COMENTARIO TECNICO

Microcontroladores de 32 bits ARM...
... O como no temerle al cambio!!

Por Ing. Marcelo E. Romeo[i] - Ing. Eduardo A. Martínez[ii]

Parte 2

1. ARM7 – Principales características.

En base a lo presentado en el primer tramo de este artículo, podemos caracterizar a la familia de procesadores ARM diciendo que:

2.1 Son microcontroladores de 32 bits de palabra de instrucción.

2.2 La mayoría de las instrucciones se ejecutan en un ciclo de máquina

2.3 Tiene un pipeline que permite incrementar la cantidad de instrucciones ejecutadas por unidad de tiempo.

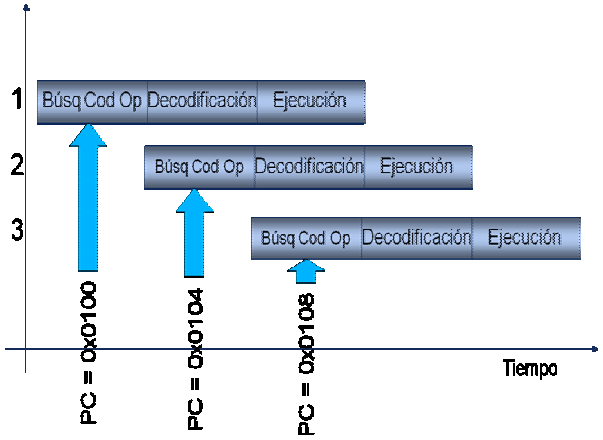

Figura 1. Pipeline de un microcontrolador ARM

Esta arquitectura permite que simultáneamente el procesador está buscando un código de operación, esta decodificando una instrucción leída anteriormente y ejecutando una instrucción aún anterior. Ello permite lograr ejecutar (aproximadamente) una instrucción por ciclo de máquina.

La ejecución de la instrucción puede explicitarse más detalladamente, como se ve en la Figura 2.

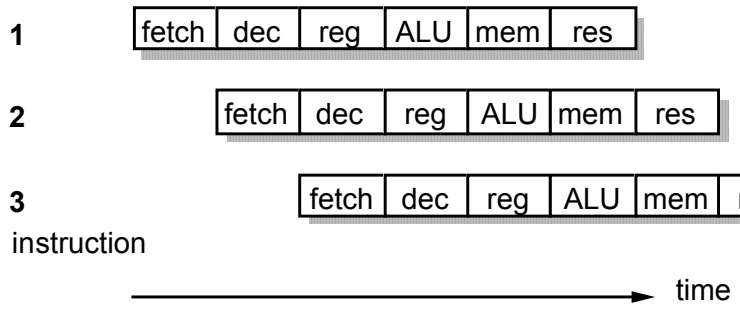

Figura 2. Pipeline en forma detallada[iii]

REG indica el acceso a operandos que puedan ser requeridos del banco de registros
ALU indica la combinación de operandos para formar un resultado o una dirección de memoria.
MEM indica el acceso a un operando de datos (si fuera necesario)
RES indica la reescritura del resultado en el banco de registros

Uno de los inconvenientes que presenta esta arquitectura (pipeline hazards), aparece cuando una instrucción necesita el resultado de una precedente. En ese caso, la segunda instrucción se debe demorar hasta que culmine la primera, como podemos ver en la Figura 3

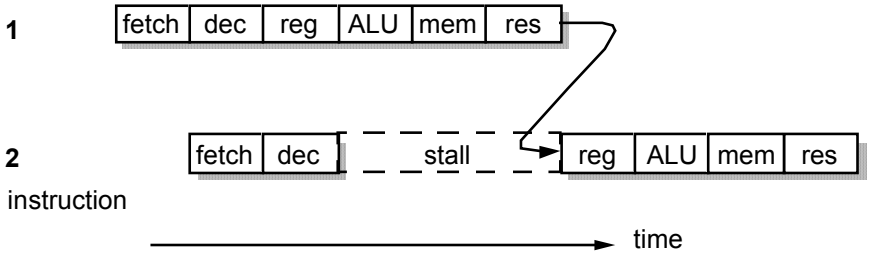

Figura 3. Demora en espera de un resultado previo (read-after-write pipeline hazard)

Esta situación produce un cambio (habitualmente no previsto) en la duración de un programa. En tal caso y con la idea de eliminar esta situación es conveniente cambiar el orden de las instrucciones del programa para evitar la espera.

