



# Fundamentos de Computadores

2º Cuatrimestre  
2013-2014

*fc<sup>2</sup>*

1

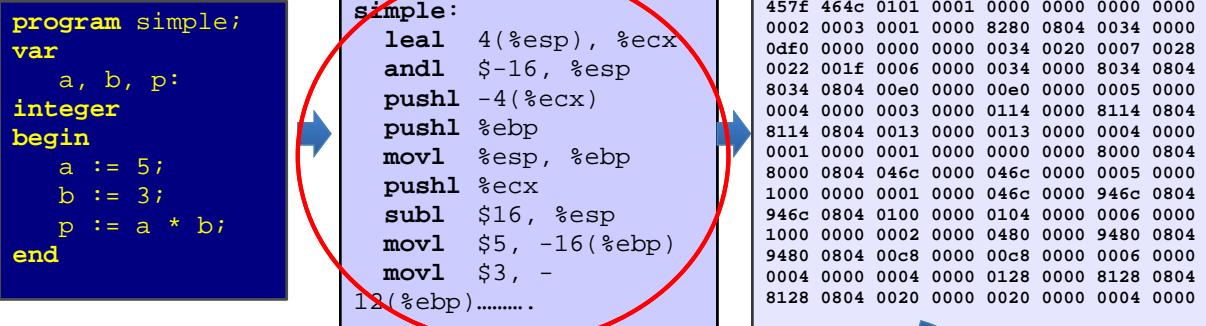


## Módulo 9: Repertorio de instrucciones y lenguaje ensamblador

*fc<sup>2</sup>*

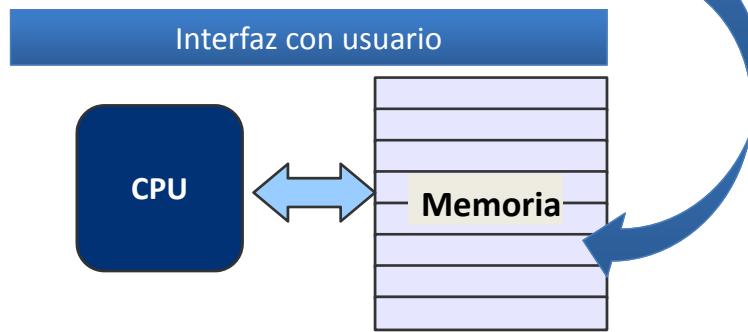
2

# ¿Qué vamos a estudiar en este tema?



fc<sup>2</sup>

3



## ¿Por qué estudiar ensamblador?

- Depende de la década donde se estudie:
  - **Los 50:** El código máquina era la única forma de programar los computadores.
  - **Los 60 y 70:** Para recodificar partes críticas del código. En 1972, sobre un PDP-9, se podía escribir código ensamblador que se ejecutaba el doble de rápido que código FORTRAN compilado.
  - **Los 80:** Para mantener gran cantidad de código heredado (legacy code), escrito en ensamblador.
  - **Hoy:** Ya hay poco “Legacy code” que mantener.

**Se estudia para entender la arquitectura de los computadores y su conexión con los compiladores de alto nivel.**

fc<sup>2</sup>

4



# ARM

- Siglas de la compañía Advanced Risc Machines Ltd.
- Fundada en 1990 por Acorn, Apple y VLSI Technologies
  - En 1993, se unió Nippon Investment and Finance
- Desarrollan procesadores RISC y SW relacionado
- NO realizan fabricación de circuitos
- Sus ingresos provienen de los “royalties” de las licencias, de las herramientas de desarrollo (HW y SW) y de servicios que ofrecen

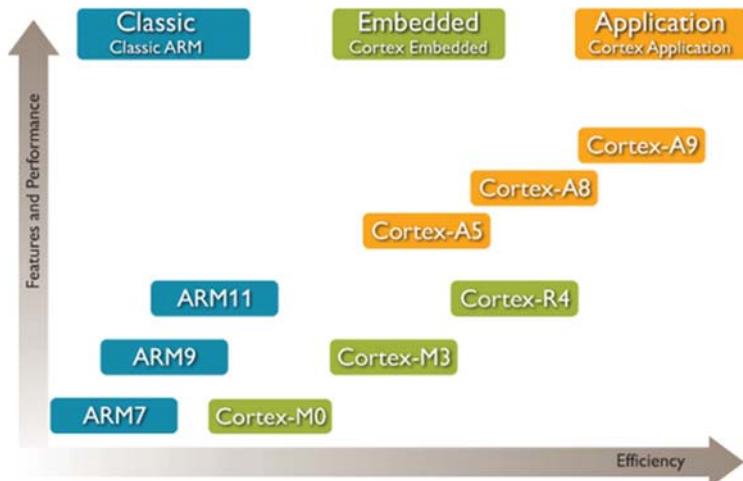
fc<sup>2</sup>

5

## Familias de procesadores ARM



- Procesadores con la misma arquitectura (compatibilidad binaria), pero distinta implementación
- Principales familias
  - ARM1, ARM2, ARM6, ARM7, ARM7TDMI, ARM9, ARM9TDMI, ARM9E, ARM10, ARM11
- Otras familias:
  - StrongARM,
  - XScale,
  - Cortex



fc<sup>2</sup>

6



# Los procesadores ARM

- Unos de los más vendidos/empleados en el mundo
  - 75% del mercado de procesadores empotrados de 32-bits
- Usados especialmente en dispositivos portátiles debido a su bajo consumo y razonable rendimiento (MIPS/Watt)
- Disponibles como hard/soft core
  - Fácil integración en Systems-On-Chip (SoC)
- Ofrecen diversas extensiones
  - Thumb (código compacto)
  - Jazelle (implementación HW de Java VM). No disponible en la versión que usaremos.

fc<sup>2</sup>

7



## Definiciones básicas

- El funcionamiento de un computador está determinado por las instrucciones que ejecuta.
- **LENGUAJE ENSAMBLADOR:** Conjunto de instrucciones, símbolos y reglas sintácticas y semánticas con el que se puede programar un ordenador para que resuelva un problema, realice una tarea/algoritmo, etc.
- El área que estudia las características de ese conjunto de instrucciones se denomina arquitectura del procesador o arquitectura del repertorio de instrucciones y engloba los siguientes aspectos:

fc<sup>2</sup>

8



# Definiciones básicas

- Repertorio de instrucciones
  - Operaciones que se pueden realizar y sobre qué tipos de datos actúan
  - Formato de instrucción
    - Descripción de las diferentes configuraciones de bits que adoptan las instrucciones máquina
- Registros de la arquitectura
  - Conjunto de registros visibles al programador (de datos, direcciones, estado, PC)
- Modos de direccionamiento
  - Forma de especificar la ubicación de los datos dentro de la instrucción y modos para acceder a ellos
- Formato de los datos
  - Tipos de datos que puede manipular el computador

fc<sup>2</sup>

9

# Lenguaje Ensamblador



- Analogía con una lengua:

Verbo	↔	Instrucción
Verbos del diccionario	↔	Repertorio de instrucciones
Lenguaje completo	↔	Lenguaje ensamblador
- Podemos decir que el código ensamblador es un conjunto de *expresiones* fácilmente recordables por el programador, en las que además se tiene en cuenta la arquitectura del procesador:
  - No se puede utilizar cualquier expresión
  - Hay que considerar donde se encuentran físicamente los datos

fc<sup>2</sup>

10



# Lenguaje Ensamblador ARM

- Cada línea del programa puede tener los campos:

Etiqueta	Instrucción/Directiva	Operandos	Comentarios
----------	-----------------------	-----------	-------------

- Etiqueta:** Referencias simbólicas de posiciones de memoria (texto + datos)
- Directiva:** acciones auxiliares durante el ensamblado (reserva de memoria)
- Instrucción:** del repertorio del ARM
- Operandos:**
  - Registros
  - Constantes: Decimales positivos y negativos, o hexadecimal (0x)
  - Etiquetas
- Comentarios:** caracteres seguidos de @. Pueden aparecer solos en una línea.

```
.global start
.equ ONE, 0x01      @Constant
.data               @Data
MYVAR: .word 0x02   @Variable
.bss
RES:   .space 4     @Program
.text
start: MOV R0, #ONE
       LDR R1, =MYVAR
       LDR R2, [R1]
       ADD R3, R0, R2
       LDR R4, =RES
       STR R3, [R4]
END:   B .
.end
```

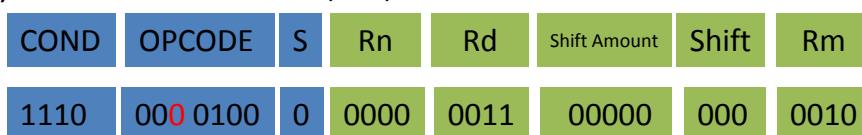
fc<sup>2</sup>



## Assembly vs. machine language

- ARM machine instructions are in fact richer than the previous simple example. Consider for instance:

- 1) Instruction: ADD R3, R0, R2



Where:

**Rd:** Destination register; **Rn and Rm:** source registers

**Shift field (XYZ)**

XY indicates what kind of shifting (lsl, lsr, asr, ror) is applied

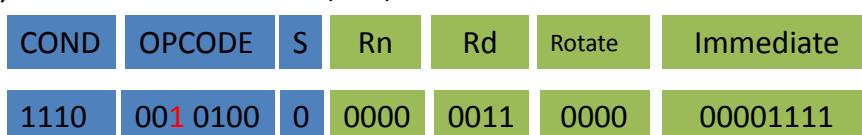
Z indicates whether the shifting is immediate or register

**Shift Amount field:**

If Z=0, indicates the number of bits to be shifted

If Z=1, the 4 leftmost bits indicate which register specifies the number of bits to be shifted and the rightmost bit is a zero

- 2) Instruction: ADD R3, R0, #15

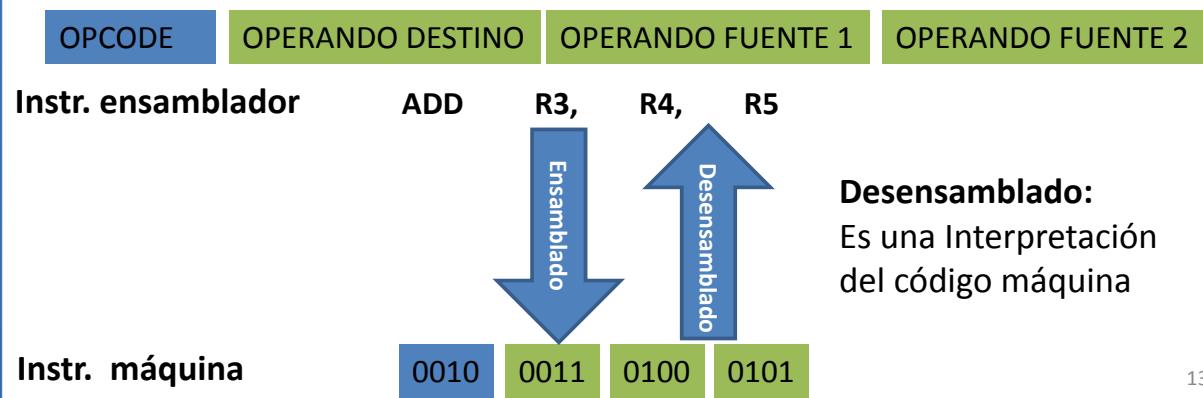


fc<sup>2</sup>



# Lenguaje Ensamblador

- Un computador **NO** entiende código ensamblador.
  - Sólo entiende ceros y unos (código máquina)
- Cada instrucción escrita en código ensamblador y cada etiqueta, son traducidos mediante el ensamblado y el enlazado en código máquina:



13

## Codificación máquina



- Las operaciones del ARM realmente son más potentes y complejas:

1) Instruction: ADD R3, R0, R2

COND	OPCODE	S	Rn	Rd	Shift Amount	Shift	Rm
1110	000 0100	0	0000	0011	00000	000	0010

Where:

**Rd:** Destination register; **Rn and Rm:** source registers

**Shift field (XYZ)**

XY indicates what kind of shifting (lsl, lsr, asr, ror) is applied

Z indicates whether the shifting is immediate or register

**Shift Amount** field:

If Z=0, indicates the number of bits to be shifted

If Z=1, the 4 leftmost bits indicate which register specifies the number of bits to be shifted and the rightmost bit is a zero

2) Instruction: ADD R3, R0, #15

COND	OPCODE	S	Rn	Rd	Rotate	Immediate
1110	001 0100	0	0000	0011	0000	00001111

14



# Tipos 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

## Ejemplo



- Ejemplo:  
Traducir a ensamblador de ARM la sentencia en C:

$$f = (g+h)*(i+j)$$



*Suponemos que inicialmente g, h, i, j están en los registros r1, r2, r3, r4 respectivamente.*

add r5,r2,r1

add r6,r3,r4

mul r7,r5,r6



# Tareas del procesador al ejec. inst.

- Pensemos un poco más a fondo qué tareas tiene que realizar el procesador cuando ejecuta una instrucción:
  1. Detectar tipo de instrucción a ejecutar → P.ej. ADD
  2. Leer de *algún lugar* los ops. fuente
  3. Realizar la suma de los dos operandos con algún HW
  4. Guardar el resultado en *algún lugar*
- ¿Dónde estarán los operandos? (datos)
  - TODOS los datos e instrucciones que manipula un programa se almacenan en la memoria
  - Temporalmente se pueden almacenar datos en los registros de la CPU (banco de registros)
  - Eventualmente pueden solicitarse datos a los dispositivos de E/S



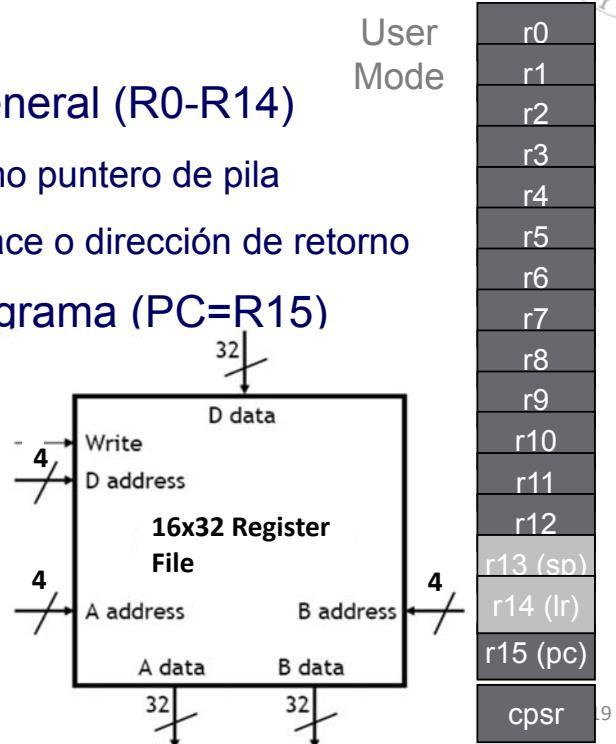
# Anatomía de una instrucción...

