UNNOBA

ARQUITECTURA II

SUBRUTINAS

EN ASSEMBLER

Mariano Ruggiero

v 1.3

Programación modular

Al desarrollar sistemas relativamente grandes, una de las técnicas más utilizadas es la de dividir el mismo en distintos módulos lo más independientes posibles entre sí, que trabajando en conjunto logran proveer la funcionalidad total requerida por el sistema. La idea es que, subdividiendo el sistema en partes más pequeñas, cada parte es más sencilla de desarrollar que el sistema en su totalidad.

Pensemos en algún programa que usemos habitualmente, por ejemplo Microsoft Word. Word tiene alrededor de 2.000.000 de líneas de código (sí, 2 millones). Imagínense si tuvieran que encontrar un error en el corrector ortográfico y el código del corrector sólo fuera un conjunto de líneas en algún lugar dentro de las 2M de líneas. Decir que sería casi imposible es poco. Este es el tipo de problemas que la programación modular resuelve: aislamos el código relativo al corrector ortográfico en un módulo independiente. Ahora, si hay un error, sabemos exactamente en qué módulo buscar.

Este ejemplo simplificado da una idea de por qué modularizar es considerado una buena práctica: se aíslan los errores, se pueden probar los módulos independientemente, se puede programar en paralelo, los módulos se pueden reutilizar en varios sistemas...

Subrutinas

La unidad de descomposición modular básica se conoce como subrutina. Una subrutina (también conocida como función, método, procedimiento o subprograma, dependiendo del lenguaje de programación) es una porción de código dentro de un programa más grande, que realiza una tarea específica y es relativamente independiente del resto del código.

Un ejemplo: Supongamos que tenemos un programa que administra las notas de alumnos en un colegio secundario. En el programa se cargan las notas de todas las materias, mes a mes, y el programa puede calcular el promedio del alumno en cada materia, en cada trimestre y el promedio general de todo el curso. Como vemos, este programa deberá calcular gran cantidad de promedios. Pero el algoritmo para calcular un promedio es siempre el mismo. Tenemos entonces un candidato claro para una subrutina. Creamos una subrutina que calcule un promedio y luego la podemos utilizar cuantas veces queramos. Si detectáramos que existe un problema en el cálculo del promedio, sabemos que debemos buscarlo dentro de esa subrutina.

Para que el programa principal de nuestro sistema de alumnos sepa cómo interactuar con la subrutina `Promedio` lo que necesita es conocer su **interfaz**. La interfaz de una subrutina indica de qué forma la subrutina recibe una entrada de datos y de qué forma produce una salida. Por ejemplo, la subrutina `Promedio`, deberá recibir una colección de números y deberá retornar el promedio (otro número).

Una subrutina en un lenguaje de alto nivel que calcule un promedio podría ser como sigue. Por simplicidad, vamos a hacer una subrutina que calcula el promedio de 2 números (en lugar de una más general, que sería lo lógico, que calcule el promedio de n números)

```
Function Promedio(a : Real, b : Real) : Real
 Return (a + b) / 2;
End Function
```

Básicamente, esta definición nos está diciendo que la función `Promedio` recibe dos valores de tipo real y devuelve otro valor real. El comando `Return` hace que se realice el cálculo y se devuelva el resultado. Esto es similar a la definición de una función en matemática (de ahí la palabra clave `Function`).

En cualquier lugar del código del programa, podríamos calcular un promedio entre dos números con una instrucción similar a esta:

```
Var Resultado : Real;
Resultado = Promedio(8, 3);
```

Al ejecutar este código, la variable Resultado terminará teniendo el valor 5.5 (el promedio entre 8 y 3).

Fíjense que este código para poder utilizar la subrutina sólo debe conocer su interfaz, es decir: debe saber cómo se llama la subrutina, qué tipo de valores darle como entrada (son los que van entre paréntesis) y qué tipo de valor esperar como salida (lo que queda asignado en `Resultado` en el ejemplo).

