Microcontroladores

Unidad 2

Programación en Lenguaje Ensamblador

M. C. Miguelangel Fraga Aguilar

Modos de direccionamiento

- Modo de direccionamiento es la manera en que una instrucción obtiene los operandos sobre los que trabajara
- Formato de la instrucción mov
  mov fuente,destino
- Ejemplos:
  - mov r4,r5 ;Copia el contenido de r4 a r5
  - mov #0x1234,r4 ;Copia la constante 0x1234 a r4
  - mov &0x0200,r4 ;lee la localidad 0x200 a r4

Los operandos pueden ser:

- El contenido de un registro (Modo registro)
- El contenido de una localidad de memoria (Modos Indexado, absoluto, simbólico y relativo)
- Una constante que se almacena después del código de instrucción (Modo inmediato)

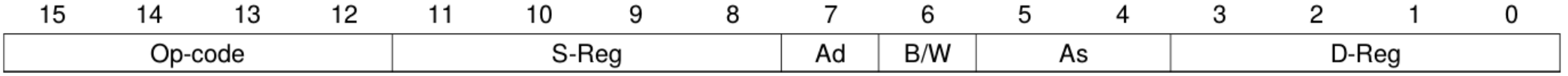
Modos de direccionamiento

As/Ad	Modo	Sintaxis	Descripción
00/0	Registro	Rn	El operando es el contenido del registro
01/1	Indexado	X(Rn)	(Rn+X) apuntan al operando. X se almacena en la siguiente palabra
01/1	Simbólico	ADDR	(PC+X) apuntan al operando. X se almacena en la siguiente palabra
01/1	Absoluto	&ADDR	La palabra siguiente a la instrucción contiene la dirección absoluta. X se almacena en la siguiente palabra. Se usa el modo indexado X(SR)
10/-	Indirecto por registro	@Rn	Rn es usado como apuntador al operando
11/-	Indirecto con auto incremento	@Rn+	Rn es usado como apuntador al operando. Rn es incrementado en 1 para instrucciones .b y en 2 para instrucciones. w
11/-	Inmediato	#N	La palabra siguiente a la instrucción contiene la constante inmediata N. Se usa el modo indirecto con auto incremento @PC+

Ejemplos en guía del usuario

Formatos de instrucciones

- Formato I – instrucciones con dos operandos



- Formato II – Instrucciones con un operando

- Saltos condicionales

Esqueleto de un programa en ensamblador para IAR

```
#include "msp430.h"                     ; #define controlled include file

        NAME    main                    ; module name

        PUBLIC  main                    ; make the main label vissible
                                        ; outside this module
        ORG     0FFFEh
        DC16    init                    ; set reset vector to 'init' label

        RSEG    CSTACK                  ; pre-declaration of segment
        RSEG    CODE                    ; place program in 'CODE' segment

init:   MOV     #SFE(CSTACK), SP        ; set up stack

main:   NOP                             ; main program
        MOV.W   #WDTPW+WDTHOLD,&WDTCTL  ; Stop watchdog timer
        JMP $                           ; jump to current location '$'
                                        ; (endless loop)
        END
```

Directivas del ensamblador en IAR

- No son instrucciones del procesador, son ordenes para el ensamblador
- ORG – indica la dirección de memoria en la que se va a almacenar el resultado de las siguientes lineas
- DC16 – almacena una constante de 16 bits
- Etiquetas. Cualquier palabra no reservada que comience en la columna cero y que termine en : se llama una etiqueta. Es un nombre simbolico para la dirección de memoria en donde se almacena el resultado de la instrucción o directiva de dicha linea.
- Es conveniente usar etiquetas para referirse a las direcciones de memoria de variables o instrucciones

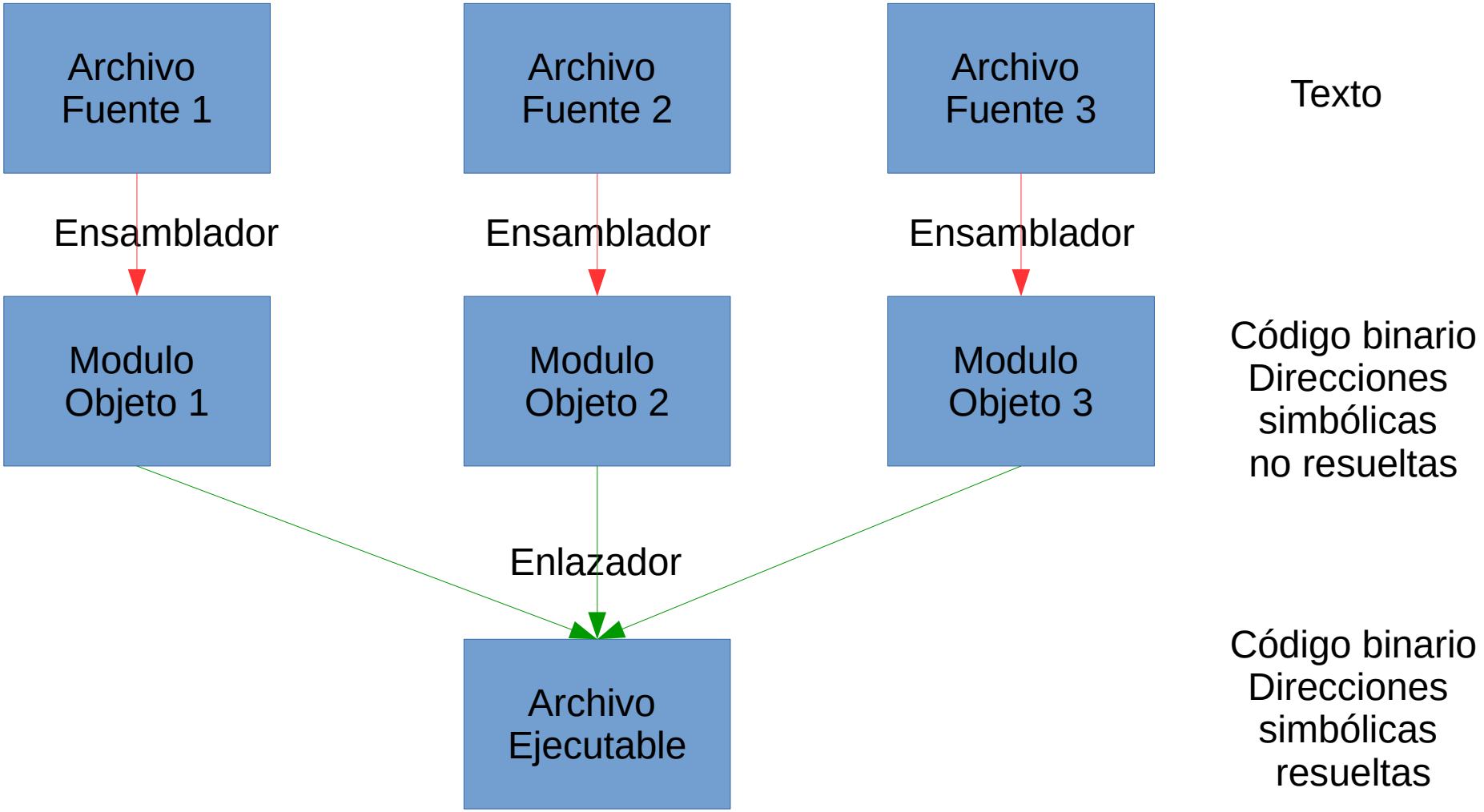
Esqueleto de un programa en ensamblador para CCS

```
;-------------------------------------------------------------------------------
; MSP430 Assembler Code Template for use with TI Code Composer Studio
;
;
;-------------------------------------------------------------------------------
            .cdecls C,LIST,"msp430.h"       ; Include device header file

;-------------------------------------------------------------------------------
            .def    RESET                   ; Export program entry-point to
                                            ; make it known to linker.
;-------------------------------------------------------------------------------
            .text                           ; Assemble into program memory.
            .retain                         ; Override ELF conditional linking
                                            ; and retain current section.
            .retainrefs                     ; And retain any sections that have
                                            ; references to current section.

;-------------------------------------------------------------------------------
RESET       mov.w   #__STACK_END,SP         ; Initialize stackpointer
StopWDT     mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer


;-------------------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------------------
CICLO   jmp CICLO

;-------------------------------------------------------------------------------
; Stack Pointer definition
;-------------------------------------------------------------------------------
            .global __STACK_END
            .sect   .stack

;-------------------------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------------------------
            .sect   ".reset"                ; MSP430 RESET Vector
            .short  RESET
```

