Conceptos básicos del ensamblador y arquitectura ARM

Leonardo Scándolo
Diego Feroldi

Arquitectura del Computador [*]
Departamento de Ciencias de la Computación
FCEIA-UNR

[*] Actualizado 18 de noviembre de 2021 (D. Feroldi, feroldi@fceia.unr.edu.ar)

Índice

1. La arquitectura ARM

ARM es una arquitectura de la familia RISC, que significa *Reduced Instruction Set Computer* (Computación/Computadora de conjunto de instrucciones reducido). Inicialmente, las siglas de ARM significaban *Acorn RISC Machines* porque la compañía Acorn fue la primera en crear la arquitectura y procesadores ARM. Eventualmente, Acorn, junto con Apple y otras compañías, crearon la compañía Advanced RISC Machines (ARM) para que controle el desarrollo y mantenimiento de la arquitectura ARM y el diseño de los procesadores. Sin embargo, dicha compañía no fabrica los procesadores, sino que vende licencias a otras compañías para que fabriquen los procesadores que ellos diseñan.

La arquitectura ARM se creó con el propósito de desarrollar procesadores que sean sencillos de fabricar, que tengan pocas instrucciones y que de esa manera sea más simple poder generar diseños sin errores. El hecho de poder procesar un conjunto pequeño de instrucciones implica que la cantidad de transistores usados en un procesador ARM es mucho menor que la de los procesadores CISC de prestaciones equivalentes. Otra ventaja de la cantidad limitada de transistores es que los procesadores que utilizan la arquitectura ARM usan menos electricidad y generan menos calor. Sin embargo, también trae desventajas, dado que los programas en una arquitectura ARM generalmente son mucho más grandes que los equivalentes en arquitecturas CISC.

Como hemos dicho, los procesadores que diseña la compañía ARM utilizan la arquitectura llamada también ARM. A lo largo de los años se han creado muchas versiones de dicha arquitectura. La primera es la denominada *ARMv1*, la cual fue lanzada en 1985. A la fecha de la creación de este documento, los procesadores ARM más comunes para sistemas embebidos de buen desempeño (como teléfonos celulares o tabletas) son los que utilizan arquitectura *ARMv7-A*. Hasta esta versión todas las arquitecturas habían sido de 32 bits (menos las primeras 2 que tenían un rango de direcciones de 26 bits). La última versión de la arquitectura, *ARMv8-A* es una arquitectura de 64 bits. En este documento hablaremos de los aspectos más importantes de la arquitectura ARM hasta la versión *ARMv7-A*. En la Tabla 1 se listan las versiones de la arquitectura ARM y algunos procesadores que utilizan dichas arquitecturas.

Tabla 1: Versiones de la arquitectura ARM y procesadores que las utilizan.

Arquitectura	Bits	Procesadores
ARMv1	32/26	ARM1
ARMv2	32/26	ARM3 ARM3
ARMv3	32	ARM6 ARM7
ARMv4	32	ARM8
ARMv4T	32	ARM7TDMI ARM9TDMI
ARMv5	32	ARM7EJ ARM9E ARM10E
ARMv6	32	ARM11
ARMv6-M	32	ARM Cortex-M0 ARM Cortex-M1
ARMv7-M	32	ARM Cortex-M3
ARMv7E-M	32	ARM Cortex-M4
ARMv7-R	32	ARM Cortex-R4 ARM Cortex-R5 ARM Cortex-R7
ARMv7-A	32	ARM Cortex-A5 ARM Cortex-A7 ARM Cortex-A8 ARM Cortex-A9 ARM Cortex-A12 ARM Cortex-A15 ARM Cortex-A17
ARMv8-A	32/64	ARM Cortex-A53 ARM Cortex-A57

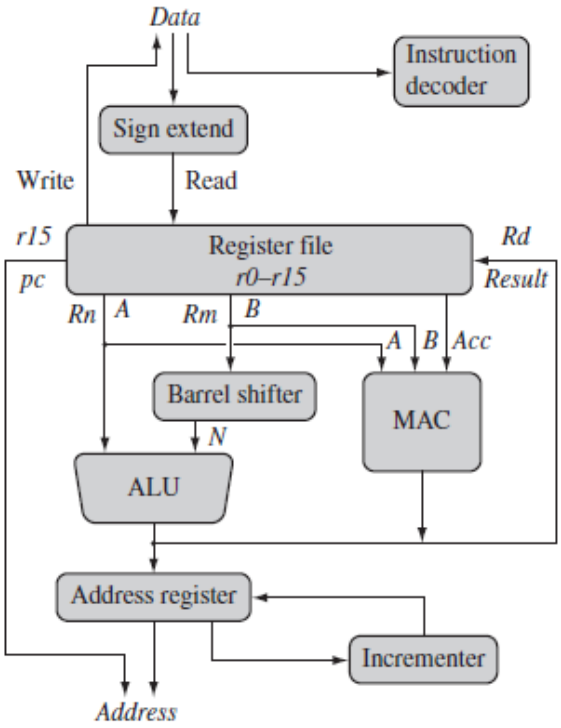


Figura 1: Arquitectura ARM tipo Von Neumann.

Como en casi todas las arquitecturas RISC, las instrucciones de la arquitectura ARM son de tamaño fijo, en este caso 32 bits. En la Fig. 1 se muestra una implementación Von Neumann de arquitectura ARM.

1.1. Registros

La arquitecturas ARM utiliza 16 registros de 32 bits para los programas en modo usuario, llamados r_0 a r_{15}, más un registro de banderas llamado CPSR (Current Program Status Register[1]). Además, y dependiendo de las características del procesador, existirán registros que serán accesibles solamente en modo de ejecución privilegiado o durante el proceso de una interrupción.

