Conceptos básicos del ensamblador y arquitectura ARM

Diego Feroldi

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

*Actualizado 23/04/2024 (D. Feroldi, feroldi@fceia.unr.edu.ar)

Índice

Notas generales:

- Este apunte de clases reseña las principales características de la arquitectura ARM y de su lenguaje ensamblador, cubriendo hasta la versión *ARMv7-A*.
- Este apunte no es para nada una referencia completa del lenguaje ensamblador ni de la arquitectura sino que debe ser utilizado como material complementario con lo visto en las clases teóricas. Para una información más detallada consultar las referencias. En particular, consultar [7] para una información más detallada sobre el uso de las instrucciones.
- Para poder compilar y depurar los ejemplos que se muestran en el apunte, ver las Secciones A y B.
- Además de los ejemplos presentados en este apunte hay una cantidad considerable de ejemplos en la Sección **Material de estudio/Código** del Campus Virtual de la asignatura.

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. Entre las principales aplicaciones se encuentran las siguientes:

- Mobile Phones, PDAs, Tablets
- Portable Games Consoles
- Portable Media Players, Camcorders
- GPS Navigation Systems
- Set Top Boxes, TVs, Hard Discs, Routers, ...

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

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 tipo Von Neumann de arquitectura ARM, la cual comparte señales y memoria para código y datos.

1.1. Modo ARM y modo Thumb

Las versiones de arquitectura ARMv4T y superiores definen un conjunto de instrucciones de 16 bits llamado conjunto de instrucciones Thumb. La funcionalidad del conjunto de instrucciones Thumb es un subconjunto de la funcionalidad del conjunto de instrucciones ARM de 32 bits. Un procesador que está ejecutando instrucciones Thumb está funcionando en estado o modo Thumb. Un procesador que ejecuta instrucciones ARM está funcionando en estado o modo ARM. Un procesador en estado ARM no puede ejecutar instrucciones Thumb y un procesador en estado Thumb no puede ejecutar instrucciones ARM. Debe asegurarse de que el procesador nunca reciba instrucciones del conjunto de instrucciones incorrecto para el estado actual. Cada conjunto de instrucciones incluye instrucciones para cambiar el estado del procesador. Los procesadores ARM siempre comienzan a ejecutar código en estado ARM.

Los contenidos de este apunte están orientados a trabajar exclusivamente en el modo ARM. Si se necesita trabajar en modo Thumb se puede consultar la documentación en [4].

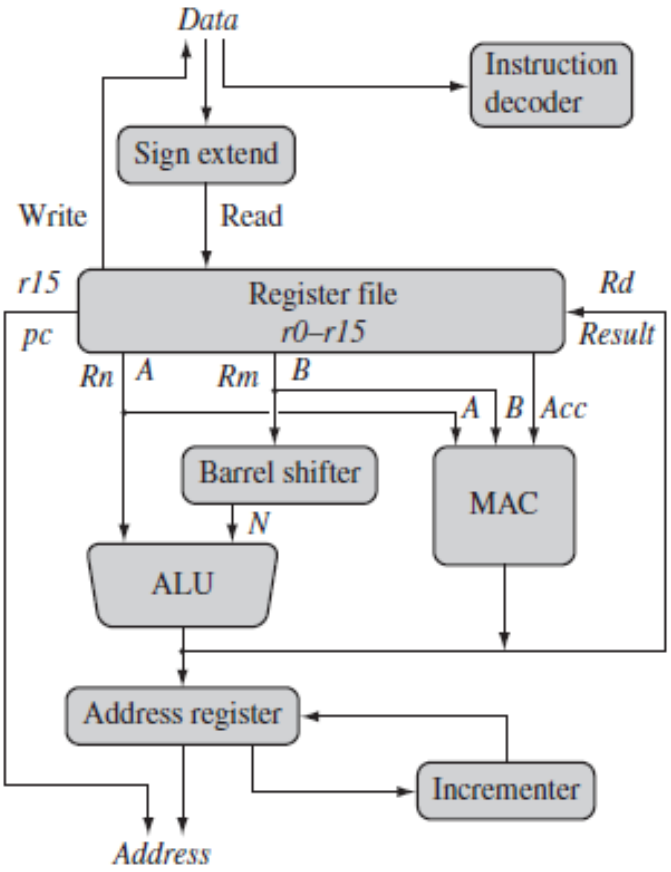


Figura 1: Arquitectura ARM tipo Von Neumann [1].

1.2. Registros

La arquitecturas ARM utiliza 16 registros de 32 bits para los programas en modo usuario, llamados `r0` a `r15`, más un registro de banderas llamado CPSR (*Current Program Status Register*). El CPSR es el registro que almacena el estado del programa actual y será visto en detalle en la Sección 1.4. 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*[1]). Dicha convención establece las siguientes reglas para llamadas a procedimientos:

- `r0` hasta `r3` (a veces también llamados `a1` hasta `a4`) son usados para pasar argumentos. Si no alcanzan estos 4 registros, el resto de los argumentos se pasa utilizando la pila.
- `r0` también se utiliza como valor de retorno de una función. Si el valor de retorno tiene más de 32 bits de ancho, también se usa `r1`. Si no alcanzan estos 2 registros, se devolverá `r0` como un puntero al lugar donde está almacenado el resultado.
- `r4` a `r11`, también llamados `v1` a `v8`, pueden ser usados como variables de propósito general.
- `r9` en algunas arquitecturas tiene un uso especial.
- `r0` a `r3` son *caller-save*.

[1] Convención de llamadas de procedimiento de la arquitectura ARM.

- `r4` a `r11` son *callee-save*.
- `r12`, llamado también `ip` (*Intra-procedure-call scratch register*). Es un registro de propósito general pero a veces es usado por el linker si es necesario para hacer saltos largos y poder usar todo el espacio de memoria de 32 bits. Su uso depende del sistema operativo.
- `r13`, llamado también `sp`, se usa como *Stack Pointer*. Al igual que los otros registros, es posible leer y escribir en este registro, pero la mayoría de las instrucciones dedicadas cambiarán el puntero de la pila según sea necesario como se verá en detalle en la Sección 6.
- `r14`, llamado también `lr` (*Link Register*), guarda la dirección de retorno. Este contiene la dirección de memoria de la instrucción que se ejecutará cuando se complete una subrutina. En efecto, contiene la dirección de memoria a la que volver una vez que termine la subrutina. Cuando el procesador encuentra una instrucción de ramificación con enlace (BL), el registro `r14` se carga con la dirección de la siguiente instrucción. Cuando finaliza la rutina, mediante la ejecución de BX se vuelve a donde estaba el programa.
- `r15`, llamado también `pc`, es el *Program Counter*. Este registro contiene la dirección de memoria de la siguiente instrucción que se obtendrá de la memoria.
- Para las arquitecturas con VFP[2], los argumentos de punto flotante se pasan, y el valor de retorno se devuelve, en los registros de punto flotante `s0` a `s15`, que son *caller-save*. Los registros `s16` a `s31` son *callee-save*.

La siguiente Tabla resume el uso y función de los registros, como así también si es preservado a través de llamadas a funciones:

Nombre	Alias	Uso	¿Preservado?
`r0`	`a1`	Argumento / valor de retorno / variable temporal	No
`r1-r3`	`a2-a4`	Argumentos / variables temporales	No
`r4-r11`	`v1-v8`	Variables salvadas	Sí
`r12`	—	Variable temporal	No
`r13`	`SP`	*Stack Pointer*	Sí
`r14`	`LR`	*Link Register*	Sí
`r15`	`PC`	*Program Counter*	—

1.3. 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` *resulta* `E=1` *(bit 9), por lo tanto opera en modo big-endian.*
- `SETEND LE`
  En el `CPSR` *resulta* `E=0` *(bit 9), por lo tanto opera en modo little-endian.*

[2] *Vector Floating Point* (Punto flotante vectorizado).

1.4. CPSR (Current Program Status Register)

El registro de status (*Current Program Status Register*, `CPSR`)[3] guarda el estado actual del procesador. Contiene banderas de condición que pueden actualizarse cuando se produce una operación en la ALU. Las instrucciones de comparación actualizan automáticamente el registro `CPSR`. La mayoría de las otras instrucciones no actualizan automáticamente el `CPSR`, pero se puede forzar a hacerlo agregando la directiva *S* después de la instrucción. La Fig. 2 muestra el contenido de algunos de sus bits.

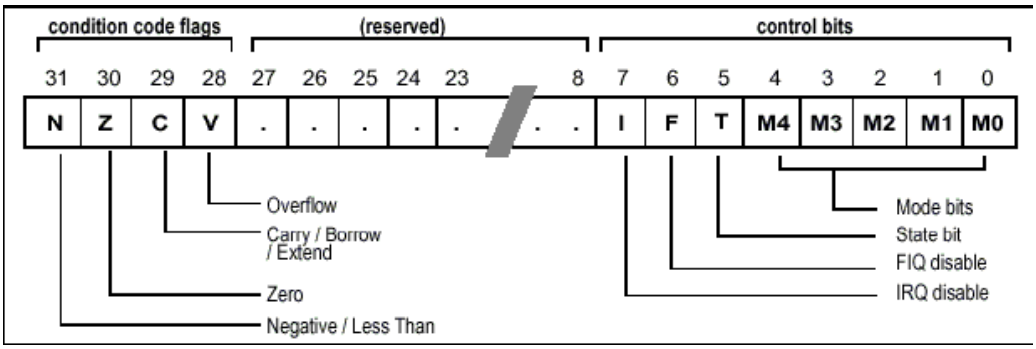


Figura 2: Estructura del registro `CPSR`.

El significado de los bits del CPSR es el siguiente:

- 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 en modo THUMB.
- 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*. Se establece en 1 cuando la operación da como resultado un acarreo, o cuando una resta no produce un acarreo; en caso contrario, se borra a 0.
- 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.

De particular importancia resultan los bits 28 a 31 para implementar ejecución condicional como se verá en la Sección 3.

[3]También llamado APSR (*Application Program Status Register*)

1.5. Directivas

A continuación se presentan algunas de las principales directivas usadas en GNU Assembler para ARM:

`.ascii "<string>"`	Inserta una cadena de caracteres.
`.asciz "<string>"`	Inserta una cadena de caracteres terminada con carácter nulo.
`.string "<string>"`	Similar a `.asciz`.
`.byte <byte1> {,<byte2>} ...`	Inserta una lista de valores tipo `char` (1 byte)[4].
`.short <short1> {,<short2>} ...`	Inserta una lista de valores tipo `short` (2 bytes).
`.word <word1> {,<word2>} ...`	Inserta una lista de valores tipo `int` (4 bytes).
`.long <long1> {,<long2>} ...`	Inserta una lista de valores tipo `int` (4 bytes). Equivalente a `.word`.
`.quad <quad1> {,<quad2>} ...`	Inserta una lista de valores tipo `long int` (8 bytes).
`.float <float1> {,<float2>} ...`	Inserta una lista de valores tipo `float` (4 bytes).
`.double <double1> {,<double2>} ...`	Inserta una lista de valores tipo `double` (8 bytes).
`.space <numero_bytes> {,<valor>} ...`	Reserva el número de bytes indicado y lo rellena con ceros (por defecto) o con el valor indicado.
`.global <símbolo>`	Declara un símbolo como global.
`.data`	Indica que lo está a continuación que va en el segmento de datos.
`.text`	Indica que lo está a continuación que va en el segmento de código.