Directivas de CCS

- .text Marca en inicio del segmento de codigo, que en el MSP430 se almacena en memoria Flash
- .data marca el inicio del segmento de datos que se almacena en la RAM
- .short almacena una constante de 16 bits
- Las etiquetas en CCS no requieren de : al final

Proceso de ensamblado y ligado (Enlazado)



Código relocalizable: puede ejecutarse en diferentes direcciones de memoria

Ensamblado y enlazado

- Compilación (C C++) o ensamblado (asm) producen un archivo de código objeto.
- El código objeto ya incluye las instrucciones en binario, pero no todas las etiquetas han sido resueltas a una dirección fija.
- El enlazador une los archivos de código objeto y las librerías que se necesitan para formar el código ejecutable. Se resuelven todas las direcciones.
- Libreria: solo se incluyen las funciones que se usan.

Otras directivas

- RSEG – Marca el inicio de una segmento relocalizable
  - CSTACK – segmento de pila rw - 0x3B0 – 0x3FF
  - CODE - Segmento de codigo r – 0xC000- 0xFF00
  - DATA16_N -segmento de datos rw – 0x200-0x3AF
- DS8 DS16 DS32 DS64 apartan espacio no inicializado – Equivalente a declarar la variable sin valor inicial

Declaración de Variables en IAR

Variables sin valor inicial

```
        RSEG    DATA16_N                ;segmento de datos no inicializados
                                        ;Se resuelve a RAM
var16:  DS16    4
var8:   DS8     1

        RSEG    CSTACK                  ; pre-declaration of segment

        RSEG    CODE                    ; place program in 'CODE' segment
                                        ; Se resulelve a FLASH
con16:  DC16    1,2,3,4
con8:   DC8     5,6,7,8

init:   MOV     #SFE(CSTACK), SP        ; set up stack

main:   NOP                             ; main program
        MOV.W   #WDTPW+WDTHOLD,&WDTCTL  ; Stop watchdog timer
        MOV     con16,R4
```

Constantes con valor inicial

Declaración de Variables en CCS

```
 8 ;-------------------------------------------------------------------------
 9             .def    RESET                   ; Export program entry-point to
10                                             ; make it known to linker.
11
12 ;-------------------------------------------------------------------------
13 ;Variables almacenadas en RAM
14             .data
15             .bss  var1,1       ;variable de 8 bits
16             .bss  var2,2       ;variable de 16 bits
17 ;-------------------------------------------------------------------------
18             .text                           ; Assemble into program memory.
19             .retain                         ; Override ELF conditional linking
20                                             ; and retain current section.
21             .retainrefs                     ; And retain any sections that have
22                                             ; references to current section.
23 ;-------------------------------------------------------------------------
24 ; Constantes almacenadas en FLASH
25 CON1         .byte 0x12          ;Constante de 8 bits
26 CON2         .short 0x5678       ;Constante de 16 bits
27
28 ;-------------------------------------------------------------------------
29 RESET       mov.w   #__STACK_END,SP         ; Initialize stackpointer
30 StopWDT     mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
31
```

Variables sin valor inicial

Constantes con valor inicial

Otras directivas CCS

- .bss reserva espacio para una variable sin valor inicial. Recibe dos parametros: el nombre de la variable y cuantos bytes se reservan para ella
- Las constantes se almacenan en Flash junto con las instrucciones
- . byte declara una constante de 8 bits con valor inicial
- . long declara una constante de 32 bits

Operaciones aritméticas

- Recordar la prioridad lógica aritmética
- MOV fuente, destino fuente → destino
  No modifica banderas
- ADD S,D S+D → D Modifica C V N Z
- ADDC S,D S+D+C → D Modifica C V N Z
- SUB S,D D-S → D Modifica C V N Z
- SUBC S,D D-S-C → D Modifica C V N Z
- INC D D+1 → D Modifica C V N Z
- DEC D D-1 → D Modifica C V N Z

Nota: Como la resta se hace sacando el complemento a dos del sustraendo y sumando, la bandera de acarreo es igual al negado del préstamo de la resta

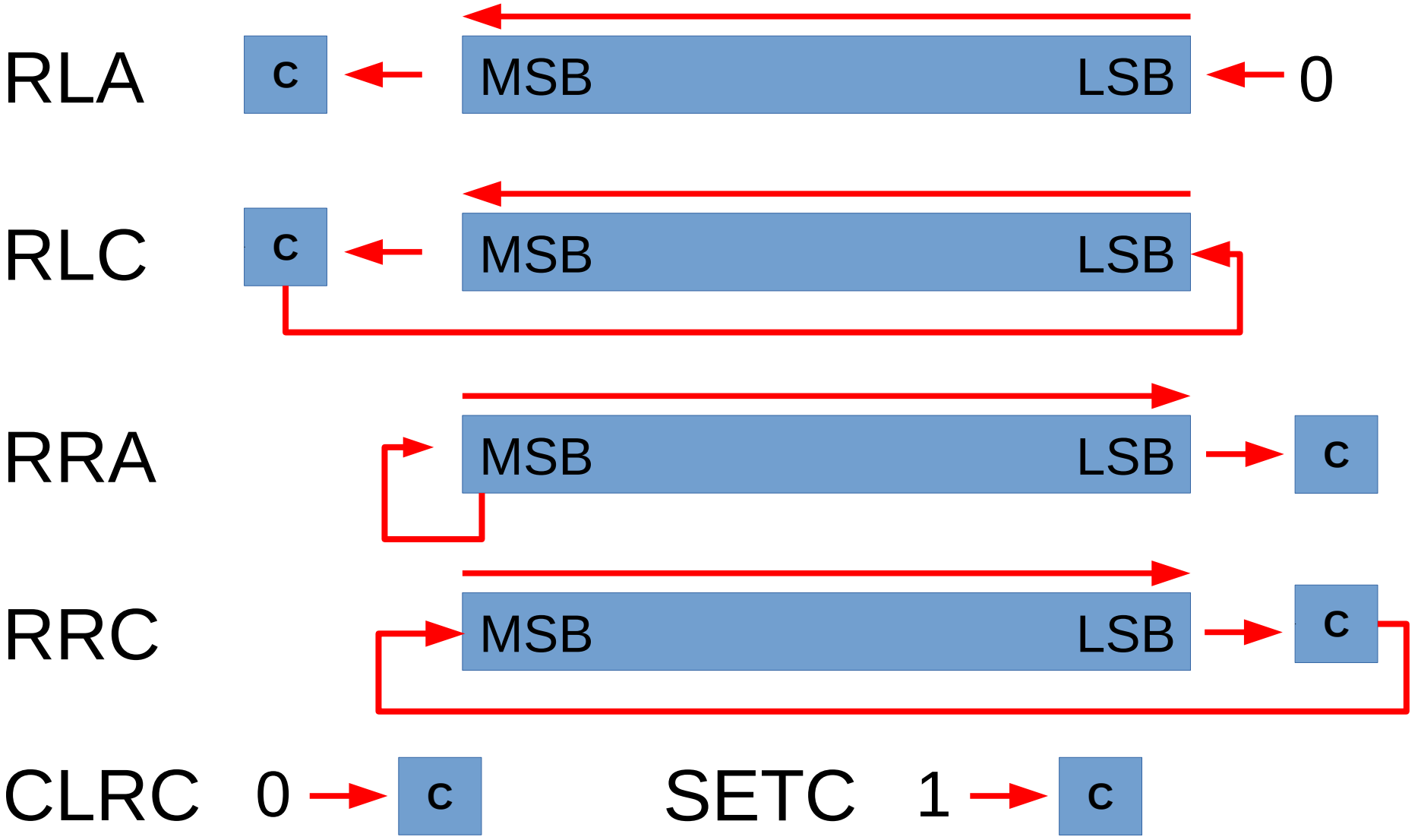
Suma multipalabra IAR

```
        RSEG    CSTACK                  ; pre-declaration of segment
        RSEG    CODE                    ; place program in 'CODE' segment
cop1:    DC32    0xfedcba98              ;operando 1
cop2:    DC32    0x76543210              ;operando 2

init:   MOV     #SFE(CSTACK), SP        ; set up stack

main:   NOP                             ; main program
        MOV.W   #WDTPW+WDTHOLD,&WDTCTL  ; Stop watchdog timer

        MOV     cop1,R4                 ;Suma la palabra baja
        ADD     cop2,R4
        MOV     R4,res                  ;Almacena el resultado
        MOV     cop1+2,R5                 ;Suma la palabra baja
        ADDC    cop2+2,R5
        MOV     R5,res+2                  ;Almacena el resultado

        JMP $                           ; jump to current location '$'
                                        ; (endless loop)
```