Supongamos que los registros r2 y r3 contienen las variables Var1 y Var2, previamente cargadas en los mismos y se desea sumarlas para guardarlas en Resul1, cuya dirección se cargó en r1

```
        ADD         r0, r2, r3      ; suma Var1 con Var2
        STR         r0, [r1]
; Se quiere escribir el resultado de la suma anterior antes de que esté disponible. Dependencia de datos. Un ciclo perdido

        LDR         r4, =Var3       ; carga r4 con la variable Var3
```

Esta espera se eliminará adelantando otra instruccón para dar tiempo a que se obtenga el resultado de la suma

```
ADD     r0, r2, r3      ; suma Var1 con Var2
LDR     r4, =Var3       ; carga r4 con la variable Var3 y se da tiempo a que
                        ; se obtenga el resultado anterior
STR     r0, [r1]
```

Para el caso de encontrarnos en la secuencia de programa con un salto (branch), el contenido del pipeline queda desactualizado y se buscan tres códigos de operación innecesarios.

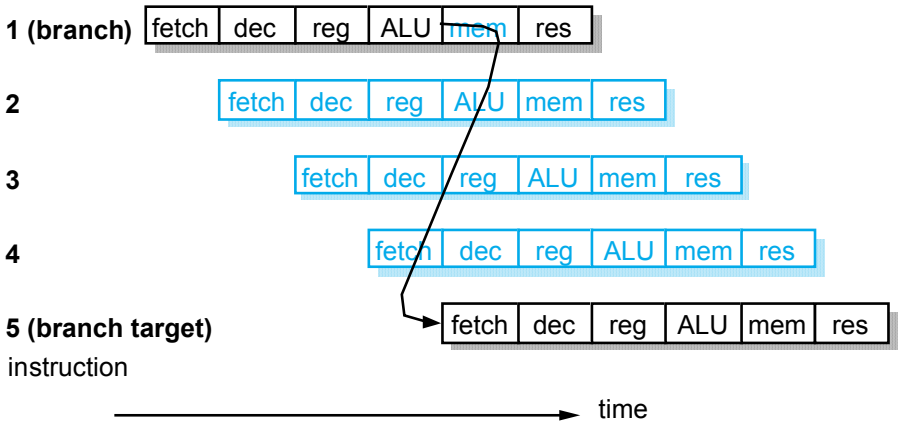

Figura 3. Desactualización del pipeline ante un salto

En este caso es fundamental que no se produzca un read-after-write pipeline hazard previo al cálculo de la dirección de salto.
La eficiencia del pipeline parecería ser mayor cuanto mayor sea su tamaño. También los problemas crecen con dicho tamaño. La experiencia indica que el tamaño óptimo del pipeline se encuentra entre 3 y *5 etapas.*

2.4 Cada instrucción puede ser ejecutada condicionalmente

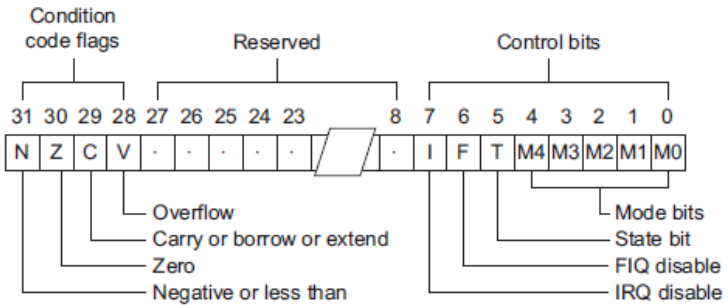

Fig 4. Registro de estado del programa (CPSR). Flags.

