GLOSARIO de las abreviaturas

codop = código de operación.
MI = memoria de instrucciones = la parte de la memoria donde están las instrucciones.
MD = memoria de datos = la parte de la memoria donde están los datos.

> [Aclaración: MI y MD no son dos memorias distintas. Son distintos sectores de la misma memoria. Cuando decimos "memoria" nos referimos básicamente a la RAM. Los registros también son memoria (en el sentido de que sirven para almacenar información), pero están dentro de la CPU. A ellos no los llamamos "memorias", sino "registros"].

MDD = modo de direccionamiento.
inst = instrucción/instrucciones.
acc = acceso/s.
B = Byte/s = 8 bits.
Ej. = ejemplo.

1) **Dada la siguiente definición de datos y el código:** $F = [(A + B)/C] - D$

Nombre	Tamaño	Valor
A	1 Byte	6
B	1 Byte	4
C	1 Byte	2
D	1 Byte	1
F	1 Byte	?

Suponiendo que se poseen las instrucciones necesarias en cada caso, escribir el programa que implemente el código anterior utilizando máquinas de 1, 2 o 3 direcciones.

Máquina de 1 dirección	Máquina de 2 direcciones	Máquina de 3 direcciones

load A
add B
div C
sub D
store F
	mov F, A
	add F, B
	div F, C
	sub F, D
		add F, A, B
		div F, F, C
		sub F, F, D

2) **Suponga que cada código de operación ocupa 6 bits y las direcciones son de 10 bits. Analice las soluciones implementadas en el ejercicio anterior y complete la siguiente tabla:**

	Máquina de 1 dirección	Máquina de 2 direcciones	Máquina de 3 direcciones
Tamaño del programa en memoria (codop + operandos)	Ej.: load (1B) + A (2B) = 3B 5 inst x 3B = **15 Bytes**	Ej.: mov (1B) + F (2B) + A (2B) = 5B 4 inst x 5B = **20 Bytes**	Ej.: add (1B) + F (2B) + A (2B) + B (2B) = 7B 3 inst x 7B = **21 Bytes**
Cantidad de accesos a memoria (instrucciones + operandos)	MI = 3 acc MD = 1 acc 3 + 1 = 4 acc x inst 5 inst x 4 acc = **20 accesos**	MI = 5 acc MD (1° inst) = 2 acc MD = 3 acc 1° inst = 7 acc 2°, 3° y 4° = 8 acc 7 acc + (3 inst x 8 acc) = **31 accesos**	MI = 7 acc MD = 3 acc 7 + 3 = 10 acc x inst 3 inst x 10 acc = **30 accesos**

¿Cómo se calcula el tamaño del programa?

Creo que lo que está implícito acá, es que *calcular la memoria del programa* significa *calcular la memoria de instrucciones del programa principal* (que en el VonSim suele ser lo que está desde la dirección 2000h), ya que no estamos contando por ejemplo lo que ocupan las variables u operandos almacenados en el sector de las variables (que en el VonSim suele ser lo que está desde la dirección 1000h).

Por ello, el tamaño del programa se calcula sencillamente sumando las instrucciones. Ahora bien, ¿cómo hacemos eso? Cada instrucción se calcula sumando el codop más la dirección al operando. En este caso, como el codop es de 6 bits, ocupará 8 bits, ya que debe ser múltiplo de 1 Byte para poder almacenarse; y las direcciones (de 10 bits) ocuparán 2 Bytes.

¿Cómo se calcula la cantidad de accesos a memoria?

Acá se asume que cada acceso a memoria sirve para leer 1 Byte. Por eso, en el caso de la máquina de 1 dirección, la línea *load A* requiere 1 lectura para la instrucción, 2 para la dirección de A y 1 para el operando A. Es decir, se necesitan 3 accesos a la MI y 1 a la MD.

Esto significa que la línea de código *load A* requiere 4 accesos a memoria. Y para obtener el valor final, se multiplica eso por cada una de las instrucciones.

Ahora bien, en el caso de la máquina de 2 direcciones sucede algo especial. La línea *mov F, A* requiere 1 acceso para *mov*, 2 para la dirección de F, 2 para la dirección de A, 1 para obtener el operando A, y otro para depositar A en F. Es decir que en total se usaron 5 accesos a MI y 2 MD. O sea, 7 accesos a memoria. Pero el resto de las instrucciones, son distintas. La línea *add F, B* por ejemplo, necesita 1 para add, 2 para la dirección de F, 2 para la dirección de B (hasta ahora igual que antes, 5 a MI), pero después necesita 1 acceso para obtener F, otro para obtener B, y otro más para depositar el resultado en F. Por eso hay 3 accesos a MD. Así, las instruciones comunes y corrientes, en esta máquina, requieren 5 accesos a MI y 3 a MD, o sea, 8 accesos a memoria. En resumen: la primera instrucción inicializa F con un valor mientras que las demás instrucciones modifican F usando su valor anterior. Por eso *mov* requiere 2 accesos a MD, mientras que *add*, *div*, *sub* requieren 3 accesos a MD cada una.

3) **Dado el siguiente código:** $F = ((A - B) \cdot C) + (\frac{D}{E})$
   a) **Implemente el código utilizando máquinas de 1, 2 y 3 direcciones.**
   b) **Realice una tabla de comparación similar a la del ejercicio 2.**
   c) **¿Cuál máquina elegiría haciendo un balance de la cantidad de instrucciones, el espacio en memoria ocupado y el tiempo de ejecución (1 acceso a memoria = 1 ms)? ¿Es ésta una conclusión general?**

3) a)

Máquina de 1 dirección	Máquina de 2 direcciones	Máquina de 3 direcciones

load A
sub B
mult C
store F
load D
div E
add F
store F
	mov F, A
	sub F, B
	mult F, C
	div D, E
	add F, D
		sub F, A, B
		mult F, F, C
		div D, D, E
		add F, F, D

3) b)

	Máquina de 1 dirección	Máquina de 2 direcciones	Máquina de 3 direcciones
Tamaño del programa en memoria (codop + operandos)	Ej.: load (1B) + A (2B) = 3B 8 inst x 3B = **24 Bytes**	Ej.: mov (1B) + F (2B) + A (2B) = 5B 5 inst x 5B = **25 Bytes**	Ej.: sub (1B) + F (2B) + A (2B) + B (2B) = 7B 4 inst x 7B = **28 Bytes**
Cantidad de accesos a memoria (instrucciones + operandos)	MI = 3 acc MD = 1 acc 3 + 1 = 4 acc x inst 8 inst x 4 acc = **32 accesos**	MI = 5 acc MD (1° inst) = 2 acc MD = 3 acc 1° inst = 7 acc el resto = 8 acc 7 acc + (4 inst x 8 acc) = **39 accesos**	MI = 7 acc MD = 3 acc 7 + 3 = 10 acc x inst 4 inst x 10 acc = **40 accesos**

3) c) La de una instrucción parece ser la mejor en programas de este tipo.

Antes de adentrarnos en el simulador VonSim, repasemos cómo está constituido:

Existen 4 registros de propósito general: AX, BX, CX, DX.

Cada registro tiene 2 Bytes.

Podemos usarlo entero y ocupar los 2 Bytes, o podemos usar alguna de las dos mitades, y usar 1 Byte.

La parte de la izquierda es la parte alta, y la de la derecha la parte baja.

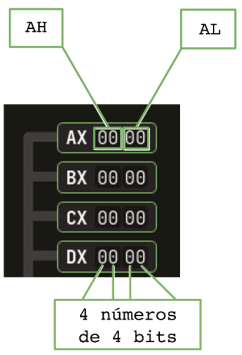
Por eso, si queremos llamar a alguna de las mitades, usamos las letras H y L, por *high* y *low*.

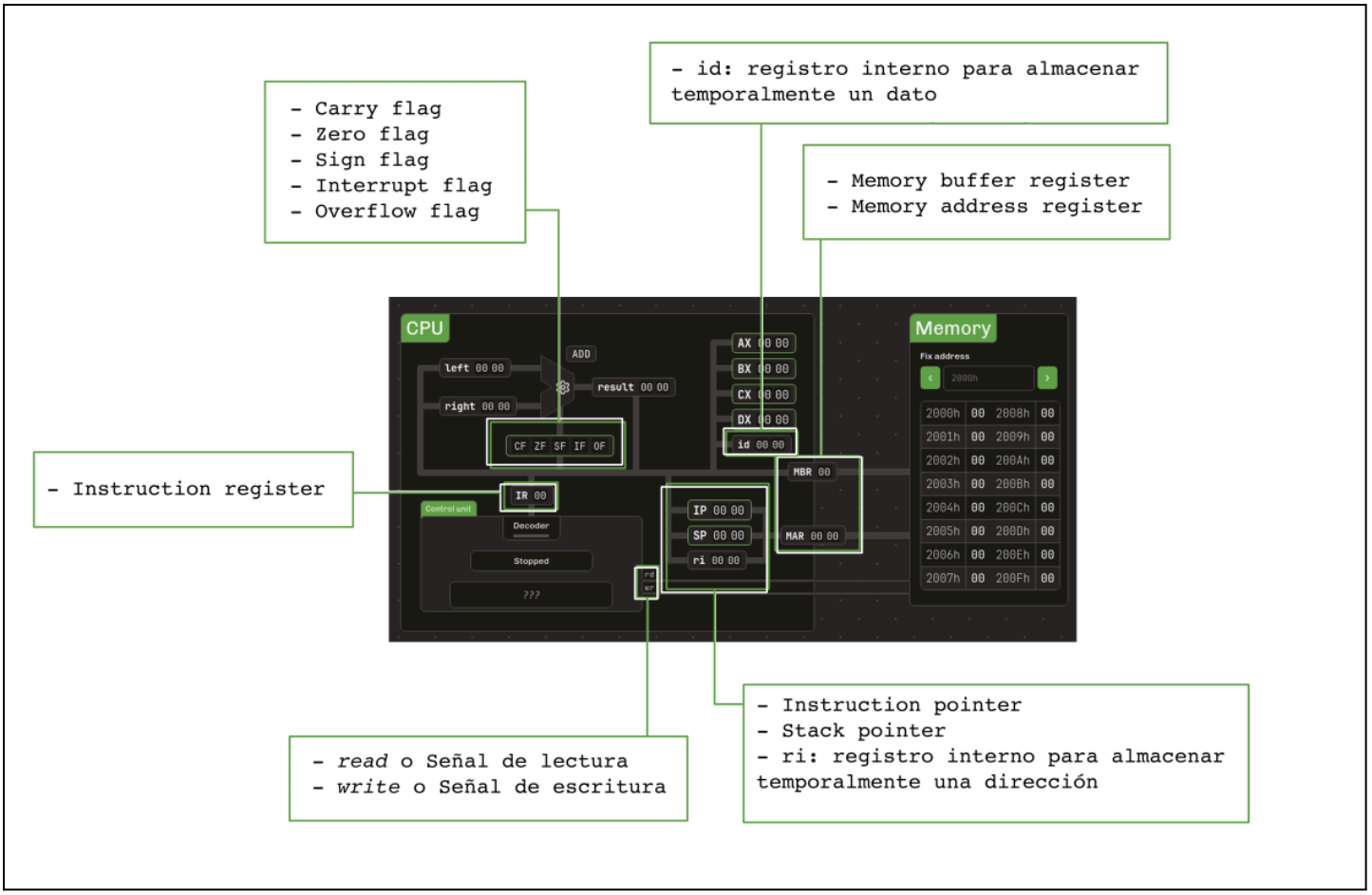
Ejemplos: AX completo es AX; la parte alta de AX sería AH; la parte baja de CX sería CL, etc.

