**📌 1️.Paso de Parámetros por Registro (AL, AX)**

✔ **La forma más simple es usar registros (AL, AX, BX, etc.) para pasar valores entre procedimientos.**  
✔ **El problema es que los registros pueden modificarse dentro del procedimiento, perdiendo el valor original.**  
✔ **Para evitar esto, se usa la pila (STACK) para almacenar los valores de manera segura.**

**📌 2️.Uso de la Pila para Pasar Parámetros (STACK)**

✔ **La pila permite almacenar datos antes de llamar a un procedimiento y recuperarlos después.**  
✔ **Cada valor almacenado en la pila ocupa 2 bytes (WORD), incluso si solo necesitamos 1 byte (BYTE).**  
✔ **Al llamar un procedimiento FAR, la pila almacena automáticamente CS (segmento de código) y IP (puntero de instrucción).**

🔹 **Estructura de la pila después de CALL FAR**

Dirección Contenido Desplazamiento (`BP+X`)

-----------------------------------------------------

FFFC carácterLeído BP+6

FFFE carácterTecleado BP+4

FFFA IP de retorno BP+2

FFF8 CS de retorno BP+0

✔ **Cada WORD ocupa 2 bytes, por eso los desplazamientos (BP+X) aumentan en múltiplos de 2.**

**📌 3️.Acceso a los Parámetros en la Pila con BP**

✔ **No se puede acceder a los valores directamente con SP, porque SP cambia durante la ejecución.**  
✔ **Por eso, BP (Base Pointer) se usa para acceder a los valores de la pila sin modificar SP.**

🔹 **Código dentro del procedimiento FAR para acceder a los parámetros**

PUSH BP ; Guarda el valor actual de BP

MOV BP, SP ; BP apunta al tope de la pila

MOV BYTE PTR [BP+4], AL ; Guarda el carácter en carácterTecleado

MOV BYTE PTR [BP+6], AL ; Guarda el carácter en carácterLeído

POP BP ; Restaura BP

RETF ; Retorna del procedimiento FAR

✔ **BP+4 apunta al primer parámetro (carácterTecleado) y BP+6 al segundo (carácterLeído).**  
✔ **Se usa BYTE PTR porque AL es de 8 bits y los parámetros en la pila están en WORD (16 bits).**

**📌 4️.Error con POP BYTE PTR y Solución**

✔ **La pila en x86 trabaja con WORDS (16 bits), por lo que POP BYTE PTR genera un error.**  
✔ **La solución es hacer POP AX para extraer un WORD, y luego mover solo el BYTE necesario.**

🔹 **Código Incorrecto (Genera Error)**

POP BYTE PTR carácterLeído ; ❌ ERROR: `POP` no puede extraer solo 1 byte.

🔹 **Código Correcto**

POP AX ; Extraer 16 bits de la pila

MOV BYTE PTR carácterLeído, AL ; Guardar solo 8 bits en carácterLeído

✔ **Esto permite recuperar correctamente los valores sin errores en la pila.**

**📌 5️.Verificación y Resultados**

✔ Después de ejecutar CALL LeerCaracter, se verificó que:

* BX contenía 61h (ASCII de a).
* carácterLeído = 61h (97 en decimal).
* carácterTecleado = 61h (97 en decimal).  
  ✔ **Esto confirma que los valores fueron correctamente transferidos y almacenados en memoria.**

**¿Qué se Explicará en la Próxima Lección?**

📌 **Revisión del proceso para reforzar el concepto.**  
📌 **Por qué es necesario usar este método con procedimientos FAR.**  
📌 **Comparación entre diferentes métodos de paso de parámetros (Registro vs Pila).**  
📌 **Ventajas y desventajas de este método en programas más grandes.**

**¿Qué hace este código?**

Parece estar realizando un **intercambio de valores (swap)** de tecla 1 y tecla 2, pero utilizando una variable temporal (otra variable). En términos de pseudocódigo:

otra\_variable = tecla1

tecla1 = tecla2

tecla2 = otra\_variable

Esto es un **swap clásico usando una variable temporal**, común en lenguajes de bajo nivel cuando no se dispone de operaciones de intercambio directas.

**¿Qué hace este código?**

El código intercambia los valores de tecla 1 y tecla 2, pero necesita un registro intermedio porque no se puede hacer un acceso directo entre dos ubicaciones de memoria. La secuencia en pseudocódigo sería algo así:

MOV AH, tecla1 ; Guarda tecla1 en AH

MOV AL, tecla2 ; Guarda tecla2 en AL

MOV tecla1, AL ; Mueve el valor de tecla2 a tecla1

MOV tecla2, AH ; Mueve el valor original de tecla1 a tecla2

Básicamente, **es un "swap" manual usando un registro intermedio** (AH en este caso).  
Este es un procedimiento común en ensamblador cuando se necesita intercambiar valores sin acceso directo entre direcciones de memoria.

**Análisis del Código**

Parece que estás describiendo el uso de la pila (stack) en ensamblador para intercambiar valores entre dos números sin usar una variable auxiliar explícita. Vamos a desglosarlo paso a paso:

1. **"PUSH de número 1 a la pila"**
   * Guarda el valor de número 1 en la pila para su uso posterior.
2. **"Muevo A número 2 le muevo al número 1"**
   * número 1 = número 2
   * En este punto, número 1 ha sido sobrescrito con el valor de número 2.
3. **"POP de número 1"**
   * Recupera el valor original de número 1 desde la pila y lo asigna nuevamente a número 2.

**¿Qué hace este código?**

Este código efectúa **un intercambio de valores entre número 1 y número 2 usando la pila en lugar de un registro intermedio**. Es equivalente a:

PUSH número1 ; Guarda número 1 en la pila

MOV número1, número2 ; Copia número 2 en número 1

POP número2 ; Recupera el valor original de número 1 y lo asigna a número 2

Al final, los valores de número 1 y número 2 se han intercambiado.  
Este método de swap es útil cuando se quiere ahorrar registros y aprovechar la pila para almacenamiento temporal.

**Concepto de Pila (Stack)**

Una pila es una estructura de datos que sigue el principio **LIFO (Last In, First Out)**, lo que significa que el último elemento en entrar es el primero en salir.

**Operaciones básicas en una pila:**

1. **PUSH (apilar):** Agrega un elemento al tope de la pila.
2. **POP (desapilar):** Retira el elemento del tope de la pila.
3. **TOP (peek o mirar):** Muestra el elemento en la cima sin retirarlo.

 **Principio de la pila:**

* Sigue la regla **LIFO (Last In, First Out)** → el último elemento en entrar es el primero en salir.

 **Ejemplo con bolas de colores:**

* Se agrega una **bolita roja** → luego una **bolita verde** → luego una **bolita azul**.
* Si la pila ya está llena, no se puede agregar otra (**bolita amarilla** en este caso).
* Esto causa un **Stack Overflow**, que ocurre cuando se intenta agregar más elementos de los que la pila puede contener.

 **Stack Overflow:**

* Ocurre cuando la pila alcanza su límite de memoria, por ejemplo, **65,536 bytes (64 KB)** en ciertos sistemas de 16 bits.

 **Extracción de elementos (POP):**

* Primero se retira la **bolita azul** (última en entrar).
* Luego la **bolita verde**.
* Finalmente la **bolita roja**.
* No se puede sacar directamente la **bolita roja** sin antes retirar las demás, porque la pila es estricta en seguir el orden **LIFO**.

**Análisis del Código**

1. **Inicialización de registros con XOR**

XOR AX, AX

XOR BX, BX

XOR CX, CX

XOR DX, DX

* + Se están limpiando (poniendo en 0) los registros AX, BX, CX, y DX.
  + **XOR con el mismo registro siempre da 0**, lo que es una forma eficiente de inicializar registros sin usar MOV.

1. **Asignación de valores a los registros**

MOV AX, 1

MOV BX, 2

MOV CX, 3

MOV DX, 4

* + AX = 1
  + BX = 2
  + CX = 3
  + DX = 4

1. **Almacenamiento de registros en la pila (PUSH)**

PUSH AX

PUSH BX

PUSH CX

PUSH DX

* + Se guardan los valores de los registros en la pila.
  + **Orden de almacenamiento en la pila (de arriba hacia abajo)**:

| DX (4) |

| CX (3) |

| BX (2) |

| AX (1) |

* + **Último en entrar (DX) será el primero en salir (LIFO).**

1. **Limpieza de los registros nuevamente**

XOR AX, AX

XOR BX, BX

XOR CX, CX

XOR DX, DX

* + Se ponen en **0** nuevamente los registros.

1. **Restauración de valores desde la pila (POP)**
   * Aunque no está escrito en el fragmento, lo más lógico sería hacer lo siguiente para restaurar los valores:

POP DX

POP CX

POP BX

POP AX

* + **Orden de recuperación** (primero sale DX, luego CX, luego BX y finalmente AX), lo cual **recupera los valores originales**.

1. **Guarda los valores de los registros AX, BX, CX, DX en la pila** usando PUSH.
2. **Limpia los registros con XOR para ponerlos en 0.**
3. **(Falta en el código) Recupera los valores originales con POP en orden inverso.**
4. **Demuestra el uso del stack en x86: Último en entrar, primero en salir (LIFO).**

**Ejemplo de salida esperada si imprimimos los valores después del POP:**

AX = 1

BX = 2

CX = 3

DX = 4

1. **Inicialización de registros**
   * Se establece AX = 1, BX = 2, CX = 3, DX = 4.
   * **Antes de cargar estos valores, todos los registros estaban en 0**.
2. **Uso del PUSH para almacenar los registros en la pila**

PUSH AX

PUSH BX

PUSH CX

PUSH DX

* + Se guardan los valores en la pila en este orden:

| DX (4) | <- Último en entrar (primero en salir)

| CX (3) |

| BX (2) |

| AX (1) | <- Primero en entrar (último en salir)

* + Como la pila funciona con **LIFO (Last In, First Out)**, **el último registro que entró (DX) será el primero en salir al hacer POP**.

