ARMv7 - Introducción al Lenguaje Ensamblador - ABI - Set de instrucciones

Alejandro Furfaro

20 de julio de 2020

Agenda

1. Lenguaje Ensamblador
   - Estructura de un programa
2. ABI: Aplication Binary Interface
   - Pila y Convención C
3. Set de Instrucciones
   - Conceptos preliminares
   - Instrucciones ARM, Thumb, Thumb-2(T32)
   - Ejecución condicional
   - Instrucciones A32 y T32
   - Acceso a memoria Load - Store
   - Constantes y literal pools
   - Loops y Branches

Archivo de texto plano con tres campos

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

Campo Instrucción

Se compone a su vez de dos sub campos:

- Código de Operación (**Opcode**). Siempre existe
- Operandos. En ocasiones los operandos no existen o están implícitos en el Opcode

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

Campo Instrucción

Se compone a su vez de dos sub campos:

- Código de Operación (**Opcode**). Siempre existe
- Operandos. En ocasiones los operandos no existen o están implícitos en el Opcode

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

Ningún campo es obligatorio

- La línea 2 contiene tan solo un comentario.
- Hay solo dos líneas con etiquetas
- Los comentarios van solo donde aplica comentar algo

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

Ningún campo es obligatorio

- La línea 2 contiene tan solo un comentario.
- Hay solo dos líneas con etiquetas
- Los comentarios van solo donde aplica comentar algo

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

Ningún campo es obligatorio

- La línea 2 contiene tan solo un comentario.
- Hay solo dos líneas con etiquetas
- Los comentarios van solo donde aplica comentar algo

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

Ningún campo es obligatorio

- La línea 2 contiene tan solo un comentario.
- Hay solo dos líneas con etiquetas
- Los comentarios van solo donde aplica comentar algo

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

label

- Sirven para identificar valores numéricos mediante nombres.
- Un valor constante que se define para uso del ensamblador.
- Una dirección de memoria que se desea identificar. Puede corresponder a una variable o a una subrutina.

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

label

- Sirven para identificar valores numéricos mediante nombres.
- Un valor constante que se define para uso del ensamblador.
- Una dirección de memoria que se desea identificar. Puede corresponder a una variable o a una subrutina.

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

label

- Sirven para identificar valores numéricos mediante nombres.
- Un valor constante que se define para uso del ensamblador.
- Una dirección de memoria que se desea identificar. Puede corresponder a una variable o a una subrutina.

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

label

- Sirven para identificar valores numéricos mediante nombres.
- Un valor constante que se define para uso del ensamblador.
- Una dirección de memoria que se desea identificar. Puede corresponder a una variable o a una subrutina.

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

Instrucciones

- Órdenes que el programa le imparte a la CPU.
- Mnemónico: Abreviatura de la acción que realiza la instrucción.
- Operando(s). Indica ubicación de los valores que requiere la instrucción para llevarse a cabo y donde guardará el resultado
- En algunos casos el operando está implícito en la instrucción. En tales casos solo se tiene el Mnemónico.

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

Instrucciones

- Órdenes que el programa le imparte a la CPU.
- Mnemónico: Abreviatura de la acción que realiza la instrucción.
- Operando(s). Indica ubicación de los valores que requiere la instrucción para llevarse a cabo y donde guardará el resultado
- En algunos casos el operando está implícito en la instrucción. En tales casos solo se tiene el Mnemónico.

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

Instrucciones

- Órdenes que el programa le imparte a la CPU.
- Mnemónico: Abreviatura de la acción que realiza la instrucción.
- Operando(s). Indica ubicación de los valores que requiere la instrucción para llevarse a cabo y donde guardará el resultado
- En algunos casos el operando está implícito en la instrucción. En tales casos solo se tiene el Mnemónico.

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

Instrucciones

- Órdenes que el programa le imparte a la CPU.
- Mnemónico: Abreviatura de la acción que realiza la instrucción.
- Operando(s). Indica ubicación de los valores que requiere la instrucción para llevarse a cabo y donde guardará el resultado
- En algunos casos el operando está implícito en la instrucción. En tales casos solo se tiene el Mnemónico.

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

Instrucciones

- Órdenes que el programa le imparte a la CPU.
- Mnemónico: Abreviatura de la acción que realiza la instrucción.
- Operando(s). Indica ubicación de los valores que requiere la instrucción para llevarse a cabo y donde guardará el resultado
- En algunos casos el operando está implícito en la instrucción. En tales casos solo se tiene el Mnemónico.

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

Directivas

- No generan código.
- Son órdenes para el ensamblador. No para la CPU
- Ayudan a definir constantes, variables, e información útil para que el ensamblador “arme” la imagen binaria del programa.
- Varían según el ensamblador que utilicemos. En nuestro caso gnu Assembler (***as***)

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

Directivas

- No generan código.
- Son órdenes para el ensamblador. No para la CPU
- Ayudan a definir constantes, variables, e información útil para que el ensamblador “arme” la imagen binaria del programa.
- Varían según el ensamblador que utilicemos. En nuestro caso gnu Assembler (***as***)

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

Directivas

- No generan código.
- Son órdenes para el ensamblador. No para la CPU
- Ayudan a definir constantes, variables, e información útil para que el ensamblador “arme” la imagen binaria del programa.
- Varían según el ensamblador que utilicemos. En nuestro caso gnu Assembler (***as***)

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

Directivas

- No generan código.
- Son órdenes para el ensamblador. No para la CPU
- Ayudan a definir constantes, variables, e información útil para que el ensamblador “arme” la imagen binaria del programa.
- Varían según el ensamblador que utilicemos. En nuestro caso gnu Assembler (***as***)

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

Directivas

- No generan código.
- Son órdenes para el ensamblador. No para la CPU
- Ayudan a definir constantes, variables, e información útil para que el ensamblador “arme” la imagen binaria del programa.
- Varían según el ensamblador que utilicemos. En nuestro caso gnu Assembler (***as***)

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

Comentarios

- No generan código.
- Inician con el caracter ';'.
- Bien empleados son la documentación de tu proyecto (doxygen por ejemplo)

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

Comentarios

- No generan código.
- Inician con el caracter ';'.
- Bien empleados son la documentación de tu proyecto (doxygen por ejemplo)

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

Comentarios

- No generan código.
- Inician con el caracter ';'.
- Bien empleados son la documentación de tu proyecto (doxygen por ejemplo)

Archivo de texto plano con tres campos

{label} {instrucción |directiva |pseudo-instrucción} {;comentario}

```
1          .section .text
2                   ; Nombre de la sección de código
3          .equ   VALUE, 3
4  start:  MOV   r0, #10   ; Establece los operandos
5          MOV   r1, VALUE
6          ADD   r0,r0,r1  ; r0 = r0 + r1
7  stop:   WFI
8          B     stop      ; bucle infinito
```

Comentarios

- No generan código.
- Inician con el caracter ';'.
- Bien empleados son la documentación de tu proyecto (doxygen por ejemplo)

Directivas - Pseudoinstrucciones

- Definiendo variables...
  - `.byte`, `.hword`, `.word`, `.quad`,
- directivas
  - `.equ`, para definir constantes que después no quedan en el archivo objeto.
  - `.align`, Deja un espacio de direcciones sin usar hasta la próxima dirección múltiplo del valor que se le establece como argumento.
  - `.fill`, repite una cantidad de veces la instrucción que le sigue.
- expresiones
  - `.`, se evalúa en la posición en memoria al principio de la línea que contiene la expresión. Es decir, es la dirección actual en la que se encuentra trabajando ***as***.

Directivas - Pseudoinstrucciones

- Definiendo variables...
  - `.byte`, `.hword`, `.word`, `.quad`,
- directivas
  - `.equ`, para definir constantes que después no quedan en el archivo objeto.
  - `.align`, Deja un espacio de direcciones sin usar hasta la próxima dirección múltiplo del valor que se le establece como argumento.
  - `.fill`, repite una cantidad de veces la instrucción que le sigue.
- expresiones
  - `.`, se evalúa en la posición en memoria al principio de la línea que contiene la expresión. Es decir, es la dirección actual en la que se encuentra trabajando ***as***.

Directivas - Pseudoinstrucciones

- Definiendo variables...
  - `.byte`, `.hword`, `.word`, `.quad`,
- directivas
  - `.equ`, para definir constantes que después no quedan en el archivo objeto.
  - `.align`, Deja un espacio de direcciones sin usar hasta la próxima dirección múltiplo del valor que se le establece como argumento.
  - `.fill`, repite una cantidad de veces la instrucción que le sigue.
- expresiones
  - `.`, se evalúa en la posición en memoria al principio de la línea que contiene la expresión. Es decir, es la dirección actual en la que se encuentra trabajando ***as***.

Directivas - Pseudoinstrucciones

- Definiendo variables...
  - `.byte`, `.hword`, `.word`, `.quad`,
- directivas
  - `.equ`, para definir constantes que después no quedan en el archivo objeto.
  - `.align`, Deja un espacio de direcciones sin usar hasta la próxima dirección múltiplo del valor que se le establece como argumento.
  - `.fill`, repite una cantidad de veces la instrucción que le sigue.
- expresiones
  - `.`, se evalúa en la posición en memoria al principio de la línea que contiene la expresión. Es decir, es la dirección actual en la que se encuentra trabajando ***as***.

Directivas - Pseudoinstrucciones

- Definiendo variables...
  - `.byte`, `.hword`, `.word`, `.quad`,
- directivas
  - `.equ`, para definir constantes que después no quedan en el archivo objeto.
  - `.align`, Deja un espacio de direcciones sin usar hasta la próxima dirección múltiplo del valor que se le establece como argumento.
  - `.fill`, repite una cantidad de veces la instrucción que le sigue.
- expresiones
  - `.`, se evalúa en la posición en memoria al principio de la línea que contiene la expresión. Es decir, es la dirección actual en la que se encuentra trabajando ***as***.

Directivas - Pseudoinstrucciones

- Definiendo variables...
  - `.byte`, `.hword`, `.word`, `.quad`,
- directivas
  - `.equ`, para definir constantes que después no quedan en el archivo objeto.
  - `.align`, Deja un espacio de direcciones sin usar hasta la próxima dirección múltiplo del valor que se le establece como argumento.
  - `.fill`, repite una cantidad de veces la instrucción que le sigue.
- expresiones
  - `.`, se evalúa en la posición en memoria al principio de la línea que contiene la expresión. Es decir, es la dirección actual en la que se encuentra trabajando **`as`**.

Directivas - Pseudoinstrucciones

- Definiendo variables...
  - `.byte`, `.hword`, `.word`, `.quad`,
- directivas
  - `.equ`, para definir constantes que después no quedan en el archivo objeto.
  - `.align`, Deja un espacio de direcciones sin usar hasta la próxima dirección múltiplo del valor que se le establece como argumento.
  - `.fill`, repite una cantidad de veces la instrucción que le sigue.
- expresiones
  - `.`, se evalúa en la posición en memoria al principio de la línea que contiene la expresión. Es decir, es la dirección actual en la que se encuentra trabajando **`as`**.

Directivas - Pseudoinstrucciones

- Definiendo variables...
  - `.byte`, `.hword`, `.word`, `.quad`,
- directivas
  - `.equ`, para definir constantes que después no quedan en el archivo objeto.
  - `.align`, Deja un espacio de direcciones sin usar hasta la próxima dirección múltiplo del valor que se le establece como argumento.
  - `.fill`, repite una cantidad de veces la instrucción que le sigue.
- expresiones
  - `.`, se evalúa en la posición en memoria al principio de la línea que contiene la expresión. Es decir, es la dirección actual en la que se encuentra trabajando **`as`**.

Directivas - Pseudoinstrucciones

- Definiendo variables...
  - `.byte`, `.hword`, `.word`, `.quad`,
- directivas
  - `.equ`, para definir constantes que después no quedan en el archivo objeto.
  - `.align`, Deja un espacio de direcciones sin usar hasta la próxima dirección múltiplo del valor que se le establece como argumento.
  - `.fill`, repite una cantidad de veces la instrucción que le sigue.
- expresiones
  - `.`, se evalúa en la posición en memoria al principio de la línea que contiene la expresión. Es decir, es la dirección actual en la que se encuentra trabajando ***as***.

Debugger

GDB + DDD

Comandos Básicos

`r | run`	Ejecuta el programa hasta el primer break
`b | break FILE:LINE`	Breakpoint en la línea
`b | break FUNCTION`	Breakpoint en la función
`info breakpoints`	Muestra información sobre los breakpoints
`c | continue`	Continúa con la ejecución
`s | step`	Siguiente línea (Into)
`n | next`	Siguiente línea (Over)
`si | stepi`	Siguiente instrucción asm (Into)
`ni | nexti`	Siguiente instrucción asm (Over)
`x/Nuf ADDR`	Muestra los datos en memoria

`N` = Cantidad (bytes)

`u` = Unidad `b|h|w|g`

`b:byte, h:half word, w:word, g:dword`

`f` = Formato `x|d|u|o|f|a`

`x`:hex, `d`:decimal, `u`:decimal sin signo, `o`:octal, `f`:float, `a`:direcciones

GDB

Configuración de GDB:

```
~/.gdbinit
```

Para usar sintaxis intel y guardar historial de comandos:

```
set disassembly-flavor intel
set history save
```

Correr `GDB` con argumentos:

```
gdb --args <ejecutable> <arg1> <arg2> ...
```

¿Que es ABI?

ABI (Application Binary Interface)

Es la especificación de una convención de llamadas entre dos módulos de Software, a nivel de lenguaje de máquina, fundamental para interfacear llamadas a funciones de bajo nivel desde lenguajes de alto nivel o viceversa, e imprescindible para que puedan diseñarse compiladores de lenguajes que trabajen de un modo predecible.

- De esta forma las funciones pueden ser llamadas sin tener en cuenta como fueron implementadas.
- La convención define:
  - Como las funciones reciben parametros.
  - Como las funciones retornan el resultado.
  - Que registros se deben preservar en una función.
- Las convenciones dependen de la arquitectura del procesador:
  - En x86 (32bits) se conoce como cdecl.
  - En x86-64 (64bits) se conoce como System V AMD64 ABI.
  - En ARMv7 esta la Procedure Call Standard for the ARM Architecture.

¿Que es ABI?

ABI (**A**pplication **B**inary **I**nterface)

Es la especificación de una convención de llamadas entre dos módulos de Software, a nivel de lenguaje de máquina, fundamental para interfacear llamadas a funciones de bajo nivel desde lenguajes de alto nivel o viceversa, e imprescindible para que puedan diseñarse compiladores de lenguajes que trabajen de un modo predecible.

- De esta forma las funciones pueden ser llamadas sin tener en cuenta como fueron implementadas.
- La convención define:
  - Como las funciones reciben parametros.
  - Como las funciones retornan el resultado.
  - Que registros se deben preservar en una llamada.
- Las convenciones dependen de la arquitectura del procesador:
  - En x86 (32bits) se conoce como cdecl.
  - En x86-64 (64bits) se conocen como System V AMD64 ABI.
  - En ARMv7 (32bits) la convención se llama AAPCS (ARM Architecture Procedure Call Standard).

¿Que es ABI?

ABI (**A**pplication **B**inary **I**nterface)

Es la especificación de una convención de llamadas entre dos módulos de Software, a nivel de lenguaje de máquina, fundamental para interfacear llamadas a funciones de bajo nivel desde lenguajes de alto nivel o viceversa, e imprescindible para que puedan diseñarse compiladores de lenguajes que trabajen de un modo predecible.

- De esta forma las funciones pueden ser llamadas sin tener en cuenta como fueron implementadas.
- La convención define:
  - Como las funciones reciben parámetros.
  - Como las funciones retornan el resultado.
  - Que registros se deben preservar en una función.
- Las convenciones dependen de la arquitectura del procesador:
  - En x86 (32bits) se pasan parámetros por stack.
  - En x86-64 (64bits) se pasan por registros (en Linux).
  - En ARMv7 (32bits) se pasan por registros.

¿Que es ABI?

ABI (**A**pplication **B**inary **I**nterface)

Es la especificación de una convención de llamadas entre dos módulos de Software, a nivel de lenguaje de máquina, fundamental para interfacear llamadas a funciones de bajo nivel desde lenguajes de alto nivel o viceversa, e imprescindible para que puedan diseñarse compiladores de lenguajes que trabajen de un modo predecible.

- De esta forma las funciones pueden ser llamadas sin tener en cuenta como fueron implementadas.
- La convención define:
  - ✓ Como las funciones reciben parámetros.
  - ✓ Como las funciones retornan el resultado.
  - ✓ Que registros se deben preservar en una función.
- Las convenciones dependen de la arquitectura del procesador:
  - ✓ En x86 (32bits) se pasan parámetros por la pila.
  - ✓ En x86-64 (64bits) se pasan parámetros por registros.
  - ✓ En ARMv7 (32bits) se pasan parámetros por registros.

¿Que es ABI?

ABI (**A**pplication **B**inary **I**nterface)

Es la especificación de una convención de llamadas entre dos módulos de Software, a nivel de lenguaje de máquina, fundamental para interfacear llamadas a funciones de bajo nivel desde lenguajes de alto nivel o viceversa, e imprescindible para que puedan diseñarse compiladores de lenguajes que trabajen de un modo predecible.

- De esta forma las funciones pueden ser llamadas sin tener en cuenta como fueron implementadas.
- La convención define:
  - ✓ Como las funciones reciben parámetros.
  - ✓ Como las funciones retornan el resultado.
  - ✓ Que registros se deben preservar en una función.
- Las convenciones dependen de la arquitectura del procesador:
  - ✓ En x86 (32bits) se pasan parámetros por stack.
  - ✓ En x86-64 (64bits) se pasan parámetros por registros.
  - ✓ En ARMv7 (32bits) se pasan parámetros por registros.

¿Que es ABI?

ABI (**A**pplication **B**inary **I**nterface)

Es la especificación de una convención de llamadas entre dos módulos de Software, a nivel de lenguaje de máquina, fundamental para interfacear llamadas a funciones de bajo nivel desde lenguajes de alto nivel o viceversa, e imprescindible para que puedan diseñarse compiladores de lenguajes que trabajen de un modo predecible.

- De esta forma las funciones pueden ser llamadas sin tener en cuenta como fueron implementadas.
- La convención define:
  - ✓ Como las funciones reciben parámetros.
  - ✓ Como las funciones retornan el resultado.
  - ✓ Que registros se deben preservar en una función.
- Las convenciones dependen de la arquitectura del procesador:
  - ✓ En x86 (32bits) se pasan parámetros por stack.
  - ✓ En x86-64 (64bits) se pasan parámetros por registros.
  - ✓ En ARMv7 (32bits) se pasan parámetros por registros.

¿Que es ABI?

ABI (**A**pplication **B**inary **I**nterface)

Es la especificación de una convención de llamadas entre dos módulos de Software, a nivel de lenguaje de máquina, fundamental para interfacear llamadas a funciones de bajo nivel desde lenguajes de alto nivel o viceversa, e imprescindible para que puedan diseñarse compiladores de lenguajes que trabajen de un modo predecible.

- De esta forma las funciones pueden ser llamadas sin tener en cuenta como fueron implementadas.
- La convención define:
  - ✓ Como las funciones reciben parámetros.
  - ✓ Como las funciones retornan el resultado.
  - ✓ Que registros se deben preservar en una función.
- Las convenciones dependen de la arquitectura del procesador:
  - ✓ En x86 (32bits) se conoce como x32 ABI.
  - ✓ En x86-64 (64bits) se conoce como System V ABI.
  - ✓ En ARMv7 (32bits) se conoce como EABI.

¿Que es ABI?

ABI (Application Binary Interface)

Es la especificación de una convención de llamadas entre dos módulos de Software, a nivel de lenguaje de máquina, fundamental para interfacear llamadas a funciones de bajo nivel desde lenguajes de alto nivel o viceversa, e imprescindible para que puedan diseñarse compiladores de lenguajes que trabajen de un modo predecible.

- De esta forma las funciones pueden ser llamadas sin tener en cuenta como fueron implementadas.
- La convención define:
  - ✓ Como las funciones reciben parámetros.
  - ✓ Como las funciones retornan el resultado.
  - ✓ Que registros se deben preservar en una función.
- Las convenciones dependen de la arquitectura del procesador:
  - ✓ En `x86` (32bits) se conoce como `x32 ABI`.
  - ✓ En `x86-64` (64bits) se denomina `System V AMD64 ABI`.
  - ✓ En `ARMv7` es la `Procedure Call Standard for the Arm Architecture`.

¿Que es ABI?

ABI (Application Binary Interface)

Es la especificación de una convención de llamadas entre dos módulos de Software, a nivel de lenguaje de máquina, fundamental para interfacear llamadas a funciones de bajo nivel desde lenguajes de alto nivel o viceversa, e imprescindible para que puedan diseñarse compiladores de lenguajes que trabajen de un modo predecible.

- De esta forma las funciones pueden ser llamadas sin tener en cuenta como fueron implementadas.
- La convención define:
  - ✓ Como las funciones reciben parámetros.
  - ✓ Como las funciones retornan el resultado.
  - ✓ Que registros se deben preservar en una función.
- Las convenciones dependen de la arquitectura del procesador:
  - ✓ En `x86` (32bits) se conoce como `x32 ABI`.
  - ✓ En `x86-64` (64bits) se denomina `System V AMD64 ABI`.
  - ✓ En `ARMv7` es la `Procedure Call Standard for the Arm Architecture`.

¿Que es ABI?

ABI (Application Binary Interface)

Es la especificación de una convención de llamadas entre dos módulos de Software, a nivel de lenguaje de máquina, fundamental para interfacear llamadas a funciones de bajo nivel desde lenguajes de alto nivel o viceversa, e imprescindible para que puedan diseñarse compiladores de lenguajes que trabajen de un modo predecible.

- De esta forma las funciones pueden ser llamadas sin tener en cuenta como fueron implementadas.
- La convención define:
  - ✓ Como las funciones reciben parámetros.
  - ✓ Como las funciones retornan el resultado.
  - ✓ Que registros se deben preservar en una función.
- Las convenciones dependen de la arquitectura del procesador:
  - ✓ En `x86` (32bits) se conoce como `x32 ABI`.
  - ✓ En `x86-64` (64bits) se denomina `System V AMD64 ABI`.
  - ✓ En `ARMv7` es la `Procedure Call Standard for the Arm Architecture`.

¿Que es ABI?

ABI (Application Binary Interface)

Es la especificación de una convención de llamadas entre dos módulos de Software, a nivel de lenguaje de máquina, fundamental para interfacear llamadas a funciones de bajo nivel desde lenguajes de alto nivel o viceversa, e imprescindible para que puedan diseñarse compiladores de lenguajes que trabajen de un modo predecible.

- De esta forma las funciones pueden ser llamadas sin tener en cuenta como fueron implementadas.
- La convención define:
  - ✓ Como las funciones reciben parámetros.
  - ✓ Como las funciones retornan el resultado.
  - ✓ Que registros se deben preservar en una función.
- Las convenciones dependen de la arquitectura del procesador:
  - ✓ En `x86` (32bits) se conoce como `x32 ABI`.
  - ✓ En `x86-64` (64bits) se denomina `System V AMD64 ABI`.
  - ✓ En `ARMv7` es la `Procedure Call Standard for the Arm Architecture`.

Tipos de Datos Fundamentales

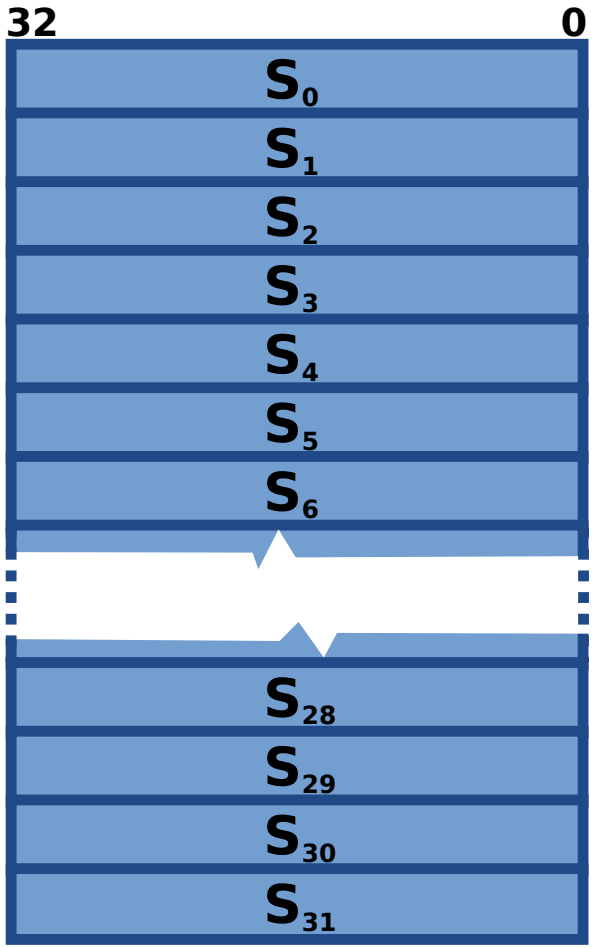
Type Class	Machine Type	Byte size	Byte align-ment	Note
Integral	Unsigned byte	1	1	Character
	Signed byte	1	1	
	Unsigned half-word	2	2	
	Signed half-word	2	2	
	Unsigned word	4	4	
	Signed word	4	4	
	Unsigned double-word	8	8	
	Signed double-word	8	8	
Floating Point	Half precision	2	2	See *Half-precision Floating Point* (page 14).
	Single precision (IEEE 754)	4	4	The encoding of floating point numbers is described in [ARMARM[5]] chapter C2, *VFP Programmer's Model*, §2.1.1 *Single-precision format*, and §2.1.2 *Double-precision format*.
	Double precision (IEEE 754)	8	8	
Containterized vector	64-bit vector	8	8	See *Containerized Vectors* (page 15).
	128-bit vector	16	8	
Pointer	Data pointer	4	4	Pointer arithmetic should be unsigned. Bit 0 of a code pointer indicates the target instruction set type (0 Arm, 1 Thumb).
	Code pointer	4	4	

Uso de los registros

Reg.	Alias	Especial	Rol en la llamada
R0	arg1		Registro scratch Argumento 1 / resultado
R1	arg2		Registro scratch Argumento 2 / resultado
R2	arg3		Registro scratch Argumento 3
R3	arg4		Registro scratch Argumento 4
R4	var1		Registro variable 1
R5	var2		Registro variable 2
R6	var3		Registro variable 3
R7	var4		Registro variable 4
R8	var5		Registro variable 5
R9	v6 SB TR		Platform defined
R10	var7		Registro variable 7
R11	var8	FP	Frame Pointer o Variable 8
R12		IP	Intra Procedure Call Scratch Register
R13		SP	Stack Pointer
R14		LR	Link Register
R15		PC	Program Counter

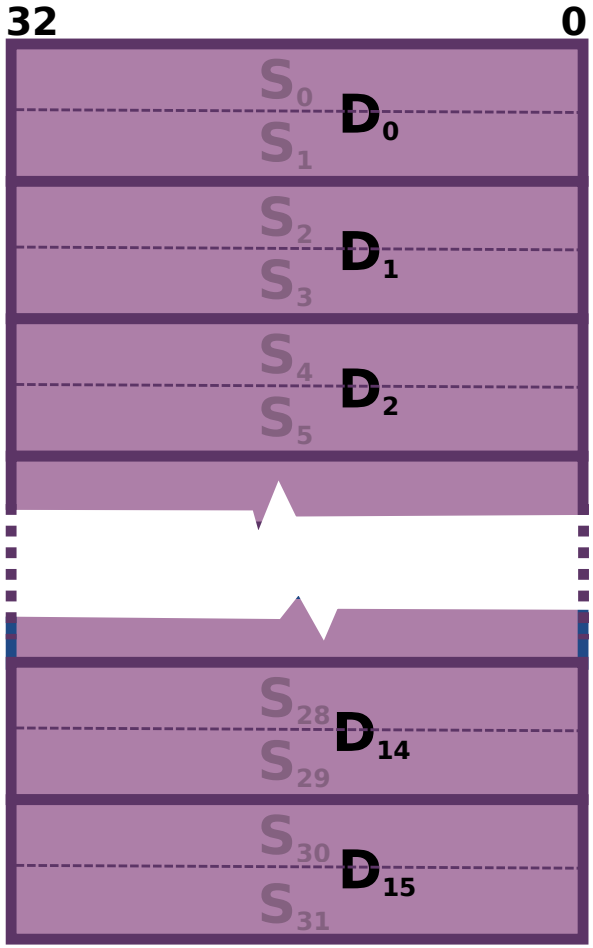
Registros de Coprocesadores

La extensión VFP-v2, se basa en un co procesador que introduce un set de 32 registros de 32 bits para almacenar números en Punto Flotante simple precisión en formato IEEE-754.



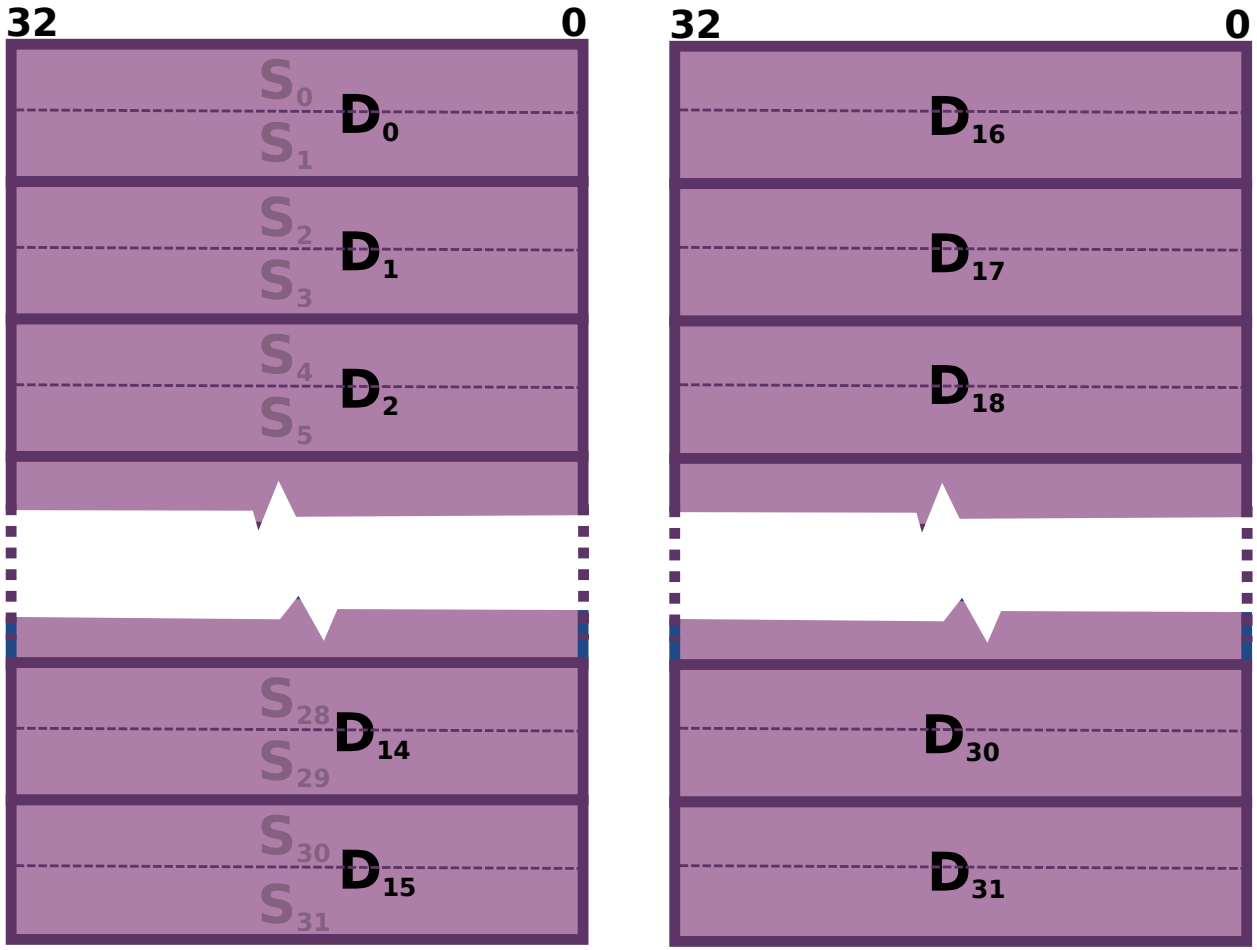
Registros de las extensiones

Además VFP-v2 permite utilizar estos mismos registros de 32 bits para almacenar números en Punto Flotante doble precisión como 16 registros de 64 bits en formato IEEE-754.



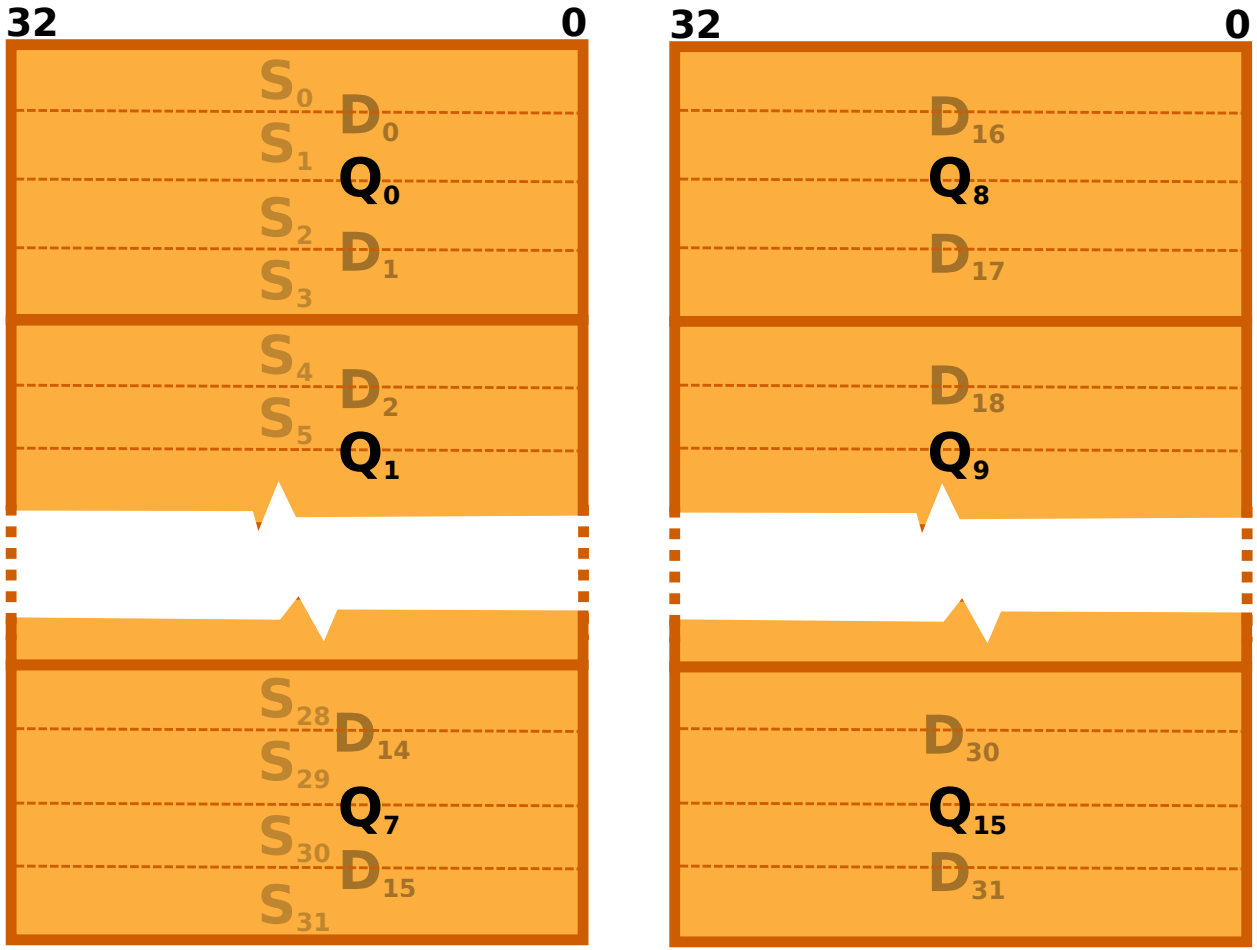
Registros de las extensiones

Con la VFPv-3 se agregan otros 16 registros de 64 bits, para Punto Flotante doble precisión y para datos empaquetados en las Extensiones avanzadas SIMD (NEON).



Registros de las extensiones

Las Extensiones avanzadas SIMD (NEON) pueden trabajar con datos empaquetados de 128 bits agrupando los 32 registros de 64 bits en 16 registros de 128 bits.



Tratamiento de los registros de extensiones en ABI

- Los registros `s0` a `s15`, (o `d0` a `d7`, o `q0` a `q3`), no se preservan y pueden ser utilizados para pasaje de argumentos o retorno de resultados en las variantes de procedure call standards.
- Deben ser preservados los registros `s16` a `s31` (o `d8` a `d15`, o `q4` a `q7`).
- De estar presentes, los registros `d16` a `d31`, (o `q8` a `q15`), no necesitan ser preservados.

Tratamiento de los registros de extensiones en ABI

- Los registros **`s0`** a **`s15`**, (o **`d0`** a **`d7`**, o **`q0`** a **`q3`**), no se preservan y pueden ser utilizados para pasaje de argumentos o retorno de resultados en las variantes de procedure call standards.
- Deben ser preservados los registros **`s16`** a **`s31`** (o **`d8`** a **`d15`**, o **`q4`** a **`q7`**).
- De estar presentes, los registros **`d16`** a **`d31`**, (o **`q8`** a **`q15`**), no necesitan ser preservados.

Tratamiento de los registros de extensiones en ABI

- Los registros **`s0`** a **`s15`**, (o **`d0`** a **`d7`**, o **`q0`** a **`q3`**), no se preservan y pueden ser utilizados para pasaje de argumentos o retorno de resultados en las variantes de procedure call standards.

- Deben ser preservados los registros **`s16`** a **`s31`** (o **`d8`** a **`d15`**, o **`q4`** a **`q7`**).

- De estar presentes, los registros `d16` a `d31`, (o `q8` a `q15`), no necesitan ser preservados.

Tratamiento de los registros de extensiones en ABI

- Los registros **`s0`** a **`s15`**, (o **`d0`** a **`d7`**, o **`q0`** a **`q3`**), no se preservan y pueden ser utilizados para pasaje de argumentos o retorno de resultados en las variantes de procedure call standards.

- Deben ser preservados los registros **`s16`** a **`s31`** (o **`d8`** a **`d15`**, o **`q4`** a **`q7`**).

- De estar presentes, los registros **`d16`** a **`d31`**, (o **`q8`** a **`q15`**), no necesitan ser preservados.

La pila en llamadas a procedimientos

- ARM define restricciones a observar en el uso de la pila, algunas son de mínima y se amplían cuando se pretende universalizar la interfaz de programación (ABI).
- El Stack debe ser un área contigua y es full descending: Crece desde una Dirección Base hacia una Dirección Tope
- Alineación: ARM define dos criterios:
  - Restricción Universal: $sp \bmod 4 = 0$
  - Restricción Interfaz Pública: $sp \bmod 8 = 0$

La pila en llamadas a procedimientos

- ARM define restricciones a observar en el uso de la pila, algunas son de mínima y se amplían cuando se pretende universalizar la interfaz de programación (ABI).
- El Stack debe ser un área contigua y es full descending: Crece desde una Dirección Base hacia una Dirección Tope
- Alineación: ARM define dos criterios:
  - Restricción Universal: sp mod 4 = 0
  - Restricción Interfaz Pública: sp mod 8 = 0

La pila en llamadas a procedimientos

- ARM define restricciones a observar en el uso de la pila, algunas son de mínima y se amplían cuando se pretende universalizar la interfaz de programación (ABI).
- El Stack debe ser un área contigua y es full descending: Crece desde una Dirección Base hacia una Dirección Tope
- Alineación: ARM define dos criterios:

Restricción Universal	sp mod 4 = 0
Restricción Interfaz Publica	sp mod 8 = 0

La pila en llamadas a procedimientos

- ARM define restricciones a observar en el uso de la pila, algunas son de mínima y se amplían cuando se pretende universalizar la interfaz de programación (ABI).
- El Stack debe ser un área contigua y es full descending: Crece desde una Dirección Base hacia una Dirección Tope
- Alineación: ARM define dos criterios:

Restricción Universal	sp mod 4 = 0
Restricción Interfaz Pública	sp mod 8 = 0

La pila en llamadas a procedimientos

- ARM define restricciones a observar en el uso de la pila, algunas son de mínima y se amplían cuando se pretende universalizar la interfaz de programación (ABI).
- El Stack debe ser un área contigua y es full descending: Crece desde una Dirección Base hacia una Dirección Tope
- Alineación: ARM define dos criterios:

Restricción Universal	sp mod 4 = 0
Restricción Interfaz Pública	sp mod 8 = 0

La pila en llamadas a procedimientos

- ARM define restricciones a observar en el uso de la pila, algunas son de mínima y se amplían cuando se pretende universalizar la interfaz de programación (ABI).
- El Stack debe ser un área contigua y es full descending: Crece desde una Dirección Base hacia una Dirección Tope
- Alineación: ARM define dos criterios:

Restricción Universal	sp mod 4 = 0
Restricción Interfaz Pública	sp mod 8 = 0

Stack frame

- Sabemos a esta altura que una función en C es ejecutada dentro de un **scope**, en el cual tienen validez sus variables locales así como los argumentos que recibe de la función invocante.
- Esto se debe a una estructura fundamental.

