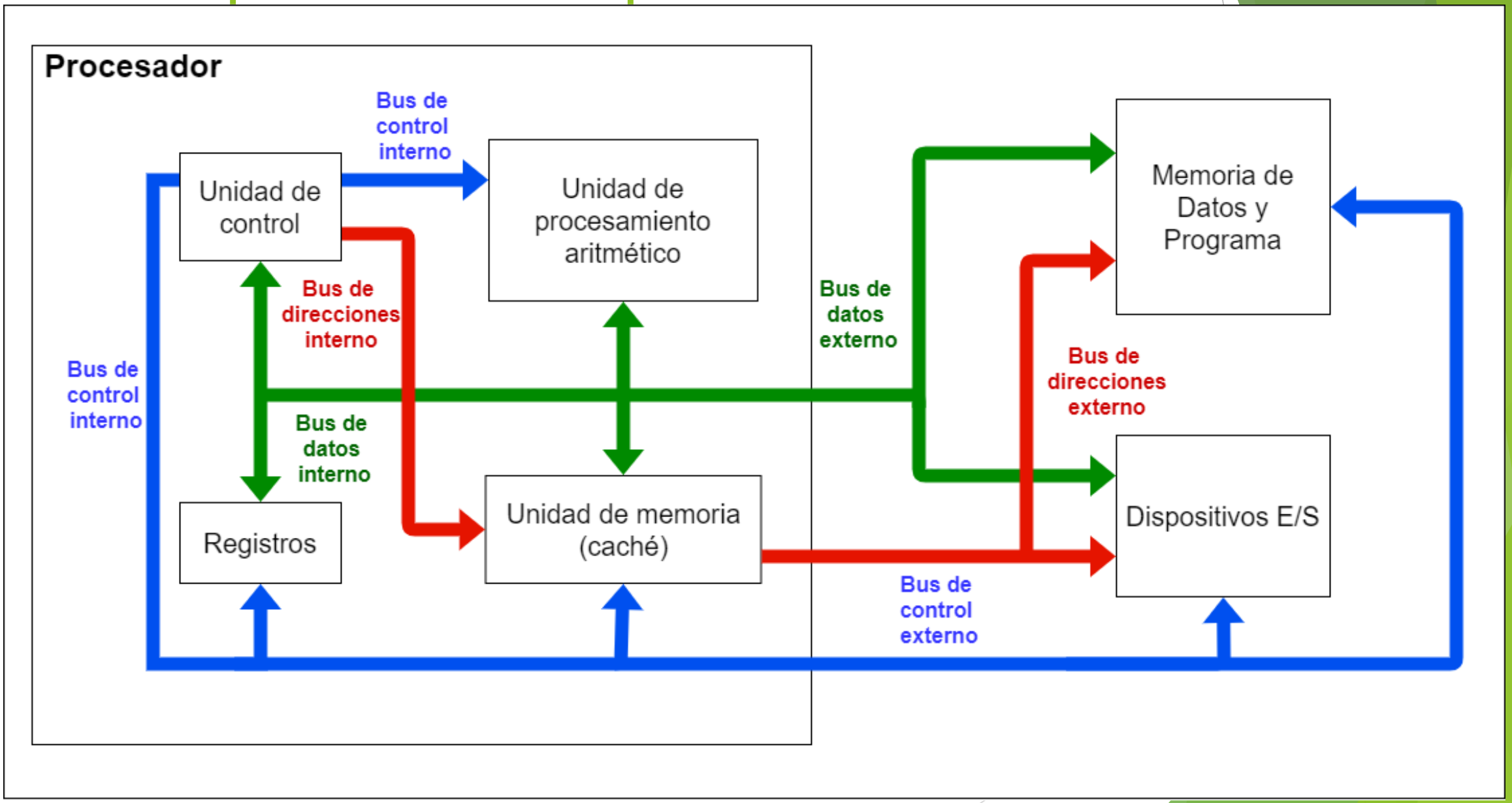
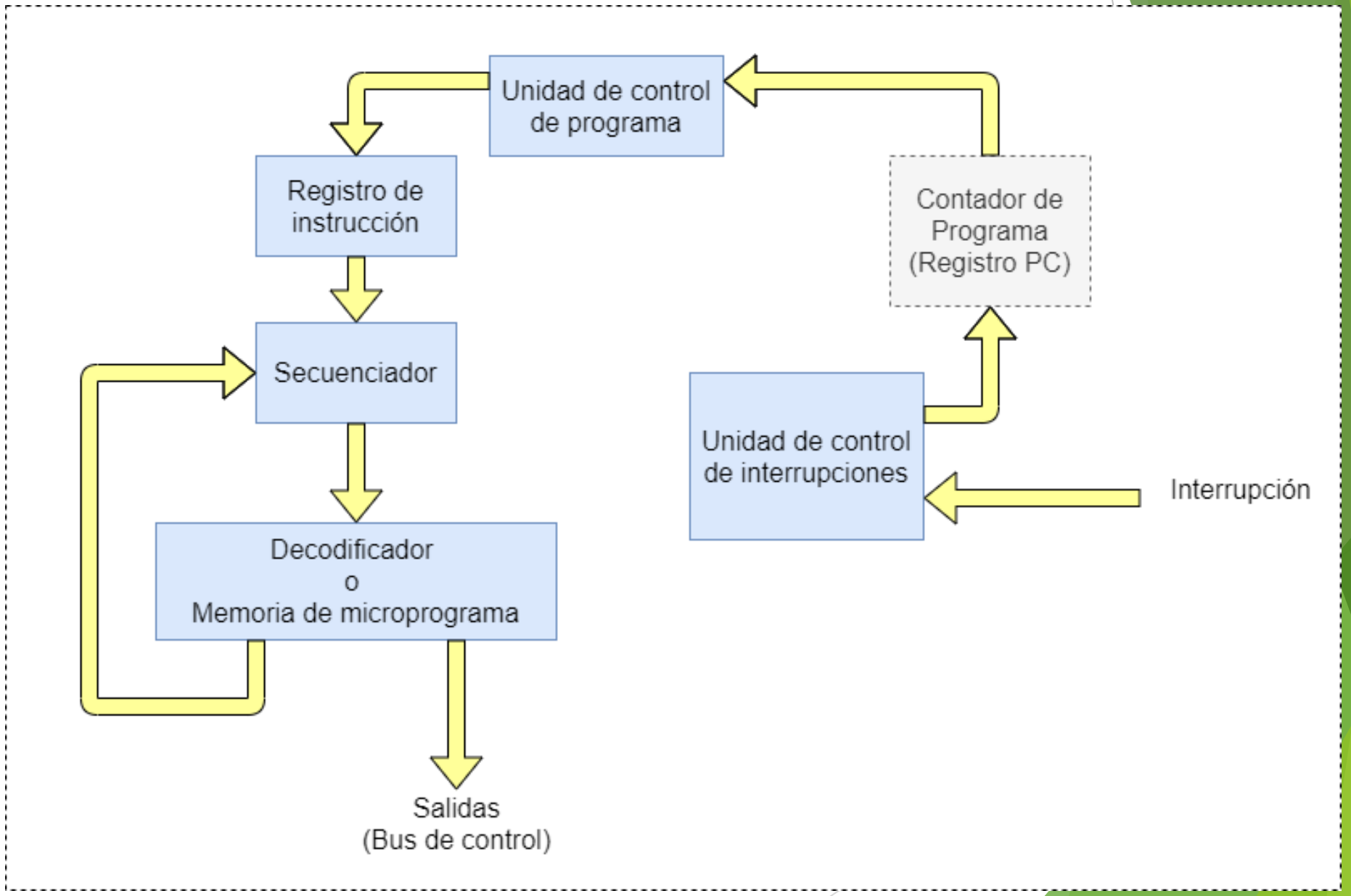
Tema 2: Presentación de un caso real (Programación en lenguaje ensamblador)

Objetivo: El alumno diseñará programas en lenguaje ensamblador para un procesador específico.

2.1: Arquitectura del procesador

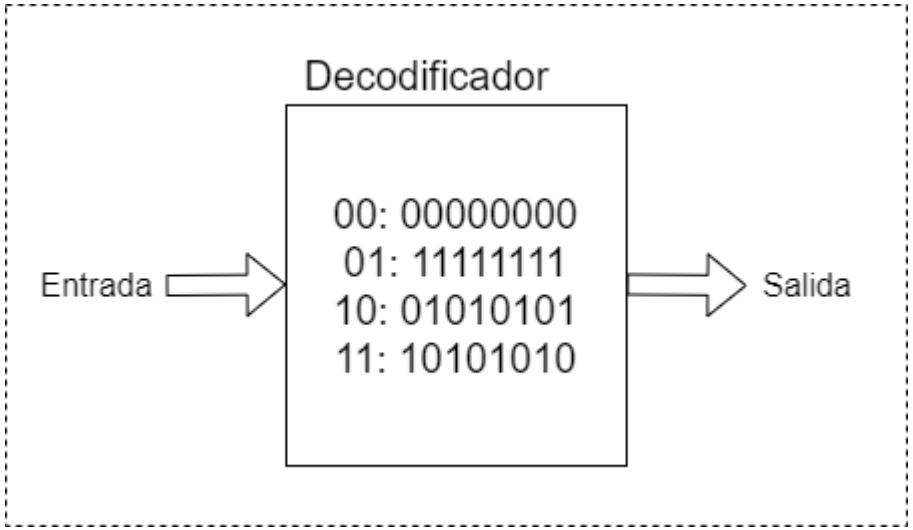


Unidad de control



Unidad de control

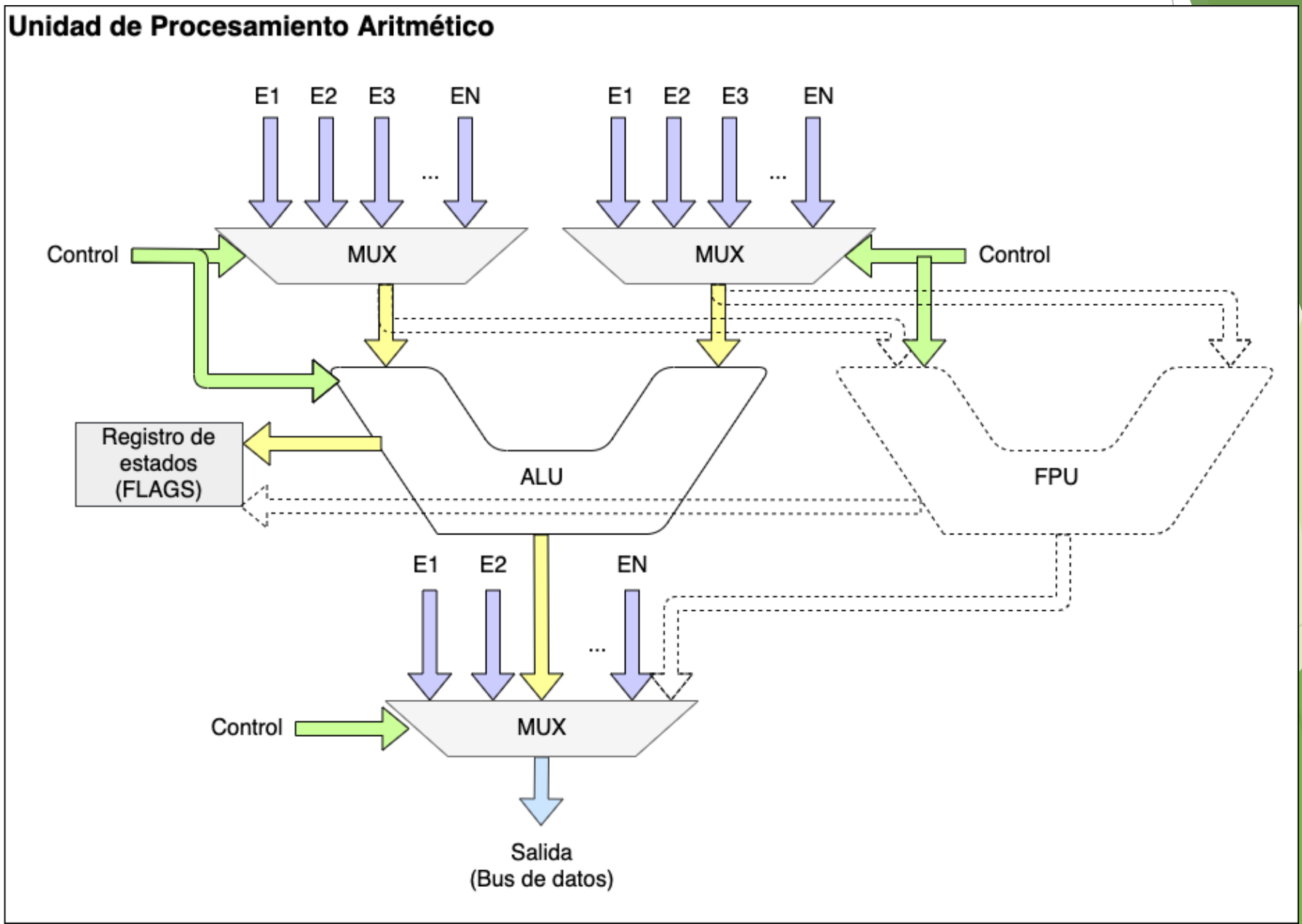
- **Secuenciador (Unidad lógica o de supervisión):** Sincroniza la ejecución de la instrucción con la velocidad de reloj.
- **Registro de instrucción:** registro que contiene la instrucción siguiente a ejecutarse (registro invisible al programador).
- **Decodificador o memoria de microprograma:** Es la lista de operaciones que se ejecutan por cada instrucción. Contiene las señales que se van a enviar dependiendo de la entrada de la instrucción



Unidad de control

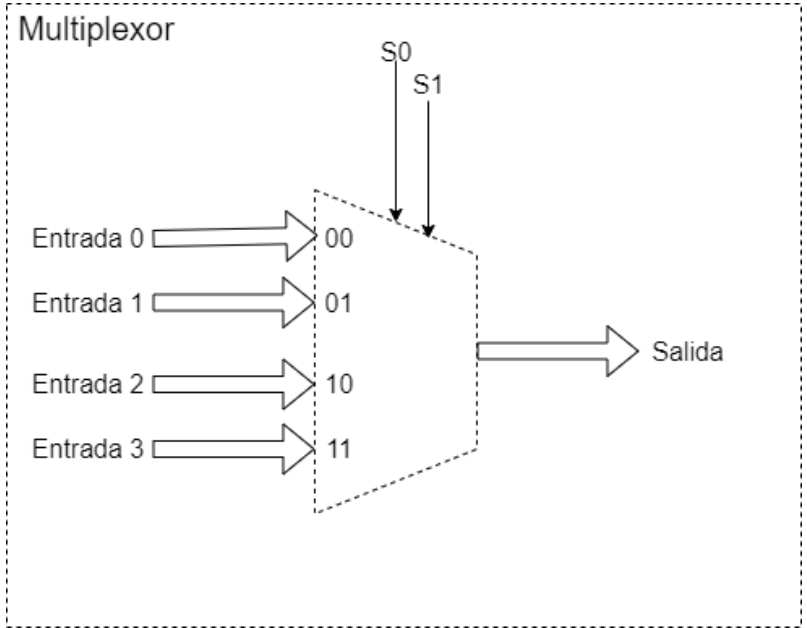
- **Unidad de control de programa:** Calcula la dirección de la siguiente instrucción a ejecutarse.
- **Unidad de control de interrupciones:** Es la unidad que se encarga del manejo de interrupciones.
  - Una **interrupción**, en computación, **es una señal de entrada** que indica que un evento importante **debe atenderse de inmediato**. El procesador va a detener su ejecución actual para atenderla.

