Descripción funcional de un microprocesador

Sistemas con Microprocesadores

Ing. Esteban Volentini (evolentini@herrera.unt.edu.ar)

http://microprocesadores.unt.edu.ar/procesadores

Cronograma

Actividad	Inicio	Descripción	Fin
Presentación	19/08	Reglamento de la Materia	✔
Tema 1	19/08	Estructura de las computadoras	✔
Tema 2	26/08	Proyecto con un microcontrolador	✔
Tema 3	30/08	Descripción funcional de microprocesador	←
Tema 4	13/09	Programación en lenguaje ensablador	
Tema 5	25/09	Descripción general de un microcontrolador	
Tema 6	27/09	Estructura general de microcontrolador	
Parcial	09/10	Primer examen parcial	
Tema 7	14/10	Sistema de Interrupciones	
Tema 8	21/10	Entradas y salidas digitales	
Tema 9	28/10	Entrada/salida con perifericos	
Tema 10	06/11	Temporizadores	
Proyectos	25/11	Seminarios de Proyectos	
Parcial	04/12	Segundo examen parcial	

Bibliografía

- ARM_Cortex_M4_User_Manual.pdf
- ARMv7M_Reference_Manual.pdf
  - Tiene los formatos de las instrucciones.
  - Tabla muy detallada
- UM10503_NXP_LPC4337.pdf – pp. 1371
  - Tabla resumen con tiempos.
- Ojo: Solo veremos las instrucciones más usadas.
- **URGENTE: Bajar User Manual y tenerlo en clase.**

ISA del ARM Cortex-M4

Modelo para el programador

- Características Generales
- Estructura Interna del CPU
- Registros Internos.
- Tiempos Significativos
- Formato de la Instrucción
- Modos de Direccionamiento y Set de Instrucciones

Características Generales

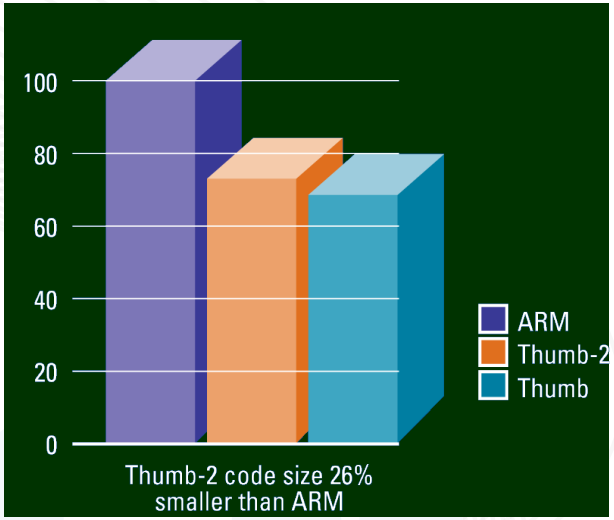
- Común a todos los MCUs con ARM-CortexM4
- ARM – Advanced (Acorn) Risc Machine.
  - Inglaterra – 1985 (diseño estudiante Stanford).
  - No vende chips. Vende el diseño del CPU.
  - Ampliamente adoptado para MCUs por distintas marcas.
  - M: significa para MCU. Otras siglas para otros usos.
  - ARM-Cortex A7 a A9 en I-phone.

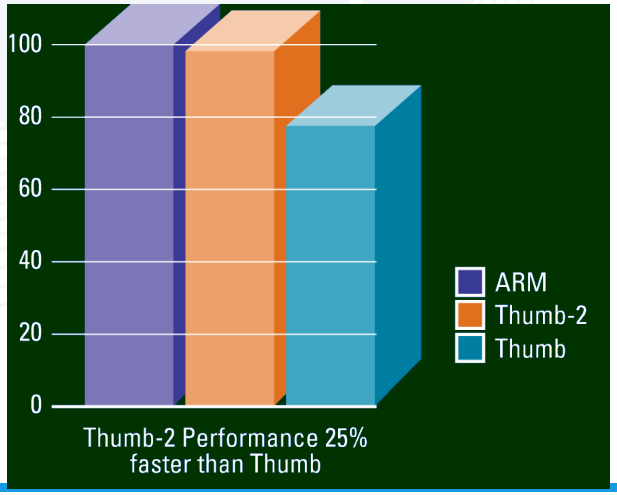
Características Generales (II)

- Tamaño de palabra de 32 bits
- Espacio de direcciones de 4 GB.
- ARM provee el diseño del núcleo, común para diversos fabricantes, se diferencian por dispositivos periféricos en MCU.
- El NXP LPC4337, frecuencia máx. 205 MHz.
- Máquina RISC.
- ISA M4 – Thumb2: subconjunto instrucciones ARM, pero de largo variable (16 o 32 bits)

Thumb2: Set de instrucciones M4

- Instrucciones de largo variable
  - Instrucciones ARM 32 bits.
  - Instrucciones Thumb fijas de 16 bits (M1, por ejemplo)
  - Thumb-2: pueden ser tanto de 16-bit o 32-bit
- Mejora en un 26% la densidad de código sobre ARM fijo de 32 bits.
- Mejora en un 25% en performance sobre Thumb
- Todas las instrucciones se expanden a ARM 32 bits en CPU

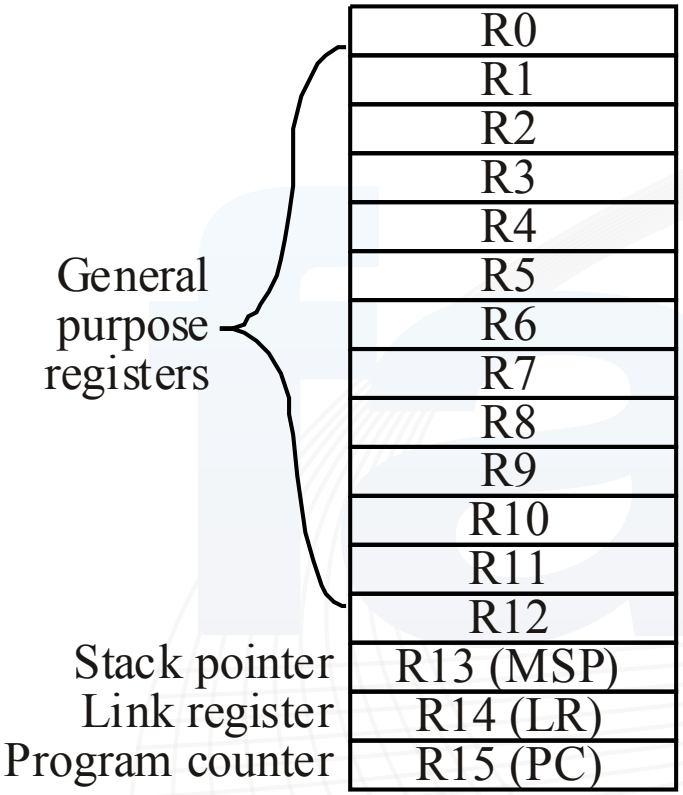




Notación

- Número Hexadecimal: 0x20 (similar a C)
- Decimal, sin prefijo. 0x20 = 32.
- Concatenación ":", se unen los dígitos del número siguiendo su orden
  - Ej: 0x20:0x32 = 0x2032
- Contenido de un lugar de Memoria:
  - (0x0000.3022) = 0x0000.0010, en el lugar de memoria cuya dirección es 0x0000.3022 está almacenado el número 0x0000.0010.

ARM ISA: Registros



- 16 registros.
- 3 de uso especial
  - Más adelante.
- Pueden contener datos o punteros a Memoria
- Ancho: 32 bits.
- R15 es el PC.
- PSR – Program Status Reg
  - N, Z, C, V

Memoria

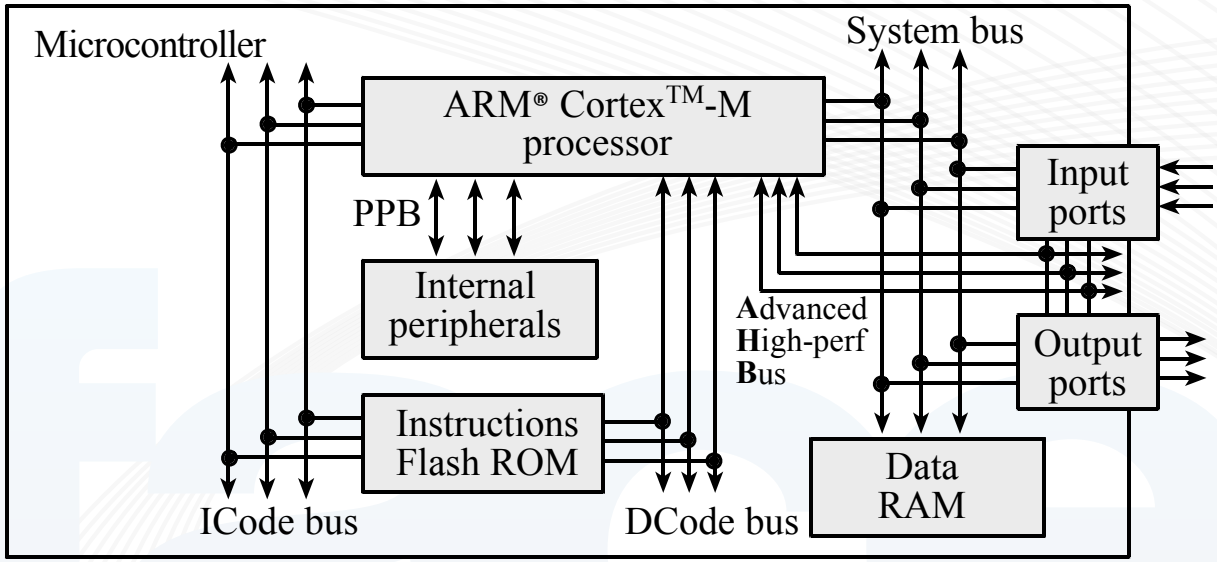
- Espacio de Memoria: 4GB
  - Máximo espacio que se puede direccionar.
- No se usa todo el espacio para M.
  - Parte se usan para dispositivos de E/S.
  - Parte de la Memoria es ROM (Flash)
  - Parte de la Memoria es RAM.
  - Parte no se usa.

