# Práctica 3.- Programación mixta C-asm x86 Linux

## 1 Resumen de objetivos

Al finalizar esta práctica, se debería ser capaz de:

- Usar las herramientas `gcc`, `as` y `ld` para compilar código C, ensamblar código ASM, enlazar ambos tipos de código objeto, estudiar el código ensamblador generado por `gcc` con y sin optimizaciones, localizar el código ASM en-línea introducido por el programador, y estudiar el correcto interfaz del mismo con el resto del programa C.
- Reconocer la estructura del código generado por `gcc` según la convención de llamada `cdecl`.
- Reproducir dicha estructura llamando a funciones C desde programa ASM, y recibiendo llamadas desde programa C a subrutinas ASM.
- Escribir fragmentos sencillos de ensamblador en-línea.
- Usar la instrucción CALL (con convención `cdecl`) desde programas ASM para hacer llamadas al sistema operativo (*kernel* Linux, sección 2) y a la librería C (sección 3 del manual).
- Enumerar los registros y algunas instrucciones de los repertorios MMX/SSE de la línea x86.
- Usar con efectividad un depurador como `gdb/ddd`.
- Argumentar la utilidad de los depuradores para ahorrar tiempo de depuración.
- Explicar la convención de llamada `cdecl` para procesadores x86.
- Recordar y practicar en una plataforma de 32bits las operaciones de cálculo de paridad, cálculo de peso Hamming (population count), suma lateral (de bits o de componentes SIMD enteros) y producto de matrices.

## 2 Convención de llamada `cdecl`

En la práctica anterior ya vimos la conveniencia de dividir el código de un programa entre varias funciones, para facilitar su legibilidad y comprensión, además de su reutilización. En la Figura 1 se vuelve a mostrar la suma de lista de enteros de 32bits, destacando estos tres aspectos que ahora nos interesan:

- La dirección de inicio de la lista y su tamaño se le pasa a la función a través de registros.
- El resultado se devuelve al programa principal a través del registro EAX.
- La subrutina preserva el valor de EDX.

Probablemente el autor de esta función considere que quien reutilice sus funciones debe aprender qué registros se deben usar para pasar los argumentos, teniendo garantizado que a la vuelta de la subrutina sólo se habrá modificado el valor del registro EAX, que contiene el valor de retorno de la función.

Se pueden usar varias alternativas para pasar parámetros a funciones y para retornar los resultados de la función al código que la ha llamado. Se llama ***convención de llamada*** (*calling convention*) al conjunto de alternativas escogidas (para pasar parámetros y devolver resultados). Corresponde a la convención determinar, por ejemplo:

- Dónde se ponen los parámetros (en registros, en la pila o en ambos).
- El orden en que se pasan los parámetros a la función.
  - Si es en registros, en cuál se pasa el parámetro 1º, 2º, etc.
  - Si es en pila, los parámetros pueden introducirse en el orden en que aparecen en la declaración de la función (como en Pascal) o al contrario (como se hace en C).
    - La primera opción exige que el lenguaje sea fuertemente tipificado, y así una función sólo podrá tener un número fijo de argumentos de tipo conocido.
    - La segunda opción permite un nº variable de argumentos de tipos variables.
- Qué registros preserva el código de llamada (invocante) y cuáles la función (código invocado).
  - Los primeros (*caller-save, salva-invocante*) pueden usarse directamente en la función.
  - Los segundos (*callee-save, salva-invocado*) deberían salvarse a pila antes de que la función los modifique, para poder restaurar su valor antes de retornar al invocante.

- Quién libera el espacio reservado en la pila para el paso de parámetros: el código de llamada (invocante, como en C) o la función (código invocado, como en Pascal).
    - La primera opción permite un número variable de argumentos. El invocante siempre sabe cuántos han sido esta vez (en cada invocación podría ser un número distinto).
    - La segunda opción exige que el lenguaje sea fuertemente tipificado, pero ahorra código: la instrucción para liberar pila aparecen una única vez, en la propia función.

La convención de llamada depende de la arquitectura, del lenguaje, y del compilador concreto. Así en un procesador con pocos registros, como los x86 de 32 bits, generalmente se prefiere pasar los parámetros a una función a través de la pila, mientras que en procesadores con muchos registros se prefiere pasar los parámetros a través de registros. En los procesadores x86_64 se usan registros y la pila.

En este guión nos centraremos en la convención `cdecl`, estándar para arquitecturas x86 de 32 bits en lenguaje C, y también en lenguaje C++ para funciones globales. Si programamos funciones ensamblador respetando la convención `cdecl`, el código objeto generado (usando `as`) podrá inter-operar con código objeto generado (mediante `gcc`) a partir de código fuente C/C++; es decir, podremos construir un programa mezclando ficheros objeto compilados desde fuentes C/C++ con ficheros objeto ensamblados desde fuentes ASM. La convención de llamada `cdecl` tiene las siguientes especificaciones:

```
# suma.s:     Sumar los elementos de una lista
#             llamando a función, pasando argumentos mediante registros
# retorna:    código retorno 0, comprobar suma en %eax mediante gdb/ddd

# SECCIÓN DE DATOS (.data, variables globales inicializadas)
.section .data
lista:
       .int 1,2,10, 1,2,0b10, 1,2,0x10
longlista:
       .int   (.-lista)/4
resultado:
       .int   0x01234567

# SECCIÓN DE CÓDIGO (.text, instrucciones máquina)
.section .text
_start: .global _start           # PROGRAMA PRINCIPAL

       mov     $lista, %ebx      # 1er arg. EBX: dirección array lista
       mov  longlista, %ecx      # 2º  arg. ECX: número elementos a sumar
       call suma                 # llamar suma(&lista, longlista);
       mov  %eax, resultado

       mov  $1, %eax             #   void _exit(int status);
       mov  $0, %ebx
       int  $0x80

# SUBRUTINA: int suma(int* lista, int longlista);
# entrada:   1) %ebx = dirección inicio array
#            2) %ecx = número de elementos a sumar
# salida:       %eax = resultado de la suma

suma:
       push    %edx              # preservar %edx
       mov  $0, %eax             # acumulador
       mov  $0, %edx             # índice
bucle:
       add  (%ebx,%edx,4), %eax
       inc        %edx
       cmp   %edx,%ecx
       jne    bucle

       pop   %edx                # restaurar %edx
       ret
```

