



Guía de  
Laboratorio

2025-1

# 04 - ISA - HL Structures

**CS3051 - Computer Architecture**

cwilliams, jgonzalez@utec.edu.pe  
TAs: lcarranza, mchinch@utec.edu.pe

# CONTENTS

<b>CHAPTER 1</b>	<b>INTRO TO ARM</b>	<b>PAGE 3</b>
1.1	What do ARM instructions look like?	3
1.2	Memory Instructions	4
1.2.1	Memoria básica: LDR y STR	4
1.3	Branch instructions	5
1.3.1	Ejecución condicional y flags	5
<b>CHAPTER 2</b>	<b>HIGH LEVEL STRUCTURES</b>	<b>PAGE 7</b>
2.1	While	7
2.2	For	7
2.3	If-Else	8
<b>CHAPTER 3</b>	<b>INSTRUCCIONES DE MEMORIA AVANZADAS</b>	<b>PAGE 9</b>
3.1	Acceso a 32 bits (dwords)	9
3.2	Acceso a 16 bits (words)	9
3.3	Acceso a 8 bits (bytes)	10

# Chapter 1

## Intro to ARM

Los procesadores ARM son una familia de CPUs que siguen el set de instrucciones reducido propio de ARM. Es decir, la arquitectura ARM es de tipo RISC (reduced instruction set). En este curso utilizaremos el ISA (instruction set architecture) de ARM, por lo que debemos familiarizarnos con las instrucciones)

### 1.1 What do ARM instructions look like?

Las instrucciones de un CPU ARM de 32 bits realmente son simplemente un conjunto 32-bits (double word) que el procesador interpreta y ejecuta, cambiando su propio estado (los valores almacenados en sus registros y sus flags). Sin embargo, al programar en Assembly, existe un formato legible para humanos, donde las instrucciones suelen seguir este formato:

**INSTR** **Rd**, **Rn**, **Rm**

Esto se interpreta de la siguiente forma:

- Se va a ejecutar la operación INSTR
- los operandos son los registros Rn y Rm.
- el resultado de la operación se almacenará en el registro Rd.

Por ejemplo:

```
ADD R3, R1, R4    // R3 <- R4 + R1
SUB R7, R7, R2    // R7 <- R7 - R2
AND R0, R4, R0    // R0 <- R4 & R0
ORR R1, R1, R1    // R1 <- R1 | R1
```

Si queremos hacer una operación usando valores constantes (como sumar 1, restar 4), podemos hacer uso de *valores inmediatos* (*immediate values*) de la siguiente forma:

```
ADD R3, R1, #1    // R3 <- R1 + 1
SUB R7, R7, #4    // R7 <- R7 - 4
AND R0, R4, #0x9f // hexadecimal
```

También tenemos la instrucción especial MOV, que Mueve un valor a un registro.

```
MOV R2, R9    // R2 <- R9
MOV R2, #0    // R2 <- 0
```

Todas estas instrucciones previas son de tipo **DP (Data Processing)**, ya que solo manipulan valores, como una calculadora. Sin embargo, para gozar de funcionalidad completa, hacen falta instrucciones de **Branching** Condicional (para cambiar la línea de código que estamos ejecutando, permitiendo hacer bucles, if-else, llamadas a funciones, etc.) y también hacen falta instrucciones de **Memoria** (para almacenar variables, estructuras de datos, usar el stack, etc.).



Name	Description	Operation
AND Rd, Rn, Src2	Bitwise AND	$Rd \leftarrow Rn \& Src2$
EOR Rd, Rn, Src2	Bitwise XOR	$Rd \leftarrow Rn \wedge Src2$
SUB Rd, Rn, Src2	Subtract	$Rd \leftarrow Rn - Src2$
RSB Rd, Rn, Src2	Reverse Subtract	$Rd \leftarrow Src2 - Rn$
ADD Rd, Rn, Src2	Add	$Rd \leftarrow Rn + Src2$
ADC Rd, Rn, Src2	Add with Carry	$Rd \leftarrow Rn + Src2 + C$
SBC Rd, Rn, Src2	Subtract with Carry	$Rd \leftarrow Rn - Src2 - \bar{C}$
RSC Rd, Rn, Src2	Reverse Sub w/ Carry	$Rd \leftarrow Src2 - Rn - \bar{C}$
TST  Rn, Src2	Test	Set flags based on $Rn \& Src2$
TEQ  Rn, Src2	Test Equivalence	Set flags based on $Rn \wedge Src2$
CMP Rn, Src2	Compare	Set flags based on $Rn - Src2$
CMN Rn, Src2	Compare Negative	Set flags based on $Rn + Src2$
ORR Rd, Rn, Src2	Bitwise OR	$Rd \leftarrow Rn   Src2$
<b>Shifts:</b>		
MOV Rd, Src2	Move	$Rd \leftarrow Src2$
LSL Rd, Rm, Rs/shamt5	Logical Shift Left	$Rd \leftarrow Rm \ll Src2$
LSR Rd, Rm, Rs/shamt5	Logical Shift Right	$Rd \leftarrow Rm \gg Src2$

Figure 1.1: Tabla de algunas instrucciones DP, del libro Harris &amp; Harris

Es importante conocer qué representan los bits dentro de cada instrucción, y cómo se convierten a binario. Para más información consultar el capítulo 6.2 del libro, y las diapositivas del curso sobre ISA Format.