Unidad de Procesamiento Aritmético



Unidad de Procesamiento Aritmético

- **ALU (Unidad Aritmético-Lógica):** Se encarga de realizar las operaciones aritméticas y lógicas en el procesador.
- **Multiplexor**: Es un circuito combinacional con varias entradas y una única salida de datos. La entrada pasa a la salida dependiendo de las líneas de selección



Unidad de Procesamiento Aritmético

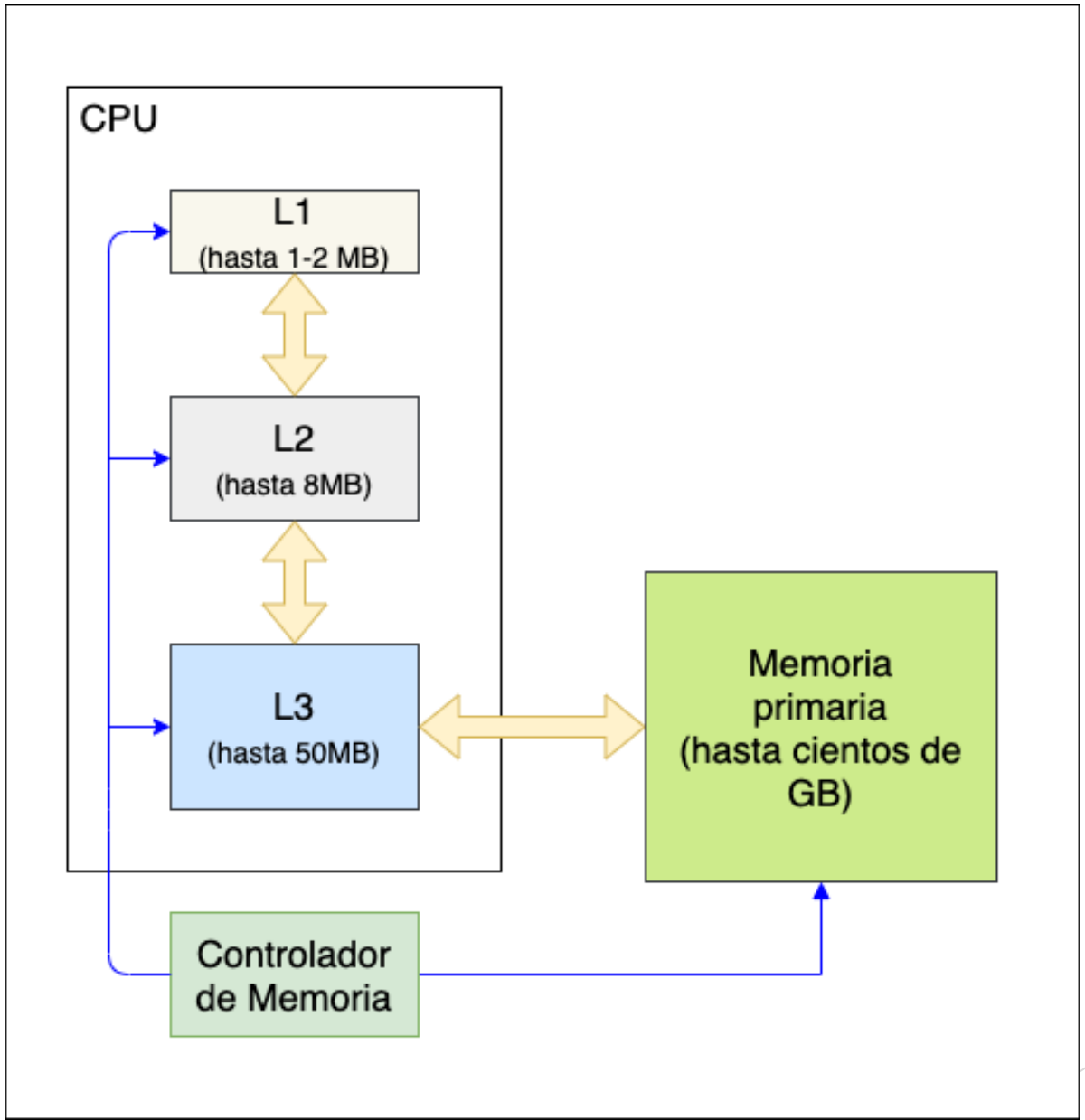
- **FPU (*Floating-Point Unit*):** también conocido como co-procesador matemático. Se encarga de todas las operaciones matemáticas que involucran números de punto flotante o fracciones.
- **Registro de estados**: Contiene los valores que indican el estado de los componentes de la arquitectura.

Registro de estados (FLAGS, en Intel x86)

Unidad de Procesamiento Aritmético

C (Acarreo): Suma o resta generan acarreo
P (Paridad): Número de 1's en un dato binario. 0 - paridad impar; 1 - paridad par
A (Acarreo auxiliar, o medio acarreo): Funciona como el acarreo pero a la mitad de los datos binarios
Z (Cero): Cuando el resultado de una operación es 0. Si resultado = 0, Z=1; si no, Z=0
S (Signo): Indica el signo del resultado de una operación aritmética. Si resultado < 0, S=1; si no, S=0
O (Desbordamiento): Cuando la operación de dos datos de un mismo tamaño requiere un dato de un tamaño mayor

Unidad de Memoria (caché)



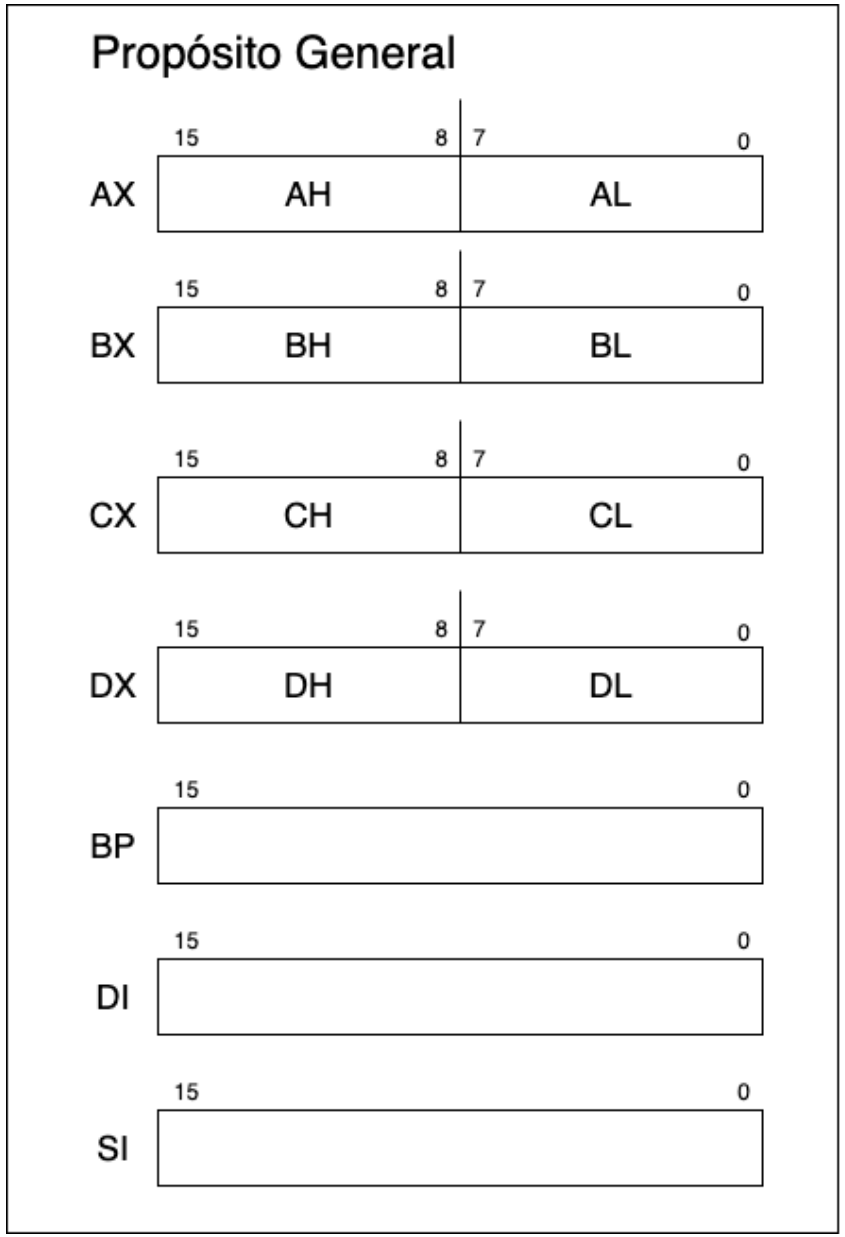
Unidad de Memoria (caché)

- La memoria caché es un **tipo de memoria específica** que está preparada para servir de **apoyo al procesador**. Actúa como un **sistema de almacenamiento de instrucciones y de datos** capaz de comunicarse con el procesador a gran velocidad.
- Dependiendo el nivel, es más veloz, pero también de menor capacidad de almacenamiento.
  - Caché L1 (nivel 1) es la más rápida, pero es la de menor tamaño, la mayoría puede llegar hasta 256 KB, pero existen algunas que llegan hasta los 2 MB.
  - Caché L2 (nivel 2) es la que sigue en velocidad, y es de mayor tamaño que la de nivel 1. Puede llegar hasta los 8 MB.
  - Caché L3 (nivel 3) es la caché más lenta dentro del procesador, pero es la de mayor tamaño, puede llegar hasta 50 MB.

Registros internos

- Propósito general
- Propósito específico
- Registros de segmento

Registros internos (propósito general)

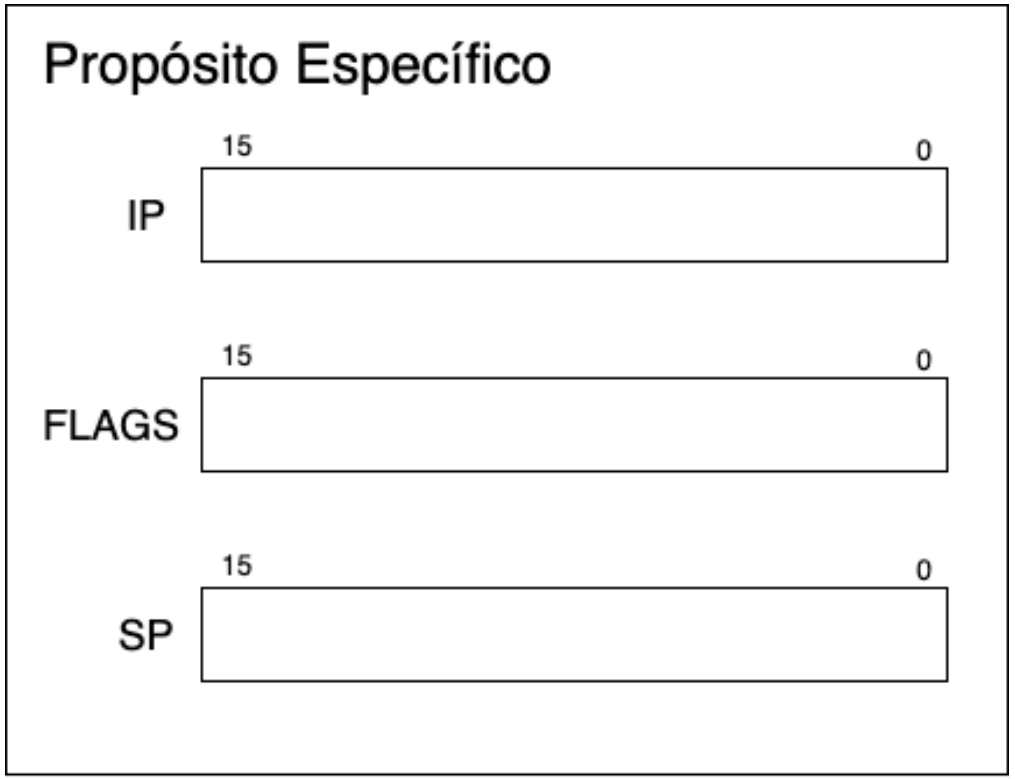


Registros internos (propósito general)

Los registros de propósito general pueden almacenar datos o direcciones de memoria, aunque en el Intel x86, estos registros pueden tener usos particulares:

- AX (Acumulador): Almacena el resultado de algunas operaciones aritméticas.
- BX (Índice Base): Apuntador a localidades de memoria de datos.
- CX (Contador): Se usa en algunas instrucciones lógicas y en loops.
- DX (Datos): Se usa en algunas operaciones aritméticas y de Entrada/Salida.
- BP (Apuntador a la Base [del *stack*]): Apuntador a la base de la pila.
- SI (Índice de origen): Se usa para apuntar al origen de operaciones de cadena.
- DI (Índice de destino): Se usa para apuntar al destino de operaciones de cadena.

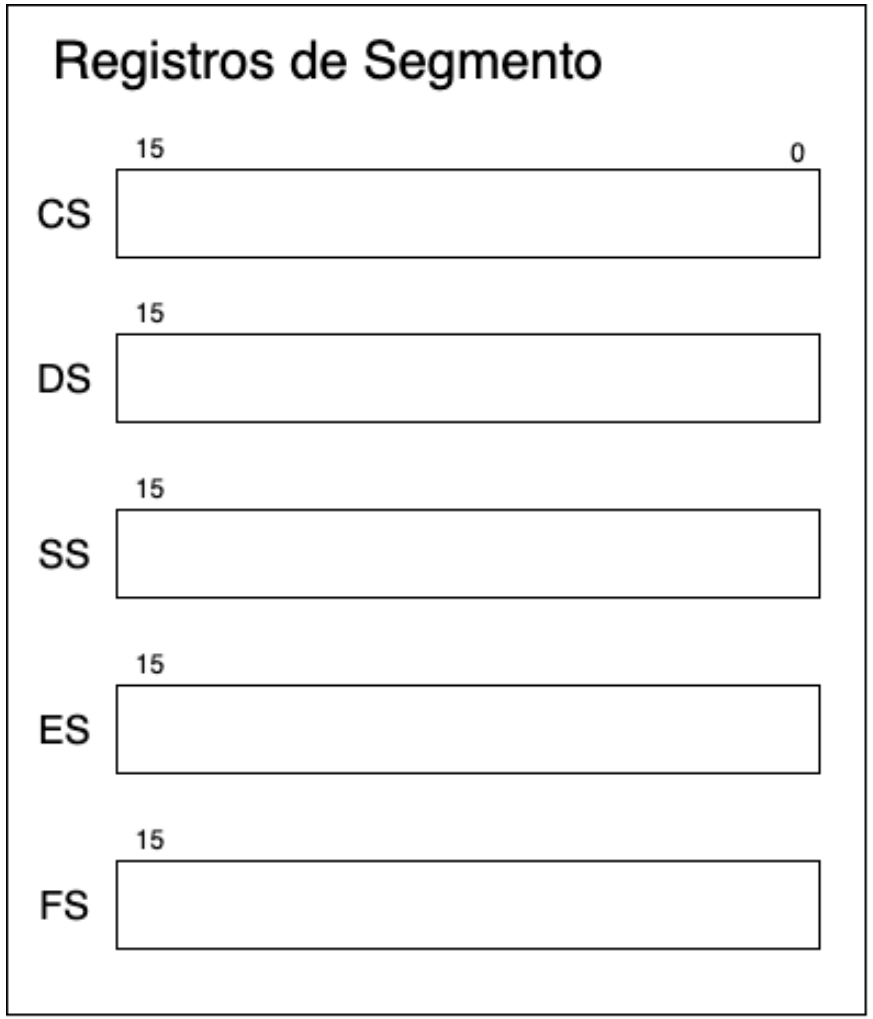
Registros internos (propósito específico)



Registros internos (propósito específico)

- IP (Apuntador a Instrucciones): Direcciona la siguiente instrucción a ejecutar del segmento de código.
- SP (Apuntador a pila): Direcciona el área de memoria de la pila.
- FLAGS (Banderas): Registro de estados del procesador, utiliza una colección de bits y ciertos resultados de operaciones.

