



# Sistemas electrónicos digitales

## Tema 3: Programación en ensamblador

```
0100000000010100011011000000100101100011
110001011101000100011111111110100000100
00101001011000011010111011010110110010001
01101100000101011001000100001110001001111
0100110010110100110110100111101111011110
0001101000100011010011010011101000011010
100100110100100100101000111010001110100
10001001int main()
10101001{
1110011000 printf("Hello World");
001000001; return 42;
0001101000100011101001110001101000011010
01001001101111010111011110000001010001110
1000100100010101100100111011101000101111
01010100111001101010111000101010100011000
1110011000001101111110101001111110001100
0100000111111101010010010011010101110110
```



## Índice del tema



Introducción



Instrucciones de Procesamiento de Datos



Instrucciones de Transferencia de Datos



Instrucciones de Control



Ejemplo de programa en ensamblador



# Introducción

 Instrucción de 32 bits → 4 billones de combinaciones

 Juego de instrucciones:

- Nuestro objetivo es el juego de instrucciones estándar de ARM
- Posibilidad de comprimir instrucciones en palabras de 16 bits con alta densidad de código
- Algunos cores soportan extensiones al juego de instrucciones, enfocadas a procesamiento de señales (DSP)

 Programación a nivel de usuario

 Clasificación de las instrucciones

- Instrucciones de procesamiento de datos
- Instrucciones de transferencia de datos
- Instrucciones de control



# Instrucciones de Procesamiento de Datos

## Reglas:

- Incluyen las operaciones aritméticas, lógicas, de comparación y de movimiento entre registros
- Todos los operandos son de 32 bits (excepto en *Long Multiply* donde el resultado es de 64 bits)
- Todos los operandos están en registros o codificados en la instrucción (operando inmediato)
- Formato de tres direcciones: los operandos fuentes y el destino se especifican de manera independiente
- La primera dirección corresponde al destino
- Un operando estará siempre almacenado en registro
- Ejemplo

```
ADD  r0, r1, r2           ; r0 := r1 + r2
                               ; El ";" se utiliza para introducir
                               ; comentarios en el programa
```



## Instrucciones de Procesamiento de Datos: Inst. Aritméticas con operandos en registros

 Operandos de 32 bits sin signo o en complemento a 2

ADD r0, r1, r2 ;  $r0 := r1 + r2$


ADC r0, r1, r2 ;  $r0 := r1 + r2 + C$

SUB r0, r1, r2 ;  $r0 := r1 - r2$

SBC r0, r1, r2 ;  $r0 := r1 - r2 + C - 1$

RSB r0, r1, r2 ;  $r0 := r2 - r1$

RSC r0, r1, r2 ;  $r0 := r2 - r1 + C - 1$

 El estado del procesador se modifica en el valor del registro destino y, **opcionalmente**, en los indicadores N, Z, C, V

 Salvo determinadas excepciones, el PC puede usarse como operando fuente o destino (retorno de subrutina)



# Instrucciones de Procesamiento de Datos: Inst. Lógicas y de movimiento de registros con operandos en registros



## Instrucciones lógicas

AND r0, r1, r2 ; r0 := r1 and r2

ORR r0, r1, r2 ; r0 := r1 or r2

EOR r0, r1, r2 ; r0 := r1 xor r2

BIC r0, r1, r2 ; r0 := r1 and not r2

- Actúa como “bit clear”. Cada bit del operando r2 activo a 1 provoca la “puesta a cero” del correspondiente bit del operando r2



## Instrucciones de movimiento de datos

■ La transferencia se realiza entre registros

■ Omiten el primer operando fuente



■ Copia directa o en complemento a 1

MOV r0, r2 ; r0 := r2

MVN r0, r2 ; r0 := not r2



## Instrucciones de Procesamiento de Datos: Inst. de Comparación con operandos en registros

-  No se modifica ninguno de los operandos y se omite la dirección del destino
-  Únicamente se utilizan para activar los códigos de condición

### Instrucciones de comparación

CMP r1, r2	; set cc on r1 – r2
CMN r1, r2	; set cc on r1 + r2
TST r1, r2	; set cc on r1 and r2
TEQ r1, r2	; set cc on r1 xor r2





# Instrucciones de Procesamiento de Datos:

## Instrucciones con operando inmediato

 El operando inmediato se especifica usando “#”

 Ejemplos:

ADD r0, r1, #5 ; r0 := r1 + 5

ADD r0, r0, #1 ; equivalente a r0++

AND r0, r1, #&FF ; #& (operando inmediato hex.)