**Figura 1: suma.s: paso de parámetros por registros**

- Los parámetros se pasan en pila, de derecha a izquierda; es decir, primero se pasa el último parámetro, después el penúltimo… y por fin el primero.
- El espacio reservado en la pila para el paso de parámetros lo libera el código que llama. Estas dos primeras alternativas de la convención permiten al lenguaje C soportar funciones con un número variable de argumentos.
- El resultado se devuelve en EAX usualmente (ver Tabla 1). También se pueden pasar punteros o referencias a una función C/C++ para que ésta modifique el valor referenciado.
- Los registros EAX, ECX, EDX son *salva-invocante* (*caller-save*): la función los puede usar directamente, sin tener que preservarlos (sin tener que guardarlos en la pila ni recuperarlos antes de retornar). Es responsabilidad del invocante (*caller*, el código que llama a la función) guardarlos en la pila si desea recuperar su valor tras el retorno de la función.
- Los registros EBX, ESI y EDI son *salva-invocado* (*callee-save*): la función debe preservarlos (guardarlos en pila) y restaurarlos antes de retornar, si necesitara modificar su contenido.
- Los registros ESP y EBP son especiales y no deben manipularse: la convención `cdecl` asume que funcionan como puntero de pila y marco de pila.

| Tipo de variable | Registro |
|---|---|
| [unsigned] long long int (64-bit) | EDX:EAX |
| [unsigned] long | EAX |
| [unsigned] int | EAX |
| [unsigned] short | AX |
| [unsigned] char | AL |
| punteros | EAX |
| float / double | ST(0) – tope de pila x87 |

**Tabla 1: Devolución de resultados de una función bajo cdecl - 32 bits**

Hay algunas variaciones en la convención `cdecl` según el sistema operativo y compilador de C/C++, aunque no afectan a lo comentado hasta ahora (más información en [3]).

## Ejercicio 1: suma_01_S_cdecl

Modificar el fichero suma.s mostrado anteriormente (Figura 1) para volverlo conforme a la convención `cdecl`. Ensamblar, enlazar, depurar, y comprobar que sigue calculando el resultado correcto. En la Figura 2 se muestran las líneas que deben modificarse.

```
# suma.s del Guión 1
# 1.- añadiéndole convención cdecl
#      as --32 -g      suma_01_S_cdecl.s -o suma_01_S_cdecl.o
#      ld -m elf_i386  suma_01_S_cdecl.o -o suma_01_S_cdecl
...
start: .global _start      # PROGRAMA PRINCIPAL
       pushl  longlista    # 2º  arg: número de elementos a sumar
       pushl     $lista    # 1er arg: dirección del array lista
       call   suma         # llamar suma(&lista, longlista);
       add    $8, %esp     # quitar args
       mov  %eax, resultado
...
# SUBRUTINA: int suma(int* lista, int longlista);
suma:
       push %ebp           # Ajuste marco pila
       mov  %esp, %ebp
                           # antes, en el original, conservar todo
       push %ebx           # push %edx
       mov  8(%ebp), %ebx  # ahora %ebx es callee-save en cdecl
       mov 12(%ebp), %ecx  # %ecx,%edx no (caller-save)
...

       pop %ebx            # Recuperar callee-save
       pop %ebp            # Deshacer marco pila
       ret
```

**Figura 2: suma_01_S_cdecl.s: paso de parámetros por pila (convención cdecl)**

Observar que, en el *programa principal*, se introducen los argumentos en pila en orden inverso, se invoca a la función, y tras el retorno se limpia la pila. En la *función*, se empieza ajustando el marco de pila, se preservan registros salva-invocado (si hace falta), y se accede a los argumentos tomando como base el marco de pila EBP. Al retornar, se restauran los registros preservados y el marco de pila anterior.

Toda función comienza ajustando el marco de pila al tope actual de pila (salvando previamente el marco anterior), de manera que durante la ejecución de la función, `(%ebp)` es el antiguo marco, `4(%ebp)` es la dirección de retorno, y a partir de `8(%ebp)` están los argumentos de la función. Las dos primeras instrucciones de cualquier función `cdecl` son las mostradas en la Figura 2. El que genera el código (ya sea programador humano o el gcc) sabe qué registros modifica la función, así que si alguno de ellos es salva-invocado debe preservar su valor (en pila) para poder restaurarlo a la vuelta. En nuestro caso necesitamos 4 registros, así que al menos uno debe ser salva-invocado. Toda función termina dejando la pila como estaba: restaurando los registros salva-invocado, el marco de pila anterior, y la dirección de retorno, esto es, retornando al invocante.

