SANTIAGO AYECHU GÁRRIZ

Práctica 3ª: Operaciones Aritméticas y Lógicos

ÍNDICE

Introducción

El objetivo de esta práctica es realizar diferentes operaciones aritméticas como suma, resta, multiplicación y división con número enteros en lenguaje ensamblador AT&T x86-32. También se realizarán diferentes operaciones lógicas bitwise de negación, multiplicación, suma, or-exlusiva así como desplazamiento tanto lógico como aritmético.
Al ejecutar cada una de las instrucciones de cada una de las diferentes operaciones aritméticas veremos cómo van cambiando el registro de banderines de la máquina. Este proceso de cambio de los flags del registro de estado EFLAGS se verá reflejado en la depuración del programa del que se trata, `op_arit_log.s`.

Finalmente, en el programa `saltos.s`, se llevaran a cabo diferentes instrucciones de comparación así como saltos condicionales. Estos se aplicarán en sentencias de lenguajes de alto nivel tipo `if`, `for`, `while`, entre otras. En esta parte es crítico ver el contenido del registro de estado EFLAGS ya que algunos saltos condicionales dependen del valor que tome los banderines de dicho registro.

Desarrollo

Módulo `op_arit_log.s`

A continuación, se muestra las respuestas a las correspondientes preguntas del apartado de autoevaluación. Se adjunta primero el trozo de código que se ejecuta y después se muestra una captura de pantalla del resultado de la ejecución de dicha instrucción.

- La operación aritmética de resta se lleva a cabo con la instrucción sub. En este caso hemos pasado el minuendo al registro EAX y el sustraendo al registro EBX de tal forma que la operación queda: EAX <- EAX – EBX. El resultado de la operación aritmética resta es la siguiente.

```
##  sub: resta
mov $5, %eax
mov $10, %ebx
sub %ebx, %eax
```

Como se puede observar, el registro EAX tiene minuendo mientras que el sustraendo se encuentra en el registro EBX. El resultado de la operación resta se guarda en el registro EAX. Como se puede observar (5 - 10 = -5)

- En el caso de la operación aritmética multiplicación (imul = Integer MULtiplicacion) sabemos por la documentación que:

Opcode	Instruction	Op/En	64-Bit-Mode	Compat/Leg Mode	Description
F6/5	IMUL r/m8	M	Valid	Valid	AX:=AL*r/m byte

En este caso, uno de los operandos se tiene que guardar en el registro AL y el otro operando se lo pasamos con la instrucción `IMUL`. El resultado de dicha multiplicación se guarda en el registro AX.

Una vez sabido eso, se muestra tanto el código a ejecutar como el resultado de la ejecución a continuación:

```
##  imul: multiplicación entera “con signo”: AX <- AL * BL
movb $-3, %bl
movb $5, %al
imulb %bl
```

```
(gdb) p $bl
$10 = -3
(gdb) p $al
$11 = 5
(gdb) n
(gdb) p $ax
$12 = -15
(gdb)
```

Uno de los operandos (-3) se guarda en el registro BL mientras que el otro operando (5) restante se guarda en el registro AL. El resultado de dicha multiplicación se guarda en el registro AX. Al imprimir el contenido del registro en dicho registro se observa que es correcto (-3 * 5 = -15)

- A la hora de realizar una división necesitamos un registro más que cuando se hace una multiplicación, ya que necesitamos guardar tanto el cociente como el resto.

Opcode	Instruction	Op/En	64-Bit-Mode	Compat/Leg Mode	Description
F6/7	IDIV r/m8	M	Valid	Valid	Divide AX by r/m8

					AL:= Quotient AH:= Remainder

Sabiendo como se divide y dónde se guarda tanto el cociente como el resto, se procede a mostrar tanto el código como el resultado de la ejecución del código.

```
##  idiv: división “con signo”. AX / (byte en registro o memoria)
                                AL = Cociente; AH = Resto.


movw $5, %ax     # dividendo
movb $3, %bl     # divisor
idivb %bl        # 5 / 3 = 1(cociente en AL) * 3(divisor) + 2(resto en AH)
```

```
(gdb) p $ax
$15 = 5
(gdb) p $bl
$16 = 3
(gdb) n
(gdb) p $al
$17 = 1
(gdb) p $ah
$18 = 2
(gdb)
```

Como se ha visto de la documentación, el dividendo debe estar almacenado en el registro AX mientras que el divisor es el argumento que se le pasa al ejecutar la instrucción `idiv`, en este caso el contenido del registro BL. Como se observa el contenido de estos registros es 5 y 3, respectivamente. Por tanto la división 5 / 3 el cociente se almacena en el registro AL y el resto en el registro AH. Por ello: 5 / 3 = 1(AL) * 3 + 2 (AH)

- Sabiendo ya cómo se multiplica y cómo se divide y donde se guardan cada uno de los respectivos resultados de las operaciones, se procede a calcular el resultado de la expresión N (N+1) / 2. Donde N es una macro inicialmente definida con valor 5. Como en los casos anteriores, se muestra el código a ejecutar y la ejecución de código a continuación.

```
##  Expresión N*(N+1)/2
movw $N, %bx
movw $(N+1), %ax
imulw %bx        # imulw Op; Op = word; DX:AX <- AX * OP
movw $2, %bx
```

```
## El resultado queda en AX y el resto DX = 0
Idivw %bx          # idivw Op; Op = word; AX <- (DX:AX)/Op; DX := Resto
```

Inicialmente se realiza la multiplicación N*(N+1) y después, el resultado de esta multiplicación se divide entre dos. Se muestra el proceso por pasos.

```
(gdb) p $bx
$19 = 5
(gdb) p $ax
$20 = 6
(gdb) n
(gdb) p $ax
$21 = 30
(gdb) p $dx
$22 = 0
```

Se mueven a registros los operandos. N = 5. N se mueve a registro BX. N+1 se mueve a registro AX. Ya se puede hacer la multiplicación. Sabemos que el resultado de dicha operación va a ser almacenado en los registros DX:AX. Como se observa el resultado (30) se encuentra en el registro AX.