- ¿Dónde puede estar los operandos/resultado de una instrucción?
  - Memoria
  - Registros
  - En la propia instrucción!!! (*inmediato*)
- ¿Cuántos operandos explícitos tiene cada instrucción?
  - 0
  - 1
  - 2 (1 fuente/destino y 1 fuente)
  - 3 (2 fuente y un destino)
  - Más de 3???

# Registros del ARM (visibles en modo usuario)



- 32-bits de longitud
- 15 registros de propósito general (R0-R14)
  - R13=SP suele usarse como puntero de pila
  - R14=LR se usa como enlace o dirección de retorno
- Un registro contador de programa (PC=R15)
- Un registro de estado actual del programa (CPSR)
- Permite acceder a la vez a:
  - 2 registros fuente
  - 1 registro destino



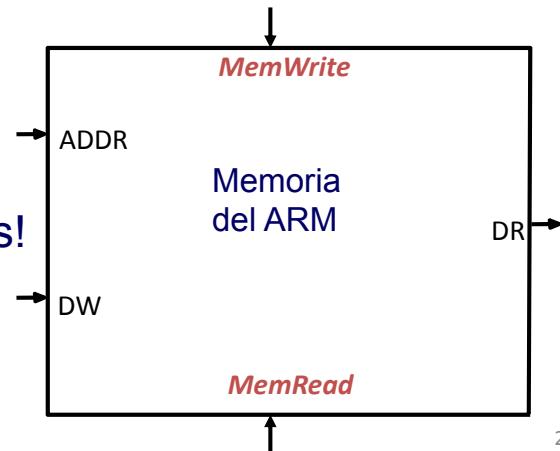
fc<sup>2</sup>

## Necesidad de la Memoria



- Lenguaje de programación: maneja estructuras de datos complejas (por ejemplo una matriz de datos).
- Necesitamos una estructura de almacenamiento de datos mucho mayor que el Banco de Registros → La MEMORIA

- Pueden contener muchos más datos que los registros
- ¡Una sola línea de direcciones!



20

fc<sup>2</sup>



# Necesidad de la Memoria

- ¿Por qué no hacer simplemente más grande el Banco de Registros, de forma que albergue todos los datos?

Cuanto más pequeña sea la estructura, más rápido funcionará.

## IDEA:

### – DATOS:

Tener todos datos del programa en Memoria

Traer al Banco de Registros sólo aquellos con los que se va a operar. Cuando se termine de operar con ellos → Guardar el/los resultado/s a Memoria.

### – INSTRUCCIONES: Programa Almacenado en Memoria

Tener el programa almacenado en memoria

Traer a un registro la instrucción que se está ejecutando



# Ejemplo

- Queremos contar el número de ceros que contiene un array de 1000 elementos → Sin memoria ... imposible.
- Alto nivel:

```
ceros=0;
for(i=0;i<1000;i++){
    if(A[i]==0){
        ceros=ceros+1;
    }
}
```
- ¿Qué debe hacer el computador en cada iteración del bucle?



# Organización de la memoria

- La memoria es una secuencia de bytes (*tabla*)
- Cada **byte** tiene asignada una **dirección** de memoria (número de entrada en la tabla)
  - Es decir, es *direccional* a nivel de byte
- Si disponemos de  $k$  bits para expresar una dirección
  - Podremos acceder a  $2^k$  bytes diferentes
  - Si  $k=16 \rightarrow 2^{16}$  bytes → 64Kbytes direccionables

# Organización de la memoria



- En las primeras prácticas no accederemos a nivel de byte:
  - Una instrucción ocupa 4 bytes
  - Un dato (por ejemplo, un entero) ocupa 4 bytes
  - Siempre accederemos a **direcciones múltiplo de 4**
- En muchos sistemas reales existen restricciones a la dirección de comienzo de un dato
  - Restricciones de **alineamiento**



# Organización de la memoria: alineamiento

- Restricción de la dirección de comienzo de un dato en **función de su tamaño**
  - Instrucción: 4 bytes → Sólo puede comenzar en direcciones múltiplos de 4
    - Misma restricción para datos de 4 bytes (int, float)
  - Dato de tamaño 2 bytes (short int) -> Sólo puede comenzar en direcciones múltiples de 2
  - Dato de tamaño byte (char) -> Puede comenzar en cualquier dirección

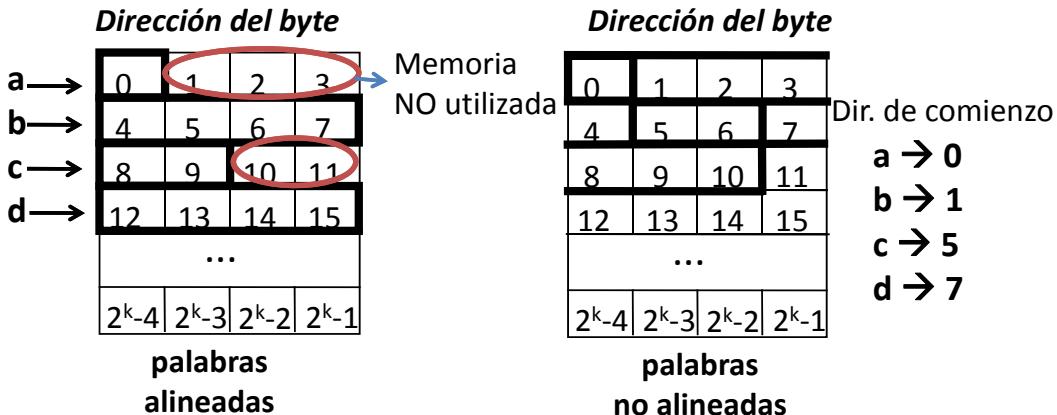
fc<sup>2</sup>

25

# Organización de la memoria: alineamiento



- Ejemplo. Declaramos cuatro variables:
  - char a; int b; short int c; int d;



- Las restricciones de alineamiento desaprovechan memoria...:
  - Pero **aceleran** notablemente el acceso
  - La mayoría de las arquitecturas OBLIGAN a realizar accesos alineados

fc<sup>2</sup>

26



# Organización de la memoria: palabras

- Una palabra es un conjunto de bytes
  - Durante el curso, una palabra será 32 bits (4 bytes)
- ARM → Accesos alineados

Nuestros accesos siempre serán a nivel de palabra

  - Una instrucción es una palabra
  - Un dato es una palabra

fc<sup>2</sup>

27



# Organización de la memoria: *endianness*

- ¿Cómo se organizan los bytes dentro de la palabra?
- Sea la palabra (4 bytes), codificada en hexadecimal, AABBCDD<sub>16</sub>
  - Dirección de comienzo: 16 (múltiplo de 4)
  - ¿Qué hay en memoria a partir de la posición 16?

Dirección	Contenido
16	AA
17	BB
18	CC
19	DD

**BIG-ENDIAN:** byte MÁS significativo primero

Dirección	Contenido
16	DD
17	CC
18	BB
19	AA

**LITTLE-ENDIAN:** byte MENOS significativo primero

fc<sup>2</sup>

28



# Visualización del contenido de memoria

- Es habitual (en simuladores, entornos gráficos) visualizar la memoria a nivel de palabra
  - Ejemplo: palabras  $AABBCCDD_{16}$  y  $9070FFAA_{16}$  a partir de la dirección 16.

Dirección	+0	+1	+2	+3
16	AA	BB	CC	DD
20	90	70	FF	AA

**BIG-ENDIAN:** lectura de *izquierda a derecha*

Dirección	+0	+1	+2	+3
16	DD	CC	BB	AA
20	AA	FF	70	90

**LITTLE-ENDIAN:** lectura de *derecha a izquierda*

- ARM admite ambas organizaciones



# Modos de direccionamiento

- ¿Cómo acceder a esos operandos?
  - **Modos de direccionamiento:** Formas que tiene la arquitectura para especificar dónde encontrar los datos/instrucciones que necesita
  - Cada arquitectura ofrece distintas posibilidades:
    - Inmediato
    - Absoluto
    - Directo de Registro
    - Indirecto de Registro
    - ...



# Inmediato

- El operando está contenido en la propia instrucción:

Instrucción:



*operando = A*

MOV R0, #0

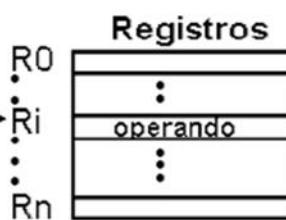
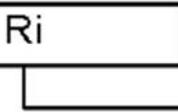
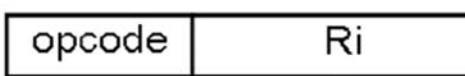
- Sirve para manejar constantes.



# Directo registro

- El operando está contenido en un registro del procesador:

Instrucción:



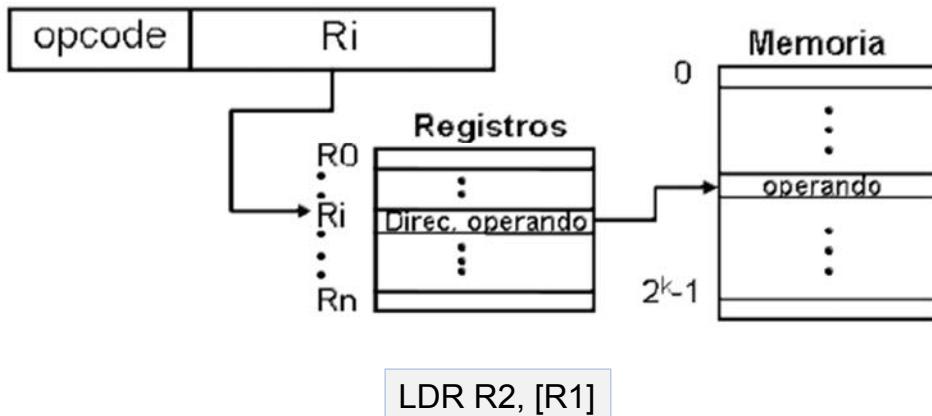
SUB R2, R2, R3



# Indirecto registro

- El operando está en memoria y la dirección de memoria donde éste se encuentra está almacenada en un registro:

Instrucción:



fc<sup>2</sup>

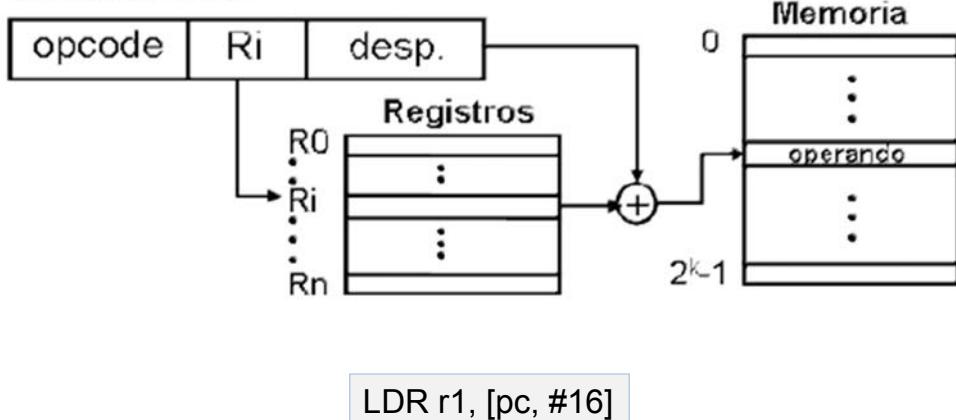
33



# Indirecto registro con desplazamiento

- El operando está en memoria
- Dir. de acceso = registro + desplazamiento

Instrucción:



fc<sup>2</sup>

34



# Instrucciones aritmético-lógicas

- Operaciones aritméticas y lógicas → Muy comunes en cualquier programa de alto nivel
- Todo computador debe ser capaz de realizar las operaciones aritméticas y lógicas básicas:
  - Aritméticas: SUMA, RESTA, ETC.
  - Lógicas: AND, OR, ETC.
- Equivalencia directa con lenguaje de alto nivel:

L. ALTO NIVEL

$C = A + B$

L. ENSAMBLADOR

$\leftarrow\rightarrow$  Instrucción de suma (add ...)

$C = A * B$

$\leftarrow\rightarrow$  Instrucción de multiplicación (mul ...)

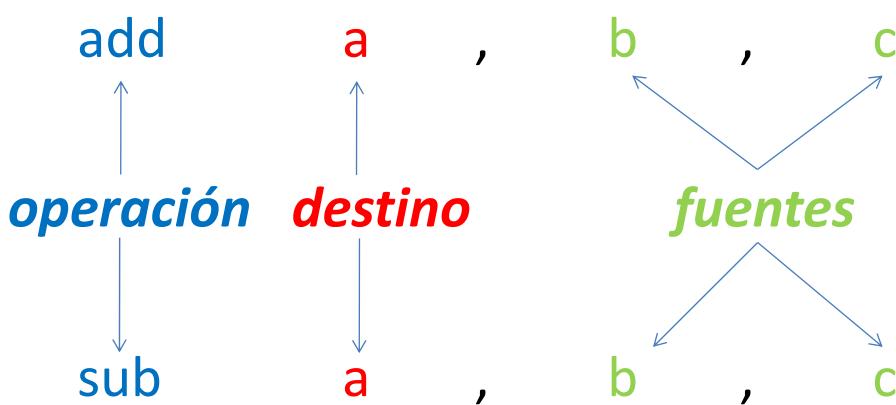
$C = A \&& B$

$\leftarrow\rightarrow$  Instrucción AND (and ...)



