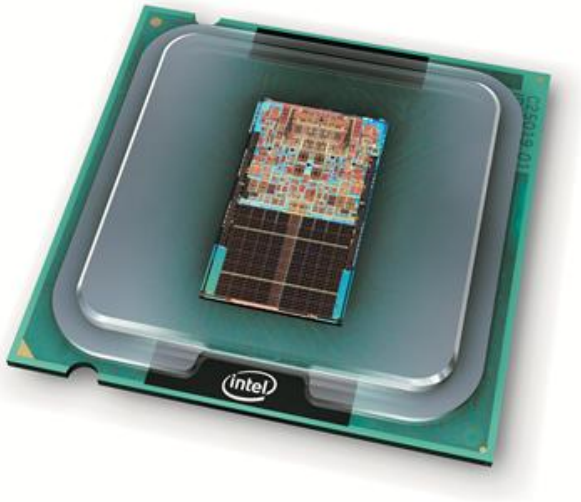



# La Pila en los Procesadores IA-32 e Intel©64

Alejandro Furfaro
Ilustraciones de David Gonzalez Marquez (tnx a lot)

Abril 2012

# Agenda


1. Funcionamiento Básico
2. Ejemplos de uso de pila
   - ¿Como funciona un llamado Near?
   - ¿Como funciona un llamado Far?
   - Interrupciones
3. Convención de llamadas C
   - Modo 32 bits
   - Modo 64 bits
   - Resultados
4. Bibliografía

# Funcionamiento básico

## Pila

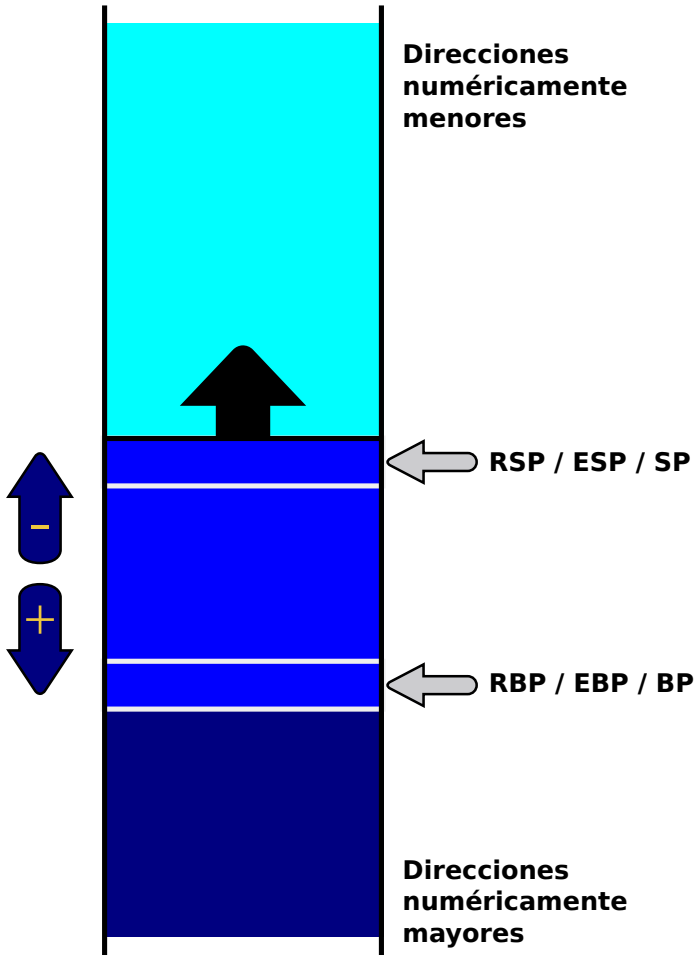



Figura: Funcionamiento Básico de la Pila.©David

- La pila (stack) es un área de memoria contigua, referenciada por un segmento cuyo selector está siempre en el registro SS del procesador.
- El tamaño de este segmento en el modo IA-32, puede llegar hasta 4 Gbytes de memoria, en especial cuando el sistema operativo utiliza el modelo de segmentación Flat (como veremos en clases subsiguientes).
- El segmento se recorre mediante un registro de propósito general, denominado habitualmente en forma genérica stack pointer, y que en estos procesadores según el modo de trabajo es el registro SP, ESP, o RSP (16, 32, o 64 bits respectivamente).

# Funcionamiento básico

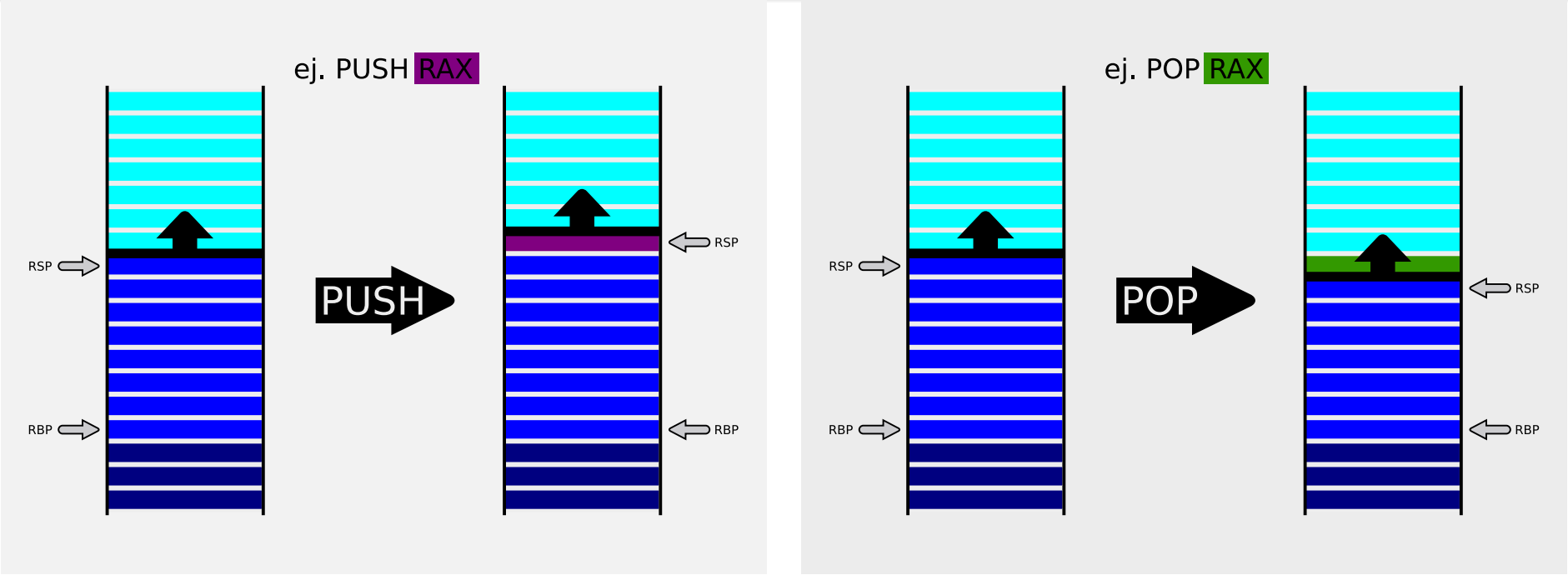



Figura: PUSH y POP.©David

- Para guardar un dato en el stack el procesador tiene la instrucción PUSH, y para retirarlo, la instrucción POP.
- Cada vez que ejecuta PUSH, el procesador decrementa el stack pointer (SP, ESP, o RSP) y luego escribe el dato en el stack, en la dirección apuntada por el registro de segmento SS, y el stack pointer correspondiente al modo de trabajo.
- Cada vez que ejecuta un POP, el procesador lee el ítem apuntado por el par SS : stack pointer, y luego incrementa éste último registro.

# Primeras conclusiones

El stack es un segmento expand down, ya que a medida que lo utilizamos (PUSH) su registro de desplazamiento se decrementa apuntando a las direcciones mas bajas (down) de memoria, es decir a aquellas numéricamente menores.