En el primer artículo mencionamos un registro de estado actual del programa (CPSR) que contenía un conjunto de indicadores (flags)
Los códigos de operación de la familia ARM no son generados al azar, sino que siguen una regla de formación con campos predefinidos y estrictos.
En la Figura 5 vemos la representación de los 32 bits de una instrucción aritmética genérica.

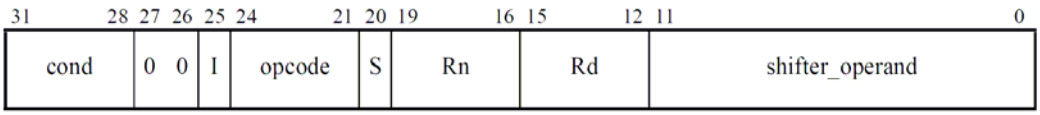

Figura 5. Código de operación de una instrucción aritmética

Los cuatro bits más significativos conllevan una condición para que la instrucción se ejecute o no. Esto quiere significar que cada instrucción se podrá ejecutar (o no) según el estado de los flags lo cual permite minimizar el tamaño de los programas.
Las condiciones que pueden ser tenidas para las ejecuciones condicionales son:

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-
1111	(NV)	See *Condition code 0b1111* on page A3-5	-

Veamos un ejemplo para tratar de fijar la idea: supongamos que si r0 es cero deseamos incrementar una variable que se halla en el registro r1. En caso contrario incrementaremos una variable que se halla en r2

```
if (r0 == 0)
{
        r1 = r1 + 1;
}
else
{
         r2 = r2 + 1;
}
```

Si empleamos las instrucciones tradicionales de cualquier procesador:

```
        CMP     r0, #0          ;Se compara la variable con 0
        BNE     no_es_cero
        ADD     r1, r1, #1      ; incrementa la variable en r1
        B       termina_ej
no_es_cero
        ADD     r2, r2, #1      ; en caso contrario incrementa la  variable en r2
termina_ej
```

Este programa lleva:

- 5 instrucciones
- 5 palabras
- 5 o 6 ciclos

Aprovechando las instrucciones condicionales, el programa se reduce a:

```
CMP     r0, #0      ; Se compara la variable con 0
ADDEQ   r1, r1, #1  ; Esta instrucción se ejecuta si la
                    ; comparación anterior resultó igual
ADDNE   r2, r2, #1  ; Esta instrucción se ejecutará si la
                    ; comparación fue distinta
```

Este programa lleva:

- 3 instrucciones
- 3 palabras
- 3 ciclos

2.5 Las instrucciones aritmético / lógicas pueden (ó no) afectar los flags

En la figura 5 vemos que dentro de los diversos campos del código de operación aparece un bit **S.** Si dicho bit se en encuentra en 0, las operaciones aritméticas o lógicas NO afectan los flags, mientras que si lo hacen si dicho bit esta en 1.

El valor de dicho bit será generado por la instrucción:

```
ADD   r0,r1,r2      ; r0 = r1 + r2. NO afecta los flags
ADDS  r0,r1,r2      ; r0 = r1 + r2. Afecta los flags
```

Al escribir nuestro programa, agregaremos (ó no, según lo que se desee hacer) la letra **S** al nemónico y el compilador generará el código de operación con el bit S adecuado.
Las instrucciones de comparación (obviamente) afectan los flags sin necesidad de que agreguemos la letra S en el nemónico.

2. Análisis del repertorio básico de Instrucciones

Enunciaremos muy someramente el repertorio de instrucciones del procesador. De ninguna manera pretende ser una enunciación exhaustiva y sólo busca mostrar las características fundamentales del procesador.
En el desarrollo de una aplicación es muy poco lo que se debe programar en assembler (habitualmente sólo el arranque ó startup) siendo recomendable el empleo de lenguajes de mayor nivel como el C.
En caso de desear profundizar en la programación a bajo nivel, recomendamos analizar la bibliografía propuesta, parte de la cual se encuentra disponible en internet.