# Formato ins. aritmético-lógicas

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





# Instrucciones aritmético-lógicas

- Las operaciones aritméticas y lógicas que vamos a utilizar más frecuentemente son:

– SUMA →	<b>add Rd, Rn, &lt;Operando&gt;</b>
– RESTA →	<b>sub Rd, Rn, &lt;Operando&gt;</b>
– MULTIPLICACIÓN →	<b>mul Rd, Rm, Rs</b>
– AND →	<b>and Rd, Rn, &lt;Operando&gt;</b>
– OR →	<b>orr Rd, Rn, &lt;Operando&gt;</b>
– XOR →	<b>eor Rd, Rn, &lt;Operando&gt;</b>
– MOVIMIENTO →	<b>mov Rd, &lt;Operando&gt;</b>
– DESPLAZAMIENTO →	<b>lsl Rd, Rm, &lt;Operando&gt;</b> <b>lsr Rd, Rm, &lt;Operando&gt;</b>

donde **<Operando>** de forma simplificada es:

- un **registro** (empleamos *direcciónamiento directo registro*)
- un **inmediato** (empleamos *direcciónamiento inmediato*).

# Instrucciones aritmético-lógicas



- Ejemplos:

- **add r1,r3,r4** → suma el contenido de los registros r3 y r4 y almacena el resultado en el registro r1. A todos los datos se accede con *direcciónamiento directo registro*.
- **and r2, r5, #2** → realiza la and lógica del contenido de r5 con 2 y almacena el resultado en el registro r2. Al segundo operando fuente se accede con *direcciónamiento inmediato*.



# Instrucciones aritmético-lógicas

## ■ Ejemplo

- Operaciones *lógicas* (and, orr, eor, etc.) funcionan a nivel de *bit*

Registro	Contenido
r1	0x000000FA
r2	0x0000F132
r3	0x00000000

→ and r3,r1,r2

Registro	Contenido
r1	0x000000FA
r2	0x0000F132
r3	<b>0x00000032</b>

# Instrucciones de transferencia de datos



- Instrucciones aritméticas y lógicas sólo pueden operar sobre registros o inmediatos.
- Los datos del programa están en memoria → Necesidad de transferir datos: **Banco Registros ↔ Memoria**
- Equivalencia inexistente con lenguajes alto nivel: en estos se trabaja con variables, que no nos preocupa dónde están. En cambio, en ensamblador, los datos están en memoria, pero para operar con ellos hay que traerlos al banco de registros.

L. ALTO NIVEL      L. ENSAMBLADOR (A, B y C en memoria)

$C = A + B \rightarrow$  [Instrucción mover dato A de Memoria a B.R.  
Instrucción mover dato B de Memoria a B.R.  
Instrucción de suma (add ...)  
Instrucción mover resultado de B.R. a Memoria]



# Instrucción de load

- Mueve un dato de una posición de la Memoria a un registro del Banco de Registros:  
**Memoria → Banco Regs**
- Sintaxis:  
**ldr Rd, <Dirección>**  
La instrucción copia el dato que hay en la posición de memoria **<Dirección>** en el registro **Rd**
- Diversas formas para especificar esa dirección:  
**ldr Rd, [Rn]**  
La instrucción copia el dato que hay en la posición de memoria indicada en el registro **Rn** (*direcccionamiento indirecto registro*) al registro **Rd**.  
**ldr Rd, [Rn,#±Desplazamiento]**  
La instrucción copia el dato que hay en la posición de memoria indicada por **Rn + Desplazamiento** (*direcccionamiento indirecto registro con despl.*) a **Rd**.



# Instrucción de store

- Mueve un dato de un registro del Banco de Registros a una posición de la Memoria:  
**Banco Regs → Memoria**
- Sintaxis:  
**str Rd, <Dirección>**  
La instrucción copia el dato que hay en el registro **Rd** en la posición de memoria **<Dirección>**
- Algunas formas para especificar esa dirección:  
**str Rd, [Rn]**  
La instrucción copia el dato que hay en el registro **Rd** a la posición de memoria indicada en el registro **Rn** (*direcc. indirecto registro*)  
**str Rd, [Rn,#±Desplazamiento]**  
La instrucción copia el dato que hay en el registro **Rd** en la posición de memoria indicada por **Rn + Desplazamiento** (*direcc. indirecto registro con despl.*)



## Ejemplos load

Dirección de Memoria	Contenido
0x00000100	0xAABBCCDD
0x00000104	0x11223344
0x00000108	0x00FF55EE

Registro	Contenido
r1	0x00000100
r2	0x0000F132
r3	0x00000000

Registro	Contenido
r1	0x00000100
r2	0x0000F132
r3	0xAABBCCDD

*ldr r3,[r1]*  
*ldr r3,[r1,#8]*

## Ejemplos store

Dirección de Memoria	Contenido
0x00000100	0xAABBCCDD
0x00000104	0x11223344
0x00000108	0x00FF55EE

Registro	Contenido
r1	0x00000100
r2	0x0000F132
r3	0x12345678

Dirección de Memoria	Contenido
0x00000100	0x12345678
0x00000104	0x11223344
0x00000108	0x00FF55EE

*str r3,[r1]  
i str r3,[r1,#8]*

Dirección de Memoria	Contenido
0x00000100	0xAABBCCDD
0x00000104	0x11223344
0x00000108	0x12345678

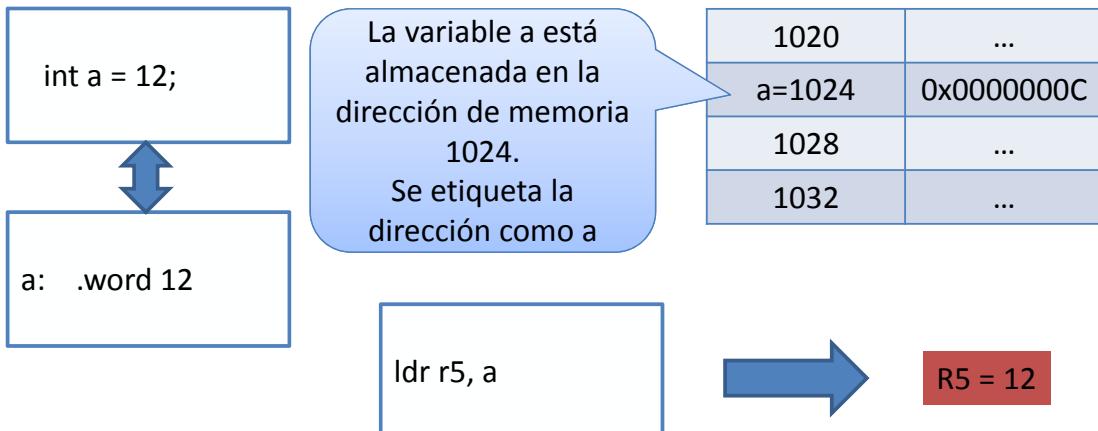


# Como se accede a una variable

## – Direccionamiento absoluto:

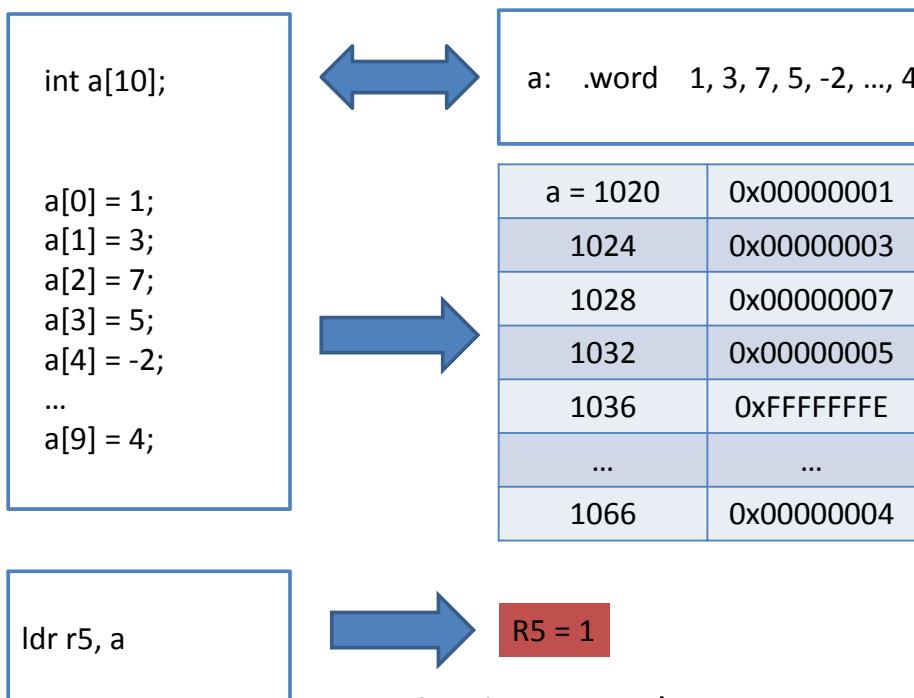
- El operando está en la dirección de memoria indicada.

## – Seudo-instrucción: No la proporciona el rep. de instr. ARM



fc<sup>2</sup>

# Como acceder a un array



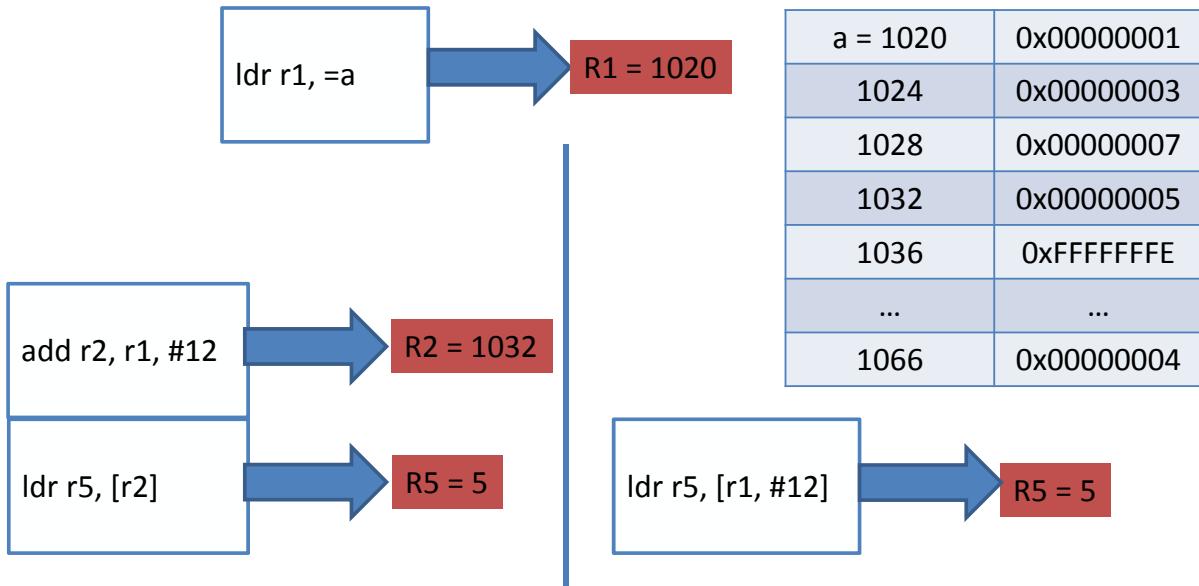
¿Y si quiero acceder a otra componente?

fc<sup>2</sup>



# Como acceder a un array

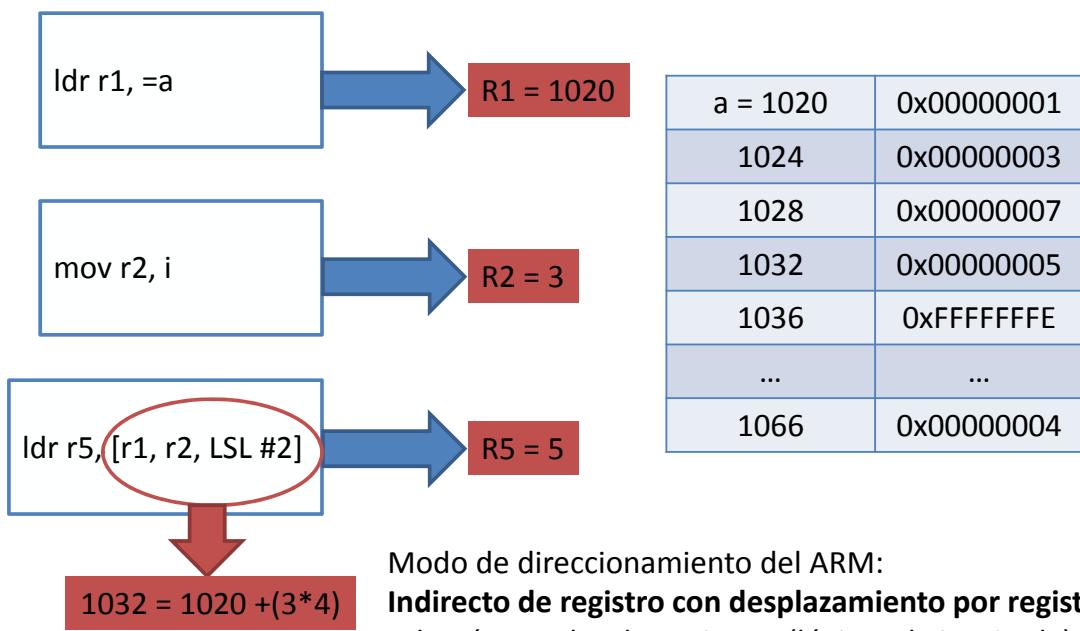
¿Dónde está la componente 3? **1032**



fc<sup>2</sup>

# Como acceder a un array

¿Dónde está la componente i? **1020 + 4\*i**



fc<sup>2</sup>



# Instrucciones para toma de decisiones

- Diferencia calculadora – computador: El computador puede tomar decisiones → Instrucciones para toma de decisiones
- Pueden romper el flujo normal del programa
  - Flujo normal → Ejecución secuencial
  - Instrucción de salto → Después de ésta instrucción, no se ejecuta la siguiente, sino una situada en otro lugar del código  
→ Se **SALTA** a otro lugar del programa
- Equivalencia con lenguaje de alto nivel:

L. ALTO NIVEL	L. ENSAMBLADOR
Condición: if(A==B) then	↔ Instrucción de salto
Bucle: for(...)	↔ Combinación de insts. salto



# Instrucción de salto

- Formato:  
**b Desplazamiento**  
En lugar de ejecutar la siguiente instrucción al salto en el orden secuencial, se ejecuta aquella que está en la posición resultante de saltar un número de **bytes** igual al *Desplazamiento* (puede ser +, en cuyo caso se salta hacia adelante, o negativo, en cuyo caso se salta hacia atrás)
- Ejemplo: Inicializar a 0 las componentes de un vector (**bucle for**)

```
V:      Componentes del vector
...
    mov    r2, #0
    ldr    r1, =V
    str    r2, [r1]
    add    r1, r1, #4
    b     .-8      @Saltar 8 bytes hacia atrás
```

Problema: Nunca salimos del bucle → Interesa poder saltar o no en función de una condición



# Instrucciones de salto condicional

- Formato:

**cmp Rn, <Operando>**

**bXX Desplazamiento**

Dependiendo de cuál sea el resultado de la condición XX evaluada sobre Rn y Operando, se ejecuta tras el salto la siguiente instrucción en el orden secuencial o bien la situada en la posición resultante de saltar un número de instrucciones igual al *Desplazamiento*

- Ejemplo: Inicializar a 0 las componentes de un vector de 10 componentes

V: Componentes del vector

...

```
mov r2, #0  
mov r3, #9  
ldr r1, =V  
cmp r3, #0  
beq .+20    @Saltar 20 bytes  
           @hacia adelante  
str r2, [r1]  
add r1, r1, #4  
sub r3, r3, #1  
b  .-20    @Saltar 20 bytes  
           @hacia atrás  
str r2, [r1]
```

51

fc<sup>2</sup>



# Condiciones

SUFIJO	DESCRIPCIÓN DE CONDICIÓN	FLAGS
<b>EQ</b>	Igual	Z=1
<b>NE</b>	No igual	Z=0
<b>CS/HS</b>	Sin signo, mayor o igual	C=1
<b>CC/LO</b>	Sin signo y menor	C=0
<b>MI</b>	Menor	N=1
<b>PL</b>	Positivo o cero (Zero)	N=0
<b>VS</b>	Desbordamiento (Overflow)	V=1
<b>VC</b>	Sin desbordamiento (No overflow)	V=0
<b>HI</b>	Sin signo, mayor	C=1 & Z=0
<b>LS</b>	Sin signo, menor o igual	C=0 or Z=1
<b>GE</b>	Mayor o igual	N=V
<b>LT</b>	Menor que	N!=V
<b>GT</b>	Mayor que	Z=0 & N=V
<b>LE</b>	Menor que o igual	Z=1 or N!=V
<b>AL</b>	Siempre	

52

fc<sup>2</sup>



# Uso de pseudo-instrucciones de salto

- Al igual que con los load/store, podemos emplear pseudo-instrucciones.
  - Por ejemplo:

**b Etiqueta**

En lugar de ejecutar la siguiente instrucción al salto en el orden secuencial se ejecuta la situada en la dirección asociada a la *Etiqueta*

**cmp r1,r2**

**beq Etiqueta**

Si r1 es igual a r2, se ejecuta tras el salto la instrucción situada en la dirección asociada a la *Etiqueta*.

Si r1 es distinto a r2, se ejecuta tras el salto la instrucción situada a continuación de éste.

## Ejemplo construcción if-then-else



- 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 Maylgu
add r1, r1, r2
b Salir
Maylgu: sub r1, r1, r2
Salir:
```



# Ejemplo construcción while

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

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

array: Componentes del vector

...

```
    mov r2, #0
    ldr r3, =array
LOOP:   cmp r2, #10
        bhs Salir
        ldr r4, [r3]
        add r4, r4, r2
        str r4, [r3]
        add r2, r2, #1
        add r3, r3, #4
        b LOOP
```

Salir:

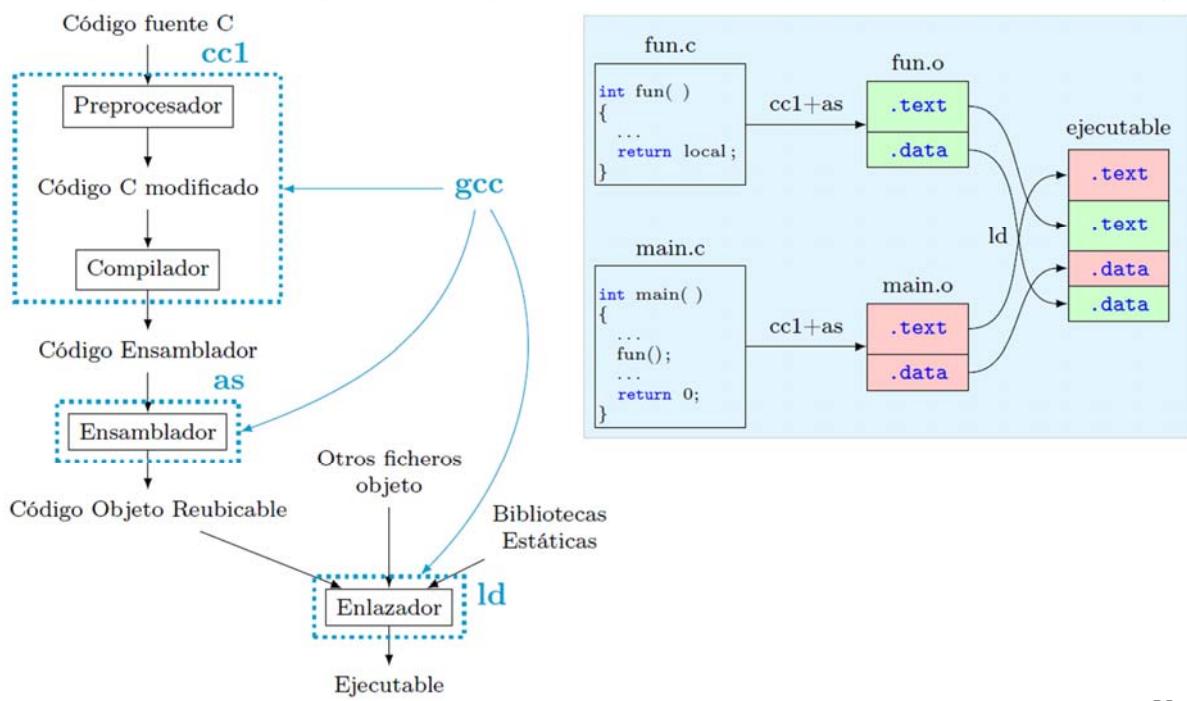
55

fc<sup>2</sup>

# Desarrollo de código ARM



- Etapas de compilación, ensamblado y enlazado



fc<sup>2</sup>



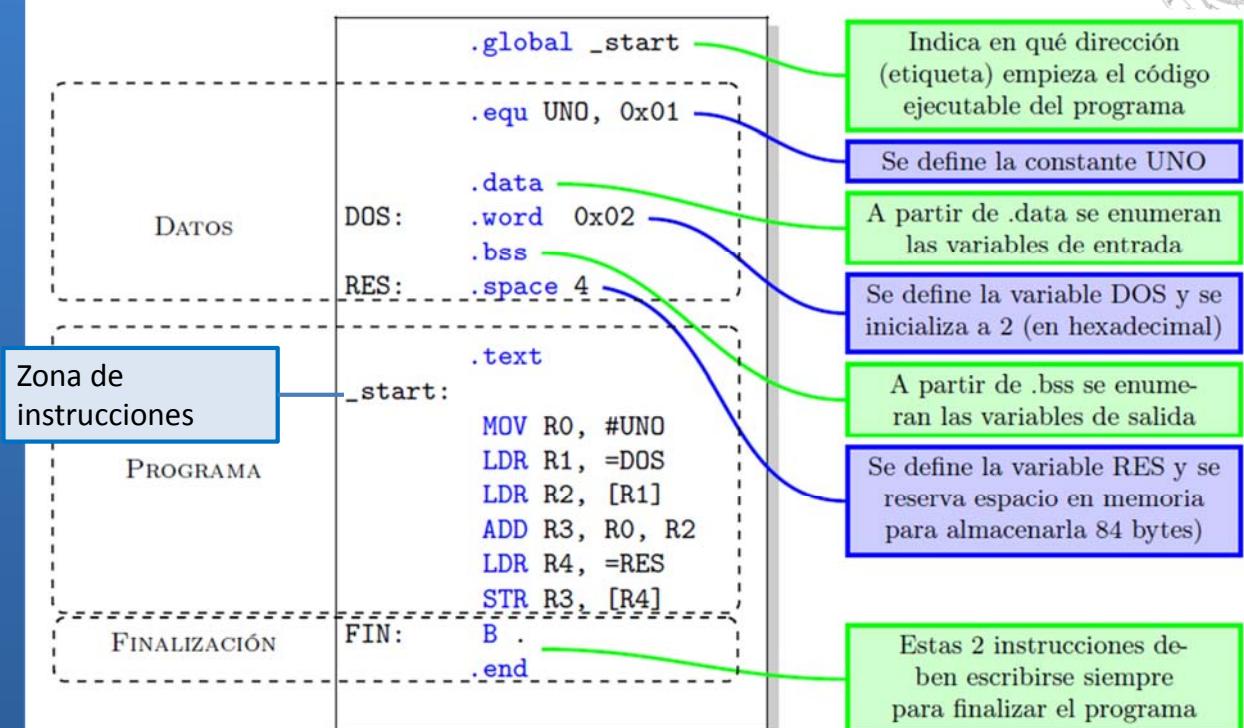
# Desarrollo de código ARM

- Directivas para desenrrollar el código en memoria

Directiva	Propósito
.text	Declara el comienzo de la sección de texto (instrucciones)
.data	Declara el comienzo de la sección de variables globales <b>con</b> valor inicial
.bss	Declara el comienzo de la sección de variables globales <b>sin</b> valor inicial
.word w1, ..., wn	Reserva <i>n</i> <b>palabras</b> en memoria e inicializa el contenido a <i>w1,...,wn</i>
.space n	Reserva <i>n</i> <b>bytes</b> de memoria
.equ nom, valor	Define una constante llamada <i>nom</i> como <i>valor</i>
.global	Exporta un símbolo para el enlaza (por ejemplo, comienzo del programa)

fc<sup>2</sup>

## Zonas de un programa ensamblador

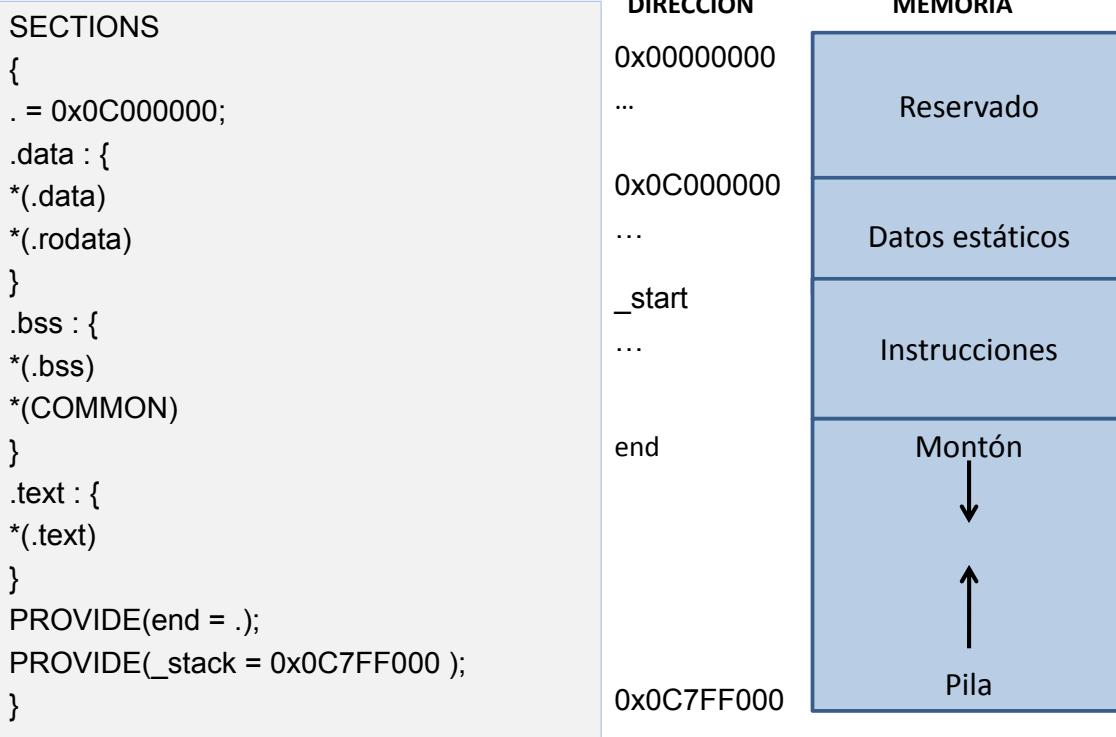


fc<sup>2</sup>



# Script de enlazado

- En él se definen las zonas de memoria donde ubicar...



59



# Uso de pseudo-instrucciones de ld y st

- El lenguaje ensamblador nos da la posibilidad de emplear pseudo-instrucciones que nos facilitan mucho la programación con etiquetas.

– Por ejemplo:

**ldr Rd, Etiqueta**

La instrucción copia el **dato** que hay en la posición de memoria asociada a *Etiqueta* al registro *Rd*.

**str Rd, Etiqueta**

La instrucción copia el dato que hay en el registro *Rd* a la posición de memoria asociada a *Etiqueta*.

**ldr Rd, =Etiqueta**

La instrucción copia la **dirección** de memoria asociada a *Etiqueta* al registro *Rd*.



# Ejemplo

- ¿Cómo nos ayudan las etiquetas y las pseudo-instrucciones?
- Por ejemplo a manejar variables.
  - Ejemplo: Inicializar a 0 las componentes de un vector de tres componentes.

Aspecto del programa en Lenguaje Ensamblador

```
V: Componentes del vector.  
...  
    mov r2, #0  
    ldr r1, =V  
    str r2, [r1]  
    add r1, r1, #4  
    str r2, [r1]  
    add r1, r1, #4  
    str r2, [r1]
```

Traducción a L. Máquina

El programa funciona igual sea cual sea la dirección X

Evitamos calcular el desplazamiento del "ldr"

Aspecto la memoria

Direc. Memoria	Contenido *
X	Desconocido
X+4	Desconocido
X+8	Desconocido
X+12	mov r2, #0
X+16	ldr r1, [pc, #24]
x+20	str r2, [r1]
x+24	add r1,r1,#4
x+28	str r2, [r1]
x+32	add r1,r1,#4
X+36	str r2, [r1]
X+40	X

Vector  
Programa

\* En realidad está en binario

61

fc<sup>2</sup>

# Llamadas a función (subrutinas)



- Grupo de instrucciones con un objetivo particular, que está separado del código principal y que se invoca desde éste.
  - Permite reutilizar código
  - Hace más comprensible el programa.

