

Arquitectura ARM

Esquema

- **Introducción**
- Registros y Modos de procesador
- Modelo de programación
 - Tipos de datos e instrucciones
 - Modos de direccionamiento
 - Excepciones
- Repertorio de instrucciones
- Subrutinas

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)

Versiones de la arquitectura ARM

- ARMv3
 - Empleada en el ARM6, primer procesador tras separarse de Acorn
 - Direcciones de 32-bits, registros CPSR/SPSR separados, soporte de memoria virtual
- ARMv4
 - Añade load/store de media palabra
 - Variante T: añade un subconjunto de instrucciones cortas (16 bits)

Versiones de la arquitectura ARM

■ ARMv5

- Mejoras procesamiento redes, instrucción CLZ (count leading-zeroes)
- Variante E: mejoras orientadas al procesamiento digital de señal (DSP) como aritmética con saturación, multiplicación de 16-bits ...
- Variante J (Jazelle): ofrece aceleración HW y SW en la ejecución del bytecode Java

■ ARMv6 y ARMv7

- Incluye las mejoras 'TEJ'
- Añade multiprocesamiento, instrucciones SIMD, etc.

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, Cortex
- Otras familias importantes
 - StrongARM, XScale, A-4, A-5, Tegra, Snapdragon...

ARM7TDMI

- Procesador en la placa de desarrollo de Embest usada en prácticas
- Implementa la versión v4T
- Segmentado (*pipeline*) de 3 etapas
 - Búsqueda - decodificación - ejecución (*fetch – decode - execute*)
- Multiplicador avanzado
- Soporte para depuración

Arquitectura ARM v4T

- Arquitectura Load-Store
- Instrucciones de longitud fija (32-bits)
- Formato de 3 operandos (2 fuente, 1 destino)
- Ejecución condicional de TODAS las instrucciones
- Instrucciones de Load-Store múltiple
- Desplazamiento de n-bits integrado en los operandos
- Instrucciones Thumb (compresión de código)

Esquema

- Introducción
- **Registros y Modos de procesador**
- Modelo de programación
 - Tipos de datos e instrucciones
 - Modos de direccionamiento
 - Excepciones
- Repertorio de instrucciones
- Subrutinas

Modos de operación del procesador

- Los procesadores ARM tienen 7 modos de operación:
 - **User (usr)**: estado normal de ejecución
 - **FIQ**: manejo de interrupciones rápidas para transferencias de datos
 - **IRQ**: manejo de interrupciones de propósito general o lentas
 - **Supervisor (svc)**: modo protegido para el sistema operativo
 - **Abort (abt)**: gestiona los fallos de acceso a memoria (causados por accesos convencionales o prebúsquedas)
 - **Undef (und)**: manejo de excepciones por códigos de operación no definidas
 - **System**: modo privilegiado para el sistema operativo, usando los mismos registros que en el modo usuario

Registros

- 37 registros de 32-bits de longitud
 - 1 contador de programa (dedicado)
 - 1 registro de estado actual del programa (dedicado)
 - 5 registros para guardar el estado del programa (dedicados)
 - 30 registros de propósito general
- ¡No todos están disponibles simultáneamente!

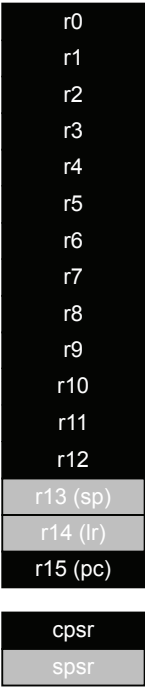
Registros y modos

- El modo en el que se encuentra el procesador determina los registros accesibles
- En cada modo se puede acceder a:
 - Un conjunto particular de registros generales (r0-r12)
 - Registros de puntero de pila (r13) y enlace (r14) particulares
 - El contador de programa (r15)
 - El registro de estado actual del programa (cpsr)
- En los modos privilegiados (excepto system) se puede acceder también a:
 - Un registro especial que almacena el estado del programa (spsr)

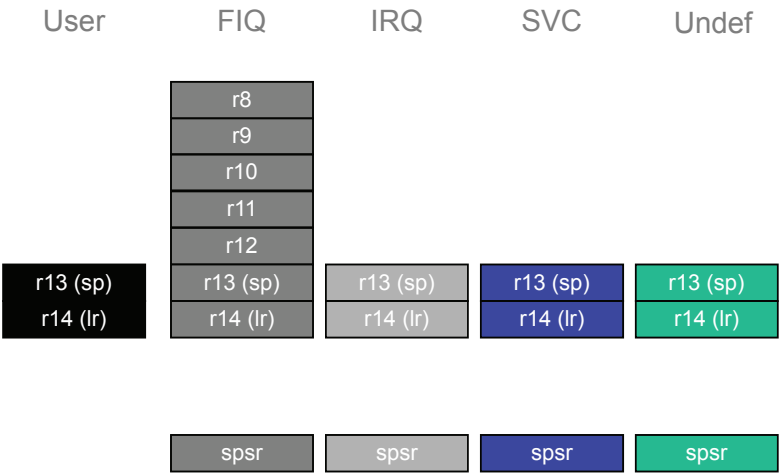
Registros y modos

Registros Visibles

Abort Mode



Registros No-Visibles



Registros y modos

User

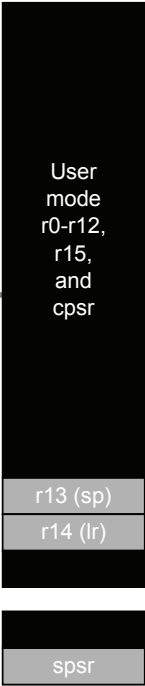
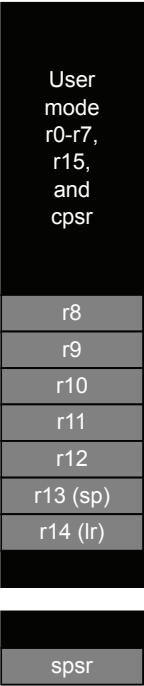
FIQ

