Conceptos básicos del ensamblador y arquitectura ARM

Leonardo Scandolo
Diego Feroldi

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

[*] Actualizado 17 de noviembre de 2022 (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

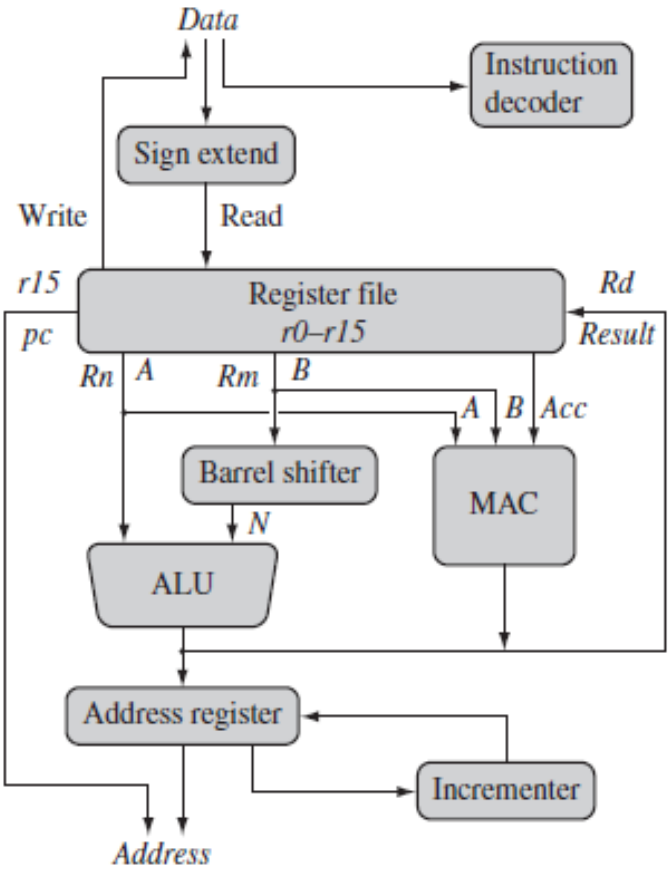


Figura 1: Arquitectura ARM tipo Von Neumann [1].

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. 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 1.3. 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.

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

- `r9` en algunas arquitecturas tiene un uso especial.
- `r0` a `r3` son *caller-save*.
- `r4` a `r11` son *callee-save*.
- `r12`, llamado también `ip` (*Instruction Pointer*), es usado por el linker si es necesario para hacer saltos largos y poder usar todo el espacio de memoria de 32 bits.
- `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 7.
- `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	Uso	¿Preservado?
`R0`	Argumento / valor de retorno / variable temporal	No
`R1-R3`	Argumentos / variables temporales	No
`R4-R11`	Variables salvadas	Sí
`R12`	Variable temporal	No
`R13 (SP)`	*Stack Pointer*	Sí
`R14 (LR)`	*Link Register*	Sí
`R15 (PC)`	*Program Counter*	—

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 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).

1.3. CPSR

El registro de status (*Current Program Status Register*, CPSR) 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 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.

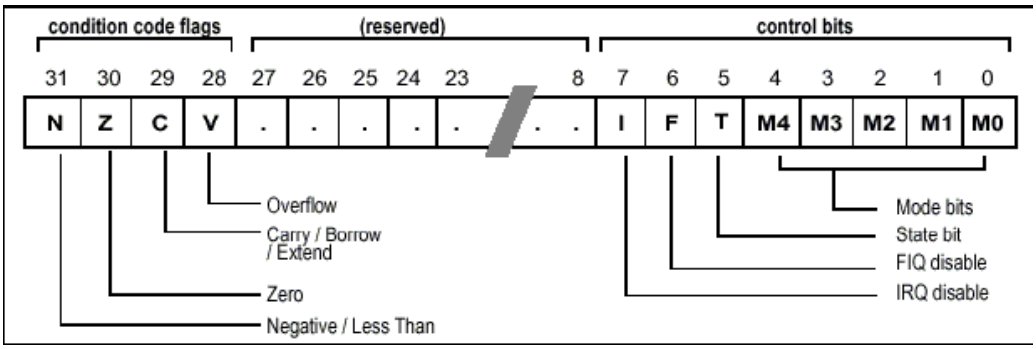


Figura 2: Estructura del registro CPSR.