Definición

Estructura en memoria constituida por el conjunto de registros preservados, las variables locales y los argumentos pasados desde la función invocante, eventualmente la dirección de retorno (en el caso de ARM no aplica), y una referencia al siguiente Stack Frame en caso de llamadas anidadas.

- La construcción del *stack frame* consiste en colocar un registro base de la pila en una dirección relativa al comienzo del área de la función llamadora. Este se denomina **Frame Pointer**.
- No se especifica el lugar exacto del Stack Frame en el que se ubica el Frame Pointer

Stack frame

- Sabemos a esta altura que una función en C es ejecutada dentro de un **scope**, en el cual tienen validez sus variables locales así como los argumentos que recibe de la función invocante.
- Esto se debe a una estructura fundamental.

Stack frame:

Estructura en memoria constituida por el conjunto de registros preservados, las variables locales y los argumentos pasados desde la función invocante, eventualmente la dirección de retorno (en el caso de ARM no aplica), y una referencia al siguiente Stack Frame en caso de llamadas anidadas.

- La construcción del *stack frame* consiste en colocar un registro base de la pila en una dirección relativa al comienzo del área de la función llamadora. Este se denomina **Frame Pointer**.
- No se especifica el lugar exacto del Stack Frame en el que se ubica el Frame Pointer

Stack frame

- Sabemos a esta altura que una función en C es ejecutada dentro de un **scope**, en el cual tienen validez sus variables locales así como los argumentos que recibe de la función invocante.
- Esto se debe a una estructura fundamental.

stack frame

Estructura en memoria constituida por el conjunto de registros preservados, las variables locales y los argumentos pasados desde la función invocante, eventualmente la dirección de retorno (en el caso de ARM no aplica), y una referencia al siguiente Stack Frame en caso de llamadas anidadas.

- La construcción del *stack frame* consiste en colocar un registro base de la pila en una dirección relativa al comienzo del área de la función llamadora. Este se denomina **Frame Pointer**.
- No se especifica el lugar exacto del Stack Frame en el que se ubica el Frame Pointer

Stack frame

- Sabemos a esta altura que una función en C es ejecutada dentro de un **scope**, en el cual tienen validez sus variables locales así como los argumentos que recibe de la función invocante.
- Esto se debe a una estructura fundamental.

stack frame

Estructura en memoria constituida por el conjunto de registros preservados, las variables locales y los argumentos pasados desde la función invocante, eventualmente la dirección de retorno (en el caso de ARM no aplica), y una referencia al siguiente Stack Frame en caso de llamadas anidadas.

- La construcción del *stack frame* consiste en colocar un registro base de la pila en una dirección relativa al comienzo del área de la función llamadora. Este se denomina **Frame Pointer**.
- No se especifica el lugar exacto del Stack Frame en el que se ubica el Frame Pointer

Stack frame

- Sabemos a esta altura que una función en C es ejecutada dentro de un **scope**, en el cual tienen validez sus variables locales así como los argumentos que recibe de la función invocante.
- Esto se debe a una estructura fundamental.

stack frame

Estructura en memoria constituida por el conjunto de registros preservados, las variables locales y los argumentos pasados desde la función invocante, eventualmente la dirección de retorno (en el caso de ARM no aplica), y una referencia al siguiente Stack Frame en caso de llamadas anidadas.

- La construcción del *stack frame* consiste en colocar un registro base de la pila en una dirección relativa al comienzo del área de la función llamadora. Este se denomina **Frame Pointer**.
- No se especifica el lugar exacto del Stack Frame en el que se ubica el Frame Pointer

Stack frame

- Sabemos a esta altura que una función en C es ejecutada dentro de un **scope**, en el cual tienen validez sus variables locales así como los argumentos que recibe de la función invocante.
- Esto se debe a una estructura fundamental.

stack frame

Estructura en memoria constituida por el conjunto de registros preservados, las variables locales y los argumentos pasados desde la función invocante, eventualmente la dirección de retorno (en el caso de ARM no aplica), y una referencia al siguiente Stack Frame en caso de llamadas anidadas.

- La construcción del *stack frame* consiste en colocar un registro base de la pila en una dirección relativa al comienzo del área de la función llamadora. Este se denomina **Frame Pointer**.
- No se especifica el lugar exacto del Stack Frame en el que se ubica el Frame Pointer

Llamada

- Se usa la instrucción **BL** (**B**ranch with **L**ink).
- Transfiere a `LR[31:1]` los 31 MSB de la dirección de la instrucción sucesora secuencial (es decir, la que sigue a BL en la secuencia de instrucciones del programa).
- El LSB del LR toma un estado acorde al Modo del procesador: `LR[0]='1'` para Modo Thumb y LR[0]='0' si está en Modo ARM.
- Luego el Program Counter toma el valor de la dirección de salto contenida en la instrucción. Los rangos de salto dependen del modo de trabajo.

Llamada

- Se usa la instrucción **BL** (**B**ranch with **L**ink).
- Transfiere a `LR[31:1]` los 31 MSB de la dirección de la instrucción sucesora secuencial (es decir, la que sigue a BL en la secuencia de instrucciones del programa).
- El LSB del LR toma un estado acorde al Modo del procesador: `LR[0]='1'` para Modo Thumb y LR[0]='0' si está en Modo ARM.
- Luego el Program Counter toma el valor de la dirección de salto contenida en la instrucción. Los rangos de salto dependen del modo de trabajo.
  - Modo ARM: ±32Mbytes
  - Modo Thumb-2: ±16Mbytes
  - Modo Thumb: ±4Mbytes

Llamada

- Se usa la instrucción **BL** (**B**ranch with **L**ink).
- Transfiere a `LR[31:1]` los 31 MSB de la dirección de la instrucción sucesora secuencial (es decir, la que sigue a BL en la secuencia de instrucciones del programa).
- El LSB del LR toma un estado acorde al Modo del procesador: `LR[0]='1'` para Modo Thumb y LR[0]='0' si está en Modo ARM.
- Luego el Program Counter toma el valor de la dirección de salto contenida en la instrucción. Los rangos de salto dependen del modo de trabajo.
  - Modo ARM: ±32Mbytes
  - Modo Thumb-2: ±16Mbytes
  - Modo Thumb: ±4Mbytes

Llamada

- Se usa la instrucción **BL** (**B**ranch with **L**ink).
- Transfiere a `LR[31:1]` los 31 MSB de la dirección de la instrucción sucesora secuencial (es decir, la que sigue a BL en la secuencia de instrucciones del programa).
- El LSB del LR toma un estado acorde al Modo del procesador: `LR[0]='1'` para Modo Thumb y LR[0]='0' si está en Modo ARM.
- Luego el Program Counter toma el valor de la dirección de salto contenida en la instrucción. Los rangos de salto dependen del modo de trabajo.

Llamada

- Se usa la instrucción **BL** (**B**ranch with **L**ink).
- Transfiere a `LR[31:1]` los 31 MSB de la dirección de la instrucción sucesora secuencial (es decir, la que sigue a BL en la secuencia de instrucciones del programa).
- El LSB del LR toma un estado acorde al Modo del procesador: `LR[0]='1'` para Modo Thumb y LR[0]='0' si está en Modo ARM.
- Luego el Program Counter toma el valor de la dirección de salto contenida en la instrucción. Los rangos de salto dependen del modo de trabajo.

Modo ARM	±32Mbytes
Modo Thumb-2	±16Mbytes
Modo Thumb	±4Mbytes

Llamada

- Se usa la instrucción **BL** (**B**ranch with **L**ink).
- Transfiere a `LR[31:1]` los 31 MSB de la dirección de la instrucción sucesora secuencial (es decir, la que sigue a BL en la secuencia de instrucciones del programa).
- El LSB del LR toma un estado acorde al Modo del procesador: `LR[0]='1'` para Modo Thumb y LR[0]='0' si está en Modo ARM.
- Luego el Program Counter toma el valor de la dirección de salto contenida en la instrucción. Los rangos de salto dependen del modo de trabajo.

Modo ARM	±32Mbytes
Modo Thumb-2	±16Mbytes
Modo Thumb	±4Mbytes

Llamada

- Se usa la instrucción **BL** (**B**ranch with **L**ink).
- Transfiere a `LR[31:1]` los 31 MSB de la dirección de la instrucción sucesora secuencial (es decir, la que sigue a BL en la secuencia de instrucciones del programa).
- El LSB del LR toma un estado acorde al Modo del procesador: `LR[0]='1'` para Modo Thumb y LR[0]='0' si está en Modo ARM.
- Luego el Program Counter toma el valor de la dirección de salto contenida en la instrucción. Los rangos de salto dependen del modo de trabajo.

Modo ARM	±32Mbytes
Modo Thumb-2	±16Mbytes
Modo Thumb	±4Mbytes

Llamada

- Se usa la instrucción **BL** (**B**ranch with **L**ink).
- Transfiere a `LR[31:1]` los 31 MSB de la dirección de la instrucción sucesora secuencial (es decir, la que sigue a BL en la secuencia de instrucciones del programa).
- El LSB del LR toma un estado acorde al Modo del procesador: `LR[0]='1'` para Modo Thumb y LR[0]='0' si está en Modo ARM.
- Luego el Program Counter toma el valor de la dirección de salto contenida en la instrucción. Los rangos de salto dependen del modo de trabajo.

Modo ARM	±32Mbytes
Modo Thumb-2	±16Mbytes
Modo Thumb	±4Mbytes

Pasaje de argumentos

- El estándar base establece utilizar los registros core, `R0` a `R3` en primer instancia, y luego continuar en el stack.
- Esto permite para subrutinas que requieren un bajo o moderado número de argumentos, resolver el pasaje con el mínimo overhead.
- Si el procesador tiene disponible (y habilitado) uno o mas coprocesadores, entonces de acuerdo al tipo de dato fundamental que corresponda al parámetro puede utilizarse un registro de un coprocesador

Pasaje de argumentos

- El estándar base establece utilizar los registros core, `R0` a `R3` en primer instancia, y luego continuar en el stack.
- Esto permite para subrutinas que requieren un bajo o moderado número de argumentos, resolver el pasaje con el mínimo overhead.
- Si el procesador tiene disponible (y habilitado) uno o mas coprocesadores, entonces de acuerdo al tipo de dato fundamental que corresponda al parámetro puede utilizarse un registro de un coprocesador

Pasaje de argumentos

- El estándar base establece utilizar los registros core, `R0` a `R3` en primer instancia, y luego continuar en el stack.
- Esto permite para subrutinas que requieren un bajo o moderado número de argumentos, resolver el pasaje con el mínimo overhead.
- Si el procesador tiene disponible (y habilitado) uno o mas coprocesadores, entonces de acuerdo al tipo de dato fundamental que corresponda al parámetro puede utilizarse un registro de un coprocesador

Pasaje de argumentos

- El estándar base establece utilizar los registros core, `R0` a `R3` en primer instancia, y luego continuar en el stack.
- Esto permite para subrutinas que requieren un bajo o moderado número de argumentos, resolver el pasaje con el mínimo overhead.
- Si el procesador tiene disponible (y habilitado) uno o mas coprocesadores, entonces de acuerdo al tipo de dato fundamental que corresponda al parámetro puede utilizarse un registro de un coprocesador

Pasaje de argumentos

- Si el argumento es un tipo de dato compuesto cuyo tamaño no puede ser determinado, la rutina invocante debe tenerlo almacenado en memoria, alineado a 4 Bytes y pasar un puntero en el próximo registro core disponible (o la pila en su defecto).
- Si el argumento es un tipo de dato compuesto cuyo tamaño no es múltiplo de 4, su tamaño se lleva al múltiplo de 4 inmediato superior.
- Tipos fundamentales de datos con tamaño menor de 4 bytes, pasan en la parte menos significativa del registro correspondiente (o pila en su defecto), con la extensión de signo o cero que corresponda al tipo de dato.

Pasaje de argumentos

- Si el argumento es un tipo de dato compuesto cuyo tamaño no puede ser determinado, la rutina invocante debe tenerlo almacenado en memoria, alineado a 4 Bytes y pasar un puntero en el próximo registro core disponible (o la pila en su defecto).
- Si el argumento es un tipo de dato compuesto cuyo tamaño no es múltiplo de 4, su tamaño se lleva al múltiplo de 4 inmediato superior.
- Tipos fundamentales de datos con tamaño menor de 4 bytes, pasan en la parte menos significativa del registro correspondiente (o pila en su defecto), con la extensión de signo o cero que corresponda al tipo de dato.

Pasaje de argumentos

- Si el argumento es un tipo de dato compuesto cuyo tamaño no puede ser determinado, la rutina invocante debe tenerlo almacenado en memoria, alineado a 4 Bytes y pasar un puntero en el próximo registro core disponible (o la pila en su defecto).
- Si el argumento es un tipo de dato compuesto cuyo tamaño no es múltiplo de 4, su tamaño se lleva al múltiplo de 4 inmediato superior.
- Tipos fundamentales de datos con tamaño menor de 4 bytes, pasan en la parte menos significativa del registro correspondiente (o pila en su defecto), con la extensión de signo o cero que corresponda al tipo de dato.

Pasaje de argumentos

- Si el argumento es un tipo de dato compuesto cuyo tamaño no puede ser determinado, la rutina invocante debe tenerlo almacenado en memoria, alineado a 4 Bytes y pasar un puntero en el próximo registro core disponible (o la pila en su defecto).
- Si el argumento es un tipo de dato compuesto cuyo tamaño no es múltiplo de 4, su tamaño se lleva al múltiplo de 4 inmediato superior.
- Tipos fundamentales de datos con tamaño menor de 4 bytes, pasan en la parte menos significativa del registro correspondiente (o pila en su defecto), con la extensión de signo o cero que corresponda al tipo de dato.

Pasaje de argumentos

- Un Tipo de Dato Fundamental (**TDF**) que requiere alineación a 8 bytes se almacena a partir de un Registro par, o en caso de estar en la pila con ese alineamiento.
- Si el tamaño en words del **TDF** no cabe en la cantidad de Registros disponibles, el argumento se divide parte en el Registro Core **hasta completar r3**, y la otra parte pasa por la pila.

Pasaje de argumentos

- Un Tipo de Dato Fundamental (**TDF**) que requiere alineación a 8 bytes se almacena a partir de un Registro par, o en caso de estar en la pila con ese alineamiento.
- Si el tamaño en words del **TDF** no cabe en la cantidad de Registros disponibles, el argumento se divide parte en el Registro Core **hasta completar r3**, y la otra parte pasa por la pila.

Pasaje de argumentos

- Un Tipo de Dato Fundamental (**TDF**) que requiere alineación a 8 bytes se almacena a partir de un Registro par, o en caso de estar en la pila con ese alineamiento.
- Si el tamaño en words del **TDF** no cabe en la cantidad de Registros disponibles, el argumento se divide parte en el Registro Core **hasta completar r3**, y la otra parte pasa por la pila.

Pasaje de argumentos si están los coprocesadores

- Half Presicion ->Parte baja de un Registro Sn. ($0 \leqslant n \leqslant 15$)
- Single Presicion ->Registro Sn ($0 \leqslant n \leqslant 15$).
- Double Presicion ->Registro Di ($0 \leqslant i \leqslant 7$).
- Vector Empaquetado en 64 bits ->Registro Di ($0 \leqslant i \leqslant 7$).
- Vector Empaquetado en 128 bits ->Registro Qj ($0 \leqslant j \leqslant 3$).

Pasaje de argumentos si están los coprocesadores

- Half Presicion ->Parte baja de un Registro Sn. ($0 \leqslant n \leqslant 15$)
- Single Presicion ->Registro Sn ($0 \leqslant n \leqslant 15$).
- Double Presicion ->Registro Di ($0 \leqslant i \leqslant 7$).
- Vector Empaquetado en 64 bits ->Registro Di ($0 \leqslant i \leqslant 7$).
- Vector Empaquetado en 128 bits ->Registro Qj ($0 \leqslant j \leqslant 3$).

Pasaje de argumentos si están los coprocesadores

- Half Presicion ->Parte baja de un Registro Sn. ($0 \leqslant n \leqslant 15$)
- Single Presicion ->Registro Sn ($0 \leqslant n \leqslant 15$).
- Double Presicion ->Registro Di ($0 \leqslant i \leqslant 7$).
- Vector Empaquetado en 64 bits ->Registro Di ($0 \leqslant i \leqslant 7$).
- Vector Empaquetado en 128 bits ->Registro Qj ($0 \leqslant j \leqslant 3$).

Pasaje de argumentos si están los coprocesadores

- Half Presicion ->Parte baja de un Registro Sn. ($0 \leqslant n \leqslant 15$)
- Single Presicion ->Registro Sn ($0 \leqslant n \leqslant 15$).
- Double Presicion ->Registro Di ($0 \leqslant i \leqslant 7$).
- Vector Empaquetado en 64 bits ->Registro Di ($0 \leqslant i \leqslant 7$).
- Vector Empaquetado en 128 bits ->Registro Qj ($0 \leqslant j \leqslant 3$).

Pasaje de argumentos si están los coprocesadores

- Half Presicion ->Parte baja de un Registro Sn. ($0 \leqslant n \leqslant 15$)
- Single Presicion ->Registro Sn ($0 \leqslant n \leqslant 15$).
- Double Presicion ->Registro Di ($0 \leqslant i \leqslant 7$).
- Vector Empaquetado en 64 bits ->Registro Di ($0 \leqslant i \leqslant 7$).
- Vector Empaquetado en 128 bits ->Registro Qj ($0 \leqslant j \leqslant 3$).

Pasaje de argumentos si están los coprocesadores

- Half Presicion ->Parte baja de un Registro Sn. ($0 \leqslant n \leqslant 15$)
- Single Presicion ->Registro Sn ($0 \leqslant n \leqslant 15$).
- Double Presicion ->Registro Di ($0 \leqslant i \leqslant 7$).
- Vector Empaquetado en 64 bits ->Registro Di ($0 \leqslant i \leqslant 7$).
- Vector Empaquetado en 128 bits ->Registro Qj ($0 \leqslant j \leqslant 3$).

Retorno de resultados base

En general se utilizan los registros `[R0:R3]`, dependiendo del tamaño del resultado.

- Half Precision Floating Point: `R0[15:0]`.
- Tipo de dato Fundamental menor de 4 bytes: `R0` con extensión de signo.
- Tipo de dato Fundamental de 4 bytes (int o float por ejemplo): `R0`.
- Tipo de dato Fundamental de 8 bytes (long long, double, o vectores de 8 bytes): `[R0:R1]`.
- Vector de 16 bytes: `[R0:R3]`.
- Tipo de dato compuesto menor de 4 bytes: `R0`.
- Tipo de dato de tamaño indeterminado: Se almacena en memoria alineado a 4 bytes y se pasa su referencia por `R0`.

Retorno de resultados base

En general se utilizan los registros `[R0:R3]`, dependiendo del tamaño del resultado.

- Half Precision Floating Point: `R0[15:0]`.
- Tipo de dato Fundamental menor de 4 bytes: `R0` con extensión de signo.
- Tipo de dato Fundamental de 4 bytes (int o float por ejemplo): `R0`.
- Tipo de dato Fundamental de 8 bytes (long long, double, o vectores de 8 bytes): `[R0:R1]`.
- Vector de 16 bytes: `[R0:R3]`.
- Tipo de dato compuesto menor de 4 bytes: `R0`.
- Tipo de dato de tamaño indeterminado: Se almacena en memoria alineado a 4 bytes y se pasa su referencia por `R0`.

Retorno de resultados base

En general se utilizan los registros `[R0:R3]`, dependiendo del tamaño del resultado.

- Half Precision Floating Point: `R0[15:0]`.
- Tipo de dato Fundamental menor de 4 bytes: `R0` con extensión de signo.
- Tipo de dato Fundamental de 4 bytes (int o float por ejemplo): `R0`.
- Tipo de dato Fundamental de 8 bytes (long long, double, o vectores de 8 bytes): `[R0:R1]`.
- Vector de 16 bytes: `[R0:R3]`.
- Tipo de dato compuesto menor de 4 bytes: `R0`.
- Tipo de dato de tamaño indeterminado: Se almacena en memoria alineado a 4 bytes y se pasa su referencia por `R0`.

Retorno de resultados base

En general se utilizan los registros `[R0:R3]`, dependiendo del tamaño del resultado.

- Half Precision Floating Point: `R0[15:0]`.
- Tipo de dato Fundamental menor de 4 bytes: `R0` con extensión de signo.
- Tipo de dato Fundamental de 4 bytes (int o float por ejemplo): `R0`.
- Tipo de dato Fundamental de 8 bytes (long long, double, o vectores de 8 bytes): `[R0:R1]`.
- Vector de 16 bytes: `[R0:R3]`.
- Tipo de dato compuesto menor de 4 bytes: `R0`.
- Tipo de dato de tamaño indeterminado: Se almacena en memoria alineado a 4 bytes y se pasa su referencia por `R0`.

Retorno de resultados base

En general se utilizan los registros `[R0:R3]`, dependiendo del tamaño del resultado.

- Half Precision Floating Point: `R0[15:0]`.
- Tipo de dato Fundamental menor de 4 bytes: `R0` con extensión de signo.
- Tipo de dato Fundamental de 4 bytes (int o float por ejemplo): `R0`.
- Tipo de dato Fundamental de 8 bytes (long long, double, o vectores de 8 bytes): `[R0:R1]`.
- Vector de 16 bytes: `[R0:R3]`.
- Tipo de dato compuesto menor de 4 bytes: `R0`.
- Tipo de dato de tamaño indeterminado: Se almacena en memoria alineado a 4 bytes y se pasa su referencia por `R0`.

Retorno de resultados base

En general se utilizan los registros `[R0:R3]`, dependiendo del tamaño del resultado.

- Half Precision Floating Point: `R0[15:0]`.
- Tipo de dato Fundamental menor de 4 bytes: `R0` con extensión de signo.
- Tipo de dato Fundamental de 4 bytes (int o float por ejemplo): `R0`.
- Tipo de dato Fundamental de 8 bytes (long long, double, o vectores de 8 bytes): `[R0:R1]`.
- Vector de 16 bytes: `[R0:R3]`.
- Tipo de dato compuesto menor de 4 bytes: `R0`.
- Tipo de dato de tamaño indeterminado: Se almacena en memoria alineado a 4 bytes y se pasa su referencia por `R0`.

Retorno de resultados base

En general se utilizan los registros `[R0:R3]`, dependiendo del tamaño del resultado.

- Half Precision Floating Point: `R0[15:0]`.
- Tipo de dato Fundamental menor de 4 bytes: `R0` con extensión de signo.
- Tipo de dato Fundamental de 4 bytes (int o float por ejemplo): `R0`.
- Tipo de dato Fundamental de 8 bytes (long long, double, o vectores de 8 bytes): `[R0:R1]`.
- Vector de 16 bytes: `[R0:R3]`.
- Tipo de dato compuesto menor de 4 bytes: `R0`.
- Tipo de dato de tamaño indeterminado: Se almacena en memoria alineado a 4 bytes y se pasa su referencia por `R0`.

Retorno de resultados base

En general se utilizan los registros `[R0:R3]`, dependiendo del tamaño del resultado.

- Half Precision Floating Point: `R0[15:0]`.
- Tipo de dato Fundamental menor de 4 bytes: `R0` con extensión de signo.
- Tipo de dato Fundamental de 4 bytes (int o float por ejemplo): `R0`.
- Tipo de dato Fundamental de 8 bytes (long long, double, o vectores de 8 bytes): `[R0:R1]`.
- Vector de 16 bytes: `[R0:R3]`.
- Tipo de dato compuesto menor de 4 bytes: `R0`.
- Tipo de dato de tamaño indeterminado: Se almacena en memoria alineado a 4 bytes y se pasa su referencia por `R0`.

Retorno de resultados base

En general se utilizan los registros `[R0:R3]`, dependiendo del tamaño del resultado.

- Half Precision Floating Point: `R0[15:0]`.
- Tipo de dato Fundamental menor de 4 bytes: `R0` con extensión de signo.
- Tipo de dato Fundamental de 4 bytes (int o float por ejemplo): `R0`.
- Tipo de dato Fundamental de 8 bytes (long long, double, o vectores de 8 bytes): `[R0:R1]`.
- Vector de 16 bytes: `[R0:R3]`.
- Tipo de dato compuesto menor de 4 bytes: `R0`.
- Tipo de dato de tamaño indeterminado: Se almacena en memoria alineado a 4 bytes y se pasa su referencia por `R0`.

Retorno de resultados con coprocesadores

- En general si está habilitado el coprocesador los valores de punto flotante o datos SIMD empaquetados retornan por el registro `S0`, `D0`, ó `Q0`, según el Tipo de Dato Fundamental que se trate.

- El resto del los Tipos de Datos Fundamentales se retornan de la manera definida en el estándar base.

Retorno de resultados con coprocesadores

- En general si está habilitado el coprocesador los valores de punto flotante o datos SIMD empaquetados retornan por el registro `S0`, `D0`, ó `Q0`, según el Tipo de Dato Fundamental que se trate.
- El resto del los Tipos de Datos Fundamentales se retornan de la manera definida en el estándar base.

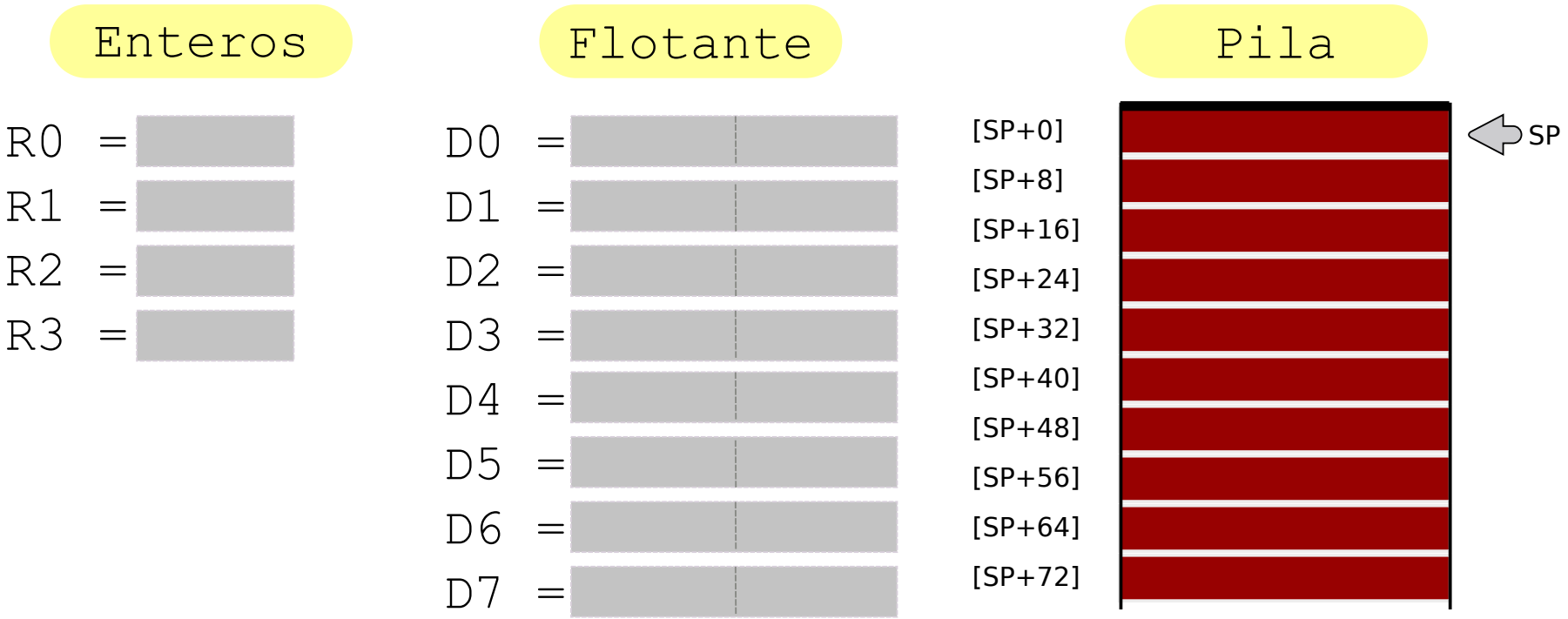
Retorno de resultados con coprocesadores

- En general si está habilitado el coprocesador los valores de punto flotante o datos SIMD empaquetados retornan por el registro `S0`, `D0`, ó `Q0`, según el Tipo de Dato Fundamental que se trate.

- El resto del los Tipos de Datos Fundamentales se retornan de la manera definida en el estándar base.

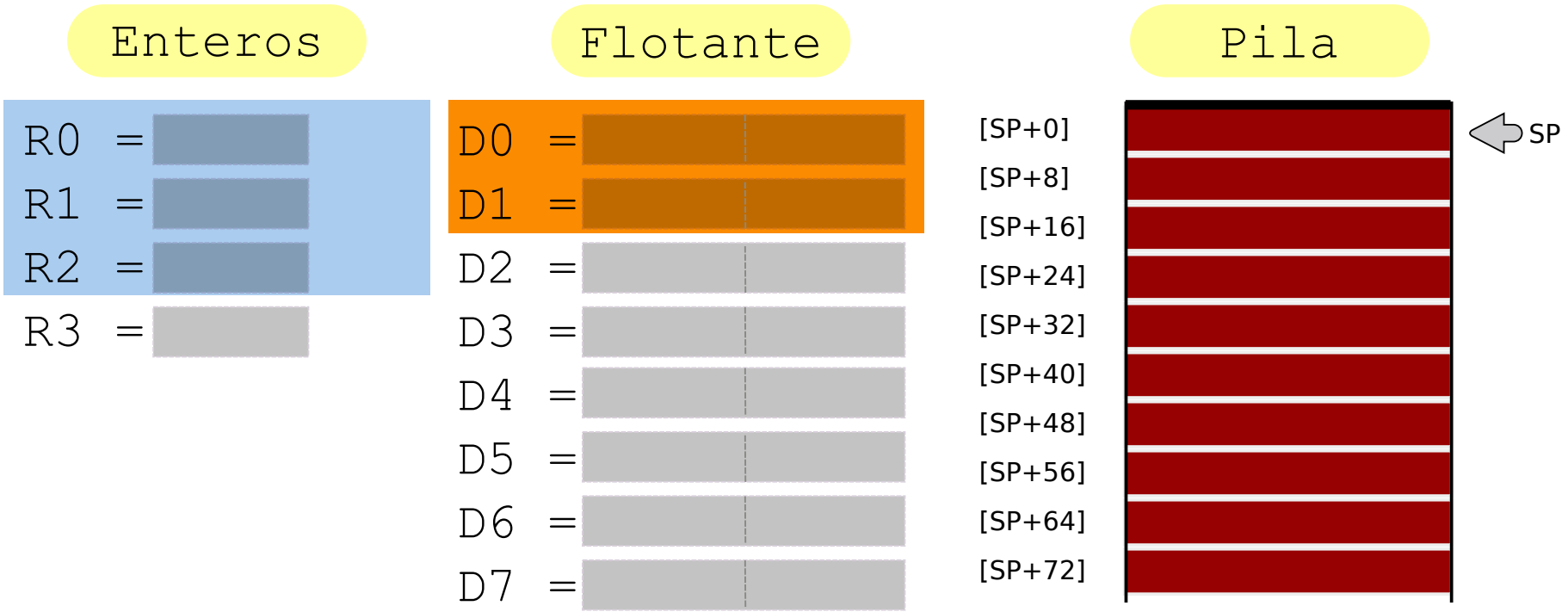
Ejemplo 1

```
int f1( int a, float b, double c, int* d, double* e)
```



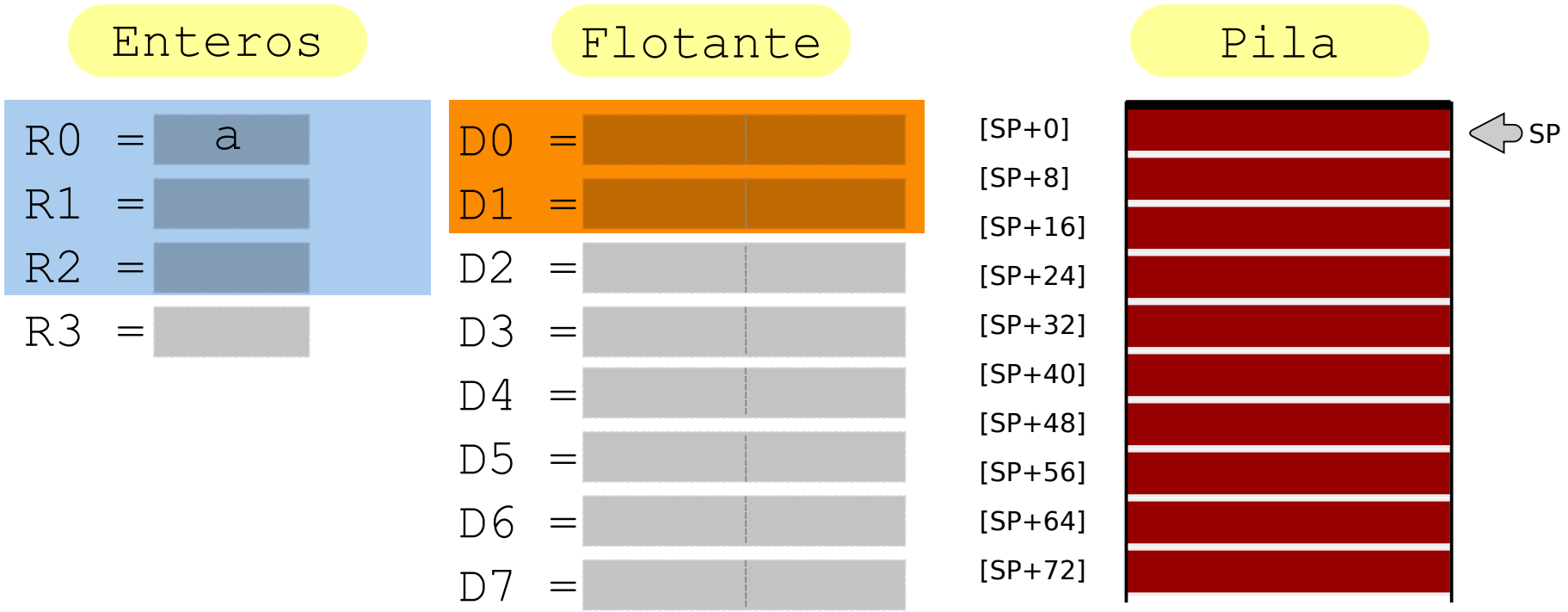
Ejemplo 1

```
int f1( int a, float b, double c, int* d, double* e)
```



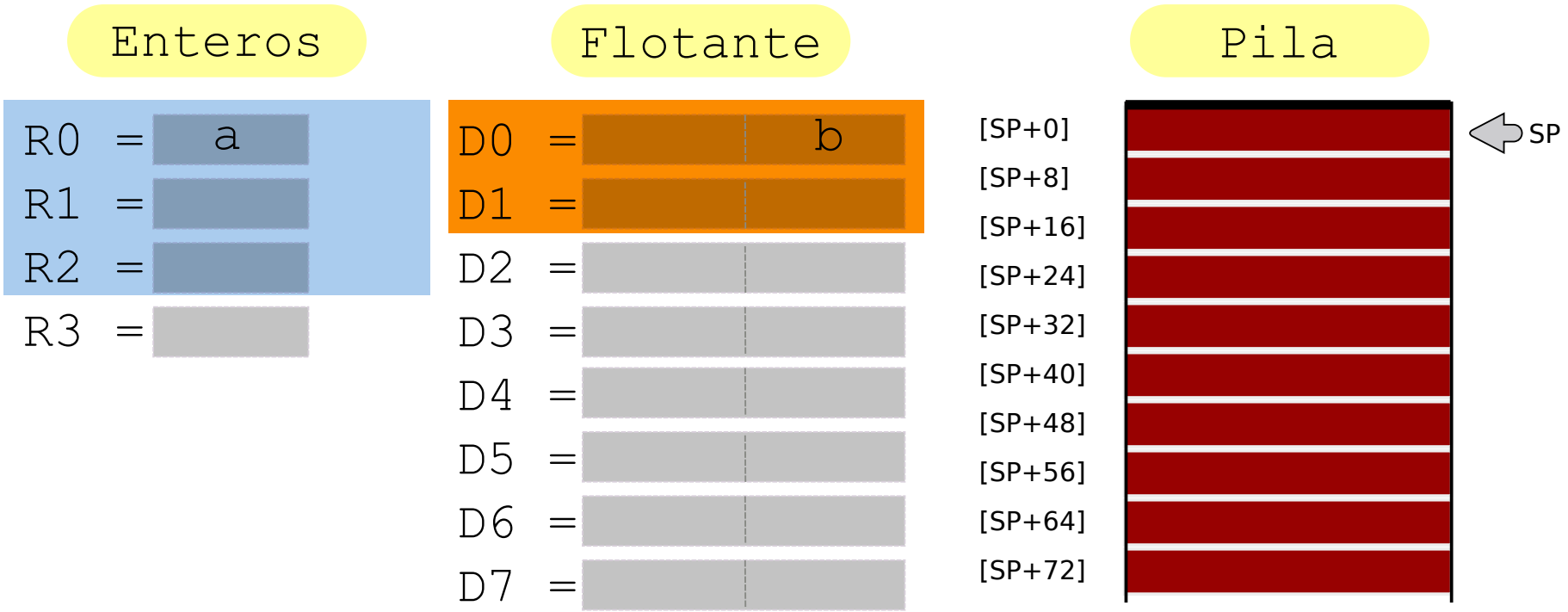
Ejemplo 1

```
int f1( int a, float b, double c, int* d, double* e)
```



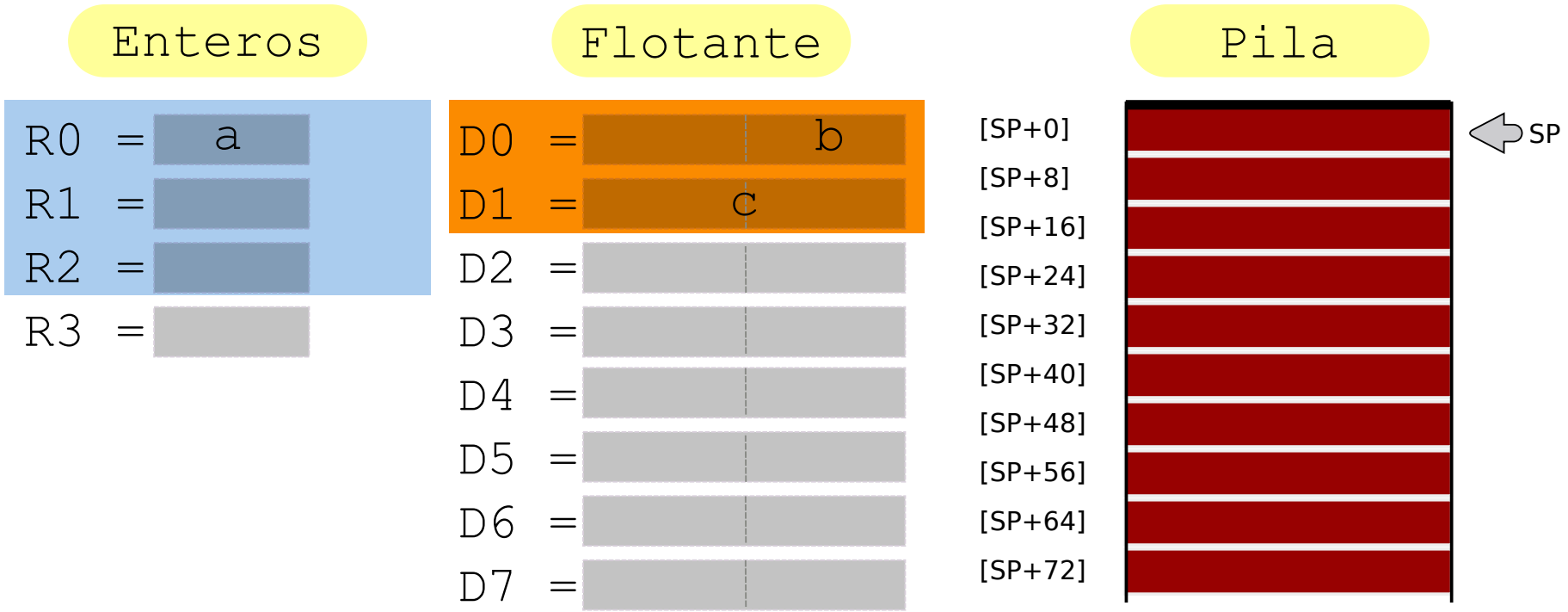
Ejemplo 1

```
int f1( int a, float b, double c, int* d, double* e)
```



