Fundamentos de Computadores

2º Cuatrimestre

2012-2013

fc²

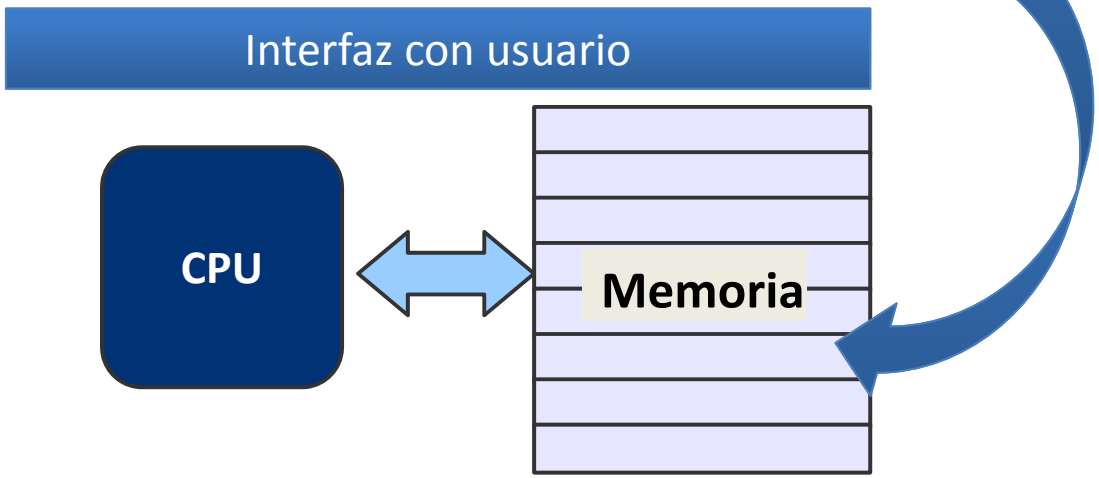
Módulo 9: Repertorio de instrucciones y lenguaje ensamblador

¿Qué vamos a estudiar en este tema?

```
program simple;
var
   a, b, p:
integer
begin
   a := 5;
   b := 3;
   p := a * b;
end
```

```
simple:
   leal   4(%esp), %ecx
   andl   $-16, %esp
   pushl  -4(%ecx)
   pushl  %ebp
   movl   %esp, %ebp
   pushl  %ecx
   subl   $16, %esp
   movl   $5, -16(%ebp)
   movl   $3, -
12(%ebp)…….
```

```
457f 464c 0101 0001 0000 0000 0000 0000
0002 0003 0001 0000 8280 0804 0034 0000
0df0 0000 0000 0000 0034 0020 0007 0028
0022 001f 0006 0000 0034 0000 8034 0804
8034 0804 00e0 0000 00e0 0000 0005 0000
0004 0000 0003 0000 0114 0000 8114 0804
8114 0804 0013 0000 0013 0000 0004 0000
0001 0000 0001 0000 0000 0000 8000 0804
8000 0804 046c 0000 046c 0000 0005 0000
1000 0000 0001 0000 046c 0000 946c 0804
946c 0804 0100 0000 0104 0000 0006 0000
1000 0000 0002 0000 0480 0000 9480 0804
9480 0804 00c8 0000 00c8 0000 0006 0000
0004 0000 0004 0000 0128 0000 8128 0804
8128 0804 0020 0000 0020 0000 0004 0000
```



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

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 (compactación de código)
  - Jazelle (implementación HW de Java VM). No disponible en la versión que usaremos.

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 importantes
  - StrongARM, XScale, Cortex

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:

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

Definiciones básicas

- Analogía con una lengua:

Palabra	←→	Instrucción
Palabras del diccionario	←→	Repertorio de instrucciones
Lenguaje completo	←→	Lenguaje ensamblador

Tipos de instrucciones

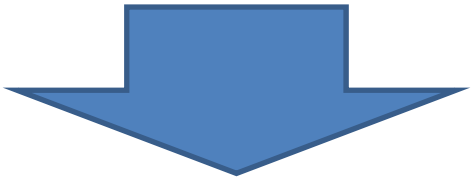
- Un computador debe ser capaz de:
  - Realizar las operaciones matemáticas elementales: **Instrucciones aritmetico-lógicas**
  - Obtener y almacenar los datos que utiliza la instrucción: **Instrucciones de acceso a memoria**
  - Modificar el flujo secuencial del programa: **Instrucciones de salto**
  - 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 r1,r2,r3,r4 respectivamente.

```
add r5,r2,r1
add r6,r3,r4
mul r7,r5,r6
```

Tareas del procesador al ejec. inst.

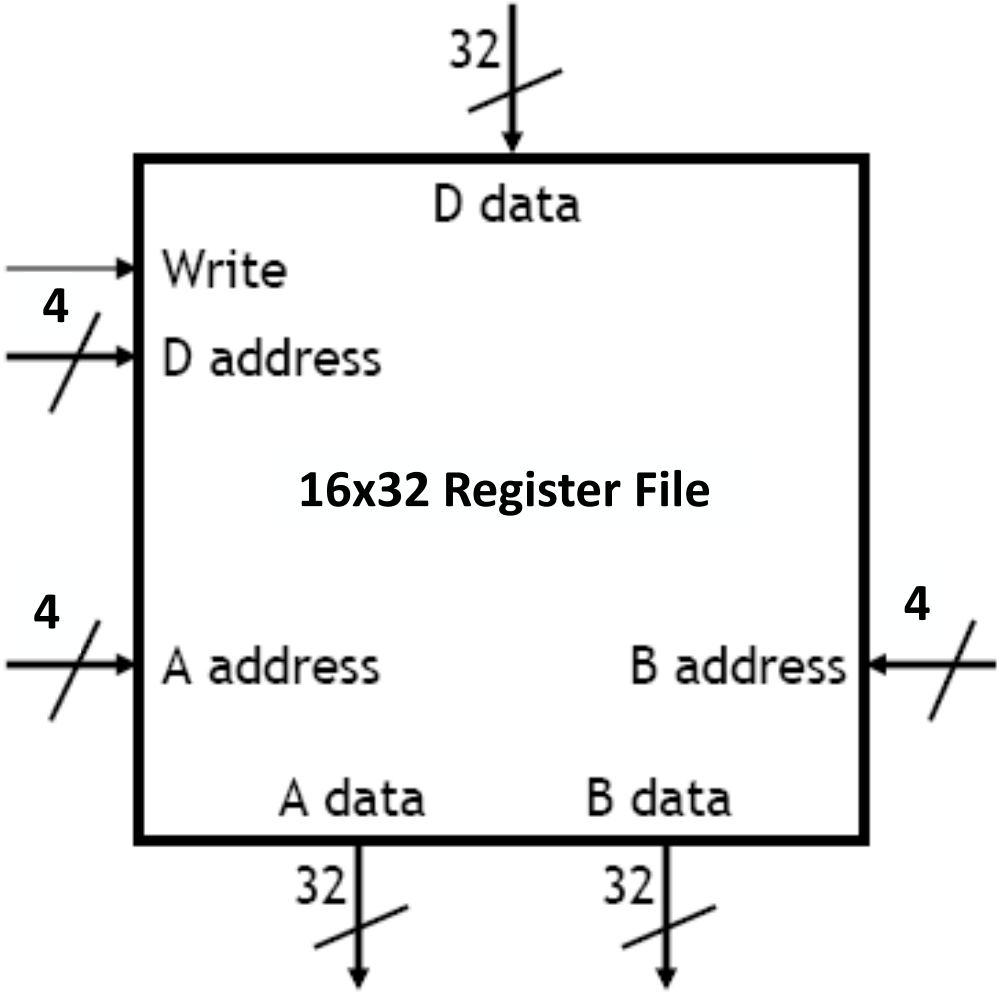
- Pensemos un poco más a fondo qué tareas tiene que realizar el procesador cuando ejecuta una instrucción:
  1. Detectar tipo de instrucción a ejecutar → P.ej. ADD
  2. Leer de *algún lugar* los ops. fuente
  3. Realizar la suma de los dos operandos con algún HW
  4. Guardar el resultado en *algún lugar*

