|  |
| --- |
| El profesor cerró la clase destacando que en la siguiente sesión se profundizará en:  🔹 Cómo el procesador maneja los procedimientos NEAR y FAR internamente.  🔹 Por qué se usa memoria específica para esto.  🔹 Cómo afecta esto al paso de parámetros en los procedimientos, que es un tema más complejo.  Lo que aprendimos hoy  📌 1. Diferencia entre procedimientos NEAR y FAR:  ✅ NEAR (RET) → Procedimiento en el mismo segmento.  ✅ FAR (RETF) → Procedimiento en otro segmento.  📌 2. Dónde declarar los procedimientos:  ✔ En segmento de código (Codigo Segment).  ❌ No en segmento de datos (Datos Segment), salvo casos especiales.  📌 3. Cómo evitar que un procedimiento se ejecute automáticamente:  ✔ Usando JMP Short inicio para saltarlo.  📌 4. Qué veremos en la siguiente clase:  ✔ Cómo el procesador maneja internamente los procedimientos NEAR y FAR.  ✔ Por qué la memoria juega un papel clave en estas llamadas.  ✔ Cómo pasar parámetros a un procedimiento (tema más avanzado). |

el uso de registros y memoria en ensamblador. Se menciona el registro **BX** y cómo asegurarse de que esté vacío usando la instrucción **XOR**. Luego, se intenta mover el contenido apuntado por **BX** al registro **AL**, pero se genera un error porque no se especifica el tamaño del dato que se quiere mover. Además, se hace una pregunta sobre el uso de los corchetes en ensamblador, confirmando que indican la dirección de memoria a la que se está apuntando en ese momento.

cómo se manejan los registros y la memoria en ensamblador. Se menciona que no se puede mover directamente el contenido de **BX** a **AL**, sino que debe ser a **AX**, porque **AX** es un registro de 16 bits. Al hacer esto, ya no se mueve un solo byte, sino una palabra (**word**, 16 bits), lo que afecta el manejo de posiciones de memoria.

Se hace una prueba para demostrar la diferencia y se observa que no da error porque el ensamblador asume que se está moviendo una palabra. Luego, se analiza el **DOM** para visualizar la memoria. Se explica que la posición **0** tiene el valor **FF**, pero la variable **tecla 2** está en la posición **1** y contiene **FE**. Para acceder correctamente a **tecla 2**, se debe asignar el valor **1** a **BX**, lo que se verifica en la ejecución.

cómo el ensamblador maneja el acceso a la memoria utilizando registros y el segmento de datos (**DS**). Se observa que **BX** contiene una dirección de memoria y se usa para acceder a su contenido. Si se mueve directamente **BX** a otro registro, simplemente se copia su valor. Sin embargo, si se quiere acceder a la dirección apuntada por **BX**, se debe indicar que se está trabajando con memoria.

El ensamblador automáticamente interpreta que se debe buscar en la dirección **DS:BX**. En este caso, como **BX** tiene el valor **0000**, el acceso se hará a la posición de memoria en **DS:0000**.

**Acceso a la memoria con DS y BX en ensamblador**

En ensamblador, cuando se usa **BX** para acceder a la memoria, se debe tener en cuenta que este registro no almacena directamente un valor, sino que puede contener una dirección de memoria.

**Diferencia entre mover un valor y acceder a la memoria**

1. **Mover directamente el contenido de BX a otro registro:**

MOV AX, BX

* + Aquí, el valor almacenado en **BX** se copia a **AX** sin modificar nada en la memoria.

1. **Acceder al contenido de la dirección apuntada por BX:**

assembly

MOV AL, [BX]

* + En este caso, se toma el valor que se encuentra en la dirección de memoria apuntada por **BX** y se guarda en **AL**.

**Uso del segmento de datos (DS)**

Cuando se accede a memoria en ensamblador, se usa el segmento de datos **DS** por defecto. Esto significa que cuando se usa [BX], el procesador realmente está accediendo a la dirección **DS:BX**.

Ejemplo práctico:

* Si **BX = 0000**, entonces [BX] accede a **DS:0000**.
* Si **BX = 0001**, entonces [BX] accede a **DS:0001**.

Por lo tanto, para leer correctamente una variable en memoria, **BX** debe contener la dirección correcta.

**Resumen:**

* **MOV AX, BX** → Copia el valor de BX a AX.
* **MOV AL, [BX]** → Lee el contenido de la dirección almacenada en BX.
* **DS se usa por defecto al acceder a la memoria con registros.**

**Orden de almacenamiento en memoria en ensamblador**

En ensamblador, los valores en memoria se almacenan en orden de **mayor a menor** cuando se interpretan en hexadecimal. Esto significa que los valores más grandes están en las posiciones de memoria inferiores, y los valores más pequeños en las posiciones superiores.

**Ejemplo de valores en memoria**

Si observamos la memoria, podríamos encontrar valores como estos:

| **Dirección** | **Valor (Hex)** | **Valor (Dec)** |
| --- | --- | --- |
| 0000 | FF | 255 |
| 0001 | FE | 254 |
| 0002 | FD | 253 |
| 0003 | FC | 252 |
| 0004 | 0F | 15 |

Esto significa que los valores se organizan de manera descendente en la memoria.

**Cómo se interpreta esto en ensamblador**

Cuando accedemos a la memoria en ensamblador, podemos ver esta estructura de almacenamiento. Si queremos leer valores desde una dirección específica, debemos considerar este orden.

Por ejemplo, si usamos:

MOV AL, [BX]

Y **BX = 0001**, **AL** tomará el valor **FE** (254 en decimal).

Si aumentamos **BX**, accederemos al siguiente valor en la secuencia:

INC BX

MOV AL, [BX] ; Ahora AL tendrá FD (253 en decimal)

* Los valores en memoria están organizados de mayor a menor.
* En hexadecimal: **FF → FE → FD → FC → 0F**
* En decimal: **255 → 254 → 253 → 252 → 15**
* Cuando se accede a la memoria en ensamblador, debemos tener en cuenta esta estructura para leer los valores correctamente.

**Acceso a la memoria y ordenamiento en Little Endian en ensamblador**

En ensamblador, la memoria se organiza en un formato llamado Little Endian, lo que significa que los bytes se almacenan en orden inverso: el byte menos significativo se almacena primero y el más significativo después.

Diferencia entre leer un byte y leer una palabra (word)

1. Lectura de un solo byte (Byte):

MOV AL, [BX]

* + Solo se extrae un byte desde la dirección apuntada por BX.

1. Lectura de una palabra (Word - 2 bytes):

MOV AX, [BX]

* + Aquí, se extraen dos bytes, el que está en BX y el siguiente en memoria.
  + Si BX = 0001, se leerán los valores en 0001 y 0002.
  + Debido al formato Little Endian, los bytes se colocarán en AX de manera invertida.

Ejemplo de cómo se leen los valores en memoria

Si la memoria contiene lo siguiente:

| Dirección | Valor (Hex) |
| --- | --- |
| 0000 | FF |
| 0001 | FE |
| 0002 | FD |
| 0003 | FC |

Y hacemos:

MOV AX, [BX] ; BX = 0001

* Se leerán los valores FE (0001) y FD (0002).
* Como la memoria usa Little Endian, AX tendrá el valor FDFE (FD primero, FE después).

Modificar la posición de BX sin alterar su contenido

Podemos acceder a otra posición de memoria sumando un valor a BX sin modificarlo:

MOV AX, [BX+2] ; Accede a la dirección BX + 2

* Si BX = 0001, ahora se leerán los valores FD (0003) y FC (0004).
* AX tendrá el valor FCFD (debido al formato Little Endian).