Mapa de Memoria – NXP4337

32 kBytes RAM	0x1000.0000 0x1000.7FFF
40 kBytes RAM	0x1008.0000 0x1008.9FFF
512 kBytes Flash	0x1A00.0000 0x1A07.FFFF
512 kBytes Flash	0x1B00.0000 0x1B07.FFFF
Periféricos	0x4008.0000 0x400E.FFFF

- Forma en que se usa el espacio de direcciones.
- Áreas sombreadas: no ocupadas.
- Tiene 6 puertos de uso general (GPIOs).
- Espacio para Puertos y para dispositivos internos.

MPU – ARM Cortex M4



- Arquitectura Harvard
  - Buses diferentes para Instrucc. y datos
- Pipeline de 3 etapas, aprox 1 instrucc/ciclo.
- Varios buses en paralelo
  - Diferentes velocidades
  - Más rápido.

Unidades Direccionables

- Bytes.
- Medias Palabras (2 bytes)
- Palabras (4 bytes)
- Algunas zonas de memoria y de E/S se cablean de manera redundante:
  - Direcciones separadas permiten acceder a cada bit de la palabra.
  - Este bit está en la posición 0 de una dirección separada.
  - Se verá más adelante.
  - Por lo tanto se pueden direccionar bits.

Códigos de Condición (PSR)

- Cuatro banderas reflejan la última operación que las cambió
  - N, Z, V, C
  - Se usan para tomar decisiones.
  - ¡No todas las instrucciones las cambian!
- **Carry (C)**: Indica que hubo un acarreo en la última operación que lo modificó. Se usa, por ejemplo, en operaciones de múltiple precisión.

Ejemplo: Uso del Carry

- Suma de Doble Precisión mediante sumas de Precisión Simple.
  - Precisión Simple: corresponde al ancho de palabra del procesador: 32 bits o 4 bytes.
  - Doble Precisión: 64 bits.
- Algoritmo similar a la suma decimal de varias cifras.
  - Primero se suma la parte baja de ambos números
  - Si el resultado de la parte baja es muy grande entonces hay acarreo que se agrega a la suma de la parte alta del numero

Ejemplo: Uso del carry

```
   0x3248.0000.E248.0000
+  0x1010.0000.20C2.0000
-------------------------
                0xE248.0000
+               0x20C2.0000
                ----------
                0x030A.0000,C=1
   0x3248.0000
+  0x1010.0000
+               C
-------------------------
   0x4258.0001
-------------------------
   0x4258.0001.030A.0000
```

Códigos de Condición (PSR)

- **Zero (Z)**: Indica que el resultado es igual a cero (Z=1)
  - ¿Cómo se obtiene?
- **Negative (N)**: Indica que el resultado es negativo.
  - ¿Cómo se obtiene?
- **Overflow (V)**: Indica que el resultado no puede ser representado con la cantidad de bits. Se usa en operaciones de números con signo.

PSR flags: a tener en cuenta.

- Reflejan la última operación que los cambió.
- En el ejemplo anterior, si entre las dos sumas se introduce una instrucción que no cambie C, el resultado no se altera.
- Instrucciones pueden o no alterar PSR
  - Se debe indicar expresamente con un sufijo S (assembler).
  - Para no equivocarse: solo afectar el PSR en las instrucciones necesarias

Lenguaje Ensamblador ARM

- Formato en Ensamblador

```
Label   Opcode    Operandos      Comment
init:   MOV       R1, #100       // tamaño de la tabla
        BX        LR
```

- Segundo Operando: <*op2*>

```
        ADD       Rd, Rn, <op2>
```

- Comentarios
  - Deberían explicar por qué o cómo.
  - No deben *explicar qué hace la instrucción.*
  - Importantes para auto-documentación

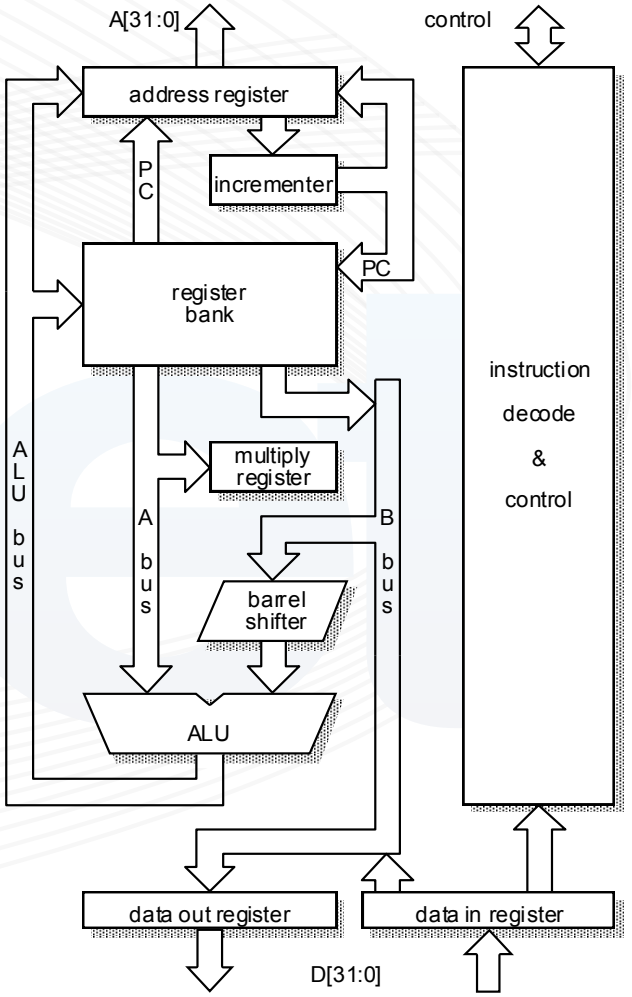
Segundo Operando “Flexible”

- Por ejemplo **`ADD Rd,Rn,<op2>`**
- Donde **`<op2>`** es una constante o un registro (**`Rm`**) que puede desplazarse hasta 31 bits a izquierda o derecha.
- Ejemplos:
  - **`ADD R0,R1,R3,LSL #3`** – registro desplazado 3 lugares

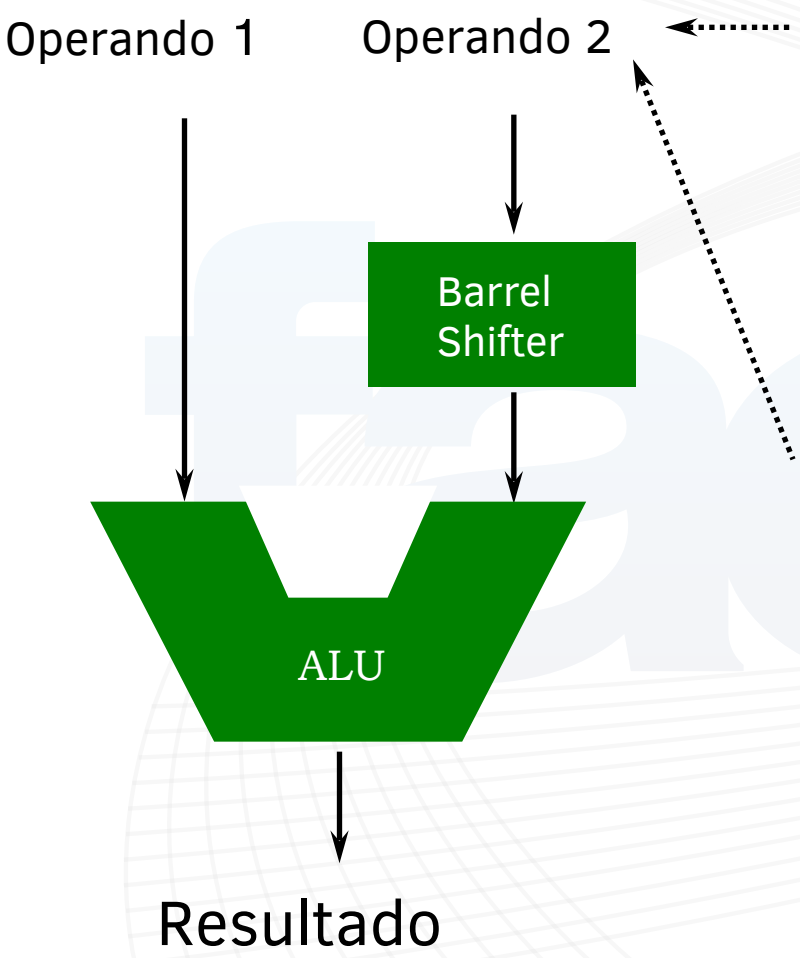
    R3 no se guarda desplazado, es solo un operando!
  - **`ADD R1,R2,cte,LSL #16`** – cte 8 bits extiende su rango.
- Todo gracias al barrel shifter en camino de datos.

El ARM por dentro

- De los dos operandos que van al ALU, uno siempre pasa por el barrel shifter
- Cuando sale la dirección de M al MAR, puede guardarse en banco de registros.
- Se incluye un multiplicador y divisor en Hw.