 Operando inmediato codificado en la instrucción por lo que el valor es limitado:

- Valor inmediato =  $(0 \dots 255) \times 2^{2n}$  donde  $0 \leq n \leq 12$
- Cubre todas las potencias de 2, si bien hay valores que no se pueden codificar:
  - Valor de un byte en cualquier posición de los 4 bytes de un registro





## Instrucciones de Procesamiento de Datos: Instrucciones con operando en registros desplazados I

- El ARM no especifica instrucciones propias de desplazamiento y rotación, si bien se pueden usar para desplazar o rotar el segundo operando:

ADD r1, r2, r3, LSL #3 ;  $r1 := r2 + 8 * r3$  r3 se desplaza  
; 3 veces a la izquierda, lo que  
; equivale a multiplicar por 8

- La ejecución de estas operaciones se realiza en un sólo ciclo de reloj

- Es posible indicar el desplazamiento a través de un tercer registro

ADD r5, r1, r2, LSL r3 ;  $r5 := r1 + r3 * 2^{r2}$

- Únicamente los 8 bits menos significativos de r2 son útiles



## Instrucciones de Procesamiento de Datos: Instrucciones con operando en registros desplazados II

 Operaciones de desplazamiento y rotación disponibles:



*Logical/Arithmetic Shift Left*

LSL #5 ; ASL #5



*Logical Shift Right*

LSR #5



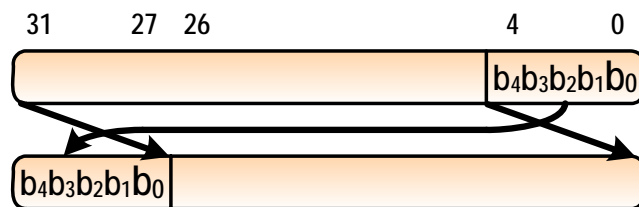
*Arithmetic Shift Right*

ASR #5; operando positivo



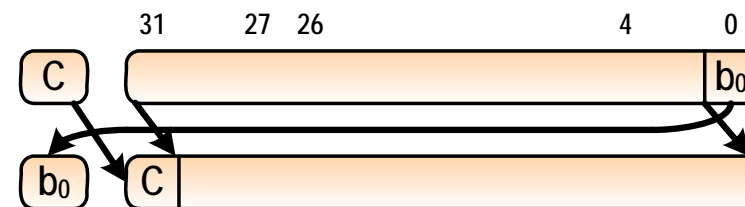
*Arithmetic Shift Right*

ASR #5; operando negativo



*Rotate Right*

ROR #5



*Rotate Right Extended (1-bit)*

RRX



# Instrucciones de Procesamiento de Datos: Códigos de condición

 Las instrucciones de procesamiento **pueden** activar los códigos de condición:

- Las instrucciones de comparación siempre activan los códigos
- Para activar los códigos de condición hay que poner el sufijo “**s**” al nemónico correspondiente
- Suma de dos operandos de 64 bits almacenados en {r1r0} y {r3r2} (manejo de datos de 64 bits):
  - **ADDs** r2, r2, r0 ; Si hay acarreo se activa C
  - **ADC** r3, r3, r1 ; y se suma a la parte alta

 Instrucciones aritméticas y de comparación

- Activan todos los flags

 Instrucciones lógicas:

- Únicamente activan N y Z, excepto en RRX donde C se actualiza



# Instrucciones de Procesamiento de Datos:

## Instrucciones de multiplicación



### Particularidades:

- No soportan dato inmediato como segundo operando
- El registro de resultado no puede coincidir con el registro del primer operando. Opción de usar sufijo “s” (flag V se mantiene)

`MUL r4, r3, r2` ;  $r4 := (r3 \times r2)_{[31:0]}$

- El resultado de la operación son 64 bits. Sin embargo, únicamente se almacenan los 32 bits menos significativos
- Existe una variable que posibilita la operación de multiplicación y acumulación:

`MLA r4, r3, r2, r1` ;  $r4 := (r3 \times r2 + r1)_{[31:0]}$

- No soportan dato inmediato como segundo operando



### Optimización: multiplicación por una constante ( $r0 \times 35$ )

`MOV r1, #35` ; El código se puede optimizar mediante:

`MOV r2, r0` `ADD r0, r0, r0, LSL #2` ;  $r0' := 5 \times r0$

`MUL r0, r1, r2` `RSB r0, r0, r0, LSL #3` ;  $r0'' := 7 \times r0' = 35 \times r0$



# Instrucciones de Transferencia de Datos: Tipos



## Transferencia desde o hacia un registro único

- Admiten tamaños de operando *byte*, *halfword* o *word*



## Transferencia desde o hacia múltiples registros:

- Ideales para salvar y recuperar registros de trabajo, principalmente en puntos de entrada y salida de rutinas



## Intercambio de información, *SWAP*

- Permite el intercambio de información entre un registro y una posición de memoria
- Realizan una operación de *load* y *store* durante su ejecución
- Especialmente pensadas para implementar “semáforos”



## Modos de direccionamiento en el ARM

- El acceso a un dato en memoria implican el uso de un registro
- Todos los modos de direccionamiento se basan en direccionamiento indirecto por registro, con muchas variantes