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 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.

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

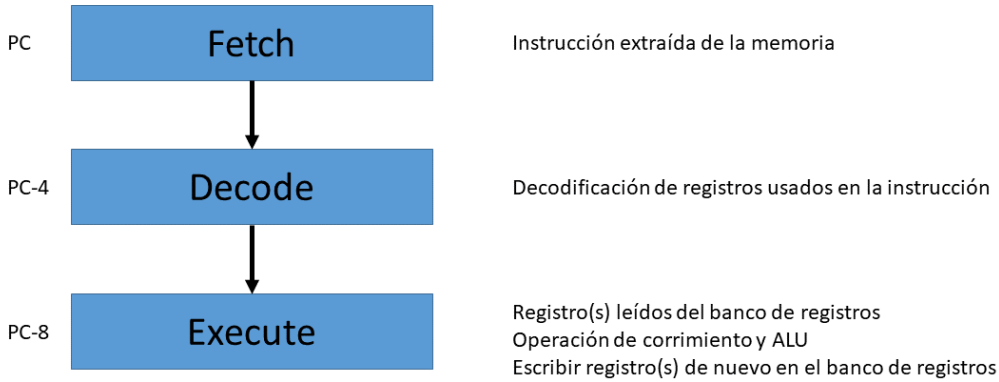


Figura 3: *Pipeline* típico de tres etapas.

1.4. Pipeline

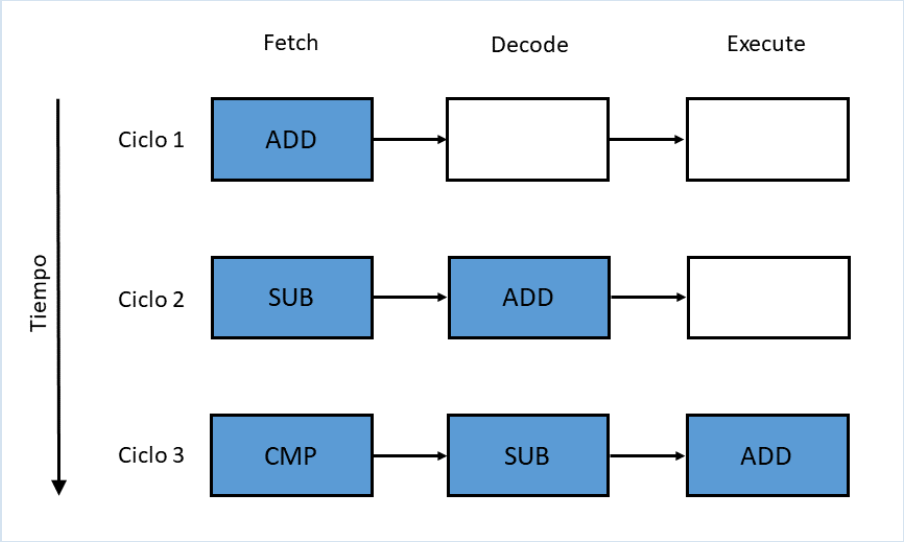
Pipeline es una técnica utilizada en el diseño de procesadores ARM (y otros) para aumentar el rendimiento de las instrucciones. En lugar de tener que buscar una instrucción, decodificarla y luego ejecutarla, se pueden hacer las tres tareas al mismo tiempo pero no sobre la misma instrucción. Este concepto 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

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:

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

- 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 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 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 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 r0, #0`
  Copiar el valor 0 a r0.
- `MOV r7, r5`
  Copiar el valor de r5 a r7
- `MVN r1, r0`
  r1 = NOT(r0).

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

  `MOVW` (*Move Wide*) copia una constante de 16 bits en un registro mientras pone a cero los 16 bits superiores del registro objetivo. Esta instrucción está disponible en núcleos ARMv7 y superiores. `MOVW` puede usar cualquier valor que se pueda expresar como un número de 16 bits.

- `MOVW r0, #0xface`
  `MOVT r0, #0xfeed`

  El resultado será `r0=0xfeedface`.

  `MOVT` (*Move Top*) copia una constante de 16 bits en la parte superior de un registro, dejando intacta la mitad inferior. Esto está disponible en núcleos ARMv7 y posteriores.