Conclusión

* Little Endian almacena los bytes en orden inverso (el menos significativo primero).
* MOV AX, [BX] extrae dos bytes y los organiza en el registro de manera invertida.
* BX puede manipularse para acceder a otras posiciones sin alterarlo directamente.

**Acceso a memoria con desplazamiento y efecto de Little Endian en ensamblador**

En este caso, estamos accediendo a la memoria en ensamblador utilizando desplazamientos en **BX** y observando cómo se extraen los valores según la estructura **Little Endian**.

**Acceso a la memoria con desplazamiento en BX**

Cuando se mueve una palabra (**word**, 2 bytes), se extrae el contenido de la dirección de memoria apuntada por **BX** y el siguiente byte inmediato.

Si tenemos la siguiente memoria:

| **Dirección** | **Valor (Hex)** |
| --- | --- |
| 0000 | FF |
| 0001 | FE |
| 0002 | FD |
| 0003 | FC |
| 0004 | 00 |
| 0005 | 0F |

Si hacemos:

MOV AX, [BX] ; BX = 0001

* Se leerán los valores en **0001 y 0002**.
* Como la memoria usa **Little Endian**, AX tendrá **FDFE** (FD primero, FE después).

Si luego aumentamos BX en 2:

MOV AX, [BX+2] ; Ahora BX apunta a 0003

* Se leerán los valores en **0003 y 0004**.
* AX tendrá **000F**, porque en memoria están almacenados como **0F 00** (ordenados al revés por Little Endian).

Para probar esto, se cambia el **00** por **07** y se vuelve a ejecutar la operación.

MOV AX, [BX+2] ; Ahora AX debería contener 0F07

**Conclusión**

* **BX+2 permite acceder a diferentes posiciones en memoria sin modificar BX.**
* **Little Endian** hace que los valores en registros se almacenen en orden inverso a como aparecen en memoria.
* **Al leer una palabra (word), se toman dos bytes consecutivos, lo que afecta el resultado según el contenido de la memoria.**

**Manipulación de direcciones de memoria en ensamblador: Suma y resta de desplazamientos**

Aquí estamos viendo cómo modificar la dirección almacenada en **BX** para acceder a diferentes posiciones de memoria.

**Acceso a memoria con desplazamiento positivo**

Si queremos mover un valor de **BX** sin modificar su contenido, podemos usar un desplazamiento en la instrucción:

MOV AX, [BX+2] ; Accede a BX + 2 sin modificar BX

Esto permite leer una palabra (**word**, 2 bytes) desde la nueva dirección sin afectar **BX**.

Si la memoria está organizada así:

| **Dirección** | **Valor (Hex)** |
| --- | --- |
| 0000 | FF |
| 0001 | FE |
| 0002 | FD |
| 0003 | FC |
| 0004 | 07 |
| 0005 | 0F |

Y **BX = 0003**, la instrucción leerá los valores en **0003 y 0004**, pero como la memoria usa **Little Endian**, AX tendrá el valor **070F**.

**Modificación directa de BX con suma y resta**

Si en lugar de solo acceder con desplazamiento queremos modificar BX, podemos usar **ADD** y **SUB**:

* **Sumar 2 a BX:**

ADD BX, 2 ; BX ahora es BX + 2

MOV AX, [BX] ; Accede a la nueva posición

Esto cambia **BX** directamente, moviendo el puntero en memoria.

* **Restar 2 a BX:**

SUB BX, 2 ; BX ahora es BX - 2

MOV AX, [BX] ; Accede a la posición anterior

Diferencia clave:

* **MOV AX, [BX+2]** → No altera **BX**, solo lee desde una posición diferente.
* **ADD BX, 2** / **SUB BX, 2** → Modifican el valor de **BX**, cambiando su referencia en memoria.

**Conclusión**

* **Podemos acceder a memoria con desplazamiento sin modificar BX.**
* **Podemos modificar BX directamente con ADD y SUB para movernos en la memoria.**
* **Little Endian siempre invierte los bytes al leer un Word.**

**Manipulación de memoria con suma, resta y limpieza de registros en ensamblador**

Aquí se está viendo cómo se puede sumar y restar valores en **BX** para cambiar la posición en memoria, además de cómo limpiar registros para una mejor visualización en el **Turbo Debugger**.

**1. Sumar y restar direcciones en BX**

Cuando se suma un valor a **BX**, cambia la dirección de memoria que está apuntando.

Ejemplo:

ADD BX, 2 ; BX = BX + 2

MOV AX, [BX] ; Se mueve el contenido de la nueva posición de memoria a AX

* Si **BX = 1**, después de la suma **BX = 3**.
* Si en memoria tenemos:

| **Dirección** | **Valor (Hex)** |
| --- | --- |
| 0001 | FE |
| 0002 | FD |
| 0003 | FC |

* AX tendrá **FC** (junto con el siguiente byte si es un word).

Si en lugar de sumar, restamos:

SUB BX, 2 ; BX = BX - 2

MOV AX, [BX] ; Se accede a la dirección anterior

* Si **BX estaba en 3**, después de restar 2, ahora **BX = 1**.
* Se mueve el contenido de la posición **1** y **2** a **AX**, lo que nos da **FD FE** en formato **Little Endian**.

**2. Limpieza de registros antes de mover datos**

Cuando se mueve un valor en ensamblador, el contenido previo de un registro sigue ahí. Para evitar confusiones en el **Turbo Debugger**, se recomienda inicializar el registro antes de cargar nuevos valores.

Para limpiar **AX**, se usa **XOR**:

XOR AX, AX ; AX = 0

Después de esta instrucción, cualquier valor nuevo en **AX** se mostrará correctamente, sin restos de datos previos.

**3. Multiplicación y división en ensamblador**

En ensamblador, la multiplicación y división también pueden afectar registros.

Ejemplo de multiplicación:

MOV BX, 3

MOV AX, 2

MUL BX ; AX = AX \* BX (3 \* 2 = 6)

* **BX = 3**, **AX = 2**
* Después de la multiplicación: **AX = 6**

Si se quiere dividir:

MOV AX, 6

MOV BX, 2

DIV BX ; AX = AX / BX (6 / 2 = 3)

* **AX se divide por BX** y el resultado queda en **AX**.

**Multiplicación y Acceso a Memoria en Ensamblador**

Aquí se está explicando cómo la multiplicación afecta el desplazamiento en memoria y cómo se deben contar las posiciones correctamente al acceder a datos almacenados.

**1. Multiplicación y uso del resultado como desplazamiento**

Cuando se multiplica un valor en ensamblador, el resultado se puede usar como un índice o desplazamiento en memoria.

Ejemplo en código ensamblador:

MOV BX, 3

MOV AX, 2

MUL BX ; AX = AX \* BX (3 \* 2 = 6)

* **BX = 3**, **AX = 2**
* Después de la multiplicación, **AX = 6**

Ahora, este resultado **(6)** se usa como desplazamiento en memoria:

MOV AX, [BX+6]

Esto significa que se está accediendo a la posición de memoria **BX + 6**.

**2. Diferencia entre empezar en 0 y contar elementos**

Cuando se cuentan posiciones en memoria, es importante recordar que se empieza desde la posición inicial, pero la cantidad de elementos se cuenta desde 1.

Ejemplo de posiciones en memoria:

| **Posición** | **Valor (Hex)** |
| --- | --- |
| 0000 | FF |
| 0001 | FE |
| 0002 | FD |
| 0003 | FC |
| 0004 | 00 |
| 0005 | 07 |
| 0006 | FF |

Si se multiplica **2 × 3 = 6**, entonces **BX + 6** apunta a la posición **0006**.

Si se está extrayendo un **word (2 bytes)** con:

MOV AX, [BX+6]

Se obtendrán los valores en **0006 y 0007**.

