Diseño Basado en Microprocesadores
Interfaz entre ensamblador y C
en la familia x86-32/x86-64

Índice

---

1. Introducción

En la actualidad, es poco frecuente que una aplicación se desarrolle enteramente en lenguaje ensamblador. Sí es más frecuente la incorporación de funciones escritas en ensamblador dentro de aplicaciones desarrolladas en su mayor parte en algún lenguaje de alto nivel tal como C o C++. La utilización del ensamblador es la única alternativa cuando

quiere exprimirse la velocidad del microprocesador hasta el límite o explotar las características incorporadas en los miembros más novedosos de la familia x86.

Para conseguir incorporar código en ensamblador en programas de alto nivel es necesario conocer, entre otras cosas, el método que el código generado por el compilador utiliza para hacer llegar a las funciones o procedimientos los valores de los argumentos así como el lugar de devolución de valores. Al sistema de paso de argumentos a las funciones que usa un compilador de lenguaje de alto nivel se le denomina convenio de llamada. Aunque parecidos en los aspectos fundamentales, los convenios de llamada empleados por los distintos lenguajes presentan ligeras diferencias que es necesario tener en cuenta para realizar el interfaz entre éstos y ensamblador de forma exitosa.

Como ocurre con la mayoría de los microprocesadores de tipo CISC, y por lo común con aquellos microprocesadores que cuentan con un reducido número de registros internos, los compiladores de 32 bits de lenguajes de alto nivel para la familia x86 utilizan generalmente la pila para realizar el paso de argumentos a los procedimientos así como para ubicar las variables locales que éstos utilizan. Este sistema es lento puesto que supone un importante tráfico de datos entre el microprocesador y la memoria. En los microprocesadores de tipo RISC, donde la unidad de control más pequeña deja espacio para un juego de registros mucho más amplio, el paso de argumentos y ubicación de variables locales suele emplear los registros internos con el consiguiente aumento de velocidad que esto conlleva. No obstante, la utilización de memorias caché en sistemas CISC tiende a reducir la desventaja. Algunos compiladores de 32 bits de lenguajes de alto nivel para la familia x86 también tienen la opción de pasar los argumentos a través de los registros.

Los compiladores para los microprocesadores de 64 bits de la familia x86 pasan los argumentos también a través de los registros ya que estos procesadores disponen de 16 registros de propósito general de 64 bits cuando funcionan en el modo de 64 bits.

2. Tamaño de los tipos estándar en diferentes compiladores

En este apartado se indica el tamaño de los distintos tipos de datos estándar de C en compiladores de 32 y 64 bits.

2.1. Compiladores GCC y Microsoft de 32 bits

En un compilador de 32 bits los tamaños de los distintos tipos estándar del C son los que se indican en la tabla 1. Puede verse que tanto el tipo `int` como el `long` y sus versiones sin signo tienen todos tamaño de 32 bits. El tipo `long long` y su versión sino signo tiene un tamaño de 64 bits.

Tabla 1: Tamaño de los tipos estándar en un compilador de 32 bits

Tipo	**Bits**	**Rango**
`int`	32	–2147483648 a 2147483647
`unsigned int`	32	0 a 4294967295
`char`	8	–128 a 127
`unsigned char`	8	0 a 255
`short`	16	–32768 a 32767
`unsigned short`	16	0 65535
`long`	32	–2147483648 a 2147483647
`unsigned long`	32	0 a 4294967295
`long long`	64	–9223372036854775808 a 9223372036854775807
`unsigned long long`	64	0 a 18446744073709551615
`float`	32	$\approx 3.4 \times 10^{\pm 38}$ (7 dígitos)
`double`	64	$\approx 1.7 \times 10^{\pm 308}$ (15 dígitos)
puntero	32	0 a 4294967295

2.2. Compilador GCC de 64 bits

Los tamaños de los tipos de datos estándar en el compilador GCC de 64 bits se indican en la tabla 2. Las diferencias respecto a un compilador de 32 bits es que el tipo `long` y su versión sin signo tienen un tamaño de 64 bits y los punteros son también cantidades de 64 bits.

