ESTRUCTURA DE COMPUTADORES

Tema 1. Arquitectura del Procesador ARM

Dpto. Arquitectura de Computadores y Automática

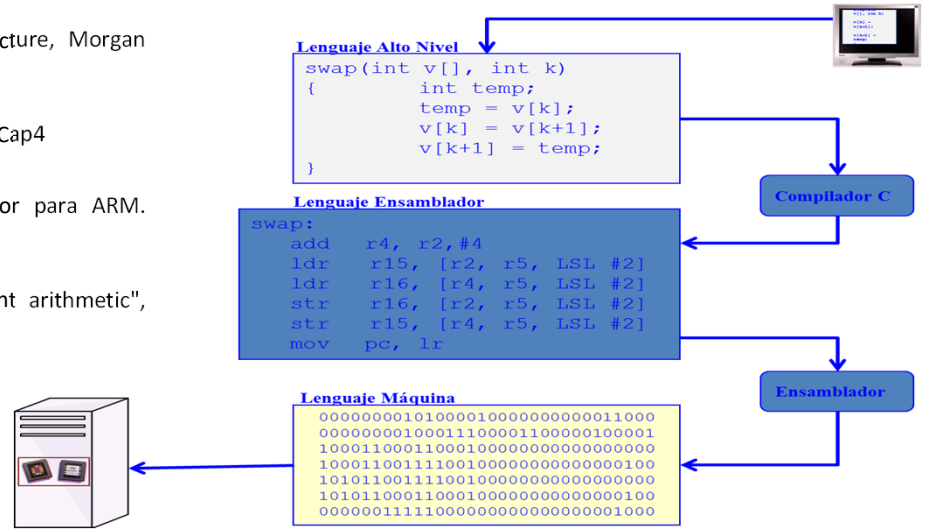
Universidad Complutense de Madrid

ARQUITECTURA DEL PROCESADOR ARM

- Índice
  - Repertorio de instrucciones del ARM
  - Subrutinas
  - Estructura básica de un programa en ensamblador
  - Compilación, paso de C a ensamblador
  - Representación en PF: IEEE 754

- Bibliografía:

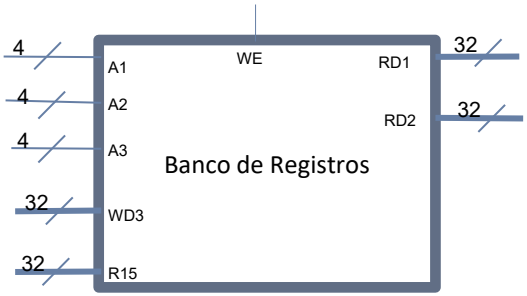
- S.L. Harris & D.M. Harris; Digital Design and Computer Architecture, Morgan Kaufmann 2016. Cap 6
- D. Seal; ARM Architecture Reference Manual, Addison Wesley 2001. Cap4
- L. Piñuel, C. Tenllado. Repaso de programación en ensamblador para ARM. Disponible en CV
- "What every computer scientist should know about floating-point arithmetic", David Goldberg, ACM Computing Surveys. V.23, n.1, pg.5-18, 1991.

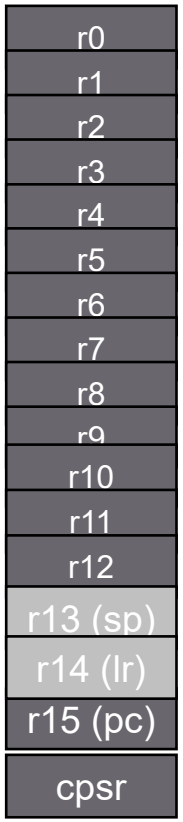


- Índice
  - Repertorio de instrucciones del ARM
  - Subrutinas
  - Estructura básica de un programa en ensamblador
  - Compilación, paso de C a ensamblador
  - Representación en PF: IEEE 754

REGISTROS DEL ARM

- ARM tiene **sólo** 15 registros de propósito general (R0-R14) agrupados en un Banco de Registros
  - R13 = SP (stack pointer) suele usarse como puntero de pila
  - R14 = LR (link register) se usa como enlace o dirección de retorno
  - R15 =PC (program counter) es una copia del Contador de programa
- Todos los registros tienen 32-bits de longitud
  - El Banco de Registros permite acceder a la vez a 2 registros fuente y 1 registro destino





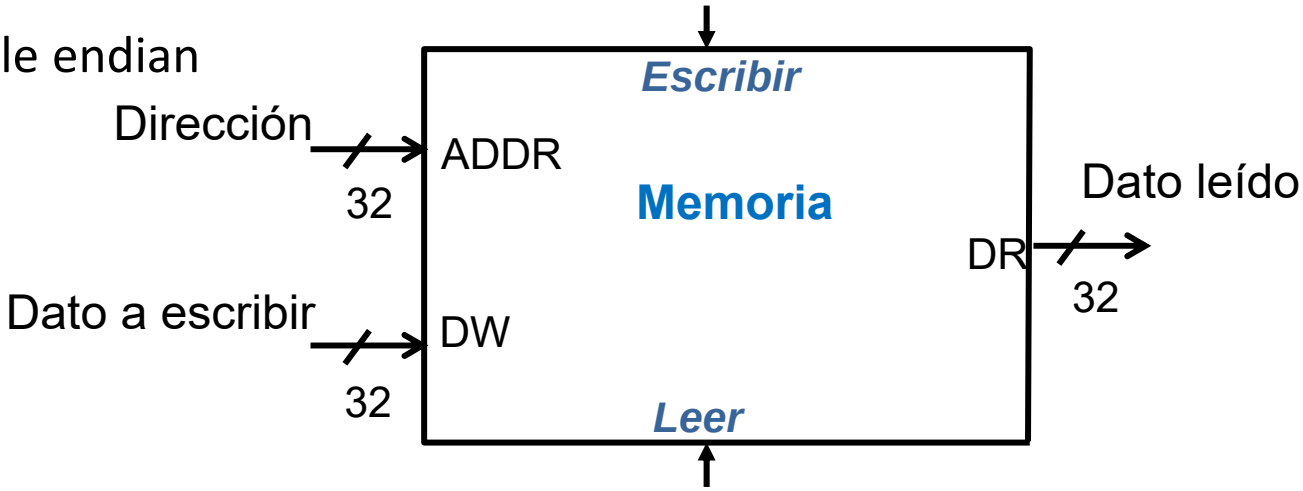
REGISTROS DEL ARM

- Registros de propósito específico:
  - **PC**. Contador de programa. Almacena la dirección en memoria de la siguiente instrucción a ejecutar
  - **cpsr** (Current Program Status Register). Registro de estado
    - Contiene los flags de condición del ARM que pueden ser:
      - Activados por una instrucción
      - Usados para ejecutar condicionalmente una instrucción.

Flag	Nombre	Descripción
N	**N**egativo	El resultado de la instrucción es negativo
Z	**Z**ero	El resultado de la instrucción es cero
C	**C**arry	La instrucción genera un carry out en operaciones sin signo
V	o**V**erflow	La instrucción genera un overflow

MEMORIA

- Memoria accesible por bytes
- 32 bits de dirección => 4 gigabytes direccionables
- Palabras de 4 bytes. Cada acceso a memoria lee o escribe 32 bits
- Accesos alineados
- Big endian/little endian

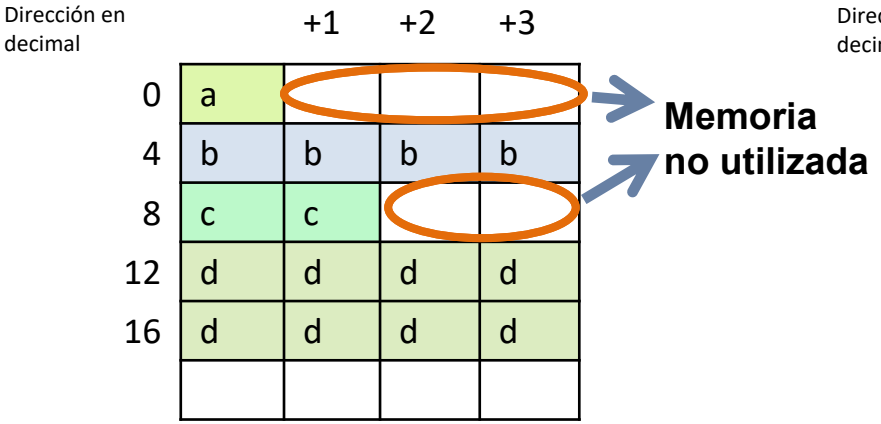


ALINEAMIENTO EN MEMORIA

- Muchas arquitecturas imponen restricciones a las direcciones donde pueden estar situadas las palabras en memoria
  - Un acceso a una palabra en memoria de tamaño N bytes debe hacerse sobre una dirección múltiplo de N
  - Limita las posibles direcciones de variables
    - **1 byte** (char): Puede colocarse en **cualquier dirección**
    - **2 bytes** (short int): en direcciones **múltiplos de 2**
    - **4 bytes** (int, float, instrucciones): en direcciones **múltiplo de 4**
    - **8 bytes** (double float): en direcciones **múltiplo de 8**