Registros internos (registros de segmento)



Registros internos (registros de segmento)

Generan direcciones de memoria cuando se combinan con otros registros (IP, BX, BP, SP, SI, DI).

- CS (Segmento de Código): Apuntador a la primera dirección de memoria en donde se almacena el código de un programa.
- DS (Segmento de Datos): Apuntador a la sección de memoria que contiene los datos utilizados por el programa.
- SS (Segmento de Pila): Apunta a la sección de memoria utilizada por la pila.
- ES (Segmento Extra): Apuntador a una sección de memoria adicional, que se puede utilizar para algunas instrucciones de cadena.
- FS y GS: Apuntadores a segmentos de memoria suplementarios.

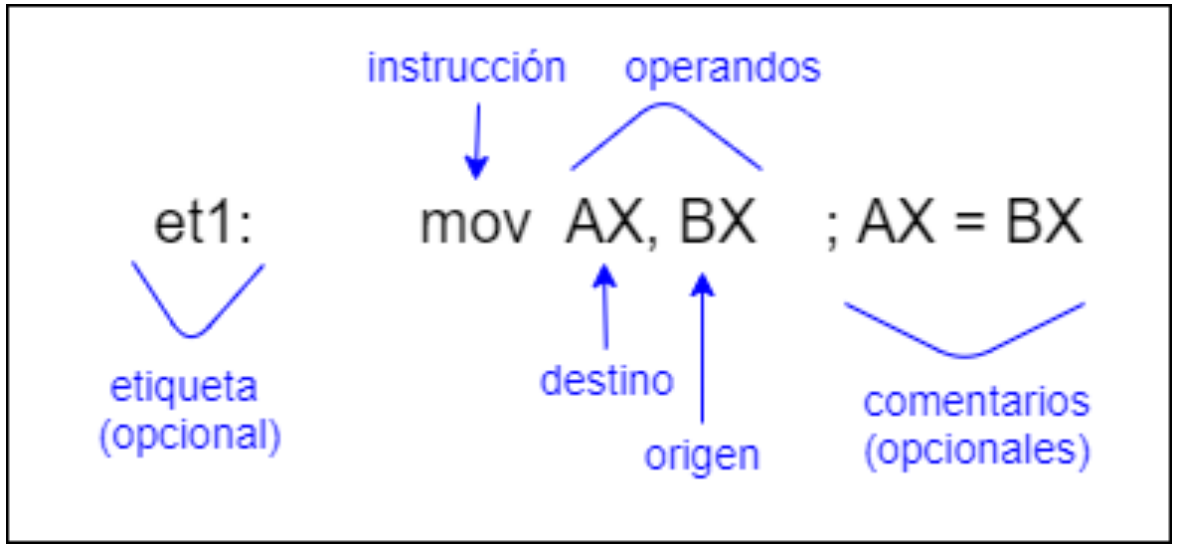
2.2 Modos de direccionamiento

- Este término se refiere a la manera en la que se especifican los operandos de una instrucción.
- Los modos de direccionamiento especifican las reglas para interpretar o modificar el campo de dirección de la instrucción antes de que sea ejecutada.
- Los modos de direccionamiento nos sirven para:
  - Para desarrollar software más eficiente.
  - Conocer el esquema de direccionamiento (0,1,2,3 direcciones) empleado por cada instrucción.
- En arquitectura Intel x86 existen diferentes modos de direccionamiento dependiendo a qué segmento de memoria se está direccionando.
  - Modos de direccionamiento de memoria de datos.
  - Modos de direccionamiento de memoria de programa.
  - Modos de direccionamiento de memoria de la pila.

Modos de direccionamiento de memoria de datos

- Implícito
- Inmediato
- Directo
- De registros
- De registro indirecto
- De base más índice
- De registro relativo
- De base relativa más índice

Antes de comenzar:
Sintaxis de una instrucción en lenguaje ensamblador (arquitectura x86)



Operandos válidos:

- Registro a Registro
- Memoria a Registro
- Registro a Memoria
- Constante a Registro
- Constante a Memoria

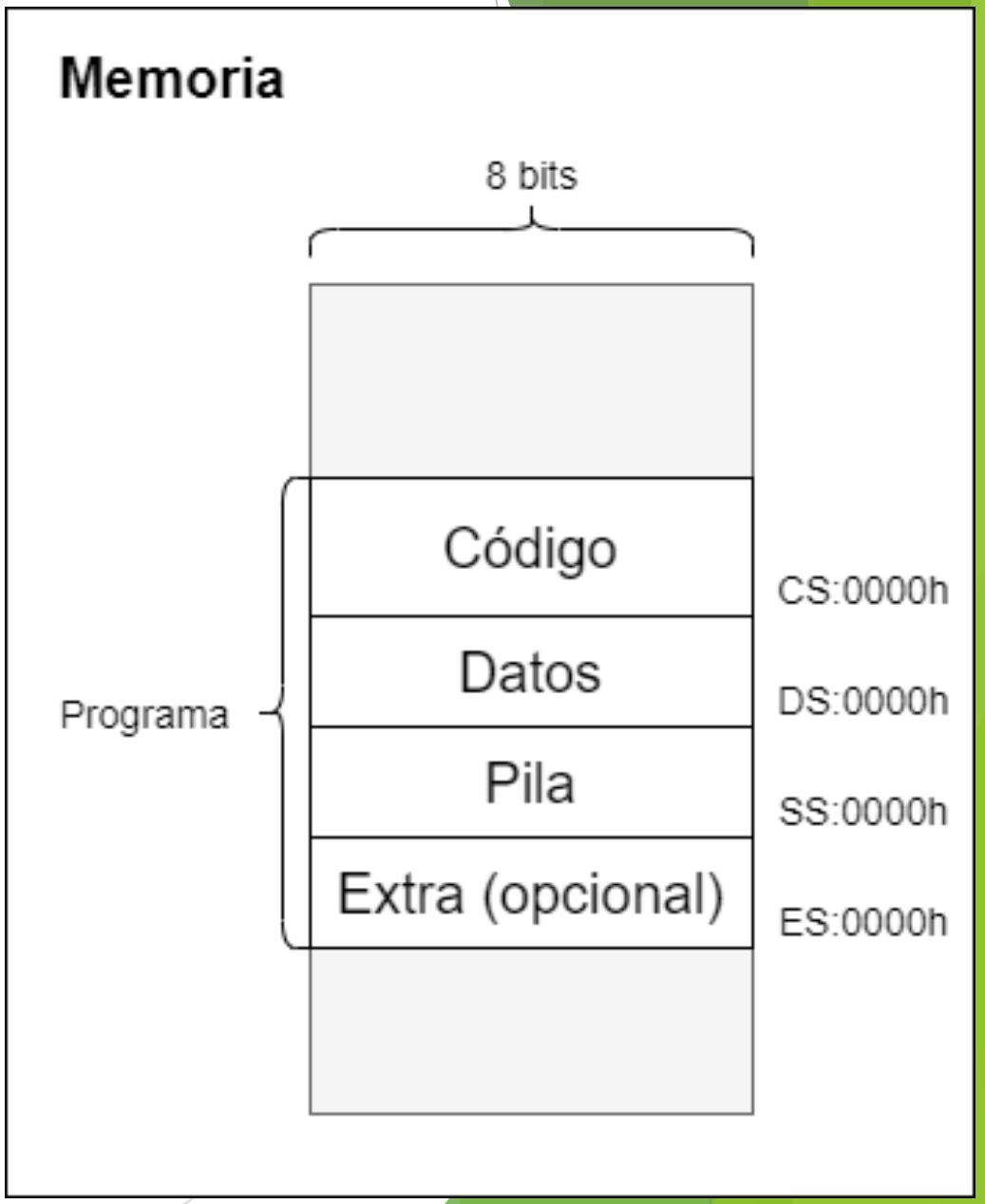
```
mov ax,bx  ; ax = bx
mov ax,[var]  ; ax = [var]
mov [var], ax ; [var] = ax
mov ax, 5  ; ax = 0005h
mov [var], 5  ; [var] = 05h
```

Antes de comenzar: Sistemas numéricos en Intel x86

- En lenguaje ensamblador para arquitectura Intel x86, podemos utilizar los sistemas de numeración siguientes:
  - Decimal (default): 241 o 241d
  - Binario: 11110001b
  - Hexadecimal: F1h
  - Octal: 361o

Antes de comenzar: Memoria

- La memoria se divide en localidades, cada localidad almacena 8 bits (1 byte).
- El Sistema Operativo divide la memoria en segmentos, un segmento será un conjunto de bytes.
- Cada programa ocupa varios segmentos de memoria para almacenar información del mismo. Un programa contendrá: un segmento de código donde se almacenan las instrucciones; un segmento de datos, donde se almacenan los datos; un segmento de pila, para el uso de la pila; y un segmento extra que puede extender el uso de memoria de datos.
- Una dirección de memoria se representa por un segmento y un desplazamiento. Ejemplo: la dirección CS:0002h representa la localidad 2 de memoria del segmento de código.



Modos de direccionamiento de memoria de datos

Implícito

El operando está especificado en la instrucción misma. Es decir, no se especifica un operando, si no el procesador ya sabe sobre qué operando se aplica la operación.

- Ejemplos:
  - clc ; clear carry, Bandera de Carry a 0
  - stc ; set carry, Bandera de Carry a 1

Modos de direccionamiento de memoria de datos

Inmediato

Transfiere un dato inmediato (constante) a un registro o localidad de memoria.

- Ejemplos:
  - mov ax,1234h ; AX = 1234h
  - mov [var], 12h ; ds:[var] = 12h

Modos de direccionamiento de memoria de datos

Directo

Transfiere un dato (de tamaño byte) entre una localidad de memoria y un registro.

- Ejemplos:
  - mov ah,[var1] ; AH = [var1]
  - mov [var2], al ; [var2] = al
  - mov bx, ds:[var3] ; bx = [var3]

Modos de direccionamiento de memoria de datos

De registros

Transfiere un dato entre 2 registros

No se permite cambiar el registro CS

No se permite MOV para mover registros de segmento a segmento

- Ejemplos:
  - mov ax,bx ; AX = BX
  - mov DS, AX ; DS = AX
  - ~~mov ss, cs~~

Modos de direccionamiento de memoria de datos

De registro indirecto

Transfiere un dato entre un registro y una localidad de memoria direccionada por un registro índice (BX, DI, SI, BP)

- Ejemplos:
  - mov al,[bx] ; AL = ds:[BX]

Modos de direccionamiento de memoria de datos

De base más índice

Transfiere un dato entre un registro y una localidad de memoria direccionada por un registro base (BX, BP) más un registro índice (DI, SI)

- Ejemplos:
  - mov al,[bx + di] ; AL = [BX+DI]

Modos de direccionamiento de memoria de datos

De registro relativo

Transfiere un dato entre un registro y una localidad de memoria direccionada por un registro base o índice, más un desplazamiento (constante)

- Ejemplos:
  - mov al,[bx + 4] ; AL = [BX+4]

Modos de direccionamiento de memoria de datos

De base relativa más indice

Transfiere un dato entre un registro y una localidad de memoria direccionada por un registro base y un índice, más un desplazamiento.

- Ejemplos:
  - mov al,[bx + di + 4] ; AL = [BX+DI+4]

Modos de direccionamiento de memoria de programa

Se aplican sobre instrucciones de salto que afectan el flujo de ejecución de un programa.

- Instrucciones de salto (jmp, jc, js, jz, ja, etc.)
- Llamadas a procedimientos (call)

- Directo
- Relativo
- Indirecto

Modos de direccionamiento de memoria de programa

Directo

Comúnmente llamado ‘salto lejano’ porque puede saltar a cualquier posición de memoria de programa dentro del segmento de código.

- Ejemplos:
  - jmp cs:0000h
  - jc cs:0040h

Modos de direccionamiento de memoria de programa

Indirecto

Utiliza el contenido de un registro de 16 bits (AX, BX, CX, DX, SP, BP, DI, SI)

- Ejemplos:
  - jmp BX
  - jc AX

Modos de direccionamiento de memoria de programa

Relativo

Es relativo al registro IP. Se realiza haciendo desplazamientos hacia adelante o atrás, sumando un offset al registro IP.

Este tipo de salto no está disponible para el programador. Se produce por el ensamblador al utilizar etiquetas, el ensamblador calcula la dirección de la siguiente instrucción.

- Ejemplos:
  - jmp 4
  - jc -3

Modos de direccionamiento de memoria de la pila

La pila es importante en los microprocesadores. Puede guardar datos de manera temporal y también almacena direcciones de retorno cuando se utilizan llamadas a procedimientos.

La memoria de la pila es LIFO (*Last Input, First Output*) que describe la manera en la que se introducen y se obtienen los datos de ella.

Algunas instrucciones que hacen uso de la pila son:

- Instrucción PUSH: para colocar datos en la pila.
- Instrucción POP: para sacar datos de la pila.
- Instrucción CALL: llamada a procedimientos (el equivalente a una función o método en lenguaje de alto nivel), guarda la dirección de retorno en la pila.
- Instrucción RET: fin de procedimiento, saca la dirección de retorno de la pila.

Modos de direccionamiento de memoria de la pila

El direccionamiento de memoria de la pila se considera IMPLÍCITO, ya que se da por hecho que los datos siempre se van a introducir u obtener del tope de la pila.

Sin embargo, se pueden acceder datos de la memoria de la pila de manera DIRECTA accediendo a una dirección específica del segmento de pila SS:0000h, o INDIRECTA utilizando los registros apuntadores a la pila (SP y BP).

Modos de direccionamiento de memoria de la pila

Implícito

Introduce u obtiene los datos directamente del tope de la pila.

- Ejemplos:
  - push ax ; Introduce el contenido del registro AX en el tope de la pila
  - pop bx ; Obtiene el contenido del tope de la pila y lo guarda en registro BX

Modos de direccionamiento de memoria de la pila

Directo

Se transfiere un dato entre una localidad de memoria específica de la pila y un registro.

