Apunte breve de ensamblador ARMV7-M (Cortex-M3).

Ferando Alberto Miranda Bonomi

2023-04-13

En este apunte breve se indican algunas instrucciones comunes del conjunto de instrucciones soportado por la familia de procesadores armv7-m a la que pertenece el Cortex-M3.

Referencias Importantes:

Guía de usuario genérica de los dispositivos Cortex-M3 (en línea, inglés). Es una referencia breve de la familia de procesadores Cortex-M3. Incluye descripciones de las instrucciones. Explica también el uso de periféricos propios del procesador Cortex-M3.

Manual de referencia de la arquitectura ARMV7-M (pdf, inglés). Manual de referencia que incluye una explicación detallada de la arquitectura ARMV7-M, que implementan los procesadores Cortex-M3. Aquí pueden encontrar explicaciones detalladas de las instrucciones con pseudocódigo. Incluye además la codificación en código máquina de las instrucciones de ensamblador. Nota: No todas las características expuestas son implementadas por los procesadores Cortex-M3. La unidad de punto flotante no es implementada en Cortex-M3, si en Cortex-M4 que tiene la misma arquitectura.

Manual del programador de STM32F10xxx/20xxx/21xxx/L1xxx (pdf,inglés). Es una referencia de ST Microelectronics para sus líneas de productos que incluyen procesadores Cortex-M3. Puede considerarse una reedición de la guía de usuario genérica particularizada para los productos de ST.

1 Operaciones entre registros; operaciones entre registros y constantes

Las operaciones entre registros se indican de la siguiente manera:

op{s} Rd, Rn, Rm{, desplazamiento #n}

Donde las partes encerradas en llaves son opcionales (las llaves *no son parte del código*), la terminación *s* indica que la operación debe afectar las banderas. El desplazamiento permite desplazar los bits del registro *Rm* a izquierda o derecha. Para más detalles consulta la sección 1.4.

Si *Rd* es el mismo registro que *Rm* y no se usa desplazamiento puede escribirse la forma reducida

op{s} Rd,Rm

Estas operaciones incluyen una variante de *operación entre registro y constante* donde en lugar de un segundo registro se da un valor numérico codificado como parte de la instrucción.

op{s} Rd,#constante

donde la constante es, dados dos dígitos hexadecimales X e Y, cualquier construcción de la forma

- `0x000000XY` desplazado a izquierda cualquier número entre 0 y 31 lugares
- `0xXY00XY00`
- `0x00XY00XY`
- `0xXYXYXYXY`

Para mayor información ver flexible second operand

1.1 Operaciónes aritméticas

Todas las operaciones de suma y resta afectan las banderas Z,C,N,V si se usa la terminación *s*.

Para mayor información referirse a ADD, ADC, SUB, SBC, and RSB en la guía de usuario genérica de Cortex-M3.

1.1.1 Suma: *ADD*

```
Ejemplos:
    ADD R2,R1,#234      // R2 ← R1 + 234
    ADD R2,R2,R3        // R2 ← R2 + R3
    ADD R2,R3           // versión corta de la anterior
    ADD R2,R2,R3,lsl #4 // R2 ← R2 + (R3 lsl 4). No tiene versión corta
```

1.1.2 Resta *SUB*

```
Ejemplos:
    SUB R3,#127          // R3 ← R3 - 127
    SUB R2,R3            // R2 ← R2 - R3
    SUBS R1,R2,R3,asr#3 // R1,banderas ← R2 - (R3 asr 3)
```

1.1.3 Suma y resta con constante de 12 bit

Las operaciónes *ADD* y *SUB* cuentan además con una forma de registro + constante donde la constante es de un número de 12 bits

ADD Rd{,Rn},#imm12
SUB Rd{,Rn},#imm12

donde imm12 es cualquier valor entre 0 y 4095. Se puede omitir *Rn* si es el mismo que *Rd*. Nota: Esta forma es solo válida para las operaciones *ADD* y *SUB*

```
Ejemplo:
    ADD R0,#1942         // R0 ← R0 + 1942 // Forma ADD Rd{,Rn},#imm12
    SUB R1,R3,#3000      // R1 ← R3 - 3000 // Forma SUB Rd{,Rn},#imm12
```

1.1.4 Suma con acarreo: *ADC*

Nota: C es la bandera de acarreo.

Notación: Si un número de 64 bits está almacenado en dos registros de modo que su *parte alta* está en R1 y su parte baja en R2 esto se escribe a continuación como `R1:R2`.

```
ej_suma_64:              // R1:R0 ← R1:R0 + R3:R2
    adds r0,r2           // R0,banderas ← R0 + R2
    adc r1,r3            // R1 ← R1 + R3 + C
    bx lr
```

1.1.5 Resta con llevo *SBC*

Nota: C es la bandera de acarreo, luego de una resta vale 0 si ocurrió un pido (borrow).

```
ej_resta_64:             // R1:R0 ← R1:R0 - R3:R2
    subs r0,r2           // R0,banderas ← R0 - R2
```

```
    sbc r1,r3           // R1 ← R1 - R3 - 1 + C
    bx lr
```

1.1.6 Comparación *CMP*

Nota: resta donde el resultado es descartado y solo se afectan las banderas. Para mayor información referirse a CMP and CMN en la guía de usuario genérica de Cortex-M3.

```
Ejemplo:
    CMP R1,#2       // banderas ← R1 - 2
    CMP R4,R2       // banderas ← R4 - R2
```

Comparación con negativo *CMN*. Suma donde el resultado es descartado y solo se afectan las banderas.

```
Ejemplo:
    CMN R4,R2       // banderas ← R4 + R2
```

1.2 Operaciones lógicas

Las operaciones lógicas, si se usa la terminación *s*, afectan las banderas Z,N,C. En el caso de C será afectada si se usa un desplazamiento opcional en el operando Rm, el último bit desplazado fuera del valor se guardará en el carry.