Cada uno de los números que vemos ahí,
es un número en hexadecimal.

Por lo tanto, cada número de esos
ocupa exactamente 4 bits.

Por eso cada registro ocupa 2 Bytes, y
2 Bytes = 16 bits = 4 números hexadecimales.





4) El siguiente programa utiliza una instrucción de transferencia de datos (instrucción MOV) con diferentes modos de direccionamiento para referenciar sus operandos. Ejecutar y analizar el funcionamiento de cada instrucción en el simulador observando el flujo de información a través del BUS DE DATOS, el BUS DE DIRECCIONES, el BUS DE CONTROL, el contenido de REGISTROS, de posiciones de MEMORIA, operaciones en la ALU, etc.

```
        ORG 1000h
NUM0 DB 0CAh      ; está en la dirección 1000h
NUM1 DB 0         ; está en la dirección 1001h
NUM2 DW ?         ; está en las direcciones 1002h y 1003h
NUM3 DW 0ABCDh ; está en las direcciones 1004h y 1005h
NUM4 DW ?         ; está en las direcciones 1006h y 1007h

        ORG 2000H
MOV BL, NUM0
MOV BH, 0FFh
```

```
MOV CH, BL
MOV AX, BX
MOV NUM1, AL
MOV NUM2, 1234h
MOV BX, OFFSET NUM3
MOV DL, [BX]
MOV AX, [BX]
MOV BX, 1006h
MOV WORD PTR [BX], 0CDEFh
HLT
END
```

a) Explicar detalladamente qué hace cada instrucción MOV del programa anterior, en función de sus operandos y su modo de direccionamiento. b) Confeccionar una tabla que contenga todas las instrucciones MOV anteriores, el modo de direccionamiento y el contenido final del operando destino de cada una de ellas.

El modo de direccionamiento es un asunto delicado. No está claro a qué MDD pertenece cada una de las instrucciones que vemos acá, ya que a veces hay más de un MDD dentro de la misma instrucción. Por lo general tiene más peso el MDD del operando de la derecha a la hora de definir el MDD de la instrucción, pero no siempre es así. Y, en realidad, para ser exactos, deberíamos hablar de MDD no por instrucción, sino por operando. Esto no lo vamos a hacer ahora, pero probablemente lo tengamos que hacer así más adelante.

Para orientarnos un poco, estos son los casos básicos:

MMD	Inmediato	Directo	Directo por registro	Indirecto por registro
Ejemplo(s)	*mov AL, 9* o *mov BX, offset num_1*	*mov AL, num_1*	*mov AL, AH*	*mov BX, 1000H* *mov AL, [BX]*
Explicación	Se utiliza un valor fijo. No se requiere ningún acceso a memoria para obtener dicho valor.	Se requiere un acceso extra a memoria para tomar el valor de la variable.	El valor del operando se encuentra en el registro indicado. No requiere acceso a memoria.	El registro BX puede servir como puntero. Requiere un acceso extra a memoria.

[Los que están en amarillo son los casos dudosos].

MOV BL, NUM0 → Usando el MDD directo, copia el contenido de NUM0 que está en memoria (en la posición $1000_{(16)}$), y lo pega en el registro BL. Ahora BL contiene el valor $CA_{(16)}$. En otras palabras, se le asigna al registro BL el valor de la variable NUM0.

MOV BH, 0FFh → Usando el MDD inmediato, le asigna al registro BH el valor $FF_{(16)}$.

MOV CH, BL → Usando el MDD directo por registro, le asigna al registro CH el valor que tiene BL. Ahora CH queda cargado con el valor $CA_{(16)}$.

MOV AX, BX → Usando el MDD directo por registro, le asigna al registro AX el valor que tiene BX. Ahora AX queda cargado con el valor $FFCA_{(16)}$.

MOV NUM1, AL → Usando el MDD directo, almacena el valor de AL en la variable NUM1 (que está en la posición $1001_{(16)}$ de la memoria). Ahora NUM1 vale $CA_{(16)}$. Esto significa que ahora la posición $1001_{(16)}$ de la memoria contiene el valor $CA_{(16)}$.

MOV NUM2, 1234h → Usando el MDD inmediato, se le asigna a la variable NUM2 (que está en las posiciones $1002_{(16)}$ y $1003_{(16)}$ de la memoria) el valor $1234_{(16)}$. Más especificamente, $1002_{(16)}$ queda con el valor $34_{(16)}$ y $1003_{(16)}$ con el valor $12_{(16)}$.

MOV BX, OFFSET NUM3 →Usando el MDD inmediato, se carga en el registro BX el valor que de la dirección de la variable o etiqueta NUM3. O sea, se copia en BX el valor $1004_{(16)}$.

MOV DL, [BX] →Usando el MDD indirecto por registro, se carga en el registro DL (de 1 byte) el valor que está almacenado en la dirección de memoria guardado en BX. Es decir, se carga en DL lo apuntado por BX. Y como BX tiene el valor $1004_{(16)}$, se carga en DL el valor $CD_{(16)}$.

MOV AX, [BX] →Usando el MDD indirecto por registro, se carga en el registro AX (de 2 bytes) el valor que está almacenado en la dirección de memoria guardado en BX. Por ello, se carga en AX el valor completo de NUM3, o sea, $ABCD_{(16)}$.

MOV BX, 1006h →Usando el MDD inmediato, se le asigna al registro BX el valor $1006_{(16)}$.

MOV WORD PTR [BX], 0CDEFh → Usando el MDD indirecto por registro, se almacena en lo apuntado por BX el valor $CDEF_{(16)}$. Es decir, se almacena en las posiciones $1006_{(16)}$ y $1007_{(16)}$ de la memoria el valor $CDEF_{(16)}$. En $1006_{(16)}$ queda $EF_{(16)}$, y en $1007_{(16)}$ queda $CD_{(16)}$.

c) Notar que durante la ejecución de algunas instrucciones MOV aparece en la pantalla del simulador un registro temporal denominado “ri”, en ocasiones acompañado por otro registro temporal denominado “id”. Explicar con detalle qué función cumplen estos registros.

Podríamos decir que son registros temporales auxiliares o buffers. Se utilizan mucho al momento de realizar intercambios de datos entre el CPU y la memoria.

El “id” sirve para almacenar temporalmente un dato. Generalmente recibe algún dato desde el MBR, y también puede enviarle algún dato al MBR o a alguno de los cuatro registros de propósito general.

El “ir” sirve para almacenar temporalmente una dirección (que suele provenir desde alguno de los cuatro registros de propósito general o desde el MBR), la cual se va a cargar posteriormente en el MAR.

5) El siguiente programa utiliza diferentes instrucciones de procesamiento de datos (instrucciones aritméticas y lógicas). Analice el comportamiento de ellas y ejecute el programa en el MSX88.

