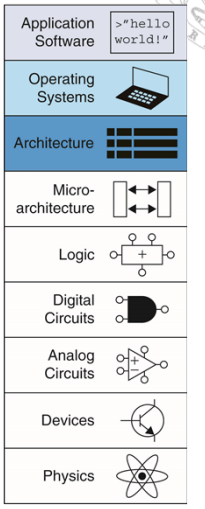
Módulo 8: Lenguaje Máquina y Ensamblador

ARM

fc²

Nivel de Abstracción

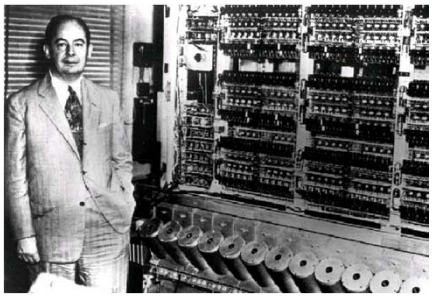
- **Architecture: programmer's view of computer**
  - Defined by instructions & operand locations
- Microarchitecture: how to implement an architecture in hardware



fc²

John von Neumann zu Margitta

- Fue un matemático húngaro-estadounidense
- Realizó contribuciones importantes en física cuántica, análisis funcional, teoría de conjuntos, informática, economía, análisis numérico, estadística …
- Fue pionero de la computadora digital moderna introduciendo el concepto de **programa almacenado en memoria.**
  - Los programas almacenados dieron a las computadoras flexibilidad y confiabilidad, haciéndolas más rápidas y menos sujetas a errores que los programas mecánicos.
  - Además se podían crear programas que escribieran en la memoria otros programas.



fc²

3

¿Cómo se genera un programa?

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

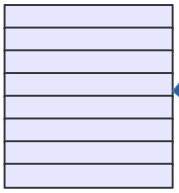
```
simple:
   leal   4(%esp), %ecx
   andl   $-16, %esp
   pushl  -4(%ecx)
   pushl  %ebp
   movl   %esp, %ebp
   pushl  %ecx
   subl   $16, %esp
   movl   $5, -16(%ebp)
   movl   $3, -
12(%ebp)…….
```

```
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
```

Table eC.1 Languages at roughly decreasing levels of abstraction

Language	Description
Matlab	Designed to facilitate heavy use of math functions
Perl	Designed for scripting
Python	Designed to emphasize code readability
Java	Designed to run securely on any computer
C	Designed for flexibility and overall system access, including device drivers
Assembly Language	Human-readable machine language
Machine Language	Binary representation of a program

Memoria



fc²

4

¿Cómo funciona un computador?

i) El usuario hace doble click sobre el icono del programa que quiere ejecutar

DATOS PROGRAMA

ii) El sistema operativo busca el programa en las unidades de almacenamiento

RAM

DIRECCIÓN DE INICIO DEL PROGRAMA

DATOS PROGRAMA

iii) El sistema operativo almacena el programa y sus datos en memoria RAM

fc²

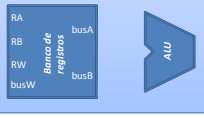
5

¿Cómo funciona un computador?

DATOS PROGRAMA

CPU

PC: DIRECCIÓN INICIO PROGRAMA

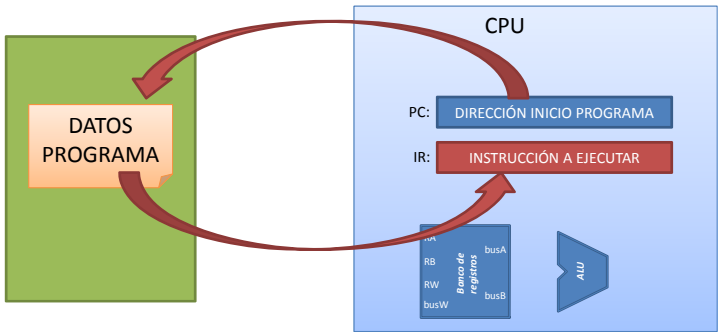


iv) El sistema operativo indica a la CPU dónde se encuentra la dirección de la primera instrucción del programa que tiene que ejecutar

fc²

6

¿Cómo funciona un computador?



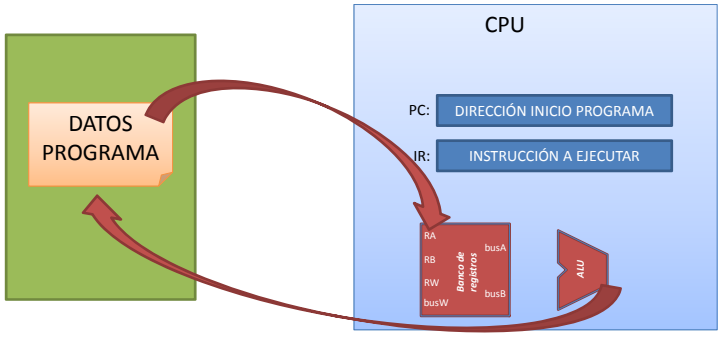
v) La CPU busca la instrucción que tiene que ejecutar

Qué tiene que hacer
Dónde están los datos
Dónde se escribe el resultado

fc²

7

¿Cómo funciona un computador?



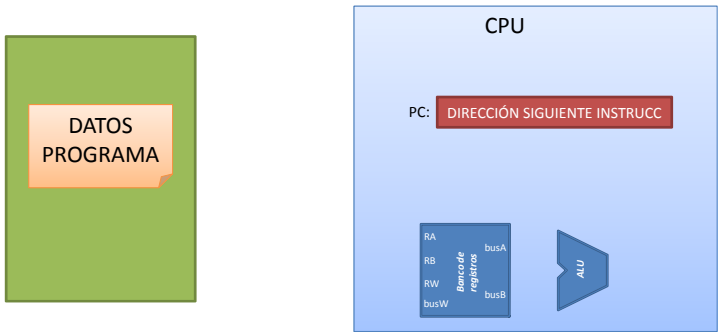
vi) La CPU ejecuta la instrucción

Leer los datos de entrada
Realizar la operación
Escribir el resultado

fc²

8

¿Cómo funciona un computador?



vii) La CPU calcula automáticamente dónde se encuentra la siguiente instrucción del programa

fc²

9

¿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. Desarrollo de Sistemas Operativos y Compiladores.

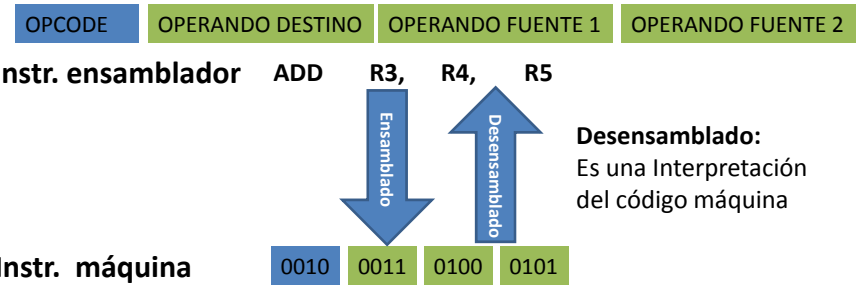
Aprender la capa de abstracción de Arquitectura. Partiendo de esta capa comprenderemos la conexión entre programa y hardware.