1. **Cómo afecta el Stack Pointer (SP)**
   * Cada vez que se hace un PUSH, el SP **se decrementa en 2 bytes** (porque los registros en x86 de 16 bits ocupan 2 bytes).
   * **Ejemplo del cambio en SP:**
     + Antes del primer PUSH, SP = 0000h.
     + Después de PUSH AX, SP = FFFEh.
     + Después de PUSH BX, SP = FFFCh.
     + Después de PUSH CX, SP = FFFAh.
     + Después de PUSH DX, SP = FFF8h.
2. **Uso del POP para recuperar valores**
   * Para recuperar los valores en orden inverso:

POP DX ; DX recibe el valor 4

POP CX ; CX recibe el valor 3

POP BX ; BX recibe el valor 2

POP AX ; AX recibe el valor 1

* + Esto es **lo esperado** porque sigue el orden **LIFO** (último en entrar, primero en salir).

**¿Qué significa el Stack Segment (SS) y el Stack Pointer (SP)?**

* **SS (Stack Segment)**: Es el segmento de memoria asignado para la pila.
* **SP (Stack Pointer)**: Es un puntero que indica la posición actual del tope de la pila.

Cuando el SP está en 0000h, significa que la pila **está vacía o en su estado inicial**.

Si SP llega a un valor muy bajo y se intenta hacer otro PUSH, se puede producir un **Stack Overflow**, lo cual **sobreescribirá memoria no asignada para la pila**, causando errores o fallos en el sistema.

**Errores y Soluciones**

1. **El "88" que aparece en la salida**
   * Puede ser un valor residual en la pila o un problema de alineación de memoria.
   * Es posible que se haya intentado acceder a una dirección de memoria incorrecta o que algún valor en SP no estuviera correctamente inicializado antes de hacer PUSH y POP.
2. **"Vamos a comentar esta línea..."**
   * Puede referirse a deshabilitar una parte del código que estaba causando un problema, probablemente relacionado con el mal manejo de SP.

**¿Cómo se almacenan los datos en la pila?**

1. **Posición inicial de la pila (SP = 0000h)**
   * La pila en x86 **crece hacia abajo**, lo que significa que cada vez que se hace un PUSH, SP **se decrementa**.
2. **Ejemplo de valores almacenados en la pila**
   * Se almacena AX = 1
   * Se almacena BX = 2
   * Cada PUSH hace que el SP decrezca en 2 bytes.
3. **Direcciones de memoria después de los PUSH**
   * PUSH AX → SP cambia de 0000h a FFFEh, almacenando el 1 en la dirección FFFEh.
   * PUSH BX → SP cambia de FFFEh a FFFCh, almacenando el 2 en FFFCh.

**Ejemplo en memoria después de estos PUSH:**

Dirección Contenido

FFFC 2 (BX)

FFFE 1 (AX)

* + Se puede notar que el Stack Pointer (SP) **decrementa de 2 en 2** porque cada registro en x86 de 16 bits ocupa **2 bytes**.

**Explicación de las direcciones mencionadas en el audio**

* **"00-1 sería la posición FFFF"**
  + Como la pila crece hacia abajo, si SP estaba en 0000h, hacer SP - 1 lo llevaría a FFFFh.
* **"00-2 sería la posición FFFE"**
  + Como cada PUSH resta **2 bytes**, el primer PUSH de AX lo almacena en FFFEh.
* **"En la posición FFFE, metió el 1"**
  + Esto confirma que AX = 1 se ha almacenado correctamente en FFFEh.
* **"En la posición FFFC, va a meter un 2, porque le estoy haciendo un PUSH BX"**
  + Después de PUSH BX, BX = 2 se almacena en la dirección FFFCh.

 **Asignación de valores a registros**

* AX = 1, BX = 2, CX = 3, DX = 4

 **Almacenar valores en la pila con PUSH**

* PUSH AX → SP = FFFEh, almacena 1 en FFFEh
* PUSH BX → SP = FFFCh, almacena 2 en FFFCh
* PUSH CX → SP = FFFAh, almacena 3 en FFFAh
* PUSH DX → SP = FFF8h, almacena 4 en FFF8h

**Estado de la pila después de los PUSH:**

Dirección Contenido

FFF8 4 (DX)

FFFA 3 (CX)

FFFC 2 (BX)

FFFE 1 (AX)

 **Concepto de LIFO (Last In, First Out)**

* Como la pila sigue la regla **LIFO**, **el último valor almacenado (DX = 4) será el primero en salir**.

 **Restaurar registros con POP**

* Se pone AX, BX, CX, DX en **cero** antes de recuperar valores.
* **El primer POP saca el valor en SP = FFF8h (que es 4) y lo coloca en DX**:

POP DX ; DX = 4

* Después de este POP, SP aumenta en 2 → SP = FFFAh.

**Proceso de recuperación con POP**

Después de hacer:

POP DX ; DX = 4

El SP aumenta en 2 (SP = FFFA), lo que significa que ahora apunta al valor **3** en FFFA.

1. **Sacar el valor 3 y asignarlo a CX:**

POP CX ; CX = 3

* + SP ahora apunta a FFFC (donde está el 2).

1. **Sacar el valor 2 y asignarlo a BX:**

POP BX ; BX = 2

* + SP ahora apunta a FFFE (donde está el 1).

1. **Sacar el valor 1 y asignarlo a AX:**

POP AX ; AX = 1

* + SP ahora vuelve a 0000, indicando que la pila está vacía.

**Estado de los registros después de los POP**

Después de estos POP, los valores originales han sido restaurados:

AX = 1

BX = 2

CX = 3

DX = 4

SP = 0000 (pila vacía)

Esto demuestra cómo la pila funciona en **ensamblador x86** y **cómo se pueden almacenar y recuperar valores de forma eficiente**.

1. **Se almacenan valores en la pila**
   * Se asignan valores a los registros (AX = 1, BX = 2, CX = 3, DX = 4).
   * Se hace PUSH de cada uno en orden.
   * **Después de los PUSH, el SP queda en FFF8 (debido a la forma en que decrece con cada PUSH).**
2. **Se limpian los registros (AX, BX, CX, DX = 0)**
   * Esto prepara la memoria para hacer POP y ver cómo los valores son recuperados.
3. **Se extraen los valores de la pila usando POP**
   * SP apunta a FFF8, donde está el 4 → Se asigna 4 a AX.
   * Luego 3 a BX, 2 a CX, 1 a DX.
   * **Esto confirma que la pila funciona bajo LIFO, pero se puede manipular.**

**Manipulación Directa de la Pila**

El profesor menciona que **la pila en ensamblador no es estrictamente inmutable**. Es decir, aunque PUSH y POP siguen un orden LIFO, se pueden modificar los valores directamente para hacer **intercambios sin necesidad de extraer y volver a insertar los valores**.

**Ejemplo de intercambio en la pila (Swap sin registros intermedios)**

En lugar de hacer múltiples POP y PUSH, se podría hacer algo como:

; Guardamos los valores en la pila

PUSH número1 ; Número 1 en la pila

PUSH número2 ; Número 2 en la pila

; Manipulación de la pila para intercambiar valores directamente

MOV AX, [SP] ; Cargar número2 desde la pila

MOV BX, [SP+2] ; Cargar número1 desde la pila

MOV [SP], BX ; Intercambiar en la pila

MOV [SP+2], AX ; Intercambiar en la pila

; Recuperamos los valores en orden intercambiado

POP número1 ; Ahora número1 tiene el valor de número2

POP número2 ; Ahora número2 tiene el valor de número1

**¿Por qué es importante esta manipulación?**

1. **Ahorra instrucciones**: No es necesario hacer múltiples POP y PUSH si solo queremos cambiar valores.
2. **Mejora la eficiencia**: Acceder directamente a la pila es más rápido en algunos casos.
3. **Se usa en optimización de llamadas a funciones**: Muchas veces, parámetros en la pila son manipulados directamente sin necesidad de restaurarlos uno por uno.

**Cómo Funciona el Intercambio en la Pila**

1. **Se hace PUSH de número 1 y luego PUSH de número 2**

PUSH número1

PUSH número2

* + Esto almacena número 1 y número 2 en la pila en este orden:

Dirección Contenido

FFFE número 1

FFFC número 2

* + **Nota:** La pila crece hacia abajo, así que número 2 es el último en entrar y el primero en salir.

1. **Se hace POP en orden inverso para intercambiar los valores**

POP número1

POP número2

* + Como POP sigue la regla **LIFO**, número 2 será asignado a número 1 y número 1 será asignado a número 2, logrando un **swap sin una tercera variable**.

**Explicación del Uso de Word PTR**

* **Word PTR tecla 1**
  + **Un Word en x86 es de 16 bits**, por lo que la pila solo almacena valores de **16 bits a la vez**.
  + Si tecla 1 es de **8 bits (1 byte)** y tecla 2 es de **16 bits (2 bytes, palabra completa)**, **al hacer PUSH, se ajustan los tamaños para que todo encaje en 16 bits**.
* **"Tecla 1 vale un byte y tecla 2 es de Debite"**
  + **Probablemente "Debite" se refería a "Double Byte" (2 bytes, 16 bits)**.
  + Como la pila **no puede almacenar bytes sueltos (solo palabras de 16 bits o más)**, al hacer PUSH tecla 1, el sistema toma **dos bytes seguidos** en memoria para completarlo a un WORD.

**Paso a Paso del Intercambio en la Pila**

1. **Valores iniciales**
   * Número 1 = 255
   * Número 2 = 254
2. **Se almacena en la pila con PUSH**

PUSH número1 ; Guarda 255 en FFFC

PUSH número2 ; Guarda 254 en FFFA

* + Ahora la pila tiene:

Dirección Contenido

FFFA 254 (Número 2)

FFFC 255 (Número 1)

1. **Se extraen los valores con POP (pero en orden inverso)**

POP número1 ; Número 1 = 254 (valor incorrecto)

POP número2 ; Número 2 = 255 (valor correcto)

* + Esto **los intercambia**, pero el problema es que ahora **Número 1 tiene un valor incorrecto (254 en lugar de 255)**.