ALINEAMIENTO DE VARIABLES

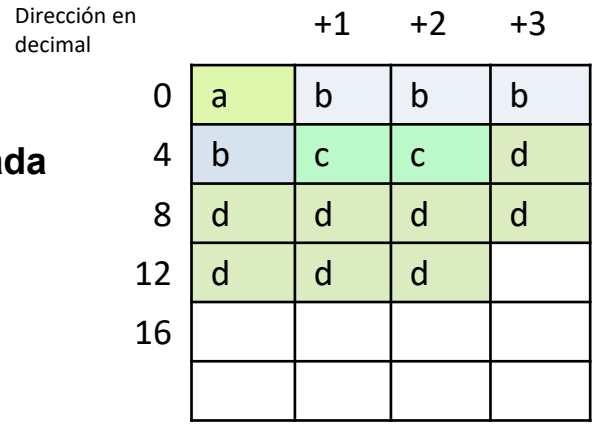
- Ejemplo. Declaramos cuatro variables:
  char a; int b; short int c; double d;



Variables alineadas

Direcciones de comienzo de las variables:

a => 0
b => 4
c => 8
d => 12



Variables no alineadas

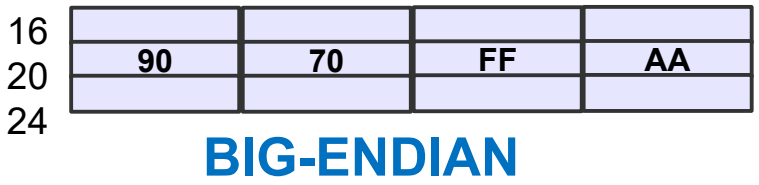
Direcciones de comienzo de las variables:

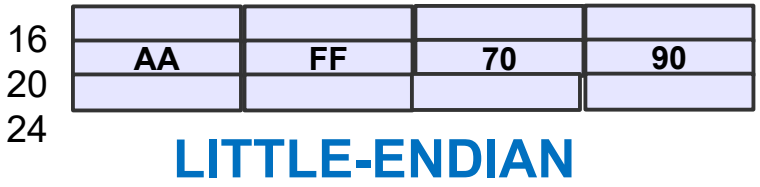
a => 0
b => 1
c => 5
d => 7

ORDEN DE BYTES EN MEMORIA

- **Ejemplo:** dato de 4 bytes (0x9070FFAA) almacenado en la dirección 20. El dato ocupará los bytes 20, 21, 22 y 23.

- ¿Qué byte se pone en la dirección 20, cuál en la dirección 21, …?
  - **BIG-ENDIAN**: La dirección de la variable coincide con la dirección del **byte MÁS significativo**
  - **LITTLE-ENDIAN**: La dirección de la variable coincide con la dirección del **byte MENOS significativo**
  - La mayor parte de los procesadores actuales pueden ser configurados para funcionar como little o big endian



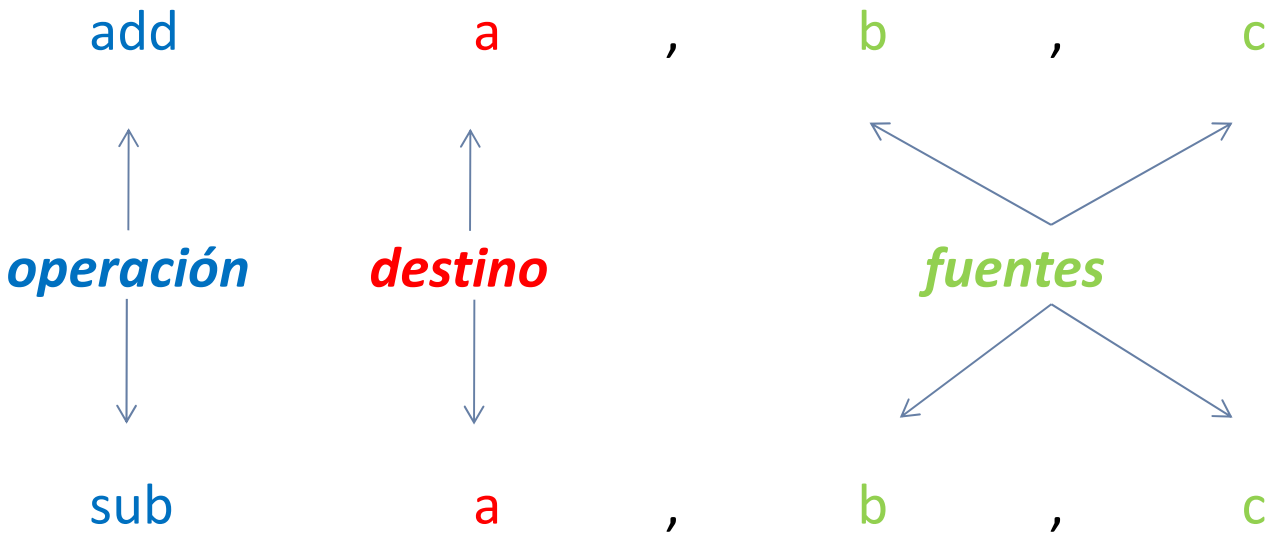


REPERTORIO DE INSTRUCCIONES

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



INSTRUCCIONES ARITMÉTICO-LÓGICAS

Nombre	Operación	Acción
AND	AND lógica	Rd <- Rn AND ShiftOp
ORR	OR lógica	Rd <- Rn OR ShiftOp
EOR	O exclusiva	Rd <- Rn EOR ShiftOp
ADD	Suma	Rd <- Rn + ShiftOp
SUB	Resta	Rd <- Rn - ShiftOp
RSB	Resta inversa	Rd <- ShiftOp - Rn
ADC	Suma con acarreo	Rd <- Rn + ShiftOp + Flag de carry
SBC	Resta con acarreo	Rd <- Rn - ShiftOp - NOT(Flag de carry)
RSC	Resta inversa con acarreo	Rd <- ShiftOp - Rn - NOT(Flag de carry)
CMP	Comparar	Rn - ShiftOp, no almacena el resultado, actualiza los flags de CPSR
CMN	Comparar negado	Rn + ShiftOp, no almacena el resultado, actualiza los flags de CPSR
MOV	Mover entre registros	Rd <- ShiftOp
MVN	Mover negado	Rd <- NOT ShiftOp
BIC	Borrado de bit (bit clear)	Rd <- Rn AND NOT(ShiftOp)
LSL,LSR,ASR,ROR	Desplazamientos	Rd <- Rn ShiftAmount (inm o reg)

INSTRUCCIONES DE ACCESO A MEMORIA

- 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:

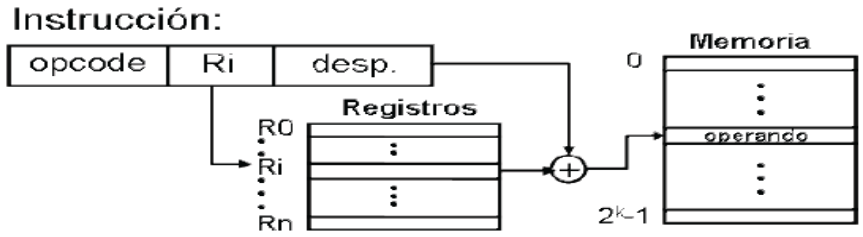
  Banco Regs → Memoria **str** Rd, <Dirección>