fc²

10

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 a código máquina:



fc²

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

fc²

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.
- 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

fc²

13

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, 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
.text                  @Program
start:  MOV R0, #ONE
        LDR R1, =MYVAR
        LDR R2, [R1]
        ADD R3, R0, R2
        LDR R4, =RES
        STR R3, [R4]
END:    B .
.end
```

fc²

ARM

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

fc²

15

Los procesadores ARM

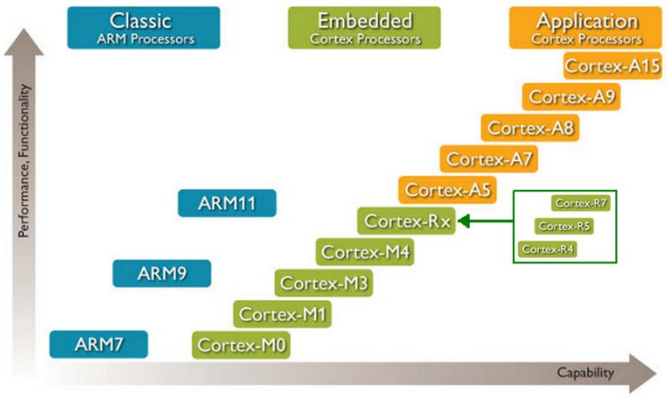
- Unos de los más vendidos/empleados en el mundo
  - 75% del mercado de CPU empotrados de 32-bits
- Usados especialmente en dispositivos portátiles debido a su bajo consumo y razonable rendimiento (MIPS/Watt)
- Disponibles como hard/soft core
  - Fácil integración en Systems-On-Chip (SoC)
- Ofrecen diversas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

fc²

16

Familias de procesadores ARM

- Familia: Procesadores con la misma arquitectura (compatibilidad binaria), pero distinta implementación



fc²

17

Arquitectura del procesador

- Repertorio de instrucciones
  - Operaciones que se pueden realizar
  - 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 (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

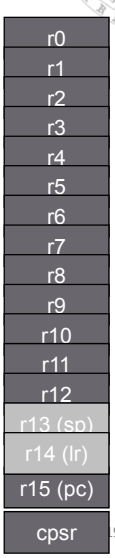
fc²

18

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 estado actual del programa (CPSR)
- Permite acceder a la vez a tres registros:
  - 2 registros fuente
  - 1 registro destino
- ¡Poca capacidad pero muy rápidos!

add r5,r2,r1



fc²

19

Ejemplo

- Queremos contar el número de ceros que contiene un vector de 1000 elementos .

```
ceros=0;
for(i=0;i<1000;i++){
        if(A[i]==0){
                ceros=ceros+1;
        }
}
```

- Necesitamos una memoria para manejar 1000 elementos.
- ¿Qué debe hacer el computador en cada iteración del bucle?

fc²

20

La Memoria

- Una secuencia (tabla) de bytes
  - Cada ***byte*** tiene asignada una **dirección** de memoria
  - **Profundidad**: Si disponemos de ***k* bits** para la dirección podremos acceder a 2^k bytes
    - Si k=10 -> Tamaño= 2^{10} bytes = 1024 bytes = 1Kbyte
    - Si k=6 -> Tamaño= 2^6 bytes = 64 bytes
  - **Anchura:** Una **palabra** es una cadena finita de bits, típicamente 8, 16, 32 o 64 bits, que son manejados como un conjunto por la máquina, es un parámetro arquitectónico.

21

fc²

Alineamiento de la memoria ARM

- Palabra de 32 bits (4 bytes)
  - En este curso accederemos casi siempre a nivel de palabra
- Memoria ARM direccionable por bytes.
- Pero con accesos alineados
  - Palabra de 4 bytes (instrucciones, int, float)
    -> Dirección múltiplo de 4
  - Dato de tamaño 2 bytes (short int)
    -> Direcciones múltiplos de 2
  - Data de tamaño byte (char)
    -> Cualquier dirección
- Las restricciones de alineamiento :
  - Desaprovechan memoria,
  - pero aceleran notablemente el acceso.
  - Lo utilizan la mayoría de las arquitecturas

22

fc²

Organización del contenido de memoria

- Es habitual en entornos de desarrollo visualizar la memoria a nivel de palabra.
  - Ejemplo: palabras $AABBCCDD_{HEX}$ y $9070FFAA_{HEX}$ 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 es MIDDLE-ENDIAN
  - admite ambas organizaciones

fc²

23

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

fc²

24

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
    - …

fc²

25

Inmediato

- El operando está contenido en la propia instrucción:

Instrucción:

opcode	A

operando = A

```
MOV R0, #0
```

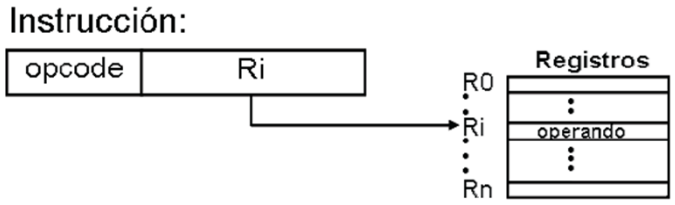
- Sirve para manejar constantes.

fc²

26

Directo registro

– El operando está contenido en un registro del procesador:



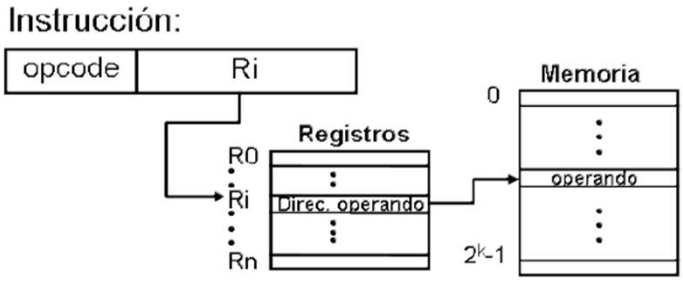
SUB R2, R2, R3

fc²

27

Indirecto registro

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



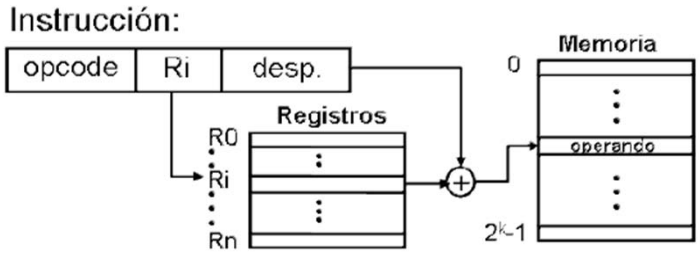
LDR R2, [R1]

fc²

28

Indirecto registro con desplazamiento

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



LDR r1, [pc, #16]

fc²

Instrucciones aritmético-lógicas

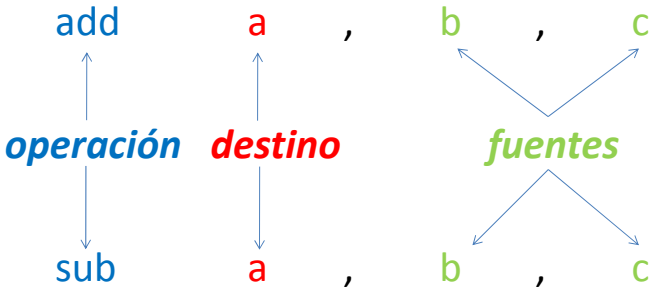
- Operaciones aritméticas y lógicas → Muy comunes en cualquier programa de alto nivel
- Todo computador debe ser capaz de realizar las operaciones aritméticas y lógicas básicas:
  - Aritméticas: SUMA, RESTA, ETC.
  - Lógicas: AND, OR, ETC.
- Equivalencia directa con lenguaje de alto nivel:

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 …)

fc²

Formato inst. aritmético-lógicas

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



fc²

31

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>*** de forma simplificada es:
  - un **registro** (empleamos *direccionamiento directo registro*)
  - un **inmediato** (empleamos *direccionamiento inmediato*).

fc²

32

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*.

fc²

33

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**

fc²

34

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 →
- 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

fc²

35

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.

fc²

36

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.*)

fc²

37

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	0x00000100
r2	0x0000F132
r3	**0xAABBCCDD**

ldr r3,[r1,#8]

Registro	Contenido
r1	0x00000100
r2	0x0000F132
r3	**0x00FF55EE**

fc²

38

Ejemplos *store*

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

Registro	Contenido
r1	0x00000100
r2	0x0000F132
r3	0x12345678

str r3,[r1]

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

str r3,[r1,#8]

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

fc²

39

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

```
int a = 12;
```

```
a:   .word 12
```

La variable a está almacenada en la dirección de memoria 1024.
Se etiqueta la dirección como a

1020	…
a=1024	0x0000000C
1028	…
1032	…

```
ldr r5, a
```

R5 = 12

fc²

Como acceder a un array

```
int a[10];

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

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

a = 1020	0x00000001
1024	0x00000003
1028	0x00000007
1032	0x00000005
1036	0xFFFFFFFE
…	…
1066	0x00000004

```
ldr r5, a
```

