# Aprendiendo Sistemas Operativos: Programación de Procesadores de Arquitectura IA-32

Este documento forma parte de la serie “Aprendiendo Sistemas Operativos” y sirve como una introducción a los aspectos básicos de la programación de procesadores de arquitectura IA-32. La serie “Aprendiendo Sistemas Operativos” puede ser usada en un entorno académico previa autorización del autor.



Autor:
Erwin Meza Vega
emezav@gmail.com

# 2 Lenguaje Ensamblador para IA-32

Los programas para procesadores de arquitectura IA-32 se pueden desarrollar en lenguaje ensamblador, o en lenguajes de alto nivel como C, C++, C#, java, entre otros. En última instancia, las instrucciones en lenguaje de alto nivel son transformadas en lenguaje ensamblador y luego en lenguaje de máquina (binario).

## 2.1 SINTAXIS AT&T E INTEL

Si bien la sintaxis Intel es la más utilizada en la documentación y los ejemplos, es conveniente conocer también los aspectos básicos de la sintaxis AT&T, ya que es la sintaxis por defecto del ensamblador de GNU.

Las sintaxis AT&T e Intel difieren en varios aspectos, entre ellos:

- Nombre de los registros. En sintaxis AT&T los registros (de 16 bits) se denominan: %ax, %bx, %cx, %dx, %si, %di, además de los registros %sp, %bp, y los registros de segmento %cs, %ds, %es, %fs y %gs. Por su parte, en la sintaxis Intel (con la directiva '.intel_syntax noprefix') no es necesario anteponer '%' al nombre de los registros.
- Orden de los operandos: En la sintaxis AT&T, el orden de los operandos en las instrucciones es el siguiente:
  ```
  instrucción fuente, destino
  ```
  Mientras que en la sintaxis Intel, el orden de los operandos es:
  ```
  instrucción destino, fuente
  ```
- Operandos inmediatos. Los operandos inmediatos son aquellos valores que se especifican de forma explícita dentro de la instrucción. En la sintaxis AT&T, los operandos inmediatos se preceden con el signo '$' Por ejemplo, la instrucción en sintaxis AT&T:
  ```
  movb $0x05, %ah
  ```
  almacena en el registro %ah (un byte) el valor 0x05 (0xXX) significa el numero XX en formato hexadecimal).
  En sintaxis Intel:
  ```
  mov ah, 0x05
  ```
- Longitud de operaciones de memoria: En la sintaxis AT&T, cada instrucción debe llevar un sufijo que indica la longitud de la operación de memoria. En sintaxis Intel sólo se debe especificar el sufijo en aquellas instrucciones cuyos operandos son direcciones de memoria. Por ejemplo, para mover un byte de una dirección de memoria 'addr', al registro AL, el código en sintaxis AT&T sería el siguiente:
  ```
  movb (addr), %al
  ```
  Observe el sufijo 'b' de la instrucción 'mov'.
  De otro lado, la sintaxis Intel sería la siguiente:
  ```
  mov al, BYTE PTR [addr]
  ```

  También es válida la siguiente sintaxis:

  ```
  movw al, [addr]
  ```

  Los sufijos válidos son: b para byte, w para word, l para long (32 bits), y q para quadword

(64 bits). Por otra parte, los prefijos de las direcciones de memoria en sintaxis Intel son: BYTE PTR, WORD PTR, DWORD PTR y QWORD PTR.

## 2.2 INSTRUCCIONES BÁSICAS DE ENSAMBLADOR PARA IA-32

El uso de las instrucciones de ensamblador depende en gran medida del tipo de programa que se desea desarrollar. Si se planea desarrollar un programa para ser ejecutado en un entorno de 16 bits (Modo Real), solo se puede tener acceso a los 16 bits menos significativos de los registros, y a un conjunto limitado de instrucciones. Por el contrario, si se planea desarrollar un programa para ser ejecutado en un entorno de 32 (o 64) bits, se puede tener acceso a la totalidad de bits de los registros y a un conjunto mayor de instrucciones.

La mayoría de instrucciones presentadas a continuación realizan operaciones a nivel de bytes (por ello tienen el sufijo 'b'), y para el acceso a memoria se utiliza `BYTE PTR` para indicar apuntadores a bytes. También es posible operar a nivel de words (2 bytes, 16 bits), usando el sufijo 'w' y el modificador `WORD PTR` en vez de `BYTE PTR` para apuntadores de tipo word en memoria en sintaxis Intel.

En modo de 32 bits, es posible operar sobre doublewords (4 bytes, 32 bits) usando el sufijo 'w', y para instrucciones de acceso a memoria se utiliza en sintaxis Intel se debe especificar `DWORD PTR` en vez de `BYTE PTR`, o usar el sufijo 'l' en la instrucción mov.

En operaciones de 64 bits se debe usar el sufijo 'q' (quadword, 8 bytes, 64 bits) para la mayoría de instrucciones y `QWORD PTR` para el acceso a memoria en sintaxis Intel.

Se debe recordar que el uso de los registros depende del modo de operación del procesador. Así, en modo real se puede tener acceso a los primeros de 8 bits y 16 bits de los registros de propósito general, en modo protegido se puede acceder a los 32 bits de los registros y en modo de 64 bits se puede tener acceso a los 64 bits de los registros.

## 2.3 MOVIMIENTO DE DATOS

**INSTRUCCIÓN mov (move)**: permite mover (copiar) datos entre dos registros, de un valor inmediato a un registro o de un valor inmediato a una posición de memoria.

**En sintaxis AT&T:**

**De inmediato a registros**:

```
movb $0x10, %ah /* Mover el byte (valor inmediato) 0x10 a %ah */
movb $0x20, %al /* Mover el byte 0x20 a %al */
movw $0x1020, %ax /* Mueve un word (2 bytes) a %ax */
movl $0x00001020, %eax /* Mueve un doubleword (4 bytes) a %eax */
movq $0x0000000000001020, %rax /* Mueve un quadword (8 bytes) a %rax */
```

**De registros a registros**:

```
movb %al, %bl /* Mover un byte de %al a %bl */
movb %ah, %bh  /* Mover un byte de %ah a %bh */
movw %ax, %bx /* Equivalente a las dos instrucciones anteriores */
movl %eax, %ebx /* 32 bits */
movw %rax, %rbx /* 64 bits */
```

**De registros a memoria**:

```
movb %al, (%si) /* Mover el byte almacenado en %al a la posición
                   de memoria  apuntada por %si */
movb %al, 4(%si) /* Mover el byte almacenado en %al a la posición
                   de memoria apuntada por (%si + 4)*/
movb %al, -2(%si) /* Mover el byte almacenado en %al a la posición
                   de memoria apuntada por (%si - 2) */
movw %ax, (%si) /* Mover el word almacenado en %ax a la posición
                    de memoria  apuntada por %si */
movl %eax, (%esi) /* 32 bits */
movq %rax, (%rsi) /* 64 bits */
```

**De memoria a registros**:

```
movb (%si), %al  /* Mover el byte de la posición de memoria
                    apuntada por (%si) a %al */
movb 4($si), %al /* Mover el byte de la posición de memoria
                   apuntada por (%si + 4) a %al */
movb -2($si), %al /* Mover el byte de la posición de memoria
                     apuntada por (%si - 2) a %al */
```

**En sintaxis Intel:**

(Recuerde que el orden de los operandos es `instrucción` ***destino***, ***fuente***)

**De inmediato a registros**:

```
mov ah, 0x10 /* Mover el byte (valor inmediato) 0x10 a ah */
mov al, 0x20 /* Mover el byte 0x20 a al */
mov ax, 0x1020 /* Equivalente a las dos operaciones anteriores */
mov eax, 0x00001020 /* 32 bits */
mov rax, 0x0000000000001020 /* 64 bits */
```

**De registros a registros**:

```
mov bl, al /* Mover un byte de al a bl */
mov bh, ah  /* Mover un byte de ah a bh */
mov bx, ax /* Equivalente a las dos instrucciones anteriores */
mov ebx, eax /* 32 bits */
mov rbx, rax /* 64 bits */
```

**De registros a memoria**:

```
mov BYTE PTR [ si ], al /* Mover el byte almacenado en al a la
                           posicion de memoria apuntada por si */
movb [ si ], al      /* Equivalente a la instrucción anterior.
                        Observe el sufijo 'w' en la instrucción. */

mov BYTE PTR [ si + 4 ], al /* Mover el byte almacenado en %al a
                               la posición de memoria apuntada por (si + 4)*/

mov BYTE PTR [ si - 2 ], al /* Mover el byte almacenado en %al a
                               la posición de memoria apuntada por (si - 2) */

mov WORD PTR [ si ], ax /* Mover el word almacenado en ax a la
```

```
                    posición de memoria apuntada por (si) */

mov DWORD PTR [ esi ], eax /* 32 bits */
mov QWORD PTR [ rsi ], rax /* 64 bits */
```

**De memoria a registros**:

```
mov al, BYTE PTR [ si ]   /* Mover el byte de la posición de
                             memoria apuntada por (si) a %al */
movw al, [ si ]   /* Equivalente a la instrucción anterior */

mov al, BYTE PTR [ si + 4] /* Mover el byte de la posición de
                              memoria apuntada por (si + 4) a al */

mov al, BYTE PTR [ si - 2 ] /* Mover el byte de la posición de
                               memoria apuntada por (si - 2) a al */

mov ax, WORD PTR [ si ]   /* Mover un word */
mov eax, DWORD PTR [ esi ]   /* 32 bits (doubleword)*/
mov rax, QWORD PTR [ rsi ]   /* 64 bits (quadword) */
```

**INSTRUCCIÓN movs (move string)**: Permite copiar datos entre dos posiciones de memoria. Automáticamente incrementa los dos apuntadores, luego de la copia (Ver REPETICIONES y DIRECCION DE INCREMENTO).

**En sintaxis AT&T:**

```
movsb (%si), (%di) /* Copia un byte de la posición de memoria apuntada
por el registro %si a la posición de memoria apuntada por el registro %di
*/
movsw (%si), (%di) /* Copia un word de (%si) a (%di) */
movsl (%esi), (%edi) /* Copia un dobuleword de (%esi) a (%edi) */
movsq (%rsi), (%rdi) /* Copia un quadword de (%esi) a (%edi) */
```

**En sintaxis Intel:**

```
mov BYTE PTR  [ si ], BYTE PTR [ di ] /* Mueve un byte de (si) a (di)*/
movb [ si ], [ di ] /* Equivalente a la instrucción anterior */
mov WORD PTR [ si ], WORD PTR [ di ] /* Mueve un word de (si) a (di) */
mov DWORD PTR [ esi ], DWORD PTR [ esi ] /* 32 bits */
mov QWORD PTR [ rsi ], QWORD PTR [ rdi ] /* 64 bits */
```

**INSTRUCCIÓN lods (load string)**: Permite copiar datos entre una posición de memoria y un registro. Automáticamente incrementa el apuntador a la memoria en el número de bytes de acuerdo con la longitud de operación.

**En sintaxis AT&T:**

```
lodsb /* También es valido lodsb %al, (%si) */
        /* Almacena un byte de la posición de memoria apuntada por
        (%si)en en el registro %al */

lodsw /* Almacena un word de la posición de memoria apuntada por
      (%si) en el registro %ax */
```

```
lodsl /* 32 bits */

lodsq /* 64 bits */
```

**En sintaxis Intel:**

```
lods al, BYTE PTR [ si ]
                      /* Almacena un byte de la posición de memoria
                      apuntada por (si) en  el registro al */
lodsb                 /* Equivalente a la instrucción anterior.
                      La sintaxis abreviada también es válida */

lods ax, WORD PTR [ si ] /* Almacena un word de la posición de
                  memoria apuntada por (si) en el registro ax */
lodsw             /* Equivalente a la instrucción anterior. */

lodsl /* 32 bits */

lodsq /* 64 bits */
```

**INSTRUCCIÓN stos (store string)**: Permite copiar datos entre un registro y una posición de memoria. Incrementa automáticamente el apuntador a la memoria.

