Otras dependiendo de las características particulares de la arquitectura

Repertorio de instrucciones

- En ARM, las operaciones aritméticas y lógicas contienen en general 2 operandos fuente y 1 operando destino. Por ejemplo:



Repertorio de instrucciones

SUMA	**add** Rd, Rn, <Operando>
RESTA	**sub** Rd, Rn, <Operando>
MULTIPLICACIÓN	**mul** Rd, Rm, Rs
AND	**and** Rd, Rn, <Operando>
OR	**orr** Rd, Rn, <Operando>
XOR	**eor** Rd, Rn, <Operando>
MOVIMIENTO	**mov** Rd, <Operando>
DESPLAZAMIENTO	**lsl** Rd, Rm, <Operando> **lsr** Rd, Rm, <Operando>

Repertorio de instrucciones

- Instrucción de LOAD
  - Mueve un dato de una posición de la Memoria a un registro del Banco de Registros:

  Memoria → Banco Regs **ldr** Rd, <Dirección>

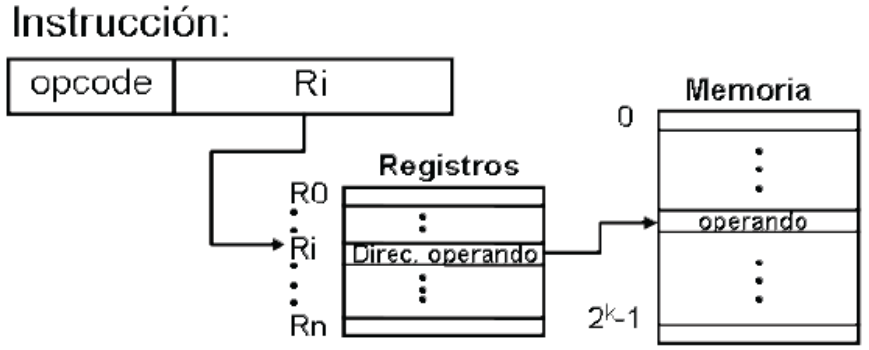
- Instrucción de STORE
  - Mueve un dato de un registro del Banco de Registros a una posición de la Memoria:

  Banco Regs → Memoria **str** Rd, <Dirección>

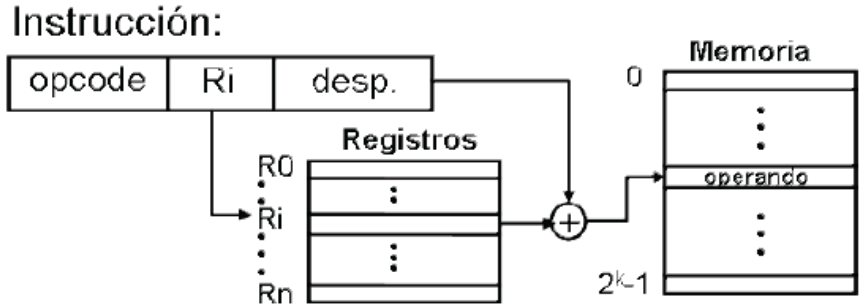
Repertorio de instrucciones

- Diversas formas para especificar la dirección donde se encuentra almacenado el dato:

ldr Rd, [Ri]

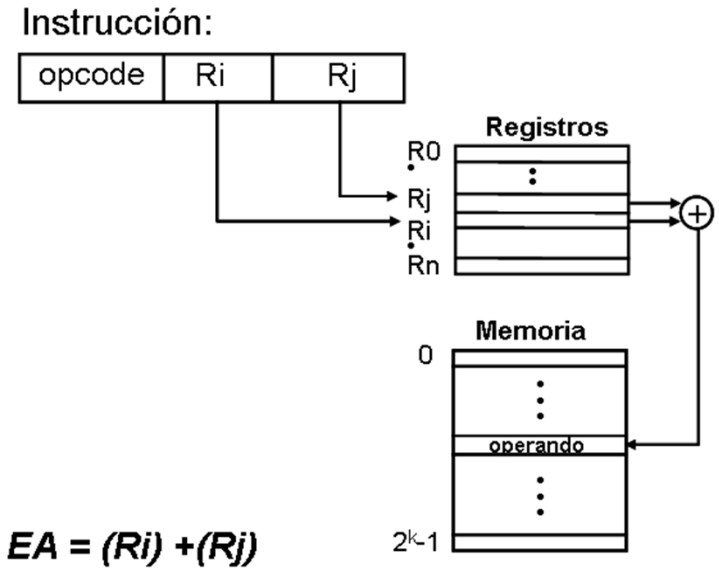


ldr Rd, [Rn, #±Desplazamiento]



Repertorio de instrucciones

ldr Rd, [Rn, Rs]



– El segundo registro puede utilizarse desplazado

ldr Rd, [Rn, Rs, LSL #N] multiplicar por potencia de 2

ldr Rd, [Rn, Rs, LSR #N] dividir por potencia de 2

Repertorio de instrucciones

- También se puede jugar con la escritura sobre el registro donde se calcula la dirección:
  - Registro base con desplazamiento / pre-indexado

    ldr Rd, [Rn, #N] str Rd, [Rn, #N]

  - Registro base con desplazamiento / pre-indexado con post-escritura

    ldr Rd, [Rn, #N]! str Rd, [Rn, #N]!

  - Registro base con desplazamiento / post-indexado con post escritura

    ldr Rd, [Rn], #N str Rd, [Rn], #N

Repertorio de instrucciones

Cómo acceder a un array

```
int a[10];


a[0] = 1;
a[1] = 3;
a[2] = 7;
a[3] = 5;
a[4] = -2;
…
a[9] = 4;
```

"Compilación"

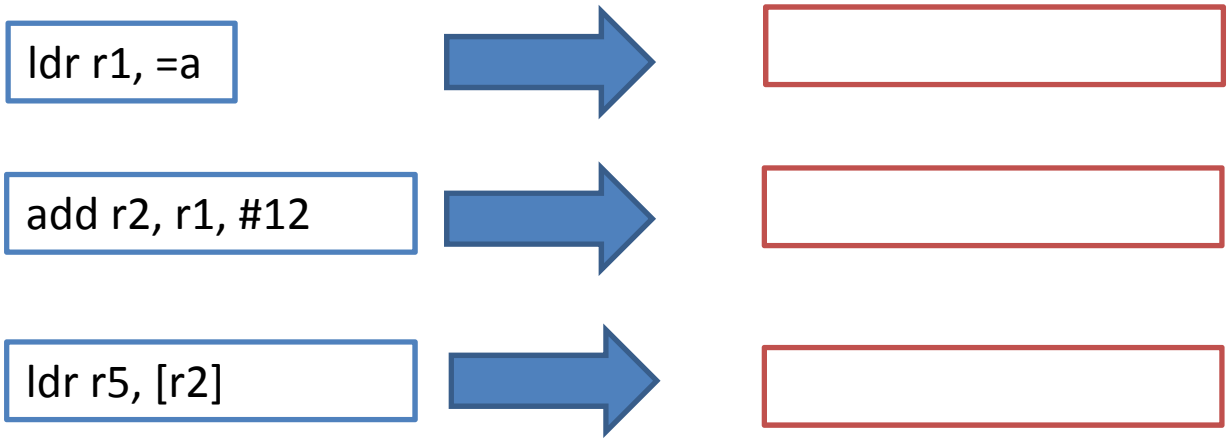
```
a:    .word    1, 3, 7, 5, -2, …, 4
```

Como se almacena en memoria

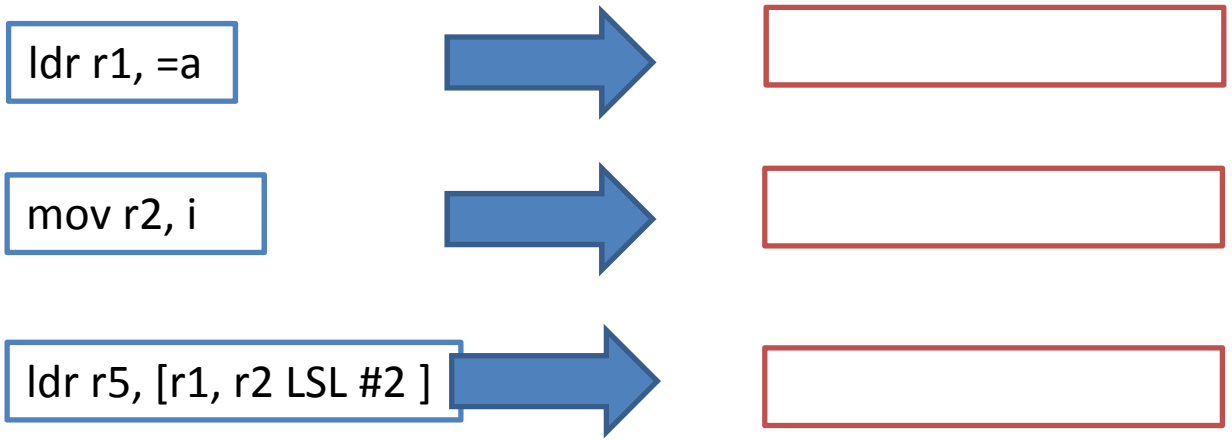
1020	0x00000001
1024	0x00000003
1028	0x00000007
1032	0x00000005
1036	0xFFFFFFFD
…	…
1056	0x00000004

Repertorio de instrucciones

¿Dónde está la componente 3? _____



¿Dónde está la componente i? _____



Repertorio de instrucciones

- Modificación del flujo del programa: Salto

  bXX #Desplazamiento

  – El Desplazamiento se calcula respecto al pc, a la hora de programar se utiliza una etiqueta y es el enlazador quien calcula el desplazamiento

  – Condiciones XX: EQ, NE, GE, LT, GT, LE …

Repertorio de instrucciones

- Traducir la siguiente sentencia de C a ensamblador (suponer que A está en r1 y B en r2):

If (A<B) then A=A+B else A=A-B

```
        cmp r1, r2
        bge else
        add r1, r1, r2
        b Salir
else:   sub r1, r1, r2
Salir:
```

Repertorio de instrucciones

- Traducir la siguiente sentencia de C a ensamblador :

```
i = 0;
while (i<10) {
        array[i]=array[i]+i;
        i++;
}
```

```
array:      .word       Componentes del vector

            mov r2, #0
            ldr r3, =array
LOOP:       cmp r2, #10
            bge Salir
            ldr r4, [r3]
            add r4, r4, r2
            str r4, [r3], #4    @ tras almacenar, r3 = r3 + 4
            add r2, r2, #1
            b LOOP
Salir:
```

Módulo 1. Repaso de ARM

Acceso a estructuras básicas de datos

ec

Acceso a estructuras básicas de datos

- Acceso secuencial a un array

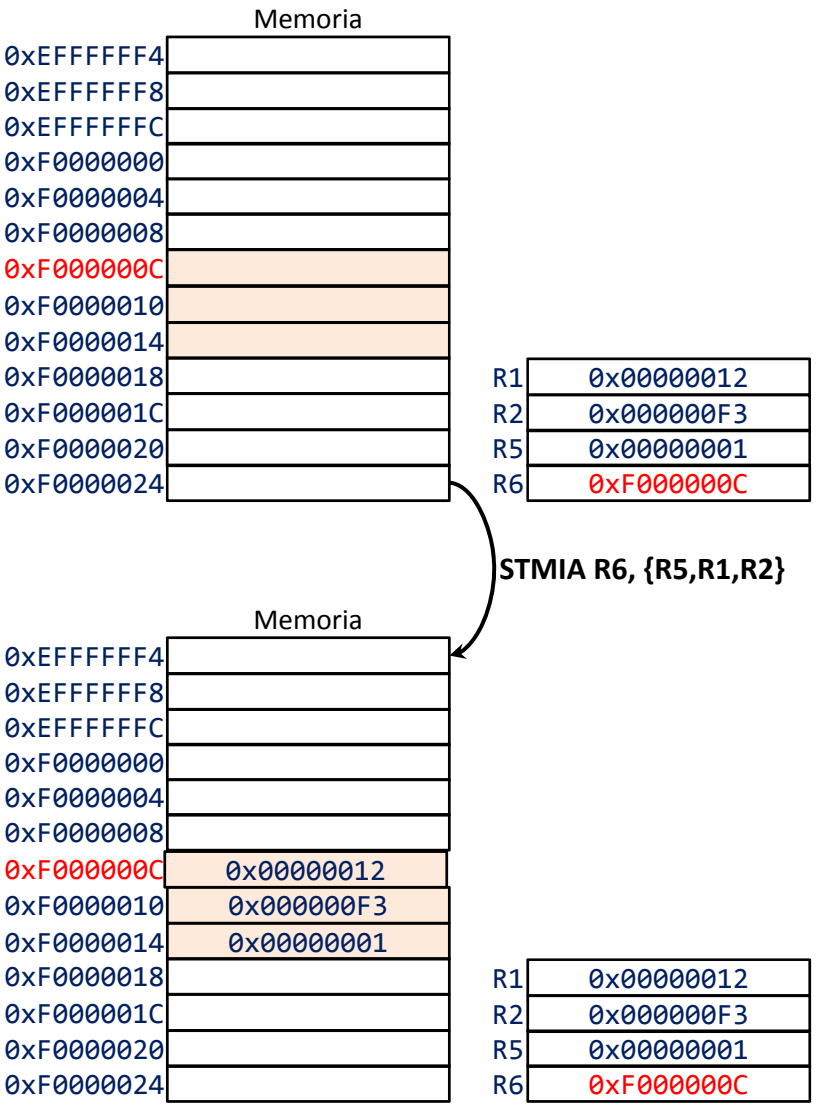
ldr rd, [ra, ri LSL #2]