Tabla 2: Tamaño de los tipos estándar en el compilador GCC de 64 bits

Tipo	**Bits**	**Rango**
`int`	32	–2147483648 a 2147483647
`unsigned int`	32	0 a 4294967295
`char`	8	–128 a 127
`unsigned char`	8	0 a 255
`short`	16	–32768 a 32767
`unsigned short`	16	0 65535
`long`	64	–9223372036854775808 a 9223372036854775807
`unsigned long`	64	0 a 18446744073709551615
`long long`	64	–9223372036854775808 a 9223372036854775807
`unsigned long long`	64	0 a 18446744073709551615
`float`	32	$\approx 3.4 \times 10^{\pm 38}$ (7 dígitos)
`double`	64	$\approx 1.7 \times 10^{\pm 308}$ (15 dígitos)
puntero	64	0 a 18446744073709551615

2.3. Compilador Microsoft de 64 bits

En la tabla 3 se muestran los tamaños de los tipos de datos estándar en un compilador Microsoft de 64 bits. Excepto los punteros, que tienen un tamaño de 64 bits en lugar de 32 bits, los tamaños coinciden con los de un compilador de 32 bits.

Tabla 3: Tamaño de los tipos estándar en el compilador Microsoft de 64 bits

Tipo	**Bits**	**Rango**
`int`	32	–2147483648 a 2147483647
`unsigned int`	32	0 a 4294967295
`char`	8	–128 a 127
`unsigned char`	8	0 a 255
`short`	16	–32768 a 32767
`unsigned short`	16	0 65535
`long`	32	–2147483648 a 2147483647
`unsigned long`	32	0 a 4294967295
`long long`	64	–9223372036854775808 a 9223372036854775807
`unsigned long long`	64	0 a 18446744073709551615
`float`	32	$\approx 3.4 \times 10^{\pm 38}$ (7 dígitos)
`double`	64	$\approx 1.7 \times 10^{\pm 308}$ (15 dígitos)
puntero	64	0 a 18446744073709551615

3. Convenio de llamada de un compilador de C de 32 bits

A continuación, se explicará el convenio de llamada del lenguaje C de los compiladores de 32 bits para los microprocesadores x86. Se considerará el caso de un compilador que genera módulos objeto en formato *elf*, tal como el compilador *gcc* para Linux pero se indicarán los cambios a realizar en caso de utilizar otros compiladores. Una buena forma de empezar es analizar el código que genera la llamada a una función escrita en C. Consideremos una función que reciba como argumentos dos enteros y devuelva asimismo un entero a través del nombre de la función. Como sólo se trata de poner al descubierto el mecanismo de llamada a la función, ésta hará una tarea muy sencilla, por ejemplo, devolver como resultado la resta de los parámetros de entrada. El prototipo de la función sería el siguiente:

```
int resta(int a, int b);
```

Queremos analizar también cómo se manejan las variables locales de una función. Por tanto, aunque realmente no sería preciso, la función `resta` declarará una variable local `r` a la que asignará el resultado de restar los parámetros `a` y `b` y luego retornará el valor de `r`. La función resta tendrá por tanto la siguiente definición:

```
int resta(int a, int b)
{
        int r;

        r = a - b;
        return r;
}
```

3.1. Código de llamada

Supongamos que la función `resta` se llama dentro de `main` para restar los valores de dos variables locales `x` e `y` y almacenar el resultado en otra variable local `z`.

```
int main(void)
{
        int x = 5, y = 3, z;
        ...
        z = resta(x, y);
        ...
}
```

En un compilador de 32 bits el tipo int tiene un tamaño de 32 bits, es decir, el tamaño de una doble palabra. El fragmento de código que el compilador genera para implementar la línea de código fuente

```
z = resta(x, y);
```

es el siguiente:

Listado 1: Código generado por el compilador para la línea `z = resta(x, y)`

```
...
; Código anterior a la llamada a resta
...
;Código generado por la línea: z = resta(x, y);
push    dword [ebp-8]  ; Introducir y (arg. b) en la pila.
push    dword [ebp-4]  ; Introducir x (arg. a) en la pila.
call    resta          ; Llamar a la función resta.
add     esp,8          ; Extraer los argumentos de la pila.
mov     [ebp-12],eax   ; Almacenar el valor devuelto en z.
...
; Código posterior a la llamada a resta.
...
```

El código de llamada consta de las siguientes partes:

1. Introducción de los valores de los argumentos en la pila (En este caso los valores de las variables `x` e `y` de la función `main`).
2. Llamada al procedimiento que implementa la funcionalidad de resta mediante un CALL.
3. Extracción de los argumentos de la pila para equilibrarla.
4. Acceso al valor devuelto por la función (en este caso, para almacenarlo en la variable `z`).

