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).

Prácticas de Estructura de Computadores, por Javier Fernández y Mancia Anguita, licencia CC BY-NC-ND.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/

- 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

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

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*

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

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.

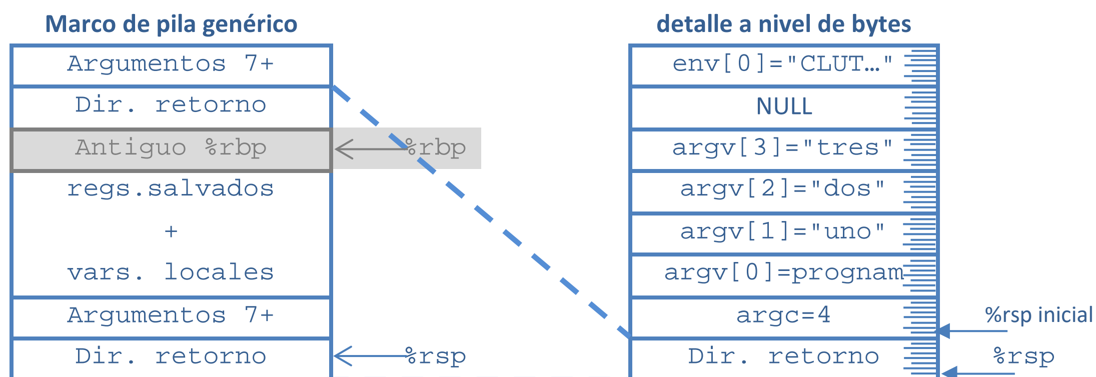


Figura 3: marco de pila genérico, y marco correspondiente a la función suma en el ejemplo

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).

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

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
```

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

```
# 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

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

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.

```
// 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.

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/

```
//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);
}
```

```
# 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:

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
CC BY-NC-ND

- 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;
```

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

```
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"

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

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**		

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
CC BY-NC-ND

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)
{
```


Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

```
    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

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

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.

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

4 Trabajo a realizar

Nos interesaría experimentar con algún ejemplo que permita obtener ventaja sobre `gcc`, y que al mismo tiempo sea lo suficientemente sencillo como para estudiarlo y programarlo en pocas sesiones de prácticas. O aún mejor, que no requiera estudio adicional, porque ya lo hayamos estudiado.

Estas condiciones las cumple por ejemplo el cálculo del peso Hamming o *"population count"*, ya visto en clase de teoría, para el cual existe una instrucción SSSE3 `pshufb` y otra SSE4.2 `popcount` cuyo uso `gcc` no podrá deducir a partir de nuestro código C.

Se trata por tanto de programar varias versiones de una función que sume los pesos Hamming (nº de bits activados) de todos los elementos de un *array*, con y sin ensamblador en-línea, y comprobando siempre la corrección del resultado calculado. Para esto último, podemos consensuar algunos ejemplos pequeños de prueba, cuyo resultado correcto pueda calcularse a mano. Pero para que el tiempo de medición sea apreciable (y reproducible) tendremos que usar *arrays* de mayor tamaño, y para conocer entonces el resultado correcto hará falta una fórmula aplicable a los datos de entrada utilizados.

Iremos sugiriendo las distintas versiones, dando pistas sobre la idea general que debe implementar el código C y/o el tramo de código ASM, y sobre las instrucciones máquina y restricciones a utilizar, al objeto de guiar, orientar y acelerar tanto la localización y lectura de información en los apuntes de clase y en el manual del repertorio de instrucciones como la programación de los tramos `asm()`.

Cada vez que se desarrolle una nueva versión debería comprobarse su corrección con los ejemplos de tamaño pequeño y con el de tamaño grande, procediendo a su revisión y/o depuración si no produjera el resultado correcto. Ofreceremos sugerencias de compilación condicional y los correspondientes comandos del *shell bash* para automatizar esta tarea lo máximo posible.