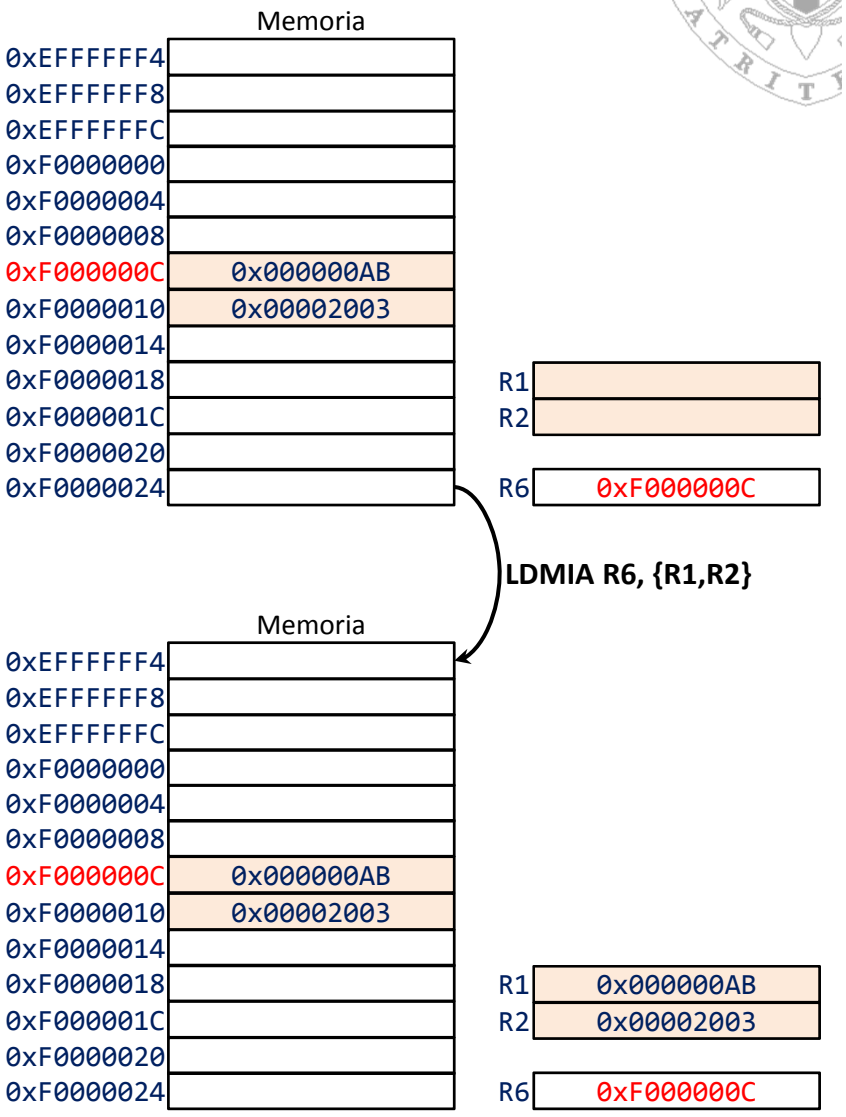
0x03FC	d[0]	0x00000001
0x0400	d[1]	0x00000003
0x0404	d[2]	0x00000007
0x0408	d[3]	0x00000005
0x040C	d[4]	0xFFFFFFFD
…		…
0x0420	d[9]	0x00000004

Acceso a estructuras básicas de datos

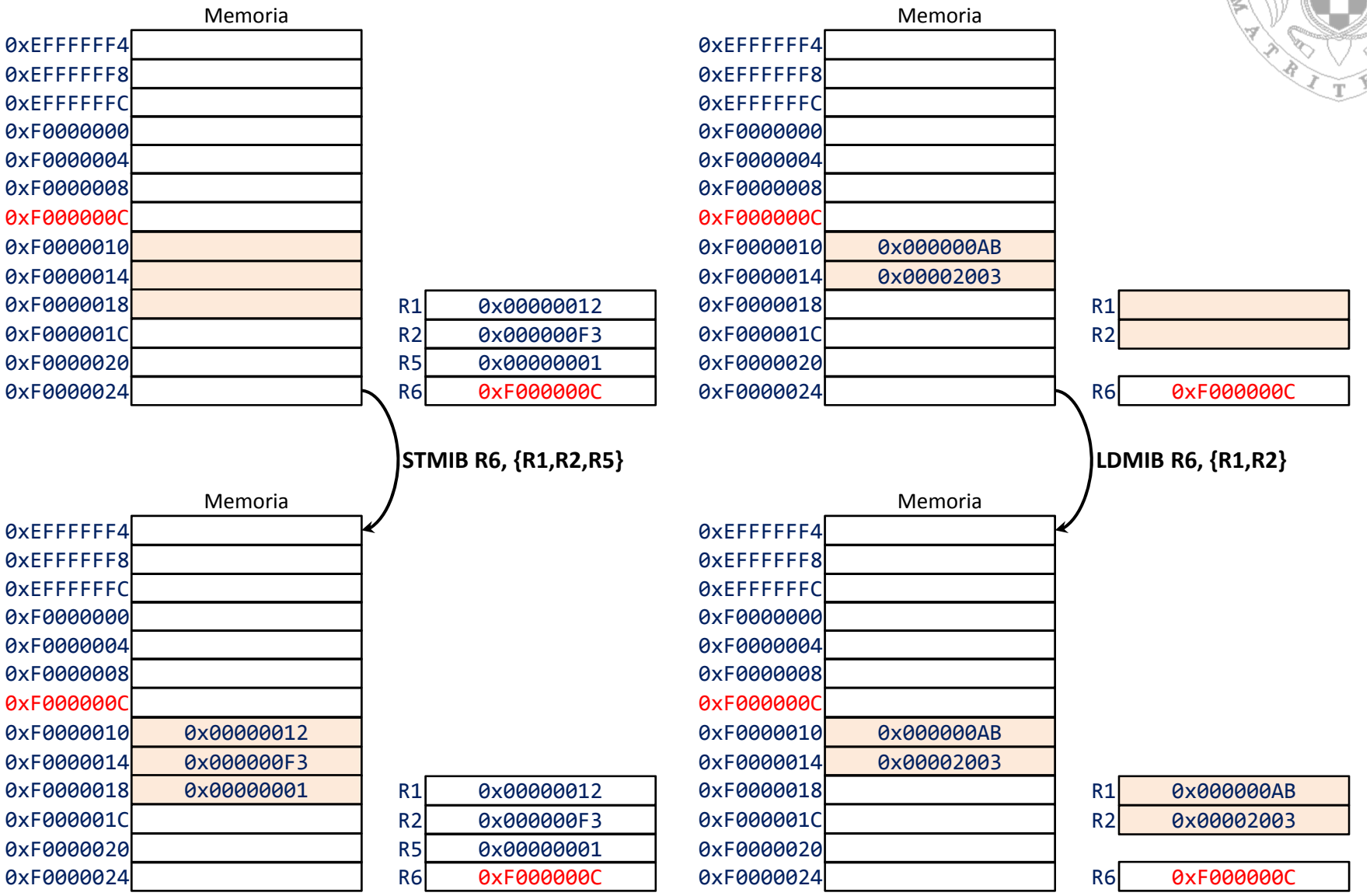
- Acceso a una estructura (acceso múltiple)
  - Sintaxis:
    - <LDM|STM>{<cond>}<modo_direc> Rb{!}, <lista registros>
  - 4 Modos de direccionamiento:
    - LDMIA / STMIA incrementar después (after)
    - LDMIB / STMIB incrementar antes (before)
    - LDMDA / STMDA decrementar después
    - LDMDB / STMDB decrementar antes