→ R5 = 1

¿Y si quiero acceder a otra componente?

fc²

Como acceder a un array

¿Dónde está la componente 3? **1032**

```
ldr r1, =a
```

→ R1 = 1020

a = 1020	0x00000001
1024	0x00000003
1028	0x00000007
1032	0x00000005
1036	0xFFFFFFFE
…	…
1066	0x00000004

```
add r2, r1, #12
```

→ R2 = 1032

```
ldr r5, [r2]
```

→ R5 = 5

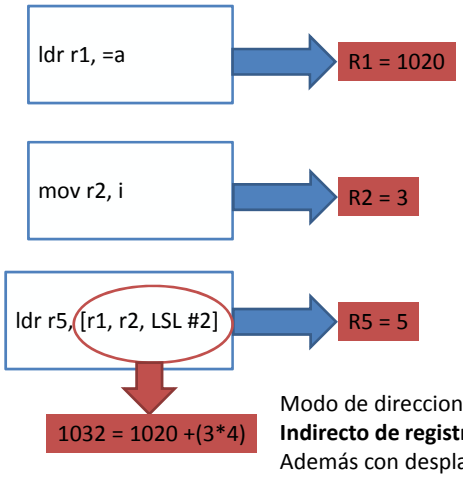
```
ldr r5, [r1, #12]
```

→ R5 = 5

fc²

Como acceder a un array

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



a = 1020	0x00000001
1024	0x00000003
1028	0x00000007
1032	0x00000005
1036	0xFFFFFFFE
…	…
1066	0x00000004

Modo de direccionamiento del ARM:
Indirecto de registro con desplazamiento por registro.
Además con desplazamiento (lógico a la izquierda).

fc²

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 inst. salto

fc²

44

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

fc²

Instrucciones de salto condicional

- Formato:

  cmp Rn, <Operando>

  bXX Desplazamiento

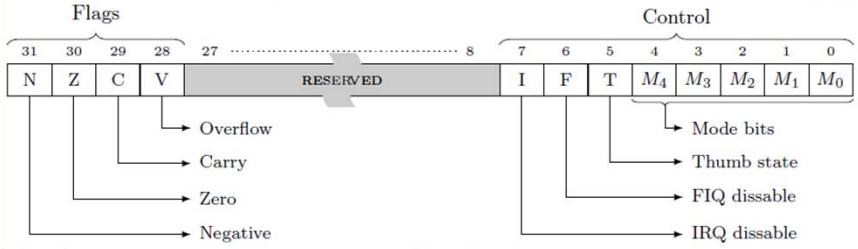
  Dependiendo de cuál sea el resultado de la condición XX evaluada sobre Rn y Operando, se ejecuta tras el salto la siguiente instrucción en el orden secuencial o bien la situada en la posición resultante de saltar un número de instrucciones igual al *Desplazamiento*

- 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]
```

fc²

Registro de Estado



Bit	Significado
N	Indica si la última operación dio como resultado un valor negativo (N = 1) o positivo (N = 0).
Z	Se activa (Z = 1) si el resultado de la última operación fue cero, de lo contrario permanece inactivo (Z = 0).
C	Su valor depende del tipo de operación: ▪ Para una suma o una comparación (CMP), C = 1 si hubo *carry*. ▪ Para las operaciones de desplazamiento, toma el valor del bit saliente.
V	En el caso de una suma o una resta, V = 1 indica que hubo un *overflow*.

fc²

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	

fc²

48

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.

fc²

49

Ejemplo construcción if-then-else

- Traducir la siguiente sentencia de C a ensamblador :

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

```
A: …
B: …
        ldr R1, A
        ldr R2, B
        cmp r1, r2
        bge Maylgu
        add r1, r1, r2
        b Salir
Maylgu: sub r1, r1, r2
Salir:  str R1, A
```

fc²

50

Ejemplo construcción while

- Traducir la siguiente sentencia de C a ensamblador :

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

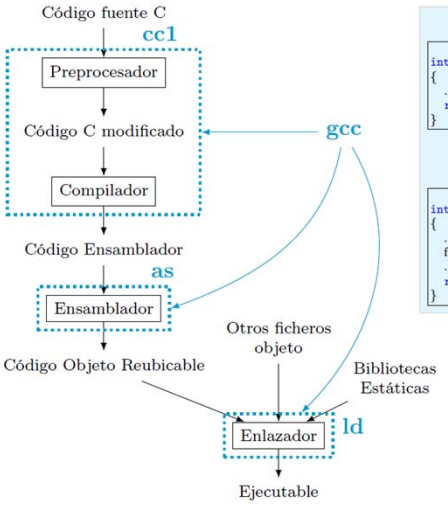
```
array:   Componentes del vector
…
         mov r2, #0
         ldr r3, =array
LOOP:    cmp r2, #10
         bhs Salir
         ldr r4, [r3]
         add r4, r4, r2
         str r4, [r3]
         add r2, r2, #1
         add r3, r3, #4
         b LOOP
Salir:
```

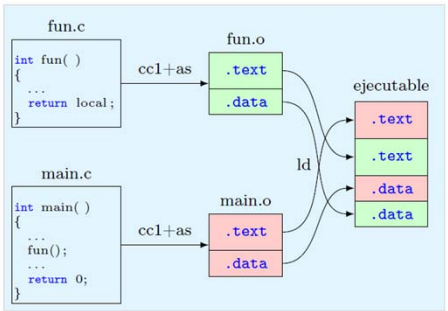
fc²

51

Desarrollo de código ARM

- Etapas de compilación, ensamblado y enlazado





fc²

--

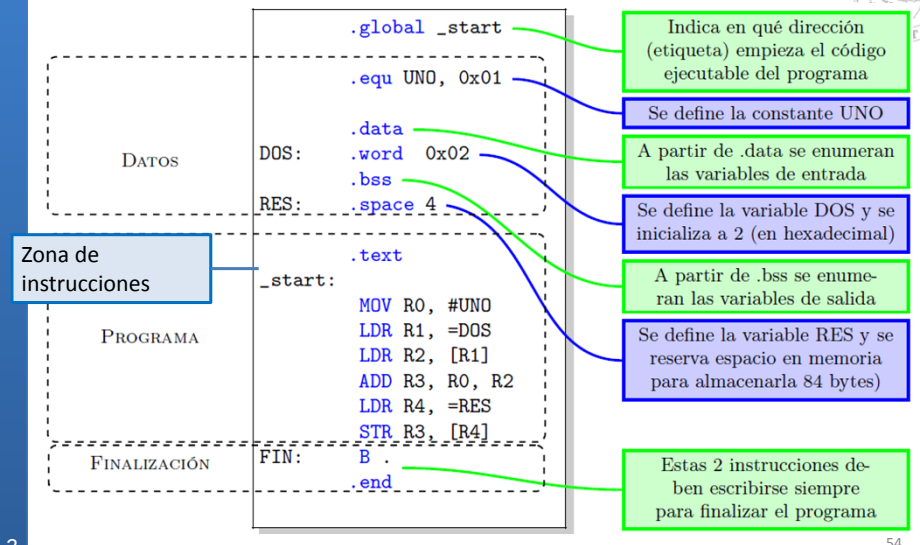
Desarrollo de código ARM

- **Directivas** para desenrollar el código en de memoria

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)

fc²

Zonas de un programa ensamblador

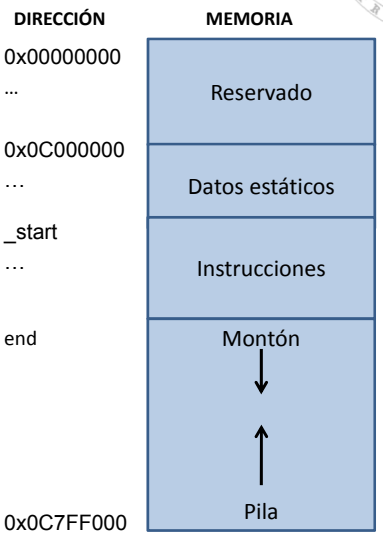


fc²

Script de enlazado

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

```
SECTIONS
{
. = 0x0C000000;
.data : {
*(.data)
*(.rodata)
}
.bss : {
*(.bss)
*(COMMON)
}
.text : {
*(.text)
}
PROVIDE(end = .);
PROVIDE(_stack = 0x0C7FF000 );
}
```



fc²

55

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.

fc²

56

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]
```

Evitamos calcular el desplazamiento del "ldr"

Traducción a L. Máquina

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

fc2

57

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;
}
```

fc2

58

¿Qué pasa con las variables locales?