Operando 2 – Barrel Shifter



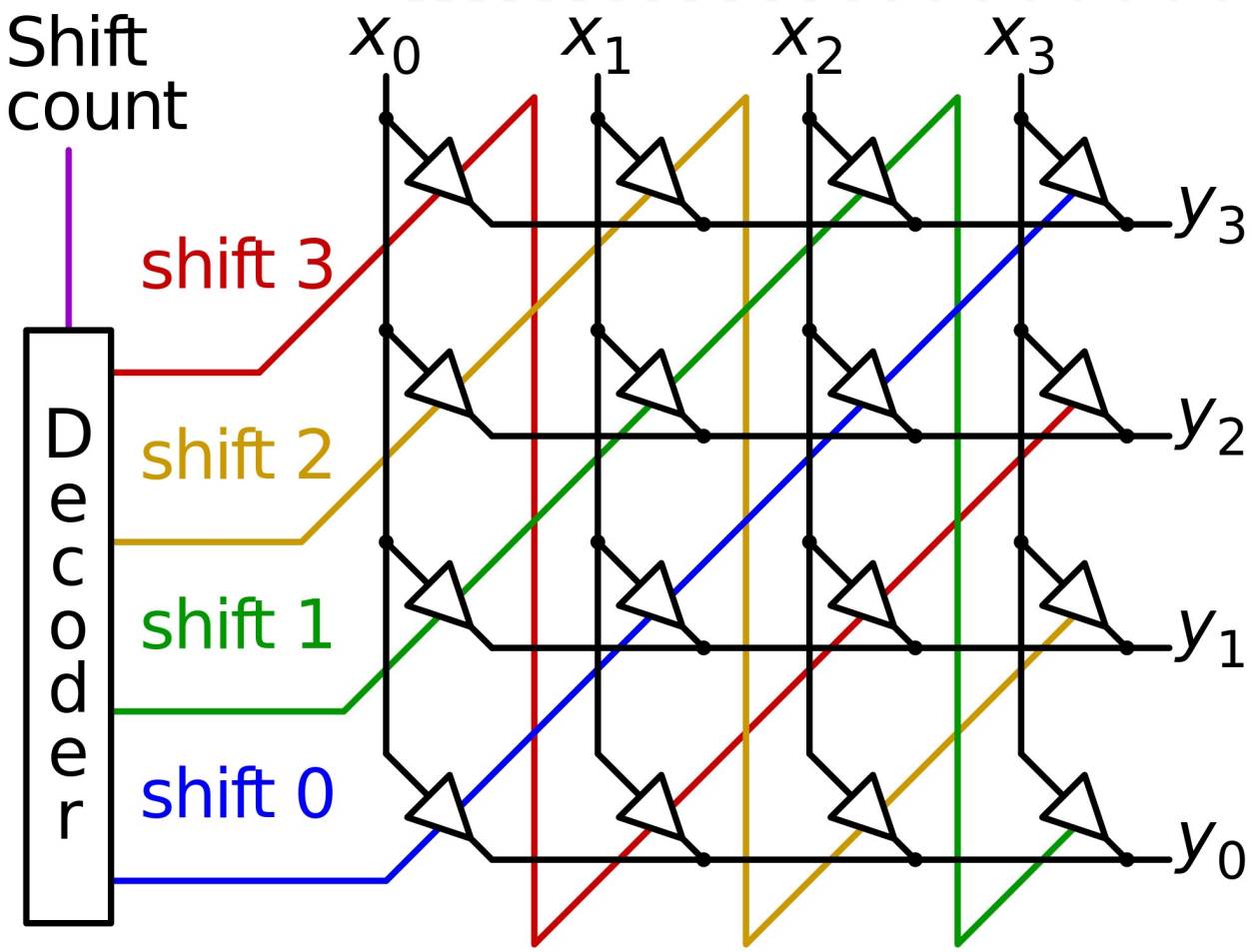
Registro, con posibilidad de shift

- El valor de shift puede ser:
  - Cte Entera sin signo de 5 bits
  - Variable en LSB de un reg.
- Se usa para multiplicar por una constante potencia de 2.

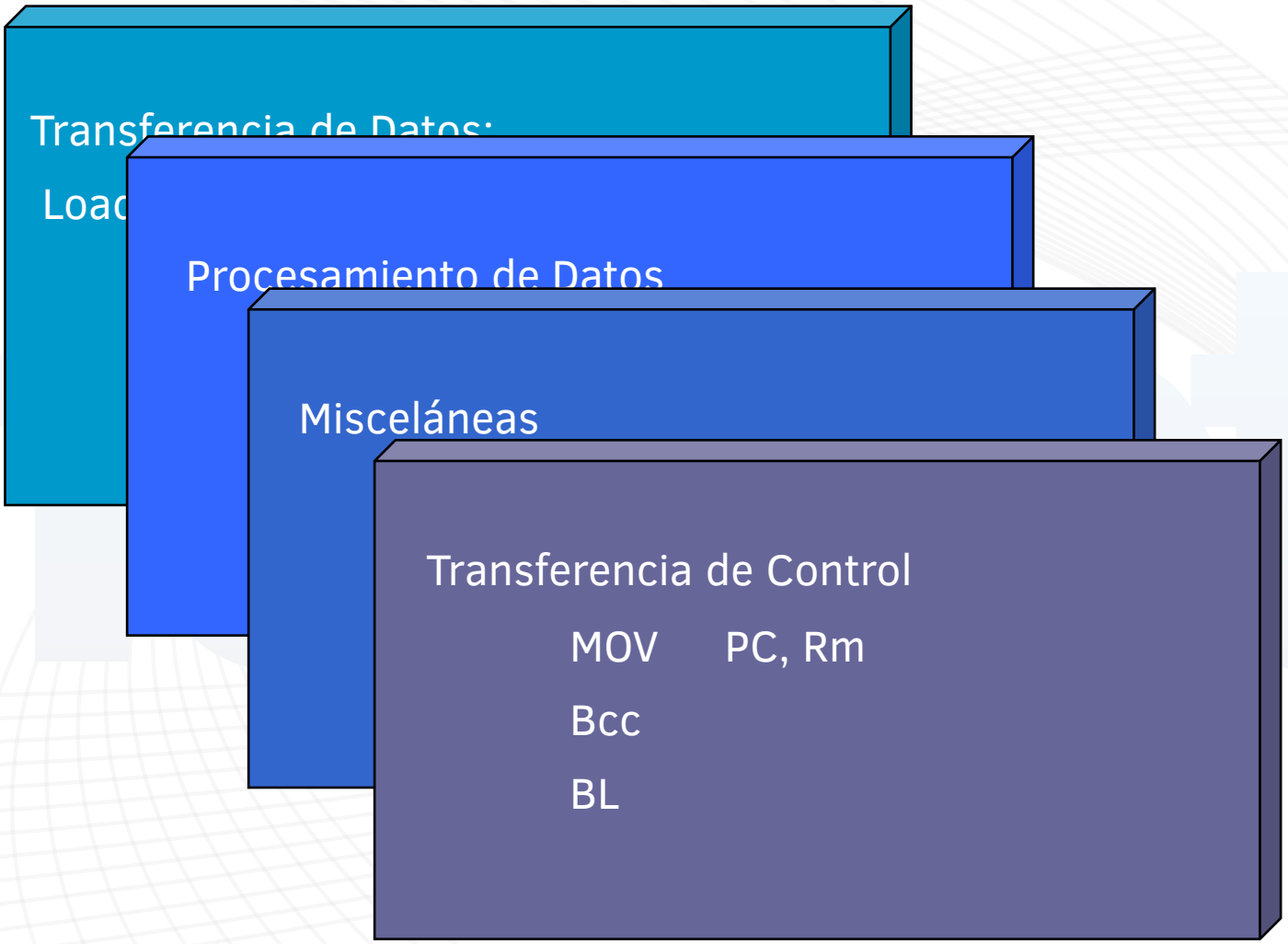
Valor Inmediato (Constante)

- Número de 8 bits, en un rango de 0-255.
  - Desplazado un número n de lugares.
- Permite generar así un rango mayor de constantes pero con precisión de 8 bits.

Ej. Barrel Shifter (rotación)



Clases de Instrucciones



Modos de Direccionamiento

- Es la forma de especificar los operandos para las instrucciones.

1. Transferencia de Datos.
   - Solo Load y Store para M.
2. Procesamiento (entre registros y constantes)
3. Saltos (fijos o variables)

- Muy simple para las dos últimas.
- Más variación en la primera.

Direccionamiento Implícito

- El operando esta implícito en la instrucción y por lo tanto no necesita ser especificado
- El operando suele ser el estado del CPU y está implícito en la instrucción
- Ejemplo: WFI (wait for interrupt).
- El procesador no hace nada hasta que no llega una interrupción y pasa a modo bajo consumo (SLEEP).

Direccionamiento Inmediato

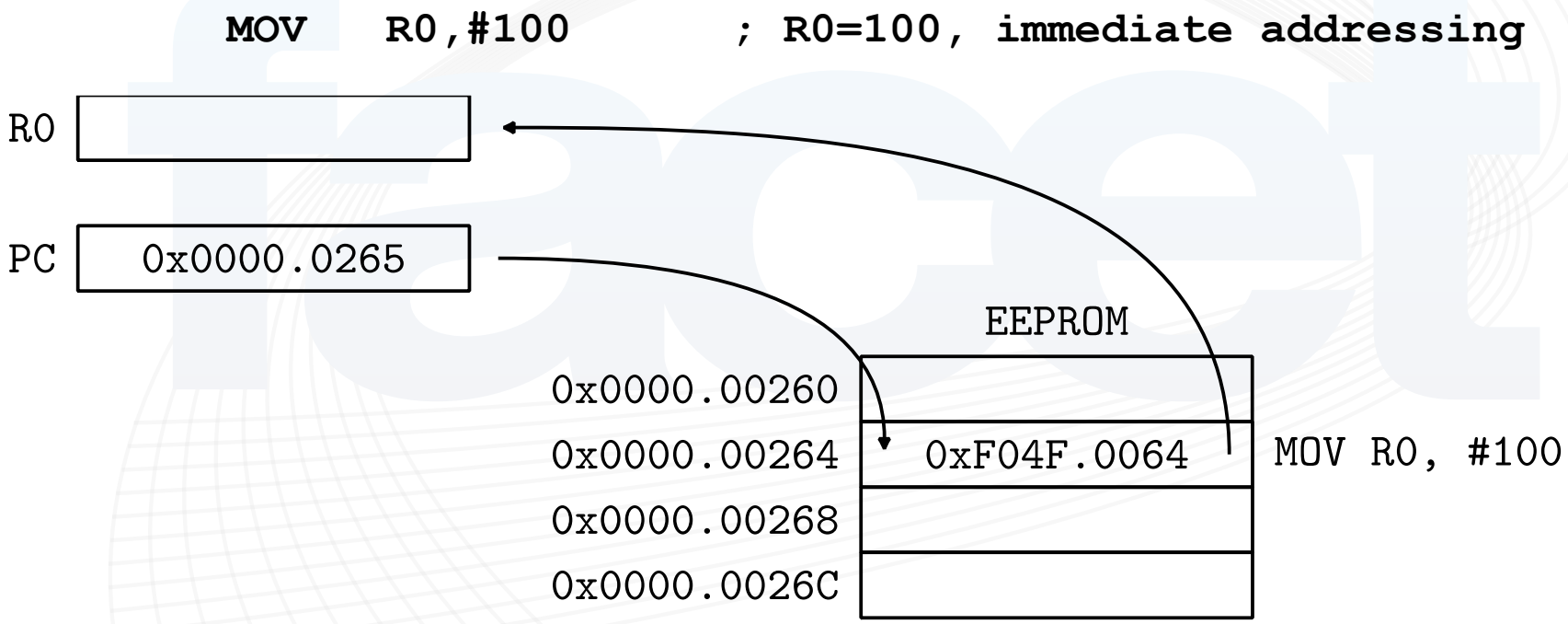
- El operando es una constante
- Se indica con el prefijo # antes de la constante

```
MOV  R0, #100   // R0 ← 100
```

