Sistemas Empotrados

Tema 2:
Procesador y mapa de memoria

Lección 5:
Arquitectura del procesador

Jesús González Peñalver

Contenidos

El procesador

Conocerlo a fondo es fundamental

Es la parte del sistema que ejecutará nuestro firmware

Las prestaciones dependerán del grado de optimización de nuestro código

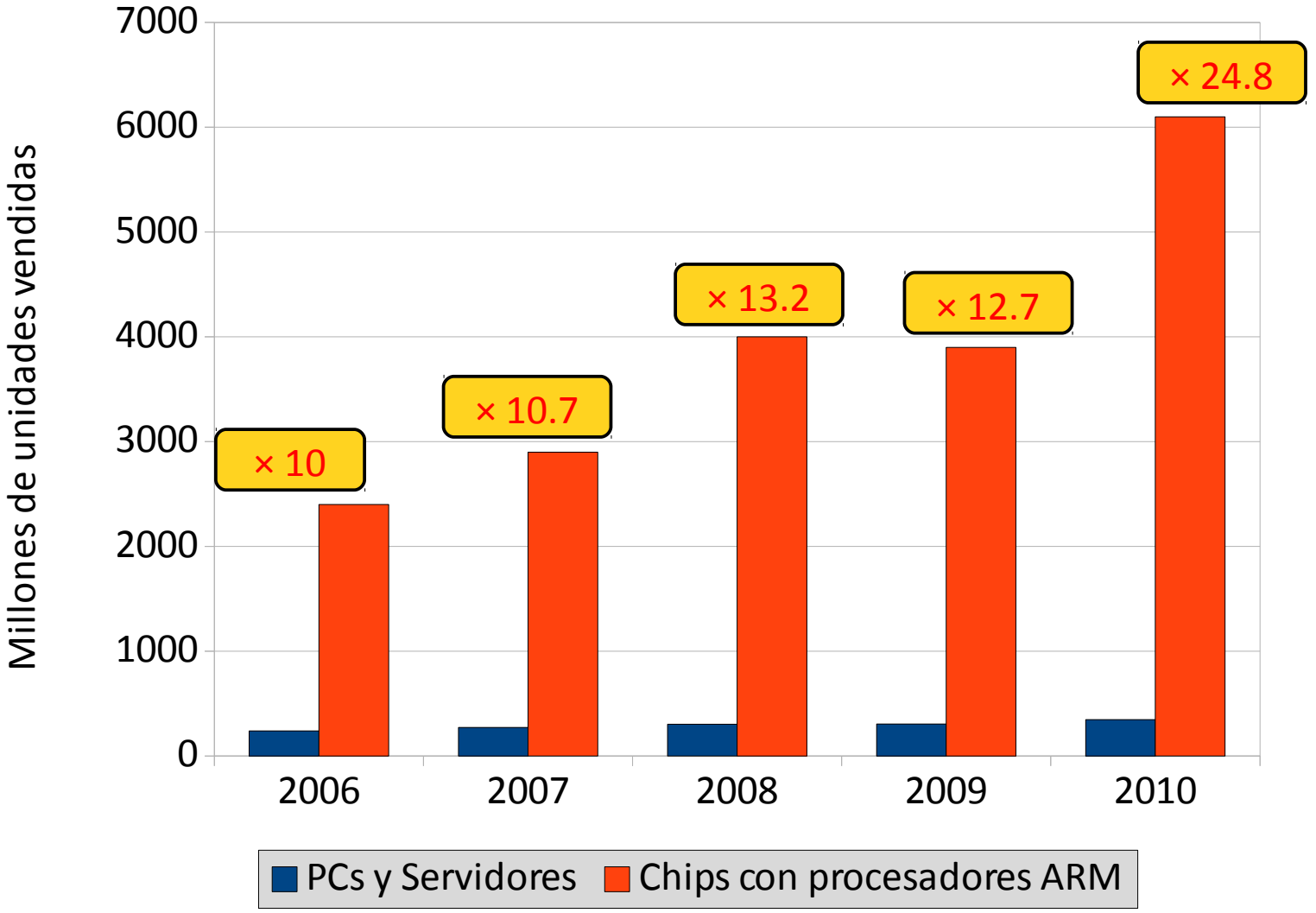
Inconveniente

Actualmente hay muchas arquitecturas/procesadores para sistemas empotrados

… y muchos más

¿Por qué ARM?

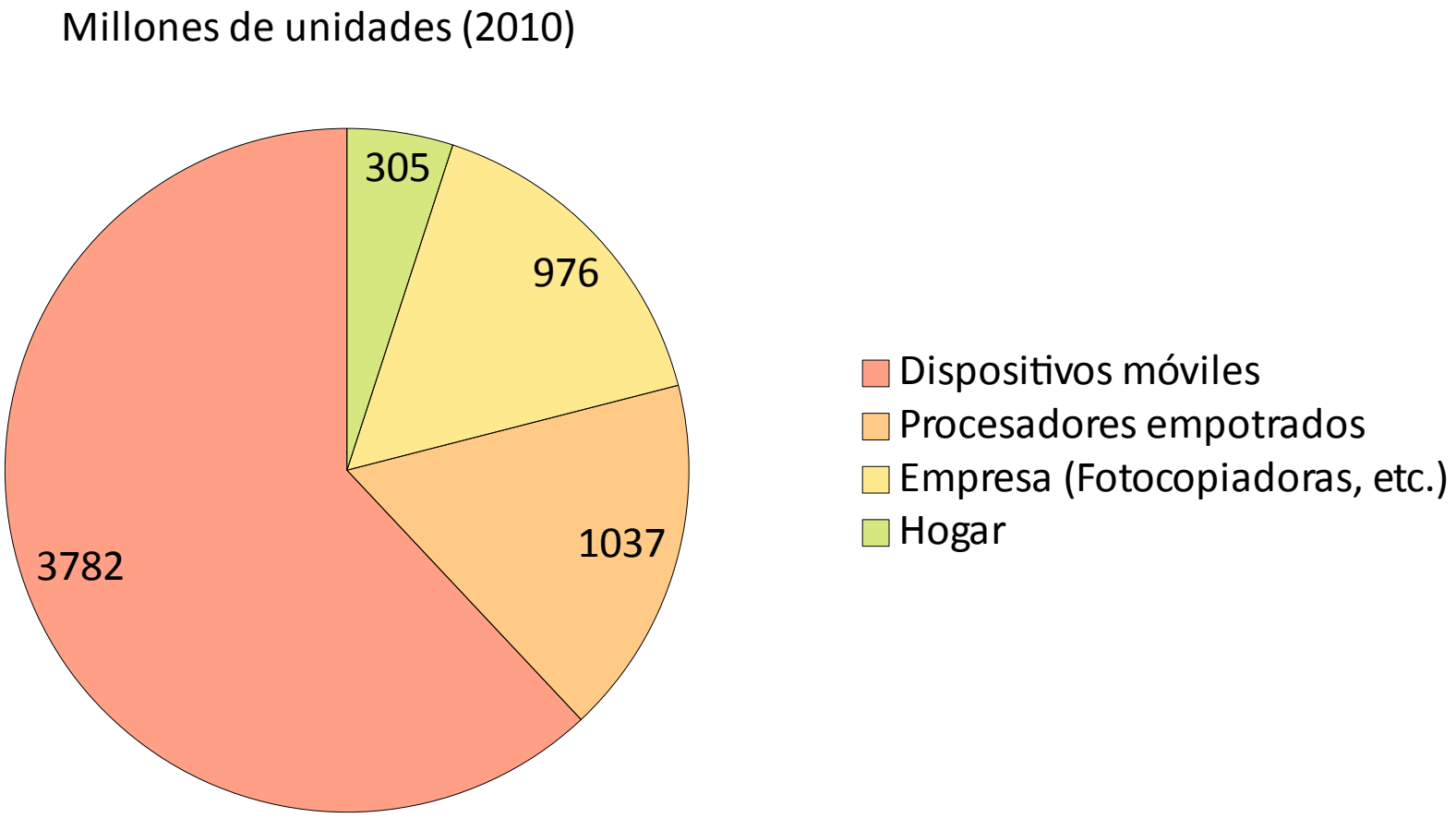
Son los procesadores más usados actualmente