Suma multipalabra en CCS

```
21 ;-------------------------------------------------------------------
22 ;definicion de constantes
23
24 CONSA       .long 0x12345678,0xaabbccdd
25 CONSB       .byte 0x01,0x2,0x3,0x4,0x5,0x6,0x7,0x8
26
27 ;-------------------------------------------------------------------
28 RESET       mov.w   #__STACK_END,SP         ; Initialize stackpointer
29 StopWDT     mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
30
31             mov  &CONSA,r4       ;r4=parte baja de A
32             mov  &CONSA+2,r5     ;r5 parte alta de A
33             add  &CONSB,r4       ;suma la parte baja de B
34             addc &CONSB,r5       ;suma la parte alta de B y el acarreo
35             mov  r4,&resultado   ;almacena la parte baja del resultado
36             mov  r5,&resultado+2 ;almacena la parta alta del resultado
37 ;-------------------------------------------------------------------
38 ; Main loop here
```

Rotaciones



RLA, RLC, RRA y RRC modifican N y Z y V=0

Corrimientos multipalabra

- Se tiene un numero de 32 bits en r4 (parte baja) y r5 (parte alta)
- Corrimiento a la izquierda de 32 bits

```
rla r4
rlc r5
```

- Corrimiento a la derecha de 32 bits

Con signo

```
rra r5
rrc r4
```

Sin signo

```
clrc
rrc r5
rrc r4
```

Multiplicación por corrimientos y sumas

- Esta técnica es útil en procesadores sin instrucción de multiplicación y en casos en que es más rápida que la instrucción de multiplicación general
- Recordar que una multiplicación es una suma abreviada. 5*a=a+a+a+a+a
- Un corrimiento a la izquierda equivale a una multiplicación por 2
- Con corrimientos a la izquierda se puede multiplicar por cualquier entero positivo. Ejemplo: 7*a=(a<<1+a)<<1+a=a<<2+a<<1+a
- Multiplicar por un entero negativo implica multiplicar por su valor absoluto y sacar el complemento a dos

Instrucciones de salto

- CMP S,D D-S Modifica C V N Z
- JMP etq Salta incondicionalmente a etq
- JEQ/JZ etq Salta si Z==1 a etq
- JNE/JNZ etq Salta si Z==0 a etq
- JC/JHS etq Salta si C==1 a etq
- JNC/JLO etq Salta si C==0 a etq
- JN etq Salta si N==1 a etq
- JGE etq Salta si (N xor V)==0 a etq
- JL etq Salta si (N xor V)==1 a etq

Si no se cumple la condición, se ejecuta la siguiente instrucción

Prueba de condiciones con comparación y saltos

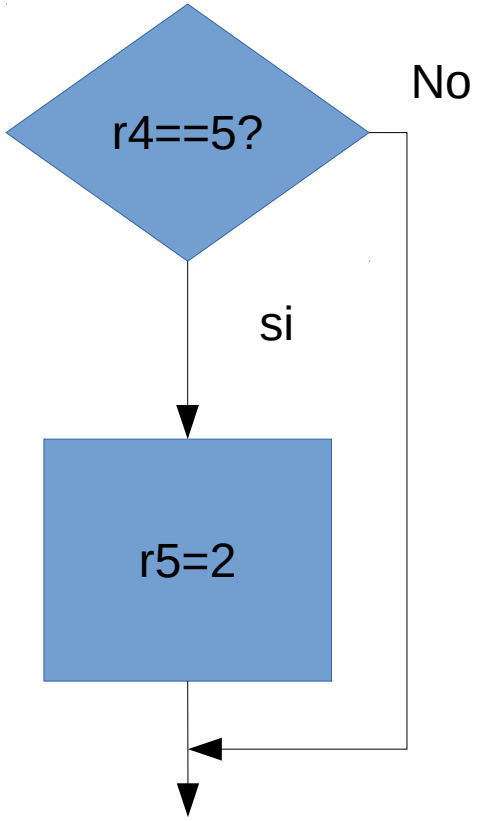
La secuencia de instrucciones

```
cmp r4,r5
jge etiqueta
```

debe interpretarse como:

si R5 es mayor igual que R4 (interpretados como números con signo), entonces brinca a Etiqueta, sino, ejecuta la instrucción siguiente.

Ejemplo estructura if



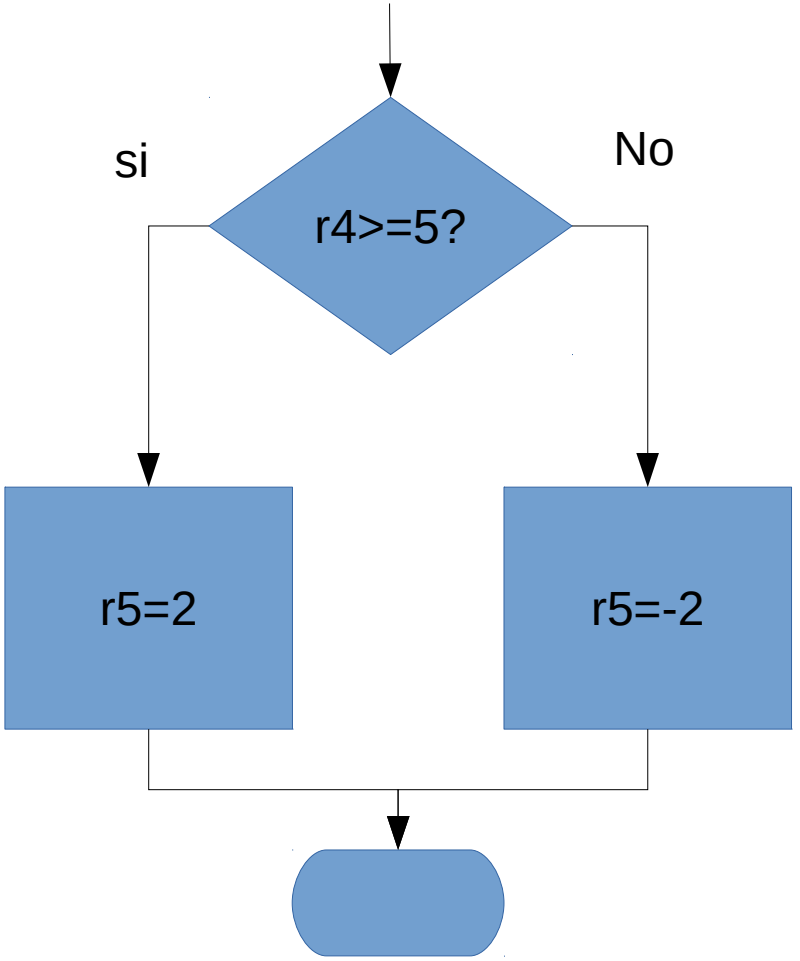

```
    cmp #5,r4
    jne FIN
        mov #2,r5
FIN:
```