Si en la memoria está almacenado:

| **Dirección** | **Valor (Hex)** |
| --- | --- |
| 0006 | FF |
| 0007 | FD |

Entonces, debido a **Little Endian**, **AX** tendrá **FDFF**.

**3. Confirmación de la operación en Turbo Debugger**

Para visualizar bien los valores en el **Turbo Debugger**, es importante seguir estos pasos:

1. **Inicializar el registro** para evitar residuos de datos previos:

XOR AX, AX

1. **Realizar la multiplicación** y almacenar el resultado:

MOV BX, 3

MOV AX, 2

MUL BX ; AX = 6

1. **Acceder a la posición correcta en memoria** basada en el resultado:

MOV AX, [BX+6]

1. **Interpretar el valor leído en formato Little Endian**.

**Conclusión**

* **La multiplicación puede usarse como desplazamiento para acceder a memoria.**
* **Es importante contar correctamente las posiciones al moverse en la memoria.**
* **Little Endian siempre organiza los bytes en orden inverso.**
* **Turbo Debugger mantiene valores previos a menos que el registro se inicialice con XOR AX, AX.**

**División en Ensamblador y Primer Quiz**

En ensamblador, la **división** es más compleja que la multiplicación porque se deben considerar los registros involucrados y cómo se almacena el cociente y el residuo.

**1. División en Ensamblador**

Para dividir dos números de **16 bits**, se usa la instrucción **DIV**:

MOV AX, valor ; Cargamos el dividendo en AX

MOV BX, divisor ; Cargamos el divisor en BX

DIV BX ; AX / BX

* **AX** almacena el dividendo (el número que se divide).
* **BX** almacena el divisor.
* **El cociente se guarda en AX y el residuo en DX.**

Ejemplo:  
Si queremos dividir **1000 / 3**:

MOV AX, 1000

MOV BX, 3

DIV BX

* **AX contendrá el cociente** (333 en decimal).
* **DX contendrá el residuo** (1 en decimal).

**2. Consideraciones Importantes en la División**

* **El dividendo de 16 bits debe estar en AX.**
* **Si el dividendo es de 32 bits**, se debe usar **DX:AX** (donde DX almacena la parte alta del número).
* **La división es más lenta que la multiplicación en ensamblador.**

**3. Descripción del Quiz**

* **Quiz #1:** Se puede trabajar en parejas, pero cada estudiante debe entregarlo individualmente.
* **Quiz #2:** Requiere hacer una división en ensamblador.
* **El objetivo:** Dividir dos números de **16 bits**.
* **Nota importante:** Los números pueden estar alambrados, lo que significa que estarán definidos en el código fuente.
* **Tiempo:** Hasta las **8:20** (aproximadamente 30 minutos).

**Ejemplo de Código para el Quiz**

Si el objetivo del quiz es dividir dos números de 16 bits, aquí tienes un posible código en ensamblador:

section .data

mensaje db "Resultado: ", 0

section .bss

resultado resw 1

section .text

global \_start

\_start:

MOV AX, 500 ; Dividendo (16 bits)

MOV BX, 25 ; Divisor (16 bits)

DIV BX ; AX = AX / BX, DX = residuo

; Guardamos el resultado

MOV [resultado], AX

; Termina el programa

MOV EAX, 1 ; syscall: exit

XOR EBX, EBX

INT 0x80

Este programa divide **500 / 25** y almacena el resultado en memoria.

**Conclusión**

* **La división en ensamblador requiere AX y BX.**
* **El cociente va en AX y el residuo en DX.**
* **Para dividir números de 16 bits, solo se usa AX y BX.**
* **Si se usa un número de 32 bits, se debe trabajar con DX:AX.**
* **El quiz se enfoca en dividir dos números de 16 bits correctamente.**

**División de Números de 16 Bits en Ensamblador**

Este ejercicio consiste en realizar una **división de dos números de 16 bits** utilizando ensamblador, asegurándonos de manejar correctamente los registros y la memoria.

**1. Rango de la División**

* **Máximo:** 65,535÷65,535=165,535 65,535 = 165,535÷65,535=1
* **Mínimo:** 65,535÷1=65,53565,535 1 = 65,53565,535÷1=65,535
* **Cualquier valor entre estos dos es válido.**

**2. ¿Qué significa que los números estén "alambrados"?**

Cuando un número está **alambrado**, significa que está definido directamente en el código fuente y no se ingresa en tiempo de ejecución.

Ejemplo de variables en ensamblador:

dividendo DW 65535 ; Número a dividir (DWORD - 2 WORDs)

divisor DW 2 ; Divisor (DWORD - 2 WORDs)

Estos valores no cambian en tiempo de ejecución, sino que están predefinidos en la memoria del programa.

**3. Implementación en Ensamblador**

realizar la **división de dos números de 16 bits**, almacenando el resultado correctamente:

section .data

dividendo DW 65535 ; Número a dividir (DWORD - 2 WORDs)

divisor DW 2 ; Divisor (DWORD - 2 WORDs)

mensaje db "Resultado: ", 0

section .bss

resultado resw 1 ; Espacio para almacenar el resultado

section .text

global \_start

\_start:

; Cargar el dividendo en AX

MOV AX, [dividendo]

; Cargar el divisor en BX

MOV BX, [divisor]

; Realizar la división

DIV BX ; AX = AX / BX, DX = residuo

; Guardar el resultado

MOV [resultado], AX

; Termina el programa

MOV EAX, 1 ; syscall: exit

XOR EBX, EBX

INT 0x80

**4. Explicación del Código**

1. **Se definen los valores alambrados** en la sección de datos (.data).
2. **Se carga el dividendo en AX** y el divisor en BX.
3. **Se ejecuta la instrucción DIV BX**, que divide AX entre BX.
   * **El cociente se almacena en AX.**
   * **El residuo se almacena en DX (si fuera necesario usarlo).**
4. **Se almacena el resultado en memoria** (resultado).
5. **Se finaliza el programa.**

**5. Consideraciones Importantes**

* **La división en ensamblador siempre usa AX para el dividendo y BX para el divisor.**
* **Si el dividendo es mayor a 16 bits (DWORD), se debe usar DX:AX como dividendo.**
* **El divisor no puede ser 0, porque generaría una excepción.**
* **DX almacena el residuo, útil si se necesita calcular un módulo.**

**6. Conclusión**

* **Este código permite dividir dos números alambrados en memoria.**
* **Funciona con cualquier valor entre 1 y 65,535.**
* **Los resultados se almacenan en AX y pueden guardarse en memoria.**
* **Es clave entender cómo Ensamblador maneja las operaciones aritméticas en registros.**

**Diferencia entre Macros y Procedimientos en Ensamblador**

En ensamblador, las **macros y los procedimientos** son formas de estructurar el código, pero funcionan de manera diferente.

**1. Las Macros No Tienen Estructura Fija**

* Como mencionaste, las **macros** no tienen una estructura definida dentro de un segmento específico.
* Cuando defines una macro, simplemente **existen en el código** pero **no ocupan una posición fija en memoria** hasta que se expanden en tiempo de ensamblaje.
* Ejemplo de una macro:

macro1 Macro

mov ax, 0

EndM

* + No está dentro de ningún segmento, simplemente se define y luego se usa donde se necesite.

**2. Los Procedimientos Necesitan un Segmento**

* **Los procedimientos sí deben estar en un segmento de código**, porque ocupan una posición específica en memoria.
* Se definen dentro del segmento de código (Codigo Segment), lo que permite que el ensamblador sepa exactamente dónde se encuentran.
* Ejemplo de un procedimiento:

Codigo Segment

Assume CS:Codigo, DS:Datos

; Definición del procedimiento

mi\_procedimiento PROC

mov ax, bx

ret

mi\_procedimiento ENDP

inicio:

call mi\_procedimiento ; Llamar al procedimiento

FinProgra

Codigo Ends

End inicio

* + **Está dentro del segmento de código (Codigo Segment).**
  + Se llama con CALL y se sale con RET, asegurando que el programa pueda volver a la ejecución normal.