Ejemplos

```
.data
str: .asciz "Hola mundo"
a: .word 1, 2, 3, 4
f: .float 3.14
h: .space 10, 0xff

.text
.global main
main:
    ......
    ......
```

1.6. Pipeline

Pipeline es una técnica utilizada en el diseño de procesadores ARM (y otros) para aumentar el rendimiento de las instrucciones. En lugar de tener que buscar una instrucción, decodificarla

[3]Los tipos numéricos se refieren al lenguaje C.

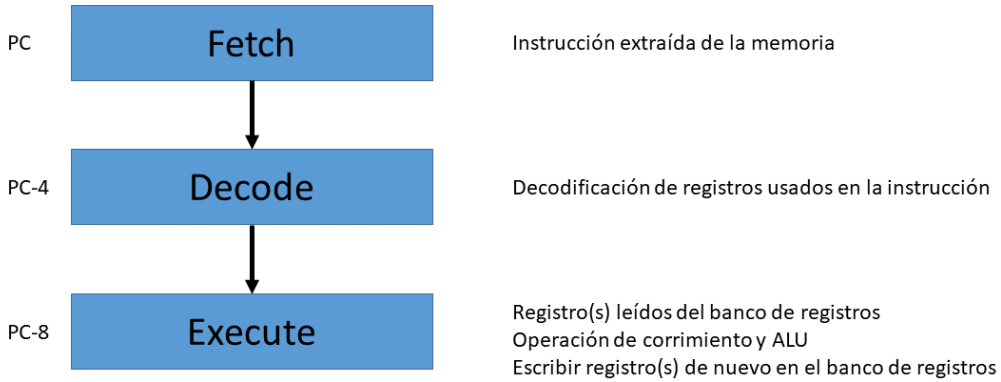


Figura 3: *Pipeline* típico de tres etapas.

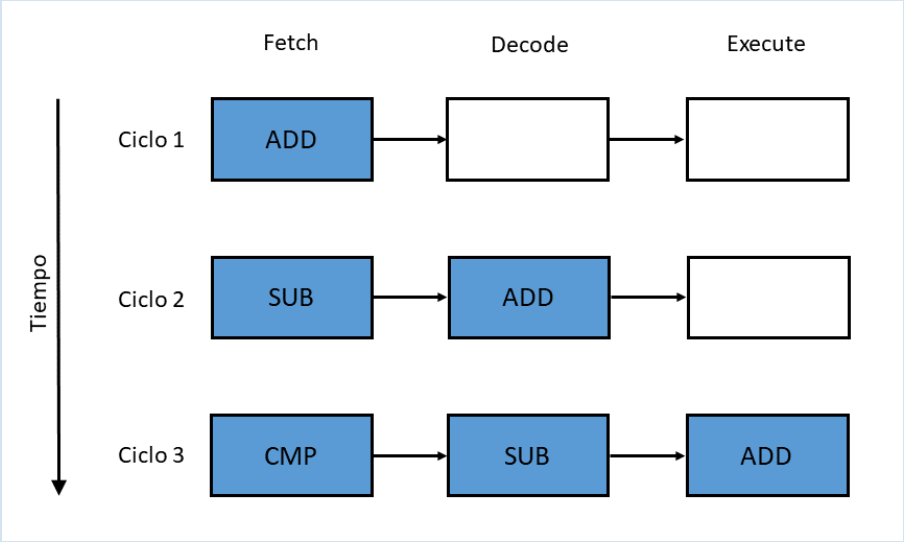
y luego ejecutarla, se pueden hacer las tres tareas al mismo tiempo pero no sobre la misma instrucción. Este concepto es análogo a la producción en serie en una fábrica: mientras un operario está ejecutando una tarea sobre una pieza, otros operarios van ejecutando otras tareas sobre otras piezas en el mismo momento. La ventaja es el aumento de la velocidad. Sin embargo, existen desventajas de un sistema de *pipeline*, en particular las paradas o bloqueos. La Fig. 3 muestra un *pipeline* típico de 3 etapas.

La dirección de la instrucción que se está obteniendo (no la que se está ejecutando) está contenida en el contador de programa (`pc`). Por lo tanto, la instrucción que se está decodificando es la que que corresponde a `pc-4` y la que se está ejecutando a `pc-8`, dado que las instrucciones en ARM son de longitud fija 4 bytes. La etapa 2 accede a los operandos del banco de registros que sean necesarios. Una vez realizado el cálculo en la etapa 3, se vuelven a escribir los resultados en el banco de registros.

Ejemplo

```
ADD r5, r1
SUB r8, r3
CMP r0, #3
```

En este ejemplo cada instrucción se ejecuta secuencialmente. La primera instrucción suma el contenido de los registros `r1` *y* `r5`*. La segunda resta el contenido de* `r3` *a* `r8`*. La tercera compara el contenido del registro* `r0` *con el valor 3. En el primer ciclo se realiza la etapa fetch de la primera instrucción. En el segundo ciclo se realiza la etapa decode de la priemra instrucción y la etapa fetch de la segunda instrucción. Finalmente, en el tercer ciclo se realiza la etapa execute de la primera instrucción, la etapa decode de la segunda y la etapa fetch de la tercera.*



En este ejemplo simple ninguna instrucción depende de resultados de las instrucciones anteriores, por lo que no se producen bloqueos. Sin embargo, en general no todas las instrucciones son independientes entre sí. La salida de una instrucción puede ser la entrada a otra instrucción. Por ejemplo, en el siguiente código la instrucción `SUB r8, r5` *depende del valor de* `r5` *que se calcula en la instrucción anterior:*

```
ADD r5, r1
SUB r8, r5
CMP r0, #3
```

En tales casos, es necesario detener el pipeline para que la instrucción anterior haya completado la ejecución.

Se produce un bloqueo o parada cuando un *pipeline* no puede continuar funcionando normalmente. Una de las principales razones de los bloqueos son los saltos o bifurcaciones. Cuando se produce una bifurcación, el *pipeline* debe llenarse con nuevas instrucciones, que probablemente estén en una ubicación de memoria diferente. Por lo tanto, el procesador necesita buscar una nueva ubicación de memoria, colocar la primera instrucción al comienzo del *pipeline* y luego comenzar a trabajar en la instrucción. Mientras tanto, la fase de "ejecución" tiene que esperar a que lleguen las instrucciones. Para evitar esto, algunos procesadores ARM tienen hardware de predicción de ramas, que "predice" de manera efectiva el resultado de un salto condicional. El predictor de rama luego se llena en el *pipeline* con el resultado predicho. Si es correcto, se evita un bloqueo porque las instrucciones ya están presentes en el *pipeline*.

Hay varios casos en los que el orden de las instrucciones puede provocar bloqueos. En el ejemplo anterior, no se requería el resultado de una recuperación de memoria, pero ¿qué pasaría si la instrucción inmediatamente posterior requiriera ese resultado? La optimización del *pipeline* no podría contrarrestar el bloqueo, y el *pipeline* podría detenerse durante un período de tiempo significativo. La respuesta a esto es reorganizar el orden de las instrucciones para evitar atascos. Si una operación de memoria puede detener un *pipeline*, el compilador puede colocar la instrucción antes, si es posible, dando así al *pipeline* un poco más de tiempo.

2. Instrucciones ARM

El conjunto de instrucciones en ARM se mantiene lo suficientemente pequeño para que el hardware requerido para decodificar la instrucción y sus operandos pueda ser simple, pequeño y rápido. Las operaciones más elaboradas (que son menos comunes) se realizan usando secuencias de múltiples instrucciones simples. De esta manera, el compilador o programador sintetiza operaciones complicadas (por ejemplo, una operación de división) combinando varias instrucciones simples. Por lo tanto, ARM es una arquitectura de computadora con conjunto de instrucciones reducido (RISC). Las arquitecturas con muchas instrucciones complejas, como la arquitectura x86 de Intel, son computadoras con conjunto de instrucciones complejas (CISC). Por ejemplo, x86 define una instrucción de "movimiento de cadena" que copia una cadena (una serie de caracteres) de una parte de la memoria a otra. Tal operación requiere muchas instrucciones simples en una máquina RISC. Sin embargo, el costo de implementar instrucciones complejas en una arquitectura CISC implica hardware adicional y sobrecarga que ralentiza las instrucciones simples.

Cada instrucción en ARM tiene una longitud fija para permitir que el *pipeline* obtenga instrucciones futuras antes de decodificar la instrucción actual. Por el contrario, en los procesadores CISC las instrucciones suelen ser de tamaño variable y requieren muchos ciclos para ejecutarse. Las instrucciones ARM comúnmente tienen dos o tres operando. En general, las operaciones lógicas, aritméticas o de comparación pueden tener la siguiente forma:

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

- El `opcode` es un nombre de 3 letras (generalmente) 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 3.
- 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 operando.
- *operando2* es el segundo operando (opcional dependiendo de la instrucción), que puede ser un registro o un valor inmediato. Este segundo operando puede también especificar una operación de desplazamiento o rotación que se aplicará antes de ser usado para la operación. Esta operación adicional se explicará en la Sección 4.

2.1. Instrucciones de movimiento

Las instrucciones de movimiento se utilizan para transferir datos a registros, ya sea los datos de otro registro (copiando de un registro a otro), o cargando datos estáticos en forma de valor inmediato. Algunos ejemplos de instrucciones de movimiento:

- `MOV` *(Move)* copia desde el operado fuente al operando destino.

`MOV r0, #0`
Copiar el valor 0 a r0.

`MOV r7, r5`
Copiar el valor de r5 a r7.

- `MVN` *(Move Negated)* copia un valor negado en un registro. Esto es útil para almacenar algunos números que `MOV` no puede manejar, números que no se pueden expresar como un valor inmediato y para mapas de bits que se componen principalmente de 1s.

```
MVN r1, r0
r1 = NOT(r0)
```

- `MOVW` (*Move Wide*) copia una constante de 16 bits en un registro mientras pone a cero los 16 bits superiores del registro objetivo. `MOVW` puede usar cualquier valor que se pueda expresar como un número de 16 bits.

```
MOVW r0, #0x1234
```
`r0` ahora contiene el valor `0x00001234`, sin importar lo que hubiera antes en `r0`.

- `MOVT` (*Move Top*) copia una constante de 16 bits en la parte superior de un registro, dejando intacta la mitad inferior. Esta instrucción y la instucción `MOVW` están disponibles en núcleos ARMv7 y superiores.

```
MOVW r0, #0xface
MOVT r0, #0xfeed
```
El resultado será `r0=0xfeedface`.