Para mayor información referirse a AND, ORR, EOR, BIC, and ORN en la guía de usuario genérica de Cortex-M3.

1.2.1 Conjunción ('Y' lógico) bit a bit *AND*

```
Ejemplos:
    AND R3,#0x04000000  // R3 ← R3 and 0x04000000
                        // pone en 0 todos los bits de R3 excepto el bit 26
    ANDS R0,R1,R3,lsl#8 // R0,banderas ← R1 and (R3 lsl 8)
    AND R2,R1           // R2 ← R2 and R1
```

1.2.2 Disyunción ('O' lógico) bit a bit *ORR*

```
Ejemplo:
    ORR R3,#0x00001000  // R3 ← R3 or 0x00001000
                        // pone en uno el bit 12 de R3
    ORR R2,R4           // R2 ← R2 or R4
```

1.2.3 Discrepancia ('O' exclusivo) bit a bit *EOR*

```
Ejemplo:
    EOR R1,#0xFF00FF00  // R1 ← R1 xor 0xFF00FF00
                        // invierte los bits 8 a 15 y 24 a 31 de R1
    EOR R0,R3           // R0 ← R0 xor R3
```

1.2.4 Y bit a bit con segundo argumento negado *BIC* (BIt Clear, por ser usado para borrar setear bits a 0)

```
Ejemplo:
    BIC R0,#0x08000000  // R0 ← R0 and (not 0x08000000)
                        // pone en cero el bit 27 de R0
    BIC R2,R4           // R2 ← R2 and (not R4)
```

1.2.5 O bit a bit con el segundo operando negado *ORN*

```
Ejemplo:
    ORN R1,#0x00FF0000  // R1 ← R1 or (not 0x00FF0000)
                        // pone en uno todos los bits de R1
                        // excepto los bits 16 a 23
    ORN R2,R4           // R2 ← R2 or (not R4)
```

1.2.6 Prueba de bits *TST*

Nota: es una operación Y bit a bit donde se descarta el resultado y se afectan las banderas. Para mayor información referirse a TST and TEQ en la guía de usuario generica de Cortex-M3.

```
Ejemplo:
    TST R2,#0x01000000  // banderas ← R2 and 0x01000000
                        // Z=0 si el bit 24 de R2 es '1',
                        // de lo contrario Z=1
    TST R1,R3           // banderas ← R1 and R3
```

1.2.7 Prueba de igualdad *TEQ*

Nota: es una operación O exclusivo bit a bit donde se descarta el resultado y se afectan las banderas. A diferencia de *CMP*, esta comparación no afecta la bandera de desborde V, y si no se usa desplazamiento tampoco la bandera de acarreo.

```
Ejemplo:
    TEQ R2,#0xFF00FF00  // banderas ← R2 xor 0xFF00FF00
                        // Z=0 si R2 es igual a 0xFF00FF00
    TEQ R3,#0x10000000  // banderas ← R3 xor 0
                        // N=0 si R2 es negativo, Z=0 si R2
                        // es el mínimo del complemento a 2
    TEQ R4,R5           // banderas ← R4 xor R5
```

1.3 Operaciones de copia y desplazamiento

Copian un registro sobre otro, aplicando posiblemente un desplazamiento. Si se usa la terminación *s* afectan las banderas N,Z y C (esta última solo si se usó desplazamiento).

Para mayor información referirse a MOV and MVN y MOVT en la guía de usuario genérica de Cortex-M3.

1.3.1 Copiar un registro en otro *MOV*

Nota: cuando se usa desplazamiento es *preferible* usar la forma de desplazamiento

desplazamiento{s} Rd,Rm,#n

Además de la forma alternativa de *MOV*, éstas instrucciones incluyen una variante donde el *desplazamiento es controlado por un registro.*

desplazamiento{s} Rd,Rm,Rs

Para mayor información sobre las instrucciones de desplazamiento consultar ASR, LSL, LSR, ROR, and RRX en la guía de usuario genérica de Cortex-M3.

```
Ejemplos:
    MOV R3,R5           // R3 ← R5
    MOV R2,R3,asr#5     // R2 ← R3 asr 5
    ASR R2,R3,#5        // R2 ← R3 asr 5, versión preferible
    ROR R1,R2,R3        // R1 ← R2 ror R3
```

1.3.2 Copiar la negación bit a bit de un registro en otro *MVN*

```
Ejemplos:
    MVN R1,0x000FF000   // R1 ← (not 0x000FF000)
                        // R1 valdrá luego 0xFFF00FFF
    MVN R3,R5           // R3 ← (not R5)
    MVN R2,R3,asr#5     // R2 ← (not R3) asr 5
```

1.3.3 Inicializar un registro con un valor constante *MOV* y *MOVT*

La instrucción *MOV* tiene una variante que acepta una constante de 16 bit.