**En sintaxis AT&T:**

```
stosb /* También es valido stosb %al, (%di) */
        /* Almacena el valor de %al a la posición de memoria apuntada por
        (%di)  */
stosw /* Almacena el valor de %ax a la posición de memoria
       apuntada por (%di) */

stosl /* 32 bits */

stosq /* 64 bits */
```

**En sintaxis Intel:**

```
stos BYTE PTR [ di ], al
        /* Almacena el valor de al a la posición de memoria
            apuntada por       (di) */

stosb   /* También es válida la instrucción abreviada */

stos WORD PTR [ di ], ax  /* Almacena el valor de ax a la posición
                             de memoria apuntada por  (di) */
stosw       /* Equivalente a la instrucción anterior. */

stosl /* 32 bits */

stosq /* 64 bits */
```

## 2.4 REPETICIONES

Las instrucciones movs, lods y stos pueden hacer uso del prefijo 'rep' (repeat), para repetir la operación incrementando los registros ESI o EDI sea el caso y la longitud de la operación, mientras el valor del registro ECX sea mayor que cero (este valor debe ser establecido antes de invocar la instrucción).

Por ejemplo, las secuencias de instrucciones en sintaxis AT&T (modo real):

```
movw $0x100, %cx
rep stosb /* 16 bits, copiar byte a byte, incrementar %di en 1*/

movw $0x80, %cx
rep stosw /* 16 bits, copiar word a word, incrementar %di en 2 */
```

en sintaxis Intel:

```
mov cx, 0x100
rep stosb BYTE PTR [ di ], al /* 16 bits, copiar byte a byte */

mov cx, 0x80
rep stosw WORD PTR [ di ], ax /* 16 bits, copiar word a word */
```

Copian el valor del registro AL (un byte) o AX (dos bytes) a la posición de memoria apuntada por (DI), y automáticamente incrementa el apuntador DI. Cada vez que se ejecuta la instrucción, el registro CX se decrementa y se compara con cero. Los cuatro ejemplos realizan la misma acción en modo real: permiten copiar 256 bytes de un registro a la memoria. En la primera forma se realizan 256 iteraciones (0x100) para copiar byte a byte, y en la segunda solo se requieren 128 iteraciones (0x80), ya que cada vez se copia un word (dos bytes) cada vez.

**DIRECCION DE INCREMENTO**

Se debe tener en cuenta que las instrucciones lods, stos y movs automáticamente incrementan los registros ESI o EDI según sea el caso y la longitud de la operación (byte, word, doubleword o quadword).

Esto se puede garantizar invocando la instrucción `cld` (clear direction flag) con la cual se realiza el incremento automáticamente. Por el contrario, la instrucción `std` (set direction flag) causa que las instrucciones decrementen automáticamente los registros ESI o EDI según sea el caso.

## 2.5 SALTOS, BIFURCACIONES Y CICLOS

Al ejecutar el programa, el procesador simplemente recupera (fetch) la siguiente instrucción apuntada por el registro EIP, la decodifica (decode) y la ejecuta (execute). Luego continúa con la siguiente instrucción. A esto se le denomina el ciclo 'fetch-decode-execute'. No obstante, en los programas de alto nivel generalmente existen instrucciones que permiten alterar la ejecución de un programa de acuerdo con determinadas condiciones, y repetir una serie de instrucciones dentro de ciclos.

### 2.5.1 SALTOS

Para evitar la linealidad de la ejecución, el procesador dispone de instrucciones que permiten 'saltar' (cambiar el valor de EIP), para poder continuar la ejecución del programa en otro sitio diferente dentro del código.

**Instrucción jmp (jump)**: Salto incondicional

Permite continuar la ejecución de forma incondicional en otra posición de memoria, designada generalmente por una etiqueta en el código fuente ensamblador.

Ejemplo (En sintaxis AT&T e Intel ):

```
label1:
 ...
(instrucciones)
...
jmp label2
(xxxinstruccionesxxx)
...
label2:
...
(instrucciones)
...
```

En este ejemplo, las instrucciones desde `jmp label2` hasta la etiqueta `label2` no son ejecutadas. Esto es especialmente útil si se requiere 'saltar' una región del código en ensamblador.

En los programas jmp generalmente se salta dentro del mismo segmento. No obstante, si el programa lo soporta, es posible saltar a otros segmentos.

***Saltos entre segmentos:***

El formato de la instrucción JMP para salto entre segmentos es la siguiente:

En sintaxis AT&T:

```
ljmp seg, offset
```

En sintaxis Intel:

```
jmp seg:offset
```

En estos formatos de jmp, CS adquiere el valor especificado en 'seg' y EIP adquiere el valor especificado en 'offset'.

**Instrucciones de salto condicional (jz, jnz, je, jne, jle, jge, jc)**: Estas instrucciones generalmente vienen precedidas por instrucciones que realizan manipulación de registros o posiciones de memoria.

La sintaxis de todas estas instrucciones es la misma:

`j… label` , donde … es la condición que se debe cumplir para realizar el salto.

Algunos ejemplos del uso de las instrucciones de salto condicional son:

```
jz label
jnz label
je label
```

etc.

Con estas instrucciones se realiza un salto a la etiqueta `label` en el código ensamblador, dependiendo si se cumple o no la condición de acuerdo con la instrucción. Si la condición no se cumple, el procesador continúa ejecutando la siguiente instrucción que se encuentre después de la instrucción de salto condicional.

Las instrucciones y condiciones más importantes son:

`jz / je label` (*jump if zero / jump if equal)*: Saltar a la etiqueta `label` si el bit ZERO del registro FLAGS se encuentra en 1, o si en una comparación inmediatamente anterior los operandos a comparar son iguales.

`jnz / jne label` (jump if not zero / jump if not equal): Contrario a jz. Saltar a la etiqueta `label` si el bit ZERO del registro FLAGS se encuentra en 0, o si en una comparación inmediatamente anterior los operandos a comparar no son iguales.

`jc label`: Saltar a la etiqueta `label` si el bit CARRY del registro FLAGS se encuentra en 1. Este bit se activa luego de que se cumplen determinadas condiciones al ejecutar otras instrucciones, tales como sumas con números enteros. Igualmente algunos servicios de DOS o de la BIOS establecen el bit CARRY del registro FLAGS en determinadas circunstancias.

`jnc label`: Contrario a jc. Saltar a la etiqueta si el bit CARRY del registro FLAGS se encuentra en 0.

**Saltos condicionales y comparaciones**

Uno de los usos más comunes de las instrucciones de salto condicional es comparar si un operando es mayor o menor que otro para ejecutar código diferente de acuerdo con el resultado. Para ello, las instrucciones de salto condicional van precedidas de instrucciones de comparación:

`cmp fuente, destino` (AT&T) o

`cmp destino, fuente` (Intel)

Otras instrucciones de salto condiciona que también se pueden utilizar para números con signo son:

`jg label`: *jump if greater*: Saltar a la etiqueta `label` si la comparación con signo determinó que el operando de destino es mayor que el operando de fuente

`jl label`: jump if less: Saltar a la etiqueta `label` si la comparación determinó que el operando de destino es menor que el operando de fuente

`jge label`: Jump if greater of equal: Saltar a la etiqueta `label` si el operando de destino es mayor o igual que el operando de fuente

`jle`: Jump if less or equal : Saltar a la etiqueta `label` si el operando de destino es menor o igual que el operando fuente.

**EJEMPLOS**

En el siguiente ejemplo se almacena el valor inmediato 100 en el registro AL y luego se realiza una serie de comparaciones con otros valores inmediatos y valores almacenados en otros registros.

**En sintaxis AT&T:**

```
movb $100, %al /* A los valores inmediatos en decimal  no se les
                  antepone '0x' como a los hexa*/
```

```
cmpb $50, %al /* Esta instrucción compara el valor de %al  con 50*/
jg es_mayor
    /* Otras instrucciones, que se ejecutan si el valor de %al no es
    mayor que 50  (en este caso no se ejecutan, %al = 100 > 50 */
jmp continuar
    /* Este salto es necesario, ya que de otro modo el procesador
          ejecutará las instrucciones anteriores y las siguientes
          también, lo cual es un error de lógica*/
es_mayor:
    /*  La ejecución continua aquí si el valor de %al (100) es mayor
    que 50*/
continuar:
   /*  Fin de la comparación. El código de este punto hacia
     Abajo se ejecutará para cualquier valor de %al */
```

**En sintaxis Intel:**

```
 mov al, 100  /* A los valores inmediatos en decimal  no se les
                 antepone '0x' como a los hexa*/
cmpb al, 50 /* Esta instrucción compara el valor de %al  con
                 50*/
jg es_mayor
    /* Otras instrucciones, que se ejecutan si el valor de %al no es
    mayor que 50  (en este caso no se ejecutan, %al = 100 > 50 */
jmp continuar
    /* Este salto es necesario, ya que de otro modo el procesador
          ejecutará las instrucciones anteriores y las siguientes
          también, lo cual es un error de lógica*/
es_mayor:
    /*  La ejecución continua aquí si el valor de %al (100) es mayor
    que 50*/
continuar:
   /*  Fin de la comparación. El código de este punto hacia
     Abajo se ejecutará para cualquier valor de %al */
```

### 2.5.2 BIFURCACIONES

A continuación se presenta un ejemplo de cómo realizar bifurcaciones en código ensamblador, usando para ello la instrucción `cmp` (compare) e inmediatamente después saltos condicionales o incondicionales. Para este ejemplo en modo real se compara el valor de los registros AX y BX, considerando AX como el registro destino y BX como el registro fuente. Se supone que AX y BX ya contienen los valores que se desean comparar.

El pseudocódigo es el siguiente:

```
si ax = bx
goto son_iguales
si ax > bx
goto es_mayor
goto es_menor
son_iguales:
(instrucciones si son iguales)
```

```
 goto continuar <-- Necesario para que el procesador 'salte' a la
                    etiqueta continuar luego de ejecutar las
                    instrucciones si ax y bx son iguales
es_mayor:
 (instrucciones si es mayor)
 goto continuar  <-- <-- Necesario para que el procesador 'salte' a la
                    etiqueta continuar luego de ejecutar las
                    instrucciones si ax es mayor que %bx
es_menor:
 (instrucciones si es menor)
continuar: <-- Fin de la bifurcación. En cualquiera de los tres casos
                         la ejecución continúa en este punto
 (instrucciones luego de la bifurcación)
```

**En sintaxis AT&T:**

```
 cmpw %bx, %ax /* Comparar %ax con %bx fuente: bx, destino: ax */
 je son_iguales /* Si son iguales, saltar a la etiqueta son_iguales */
 jg es_mayor /* Si %ax es mayor, saltar a la etiqueta  es_mayor*/
 jmp es_menor /* en el caso contrario, saltar a la etiqueta es_menor */
 son_iguales:
    /* Instrucciones a ejecutar si %ax y %bx son iguales */
 jmp continuar /* Saltar a la etiqueta continuar (mas abajo), de
                lo  contrario  el procesador continúa ejecutando
                el código siguiente!!*/
 es_mayor:
  /* Instrucciones a ejecutar si %ax es mayor que %bx */
  jmp continuar /* También se debe saltar a la etiqueta
           continuar, para  evitar que el procesador también ejecute
           el  código siguiente */
 es_menor:
  /* Instrucciones a ejecutar si %ax es menor que %bx */
 continuar:
 /* Instrucciones que se ejecutarán luego de la bifurcación */
```

**En sintaxis Intel:**

```
 cmp ax, bx /* Comparar ax con bx fuente: bx, destino: ax */
 je son_iguales /* Si son iguales, saltar a la etiqueta
                son_iguales */
 jg es_mayor /* Si ax es mayor, saltar a la etiqueta  es_mayor*/
 jmp es_menor /* en el caso contrario, saltar a la etiqueta
                es_menor */
 son_iguales:
    /* Instrucciones a ejecutar si ax y bx son iguales */
 jmp continuar /* Saltar a la etiqueta continuar (mas abajo), de
                     lo contrario el procesador continúa
                     ejecutando el código siguiente!!*/
```

```
es_mayor:
 /* Instrucciones a ejecutar si ax es mayor que bx */
jmp continuar /* También se debe saltar a la etiqueta continuar,
                 para evitar que el procesador también ejecute
                 el código   siguiente */
es_menor:
 /* Instrucciones a ejecutar si ax es menor que bx */
continuar:
 /* Instrucciones que se ejecutaran luego de la bifurcación */
```

### 2.5.3 CICLOS

Los ciclos son un componente fundamental de cualquier programa, ya que permiten repetir una serie de instrucciones un número determinado de veces. Existen varias formas de implementar los ciclos. Las dos formas más acostumbradas son 1) combinar un registro que sirve de variable, una comparación de este registro y una instrucción de salto condicional para terminar el ciclo, o 2) usar la instrucción loop y el registro CX. A continuación se ilustran los dos casos. En ciclos más avanzados, las condiciones son complejas e involucran el valor de una o más variables o registros.