2.2. Instrucciones aritméticas

Las instrucciones aritméticas son la base de cualquier unidad central de procesamiento (CPU). Las instrucciones aritméticas pueden realizar la mayoría de las instrucciones matemáticas básicas, pero hay algunas excepciones. Los núcleos ARM pueden sumar, restar y multiplicar. Algunos núcleos ARM no tienen división por *hardware*, pero, por supuesto, hay otras formas de hacerlo.

Todas las instrucciones aritméticas funcionan directamente desde y hacia registros solamente. Es decir, no pueden leer de la memoria principal o incluso de la memoria caché. Por lo tanto, para realizar cálculos, los datos deben leerse previamente en registros.

Las instrucciones aritméticas pueden funcionar directamente con números sin signo y con signo, usando la notación en complemento a dos.

La siguiente es una lista de algunas de las instrucciones matemáticas incluidas en muchos núcleos ARM.

- `ADD` (Suma)

`ADD r0, #5`
Suma el contenido de `r0` con el valor inmediato 5 y guarda el resultado en `r0`.

`ADD r0, r1`
Suma el contenido de `r1` y `r0` y guarda el resultado en `r0`.

`ADD r0, r1, r2`
Suma el contenido de `r1` y `r2` y guarda el resultado en `r0`.

- `ADC` (Suma con carry)

`ADC r0, r1, r2`
Suma con acarreo, `r0:=r1+r2+carry`.

- `SUB` (Resta)

```
SUB r5, r3, r0
r5 := r3 - r0
```

- `SBC` (Resta con carry)

`SBC r5, r3, r0`
Resta con acarreo, `r5:=r3-r0-NOT(carry)`.

- `RSB` (Resta inversa)

`RSB r0, r0, r1`
Resta inversa, `r0:=r1-r0`.

- `MUL` (Multiplicación con signo)

`MUL r0, r1, r2`
Multiplica `r1` con `r2` y guarda el resultado en `r0`.

- `UMULL` (Multiplicación sin signo)

`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`×`r3`

Observación

Notar que estas instrucciones NO modifican las banderas del CPSR a menos que se agregue el sufijo S*:*

```
MOV r1, #1          @ r1 := 1
MVN r2, #0          @ r2 := 0xffffffff
ADDS r0, r1, r2     @ r0 := 0, CF=1
```

El resultado es correcto si los valores son interpretados como números con signo:

$$1 + (-1) = 0$$

En cambio, no es correcto si se interpretan como números sin signo:

$$1 + 4.294.967.295 \neq 0$$

Por ese motivo, `OF=0` *y* `CF=1`*. Notar que si no hubiéramos agregado el sufijo S la bandera* `CF` *no se hubiera encendido.*

2.3. Instrucciones lógicas

Los operadores lógicos realizan operaciones bit a bit entre dos números. A continuación se presentan algunas de las instrucciones disponibles:

- Operación AND bit a bit

```
MOV r7, #0xff
MOV r2, #0xf
AND r8, r7, r2        @ r8 = 0xf
```

- Operación OR bit a bit

```
MOV r11, #0x16
ORR r11, r11, #1      @ r11 = 0x17
```

- Operación *bit clear* (`BIC`)

```
MOV r11, #0x16
BIC r11, r11, #2      @ r11 = 0x14 (b1=0)
```

- Operación OR exclusiva bit a bit

```
MOV r11, #0x16
EOR r11, r11, #1      @ r11 = 0x17
```

2.4. Instrucciones de comparación

Las instrucciones de comparación son instrucciones que no devuelven ningún resultado, pero las banderas de condición del CPSR se actualizan. Son extremadamente útiles, ya que permiten al programador hacer comparaciones sin usar un nuevo registro. El CPSR se actualiza automáticamente y no es necesario especificar el sufijo *S*.

La siguiente es una lista de instrucciones de comparación utilizadas en procesadores ARM:

- `CMP` es la instrucción utilizada para comparar dos números. Lo hace restando uno del otro y actualizando las banderas de estado de acuerdo con el resultado.

`CMP r0, #3`

Compara `r0` con 3 (calcula `r0-3` y modifica las banderas del CPSR pero no modifica `r0`).

`CMP r2, r1`

Compara `r2` con `r1` (calcula `r2-r1` y modifica las banderas del CPSR pero no modifica `r2`).

- `CMN` Comparación negativa. Esta instrucción es en realidad la inversa de `CMP`.

`CMN r2, #42`

Compara `r2` to -42.

- `TST` es una instrucción que prueba si uno o más bits de un registro están limpios o si al menos un bit está encendido. No hay salida para esta instrucción. En su lugar, se actualizan las banderas de condición de CPSR. Este es el equivalente de `operando1 AND operando2`.

`TST r11, #1`

Testea el bit cero.

- `TEQ` compara *operando1* y *operando2* usando una instrucción `OR` exclusivo bit a bit y prueba la igualdad, actualizando el CPSR. Es el equivalente a una instrucción `EORS`, excepto que el resultado se descarta. Esto es especialmente útil cuando se compara un registro y un valor, devuelve cero cuando los registros son idénticos y devuelve 1 para cada bit que es diferente.

`TEQ r8, r9`

Testea si `r8` es igual a `r9`.

2.5. 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.

- `B` (*Branch*)

  `B{cond} label`

  Le dice al contador del programa actual que la siguiente instrucción estará en la dirección `<label>`. Luego el valor del contador de programa vale `r15 := label`. Este es un salto permanente y no es posible el retorno. Se utiliza principalmente en bucles o para dar control a otra parte del programa.

  Ejemplo

```
        B   forward         @ Salta a la etiqueta 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* `ADD` *nunca se ejecutan.*

- `BL` (*Branch with Link*)

  `BL{cond} label`

  Salta de la misma forma que la instrucción `B` pero luego `r14 :=` dirección de la próxima instrucción y `r15 := label`. Es decir, el `pc` se actualizará con la dirección especificada, y además la dirección inmediatamente después de la instrucción `BL` se colocará en `r14` (*link register*). Esto permite que el programa retorne a donde estaba cuando finalice la subrutina.

  Ejemplo

```
    [ ... ]
    [ ... ]
    BL calc
    [ ... ]         @ Próxima instrucción
    [ ... ]
calc:
    ADD r0, r1, r2
    BX lr           @ Regresa a la siguiente instrucción luego de BL calc
```

  En este ejemplo, durante la aplicación principal, se ramifica con un enlace a `calc`*. Una vez realizado el cálculo, puede volver al programa principal mediante una instrucción* `BX`*.*

- `BX` (*Branch and optionally Exchange*)

  ```
  BX{cond} Rm
  ```

  La instrucción `BX` provoca una bifurcación a una dirección especificada por un registro. Además, `BX` es una instrucción que permite al programa cambiar entre el modo ARM y el modo THUMB, para núcleos que admitan ambos estados. Esto permite una integración perfecta de código ARM y código THUMB porque el cambio se realiza en una sola instrucción. Por lo tanto, luego de ejecutarse la instrucción se actualiza el contador de programa (`r15 := Rm`) y además cambia a modo THUMB si así es requerido o se mantiene en modo ARM en caso contrario (ver [4] para una descripción más detallada).

- `BLX` (*Branch with Link and optionally Exchange*)

  ```
  BLX label     @ r14 := dirección de la siguiente instrucción
                @ r15 := label
                @ Cambia a modo Thumb
  ```

  `BLX` es similar a la instrucción `BX`. Esta instrucción también cambia desde y hacia el modo Thumb, pero también actualiza el registro de enlace, lo que permite volver a la ubicación actual.

Observación

¿Cómo volvemos de la subrutina que invocó `BL`*?*

En principio para retornar de la subrutina basta con modificar el `pc` *usando la dirección de retorno que tenemos guardada en el link register:*

```
MOV pc, r14
```

Sin embargo, es más seguro retornar utilizando la instrucción `BX` *(disponible en ARMv4T o posterior) en códigos donde se mezcle modo ARM y modo Thumb:*

```
BX r14
```

Asimismo, se puede usar el alias para `r14`*, por lo tanto podemos escribir:*

```
BX lr
```

2.6. Instrucciones de carga y guardado en memoria

La arquitectura ARM es una arquitectura de tipo *Load/Store*. Esto quiere decir que para cualquier operación aritmética o lógica los operandos siempre serán registros o valores inmediatos. Por lo tanto, no se pueden hacer cálculos directamente desde la memoria del sistema. Es decir, para hacer un cálculo con valores en memoria previamente hay que cargar dichos valores en registros. Cuando terminamos de hacer los cálculos, podemos almacenar los resultados en memoria.

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. Para cargar datos de memoria en registros usamos la instrucción `ldr`. Por lo general, es un proceso de dos pasos. Primero cargamos la dirección de los datos que queremos, luego cargamos los datos en sí. Necesitamos este proceso de dos pasos porque la dirección de memoria desde la que queremos cargar es un valor de 32 bits; no se puede incluir dentro de la instrucción porque es demasiado grande. La dirección debe cargarse en un registro que pueda contener el valor completo de 32 bits:

$$\boxed{\texttt{ldr} \quad r_d, \texttt{=label}}$$

carga r_d con la dirección que corresponde al dato etiquetado con `label`[5]. Esto obtiene la dirección de los datos, no los datos en sí. Por lo tanto, es similar al operador `&` en C/C++. Luego:

$$\boxed{\texttt{ldr} \quad r_d, [r_n]}$$

carga r_d con la palabra de memoria en la ubicación almacenada en r_n . Esto carga los datos reales apuntados por el registro. Esto es similar al operador `*` en C/C++.

Ejemplo