- La constante puede ser multiplicada por potencias de 2 para convertirse en cualquier numero de 32 bits.
- ¿Ventajas del Direccionamiento Inmediato?

Direccionamiento Inmediato

Se muestra la ubicación de la instrucción en M y el valor del PC (siempre es impar, compatibilidad)



Mover una constante > 16 bits?

- Mover primero la parte baja (16 bits).
  `MOV R0, #100 // R0 ← 0x00000064`
- R1 toma el valor decimal 100 (no signado).
- El resto de R0 se rellena con ceros: **`0x00000064`**
- La constante es de 16 bits.
- Para constantes mayores? Move to top (MOVT)
  `MOVT R0, #0x1234 // R0 ← 0x12340064`
- Así se puede cargar con dos instrucciones una cte de 32 bits. Muchas veces basta con 16 bits ☺

Direccionamiento de Registro

- El campo operando contiene el número de un registro (0-15).
- Ejemplo

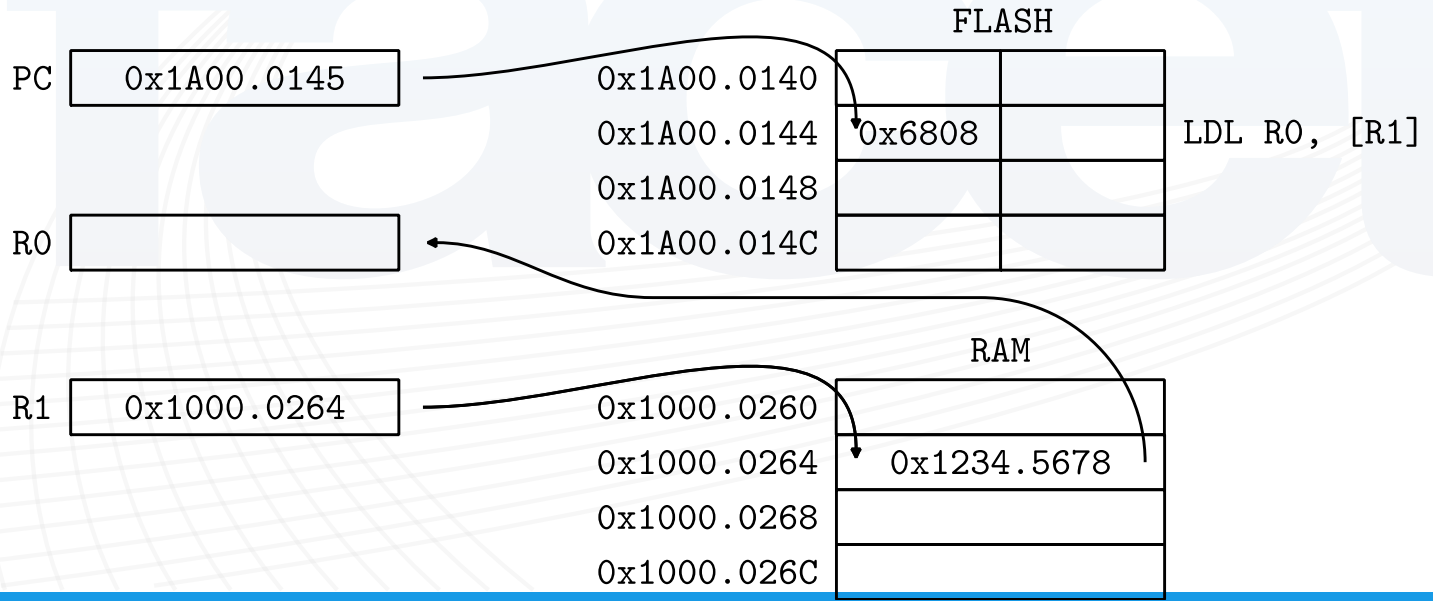
```
ADD R2,R3,#100 // R2 ← R3 +100
```

- Si el contenido previo de r3=300
- Entonces ahora r2=400
- Se toman valores decimales por simplicidad.

Direcc. Registro Indirecto

Se usa para acceder a datos en M usando un registro como puntero, es decir que contiene la dirección de memoria del dato

LDR R0,[R1] // R0 ← M[R1]

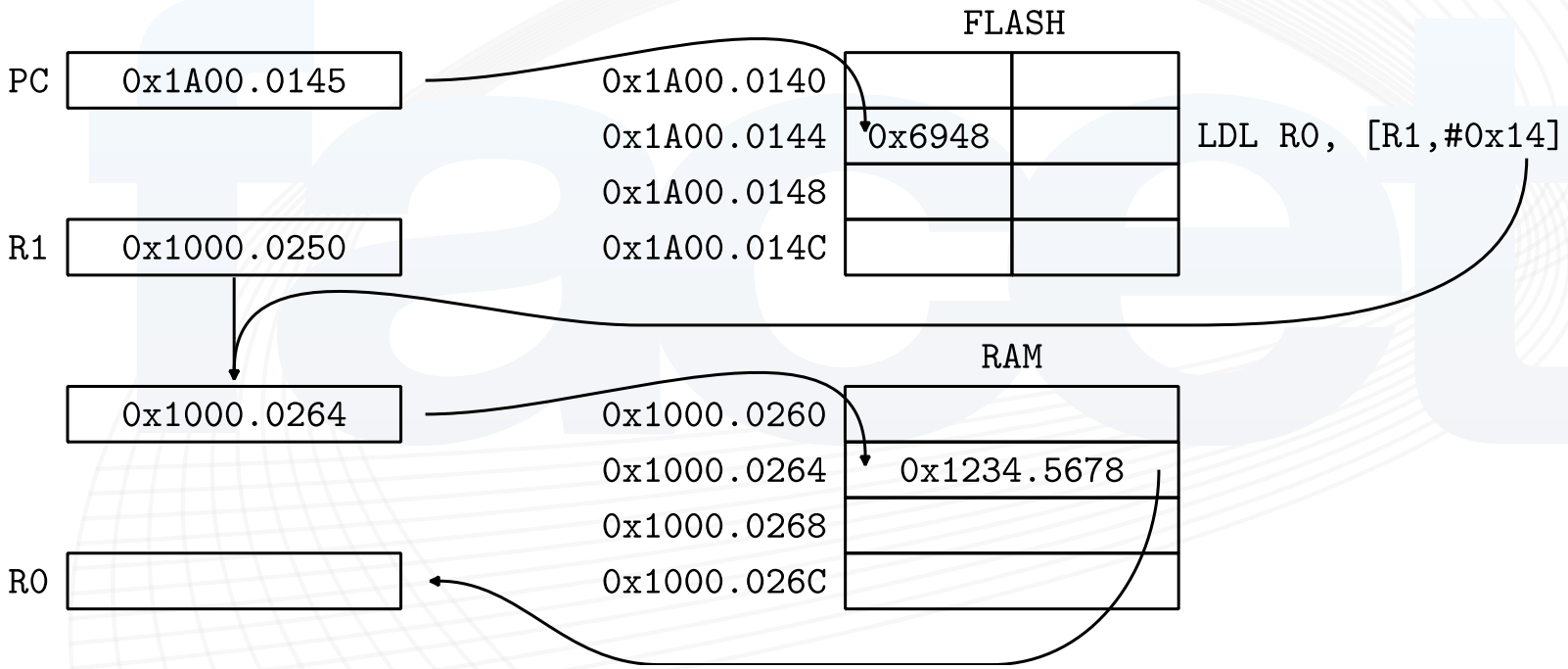


Direccionamiento Indexado

- La dirección del dato se obtiene de sumar un registro más una constante.
- La constante se llama offset o desplazamiento.
- En ARM son12 bits con complemento a 2.
- El registro se llama dirección base.
  - `LDR R0,[R1,#20] // R0 ← M[R1+20]`
- Registro indirecto, caso particular con offset=0
- ¿Se puede acceder a toda la Memoria?
- ¿Para qué sirve el desplazamiento?