Fuente: ARM Annual Reports and Accounts.
http://www.arm.com/annualreport10/

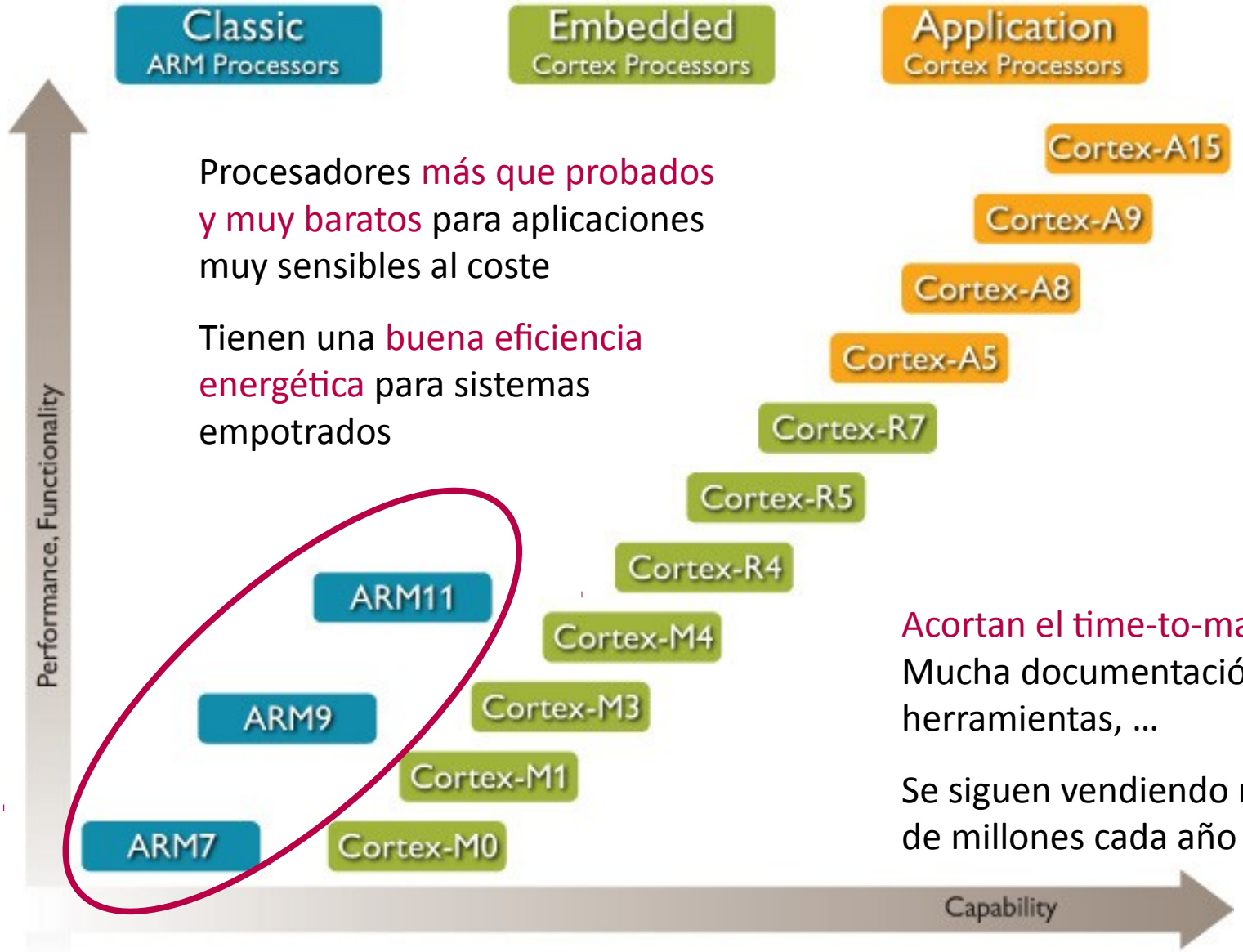
¿Por qué ARM?

Se emplean fundamentalmente en sistemas empotrados y dispositivos móviles



Fuente: ARM Annual Reports and Accounts.
http://www.arm.com/annualreport10/

Ok, ARM, pero... ¿qué familia?



Procesadores más que probados y muy baratos para aplicaciones muy sensibles al coste

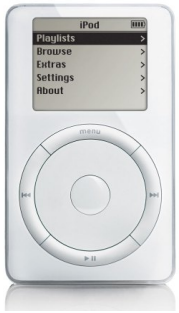
Tienen una buena eficiencia energética para sistemas empotrados

Acortan el time-to-market: Mucha documentación, herramientas, …

Se siguen vendiendo miles de millones cada año

Aplicaciones del ARM7

Nintendo Gameboy Advance (2001)
Consola de videojuegos



Apple iPod (2001)
Reproductor multimedia

Nokia 3300 (2003)
Teléfono móvil

Kodak EasyShare LS753 (2004)
Cámara

Ramos RM970 PMP (2007)
Reproductor multimedia

Allerta inPulse (2011)
Smartwatch

Aplicaciones del ARM9

Nintendo DS (2004)
Consola de videojuegos

HTC TyTN (2006)
Teléfono móvil 3G

Canon EOS 5D Mark II (2008)
Cámara

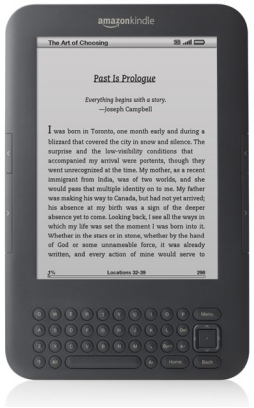
LG Arena (2009)
Smartphone

Archos 7 (2010)
Tablet

Hero H2000 (2011)
Smartphone chino Dual-sim

Aplicaciones del ARM11

Apple iPhone 3G (2008)
Smartphone



Amazon Kindle 2 (2009)
Lector de e-books

Mobinova Elan (2009)
Netbook

iRobot aPad (2010)
Clon chino del iPad

Nintendo 3DS (2011)
Consola de videojuegos

Nokia X7 (2011)
Smartphone

Contenidos

Los procesadores ARM tienen arquitectura RISC

CISC (60's)	RISC (80's)
Objetivos: Programación en ensamblador, códigos compactos	Objetivos: Generación automática de código, mejorar las prestaciones
Instrucciones complejas que requieren múltiples ciclos Diferentes formatos y tamaños de instrucción, difíciles de decodificar, microcódigo	Instrucciones sencillas que sólo requieren un ciclo Formato fijo, decodificación y modos de direccionamiento sencillos, sin microcódigo
Muchas instrucciones pueden acceder a datos en memoria	Sólo los LOADS y STORES pueden acceder a datos en memoria
Las instrucciones se ejecutan de una en una	Usa segmentación de cauce para solapar la ejecución de instrucciones
Pocos registros de propósito general	Muchos registros de propósito general
El código tiende a ser más compacto	El código tiende a ocupar más memoria