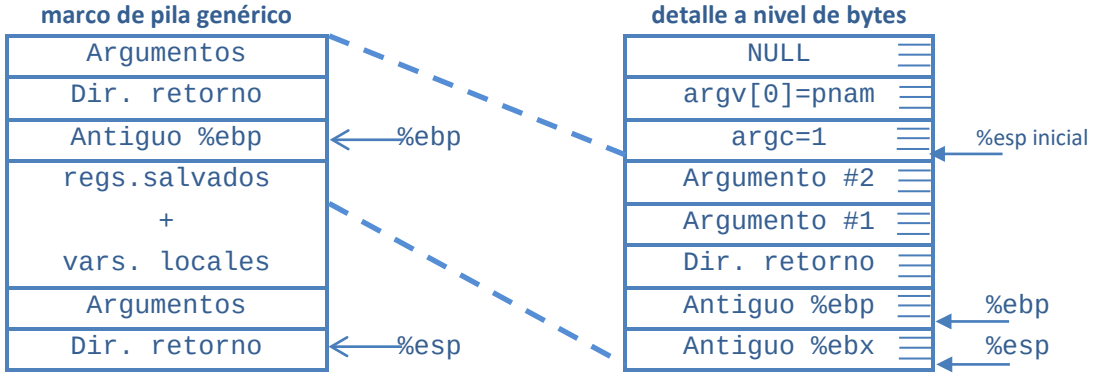


**Figura 3: marco de pila genérico, y marco correspondiente al ejemplo**

Ejecutar paso a paso con ddd el programa suma_01_S_cdecl de la Figura 2 y comprobar que todo sucede como se ha indicado. En el Apéndice 2 se ofrecen unas preguntas de autocomprobación para comprobar la correcta comprensión de este ejercicio.

## Ejercicio 2: suma_02_S_libC

La ventaja de usar la convención `cdecl` es que podemos inter-operar con otras funciones conformes a `cdecl`, como son obviamente todas las funciones de la librería C. En la sección 2 del manual se documentan los *wrappers* a llamadas al sistema (p.ej.: `man 2 exit`), y en la sección 3 las funciones de librería (p.ej.: `man 3 printf`).

Modificar el programa anterior, añadiéndole una llamada a `printf()` para sacar por pantalla el resultado en decimal y hexadecimal, y sustituyendo la llamada directa al *kernel* Linux por el correspondiente *wrapper* libC. Ensamblar, enlazar, depurar, y comprobar que sigue calculando el resultado correcto. En la Figura 4 se muestran las líneas que deben modificarse, y aparece como comentario el comando utilizado para enlazar con la librería C.

```
# suma.s del Guión 1
# 1.- añadiéndole convención cdecl
# 2.- añadiéndole printf() y cambiando syscall por exit()
#      as --32 -g      suma_02_S_libC.s -o suma_02_S_libC.o
#      ld -m elf_i386  suma_02_S_libC.o -o suma_02_S_libC \
#            -lc   -dynamic-linker /lib/ld-linux.so.2

.section .data
lista:       .int 1,2,10,  1,2,0b10,  1,2,0x10
longlista:   .int (.-lista)/4
resultado:   .int 0x01234567
formato:     .ascii "resultado = %d = %0x hex\n\0"
                        # formato para printf() libC
.section .text
_start: .global _start          # PROGRAMA PRINCIPAL
```

```
        pushl longlista
        pushl    $lista
        call  suma
        add   $8, %esp             # quitar args
        mov   %eax, resultado      # resultado=suma(&lista, longlista)

        push %eax             # versión libC de syscall __NR_write
        push %eax             # ventaja: printf() con formato "%d" / "%x"
        push $formato         #      traduce resultado a ASCII decimal/hex
        call  printf          # == printf(formato,resultado,resultado)
        add  $12, %esp

                              # versión libC de syscall __NR_exit
        pushl $0              #      mov $1, %eax
        call exit             #      mov $0, %ebx
#       add $4, %esp  (no ret)#      int $0x80     == _exit(0)
...
```

**Figura 4: suma_02_S_libC.s: llamando a libC desde ASM**

Observar que, para cada llamada a función, el *programa principal* introduce los argumentos en pila en orden inverso, y tras el retorno se limpia la pila. En la Figura 5 se ilustra la situación de la pila antes de llamar a cada función. Notar que no habrá oportunidad de limpiar los argumentos de `exit()`.

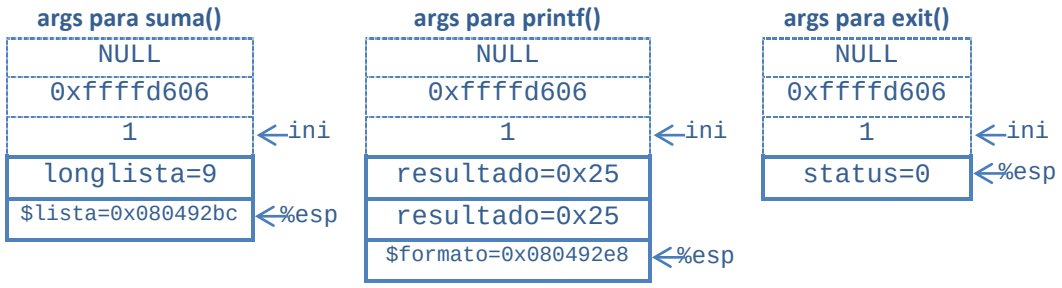


**Figura 5: paso de argumentos correspondientes al ejemplo libC**

Como se usan dos funciones de la librería C, es necesario enlazar con dicha librería (*switch* `-lc`). Si no se hace, las funciones `printf()/exit()` no se pueden resolver, es decir, el enlazador no sabe a qué dirección de subrutina saltar. Además, una instalación normal de `gcc` espera que las aplicaciones se compilen para usar la librería C dinámica (`libc.so`, por *shared object*), por lo cual necesitaremos especificar el enlazador dinámico a usar (el de 32bits, en nuestro caso).

Ejecutar desde línea de comandos el programa suma_02_S_libC de la Figura 4, aprovechando que ahora imprime el resultado por pantalla. Depurarlo también paso a paso con ddd, comprobando que se producen las configuraciones de pila mostradas en la Figura 5. En el Apéndice 2 se ofrecen unas preguntas de autocomprobación para comprobar la correcta comprensión de este ejercicio.