MOV Rd,#imm16

donde imm16 es un número entre 0 y 65535.

Existe además una versión complementaria *MOVT* que toma un entero de 16 bit y lo copia solo *sobre la parte más significativa del registro* sin afectar los 16 bit menos significativos.

MOVT Rd,#imm16

donde imm16 es un número entre 0 y 65535.

La combinación de estas dos instrucciones permite inicializar un registro con cualquier valor de 32 bit.

Nota: Usar siempre *MOVT* luego de *MOV* puesto que *MOV* sobreescribe los bits más significativos.

```
Ejemplo_de_uso:
    MOV R0,0x5678       // R0 ← 0x5678
    MOVT R0,0x1234      // R0(31..16) ← 0x1234
                        // Luego de estas dos instrucciones R0 vale 0x12345678
```

1.4 Modos de desplazamiento

Aquí se indica el efecto que tienen los distintos modos de desplazamiento.

Desplazamiento lógico a la izquierda *LSL #n*. Desplaza n bits a la izquierda introduciendo ceros por la derecha

```
Ejemplo:
    MOV R0,#0x4321
    MOVT R0,#0x8765     // R0 ← 0x87654321
    LSL R1,R0,#4        // Luego R1 vale 0x76543210
```

Desplazamiento lógico a la derecha *LSR #n*. Desplaza n bits a la derecha introduciendo ceros por la izquierda. Equivale a dividir por 2^n

```
Ejemplo:
    MOV R0,#0x4321
    MOVT R0,#0x8765     // R0 ← 0x87654321
    LSR R1,R0, #4       // Luego R1 vale 0x08765432
```

Desplazamiento aritmético a la derecha *LSR #n*. Desplaza n bits a la derecha introduciendo por la izquierda copias del bit de signo (bit más significativo). Equivale a dividir por 2^n con signo.

Ejemplo:

```
Ejemplo:
    MOV R0,#0x4321
    MOVT R0,#0x8765     // R0 ← 0x87654321
    ASR R1,R0,#4        // Luego R1 vale 0xf8765432
    MOV R2,#0x5678
    MOVT R2,#0x1234     // R2 ← 0x12345678
    ASR R3,R2,#4        // Luego R3 vale 0x01234567
```

Rotación a la derecha *ROR #n*. Rota los bits del registro n posiciones a la derecha. Desplaza los bits a la derecha ingresando por el extremo izquierdo los bits que salen por el derecho.

```
Ejemplo:
    MOV R0,#0x4321
    MOVT R0,#0x8765     // R0 ← 0x87654321
    ROR R1,R0,#4        // Luego R1 vale 0x18765432
    ROR R3,R0,#16       // Luego R3 vale 0x43218765
```

Rotación extendida un bit a la derecha *RRX*. Desplaza los bits del registro una posición a la derecha, introduce por la izquierda el bit de la bandera de carry y coloca en la bandera de carry el bit desplazado del extremo derecho.

```
Ejemplo:
    MOV R0,#1
    MOVT R0,#0x8000     // luego R0 ← 0x80000001
    cmn R0,#0           // banderas ← R0 + 0
                        // Luego C=0
    RRX R1,R0           // Luego C=1, R1=0x40000000
    RRX R1,R1           // Luego C=0, R1=0xA0000000
```

2 Operaciones de salto

Las operaciones de salto permiten modificar el valor del contador de programa para continuar la ejecución del código en una posición diferente a la que correspondería en el flujo normal de ejecución del programa. Estas instrucciones. De especial interés son las instrucciones de salto condicional, pues son la principal forma de control de flujo en los programas de lenguaje ensamblador.

Si bien cualquier instruccion del conjunto thumb y thumb2 soportado por armv7-m puede convertirse en condicional usando un bloque if-then (ver IT en la guía de usuario genérica Cortex M3), la única instrucción que puede ser condicional en sí misma es la instrucción de salto *B*.

Para mayor información consultar la sección B, BL, BX, and BLX en la guía de usuario genérica Cortex M3.

2.1 El salto *B* y el salto condicional *B{cond}*

La instrucción *B* permite hacer un salto con un desplazamiento constante que es codificado como parte de la instrucción. Normalmente el desplazamiento es calculado por el ensamblador en forma automática y solo debemos indicar la etiqueta donde deseamos que sea el destino del salto. Tiene la forma

B{cond} etiqueta

Donde *cond* es una condición opcional (las llaves no se escriben, solo indican que cond es opcional).

```
Ejemplo:            // Lazo con 130 repeticiones
    MOV R0,#130
0:                  // esta es una etiqueta local especial
    SUBS R0,#1      // R0, banderas ← R0 - 1
    BNE 0b          // Si el resultado de la suma es distinto de 0
                    // Salta a la etiqueta 0 más próxima hacia atrás
```

2.1.1 Condiciones de ejecución

Una instrucción de ejecució condicional tiene efecto si se cumple la condición indicada, de lo contrario su ejecución carece de efecto. Las condiciones de ejecución se indican por sufijos como se detalla a continuación:

2.1.1.1 Igualdad *EQ*

La instrucción se ejecuta si la bandera Z vale 1. Si la última operación que modificó las banderas fue una resta (SUBS, CMP) entonces la condición se cumple siempre que minuendo y sustraendo hayan sido *iguales*.

```
Ejemplo:
    CMP R1,R2   // Compara R1 y R2
    BEQ 1f
    MOV R0,R3   // Esto se ejecuta solo si R1 es distinto a R2
1:
    BX LR       // Si R1 es igual a R2 la ejecución continúa en este punto
```

2.1.1.2 Desigualdad *NE*

La instrucción se ejecutará si la bandera Z vale 0. Si la última operación que modificó las banderas fue una resta, esta condición se cumplirá siempre que minuendo y sustraendo hayan sido *números distintos*.

```
Ejemplo:
    CMP R1,R2   // Compara R1 y R2
    BNE 1f
    MOV R0,R3   // Esto se ejecuta solo si R1 es IGUAL a R2
1:
    BX LR       // Si R1 es distinto de R2 la ejecución continúa en este punto
```

2.1.1.3 Comparación con signo, *LT*, *LE*, *GE*, *GT*

Las condiciones de comparación sin signo permiten actuar en función de la relación entre dos números interpretados en código complemento a dos. Las condiciones son nombradas considerando que la última operación en afectar las banderas fue una resta (CMP o SUBS). Para lo que sigue consideremos que la operación fue `CMP R1,R0`

- *LT* less than, menor qué. La instrucción se ejecuta si R1 es menor que R2. Si esto ocurre la bandera N (negativo) es distinta de V (desborde). Si R1 y R0 tienen igual signo la operación no desborda nunca (V=0). Si el resultado es negativo (N=1) significa que R1 es menor que R0. Si tienen distinto signo puede ocurrir que haya un desborde. En ese caso V=1 (desborde), entonces si N=0 (positivo) el resultado fue número menor que el mínimo representable y R1 es menor que R0.
- *LE* less than or equal, menor qué o igual. La instrucción se ejecuta si R1 es menor o igual a R0. Si esto ocurre la bandera N es distinta a V (menor) o bien Z=1 (igual).
- *GE* greater than or equal, mayor qué o igual. La instrucción se ejecuta si R1 es mayor o igual a R0. Si esto ocurre el valor de la bandera N (negativo) es igual a V (desborde). Si R1 y R0 son del mismo signo V=0 siempre, si R1 es mayor o igual a R0 el resultado será positivo (N=0). Si R1 y R2 son de distinto signo puede ocurrir que haya un desborde (V=1) en ese caso si N=1 en realidad el resultado fue mayor que el máximo representable (y positivo).
- *GT* greater than, mayor qué. La instrucción se ejecuta si R1 es estríctamente mayor que R0. Si esto ocurre entonces N=V (caso GE) y Z=0 (no son iguales).

```
// int32_t maximo(int32_t a, int32_t b);
maximo:
    CMP R1,R0
    BLE 1f
    MOV R0,R1   // Continúa aquí si R1 es el máximo, copiándolo a R0
1:
    BX LR       // Salta aquí si R1 <= R0, osea que R0 es el máximo
```

2.1.1.4 Comparación sin signo, *LO*, *LS*, *HS*, *HI*

Las condiciones de comparación sin signo permiten actuar en función de la relación entre dos números interpretados en código binaro natural (esto es, sin signo). Las condiciones son nombradas considerando que la última operación en afectar las banderas fue una resta (CMP o SUBS). Para lo que sigue consideremos que la operación fue `CMP R1,R0`

- *LO* lower, inferior. La instrucción se ejecuta si C=0 (la resta generó un llevo o borrow). Esto ocurre cuando R1 es menor que R0.

- *LS* lower or same, menor o igual (lit. inferior o el mismo, para que las letras sean distintas a la comparación con signo). La instrucción se ejecuta si C=0 (hubo llevo, R1 menor que R0) o Z=1 (el resultado fue cero, son iguales).
- *HS* higher or same, mayor o igual (lit. superior o el mismo, para distinguirlo de la comparación con signo). La instrucción se ejecuta si C=1 (no hubo llevo, R1 no es menor que R0).
- *HI* higher, mayor. La instrucción se ejecuta si C=1 (no hubo llevo, R1 no es menor que R0) y Z=0 (el resultado no fue cero, así que los números son distintos).

```
// notar los tipos sin signo uint32_t
// uint32_t maximo(uint32_t a, uint32_t b);
maximo:
    CMP R1,R0
    BLS 1f
    MOV R0,R1   // Continúa aquí si R1 es el máximo, copiándolo a R0
1:
    BX LR       // Salta aquí si R1 <= R0, osea que R0 es el máximo
```