```
void foo (int a, int b) {  
    ....  
}  
  
int main() {  
    int y;  
    y = fuu(3);  
    foo(y,4);  
    ...  
}
```

```
int fuu (int a) {  
    int x;  
    ...  
    return x;  
}
```

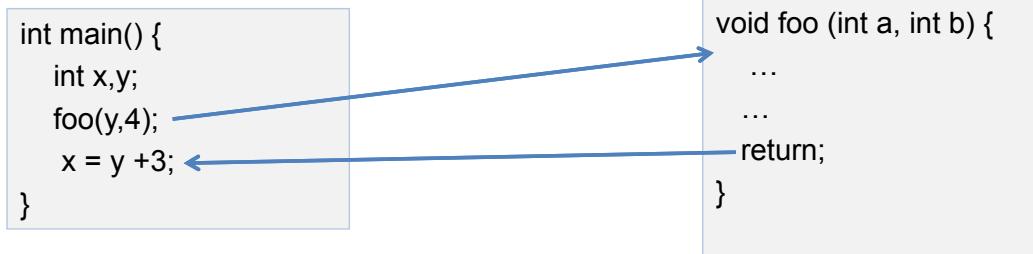
fc<sup>2</sup>

62



# Llamadas a función: invocación

- ¿Cómo invocar una función?

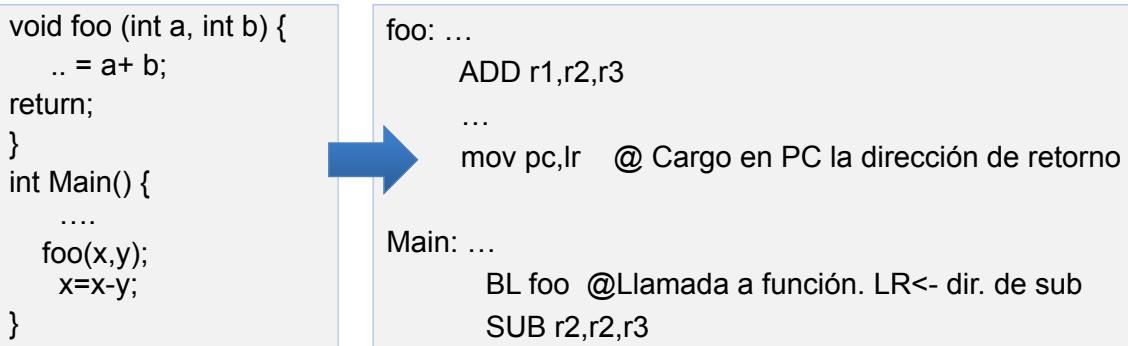


- Salto incondicional al comienzo de la función *foo*
- ¡¡ Pero necesitamos *recordar* la dirección a la que hay que volver tras ejecutar la función !!



# Llamadas a función: instrucciones ARM

- Instrucción Branch and Link: **BL <etiqueta>**
  - Salto incondicional a <etiqueta>
  - Almacena en el registro **LR** la dirección de la siguiente instrucción
- Volvemos de la función reestableciendo el PC
  - Podemos usar la instrucción **mov pc,lr**





# Llamadas a función: argumentos

- ¿Cómo comunicar argumentos a una función?

```
int main() {  
    int x,y;
```

```
    x=fuu(3);  
    y=fuu(7);  
    foo(3,x);  
    foo(y,4);
```

```
}
```

Llamadas a las mismas  
funciones con  
diferentes argumentos

```
int fuu (int a) {
```

```
    int x;
```

```
    ...
```

```
    return x;
```

```
}
```

```
void foo(int a, int b) {
```

```
    ...
```

```
    return;
```

```
}
```

- ¿Cómo sabe la función *fuu* dónde escribir el valor final de la variable *x*?

fc<sup>2</sup>

65

# Llamadas a función: argumentos



- Idea sencilla: usar registros
  - ARM sigue el estándar AAPCS
    - Usar los registros **r0-r3** para pasar los cuatro primeros argumentos de la función
    - El valor de retorno se devuelve por **r0**

```
int x;  
int Main() {  
    x=fee(3,4);  
}  
int fee (int a, int b) {  
    return a+b;  
}
```

Main: ...

MOV r0,#3 @Primer argumento en r0

MOV r1,#4 @Segundo argumento en r1

BL fee

STR r0, x @ El resultado estará en r0

...

fee: ADD r0,r0,r1 @ Escribe el valor de retorno en r0  
MOV pc,lr

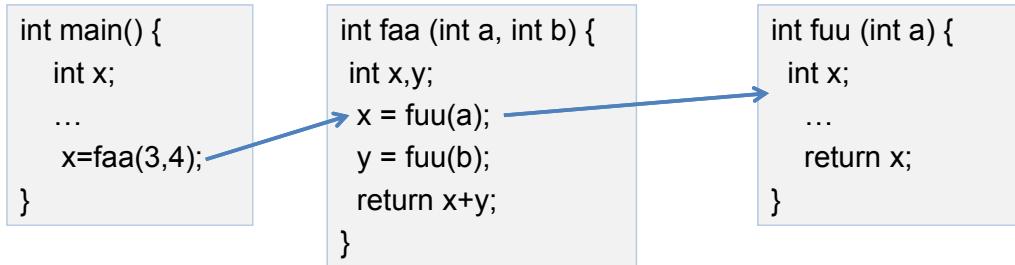
fc<sup>2</sup>

66



# Complicando las llamadas a función.

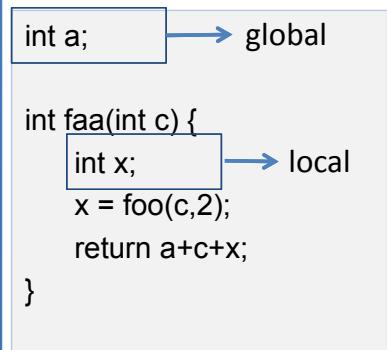
- ¿Y si hay más de cuatro argumentos?
- ¿Y si hay llamadas anidadas?



- Al llamar a *faa* se usan los registros r0 y r1
- ¿Qué hacemos para llamar a *fuu*?



# ¿Qué pasa con las variables locales?

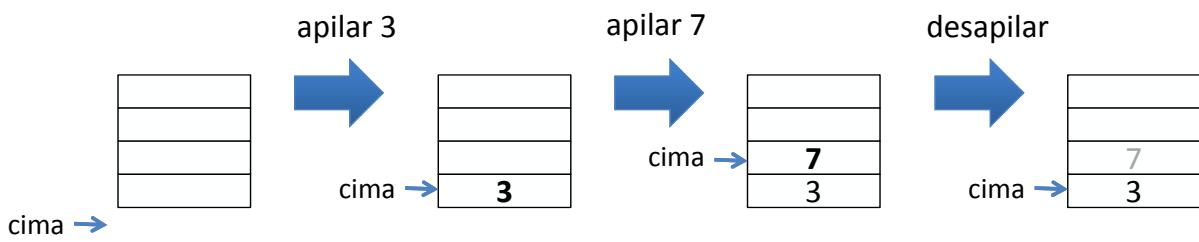


- Variables globales
  - *Vivas* durante la ejecución de todo el programa
  - Tienen una posición fija en memoria
- Variables locales
  - Sólo están *vivas* mientras estemos ejecutando la función que las declaró
  - ¿Dónde se almacenan?



# Llamadas a función: la pila

- Una estructura tipo "pila" (*last in – first out*) es idónea para resolver estos problemas
  - La *pila* es una zona de memoria reservada para esta tarea. NO es una memoria físicamente separada.
  - Cada hilo en ejecución debe tener su propia pila (pues tendrá su propio árbol de llamadas a funciones)



fc<sup>2</sup>

69

# Llamadas a función: la pila

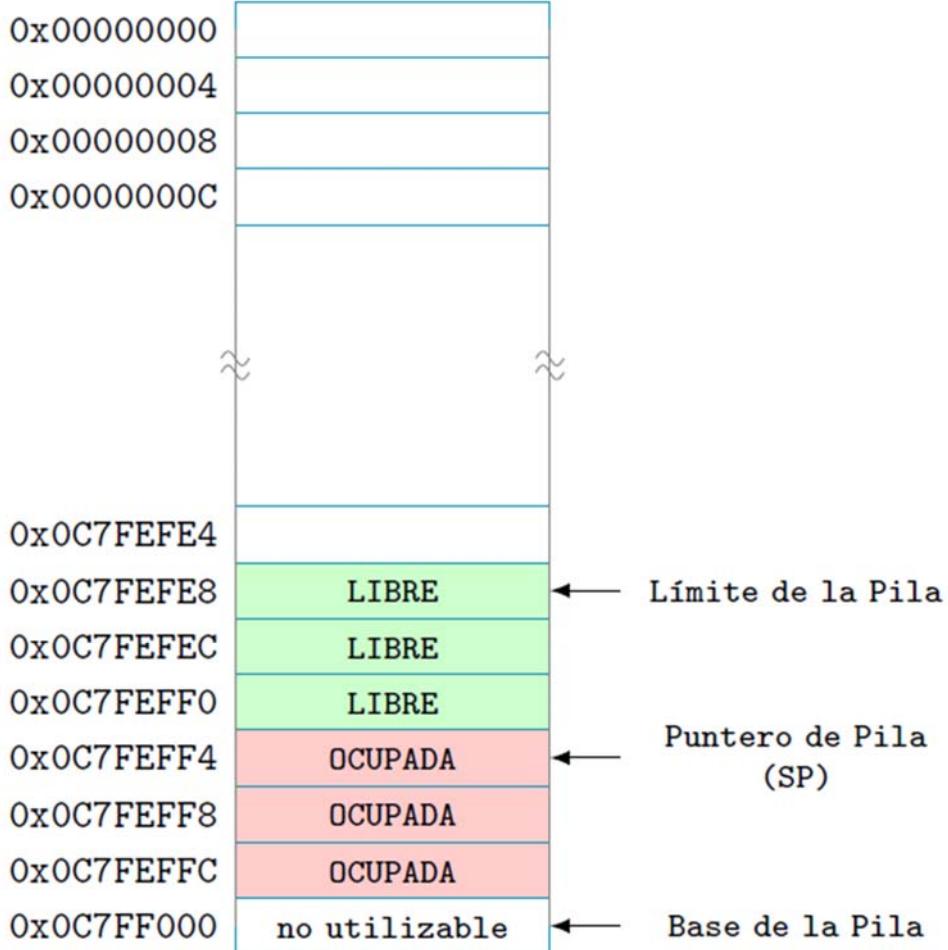


- La pila en ARM: el registro *SP*
  - El registro R13 (*SP -> stack pointer*) contiene la dirección de la cima de la pila (última posición ocupada)
  - “Full Descending”: La pila “crece” de direcciones superiores a direcciones inferiores de memoria
  - En el script de enlazado inicializamos la dirección de la pila.

```
PROVIDE(_stack = 0x0C7FF000 );
```
- ¿Qué se almacena en la pila?
  - Argumentos de entrada: del quinto en adelante
  - Espacio para albergar las variables locales
  - Registros cuyo contenido se debe preservar

fc<sup>2</sup>

70



## Pasos ejecución subrutina



1. Situar parámetros en lugar accesible a la función llamada (subrutina)
  - En Registros y en pila si es necesario
2. Transferir el control a la subrutina (*BL*)
3. Reservar espacio para la ejecución
  - Var. locales y registros que se deseen preservar
4. Ejecutar las tareas propias de la subrutina
5. Situar resultado en lugar accesible a la función llamante (*r0*)
6. Devolver el control al punto de llamada

Función llamante

Función llamada



# Salvado de registros

- Durante la ejecución de la subrutina se puede hacer uso de cualquier registro disponible.
- Suele ser necesario que la *función llamante* no pierda los datos que tenía en esos registros.
- El ARM Architecture Procedure Call Standard (AAPCS) regula las llamadas a subrutinas en la arquitectura ARM:
  - La *función llamada* DEBE preservar:
    - los registros *r4-r10*,
    - el registro *r11 (FP)*
    - el registro *r13 (SP)* .
  - Si se modifican durante la llamada, deberán apilarse al principio y desapilarse al final.
  - Se pueden usar los registros *r0-r3* con total libertad
    - La *función llamante* NO debe asumir que conservan su valor tras la llamada...

fc<sup>2</sup>

73



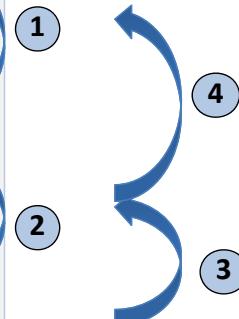
# Salvado de registros: llamadas anidadas

- Siguiendo el AAPCSS, ¿qué ocurre con el registro LR?