1. **Problema: Pila inconsistente**
   * El SP quedó en **FFFA**, lo que significa que **no sacó todos los valores correctamente**.
   * **Causa:** Se hizo PUSH dos veces, pero al hacer POP, se sacaron en el orden incorrecto.

**Soluciones para evitar una pila inconsistente**

**1. Extraer los valores correctamente**

* Asegurarse de hacer POP en el orden correcto, de acuerdo con **LIFO**.

PUSH número1

PUSH número2

POP número2

POP número1

* Esto garantiza que los valores se intercambien sin errores.

**2. Restaurar el SP manualmente**

* Si se hizo PUSH dos veces pero solo un POP, el SP estará desfasado.
* Se puede corregir manualmente con:

ADD SP, 2 ; Avanza el SP para "descartar" un valor de la pila

* Esto hace que el SP vuelva a su posición correcta sin afectar los registros.

**3. Limpiar la pila al final**

* Si se quiere **resetear la pila** para evitar valores residuales, se pueden hacer POP adicionales o **asignar un valor nulo**:

XOR AX, AX

MOV SP, 0000 ; Reinicia la pila (peligroso si afecta otras funciones)

* Sin embargo, **resetear la pila directamente puede causar problemas si hay valores importantes almacenados**.

**Problema Original**

1. Se hizo **PUSH más veces de lo que se hizo POP**, dejando valores extra en la pila.
2. **El SP quedó desfasado** (no volvió a su posición inicial).
3. Intentar hacer **POP de nada** es una forma de corregir el problema sin afectar registros importantes.

**Soluciones para Restaurar la Pila Correctamente**

**1. Hacer POP a una variable "basura"**

* Si hay un valor extra en la pila, se puede hacer un POP y almacenarlo en una variable irrelevante:

POP OtraBar ; Saca el valor sobrante de la pila y lo almacena en OtraBar

* Esto garantiza que el SP avance **sin alterar registros importantes**.

**2. Hacer POP en el orden correcto**

* Si se hizo PUSH dos veces, hay que asegurarse de hacer POP dos veces:

PUSH número1

PUSH número2

POP número2

POP número1

* + Esto mantiene la pila consistente.

**3. Descartar el valor manualmente**

* Si no se necesita el valor sobrante, se puede ajustar el SP directamente:

ADD SP, 2 ; Descartar 1 valor extra de la pila

* + Esto **avanza el SP manualmente**, evitando el POP.

**4. Verificar el estado del SP**

* El profesor menciona que **si el SP inició en 0000h, debe terminar en 0000h**.
* Para asegurarse, se puede imprimir el valor del SP antes y después del código para verificar que no haya cambios inesperados.

**Soluciones Correctas para Mantener la Pila Consistente**

**1. Extraer todos los valores correctamente (POP en el orden correcto)**

* Si hiciste **dos PUSH, necesitas dos POP**:

PUSH número1

PUSH número2

POP número2 ; Recupera número2

POP número1 ; Recupera número1

* **Esto garantiza que la pila vuelve a su estado inicial.**

**2. Usar un POP para descartar el valor sobrante**

* Si hay un valor que no se necesita, se puede hacer:

POP basura ; Saca el valor restante de la pila y lo almacena en una variable irrelevante

* **Así, el ESP se ajusta correctamente sin afectar valores importantes.**

**3. Ajustar el ESP manualmente**

* Si no se hizo POP suficiente, se puede avanzar el ESP manualmente:

ADD ESP, 2 ; Ajusta el puntero de pila para ignorar un valor extra

* **Esto es útil si no queremos perder un registro en el POP.**

**4. Comprobar el estado del ESP antes y después**

* El profesor menciona que **el ESP debe terminar en el mismo valor en el que empezó (0000h)**.
* Para verificarlo:

MOV AX, ESP

* Se puede imprimir el valor de ESP antes y después del código para asegurarse de que quedó consistente.

1. **Inicialización del Segmento de Datos**
   * Antes de hacer una llamada (CALL), el **Stack Pointer (ESP)** está en 0000h, lo que significa que la pila está vacía.
2. **Ejemplo de CALL y cómo afecta la pila**
   * Cuando se ejecuta un CALL, **la dirección de retorno (IP) se guarda en la pila** antes de saltar a la función.
   * Si el IP (Instruction Pointer) estaba en la **posición 19h**, al hacer CALL, la pila almacena esta dirección de retorno.
   * Luego, el IP se incrementa para ejecutar la siguiente instrucción.

Dirección de memoria Contenido

FFFEh 0019h ; Dirección de retorno (IP antes del CALL)

1. **Cálculo de direcciones en memoria**
   * CALL guarda la dirección actual (IP = 19h) en la pila.
   * Luego, IP avanza en el código:
     + **19h + 1 → 1Ah**
     + **1Ah + 1 → 1Bh**
     + **1Bh + 1 → 1Ch**
   * **Por eso el IP está en 1Ch después del CALL.**

**¿Cómo se regresa de la función? (RET)**

1. Cuando se ejecuta RET:
   * **Saca la dirección almacenada en la pila (IP = 19h) con POP IP**.
   * **Salta de vuelta a la dirección almacenada (IP = 19h)**.
   * **El ESP aumenta en 2 para limpiar la pila**.

**¿Qué es un Procedimiento NEAR en Ensamblador x86?**

En ensamblador x86, una **llamada a procedimiento (CALL) puede ser de dos tipos**:

1. **NEAR (Cercana)**
   * El **procedimiento (subrutina) está dentro del mismo segmento de código**.
   * Solo se almacena la **dirección de retorno (IP) en la pila**.
   * **Ejemplo**:

CALL procedimiento\_near ; Salta a la función dentro del mismo segmento

1. **FAR (Lejana)**
   * El procedimiento está en **otro segmento de código**.
   * Se almacenan en la pila **el segmento (CS) y la dirección de retorno (IP)**.
   * **Ejemplo**:

CALL procedimiento\_far ; Salta a otra función en otro segmento

**Paso a Paso de lo que ocurre con CALL en un procedimiento NEAR**

1. **Antes de la llamada (CALL)**
   * El IP (Instruction Pointer) está en **001Ch**.
   * Cuando se ejecuta CALL, el procesador **almacena la dirección de retorno (001Ch) en la pila**.
2. **¿Dónde se almacena la dirección de retorno?**
   * Como la pila **crece hacia abajo**, SP se **decrementa en 2**.
   * La dirección 001Ch se almacena en FFFEh:

Dirección Contenido

FFFEh 001Ch ; Dirección de retorno después del procedimiento

* + **Luego, el IP salta a la dirección donde está el procedimiento**.

**¿Qué pasa al ejecutar RET?**

1. **Cuando la subrutina termina, RET saca la dirección de retorno de la pila**:

RET

1. **El IP recupera el valor almacenado (001Ch), regresando al código original.**
2. **El ESP aumenta en 2, restaurando el estado de la pila.**

**1. Procedimiento NEAR (NIR en el audio)**

✔ **Un procedimiento NEAR (cercano) está dentro del mismo segmento de código**.  
✔ **Cuando se ejecuta CALL, solo guarda el IP en la pila**.  
✔ **Cuando se ejecuta RET, extrae el IP y vuelve a la dirección original**.

**📌 Flujo de ejecución con un procedimiento NEAR**

1. **Antes del CALL**, el IP (Instruction Pointer) es **001Ch**.
2. **Se ejecuta CALL**, lo que hace:
   * Guarda IP = 001Ch en la pila (SP se reduce en 2).
   * Salta a la dirección del procedimiento.
3. **El procedimiento ejecuta su código (bla bla bla)**.
4. **Se ejecuta RET**, lo que hace:
   * Recupera IP = 001Ch desde la pila.
   * **Salta de vuelta a 001Ch**, reanudando la ejecución normal.

**2. Procedimiento FAR (P2 en el código)**

✔ **Un procedimiento FAR está en otro segmento de código**.  
✔ **Cuando se ejecuta CALL, guarda el CS (Code Segment) y IP en la pila**.  
✔ **Cuando se ejecuta RET, extrae el IP y CS y salta de vuelta al código original.**

**📌 Flujo de ejecución con un procedimiento FAR**

1. **Antes del CALL**, el IP es **0020h** y el CS es **087Eh**.
2. **Se ejecuta CALL**, lo que hace:
   * **Guarda CS = 087Eh en la pila**.
   * **Guarda IP = 0020h en la pila**.
   * Salta a la dirección del procedimiento en otro segmento.
3. **El procedimiento ejecuta su código**.
4. **Se ejecuta RET**, lo que hace:
   * Recupera IP = 0020h desde la pila.
   * Recupera CS = 087Eh desde la pila.
   * **Salta de vuelta al código original en el segmento correcto**.

**📌 Diferencias Clave entre CALL NEAR y CALL FAR**

| **Característica** | **CALL NEAR** | **CALL FAR** |
| --- | --- | --- |
| Ubicación del procedimiento | Mismo segmento | Otro segmento |
| Datos guardados en la pila | Solo IP (2 bytes) | CS + IP (4 bytes) |
| Regreso con RET | Extrae solo IP | Extrae CS e IP |
| Cambio de segmento | No | Sí |

**📌 Resumen de CALL en Procedimientos FAR**

1. **Se ejecuta CALL en un procedimiento FAR**
   * Se almacena el CS (Code Segment) y el IP (Instruction Pointer) en la pila.
   * **Ejemplo de valores almacenados:**

Dirección Contenido

FFFC 0020 ; IP (dirección de retorno)

FFFE 087E ; CS (segmento de retorno)

* + Luego, el procesador **salta al nuevo segmento de código** para ejecutar el procedimiento.

1. **Se ejecuta el procedimiento (bla bla bla)**
   * El código dentro del procedimiento FAR se ejecuta normalmente.
2. **Se ejecuta RET y se sacan los valores de la pila**
   * RET extrae **primero el IP y luego el CS** para regresar al punto original del CALL:

POP IP ; Recupera la dirección de retorno (0020h)

POP CS ; Recupera el segmento de código original (087Eh)

* + Luego **salta de regreso al segmento correcto y al IP original**.