La convención de llamadas de ARM es llamada AAPCS (*Arm Arquitecture Procedure Call Standard*[2]). Dicha convención establece las siguientes reglas para llamadas a procedimientos:

- r_0 hasta r_3 (a veces también llamados a_1 hasta a_4) son usados para pasar argumentos. Si no alcanzan estos 4 registros, el resto de los argumentos se pasa utilizando la pila.
- r_0 y r_1 son usados para almacenar el valor de salida. Si no alcanzan estos 2 registros, se devolverá r_0 como un puntero al lugar donde está almacenado el resultado.
- r_4 a r_{11}, también llamados v_1 a v_8, pueden ser usados como variables de propósito general.
- r_9 en algunas arquitecturas tiene un uso especial.
- r_0 a r_3 son caller-save.
- r_4 a r_{11} son callee-save.

[1]Registro de estado del programa actual.
[2]Convención de llamadas de procedimiento de la arquitectura ARM.

- r_{12}, llamado también IP, es usado por el linker si es necesario para hacer saltos largos y poder usar todo el espacio de memoria de 32 bits.
- r_{13}, llamado también SP, se usa como *stack pointer*.
- r_{14}, llamado también LR[3], guarda la dirección de retorno.
- r_{15}, llamado también PC, es el *program counter*.
- Para las arquitecturas con VFP[4], los argumentos de punto flotante se pasan, y el valor de retorno se devuelve, en los registros de punto flotante s_0 a s_{15}, que son caller-save. Los registros s_{16} a s_{31} son callee-save.

1.2. Endianness

La arquitectura ARM es denominada *bi-endian*. Esto quiere decir que puede ser configurada para funcionar en modo *little-endian* y en modo *big-endian*. La manera de configurarlo es con una instrucción especial llamada `SETEND`. A partir de la arquitectura ARMv6 Se puede saber si el procesador está funcionando en modo big-endian o little-endian inspeccionando el registro CPSR que se describe a continuación.

Ejemplos:

- `SETEND BE`
  En el CPSR es E=1, por lo tanto opera en modo *big-endian*
- `SETEND LE`
  En el CPSR es E=0, por lo tanto opera en modo *little-endian*

1.3. CPSR

El registro de status (CPSR) guarda el estado actual del procesador. La Fig. 2 muestra el contenido de algunos de sus bits. De particular importancia resultan los bits 28 a 31 para implementar ejecución condicional como se verá a la sección correspondiente.

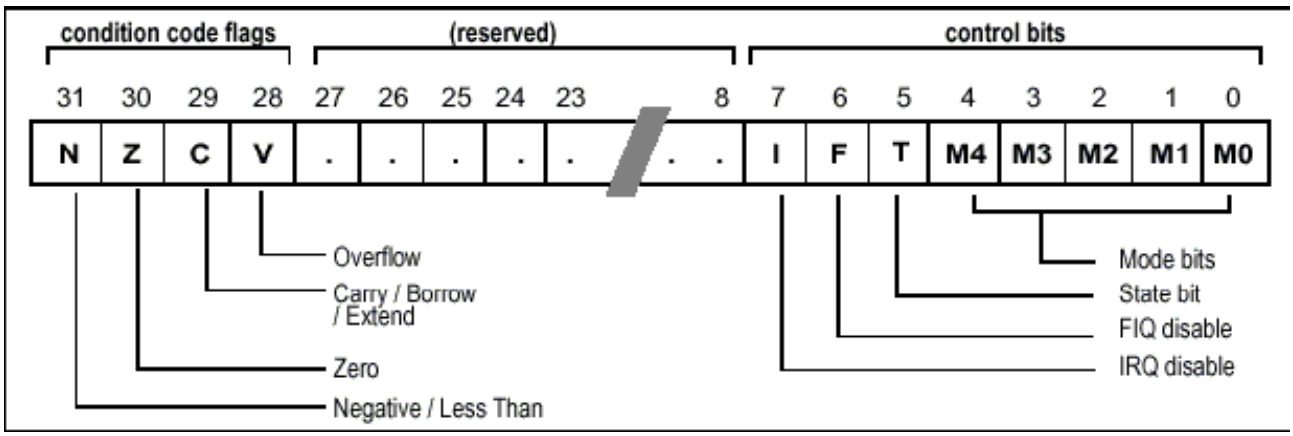


Figura 2: Estructura del registro CPSR.

El significado de los bits del CPSR es el siguiente:

[3] *Link Register.*
[4] *Vector Floating Point* (Punto flotante vectorizado).

- Los bits 0 a 4 contienen el código del modo en el cual se encuentra el procesador. Puede indicar que está en modo privilegiado, modo usuario o respondiendo a una interrupción, entre otros.
- El bit 5 indica si el procesador se encuentra en modo ARM o modo THUMB. THUMB es un modo especial que permite instrucciones más cortas. Para más información ver [4].
- El bit 6 indica si las interrupciones rápidas están habilitadas.
- El bit 7 indica si las interrupciones normales están habilitadas.
- El bit 9 (a partir de ARMv6) indica si el procesador funciona en modo *big-endian* (1) o *little-endian* (0). Puede ser cambiado con la instrucción `SETEND`.
- El bit 28, también llamado V, es el bit de *overflow*. Indica si una operación con signo tuvo un resultado más grande que 31 bits.
- El bit 29, también llamado C, es el bit de *carry*. Indica si una operación de suma, resta, comparación o *shift* tuvo un resultado más grande que 32 bits.
- El bit 30, también llamado Z, indica si el resultado de una operación fue cero.
- El bit 31, también llamado N, indica si el resultado de una operación fue negativo.

2. Instrucciones ARM

La arquitectura ARM es una arquitectura *Load/Store*. Esto quiere decir que para cualquier operación aritmética o lógica los operandos siempre serán registros o valores inmediatos. La única manera de acceder a memoria general es usar operaciones especiales que cargan valores de memoria a registros o que guardan valores de registros a memoria. Las instrucciones ARM comúnmente tienen dos o tres operando.