INSTRUCCIONES DE ACCESO A MEMORIA

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

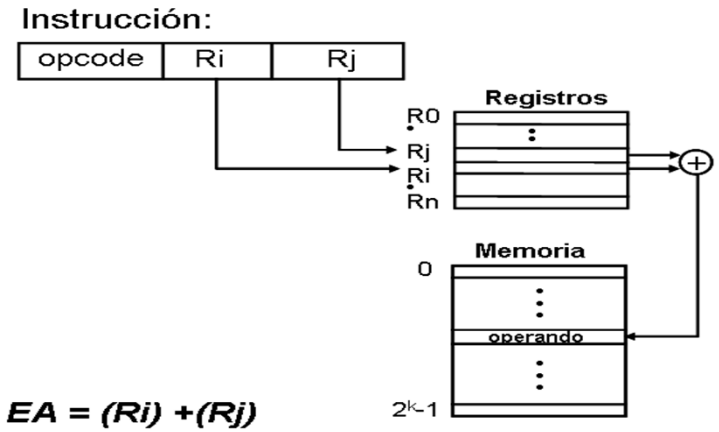
ldr Rd, [Ri]

ldr Rd, [Ri, #±Desplazamiento]



INSTRUCCIONES DE ACCESO A MEMORIA

ldr Rd, [Ri, Rj]



El segundo registro puede utilizarse desplazado

ldr Rd, [Ri, Rj, LSL #N] multiplicar por potencia de 2

ldr Rd, [Ri, Rj, 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:
  - Modo pre-indexado
    - ldr Rd, [Ri, #N]! Se accede a la dirección de memoria [Ri+#N]

      Se actualiza Ri como Ri+#N
  - Modo post-indexado
    - ldr Rd, [Ri], #N Se accede a la dirección de memoria [Ri]

      Se actualiza Ri como Ri+#N

INSTRUCCIONES DE ACCESO A MEMORIA

- En ARM se puede acceder también a media palabra (16 bits) o a un Byte
- Se utilizan las 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, esta tiene que estar alineada en memoria
  - Palabra → dirección múltiplo de 4
  - Media palabra → dirección múltiplo de 2
  - Char → cualquier dirección

INSTRUCCIONES DE SALTO

- 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 …

INSTRUCCIONES DE SALTO

cond	Nemotécnico	Nombre	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: LLAMADAS A FUNCIÓN

- **¿Cómo se hacen las llamadas a función en ensamblador?**
  - **Función que llama:** Se usa la instrucción branch and link

    BL #inm

    Esta función guarda el PC en el Link register, LR (R14) y suma el valor inm al PC para saltar a la dirección de comienzo de la función

  - **Retorno** desde la función llamada: la última instrucción de la función llamada mueve el **link register** al **PC**:

    MOV PC, LR

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*)

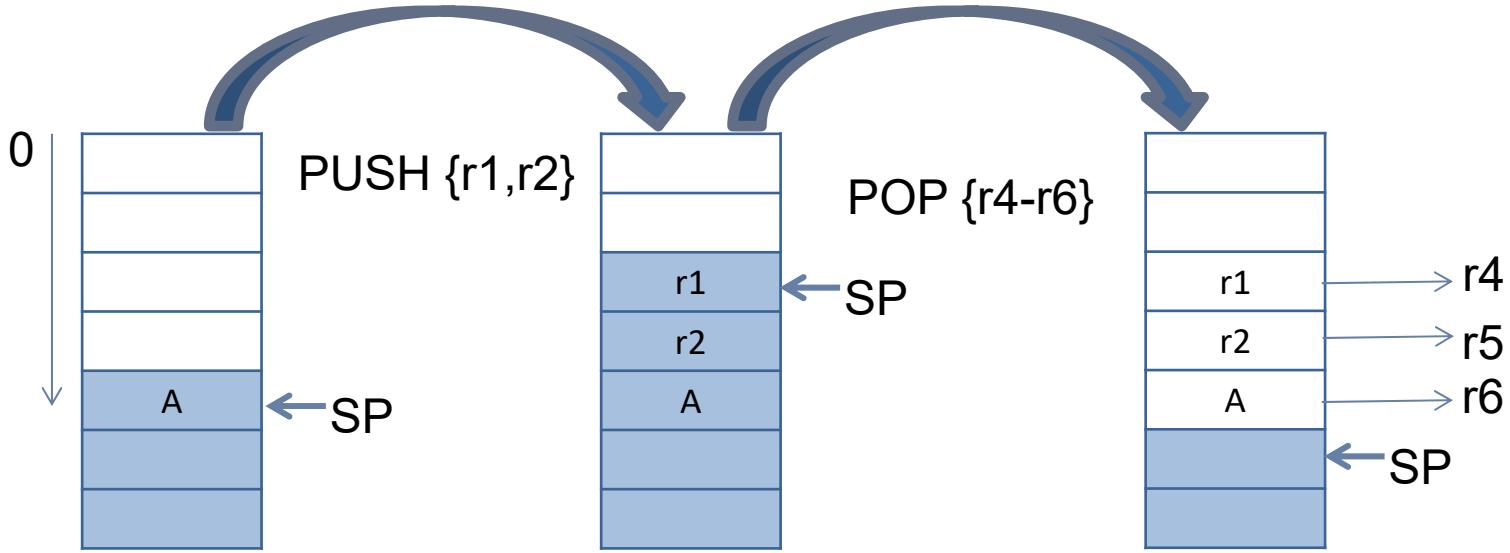
ACCESO A ESTRUCTURAS BÁSICAS DE DATOS

- 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



ARQUITECTURA DEL PROCESADOR ARM

- Índice
  - Repertorio de instrucciones del ARM
  - Subrutinas
  - Estructura básica de un programa en ensamblador
  - Compilación, paso de C a ensamblador
  - Representación en PF: IEEE 754

SUBRUTINAS: ARM PROCEDURE CALL STANDARD

- 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 r0 a r3 sirven para pasar los argumentos
- r0 para escribir el resultado
- De r4 a r11 para almacenar las variables locales
- ¿Cómo 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 r11, 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

```
main
…
mov r0, #dato_in
BL funcion
…
```

```
funcion
…
MOV PC, LR
```

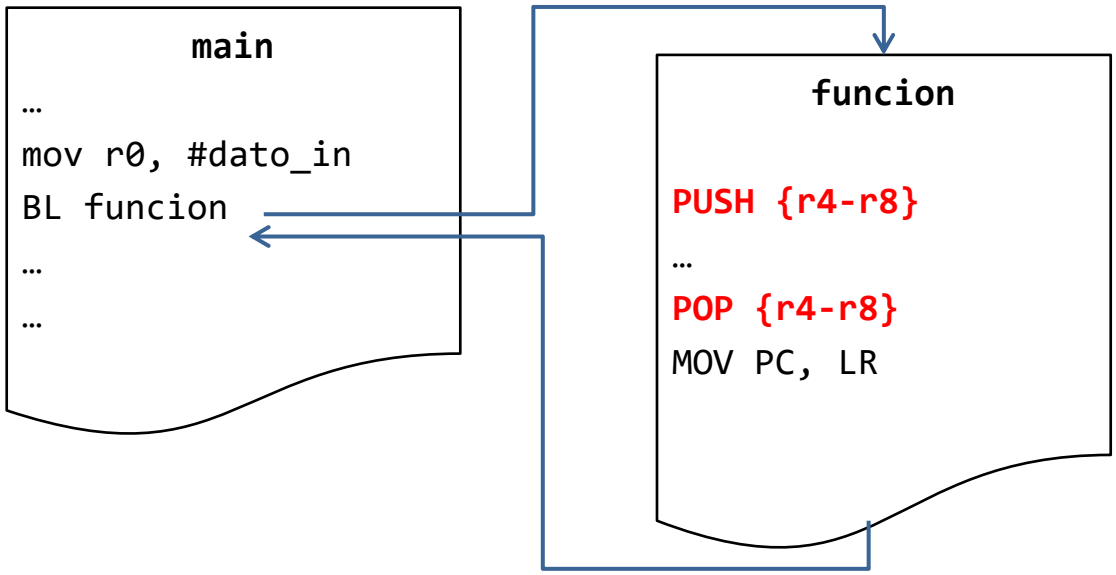
La subrutina funcion no utiliza ningún registro de r4 en adelante para calcular el valor a devolver

```
main
…
mov r0, #dato_in0
mov r1, #dato_in1
mov r2, #dato_in2
PUSH {r1, r2}
BL funcion
POP {r1, r2}
…
```

```
funcion
…
…
…
MOV PC, LR
```