Una vez hecha la multiplicación de N*(N+1) solo resta hacer la división entre 2. Es decir, dividir el contenido del registro AX entre 2. Así pues, se pasa a un registro el divisor y el cociente de dicha división se almacena en el registro AX mientras que el resto se almacena en el registro DX. Como en este caso, 30/2 = 15 el resto es 0, se almacena el cociente (15) en AX y el resto (0) en DX.

```
(gdb) p $bx
$23 = 2
(gdb) n
(gdb) p $ax
$24 = 15
(gdb) p $dx
$25 = 0
(gdb)
```

Cabe destacar que, a diferencia de las multiplicaciones y divisiones realizadas en apartados anteriores, en este apartado se han realizado las multiplicaciones y divisiones con el sufijo WORD (w) que equivale a 2 bytes. La única diferencia es que los registros de los operandos y los registros donde se guardan el resultado de dichas operaciones cambian.

En el caso de la multiplicación (`imulw r/m16`):

```
DX:AX := AX * r/m16 word
```

Mientras que en la división (idivw r/m16):

```
Signed divide DX:AX by r/m16, with results stored in AX :=
Cociente, DX := Resto
```

- Por último, en lo que se refiere al desplazamiento de bits, existen dos posibilidades. El desplazamiento arimético o el desplazamiento lógico. De cada uno de ellos, existe la posibilidad de desplazar tanto a la izquierda como a la derecha. Cabe destacar que cuando se realiza un desplazamiento aritmético a la derecha, en caso de ser un número negativo (primer bit empezando por la izquierda es 1), en vez de "insertar" ceros como ocurre en el resto de los casos, hay que meter unos para hacer una extensión del signo. Para realizar un desplazamiento lógico se utilizan las instrucciones `shl` y `shr` (para desplazar a la izquierda o derecha, respectivamente) y para realizar un desplazamiento aritmético se utilizan las instrucciones `sal` y `sar` (para realizar un desplazamiento aritmético a la izquierda o un desplazamiento aritmético a la derecha, respetivamente). Se muestra el código a ejecutar y su ejecución a continuación:

```
## Desplazamiento de bits
shr $4, %eax      # desplazamiento lógico a la derecha: se introducen 4 bits    por la izquierda
sar $4, %eax      # desplazamiento aritm. a la derecha: se introducen 4 bits
                  por la izquierda
```

Se muestra el resultado a continuación. He decido mostrarlo en formato usando el argumento /t ya que considero que en binario se entiende mejor.

```
(gdb) p /t $eax
$26 = 11111111111111111111111100001111
(gdb) n
(gdb) p /t $eax
$27 = 11111111111111111111111110000
(gdb)
```

En esta primera captura se ha ejecutado el desplazamiento lógico a la derecha. Se ha especificado en la propia instrucción que se desplazan 4 bits. Por ello, los 4 bits de la derecha (1111) "salen" y entran 4 ceros en su lugar por la izquierda. No se muestran al imprimir, pero realmente hay 4 ceros al comienzo.

```
(gdb) p /t $eax
$27 = 11111111111111111111111110000
(gdb) n
(gdb) p /t $eax
$28 = 11111111111111111111111
(gdb)
```

En el caso del desplazamiento aritmético de 4 bits ocurre de una manera similar, pero hay que tener en cuenta la condición de extensión del signo comentada anteriormente. En este caso, como se trata de un número positivo (el bit más

significativo es un 0), no hace falta realizar una extensión de signo. Al igual que en el caso anterior, salen los 4 bits menos significativos y se insertan 4 ceros al comienzo del número binario. Cabe recordar, de nuevo, que no se han impreso los ceros iniciales, pero realmente sí que están.

Módulo `saltos.s`

Para la realización de los ejercicios de este apartado, se han creado dos pequeños programas con las instrucciones indicadas en cada uno de los apartados. Se muestran dichos programas a continuación. En primer lugar, el programa correspondiente al apartado "Registro de Flags" y, a continuación, el programa correspondiente al apartado "Saltos"

```
        ## MACROS
        .equ SYS_EXIT,  1
        .equ SUCCESS,  0

        ## VARIABLES LOCALES
        .data



        ## INSTRUCCIONES
        .global main
        .text
main:

        ## RESET
        xor  %eax,%eax
        xor  %ebx,%ebx



        mov $0xFFFFFFFF, %eax
        shr $1, %eax
        add %eax, %eax
        test $0xFF000000, %eax
        cmpl $0xFFFFFFFF, %eax

        ## SALIDA
        mov $SYS_EXIT, %eax
        mov $SUCCESS, %ebx
        int $0x80
```

```
.end
```

Se muestra el resultado de la ejecución de cada una de las instrucciones a continuación:

mov $0xFFFFFFFF, %eax

```
(gdb) p /x $eax
$1 = 0xffffffff
```

shr $1, %eax

```
(gdb) p /x $eax
$1 = 0xffffffff
(gdb) n

Watchpoint 2: $eflags

Old value = [ PF ZF IF ]
New value = [ CF PF IF OF ]
main () at memoria.s:21
(gdb) p /x $eax
$2 = 0x7fffffff
```

Cuando hacemos un desplazamiento lógico a la derecha, insertamos un 0 por la izquierda y "sacamos" el bit de más a la derecha. Es por ello que inicialmente teníamos una F, equivalente a 1111 en binario y ahora tenemos 0111 equivalente a 8 en hexadecimal. Como es un número distinto de cero, el Zero Flag se desactiva. El LSB 0xFF tiene un número par de 1's por lo que el Parity Flag se mantiene "activado", con bit 1. El último bit salido queda en CF. SF = 0 ya que ha entrado un cero en el MSB.

add %eax, %eax ## EAX <- EAX + EAX

```
(gdb) p /x $eax
$2 = 0x7fffffff
(gdb) n

Watchpoint 2: $eflags

Old value = [ CF PF IF OF ]
New value = [ AF SF IF OF ]
main () at memoria.s:22
(gdb) p /x $eax
$3 = 0xfffffffe
```

test $0xFF000000, %eax

```
(gdb) p /x $eax
$3 = 0xfffffffe
(gdb) n

Watchpoint 2: $eflags

Old value = [ AF SF IF OF ]
New value = [ PF SF IF ]
main () at memoria.s:23
```

La instrucción test afecta a los flags de EFLAGS. Esta instrucción equivale a hacer la operación lógica bitwise AND, pero no guarda el resultado en el operando destino. La instrucción ejecutada equivale a:
`0xFF000000 AND 0XFFFFFFFE = 0XFF000000.`
Se mantiene activado el Sign Flag ya que, en ambos casos, los números expresados en hexadecimal son números negativos, su primer bit del MSB empieza por 1.

cmpl $0xFFFFFFFF, %eax

```
(gdb) p /x $eax
$4 = 0xfffffffe
(gdb) n

Watchpoint 2: $eflags

Old value = [ PF SF IF ]
New value = [ CF PF AF SF IF ]
main () at memoria.s:28
```

La instrucción `cmp` realiza la operación SUB afectando a los flags de EFLAGS, pero no guarda el resultado en el operando destino.
EAX – 0xFFFFFFFF = 0xFFFFFFFE – 0xFFFFFFFF.
Se mantiene activado el Sign Flag ya que, en ambos casos, los números expresados en hexadecimal son números negativos, su primer bit del MSB empieza por 1.