```
.data
x:  .word   0x12345678

.text
.global main
main:
    ldr   r1, =x        @ r1 <-- dirección de x (int* r1 = &x; in C)

    ldr   r2, [r1]      @ r2 <-- 0x12345678 (int r2 = *r1; in C)

    bx lr
```

Observación

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.

2.6.1. Instrucciones de carga y guardado simples

Las instrucciones de carga y guardado simples tienen la forma:

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

El *opcode* puede ser:

- `LDR`, significa *Load Register* (Cargar registro). La operación será:
  $r_d \leftarrow$ valor en *dirección*
- `STR`, significa *Store Register* (Guardar registro). La operación será:
  valor en *dirección* $\leftarrow r_d$

[5]El acrónimo `LDR` también refiere a una pseudo-instrucción que será vista en detalle en la Sección 2.7.

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 se puede especificar de las siguientes formas:

Modo	**Descripción**
[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 $r_n + (r_m \ll I)$. En lugar de LSR puede usarse también ASR, ROR, RRX.
Etiqueta	La dirección la da una etiqueta en el propio segmento de código (`.text`).

Ejemplos

`STR r0, [r1, #4]`	*Se guarda* `r0` *en la dirección* `r1+4`.
`STR r0, [r1], #4`	*Se guarda* `r0` *en la dirección* `r1` *y luego se incrementa* r_1 *en 4.*
`STRB r2, [r1, #1]!`	*Se guardan los primeros 8 bits de* `r2` *en la dirección* `r1+1` *y luego se incrementa* `r1` *en 1.*
`LDRSH r2, [r1, r2, LSL #4]!`	*Se cargan (con signo) a r2 los 16 bits en memoria desde la dirección* `r1+r2*16` *y luego se guarda esa dirección en* `r1`.
`LDR r0, a`	*Se cargan en* `r0` *32 bits a partir de la dirección de memoria con etiqueta* `a`.

Observación

Notar que en el último ejemplo la etiqueta no está precedida por el símbolo `=`*. Esto se debe a que cuando la etiqueta está dentro del mismo segmento no se debe utilizar dicho símbolo. Sin embargo, si la etiqueta está definida en otro segmento, entonces sí se debe colocar el símbolo* `=`*, previo al nombre de la etiqueta, tal como se mostró en el ejemplo de la página 16.*

2.6.2. Instrucciones de carga y guardado múltiples

ARM tiene instrucciones que pueden guardar varios registros en la memoria a la vez y viceversa. Estas instrucciones tienen la forma:

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

El *opcode* puede ser:

- `LDM`, significa *Load Multiple* (Cargar múltiple). Esta operación carga el contenido de memoria a partir del primer operando en los registros de la lista especificada en el segundo operando.
- `STM`, significa *Store Multiple* (Guardar múltiple). Esta operación guarda el contenido de los registros indicados en la lista en memoria a partir de la dirección especificada por el primer operando.

Estas instrucciones tienen un parámetro *modo*. El *modo* especifica cómo debe usarse la dirección provista en el primer operando, con las siguientes opciones:

- IA, significa *Increment After* (incrementar después): La dirección de registro base (r_n) se incrementa luego de cargar/guardar cada registro.
- IB, significa *Increment Before* (incrementar antes): La dirección de registro base (r_n) se incrementa antes de cargar/guardar cada registro.
- DA, significa *Decrement After* (decrementar después): La dirección de registro base (r_n) se decrementa luego de cargar/guardar cada registro.
- DB, significa *Decrement Before* (decrementar antes): La dirección de registro base (r_n) se decrementa antes de cargar/guardar cada registro.

A su vez, si el signo de admiración (!) se especifica, el registro base se actualiza de acuerdo con el modo. La Fig. 4 muestra el resultado de la instrucción `stm[modo] r0!, {r0-r6}` con los cuatro modos de operación diferentes, todas usando un registro base `r0` con valor `0x8000` como ejemplo.

Para facilitar el trabajo al programador, pueden usarse modos que toman este comportamiento distinto para `STM` y `LDM` automáticamente. A saber, estos son[6]:

[6]Si el modo no se especifica, es por defecto IA.

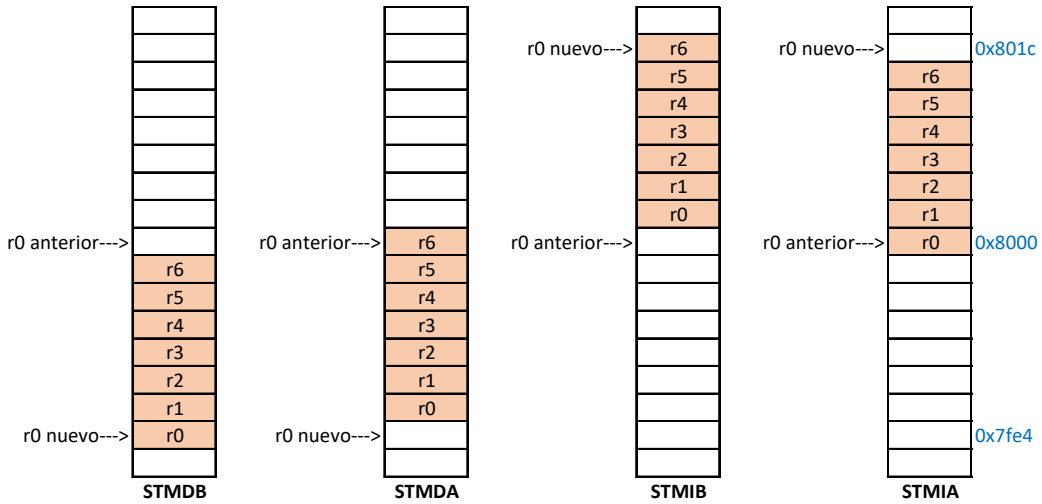


Figura 4: Ejemplo mostrando los diferentes modos de almacenamientos con múltiples datos: `stm[modo] r0!, {r0-r6}`.

Modo	Significado	Instrucción STM	Instrucción LDM
FD	*Full descending*	STMDB	LDMIA
FA	*Full ascending*	STMIB	LDMDA
ED	*Empty descending*	STMDA	LDMIB
EA	*Empty ascending*	STMIA	LDMDB

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}

Ejemplos

- `LDMIA r1, {r2,r7,r8}`
  Cargar las 3 palabras empezando en la dirección `r1` *en los registros* `r2`, `r7` *y* `r8`, *respectivamente. Incrementar la dirección luego de cargar cada registro.*
- `STMDB r0!, {r1-r4}`
  Guardar los registros `r1`, `r2`, `r3` *y* `r4` *en memoria empezando en la dirección* `r0`. *Decrementar la dirección antes de guardar cada registro. Escribir la última dirección en* `r0`.

Observación

Los registros enumerados en la lista se leen en orden lógico (`r0-r15`*), no en el orden expresado en la línea de instrucción.*

Ejemplo

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

.text
.global main
main:
    ldr r9, =a
    ldmfd r9, {r7,r1-r3}
    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, mientras que en la instrucción* `ldmfd r9, {r7,r1-r3}` *el orden de los registros es* `r1,r2,r3,r7`.

2.7. Pseudo-instrucción LDR

La pseudo-instrucción LDR tiene el mismo acrónimo que la instrucción LDR pero es diferente. Su sintaxis es la siguiente:

```
LDR{cond}{W} Rt, =expr

LDR{cond}{W} Rt, =label_expr
```

donde:

- `cond` es un sufijo de ejecución condicional opcional.
- `W` es un sufijo especificador de tamaño opcional.
- `Rt` es el registro que será cargado.
- `expr` evalúa una expresión numérica.
- `label_expr` es una expresión relativa al `PC` o una dirección en forma de etiqueta más/menos un valor numérico.

Cuando se utiliza la pseudo-instrucción `LDR` si el valor de `expr` se puede cargar con una instrucción `MOV` o `MVN` válida, el ensamblador usa esa instrucción. En caso contrario, o si se utiliza la sintaxis `label_expr`, el ensamblador coloca la constante en un grupo de literales (*literal pool*) y genera una instrucción `LDR` relativa al `PC` que lee la constante del grupo de literales.

Ejemplo
Dado el siguiente código:

```
    .data
    place: .word -1

    .text
    .global main
    main:
        LDR  r3, =0xff0
        LDR  r1, =0xffff
        LDR  r2, =place
        bx   lr
```

A continuación vemos el código equivalente que genera el compilador (lo podemos visualizar con `disassemble/r` *dentro de una sesión de GDB):*

```
0x000103e8 <+0>:    ff 3e a0 e3     mov     r3, #4080       ; 0xff0
0x000103ec <+4>:    04 10 9f e5     ldr     r1, [pc, #4]    ; 0x103f8 <main+16>
0x000103f0 <+8>:    04 20 9f e5     ldr     r2, [pc, #4]    ; 0x103fc <main+20>
0x000103f4 <+12>:   1e ff 2f e1     bx      lr
0x000103f8 <+16>:   ff ff 00 00     strdeq  pc, [r0], -pc   ; <UNPREDICTABLE>
0x000103fc <+20>:   88 90 07 00     andeq   r9, r7, r8, lsl #1
```

Vemos que la primera pseudo-instrucción `LDR` *fue reemplazada por la instrucción* `mov r3, #4080` *dado que el valor* `0xff0` *puede ser representado con un valor inmediato de 12 bits (*`0xFF ROR 28`*) como se verá en la Sección 5.*

La segunda pseudo-instrucción `LDR` *fue reemplazada por la instrucción* `ldr r1, [pc, #4]` *que corresponde a una instrucción* `ldr` *que usa una expresión relativa al* `PC`*, refiriéndose a la dirección* `0x000103f8` *donde está el literal pool* `ff ff 00 00`*.*

Finalmente, la tercera pseudo-instrucción `LDR` *fue reemplazada por la instrucción* `ldr r2, [pc, #4]` *que corresponde a una instrucción* `ldr` *que refiere a la dirección* `0x000103fc` *del propio segmento* `.text` *donde está guardada la dirección de la etiqueta* `place` *en el segmento* `.data` *(*`0x00079088`*).*

3. 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.

Ejemplo

`ADDEQ r2, r1, r0`

En este caso a la instrucción `add` *se le agrega el sufijo* `eq`*. Si al momento de ejecutarse la instrucción la bandera* `Z` *del registro CPSR es igual a uno, entonces se realiza la suma de los registro. De lo contrario, no se realiza la suma.*

En la Tabla 2 se muestran todas las condiciones posibles. Las banderas (V, C, Z, N) corresponden a los bits 28 a 31 del CPSR (ver Fig. 2).

Tabla 2: Sufijos utilizados para instrucciones condicionales.

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	Positivo 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

Ejemplo

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 fragmento 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* `r0` *y* `r1`*, respectivamente):*

```
CMP    r0, #0       @ Se realiza la comparación y se setean la banderas.
ADDEQ  r1, r1, r0   @ Si Z=1, se realiza la suma. Si Z=0, se saltea.
MOVNE  r1, #1       @ Si Z=0, se realiza el movimiento. Si Z=1, se saltea.
```

Observación

Las instrucciones 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 sufijo S al final de la instrucción:

```
MOV r0, #0x7fffffff
ADDS r0, #1
```

Luego de ejecutarse la instrucción `ADDS` *el bit 31* `(N)` *y el el bit 28* `(N)` *se encienden, indicando que el resultado es negativo y que hubo overflow.*