A menos que se especifique otra cosa, los valores en decimal (en nuestra explicación) provienen de la interpretación en Ca2.

```
        ORG 1000h
NUM0 DB     80h          ; está en la dirección 1000h con el valor -128 o 80(16)
NUM1 DB     200          ; está en 1001h con el valor C8(16) que en BSS es 200
NUM2 DB     -1           ; está en 1002h con el valor -1 o FF(16)
BYTE0 DB    01111111B    ; está en 1003h con el valor 127 o 7F(16)
BYTE1 DB    10101010B    ; está en 1004h con el valor -86 o AA(16)

        ORG 2000H
MOV AL, NUM0 ; copia NUM0 en AL. Ahora AL = 80(16) = -128
ADD AL, AL ; realiza la operación -128 + (-128) y la almacena en AL. Ahora AL = 0
INC NUM1 ; le suma 1 a NUM1. Ahora NUM1 = 201(BSS) = C9(16)
MOV BH, NUM1 ; copia NUM1 en BH. Ahora BH = 201(BSS) = C9(16)
MOV BL, BH ; copia BH en BL. Ahora BL = C9(16)
DEC BL ; le resta 1 a BL. Ahora BL = C8(16)
SUB BL, BH ; realiza la operación BL - BH, es decir, 200(BSS) - 201(BSS), y guarda el resultado
en BL. Ahora BL = -1 = FF(16)
MOV CH, BYTE1 ; copia 10101010 en CH. Ahora CH = AA(16)
AND CH, BYTE0 ; realiza la operación 10101010 AND 01111111 y almacena el resultado en
CH. Ahora CH contiene la cadena 00101010 = 2A(16)
NOT BYTE0 ; niega el BYTE0, entonces BYTE0 queda en 10000000 = 80(16)
OR CH, BYTE0 ; realiza la operación 00101010 OR 10000000 y almacena el resultado en
CH. Ahora CH contiene la cadena 10101010 = AA(16)
XOR CH, 11111111B ; realiza la operación 10101010 XOR 11111111 y almacena el resultado
en CH. Ahora CH contiene la cadena 01010101 = 55(16)
HLT
END
```

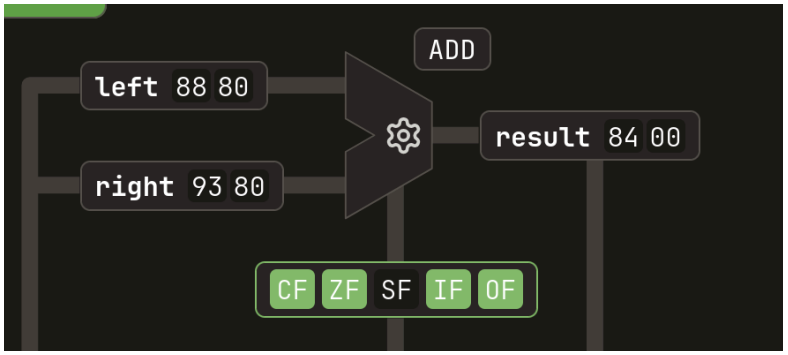
5. 1) ¿Cuál es el estado de los FLAGS después de la ejecución de las instrucciones ADD y SUB del programa anterior? Justificar el estado (1 o 0) de cada uno de ellos. ¿Dan alguna indicación acerca de la correctitud de los resultados?

La instrucción *ADD AL, AL* realiza la siguiente suma:

```
    10000000
ADD 10000000
   [1]00000000      → el bit en negrita es de carry, no entra en el resultado de la máquina
```

Esta operación, interpretada en Ca2, sería así: (-128) + (-128) y debería dar -256, pero está fuera de rango, y el resultado que efectivamente obtenemos es 0.

Por eso los FLAGS quedan así: C = 1, Z = 1, S = 0, O = 1. Por ello, el resultado es incorrecto tanto desde la interpretación en BSS como desde la interpretación en Ca2.



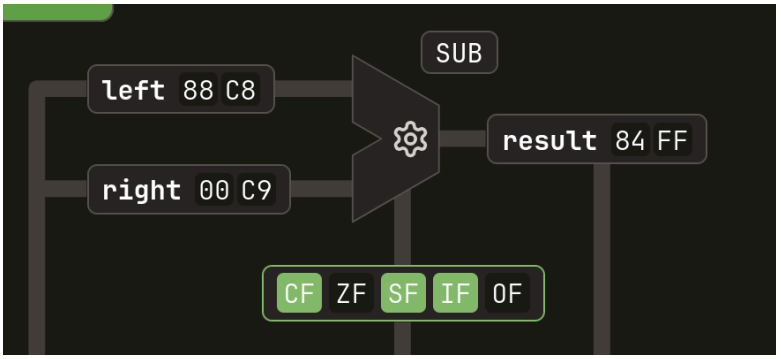
La instrucción *SUB BL, BH* realiza la siguiente resta:

```
   [1]11001000 → el bit en negrita es de borrow, no entra en el resultado de la máquina
SUB 11001001
    11111111
```

Esta operación, interpretada en Ca2, sería así: (-56) - (-55), lo cual debería dar -1, y efectivamente da -1.

Interpretarla en BSS es obviamente problemático, ya que el minuendo (200) es menor al sustraendo (201), con lo cual el resultado debería ser negativo, pero no hay números negativos en BSS.

Por eso los FLAGS quedan así: C = 1, Z = 0, S = 1, O = 0. Por ello, el resultado es incorrecto desde la interpretación en BSS, pero es correcto desde la interpretación en Ca2.



6) El siguiente programa implementa un contador utilizando una instrucción de transferencia de control. Analice el funcionamiento de cada instrucción y en particular las del lazo repetitivo que provoca la cuenta.

```
        ORG 1000H
INI DB 0 ; está en la dirección 1000h con el valor 0
FIN DB 15 ; está en la dirección 1001h con el valor 15

        ORG 2000H
MOV AL, INI ; AL := 0
MOV AH, FIN ; AH := 15
SUMA: INC AL ; AL := AL + 1
        CMP AL, AH ; AL - AH →esto actualiza los flags
        JNZ SUMA ; if (no da cero) then, go back to SUMA
HLT
END
```

a) ¿Cuántas veces se ejecuta el lazo? ¿De qué variables depende esto en el caso general?

Se ejecuta 15 veces. 1 porque viene de la secuencia, y 14 porque vuelve desde el JNZ.

En el caso general, depende tanto de la variable que se usa como contador, como de la variable que se usa para ponerle el límite al bucle.

b) Analice y ejecute el programa reemplazando la instrucción de salto condicional JNZ por las siguientes, indicando en cada caso el contenido final del registro AL:

1°) JS → aquí el programa salta cuando el resultado de la comparación es un número negativo. En este caso, la primera comparación es 1 - 15 = -14, así que salta; la segunda es 2 - 15 = -13, así que también salta. Y así siguiendo hata que llegue a 15 - 15 = 0, que no es negativo, y por lo tanto no salta. Ergo, la cantidad de veces que se ejecuta el lazo es la misma que antes.

2°) JZ → aquí el programa salta cuando el resultado de la comparación es el 0. Por ello, en el presente caso, si usamos el JZ nunca llega a saltar.

3°) JMP → aquí salta siempre sin importar la condición, así que a menos que hagamos algo para cortarlo, estaremos ante un bucle infinito.

7) Escribir un programa en lenguaje assembly del MSX88 que implemente la sentencia condicional de un lenguaje de alto nivel

IF A < B THEN
    C = A
ELSE
    C = B

Considerar que las variables de la sentencia están almacenadas en los registros internos de la CPU del siguiente modo A en AL, B en BL y C en CL.

```
; Versión 1 de 3
        ORG 2000H
CMP AL, BL ; si AL < BL, entonces la operación AL - BL debe dar negativo y queda S = 1
JNS ELSE ; si S = 0, entonces no es cierto que AL < BL, así que salta a la etiqueta ELSE
        MOV CL, AL
        JMP END_IF ; si llegó a esta línea, es porque AL < BL, entonces no debe ejecutar la siguiente, por eso salta a la etiqueta END_IF
ELSE: MOV CL, BL
END_IF:
HLT
END
```

Determine las modificaciones que debería hacer al programa si la condición de la sentencia IF fuera A <= B y A = B

```
; Versión 2 de 3
        ORG 2000H
CMP BL, AL ; si AL <= BL, entonces BL - AL debe dar no negativo y queda S = 0
JS ELSE ; si S = 1, entonces no es cierto que AL <= BL, así que salta a la etiqueta ELSE
        MOV CL, AL
        JMP END_IF
ELSE: MOV CL, BL
END_IF:
HLT
END
```

```
; Versión 3 de 3
        ORG 2000H
CMP BL, AL ; si AL = BL, entonces BL - AL debe dar 0 y queda Z = 1
JNZ ELSE ; si Z = 0, entonces no es cierto que AL = BL, así que salta a la etiqueta ELSE
        MOV CL, AL
        JMP END_IF
ELSE: MOV CL, BL
END_IF:
HLT
END
```

8) El siguiente programa suma todos los elementos de una tabla almacenada a partir de la dirección 1000H de la memoria del simulador. Analice el funcionamiento y determine el resultado de la suma. Comprobar resultado en el MSX88.

```
        ORG 1000H
TABLA DB 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 ; array de 10 números que están almacenados desde la celda 1000(16) hasta la 1009(16)
FIN DB ? ; está en 100A(16)
TOTAL DB ? ; está en 100B(16)
MAX DB 13 ; está en 100C(16)

        ORG 2000H
MOV AL, 0 ; acá se irán acumulando las sumas, por eso lo inicializa en cero
MOV CL, OFFSET FIN - OFFSET TABLA ; esto sirve para tomar la medida del array, hace la operación 100A(16) - 1000(16), y pone el resultado A(16) en CL. Recordemos que A(16) = 10(10). Esto servirá como límite del contador
MOV BX, OFFSET TABLA ; BX := 1000(16)
SUMA: ADD AL, [BX] ; le suma a AL lo apuntado por BX. En la primera iteración será 2, después 4, y así hasta terminar el array
        INC BX ; incrementa BX para pasar al siguiente elemento del array
        DEC CL ; decrementa CL para llevar la cuenta de la cantidad de iteraciones
JNZ SUMA ; cuando CL sea 0, el bucle se habrá ejecutado 10 veces, y termina
HLT
END
```

Los registros quedaron así:

AL = $6E_{(16)} = 110_{(10)}$, lo cual es correcto: esa es la suma del array

BX = $100A_{(16)}$, lo cual es correcto: significa que la última dirección procesada fue $1009_{(16)}$

CL = $0_{(16)}$, corroboramos que el contador pasó de 10 a 0

¿Qué modificaciones deberá hacer en el programa para que el mismo almacene el resultado de la suma en la celda etiquetada TOTAL?