Mejoras de ARM a la arquitectura RISC para Sistemas Empotrados

Tiempo de ejecución variable para ciertas instrucciones: Load-Store-múltiple

- Aumenta la densidad de código, ya que se usa en todas las funciones para meter y sacar registros en la pila
- Su tiempo de ejecución depende del número de registros transferidos y de si las direcciones son consecutivas

Uso de un desplazador de bloques en una de las entradas de la ALU:

- Genera constantes de 32 bits a partir de un operando inmediato que se desplazará antes de entrar a la ALU
- Se aumentan las prestaciones y la densidad de código

Estados (Thumb, Jazelle, Thumb2):

- Thumb es un repertorio de instrucciones de 16 bits. Mejora la densidad de código sobre el 30%
- Jazelle permite ejecutar bytecodes de Java directamente por el procesador (sin máquina virtual)

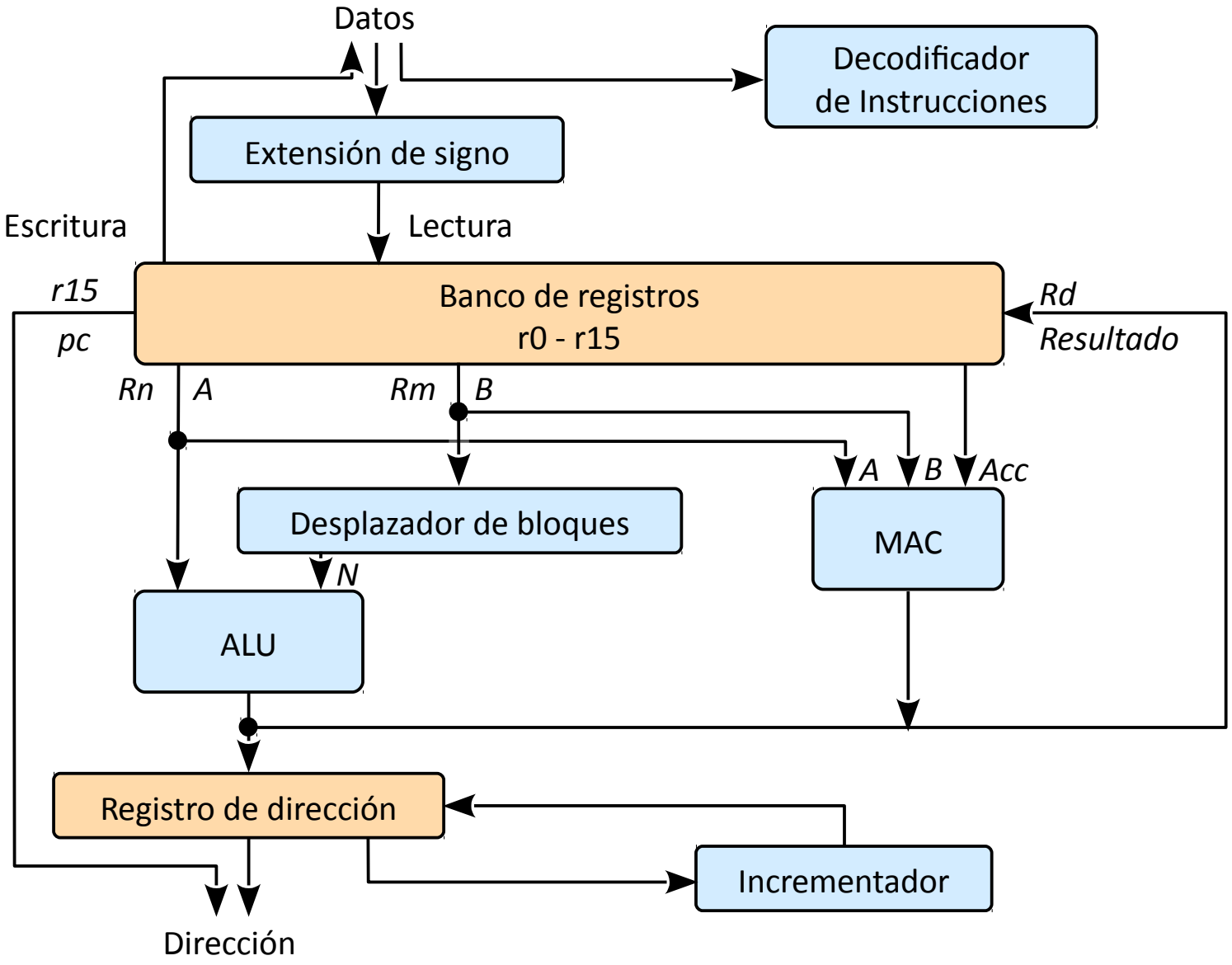
Ejecución condicional de instrucciones:

- Aumenta la densidad de código
- Mejora las prestaciones al evitar muchos saltos condicionales

Adición de instrucciones específicas:

- Instrucciones específicas para aplicaciones multimedia, DSP, FP, seguridad, etc.
- Mejoran las prestaciones de ciertos algoritmos muy usados en aplicaciones actuales