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 r0, r1, r2`
  Suma `r1` y `r2` y guarda el resultado en `r0`.
- `ADC r0, r1, r2`
  Suma con acarreo, `r0:=r1+r2+carry`.
- `SUB r5, r3, r0`
  `r5 := r3 - r0`.
- `SBC r5, r3, r0`
  Resta con acarreo, `r5:=r3-r0-NOT(carry)`.
- `RSB r0, r0, r1`
  Resta inversa, `r0:=r1-r0`.
- `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`×`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. Hay cuatro operaciones lógicas: AND, NOT, OR y XOR.

- ```
  MOV r7, #0xff
  MOV r2, #0xf
  AND r8, r7, r2        @ r8 = 0xf
  ```
- ```
  MOV r1, #0x16
  ORR r11, r11, #1      @ r11 = 0x17
  ```
- ```
  MOV r1, #0x16
  BIC r11, r11, #2      @ r11 = 0x14 (r11 &= ~1, Clear bit a bit)
  ```
- ```
  MOV r1, #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*. Hay cuatro instrucciones, todas ejecutando una lógica ADD, SUB, OR y AND.

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 (modifica las banderas del CPSR).

- `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*) le dice al contador del programa actual que la siguiente instrucción estará en la dirección `<label>`:

  `B{cond} 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
        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.*

- `BL` (*Branch with Link*) salta de la misma forma que la instrucción B:

  `BL{cond} label`

pero luego `r14 := dirección de la próxima instrucción` y `r15 := label`. Es decir, el `pc` se cambiará 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          @ Regresar a donde estábamos
```

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 Exchange*) 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.

  `BX{cond} Rm`

  Luego `r15 := Rm` y además cambia a modo THUMB si `Rm[0]` es 1 y se mantiene en modo ARM si `Rm[0]` es 0.

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
```

- `BLX` (*Branch with Link and Exchange*) es como 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.

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

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.

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	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

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

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

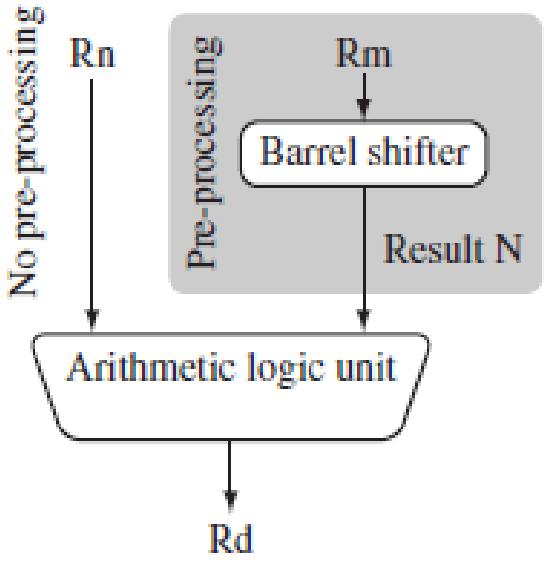


Figura 4: *Barrel Shifter* y ALU [1].

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, #0xF     @ r0 = 1111  y CPSR=0x400f0010
EOR  r0, #2       @ 1111 EOR 0010 = 1101 (no modifica el CPSR)
EORS r0, #2       @ 1101 EOR 0010 = 1111 (CPSR=0xf0010)
```

Luego de ejecutarse la instrucción `EORS` *el bit 30 (*`Z`*) pasó de estar en 1 a 0.*

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. 4 muestra el *barrel shifter* y la ALU.

Las operaciones posibles son:

- **LSL** : *Logical Shift Left* (corrimiento lógico a la izquierda), mueve los bits a la izquierda, ingresando ceros por la derecha.
- **LSR** : *Logical Shift Righ*t (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` $\ll 2$)
- *MOV r0, r1, ROR #2*
  `r0 := r1`$/4 =$ (`r0` $\gg 2$)
- *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 `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 `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.