El pseudocódigo es el siguiente:

```
 cx = N
-> ciclo:
|   si cx = 0
|    goto fin_ciclo
| (Demás instrucciones del ciclo)
| decrementar cx
|_goto ciclo
fin_ciclo:
Instrucciones a ejecutar luego del ciclo
```

a. ***Implementación usando a %cx como contador y realizando la comparación***:

**En sintaxis AT&T:**

```
movw $10, %cx /* Para el ejemplo, repetir 10 veces */
ciclo:
cmpw $0, %cx /* Comparar %cx con cero*/
je fin_ciclo /* Si %cx = 0, ir a la etiqueta fin_ciclo */
/* Demás instrucciones del ciclo */
decw %cx /* %cx = %cx - 1 */
jmp ciclo /* salto incondicional a la etiqueta ciclo*/
fin_ciclo:
/* Instrucciones a ejecutar después del ciclo */
```

**En sintaxis Intel:**

```
mov cx, 10 /* Para el ejemplo, repetir 10 veces */
ciclo:
cmp cx, 0 /* Comparar cx con cero*/
je fin_ciclo /* Si cx = 0, ir a la etiqueta fin_ciclo */
/* Demás instrucciones del ciclo */
dec cx /* cx = cx - 1 */
jmp ciclo /* salto incondicional a la etiqueta ciclo*/
fin_ciclo:
/* Instrucciones a ejecutar después del ciclo */
```

***b. Implementación usando la instrucción loop y el registro %cx:***

**En sintaxis AT&T:**

```
 movw $10, %cx /* Para el ejemplo, repetir 10 veces */
 ciclo:
  /* Demás instrucciones dentro del ciclo
     Importante: Recuerde que para el ciclo, se utiliza el registro %cx.
     Por esta razón no es aconsejable utilizarlo dentro del ciclo. */

 loop ciclo /* Decrementar automáticamente %cx y verificar si es
               mayor  que cero. Si %cx es mayor que cero, saltar
               a la  etiqueta 'ciclo'. En caso contrario,
               continuar la ejecución en la instrucción

               siguiente*/
```

**En sintaxis Intel:**

```
 mov cx, 10 /* Para el ejemplo, repetir 10 veces */

ciclo:
  /* Demás instrucciones dentro del ciclo
     Importante: Recuerde que para el ciclo, se utiliza el registro cx.
     Por esta razón no es aconsejable utilizarlo dentro del ciclo.
  */
loop ciclo /* Decrementar automáticamente cx y verificar si es
            mayor que cero. Si cx es mayor que cero, saltar a la
            etiqueta 'ciclo' En caso contrario, continuar la
            ejecución en la instrucción siguiente*/
```

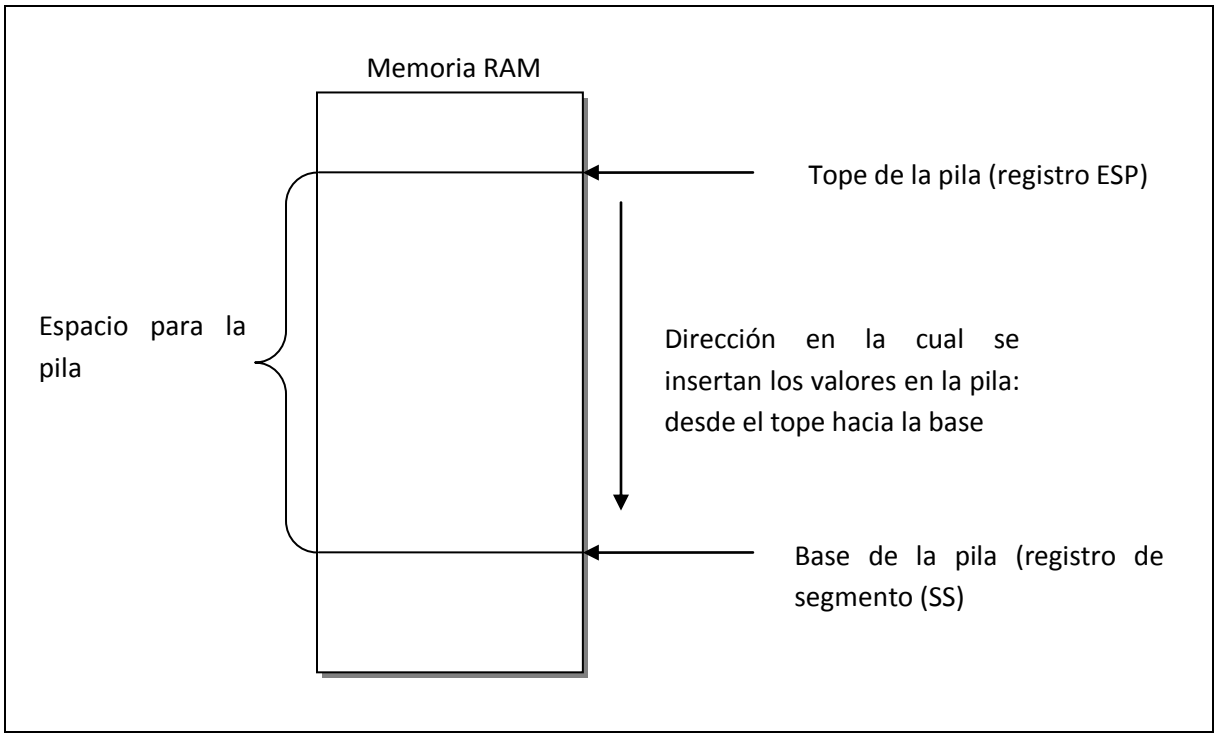
Ambas estrategias son válidas. Generalmente se utiliza la instrucción 'loop' y el registro CX. Sin embargo, cuando la condición es más compleja, se utiliza la primera aproximación.

## 2.6 USO DE LA PILA EN IA-32

Los procesadores de arquitectura IA-32 proporcionan tres registros para manipular la pila: el registro SS (stack segment), el registro ESP (stack pointer), y el registro BP (base pointer)[1].

La pila tiene la siguiente disposición:

[1] En modo real solo se tiene acceso a los 16 bits menos significativos: SP y BP. En modo de 64 bits ESP se expande a RSP y EBP se expande a RBP (registros de 64 bits).




Es importante resaltar que en los procesadores IA-32 la pila crece de una posición más alta a una posición más baja en la memoria. Es decir, cada vez que se almacena un byte o un word en la pila, ESP se decrementa y apunta a una dirección menor en la memoria.

### 2.6.1 Operaciones sobre la pila

Las operaciones sobre la pila dependen del modo de ejecución del procesador, de la siguiente forma:

- **Modo real**: En modo real (16 bits), la unidad mínima de almacenamiento en la pila es un word. Esto significa que si un programa intenta almacenar un byte en la pila (que es válido), el procesador insertará automáticamente un byte vacío para mantener una pila uniforme con unidades de almacenamiento de dos bytes (16 bits).
- **Modo protegido**: En este modo la unidad mínima de almacenamiento es un doubleword (4 bytes, 32 bits). Si se almacena un byte o un word, automáticamente se insertan los bytes necesarios para tener una pila uniforme con unidades de almacenamiento de cuatro bytes (32 bits). En el modo de compatibilidad de 64 bits también se usa esta unidad de almacenamiento.
- **Modo de 64 bits**: La unidad mínima de almacenamiento es un quadword (8 bytes, 64 bits). Tambien se pueden almacenar bytes, words o doublewords, ya que el procesador inserta los bytes necesarios en cada caso para tener unidades uniformes de 8 bytes (64 bits).

A continuación se presentan las instrucciones básicas para el manejo de la pila.

**Instrucción push:** La instrucción push almacena un valor inmediato (constante), o el valor de un registro en el tope de la pila. El apuntador al tope de la pila (ESP) se decrementa en el número de bytes almacenados.

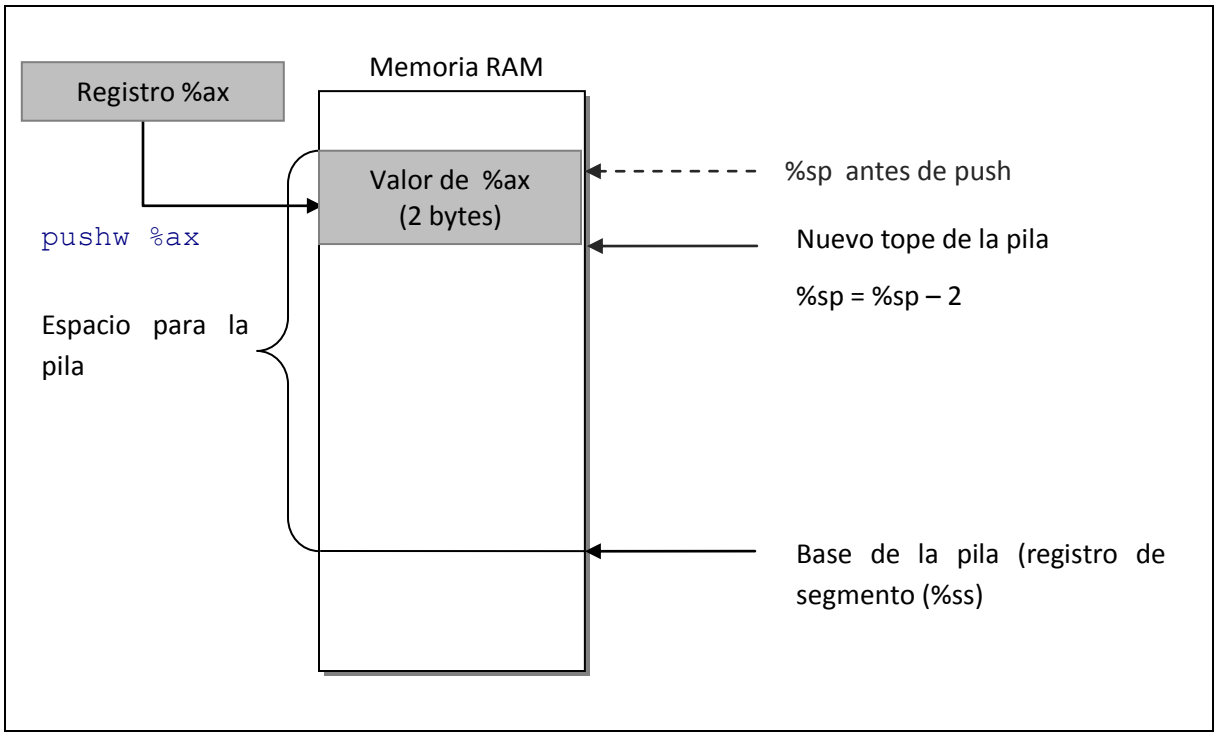
La siguiente instrucción en modo real permite almacenar el valor del registro AX en la pila (un word):

```
pushw %ax
```

En sintaxis Intel, simplemente se omite el sufijo 'w':

```
push ax
```

Esta instrucción almacena en la pila el valor de AX (2 bytes), y decrementa el valor de SP en 2. La pila lucirá así:




La instrucción push permite almacenar en la pila los valores de los registros del procesador y también valores inmediatos.