En general, las operaciones lógicas, aritméticas o de comparación pueden tener la siguiente forma:

$$\texttt{opcode}[condición][S] \quad r_d, r_n, \{operando2\}$$

- El `opcode` es un nombre de 3 letras que designa a la operación (ej: `MOV`, `ADD`, etc.).
- El sufijo de *condición* es un código de 2 letras opcional que permite ejecutar la operación sólo si algunos bits del CPSR cumplen alguna condición. Esto será explicado en la sección siguiente.
- El sufijo S se utiliza para designar que una operación modifique las banderas (últimos 4 bits) del CPSR. En las operaciones de comparación no es necesario usarlo.
- r_d es el registro destino.
- r_n es el registro del primer argumento.
- *operando2* es el segundo argumento (opcional dependiendo de la instrucción), que puede ser un registro o un valor inmediato. Este segundo argumento puede también especificar una operación de *shift* o *roll* que se aplicará antes de ser usado para la operación. Esta operación adicional se explicará en la Sección 2.2.

Algunos ejemplos simples de instrucciones:

- `MOV r0, #0`
  Mover el valor 0 a r0.
- `MOV r7, r5`
  Mover el valor de r5 a r7
- `MOV r7, r5, LSL #2`
  r7:= r5×4 = (r5 ≪ 2) (El operador LSL se verá en la Sección 2.3)
- `ADD r0, r1, r2`
  Sumar r1 y r2, y guarda el resultado en r0.
- `ADC r0, r1, r2`
  Suma con acarreo, r0:=r1+r2+*carry*.
- `SUB r5, r3, r0`
  Resta r0 a r3 y guarda el resultado en r5.[5]
- `SBC r5, r3, r0`
  Resta con acarreo, r5:=r3-r0-NOT(*carry*).
- `MUL r0, r1, r2`
  Multiplica r1 con r2 y guarda el resultado en r0.
- `UMULL r0, r1, r2, r3`
  Multiplica r3 con r2 y guarda el resultado en r0 (bits menos significativos) y r1 (bits más significativos): $[r1, r0] = r2 \times r3$
- `CMP r0, #3`
  Comparar r0 con 3 (modifica las banderas del CPSR)

2.1. Instrucciones de ramificación

Las instrucciones de ramificación permiten cambiar el flujo de ejecución o llamar a rutinas. Estas instrucciones permiten tener subrutinas, estructuras *if-then-else* y bucles. Estas instrucciones cambian el flujo de ejecución forzando al contador de programa a apuntar a la nueva dirección.

Ejemplo:

```
        B   forward
        ADD r1, r2, #4
        ADD r0, r6, #2
        ADD r3, r7, #4
forward
        SUB r1, r2, #4
```

En este ejemplo la ejecución salta a `forward` y las tres instrucciones de suma no se ejecutan.

[5] Notar que esta instrucción no modifica las banderas del CPSR a menos que se agregue el sufijo *s*: `SUBS r5, r3, r0`.

2.2. Ejecución condicional

La arquitectura ARM permite designar algunas operaciones para su ejecución condicional. Esto quiere decir que bajo ciertas condiciones la operación se ejecutará y bajo otras la operación será interpretada por el procesador como un NOP (no-operation). La manera de designar las condiciones a cumplir para la ejecución es a través de un sufijo de 2 letras que se adosa al nombre de la instrucción.

Las condiciones posibles son:

Sufijo	Descripción	Banderas
EQ	Igual / Resultado fue 0	Z
NE	Distinto / Resultado no fue 0	!Z
CS/HS	Carry activo / Mayor o igual (sin signo)	C
CC/LO	Carry inactivo / Menor (sin signo)	!C
MI	Negativo	N
PL	Positive o cero	!N
VS	Overflow	V
VC	Sin overflow	!V
HI	Mayor (sin signo)	$C \wedge !Z$
LS	Menor o igual (sin signo)	$!C \vee Z$
GE	Mayor o igual (con signo)	$N \equiv V$
LT	Menor (con signo)	$!(N \equiv V)$
GT	Mayor (con signo)	$!Z \wedge (N \equiv V)$
LE	Menor o igual (con signo)	$Z \vee !(N \equiv V)$
AL	Siempre se ejecuta (default)	Cualquiera

Las banderas (V,C,Z,N) corresponden a los bits 28 a 31 del CPSR (ver Fig. 2).

Un ejemplo del uso de esta capacidad de la arquitectura ARM es para traducir una estructura de flujo de control `if-then-else` sin necesidad de usar saltos. Si se tiene el siguiente pedazo de código C:

```
if (x == 0)
   y += x;
else
   y = 1;
```

Podemos escribirlo en assembler para ARM de la siguiente manera (asumiendo que `x` e `y` son enteros y están en r_0 y r_1, respectivamente):

```
CMP    r0, #0
ADDEQ  r1, r1, r0
MOVNE  r1, #1
```

Las intrucciones aritméticas/lógicas por defecto no modifican el estado del registro de banderas. Si necesitamos modificar el estado del registro para luego utilizarlo en una instrucción de salto condicional, tenemos que agregar el sujijo `S` al final de la instrucción.

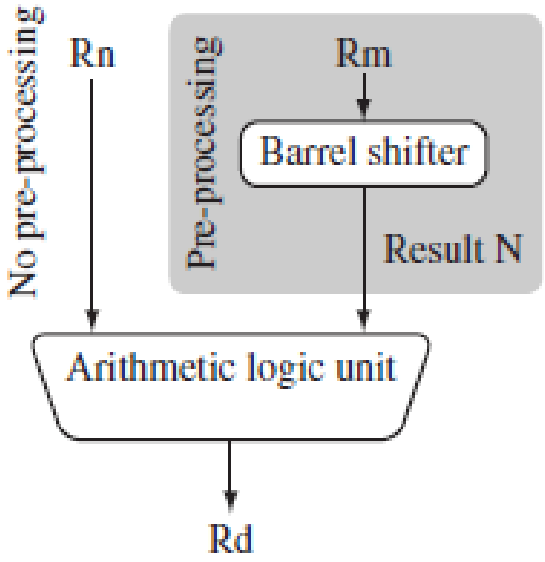


Figura 3: *Barrel Shifter* y ALU.

