Explicación de práctica Nº 5

Lenguaje Assembly - 2da. parte

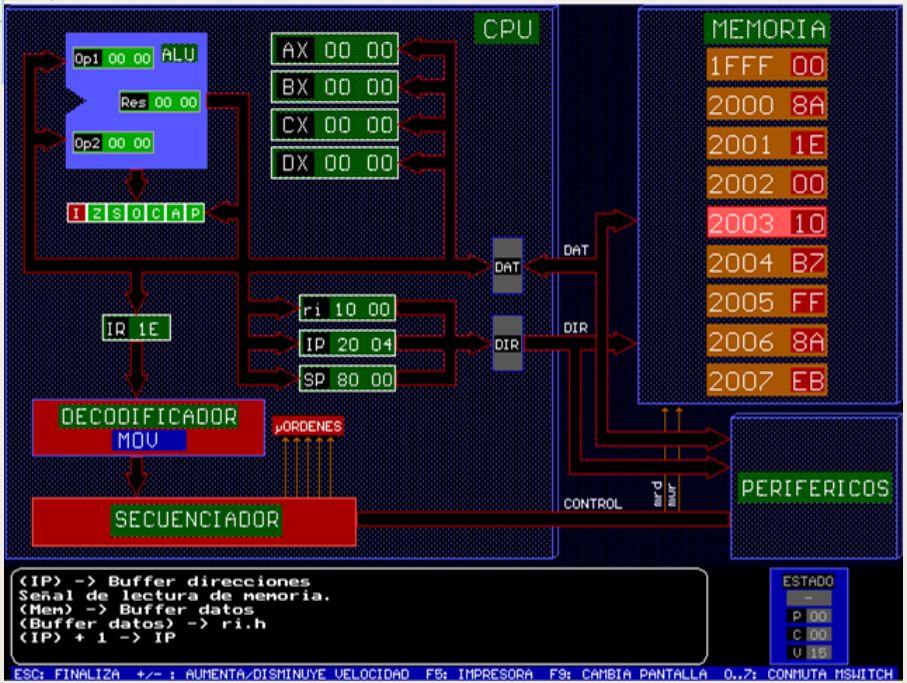
Ejercicio 1 (datos)

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

Ejercicio 1 (instrucciones)

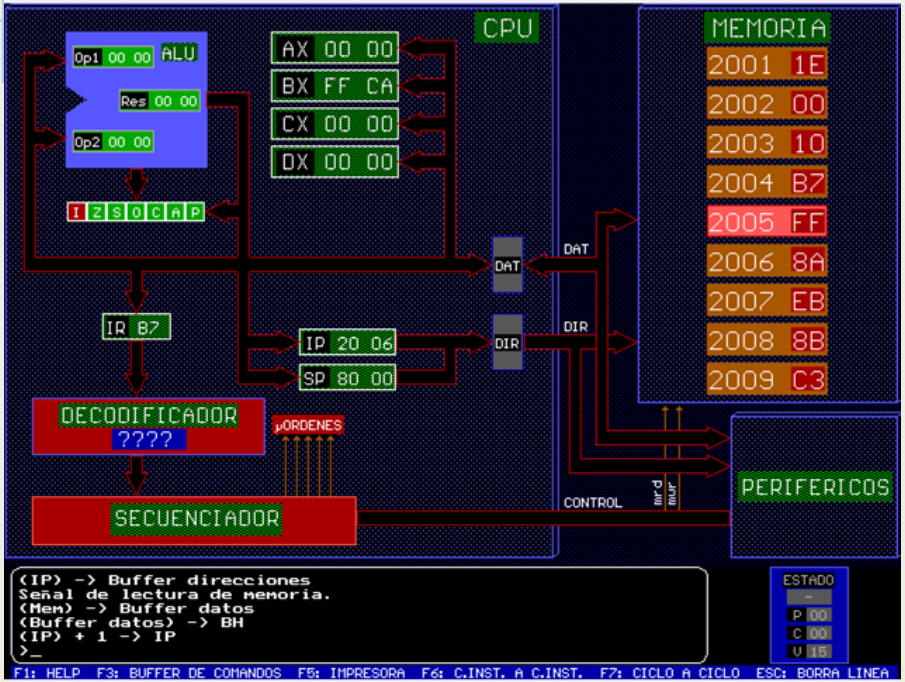
Dir.	Código máquina	Lín.	Código ensamble
		8	ORG 2000H
2000	8A 1E 00 10	9	MOV BL, NUM0
2004	B7 FF	10	MOV BH, 0FFH
2006	8A EB	11	MOV CH, BL
2008	8B C3	12	MOV AX, BX
200A	88 06 01 10	13	MOV NUM1, AL
200E	C7 06 02 10 34 12	14	MOV NUM2, 1234H
2014	BB 04 10	15	MOV BX, OFFSET NUM3
2017	8A 17	16	MOV DL, [BX]
2019	8B 07	17	MOV AX, [BX]
201B	BB 06 10	18	MOV BX, 1006H
201E	C7 07 06 10	19	MOV WORD PTR [BX], 1006H

Ej. 1: MOV BL, NUM0



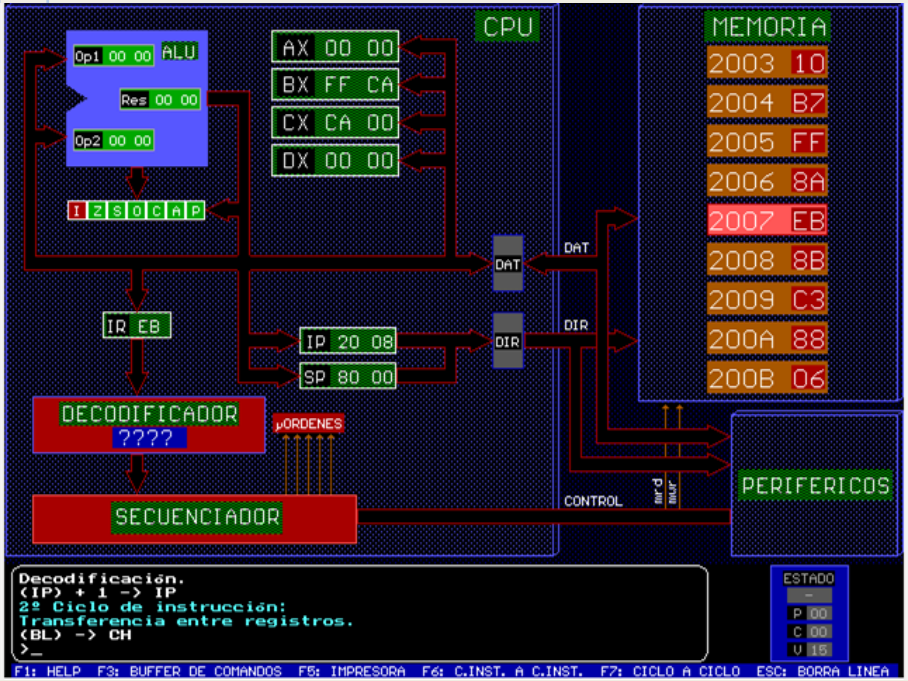
Origen	Memoria (NUM0:1000h)
Destino	Registro (BL)
Tamaño	8 bits (BL)
Modo	Directo
Dato	CAh
Resultado	BX = 00CAh

