Resumen OC - Esta completo

Organización de Computadoras (Universidad Nacional de La Plata)

Practica 5

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 ó 3 direcciones.

Maq. de 1 dirección	Maq. de 2 direcciones	Maq. de 3 direcciones

Para resolverlo tenemos que ver la operación que en este caso es F= [(A+B)/C]-D, si vemos bien F es igual A más B dividido C menos D.

Aclaración: Una variable se almacena en memoria y la memoria se maneja con direcciones. Entonces cada dirección referencia a una celda donde yo tengo una variable.

Entonces cuando me hablan de una Maquina de 1 o 2 o 3 direcciones significa que por instrucción puedo manejar una, dos o tres variables al mismo tiempo, es decir la maquina 1 solo puedo trabajar con una sola variable al mismo tiempo, la maquina 2 con dos variables y la maquina 3 con tres variables al mismo tiempo.

Paso 1) Entonces el ejercicio me pide que implemente esa formulita en el Lenguaje Ensamblador, donde tengo que sumar A+B, dividido C, menos D y me dan algunos valores para el ejemplo. Si reemplazamos los valores de A= 6, B= 4, C= 2, D=1, F=¿?, vemos que ((6+4)/2))-1 = 4.

Entonces para implementar esas operaciones en cualquier máquina, necesitamos conocer las instrucciones como cargar, sumar, restar, dividir y almacenar en el ensamblador, estas operaciones las llamamos **Código de Operación,** por ejemplo**: Load, add, div, sub, store, etc.** Estos Cod. De operación ocupa espacio en la memoria.

Las direcciones son las letras (operando o variables): A, B, C, D, E, etc. Estas también ocupan lugar en la memoria, dependiendo en cada caso.

Las operaciones son:

- load: carga la variable en contexto (para poder usarla)
- add, sub, div: operación de suma, resta y división
- store: almacena del contexto a una variable

- **Máquina de 1 dirección**: Como esta maquina solo puedo trabajar con una sola variable.

  1) Entonces lo primero que hago es cargar en la memoria la variable A **(load A)**, para poder comenzar a trabajar y a partir de ahí yo puedo trabajar.

Maq. de 1 dirección
load A

6

Luego tengo que sumarle a "A" el valor de B, pero como no podemos trabajar con más de dos variables entonces trabajo abajo.

2) Luego sumo B, es decir (add B)

Maq. de 1 dirección
load A add B

6 + 4 = 10

3) Luego divido C, es decir (div C)

Maq. de 1 dirección
load A add B div c

10/2 = 5

4) Luego resto D, es decir (sub D)

Maq. de 1 dirección
load A add B div C sub D

5 – 1 = 4

5) Listo, ahora para que ese resultado se almacene, entonces uso (store f)

Maq. de 1 dirección
load A add B div C sub D store F

4

- **<u>Maquina 2 dirección</u>**: con esta maquina puedo trabajar con dos variables.

  1) Entonces en esta segunda maquina no hace falta cargar, sino que asignamos a “F” el valor de “A”, porque podemos trabajar con dos variables. Entonces acá para poder asignar el valor de ”A“ a F usamos la el código de operación “**mov”,** entonces quedaría “Mov F, A” que significa que a F asígnale el valor de A ó que a la variable F le asignamos el valor de A.

  Entonces en resumen en la máquina de 2 dirección si o si primero comienzo asignándole a F el valor de A, en la máquina de 1 dirección era cargando el valor de A.

Maq. de 1 dirección
mov F, A

F = A

  2) Luego como a F le asigne el valor de A, y como ahora tengo que sumar A+B, pero como se que F = A, entonces quedaría (add F, B), es decir a “Súmale a F el valor de B” y ese resultado ya queda guardado en F

Maq. de 1 dirección
mov F, A add F, B

F = F + B

  3) luego tengo que dividir el resultado de esa suma, es decir a F le tengo que dividir C, (div F, C)

Maq. de 1 dirección

mov F, A
add F, A
div F, C

F = F / C

  4) Luego le resto al resultado de dividir por C a F, es decir a F ya que se guarda el resultado en F, (sub F, D)

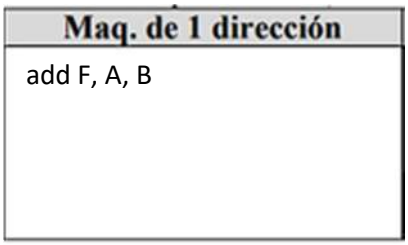
Maq. de 1 dirección

mov F, A
add F, A
div F, C
sub F, D

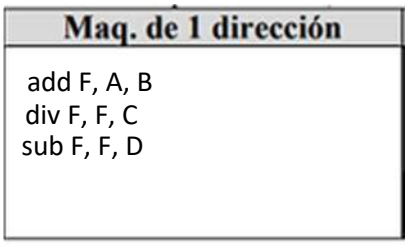
F = F - D

- Maquina de 3 direcciones: acá ya podemos usar 3 variables al mismo tiempo.

1) Entonces acá no hace falta usar ni “load”, ni “mov”, sino que acá directamente comenzamos sumando y guardándolo en F, es decir (add F, A, B) esto se lee como, “sumo A + B y lo almaceno en F



2) Luego como el resultado de A+B se guardo en F y sabiendo que luego tengo que dividir ese resultado por C, entonces (div F, F, C) es decir, divido el valor de F con C y el resultado se guarda en F.



3) Luego como el resultado de esa división se guarda en F y sabiendo que luego tengo que restar ese resultado por D, entonces (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:

Esto es solo una suposición, dependiendo en cada caso puede ocupar mas o menos, además sabemos que cada maquina de direcciones tiene su ventaja y desventaja, y es que la Maquina 1 de direcciones ocupa menos espacio en la memoria que la 2 y la 3 maquina, y la 2 maquina de direcciones ocupa menos espacio que la tercera, pero no mucho menos que la primera y la tercera es la que más memoria ocupa de todas. Entonces.

Ahora Calcularemos lo que ocupa el Programa en la Memoria

Para calcular la cantidad de espacio que ocupa el programa en la memoria es sumar la cantidad de espacio que ocupa el código de operaciones + lo que ocupa las direcciones.

M. de 1 dirección :

Entonces **para calcular la cantidad de espacio que ocupa el programa en la memoria es** sumar la cantidad de espacio que ocupa el código de operaciones + lo que ocupa las direcciones.

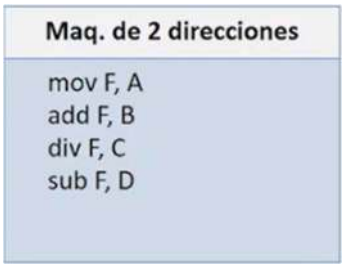
Códigos de Operaciones, para calcular lo que ocupa la cantidad de operaciones, en el caso de la primera maquina tenemos **5** en total (load, add, div, sub, store) y como en el enunciado nos dice que cada código de operación ocupa 6 bits, entonces multiplico 5 * 6 = 30.

Direcciones, para calcular lo que ocupa las direcciones, en el caso de la primera máquina tenemos **5** en total (A, B, C, D, F) y como en el enunciado nos dice que ocupa 10 bits cada dirección, entonces multiplico 5 * 10 = 50.

Entonces…

Cantidad que ocupa en la memoria = 30 + 50 = 80 bits. Si queremos saber lo que ocupa en bytes, dividimos 80/8, es decir 10 bytes.

	M. de 1 dirección
Tam del prog en mem (cod.operación + op)	(5*6) + (5*10) = = 80 bits = 10 bytes



M. de 2 dirección :

Entonces **para calcular la cantidad de espacio que ocupa el programa en la memoria es** sumar la cantidad de espacio que ocupa el código de operaciones + lo que ocupa las direcciones.

Códigos de Operaciones, para calcular lo que ocupa la cantidad de operaciones, en el caso de la segunda máquina tenemos **4** en total (mov, add, div, sub) y como en el enunciado nos dice que cada código de operación ocupa 6 bits, entonces multiplico 4 * 6 = 24.

Direcciones, para calcular lo que ocupa las direcciones, en el caso de la segunda máquina tenemos **8** en total (F, A, F, B, F, C, F, D) y como en el enunciado nos dice que ocupa 10 bits cada dirección, entonces multiplico 8 * 10 = 80.

Entonces…

Cantidad que ocupa en la memoria = 24 + 80 = 104 bits. Si queremos saber cuánto ocupa en valores Bytes, dividimos 104/8, es decir 13 bytes.

M. de 2 direcciones
(4*6) + (4*(10+10)) = = 104 bits = 13 bytes

Maq. de 3 direcciones

add F, A, B
div F, F, C
sub F, F, D

M. de 3 dirección :

Entonces **para calcular la cantidad de espacio que ocupa el programa en la memoria es** sumar la cantidad de espacio que ocupa el código de operaciones + lo que ocupa las direcciones.

Códigos de Operaciones, para calcular lo que ocupa la cantidad de operaciones, en el caso de la tercera máquina tenemos **3** en total (add, div, sub) y como en el enunciado nos dice que cada código de operación ocupa 6 bits, entonces multiplico 3 * 6 = 18.

Direcciones, para calcular lo que ocupa las direcciones, en el caso de la segunda máquina tenemos **9** en total (F, A, B F, F, C, F, F, D) y como en el enunciado nos dice que ocupa 10 bits cada dirección, entonces multiplico 9* 10 = 90.

Entonces…

Cantidad que ocupa en la memoria = 18 + 90 = 108 bits. Si queremos saber cuánto ocupa en valores Bytes, dividimos 108/8, es decir 13.5 bytes.

M. de 3 direcciones
(3*6) + (3*(10+10+10)) = = 108 bits = 13,5 bytes

Ahora veremos como calcular la Cantidad de acceso a memoria que tengo dependiendo de cada máquina de direcciones.

Entonces **para calcular la cantidad de accesos que tengo en cada maquina es:** sumar la cantidad de instrucciones que tengo + los operandos que tengo o variables.

Maq. de 1 dirección
load A add B div C sub D store F

M. de 1 dirección :

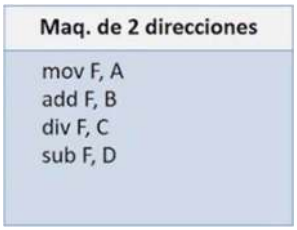
Entonces **para calcular la cantidad de accesos que tengo en cada maquina es:** sumar la cantidad de instrucciones que tengo + los operandos que tengo o variables.

Instrucciones, para calcular lo que ocupa la cantidad de instrucciones es sumar la cantidad de operaciones o instrucciones que estoy utilizando, en este caso **5** instrucciones (load, add, div, sub, store)

Operandos, para calcular lo que ocupa las operandos o variables, en el caso de la primera máquina tenemos **5** en total (A, B, C, D, F)

Cantidad que accesos = 5 + 5 = 10 accesos

Cant de accesos a mem (instrucciones + op)	(5 + 5) = 10 accesos



M. de 2 dirección :

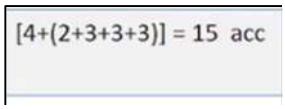
Entonces **para calcular la cantidad de accesos que tengo en cada máquina es:** sumar la cantidad de instrucciones que tengo + los operandos que tengo o variables.

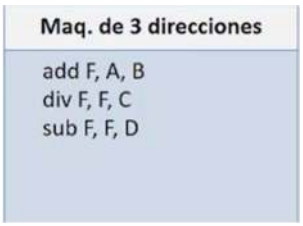
Instrucciones, para calcular lo que ocupa la cantidad de instrucciones es sumar la cantidad de operaciones o instrucciones que estoy utilizando, en este caso **4** instrucciones (mov, add, div, sub)

Operandos, para calcular lo que ocupa las operandos o variables, en el caso de la segunda máquina tenemos **11** en total (F, A = ocupa 2 + F, B = ocupa 3 + F, C = ocupa 3 + F, D = ocupa 3) 2+3+3+3.

Seguro se preguntarán ¿Por qué ocupa 3 la suma, la división y la resta y el mov solo 2? Primero que nada "Mov F, A" significa que acceso a F (+1) para asignarle el valor de "A" (+1) en total ocupa 2. Luego cuando accedo a B para tener el valor de "B" (+1), luego accedo "F" para sumar el valor de "B" (+1), y accedo a F de nuevo para guardar el valor de la suma (+1) en total ocupa 3. Lo mismo sucede con la suma, división y resta.

Cantidad que accesos = 4 + 11 = 15 accesos.





M. de 3 dirección :

Entonces **para calcular la cantidad de accesos que tengo en cada máquina es:** sumar la cantidad de instrucciones que tengo + los operandos que tengo o variables.