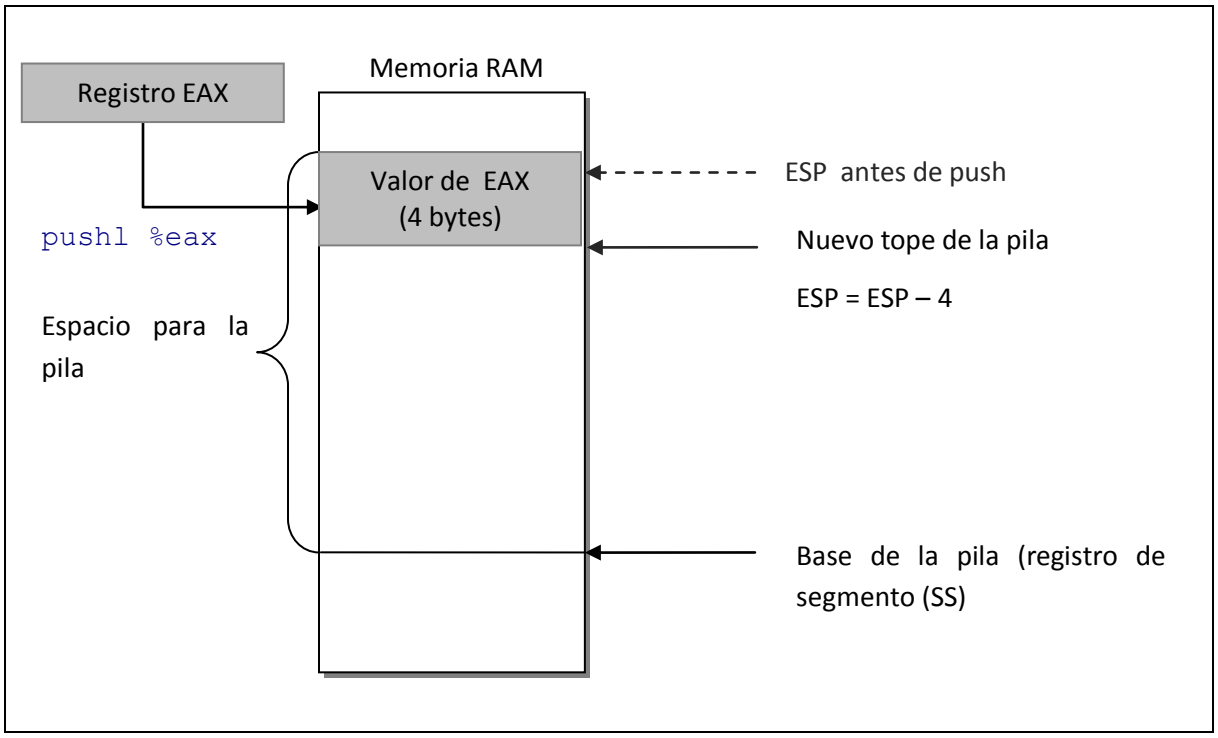
La siguiente instrucción en modo protegido de 32 bits permite almacenar el valor del registro EAX en la pila (un doubleword, 32 bits):

```
pushl %eax
```

En sintaxis Intel, simplemente se omite el sufijo 'l':

```
push eax
```

Esta instrucción almacena en la pila el valor de EAX (4 bytes), y decrementa el valor de ESP en 4. La pila lucirá así:




**Instrucción pop:** Por su parte, la instrucción *pop* retira un valor de la pila (word, doubleword o quadword según el modo de operación), lo almacena en el registro destino especificado e incrementa SP (ESP o RSP según el modo de operación) en 2, 4 o 8. Se debe tener en cuenta que luego de sacar un valor de la pila, no se puede garantizar que el valor sacado se conserve en la pila.

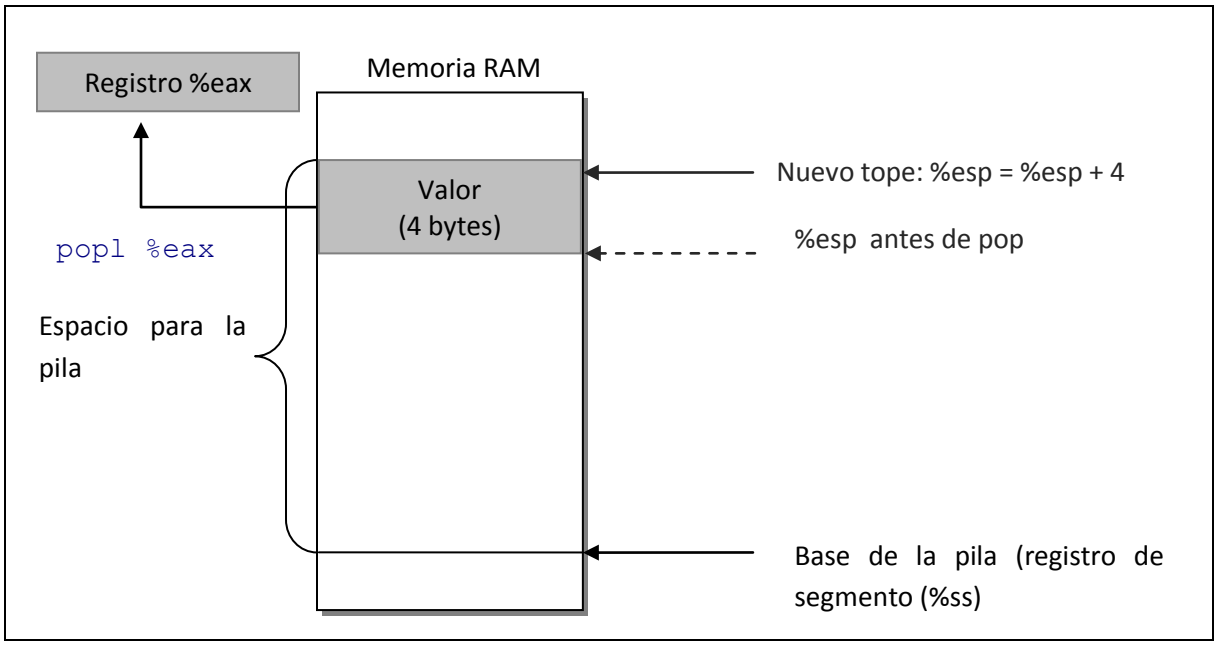
Por ejemplo, para sacar un valor del tope de la pila en modo protegido de 32 bits y almacenarlo en el registro EAX, se usa la siguiente instrucción:

```
popl %eax
```

En sintaxis Intel:

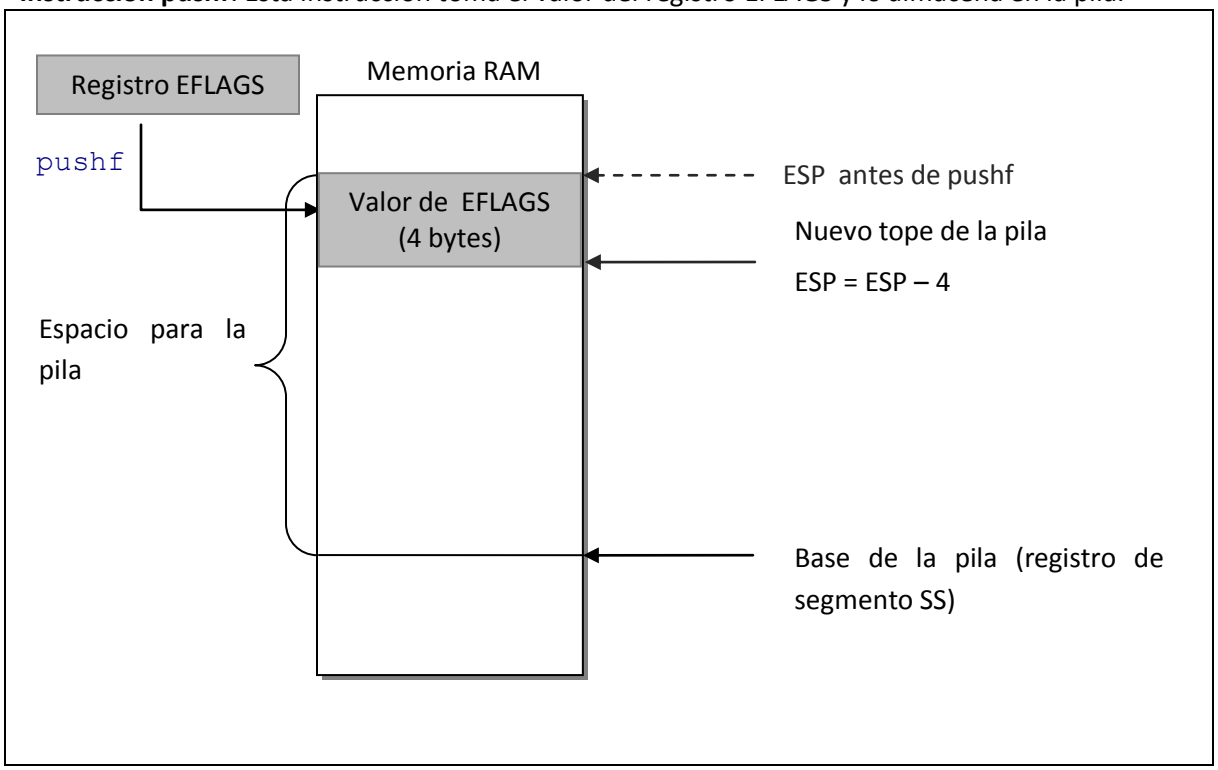
```
pop eax
```

Esta instrucción saca del tope de la pila un doubleword (cuatro bytes) y los almacena en el registro EAX, como lo muestra la siguiente figura.