Por otro lado se muestra el programa correspondiente al apartado de Saltos con la ejecución de cada una de las instrucciones, junto con una breve explicación.

```
## MACROS
.equ SYS_EXIT,  1
.equ SUCCESS,  0


## VARIABLES LOCALES
.data



## INSTRUCCIONES
.global main
```

```
        .text

main:
        ## RESET
        xor  %eax,%eax
        xor  %ebx,%ebx

        mov $0x00AA, %ax
        mov $0xFF00, %bx
        cmp %bx, %ax
        ja salto1
        jg salto2

salto1:
        mov $0xFF, %ebx

salto2:
        mov $1, %eax

        ## SALIDA
        int $0x80

        .end
```

mov $0x00AA, %ax

```
(gdb) p /x $al
$1 = 0xaa
```

mov $0xFF00, %bx

```
(gdb) p /x $bx
$2 = 0xff00
```

cmp %bx, %ax

```
(gdb) n

Watchpoint 2: $eflags

Old value = [ PF ZF IF ]
New value = [ CF PF IF ]
main () at memoria.s:22
```

ja salto1

NO CUMPLE CONDICIÓN, POR TANTO, NO HACE EL SALTO A ETIQUETA “salto1”

jg salto2

```
(gdb) n
salto2 () at memoria.s:29
```

salto1: mov $0xFF, %ebx

COMO NO SE HA HECHO EL SALTO A SALTO1, Y SE CUMPLE LA INSTRUCCIÓN DE JUMP GREATER, SE PRODUCE EL SALTO A “SALTO2” POR TANTO EL CODIGO DENTRO DE LA ETIQUETA SALTO1 NO SE EJECUTA.

salto2: mov $1, %eax

```
(gdb) p $eax
$3 = 1
```

int $0x80

```
(gdb) n
[Inferior 1 (process 16106) exited normally]
```

Módulos Fuente Comentados

A continuación, se muestran los dos módulos fuente usados en esta práctica (`op_arit_log.s`, `datos_saltos.s`) con sus respectivos comentarios en rojo.

`op_arit_log.s`

```
### Programa:     op_arit_log.s
### Descripción: Emplear estructuras de datos con diferentes operaciones lógicas y aritméticas.
### Compilación: gcc -m32 -g -o op_arit_log op_arit_log.s

        ## MACROS En PREPROCESAMIENTO, se sustituye MACROS por dato asociado
        .equ SYS_EXIT, 1            Se sustituye donde ponga SYS_EXIT por valor 1
        .equ SUCCESS,   0           Se sustituye donde ponga SUCCESS por valor 0
        .equ N, 5               Se sustituye donde ponga N por valor 5

        ## INSTRUCCIONES
```

```
        .global main            Etiqueta main
        .text
main:

        ## RESET DE REGISTROS

        xor  %eax,%eax  La operación lógica XOR vale 0 si los dos
        xor  %ebx,%ebx  operandos son iguales.
        xor  %ecx,%ecx  Por tanto, haciendo la operación sobre dos
        xor  %edx,%edx  registros iguales, el valor que toma cada
        xor  %esi,%esi  uno de los registros (iguales) es 0.
        xor  %edi,%edi


        ## OPERACIONES ARITMETICAS con NUMEROS ENTEROS

        ## add: suma            Operación aritmética suma
        mov $5,%eax             Mover un sumando a un registro (EAX)
        mov $10,%ebx            Mover el otro sumando a un registro (EBX)
        add %ebx,%eax           Sumar el contenido de registro EBX al contenido de registro EAX y
        guardar el resultado en registro EAX. EAX <- EAX + EBX

        ## sub: resta           Operación aritmética resta
        mov $5,%eax             Mover minuendo a registro EAX
        mov $10,%ebx            Mover sustraendo a registro EBX
        sub %ebx,%eax           Minuendo (EAX) – Sustraendo (EBX) y guarda el resultado en el
        registro destino, EAX. EAX <- EAX - EBX

        ## imul: multiplicación entera      "con signo": AX <- BL * AL (r/m8)
        movb $-3,%bl            Mover un operando a registro BL
        movb $5,%al             Mover el otro operando a registro AL
        imulb %bl               Realizar multiplicación del operando que se le pasa (contenido registro
BL) por el contenido en registro AL,guarda resultado en AX

    ## idiv: división "con signo" (AL=Cociente, AH=Resto) <- AX/(byte en registro o memoria)
        movw $5,%ax             # mover dividendo a registro AX
        movb $3,%bl             # mover divisor a registro BL
        idivb %bl               # realizar división entre dividendo y divisor. Guardar el cociente en
registro AL, y el resto en registro AH.
        5 (dividendo AX) = 1(cociente AL) * 3 (divisor BL) + 2 (resto AH)
```

```
## complemento a 2: equivalente a cambiar de signo negación
    negb %bl                    # complemento a 2 del contenido del registro bl

    ## Expresión N*(N+1)/2      realizar multiplicación N*(N+1) y luego div 2
    movw $N,%bx                 mover un operando, N (N=5), a registro BX
    movw $(N+1),%ax             mover el otro operando, N+1, a registro AX
    imulw %bx                   #imulw Op ; Op=word ; DX:AX<- AX*Op
    movw $2,%bx                 mover divisor (2) a registro BX
    ## El resultado queda en AX y el resto DX=0
    idivw %bx                   resultado de división AX: cociente | DX: resto


    ## OPERACIONES LOGICAS

    mov $0xFFFF1F, %eax             movemos uno de los operandos a registro EAX
    mov $0x0000F1, %ebx             movemos el otro operando a registro EBX
    not %eax                        INVERSION, 10010110 -> 01101001
    and %ebx,%eax                   PRODUCTO LÓGICO
    or  %ebx,%eax                   SUMA LÓGICA

    ## Complemento a 2 mediante operación lógica not()+1
    mov %ebx,%eax               la instrucción neg (complemento a 2) equivale
    not %eax                    a hacer el complemento a 1 y sumarle 1
    inc %eax            para ello, usamos la instrucción not e inc (incrementar)

    ## Desplazamiento de bits
    shr $4,%eax         #desplazamiento lógico: bits a introduccir -> 0..
    sar $4,%eax         #desplazamiento aritmético: OJO -> extensión del signo

    ## SALIDA
    mov $SYS_EXIT, %eax     Código de llamada al s.o.: subrutina exit
    mov $SUCCESS,  %ebx     argumento de salida al s.o. a través de EBX según convenio ABI i386
    int $0x80 llamada al s.o para ejecutar subrutina según el valor de eax

    .end        fin de programa
```