3.1.1. Introducción de los valores de los argumentos en la pila

Como se ha dicho antes, las funciones reciben sus argumentos a través de la pila y crean también en la pila el espacio para las variables locales. En la función llamadora, los valores de los argumentos se introducen en la pila mediante instrucciones PUSH. Dentro de una función, el registro puntero base EBP se mantiene apuntando a la zona de la pila ocupada por los argumentos y las variables locales. Por tanto, en el código creado por el compilador, las referencias a los argumentos y a las variables locales se realizan empleando el modo de direccionamiento indirecto relativo al registro EBP. Recordemos que cuando un modo de direccionamiento indirecto hace uso del registro EBP el procesador utiliza el registro SS para obtener la parte de segmento de la dirección. De esta forma, la utilización

de EBP dirige los accesos a la pila automáticamente. En este aspecto, la función `main` no se diferencia de las demás, es decir, dentro de main las variables `x`, `y` y `z` del ejemplo se referencian también con ayuda del registro EBP. La tabla 4 da la posición de cada una.

El porqué de esta asignación se comprenderá cuando se vea como la función `resta` maneja sus propios argumentos y variables locales. Lo importante en este momento es observar el orden en el que los valores de los argumentos se introducen en la pila durante la llamada a resta: primero se introduce en la pila el valor de la variable `y` (que está en SS:EBP–8) mediante `push dword [EBP-8]` y a continuación el valor de la variable `x` mediante `push dword [EBP-4]`. El orden en el que se introducen en la pila los argumentos de las funciones es una de las característica del convenio de llamada de un lenguaje. En concreto:

En el convenio de llamada del lenguaje C, los valores de los argumentos se introducen en la pila en orden inverso al que tienen estos en el prototipo de la función. Es decir, podemos conocer el orden en el que se introducen en la pila los valores para los argumentos leyendo el prototipo de la función de derecha a izquierda.

En un compilador de 32 bits, si un argumento tiene un tamaño de 8 bits (`char` o `unsigned char`) o 16 bits (`short int` o `unsigned short int`) se introduce en la pila una doble palabra en la que el byte bajo, en el primer caso, o la palabra baja, en el segundo, lleva el valor del argumento. Esto se debe, por una parte, a que una instrucción PUSH sólo pueden introducir en la pila una palabra o una doble palabra y, por otra, a la conveniencia de mantener alineada la pila.

3.1.2. Llamada a la función mediante CALL

Una vez introducidos los valores de los argumentos en la pila, se llama a la función mediante una instrucción CALL cercana. Una llamada cercana introduce en la pila el valor del registro EIP, es decir, el offset de la dirección de retorno.

3.1.3. Extracción de los argumentos de la pila

Una vez que la función `resta` ha llevado a cabo su trabajo, la instrucción RET con la que ésta termina devolverá el control a la función `main`. La instrucción RET extrae de la pila la dirección de retorno pero los argumentos continúan en la pila. Para evitar que la pila crezca sin parar a medida que se suceden las llamadas a las distintas funciones del programa, los argumentos deben retirarse de la pila liberando así el espacio que ocupan. En nuestro ejemplo en C, la función `resta` no ha hecho nada por retirar de la pila los argumentos que fueron puestos por allí por `main`, de forma que es la función llamadora (`main` en nuestro ejemplo) la que tiene que hacerlo. El que sea el procedimiento llamado o el llamador el encargado de retirar de la pila los argumentos es otra característica que distingue a los diferentes convenios de llamada. En el caso del lenguaje C:

Tabla 4: Ubicación de las variables `x`, `y` y `z` de `main`

Variable	Ubicación
x	doble palabra en SS:[EBP–4]
y	doble palabra en SS:[EBP–8]
z	doble palabra en SS:[EBP–12]

En el convenio de llamada del lenguaje C, la función llamadora es la encargada de retirar de la pila los argumentos pasados a la función llamada.