**📌 Diferencias entre CALL NEAR y CALL FAR**

| **Característica** | **CALL NEAR** | **CALL FAR** |
| --- | --- | --- |
| Ubicación del procedimiento | Mismo segmento | Otro segmento |
| Datos guardados en la pila | Solo IP (2 bytes) | CS + IP (4 bytes) |
| Regreso con RET | Extrae solo IP | Extrae CS e IP |
| Cambio de segmento | No | Sí |
| Tamaño del CALL | 3 bytes (E8) | 5 bytes (9A) |

**📌 ¿Cómo Saber si un Procedimiento es NEAR o FAR?**

1. **Si el procedimiento no tiene etiqueta (P4 en este caso), se considera NEAR.**
2. **Si tiene una etiqueta FAR, se almacena CS y IP en la pila.**

**📌 Explicación con CALL P4 (Procedimiento NEAR)**

✔ **Como P4 no tiene etiqueta FAR, se asume que es NEAR.**  
✔ **CALL P4 guarda solo el IP en la pila.**  
✔ **Después del RET, extrae el IP y continúa la ejecución.**

**Flujo de Ejecución de un Procedimiento NEAR**

CALL P4 ; Salta a P4 y guarda solo el IP en la pila

...

P4 PROC ; Definición del procedimiento NEAR (sin etiqueta)

; Código del procedimiento

RET ; Extrae IP y regresa

ENDP

**📌 Explicación con CALL P2 (Procedimiento FAR)**

✔ **Como P2 es FAR, se almacena el CS y el IP en la pila.**  
✔ **Después del RET, extrae el CS y IP para regresar al segmento correcto.**

**Flujo de Ejecución de un Procedimiento FAR**

CALL P2 ; Salta a P2 y guarda CS + IP en la pila

...

P2 PROC FAR ; Definición del procedimiento FAR

; Código del procedimiento

RETF ; Extrae CS + IP y regresa

ENDP

**📌 Resumen de las Diferencias**

| **Característica** | **CALL NEAR** | **CALL FAR** |
| --- | --- | --- |
| Segmento de ejecución | Mismo segmento | Diferente segmento |
| Datos guardados en la pila | Solo IP (2 bytes) | CS + IP (4 bytes) |
| Regreso con RET | Extrae solo IP | Extrae CS e IP |
| Uso de etiquetas | No necesita FAR | Debe declararse como FAR |
| Cambio de CS | No | Sí |

**📌 Explicación del Comportamiento**

1. **Si un procedimiento está dentro del mismo segmento y no tiene la etiqueta FAR → se considera NEAR.**
   * Solo **almacena IP en la pila** cuando se llama.
   * RET solo extrae IP y vuelve a la ejecución normal.
2. **Si un procedimiento está en otro segmento o tiene la etiqueta FAR → se considera FAR.**
   * Al hacer CALL, **se almacenan CS y IP en la pila**.
   * RETF saca ambos valores y regresa al punto correcto.

**📌 Ubicación y Estructura en el Código**

1. **Si el procedimiento P1 está dentro del código principal (NEAR)**:

CALL P1 ; Salta a P1

...

P1 PROC ; Definición de P1 (NEAR por defecto)

; Código de P1

RET ; Extrae solo IP de la pila

ENDP

* + Solo almacena **IP en la pila**.
  + **Ejemplo de pila tras CALL P1:**

Dirección Contenido

FFFEh 0030 ; IP (dirección de retorno)

1. **Si el procedimiento P2 está en otra sección y es FAR**:

CALL P2 FAR ; Salta a P2

...

P2 PROC FAR ; Procedimiento en otro segmento

; Código de P2

RETF ; Extrae CS + IP

ENDP

* + **Guarda CS y IP en la pila.**
  + **Ejemplo de pila tras CALL P2:**

Dirección Contenido

FFFC 0030 ; IP (dirección de retorno)

FFFE 087E ; CS (segmento de retorno)

**📌 Clave: Ubicación en el Código**

* **Si el procedimiento está al final del programa pero dentro del mismo segmento**, sigue siendo NEAR.
* **Si el procedimiento está en otro segmento**, se debe marcar como FAR, y CALL almacenará CS + IP.
* **Los procedimientos NEAR son el comportamiento predeterminado si no se especifica FAR.**

**📌 Diferencia Clave entre NEAR y FAR**

| **Característica** | **CALL NEAR** | **CALL FAR** |
| --- | --- | --- |
| Ubicación | Dentro del mismo segmento | Otro segmento |
| Datos en la pila | Solo IP (2 bytes) | CS + IP (4 bytes) |
| Regreso con | RET | RETF |
| Cambia de segmento | No | Sí |
| Necesita FAR en la declaración | No | Sí |

**📌 Concepto de una Librería de Procedimientos en Ensamblador**

✔ **Una librería de procedimientos es un conjunto de funciones (PROCs) que pueden ser reutilizadas en diferentes programas.**  
✔ **Puede estar en el mismo archivo (NEAR) o en otro segmento (FAR).**  
✔ **Se pueden almacenar en archivos separados (.ASM), ensamblarlos y enlazarlos (.OBJ).**  
✔ **Se utilizan directivas como EXTERN y PUBLIC para la comunicación entre módulos.**

**📌 Estructura de una Librería de Procedimientos**

Una **librería de procedimientos en ensamblador** se puede organizar en **módulos**.

**1️⃣ Archivo de la Librería (procedimientos.asm)**

.MODEL SMALL

.STACK 100h

.DATA

mensaje DB "Hola desde la librería!", 0Dh, 0Ah, "$"

.CODE

PUBLIC MostrarMensaje ; Hacer que el procedimiento sea accesible desde otros archivos

MostrarMensaje PROC FAR

PUSH DS

MOV DX, OFFSET mensaje

MOV AH, 09h

INT 21h

POP DS

RETF

MostrarMensaje ENDP

END

✔ Este código **define un procedimiento FAR** que imprime un mensaje.  
✔ **PUBLIC MostrarMensaje** permite que otros archivos puedan llamar a esta función.

**2️. Archivo Principal del Programa (main.asm)**

.MODEL SMALL

.STACK 100h

.DATA

.CODE

EXTERN MostrarMensaje: FAR ; Indica que el procedimiento está en otro archivo

MAIN PROC

CALL MostrarMensaje ; Llamada a la función desde la librería

MOV AX, 4C00h

INT 21h

MAIN ENDP

END MAIN

✔ **EXTERN MostrarMensaje: FAR** indica que MostrarMensaje está en otro archivo.  
✔ **El CALL MostrarMensaje llama al procedimiento de la librería.**  
✔ **Se usa RETF en la librería porque el procedimiento es FAR (está en otro segmento).**

**📌 Cómo Ensamblar y Enlazar los Módulos**

Para crear la librería y el programa principal, se deben **ensamblar y enlazar**:

1. **Compilar cada archivo por separado:**

tasm procedimientos.asm

tasm main.asm

1. **Enlazar ambos archivos en un solo ejecutable:**

tlink main.obj procedimientos.obj

1. **Ejecutar el programa:**

main.exe

**📌 ¿Debería Declarar Segmentos Dentro del PROC en una Librería de Procedimientos?**

La respuesta corta es **no**. **Los segmentos no deben declararse dentro de cada PROC**, sino a nivel de la librería en su conjunto.

**📌 ¿Por qué una Librería de Procedimientos Necesita un Segmento?**

✔ **Una librería de procedimientos en ensamblador necesita un segmento de código (.CODE)** donde se definirán todas las funciones.  
✔ **Cada procedimiento dentro de la librería debe ser declarado PUBLIC** si se va a usar desde otro módulo.  
✔ **Los segmentos de datos (.DATA) y pila (.STACK) pueden declararse en el programa principal** para evitar redundancia.

**📌 Estructura Correcta de una Librería de Procedimientos en Ensamblador x86**

Una librería bien estructurada debe incluir:

1. **Declaración del modelo de memoria y segmentos**
2. **Definición de procedimientos (PUBLIC para exportarlos)**
3. **Uso de END para finalizar el módulo**

**📌 Ejemplo: Librería de Entrada (input.asm)**

.MODEL SMALL

.CODE ; El segmento de código de la librería

PUBLIC LeerCaracter ; Declarar el procedimiento como público

LeerCaracter PROC FAR

MOV AH, 01h ; Interrupción para leer un carácter

INT 21h

RETF ; Retorna de procedimiento FAR

LeerCaracter ENDP

END

✔ **El segmento se define una vez (.CODE) y no dentro de cada PROC.**  
✔ **PUBLIC LeerCaracter permite que el programa principal acceda al procedimiento.**  
✔ **RETF se usa porque el procedimiento es FAR.**

**📌 Cómo Usar la Librería en el Programa Principal (main.asm)**

.MODEL SMALL

.STACK 100h

.DATA

mensaje DB "Presiona una tecla: $"

.CODE

EXTERN LeerCaracter: FAR ; Importar el procedimiento desde la librería

MAIN PROC

MOV DX, OFFSET mensaje

MOV AH, 09h

INT 21h ; Imprimir mensaje

CALL LeerCaracter ; Llamar al procedimiento de la librería

MOV AX, 4C00h

INT 21h ; Salir del programa

MAIN ENDP

END MAIN

✔ **EXTERN LeerCaracter: FAR indica que LeerCaracter está en otro archivo.**  
✔ **No se redefine el segmento de código dentro del PROC.**

**📌 ¿Por qué el Concepto de PUBLIC fue "Robado" por Otros Lenguajes?**

El concepto de **PUBLIC y EXTERN** en ensamblador es **el origen de las funciones exportadas e importadas en lenguajes modernos** como:  
✔ extern en C/C++  
✔ public y import en Java, C#, Python

Básicamente, **las librerías dinámicas (DLLs) y estáticas (.LIB) modernas funcionan de la misma forma que las librerías en ensamblador**.

**📌 Manejo de Procedimientos FAR, Segmentos de Datos y Paso de Parámetros en Ensamblador x86**

El profesor está explicando los desafíos de trabajar con **procedimientos FAR**, **segmentos de datos incompatibles** y **cómo se pasan parámetros entre módulos en ensamblador x86**.