La interfaz de la subrutina está formada por:

- El **nombre** de la subrutina.
- Los **parámetros**: los valores de entrada sobre los que la subrutina opera (en el caso anterior `a` y `b`, dos números reales). Una subrutina puede recibir ninguno, uno o más parámetros.
- El **valor de retorno**: es el valor que la subrutina devuelve, como en el ejemplo anterior. Hay subrutinas que no devuelven un valor, sino que realizan una tarea (por ejemplo: una subrutina que muestra un mensaje en pantalla).
- Una descripción de qué hace la subrutina. En una subrutina tan simple como la anterior, la funcionalidad de la subrutina puede inferirse del nombre, pero subrutinas más complejas pueden requerir una aclaración especial

Es claro que con sólo conocer la interfaz basta para poder usar la subrutina. En el ejemplo anterior, la interfaz sería: "esta subrutina recibe dos parámetros a y b de tipo real y retorna el promedio entre ambos". El programa no necesita saber cómo realiza el cálculo. Por eso se dice que las subrutinas son como *cajas negras*. Es decir, al utilizarlas, no necesitamos saber qué es lo que ocurre en el código interno de la subrutina, sólo nos interesa qué valores pasarle y qué nos devuelve. Esto tiene la ventaja de que, una vez establecida la interfaz, el programa principal y la subrutina pueden ser desarrollados independientemente, en paralelo, e incluso por personas distintas. (En un ejemplo tan simple como el cálculo de un promedio esto puede parecer inútil, pero volvamos a un ejemplo del "mundo real": Sí es útil que las subrutinas que se encargan de la corrección ortográfica en Word puedan desarrollarse en forma paralela e independientemente de aquellas que permiten dar formato al texto. Pensemos que ambas funcionalidades son totalmente independientes y bastante complejas en sí mismas).

Subrutinas en Assembler

¿Cómo hacen los programas que utilizan subrutinas para trabajar de la forma en que lo hacen? El programa principal llama a una subrutina y queda dormido hasta que la subrutina termine. ¿Cómo se logra esto?

Repasemos cómo trabaja el procesador para ejecutar un programa:

1) Busca en el segmento de código (CS) la instrucción ubicada en la dirección indicada por el registro IP
2) Se decodifica y ejecuta la instrucción
3) IP se incrementa para encontrar la siguiente instrucción

Y esto se repite continuamente...

¿Qué pasaría si en el paso 2 la instrucción alterara el contenido del registro IP? Al incrementar IP para buscar la próxima instrucción, el procesador "saltaría" a otra área de código.

Esto es lo que pasa siempre que utilizamos una instrucción de salto (JMP, JNE, JZ, JGE, etc.). Por ejemplo, JNE se fija si la comparación anterior dio "distinto" y, si es así, cambia el contenido del IP para que el procesador salte a otra línea.

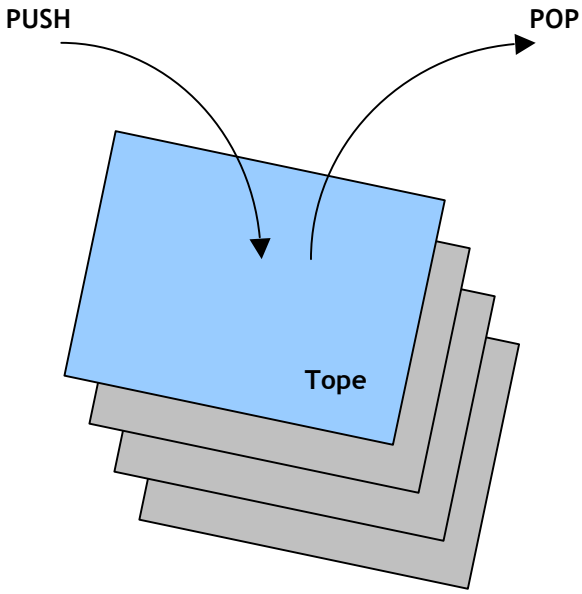
Entonces, podríamos pensar que una subrutina se implementa simplemente como un salto... Pero hay un problema: *el salto sabe ir pero no volver*. Es decir, con el salto voy a donde comienza la subrutina, pero después, desde la subrutina, ¿cómo sé a qué punto del programa principal volver? (Recuerden que una subrutina puede ser llamada muchas veces en distintos puntos del programa, incluso desde otras subrutinas).

Para resolver este problema, el procesador cuenta con dos instrucciones especiales **CALL** (llamada a subrutina) y **RET** (retorno de subrutina). Ambas instrucciones hacen uso de un área especial de memoria llamada **pila** (o **stack**, por su nombre en inglés).

Pilas

Una **pila** es una estructura de datos de tipo **LIFO** (*Last In First Out*, el último en entrar es el primero en salir) que permite almacenar y recuperar datos usando dos operaciones: **PUSH** (apilar) y **POP** (desapilar). Esta estructura de datos funciona como si fuera una pila de platos: cuando apilo un plato sólo puedo hacerlo en el **tope** de la pila y para quitar un plato sólo puedo hacerlo desde el tope de la pila. Una pila bien definida no permite acceder a otro elemento que no sea el que se encuentre en el tope; sólo una vez

que haya quitado el elemento del tope, puedo sacar el que estaba inmediatamente debajo (y ningún otro más hasta que no quite éste, y así sucesivamente).