Acceso a estructuras básicas



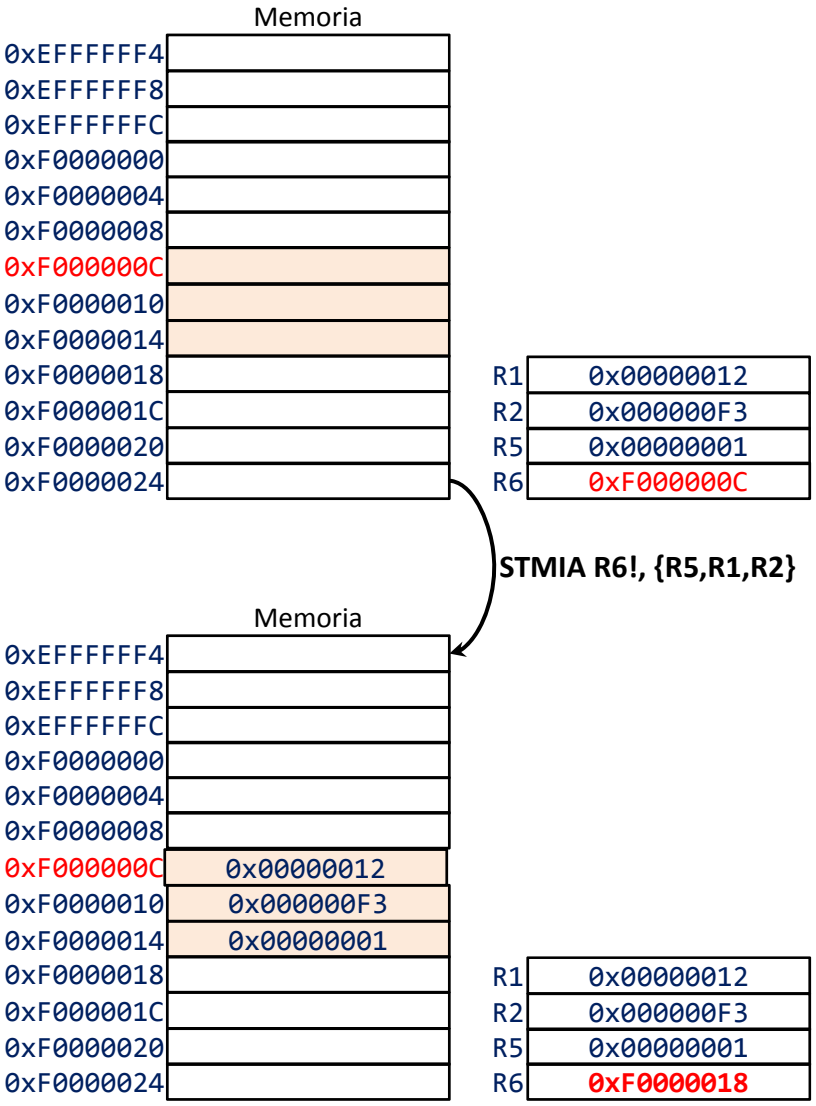


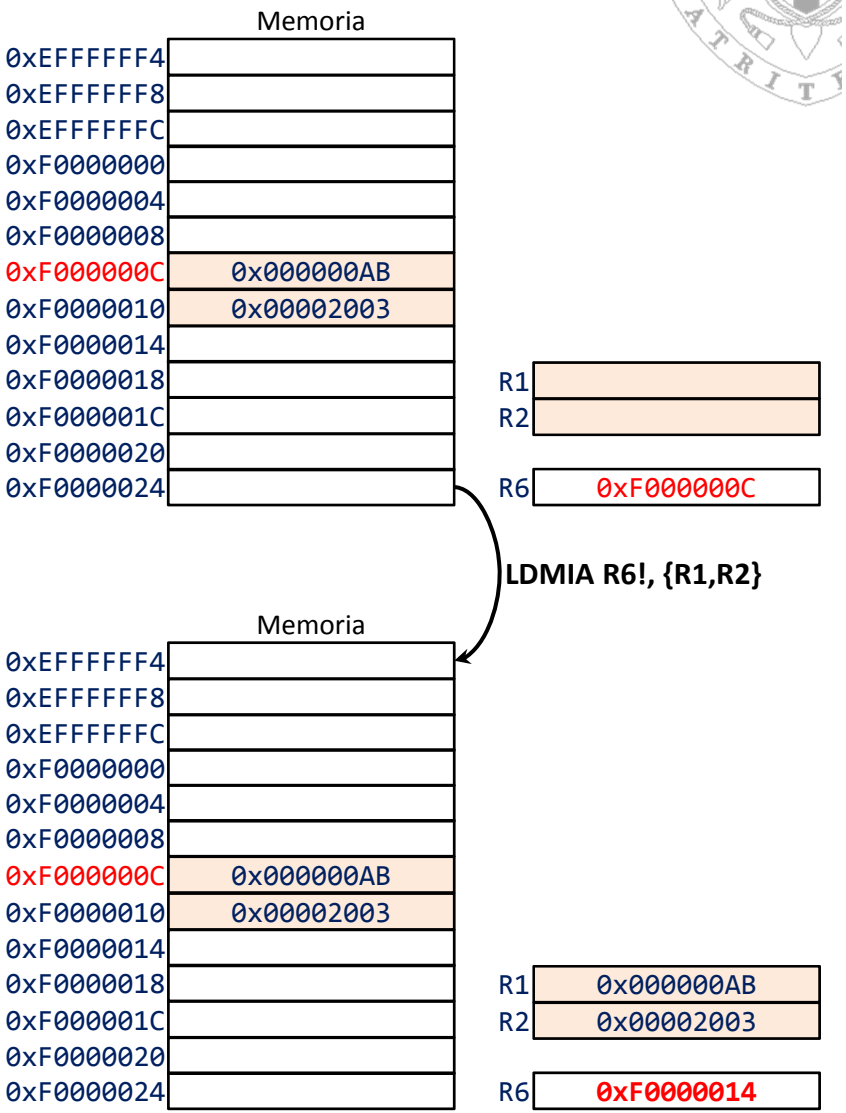
Acceso a estructuras básicas



ec

Acceso a estructuras básicas





ec

Acceso a estructuras básicas de datos

- Acceso a pila:
  - La pila es una región continua de memoria, cuyos accesos siguen una política LIFO (*Last-In-First-Out*)
  - La gestión de tipo LIFO se lleva a cabo mediante un puntero al último elemento de la pila (*cima*) que recibe el nombre de *stack pointer* (SP)
  - La pila crece hacia direcciones inferiores
  - Más adelante veremos la gestión de la pila asociada a subrutinas (*Marco de Pila de la subrutina*)

Acceso a estructuras básicas de datos

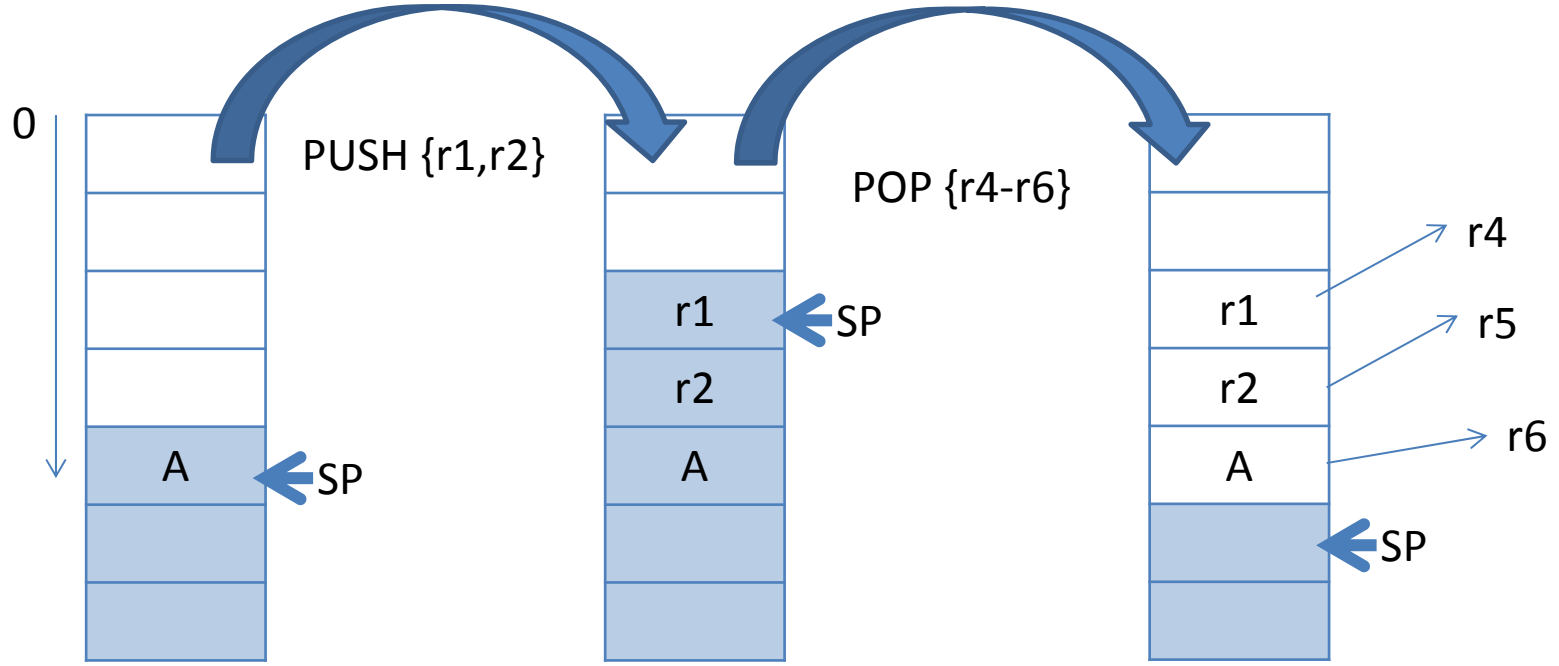
- PUSH: Introducir en la pila
  - PUSH {r1, r2, r3}
  - PUSH {r1, lr}

  sp = sp - 4·N
- POP: Sacar de la pila
  - POP {r1, r3-r5}
  - POP {r2, pc}

  sp = sp + 4·N

Acceso a estructuras básicas de datos

- Ejemplo funcionamiento de la pila



Acceso a estructuras básicas de datos

- En ARM se puede acceder también a media palabra (16 bits) o a un Byte
- Se utilizan la instrucciones de load o store estudiadas pero añadiéndoles un sufijo .H o .B
- En ARM tanto si se accede a una palabra o media palabra, ésta tiene que estar alineada en memoria
  - Palabra → dirección múltiplo de 4
  - Media palabra → dirección múltiplo de 2

Acceso a estructuras básicas de datos

0x0C000018	A1	92	83	75

R1 : 0x0C000018
R2 : 0x89ABCDEF

¿Qué ocurre con R1 y R2 si las siguientes instrucciones se ejecutan de manera independiente?

LDR R2,[R1]

LDRH R2,[R1, #2]!

LDRB R2,[R1, #3]!

LDRSB R2,[R1]

LDRSH R2,[R1]

Acceso a estructuras básicas de datos

0x0C000018	A1	92	83	75

R1 : 0x0C000018
R2 : 0x89ABCDEF

¿Qué ocurre con R1 y en memoria si las siguientes instrucciones se ejecutan de manera independiente?

STR R2,[R1]

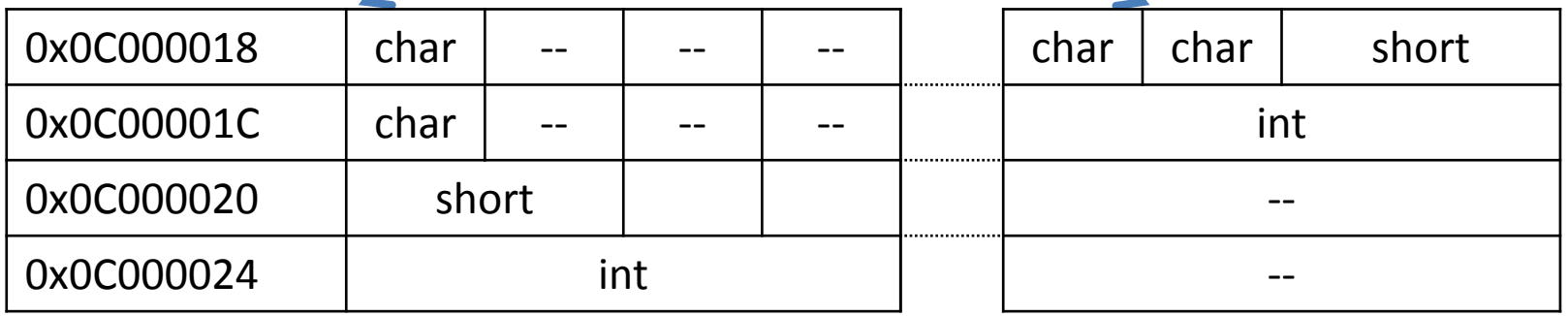
STRH R2,[R1], #2

STRB R2,[R1, #1]!

Acceso a estructuras básicas de datos

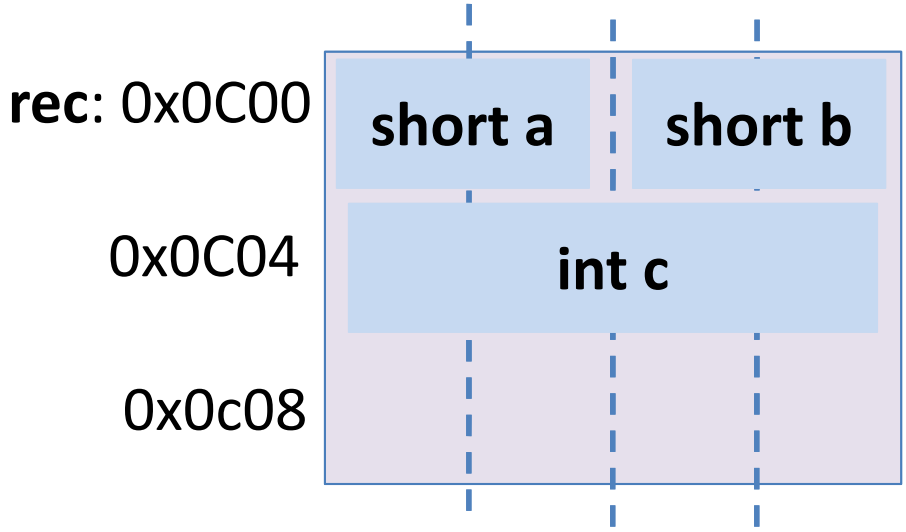
Tipo C	Número de bits
char	**8**
short	**16**
int	**32**
long	**32**
Los mismo tamaños se aplican para la versión unsigned	

Cómo se almacenan en memoria



Acceso a estructuras básicas de datos

```
struct mistruct {
      short int a;
      short int b;
      int c;
};
struct mistruct rec;
```