Para retirar de la pila los argumentos, lo más sencillo es sumar al puntero de pila ESP el número de posiciones de memoria que ocupan en la pila los argumentos pasados al procedimiento recién llamado. Esto tiene el efecto pretendido de restituir ESP a la antigua cima de la pila y es más rápido que emplear instrucciones POP (se requerirían tantos POPs como dobles palabras hubieran sido introducidas). No obstante, cuando los argumentos sólo ocupan una doble palabra en la pila el compilador puede generar una instrucción `pop reg32` siendo `reg32` un registro de 32 bits que esté libre en ese momento.

En otros lenguajes es la función llamada la encargada de equilibrar la pila, empleando habitualmente para ello una instrucción `ret inm16`. Esta instrucción extrae de la pila la dirección de retorno y vuelve al lugar de llamada igual que lo haría una instrucción RET pero, además, suma a ESP la constante inmediata de 16 bits `inm16`.

3.1.4. Acceso al valor devuelto por la función

Como puede intuirse analizando la última instrucción del código de llamada a `resta`, el valor de tipo `int` devuelto por ésta llega a través del registro EAX. El valor devuelto se almacena en la variable `z` (en SS:EBP–12) como requería el código fuente de `main`. Según sea el tipo de una función, el valor devuelto se transfiere en un lugar diferente:

- Cuando el tipo devuelto por la función es un entero de 8, 16 o 32 bits o una estructura de no más de 4 bytes se usa el registro EAX para devolver el valor.
- Si la función es de tipo `float` o `double` entonces el valor se devuelve a través del registro ST0 de la FPU.
- Si la función devuelve una estructura con más de 4 bytes, la función llamadora ubica espacio para la estructura a devolver (normalmente en la pila) y pasa un puntero a este espacio como un argumento oculto adicional. La función llamada copia la estructura en la dirección indicada por el puntero y devuelve el puntero a través de EAX. La función llamadora puede entonces copiar la estructura allí donde precise.

3.2. Código de la función llamada

Veamos ahora como es el código que crea el compilador para implemetar la función `resta`. Suponiendo que están desactivadas las opciones de optimización del compilador el código que generará el compilador será una traducción línea a línea del código fuente original.

Listado 2: Código de la función `resta`

```
resta:  push    ebp             ; Salvar EBP de la función llamadora.
        mov     ebp,esp         ; EBP apunta a la trama de la pila.
        sub     esp,4           ; Crear espacio para variable local r.
        mov     eax,[ebp+8]     ; Cargar en EAX el argumento a.
        sub     eax,[ebp+12]    ; Restarle a EAX el argumento b.
        mov     [ebp-4],eax     ; Almacenar en la variable local r.
        mov     eax,[ebp-4]     ; Devolver r a través de EAX.
        mov     esp,ebp         ; Liberar la memoria ocupada por r.
        pop     ebp             ; Recuperar el valor inicial de EBP.
        ret                     ; Retornar.
```

El código de la función se compone de los siguientes bloques:

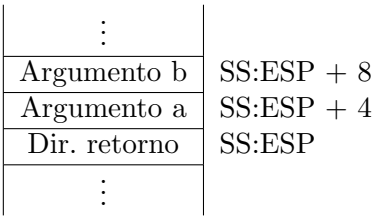
1. Código de prólogo.
2. Código que implementa la definición de la función.
3. Código de epílogo.

3.2.1. Código de prólogo

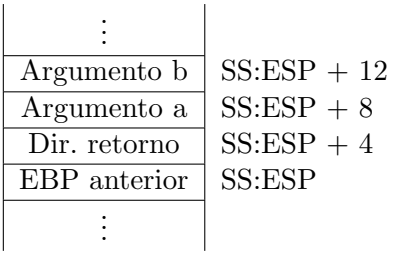
El código de prólogo es el código con el que se inicia la función. Su misión es hacer los preparativos necesarios para permitir que el resto de la función tenga acceso a los argumentos de entrada y a las variables locales y ubicar espacio en la pila para estas últimas. En la función `resta`, el código de prólogo lo forman las instrucciones:

```
push    ebp
mov     ebp,esp
sub     esp,4
```