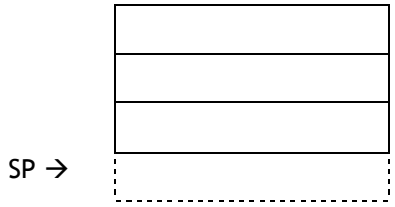
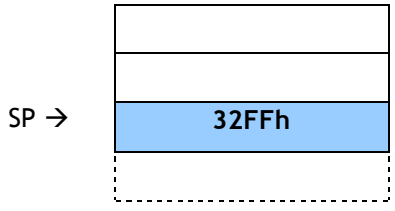


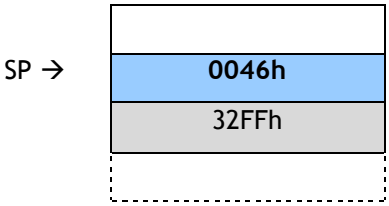
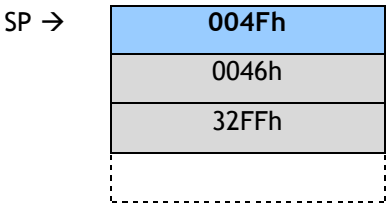
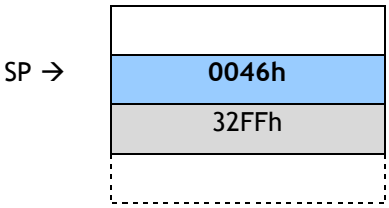
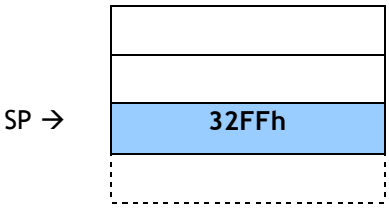
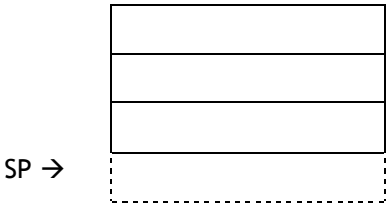
Los sistemas con arquitectura Intel, como los que hemos visto, implementan una estructura de pila que el programa puede utilizar como un área para almacenar datos temporalmente. Para esto se tiene:

- Un **segmento de pila** (otra área similar a los segmentos de datos y de código). La dirección de este segmento está indicada por el registro **SS**.
- Un **puntero de pila**, que es el registro **SP**. Este registro indica la dirección del dato que se encuentra en el tope de la pila.
- Las instrucciones **PUSH** y **POP**, que funcionan como vimos recién.

Como restricción, todos los elementos que apilemos deben ser de **16 bits**.

La pila funciona de la siguiente manera:

CÓDIGO	RESULTADO DE LA OPERACIÓN
	 La pila está vacía. SP apunta a una celda de memoria *fuera* de la pila.
PUSH 32FFh	 SP se **decrementa en 2** y se escribe el valor en la dirección SS:SP (el tope de la pila).