## Ejercicio 3: suma_03_SC

La ventaja de usar la convención `cdecl` es que podemos inter-operar con otras funciones conformes a `cdecl`. Hemos probado con funciones de la librería C, y ahora experimentaremos con nuestra propia función `suma()`, pasándola a lenguaje C.

Modificar el programa anterior, eliminando el código ensamblador de `suma()` y creando un nuevo módulo equivalente en lenguaje C. Ensamblar, enlazar, depurar, y comprobar que sigue calculando el resultado correcto. En la Figura 6 se muestran las líneas que deben modificarse, y los comandos utilizados para compilar, ensamblar y enlazar los dos módulos. Notar que dependiendo de la versión de gcc y opciones de optimización, puede ser necesario usar la opción `-fno-omit-frame-pointer`.

```
# MODULO suma_03_SC_s.s:  suma.s del Guión 1
# 1.- añadiéndole convención cdecl
# 2.- añadiéndole printf() y cambiando syscall por exit()
# 3.- extrayendo suma a módulo C para linkar
#      gcc -m32 -O1 -g -c suma_03_SC_c.c    [-fno-omit-frame-pointer]
#      as --32      -g    suma_03_SC_s.s -o suma_03_SC_s.o
#      ld -m elf_i386     suma_03_SC_c.o    suma_03_SC_s.o -o suma_03_SC \
#             -lc   -dynamic-linker /lib/ld-linux.so.2
```

```
...
formato:      .ascii "resultado = %d = %0x hex\n\0"
                              # formato para printf() libC (asciiz)
...
```

```
# MODULO suma_03_SC_c.c
int suma(int* array, int len)
{
    int  i,   res=0;
    for (i=0; i<len; i++)
        res += array[i];
    return res;
}
```

Figura 6: Aplicación suma_03_SC: llamando a módulo C desde módulo ASM

En este ejercicio se ha usado sufijo _SC_ para indicar que se llama desde ASM a C, y los módulos repiten en su nombre la extensión (_s.s, _c.c) para que no coincidan los nombres de los ficheros objeto. Recordar que gcc -c reutiliza el nombre del fuente, y que con as hay que indicar el nombre del objeto.

Ejecutar desde línea de comandos el programa resultante, comprobando que imprime el mismo resultado por pantalla. Depurarlo también paso a paso con ddd, comprobando que al pasar a lenguaje C los argumentos formales adquieren el valor de los parámetros actuales pasados en pila. Probablemente necesitemos listar la subrutina _start desde línea de comandos gdb, ya que al enlazar hemos puesto el objeto C como primer módulo, y por tanto ése será el que muestre ddd al inicio. En el Apéndice 2 se ofrecen las preguntas de autocomprobación para comprobar la correcta comprensión de este ejercicio.

## Ejercicio 4: suma_04_SC

Antes de llevárnoslo todo a lenguaje C, vamos a probar a dejar únicamente los datos y el punto de entrada en ensamblador. Nuestra única instrucción va a ser un salto (no llamada) a la subrutina suma, y ésta accederá a los datos globalmente, imprimirá el resultado y terminará el programa. No retornaremos de suma, ni usaremos instrucciones ensamblador para pasarle parámetros. Los cambios necesarios se ilustran en la Figura 7. No hay cambios en las instrucciones para compilar, ensamblar y enlazar los dos módulos. Hacerlo, y comprobar que se sigue calculando el resultado correcto.

```
# MODULO suma_04_SC_s.s
# suma.s del Guión 1
# 1.- añadiéndole convención cdecl
# 2.- añadiéndole printf() y cambiando syscall por exit()
# 3.- extrayendo suma a módulo C para linkar
# 4.- dejando sólo los datos, que el resto lo haga suma() en módulo C
...
.global lista, longlista, resultado, formato
.section .text
_start:  .global _start
       jmp suma
```

```
# MODULO suma_04_SC_c.c
#include <stdio.h>  // para printf()
#include <stdlib.h> // para exit()

extern int lista[];
extern int longlista, resultado;
extern char formato[];

void suma()
{
    int  i,   res=0;
    for (i=0; i<longlista; i++)
        res += lista[i];
    resultado = res;

    printf(formato,res,res);
    exit(0);
}
```

Figura 7: Aplicación suma_04_SC: dejando sólo datos y punto de entrada en módulo ASM

Notar que en el módulo C se añaden los `includes` necesarios, cambia la signatura de la función `suma` (ni toma argumentos ni produce resultado), y se usan los nombres de las variables globales, no de los parámetros. También se imprime el resultado y se finaliza el programa.

En el módulo ASM se declaran globales los símbolos exportados. En el módulo C se declaran externos. A gcc le basta con saber el tipo de esas variables, para generar las instrucciones que acceden a ellas (salvo la dirección, que se deja a cero, sin rellenar). En tiempo de enlace se resuelven estos símbolos: por el nombre se localiza la definición en las tablas de símbolos y se descubre la dirección que ocupan.

Ejecutar desde línea de comandos el programa resultante, comprobando que imprime el mismo resultado por pantalla. Depurarlo también paso a paso con ddd, comprobando que el salto a lenguaje C no toca la pila, y que una vez en C las variables globales son exactamente las definidas en ASM. Probablemente necesitemos listar la subrutina `_start` desde línea de comandos gdb, debido al orden de enlace escogido. En el Apéndice 2 se ofrecen las habituales preguntas de autocomprobación.