Instrucciones, para calcular lo que ocupa la cantidad de instrucciones es sumar la cantidad de operaciones o instrucciones que estoy utilizando, en este caso **3** instrucciones (mov, add, div, sub)

Operandos, para calcular lo que ocupa las operandos o variables, en el caso de la segunda máquina tenemos **9** en total (F, A, B = ocupa 3 + F, F, C = ocupa 3 + F, F, D = ocupa 3)

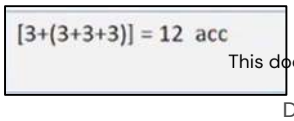
3+3+3

Seguro se preguntarán ¿Por qué ocupa 3 la suma, la división y la resta y el mov ? Primero cuando accedo a B para tener el valor de "B" (+1), luego accedo a A para acceder el valor de "A", sumo A+B y el resultado se guarda en "F" , por ende, accedo a F para guardarlo (+1), si sumo todo, me da que en total ocupa 3.

Luego cuando accedo al valor de "F" (+1) y a también accedo al valor de C (+1) para poder dividir, el resultado se guarda en F, por ende, accedo a "F" (+1), en total ocupa 3.

y lo mismo con la resta, que ocupa 3. En total sumado todos, da 9. Entonces

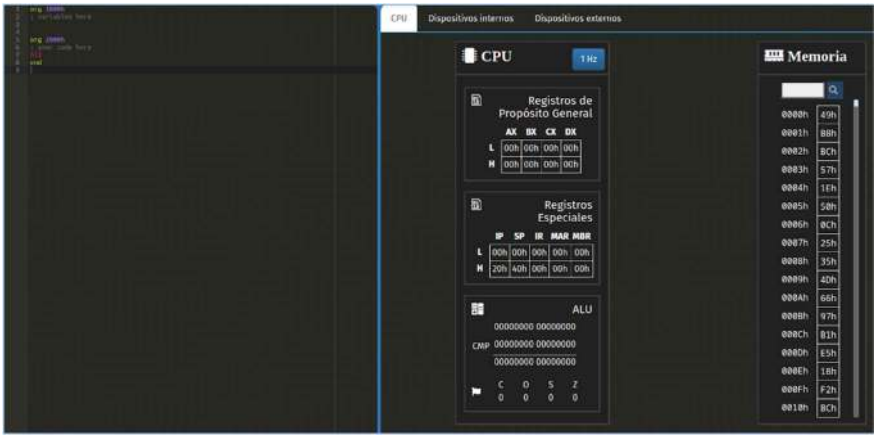
Cantidad que accesos = 3 + 9 = 12 accesos.



Assembler

En esta cursada veremos el lenguaje ensamblador de los procesadores de la familia Intel 8086.

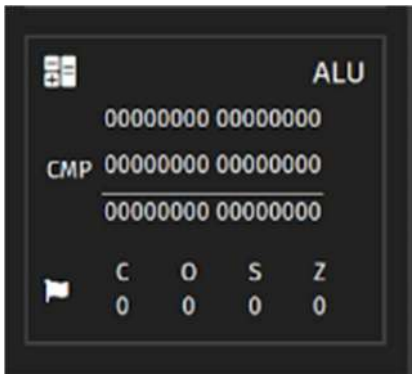
- Maneja operaciones de 2 direcciones.
- Posee 4 registros: **AX, BX, CX, DX**
- Los números negativos se representan en Ca2
- Vamos a utilizar el simulador Vonsim. Y se puede acceder a través de https://vonsim.github.io/
- También se puede hacer uso del simulador MSX88 (solo Windows) que se puede descargar en la plataforma de la materia.



Registros especiales, no me importa

- Acá tenemos la CPU con los registros que habíamos mencionado anteriormente

Y acá tenemos la memoria



- Acá tenemos la ALU que es donde dijimos que se hacían los sumas y restas (ADD Y SUB) y abajo tenemos una banderita que son los Flags, si se acuerdan estaban el **C** = Carry, **O** = Overflow, **S** = Signo y **Z** = Zero.

Comencemos

1) Almacenamiento

Los datos pueden estar en:

- En **Memoria** (variables, códigos, etc.) Ó
- En **registros**

La diferencia entre memoria o registro es que la memoria tiene un coste, es un poquito más lenta que los registros, pero los registros son menos. Van a ver las memorias guardan muchísima información y en los registros son solo 4 (AX, BX, CX, DX)

¿Cómo definimos una variable? Ponemos nombre de la variable, el tipo y el valor, solo hay 2 tipos de variables, **DB** (Data Byte) Y **DW** (Data Word).

<nombre> **DB** <valor inicial> (8 bits/1 byte)
<nombre> **DW** <valor inicial> (16 bits/2 bytes)

¿Cuál es la diferencia entre Data Byte (DB) y Data Word (BW)? La diferencia es que DB (data byte) solo ocupa solo 8

- Por ejemplo, si yo defino una variable "NUM" y uso el tipo "DB", quedando como "NUM DB ....." " ¿Qué valor puedo poner acá? ¿Puedo poner el 256 por ejemplo? Para saber esto, si se acuerdan, como DB trabaja con 1 byte (8 bits)

XXXXXXXX $\left[0 ; 2^{N} - 1 \right]$ Rango

1 1 1 1 1 1 1 1 $= 2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7 = 255$

Vemos que lo máximo que yo podría representar en 8 bits es 255, entonces 256 no se puede representar en DB (data byte) porque DB trabaja solo con 1 byte (8 bits) y el 256 esta fuera de rango. Entonces si queremos representar el 256 tenemos que usar el BW que puede almacenar mucho más.

Más Ejemplos:

Ejemplos

var1	DB	10
var2	DW	? (No inicializado)
tabla	DB	1, 2, 3, 4, 5
string	DB	"Esto es un string" (ASCII)
var3	DW	**0**ABCD**H** (Hexadecimal)
var4	DB	10101010**B** (Binario)

Var1: Vemos que acá en el ejemplo, puso el nombre a la variable "var1" y uso el tipo "DB" (data byte, ocupa 1 byte), y le puso un valor, en este ejemplo el 10 (cuando uso DB el valor que le doy no puede superar los 255, es decir el máximo es 255).

Var2: Vemos que acá en el ejemplo, puso el nombre a la variable "var2" y uso el tipo "DW" (data Word, ocupa 2 byte) y en este ejemplo no le puso ningún valor, sino que puso el signo de pregunta "?" que indica que no está inicializado.

Tabla: vemos que podría armar un arreglo, puso el nombre a la variable "Tabla" y uso el tipo "DB" (ocupa 1 byte) y le dio el valor de 1, 2, 3, 4 ,5. Cuando están separados con comas se llaman arreglos, en donde cada uno ocupa 1 byte porque use "DB", es decir el 1 ocupa un byte, el 2 también, y así....

string: Vemos que esto se llama arreglo de números, puso el nombre “string” y uso el tipo “DB” (ocupa 1 byte, 8 bits) y le dio valor “Esto es string” (ASCII) eso es un código asqui, en donde un numero significa un carácter

Var3: Vemos que, en el ejemplo, puso el nombre a la variable “var3” y uso el tipo “DW” (ocupa 2 byte, 16 bits) y le dio el valor “**0**ANCD**H**”, en donde si le pongo un cero adelante y un H al final, lo que digo es que lo del medio es un hexadecimal.

Var4: Vemos que, en el ejemplo, puso el nombre a la variable “var4” y eso el tipo “DB” (ocupa 1 byte, 8 bits) y le dio el valor “10101010**B**”, esto no es más que un numero en binario y al final lo puso la letra B para indicar que es en binario.

Registros

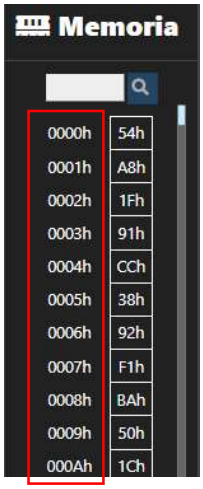
Los registros se usan almacenar información que sean muy rápido y también lo voy a usar para operar

Son 4 en total = AX, BX, CX, DX. Cada uno tiene su parte baja en inglés “Low” representado con la letra **(L)** y parte alta que en inglés es High representado con la letra **(H)** . Y cada registro puede almacenar 2 bytes, en donde cada parte almacena 1 byte.

Es decir, por ejemplo, **AX** puede almacenar 2 bytes, en donde la parte baja (L) está compuesta por 1 byte y la parte alta (H) compuesta por 1 byte, siendo así en total 2 bytes. Lo mismo sucede con BX, CX, DX.

2) Memoria y Posicionamiento

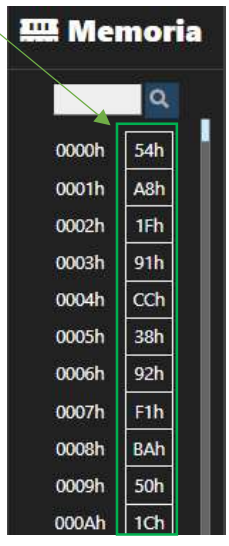
Este procesador puede direccionar una memoria de 16 bits. Es decir, que tenemos 2^16 celdas de memoria que podemos utilizar.



- Por simpleza, la dirección de cada celda se pone en Hexadecimal.

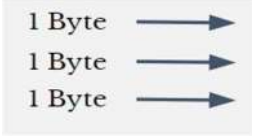
para no poner 16 ceros hasta 16 unos, ponemos la dirección en Hexadecimal, en vez de poner 16 ceros lo pongo en Hexadecimal, **0**000**h** *(dirección de la primera celda)*, en donde el primer 0 y el ultimo h indica que está en hexadecimal, y lo que esta en el medio es el valor 0. Luego está la dirección de la segunda celda de memoria “0001h”.

El valor de las celdas



- Cada celda de memoria puede almacenar 1 byte de datos.

Es decir, tengo 1 byte acá, otro byte acá, tengo otro acá, etc.



Sepan que cuando hablamos de Windows 32 o 64 bits, hablamos de cuanta memoria maneja un sistema operativo.

Luego si nos fijamos bien**, tenemos el valor de la celda** ahí (marcado con verde), en este momento no se que valor tiene, tiene un valor basura, que también se pone en Hexadecimal. Fijensen que dos dígitos hexadecimales (54) son 8 bits, o sea que por comodidad para poder leerlo mejor me lo ponen en Hexadecimal (54h)

Cuando escribimos código **debemos indicar en qué posición de la memoria** queremos operar.

Esto se consigue con la sentencia ORG.

En Pascal todos los datos iban en la memoria, bueno en este procesador en muy importante que indiquemos en que posición queremos empezar a escribir.

Entonces tenemos que decir en que parte de la memoria empezamos a definir variables y en que parte de la memoria empezamos a definir código.

Sintaxis

ORG <dirección de memoria>

Entonces la sintaxis es ORG y luego la memoria en donde queremos empezar a trabajar.

Usualmente definimos las variables a partir de la posición 1000H y el código de nuestro programa a partir de la posición 2000H (esto obviamente en valor hexadecimal, no es que vale 2000.) Entonces veamos el código más simple que conoceremos:

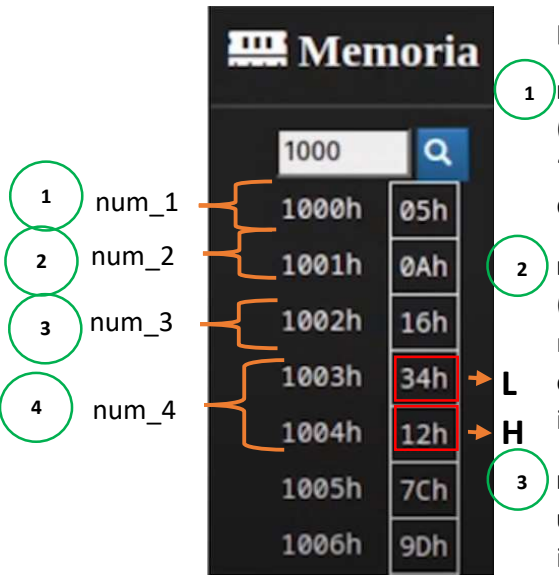
```
ORG 1000H
    num_1    DB 5
    num_2    DB 10
    num_3    DB 016H
    num_4    DW 01234H

ORG 2000H
    ... ; El código de mi programa principal
    HLT ; Esta línea y la de abajo siempre deben ir
    END
```

Explicación

ORG 1000H (me está indicando "posiciónate en la posición 1000H"). Luego tenemos 4 variables (num_1, num_2, num_3, num_4).

Después me voy a la posición 1000H y empiezo a escribir el código de mi programa, esas dos líneas siempre tienen que estar: "**END**" le dice al compilador que deje fijarse en el código y el "**HLT**" (hall) que le dice al compilador que el programa termina acá.