2.2 Llamada a subrutina con *BL* y retorno de subrutina terminal con *BX LR*

Para hacer un llamado a subrutina hay que mantener una dirección de retorno de modo que el programa principal pueda continuar su ejecución al retornar. La instrucción *BL* (branch with link) guarda la dirección de la siguiente instrucción del programa principal en el registro LR (link register). La instrucción *BX* (branch indirect) realiza un salto a la dirección almacenada en un registro, luego *BX LR* permite retornar al punto siguiente a un llamado a subrutina si no se ha modificado *LR* en el camino. Esto ocurre naturalmente en una subrutina que no llame a su vez a ninguna otra (llamada terminal u hoja). Si una subrutina no es terminal entonces deberá guardar el valor del registro LR en memoria (usualmente en la pila) para luego recuperarlo y poder retornar al punto siguiente del llamado.

```
// int32_t max2(int32_t a,int32_t b)
max2:
    cmp R1,R0
    ble 1f
    mov R0,R1
1:
    bx lr       // es una rutina terminal y no modifica LR, por tanto retorna
                // con BX LR

// int32_t max3(int32_t a,int32_t b,int32_t c)
max3:
```

```
    push {R4,LR}    // Guarda en la pila R4 y LR
    mov R4,R2       // Guarda el tercer argumento en R4, que no puede ser
                    // modificado por una subrutina
    bl max2         // R0 y R1 son los argumentos de max2, el resultado queda en R0
    mov R1,R4
    bl max2         // Llama nuevamente a max2, ahora con el máximo de los
                    // dos valores anteriores y el valor restante.
                    // el máximo queda en R0, listo para retornar.
    pop {R4,PC}     // Recupera R4 y carga el valor guardado de LR en
                    // el contador de programa (PC).

main:
    mov R0,#120     // R0 ← 120
    mov R1,#50      // R1 ← 50
    mov R3,#230     // R2 ← 230
    bl max3         // Llama a subrutina max3, copiando en el registro LR
                    // la dirección de la instrucción siguiente (mov R1,R0)
    mov R1,R0       // Al retornar de max3, R0 contiene el máximo
    ldr R0,=mensaje
    bl printf       // Muestra mensaje
    b .             // Lazo infinito

.pool
mensaje: .asciz "El máximo es %d\n"
```

3 Operaciones de acceso a memoria

El acceso a memoria puede ser de lectura o carga, donde datos de memoria son copiados en registros del procesador, y escritura o almacenaje, donde datos de los registros del procesador son copiados a memoria. Si bien el bus de datos es de 32 bits, la granularidad del acceso a memoria es de 8 bits. Esto significa que cada dirección de memoria distinta se refiere a una posición de 8 bits de datos. Existen instrucciones de acceso de 8 bit, 16 bit, 32 bit y múltiplos de 32 bit.

Las direcciones en memoria son de 32 bit y se refieren a 8 bit de datos, por lo que pueden direccionarse $2^{32} \times 8$ bit o 4 GiB. No todos los datos direccionables están implementados físicamente. En el caso de STM32F103C8 por ejemplo, solo están implementados 64 KiB de memoria de programa a partir de la dirección 0x08000000 y 20 KiB de memoria de datos SRAM a partir de la posición 0x20000000. Además de la memoria de programa y de datos, en el espacio de direcciones hay posiciones especiales donde se ubican los registros de los distintos periféricos del microcontrolador. A partir de la dirección 0x40000000 se ubican los registros

de los periféricos del microcontrolador y entre 0xE0000000 y 0xE00FFFFF se encuentran los registros de los periféricos internos de la plataforma Cortex-M3 como el timer SysTick y el controlador de interrupciones.

3.1 Modos de Direccionamiento

Las direcciones de memoria son de 32 bits mientras que las instrucciones en código máquina tienen un ancho de 16 bit o 32 bit. Luego es imposible especificar directamente una dirección de memoria absoluta en una instrucción. En su lugar se usa *direccionamiento indirecto*. La dirección de memoria es formada por una dirección base, almacenada en un registro, y un desplazamiento (offset) opcional a partir de dicha dirección. Existen también modos pre-indexado y post-indexado que modifican el valor del registro de dirección antes o después del acceso a memoria, sumando una constante. En la guía de usuario de Cortex M3, los modos con offset inmediato, pre-indexado y post-indexado son descritos en LDR and STR, immediate offset, los modos de direccionamiento con offset en registro son descritos en LDR and STR, register offset.

3.2 Operaciones con Base y Offset Inmediato

Este modo de direccionamiento puede usarse en operaciones de carga y almacenamiento de 8, 16, 32 y 64 bits (ver para más información). Tienen la forma

op{tipo} Rt, [Rn,{, #offset}]
opD Rt, Rt2, [Rn,{, #offset}]

donde *op* es STR (almacenaje, registro a memoria) o LDR (carga, memoria a registro). El sufijo indica la cantidad de bits de la operación. Los registros indicados como *Rt* y *Rt2* son los registros de datos. El registro indicado como *Rn* es el registro de dirección base. El offset es un valor entre -255 y 4096 excepto para la variante D, en cuyo caso es un número *múltiplo de 4* entre -1020 y 1020.