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. 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:

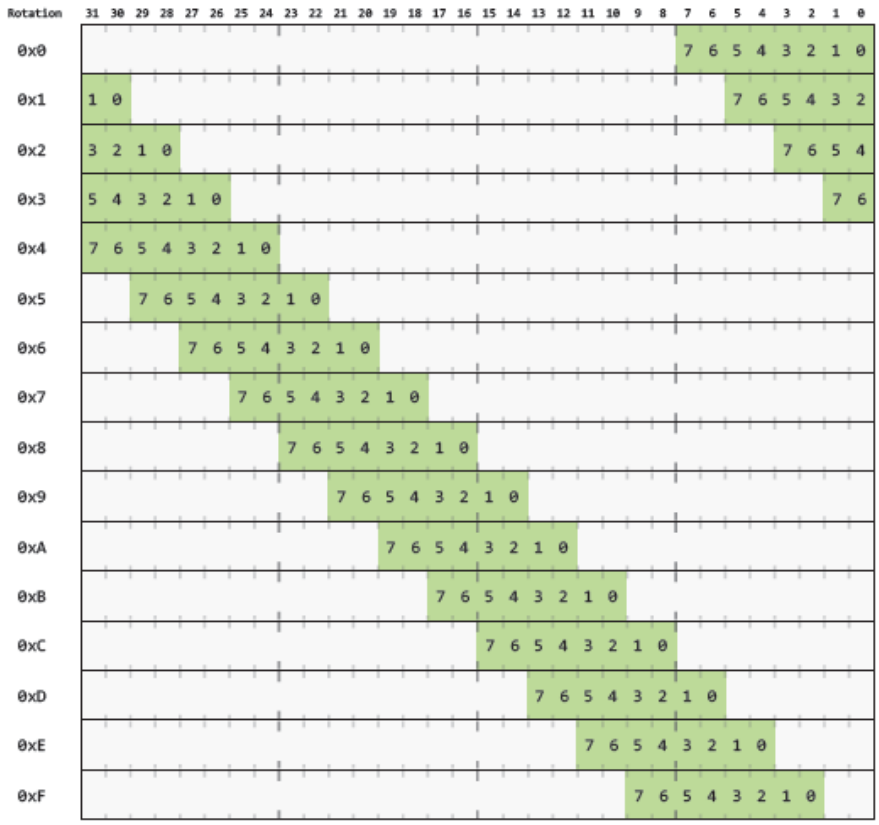
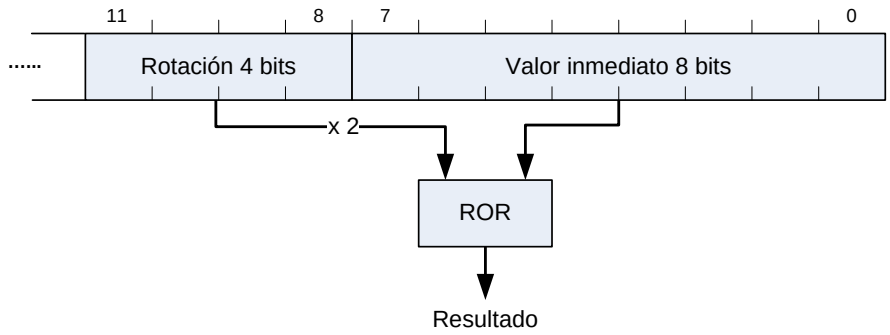


Figura 5: Esquema de posible rotaciones.



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. 5 muestra todas las posibles rotaciones.

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`)

- `0xF000F000` × (La constante rotada es demasiado grande)
- `0x00000123` × (La constante rotada es demasiado grande)
- `0x0F0000F0` × (La constante rotada es demasiado grande)

Ejemplos

A continuación vemos algunos 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 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 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:*

```
ldr  r1, =num
ldr  r0, [r1]
```

6. 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. 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.

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`. Esto obtiene la dirección de los datos, no los datos en sí. Similar al operador `&` en C/C++.

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

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
        ldr   r1, =x        @ r1 <-- dirección de x
                            @ int* r1 = &x; in c++

        ldr   r2, [r1]      @ r2 <-- valor en la dirección almacenada en r1
                            @ 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.

6.1. Operaciones de carga y guardado simples

Las operaciones de carga y guardado simples tienen la forma:

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

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* $\leftarrow r_d$

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:

[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 código.

Ejemplos

`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 $r1$.*

6.2. Operaciones de carga y guardado múltiples

Estas operaciones tienen la forma:

`opcode`[*modo*] 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.