Usar la condición contraria

```
    cmp #5,r4
    je SI
    jmp FIN
SI:    mov #2,r5
FIN:
```

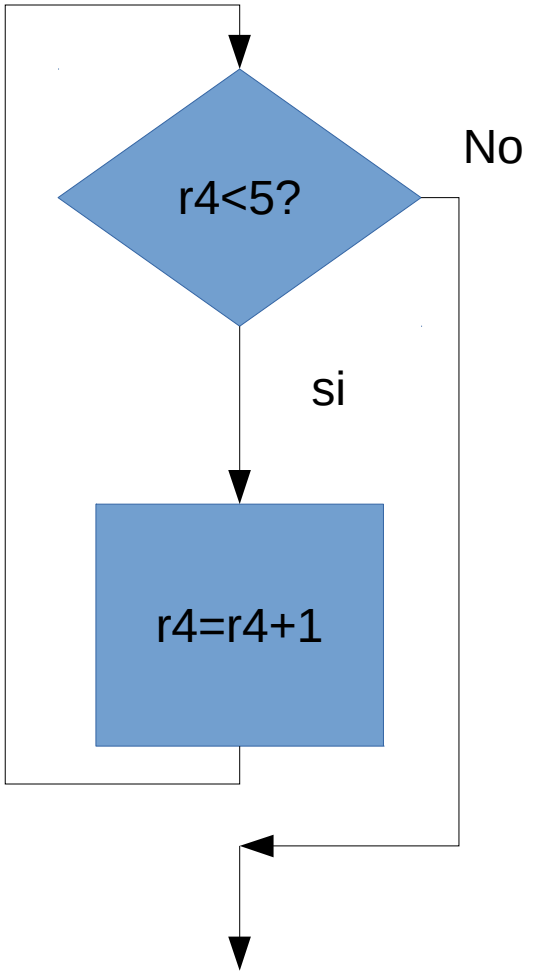
Usar la condición normal para saltar un salto incondicional

Ejemplo estructura if else




```
        cmp #5,R4
        jlo ELSE
            mov #2,R5
        jmp FIN
ELSE:   mov #-2,R5
FIN:
```

Ejemplo estructura while



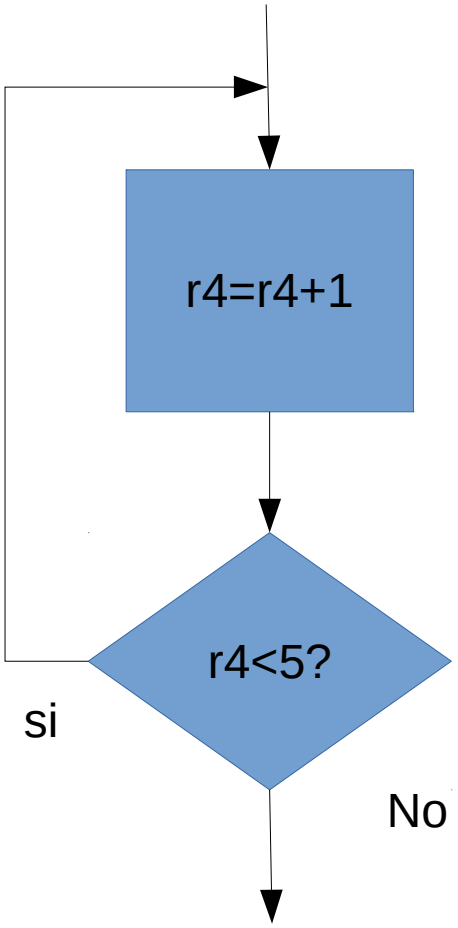

```
WHILE: cmp #5,r4
       jhs FIN
            inc r4
       jmp WHILE
FIN:
```

Usar la condición contraria

```
WHILE  cmp #5,r4
       jlo SI
       jmp FIN
SI:         inc r4
       jmp WHILE
FIN:
```

Usar la condición normal para saltar un salto incondicional

Ejemplo estructura do while




```
DO:     inc r4
        cmp #5,r4
        jlo DO
```

Estructura en C	== / != / >= / <

`if (a op b) {`
`  **Bloque1;**`
`}`
`...`
	`  cmp b,a`
	`  jne/jeq/jlo/jhs FINSI`
	`  **Bloque1**`
	`FINSI: ...`

`if (a op b) {`
`  **Bloque1;**`
`}else{`
`  **Bloque2;**`
`}`
`...`
	`  cmp b,a`
	`  jne/jeq/jlo/jhs ELSE`
	`  **Bloque1**`
	`  jmp FINSI`
	`ELSE:  **Bloque2**`
	`FINSI: ...`

`while (a op b) {`
`  Bloque1;`
`}`
`...`
	`WHILE: cmp  b,a`
	`  jne/jeq/jlo/jhs FINW`
	`  **Bloque1**`
	`  jmp WHILE`
	`FINW: ...`

`do  {`
`  Bloque1`
`}while (a op M);`
`...`
	`DO:  **Bloque1**`
	`  cmp b,a`
	`  jeq/jne/jhs/jlo DO`
	`  ...`

Estructura en C	>	<=

`if (a op b) {`
`  **Bloque1;**`
`}`
`...`
	`  cmp b,a`
	`  jlo FINSI`
	`  jeq FINSI`
	`  **Bloque1**`
	`FINSI: ...`
		`  cmp b,a`
		`  jlo ETQ1`
		`  jne FINSI`
		`ETQ1:  **Bloque1**`
		`FINSI: ...`

`if (a op b) {`
`  **Bloque1;**`
`}else{`
`  **Bloque2;**`
`}`
`...`
	`  cmp b,a`
	`  jlo ELSE`
	`  jeq ELSE`
	`  **Bloque1**`
	`  jmp FINSI`
	`ELSE:  **Bloque2**`
	`FINSI: ...`
		`  cmp b,a`
		`  jlo ETQ1`
		`  jne ELSE`
		`ETQ1:  **Bloque1**`
		`  jmp FINSI`
		`ELSE:  **Bloque2**`
		`FINSI: ...`

`while (a op b) {`
`  Bloque1;`
`}`
`...`
	`WHILE: cmp  b,a`
	`  jlo FINW`
	`  jeq FINW`
	`  **Bloque1**`
	`  jmp WHILE`
	`FINW: ...`
		`WHILE: cmp b,a`
		`  jlo ETQ1`
		`  jne FINW`
		`ETQ1:  **Bloque1**`
		`  jmp WHILE`
		`FINW: ...`

`do  {`
`  Bloque1`
`}while (a op M);`
`...`
	`DO:  **Bloque1**`
	`  cmp b,a`
	`  jlo FINDO`
	`  jne DO`
	`FINDO: ...`
		`DO:  **Bloque1**`
		`  cmp b,a`
		`  jlo DO`
		`  jeq DO`
		`  ...`