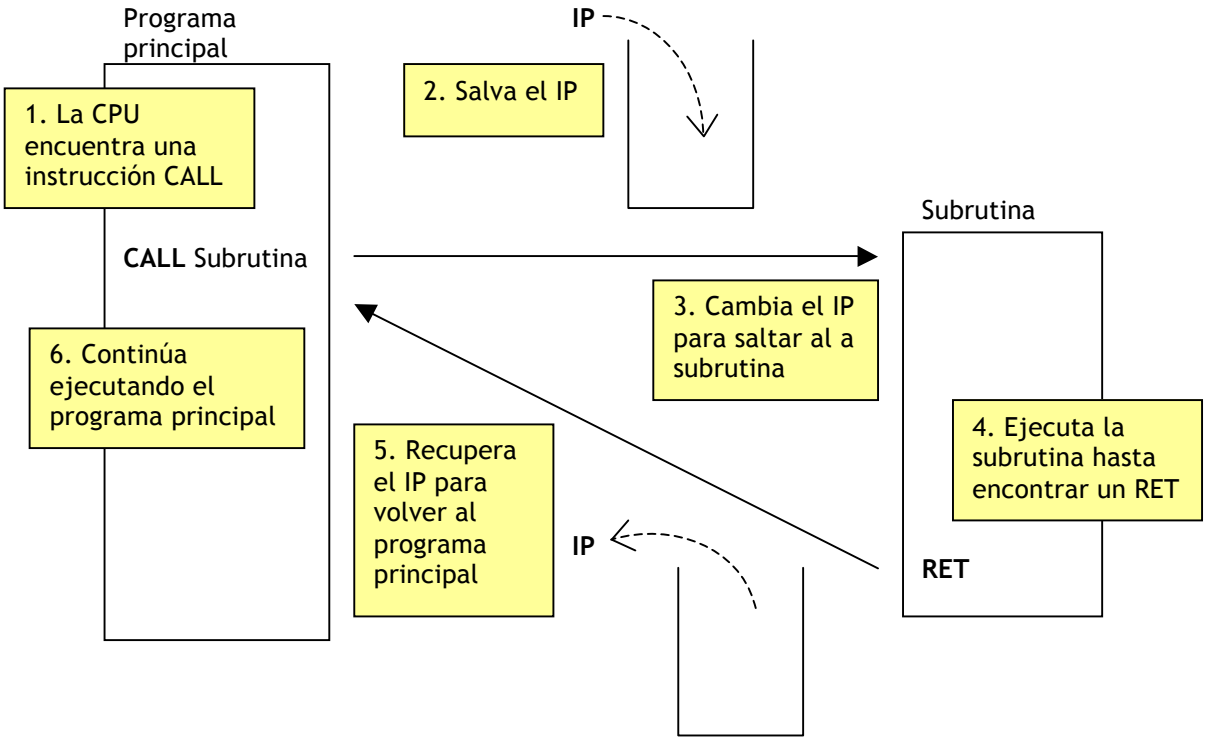
MOV AX, 46h PUSH AX	 SP se decrementa en 2 y se escribe el contenido de AX en SS:SP.
ADD AX, 9h PUSH AX	 Se apila AX + 9h.
POP CX	 SP copia el tope de la pila (el contenido de la dirección SS:SP) sobre el registro CX (CX = 004Fh) y SP se **incrementa en 2**.
POP Numero (Numero es una variable declarada en el segmento de datos)	 Se copia el tope de la pila sobre la variable Numero (Numero = 0046h) y SP se incrementa en 2.
POP AX	 Se desapila sobre AX (AX = 32FFh).

Subrutinas y la pila

Pero, ¿qué tiene que ver la pila con las subrutinas?

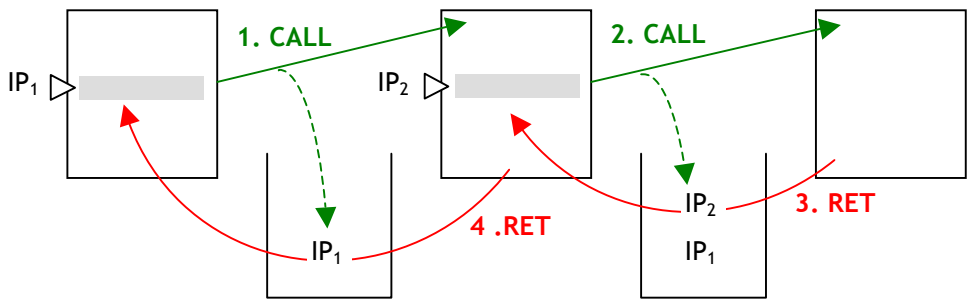
Como dijimos, si estamos ejecutando un código y saltamos a otro punto (al comienzo de la subrutina) tenemos el problema de que no sabemos a dónde volver luego de ejecutar la subrutina. Para poder volver necesitaríamos restaurar el valor que el registro IP tenía *antes* de saltar a la subrutina.

Las instrucciones CALL y RET resuelven este problema haciendo uso de la pila. Cuando se realiza una llamada a una subrutina con CALL, esta instrucción automáticamente guarda el contenido del registro IP en la pila y salta al comienzo de la subrutina. En algún punto de la subrutina el procesador se encontrará con la instrucción RET que, al ejecutarla, desapilará el valor del IP guardado en la pila de modo que cuando el procesador avance a la siguiente instrucción (incremente IP), estará de vuelta en la instrucción siguiente a donde se hizo la llamada.