```
int a;                    → global

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

- 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?

fc²

59

Llamadas a función: invocación

- ¿Cómo invocar una función?

```
int main() {
    int x,y;
    foo(y,4);
    x = y +3;
}
```

```
void foo (int a, int b) {
    …
    …
    return;
}
```

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

fc²

60

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: …
        ADD r1,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
```

fc²

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*?

fc²

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?

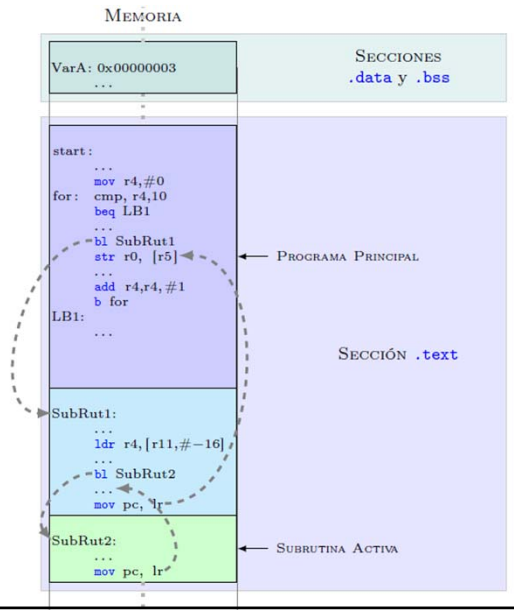
```
int main() {
    int x;
    …
     x=faa(3,4);
}
```

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

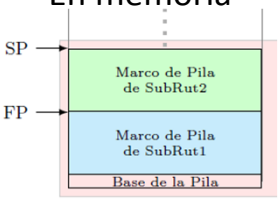
```
int fuu (int a) {
  int x;
    …
   return x;
}
```

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

Llamadas a subrutinas



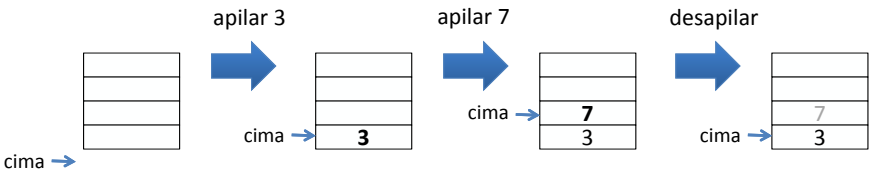
- Para resolver:
  - Preservar Ret.
  - Variables locales
  - Paso de parám.
  - Preservar Reg.
- Se utiliza la PILA
  - En memoria



fc2 65

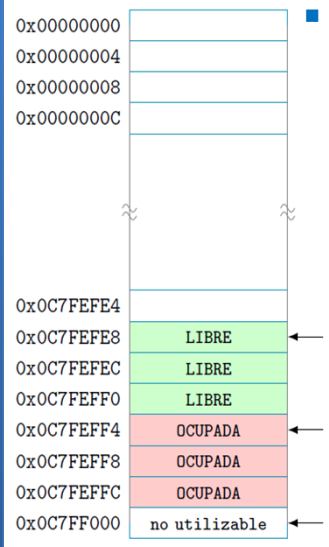
Llamadas a función: la pila

- 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)



fc2 66

Llamadas a función: la pila



- La pila en ARM: el registro *SP*
  - En el script de enlazado se inicializa la base de la pila.

    `PROVIDE(_stack = 0x0C7FF000 );`
  - El registro R13 (*SP -> stack pointer*) dirección de la cima de la pila (última posición ocupada)
  - "Full Descending": La pila "crece" de direcciones superiores a direcciones inferiores de memoria
- En la pila se almacenan:
  - Argumentos de entrada (del 5º en adelante).
  - Las variables locales.
  - Registros que se deben preservar.

fc²

Preservar 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…

fc²

68

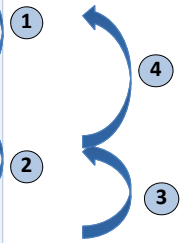
Salvado de registros: llamadas anidadas

- Siguiendo el AAPCS, ¿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.

fc²

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

fc²

Pasos ejecución subrutina

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

Función llamante

3. Prólogo (construye el marco)
   - Reservar espacio para var. locales y registros
   - Preservar información
4. Cuerpo
   - Ejecutar las tareas propias de la subrutina
   - Situar resultado en r0
5. Epílogo (destruye el marco)
   - Restaura información
   - Libera espacio de var. locales y registros
6. Devolver el control al punto de llamada (MOV PC,*BL*)

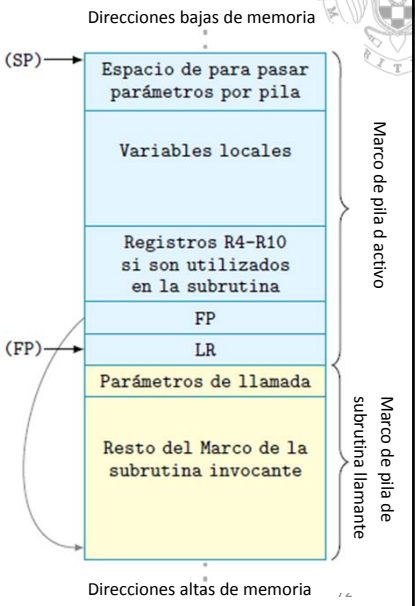
Función llamada

fc²

71

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).



fc²

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, #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
```

fc2

73

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=0x0C7F EFF4; FP= 0x0C7F EFFC
LR=0x0C00 2000; R4=3; R5=9

	Dirección	Contenido
	0x0C7F EFD8	
	0x0C7F EFDC	
	0x0C7F EFE0	
	0x0C7F EFE4	
	0x0C7F EFE8	
	0x0C7F EFEC	
	0x0C7F EFF0	
SP →	0x0C7F EFF4	
	0x0C7F EFF8	
FP →	0x0C7F EFFC	

Pila después del prologo

SP=0x0C7F EFD8 FP= 0x0C7F EFF0
LR=0x0C00 2000 R4=3 R5=9

	Dirección	Contenido
	0x0C7F EFD8	
SP →	0x0C7F EFDC	Space for Local_var 2
	0x0C7F EFE0	Space for Local_var 1
	0x0C7F EFE4	3
	0x0C7F EFE8	9
	0x0C7F EFEC	0x0C7F EFFC
FP →	0x0C7F EFF0	0x0C00 2000
	0x0C7F EFF4	
	0x0C7F EFF8	
	0x0C7F EFFC	

fc2

74

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

fc2

75

Load y Store Múltiple

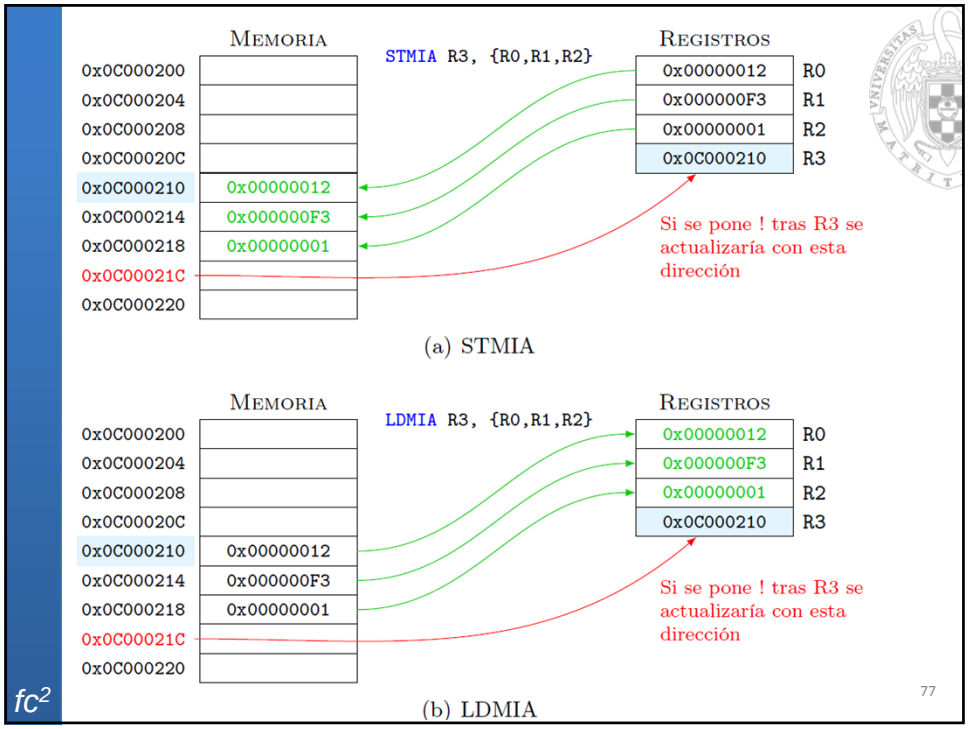
- La arquitectura ARM ofrece instrucciones de load y store múltiple.
- Permiten la carga/guarda simultánea de varios registros en posiciones de memoria consecutivas
- Donde:

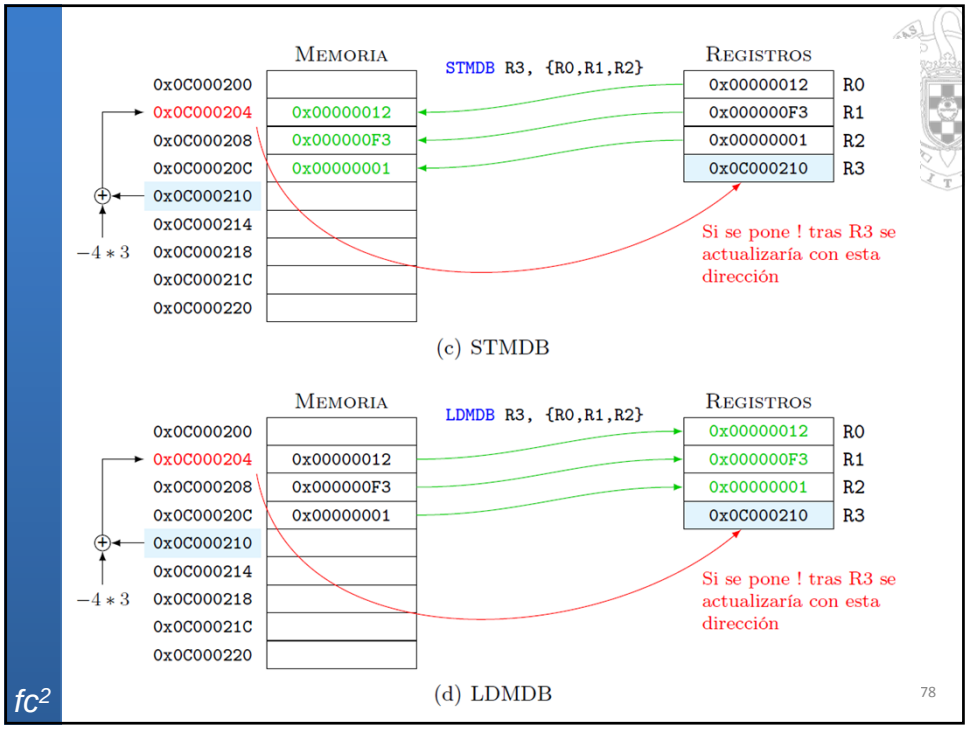
```
LDM<Modo> Rb[!], {lista de registros}
STM<Modo> Rb[!], {lista de registros}
```

  - **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.

fc2

76


MEMORIA
STMIA R3, {R0,R1,R2}
REGISTROS
0x0C000200
0x0C000204
0x0C000208
0x0C00020C
0x0C000210
0x0C000214
0x0C000218
0x0C00021C
0x0C000220
0x00000012
0x000000F3
0x00000001
0x00000012 R0
0x000000F3 R1
0x00000001 R2
0x0C000210 R3
Si se pone ! tras R3 se actualizaría con esta dirección
(a) STMIA
MEMORIA
LDMIA R3, {R0,R1,R2}
REGISTROS
Si se pone ! tras R3 se actualizaría con esta dirección
(b) LDMIA
fc2
77


MEMORIA
STMDB R3, {R0,R1,R2}
REGISTROS
0x0C000200
0x0C000204
0x0C000208
0x0C00020C
0x0C000210
0x0C000214
0x0C000218
0x0C00021C
0x0C000220
−4 ∗ 3
0x00000012
0x000000F3
0x00000001
0x00000012 R0
0x000000F3 R1
0x00000001 R2
0x0C000210 R3
Si se pone ! tras R3 se actualizaría con esta dirección
(c) STMDB
MEMORIA
LDMDB R3, {R0,R1,R2}
REGISTROS
Si se pone ! tras R3 se actualizaría con esta dirección
(d) LDMDB
fc2
78

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}
    - STMFD 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

fc2

79

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
```

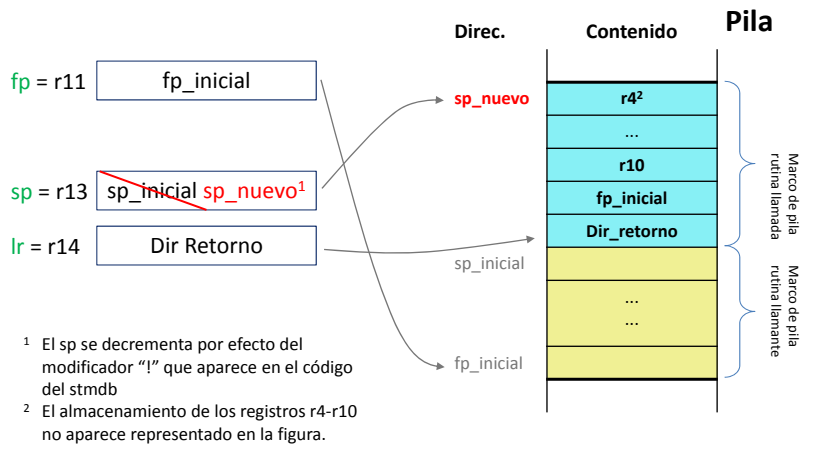
- Existen algunos casos especiales en los que podemos simplificar este prólogo:
  - Si no pasamos más de cuatro parámetros a ninguna subrutina no tendremos que reservar espacio extra. Podemos entonces eliminar la tercera instrucción.
  - Si sólo apilamos un registro (sucedería en una subrutina hoja que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

fc2

80

El prólogo paso a paso (1)

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



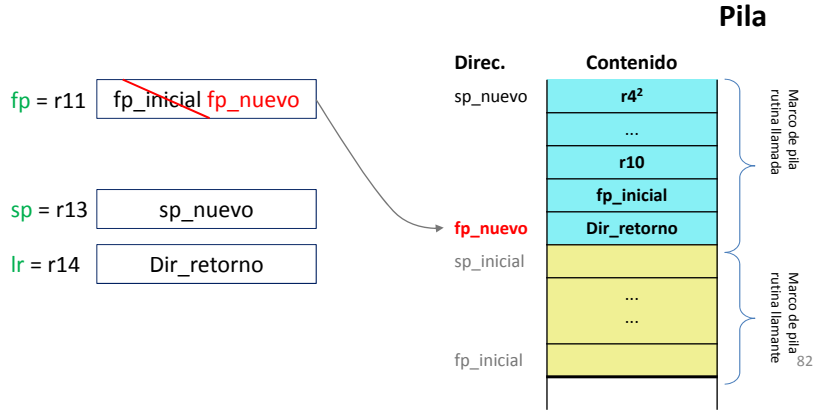
[1] El sp se decrementa por efecto del modificador "!" que aparece en el código del stmdb
[2] El almacenamiento de los registros r4-r10 no aparece representado en la figura.

81

fc²

El prólogo paso a paso (2)

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

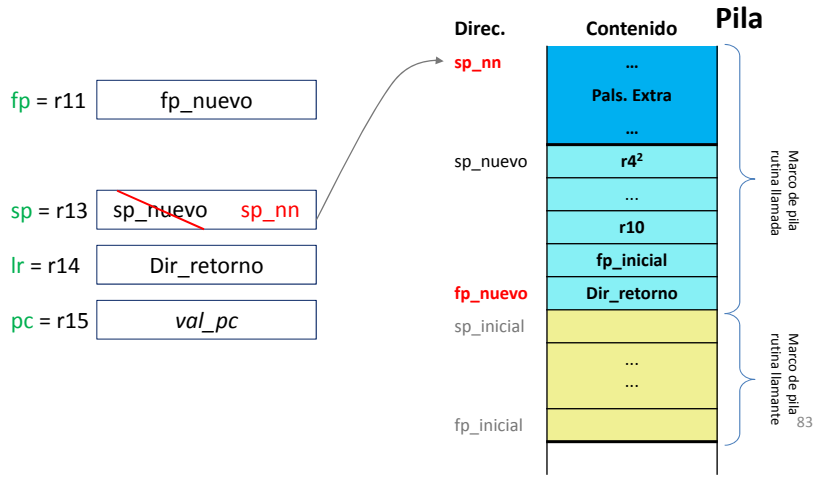


82

fc²

El prólogo paso a paso (3)

- SUB SP, SP, #4*NumPalabrasExtra



fc2

Marco de pila: epílogo generalista

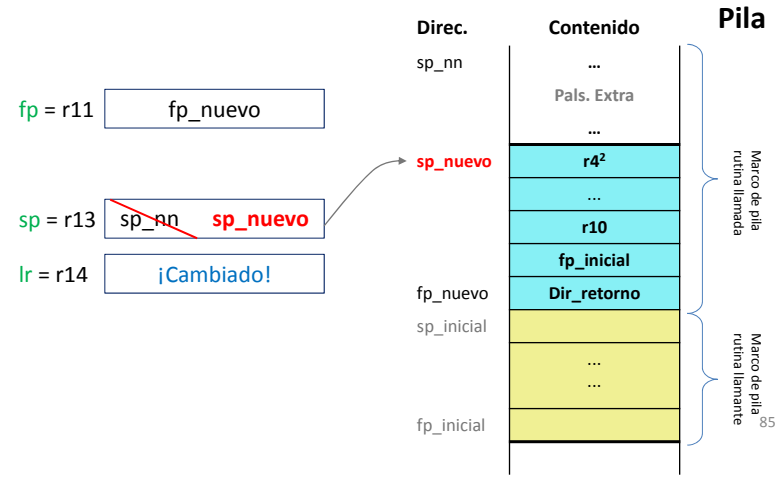
```
SUB SP, FP, #(4*NumRegistrosApilados-4) @ SP   dirección del 1er registro apilado
                                        @ Se descartan las variables locales y argumentos
POP {R4-R10,FP,LR}                      @ Restaura los registros desde la pila
BX LR                                   @ Retorno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