- ¿**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???

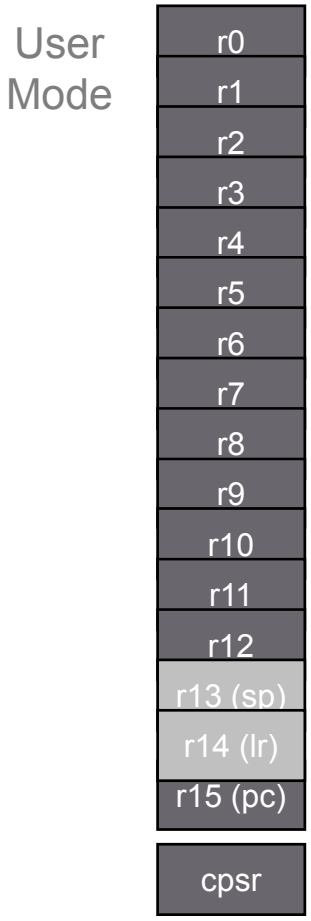
Banco de Registros del ARM



Registros del ARM

- 32-bits de longitud
- En cada momento, se puede acceder a:
  - 15 registros de propósito general (R0-R14)
    - R13 suele usarse como puntero de pila
    - R14 se usa como enlace o dirección de retorno
  - Un registro (PC) contador de programa (R15)
  - Un registro de estado actual del programa (CPSR)

Registros visibles en modo usuario



Necesidad de la Memoria

- Lenguaje de programación: maneja estructuras de datos complejas (por ejemplo una matriz de datos).
- Pueden contener muchos más datos que el número de registros que nos proporciona el computador (en nuestro caso, 15)
- Necesitamos una estructura de almacenamiento de datos mucho mayor que el Banco de Registros → La MEMORIA

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

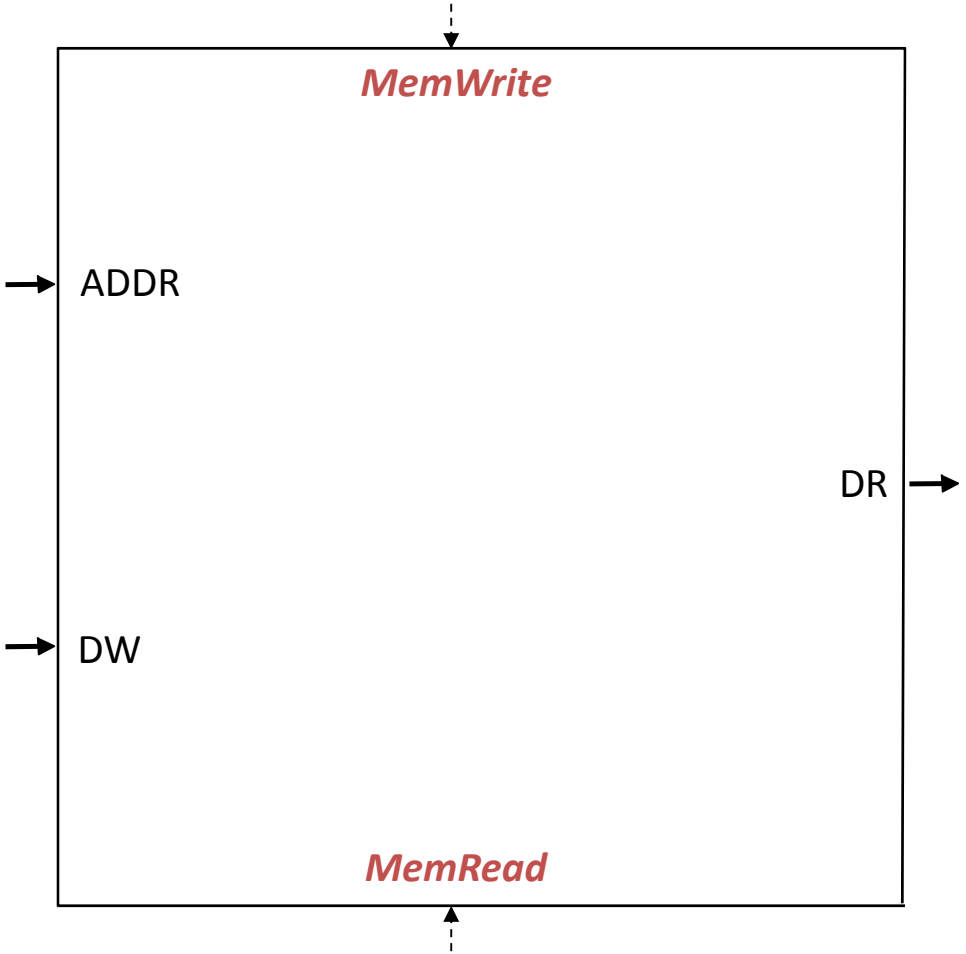
Ejemplo

- 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?