Ej. 1: MOV BH, 0FFH



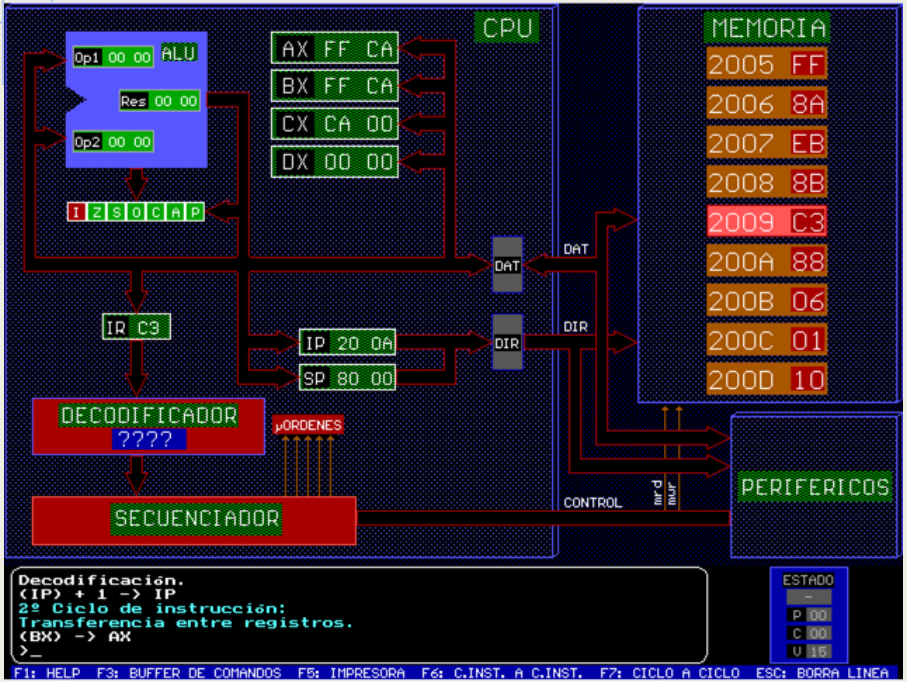
Origen	Memoria (Instrucción:2005h)
Destino	Registro (BH)
Tamaño	8 bits (BH)
Modo	Inmediato
Dato	FFh
Resultado	BX = FFCAh

Ej. 1: MOV CH, BL



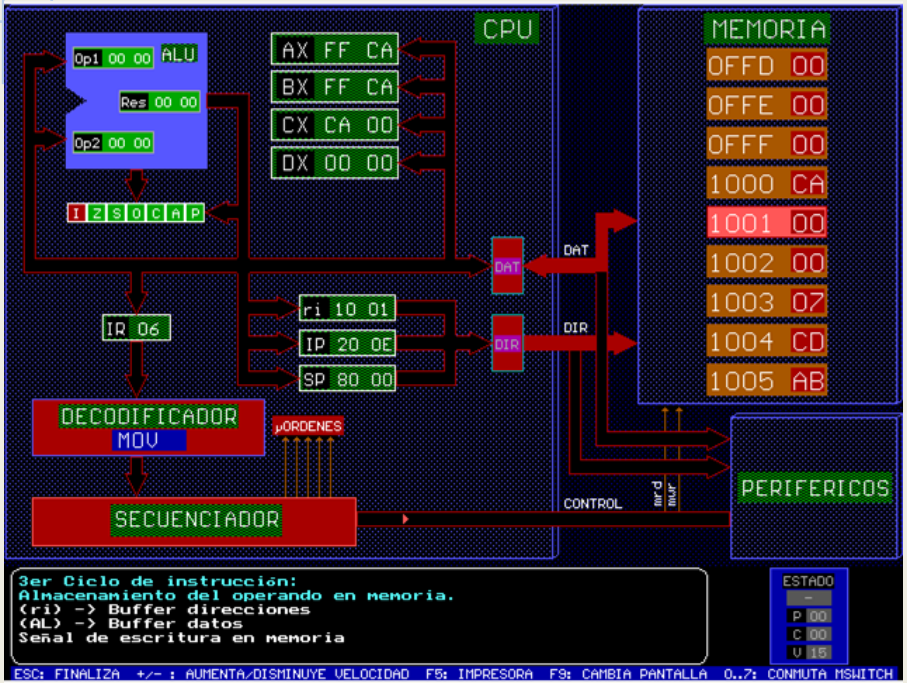
Origen	Registro (BL)
Destino	Registro (CH)
Tamaño	8 bits (CH)
Modo	Directo (registro)
Dato	CAh
Resultado	CX = CA00h

Ej. 1: MOV AX, BX



Origen	Registro (BX)
Destino	Registro (AX)
Tamaño	16 bits (AX)
Modo	Directo (registro)
Dato	FFCAh
Resultado	AX = FFCAh

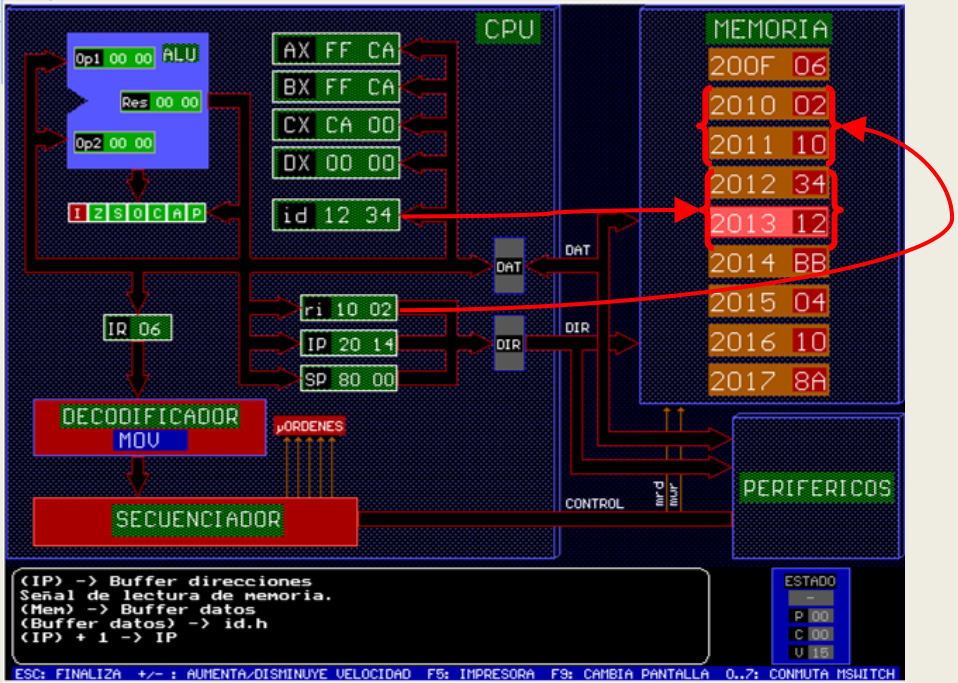
Ej. 1: MOV NUM1, AL



Origen	Registro (AL)
Destino	Memoria (NUM1:1001h)
Tamaño	8 bits (AL)
Modo	Directo
Dato	CAh
Resultado	NUM1 = CAh

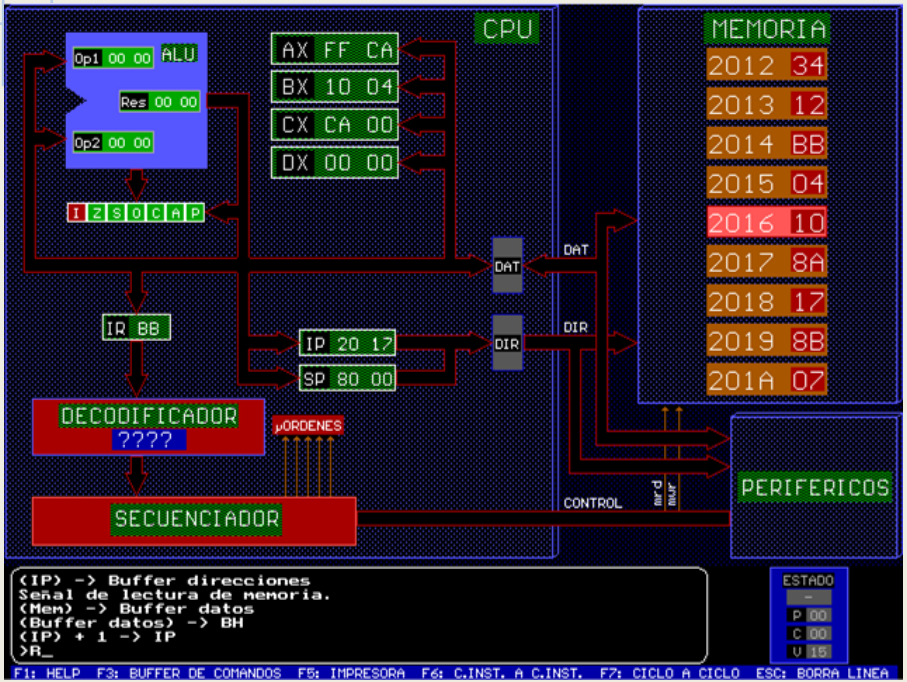
Nota: la imagen muestra el momento en el que el dato es transferido al bus de datos con la dirección 1001h puesta en el bus de direcciones desde el registro temporal RI.