## Ejercicio 5: suma_05_C

Pasar todo el código a lenguaje C. Comprobar que se sigue calculando el resultado correcto.

```
# MODULO suma_05_C.c
#include <stdio.h>  // para printf()
#include <stdlib.h> // para exit()

int lista[]={1,2,10,  1,2,0b10,  1,2,0x10};
int longlista= sizeof(lista)/sizeof(int);
int resultado= 0x01234567;

int suma(int* array, int len)
{
    int  i,   res=0;
    for (i=0; i<len; i++)
        res += array[i];
    return res;
}

int main()
{
    resultado = suma(lista, longlista);
    printf("resultado = %d = %0x hex\n",
           resultado,resultado);
    exit(0);
}
```

Figura 8: suma_05_C: código C puro

Notar que se deshace el cambio de signatura de la función `suma`, las variables globales se definen en C, y el programa principal llama a nuestra función y a las de librería. El punto de entrada es ahora `main`. Notar la sintaxis para declarar e inicializar arrays, si se desconocía. El operador sizeof resulta útil para reproducir los cálculos que hacíamos en el fuente ASM.

Ejecutar desde línea de comandos el programa resultante, comprobando que imprime el mismo resultado por pantalla. Depurarlo con ddd. En el Apéndice 2 se ofrecen las habituales preguntas de autocomprobación.

## Ejercicio 6: suma_06_CS

Volver a pasar la función `suma` a un módulo ensamblador separado. En la Figura 9 se ilustra cómo quedaría el módulo C, y se recuerdan las instrucciones para compilar, ensamblar y enlazar (varias alternativas posibles). Comprobar que se sigue calculando el resultado correcto.

Las distintas alternativas para obtener el ejecutable son un recordatorio de lo estudiado en la Práctica 1 sobre compilación, ensamblado y enlazado. Como tenemos un módulo C y otro ASM, podemos:

1. compilarlo todo desde gcc (es la opción preferible), ó
2. compilar (gcc) y ensamblar (as) los fuentes a objetos, y enlazarlos con gcc, ó
3. enlazar esos mismos objetos con `ld`.

```c
# MODULO suma_06_CS_c.c
#include <stdio.h>  // para printf()
#include <stdlib.h> // para exit()
extern int suma(int* array, int len);

int lista[]={1,2,10,  1,2,0b10,  1,2,0x10};
int longlista= sizeof(lista)/sizeof(int);
int resultado= 0x01234567;

int main()
{
    resultado = suma(lista, longlista);
    printf("resultado = %d = %0x hex\n",
           resultado,resultado);
    exit(0);
}
```

```
# MODULO suma_06_SC_s.s
# ...
# 5.- entero en C
# 6.- volviendo a sacar la suma a ensamblador
#      gcc -m32 -O1 -g    suma_06_CS_c.c    suma_06_CS_s.s -o suma_06_CS
#
#      gcc -m32 -O1 -g -c suma_06_CS_c.c
#      as  --32     -g    suma_06_CS_s.s -o suma_06_CS_s.o
#      gcc -m32           suma_06_CS_c.o    suma_06_CS_s.o -o suma_06_CS
#
#      LDIR=`gcc -print-file-name=`
#      ld -m elf_i386     suma_06_CS_c.o    suma_06_CS_s.o -o suma_06_CS  \
#          -dynamic-linker /lib/ld-linux.so.2                             \
#          /usr/lib32/crt1.o /usr/lib32/crti.o /usr/lib32/crtn.o          \
#          $LDIR/32/crtbegin.o   $LDIR/32/crtend.o        -lc
...
```

**Figura 9: suma_06_CS: programa C llamando a función asm cdecl**

Notar que se indica que `suma` es `extern`, como antes lo fueron `lista` y `longlista`. Es tan frecuente que las funciones estén en otro módulo, que no hace falta indicar `extern` al compilador, basta con mostrarle el prototipo. Incluso si no se indicara prototipo, el compilador asumiría que la función es `int func()` (que devolverá int), produciéndose un aviso si resultara tener argumentos o devolver otra cosa. Los prototipos de una librería suelen recolectarse en un fichero `<librería>.h`, para su inclusión en programas que utilicen la librería.

Notar que es preferible compilar, ensamblar y enlazar la aplicación con `gcc`, ya que el punto de entrada es `main` (y vamos a usar la librería C). Anteriormente hemos preferido usar `as` porque el punto de entrada era `_start`, teniendo que enlazar explícitamente con la librería C y el enlazador dinámico cuando hemos usado funciones libC. El compilador `gcc` añade esas opciones (y otras para soporte en tiempo de ejecución), admite ficheros fuente C y ASM en una sola línea de comandos, y puede compilar, ensamblar y enlazar en un solo comando, por lo cual es preferible en el caso actual. Sólo para demostrar que pueden seguir usándose `as` y `ld`, se ofrecen las instrucciones alternativas. Notar que en ese caso, hace falta también indicar explícitamente el soporte en tiempo de ejecución (*C runtime*).

Ejecutar desde línea de comandos el programa resultante, comprobando que imprime el mismo resultado por pantalla. Depurarlo con ddd, comprobando que el marco de pila generado para `suma` es idéntico a cuando la función estaba programada en lenguaje C.

# 3 Ensamblador en-línea (*inline assembly*) con `asm()`

Hay ocasiones especiales en que resultaría conveniente introducir unas pocas instrucciones de lenguaje ensamblador entre (*en-línea con*) el código C, por motivos muy concretos:

- Utilizar alguna instrucción de lenguaje máquina que el compilador no conozca, o no utilice nunca, o no use en el caso concreto que nos interesa (`rdtsc`, `xchg`, etc)

- Aprovechar alguna característica (registro, etc) de la arquitectura que el compilador no utilice (*timestamp counter*, *performance counters*, etc).
- Conseguir alguna optimización que no sea posible mediante *switches* u otras características del compilador (*builtins*, etc), del lenguaje (*keywords* como *register*, etc), o mediante librerías optimizadas.