Veamos qué es lo que pasa en la memoria cuando escribí eso:



Explicación

ORG 1000H (me está indicando "posiciónate en la posición 1000H"). Si yo voy a la posición 1000H en la memoria ¿Qué pasa ahora?

1. **num_1:** Dijimos vamos a definir una variable que se llame "num_1" del tipo "DB" (que ocupe 1 byte, es decir **ocupa una celda de memoria**) y que inicie con el valor "5" y si se fijan en 1000 se inicializo con el valor 5, como el 5 en decimal es lo mismo que en hexadecimal "05h".

2. **num_2:** Ahora en la próxima celda defino la variable "num_2" con el tipo "DB" (ocupa 1 byte, es decir **una celda de memoria**) con el valor "10", como en la memoria trabaja con hexadecimal, entonces se guarda en hexadecimal que es A, entonces quedaria"0Ah", recuerde que se pone un cero adelante y al final un h indicando que está en hexadecimal.

3. **num_3**: Lo mismo con la variable "num_3" con el tipo "DB" (ocupa 1 byte, es decir **una celda de memoria**) y que inicie con el valor "**0**16**H**" el cero al inicio y el H al final indicando que inicialice 16 en hexadecimal.

4. **num_4:** Acá le ponemos el nombre a la variable "num_4" con el tipo "DW" (ocupa 2 byte, es decir **ocupa dos celdas de memoria**) con el valor "**0**1234**H**", es decir con el valor 1234 en hexadecimal, acuérdense que 1234 no me entra en "DB", es decir no me entra en una celda de memoria, porque DB llega hasta 255 como máximo y yo quiero trabajar con 1234, y en cambio "DW" como máximo llega hasta 65535, ocupando 2 celdas. Como 1234 ocupa dos celdas de memoria, la parte alta (**H**) representado por 12h y la parte baja representado por 34(**L**), la parte alta siempre va abajo (H) (+ significativo) y la parte baja (L) va hacia la parte de arriba (+ significativo)

Decimal	Binario	Hexadecimal
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

3) TRANSFERENCIA

Aclaración: La manera de transferir información es a partir de la sentencia **MOV.**
¿Cómo se usa? mov <*destino*>, <*origen*>. Por ejemplo "mov AL, num_1" que significa moveme el valor de la variable num1 a la parte baja del registro AX

Entonces del ejemplo que teníamos anteriormente…, ahora vamos a empezar a usar los Registros.

```
ORG 1000H
    num_1    DB 5
    num_2    DB 10
    num_3    DB 016H
    num_4    DW 01234H

ORG 2000H
    mov AL, num_1 ; Copio num_1 a la parte baja de AX
    mov AH, num_2 ; Copio num_2 a la parte alta de AX
    mov DX, num_4 ; Copio los 16 bits de num_4 al registro DX
    mov CX, DX ; Copio el registro DX al registro CX
    mov num_1, AH ; Copio el valor de AH a la variable num_1
    mov num_1, num_2 ; PROHIBIDO
    mov num_1, [BX] ; PROHIBIDO
    hlt
    end
```

1 **mov AL, num_1:** significa cópiame el valor de la variable "num_1" y movelo a la parte baja del registro AX

2 **mov AH, num_2**: significa cópiame el valor de la variable "num_2" y movelo a la parte alta del registro AX

3 **mov DX, num_4**: significa cópiame el valor de la variable "num_4" (que ocupa 2 bytes, es decir16 bits) y movelo al registro DX, cuando trabajo con "DW" (2 bytes) yo puedo copiar una variable de dos bytes a un registro de 2 bytes. Es decir, cuando uso DW (2 bytes) y por ejemplo hago esto: "mov DL, num_4", acá lo que esta pasando es que estoy perdiendo 1 byte de información, porque le estoy diciendo al compilar que copie algo de 2 bytes ("DW") en algo de 1 byte DL. Entonces por eso uso el Registro Completo DX, "mov DX_ num_4"

4 **mov CX, DX** : significa cópiame el registro DX y movelo al registro CX. Es decir, que Incluso también puedo copiar de registro a otro registro, de DX copio y lo muevo a CX.

Mov es simplemente asignación. **Lo que no puedo hacer es mover una memoria a otra memoria**, es decir: si por ejemplo yo quiero pasar el valor 10 (num_2) a num_1 y hago esto "mov num_1, num:2" (Prohibido), está prohibido pasar de una memoria a otra. Lo que sí podría hacer es hacer "mov num_1, AL", es decir como AL (parte baja de AX) vale 10, lo que hago copiar AL y lo muevo a num_1, que es lo que puse anteriormente.

Tampoco lo que no se puede hacer es mover el registro completo a una memoria, por ejemplo si hago "num_1, num_2" (PROBIBIDO)

El resto de las combinaciones si se puede hacer, "de registro a registro", "de memoria a registro", "de registro a memoria".

3) MODOS DE DIRECCIONAMIENTO

Hay 4 modos de direccionamiento que se pueden utilizar

Inmediato	Directo	Directo por registro	Indirecto por reg.
mov AL, 9	*mov AL, num_1*	*mov AL, AH*	*mov BX, 1000H* *mov AL, [BX]*
Se utiliza un valor fijo. No requiere 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.

Indirecto por registro: El registro BX, puede servir como puntero, es el único registro que puede servir como puntero.

mov BX, 1000H
mov AL, [BX]

¿Qué sucede acá? veamos

Primero veamos como funciona:

Tenemos un valor que queremos usar (que se encuentra en la memoria), **llamado operando** y del otro lado tenemos los registro con información **AX, BX, CX, DX**, pero en el único que me voy a concentrar es BX (dirección del operando) que puede tener cualquier valor, ya sea un 5 o un 8 o lo que carajo sea, pero tengo una dirección de memoria, es la dirección del operando que quiero usar, entonces.

¿Qué va a pasar si hacemos "mov BX, 1000H" **en nuestro código?** Pero antes aclararemos algunas cosas, primero es que cuando pongo un numero en hexadecimal por ejemplo "**0**10**H**", es lo mismo que ponerlo así "10**H**", para que no piensen que es distinto. **¿Por qué en BX y no en BL?** Porque 1000 ocupa 2 bytes, y si ocupa 2 bytes no puedo usar BL porque BL trabaja con 1 byte, y si eso pasara estaría perdiendo 1 byte de información, es decir no puedo almacenarlo en la parte baja y alta al "BW" que usamos el valor 1000, necesito el registro completo en "BX".

¿Qué pasa si agrego abajo en el compilador lo siguiente "mov AL, {DX}"? Estos corchetes me están diciendo "anda a donde está apuntando BX.

<u>Ahora si realmente veamos qué es lo que sucede si hacemos lo siguiente</u>:

ORG 2000H
mov BX, 1000H
mov AL, [BX]

Como 1000 es un valor grande, que solo entraría en "DW", por lo tanto, ocupa 2 bytes, 1 byte en parte alta y otra en parte baja, la parte 10h (parte alta) 00h (parte baja)

1

Trabajamos con el anterior ejemplo

Paso 1**) "mov BX, 1000H"**: copio el valor de 1000H y lo muevo a BX

Paso 2) **"mov AL, [BX]"**: cuando ejecute esa segunda instrucción, lo que va a pasar es que va a copiar el valor de la celda de 1000H ubicada en la memoria (parte derecha), como habíamos visto anteriormente, por ejemplo el valor de 1000H en la celda ubicado en la memoria vale 05h en binario, es decir 5, bueno copia ese 5 y luego de que la copie lo va a mover a "AL"

Ahora ¿Qué sucede si hacemos la siguiente instrucción?

ORG 2000H
mov BX, OFFSET num_3
mov AL, [BX]

1) "Mov BX, OFFSET num_3" Si depuramos eso, lo que va a hacer es buscar el valor de **num_3**, vemos que vale "16H" en la celda de memoria (en hexadecimal) luego lo que va a hacer es que va a buscar en la memoria de direcciones la memoria en donde contenga en su celda ese valor ***16h,*** si vemos en la memoria, la memoria de dirección que tiene como valor de celda 16h es ***1002h***. Entonces ese valor de la memoria de direcciones es la que va a copiar y lo va a mover a BX, como vemos en la captura "C", en donde BL (Parte Baja) 02h y BH (Parte Alta) 10h

A

```
1  ORG 1000H
2  num_1    DB 5
3  num_2    DB 10
4  num_3    DB 16H
5  num_4    DW 1234H
```

C

B

¿Por qué? Porque si nos fijamos en la memoria el ***1000h*** con el valor de la celda "05h", es decir 5, no es más que el valor de **num_1**, es decir ***1000h*** es el **num_1**, y ***1001h*** con el valor de la celda que vale "A (en hexadecimal)" que en decimal es 10, no es más que **num_2,** es decir ***1001h*** es el **num_2** y el ***1002h*** es el ***num_3***.

Entonces con la instrucción "OFFSET" yo ya no tengo que poner fijo el numerito porque a veces no me sirve, es decir a veces me tengo que manejar de manera dinámica (que me diga el procesador donde esta esa

2) "mov AL, [BX]": lo que va a hacer es que lo que apunta BX, y si revisamos en BX apunta ***1002h***, y entonces nos vamos a la memoria y nos posicionamos en la dirección de la celda (marcado con verde "1002H" y vemos cuánto vale la celda (marcado con rojo) y vemos que vale "16h", entonces ese "16h" cópialo y movelo a "AL" y entonces ahora "AL" quedaría con el valor "16h"

A

B

C

Aclaración (de vuelta)

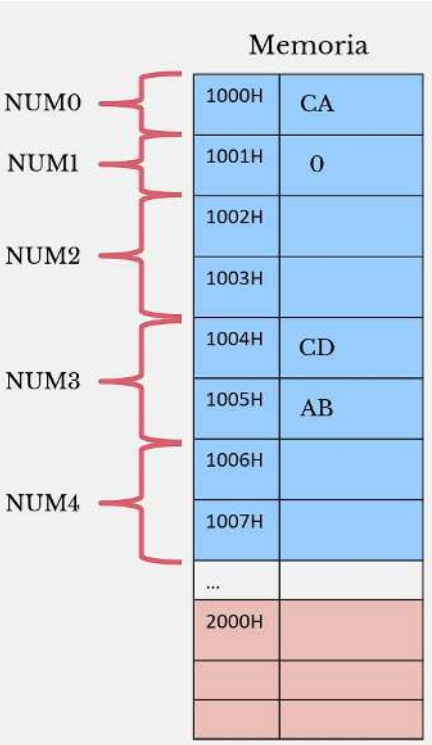
Como vimos anteriormente, en la memoria habíamos dicho que generalmente por convención comenzábamos a trabajar en la posición 1000H para definir los datos o las variables (memoria de datos).

Y partir de la dirección de memoria 2000h, metíamos instrucciones o códigos (memoria de instrucciones).

A continuación, analizaremos lo siguiente para practicar, veamos el ejemplo:

```
ORG 1000H
    NUM0    DB 0CAH
    NUM1    DB 0
    NUM2    DW ?
    NUM3    DW 0ABCDH
    NUM4    DW ?
```

	AX	BX	CX	DX
L				
H				



ORG 1000H: Primero siempre ponemos en qué dirección de la celda memoria queremos comenzar, como habíamos dicho por lo general comenzamos en 1000H, ahora podemos asignarle datos a la memoria.

NUM0 DB 0CA**H**: le asignamos a la "primera dirección de memoria" el valor **0**CA**h** (hexadecimal) que es el CA**h**, con el nombre NUM0 y con el tipo "DB" (ocupando solo 1 byte).
NUM1 DB 0: con el nombre de NUM1 del tipo "DB" (ocupando solo 1 byte en la memoria) le asignamos a la segunda dirección de memoria el valor 0 (en decimal), pero en la memoria lo guarda en hexadecimal, es decir 00**h**

NUM2 DW ? : con el nombre NUM2 del tipo "DW" (ocupando 2 bytes, una parte alta y otra baja), con el valor "?", es decir sin ningún valor (no inicializado)

NUM3 DW 0ABCDH: con el nombre de NUM3 del tipo "DW" (ocupando 2 bytes, parte baja y alta) con el valor **0**ABCD**H** (en hexadecimal).

NUM4 DW ? : con el nombre NUM4 del tipo "DW" (ocupando 2 bytes, una parte alta y otra baja), con el valor "?", es decir sin ningún valor (no inicializado)

```
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], 1006H
HLT
END
```

ORG 2000H: ahora empezamos a realizar las instrucciones, como habíamos dicho que generalmente empezamos en 2000H para realizar las instrucciones.