```
MOV r1, #0x80000000
ADDS r1, r1
```

Luego de ejecutarse la instrucción `ADDS` *el bit 30* `(Z)`*, el bit 29* `(C)` *y el bit 28* `(V)` *se encienden, indicando que el resultado es cero, que hubo acarreo y overflow.*

4. Barrel shifter

El *barrel shifter* es una unidad lógica que permite ejecutar ciertas operaciones de movimiento de bits en el segundo 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. 5 muestra el *barrel shifter* y la ALU, donde R_n y R_m son contenidos de registros. En esta figura se muestra que al contenido del registro R_m es posible realizare un pre-procesamiento antes de realizar la operación en la ALU. Este pre-procesamiento consiste en alguna de las siguientes operaciones:

- **LSL** : *Logical Shift Left* (corrimiento lógico a la izquierda), mueve los bits a la izquierda, ingresando ceros por la derecha.
- **LSR** : *Logical Shift Right* (corrimiento lógico a la derecha), mueve los bits a la derecha, ingresando ceros por la izquierda.
- **ASR** : *Arithmetic Shift Right* (corrimiento aritmético a la derecha), mueve los bits a la derecha, ingresando por la izquierda el mismo valor que el bit más significativo.
- **ROR** : *Rotate Right* (Rotación a la derecha), mueve los bits a la derecha, ingresando por la izquierda el mismo valor que se va por la derecha.
- **RRX** : *Rotate Right Extended* (Rotación a la derecha extendida), funciona como la operación ROR, pero sobre una palabra de 33 bits, donde el bit de carry del CPSR es el bit 33.

Ejemplos

- `MOV r7, r5, LSL #2`
  `r7 := r5`$\times 4 =$ `r5 << 2`
- `LSL r7, r5, #2`
  `r7 := r5`$\times 4 =$ `r5 << 2`. *Esta instrucción es equivalente a la anterior.*
- `MOV r0, r1, ROR #2`
  `r0 := r1`$/4 =$ `r0 >> 2`
- `MOV r2, r2, ROR #16`
  Intercambia los 16 bits más significativos de `r2` *con sus 16 bits menos significativos.*

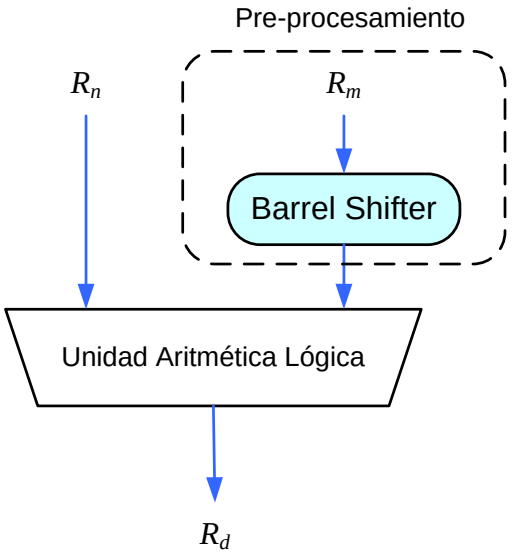


Figura 5: *Barrel Shifter* y ALU.

- `ADD r1, r1, LSL #5`
  Multiplica `r1` *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*

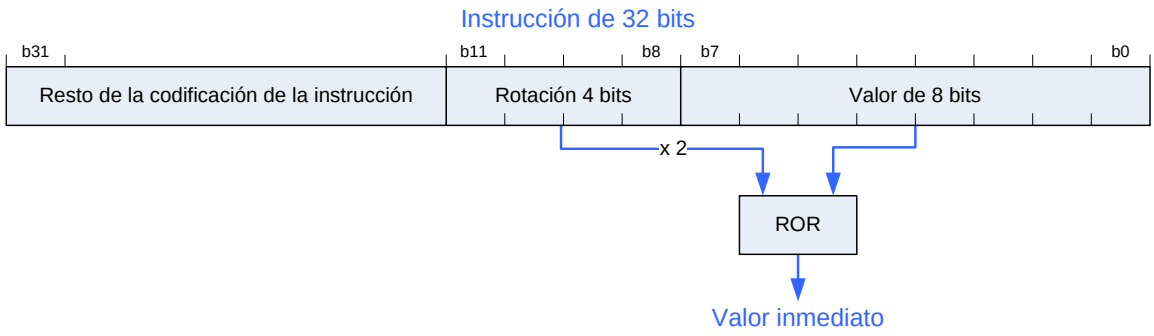
Una instrucción multiplicación (multiplicando por una constante) implica primero cargar la constante en un registro y luego esperar un número de ciclos internos para completar la instrucción. Sin embargo, generalmente se puede hallar una solución más eficiente mediante alguna combinación de instrucciones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

5. 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 (en las instrucciones que involucran valores inmediatos). 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: los 8 bits menos significativos se usan para definir un valor de 0 a 255 y los 4 bits restantes 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 que muestra como se codifica una instrucción:



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.

Ejemplo

El siguiente esquema muestra cómo se codifica el número `0x104`*:*

En el esquema superior vemos cómo el número `0x104` *se codifica dentro de la instucción. Al aplicar la rotación mostrada se llega al número* `0x104`*, mostrado en el esquema inferior.*

Veamos una instrucción en concreto que utiliza el valor inmediato del ejemplo. Por ejemplo, la instrucción `mov r1, #0x104` *se codifica como* `0xe3a01f41` *(esto lo podemos verificar usando el comando* `disassemble` *en GDB). Aquí se corrobora lo mostrado en el esquema superior.*

La Fig. 6 muestra todas las posibles rotaciones. Por lo tanto, algunos ejemplos de números posibles y no posibles de representar son:

- `0x00000000` ✓ (`0x0 ROR 0`)
- `0x000000FF` ✓ (`0xFF ROR 0`)
- `0xFF000000` ✓ (`0xFF ROR 8`)
- `0x00000FF0` ✓ (`0xFF ROR 28`)
- `0x007F0000` ✓ (`0x7F ROR 16`)
- `0xF000000F` ✓ (`0XFF ROR 4`)
- `0x00000104` ✓ (`0x41 ROR 30`)
- `0x00000102` × (La constante requiere una rotación impar)
- `0xF000F000` × (La constante rotada es demasiado grande)
- `0x00000123` × (La constante rotada es demasiado grande)

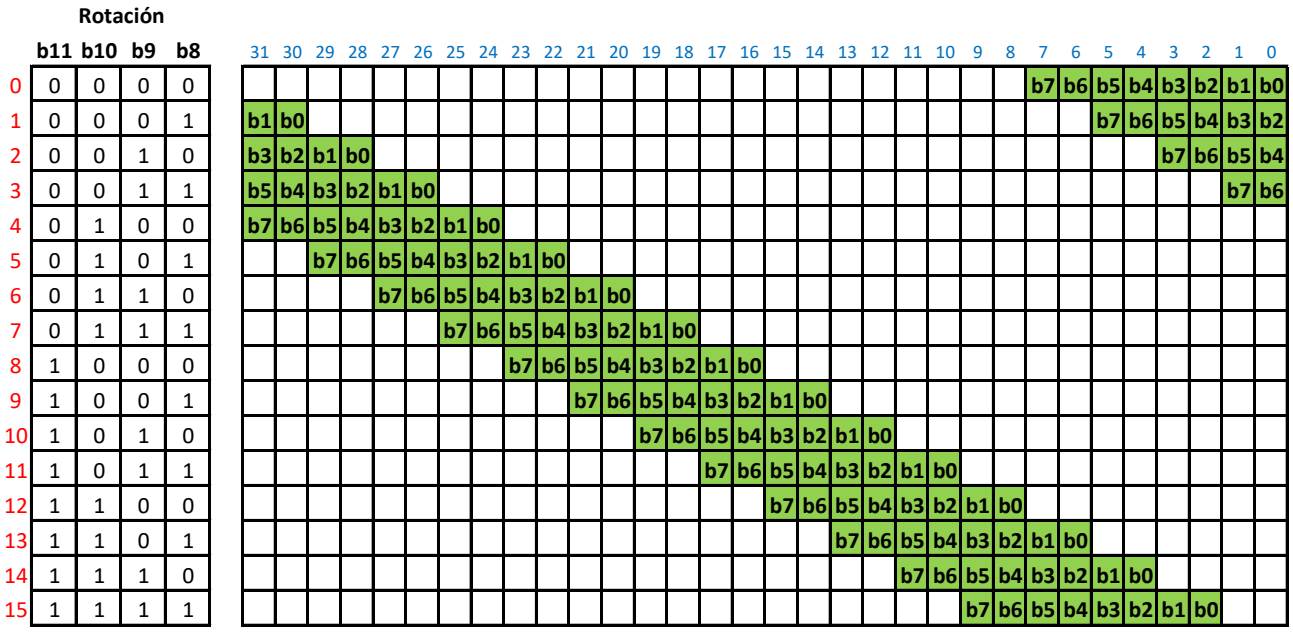


Figura 6: Esquema de valores inmediatos representables con las diferentes rotaciones posibles.

- `0x0F0000F0` × (La constante rotada es demasiado grande)

Ejemplos

A continuación vemos algunos ejemplos de uso de valores inmediatos:

```
e3a0007b    MOV r0, #123                @ 123 = 0x7b ROR 0
e3855902    ORR r5, r5, #0x8000         @ 0x8000 = 0x2 ROR 18
e2299102    EOR r9, r9, #0x80000000     @ 0x80000000 = 0x2 ROR 2
```

La columna de la izquierda muestra el código de operación (lenguaje de máquina) en hexadecimal donde se puede observar cómo se codifican los valores inmediatos con rotaciones adecuadas.

Si un programa usa una constante que no puede ser expresada a través de uno de los valores inmediatos posibles de implementar como se acaba de explicar, existen varias opciones como se muestra en el siguiente ejemplo:

Ejemplo

Movimiento de un valor inmediato de 32 bits

Supongamos que queremos cargar el valor inmediato `0x11223344` *en el registro* `r0`*. Este valor inmediato es de 32 bits y no se puede expresar como un valor de 8 bits con una rotación a la derecha una cantidad de veces par, tal como se acaba de explicar. Por lo tanto, NO lo podemos cargar de manera directa usando:*

```
MOV r0, #0x11223344
```

Sin embargo, existen varias formas alternativas de lograrlo:

Opción 0 *La forma más trivial es ir cargando el valor por partes, usando operaciones aritméticas:*

```
MOV  r0, #0x00000044
ADD  r0, r0, #0x00003300
ADD  r0, r0, #0x00220000
ADD  r0, r0, #0x11000000
```

Opción 1 *Otra opción es cargarlo en dos pasos, usando las instrucciones de movimiento de 16 bits ya vistas:*

```
MOVW r1, #0x3344
MOVT r1, #0x1122
```

En la primera instrucción se carga 0x3344 en los 16 bits menos significativos y los restantes quedan en cero mientras que en la segunda se carga 0x1122 en los 16 bits más significativos y los bits menos significativos no se modifican.

Opción 2 *Otra opción es usar la pseudo-instrucción* `LDR` *(La instrucción* `LDR` *se verá en detalle en la Sección 2.6):*

```
LDR  r0, =0x11223344
```

La pseudo-instrucción `LDR` *inserta una instrucción* `MOV` *o* `MVN` *para generar un valor (si es posible) o genera una instrucción* `ldr` *con una dirección relativa al* `pc` *para leer la constante desde un literal pool (un área de datos incrustada dentro del código de texto).*

Por ejemplo, el valor 0x11223344 no puede ser generado mediante una instrucción de movimiento. Por lo tanto, el compilador y el ensamblador convierten la pseudo-instrucción `ldr r0, =0x11223344` *en:*