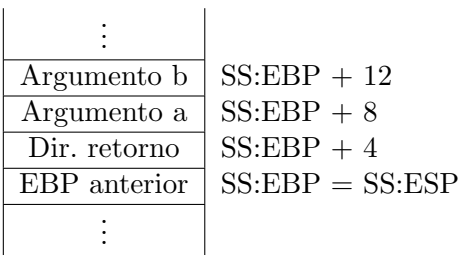
Para comprender como funciona, recordemos las operaciones que el código de llamada ha realizado sobre la pila: primero se introdujeron los valores de los argumentos en orden inverso y luego se ejecutó una instrucción CALL cercana que introdujo en la pila el offset de la dirección de retorno. Por tanto, al comenzar la ejecución de la función `resta` las posiciones cercanas a la cima de la pila presentan el siguiente aspecto:



Como la función `resta` va a utilizar el registro EBP para tener acceso a los argumentos y a la variable local y este registro ya está en uso en la función llamadora, lo primero que hace el código de prólogo es salvar en la pila el valor de EBP mediante una instrucción `push ebp`. Tras ejecutar esta instrucción, la pila queda:



A continuación, EBP se carga con el valor en ESP mediante una instrucción `mov ebp,esp` de forma que ahora EBP apunta también a la cima de la pila:



Seguidamente, se crea espacio en la pila para la variable local `r`. Esto se hace decrementando el puntero de pila ESP en cuatro unidades ya que, como hemos dicho, el tipo `int` de un compilador de 32 bits ocupa una doble palabra. En este momento la imagen de la pila es:

⋮	
Argumento b	SS:EBP + 12
Argumento a	SS:EBP + 8
Dir. retorno	SS:EBP + 4
EBP anterior	SS:EBP = SS:ESP
variable r	SS:EBP – 4
⋮	

Si la función `resta` hubiese declarado dos variables locales de tipo `int`, el puntero de pila se habría decrementado en ocho unidades. En general, ESP resultará decrementado en tantas unidades como posiciones de memoria necesiten las variables locales. Las sucesivas variables locales que declara una función van ocupando posiciones sucesivamente decrecientes en la pila. La estructura de datos formada en la pila con los argumentos pasados a la función, la dirección de retorno, el antiguo valor de EBP y las variables locales se denomina trama de pila (stack frame). Si dentro de la función se emplean instrucciones PUSH para almacenar datos en la pila, ésta crecerá por debajo de las variables locales. Esto no cambiará las posiciones relativas a EBP de los argumentos y las variables locales. La función `main` crea su trama de pila de forma totalmente análoga. De ahí que en `main` las variables locales ocupen las posiciones indicadas por la tabla 4.

Volviendo a resta, en este momento, en las posiciones EBP + 8, EBP + 12 y EBP – 4 tenemos localizados los argumentos `a` y `b` y la variable local `r`, respectivamente. La función puede acceder a todos ellos empleando el desplazamiento relativo a EBP correspondiente. Esto es justo lo que se hace en el código que sigue para restar los valores de los argumentos y almacenar el resultado en `r`.

3.2.2. Código que implementa la definición de la función

Una vez terminado el código de prólogo, comienza el código resultante de compilar el cuerpo de la función. En la función resta este código es:

```
mov     eax,[ebp+8]
sub     eax,[ebp+12]
mov     [ebp-4],eax
mov     eax,[ebp-4]
```

Nótese como, al estar desactivadas las opciones de optimización, el compilador genera una instrucción instrucción `mov eax,[ebp-4]` para traducir la línea `return(r);` a pesar de que EAX ya tiene el valor de `r`.

3.2.3. Registros que deben preservarse dentro de la función

Además del registro EBP, el código de la función llamadora generado por el compilador puede estar usando otros registros del procesador. Así, los registros ESI y EDI son usados por el compilador para almacenar variables de tipo registro (variables declaradas con el modificador `register`). Los registros de segmento DS y SS también contienen valores importantes puesto que apuntan a los segmentos de datos y de pila manejados por

el programa. Los registros EBX y FS también son usados por distintos compiladores o el sistema operativo para apuntar a estructuras de datos internas. Por tanto, si una función implementada en ensamblador necesita modificar alguno de estos registros deberá almacenar sus contenidos en la pila y recuperarlos antes de salir. Por el contrario, el registro de segmento ES puede usarse libremente.