En general es difícil, a menudo muy difícil, y siempre muy tedioso, intentar ganar a gcc o cualquier compilador optimizador en lo que se refiere a movimientos de datos, bucles y estructuras de control, que usualmente es un gran porcentaje del texto de cualquier programa. Resulta más productivo estudiar el manual del compilador y usar los *switches* correspondientes (p.ej.: `-mtune=core2`, `-msse4.2`) para que éste genere instrucciones específicas de la arquitectura (si decide que son ventajosas), y reordene y alinee instrucciones y datos teniendo en cuenta detalles de la microarquitectura ignorados u obviados por la mayoría de los programadores (y aún prestándoles atención, se necesitarían manuales y simuladores para aprovecharlos en igual grado que gcc).

Por otro lado, puede suceder que sólo deseemos utilizar unas pocas instrucciones *entre medias* de nuestro código C para intentar mejorar sus prestaciones (por alguno de los motivos citados anteriormente). Para posibilitar esa inserción de unas pocas instrucciones ensamblador *en-línea* con el código C, gcc también dispone (igual que otros compiladores) de una sentencia `asm( )`, con la siguiente sintaxis:

- Básica: `asm("<sentencia ensamblador>")`
- Extendida: `asm("<sentencia asm>":<salidas>:<entradas>:<sobrescritos>)`

Aunque en principio el mecanismo está pensado para una única instrucción ensamblador, se puede aprovechar la concatenación de strings (dos strings seguidos en un fuente C se concatenan automáticamente) y los caracteres `"\n\t"` como terminación, para insertar varias líneas que el ensamblador interprete posteriormente como instrucciones distintas de código fuente ASM.

Si el código *inline* es totalmente independiente del código C, en el sentido de no necesitar coordinación con objetos controlados por el compilador (variables, registros de la CPU, etc), puede usar la sintaxis básica (sin *restricciones*). Pero habitualmente, desearemos que el código *inline* se coordine con el código C, porque queramos modificar el valor de alguna variable (***restricciones*** de `<salida>`), o consultarlo (***restricciones*** de `<entrada>`), o simplemente para no interferir con las optimizaciones en curso (***restricciones*** `<sobrescritos>`). En ese caso, usaremos la sintaxis extendida (con restricciones).

## Ejercicio 7: suma_07_Casm

Una ventaja de usar ensamblador *inline* es que podemos incorporar lo que de otra forma se hubiera convertido en un pequeño módulo ASM en el propio código C, facilitando el estudio de la aplicación y evitando la necesidad de ensamblar y enlazar separadamente, o al menos reduciendo el número de ficheros fuente implicados.

Modificar el ejemplo anterior, volviendo a incorporar el código ensamblador de `suma` como ensamblador en-línea. Compilar (tal vez haga falta usar `–fno-omit-frame-pointer`), ejecutar, y comprobar que sigue calculando el resultado correcto. En la Figura 10 se muestra el fuente resultante.

```
//  usar gcc –fno-omit-frame-pointer si gcc quitara el marco pila (%ebp)
#include <stdio.h>  // para printf()
#include <stdlib.h> // para exit()

int lista[]={1,2,10,  1,2,0b10,  1,2,0x10};
int longlista= sizeof(lista)/sizeof(int);
int resultado= 0x01234567;

int suma(int* array, int len)
{
//  int  i,   res=0;             // cuando gcc compila este código C
//  for (i=0; i<len; i++)        //   produce un código ASM
//      res += array[i];         //   similar al incorporado más abajo
//  return res;                  //   mediante la sentencia asm()
```

```
    asm("push      %ebx    \n"    // clobber (sobrescritos):
"      mov  8(%ebp),%ebx   \n"    // EBX
"      mov 12(%ebp), %ecx  \n"    // ECX
"                          \n"
"      mov $0, %eax        \n"    // EAX
"      mov $0, %edx        \n"    // EDX
"bucle:                    \n"
"      add (%ebx,%edx,4), %eax\n"
"      inc       %edx      \n"
"      cmp  %edx,%ecx      \n"
"      jne bucle           \n"
"                          \n"
"      pop  %ebx           \n"    // La sintaxis extendida incluiría:
//   :                             // output
//   :                             // input
//   : "cc",                       // clobber
//     "eax","ebx","ecx","edx"     // en este caso, la hemos comentado
    );
}

int main()
{
    resultado = suma(lista, longlista);
    printf("resultado = %d = %0x hex\n",
           resultado,resultado);
    exit(0);
}
```

**Figura 10: suma_07_Casm: incorporando módulo ASM como inline-asm**

Se ha escogido el nombre `_Casm_` para indicar que se usa asm *inline*. Notar que, como conocemos la convención `cdecl`, podemos obtener los valores de los argumentos `array` y `len` sin necesidad de coordinarnos con `gcc` mediante *restricciones* de entrada, y producir el valor de retorno (en EAX) sin indicar *restricciones* de salida. Ni siquiera necesitamos avisar a `gcc` de los registros que alteramos (restricciones de sobreescritos, o *clobber constraints*). El registro "cc" son los flags de estado (*condition codes*). A veces puede ser necesario indicar a `gcc` que nuestro código *inline* modifica los flags.

En el Apéndice 2 se ofrecen las habituales preguntas de autocomprobación.