Las variantes según el sufijo son

sin sufijo: Transferencia de 32 bits.
sufijo *B* (byte): Transferencia de 8 bits sin signo. La extensión de 8 a 32 bit en la operación de carga (*LDRB*) se realiza poniendo a cero bits 31 a 8.
sufijo *SB* (signed byte): Transferencia de 8 bits con signo. La extensión de 8 a 32 bit en la operación de carga (*LDRSB*) se realiza copiando el bit de signo (bit 7) en los bits 31 a 8.
sufijo *H* (halfword): Transferencia de 16 bit sin signo. La extensión de 16 a 32 bit en la carga (*LDRH*) se realiza poniendo a cero los bits 31 a 16.
sufijo *SH* (signed halfword): Transferencia de 16 bits con signo. La extensión de 16 a 32 bit en la operación de carga (*LDRSH*) se realiza copiando el bit de signo (bit 15) en los bits 31 a 16. **sufijo *D* (double word)**: Transferencia de 64 bit. Los primeros 32 bit se cargan

o toman del registro *Rt* y los siguientes 32 bit se cargan o toman del registro *Rt2*. Deben especificarse registros diferentes para *Rt* y *Rt2*. En este caso *la dirección de memoria debe ser múltiplo de 4*.

```
Ejemplos:
    STR R3,[R4,#14]      // Guarda R3 en la posición dada por (R4+14)
    STRD R0,R1,[R3]      // Guarda R0 en la posición dada por R3
                         // y R1 en la posición (R3 + 4)
    LDRSB R1,[R4,#-44]   // Carga 8 bit de la posición dada por (R4-44)
                         // en R1(7..0) y extiende R1(7) en R1(31..8)
    LDRB R3,[R5,#23]     // Carga 8 bit de la posición dada por (R5+23)
                         // en R3(7..0) y pone en 0 R3(31..8)
    STRB R3,[R4]         // Almacena el byte R3(7..0) en la posición de
                         // memoria dada por R4
    LDRD R0,R1,[R1,#20] // Carga 32 bits de la posición (R1+20), que
                         // debe ser múltiplo de 4, en R0 y 32 bit
                         // de la posición (R1+24) en R1
    LDRH R2,[R3,#1000]   // Carga 16 bits de la posición (R3+1000) en
                         // R2(15..0) y pone en 0 R2(31..16)
```

3.2.1 Carga con direccionamiento relativo al contador de programa (PC)

Existe una forma especial de instrucción de carga con direccionamiento relativo al contador de programa. Esta forma es usualmente empleada para cargar constantes en registros desde la memoria de programa (ver la entrada en la guía de usuario de Cortex M3).

LDR{tipo} Rt, etiqueta
LDRD Rt, Rt2, etiqueta

Donde las variantes son las mismas que para el caso de base y offset inmediato, pero solo se permite operación de carga y la etiqueta debe referirse a una posición con un offset entre -4095 y 4095 para el caso de 8 a 32 bit o entre -1020 y 1020 para el caso de 64 bit.

Existe también una pseudoinstrucción (instrucción de ensamblador que no corresponde exactamente a una instrucción de máquina) que crea una constante en memoria de programa y emite la instrucción de carga a la vez.

LDR Rt,=valor

donde, valor es un valor de 32 bit, que genera una constante de 32 bit con el valor indicado en la memoria de programa y emite una instrucción con offset relativo al *PC* apuntando a dicha constante.

```
ejemplo:
    ldrd R0,R1,valor    // R0 ← 0x76543210, R1 ← 0xFEDCBA98
    ldr R2,=0x12345678  // R2 ← 0x12345678 (crea una nueva constante
                        // en memoria de programa)
    bx lr
valor:  // Constante en memoria de programa
    .word 0x76543210, 0xFEDCBA98
```

3.3 Operaciones pre-indexadas y post-indexadas

Estas operaciones son útiles para acceder de manera secuencial a arreglos en memoria (ver la entrada en la guía de usuario de Cortex M3). Tienen la siguiente forma

Pre-indexado:
op{tipo} Rt, [Rn, #offset]!
opD Rt, Rt2, [Rn, #offset]!
Post-indexado:
op{tipo} Rt, [Rn],#offset
opD Rt, Rt2, [Rn],#offset

Donde las variantes disponibles son las mismas que para las operaciones con base y offset inmediato, y el valor del offset en el caso de las variantes de 8 a 32 bits queda limitado al rango -255 a 255, manteniéndose para el caso de 64 bit el offset múltiplo de 4 entre -1020 y 1020.

Nota: Los registros *Rt* y *Rt2* deben ser distintos de *Rn* en esta modalidad (esta limitación no existe en el caso de direccionamiento con offset).

En las formas pre-indexadas se suma el offset al valor de *Rn* **modificando el valor del registro** y luego se accede a la posición de memoria resultante. En las formas post-indexadas **se accede a la posición indicada por *Rn* y luego del acceso se suma a *Rn* el offset**.

```
Ejemplos:
    STR R3,[R4,#14]!    // Guarda R3 en la posición dada por (R4+14)
                        // Y guarda en R4 el valor (R4+14)
    STRD R0,R1,[R3],#8  // Guarda R0 en la posición dada por R3
                        // y R1 en la posición (R3 + 4).
                        // Guarda en R3 el valor (R3 + 8).
    LDRSB R1,[R4,#-44]! // Carga 8 bit de la posición dada por (R4-44)
                        // en R1(7..0) y extiende R1(7) en R1(31..8).
                        // Guarda en R4 el valor (R4-44).
    LDRB R3,[R5],#23    // Carga 8 bit de la posición dada por R5
```

```
                        // en R3(7..0) y pone en 0 R3(31..8).
                        // Guarda en R5 el valor (R5 + 23).
```