## 1.2 Memory Instructions

Las instrucciones de memoria trabajan con direcciones en RAM (punteros).

### 1.2.1 Memoria básica: LDR y STR

Para cargar y almacenar valores usamos:

- LDR Rt, [Rn, #offset] – Carga un word (32 bits) desde memoria al registro Rt.
- STR Rt, [Rn, #offset] – Almacena un word (32 bits) desde Rt a memoria.

El desplazamiento (#offset) está en bytes.

**Ejemplo simple:**

```
// R0 = 0x00f4250C
LDR R1, [R0]      // R1 <- Mem[R0]
ADD R1, R1, #5     // R1 <- R1 + 5
STR R1, [R0]      // Mem[R0] <- R1
```

Más adelante pasaremos a las instrucciones avanzadas de memoria (arrays, halfword, byte).

## 1.3 Branch instructions

Las instrucciones de **Branching** cambian el Program Counter (PC, R15).

```
_start:
    MOV R0, #1
    B label
    ADD R0, R0, R0      // no se ejecuta
label:
    MOV R1, #1          // se ejecuta
```

Esto es muy útil, y de hecho podemos hacer bucles infinitos.

```
    MOV R1, #20
loop:
    SUB R1, R1, #1
    B loop
```

Sin embargo, en código real, necesitamos condiciones de parada, o bloques de código que se pueden ejecutar o not. Para eso necesitamos ejecución condicional.

### 1.3.1 Ejecución condicional y flags

ARM permite ejecutar cualquier instrucción de forma condicional según los flags del registro de estado:

N (negative), Z (zero), C (carry), V (overflow).

Para ajustar flags usamos la variante con "S":

```
    CMP R1, #0          // R1 - 0, actualiza N y Z
    SUBS R2, R2, #1     // resta y actualiza todos los flags
```

Luego añadimos el sufijo de condición a la instrucción:

- BEQ label – Branch if equal (Z == 1).
- BNE label – Branch if not equal (Z == 0).
- BGT label – Branch if greater than (Z == 0 y N == V).
- BLT label – Branch if less than (N != V).
- ...también podemos poner ejecución condicional a otras instrucciones...
- ADDGE – ADD if greater or equal (N == V)
- SUBVS – SUB if overflow set (V == 1)
- EORMI – EOR/XOR if negative (N == 1)
- ANDPL – AND if positive (N == 0)
- y muchas más...

**Note:-**

Ahora que sabemos de ejecución condicional, podemos hacer bucles finitos con condición de parada.

```
MOV R1, #20
loop:
SUBS R1, R1, #1    // S: Set Flags
BTG loop           // GT: Greater Than (Z==0 & N==0)
// ...
// llegará acá cuando R1 == 0
```

Esto es equivalente a:

```
int x = 20;
do {
    x = x - 1;
} while (x > 0)
// ...
// llegará acá cuando x == 0
```

Ahora, veremos como convertir otras abstracciones de un lenguaje de nivel más alto como C, a assembly ARM.

## Chapter 2

# High Level Structures

### 2.1 While

Un while loop es diferente al do-while visto antes. El bloque do-while ejecuta al menos una vez, y revisa si la condición se cumple después. En cambio el bloque while revisa la condición primero, y luego ejecuta su interior. Esto hace que necesitemos 2 instrucciones de branch, una condicional para chequear si se cumple la condición, y un Branch incondicional para el bucle.

#### Bucle While en C

```
int a = 0;
int b = 0;
while (a <= 500) {
    a = a << 1;
    b += 1;
}
a += 1;
```

#### Bucle While en ARM

```
MOV R1, #0      // R1: a
MOV R2, #0      // R2: b
while:
CMP R1, #500
BGT endwhile    // sale cuando R1 > 500

LSL R1, R1, #1
ADD R2, R2, #1

B while
endwhile:
ADD R1, R1, #1
```

### 2.2 For

Un bucle for necesita una variable adicional para iterar, y al igual que el while, debemos chequear si se sigue cumpliendo la condición antes de iterar. Asimismo, al final de cada iteración, tenemos que realizar la operación de incremento.