2.1 Saltos

Las instrucciones de ramificación o salto son la base de la elaboración de sentencias lógicas dentro de un programa.
En el caso de la familia ARM, el salto incondicional (Branch ó Branch Always) son un caso dentro de los posibles de las ramificaciones.

Branch	Interpretación	Uso normal
B	Incondicional	Siempre se ejecuta
BAL	Always	Siempre se ejecuta
BEQ	Equal	Comparación con resultado igual ó cero
BNE	Not equal	Comparación con resultado desigual ó no-cero
BPL	Plus	Resultado positivo o cero
BMI	Minus	Resultado negativo
BCC	Carry clear	Operación Aritmética no generó acarreo
BLO	Lower	Comparación no signada resultó menor
BCS	Carry set	Operación Aritmética generó acarreo
BHS	Higher or same	Comparación no signada resultó mayor ó igual
BVC	Overflow clear	Operación con entero signado. NO generó desborde
BVS	Overflow set	Operación con entero signado. Generó desborde
BGT	Greater than	Operación con entero signado. Mayor que
BGE	Greater or equal	Operación con entero signado Mayor o igual que
BLT	Less than	Operación con entero signado Menor que
BLE	Less or equal	Operación con entero signado Menor o igual que
BHI	Higher	Comparación NO signada. Mayor.
BLS	Lower or same	Comparación NO signada. Menor o igual

Veremos algunos ejemplos para detectar algunas particularidades propias de la familia:

```
B       label    ; branch incondicional a label
BCC     label    ; branch a label si C = 0
BEQ     label    ; branch a label si el flag Z =1 (o sea, que la última operación aritmético
                 ; lógica que afectó los flags fue cero)
BL      func     ; Llamado a subrutina. Copia el valor del PC al R14 (link register) para
                 ; garantizar el retorno
MOV     PC, #0   ; R15 = 0, branch a la posición 0x00
```

Esta última expresión resalta el carácter ortogonal de la arquitectura ARM. El registro r15 ó PC es un registro más y puede inicializarse como cualquier otro y también emplearse como puntero a memoria.

2.2 Instrucciones de Procesamiento de datos

Arm dispone de una Unidad Aritmético lógica que permite realizar operaciones aritméticas y lógicas entre dos operandos para generar un resultado en un registro.

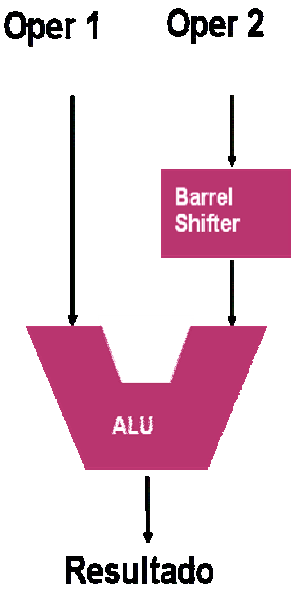

Figura 6: Unidad Aritmético Lógica

Este esquema se asemeja a una Unidad Aritmético Lógica de un Procesador Digital de Señales (DSP) debido a la aparición del desplazador que afecta opcionalmente al Operando 2, perimiendo rotar hacia la derecha o izquierda el segundo operando.
Los bits 21 a 24 de la Figura 5 marcan el código de operación de la instrucción y que corresponde a:

0000 AND AND Lógico	Rd := Rn AND Oper 2 desplazado
0001 EOR OR Lógica Exclusiva	Rd := Rn EOR Oper 2 desplazado
0010 SUB Resta	Rd := Rn - Oper 2 desplazado
0011 RSB Resta inversa	Rd := Oper 2 desplazado - Rn
0100 ADD Suma	Rd := Rn + Oper 2 desplazado
0101 ADC Suma con acarreo	Rd := Rn + Oper 2 desplazado + Carry Flag
0110 SBC Resta con acarreo	Rd := Rn - Oper 2 desplazado - NOT(Carry Flag)
0111 RSC Resta inversa con Acarreo	Rd := Oper 2 desplazado - Rn - NOT(Carry Flag)
1000 TST Test	Actualiza flags luego de Rn AND Oper 2 desplazado
1001 TEQ Test Equivalence	Actualiza flags luego de Rn EOR Oper 2 desplazado
1010 CMP Comparación	Actualiza flags luego de Rn - Oper 2 desplazado
1011 CMN Comparación Negada	Actualiza flags luego de Rn + Oper 2 desplazado
1100 ORR Logica (inclusiva)	OR Rd := Rn OR Oper 2 desplazado
1101 MOV Move	Rd := Oper 2 desplazado (sin primer operando)
1110 BIC Bit Clear	Rd := Rn AND NOT(shifter_operand)
1111 MVN Move Not	Rd := NOT Oper 2 desplazado (sin primer operando)