```
0x000103f0 <+8>:     04 20 9f e5     ldr r2, [pc, #4]; 0x103fc <main+20>
...................
...................
0x000103fc <+20>:    44 33 22 11     ; <UNDEFINED> instruction: 0x11223344
```

Opción 3 *Finalmente, el valor inmediato se puede cargar desde memoria:*

```
.data
num: .word 0x11223344

.text
.......
    LDR  r1, =num
    LDR  r0, [r1]
.......
```

6. La pila

Como ya hemos visto en la arquitectura x86-64, es necesario almacenar en la pila el estado del procesador para realizar llamados a funciones. La pila también es útil en cualquier programa que maneje grandes cantidades de datos. Las instrucciones de transferencia de datos múltiples vistas previamente (`LDM` y `STM`) proveen un mecanismo para almacenar datos en la pila.

Tradicionalmente, una pila crece hacia abajo en memoria, lo que significa que el último valor "*pusheado*" estará en la dirección más baja. ARM también admite pilas ascendentes utilizando, lo que significa que la estructura de la pila también puede crecer hacia arriba a través de la memoria. Sin embargo, el modo más común es el *full descending*.

En ARM el registro que apunta al tope de la pila (*Stack Pointer*) es el registro `r13`. Recordar que las instrucciones `STM` y `LDM` tienen distintos modos de acceso. Por lo tanto podemos utilizar estas instrucciones para implementar diferentes modos de pila.

Ejemplo

El modo más usual es el full descending que se muestra a continuación:

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

  Apila `r4`, `r5`, `r6` *y* `r7` *en la pila y* `r13` *queda apuntando al final de la pila (full). Es equivalente a* `push {r4-r7}`.

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

  Desapila `r4`, `r5`, `r6` *y* `r7` *desde la pila. Es equivalente a* `pop {r4-r7}`.

Finalmente, las operaciones `PUSH` y `POP` son sinónimos de `STMDB` y `LDMIA`, respectivamente, usando implícitamente el registro `r13` como *stack pointer*.

Ejemplo

```
.......              @ algún código
PUSH {r0-r3,r12,lr} @ "Pushea" los registros de trabajo, r12 y el link register
BL my_function
.......              @ my_function retornará aquí
POP {r0-r3,r12,lr}  @ "Popea" los registros de trabajo, r12 y el link register
.......              @ más código
```

En este ejemplo, antes de ingresar a `my_function` *se preservan los registros* `r0-r3` *dado que son caller-save, el* `r12` *(Instruction Pointer) y el Link Register.*

7. Llamada a función

Cuando una función llama a otra, la función que llama, *caller*, y la función llamada, *callee*, deben ponerse de acuerdo sobre dónde colocar los argumentos y el valor de retorno. En ARM, el *caller* coloca convencionalmente hasta cuatro argumentos en los registros `r0-r3` antes de realizar la llamada a la función, y el *callee* coloca el valor de retorno en el registro `r0` antes de terminar. Siguiendo esta convención, ambas funciones saben dónde encontrar los argumentos y devolver el valor, incluso si el *caller* y el *callee* fueron escritas por diferentes personas.

El *callee* no debe interferir con el comportamiento del *caller*. Esto significa que el *callee* debe saber adónde regresar después de que se complete y no debe "pisar" ningún registro o memoria que necesite el *caller*. El *caller* almacena la dirección de retorno en el registro de enlace `lr` al mismo tiempo que salta al destino de la llamada utilizando la instrucción de *bifurcación y enlace* (`bl`). El *callee* no debe sobrescribir ningún registro o lugar de memoria de la que dependa el *caller*. Específicamente, el *callee* debe dejar los registros guardados (`r4-r11` y `lr`) y la pila sin modificar.

ARM usa la instrucción de bifurcación y enlace (`BL`) para llamar a una función. De esta manera, la instrucción `BL etiqueta` realiza dos tareas:

1. Almacena la dirección de retorno de la siguiente instrucción (la instrucción después de `BL`) en el registro de enlace (`lr`).

2. Bifurca la ejecución a la dirección de correspondiente a `etiqueta`.

Luego, una vez realiza las tareas de la función llamada se puede retornar con `BX lr`. Esta instrucción mueve el contenido del registro de enlace al `pc` (es decir, es equivalente a `MOV pc, lr`) para regresar al *caller* (recordar que en la Sección 2.5 vimos que `MOV pc, lr` es equivalente a `BX lr`).

Ejemplo

En este ejemplo se muestra un programa que reemplaza llama a una función `suma` *con tres argumentos de tipo entero, la cual retorna la suma de los tres argumentos y luego imprime dicho resultado:*

```
#include <stdio.h>

int suma(int a, int b, int c){
    return a+b+c;
}

int main(){
    printf("%d\n", suma(4, 5, 6));
    return 0;
}
```

El siguiente sería un posible código equivalente en Assembler ARM:

```
.data
str: .asciz "%d\n"

.text
suma:
     sub     sp, sp, #16
     str     r0, [sp, #12]
     str     r1, [sp, #8]
     str     r2, [sp, #4]
     ldr     r2, [sp, #12]
     ldr     r3, [sp, #8]
     add     r2, r2, r3
     ldr     r3, [sp, #4]
     add     r3, r3, r2
     mov     r0, r3            @ retornamos el resultado de la suma
     add     sp, sp, #16
     bx      lr                @ Se retorna a main

.global main
main:
    push    {lr}       @ Se preserva el valor de LR
    mov     r2, #6
    mov     r1, #5
    mov     r0, #4
    bl      suma       @ Se carga en LR la dirección de retorno y se bifurca
    mov     r1, r0
    ldr     r0, =str
    bl      printf
    mov     r0, #0
    pop     {lr}       @ Se restaura el valor de LR
    bx      lr         @ Retorna
```

Observación

Supongamos ahora que la función `suma` *llama a vez a otra función. Por ejemplo, llama a* `printf` *de la siguiente manera:*

```
#include <stdio.h>

int suma(int a, int b, int c){
    printf("Calculando...\n");
    return a+b+c;
}

int main(){
    printf("%d\n", suma(4, 5, 6));
    return 0;
}
```

Dijimos que la dirección de retorno es guardada en el registro `lr`*, pero ahora al llamar a* `printf` *dentro de la función* `suma` *se "pisaría" el valor con la nueva dirección de retorno. ¿Cómo lo podemos solucionar? Respuesta: preservando el valor de la dirección de retorno de* `suma` *en la pila antes de llamar a* `printf`*:*

```
.data
str: .asciz "%d\n"
str2: .asciz "Calculando...\n"

.text
suma:
    push    {lr}            @ Se preserva el valor de LR
    sub     sp, sp, #16
    str     r0, [sp, #12]
    str     r1, [sp, #8]
    str     r2, [sp, #4]
    ldr     r0, =str2
    bl      printf
    ldr     r2, [sp, #12]
    ldr     r3, [sp, #8]
    add     r2, r2, r3
    ldr     r3, [sp, #4]
    add     r3, r3, r2
    mov     r0, r3
    add     sp, sp, #16
    pop     {lr}            @ Se restaura el valor de LR
    bx      lr              @ Se retorna a main

.global main
main:
    push    {lr}        @ Se preserva el valor de LR
    movs    r2, #6
    movs    r1, #5
    movs    r0, #4
    bl      suma        @ Se guarda la dirección de retorna en LR y se bifurca
```

```
    mov     r3, r0
    mov     r1, r3
    ldr     r0, =str
    bl      printf
    mov     r0, #0
    pop     {lr}        @ Se restaura el valor de LR
    bx lr               @ Retorna
```

8. Operaciones de punto flotante

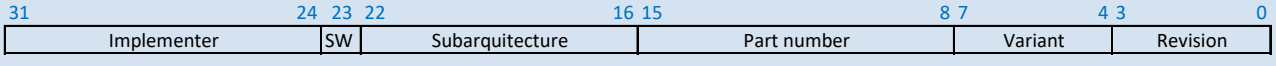
Las primeras generaciones de procesadores ARM no tenían soporte para operaciones de punto flotante. Esto significaba hacer operaciones con enteros o utilizar un coprocesador externo para hacer operaciones de punto flotante.

Para versiones de la arquitectura ARM más recientes, la compañía creó una familia de coprocesadores de punto flotante que denominó VFP (*Vector Floating Point*)[7]. Estos coprocesadores vienen opcionalmente con algunos procesadores. Existen varias versiones de VFP, a saber:

Versión	**Descripción**
VFPv1	Versión 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 $a \leftarrow a + b \times c$).

La información sobre el tipo de coprocesador de punto flotante está disponible en un registro especial de sólo lectura llamado FPSID.

En ARMv7, la asignación de bits en el registro FPSID es la siguiente:



[7] Punto flotante vectorizado

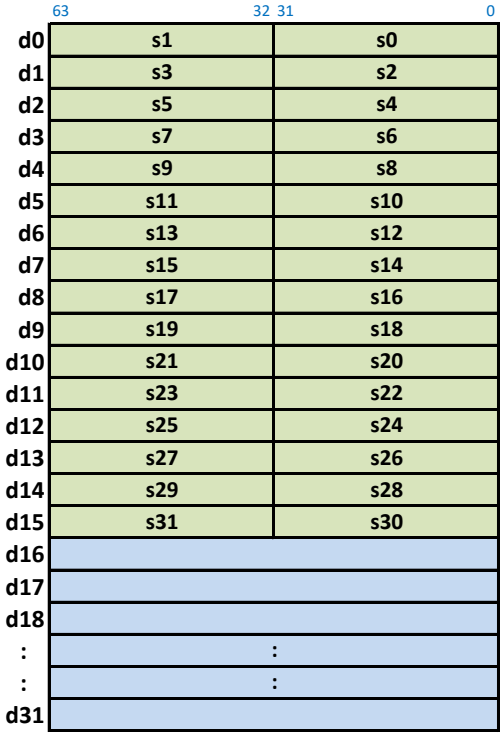


Figura 7: Relación entre registros de punto flotante.

En particular, el bit 23 es el bit de software. Este bit indica si un sistema proporciona solo emulación de software de las instrucciones de punto flotante proporcionadas por la extensión de punto flotante. Si está en "0" indica que el sistema incluye soporte de hardware para las instrucciones de punto flotante proporcionadas por la extensión de punto flotante. Si está en "1" indica que el sistema proporciona únicamente emulación de software de las instrucciones de punto flotante proporcionadas por la extensión de punto flotante.

8.1. Registros de punto flotante

Las operaciones de punto flotante utilizan registros especiales para hacer sus operaciones (registros FPU). 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. Es decir, existe la siguiente relación entre registros:

$$d\left[x\right] \Longleftrightarrow \left\{s\left[(2x)+1\right], s\left[2x\right]\right\}, \quad \text{para } 0 \leq x < 15$$

lo cual se grafica en la Fig. 7.

Observación

Los registros FPU pueden utilizarse para contener uno o más datos de punto flotante (dependiendo de la precisión), y además pueden contener uno o más datos enteros (dependiendo del tipo de dato). Veremos en la Sección 8.8 ejemplos al respecto.

8.2. 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)[8]. 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 VCMP, que toma dos registros de punto flotante como parámetros.

La Fig. 8 muestra el registro FPSCR. En particular, se muestran los bits correspondientes a las banderas N (*negative*), Z (*zero*), C (*carry*) y V (*overflow*).

Figura 8: Registro FPSCR.

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 algunas diferencias. Una de las diferencias es que no existen sufijos para comparaciones con signo o sin signo, ya que todos los números de punto flotante tienen signo. Otra diferencia importante tiene que ver con el valor especial NaN (*Not a Number*), que designa en la norma IEEE 754 de punto flotante 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 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

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

8.3. Instrucciones *Load/Store*

La instrucción de carga de un número de punto flotante desde memoria a un registro de punto flotante es `VLDR`, mientras que 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 tiene las siguientes forma:

$$\texttt{VLDR|VSTR}[cond][.tipo]\ F_d\,,\ [r_n\{,\ \#\text{offset}\}]$$

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 el `offset` para determinar la dirección en memoria para guardar/cargar el registro.

Ejemplo