Direccionamiento Indexado

LDR R0,[R1,#20] // R0 ← M[R1+20]



Direccionamiento Relativo

- Permite especificar la dirección del operando como la suma de un desplazamiento respecto al PC actual.
- Ventaja: si muevo el programa en M los operandos no cambian.
- El desplazamiento es de dos bytes con signo, y marca la distancia entre la dirección a que apunta PC y el dato.
- En assembler uno puede poner una etiqueta simbólica, ej: “dato1”
- El assembler calcula el desplazamiento ☺

Direccionamiento Relativo

```
tabla:    .word 0x0000.1230
          LDR R1, tabla ; r1 ← M(tabla)
```

- El assembler calcula la diferencia entre la dirección “tabla” y el valor del PC.
- Reemplaza el valor del offset al traducir.

```
LDR R1, [PC,#offset]
```

- Hay un límite para offset, si se pasa da un error de ensamblado (ver manual ARM Cortex-M4 pag 72).
- Como R15 es PC en ARM, este es un caso particular de direccionamiento indexado.

Carga de Puntero a ROM

- Cargar el registro con la dirección de memoria de un dato, cuya dirección es “label”.

```
ADR   R1, label
```

- Es una “seudoinstrucción”, el assembler la traduce así

```
ADD R1, PC, #offset
```

- Offset es el desplazamiento entre PC y label, puede ser una suma o una resta.
- Tamaño de offset ⇒ 12 bits ⇒ 4095
- Puede usarse SUB cuando es para atrás.
- Assembler suma o resta según corresponda.

Carga de Puntero a RAM y ROM

- Se usa una seudo-instrucción para cargar una constante alta, en general una dirección

```
LDR R3, =0x1234.5678
LDR R3, =label
```

- La constante de 32 bits se pone en un lugar reservado para constantes.
- Assembler calcula offset desde PC y se traduce:

```
LDR R3, [PC,offset]
```

Direccionamiento Relativo

- Se usa también para especificar destino de Saltos.
- Relativo a PC ⇒ código relocalizable.
- Volveremos en la descripción de saltos.
- Valor máximo de desplazamiento es mayor en este caso.

Indexados Adicionales

- ARM presenta varias alternativas en modo indexado.
- Puede o no haber un desplazamiento, como ya se vio.
- El registro base puede mantenerse constante, ya se vio.
- El registro base puede cambiar
- Antes del acceso (pre)
- Posterior al acceso (post)
- SP o PC se pueden usar como un registro también, en ARM, son GPRs.

Indexado Preincrementado

- Es igual al modo indexado: el puntero se incrementa previamente a acceder al dato, luego se guarda.
- Se distingue porque utiliza el símbolo ! a continuación del operando:

```
LDR R1,[R2,#4]!
```

- El operando se busca en memoria R1 ⇐ M(R2+#4)
- A continuación se actualiza R2 ⇐ R2 + 4
- ¿Para qué sirve?

Indexado Postincrementado

- Es igual al modo indexado el puntero se cambia posteriormente a acceder al dato y se guarda.
- Se distingue porque utiliza el desplazamiento fuera de los corchetes del registro indice

```
LDR R1,[R2],#4
```

- El operando se busca en memoria R1 ⇐ M(R2)
- A continuación se actualiza R2 ⇐ R2 + 4

Indexado con registro

- Es igual al modo indexado pero el desplazamiento no es constante sino que esta almacenado en otro registro.
- Se distingue porque se utilizan dos registros dentro de los cochetes.

```
LDR R1,[R2,R3]
```

- El operando se busca en memoria R1 ⇐ M(R2+R3)
- Es muy útil para acceder a tablas utilizando indice variables, por ejemplo en lazos.

Indexado con registro escalado

- Es igual al modo indexado con registro pero al desplazamiento se le puede hacer un shift a la izquierda de hasta 3 lugares.
- Se distingue porque se utilizan dos registros y una constante dentro de los cochetes.

```
LDR R1,[R2,R3,LSL #2]
```

- El operando se busca en R1 ⇐ M(R2+4*R3)
- ¿Para que sirve?.

Indexado con registro escalado

```
LDR R0,[R1,R2,LSL #2]
```

- R1 = 0x1000.3000
- R2 = 3
- Dato en **M(0x1000.3000+3*4) = M(0x1000.300C)**
- Incrementando R2 en 1 accedo 4 bytes más adelante (una palabra)

Facilidades del Ensamblador

- Pseudo-instrucciones
  - Ej: ADR – para cargar una dirección.
  - Se traduce como una suma o resta de PC más offset que se guarda en un reg
  - En gral se traducen como una o más instrucciones
- Directivas al ensamblador
  - Permiten configurar el comportamiento del programa ensamblador
  - Son diferentes para cada ensamblador, nosotros utilizamos las de GCC

Directivas al ensamblador

- `.cpu cortex-m4`
  - Indica para cuál procesador esta escrito el código.
- `.syntax unified`
  - Usa notacion para codigo THUMB2
- `.thumb`
  - Ensamblar el código usando instrucciones THUMB
- `.section .text / .data`
  - Determina la memoria de destino para código o datos
- `.word / .hword / .byte`
  - Almacena constantes en memoria en 4, 2 y 1 bytes

Directivas al ensamblador

- `.align`
  - Realinea el código o dato siguiente a una direccion múltiplo de 4.
- `.pool`
  - Indica al ensamblador el punto para ingresar las constantes necesarias para las pseudo-instrucciones como ADR
- `.global etiqueta`
  - Declara la etiqueta como global y por lo tanto visible al resto de módulos del proyecto
- `.space n`
  - Reserva una cantidad de memoria igual a n bytes.

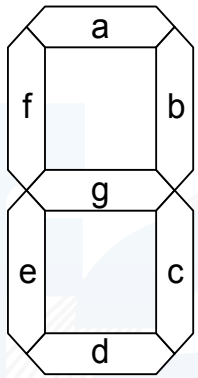
Alineación

- El procesador lee palabras completas de memoria.
- Si un dato de una palabra no está alineado
  - se requiere dos accesos a M para leerlo.
  - Demora el doble.
- Conclusión:
  - Colocar datos de una palabra primero
  - Luego de media y finalmente de una.
- Para instrucciones no importa.

Ejemplo: Conversor de Código

- Se debe tomar un número de varios dígitos, almacenados en un bloque de bytes en M terminado por un valor 0xFF
- El número está almacenado en bytes en RAM a partir de la dirección **“origen”.**
- El número está en BCD sin compactar (cada digito decimal ocupa un byte)
- Se debe traducir cada dígito a su representación para un display de siete segmentos, generando en M un bloque que también finaliza en 0xFF.
- Los dígitos convertidos se deben almacenar a partir de la dirección **“destino”.**

Tabla de Conversión



b7	b6	b5	b4	b3	b2	b1	b0
a	b	c	d	e	f	g	nc

¿Cómo accedo para convertir 3?

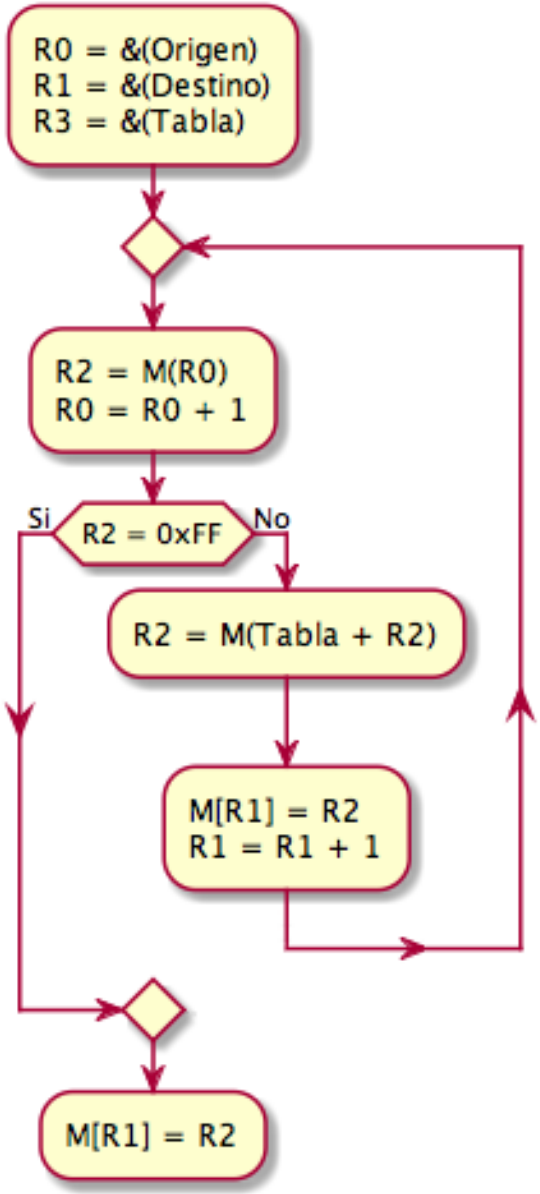
Digito	a	b	c	d	e	f	g	x	Valor
0	1	1	1	1	1	1	0	0	0xFC
1	0	1	1	0	0	0	0	0	0x60
2	1	1	0	1	1	0	1	0	0xDA
3	1	1	1	1	0	0	1	0	0xF2
4	0	1	1	0	0	1	1	0	0x66
5	1	0	1	1	0	1	1	0	0xB6
6	1	0	1	1	1	1	1	0	0xBE
7	1	1	1	0	0	0	0	0	0xE0
8	1	1	1	1	1	1	1	0	0xFE
9	1	1	1	1	0	1	1	0	0xF6

Estructura de Datos y Algoritmo

- Estructura de Datos.
  - Bloque a convertir desde **origen** (máx 20 bytes).
  - Bloque convertido desde **destino**.
  - Carácter de terminación para ambos: 0xFF.
  - Tabla de Conversión.
- Algoritmo.
  - Tomar números de **origen** uno por uno.
  - Convertirlos mediante tabla.
  - Guardarlos en **destino**.
  - Fin: cuando carácter = 0xFF, guardar 0xFF.

Lazo

- **Inicialización**:
  - Punteros a las tablas.
- **Cuerpo**:
  - Tomar un dato.
  - Convertirlo.
  - Guardarlo.
- **Terminación**:
  - ¿Carácter = 0xFF?
  - Escribirlo en destino.
- Terminación puede estar antes de cu
- ¿Dónde Optimizar?