El segundo operando puede ser un registro o un valor inmediato determinado por el flag I de la Figura 5.

En caso de que se decida trabajar con un valor inmediato los 12 bits menos significativos pueden ser interpretados como 8 bits inmediatos (los menos significativos) y 4 de desplazamiento (los subsiguientes).

Ejemplos:

```
ADD   r4,PC,#8          ; r4 = r15 + 8
ADD   r1,r1,#1          ; Incrementa r1 (Por ser RISC no tiene instrucción de
                        ; incremento)
ADD   r3,r2,r1, LSL #3 ; r3 := r2 + 8 * r1
ADD   r5,r5,r3, LSL r2 ; r5 := r5 + r3 * 2^r2
AND   r8, r7,#0xff      ; r8 := r7[7:0]    Deja solo los 8 bits menos significativos.
AND   r0,r1,r2          ; r0 := r1.r2
ORR   r0,r1,r2          ; r0 := r1 + r2
EOR   r0,r1,r2          ; r0 := r1 xor r2
BIC   r0,r1,r2          ; r0 := r1 and not r2
```

2.3 Instrucciones de Transferencias de Datos

Son las instrucciones que permiten mover datos entre registros del procesador.

Ejemplos

- MOV r0,r2
- MVN r0,r2 ; r0:= not r2
- MOV PC,LR

Respecto de esta última instrucción cabe recalcar que es la inversa de la BL que mencionamos en los branches. Aquella copiaba el PC en el R14 (LR) para facilitar el retorno de una subrutina. Debido a que tenemos una arquitectura RISC NO tenemos una instrucción de retorno ni de retorno de interrupción, por lo que se debe implementar por medio del MOV mencionado

2.4 Instrucciones de carga y descarga (Load / Store)

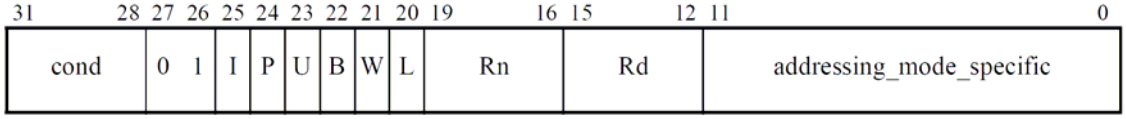

Figura 7: Instrucciones de transferencia de datos con memoria

Donde:

I, P, U, W permiten distinguier entre los distintos <modos de direccionamiento>.
L bit Distingue entre Carga (L==1) y almacenamiento (L==0).
B bit Distingue entre el acceso a un byte no signado (B==1) y una palabra (B==0).
Rn Especifica en registro base para <modos de direccionamiento>.
Rd Especifica el registro que sera almacenado o cargado.

Ejemplos:

```
LDR   R1, [R0]          ; Carga R1 de la dirección apuntada por R0
LDR   R8, [R3, #4]      ; Carga R8 de la dirección apuntada por R3 + 4
LDR   R12, [R13, #-4] ; Carga R12 de la dirección apuntada por R13 - 4
STR   R2, [R1, #0x100] ; Almacena R2 en la dirección apuntada por R1 + 0x100
LDRB  R5, [R9]          ; Carga byte en R5 de la dirección apuntada por R9
                        ; (24 bits más signiticativos = 0)
LDRB  R3, [R8, #3]      ; Carga byte en R3 de la dirección apuntada por R8 + 3
                        ; (24 bits más signiticativos = 0)
STRB  R4, [R10, #0x200] ; Almacena byte R4 en la dirección apuntada por
                        ; R10 + 0x200
LDR   R11, [R1, R2]     ; Carga R11 de la dirección apuntada por R1 + R2
STRB  R10, [R7, -R4]    ; Almacena byte de R10 en la dirección apuntada por R7 - R4
```