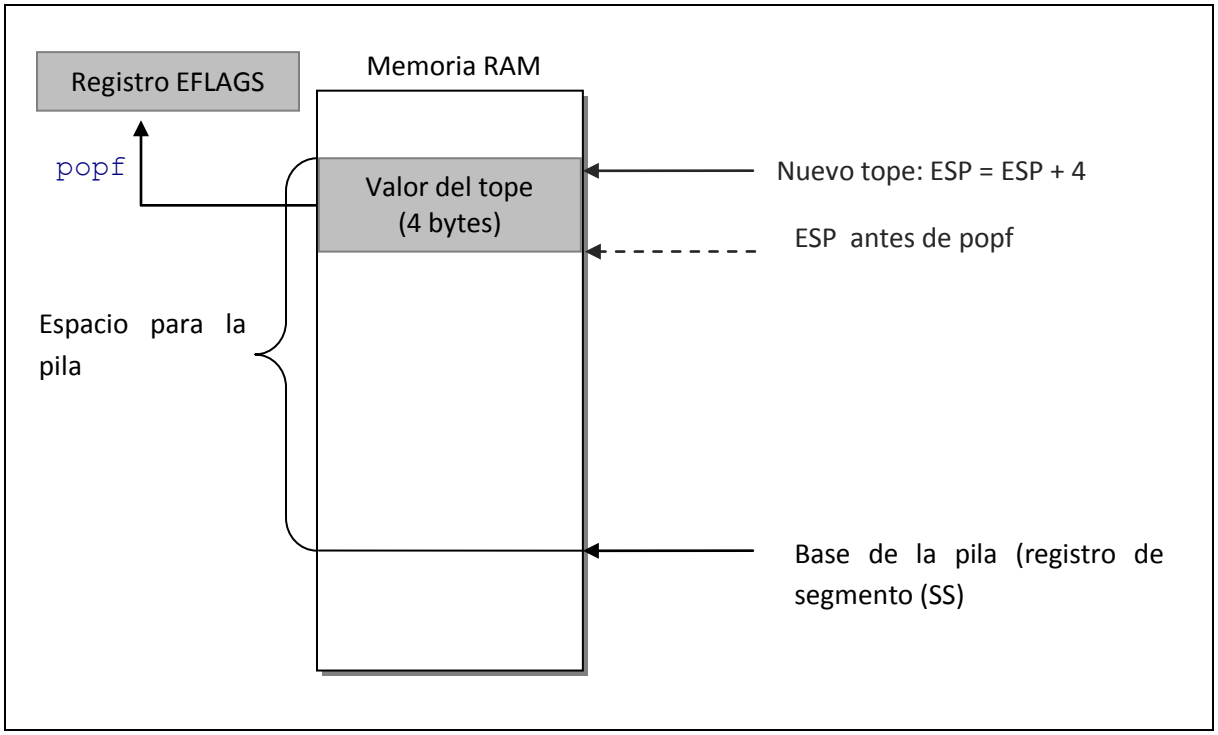

La instrucción *pop* toma el valor del tope de la pila y lo almacena en el registro de destino especificado en la misma operación. Se debe tener en cuenta que luego de extraer un valor de la pila no se garantiza que aún siga allí. En modo real la instrucción *pop* retira dos bytes de la pila, y en modo de 64 bits retira 8 bytes de la pila.

**Instrucción pushf**: Esta instrucción toma el valor del registro EFLAGS y lo almacena en la pila.




En modo real solo se puede tener acceso a los 16 bits menos significativos de EFLAGS, por lo cual solo se ocupan 2 bytes en la pila y SP se decrementa en 2. En modo de 64 bits se ocupan 8 bytes (64 bits) para el registro RFLAGS.

**Instrucción popf**: Esta instrucción toma el valor del tope de la pila y lo almacena en el registro EFLAGS (32 bits), los 16 bits menos significativos de EFLAGS en modo real y RFLAGS en modo de 64 bits.

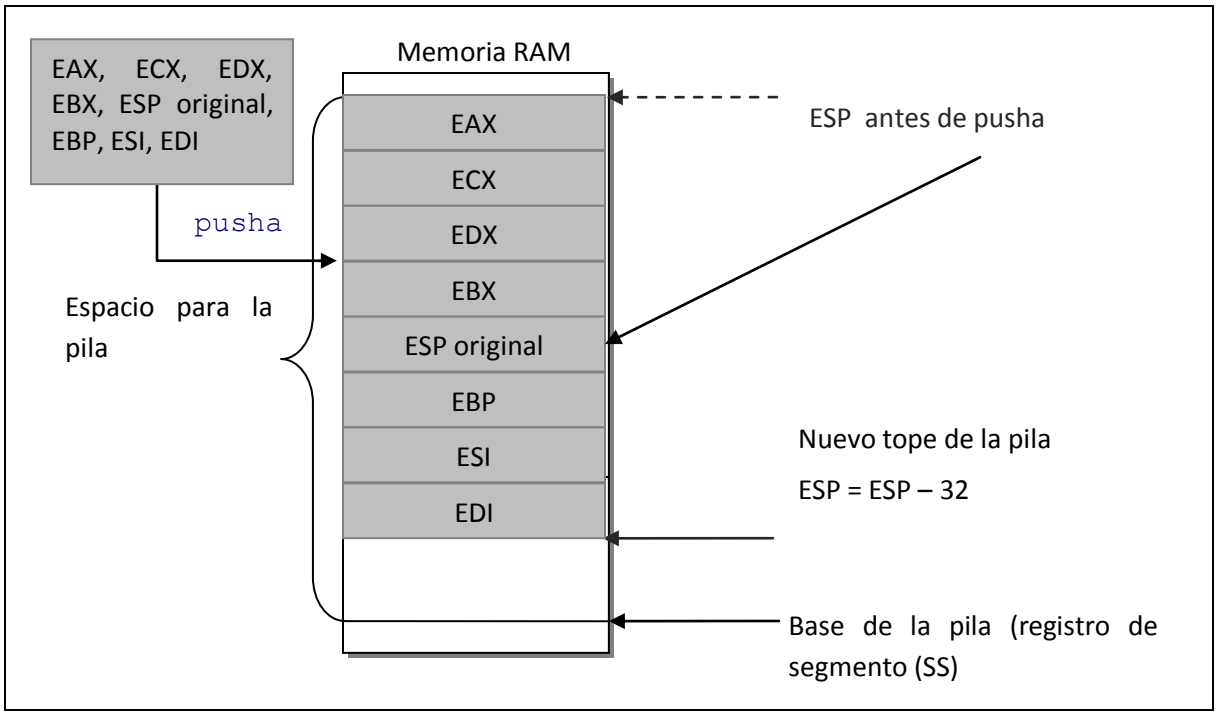



**Instrucción pusha:** Esta instrucción almacena el valor de los registros de propósito general en la pila. De acuerdo con el modo de operación del procesador, se almacenarán los registros en el siguiente orden:

- En modo protegido de 32 bits, se almacenan EAX, ECX, EDX, EBX, valor de ESP antes de pusha, EBP, ESI y EDI.
- En modo real (16 bits), se almacenan AX, CX, DX, BX, valor de SP antes de pusha, BP, SI y DI.
- En modo de 64 bits, se almacenan RAX, RCX, RDX, RBX, valor de RSP antes de pusha, RBP, RSI y RDI.

Así, en modo protegido de 32 bits cada valor almacenado en la pila tomará cuatro bytes. En modo real, tomará dos bytes y en modo de 64 bits cada valor tomará 8 bytes.

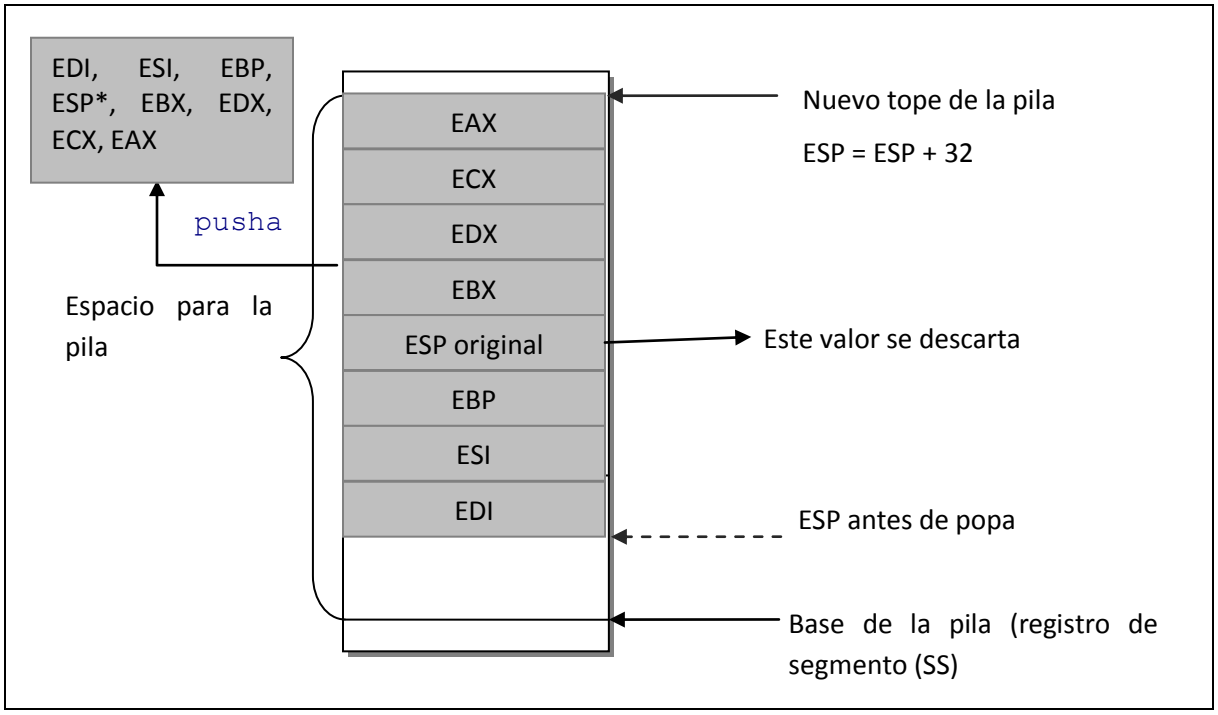
A continuación se presenta un diagrama del funcionamiento de la instrucción PUSHA en modo protegido de 32 bits.




**Instrucción popa**: Esta instrucción extrae de la pila ocho valores, y los almacena en los registros de propósito general, en el siguiente orden (inverso al orden de pusha):

EDI, ESI, EBP, ESP*, EBX, EDX, ECX, EAX

*No obstante, el valor de ESP sacado de la pila se descarta.



## 2.7 IMPLEMENTACIÓN DE RUTINAS

En los lenguajes de alto nivel, además de bifurcaciones y ciclos, también se usan las rutinas como mecanismos para organizar la funcionalidad de un programa. Para implementar rutinas, el procesador incluye entre otras las instrucciones `call` (invocar una rutina) y `ret` (retornar de la rutina).

Una rutina consta de varios componentes, entre los cuales sobresalen:

1. Nombre de la rutina: Símbolo dentro de la sección de texto que indica el inicio de la rutina.
2. Parámetros de entrada: Se utiliza la pila para pasar los parámetros a las funciones.
3. Dirección de retorno: Dirección a la cual se debe retornar una vez que se ejecuta la rutina

Por ejemplo, para definir una rutina llamada `nombre_rutina`, el código sería el siguiente:

```
nombre_rutina:/* Inicio de la rutina */
   /* Instrucciones de la rutina*/
  ret /* Fin de la rutina  (retornar) */
```

Es necesario notar que la definición de una etiqueta no necesariamente implica la definición de una rutina. El concepto de "Rutina" lo da el uso que se haga de la etiqueta. Si para saltar a una etiqueta se usa la instrucción de salto incondicional (`jmp`) o alguna instrucción de salto condicional (`j..`), esta no es una rutina. Si por el contrario, para saltar a una etiqueta se usa la instrucción call (ver explicación más adelante), y después de esa etiqueta existe una instrucción `ret` a la cual se llega sin importar la lógica de programación, entonces la etiqueta sí puede ser considerada una "Rutina".

### 2.7.1 Llamada a rutinas

La llamada a una rutina se realiza por medio de la instrucción `call` (en sintaxis AT&T e Intel), especificando la etiqueta (el nombre de la rutina) definido en ensamblador:

```
call nombre_rutina
```

De esta forma, se estará invocando a la rutina `nombre_rutina`, sin pasarle parámetros.

También es posible, aunque poco común, realizar llamadas a rutinas que se encuentran en otros segmentos de memoria. En este caso se utiliza la instrucción `lcall`.

### 2.7.2 Parámetros de entrada de las rutinas

Si se desean pasar parámetros a una rutina, éstos se deben almacenar en la pila mediante la instrucción *push*, en el orden inverso en el que se van a utilizar en la rutina antes de ejecutar la instrucción *call*.