MOV BL, NUM0: copiamos el valor de NUM0 (CAh) y lo movemos a BL

MOV BH, 0FFH: copiamos 0FFH y lo movemos a BH (direccionamiento directo)

MOV CH, BL: copiamos el registro de la parte baja BL (CAh) y lo movemos a CH

MOV AX, BX: copiamos el registro BX (BL = CA y BH = FFH) y lo movemos al registro AX, (quedando AL = CA Y AH = FFH)

MOV NUM1, AL: copiamos el registro de la parte baja (CA) (1 byte) y lo movemos a NUM1 (1byte), quedando ahora en la memoria CA y no 0

MOV NUM2, 1234H: copiamos 1234H (direccionamiento directo) (2 byte) y lo mandamos a NUM2 (que ocupa 2 byte), quedando en la memoria 12h (parte alta) y 34h (parte baja)

MOV BX, OFFSET NUM3: copiamos la "celda de memoria de direccionamiento" (parte izquierda) en donde tenga como valor el NUM3, como NUM3 trabaja con 2 bytes, tenemos 2 celdas, de esas dos celdas siempre se elige el menor, es decir 1004H

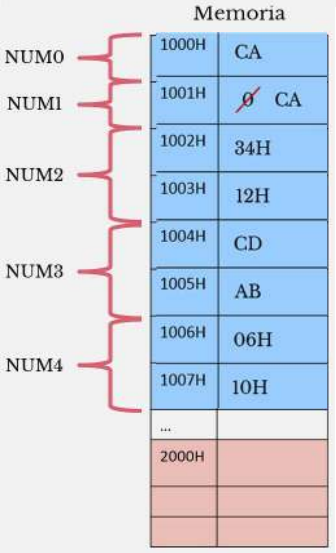
MOV DL, [BX]: acordémonos de los corchetes, que funcionaba como un puntero, es decir, apunta a BX, vemos cuando vale en los registros BX (BL = 04h Y BH = 10h)" BX = 1004H", ahora nos vamos en la memoria y en las celdas de direccionamiento buscamos "1004H", una vez encontrado vemos que el valor de esa celda de "1004h" es "CD", ese valor "CD" lo copiamos y lo movemos a DL, quedando ahora DL en CD.

MOV AX, [BX]: copio el valor de la celda apuntada, si en BX esta el 1004H, entonces buscamos en la memoria la celda de direccionamiento que tenga 1004H y copiamos el valor de esa celda y lo movemos a AX, como BX vale 1004H (es decir trabaja con "DW", 2 bytes) y como vemos que 1004H esta acompañado de otro byte, es decir 1004H tiene como valor de Celda "CD" y el de abajo que es 1005H también lo copio, porque están juntos, es decir 2 bytes, entonces me pide que lo mueva a AX (parte baja y parte Alta), quedando AX (AL = CD, Y AH = AB)

MOV BX, 1006H: copio 1006H y lo muevo a BX (direccionamiento directo).

MOV WORD PTR [BX], 1006H: copio el valor de 1006H y lo muevo a BX (registro), quedando 1006H en BX (BL = 06h y BH= 10)

	AX	BX	CX	DX
L	~~CA~~ CD	~~CA~~04H 06H		CD
H	~~FF~~ AB	~~FF~~ 10H 10H	CA	



Assembler

Ejemplo

```
ORG 1000H
    NUM0    DB 0CAH
    NUM1    DB 0
    NUM2    DW ?
    NUM3    DW 0ABCDH
    NUM4    DW ?
ORG 2000H
    MOV BL, NUM0        ; Copia valor de NUM0 en BL
    MOV BH, 0FFH        ; Copia valor FFH en BH
    MOV CH, BL          ; Copia BL en CH
    MOV AX, BX          ; Copia BX en AX
    MOV NUM1, AL        ; Copia AL en NUM1
    MOV NUM2, 1234H     ; Copia el valor 1234H en NUM2
    MOV BX, OFFSET NUM3 ; Copia la dir. de NUM3 en BX
    MOV DL, [BX]        ; Copia en DL el contenido de la celda apuntada por BX
    MOV AX, [BX]        ; Copia en AX el contenido de la celda apuntada por BX
    MOV BX, 1006H       ; Copia el valor 1006H en BX
    MOV WORD PTR [BX], 1006H ; Copia el valor 1006H a la celda apuntada por BX
HLT
END
```

	AX	BX	CX	DX
L	~~CA~~ CD	CA ~~04H~~ 06H		CD
H	~~FF~~ AB	~~FF~~ ~~10H~~ 10H	CA	

NUM0, NUM1, NUM2, NUM3, NUM4

Memoria

1000H	CA
1001H	~~0~~ CA
1002H	34H
1003H	12H
1004H	CD
1005H	AB
1006H	06H
1007H	10H
...	
2000H	

Y así es como quedo.

4) A) ARITMETICA

Contamos con varias operaciones aritméticas.

ADD = SUMA
SUB = RESTA

¿Cómo hacemos si queremos sumar dos operaciones? Hacemos:
ADD OP1, OP2 → en donde el resultado se guarda en la OP1

¿Cómo hacemos si queremos restar dos operaciones? Hacemos:
SUB OP1, OP2 → en donde el resultado se guarda en la OP1.

ADD OP1, OP2 → OP1 := OP1 + OP2
SUB OP1, OP2 → OP1 := OP1 - OP2

Por ejemplo, si tenemos lo siguientes

```
ORG 1000H
    num_1       DW 5
    num_2       DW 10
    res         DW ?

ORG 2000H
    MOV AX, num_1 ; Copio valor de num_1 a AX
    ADD AX, num_2 ; Sumo el valor de num_2 a AX
    MOV res, AX ; Guardo el resultado en la variable res
    HLT
    END
```

Pasos

Primero definimos en qué dirección queremos comenzar, en este caso "1000H" que es para definir los datos en la memoria.

En donde agrego la variable **"num_1"** con el tipo "DW" con valor 5, eso lo guarda en la memoria como "05h".

Luego hacemos lo mismo con **"num_2"** de tipo DW con valor 10.

Y agregamos otra variable con el nombre **"res"** con el tipo "DW" con el valor **"?"**, que es donde vamos a guardar el resultado.

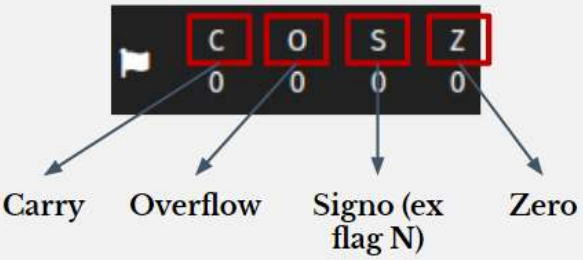
Ahora nos transportamos a 2000H para empezar a ejecutar las instrucciones.

MOV AX, num_1: copio el valor de num_1 y lo muevo a AX, es decir el valor 5 lo copio y lo muevo a AX (quedando en AL= **05h** y en AH (**00h**).

ADD AX, num_2 : sumo AX + num_2, como AX = 5 y num_2 = 10, da como resultado 15, ese resultado lo guardo en "AX" en hexadecimal.

MOV res, AX: Copio el resultado que me dio que se había guardado en AX (en registros) y lo muevo al dato "res" que habíamos definido anteriormente y no le había asignado ningún valor, quedando así guardado el resultado.

¡Estas operaciones setean los Flags que conocemos!



Es decir, si cuando opero con sumas, restas, o lo que fuese, durante esa operación puede hacer Carry, Overflow, Signo o Zero, por eso decimos que setean los Flags, es decir puede modificar los valores de los Flags.

De seguro se preguntarán. ¿Y no podría yo haber sumado directamente **"num_1 y num_2"?** ¿ Es decir hacer esto: **"ADD num1, num2"**? No, esta operación no se puede realizar, está prohibido porque básicamente no pueden acceder a la memoria

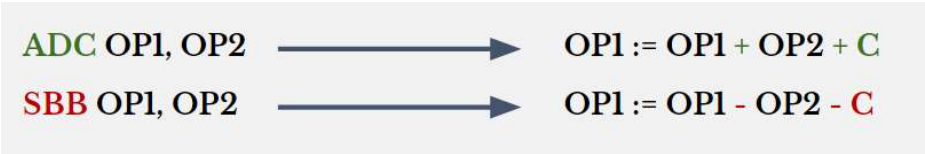
Es muy importante saber que cuando operamos no se tiene en cuenta los Carry, por ejemplo, la suma 80 + 80 que, en Hexadecimal, vendría ser 1000 0000 + 1000 0000 = 00000000 con Carry = 1, bueno ese Carry en el programa no lo va a tener en cuenta, pero hay una solución para ese problema. Por suerte contamos con la siguiente instrucción. Por ejemplo,

En la suma:

ADC OP1, OP2 → en donde el resultado se guarda en OP1, (esto se lee como la suma de la operación 1 + la operación 2 + el Flags Carry

En la resta:

SBC OP1, OP2 → en donde el resultado se guarda en OP1, (esto se lee como la resta de la operación 1 - la operación 2 - el Flags Carry



Útil para aquellas operaciones que podrían "irse de Rango"

También contamos con las instrucciones de Incremento y Decremento, cuando hablamos de incremento es básicamente incrementar en una unidad a la operación que se quiere aumentar de valor (+1), usamos la instrucción "INC" y decremento es lo mismo solo que cuando le queremos bajar de valor (-1), usamos la instrucción "DEC". Por ejemplo,

Incremento

INC OP1 → incremento en 1 (+1), la operación 1, el resultado se guarda en la OP1

Decremento

DEC OP1 → decremento en 1 (-1), la operación 1, el resultado se guarda en la OP1

También tenemos la instrucción de "Comparar", usando la Instrucción "CMP", esto solo sirve para consultar el estado de los Flags sin afectar los operandos, cuando uso "CMP" lo que estoy haciendo es restando dos operandos, pero no modifica el valor ni se guarda, solo es para saber el estado de los Flags. Por ejemplo,

CMP OP1, OP2 → comparo, OP1 – OP2

Solo setea los Flags. No modifica la Operación.

Veremos que esto resulta indispensable cuando veamos los saltos

4) B) Lógica

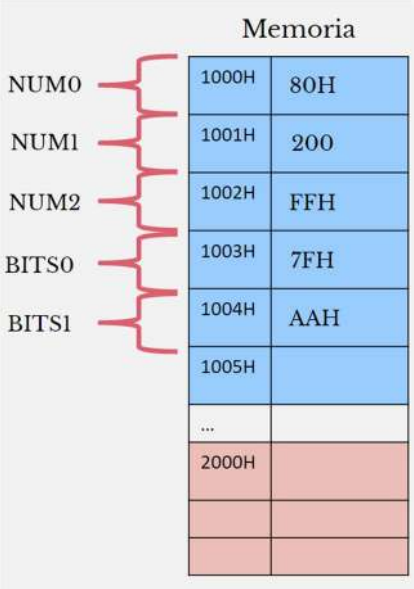
También tenemos las operaciones lógicas conocidas.

AND OP1, OP2	OP1 := OP1 AND OP2
OR OP1, OP2	OP1 := OP1 OR OP2
XOR OP1, OP2	OP1 := OP1 XOR OP2
NOT OP1	OP1 := ¬OP1 (Ca1)
NEG OP1	OP1 := ¬OP1 (Ca2)

¡Todas estas operaciones también setean los Flags!

Veamos un ejemplo con las instrucciones que mencionamos anteriormente.

```
ORG 1000H
    NUM0 DB 80H
    NUM1 DB 200
    NUM2 DB -1
    BITS0 DB 01111111B
    BITS1 DB 10101010B
```




```
ORG 2000H
    MOV AL, NUM0
    ADD AL, AL
    INC NUM1
    MOV BH, NUM1
    MOV BL, BH
    DEC BL
    SUB BL, BH
    MOV CH, BITS1
    AND CH, BITS0
    NOT BITS0
    OR CH, BITS0
HLT
END
```

ORG 1000H : Quiero empezar a definir los datos en la celda de memoria 1000H

NUM0 DB 80H: asigno el nombre de la variable num0 con el tipo DB con el valor 80H (en hexadecimal) en la dirección de celda 1000H

NUM1 DB 200: asigno el nombre de la variable num1 con el tipo DB con el valor 200 (en decimal, pero en la memoria lo guarda en hexadecimal), guardándolo en hexadecimal en la dirección de celda 1001H

NUM2 DB -1: asigno el nombre de la variable num2 con el tipo DB con el valor -1 (en decimal, pero en la celda de memoria lo guarda en hexadecimal) guardándolo en hexadecimal en la dirección de celda 1002H

BITS0 DB 01111111B: asigno el nombre de la variable BITS0 con el tipo DB con el valor 01111111B (en binario, este binario tiene que valer menor a 255 porque trabajamos con 1 byte/8 bits), guardándolo en hexadecimal en la dirección de celda 1003H

BITS1 DB 10101010B: asigno el nombre de la variable BITS1 con el tipo DB con el valor 10101010B (en binario, este binario tiene que valer menor a 255 porque trabajamos con 1 byte/8 bits) guardándolo en hexadecimal en la dirección de celda 1004H

ORG 2000H: Empiezo generalmente en dirección de memoria para definir instrucciones.