IRQ

SVC

Undef

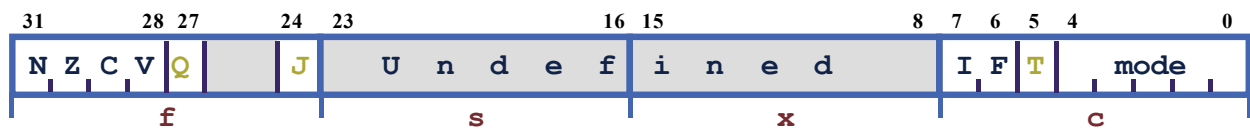
Abort



repertorio
Thumb
Registros
inferiores

repertorio Thumb
Registros
superiores

Registro de estado



- Códigos de condición
 - N = resultado de la ALU negativo (C2)
 - Z = resultado de la ALU cero
 - C = operación de ALU produce acarreo
 - V = operación de ALU desborda (C2)
- Q y J
 - Sólo para la arquitectura 5TE/J
- Bits de habilitación de interrupciones
 - I = 1: Deshabilita IRQ.
 - F = 1: Deshabilita FIQ.
- T Bit
 - Sólo para arquitecturas -T
 - T = 0: repertorio ARM
 - T = 1: repertorio Thumb
- Bits de modo
 - Especifican el modo del procesador

Contador de programa (r15)

- Cuando el procesador trabaja en repertorio **ARM**:
 - Todas las instrucciones son de **32 bits** y tienen alineamiento de tamaño palabra (32 bits)
 - Por lo tanto, el PC sólo guarda los bits [31:2], dejando los bits [1:0] indefinidos
- Cuando el procesador trabaja en repertorio **Thumb**:
 - Todas las instrucciones son de **16 bits** y tienen alineamiento de tamaño media-palabra (16 bits)
 - Por lo tanto, el PC sólo guarda los bits [31:1], dejando el bit [0] indefinido

Esquema

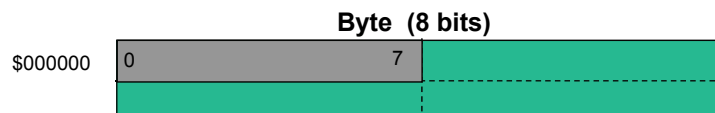
- Introducción
- Registros y Modos de procesador
- **Modelo de programación**
 - Tamaño de los operandos
 - Modos de direccionamiento
- Repertorio de instrucciones
- Subrutinas

Tipos de instrucciones

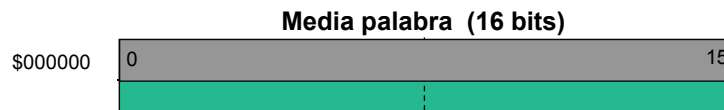
- Los procesadores ARM son arquitecturas de 32 bits, implementan dos repertorios de instrucciones
 - Repertorio ARM estándar de 32 bits
 - Repertorio Thumb de 16 bits (versión compacta)
- Las versiones con Jazelle ejecutan Java bytecode directamente en hardware

Tamaño de los operandos

- **Byte**, 8 bits



- **Halfword** (media palabra), 16 bits, alineación: 2 bytes

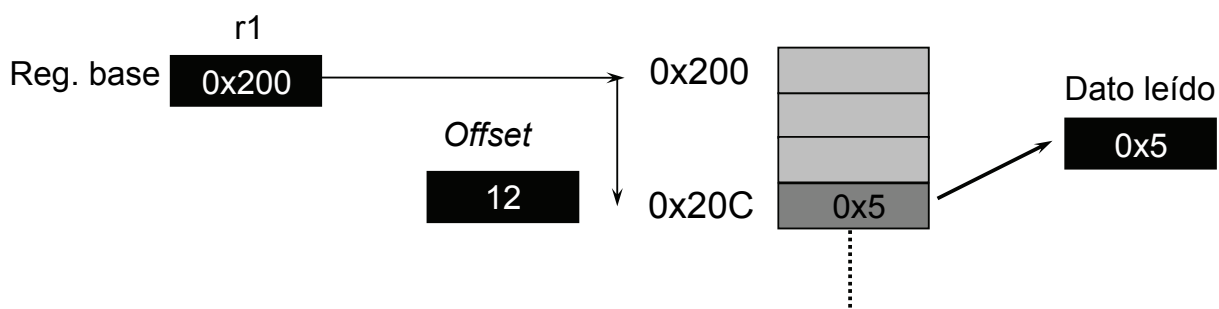


- **Word** (palabra), 32 bits, alineación: 4 bytes



Modos de direccionamiento

- Diversas variantes de registro-base



- Opciones para el offset:

- Valor inmediato
- Valor de registro pre/post-incrementado/decrementado

Ordenación y alineamiento de los accesos

■ Ordenación:

- Admite tanto *big-endian* como *little-endian* (configurable)
- Ejemplo: *little-endian*

\$0000000	0	Byte-0	7	8	Byte-1	15
	16	Byte-2	23	24	Byte-3	31

■ Alineamiento:

- ¡ ARMv4T sólo admite accesos alineados !