Por ejemplo, para invocar a una rutina y pasarle n parámetros (parámetro 1, parámetro 2, .. , parámetro n), primero se deben insertar los parámetros en orden inverso en la pila (del último al primero) antes de la instrucción *call*:

```
push parametro n
push parametro n-1
...
```

```
push parametro 2
push parametro 1
```

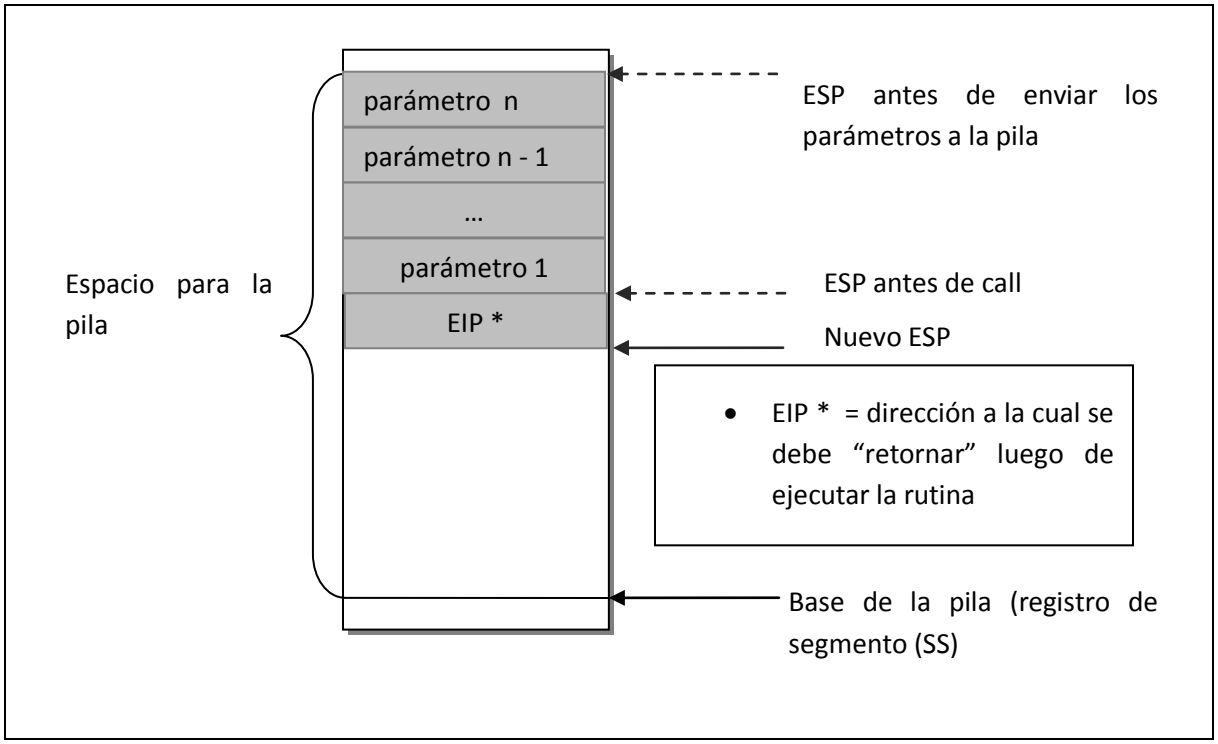
La cantidad de bytes que ocupa cada parámetro en la pila depende del modo de operación del procesador. En modo real, cada parámetro ocupa dos bytes (16 bits). En modo protegido, cuatro bytes (32 bits) y en modo de 64 bits cada parámetro ocupa ocho bytes.

Luego se utiliza la instrucción `call`, especificando el símbolo (la etiqueta) de la rutina que se desea ejecutar:

```
call nombre_rutina
```

*La instrucción call almacena automáticamente en la pila dirección de memoria de la próxima instrucción a ejecutar luego del call (la dirección de retorno), y establece el registro EIP (instruction pointer) al desplazamiento en el segmento de código en la cual se encuentra definido el símbolo con el nombre de la rutina*.

De esta forma, en el momento de llamar a una rutina, la pila se encuentra así:

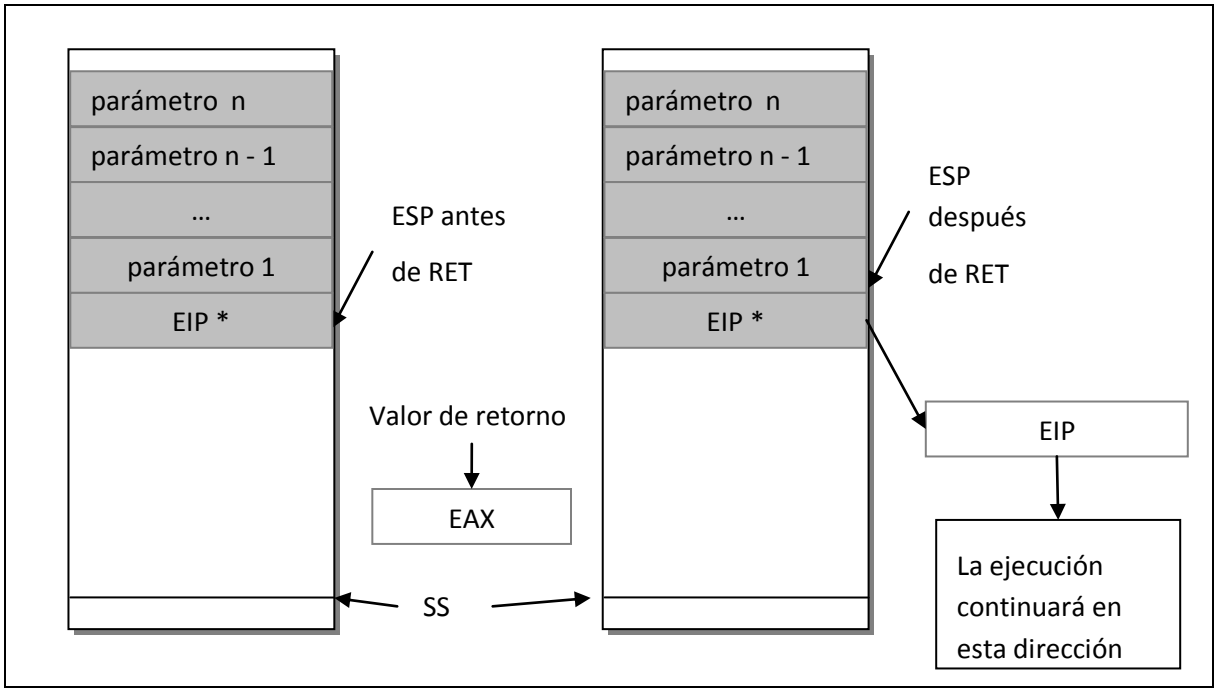



## 2.7.3 Valor de retorno de las rutinas

Cuando dentro de una rutina se busque retornar la ejecución al punto en el cual fue invocada, se debe usar la instrucción *ret*. Esta instrucción extrae el valor que se encuentre en el tope de la pila y establece el registro EIP con el valor extraído. Así, si dentro de la rutina se ha modificado la pila (se han insertado nuevos valores y se han extraído otros), es importante que el código de la rutina garantice que el valor encontrado el tope de la pila antes de ejecutar la instrucción ret sea la dirección de retorno (la posición de memoria en la cual debe continua la ejecución).

Es importante tener en cuenta que la instrucción *ret* difiere un poco de la instrucción *return* de lenguajes de alto nivel, en la cual se puede retornar un valor. En ensamblador, el valor de retorno por convención se almacena siempre en el registro AL, AX, EAX o RAX de acuerdo con el modo de operación del procesador. Así, una de las últimas instrucciones dentro de la rutina antes de *ret* deberá almacenar el valor de retorno en el registro EAX.

El siguiente diagrama ilustra el funcionamiento de la instrucción ret.




Dado que al retornar de la rutina los parámetros aún se encuentran en la pila, es necesario avanzar ESP para que apunte a la posición de memoria en la cual se encontraba antes de insertar los parámetros. Para lograr este propósito se adiciona un número de bytes a ESP:

```
add esp, #
```

Donde '#' corresponde al número de bytes que se almacenaron en la pila como parámetros. En modo real cada parámetro ocupa dos bytes en la pila, por lo cual se deberá sumar 2 * el número de parámetros a SP. En modo protegido cada parámetro ocupa cuatro bytes en la pila, por lo que se deberá sumar 4 * el número de parámetros a ESP. En modo de 64 bits se deberá sumar 8 * el número de parámetros a RSP.

De forma general, el formato para invocar una rutina que recibe n parámetros es el siguiente:

```
push parametro n
push parametro n-1
...
push parametro 2
push parametro 1

call nombre_rutina

add esp, #
```

## 2.7.4 Implementación de una rutina en lenguaje ensamblador

A continuación se muestra la implementación general de una rutina en lenguaje ensamblador. Dentro de la rutina se crea un "marco de pila", necesario para manejar correctamente las variables que fueron pasadas como parámetro en la pila y las variables locales. El concepto de "marco de pila" se explicará tomando como base la plantilla de rutina en modo real. En los demás modos de operación del procesador el marco de pila funciona en forma general, pero se deben expandir los registros a sus equivalentes en 32 y 64 bits.

### *2.7.4.1 Plantilla de rutina en modo real*

Los siguientes ejemplos presentan una plantilla que puede ser usada para implementar una rutina en modo real.

**En sintaxis AT&T:**

```
nombre_rutina:
   pushw %bp    /*Almacenar %bp en la pila*/
   movw %sp, %bp  /*Establecer %bp con el valor de %sp*/
              /*Ya se ha creado un marco de pila*/
   ...
   (instrucciones de la rutina)
   ...

              /*Cerrar el marco de pila:*/
   movw %bp, %sp  /*Mover %bp a %sp*/
   popw %bp/*Recuperar el valor original de %bp */

   ret /* Retornar de la rutina */
```

**En sintaxis Intel:**

```
nombre_rutina:
   push bp    /*Almacenar bp en la pila*/
   mov bp, sp  /*Establecer bp con el valor de sp*/
              /*Ya se ha creado un marco de pila*/
   ...
   (instrucciones de la rutina)
   ...

              /*Cerrar el marco de pila:*/
   mov sp, bp  /*Mover bp a sp*/
   pop bp/*Recuperar el valor original de %bp */

   ret  /* Retornar de la rutina*/
```

Ahora se explicará en detalle esta plantilla.