MOV AL, NUM0: copio el valor de num0 y lo muevo al registro de la parte baja de AX, es decir a AL, como el valor de num0 es 80H, es decir un valor que ocupa 1byte, copio ese 80 y lo mando a AL

ADD AL, AL: ahora sumo lo que vale en el registro AL + AL, es decir 80H + 80H, como 80h = 1000 0000, entonces seria 1000 0000 + 1000 0000 = 00000000 con Carry = 1, entonces el resultado que es todo cero lo guarda en AL como **00h**

INC NUM1: acá básicamente incremento en uno el valor de num1, si num1 vale 200, entonces 200 + 1 = 201, ese resultado queda guardado en num1 como 201

MOV BH, NUM1: acá copio el valor de num1 que es 201 y lo muevo a BH, es decir la parte alta de BX, entonces ahora en BH quedaría 201.

MOV BL, BH: acá copio el valor de BH que es 201 y lo muevo a la parte baja de BX, quedando también ahora BL = 201

DEC BL: acá básicamente decremento en 1 el valor de BL, como BL vale 201, entonces 201 – 1 = 200, entonces ahora BL pasaría a tener un valor en el registro de 200.

SUB BL, BH: acá básicamente resto el valor de BL con el valor de BH, como BL = 200 Y BH =201, entonces 200 – 201 = -1, el resultado se guarda en BL, quedaron ahora BL = -1 en hexadecimal que es FF**h**

MOV CH, BITS1: acá copio el valor de BITS1 que vale AAH y lo muevo a CH (parte alta de CX), quedando también ahora CH = AAh

AND CH, BITS0: acá multiplico el valor de CH que vale AAh por el valor de BITS0 que vale 7FH, multiplicamos y el resultado lo guardamos en el registro CH en hexadecimal (2A).

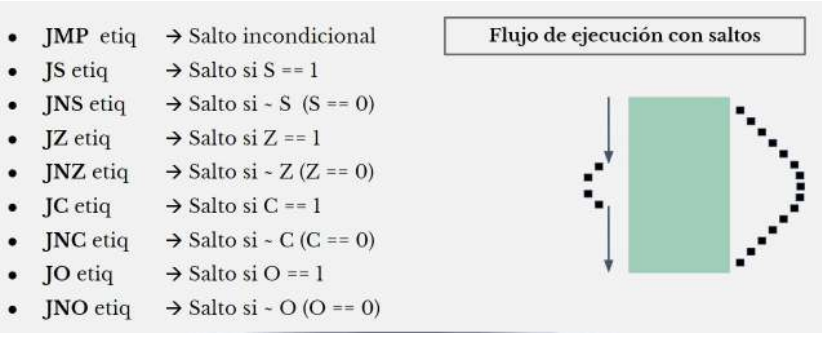
NOT BITS0: acá básicamente niego lo que es BITS0 que vale 7F**H** utilizando Ca1 y el resultado queda almacenado en BITS0

OR CH, BITS0: acá suma con OR para saber si luego hay Carry, entonces sumo CH que vale 2A, y BITS0 que vale 80H, quedando 2A + 80H, el resultado lo almaceno en CH (AA)

5) Saltos

Los saltos permiten ir a un determinado punto del código indicado por una etiqueta.

Las sentencias de saltos son las siguientes. Las condiciones se **basan sobre los Flags**



Ejemplo:

```
ORG 1000H
    INI   DB  0
    FIN   DB  15
ORG 2000H
    MOV  AL, INI
    MOV  AH, FIN
    SUMA:  INC  AL
        CMP  AL, AH
        JNZ  SUMA
HLT
END
```

ORG 2000H: Dirección de la memoria en donde quiero comenzar a poner las instrucciones.

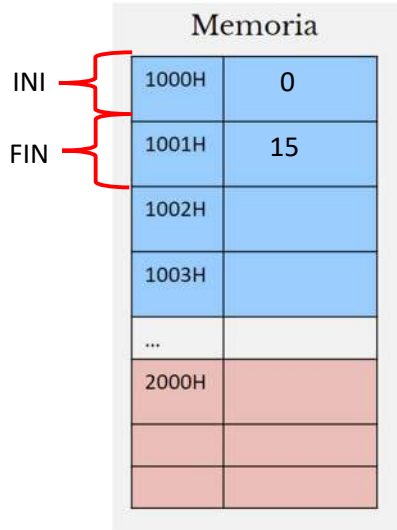
MOV, AL, INI: Copio el valor de "INI" que es 0, y lo muevo a AL (parte baja del registro AX).

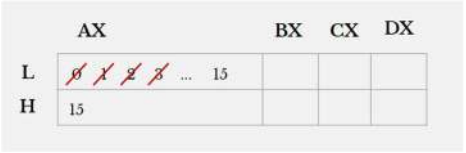
MOV AH, FIN: Copio el valor de "FIN" que es 15 y lo muevo a AH (parte alta del registro AX).

SUMA: INC AL: suma será el lazo en este caso, ya que luego veremos que JNZ "SUMA", hasta ahora es el nombre a una etiqueta, "INC AL" acá incremento AL en una unidad, es decir AL + 1.

CMP AL, AH: Comparo AL con AH, es decir resto AL y AH, (acuérdense que cuando hago CMP solo comparo, no guardo el resultado en ningún lado).

JNZ SUMA: JNZ significa que "Salta si el ultimo valor calculado no es cero, es decir si Z= 0" ¿hacia a donde salta? Hacia la etiqueta "SUMA", y cuando nos dice el "si el ultimo valor calculado no es cero" se refiere a la operación anterior que fue "CMP" AL, AH, entonces JNZ salta si "SUMA" solo si el resultado de hacer CMP, AL, AH no es cero, entonces salta hacia "SUMA" y vuelve a ejecutar otra vez a ejecutar las instrucciones de "SUMA" hacia adelante, se va a repetir hasta que el resultado de ese CMP sea cero, es decir hasta que Flag Z= 1. .

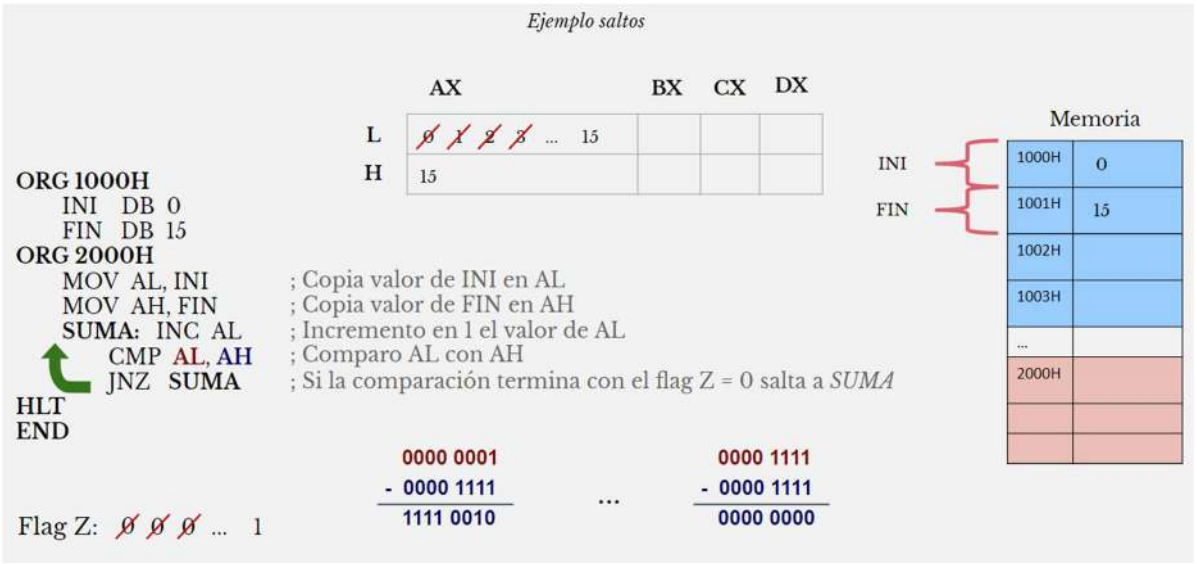




En un principio AL= 0, pero luego va aumentando en 1 por la instrucción "INC AL", pero como tenemos la condición de "JNZ SUMA", entonces se va a repetir tantas veces hasta que cuando haga CMP AL, AH y me dé Flags= 1, como vemos en la imagen **A)** en este caso se repitió 14 veces el laxo "SUMA", como

A

Así quedaría la imagen completa

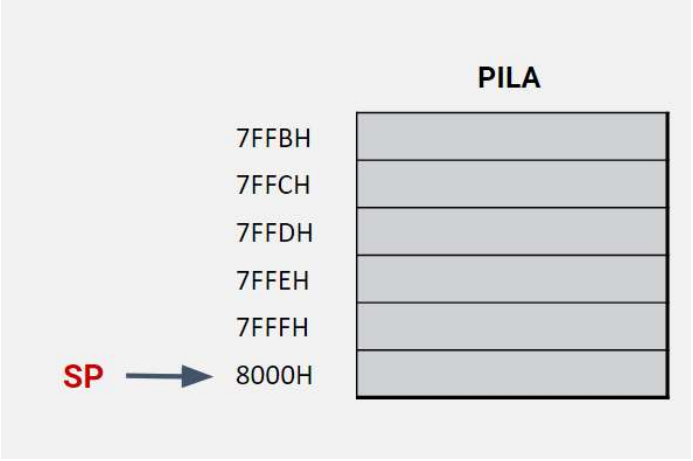


PRACTICA 6

TEMA: PILA (También llamado Stack)

La Pila es un **sector de la memoria**, en donde el acceso a este tipo de memoria es del tipo **LIFO**, ¿Qué carajos es LIFO? "Last in first out", o sea lo ultimo que entro es lo primero que va a salir de la pila.

Para usar la Pila se requiere un **Registro Especial** que se llama **Puntero de Pila: SP** (que es inglés es Stack Pointer) el SP se encuentra implícitamente en el tope de la Pila, nosotros no la tenemos que inicializar ni nada por el estilo.



Entonces la Pila es un sector de la memoria, la Pila (SP) en este caso arranca en 8000H, en otros casos arranca en 4000H.
Y **tenemos 2 instrucciones para trabajar con la Pila,** si queremos poner datos usamos la instrucción "PUSH" más el dato que queremos poner en la pila, y si queremos sacar datos y guardarlo en un registro usamos la instrucción "POP" más el destino a donde lo queremos guardar.

Tanto "POP" como "PUSH" se tienen que manejar con registros enteros (2 bytes/16 bits), es decir por ejemplo yo no puedo hacer "PUSH AL o PUSH DH", si o si tiene que ser "PUSH AX, BX, CX, DX", Y lo mismo con "POP" solo con registros enteros.

PUSH

La operación "PUSH" mete un dato (16 bits) en la pila, es decir guarda un dato en la pila.