Una vez que termina el bucle, hay que poner: MOV TOTAL, AL

9) Escribir un programa que determine cuántos de los elementos de TABLA son menores o iguales que MAX. Dicha cantidad debe almacenarse en la celda TOTAL.

```
        ORG 1000H
TABLA DB 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 ; están desde 1000(16) hasta 1009(16)
FIN DB ? ; está en 100A(16)
TOTAL DB ? ; está en 100B(16)
MAX DB 13 ; está en 100C(16)

        ORG 2000H
MOV AL, 0 ; acá se irán acumulando las sumas, por eso lo inicializa en cero
MOV AH, 0 ; este es el contador de elementos <= MAX, por eso lo inicializa en cero
MOV CL, OFFSET FIN - OFFSET TABLA ; esto sirve para tomar la medida del array
MOV BX, OFFSET TABLA ; BX := 1000(16)
MOV DH, MAX ; DH := 13(10)
SUMA: MOV DL, [BX] ; DL pasa a contener algún elemento del array
        ADD AL, DL ; le suma a AL algún elemento del array
        CMP DH, DL ; si DL <= DH, entonces S = 0
        JS NoContar ; si S = 1, es falso que DL <= DH, entonces no lo cuenta y salta
                INC AH
        NoContar: INC BX ; incrementa BX para pasar al siguiente elemento del array
        DEC CL ; decrementa CL para llevar la cuenta de la cantidad de iteraciones
JNZ SUMA ; cuando CL sea 0, el bucle se habrá ejecutado 10 veces, y termina
MOV TOTAL, AH
HLT
END
```

10) Analizar el funcionamiento del siguiente programa.

```
        ORG 2000H
MOV AX, 1 ; AX := 0001h
MOV BX, 1000h ; BX := 1000h
CARGA: MOV [BX], AX ; se copia en lo apuntado por BX, lo que haya en AX (notemos que, como AX son dos Bytes, se ocuparán dos Bytes en la memoria, y por eso la siguiente instrucción incrementa BX con 2 en vez de con 1)
        ADD BX, 2
        ADD AX, AX ; AX := AX + AX o sea que AX := AX*2
        CMP AX, 200 ; se efectúa la operación AX - 200
        JS CARGA ; salta a la etiqueta CARGA siempre que AX < 200
HLT
END
```

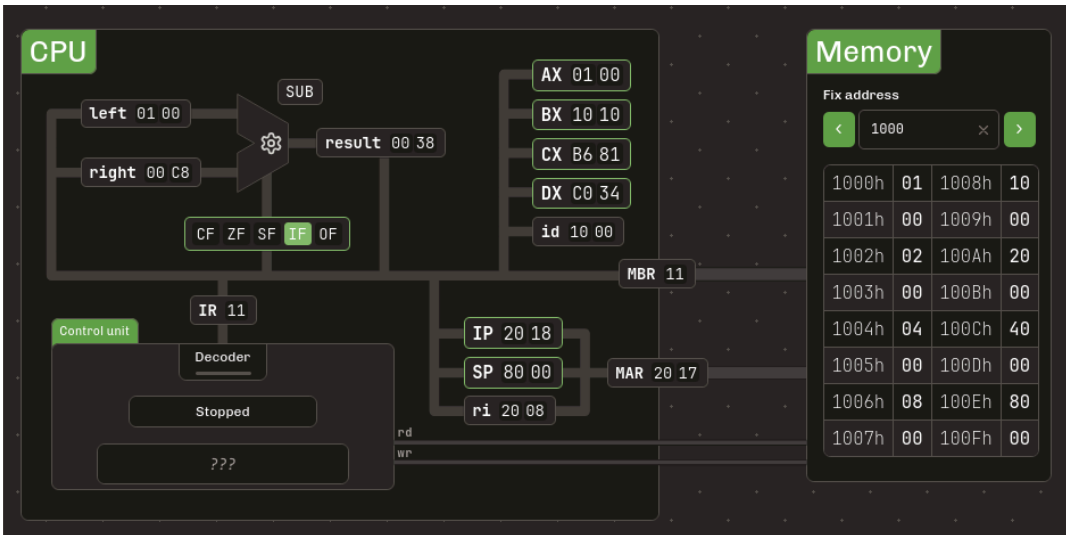
Respuestas al cuestionario:

El programa genera una tabla o array con las potencias de 2, comenzando desde el número 1 y terminando con 128. Las va almacenando de a dos bytes empezando la posición 1000h, luego la 1002h, la 1004h, etc. Termina porque el bucle puso como límite que el número que se va agrandando (que se va sumando a sí mismo) sea menor a 200. Después del 128 sigue el 256, por eso el último que se "procesa" es el 128.

La tabla (con el contenido de la memoria expresada en decimal) queda así:

1000h	1002h	1004h	1006h	1008h	100Ah	100Ch	100Eh
1	2	4	8	16	32	64	128

En el simulador (que muestra los bits en hexadecimal) queda así:

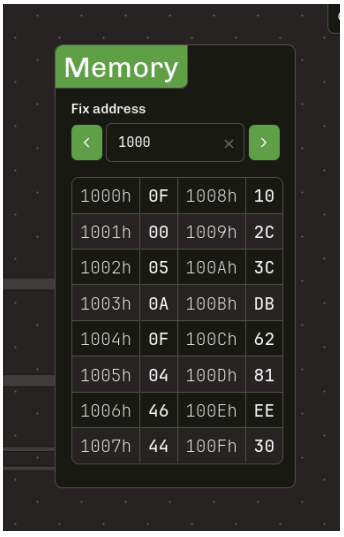


11) Escribir un programa que genere una tabla a partir de la dirección de memoria almacenada en la celda DIR con los múltiplos de 5 desde cero hasta MAX.

Interpretamos que esto significa que el programa debe almacenar los múltiplos mientras sean menores o iguales a MAX.

```
        ORG 1000h
max db 15 ; acá elegimos un límite
tabla db ?

        ORG 2000h
MOV BX, OFFSET tabla
MOV AL, max
MOV AH, 0 ; en AH vamos almacenando los múltiplos
REPETIR: MOV [BX], AH
        INC BX
        ADD AH, 5
        CMP AL, AH
        JNS REPETIR ; deja de saltar cuando el múltiplo en AH sea > max
HLT
END
```

Este programa genera un array con los números 0, 5, 10 y 15, almacenados desde 1001h hasta 1004h. En la memoria, quedan así:



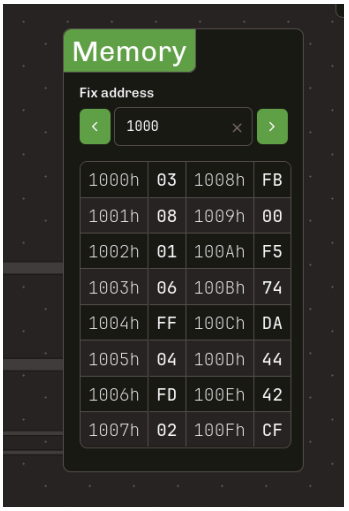
12) Escribir un programa que, dado un número X, genere un arreglo con todos los resultados que se obtienen hasta llegar a 0, aplicando la siguiente fórmula: si X es par, se le resta 7; si es impar, se le suma 5, y al resultado se le aplica nuevamente la misma fórmula. Ej: si X = 3 entonces el arreglo tendrá: 8, 1, 6, -1, 4, -3, 2, -5, 0.

Detalle clave: la instrucción TEST sirve para realizar un AND pero sin almacenar el resultado en ningún lado.

```
        ORG 1000h
num_x db 3 ; este es el número inicial (podemos elegir cualquiera)
tabla db ? ; acá se irá almacenando el array a partir de la posición 1001h

        ORG 2000h
MOV BX, OFFSET tabla
MOV AL, num_x
REPETIR: TEST AL, 00000001B
            JNZ ES_IMPAR
                SUB AL, 7
                JMP FIN_IF
            ES_IMPAR: ADD AL, 5
            FIN_IF: MOV [BX], AL
            INC BX
            CMP AL, 0
            JNZ REPETIR
HLT
END
```

Y nos queda el array que pedía el ejercicio. Podemos verlo en la memoria desde la posición 1001h:

Recordemos que:
-1 en Ca2 = 11111111 = FF
-3 en Ca2 = 11111101 = FD
-5 en Ca2 = 11111011 = FB



13) Dada la frase "Organización y la Computación", almacenada en la memoria, escriba un programa que determine cuantas letras ‘a’ seguidas de ‘c’ hay en ella.

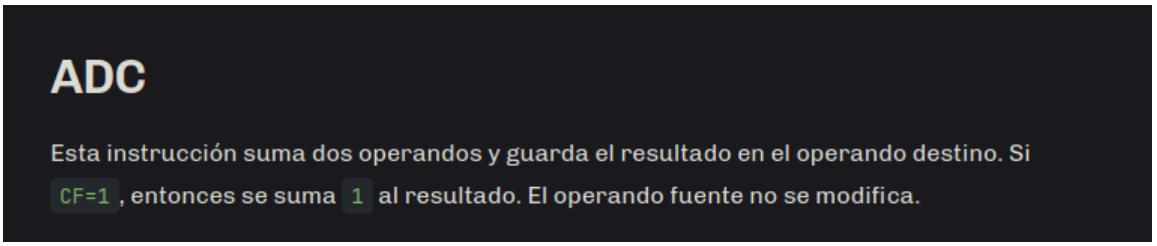
Aclaraciones:

- las etiquetas ES_A y ES_AC no cumplen ninguna función lógica, están únicamente para mejorar la legibilidad del programa.
- el programa cuenta la cantidad de substrings ‘ac’ que existen en el string “Organización y la Computación”. Para ello se fija recorre todo el string buscando la letra ‘a’, y si la encuentra, se fija si le sigue una ‘c’. Si encuentra un substring ‘ac’, le suma uno al contador que está temporalmente almacenado en AH. Una vez que finaliza, guarda el resultado en la variable cant, que está en la posición 1000h.
- En este caso, el resultado será 2, ya que el substring ‘ac’ aparece dos veces en el string “"Organización y la Computación"”.

```
   ORG 1000h
cant db ? ; 1000h, acá guardaremos el resultado del análisis
frase db "Organización y la Computación" ; empieza en 1001h
fin_frase db ? ; 101Eh

   ORG 2000h
MOV BX, OFFSET frase
MOV AL, OFFSET fin_frase - OFFSET frase ; 29 en decimal o 1D en hexadecimal
MOV AH, 0
REPETIR: MOV CL, [BX]
            CMP CL, 'a'
            JNZ NOT_ac
            ES_A: INC BX
                  DEC AL
                  JZ FIN_REP ; si se terminó el array
                  MOV CL, [BX]
                  CMP CL, 'c'
                  JNZ NOT_ac
                     ES_AC: INC AH ; si se encontró un par 'ac'
            NOT_ac: INC BX
            DEC AL
            JNZ REPETIR
FIN_REP: MOV cant, AH
HLT
END
```

14) Escribir un programa que sume dos números representados en Ca2 de 32 bits almacenados en memoria de datos y etiquetados NUM1 y NUM2 y guarde el resultado en RESUL (en este caso cada dato y el resultado ocuparán 4 celdas consecutivas de memoria). Verifique el resultado final y almacene 0FFH en la celda BIEN en caso de ser correcto o en otra MAL en caso de no serlo. Recordar que el MSX88 trabaja con números en Ca2 pero tener en cuenta que las operaciones con los 16 bits menos significativos de cada número deben realizarse en BSS.