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

SUBRUTINAS

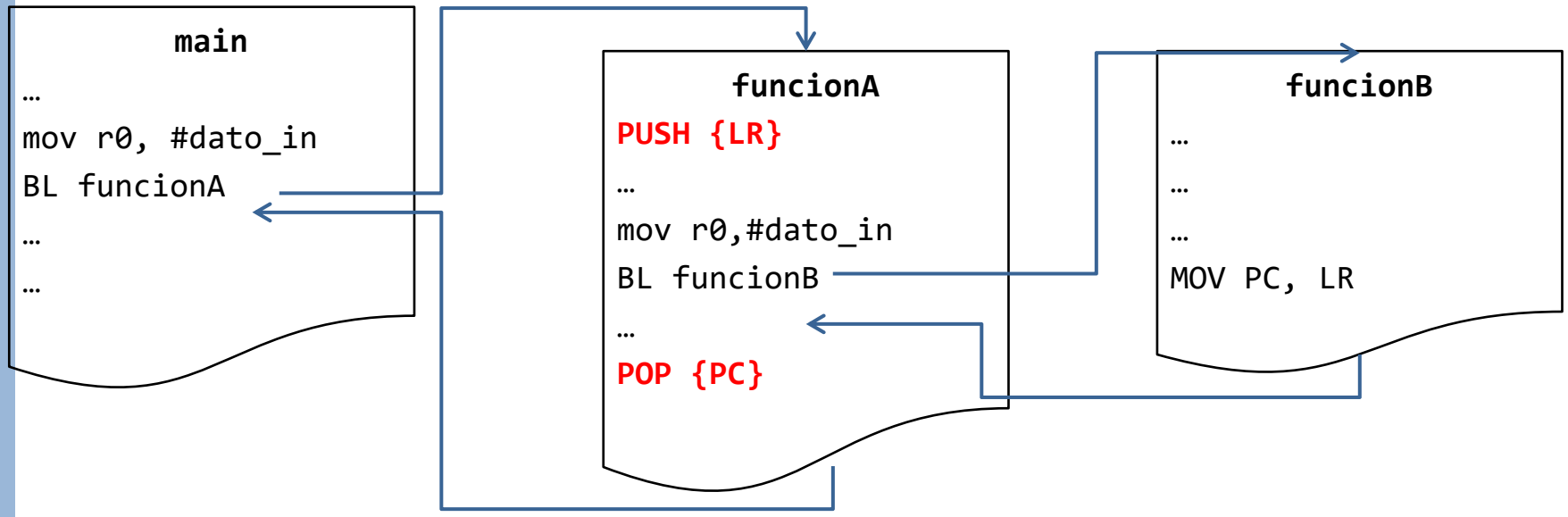


La subrutina funcion utiliza los registros r4 a r8 para calcular el valor a devolver

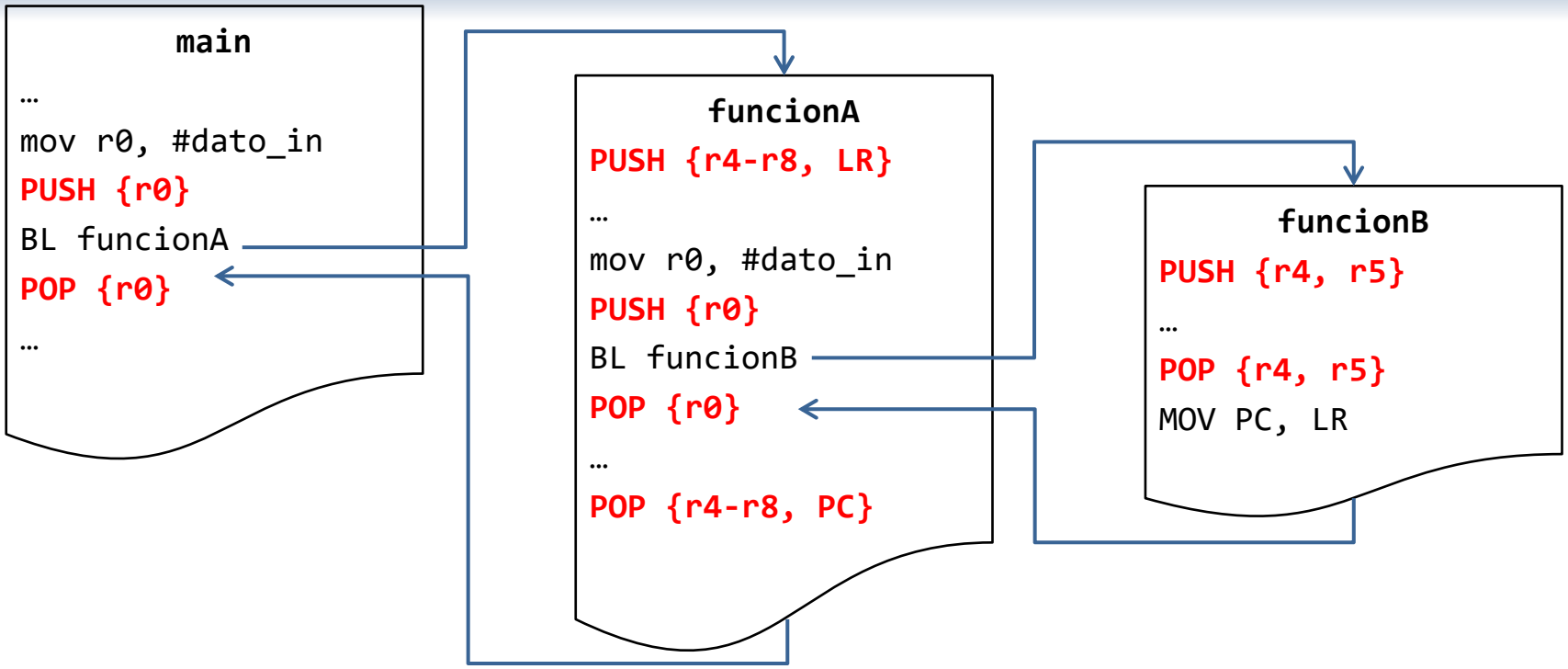
SUBRUTINAS

- 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 {lr}

SUBRUTINAS



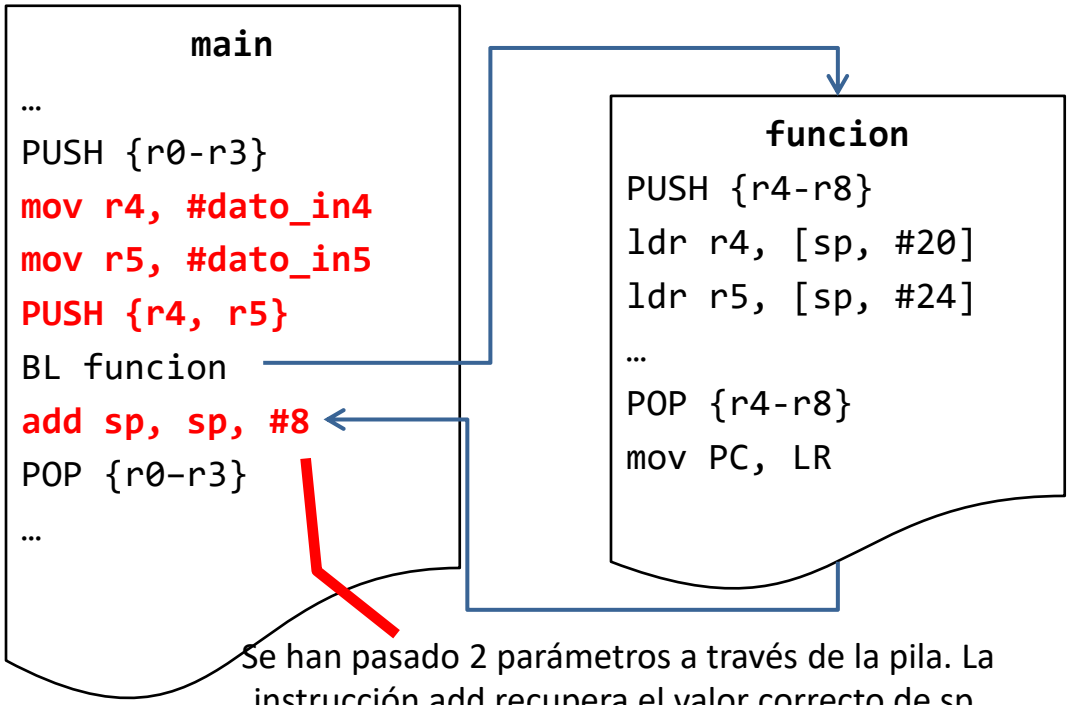
La subrutinas funcionA y funcionB no utilizan ningún registro de r4 en adelante para calcular el valor a devolver