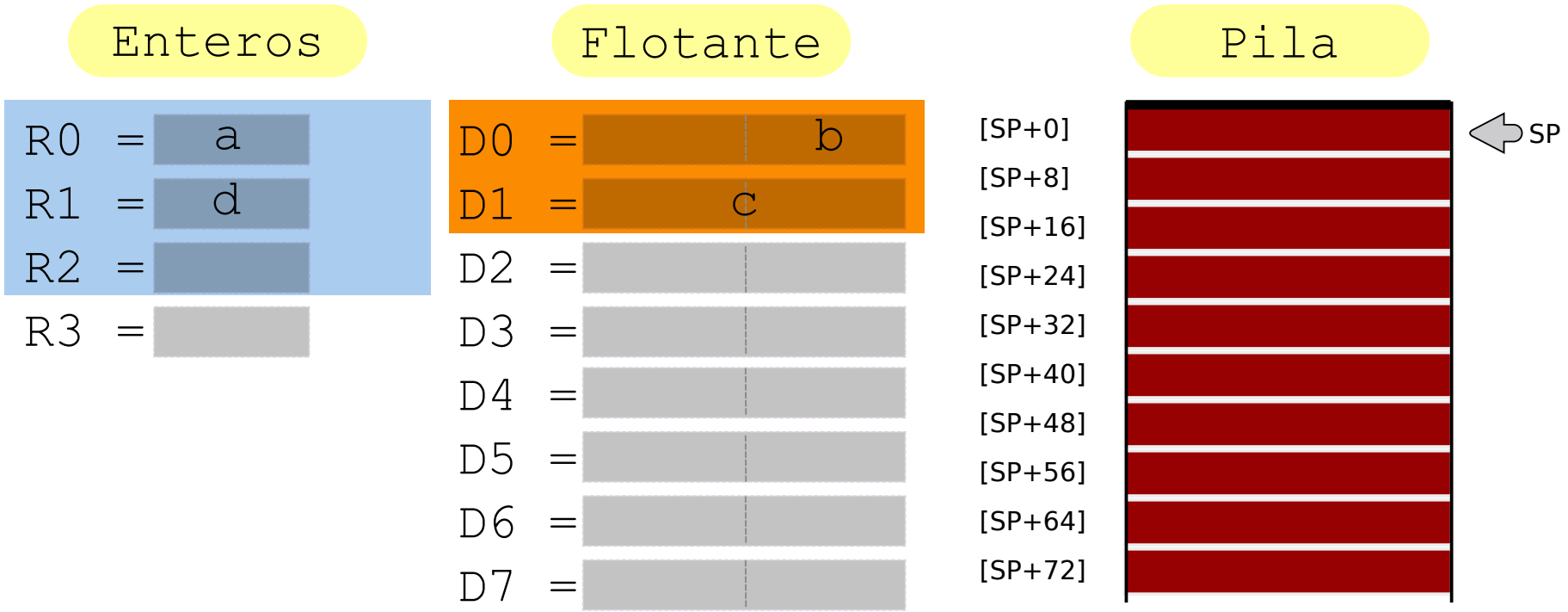
Ejemplo 1

```
int f1( int a, float b, double c, int* d, double* e)
```



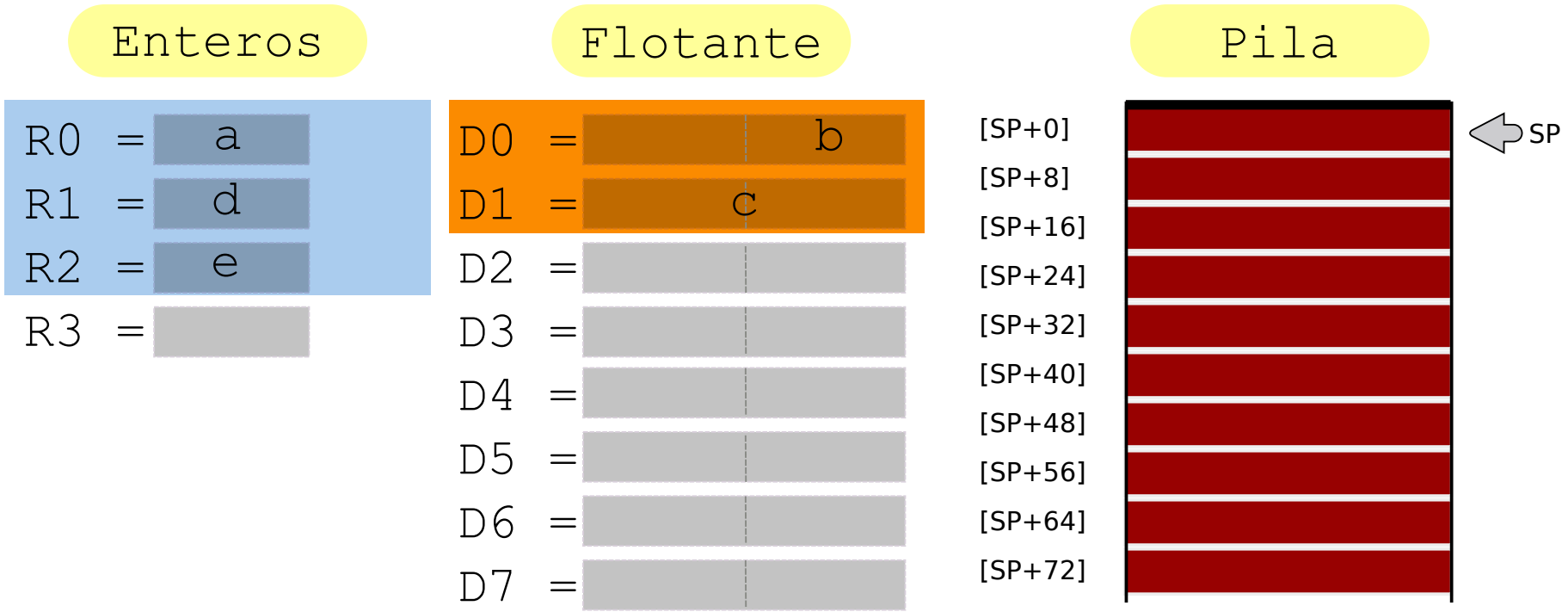
Ejemplo 1

```
int f1( int a, float b, double c, int* d, double* e)
```



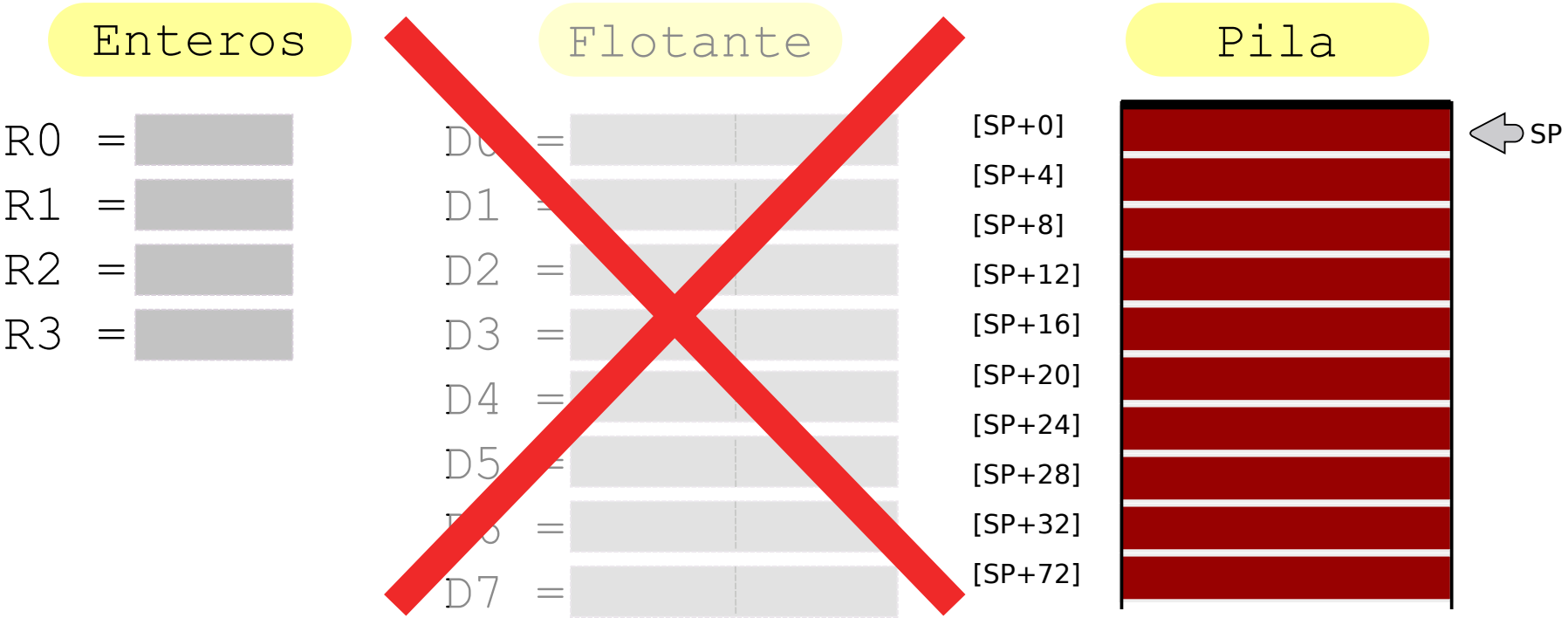
Ejemplo 1

```
int f1( int a, float b, double c, int* d, double* e)
```



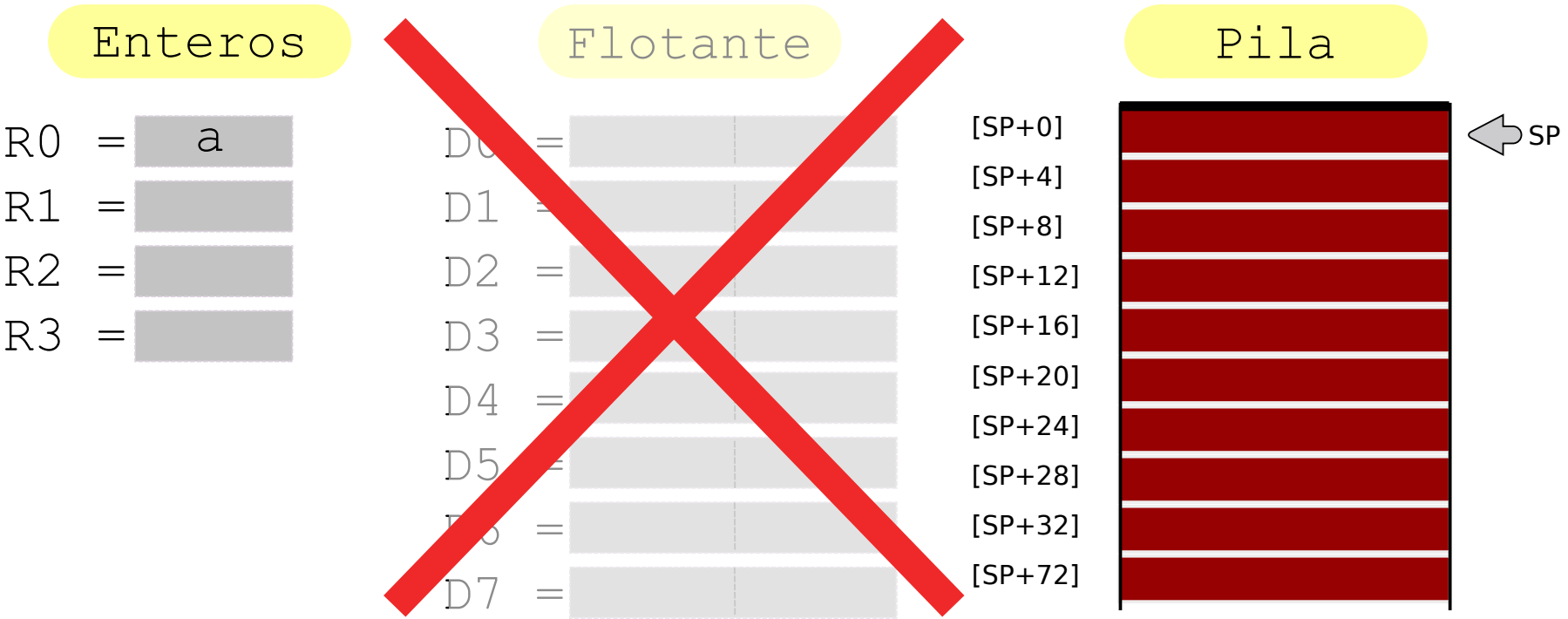
Ejemplo 1

```
int f1( int a, float b, double c, int* d, double* e)
```



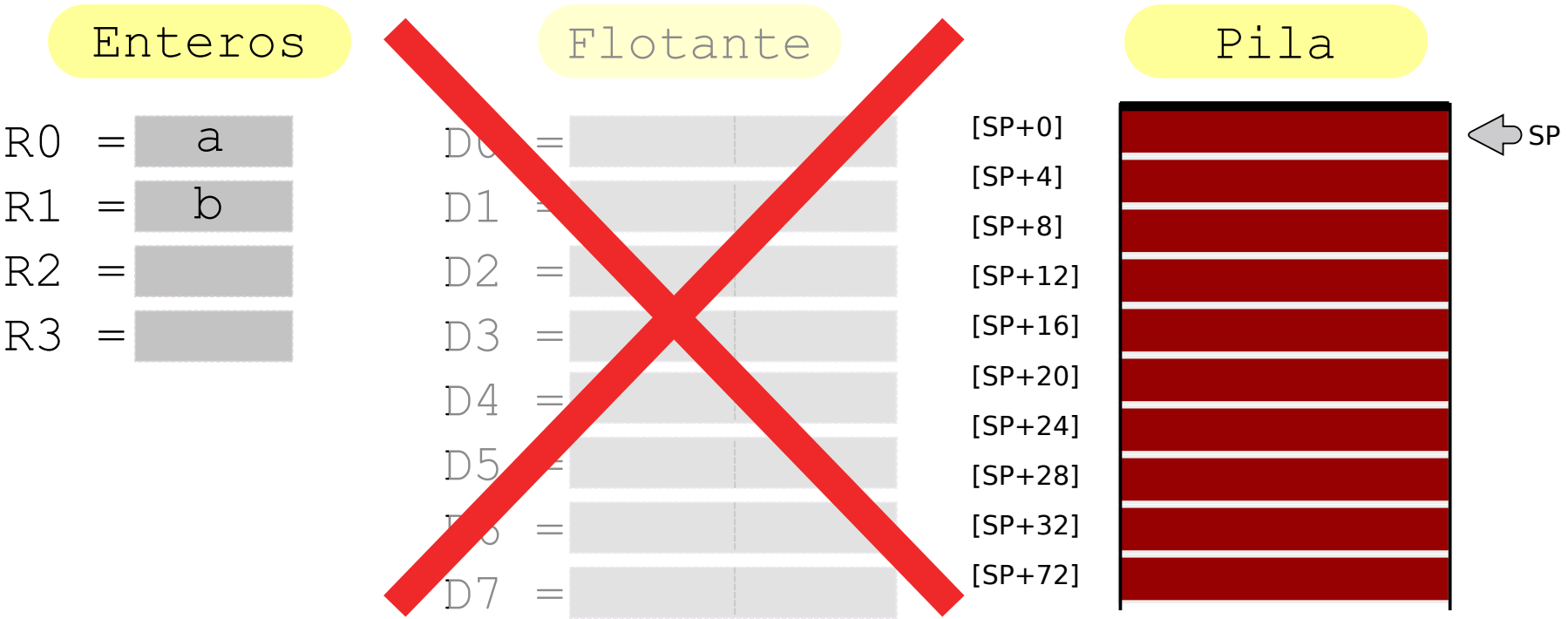
Ejemplo 1

```
int f1( int a, float b, double c, int* d, double* e)
```



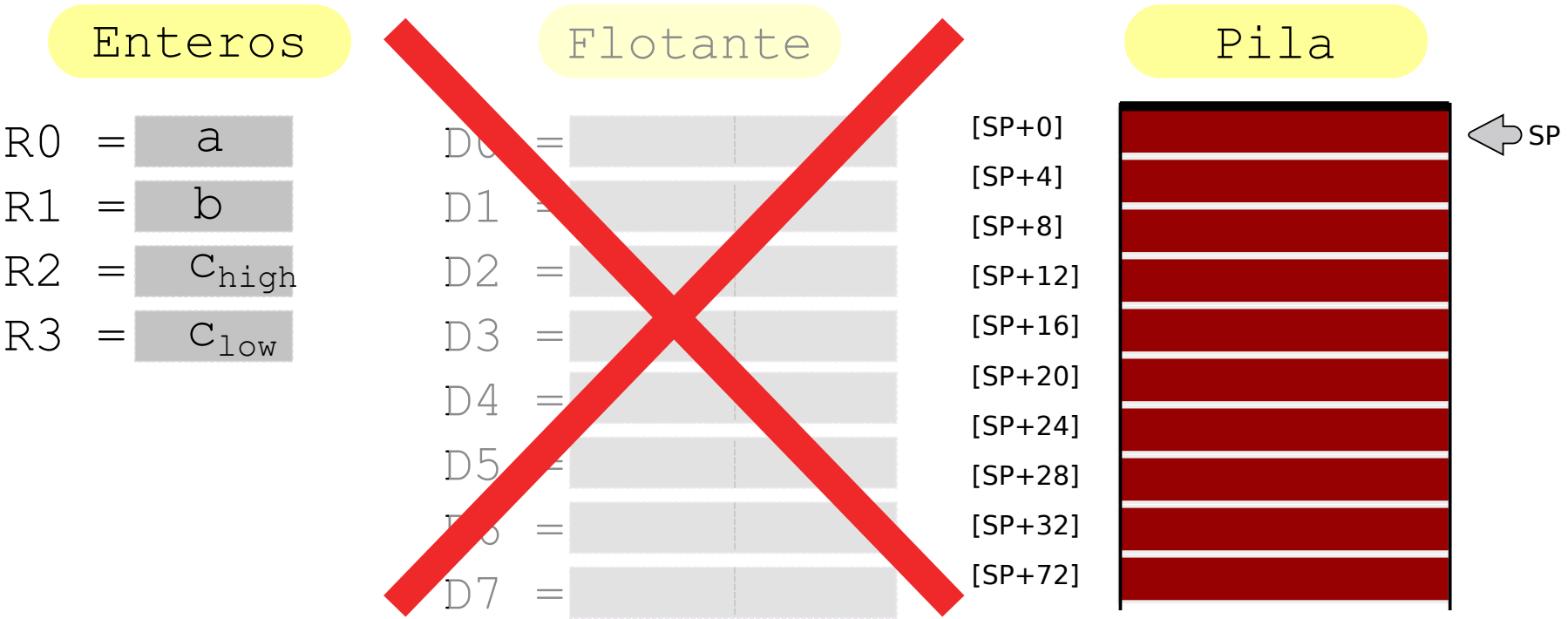
Ejemplo 1

```
int f1( int a, float b, double c, int* d, double* e)
```



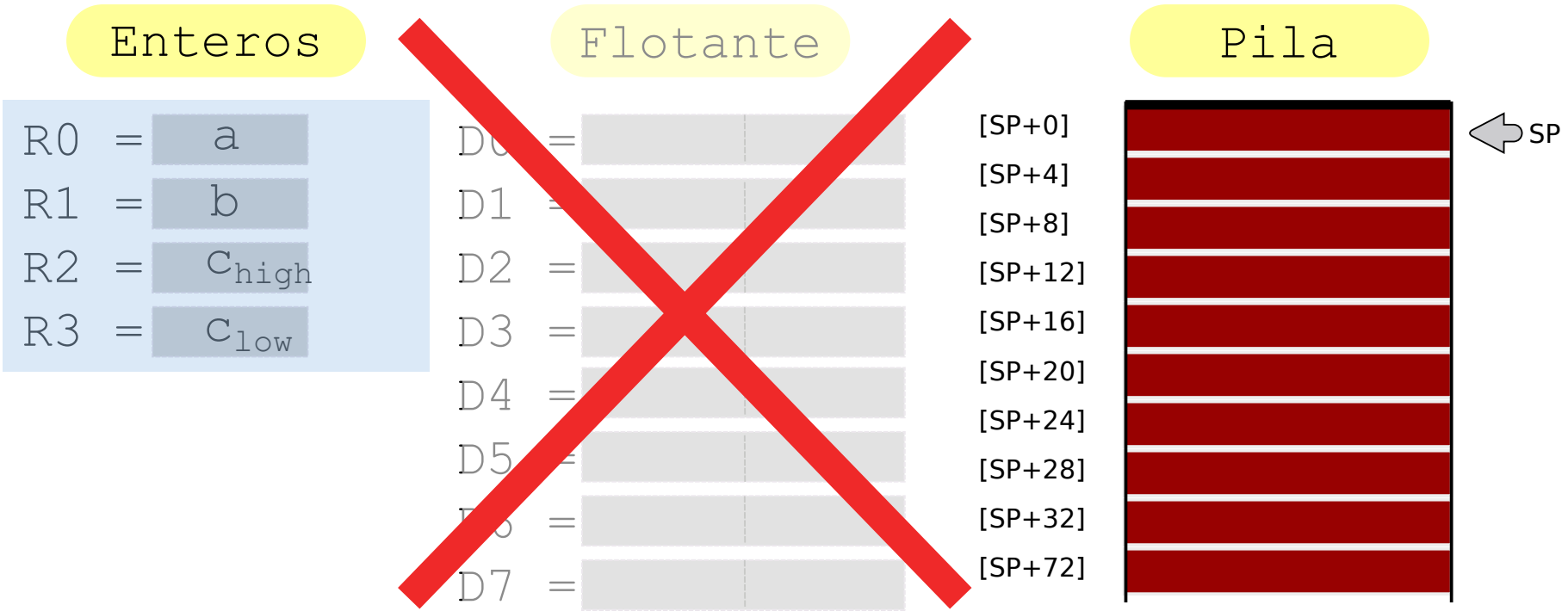
Ejemplo 1

```
int f1( int a, float b, double c, int* d, double* e)
```



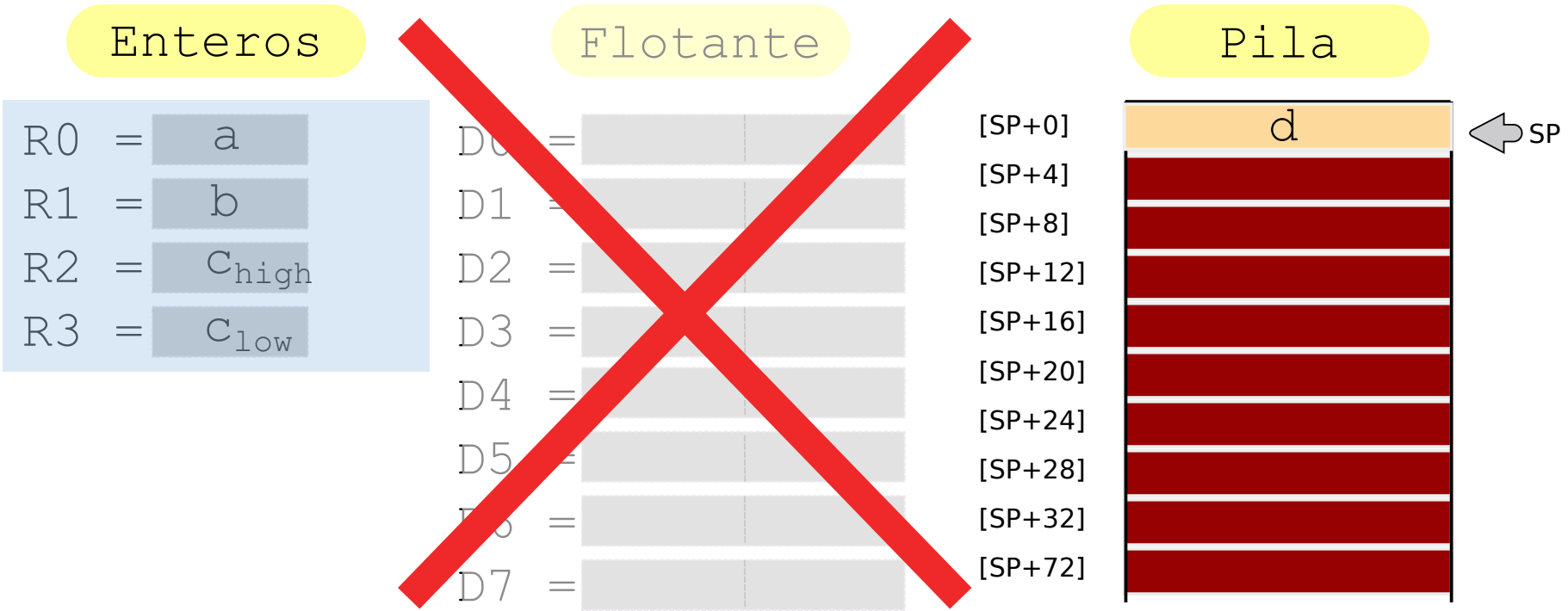
Ejemplo 1

```
int f1( int a, float b, double c, int* d, double* e)
```



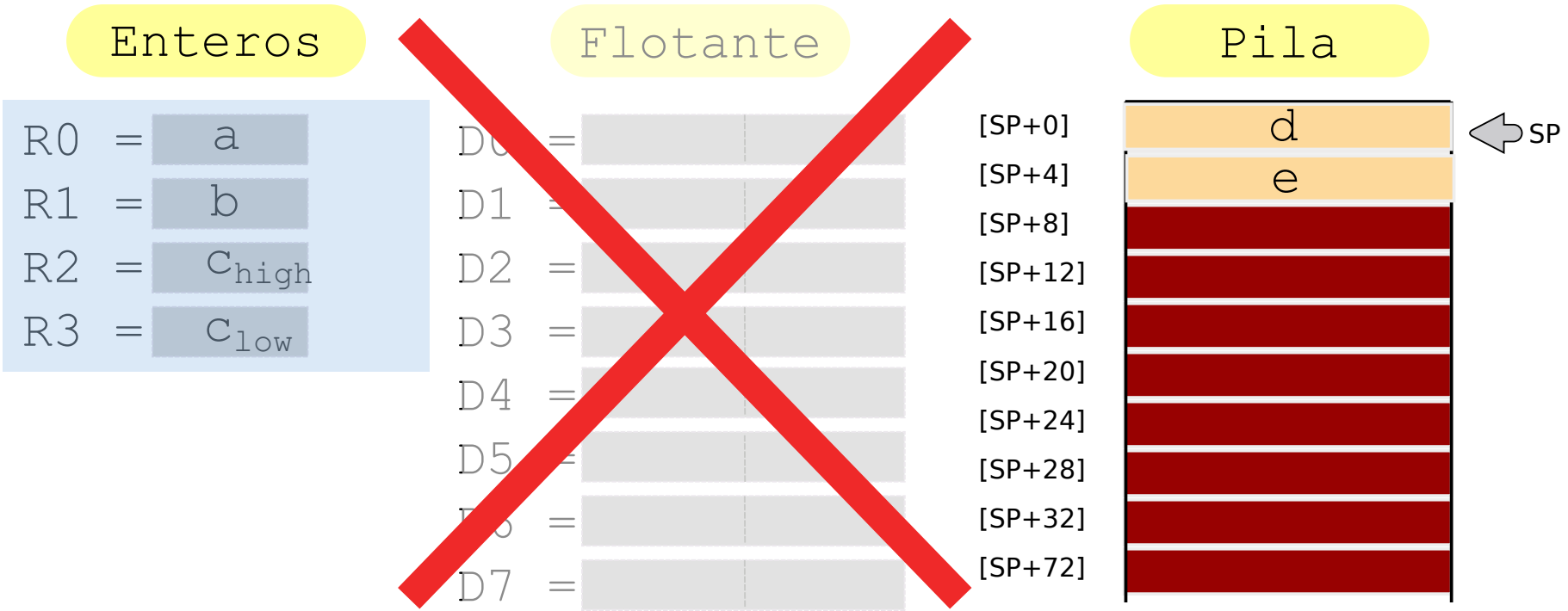
Ejemplo 1

```
int f1( int a, float b, double c, int* d, double* e)
```



Ejemplo 1

```
int f1( int a, float b, double c, int* d, double* e)
```



Ejemplo insano

```
int f( int a1, float a2, double a3, int a4, float a5,
double a6, int* a7, double* a8, int* a9, double a10,
int** a11, float* a12, double** a13, int* a14, float a15)
```

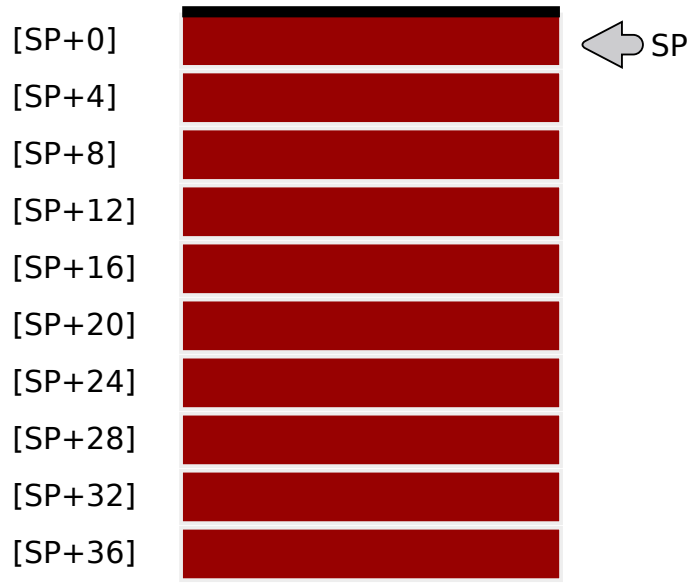
Enteros

R0 =
R1 =
R2 =
R3 =

Flotante

D0 =
D1 =
D2 =
D3 =
D4 =
D5 =
D6 =
D7 =

Pila



Ejemplo insano

```
int f( int a1, float a2, double a3, int a4, float a5,
double a6, int* a7, double* a8, int* a9, double a10,
int** a11, float* a12, double** a13, int* a14, float a15)
```

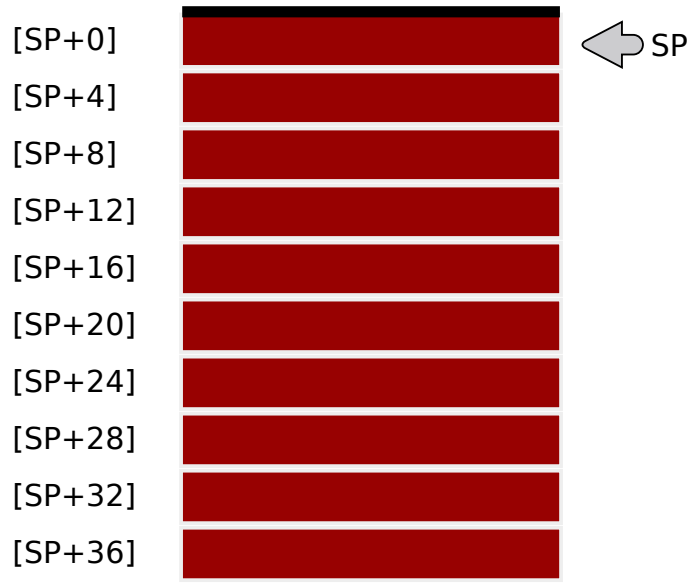
Enteros

R0 = a1
R1 =
R2 =
R3 =

Flotante

D0 =
D1 =
D2 =
D3 =
D4 =
D5 =
D6 =
D7 =

Pila



Ejemplo insano

```
int f( int a1, float a2, double a3, int a4, float a5,
double a6, int* a7, double* a8, int* a9, double a10,
int** a11, float* a12, double** a13, int* a14, float a15)
```

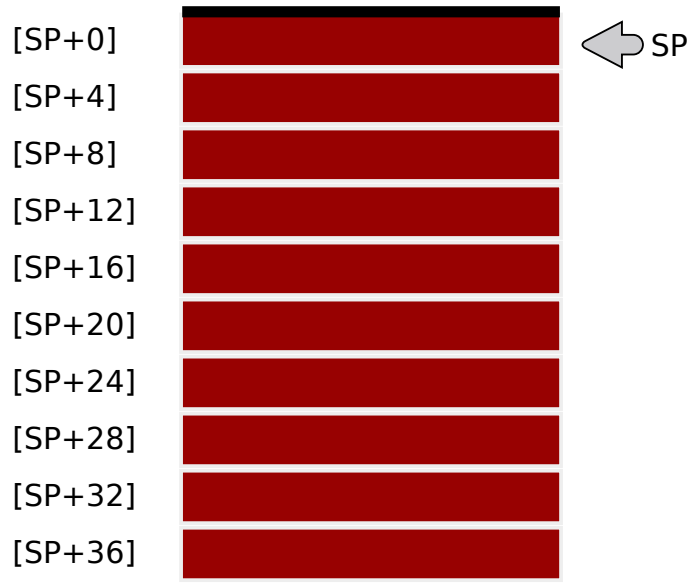
Enteros

R0 = a1
R1 =
R2 =
R3 =

Flotante

D0 = a2
D1 =
D2 =
D3 =
D4 =
D5 =
D6 =
D7 =

Pila



Ejemplo insano

```
int f( int a1, float a2, double a3, int a4, float a5,
double a6, int* a7, double* a8, int* a9, double a10,
int** a11, float* a12, double** a13, int* a14, float a15)
```

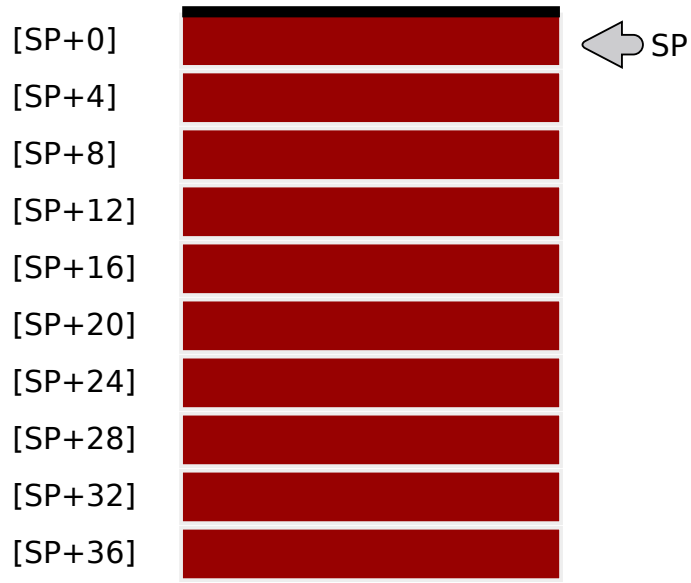
Enteros

R0 = a1
R1 =
R2 =
R3 =

Flotante

D0 = a2
D1 = a3
D2 =
D3 =
D4 =
D5 =
D6 =
D7 =

Pila



Ejemplo insano

```
int f( int a1, float a2, double a3, int a4, float a5,
double a6, int* a7, double* a8, int* a9, double a10,
int** a11, float* a12, double** a13, int* a14, float a15)
```

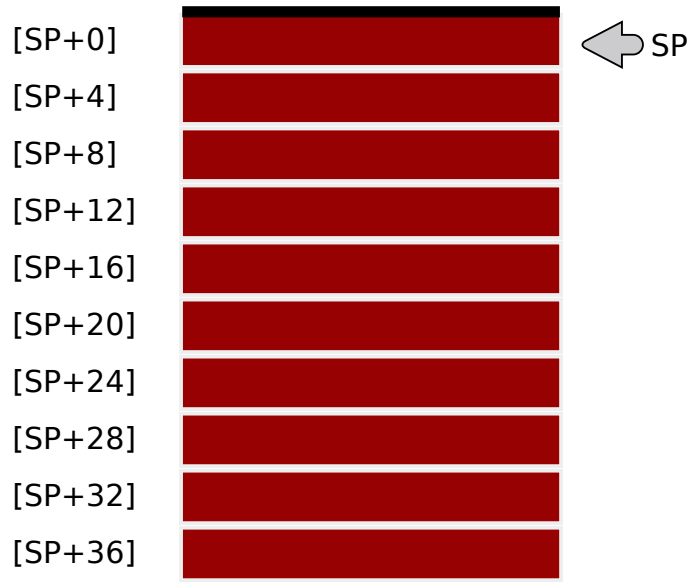
Enteros

R0 = a1
R1 = a4
R2 =
R3 =

Flotante

D0 = a2
D1 = a3
D2 =
D3 =
D4 =
D5 =
D6 =
D7 =

Pila



Ejemplo insano

```
int f( int a1, float a2, double a3, int a4, float a5,
double a6, int* a7, double* a8, int* a9, double a10,
int** a11, float* a12, double** a13, int* a14, float a15)
```

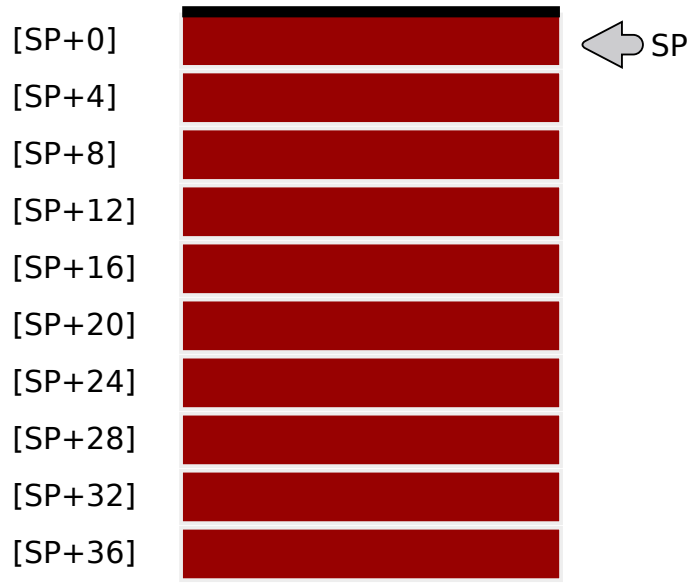
Enteros

R0 = a1
R1 = a4
R2 =
R3 =

Flotante

D0 = a2
D1 = a3
D2 = a5
D3 =
D4 =
D5 =
D6 =
D7 =

Pila



Ejemplo insano

```
int f( int a1, float a2, double a3, int a4, float a5,
double a6, int* a7, double* a8, int* a9, double a10,
int** a11, float* a12, double** a13, int* a14, float a15)
```

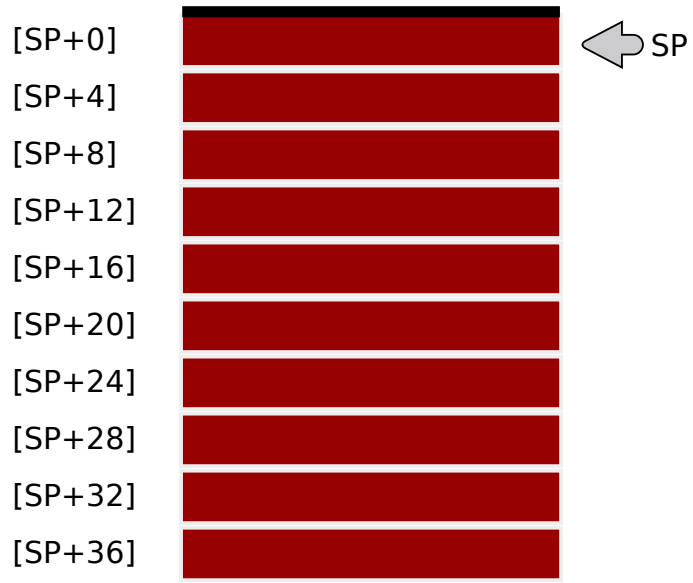
Enteros

R0 = a1
R1 = a4
R2 =
R3 =

Flotante

D0 = a2
D1 = a3
D2 = a5
D3 = a6
D4 =
D5 =
D6 =
D7 =

Pila



Ejemplo insano

```
int f( int a1, float a2, double a3, int a4, float a5,
double a6, int* a7, double* a8, int* a9, double a10,
int** a11, float* a12, double** a13, int* a14, float a15)
```

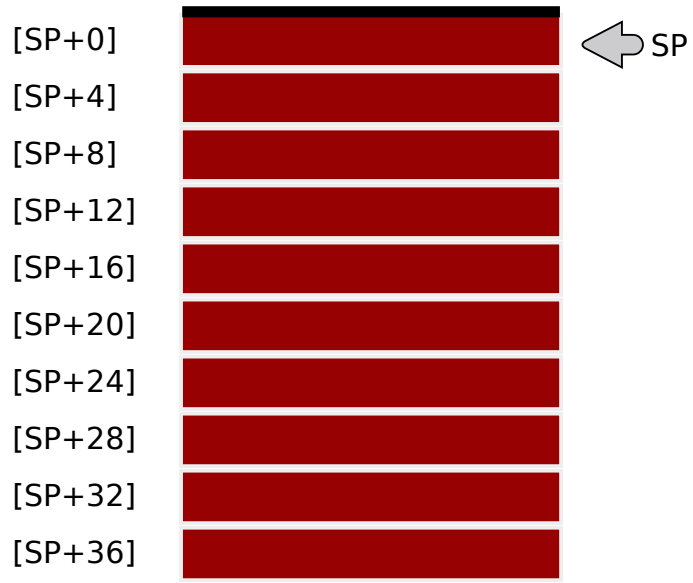
Enteros

R0 = a1
R1 = a4
R2 = a7
R3 =

Flotante

D0 = a2
D1 = a3
D2 = a5
D3 = a6
D4 =
D5 =
D6 =
D7 =

Pila



Ejemplo insano

```
int f( int a1, float a2, double a3, int a4, float a5,
double a6, int* a7, double* a8, int* a9, double a10,
int** a11, float* a12, double** a13, int* a14, float a15)
```