saltos.s

```
    /*
```

```
Program:    saltos.s
Descripción: Uso de diferentes tipos de saltos dependiendo del valor de los        registros de estado
EFLAGS
        */

        ## MACROS En PREPROCESAMIENTO, se sustituye MACROS por dato asociado
        .equ SYS_EXIT, 1        Se sustituye donde ponga SYS_EXIT por valor 1
        .equ SUCCESS, 0         Se sustituye donde ponga SUCCESS por valor 0

        ## VARIABLES LOCALES
        .data               SECCION DE DATOS



        ## INSTRUCCIONES
        .global main            ETIQUETA MAIN -> punto de entrada programa
        .text                   Sección de Instrucciones
main:

        ## RESET

        xor  %eax,%eax  La operación lógica XOR vale 0 si los dos
        xor  %ebx,%ebx  operandos son iguales.
        xor  %ecx,%ecx  Por tanto, haciendo la operación sobre dos
        xor  %edx,%edx  registros iguales, el valor que toma cada
        xor  %esi,%esi  uno de los registros (iguales) es 0.
        xor  %edi,%edi  Otra forma de inicializar a 0 -> sub %edi, %edi


        ## FLAGS DEL REGISTRO DE BANDERINES EFLAGS
        /*
        los flags se activan al realizar operaciones aritméticas, lógicas, etc dependiendo del resultado de
dicha operación

CF: El resultado de la operación tiene llevada del bit MSB del destino
OF: El resultado de la operación con signo se  desborda, su tamaño supera el permitido.
ZF: el resultado de la operación  tiene  valor  cero
SF: el resultado de la oeración tiene valor  negativo
PF: el resultado de la operación tiene el byte LSB con un  número par de bits
        */
        xor %eax,%eax   como resultado cero, ON -> ZF y PF; OFF -> CF,OF,SF
```

```
        inc %eax            resultado != 0 -> desactiva ZF y PF
        neg %eax  ON -> SF,PF y CF:resta de la definición de C2 :(0-N)
        shr $1,%eax         desplz. lógico dcha: desplaza n bits el operando destino

/* Salen bits por la dcha y entran ceros por la izda.
 El último bit salido queda en CF.
 SF=0 ya que ha entrado un cero en el MSB
 MANUAL INTEL: http://www.cs.nyu.edu/~mwalfish/classes/ut/s13-cs439/ref/i386/SAL.htm
 For SHR, OF is set to the high-order bit of the original operand.
        OF=MSB=1
 The OF flag is affected only on 1-bit shifts.
 Equivale a dividir 2^n si desplazo a la dcha y a multiplicar 2^n hacia la izda (posible overflow).
*/
        shl $1,%eax             desplazar (lógicamente) 1 bit a la izquierda EAX
        clc                 clear CarryFlag -> CF=0
        xor %eax,%eax           resultado cero -> ON -> ZF y PF; OFF -> CF,OF,SF
        movw $0xFFFF,%ax        La instrucción MOV NO afecta a ningún flag
        addw $0xFFFF,%ax        activa SF y CF pero no OF
        clc                     clear CarryFlag -> CF=0
        movw $0x7FFF,%ax        mover 0x7FFF a registro AX -> no afecta flags
        addw $1,%ax             ON -> OF, SF


        ## INSTRUCCIONES COMPARATIVAS: TEST,CMP

        ## Comprobar si el bit de la posicion 5  es cero con la mascara 0x0010 que aisla dicha posicion
        ## test realiza la operación AND afectando a los flags de EFLAGS pero no guarda el resultado en
el operando destino
        movw $0xABFF, %ax       mover 0xABFF a registro AX
        movw $0x0BCF, %bx       mover 0x0BCF a registro AX
        test $0x0010, %ax       AX·0x0010=0x0010>0 -> SF=0, LSB=0x10 impar->PF=0,
                                El manual dice -> The OF and CF flags are cleared
        test $0xFFFF, %ax       SF=1 xq AX·0xFFFF=AX<0, LSB=AL= par-> PF=1
        test $0b0000000000010000, %bx   SF=0; FNT·DEST>0,ZF=1 PF=0

        ## Comprobar si el valor de una variable es mayor, menor o igual al valor 0x00FF
        ## cmp realiza la operacion SUB afectando a los flags de EFLAGS pero no guarda el resultado
en el operando destino
        ## SUB: It evaluates the result for both signed and unsigned integer operands and sets the OF
and CF flags
```

```
        ## to indicate an overflow in the signed or unsigned result, respectively

        movw $0x01FF, %ax       mover a registros los respectivos operandos
        movw $0x0001, %bx       que se usaran a continuación para el uso de
        movw $0x00FF, %cx       la instrucción cmp; mov no altera banderines
        cmp  $0x00FF, %ax       AX-0x00FF=0x0100>0 -> ZF=0,SF=0, LSB=00 -> PF=1
        cmp  $0x00FF, %bx       BX-0x00FF=0xFF02<0 -> SF=1, LSB=0x02 impar -> PF=0,
                                unsigned overflow -> CF=1, signed not overflow OF=0
        cmp  $0x00FF, %cx       CX-0x00FF=0 -> ZF=1, SF=0, LSB=0xFF->PF=1,CF=0,OF=0

        ## SALTOS CONDICIONALES

        movw $0x01FF, %ax       mover operandos a registros (ax, bx, cx)
        movw $0x0001, %bx       para realizar distintos saltos condicionales
        movw $0x00FF, %cx       dependiendo del valor de los banderines.
        cmp  $0x00FF, %ax       AX-0x00FF=0x0100>0, luego ZF=0 y SF=0, PF=1
        jg   salto4             great jump ->como SF = 0 ->salta porque AX > 0x00FF
        nop                     como salta, no se ejecuta esta instrucción
salto4:
        cmp  $0x00FF, %bx       BX-0x00FF=0x0001+0xFF01=0xFF02<0, ZF=0 , SF=1,
                                unsigned over CF=1 y not signed over OF=0
        jl   salto5             JumpLess->resta de numeros con signo->SF=1 -> salta
        nop                     no se ejecuta, salto a etiqueta salto5
salto5:
        movw $0x8000, %ax       0x8000 vale -32768 con signo y 32768 sin signo
        cmp  $0x0001, %ax       Signed->0x8000-0x1 = 0x8000+0xFFFF=0x7FFFF>0; SF=0

OF=1 ya que la suma de dos negativos ha dado positivo
CF=0 ya que en binario puro 0x01FF-0x00001=0x01FE, no overflow
0xFF es par -> PF=1

        ja salto6 above jump -> resta de números sin signo -> 32768-1>0
        nop
salto6:
        cmp  $0x00FF, %cx       CX-0x00FF = 0, luego ZF=1 y SF=0
        je   salto7             equal jump -> como ZF = 1 -> salta
        nop


        ## SALIDA
```

```
salto7:
        mov $SYS_EXIT, %eax      Código de llamada al s.o.: subrutina exit
        mov $SUCCESS,  %ebx argumento de salida al s.o. a través de EBX según convenio ABI i386
        int $0x80 llamada al s.o para ejecutar subrutina según el valor de eax


        .end      FIN PROGRAMA
```