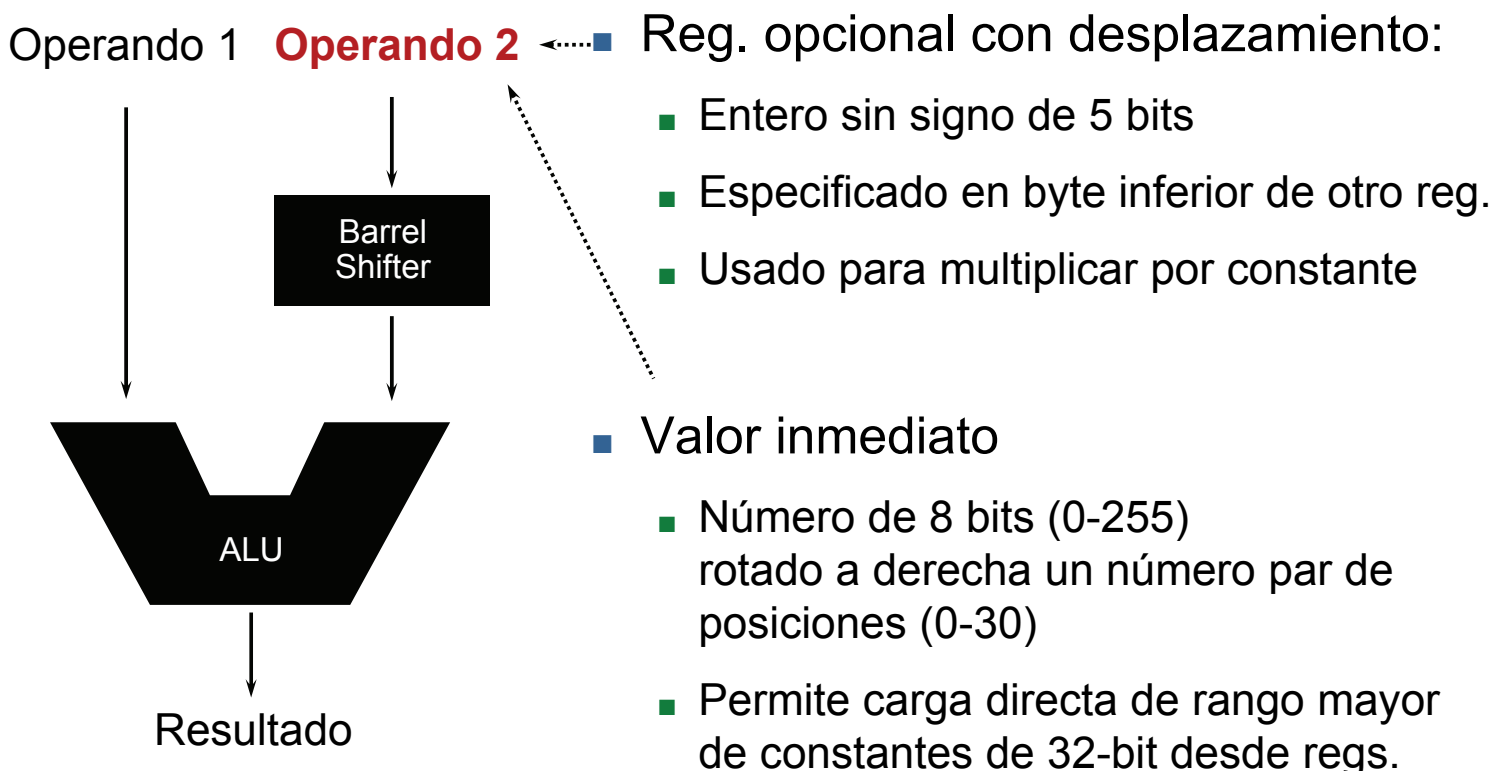
Esquema

- Introducción
- Modelo de programación
- **Repertorio de instrucciones**
 - Inst. procesado de datos (aritméticas, lógicas, etc.)
 - Inst. de control
 - Inst. de acceso a memoria
 - Inst. manipulación del registro de estado
 - Cambio de modo de ejecución y estado (ARM o Thumb)
- Subrutinas

Instrucciones de procesamiento de datos

- Sólo trabajan con registros (NO con memoria), tipos:
 - Aritméticas: **ADD** **ADC** **SUB** **SBC** **RSB** **RSC**
 - Operaciones lógicas: **AND** **ORR** **EOR** **BIC**
 - Comparaciones: **CMP** **CMN** **TST** **TEQ**
 - Movimientos de datos: **MOV** **MVN**
- Sintaxis: **<operación>{<cond>}{S} Rd, Rn, operand2**
 - Las comparaciones sólo fijan los flags (no el valor de Rd)
 - Los movimientos de datos no especifican Rn
- El segundo operando se envía a la ALU a través de una operación de desplazamiento de registro (*barrel shifter*)

Uso de barrel Shifter: segundo operando



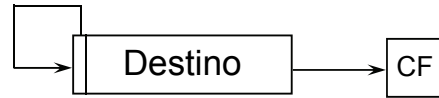
Operaciones de desplazamiento de registro

LSL : Logical Left Shift



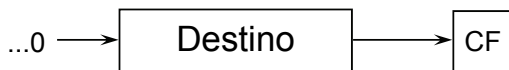
Multiplicación por potencia de 2

ASR: Arithmetic Right Shift



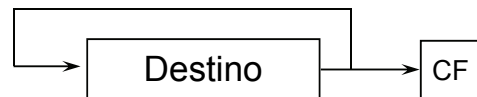
División entre potencia de 2,
preserva el bit de signo

LSR : Logical Shift Right



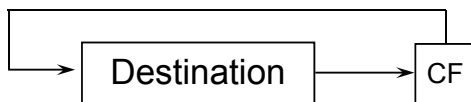
División entre potencia de 2

ROR: Rotate Right



Rotar bit con realimentación
del menos al más significativo

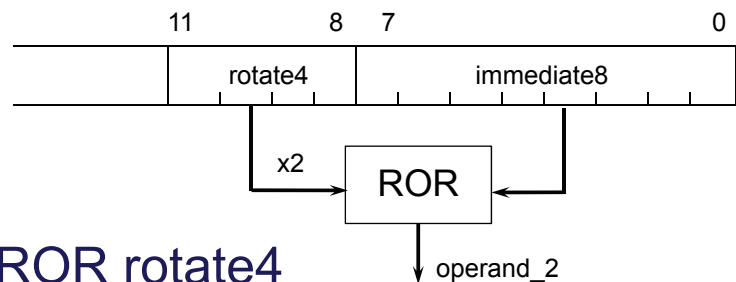
RRX: Rotate Right Extended



Rotar bit (1 posición) con realimentación
de CF al bit más significativo

Valores inmediatos (1/3)

- Las instrucciones no contienen constantes de 32 bits
 - Todas las instrucciones ARM tienen un tamaño fijo de 32 bits
- Instr. procesamiento datos: **12 bits para operando 2**
 - En principio, sólo pueden codificarse valores entre 0 y 4095
 - Pero ...