**📌 Problema Principal**

✔ **Un procedimiento FAR está en otro segmento, lo que significa que no puede acceder directamente a las variables del programa principal.**  
✔ **Si el procedimiento LeerCaracter está en otro segmento, no puede modificar carácter leído directamente.**  
✔ **Se debe pasar información entre módulos usando registros o la pila.**

**📌 Solución 1: Paso de Parámetros por Registro**

✔ **Se usa AL para pasar el carácter leído entre módulos.**  
✔ **El procedimiento LeerCaracter devuelve el valor en AL.**

**Ejemplo de Procedimiento FAR que Retorna un Carácter**

.MODEL SMALL

.CODE

PUBLIC LeerCaracter

LeerCaracter PROC FAR

MOV AH, 01h ; Interrupción DOS para leer carácter

INT 21h ; AL contiene el carácter leído

RETF ; Retorna a otro segmento

LeerCaracter ENDP

END

✔ **AL contendrá el carácter leído cuando el procedimiento termine.**  
✔ **No se usa una variable de datos porque los segmentos son incompatibles.**

**📌 Solución 2: Paso de Parámetros con EXTERN y CALL FAR**

✔ **El programa principal debe declarar EXTERN para usar el procedimiento FAR.**

**Ejemplo del Programa Principal**

.MODEL SMALL

.STACK 100h

.DATA

caracterLeido DB ? ; Variable para almacenar el carácter

.CODE

EXTERN LeerCaracter: FAR ; Importar el procedimiento desde otro módulo

MAIN PROC

CALL LeerCaracter ; Llama al procedimiento FAR

MOV caracterLeido, AL ; Guarda el carácter leído

MOV AX, 4C00h

INT 21h ; Salir del programa

MAIN ENDP

END MAIN

✔ **CALL LeerCaracter llama al procedimiento en otro segmento.**  
✔ **El carácter leído se almacena en AL y luego se guarda en caracterLeido.**

**📌 Manejo de Librerías de Procedimientos vs. Librerías de Macros**

1. **Librería de Procedimientos (CALL FAR o NEAR)**
   * Se compilan en módulos separados (.OBJ).
   * Usa PUBLIC y EXTERN para comunicación entre módulos.
   * Se pueden reutilizar en múltiples programas.
2. **Librería de Macros (MACRO)**
   * Se define en el mismo archivo fuente (.ASM).
   * Se expanden en línea en el código (sin CALL).
   * No necesita EXTERN, pero hace el código más grande.

**📌 Cómo Compilar y Enlazar una Librería de Procedimientos en Ensamblador x86**

El profesor está explicando el **proceso de compilación y ensamblado de un programa en ensamblador con procedimientos separados**. También menciona un **error de compilación sobre una "clase diferente"**, lo cual generalmente ocurre cuando un procedimiento FAR y su declaración EXTERN no coinciden.

**📌 Pasos para Compilar y Enlazar Módulos Separados en Ensamblador x86**

Cuando el código está dividido en **una librería de procedimientos (Proce.ASM) y un programa principal (Main.ASM)**, el proceso de compilación tiene **dos fases**:

1. **Compilación de cada módulo (TASM o MASM)**
2. **Enlace de los archivos .OBJ (TLINK)**

**📌 1️⃣ Compilación de la Librería de Procedimientos (Proce.ASM)**

El archivo **Proce.ASM** debe incluir:  
✔ **Un segmento de código (.CODE)**  
✔ **Procedimientos PUBLIC para exportarlos**  
✔ **Finalizar correctamente con END**

Ejemplo de **Proce.ASM (Librería de Procedimientos)**

.MODEL SMALL

.CODE

PUBLIC LeerCaracter ; Hacer público el procedimiento

LeerCaracter PROC FAR

MOV AH, 01h ; Interrupción para leer un carácter

INT 21h ; AL contiene el carácter leído

RETF ; Retorna de procedimiento FAR

LeerCaracter ENDP

END

**Compilar con TASM o MASM:**

tasm /zi /l Proce.ASM

✔ **El parámetro /zi incluye información de depuración.**  
✔ **El parámetro /l genera un archivo de lista (.LST).**  
✔ **Se genera Proce.OBJ, que será usado en el siguiente paso.**

**📌 2️⃣ Compilación del Programa Principal (Main.ASM)**

El archivo **Main.ASM** debe incluir:  
✔ **Declarar EXTERN para importar el procedimiento**  
✔ **Usar CALL FAR si el procedimiento está en otro segmento**  
✔ **Finalizar con END MAIN**

Ejemplo de **Main.ASM (Programa Principal)**

.MODEL SMALL

.STACK 100h

.DATA

caracterLeido DB ? ; Variable para almacenar el carácter

.CODE

EXTERN LeerCaracter: FAR ; Importar el procedimiento desde otro módulo

MAIN PROC

CALL LeerCaracter ; Llama al procedimiento FAR

MOV caracterLeido, AL ; Guarda el carácter leído

MOV AX, 4C00h

INT 21h ; Salir del programa

MAIN ENDP

END MAIN

**Compilar con TASM o MASM:**

tasm /zi /l Main.ASM

Esto generará **Main.OBJ**, que se enlazará en el siguiente paso.

**📌 3️.Enlazar los Archivos .OBJ con TLINK**

Después de compilar los dos archivos .ASM, hay que enlazarlos en un **ejecutable final** (.EXE):

tlink /v Main.OBJ Proce.OBJ

✔ **El parámetro /v agrega depuración al ejecutable.**  
✔ **Esto generará Main.EXE, que se puede ejecutar.**

**📌 Posible Error: "El símbolo es de diferente clase y no está usado"**

Este error suele ocurrir si hay una **desincronización entre PUBLIC y EXTERN**, como por ejemplo:

❌ **Error Común: EXTERN y PUBLIC no coinciden**

assembly

; En la librería de procedimientos (Proce.ASM)

PUBLIC LeerCaracter ; Procedimiento declarado como FAR

LeerCaracter PROC FAR

MOV AH, 01h

INT 21h

RETF

LeerCaracter ENDP

; En el programa principal (Main.ASM)

EXTERN LeerCaracter: NEAR ; ERROR: Aquí se declaró como NEAR

✔ **Solución:** Asegurar que EXTERN y PUBLIC coincidan:

EXTERN LeerCaracter: FAR

Si el error persiste, revisar si LeerCaracter **está correctamente escrito en ambos archivos** y si la compilación de Proce.ASM fue exitosa.

**📌 Solución al Problema de Interferencia entre un Procedimiento y una Macro en Ensamblador x86**

El profesor menciona un problema de **nombres en conflicto** entre un procedimiento (LeerCaracter) y una macro con el mismo nombre. **Este tipo de interferencias pueden generar errores en la compilación**.

**📌 ¿Por qué ocurre esta interferencia?**

✔ **En ensamblador, los nombres de macros y procedimientos comparten el mismo espacio de nombres.**  
✔ **Si una macro (MACRO) y un procedimiento (PROC) tienen el mismo nombre, el ensamblador puede confundirse sobre cuál usar.**  
✔ **La solución es renombrar uno de ellos para evitar conflictos.**

**📌 Solución: Renombrar la Macro**

✔ **El profesor cambia el nombre de la macro a LeerCaracterM para evitar la interferencia.**  
✔ **Ahora, LeerCaracterM se usará como macro, y LeerCaracter seguirá siendo el procedimiento FAR.**

LeerCaracterM MACRO

MOV AH, 01h

INT 21h

ENDM

✔ **Esto evita el conflicto con el procedimiento LeerCaracter PROC.**

**📌 Confirmación del Error en la Línea 191**

Después de renombrar la macro, si el error persiste en la línea **191**, es recomendable:

1. **Verificar la línea donde se usa CALL LeerCaracter**
2. **Confirmar que EXTERN LeerCaracter: FAR está bien escrito**
3. **Asegurar que LeerCaracter y LeerCaracterM no se llaman accidentalmente en el mismo contexto**

**📌 Proceso de Compilación y Enlace Explicado**

1. **Se ejecuta TASM con los parámetros /ZI y /L**
   * **/ZI**: Incluye información de depuración.
   * **/L**: Genera un archivo de lista para analizar símbolos.

tasm /zi /l primero.asm

tasm /zi /l proce.asm

1. **Se enlazan los módulos con TLINK**
   * **Falta el parámetro /V**, lo cual genera el error **"No hay tabla de símbolos"**.

tlink /v primero.obj proce.obj

* + **Solución:** Agregar /V para habilitar símbolos y depuración.

1. **Al ejecutar el programa (primero.exe), se llama LeerCaracter**
   * Se observa cómo CALL LeerCaracter almacena **CS (Code Segment) y IP (Instruction Pointer)** en la pila.
   * **Ejemplo de valores en la pila tras CALL:**

Dirección Contenido

FFFC 001E ; IP de retorno (dirección después del `CALL`)

FFFE 087B ; CS de retorno (segmento del programa principal)

**📌 Cambio de Segmento al Llamar un Procedimiento FAR**

✔ **Antes de CALL LeerCaracter, CS = 087E**  
✔ **Después del CALL, CS cambia a 0893 porque el procedimiento LeerCaracter está en otro segmento.**

CALL LeerCaracter ; Salta al segmento 0893h

✔ **Esto demuestra que CALL FAR cambia CS para ejecutar código en otro segmento.**

**📌 Resultado: Captura de Carácter en AL**

✔ **El usuario ingresa una a minúscula (ASCII 97 o 61h).**  
✔ **El procedimiento almacena este valor en AL.**  
✔ **Se verifica en memoria que AL = 97 (61h), lo que confirma que la interrupción INT 21h funcionó correctamente.**

MOV AH, 01h ; Leer carácter

INT 21h ; AL contiene el carácter leído

**📌 Paso de Parámetros por Registro en Ensamblador x86**

El profesor está explicando **cómo se pasa información entre procedimientos en ensamblador x86 usando registros en lugar de macros**. Este enfoque se llama **"Paso de Parámetros por Registro"** y es una técnica común en ensamblador para **optimizar el intercambio de datos entre funciones**.