Memoria del ARM



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 *direccionable* a nivel de byte
- Si disponemos de *k* bits para expresar una dirección
  - Podremos acceder a 2^k bytes diferentes
  - Si k=16 -> 2^{16} bytes → 64Kbytes direccionables

Organización de la memoria

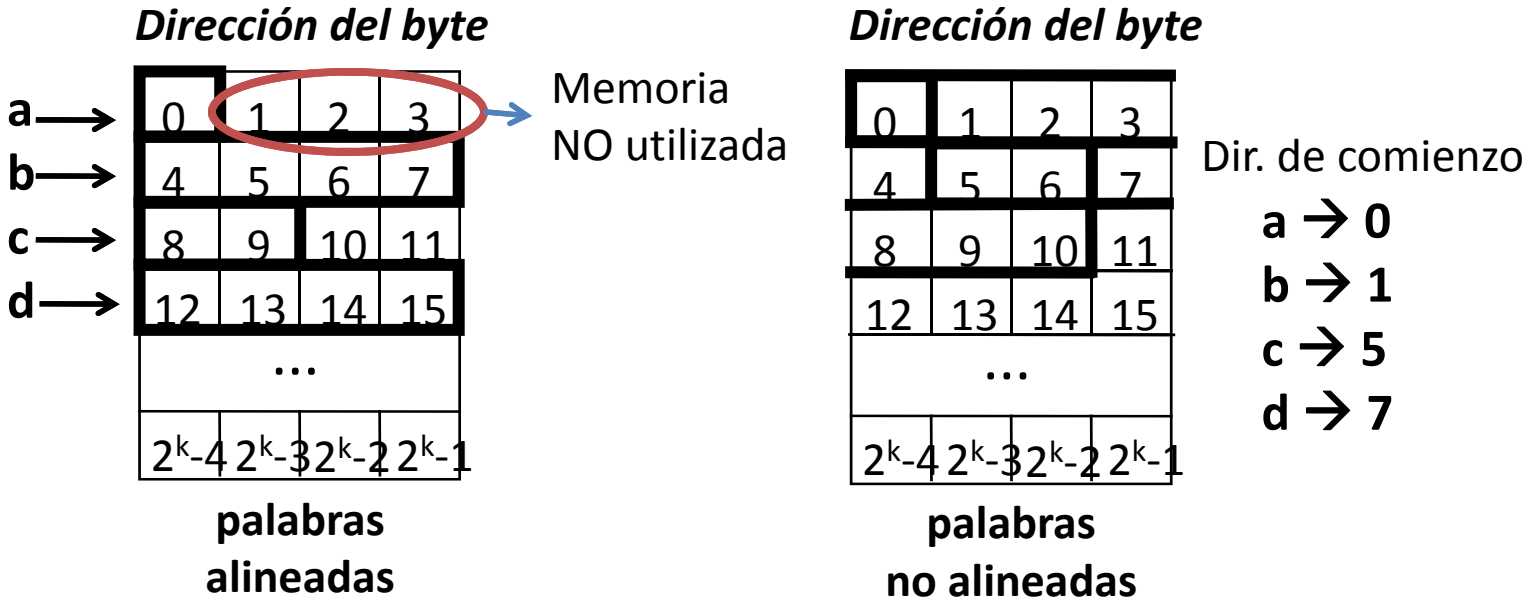
- En la práctica, y durante este curso, NUNCA accederemos a nivel de byte:
  - Una instrucción ocupa 4 bytes
  - Un dato (por ejemplo, un entero) ocupa 4 bytes
  - Siempre accederemos **a direcciones múltiplo de 4**
- En muchos sistemas reales existen restricciones a la dirección de comienzo de un dato
  - 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 múltiplo de 4
    - Misma restricción para datos de 4 bytes (int, float)
  - Dato de tamaño 2 bytes (short int) -> Sólo puede comenzar en direcciones múltiplos de 2
  - Data de tamaño byte (char) -> Puede comenzar en cualquier dirección

Organización de la memoria: alineamiento

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



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

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
- ¿Cómo se organizan los bytes dentro de la palabra?
  - Concepto de *endiannes*

Organización de la memoria: *endiannes*

- Sea la palabra (4 bytes), codficada en hexadecimal, $AABBCCDD_{16}$
  - 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

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*

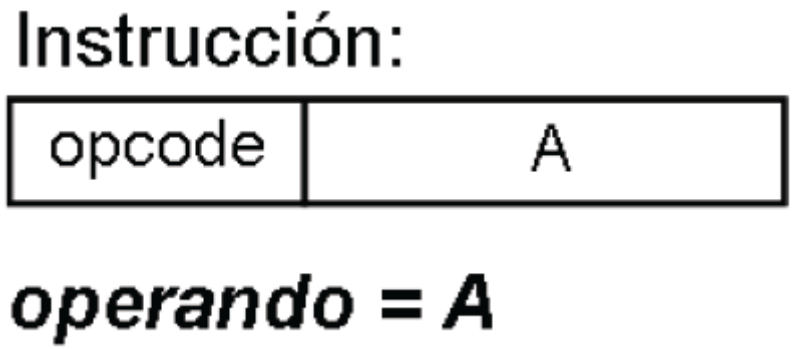
- ARM admite ambas organizaciones

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
  - Distintas posibilidades:

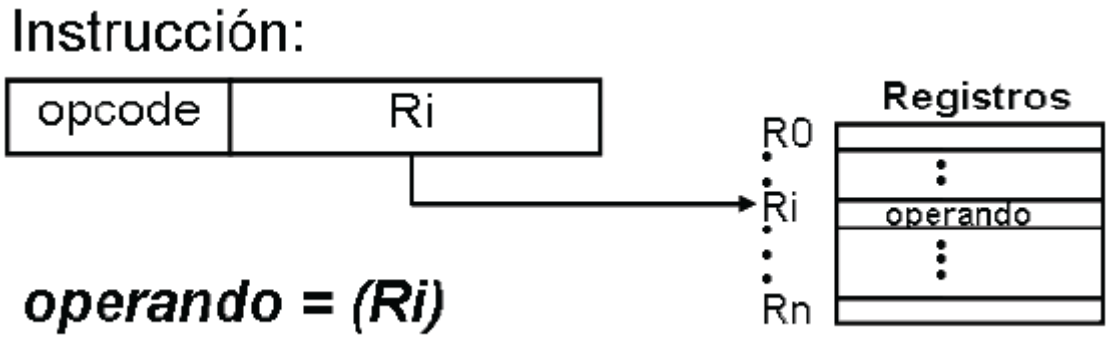
Inmediato

– El operando está contenido en la propia instrucción:



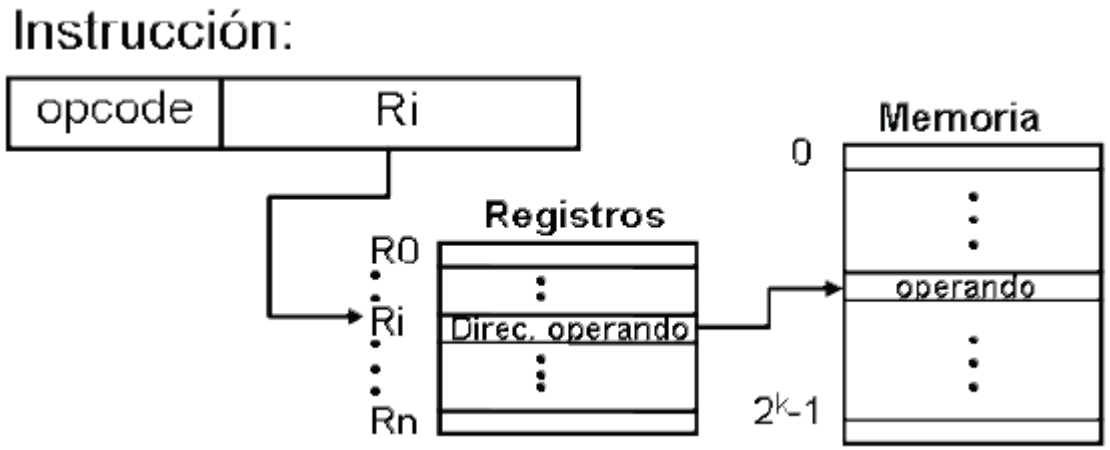
Directo registro

– El operando está contenido en un registro del procesador:



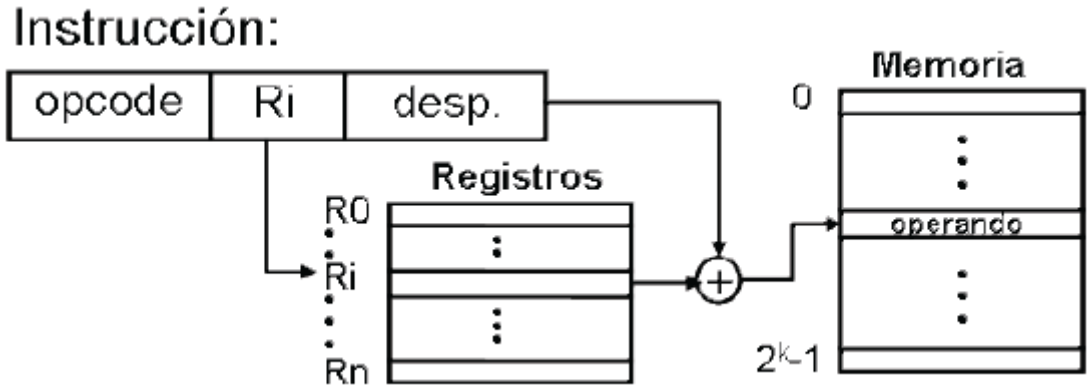
Indirecto registro

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



Indirecto registro con desplazamiento

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



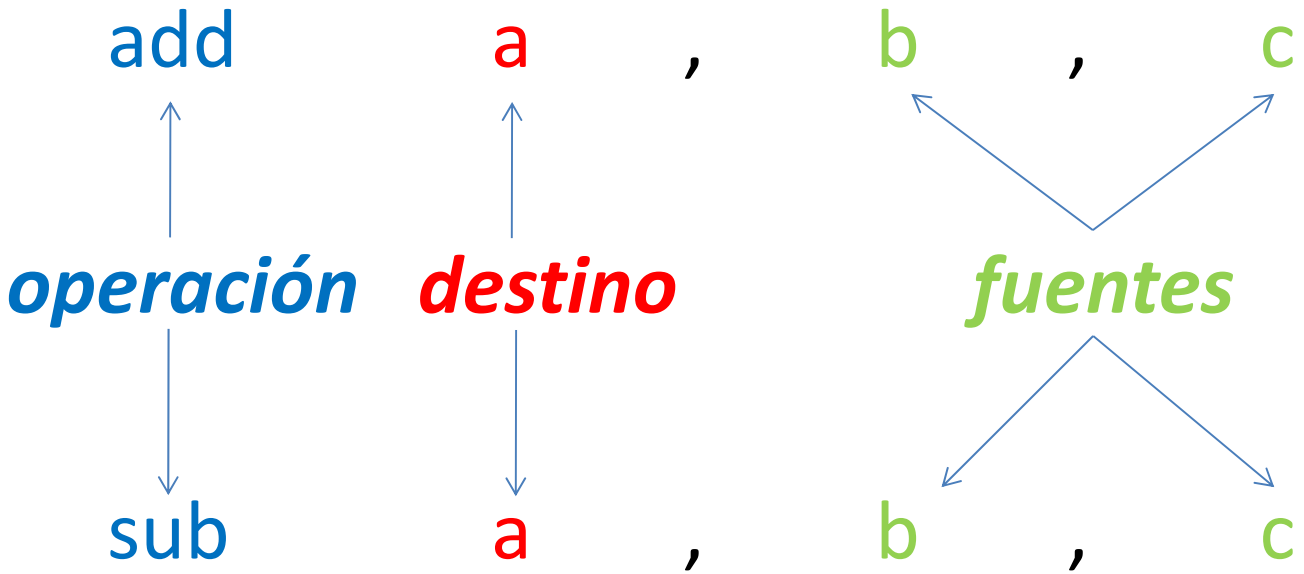
Instrucciones aritmético-lógicas

- 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		L. ENSAMBLADOR
C= A+B	←→	Instrucción de suma (add …)
C=A*B	←→	Instrucción de multiplicación (mul …)
C=A && B	←→	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 → **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 >

donde ***<Operando>*** es un **registro** (en cuyo caso empleamos *direccionamiento directo registro*) o un **inmediato** (en cuyo caso empleamos *direccionamiento inmediato*).