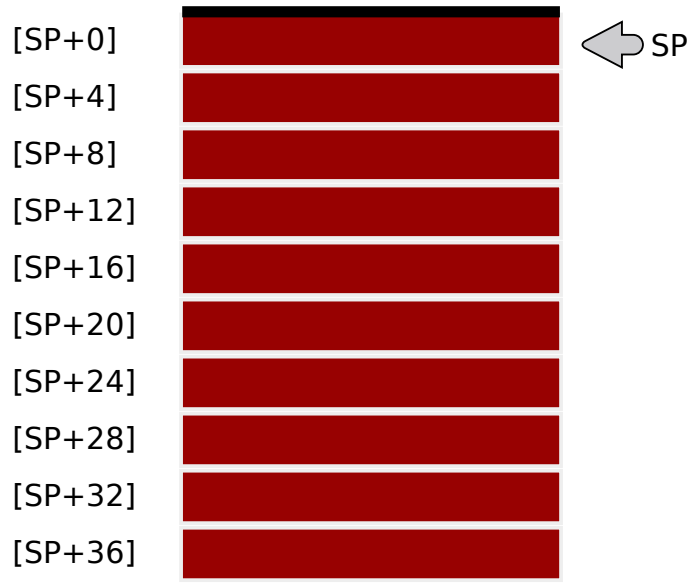
Enteros

R0 = a1
R1 = a4
R2 = a7
R3 = a8

Flotante

D0 = a2
D1 = a3
D2 = a5
D3 = a6
D4 =
D5 =
D6 =
D7 =

Pila



Ejemplo insano

```
int f( int a1, float a2, double a3, int a4, float a5,
double a6, int* a7, double* a8, int* a9, double a10,
int** a11, float* a12, double** a13, int* a14, float a15)
```

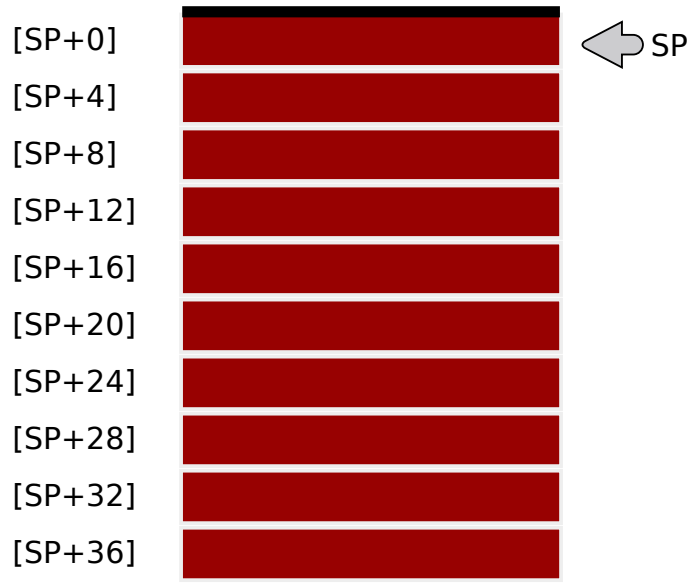
Enteros

R0	=	a1
R1	=	a4
R2	=	a7
R3	=	a8

Flotante

D0	=	a2
D1	=	a3
D2	=	a5
D3	=	a6
D4	=	
D5	=	
D6	=	
D7	=	

Pila



Ejemplo insano

```
int f( int a1, float a2, double a3, int a4, float a5,
double a6, int* a7, double* a8, int* a9, double a10,
int** a11, float* a12, double** a13, int* a14, float a15)
```

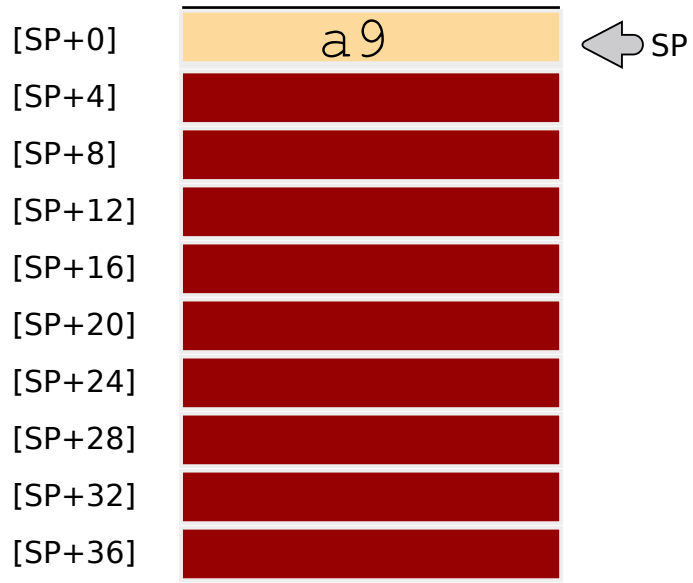
Enteros

R0	=	a1
R1	=	a4
R2	=	a7
R3	=	a8

Flotante

D0	=	a2
D1	=	a3
D2	=	a5
D3	=	a6
D4	=	
D5	=	
D6	=	
D7	=	

Pila



Ejemplo insano

```
int f( int a1, float a2, double a3, int a4, float a5,
double a6, int* a7, double* a8, int* a9, double a10,
int** a11, float* a12, double** a13, int* a14, float a15)
```

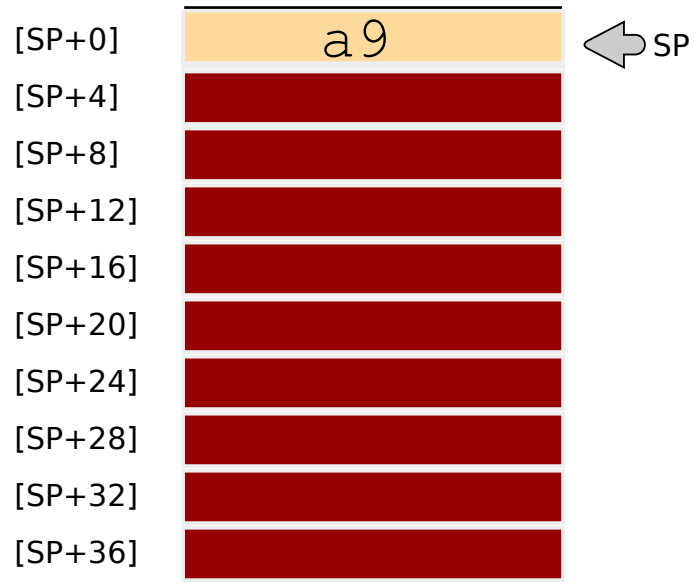
Enteros

R0	=	a1
R1	=	a4
R2	=	a7
R3	=	a8

Flotante

D0	=	a2
D1	=	a3
D2	=	a5
D3	=	a6
D4	=	a10
D5	=	
D6	=	
D7	=	

Pila



Ejemplo insano

```
int f( int a1, float a2, double a3, int a4, float a5,
double a6, int* a7, double* a8, int* a9, double a10,
int** a11, float* a12, double** a13, int* a14, float a15)
```

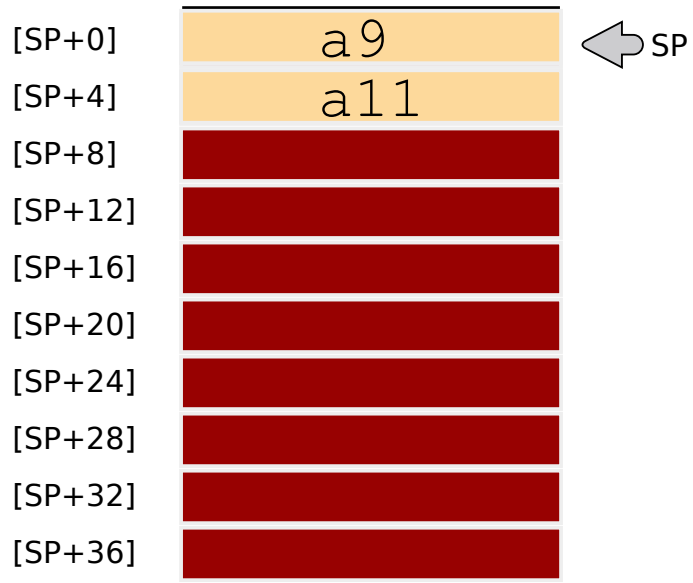
Enteros

```
R0 = a1
R1 = a4
R2 = a7
R3 = a8
```

Flotante

```
D0 = a2
D1 = a3
D2 = a5
D3 = a6
D4 = a10
D5 =
D6 =
D7 =
```

Pila



Ejemplo insano

```
int f( int a1, float a2, double a3, int a4, float a5,
double a6, int* a7, double* a8, int* a9, double a10,
int** a11, float* a12, double** a13, int* a14, float a15)
```

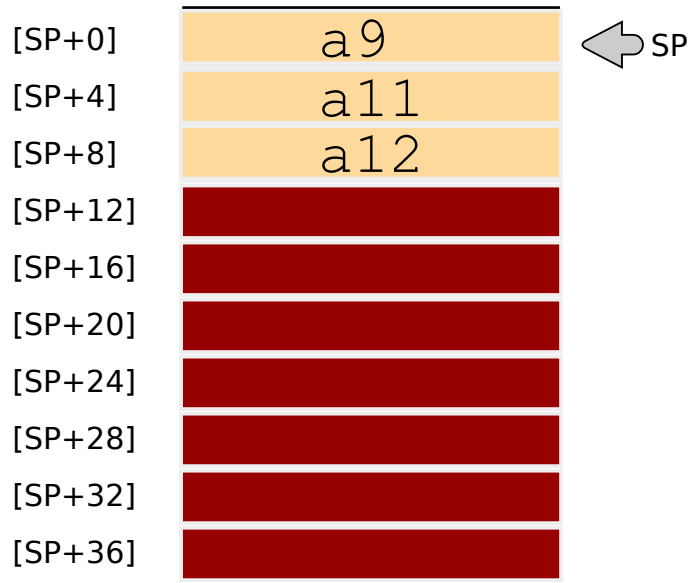
Enteros

R0	=	a1
R1	=	a4
R2	=	a7
R3	=	a8

Flotante

D0	=	a2
D1	=	a3
D2	=	a5
D3	=	a6
D4	=	a10
D5	=	
D6	=	
D7	=	

Pila



Ejemplo insano

```
int f( int a1, float a2, double a3, int a4, float a5,
double a6, int* a7, double* a8, int* a9, double a10,
int** a11, float* a12, double** a13, int* a14, float a15)
```

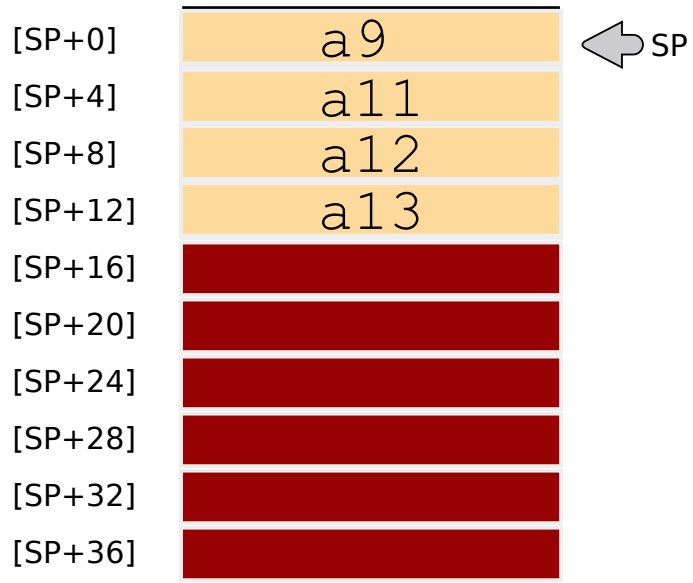
Enteros

R0	=	a1
R1	=	a4
R2	=	a7
R3	=	a8

Flotante

D0	=	a2
D1	=	a3
D2	=	a5
D3	=	a6
D4	=	a10
D5	=	
D6	=	
D7	=	

Pila



Ejemplo insano

```
int f( int a1, float a2, double a3, int a4, float a5,
double a6, int* a7, double* a8, int* a9, double a10,
int** a11, float* a12, double** a13, int* a14, float a15)
```

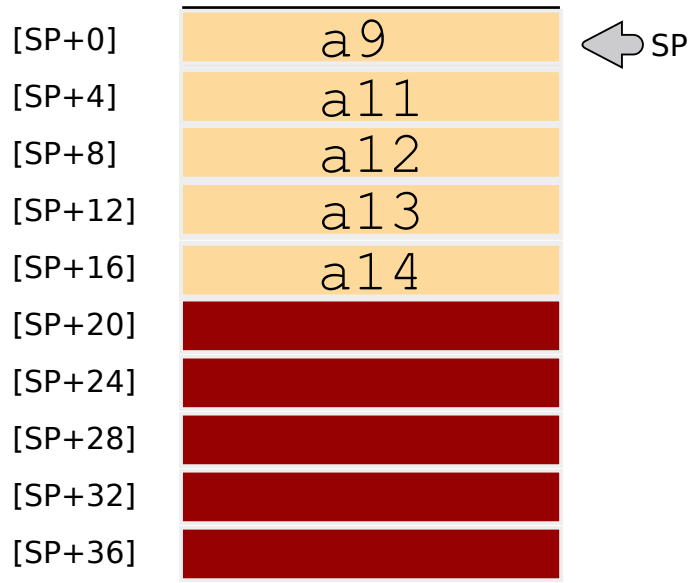
Enteros

R0	=	a1
R1	=	a4
R2	=	a7
R3	=	a8

Flotante

D0	=	a2
D1	=	a3
D2	=	a5
D3	=	a6
D4	=	a10
D5	=	
D6	=	
D7	=	

Pila



Ejemplo insano

```
int f( int a1, float a2, double a3, int a4, float a5,
double a6, int* a7, double* a8, int* a9, double a10,
int** a11, float* a12, double** a13, int* a14, float a15)
```

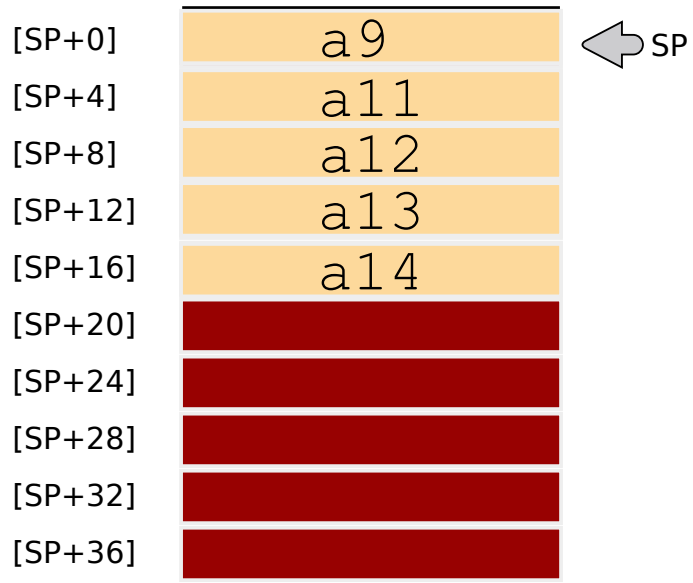
Enteros

R0	=	a1
R1	=	a4
R2	=	a7
R3	=	a8

Flotante

D0	=	a2
D1	=	a3
D2	=	a5
D3	=	a6
D4	=	a10
D5	=	a15
D6	=	
D7	=	

Pila



Ejemplo insano

```
int f( int a1, float a2, double a3, int a4, float a5,
double a6, int* a7, double* a8, int* a9, double a10,
int** a11, float* a12, double** a13, int* a14, float a15)
```

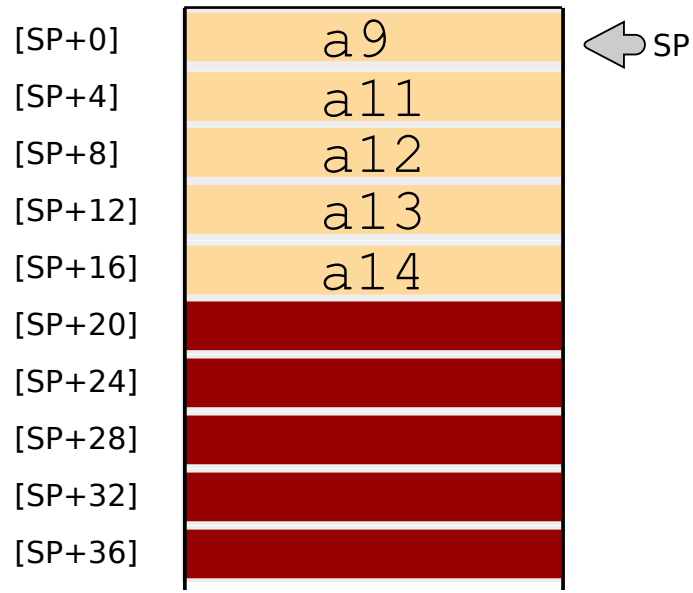
Enteros

```
R0 =  a1
R1 =  a4
R2 =  a7
R3 =  a8
```

Flotante

```
D0 =   a2
D1 =   a3
D2 =   a5
D3 =   a6
D4 =  a10
D5 =  a15
D6 =
D7 =
```

Pila



Un poco de práctica antes de seguir

Ejercicio Nro.1

Muestre cómo se almacenan en memoria los siguientes datos en procesadores Big-Endian y Little-Endian:

```
.byte 0x12                  .word 0x12345678
.byte 0x12, 0x34            .word 0x12345678, 0x9ABCDEF1
.hword 0x1234               .quad 0x123456789ABCDEF1
.hword 0x1234, 0x5678       .byte '1,'2,'3,'4
```

Un poco de práctica antes de seguir

Ejercicio Nro.1

Muestre cómo se almacenan en memoria los siguientes datos en procesadores Big-Endian y Little-Endian:

```
.byte 0x12                  .word 0x12345678
.byte 0x12, 0x34            .word 0x12345678, 0x9ABCDEF1
.hword 0x1234               .quad 0x123456789ABCDEF1
.hword 0x1234, 0x5678       .byte '1,'2,'3,'4
```

Big Endian: el byte más significativo en la posición de memoria menos significativa.
Little Endian: el byte más significativo en la posición de memoria más significativa.

Ejercicio Nro.1. Respuestas

`.byte 0x12`	– | 12 | –big endian
	– | 12 | –little endian
`.byte 12h, 0x34`	– | 12 | 34 | –big endian
	– | 12 | 34 | –little endian
`.hword 0x1234`	– | 12 | 34 | –big endian
	– | 34 | 12 | –little endian
`.hword 1234h, 0x5678`	– | 12 | 34 | 56 | 78 | –big endian
	– | 34 | 12 | 78 | 56 | –little endian
`.word 0x12345678`	– | 12 | 34 | 56 | 78 | –big endian
	– | 78 | 56 | 34 | 12 | –little endian
`.word 0x12345678,0x9ABCDEF1`	– | 12 | 34 | 56 | 78 | 9A | BC | DE | F1 | –big endian
	– | 78 | 56 | 34 | 12 | F1 | DE | BC | 9A | –little endian
`.quad 0x123456789ABCDEF1`	– | 12 | 34 | 56 | 78 | 9A | BC | DE | F1 | –big endian
	– | F1 | DE | BC | 9A | 78 | 56 | 34 | 12 | –little endian
`.byte '1,'2,'3,'4`	– | 31 | 32 | 33 | 34 | –big endian
	– | 31 | 32 | 33 | 34 | –little endian

Aritmética

Ejercicio Nro.2: Rangos de Representación

¿Cuál es el rango de representación de los números enteros sin signo de 8, 16 y 32 bits?
¿Cuál es el rango de representación de los números enteros con signo en complemento a dos de 8, 16 y 32 bits?

Sin signo	0 a $2^n - 1$
Con signo	-2^{n-1} a $2^{n-1} - 1$

	Sin signo	Con signo
8	0 a 255	−128 a 127
16	0 a 65535	−32768 a 32767
32	0 a 4294967295	−2147483648 a 2147483647

Aritmética

Ejercicio Nro.2: Rangos de Representación

¿Cuál es el rango de representación de los números enteros sin signo de 8, 16 y 32 bits?
¿Cuál es el rango de representación de los números enteros con signo en complemento a dos de 8, 16 y 32 bits?

Sin signo	0 a $2^n - 1$
Con signo	-2^{n-1} a $2^{n-1} - 1$

	Sin signo	Con signo
8	0 a 255	−128 a 127
16	0 a 65535	−32768 a 32767
32	0 a 4294967295	−2147483648 a 2147483647

Aritmética

Ejercicio Nro.3:

- Expresar los números 133 y 123 en notación binaria de 8 bits (notación sin signo)
- Sumar ambos números bit a bit
- Expresar los números -123 y 123 en notación complemento a dos de 8 bits y sumarlos bit a bit.
- ¿Qué conclusión se puede sacar observando el resultados de ambas operaciones?

Aritmética

Ejercicio Nro.3:

- Expresar los números 133 y 123 en notación binaria de 8 bits (notación sin signo)
- Sumar ambos números bit a bit
- Expresar los números -123 y 123 en notación complemento a dos de 8 bits y sumarlos bit a bit.
- ¿Qué conclusión se puede sacar observando el resultados de ambas operaciones?

Aritmética

Ejercicio Nro.3:

- Expresar los números 133 y 123 en notación binaria de 8 bits (notación sin signo)
- Sumar ambos números bit a bit
- Expresar los números -123 y 123 en notación complemento a dos de 8 bits y sumarlos bit a bit.
- ¿Qué conclusión se puede sacar observando el resultados de ambas operaciones?

Aritmética

Ejercicio Nro.3:

- Expresar los números 133 y 123 en notación binaria de 8 bits (notación sin signo)
- Sumar ambos números bit a bit
- Expresar los números -123 y 123 en notación complemento a dos de 8 bits y sumarlos bit a bit.
- ¿Qué conclusión se puede sacar observando el resultados de ambas operaciones?

Aritmética

Ejercicio Nro.3:

- Expresar los números 133 y 123 en notación binaria de 8 bits (notación sin signo)
- Sumar ambos números bit a bit
- Expresar los números -123 y 123 en notación complemento a dos de 8 bits y sumarlos bit a bit.
- ¿Qué conclusión se puede sacar observando el resultados de ambas operaciones?

Aritmética

Resultado Ejercicio Nro. 3:

$$\begin{array}{r} 123 = 01111011 \\ -123 = 10000101 \\ \hline 100000000 \end{array} \qquad \begin{array}{r} 123 = 01111011 \\ 133 = 10000101 \\ \hline 100000000 \end{array}$$

- Esa es la razón por la cual no hay una instrucción para sumar enteros signados y otra diferente para enteros no signados. Lo mismo para la resta. La representación en Ca2 de los números negativos asegura esta propiedad

- **Es responsabilidad del programador saber con qué tipo de números se está operando, y prestar atención a los flags correctos.**

Aritmética

Resultado Ejercicio Nro. 3:

```
  123 = 01111011
 -123 = 10000101
-------------------
      100000000
```

```
  123 = 01111011
  133 = 10000101
-------------------
      100000000
```

- Esa es la razón por la cual no hay una instrucción para sumar enteros signados y otra diferente para enteros no signados. Lo mismo para la resta. La representación en Ca2 de los números negativos asegura esta propiedad

- Es responsabilidad del programador saber con qué tipo de números se está operando, y prestar atención a los flags correctos.

Aritmética

Resultado Ejercicio Nro. 3:

```
  123 = 01111011          123 = 01111011
 -123 = 10000101          133 = 10000101
-----------------       -----------------
      100000000               100000000
```

- Esa es la razón por la cual no hay una instrucción para sumar enteros signados y otra diferente para enteros no signados. Lo mismo para la resta. La representación en Ca2 de los números negativos asegura esta propiedad

- **Es responsabilidad del programador saber con qué tipo de números se está operando, y prestar atención a los flags correctos.**

Modos e instrucciones

- Originalmente las instrucciones ARM son de 32 bits. Por ejemplo en ARM7TDMI.
- Instrucciones Thumb = Subset de las instrucciones ARM.
  - Trabajan con datos de 32 bits como las ARM
  - Su tamaño es de 16 bits (la mitad)
  - Su funcionalidad es acotada respecto de las ARM.
- Por lo tanto, hay cambios en la sintaxis:

```
ARM: ADD R0, R1, R6
Thumb: ADD R0, R6
```

- En modo Thumb el primer operando es al mismo tiempo el operando destino. Siempre se destruye el primer operando
- En modo ARM se conserva el primer operando reemplazando el registro resultado por otro diferente de los dos registros operando. Aquí tenemos rápidamente un primer ejemplo de perdida de funcionalidad en el Modo Thumb.

Modos e instrucciones

- Originalmente las instrucciones ARM son de 32 bits. Por ejemplo en ARM7TDMI.
- Instrucciones Thumb = Subset de las instrucciones ARM.
  - Trabajan con datos de 32 bits como las ARM.
  - Su tamaño es de 16 bits (la mitad)
  - Su funcionalidad es acotada respecto de las ARM.
- Por lo tanto, hay cambios en la sintaxis:

```
ARM: ADD R0, R1, R6
Thumb: ADD R0, R6
```

- En modo Thumb el primer operando es al mismo tiempo el operando destino. Siempre se destruye el primer operando
- En modo ARM se conserva el primer operando reemplazando el registro resultado por otro diferente de los dos registros operando. Aquí tenemos rápidamente un primer ejemplo de perdida de funcionalidad en el Modo Thumb.

Modos e instrucciones

- Originalmente las instrucciones ARM son de 32 bits. Por ejemplo en ARM7TDMI.
- Instrucciones Thumb = Subset de las instrucciones ARM.
  - **Trabajan con datos de 32 bits como las ARM**.
  - Su tamaño es de 16 bits (la mitad)
  - Su funcionalidad es acotada respecto de las ARM.
- Por lo tanto, hay cambios en la sintaxis:

  ARM: ADD R0, R1, R2

  Thumb: ADD R0, R1
- En modo Thumb el primer operando es al mismo tiempo el operando destino. Siempre se destruye el primer operando
- En modo ARM se conserva el primer operando reemplazando el registro resultado por otro diferente de los dos registros operando. Aquí tenemos rápidamente un primer ejemplo de perdida de funcionalidad en el Modo Thumb.

Modos e instrucciones

- Originalmente las instrucciones ARM son de 32 bits. Por ejemplo en ARM7TDMI.
- Instrucciones Thumb = Subset de las instrucciones ARM.
  - **Trabajan con datos de 32 bits como las ARM**.
  - Su tamaño es de 16 bits (la mitad)
  - Su funcionalidad es acotada respecto de las ARM.
- Por lo tanto, hay cambios en la sintaxis:

  ARM: ADD R0, R1, R2

  Thumb: ADD R0, R1
- En modo Thumb el primer operando es al mismo tiempo el operando destino. Siempre se destruye el primer operando
- En modo ARM se conserva el primer operando reemplazando el registro resultado por otro diferente de los dos registros operando. Aquí tenemos rápidamente un primer ejemplo de perdida de funcionalidad en el Modo Thumb.

Modos e instrucciones

- Originalmente las instrucciones ARM son de 32 bits. Por ejemplo en ARM7TDMI.
- Instrucciones Thumb = Subset de las instrucciones ARM.
  - **Trabajan con datos de 32 bits como las ARM**.
  - Su tamaño es de 16 bits (la mitad)
  - Su funcionalidad es acotada respecto de las ARM.
- Por lo tanto, hay cambios en la sintaxis:

  ARM: ADD R0, R1, R2

  Thumb: ADD R0, R1
- En modo Thumb el primer operando es al mismo tiempo el operando destino. Siempre se destruye el primer operando
- En modo ARM se conserva el primer operando reemplazando el registro resultado por otro diferente de los dos registros operando. Aquí tenemos rápidamente un primer ejemplo de perdida de funcionalidad en el Modo Thumb.

Modos e instrucciones

- Originalmente las instrucciones ARM son de 32 bits. Por ejemplo en ARM7TDMI.
- Instrucciones Thumb = Subset de las instrucciones ARM.
  - **Trabajan con datos de 32 bits como las ARM**.
  - Su tamaño es de 16 bits (la mitad)
  - Su funcionalidad es acotada respecto de las ARM.
- Por lo tanto, hay cambios en la sintaxis:

```
ARM   ADD R1,R1,R6
Thumb  ADD R1,R6
```

- En modo Thumb el primer operando es al mismo tiempo el operando destino. Siempre se destruye el primer operando
- En modo ARM se conserva el primer operando reemplazando el registro resultado por otro diferente de los dos registros operando. Aquí tenemos rápidamente un primer ejemplo de perdida de funcionalidad en el Modo Thumb.

Modos e instrucciones

- Originalmente las instrucciones ARM son de 32 bits. Por ejemplo en ARM7TDMI.
- Instrucciones Thumb = Subset de las instrucciones ARM.
  - **Trabajan con datos de 32 bits como las ARM**.
  - Su tamaño es de 16 bits (la mitad)
  - Su funcionalidad es acotada respecto de las ARM.
- Por lo tanto, hay cambios en la sintaxis:

```
ARM   ADD R1,R1,R6
Thumb  ADD R1,R6
```

- En modo Thumb el primer operando es al mismo tiempo el operando destino. Siempre se destruye el primer operando
- En modo ARM se conserva el primer operando reemplazando el registro resultado por otro diferente de los dos registros operando. Aquí tenemos rápidamente un primer ejemplo de perdida de funcionalidad en el Modo Thumb.

Modos e instrucciones

- Originalmente las instrucciones ARM son de 32 bits. Por ejemplo en ARM7TDMI.
- Instrucciones Thumb = Subset de las instrucciones ARM.
  - **Trabajan con datos de 32 bits como las ARM**.
  - Su tamaño es de 16 bits (la mitad)
  - Su funcionalidad es acotada respecto de las ARM.
- Por lo tanto, hay cambios en la sintaxis:

  ARM `ADD R1,R1,R6`

  Thumb `ADD R1,R6`

- En modo Thumb el primer operando es al mismo tiempo el operando destino. Siempre se destruye el primer operando
- En modo ARM se conserva el primer operando reemplazando el registro resultado por otro diferente de los dos registros operando. Aquí tenemos rápidamente un primer ejemplo de perdida de funcionalidad en el Modo Thumb.

Modos e instrucciones

- Originalmente las instrucciones ARM son de 32 bits. Por ejemplo en ARM7TDMI.
- Instrucciones Thumb = Subset de las instrucciones ARM.
  - **Trabajan con datos de 32 bits como las ARM**.
  - Su tamaño es de 16 bits (la mitad)
  - Su funcionalidad es acotada respecto de las ARM.
- Por lo tanto, hay cambios en la sintaxis:

  ARM `ADD R1,R1,R6`

  Thumb `ADD R1,R6`

- En modo Thumb el primer operando es al mismo tiempo el operando destino. Siempre se destruye el primer operando
- En modo ARM se conserva el primer operando reemplazando el registro resultado por otro diferente de los dos registros operando. Aquí tenemos rápidamente un primer ejemplo de perdida de funcionalidad en el Modo Thumb.

Modos e instrucciones

- Originalmente las instrucciones ARM son de 32 bits. Por ejemplo en ARM7TDMI.
- Instrucciones Thumb = Subset de las instrucciones ARM.
  - **Trabajan con datos de 32 bits como las ARM**.
  - Su tamaño es de 16 bits (la mitad)
  - Su funcionalidad es acotada respecto de las ARM.
- Por lo tanto, hay cambios en la sintaxis:

  ARM `ADD R1,R1,R6`

  Thumb `ADD R1,R6`

- En modo Thumb el primer operando es al mismo tiempo el operando destino. Siempre se destruye el primer operando
- En modo ARM se conserva el primer operando reemplazando el registro resultado por otro diferente de los dos registros operando. Aquí tenemos rápidamente un primer ejemplo de perdida de funcionalidad en el Modo Thumb.

Modos e instrucciones

- Originalmente las instrucciones ARM son de 32 bits. Por ejemplo en ARM7TDMI.
- Instrucciones Thumb = Subset de las instrucciones ARM.
  - **Trabajan con datos de 32 bits como las ARM**.
  - Su tamaño es de 16 bits (la mitad)
  - Su funcionalidad es acotada respecto de las ARM.
- Por lo tanto, hay cambios en la sintaxis:

  ARM `ADD R1,R1,R6`

  Thumb `ADD R1,R6`

- En modo Thumb el primer operando es al mismo tiempo el operando destino. Siempre se destruye el primer operando
- En modo ARM se conserva el primer operando reemplazando el registro resultado por otro diferente de los dos registros operando. Aquí tenemos rápidamente un primer ejemplo de perdida de funcionalidad en el Modo Thumb.

Unified Assembler Language (*UAL*)

- Las instrucciones de la versión v4T pueden tomarse de base para empezar a trabajar. Excepto SWI que fue considerada en 2007 obsoleta y recomendado el uso de SVI en su lugar, el resto se mantiene.
- Una forma de comenzar es mediante estas instrucciones e ir incorporando cuestiones mas novedosas de versiones posteriores
- Una cuestión muy interesante para homogeneizar el uso del lenguaje es lo que se conoce como Unified Assembler Language, *UAL*.

Unified Assembler Language (*UAL*)

- Las instrucciones de la versión v4T pueden tomarse de base para empezar a trabajar. Excepto SWI que fue considerada en 2007 obsoleta y recomendado el uso de SVI en su lugar, el resto se mantiene.
- Una forma de comenzar es mediante estas instrucciones e ir incorporando cuestiones mas novedosas de versiones posteriores
- Una cuestión muy interesante para homogeneizar el uso del lenguaje es lo que se conoce como Unified Assembler Language, *UAL*.

Unified Assembler Language (*UAL*)

- Las instrucciones de la versión v4T pueden tomarse de base para empezar a trabajar. Excepto SWI que fue considerada en 2007 obsoleta y recomendado el uso de SVI en su lugar, el resto se mantiene.
- Una forma de comenzar es mediante estas instrucciones e ir incorporando cuestiones mas novedosas de versiones posteriores
- Una cuestión muy interesante para homogeneizar el uso del lenguaje es lo que se conoce como Unified Assembler Language, *UAL*.

Unified Assembler Language (*UAL*)

- Las instrucciones de la versión v4T pueden tomarse de base para empezar a trabajar. Excepto SWI que fue considerada en 2007 obsoleta y recomendado el uso de SVI en su lugar, el resto se mantiene.
- Una forma de comenzar es mediante estas instrucciones e ir incorporando cuestiones mas novedosas de versiones posteriores
- Una cuestión muy interesante para homogeneizar el uso del lenguaje es lo que se conoce como Unified Assembler Language, ***UAL***.

Unified Assembler Language (*UAL*)

- Las instrucciones de la versión v4T pueden tomarse de base para empezar a trabajar. Excepto SWI que fue considerada en 2007 obsoleta y recomendado el uso de SVI en su lugar, el resto se mantiene.
- Una forma de comenzar es mediante estas instrucciones e ir incorporando cuestiones mas novedosas de versiones posteriores
- Una cuestión muy interesante para homogeneizar el uso del lenguaje es lo que se conoce como Unified Assembler Language, ***UAL***.

Formato ARM/Thumb

MOV <Rd >, <Rn >, LSL shift
LDR{cond}SB <Rd >,<Rd ,offset>
LDMFD sp!,{reglist}

Formato UAL

LSL{S}{Cond} <Rd >, <Rn >, shift
LDRSB{cond} <Rd >, <Rd ,#offset>
PUSH {cond}{reglist}

Ejecución condicional para evitar saltos

Ventajas de la ejecución condicional

- Evitar un salto es quitar una instrucción. Esto mejora la compresión del código.
- Las instrucciones se ejecutan si la condición establecida es `TRUE`.
- De lo contrario ejecutan No Operate, es decir un ciclo de clock idle y continúan con la instrucción sucesora secuencial (Es decir, la que está a continuación).
- Nunca hay branch. ¡¡Nunca se vacía el pipeline!!.

Ejecución condicional para evitar saltos

Ventajas de la ejecución condicional

- Evitar un salto es quitar una instrucción. Esto mejora la compresión del código.
- Las instrucciones se ejecutan si la condición establecida es `TRUE`.
- De lo contrario ejecutan No Operate, es decir un ciclo de clock idle y continúan con la instrucción sucesora secuencial (Es decir, la que está a continuación).
- Nunca hay branch. ¡¡Nunca se vacía el pipeline!!.

Ejecución condicional para evitar saltos

Ventajas de la ejecución condicional

- Evitar un salto es quitar una instrucción. Esto mejora la compresión del código.
- Las instrucciones se ejecutan si la condición establecida es **`TRUE`**.
- De lo contrario ejecutan No Operate, es decir un ciclo de clock idle y continúan con la instrucción sucesora secuencial (Es decir, la que está a continuación).
- Nunca hay branch. ¡¡Nunca se vacía el pipeline!!.

Ejecución condicional para evitar saltos

Ventajas de la ejecución condicional

- Evitar un salto es quitar una instrucción. Esto mejora la compresión del código.
- Las instrucciones se ejecutan si la condición establecida es **`TRUE`**.
- De lo contrario ejecutan No Operate, es decir un ciclo de clock idle y continúan con la instrucción sucesora secuencial (Es decir, la que está a continuación).
- Nunca hay branch. ¡¡Nunca se vacía el pipeline!!.

Ejecución condicional para evitar saltos

Ventajas de la ejecución condicional

- Evitar un salto es quitar una instrucción. Esto mejora la compresión del código.
- Las instrucciones se ejecutan si la condición establecida es **`TRUE`**.
- De lo contrario ejecutan No Operate, es decir un ciclo de clock idle y continúan con la instrucción sucesora secuencial (Es decir, la que está a continuación).
- Nunca hay branch. ¡¡Nunca se vacía el pipeline!!.

Ejecución condicional para evitar saltos

Ventajas de la ejecución condicional

- Evitar un salto es quitar una instrucción. Esto mejora la compresión del código.
- Las instrucciones se ejecutan si la condición establecida es **`TRUE`**.
- De lo contrario ejecutan No Operate, es decir un ciclo de clock idle y continúan con la instrucción sucesora secuencial (Es decir, la que está a continuación).
- Nunca hay branch. ¡¡Nunca se vacía el pipeline!!.

La ejecución condicional compara en runtime la condición establecida en los cuatro bits mas significativos del código de operación con los bits NCZV del registro CSPR.

Ejecución condicional - Como

- Una instrucción se ejecuta condicionalmente si:
  - Se ejecuta inmediatamente posterior a una instrucción que afectó los flags
  - Se ejecuta luego de varias otras instrucciones que no modifiquen los flags
  - Para ello existe un sufijo en el Mnemónico que se traduce en un bit en el opcode que permite ejecutar la instrucción afectando los flags NZCV
- Una instrucción condicional cuya evaluación de la condición resulta FALSE:
  - No ejecuta nada (equivale a la instrucción NOP)
  - No modifica ningún registro (ni siquiera el registro destino)
  - No afecta ningún flag
  - No genera ninguna excepción

Ejecución condicional - Como

- Una instrucción se ejecuta condicionalmente si:
  - Se ejecuta inmediatamente posterior a una instrucción que afectó los flags
  - Se ejecuta luego de varias otras instrucciones que no modifiquen los flags
  - Para ello existe un sufijo en el Mnemónico que se traduce en un bit en el opcode que permite ejecutar la instrucción afectando los flags NZCV.
- Una instrucción condicional cuya evaluación de la condición resulta FALSE:
  - No ejecuta nada (equivale a la instrucción NOP)
  - No modifica ningún registro (ni siquiera el registro destino)
  - No afecta ningún flag
  - No genera ninguna excepción

Ejecución condicional - Como

- Una instrucción se ejecuta condicionalmente si:
  - Se ejecuta inmediatamente posterior a una instrucción que afectó los flags
  - Se ejecuta luego de varias otras instrucciones que no modifiquen los flags
  - Para ello existe un sufijo en el Mnemónico que se traduce en un bit en el opcode que permite ejecutar la instrucción afectando los flags NZCV.
- Una instrucción condicional cuya evaluación de la condición resulta FALSE:
  - No ejecuta nada (equivale a la instrucción NOP)
  - No modifica ningún registro (ni siquiera el registro destino)
  - No afecta ningún flag
  - No genera ninguna excepción

Ejecución condicional - Como

- Una instrucción se ejecuta condicionalmente si:
  - Se ejecuta inmediatamente posterior a una instrucción que afectó los flags
  - Se ejecuta luego de varias otras instrucciones que no modifiquen los flags
  - Para ello existe un sufijo en el Mnemónico que se traduce en un bit en el opcode que permite ejecutar la instrucción afectando los flags NZCV.
- Una instrucción condicional cuya evaluación de la condición resulta FALSE:
  - No ejecuta nada (equivale a la instrucción NOP)
  - No modifica ningún registro (ni siquiera el registro de flags)
  - No afecta ningún flag
  - No genera ninguna excepción

Ejecución condicional - Como

- Una instrucción se ejecuta condicionalmente si:
  - Se ejecuta inmediatamente posterior a una instrucción que afectó los flags
  - Se ejecuta luego de varias otras instrucciones que no modifiquen los flags
  - Para ello existe un sufijo en el Mnemónico que se traduce en un bit en el opcode que permite ejecutar la instrucción afectando los flags NZCV.
- Una instrucción condicional cuya evaluación de la condición resulta FALSE:
  - No ejecuta nada (equivale a la instrucción NOP)
  - No modifica ninguna registro (ni siquiera el registro destino)
  - No afecta ningun flag
  - No genera ninguna excepción

Ejecución condicional - Como

- Una instrucción se ejecuta condicionalmente si:
  - Se ejecuta inmediatamente posterior a una instrucción que afectó los flags
  - Se ejecuta luego de varias otras instrucciones que no modifiquen los flags
  - Para ello existe un sufijo en el Mnemónico que se traduce en un bit en el opcode que permite ejecutar la instrucción afectando los flags NZCV.
- Una instrucción condicional cuya evaluación de la condición resulta FALSE:
  - No ejecuta nada (equivale a la instrucción NOP)
  - No modifica ningún registro (ni siquiera al registro destino)
  - No afecta ningún flag.
  - No genera ninguna excepción.

