Práctica 3.- Programación mixta C-asm x86-64 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 *SystemV AMD64 ABI*.
- 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 *SystemV AMD64*) 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 x86-64.
- Usar con efectividad un depurador como `gdb`/`ddd`, o tal vez incluso depurar usando Eclipse.
- Argumentar la utilidad de los depuradores para ahorrar tiempo de depuración.
- Explicar la convención de llamada *SystemV ABI* para procesadores x86-64 (*AMD64*).
- Recordar y practicar en una plataforma de 64 bits la operación *popcount* (*population count*, peso Hamming, o suma lateral de bits).

2 Convención de llamada *SystemV AMD64 ABI*

En la práctica anterior ya vimos la conveniencia de dividir el código de un programa en varias funciones, para facilitar su legibilidad y comprensión, así como su reutilización. En la Figura 1 se muestra una versión simplificada de la suma de una lista de enteros de 32 bits que ya vimos entonces, destacando estos tres aspectos que ahora nos interesan:

- El resultado se devuelve al programa principal a través del registro EAX.
- Los argumentos (posición y tamaño de la lista) se pasan a la función a través de RBX/ECX.
- La subrutina preserva el valor de RDX (y otros registros).

Puede que el autor de esta función siga la convención de devolver el resultado de sus funciones en el registro A (del tamaño apropiado: RAX/EAX/AX/AL), de esperar que le pasen los argumentos en los registros B, C, D (en ese orden), hasta un total de 3 argumentos, y de garantizar que a la vuelta de la subrutina no se habrán modificado los registros no implicados en la llamada. Seguir siempre la misma convención ayuda a sus usuarios a (memorizar las normas y a) programar las llamadas sin necesidad de consultar continuamente los manuales.

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 denomina ***convención de llamada*** (*calling convention*) al conjunto de alternativas escogidas (para pasar parámetros, devolver resultados, preservar registros, etc.). 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, volátiles*) ***pueden usarse*** directamente en la función, y ***no puede confiarse*** en que conserven su valor después de realizar una llamada a otra función, de ahí el nombre *salva-invocante* (o *volátiles*).
    - Los segundos (*callee-save, salva-invocado, no volátiles*) deberían salvarse a pila antes de modificarlos, para poder restaurar su valor antes de retornar al invocante, quien ***puede confiar*** en que preservarán su valor después de llamar a otra función.
- 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 trabajaremos la convención *SystemV AMD64 ABI*, estándar para arquitecturas x86 de 64 bits en lenguaje C bajo Linux (compilador gcc). Para llamar al *kernel* también se usa una variante de *SystemV* (cambiando RCX por R10). Si programamos funciones ensamblador respetando la convención *SystemV*, 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.

```
# suma.s anterior, simplificada: 1 solo call, exit del kernel, no libC
# 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   0

# SECCIÓN DE CÓDIGO (.text, instrucciones máquina)
.section .text
_start: .global _start          # PROGRAMA PRINCIPAL
       mov     $lista, %rbx
       mov  longlista, %ecx
       call suma
       mov  %eax, resultado     # res = suma(&lista, longlista);

       mov   $60, %rax
       mov    $0, %edi
       syscall                  # _exit(int status);

# SUBRUTINA: int suma(int* lista, int longlista);
# entrada:   1) %rbx = dirección inicio array
#            2) %ecx = número de elementos a sumar
# salida:       %eax = resultado de la suma
suma:
       push    %rdx             # preservar %rdx (índice)
       mov  $0, %eax            # acumulador
       mov  $0, %rdx            # índice
bucle:
       add  (%rbx,%rdx,4), %eax
       inc   %edx
       cmp   %edx,%ecx
       jne    bucle

       pop   %rdx               # restaurar %rdx
       ret
```

Figura 1: suma.s: convención de llamada inventada

La convención de llamada *SystemV* incluye entre muchas otras (es una *ABI* completa) las siguientes especificaciones:

- Los seis primeros parámetros se pasan en los registros RDI, RSI, RDX, RCX, R8, R9.
- Los parámetros adicionales se pasan en pila, de derecha a izquierda; es decir, primero se pasa el último parámetro, después el penúltimo… y por fin el séptimo.
- El espacio reservado en la pila para el paso de parámetros lo libera el código que llama (el lenguaje C soporta funciones con un número variable de argumentos).
- El resultado se devuelve en RAX usualmente (ver Tabla 1). También se puede pasar un puntero como argumento a una función C/C++ para que ésta modifique el valor referenciado.
- Los registros parámetros (RDI…R9), retorno (RAX) y R10-R11 son *salva-invocante*: la función los puede usar, sin tener que preservarlos. Es responsabilidad del invocante (el código que llama a esta función) guardarlos en la pila si desea conservar su valor tras el retorno de esta función.
- Los registros RBX, RBP y R12-R15 son *salva-invocado*: la función debe preservarlos (guardarlos en pila) y restaurarlos antes de retornar, si necesitara modificar su contenido.
- RSP no debe manipularse: la convención *SystemV* asume que funciona como puntero de pila.
- Al llamar a función *variádica* debe indicarse en AL el número de argumentos en registros XMM.