# Cuando se utiliza el stack

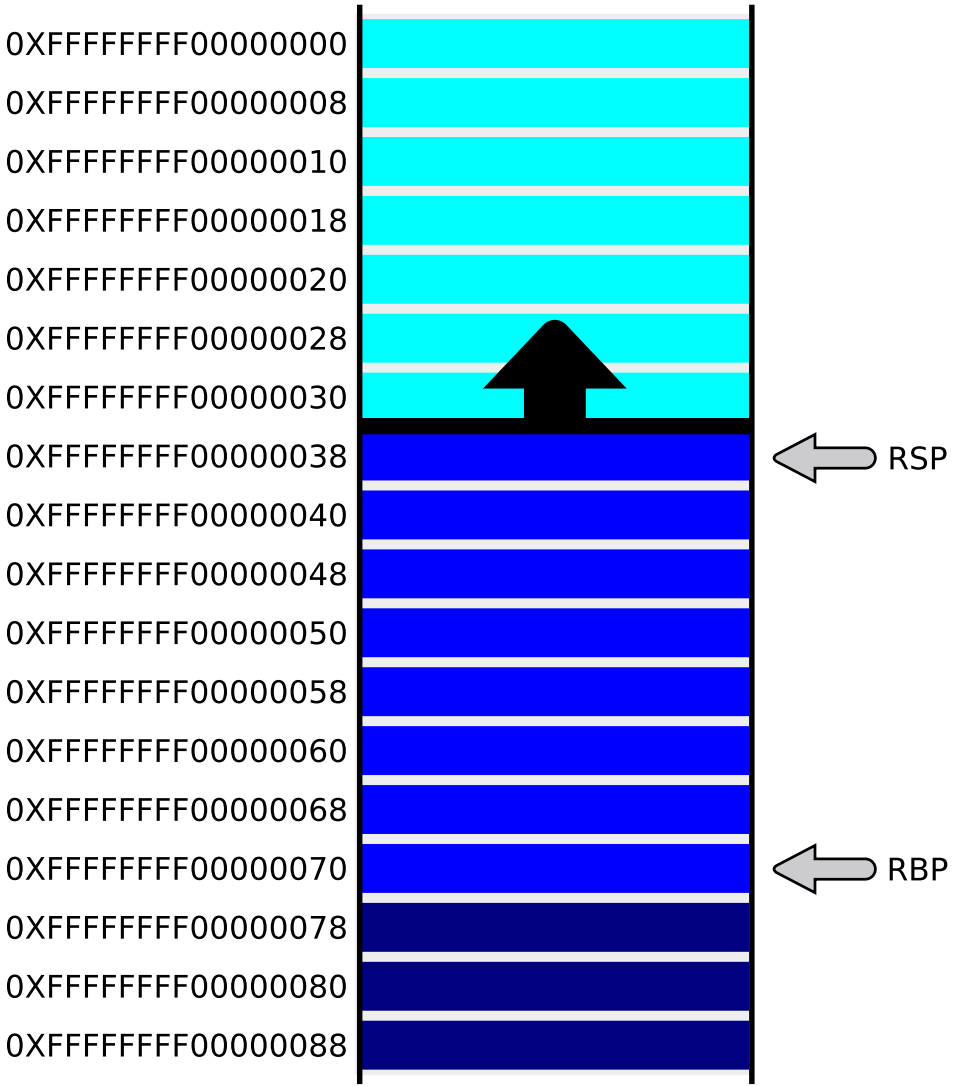
Las operaciones de pila se pueden realizar en cualquier momento, pero hablando mas generalmente, podemos afirmar que la pila se usa cuando:

- Cuando llamamos a una subrutina desde un programa en Assembler, mediante la instrucción CALL.
- Cuando el hardware mediante la interfaz adecuada envía una Interrupción al Procesador.
- Cuando desde una aplicación, ejecutamos una Interrupción de software mediante la instrucción INT type.
- Cuando desde un lenguaje como el C se invoca a una función cualquiera.

# Alineación del Stack

- El stack pointer debe apuntar a direcciones de memoria alineadas de acuerdo con su ancho de bits.
- Por ejemplo, el ESP (32 bits) debe estar alineado a double words.
- Al definir un stack en memoria se debe cuidar el detalle de la alineación.
- El tamaño de cada elemento de la pila se corresponde con el atributo de tamaño del segmento (16, 32, o 64 bits), es decir, con el modo de trabajo en el que está el procesador, y no con el del operando en sí.
- Ej: PUSH AL, consume 16, 32, o 64 bits dependiendo del tamaño del segmento. Nunca consume 8 bits.
- El valor en que se decrementa el Stack Pointer se corresponde con el tamaño del segmento (2, 4, u 8 bytes).

# Alineación del Stack




**Alineación**

4 bytes (en 32bits)

~~8 bytes~~ (en 64bits)

16 bytes

**Instrucciones**

- PUSH
- POP

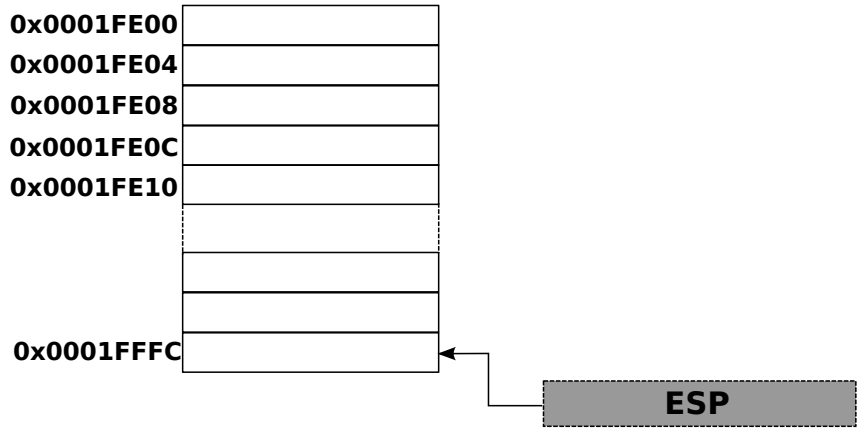
**Registros**

- RSP
- RBP

Figura: Alineación según el modo de trabajo.©David

# Como en un debuger :)

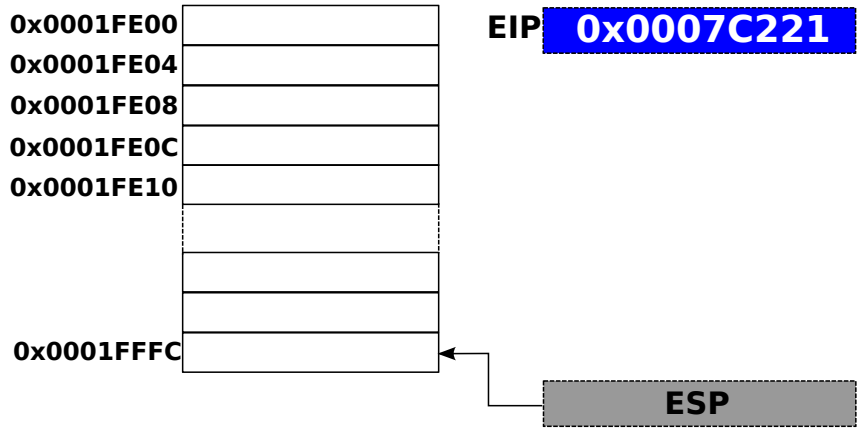
- Ejecutamos la primer instrucción
- Lee el port de E/S
- Y luego.....




```
1  %define mask 0xfff0
2  main:
3     ...
4     ...
5   in ax,0x300 ;lee port
6     call setmask ;llama a subrutina para aplicar una mascara
7     ...
8     ...
9  setmask:
10    and ax,mask    ;aplica la mascara
11    ret            ;retorna
```

# Estamos a punto de ejecutar CALL

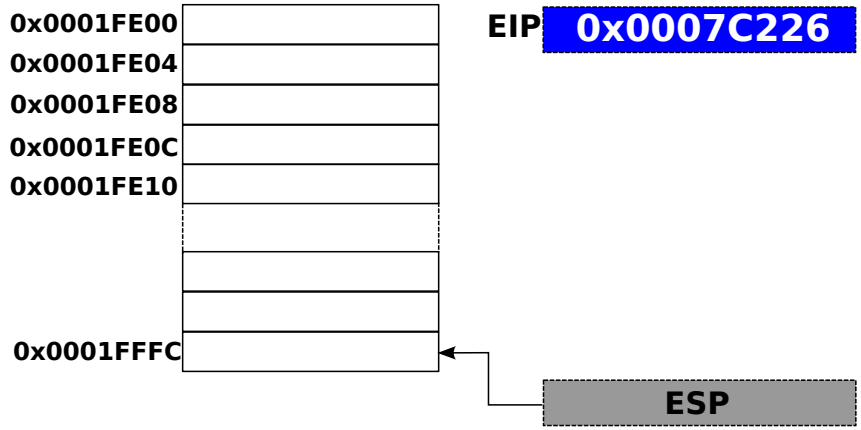
- ...ejecutamos la instrucción Call.
- La misma está almacenada a partir de la dirección de memoria contenida por **EIP**.
- El ESP apunta a la base de la pila.




```
1  %define mask 0xfff0
2  main:
3     ...
4     ...
5  in    ax,0x300 ;lee port
6  call setmask ;llama a subrutina para aplicar una mascara
7     ...
8     ...
9  setmask:
10   and ax,mask    ;aplica la mascara
11   ret            ;retorna
```