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

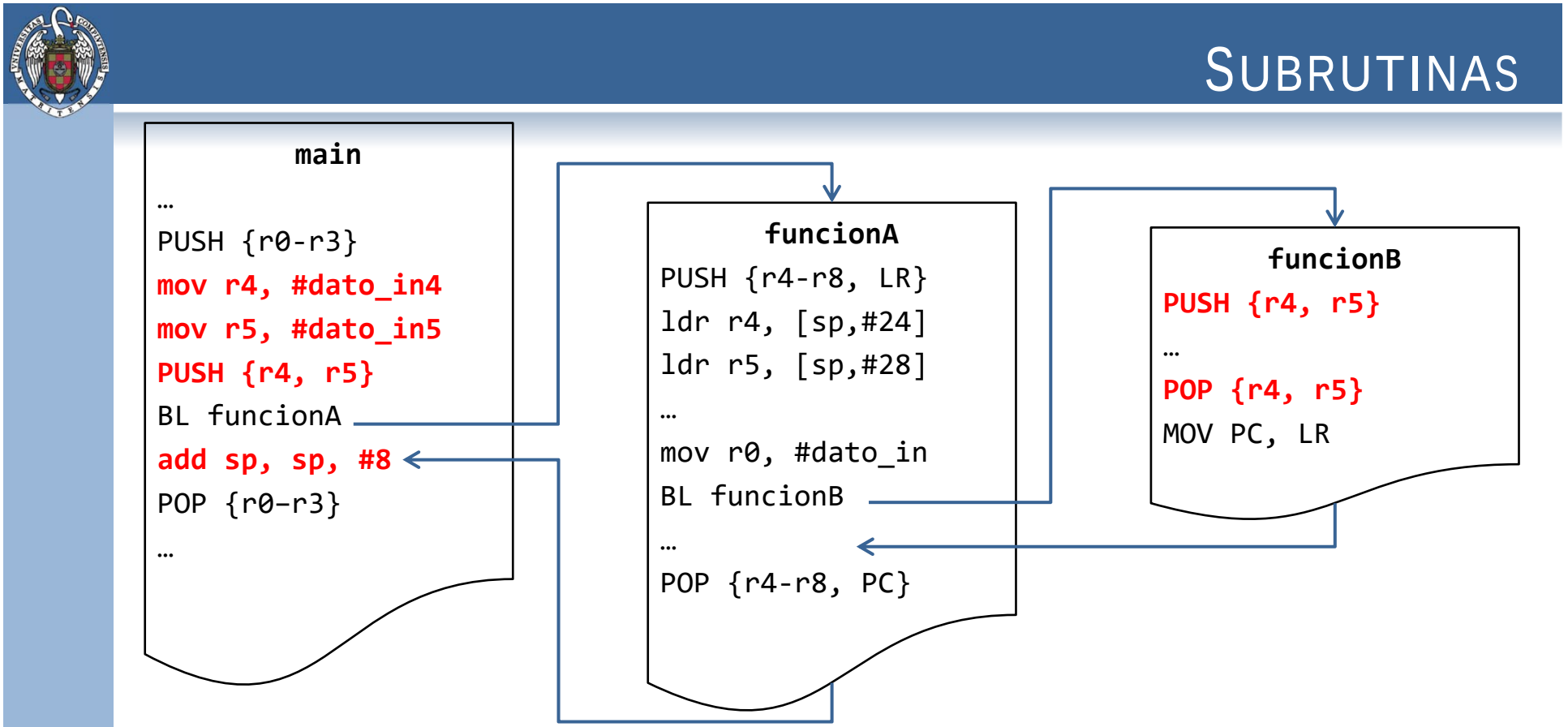
SUBRUTINAS

- ¿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
    - Hay que apilar los registros de los cuales interesa mantener su valor a la vuelta de la subrutina
    - 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



Se han pasado 2 parámetros a través de la pila. La instrucción add recupera el valor correcto de sp.

La subrutina funcion utiliza dos parámetros pasados por pila



SUBRUTINAS

main

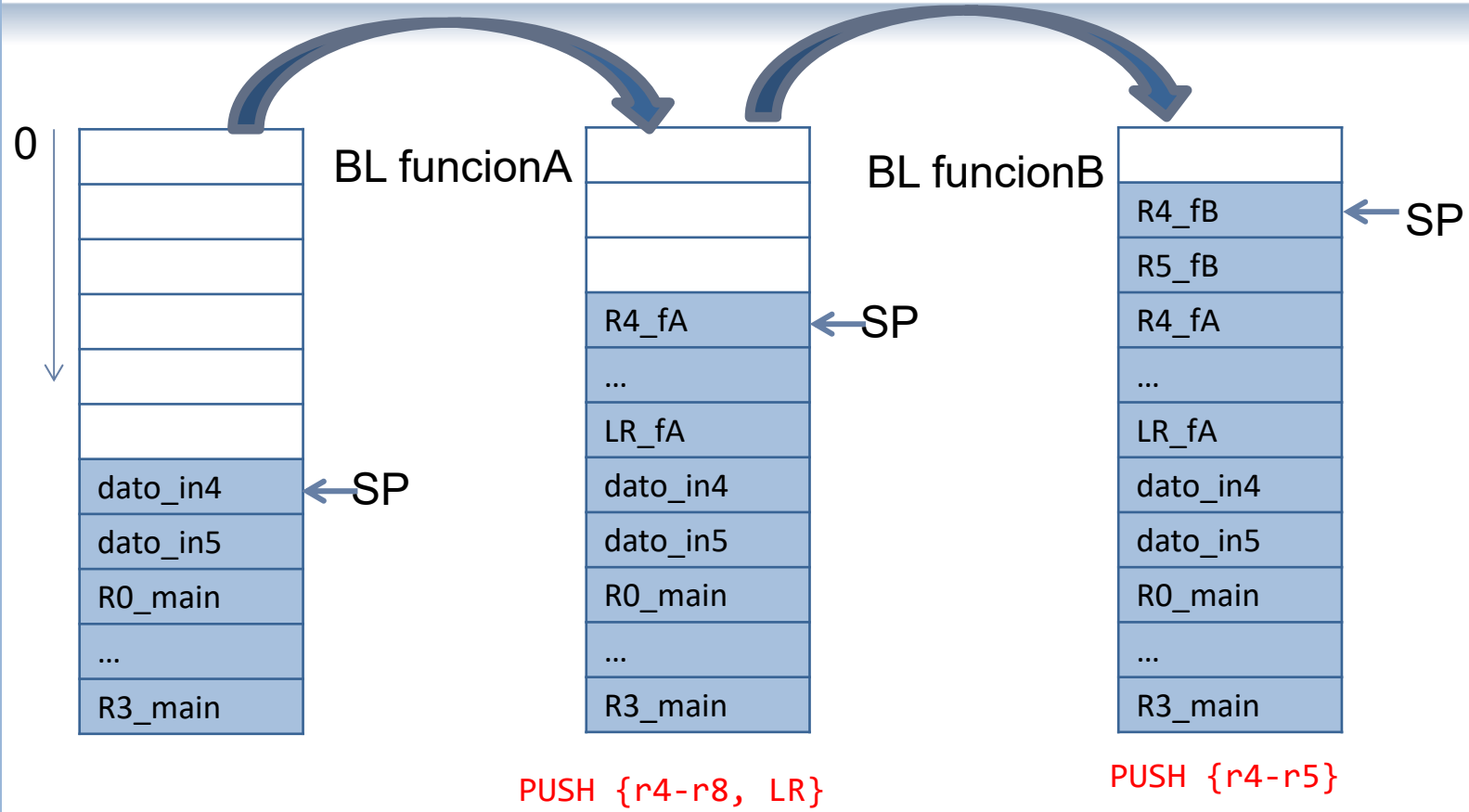
```
…
PUSH {r0-r3}
mov r4, #dato_in4
mov r5, #dato_in5
PUSH {r4, r5}
BL funcionA
add sp, sp, #8
POP {r0-r3}
…
```