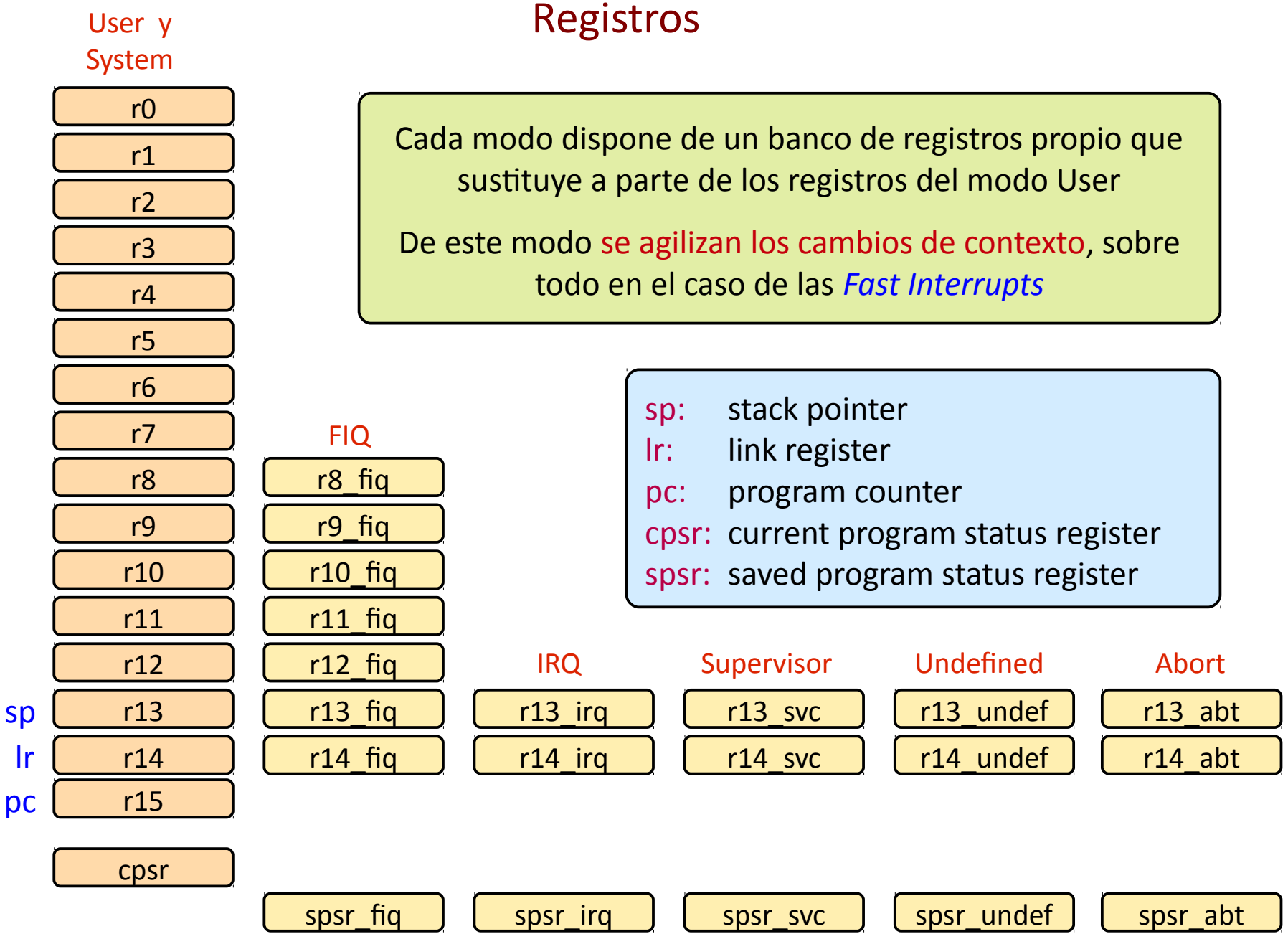
Modelo de flujo de datos de un procesador ARM



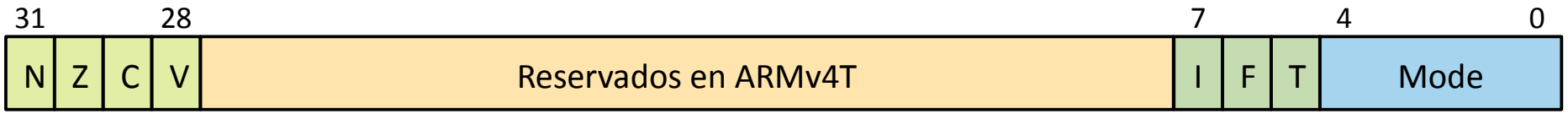
Modos del procesador

Modo	Descripción	
Supervisor (SVC)	Se entra cada vez que se hace un reset y cuando ocurre una Interrupción Software (SWI)	Modos privilegiados Para servir excepciones o acceder a recursos protegidos
FIQ	Se entra cada vez que ocurre una Interrupción de alta prioridad (Fast)	
IRQ	Se entra cada vez que ocurre una Interrupción de baja prioridad (normal)	
Abort	Para manejar violaciones de acceso a memoria	
Undef	Para manejar instrucciones no definidas	
System	Modo privilegiado que usa los mismos registros que el modo User	
User	Modo en el que se ejecutan la mayoría de aplicaciones y tareas del sistema operativo	Modo sin privilegios

Registros


User y System
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
sp
r13
lr
r14
pc
r15
cpsr
FIQ
r8_fiq
r9_fiq
r10_fiq
r11_fiq
r12_fiq
r13_fiq
r14_fiq
spsr_fiq
IRQ
r13_irq
r14_irq
spsr_irq
Supervisor
r13_svc
r14_svc
spsr_svc
Undefined
r13_undef
r14_undef
spsr_undef
Abort
r13_abt
r14_abt
spsr_abt
Cada modo dispone de un banco de registros propio que sustituye a parte de los registros del modo User
De este modo se agilizan los cambios de contexto, sobre todo en el caso de las Fast Interrupts
sp: stack pointer
lr: link register
pc: program counter
cpsr: current program status register
spsr: saved program status register

Los registros de estado (CPSR y SPSR)



Flags de condición — Solo lectura

N = Resultado Negativo en la ALU
Z = El resultado de la ALU es Zero
C = Se ha producido aCarreo en la ALU
V = Se ha producido oVerflow en la ALU

Máscaras de interrupción — Se pueden modificar en los modos privilegiados

I = 1, deshabilita las IRQ
F = 1, deshabilita las FIQ

Modo de ejecución — Solo lectura

T = 0, procesador en estado ARM
T = 1, procesador en estado Thumb

Bits de modo — Se pueden modificar en los modos privilegiados

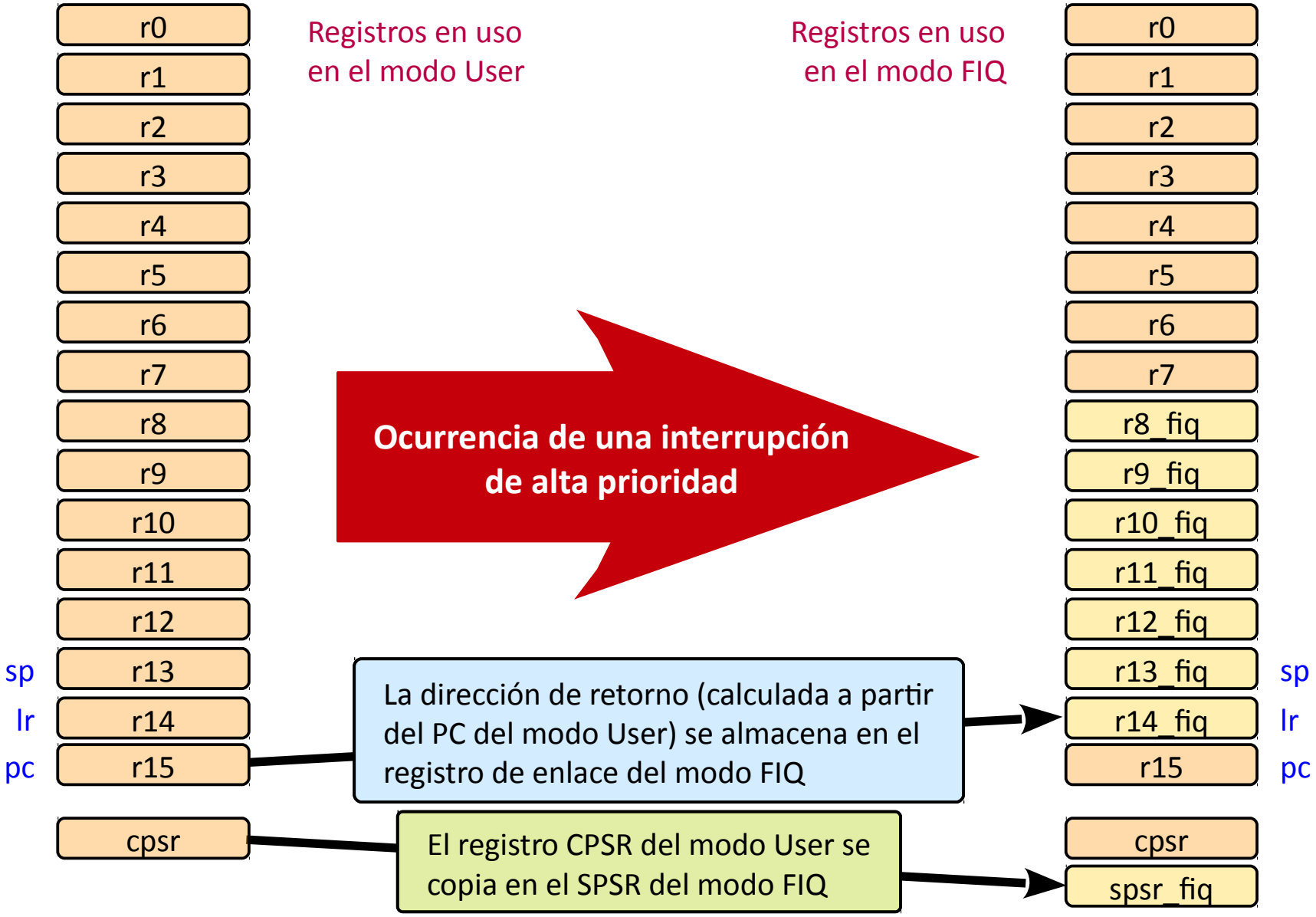
10000 : User
10001 : FIQ
10010 : IRQ
10011 : Supervisor
10111 : Abort
11011 : Undefined
11111 : System