- Ejemplos:
  - mov ax,ss:[0000h]
  - mov ss:[0000h],bx

Modos de direccionamiento de memoria de la pila

Indirecto

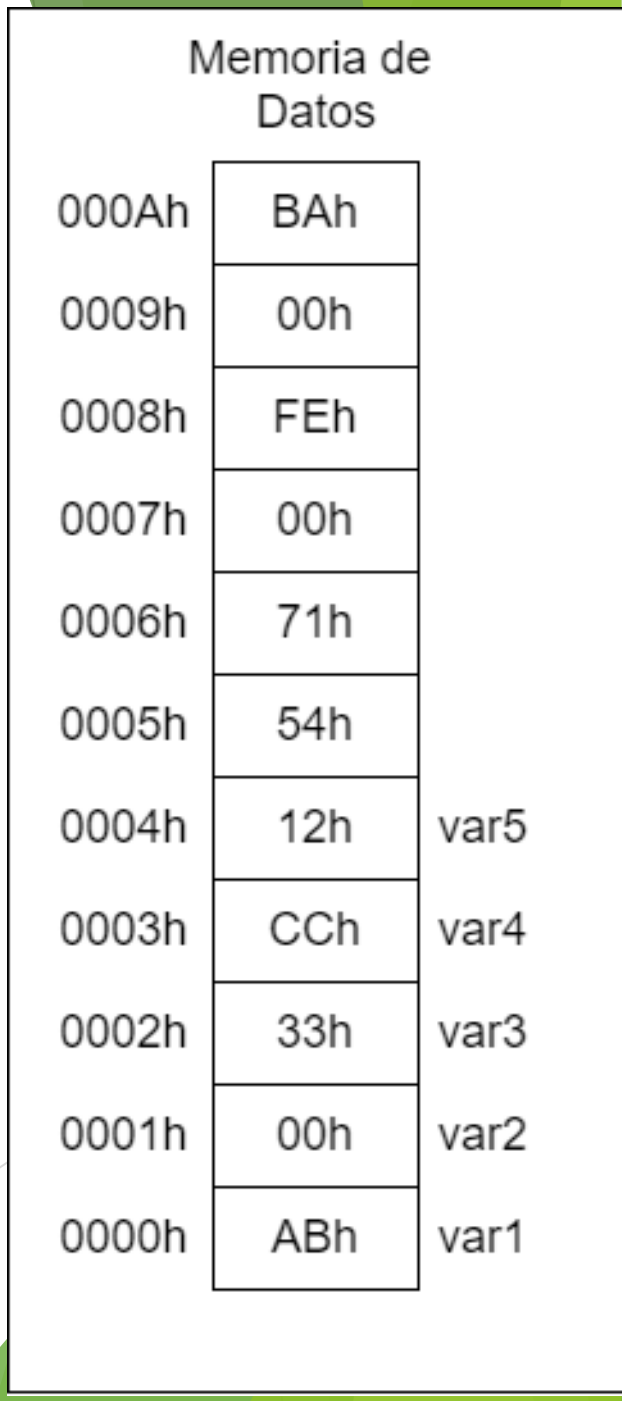
Se transfiere un dato entre una localidad de memoria de pila apuntada por un registro apuntador de pila (SP, BP) y un registro.

- Ejemplos:
  - mov ax,[bp]
  - mov ss:[sp],bx

Ejemplo de modos de direccionamiento de memoria de datos

Instrucción	Resultado	Modo de Direccionamiento
mov ax, 5d		
mov cl, [var3]		
mov cx, ax		
mov ah, [bx+di]		
mov [bx], ah		
mov ch, [bx+5]		
mov ax, [bx+si+2]		

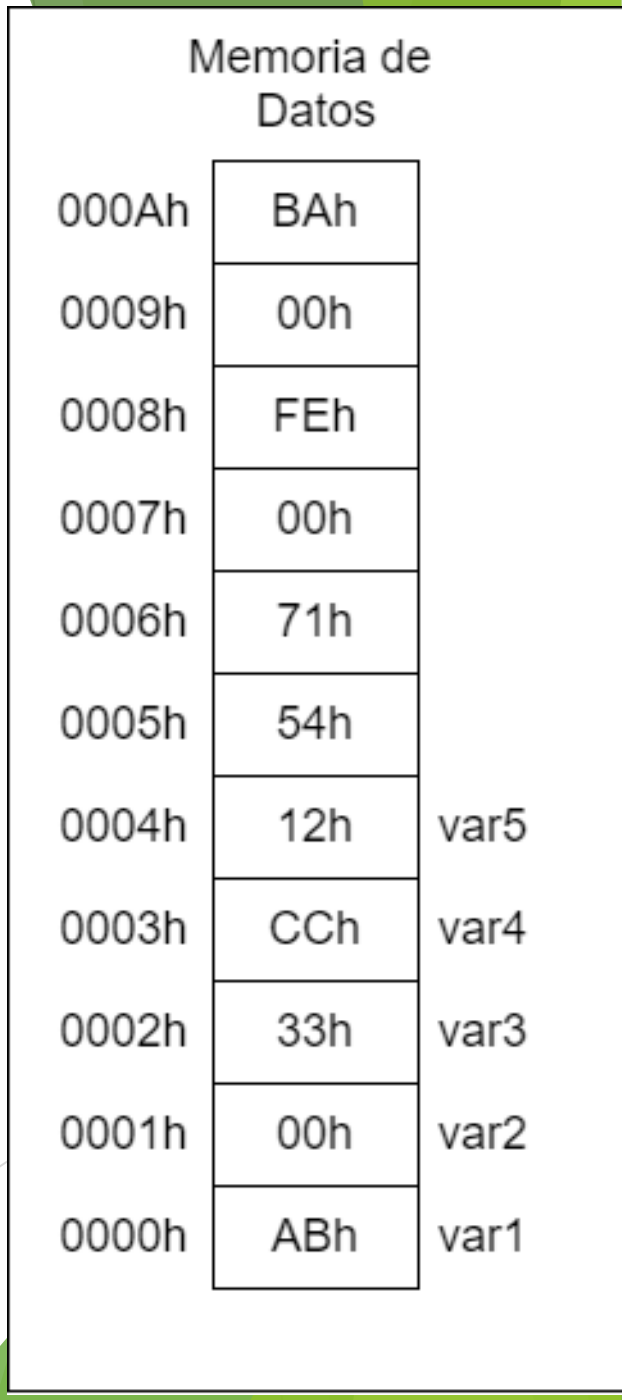
AX = 0000h
BX = 0001h
CX = 0002h
DI = 0003h
SI = 0002h



Ejemplo de modos de direccionamiento de memoria de datos

AX = 0005h
BX = 0001h
CX = 0002h
DI = 0003h
SI = 0002h

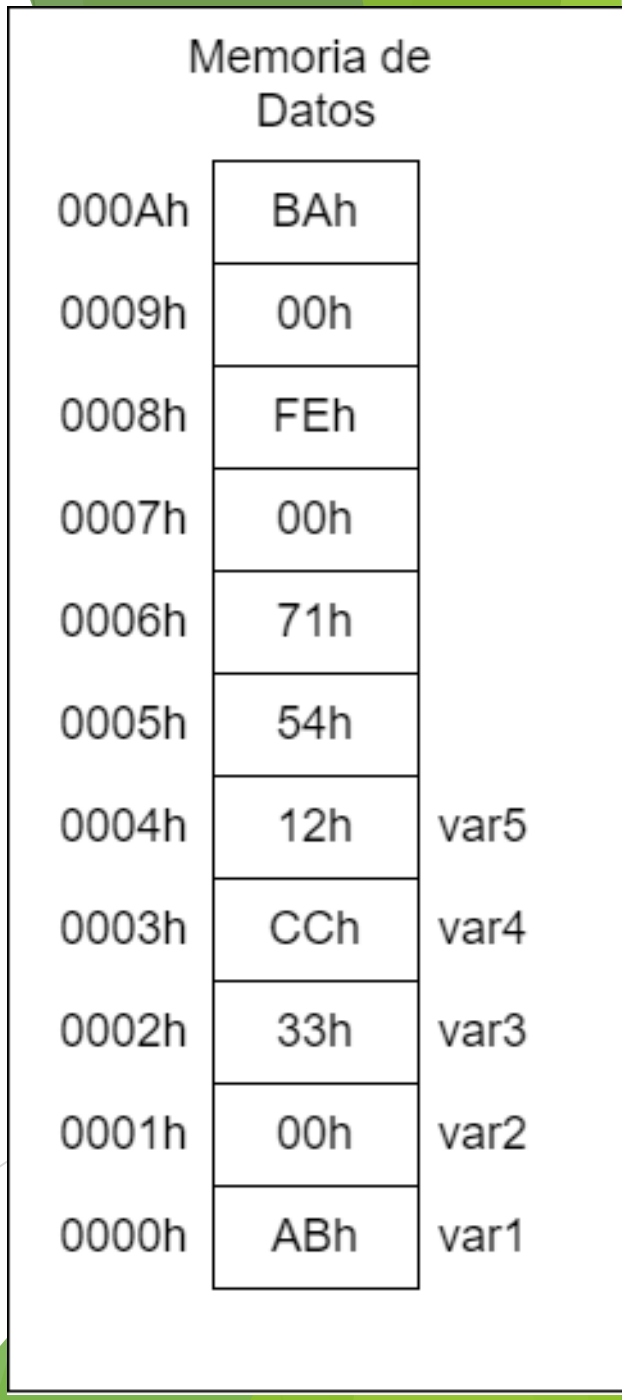
Instrucción	Resultado	Modo de Direccionamiento
mov ax, 5d	AX = 0005h	Inmediato
mov cl, [var3]		
mov cx, ax		
mov ah, [bx+di]		
mov [bx], ah		
mov ch, [bx+5]		
mov ax, [bx+si+2]		



Ejemplo de modos de direccionamiento de memoria de datos

Instrucción	Resultado	Modo de Direccionamiento
mov ax, 5d	AX = 0005h	Inmediato
mov cl, [var3]	CX = 0033h	Directo
mov cx, ax		
mov ah, [bx+di]		
mov [bx], ah		
mov ch, [bx+5]		
mov ax, [bx+si+2]		

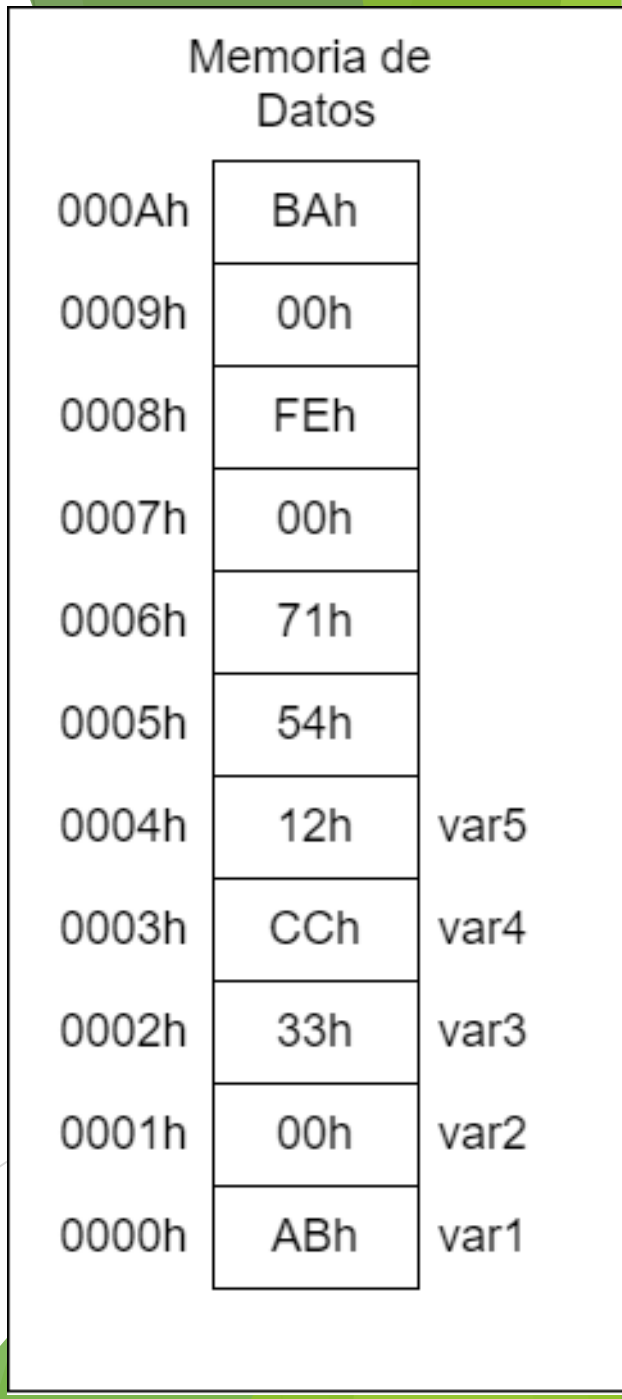
AX = 0005h
BX = 0001h
CX = 0033h
DI = 0003h
SI = 0002h



Ejemplo de modos de direccionamiento de memoria de datos

AX = 0005h
BX = 0001h
CX = 0005h
DI = 0003h
SI = 0002h

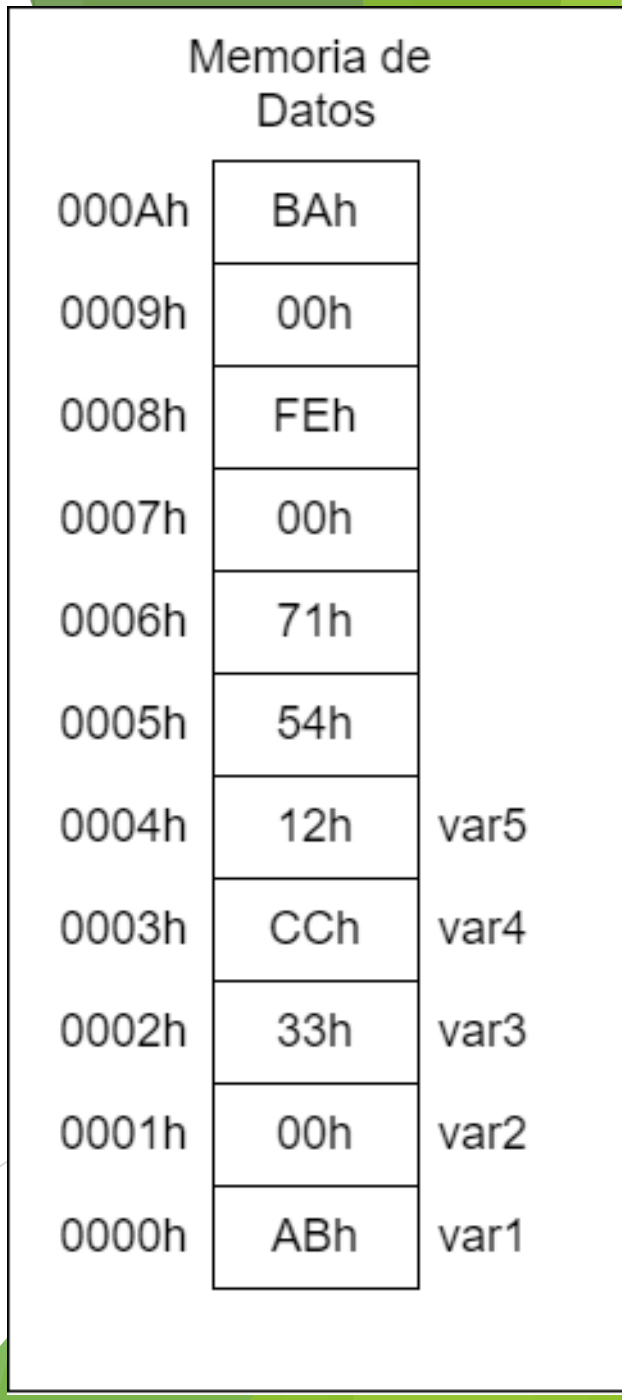
Instrucción	Resultado	Modo de Direccionamiento
mov ax, 5d	AX = 0005h	Inmediato
mov cl, [var3]	CX = 0033h	Directo
mov cx, ax	CX = 0005h	De registros
mov ah, [bx+di]		
mov [bx], ah		
mov ch, [bx+5]		
mov ax, [bx+si+2]		