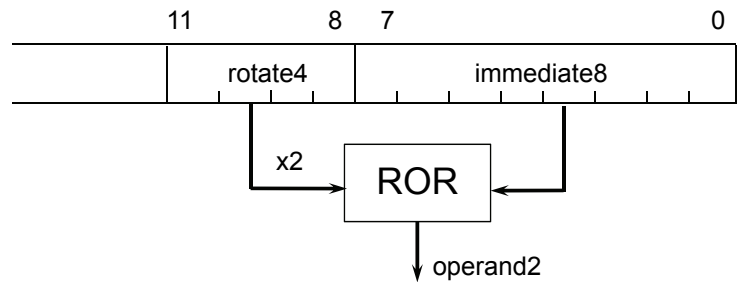


- operand_2 = immediate8 ROR rotate4**
 - Valor inmediato: 8 bits (0-255)
 - Valor rotación: 4 bits (0-15), se multiplica por 2 (0-30, paso 2)
 - Regla: los últimos 8 bits rotan hacia la derecha un número par de posiciones

Valores inmediatos (2/3)

■ Desensamblar la instrucción

0xE3A004FF



■ MOV r0, #???

■ Doce últimos bits: 0x4FF

■ rotate4: 4

■ immediate8: 0xFF = 255

MOV r0, #0x000000FF ror 8 => MOV r0, #0xFF000000

Valores inmediatos (3/3)

■ Ejemplos:

ror #0	0 0	rango 0-0x000000ff paso 0x00000001
ror #8	0 0	rango 0-0xff000000 paso 0x01000000
ror #30	0 0	rango 0-0x000003fc paso 0x00000004

■ El ensamblador convierte valores inmediatos a formas rotadas

■ ADD r1,r2,#0x00FF0000 ; 0x000000FF ror 16

■ MOV r0,#0x00001000 ; 0x40 ror 26

■ Los valores que no puede generar así producen error

■ MOV r0,#0x01FE; Error: invalid constant (1fe) after fixup

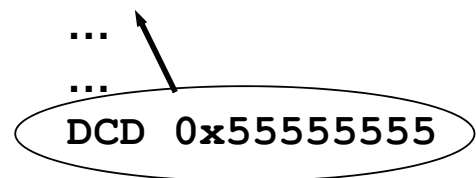
■ Algunos ensambladores usan trucos con valores "imposibles"

■ MOV r0, #0xFFFFFFFF ; -> MVN r0, #0

■ Script interactivo: <http://alisdair.mcdiarmid.org/2014/01/12/arm-immediate-value-encoding.html>

Cargando constantes de 32 bits a registros

- El ensamblador usa pseudo-instrucción para ctes grandes:
 - `LDR rd, =const`
- Esto produce:
 - a) Una instrucción **MOV** o **MVN** para generar el valor.
 - `LDR r0,=0xFF` \rightarrow `MOV r0,#0xFF`
 - b) Una instrucción **LDR** con un desplazamiento relativo del PC para leer la constante de una zona de constantes en el código (*literal pool*)
 - `LDR r0,=0x55555555` \rightarrow `LDR r0,[PC,#Imm12]`



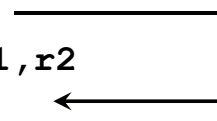
Multiplicación

- Sintaxis:
 - `MUL{<cond>}{S} Rd, Rm, Rs` ; $Rd = Rm * Rs$
 - `MLA{<cond>}{S} Rd,Rm,Rs,Rn` ; $Rd = (Rm * Rs) + Rn$
 - `[U|S]MULL{<cond>}{S} RdLo, RdHi, Rm, Rs` ; $Rd_{Hi}, Rd_{Lo} := Rm * Rs$
 - `[U|S]MLAL{<cond>}{S} RdHi, RdLo, Rm, Rs` ; $Rd_{Hi}, Rd_{Lo} := (Rm * Rs) + Rd_{Hi}, Rd_{Lo}$
- Tiempo ejecución según procesador (“terminac. anticipada”)
 - Operación MUL básica
 - 2-5 ciclos ARM7TDMI; 1-3 en StrongARM/XScale; 2 en ARM9E/ARM102xE
 - +1 operandos “long”
 - MULS tarda 4 ciclos; pero MULLS y MLALS tardan siempre 5 ciclos

Ejecución condicional y activación de flags

- Las instrucciones ARM se pueden ejecutar condicionalmente con un **campo de condición**
 - Mejora la densidad de código y reduce el número de saltos

```
CMP    r3,#0
BEQ    skip
ADD    r0,r1,r2
skip
```

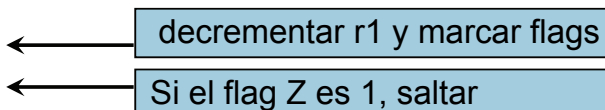


```
CMP    r3,#0
ADDNE  r0,r1,r2
```

- Las instrucciones de procesamiento sólo afectan a los *flags* de condición si se activa “S” (CMP **no** lo necesita)

loop

```
...
SUBS   r1,r1,#1
BNE    loop
```



decrementar r1 y marcar flags

Si el flag Z es 1, saltar

Condiciones admitidas

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	

Nota: AL se define por defecto y no requiere indicarlo

Ejemplo de ejecución condicional

- Secuencia de instrucciones condicionales

```
if (a==0) func(1);  —————>  CMP    r0,#0
                                   MOVEQ  r0,#1
                                   BLEQ   func
```

Ejemplos de ejecución condicional (2/2)

- Fijar los flags, y después usar varios códigos de condición

```
if (a==0) x=0;
if (a>0)  x=1;
```

- Usar instrucciones de comparación condicionales

```
if (a==4 || a==10)
    x=0;
```

Nota: el valor de **a** está cargado en **r0**, el de **x** en **r1**
Ayuda: usad **cmp rs1,rs2** y **mov rd,rs**

Ejemplos de ejecución condicional (2/2)

- Fijar los flags, y después usar varios códigos de condición

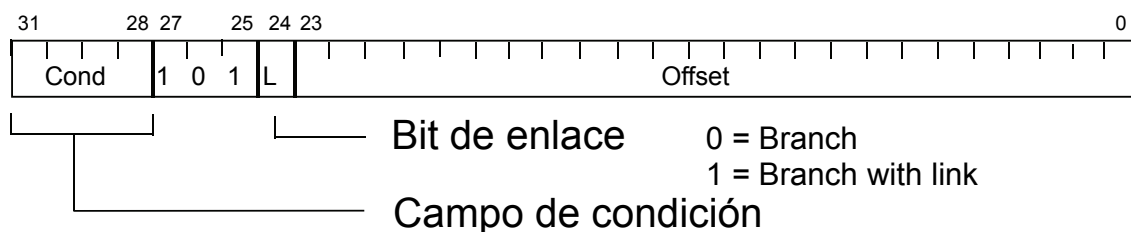
<code>if (a==0) x=0;</code>	<code>CMP</code>	<code>r0, #0</code>
<code>if (a>0) x=1;</code>	<code>MOVEQ</code>	<code>r1, #0</code>
	<code>MOVGT</code>	<code>r1, #1</code>