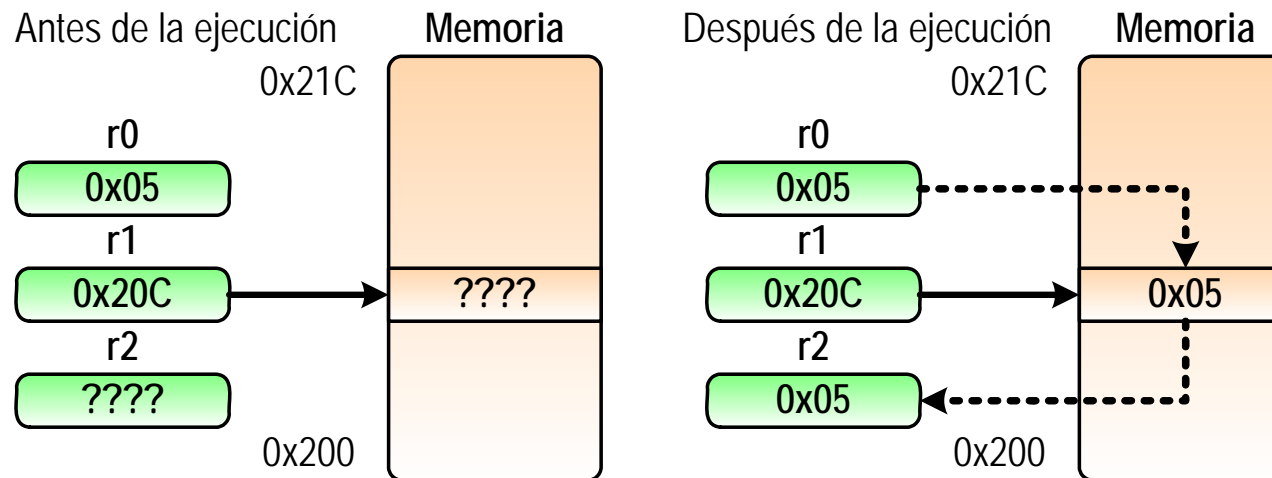
# Instrucciones de Transferencia de Datos: Direccionamiento Indirecto de registro I

## Direccionamiento indirecto por registro:

- La dirección efectiva está contenida en un registro (base)

STR r0, [r1] ;  $\text{mem}_{32}[\text{r1}] := \text{r0}$

LDR r2, [r1] ;  $\text{r2} := \text{mem}_{32}[\text{r1}]$



- El puntero debe apuntar a la posición de memoria o, en su defecto, a una dirección dentro de un rango de 4Kbytes



# Instrucciones de Transferencia de Datos: Direccionamiento Indirecto de registro II

## Inicialización del registro puntero:

- Se puede usar una instrucción de movimiento de datos
- Uso de *pseudo-instrucciones* → ADR (se traduce por una simple instrucción de suma o resta)

```
COPY      ADR r1, TABLE1    ; r1 apunta a TABLE1
          ADR r2, TABLE2    ; r2 apunta a TABLE2
          LDR r0, [r1]        ; Carga el primer valor de TABLE1 y
          STR r0, [r2]        ; lo almacena en TABLE2
          ...
TABLE1     ...                ; Localización de TABLE1
          ...
TABLE2     ...                ; Localización de TABLE1
```

- Uso de etiquetas:
  - Simplifica la programación a nivel ensamblador





# Instrucciones de Transferencia de Datos: Direccionamiento Indirecto de registro III

## Actualización de los punteros:

- Mediante instrucciones simples de suma/resta

```
COPY      ADR r1, TABLE1    ; r1 apunta a TABLE1
          ADR r2, TABLE2    ; r2 apunta a TABLE2
          LDR r0, [r1]        ; Carga el primer valor de TABLE1 y
          STR r0, [r2]        ; lo almacena en TABLE2
          ADD r1, r1, #4      ; Avanza r1 hacia la segunda palabra
          ADD r2, r2, #4      ; Avanza r2 a la segunda posición
          ...
TABLE1     ...                ; Localización de TABLE1
          ...
TABLE2     ...                ; Localización de TABLE1
```



## Instrucciones de Transferencia de Datos: Direccionamiento Indirecto de registro con desplazamiento I

La dirección efectiva se calcula mediante la suma del registro base más un *offset* inmediato

Múltiples posibilidades

- Direccionamiento preindexado:

LDR r0, [r1, #4] ; r0 := mem<sub>32</sub>[r1 + 4]

- Direccionamiento preindexado con autoindexación:

LDR r0, [r1, #4]! ; r0 := mem<sub>32</sub>[r1 + 4] (! indica autoindexación)  
; r1 := r1 + 4 (no requiere ciclos adicionales)

- Direccionamiento postindexado:

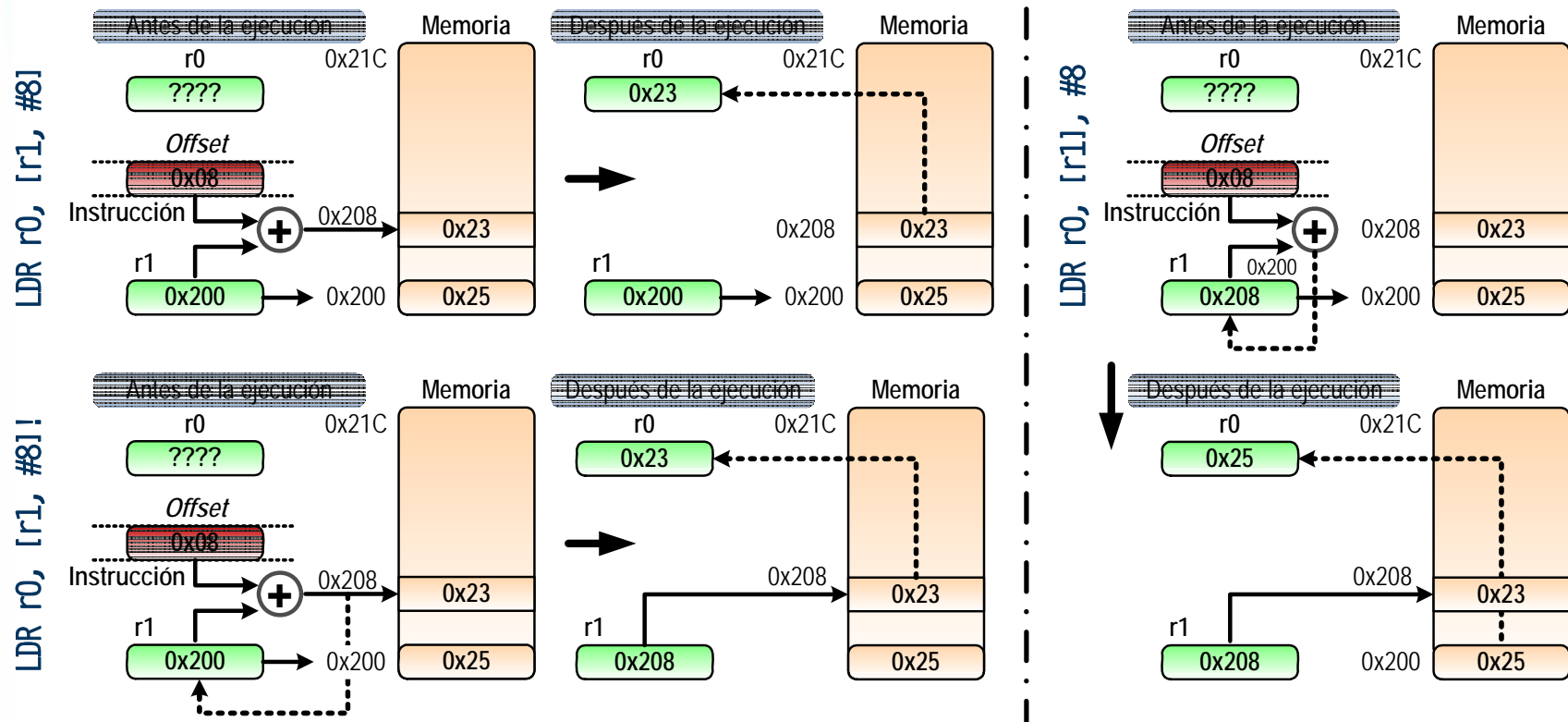
LDR r0, [r1], #4 ; r0 := mem<sub>32</sub>[r1]  
; r1 := r1 + 4 (no requiere ciclos adicionales)

Útil en el manejo de vectores, tablas, etc.



# Instrucciones de Transferencia de Datos: Direcccionamiento Indirecto de registro con desplazamiento II

## Direcccionamiento preindexado:





## Instrucciones de Transferencia de Datos: Direccionamiento Indirecto de registro con desplazamiento III



### Autoincremento de los punteros:

- Mediante instrucciones simples de suma/resta

```
COPY      ADR r1, TABLE1    ; r1 apunta a TABLE1
          ADR r2, TABLE2    ; r2 apunta a TABLE2
          LDR r0, [r1], #4    ; carga y actualiza el puntero
          STR r0, [r2], #4    ; almacena y actualiza el puntero
          ...
TABLE1    ...                ; Localización de TABLE1
          ...
TABLE2    ...                ; Localización de TABLE1
```



### Desplazamiento:

- Dato inmediato codificado en la instrucción
- Puede usarse un registro y, opcionalmente, someterlo a un desplazamiento antes de calcular la dirección efectiva



# Instrucciones de Transferencia de Datos: Selección del tamaño del operando

## Selección del tamaño del operando:

- La carga de operandos tamaño *byte* o *halfword* se consigue añadiendo modificadores a las instrucciones vistas:

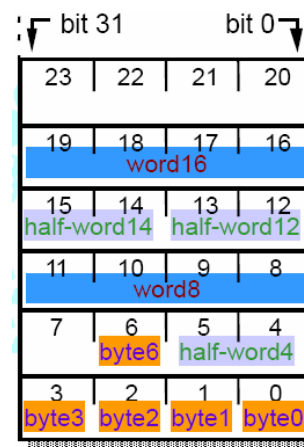
LDRB r0, [r1] ; r0 := mem<sub>8</sub>[r1] → *Load byte* sin signo

LDRH r0, [r1] ; r0 := mem<sub>16</sub>[r1] → *Load halfword* sin signo

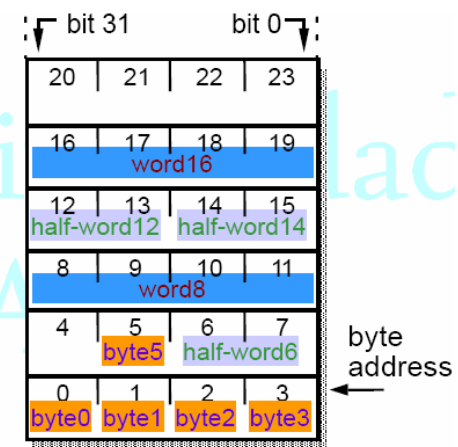
LDRSB r0, [r1] ; r0 := mem<sub>8</sub>[r1] → *Load byte* con signo

LDRSH r0, [r1] ; r0 := mem<sub>16</sub>[r1] → *Load halfword* con signo

- Las direcciones de operandos *halfword* deben estar alineadas en direcciones pares



Little-endian memory organization



Big-endian memory organization



# Instrucciones de Transferencia de Datos:

## Transferencia por bloque

### Transferencia hacia/desde múltiples registros

- Carga el contenido de posiciones de memoria consecutivas en varios registros

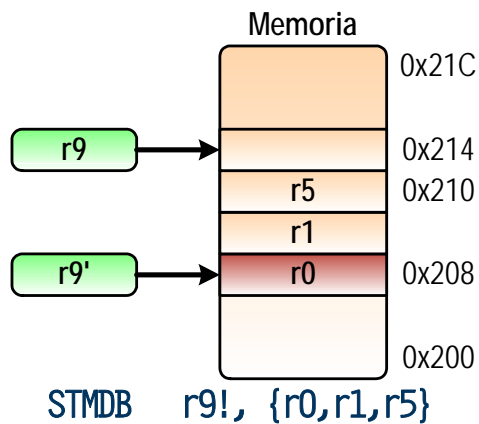
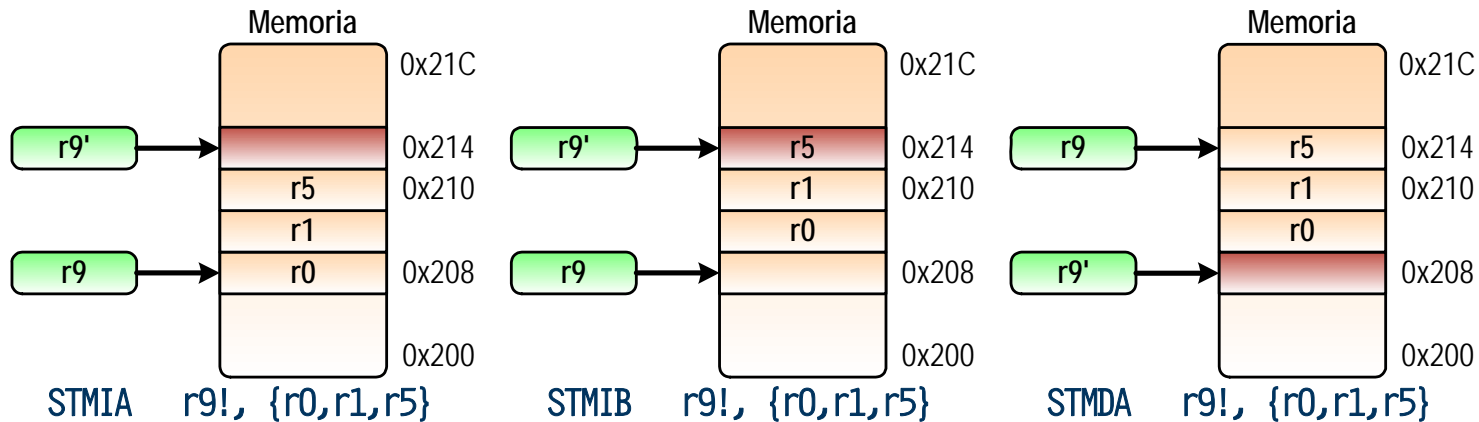
```
LDMIA    r1!, {r0,r2,r5}    ; r0 := mem32[r1]
                                   ; r2 := mem32[r1 + 4]
                                   ; r5 := mem32[r1 + 8]
                                   ; r1 := r1 + 8 “el puntero se actualiza”
```

```
STMIB    r1, {r0-r2}        ; mem32[r1 + 4] := r0
                                   ; mem32[r1 + 8] := r1
                                   ; mem32[r1 + 12] := r2
```