Ejecución condicional - Como

- Una instrucción se ejecuta condicionalmente si:
  - Se ejecuta inmediatamente posterior a una instrucción que afectó los flags
  - Se ejecuta luego de varias otras instrucciones que no modifiquen los flags
  - Para ello existe un sufijo en el Mnemónico que se traduce en un bit en el opcode que permite ejecutar la instrucción afectando los flags NZCV.
- Una instrucción condicional cuya evaluación de la condición resulta FALSE:
  - No ejecuta nada (equivale a la instrucción NOP)
  - No modifica ningún registro (ni siquiera al registro destino)
  - No afecta ningún flag.
  - No genera ninguna excepción.

Ejecución condicional - Como

- Una instrucción se ejecuta condicionalmente si:
  - Se ejecuta inmediatamente posterior a una instrucción que afectó los flags
  - Se ejecuta luego de varias otras instrucciones que no modifiquen los flags
  - Para ello existe un sufijo en el Mnemónico que se traduce en un bit en el opcode que permite ejecutar la instrucción afectando los flags NZCV.
- Una instrucción condicional cuya evaluación de la condición resulta FALSE:
  - No ejecuta nada (equivale a la instrucción NOP)
  - No modifica ningún registro (ni siquiera al registro destino)
  - No afecta ningún flag.
  - No genera ninguna excepción.

Ejecución condicional - Como

- Una instrucción se ejecuta condicionalmente si:
  - Se ejecuta inmediatamente posterior a una instrucción que afectó los flags
  - Se ejecuta luego de varias otras instrucciones que no modifiquen los flags
  - Para ello existe un sufijo en el Mnemónico que se traduce en un bit en el opcode que permite ejecutar la instrucción afectando los flags NZCV.
- Una instrucción condicional cuya evaluación de la condición resulta FALSE:
  - No ejecuta nada (equivale a la instrucción NOP)
  - No modifica ningún registro (ni siquiera al registro destino)
  - No afecta ningún flag.
  - No genera ninguna excepción.

Ejecución condicional - Como

- Una instrucción se ejecuta condicionalmente si:
  - Se ejecuta inmediatamente posterior a una instrucción que afectó los flags
  - Se ejecuta luego de varias otras instrucciones que no modifiquen los flags
  - Para ello existe un sufijo en el Mnemónico que se traduce en un bit en el opcode que permite ejecutar la instrucción afectando los flags NZCV.
- Una instrucción condicional cuya evaluación de la condición resulta FALSE:
  - No ejecuta nada (equivale a la instrucción NOP)
  - No modifica ningún registro (ni siquiera al registro destino)
  - No afecta ningún flag.
  - No genera ninguna excepción.

Flags de condición

Bit	Mn	Nombre	Descripción
31	N	Negative	’1’ si el resultado de una operación es un valor negativo. De otro modo ’0’.
30	Z	Zero	’1’ si el resultado de una operación es Cero. De otro modo ’0’.
29	C	Carry	’1’ si el resultado de una operación genera carry o si una resta genera borrow. De otro modo ’0’.
28	V	Overflow	’1’ si el resultado de una operación causa overflow. De otro modo ’0’.

Codificación

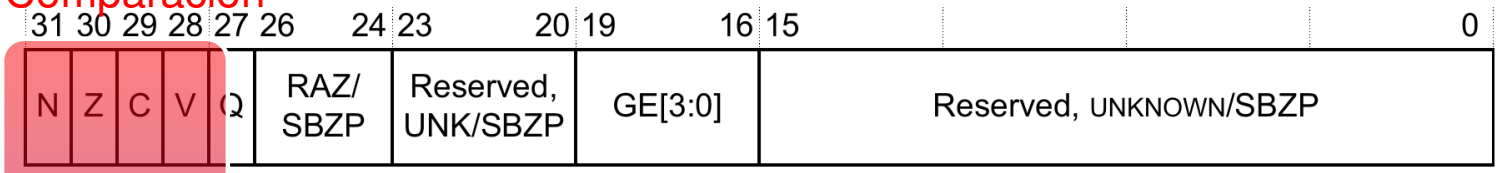
31	30	29	28	27	26 … 24	23 … 20	19 … 16	15 … 0
N	Z	C	V	Q	RAZ/ SBZP	Reserved, UNK/SBZP	GE[3:0]	Reserved, UNKNOWN/SBZP

APSR Register

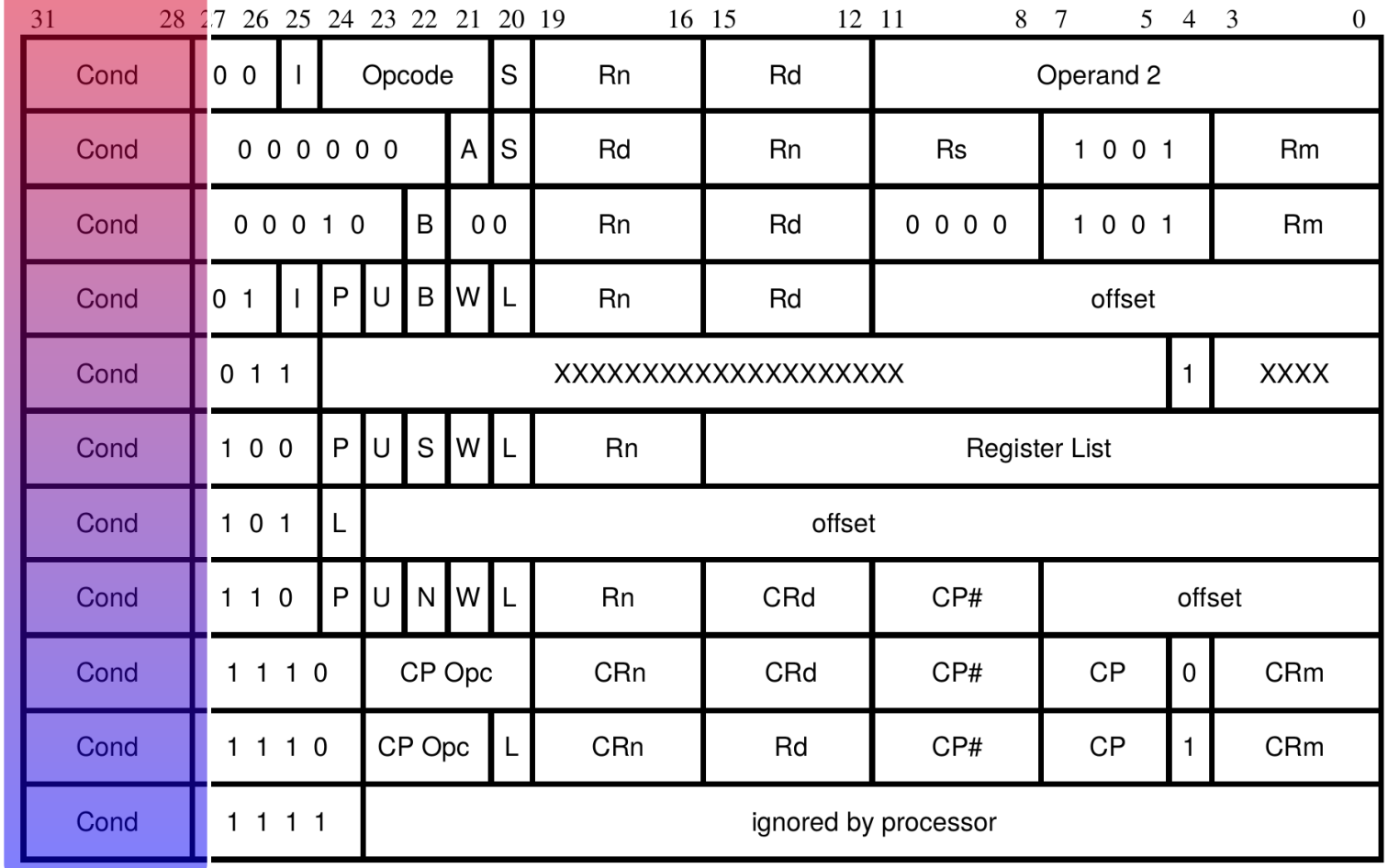
31 … 28	27	26	25	24	23	22	21	20	19 … 16	15 … 12	11 … 8	7 … 5	4	3 … 0	
Cond	0	0	I	Opcode				S	Rn	Rd	Operand 2				**Data Processing PSR Transfer**
Cond	0	0	0	0	0	0	A	S	Rd	Rn	Rs	1	0	0 1	**Multiply**
Cond	0	0	0	1	0	B	0	0	Rn	Rd	0 0 0 0	1	0	0 1	**Single Data Swap**
Cond	0	1	I	P	U	B	W	L	Rn	Rd	offset				**Single Data Transfer**
Cond	0	1	1	XXXXXXXXXXXXXXXXXXX				1	XXXX						**Undefined**
Cond	1	0	0	P	U	S	W	L	Rn	Register List					**Block Data Transfer**
Cond	1	0	1	L	offset										**Branch**
Cond	1	1	0	P	U	N	W	L	Rn	CRd	CP#	offset			**Coproc Data Transfer**
Cond	1	1	1	0	CP Opc				CRn	CRd	CP#	CP	0	CRm	**Coproc Data Operation**
Cond	1	1	1	0	CP Opc			L	CRn	Rd	CP#	CP	1	CRm	**Coproc Register Transfer**
Cond	1	1	1	1	ignored by processor										**Software Interrupt**

Codificación

Comparación



APSR Register



Códigos de condición y su significado

Mnemónico	Flag	Significado	Código
EQ	Z=1	Equal	0000
NE	Z=0	Not Equal	0001
CS/HS	C=1	$\geqslant$ No Signado	0010
CC/LO	C=0	$<$ No Signado	0011
MI	N=1	Negativo	0100
PL	N=0	Positivo o Cero	0101
VS	V=1	Overflow	0110
VC	V=0	No Overflow	0111
HI	C=1, Z=0	$>$ No Signado	1000
LS	C=0, Z=1	$\leqslant$ No Signado	1001
GE	$N \geqslant V$	$\geqslant$ Signado	1010
LT	$N \neq V$	$<$Signado	1011
GT	Z=0, N=V	$>$Signado	1100
LE	Z=1, $N \neq V$	$\leqslant$Signado	1101
AL	Siempre	Default	1110

Ejemplo en Modo ARM. Máximo Común Divisor

Algoritmo de Euclides

```
1 while (a != b) {  /*a y b son los dos números*/
2         if (a > b) a = a - b;
3         else b = b - a;
4 }
```

Ejemplo en Modo ARM. Máximo Común Divisor

Algoritmo de Euclides

```
1 while (a != b) {  /*a y b son los dos números*/
2         if (a > b) a = a - b;
3         else b = b - a;
4 }
```

Con saltos

```
1 gcd: CMP r0,r1      ;a > b?
2      BEQ end        ;a=b? termina
3      BLT less       ;a < b salta
4      SUB r0,r0,r1   ;a = a-b
5      B gcd          ;otro loop
6 less:SUB r1,r1,r0   ;b = b-a
7      B gcd
```

Ejemplo en Modo ARM. Máximo Común Divisor

Algoritmo de Euclides

```
1 while (a != b) {  /*a y b son los dos números*/
2         if (a > b) a = a - b;
3         else b = b - a;
4 }
```

Con saltos

```
1 gcd: CMP r0,r1      ;a > b?
2      BEQ end        ;a=b? termina
3      BLT less       ;a < b salta
4      SUB r0,r0,r1   ;a = a-b
5      B gcd          ;otro loop
6 less:SUB r1,r1,r0   ;b = b-a
7      B gcd
```

Ejecución Condicional

```
1 gcd:  CMP   r0, r1
2       SUBGT r0, r0, r1
3       SUBLT r1, r1, r0
4       BNE gcd
```

Ejemplo en Modo T32. Máximo Común Divisor

Algoritmo de Euclides

```
1 while (a != b) {  /*a y b son los dos números*/
2         if (a > b) a = a - b;
3         else b = b - a;
4 }
```

Con saltos

```
1 gcd: CMP r0,r1      ;a > b?
2      BEQ end        ;a=b? termina
3      BLT less       ;a < b salta
4      SUB r0,r0,r1   ;a = a-b
5      B gcd          ;otro loop
6 less:SUB r1,r1,r0   ;b = b-a
7      B gcd
```

Ejemplo en Modo T32. Máximo Común Divisor

Algoritmo de Euclides

```
1 while (a != b) {  /*a y b son los dos números*/
2         if (a > b) a = a - b;
3         else b = b - a;
4 }
```

Con saltos

```
1 gcd: CMP r0,r1      ;a > b?
2      BEQ end        ;a=b? termina
3      BLT less       ;a < b salta
4      SUB r0,r0,r1   ;a = a-b
5      B gcd          ;otro loop
6 less:SUB r1,r1,r0   ;b = b-a
7      B gcd
```

Ejecución Condicional

```
1 gcd:    ITE  GT
2         SUBGT r0, r0, r1
3         SUBLT r1, r1, r0
4         BNE gcd
```

Análisis temporal

- La eficiencia de la ejecución condicional respecto de la compresión de código es evidente: 7 instrucciones (28 bytes) se reducen a 4 tan solo instrucciones (16 bytes).
- Esto Resulta en una compresión del código de 43 % aproximadamente.
- Nos proponemos analizar la eficiencia de la ejecución condicional en términos de su impacto en el pipeline
- Para ello es necesario recordar que las instrucciones de ARM se ejecutan en 1 ciclo de clock, con excepciones.

Análisis temporal

- La eficiencia de la ejecución condicional respecto de la compresión de código es evidente: 7 instrucciones (28 bytes) se reducen a 4 tan solo instrucciones (16 bytes).
- Esto Resulta en una compresión del código de 43 % aproximadamente.
- Nos proponemos analizar la eficiencia de la ejecución condicional en términos de su impacto en el pipeline
- Para ello es necesario recordar que las instrucciones de ARM se ejecutan en 1 ciclo de clock, con excepciones.

Análisis temporal

- La eficiencia de la ejecución condicional respecto de la compresión de código es evidente: 7 instrucciones (28 bytes) se reducen a 4 tan solo instrucciones (16 bytes).
- Esto Resulta en una compresión del código de 43 % aproximadamente.
- Nos proponemos analizar la eficiencia de la ejecución condicional en términos de su impacto en el pipeline
- Para ello es necesario recordar que las instrucciones de ARM se ejecutan en 1 ciclo de clock, con excepciones.

Análisis temporal

- La eficiencia de la ejecución condicional respecto de la compresión de código es evidente: 7 instrucciones (28 bytes) se reducen a 4 tan solo instrucciones (16 bytes).
- Esto Resulta en una compresión del código de 43 % aproximadamente.
- Nos proponemos analizar la eficiencia de la ejecución condicional en términos de su impacto en el pipeline
- Para ello es necesario recordar que las instrucciones de ARM se ejecutan en 1 ciclo de clock, con excepciones.

Análisis temporal

- La eficiencia de la ejecución condicional respecto de la compresión de código es evidente: 7 instrucciones (28 bytes) se reducen a 4 tan solo instrucciones (16 bytes).
- Esto Resulta en una compresión del código de 43 % aproximadamente.
- Nos proponemos analizar la eficiencia de la ejecución condicional en términos de su impacto en el pipeline
- Para ello es necesario recordar que las instrucciones de ARM se ejecutan en 1 ciclo de clock, con excepciones.

Análisis temporal

- Los branches son una de esas excepciones. Tienen dos situaciones posibles:
  - [-] **Branch No Taken** No salta por lo tanto continúa con la instrucción siguiente al branch, a la que denominamos *sucesora secuencial*.
  - [-] **Branch Taken** Se rompe la secuencia de OPCODES NEXT. El PC cambia a la instrucción *Target* (se denomina así a la instrucción destino del salto). Se vacía el pipeline ya que las instrucciones que contiene corresponden a las sucesoras secuenciales.
- Un **Branch No Taken** toma 1 ciclo de clock y no ejecuta (no consume recursos de la CPU).
- Un **Branch Taken** consume 3 ciclos de clock (uno por cada etapa del pipeline que debe volver a llenarse).

Análisis temporal

- Los branches son una de esas excepciones. Tienen dos situaciones posibles:
  - [-] **Branch No Taken** No salta por lo tanto continúa con la instrucción siguiente al branch, a la que denominamos *sucesora secuencial*.
  - [-] **Branch Taken** Se rompe la secuencia de OPCODE FETCH. El PC cambia a la instrucción *Target* (se denomina así a la instrucción destino del salto). Se vacía el pipeline ya que las instrucciones que contiene corresponden a las *sucesoras secuenciales*.
- Un **Branch No Taken** toma 1 ciclo de clock y no ejecuta (no consume recursos de la CPU).
- Un **Branch Taken** consume 3 ciclos de clock (uno por cada etapa del pipeline que debe volver a llenarse).

Análisis temporal

- Los branches son una de esas excepciones. Tienen dos situaciones posibles:
  - [-] **Branch No Taken** No salta por lo tanto continúa con la instrucción siguiente al branch, a la que denominamos *sucesora secuencial*.
  - [-] **Branch Taken** Se rompe la secuencia de OPCODE FETCH. El PC cambia a la instrucción *Target* (se denomina así a la instrucción destino del salto). Se vacía el pipeline ya que las instrucciones que contiene corresponden a las *sucesoras secuenciales*.
- Un **Branch No Taken** toma 1 ciclo de clock y no ejecuta (no consume recursos de la CPU).
- Un **Branch Taken** consume 3 ciclos de clock (uno por cada etapa del pipeline que debe volver a llenarse).

Análisis temporal

- Los branches son una de esas excepciones. Tienen dos situaciones posibles:
  - [-] **Branch No Taken** No salta por lo tanto continúa con la instrucción siguiente al branch, a la que denominamos *sucesora secuencial*.
  - [-] **Branch Taken** Se rompe la secuencia de OPCODE FETCH. El PC cambia a la instrucción *Target* (se denomina así a la instrucción destino del salto). Se vacía el pipeline ya que las instrucciones que contiene corresponden a las *sucesoras secuenciales*.
- Un **Branch No Taken** toma 1 ciclo de clock y no ejecuta (no consume recursos de la CPU).
- Un **Branch Taken** consume 3 ciclos de clock (uno por cada etapa del pipeline que debe volver a llenarse).

Análisis temporal

- Los branches son una de esas excepciones. Tienen dos situaciones posibles:
  - [-] **Branch No Taken** No salta por lo tanto continúa con la instrucción siguiente al branch, a la que denominamos *sucesora secuencial*.
  - [-] **Branch Taken** Se rompe la secuencia de OPCODE FETCH. El PC cambia a la instrucción *Target* (se denomina así a la instrucción destino del salto). Se vacía el pipeline ya que las instrucciones que contiene corresponden a las *sucesoras secuenciales*.
- Un **Branch No Taken** toma 1 ciclo de clock y no ejecuta (no consume recursos de la CPU).
- Un **Branch Taken** consume 3 ciclos de clock (uno por cada etapa del pipeline que debe volver a llenarse).

Análisis temporal

- Los branches son una de esas excepciones. Tienen dos situaciones posibles:
  - [-] **Branch No Taken** No salta por lo tanto continúa con la instrucción siguiente al branch, a la que denominamos *sucesora secuencial*.
  - [-] **Branch Taken** Se rompe la secuencia de OPCODE FETCH. El PC cambia a la instrucción *Target* (se denomina así a la instrucción destino del salto). Se vacía el pipeline ya que las instrucciones que contiene corresponden a las *sucesoras secuenciales*.
- Un **Branch No Taken** toma 1 ciclo de clock y no ejecuta (no consume recursos de la CPU).
- Un **Branch Taken** consume 3 ciclos de clock (uno por cada etapa del pipeline que debe volver a llenarse).

Análisis temporal

R0:a	R1:b	Instrucción	Ciclos (ARM7)
1	2	CMP r0, r1	1
1	2	BEQ end	1 (no-taken)
1	2	BLT less	3 (taken)
1	2	SUB r1,r1,r0	1
1	2	B gcd	3 (taken)
1	1	CMP r0, r1	1
1	1	BEQ end	3 (taken)
			Total: 13

R0:a	R1:b	Instrucción	Ciclos (ARM7)
1	2	CMP r0, r1	1
1	2	SUBGT r0,r0,r1	1(no ejecutada)
1	1	SUBLT r1,r1,r0	1
1	1	BNE gcd	3 (taken)
1	1	CMP r0, r1	1
1	1	SUBGT r0,r0,r1	1 (no ejecutada
1	1	SUBLT r1,r1,r0	1(no ejecutada)
1	1	BNE gcd	1 (no taken)
			Total: 10

Usando Branches

```
1 gcd:  CMP  r0,r1
2       BEQ  end
3       BLT  less
4       SUB  r0,r0,r1
5       B  gcd
6 less:SUB  r1,r1,r0
7       B  gcd
```

Ejecución Condicional

```
1 gcd:  CMP    r0,  r1
2       SUBGT  r0,  r0,  r1
3       SUBLT  r1,  r1,  r0
4       BNE  gcd
```

Ejecución condicional en Modo Thumb-2 (T32)

- Las instrucciones Thumb son de 16 bits. No hay espacio para los cuatro de condición
- Las instrucciones Thumb-2 de 32 bits (T32) tienen esos bits en 1111 (combinación inválida de código de condición), de modo de mantiene el mecanismo de ejecución condicional Thumb.
- La diferencia es que en Modo Thumb en caso de generarse el salto la distancia es desde la instrucción de salto -256 bytes o +254 bytes, mientras en el Modo Thumb 2 al poder trabajar en determinadas instrucciones con 32 bits de código de operación extiende esa capacidad a $\pm$ 1Mbyte.
- La construcción IT (IF THEN) del ejemplo anterior es la solución que encontraron para resolver el problema.

Ejecución condicional en Modo Thumb-2 (T32)

- Las instrucciones Thumb son de 16 bits. No hay espacio para los cuatro de condición
- Las instrucciones Thumb-2 de 32 bits (T32) tienen esos bits en 1111 (combinación inválida de código de condición), de modo de mantiene el mecanismo de ejecución condicional Thumb.
- La diferencia es que en Modo Thumb en caso de generarse el salto la distancia es desde la instrucción de salto -256 bytes o +254 bytes, mientras en el Modo Thumb 2 al poder trabajar en determinadas instrucciones con 32 bits de código de operación extiende esa capacidad a $\pm$ 1Mbyte.
- La construcción IT (IF THEN) del ejemplo anterior es la solución que encontraron para resolver el problema.

Ejecución condicional en Modo Thumb-2 (T32)

- Las instrucciones Thumb son de 16 bits. No hay espacio para los cuatro de condición
- Las instrucciones Thumb-2 de 32 bits (T32) tienen esos bits en 1111 (combinación inválida de código de condición), de modo de mantiene el mecanismo de ejecución condicional Thumb.
- La diferencia es que en Modo Thumb en caso de generarse el salto la distancia es desde la instrucción de salto -256 bytes o +254 bytes, mientras en el Modo Thumb 2 al poder trabajar en determinadas instrucciones con 32 bits de código de operación extiende esa capacidad a $\pm$ 1Mbyte.
- La construcción IT (IF THEN) del ejemplo anterior es la solución que encontraron para resolver el problema.

Ejecución condicional en Modo Thumb-2 (T32)

- Las instrucciones Thumb son de 16 bits. No hay espacio para los cuatro de condición
- Las instrucciones Thumb-2 de 32 bits (T32) tienen esos bits en 1111 (combinación inválida de código de condición), de modo de mantiene el mecanismo de ejecución condicional Thumb.
- La diferencia es que en Modo Thumb en caso de generarse el salto la distancia es desde la instrucción de salto -256 bytes o +254 bytes, mientras en el Modo Thumb 2 al poder trabajar en determinadas instrucciones con 32 bits de código de operación extiende esa capacidad a $\pm$ 1Mbyte.
- La construcción IT (IF THEN) del ejemplo anterior es la solución que encontraron para resolver el problema.

Ejecución condicional en Modo Thumb-2 (T32)

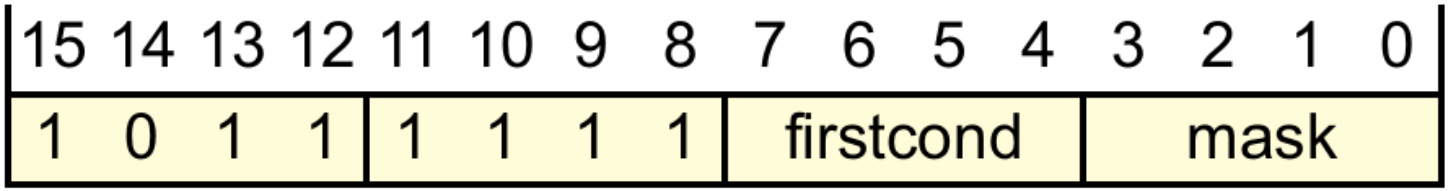
- Las instrucciones Thumb son de 16 bits. No hay espacio para los cuatro de condición
- Las instrucciones Thumb-2 de 32 bits (T32) tienen esos bits en 1111 (combinación inválida de código de condición), de modo de mantiene el mecanismo de ejecución condicional Thumb.
- La diferencia es que en Modo Thumb en caso de generarse el salto la distancia es desde la instrucción de salto -256 bytes o +254 bytes, mientras en el Modo Thumb 2 al poder trabajar en determinadas instrucciones con 32 bits de código de operación extiende esa capacidad a $\pm$ 1Mbyte.
- La construcción IT (IF THEN) del ejemplo anterior es la solución que encontraron para resolver el problema.

Ejecución condicional en Modo Thumb-2 (T32)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	firstcond				mask			

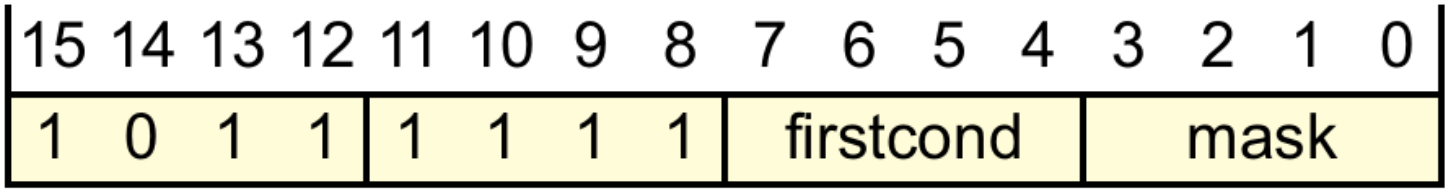
- `IT xyz`, es una instrucción de 16 bits que evalua la condición que le viene codificada en ***firstcond***.
- x condición para la segunda instrucción del bloque IT.
- y condición para la tercer instrucción del bloque IT.
- z condición para la cuarta instrucción del bloque IT.
- T es la condición para ***firstcond*** TRUE
- E es la condición para ***firstcond*** FALSE
- ***No soporta mas de tres instrucciones luego del bloque IT***

Ejecución condicional en Modo Thumb-2 (T32)



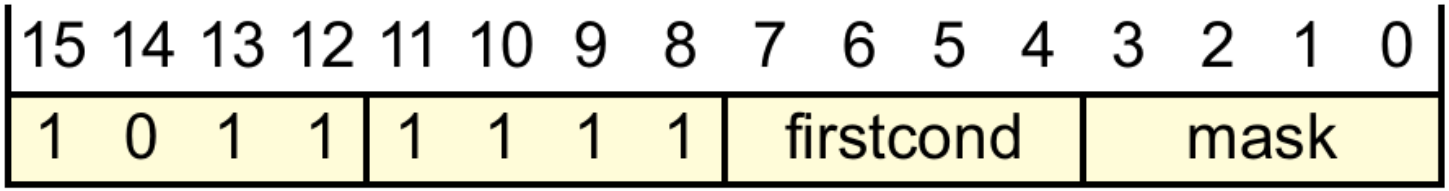
- `IT xyz`, es una instrucción de 16 bits que evalua la condición que le viene codificada en ***firstcond***.
- x condición para la segunda instrucción del bloque IT.
- y condición para la tercer instrucción del bloque IT.
- z condición para la cuarta instrucción del bloque IT.
- T es la condición para ***firstcond*** TRUE
- E es la condición para ***firstcond*** FALSE
- ***No soporta mas de tres instrucciones luego del bloque IT***

Ejecución condicional en Modo Thumb-2 (T32)



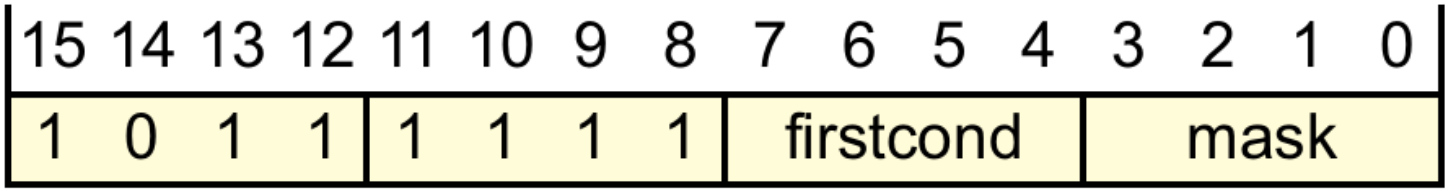
- `IT xyz`, es una instrucción de 16 bits que evalua la condición que le viene codificada en ***firstcond***.
- x condición para la segunda instrucción del bloque IT.
- y condición para la tercer instrucción del bloque IT.
- z condición para la cuarta instrucción del bloque IT.
- T es la condición para ***firstcond*** TRUE
- E es la condición para ***firstcond*** FALSE
- ***No soporta mas de tres instrucciones luego del bloque IT***

Ejecución condicional en Modo Thumb-2 (T32)



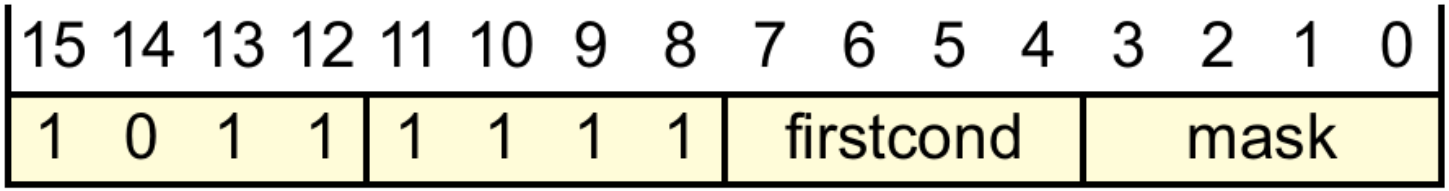
- `IT xyz`, es una instrucción de 16 bits que evalua la condición que le viene codificada en ***firstcond***.
- x condición para la segunda instrucción del bloque IT.
- y condición para la tercer instrucción del bloque IT.
- z condición para la cuarta instrucción del bloque IT.
- T es la condición para ***firstcond*** TRUE
- E es la condición para ***firstcond*** FALSE
- ***No soporta mas de tres instrucciones luego del bloque IT***

Ejecución condicional en Modo Thumb-2 (T32)



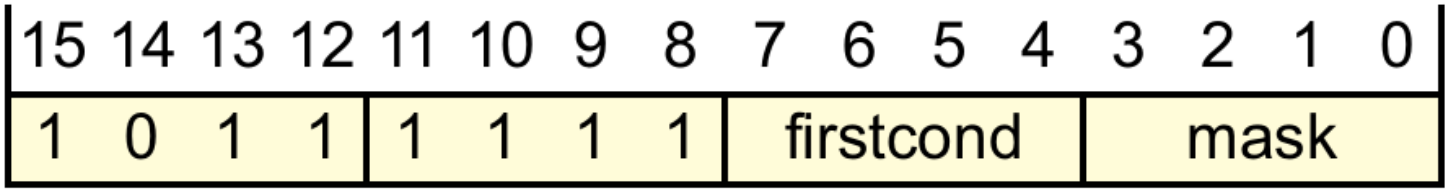
- `IT xyz`, es una instrucción de 16 bits que evalua la condición que le viene codificada en ***firstcond***.
- x condición para la segunda instrucción del bloque IT.
- y condición para la tercer instrucción del bloque IT.
- z condición para la cuarta instrucción del bloque IT.
- T es la condición para ***firstcond*** TRUE
- E es la condición para ***firstcond*** FALSE
- ***No soporta mas de tres instrucciones luego del bloque IT***

Ejecución condicional en Modo Thumb-2 (T32)



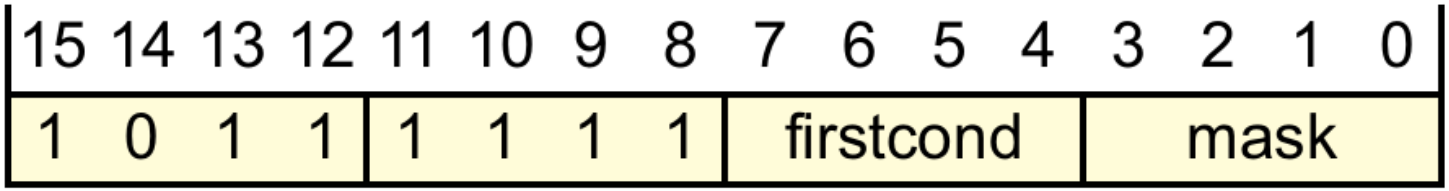
- `IT xyz`, es una instrucción de 16 bits que evalua la condición que le viene codificada en ***firstcond***.
- x condición para la segunda instrucción del bloque IT.
- y condición para la tercer instrucción del bloque IT.
- z condición para la cuarta instrucción del bloque IT.
- T es la condición para ***firstcond*** TRUE
- E es la condición para ***firstcond*** FALSE
- ***No soporta mas de tres instrucciones luego del bloque IT***

Ejecución condicional en Modo Thumb-2 (T32)



- `IT xyz`, es una instrucción de 16 bits que evalua la condición que le viene codificada en ***firstcond***.
- x condición para la segunda instrucción del bloque IT.
- y condición para la tercer instrucción del bloque IT.
- z condición para la cuarta instrucción del bloque IT.
- T es la condición para ***firstcond*** TRUE
- E es la condición para ***firstcond*** FALSE
- ***No soporta mas de tres instrucciones luego del bloque IT***

Ejecución condicional en Modo Thumb-2 (T32)



- `IT xyz`, es una instrucción de 16 bits que evalua la condición que le viene codificada en ***firstcond***.
- x condición para la segunda instrucción del bloque IT.
- y condición para la tercer instrucción del bloque IT.
- z condición para la cuarta instrucción del bloque IT.
- T es la condición para ***firstcond*** TRUE
- E es la condición para ***firstcond*** FALSE
- ***No soporta mas de tres instrucciones luego del bloque IT***

Actualización de los flags en ARM y T32

- Las instrucciones utilizadas en los ejemplos anteriores **no actualizan los flags luego de su ejecución**.
- De este modo es posible proceder en forma condicional a continuación de una suma, en base al estado de máquina establecido por una instrucción previa a la suma.
- Si la suma afectase el carry alteraría la condición original que regula la ejecución del bloque condicional.
- Sin embargo la conducta esperada de una instrucción de suma es que afecte los flags (No solo el carry).
- Cuando las mismas instrucciones no se ejecutan en forma condicional llevan un sufijo **S** en su Opcode de modo que el ensamblador utilice un código de operación diferente y la Unidad de Control luego de la ejecución actualice los flags correspondientes.

Actualización de los flags en ARM y T32

- Las instrucciones utilizadas en los ejemplos anteriores **no actualizan los flags luego de su ejecución**.
- De este modo es posible proceder en forma condicional a continuación de una suma, en base al estado de máquina establecido por una instrucción previa a la suma.
- Si la suma afectase el carry alteraría la condición original que regula la ejecución del bloque condicional.
- Sin embargo la conducta esperada de una instrucción de suma es que afecte los flags (No solo el carry).
- Cuando las mismas instrucciones no se ejecutan en forma condicional llevan un sufijo **S** en su Opcode de modo que el ensamblador utilice un código de operación diferente y la Unidad de Control luego de la ejecución actualice los flags correspondientes.

Actualización de los flags en ARM y T32

- Las instrucciones utilizadas en los ejemplos anteriores **no actualizan los flags luego de su ejecución**.
- De este modo es posible proceder en forma condicional a continuación de una suma, en base al estado de máquina establecido por una instrucción previa a la suma.
- Si la suma afectase el carry alteraría la condición original que regula la ejecución del bloque condicional.
- Sin embargo la conducta esperada de una instrucción de suma es que afecte los flags (No solo el carry).
- Cuando las mismas instrucciones no se ejecutan en forma condicional llevan un sufijo **S** en su Opcode de modo que el ensamblador utilice un código de operación diferente y la Unidad de Control luego de la ejecución actualice los flags correspondientes.

Actualización de los flags en ARM y T32

- Las instrucciones utilizadas en los ejemplos anteriores **no actualizan los flags luego de su ejecución**.
- De este modo es posible proceder en forma condicional a continuación de una suma, en base al estado de máquina establecido por una instrucción previa a la suma.
- Si la suma afectase el carry alteraría la condición original que regula la ejecución del bloque condicional.
- Sin embargo la conducta esperada de una instrucción de suma es que afecte los flags (No solo el carry).
- Cuando las mismas instrucciones no se ejecutan en forma condicional llevan un sufijo **S** en su Opcode de modo que el ensamblador utilice un código de operación diferente y la Unidad de Control luego de la ejecución actualice los flags correspondientes.

Actualización de los flags en ARM y T32

- Las instrucciones utilizadas en los ejemplos anteriores **no actualizan los flags luego de su ejecución**.
- De este modo es posible proceder en forma condicional a continuación de una suma, en base al estado de máquina establecido por una instrucción previa a la suma.
- Si la suma afectase el carry alteraría la condición original que regula la ejecución del bloque condicional.
- Sin embargo la conducta esperada de una instrucción de suma es que afecte los flags (No solo el carry).
- Cuando las mismas instrucciones no se ejecutan en forma condicional llevan un sufijo **S** en su Opcode de modo que el ensamblador utilice un código de operación diferente y la Unidad de Control luego de la ejecución actualice los flags correspondientes.

Actualización de los flags en ARM y T32

- Las instrucciones utilizadas en los ejemplos anteriores **no actualizan los flags luego de su ejecución**.
- De este modo es posible proceder en forma condicional a continuación de una suma, en base al estado de máquina establecido por una instrucción previa a la suma.
- Si la suma afectase el carry alteraría la condición original que regula la ejecución del bloque condicional.
- Sin embargo la conducta esperada de una instrucción de suma es que afecte los flags (No solo el carry).
- Cuando las mismas instrucciones no se ejecutan en forma condicional llevan un sufijo **S** en su Opcode de modo que el ensamblador utilice un código de operación diferente y la Unidad de Control luego de la ejecución actualice los flags correspondientes.

Especificadores de ancho

- Dos sufijos para las instrucciones T32: **.W** y **.N**
- Controlan el tamaño de los códigos que se generarán para las instrucciones T32.
- En modo T32 **.W** fuerza al ensamblador a generar código de 32 bits, aun cuando sea posible generar esa instrucción con codificación de 16 bits.
- **.W** no tiene efecto si se trabaja en Modo ARM (A32).
- En código T32 **.N** fuerza al ensamblador a generar código de 16 bits. En caso de no ser posible codificar la instrucción T32 en 16 bits, o si **.N** se aplica a código A32, el ensamblador arrojará error y no generará código.

Especificadores de ancho

- Dos sufijos para las instrucciones T32: **.W** y **.N**
- Controlan el tamaño de los códigos que se generarán para las instrucciones T32.
- En modo T32 **.W** fuerza al ensamblador a generar código de 32 bits, aun cuando sea posible generar esa instrucción con codificación de 16 bits.
- **.W** no tiene efecto si se trabaja en Modo ARM (A32).
- En código T32 **.N** fuerza al ensamblador a generar código de 16 bits. En caso de no ser posible codificar la instrucción T32 en 16 bits, o si **.N** se aplica a código A32, el ensamblador arrojará error y no generará código.