Ejemplo:

```
MOV  r0, #0xF     @ r0 = 1111
EOR  r0, #2       @ 1111 EOR 0010 = 1101 (no modifica el CPSR)
EORS r0, #2       @ 1101 EOR 0010 = 1111 (y además modifica el CPSR)
```

2.3. Barrel shifter

El *barrel shifter* es una unidad lógica que permite ejecutar ciertas operaciones de movimiento de bits en el último operando de algunas operaciones. Para esto, luego del último operando, se coloca un código de tres dígitos que designa la operación y un valor inmediato que designa la cantidad de bits que mueve la operación. La Fig. 3 muestra el *barrel shifter* y la ALU.

Las operaciones posibles son:

- **LSL** : Logical Shift Left[6], mueve los bits a la izquierda, ingresando ceros por la derecha.
- **LSR** : Logical Shift Right[7], mueve los bits a la derecha, ingresando ceros por la izquierda.
- **ASR** : Arithmetic Shift Right[8], mueve los bits a la derecha, ingresando por la izquierda el mismo valor que el bit más significativo.
- **ROR** : Rotate Right[9], mueve los bits a la derecha, ingresando por la izquierda el mismo valor que se va por la derecha.
- **RRX** : Rotate Right Extended[10], funciona como la operación ROR, pero sobre una palabra de 33 bits, donde el bit de carry del CPSR es el bit 33.

[6] *Shift lógico a la izquierda.*
[7] *Shift lógico a la derecha.*
[8] *Shift aritmético a la derecha.*
[9] *Rotación a la derecha.*
[10] *Rotación a la derecha extendida.*

Algunos ejemplos de uso de esta capacidad:

- `MOV r0, r1, LSL #2`
  Asigna $r_1 \times 4$ a r_0.
- `MOV r2, r2, ROR #16`
  Intercambia los 16 bits más significativos de r2 con sus 16 bits menos significativos.
- `ADD r1, r1, LSL #5`
  Multiplica r_1 por 33 ($r1 = r1 + r1 \times 32$).
- `ADD r9, r5, r5, LSL #3`
  $r9 = r5 + r5 \times 8 = r5 \times 9$
- `RSB r9, r5, r5, LSL #4`
  $r9 = r5 \times 16 - r5 = r5 \times 15$

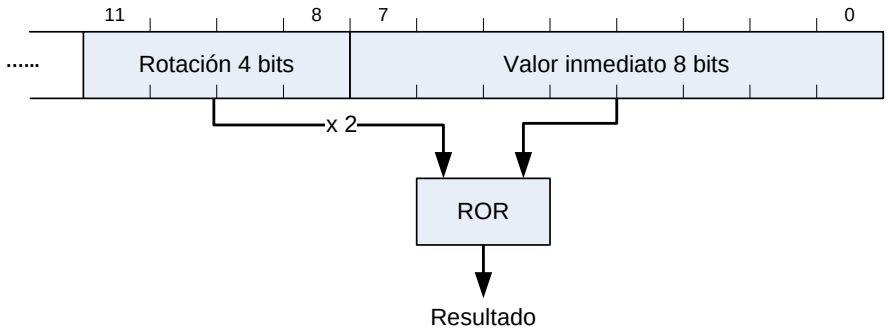
Para usar una instrucción multiplicación, multiplicando por una constante, implica primero cargar la constante en un registro y luego esperar un número de ciclos internos para completar la instrucción. Sin embargo, generalmente se puede hallar una solución más eficiente mediante alguna combinación de `MOVs`, `ADDs`, `SUBs` y `RSBs` con corrimientos. En efecto, la multiplicación por una constante igual a ((potencia de 2) $\pm$ 1) puede ser realizada en un ciclo.

2.4. Valores inmediatos

La arquitectura ARM tiene instrucciones de tamaño fijo de 32 bits donde los 12 bits menos significativos están destinados a almacenar un valor inmediato. Sin embargo, esto no significa que sólo valores entre 0 y $2^{12} - 1$ pueden ser usados como valores inmediatos dado que ARM no interpreta estos 12 bits como un número de 12 bits. En realidad, la estructura que se utiliza para estos 12 bits es la siguiente: 8 bits se usan para definir un valor de 0 a 255 y 4 bits se usan para definir una rotación a la derecha. Si llamamos I al número de 8 bits y R al número de 4 bits usado para rotación, la fórmula utilizada es:

$$\text{Valor inmediato} \leftarrow I \bullet (2 \times R),$$

donde el operador $\bullet$ representa la rotación a la derecha. Este concepto se puede visualizar en el siguiente diagrama:



Dado que el número de 4 bits se duplica, esto implica que se puede hacer una rotación de hasta 30 espacios. Entonces, el procedimiento para formar una constante es el siguiente: los 8 bits del valor inmediato se extienden a 32 bits agregando ceros y luego se rota hacia la derecha la cantidad de veces especificada. La Fig. 4 muestra todas las posibles rotaciones.