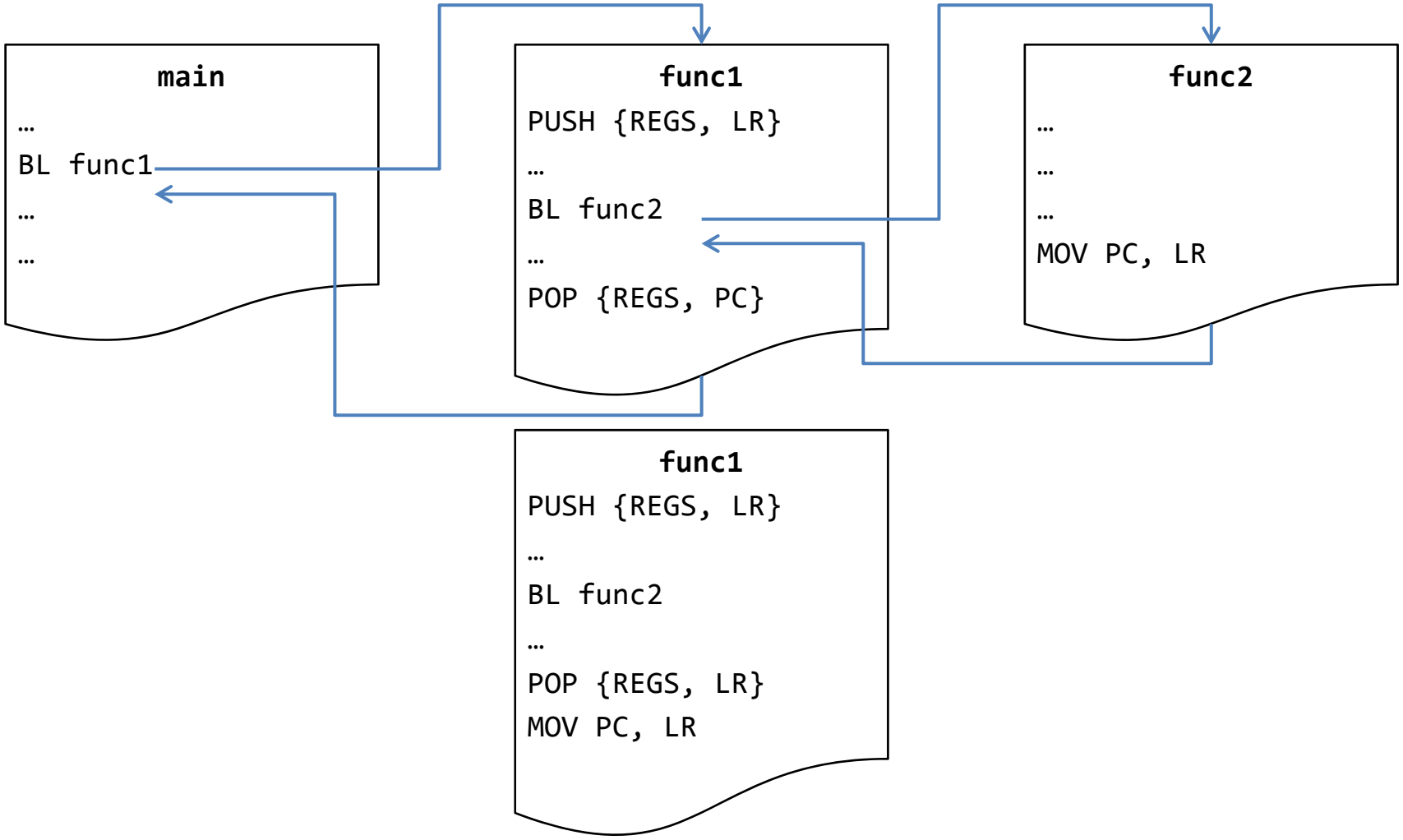
- Los elementos de un *struct* ocupan posiciones consecutivas de memoria
- ¿Cómo sería un *array* de *structs*? ¿Y un *struct* con un *array* como elemento?

Módulo 1. Repaso de ARM

Subrutinas

ec

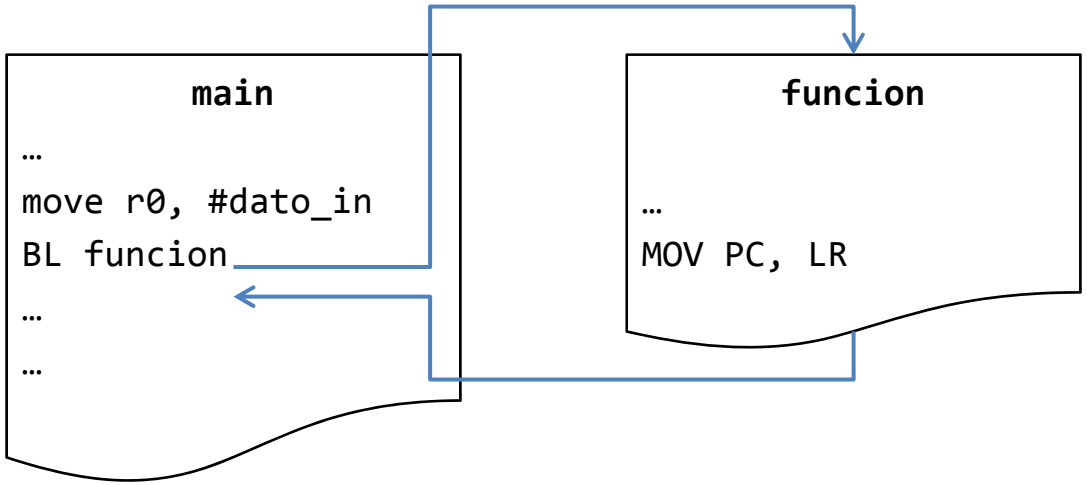
Subrutinas


main
…
BL func1
…
…
func1
PUSH {REGS, LR}
…
BL func2
…
POP {REGS, PC}
func2
…
…
…
MOV PC, LR
func1
PUSH {REGS, LR}
…
BL func2
…
POP {REGS, LR}
MOV PC, LR

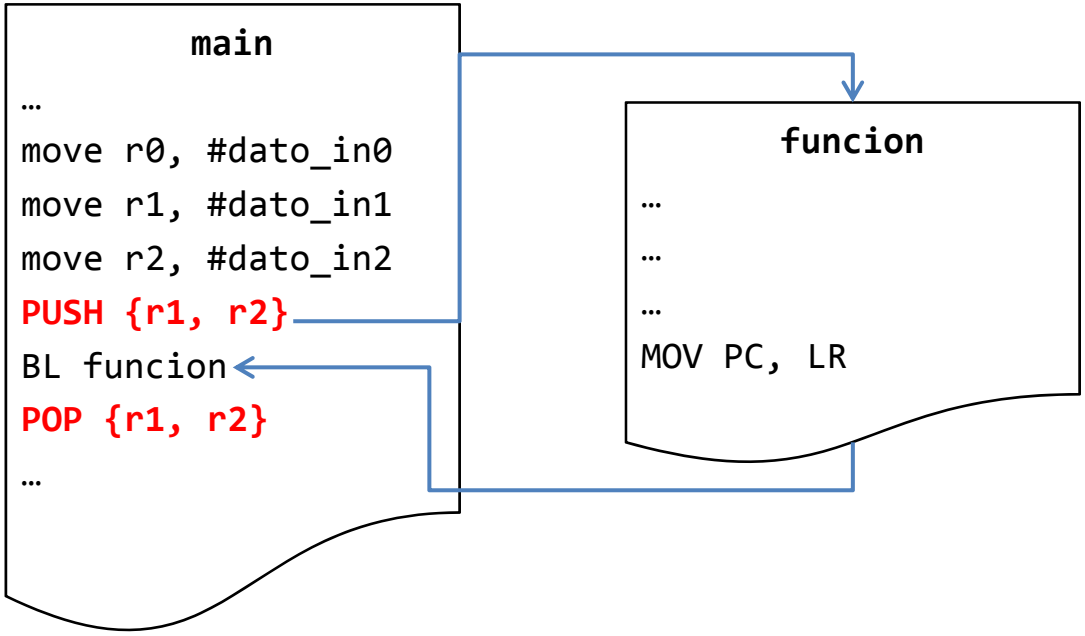
Subrutinas

- ARM Procedure Call Standards (AAPCS)
  - lr: link register (r14), mantiene la dirección a la cual hay que volver tras ejecutar la subrutina, su valor se puede cargar directamente en el PC para volver de la subrutina
  - Los registros de r0 a r3 sirven para pasar los argumentos
  - r0 para escribir el resultado
  - De r4 a r11 para almacenar las variables locales
  - Como preservar la integridad del código:
    - Si el **programa que llama a la subrutina** necesita los valores almacenados de r0 a r3 después de llamar a la subrutina, debe preservar su valor antes de llamar a la subrutina
    - Si la **subrutina** modifica el valor de los registros de r4 a r8, como no conoce si el programa que la ha invocado los necesita debe preservar su valor
    - Esto implica la utilización de un prólogo (para preservar los valores) y un epílogo (para recuperarlos) por parte del programa que llama a la subrutina y por parte de la subrutina
    - Este proceso se resuelve fácilmente mediante la utilización de una pila

Subrutinas

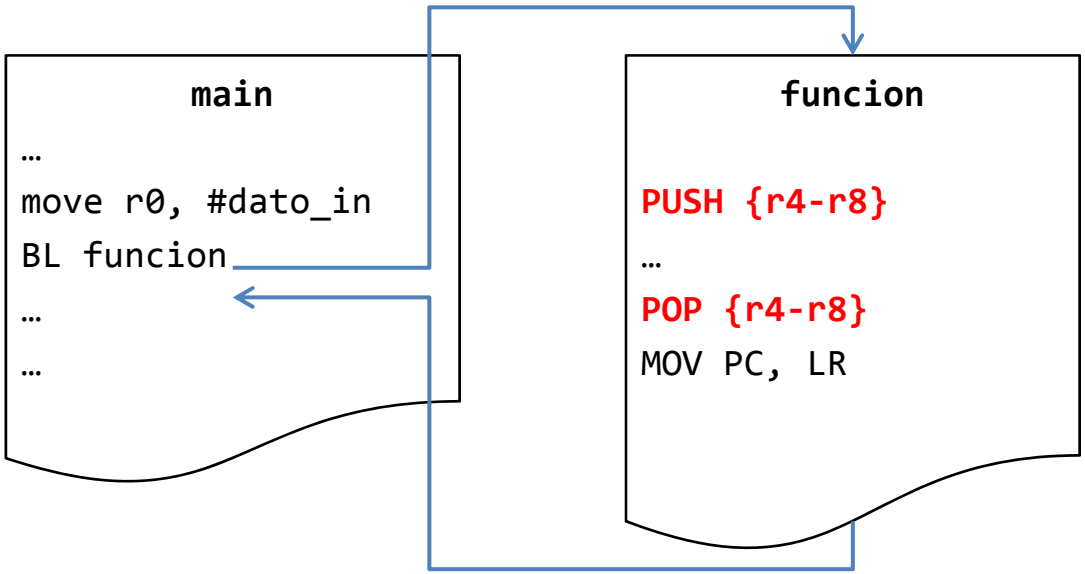


La subrutina funcion no utiliza ningún registro de r4 en adelante para calcular el valor a devolver



El programa main utiliza el valor de dos de los parámetros de entrada (r1 y r2) después de llamar a la subrutina

Subrutinas

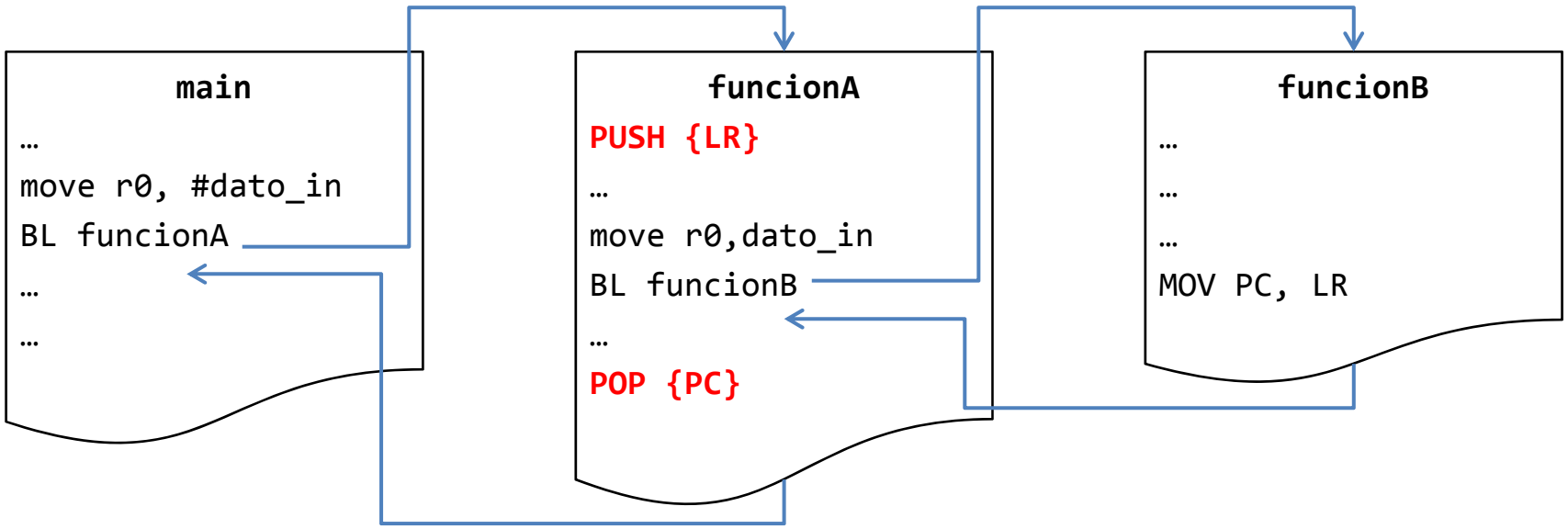


La subrutina funcion utiliza los registros de r4 a r8 para calcular el valor a devolver