## Restricciones de salida, de entrada, y sobrescritos

Como ya se comentó, es difícil ganar a `gcc` en movimiento de datos o control de flujo, y tedioso el simple hecho de intentarlo. El ensamblador en-línea es más efectivo para las situaciones en que conocemos alguna funcionalidad o mejora que `gcc` ha pasado por alto. Usualmente se trataría de insertar una única instrucción ensamblador (o pocas), pero que necesitamos coordinar con `gcc` porque:

- Modifican alguna variable (restricciones de salida)
- Necesitan el valor de alguna variable, constante, dirección… (restricciones de entrada)
- Modifican estado de la CPU que pueda estar usando `gcc` (restricciones sobrescritos)

Esa coordinación se expresa mediante las denominadas *restricciones* (*constraints*), con esta sintaxis:

- Salidas: [<nombre ASM >] "=<restricción>" (<nombre C >)
- Entradas: [<nombre ASM >] "<restricción>" (<expresión C >)
- Sobrescritos: "<reg>" | "cc" | "memory"

La idea general de funcionamiento de las restricciones es como sigue: antes de entrar a ejecutar la sentencia `asm()`, `gcc` satisface las condiciones de entrada, copiando cada <expresión C> a un recurso ensamblador (registro, inmediato, memoria, ver Tabla 2) que cumpla la restricción indicada (por eso se llaman *restricciones* de entrada). Durante la ejecución de la sentencia `asm()`, nos podremos referir al recurso mediante el <nombre ASM> escogido. Después de ejecutar la sentencia `asm()`, `gcc` satisface las condiciones de salida, copiando los recursos <nombre ASM> que lleven "=<restricción>" de salida a la variable <nombre C > que se indique.

Por poner un ejemplo para fijar conceptos, se podría escribir **`[arr] "r" (array)`** en restricciones de entrada para indicar a gcc que lo que en C llamamos **`array`** (su dirección de inicio) se debe almacenar en un registro cualquiera (restricción **`"r"`**, ver Tabla 2) antes de entrar en la sentencia `asm()`, y como no sabemos en cuál registro decidirá almacenarlo, vamos a llamarlo **`%[arr]`** en el código ensamblador.

Mediante estas copias, gcc asocia (*coordina*) el nombre o expresión C con algún recurso ensamblador (registro de la CPU, registro del coprocesador, operando inmediato, operando de memoria) que cumpla la restricción indicada. En la Tabla 2 se resumen las restricciones más comúnmente usadas.

El nombre ensamblador es opcional. Si se indica en la restricción, podremos hacer referencia a dicho recurso en nuestro código *inline* como %[<nombre ASM>]. Si no, el recurso se referenciará como %0, %1, %2… en el orden en que aparezca en la lista de restricciones, empezando con las restricciones de salida y terminando con las de entrada.

Notar que las restricciones de salida deben llevar el modificador "=" (o también "+" para entrada y salida, ver Tabla 2). Como son de salida, no se puede indicar una <expresión C> cualquiera, tiene que ser un <nombre C> de una variable (*L-value*) que pueda almacenar el valor del recurso al acabar la sentencia `asm()`.

En el apartado de sobrescritos se deben indicar los recursos que modifica nuestro código *inline*, a fin de que gcc no optimice erróneamente el acceso a los mismos, ignorando que han sido alterados en nuestra sentencia `asm`. En general, es buena idea comprobar el código ensamblador generado alrededor de nuestra sentencia `asm`, para anticipar (si lo vemos antes) o corregir (si no lo hemos visto antes) un posible error de coordinación con gcc, debido a haber especificado unas restricciones incorrectas. Conviene recordar que el mecanismo `asm` fue pensado inicialmente para una única instrucción máquina, y así veremos que a veces una restricción "=r" (salida registro) reutiliza el mismo registro que una entrada "r". A menudo, usando la restricción "+r" (o "=&r") desaparece el problema (*¿por qué?*).

El manual de gcc [7] y su *Inline assembly HOWTO* [8] son los documentos de referencia para las distintas restricciones disponibles, tanto en general para todos los procesadores soportados, como en particular para los procesadores de las familias x86 y x86-64. Existen también numerosos tutoriales y documentos web (ver por ejemplo la *Linux Assembly HOWTO* [9] y los tutoriales *SourceForge* [10]) sobre esta temática. Para nuestros objetivos, seguramente nos baste conocer las restricciones más básicas:

| Restricción | Registro | Restricción | Operando |
|---|---|---|---|
| a | EAX | **m** | **operando de memoria** |
| b | EBX | q | registros eax, ebx, ecx, edx |
| c | ECX | **r** | **registros eax, ebx, ecx, edx, esi, edi** |
| d | EDX | g | registro (q) o memoria (m) |
| S | ESI | I | valor inmediato 0..31 (despl-rotación) |
| D | EDI | **i** | **valor inmediato entero** |
| A | EDX:EAX | G | valor inmediato punto flotante |
| f | ST(i) – registro p.flotante | <n> | en restricción de entrada, un número indica que el operando también es de salida, la salida número %<n> |
| t | ST(0) – tope de pila x87 | | |
| u | ST(1) – siguiente al tope | | |
| | | | |
| **Modificadores** | | | |
| = | **Salida (write-only)** | =& | Early-clobber (salida sobrescrita antes de leer todas las entradas) |
| + | **Entrada-Salida** | | |

**Tabla 2: Restricciones (constraints) y modificadores más utilizados**

La mayoría de los fragmentos *inline* pueden resolverse con las restricciones que hemos retintado.

## Ejercicio 8: suma_08_Casm

Modificar el ejemplo anterior, reduciendo el código ensamblador en-línea al cuerpo del bucle `for`. Compilar, ejecutar, y comprobar que sigue calculando el resultado correcto. En la Figura 11 se muestra el fragmento relevante.

```
int suma(int* array, int len)
{
    int  i,   res=0;
    for (i=0; i<len; i++)
//      res += array[i];          // traducir sólo esta línea a ASM
    asm("add (%[a],%[i],4),%[r]"
     : [r] "+r" (res)           // output-input
     : [i]  "r" (i),            // input
       [a]  "r" (array)
//   : "cc"                     // clobber
    );
    return res;
}
```

Figura 11: suma_08_Casm: inline-asm con restricciones

Notar que para redactar este código *inline* no es necesario conocer la convención `cdecl`, y podemos obtener referencias a `array`, `i` y `res` coordinándonos con `gcc` mediante restricciones de salida y entrada.