- Usar instrucciones de comparación condicionales

<code>if (a==4 a==10)</code>	<code>CMP</code>	<code>r0, #4</code>
<code>x=0;</code>	<code>CMPNE</code>	<code>r0, #10</code>
	<code>MOVEQ</code>	<code>r1, #0</code>

Instrucciones de control

- Salto: **B**{<cond>} label
- Salto con enlace: **BL**{<cond>} etiqueta_subrutina

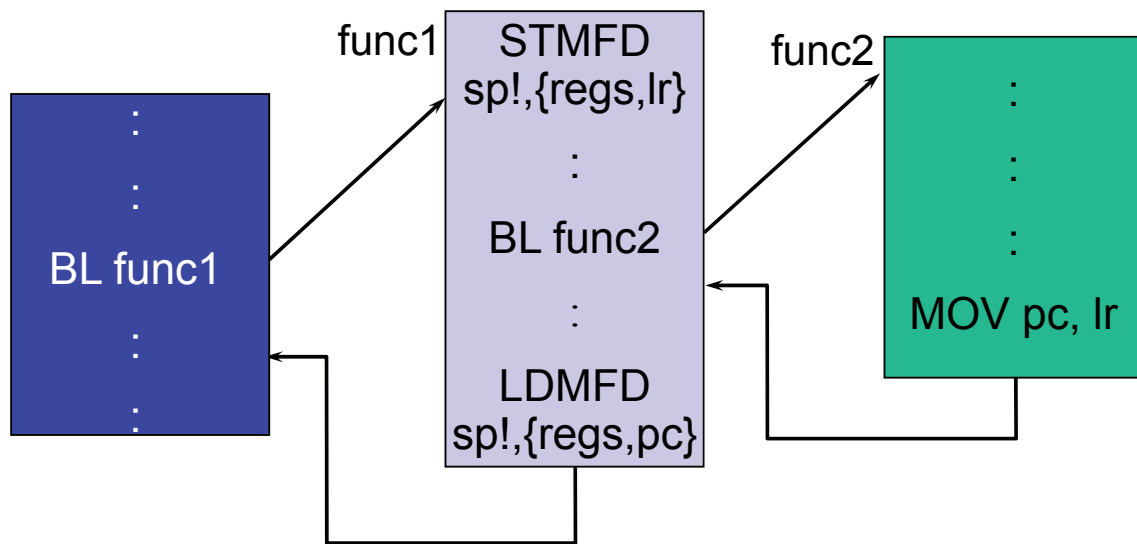


- El procesador desplaza dos posiciones a la izquierda el campo de offset, extiende su signo y lo suma al **PC**
 - Rango de ± 32 Mbytes
 - Problema con saltos mayores (hasta 32 bits)
 - Fijar **LR** a mano y cargarlo en **PC**

Salto y subrutinas

■ Sintaxis:

- **B** <etiqueta> ; salto relativo al PC, con rango ± 32 Mbytes
- **BL** <subrutina> ; salto a subrutina, dir. de retorno guardada en LR
 - Retorno implementado restaurando el PC desde LR



Instrucciones de acceso a memoria

- El sistema de memoria soporta todos los tamaños de datos

■ Sintaxis:

- **LDR**{<cond>}{<size>} Rd, <address>
- **STR**{<cond>}{<size>} Rd, <address>

■ Ejemplos:

- **LDR STR** word
- **LDRB STRB** byte
- **LDRH STRH** halfword
- **LDRSB** carga de byte con signo
- **LDRSH** carga de media palabra con signo
- **LDREQB** ¿qué operación realiza?

Direcccionamiento

■ LDR/STR: registro base + *offset*

1. *Offset* para palabras y bytes sin signo

a) Valor inmediato de 12 bits (0 - 4095 bytes): `LDR r0, [r1, #8]`

b) Un registro (desplazado por un inmediato, opcionalmente)

`r0, [r1, r2]`

`LDR r0, [r1, r2, LSL#2]`

■ Puede sumarse/restarse:

`LDR r0, [r1, #-8]`

`LDR r0, [r1, -r2]`

`LDR r0, [r1, -r2, LSL#2]`

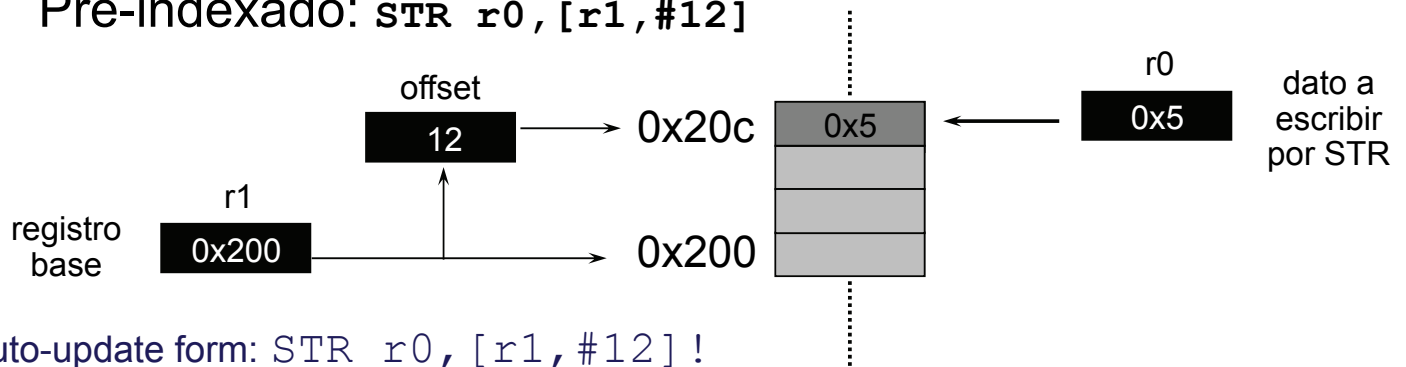
2. *Offset* para medias-palabras y bytes con signo

a) Valor inmediato de 8 bits (0 - 255 bytes)