Subrutinas

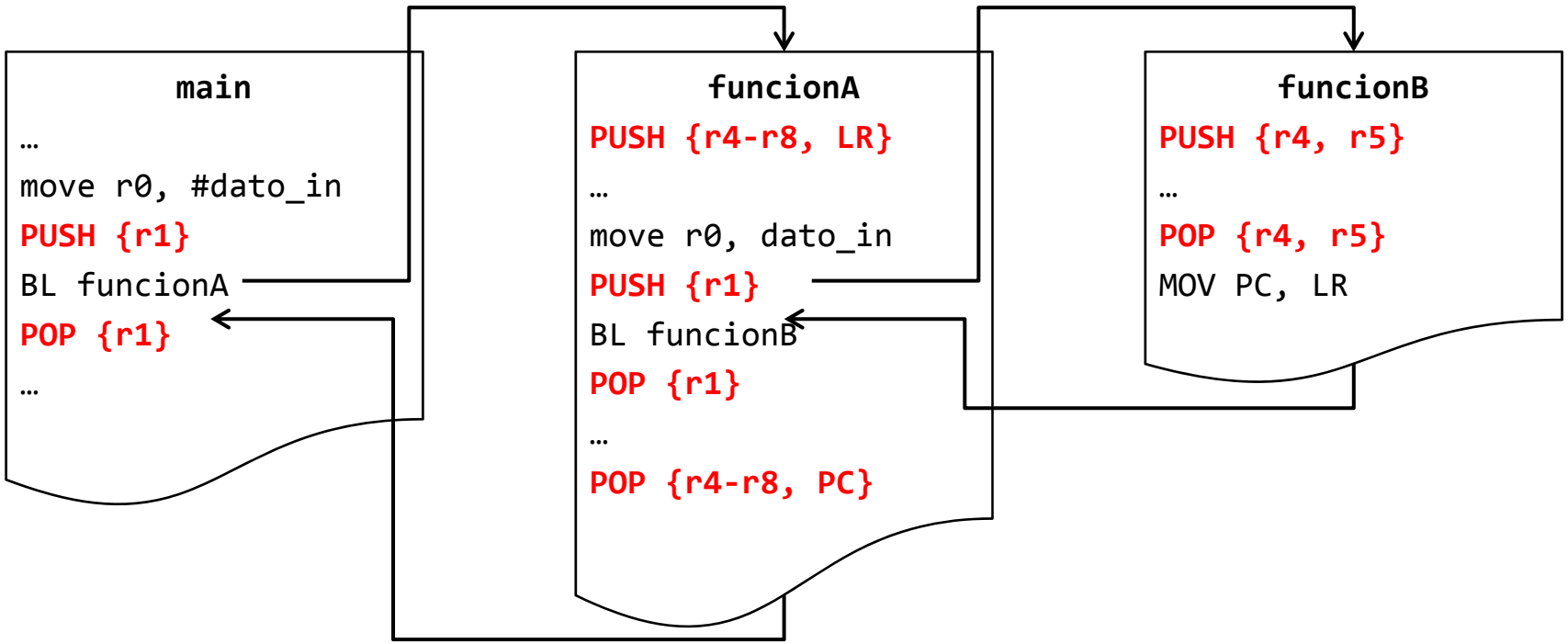
- Si una subrutina llama a otra subrutina, se le denomina **subrutina no hoja**, tiene que comportarse como una *subrutina* y como un *programa invocante*
  - Es obligatorio que el prólogo de la subrutina no hoja tenga un PUSH {lr}
  - Es obligatorio que el epílogo de la subrutina no hoja tenga un POP {pc}

Subrutinas



La subrutinas funcionA y funcionB no utilizan ningún registro de r4 en adelante para calcular el valor a devolver

Subrutinas

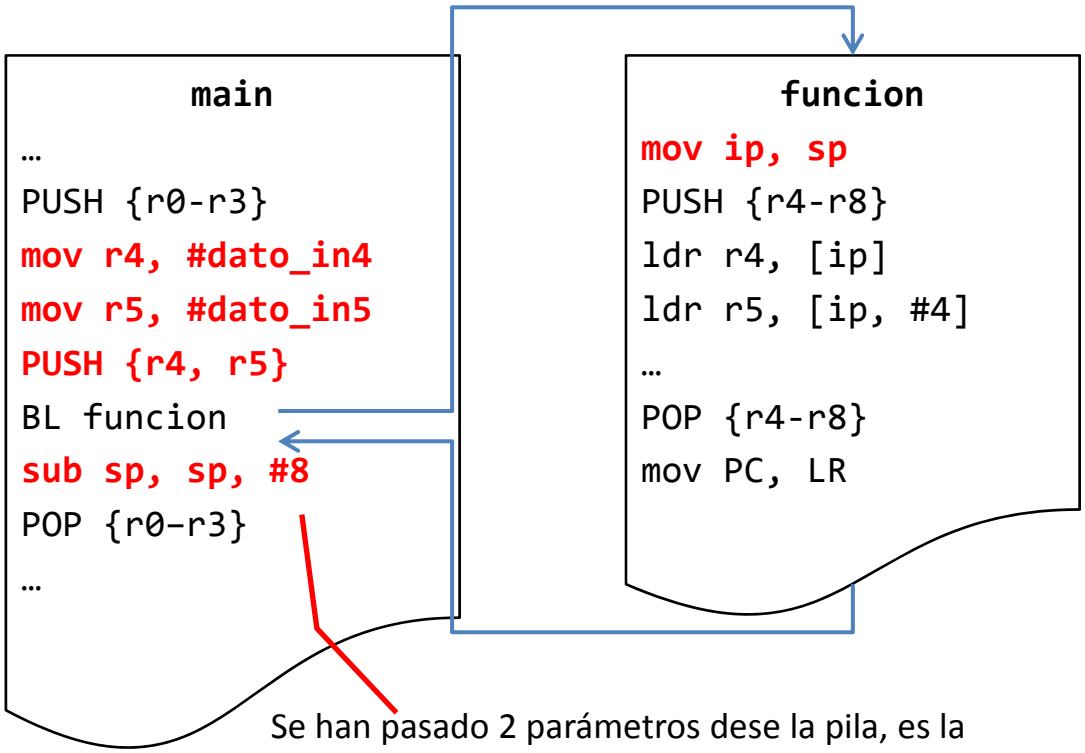


El programa main y las subrutinas funcionA y funcionB guardan multiples registros para que su valor no se vea afectado por llamar a subrutinas

Subrutinas

- Qué ocurre cuando una subrutina tiene que pasar más de 4 parámetros
  - Los parámetros que se pasan de más lo hacen a través de la pila
    i. Hay que apilar los registros de los cuales interesa mantener su valor a la vuelta de la subrutina
    ii. Hay que apilar los parámetros de entrada a la subrutina que se van a pasar a través de la pila
  - En la pila tenemos dos regiones diferenciadas, una dedicada a mantener el contexto y otra dedicada a almacenar los parámetros propios de la subrutina
  - Puede ser interesante tener dos punteros

Subrutinas

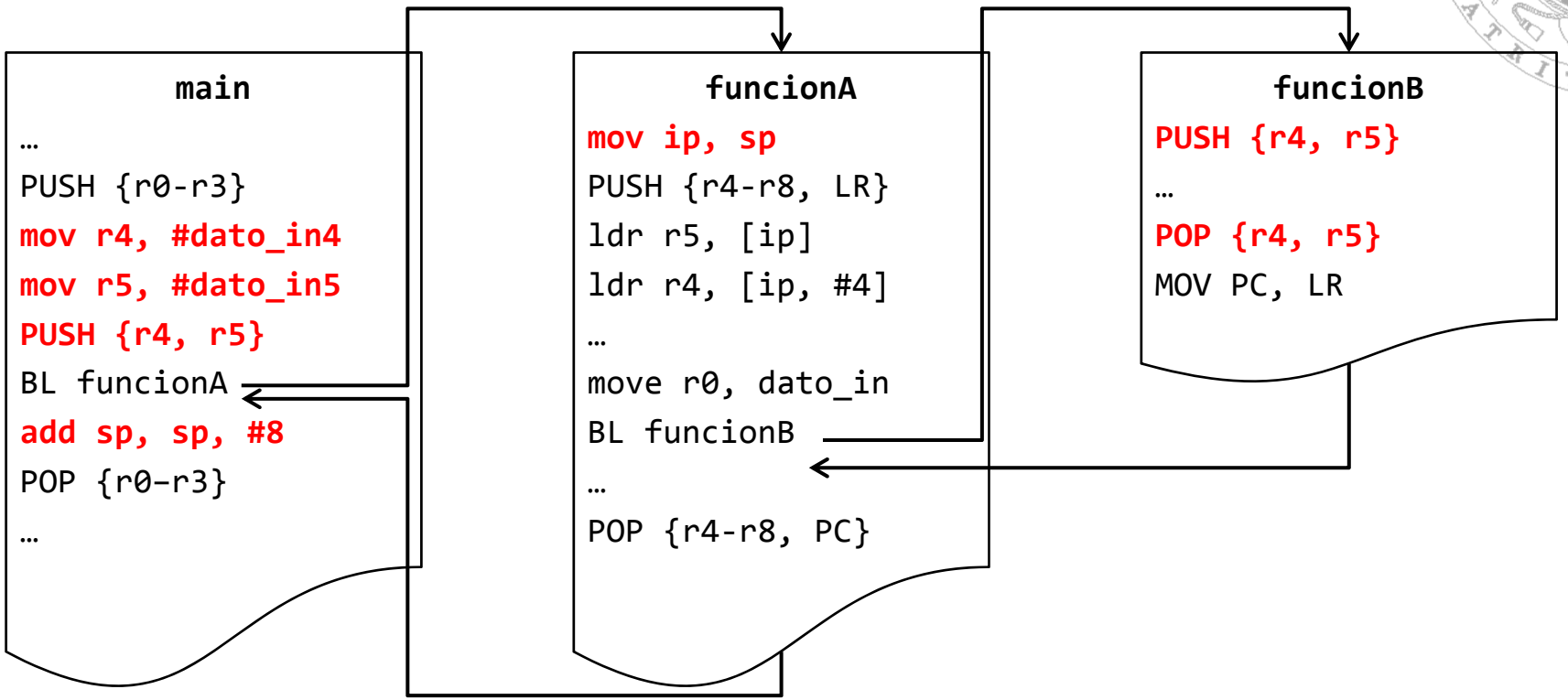


Se han pasado 2 parámetros dese la pila, es la instrucción equivalente a POP {r4, r5} (cualquiera de las dos sería válida)