Algunos ejemplos de números posibles y no posibles de representar son:

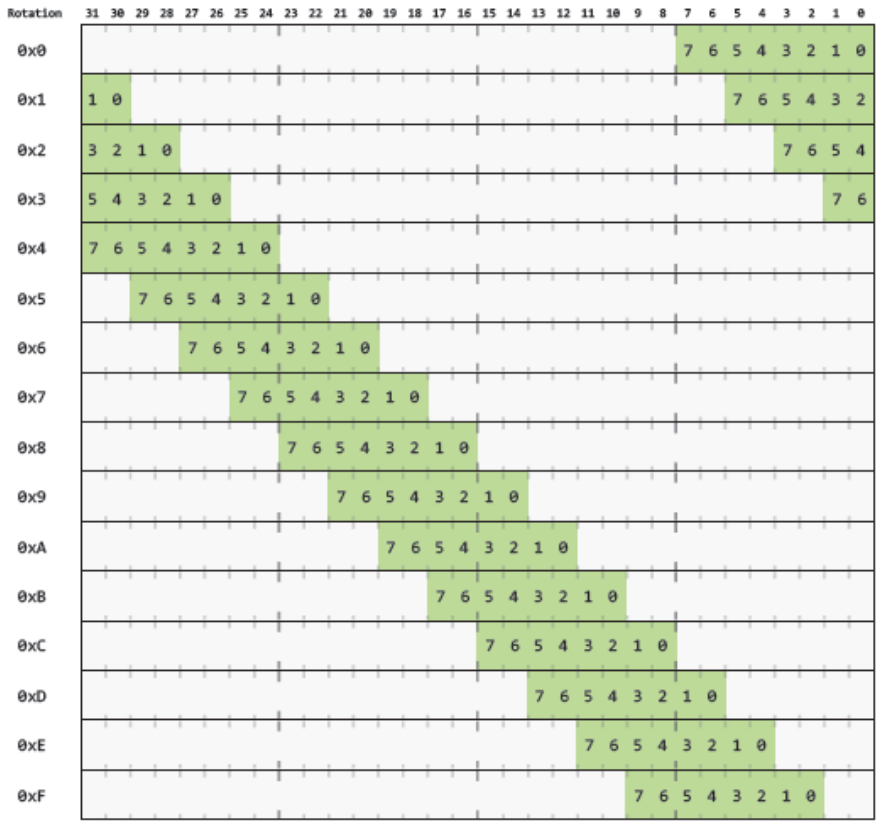


Figura 4: Esquema de posible rotaciones.

- `0x00000000` ✓ (`0x0 ROR 0`)
- `0x000000FF` ✓ (`0xFF ROR 0`)
- `0xFF000000` ✓ (`0xFF ROR 8`)
- `0x00000FF0` ✓ (`0xFF ROR 28`)
- `0x007F0000` ✓ (`0x7F ROR 16`)
- `0xF000000F` ✓ (`0XFF ROR 4`)
- `0xF000F000` × (La constante rotada es demasiado grande)
- `0x00000123` × (La constante rotada es demasiado grande)
- `0x0F0000F0` × (La constante rotada es demasiado grande)

Ejemplos de uso de valores inmediatos:

```
MOV r0, #123            ; Carga el valor 123 en el registro r0
ORR r5, r5, #0x8000     ; Setea el bit 15 del registro r5
EOR r9, r9, #0x80000000 ; Invierte el bit 31 del registro r9
```

Si un programa en C usa una constante que no puede ser expresada a través de uno de los posibles valores inmediatos, deberá guardarse en el ejecutable y ser cargado de memoria a un registro para poder ser usado.

2.5. Operaciones de carga y guardado en memoria

Como se aclaró en secciones anteriores, la arquitectura ARM es una arquitectura que sólo permite acceder a memoria principal del sistema a través de operaciones especializadas de carga y guardado de registros. Existen operaciones de carga y guardado simples (de un solo registro) y operaciones que funcionan con múltiples registros. Ninguna de las operaciones de carga o guardado modifican las banderas del CPSR.

2.5.1. Operaciones de carga y guardado simples

Estas operaciones tienen la forma:

$$\texttt{opcode}[\textit{condición}][\textit{tamaño}] \quad r_d, \{\textit{dirección}\}$$

El *opcode* puede ser:

- `LDR`, significa Load Register[11]. La operación será:
  $r_d \leftarrow$ valor en *dirección*
- `STR`, significa Store Register[12]. La operación será:
  valor en *dirección* $\leftarrow r_d$

Por defecto estas operaciones transfieren el registro completo (32 bits). El sufijo *tamaño* puede usarse para especificar una cantidad de bits distinta a trasferir. Puede ser:

- `B`: Byte sin signo
- `SB`: Byte con signo
- `H`: Media palabra (16 bits) sin signo
- `SH`: Media palabra (16 bits) con signo

Una restricción a tener en cuenta es que la dirección de memoria a utilizar debe ser divisible por la cantidad de bits a transferir. La dirección a usar puede especificarse de las siguientes formas:

[r_n]	La dirección es r_n
[r_n, r_m]	La dirección es $r_n + r_m$
[r_n, #I]	La dirección es $r_n + I$. I es un valor inmediato
[r_n, r_m]!	La dirección es $r_n + r_m$. Esa dirección luego se escribe en r_n.
[r_n, #I]!	La dirección es r_n+I. Esa dirección luego se escribe en r_n.
[r_n], r_m	La dirección es r_n. Luego se escribe $r_n + r_m$ en r_n.
[r_n], #I	La dirección es r_n. Luego se escribe $r_n + I$ en r_n.
[r_n, r_m, LSL #I]	La dirección es $r_n + (r_m << I)$. Puede usarse también LSR, ASR, ROR, RRX.
Etiqueta	La dirección la da una etiqueta en el código.

Algunos ejemplos de instrucciones de carga pueden ser:

[11] Cargar registro.
[12] Guardar registro.

`STR r0, [r1], #4`	Guardar r_0 en la dirección r_1. Luego incrementar r_1 en 4.
`STRB r2, [r1, #1]!`	Guardar los primeros 8 bits de r_2 en la dirección $r_1 + 1$. Luego incrementar r_1 en 1.
`LDRSH r2, [r1, r2, LSL #4]!`	Cargar (con signo) a r2 los 16 bits en la dirección $r1 + r2 \times 16$. Luego guardar esa dirección en $r1$.

2.5.2. Operaciones de carga y guardado múltiples

Estas operaciones tienen la forma:

$$\texttt{opcode}[modo] \quad r_n[!],\ \{lista\ de\ registros\}$$

El *opcode* puede ser:

- `LDM`, significa *Load Multiple*[13]. Esta operación carga la lista de registros con la información que hay en memoria en la dirección especificada por el primer operando.
- `STM`, significa *Store Multiple*[14]. Esta operación guarda la lista de registros en la dirección especificada por el primer operando.