Ej. 1: MOV NUM2, 1234H



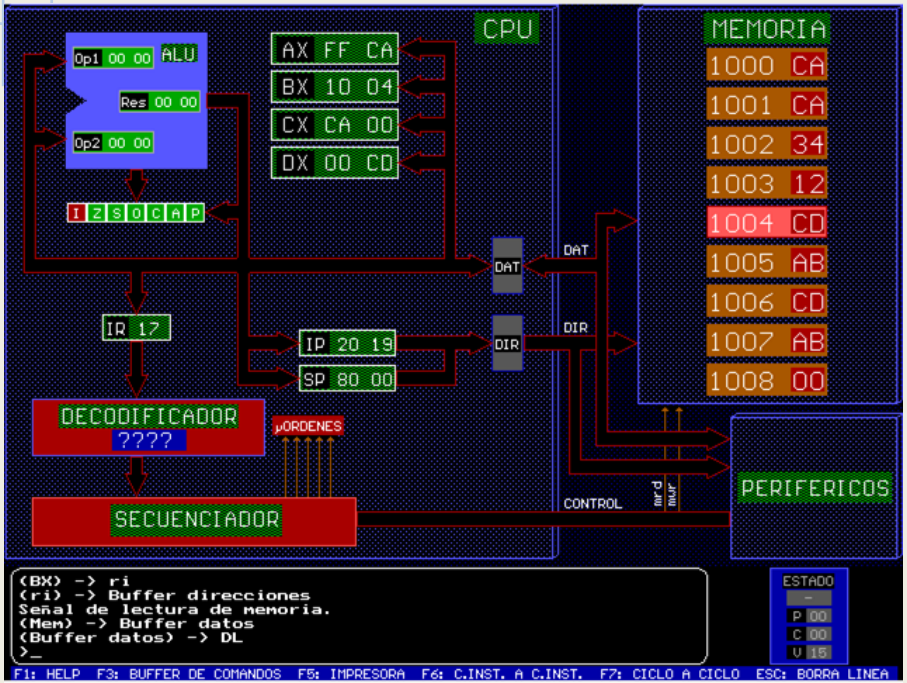
Origen	Memoria (Instrucción:2012h)
Destino	Memoria (NUM2:1002h)
Tamaño	16 bits (AX)
Modo	Inmediato
Dato	1234h
Resultado	NUM2 = 1234h

Ej. 1: MOV BX, OFFSET NUM3



Origen	Memoria (Instrucción:2015h)
Destino	Registro (BX)
Tamaño	16 bits (BX)
Modo	Inmediato
Dato	1004h
Resultado	BX = 1004h

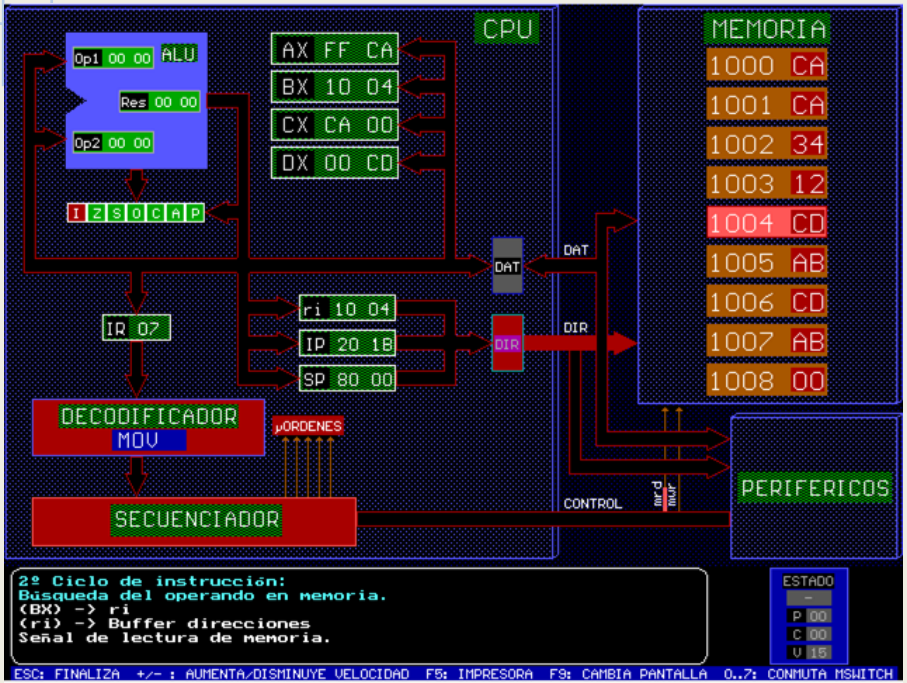
Ej. 1: MOV DL, [BX]



Origen	Memoria (*BX:1004h)
Destino	Registro (DL)
Tamaño	8 bits (DL)
Modo	Indirecto por registro (BX)
Dato	CDh
Resultado	DX = 00CDh

Nota: el contenido del registro BX (1004h) es transferido al registro temporal RI para poder direccionar a esa posición de memoria.

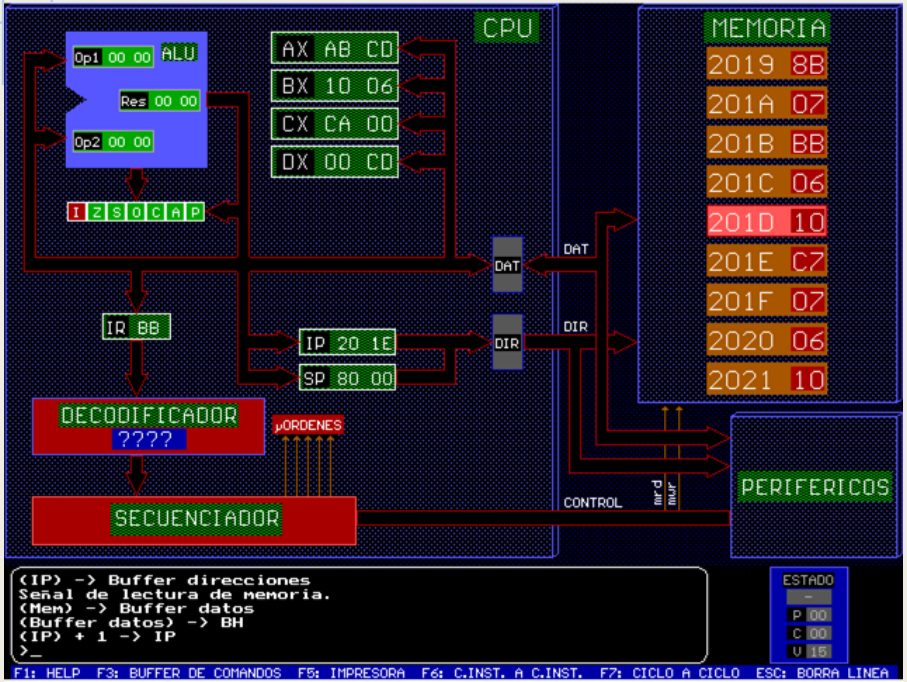
Ej. 1: MOV AX, [BX]