Tipo de variable	Registro
enteros de 128 bits	RDX:RAX
[unsigned] long	RAX
[unsigned] int	EAX
[unsigned] short	AX
[unsigned] char	AL
punteros	RAX
float / double	XMM0 / XMM1

Tabla 1: Devolución de resultados de una función bajo *SystemV AMD64*

Hay otras especificaciones en la convención *SystemV* que no hemos comentado, aunque no afectan a los ejemplos que vamos a estudiar en teoría y prácticas (más información en [3]).

Ejercicio 1: suma_01_S_SystemV

Reprogramar suma.s (Figura 1) conforme a *SystemV*. Ensamblar, enlazar, depurar, y comprobar que sigue calculando el resultado correcto. En la Figura 2 se muestran resaltados los cambios a realizar.

```
# suma.s del Guión anterior
# 1.- cambiando a convención SystemV
#      as -g suma_01_S_SysV.s -o suma_01_S_SysV.o
#      ld    suma_01_S_SysV.o -o suma_01_S_SysV
...
_start: .global _start          # PROGRAMA PRINCIPAL
       mov     $lista, %rdi     # 1er arg. RBX: dirección array lista
       mov  longlista, %esi     # 2º  arg. ECX: número elementos a sumar
       call suma
       mov  %eax, resultado     # res = suma(&lista, longlista);
...
# SUBRUTINA: int suma(int* lista, int longlista);
suma:
       push    %rdx             # preservar %rdx (índice)
       mov  $0, %eax            # acumulador
       mov  $0, %rdx            # índice
bucle:
       add  (%rdi,%rdx,4), %eax
       inc   %edx
       cmp   %edx,%esi
       jne    bucle

       pop   %rdx               # restaurar %rdx
       ret
```

Figura 2: suma_01_S_SysV.s: siguiendo convención *SystemV AMD64 ABI*

Si la *función* suma hubiera necesitado más de 6 argumentos (por ejemplo 10), en el *programa principal* se hubieran introducido en pila los argumentos del 10º al 7º (en orden inverso), se hubiera invocado a suma, y tras el retorno se hubiera limpiado la pila (con 4 pop o mejor con `add $32, %rsp` que no modifica registros destino). Mientras se ejecuta la *función* suma, la dirección de retorno estaría en `(%rsp)` y los argumentos 7…10 se direccionarían como `8(%rsp)…32(%rsp)`. Si fuera necesario espacio para variables locales (porque no hubiera suficientes registros disponibles) la *ABI* prevé el uso de hasta 128 bytes por debajo de `(%rsp)`, direccionables como `-8(%rsp)…-128(%rsp)`, sin necesidad de reservar espacio decrementando RSP (la así llamada ***zona roja*** o *red zone*). Según el caso podría ser necesario preservar registros salva-invocado (realizando los correspondientes push nada más entrar en la función), siendo entonces necesario recuperar su valor justo antes de retornar (realizando los correspondientes pop en orden inverso). Naturalmente, si se va a realizar alguna llamada es necesario proteger las variables locales (y/o los registros salva-invocado y/o los salva-invocante en su caso) decrementando RSP (para que la dirección de retorno no se salve sobre dichas variables o registros preservados). Aunque en la *SystemV AMD64 ABI* el registro RBP no es especial, gcc permite (mediante `-fno-omit-frame-pointer`) seguir ajustando un ***marco de pila*** (o *stack frame*) a la entrada (haciendo `push %rbp / mov %rsp, %rbp`) y destruirlo a la salida (haciendo `mov %rbp, %rsp / pop %rbp`, o el equivalente `leave`, o incluso `pop %rbp` si gcc sabe que RSP ya apunta al marco a liberar).