# CALL por dentro...

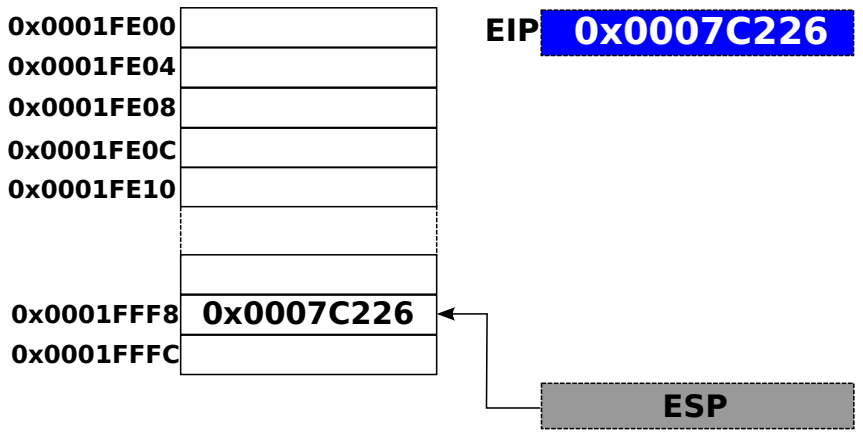
- En primer lugar el procesador apunta con EIP a la siguiente instrucción.
- Un CALL near se compone de 1 byte de código de operación y cuatro bytes para la dirección efectiva (offset), ya que estamos en 32 bits.
- Por eso el EIP apunta 5 bytes mas adelante, ya que allí comienza la siguiente instrucción del CALL.




```
1  %define mask 0xfff0
2  main:
3     ...
4     ...
5     in    ax,0x300 ;lee port
6     call  setmask  ;llama a subrutina para aplicar una mascara
7     ...
8     ...
9  setmask:
10    and ax,mask    ;aplica la mascara
11    ret            ;retorna
```

# CALL por dentro...

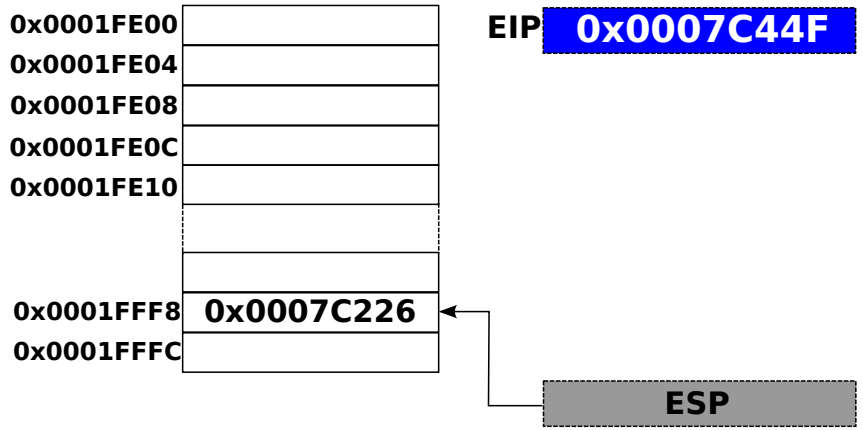
- El procesador decrementa ESP y guarda el valor de EIP.
- Así resguarda su dirección de retorno a la instrucción siguiente a CALL.
- Para saber a donde debe saltar saca de la instrucción CALL la dirección efectiva de la subrutina *setmask*.
- En nuestro caso 0x0007C44F.




```
1  %define mask 0xfff0
2  main:
3     ...
4     ...
5     in    ax,0x300 ;lee port
6     call  setmask  ;llama a subrutina para aplicar una mascara
7     ...
8     ...
9  setmask:
10    and ax,mask    ;aplica la mascara
11    ret            ;retorna
```

# Resultado del CALL

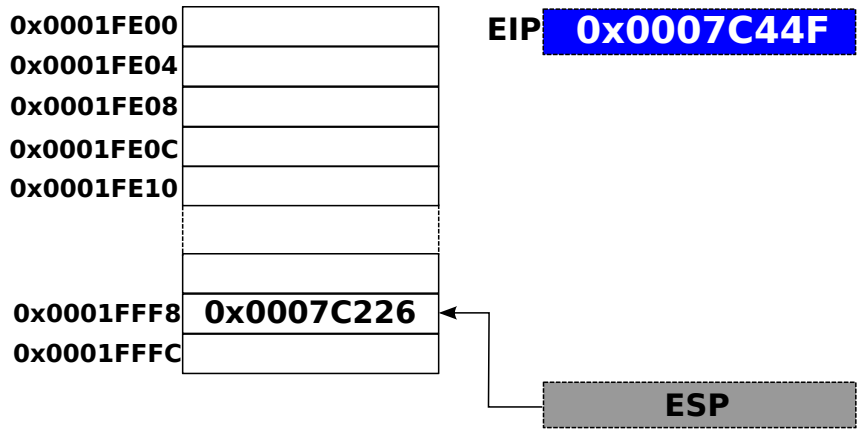
- Como resultado el valor de EIP, es reemplazado por la dirección efectiva de la subrutina *setmask*.
- Y sin mas.... el procesador está buscando la primer instrucción de la subrutina *setmask*, en este caso, la operación and.




```
1  %define mask 0xfff0
2  main:
3     ...
4     ...
5     in    ax,0x300 ;lee port
6     call  setmask  ;llama a subrutina para aplicar una mascara
7     ...
8     ...
9  setmask:
10    and ax,mask ;aplica la mascara
11    ret           ;retorna
```

# Volver.....

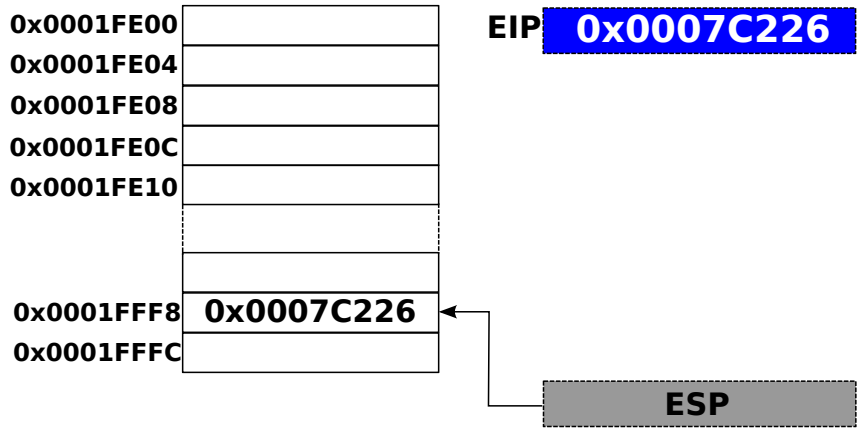
- Esta subrutina es trivial a los efectos del ejemplo.
- Para volver (sin la frente marchita)...
- Es necesario **ret**ornar




```
1  %define mask 0xfff0
2  main:
3     ...
4     ...
5     in    ax,0x300 ;lee port
6     call  setmask  ;llama a subrutina para aplicar una mascara
7     ...
8     ...
9  setmask:
10    and ax,mask    ;aplica la mascara
11    ret ;retorna
```

# Volviendo.....

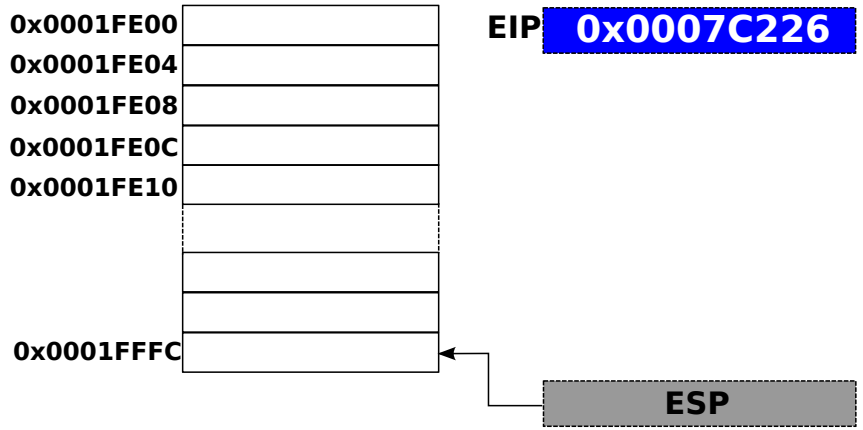
- La ejecución de **ret** consiste en recuperar de la pila la dirección de retorno.
- Esa dirección se debe cargar en EIP
- Una vez hecho.....




```
1  %define mask 0xfff0
2  main:
3     ...
4     ...
5     in    ax,0x300 ;lee port
6     call  setmask  ;llama a subrutina para aplicar una mascara
7     ...
8     ...
9  setmask:
10    and ax,mask    ;aplica la mascara
11    ret            ;retorna
```