```
vldr.f32 s5, [r6, #8]
```

Carga un valor de 32 bits ubicado en memoria en el registro FPU `s5`*. La dirección se crea a partir del valor en el registro* `r6` *más un desplazamiento de 8. No es necesario incluir el* `.f32` *en el formato de instrucción anterior, pero es una buena práctica hacer explícito el tipo de datos siempre que sea posible.*

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:

$$\boxed{\texttt{VLDM|VSTM}\{modo\}[cond]\ R_n[!]\,,\{\text{Lista de registros}\}}$$

donde *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.

Ejemplo

```
vldm.f32 r0!, {s0,s1}
```

Carga en los registros `s0` *y* `s1` *las palabras (de 32 bits) a partir de la dirección en* `r0`*. Luego incrementa* `r0` *en 8. Notar que por defecto corresponde el modo* `ia`*.*

8.4. Instrucciones de manejo de pila

Existen operaciones específicas para manejar la pila que tienen las siguientes equivalencias:

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

Ejemplo

```
vpop.f32 {s0,s1}
vldmia.f32 sp!, {s0,s1}
```

Estas dos instrucciones son equivalentes y por lo tanto en los registros `s0` *y* `s1` *quedan almacenados los mismos valores. En la segunda instrucción se hace uso explícito del registro* `sp` *mientras que en la primera el uso del mismo es implícito.*

8.5. Instrucciones de movimiento de datos

La instrucción `VMOV` permite copiar datos entre los registros ARM y los registros FPU:

```
VMOV{<cond>}.F32 <Sd>, <Rt>
VMOV{<cond>}.F32 <Rt>, <Sn>
```

La primera de estas instrucciones transfiere un operando de 32 bits de un registro ARM a un registro FPU; la segunda entre un registro FPU a un registro ARM. El formato del tipo de datos se asume por defecto en la extensión `.F32`. De lo contrario, se requiere incluir el tipo de datos. El tipo de datos puede ser *half-precision* (`.F16`), *single-precision* (`.F32` o `.F`), o *double-precision* (`.F64` o `.D`).

Ejemplos

```
vmov.f32 s0, r0
vmov.f32 r1, s1
```

En la primera instrucción de hace un movimiento de registro de propósito general a un registro FPU, mientras que en la segunda se hace lo opuesto.

La instrucción `VMOV` también se puede usar para transferir datos entre registros FPU. La sintaxis es:

```
VMOV{<cond>}.F32 <Sd>, <Sn>
```

Ejemplo

```
vmov.f32 s0, s1
```

Carga en `s0` *el contenido de* `s1`.

La instrucción `VMOV` también permite mover dos registros ARM a dos registros simple precisión, dos registros simple precisión a dos registros ARM o dos registros ARM a un registro doble precisión:

```
VMOV{<cond>}.F32 <Sn>, <Sm>, <Rd>, <Rn>
VMOV{<cond>}.F32 <Rd>, <Rn>, <Sn>, <Sm>
VMOV{<cond>}.F32 <Dm>, <Rd>, <Rn>
```

donde `Sm` debe ser `S(n+1)`, acuerdo a lo visto en la descripción de los registros FPU (Fig. 7).

Ejemplo

```
mov r0, #5
mov r1, #4
vmov.f64 s8, s9, r0, r1
```

Luego de ejecutarse, `s8=0x00000005` *y* `s9=0x00000004`*. Por lo tanto,* `d4=0x0000000400000005`.

8.6. 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:

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

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

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 (estas opciones están disponibles en la extensión FPv5). 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

Ejemplo

```
.data
a: .word 0x40600000     @ a=3.5
b: .word 2

.text
.global main
main:
    ldr r0, =a
    ldr r1, [r0]
    vmov s1, r1
```

```
    vcvt.s32.f32 s0, s1     @ s0=3

    ldr r0, =b
    ldr r1, [r0]
    vmov.f32 s1, r1
    vcvt.f32.s32 s0, s1     @ s0=0x40000000 (2.0)
    bx lr
```

En la primera instrucción `vcvt` *se realiza una conversión de punto flotante a entero (redondeando el valor al entero más próximo hacia cero), mientras que en la segunda se realiza una conversión de entero a punto flotante.*

8.7. 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.

Ejemplos

```
vcvt.f64.f32 d0, s0      @ d0=0x4010000000000000 (d0=4.0)
```

Suponiendo que `s0=0x40800000` *antes de ejecutarse la instrucción.*

```
mov r0, #2
vmov s0, r0
vcvt.f32.s32 s1, s0     @ s0=0x40000000
vcvtt.f16.f32 s2, s1    @ s2=0x40000000     (s2=2.0)
vcvtb.f16.f32 s3, s1    @ s3=0x00004000     (s3=2.2958874e-41)
```

Notar la diferencia entre las dos últimas instrucciones. Si el resultado luego se interpreta como un número de 32 bits, el resultado de la segunda instrucción es incorrecto. En cambio si solo se tienen en cuenta los 16 bits menos significativo, el resultado es correcto.

8.8. Instrucciones para procesamiento de datos

La sintaxis para las instrucciones de procesamiento de datos en punto flotante es la siguiente:

$$\texttt{V} < \texttt{operación} > \{cond\}.F32\ \{< \texttt{dest} >\},\ < \texttt{src1} >,\ < \texttt{src2} >$$

Los registros `src1`, `src2` y `dest` pueden ser cualquiera de los registros de precisión simple (`s0` a `s31`). Los registros `src1`, `src2` y `dest` pueden ser el mismo registro, registros diferentes, o cualquiera de los dos puede ser el mismo registro. Por ejemplo, para elevar al cuadrado un valor en el registro `s9` y colocar el resultado en el registro `s0`, se podría usar la siguiente instrucción de multiplicación: `VMUL.F32 s0, s9, s9`

La siguiente muestra las principales instrucciones disponibles de procesamiento de datos en punto flotante:

Operación	**Formato**	**Descripción**
Valor absoluto	`VABS{cond}.F32 <Sd>, <Sm>`	`Sd = |Sm|`
Negación	`VNEG{cond}.F32 <Sd>, <Sm>`	`Sd = -1 * Sn`
Suma	`VADD{cond}.F32 <Sd>, <Sn>, <Sm>`	`Sd = Sn + Sm`
Resta	`VSUB{cond}.F32 <Sd>, <Sn>, <Sm>`	`Sd = Sn - Sm`
Multiplicación	`VMUL{cond}.F32 <Sd>, <Sn>, <Sm>`	`Sd = Sn * Sm`
División	`VDIV{cond}.F32 <Sd>, <Sn>, <Sm>`	`Sd = Sn/Sm`
Raíz cuadrada	`VSQRT{cond} <Sd>, <Sm>`	`Sd = Sqrt(Sm)`
Comparación	`VCMP{E}{cond}.F32 <Sd>, <Sm>`	Setea las banderas del FPSCR en base a la comparación de `Sd` y `Sm`

Ejemplos

```
vabs.f32 s1,s0          @ s1=0x40800000 (s1=4.0)
vneg.f32 s2,s0          @ s2=0x40800000 (s2=4.0)
vadd.f32 s3, s2, s1     @ s3=0x41000000 (s3=8.0)
```

Suponiendo que inicialmente `s0=0xc0800000 (s0=-4.0)`.

Observación

Como hemos mencionado, los registros FPU pueden usarse como almacenamiento temporal para cualquier valor, tanto de punto flotante como entero. Incluso un registro puede contener más de un dato, dependiendo del tipo de dato.

Ejemplo

```
mov r0, #5              @ r0=0x00000005
mov r1, #4              @ r1=0x00000004
vmov.f64 d2, r0, r1     @ d2=0x0000000400000005
vadd.f64 d3, d2, d2     @ d3=0x000000080000000a
```

La instrucción `vmov.f64 d2, r0, r1` *permite mover el contenido de dos registros de propósito general a un registro FPU de 64 bits, mientras que la instrucción* `vadd.f64 d3, d2, d2` *realiza una suma vectorial. Notar además que* `s6=0x0000000a` *y* `s7=0x00000008`*, de acuerdo a lo mostrado en la Fig. 7.*

8.9. Ejemplo general con instrucciones de punto flotante

Ejemplo

A continuación se muestra un ejemplo general que utiliza varias instrucciones en punto flotante para realizar la operación $10 + 1.75 \times 2$ *y retornar el resultado como entero:*

```
.data
valFloat:  .word 0x3fe00000   @ 1.75
valFloat2: .word 0x40000000   @ 2.0

.text
.global main
main:
    ldr   r1, =valFloat     @ En r1 queda la dirección de valFloat
    vldr  s0, [r1]          @ En s0 queda el valor 1.75

    ldr   r1, =valFloat2    @ En r1 queda la dirección de valFloat2
    vldr  s1, [r1]          @ En s1 queda el valor 2.0

    vmul.f32 s2, s0, s1     @ s2=s0*s1=3.5
    mov r0, #10             @ r0=10
    vmov.f32 s0, r0         @ s0=10.0
    vcvt.f32.s32 s3, s0     @ s3=10.0
    vadd.f32 s4, s3, s2     @ s4=s2+s3=13.5

    vcvt.s32.f32 s0, s4     @ Conversión de punto flotante a entero (s0=13)
    vmov  r0, s0            @ Copia del registro s0 al registro r0 (r0=13)

    bx    lr                @ Retorna r0=13
```

8.10. Llamada a función con argumentos de punto flotante

Ejemplo

En este ejemplo se muestra un código en ARM que es equivalente al siguiente programa en lenguaje C, donde se llama a una función que realiza la suma de varios valores (enteros y flotantes):