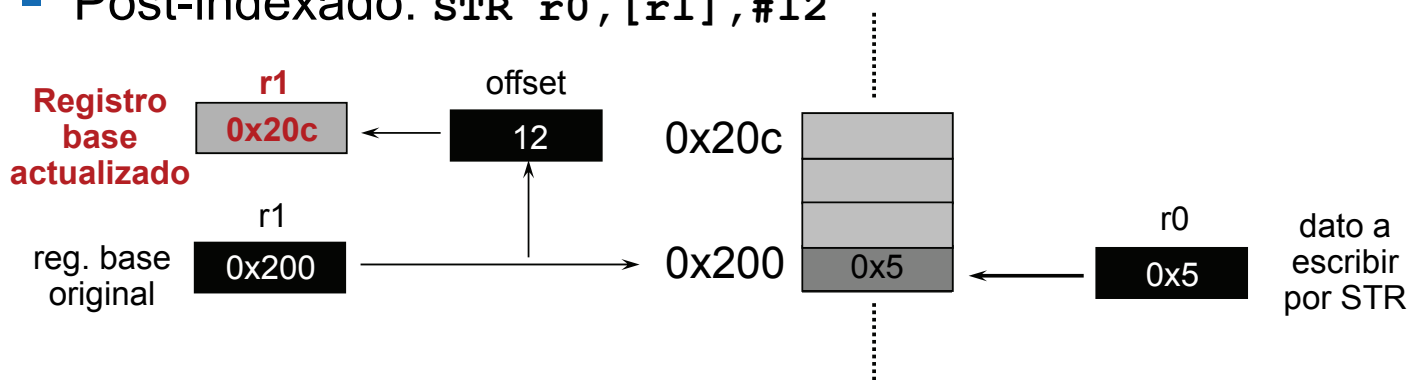
b) Un registro (sin desplazamiento)

Direcccionamiento Pre/Post-Indexado

■ Pre-indexado: `STR r0, [r1, #12]`



■ Post-indexado: `STR r0, [r1], #12`



Instrucciones de LOAD/STORE múltiple

■ Sintaxis:

<LDM | STM>{<cond>}<modo_direccionamiento> Rb{!}, <lista registros>

■ Modos de direccionamiento:

Modo de direccionam.	@ inicial	@ final	Rb!
IA (increment after)	Rb	$Rb + 4*N - 4$	$Rb + 4*N$
IB (increment before)	$Rb + 4$	$Rb + 4*N$	$Rb + 4*N$
DA (decrement after)	$Rb - 4*N + 4$	Rb	$Rb - 4*N$
DB (decrement before)	$Rb - 4*N$	$Rb - 4$	$Rb - 4*N$

■ Ejemplos:

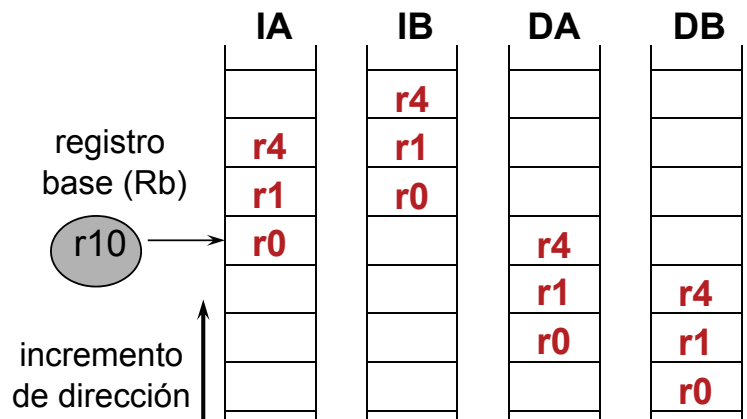
STMIA r10!, {r0,r1,r4}

LDMDB r10!, {r0,r1,r4}

STMIB r10!, {r0,r1,r4}

LDMDA r10!, {r0,r1,r4}

...

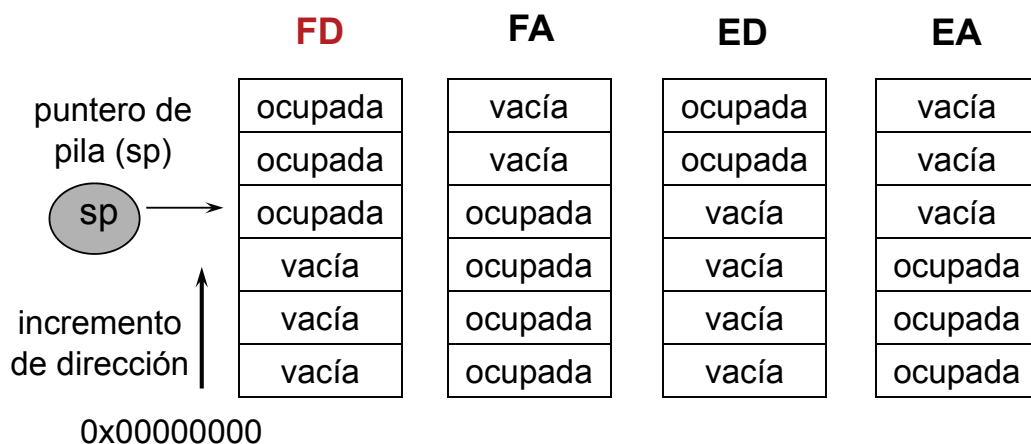


Operaciones de pila con LDM/STM (1/2)

■ Tipos de **pila**:

- Descendente / ascendente: crece hacia direcciones decrecientes (*descending stack*) / crecientes (*ascending stack*)
- Llena / vacía: el puntero de pila apunta al último objeto de la pila (*full stack*) / a la siguiente posición vacía (*empty stack*)

■ Combinaciones:



Operaciones de pila con LDM/STM (2/2)

- Modos de direccionamiento para cada tipo de pila:

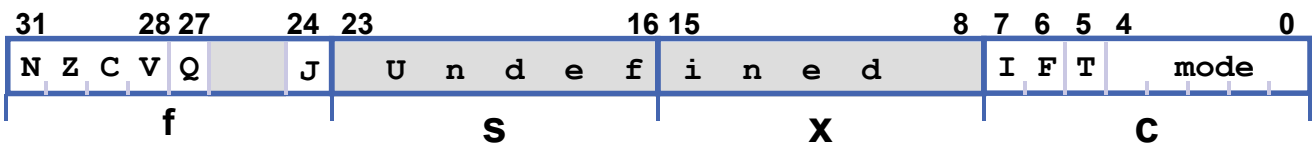
Tipo pila	store / push	load / pop
FD (Full Descending stack)	DB (Decrement Before)	IA (Increment After)
FA (Full Ascending stack)	IB (Increment Before)	DA (Decrement After)
ED (Empty Descending stack)	DA (Decrement After)	IB (Increment Before)
EA (Empty Ascending stack)	IA (Increment After)	DB (Decrement Before)

- Correspondencias

Tipo pila	store / push	load / pop
FD (Full Descending stack)	STMFD → STMDB	LDMFD → LDMIA
FA (Full Ascending stack)	STMFA → STMIB	LDMFA → LMDMA
ED (Empty Descending stack)	STMED → STMDA	LDMED → LDMIB
EA (Empty Ascending stack)	STMEA → STMIA	LDMEA → LDMDB

Referencia:
<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0473c/Cacbgchh.html>

Manipulación del registro de estado



- MRS/MSR transfieren CPSR / SPSR desde/a un registro

- Sintaxis:

- `MRS{<cond>} Rd,<psr>` ; Rd = <psr>
- `MSR{<cond>} <psr[_campos]>,<Rm>` ; <psr[_campos]> = Rm

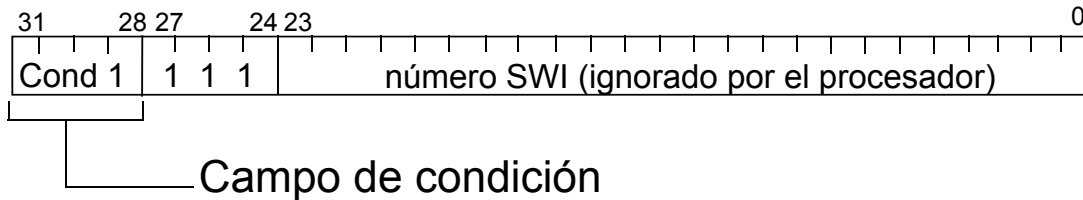
donde: <psr> = CPSR o SPSR ; [_campos] = combinación de 'fsxc'

- Con campo inmediato: `MSR{<cond>} <psr_campos>,#Inmediato`
- En modo *user*, todos los bits se pueden leer, pero sólo los flags de condición (`_f`) se pueden escribir

Interrupción Software (SWI)

■ Sintaxis:

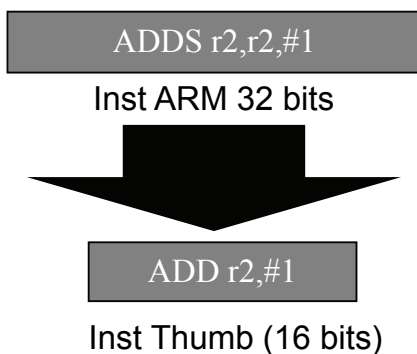
■ **SWI{<cond>} <número SWI>**



- Provoca una excepción y salto al vector hardware de SWI
- El manejador (*handler*) de SWI puede examinar el número SWI para decidir qué operación se ha solicitado.
- Usado por el SO para implementar operaciones privilegiadas que las aplicaciones en modo usuario pueden solicitar

Thumb: repertorio de instrucciones compacto

- Subconjunto repertorio de operaciones de ARM en 16 bits
 - Optimizado para densidad código de C (~65% del tamaño ARM)
 - Mejora de rendimiento para memorias de ancho menor
- Estado de ejecución adicional (Thumb) para este repertorio
 - Cambio entre el modo ARM y el modo Thumb usando la instr. **BX**

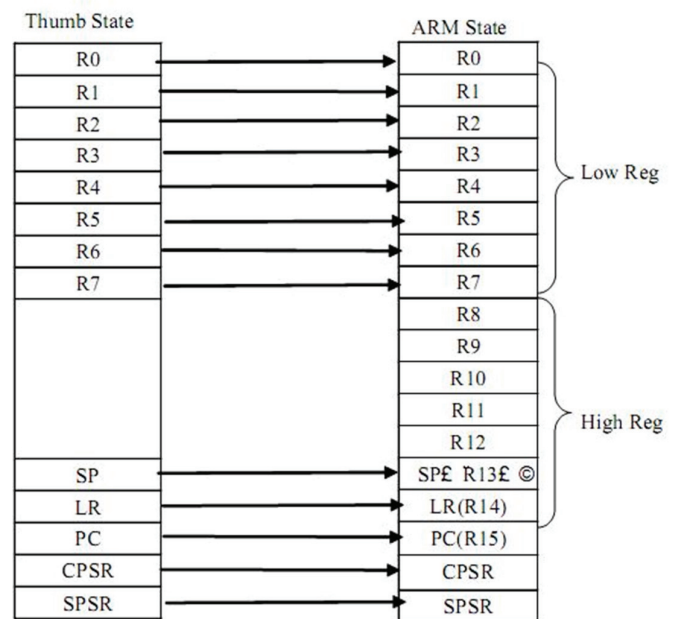


Principales razones para código generado (gcc):

- Ejecución condicional no usada
- Registros fuente y destino son idénticos
- Pocos registros usados; normalmente sólo los “low registers” y sólo a veces los “high registers”
- Ctes. de tamaño limitado (*barrel shift* sin uso)

Thumb: sólo hay 8 registros de propósito general

- En general las instrucciones trabajan con los registros r0-r7
- Algunas instrucciones pueden acceder a los registros r8-r12
- Se pueden utilizar para almacenar datos usando **mov**
- Se pueden usar para comparaciones usando **cmp**