En la Figura 3 se resume gráficamente la posible utilidad de las distintas partes de un marco de pila, definido como la extensión de pila entre una dirección de retorno y la siguiente. El uso de RBP como puntero de marco es opcional de gcc. Nosotros no lo usaremos, y en realidad la *SysV ABI* intenta que los marcos de pila resultantes sean vacíos y que por tanto en la pila sólo haya direcciones de retorno (mientras sea posible). Para reproducir la parte derecha de la Figura 3, en donde se muestra el marco de pila vacío de la función suma, podemos depurar nuestro programa con la línea de comandos **`gdb -tui --args suma_01_S_SysV uno dos tres`**, poner un punto de ruptura `br _start`, lanzar con `run`, y volcar entonces la pila con `x/8xg $rsp`. Obtenemos la Figura a la derecha, desde la marca "%rsp inicial". Al programa `_start` el cargador ELF Linux le proporciona `int argc` y `char** argv` (y las variables de entorno) en la propia pila. Es fácil extraer el número de argumentos con `p *(long*)$rsp`, y sólo ligeramente más complicado extraer los argumentos del 0 al 3 con `p *(char**)($rsp+8…32)`. Si por curiosidad se prueban los desplazamientos por encima del NULL (48, 56…) se obtienen las variables de entorno "exportadas" al programa recién lanzado.

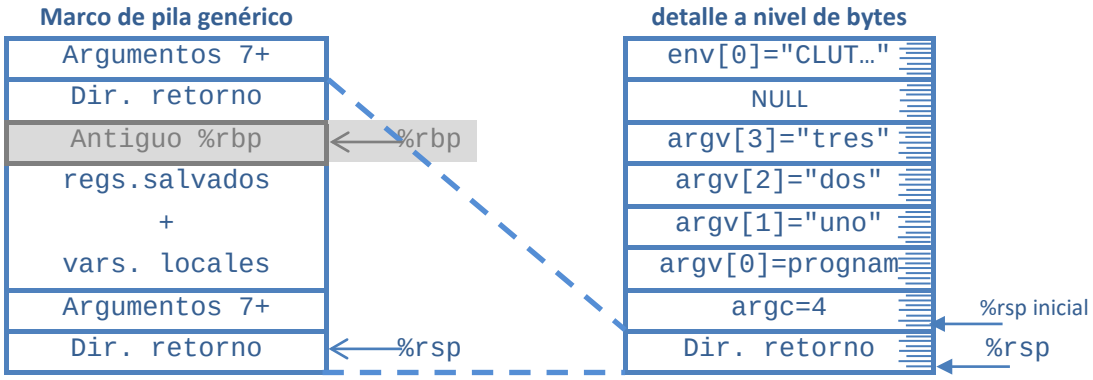

Figura 3: marco de pila genérico, y marco correspondiente a la función suma en el ejemplo

Se puede entonces avanzar hasta la función (`br suma / cont`) para obtener un marco idéntico al de la Figura derecha. Para repetir el comando `x` seguramente se deseará cambiar el foco a la consola gdb (tecleando **`<Ctrl>-x o`**) y pulsar cursor arriba. Se puede comprobar que la dirección de retorno es correcta observando el resultado de `disas *(void**)$rsp` o, alternativamente, `disas _start` (sirve solo porque nuestro ejemplo es muy corto) o, alternativamente, consultando `layout asm/src`.

Ejercicio 2: suma_02_S_libC

La ventaja de usar la convención *SystemV* es que podemos inter-operar con otras funciones conformes a *SystemV*, 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 (así evitaremos tener que depurar para comprobar la corrección del resultado), y sustituyendo la llamada directa al *kernel* Linux por el correspondiente *wrapper* libC (ya puestos, usamos libC para todo). Ensamblar, enlazar, ~~depurar~~, y comprobar que sigue calculando el resultado correcto. En la Figura 4 se resaltan las líneas que deben modificarse, y aparece como comentario el comando utilizado para enlazar con la librería C.

```
# suma.s del Guión anterior
# 1.- cambiando a convención SystemV
# 2.- añadiéndole printf() y cambiando syscall por exit()
#      as -g suma_02_S_libC.s -o suma_02_S_libC.o
#      ld    suma_02_S_libC.o -o suma_02_S_libC \
#             -lc -dynamic-linker /lib64/ld-linux-x86-64.so.2

.section .data
...
resultado:    .int 0
formato:      .asciz "resultado = %d = 0x%x hex\n"
                                  # formato para printf() libC
.section .text
_start: .global _start            # PROGRAMA PRINCIPAL
        mov     $lista, %rdi
        mov  longlista, %esi
        call suma
        mov  %eax, resultado      # res = suma(&lista, longlista);

        mov   $formato, %rdi
        mov  resultado, %esi      # podría haber sido %eax
        mov  resultado, %edx
        mov         $0, %eax
        call printf               # printf(formato, resultado, resultado);

        mov         $0, %edi
        call exit                 # exit(status);

suma:
...
```

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 los registros apropiados según la posición y tamaño del argumento. Tras la llamada, si en RAX hay algún valor de retorno se salva o utiliza como se desee. Si es una llamada a función *variádica* (como *printf*), se indica en AL el número de argumentos en registros XMM.