**📌 Explicación Paso a Paso**

1. **El procedimiento LeerCaracter lee un carácter y lo almacena en AL**

MOV AH, 01h ; Llamar a la interrupción DOS para leer un carácter

INT 21h ; El carácter leído se almacena en AL

RETF ; Retorna al programa principal

✔ **Después de ejecutar CALL LeerCaracter, AL contiene el carácter leído.**

1. **El programa principal captura AL y lo almacena en carácter leído**

CALL LeerCaracter ; Llama al procedimiento FAR

MOV carácterLeído, AL ; Guarda el carácter leído en la variable

✔ **Se transfiere el valor de AL a la variable carácterLeído.**  
✔ **Ahora carácterLeído en el segmento de datos contiene 97 (61h en ASCII), que corresponde a a minúscula.**

**📌 Diferencias entre Paso de Parámetros por Registro y por Macros**

| **Característica** | **Paso por Registro** | **Paso por Macros** |
| --- | --- | --- |
| Forma de pasar datos | Usa registros (AL, AX, etc.) | Usa nombres de macros |
| Flexibilidad | Más eficiente en procedimientos reutilizables | Más simple en código inline |
| Complejidad | Necesita más control manual | Se expande automáticamente en el código |
| Uso en segmentación | Funciona bien con CALL NEAR/FAR | No necesita CALL, ya que es inline |

✔ **El paso por registro es útil cuando se trabaja con procedimientos FAR, ya que evita problemas de acceso a memoria entre segmentos.**  
✔ **Las macros son útiles para pequeñas rutinas repetitivas donde no es necesario un CALL.**

**📌 Restricciones en el Acceso a Variables entre Segmentos en Ensamblador x86**

El profesor está explicando **por qué un procedimiento FAR no puede acceder directamente a las variables del programa principal** y cómo esto **complica el paso de parámetros** en ensamblador.

**📌 ¿Por qué un Procedimiento FAR no Puede Acceder Directamente a una Variable del DATA Segment?**

✔ **Cada segmento de código (CODE) y datos (DATA) tiene su propia dirección de memoria.**  
✔ **Cuando un procedimiento FAR se ejecuta, cambia el CS (Code Segment), pero DS (Data Segment) no cambia automáticamente.**  
✔ **El procedimiento FAR no "ve" el segmento de datos del programa principal.**  
✔ **Incluso si dos programas tienen segmentos DATA, pueden llamarse diferente (datos, data, pajarito, etc.), lo que hace que el acceso directo sea imposible.**

**📌 ¿Cómo Funciona el Servicio de Interrupción INT 21H con AH = 01H?**

✔ **Este servicio (INT 21H, AH=01H) lee un carácter desde el teclado y lo devuelve en AL.**  
✔ **Esto significa que después de ejecutar la interrupción, AL contendrá el carácter ingresado.**

MOV AH, 01h ; Configurar interrupción DOS para leer un carácter

INT 21h ; AL ahora contiene el carácter leído

✔ **Si este código está en el mismo segmento, AL se puede mover directamente a una variable.**  
✔ **Si está en otro segmento (FAR), no se puede hacer MOV carácterLeído, AL directamente.**

**📌 Solución: Paso de Parámetros Manual entre Segmentos**

✔ **Ya vimos la opción de pasar datos con registros (AL).**  
✔ **Ahora veremos una opción más avanzada: pasar parámetros a través de la pila (STACK).**

**📌 Ejemplo de Paso de Parámetros con la Pila**

En la **Librería de Procedimientos (proce.asm)**:

.MODEL SMALL

.CODE

PUBLIC LeerCaracter

LeerCaracter PROC FAR

MOV AH, 01h ; Leer un carácter

INT 21h ; AL contiene el carácter

PUSH AX ; Guardar el valor en la pila

RETF

LeerCaracter ENDP

END

✔ **Aquí PUSH AX almacena el carácter leído en la pila antes de regresar.**

En el **Programa Principal (main.asm)**:

.MODEL SMALL

.STACK 100h

.DATA

carácterLeído DB ? ; Variable para almacenar el carácter

.CODE

EXTERN LeerCaracter: FAR ; Importar el procedimiento

MAIN PROC

CALL LeerCaracter ; Llamar al procedimiento FAR

POP AX ; Recuperar el carácter desde la pila

MOV carácterLeído, AL ; Guardarlo en la variable

MOV AX, 4C00h

INT 21h ; Salir del programa

MAIN ENDP

END MAIN

✔ **Se usa POP AX en el programa principal para recuperar el valor almacenado en la pila por PUSH AX.**  
✔ **Ahora AL contiene el carácter y puede guardarse en carácterLeído.**

**📌 ¿Quién Maneja la Dirección de Memoria en Ensamblador x86?**

El profesor está explicando **cómo se enlazan y organizan los segmentos en ensamblador x86**, y **cómo TLINK genera un archivo .MAP que contiene la ubicación exacta de cada segmento en memoria**.

**📌 ¿Quién Maneja las Direcciones de Memoria en Ensamblador?**

✔ **El enlazador (TLINK) es el que asigna las direcciones de memoria a cada segmento**.  
✔ **Cuando se compila y enlaza el programa (TLINK /V), se genera un archivo .MAP** que muestra:

* **Dónde empieza y termina cada segmento (DATA, CODE, PROCEDIMIENTOS, etc.).**
* **Cómo se asignan las direcciones de memoria en un sistema de 16 bits.**

**📌 ¿Qué Contiene el Archivo .MAP?**

El archivo **PRIMERO.MAP** contiene **el mapa de direcciones de memoria** de los segmentos del programa.

1. **Ejemplo de un .MAP generado por TLINK**

Start Stop Length Name

087E0 0893F 00160 \_CODE

08940 08AFF 001C0 PROCEDIMIENTOS

08B00 08BFF 00100 \_DATA

* + \_CODE **(087E0 - 0893F)** → Segmento de código del programa principal.
  + PROCEDIMIENTOS **(08940 - 08AFF)** → Segmento de la librería de procedimientos.
  + \_DATA **(08B00 - 08BFF)** → Segmento de datos.

1. **¿Por qué hay dos segmentos de código?**
   * Uno corresponde al **código del programa principal** (\_CODE).
   * El otro corresponde a **los procedimientos (PROCEDIMIENTOS)** definidos en otro módulo.

**📌 ¿Cómo se Calculan las Direcciones de Memoria en un Sistema de 16 bits?**

✔ En un procesador **8086**, las direcciones de memoria son **de 20 bits**.  
✔ Sin embargo, los registros de segmento (CS, DS, SS, ES) **solo tienen 16 bits**.  
✔ Para convertir **direcciones de 16 bits en direcciones de 20 bits**, se usa la **segmentación de memoria**:

**📌 Cálculo de Direcciones de 20 Bits**

La dirección física se obtiene con:

Dirección física = (Segmento × 16) + Offset

✔ **Ejemplo:** Si CS = 0893h y IP = 0010h:

Dirección física = (0893h × 10h) + 0010h

= 08930h + 0010h

= 08940h

✔ **Por eso, en el .MAP, todas las direcciones empiezan con 0 para mantener el formato de 20 bits.**

**📌 ¿Por qué no se Puede Conocer la Dirección de Memoria en Tiempo de Compilación en Ensamblador x86?**

El profesor está explicando **por qué no se puede conocer con precisión la dirección de memoria en tiempo de compilación** y **cómo en tiempo de ejecución se pueden hacer trucos avanzados (como los virus de los 90)**.

**📌 1️¿Por qué la Dirección de Memoria No Se Conoce en Tiempo de Compilación?**

✔ **Cuando se escribe código en ensamblador, las direcciones de memoria específicas no están definidas.**  
✔ **El ensamblador (TASM o MASM) genera código relativo, sin asignarle direcciones físicas.**  
✔ **El enlazador (TLINK) es quien finalmente asigna las direcciones en el .MAP en función del diseño de memoria.**  
✔ **Las direcciones finales dependen del sistema operativo y de cómo se cargue el programa en memoria.**

🔴 **Ejemplo de código relativo en ensamblador:**

MOV AX, [MI\_VARIABLE] ; En tiempo de compilación, no sabemos la dirección real de MI\_VARIABLE

✔ **El ensamblador no coloca la dirección física, sino una referencia.**  
✔ **Cuando el programa se carga en memoria, TLINK resuelve la dirección real.**

**📌 2️¿Se Puede Determinar la Dirección en Tiempo de Ejecución?**

✔ Sí, pero **solo cuando el programa ya está corriendo**, accediendo a CS, DS, o SS.  
✔ Se puede usar instrucciones como LEA, LDS, LES o acceder a la tabla de segmentos.

🔵 **Ejemplo de cómo obtener la dirección del segmento en tiempo de ejecución:**

MOV AX, CS ; Obtener el segmento de código actual

MOV DX, DS ; Obtener el segmento de datos actual

✔ **Esto solo funciona en modo real (como DOS o DOSBox), pero no en sistemas modernos como Windows en modo protegido.**

**📌 3️¿Cómo se Manipulaban las Direcciones en Virus de los Años 90?**

✔ **En los 90, los virus podían acceder a direcciones de memoria específicas porque el sistema corría en Modo Real (8086).**  
✔ **Se podía leer o escribir directamente en sectores del disco usando INT 13H**.  
✔ **Un truco común era modificar la tabla de direcciones (IVT en 0000:0400h) para interceptar interrupciones.**

🔴 **Ejemplo de acceso al disco en modo real (usado en algunos virus de los 90):**

MOV AH, 02h ; Leer desde el disco

MOV AL, 01h ; Número de sectores a leer

MOV CH, 00h ; Cilindro

MOV CL, 01h ; Sector

MOV DH, 00h ; Cabeza

MOV DL, 80h ; Disco (0 = floppy, 80h = HDD)

INT 13h ; Interrupción BIOS para acceder al disco

✔ **Hoy esto no funciona porque Windows está en Modo Protegido y bloquea accesos directos al hardware.**  
✔ **Si intentas acceder al disco directamente en Windows, el sistema se bloquea o lanza un error.**

**📌 4️¿Por Qué Hoy es Más Difícil Hacer Esto?**

✔ **Windows y Mac usan Modo Protegido (Protected Mode), que impide acceso directo a la memoria y hardware.**  
✔ **Los programas no pueden modificar memoria fuera de su espacio asignado.**  
✔ **Los sistemas modernos usan Memory Address Randomization para evitar ataques.**  
✔ **Se requieren permisos especiales o explotar vulnerabilidades para acceder a la memoria del sistema.**

**📌 Paso de Parámetros por Pila en Ensamblador x86**

El profesor está introduciendo **el paso de parámetros por pila (STACK)** como una alternativa al paso por registro. **Este método es más seguro** porque evita que el valor se sobrescriba accidentalmente cuando se usa un procedimiento CALL.