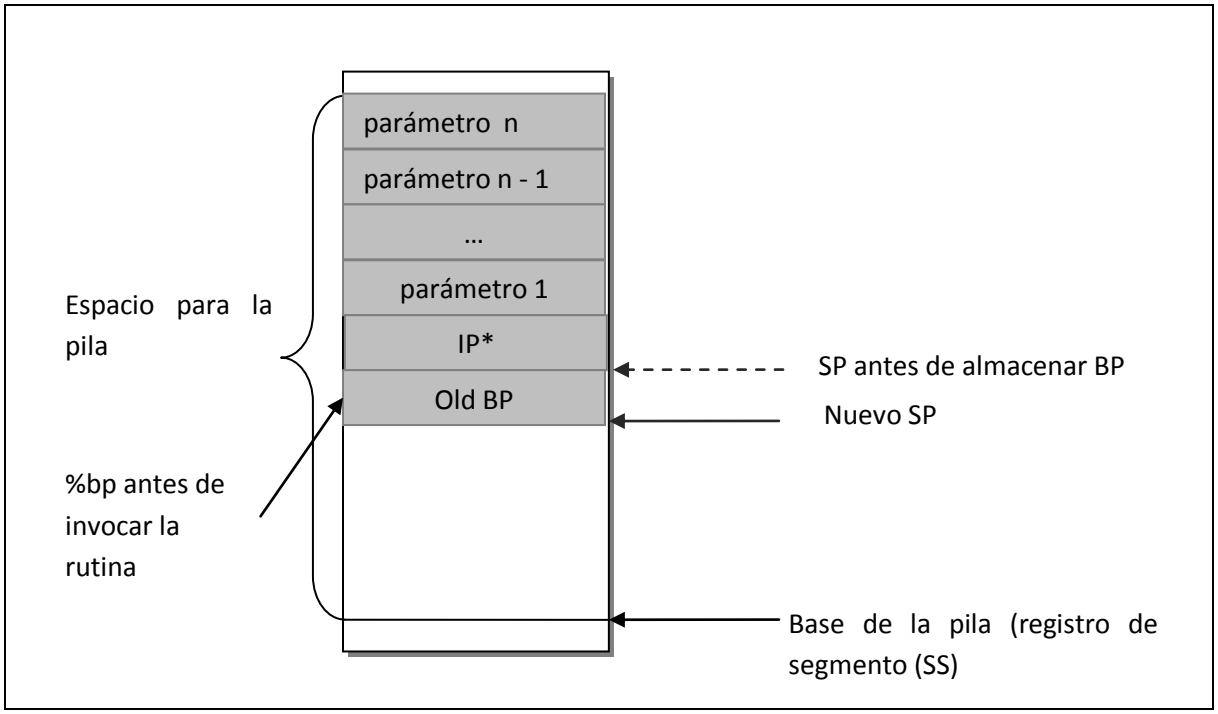
Con la instrucción

```
pushw %bp     /* Sintaxis AT&T */
```

ó

```
push bp           /* Sintaxis Intel */
```

La pila queda dispuesta de la siguiente forma:




Observe que SP apunta a la posición de memoria en la pila en la cual se almacenó el valor que tenía BP originalmente. Esto permite recuperar el valor original de BP, luego de terminadas las instrucciones de la rutina y antes de retornar al punto desde el cual se invocó.

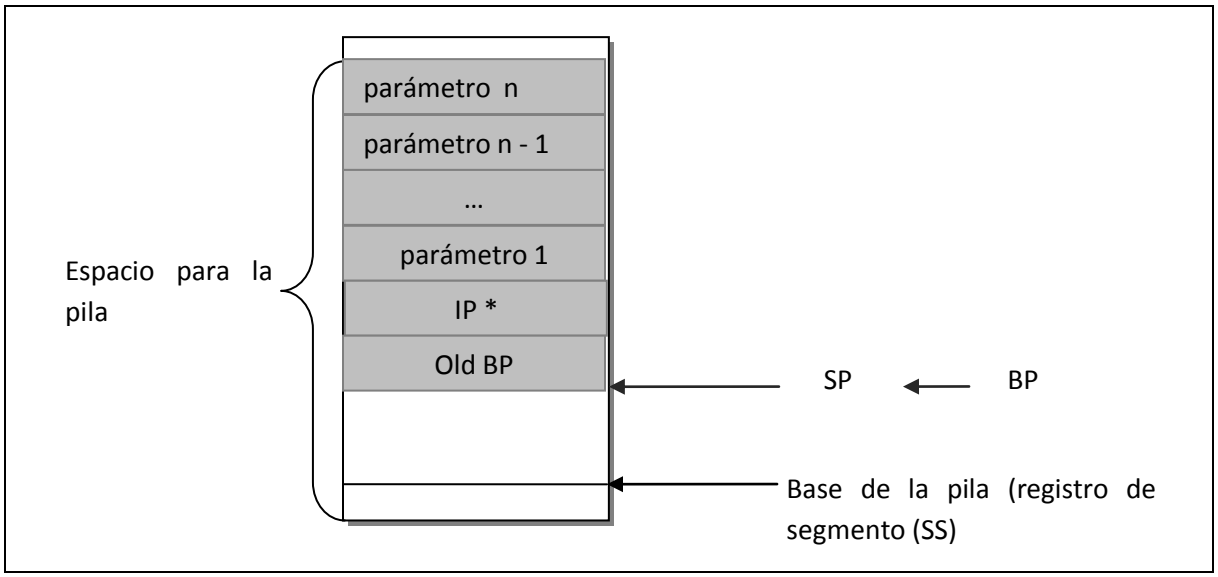
La instrucción

```
movw %sp, %bp /* Sintaxis AT&T */
```

ó

```
mov bp, sp /* Sintaxis Intel */
```

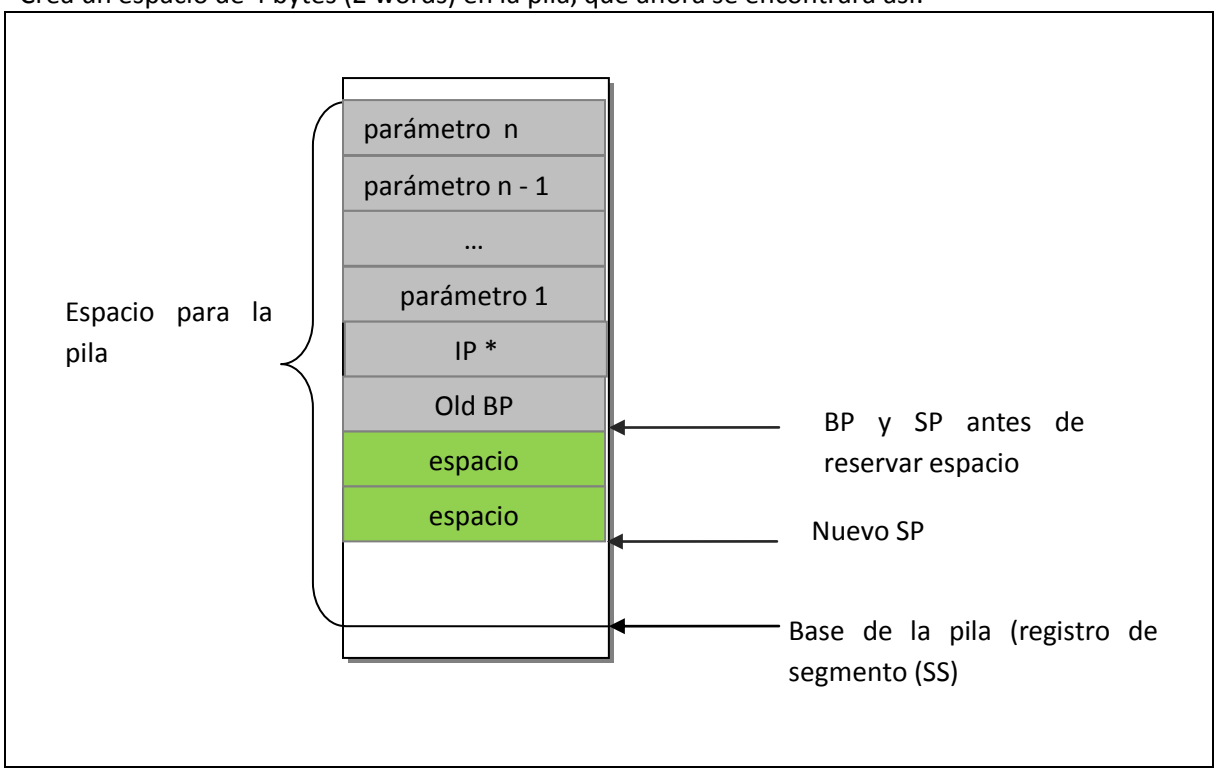
Establece a BP con el mismo valor de SP, con lo cual BP apunta a la misma dirección de memoria a la cual apunta SP:




Con esta instrucción se termina el proceso de crear el marco de pila. Ahora es totalmente seguro decrementar el valor de SP con el propósito de crear espacios para las variables locales a la rutina. Por ejemplo, la instrucción

```
subw $4, %sp /* Sintaxis AT&T */ ó sub sp, 4 /* Sintaxis Intel */
```

Crea un espacio de 4 bytes (2 words) en la pila, que ahora se encontrará así:



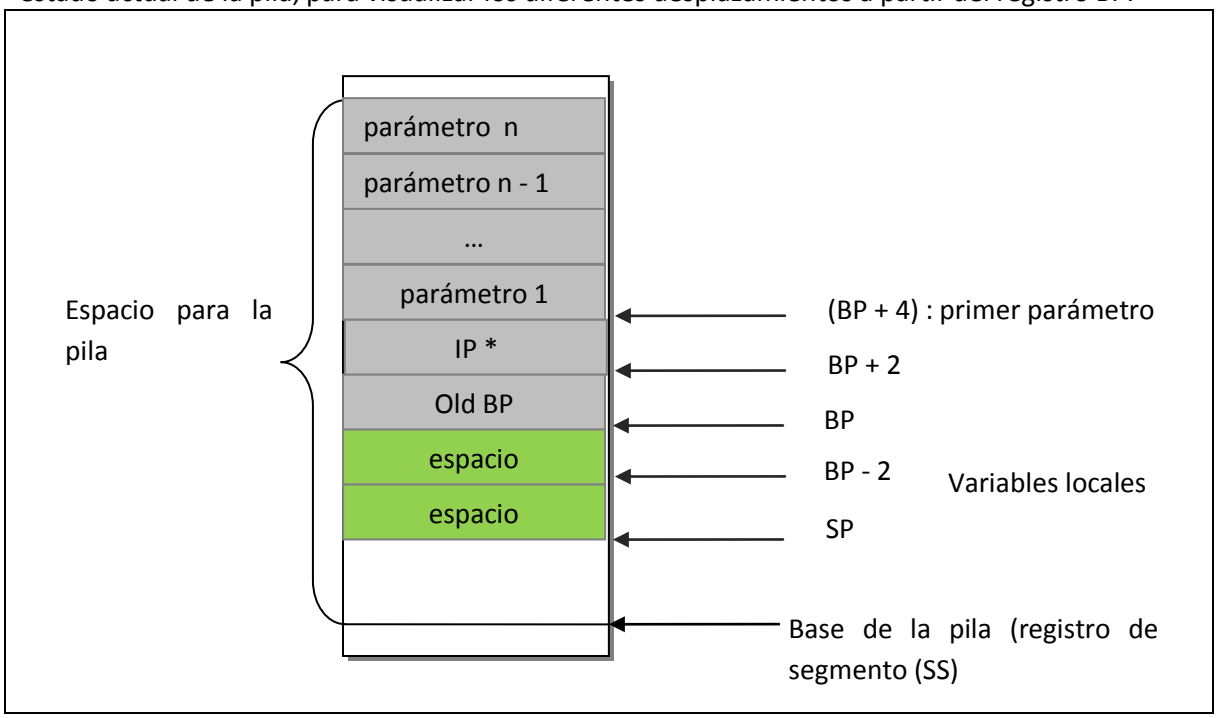

Observe que SP se decrementa, pero BP sigue apuntando al inicio del marco de pila. Por esta razón, el puntero EBP se denomina ***Base Pointer*** (puntero base), ya que con respecto a él es posible acceder tanto los parámetros enviados a la rutina, como las variables locales creadas en esta. Por ejemplo, la instrucción

```
movw 4(%bp), %ax /* Sintaxis AT&T */
```

ó

```
mov ax, WORD PTR [ bp + 4 ] /* Sintaxis Intel */
```

Mueve el contenido de la memoria en la posición (BP + 4) al registro AX, es decir que almacena el primer parámetro pasado a la rutina en el registro AX. A continuación se presenta de nuevo el estado actual de la pila, para visualizar los diferentes desplazamientos a partir del registro BP.




Por otro lado, la instrucción

```
movw %ax, -2(%bp) /* Sintaxis AT&T */
```

ó

```
mov WORD PTR [ bp - 2 ], ax /* Sintaxis Intel */
```

Almacena el valor del registro AX en el primer word de espacio de la pila.