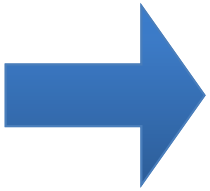
Instrucciones aritmético-lógicas

- 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 *direccionamiento 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 *direccionamiento 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	**0x00000032**

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 y B están en memoria)
C= A+B	←→	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 (C)

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 (*direccionamiento 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 (*direccionamiento indirecto registro con despl.*) a Rd.

  ldr Rd, [Rn,Rs]

  La instrucción copia el dato que hay en la posición de memoria indicada por Rn+Rs (*direccionamiento 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>

- Diversas formas para especificar esa dirección:

  str Rd, [Rn]

  La instrucción copia el dato que hay en el registro Rd en la posición de memoria indicada en el registro Rn (*direccionamiento 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 (*direccionamiento indirecto registro con despl.*)

  str Rd, [Rn,Rs]

  La instrucción copia el dato que hay en el registro Rd en la posición de memoria indicada por Rn+Rs (*direccionamiento indirecto registro con despl.*)

Ejemplos *load*

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

Registro	Contenido
r1	0x00000100
r2	0x0000F132
r3	0x00000000

ldr r3,[r1]

Registro	Contenido
r1	0x000000FA
r2	0x0000F132
r3	**0xAABBCCDD**

ldr r3,[r1,#8]

Registro	Contenido
r1	0x000000FA
r2	0x0000F132
r3	**0x00FF55EE**

Ejemplos *store*

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

Registro	Contenido
r1	0x00000100
r2	0x0000F132
r3	0x12345678

str r3,[r1]

str r3,[r1,#8]

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

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

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.
  - 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?
- Ejemplo: Inicializar a 0 las componentes de un vector de tres componentes.

Aspecto del programa en L. 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

Evitamos calcular el desplazamiento del "ldr"

El programa funciona igual sea cual sea la dirección X

Aspecto la memoria

Direc. Memoria	Contenido *	
X	Desconocido	Vector
X+4	Desconocido	
X+8	Desconocido	
X+12	mov r2, #0	Programa
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	

* En realidad está en binario

Instrucciones para toma de decisiones

- Diferencia calculadora – computador: El computador puede tomar decisiones → Instrucciones para la 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]
```

Condiciones

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

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 MayIgu
        add r1, r1, r2
        b Salir
MayIgu: sub r1, r1, r2
Salir:
```

Ejemplo construcción while

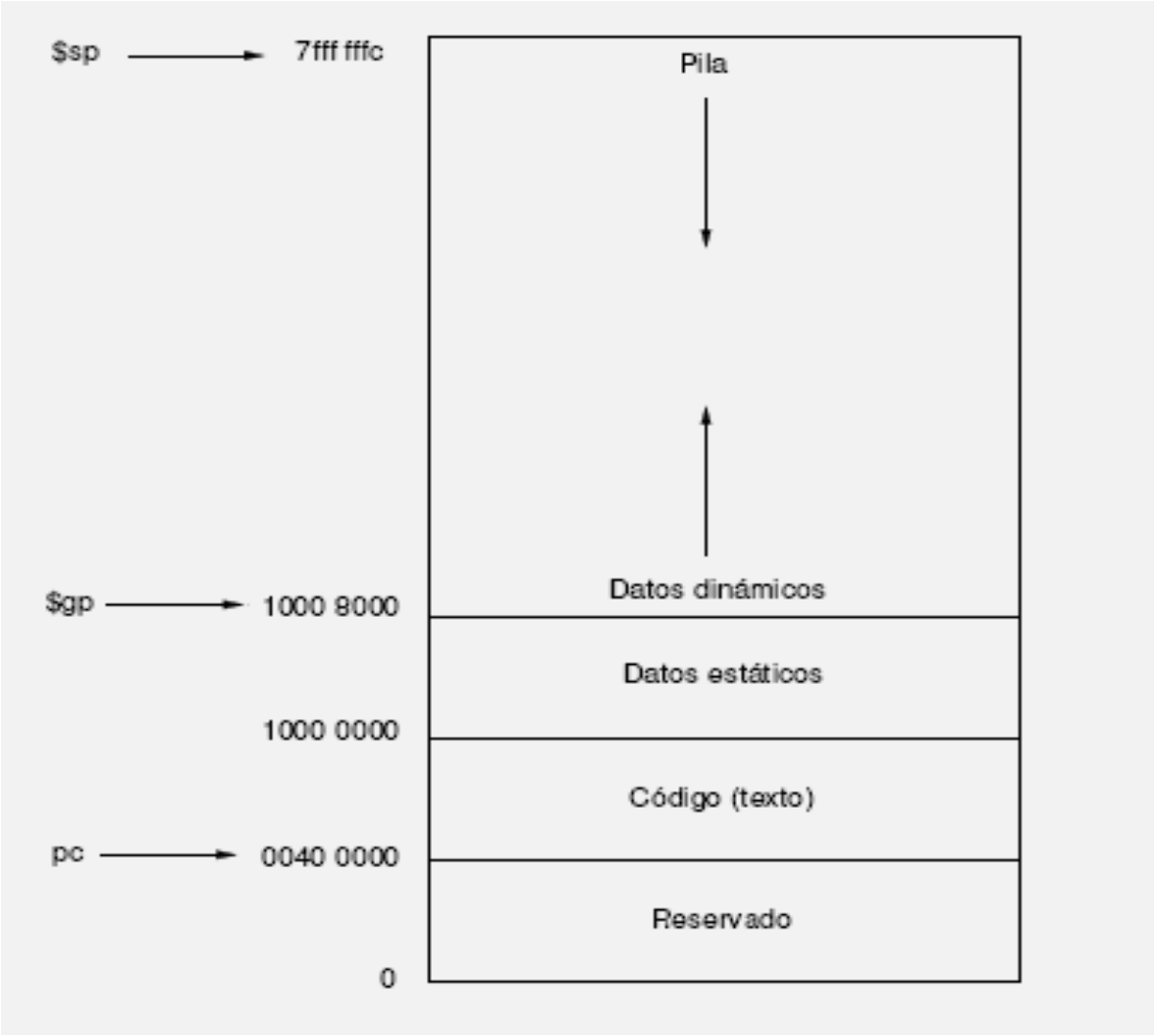
- Traducir la siguiente sentencia de C a ensamblador (suponer que A está en r1 y B en r2):

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:
```

Regiones de un programa



Desarrollo de código en 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:** todos aquellos caracteres seguidos de @. Pueden aparecer solos en una línea.

Desarrollo de código: directivas

- No son traducibles a instrucciones máquina

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

Ejemplo código