Para este caso viene bien la instrucción ADC:




```
        ORG 1000h
NUM1L DW 0EF11h
NUM1H DW 0ABCDh ; NUM1 = ABCDEF11h
NUM2L DW 5678h
NUM2H DW 1234h ; NUM2 = 12345678h
RESULL DW ? ; está en 1008h..1009h
RESULH DW ? ; está en 100Ah..100Bh
BIEN DB 0 ; está en 100Ch
MAL DB 0 ; está en 100Dh

        ORG 2000h
MOV AX, NUM1L ; AX se usará para sumar la parte baja
MOV BX, NUM1H ; BX se usará para sumar la parte alta
ADD AX, NUM2L
ADC BX, NUM2H
JO MALIOSAL ; si O = 1, salió mal
    MOV BIEN, 0FFh ; si no, salió all right
    JMP FIN_IF
MALIOSAL: MOV MAL, 0FFh
FIN_IF: MOV RESULL, AX
MOV RESULH, BX
HLT
END
```

15) Escribir un programa que efectúe la suma de dos vectores de 6 elementos cada uno (donde cada elemento es un número de 32 bits) almacenados en memoria de datos y etiquetados TAB1 y TAB2 y guarde el resultado en TAB3. Suponer en primera instancia que no existirán errores de tipo aritmético (ni carry ni overflow), luego analizar y definir los cambios y agregados necesarios que deberían realizarse al programa para tenerlos en cuenta.

Aclaraciones:

- Cada tabla contiene 6 números de 32 bits (4 Bytes). Como la capacidad máxima de almacenamiento es 16 bits (2 Bytes), cada número ocupa dos lugares (primero la parte baja, después la parte alta). Por eso el tamaño de las tablas es 12 (en el sentido de que tiene 12 elementos).
- Ahora bien, las celdas son de 8 bits (1 Bytes), así que cada medio número ocupa dos celdas (2 Bytes). Por eso cada tabla necesita 24 Bytes, y requiere, por ello, 24 direcciones de memoria. Esto significa que desde una posición en una tabla, a la misma posición en la siguiente tabla hay 24 direcciones de distancia.

```
        ORG 1000h
CANT_NUM DB 6 ; 1000h
TABLA1 DW 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6 ; de 1001h hasta 1018h
TABLA2 DW 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6 ; de 1019h hasta 1030h
TABLA3 DW ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ? ; de 1031h hasta 1048h

        ORG 2000h
MOV BX, OFFSET TABLA1
MOV DL, CANT_NUM
REPETIR:  MOV AX, [BX] ; carga en AX la parte baja de un elemento de TABLA1
        ADD AX, [BX + 24] ; suma las partes bajas de un elemento de TABLA1 y TABLA2
        PUSHF ; guarda las flags de la suma de la parte baja
        MOV [BX + 48], AX ; almacena el resultado de la parte baja en TABLA3
        ADD BX, 2 ; avanza un elemento en el array
        MOV AX, [BX] ; carga en AX la parte alta de un elemento de TABLA1
        POPF ; restaura las flags
        ADC AX, [BX + 24] ; suma las partes altas de un elemento de TABLA1 y TABLA2
        MOV [BX + 48], AX ; almacena el resultado de la parte alta en TABLA3
        ADD BX, 2 ; avanza un elemento en el array
        DEC DL
        JNZ REPETIR
HLT
END
```

Para tener en cuenta los dos posibles errores mencionados por la consigna, deberíamos agregar cuatro condicionales después de la operación ADC, uno por cada posible acontecimiento: 1) C = 0 y O = 0, 2) C = 1 y O = 0, 3) C = 0 y O = 1 y 4) C = 1 y O = 1.

16) Los siguientes programas realizan la misma tarea, en uno de ellos se utiliza una instrucción de transferencia de control con retorno. Analícelos y compruebe la equivalencia funcional.

PRIMER PROGRAMA

```
  ORG 1000h
NUM1 db 5
NUM2 db 3

  ORG 2000h
MOV AL, NUM1
CMP AL, 0
JZ FIN
MOV AH, 0
MOV DX, 0
MOV CL, NUM2
LOOP: CMP CL, 0
        JZ FIN
        ADD DX, AX
        DEC CL
        JMP LOOP
FIN: HLT
END
```

SEGUNDO PROGRAMA

```
  ORG 1000h
NUM1 db 5
NUM2 db 3

  ORG 3000h
SUB1:  CMP AL, 0
        JZ FIN
        CMP CL, 0
        JZ FIN
        MOV AH, 0
        MOV DX, 0
LAZO: ADD DX, AX
        DEC CX
        JNZ LAZO
FIN:  RET

  ORG 2000h
MOV AL, NUM1
MOV CL, NUM2
CALL SUB1
HLT
END
```

Ambos programas multiplican NUM1 por NUM2. ¿Cómo? Suman NUM1 a sí mismo (en un registro auxiliar (en este caso DX)) tantas veces como lo indique NUM2. Por eso, como en estos programas NUM1 = 5 y NUM2 = 3, el producto será 5 * 3 y se calcula realizando la operación 5 + 5 + 5. El resultado queda almacenado en DX.

(Aclaración: el segundo programa por lo general funciona mal, ya que en vez de decrementar CL decrementa CX. Y si en CH hay algún valor basura distinto de 0, esto hará que el lazo se ejecute una cantidad de veces totalmente errónea. Lo único que hay que acomodar ahí para que funcione bien es poner CL en vez de CX).

Ambos programas revisan que ninguno de los operandos sea 0 antes de ponerse a realizar sumas sucesivas, ya que si alguno de los dos es 0, el resultado será 0. Lo que no está claro es

si eso está bien implementado: en el primer programa, si el primer operando es 0 se realiza un salto hacia el final sin dejar ningún indicador de que el resultado fue 0, mientras que si el primero tenía algún valor y el segundo era 0, DX se dejaba en 0; en el segundo programa, en cambio, no se deja nada indicado en ningún caso.

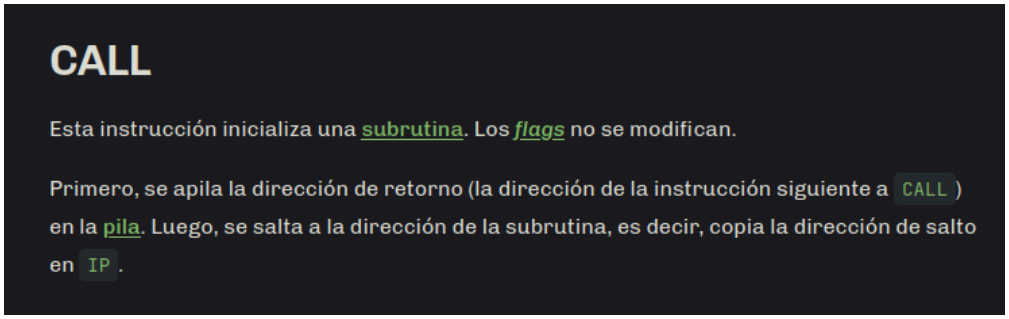
En cuanto al tiempo de ejecución, puede que haya algunas ventajas con los números pequeños para el primer programa, ya que no tiene el llamado a subrutina. Pero como el bucle de las sumas sucesivas está mejor implementado en la subrutina, si tiene que hacer muchas repeticiones, a la larga será mejor el segundo programa.

Explicar detalladamente:

a) Todas las acciones que tienen lugar al ejecutarse la instrucción CALL SUB1.

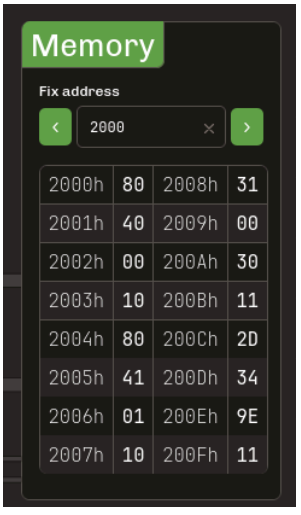
b) ¿Qué operación se realiza con la instrucción RET?, ¿cómo sabe la CPU a qué dirección de memoria debe retornar desde la subrutina al programa principal?

Tal como lo dice la documentación:



Ahora veamos paso a paso cómo realiza el VonSim esta llamada utilizando como ejemplo el programa anterior.