# Volvimos!

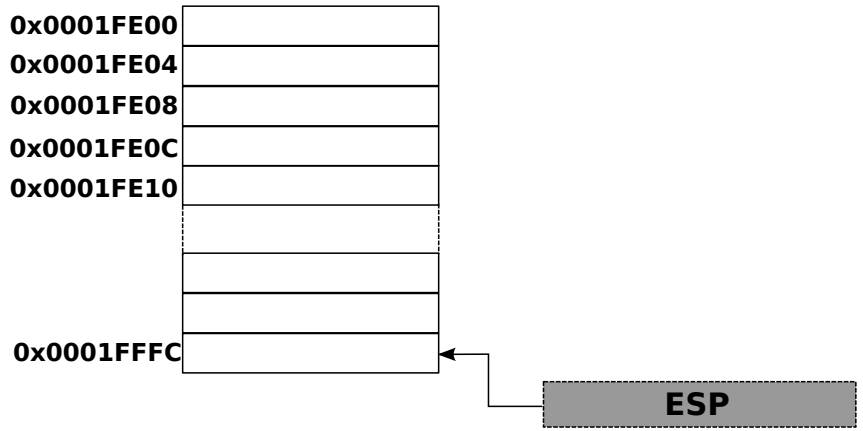
- Finalizada la ejecución de **ret** estamos otra vez en el código llamador.
- Pero en la instrucción siguiente a CALL




```
1  %define mask 0xfff0
2  main:
3     ...
4     ...
5     in    ax,0x300 ;lee port
6     call  setmask  ;llama a subrutina para aplicar una mascara
7  ...
8     ...
9  setmask:
10    and ax,mask    ;aplica la mascara
11    ret            ;retorna
```

# Ahora el destino está en un segmento diferente

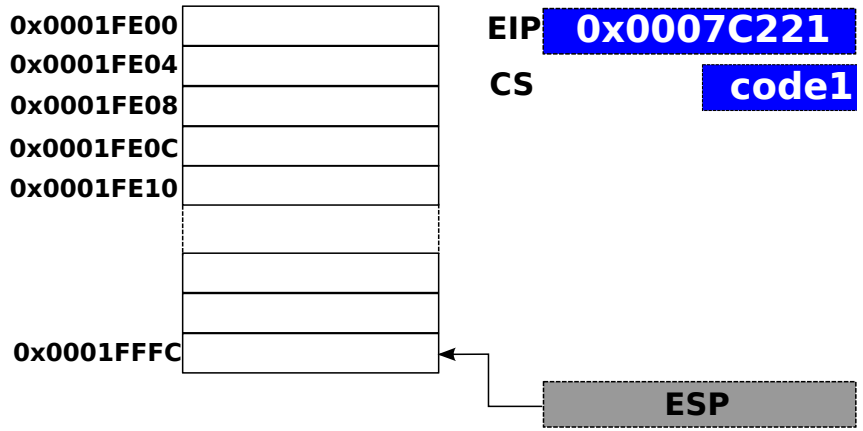
- Ejecutamos la primer instrucción
- Lee el port de E/S
- Y luego.....




```
1  %define mask 0xfff0
2  section code1
3  main:
4    ...
5   in ax,0x300 ;lee port
6    call code2:setmask ;llama a aplicar m scara
7    ...
8    ...
9  section code2
10 setmask:
11   and ax,mask ;aplica la m scara
12   retf ;retorna
```

# Por lo tanto importa el valor de CS...

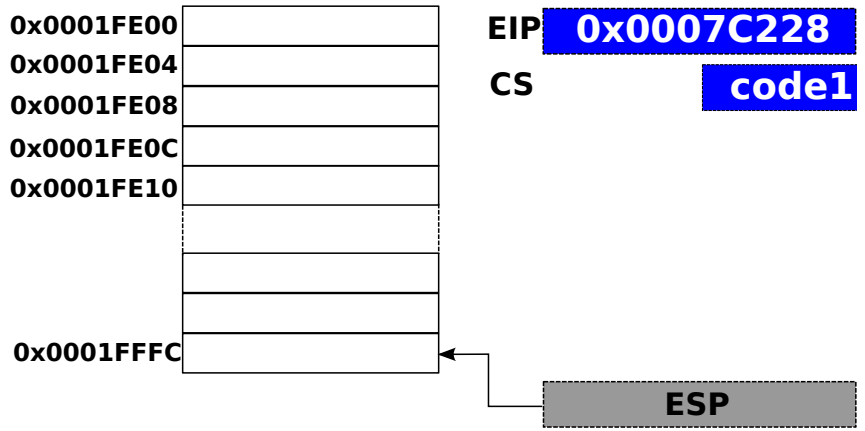
- Nuevamente nos paramos en el CALL
- Pero ahora necesitamos memorizar EIP, y también CS
- Ya que al estar el destino en otro segmento CS se modificará




```
1  %define mask 0xfff0
2  section code1
3  main:
4     ...
5  in    ax,0x300 ;lee port
6  call code2:setmask ;llama a subrutina para aplicar una mascara
7     ...
8     ...
9  section code2
10 setmask:
11    and ax,mask ;aplica la mscara
12    retf ;retorna
```

# Otra vez adentro del CALL... pero FAR

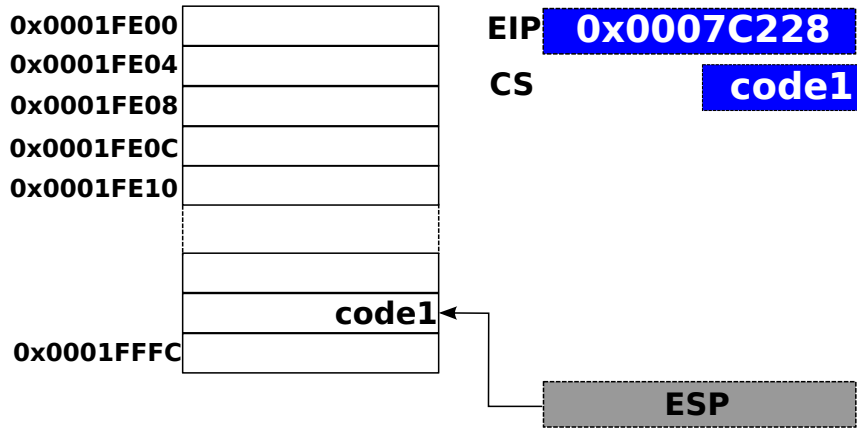
- Ahora la instrucción mide 7 bytes ya que se agrega el segmento
- Por lo tanto el EIP se incrementa 7 lugares
- Y se memoriza en la pila la dirección FAR.




```
1  %define mask 0xfff0
2  section code1
3  main:
4     ...
5     in    ax,0x300 ;lee port
6     call  code2:setmask ;llama a subrutina para aplicar una mascara
7     ...
8     ...
9  section code2
10 setmask:
11    and ax,mask ;aplica la mscara
12    retf ;retorna
```

# Otra vez adentro del CALL... pero FAR

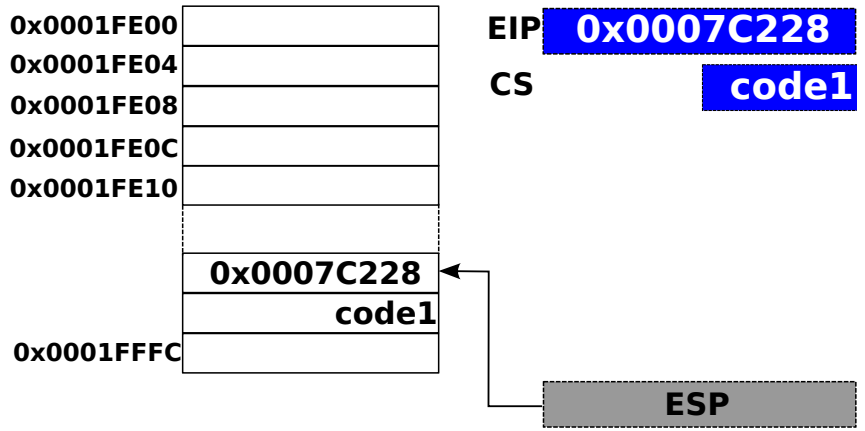
- En primer lugar guarda en la pila, el valor del segmento al cual debe retornar.
- Siempre antes de almacenar nada en la pila, debe antes decrementar el valor del ESP.




```
1  %define mask 0xfff0
2  section code1
3  main:
4     ...
5     in    ax,0x300 ;lee port
6     call code2:setmask ;llama a subrutina para aplicar una mascara
7     ...
8     ...
9  section code2
10 setmask:
11    and ax,mask ;aplica la mscara
12    retf ;retorna
```

# Otra vez adentro del CALL... pero FAR

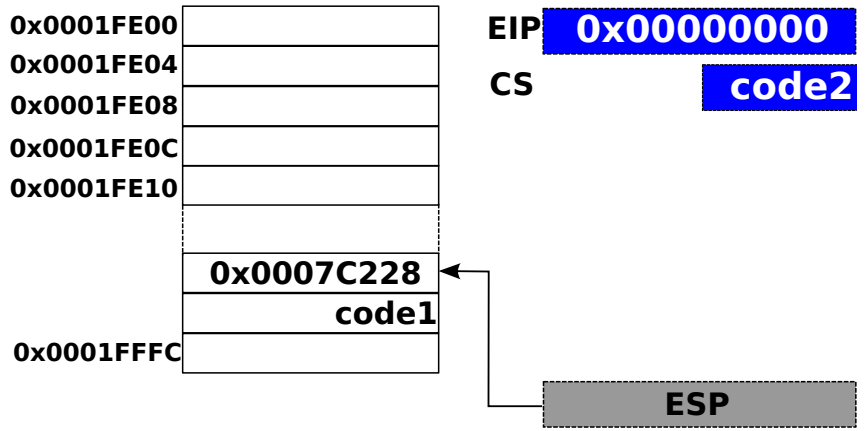
- Luego del valor del segmento guarda en la pila, el valor de EIP al cual debe retornar, y que lo llevará a buscar la siguiente instrucción al CALL.
- Decrementará nuevamente el valor del ESP, antes de almacenar.




```
1  %define mask 0xfff0
2  section code1
3  main:
4     ...
5     in    ax,0x300 ;lee port
6     call code2:setmask ;llama a subrutina para aplicar una mascara
7     ...
8     ...
9  section code2
10 setmask:
11    and ax,mask ;aplica la mscara
12    retf ;retorna
```