Cualquier valor diferente provocará resultados impredecibles

Bits reservados — No se deben modificar

Se leen como cero

Ejemplo de cambio de modo



Ejecución condicional

Justificación

- Permite la ejecución condicionada de instrucciones sin hacer uso de saltos
- Las instrucciones no ejecutadas sólo gastan 1 ciclo de CPU
- Se reduce considerablemente el número de saltos condicionales del programa, que causan el vaciado del cauce y la consecuente penalización

Uso

<instrucción>{<cond>}{sufijos} <operandos>

Condiciones

eq:	Igual	ne:	No igual
gt:	Mayor	hi:	Mayor sin signo
ge:	Mayor o igual	hs/cs:	Mayor o igual sin signo / con acarreo
lt:	Menor	lo/cc:	Menor sin signo / sin acarreo
le:	Menor o igual	ls:	Menor o igual sin signo
pl:	Positivo o cero	mi:	Negativo
vs:	Overflow	vc:	No desbordamiento
al:	Siempre	nv:	Reservado

Ejecución condicional

Instrucciones que modifican los *flags* de condición

- Comparaciones
- Instrucciones de procesamiento de datos con el sufijo “s”

Ejemplos de uso

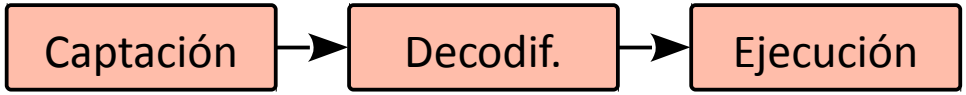
```
movs    r7, r5          @ se modifican los flags según el valor de r7
ldrmi   r2, [r3]        @ se realiza la carga si r7 es negativo
```

```
add     r1, r2, r3
subs    r4, r4, r1      @ se modifican los flags según el resultado
sub     r5, r5, #1
addgt   r6, r6, r4      @ se suma sólo si r4 > 0
```

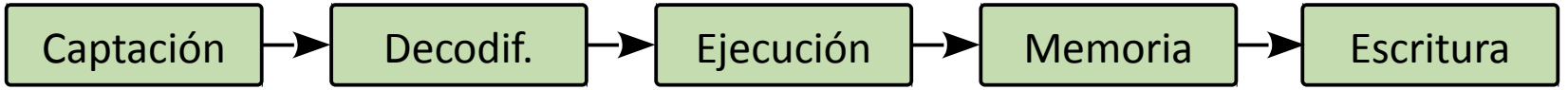
```
cmp     r0, r1          @ se modifican los flags según el resultado
subgt   r0, r0, r1      @ sólo se ejecuta si r0 es mayor que r1
addlt   r1, r1, r0      @ sólo se ejecuta si r0 es menor que r1
```

Implicaciones de la segmentación de cauce

Cauce del ARM7

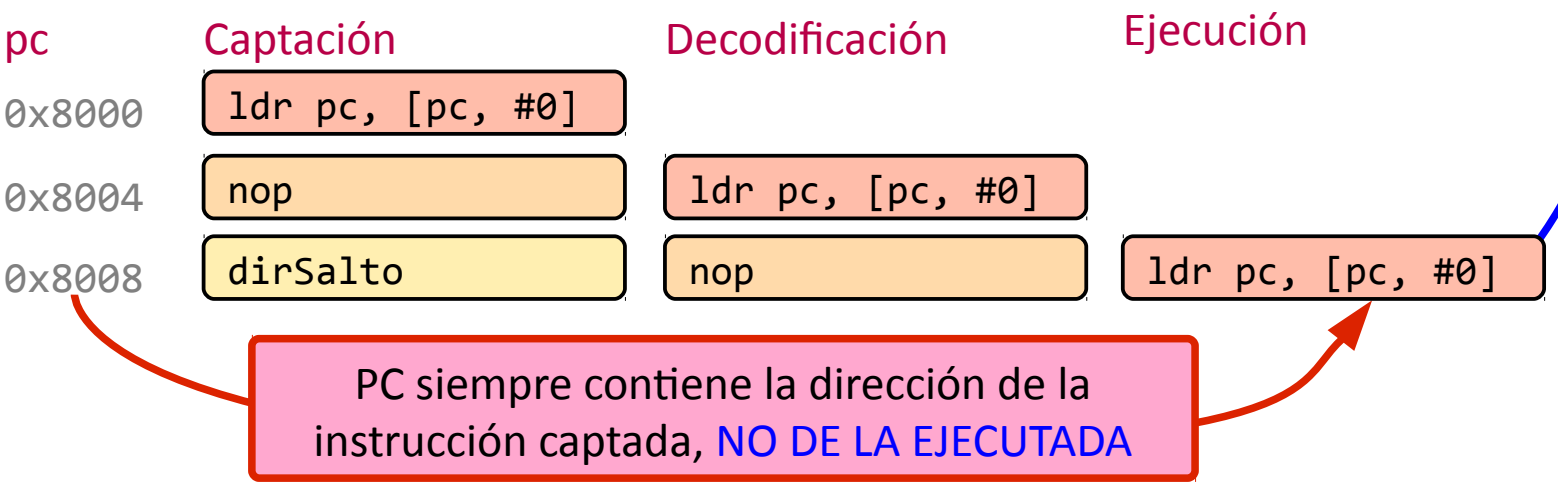


Cauce del ARM9




```
pc ← dirSalto
```

```
0x8000          ldr pc, [pc, #0]   @ pc ← mem<32>[pc]
0x8004          nop
0x8008  .word   dirSalto           @ definimos la dirección del salto
```