Primero revisemos cómo está la memoria del programa principal:



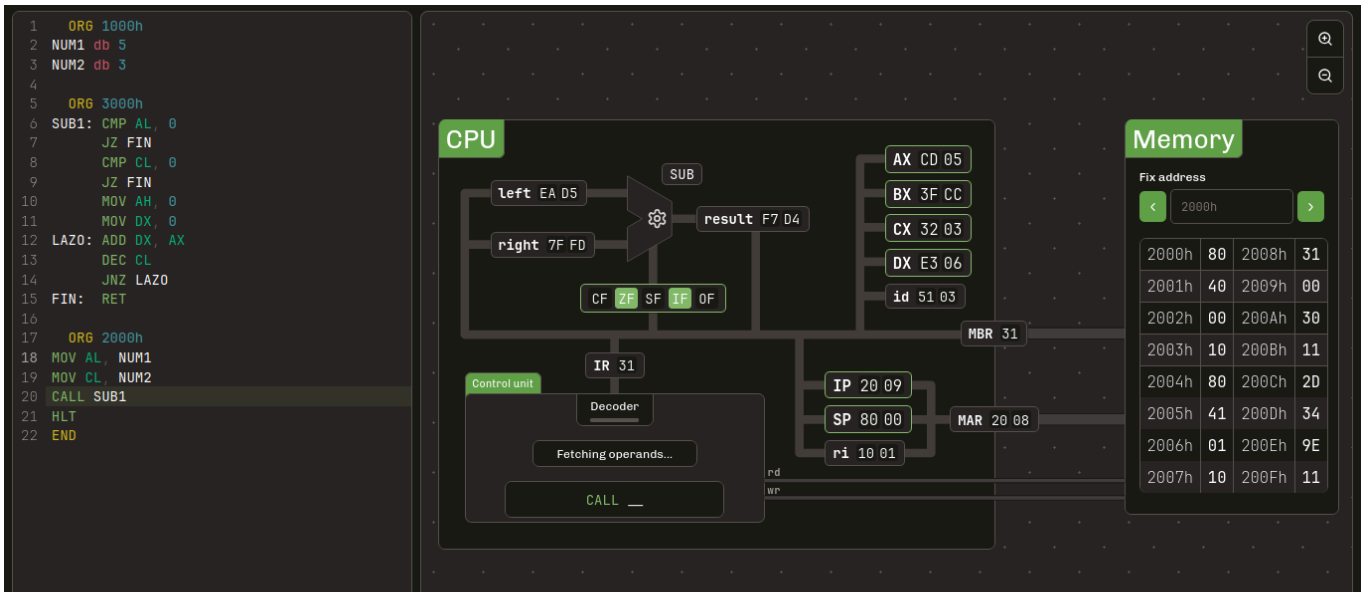
_ En 2008h está CALL

_ En 2009h y 200Ah está su “operando”, o sea la dirección a la cual va a llamar (3000h).

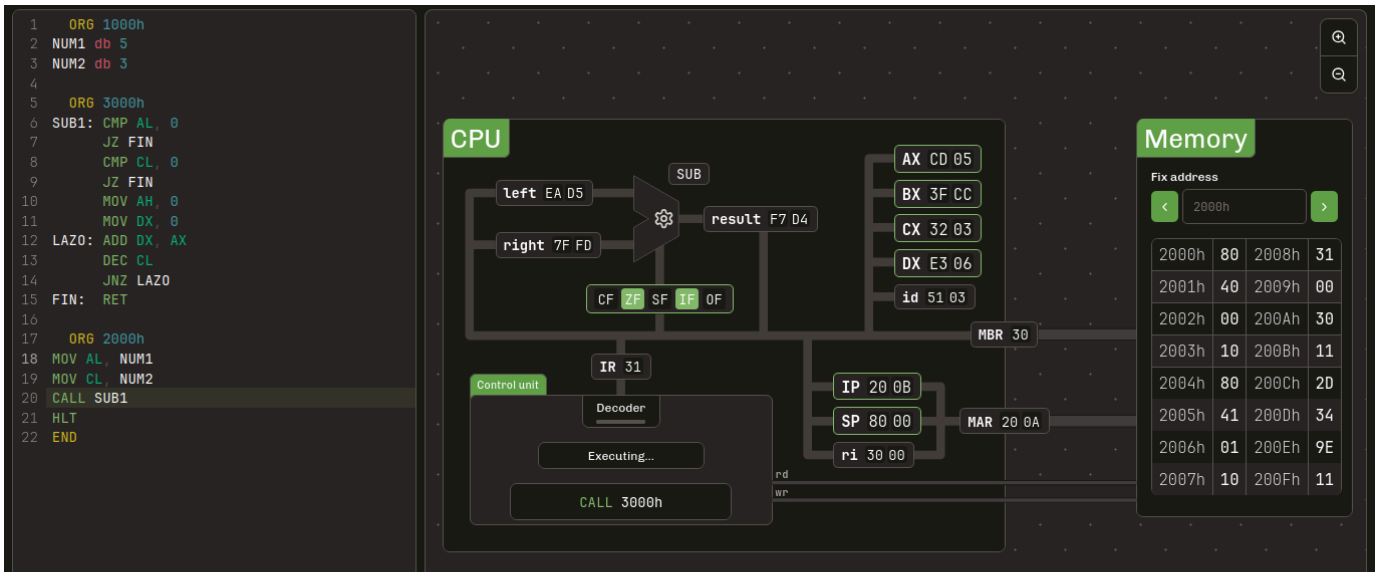
_ En 200Bh está HLT

Por ello, la próxima instrucción de CALL es HLT

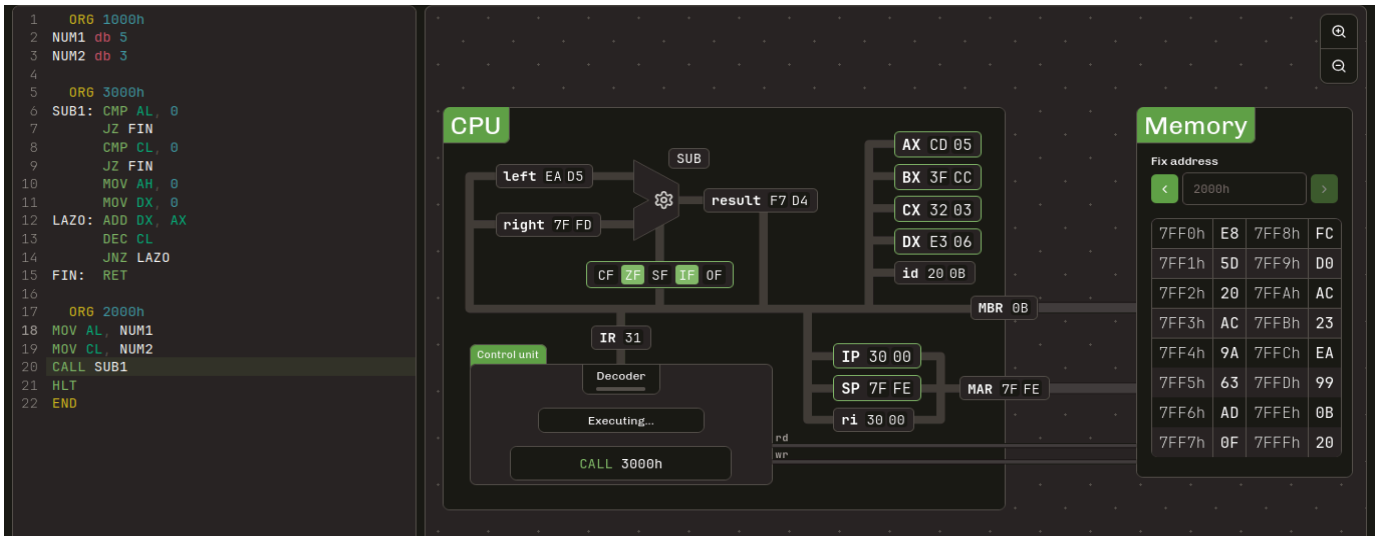
Comencemos. Primero que nada, VonSim lee lo que está en la dirección 2008h y ve que es un CALL.



Ahora: ¿a quién llamo? —se pregunta el buen VonSim. *A lo que tengo en 2009h y 200Ah* —se responde a sí mismo. Entonces va a esas direcciones y ve que allí está el 3000h.



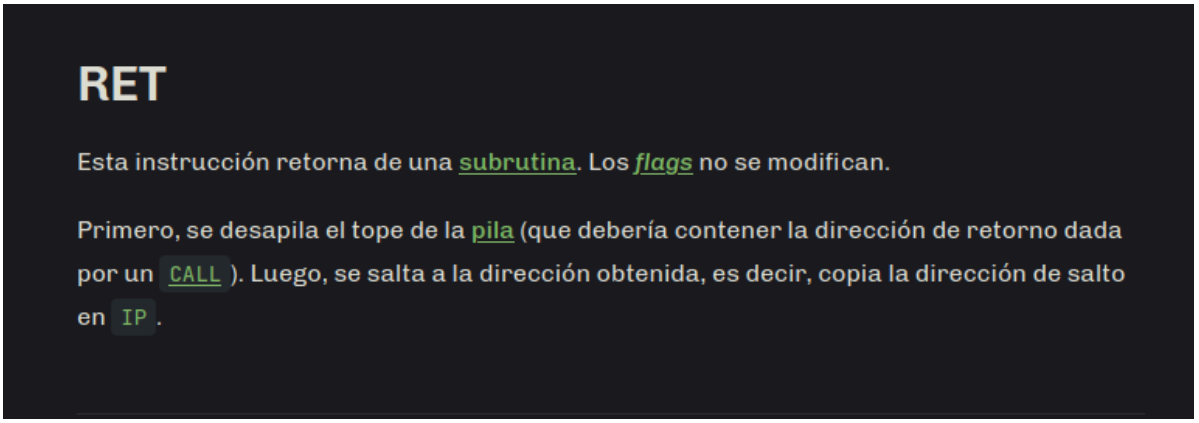
Por lo tanto —concluye—, *la próxima instrucción será la que esté en esa dirección, pero antes debo guardar la que tenía guardada en IP como la dirección de mi próxima instrucción* — y por eso almacena 200Bh en el stack (en las direcciones 7FFFh y 7FFEh).