Los pasos que se ejecutan internamente cuando hacemos "PUSH" son:

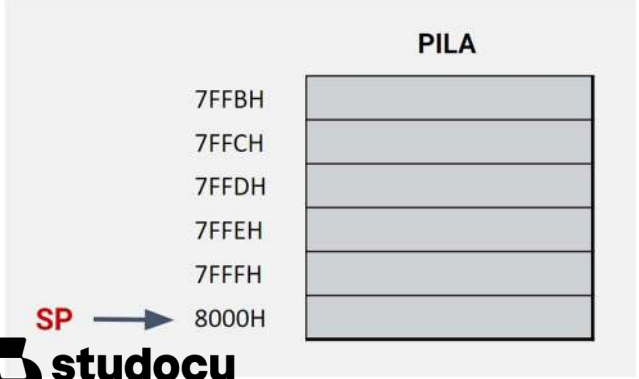
1) Decrementa en 1 SP
2) Inserta parte alta del dato
3) Decrementa en 1 SP
4) Inserta parte baja del dato

Veamos un ejemplo

MOV AX, 1234H: Copia el valor 1234H y lo mueve a AX
MOV CX, 96FCH: Copia el valor 96FCH y lo mueve a CX
PUSH AX: Guarda AX en la Pila (meto AX en la Pila)
PUSH CX: Guarda AX en la Pila (meto AX en la Pila)

	AX	CX
L	34	FC
H	12	96

inicio

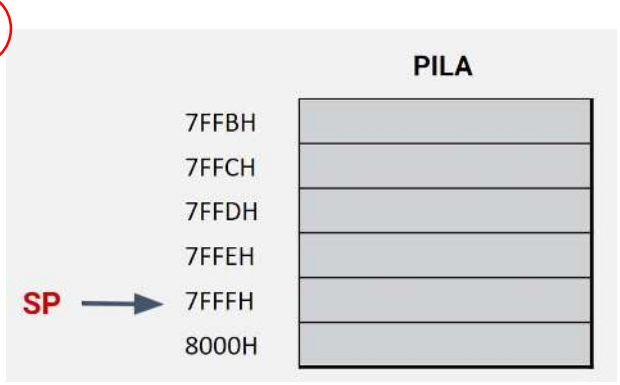


Analicemos bien lo que sucede con “PUSH AX”

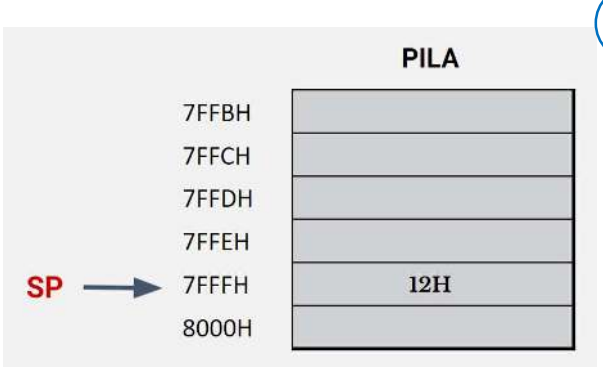
PUSH AX: como vimos anteriormente cuando uso “PUSH” se ejecutan 4 pasos: *decremento en 1 SP, inserta parte alta del dato, decrementa en 1 SP, nuevamente inserta parte baja del dato*.

Entonces sabiendo que el SP está en “8000H” **(inicio)** (en este caso), y ejecutando los pasos nos dice primero: *decremento en 1 SP*, entonces si comenzamos en "8000H” y “*decrementamos en 1 SP*”, me muevo 1 hacia arriba y ahora el SP va a estar en la posición “7FFFH” **(A)** , luego nos dice que “*inserta parte alta del dato*”, entonces en la posición del “SP = 7FFFH”, ahí colocamos la parte alta de “AX”, es decir “AH” que vale 12, entonces ahí coloco 12h **(B)**.

A

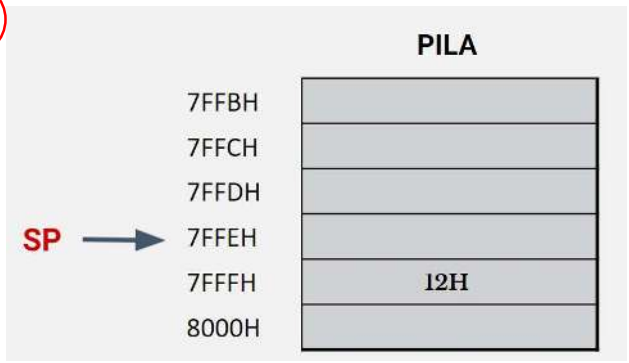


B

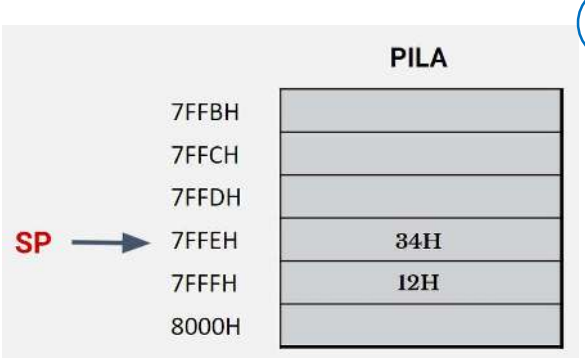


ahora seguimos el siguiente paso que es otra vez “*decrementa en 1 SP*”, me muevo 1 hacia arriba y el SP ahora está en la posición “7FFEH” **(C),** luego el siguiente paso es “inserta parte baja del dato”, entonces en la posición que estamos ahora colocamos la parte baja del dato “AX”, que “AL”, como “AL” vale 34, entonces ahí coloco 34h **(D).**

C



D

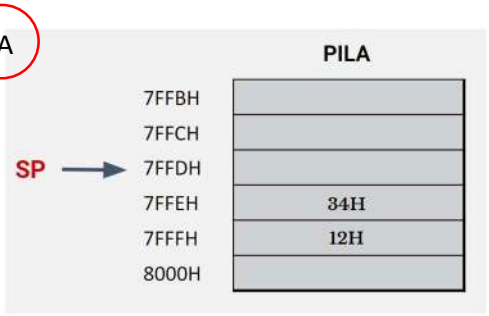


Listo, ahora hacemos lo mismo con “PUSH CX”, para no repetir toda la explicación les mostramos la foto a continuación.

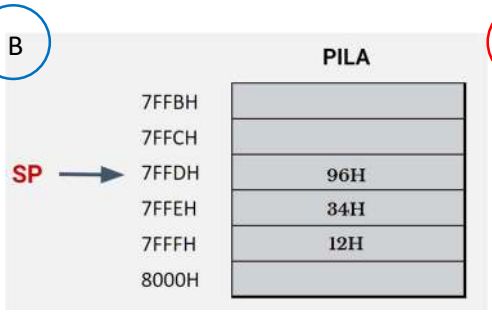
PUSH CX: de la posición “7FFEH” que era la última posición en donde había nos habíamos movido, de ahí ejecuto los paso otra vez de “PUSH”

1) **Decremento en 1 SP**: Entonces ahora estoy en “7FFDH” **(A)**
2) **Inserto parte alta del dato:** Entonces si trabajo con “CX”, la parte alta es “CH”, que vale “96h”, entonces en “7FFDH” queda el valor 96h. **(B)**
3) **Decremento en 1 SP**: Entonces ahora estoy en “7FFCH”. **(C)**
4) **Inserto parte baja del dato:** Entonces si trabajo con “CX”, la parte alta es “CL”, que vale “FCH”, entonces en “7FFCH” queda el valor “FCH”. **(D)**

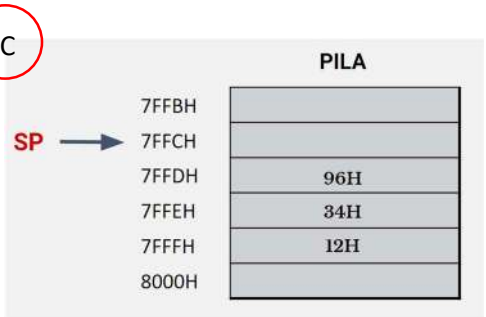
A



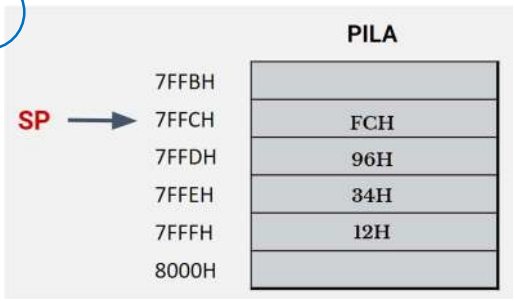
B



C



D



POP

La operación "POP" desapila un dato (de 16 bits), es decir saca un dato de la pila y lo guardo en un registro.

Los pasos que se ejecutan internamente cuando hacemos un POP son:

1) Guarda parte baja del dato en la parte baja del registro
2) Incrementa en 1 SP
3) Guarda parte alta del dato en la parte alta del registro
4) Incrementa en 1 SP nuevamente

Por ejemplo, siguiendo el ejemplo anterior:

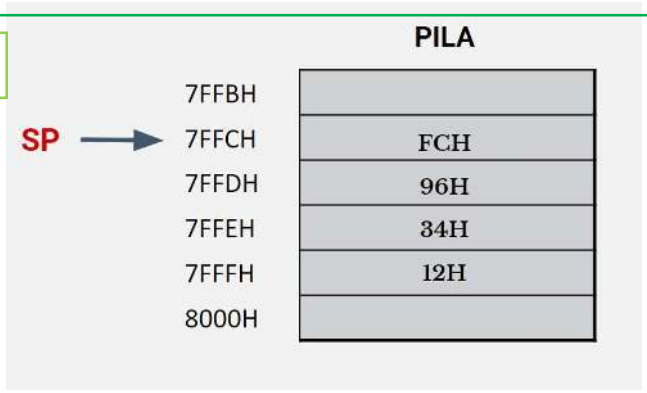
MOV AX, 1234H: Copia el valor 1234H y lo mueve a AX
MOV CX, 96FCH: Copia el valor 96FCH y lo mueve a CX
PUSH AX: Guarda AX en la Pila (meto AX en la Pila)
PUSH CX: Guarda AX en la Pila (meto AX en la Pila)

POP AX: Desapilo un elemento y lo guardo en AX
POP CX: Desapilo un elemento y lo guardo en CX

	AX	CX
L	34	FC
H	12	96

Donde había quedado la ultima vez el "SP" tras ejecutar "PUSH CX"

INICIO

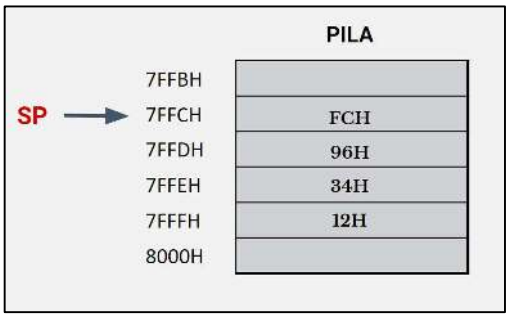


Analizaremos que sucede con "POP AX"

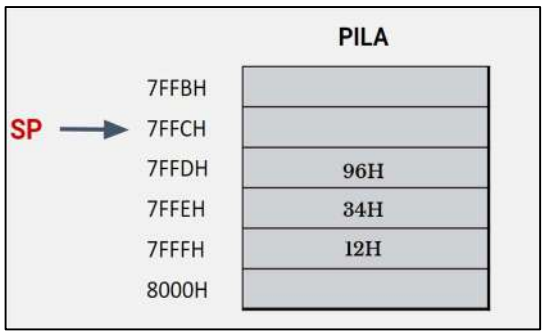
"**POP AX**": sabiendo que cuando uso "POP" tengo que ejecutar los pasos mencionados anteriormente, y también tenemos que saber en qué posición del SP estamos parados ahora mismo, en este caso SP está en "7FFCH" tras ejecutar "PUSH CX" **(INICIO)**. Entonces estando en "7FFCH" ejecutamos los pasos

1) **Guarda parte baja del dato en la parte baja del registro**: Si estamos en SP "7FFCH" y vemos que vale "FCH", entonces no pide que guardemos ese dato en la parte baja del registro, en este caso no dice "POP AX" entonces la parte baja de "AX" es "AL" que vale "FCH" y lo guardamos en "AL", entonces si vemos en el registro "AL", vemos que vale 34h, pero tras ejecutar "POP AX", ahora "AL" va a valer "FCH" y no "34h". **(A)**
2) **Incrementa en 1 SP:** De la posición en donde estamos "7FFCH", incrementamos en 1 SP, es decir bajo hacia abajo en 1, quedándonos en la posición del SP en "7FFDH". **(B)**
3) **Guarda parte alta del dato en la parte alta del registro**: De la posición en donde estamos "7FFDH", vemos que la Pila vale "96H", entonces ese valor de la Pila lo guardamos en la parte alta del Registro "AX", es decir en "AL", si vemos en el Registro "AL" vemos que ahora vale 34h, pero tras guardar en la parte alta del registro AX, ahora va a quedar el valor de la Pila en la posición "7FFDH" que vale "96H", entonces ahora AL queda en "96H". **(C)**
4) **Incrementa en 1 SP nuevamente**: De la posición en donde estamos "7FFDH", incrementamos en 1 SP, es decir bajo hacia abajo en 1, quedándonos en la posición del SP en "7FFEH".**(D)**