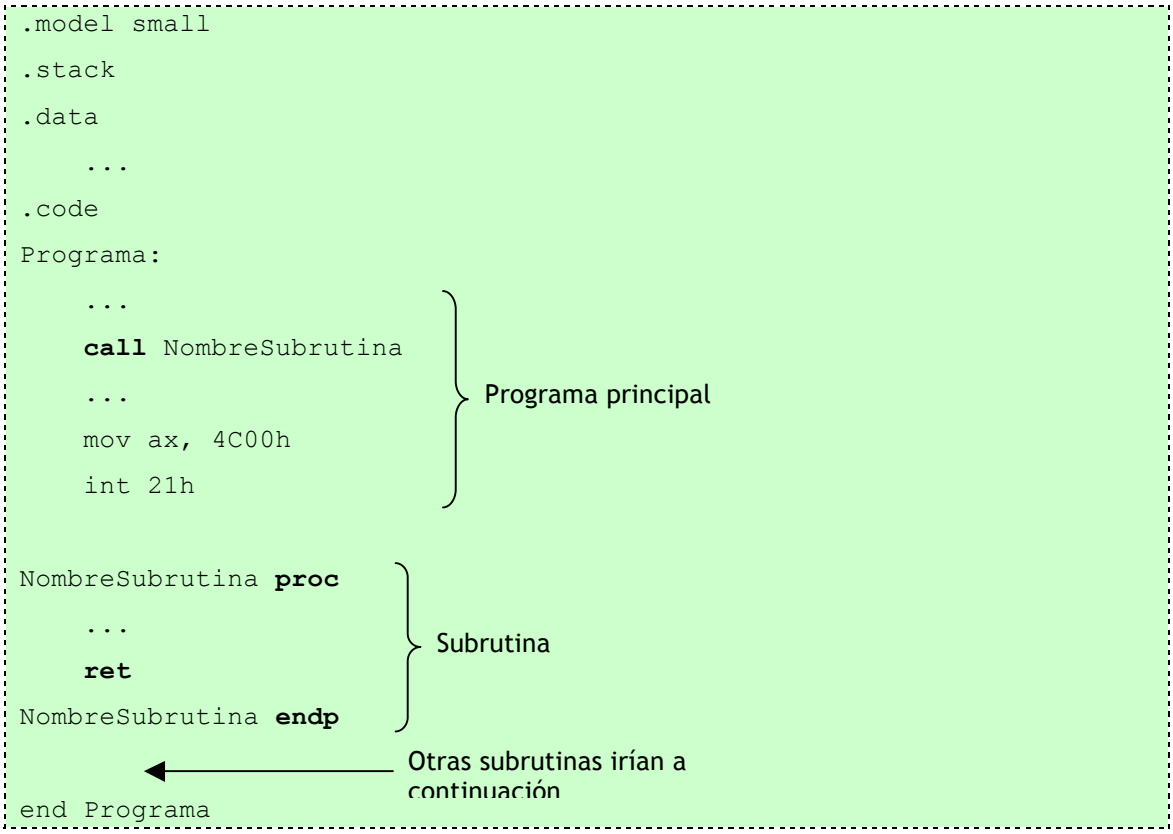


¡Ojo! Hay que asegurarse de que al ejecutar la instrucción RET, el dato que se encuentra en el tope de la pila sea la dirección IP de retorno, de lo contrario se desapilará un valor incorrecto y el procesador irá a ejecutar instrucciones a otra área de memoria (lo que terminará, probablemente, con que el programa se cuelgue). Entonces, siempre que una subrutina apile datos (PUSH) debe recordar desapilarlos (POP).

Si lo pensamos, nos daremos cuenta de que este mecanismo nos permite con igual facilidad ir del programa principal a una subrutina como de una subrutina a otra. De hecho, se pueden "anidar" las llamadas a subrutinas, es decir tener un programa que llama a una subrutina, que llama a otra subrutina, que llama a otra, etc... Lo que sucederá en ese caso es que se apilarán sucesivamente varios valores de IP. Por el mecanismo de la pila, el último IP en entrar en la pila será el primero en desapilarse, de modo que esto nos permitirá volver de una subrutina al código que nos llamó sin problemas:



Formato de un programa con subrutinas



Salvar el estado

La pila puede ser utilizada dentro de una subrutina para salvar el estado de los registros que vamos a modificar. Por ejemplo, supongamos que en la subrutina vamos a utilizar CX para un LOOP, ¿qué pasa si esta subrutina fue llamada desde dentro de un LOOP en el programa principal? Tanto la subrutina como el programa están alterando el valor de CX. Cuando la subrutina termine el LOOP, CX va a valer 0. Cuando la subrutina vuelva, el LOOP del programa principal va a ver un 0 en CX y va a terminar, pero sólo realizó una iteración... Esto sucedió porque perdió el valor de CX que iba contando la cantidad de vueltas que faltaban.

Esto se resuelve fácilmente utilizando el mismo mecanismo que usan CALL y RET para salvar y recuperar el IP: Al principio de la subrutina, guardamos en la pila el valor de los registros que vamos a escribir y, antes del RET, desapilamos todos los registros para recuperar su valor original. De esta forma, el valor que tuvieran en el programa principal se conserva y el programa no se entera de que hubo una modificación.