Como todas las instrucciones son de 32 bits, no se puede cargar un registro en forma inmediata con una dirección de 32 bits (no habría espacio para el código de operación), por lo que los ensambladores admiten el uso de una seudointrucción (símil a un macro) que facilita la actuación del programador.

Por ejemplo si escribimos:

```
LDR    r0,=0x55555555
MOV    r1,#0x12
ADD    r2,r0,r1
```

El compilador lee el signo = y entiende que pretendemos cargar el registro r0 con una constante de 32 bits, por lo que crea una constante en memoria de programa con el valor 0x55555555 y luego carga r0 desde esa posición de memória.

Así, El compilador realmente crea:

```
LDR    R0,[PC,#0x0004]
MOV    R1,#0x00000012
ADD    R2,R0,R1
DD     0x55555555          ; Constante de 32 bits en memoria de programa
```

Para entender el valor ,[PC,#0x0004], debe recordarse que debido a la arquitectura pipeline, cuando el procesador ejecuta la primera instrucción ell contador de programa PC se halla 8 bytes delante (Ver Figura 1).

2.5 Instrucciones de Load y Store Multiples

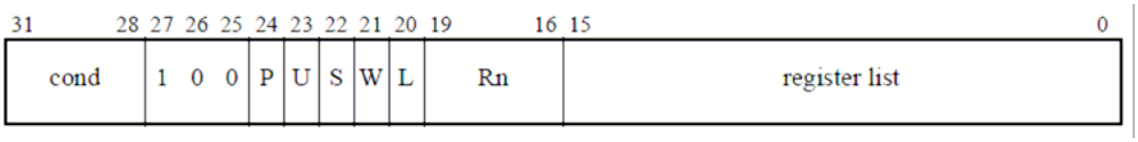

Figura 8: Carga y almacenamiento de múltiples datos

Una característica interesante de la familia ARM es que en una sola instrucción (pero no en un solo ciclo de máquina) se pueden cargar o almacenar múltiples valores incrementando o decrementando automáticamente los registros punteros.

- Sintaxis:
  - **<LDM|STM>**{<cond>}<modo de direccionamiento> Rb{!}, <lista de registros>
  -
- 4 modos de direccionamiento:

  - **LDMIA / STMIA** incremento posterior de registro puntero
  - **LDMIB / STMIB** incremento previo de registro puntero
  - **LDMDA / STMDA** decremento posterior de registro puntero
  - **LDMDB / STMDB** decremento previo de registro puntero

  Y que ejemplificamos en la Figura 9

```
LDMxx r10, {r0,r1,r4}
STMxx r10, {r0,r1,r4}
```

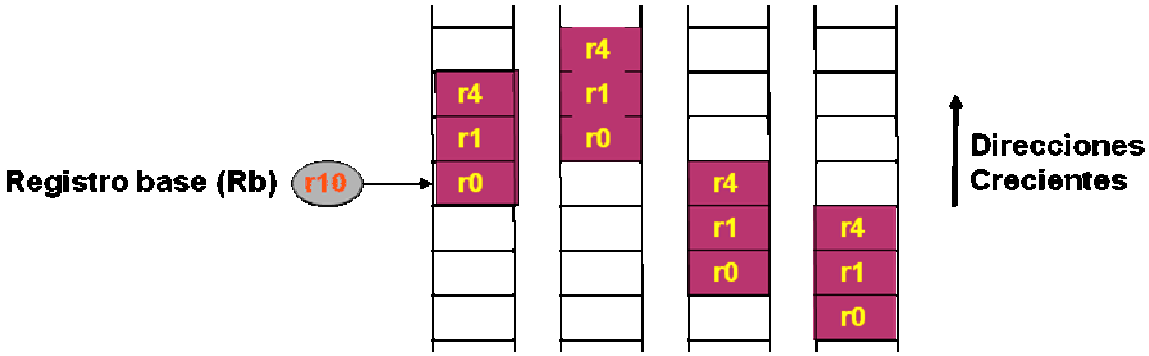

Figura 9: Instrucciones de múltiple almacenamiento y carga de datos

2.6 Preindexado y Postindexado

También disponemos de la funcionalidad de actualización de punteros en forma automática cuando efectuamos un movimiento de LOAD / STORE.