**3. ¿Por qué los Procedimientos Necesitan un Segmento?**

1. Porque ocupan memoria en el código ejecutable.
2. Porque el ensamblador necesita saber su dirección exacta.
3. Porque su ejecución sigue el flujo normal del programa.
4. Porque necesitan una estructura clara dentro del código.

En cambio, **las macros son solo expansiones de código y no requieren una estructura en memoria**.

**Conclusión**

* **Las macros se pueden definir en cualquier parte del código y simplemente expanden instrucciones en el punto donde se llaman.**
* **Los procedimientos deben estar dentro de un segmento (normalmente en Codigo Segment) porque ocupan un espacio específico en memoria y deben ser llamados con CALL y terminar con RET.**
* **El ensamblador necesita saber dónde comienza y termina cada procedimiento, por eso deben estar bien estructurados dentro del segmento de código.**

**Diferencia entre llamadas de procedimientos NEAR y FAR en ensamblador**

En ensamblador **x86**, hay dos tipos principales de llamadas a procedimientos:

1. **NEAR (cercano)**
2. **FAR (lejano)**

**1. ¿Qué es un procedimiento NEAR?**

Un **procedimiento NEAR** (cercano) es aquel que **se encuentra en el mismo segmento de código** donde se hace la llamada.

* Se usa la instrucción CALL sin necesidad de especificar el segmento.
* La dirección de retorno se almacena en la pila con **solo la dirección de offset**.
* RET devuelve la ejecución al punto de la llamada sin modificar el segmento de código.

**Ejemplo de un procedimiento NEAR:**

Codigo Segment

Assume CS:Codigo, DS:Datos

mi\_procedimiento\_near PROC NEAR

mov ax, bx

ret ; Retorna sin modificar CS

mi\_procedimiento\_near ENDP

inicio:

call mi\_procedimiento\_near ; Llamada a procedimiento NEAR

FinProgra

Codigo Ends

End inicio

**Características del procedimiento NEAR:**  
✔ Se usa cuando el código se ejecuta dentro del **mismo segmento**.  
✔ CALL solo almacena en la pila la dirección de **offset**.  
✔ RET usa ese offset para volver al punto de la llamada.

**2. ¿Qué es un procedimiento FAR?**

Un **procedimiento FAR** (lejano) es aquel que **está en un segmento diferente** al código que lo llama.

* Se usa CALL con un segmento y un offset.
* La dirección de retorno se guarda en la pila con **el segmento y el offset**.
* RET debe limpiar ambos valores al regresar.

**Ejemplo de un procedimiento FAR:**

OtroCodigo Segment

Assume CS:OtroCodigo, DS:Datos

mi\_procedimiento\_far PROC FAR

mov ax, bx

retf ; Retorna restaurando CS y la dirección de ejecución

mi\_procedimiento\_far ENDP

OtroCodigo Ends

Luego, en el código principal:

Codigo Segment

Assume CS:Codigo, DS:Datos

inicio:

call mi\_procedimiento\_far ; Llamada a procedimiento FAR

FinProgra

Codigo Ends

End inicio

**Características del procedimiento FAR:**  
✔ Se usa cuando el código **llama a otro segmento**.  
✔ CALL guarda en la pila **el segmento y el offset**.  
✔ RETF limpia ambos valores para restaurar la ejecución.

**3. ¿Cómo afecta esto a las variables y registros?**

El profesor mencionó **carácter leído** en relación con los procedimientos.

**Caso sin procedimientos**

Si una variable se usa directamente en un macro o un código global, se puede declarar así:

caracter\_leido db ? ; Variable en memoria para almacenar el carácter leído

Pero si estamos dentro de un procedimiento, es común usar **BP** para acceder a parámetros o valores temporales en la pila.

**Caso con procedimientos**

Si un procedimiento debe modificar **carácter leído**, lo puede hacer así:

mi\_procedimiento\_near PROC NEAR

mov al, [caracter\_leido] ; Leer el carácter

mov bp, al ; Guardarlo en BP (por ejemplo)

ret

mi\_procedimiento\_near ENDP

Si el procedimiento es FAR, hay que asegurarse de que DS apunte correctamente al segmento de datos antes de acceder a las variables.

**4. ¿Cómo afecta esto a la estructura del código?**

El profesor enfatizó que **los procedimientos necesitan estar en un segmento**, a diferencia de las macros, que están "en el limbo".  
Por eso, los procedimientos **deben estar organizados dentro del segmento de código** para que el ensamblador sepa dónde empiezan y terminan.

**Conclusión**

* **NEAR** se usa dentro del mismo segmento.
* **FAR** se usa cuando el código salta a otro segmento.
* **Las variables globales pueden declararse en la sección de datos** o manejarse con registros como **BP** dentro de un procedimiento.
* **Los procedimientos deben definirse dentro de un segmento de código**, a diferencia de las macros, que pueden estar en cualquier parte del código fuente.

**Definir y llamar procedimientos en ensamblador (NEAR y FAR)**

En ensamblador, los **procedimientos pueden definirse en cualquier parte del código**, pero lo importante es **cómo se llaman y cómo afecta al flujo del programa**.

**1. ¿Qué pasa si defines un procedimiento en medio del código?**

Si defines un procedimiento **dentro del flujo del programa principal y no lo llamas con CALL**, el ensamblador simplemente lo **ejecutará como si fuera parte normal del código**.

**Ejemplo 1: Procedimiento en medio del código, sin CALL**

Codigo Segment

Assume CS:Codigo, DS:Datos

inicio:

IniciaDatos datos ; Inicializa DS con el segmento de datos

mi\_procedimiento PROC

mov al, 'L' ; Guarda 'L' en AL

mov bl, 'B' ; Guarda 'B' en BL

ret

mi\_procedimiento ENDP

; Código normal

mov ah, 01h

int 21h

FinProgra

Codigo Ends

End inicio

🔹 **¿Qué pasa aquí?**

* Como mi\_procedimiento está definido **sin una llamada CALL**, el flujo del programa lo ejecutará como si fuera código normal y luego seguirá con las instrucciones que están después.
* **No hay separación real** entre el código principal y el procedimiento, lo que puede causar errores si no está planeado.

**2. ¿Qué pasa si defines un procedimiento pero no lo llamas?**

Si defines un procedimiento pero nunca lo llamas con CALL, entonces **nunca se ejecutará**.

**Ejemplo 2: Procedimiento definido pero sin llamar**

Codigo Segment

Assume CS:Codigo, DS:Datos

inicio:

IniciaDatos datos

; Código normal aquí

FinProgra ; El programa termina sin llamar al procedimiento

mi\_procedimiento PROC

mov al, 'X'

mov bl, 'Y'

ret

mi\_procedimiento ENDP

Codigo Ends

End inicio

🔹 **¿Qué pasa aquí?**

* **mi\_procedimiento nunca se ejecuta** porque no hay un CALL que lo invoque.
* El ensamblador lo incluye en el código, pero **queda inactivo** a menos que se llame explícitamente.

**3. Llamando un procedimiento NEAR (CALL y RET)**

Los procedimientos NEAR (cercanos) son aquellos que están en **el mismo segmento de código**.