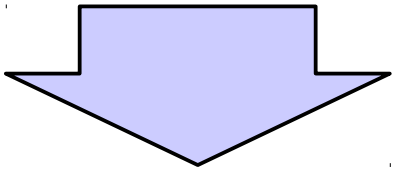
Thumb

Características

- Subconjunto del repertorio ARM de 32 bits codificado sólo con 16 bits
- Optimizado para mejorar la densidad del código C (~65% del tamaño de código ARM)
- Mejora las prestaciones en los sistemas con memoria de 16 bits (más barata)
- Aunque las instrucciones son de 16 bits, los datos (registros) siguen siendo de 32
- El cambio de estado entre ARM y Thumb se realiza mediante la instrucción de salto bx

```
adds r2, r2, #1
```

Instrucción ARM de 32 bits



```
add r2, #1
```

Instrucción Thumb de 16 bits

Diferencias más significativas

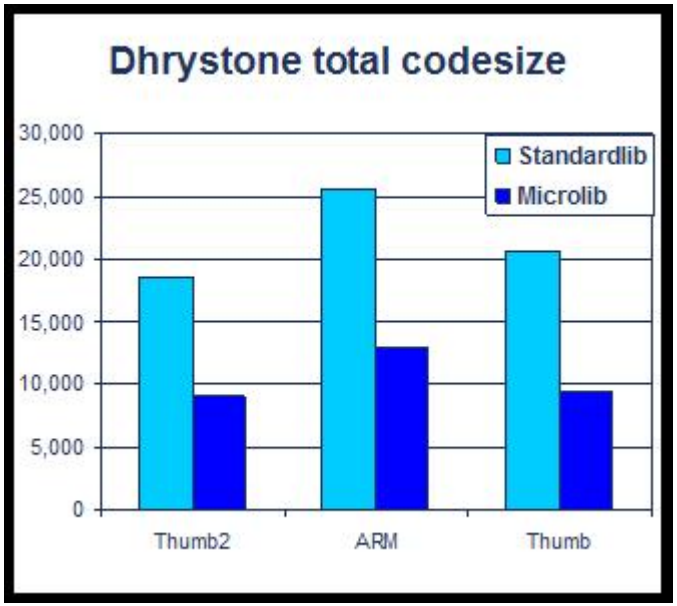
- Todas las instrucciones ALU modifican los flags
- No soporta ejecución condicional
- El registro de destino debe ser uno de los fuentes
- Sólo se usan los registros bajos (r0 – r7)
- Tamaño de los operandos inmediatos limitado a 8 bits
- No se usa el desplazador de bloques

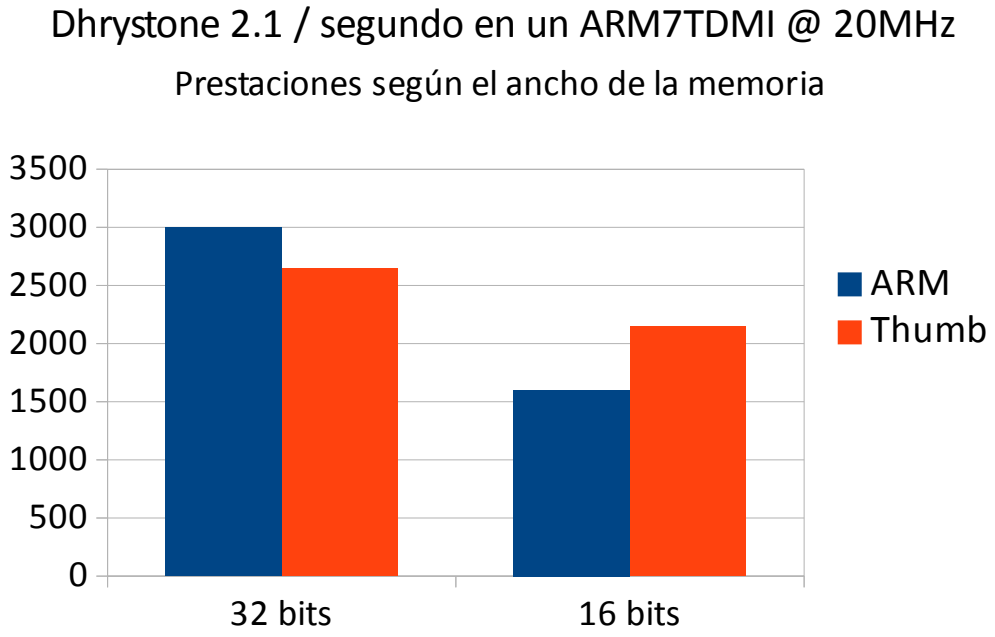
Prestaciones y densidad de código de Thumb y Thumb2

Introducido en la arquitectura ARMv6

Cuándo usar Thumb (Thumb2)

- Si las restricciones de coste son muy altas
  - Necesitamos menos memoria y memoria más barata (de 16 bits)
- Si las prestaciones que obtenemos son suficientes para nuestra aplicación





Fuentes:

ARM. *ARM Compiler*. http://www.arm.com/products/tools/software-tools/rvds/arm-compiler.php

W. Hohl. *ARM Assembly Language. Fundamentals and Techniques*. CRC Press, 2009. Capítulo 13

Contenidos

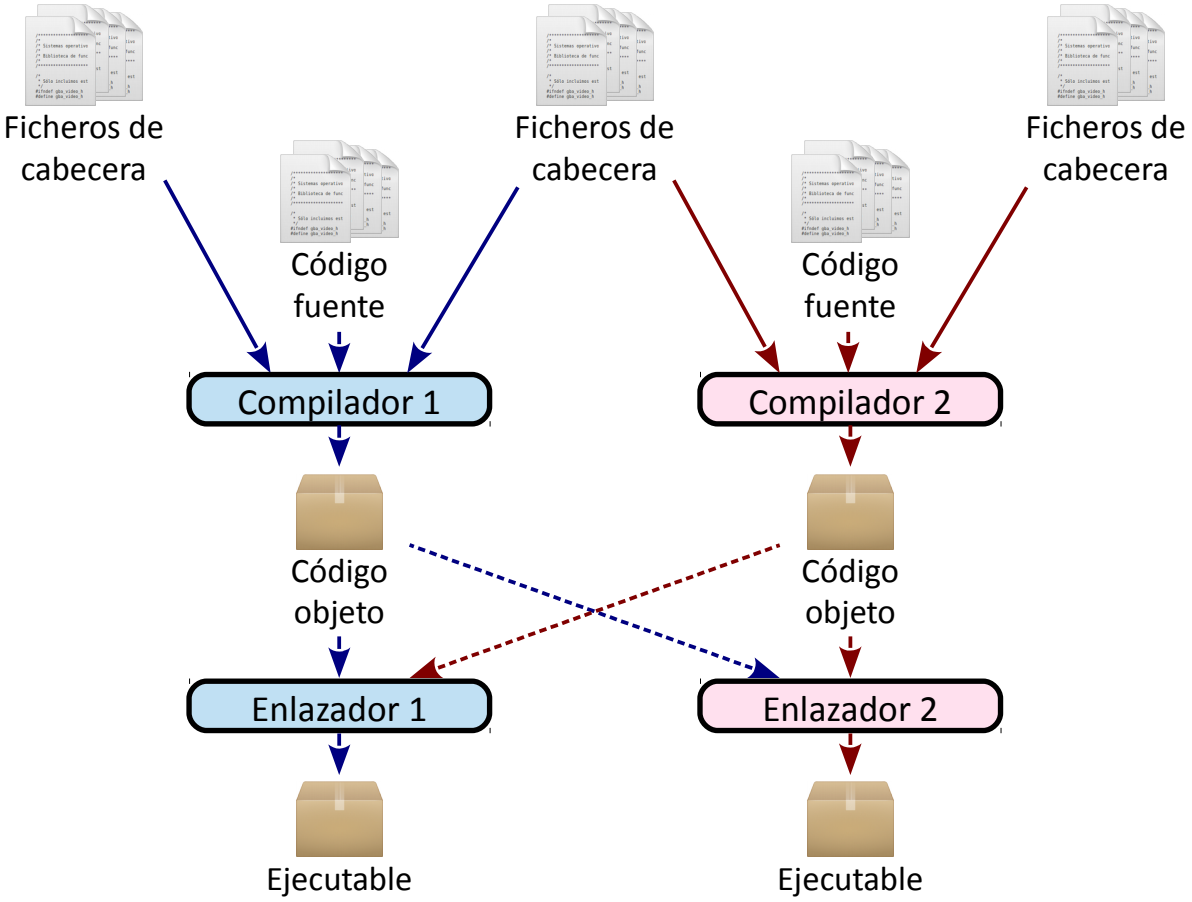
Application Binary Interface (ABI)