Origen	Memoria (*BX:1004h)
Destino	Registro (AX)
Tamaño	16 bits (AX)
Modo	Indirecto por registro (BX)
Dato	ABCDh
Resultado	AX = ABCDh

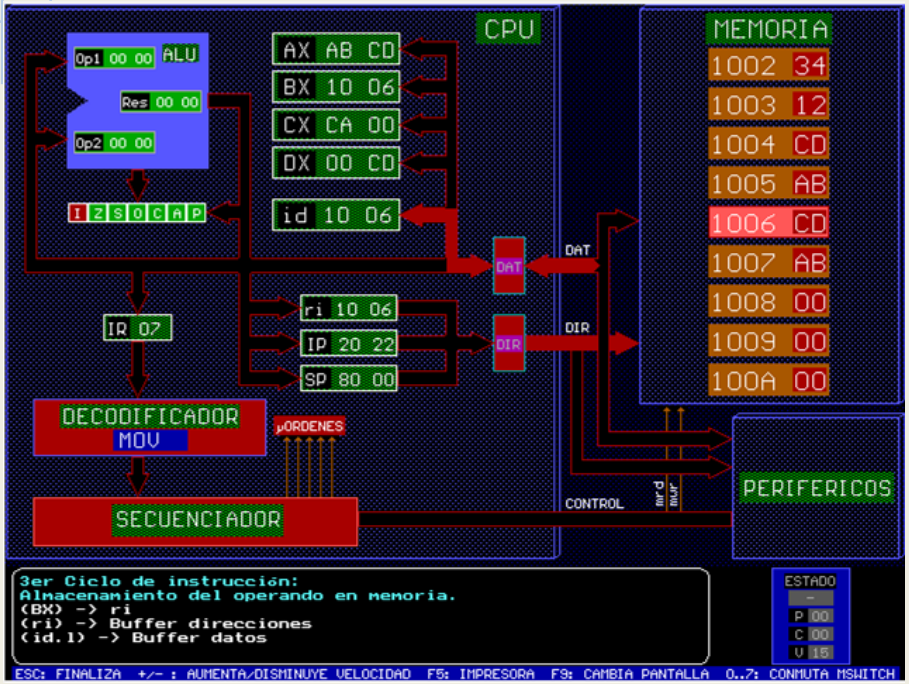
Nota: la imagen muestra el momento en que se direcciona la memoria con el registro temporal RI (el dato aún no fue almacenado en AX).

Ej. 1: MOV BX, 1006H



Origen	Memoria (Instrucción:201Ch)
Destino	Registro (BX)
Tamaño	16 bits (BX)
Modo	Inmediato
Dato	1006h
Resultado	BX = 1006h

Ej. 1: MOV WORD PTR [BX], 1006H



Origen	Memoria (Instrucción:2020h)
Destino	Memoria (*BX:1006h)
Tamaño	16 bits (WORD PTR)
Modo	Inmediato
Dato	1006h
Resultado	NUM4(1006h) = 1006h

ID tiene el valor 1006h leído desde la instrucción (dato inmediato).
RI tiene el valor 1006h transferido desde BX (direccionamiento indirecto).

Ejercicio 2

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;
```

A en AL, B en BL y C en CL.

1. Identificar la condición como el estado de un FLAG a partir de una operación.
2. Determinar qué instrucción de salto conviene utilizar.
3. Expresar las instrucciones simples en lenguaje assembly y etiquetar los bloques.
4. Etiquetar la salida del condicional.

Ejercicio 2 (cont.)

Identificar la condición como el estado de un FLAG a partir de una operación

- La condición se debe evaluar como una operación aritmética o una operación lógica de las disponibles.
- Tenemos que comparar dos valores.

  CMP AL, BL
- Observamos lo que pasa con los FLAGS en los distintos casos límite:

  Si AL = 1 y BL = 2, ZSOC = 0100 (cuando AL < BL)

  Si AL = 2 y BL = 2, ZSOC = 1000 (cuando AL = BL)

  Si AL = 2 y BL = 1, ZSOC = 0000 (cuando AL > BL)
- Esta resolución no siempre es tan trivial.

  ¿Qué pasa si la condición fuera A>=B?

Ejercicio 2 (cont.)

Determinar qué instrucción de salto conviene utilizar.
El flag a utilizar es el S (signo).
Las instrucciones candidatas son JS y JNS.

- JS: el bit de signo vale 1, AL<BL es verdadero, salta a THEN.

```
CMP AL, BL
JS then
  ;instrucciones del bloque ELSE
then:
  ;instrucciones del bloque THEN
```

- JNS: el bit de signo vale 0, AL<BL es falso, salta a ELSE.

```
CMP AL, BL
JNS else
  ;instrucciones del bloque THEN
else:
  ;instrucciones del bloque ELSE
```

Ejercicio 2 (cont.)

Expresar las instrucciones simples en lenguaje assembly y etiquetarlas

C := A (then)
then: MOV CL, AL

C := B (else)
else: MOV CL, BL

```
CMP AL, BL
JNS else          ;es la forma natural del if
MOV CL, AL
else: MOV CL, BL
```

Este código no funciona correctamente, ¿por qué?

Ejercicio 2 (cont.)

Etiquetar la salida del condicional

```
CMP AL, BL
JNS else
  MOV CL, AL
  JMP fin
else: MOV CL, BL
fin: HLT
END
```

- El procesador ejecuta las instrucciones secuencialmente.
- Las instrucciones de salto modifican el registro PC para alterar el orden normal.
- Si no se ejecuta un salto luego del primer bloque de instrucciones, se sigue ejecutando en secuencia el *else*.

Ejercicio 2 (cont.: A<=B)

Si la condición fuese A<=B

- Tenemos que comparar dos valores.

  CMP AL, BL

- Observamos lo que pasa con los FLAGS en los distintos casos límite:

  Si AL = 1 y BL = 2, ZSOC = 0100 (cuando AL < BL)

  Si AL = 2 y BL = 2, ZSOC = 1000 (cuando AL = BL)

  Si AL = 2 y BL = 1, ZSOC = 0000 (cuando AL > BL)

- La condición implica evaluar dos FLAGS (o bien S=0 o bien Z=0).
- Se pueden encadenar instrucciones de salto por los bits que valen 1:

```
JS then
JZ then
  else: JMP fin
then: hlt
```

Ejercicio 2 (cont.: A<=B)

Si la condición fuese A<=B

- Una solución más elegante y eficiente es invertir la condición.
  A<=B es *falso* cuando A > B
- Tenemos que comparar dos valores.
  CMP BL, AL
- Observamos lo que pasa con los FLAGS en los distintos casos límite:
  Si AL = 1 y BL = 2, ZSOC = 0000 (cuando AL < BL)
  Si AL = 2 y BL = 2, ZSOC = 1000 (cuando AL = BL)
  Si AL = 2 y BL = 1, ZSOC = 0100 (cuando AL > BL)
- Ahora la condición es verdadera cuando el bit S=0.

Ejercicio 2 (cont.: A<=B)

Dir.	Cód. máquina	Lín.	Cód. ensamble	
		1	ORG 2000h	
2000	3A D8	2	CMP BL, AL	
2002	79 03	3	JNS else	;S=0 cuando AL>BL
2004	E9 08 20	4	then: JMP fin	
2007	90	5	else: NOP	;instrucciones del else
2008	F4	6	HLT	
		7	END	

Pila

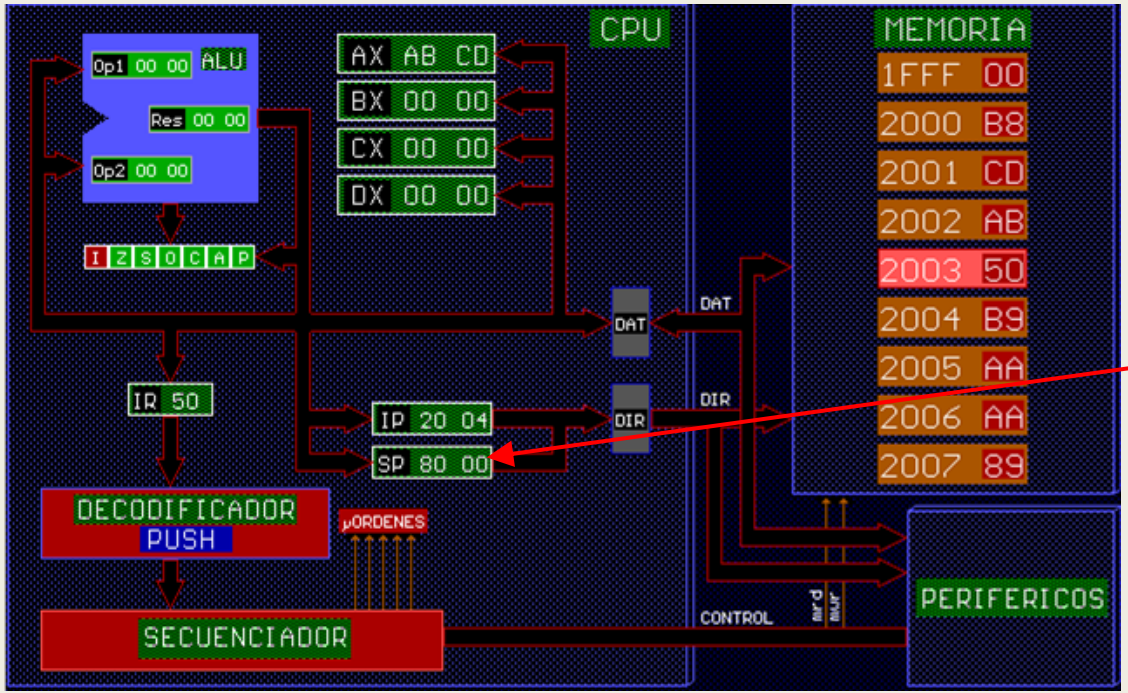
La pila es un área de la memoria que:

- comienza en una dirección fija (en nuestro caso: 7FFFh)
- como estructura, tamaño variable (crece a medida que *apilamos* datos y decrece a medida que los *desapilamos*)
- está apuntada por un registro (SP: stack pointer) que se modifica automáticamente en cada operación (se inicializa en 8000h)
- se corresponde a una estructura LIFO (last in - first out) -> SP apunta al último dato apilado

Pila (instrucciones)

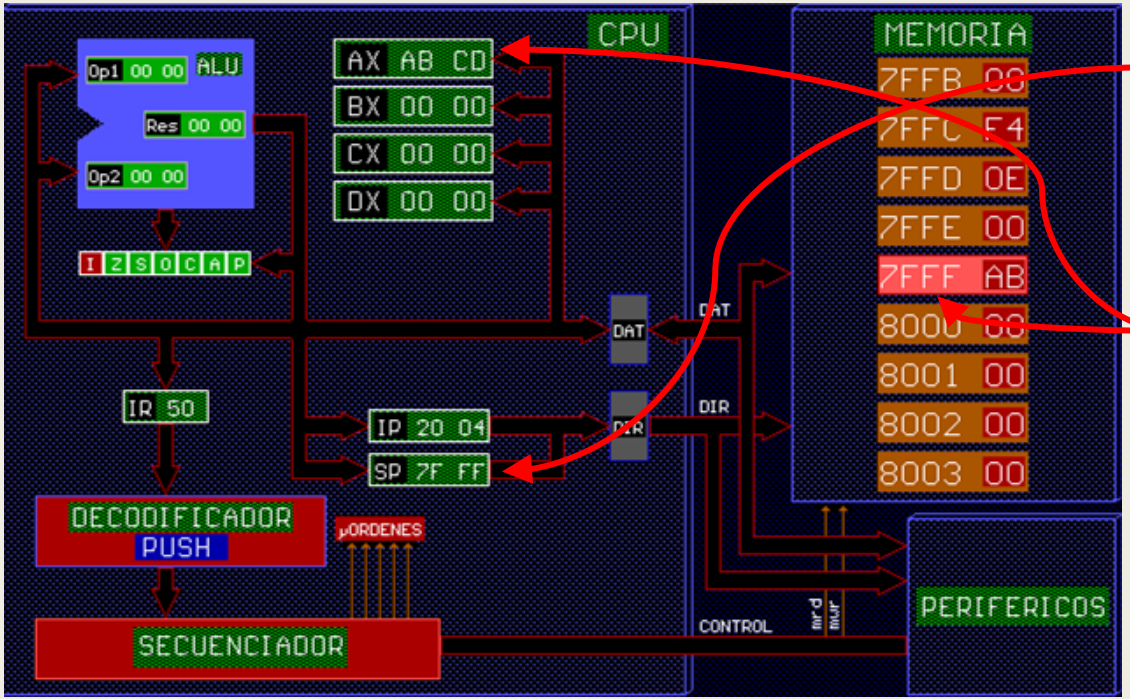
- PUSH: *decrementa* el SP y apila el contenido del registro de 16 bits.
- POP: transfiere 16 bits de la pila al registro e *incrementa* el SP.
- Son instrucciones unarias (requieren un solo operando).
- El destino (PUSH) y el origen (POP) están implícitos.
- El registro SP se modifica automáticamente.
- El espacio de memoria no está protegido (se puede modificar apuntando a direcciones del área 7FFF).

Pila: PUSH AX

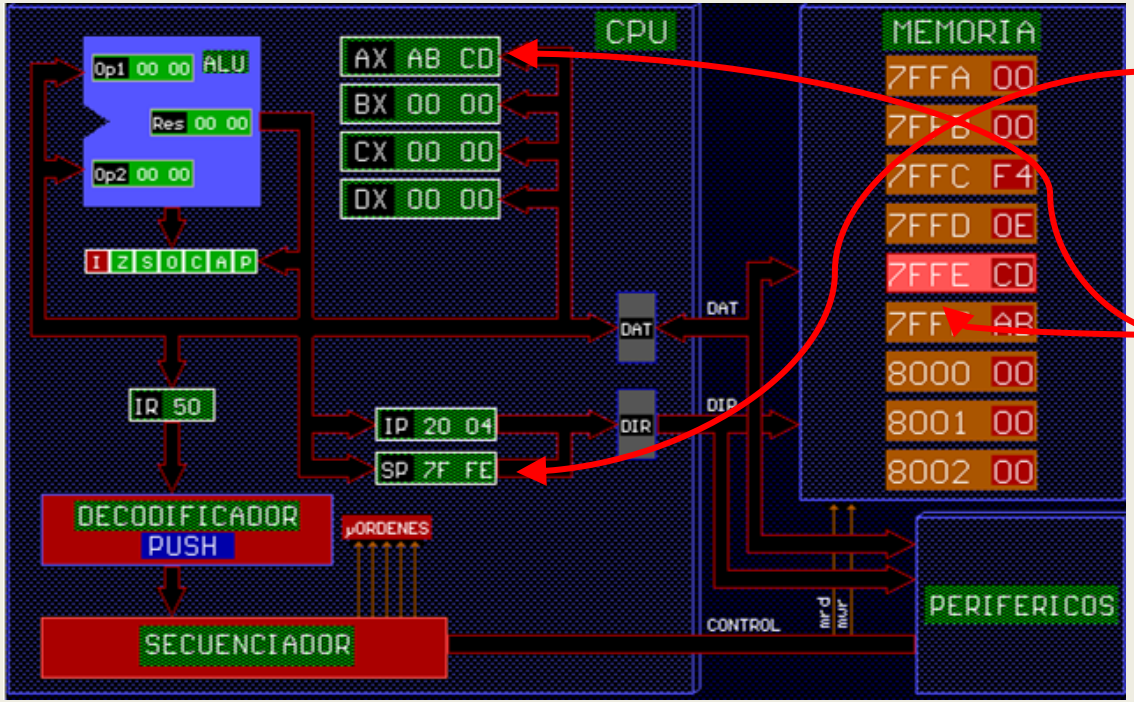


SP inicializado en 8000h

Pila: PUSH AX


CPU
ALU
Op1 00 00
Res 00 00