- Los operandos siempre serán de 32 bits
- El orden de los registros entre llaves es irrelevante
- No son instrucciones puramente RISC, aunque son hasta cuatro veces más rápidas que su implementación mediante instrucciones de transferencia simple



# Instrucciones de Transferencia de Datos: Transferencia por bloque y su relación con la pila



Muy útil en mecanismos de llamada y retorno de rutinas

		Ascending		Descending	
		Full	Empty	Full	Empty
Increment	Before	STMIB STMFA			LDMIB LDMED
	After		STMIA STMEA	LDMIA LDMFD	
Decrement	Before		LDMDB LDMEA	STMDB STMFD	
	After	LDMDA LDMFA			STMDA STMED





# Instrucciones de Transferencia de Datos: Manejo de la Pila I



## Control de la pila:

- No existen instrucciones específicas para el manejo de la pila. Ésta se gestiona con instrucciones de transferencia, simple o por bloques
- El funcionamiento es tipo LIFO
- El ARM permite un control total a la hora de implementar la pila, ésta puede hacerse en orden creciente o decreciente
  - *Ascending-stack*
  - *Descendig-stack*
- Del mismo modo, el *Stack Pointer* puede apuntar al último dato almacenado en la pila o bien a la primera posición libre en la pila
  - *Full-stack*
  - *Empty-stack*



# Instrucciones de Transferencia de Datos: Manejo de la Pila II

## Gestión de la pila por transferencias simples

- Almacenamiento de un dato en la pila

STR r0, [r13, #4]! ;  $\text{mem}_{32}[\text{r13} + 4] := \text{r0}$ ;  $\text{r13} := \text{r13} + 4$

- Recuperación de un dato desde la pila

LDR r0, [r13], #-4 ;  $\text{r0} := \text{mem}_{32}[\text{r13}]; + 4$ ;  $\text{r13} := \text{r13} - 4$

- En el ejemplo expuesto se trata de una pila *full-ascending*.  
Posibilidad de implementar cualquier combinación

## Gestión de la pila por transferencias de bloque

- Preservar el contexto de los registros r2 a r9

STMFD r13!, {r2-r9} ; Salva los registros en la pila

.....

LDMFD r13!, {r2-r9} ; Recupera los registros de la pila

- Para la correcta gestión de la pila, los modos de almacenamiento y recuperación, deben ser los **duales**, lo que asegura que la pila queda en su estado inicial tras recuperar los datos



## Instrucciones de Control: Instrucción *Branch*

- Permiten determinar qué instrucción/es se van a ejecutar. No modifican ni transfieren datos
- La instrucción más común dentro de esta categoría es la instrucción de salto. Existen dos tipos de salto

- Salto incondicional: Siempre se realiza el salto al ejecutar la instrucción

```
                B        LABEL    ; Salta a la instrucción etiquetada como
                ...              ; LABEL
LABEL           ...
```

- Salto condicional: El salto se ejecuta sólo si se cumple una determinada condición, basada en los *flags* de estado

```
                ; for (i=0; i<10; i++){...}
MOV             r0, #0                ; Inicialización de un contador
LOOP           ...
                ADD        r0, r0, #1    ; Incremento del contador
                CMP        r0, #10       ; Compara con límite
                BNE        LOOP           ; Repetimos bucle si no es igual
                ...
```



# Instrucciones de Control: Condiciones de salto I

## Saltos condicionales

Salto	Interpretación	Definición
B BAL	Unconditional Always	Siempre se realiza el salto Siempre se realiza el salto
BEQ	Equal	La comparación es igual o el resultado es cero
BNE	Not equal	La comparación no es igual o el resultado no es cero
BPL	Plus	Resultado positivo o cero
BMI	Minus	Resultado menor o negativo
BCC BLO	Carry Clear Lower	Operación aritmética produce acarreo Inferior; Comparación sin signo
BCS BHS	Carry Set Higher or Same	Operación aritmética no produce acarreo Superior o igual; Comparación sin signo
BVC	Overflow Clear	No hay overflow; Aritmética entera con signo
BVS	Overflow Set	Hay overflow; Aritmética entera con signo
BGT	Greater Than	Mayor que; Aritmética entera con signo
BGE	Greater or Equal	Mayor or igual; Aritmética entera con signo



# Instrucciones de Control: Condiciones de salto II

## Saltos condicionales

Salto	Interpretación	Definición
BLT	Less Than	Menor que; Aritmética entera con signo
BLE	Less or Equal	Menor o igual; Aritmética entera con signo
BHI	Higher	Superior; Comparación sin signo
BLS	Lower or Same	Inferior o igual; Comparación sin signo

### ■ Ejemplo de uso: **if (r0 == 5) {r1=2\*r1-r2}**

```
CMP    r0, #5           ; Activación de los flags
BNE    NEXT             ; if (r0 == 5)
ADD    r1, r1, r1        ; { r1 = 2*r1
SUB    r1, r1, r2        ; r1 = 2*r1 - r2}
NEXT   ...
```