Y listo, ya estamos dentro de la subrutina, ya que la próxima instrucción (tal como se ve en el IP) será aquella que esté en 3000h, y esa es el CMP AL, 0.

Ahora veamos qué pasa cuando termina la subrutina.

Según la documentación, sucede lo siguiente:



En nuestro ejemplo, una vez que el VonSim llega RET, copia la dirección que está en SP (7FFEh) en el MAR, y desapila así la primera parte del dato que necesitamos, y luego le suma uno a SP (7FFFh) y desapila la segunda parte — y SP queda en 8000h. Así obtiene el dato completo, que es la dirección que habíamos guardado anteriormente (200Bh).

Ahora bien, tal como lo advierte la documentación, hay que tener cuidado con el stack cuando estamos dentro de una subrutina, ya que no podemos dejar el SP con una dirección cualquiera: tiene que contener la misma que tenía al inicio de la subrutina. De lo contrario, RET nos mandará a cualquier lado.

Aclaración sobre el funcionamiento del stack. El SP está inicializado en 8000h, pero siempre que va almacenar un dato, comienza por restarle uno a la dirección que tiene. Por eso el primer dato siempre se guarda en las celdas 7FFFh y 7FFEh. La 8000h en realidad nunca se usa (y, de hecho, no existe).

17) El siguiente programa es otra forma de implementación de la tarea del punto anterior (ejercicio 16). Analizar y establecer las diferencias con las anteriores, en particular las relacionadas a la forma de 'proveer' los operandos a las subrutinas.

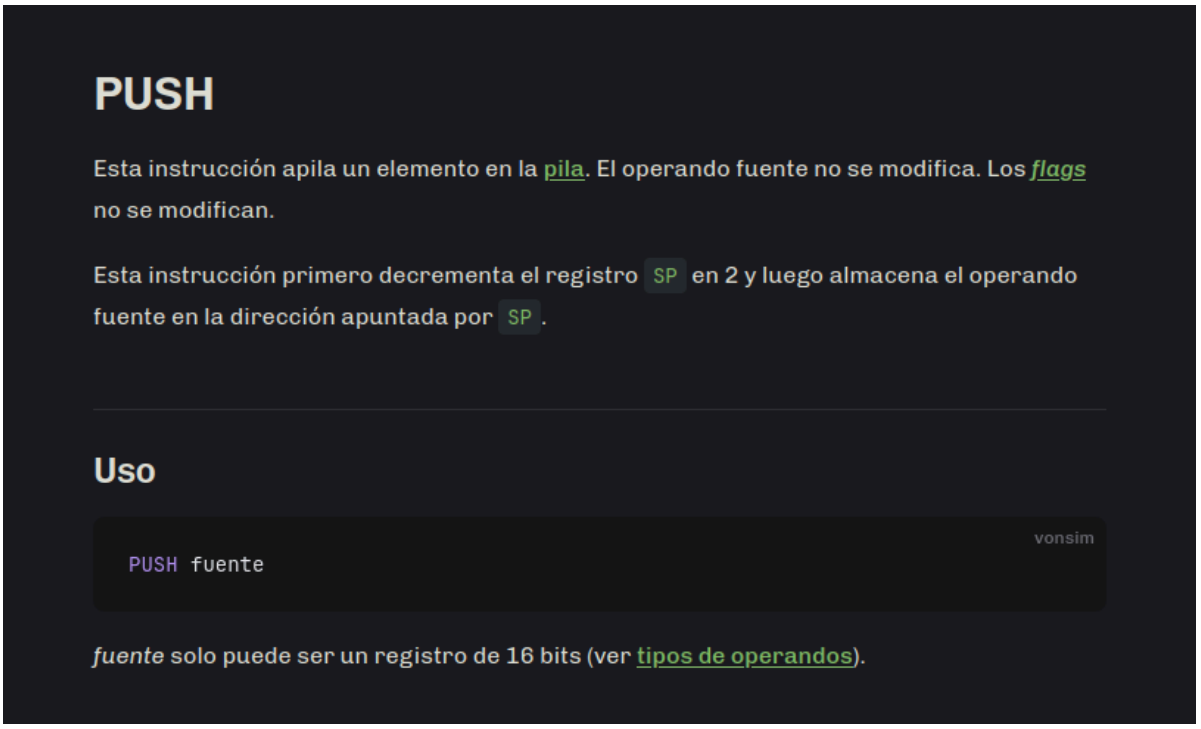
```
        ORG 1000H
NUM1 DW 5H ; NUM1 y NUM2 deben ser mayores que cero
NUM2 DW 3H

        ORG 3000H
SUB2: MOV DX, 0 ; se inicializa DX en 0
LAZO: MOV BX, AX ; se carga la dirección de NUM1 en BX
        ADD DX, [BX] ; se le suma NUM1 a lo que haya en DX
        PUSH DX ; se guarda DX (con el resultado parcial de la multiplicación) en el stack
        MOV BX, CX
        MOV DX, [BX] ; se carga NUM2 en DX
        DEC DX ; NUM2 := NUM2 - 1
        MOV [BX], DX ; se guarda el NUM2 actualizado
        POP DX ; volvemos a poner el resultado parcial de la multiplicación en DX
        JNZ LAZO ; si NUM2 no llegó a 0, volvemos al lazo
        RET

        ORG 2000H
MOV AX, OFFSET NUM1
MOV CX, OFFSET NUM2
CALL SUB2
HLT
END
```

La idea principal es la misma que antes: realizar una multiplicación mediante sumas. La diferencia es que no se tiene en cuenta la posibilidad de que alguno de los operandos sea cero, y en vez de trabajar de forma sencilla sumando y decrementando registros, se utiliza el direccionamiento indirecto y el stack.

Explicar detalladamente:

a) Todas las acciones que tienen lugar al ejecutarse las instrucciones PUSH DX y POP DX.

Según la documentación:



PUSH

Esta instrucción apila un elemento en la pila. El operando fuente no se modifica. Los *flags* no se modifican.

Esta instrucción primero decrementa el registro `SP` en 2 y luego almacena el operando fuente en la dirección apuntada por `SP`.

Uso

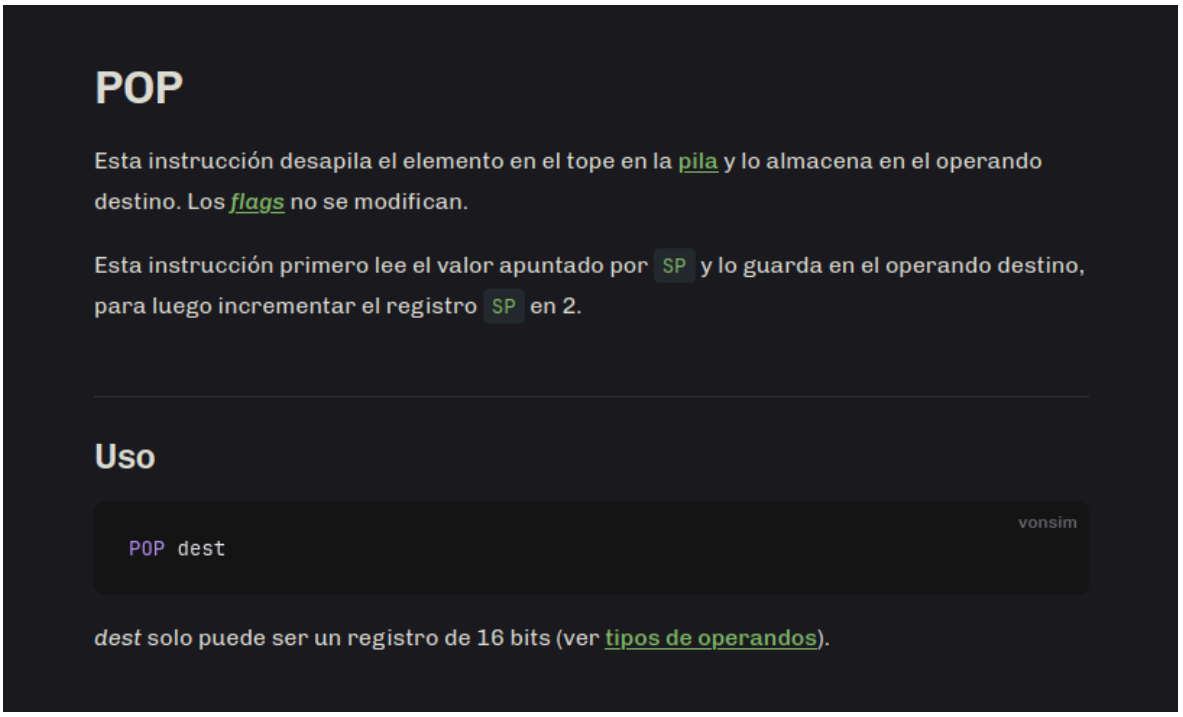
vonsim

```
PUSH fuente
```

fuente solo puede ser un registro de 16 bits (ver tipos de operandos).

Resulta conveniente repetir algo que se dijo antes: el SP está inicializado en 8000h, pero siempre que va almacenar un dato, VonSim comienza por restarle uno a la dirección que tiene. Por eso el primer dato siempre se guarda en las celdas 7FFFh y 7FFEh. La 8000h en realidad nunca se usa (y, de hecho, no existe).

Otra cosa: el stack siempre guarda de a 2 Bytes, por eso siempre ocupa o desocupa dos celdas de memoria.



POP

Esta instrucción desapila el elemento en el tope en la pila y lo almacena en el operando destino. Los *flags* no se modifican.

Esta instrucción primero lee el valor apuntado por `SP` y lo guarda en el operando destino, para luego incrementar el registro `SP` en 2.

Uso

```vonsim
POP dest
```

dest solo puede ser un registro de 16 bits (ver tipos de operandos).

b) Cuáles son los dos usos que tiene el registro DX en la subrutina SUB2.