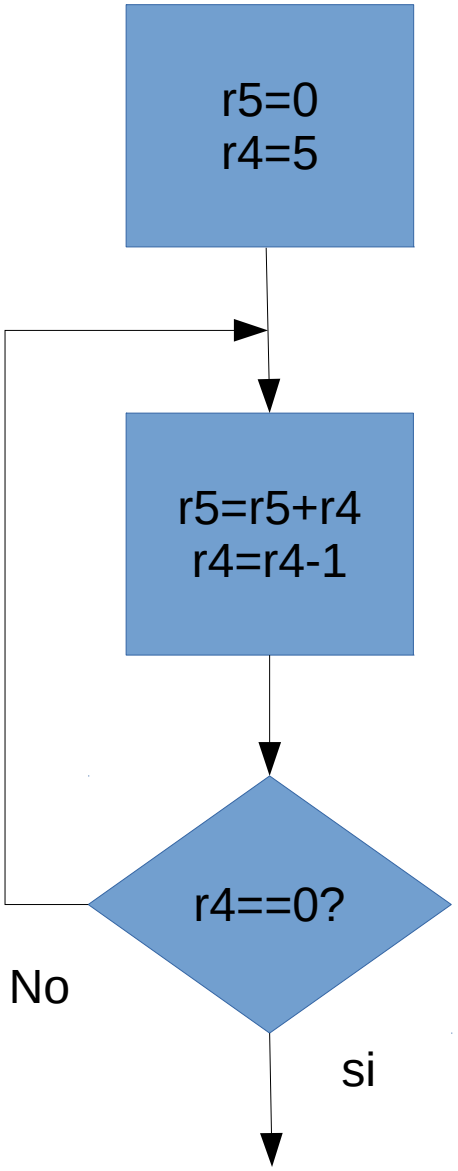
Saltos para números con signo y sin signo

Operador	Con signo		Sin signo	
>=	JGE	Jump if Greater Equal	JHS	Jump if Higher Same
>	JL	Jump if Less than	JLO	Jump if Lower

Ciclo for

- En lenguajes de alto nivel lo usual es que el contador comience en 0 o 1 y se incremente
- En lenguaje ensamblador es más usual que el contador inicie en el número de iteraciones que debe hacer el ciclo for y se decremente hasta llegar a 0

Ejemplo ciclo for



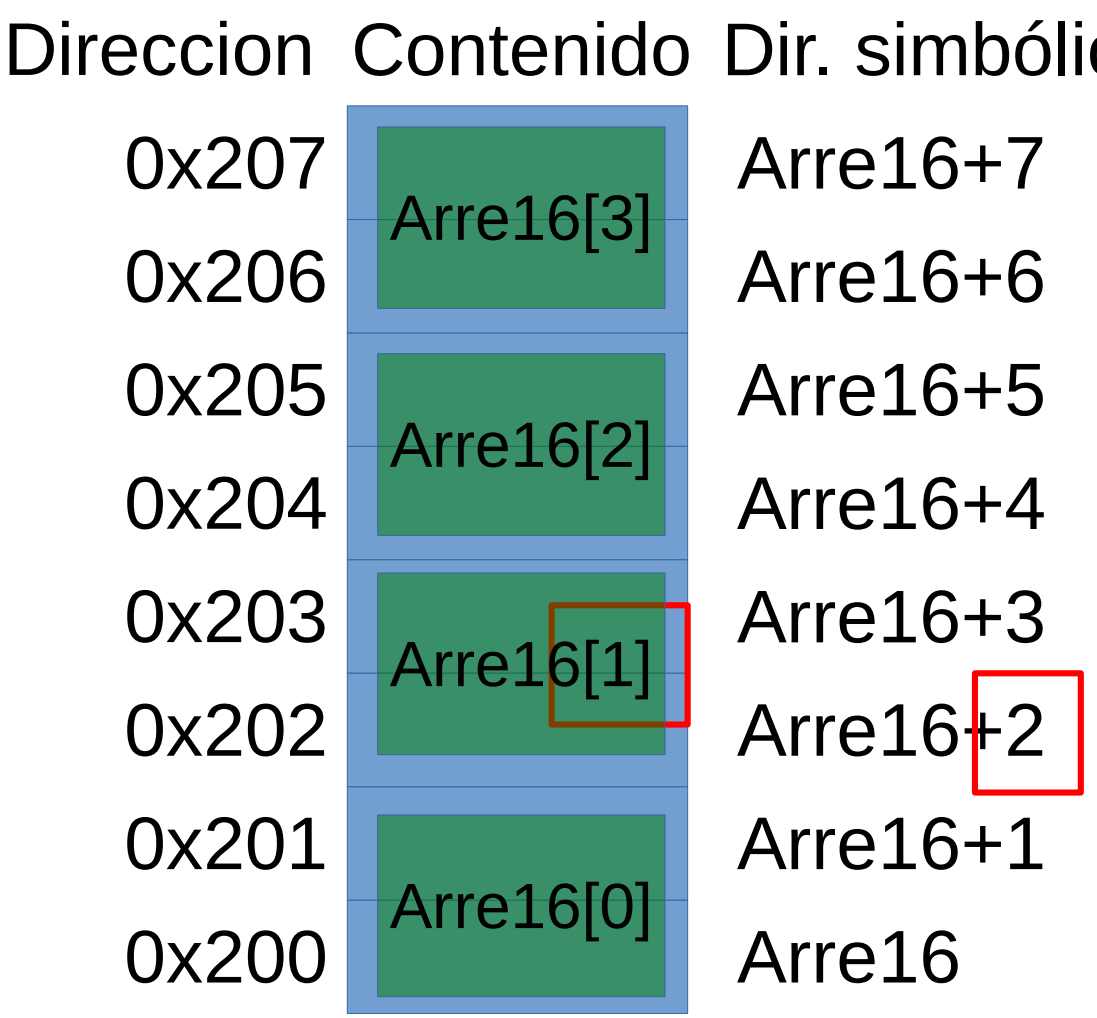

```
        mov #0,r5
        mov #5,r4
FOR:        add r4,r5
            dec r4
        jnz FOR
        ...
```

Manejo de arreglos de números de 8 bits

Direccion	Contenido	Dir. simbólica
0x207	Arre8[7]	Arre8+7
0x206	Arre8[6]	Arre8+6
0x205	Arre8[5]	Arre8+5
0x204	Arre8[4]	Arre8+4
0x203	Arre8[3]	Arre8+3
0x202	Arre8[2]	Arre8+2
0x201	Arre8[1]	Arre8+1
0x200	Arre8[0]	Arre8

- Recordar que cada localidad de memoria puede almacenar solo un byte
- Con números de 8 bits, cada elemento del arreglo ocupa una localidad de memoria
- El indice del elemento y el desplazamiento de la dirección de inicio del arreglo coinciden

Manejo de arreglos de números de 16 bits

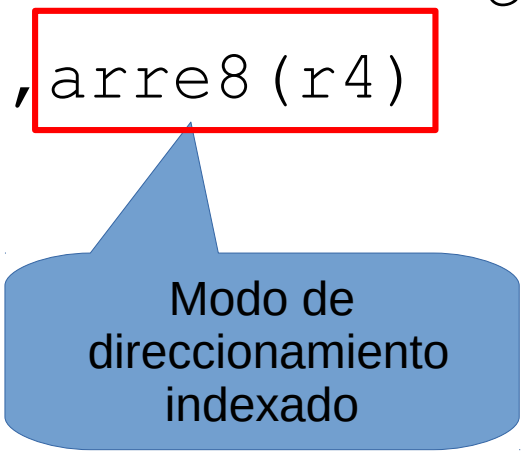


- Con números de 16 bits, cada elemento del arreglo ocupa dos localidades de memoria
- El desplazamiento de la dirección de inicio del arreglo es el doble del indice del elemento

Ejemplo: llenar un arreglo con los valores del 1 al 5

- Números de 8 bits

```
  clr r4
  mov #1,r5
ciclo:
    mov.b r5,arre8(r4)
    inc r5
    inc r4

  cmp #5,r4
  jlo ciclo

  ...
```