Ejemplo de modos de direccionamiento de memoria de datos

Instrucción	Resultado	Modo de Direccionamiento
mov ax, 5d	AX = 0005h	Inmediato
mov cl, [var3]	CX = 0033h	Directo
mov cx, ax	CX = 0005h	De registros
mov ah, [bx+di]	AX = 1205h	De base más índice
mov [bx], ah		
mov ch, [bx+5]		
mov ax, [bx+si+2]		

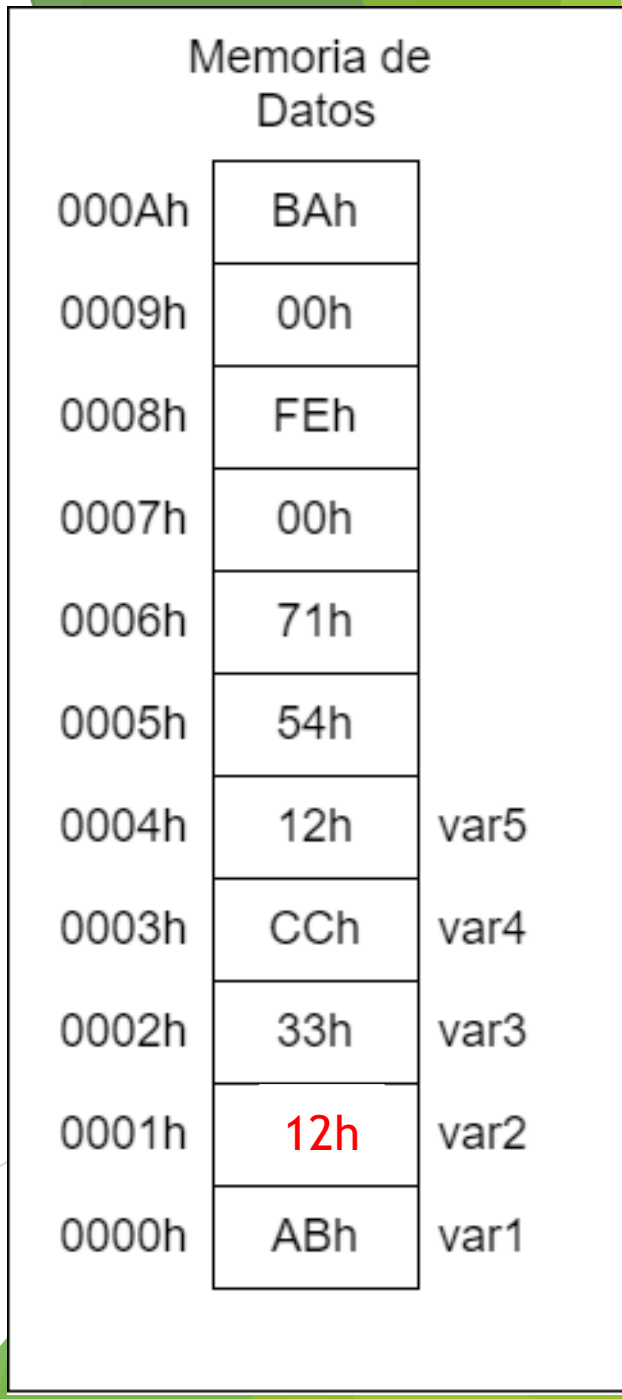
AX = 1205h
BX = 0001h
CX = 0005h
DI = 0003h
SI = 0002h



Ejemplo de modos de direccionamiento de memoria de datos

Instrucción	Resultado	Modo de Direccionamiento
mov ax, 5d	AX = 0005h	Inmediato
mov cl, [var3]	CX = 0033h	Directo
mov cx, ax	CX = 0005h	De registros
mov ah, [bx+di]	AX = 1205h	De base más índice
mov [bx], ah	[0001h] = 12h	De registro indirecto
mov ch, [bx+5]		
mov ax, [bx+si+2]		

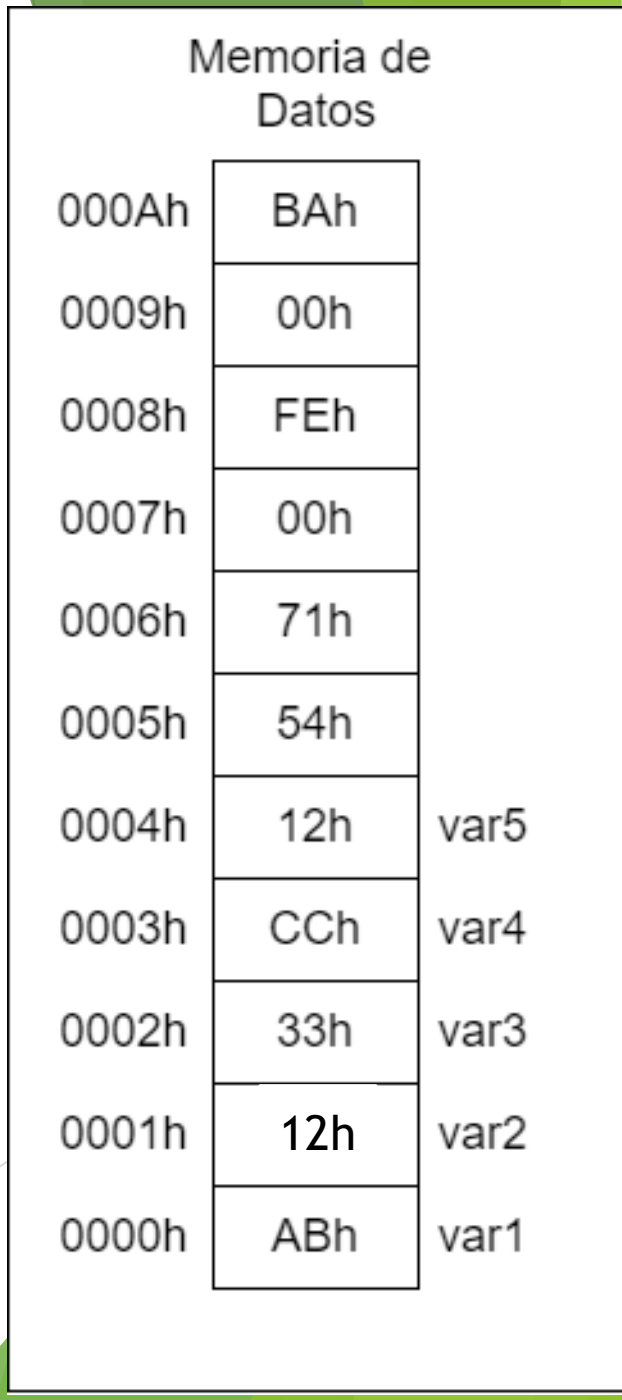
AX = 1205h
BX = 0001h
CX = 0005h
DI = 0003h
SI = 0002h



Ejemplo de modos de direccionamiento de memoria de datos

AX = 1205h
BX = 0001h
CX = 7105h
DI = 0003h
SI = 0002h

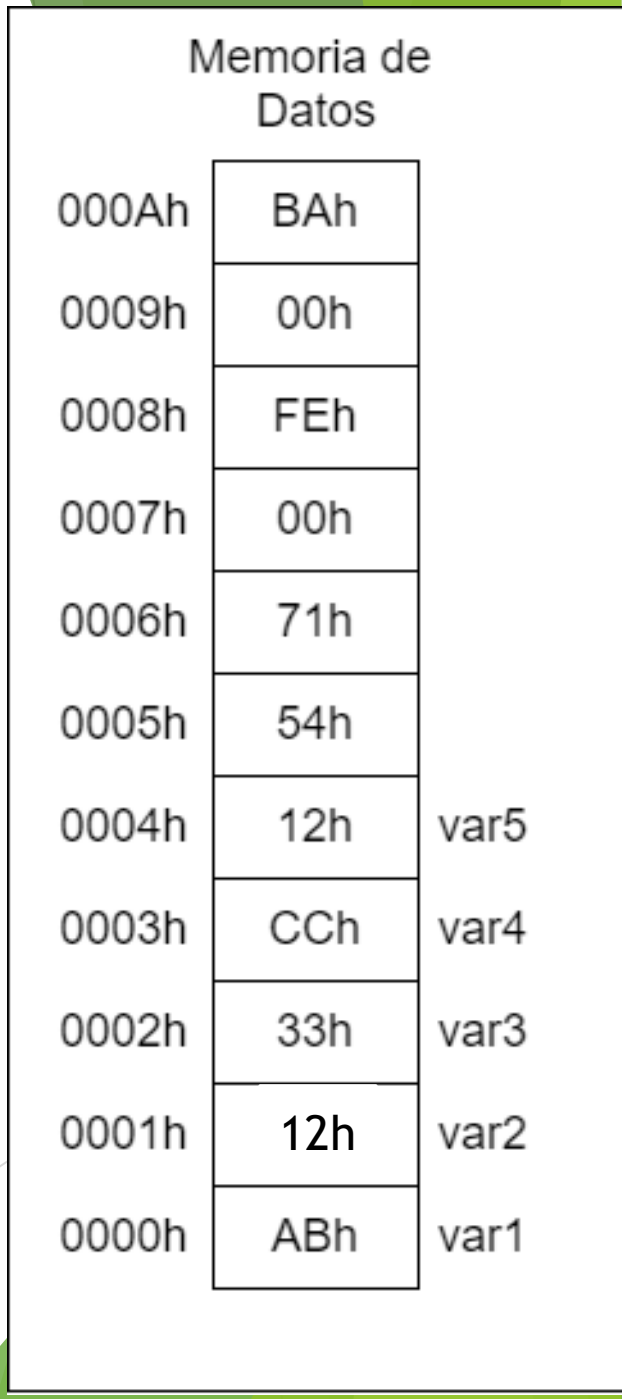
Instrucción	Resultado	Modo de Direccionamiento
mov ax, 5d	AX = 0005h	Inmediato
mov cl, [var3]	CX = 0033h	Directo
mov cx, ax	CX = 0005h	De registros
mov ah, [bx+di]	AX = 1205h	De base más índice
mov [bx], ah	[0001h] = 12h	De registro indirecto
mov ch, [bx+5]	CX = 7105h	De registro relativo
mov ax, [bx+si+2]		



Ejemplo de modos de direccionamiento de memoria de datos

AX = 7154h
BX = 0001h
CX = 7105h
DI = 0003h
SI = 0002h

Instrucción	Resultado	Modo de Direccionamiento
mov ax, 5d	AX = 0005h	Inmediato
mov cl, [var3]	CX = 0033h	Directo
mov cx, ax	CX = 0005h	De registros
mov ah, [bx+di]	AX = 1205h	De base más índice
mov [bx], ah	[0001h] = 12h	De registro indirecto
mov ch, [bx+5]	CX = 7105h	De registro relativo
mov ax, [bx+si+2]	AX = 7154h	De base relativa más índice



2.3 Conjunto de Instrucciones Lenguaje ensamblador

Lenguaje ensamblador $\neq$ Ensamblador

Lenguaje de programación a bajo nivel (acceso directo al hardware) que está constituido de mnemónicos.

Traductor del lenguaje ensamblador

Nmenónicos son palabras o conjuntos de caracteres que facilitan la comprensión y el entendimiento del lenguaje. Un mnemónico en lenguaje ensamblador sustituye a una operación en lenguaje máquina.

Lenguaje ensamblador

- Está específicamente diseñado para cada procesador, es decir, cada procesador tiene su propio lenguaje ensamblador.
- Cada instrucción en lenguaje ensamblador representa una instrucción en lenguage máquina (0s y 1s).

Lenguaje ensamblador

- Características
  - Es difícil de leer y entender
  - Difícil de aprender
  - Mantenimiento complicado
  - Difícil de escribir
  - Su velocidad de ejecución puede ser hasta 10 veces más rápido que un lenguaje de alto nivel
  - Optimiza espacio de memoria
  - Capacidad, se pueden implementar funciones difíciles o imposibles en lenguaje de alto nivel

Tipos de datos

- Se clasifican por su tamaño (longitud de bits)
  - Byte - 8 bits

```
var1    db      12h
```

  - Word - 16 bits = 2 bytes

```
var1    dw      1234h
```

  - Doubleword – 32 bits = 4 bytes

```
var1    dd      12345678h
```

  - Quadword – 64 bits = 8 bytes

```
var1    dq      123456789ABCDEF0h
```

Conjunto de instrucciones

- Operaciones de transferencia
  - MOV, LEA, PUSH, POP
- Operaciones aritméticas
  - ADD, SUB, INC, DEC, ADC, CMP, MUL, DIV
- Operaciones lógicas
  - AND, OR, NOT, XOR, TEST
- Operaciones de control de flujo
  - LOOP, CALL, JMP, JA, JAE, JE, JNE, JC, JNC, JZ, JNZ, etc.

Operaciones de transferencia

Operaciones de transferencia
MOV - *Move*

- Función

Copia el valor del operando origen al destino

- Sintaxis

```
mov   destino,   origen
        {reg/mem}  {reg/mem/constante}
```

- Ejemplos

```
mov  ax, 1234h
mov  [var1], ax
mov  bx, [var1]
```

- Banderas afectadas

El valor de las banderas C, S, Z, A, P, y O quedan indefinidos (puede ser 0 o 1).

Operaciones de transferencia
LEA – *Load Effective Address*

- Función

Obtiene la dirección de desplazamiento de algún símbolo (variable o etiqueta) utilizado dentro del código fuente.

- Sintaxis

```
lea    destino,  origen
       {reg}     {símbolo}
```

- Ejemplos

```
lea  ax, [et1]          ;AX = localidad de memoria del segmento de código
lea  bx, [var1]         ;BX = localidad de memoria del segmento de datos
```

- Banderas afectadas

No afecta al registro de banderas

Operaciones de transferencia
PUSH – *Push to stack*

- Función

Introduce un valor de 2 bytes al tope de la pila de programa. Modifica el valor del registro SP.

- Sintaxis

push origen
{reg/mem/constante} (16 bits)

- Ejemplos

```
push 1234h        ;constante de 16 bits
push ax           ;registro de 16 bits
push [var1]       ;var1 debe ser de tipo word
```

- Banderas afectadas

No afecta al registro de banderas

Operaciones de transferencia
POP – *Pop from stack*

- Función

Saca un valor de 2 bytes del tope de la pila de programa y lo almacena en el destino. Modifica el valor del registro SP.