Tras finalizar el desarrollo de las versiones, deberíamos cronometrarlas de forma justa y equitativa. Repetiremos 10 veces la ejecución del ejemplo grande para promediar los tiempos de ejecución de cada versión, y repetiremos el estudio para distintos niveles de optimización. Los tiempos promediados se pueden presentar en una gráfica del paquete ofimático disponible en Ubuntu (hoja de cálculo Calc). Daremos también sugerencias de compilación condicional y los correspondientes comandos del *shell bash* para automatizar las mediciones. También daremos indicaciones sobre el formato gráfico deseado.

Se propondrá comenzar con las funciones C estudiadas en clase (adaptadas a calcular el *popcount* del *array* completo), primero la más costosa, después la más eficiente. Propondremos entonces una versión ASM intentando mejorarla, como ejercicio preparatorio. Seguiremos con una ingeniosa propuesta de nuevo en lenguaje C, y por último recurriremos a las citadas instrucciones del repertorio multimedia.

Según la temporización de cada curso, se procurarán realizar guiadamente (como Seminario Práctico) los Ejercicios 1-6 aproximadamente (incluso 7-9 si sobrara tiempo). Aunque no diera tiempo a tanto, comprender los programas mostrados y ejercitarse en el uso de las herramientas son competencias que cada uno debe conseguir personalmente.

Para aprender el funcionamiento de nuevas instrucciones (sean o no del repertorio SSE) basta con leer el manual de Intel y probarlas en la propia sentencia ASM *inline* (y depurarlas con `gdb -tui`, si no produjeran el resultado esperado).

Al objeto de facilitar el desarrollo progresivo de la práctica, se sugiere realizar en orden las siguientes tareas:

4.0 Repasar los apuntes de clase

Esta práctica es posterior o simultánea al estudio en clase de teoría de diversos conceptos relevantes [1], como por ejemplo: códigos de condición (Tema 2.2, tr. 3-11), bucles (tr. 22-33), estructura de la pila (Tema 2.3, tr. 8-11), convenciones de llamada (tr. 19-21, tr. 44-49), punteros y variables locales (tr. 50-51), declaración y acceso a *arrays* (Tema 2.4, tr. 3-14). Se recomienda su estudio detallado.

Prácticas de Estructura de Computadores, por Javier Fernández y Mancia Anguita, licencia CC BY-NC-ND.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/

4.1 Calcular la suma de bits de una lista de enteros sin signo

Utilizar el programa `suma_09` de la Figura 11 como esqueleto para cronometrar diversas versiones de una función que sume los bits (peso Hamming, *popcount*) de los elementos de una lista de N enteros. Notar que la suma puede llegar a ser 32*N, si todos valieran 2^{32}-1 (y por consiguiente tuvieran activados todos los bits). Concluir que basta calcular la suma en un entero, para cualquier valor práctico de N. *¿Cómo de grande puede ser N en dicho peor caso?*

Para tener alguna posibilidad de detectar errores en nuestro código, lo comprobaremos con algunos ejemplos sencillos como los siguientes:

- `unsigned lista[SIZE]={0x80000000, 0x00400000, 0x00000200, 0x00000001};`
- `unsigned lista[SIZE]={0x7fffffff, 0xffbfffff, 0xfffffdff, 0xfffffffe,`
  `0x01000023, 0x00456700, 0x8900ab00, 0x00cd00ef};`
- `unsigned lista[SIZE]={0x0        , 0x01020408, 0x35906a0c, 0x70b0d0e0,`
  `0xffffffff, 0x12345678, 0x9abcdef0, 0xdeadbeef};`

Los resultados deberían ser 4, 156, 116, respectivamente. Notar que la lista se declara sin signo para que los desplazamientos (que previsiblemente tendremos que utilizar) no dupliquen ningún bit de signo. *¿Por qué?*

Para poder comparar los tiempos de ejecución de distintos programas (de distintos usuarios) en el laboratorio (con los mismos ordenadores), y también de un mismo programa (de un estudiante) en distintos ordenadores (laboratorio y portátil personal), necesitaremos acordar algún ejemplo de tamaño mayor, como por ejemplo SIZE=2^{20} elementos, inicializados en secuencia desde 0 hasta SIZE-1. Calcular qué peso Hamming tiene ese ejemplo (en función de SIZE) y modificar la fórmula del programa `suma_09` acordemente. Para ello podemos (en realidad debemos, si es que queremos calcularlo con una fórmula) aprovechar razonamientos específicos para el *array* en cuestión. Pista: ¿se puede aplicar algún razonamiento al bit 0 de todos los elementos? ¿Y al bit 19? ¿Y a los intermedios? *¿Qué fórmula sale?*