Programa

```
            .cpu cortex-m4          // Indica el procesador de destino
            .syntax unified         // Habilita las instrucciones Thumb-2
            .thumb                  // Usar instrucciones Thumb y no ARM
            .section .data          // Define la sección de variables (RAM)
origen:     .byte 6,2,8,5,7         // Variable inicializada de 5 bytes
            .space 15,0xFF          // Completo los 20 lugares de origen con 0xFF
destino:    .space 20,0x00          // Variable de 20 bytes en blanco
            .section .text          // Define la sección de código (FLASH)
            .global reset           // Define el punto de entrada del código
reset:      LDR  R0,=origen         // Apunta R0 al bloque de origen
            LDR  R1,=destino        // Apunta R1 al bloque de destino
            LDR  R3,=tabla          // Apunta R3 al bloque con la tabla
lazo:       LDRB R2,[R0],#1         // Carga en R2 el elemento a convertir, inc
R0
            CMP  R2,0xFF            // Determina si es el fin de conversión
            BEQ  final              // Terminar si es fin de conversión
            LDRB R2,[R3,R2]         // Cargar en R2 el elemento convertido
            STRB R2,[R1],#1         // Guardar el elemento convertido
            B    lazo               // Repetir el lazo de conversión
final:      STRB R2,[R1]            // Guardar el fin de conversión en destino
stop:       B    stop               // Lazo infinito para terminar la ejecución
            .pool                   // Almacenar las constantes fijas (FLASH)
tabla:      .byte 0xFC,0x60,0xDA,0xF2,0x66
            .byte 0xB6,0xBE,0xE0,0xFE,0xF6
```

Duraciones

```
            .cpu cortex-m4
            .syntax unified
            .thumb
            .section .data
origen:     .byte 6,2,8,5,7
            .space 15,0xFF
destino:    .space 20,0x00
            .section .text
            .global reset
reset:      LDR  R0,=origen       // 2T
            LDR  R1,=destino      // 2T
            LDR  R3,=tabla        // 2T
lazo:       LDRB R2,[R0],#1       // 2T
            CMP  R2,0xFF          // 1T Determina si es el fin de conversión
            BEQ  final            // 3T Si salta, 1T Si no salta
            LDRB R2,[R3,R2]       // 2T
            STRB R2,[R1],#1       // 2T
            B    lazo             // 3T
final:      STRB R2,[R1]
stop:       B    stop
            .pool
tabla:      .byte 0xFC,0x60,0xDA,0xF2,0x66
            .byte 0xB6,0xBE,0xE0,0xFE,0xF6
```

¿Cuánto ocupa, Cuánto demora?

- En total 11 instrucciones.
  - 3 Instrucciones de 32 bits y 8 de 16 bits
  - Espacio total de 28 bytes de ROM
  - El assembler determina como las genera.
- Tabla 10 bytes en ROM
- Datos 40 bytes en RAM
- Duración en peor caso : ¡Hay que mirar el lazo!
  - Hasta 20 veces máximo.
  - 6 instrucciones en el lazo.
    - 3 de 2T y 1 de 1T.
    - 2 Saltos: 1 de 1T y 1 de 3T.
    - Total: (6+1+1+3).20=220T
  - Imposible saber duración exacta!
  - Peor caso: 220T.

Comparación con D. Lógico

- Unos cuantos bytes Vs Hw.
- Si el MCU se usa para otra cosa:
  - Más económico
  - Más flexible.
- **Desventajas?**

Criterios para Programar.

- ¿Una sucesión de “IF” se pone en cualquier orden?
- ¿Cuándo un programa es mejor que otro?
- Tiempo.
- Memoria.
- Claridad
  - Modularidad.
  - Simplicidad.
  - Mantenimiento Futuro.

Síntesis Modos

Implícito	Estado CPU
Inmediato	Constantes
Registros	R1… R3…
Indexado (HX / SP)	
‣ Sin desplazamiento (indirecto)	`[R1]`
‣ Con desplazamiento	`[R1, #offset]`
‣ Preindexado	`[R1, #cte]!`
‣ Postindexado	`[R1],#cte`
‣ Variable	`[R1, R2]`
‣ Variable Escalado	`[R1, R2, LSL #c]`
Relativo (R15=PC)	`[R15, #offset]`

Movimiento de Datos (R, M)

```
LDR{<cond>}{type} Rt,[Rn,{offset}]
STR{<cond>}{type} Rt,[Rn,{offset}]
```

- Type = B, SB, H, SH - tamaño: B byte, H media palabra. S extension en signo.
- No modifican Flags.
- **Cond**: caso de ejecución condicional más adelante.
- ¿Por qué no hay modo inmediato?
- ¿cuándo se carga un byte, se opera en ese tamaño también?

Ejemplos:

Load: dato en memoria a un registro:

- **LDR** – load 32 bits
- **LDRH** – load halfword(16 bit unsigned #) / zero-extend a 32 bits
- **LDRSH** – load signed halfword/ sign-extend a 32 bits
- **LDRB** – load byte (8 bit unsigned #) / zero-extend a 32 bits
- **LDRSB** – load signed byte / sign-extend a 32 bits

Store: dato en un register a memoria

- **STR** –store 32-bit word
- **STRH** – store 16-bit halfword (right-most16 bits of register)
- **STRB** – store 8-bit byte (right-most 8 bits of register)
- ¿No existe STRSH o STRSB?

Ejemplo

Dirección(M)	Dato
0x20000003	0x87
0x20000002	0x65
0x20000001	0xE3
0x20000000	0xE1

- **Little endian:** el byte menos significativo en la menor dirección.
- Supongamos R2 = 0x20000000
- **`LDR R1,[R2]`** → R1 = 0x8765E3E1
- **`LDRB R1,[R2]`** → R1=0x000000E1
- **`LDRSB R1,[R2]`** → R1=0xFFFFFFE1
- **`LDRH R1,[R2]`** → R1= 0x0000E3E1

Modos de direccionamiento

- Indexado y todas sus variables, también para STR
- Registro indirecto: `LDR R0,[R1]`
- Desplazamiento variable: `LDR R0,[R1,R2]`
- Desplazamiento constante: `LDR R0,[R1,#4]`

  `LDR R0,[R1,#-4]`
- Indexado PreIncrementado: `LDR R0,[R1,#4]!`
- Indexado PostIncrementado: `LDR R0,[R1],#8`
- Indexado Escalado: `LDR R1,[R2,R3,LSL #2]`
  - address=R2+R3 x 4
- El registro R3 no se altera en indexado escalado.

Ejemplo

`STR R1,[R0],#4`

- Antes de la instrucción:
  - R0=0x20008000
  - R1=0x76543210
- ¿Cuál es el valor final de R0?

Dirección M	Datos en M
0x20008007	0x00
0x20008006	0x00
0x20008005	0x00
0x20008004	0x00
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

Ejemplo

`STR R1,[R0],#4`

- Antes de la instrucción:
  - R0=0x20008000
  - R1=0x76543210
- ¿Cuál es el valor final de R0?
  - R0= 0x20008004

Dirección M	Datos en M
0x20008007	0x00
0x20008006	0x00
0x20008005	0x00
0x20008004	0x00
0x20008003	**0x76**
0x20008002	**0x54**
0x20008001	**0x32**
0x20008000	**0x10**

Ejemplo

`STR R1,[R0,#4]!`

- Antes de la instrucción:
  - R0=0x20008000
  - R1=0x76543210
- ¿Cuál es el valor final de R0?

Dirección M	Datos en M
0x20008007	0x00
0x20008006	0x00
0x20008005	0x00
0x20008004	0x00
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

Ejemplo

`STR R1,[R0,#4]!`

- Antes de la instrucción:
  - R0=0x20008000
  - R1=0x76543210
- ¿Cuál es el valor final de R0?
  - R0= 0x20008004

Dirección M	Datos en M
0x20008007	**0x76**
0x20008006	**0x54**
0x20008005	**0x32**
0x20008004	**0x10**
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

Convención

- Algunos autores dibujan hacia arriba las direcciones crecientes.
- Otros lo hacen para abajo – es mi costumbre.
- En el ejemplo están para arriba.

Tamaño offset

- Indexado: -255 a 4095.
- Pre y post indexado: -255 a 255.
- Relativo: -4095 a 4095
- ¿Cuánto el desplazamiento variable con registro?
- Algunas variantes se verán más adelante.
- Alineación de los datos
  - Los datos de media palabra deben estar alineados en direcciones pares.
  - De palabra completa en direcciones múltiplos de 4.
  - Se accede palabras completas, sin alinear toma más tiempo.
  - ¿De a bytes cómo se alinean?

Ejemplo de Aplicación

Poner en 0 un área de memoria de 15 bytes a partir de la dirección 0x1008.0000

```
        .equ bloque 0x10080000
        …
        MOV  R1,#15
        MOV  R2,#0
        LDR  R3,=bloque
lazo:   STRB R2,[R3],#1 // M(R3) = 0 y R3 = R3 + 1
        SUBS R1,#1      // R1 = R1 - 1
        BNE  lazo
        …
```

Inst. Procesamiento de Datos

- En resumen:
  - Aritméticas: **ADD ADC SUB SBC RSB RSC**
  - Lógicas: **AND ORR EOR BIC**
  - Comparaciones: **CMP CMN TST TEQ**
  - Movimiento de datos: **MOV MVN**
- Estas instrucciones solo funcionan en registros **no en memoria**.
- Syntaxis:

  <Operation>{<cond>}{S} Rd, Rn, Operand2

  - Comparación solo cambia flags – no especifican Rd
  - **{S}** sufijo que indica que se afecta el valor de los flags.
  - El segundo operando va al ALU via barrel shifter.
  - **<cond>**: en bloques condicionales, más adelante.
  - Si se omite un operando, el primero es fuente y destino.