INICIO

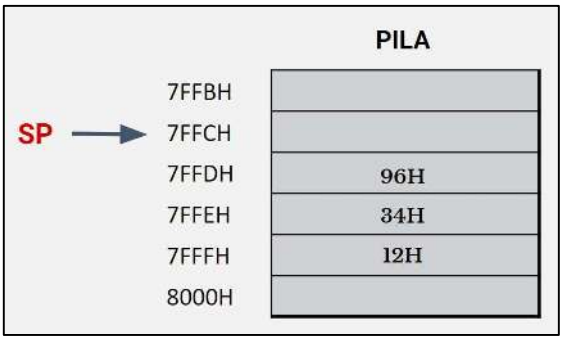


A

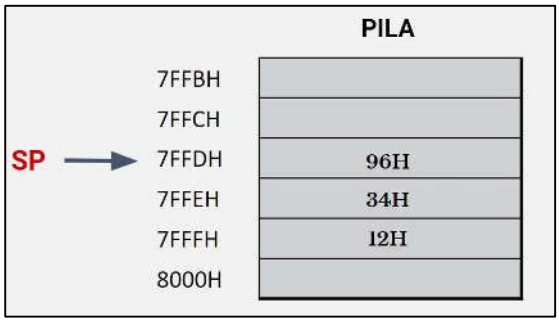


A

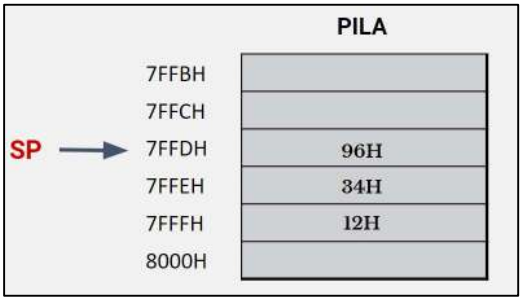
	AX	CX
L	~~34~~ FC	FC
H	12	96



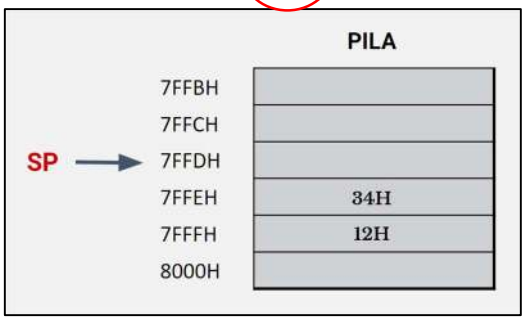
B



C

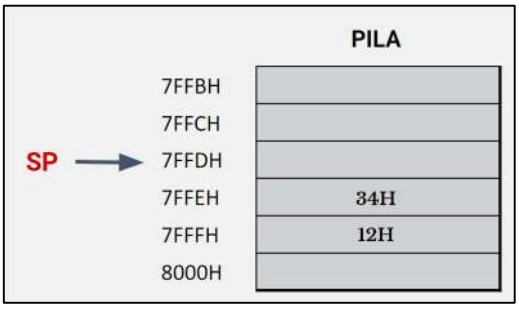


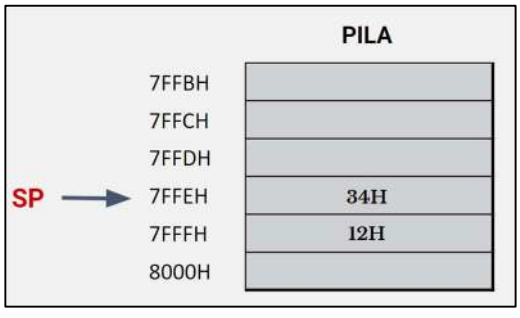
C



	AX	CX
L	~~34~~ FC	FC
H	~~12~~ 96	96

D





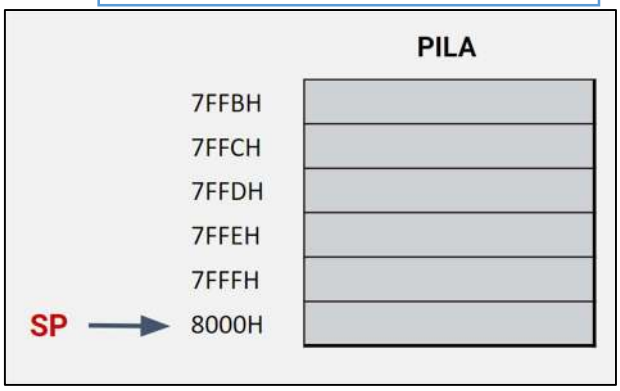
Ahora que hacemos lo mismo con “POP CX”

POP CX: de la posición “7FFEH” que era la última posición en donde había nos habíamos movido, de ahí ejecuto los paso otra vez de “POP”

1) **Guarda parte baja del dato en la parte baja del registro**: Entonces si estoy en la posición “7FFEH”, y vemos que vale “34H”, entonces ese valor lo guardo en la parte baja del Registro CX, que es “CL”, antes “CL” Valia FC , ahora tras guardar el nuevo valor quedaría “CL” = FCH
2) **Incrementa en 1 SP**: Si estoy en “7FFEH” y ahora nos pide incremente en 1 SP, ahora estamos posición del SP en “7FFFH” **(B)**
3) **Guarda parte alta del dato en la parte alta del registro**: Entonces si estoy en la posición “7FFFH”, y vemos que vale “12H”, entonces ese valor lo guardo en la parte alta del Registro CX, que es “CH”, antes “CH” Valia 96h , ahora tras guardar el nuevo valor quedaría “CH” = 12H.
4) **Incrementa en 1 SP nuevamente:** Si estoy en “7FFFH” y ahora nos pide incremente en 1 SP, ahora estamos posición del SP en “8000H””.

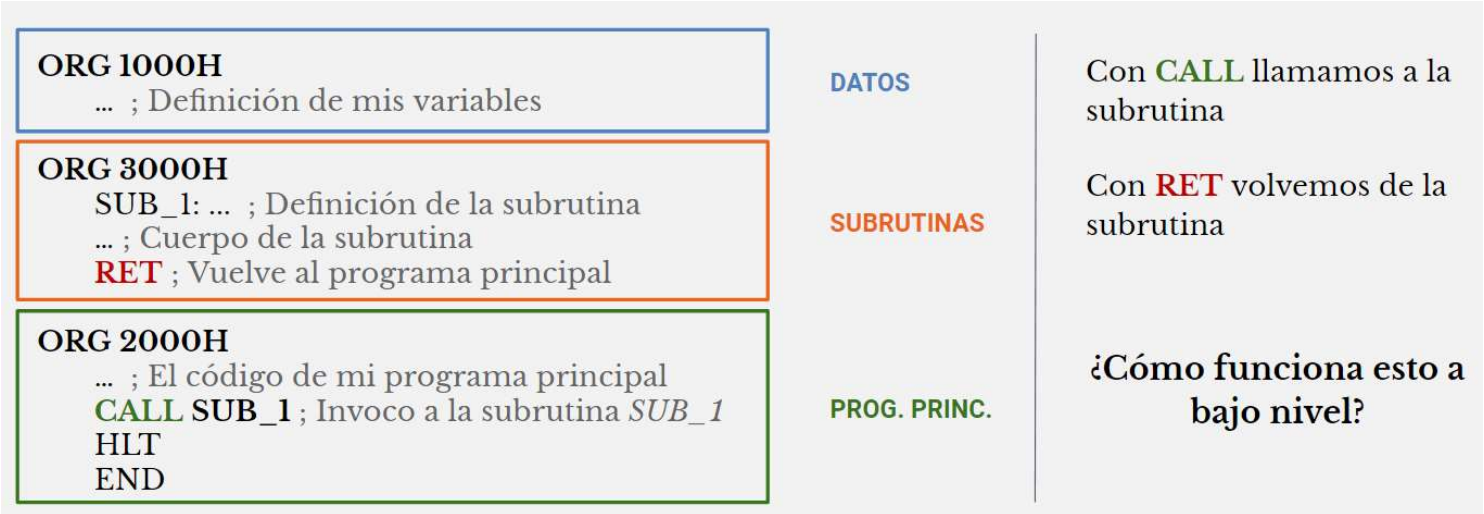
Así quedaría al final el Registro

Así quedaría la Pila y la posición del SP



TEMA: SUBRUTINAS : PROGRAMACION MODULAR EN ASSEMBLER

Concepto: Al igual que otros lenguajes, Assembler también permite programar de manera modular. Las subrutinas son similares a los procedures o functions en Pascal.



Para poder entender como funcionan las subrutinas es importantísimo entender un registro especial, hablamos del Registro **IP (Instruction Pointer)**, el IP nos indica que instrucción es la que hay que ejecutar y por lo tanto que instrucción es la siguiente a ejecutar.

O sea, le dice a la maquina “flaco entra a dirección de memoria, eso es lo que tenes que ejecutar”, entonces va a la dirección de memoria, lee el código de instrucción “ ahhh esto es un MOV, ahh esto es una suma” y lo ejecuta.

Dicho registro se incrementa a la dirección de la próxima instrucción a ejecutar. Veamos más a fondo desde el programa.

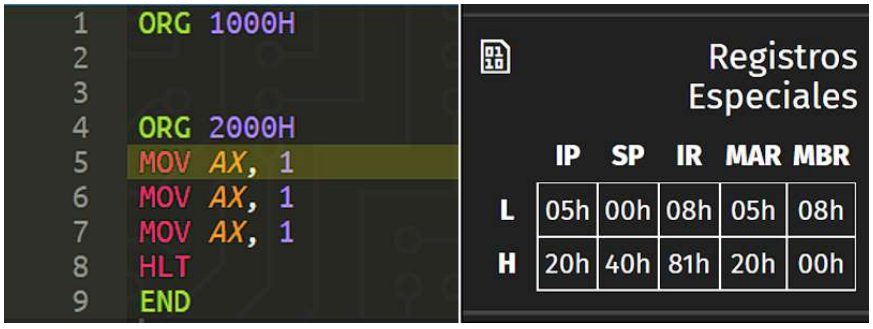
Veamos este ejemplo sencillo para entenderlo mejor. Primero que nada, se preguntaran **¿Por qué el IP arranca en 2000h?** (marcado con amarillo) esto es porque nosotros pusimos nuestro programa arranque ORG 2000h, que es donde quería empezar a poner las instrucciones.

¿Qué es lo que sucede en la memoria? ¿Qué significa el valor de estas 5 celdas a partir de 2000h? (marcado con verde), bueno resulta que lo que este marcado con verde es el primer “**MOV AX, 1**” que hicimos en la parte de 2000H.

Y ¿Qué significa el valor de las 5 siguientes celdas? (marcado con celeste), bueno resulta que es el segundo “MOV AX, 1” que pusimos como instrucción en la parte de 2000H.

Y ¿Qué significa el valor de las 5 siguientes celdas? (marcado con Rojo), bueno resulta que es el tercer “MOV AX, 1” que pusimos como instrucción en la parte de 2000H.

1) **Bueno ahora cuando se ejecuta el primer MOV AX, 1**, ¿Qué paso con el IP? Salto de 2000h hacia la 2005h



Como vemos en la imagen, tras ejecutar “MOV AX, 1” el IP se colocó en la posición 2005h de memoria. ¿Qué significa esto? Esto me está diciendo “Che flaco lo próximo que tenes que ejecutar esta en la posición de la memoria 2005h que es ni mas ni menos que el segundo “MOV AX, 1”.

2) **Y cuando ejecute el segundo “MOV AX, 1**” **¿Qué va a pasar con el IP?** Se va a mover de nuevo, de la posición 2005h hacia la posición de memoria 200Ah que es la posición de la siguiente instrucción, en este caso “MOV AX, 1”

3**) Ahora cuando se ejecute el tercer "MOV AX, 1" ¿Qué va a pasar con el IP?** Se va a mover de nuevo, de la posición 200Ah a la 200Fh, que es la posición de la siguiente instrucción, que es el HLT en este caso.

```
1   ORG 1000H
2
3
4   ORG 2000H
5   MOV AX, 1
6   MOV AX, 1
7   MOV AX, 1
8   HLT
9   END
```

Registros Especiales

	IP	SP	IR	MAR	MBR
L	0Fh	00h	08h	0Fh	08h
H	20h	40h	81h	20h	00h

Entonces con esto sabemos que el IP no lo manejamos nosotros, es decir no incrementamos ni decrementamos, sino que la hace automáticamente la computadora.

Ahora entendiendo esto, volvemos con el Tema "CALL", es decir con las subrutinas.

Dijimos que las subrutinas las poníamos en la posición 3000H (por convención, no es obligatorio), entonces si tenemos el siguiente programa

```
1   ORG 1000H
2
3   ORG 3000H
4   SUMA: ADD AX, BX
5   RET
6
7
8   ORG 2000H
9   MOV AX, 1
10  MOV BX, 3
11  CALL SUMA
12  HLT
13  END
```

Entonces primero "ORG 2000H", copio el 1 y lo muevo a AX, quedando AL= 01h y AH= 00h. Luego copio el 3 y lo muevo a BX, quedando BL = 03h y BH = 00h.