**Ejemplo 3: Llamando un procedimiento NEAR**

Codigo Segment

Assume CS:Codigo, DS:Datos

inicio:

IniciaDatos datos

call mi\_procedimiento ; Llamar al procedimiento NEAR

FinProgra ; Luego, termina el programa

mi\_procedimiento PROC NEAR

mov al, 'L' ; Guarda 'L' en AL

mov bl, 'B' ; Guarda 'B' en BL

ret ; Retorna a la instrucción después de CALL

mi\_procedimiento ENDP

Codigo Ends

End inicio

🔹 **¿Qué pasa aquí?**

* CALL mi\_procedimiento **guarda la dirección de retorno en la pila** y salta a la dirección del procedimiento.
* RET recupera la dirección guardada en la pila y **vuelve a ejecutar la siguiente instrucción después de CALL**.

**4. Viendo el IP (Instruction Pointer) al llamar un procedimiento**

Cuando se ejecuta un procedimiento, el registro **IP (Instruction Pointer)** cambia, ya que apunta a la nueva posición en memoria donde está el procedimiento.

🔹 **Ejemplo del IP en un procedimiento NEAR:**  
Si CALL se ejecuta cuando el IP está en 0017h, el programa saltará a la dirección del procedimiento.  
Después de RET, el **IP vuelve a la siguiente instrucción después de CALL**.

Ejemplo visual del cambio en el **IP**:

python

Antes del CALL: IP = 0017h

Después del CALL: IP = Dirección del procedimiento

Después del RET: IP = 001A (siguiente instrucción)

📌 **Este comportamiento es fundamental para entender cómo los procedimientos afectan el flujo del programa.**

**Conclusión**

* **Si defines un procedimiento dentro del código sin CALL, se ejecutará como código normal.**
* **Si defines un procedimiento pero no lo llamas, nunca se ejecutará.**
* **Para llamar correctamente un procedimiento, se usa CALL y se retorna con RET.**
* **El IP cambia cuando se ejecuta un CALL, y RET lo devuelve a la dirección original.**
* **Los procedimientos deben definirse en un segmento y llamarse correctamente para evitar errores.**

**Por qué el programa se queda en un ciclo infinito y cómo solucionarlo**

El problema ocurre porque el programa **define procedimientos dentro del segmento de datos**, lo que hace que el flujo de ejecución **se altere de manera inesperada**. Como resultado, el código entra en un **bucle infinito** sin posibilidad de salir.

**1. ¿Por qué el programa salta a la posición 0000?**

Cuando defines un procedimiento dentro del **segmento de datos** y luego lo llamas sin CALL, la ejecución **interpreta la dirección 0000** como el inicio del procedimiento.

📌 **Ejemplo del error:**

Datos Segment

procedimiento1 db ? ; ¡Esto no debería estar en la sección de datos!

Datos Ends

* Como el procedimiento se almacena en el segmento de datos, el código **salta a la dirección 0000** en lugar de continuar correctamente en el segmento de código.

🔹 **¿Qué pasa luego?**

1. Se ejecuta el **procedimiento número 1** porque el flujo de ejecución lo alcanza naturalmente.
2. Luego, sin un RET, el programa sigue con **procedimiento 2**.
3. Finalmente, vuelve al segmento de código, pero reinicia el flujo de ejecución desde el inicio.
4. **Esto causa un ciclo infinito**, donde el procedimiento se sigue ejecutando sin control.

**2. ¿Por qué el programa sobrescribe el valor de "carácter leído"?**

Si la variable **carácter leído** se almacena en el segmento de datos y es modificada en cada iteración del ciclo infinito, el valor se sobreescribe sin control.

Ejemplo de problema:

caracter\_leido db 'J' ; Inicialmente tiene 'J'

* Luego, en el código:

mov caracter\_leido, 'P' ; Se sobrescribe con 'P'

📌 **El resultado es que carácter leído pasa de 'J' a 'P' y sigue cambiando en cada ciclo.**

**3. ¿Cómo evitar el ciclo infinito?**

Para corregir este problema, debes:

1. **Definir los procedimientos dentro del segmento de código.**
   * No los pongas en el segmento de datos.
2. **Asegurar que cada procedimiento termine con RET.**
   * Esto evita que el programa continúe ejecutando código innecesario.
3. **Usar CALL para llamar a los procedimientos.**
   * CALL almacena la dirección de retorno en la pila, permitiendo volver correctamente después de ejecutar el procedimiento.

📌 **Corrección del código:**

Codigo Segment

Assume CS:Codigo, DS:Datos

inicio:

IniciaDatos datos

call procedimiento1 ; Llamar a procedimiento 1

call procedimiento2 ; Llamar a procedimiento 2

FinProgra

procedimiento1 PROC NEAR

mov al, 'P' ; Modifica carácter leído

mov [caracter\_leido], al

ret ; Regresa al punto de la llamada

procedimiento1 ENDP

procedimiento2 PROC NEAR

; Otra operación...

ret

procedimiento2 ENDP

Codigo Ends

End inicio

🔹 **¿Qué cambia aquí?**  
✔ CALL llama a los procedimientos correctamente.  
✔ RET devuelve la ejecución al código principal, evitando ciclos infinitos.  
✔ Los procedimientos están dentro del **segmento de código**, evitando saltos erróneos a 0000.

**4. Conclusión**

📌 **Para evitar que el programa se quede en un ciclo infinito:**  
✅ **No definas procedimientos en el segmento de datos.**  
✅ **Usa CALL para invocar los procedimientos.**  
✅ **Asegura que cada procedimiento termine con RET.**  
✅ **Verifica que el flujo del programa regrese correctamente después de la ejecución del procedimiento.**

**Uso correcto de RET en procedimientos NEAR y FAR**

Cuando defines un procedimiento en ensamblador, es fundamental **indicar cómo debe retornar la ejecución**. Esto se hace con la instrucción RET, pero hay una diferencia entre **procedimientos NEAR y FAR**.

**1. RET en procedimientos NEAR**

Un **procedimiento NEAR** es aquel que se ejecuta **dentro del mismo segmento de código** que lo llama.

🔹 **Ejemplo de procedimiento NEAR:**

mi\_procedimiento\_near PROC NEAR

mov ax, bx

ret ; Regresa a la instrucción después de CALL en el mismo segmento

mi\_procedimiento\_near ENDP

📌 **Cómo funciona:**

* CALL **solo almacena la dirección de retorno (offset) en la pila**.
* RET recupera el offset desde la pila y regresa a la instrucción siguiente a CALL.
* **Como el segmento no cambia, no es necesario restaurar CS (Code Segment).**

🔹 **Ejemplo de llamada:**

call mi\_procedimiento\_near ; Salta a la dirección del procedimiento

🔹 **Cuando RET se ejecuta:**

* Recupera la dirección de retorno desde la pila.
* La ejecución continúa en la siguiente instrucción después de CALL.

**2. RETF en procedimientos FAR**

Un **procedimiento FAR** es aquel que está en **un segmento de código diferente** al código que lo llama.

🔹 **Ejemplo de procedimiento FAR:**

mi\_procedimiento\_far PROC FAR

mov ax, bx

retf ; Retorna restaurando CS y la dirección de ejecución

mi\_procedimiento\_far ENDP

📌 **Cómo funciona:**

* CALL FAR **almacena en la pila tanto el offset como el segmento del procedimiento**.
* RETF limpia ambos valores de la pila y restaura **CS:IP** (Code Segment e Instruction Pointer).
* Es útil cuando un procedimiento está en otro segmento de código.

🔹 **Ejemplo de llamada:**

call mi\_procedimiento\_far ; Salta a un segmento diferente

🔹 **Cuando RETF se ejecuta:**

* Recupera la dirección de retorno **y el segmento** desde la pila.
* La ejecución regresa correctamente, incluso si estaba en otro segmento.