fc2

El epílogo paso (1)

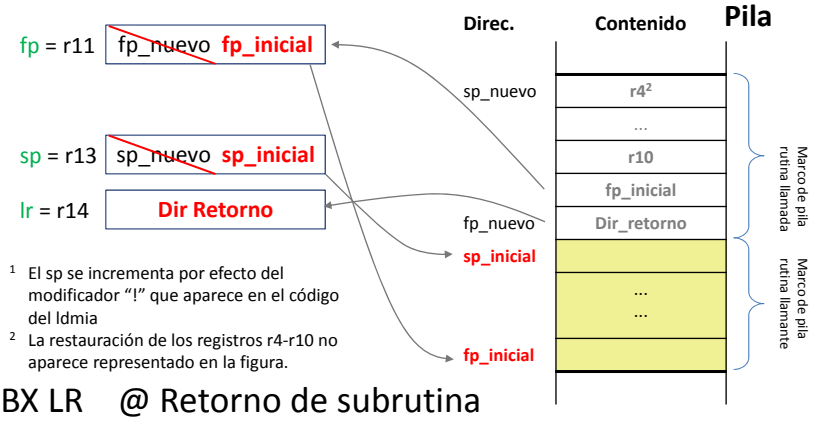
- SUB SP, FP, #(4*NumRegistrosApilados-4)



fc2

El epílogo paso a paso (2 y 3)

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



[1] El sp se incrementa por efecto del modificador "!" que aparece en el código del ldmia

[2] La restauración de los registros r4-r10 no aparece representado en la figura.

- BX LR @ Retorno de subrutina

pc = r15 | ~~Cambiado~~ *Dir_retorno*

fc2

C Real Time 0

- Para que la función *main* pueda comenzar, el sistema debe haber inicializado el registro de pila (SP) con la dirección base de la pila.
- En nuestro caso estamos utilizando herramientas de compilación para un sistema *bare metal*, es decir, sin sistema operativo.
- En este caso, debemos proporcionar nosotros el código de arranque.

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

fc²

87

Variables locales en funciones

```
int funA( int a1, int a2, int a3, int a4, int a5);
```

- La función tendrá, al menos, 5 variables locales llamadas a1-a5 respectivamente.
  - Si *funA* es una función recursiva, tendremos varias instancias activas de *funA*.
  - Cada instancias debe tener su propia variable *a1-a5*, accesible sólo desde esa instancia.
- Necesitamos por tanto un espacio privado para cada instancia de la función, donde podamos alojar estas variables.
- El espacio más natural para ello es el marco de activación de la subrutina.

fc²

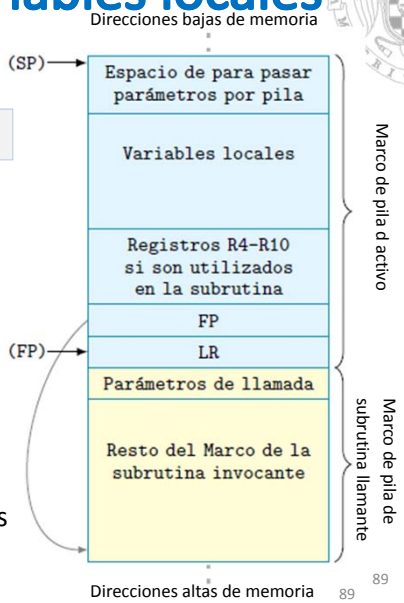
88

Ampliación de variables locales

- Reservar espacio en pila:

```
SUB SP, SP, #4*NumPalabrasExtra
```

- Las variables locales suelen direccionarse con **desplazamientos fijos relativos a FP**
  - Para variables creadas dentro de la función con desplazamientos negativos.
  - Para parámetros pasados por pila, se direccionan sobre el marco de pila de la función llamante con desplazamientos positivos.



fc²

Ampliación de variables locales

```
int Mayor(int X, int Y)
{
int m;
if(X>Y)
        m = X;
else
        m = Y;
return m;
}
```

```
Mayor:
push {fp}
mov fp, sp
sub sp, sp, #12
str r0, [fp,#-4] @ Ini. X con el primer parámetro
str r1, [fp,#-8] @ Ini. Y con el segundo parámetro

@ if( X > Y )
ldr r0, [fp,#-4] @ r0   X
ldr r1, [fp,#-8] @ r1   Y
cmp r0, r1
ble ELS
@ then
        ldr r0, [fp,#-4] @ m = X;
        str r0, [fp,#-12]
        b RET
@ else
        ELS: ldr r0, [fp,#-8] @ m = Y;
        str r0, [fp,#-12]

RET: @ return m
ldr r0, [fp,#-12] @ valor de retorno
mov sp, fp
pop {fp}
mov pc, lr
```