El efecto de "**`res+=array[i];`**" se puede conseguir con una única sentencia ASM del estilo "**`add (%ebx,%edx,4),%eax`**" con tal de que en EBX esté la dirección del **`array`**, en EDX el índice **i** (ambos de entrada) y EAX se corresponda con la variable **res** (entrada-salida, ya que acumulamos sobre dicha variable). De hecho, nos daría igual que fueran esos u otros registros. Por eso ponemos restricción "r" en lugar de algo más concreto que tal vez podría interferir en las optimizaciones que esté realizando gcc alrededor de este código. En el Apéndice 2 se ofrecen las habituales preguntas de autocomprobación.

## Ejercicio 9: suma_09_Casm

Como el código generado es el mismo, no se espera que haya ninguna diferencia en cuanto a prestaciones entre los últimos tres ejemplos.

Para comprobarlo, crear un programa que incorpore las tres alternativas de `suma`, y que ejecute cada una cronometrando su tiempo de ejecución, usando la función de librería C `gettimeofday`. Compilar (tal vez haga falta usar `-fno-omit-frame-pointer`), ejecutar, comprobar que las tres versiones producen el mismo resultado, y calcular el tiempo de ejecución promedio (de cada versión) sobre 10 ejecuciones consecutivas. En la Figura 12 se muestra el programa sugerido.

```
//  según la versión de gcc y opciones de optimización usadas, tal vez haga falta
//  usar gcc -fno-omit-frame-pointer si gcc quitara el marco pila (%ebp)

#include <stdio.h>              // para printf()
#include <stdlib.h>             // para exit()
#include <sys/time.h>           // para gettimeofday(), struct timeval

#define SIZE (1<<16)            // tamaño suficiente para tiempo apreciable
int lista[SIZE];
int resultado=0;

int suma1(int* array, int len)
{
    int  i,   res=0;
    for (i=0; i<len; i++)
        res += array[i];
    return res;
}

int suma2(int* array, int len)
{
    int  i,   res=0;
    for (i=0; i<len; i++)
//      res += array[i];
    asm("add (%[a],%[i],4),%[r]"
     : [r] "+r" (res)           // output-input
     : [i]  "r" (i),            // input
       [a]  "r" (array)
//   : "cc"                     // clobber
    );
    return res;
}
```

```
int suma3(int* array, int len)
{
    asm("mov 8(%%ebp), %%ebx          \n"     // array
"       mov  12(%%ebp), %%ecx         \n"     // len
"                                     \n"
"       mov $0, %%eax                 \n"     // retval
"       mov $0, %%edx                 \n"     // index
"bucle:                               \n"
"       add (%%ebx,%%edx,4), %eax     \n"
"       inc         %%edx             \n"
"       cmp  %%edx,%%ecx              \n"
"       jne bucle                     \n"
     :                                        // output
     :                                        // input
     : "ebx"                                  // clobber
    );
}

void crono(int (*func)(), char* msg){
    struct timeval tv1,tv2;                   // gettimeofday() secs-usecs
    long           tv_usecs;                  // y sus cuentas

    gettimeofday(&tv1,NULL);
    resultado = func(lista, SIZE);
    gettimeofday(&tv2,NULL);

    tv_usecs=(tv2.tv_sec -tv1.tv_sec )*1E6+
             (tv2.tv_usec-tv1.tv_usec);
    printf("resultado = %d\t", resultado);
    printf("%s:%9ld us\n", msg, tv_usecs);
}

int main()
{
    int i;                                    // inicializar array
    for (i=0; i<SIZE; i++)                    // se queda en cache
        lista[i]=i;

    crono(suma1, "suma1 (en lenguaje C    )");
    crono(suma2, "suma2 (1 instrucción asm)");
    crono(suma3, "suma3 (bloque asm entero)");
    printf("N*(N+1)/2 = %d\n", (SIZE-1)*(SIZE/2)); /*OF*/

    exit(0);
}
```

**Figura 12: suma_09_Casm: esqueleto de programa para comparar tiempos de ejecución**

Notar que se ha definido un tamaño de array lo suficientemente grande como para que el tiempo de ejecución sea apreciable. De hecho, el motivo para no poner un tamaño mayor ha sido la incomodidad para calcular el resultado correcto mediante la fórmula correspondiente. En cualquier caso, incluso para tamaños mucho menores se venía cumpliendo que el tiempo de ejecución crecía linealmente con el tamaño del array (tamaño doble→tiempo doble), lo cual indica, para un algoritmo de complejidad lineal como éste, que el tiempo cronometrado no está dominado por otros factores ajenos, sino por el propio proceso realizado (sumar los N elementos, en este caso).

Notar que el tamaño del array no supone perjuicio para el cronometraje de ninguna versión. En nuestro caso es lo suficientemente pequeño como para caber en cache L2 y estar disponible para las tres ejecuciones, una vez inicializado el array. Si fuera demasiado grande tampoco importaría, porque al no caber, igual no cabe al inicializar, que no cabe al cronometrar la versión 1, que no cabe al cronometrar ninguna otra. En este caso, el orden de ejecución de las versiones no afecta a su cronometraje. Tampoco afecta cuál sea la primera que se ejecute, tras inicializar el array. En general, ese no es el caso, y se debe meditar cuidadosamente cómo realizar la medición de forma justa y equitativa para todas las versiones.

Notar que se ha introducido una ligera variante en la versión 3, y que cuando hay lista de sobrescritos, los registros se referencian como %%<reg>. Cuando no hay sobrescritos, basta con %<reg>. En el Apéndice 2 se ofrecen las habituales preguntas de autocomprobación.

Este mismo programa nos puede servir de esqueleto para el resto de trabajos de optimización y medición de tiempos (cronometraje) contemplados en esta práctica.