```
.global start
.data
X:   .word 0x03
Y:   .word 0x0A
.bss
Mayor: .space 4
.text
start:
            LDR R4, =X
            LDR R3, =Y
            LDR R5, =Mayor
            LDR R1, [R4]
            LDR R2, [R3]
            CMP R1, R2
            BLE else
            STR R1, [R5]
            B FIN
else:       STR R2, [R5]
FIN:        B .
.end
```

Llamadas a función (subrutinas)

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

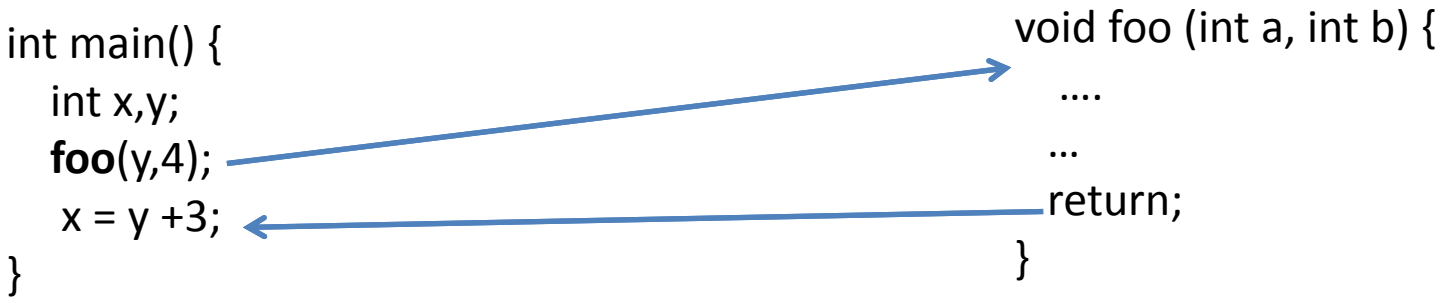
```
void foo (int a, int b) {
 ....
}

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

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

Llamadas a función: invocación

- ¿Cómo invocar una función?

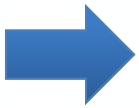


  - Salto incondicional al comienzo de la función *foo*
  - ¡¡ Pero necesitamos *recordar* la dirección a la que hay que volver tras ejecutar la función !!

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***

```
void foo (int a, int b) {
    .. = a+ b;
return;
}
int Main() {
    ....
   foo(x,y);
   x=x-y;
}
```



```
foo: …
        ADDr1,r2,r3
        …
        mov pc,lr        @ Cargo en PC la dirección de
                           retorno

Main: …
        BL foo  @Llamada a función. LR<- dir. de sub
        SUB r2,r2,r3
```

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*?

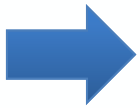
Llamadas a función: argumentos

- 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
```

Complicando las llamadas a función.

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

```
int main() {
   int x;
   …
    x=faa(3,4);
}
```

```
int faa (int a, int b) {
 int x,y;
  x = fuu(a);
  y = fuu(b);
  return x+y;
}
```

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

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

¿Qué pasa con las variables locales?

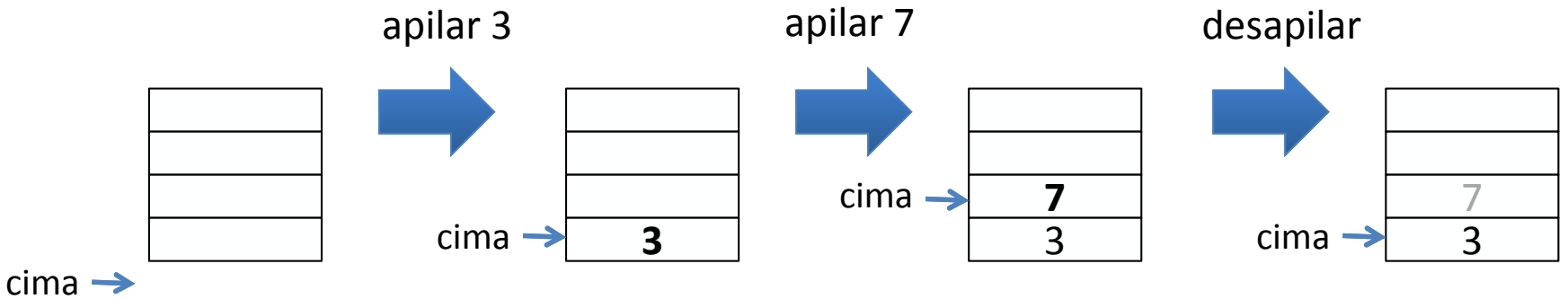
```
int a;
```
→ global