**📌 Problema del Paso de Parámetros por Registro (AL)**

✔ **Cuando un procedimiento CALL se ejecuta, los registros pueden ser modificados.**  
✔ **Si AL contiene el carácter leído, pero otro código modifica AL, el valor original se pierde.**  
✔ **Para evitar esto, se debe almacenar el valor en la pila antes de modificar registros.**

**📌 Solución: Paso de Parámetros por Pila (STACK)**

✔ **En la pila, los datos se almacenan en WORDS (2 bytes), pero un carácter (BYTE) solo ocupa 1 byte.**  
✔ **Para evitar problemas de alineación, se puede almacenar un WORD con 0xFF en el byte alto.**

**📌 Opciones para Manejar Caracteres (BYTE) en la Pila**

1. **Convertir carácter leído en WORD**

carácterLeído DW ? ; Se define como WORD (2 bytes)

* + **Ventaja:** Se puede hacer PUSH carácterLeído sin problemas de tamaño.
  + **Desventaja:** Se desperdicia un byte de memoria.

1. **Almacenar un BYTE en la Pila con un WORD**

MOV AH, 00h ; Limpiar el byte alto

PUSH AX ; Guardar el carácter leído en la pila como WORD

* + **Ventaja:** Se mantiene el alineamiento de la pila.
  + **Desventaja:** Se requiere limpiar AH antes de PUSH.

**📌 Implementación del Paso de Parámetros por Pila**

**📌 1️⃣ Procedimiento LeerCaracter (FAR)**

.MODEL SMALL

.CODE

PUBLIC LeerCaracter

LeerCaracter PROC FAR

MOV AH, 01h ; Leer un carácter

INT 21h ; AL contiene el carácter leído

MOV AH, 00h ; Asegurar que se almacene un WORD completo

PUSH AX ; Guardar el carácter en la pila

RETF

LeerCaracter ENDP

END

✔ **Este código almacena AL en la pila como WORD (AX con AH=0).**  
✔ **Evita que el valor se pierda si AL se sobrescribe.**

**📌 2️⃣ Programa Principal (main.asm)**

.MODEL SMALL

.STACK 100h

.DATA

carácterLeído DB ? ; Variable para almacenar el carácter

.CODE

EXTERN LeerCaracter: FAR ; Importar el procedimiento

MAIN PROC

CALL LeerCaracter ; Llamar al procedimiento FAR

POP AX ; Recuperar el carácter de la pila

MOV carácterLeído, AL ; Guardarlo en la variable

MOV AX, 4C00h

INT 21h ; Salir del programa

MAIN ENDP

END MAIN

✔ **Después de CALL LeerCaracter, el POP AX recupera el carácter almacenado en la pila.**  
✔ **MOV carácterLeído, AL guarda solo el byte bajo en la variable.**

**📌 Uso de BYTE PTR para Acceder a Datos en Ensamblador x86**

El profesor está mostrando cómo **usar BYTE PTR para acceder correctamente a un BYTE dentro de una WORD almacenada en la pila**. Esto es importante porque **la pila en x86 almacena valores de 16 bits (WORD), pero en algunos casos solo se necesita acceder a 1 byte (BYTE)**.

**📌 ¿Qué Hace BYTE PTR?**

✔ **BYTE PTR indica explícitamente que se está accediendo a un BYTE, aunque la memoria esté alineada a WORD (2 bytes).**  
✔ **Se usa cuando una variable es de tipo BYTE, pero la instrucción por defecto trata de acceder a WORD.**

**📌 Ejemplo de Uso de BYTE PTR en el Paso de Parámetros por Pila**

MOV BYTE PTR carácterLeído, AL

✔ **Esto asegura que AL (que es de 8 bits) se almacene correctamente en carácterLeído sin afectar otros bytes de memoria.**  
✔ **Si carácterLeído estuviera en un WORD, BYTE PTR evitaría modificar el byte alto accidentalmente.**

**📌 ¿Por Qué Usar BYTE PTR en la Pila?**

Cuando se usa la pila (STACK), la memoria se maneja en WORDS (16 bits).  
Si se almacena un BYTE en la pila, el valor realmente ocupa **2 bytes** en memoria, por lo que hay que asegurarse de **leer solo el byte correcto**.

🔵 **Ejemplo de Error Común (Sin BYTE PTR)**

POP carácterLeído ; ERROR: Trata de escribir un WORD (16 bits)

✔ **Solución con BYTE PTR**

POP AX ; Recupera 16 bits de la pila

MOV BYTE PTR carácterLeído, AL ; Guarda solo el byte bajo (8 bits)

✔ **BYTE PTR evita que se sobrescriban datos innecesarios.**

**📌 Paso de Parámetros por Pila Usando BYTE PTR y WORD PTR en Ensamblador x86**

El profesor está explicando **cómo almacenar y recuperar datos correctamente en la pila** cuando los parámetros tienen diferentes tamaños (BYTE o WORD).

**📌 ¿Por Qué Usar BYTE PTR en Paso de Parámetros por Pila?**

✔ La pila almacena datos en **palabras (WORD, 16 bits)**.  
✔ Cuando un valor es un **BYTE (8 bits)**, hay que asegurarse de no modificar la parte alta accidentalmente.  
✔ **BYTE PTR se usa para acceder solo al byte bajo (AL), sin alterar otros datos.**

🔵 **Ejemplo de Cómo Guardar un BYTE en una Variable:**

MOV BYTE PTR carácterTecleado, AL ; Guardar solo 1 byte

✔ **Esto mueve AL (8 bits) a carácterTecleado, sin afectar otros bytes en memoria.**

**📌 Almacenar un BYTE en la Pila**

✔ **Como la pila solo acepta WORD (16 bits), hay que asegurarse de meterlo correctamente.**

🔵 **Método 1: Convertir BYTE a WORD Antes de PUSH**

MOV AH, 00h ; Limpiar byte alto

PUSH AX ; Almacenar en la pila como WORD

✔ **Esto guarda AL como WORD en la pila (AH=0 evita datos basura).**

**📌 Almacenar un WORD en la Pila**

✔ Si la variable ya es WORD, no hay problema con PUSH.

🔵 **Ejemplo de PUSH Directo con WORD PTR**

PUSH WORD PTR carácterLeído ; Almacenar en la pila

✔ **Como carácterLeído ya es WORD, no se necesita conversión.**

**📌 Flujo Completo: Leer Caracter, Guardar en la Pila y Recuperarlo**

1️**Leer el carácter y guardarlo en una variable (BYTE)**

MOV AH, 01h

INT 21h ; AL contiene el carácter leído

MOV BYTE PTR carácterTecleado, AL ; Guardar en la variable

2️**Convertir BYTE a WORD y Guardarlo en la Pila**

MOV AH, 00h

PUSH AX ; Almacenar en la pila

3️**Recuperar el Valor desde la Pila y Guardarlo en carácterLeído**

POP AX ; Extraer valor de la pila

MOV BYTE PTR carácterLeído, AL ; Guardarlo correctamente

✔ **Ahora carácterLeído contiene el mismo valor que carácterTecleado.**

**📌 Paso de Parámetros por Pila en Ensamblador x86 (Usando BP y SP)**

El profesor está explicando **cómo manejar correctamente los parámetros en la pila** al llamar a un **procedimiento FAR**, utilizando **BP (Base Pointer) para acceder a los valores sin modificar SP (Stack Pointer).**

**📌 ¿Cómo se Organiza la Pila Antes de Llamar al Procedimiento?**

Cuando se llama a un procedimiento FAR, **la pila contiene los siguientes valores en orden:**

Dirección Contenido

--------------------

[SP] Carácter tecleado (WORD)

[SP+2] Carácter leído (WORD)

[SP+4] Dirección de Retorno (IP)

[SP+6] Segmento de Retorno (CS)

✔ **Cada valor en la pila ocupa 2 bytes (WORD).**  
✔ **El CS y IP son almacenados automáticamente por CALL FAR.**

**📌 Paso de Parámetros por Pila: Código Explicado**

**1️⃣ Antes de Llamar al Procedimiento**

Antes de ejecutar CALL, **los parámetros se pasan a la pila en orden inverso**:

PUSH WORD PTR carácterTecleado ; Paso carácter tecleado a la pila

PUSH WORD PTR carácterLeído ; Paso carácter leído a la pila

CALL LeerCaracter ; Llamar al procedimiento FAR

✔ **Ahora, la pila tiene los parámetros correctamente organizados.**

**2️⃣ Dentro del Procedimiento FAR**

En el procedimiento, **no se puede modificar SP directamente** porque es el puntero de la pila.  
Para acceder a los parámetros, **se usa BP (Base Pointer) como referencia estable.**

LeerCaracter PROC FAR

PUSH BP ; Guardar el valor actual de BP

MOV BP, SP ; BP ahora apunta al tope de la pila

✔ **BP se usa como referencia fija mientras el procedimiento usa la pila.**  
✔ **Ahora BP tiene la dirección de los parámetros.**

**3️⃣ Acceder a los Parámetros en la Pila**

Para acceder a los valores pasados en la pila, **se usa BP con desplazamientos (BP+X)**:

MOV AX, [BP+4] ; Obtener carácter tecleado

MOV BX, [BP+6] ; Obtener carácter leído

✔ **Se usa BP+4 porque BP apunta a CS, y los parámetros están 4 bytes más abajo.**  
✔ **Se usa BP+6 para el siguiente parámetro.**

**4️⃣ Restaurar la Pila y Salir**

Cuando el procedimiento termina, se debe **restaurar BP antes de RET**:

POP BP ; Restaurar BP al valor original

RETF ; Retornar al programa principal

✔ **Esto asegura que la pila se mantenga estable.**

**📌 Paso de Parámetros por Pila con BP en Ensamblador x86 (Accediendo a los Valores con BYTE PTR)**

El profesor está mostrando **cómo se acceden a los valores en la pila usando BP con desplazamientos** (BP+4, BP+6, etc.), y cómo usar BYTE PTR para modificar solo un BYTE dentro de un WORD.