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

- IA, significa *Increment After* (incrementar después): La dirección se incrementa luego de cargar/guardar cada registro

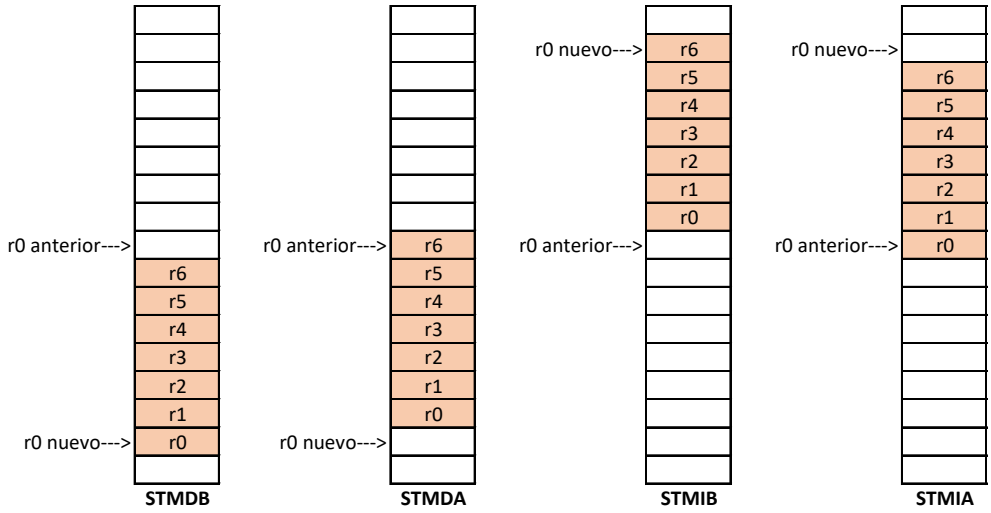


Figura 6: Ejemplo de diferentes tipos de almacenamientos con múltiples datos.

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

La Fig. 6 muestra el resultado de la instrucción `stm[modo] r0!, {r0-r6}` con 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:

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

Si el modo no se especifica, es por defecto modo *full descending*. 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}

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

- *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.*

7. 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.*

8. Llamada a función

Llamar y regresar de una función implica cinco fases.

1. Llamada a la función (realizada por la función llamada):
   - Cualquier dato en `r0-r3` que deba guardarse se empuja a la pila.
   - Los parámetros se colocan en `r0`, `r1`, `r2` y `r3` (en orden).
   - Usar `BL` para bifurcar a la función.
2. Prólogo de la función (hecho por función llamada)
   - Si la función utilizará alguno de los registros `r4-r11`, guardar los valores actuales en la pila.
3. Cuerpo de la función (hecho por la función llamada)
   - Realizar los cálculos necesarios. Acceder a los parámetros pasados en `r0-r3`. Solo modificar los registros `r0-r3` y cualquier registro que haya sido preservado en el prólogo.
4. Epílogo de función (hecho por función llamada)
   - Colocar el valor devuelto en `r0`.
   - Retire cualquier registro almacenado (`r4-r11`) de la pila para restaurar sus valores anteriores.
   - Retornar a la función llamante con `BX lr`.
5. Control de reanudación (realizado por la función llamante)
   - El valor devuelto está en en registro `r0`.
   - Restaurar registros almacenados (`r0-r3`) que se preservaron antes de llamar.

Ejemplo

En este ejemplo se muestra un programa que reemplaza `r4` *y* `r5` *con sus valores absolutos. Para hacer esto, utiliza una función* `abs`*. Para llamar a la función, la parte principal del programa guarda* `r1`*, que está en uso, luego coloca el parámetro (valor para tomar el valor absoluto) en* `r0`*. La función conserva* `r4` *y* `r5` *para poder usarlos y luego hace los cálculos necesarios. Termina colocando el resultado en* `r0`*, restaurando* `r4` *y* `r5` *y ramificando de nuevo. Luego, la parte principal del programa saca el valor devuelto de* `r0` *y restaura* `r1`*.*

```
/*
El programa está usando los registros 1, 4 y 5. Por lo tanto, es responsable de
 guardar/restaurar r1
*/

.text
.global main
main:
    MOV   r1, #0xAA
    MOV   r4, #30
    MOV   r5, #-12

    PUSH  {r1}          @ Se guarda r1
    MOV   r0, r4        @ Configurar parámetro para llamada a función
    BL    abs           @ Se llama a la función
    MOV   r4, r0        @ Salvar resultado a r4
    POP   {r1}          @ Restaurar r1

    PUSH  {r1}         @ Se guarda r1
    MOV   r0, r5       @ Configurar parámetro para llamada a función
    BL    abs          @ Se llama a la función
    MOV   r5, r0       @ Salvar resultado
    POP   {r1}         @ Restaurar r1

    BX     lr

abs:
    PUSH  {r4, r5}      @ Guarda registros calle saved
    MOV   r1, #0
    CMP   r0, r1        @ Compara parámetro en r0 con cero
    BGE   end           @ Si r0 es >= saltar
    MVN   r4, r0        @ Copiar su valor negado (bit a bit) a r4
    ADD   r5, r4, #1    @ Sumar 1 para tener el complemnto a dos en r5
    MOV   r0, r5        @ Valor de retorno en r0
end:
    POP   {r4, r5}      @ Restaurar registros
    BX    lr            @ Retornar
```