- Números de 16 bits

```
  clr r4
  mov #1,r5
ciclo:
    mov r5,arre16(r4)
    inc r5
    inc r4
    inc r4
  cmp #5*2,r4
  jlo ciclo

  ...
```

Instrucciones lógicas bit a bit

- AND s,d s and d → d, modifica N,Z,C V=0
- XOR s,d s xor d → d, modifica N,Z,C,V
- BIT s,d s and d, modifica N,Z,C V=0
- BIS s,d s or d → d, no modifica N,Z,C,V
- BIC s,d not s and d → d, no modifica N,Z,C,V

Ejemplos de manipulación de bits

- Se desea probar el bit 5 del registro de entrada del puerto1, si es cero incrementar r4, sino decrementar r4 :

```
        bit #00100000,&P1IN
        jnz ELSE
            inc r4
        jmp FINSI
ELSE      dec r4
FINSI ...
```

Ejemplos de manipulación de bits 2

- Poner el bit 3 del registro de salida del puerto1 en uno:

```
bis #00001000,&P1OUT
```

- Poner el bit 1 del registro de salida del puerto1 en cero:

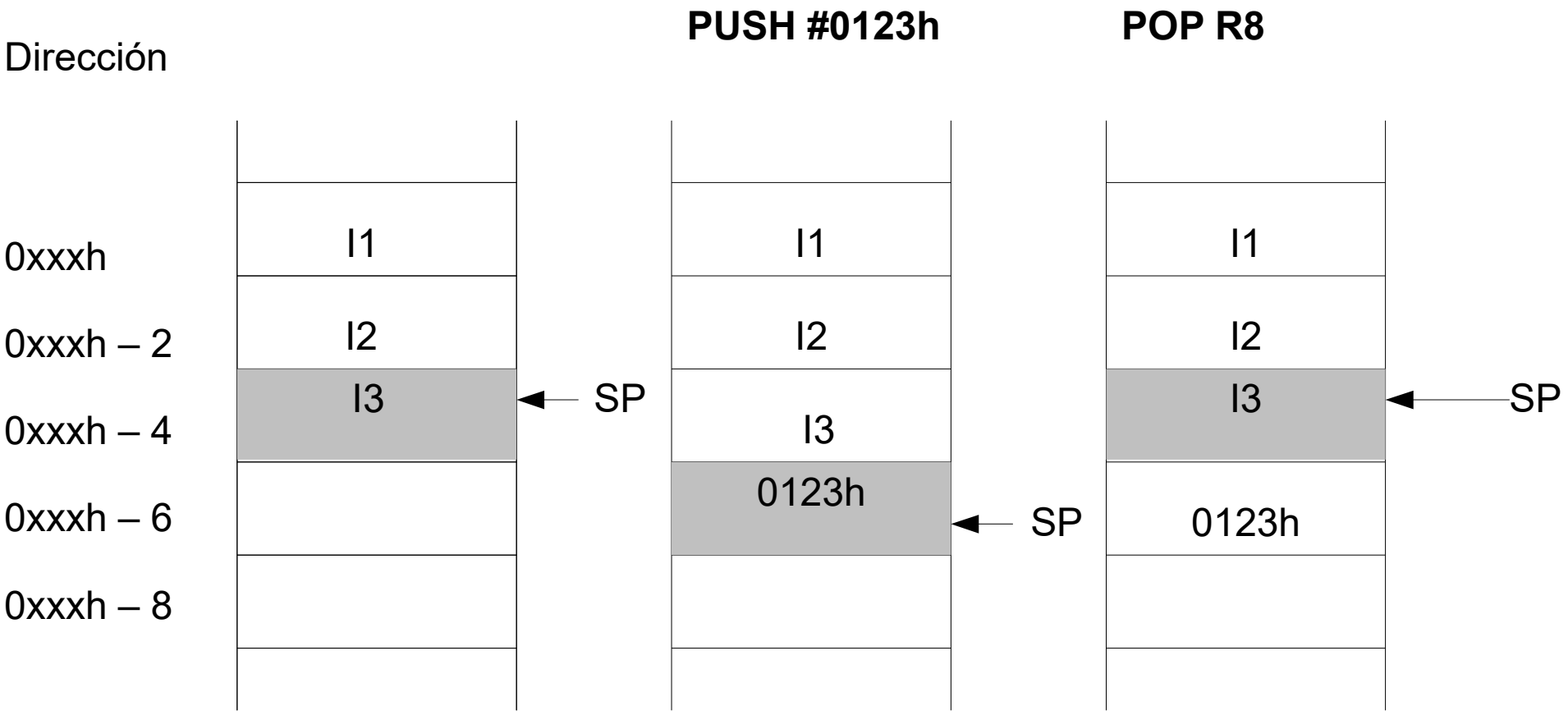
```
bic #00000010,&P1OUT
```

- Negar el bit 6 del registro de salida del puerto1:

```
xor #01000000,&P1OUT
```

Manejo de Pila del MSP430

- PUSH s Mete una palabra a la pila sp-2 → sp, s → @sp
- POP d Saca una palabra de la pila @sp → d, sp+2 → sp



Otras instrucciones

- CALL etq SP-2 → SP, PC+2 → @SP, etq → PC
  Llamada a subrutina (push PC, jmp etq)
- RET @SP → PC, SP+2 → SP
  Retorno de subrutina (pop PC)
- RETI @SP → SR, SP+2 → SP (POP SR)
  @SP → PC, SP+2 → SP (POP PC)
- SWPB d intercambia bytes
- SXT d extensión de signo

Multiplicación Binaria

- Multiplicación de un número por un bit

```
0101 1100 x0=0000 0000
0101 1100 x1=0101 1100
```

Multiplicación binaria

- Multiplicación de dos números de 4 bits

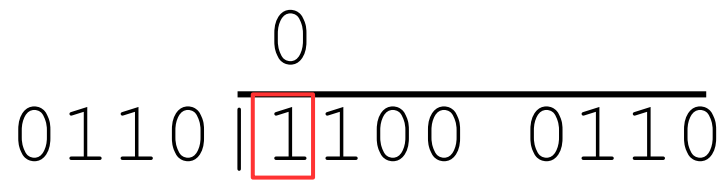
```
   0110
  x1100
------
   0000
  00000
 011000
0110000
-------
1001000
```

Algoritmo de multiplicación por sumas y corrimientos

```
Entradas:
multiplicador y multiplicando

Producto=0

Repetir el numero de bits{
     Producto<<=1;
     si(MSB del multiplicador!=0)
          producto+=multiplicando;
     multiplicador<<=1;

}
```