Especificadores de ancho

- Dos sufijos para las instrucciones T32: **.W** y **.N**
- Controlan el tamaño de los códigos que se generarán para las instrucciones T32.
- En modo T32 **.W** fuerza al ensamblador a generar código de 32 bits, aun cuando sea posible generar esa instrucción con codificación de 16 bits.
- **.W** no tiene efecto si se trabaja en Modo ARM (A32).
- En código T32 **.N** fuerza al ensamblador a generar código de 16 bits. En caso de no ser posible codificar la instrucción T32 en 16 bits, o si **.N** se aplica a código A32, el ensamblador arrojará error y no generará código.

Especificadores de ancho

- Dos sufijos para las instrucciones T32: **.W** y **.N**
- Controlan el tamaño de los códigos que se generarán para las instrucciones T32.
- En modo T32 **.W** fuerza al ensamblador a generar código de 32 bits, aun cuando sea posible generar esa instrucción con codificación de 16 bits.
- **.W** no tiene efecto si se trabaja en Modo ARM (A32).
- En código T32 **.N** fuerza al ensamblador a generar código de 16 bits. En caso de no ser posible codificar la instrucción T32 en 16 bits, o si **.N** se aplica a código A32, el ensamblador arrojará error y no generará código.

Especificadores de ancho

- Dos sufijos para las instrucciones T32: **.W** y **.N**
- Controlan el tamaño de los códigos que se generarán para las instrucciones T32.
- En modo T32 **.W** fuerza al ensamblador a generar código de 32 bits, aun cuando sea posible generar esa instrucción con codificación de 16 bits.
- **.W** no tiene efecto si se trabaja en Modo ARM (A32).
- En código T32 **.N** fuerza al ensamblador a generar código de 16 bits. En caso de no ser posible codificar la instrucción T32 en 16 bits, o si **.N** se aplica a código A32, el ensamblador arrojará error y no generará código.

Especificadores de ancho

- Dos sufijos para las instrucciones T32: **.W** y **.N**
- Controlan el tamaño de los códigos que se generarán para las instrucciones T32.
- En modo T32 **.W** fuerza al ensamblador a generar código de 32 bits, aun cuando sea posible generar esa instrucción con codificación de 16 bits.
- **.W** no tiene efecto si se trabaja en Modo ARM (A32).
- En código T32 **.N** fuerza al ensamblador a generar código de 16 bits. En caso de no ser posible codificar la instrucción T32 en 16 bits, o si **.N** se aplica a código A32, el ensamblador arrojará error y no generará código.

Dos ejemplos interesantes

```
1 BCS.W label ;Fuerza códidgo de 32-bit aun para salto corto.
2 B.N   label ;Falla si label cae fuera de rango para código de 16-bit.
```

El segundo operando

- Puede ser una constante. `#valor`.
- Puede ser un registro. `Rm`
- El registro puede llevar desplazamiento. `Rm, {shift}`
  - `shift` es un valor constante, o controlado por otro registro, pero que se aplica al registro `Rm`
  - Si omitimos `shift`, la instrucción usará directamente el valor almacenado en `Rm`

El segundo operando

- Puede ser una constante. **`#valor`**.
- Puede ser un registro. `Rm`
- El registro puede llevar desplazamiento. `Rm, {shift}`
  - `shift` es un valor constante, o controlado por otro registro, pero que se aplica al registro `Rm`
  - Si omitimos `shift`, la instrucción usa directamente el valor almacenado en `Rm`

El segundo operando

- Puede ser una constante. **`#valor`**.
- Puede ser un registro. **`Rm`**
- El registro puede llevar desplazamiento. **`Rm, {shift}`**
  - **`shift`** es un valor constante, o controlado por otro registro, pero que se aplica al registro **`Rm`**
  - Si omitimos **`shift`**, la instrucción usará directamente el valor almacenado en **`Rm`**

El segundo operando

- Puede ser una constante. **`#valor`**.
- Puede ser un registro. **`Rm`**
- El registro puede llevar desplazamiento. **`Rm, {shift}`**
  - **`shift`** es un valor constante, o controlado por otro registro, pero que se aplica al registro **`Rm`**
    - **`ASR #n`**. Arithmetic Shift Right, **`#n`** bits, con $1 \leqslant n \leqslant 32$
    - **`LSL #n`**. Logical Shift Left, **`#n`** bits, con $1 \leqslant n \leqslant 31$
    - **`LSR #n`**. Logical Shift Right, **`#n`** bits, con $1 \leqslant n \leqslant 32$
    - **`ROR #n`**. Rotate Right, **`#n`** bits, con $1 \leqslant n \leqslant 31$
    - **`RRX`**. Rotate Right, one bit with extend.
    - **`type Rs`**. Controlado por registro. **`type`** es **`ASR`**, **`LSL`**, **`LSR`** o **`ROR`**, y **`Rs`** es el registro en el que se especifica la cantidad de bits
    - En este caso no hay desplazamiento (se aplica **`LSL #0`**)
  - Si omitimos **`shift`**, la instrucción usará directamente el calor almacenado en **`Rm`**

El segundo operando

- Puede ser una constante. **`#valor`**.
- Puede ser un registro. **`Rm`**
- El registro puede llevar desplazamiento. **`Rm, {shift}`**
  - **`shift`** es un valor constante, o controlado por otro registro, pero que se aplica al registro **`Rm`**
    - **`ASR #n`**. **A**rithmetic **S**hift **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 32$
    - **`LSL #n`**. **L**ogical **S**hift **L**eft, **`#n`** bits, con $1 \leqslant n \leqslant 31$
    - **`LSR #n`**. **L**ogical **S**hift **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 32$
    - **`ROR #n`**. **RO**tate **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 31$
    - **`RRX #n`**. **R**otate **R**ight, one bit with e**X**tend.
    - ***`type Rs`***. Controlado por registro: ***`type`*** es **ASR**, **LSL**, **LSR**, o **ROR**,y ***`Rn`*** es el registro en el que se especifica la cantidad de bits.
    - –. En este caso no hay desplazamiento (se aplica **`LSL #0`**).
  - Si omitimos **`shift`**, la instrucción usará directamente el calor almacenado en **`Rm`**

El segundo operando

- Puede ser una constante. **`#valor`**.
- Puede ser un registro. **`Rm`**
- El registro puede llevar desplazamiento. **`Rm, {shift}`**
  - **`shift`** es un valor constante, o controlado por otro registro, pero que se aplica al registro **`Rm`**
    - **`ASR #n`**. **A**rithmetic **S**hift **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 32$
    - **`LSL #n`**. **L**ogical **S**hift **L**eft, **`#n`** bits, con $1 \leqslant n \leqslant 31$
    - **`LSR #n`**. **L**ogical **S**hift **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 32$
    - **`ROR #n`**. **RO**tate **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 31$
    - **`RRX #n`**. **R**otate **R**ight, one bit with e**X**tend.
    - ***`type Rs`***. Controlado por registro: ***`type`*** es **ASR**, **LSL**, **LSR**, o **ROR**,y ***`Rn`*** es el registro en el que se especifica la cantidad de bits.
    - –. En este caso no hay desplazamiento (se aplica **`LSL #0`**).
  - Si omitimos **`shift`**, la instrucción usará directamente el calor almacenado en **`Rm`**

El segundo operando

- Puede ser una constante. **`#valor`**.
- Puede ser un registro. **`Rm`**
- El registro puede llevar desplazamiento. **`Rm, {shift}`**
  - **`shift`** es un valor constante, o controlado por otro registro, pero que se aplica al registro **`Rm`**
    - **`ASR #n`**. **A**rithmetic **S**hift **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 32$
    - **`LSL #n`**. **L**ogical **S**hift **L**eft, **`#n`** bits, con $1 \leqslant n \leqslant 31$
    - **`LSR #n`**. **L**ogical **S**hift **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 32$
    - **`ROR #n`**. **RO**tate **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 31$
    - **`RRX #n`**. **R**otate **R**ight, one bit with e**X**tend.
    - ***`type Rs`***. Controlado por registro: ***`type`*** es **ASR**, **LSL**, **LSR**, o **ROR**,y ***`Rn`*** es el registro en el que se especifica la cantidad de bits.
    - **–**. En este caso no hay desplazamiento (se aplica **`LSL #0`**).
  - Si omitimos **`shift`**, la instrucción usará directamente el calor almacenado en **`Rm`**

El segundo operando

- Puede ser una constante. **`#valor`**.
- Puede ser un registro. **`Rm`**
- El registro puede llevar desplazamiento. **`Rm, {shift}`**
  - **`shift`** es un valor constante, o controlado por otro registro, pero que se aplica al registro **`Rm`**
    - **`ASR #n`**. **A**rithmetic **S**hift **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 32$
    - **`LSL #n`**. **L**ogical **S**hift **L**eft, **`#n`** bits, con $1 \leqslant n \leqslant 31$
    - **`LSR #n`**. **L**ogical **S**hift **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 32$
    - **`ROR #n`**. **RO**tate **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 31$
    - **`RRX #n`**. **R**otate **R**ight, one bit with e**X**tend.
    - ***`type Rs`***. Controlado por registro: ***`type`*** es **ASR**, **LSL**, **LSR**, o **ROR**,y ***`Rn`*** es el registro en el que se especifica la cantidad de bits.
    - **–**. En este caso no hay desplazamiento (se aplica **`LSL #0`**).
  - Si omitimos **`shift`**, la instrucción usará directamente el calor almacenado en **`Rm`**

El segundo operando

- Puede ser una constante. **`#valor`**.
- Puede ser un registro. **`Rm`**
- El registro puede llevar desplazamiento. **`Rm, {shift}`**
  - **`shift`** es un valor constante, o controlado por otro registro, pero que se aplica al registro **`Rm`**
    - **`ASR #n`**. **A**rithmetic **S**hift **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 32$
    - **`LSL #n`**. **L**ogical **S**hift **L**eft, **`#n`** bits, con $1 \leqslant n \leqslant 31$
    - **`LSR #n`**. **L**ogical **S**hift **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 32$
    - **`ROR #n`**. **RO**tate **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 31$
    - **`RRX #n`**. **R**otate **R**ight, one bit with e**X**tend.
    - ***`type Rs`***. Controlado por registro: ***`type`*** es **ASR**, **LSL**, **LSR**, o **ROR**,y ***`Rn`*** es el registro en el que se especifica la cantidad de bits.
    - **–**. En este caso no hay desplazamiento (se aplica **`LSL #0`**).
  - Si omitimos **`shift`**, la instrucción usará directamente el calor almacenado en **`Rm`**

El segundo operando

- Puede ser una constante. **`#valor`**.
- Puede ser un registro. **`Rm`**
- El registro puede llevar desplazamiento. **`Rm, {shift}`**
  - **`shift`** es un valor constante, o controlado por otro registro, pero que se aplica al registro **`Rm`**
    - **`ASR #n`**. **A**rithmetic **S**hift **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 32$
    - **`LSL #n`**. **L**ogical **S**hift **L**eft, **`#n`** bits, con $1 \leqslant n \leqslant 31$
    - **`LSR #n`**. **L**ogical **S**hift **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 32$
    - **`ROR #n`**. **RO**tate **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 31$
    - **`RRX #n`**. **R**otate **R**ight, one bit with e**X**tend.
    - ***`type Rs`***. Controlado por registro: ***`type`*** es **ASR**, **LSL**, **LSR**, o **ROR**,y ***`Rn`*** es el registro en el que se especifica la cantidad de bits.
    - **–**. En este caso no hay desplazamiento (se aplica **`LSL #0`**).
  - Si omitimos **`shift`**, la instrucción usará directamente el calor almacenado en **`Rm`**

El segundo operando

- Puede ser una constante. **`#valor`**.
- Puede ser un registro. **`Rm`**
- El registro puede llevar desplazamiento. **`Rm, {shift}`**
  - **`shift`** es un valor constante, o controlado por otro registro, pero que se aplica al registro **`Rm`**
    - **`ASR #n`**. **A**rithmetic **S**hift **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 32$
    - **`LSL #n`**. **L**ogical **S**hift **L**eft, **`#n`** bits, con $1 \leqslant n \leqslant 31$
    - **`LSR #n`**. **L**ogical **S**hift **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 32$
    - **`ROR #n`**. **RO**tate **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 31$
    - **`RRX #n`**. **R**otate **R**ight, one bit with e**X**tend.
    - ***`type Rs`***. Controlado por registro: ***`type`*** es **ASR**, **LSL**, **LSR**, o **ROR**,y ***`Rn`*** es el registro en el que se especifica la cantidad de bits.
    - –. En este caso no hay desplazamiento (se aplica **`LSL #0`**).
  - Si omitimos **`shift`**, la instrucción usará directamente el calor almacenado en **`Rm`**

El segundo operando

- Puede ser una constante. **`#valor`**.
- Puede ser un registro. **`Rm`**
- El registro puede llevar desplazamiento. **`Rm, {shift}`**
  - **`shift`** es un valor constante, o controlado por otro registro, pero que se aplica al registro **`Rm`**
    - **`ASR #n`**. **A**rithmetic **S**hift **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 32$
    - **`LSL #n`**. **L**ogical **S**hift **L**eft, **`#n`** bits, con $1 \leqslant n \leqslant 31$
    - **`LSR #n`**. **L**ogical **S**hift **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 32$
    - **`ROR #n`**. **RO**tate **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 31$
    - **`RRX #n`**. **R**otate **R**ight, one bit with e**X**tend.
    - ***`type Rs`***. Controlado por registro: ***`type`*** es **ASR**, **LSL**, **LSR**, o **ROR**,y ***`Rn`*** es el registro en el que se especifica la cantidad de bits.
    - **–**. En este caso no hay desplazamiento (se aplica **`LSL #0`**).
  - Si omitimos **`shift`**, la instrucción usará directamente el calor almacenado en **`Rm`**

El segundo operando

- Puede ser una constante. **`#valor`**.
- Puede ser un registro. **`Rm`**
- El registro puede llevar desplazamiento. **`Rm, {shift}`**
  - **`shift`** es un valor constante, o controlado por otro registro, pero que se aplica al registro **`Rm`**
    - **`ASR #n`**. **A**rithmetic **S**hift **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 32$
    - **`LSL #n`**. **L**ogical **S**hift **L**eft, **`#n`** bits, con $1 \leqslant n \leqslant 31$
    - **`LSR #n`**. **L**ogical **S**hift **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 32$
    - **`ROR #n`**. **RO**tate **R**ight, **`#n`** bits, con $1 \leqslant n \leqslant 31$
    - **`RRX #n`**. **R**otate **R**ight, one bit with e**X**tend.
    - ***`type Rs`***. Controlado por registro: ***`type`*** es **ASR**, **LSL**, **LSR**, o **ROR**,y ***`Rn`*** es el registro en el que se especifica la cantidad de bits.
    - **`–`**. En este caso no hay desplazamiento (se aplica **`LSL #0`**).
  - Si omitimos **`shift`**, la instrucción usará directamente el calor almacenado en **`Rm`**

Operaciones de shift

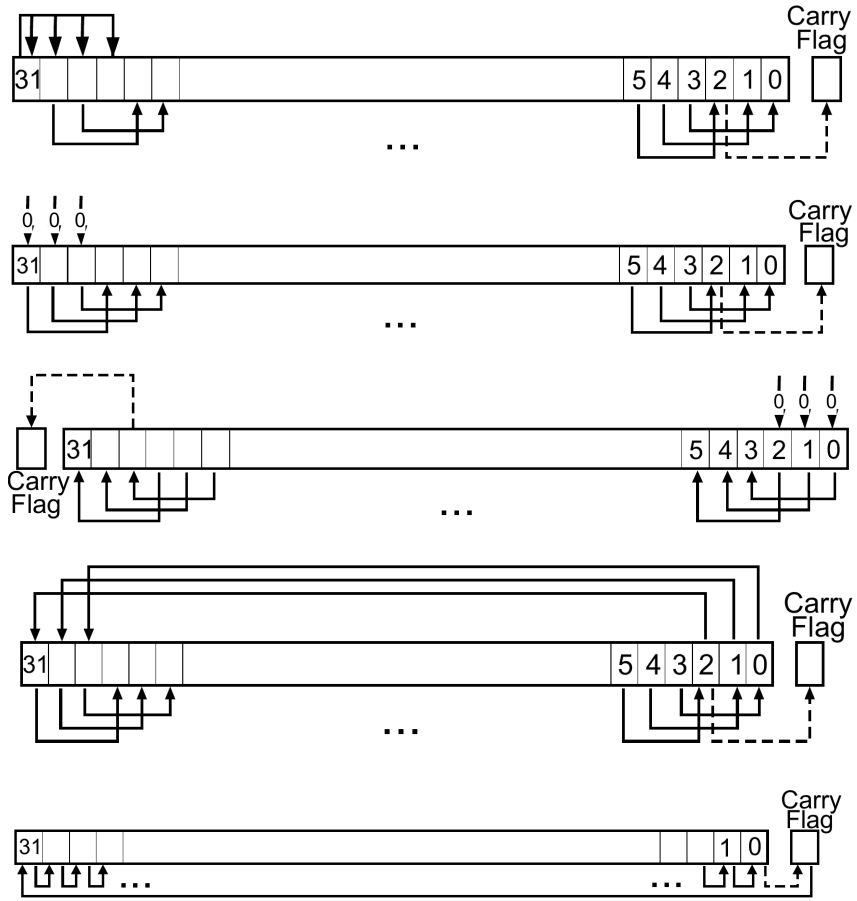
Arithmetic Shift Right

Logical Shift Right

Logical Shift Left

Rotate Right

Rotate Right with eXtend



Lectura: Load

- Lectura simple
  - Direccionamiento Inmediato
  - Direccionamiento Relativo a Program Counter (R15)
  - Direccionamiento con Offset en Registro
- Lectura Múltiple

Load con Desplazamiento Inmediato

- Sintaxis

```
LDR{type}{cond} Rt, [Rn {, #offset}] ; immediate offset
LDR{type}{cond} Rt, [Rn, #offset]!   ; pre-indexed
LDR{type}{cond} Rt, [Rn], #offset    ; post-indexed
LDRD{cond} Rt, Rt2, [Rn {, #offset}] ; immediate offset, doublewor
LDRD{cond} Rt, Rt2, [Rn, #offset]!   ; pre-indexed, doubleword
LDRD{cond} Rt, Rt2, [Rn], #offset    ; post-indexed, doubleword
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, **-** Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

Rn: Registro puntero a memoria.

#offset: Offset (optativo). Si se incluye se suma a *Rn*

Rt2: Si la operación es de doble word, este registro carga la word alta.

Load con Desplazamiento Inmediato

- Sintaxis

```
LDR{type}{cond} Rt, [Rn {, #offset}] ; immediate offset
LDR{type}{cond} Rt, [Rn, #offset]!   ; pre-indexed
LDR{type}{cond} Rt, [Rn], #offset    ; post-indexed
LDRD{cond} Rt, Rt2, [Rn {, #offset}] ; immediate offset, doublewor
LDRD{cond} Rt, Rt2, [Rn, #offset]!   ; pre-indexed, doubleword
LDRD{cond} Rt, Rt2, [Rn], #offset    ; post-indexed, doubleword
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, **-** Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

Rn: Registro puntero a memoria.

#offset: Offset (optativo). Si se incluye se suma a *Rn*

Rt2: Si la operación es de doble word, este registro carga la word alta.

Load con Desplazamiento Inmediato

- Sintaxis

```
LDR{type}{cond} Rt, [Rn {, #offset}] ; immediate offset
LDR{type}{cond} Rt, [Rn, #offset]!   ; pre-indexed
LDR{type}{cond} Rt, [Rn], #offset    ; post-indexed
LDRD{cond} Rt, Rt2, [Rn {, #offset}] ; immediate offset, doublewor
LDRD{cond} Rt, Rt2, [Rn, #offset]!   ; pre-indexed, doubleword
LDRD{cond} Rt, Rt2, [Rn], #offset    ; post-indexed, doubleword
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, **-** Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

Rn: Registro puntero a memoria.

#offset: Offset (optativo). Si se incluye se suma a *Rn*

Rt2: Si la operación es de doble word, este registro carga la word alta.

Load con Desplazamiento Inmediato

- Sintaxis

```
LDR{type}{cond} Rt, [Rn {, #offset}] ; immediate offset
LDR{type}{cond} Rt, [Rn, #offset]!   ; pre-indexed
LDR{type}{cond} Rt, [Rn], #offset    ; post-indexed
LDRD{cond} Rt, Rt2, [Rn {, #offset}] ; immediate offset, doublewor
LDRD{cond} Rt, Rt2, [Rn, #offset]!   ; pre-indexed, doubleword
LDRD{cond} Rt, Rt2, [Rn], #offset    ; post-indexed, doubleword
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, **-** Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

Rn: Registro puntero a memoria.

#offset: Offset (optativo). Si se incluye se suma a *Rn*

Rt2: Si la operación es de doble word, este registro carga la word alta.

Load con Desplazamiento Inmediato

- Sintaxis

```
LDR{type}{cond} Rt, [Rn {, #offset}] ; immediate offset
LDR{type}{cond} Rt, [Rn, #offset]!   ; pre-indexed
LDR{type}{cond} Rt, [Rn], #offset    ; post-indexed
LDRD{cond} Rt, Rt2, [Rn {, #offset}] ; immediate offset, doublewor
LDRD{cond} Rt, Rt2, [Rn, #offset]!   ; pre-indexed, doubleword
LDRD{cond} Rt, Rt2, [Rn], #offset    ; post-indexed, doubleword
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, **-** Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

Rn: Registro puntero a memoria.

#offset: Offset (optativo). Si se incluye se suma a *Rn*

Rt2: Si la operación es de doble word, este registro carga la word alta.

Load con Desplazamiento Inmediato

- Sintaxis

```
LDR{type}{cond} Rt, [Rn {, #offset}] ; immediate offset
LDR{type}{cond} Rt, [Rn, #offset]!   ; pre-indexed
LDR{type}{cond} Rt, [Rn], #offset    ; post-indexed
LDRD{cond} Rt, Rt2, [Rn {, #offset}] ; immediate offset, doublewor
LDRD{cond} Rt, Rt2, [Rn, #offset]!   ; pre-indexed, doubleword
LDRD{cond} Rt, Rt2, [Rn], #offset    ; post-indexed, doubleword
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, **-** Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

Rn: Registro puntero a memoria.

#offset: Offset (optativo). Si se incluye se suma a *Rn*

Rt2: Si la operación es de doble word, este registro carga la word alta.

Load con Desplazamiento Inmediato

- Sintaxis

```
LDR{type}{cond} Rt, [Rn {, #offset}] ; immediate offset
LDR{type}{cond} Rt, [Rn, #offset]!   ; pre-indexed
LDR{type}{cond} Rt, [Rn], #offset    ; post-indexed
LDRD{cond} Rt, Rt2, [Rn {, #offset}] ; immediate offset, doublewor
LDRD{cond} Rt, Rt2, [Rn, #offset]!   ; pre-indexed, doubleword
LDRD{cond} Rt, Rt2, [Rn], #offset    ; post-indexed, doubleword
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, **-** Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

Rn: Registro puntero a memoria.

#offset: Offset (optativo). Si se incluye se suma a *Rn*

Rt2: Si la operación es de doble word, este registro carga la word alta.

Ejemplos

```
1  LDR     r8,[r10]       ;carga R8 desde la dirección apuntada por R10.
2  LDRNE   r2,[r5,#960]!  ;Carga R2 condicionalmente desde la dirección
3                         ;apuntada por (R5+960), e incrementa R5 en 960.
4  ;Indexación = Autoincremento. Tres casos:
5  LDR     r0,[r4, #4]    ;offset simple: R0 = *(int*)(R4+4); R4 no cambia
6  LDR     r0,[r4, #4]!   ;pre-indexed:   R0 = *(int*)(R4+4); R4 = R4+4
7  LDR     r0,[r4], #4    ;post-indexed:  R0 = *(int*)(R4+0); R4 = R4+4
8  ;Indexdo con desplazamiento
9  LDR     r9,[r10, r2, LSL #4] ;r9 = *(int*) (r10 + (r2 << 4) )
10 ;Lectura de Doble word -> Solo disponible en Modo ARM
11 LDRD    r6,r7,[r10]    ; r7 = *(int*)(r10) y r6 = *(int*)(r10+4)
12 ; es equivalente a esta secuencia
13 LDR     r6,[r10]
14 LDR     r7,[r10, #4]
15 ;Números signados
16 LDRSH   r5,[r0]        ; Valores iniciales r0 = 0x5F006EC0, r5 =
       0x12345678
17                        ; Memoria:  0x5F006EC0  0xFA----
18                        ;           0x5F006EC1  0x91--|---
19                        ;           0x5F006EC2  0x23  |  |
20                        ;           0x5F006EC3  0xA5  |  |
21                        ; Resultado r5 = 0xFFFF91FA<--   |
22                        ;                  ^-------------
```

Load Relativo al Program Counter

- Sintaxis

```
LDR{type}{cond}{.W} Rt, label
LDRD{cond} Rt, Rt2, label ; Doubleword
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, **-** Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

.W: Especificador de ancho de instrucción.

Label: El el número a sumar al PC (en Ca2)

Rt2: Si la operación es de doble word, este registro carga la word alta.

- Ejemplo:

Load Relativo al Program Counter

- Sintaxis

```
LDR{type}{cond}{.W} Rt, label
LDRD{cond} Rt, Rt2, label ; Doubleword
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, **-** Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

.W: Especificador de ancho de instrucción.

Label: El el número a sumar al PC (en Ca2)

Rt2: Si la operación es de doble word, este registro carga la word alta.

- Ejemplo:

Load Relativo al Program Counter

- Sintaxis

```
LDR{type}{cond}{.W} Rt, label
LDRD{cond} Rt, Rt2, label ; Doubleword
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, **-** Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

.W: Especificador de ancho de instrucción.

Label: El el número a sumar al PC (en Ca2)

Rt2: Si la operación es de doble word, este registro carga la word alta.

- Ejemplo:

Load Relativo al Program Counter

- Sintaxis

```
LDR{type}{cond}{.W} Rt, label
LDRD{cond} Rt, Rt2, label ; Doubleword
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, **-** Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

.W: Especificador de ancho de instrucción.

Label: El el número a sumar al PC (en Ca2)

Rt2: Si la operación es de doble word, este registro carga la word alta.

- Ejemplo:

Load Relativo al Program Counter

- Sintaxis

```
LDR{type}{cond}{.W} Rt, label
LDRD{cond} Rt, Rt2, label ; Doubleword
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, **-** Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

.W: Especificador de ancho de instrucción.

Label: El el número a sumar al PC (en Ca2)

Rt2: Si la operación es de doble word, este registro carga la word alta.

- Ejemplo:

Load Relativo al Program Counter

- Sintaxis

```
LDR{type}{cond}{.W} Rt, label
LDRD{cond} Rt, Rt2, label ; Doubleword
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, **-** Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

.W: Especificador de ancho de instrucción.

Label: El el número a sumar al PC (en Ca2)

Rt2: Si la operación es de doble word, este registro carga la word alta.

- Ejemplo:

Load Relativo al Program Counter

- Sintaxis

```
LDR{type}{cond}{.W} Rt, label
LDRD{cond} Rt, Rt2, label ; Doubleword
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, **-** Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

.W: Especificador de ancho de instrucción.

Label: El el número a sumar al PC (en Ca2)

Rt2: Si la operación es de doble word, este registro carga la word alta.

- Ejemplo:

Load Relativo al Program Counter

- Sintaxis

```
LDR{type}{cond}{.W} Rt, label
LDRD{cond} Rt, Rt2, label ; Doubleword
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, **-** Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

.W: Especificador de ancho de instrucción.

Label: El el número a sumar al PC (en Ca2)

Rt2: Si la operación es de doble word, este registro carga la word alta.

- Ejemplo:

```
1 .section .text
2 .global _start
3 _start:
4     ldr r0, =jump          /* load the address of the function label
          jump into R0 */
5     ....
6 jump:
7     ldr r2, =511           /* load the value 511 into R2 */
8     bkpt
```

Load con Desplazamiento en Registro

- Sintaxis

```
1 LDR{type}{cond} Rt,[Rn,± Rm {shift}] ;offset registro
2 LDR{type}{cond} Rt,[Rn,± Rm {shift}]!;pre-indexed; (A32)
3 LDR{type}{cond} Rt,[Rn],± Rm {shift};post-indexed; (A32)
4 LDRD{cond} Rt, Rt2,[Rn,± Rm] ;offset registro, dword; (A32)
5 LDRD{cond} Rt, Rt2,[Rn,± Rm]!;pre-indexed, dword; (A32)
6 LDRD{cond} Rt, Rt2,[Rn],± Rm;post-indexed, dword; (A32)
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, - Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

Rn: Registro puntero a memoria.

Rm: Registro que contiene el Offset. No permitido en T32

Rt2: Si la operación es de doble word, este registro carga la word alta.

- Ejemplos:

Load con Desplazamiento en Registro

- Sintaxis

```
1 LDR{type}{cond} Rt,[Rn,± Rm {shift}] ;offset registro
2 LDR{type}{cond} Rt,[Rn,± Rm {shift}]!;pre-indexed; (A32)
3 LDR{type}{cond} Rt,[Rn],± Rm {shift};post-indexed; (A32)
4 LDRD{cond} Rt, Rt2,[Rn,± Rm] ;offset registro, dword; (A32)
5 LDRD{cond} Rt, Rt2,[Rn,± Rm]!;pre-indexed, dword; (A32)
6 LDRD{cond} Rt, Rt2,[Rn],± Rm;post-indexed, dword; (A32)
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, **-** Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

Rn: Registro puntero a memoria.

Rm: Registro que contiene el Offset. No permitido en T32

Rt2: Si la operación es de doble word, este registro carga la word alta.

- Ejemplos:

Load con Desplazamiento en Registro

- Sintaxis

```
1 LDR{type}{cond} Rt,[Rn,± Rm {shift}] ;offset registro
2 LDR{type}{cond} Rt,[Rn,± Rm {shift}]!;pre-indexed; (A32)
3 LDR{type}{cond} Rt,[Rn],± Rm {shift};post-indexed; (A32)
4 LDRD{cond} Rt, Rt2,[Rn,± Rm] ;offset registro, dword; (A32)
5 LDRD{cond} Rt, Rt2,[Rn,± Rm]!;pre-indexed, dword; (A32)
6 LDRD{cond} Rt, Rt2,[Rn],± Rm;post-indexed, dword; (A32)
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, **-** Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

Rn: Registro puntero a memoria.

Rm: Registro que contiene el Offset. No permitido en T32

Rt2: Si la operación es de doble word, este registro carga la word alta.

- Ejemplos:

Load con Desplazamiento en Registro

- Sintaxis

```
1 LDR{type}{cond} Rt,[Rn,± Rm {shift}] ;offset registro
2 LDR{type}{cond} Rt,[Rn,± Rm {shift}]!;pre-indexed; (A32)
3 LDR{type}{cond} Rt,[Rn],± Rm {shift};post-indexed; (A32)
4 LDRD{cond} Rt, Rt2,[Rn,± Rm] ;offset registro, dword; (A32)
5 LDRD{cond} Rt, Rt2,[Rn,± Rm]!;pre-indexed, dword; (A32)
6 LDRD{cond} Rt, Rt2,[Rn],± Rm;post-indexed, dword; (A32)
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, **-** Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

Rn: Registro puntero a memoria.

Rm: Registro que contiene el Offset. No permitido en T32

Rt2: Si la operación es de doble word, este registro carga la word alta.

- Ejemplos:

Load con Desplazamiento en Registro

- Sintaxis

```
1 LDR{type}{cond} Rt,[Rn,± Rm {shift}] ;offset registro
2 LDR{type}{cond} Rt,[Rn,± Rm {shift}]!;pre-indexed; (A32)
3 LDR{type}{cond} Rt,[Rn],± Rm {shift};post-indexed; (A32)
4 LDRD{cond} Rt, Rt2,[Rn,± Rm] ;offset registro, dword; (A32)
5 LDRD{cond} Rt, Rt2,[Rn,± Rm]!;pre-indexed, dword; (A32)
6 LDRD{cond} Rt, Rt2,[Rn],± Rm;post-indexed, dword; (A32)
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, **-** Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

Rn: Registro puntero a memoria.

Rm: Registro que contiene el Offset. No permitido en T32

Rt2: Si la operación es de doble word, este registro carga la word alta.

- Ejemplos:

Load con Desplazamiento en Registro

- Sintaxis

```
1 LDR{type}{cond} Rt,[Rn,± Rm {shift}] ;offset registro
2 LDR{type}{cond} Rt,[Rn,± Rm {shift}]!;pre-indexed; (A32)
3 LDR{type}{cond} Rt,[Rn],± Rm {shift};post-indexed; (A32)
4 LDRD{cond} Rt, Rt2,[Rn,± Rm] ;offset registro, dword; (A32)
5 LDRD{cond} Rt, Rt2,[Rn,± Rm]!;pre-indexed, dword; (A32)
6 LDRD{cond} Rt, Rt2,[Rn],± Rm;post-indexed, dword; (A32)
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, **-** Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

Rn: Registro puntero a memoria.

Rm: Registro que contiene el Offset. No permitido en T32

Rt2: Si la operación es de doble word, este registro carga la word alta.

- Ejemplos:

Load con Desplazamiento en Registro

- Sintaxis

```
1 LDR{type}{cond} Rt,[Rn,± Rm {shift}] ;offset registro
2 LDR{type}{cond} Rt,[Rn,± Rm {shift}]!;pre-indexed; (A32)
3 LDR{type}{cond} Rt,[Rn],± Rm {shift};post-indexed; (A32)
4 LDRD{cond} Rt, Rt2,[Rn,± Rm] ;offset registro, dword; (A32)
5 LDRD{cond} Rt, Rt2,[Rn,± Rm]!;pre-indexed, dword; (A32)
6 LDRD{cond} Rt, Rt2,[Rn],± Rm;post-indexed, dword; (A32)
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, **-** Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

Rn: Registro puntero a memoria.

Rm: Registro que contiene el Offset. No permitido en T32

Rt2: Si la operación es de doble word, este registro carga la word alta.

- Ejemplos:

Load con Desplazamiento en Registro

- Sintaxis

```
1 LDR{type}{cond} Rt,[Rn,± Rm {shift}] ;offset registro
2 LDR{type}{cond} Rt,[Rn,± Rm {shift}]!;pre-indexed; (A32)
3 LDR{type}{cond} Rt,[Rn],± Rm {shift};post-indexed; (A32)
4 LDRD{cond} Rt, Rt2,[Rn,± Rm] ;offset registro, dword; (A32)
5 LDRD{cond} Rt, Rt2,[Rn,± Rm]!;pre-indexed, dword; (A32)
6 LDRD{cond} Rt, Rt2,[Rn],± Rm;post-indexed, dword; (A32)
```

type: **B** unsigned Byte, **H** unsigned Half-word, **SB** Signed Byte, **SH** Signed Half-word, **-** Word

cond: Código de Condición (opcional)

Rt: Registro a cargar (operando destino)

Rn: Registro puntero a memoria.

Rm: Registro que contiene el Offset. No permitido en T32

Rt2: Si la operación es de doble word, este registro carga la word alta.

- Ejemplos:

```
1 LDR r1,[r0,r3]     ;Carga r1 con la word apuntada por r0+r3
2 LDRD r1,r2 [r0,r3];Igual que el caso anterior pero lee dword
3 LDR r1,[r0],r3     ;Primer caso pero con post-incremento.
```

Load Multiple

- Sintaxis

```
LDM{addr_mode}{cond} Rn{!}, reglist{^}
```

addr_mode: Tratamiento de *Rn*: **IA** Increment After, **IB** Increment Before, **DA** Decrement After, **DB** Decrement Before.

cond: Código de Condición (opcional)

Rn: Registro puntero Base a memoria.

!: Escribe dirección final en *Rn*

reglist: Lista de registros destino. Puede contener rangos

- Ejemplos:

Load Multiple

- Sintaxis

```
LDM{addr_mode}{cond} Rn{!}, reglist{^}
```

addr_mode: Tratamiento de *Rn*: **IA** Increment After, **IB** Increment Before, **DA** Decrement After, **DB** Decrement Before.

cond: Código de Condición (opcional)

Rn: Registro puntero Base a memoria.

!: Escribe dirección final en *Rn*

reglist: Lista de registros destino. Puede contener rangos

- Ejemplos:

Load Multiple

- Sintaxis

```
LDM{addr_mode}{cond} Rn{!}, reglist{^}
```

addr_mode: Tratamiento de *Rn*: **IA** Increment After, **IB** Increment Before, **DA** Decrement After, **DB** Decrement Before.

cond: Código de Condición (opcional)

Rn: Registro puntero Base a memoria.

!: Escribe dirección final en *Rn*

reglist: Lista de registros destino. Puede contener rangos

- Ejemplos:

Load Multiple

- Sintaxis

```
LDM{addr_mode}{cond} Rn{!}, reglist{^}
```

addr_mode: Tratamiento de *Rn*: **IA** Increment After, **IB** Increment Before, **DA** Decrement After, **DB** Decrement Before.

cond: Código de Condición (opcional)

Rn: Registro puntero Base a memoria.

!: Escribe dirección final en *Rn*

reglist: Lista de registros destino. Puede contener rangos

- Ejemplos:

Load Multiple

- Sintaxis

```
LDM{addr_mode}{cond} Rn{!}, reglist{^}
```

addr_mode: Tratamiento de *Rn*: **IA** Increment After, **IB** Increment Before, **DA** Decrement After, **DB** Decrement Before.

cond: Código de Condición (opcional)

Rn: Registro puntero Base a memoria.

!: Escribe dirección final en *Rn*

reglist: Lista de registros destino. Puede contener rangos

- Ejemplos:

Load Multiple

- Sintaxis

```
LDM{addr_mode}{cond} Rn{!}, reglist{^}
```

addr_mode: Tratamiento de *Rn*: **IA** Increment After, **IB** Increment Before, **DA** Decrement After, **DB** Decrement Before.

cond: Código de Condición (opcional)

Rn: Registro puntero Base a memoria.

!: Escribe dirección final en *Rn*

reglist: Lista de registros destino. Puede contener rangos

- Ejemplos:

Load Multiple

- Sintaxis

```
LDM{addr_mode}{cond} Rn{!}, reglist{^}
```

addr_mode: Tratamiento de *Rn*: **IA** Increment After, **IB** Increment Before, **DA** Decrement After, **DB** Decrement Before.

cond: Código de Condición (opcional)

Rn: Registro puntero Base a memoria.

!: Escribe dirección final en *Rn*

reglist: Lista de registros destino. Puede contener rangos

- Ejemplos:

```
1 LDMIA r7,{r0-r3,r8,r12} ;Carga r0 a r3, r8 y r9, con las seis words
2                         ;almacenadas a partir de la dirección
3                         ;apuntada por r7
```

Escrituras: Store

- La sintaxis es la misma, a excepción del Mnemónico, que en este caso es STR, STRD (para dobles word)
- Modos de direccionamiento: Offset inmediato, y Offset Registro.
- Para bytes y hwords no extiende signo (tamaño destino idéntico).
- Ejemplos:

Escrituras: Store

- La sintaxis es la misma, a excepción del Mnemónico, que en este caso es STR, STRD (para dobles word)
- Modos de direccionamiento: Offset inmediato, y Offset Registro.
- Para bytes y hwords no extiende signo (tamaño destino idéntico).
- Ejemplos:

Escrituras: Store

- La sintaxis es la misma, a excepción del Mnemónico, que en este caso es STR, STRD (para dobles word)
- Modos de direccionamiento: Offset inmediato, y Offset Registro.
- Para bytes y hwords no extiende signo (tamaño destino idéntico).
- Ejemplos:

Escrituras: Store

- La sintaxis es la misma, a excepción del Mnemónico, que en este caso es STR, STRD (para dobles word)
- Modos de direccionamiento: Offset inmediato, y Offset Registro.
- Para bytes y hwords no extiende signo (tamaño destino idéntico).
- Ejemplos:

Escrituras: Store

- La sintaxis es la misma, a excepción del Mnemónico, que en este caso es STR, STRD (para dobles word)
- Modos de direccionamiento: Offset inmediato, y Offset Registro.
- Para bytes y hwords no extiende signo (tamaño destino idéntico).
- Ejemplos:

```
1  STR     r8,[r10]       ;Escribe R8 en la dirección apuntada por R10.
2  STRNE   r2,[r5,#960]!  ;Escribe R2 condicionalmente en la dirección
3                         ;apuntada por (R5+960), e incrementa R5 en 960.
4  ;Indexación = Autoincremento. Tres casos:
5  STR     r0,[r4, #4]    ;offset simple: *(int*)(R4+4) = R0; R4 no cambia
6  STR     r0,[r4, #4]!   ;pre-indexed:   *(int*)(R4+4) = R0; R4 = R4+4
7  STR     r0,[r4], #4    ;post-indexed:  *(int*)(R4+0) = R0; R4 = R4+4
8  ;Indexdo con desplazamiento
9  STR     r9,[r10, r2, LSL #4] ;*(int*) (r10 + (r2 << 4) ) = r9
10 ;Lectura de Doble word -> Solo disponible en Modo ARM
11 STRD    r6,r7,[r10]    ; *(int*)(r10) = r7 y *(int*)(r10+4) = r6
12 ; es equivalente a esta secuencia
13 STR     r6,[r10]
14 STR     r7,[r10, #4]
15 STMIA   r5,{r0-r3}     ; Al igual que LDM IA es el default
```

“One of the best things about learning assembly language is that you deal directly with hardware, and as a result, learn about computer architecture in a very direct way ”.

ARM Assembly Language
William Hohl, Christopher Hinds.