Op2 00 00
I Z S O C A P
AX AB CD
BX 00 00
CX 00 00
DX 00 00
IR 50
IP 20 04
SP 7F FF
DAT
DIR
DECODIFICADOR
PUSH
µORDENES
SECUENCIADOR
CONTROL
MEMORIA
7FFB 00
7FFC F4
7FFD 0E
7FFE 00
7FFF AB
8000 00
8001 00
8002 00
8003 00
PERIFERICOS
Decrementa SP
Transfiere la parte alta (AH)

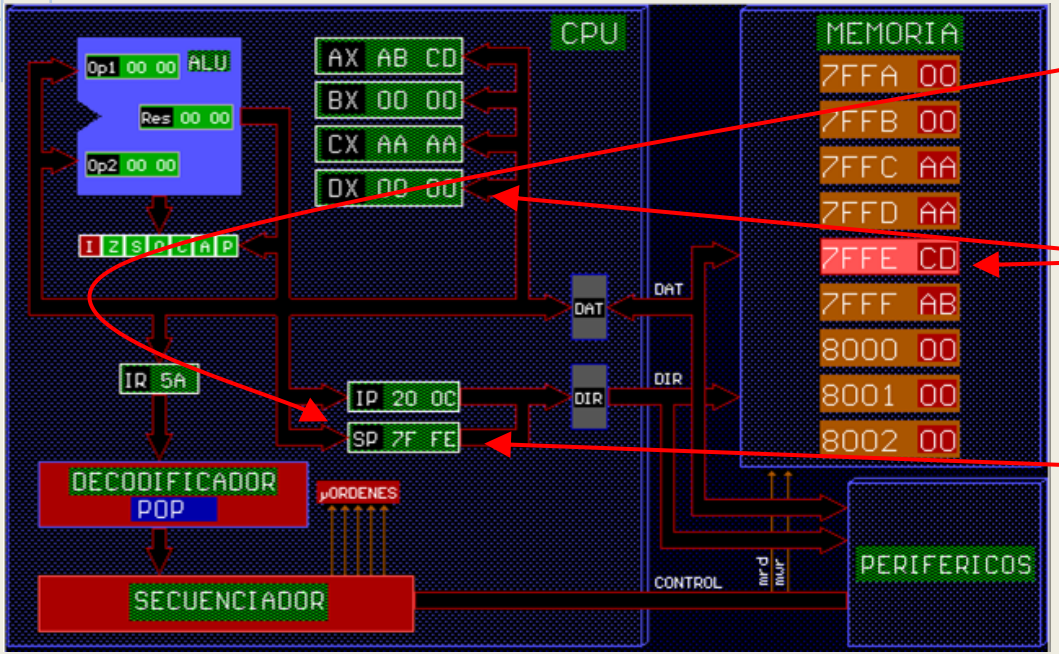
Pila: PUSH AX



Decrementa SP

Transfiere la parte baja (AL)

Pila: POP DX

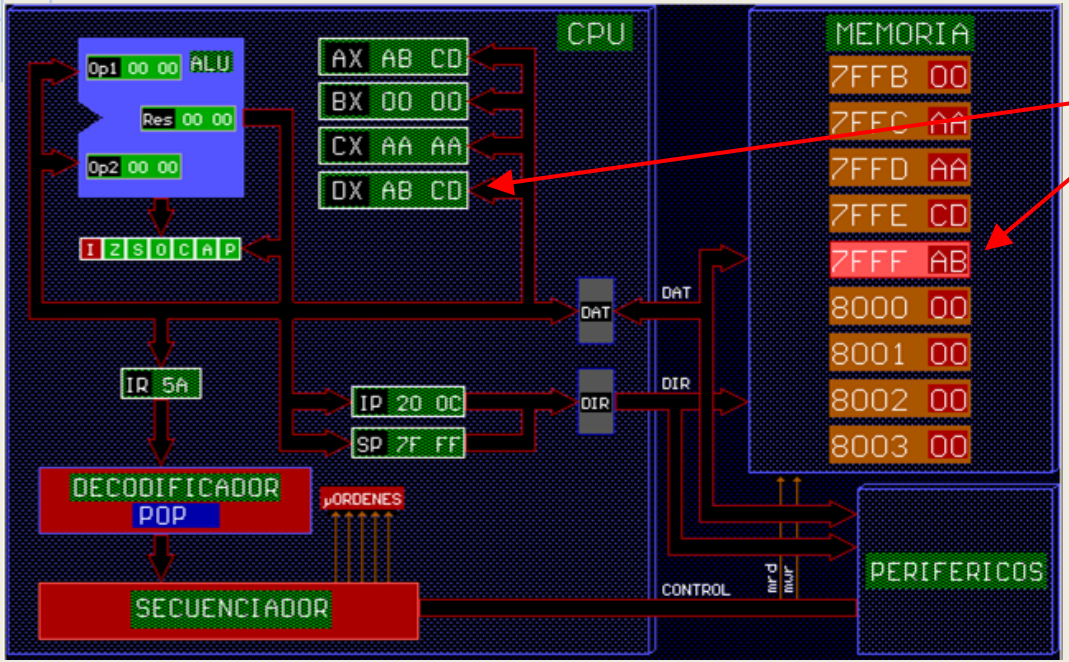


Coloca el contenido de SP en la línea de direcciones

Lee la posición de memoria direccionada y la transfiere a DL

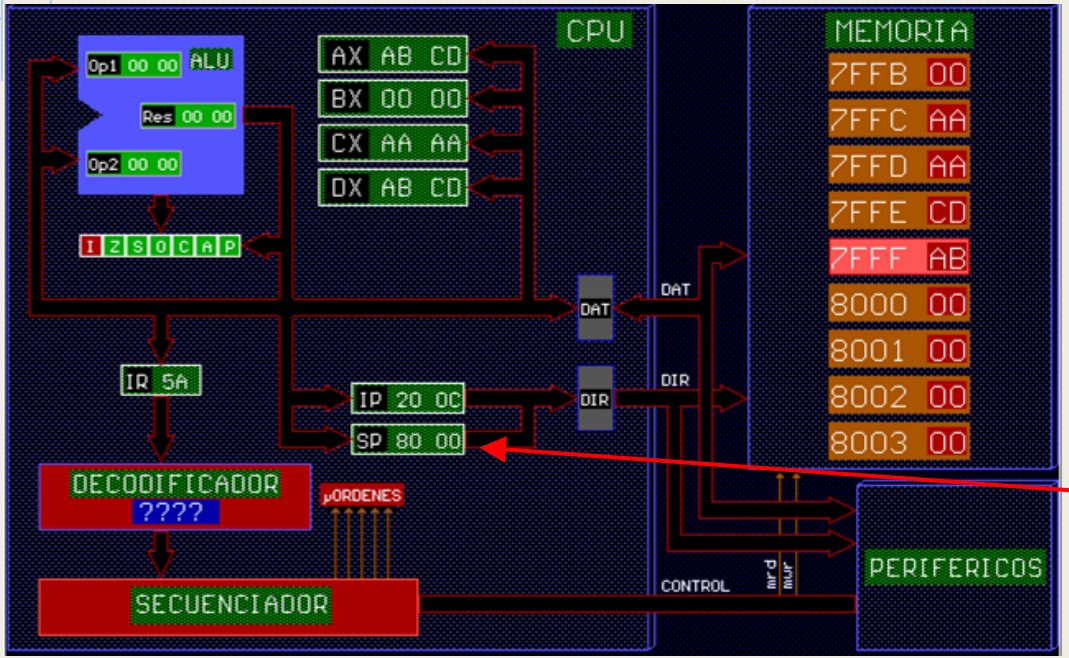
Después de la transferencia incrementa SP

Pila: POP DX



Transfiere el contenido de la posición apuntada por SP a la parte alta de DX (DH)

Pila: POP DX



¿Qué dato contiene la posición 7FFEh después de desapilar?

Llamadas a subrutina

Para mejorar la legibilidad del código es útil implementar subrutinas (similares a los procedimientos y las funciones de lenguajes de alto nivel), modularizando así la solución.

Las instrucciones son captadas desde memoria en secuencia → *Existen instrucciones de salto*

Las llamadas a procedimiento retornan al punto de invocación → *Se necesita una técnica que permita 'guardar' el estado previo a la invocación*

Instrucción de salto que almacene el punto de retorno →

CALL
…
RET

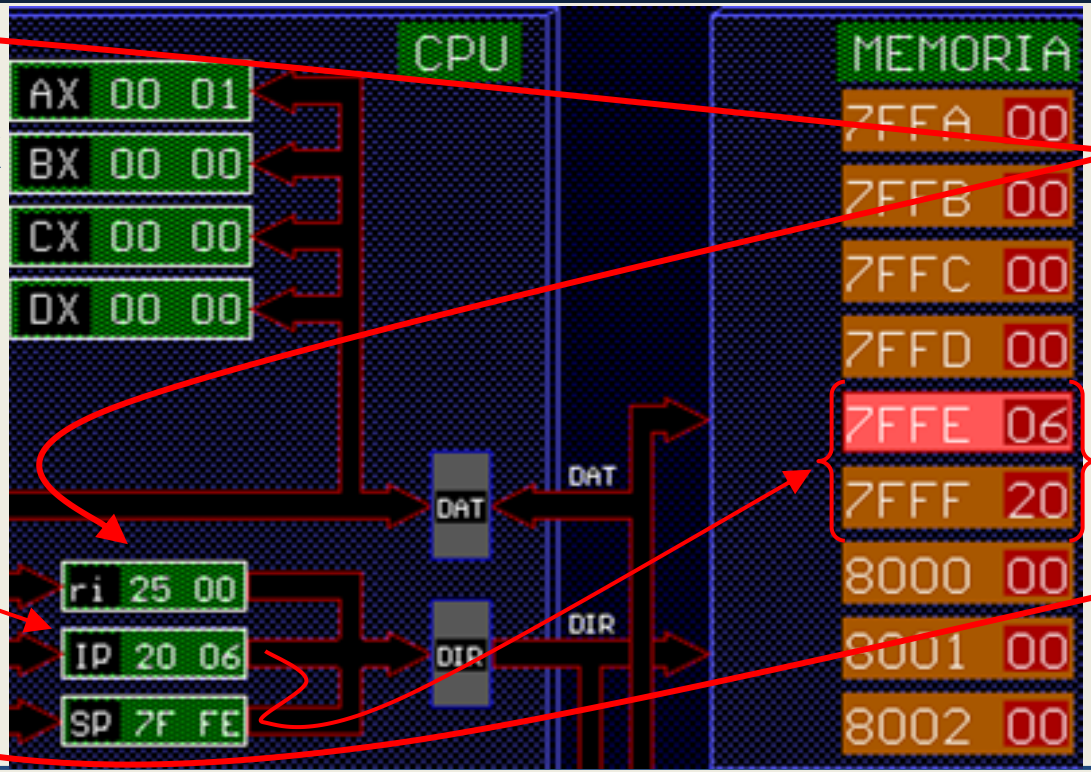
Subrutinas: CALL

La instrucción CALL modifica el contador de programa para alterar la secuencia del programa *apilando* antes el valor de ese registro.

- Es una instrucción unaria (CALL etiqueta_destino)
- Hace uso de la pila y por lo tanto modifica el SP
- Apila el valor actual del registro PC *antes* de modificarlo
- Coloca en el registro PC la dirección identificada por etiqueta_destino
- *Nada dice del retorno al bloque de código que hace la invocación*

Subrutinas: CALL