3.4 Operaciones con offset en registro

Son operaciones de carga/almacenaje donde el offset es dado por el valor de un registro al que opcionalmente se le aplica un desplazamiento a la derecha (multiplicación por potencia de dos) (ver para más información). El desplazamiento permite conseguir el offset en bytes a partir del índice de un arreglo siempre que el tamaño de los elementos del arreglo, en bytes, sea potencia de dos (por ejemplo, un arreglo de enteros de 32 bit). La forma general es:

op{tipo} Rt,[Rn,Rm{,LSL #n}]

Donde la operación *op* puede ser carga (LDR) o almacenaje (STR). Los sufijos de tipo son los mismos que para las operaciones con offset inmediato, excepto que solo soporta operaciones hasta 32 bit.

```
.global posicion_de_valor_en_arreglo
/**
 * Busca un valor en un arreglo de enteros de 32 bit sin ordenar
 * y devuelve su índice o -1 si no se encuentra el valor.
 * R0: nro de elementos en arreglo
 * R1: dirección base del arreglo de enteros de 32 bit.
 * R2: valor a buscar
 *
 * Retorna R0: índice del primer elemento con el valor dado
 *              o -1 si no se encuentra el valor
 */
posicion_de_valor_en_arreglo:
    MOV R12,R0              // Guarda el tamaño del arreglo
    MOV R0,#0               // Usa R0 como índice
0:
    CMP R0,R12
    BGE no_encontrado       // Se llegó al fin del arreglo sin encontrar nada
    LDR R3,[R1,R0,LSL #2]   // Carga en R3 el valor en la posición
                            // (R1 + R0 * 4) (elemento de índice R0)
    CMP R2,R3               // Compara con el elemento buscado
    BEQ fin                 // Si coincide, retorna el índice actual
    ADD R0,#1               // Si no, incrementa el índice
    B 0b                    // siguiente iteración del lazo de búsqueda
no_encontrado:              // No hubo coincidencias, poner R0 en -1
```

```
    MOV R0,#0
    SUBS R0,#1
fin:
    BX LR                    // Retorna al invocador
```

3.5 Operaciones de carga y almacenamiento múltiples

Las operaciónes de carga y almacenamiento múltiples permiten cargar/guardar varios registros en una sola operación (en este caso la entrada en la guía de usuario genérica de la plataforam cortex M3 podría ser confusa, se recomienda ver la sección correspondiente de armv7-m architecture reference manual, secciones A7.7.41, 42, 159 y 160). Estas instrucciónes sirven para implementar estructuras de memoria llamadas **pilas** de tipo LIFO (Last In First Out, lo último en entrar es lo primero en salir). También sirven para cargar/guardar tipos como los struct de C en una sola instrucción-.

op{modo} Rn{!},registros

Donde las llaves indican partes opcionales (y no se escriben en el código). La operación *op* puede ser *LDM* para carga múltiple o *STM* para almacenaje múltiple, *Rn* es un registro que contiene la dirección base desde la cual inicia la operación y *registros* es un listado de registros a cargar o almacenar. El sufijo *modo* indica como se ubican en memoria los datos a partir de la dirección base. Si se incluye el signo de admiración entonces *Rn* es actualizado con la útima dirección calculada, **en caso de actualizar *Rn* dicho registro no puede ser incluido en el listado a cargar/almacenar**. El puntero de pila, *SP*, no puede ser parte de la lista de registros en ningún caso. El contador de programa, *PC*, no puede estar en la lista para almacenamiento, y si está en la lista de carga se genera un salto.

Nota: El orden de carga de los registros es **ascendente por número de registro** y no depende de la posición del registro en la lista (se sugiere ordenar la lista para evitar confusión al interpretar el código).
Nota 2: El listado de registros **va escrito entre llaves en el código**, los registros se separan por comas y pueden escribirse rangos por ejemplo `LDM R0,{R0,R2-R5,R7}`.

Debido a la existencia de una doble nomenclatura y valores por defecto para el *modo* resulta conveniente hacer una tabla detallando todas las opciones para una mayor clarirad.

Instrucciónes	Acción	Posiciones de memoria	Actualización con !
LDM, LDMIA o LDMFD	Carga	$Rn, \dots, Rn + 4(N-1)$	$Rn \leftarrow Rn + 4N$
LDMDB o LDMEA	Carga	$Rn - 4N, \dots, Rn - 4$	$Rn \leftarrow Rn - 4N$

Instrucciónes	Acción	Posiciones de memoria	Actualización con !
STM, STMIA o STMEA	Guarda	$Rn, \dots, Rn + 4(N-1)$	$Rn \leftarrow Rn + 4N$
STMDB, STMFD	Guarda	$Rn - 4N, \dots, Rn - 4$	$Rn \leftarrow Rn - 4N$

Significado de los sufijos:

IA: Increment After access, incrementar luego del acceso. *DB*: Decrement Before access, incrementar antes del acceso. *FD*: Full Descending, Completo Descendente, un tipo de pila que crece "hacia abajo", el último dato ingresado está en la menor dirección, y el puntero apunta al último elemento ingresado. *EA*: Empty Ascending, Vacío Ascendente, un tipo de pila que crece "hacia arriba", el último dato ingresado está en la mayor dirección, y el puntero apunta a la primera dirección vacía.