Cambiar de ARM a thumb

- El ensamblador ARM y Thumb pueden mezclarse:
 - `.arm` indica que vamos a usar el repertorio ARM
 - `.thumb` indica que vamos a usar el repertorio Thumb
 - Para pasar de ensamblador ARM a thumb se usa la instrucción **BX**
- **BX Rn**
 - Salta a la dirección especificada en los bit Rn[31:1]
 - Copia Rn[0] en el bit T del registro de estado (este bit indica si se está ejecutando ensamblador ARM o Thumb)

Ejemplo

```
.arm                                /* Subsequent instructions are ARM */
header:
    ADR    r0, Tstart + 1          /* Processor starts in ARM state, */
    BX     r0                      /* small ARM code header used to call Thumb main
    program. */
    NOP
.thumb
Tstart:
    MOV     r0, #10                /* Set up parameters */
```


Esquema

- Introducción
- Registros y Modos de procesador
- Modelo de programación
 - Tipos de datos e instrucciones
 - Modos de direccionamiento
 - Excepciones
- Repertorio de instrucciones
- **Subrutinas**

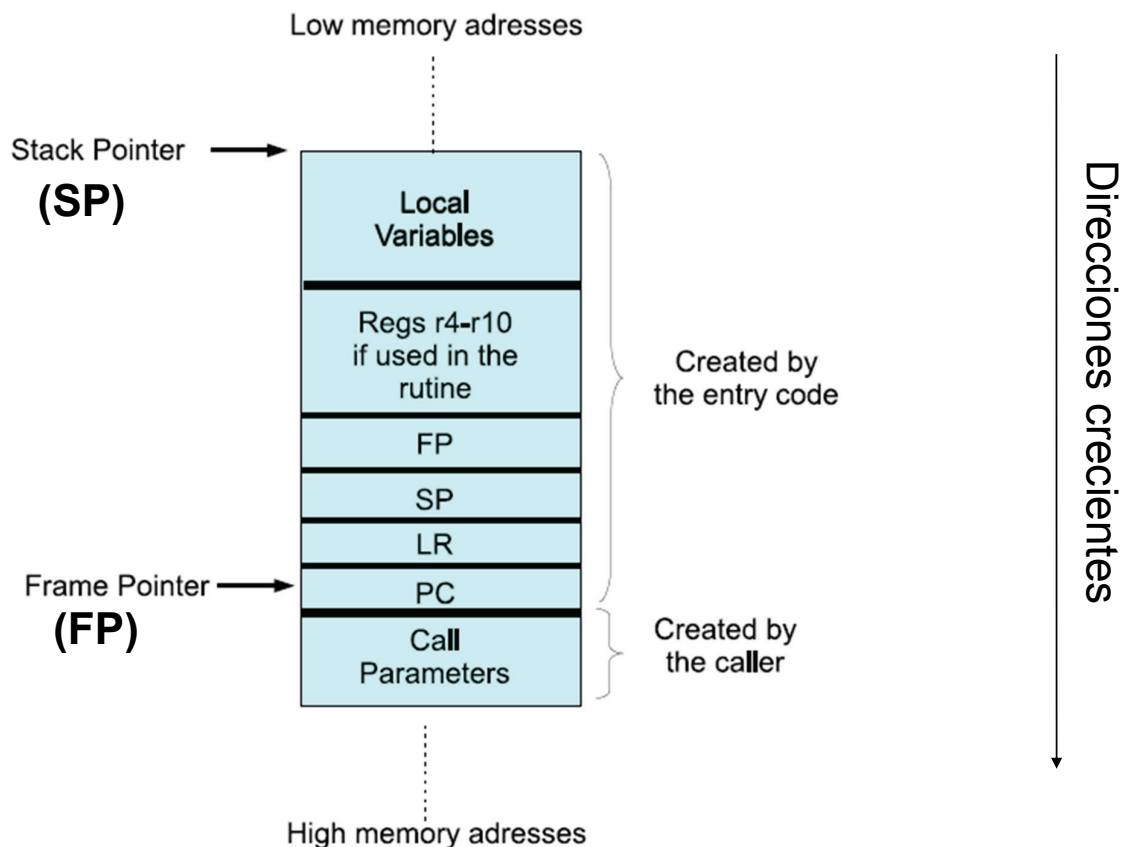
Llamadas a funciones y paso de parámetros

- En esta práctica vamos a usar el compilador GCC, que aplica el ATPCS (ARM-THUMB Procedure Call Standard) para las llamadas a funciones
- Uso de los registros en ATPCS:
 - r0-r3 (a1-a4): pasar argumentos, devolver resultado y guardar datos temporales dentro de la rutina. **No hace falta guardarlos en pila**
 - r4-r10 (v1-v7): almacenar variables dentro de la rutina
 - r12 (IP): registro auxiliar que guarda el valor original del SP. **Las funciones pueden borrarlo sin salvarlo**
 - r13 (SP), r14 (LR), r15 (PC): registros de uso especial
 - r11 (FP) también es especial en algunas variantes de ATPCS: apunta a la base del marco de pila

Marco de pila (versión FP)

- ATPCS asume una pila **Full Descending (FD)**
- Cada rutina crea un marco en la pila con:
 - Los registros que debe preservar
 - La dirección de retorno
 - Las variables locales de la rutina
- SP apunta a la cima del marco
- FP apunta a la base del marco
 - **Las variables locales son accedidas a través del FP**

Marco de pila



Marco de pila

■ Estructura de una rutina:

Código de entrada (prólogo) ← **Construye el marco**

Cuerpo de la rutina

Código de salida (epílogo) ← **Destruye el marco y
retorna a función llamadora**

- El prólogo y el epílogo no son siempre iguales: ¡dependen del compilador!
- En las prácticas seguiremos la misma estructura que el compilador que usemos

Marco de pila

■ Ejemplo de prologo y epílogo:

■ Prólogo

```
MOV    IP, SP
STMDB  SP!, {r4-r10,FP,IP,LR,PC}
SUB    FP, IP, #4
SUB    SP, #SpaceForLocalVariables
```

■ Epílogo

```
LDMDB  FP, {r4-r10,FP,SP,PC}
```

IP = SPprólogo LR