# Otra vez adentro del CALL... pero FAR

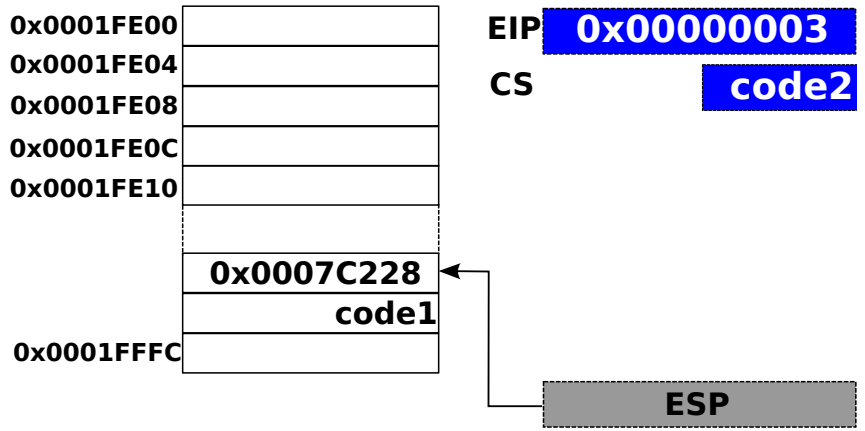
- La dirección de la rutina *setmask*, ahora es code2:offset.
- Como comienza justo al inicio del segmento, su offset es 0x00000000.




```
1  %define mask 0xfff0
2  section code1
3  main:
4     ...
5     in    ax,0x300 ;lee port
6     call  code2:setmask ;llama a subrutina para aplicar una mascara
7     ...
8     ...
9  section code2
10 setmask:
11    and ax,mask ;aplica la máscara
12    retf  ;retorna
```

# Retornando de un Call Far

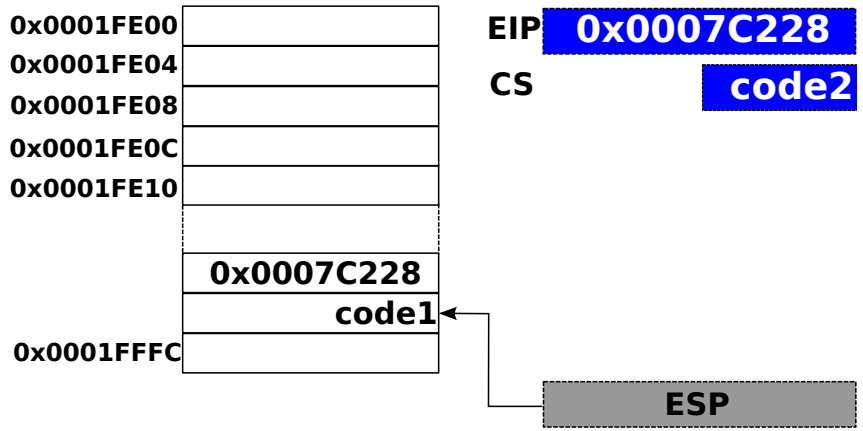
- Para volver de un call far hay que sacar de la pila no solo el offset sino también el segmento.
- Entonces no sirve la misma instrucción que se usa para volver de una rutina Near.




```
1  %define mask 0xfff0
2  section code1
3  main:
4    ...
5    in    ax,0x300 ;lee port
6    call  code2:setmask ;llama a subrutina para aplicar una mascara
7    ...
8    ...
9  section code2
10 setmask:
11   and ax,mask ;aplica la mscara
12 retf ;retorna
```

# Retornando de un Call Far

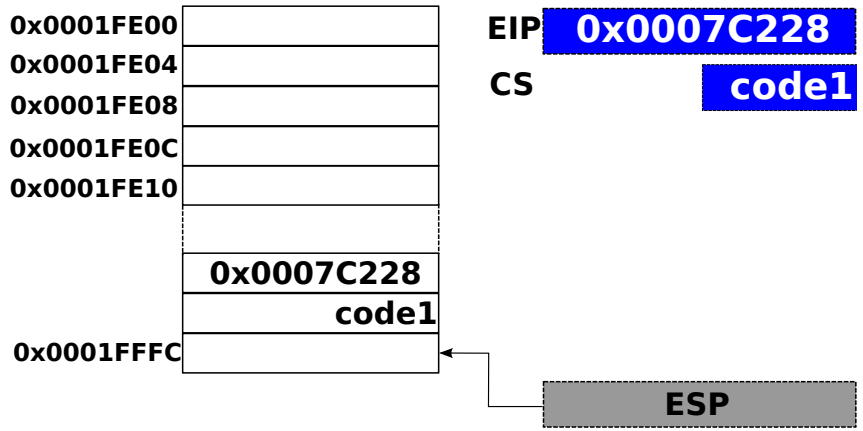
- Recupera la dirección efectiva
- Luego decrementa el Stack Pointer




```
1  %define mask 0xfff0
2  section code1
3  main:
4     ...
5     in    ax,0x300 ;lee port
6     call code2:setmask ;llama a subrutina para aplicar una mascara
7     ...
8     ...
9  section code2
10 setmask:
11    and ax,mask ;aplica la mscara
12    retf ;retorna
```

# Retornando de un Call Far

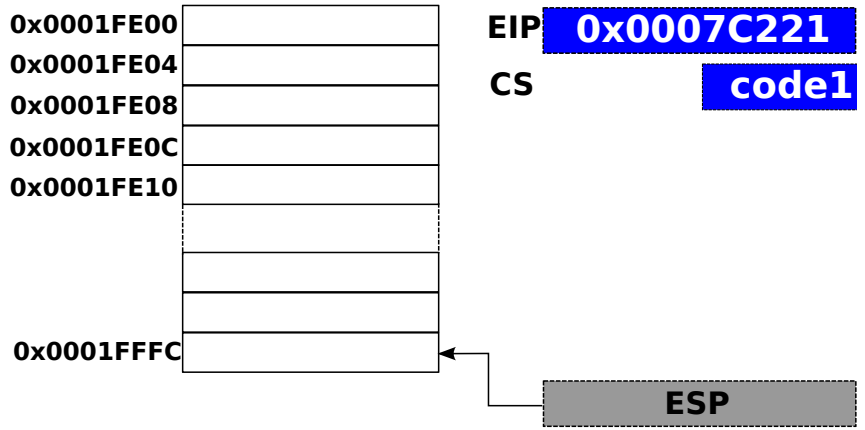
- Recupera el valor del segmento
- Luego decrementa el Stack Pointer
- ...y volvió...




```
1  %define mask 0xfff0
2  section code1
3  main:
4     ...
5     in    ax,0x300 ;lee port
6     call code2:setmask ;llama a subrutina para aplicar una mascara
7  ...
8     ...
9  section code2
10 setmask:
11    and ax,mask ;aplica la mscara
12    retf ;retorna
```

# ¿Que pasa cuando se produce una Interrupción?

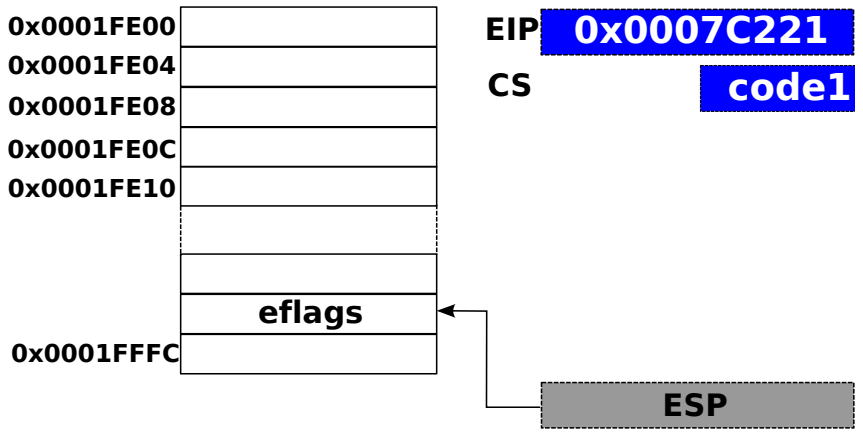
- Ejecutamos una instrucción cualquiera
- y en el medio de esa instrucción se produce una interrupción
- ...




```
1  section code
2  main:
3     ...
4  next:
5   test [var],1 ;chequea bit 0 de variable
6     jnz next
7     ...
8  section kernel
9  handler_int:
10    in al,port ;lee port de E/S
11    iret ;retorna
```