Comandos de Compilación

A continuación, se muestran los comandos usados para la compilación de ambos programas sobre los que trata esta práctica.

- Los programas `op_arit_log.s` y `datos_saltos.s` se compilaron haciendo uso del Toolchain automático, con el siguiente comando:

```
gcc -m32 -g -o op_arit_log op_arit_log.s
gcc -m32 -g -o datos_saltos datos_saltos.s
```

  Añadiendo los siguientes argumentos:

  -m32: módulos fuente y objeto para la arquitectura i386.
  -g para cargar tabla de símbolos

  Estos comandos nos generan el módulo binario ejecutable (`op_arit_log` y `datos_satos`) listos para ser cargados en memoria.

  Cabe destacar que al no tener un punto de entrada `_start` en ambos programas, no se ha añadido `-nostartfiles`

Historial Comandos GDB + Salida

A continuación, se muestra el `.txt` generado a partir de la depuración del módulo `op_arit_log.s`

Cabe destacar que también se adjunta los comandos, junto a su salida, con sus respectivos comentarios (en rojo). Cuando hacemos un examine sobre una dirección de memoria, volcamos el contenido en memoria a partir de la dirección de memoria indicada en el comando eXamine.

```
+file op_arit_log          cargar a la ventana el modulo op_arit_log
Reading symbols from op_arit_log...
+layout regs               cargar Ventana de registros
+focus cmd                 ”enfocar” a terminal
Focus set to cmd window.
+b main punto de interrupción en etiqueta main
```

```
Punto de interrupción 1 at 0x118d: file op_arit_log.s, line 20.
+run              empezar a ejecutar el programa (desde el punto de interrupción)
Starting program: /home/sayechu/Escritorio/EECC/P3/1/op_arit_log
[Depuración de hilo usando libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at op_arit_log.s:20
+n                ejecutar siguiente instrucción (xor %eax, %eax)
+n                ejecutar siguiente instrucción (xor %ebx, %ebx)
+n                ejecutar siguiente instrucción (xor %ecx, %ecx)
+n                ejecutar siguiente instrucción (xor %edx, %edx)
+n                ejecutar siguiente instrucción (xor %esi, %esi)
+n                ejecutar siguiente instrucción (xor %edi, %edi)
+p $eax           imprimir contenido registro EAX -> comprobar instr. XOR
$1 = 0
+p $ebx           imprimir contenido registro EBX -> comprobar instr. XOR
$2 = 0
+p $ecx           imprimir contenido registro ECX -> comprobar instr. XOR
$3 = 0
+p $edx           imprimir contenido registro EDX -> comprobar instr. XOR
$4 = 0
+p $esi           imprimir contenido registro ESI -> comprobar instr. XOR
$5 = 0
+p $edi           imprimir contenido registro EDI -> comprobar instr. XOR
$6 = 0
+p $eax           imprimir contenido registro EAX -> antes de instr. mov
$7 = 0
+n                ejecutar siguiente instrucción (mov $5, %eax)
+p $eax           imprimir contenido registro EAX -> comprobar instr. mov
$8 = 5
+p $ebx           imprimir contenido registro EBX -> antes de instr. mov
$9 = 0
+n                ejecutar siguiente instrucción (mov $10, %ebx)
+p $ebx           imprimir contenido registro EBX -> comprobar instr. mov
$10 = 10
+n                ejecutar siguiente instrucción (add %ebx, %eax) -> EAX<-EAX + EBX
+p $eax           imprimir contenido registro EAX -> comprobar instr. add
$11 = 15                  15 <- 5 + 10 ------→ OK
+n                ejecutar siguiente instrucción (mov $5, %eax)
+p $eax           imprimir contenido registro EAX -> comprobar instr. mov
```

```
$12 = 5
+p $ebx              imprimir contenido registro EBX -> antes de instr. mov
$13 = 10
+n                   ejecutar siguiente instrucción (mov $10, %ebx)
+p $ebx              imprimir contenido registro EBX -> comprobar instr. mov
$14 = 10
+n                   ejecutar siguiente instrucción (sub %ebx, %eax) -> EAX<-EAX-EBX
+p $eax              imprimir contenido registro EAX -> comprobar instr. sub
$15 = -5                        -5 <- 5 – 10 ------→ OK
+p $bl                          imprimir contenido registro BL -> antes instr. mov
$16 = 10
+n                   ejecutar siguiente instrucción (movb $-3, %bl)
+p $bl                          imprimir contenido registro BL -> comprobar instr. mov
$17 = -3
+p $al                          imprimir contenido registro AL -> antes instr. mov
$18 = -5
+n                   ejecutar siguiente instrucción (movb $5, %al)
+p $al                          imprimir contenido registro AL -> comprobar instr. mov
$19 = 5
+n                   ejecutar siguiente instrucción (imulb %bl) -> AX <- AL * BL
+p $ax                          imprimir contenido registro AX -> comprobar instr. imul
$20 = -15                               -15 <- -3 * 5 ------→ OK
+p $eflags                      imprimir contenido registro de estado EFLAGS
$21 = [ SF IF ]     como se observa, como el resultado -15 < 0 -> SF = 1
+p $ax                          imprimir contenido registro AX -> antes intr. mov
$22 = -15
+n                   ejecutar siguiente instrucción (movw $5, %ax)
+p $ax                          imprimir contenido registro AX -> comprobar instr. mov
$23 = 5
+p $bl                          imprimir contenido registro BL -> antes instr. mov
$24 = -3