```
00h Main: ...
04h      BL fun1
08h ...
0Ch

10h fun1: ADD r0,r0,r1
14h      BL fun2
18h      MOV pc,lr

1Ch fun2: SUB r0,r0,#3
20h      MOV pc,lr
```



1. Llamada a fun1
  - LR ← 08h. PC ← 10h
2. Llamada a fun2
  - PC ← 1Ch. LR ← 18h
3. Salida de fun2
  - PC ← 18h
4. Salida de fun1
  - PC ← 18h

- Sólo se debe preservar si no es una rutina hoja
  - Rutina hoja es aquella que no llama a otras subrutinas.

fc<sup>2</sup>

74



# Marco de pila activo

- Zona de la pila que *pertenece* a la función en ejecución
  - Importante porque determina el ámbito de las variables locales
  - Puede estar acotado únicamente por el SP (sabiendo el tamaño del marco)
  - Es habitual contar con un segundo registro para acotar la base inferior del marco, FP =R11 (frame pointer).

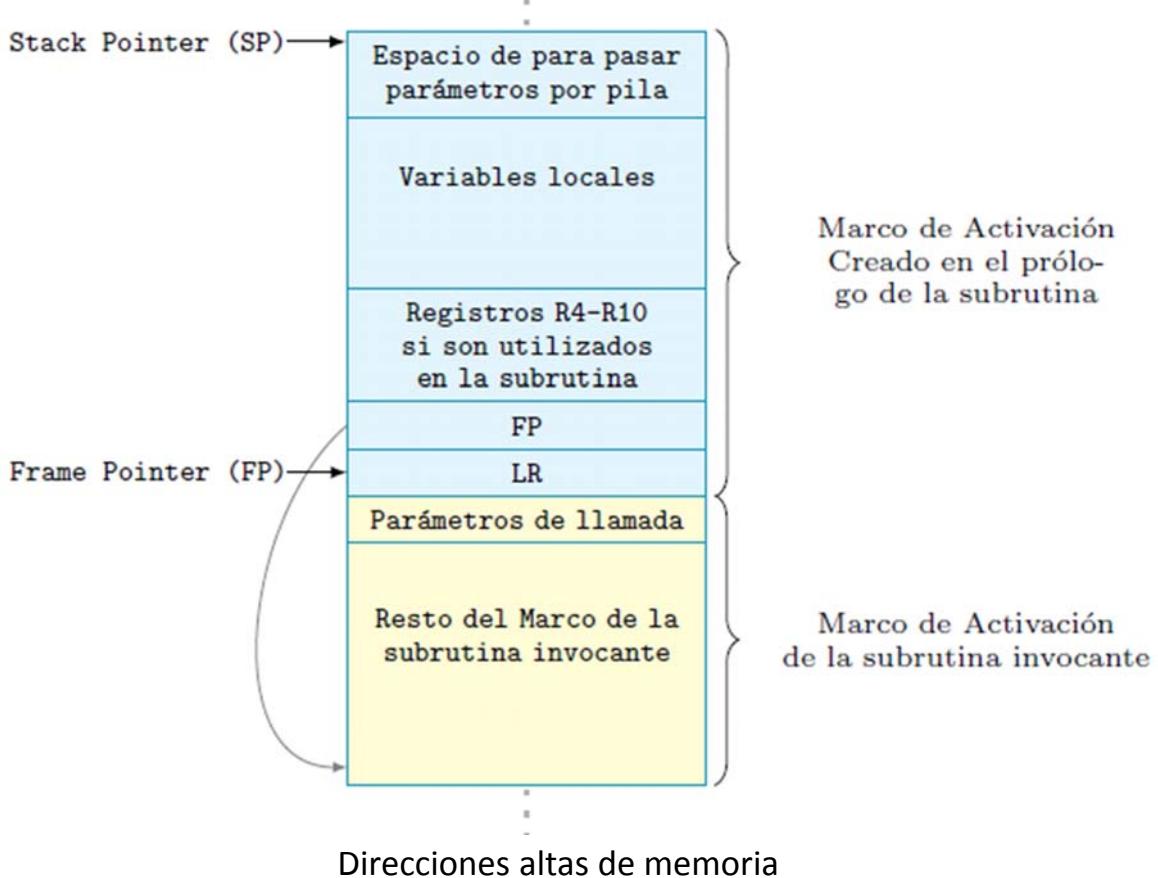
# Estructura de una rutina



- Estructura de una rutina:
  - **Código de entrada (prólogo):**  
*Construye el marco*
  - **Cuerpo de la rutina**  
*Implementa la funcionalidad*
  - **Código de salida (epílogo)**  
*Destruye el marco*  
y  
*hace el retorno*



## Direcciones bajas de memoria



fc<sup>2</sup>

77



# Variables locales y globales

- Variables globales
  - Almacenadas en secciones .data o .bss
  - Persisten en memoria durante todo el programa
- Variables locales
  - Almacenadas en el marco de pila de la rutina
  - Activas sólo en el cuerpo de la rutina

fc<sup>2</sup>

78



# Marco de pila: prólogo y epílogo

- Ejemplo rutina que utiliza r2,r3,r4 y r5

## Prólogo

```

SUB      SP, SP, #16 @ Actualizar SP para apilar contexto
STR     R4, [SP,#0]
STR     R5, [SP,#4]
STR     FP, [SP,#8] @ Apilamos valor actual de FP (opcional)
STR     LR, [SP,#12] @ Apilamos LR (si rutina no es hoja)
ADD     FP, SP, #12 @ Set the new value of FP
SUB     SP, SP, #SpaceForLocalVariables @ Reserva espacio
    
```

## Cuerpo

código de la rutina (hace uso de r2, r3, r4 y r5)  
Potencialmente, hay llamadas a otras rutinas

## Epílogo

```

ADD   SP, SP, #SpaceForLocalVariables @Preparo SP para restaurar
LDR   LR, [SP,#12] @ Restauro LR
LDR   FP, [SP,#8] @ Restauro FP
LDR   R5, [SP,#4] @ Restauro R5
LDR   R4, [SP,#0] @ Restauro R4
ADD   SP, SP, #16 @ Dejamos SP a su valor original
MOV   PC, LR @ Vuelta a la función llamante
    
```

79

fc<sup>2</sup>

# Marco de Pila: prólogo y epólogo

- Ejemplo de subrutina, que modifica r2,r3,r4 y r5, usa 2 variables locales y no hace llamadas con >4 argumentos

### Pila antes/después llamada

Sup: SP=0xC7F EFF4; FP= 0xC7F EFFC  
LR=0xC00 2000; R4=3; R5=9

### Pila después del prologo

SP=0xC7F EFD8 FP= 0xC7F EFF0  
LR=0xC00 2000 R4=3 R5=9

Dirección	Contenido
0xC7F EFD8	
0xC7F EFDC	
0xC7F EFE0	
0xC7F EFE4	
0xC7F EFE8	
0xC7F EFEC	
0xC7F EFF0	
0xC7F EFF4	
0xC7F EFF8	
0xC7F EFFC	

SP →	Dirección	Contenido
	0xC7F EFD8	
	0xC7F EFDC	Space for Local_var 2
	0xC7F EFE0	Space for Local_var 1
	0xC7F EFE4	3
	0xC7F EFE8	9
	0xC7F EFEC	0xC7F EFFC
	0xC7F EFF0	0xC00 2000
	0xC7F EFF4	
	0xC7F EFF8	
	0xC7F EFFC	

FP →	Dirección	Contenido
	0xC7F EFD8	
	0xC7F EFDC	Space for Local_var 2
	0xC7F EFE0	Space for Local_var 1
	0xC7F EFE4	3
	0xC7F EFE8	9
	0xC7F EFEC	0xC7F EFFC
	0xC7F EFF0	0xC00 2000
	0xC7F EFF4	
	0xC7F EFF8	
	0xC7F EFFC	

80

fc<sup>2</sup>



# Ejercicio

```
int x=3,y=-7,z;
int main() {
    z=abs(x)+abs(y);
    return 0;
}

int abs(int a) {
    int r;
    if (a<0)
        r=opuesto(a);
    else
        r=a;
    return r;
}
int opuesto(int a) {
    return -a;
}
```

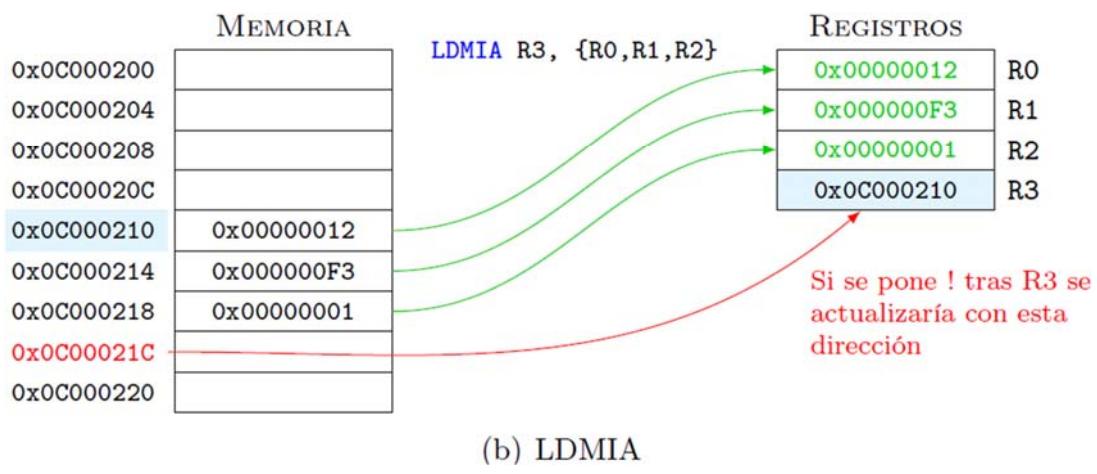
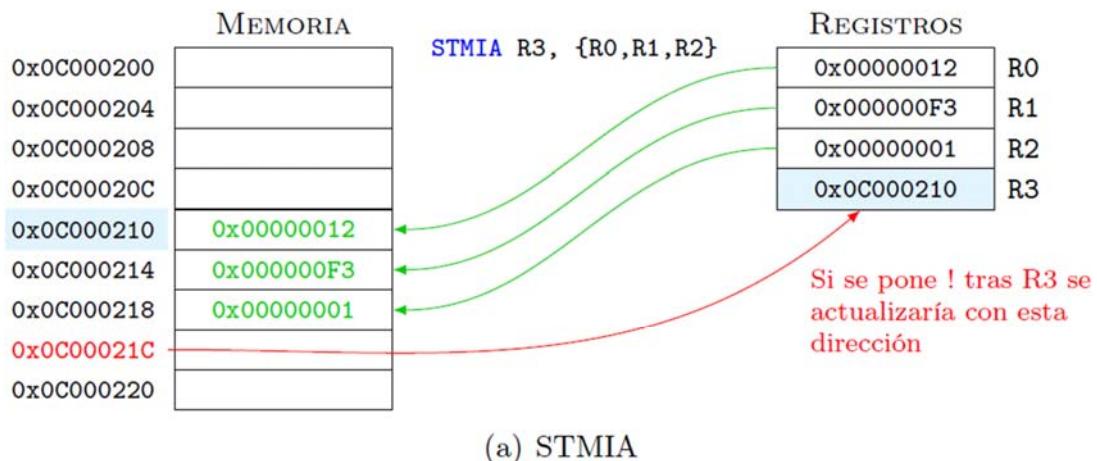
- Escribe el cuerpo de cada función
  - Sin prólogos ni epílogos
  - Los argumentos de entrada estarán en r0, r1....
- ¿Cómo evoluciona la pila?
  - ¿Qué debemos apilar antes de cada llamada?
  - ¿Qué debemos apilar al comienzo de cada función?
- Escribe el código ARM completo de cada función
  - Ya es posible determinar qué hay que incluir en prólogos y epílogos



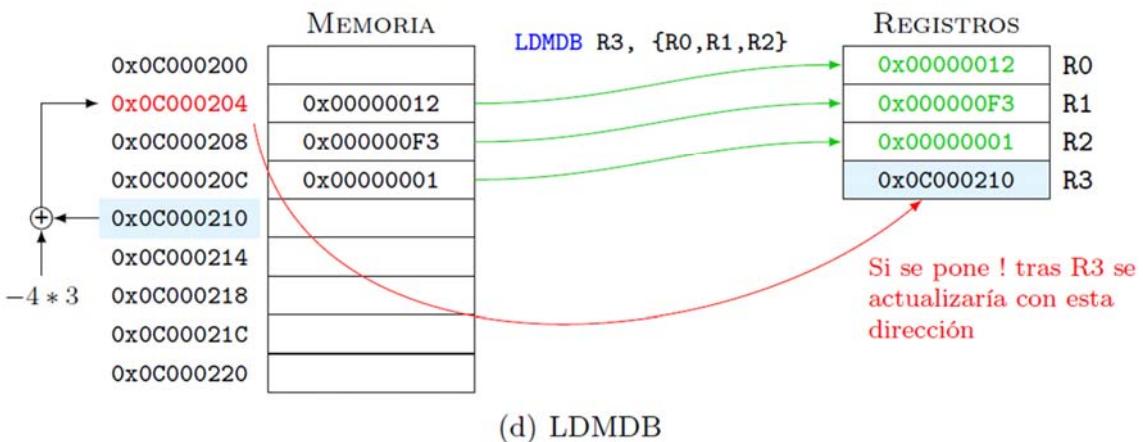
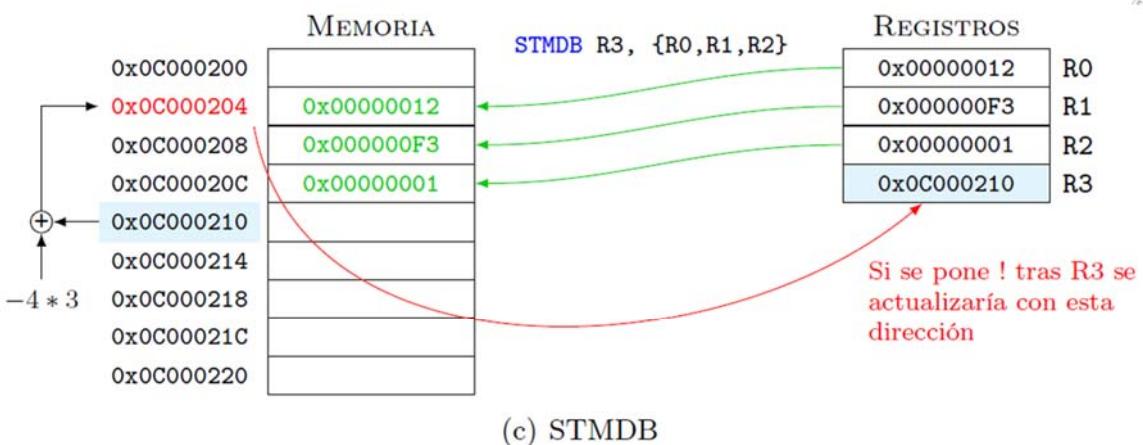
# Load y Store Múltiple

- La arquitectura ARM ofrece instrucciones de load y store múltiple.
- Permiten la carga/guarda simultánea de varios registros en posiciones de memoria consecutivas
- Donde:
  - **Rb** es el registro base que contiene una dirección de memoria por la que comienza la operación.
  - El signo ! tras el registro base es opcional, si se pone, el registro base quedará actualizado adecuadamente para encadenar varios accesos de este tipo.
  - **Modo** es el modo de direccionamiento, existen 4 modos: IA, IB, DA, DB.