Importante: Para implementar una pila descendente deben usarse *LDMFD* y *STMFD*. Notar que *LDMFD* es sinónimo de *LDM* pero *STMFD* **no es sinónimo** de *STM*. Algo similar ocurre para *LDMEA* y *STMEA*, por lo tanto en caso de usar las instrucciones para implementar pilas **usar solamente los prefijos *FD* o *EA*, y nunca omitirlos**. Por otra parte, para cargar/guardar tipos struct lo más conveniente es usar *LDM* y *STM*, puesto que almacenan/cargan los registros a partir de la dirección indicada y no hacia atrás.

```
/* En la pila ascendente de puntero vacío R0,
 * guardar el dato de 96 bit en R1 a R3,
 * retornar en R0 el nuevo puntero vacío
 *      ↓R0                                ↓R0
 * XXX  ___  ___  ___ ___   XXX #R1 #R2 #R3 ___
 * donde #Rn valor de Rn
 * ___ espacio libre
 */
.global push_3_ea:
push_3_ea:
    STMEA R0!,{R1-R3}
    BX LR

/* Sacar de la pila ascendente de puntero vacío R0
 * el último dato de 96 bit almacenado, retornarlo
 * en R1 a R3. Retornar en R0 el valor del nuevo
 * puntero vacío.
 *                 ↓R0        ↓R0
 * XXX #V1 #V2 #V3 ___   XXX  ___  ___  ___ ___
 * R1 ← #V1, R2 ← #V2, R3 ← #V3
```

```
 * ___ espacio libre
 */
.global pop_3_ea:
pop_3_ea:
    LDMEA R0!,{R1,R2,R3}
    BX LR

/* En la pila descendente de puntero a tope R0,
 * guardar el dato de 96 bit en R1 a R3,
 * retornar en R0 el nuevo puntero a tope.
 *                 ↓R0       ↓R0
 * ___ ___ ___ ___ XXX   ___ #R1 #R2 #R3 XXX
 * #Rn : valor del registro Rn
 * ___ espacio libre
 */
.global push_3_fd:
push_3_fd:
    STMFD R0!,{R1,R2,R3}
    BX LR

/* Sacar de la pila descendente de puntero a tope R0,
 * el último dato de 96 bit almacenado, retornarlo
 * en R1 a R3. Retornar en R0 el valor del nuevo
 * puntero a tope.
 *     ↓R0                          ↓R0
 * ___ #V1 #V2 #V3 XXX   ___ ___ ___ XXX
 * R1 ← #V1, R2 ← #V2, R3 ← #V3
 * ___ espacio libre
 */
.global pop_3_fd:
pop_3_fd:
    LDMFD R0!,{R1-R3}
    BX LR

/* Guarda un entero de 96 bits dado en R1..R3,
 * a partir de la dirección R0
 */
.global guarda_96b:
guarda_96b:
    STM R0,{R1-R3}
    BX LR
```

```
/* Carga un entero de 96 bits dado en R1..R3,
 * ubicado a partir de la dirección R0
 */
.global guarda_96b:
guarda_96b:
    LDM R0,{R1-R3}
    BX LR
```

4 La pila de ejecución o pila de llamados

La pila de ejecución es una estructura en memoria, una pila, que sirve para guardar datos relativos a la ejecución del programa en forma temporal. Es una especie de borrador para nuestro programa. Es una pila descendente que mantiene puntero al último elemento guardado, llamado *tope de pila*. La pila principal, que es la que usaremos por defecto, es inicializada en la posición más alta disponible de memoria SRAM. A partir de allí se usan las instrucciones *PUSH* y *POP*, que son un caso especial de *LDMFD* y *STMFD*, para guardar datos (PUSH) y recuperar los datos guardados (POP). Cuando se recuperan datos *se libera la memoria en la pila*, por lo tanto ya no están guardados. Solo se pueden recuperar datos previamente guardados, de lo contrario ocurre lo que se denomina desbordamiento, tomando datos inexistentes (ver la entrada en la guía de usuario de Cortex M3). Forma de uso:

PUSH registros
POP registros

Donde *registros* es un listado con los registros a guardar o recuperar. Los registros son escritos en orden según su número y no según su posición en la lista. La lista debe mantenerse en orden para evitar inconvenientes a la hora de leer el código. Las operaciones *POP* deben ocurrir en orden inverso a las operaciones *PUSH* y es conveniente usar las mismas listas de registros, con una excepción: Si se guarda la dirección de retorno con `PUSH {...,LR}` entonces puede usarse `POP {...,PC}` para restaurar los demás registros guardados y a la vez retornar el flujo del programa al invocador de la subrutina.

Nota: Para respetar el estándar de llamado a procedimiento, si una subrutina no es terminal (llama a su vez a otras subrutinas) entonces solo debe guardar en la pila *números pares de registros*. Además, en este caso siempre debe guardar *LR* y usualmente retorna con la última instrucción POP.

```
hola_mundo: .asciz "Hola Mundo!"
.global ejemplo
ejemplo:
    push {R4,LR}        // Preserva la dirección de retorno, y también R4
```

```
                    // para que sea un número par de registros
ldr R0,=hola_mundo  // Notar el '=', guarda en R0 la dirección de la etiqueta.
bl puts             // llamado a subrutina
pop {R4,PC}         // Recupera R4 y retorna
```