+n                   ejecutar siguiente instrucción (movb $3, %bl)
+p $bl                          imprimir contenido registro BL -> comprobar instr. mov
$25 = 3
+n                   ejecutar siguiente instrucción (idivb %bl) -> AL:Quot. AH:Remain
+p $al                          imprimir contenido registro AL -> cociente de división
$26 = 1  5 (dividendo) / 3 (divisor) = 1 (cociente) * 3 (divisor) + 2 (resto)
+p $ah                          imprimir contenido registro AH -> resto de división
$27 = 2
+p $bl                          imprimir contenido registro BL -> antes instr. negb
```

```
$28 = 3
+p /x $bl
$29 = 0x3                CONSIDERO QUE ES MAS FACIL VER EL COMPLEMENTO A 2 EN
+p /t $bl          BINARIO, POR LO QUE IMPRIMO BL EN BINARIO ANTES DE
$30 = 11           EJECUTAR LA INSTRUCCIÓN E INMEDIATAMENTE DESPUÉS DE
+n                       EJECUTAR LA INSTRUCCIÓN NEGB
+p /t $bl
$31 = 11111101     bl registro 1 byte -> 0000001 | 1 -> 1111110 | 1
+p $bx                   imprimir contenido registro BX -> antes instr. mov
$32 = 253
+n                 ejecutar siguiente instrucción (movw $N, %bx)
+p $bx                   imprimir contenido registro BX -> comprobar instr. mov
$33 = 5
+p $ax                   imprimir contenido registro AX -> antes instr. mov
$34 = 513
+n                 ejecutar siguiente instrucción (movw $(N+1), %ax)
+p $ax                   imprimir contenido registro AX -> comprobar instr. mov
$35 = 6
+n                 ejecutar siguiente instrucción imulw %bx -> DX:AX resultado
+p $ax                   imprimir contenido registro AX -> resultado instr. imulw
$36 = 30
+p $bx                   imprimir contenido registro BX -> antes instr. mov
$37 = 5
+n                 ejecutar siguiente instrucción (movw $2, %bx)
+p $bx                   imprimir contenido registro BX -> comprobar instr. mov
$38 = 2
+n                 ejecutar siguiente instrucción (idivw %bx) -> AL Quot; AH Rema.
+p $al                   imprimir contenido registro AL -> cociente división
$39 = 15                 15 = 30 / 2 ------→ OK
+p $ah                   imprimir contenido registro AH -> resto división
$40 = 0                  30 mod 2 = 0 ------→ OK
+p $eax           imprimir contenido registro EAX -> antes instr. mov
$41 = -65521
+n                 ejecutar siguiente instrucción (mov $0xFFFF1F, %eax)
+p /x $eax               imprimir contenido registro EAX en formato hexadecimal
$42 = 0xffff1f
+p /x $ebx               imprimir contenido registro EBX hex. -> antes instr. mov
$43 = 0x2
+n                 ejecutar siguiente instrucción (mov $0x0000F1, %ebx)
+p /x $ebx               imprimir contenido registro EBX hex -> comprobar inst. mov
```

```
$44 = 0xf1
+n                ejecutar siguiente instrucción (not %eax) -> INVERSIÓN
+p /x $eax                imprimir contenido registro EAX hex -> comprobar inst. not
$45 = 0xff0000e0
+n                ejecutar siguiente instrucción (and %ebx, %eax)
+p /x $eax                imprimir contenido registro EAX hex -> comprobar inst. and
$46 = 0xe0
+n                ejecutar siguiente instrucción (or %ebx, %eax)
+p /x $eax                imprimir contenido registro EAX hex -> comprobar instr. or
$47 = 0xf1
+p /x $ebx                imprimir contenido registro EBX -> antes instr. mov
$48 = 0xf1
+p /x $eax                imprimir contenido registro EAX -> antes instr. mov
$49 = 0xf1
+n                ejecutar siguiente instrucción (mov %ebx, %eax)
+p /x $eax                imprimir contenido registro EAX -> comprobar instr. mov
$50 = 0xf1
+n                ejecutar siguiente instrucción (not %eax)
+p /x $eax                imprimir contenido registro EAX -> comprobar instr. not
$51 = 0xffffff0e
+n                ejecutar siguiente instrucción (inc %eax)
+p /x $eax                imprimir contenido registro EAX -> comprobar instr. inc
$52 = 0xffffff0f
+n                ejecutar siguiente instrucción (shr $4, %eax)
+p /x $eax                imprimir contenido registro EAX -> comprobar instr. shr
$53 = 0xffffff0
+n                ejecutar siguiente instrucción (sar $4, %eax)
+p /x $eax                imprimir contenido registro EAX -> comprobar instr. sar
$54 = 0xffffff
+n                ejecutar siguiente instrucción (mov $SYS_EXIT, %eax)
+p $eax           imprimir contenido registro EAX
$55 = 1
+n                ejecutar siguiente instrucción (mov $SUCCESS, %ebx)
+p $ebx           imprimir contenido registro EBX
$56 = 0
+n        ejecutar siguiente instrucción $0x80 -> llamada S.O con código EAX
[Inferior 1 (process 12919) exited normally]
+quit             fin de depuración
```

Una vez vista la depuración del primer módulo fuente, se procede con la depuración del otro módulo fuente (`saltos.s`). Como ocurre en este caso, se adjunta tanto los comandos de la depuración como los comentarios añadidos en rojo. En dichos comentarios se muestra la instrucción ejecutada. Además, en la depuración del programa, antes de modificar el valor de un registro, se muestra el valor actual y el inmediatamente posterior a ejecutar la instrucción que varía el valor de dicho registro. Se adjunta el `.txt` generado (usando los comandos `set trace-commands on`), a partir de la depuración a continuación.