- Sintaxis

pop destino
{reg/mem} (16 bits)

- Ejemplos

```
pop ax          ;pasa lo que hay en el tope de la pila a AX
pop [var1]      ;pasa lo que hay en el tope de la pila a [var1] (de tipo word)
```

- Banderas afectadas

No afecta al registro de banderas

Operaciones aritméticas

Operaciones aritméticas
ADD - *Addition*

- Función

Suma dos números binarios. Suma el operando destino con el operando origen y guarda el resultado en el operando destino.
Ambos operandos deben ser del mismo tamaño.

- Sintaxis

```
add   destino,  origen
      {reg/mem} {reg/mem/constante}
```

- Ejemplos

```
add  ax, 1234h        ;AX = AX + 1234h
add  [var1], ah       ;[var1] = [var1] + AH, var1 debe ser de tipo byte
add  bx, [var2]       ;BX = BX + [var2], var2 debe ser de tipo word
```

- Banderas afectadas

El valor de las banderas C, S, Z, A, P, y O se modifica de acuerdo al resultado.

Operaciones aritméticas
SUB - *Substraction*

- Función

Resta de dos números binarios. Resta el operando destino menos el operando origen y guarda el resultado en el operando destino.
Ambos operandos deben ser del mismo tamaño.

- Sintaxis

```
sub    destino,   origen
       {reg/mem}  {reg/mem/constante}
```

- Ejemplos

```
sub  ax, 1234h          ;AX = AX - 1234h
sub  [var1], ah         ;[var1] = [var1] - AH, var1 debe ser de tipo byte
sub  bx, [var2]         ;BX = BX - [var2], var2 debe ser de tipo word
```

- Banderas afectadas

El valor de las banderas C (borrow), S, Z, A, P, y O se modifica de acuerdo al resultado.

Operaciones aritméticas
INC - *Increment*

- Función

Realiza un incremento de 1 al operando destino.

- Sintaxis

```
inc     destino
        {reg/mem}
```

- Ejemplos

```
inc   ax            ;AX = AX + 1
inc   bl            ;BL = BL + 1
inc [var1]          ;[var1] = [var1] + 1, var1 puede ser de cualquier tipo
```

- Banderas afectadas

El valor de las banderas S, Z, P, y O se modifica de acuerdo al resultado.
No se afectan C ni A.

Operaciones aritméticas
DEC - *Decrement*

- Función

Realiza un decremento de 1 al operando destino.

- Sintaxis

```
dec    destino
       {reg/mem}
```

- Ejemplos

```
dec   ax          ;AX = AX + 1
dec   bl          ;BL = BL + 1
dec   [var1]      ;[var1] = [var1] + 1, var1 puede ser de cualquier tipo
```

- Banderas afectadas

El valor de las banderas S, Z, P, y O se modifica de acuerdo al resultado.
No se afectan C ni A.

Operaciones aritméticas
ADC – *Addition with carry*

- Función

Suma dos números binarios más el bit de carry. Suma el operando destino con el operando origen y el valor del bit C, guarda el resultado en el operando destino. Operandos destino y origen deben ser del mismo tamaño.

- Sintaxis

```
adc   destino,  origen
      {reg/mem} {reg/mem/constante}
```

- Ejemplos

```
adc  ax, 1234h        ;AX = AX + 1234h + C
adc  [var1], ah       ;[var1] = [var1] + AH + C, var1 debe ser de tipo byte
adc  bx, [var2]       ;BX = BX + [var2] + C, var2 debe ser de tipo word
```

- Banderas afectadas

El valor de las banderas C, S, Z, A, P, y O se modifica de acuerdo al resultado.

Operaciones aritméticas
SBB – *Substraction with borrow*

- Función

Resta de dos números binarios menos el bit de carry. Resta el operando destino menos el operando origen y menos el valor del bit C, guarda el resultado en el operando destino.
Operandos destino y origen deben ser del mismo tamaño.

- Sintaxis

```
sbb    destino,  origen
       {reg/mem} {reg/mem/constante}
```

- Ejemplos

```
sbb  ax, 1234h          ;AX = AX - 1234h - C
sbb  [var1], ah         ;[var1] = [var1] – AH - C, var1 debe ser de tipo byte
sbb  bx, [var2]         ;BX = BX - [var2] - C, var2 debe ser de tipo word
```

- Banderas afectadas

El valor de las banderas C (borrow), S, Z, A, P, y O se modifica de acuerdo al resultado.

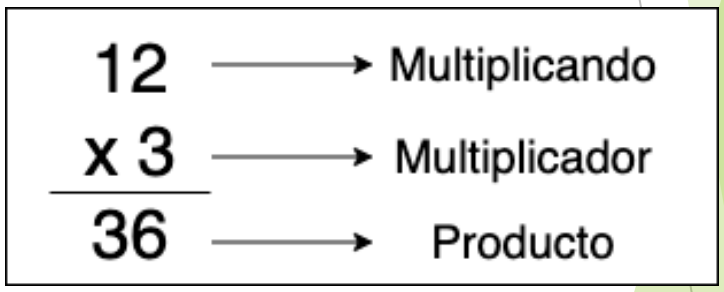
Operaciones aritméticas
MUL – (Unsigned) *Multiplication*

- Función

Realiza la multiplicación sin signo entre un multiplicando (implícito) y un operando multiplicador (explícito). El resultado se almacena en AX si los operandos son de 8 bits, o en DX y AX si los operandos son más grandes.

- Sintaxis

mul multiplicador
{8 bits: reg/mem}
{16 bits: reg/mem}



Multiplicador	Multiplicando (implícito)	Producto
reg/mem (8 bits)	registro AL (8 bits)	AX (16 bits)
reg/mem (16 bits)	registro AX (16 bits)	DX:AX (32 bits)
reg/mem (32 bits)*	registro EAX (32 bits)	EDX:EAX (64 bits)

*existe pero no se abarcará en clase

Operaciones aritméticas
MUL – (Unsigned) *Multiplication*

- ¿Por qué el producto es el doble de tamaño que los operandos?

Supongamos dos números binarios del mismo tamaño (4 bits), donde el valor máximo de cada uno es 1111.

```
    1111
  x 1111
  ------
    1111
+   1111
   1111
  1111
  ------
 11100001
```

Haciendo la multiplicación manualmente del valor máximo de cada números se puede observar que el resultado requiere 8 bits.

Asumimos que en el peor caso requeriremos 8 bits al hacer la multiplicación, a partir del máximo obtenido de la multiplicación, consideramos la longitud de ese resultado como base para los demás valores, pues éstos se podrán representar con los mismos 8 bits.

Operaciones aritméticas
MUL – (Unsigned) *Multiplication*

- Ejemplos

```
      .data
var1      db        20h
var2      dw        1234h
      .code

...

mov al, 0FFh      ;AL = FFh
mov bl, 0FFh      ;BL = FFh
mul  bl           ;AX = BL * AL = FFh * FFh = FE01h
...
mov al, 35h       ;AL = 35h
mul  [var1]       ;AX = [var1] * AL = 20h * 35h = 06A0h

...

mov ax, 1200h     ;AX = 1200h
mul  [var2]       ;DX:AX = [var2] * AX = 1234h * 1200h = 0147A800h
                  ;DX = 0147h, AX = A800h
```

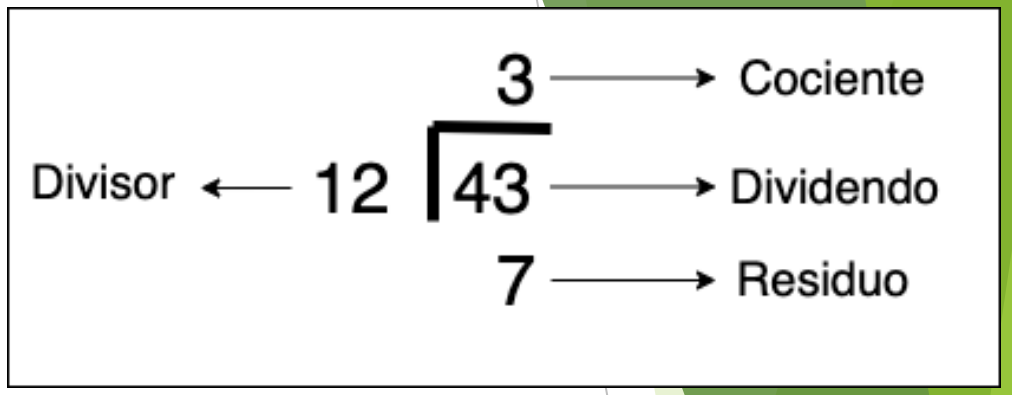
Operaciones aritméticas
MUL – (Unsigned) *Multiplication*

Banderas afectadas

c: } Si la parte alta es diferente de 0, entonces C = 1 y O = 1
o: } Si la parte alta es igual a 0, entonces C = 0 y O = 0

z: ?
p: ?
s: ?
a: ?

Operaciones aritméticas
DIV – (Unsigned) *Division*



- Función

Realiza la división sin signo entre un dividendo (implícito) y un operando divisor (explícito). El resultado de la operación constará de dos partes: el cociente y el residuo. El cociente se almacena en la parte baja y el residuo en la parte alta. Dependiendo el tamaño de los operandos, el resultado se guardará en AX y DX (ver tabla a continuación).

- Sintaxis

div divisor
{8 bits: reg/mem}
{16 bits: reg/mem}

Divisor	Dividendo (implícito)	Cociente	Residuo
reg/mem (8 bits)	registro AX (16 bits)	AL (8 bits)	AH (8 bits)
reg/mem (16 bits)	registro DX:AX (32 bits)	AX (16 bits)	DX (16 bits)
reg/mem (32 bits)*	registro EDX: EAX (64 bits)	EAX (32 bits)	EDX (32 bits)

*existe pero no se abarcará en clase

Operaciones aritméticas
DIV – (Unsigned) *Division*

- Ejemplos

```
      .data
var1        db          20h
var2        dw          1234h
      .code

...

mov ax, 0FE02h     ;AX = FE02h
mov bl, 0FFh       ;BL = FFh
div  bl            ;AL = AX / BL = FE02h / FFh = FFh
                   ;AH = AX % BL = FE02h % FFh = 01h
...
mov ax, 06A5h      ;AX = 06A5h
div  [var1]        ;AL = AX / [var1] = 06A5h / 20h = 35h
                   ;AH = AX % [var1] = 06A5h % 20h = 05h

...

mov dx, 0147h      ;DX = 0147h
mov ax, 0A80Ah     ;AX = A80Ah
div  [var2]        ;AX = DX:AX / [var2] = 0147A80Ah / 1234h = 1200h
                   ;DX = DX:AX % [var2] = 0147A80Ah % 1234h = 000Ah
```

Operaciones aritméticas
DIV – (Unsigned) *Division*

- Banderas afectadas

c: ?
o: ?
z: ?
p: ?
s: ?
a: ?

NOTA: División entre 0 produce una interrupción

Operaciones aritméticas
CMP – *Arithmetic comparison*

- Función

Compara dos números binarios restándolos. Resta el operando destino menos el operando origen. A diferencia de SUB, el resultado se descarta.

- Sintaxis

```
cmp   destino,   origen
        {reg/mem}  {reg/mem/constante}
```

- Ejemplos

```
cmp  ax, 1234h        ;AX - 1234h, afecta banderas
cmp  [var1], ah       ;[var1] - AH, var1 debe ser de tipo byte, afecta banderas
cmp  bx, [var2]       ;BX - [var2], var2 debe ser de tipo word, afecta banderas
```

- Banderas afectadas

El valor de las banderas C (borrow), S, Z, A, P, y O se modifica de acuerdo al resultado.

Operaciones lógicas

IMPORTANTE: Las operaciones lógicas se realizan bit a bit y los operandos deben ser del mismo tamaño

Operaciones lógicas
AND – *Logical AND*

- Función

Conjunción lógica entre dos operandos. Se realiza la conjunción entre un operando destino y un operando origen, el resultado se almacena en el operando destino.

- Sintaxis

and destino, origen
{reg/mem} {reg/mem/constante}

- Ejemplos

```
and  ax, 00FFh
and  [var1], ah
and  bx, [var2]
```

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

Banderas afectadas

c: 0
o: 0
s: } Dependen del resultado
z: }
p: }
a: Indefinido

Operaciones lógicas
OR – *Logical OR*

- Función

Disyunción lógica entre dos operandos. Se realiza la disyunción entre un operando destino y un operando origen, el resultado se almacena en el operando destino.

- Sintaxis

or destino, origen
{reg/mem} {reg/mem/constante}

- Ejemplos

```
or  al, 30h
or  [var1], ah
or  bh, al
or  bx, [var2]
```

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

Banderas afectadas

c: 0
o: 0
s: } Dependen del resultado
z:
p:
a: Indefinido

Operaciones lógicas
XOR – *Logical XOR (eXclusive OR)*

- Función

Disyunción exclusiva lógica entre dos operandos. Se realiza la disyunción exclusiva entre un operando destino y un operando origen, el resultado se almacena en el operando destino.

- Sintaxis

xor destino, origen
{reg/mem} {reg/mem/constante}

- Ejemplos

```
xor  ax, bx
xor  [var1], ah
xor  ax, 0000h
xor  bx, [var2]
```

A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

Banderas afectadas

c: 0
o: 0
s: } Dependen del resultado
z: }
p: }
a: Indefinido

Operaciones lógicas
NOT – *Logical NOT (Complemento a 1)*

- Función

Aplica el complemento a 1 a un operando. Se realiza el complemento a 1 del operando destino y se almacena el resultado en el mismo operando.

- Sintaxis

not destino
{reg/mem}

- Ejemplos

not ax
not [var1]

A	not A
0	1
1	0

Banderas afectadas

No afecta el registro de banderas

Operaciones lógicas
NEG – *Arithmetic NOT (Complemento a 2)*

- Función

Aplica el complemento a 2 a un operando. Se realiza el complemento a 2 del operando destino y se almacena el resultado en el mismo operando.

- Sintaxis

neg destino
{reg/mem}

- Ejemplos

neg ax
neg [var1]

Banderas afectadas

c: 0 y o: 0, si destino = 0
c: 1 y o: 1, si destino ≠ 0
s:
z:
p:
a:

Dependen del resultado

Operaciones lógicas
TEST – *Logical comparison AND*

- Función

Conjunción lógica entre dos operandos. Se realiza la conjunción entre un operando destino y un operando origen. A diferencia de AND, en TEST el resultado se descarta.

- Sintaxis

test destino, origen
{reg/mem} {reg/mem/constante}

- Ejemplos

```
test  ax, 00FFh
test  [var1], ah
test  bx, [var2]
```

Banderas afectadas

c: 0
o: 0
s: } Dependen del resultado
z: } Dependen del resultado
p: } Dependen del resultado
a: Indefinido

Operaciones de control de flujo

Operaciones de control de flujo
JMP - Salto incondicional

- Función

Cambia el flujo de ejecución de un programa hacia una localidad de memoria de código especificada.