### ■ Optimización:

```
CMP    r0, #5           ; Activación de los flags
BNE    NEXT             ; if (r0 == 5)
RSB    r1, r2, r1, LSL #1 ; r1 = 2*r1 - r2
NEXT   ...
```



# Instrucciones de Control: Ejecución condicional de las instrucciones

 ARM permite la ejecución condicional de cada instrucción

## ■ Ejemplo: ejecución con salto y comparación

```
CMP      r0, #5           ; Activación de los flags
BNE      NEXT            ; if (r0 == 5)
ADD      r1, r1, r1       ; { r1 = 2*r1
SUB      r1, r1, r2       ; r1 = 2*r1 - r2}
NEXT     ...
```

## ■ Ejemplo: ejecución condicional de instrucciones

```
CMP      r0, #5           ; if (r0 == 5)
ADDEQ    r1, r1, r1       ; { r1 = 2*r1
SUBEQ    r1, r1, r2       ; r1 = 2*r1 - r2}
```

- Se consigue añadiendo el sufijo de condición a la instrucción (antes de cualquier otro modificador tal como “S”, “B” o “H”)

## ■ Posibilidad de escribir código compacto

```
CMP      r0, r1           ; if ((a == b) && (c == d))
CMPEQ    r2, r3           ;
ADDEQ    r4, r4, #1       ; { e++}
```



# Instrucciones de Control: Llamadas a subrutinas I

## Llamadas a subrutinas:

- En las llamadas a subrutinas, el procesador debe almacenar la dirección de retorno en la Pila. El ARM no tiene control automático de la Pila, por lo que no salvará la dirección de retorno
- La llamada a subrutina en el ARM se ejecuta mediante la instrucción BL (*Branch and Link*)

	BL	SUBR	; Llamada a SUBR
	...		; Punto de retorno de SUBR
SUBR	...		; Punto de entrada a SUBR
	MOV	pc, r14	; Retorno de SUBR

- Anidamiento de subrutinas; Estado de la máquina

	BL	SUB1	; Llamada a SUB1
	...		; Punto de retorno de SUB1
SUB1	STMFD	r13!, {r0-r2, r14}	; Salvar el estado de la máquina
	...		; Cuerpo de la rutina
	BL	SUB2	; Llamada a SUB2
	...		; Punto de retorno de SUB2
SUB2	...		; Punto de entrada a SUB2





## Instrucciones de Control: Llamadas a subrutinas II

### Uso de r14 dentro de un mismo modo:

- Al disponer de un único registro r14, éste debe almacenarse en la pila en caso de anidamiento de rutinas

- Anidamiento de subrutinas; Estado de la máquina

	BL	SUB1	; Llamada a SUB1
	...		; Punto de retorno de SUB1
SUB1	STMFD	r13!, {r0-r2, r14}	; Salvar el estado de la máquina
	...		; Cuerpo de la rutina
	BL	SUB2	; Llamada a SUB2
	...		; Punto de retorno de SUB2
	LDMFD	r13!, {r0-r2, r14}	; Recuperar el estado
	MOVE	pc, r14	; Retorno de SUB1
SUB2	...		; Punto de entrada a SUB2
	...		; Cuerpo de la rutina
	MOVE	pc, r14	; Retorno de SUB2

- Posibilidad de recuperar directamente el PC desde la pila

	LDMFD	r13!, {r0-r2, pc}	; Recuperar el estado y la ; dirección de retorno
--	-------	-------------------	--

- *Full-descending* es la manera más común de implementar la pila en el ARM. !Tú decides!



## Instrucciones de Control: *Supervisor calls*

- Supervisor: programa que opera en modo privilegiado y tiene acceso a recursos que en modo usuario no son accesibles, por ejemplo recursos de entrada/salida. En la mayoría de los casos el supervisor corresponde al Sistema Operativo
- Para ejecutar un rutina en modo supervisor se utiliza la instrucción SWI, también denominada llamadas a supervisor
  - Llamada al Supervisor para sacar un carácter por pantalla  
`SWI        SWI_WriteC        ; output r0[7:0]`
  - Punto de retorno de un programa de usuario  
`SWI        SWI_Exit        ; Retorno al programa monitor`
  - Vector de interrupción del SWI **0x00000008**
  - Selección de la función mediante la tarea indicada



## Instrucciones de Control: Tablas de salto I

■ Útiles cuando se desea saltar a una rutina, dentro de un conjunto de rutinas, dependiendo de un valor determinado en tiempo de ejecución

■ Salto a una rutina dependiendo del valor de r0

	BL	JUMPTAB	; Llamada a rutina de selección
	...		
JUMPTAB	CMP	r0, #0	; Compara con 0
	BEQ	SUB0	; De ser igual se salta a SUB0
	CMP	r0, #1	; Compara con 1
	BEQ	SUB1	; De ser igual se salta a SUB1
	...		