9. 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*)[3]. Estos coprocesadores vienen opcionalmente con algunos procesadores. Existen varias versiones de VFP, a saber:

[3]Punto flotante vectorizado

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 $a \leftarrow a + b \times c$).

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 (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:

$$d\left[x\right] \Longleftrightarrow \left\{s\left[(2x)+1\right], s\left[2x\right]\right\},$$

lo cual se grafica en la Fig. 7.

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.

9.1. 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)[4]. 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*). 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

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

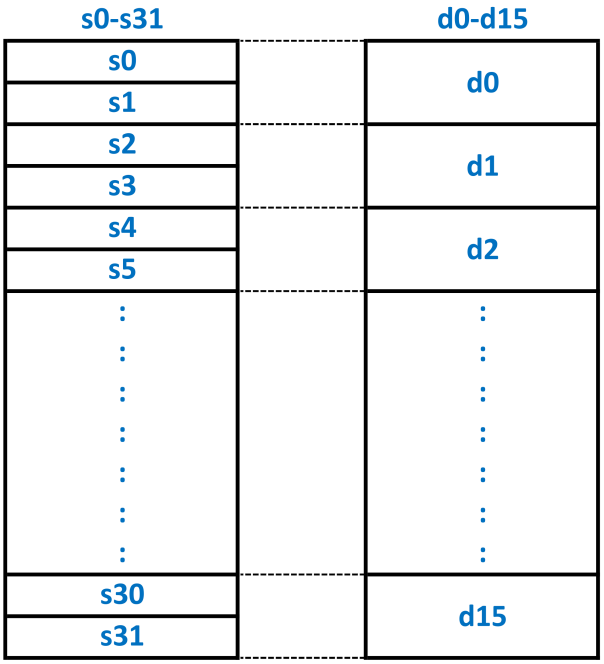


Figura 7: Relación entre registros de punto flotante.

Figura 8: Registro FPSCR.

números de punto flotante tienen signo. Otro cambio importante tiene que ver con el valor especial NaN (*Not a Number*)[5], 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

[5]No es un número.

9.2. Instrucciones *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:

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

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

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:

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

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:

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

9.3. 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`).

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

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

Observación

Los registros FPU podrían usarse como almacenamiento temporal para cualquier cantidad de 32 bits. Por ejemplo, podrían contener dos operandos de precisión media o un valor entero.

9.4. 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

9.5. 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.

9.6. Instrucciones para procesamiento de datos

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

`V<operation>{cond}.F32 {<dest>}, <src1>, <src2>`

Los registros `src1`, `src2` y `dest` pueden ser cualquiera de los registros de precisión simple, `s0` a `s31`, en el archivo de registro. 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`

9.7. Ejemplo general con instrucciones de punto flotante

Ejemplo

A continuación un ejemplo general que muestra varias instrucciones en punto flotante:

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

.text
.arm
.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
      vmul.f32 s2, s2, s2      @ s2=s2*s2=12.25

      vcvt.s32.f32 s0, s2      @ Conversión de punto flotante a entero (s0=12)
      vmov  r0, s0             @ Copia del registro s0 al registro r0 (r0=12)

      bx    lr                 @ Retorna r0=12
```

10. 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 `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 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:*

```
.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 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. 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.

Si se usan valores en punto flotante, compilar de la siguiente manera[6]:

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

11. 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 -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 (no usar el comando run!)
> info reg
> next
```

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.

[6] "*ARM hard float*" es la arquitectura ABI ARM de punto flotante.

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.