# ¿Que pasa cuando se produce una Interrupción?

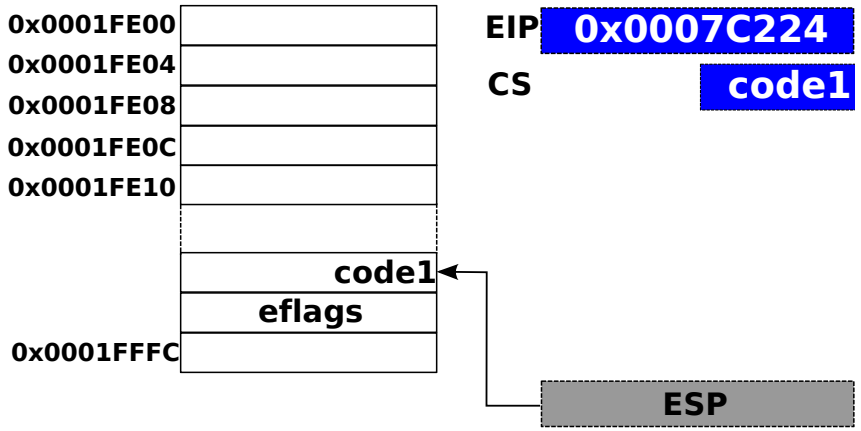
- Es necesario guardar además de la dirección de retorno, el estado del procesador.
- De otro modo si al final de la interrupción alguna instrucción modifica un flag, el estado de la máquina se altera y le vuelve al programa modificado.
- Esto puede tener resultados impredecibles si al retorno hay que usar el flag que cambió.




```
1  section code
2  main:
3    ...
4  next:
5    test [var],1 ;chequea bit 0 de variable
6    jnz next
7    ...
8  section kernel
9  handler_int:
10   in al,port ;lee port de E/S
11   iret ;retorna
```

# ¿Que pasa cuando se produce una Interrupción?

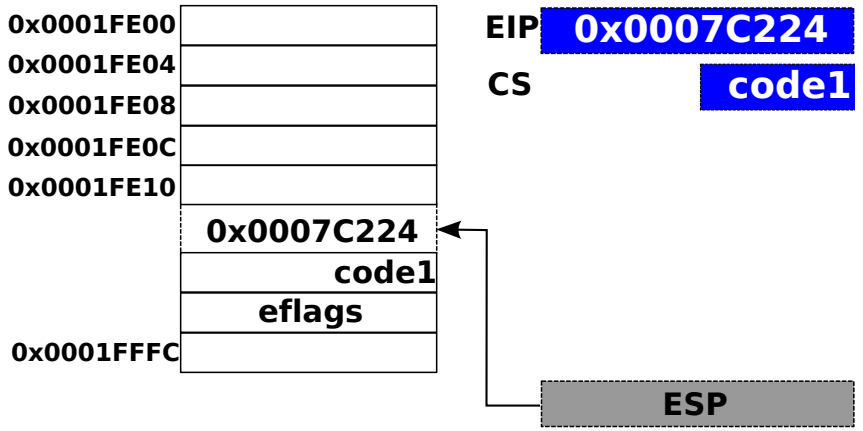
- La dirección de retorno es far.
- Especialmente en sistemas multitasking donde cada proceso tiene una pila de kernel diferente.
- Así que luego de los flags se guarda el segmento de código.




```
1  section code
2  main:
3    ...
4  next:
5    test [var],1 ;chequea bit 0 de variable
6    jnz next
7    ...
8  section kernel
9  handler_int:
10   in al,port ;lee port de E/S
11   iret ;retorna
```

# ¿Que pasa cuando se produce una Interrupción?

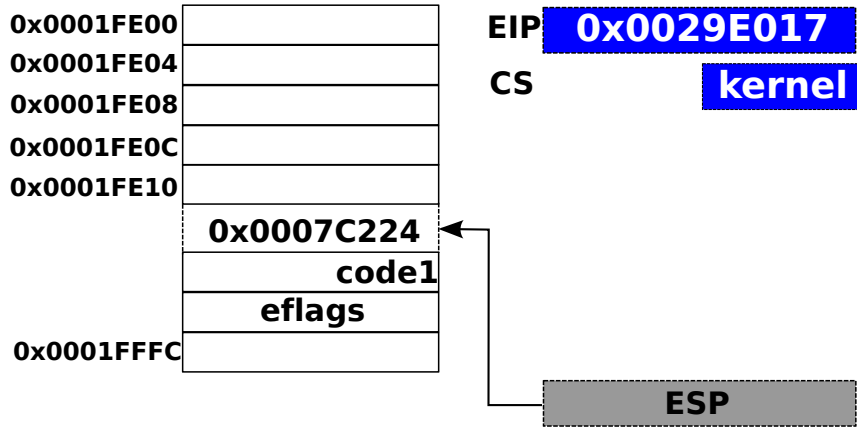
- Se resguarda finalmente la dirección efectiva
- Notar que es la de la instrucción siguiente a la de la interrupción




```
1  section code
2  main:
3    ...
4  next:
5    test [var],1 ;chequea bit 0 de variable
6    jnz next
7    ...
8  section kernel
9  handler_int:
10   in al,port ;lee port de E/S
11   iret ;retorna
```

# ¿Que pasa cuando se produce una Interrupción?

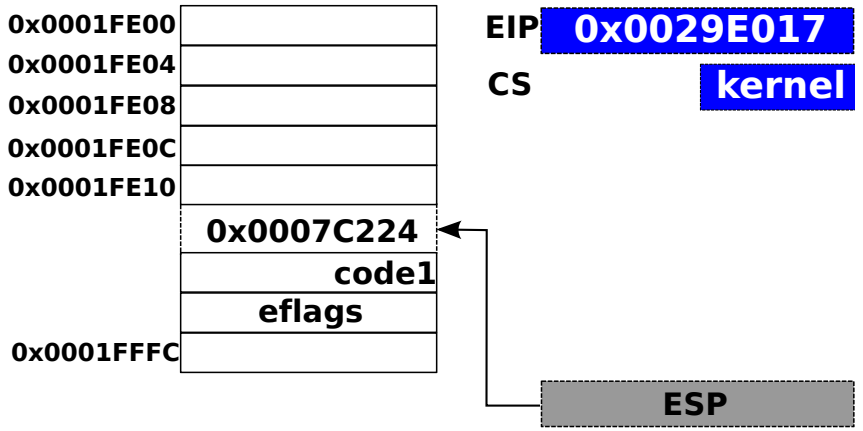
- Los nuevos valores de segmento y desplazamiento que debe cargar en CS:EIP, los obtiene del vector de interrupciones en modo real, o de la Tabla de descriptores de interrupción en modo protegido, o en el modo 64 bits.




```
1  section code
2  main:
3     ...
4  next:
5     test [var],1 ;chequea bit 0 de variable
6     jnz next
7     ...
8  section kernel
9  handler_int:
10    in al,port ;lee port de E/S
11    iret ;retorna
```

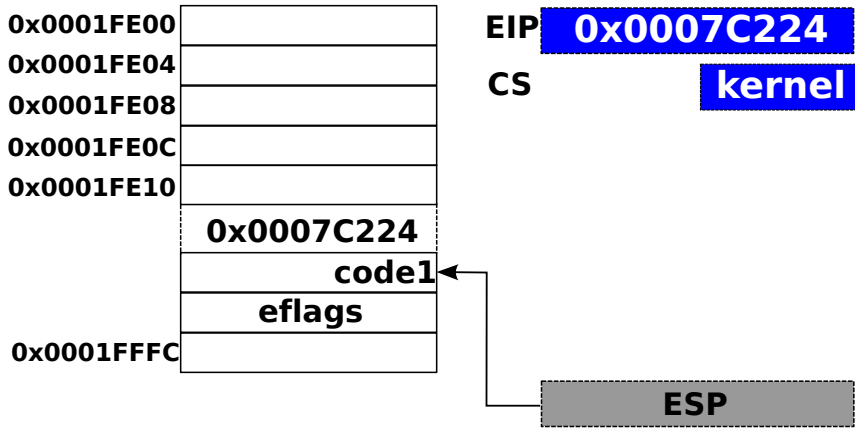
# ¿Como se vuelve de una Interrupción?

- Recuperando además de la dirección de retorno, los flags
- Por lo tanto necesitamos otra instrucción particular de retorno...
- ... **iret**...