**📌 ¿Cómo se Organiza la Pila Antes de Ejecutar el Procedimiento?**

Antes de llamar al procedimiento FAR, **se pasan los parámetros a la pila en orden inverso**:

PUSH WORD PTR carácterTecleado ; Parámetro 1

PUSH WORD PTR carácterLeído ; Parámetro 2

CALL LeerCaracter ; Llamar al procedimiento FAR

✔ **La pila queda organizada así (de arriba hacia abajo, direcciones decrecientes):**

Dirección Contenido

---------------------

FFFE carácterTecleado (WORD)

FFFC carácterLeído (WORD)

FFFA IP de Retorno

FFF8 CS de Retorno

✔ **Cada WORD ocupa 2 bytes, por lo que los desplazamientos son múltiplos de 2 (BP+4, BP+6, etc.).**

**📌 Cómo Acceder a los Parámetros en la Pila con BP**

1. **Antes de acceder a los valores, BP se usa como referencia fija**:

PUSH BP ; Guardar BP actual

MOV BP, SP ; BP ahora apunta a la pila

✔ **Esto evita modificar SP, asegurando estabilidad en la pila.**

1. **Se accede a los parámetros con desplazamientos (BP+4, BP+6)**:

MOV BYTE PTR [BP+4], AL ; Mueve el valor de AL a carácterTecleado

MOV BYTE PTR [BP+6], AL ; Mueve el valor de AL a carácterLeído

✔ **Se usa BYTE PTR porque AL es un BYTE y carácterTecleado está en un WORD.**  
✔ **Si no se usara BYTE PTR, la instrucción intentaría escribir 16 bits en lugar de 8 bits.**

**📌 ¿Cómo se Calculan los Desplazamientos?**

✔ BP+0 → Apunta a CS (Segmento de Retorno)  
✔ BP+2 → Apunta a IP (Dirección de Retorno)  
✔ BP+4 → Apunta a carácterTecleado  
✔ BP+6 → Apunta a carácterLeído

**Ejemplo con valores en la pila:**

Dirección Contenido BP+X

---------------------------------

FFFE carácterTecleado BP+4

FFFC carácterLeído BP+6

FFFA IP de Retorno BP+2

FFF8 CS de Retorno BP+0

✔ **Cada WORD ocupa 2 bytes, por eso los desplazamientos aumentan en múltiplos de 2.**

**📌 Código Completo del Procedimiento LeerCaracter**

LeerCaracter PROC FAR

PUSH BP ; Guardar BP actual

MOV BP, SP ; BP ahora apunta a la pila

MOV AH, 01h ; Leer un carácter

INT 21h ; AL contiene el carácter leído

MOV BYTE PTR [BP+4], AL ; Guardar carácter en carácterTecleado

MOV BYTE PTR [BP+6], AL ; Guardar carácter en carácterLeído

POP BP ; Restaurar BP

RETF ; Retornar al programa principal

LeerCaracter ENDP

✔ **El procedimiento accede correctamente a los parámetros usando BP y BYTE PTR.**  
✔ **BYTE PTR evita modificar más bytes de los necesarios en memoria.**  
✔ **Al final, BP se restaura antes de RETF para mantener la pila estable.**

**📌 Explicación Detallada del Paso de Parámetros por Pila y la Manipulación de BP en Ensamblador x86**

El profesor está mostrando **cómo se alinean los valores en la pila, cómo BP ayuda a acceder a los parámetros y cómo los valores se mueven correctamente en memoria.**

**📌 1️Organización de la Pila Antes de la Llamada al Procedimiento FAR**

Cuando se ejecuta un CALL FAR, la pila contiene lo siguiente en orden:

Dirección Contenido Desplazamiento desde BP

-----------------------------------------------------

FFFC carácterLeído BP+6

FFFE carácterTecleado BP+4

FFFA IP de Retorno BP+2

FFF8 CS de Retorno BP+0

✔ **Cada WORD ocupa 2 bytes, por lo que los desplazamientos (BP+X) aumentan en múltiplos de 2.**  
✔ **El BP se usa como referencia para acceder a estos valores sin modificar SP.**

**📌 2️Cómo BP Permite Acceder a los Parámetros**

Dentro del procedimiento FAR, **el BP se alinea con SP** para mantener referencia fija en la pila.

MOV BP, SP ; BP ahora apunta a la pila

✔ **Después de este movimiento, BP y SP apuntan al mismo lugar.**

Para acceder a los valores de los parámetros, se usa:

MOV BYTE PTR [BP+4], AL ; Guarda AL en carácterTecleado

MOV BYTE PTR [BP+6], AL ; Guarda AL en carácterLeído

✔ **Esto permite escribir en la memoria sin afectar otros valores en la pila.**  
✔ **Se usa BYTE PTR porque AL es de 8 bits y los parámetros son WORD (16 bits).**

**📌 3️.Ejecución Paso a Paso**

🔹 **Antes del CALL**

* **CS = 087F**, almacenado en FFFA.
* **IP = 0026**, almacenado en FFF8.
* **carácterLeído = FF00**, almacenado en FFFC.
* **carácterTecleado = 0000**, almacenado en FFFE.

🔹 **Después de CALL LeerCaracter**

* Se almacena CS e IP en la pila.
* SP y BP ahora apuntan a FFF8 (dirección de retorno).
* Para acceder a carácterLeído:

BP + 6 = FFFC (donde está el valor FF00)

* Se mueve AL (61h o 97 en decimal) a esta posición.

🔹 **Salida del Procedimiento**

* POP BP para restaurar el puntero de base.
* RETF saca IP y CS de la pila y salta de vuelta al programa.

**📌 Corrección del Error en el POP BYTE PTR y Solución para el Paso de Parámetros por Pila en Ensamblador x86**

* El profesor encontró un **error en la instrucción POP BYTE PTR**, que impide extraer un solo byte de la pila. **Esto se debe a que la pila en x86 trabaja con WORD (16 bits), y POP solo puede operar con registros o WORD en memoria**.

**📌 1️.¿Por Qué No Se Puede Hacer POP BYTE PTR Directamente?**

* ✔ La pila almacena valores en **WORDS (2 bytes o 16 bits)**.  
  ✔ **POP solo extrae WORDS (16 bits) a la vez**, no BYTES (8 bits).  
  ✔ Intentar hacer POP BYTE PTR genera un error porque **POP no puede escribir solo 1 byte en memoria**.

**📌 2️.Solución: Extraer WORD y Luego Mover Solo el BYTE Requerido**

✔ **Se hace POP AX para sacar los 16 bits de la pila.**  
✔ **Luego, se mueve AL (byte bajo) a carácterLeído usando BYTE PTR.**

**Código Correcto:**

POP AX ; Extraer WORD (16 bits) de la pila

MOV BYTE PTR carácterLeído, AL ; Mover solo el byte bajo (8 bits)

✔ **Esto funciona porque POP AX extrae un WORD completo, pero MOV BYTE PTR selecciona solo los 8 bits inferiores (AL).**

**📌 3️¿Qué Pasa si Se Quieren Extraer Dos Valores de la Pila?**

✔ **Si hay dos parámetros (carácterLeído y carácterTecleado), se hace doble POP:**

**Código Completo para Extraer Ambos Valores**

POP AX ; Extraer primer parámetro (carácterLeído)

MOV BYTE PTR carácterLeído, AL ; Guardar solo el byte bajo en carácterLeído

POP AX ; Extraer segundo parámetro (carácterTecleado)

MOV BYTE PTR carácterTecleado, AL ; Guardar solo el byte bajo en carácterTecleado

✔ **Cada POP AX saca 16 bits, pero MOV BYTE PTR solo usa los 8 bits necesarios.**

* **📌 4️.¿Qué Hacer si No Se Necesita el Valor Extraído?**
* Si el valor extraído no se necesita, se puede hacer un **POP a un registro temporal (AX, BX) y descartarlo.**
* **Ejemplo: Extraer y Descartar un Valor de la Pila**
* POP AX ; Extraer valor de la pila, pero no usarlo (se descarta)
* ✔ **Esto es útil cuando se pasan valores innecesarios en la pila y se quiere limpi**

Ar

**📌 Resumen Final: Paso de Parámetros por Pila en Ensamblador x86**

El profesor **ha completado la demostración del paso de parámetros por pila**, mostrando cómo **los valores son extraídos, movidos a registros (BX y BN) y finalmente almacenados en las variables de datos (carácterLeído y carácterTecleado)**.

**📌 ¿Qué se Logró en Esta Lección?**

✔ **Se leyó un carácter (A) con INT 21H, AH=01H, almacenándolo en AL.**  
✔ **Se almacenó el valor en la pila correctamente como WORD.**  
✔ **Se usó POP para extraer el valor de la pila y se movió a BX para verificar el resultado.**  
✔ **Se usó MOV BYTE PTR para almacenar correctamente el BYTE en carácterLeído y carácterTecleado.**  
✔ **Se comprobó que ambos valores (carácterLeído y carácterTecleado) contenían 61h (97 en decimal), confirmando que el método funciona.**

**📌 Claves del Paso de Parámetros por Pila**

**Los valores deben pasarse a la pila con PUSH en orden inverso antes de CALL.**  
**En el procedimiento FAR, BP se usa para acceder a los valores sin alterar SP.**  
**POP solo extrae WORDS (16 bits), por lo que los BYTE deben manejarse con MOV BYTE PTR.**  
**Al salir del procedimiento, los valores son extraídos con POP y almacenados en variables del segmento de datos.**  
**Se verificó que BX y BN tenían el valor correcto, confirmando que los datos fueron transferidos correctamente.**