3.2.4. Código de epílogo

El código de epílogo finaliza la ejecución de una función. Se encarga de liberar el espacio de pila usado por las variables locales, restablece el valor que tenía EBP al entrar en la función para asegurar que la función llamadora siga teniendo acceso a sus propias variables locales y argumentos y, finalmente, retorna. En la función `resta`, el código de epílogo es:

```
mov     esp,ebp
pop     ebp
ret
```

Para liberar el espacio ocupado por las variables locales, el puntero de pila ESP se carga con el valor en EBP. Con esto ESP vuelve a apuntar a la misma posición que tras la instrucción `push ebp` del prólogo, retirando con ello de la pila las variables locales. En caso de que no se creen variables locales esta instrucción no está presente. La instrucción `pop ebp` restaura el valor inicial de EBP. Por último, la instrucción `ret` extrae el offset de la dirección de retorno de la pila devolviendo el control a la función llamadora. Los argumentos de la función quedan en la pila. La función llamadora será la encargada de retirarlos.

3.3. Nombres de los símbolos en otros compiladores

Si estamos usando un compilador distinto al gcc para Linux, por ejemplo un compilador para Windows, hay que tener en cuenta que, por defecto, estos compiladores añaden un carácter de subrayado '_' delante de todos los identificadores cuando generan el módulo objeto. Por ejemplo, en el caso de la función `resta` el código de llamada sería

Listado 3: Código de llamada si el compilador añade '_'

```
...
; Código anterior a la llamada a resta
...
Código generado por la línea: z = resta( x, y );
push    dword [ebp-8] ; Introducir y (arg. b) en la pila.
push    dword [ebp-4] ; Introducir x (arg. a) en la pila.
call    _resta        ; Llamar a la función resta.
add     esp,8         ; Extraer los argumentos de la pila.
mov     [ebp-12],ax   ; Almacenar el valor devuelto en z.
...
; Código posterior a la llamada a resta.
...
```

y el de la propia función `resta`

Listado 4: Código de la función si el compilador añade '_'

```
_resta: push    ebp          ; Salvar EBP de la función llamadora.
        mov     ebp,esp      ; EBP apunta a la trama de la pila.
        sub     esp,4        ; Crear espacio para variable local r.
        mov     eax,[ebp+8]  ; Cargar en EAX el argumento a.
```

```
sub     eax,[ebp+12]; Restarle a EAX el argumento b.
mov     [ebp-4],eax ; Almacenar en la variable local r.
mov     eax,[ebp-4] ; Devolver r a través de EAX.
mov     esp,ebp     ; Liberar la memoria ocupada por r.
pop     ebp         ; Recuperar el valor inicial de EBP.
ret                 ; Retornar.
```

Este hecho pasa normalmente desapercibido ya que es algo que forma parte del funcionamiento interno de compilador, el enlazador y las librerías. Sin embargo, debe ser tenido en cuenta cuando se pretende enlazar con C un módulo creado en otro lenguaje. De cara a nuestro objetivo de enlazar módulos en ensamblador con módulos en C, la consecuencia es que, si usamos uno de estos compiladores, en el fuente en ensamblador tendremos que escribir precedidos de un carácter de subrayado todos los identificadores (nombres de procedimientos y variables) a los que deseemos tener acceso desde C. La misma indicación es válida si lo que pretendemos es llamar desde ensamblador una función definida en un módulo o librería de C.

4. Plantilla de módulo en ensamblador para enlazar con C

A continuación se indicará la plantilla de definición de segmentos que en general podemos usar para los ficheros fuentes en ensamblador que contengan procedimientos destinados a enlazarse con C según el modelo de memoria *flat*. La sintaxis es la usada por el ensamblador NASM usado en la asignatura.

Como vemos en el listado 5 se definen dos secciones de datos, `.data` y `.bss`, y la sección de código, llamada `.text`. La sección `.data` contiene los datos estáticos inicializados, es decir, que tienen un valor que ha sido definido explícitamente por el programador. Ejemplos de estos datos serían las cadenas de formato de las funciones `printf` o las variables globales inicializadas. Por su parte, la sección `.bss` [1] contiene las variables estáticas a las que el programador no ha dado un valor inicial. Normalmente, el sistema operativo o el código de inicialización que el enlazador añade al ejecutable inicializan la sección `bss` a cero. La sección `.text` contiene el código de las funciones que componen el módulo fuente incluida la función `main` si ésta forma parte de dicho módulo fuente (este siempre es el caso si el programa se componen de un único módulo fuente en C).