Propósito

- Define la interfaz a bajo nivel entre un programa y el sistema operativo u otro programa
- Proporciona inter-operabilidad de ficheros binarios entre diferentes compiladores, plataformas, etc.
- No confundir con API (*Application Programming Interface*)

El ABI define

- Tipos de datos, tamaño y alineación
- Convención de llamadas a funciones
- Cómo hacer las llamadas al sistema
- Formato de los ficheros objeto
- Formato de la información de depuración
- Gestión de excepciones



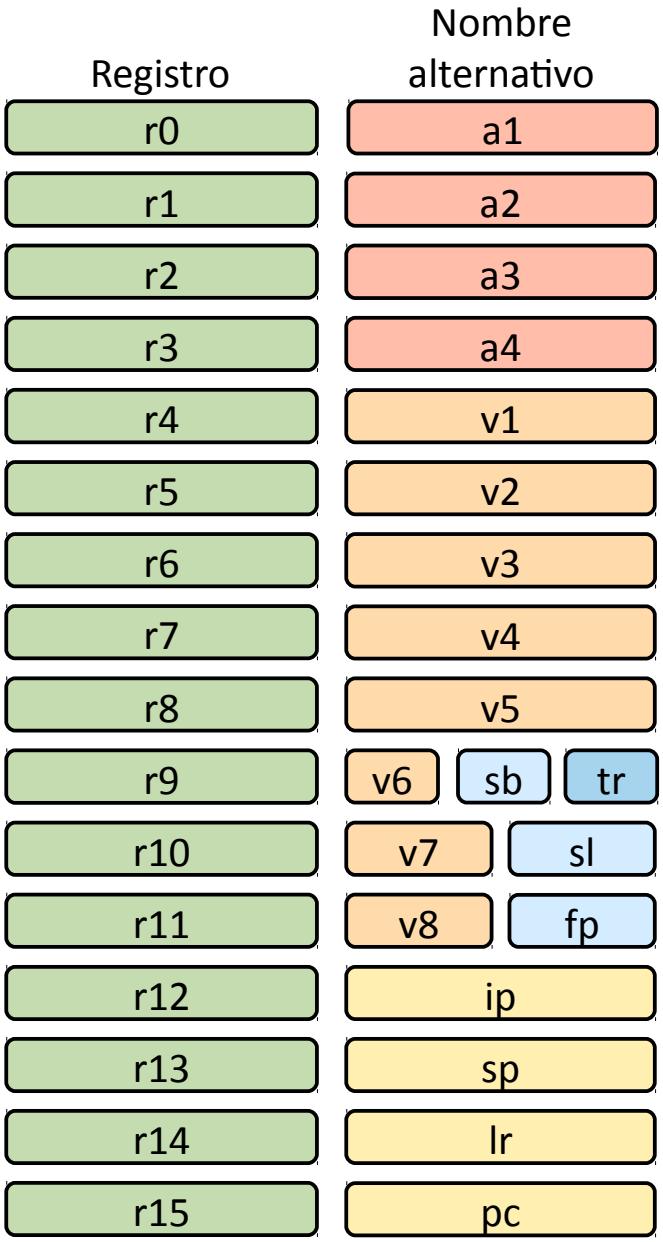
Tipos de dato, tamaños y alineación

Clase	Tipo	Tamaño (bytes)	Alineación (bytes)
Enteros	Unsigned byte	1	1
	Signed byte	1	1
	Unsigned half-word	2	2
	Signed half-word	2	2
	Unsigned word	4	4
	Signed word	4	4
	Unsigned double-word	8	8
	Signed double-word	8	8
Coma flotante	Half precision	2	2
	Single precision (IEEE 754)	4	4
	Double precision (IEEE 754)	8	8
Vector	64-bit vector	8	8
	128-bit vector	16	8
Puntero	Data pointer	4	4
	Code pointer	4	4

ARM Architecture Procedure Call Standard (AAPCS)

Registro	Nombre alternativo	Uso
r0	a1	Argument registers - Se usan para pasar argumentos a la función y devolver su resultado - Si hay más de 4 argumentos se deben pasar por la pila - La función puede corromperlos (no tiene por qué preservarlos)
r1	a2	
r2	a3	
r3	a4	
r4	v1	General Variable registers - Mantienen las variables locales de la función - Si se usan más variables locales, se deben alojar en la pila - La función debe preservarlos antes de usarlos - Según la plataforma o el compilador, los registros v6 – v8 pueden tener usos alternativos
r5	v2	
r6	v3	
r7	v4	
r8	v5	
r9	v6 / sb / tr	
r10	v7 / sl	
r11	v8 / fp	
r12	ip	Intra-procedure call. Usado por el linker para saltos lejanos
r13	sp	Stack pointer. Apunta a una pila *full descending*
r14	lr	Link register. Almacena la dirección de retorno de una función
r15	pc	Program counter

ARM Architecture Procedure Call Standard (AAPCS)