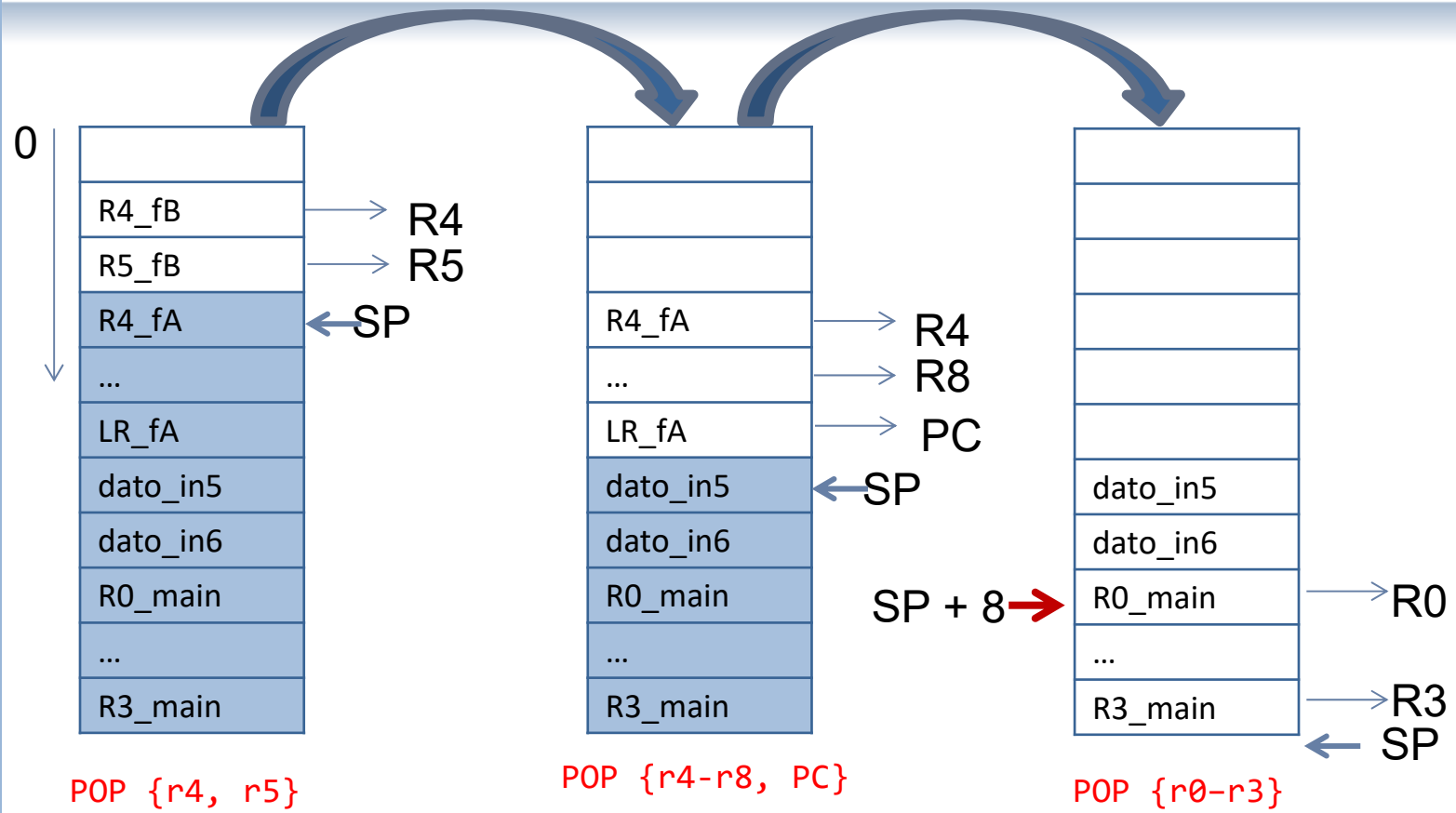
funcionA

```
PUSH {r4-r8, LR}
ldr r4, [sp,#24]
ldr r5, [sp,#28]
…
mov r0, #dato_in
BL funcionB
…
POP {r4-r8, PC}
```

funcionB

```
PUSH {r4, r5}
…
POP {r4, r5}
MOV PC, LR
```


0
BL funcionA
BL funcionB
dato_in4
SP
dato_in5
R0_main
...
R3_main
R4_fA
SP
...
LR_fA
dato_in4
dato_in5
R0_main
...
R3_main
R4_fB
SP
R5_fB
R4_fA
...
LR_fA
dato_in4
dato_in5
R0_main
...
R3_main
PUSH {r4-r8, LR}
PUSH {r4-r5}


0
R4_fB
R5_fB
R4_fA
...
LR_fA
dato_in5
dato_in6
R0_main
...
R3_main
R4
R5
SP
POP {r4, r5}
R4_fA
...
LR_fA
dato_in5
dato_in6
R0_main
...
R3_main
R4
R8
PC
SP
POP {r4-r8, PC}
SP + 8
dato_in5
dato_in6
R0_main
...
R3_main
R0
R3
SP
POP {r0-r3}

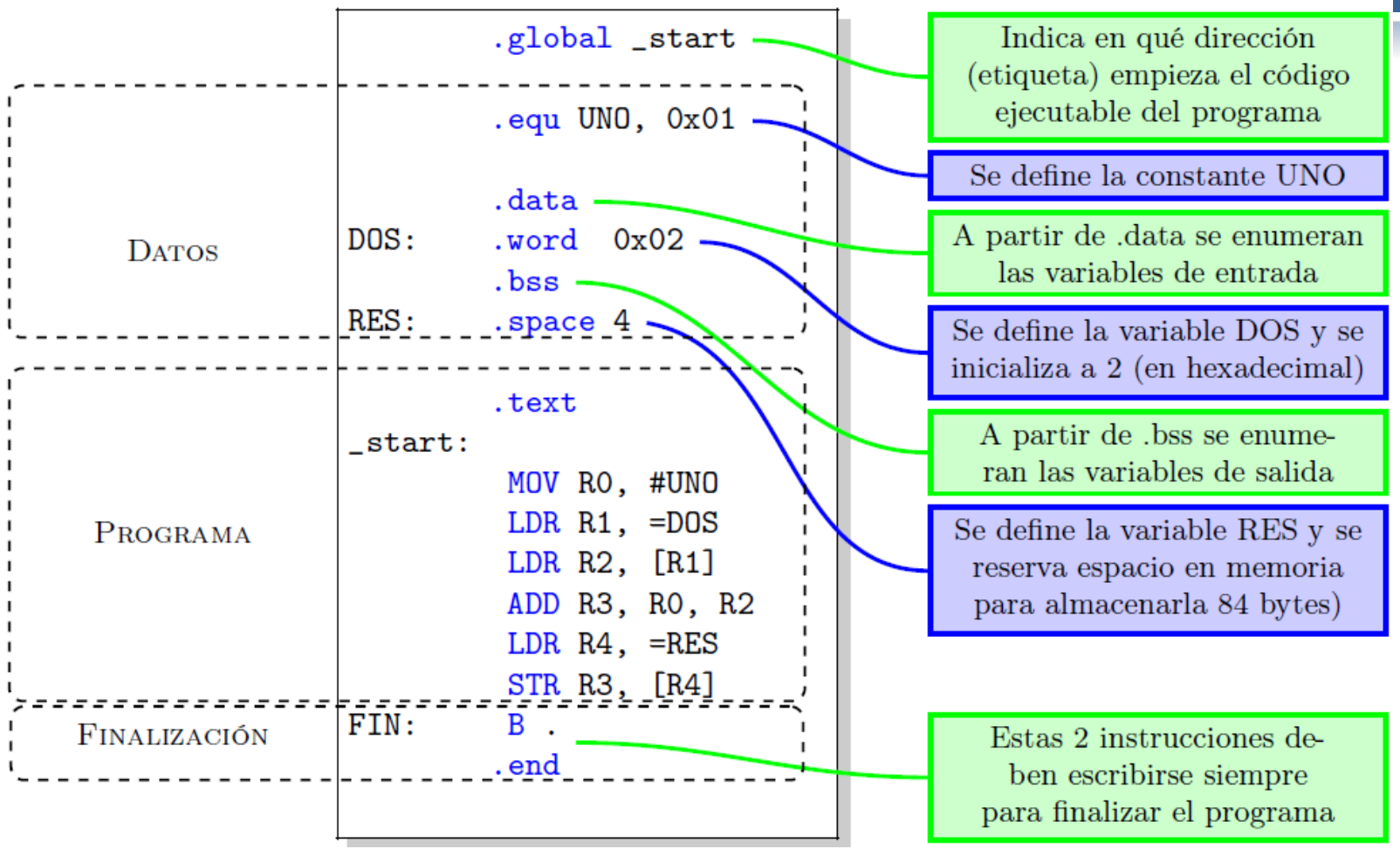
ARQUITECTURA DEL PROCESADOR ARM

- Índice
  - Repertorio de instrucciones del ARM
  - Subrutinas
  - Estructura básica de un programa en ensamblador
  - Compilación, paso de C a ensamblador
  - Representación en PF: IEEE 754

PSEUDO-INSTRUCCIONES DE LOAD Y STORE

- 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**.