Atención: al desapilar los registros salvados, hay que hacerlo **en orden inverso** a como se los apiló. Recordar que en la pila lo último que se apiló es lo primero que se desapila.

Pasaje de parámetros

En Assembler podemos utilizar dos mecanismos para comunicarnos con una subrutina:

- los registros
- la pila

Pasaje de parámetros en registros

La idea es simple: el programa principal escribe los datos a pasar a la subrutina en uno o varios registros. Luego, la subrutina lee esos mismos registros para obtener los datos. A su vez, la subrutina puede devolver datos al programa principal utilizando el mismo mecanismo.

Una vez más, lo más importante en este caso es que el programa principal y la subrutina respeten una misma interfaz. Veamos un ejemplo: la interfaz de una subrutina que devuelva el mayor de dos números podría ser: recibe los números en AX y BX y devuelve el mayor entre ambos en AX. Entonces, el invocador deberá asegurarse de copiar en AX y BX los valores *antes* de llamar a la subrutina. Cuando la subrutina arranca supone que AX y BX tienen cargados los números.

```
.model small
.stack
.data
    Numero1 dw 100Ah    ; Los números a comparar
    Numero2 dw F53Ch    ;
    ElMayor dw ?        ; Para almacenar el mayor
.code
Programa:
    mov ax, 4C00h       ; Inicializar DS
    mov ds, ax          ;

    mov ax, Numero1     ; Copiar los números en AX y BX, como
    mov bx, Numero2     ; pide la subrutina

    call Mayor          ; Invocar a la subrutina

    ; Por la interfaz, sabemos que en este punto AX debe contener
    ; el mayor de los dos números

    mov ElMayor, ax     ; Guardar el mayor

    mov ax, 4C00h       ; Terminar
    int 21h             ;

; Subrutina que halla el mayor entre dos números contenidos en AX y BX
; Devuelve en AX el mayor de ambos
Mayor proc
    cmp ax, bx          ; Comparar AX con BX
    jg MayorFin         ; Si AX y es el mayor, terminar
    mov ax, bx          ; Como BX es el mayor, lo copiamos en AX
MayorFin:
    ret                 ; Retornar
Mayor endp

end Programa
```

Pasaje de parámetros en la pila

Para pasar parámetros en la pila, se apilan los datos (PUSH) antes de invocar a la subrutina. La subrutina leerá los datos desde la pila y operará sobre ellos. Se puede utilizar la pila tanto para enviar datos a una subrutina como para recibir datos desde ésta o bien sólo recibir por la pila y devolver por registros.

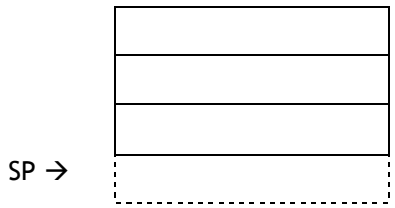
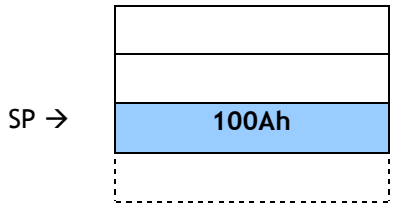
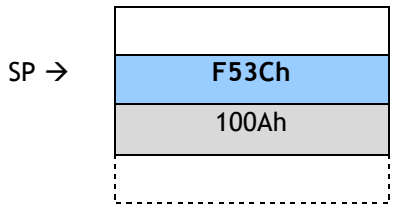
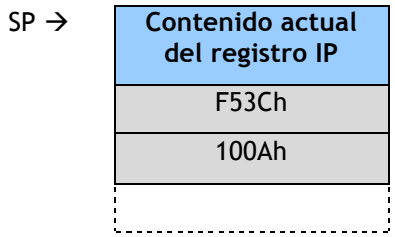
Si reescribimos la subrutina anterior para que ahora utilice la pila para recibir los datos, la nueva interfaz podría ser: la subrutina Mayor recibe dos números en la pila y devuelve en AX el mayor de ambos.

El programa principal sigue siendo el mismo, excepto en la llamada a la subrutina. Ahora tenemos:

```
    push Numero1        ; Apilar los dos números, como
    push Numero2        ; pide la subrutina

    call Mayor          ; Invocar a la subrutina
```

Ahora, veamos cómo va quedando la pila al ejecutar cada una de estas instrucciones:

CÓDIGO	RESULTADO DE LA OPERACIÓN
	 Supongamos que la pila está inicialmente vacía.
PUSH Numero1	
PUSH Numero2	
CALL Mayor	 Cambia el IP. Ahora, **IP = Dirección de la subrutina Mayor**.