Usos alternativos

r9

- Las plataformas que implementan un modelo de datos independiente de la posición lo usan como *static base* (sb)
- Otras plataformas que implementan un modelo de hebras con almacenamiento local lo usan como *thread register* (tr)
- Si no hay necesidad de estos usos, se puede usar como v6

r10

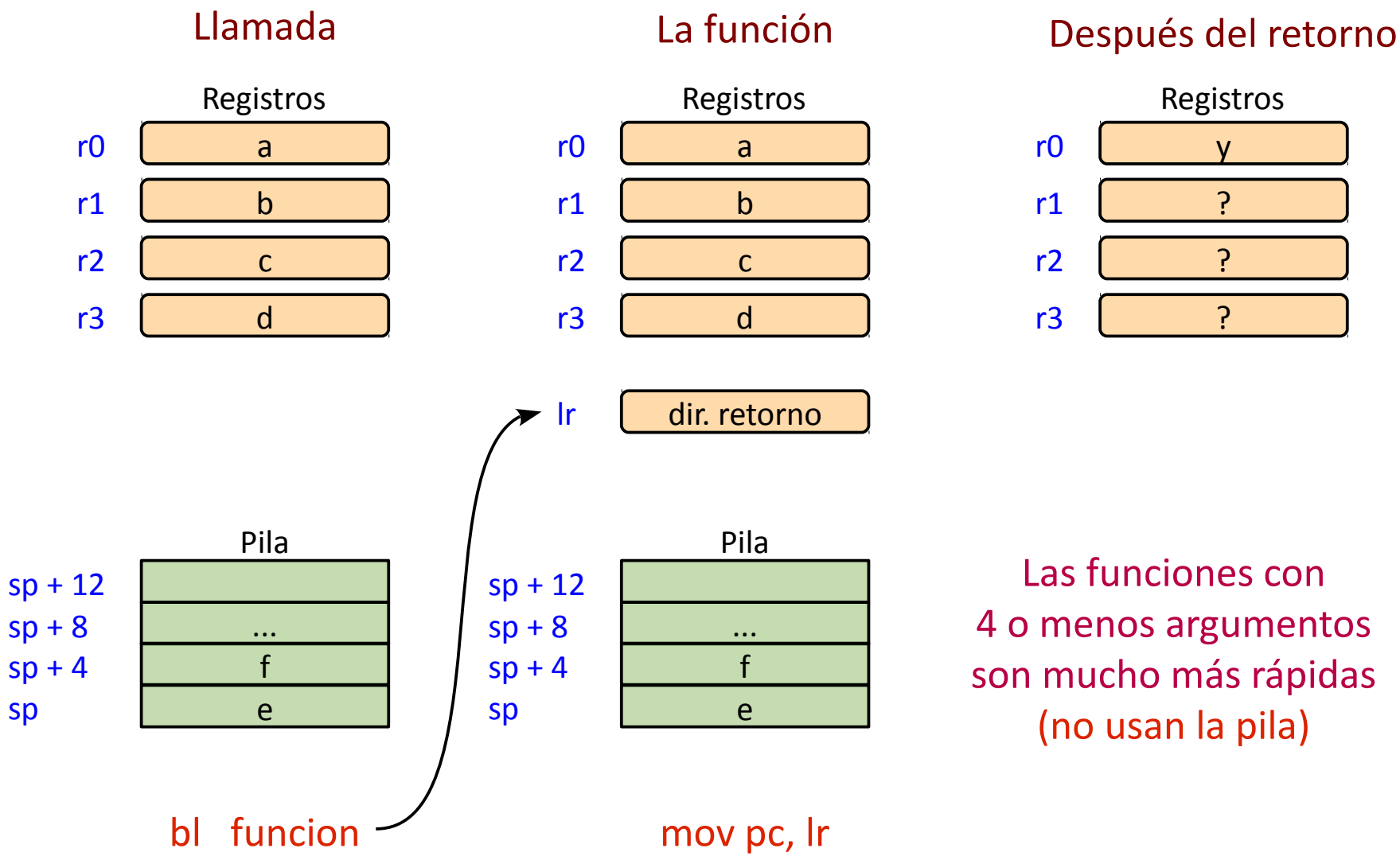
- Algunas plataformas lo usan para gestionar el tamaño de la pila dinámicamente, *stack limit* (sl)
- Si no hay necesidad de estos usos, se puede usar como v7

r11

- Algunas compiladores lo usan para gestionar el marco de la pila, *frame pointer* (fp)
- Si no hay necesidad de estos usos, se puede usar como v8

Ejemplo de llamada a función

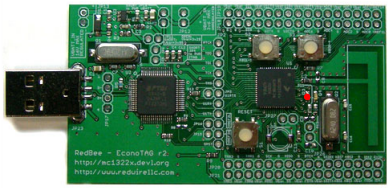
y = funcion(a, b, c, d, e, f);



Las funciones con
4 o menos argumentos
son mucho más rápidas
(no usan la pila)

Contenidos

El Hola Mundo en la Redwire Econotag



```
@ Constantes
    .set GPIO_PAD_DIR1,     0x80000004
    .set GPIO_DATA_SET1,    0x8000004c
    .set GPIO_DATA_RESET1, 0x80000054
    .set LED_RED_MASK,      0x00001000
    .set DELAY,             0x00200000

@ Punto de entrada
    .text
    .global _start
    .type   _start, %function

_start:
    @ El pin del led debe ser de salida
    ldr     r4, =GPIO_PAD_DIR1
    ldr     r5, =LED_RED_MASK
    str     r5, [r4]

    @ Direcciones de los registros de
    @ activación y desactivación del pin
    ldr     r6, =GPIO_DATA_SET1
    ldr     r7, =GPIO_DATA_RESET1
```

```
loop:
    @ Encendemos el led
    str     r5, [r6]

    @ Pausa corta
    ldr     r0, =DELAY
    bl      pause

    @ Apagamos el led
    str     r5, [r7]

    @ Pausa corta
    ldr     r0, =DELAY
    bl      pause

    @ Bucle infinito
    b       loop

@ Función que produce un retardo
@ r0: iteraciones del retardo
    .type   pause, %function
pause:
    subs    r0, r0, #1
    bne     pause
    mov     pc, lr
```

Lecturas recomendadas

Familias y arquitecturas de los procesadores ARM:

ARM. *ARM Processors*. http://www.arm.com/products/processors/

Wikipedia. *ARM Architecture*. http://en.wikipedia.org/wiki/ARM

Arquitectura ARMv4T:

W. Hohl. *ARM Assembly Language. Fundamentals and Techniques*. CRC Press, 2009. Capítulos 1 y 2

A. N. Sloss, D. Symes, C. Wright. *ARM System Developer's Guide*. Morgan Kaufmann, 2004. Capítulos 1 y 2

ARM University Program. *The ARM Instruction Set, v1.0*. http://simplemachines.it/doc/arm_inst.pdf

ARM. *ARM Architecture Reference Manual*, DDI 0100I. Capítulos 1 y 2. http://infocenter.arm.com

Lecturas recomendadas

Thumb:

W. Hohl. *ARM Assembly Language. Fundamentals and Techniques*. CRC Press, 2009.
Capítulo 13

A. N. Sloss, D. Symes, C. Wright. *ARM System Developer's Guide*. Morgan Kaufmann, 2004.
Capítulo 4

J. Lemieux. *Introduction to ARM Thumb*, Beginner's Corner, Embedded.com.
http://www.embedded.com/electronics-blogs/beginner-s-corner/4024632/Introduction-to-ARM-thumb

ARM ABI:

ARM. *ABI for the ARM Architecture*, IHI 0036B.
http://infocenter.arm.com/help/topic/com.arm.doc.ihi0036b/

A. Calderón y N. Castillo. *Why ARM's EABI Matters*. LinuxDevices.com, 2007.
http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/Why-ARMs-EABI-matters/