Luego ejecuta "CALL SUMA", o sea saltamos hacia la parte de 3000H que es donde está la subrutina "SUMA" y ejecuto la instrucción "ADD AX, BX", suma y el resultado que es 04h lo guarda en el registro AX, luego está la instrucción "RET" que me vuelve a la instrucción que seguía luego de "CALL SUMA" que es la instrucción HLT y END, terminando así el programa.

Registros de Propósito General

	AX	BX	CX	DX
L	01h	03h	00h	00h
H	00h	00h	00h	00h

Registros de Propósito General

	AX	BX	CX	DX
L	04h	03h	00h	00h
H	00h	00h	00h	00h

Ahora sabiendo lo que sucede con la instrucción "Call" (subrutina), veremos que es lo que le sucede al IP cuando usamos "Call "y "Ret".

Entonces cuando ejecutamos la sentencia **CALL** ocurre lo siguiente:

1) **Guarda la Dirección de la siguiente instrucción en la Pila (PUSH IP)**, esto es para después poder volver, por ejemplo, la maquina me dice "che la siguiente es esta, pero eh para para, me está diciendo que vaya a una subrutina (me tengo que ir a otra parte), a bueno entonces guardemos partida, vos que estabas parado acá lo guardo en la Pila.
2) **Asigna la dirección de la subrutina a IP para poder ejecutarla,** acá lo que hace es cambiar el registro IP, para que la maquina ahora lo que ejecute sea el código de la subrutina y no el programa principal que es donde estaba parado.
3) **Sigue la instrucción como siempre. Solo que ahora está en la dirección de la subrutina,** después acá sigue la instrucción como siempre, solamente que ahora el IP esta en la subrutina.

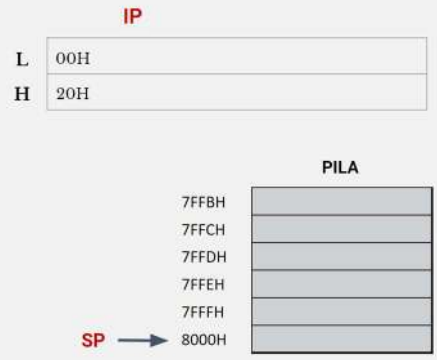
Entonces ahora una vez que termina el cuerpo de la subrutina se debe hacer uso de la sentencia **RET**. De esta manera el programa podrá continuar por donde estaba antes de entrar la subrutina. ¿Pero cómo funciona?

Cuando ejecutamos la sentencia RET ocurre lo siguiente:

1) **Desapila la instrucción que habíamos salvado en la pila y la guarda en IP (POP IP)**, acá lo que hace es sacar la "partida guardada" por así decirlo que habíamos tenido almacenado en la pila y lo pone en el IP
2) **¡Listo! Sigue la ejecución como siempre. Solo que ahora está en la dirección donde nos había quedamos,** y ahora continua la ejecución donde nos habíamos quedado.

Veamos un ejemplo, de que es lo que sucede con "Call" y "Ret" en el registro especial IP.

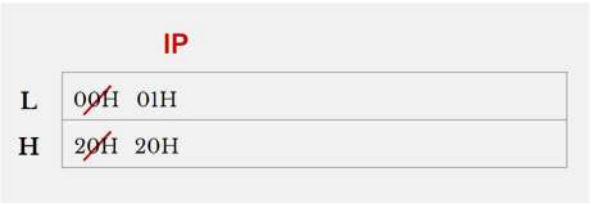
```
ORG 1000H
    ... ; Definición de mis variables

ORG 3000H
    SUB_1: ...   ; Inst. de subrutina 1
    ...          ; Inst. de subrutina 2
    RET          ; Vuelve al Prog. P.

ORG 2000H
    MOV AL, 1    ; Instrucción inútil
    CALL SUB_1   ; Invoco subrutina
    MOV AL, 2    ; Otra instrucción inútil
    HLT
    END
```



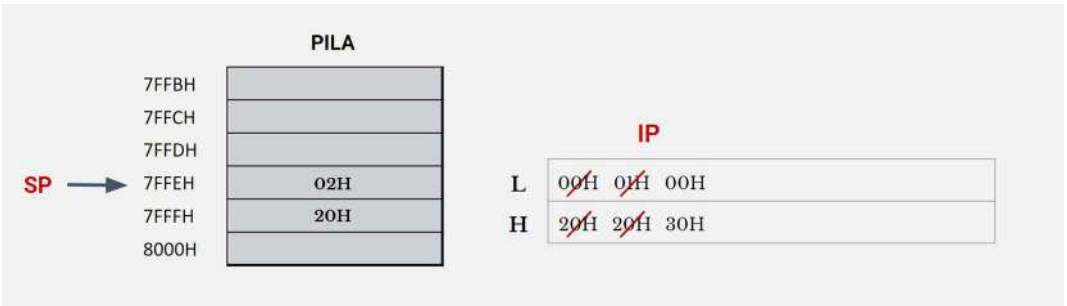
ORG 1000H, defino las variables que van en la memoria

ORG 2000H: parte principal del programa, este 2000H está indicándonos que el IP comenzara en 2000h
1) **MOV AL, 1**: entonces comienza el programa ejecutando en la línea 2000H con la instrucción "MOV AL, 1", una vez ejecutada la instrucción, el IP cambia de valor a 2001H, que es la próxima instrucción a ejecutar.

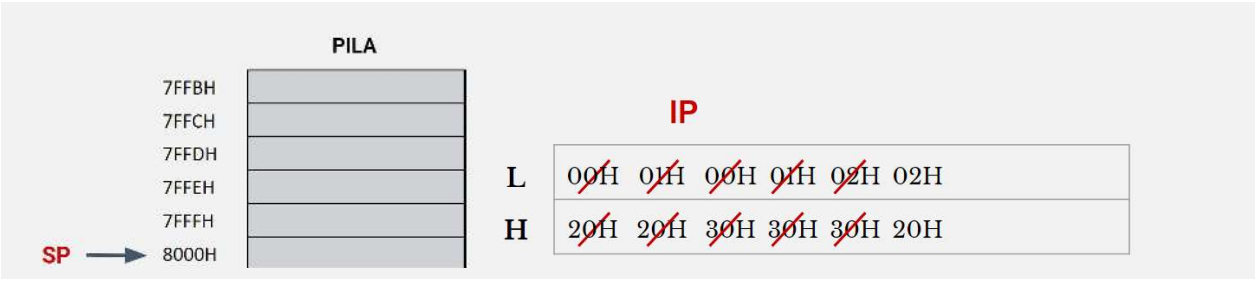
```
ORG 2000H
    MOV AL, 1    ; Instrucción inútil   <-
    CALL SUB_1   ; Invoco subrutina
    MOV AL, 2    ; Otra instrucción inútil
    HLT
    END
```



2) **CALL SUB_1**: entonces pasamos a IP 2001H y ejecutamos la instrucción "CALL SUB_1" y el IP cambia de vuelta, pero no a la dirección 2002H que sería "MOV AL, 2" porque tenemos tenes un "Call" que llama a SUB_1, ¿Y en que posición esta esa subrutina? En la 3000H, entonces el IP tiene que cambiar a 3000H, pero antes de cambiar a 3000H, lo que hace "Call" implícitamente es hacer un "PUSH IP", entonces ¿Qué dirección guarda en la Pila? Guardamos la dirección que sigue luego del "Call" que es la dirección del "MOV AL, 2" que esta ubicado con la posición 2002H, hago los pasos de "PUSH": *decremento 1 SP, guardo la parte alta, decremento 1 SP, guardo la parte baja*, una vez guardado los datos en la pila, procede a cambiar el IP en 3000H, que es donde esta la subrutina.



ORG 3000H: Acá es donde generalmente pongo la subrutina, entonces acá ejecuto la instrucción subrutina 1, y el IP cambia de 3000H a 3001H. Entonces ahora se ejecuta la subrutina 2 y el IP cambia a 3002H que es donde está el "RET", estando acá el IP cambia de vuelta, pero no hacia 3003H porque ya no hay ninguna subrutina más, sino que estando en "RET" tenemos que volver al programa principal. ¿Pero en qué dirección esta? Entonces en la posición "RET" el programa implícitamente hace un "**POP IP**" siguiendo los pasos: *saca la parte baja de la pila, incremento en 1 SP, saca la parte alta de la Pila, incremento en 1 SP*. Como el "POP" saca datos y lo guarda, en este caso saca el dato y lo guarda en el IP, quedando el IP ahora en 2002H



Como ahora el IP nos dice “Che flaco ejecuta la instrucción 2002H” que es donde habíamos quedado continuado antes de ejecutar el “Call”. El que esta en la posición 2002H no es ni mas ni menos que “MOV AL, 2”.

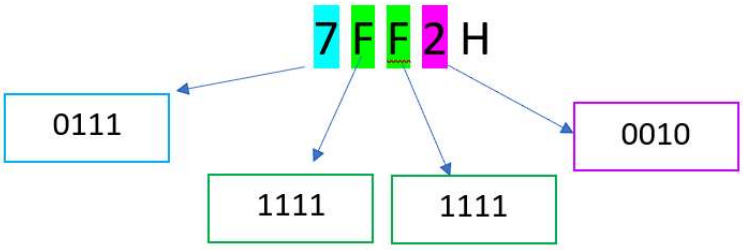
Pregunta de Parcial:

1) **Si el SP = 7FF2H. ¿Qué valor va a tener cuando se ejecute la sentencia “CALL”?** Respuesta: 7FF1H, porque hicimos SP – 2.

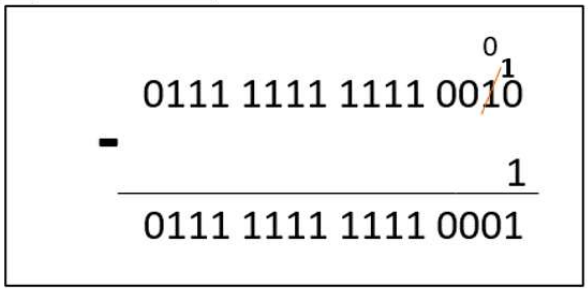
Explicación: Sí cuando ejecuto “Call” habíamos dicho que implícitamente se ejecutaba un “PUSH IP”, entonces si recordamos el “PUSH” *nos dice: decremento 1 SP, guardo la parte alta, decremento 1 SP, y guardo la parte baja*, Entonces sí sabemos que el SP decrementa 2 veces en total y nosotros queremos saber el valor del SP, entonces al SP en donde estamos le restamos 2 y el resultado es donde vamos a estar parados luego de ejecutar “CALL”.

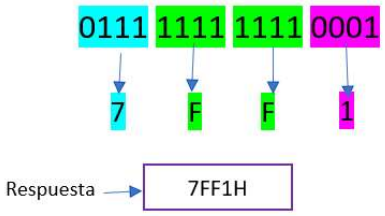
Entonces si tenemos 7FF2H y le restamos 2, para hacerlo más fácil, pasemos a binario el 7FF2H, para eso separamos cada uno de ellos y trabajamos con 4 bits, y luego le restamos 1. Entonces

Paso Hexadecimal a Binario



Resto 1 al Binario y lo paso a Hexadecimal





2) **Si el SP = 7FF2H. ¿Qué valor va a tener cuando se ejecute la sentencia “RET”?** Respuesta = 7FF4H, porque hicimos SP + 2-

Explicación: si se acuerdan cuando se ejecuta la sentencia “Ret”, implícitamente se ejecuta un “POP IP”, es decir acordémonos los pasos que hace el “POP*”: guardo la parte la parte baja, incremento 1 SP, guardo la parte alta, “incremento 1 SP*. Entonces como incrementa 2 veces, sumamos al 7FF2H un 2. No hace falta pasarlo a binario para sumarle porque el ultimo digito es un 2, es decir no esta en hexadecimal, por ende, le sumamos un 2 y quedaría “7FF4H”.

3) ¿En una SUBRUTINA puedo hacer PUSH dato?

Va a ejecutarlo, pero cuando llegue al “RET” que como sabemos hace implícitamente un “PUSH IP” para volver a donde está el programa principal, pero yo en la Pila ya tengo otro dato cargado que es ese “PUSH dato”, entonces cuando se ejecute “RET” el IP va a quedar el valor “dato”, y ahora como el IP nos dice que instrucciones siguientes debemos ejecutar, entonces como IP = “dato”, y como en la posición de la memoria “dato” hay basura, no se ni lo que hace, entonces se rompe el programa. Entonces si quiero usar un PUSH dentro de una subrutina luego tengo que usar si o si un “POP dato” para limpiar la basura que tenia