**3. Diferencia entre RET y RETF en la práctica**

| **Tipo de Procedimiento** | **Dirección de Retorno en la Pila** | **Retorno Correcto** |
| --- | --- | --- |
| **NEAR** | Solo guarda el **offset** de la dirección de retorno. | RET |
| **FAR** | Guarda el **offset** + **segmento** de la dirección de retorno. | RETF |

**4. Cómo diferenciar los procedimientos NEAR y FAR en el código**

El ensamblador **asume por defecto** que un procedimiento **sin etiqueta es NEAR**, a menos que se especifique FAR.

🔹 **Ejemplo con etiquetas explícitas:**

procedimiento\_near PROC NEAR

ret

procedimiento\_near ENDP

procedimiento\_far PROC FAR

retf

procedimiento\_far ENDP

🔹 **Ejemplo sin etiquetas (NEAR por defecto):**

procedimiento\_sin\_etiqueta PROC

ret ; Se considera NEAR automáticamente

procedimiento\_sin\_etiqueta ENDP

📌 **Conclusión:**

* Si **no especificas NEAR o FAR**, el ensamblador **asume que es NEAR**.
* Si el procedimiento es **FAR**, es recomendable usar RETF para claridad.
* Aunque RET funciona en procedimientos FAR, es una **buena práctica usar RETF** para diferenciarlos.

**5. Corrección del código para asegurar un retorno correcto**

Si el profesor enfatiza el uso correcto de RET y RETF, puedes asegurarte de que todos los procedimientos tengan la instrucción correcta:

Codigo Segment

Assume CS:Codigo, DS:Datos

inicio:

call procedimiento\_near

call procedimiento\_far

call procedimiento\_sin\_etiqueta

FinProgra

procedimiento\_near PROC NEAR

mov al, 'X'

ret ; Retorno NEAR

procedimiento\_near ENDP

procedimiento\_far PROC FAR

mov al, 'Y'

retf ; Retorno FAR

procedimiento\_far ENDP

procedimiento\_sin\_etiqueta PROC

mov al, 'Z'

ret ; Se considera NEAR automáticamente

procedimiento\_sin\_etiqueta ENDP

Codigo Ends

End inicio

**6. Conclusión General**

✅ **Usa RET para procedimientos NEAR.**  
✅ **Usa RETF para procedimientos FAR (aunque RET funcione, RETF es más claro).**  
✅ **Si un procedimiento no tiene NEAR o FAR, el ensamblador lo toma como NEAR.**  
✅ **Si no usas RET o RETF, el código continuará ejecutándose y puede causar errores.**

**Explicación: Cómo funciona el retorno de un procedimiento en ensamblador**

En esta explicación, vamos a analizar **cómo el programa ejecuta un procedimiento, almacena la dirección de retorno, y cómo el Instruction Pointer (IP) avanza correctamente después de CALL y RET**.

**1. Verificando si el procedimiento retorna correctamente**

Para asegurarse de que un procedimiento **no cause un bucle infinito**, es necesario **ver qué dirección de memoria se ejecuta después de la llamada**.

**Ejemplo de flujo de ejecución con CALL y RET**

inicio:

call procedimiento1 ; Llama al procedimiento

xor bx, bx ; Siguiente instrucción después del `CALL`

FinProgra

procedimiento1 PROC NEAR

mov al, 'F' ; Modifica una variable o registro

mov [caracter\_leido], al

ret ; Retorna a la instrucción siguiente a `CALL`

procedimiento1 ENDP

📌 **¿Qué sucede aquí?**

1. CALL procedimiento1 → **Guarda en la pila** la dirección de la siguiente instrucción.
2. **Ejecuta el procedimiento** (mov al, 'F', etc.).
3. RET → **Extrae de la pila la dirección de retorno y la carga en IP**.
4. La **ejecución continúa en la siguiente instrucción después del CALL**, que es xor bx, bx.

**2. ¿Cómo verificar dónde retorna el procedimiento?**

El profesor menciona que hay que fijarse en el **IP (Instruction Pointer)**.

**📌 Ejemplo de cómo cambia el IP**

Supongamos que **antes del CALL**, el IP está en **0019h**.

* **Después de ejecutar CALL procedimiento1**, IP cambia a la dirección del procedimiento.
* Una vez que RET se ejecuta, **el IP debe volver a la siguiente instrucción después del CALL**.

Si el procedimiento ocupa **3 bytes de código**, la siguiente instrucción estará en la posición **001C**.

**3. Comprobando el flujo de ejecución en la memoria**

Si se observa en un debugger o en un análisis del código, se verá algo como esto:

vbnet

CALL procedimiento1 → IP = 0019h

Ejecuta procedimiento1

RET → IP vuelve a 001C (siguiente instrucción)

XOR BX, BX → IP sigue en 001C

🔹 **Aquí es donde se confirma que el procedimiento retorna correctamente**.  
🔹 **Si el IP no vuelve al punto correcto, significa que RET no se ejecutó o que hay un problema en la pila.**

**4. ¿Qué significa que siga en la posición 001C?**

El profesor menciona que después de CALL, el programa **no debe quedarse en el procedimiento**.

**Ejemplo de código con error:**

procedimiento1 PROC

mov al, 'F'

mov [caracter\_leido], al

; FALTA `RET` AQUÍ (el programa seguirá ejecutando instrucciones incorrectas)

procedimiento1 ENDP

Si RET falta, la ejecución **seguirá con cualquier código después del procedimiento**, causando **comportamiento inesperado** o incluso **un bucle infinito**.

🔹 **El código correcto siempre debe incluir RET en los procedimientos.**

**5. Verificando el avance del IP después de CALL**

El profesor menciona que después del CALL, el IP avanza en este orden:

1. **0019 → CALL procedimiento1**
2. **0021 → Siguiente instrucción (XOR BX, BX)**

Es decir:

* CALL → Salta al procedimiento y guarda la dirección de retorno.
* RET → Recupera la dirección guardada y la coloca en IP.
* La siguiente instrucción se ejecuta correctamente en la posición esperada.

**Conclusión: ¿Qué hay que hacer para que funcione bien?**

📌 **Para evitar ciclos infinitos y garantizar el retorno correcto:**  
✅ **Siempre agregar RET en procedimientos NEAR**.  
✅ **Usar RETF en procedimientos FAR si es necesario**.  
✅ **Verificar en un debugger que IP vuelve a la dirección correcta después del CALL**.  
✅ **Evitar definir procedimientos dentro de segmentos incorrectos (ej. segmento de datos)**.

**¿Cómo funciona la pila en ensamblador al llamar un procedimiento NEAR?**

En ensamblador, **la pila (stack)** es una **zona de memoria** donde se almacenan temporalmente valores importantes, como **direcciones de retorno, registros y variables locales**.

Cuando se ejecuta un **procedimiento NEAR (CALL y RET)**, la pila juega un papel crucial en el control del flujo del programa.

**1. ¿Qué sucede en la pila cuando se ejecuta CALL en un procedimiento NEAR?**

📌 **Ejemplo de código:**

inicio:

call procedimiento\_near ; Llama al procedimiento

xor bx, bx ; Siguiente instrucción después del CALL

FinProgra

procedimiento\_near PROC NEAR

mov al, 'F'

ret ; Retorna al punto donde se llamó

procedimiento\_near ENDP

🔹 **Flujo de ejecución:**

1. CALL procedimiento\_near
   * **Guarda en la pila la dirección de retorno (IP = 001C)**.
   * **El SP (Stack Pointer) se decrementa en 2**, porque en modo real x86, cada dirección de retorno ocupa **2 bytes**.
   * La ejecución salta a procedimiento\_near.
2. **Ejecución de procedimiento\_near**
   * Ejecuta mov al, 'F' y cualquier otra instrucción.
   * Llega a RET, que **saca de la pila la dirección de retorno** y la coloca en IP.
3. **IP se actualiza y el programa continúa**
   * **IP vuelve a 001C**, que es la instrucción después del CALL (xor bx, bx).
   * La ejecución sigue normalmente.