Nos encontramos en un problema: al hacer CALL, se apila la dirección de retorno (IP actual), de modo que cuando vayamos a la subrutina **no podemos hacer POP para recuperar los parámetros**.

Para resolver este problema, lo que tendremos que hacer es quebrantar las reglas de la estructura de datos pila: vamos a acceder a los elementos que no están en el tope de la pila y sin utilizar POP. Para eso, usaremos direccionamiento relativo al registro **BP**, que es el único que nos sirve para expresar direcciones en la pila. Recordemos que entre corchetes (dirección de memoria) sólo podemos utilizar los registros BX, BP, SI y DI. BX, SI y DI se utilizan sobre el segmento de datos (DS) y BP sobre el segmento de pila (SS).

De manera que toda subrutina que utilice la pila deberá comenzar con estas dos líneas:

```
push bp
mov bp, sp
```

Como vamos a modificar el registro BP, primero lo guardamos en la pila. Esto es muy importante ya que todas las subrutinas que reciben parámetros por la pila usan BP; tenemos que asegurarnos de que al salir BP recupere el valor que tenía.

Al ejecutar esas instrucciones, nos queda:

CÓDIGO	RESULTADO DE LA OPERACIÓN
PUSH BP	SP → **Contenido actual del registro BP** Contenido anterior del registro IP F53Ch 100Ah El valor de BP se guarda en la pila. Con esto, de alguna manera estamos empeorando el problema... pero es necesario preservar BP.
MOV BP, SP	SP → BP → **Contenido actual del registro BP** Contenido anterior del registro IP F53Ch 100Ah Ahora igualamos BP a SP, de manera que ambos apuntan al tope de la pila.

Ahora, para acceder a los datos podemos utilizar expresiones como **SS:[BP+4]**, por ejemplo, para acceder al segundo parámetro (F53Ch). Recordemos que cada elemento de la pila ocupa **2 bytes**, o sea que si tenemos que saltear 2 elementos para llegar al dato que necesitamos debemos sumar 4. SS:[BP+6] nos permite acceder al primer parámetro (100Ah).

Veamos como queda ahora nuestro código:

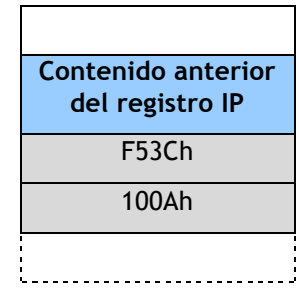
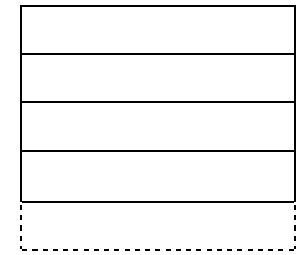
```
; Subrutina que halla el mayor entre dos números contenidos en la pila
; Devuelve en AX el mayor de ambos
Mayor proc
    push bp             ; Salvar BP
    mov bp, sp          ; BP = SP

    mov ax, ss:[bp + 4] ; Obtener uno de los parámetros
    cmp ax, ss:[bp + 6] ; Comparar como el otro
    jg MayorFin         ; Si es mayor, terminar

    mov ax, ss:[bp + 6] ; Como el mayor era el otro, lo copiamos en AX

MayorFin:
    pop bp              ; Restaurar BP
    ret 4               ; Retornar y desapilar los parámetros
Mayor endp
```

Antes de retornar debemos desapilar BP y todos los parámetros. Esto último se logra poniendo un número a continuación de la instrucción RET que indica **el número de bytes a desapilar además del IP**. En el ejemplo estamos indicando que desapile 4 bytes (los dos parámetros). Esto nos evita tener que desapilar a mano (usando PUSH) los parámetros al volver de la subrutina.

CÓDIGO	RESULTADO DE LA OPERACIÓN
POP BP	 BP toma el valor origina que estaba guardado en la pila (ver página anterior).
RET 4	 IP recuperar el valor anterior para poder volver al punto de llamada. Y se desapilan otros 4 bytes, con lo que la pila queda limpia.

Subrutinas recursivas

Un **algoritmo recursivo** es un algoritmo que se define en términos de sí mismo. Una subrutina recursiva es aquella que incluye en su código una llamada a la misma subrutina.