Rango de Direccionamiento

- ARM al igual que otras arquitecturas tiene tamaño de instrucciones fijo en 32bits.
- Sin embargo muchas veces se necesita trabajar con operandos inmediatos de 32 bits. Por ejemplo una dirección de memoria.
- En particular la arquitectura ARM mapea en memoria todos los dispositivos de E/S, de modo que cuando se los quiera acceder hay que escribir su dirección en un registro del core para utilizarlo como puntero.
- Por ejemplo, en un NXP4337 (Cortex-M4), los diferentes periféricos se despliegan a partir de la dirección de memoria 0x40000000.
- ¿Como es posible almacenar esta dirección en un registro del core para utilizarlo como puntero base?.
- La instrucción que se requiere es del tipo:

Rango de Direccionamiento

- ARM al igual que otras arquitecturas tiene tamaño de instrucciones fijo en 32bits.
- Sin embargo muchas veces se necesita trabajar con operandos inmediatos de 32 bits. Por ejemplo una dirección de memoria.
- En particular la arquitectura ARM mapea en memoria todos los dispositivos de E/S, de modo que cuando se los quiera acceder hay que escribir su dirección en un registro del core para utilizarlo como puntero.
- Por ejemplo, en un NXP4337 (Cortex-M4), los diferentes periféricos se despliegan a partir de la dirección de memoria 0x40000000.
- ¿Como es posible almacenar esta dirección en un registro del core para utilizarlo como puntero base?.
- La instrucción que se requiere es del tipo:

Rango de Direccionamiento

- ARM al igual que otras arquitecturas tiene tamaño de instrucciones fijo en 32bits.
- Sin embargo muchas veces se necesita trabajar con operandos inmediatos de 32 bits. Por ejemplo una dirección de memoria.
- En particular la arquitectura ARM mapea en memoria todos los dispositivos de E/S, de modo que cuando se los quiera acceder hay que escribir su dirección en un registro del core para utilizarlo como puntero.
- Por ejemplo, en un NXP4337 (Cortex-M4), los diferentes periféricos se despliegan a partir de la dirección de memoria 0x40000000.
- ¿Como es posible almacenar esta dirección en un registro del core para utilizarlo como puntero base?.
- La instrucción que se requiere es del tipo:

Rango de Direccionamiento

- ARM al igual que otras arquitecturas tiene tamaño de instrucciones fijo en 32bits.
- Sin embargo muchas veces se necesita trabajar con operandos inmediatos de 32 bits. Por ejemplo una dirección de memoria.
- En particular la arquitectura ARM mapea en memoria todos los dispositivos de E/S, de modo que cuando se los quiera acceder hay que escribir su dirección en un registro del core para utilizarlo como puntero.
- Por ejemplo, en un NXP4337 (Cortex-M4), los diferentes periféricos se despliegan a partir de la dirección de memoria 0x40000000.
- ¿Como es posible almacenar esta dirección en un registro del core para utilizarlo como puntero base?.
- La instrucción que se requiere es del tipo:

Rango de Direccionamiento

- ARM al igual que otras arquitecturas tiene tamaño de instrucciones fijo en 32bits.
- Sin embargo muchas veces se necesita trabajar con operandos inmediatos de 32 bits. Por ejemplo una dirección de memoria.
- En particular la arquitectura ARM mapea en memoria todos los dispositivos de E/S, de modo que cuando se los quiera acceder hay que escribir su dirección en un registro del core para utilizarlo como puntero.
- Por ejemplo, en un NXP4337 (Cortex-M4), los diferentes periféricos se despliegan a partir de la dirección de memoria 0x40000000.
- ¿Como es posible almacenar esta dirección en un registro del core para utilizarlo como puntero base?.
- La instrucción que se requiere es del tipo:

Rango de Direccionamiento

- ARM al igual que otras arquitecturas tiene tamaño de instrucciones fijo en 32bits.
- Sin embargo muchas veces se necesita trabajar con operandos inmediatos de 32 bits. Por ejemplo una dirección de memoria.
- En particular la arquitectura ARM mapea en memoria todos los dispositivos de E/S, de modo que cuando se los quiera acceder hay que escribir su dirección en un registro del core para utilizarlo como puntero.
- Por ejemplo, en un NXP4337 (Cortex-M4), los diferentes periféricos se despliegan a partir de la dirección de memoria 0x40000000.
- ¿Como es posible almacenar esta dirección en un registro del core para utilizarlo como puntero base?.
- La instrucción que se requiere es del tipo:

Rango de Direccionamiento

- ARM al igual que otras arquitecturas tiene tamaño de instrucciones fijo en 32bits.
- Sin embargo muchas veces se necesita trabajar con operandos inmediatos de 32 bits. Por ejemplo una dirección de memoria.
- En particular la arquitectura ARM mapea en memoria todos los dispositivos de E/S, de modo que cuando se los quiera acceder hay que escribir su dirección en un registro del core para utilizarlo como puntero.
- Por ejemplo, en un NXP4337 (Cortex-M4), los diferentes periféricos se despliegan a partir de la dirección de memoria 0x40000000.
- ¿Como es posible almacenar esta dirección en un registro del core para utilizarlo como puntero base?.
- La instrucción que se requiere es del tipo:

```
1       MOV  r6,#0x40000000
```

Usando constantes de 32 bits

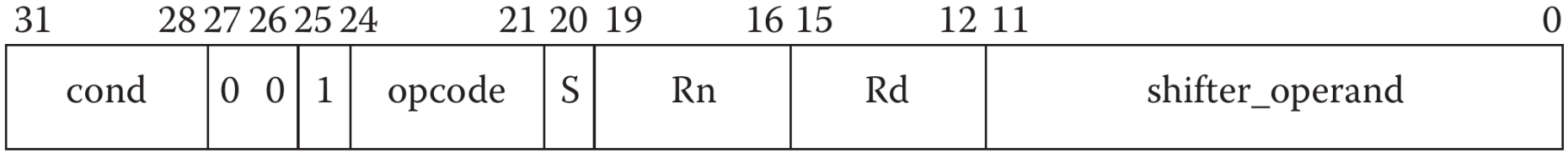
- ¿Como es el formato de la instrucción `MOV`?

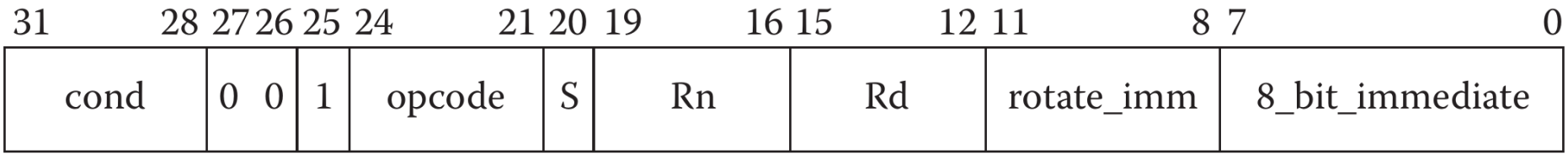
`MOV` tiene dos opciones:

- La primera recibe un offset de 12 bits relativo al *Program Counter* que se toma en Cx2 y permite referenciar direcciones entre +-4Kb y los.
- La segunda envía el campo de 12 bits en dos: en los 8 LSB un offset, y en los 4 MSB un valor que se multiplica por 2 y permite rotar a la derecha al offset en esa cantidad de bits, a través de un registro temporario interno de 32 bits.

Usando constantes de 32 bits

- ¿Como es el formato de la instrucción `MOV`?

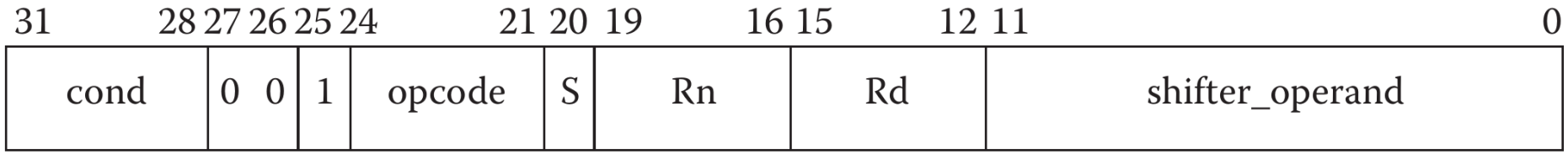


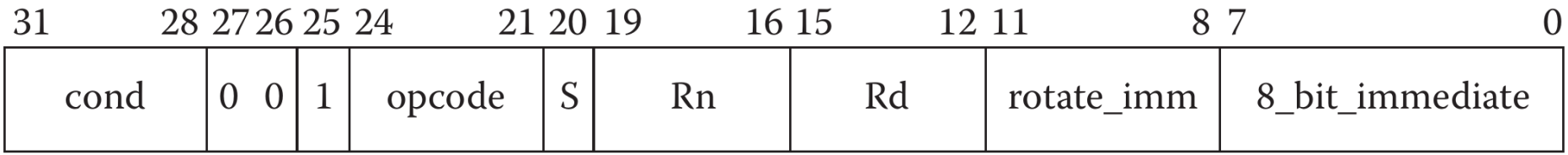


- `MOV` tiene dos opciones:
  - ✓ la primera recibe un offset de 12 bits relativo al *Program Counter* que se toma en Ca2 y permite referenciar direcciones entre ± 4Kbytes.
  - ✓ La segunda divide el campo de 12 bits en dos: en los 8 LSB un offset y en los 4 MSB un valor que es multiplicado por 2 y permite rotar a la derecha el offset en esa cantidad de bits. Luego se lo suma a un registro (generalmente el PC) para obtener una dirección de 32 bits.

Usando constantes de 32 bits

- ¿Como es el formato de la instrucción `MOV`?

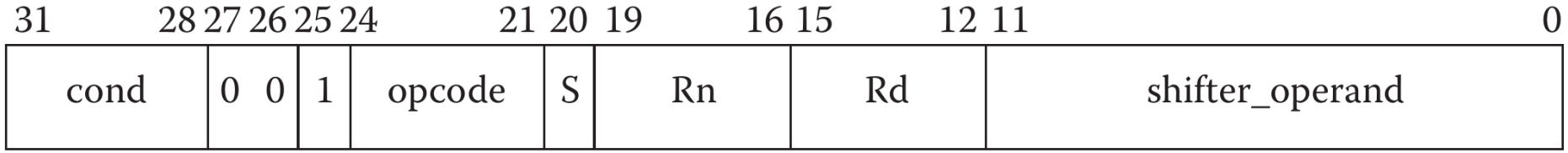


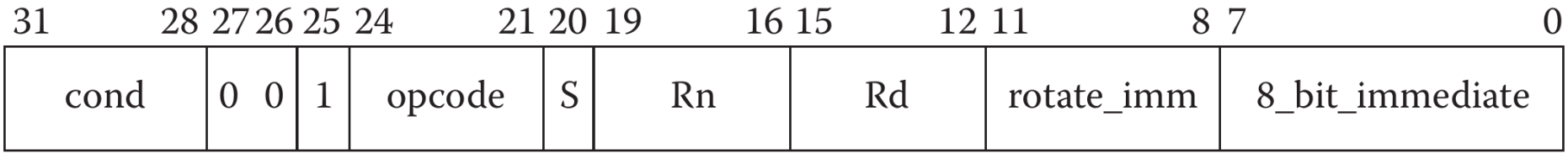


- `MOV` tiene dos opciones:
  - ✓ la primera recibe un offset de 12 bits relativo al *Program Counter* que se toma en Ca2 y permite referenciar direcciones entre $\pm$ 4Kbytes.
  - ✓ La segunda divide el campo de 12 bits en dos: en los 8 LSB un offset, y en los 4 MSB un valor que se multiplica por 2 y permite rotar a la derecha al offset en esa cantidad de bits, a través de un registro temporario interno de 32 bits.

Usando constantes de 32 bits

- ¿Como es el formato de la instrucción `MOV`?

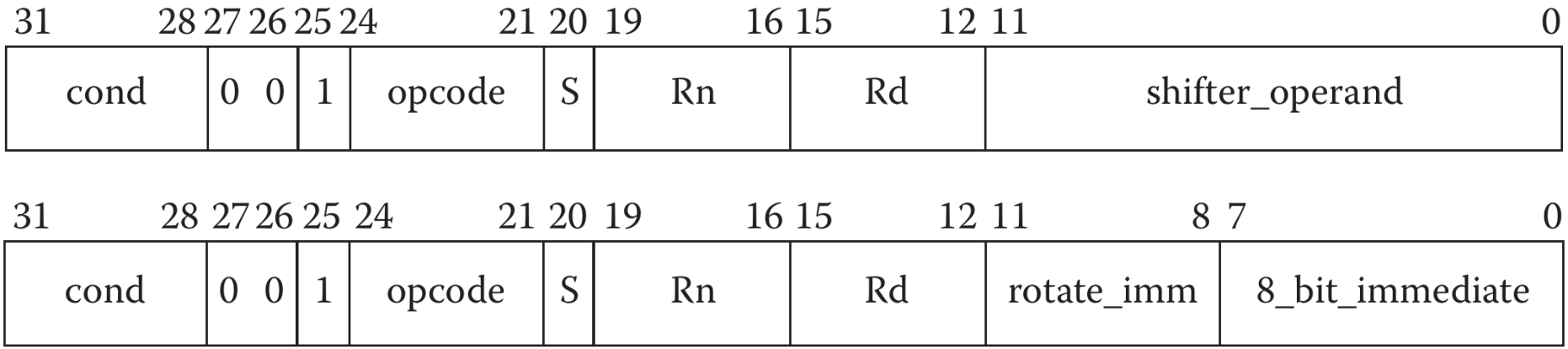




- `MOV` tiene dos opciones:
  - ✓ la primera recibe un offset de 12 bits relativo al *Program Counter* que se toma en Ca2 y permite referenciar direcciones entre $\pm$ 4Kbytes.
  - ✓ La segunda divide el campo de 12 bits en dos: en los 8 LSB un offset, y en los 4 MSB un valor que se multiplica por 2 y permite rotar a la derecha al offset en esa cantidad de bits, a través de un registro temporario interno de 32 bits.

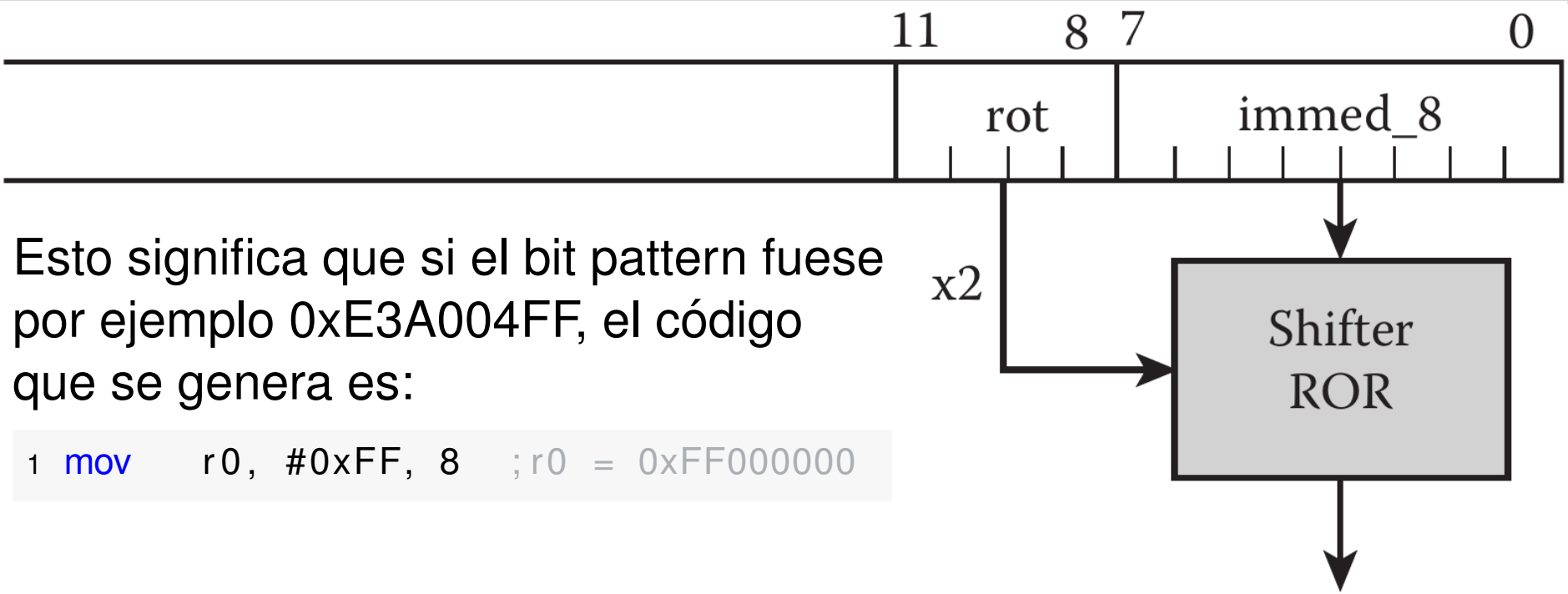
Usando constantes de 32 bits

- ¿Como es el formato de la instrucción `MOV`?



- `MOV` tiene dos opciones:
  - ✓ la primera recibe un offset de 12 bits relativo al *Program Counter* que se toma en Ca2 y permite referenciar direcciones entre $\pm$ 4Kbytes.
  - ✓ La segunda divide el campo de 12 bits en dos: en los 8 LSB un offset, y en los 4 MSB un valor que se multiplica por 2 y permite rotar a la derecha al offset en esa cantidad de bits, a través de un registro temporario interno de 32 bits.

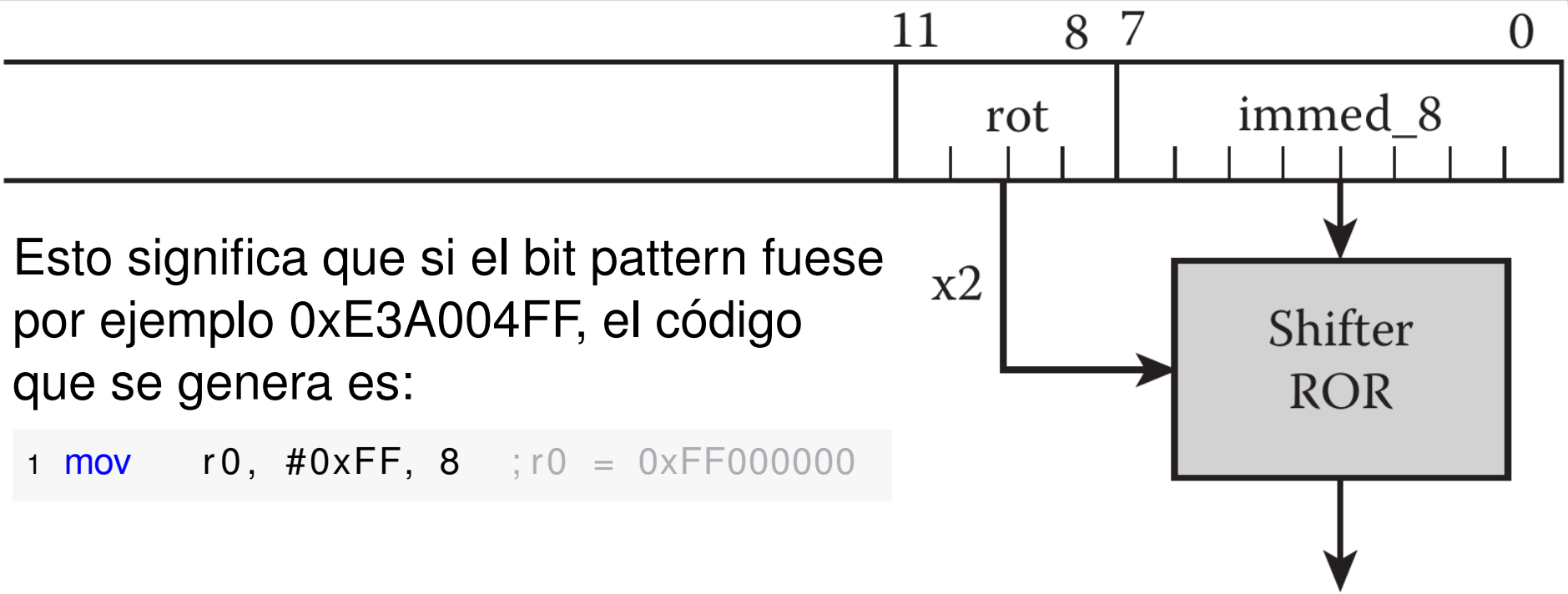
El esquema de rotación



Esto significa que si el bit pattern fuese por ejemplo 0xE3A004FF, el código que se genera es:

```
1 mov    r0, #0xFF, 8   ;r0 = 0xFF000000
```

El esquema de rotación

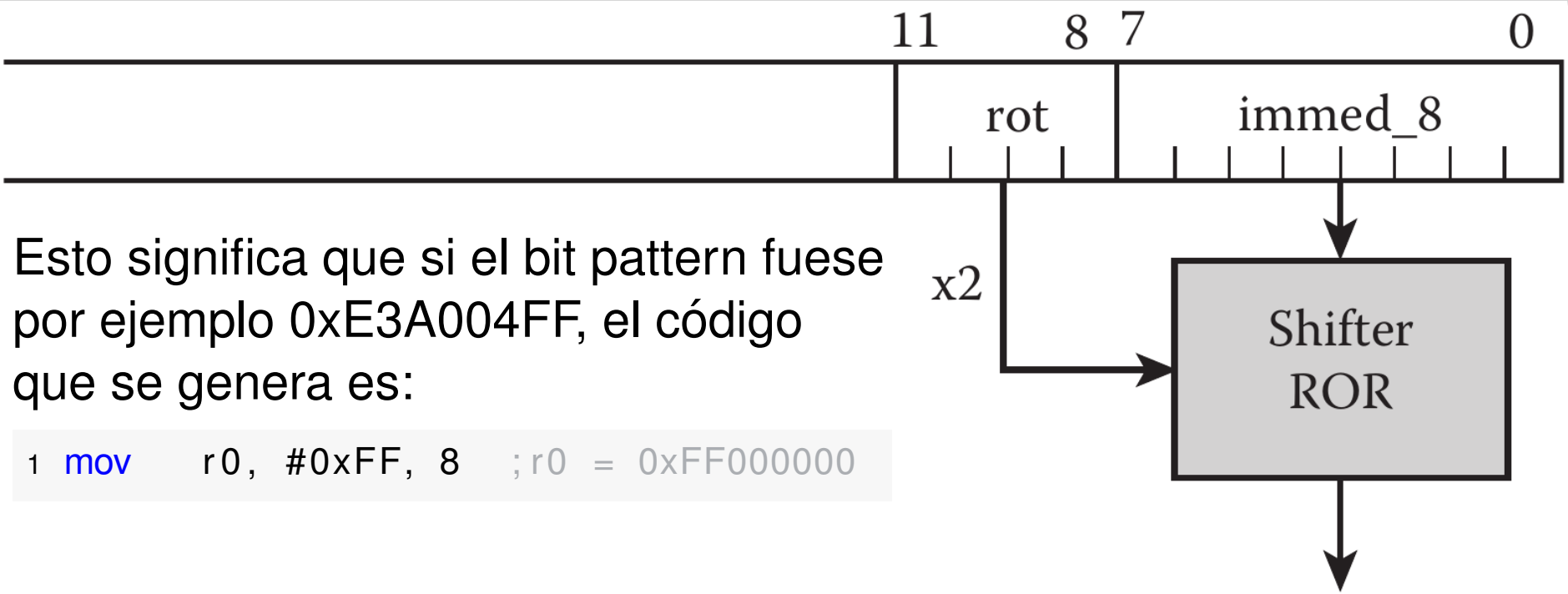


Esto significa que si el bit pattern fuese por ejemplo 0xE3A004FF, el código que se genera es:

```
1 mov    r0, #0xFF, 8   ;r0 = 0xFF000000
```

Lo que significa rotar 0xFF 8 lugares a la derecha

El esquema de rotación

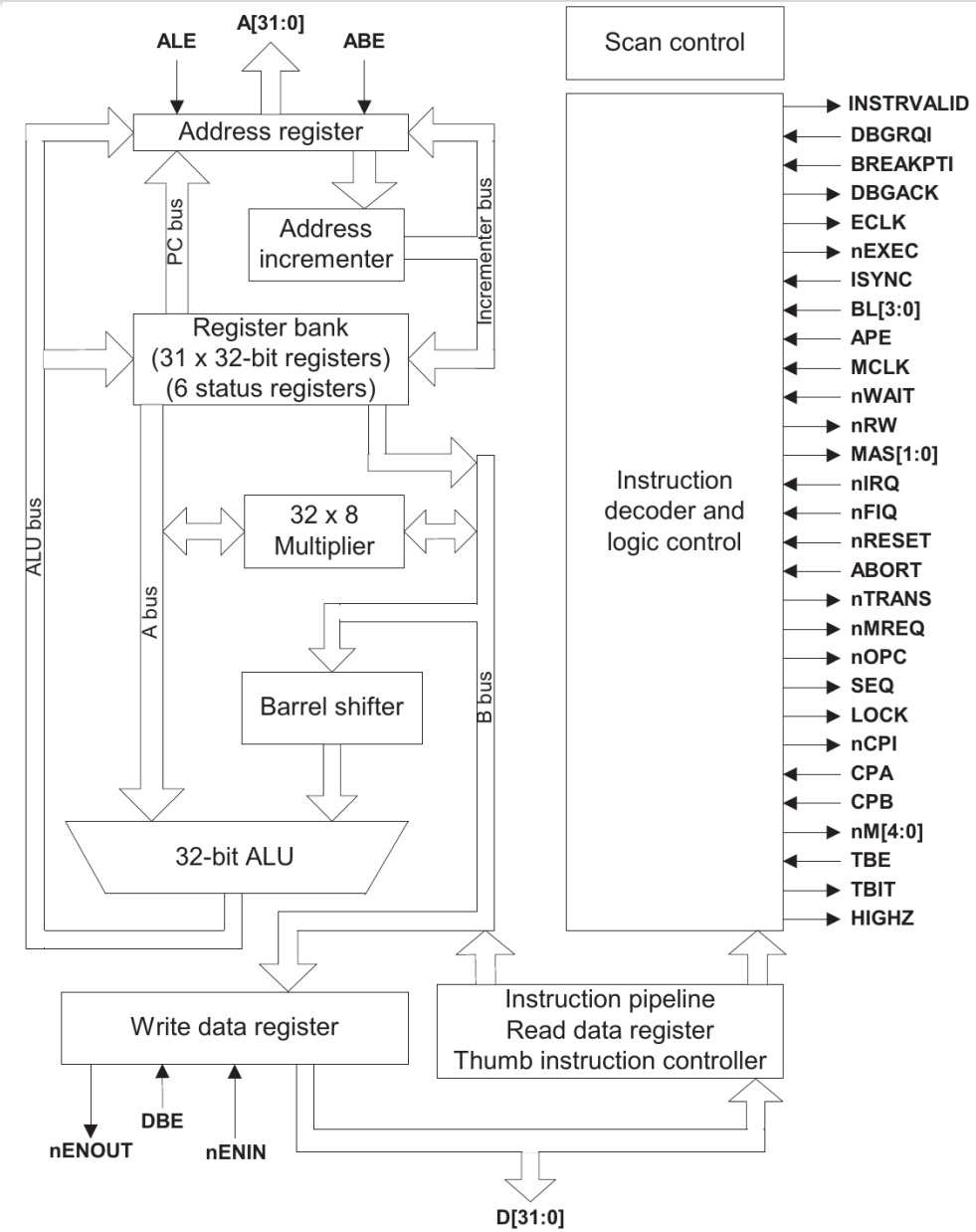


Esto significa que si el bit pattern fuese por ejemplo 0xE3A004FF, el código que se genera es:

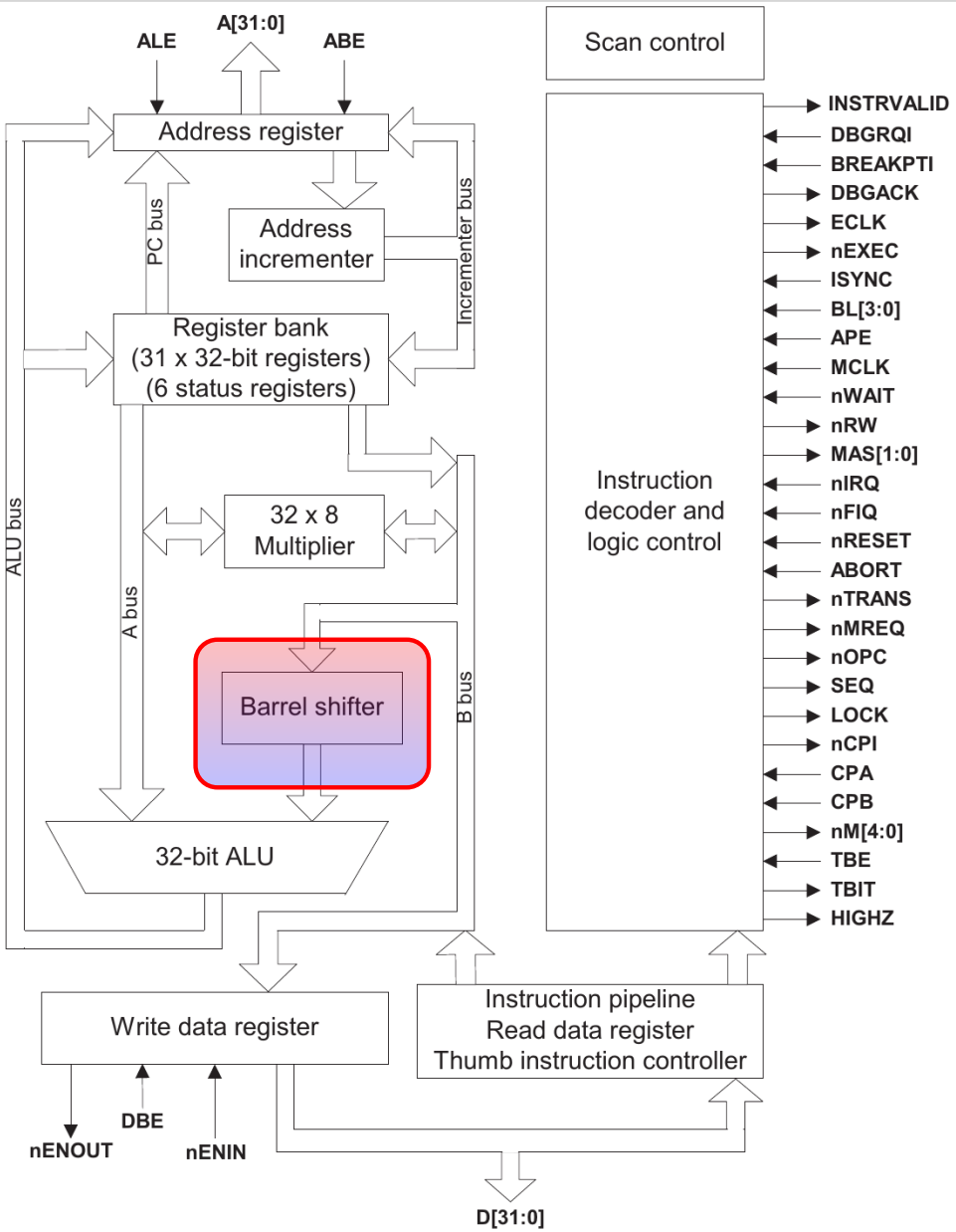
```
1 mov    r0, #0xFF, 8   ;r0 = 0xFF000000
```

Lo que significa rotar 0xFF 8 lugares a la derecha
Este funcionamiento es consecuencia de haber incluido en el datapath un barrel shifter en el camino de los operandos de la ALU

¿Recuerdan?


ALE
A[31:0]
ABE
Scan control
Address register
PC bus
Address incrementer
Incrementer bus
Register bank
(31 x 32-bit registers)
(6 status registers)
ALU bus
32 x 8
Multiplier
A bus
B bus
Barrel shifter
32-bit ALU
Instruction decoder and logic control
INSTRVALID
DBGRQI
BREAKPTI
DBGACK
ECLK
nEXEC
ISYNC
BL[3:0]
APE
MCLK
nWAIT
nRW
MAS[1:0]
nIRQ
nFIQ
nRESET
ABORT
nTRANS
nMREQ
nOPC
SEQ
LOCK
nCPI
CPA
CPB
nM[4:0]
TBE
TBIT
HIGHZ
Write data register
Instruction pipeline
Read data register
Thumb instruction controller
nENOUT
DBE
nENIN
D[31:0]

¿Recuerdan?


ALE
A[31:0]
ABE
Scan control
Address register
PC bus
Address incrementer
Incrementer bus
Register bank
(31 x 32-bit registers)
(6 status registers)
ALU bus
A bus
32 x 8
Multiplier
Barrel shifter
B bus
32-bit ALU
Instruction decoder and logic control
INSTRVALID
DBGRQI
BREAKPTI
DBGACK
ECLK
nEXEC
ISYNC
BL[3:0]
APE
MCLK
nWAIT
nRW
MAS[1:0]
nIRQ
nFIQ
nRESET
ABORT
nTRANS
nMREQ
nOPC
SEQ
LOCK
nCPI
CPA
CPB
nM[4:0]
TBE
TBIT
HIGHZ
Write data register
Instruction pipeline
Read data register
Thumb instruction controller
DBE
nENOUT
nENIN
D[31:0]

Estudios previos

- La idea es generar *Clases de números* en lugar del rango de valores posibles con 32 bits: 0 a $2^{32} - 1$.
- Como cada vez que no alcanza el recurso de hardware, la solución surge de observar como funciona el software.
- "Mediciones"(Análisis de código de uso práctico):
  - El 60 % de los valores constantes (operandos inmediatos que utilizan los programas de uso práctico) está en el rango $[-10, 10]$
  - El 90 % está en el rango $[-511, 511]$
- De modo que este modelo parece ser suficiente para resolver una parte significativa del problema, aunque nunca el 100 % de los casos.

Estudios previos

- La idea es generar *Clases de números* en lugar del rango de valores posibles con 32 bits: 0 a $2^{32} - 1$.
- Como cada vez que no alcanza el recurso de hardware, la solución surge de observar como funciona el software.
- "Mediciones"(Análisis de código de uso práctico):
  - El 60 % de los valores constantes (operandos inmediatos que utilizan los programas de uso práctico) está en el rango $[0, 2^{16})$
  - El 90 % está en el rango $[0, 2^{24})$
- De modo que este modelo parece ser suficiente para resolver una parte significativa del problema, aunque nunca el 100 % de los casos.

Estudios previos

- La idea es generar *Clases de números* en lugar del rango de valores posibles con 32 bits: 0 a $2^{32} - 1$.
- Como cada vez que no alcanza el recurso de hardware, la solución surge de observar como funciona el software.
- "Mediciones"(Análisis de código de uso práctico):
  - El 50 % de los valores constantes (operandos inmediatos que utilizan los programas de uso práctico) está en el rango [-15,15]
  - El 90 % está en el rango [-511, 511]
- De modo que este modelo parece ser suficiente para resolver una parte significativa del problema, aunque nunca el 100 % de los casos.

Estudios previos

- La idea es generar *Clases de números* en lugar del rango de valores posibles con 32 bits: 0 a $2^{32} - 1$.
- Como cada vez que no alcanza el recurso de hardware, la solución surge de observar como funciona el software.
- "Mediciones"(Análisis de código de uso práctico):
  - El 50 % de los valores constantes (operandos inmediatos que utilizan los programas de uso práctico) está en el rango [-15,15]
  - El 90 % está en el rango [-511, 511].
- De modo que este modelo parece ser suficiente para resolver una parte significativa del problema, aunque nunca el 100 % de los casos.

Estudios previos

- La idea es generar *Clases de números* en lugar del rango de valores posibles con 32 bits: 0 a $2^{32} - 1$.
- Como cada vez que no alcanza el recurso de hardware, la solución surge de observar como funciona el software.
- "Mediciones"(Análisis de código de uso práctico):
  - El 50 % de los valores constantes (operandos inmediatos que utilizan los programas de uso práctico) está en el rango [-15,15]
  - El 90 % está en el rango [-511, 511].
- De modo que este modelo parece ser suficiente para resolver una parte significativa del problema, aunque nunca el 100 % de los casos.

Estudios previos

- La idea es generar *Clases de números* en lugar del rango de valores posibles con 32 bits: 0 a $2^{32} - 1$.
- Como cada vez que no alcanza el recurso de hardware, la solución surge de observar como funciona el software.
- "Mediciones"(Análisis de código de uso práctico):
  - El 50 % de los valores constantes (operandos inmediatos que utilizan los programas de uso práctico) está en el rango [-15,15]
  - El 90 % está en el rango [-511, 511].
- De modo que este modelo parece ser suficiente para resolver una parte significativa del problema, aunque nunca el 100 % de los casos.

Estudios previos

- La idea es generar *Clases de números* en lugar del rango de valores posibles con 32 bits: 0 a $2^{32} - 1$.
- Como cada vez que no alcanza el recurso de hardware, la solución surge de observar como funciona el software.
- “Mediciones”(Análisis de código de uso práctico):
  - El 50 % de los valores constantes (operandos inmediatos que utilizan los programas de uso práctico) está en el rango [-15,15]
  - El 90 % está en el rango [-511, 511].
- De modo que este modelo parece ser suficiente para resolver una parte significativa del problema, aunque nunca el 100 % de los casos.

Trabajando con Clases de Números

- Clases de números: Son conjuntos de números con características similares. Por ejemplo, máscaras de 32 bits.
- Una forma de generar otro tipo de máscaras la proporciona la instrucción **MVN**.
- Es una variante de **MOV**, que opera de forma idéntica, solo que luego de rotar el número le efectúa la operación NOT (o Ca1).
- Ejemplos de clases de máscaras generadas mediante rotación a derecha de un offset de un byte, o con **MVN**.

Trabajando con Clases de Números

- Clases de números: Son conjuntos de números con características similares. Por ejemplo, máscaras de 32 bits.
- Una forma de generar otro tipo de máscaras la proporciona la instrucción **MVN**.
- Es una variante de **MOV**, que opera de forma idéntica, solo que luego de rotar el número le efectúa la operación NOT (o Ca1).
- Ejemplos de clases de máscaras generadas mediante rotación a derecha de un offset de un byte, o con **MVN**.

Trabajando con Clases de Números

- Clases de números: Son conjuntos de números con características similares. Por ejemplo, máscaras de 32 bits.
- Una forma de generar otro tipo de máscaras la proporciona la instrucción **MVN**.
- Es una variante de **MOV**, que opera de forma idéntica, solo que luego de rotar el número le efectúa la operación NOT (o Ca1).
- Ejemplos de clases de máscaras generadas mediante rotación a derecha de un offset de un byte, o con **MVN**.

Trabajando con Clases de Números

- Clases de números: Son conjuntos de números con características similares. Por ejemplo, máscaras de 32 bits.
- Una forma de generar otro tipo de máscaras la proporciona la instrucción **MVN**.
- Es una variante de **MOV**, que opera de forma idéntica, solo que luego de rotar el número le efectúa la operación NOT (o Ca1).
- Ejemplos de clases de máscaras generadas mediante rotación a derecha de un offset de un byte, o con **MVN**.

Trabajando con Clases de Números

- Clases de números: Son conjuntos de números con características similares. Por ejemplo, máscaras de 32 bits.
- Una forma de generar otro tipo de máscaras la proporciona la instrucción **`MVN`**.
- Es una variante de **`MOV`**, que opera de forma idéntica, solo que luego de rotar el número le efectúa la operación NOT (o Ca1).
- Ejemplos de clases de máscaras generadas mediante rotación a derecha de un offset de un byte, o con **`MVN`**.

Trabajando con Clases de Números

- Clases de números: Son conjuntos de números con características similares. Por ejemplo, máscaras de 32 bits.
- Una forma de generar otro tipo de máscaras la proporciona la instrucción **MVN**.
- Es una variante de **MOV**, que opera de forma idéntica, solo que luego de rotar el número le efectúa la operación NOT (o Ca1).
- Ejemplos de clases de máscaras generadas mediante rotación a derecha de un offset de un byte, o con **MVN**.