Operaciones Aritméticas

- **ADD{S}**: [Rd] ← Op1 + Op2
- **SUB{S}**: [Rd] ← Op1 – Op2
- **RSB{S}** (reverse subtract): [Rd] ← Op2 – Op1
- ¿Para qué RSB si tenemos SUB? (Op2 opciones?)
- El segundo operando es flexible, como se vio en el tema anterior (barrel shift).

- **ADD/SUB/RSB** se ejecutan solo con operandos de 32-bit
- **ADDS/SUBS/RSBS** para modificar los flags Z/N/C/V

- Y si tenemos datos de 8 bits o 16 bits?
  - **Los flags no reflejarían los resultados de 8 o 16 bits.**
  - Los datos deben estar extendidos apropiadamente a palabras.
  - Reflejar resultado de operaciones de palabras extendidas no siempre refleja resultado operaciones de 8 o 16 bits.
  - Hay solo un ALU de 32 bits en el CPU

Suma (resta) con Carry (borrow)

- CPU: suma/resta operandos de 32-bit con carry/borrow de una operación previa.

- **ADC**(add con carry): [Rd] ← Op1 + Op2 + C
- **SBC**(subtract con carry*): [Rd] ← Op1 – Op2 + (C –1)
- **RSC**(reverse subt. con carry*): [Rd] ← OP2 – Op1 + (C –1)

- * C=0 indica “borrow” para la resta.

Ej: Suma de doble precisión

Sumar dos números de 8 bytes c/u, almacenados cada uno en ocho direcciones consecutivas a partir de la dirección **datos**. Almacenar el resultado a continuación del segundo operando.

```
…
LDR   R0,=datos     // R0 ← dirección de 1er dato
LDR   R1,[R0],#4    // R1 palabra menos signif. 1°
LDR   R2,[R0],#4    // R2 palabra más signif. 1°
LDR   R3,[R0],#4    // R3 palabra menos signif 2°
LDR   R4,[R0],#4    // R4 palabra más signif. 2°
ADDS  R1,R3         // Sumo y toco flags - Carry
STR   R1,[R0],#4    // Guardo Result menos signif.
ADC   R2,R4         // sumo con el carry anterior
STR   R2,[R0],#4    // Guardo Result más signif.
…
```

Desplazamientos y Rotaciones

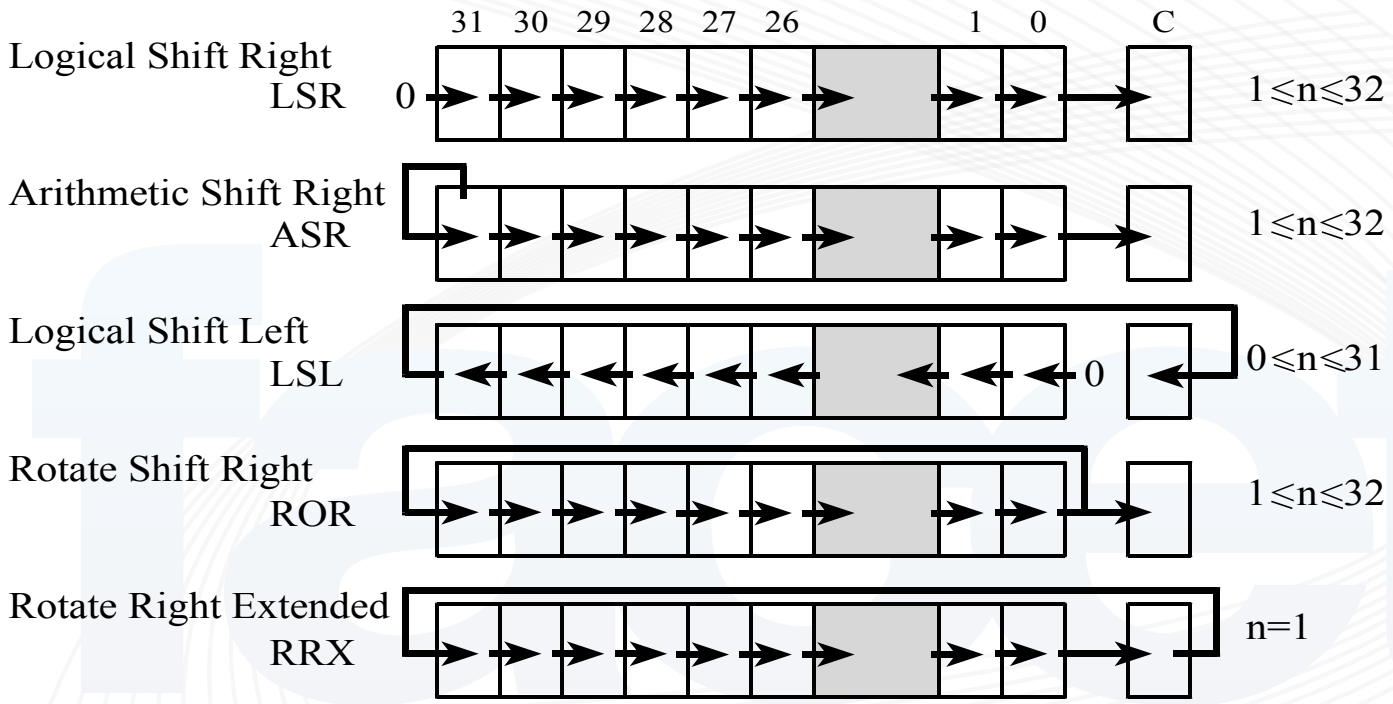
- Repaso: El segundo operando al ALU llega a través de un Hw especial llamado ***Barrel Shifter.***
- Ello permite generar el operando flexible: “Operand2” en forma de registro o constante
- Ejemplo registro:

```
ADD R1,R0,R0,LSL #3 // R1=R0+R0<<3 = 9*R0
```

- La constante inmediata de 0 a 31.
- Otra forma del operando2 flexible es que sea una constante de 8 bits que se puede desplazar.
- Ejemplo Constante

```
ADD R1,R0,#0x1A, LSL #8 ; R1 = R0 + #0x1A00
```

Operaciones de Desplazamiento



- Usar ASR para dividir por 2n con números signados. LSR con números no signados. Operaciones para Operand2 o instrucciones en sí.
- OJO: solo toca el carry si usa {S}
- ¿por qué n<=32? ¿No hay ASL? ¿No hay RLX o ROL?

Ejemplo de Aplicación

- Estas instrucciones permiten multiplicar y dividir en potencias de 2 en forma rápida.
- Por ejemplo, calcular el promedio de dos números almacenados en direcciones consecutivas a partir de “datos” y colocar el resultado a continuación.

```
LDR R0,=datos
LDR R1,[R0]
LDR R2,[R0,#4]
ADD R1,R2
ASR R1,R1,#1
STR R1,[R0,#8]
…
```

Instrucciones de Desplazamiento

- Cuando se usan operaciones en el Operand2
  - El registro desplazado permanece inalterado.
- Para alterar el registro
  - Usar Instrucciones de Desplazamiento
  - Similares a las Operaciones, pero cambian el registro.
- Si se desea alterar cualquier flag con desplazamientos, debe usarse el sufijo “S”.

Ejemplo de Aplicación

- Determinar si el valor almacenado en la dirección **dir** es un numero par (shift right y ver carry)

```
        LDR    R1,=dir
        LDR    R1,[R1]
        MOVS   R0,R1,LSR #1
        BCC    par
        ...

par:    ...
```

Multiplicación

- Productos de dos números de 32 bits entran en 64 bits.
  - Multiplicador y Divisor por Hw.
- **`MUL{S} {Rd,}Rm,Rn`**
  - Guarda solo los 32 bits menos significativos del producto en Rd
  - Resultado vale tanto para operandos con o sin signo
  - No hay una forma inmediata para Op2
- **`MULS {Rd,}Rm,Rs`**
  - MULS actualiza los flags N y Z (C y V no cambian)
- **`UMULL/SMULL RdLo,RdHi,Rm,Rs`**
  - Unsigned (UMULL) y Signed (SMULL): **"Long Multiply"**
  - Producto de 64-bit; P[63:0] se guarda en dos registros:
  - [RdHi] ← P[63:32], [RdLo] ← P[31:0]
  - Los códigos de condición no varían en UMULL.
- **Rm,Rs solo pueden ser los registros R0-R7**

División

- **UDIV/SDIV Rd,Rn,Rm**
  - Unsigned (UDIV) y Signed (SDIV)
  - División entera: Rd = Rn÷Rm(= Rn/Rm)
  - Puede emplearse la forma
    - UDIV/SDIV “Rn, Rm” /Rn = Rn÷Rm
  - El Resultado se trunca (redondea para abajo siempre)
- Resultado = “cociente”, el “resto” se descarta.
- Los flags no se afectan.