```
int faa(int c) {
   int x;
   x = foo(c,2);
   return a+c+x;
}
```
→ local

- 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)



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)
  - La pila "crece" de direcciones superiores a direcciones inferiores de memoria
- ¿Qué se almacena en la pila?
  - Argumentos de entrada: del quinto en adelante (en orden inverso)
  - Se deja espacio para albergar las variables locales
  - Registros cuyo contenido se debe preservar
    - SP debe mantener su valor anterior a la llamada
    - ¿otros?

Pasos ejecución subrutina

Función llamante

1. Situar parámetros en lugar accesible a la subrutina
   - Registros y pila si es necesario
2. Transferir el control a la subrutina (*BL*)

Función llamada

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 principal (*r0)*
6. Devolver el control al punto de 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
  - Estándar ARM: la *función llamada* DEBE preservar los registros *r4-r11* y 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…

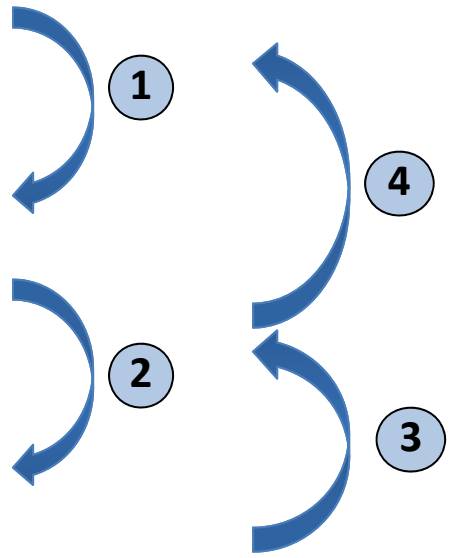
Salvado de registros: llamadas anidadas

- Se sigue el mismo criterio expuesto anteriormente pero…
  - ¿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**

Marco de pila

- 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 (frame pointer)

Esrtuctura de una rutina

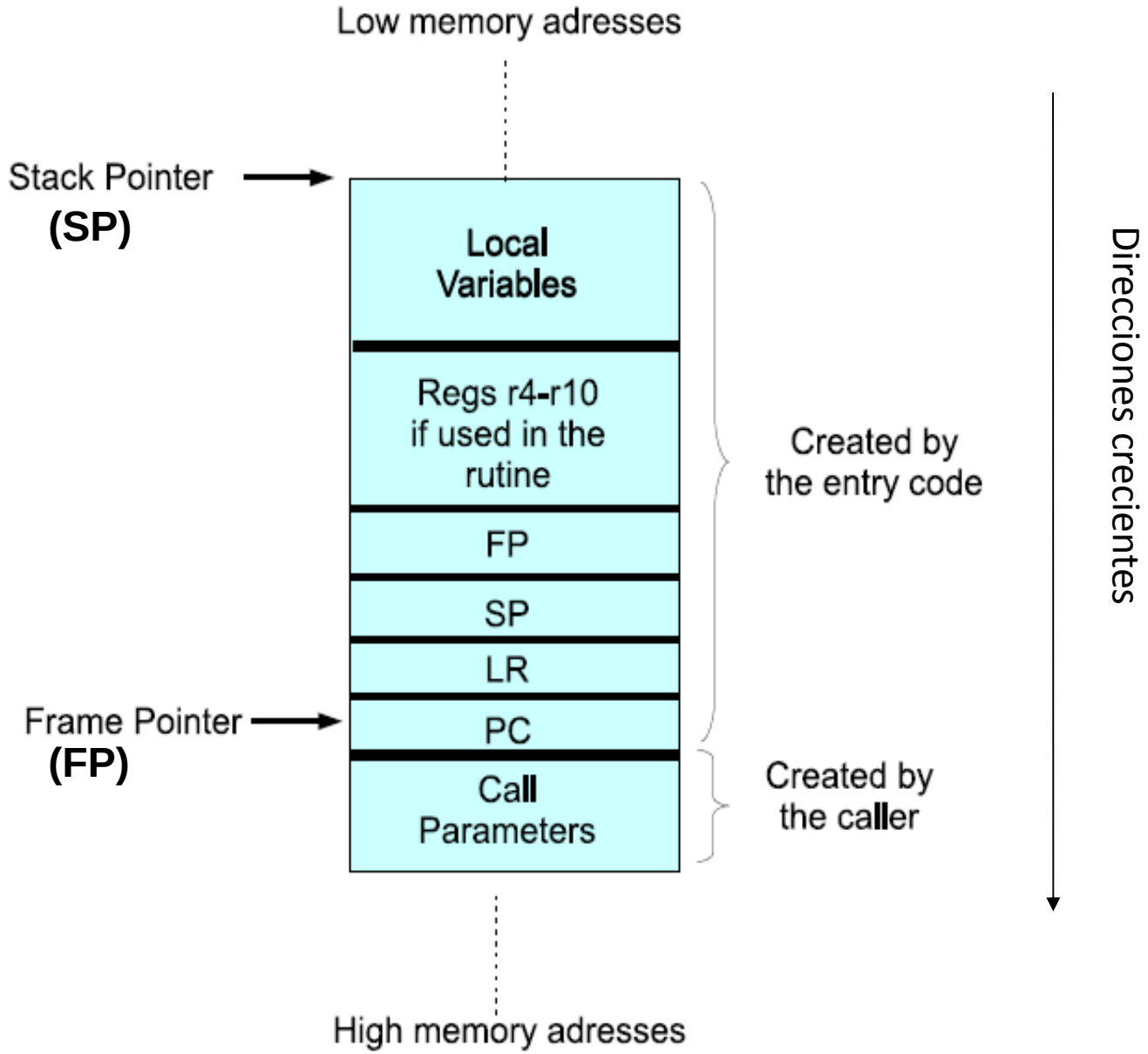
- Estructura de una rutina:

Código de entrada (prólogo) ← Construye el marco

Cuerpo de la rutina

Código de salida (epílogo) ← Destruye el marco y hace el retorno

Marco de pila: caso genérico



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 es no-hoja)
SUB     SP, SP, #SpaceForLocalVaribles @ 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, #SpaceForLocalVaribles  @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
```