No se observa la definición de una sección específica para la pila ya que esto se realiza en un módulo objeto aparte llamado módulo de inicialización, o *start-up*, que posteriormente se enlaza con el modulo resultante de la compilación del fuente en C y con las librerías para generar el ejecutable. El módulo de *start-up* contiene también el código necesario para ajustar convenientemente el registro ESP al comienzo de la ejecución del programa.

Si la función no declara o utiliza datos en ninguno de las secciones `.data` y `.bss` pueden suprimirse del listado en ensamblador las correspondientes directivas de apertura de estas secciones.

[1] El nombre bss proviene de una pseudoinstrucción o directiva del UA-SAP (United Aircraft Symbolic Assembly Program), un ensamblador desarrollado a mediados de la década de 1950 para el ordenador IBM 740. Es un acrónimo de *Block Started by Simbol* (bloque comenzado por símbolo). En el ensamblador UA-SAP la pseudoinstrucción BSS servía para reservar un bloque de memoria compuesto por un determinado número de palabras de memoria la primera de las cuales quedaba identificada por una etiqueta de forma análoga a como lo hacen las directivas RESB, RESW, etc. del ensamblador NASM usado por nosotros.

Listado 5: Plantilla de definición de secciones

```
        section .data

        ...
        ; La sección .data contiene datos inicializados.
        ...


        section .bss

        ...
        ; La sección .bss contiene espacio para datos no inicializados.
        ...

        section .text

        ...
        ; La sección .text contiene el código.
        ...

funcion1:
        ...
        ; Código de la función funcion1.
        ...

funcion2:
        ...
        ; Código de la función funcion2.
        ...

        ...
        ; Código de otras funciones.
        ...
```

5. Ejemplo de módulo en ensamblador para enlazar con C

En este apartado recogeremos lo expuesto en las precedentes para crear un módulo fuente en ensamblador que contenga una función destinada a ser llamada desde un programa en C. La función se llamará `producto_escalar` y servirá para calcular el producto escalar de dos vectores de enteros.

Listado 6: Ejemplo de función en ensamblador para enlazar con C

```
;=================================================================
; int producto_escalar(int * ptr_x , int * ptr_y , int n);
;
; ptr_x : puntero al primer vector.
; ptr_y : puntero al segundo vector.
; n: número de valores en los vectores.
;
;=================================================================

        section .text

        global  producto_escalar
```

```
producto_escalar:

%define arg_ptr_x dword [ebp+8]
%define arg_ptr_y dword [ebp+12]
%define arg_n     dword [ebp+16]

        push    ebp
        mov     ebp,esp
        push    ebx
        push    esi
        push    edi

        xor     eax,eax
        mov     esi,arg_ptr_x
        test    esi,esi
        jz      salida
        mov     edi,arg_ptr_y
        test    edi,edi
        jz      salida
        mov     ecx,arg_n
        cmp     ecx,0
        jle     salida

bucle:  mov     ebx,[esi]
        imul    ebx,[edi]
        add     eax,ebx
        add     esi,4
        add     edi,4
        loop    bucle

salida: pop     edi
        pop     esi
        pop     ebx
        pop     ebp
        ret
```

Los símbolos `arg_ptr_x`, `arg_ptr_y` y `arg_n` se definen para hacer más clara la ubicación de los argumentos.

5.1. La directiva GLOBAL

En el módulo en ensamblador anterior podemos observar la siguiente línea:

```
global producto_escalar
```

La finalidad de esta línea es hacer que el símbolo `producto_escalar` sea accesible desde otros módulos. Por defecto, el ámbito de los símbolos definidos en ensamblador es el del módulo fuente que los contiene. En nuestro caso, el símbolo `producto_escalar` deberá ser accesible desde el módulo en C en el que se llame a la función, por tanto, es necesario hacerlo global mediante la directiva GLOBAL.