```
     ORG 2500h
SUBR: ADD AX, AX
      CMP AX, 8
      JNZ SUBR
RET

     ORG 2000h
MOV AX, 1
CALL SUBR
MOV num, AX
HLT
END
```

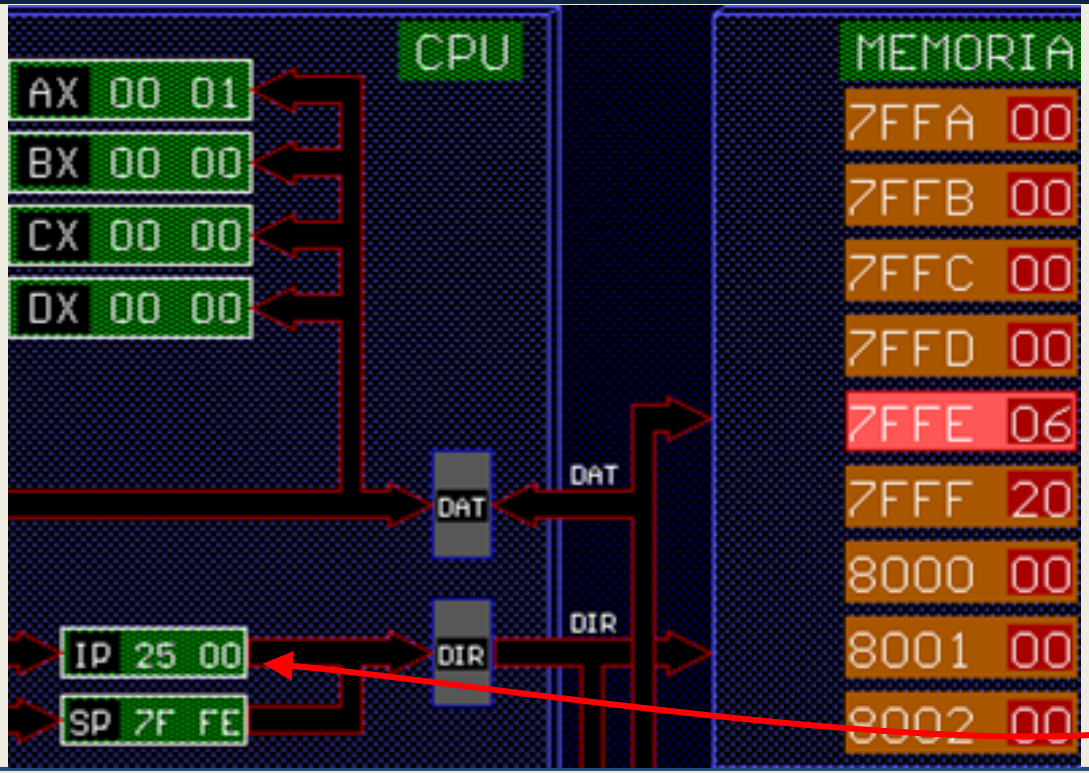


Almacena temporalmente la dirección de SUBR en RI

Apila la próxima instrucción a ejecutar (registro IP)

Subrutinas: CALL