- Sintaxis

```
jmp   etiqueta
```

- Ejemplos

```
      mov ax,10d
      add cx,ax
      jmp et1
      sub cx,bx
      inc cx
et1:
      mov [var1],cx
```

Banderas afectadas

No afecta el registro de banderas

Operaciones de control de flujo
LOOP - Salto condicional

- Función

- Ejecuta un ciclo dependiendo del valor de CX.
- LOOP revisa el contenido de CX, si CX ≠ 0, hace el salto y decrementa CX en 1.
- En un ciclo cerrado, LOOP se ejecutará CX veces

- Sintaxis

loop etiqueta

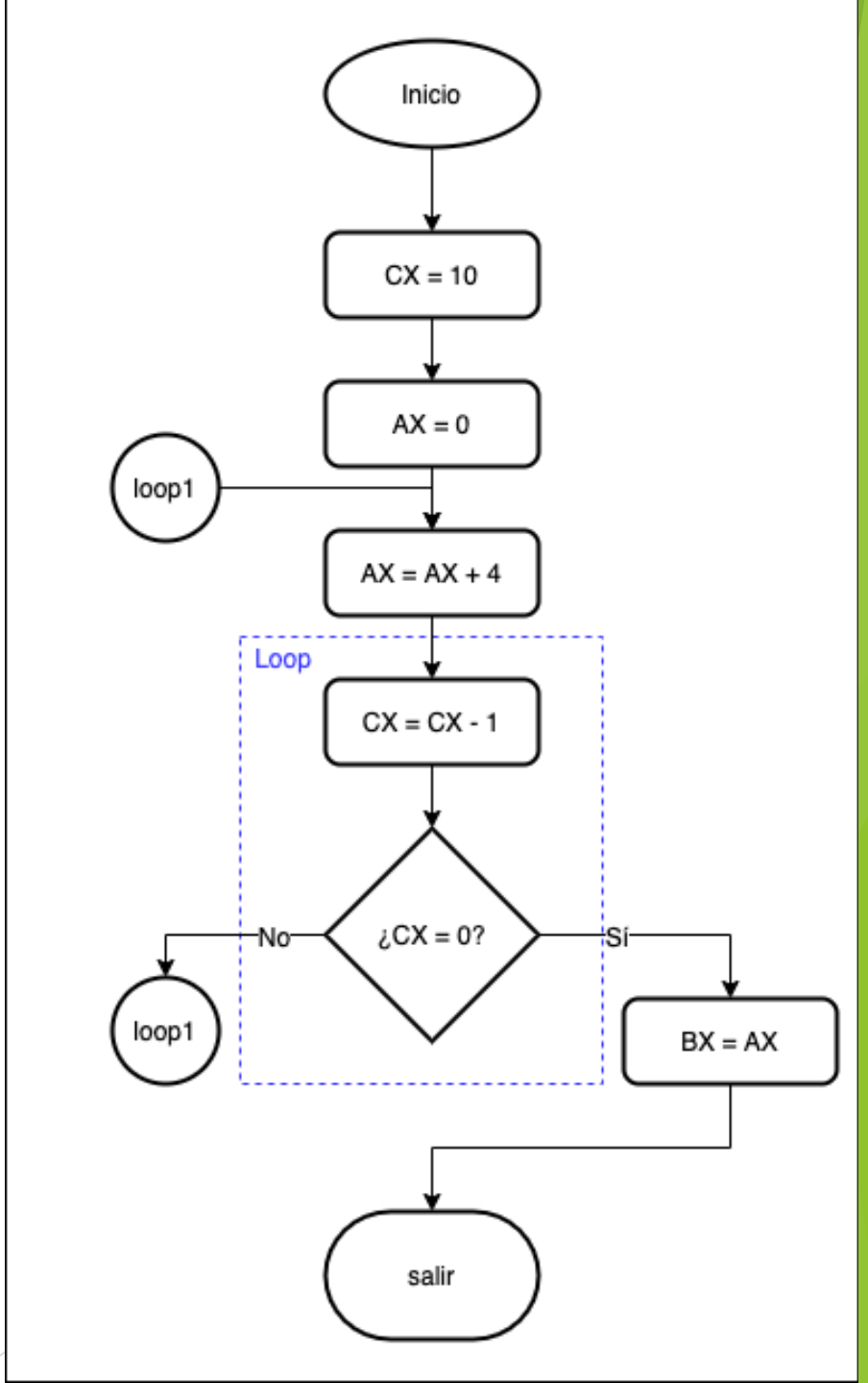
Banderas afectadas

No afecta el registro de banderas

Operaciones de control de flujo
LOOP - Salto condicional

Ejemplos

```
        ...
        mov ax,0        ;AX = 0000h
        mov cx,10d      ;CX = 000Ah
loop1:
        add ax,4        ;AX = AX + 4
        loop loop1      ;Si CX != 0, salta a 'loop1' y CX = CX - 1
        mov bx,ax       ;BX = AX
salir:
        ....
```



Operaciones de control de flujo
Jump if... – Saltos condicionales

- Función

Conjunto de **instrucciones de salto condicional**.

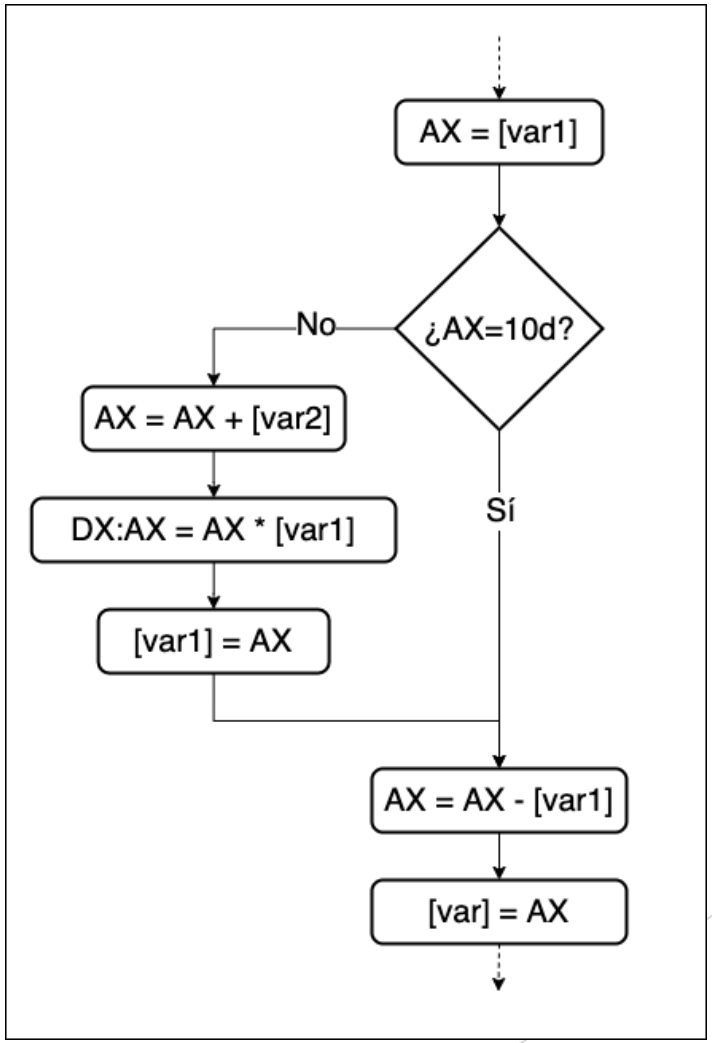
Estas instrucciones de salto condicional hacen **uso del registro de banderas** para validar condiciones. Regularmente, estas instrucciones de salto están precedidas por una instrucción CMP o TEST, aunque también se pueden preceder por cualquier instrucción aritmética o lógica.

Instrucción	Descripción
ja	Jump if above (C=0 y Z=0)
jae	Jump if above or equal (C=0)
jb	Jump if below (C=1)
jbe	Jump if below or equal (C=1 y Z=1)
jg	Jump if greater (Z=0 y S=0)
jge	Jump if greater or equal (S=O)
jc	Jump if carry flag set (C=1)
js	Jump if sign flag set (S=1)
jz	Jump if zero flag set (Z=1)
jo	Jump if overflow flag set (O=1)
je	Jump if equals (Z=1)
jnz	Jump if not zero set (Z=0)
...	

Operaciones de control de flujo
Jump if... - Saltos condicionales

- Ejemplo:

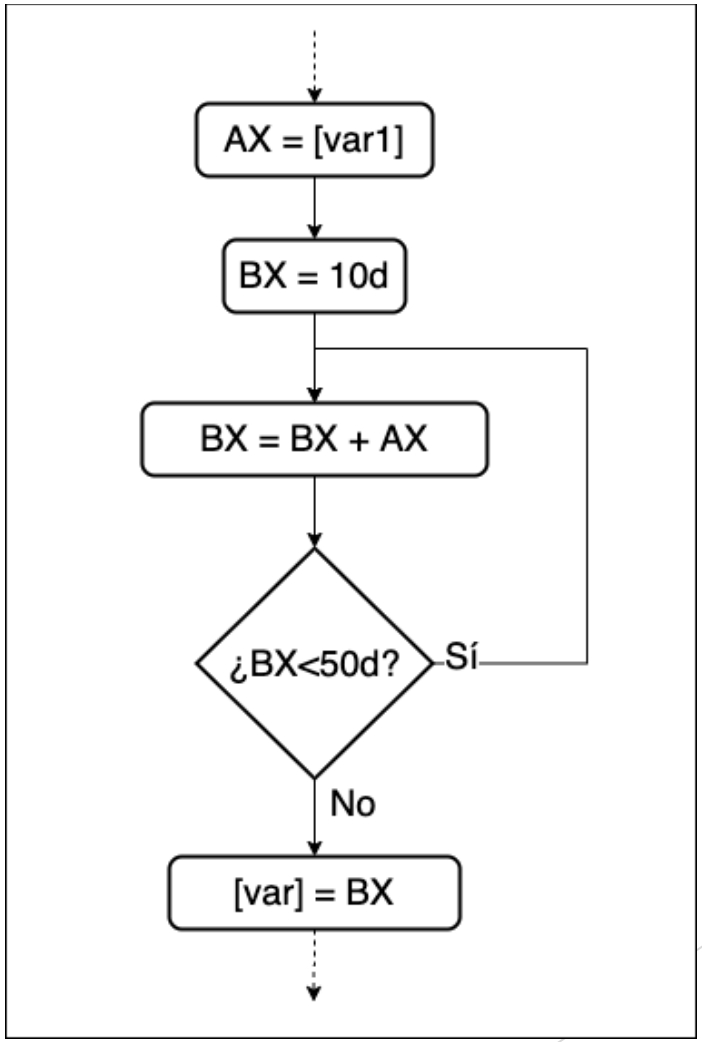
```
        ...
        mov ax,[var1]
        cmp ax,10
        je flujo
        add ax,[var2]
        mul [var1]
        mov [var1],ax
flujo:
        sub ax,[var1]
        mov [var1],ax
        ...
```



Operaciones de control de flujo
Jump if... - Saltos condicionales

- Ejemplo:

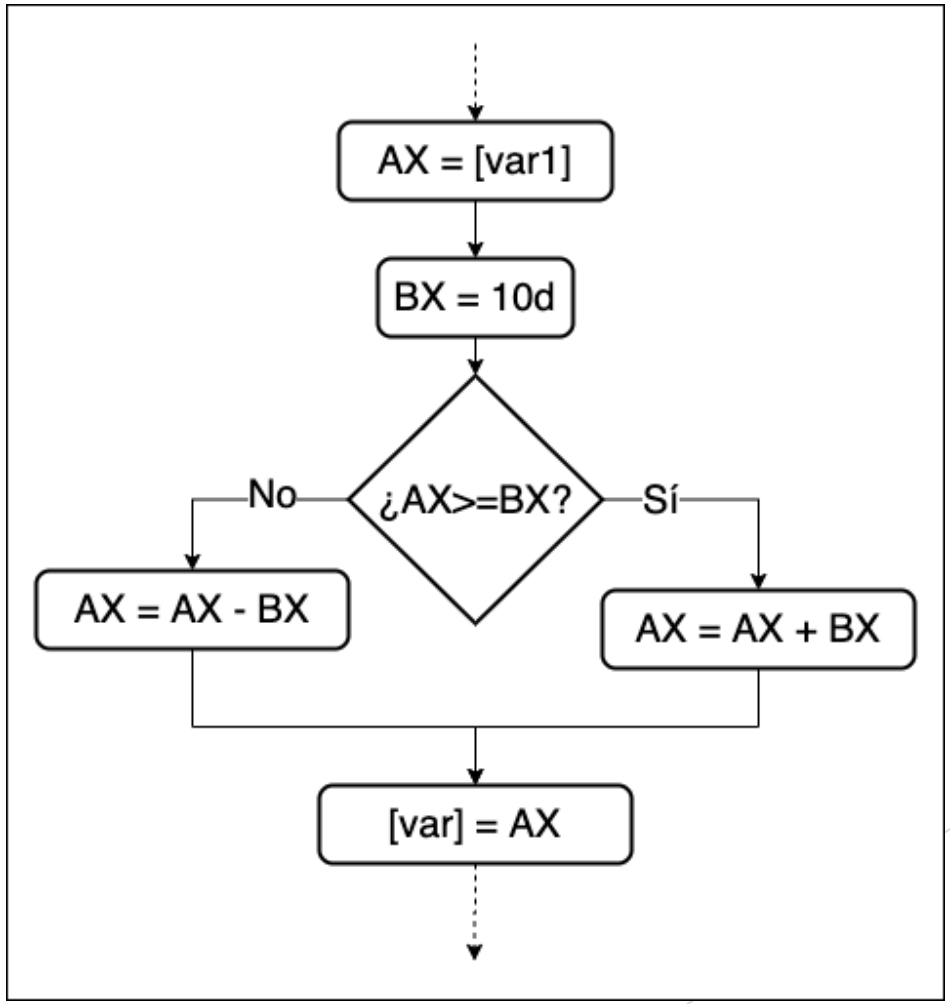
```
        ...
        mov ax,[var1]
        mov bx,10
flujo1:
        add bx,ax
        cmp bx,50
        jb flujo1
        mov [var],bx

        ...
```



Operaciones de control de flujo
Jump if... - Saltos condicionales

- Ejemplo:

```
        ...
        mov ax,[var1]
        mov bx,10
        cmp ax,bx
        jae flujo1
        sub ax,bx
        jmp flujo2
flujo1:
        add ax,bx
flujo2:
        mov [var],ax
        ...
```



Operaciones de control de flujo Procedimientos

- Procedimiento, o subrutina:
  - Es el equivalente a una función o método en lenguaje de alto nivel
  - Es una sección de código que se encuentra fuera del código principal y permite realizar una tarea bien definida.
  - Se utilizan para reducir líneas de código y cuando una tarea se realiza en reiteradas ocasiones.