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 64 bits, en nuestro caso).

Ejercicio 3: suma_03_SC

La ventaja de usar la convención *SystemV* es que podemos inter-operar con otras funciones conformes a *SystemV*. 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 5 se muestran las líneas que deben modificarse, y los comandos utilizados para compilar, ensamblar y enlazar los dos módulos.

```
# MODULO suma_03_SC_s.s:  suma.s del Guión anterior
# 1.- cambiando a convención SystemV
```

```
# 2.- añadiendo printf() y cambiando syscall por exit()
# 3.- extrayendo suma a módulo C para linkar
#      gcc -Og -g -c suma_03_SC_c.c
#      as      -g    suma_03_SC_s.s -o suma_03_SC_s.o
#      ld            suma_03_SC_c.o    suma_03_SC_s.o -o suma_03_SC \
#             -lc   -dynamic-linker /lib64/ld-linux-x86-64.so.2

.section .data
...
.section .text
...
       mov          $0, %edi
       call exit                    # exit(status);
# suma: pasada a módulo C
```

```
//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 5: 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 gdb -tui, comprobando que al pasar a lenguaje C los argumentos formales adquieren el valor de los parámetros actuales pasados por registro.

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 6. 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 anterior
# 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 6: 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 `gdb -tui`, 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. Comprobar usando `objdump` sobre el objeto C y el ejecutable lo afirmado sobre direcciones a 0.

Ejercicio 5: suma_05_C

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

```c
// 5.- gcc -Og -g suma_05_C.c -o suma_05_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= 0;

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 7: 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 `gdb -tui`.

Ejercicio 6: suma_06_CS

Volver a pasar la función `suma` a un módulo ensamblador separado. En la Figura 8 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 anterior 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);   // extern opcional

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

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

```asm
# MODULO suma_06_SC_s.s
# ...
# 5.- entero en C
# 6.- volviendo a sacar la suma a ensamblador
#      gcc -Og -g    suma_06_CS_c.c    suma_06_CS_s.s -o suma_06_CS
#
#      gcc -Og -g -c suma_06_CS_c.c
#      as      -g    suma_06_CS_s.s -o suma_06_CS_s.o
#      gcc           suma_06_CS_c.o    suma_06_CS_s.o -o suma_06_CS
#
#      ld            suma_06_CS_c.o    suma_06_CS_s.o -o suma_06_CS \
#                                       /usr/lib/x86*/crt?.o -lc     \
#                                       -dynamic-linker /lib64/ld-linux*

.section .text
suma: .global suma
      mov  $0, %eax             # acumulador
      mov  $0, %rdx             # índice
bucle:
      ...
```

Figura 8: suma_06_CS: programa C llamando a función asm *SystemV*

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 (y objeto) 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. Se ofrecen las instrucciones alternativas sólo para demostrar que pueden seguir usándose `as` y `ld`. 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 `gdb -tui`.

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. Notar que nuestro código ensamblador asume una serie de hechos (*¿cuáles?*) que le hace inferior en calidad al generado por `gcc`.