El *modo* especifica cómo debe usarse la dirección provista en el primer operando. Tiene estas opciones:

- IA[15]: La dirección se incrementa luego de cargar/guardar cada registro
- IB[16]: La dirección se incrementa antes de cargar/guardar cada registro
- DA[17]: La dirección se decrementa luego de cargar/guardar cada registro
- DB[18]: La dirección se decrementa antes de cargar/guardar cada registro

Una pila puede ser ascendente o descendente, esto significa que los espacios de memoria pueden ir incrementandose cuando la pila aumenta de tamaño, o decrementandose, respectivamente. Normalmente existe un puntero de la pila, que apunta al final de la pila. Este puntero puede apuntar a la última posición llena o la primer posición vacía. Dependiendo de estas dos combinaciones (ascendente/descendente, llena/vacía), se deberán usar diferentes modos para cargar o guardar múltiples registros en una pila. Por ejemplo, si una pila es llena y descendiente, para guardar a pila se deberá usar `STMDB` y para cargar de la pila se deberá usar `LDMIA`.

Para facilitar el trabajo al programador, pueden usarse modos que toman este comportamiento distinto para `STM` y `LDM` automáticamente. A saber, estos son:

Modo	Significado	Modo en STM	Modo en LDM
FD	Full descending	DB	IA
FA	Full ascending	IB	DA
ED	Empty descending	DA	IB
EA	Empty ascending	IA	DB

[13] Cargar múltiple.
[14] Guardar múltiple.
[15] *Increment After* (incrementar después).
[16] *Increment Before* (incrementar antes).
[17] *Decrement After* (decrementar después).
[18] *Decrement Before* (decrementar antes).

Si el modo no se especifica, es por defecto IA. Si se escribe un signo de admiración (!) luego del registro de dirección, la última dirección calculada será escrita en el registro. Finalmente, la lista de registros puede especificarse de dos formas distintas:

$\{r_{n_1}, r_{n_2}, ..., r_{n_m}\}$	Es una lista de registros
$\{r_{n_m} - r_{n_{m+x}}\}$	Son todos los registros del n_m al n_{m+x}

Algunos ejemplos del uso de estas instrucciones:

- `LDMIA r1, r2,r7,r8`
  Cargar las 3 palabras empezando en la dirección r_1 en los registros r_2, r_7 y r_8, respectivamente. Incrementar la dirección luego de cada carga.
- `STMDB r0!, r1-r4`
  Guardar los registros r_1, r_2, r_3 y r_4 en memoria empezando en la dirección r_0. Decrementar la dirección antes de guardar cada registro. Escribir la última dirección en r_0.

Aclaración:

- En la instrucción `STM`, los registros se leen en orden lógico (`r0-r15`), no en el orden expresado en la línea de instrucción.
- En la instrucción `LDM` el orden de los registros tiene que ser ascendente.

Ejemplo:

```
.data
a:  .word 1
    .word 2
    .word 3
    .word 4

.text
.global main
main:
    ldr r9, =a
    ldmfd r9, {r1-r4}
    stmia r9, {r2,r1,r4,r3}
    bx lr
```

En la instrucción `stmia r9, {r2,r1,r4,r3}` carga el valor de los registros `r1-r4` en este orden a partir de la dirección de memoria `a` a pesar de que el orden indicado en la instrucción es otro.

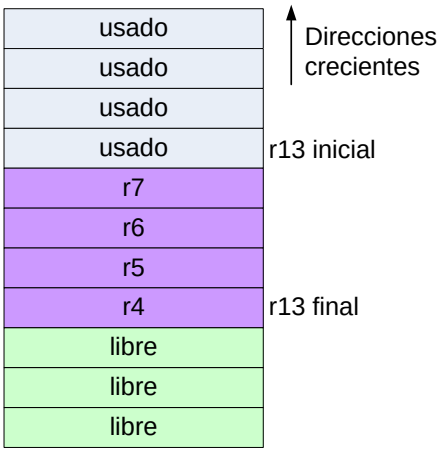
2.6. La pila

Como ya hemos visto en la arquitectura X86-64, es necesario almacenar el estado del procesador para realizar llamados a funciones anidadas. Las instrucciones de transferencia de datos múltiples vistas previamente proveen un mecanismo para almacenar dicho estado en la pila (apuntado por el registro r13). Recordar que las instrucciones STM y LDM tienen distintos modos de acceso. Sin embargo, para acceder a la pila el modo más común es el *full descending*.

Ejemplos:

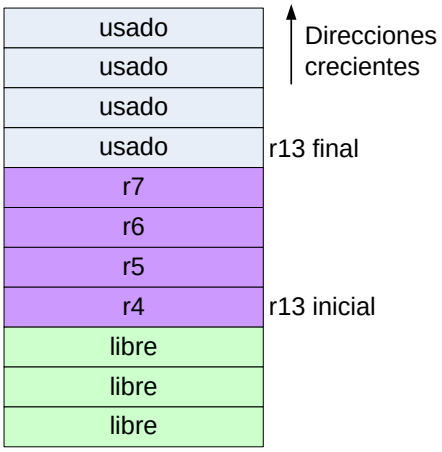
- `STMFD r13!, {r4-r7}`

  Apila r4, r5, r6 y r7 en la pila.



- `LDMFD r13!, {r4-r7}`

  Desapila r4, r5, r6 y r7 desde la pila.



3. Operaciones de punto flotante

Las primeras generaciones de procesadores ARM no tenían soporte para operaciones de punto flotante. Esto significaba hacer operaciones con enteros o utilizar un coprocesador externo para hacer operaciones de punto flotante.