Estructura básica de un programa ensamblador



ESTRUCTURA BÁSICA DE UN PROGRAMA ENSAMBLADOR

- Proceso de enlazado de un programa en ensamblador:
  - Permite 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 …

ARQUITECTURA DEL PROCESADOR ARM

- Índice
  - Repertorio de instrucciones del ARM
  - Subrutinas
  - Estructura básica de un programa en ensamblador
  - Compilación, paso de C a ensamblador
  - Representación en PF: IEEE 754

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
  - Contras
    - Limitaciones en el acceso y gestión de los registros
    - 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
  - Contras
    - Más difícil de aprender
    - Gestión casi imposible de estructuras de datos complicadas
    - Ninguna portabilidad (sólo en la misma familia arquitectónica)

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
  - 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(), inicializa el *heap*
  - Después de estas inicializaciones, el programa (*C startup code)* salta a 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 a cabo una tarea específica.
  - También hay varios tipos de datos:
    - Variables locales
    - Variables globales
    - Constantes

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 sea 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 programan en ensamblador aquellas partes donde haya requisitos estrictos de eficiencia o aquellas en las que 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 **DEFINIDA**

programa.c

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

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 la rutina de 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
    - Un fichero objeto es un fichero binario estructurado que contiene una lista de secciones con su contenido, y una serie de estructuras adicionales
    - La *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

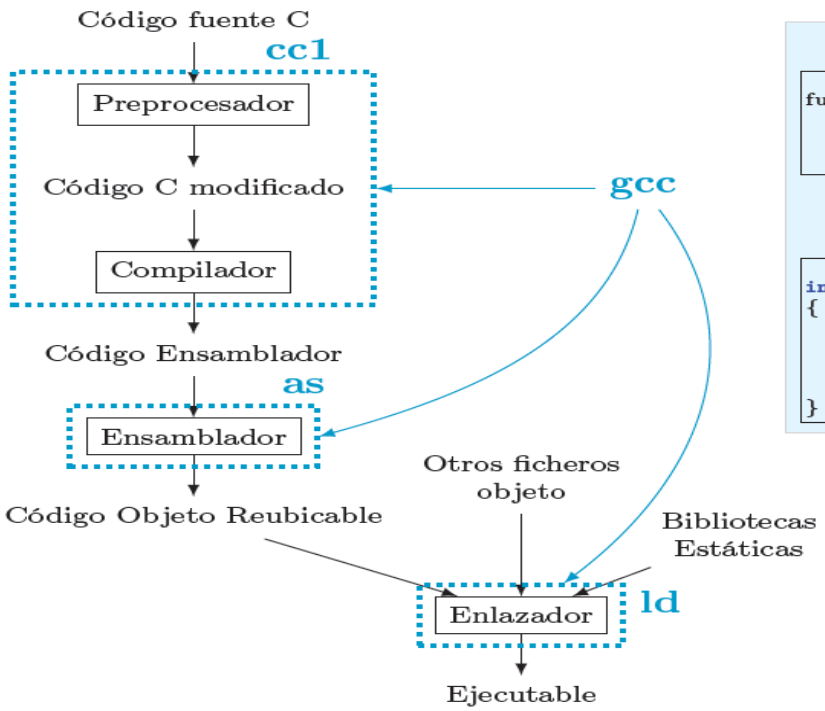
COMPILACIÓN, PASO DE C A ENSAMBLADOR

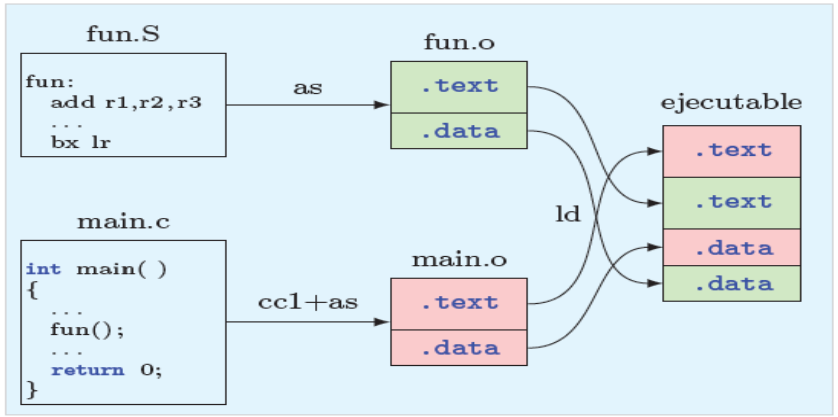
- Creación del fichero ejecutable (I):
  - Cada fichero en código C pasa por el preprocesador que modifica el código fuente original siguiendo directivas de preprocesado (#define, #include, etc.).
  - El código resultante es compilado, transformándolo en código ensamblador.
  - El ensamblador genera 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



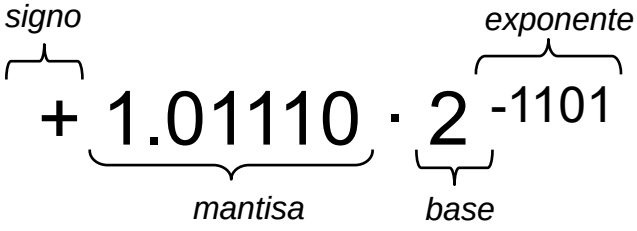


ARQUITECTURA DEL PROCESADOR ARM

- Índice
  - Repertorio de instrucciones del ARM
  - Subrutinas
  - Estructura básica de un programa en ensamblador
  - Compilación, paso de C a ensamblador
  - Representación en PF: IEEE 754

REPRESENTACIÓN EN COMA FLOTANTE

- La representación en coma flotante está basada en la **notación científica**:
  - La coma decimal no se halla en una posición fija dentro de la secuencia de bits, sino que su posición se indica como una potencia de la base:



- En todo número en coma flotante se distinguen tres componentes:
  - **Signo**: indica el signo del número (0= positivo, 1=negativo)
  - **Mantisa**: contiene la magnitud del número (en binario puro)
  - **Exponente**: contiene el valor de la potencia de la base (sesgado)
  - La **base** queda implícita y es común a todos los números, la más usada es 2

REPRESENTACIÓN EN COMA FLOTANTE

- El **valor** de la secuencia de bits (s, e_{p-1}, ..., e_0, m_{q-1}, ..., m_0) es: $(-1)^s \cdot V(m) \cdot 2^{V(e)}$
- Dado que un mismo número puede tener varias representaciones

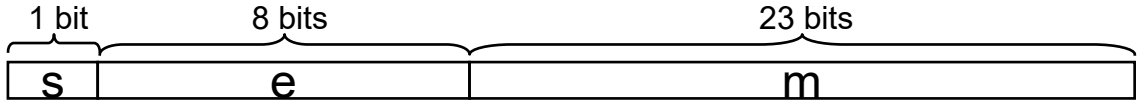
  $(0.110 \cdot 2^5 = 110 \cdot 2^2 = 0.0110 \cdot 2^6)$

  los números suelen estar normalizados:

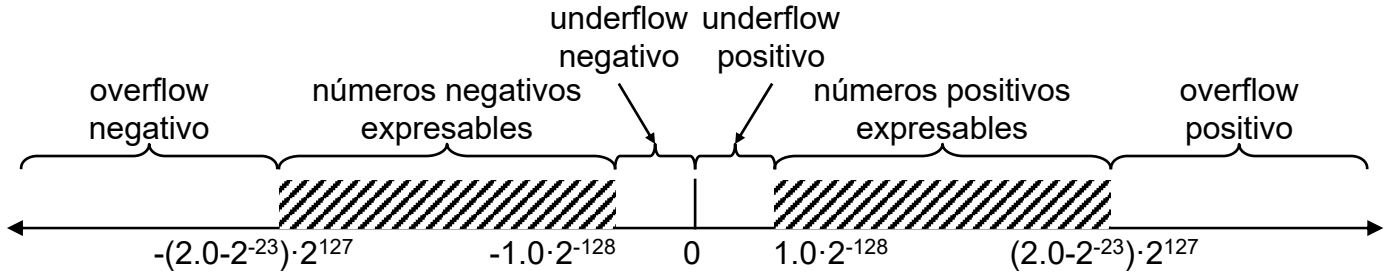
  - Un número está normalizado si tiene la forma $1.xx... \cdot 2^{xx...}$ (ó $0.1xx ... \cdot 2^{xx...}$)
  - Dado que los números normalizados en base 2 tienen siempre un 1 a la izquierda, éste suele quedar implícito (pero debe ser tenido en cuenta al calcular el valor de la secuencia)

REPRESENTACIÓN EN COMA FLOTANTE

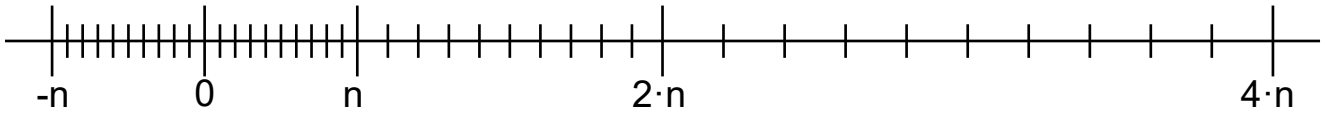
- Sea el siguiente formato de coma flotante de 32 bits (base 2, normalizado)



  - El rango de valores representable por cada uno de los campos es:
    - **Exponente** (8 bits con sesgo de 128) : -128 ... +127
    - **Mantisa** (23 bits normalizados) : los valores binarios representables oscilan entre 1.00...00 y 1.11...11, es decir entre 1 y $2-2^{-23}$ (2-ulp) (1.11...1 = 10.00...0 - 0.0...1)



  - Obsérvese que la cantidad de números representables es 2^{32} (igual que en coma fija). Lo que permite la representación en coma flotante es ampliar el rango representable a costa de aumentar el espacio entre números representable (un espacio que no es uniforme).



IEEE Standard 754 for Binary Floating-Point Arithmetic.

Prof. Kahan

1989
ACM Turing
Award Winner!

http://www.cs.berkeley.edu/~wkahan/

IEEE 754

- **2 formatos** con signo explícito, representación sesgada del exponente (sesgo igual a $(2^{n-1}-1)$), mantisa normalizada con un 1 implícito (1.M) y base 2.
  - ***precisión simple*** (32 bits): 1 bit de signo, 8 de exponente, 23 de mantisa
    - $1.0 \cdot 2^{-126}$ ... $(2-2^{-23}) \cdot 2^{127} = 1.2 \cdot 10^{-38}$ .. $3.4 \cdot 10^{38}$
  - ***precisión doble*** (64 bits): 1 bit de signo, 11 de exponente, 52 de mantisa
    - $1.0 \cdot 2^{-1022}$ ... $(2-2^{-52}) \cdot 2^{1023} = 2.2 \cdot 10^{-308}$ .. $1{,}8 \cdot 10^{308}$
- **2 formatos** ampliados para cálculos intermedios (43 y 79 bits).

IEEE 754

- **Codificaciones** con significado especial
  - ***Infinito*** (e=255, m=0): representan cualquier valor de la región de overflow
  - ***NaN*** (*Not-a-Number*) (e=255, m>0): se obtienen como resultado de operaciones inválidas
  - ***Número denormalizado*** (e=0, m>0): es un número sin normalizar cuyo bit implícito se supone que es 0. Al ser el exponente 0, permiten representar números en las regiones de underflow. El valor del exponente es el del exponente más pequeño de los números no denormalizados: -126 en precisión simple y –1022 en doble.
  - ***Cero*** (e=0, m=0): número no normalizado que representa al cero (en lugar de al 1)

- **Excepciones**:
  - **Operación inválida**: $\infty \pm \infty$, $0 \times \infty$, $0 \div 0$, $\infty \div \infty$, x mod 0, $\sqrt{x}$ cuando $x<0$, $x=\infty$
  - **Inexacto**: el resultado redondeado no coincide con el real
  - **Overflow** y **underflow**
  - **División por cero**

IEEE 754

- Ejemplo:

0	0110 1000	101 0101 0100 0011 0100 0010

- Signo: 0 => positivo
- Exponente:
  - $0110\ 1000_{dos} = 104_{diez}$
  - Eliminación del sesgo: 104 - 127 = -23
- Mantisa:

$1 + 1x2^{-1} + 0x2^{-2} + 1x2^{-3} + 0x2^{-4} + 1x2^{-5} + ...$
$= 1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-22} = 1.0 + 0.666115$

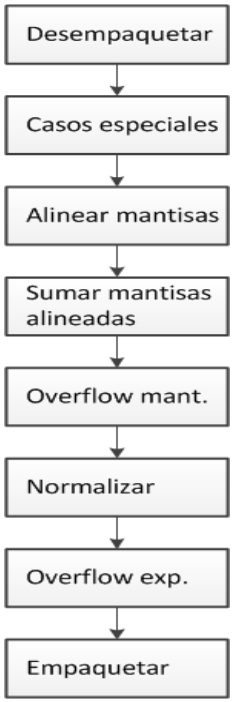
Representa: $1.666115_{diez} * 2^{-23} \sim 1.986 * 10^{-7}$

- **Redondeo:**
  - El estándar exige que el resultado de las operaciones sea el mismo que se obtendría si se realizasen con precisión absoluta y después se redondease.
  - Hacer la operación con precisión absoluta no tiene sentido pues se podrían necesitar operandos de mucha anchura.
  - **Existen 4 modos de redondeo**:
    - Redondeo al **más cercano** (al par en caso de empate)
    - Redondeo a **más infinito** (por exceso)
    - Redondeo a **menos infinito** (por defecto)
    - Redondeo a **cero** (truncamiento)

- Al realizar una operación ¿cuántos bits adicionales se necesitan para tener la precisión requerida?
  - Un bit **r** para el redondeo
  - Un bit **s** (sticky) para determinar, cuando r=1, si el número está por encima de 0,5

Operaciones de redondeo		
Tipo de redondeo	Signo del resultado≥0	Signo del resultado<0
- ∞		+1 si (r or s)
+ ∞	+1 si (r or s)	
0		
Más próximo	+1 si (r and p_0) or (r and s)	+1 si (r and p_0) or (r and s)

- Objetivo: Sumar en formato punto fijo de números con el exponente alineado

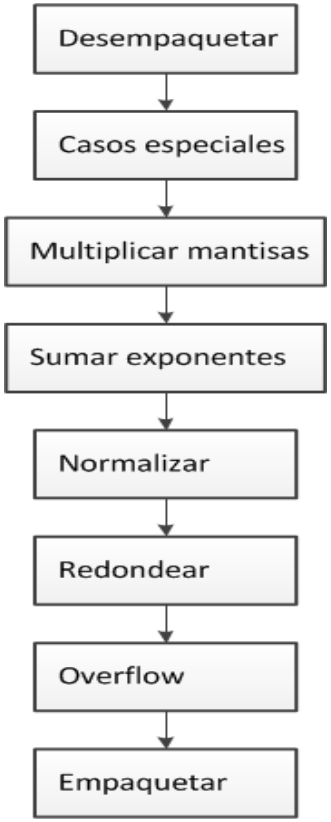


Ejemplo:

$$\begin{array}{r} 9.555\cdot10^{4} \\ +\ 9.996\cdot10^{7} \\ \hline \end{array} \longrightarrow \begin{array}{r} 0.009555\cdot10^{7} \\ +\ 9.996000\cdot10^{7} \\ \hline 10.005555\cdot10^{7} \end{array} \longrightarrow 1.0005555\cdot10^{8} \longrightarrow 1.001\cdot10^{8}$$

alinear normalizar redondeo

IEEE 754: MULTIPLICACIÓN



Ejemplo:

$$\begin{array}{r} 4.555 \cdot 10^{4} \\ \times\ 2.996 \cdot 10^{7} \\ \hline 13.646780 \cdot 10^{11} \end{array} \longrightarrow 1.3646780 \cdot 10^{12} \longrightarrow 1.365 \cdot 10^{12}$$

normalizar redondeo