fc²

Pasando de C a Ensamblador

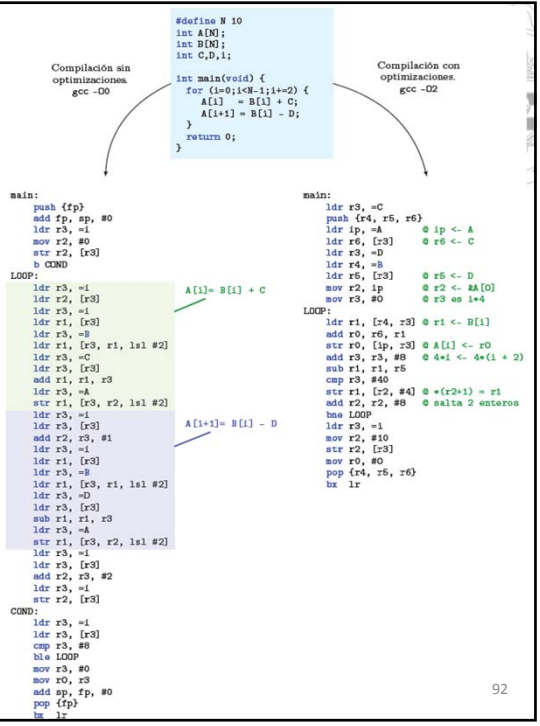
- Los compiladores se estructuran generalmente en tres partes:
  - **Front end o parser** que se encarga de comprobar que el código escrito es correcto sintácticamente y de traducirlo a una representación intermedia independiente del lenguaje.
  - **Middle end** que se encarga de **realizar optimizaciones**, que pueden tener distintos objetivos, por ejemplo:
    - reducir el tiempo de ejecución del código final,
    - reducir la cantidad de memoria que utiliza o
    - reducir el consumo energético en su ejecución.
  - **Back end** que se encarga de generar el código máquina para la arquitectura destino (generalmente dan código ensamblador como salida y es el ensamblador el que produce el código máquina final).

fc²

91

Pasando de C a Ensamblador

- Sin optimizar se puede identificar claramente los bloques de instrucciones ensamblador por las que se ha traducido cada sentencia C.
- Para poder hacer una depuración correcta a nivel de C es necesario generar código sin optimización.
- Es habitual tener dos configuraciones:
  - Debug
  - Realease



fc²

92

-O0

```
#define N 10
int A[N];
int B[N];
int C,D,i;

int main(void) {
  for (i=0;i<N-1;i+=2) {
     A[i]   = B[i] + C;
     A[i+1] = B[i] - D;
  }
  return 0;
}
```

```
main:
     push {fp}
     add fp, sp, #0
     ldr r3, =i
     mov r2, #0
     str r2, [r3]
     b COND
```

```
LOOP:
     ldr r3, =i
     ldr r2, [r3]
     ldr r3, =i
     ldr r1, [r3]
     ldr r3, =B
     ldr r1, [r3, r1, lsl #2]
     ldr r3, =C
     ldr r3, [r3]
     add r1, r1, r3
     ldr r3, =A
     str r1, [r3, r2, lsl #2]
     ldr r3, =i
     ldr r3, [r3]
     add r2, r3, #1
     ldr r3, =i
     ldr r1, [r3]
     ldr r3, =B
     ldr r1, [r3, r1, lsl #2]
     ldr r3, =D
     ldr r3, [r3]
     sub r1, r1, r3
     ldr r3, =A
     str r1, [r3, r2, lsl #2]
     ldr r3, =i
     ldr r3, [r3]
     add r2, r3, #2
     ldr r3, =i
     str r2, [r3]
```

```
COND:
     ldr r3, =i
     ldr r3, [r3]
     cmp r3, #8
     ble LOOP
     mov r3, #0
     mov r0, r3
     add sp, fp, #0
     pop {fp}
     bx  lr
```

fc²

93

-O2

```
#define N 10
int A[N];
int B[N];
int C,D,i;

int main(void) {
  for (i=0;i<N-1;i+=2) {
     A[i]   = B[i] + C;
     A[i+1] = B[i] - D;
  }
  return 0;
}
```

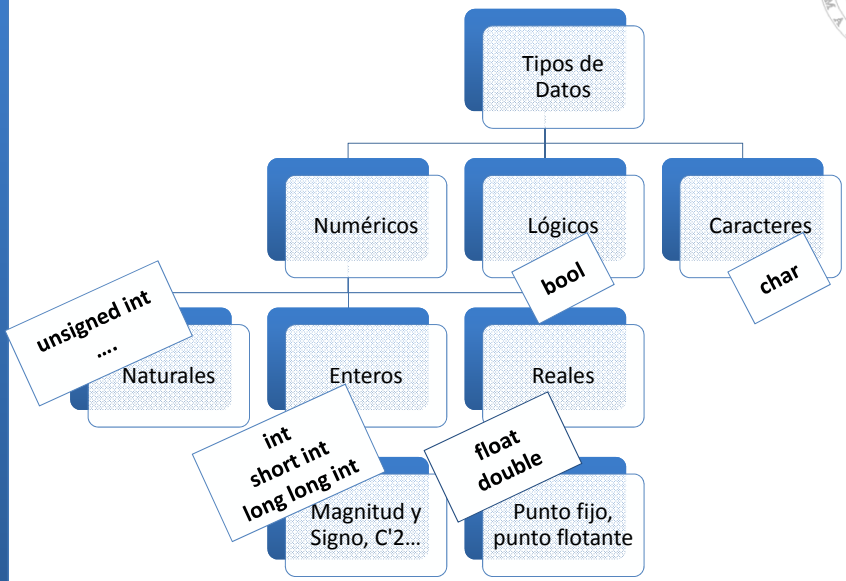
```
main:
     ldr r3, =C
     push {r4, r5, r6}
     ldr ip, =A          @ ip <- A
     ldr r6, [r3]        @ r6 <- C
     ldr r3, =D
     ldr r4, =B
     ldr r5, [r3]        @ r5 <- D
     mov r2, ip          @ r2 <- &A[0]
     mov r3, #0          @ r3 es i*4
LOOP:
     ldr r1, [r4, r3] @ r1 <- B[i]
     add r0, r6, r1
     str r0, [ip, r3] @ A[i] <- r0
     add r3, r3, #8   @ 4*i <- 4*(i + 2)
     sub r1, r1, r5
     cmp r3, #40
     str r1, [r2, #4] @ *(r2+1) = r1
     add r2, r2, #8   @ salta 2 enteros
     bne LOOP
     ldr r3, =i
     mov r2, #10
     str r2, [r3]
     mov r0, #0
     pop {r4, r5, r6}
     bx  lr
```

fc²

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



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

fc²

97

Tipos de datos básicos

- Tamaño *habitual* de los datos básicos en los lenguajes de alto nivel (p.e.: C/C++)

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

- Tanto el hardware como el lenguaje ensamblador nos deben permitir acceder a la memoria con estos tamaños de datos.

fc²

98

Accesos a memoria: tamaño y alineamiento

- En ARMv4 el modo de direccionamiento de las instrucciones ldr/str permite seleccionar accesos de tamaño:
  - byte,
  - media palabra (half word, 16 bits),
  - palabra (word, 32 bits)
  - doble palabra (double word, 64 bits)
- Para indicar el tamaño del acceso en lenguaje ensamblador se añaden sufijos a las instrucciones ldr/str normales.
- Además se da soporte a una interpretación sin signo, para dar soporte a los tipos *unsigned*.
- **Existen restricciones de alineamiento**

fc²

99

Accesos a memoria: tamaño y alineamiento

- **LDRB:** load de un entero sin signo de tamaño byte.
- **LDRSB:** load de un entero con signo de tamaño byte. Se extiende con el bit de signo hasta los 32 bits.
- **STRB:** se escribe en memoria un entero de tamaño byte.
- **LDRH:** load de un entero sin signo de 16 bits.
- **LDRSH**: load de un entero con signo de 16 bits. Se extiende con el bit de signo hasta los 32 bits.
- **STRH:** se escribe en memoria un entero de 16 bits.
- **LDRD:** carga dos registros consecutivos con dos palabras (32 bits) consecutivas de memoria.
- **STRD:** es la operación contraria a la anterior.