Para versiones de la arquitectura ARM más recientes, la compañía creó una familia de coprocesadores de punto flotante que denominó VFP (*Vector Floating Point*)[19]. Estos coprocesadores vienen opcionalmente con algunos procesadores. Existen varias versiones de VFP, a saber:

[19] Punto flotante vectorizado

VFPv1	Obsoleta.
VFPv2	Extensión opcional para las arquitecturas ARMv5TE, ARMv5TEJ y ARMv6. Tiene 16 registros adicionales de 64 bits para punto flotante.
VFPv3-D32	Extensión opcional para las arquitecturas ARMv7-A y ARMv7-R. Tiene 32 registros de 64 bits para punto flotante.
VFPv3-D16	Extensión opcional para las arquitecturas ARMv7-A y ARMv7-R. Tiene 16 registros de 64 bits para punto flotante.
VFPv3-F16	Es una extensión equivalente a VFPv3-D32 y agrega instrucciones para hacer operaciones en media precisión (IEEE 754-2008).
VFPv4	Extensión opcional para las arquitecturas ARMv7. Tiene 32 registros de 64 bits para punto flotante. También provee operaciones de punto flotante de media precisión y operaciones de multiplicación y adición simultanea (del tipo $a \leftarrow a + b \times c$).

El tipo coprocesador de punto flotante está disponible en un registro especial de sólo lectura llamado *FPSID*.

Las operaciones de punto flotante utilizan registros especiales para hacer sus operaciones. Dependiendo de la versión de VFP utilizada puede haber de 16 a 32 registros de precisión doble para operaciones de punto flotante. Estos registros son referenciados como d_0 a d_{15} o d_{31} dependiendo de la cantidad existente en el procesador. Sin embargo, la mayoría de las operaciones pueden usarse con números de punto flotante de precisión simple (32 bits). Por lo tanto, los mismos registros de precisión doble pueden accederse como registros de precisión simple. Por cada registro de precisión doble d_n, pueden usarse dos registros de precisión simple s_{2n} y s_{2n+1} que corresponden a sus 32 bits más altos y más bajos. Si se usan registros de precisión simple y dobles que se corresponden en la misma operación se obtendrá un resultado incorrecto. Las operaciones con punto flotante de media precisión usan los 16 bits menos significativos o más significativos (dependiendo de la operación y sus parámetros) de los registros de presición simple.

3.1. FPSCR (Floating-Point Status Control Register)

Para poder alterar el control de flujo de un programa usando punto flotante, también existe un registro especial llamado *FPSCR* (Floating-point Status Control Register)[20]. Este registro funciona de manera parecida al *CSPR* pero sólo para operaciones de punto flotante. La única operación que modifica el contenido del *FPSCR* es la operación *FCMP*, que toma dos registros de punto flotante como parámetros.

La mayoría de las operaciones de punto flotante admiten un sufijo que controla su ejecución condicional. Estos sufijos son parecidos a los que utilizan las operaciones con enteros, pero con algunos cambios. Uno de los cambios es que no existen sufijos para comparaciones con signo o sin signo, ya que todos los números de punto flotante tienen signo. Otro cambio importante tiene que ver con el valor especial NaN (*Not a Number*)[21], que designa en las convenciones de punto flotante de la IEEE a los valores que no son válidos o son indefinidos, como el resultado de la operación 0/0. El valor NaN no puede ser ordenado con ningún otro valor, por lo cual se

[20]Registro de control de estado de punto flotante.

[21]No es un número.

considera que el resultado de una comparación donde uno de los dos operandos es NaN como *desordenado*.

La siguiente tabla explica los sufijos posibles para usar en operaciones de punto flotante:

Sufijo	Significado
EQ	Igual
NE	Diferente o desordenado
VS	Desordenado
VC	No desordenado
GE	Mayor o igual
LS	Menor o igual
GT	Mayor
CC / LO / MI	Menor
CS / HS / PL	Mayor o igual, o desordenado
LE	Menor o igual, o desordenado
HI	Mayor o desordenado
LT	Menor o desordenado
AL	Siempre válido

3.2. Load/Store

La instrucción de carga de un número de punto flotante desde memoria a un registro de punto flotante es $VLDR$. La instrucción de guardado de un número de punto flotante en un registro de punto flotante a memoria es $VSTR$. El uso de estas instrucciones tienen las siguientes forma:

$$\texttt{opcode}[cond][.tipo]\ F_d\,,\ [r_n\{,\ \#offset\}]$$

$$\texttt{opcode}[cond][.tipo]\ F_d\,,\ etiqueta$$

En este caso *opcode* puede ser la operación de carga o de guardado. El *tipo* indica que precisión usa el registro a utilizar. Para números de punto flotante de doble precisión deberá ser $F64$, para precisión simple deberá ser $F32$, y para media precisión deberá especificarse como $F16$. El registro r_n debe contener una dirección a memoria a la que se le sumará *offset* para determinar la dirección en memoria para guardar/cargar el registro. También puede utilizarse una etiqueta para determinar la dirección de memoria a utilizar.

Para guardar o cargar más de un registro, pueden usarse las operaciones de guardado o carga múltiple, $VLDM$ y $VSTM$. La forma que toman estas operaciones es:

$$\texttt{opcode}\{modo\}[cond]\ R_n[\!]\,, Lista\,de\,registros$$

donde *opcode* será el nombre de la operación, *modo* es el modo de la pila (como fue descripto para las operaciones de carga/guardado de registros de propósito general) y la lista de registros simplemente será una lista separada por comas de los registros a guardar o cargar, entre corchetes.