Preindexado significa que por ejemplo se suma la base y el desplazamiento antes de realizar la operación

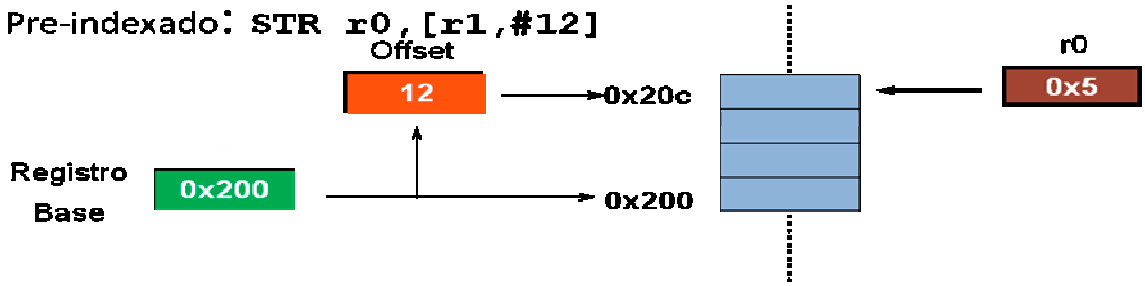

Figura 10: Almacenamiento Preindexado

Mientras que en el postindexado (indicado con un signo de admiración), el puntero se actualiza para apuntar a la próxima variable incrementándose en el offset propuesto.

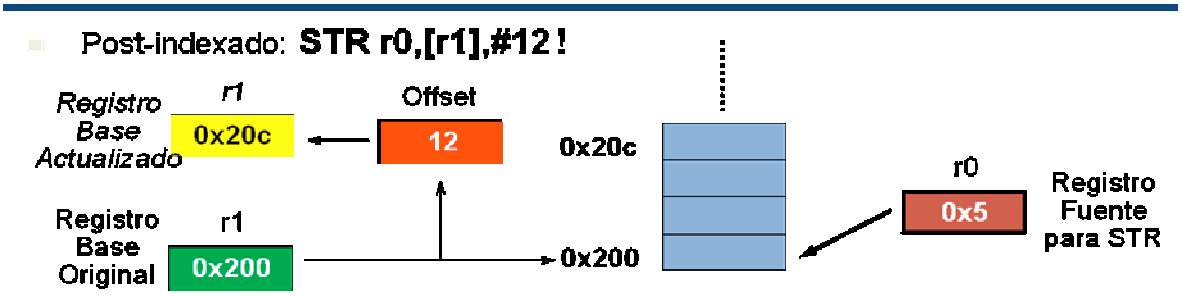

Figura 11: Almacenamiento Posindexado

Continuará en la Parte 3: Excepciones y el Uso de ARM en Sistema Operativos.

Nota de Radacción: El lector puede descargar este artículo y artículos anteriores desde la sección ***"Artículos Técnicos"*** en el sitio web de **EduDevices (www.edudevices.com.ar)**

3. Bibliografía

- ARM System-on-Chip Architecture (2nd Edition) - Steve Furber – Addison Wesley
- ARM Architecture Reference Manual - DDI0100E_ARM_ARM.pdf - http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406b/index.html
- ARM Architecture Reference Manual ARM®v7-A and ARM®v7-R edition - http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf

[i] Universidad de Belgrano – Universidad de San Martín – UTN - FRBA

[ii] Universidad de Belgrano

[iii] Figuras con autorización de Steve Furber