**2. ¿Cómo se almacena la dirección de retorno en la pila?**

Cuando se ejecuta CALL, la dirección de retorno se **almacena en la pila** en dos pasos:

* **Ejemplo visual de cómo cambia la pila:**

| **SP antes de CALL** | **Contenido guardado** | **SP después de CALL** |
| --- | --- | --- |
| SP = 0000h | 001C (dirección de retorno) | SP = FFFEh |

🔹 **¿Por qué SP pasa de 0000h a FFFEh?**

* En **modo real de x86**, la pila **crece hacia abajo** en memoria.
* Restar 2 a SP es equivalente a darle la vuelta al segmento, pasando de 0000h a FFFEh.
* Por eso, el valor 001C (dirección de retorno) se guarda en FFFEh.

**3. Ejemplo visual de cómo cambia la pila**

Si antes del CALL, SP = 0000h, después del CALL, SP será FFFEh y la pila contendrá:

ini

SP = 0000h → CALL procedimiento\_near

SP = FFFEh → Se guarda `001C` en la pila

Si después del RET, el SP vuelve a 0000h, la ejecución continuará después del CALL.

**4. Resumen del funcionamiento de la pila en CALL y RET**

📌 **Al ejecutar CALL procedimiento\_near:**  
✅ La **dirección de retorno (IP = 001C) se guarda en la pila**.  
✅ **SP se decrementa en 2** para almacenar la dirección.  
✅ La ejecución salta al procedimiento.

📌 **Al ejecutar RET:**  
✅ **Saca la dirección de la pila y la carga en IP**.  
✅ **SP se incrementa en 2** para restaurar el estado anterior.  
✅ **La ejecución continúa en la dirección almacenada (001C).**

**Cómo manejar procedimientos declarados antes del inicio en ensamblador**

En ensamblador, es posible definir procedimientos **antes de la etiqueta inicio**, pero para que el código funcione correctamente, es necesario asegurarse de que el flujo de ejecución **no entre directamente a esos procedimientos** al inicio del programa.

Una **forma correcta de hacerlo** es usando un **salto (JMP Short)** al inicio del programa para saltar los procedimientos y comenzar la ejecución en la parte principal del código.

**1. ¿Qué problema ocurre si los procedimientos están antes de inicio?**

Si un procedimiento está antes de inicio, el ensamblador lo ejecutará directamente **sin necesidad de CALL**, lo que puede causar errores en el flujo del programa.

📌 **Ejemplo de código problemático:**

Codigo Segment

Assume CS:Codigo, DS:Datos

procedimiento1 PROC NEAR

mov al, 'X'

ret

procedimiento1 ENDP

procedimiento2 PROC NEAR

mov al, 'Y'

ret

procedimiento2 ENDP

inicio:

IniciaDatos datos

call procedimiento1

call procedimiento2

FinProgra

Codigo Ends

End inicio

🔹 **¿Qué pasa aquí?**

* El ensamblador **ejecutará directamente procedimiento1 y procedimiento2** antes de llegar a inicio, lo que puede causar un comportamiento inesperado.
* **Los procedimientos deben ejecutarse solo cuando son llamados con CALL, no al inicio del programa.**

**2. Solución: Usar JMP Short para saltar a inicio**

Para evitar que el ensamblador **ejecute directamente los procedimientos**, podemos agregar un **salto corto (JMP Short)** para que el flujo del programa **salte directamente a inicio**, dejando los procedimientos listos para ser llamados después.

📌 **Código corregido con JMP Short**

Codigo Segment

Assume CS:Codigo, DS:Datos

; Saltar los procedimientos y empezar en `inicio`

JMP Short inicio

procedimiento1 PROC NEAR

mov al, 'X'

ret

procedimiento1 ENDP

procedimiento2 PROC NEAR

mov al, 'Y'

ret

procedimiento2 ENDP

inicio:

IniciaDatos datos

call procedimiento1

call procedimiento2

FinProgra

Codigo Ends

End inicio

🔹 **¿Qué cambia aquí?**

* **JMP Short inicio** hace que el programa **salte directamente a inicio**, ignorando los procedimientos al comienzo.
* **Ahora, procedimiento1 y procedimiento2 solo se ejecutarán cuando se llame con CALL.**

**3. ¿Por qué JMP Short es importante?**

✔ Evita que el ensamblador **ejecute automáticamente los procedimientos** antes del inicio del programa.  
✔ Asegura que el código **siga el flujo correcto** sin entrar en los procedimientos hasta que sean llamados.  
✔ **Optimiza la estructura del código**, permitiendo que los procedimientos estén en la parte superior sin causar problemas.

**Cómo funciona el flujo de ejecución y el retorno en procedimientos NEAR**

El profesor está mostrando **cómo un procedimiento NEAR solo se ejecuta si se llama explícitamente con CALL**. Además, se está analizando **cómo el IP (Instruction Pointer) cambia antes y después de ejecutar un procedimiento**.

**1. Diferencia entre F8 y F7 en la depuración**

En un **depurador de ensamblador** (como Turbo Debugger o DOSBox Debug), los comandos **F8** y **F7** tienen comportamientos diferentes:

🔹 **F8 (Step Over / Paso por encima):**

* Ejecuta la instrucción actual, pero **si es CALL**, **no entra al procedimiento**, sino que **lo ejecuta completamente y pasa a la siguiente instrucción después del CALL**.

🔹 **F7 (Step Into / Paso dentro):**

* Ejecuta la instrucción actual y **si es CALL, entra en el procedimiento**, permitiendo ver cada instrucción dentro del procedimiento.

📌 **Ejemplo en el depurador:**

call procedimiento\_near

xor bx, bx

Si usas **F8**, el depurador **ejecuta CALL procedimiento\_near pero no muestra su ejecución interna**.  
Si usas **F7**, el depurador **entra a procedimiento\_near y permite ver cada línea dentro del procedimiento**.

**2. Verificando el IP antes y después del CALL**

El **IP (Instruction Pointer)** cambia antes y después de CALL y RET.

📌 **Ejemplo de flujo de ejecución con IP:**

objectivec

CALL procedimiento\_near → IP = 0019h (antes del salto)

Ejecuta procedimiento

RET → IP vuelve a 001C (dirección después de CALL)

🔹 **Antes de CALL**:

* IP apunta a la dirección de CALL, por ejemplo, **0019h**.

🔹 **Después de CALL**:

* IP apunta al inicio del procedimiento.

🔹 **Después de RET**:

* IP **recupera la dirección de retorno desde la pila** y vuelve a **001C**, que es la instrucción siguiente a CALL.

**3. ¿Se puede modificar el valor de IP al hacer RET?**

La pregunta que hicieron en clase es:  
*"¿Se puede hacer que RET salte a una dirección diferente en la pila?"*

📌 **Respuesta: Sí y No.**

✅ **Sí:**  
Puedes **manipular la pila manualmente** para cambiar la dirección a la que retorna el procedimiento.

assembly

push 0030h ; Guarda una dirección falsa en la pila

ret ; RET tomará la dirección 0030h en vez de la original

⚠ **Pero esto puede romper el flujo del programa** si no se maneja correctamente.

❌ **No:**  
Si CALL guarda la dirección correcta de retorno y RET la usa normalmente, no puedes cambiar el valor de IP **sin manipular la pila manualmente**.

**4. ¿Cómo funciona la manipulación de la pila para modificar RET?**

El profesor mencionó un archivo llamado pila.asm. Seguramente es un código donde se manipula la pila para **modificar el retorno de un procedimiento**.