Un clásico ejemplo humorístico de un texto recursivo que parodia las definiciones de los diccionarios es:

Recursividad: *Véase "Recursividad".*

Sin embargo, este texto es un ejemplo de una definición recursiva errónea. El error está en que no existe una condición de finalización, lo que lleva a una **recursividad infinita**. Para entender la recursividad, hay que darse cuenta de que la clave está en la **condición de finalización**. Una versión más correcta de la definición anterior podría ser:

Recursividad: *Si aún no se entiende, véase "Recursividad".*

Veamos un ejemplo de un procedimiento iterativo (no recursivo) para calcular la suma de los números naturales entre 1 y n (obviamente hay formas muchísimo más eficientes de hacer esto, pero es sólo un ejemplo):

```
Function Suma1aN(n : Integer) : Integer
    Var s : Integer;
    s = 0;
    For i = 1 to n
         s = s + i;
    End For
    Return s;
End Function
```

O sea, lo que codificamos fue exactamente la definición básica de la sumatoria de 1 a n:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n$$

Pero existe una forma recursiva de definir la misma sumatoria:

- $\sum_{i=1}^{n} i = n + \sum_{i=1}^{n-1} i$
- $\sum_{i=1}^{1} i = 1$

El primero es el paso recursivo: "la sumatoria de *i* entre 1 y *n* es igual a *n* más la sumatoria de *i* entre 1 y *n*-1". Fíjense que usamos la *sumatoria* para definir a la *sumatoria*. Si nos quedáramos ahí, tendríamos una recursividad infinita, pero el segundo punto nos da el **caso base** (la condición de fin): "la suma *i* entre 1 y 1 es 1". Esta vez no hay recursividad. Veamos, usando la definición recursiva:

$$\sum_{i=1}^{3} i = 3 + \sum_{i=1}^{2} i = 3 + 2 + \sum_{i=1}^{1} i = 3 + 2 + 1$$

Aunque en este ejemplo la recursividad parezca sólo una complicación, hay muchos algoritmos que pueden expresarse naturalmente y de manera más sencilla si se utiliza recursividad. (Ejemplos de esto son recorridos de estructuras de datos como árboles o grafos, búsquedas y algoritmos de programación dinámica... todos temas interesantes, pero que exceden nuestro objetivo en este momento).

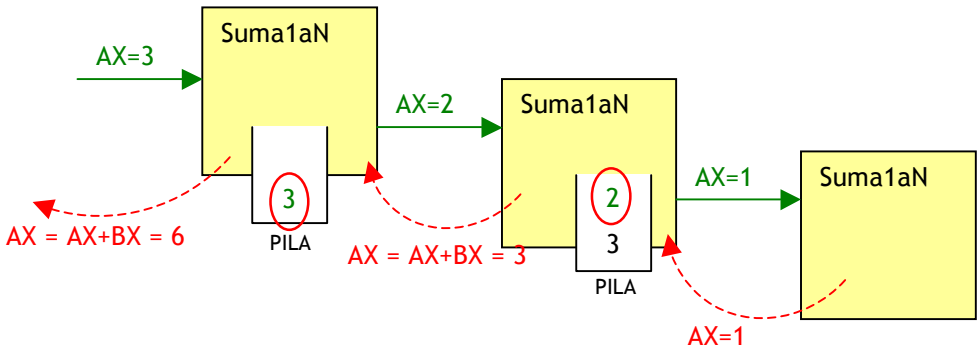
Veamos, entonces, cómo se implementa la misma función Suma1aN, pero esta vez de forma recursiva, usando la definición de la sumatoria que acabamos de ver:

```
Function Suma1aN(n : Integer) : Integer
    If n = 1 Then
        Return 1;
    Else
        Return n + Suma1aN(n - 1);
    End If
End Function
```

Subrutinas recursivas en Assembler

Veamos la implementación de la subrutina anterior utilizando Assembler:

```
; Subrutina que calcula la suma de los naturales entre 1 y N.
; N se recibe en AX. Devuelve la suma en AX.
Suma1aN proc
    cmp ax, 1       ; Si es AX = 1 (caso base),
    je Fin          ; retornar
    ; AX > 1
    push ax         ; Guardar N en la pila
    dec ax          ; Obtener AX = N-1
    call Suma1aN    ; Invocar a la subrutina pasando AX = N-1
    ; Al volver de la subrutina, AX tiene la suma de 1 a N-1
    pop bx          ; Recuperar N de la pila
    add ax, bx      ; Sumarle N
Fin:
    ret
Suma1aN endp
```



El mecanismo a través del cual se logran llamadas recursivas en Assembler es exactamente el mismo que en el de llamadas estándar. Cada vez que hay una llamada (a la misma subrutina en este caso) se apila el IP y al retornar se desapila. En el caso de subrutinas recursivas, hay que tener más presente que nunca que cada ejecución de la subrutina utiliza los *mismos* registros. Por ejemplo, en la subrutina anterior, se salvó el valor de AX en la pila antes de volver a invocar a la subrutina de modo de no perder el valor original.