Compilar, ejecutar, y comprobar que sigue calculando el resultado correcto. En la Figura 9 se muestra el fuente resultante.

```
// 7.- gcc -Og -g suma_07_Casm.c -o suma_07_Casm

#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= 0;
```

```
int suma(int* array, int len)
{
//  int  i,   res=0;             // gcc7.3 -Og al compilar este código C
//  for (i=0; i<len; i++)        //   usaba jump-in-the-middle, add $1,
//      res += array[i];         //   índice 32b con signo, y repz retq
//  return res;                  //   comprobarlo con https://godbolt.org

  asm("mov  $0, %eax       \n"    // EAX – gcc salva-invocante
"      mov  $0, %rdx       \n"    // RDX – gcc salva-invocante
"bucle:                    \n"
"      add  (%rdi,%rdx,4), %eax\n"
"      inc   %edx          \n"
"      cmp   %edx,%esi     \n"
"      jne    bucle          "

// La sintaxis extendida incluiría:
//   :                     // output
//   :                     // input
//   : "cc",               // clobber
//     "rax","rdx"         // en este caso, la hemos comentado
    );                     // y nos ahorramos muchos %% arriba
}

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

Figura 9: 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 *SystemV*, 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.

Consultar el código ensamblador generado por `gcc –S -Og [-fno-asynchronous-unwind-tables]` para este ejemplo. Localizar la función `suma` y observar cómo marca gcc las instrucciones ensamblador insertadas. Realmente habría que decir "*la instrucción*" insertada. Observar que en nuestro fuente C la primera línea ASM no lleva tabulación a la izquierda y la última no lleva retorno de carro a la derecha, y sin embargo el listado ensamblador generado por gcc está perfectamente indentado. Eliminar algunas tabulaciones y/o añadir retorno de carro a la última línea, observar el código ensamblador generado, y argumentar que `asm()` está pensada en principio para una sola línea de ensamblador.

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` [8] y su *Inline assembly HOWTO* [9] 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* [10] y los tutoriales *SourceForge* [11]) sobre esta temática. Para nuestros objetivos, seguramente nos baste conocer las restricciones más básicas:

Restricción	Registro	Restricción	Operando
a	RAX	**m**	**operando de memoria**
b	RBX	q	registros con parte “L”
c	RCX	**r**	**registros uso general**
d	RDX	g	registro (r) o memoria (m)
S	RSI	J	valor inmediato 0..63 (despl-rotación)
D	RDI	**i**	**valor inmediato entero**
A	RDX:RAX	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 10 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" ((long)i),        // input
       [a]  "r"   (array)
//   : "cc"                       // clobber
    );
    return res;
}
```

Figura 10: suma_08_Casm: inline-asm con restricciones

Notar que para redactar este código *inline* no es necesario conocer la convención *SystemV*, 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 (%rbx,%rdx,4),%eax`**" con tal de que en RBX 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.

El *typecast* del entero `i` a `(long)` es necesario para que la restricción entienda que lo necesitamos en un registro de 64 bits apropiado para direccionar a memoria. Se puede comprobar el error obtenido si se elimina el *typecast*.

Consultar el código ensamblador generado por `gcc -S –Og [-fno-asynchronous-unwind-tables]` para este ejemplo. Localizar la función `suma` y observar cómo satisface `gcc` la restricción de `[i]` mediante una instrucción `movslq`. Cambiando el *typecast* a `(long)(unsigned)`puede evitarse que `gcc` use `movslq` (aunque usa `mov`), y elevar la optimización a `-O` basta para eliminar también el `mov`.

Ejercicio 9: suma_09_Casm

El ensamblador generado por `gcc -O` es parecido a lo que hubiera redactado un humano, aunque con ciertas sofisticaciones (como la copia del test o el prefijo `rep ret`), en las que no se nos ocurrió pensar. En principio se podría dudar si esas sofisticaciones son costosas en tiempo de ejecución, o son inocuas.

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, 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 11 se muestra el programa sugerido.

```
#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];            // traducir sólo esta línea a ASM
    asm("add (%[a],%[i],4),%[r]"
     : [r] "+r"     (res)          // output-input
     : [i]  "r" ((long)i),         // input
       [a]  "r"   (array)
//   : "cc"                        // clobber
    );
    return res;
}

int suma3(int* array, int len)
{
  asm("mov  $0, %%eax         \n"     // EAX - gcc salva-invocante
"       mov  $0, %%rdx         \n"     // RDX - gcc salva-invocante
"bucle:                        \n"
"       add  (%%rdi,%%rdx,4), %%eax\n"
"       inc   %%edx            \n"
"       cmp   %%edx,%%esi      \n"
"       jne    bucle             "
     :                                 // output
     :                                 // input
     : "cc",                           // clobber - como usamos sintaxis extendida
       "eax","rdx"                     // hay que referirse a los registros con %%
    );
}

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 11: suma_09_Casm: esqueleto de programa para comparar tiempos de ejecución

Notar que se ha introducido una ligera variante en la versión 3, y que cuando se usa la sintaxis extendida, los registros se referencian como %%<reg>. Cuando se usa la sintaxis básica, basta con %<reg>. En este caso era además innecesario: se puede eliminar la especificación de *clobbers* y toda la sintaxis extendida (pudiendo volver a escribir %eax) y el ensamblador generado es idéntico.

Notar que el motivo para no poner un mayor tamaño de *array* 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). Seguramente será conveniente compilar con optimización `-O` en lugar de para depuración `-Og -g`, para obtener tiempos más reproducibles.

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.

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