## Bucle For en C

```
int array[40];
for (int i = 0; i < 40; i++) {
    array[i] = i;
}
```

## Bucle For en ARM

```
// asumiendo que R1: &array
MOV R2, #0    // R2: i
for:
...// comparar y setear banderas
... // detener si i >= 40

STR R2, [R1, R2, LSL #2]    // Guardamos R2 en R1 + R2*4

...// incrementar el contador
endfor:
```

## 2.3 If-Else

Una estructura `if-else` permite ejecutar distintos bloques de código dependiendo de si una condición se cumple o no. Por ejemplo:

## If Else in C

```
int x = 4;
int y;
if (x < 10) {
    y = 1;
} else {
    y = 2;
}
```

Traducción a Assembly ARM:

## If Else in ARM

```
MOV R0, #4    // x = 4
// R1 será y

... // comparar
... // if x >= 10, ir a else

... // y = 1 usar MOV
B endif

else:
... // y = 2 usar MOV
endif:
```



## Chapter 3

# Instrucciones de Memoria Avanzadas

### 3.1 Acceso a 32 bits (dwords)

- LDR — carga 32 bits desde memoria.
- STR — almacena 32 bits en memoria.

**Ejemplo:** Crea y recorre un array de `uint32_t` de 10 elementos.

C code

```
// C code
uint32_t arr[10];
for (int i = 0; i < 10; i++) {
    arr[i] = i + 3;
}
```

ARM code

```
MOV    R0, #0x10000000 // en ARM los immediates son de 8-bit
ORR    R0, #0x00aa0000 // es decir, de 0x00 a 0xff (0 a 255)
ORR    R0, #0x0000ff00 // pero se pueden rotar de esta forma
ORR    R0, #00000000x30 // R0 = 0x10aaff30, esto es solo para escoger
                        // un espacio de memoria específico.
MOV    R1, #0          // R1 = i

loop32:
    ... // completar usando el ejemplo del FOR previo como
    ... // referencia
end32:
//hacer una grafica explicando como queda finalmente su memoria
```

### 3.2 Acceso a 16 bits (words)

- LDRH — carga 16 bits (halfword).
- STRH — almacena 16 bits (halfword).

**Ejemplo:** Crea y recorre un array de `uint16_t` de 8 elementos.

## ARM code

```

    // R0 = &arr
    MOV    R1, #0           // índice

loop16:
    CMP    R1, #8
    BGE    end16

    // almacena el valor 0xABCD
    MOV    R4, #0xABCD
    STRH   R4, [R0, R1, LSL #1]

    ADD    R1, R1, #1
    B      loop16

end16:
    //hacer una grafica explicando como queda finalmente su memoria

```

### 3.3 Acceso a 8 bits (bytes)

Para un array de 8-bits, o un `char [ ]` en C, no necesitamos multiplicar nuestro offset.

- LDRB — carga un byte (8 bits).
- STRB — almacena un byte (8 bits).

**Ejemplo:** Copiar 16 bytes de un buffer origen a destino.

## ARM code

```

    // R0 = &source
    // R1 = &dest
    MOV    R2, #0           // index

copy_bytes:
    CMP    R2, #16
    BGE    done_copy

    // cargar un byte y almacenarlo
    LDRB   R3, [R0, R2]     // copy
    STRB   R3, [R1, R2]     // paste

    ADD    R2, R2, #1
    B      copy_bytes

    //hacer una grafica explicando como queda finalmente su memoria

done_copy:

```

## Notas sobre offsets y escalado

- El `#offset` siempre está en bytes, y no necesita escalarse manualmente en las instrucciones de load/store con formato inmediato.
- Para acceder al índice  $i$ -ésimo, podemos usar un registro para acceder a un offset variable. Sin embargo, si nuestro array es de elementos de 16 o 32 bits, tendremos que multiplicar nuestro offset por 2 o 4 respectivamente, ya que la memoria es byte addressable. Recordamos que multiplicar por 2 o por 4 es equivalente a hacer `LSL #1`, o `LSL #2`, respectivamente. Podemos agregar un `LSL` dentro de la instrucción.

$\text{offset} = \text{índice} \times \text{tamaño\_en\_bytes} \rightarrow \text{LSL Rd, [Rn, R\_offset, LSL \# [1/2]]}$

- ARM permite especificar offset post-index o pre-index para actualizar el puntero y acceder a la memoria con una sola instrucción. Esto nos permite recorrer un array sin tener que incrementar o reducir el puntero manualmente cada vez con un `ADD` o `SUB`.

### ARM code

```
LDR R3, [R0], #4    // R3 = Mem[R0], y luego R0 += 4 (post-index)
STR R3, [R1, #-4]!  // R1 -= 4, y luego R3 = Mem[R1] (pre-index)
```