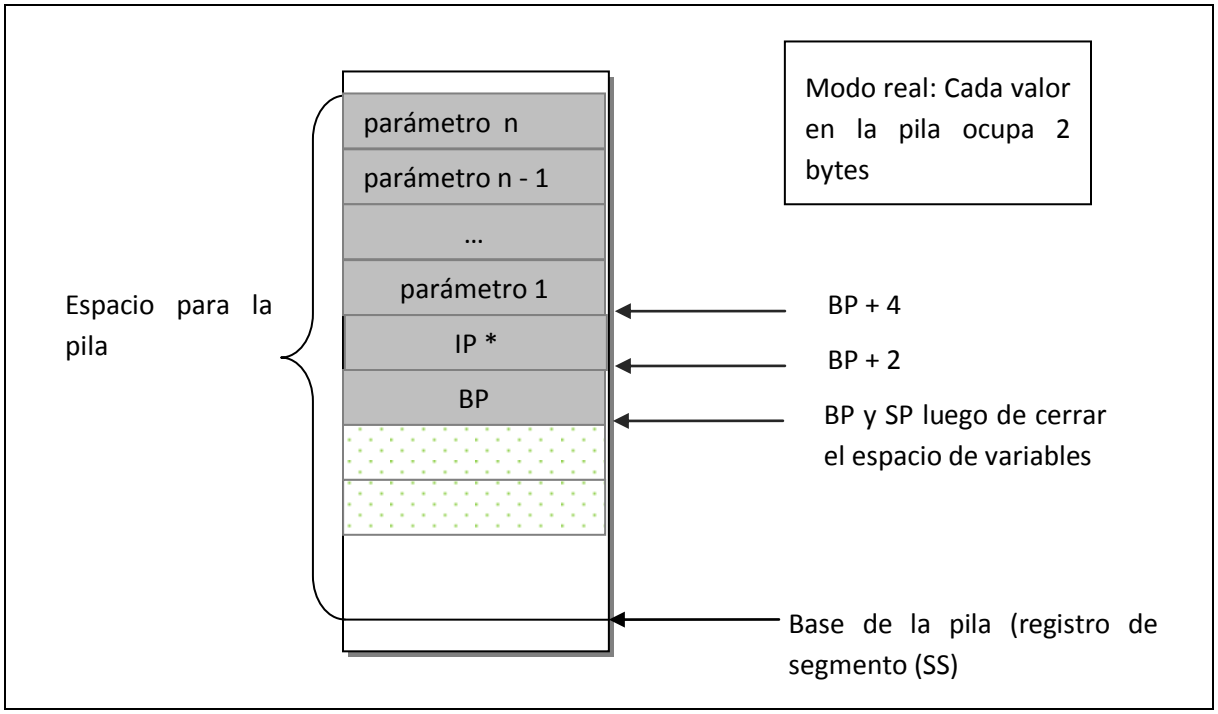
Se puede observar que si se crea un marco de pila estándar con las instrucciones encionadas, siempre el primer parámetro que se paso a la rutina se encontrará en (BP + 4), el segundo en (BP + 6) y así sucesivamente.

Se debe tener en cuenta que si dentro de la rutina se modifica el valor del registro BP, se deberá almacenar su valor con anterioridad en la pila o en otro registro.

Al finalizar la rutina se deberá cerrar el marco de pila creado. La instrucción

```
movw %bp, %sp /* Sintaxis AT&T */
mov sp, bp /* Sintaxis Intel */
```

Cierra el espacio creado para las variables locales, al apuntar SP a la misma dirección de memoria en la pila a la que BP. Luego de esta instrucción la pila lucirá así:




En este momento ya no es seguro acceder a los valores almacenados en el espacio para variables locales.

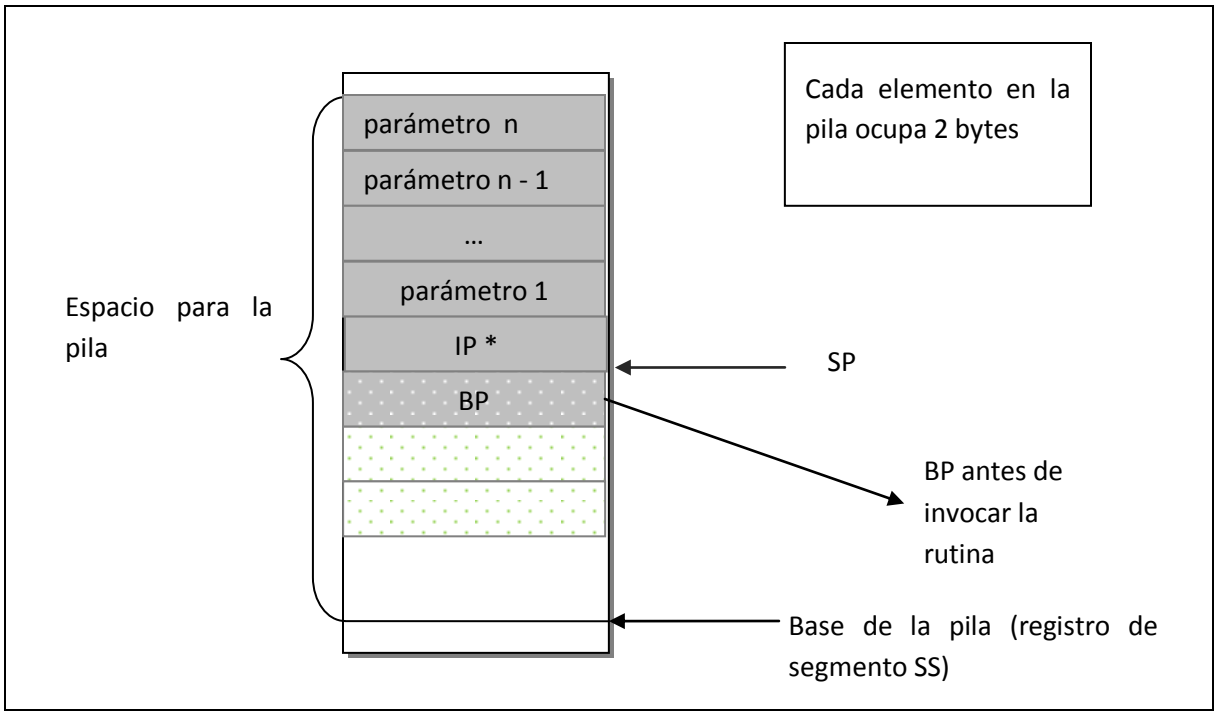
Ahora se deberá recuperar el valor original de BP antes de crear el marco de pila:

```
popw %bp /* Sintaxis AT&T */
```

ó

```
pop bp /* Sintaxis Intel */
```

Con ello la pila se encontrará en el siguiente estado:

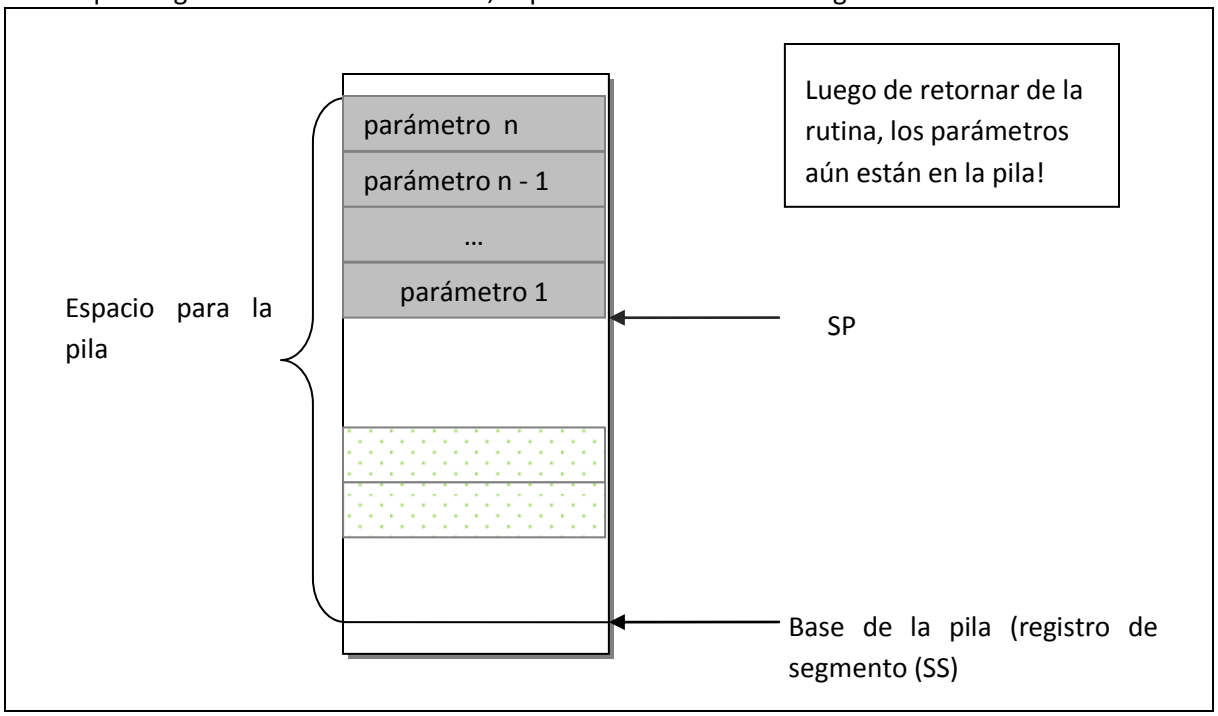



Finalmente, la instrucción

`ret`

Toma de la pila la dirección de retorno (la dirección de memoria de la instrucción siguiente a la cual se llamó la rutina mediante `call`), y realiza un *jmp* a esa dirección.

Note que luego de retornar de la rutina, la pila se encontrará en el siguiente estado:



Por esta razón es necesario avanzar SS en un valor igual al número de bytes que se enviaron como parámetro a la rutina. Si se enviaron N parámetros a la pila, el numero de bytes que se deberán sumar a %sp son M = 2 * N (En modo real cada parámetro ocupa un word = 2 bytes).

```
addw $M, %sp /* Sintaxis AT&T */
```

ó

```
add sp, M /* Sintaxis Intel */
```

Donde M representa el número de bytes a desplazar SP.

Con estas instrucciones la pila se encontrará en el mismo estado que antes de invocar la rutina.

### *2.7.4.2 Plantilla de rutina en modo protegido de 32 bits*

La plantilla de una rutina en modo protegido de 32 bits es muy similar a la de modo real. La principal diferencia entre las dos consiste en el tamaño de los registros, que se expanden de 16 a 32 bits (BP se expande a EBP y SP se expande a ESP). También es importante recordad que cada parámetro almacenado en la pila ocupa 4 bytes (32 bits).

**En sintaxis AT&T:**

```
nombre_rutina:
   pushl %ebp    /*Almacenar %ebp en la pila*/
   movw %esp, %ebp  /*Establecer %ebp con el valor de %esp*/
              /*Ya se ha creado un marco de pila*/
   ...
   (instrucciones de la rutina)
   ...

              /*Cerrar el marco de pila:*/
   movw %ebp, %esp  /*Mover %ebp a %esp*/
   popw %ebp/*Recuperar el valor original de %ebp */

   ret /* Retornar de la rutina */
```

**En sintaxis Intel:**

```
nombre_rutina:
   push ebp    /*Almacenar ebp en la pila*/
   mov ebp, esp  /*Establecer ebp con el valor de esp*/
              /*Ya se ha creado un marco de pila*/
   ...
   (instrucciones de la rutina)
   ...

              /*Cerrar el marco de pila:*/
   mov esp, ebp  /*Mover ebp a esp*/
   pop ebp/*Recuperar el valor original de ebp */

   ret  /* Retornar de la rutina*/
```