Ejemplo: sentencia en C

y = a*(b+c);

- Assembler:

```
LDR R4,=b          // dirección de b
LDR R0,[R4]        // valor de b
LDR R4,=c          // dirección de c
LDR R1,[R4]        // valor de c
ADD R2,R0,R1       // calcular resultado parcial
LDR R4,=a          // dirección de a
LDR R0,[R4]        // valor de a
MUL R2,R2,R0       // valor final de y
LDR R4,=y          // dirección de y
STR R2,[R4]        // guardar en memoria y
```

Operaciones Lógicas

`AND {Rd,} Rn,Op2`	AND bit a bit. Rd ← Rn & operand2
`ORR {Rd,} Rn,Op2`	OR bit a bit. Rd ← Rn | operand2
`EOR {Rd,} Rn,Op2`	XOR bit a bit. Rd ← Rn ^ operand2
`ORN {Rd,} Rn,Op2`	OR NOT bit a bit. Rd ← Rn | (NOT operand2)
`BIC {Rd,} Rn,Op2`	Bit clear. Rd ← Rn & NOT operand2
`BFC Rd,#lsb,#width`	Bit field clear. Rd[(width-1+lsb):lsb] ← 0
`BFI Rd,Rn,#lsb,#width`	Bit field insert. Rd[(width-1+lsb):lsb] ← Rn[(width-1):0]
`MVN Rd,Op2`	Move NOT, logically negate all bits. Rd ← NOT Op2

- Por simplicidad no se incluyó {S}, usar para cambiar flags

Manipulación de Bits

- La función AND entre una variable y constante permite fijar bits en 0

X_7	X_6	X_5	X_4	X_3	X_2	X_1	X_0	Registro (Variable)
0	0	0	0	1	1	1	1	Mascara (Constante)
0	**0**	**0**	**0**	X_3	X_2	X_1	X_0	Resultado AND

- La función OR entre una variable y constante permite fijar bits en 1

X_7	X_6	X_5	X_4	X_3	X_2	X_1	X_0	Registro (Variable)
0	0	0	0	1	1	1	1	Mascara (Constante)
X_7	X_6	X_5	X_4	**1**	**1**	**1**	**1**	Resultado OR

Manipulación de Bits

- Una operación AND con una mascara filtra los bits en cero y mantiene los bits en 1 de la variable. (R = V & M)
- Actualizar una variable usando una operación OR con una mascara fija los bits en 1 y mantiene el resto sin cambios (V = V | M).
- Actualizar una variable usando una operación AND con la mascara invertida fija los bits en 0 y mantiene el resto sin cambios (V = V & ~M)

Manipulación de Bits

- Actualizar una variable usando una operación XOR con una mascara invierte los bits en 1 y mantiene el resto sin cambios (V = V ^ M).
- La mascara M puede ser una constante o una variable, pero siembre se conocer su valor de antemano.
- No se conoce el valor de la variable V y se busca modificar o analizar solo algunos bits de la misma.
- Estas operaciones se llaman enmascaramientos.

Ejemplo de aplicación

- Una forma simple de detectar alteraciones en los datos se agregar un bit de paridad.
- Escriba un programa que cambie el valor del bit 0 para mantener la cantidad de 1 siempre en un numero par.

Ejemplo de aplicación

- Compactar en un solo byte en la dirección **result** los dos numero BCD de 4 bits almacenados en las direcciones **high** y **low**

result = (high << 4) | low

```
LDR  R0,=high
LDRB R1,[R0]        // leer valor de High
LSL  R1,R1,#4       // moverlo a su posición final
LDR  R0,=low
LDRB R2,[R0]        // leer el valor de Low
ORR  R1,R1,R2       // combinar ambos valores
LDR  R0,=result
STRB R1,[R0]        // guardar el resultado
```

0	0	0	0	h_3	h_2	h_1	h_0	Valor de **high** en **R1**
h_3	h_2	h_1	h_0	0	0	0	0	Valor de **R1** luego de **LSL**
0	0	0	0	l_3	l_2	l_1	l_0	Valor de **low** en **R2**
h_3	h_2	h_1	h_0	l_3	l_2	l_1	l_0	Resultado de la instrucción **ORR**

Alternativa

- Compactar en un solo byte en la dirección **result** los dos numero BCD de 4 bits almacenados en las direcciones **high** y **low**

result = (high << 4) | low

```
LDR  R0,=high
LDRB R1,[R0]        // leer valor de High
LDR  R0,=low
LDRB R2,[R0]        // leer el valor de Low
BFI  R2,R1,#4,#4    // combinar ambos valores
LDR  R0,=result
STRB R2,[R0]        // guardar el resultado
```

0	0	0	0	h_3	h_2	h_1	h_0	Valor de **high** en **R1**
0	0	0	0	l_3	l_2	l_1	l_0	Valor de **low** en **R2**
h_3	h_2	h_1	h_0	l_3	l_2	l_1	l_0	Resultado de la instrucción **BFI**

Resumen operaciones de bits

`AND {Rd,}Rn,Op2`	Para poner bits en 0.
`ORR {Rd,}Rn,Op2`	Para poner bits a 1.
`EOR {Rd,}Rn,Op2`	Para invertir bits.
`ORN {Rd,}Rn,Op2`	Todo bit en 0 en Op2 se pone en 1 en destino.
`BIC {Rd,}Rn,Op2`	Todo bit en 1 en Op2 se pone en 0 en destino.
`BFC Rd,#lsb,#width`	Poner en cero un campo de longitud width a partir de un determinado bit lsb.
`BFI Rd,Rn,#lsb,#width`	Inserta un campo de Rn en Rd
`MVN Rd,Op2`	Poner el operando negado lógico en Rd

Op2 – op. flexible, registro o cte 8 bits, desplazados o no

0x000000XY, 0x0000XY00, 0x00XY0000, 0xXY000000

Comparación y TEST

- Hace una resta sin guardar el resultado. Se usa para cambiar los flags.

  CMP: Rn – Op2 Afecta **Z**, **N**, **V** y **C**

- Test para saber si algún bit es 1.

  TST: Rn and Op2 Afecta solo a **Z** y **N**

- TEQ para saber si dos operandos son iguales.

  TEQ: Rn xor Op2 Afecta solo a **Z** y **N**

- Usar en lugar de **CMP** para comprobar igualdad, si no se desea modificar a **C** y **V** (ej. para aritmética múltiple precisión)
- A Op2 se le llama máscara - (2º operando flexible).

Saltos

- Toda instrucción que modifique el PC es un salto.
- Salto incondicional relativo, destino < ±16MB:

```
B label      // PC ⇦ label
```

- Salto condicional relativo, destino < ±1MB (cc es la condición):

```
BNE label    // Si Z entonces PC ⇦ label
```

- Saltos absoluto con direccionamiento de registro indirecto

```
BX Rn        // PC ⇦ Rn (El bit 0 de Rn = 1)
```

  - El bit cero de Rn debe ser 1 para indicar modo THUMB, para compatibilidad, sino error en ejecución.
- Se recomienda usar saltos y no usar PC como registro general. Leer set de instrucciones

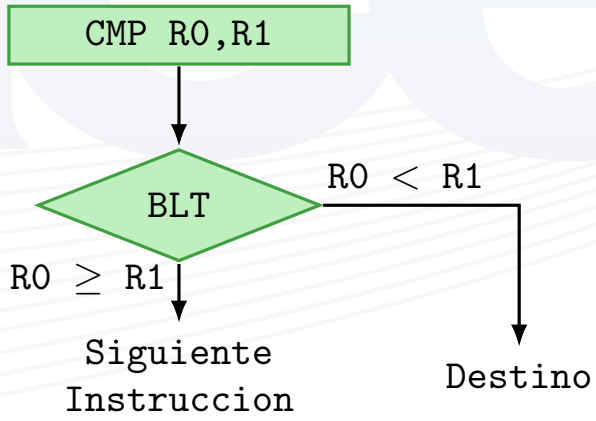
Condiciones

Sufijo	Significado	Condición	Flags	Inverso	
EQ	Equal	Rn = Op2	Z=1	NE	Indistinto
NE	Not Equal	Rn ≠ Op2	Z=0	EQ	
GT	Greater than	Rn > Op2	Z=0 and N=V	LE	Con signo
GE	Greater or equal	Rn ≥ Op2	N=V	LT	
LE	Less or equal	Rn ≤ Op2	Z=1 or N<>V	GT	
LT	Less than	Rn < Op2	N<>V	GE	
HI	Higher	Rn > Op2	C=1 and Z=0	LS	Sin Signo
HS	Higher or same	Rn ≥ Op2	C=1	LO	
LS	Lower or same	Rn ≤ Op2	C=0 or Z=1	HI	
LO	Lower	Rn < Op2	C=0	HS	
MI	Minus	Rn < 0	N=1	PL	
PL	Plus	Rn ≥ 0	N=0	MI	
CS	Carry set	Acarrero	C=1	CC	Simples
CC	Carry clear	Sin acarreo	C=0	CS	
VS	Overflow set	Overflow	V=1	VC	
VC	Overflow clear	Sin overflow	V=0	VS	

Saltos condicionales con signo

- Este grupo de saltos esta pensado para ser ejecutado después de una comparación de números signados

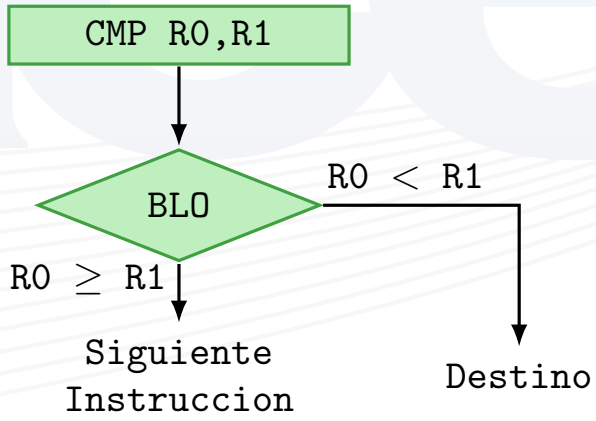
```
SUBS, CMN or CMP
BLT target      // menor que, signada
BLE target      // menor o igual que, signada
BGE target      // mayor o igual que, signada
BGT target      // mayor, signada
```



Saltos condicionales sin signo

- Este grupo de saltos esta pensado para ser ejecutado después de una comparación de números signados

```
SUBS, CMN or CMP
BLO target      // menor que, sin signo
BLS target      // menor o igual, sin signo
BGS target      // mayor o igual que, sin signo
BGT target      // mayor que, sin signo
```



Comparar Rn con 0 y saltar

```
CBZ     Rn,label      // si Rn=0 salte a label
CBNZ    Rn,label      // si Rn≠0 salte a label
```

- Permite testear el valor de un registro contra cero y no cambia los flags.
- CBNZ para terminar un lazo “for”, con Rn inicializado en la cantidad de veces que se ejecuta el lazo.
- CBZ similar, para poner al comienzo del cuerpo del lazo.
- Solo se puede saltar hacia delante (4 a 130 bytes).
- Solo se puede usar R0 a R7.