Existen operaciones de *pop* y *push* para la pila que tienen las siguientes equivalencias:

$$\texttt{VPOP}[cond]\ Lista\,de\,registros \equiv \texttt{VLDMIA}[cond]\ sp!\,, Lista\,de\,registros$$

$$\texttt{VPUSH}[cond]\ Lista\,de\,registros \equiv \texttt{VSTMDB}[cond]\ sp!\,, Lista\,de\,registros$$

3.3. Instrucciones de conversión entre enteros y punto flotante

La instrucción de conversión entre enteros de 32 bits y números de punto flotante es *VCVT*. Esta instrucción toma como operandos registros de punto flotante. Una instrucción utilizando *VCVT* puede tener las siguientes formas:

$$\texttt{VCTV}\{modo\}\{cond\}\{.tipo\}.F64\ S_d,\ D_m$$
$$\texttt{VCTV}\{modo\}\{cond\}\{.tipo\}.F32\ S_d,\ S_m$$

$$\texttt{VCTV}\{cond\}.F64\{.tipo\}\ D_d,\ S_m$$
$$\texttt{VCTV}\{cond\}.F32\{.tipo\}\ S_d,\ S_m$$

donde *cond* es un sufijo de ejecución condicional para operaciones de punto flotante. El *tipo* puede ser *S32* para enteros con signo o *U32* para enteros sin signo. Las primeras dos instrucciones convierten un número de punto flotante a un entero. Las segundas dos operaciones convierten un entero a un número de punto flotante. El *modo* indica como se realizará el redondeo para la conversión. Las opciones son:

Código	Significado
A	Redondeo al entero más cercano, 0.5 va al cero
N	Redondeo al entero más cercano, 0.5 va al número par más cercano
P	Redondeo al infinito
M	Redondeo al infinito negativo
R	Usar el modo indicado en el FPSCR

3.4. Instrucciones de conversión de precisión

Las instrucciones de conversión de precisión son las siguientes:

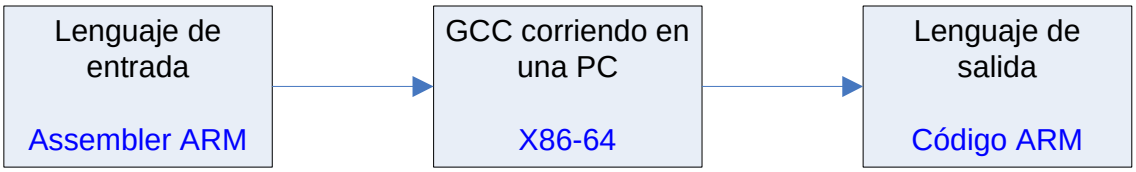
Instrucción	Significado
VCVT{cond}.F64.F32 D_d, S_m	Conversión de precisión simple a doble
VCVT{cond}.F32.F64 S_d, D_m	Conversión de precisión doble a simple
VCVT{lado}{cond}.F32.F16 S_d, S_m	Conversión de precisión simple a media
VCVT{lado}{cond}.F16.F32 S_d, S_m	Conversión de precisión media a simple

donde *cond* es un sufijo de ejecución condicional para operaciones de punto flotante. Para conversiones donde interviene un número de precisión media, el sufijo *lado* puede ser T para indicar que se usan los 16 bits más significativos o B para indicar que se usan los 16 bits menos significativos.

4. Compilación

Al momento de querer probar código escrito para la arquitectura ARM es muy probable que no contemos con una computadora con arquitectura ARM. Por este motivo, es necesario utilizar un emulador de esta arquitectura que nos permita trabajar como si tuviéramos una máquina con arquitectura ARM. En efecto, un emulador es un software que permite ejecutar programas en una arquitectura de hardware diferente de aquella para la cual fueron escritos originalmente. La compilación de código fuente, que realizada bajo una determinada arquitectura genera código ejecutable para una arquitectura diferente, se denomina **compilación cruzada**.

Una posibilidad es utilizar el emulador *QEMU*. El siguiente esquema ilustra el proceso de emulación:



A continuación se muestra un ejemplo de cómo compilar y ejecutar un código `p.s` escrito en ARM:

```
arm-linux-gnueabi-gcc -static p.s
```

Compila el programa `p.s` escrito en ARM. La opción `-static` es necesaria para compilar de forma estática.

```
qemu-arm ./a.out
```

Ejecuta el archivo binario usando el emulador *Qemu*.

Sin embargo, al igual que lo que ocurre en la arquitectura X86-64, escribir todo el programa en ensamblador no es la mejor opción. Es mejor escribir solo la parte que necesariamente debe ser escrita en ensamblador (con fines de optimizar, acceder al hardware, etc). Por ello, podemos mezclar código C con ensamblador siempre y cuando se respete la convención de llamada.

Vemos como ejemplo un programa básico que realice la suma de dos números y muestre el resultado por pantalla. Por un lado podemos tener un archivo `suma.s` escrito en ARM:

```
.text
.global suma

@ Argumentos: r0 y r1
@ Retorna: r0

suma:

     add r0, r0, r1

     bx  lr
```

y por otro lado un archivo `main.c` que invoque a `suma` y luego imprima el resultado por pantalla:

```
#include<stdio.h>

int suma(int a, int b);

int main()
{
     int x=2, y=3;
     printf("La suma es: %d\n", suma(x, y));
     return 0;
}
```

Luego podemos compilar de la siguiente manera:

```
arm-linux-gnueabi-gcc -static -marm -o suma main.c suma.s
```

y ejecutar:

```
qemu-arm suma
```

para finalmente obtener:

```
La suma es: 5
```

Notar que para compilar se han utilizado opciones `-static` y `-marm`. La opción `-static` es necesaria para forzar un *static linking* mientras que la opción `-marm` ha sido utilizada para forzar el modo el modo ARM en lugar del modo THUMB.

5. Depuración

Para depurar un programa (*debuggear*) ver el documento **Guía de desarrollo para otras arquitecturas** en la Sección **Apuntes propios de la asignatura** del Campus Virtual.

Referencias

[1] Andrew Sloss, ARM System Developer's Guide, 2004

[2] William Hohl, *ARM Assembly Language: Fundamentals and Techniques*, 2009

[3] ARM Holdings, Documentación oficial online de ARM en http://infocenter.arm.com

[4] ARM Architecture Reference Manual, 2005

[5] Professional Embedded ARM Development, James A. Langbridge, John Wiley & Sons, Inc., 2014.