Operaciones de control de flujo
CALL – Llamada a procedimientos

- Función

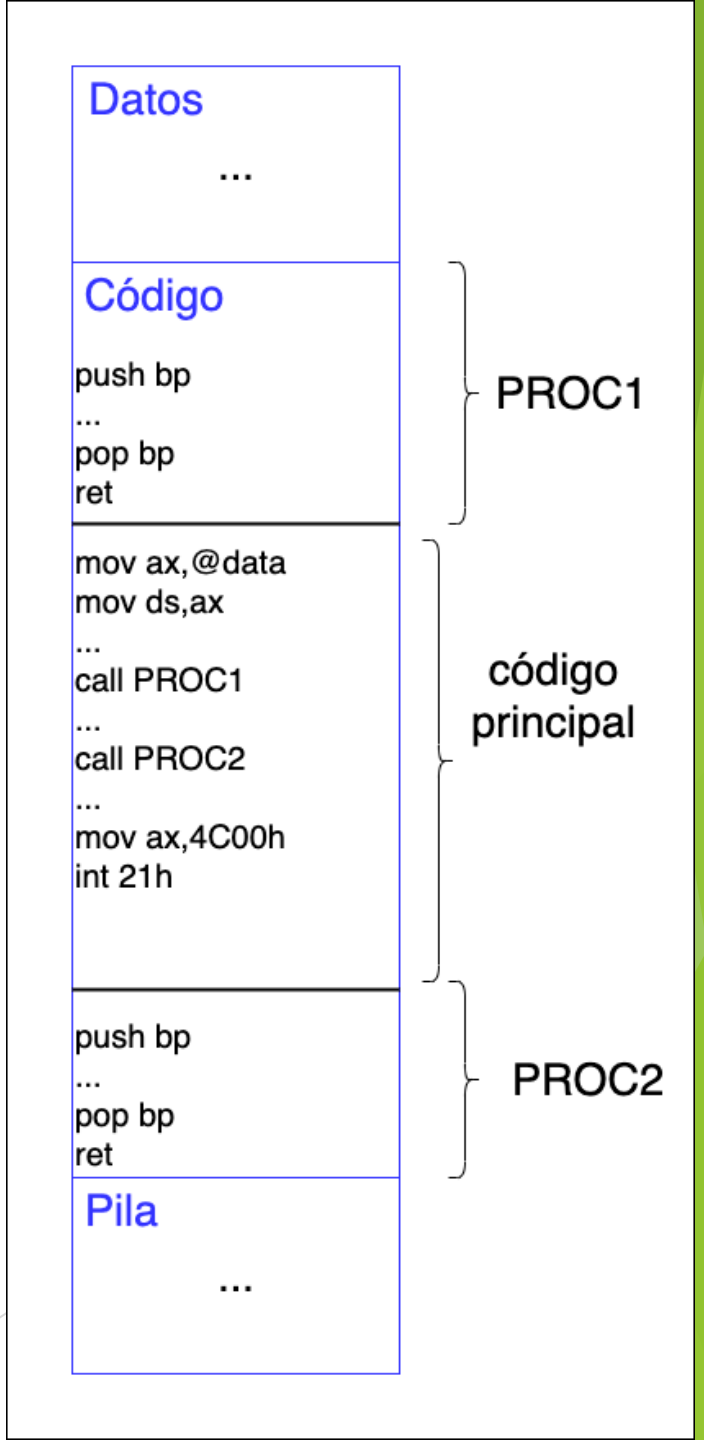
La instrucción CALL permite ejecutar un procedimiento que se encuentra en otra sección del segmento de código.

La instrucción CALL modifica el flujo de control del programa. Sin embargo, al finalizar el procedimiento, se devuelve el flujo utilizando la instrucción RET.

Internamente, la instrucción CALL almacena en la pila el contenido del registro IP (“push IP”) y lo modifica con la dirección de memoria del procedimiento para continuar el flujo en esa dirección. Una vez se ha terminado de ejecutar el procedimiento se deberá utilizar la instrucción RET que obtiene el valor en la cima de la pila y se guarda en IP (“pop IP”) para continuar el flujo antes de la instrucción CALL.

Operaciones de control de flujo CALL – Llamada a procedimientos

- Un procedimiento se define utilizando las directivas *proc* y *endp*. La definición de un procedimiento inicia con el nombre del procedimiento, seguido de la directiva *proc*. Para indicar el fin del procedimiento, se utiliza *endp*.
- La definición de un procedimiento se debe hacer dentro del segmento de código del programa.
- Antes de finalizar el procedimiento es importante utilizar la instrucción RET. RET permite devolver el valor al registro IP una vez que el procedimiento finaliza su ejecución para continuar con la ejecución del código principal.



Operaciones de control de flujo
CALL – Llamada a procedimientos

- Sintaxis

```
call   PROCEDIMIENTO
```

- Ejemplos

```
CUADRADO   proc
      mul al
      ret
      endp

...

mov al,5
call  CUADRADO
mov [resultado],ax
...
```

Banderas afectadas

No afecta el registro de banderas

CALL - Ejemplo de funcionamiento

```
title "Ejemplo Funcionamiento de Call"
    .model small
    .386
    .stack 64
    .data
resultado   dw  0
    .code
inicio:
    mov ax,@data
    mov ds,ax
    mov bx,25d
    call CUADRADO
salir:
    mov ah,4Ch
    mov al,0
    int 21h
CUADRADO proc
    mov ax,bx
    mul ax
    mov [resultado],ax
    ret
    endp
    end inicio
```

Dirección	Referencia	Instrucción	Dir. de la siguiente instrucción a ejecutar
0000h	inicio	mov ax,@data	IP = 0003h
0003h		mov ds,ax	IP = 0005h
0005h		mov bx,25d	IP = 0008h
0008h		call CUADRADO	IP = 0011h Pila = {000Bh}
000Bh	salir	mov ah,4Ch	IP = 000Dh
000Dh		mov al,0	IP = 000Fh
000Fh		int 21h	IP = ????, fin de programa
0011h	CUADRADO	mov ax,bx	IP = 0013h
0013h		mul ax	IP = 0015h
0015h		mov [resultado],ax	IP = 0018h
0018h		ret	IP = 000Bh Pila = { }

Operaciones de control de flujo
CALL – Llamada a procedimientos

- El paso de parámetros de un procedimiento se puede realizar de las siguientes 3 formas:
  - **Paso por variables:** Se utilizan variables definidas globalmente en el programa para accederlas dentro y fuera del procedimiento. Los valores de las variables se pueden modificar dentro del procedimiento.
  - **Paso por registros:** Los valores de los registros se pueden acceder dentro y fuera de un procedimiento. Es importante considerar que el valor de los registros se puede modificar dentro del procedimiento.
  - **Paso a través de la pila:** Se introducen valores en la pila antes de la llamada al procedimiento y se acceden dentro del procedimiento. Una vez termina el procedimiento, los valores se pueden recuperar o no de la pila.

Operaciones de control de flujo
CALL – Llamada a procedimientos

- Paso de parámetros por variables:

```
        .data
var1                db      0
num_cuadrado        dw      0
resultado           dw      ?

        .code

CUADRADO   proc
        mov al, [var1]
        mul al
        mov [num_cuadrado],ax
        ret
        endp

inicio:
        mov ax,@data
        mov ds,ax
        mov [var1],10d
        call CUADRADO
        mov bx,[num_cuadrado]
        mov [resultado],bx
...
```

- Paso de parámetros por registros:

```
        .data
resultado           dw      ?

        .code

CUADRADO   proc
        mul al
        ret
        endp

inicio:
        mov ax,@data
        mov ds,ax
        mov al,10d
        call CUADRADO
        mov [resultado],ax
...
```

Operaciones de control de flujo
CALL – Llamada a procedimientos

- Paso de parámetros a través de la pila:

```
        .data
resultado               dw          ?

        .code

CUADRADO   proc
        mov bp,sp
        mov ax,[bp+2]
        mul al
        mov [bp+2],ax
        ret
        endp

inicio:
        mov ax,@data
        mov ds,ax
        mov ax,10d
        push ax
        call CUADRADO
        pop ax
        mov [resultado],ax
...
```

Operaciones de control de flujo
CALL – Llamada a procedimientos

- Recomendaciones en el uso de procedimientos:
  - No modificar el valor de IP manualmente.
  - Dentro del procedimiento, al inicio y al final, la cima de la pila debe tener el valor de la dirección a la instrucción siguiente de CALL que hizo la llamada a el procedimiento en ejecución. De lo contrario, el flujo del programa se pierde.
  - Es posible anidar llamadas a procedimientos. Se recomienda que al inicio de cada procedimiento se almacene el contenido del registro BP en la pila, e inicializarlo con SP. Es decir, ejecutar las siguientes instrucciones al inicio del procedimiento:
    push bp
    mov bp,sp
    Del mismo modo, antes de finalizar el procedimiento con RET, recuperar los valores almacenados:
    mov sp,bp
    pop bp
  - Utilizar direccionamiento de registro relativo con BP dentro del procedimiento para acceder datos de la pila:
    mov ax,[bp+4]

Macroinstrucciones

Macroinstrucciones

- Una macroinstrucción, o **macro**, es una secuencia de instrucciones, a las cuales se les asigna un nombre y pueden ser usadas en cualquier parte de un programa.
- Una macro es similar a un procedimiento aunque conceptualmente son diferentes:
  - Un procedimiento es una sección de código que se escribe una sola vez y se puede utilizar tantas veces sea necesario. En memoria de código, se almacena una sola copia.
  - Una macro es una sección de código que el programador escribe una sola vez y puede utilizar tantas veces sea necesario. En memoria de código, se duplica la sección de código por cada ocasión que se utiliza la macro.

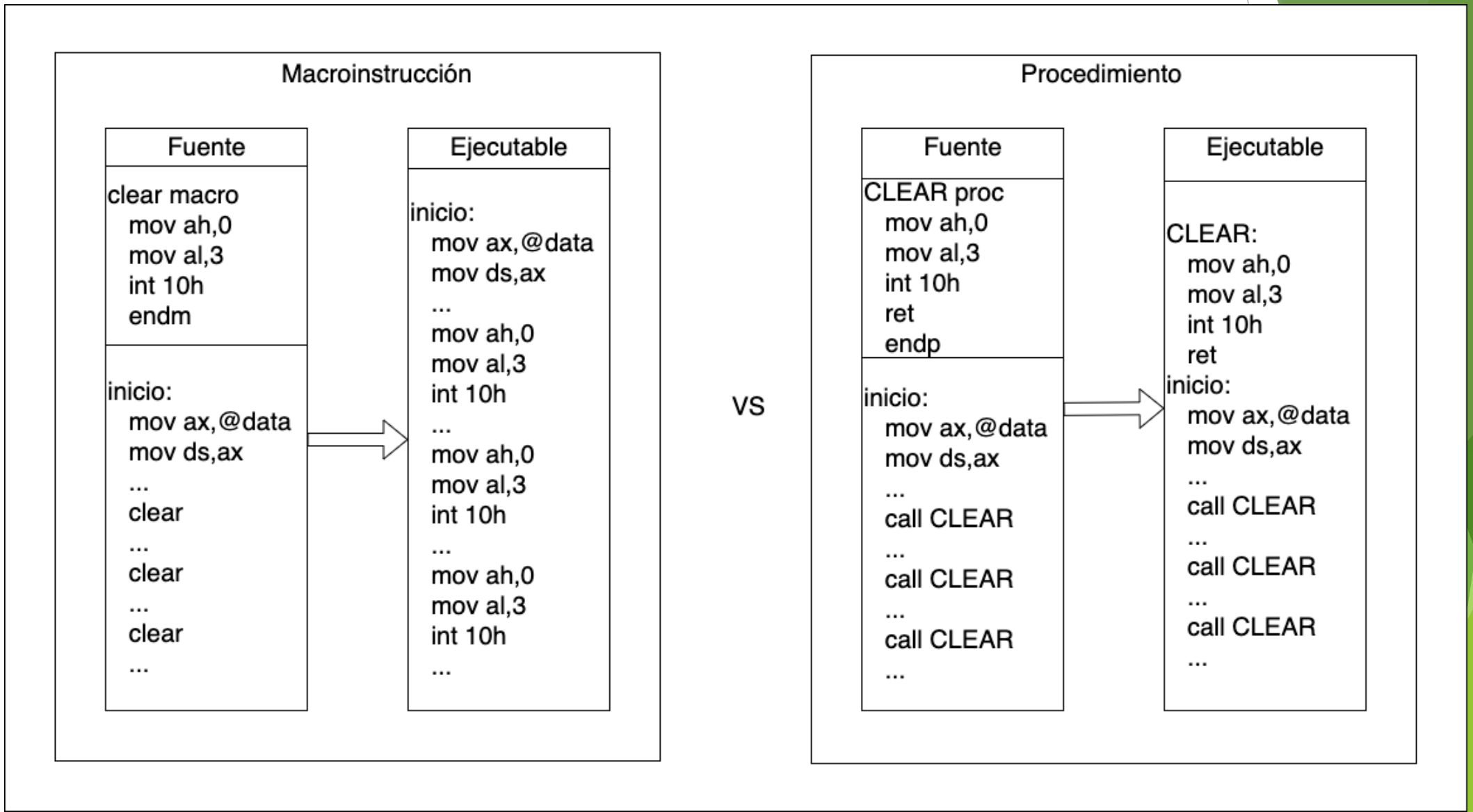
Macroinstrucciones

- Otra diferencia entre procedimiento y macro es que la primera se ejecuta con ayuda de una instrucción (CALL) y la segunda se maneja de manera diferente. Una macro se especifica con ayuda de una directiva que se ejecuta por el ensamblador.
- Cuando el ensamblador detecta el uso de una macro, la sustituye por la sección de código que representa. A este proceso se le denomina **macroexpansión.**
- La macro se define fuera de los segmentos de datos y de código, en el archivo fuente.

Macroinstrucciones

- El uso de macros se recomienda cuando existe un bloque pequeño de instrucciones que se requieren utilizar continuamente. Cuando el bloque de instrucciones es muy grande y se utilizan macros, por cada uso se duplicará el código, haciendo que se extienda.

Macroinstrucciones vs Procedimiento

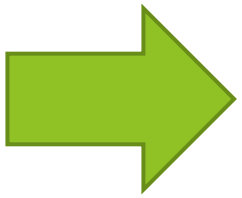


Macroinstrucciones

- Ejemplo: (código fuente)

```
        .model small
        .386
        .stack 64
;Macro que suma AX más BX y
;guarda el resultado en CX
suma_cx_ax_bx   macro
        add ax,bx
        mov cx,ax
endm
        .data
        .code
inicio:
...
        mov ax,25d
        mov bx,832d
        suma_cx_ax_bx
        mov ax,0B15h
        mov bx,5C3h
        suma_cx_ax_bx
...
```

Ensamblado



- Ejemplo: (ejecutable)

```
...
  mov ax,25d
  mov bx,832d
  add ax,bx
  mov cx,ax
  mov ax,0B15h
  mov bx,05C3h
  add ax,bx
  mov cx,ax
...
```

Macroinstrucciones

- Paso de parámetros

Se pueden agregar parámetros a una macro.

Para definir los parámetros de la macro, se agregan después de la directiva *macro* y separados por coma. Un ejemplo a continuación:

```
suma macro a,b
      mov ax,a
      mov bx,b
      add ax,bx
endm
```

La macro anterior tiene dos parámetros: *a* y *b*. El primer parámetro, *a*, se utiliza en la línea mov ax,a; y el segundo, *b*, se utiliza en la línea mov bx,b.

Un ejemplo de implementación de la macro anterior es el siguiente:

```
suma 10h,25h
```

Macroinstrucciones

- Paso de parámetros (continuación)

En la implementación de una macro, es posible pasar como argumentos: constantes, variables o registros. Sin embargo, es importante considerar que dentro de la macro, literalmente, se sustituye el nombre del parámetro por el argumento. Es decir, tomando como ejemplo la macro **suma**:

```
suma 10h,25h =>  mov ax,10h
                 mov bx,25h
                 add ax,bx
```

```
suma cx,dx =>  mov ax,cx
               mov bx,dx
               add ax,bx
```

```
suma [num1],[num2] =>  mov ax,[num1]
                       mov bx,[num2]
                       add ax,bx
```

...o cualquier combinación de las anteriores

NOTA:
Es importante considerar el tamaño de los argumentos de acuerdo al uso que se le da dentro de la macro