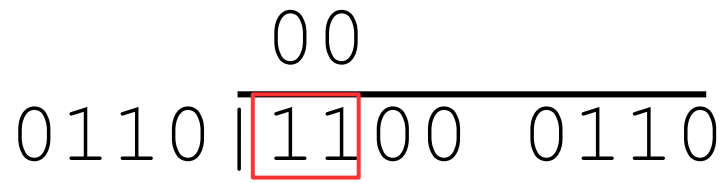
División binaria

- División de dos números de 4 bits



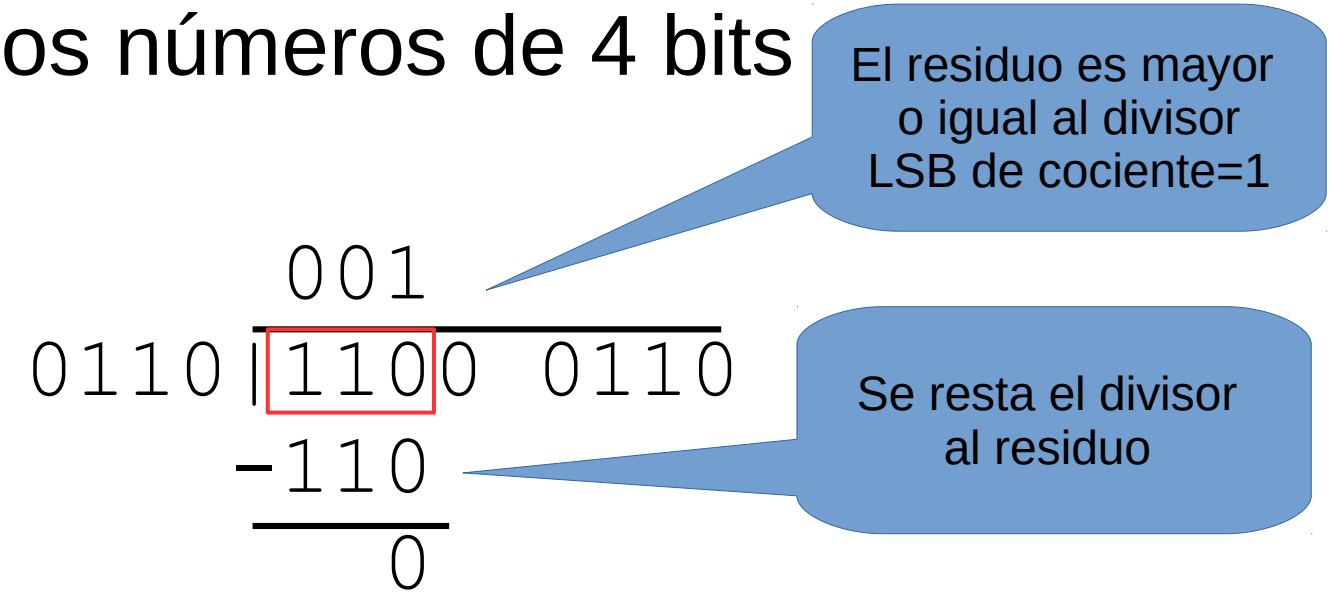
División binaria

- División de dos números de 4 bits



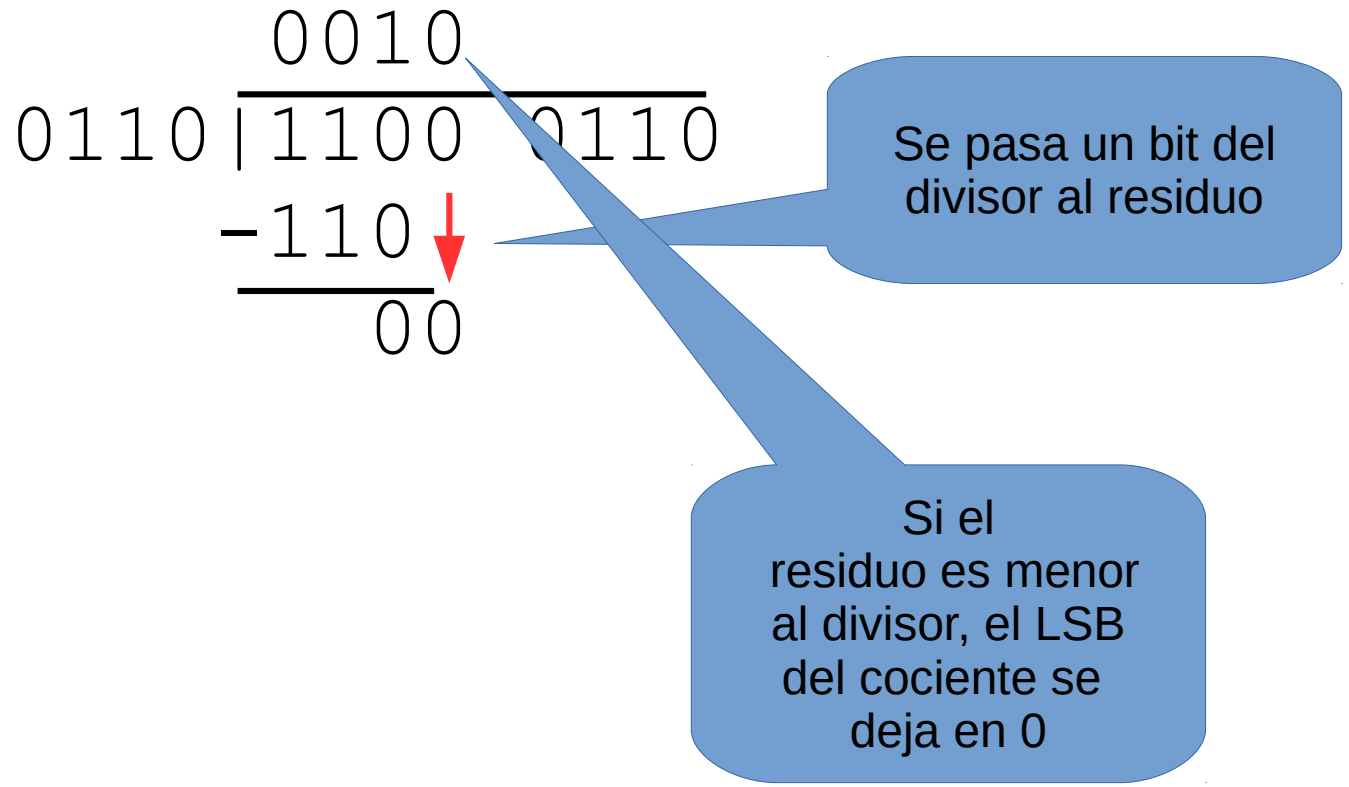
División binaria

- División de dos números de 4 bits



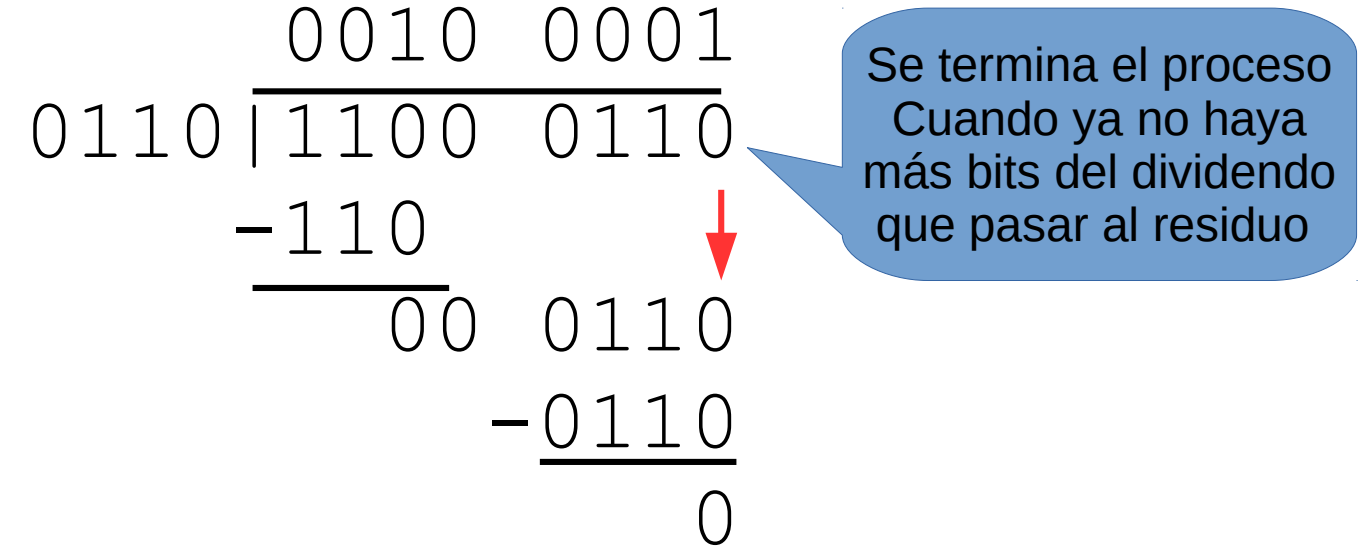
División binaria

- División de dos números de 4 bits



División binaria

- División de dos números de 4 bits



Algoritmo de división por restas y corrimientos

```
Entradas: dividendo y divisor

cociente=0;
residuo=0;
Repetir el numero de bits del dividendo
{
     cociente<<=1;
     Pasar MSB del dividendo al residuo
     si(residuo>=divisor){
          residuo=residuo-divisor;
          LSB de cociente=1;

}
```