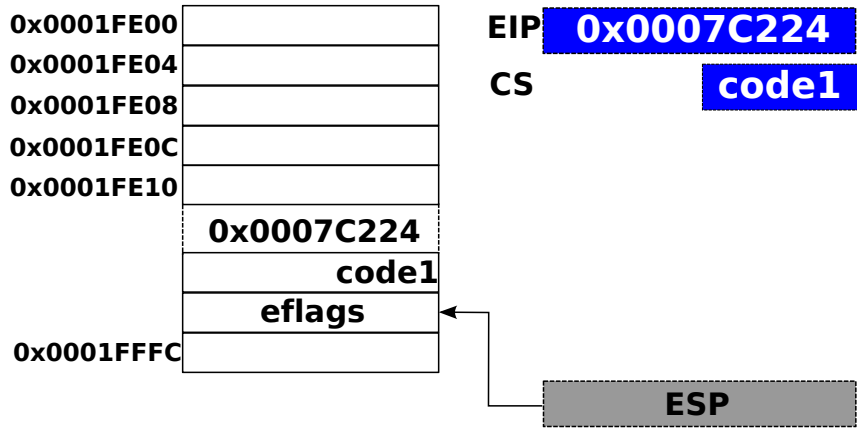

```
1  section code
2  main:
3    ...
4  next:
5    test [var],1 ;chequea bit 0 de variable
6    jnz next
7    ...
8  section kernel
9  handler_int:
10   in al,port ;lee port de E/S
11 iret ;retorna
```

# Volviendo...



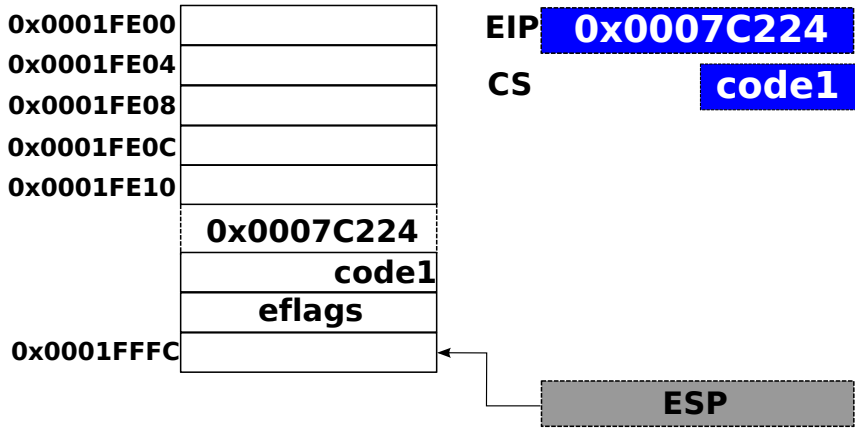

```
1  section code
2  main:
3    ...
4  next:
5    test [var],1 ;chequea bit 0 de variable
6    jnz next
7    ...
8  section kernel
9  handler_int:
10   in al,port ;lee port de E/S
11   iret ;retorna
```

# Volviendo...




```
1  section code
2  main:
3    ...
4  next:
5    test [var],1 ;chequea bit 0 de variable
6    jnz next
7    ...
8  section kernel
9  handler_int:
10   in al,port ;lee port de E/S
11   iret ;retorna
```

# Volviendo...




```
1  section code
2  main:
3    ...
4  next:
5    test [var],1 ;chequea bit 0 de variable
6  jnz next
7    ...
8  section kernel
9  handler_int:
10   in al,port ;lee port de E/S
11   iret ;retorna
```

# Llamadas a función

- En general en el lenguaje C una función se invoca de la siguiente forma

```
1 type function (arg1, arg2,..., argn);
```

- ***type***, es siempre un tipo de dato básico (int, char, float, double), o un puntero, o void en caso en que no devuelva nada.
- El manejo de la interfaz entre el programa invocante y la función llamada la resuelve el compilador, de una manera perfectamente definida.
- Sin embargo los pormenores son diferentes según se trabaje en 32 bits o en 64 bits

# Stack Frame

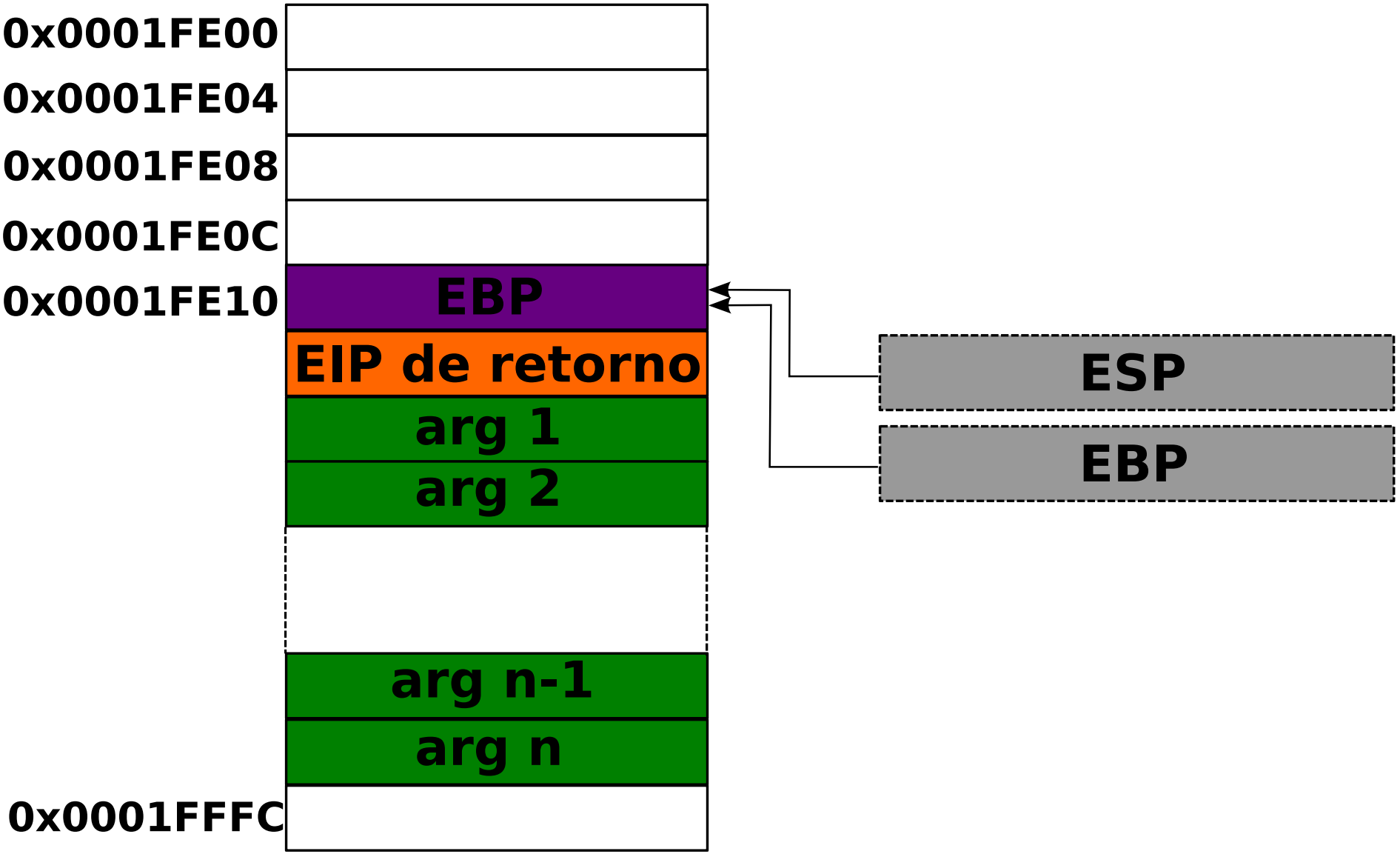
- El compilador traduce el llamado en el siguiente código assembler:

```
1  push argn
2  ...
3  push arg2
4  push arg1
5  call function        ; o sea un CALL Near!!
```

- Los argumentos se apilan desde la derecha hacia la izquierda.
- Una vez dentro de la subrutina “function”, el compilador agrega el siguiente código

```
1  push   ebp           ;resguardamos ebp
2  mov    ebp,esp       ;lo apuntamos a la pila
```