■ Solución optimizada: uso del PC como registro PG

	BL	JUMPTAB	; Llamada a rutina de selección
	...		
JUMPTAB	ADR	r1, SUBTAB	; r1 → SUBTAB
	CMP	r0, #SUBMAX	; Check for overrun
	LDRLS	PC, [r1, r0, LSL #2]	; Carga en PC la dirección
	B	ERROR	; Salta a tratar el error
SUBTAB	DCD	SUB0	; SUB0 entry point
	DCD	SUB1	; SUB1 entry point
	DCD	SUB2	; SUB2 entry point



# Instrucciones de Control: Tablas de salto II

## ■ Uso de la *pseudo-instrucción* **DCD**

- Provoca que el ensamblador reserve una palabra inicializada al valor de la expresión que se especifica a su derecha
- Solución estudiada : optimizada con frecuencia de salto

BL JUMPTAB ; Llamada a rutina de selección

JUMPTAB   <sup>...</sup>ADR    r1, SUBTAB       ; r1 → SUBTAB

CMP       r0, #SUBMAX       ; Check for overrun

LDRLS     PC, [r1,r0, LSL #2] ; Carga en PC la dirección

B         ERROR             ; Salta a tratar el error

SUBTAB    DCD       SUB0       ; SUB0 entry point

          DCD       SUB1       ; SUB1 entry point

          DCD       SUB2       ; SUB2 entry point

- Solución alternativa: optimizada con frecuencia de error

CMP       r0, #SUBMAX       ; Check for overrun

BHI       ERROR             ; Salta a tratar el error


LDR       PC, [r1,r0, LSL #2] ; Carga en PC la dirección



## Ejemplo de programa en ensamblador: “Hello Word”

### Código en C de “Hello Word”

```
int    main()                ; Cabecera
{                                           ; Punto de entrada del programa
    printf("Hello Word");    ; Cuerpo del programa
    return 0;                ; Final del programa
}                               ; Final del código de programa
```



```
AREA    HelloW, CODE, READONLY    ; Cabecera
SWI_WriteC EQU    &0                ; SWI Output character in r0
SWI_Exit EQU    &11                ; SWI Finish program
ENTRY   ; Punto de entrada del programa
START  ADR    r1, TEXT                ; r1 → TEXT
LOOP   LDRB   r0, [r1], #1            ; Lee carácter (byte)
       CMP   r0, #0                ; Comprobar si es último
       SWINE SWI_WriteC                ; Imprime carácter
       BNE   LOOP                  ; Seguimos en el bucle
       SWI   SWI_Exit                ; Final de programa
TEXT   =      "Hello Word", &0a, &0d, 0
END                                           ; Final del código de programa
```



# Ejemplo de programa en ensamblador: Comentarios

## Comentarios al programa desarrollado

- Uso de la directiva **AREA** para la declaración del código. Uso de atributos *CodeName*, *CodeType*, *CodeMode*

**AREA**      HelloW, CODE, READONLY

- Uso de la directiva EQU para la declaración de símbolos. En códigos extensos estos símbolos pueden estar declarados en otros módulos y ser incluidos en el programa

- Sintaxis: *SymbolName* EQU *Value*

- Uso de la directiva **ENTRY** para la declaración del punto de entrada al programa. PC → ENTRY

- Uso de la *pseudo-instrucción* “=” para la declaración “inicializada” de variables en memoria. Uso de una etiqueta para su referencia. Cadena de caracteres → delimitador “0”

TEXT      =      “Hello Word”, &0a, &0d, 0

- Uso de la directiva **END** para finalizar el ensamblado





## Ejemplo de programa en ensamblador: *Block Copy*

Copia de una zona de memoria (TABLE1) en otra zona (TABLE2)

```

AREA BlkCpy, CODE, READONLY           ; Cabecera
SWI_WriteC EQU &0                      ; SWI Output character in r0
SWI_Exit EQU &11                       ; SWI Finish program
ENTRY                                   ; Punto de entrada del programa
ADR r1, TABLE1                       ; r1 → TABLE1
ADR r2, TABLE2                       ; r2 → TABLE2
ADR r3, T1END                         ; r3 → T1END
LOOP1 LDR r0, [r1], #4                 ; r0 ← word
      STR r0, [r2], #4                 ; Copia palabra
      CMP r1, r3                     ; Condición de finalización
      BLT LOOP1                      ; Seguimos en el bucle
      ADR r1, TABLE2               ; r1 → TABLE2
LOOP2 LDR r0, [r1], #1                 ; r0 ← byte
      CMP r0, #0                     ; Condición de finalización
      SWINE SWI_WriteC                ; Imprime carácter
      BNE LOOP2                     ; Seguimos en el bucle
      SWI SWI_Exit                   ; Final de programa
TABLE1 = "Esta es la buena", &0a, &0d, 0
T1END
TABLE2 ALIGN = "Esta es la mala", 0 ; Alineamiento de palabras
END                                   ; Final del código de programa
```