```
LDM<Modo> Rb[!], {lista de registros}  
STM<Modo> Rb[!], {lista de registros}
```



83



84

fc<sup>2</sup>



# Load y Store Múltiple

- Para codificar el prólogo y el epílogo solo necesitamos dos: STMDB y LDMIA.
- El ensamblador del ARM proporciona alias:
  - Inserción:
    - STMDB SP!, {R4-R10,FP,LR}
    - PUSH {R4-R10,FP,LR}
    - STMDP SP!, {R4-R10,FP,LR} @FD->full descending.
  - Extracción:
    - LDMIA SP!, {R4-R10,FP,LR}
    - POP {R4-R10,FP,LR}
    - LDMFD SP!, {R4-R10,FP,LR} @FD->full descending



## Marco de pila: prólogo generalista

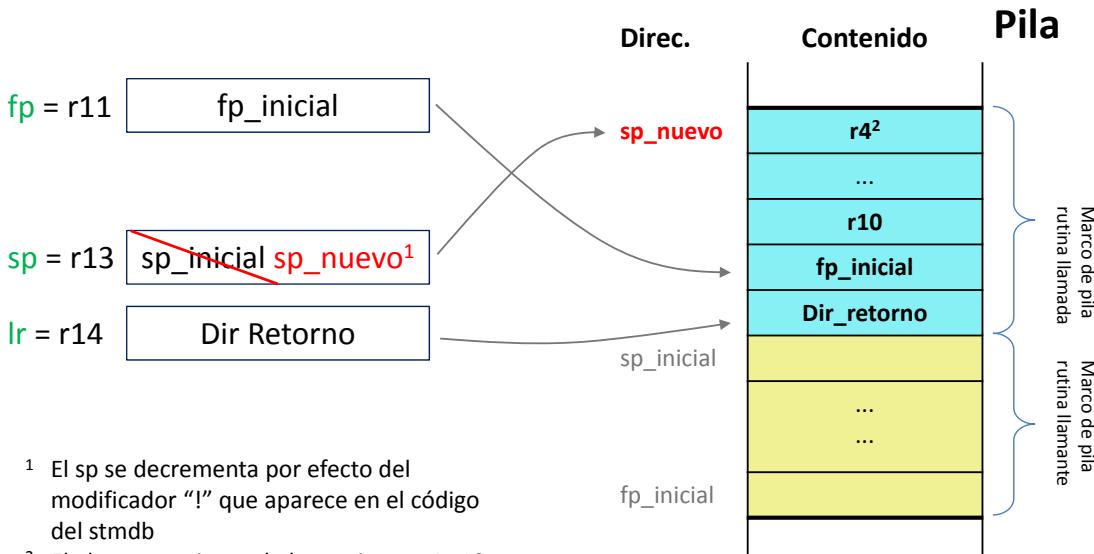
PUSH {R4-R10,FP,LR}	@ Copiar registros en la pila
ADD FP, SP, #(4*NumRegistrosApilados-4)	@ FP dirección base del marco
SUB SP, SP, #4*NumPalabrasExtra	@ Espacio extra necesario

- Existen algunos casos especiales en los que podemos simplificar este prólogo:
  - Si no pasamos más de cuatro parámetros a ninguna subrutina no tendremos que reservar espacio extra. Podemos entonces eliminar la tercera instrucción.
  - Si sólo apilamos un registro (sucedería en una subrutina hoja que no modifique ninguno de los registros r4-r10) la segunda instrucción se convertiría en ADD FP,SP, #0. En este caso quizás quede más claro codificarla como MOV FP, SP.



# El prólogo paso a paso (1)

- stmdb sp!, {r4-r10,fp,lr} ó PUSH {R4-R10,FP,LR}

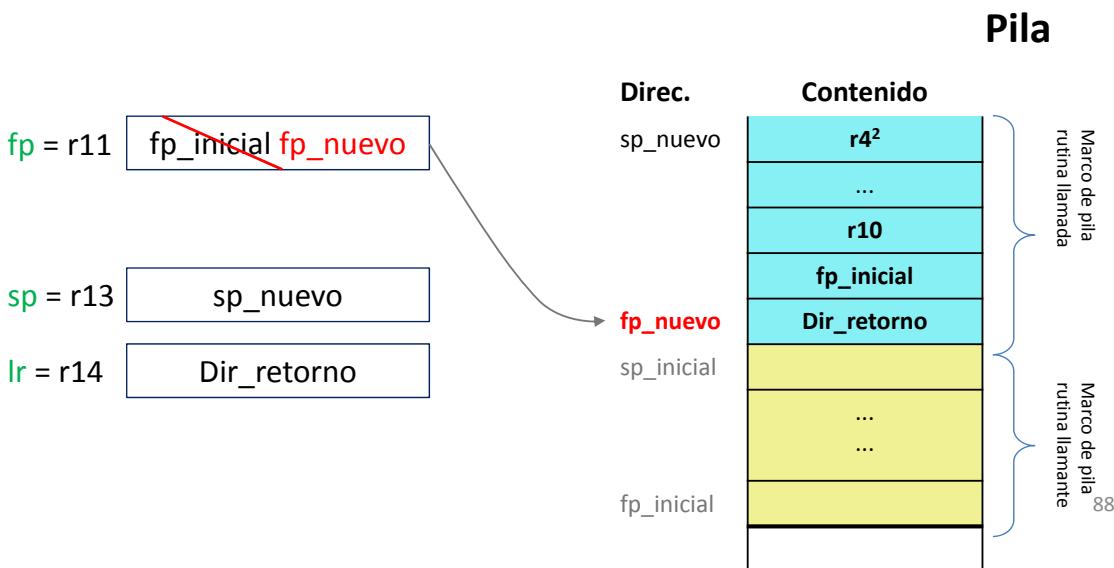


87

fc<sup>2</sup>

# El prólogo paso a paso (2)

- ADD FP, SP, #(4\*NumRegistrosApilados-4)



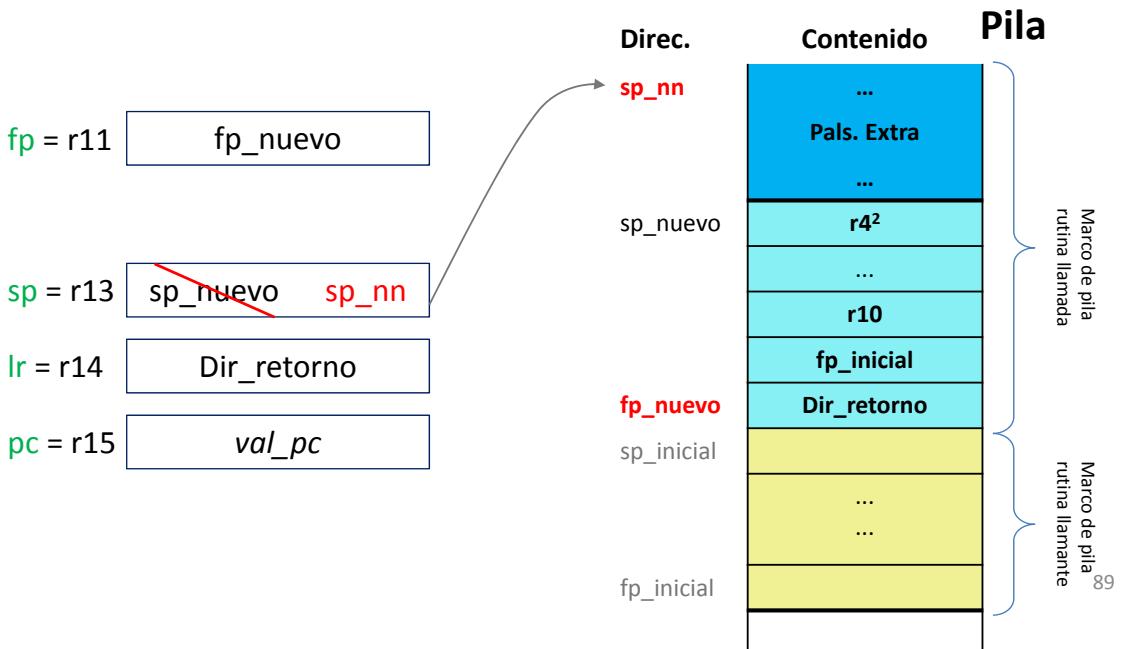
88

fc<sup>2</sup>



# El prólogo paso a paso (3)

## ■ SUB SP, SP, #4\*NumPalabrasExtra



fc<sup>2</sup>

## Marco de pila: epílogo generalista

SUB SP, FP, #(4\*NumRegistrosApilados-4) @ SP dirección del 1er registro apilado  
 @ Se descartan las variables locales y argumentos  
 POP {R4-R10,FPLR} @ Restaura los registros desde la pila  
 BX LR @ Retorno de subrutina



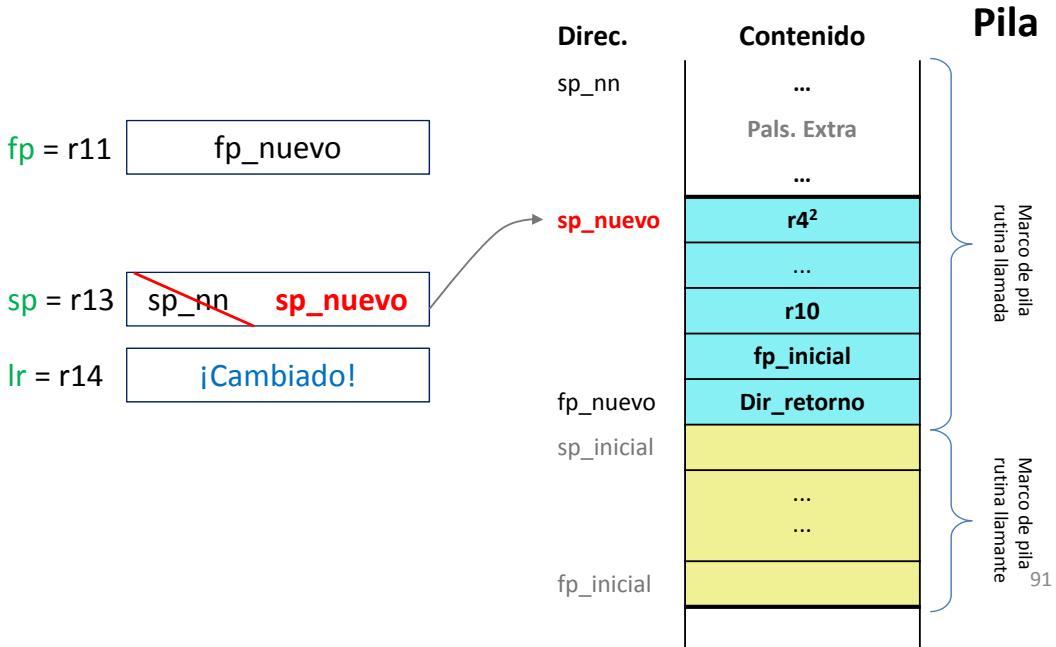
- Situaciones para simplificar este epílogo:
  - Si sólo apilamos un registro la primera instrucción del epílogo se reduce a SUB SP,FP, #0. En este caso quizá quede más claro codificarla como MOV SP, FP.
  - Si no hemos reservado espacio extra (NumPalabrasExtra=0) y no hemos alterado el valor de SP en el cuerpo de la subrutina, podemos incluso eliminar esta misma instrucción del epílogo, ya que SP tendrá el valor que le asignó la primera instrucción del prólogo.

fc<sup>2</sup>



# El epílogo paso (1)

- SUB SP, FP, #(4\*NumRegistrosApilados-4)

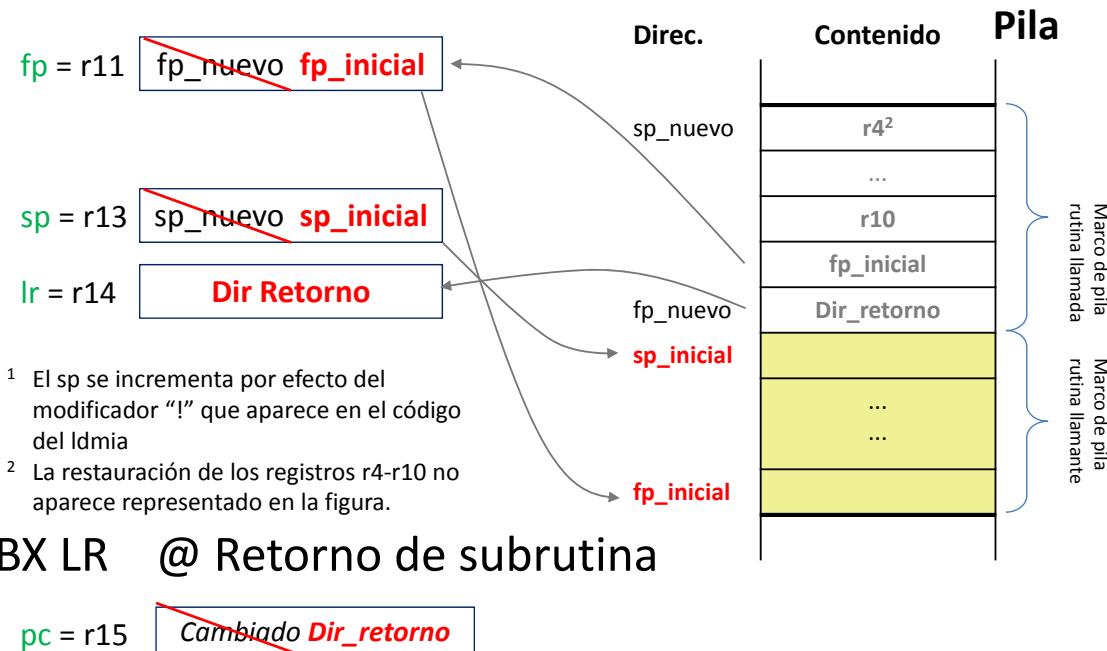


fc<sup>2</sup>



# El epílogo paso a paso (2 y 3)

- LDMIA SP!, {R4-R10,FP,LR} ó POP {R4-R10,FP,LR}



fc<sup>2</sup>

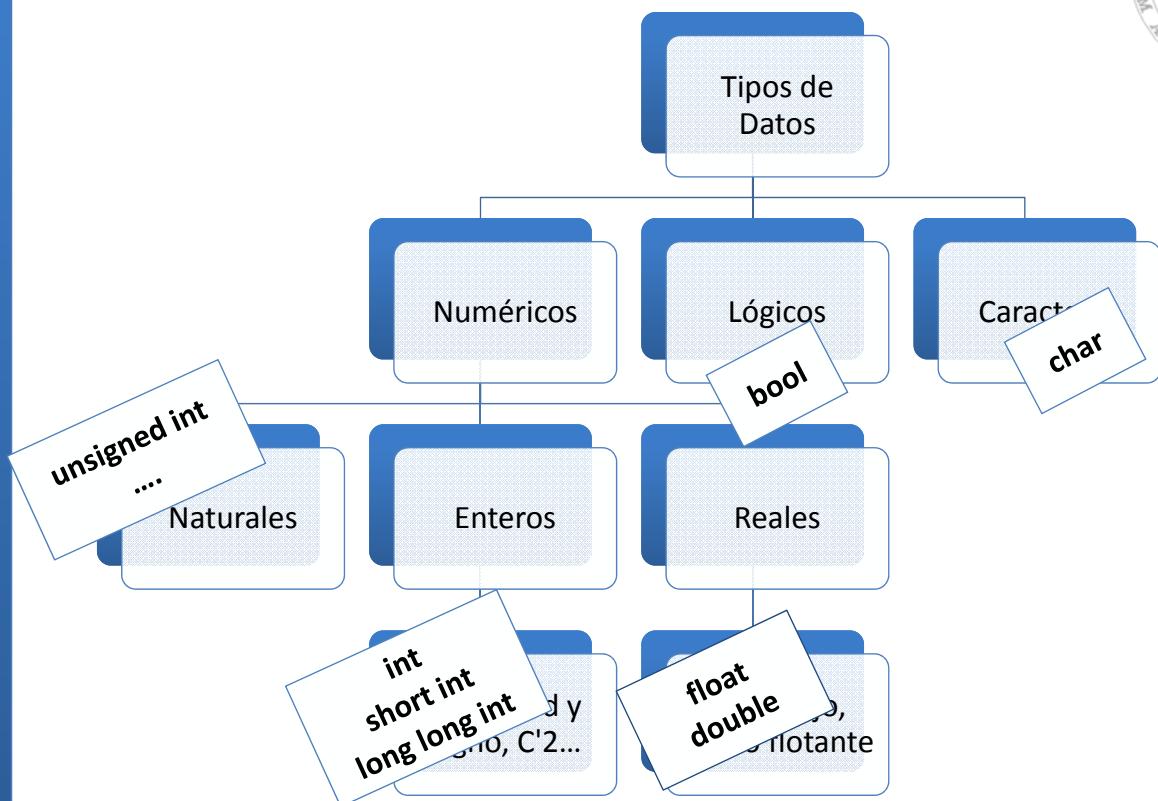


# Formato de datos

- En una posición de memoria encontramos la siguiente número 0x39383700. ¿Qué representa el dato leído?
  - El número entero (*int*) 95998548
  - La cadena de caracteres "987"
  - El número real (*float*) 0.0001756809
  - La instrucción *xor \$24,\$9,0x3700*



# Tipos de datos básicos





# Tipos de datos básicos

- Tamaño *habitual* de los datos básicos de lenguajes de alto nivel

Tipo de dato	Tamaño
char / unsigned char	1 byte
short int	2 bytes
int / unsigned int	4 bytes
float	4 bytes
double	8 bytes



# Codificación de caracteres

- ¿Cómo se representa el carácter 'a' en memoria?
- ¿Qué ocurre cuando se escribe la cadena "Hola Mundo" en una variable en C?
- Cada carácter tiene asociado una codificación binaria (generalmente de 8 bits)
  - ASCII (Extended ASCII).
  - UNICODE (UTF-8, UTF-16)
  - ISO 8859-1 (latin1), ISO 8859-15 (latin 9)

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0 000	000	<b>NUL</b> (null)	32	20 040	000	&#32;	<b>Space</b>	64	40 100	000	&#64;	<b>Ø</b>	96	60 140	000	&#96;	'
1	1 001	001	<b>SOH</b> (start of heading)	33	21 041	001	&#33;	!	65	41 101	001	&#65;	<b>A</b>	97	61 141	001	&#97;	<b>a</b>
2	2 002	002	<b>STX</b> (start of text)	34	22 042	002	&#34;	"	66	42 102	002	&#66;	<b>B</b>	98	62 142	002	&#98;	<b>b</b>
3	3 003	003	<b>ETX</b> (end of text)	35	23 043	003	&#35;	#	67	43 103	003	&#67;	<b>C</b>	99	63 143	003	&#99;	<b>c</b>
4	4 004	004	<b>EOT</b> (end of transmission)	36	24 044	004	&#36;	\$	68	44 104	004	&#68;	<b>D</b>	100	64 144	004	&#100;	<b>d</b>
5	5 005	005	<b>ENQ</b> (enquiry)	37	25 045	005	&#37;	%	69	45 105	005	&#69;	<b>E</b>	101	65 145	005	&#101;	<b>e</b>
6	6 006	006	<b>ACK</b> (acknowledge)	38	26 046	006	&#38;	&	70	46 106	006	&#70;	<b>F</b>	102	66 146	006	&#102;	<b>f</b>