La subrutina funcion utiliza dos parámetros por pila

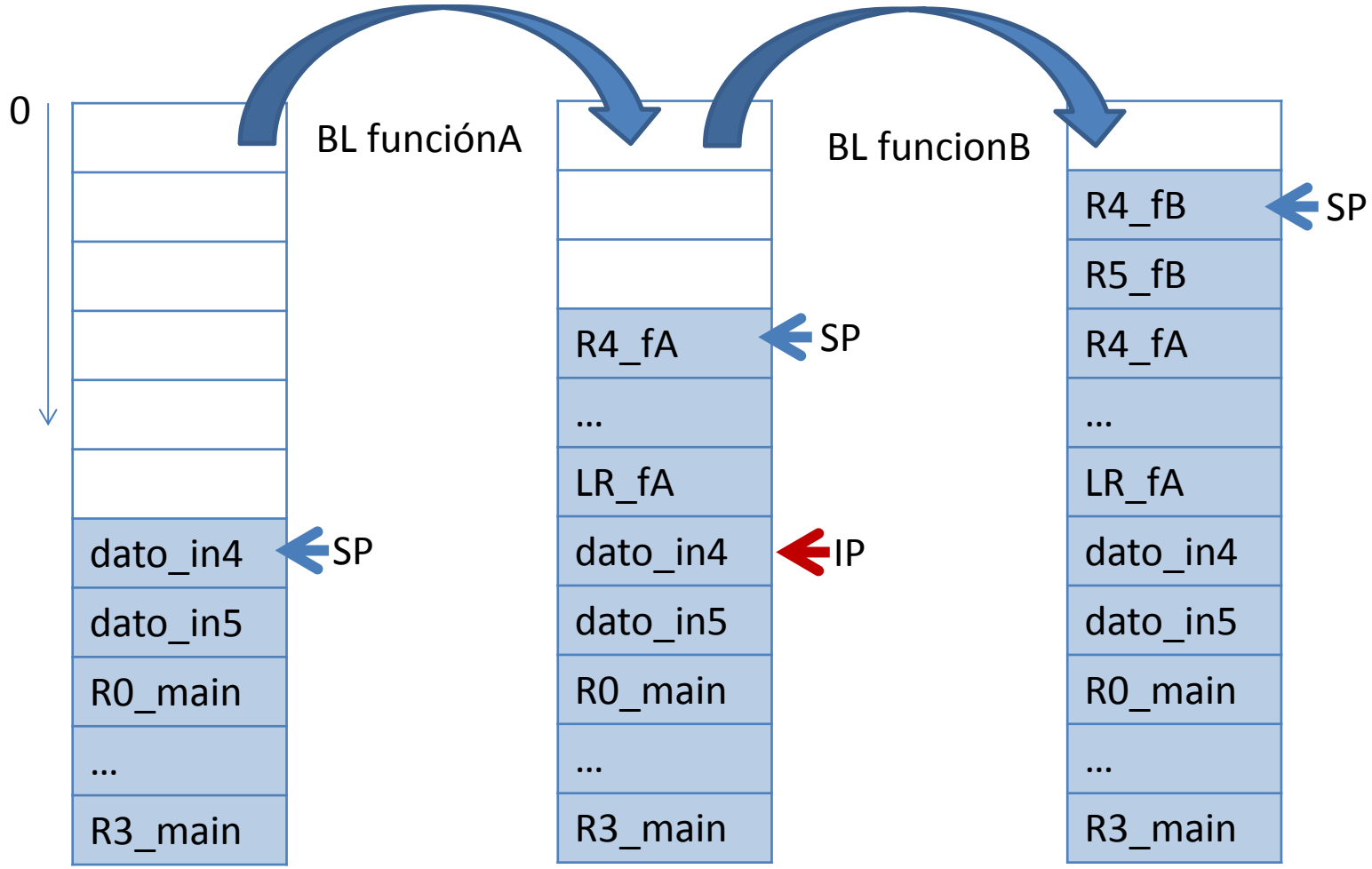
El registro ip (r12) se denomina según el estandar “Intra-procedure-call scratch register”: registro “borrador” para llamadas entre procedimientos

Subrutinas

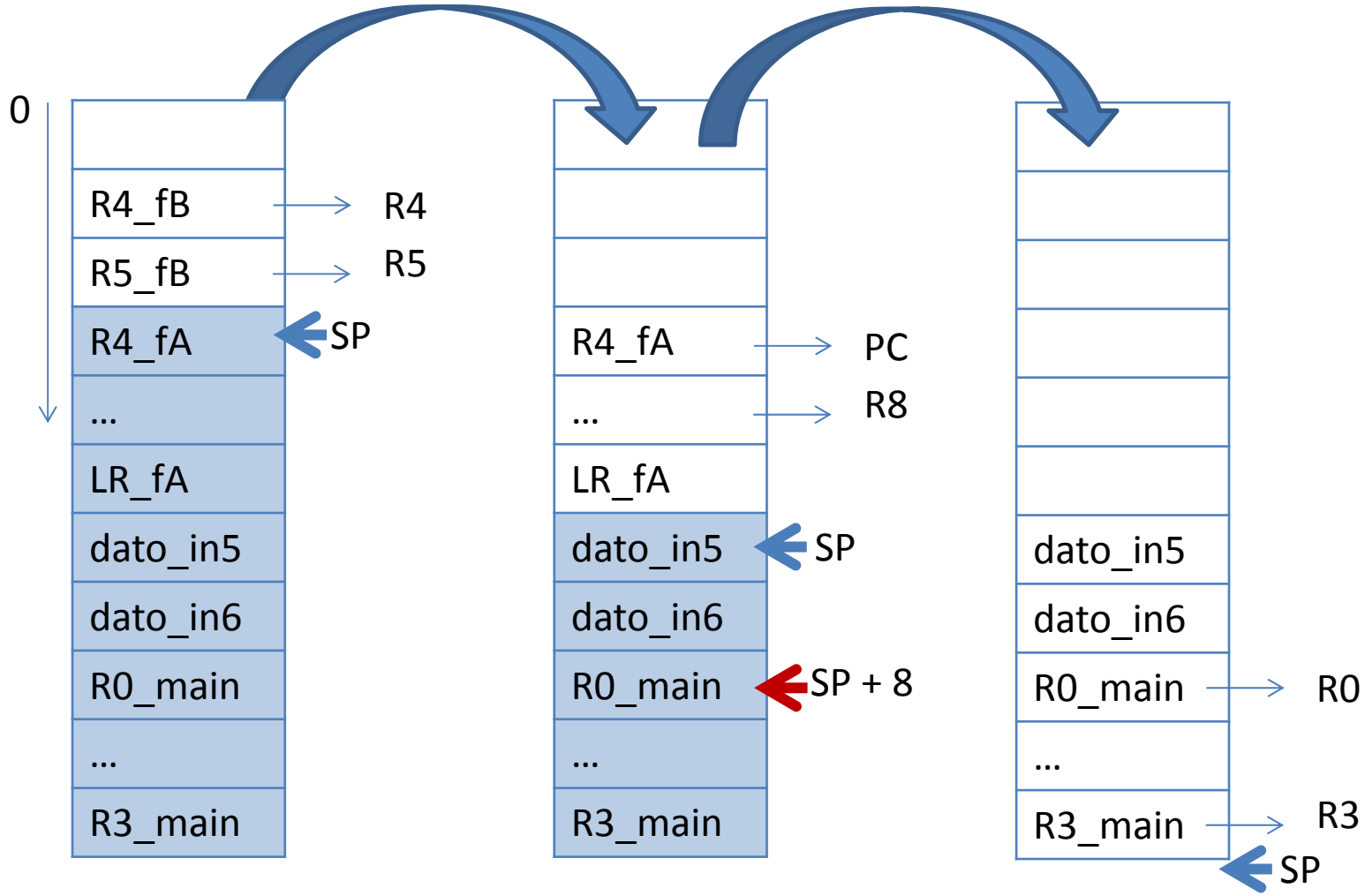


El programa main pasa a la subrutina funcionA 6 parámetros

Subrutinas


0
BL funciónA
BL funcionB
dato_in4
SP
dato_in5
R0_main
...
R3_main
R4_fA
SP
...
LR_fA
dato_in4
IP
dato_in5
R0_main
...
R3_main
R4_fB
SP
R5_fB
R4_fA
...
LR_fA
dato_in4
dato_in5
R0_main
...
R3_main

Subrutinas


0
R4_fB
R5_fB
R4_fA
...
LR_fA
dato_in5
dato_in6
R0_main
...
R3_main
R4
R5
SP
R4_fA
...
LR_fA
dato_in5
dato_in6
R0_main
...
R3_main
PC
R8
SP
SP + 8
dato_in5
dato_in6
R0_main
...
R3_main
R0
R3
SP

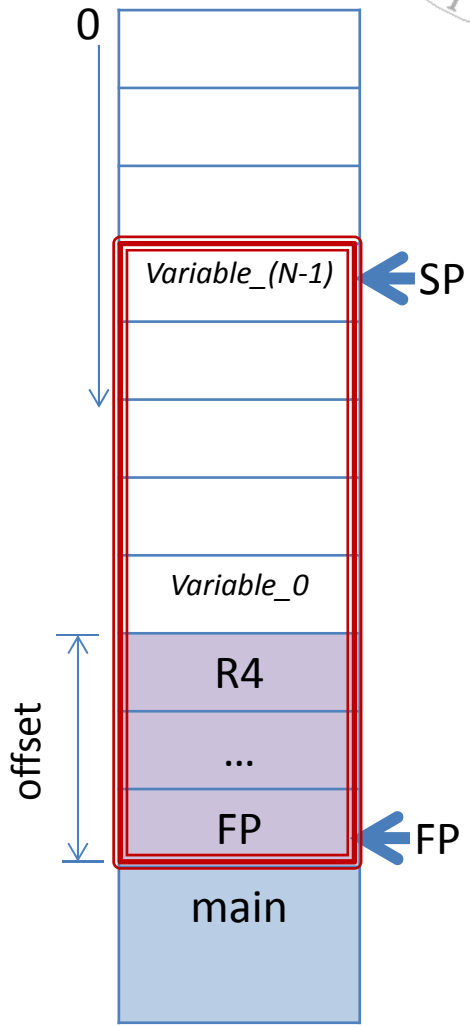
Subrutinas

- Marco de activación (frame pointer)
  - Puntero a el primero de los parámetros que se pasa a través de la pila
  - Cada subrutina puede tener su propio marco de activación que conviene preservar cuando se llama a una subrutina
  - Utilizar el FP facilita el acceso a los variables locales de la subrutina pero no es estrictamente necesario
    - Se puede hacer directamente desde SP
    - Según el estándar AAPCS su uso está obsoleto, de esta forma el ARM posee un registro más de “propósito general”
  - La versión de gcc que vamos a utilizar en el laboratorio utiliza FP

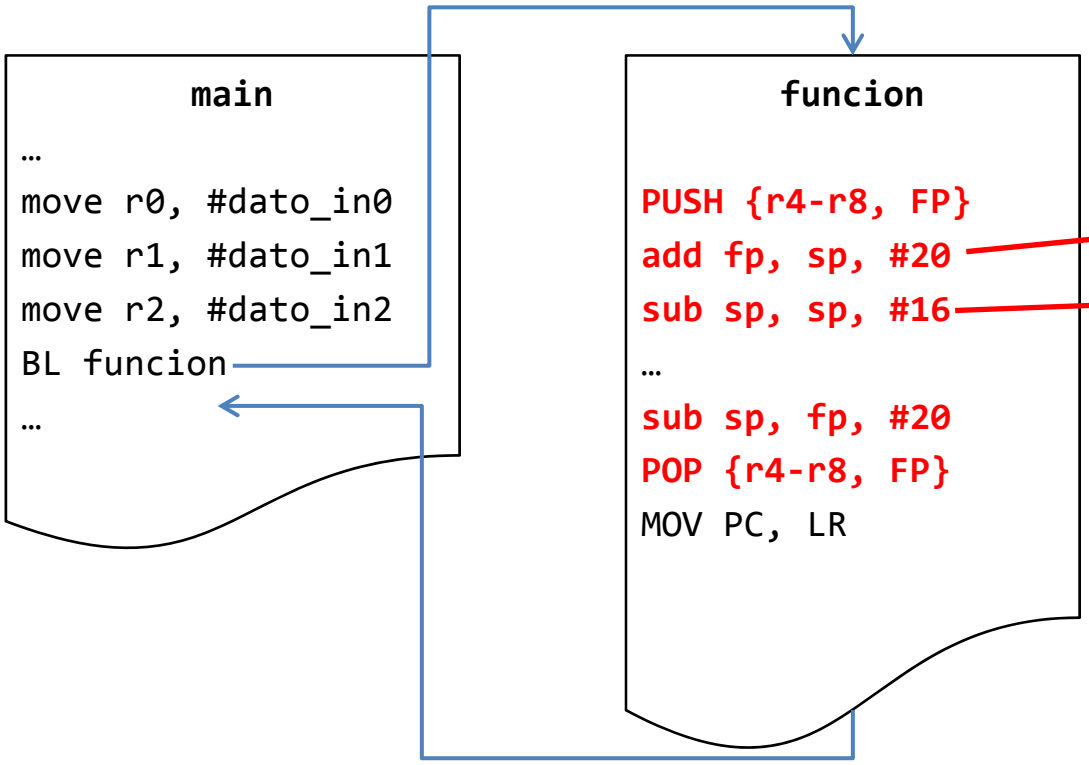
Subrutinas

- Prólogo utilizando FP
  - i. Preservar FP de ***main*** y los registros que se van a modificar en ***funcion*** de r4 a r10
  - ii. Calcular el nuevo FP
    - El FP para ***funcion*** será puntero al primer registro apilado en su marco de activación
    - Partiendo del valor de SP se puede calcular el valor de FP
    - El nuevo FP = SP + 4*(nº de parametros del prólogo-1)
  - iii. Calcular el marco
    - El marco de la subrutina empieza en FP y acaba en un nuevo SP
    - SP = SP - 4*(nº de variables)
  - iv. Obtener las variables
    - Cada variable se encuentra en:

    FP - (offset + 4*#variable)