Directivas para creación de librerías

- CCS
  - .def exporta un símbolo para que el enlazador lo pueda usar
  - .ref indica que un símbolo esta definido en otro archivo
- IEW
  - PUBLIC exporta un símbolo para que el enlazador lo pueda usar
  - EXTERN indica que un símbolo esta definido en otro archivo

Multiplicador por Hardware

- Periférico externo al CPU capaz de realizar multiplicación sin signo, con signo, Multiplica y acumula con signo y sin signo
- Puede trabajar con todas las combinaciones de operandos de 8 y 16 bits
- La operación que realiza depende de donde se almacene el primer operando
- El resultado esta listo tres ciclos de reloj después de escribir el operando 2

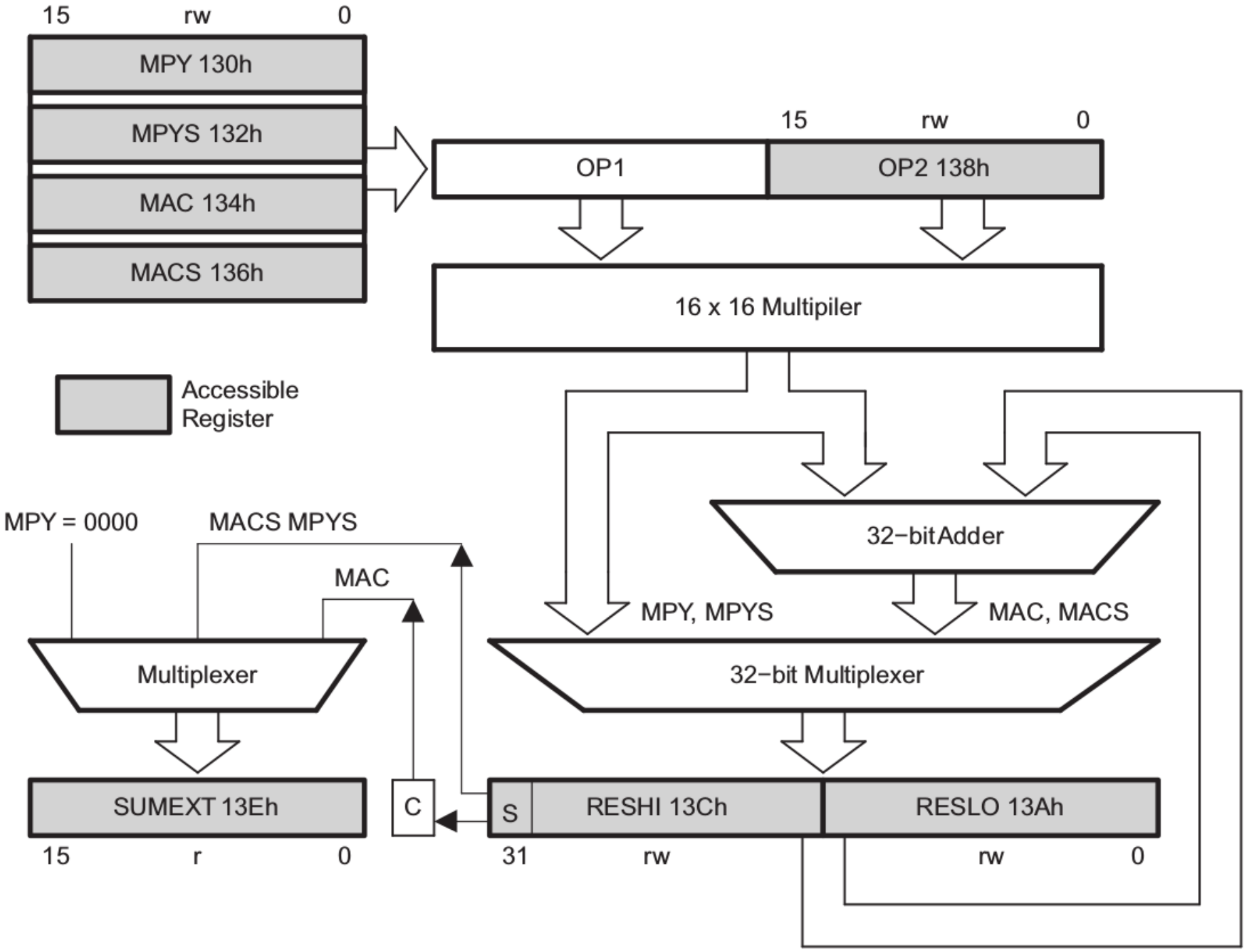


Figure 11-1. Hardware Multiplier Block Diagram

Interrupciones

- Es una llamada a subrutina hecha por el hardware.
- La rutina llamada se conoce como rutina de atención de interrupción
- Un periférico puede tener una o mas interrupciones asociadas
- La dirección de inicio de la rutina se almacena en una dirección de memoria especifica, llamada vector de interrupción

Tabla de vectores de interrupción

- Todos los vectores de interrupción se encuentran agrupados en la tabla de vectores de interrupción
- Cada vector es de 16 bits
- Tambien se encuentra el vector de reset en la tabla de vectores de interrupciones. Este contiene la dirección de la primera instrucción a ejecutar

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
Power-Up External Reset Watchdog Timer+ Flash key violation PC out-of-range(1)	PORIFG RSTIFG WDTIFG KEYV(2)	Reset	0FFFEh	31, highest
NMI Oscillator fault Flash memory access violation	NMIIFG OFIFG ACCVIFG(2)(3)	(non)-maskable (non)-maskable (non)-maskable	0FFFCh	30
Timer1_A3	TA1CCR0 CCIFG(4)	maskable	0FFFAh	29
Timer1_A3	TA1CCR2 TA1CCR1 CCIFG, TAIFG(2)(4)	maskable	0FFF8h	28
Comparator_A+	CAIFG(4)	maskable	0FFF6h	27
Watchdog Timer+	WDTIFG	maskable	0FFF4h	26
Timer0_A3	TA0CCR0 CCIFG(4)	maskable	0FFF2h	25
Timer0_A3	TA0CCR2 TA0CCR1 CCIFG, TAIFG (5)(4)	maskable	0FFF0h	24
USCI_A0/USCI_B0 receive USCI_B0 I2C status	UCA0RXIFG, UCB0RXIFG(2)(5)	maskable	0FFEEh	23
USCI_A0/USCI_B0 transmit USCI_B0 I2C receive/transmit	UCA0TXIFG, UCB0TXIFG(2)(6)	maskable	0FFECh	22
ADC10 (MSP430G2x53 only)	ADC10IFG(4)	maskable	0FFEAh	21
			0FFE8h	20
I/O Port P2 (up to eight flags)	P2IFG.0 to P2IFG.7(2)(4)	maskable	0FFE6h	19
I/O Port P1 (up to eight flags)	P1IFG.0 to P1IFG.7(2)(4)	maskable	0FFE4h	18
			0FFE2h	17
			0FFE0h	16
See (7)			0FFDEh	15
See (8)			0FFDEh to 0FFC0h	14 to 0, lowest