Cabe destacar que en este módulo se realizan diferentes saltos condicionales. Estos saltos dependen del valor que tengan los banderines en ese momento. Por ello, para ver el valor de los banderines cada vez que se modifican se ha usado la opción `watch $eflags`. Con esto se consigue que al ejecutar una instrucción que altere el valor de los banderines, se imprima el tanto el nuevo valor como el antiguo valor de los banderines, como se puede observar en la depuración.

```
+file saltos          cargar modulo a depurar habiendo compilado cargando tabla simb.
Reading symbols from saltos...
+b main punto de interrupción etiqueta main
Punto de interrupción 1 at 0x118d: file saltos.s, line 20.
+layout regs                cargar pantalla de registros
+focus cmd
Focus set to cmd window.
+run                ejecutar depuración programa cargado (saltos)
Starting program: /home/sayechu/Escritorio/EECC/P3/2/saltos
[Depuración de hilo usando libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".


Breakpoint 1, main () at saltos.s:20
+n        ejecutar siguiente instrucción (xor %eax, %eax)
+n        ejecutar siguiente instrucción (xor %ebx, %ebx)
+n        ejecutar siguiente instrucción (xor %ecx, %ecx)
+n        ejecutar siguiente instrucción (xor %edx, %edx)
+n        ejecutar siguiente instrucción (xor %esi, %esi)
+n        ejecutar siguiente instrucción (xor %edi, %edi)
+p $eax imprimir contenido registro EAX -> comprobar instr xor
$1 = 0
+p $ebx imprimir contenido registro EBX -> comprobar instr xor
$2 = 0
+p $ecx imprimir contenido registro ECX -> comprobar instr xor
$3 = 0
+p $edx imprimir contenido registro EDX -> comprobar instr xor
```

```
$4 = 0
+p $esi   imprimir contenido registro ESI -> comprobar instr xor
$5 = 0
+p $edi   imprimir contenido registro EDI-> comprobar instr xor
$6 = 0
+watch $eflags     cada vez que cambie banderin, mostrar nuevo valor
Watchpoint 2: $eflags
+info watch                    comprobar que esta haciendo watch de registro EFLAGS
Num     Type           Disp Enb Address    What
2       watchpoint     keep y              $eflags
+n          ejecutar siguiente instrucción (jmp salto1)
+n          ejecutar siguiente instrucción (inc %eax)

Watchpoint 2: $eflags

Old value = [ PF ZF IF ]
New value = [ IF ]
main () at saltos.s:40
+p $eax imprimir contenido registro EAX -> comprobar instr. inc
$7 = 1
+p /t $eax
$8 = 1
+n          ejecutar siguiente instrucción (neg %eax)

Watchpoint 2: $eflags

Old value = [ IF ]
New value = [ CF PF AF SF IF ]
main () at saltos.s:41
+p /t $eax            imprimir contenido registro EAX en binario -> comprobar inst neg
$9 = 11111111111111111111111111111111
+p /x $eax
$10 = 0xffffffff
+n          ejecutar siguiente instrucción (shr $1, %eax) -> meter un 0 en MSbit

Watchpoint 2: $eflags

Old value = [ CF PF AF SF IF ]
New value = [ CF PF IF OF ]
main () at saltos.s:51
```

```
+p /x $eax          imprimir contenido registro EAX -> se va el signo por meter 0
$11 = 0x7fffffff
+n          ejecutar siguiente instrucción (shl $1, %eax)

Watchpoint 2: $eflags

Old value = [ CF PF IF OF ]
New value = [ SF IF OF ]
main () at saltos.s:52
+p /x $eax                    imprimir contenido registro EAX -> comprobar instr shl
$12 = 0xfffffffe     LSB = 0xFE = 1111 1110 -> PF = 0
+n          ejecutar siguiente instrucción (clc)
+n          ejecutar siguiente instrucción (xor %eax, %eax) -> ZF = 1; PF = 1;

Watchpoint 2: $eflags

Old value = [ SF IF OF ]
New value = [ PF ZF IF ]
main () at saltos.s:54
+n          ejecutar siguiente instrucción (movw $0xFFFF, %ax) -> no altera mov
+p /x $ax          imprimir contenido registro AX -> comprobar instr. mov
$13 = 0xffff
+n          ejecutar siguiente instrucción (addw $0xFFFF, %ax)

Watchpoint 2: $eflags

Old value = [ PF ZF IF ]
New value = [ CF AF SF IF ]          SE ACTIVA SF Y CF
main () at saltos.s:56
+p /x $ax          imprimir contenido registro AX -> comprobar instrucción add
$14 = 0xfffe
+n          ejecutar siguiente instrucción (clc)

Watchpoint 2: $eflags

Old value = [ CF AF SF IF ]
New value = [ AF SF IF ]   SE DESACTIVA CF POR CLC -> CLEAR CF
main () at saltos.s:57
+n          ejecutar siguiente instrucción (movw $0x7FFF, %ax)
+p /x $ax          imprimir contenido registro AX -> comprobar instrucción mov
```

```
$15 = 0x7fff
+n        ejecutar siguiente instrucción (addw $1, %ax)

Watchpoint 2: $eflags

Old value = [ AF SF IF ]
New value = [ PF AF SF IF OF ]     se active OF, PF = 1 -> LSB = 0x00
main () at saltos.s:65
+p /x $ax          imprimir contenido registro AX -> comprobar instr. add
$16 = 0x8000
+n        ejecutar siguiente instrucción(movw $0xABFF, %ax)->mov no afecta EFLAGS
+n        ejecutar siguiente instrucción(movw $0x0BCF, %bx)->mov no afecta EFLAGS
+p /x $ax          imprimir contenido registro AX -> antes instr y comprobar
$17 = 0xabff
+n        ejecutar siguiente instrucción (test $0x0010, %ax)

Watchpoint 2: $eflags

Old value = [ PF AF SF IF OF ]      0x0010 · 0xABFF > 0;
New value = [ IF ]                  SF=0; LSB=0x10 PF=0;
main () at saltos.s:69
+n        ejecutar siguiente instrucción (test $0xFFFF, %ax)

Watchpoint 2: $eflags

Old value = [ IF ]            AX · 0xABFF = AX < 0
New value = [ PF SF IF ]   SF=1; LSB=0xFF PF=1;
main () at saltos.s:70
+n        ejecutar siguiente instrucción (test $0b0000000000010000, %bx)

Watchpoint 2: $eflags

Old value = [ PF SF IF ]
New value = [ PF ZF IF ]
main () at saltos.s:77
+n        ejecutar siguiente instrucción (movw $0x01FF, %ax)
+n        ejecutar siguiente instrucción (movw $0x0001, %bx)
+n        ejecutar siguiente instrucción (movw $0x00FF, %cx)
+p /x $ax          imprimir contenido registro AX -> comprobar instr. mov
$18 = 0x1ff
```

```
+p /x $bx          imprimir contenido registro BX -> comprobar instr. mov
$19 = 0x1
+p /x $cx          imprimir contenido registro CX -> comprobar instr. mov
$20 = 0xff
+n        ejecutar siguiente instrucción (cmp $0x00FF, %ax)

Watchpoint 2: $eflags

Old value = [ PF ZF IF ]    0x01FF – 0x00FF = 0x0100 > 0
New value = [ PF IF ]       ZF = 0; SF = 0; LSB = 0x00; PF = 1
main () at saltos.s:81
+n        ejecutar siguiente instrucción (cmp $0x00FF, %bx)

Watchpoint 2: $eflags

Old value = [ PF IF ]                0x0BCF – 0x00FF = 0xFF02 < 0
New value = [ CF AF SF IF ]          SF = 1; LSB=0x02 PF = 0; CF=1
main () at saltos.s:83
+n        ejecutar siguiente instrucción (cmp $0x00FF, %cx)

Watchpoint 2: $eflags

Old value = [ CF AF SF IF ]          0x00FF – 0x00FF = 0
New value = [ PF ZF IF ]             ZF = 1; SF = 0; LSB = 0xFF PF=1
main () at saltos.s:87
+n        ejecutar siguiente instrucción (movw $0x01FF, %ax)
+n        ejecutar siguiente instrucción (movw $0x0001, %bx)
+n        ejecutar siguiente instrucción (movw $0x00FF, %cx)
+p /x $ax          imprimir contenido registro AX -> comprobar instr. mov
$21 = 0x1ff
+p /x $bx          imprimir contenido registro BX -> comprobar instr. mov
$22 = 0x1
+p /x $cx          imprimir contenido registro CX -> comprobar instr. mov
$23 = 0xff
+n        ejecutar siguiente instrucción (cmp $0x00FF, %ax)

Watchpoint 2: $eflags

Old value = [ PF ZF IF ]    0x01FF-0x00FF = 0x0100 > 0;
New value = [ PF IF ]       ZF = 0; SF = 0; LSB = 0x00 PF = 1
```

```
main () at saltos.s:91
+n        ejecutar siguiente instrucción (jg salto4) -> SF = 0 -> SALTA
salto4 () at saltos.s:93
+n        ejecutar siguiente instrucción (cmp $0x00FF, %bx)

Watchpoint 2: $eflags

Old value = [ PF IF ]                0x0001 – 0x00FF = 0xFF02 < 0
New value = [ CF AF SF IF ]          SF=1; ZF=0; CF=1;
salto4 () at saltos.s:95
+n        ejecutar siguiente instrucción (jl salto5)
salto5 () at saltos.s:97
+n        ejecutar siguiente instrucción (movw $0x8000, %ax)
+n        ejecutar siguiente instrucción (cmp $0x0001, %ax)

Watchpoint 2: $eflags

Old value = [ CF AF SF IF ]          0x8000 - 0x0001 = 0x7FFFF > 0
New value = [ PF AF IF OF ]          ZF=0; SF=0; OF=1; CF=0
salto5 () at saltos.s:103
+n        ejecutar siguiente instrucción (ja salto6) -> SALTA
salto6 () at saltos.s:105
+n        ejecutar siguiente instrucción (cmp $0x00FF, %cx)

Watchpoint 2: $eflags

Old value = [ PF AF IF OF ]          0x00FF-0x00FF = 0
New value = [ PF ZF IF ]             ZF=1; SF=0; PF=1;
salto6 () at saltos.s:106
+n        ejecutar siguiente instrucción (je salto7) -> ZF = 1 -> SALTA
salto7 () at saltos.s:112
+n        ejecutar siguiente instrucción (mov $SYS_EXIT, %eax)
+n        ejecutar siguiente instrucción (mov $SUCCESS, %ebx)
+n        ejecutar siguiente instrucción (int $0x80)
[Inferior 1 (process 13902) exited normally]
+quit     salir de depuración de programa
```