Subrutinas

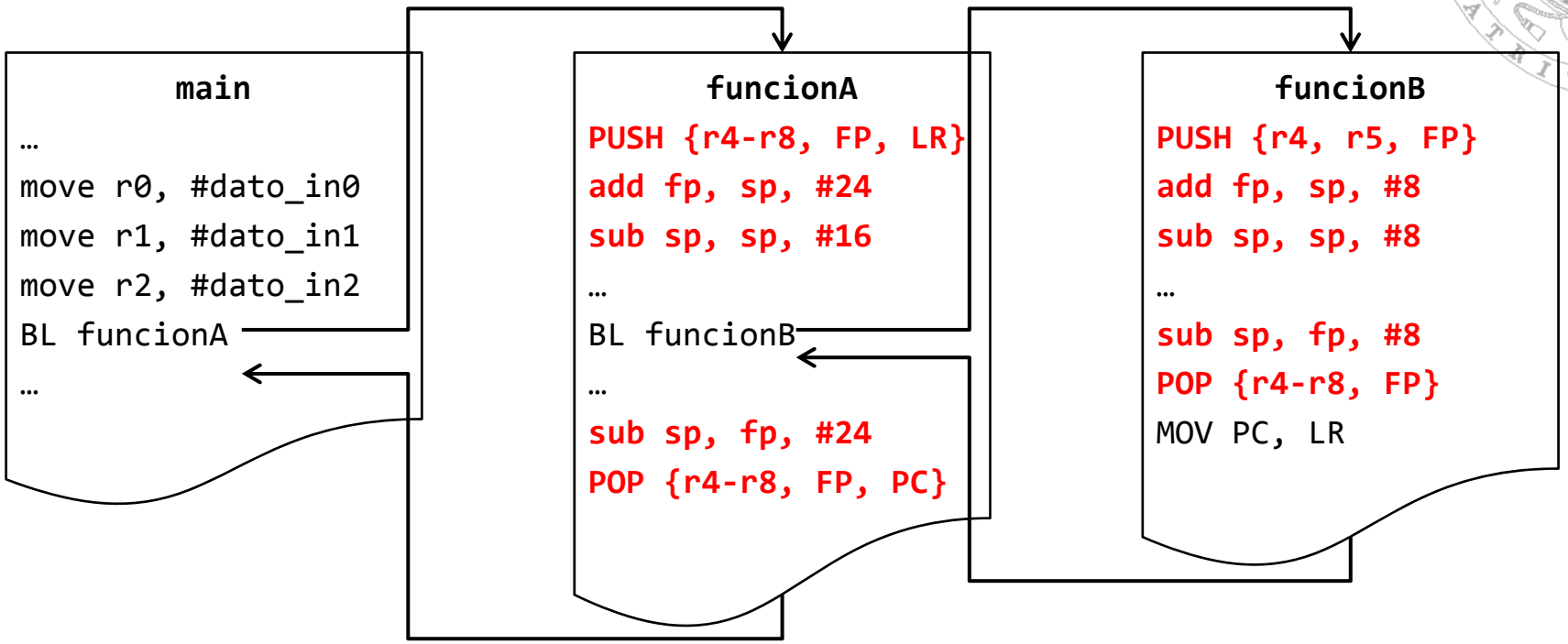


4*(Número de registros apilados -1)

La subrutina necesita espacio para 4 palabras en su marco

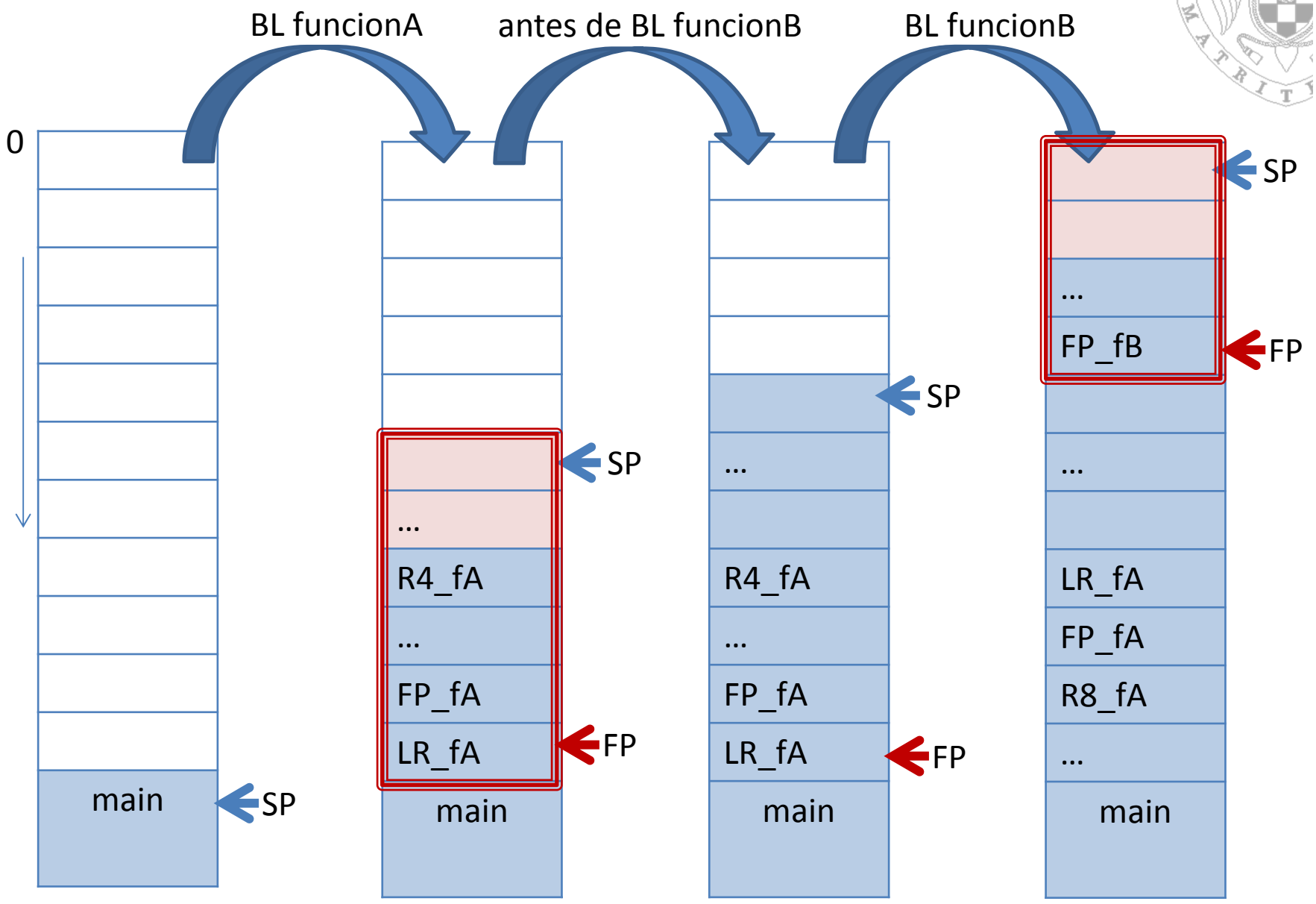
La subrutina funcion utiliza los registros de r4 a r8 para calcular el valor a devolver

Subrutinas



Las subrutinas funcionA y funcionB guardan múltiples registros para que su valor no se vea afectado por llamar a subrutinas

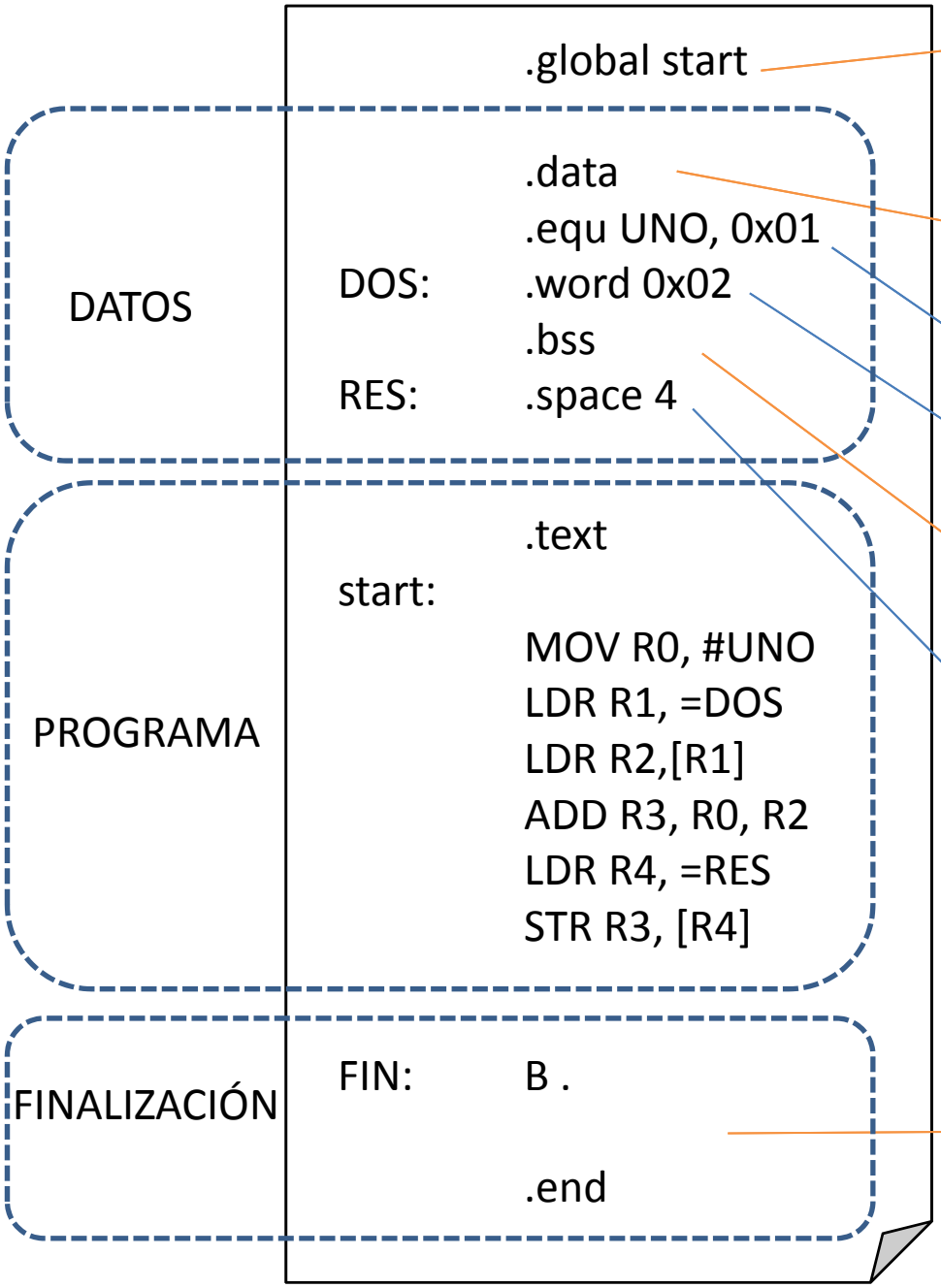
Subrutinas


BL funcionA
antes de BL funcionB
BL funcionB
0
SP
main
SP
...
R4_fA
...
FP_fA
LR_fA
FP
main
SP
...
R4_fA
...
FP_fA
LR_fA
FP
main
SP
...
FP_fB
FP
...
LR_fA
FP_fA
R8_fA
...
main