7	7 007	007	<b>BEL</b> (bell)	39	27 047	007	&#39;	'	71	47 107	007	&#71;	<b>G</b>	103	67 147	007	&#103;	<b>g</b>
8	8 010	010	<b>BS</b> (backspace)	40	28 050	010	&#40;	(	72	48 110	010	&#72;	<b>H</b>	104	68 150	010	&#104;	<b>h</b>
9	9 011	011	<b>TAB</b> (horizontal tab)	41	29 051	011	&#41;	)	73	49 111	011	&#73;	<b>I</b>	105	69 151	011	&#105;	<b>i</b>
10	A 012	012	<b>LF</b> (NL line feed, new line)	42	2A 052	012	&#42;	*	74	4A 112	012	&#74;	<b>J</b>	106	6A 152	012	&#106;	<b>j</b>
11	B 013	013	<b>VT</b> (vertical tab)	43	2B 053	013	&#43;	+	75	4B 113	013	&#75;	<b>K</b>	107	6B 153	013	&#107;	<b>k</b>
12	C 014	014	<b>FF</b> (NP form feed, new page)	44	2C 054	014	&#44;	,	76	4C 114	014	&#76;	<b>L</b>	108	6C 154	014	&#108;	<b>l</b>
13	D 015	015	<b>CR</b> (carriage return)	45	2D 055	015	&#45;	-	77	4D 115	015	&#77;	<b>M</b>	109	6D 155	015	&#109;	<b>m</b>
14	E 016	016	<b>SO</b> (shift out)	46	2E 056	016	&#46;	.	78	4E 116	016	&#78;	<b>N</b>	110	6E 156	016	&#110;	<b>n</b>
15	F 017	017	<b>SI</b> (shift in)	47	2F 057	017	&#47;	/	79	4F 117	017	&#79;	<b>O</b>	111	6F 157	017	&#111;	<b>o</b>
16	10 020	020	<b>DLE</b> (data link escape)	48	30 060	020	&#48;	0	80	50 120	020	&#80;	<b>P</b>	112	70 160	020	&#112;	<b>p</b>
17	11 021	021	<b>DC1</b> (device control 1)	49	31 061	021	&#49;	1	81	51 121	021	&#81;	<b>Q</b>	113	71 161	021	&#113;	<b>q</b>
18	12 022	022	<b>DC2</b> (device control 2)	50	32 062	022	&#50;	2	82	52 122	022	&#82;	<b>R</b>	114	72 162	022	&#114;	<b>r</b>
19	13 023	023	<b>DC3</b> (device control 3)	51	33 063	023	&#51;	3	83	53 123	023	&#83;	<b>S</b>	115	73 163	023	&#115;	<b>s</b>
20	14 024	024	<b>DC4</b> (device control 4)	52	34 064	024	&#52;	4	84	54 124	024	&#84;	<b>T</b>	116	74 164	024	&#116;	<b>t</b>
21	15 025	025	<b>NAK</b> (negative acknowledge)	53	35 065	025	&#53;	5	85	55 125	025	&#85;	<b>U</b>	117	75 165	025	&#117;	<b>u</b>
22	16 026	026	<b>SYN</b> (synchronous idle)	54	36 066	026	&#54;	6	86	56 126	026	&#86;	<b>V</b>	118	76 166	026	&#118;	<b>v</b>
23	17 027	027	<b>ETB</b> (end of trans. block)	55	37 067	027	&#55;	7	87	57 127	027	&#87;	<b>W</b>	119	77 167	027	&#119;	<b>w</b>
24	18 030	030	<b>CAN</b> (cancel)	56	38 070	030	&#56;	8	88	58 130	030	&#88;	<b>X</b>	120	78 170	030	&#120;	<b>x</b>
25	19 031	031	<b>EM</b> (end of medium)	57	39 071	031	&#57;	9	89	59 131	031	&#89;	<b>Y</b>	121	79 171	031	&#121;	<b>y</b>
26	1A 032	032	<b>SUB</b> (substitute)	58	3A 072	032	&#58;	:	90	5A 132	032	&#90;	<b>Z</b>	122	7A 172	032	&#122;	<b>z</b>
27	1B 033	033	<b>ESC</b> (escape)	59	3B 073	033	&#59;	:	91	5B 133	033	&#91;	[	123	7B 173	033	&#123;	{
28	1C 034	034	<b>FS</b> (file separator)	60	3C 074	034	&#60;	<	92	5C 134	034	&#92;	\	124	7C 174	034	&#124;	
29	1D 035	035	<b>GS</b> (group separator)	61	3D 075	035	&#61;	=	93	5D 135	035	&#93;	]	125	7D 175	035	&#125;	}
30	1E 036	036	<b>RS</b> (record separator)	62	3E 076	036	&#62;	>	94	5E 136	036	&#94;	^	126	7E 176	036	&#126;	~
31	1F 037	037	<b>US</b> (unit separator)	63	3F 077	037	&#63;	?	95	5F 137	037	&#95;	_	127	7F 177	037	&#127;	<b>DEL</b>

Source: [www.LookupTables.com](http://www.LookupTables.com)



## Codificación de enteros

- Magnitud y signo (MS)
  - Simétrico
  - Dos representaciones para el cero:
    - + 0 → 000...00
    - - 0 → 100...00
  - Rango (tamaño = n bits)
    - $-(2^{n-1}-1) \leq x \leq +(2^{n-1}-1)$
- Complemento a 2 (C2)
  - No simétrico
  - Una única representación para el cero
  - Rango (tamaño = n bits)
    - $-(2^{n-1}) \leq x \leq +(2^{n-1}-1)$
- ¿Qué ocurre en la sentencia:
  - short a = pow(2,15) +3;?

<b>Ejemplo (tamaño = 4 bits)</b>		
<b>POSITIVOS</b>	<b>MS</b>	<b>C<sub>2</sub></b>
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-0	-8
1001	-1	-7
1010	-2	-6
1011	-3	-5
1100	-4	-4
1101	-5	-3
1110	-6	-2
1111	-7	-1



# Arrays

```
char cadena[] = "hola mundo\n";
```

cadena: 0x0C0002B8

0x0C0002BC

0x0C0002C0

0x0C0002C4

0x0C0002BC

	h	o	l	a
	m	u		n
	d	o	\n	0

fc<sup>2</sup>

99

- Los elementos de un array ocupan posiciones consecutivas de memoria
- Las cadenas de caracteres en C acaban en \0
- ¿En qué dirección está v[16] del array *int vector[100];* si su primer elemento está en la dirección 0x0C00?

# Estructuras

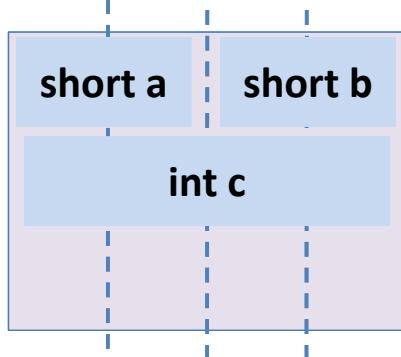
```
struct mistruct {  
    short int a;  
    short int b;  
    int c;  
};
```

```
struct mistruct rec;
```

rec: 0x0C00

0x0C04

0x0C08



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

fc<sup>2</sup>

100