Conclusiones

En esta práctica se empieza viendo algunas de las instrucciones para realizar diferentes operaciones. Entre las cuales cabe destacar las operaciones lógicas (como la negación, desplazamiento de bits), complemento a 2 y algunas operaciones aritméticas (suma, resta, multiplicación y división). Tras haber visto el registro donde se guardan cada uno de los resultados de dichas operaciones y cómo funcionan, se procede a la depuración de dicho programa en la se observa el valor de los resultados de dichas operaciones. Cabe destacar, que en la parte de Desarrollo en cada una de las instrucciones se ha adjuntado una tabla de la documentación de dicha instrucción para saber qué registros son necesarios usar dependiendo de los bytes de los operandos.

Tras haber visto las bases de algunas operaciones lógicas, como el desplazamiento de bits, tanto lógico como aritmético, se procede con el segundo módulo fuente (saltos.s). En este se ven como, dependiendo del valor de los banderines, se producen algunos saltos condicionales, o no, dependiendo del valor de los mismos. Se introducen las instrucciones de `test` y `cmp` que, como ya se ha visto a lo largo de la práctica, estas instrucciones no modifican el valor de ningún registro fuente o destino, si no que solo sirven para modificar el valor del registro de estado EFLAGS.
Un ejemplo de un salto condicional puede ser, teniendo 2 valores idénticos en 2 registros distintos, si realizamos un compare (cmp) entre ambos registros, como son dos valores idénticos, al realizar la resta de ellos da cero. Esto activa el banderín 'Zero Flag'. Si ejecutamos una instrucción de salto en caso de que los valores sean iguales (JumpEqual), ésta instrucción se fija en el valor del banderín Zero Flag. En este caso como está activado (ZF = 1), haría el salto a la etiqueta o el punto indicado junto a la instrucción jump.

Tras haber visto el programa y probado a realizar diferentes saltos condicionales, se procede a la depuración que, gracias al comando watch, nos permite ver el valor de los banderines cada vez que uno de estos cambia al ejecutar una instrucción.