# Stack Frame Resultante



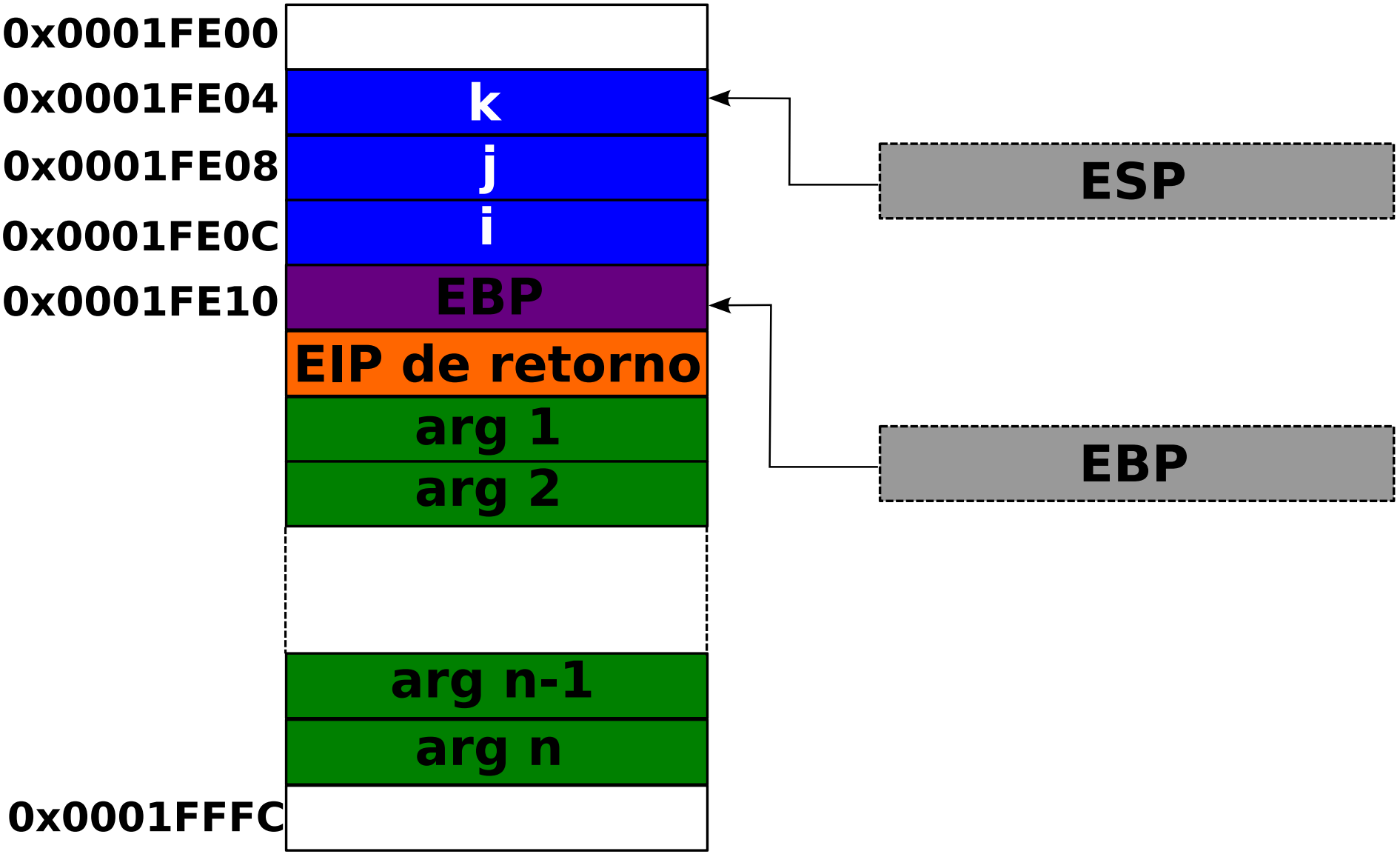
# Variables Locales

- Una vez dentro de la función invocada en un programa C utilizamos por lo general variables locales. Solo tienen validez dentro de la función en la que se las declara.
- Una vez finalizada esta función no existen mas.
- Se crean frames en el stack para albergar dichas variables. Simplemente moviendo esp hacia el fondo del stack, es decir:

```
1 sub esp, n
```

- Siendo n la cantidad de bytes a reservar para variables

# Stack Frame con Variables Locales



# Acceso a argumentos y variables locales

```
1   push   ebp
2   mov    ebp,esp
3   sub    esp, 12                    ;Frame para i,j, y k
4
5   mov    eax,dword [ebp+0x10]       ;pone en eax, el arg3
6   test   dword [ebp+0x0C],3         ;bits 0 y 1 de arg2 deben
7                                     ;ser '0'
8   add    esi, dword [ebp+8]         ;agrego a esi el arg1
9
10  inc    dword [ebp-4]              ;i++
11  dec    dword [ebp-8]              ;j--
12  add    dword [ebp-0xC],size       ;k apunta al siguiente
13                                    ;elemento de tama o
14                                    ;= size de un arreglo
```

# System V Application Binary Interface

- Conocida como ABI, establece el pasaje de argumentos desde una función llamante a una función llamada, y como se retornan los resultados.

**En 64bits:**

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Entero o Puntero | RDI | RSI | RDX | RCX | R8 | R9 | | [RBP+16] | [RBP+24] | [RBP+32] | [RBP+40] | [RBP+48] |
| Flotante | XMM0 | XMM1 | XMM2 | XMM3 | XMM4 | XMM5 | XMM6 | XMM7 | | | | |

- Los registros se usan en orden dependiendo del tipo
- Los registros de enteros guardan parámetros de tipo Entero o Puntero
- Los registros XMM guardan parámetros de tipo Flotante
- Si no hay más registros disponibles se usa la PILA
- Los parámetros en la PILA deben quedar ordenados desde la dirección más baja a la más alta.

Figura: Convención C para 64 bits (©David)

# Ejemplo sencillo

En 64 bits:

```
int f1( int a, float b, double c, int* d, double* e)
```

**Enteros**

RDI = a
RSI = d
RDX = e
RCX =
R8 =
R9 =

**Flotante**

XMM0 = b
XMM1 = c
XMM2 =
XMM3 =
XMM4 =
XMM5 =
xMM6 =
xMM7 =

**Pila**

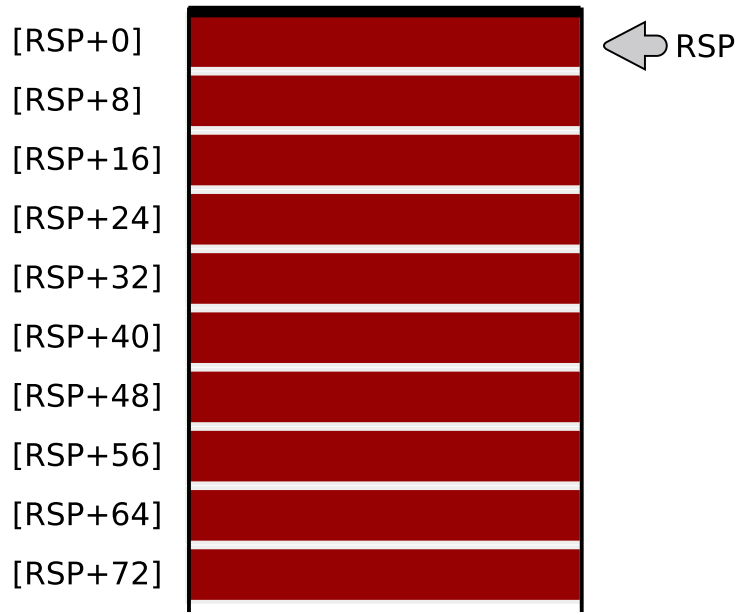



Figura: Resolución de llamada - Ej1 (©David)

# Ejemplo sencillo

En 64 bits:

```
int f2( int a1, float a2, double a3, int a4, float a5,
        double a6, int* a7, double* a8, int* a9,
        double a10, int** a11, float* a12, double** a13,
        int* 14, double a15)
```

**Enteros**

```
RDI = a1
RSI = a4
RDX = a7
RCX = a8
R8  = a9
R9  = a11
```

**Flotante**

```
XMM0 = a2
XMM1 = a3
XMM2 = a5
XMM3 = a6
XMM4 = a10
XMM5 = a15
xMM6 =
xMM7 =
```

**Pila**

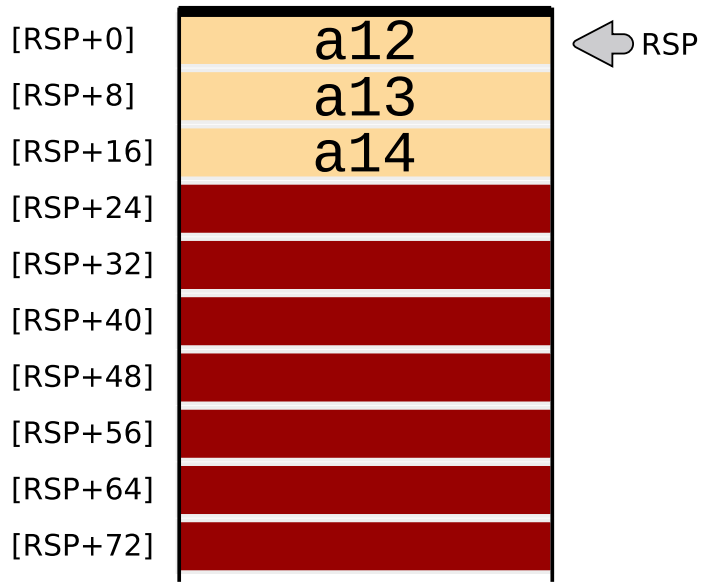



Figura: Resolución de llamada - Ej2 (©David)

# Resultados

- En 32 bits los resultados enteros y punteros se devuelven por eax.
- En 64 bits los enteros y punteros se devuelven en RAX, y si son floats o doubles, en XMM0 y/o XMM1.

- Intel® 64 and IA-32 Architectures Software Developer's Manual: Vol I. Basic Architecture.

  Capítulo 6

- System V Application Binary Interface. AMD64 Architecture Processor Supplement. Draft Version 0.99.5. Edited by Michael Matz, Jan Hubička, Andreas Jaeger, Mark Mitchell. September 3, 2010.

- System V Application Binary Interface. Intel386™ Architecture Processor Supplement. Fourth Edition