📌 **Ejemplo de manipulación de pila en un procedimiento:**

procedimiento\_modificado PROC NEAR

add sp, 2 ; Elimina la dirección de retorno de la pila

push 0030h ; Empuja una nueva dirección de retorno falsa

ret

procedimiento\_modificado ENDP

🔹 **¿Qué pasa aquí?**

* add sp, 2 **elimina la dirección de retorno original**.
* push 0030h **cambia la dirección de retorno**.
* RET **saltará a la dirección 0030h en vez de la original.**

⚠ **Esto puede ser peligroso y generar errores si se usa incorrectamente.**

El profesor : **los procedimientos pueden definirse en cualquier parte del código, pero es necesario manejar correctamente el flujo de ejecución**. También menciona **cómo los valores en la pila afectan el retorno (RET), el uso de CALL dentro de un procedimiento y los saltos (JMP)**.

**1. La importancia del IP (Instruction Pointer) y la "gravedad" del código**

El profesor menciona que el IP (Instruction Pointer) **sigue el código por "gravedad"**, lo que significa que, a menos que haya un **salto (JMP) o una llamada (CALL)**, la ejecución sigue secuencialmente.

📌 **Ejemplo de código sin saltos ni CALL:**

Codigo Segment

Assume CS:Codigo, DS:Datos

inicio:

mov al, 'A' ; Se ejecuta primero

mov al, 'B' ; Se ejecuta después

mov al, 'C' ; Se ejecuta después de 'B'

FinProgra ; Fin del programa

Codigo Ends

End inicio

🔹 **El código se ejecuta en orden sin interrupciones**.

Pero si **un procedimiento está antes del inicio**, es necesario **hacer un salto (JMP) para evitar que se ejecute automáticamente**.

📌 **Ejemplo con JMP para evitar ejecución inmediata:**

Codigo Segment

Assume CS:Codigo, DS:Datos

JMP Short inicio ; Salta el procedimiento para evitar que se ejecute inmediatamente

procedimiento1 PROC NEAR

mov al, 'X'

ret

procedimiento1 ENDP

inicio:

IniciaDatos datos

call procedimiento1

FinProgra

Codigo Ends

End inicio

✅ **Aquí el programa comienza en inicio y solo entra al procedimiento cuando CALL lo llama.**

**2. ¿Por qué CALL dentro de un procedimiento en lugar de RET?**

El profesor menciona que dentro de un procedimiento **se puede hacer otro CALL en lugar de RET**, lo que permite **llamar a otro procedimiento antes de retornar**.

📌 **Ejemplo de CALL dentro de un procedimiento:**

procedimiento1 PROC NEAR

mov al, 'A'

call procedimiento2 ; Llama a otro procedimiento antes de retornar

ret

procedimiento1 ENDP

procedimiento2 PROC NEAR

mov al, 'B'

ret

procedimiento2 ENDP

🔹 **¿Qué pasa aquí?**

1. CALL procedimiento1 se ejecuta.
2. Dentro de procedimiento1, se ejecuta CALL procedimiento2.
3. procedimiento2 ejecuta mov al, 'B' y luego **retorna a procedimiento1**.
4. procedimiento1 finalmente ejecuta RET y regresa al código principal.

✅ **Esto es útil cuando un procedimiento necesita ejecutar otro antes de salir.**

**3. ¿Qué significan los números en RET y JMP?**

El profesor menciona que **hay números que aparecen junto a RET y JMP**, y que se analizarán más adelante.

📌 **Ejemplo de RET 2:**

procedimiento1 PROC NEAR

mov al, 'X'

ret 2 ; Retorna y además ajusta la pila en 2 bytes

procedimiento1 ENDP

🔹 **¿Qué hace RET 2?**

* RET normalmente **saca de la pila la dirección de retorno**.
* RET 2 **saca la dirección de retorno y además ajusta la pila eliminando 2 bytes adicionales**.
* Se usa en procedimientos que toman parámetros desde la pila.

📌 **Ejemplo de JMP con números (JMP SHORT, JMP NEAR, JMP FAR)**

JMP SHORT etiqueta ; Salta a 'etiqueta' dentro del mismo segmento, a una distancia corta

JMP NEAR etiqueta ; Salta a 'etiqueta' dentro del mismo segmento, pero a mayor distancia

JMP FAR etiqueta ; Salta a 'etiqueta' en otro segmento

🔹 **Diferencias entre JMP SHORT, NEAR y FAR:**

| **Instrucción** | **Segmento** | **Rango de salto** |
| --- | --- | --- |
| JMP SHORT | Mismo segmento | Máximo 127 bytes adelante o -128 bytes atrás |
| JMP NEAR | Mismo segmento | Más de 127 bytes adelante |
| JMP FAR | Otro segmento | Salto a cualquier dirección en memoria |

✅ **El ensamblador decide automáticamente si JMP debe ser SHORT o NEAR si no se especifica.**

**1. Diferencia entre procedimientos NEAR y FAR**

Los procedimientos pueden ser de dos tipos:

✅ **NEAR (RET)**

* Se encuentran dentro del mismo **segmento de código**.
* CALL solo guarda la dirección **offset** en la pila.
* RET recupera la dirección de la pila y continúa la ejecución.

✅ **FAR (RETF)**

* Se encuentran en un **segmento diferente**.
* CALL guarda en la pila **segmento + offset**.
* RETF extrae ambos valores de la pila y salta de regreso.

📌 **Ejemplo de ambos procedimientos:**

procedimiento\_near PROC NEAR

mov al, 'X'

ret

procedimiento\_near ENDP

procedimiento\_far PROC FAR

mov al, 'Y'

retf

procedimiento\_far ENDP

**2. ¿Dónde se declaran los procedimientos?**

Los procedimientos **siempre deben declararse en un segmento**.  
📌 **Opciones válidas:**  
✔ **En el segmento de código (Codigo Segment).**  
✔ **En otro segmento de código de una librería (OtroCodigo Segment).**  
❌ **No en el segmento de datos (Datos Segment)** (a menos que haya una razón específica).

📌 **Ejemplo correcto de declaración en un segmento:**

Codigo Segment

Assume CS:Codigo, DS:Datos

procedimiento1 PROC NEAR

mov al, 'X'

ret

procedimiento1 ENDP

inicio:

call procedimiento1

FinProgra

Codigo Ends

End inicio

**3. ¿Cómo evitar que el IP entre en un procedimiento automáticamente?**

Si un procedimiento está **antes de inicio**, el programa podría ejecutarlo directamente.  
Para evitarlo, se usa **un JMP Short inicio** para saltarlo y llegar al punto correcto del programa.

📌 **Ejemplo con JMP para evitar la ejecución automática:**

Codigo Segment

Assume CS:Codigo, DS:Datos

JMP Short inicio ; Salta el procedimiento al inicio

procedimiento1 PROC NEAR

mov al, 'X'

ret

procedimiento1 ENDP

inicio:

IniciaDatos datos

call procedimiento1

FinProgra

Codigo Ends

End inicio

✅ **Ahora el programa comienza en inicio, sin ejecutar automáticamente procedimiento1.**

**4. ¿Cómo manejar variables dentro de un procedimiento en una librería?**

El profesor plantea un problema:  
*"Si tengo una librería sin segmento de datos, ¿cómo uso una variable en un procedimiento?"*

🔹 **Las macros funcionan sin problema porque se expanden en el código principal.**  
🔹 **Pero los procedimientos necesitan un segmento válido para acceder a variables.**

📌 **Posibles soluciones (que se explicarán más adelante):**

1. **Usar parámetros en la pila** para pasar valores a un procedimiento.
2. **Acceder a una variable global en otro segmento**.
3. **Reservar espacio en la pila dentro del procedimiento** (PUSH / POP).

Esto es un tema avanzado, pero **se resolverá en lecciones futuras**.