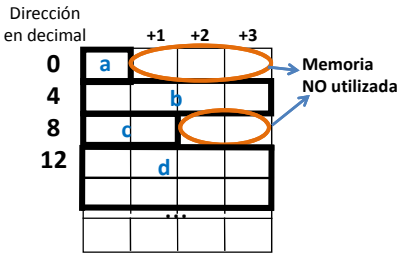
```
LDRB R5, [R9] @ Carga en el byte 0 de R5 Mem(R9) y pone los bytes 1,2 y 3 a 0
LDRB R3, [R8, #3] @ Carga en el byte 0 de R3 Mem(R8 + 3) y pone los bytes 1,2 y 3 a 0
STRB R4, [R10, #0x200] @ Almacena el byte 0 de R4 en Mem(R10+0x200)
STRB R10, [R7, R4] @ Almacena el byte 0 de R10 en Mem(R10+R4)
LDRH R1, [R0] @ Carga en los byte 0,1 de R1 Mem(R0) y pone los bytes 2,3 a 0
LDRH R8, [R3, #2] @ Carga en los byte 0,1 de R8 Mem(R3+2) y pone los bytes 2,3 a 0
STRH R2, [R1, #0X80] @ Almacena los bytes 0,1 de R2 en Mem(R1+0x80)
LDRD R4, [R9] @ Carga en R4 una palabra de Mem(R9) y en R5 una palabra de Mem(R9+4)
STRD R8, [R2, #0x28] @ Almacena R8 en Mem(R2+0x28) y R9 en Mem(R2+0x2C)
```

fc²

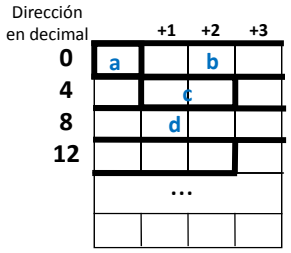
100

Alineamiento de variables

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



Variables alineadas



Variables no alineadas

Dirección de comienzo de la variable

a → 0
b → 4
c → 8
d → 12

a → 0
b → 1
c → 5
d → 7

fc²

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)

fc²

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

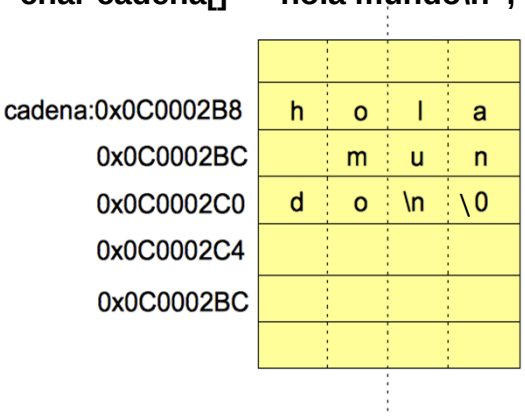
Source: www.LookupTables.com

fc²

103

Arrays

char cadena[] = “hola mundo\n”;



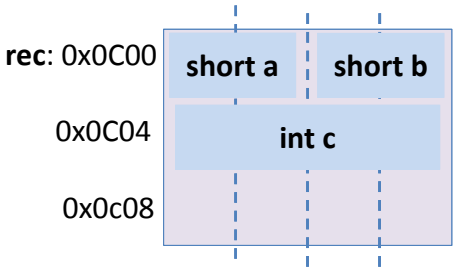
- Los elementos de un array ocupan posiciones consecutivas de memoria
- Las cadenas de caracteres en C acaban en \0
- ¿En qué dirección está v[16] del array *int vector[100];* si su primer elemento está en la dirección 0x0C00?

fc²

104

Estructuras

```
struct mistruct {
    char primero;
    short int segundo;
    int tercero;
};
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?

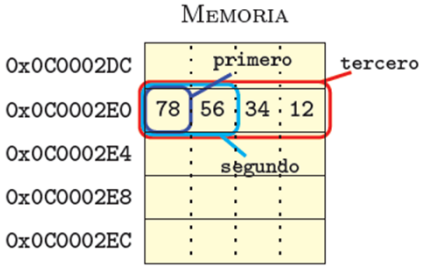
fc²

105

Uniones

- Una unión tiene varios campos que utilizan una región de memoria común.
- La cantidad de memoria coincide con la del campo de mayor tamaño.
- Se emplaza en una dirección que satisfaga las restricciones de alineamiento.

```
union miunion {
        char primero;
        short int segundo;
        int tercero;
};
union miunion un;
```



fc²

Varios ficheros fuente

- Normalmente programaremos en lenguaje de alto nivel.
- En ensamblador solamente lo estrictamente necesario:
  - por requisitos de eficiencia,
  - Para utilizar instrucciones especiales de la arquitectura.
- En la etapa de enlazado se deben resolver referencias cruzadas entre los ficheros objeto, para ello se crean *Tablas de Símbolos*.

> arm-none-eabi-nm -SP -f sysv ejemplo.o

```
Symbols from ejemplo.o:

Name              Value    Class      Type       Size     Line  Section

globalA          |00000000|   D  |     OBJECT|00000002|      |.data
globalB          |        |   U  |     NOTYPE|        |      |*UND*
main             |00000000|   T  |       FUNC|00000054|      |.text
printf           |        |   U  |     NOTYPE|        |      |*UND*
```


Símbolos en C

- *extern:* exportar /importar la visibilidad a/de otro fichero
- *static:* para restringir su visibilidad al fichero actual
- En C todas las variables globales y todas las funciones son por defecto símbolos globales exportados

Cuadro 1 Ejemplo de exportación de símbolos.

// fichero fun.c

```c
//declaración de variable global
//definida en otro sitio
extern int var1;

//definición de var2
//sólo accesible desde func.c
static int var2;

//declaración adelantada de one
void one(void);

//definición de two
//al ser static el símbolo no se
//exporta, está restringida a este
//fichero
static void two(void)
{
   ...
   var1++;
   ...
}

void fun(void)
{
   ...
   //acceso al único var1
   var1+=5;
   //acceso a var2 de fun.c
   var2=var1+1;
   ...
   one();
   two();
   ...
}
```

// fichero main.c

```c
//declaración de variable global
//definida en otro sitio (más abajo)
extern int var1;

//definición de var2
//sólo accesible desde main.c
static int var2;

//declaración adelantada de one
void one(void);

//declaración adelantada de fun
void fun(void);

int main(void)
{
   ...
   //acceso al único var1
   var1 = 1;
   ...
   one();
   fun();
   ...
}

//definición de var1
int var1;

void one(void)
{
   ...
   //acceso al único var1
   var1++;
   //acceso a var2 de main.c
   var2=var1-1;
   ...
}
```

Símbolos en Ensamblador

- En ensamblador los símbolos son por defecto locales, no visibles desde otro fichero.
- Si queremos hacerlos globales debemos exportarlos con la directiva

  .global
- Si queramos hacer referencia a un símbolo definido en otro fichero se utiliza la directiva

  .extern

```
.extern FOO @importamos un símbolo externo
.global start  @exportamos un símbolo local
```

fc²

109

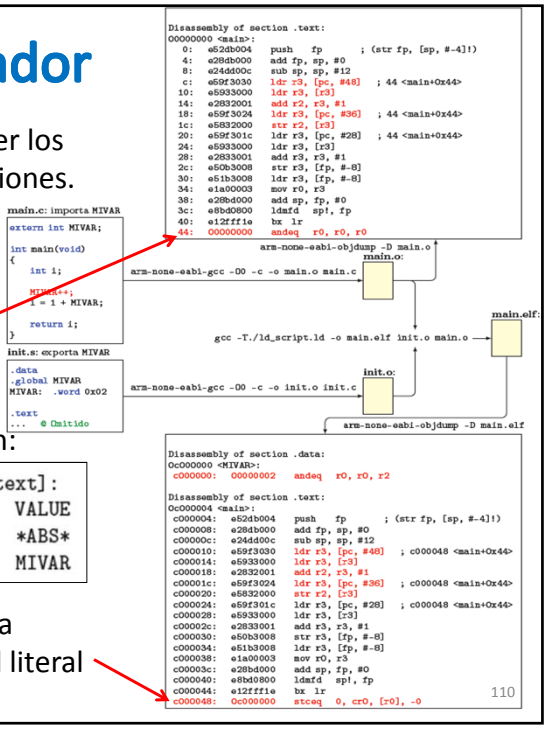
C y Ensamblador

- El enlazador debe resolver los símbolos externos y secciones.

  P.e. MIVAR
- El compil./ensambl.:
  - Crea un literal al final de *.text*
  - Crea una entrada en tabla de relocalización:

```
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000040 R_ARM_V4BX        *ABS*
00000044 R_ARM_ABS32       MIVAR
```

- El enlazador resuelve esta relocalización y rellena el literal



fc²

110