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

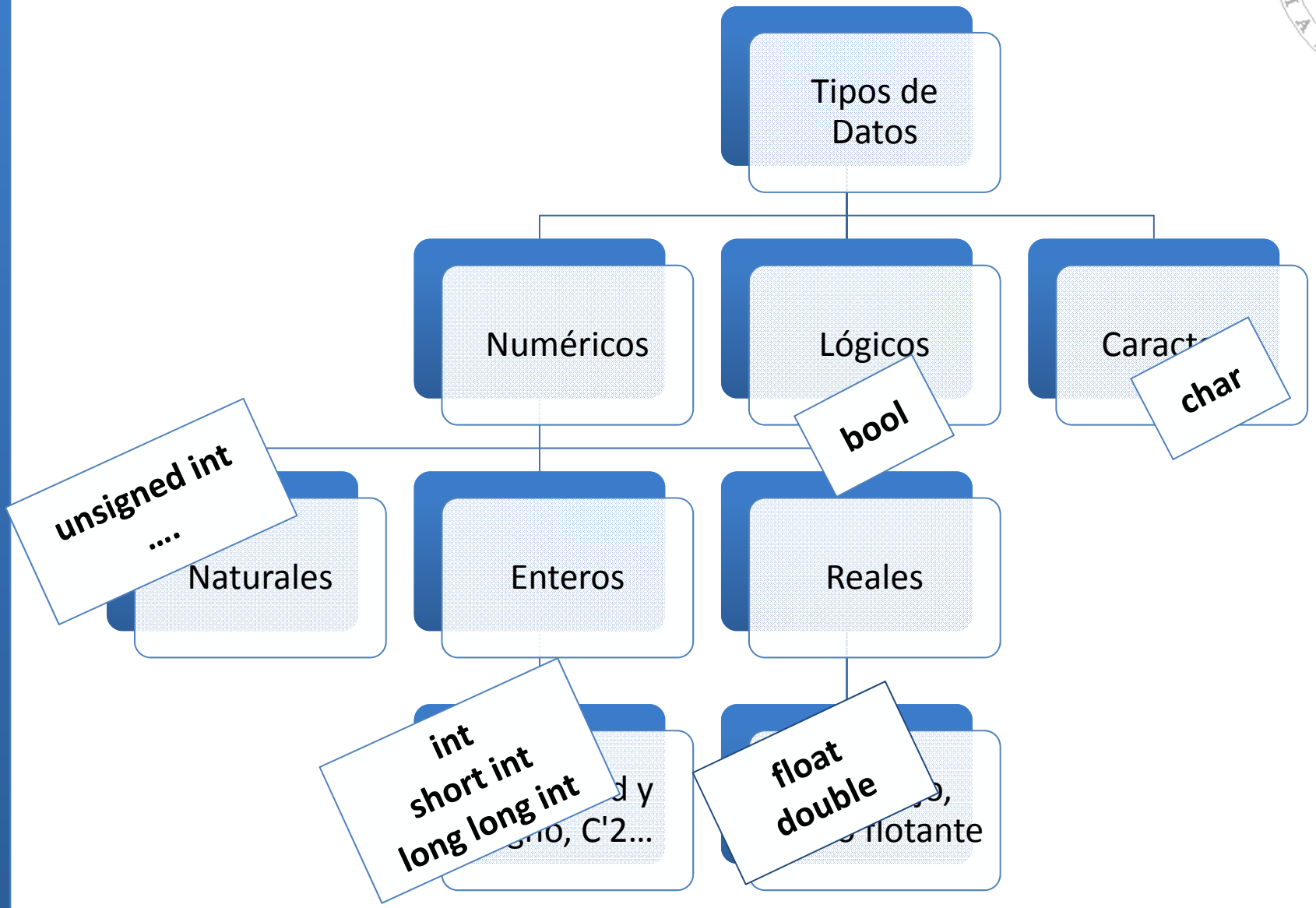
Variables locales y globales

- 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

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

Source: www.LookupTables.com

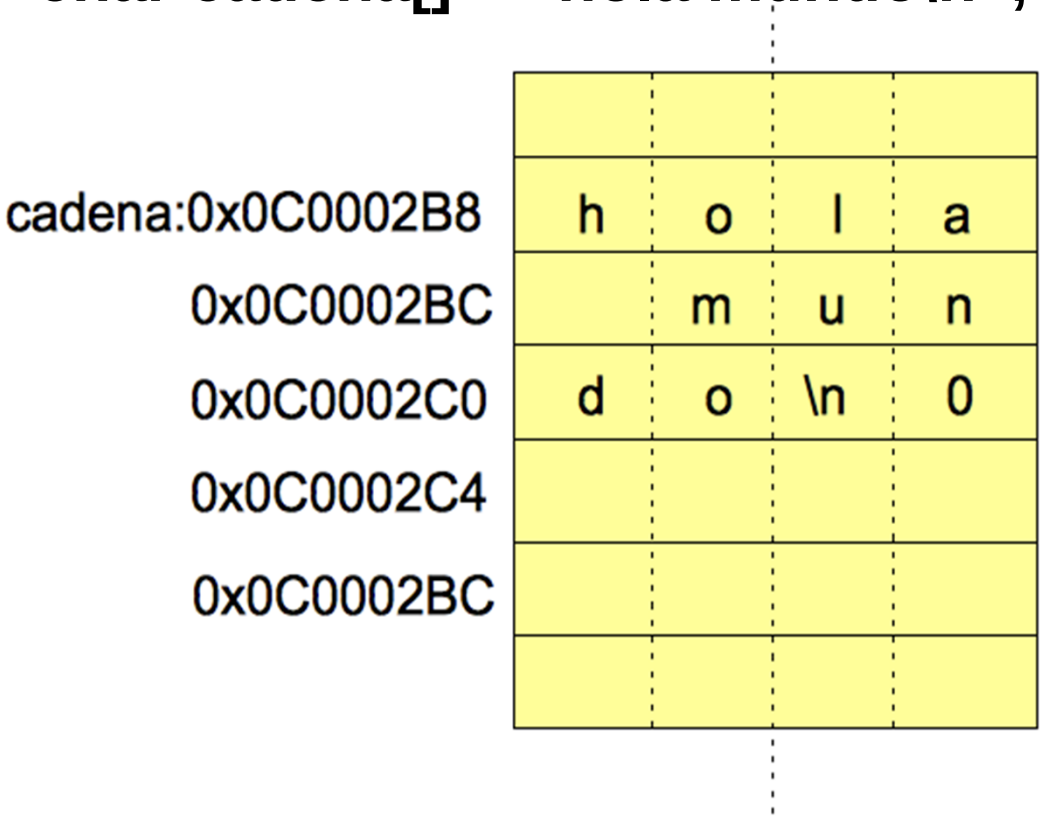
Codificación de enteros

- 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 representaciones 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;?

	Ejemplo (tamaño = 4 bits)		
	$b_3 b_2 b_1 b_0$	MS	C_2
POSITIVOS	0000	0	0
	0001	1	1
	0010	2	2
	0011	3	3
	0100	4	4
	0101	5	5
	0110	6	6
	0111	7	7
NEGATIVOS	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”;

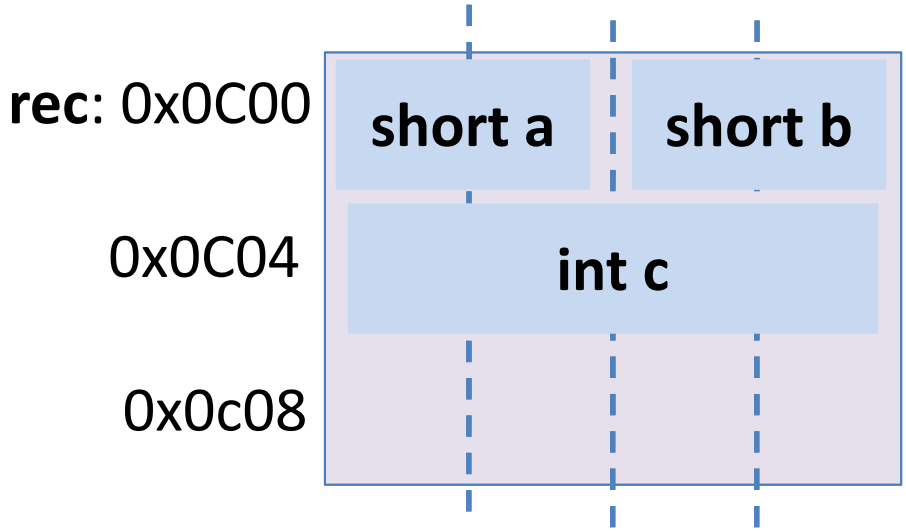


- Los elementos de un array ocupan posiciones consecutivas de memoria
- Las cadenas de caracteres en C acaban en \0
- ¿En qué dirección está v[16] del array *int vector[100];* si su primer elemento está en la dirección 0x0C00?

Estructuras

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

struct mistruct rec;
```



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