DX se usa para ir almacenando los resultados parciales de la multiplicación, y para almacenar temporalmente NUM2 (para así poder restarle 2).

18) Escribir un programa que sume 2 vectores de 6 elementos (similar al realizado en el ejercicio 15), de modo tal que utilice una subrutina que sume números de 32 bits (similar al programa escrito en ejercicio 14).

Algunas abreviaturas: “+” = suma, “pp.” = partes, “bs.” = bajas, y “as.” = altas.

```
        ORG 1000h
CANT_NUMEROS DB 6 ; 1000h
TABLA1 DW 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6 ; de 1001h hasta 1018h
TABLA2 DW 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6 ; de 1019h hasta 1030h
TABLA3 DW ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ? ; de 1031h hasta 1048h

        ORG 3000h
SUM_VEC:    MOV AX, [BX] ; carga en AX la parte baja de un elemento de TABLA1
            ADD AX, [BX + 24] ; + las pp. bs. de un elemento de TABLA1 y TABLA2
            PUSHF ; guarda las flags de la suma de la parte baja
            MOV [BX + 48], AX ; almacena el resultado de la parte baja en TABLA3
            ADD BX, 2 ; avanza un elemento en el array
            MOV AX, [BX] ; carga en AX la parte alta de un elemento de TABLA1
            POPF ; restaura las flags
            ADC AX, [BX + 24] ; + las pp. as. de un elemento de TABLA1 y TABLA2
            MOV [BX + 48], AX ; almacena el resultado de la parte alta en TABLA3
            ADD BX, 2 ; avanza un elemento en el array
            DEC DL
            JNZ SUM_VEC
            RET

        ORG 2000h
MOV BX, OFFSET TABLA1
MOV DL, CANT_NUMEROS
CALL SUM_VEC
HLT
END
```

19) Escriba una subrutina que reciba la mantisa entera en BSS y el exponente en BSS de un número en los registros AH y AL respectivamente y devuelva, en ellos, una representación equivalente del mismo pero con el exponente disminuido en 1 y la mantisa ajustada. De no ser posible el ajuste, BL debe contener 0FFH en vez de 00H en el retorno.

Aclaraciones preliminares:

- En este caso, la mantisa puede ajustarse si y solo si el bit más significativo es 0, ya que al disminuir el exponente en 1, deberemos aumentar la mantisa (o sea, correr los bits de la mantisa una posición a la izquierda).
- Correr los bits de un número en BSS una posición a la izquierda, equivale a multiplicarlo por 2. Y multiplicar por 2 es lo mismo que sumar el número a sí mismo.
- El exponente no puede ser disminuido si es 0.

```
        ORG 1000h
MANTISA DB 00000110b ; sería el 6
EXPONENTE DB 00010000b ; sería el 16

        ORG 3000h
FLOTANTE:  TEST AH, 10000000b ; 1er condición: ver si podemos ajustar la mantisa
           JNZ NO_PODEMO
                CMP AL, 0 ; 2da condición: ver si podemos decrementar el exponente
                JZ NO_PODEMO
                     DEC AL ; si está todo ok, restamos uno al exponente
                     ADD AH, AH ; multiplicamos por dos la mantisa
                     MOV BL, 00h ; avisamos que salió bien
                     JMP FIN_IF ; nos vamos
           NO_PODEMO: MOV BL, 0FFh
           FIN_IF: RET

        ORG 2000h
MOV AH, MANTISA
MOV AL, EXPONENTE
CALL FLOTANTE
HLT
END
```

21) Modifique la subrutina del ejercicio 19 para el caso en que la mantisa y el exponente estén representados en BCS.

Aclaraciones preliminares:

- Ahora, la mantisa no puede ajustarse cuando el segundo bit más significativo es 1.
- Si el número es positivo, podemos correr los bits igual que antes. Si es negativo, hay que elegir otra estrategia. Una posibilidad es sumarlo a sí mismo (lo mismo que para el positivo), y luego agregarle el 1 del negativo (que se fue con el carry).
- El exponente no puede ser disminuido si está en el límite inferior del rango, es decir, si es 11111111.
- Si el exponente es negativo, para decrementarlo debemos sumarle 1.

```
        ORG 1000h
MANTISA DB 01110000b
EXPONENTE DB 10000000b

        ORG 3000h
FLOTANTE:  TEST AH, 01000000b ; 1ro: ver si podemos ajustar la mantisa
           JNZ NO_PODEMO
                CMP AL, 0FFh ; 2do: ver si podemos decrementar el exponente
                JZ NO_PODEMO
                     TEST AL, 10000000b ; chequeamos el signo del exponente
                     JZ EX_POS
                          INC AL ; exponente negativo
                          JMP FIN_EX
                     EX_POS: DEC AL ; exponente positivo
                     FIN_EX: ADD AH, AH ; multiplicamos por dos la mantisa
                     JNC MAN_POS ; si no hubo carry, es porque era positiva
                          OR AH, 10000000b
                     MAN_POS: MOV BL, 00h ; avisamos que salió bien
                     JMP FIN_IF ; nos vamos
           NO_PODEMO: MOV BL, 0FFh
           FIN_IF: RET

        ORG 2000h
MOV AH, MANTISA
MOV AL, EXPONENTE
CALL FLOTANTE
HLT
END
```

20) Escriba una subrutina que reciba como parámetro un número en el formato IEEE 754 de simple precisión y analice/verifique las características del mismo devolviendo en el registro CL un valor igual a

- **0 si el número está sin normalizar,**
- **1 en caso de ser +/- infinito,**
- **2 si es un NAN,**
- **3 si es un +/- cero y**
- **4 si es un número normalizado.**

La subrutina recibe en AX la parte alta del número y en BX la parte baja.

Máscaras a utilizar:

- 0111111110000000 → 7F80h →para analizar el exponente
- 0000000001111111 → 007Fh →para analizar la parte alta de la mantisa
- 1111111111111111 → 0FFFFh → para analizar la parte baja de la mantisa

```
        ORG 1000h
PARTEL DW 0h
PARTEH DW 0FFF0h

        ORG 3000h
IEEE:  TEST AX, 7F80h ; primero revisamos si el exponente es cero
        JNZ NOT_Z
        ; si el exponente es 0, hay dos posibilidades
           TEST AX, 007Fh
           JNZ DENORMAL
              TEST BX, 0FFFFh
              JNZ DENORMAL
                 MOV CL, 3 ; es un cero
                 JMP FIN_ANALISIS
           DENORMAL: MOV CL, 0 ; es un número denormalizado
                        JMP FIN_ANALISIS
        ; si el exponente no es 0, hay otras tres posibilidades
        NOT_Z:  XOR AX, 7F80h ; invierto los bits del exponente, y dejo el resto igual
        ; si los bits del exponente eran todos unos, ahora son todos ceros
                    TEST AX, 7F80h ; entonces, si esto da 0, es porque eran todos unos
                    JNZ NZ_N1 ; si el exponente "son" todos unos, hay dos posibilidades
                    TEST AX, 007Fh
                    JNZ NAN
```

```
        TEST BX, 0FFFFh
        JNZ NAN
            MOV CL, 1 ; es un infinito
            JMP FIN_ANALISIS
NAN:    MOV CL, 2 ; es un NAN
            JMP FIN_ANALISIS
NZ_N1: MOV CL, 4 ; es un número normalizado
FIN_ANALISIS: RET

    ORG 2000h
MOV AX, PARTEH
MOV BX, PARTEL
CALL IEEE
HLT
END
```

ANEXO: instrucciones válidas e inválidas.

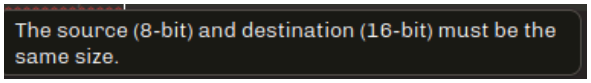
ALGUNAS VÁLIDAS

```
  ORG 1000h
num1 db 2
num2 db 3

  ORG 2000h
MOV AX, 228h
MOV AX, AX
MOV AX, 2
MOV AX, OFFSET num1
MOV BX, AX
MOV BYTE PTR [BX], 33h
MOV CH, num1
MOV [BX], AL
HLT
END
```

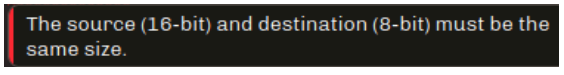
ALGUNAS INVÁLIDAS

```
  ORG 1000h
num1 db 2
num2 db 3

  ORG 2000h
```

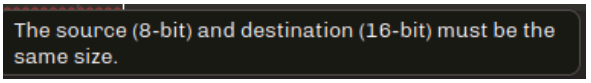
MOV AX, BL



MOV AL, BX

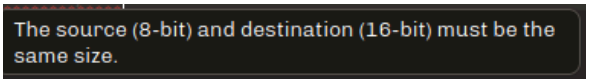


MOV AX, NUM1

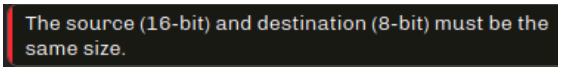


PUSH NUM1

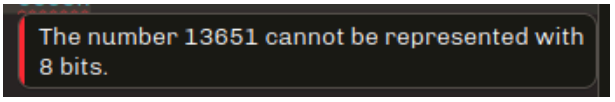
ADD AL, BX



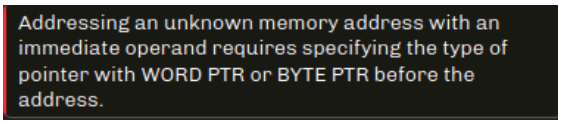
ADD BX, AL



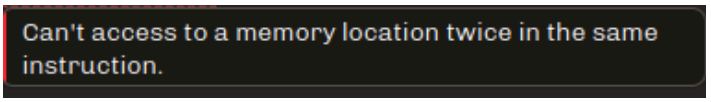
MOV BYTE PTR [BX], 3553h



MOV [BX], 3553h



MOV [BX], NUM2



HLT
END