```
#include<stdio.h>
float calculo(int a, int b, int c, int d, float e, double f, int g, int h){
    return a + b + c + d + e + f + g + h;
}

int main(){
    printf("La suma es: %f\f", calculo(1, 2, 3, 4, 5.0, 6.0, 7, 8));
    return 0;
}
```

Los argumentos `a`, `b`, `c` *y* `d` *son enteros y por lo tanto se pasan a través de los registros* `r0`, `r1`, `r2` *y* `r3`*, respectivamente. Los argumentos* `e` *y* `f` *son flotantes (*`float` *y* `double`*, respectivamente)*

y por lo tanto se pasan a través de los registros `s0` *y* `d1`*, respectivamente. Los argumentos* `g` *y* `h` *son de tipo enteros pero, como los registros* `r0-r3` *ya fueron usados, se deben pasar por pila en sentido inverso. Finalmente, se retorna el resultado del cálculo realizado a través del registro* `s0` *y se imprime llamando a la función* `printf`*.*

```
calculo:
    sub     sp, sp, #32
    str     r0, [sp, #28]           @ a
    str     r1, [sp, #24]           @ b
    str     r2, [sp, #20]           @ c
    str     r3, [sp, #16]           @ d
    vstr.32 s0, [sp, #12]           @ e
    vstr.64 d1, [sp]                @ f
    ldr     r2, [sp, #28]
    ldr     r3, [sp, #24]
    add     r2, r2, r3              @ a + b
    ldr     r3, [sp, #20]
    add     r2, r2, r3              @ a + b + c
    ldr     r3, [sp, #16]
    add     r3, r3, r2              @ a + b + c + d
    vmov    s1, r3
    vcvt.f32.s32    s0, s1          @ convertimos a float
    vldr.32         s1, [sp, #12]   @ e
    vadd.f32        s1, s0, s1      @ a + b + c + d + e
    vcvt.f64.f32    d1, s1          @ convertimos a double
    vldr.64         d0, [sp]        @ f
    vadd.f64        d1, d1, d0      @ a + b + c + d + e + f
    ldr             r3, [sp, #32]   @ g
    vmov            s1, r3
    vcvt.f64.s32    d0, s1          @ convertimos a double
    vadd.f64        d1, d1, d0      @ a + b + c + d + e + f + g
    ldr             r3, [sp, #36]   @ h
    vmov            s1, r3
    vcvt.f64.s32    d0, s1          @ convertimos a double
    vadd.f64        d0, d1, d0      @ a + b + c + d + e + f + g + h
    vcvt.f32.f64    s1, d0          @ convertimos a float
    vmov.f32        s0, s1          @ valor de retorno de calculo
    add             sp, sp, #32
    bx              lr

.data
str: .asciz  "La suma es: %f\n"

.text
.global main
main:
    push    {lr}
    sub     sp, sp, #12
    movs    r3, #8
```

```
        str     r3, [sp, #4]        @ h
        movs    r3, #7
        str     r3, [sp]            @ g
        vmov.f64    d1, #6.0e+0     @ f
        vmov.f32    s0, #5.0e+0     @ e
        movs    r3, #4              @ d
        movs    r2, #3              @ c
        movs    r1, #2              @ b
        movs    r0, #1              @ a
        bl      calculo
        vcvt.f64.f32    d1, s0      @ convertir a double el resultado
        vmov    r2, r3, d1          @ segundo argumento de printf
        ldr     r0, =str            @ primer argumento de printf
        bl      printf
        mov     r0, #0              @ valor de retorno de main
        pop     {lr}
        bx      lr
```

Observación

El segundo argumento de la función `printf` *es un* `float`*. Sin embargo, la* `printf` *imprime valores tipo* `double`*. Por lo tanto, debemos hacer la conversión antes de realizar el llamado a* `printf`*. Además, debemos notar que en arquitectura ARM, a diferencia de X86-64, no se le pasa la cantidad de argumentos de tipo flotante en un registros. En realidad, cuando llamamos a una función variádica (por ejemplo,* `printf`*) en ARM todos los argumentos se pasan a través de los registros* `r0-r3` *de 32 bits y por pila. Entonces, la dirección de la cadena de formato la pasamos a través de* `r0`*, luego el valor* `double` *ocupa dos registros consecutivos de 32 bits. Sin embargo, no usamos el registro* `r1` *porque por convención el primero de estos dos registros consecutivos debe ser par. Por lo tanto, usamos los registros* `r2` *y* `r3`*.*

Como ejercicio, analizar utilizando el compilador el código equivalente de la siguiente llamada a `printf`*:*

```
#include<stdio.h>
int main(){
    printf("%d %f %d %f\f", 1, 2.0, 3, 4);
    return 0;
}
```

A. 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**.

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

```
$ arm-linux-gnueabi-gcc -static -o prueba prueba.s
```

Compila el programa `prueba.s` escrito en ARM. La opción `-static` es necesaria para compilar de forma estática. Luego, se ejecuta el archivo binario usando el emulador QEMU:

```
$ qemu-arm-static prueba
```

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 en lenguaje C con ensamblador siempre y cuando se respete la convención de llamada.

Ejemplo

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 con la etiqueta* `suma` *declarada como global:*

```
.text
.global suma
suma:
     add r0, r0, r1     @ Argumentos: r0 y r1, retorna: r0
     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-static suma
```

para finalmente obtener:

```
La suma es: 5
```

Observación

Notar que para compilar se han utilizado las 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. Si se quiere compilar en modo THUMB usar la opción* `-mthumb`*. Por defecto compila en modo ARM, por lo cual en general no necesitaremos indicar esta opción. Sin embargo, en algunas ocasiones, por ejemplo cuando se mezcla código C y código ARM, es necesario utilizar la opción* `-marm` *de manera explícita.*

Por otra parte, si se usan valores en punto flotante, compilar utilizando el paquete `arm-linux-gnueabihf-gcc` *de la siguiente manera*[9]*:*

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

B. Depuración

Para depurar un programa (*debuggear*), primero compilar usando la opción `-g`:

```
$ arm-linux-gnueabi-gcc -static -g -o hola hola.c
```

Luego, ejecutar agregando un número de puerto de sistema arbitrario, por ejemplo 1234:

```
$ qemu-arm-static -g 1234 hola
```

Una vez ejecutado, se queda esperando conexión con el puerto del sistema indicado.

Luego, en otra terminal, ejecutar GDB:

```
$ gdb-multiarch hola
```

Una vez iniciada la sesión de depuración, conectar con el puerto local seleccionado y ya se puede comenzar a depurar:

```
> target remote localhost:1234
> br main
> continue
> info reg
> next
```

Observación

Para comenzar la ejecución de las instrucciones usar el comando `continue` *y NO usar el comando* `run`*, a diferencia de lo visto en x86-64.*

Para una información más detallada, ver el documento **Guía de desarrollo para otras arquitecturas** en la Sección **Apuntes propios de la asignatura** del Campus Virtual.

C. Llamado a la función `printf`

A continuación vemos un ejemplo para ver cómo imprimir valores de tipo entero usando la función `printf`:

```
.data
str: .string "%d\n"

.text
.global main
main:
    push {ip, lr}   @ push return addres + dummy register para alineación
    ldr r0, =str    @ primer argumento (dirección de la cadena de formato)
    mov r1, #45     @ segundo argumento (valor de tipo entero)
    bl printf       @ llamado a printf
    mov r0, #0
    pop {ip, lr}
    bx lr
```

Observación

La convención de llamada establece que los argumentos se pasan usando los registros `r0-r3` *para pasar los primeros 4 argumentos. En este ejemplo fueron suficientes porque solo necesitamos pasar dos argumentos: la dirección de la cadena de formato y el valor de tipo entero ¿Qué pasa si necesitamos pasar más de 4 argumentos? Tenemos que usar la pila. Veamos el siguiente ejemplo dónde se imprimen 6 enteros: los 4 primeros argumentos se pasan a través de los registros* `r0-r3` *y los restante usando la pila.*

```
.data
array: .word 1, 2, 3, 4, 5, 6
str: .asciz "%d %d %d %d %d %d\n"

.text
.global main
main:
    push {ip,lr}
    ldr r0, =str        @ primer argumento (dirección de la cadena de formato)
    ldr r7, =array
    ldr r1, [r7]            @ segundo argumento
    ldr r2, [r7,#4]         @ tercer argumento
    ldr r3, [r7,#8]         @ cuarto argumento
    ldr r4, [r7,#12]        @ quinto argumento
    ldr r5, [r7,#16]        @ sexto argumento
    ldr r6, [r7,#20]        @ séptimo argumento
    push {r4-r6}            @ se pushean en la pila los argumentos 5, 6 y 7
    bl printf
    pop {r4-r6}
    pop {ip,lr}
    bx lr
```

Finalmente, vemos a continuación un ejemplo para ver cómo imprimir valores de tipo flotante:

```
.data
f: .float   3.14
str: .asciz  "%f\n"

.text
.global main
main:
    push {r7, lr}
    add r7, sp, #0
    ldr r3, =f
    vldr.32 s15, [r3]
    vcvt.f64.f32 d1, s15
    vmov r2, r3, d1
    ldr r0, =str
    bl printf
    mov r0, #0
    pop {r7, pc}
    bx lr
```

Observación

Notar que a diferencia de `x86-64` *donde explícitamente le pasábamos la cantidad de argumentos a imprimir de punto flotante a través de un registro específico, en ARM no lo hacemos así. Entonces, ¿cómo funciona? Viendo el ejemplo anterior vemos que el procedimiento es un poco más complejo. En primer lugar, observemos que para pasar argumento solo se utilizan registros de tipo entero. Es decir, los registros* `r0`, `r1-r3`. *Observar entonces que el contenido del registro* `d1` *(64 bits) se carga en los registros* `r2` *y* `r3` *(32 bits cada uno). Recordemos que la función* `printf` *imprime valores tipo* `double` *por lo tanto hay que realizar la conversión de manera explícita a través de la instrucción* `vcvt.f64.f32 d1, s15`. *Entonces, el primer argumento (la cadena de formato) se pasa a través del registro* `r0` *y el argumentos de punto flotante a través de los registros* `r2` *y* `r3` *(una concatenación de valores usando la instrucción* `vmov r2, r3, d1`*). La pregunta que debería surgir es la siguiente: ¿por qué no se utilizó el registro* `r1` *y el registro* `r2`*? La respuesta es porque la convención de llamada establece que se utilicen registros consecutivos empezando por el siguiente registro con índice par, en este caso el registro* `r2`.

Referencias

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

[2] William Hohl, Christopher Hinds, *ARM Assembly Language: Fundamentals and Techniques*, Segunda edición, CRC Press, 2015.

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

[4] ARM Architecture Reference Manual, 2005.

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

[6] Digital design and computer architecture ARM Edition, Harris, Sarah L & Harris, David, Morgan Kaufmann, 2015.

[7] ARMv7-M Architecture Reference Manual, ARM Limited, 2021.