6. Convenio de llamada de un compilador de C de 64 bits

En este apartado se describirá como desarrollar funciones en ensamblador para ser enlazadas con código generado por un compilador de C de 64 bits. Las principales diferen-

cias que hay que tener en cuenta respecto a los explicado previamente para el caso de un compilador de 32 bits son:

- Las direcciones son ahora datos de 64 bits.
- El paso de argumentos se realiza principalmente a través a de los registros.
- Para las operaciones con números en punto flotante no se usan las instrucciones de la FPU sino las extensiones SSE2.

6.1. Convenio de llamada de C en Linux de 64 bits

Hasta seis argumentos enteros se pasan a las funciones a través de registros. De izquierda a derecha, los argumentos se pasan en los registros RDI, RSI, RDX, RCX, R8 y R9. Si una función tiene más de seis argumentos, los restantes se pasan a través de la pila.

En Linux de 64 bits, el tipo `long` (y `unsigned long`) tiene 64 bits. El tipo `int` (y `unsigned int`) sigue teniendo un tamaño de 32 bits, como en un compilador de 32 bits.

Los argumentos con un tamaño de 32, 16 u 8 bits se encontrarán en los 32, 16 u 8 bits de los registros. Los argumentos de 64 bits ocuparán el registro completo.

Los registros RDI, RSI, RDX, RCX, R8 y R9 así como los registros RAX, R10 y R11 pueden usarse dentro de las funciones sin necesidad de preservar sus valores.

Los valores retornados se devuelven a través de RAX. En caso se que la función devuelva un dato de más 64 bits también se usan los registros RAX y RDX (Por ejemplo, sería el caso de una función que devuelve por valor una estructura de más de 64 bits).

Las operaciones de punto flotante se realizan mediante las instrucciones y los registros SSE, excepto para el caso de los datos de tipo `long double`. Hasta ocho argumentos en punto flotante se pasan en los registros XMM0 a XMM7. El retorno de valores de punto flotante se realiza a través de XMM0 (y XMM1 si es necesario). Los argumentos de tipo `long double` se pasan a través de la pila y los resultados de tipo `long double` se devuelven en la cima de la pila de la FPU (ST0).

Pueden usarse todos los registros SSE y de la FPU sin necesidad de preservarlos.

Los argumentos enteros y de punto flotante se cuentan por separado, es decir, en una función como la siguiente

```
void funcion(long a, double b, int c);
```

El argumento `a` se pasa a través de RDI, el argumento `b` se pasa a través de XMM0 y el argumento `c` se pasa a través del registro ESI (los 32 bits bajos de RSI).

6.2. Convenio de llamada de C en Windows de 64 bits

Hasta cuatro argumentos enteros se pasan a las funciones a través de registros. De izquierda a derecha, los argumentos se pasan en los registros RCX, RDX, R8 y R9. Si una función tiene más de cuatro argumentos, los restantes se pasan a través de la pila.

En Windows de 64 bits, el tipo `long` (y `unsigned long`) tiene 32 bits (igual que los tipos `int` y `unsigned int`). El tipo `long long` (así como el `unsigned long long`) y el tipo `_int64` tienen un tamaño de 64 bits.

Los argumentos con un tamaño de 32, 16 u 8 bits se encontrarán en los 32, 16 u 8 bits de los registros. Los argumentos de 64 bits (tipos `long` y `unsigned long`) ocuparán el registro completo.

Los registros RCX, RDX, R8 y R9 así como los registros RAX, R10 y R11 pueden usarse dentro de las funciones sin necesidad de preservar sus valores.

Los valores retornados se devuelven a través de RAX.

Las operaciones de punto flotante se realizan mediante las instrucciones y los registros SSE, excepto para el caso de los datos de tipo `long double`. Hasta cuatro argumentos en punto flotante se pasan en los registros XMM0 a XMM3. El retorno de valores en punto flotante se realiza a través de XMM0. Los argumentos de tipo `long double` se pasan a través de la pila y los resultados de tipo `long double` se devuelven en la cima de la pila de la FPU (ST0).

Pueden usarse todos los registros SSE y de la FPU sin necesidad de preservarlos.

Los argumentos enteros y de punto flotante se cuentan juntos, es decir, en una función como la siguiente

```
void funcion(long long a, double b, int c);
```

El argumento `a` se pasa a través de RCX, el argumento `b` se pasa a través de XMM1 y el argumento `c` se pasa a través del registro R8D (los 32 bits bajos de R8).