Módulo 1. Repaso de ARM

Estructura básica de un programa en ensamblador

ec



Indica en qué dirección (etiqueta) empieza el código ejecutable del programa

A partir de .data se enumeran las variables de entrada

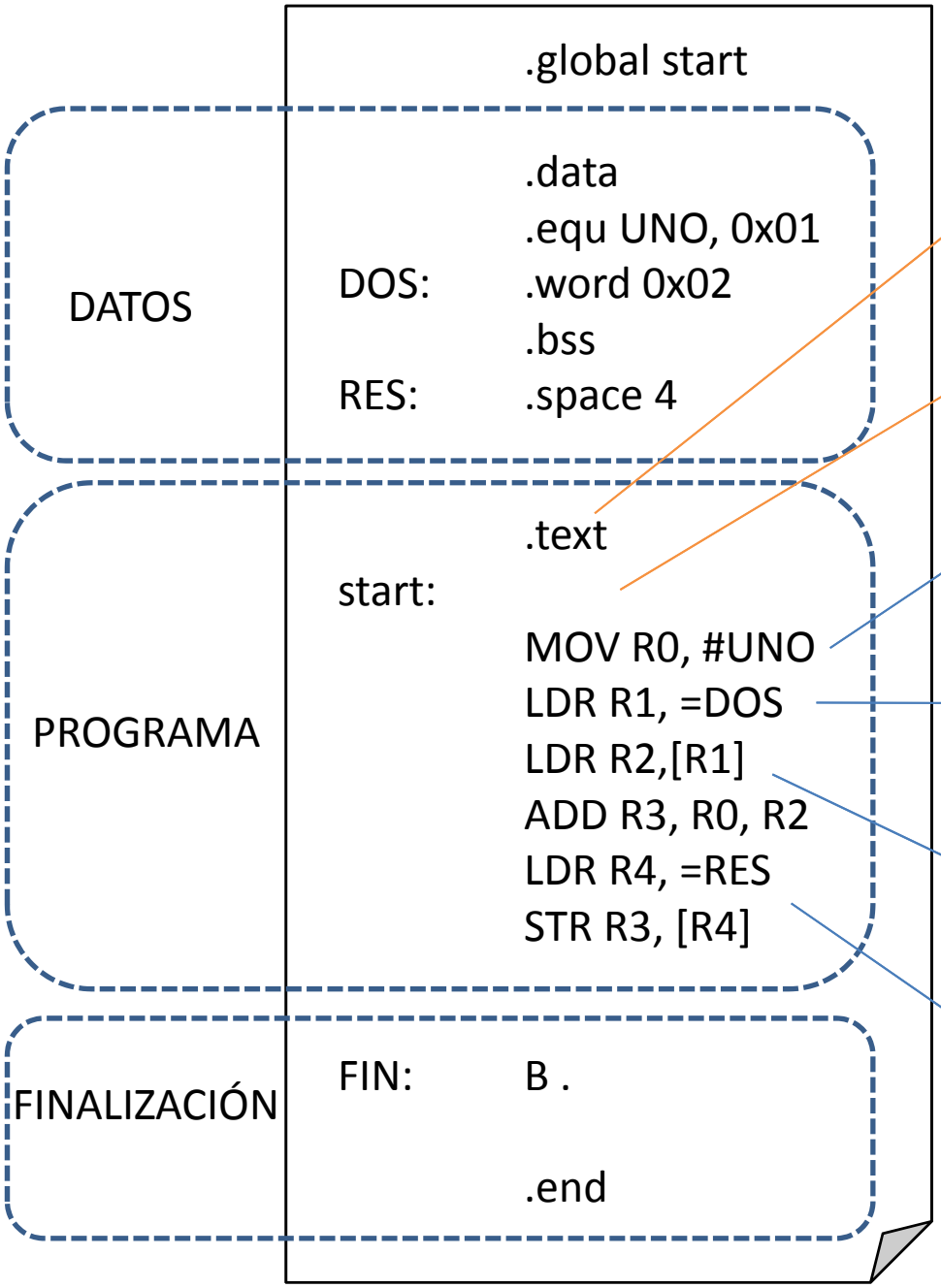
Se define la constante UNO

Se define la variable DOS y se inicializa a 2 (en hexadecimal)

A partir de .bss se enumeran las variables de salida

Se define la variable RES y se reserva espacio en memoria para almacenarla (4 bytes)

Estas 2 instrucciones deben escribirse siempre para finalizar el programa



A partir de **.text** están las instrucciones que forman el programa

Esta "etiqueta:", debe coincidir con la de **.global** y se coloca dónde comienza el programa

Mueve el valor de la constante UNO a R0 para poder operar con ella.

Guarda en R1 la dirección de memoria donde está almacenada la variable DOS

Almacena en R2 el valor de la variable DOS

Almacena en memoria (en la posición reservada para la variable RES)

Estructura básica de un programa ensamblador

- Proceso de enlazado de un programa en ensamblador:
  - Alojar programa, datos y resultados en la memoria
  - Traducir las etiquetas

```
SECTIONS
{
          . = 0x0C000000;
          .text : { *(.text) }
          _bdata = .;
          .data : { *(.data) }
          _edata = .;
          .rodata : { *(.rodata) }
          _bbss = .;
          .bss : { *(.bss) }
          _ebss = .;
}
```

Estructura básica de un programa ensamblador

- Traducción de ensamblador a código máquina

0x0C000000	MOV r0, #1	E3A00001
0x0C000004	LDR r1, [pc, #16]	E59F1010
0x0C000008	LDR r2,[r1]	E5912000
0x0C00000C	ADD r3, r0, r2	E0803002
0x0C000010	LDR r1, [pc, #8]	E59F4008
0x0C000014	STR r3, [r4]	E5843000
0x0C000018	B 0xC000018	

Ya se han traducido las etiquetas: DOS, UNO …

Módulo 1. Repaso de ARM

Compilación, paso de C a ensamblador

ec

Compilación, paso de C a ensamblador

- **Programación en C**
  - Pros
    - Fácil de aprender
    - Portable, independiente de la arquitectura
    - Facilidad para construir y gestionar estructuras de datos
  - Cons
    - Limitaciones en el acceso y gestión a los registros y la pila
    - No se tiene control directo sobre la secuencia de generación de instrucciones (una instrucción de C no se corresponde con una única instrucción de ensamblador)
    - No se puede gestionar directamente la pila
- Programación en ensamblador
  - Pros
    - Permite un control directo sobre cada instrucción y sobre todas las operaciones con memoria
    - Permite la utilización de instrucciones que no genera el compilador de C
  - Cons
    - Más difícil de aprender
    - Gestión casi imposible de estructuras de datos complicadas
    - Ninguna portabilidad (sólo en la misma familia arquitectónica)
- Falsos mitos:
  - Programando directamente en lenguaje ensamblador SIEMPRE se obtienen mejores códigos que los obtenidos tras compilar C
  - C con las adecuadas opciones de compilación SIEMPRE puede obtener un código que cumpla los requisitos de rendimiento sin necesidad de tocar el código ensamblador

Compilación, paso de C a ensamblador

- ***C Startup Code* (código de arranque en C)**
  - Este código se utiliza para "inicializar" la memoria, por ejemplo con las variables globales
    - Inicializa la pila
    - Inicializa el frame pointer
  - Pone parte de memoria a cero para aquellas variables que no se inicializan en tiempo de carga (*load time*)
  - Para las aplicaciones de C que utilizan funciones como malloc(), este programa inicializa el *heap*
  - Después de estas inicializaciones, el programa (*C startup code)* salta la posición de memoria donde comienza el programa **main()**
  - El compilador/enlazador es el encargado de insertar automáticamente el programa *C startup code*
  - Este programa no es necesario crearlo si se está trabajando directamente en ensamblador
- Los compiladores de ARM denominan al *C startup code* como "_main," mientras que los compiladores de C en GNU lo suelen llamar "_start."

Compilación, paso de C a ensamblador

- Ejemplo de un código de arranque en C

```
        .extern main
        .extern _stack
        .global start
start:
        ldr sp,=_stack
        mov fp,#0
        bl main
End:
        b End
.end
```

Compilación, paso de C a ensamblador

- ***Mi Programa***
  - Conjunto de programas, bibliotecas … necesarios para ejecutar la aplicación que se está desarrollando
  - El programa a ejecutar no está sólo formado por las instrucciones necesarias para llevar acabo una tarea específica.
  - También hay varios tipos de datos:
    - Variables locales
    - Variables globales
    - Contantes, no tienen porqué ser sólo datos constantes, también pueden ser direcciones de memoria

Compilación, paso de C a ensamblador

- Tratamiento de las variables globales
  - Una variable global es aquella que puede ser accedida por cualquier función, es decir, su ámbito son todas las funciones que componen el programa
  - Puede ocurrir que queramos que esa variable sea accedida por las funciones de otro fichero.c

    extern int var1;
  - O que se accedida sólo por las funciones del fichero.c donde se encuentra

    static int var2;
  - Este mismo protocolo se puede aplicar a las funciones

- Aunque no es exactamente lo mismo, en ensamblador existen los símbolos globales
  - Debemos exportarlos con la directiva .global.
  - Además, se evita que pueda haber más de un símbolo con el mismo nombre en varios ficheros fuente

    .global start

Compilación, paso de C a ensamblador

- Mezclando C y ensamblador (I)
  - La mayor parte de los proyectos reales se componen de varios ficheros fuente.
    - La mayoría en un lenguaje de alto nivel C/C++
    - Se programa en ensamblador aquellas partes donde se tengan requisitos estrictos de eficiencia o bien porque se necesite utilizar directamente algunas instrucciones especiales de la arquitectura.
  - Para combinar código C con ensamblador resulta conveniente dividir el programa en varios ficheros fuente
    - Los ficheros en C deben ser compilados
    - Los ficheros en ensamblador sólo deben ser ensamblados.

Compilación, paso de C a ensamblador

- Mezclando C y ensamblador (II):
  - Si queremos **usar en C una rutina implementada en ensamblador**. Tenemos el problema de que el identificador usado como nombre de una rutina es la dirección de comienzo de la misma.
  - Para poder usarla en C, el símbolo (nombre de la rutina) debe haberse declarado como externo en el programa ensamblador

```
.global subrutinaARM
```

  - Además debemos declarar en el fichero C una función con el mismo nombre que dicho símbolo, así como el tipo de todos los parámetros que recibirá la función y el valor que devuelve

```
extern int subrutinaARM( int, int);
```

  - El mismo protocolo se sigue para las variables compartidas

Compilación, paso de C a ensamblador

UTILIZADA

programa.c

```
…
extern int subrutinaARM( int, int);
…
valor = subrutinaARM(A, B);
…
```

DEFINIDA

ensamblado.asm

```
.global subrutinaARM
…
.text
subrutinaARM:
… @cuerpo de la función
```

programa.c

```
…
extern int valor;
…
valor = 0x00;
…
```

ensamblado.asm

```
.bss
.global valor
valor:      .word
…
.text
```

Compilación, paso de C a ensamblador

- Mezclando C y ensamblador (III):
  - Si queremos usar en ensamblador una rutina implementada en C, el nombre de la rutina debe haberse declarado como externo en C

    extern int subrutinaC (**int, int**);

  - Además debemos declarar a la rutina en C en el código ensamblador

    .extern subrutinaC;

  - El mismo protocolo se sigue para las variables compartidas

Compilación, paso de C a ensamblador

UTILIZADA

ensamblado.asm

```
.global start
...
.extern subrutinaC
...
.text
start:
...
BL subrutinaC
...
```

ensamblado.asm

```
.global start
...
.extern valor
...
.text
start:
...
LDR r1, =valor
...
```

DEFINIDA

programa.c

```
...
extern int subrutinaC( int, int)
{
// cuerpo de la funcion
}
```

programa.c

```
...
extern int valor;
...
```

Compilación, paso de C a ensamblador

- Resolución de símbolos
  - La etapa de compilación se hace de forma independiente sobre cada fichero
  - Posteriormente es necesario resolver todas las referencias cruzadas entre los ficheros objeto
    - El contenido de un fichero objeto es independiente del lenguaje en que fue escrito el fichero fuente. Es un fichero binario estructurado que contiene una lista de secciones con su contenido, y una serie de estructuras adicionales
    - Tabla de Símbolos contiene información sobre los símbolos utilizados en el fichero fuente.
  - Las Tablas de Símbolos de los ficheros objeto se utilizan durante el enlazado para resolver todas las referencias pendientes, esta tabla indica:
    - Si un símbolo está definido en su fichero fuente asociado o tenemos que buscarlo en otro fichero
    - En qué sección se encuentra almacenado
    - Su posición en memoria relativa a la sección
    - Y su tamaño

Name	Value	Class	Type	Size	Line	Section
globalA	00000000	D	OBJECT	00000002		.data
globalB		U	NOTYPE			*UND*
main	00000000	T	FUNC	00000054		.text

Compilación, paso de C a ensamblador

- Creación del fichero ejecutable (I):
  i. Cada fichero en código C pasa por el preprocesador que modifica el código fuente original siguiendo directivas de preprocesado (#define, #include, etc.).
  ii. El código resultante es compilado, transformándolo en código ensamblador.
  iii. El ensamblador genera el código objeto para la arquitectura destino (ARM en nuestro caso).
    - Si partimos de código ensamblador, sólo sería necesaria esta última etapa para generar el código objeto

Compilación, paso de C a ensamblador

- Creación del fichero ejecutable (II):
  - Los ficheros objeto (fichero.o) no son todavía ejecutables.
  - El ensamblador ha generado el contenido de cada una de las secciones, pero falta decidir en qué direcciones de memoria se van a colocar dichas secciones y resolver los símbolos.
  - ¿Por qué no decide el ensamblador la dirección de cada sección?
    - El programa final se implementará generalmente con más de un fichero fuente, pero las etapas de compilación y ensamblado se hacen fichero a fichero
  - Es necesario añadir una etapa de enlazado para componer el ejecutable final

Compilación, paso de C a ensamblador