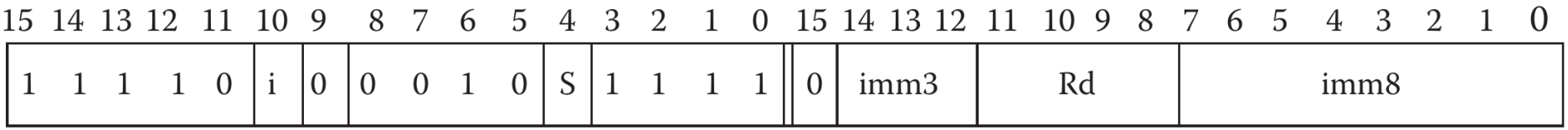
```
1   mov r6, #0xFF,24    /* OPCODE = E3A06CFF Pone r6 = 0x0000FF00 */
2   mov r6, #0x01,30    /* OPCODE = E3A06F01 Pone r6 = 0x00000004 */
3   mov r6, #0x01,26    /* OPCODE = E3A06D01 Pone r6 = 0x00000040 */
4   mvn r8, #0x00       /* OPCODE = E3E08000 Pone r8 = 0xFFFFFFFF */
5   mvn r8, #0x10       /* OPCODE = E3E08010 Pone r8 = 0xFFFFFFEF */
```

Formato de la instrucción MOV en Thumb-2

- Las instrucciones Thumb-2 (en las que se basa fuertemente Cortex-M) introducen métodos que permiten generar máscaras algo mas flexibles, *rotando a izquierda* un valor de 8 bits dentro de una word en cualquier valor n / $0 \leqslant n \leqslant 32$:
- Con ello podemos [imm12] = 0:imm3:imm8] o bien [imm12] = hword1[10] , hword2[14:12,7:0]
  - Si el dato está en el rango 0-255 el valor constante es imm12.
  - constante de la forma 0x00XY00XY (si imm12[11:8]= 0x01)
  - constante de la forma 0xXY00XY00 (si imm12[11:8]= 0x02)
  - constante de la forma 0xXYXYXYXY (si imm12[11:8]= 0x03)
- Así un Cortex-M4 puede trabajar con códigos de MOV de este tipo:

Formato de la instrucción `MOV` en Thumb-2

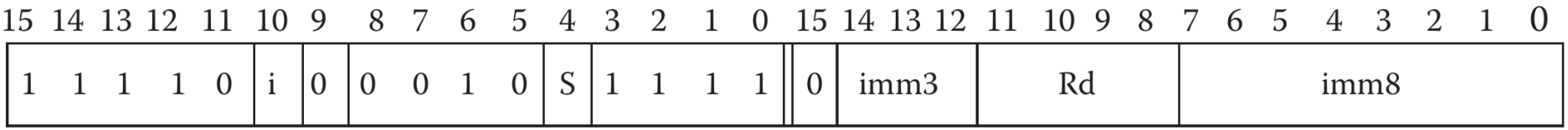
- Las instrucciones Thumb-2 (en las que se basa fuertemente Cortex-M) introducen métodos que permiten generar máscaras algo mas flexibles, *rotando a izquierda* un valor de 8 bits dentro de una word en cualquier valor n / $0 \leqslant n \leqslant 32$:



- Considerando a: `[imm12]=[i:imm3:imm8]` o bien `[imm12]` = hword1[10] , hword2[14:12,7:0]
  - Si el dato está en el rango 0-255 el valor constante es `imm12`.
  - constante de la forma 0x00XY00XY (si imm12[11:8]= 0b0001)
  - constante de la forma 0xXY00XY00 (si imm12[11:8]= 0b0010)
  - constante de la forma 0xXYXYXYXY (si imm12[11:8]= 0b0011)
- Así un Cortex-M4 puede trabajar con códigos de `MOV` de este tipo:

Formato de la instrucción `MOV` en Thumb-2

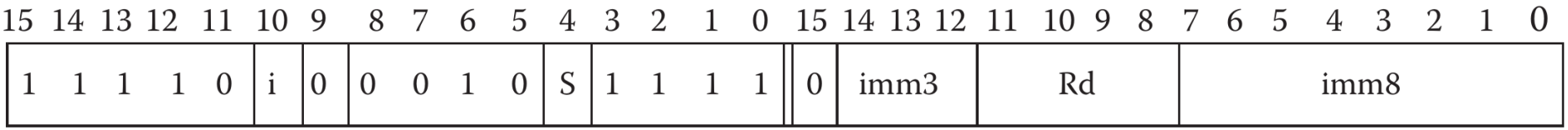
- Las instrucciones Thumb-2 (en las que se basa fuertemente Cortex-M) introducen métodos que permiten generar máscaras algo mas flexibles, *rotando a izquierda* un valor de 8 bits dentro de una word en cualquier valor n / $0 \leqslant n \leqslant 32$:



- Considerando a: `[imm12]=[i:imm3:imm8]` o bien `[imm12]` = hword1[10] , hword2[14:12,7:0]
  - Si el dato está en el rango 0-255 el valor constante es `imm12`.
  - constante de la forma 0x00XY00XY (si imm12[11:8]= 0b0001)
  - constante de la forma 0xXY00XY00 (si imm12[11:8]= 0b0010)
  - constante de la forma 0xXYXYXYXY (si imm12[11:8]= 0b0011)
- Así un Cortex-M4 puede trabajar con códigos de `MOV` de este tipo:

Formato de la instrucción `MOV` en Thumb-2

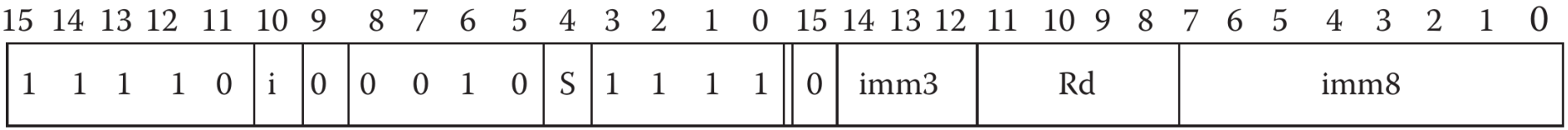
- Las instrucciones Thumb-2 (en las que se basa fuertemente Cortex-M) introducen métodos que permiten generar máscaras algo mas flexibles, *rotando a izquierda* un valor de 8 bits dentro de una word en cualquier valor n / $0 \leqslant n \leqslant 32$:



- Considerando a: `[imm12]=[i:imm3:imm8]` o bien `[imm12]` = hword1[10] , hword2[14:12,7:0]
  - Si el dato está en el rango 0-255 el valor constante es `imm12`.
  - constante de la forma 0x00XY00XY (si imm12[11:8]= 0b0001)
  - constante de la forma 0xXY00XY00 (si imm12[11:8]= 0b0010)
  - constante de la forma 0xXYXYXYXY (si imm12[11:8]= 0b0011)
- Así un Cortex-M4 puede trabajar con códigos de `MOV` de este tipo:

Formato de la instrucción `MOV` en Thumb-2

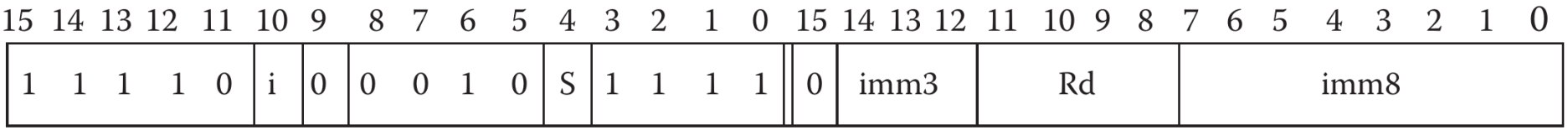
- Las instrucciones Thumb-2 (en las que se basa fuertemente Cortex-M) introducen métodos que permiten generar máscaras algo mas flexibles, *rotando a izquierda* un valor de 8 bits dentro de una word en cualquier valor n / $0 \leqslant n \leqslant 32$:



- Considerando a: `[imm12]=[i:imm3:imm8]` o bien `[imm12]` = hword1[10] , hword2[14:12,7:0]
  - Si el dato está en el rango 0-255 el valor constante es `imm12`.
  - constante de la forma 0x00XY00XY (si imm12[11:8]= 0b0001)
  - constante de la forma 0xXY00XY00 (si imm12[11:8]= 0b0010)
  - constante de la forma 0xXYXYXYXY (si imm12[11:8]= 0b0011)
- Así un Cortex-M4 puede trabajar con códigos de `MOV` de este tipo:

Formato de la instrucción `MOV` en Thumb-2

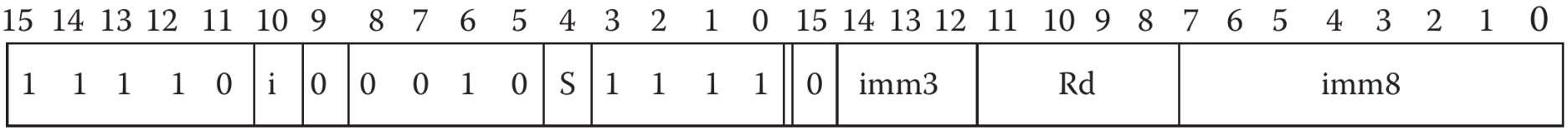
- Las instrucciones Thumb-2 (en las que se basa fuertemente Cortex-M) introducen métodos que permiten generar máscaras algo mas flexibles, *rotando a izquierda* un valor de 8 bits dentro de una word en cualquier valor n / $0 \leqslant n \leqslant 32$:



- Considerando a: `[imm12]=[i:imm3:imm8]` o bien `[imm12]` = hword1[10] , hword2[14:12,7:0]
  - Si el dato está en el rango 0-255 el valor constante es `imm12`.
  - constante de la forma 0x00XY00XY (si imm12[11:8]= 0b0001)
  - constante de la forma 0xXY00XY00 (si imm12[11:8]= 0b0010)
  - constante de la forma 0xXYXYXYXY (si imm12[11:8]= 0b0011)
- Así un Cortex-M4 puede trabajar con códigos de `MOV` de este tipo:

Formato de la instrucción `MOV` en Thumb-2

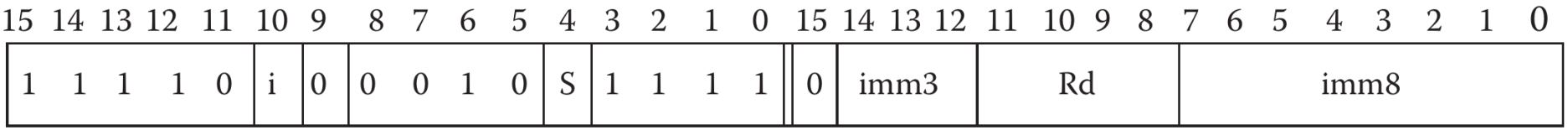
- Las instrucciones Thumb-2 (en las que se basa fuertemente Cortex-M) introducen métodos que permiten generar máscaras algo mas flexibles, *rotando a izquierda* un valor de 8 bits dentro de una word en cualquier valor n / $0 \leqslant n \leqslant 32$:



- Considerando a: `[imm12]=[i:imm3:imm8]` o bien `[imm12]` = hword1[10] , hword2[14:12,7:0]
  - Si el dato está en el rango 0-255 el valor constante es `imm12`.
  - constante de la forma 0x00XY00XY (si imm12[11:8]= 0b0001)
  - constante de la forma 0xXY00XY00 (si imm12[11:8]= 0b0010)
  - constante de la forma 0xXYXYXYXY (si imm12[11:8]= 0b0011)
- Así un Cortex-M4 puede trabajar con códigos de `MOV` de este tipo:

Formato de la instrucción MOV en Thumb-2

- Las instrucciones Thumb-2 (en las que se basa fuertemente Cortex-M) introducen métodos que permiten generar máscaras algo mas flexibles, *rotando a izquierda* un valor de 8 bits dentro de una word en cualquier valor n / $0 \leqslant n \leqslant 32$:



- Considerando a: `[imm12]=[i:imm3:imm8]` o bien `[imm12]` = hword1[10] , hword2[14:12,7:0]
  - Si el dato está en el rango 0-255 el valor constante es `imm12`.
  - constante de la forma 0x00XY00XY (si imm12[11:8]= 0b0001)
  - constante de la forma 0xXY00XY00 (si imm12[11:8]= 0b0010)
  - constante de la forma 0xXYXYXYXY (si imm12[11:8]= 0b0011)
- Así un Cortex-M4 puede trabajar con códigos de `MOV` de este tipo:

```
1        MOV r3, #0x55555555
```

Ejemplo Thumb-2

- Siempre el mejor ejemplo práctico es el acceso a periféricos.
- Tomemos el NXP LPC-4537. El Controlador Ethernet tiene sus registros mapeados a partir de la dirección 0x40010000.
- Los controladores Ethernet manejan timestamps.
- A partir del offset 0x700 hasta 0x728, tienen un set de registros de 4 bytes, consecutivos destinados a este fin.
- Asumiendo que en `r0` almacenamos la dirección base (en breve explicaremos como lograrlo), podemos utilizar un registro cualquiera para movernos por las direcciones de los timestamps registers.

Ejemplo Thumb-2

- Siempre el mejor ejemplo práctico es el acceso a periféricos.
- Tomemos el NXP LPC-4537. El Controlador Ethernet tiene sus registros mapeados a partir de la dirección 0x40010000.
- Los controladores Ethernet manejan timestamps.
- A partir del offset 0x700 hasta 0x728, tienen un set de registros de 4 bytes, consecutivos destinados a este fin.
- Asumiendo que en `r0` almacenamos la dirección base (en breve explicaremos como lograrlo), podemos utilizar un registro cualquiera para movernos por las direcciones de los timestamps registers.

Ejemplo Thumb-2

- Siempre el mejor ejemplo práctico es el acceso a periféricos.
- Tomemos el NXP LPC-4537. El Controlador Ethernet tiene sus registros mapeados a partir de la dirección 0x40010000.
- Los controladores Ethernet manejan timestamps.
- A partir del offset 0x700 hasta 0x728, tienen un set de registros de 4 bytes, consecutivos destinados a este fin.
- Asumiendo que en `r0` almacenamos la dirección base (en breve explicaremos como lograrlo), podemos utilizar un registro cualquiera para movernos por las direcciones de los timestamps registers.

Ejemplo Thumb-2

- Siempre el mejor ejemplo práctico es el acceso a periféricos.
- Tomemos el NXP LPC-4537. El Controlador Ethernet tiene sus registros mapeados a partir de la dirección 0x40010000.
- Los controladores Ethernet manejan timestamps.
- A partir del offset 0x700 hasta 0x728, tienen un set de registros de 4 bytes, consecutivos destinados a este fin.
- Asumiendo que en `r0` almacenamos la dirección base (en breve explicaremos como lograrlo), podemos utilizar un registro cualquiera para movernos por las direcciones de los timestamps registers.

Ejemplo Thumb-2

- Siempre el mejor ejemplo práctico es el acceso a periféricos.
- Tomemos el NXP LPC-4537. El Controlador Ethernet tiene sus registros mapeados a partir de la dirección 0x40010000.
- Los controladores Ethernet manejan timestamps.
- A partir del offset 0x700 hasta 0x728, tienen un set de registros de 4 bytes, consecutivos destinados a este fin.
- Asumiendo que en `r0` almacenamos la dirección base (en breve explicaremos como lograrlo), podemos utilizar un registro cualquiera para movernos por las direcciones de los timestamps registers.

Ejemplo Thumb-2

- Siempre el mejor ejemplo práctico es el acceso a periféricos.
- Tomemos el NXP LPC-4537. El Controlador Ethernet tiene sus registros mapeados a partir de la dirección 0x40010000.
- Los controladores Ethernet manejan timestamps.
- A partir del offset 0x700 hasta 0x728, tienen un set de registros de 4 bytes, consecutivos destinados a este fin.
- Asumiendo que en `r0` almacenamos la dirección base (en breve explicaremos como lograrlo), podemos utilizar un registro cualquiera para movernos por las direcciones de los timestamps registers.

Ejemplo Thumb-2

- Siempre el mejor ejemplo práctico es el acceso a periféricos.
- Tomemos el NXP LPC-4537. El Controlador Ethernet tiene sus registros mapeados a partir de la dirección 0x40010000.
- Los controladores Ethernet manejan timestamps.
- A partir del offset 0x700 hasta 0x728, tienen un set de registros de 4 bytes, consecutivos destinados a este fin.
- Asumiendo que en **`r0`** almacenamos la dirección base (en breve explicaremos como lograrlo), podemos utilizar un registro cualquiera para movernos por las direcciones de los timestamps registers.

```
1       ldr   r2, [r0,r3]
```

Ejemplo Thumb-2

- En `r3`, escribimos en forma inmediata el offset relativo a la base del registro específico que deseamos acceder.
- Para inicializar `r3` con el offest de inicio de éstos registros respecto de la `Dirección Base` del controlador, una alternativa puede ser:
- Este tipo de acceso es útil solo si se necesita acceder a los registros de manera consecutiva, ya que solo demandaría un incremento o decremento en 4 de `r3`, por cada acceso.

Ejemplo Thumb-2

- En **`r3`**, escribimos en forma inmediata el offset relativo a la base del registro específico que deseamos acceder.
- Para inicializar **`r3`** con el offest de inicio de éstos registros respecto de la **`Dirección Base`** del controlador, una alternativa puede ser:
- Este tipo de acceso es útil solo si se necesita acceder a los registros de manera consecutiva, ya que solo demandaría un incremento o decremento en 4 de `r3`, por cada acceso.

Ejemplo Thumb-2

- En **r3**, escribimos en forma inmediata el offset relativo a la base del registro específico que deseamos acceder.
- Para inicializar **r3** con el offest de inicio de éstos registros respecto de la **Dirección Base** del controlador, una alternativa puede ser:

```
1       mov   r3, #0x700
```

- Este tipo de acceso es útil solo si se necesita acceder a los registros de manera consecutiva, ya que solo demandaría un incremento o decremento en 4 de **r3**, por cada acceso.

Ejemplo Thumb-2

- En **r3**, escribimos en forma inmediata el offset relativo a la base del registro específico que deseamos acceder.
- Para inicializar **r3** con el offest de inicio de éstos registros respecto de la **Dirección Base** del controlador, una alternativa puede ser:

```
1       mov   r3, #0x700
```

- Este tipo de acceso es útil solo si se necesita acceder a los registros de manera consecutiva, ya que solo demandaría un incremento o decremento en 4 de **r3**, por cada acceso.

Thumb-2 es diferente

- En la instrucción `mov` anterior, el ensamblador generará para el modo Thumb-2 una instrucción codificada como 0xF44F63E0, de acuerdo con el formato de instrucción T2.
- El Modo Thumb-2 El valor de 8 bits se rota hacia la izquierda.
- La cantidad de bits a desplazar se almacena en $imm12[11:7]$ o $i:imm3:imm8[7]$.
- Total 5 bits, es decir desplaza $2^n - 1 bits, donde 0 \leqslant n \leqslant 31$.
- Tomado el dato inmediato de 32 bits, la cantidad de bits a desplazar a la izquierda es igual a la cantidad necesaria para que su bit mas significativo en '1' vaya a parar al bit $imm8[7]$ (los '0's a la izquierda se descartan).
- En nuestro caso son los bits A10 y A7 de la instrucción T2. Implica rotar a izquierda 29 bits.

Thumb-2 es diferente

- En la instrucción **`mov`** anterior, el ensamblador generará para el modo Thumb-2 una instrucción codificada como 0xF44F63E0, de acuerdo con el formato de instrucción T2.
- El Modo Thumb-2 El valor de 8 bits se rota hacia la izquierda.
- La cantidad de bits a desplazar se almacena en $imm12[11:7]$ o $i:imm3:imm8[7]$.
- Total 5 bits, es decir desplaza $2^n - 1 bits, donde 0 \leqslant n \leqslant 31$.
- Tomado el dato inmediato de 32 bits, la cantidad de bits a desplazar a la izquierda es igual a la cantidad necesaria para que su bit mas significativo en '1' vaya a parar al bit $imm8[7]$ (los '0's a la izquierda se descartan).
- En nuestro caso son los bits A10 y A7 de la instrucción T2. Implica rotar a izquierda 29 bits.

Thumb-2 es diferente

- En la instrucción **`mov`** anterior, el ensamblador generará para el modo Thumb-2 una instrucción codificada como 0xF44F63E0, de acuerdo con el formato de instrucción T2.
- El Modo Thumb-2 El valor de 8 bits se rota hacia la izquierda.
- La cantidad de bits a desplazar se almacena en $imm12[11:7]$ o $i:imm3:imm8[7]$.
- Total 5 bits, es decir desplaza $2^n - 1 bits, donde 0 \leqslant n \leqslant 31$.
- Tomado el dato inmediato de 32 bits, la cantidad de bits a desplazar a la izquierda es igual a la cantidad necesaria para que su bit mas significativo en '1' vaya a parar al bit $imm8[7]$ (los '0's a la izquierda se descartan).
- En nuestro caso son los bits A10 y A7 de la instrucción T2. Implica rotar a izquierda 29 bits.

Thumb-2 es diferente

- En la instrucción **`mov`** anterior, el ensamblador generará para el modo Thumb-2 una instrucción codificada como 0xF44F63E0, de acuerdo con el formato de instrucción T2.
- El Modo Thumb-2 El valor de 8 bits se rota hacia la izquierda.
- La cantidad de bits a desplazar se almacena en $imm12[11:7]$ o $i:imm3:imm8[7]$.
- Total 5 bits, es decir desplaza $2^n - 1 bits, donde 0 \leqslant n \leqslant 31$.
- Tomado el dato inmediato de 32 bits, la cantidad de bits a desplazar a la izquierda es igual a la cantidad necesaria para que su bit mas significativo en '1' vaya a parar al bit $imm8[7]$ (los '0's a la izquierda se descartan).
- En nuestro caso son los bits A10 y A7 de la instrucción T2. Implica rotar a izquierda 29 bits.

Thumb-2 es diferente

- En la instrucción **`mov`** anterior, el ensamblador generará para el modo Thumb-2 una instrucción codificada como 0xF44F63E0, de acuerdo con el formato de instrucción T2.
- El Modo Thumb-2 El valor de 8 bits se rota hacia la izquierda.
- La cantidad de bits a desplazar se almacena en $imm12[11:7]$ o $i:imm3:imm8[7]$.
- Total 5 bits, es decir desplaza $2^n - 1 bits, donde 0 \leqslant n \leqslant 31$.
- Tomado el dato inmediato de 32 bits, la cantidad de bits a desplazar a la izquierda es igual a la cantidad necesaria para que su bit mas significativo en '1' vaya a parar al bit $imm8[7]$ (los '0's a la izquierda se descartan).
- En nuestro caso son los bits A10 y A7 de la instrucción T2. Implica rotar a izquierda 29 bits.

Thumb-2 es diferente

- En la instrucción **mov** anterior, el ensamblador generará para el modo Thumb-2 una instrucción codificada como 0xF44F63E0, de acuerdo con el formato de instrucción T2.
- El Modo Thumb-2 El valor de 8 bits se rota hacia la izquierda.
- La cantidad de bits a desplazar se almacena en $imm12[11:7]$ o $i:imm3:imm8[7]$.
- Total 5 bits, es decir desplaza $2^n - 1 bits, donde 0 \leqslant n \leqslant 31$.
- Tomado el dato inmediato de 32 bits, la cantidad de bits a desplazar a la izquierda es igual a la cantidad necesaria para que su bit mas significativo en '1' vaya a parar al bit $imm8[7]$ (los '0's a la izquierda se descartan).
- En nuestro caso son los bits A10 y A7 de la instrucción T2. Implica rotar a izquierda 29 bits.

Thumb-2 es diferente

- En la instrucción **mov** anterior, el ensamblador generará para el modo Thumb-2 una instrucción codificada como 0xF44F63E0, de acuerdo con el formato de instrucción T2.
- El Modo Thumb-2 El valor de 8 bits se rota hacia la izquierda.
- La cantidad de bits a desplazar se almacena en $imm12[11:7]$ o $i:imm3:imm8[7]$.
- Total 5 bits, es decir desplaza $2^n - 1 bits, donde 0 \leqslant n \leqslant 31$.
- Tomado el dato inmediato de 32 bits, la cantidad de bits a desplazar a la izquierda es igual a la cantidad necesaria para que su bit mas significativo en '1' vaya a parar al bit $imm8[7]$ (los '0's a la izquierda se descartan).
- En nuestro caso son los bits A10 y A7 de la instrucción T2. Implica rotar a izquierda 29 bits.

Thumb-2 es diferente

- En la instrucción **mov** anterior, el ensamblador generará para el modo Thumb-2 una instrucción codificada como 0xF44F63E0, de acuerdo con el formato de instrucción T2.
- El Modo Thumb-2 El valor de 8 bits se rota hacia la izquierda.
- La cantidad de bits a desplazar se almacena en $imm12[11:7]$ o $i:imm3:imm8[7]$.
- Total 5 bits, es decir desplaza $2^n - 1 bits, donde 0 \leqslant n \leqslant 31$.
- Tomado el dato inmediato de 32 bits, la cantidad de bits a desplazar a la izquierda es igual a la cantidad necesaria para que su bit mas significativo en '1' vaya a parar al bit $imm8[7]$ (los '0's a la izquierda se descartan).
- En nuestro caso son los bits A10 y A7 de la instrucción T2. Implica rotar a izquierda 29 bits.

A12	A11	A10	A09	A08	A07	A06	A05	A04	A03	A02	A01	A00
⩽0	0	**1**	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	**1**	1	1	0	0	0	0	0⩽

Una mejor solución: Literal pools

Literal

En Ciencias de la Computación se denomina *literal* a cualquier valor arbitrario asignado en un programa a una variable o a un registro

Literal Pool

En Ciencias de la Computación, y mas específicamente en diseño de compiladores y ensambladores, se denomina *literal pool* a una lookup table utilizada para almacenar *literals*, del mismo tipo o de diferentes tipos.

Una mejor solución: Literal pools

Literal

En Ciencias de la Computación se denomina *literal* a cualquier valor arbitrario asignado en un programa a una variable o a un registro

Ejemplo:

```
1    .word      BASE_ADRR 0x40010000 /* Direccion Base
         Controlador Ethernet LPC4337*/
```

Literal Pool

En Ciencias de la Computación, y mas específicamente en diseño de compiladores y ensambladores, se denomina *literal pool* a una lookup table utilizada para almacenar *literals*, del mismo tipo o de diferentes tipos.

Una mejor solución: Literal pools

Literal

En Ciencias de la Computación se denomina *literal* a cualquier valor arbitrario asignado en un programa a una variable o a un registro

Ejemplo:

```
1    .word      BASE_ADRR 0x40010000 /* Direccion Base
         Controlador Ethernet LPC4337*/
```

Literal Pool

En Ciencias de la Computación, y mas específicamente en diseño de compiladores y ensambladores, se denomina *literal pool* a una lookup table utilizada para almacenar *literals*, del mismo tipo o de diferentes tipos.

¿A que viene esto?

Operandos inmediatos

Varias arquitecturas de microprocesadores (ARM, IBM System/360 entre otras), ya sea por utilizar tamaño fijo de instrucción o tener el set de instrucciones optimizado para saltos cortos, encuentra limitaciones cuando tienen que expresar en forma inmediata un operando cuyo tamaño en bits supera el de campo reservado a tal fin en su código de operación.

¿A que viene esto?

Operandos inmediatos

Varias arquitecturas de microprocesadores (ARM, IBM System/360 entre otras), ya sea por utilizar tamaño fijo de instrucción o tener el set de instrucciones optimizado para saltos cortos, encuentra limitaciones cuando tienen que expresar en forma inmediata un operando cuyo tamaño en bits supera el de campo reservado a tal fin en su código de operación.

¿A que viene esto?

- Esta limitación afecta a instrucciones de:
  - Carga inmediata de constantes en un registro.
  - Saltos a direcciones cuyo *target* necesita para ser expresado una cantidad de bits mayor que la disponible dentro del campo de operación de 32 bits.
- En general hay dos formas de especificar la dirección de un salto en forma inmediata:
  - Con el valor completo de la dirección (32 bits en un solo campo)
  - En forma relativa al Program Counter.
- En general en el OPCODE de 32 bits quedan solo 12 para una dirección o un valor inmediato. Es decir para valores por encima de 4095 (0xFFF) **el procesador no puede resolverlo**.

¿A que viene esto?

- Esta limitación afecta a instrucciones de:
  - Carga inmediata de constantes en un registro.
  - Saltos a direcciones cuyo target necesite para ser expresado una cantidad de bits mayor que la disponible dentro del código de operación de 32 bits.
- En general hay dos formas de especificar la dirección de un salto en forma inmediata:
  - Con el valor completo de la dirección (32 bits en un solo paso).
  - En forma relativa al Program Counter.
- En general en el OPCODE de 32 bits quedan solo 12 para una dirección o un valor inmediato. Es decir para valores por encima de 4095 (0xFFF) **el procesador no puede resolverlo**.

¿A que viene esto?

- Esta limitación afecta a instrucciones de:
  - Carga inmediata de constantes en un registro.
  - Saltos a direcciones cuyo target necesite para ser expresado una cantidad de bits mayor que la disponible dentro del código de operación de 32 bits.
- En general hay dos formas de especificar la dirección de un salto en forma inmediata:
  - Con el valor completo de la dirección (32 bits en un solo paso).
  - En forma relativa al Program Counter.
- En general en el OPCODE de 32 bits quedan solo 12 para una dirección o un valor inmediato. Es decir para valores por encima de 4095 (0xFFF) **el procesador no puede resolverlo**.

¿A que viene esto?

- Esta limitación afecta a instrucciones de:
  - Carga inmediata de constantes en un registro.
  - Saltos a direcciones cuyo target necesite para ser expresado una cantidad de bits mayor que la disponible dentro del código de operación de 32 bits.
- En general hay dos formas de especificar la dirección de un salto en forma inmediata:
  - Con el valor completo de la dirección (32 bits en nuestro caso)
  - En forma relativa al Program Counter.
- En general en el OPCODE de 32 bits quedan solo 12 para una dirección o un valor inmediato. Es decir para valores por encima de 4095 (0xFFF) **el procesador no puede resolverlo**.

¿A que viene esto?

- Esta limitación afecta a instrucciones de:
  - Carga inmediata de constantes en un registro.
  - Saltos a direcciones cuyo target necesite para ser expresado una cantidad de bits mayor que la disponible dentro del código de operación de 32 bits.
- En general hay dos formas de especificar la dirección de un salto en forma inmediata:
  - Con el valor completo de la dirección (32 bits en nuestro caso)
  - En forma relativa al Program Counter.
- En general en el OPCODE de 32 bits quedan solo 12 para una dirección o un valor inmediato. Es decir para valores por encima de 4095 (0xFFF) **el procesador no puede resolverlo**.

¿A que viene esto?

- Esta limitación afecta a instrucciones de:
  - Carga inmediata de constantes en un registro.
  - Saltos a direcciones cuyo target necesite para ser expresado una cantidad de bits mayor que la disponible dentro del código de operación de 32 bits.
- En general hay dos formas de especificar la dirección de un salto en forma inmediata:
  - Con el valor completo de la dirección (32 bits en nuestro caso)
  - En forma relativa al Program Counter.
- En general en el OPCODE de 32 bits quedan solo 12 para una dirección o un valor inmediato. Es decir para valores por encima de 4095 (0xFFF) **el procesador no puede resolverlo**.

¿A que viene esto?

- Esta limitación afecta a instrucciones de:
  - Carga inmediata de constantes en un registro.
  - Saltos a direcciones cuyo target necesite para ser expresado una cantidad de bits mayor que la disponible dentro del código de operación de 32 bits.
- En general hay dos formas de especificar la dirección de un salto en forma inmediata:
  - Con el valor completo de la dirección (32 bits en nuestro caso)
  - En forma relativa al Program Counter.
- En general en el OPCODE de 32 bits quedan solo 12 para una dirección o un valor inmediato. Es decir para valores por encima de 4095 (0xFFF) **el procesador no puede resolverlo**.

¿A que viene esto?

- Esta limitación afecta a instrucciones de:
  - Carga inmediata de constantes en un registro.
  - Saltos a direcciones cuyo target necesite para ser expresado una cantidad de bits mayor que la disponible dentro del código de operación de 32 bits.
- En general hay dos formas de especificar la dirección de un salto en forma inmediata:
  - Con el valor completo de la dirección (32 bits en nuestro caso)
  - En forma relativa al Program Counter.
- En general en el OPCODE de 32 bits quedan solo 12 para una dirección o un valor inmediato. Es decir para valores por encima de 4095 (0xFFF) **el procesador no puede resolverlo**.

Si el procesador no puede....

Aparece el compilador / ensamblador

Cuando no puede resolver una instrucción que carga un literal en un registro, o un salto a una dirección “larga”, el ensamblador (o el compilador en lenguajes de alto nivel), arma una tabla de literales ubicada en direcciones accesibles mediante las instrucciones disponibles colocando en esa tabla los valores “largos”mediante instrucciones de acceso indirecto.

Si el procesador no puede....

Aparece el compilador / ensamblador

Cuando no puede resolver una instrucción que carga un literal en un registro, o un salto a una dirección “larga”, el ensamblador (o el compilador en lenguajes de alto nivel), arma una tabla de literales ubicada en direcciones accesibles mediante las instrucciones disponibles colocando en esa tabla los valores “largos”mediante instrucciones de acceso indirecto.

Lineamientos para un literal pool

- El compilador/ensamblador arma lookup tables y las ubica intercaladas con el código cada no mas de 4 Kbytes (si su limite es 12 bits).
- En el caso de los saltos, si la dirección es suficientemente cercana la incluye en el OPCODE del branch.
- Si es “mas larga ”que el límite soportado por la arquitectura crea una entrada en la lookup table, y utiliza un salto indirecto a esa etiqueta.
- Para constantes algunas arquitecturas definen pseudo-instrucciones para el compilador. En el caso de ARM:

Lineamientos para un literal pool

- El compilador/ensamblador arma lookup tables y las ubica intercaladas con el código cada no mas de 4 Kbytes (si su limite es 12 bits).
- En el caso de los saltos, si la dirección es suficientemente cercana la incluye en el OPCODE del branch.
- Si es “mas larga ”que el límite soportado por la arquitectura crea una entrada en la lookup table, y utiliza un salto indirecto a esa etiqueta.
- Para constantes algunas arquitecturas definen pseudo-instrucciones para el compilador. En el caso de ARM:

Lineamientos para un literal pool

- El compilador/ensamblador arma lookup tables y las ubica intercaladas con el código cada no mas de 4 Kbytes (si su limite es 12 bits).
- En el caso de los saltos, si la dirección es suficientemente cercana la incluye en el OPCODE del branch.
- Si es “mas larga ”que el límite soportado por la arquitectura crea una entrada en la lookup table, y utiliza un salto indirecto a esa etiqueta.
- Para constantes algunas arquitecturas definen pseudo-instrucciones para el compilador. En el caso de ARM:

Lineamientos para un literal pool

- El compilador/ensamblador arma lookup tables y las ubica intercaladas con el código cada no mas de 4 Kbytes (si su limite es 12 bits).
- En el caso de los saltos, si la dirección es suficientemente cercana la incluye en el OPCODE del branch.
- Si es “mas larga ”que el límite soportado por la arquitectura crea una entrada en la lookup table, y utiliza un salto indirecto a esa etiqueta.
- Para constantes algunas arquitecturas definen pseudo-instrucciones para el compilador. En el caso de ARM:

Lineamientos para un literal pool

- El compilador/ensamblador arma lookup tables y las ubica intercaladas con el código cada no mas de 4 Kbytes (si su limite es 12 bits).
- En el caso de los saltos, si la dirección es suficientemente cercana la incluye en el OPCODE del branch.
- Si es “mas larga ”que el límite soportado por la arquitectura crea una entrada en la lookup table, y utiliza un salto indirecto a esa etiqueta.
- Para constantes algunas arquitecturas definen pseudo-instrucciones para el compilador. En el caso de ARM:

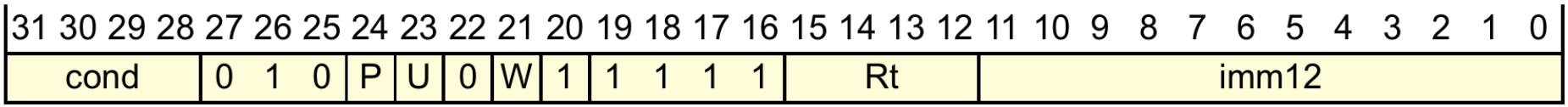
```
1    ldr   r0,= 0x40010000 /*Asi resolvemos la carga de la
         Direccion Base Controlador Ethernet LPC4337*/
```

Pseudo-Instrucción LDR rn,=#valor

Supongamos la siguiente pseudo-instrucción:

```
1    ldr   r7,= 0x40040000
```

Cuando miramos el listado vemos que el ensamblador generó este código de 32 bits para resolverla: 0xE59F706C



1 1 1 0 0 1 0 1 1 0 0 1 1 1 1 1 0 1 1 1 0 0 0 0 1 1 0 1 1 0 0

Vemos que Rt es 7 (R7) y el valor imm12 = 0x06C es lo que debe sumarse al valor actual del PC para encontrar el literal que se cargará en R7.

Accesorias del compilador / ensamblador

- El programa objeto generado ya sea con un compilador o con un ensamblador, tiene indefectiblemente un encabezado, precedente al código binario.
- Este encabezado es insumo fundamental del linker editor.
- En dicho encabezado, el ensamblador en este caso guarda el o los literal pools en la tabla de símbolos reubicables, de modo que el linker al armar la imagen del programa ejecutable los pueda terminar de ubicar combinándolos eventualmente con los literales de los restantes objetos que componen el proyecto.

Accesorias del compilador / ensamblador

- El programa objeto generado ya sea con un compilador o con un ensamblador, tiene indefectiblemente un encabezado, precedente al código binario.
- Este encabezado es insumo fundamental del linker editor.
- En dicho encabezado, el ensamblador en este caso guarda el o los literal pools en la tabla de símbolos reubicables, de modo que el linker al armar la imagen del programa ejecutable los pueda terminar de ubicar combinándolos eventualmente con los literales de los restantes objetos que componen el proyecto.

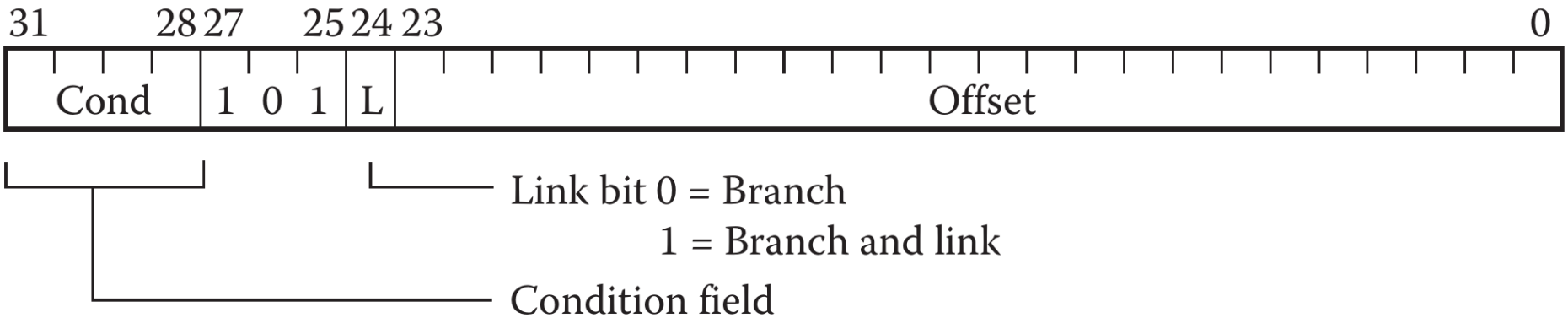
Accesorias del compilador / ensamblador

- El programa objeto generado ya sea con un compilador o con un ensamblador, tiene indefectiblemente un encabezado, precedente al código binario.
- Este encabezado es insumo fundamental del linker editor.
- En dicho encabezado, el ensamblador en este caso guarda el o los literal pools en la tabla de símbolos reubicables, de modo que el linker al armar la imagen del programa ejecutable los pueda terminar de ubicar combinándolos eventualmente con los literales de los restantes objetos que componen el proyecto.

Lo inevitable

- Son un mal necesario
- pero inevitable
- En algún momento necesitamos discontinuar el flujo de ejecución
  - Por una interrupción
  - Por invocar a una función
  - Porque evaluada una condición hay que continuar por otra dirección
  - Por estricta necesidad

Tipos de salto



Tanto para B como BL, o BX, la cuestión es que se puede saltar alrededor de 2^{24} bytes del valor del Program Counter al momento de ejecutar el salto.
Esto significa que el PC está apuntando a la instrucción sucesora secuencial del salto.

Ej:

```
1  b   DESTINO_ADDR
2  add  R1,R2,R6      /* Intrucción sucesora secuencial */
```

Si DESTINO_ADDR $>2^{24}$, entonces se arma en un literal pool

Saltos condicionales

Mnemónico	Flag	Significado	Código
EQ	Z=1	Equal	0000
NE	Z=0	Not Equal	0001
CS/HS	C=1	$\geqslant$ No Signado	0010
CC/LO	C=0	$<$ No Signado	0011
MI	N=1	Negativo	0100
PL	N=0	Positivo o Cero	0101
VS	V=1	Overflow	0110
VC	V=0	No Overflow	0111
HI	C=1, Z=0	$>$ No Signado	1000
LS	C=0, Z=1	$\leqslant$ No Signado	1001
GE	$N \geqslant V$	$\geqslant$ Signado	1010
LT	$N \neq V$	$<$Signado	1011
GT	Z=0, N=V	$>$Signado	1100
LE	Z=1, $N \neq V$	$\leqslant$Signado	1101
AL	Siempre	Default	1110

Loops

- Trabajan con los saltos condicionales.
- CBZ y CBNZ Comparan y saltan si el registro operando es cero, o no es cero respectivamente. Útiles en loops.

Loops

- Trabajan con los saltos condicionales.
- CBZ y CBNZ Comparan y saltan si el registro operando es cero, o no es cero respectivamente. Útiles en loops.

Conclusiones

- Hay instrucciones aritméticas, lógicas, de comparación, etc
- La mejor (y única) manera de aprenderlas es experimentando
- Hay Hojas de ayuda para tener un panorama rápido del set de instrucciones y poder luego buscar en la documentación los detalles (ruta recomendada)