```
      ORG 2500h
SUBR: ADD AX, AX
      CMP AX, 8
      JNZ SUBR
RET

      ORG 2000h
MOV AX, 1
CALL SUBR
MOV num, AX
HLT
END
```



Coloca la dirección etiquetada como SUBR en el contador de programa

Subrutinas: RET

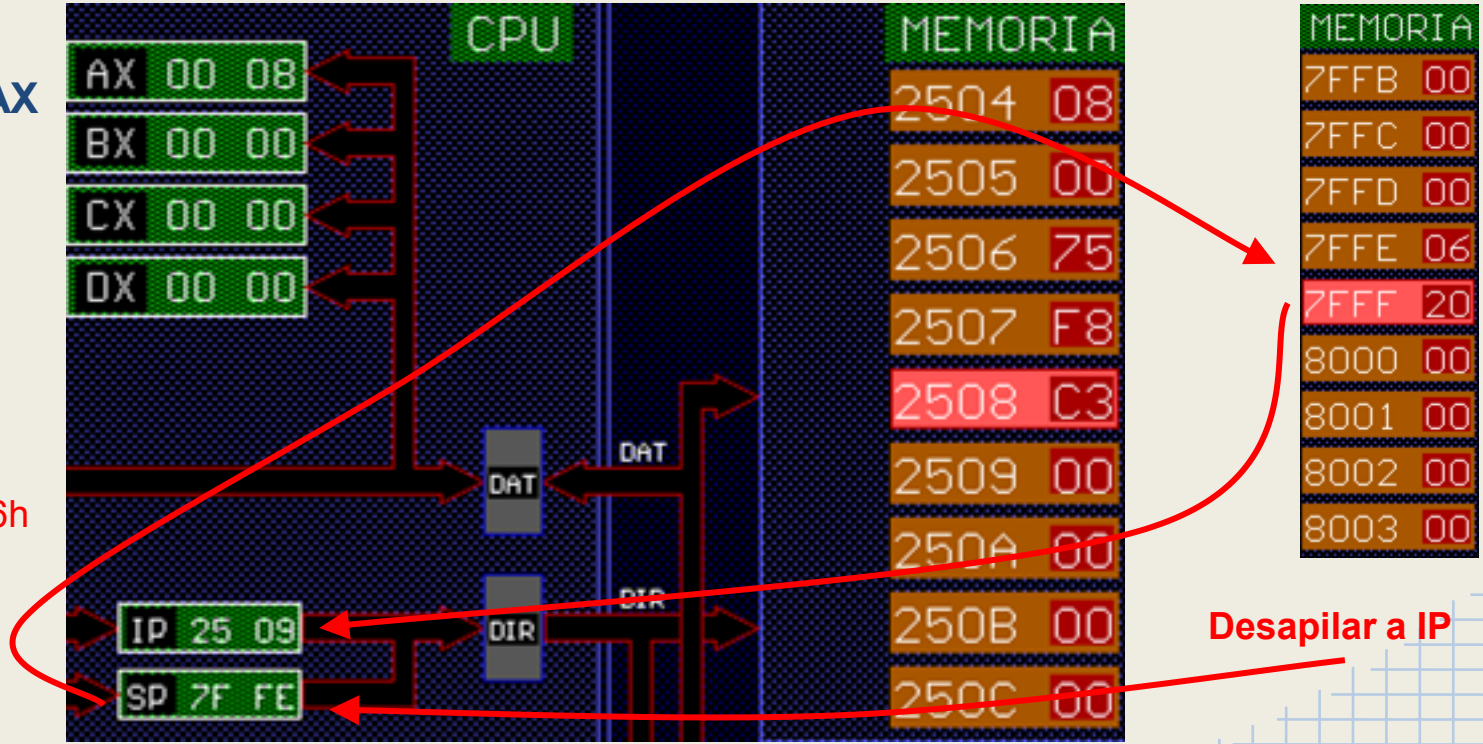
Al finalizar la ejecución de la subrutina se debe volver al lugar de la invocación (instrucción siguiente al CALL). La dirección de esa instrucción fue *apilada* al ejecutar CALL.

La instrucción RET:

- No necesita operandos
- Desapila un dato (modifica SP)
- El dato desapilado se transfiere al contador de programa
- Corresponde al usuario segurar que SP apunta a la dirección donde apiló CALL *(si hace uso de la pila dentro de la subrutina)*

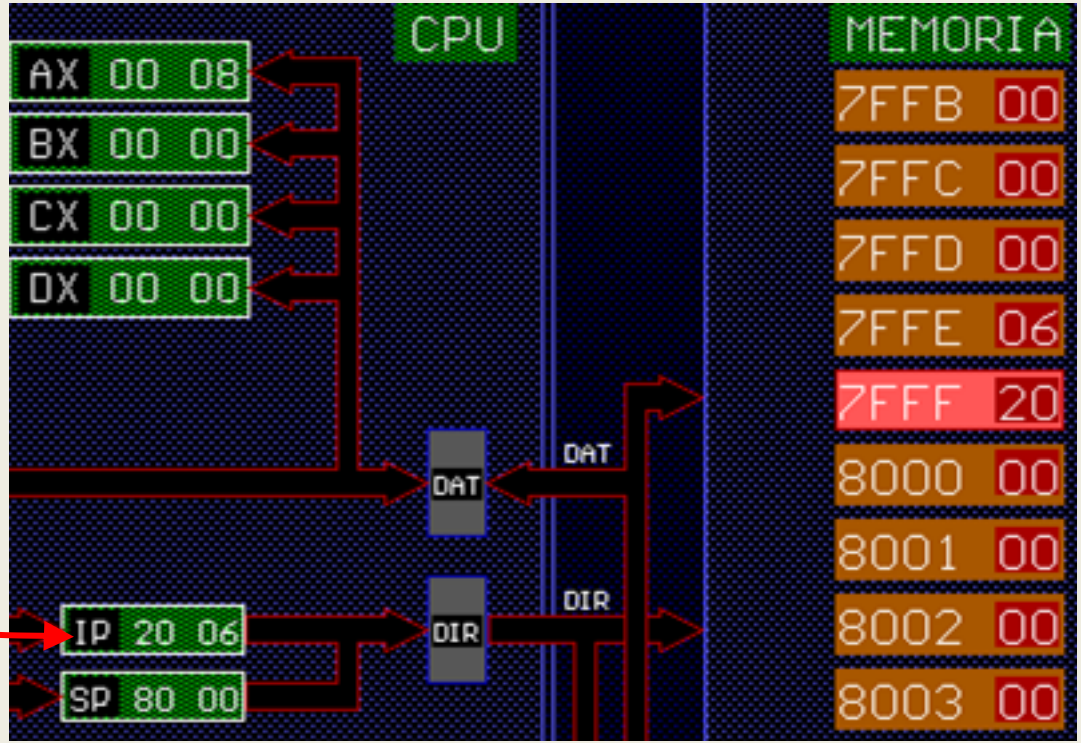
Subrutinas: RET

```
      ORG 2500h
SUBR: ADD AX, AX
      CMP AX, 8
      JNZ SUBR
RET                2508h

      ORG 2000h
MOV AX, 1
CALL SUBR
MOV num, AX        2006h
HLT
END
```



Subrutinas: RET

```
      ORG 2500h
SUBR: ADD AX, AX
      CMP AX, 8
      JNZ SUBR
RET

      ORG 2000h
MOV AX, 1
CALL SUBR
MOV num, AX
HLT
END
```



Estado tras la ejecución de RET

Pasaje de parámetros

En el ejemplo anterior, el valor inicial a sumar (1) es pasado a través del registro AX. Al retornar, el resultado es devuelto en el mismo registro y luego se almacena en la dirección con etiqueta ‘num’.

- La subrutina recibe el valor a utilizar como parámetro
- Procesa ese valor y devuelve el resultado final en un registro
- El parámetro es pasado *por valor*

Pasaje de parámetros

```
      ORG 1000h
num DB 1
```

```
        ORG 2500h
SUBR: MOV CL, [BX]
lazo: ADD CL, CL
      CMP CL, 8
      JNZ lazo
RET
```

```
        ORG 2000h
MOV BX, OFFSET num
CALL SUBR
MOV num, CL
HLT
END
```

BX = Dirección de num

El resultado es devuelto en el registro CL

SUBR recibe la dirección de memoria donde se encuentra el parámetro

Al retornar, el dato en la dirección *num* permanece inalterado

Pasaje de parámetros

En este ejemplo,la subrutina recibe a través del registro BX la dirección de memoria desde donde debe cargar el valor inicial a sumar.

- La subrutina recibe como parámetro la dirección del dato a utilizar
- Transfiere el dato desde la dirección indicada a un registro
- Realiza el proceso con este registro
- El dato original en memoria no se modifica en ningún momento
- El parámetro es pasado *por referencia*