Recomendamos usar un esquema de compilación condicional similar al usado en la práctica anterior, para incorporar la definición de los tests y el resultado esperado.

```
#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;

#ifndef TEST
#define TEST 5
#endif

/* -------------------------------------------------------------------- */
  #if   TEST==1
/* -------------------------------------------------------------------- */
  #define SIZE 4
  unsigned lista[SIZE]={0x80000000, 0x00400000, 0x00000200, 0x00000001};
  #define RESULT 4
/* -------------------------------------------------------------------- */
#elif   TEST==2
/* -------------------------------------------------------------------- */
  ...
/* -------------------------------------------------------------------- */
#elif   TEST==4 || TEST==0
/* -------------------------------------------------------------------- */
  #define NBITS 20
  #define SIZE (1<<NBITS)       // tamaño suficiente para tiempo apreciable
  unsigned lista[SIZE];         // unsigned para desplazamiento derecha lógico
  #define RESULT ( ? * ( ? << ?-1 ) )          // pistas para deducir fórmula
/* -------------------------------------------------------------------- */
#else
        #error "Definir TEST entre 0..4"
#endif
/* -------------------------------------------------------------------- */

int popcount1(unsigned* array, size_t len)
{
    ...
}
```


Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

```
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);
#if TEST==0
    printf(    "%ld" "\n",      tv_usecs);
#else
    printf("resultado = %d\t", resultado);
    printf("%s:%9ld us\n", msg, tv_usecs);
#endif
}

int main()
{
#if TEST==0 || TEST==4
    size_t i;                               // inicializar array
    for (i=0; i<SIZE; i++)
        lista[i]=i;
#endif

    crono(popcount1 , "popcount1 (lenguaje C -       for)");
    crono(popcount2 , "popcount2 (lenguaje C -     while)");
    crono(popcount3 , "popcount3 (leng.ASM-body while 4i)");
    crono(popcount4 , "popcount4 (leng.ASM-body while 3i)");
    crono(popcount5 , "popcount5 (CS:APP2e 3.49-group 8b)");
    crono(popcount6 , "popcount6 (Wikipedia- naive - 32b)");
    crono(popcount7 , "popcount7 (Wikipedia- naive -128b)");
    crono(popcount8 , "popcount8 (asm SSE3 - pshufb 128b)");
    crono(popcount9 , "popcount9 (asm SSE4- popcount 32b)");
    crono(popcount10, "popcount10(asm SSE4- popcount128b)");

#if TEST != 0
    printf("calculado = %d\n", RESULT);
#endif

    exit(0);
}
```

Figura 12: popcount.c: esqueleto de suma_09 con sugerencias de compilación condicional para los tests

Observar que `lista` es ahora `unsigned`, que las definiciones de `lista`, `SIZE` y `RESULT` varían con el test escogido, que los tests pequeños (1, 2, 3) se definen fácilmente mediante inicializador, y que el test grande (4) es mejor definirlo programáticamente. Observar que se ha definido un test 0 igual al 4 pero que imprime tan sólo el valor numérico del tiempo cronometrado. Posteriormente se explicará por qué. En el esquema quedan tan sólo por rellenar las distintas versiones, que procedemos a sugerir.

Realizar una **primera** y **segunda** versiones C parecidas las vistas en clase (Tema 2.2, tr.30 y 29, pero aplicadas a enteros de 4 bytes, y de hecho a un *array* de ellos), recorriendo en ambas el *array* con un bucle `for`, y recorriendo los bits con bucle `for` (1ª versión, p.30) o con un bucle `while` (2ª versión, p.29), aplicando ***en ambos*** máscara 0x1 y desplazamiento a la derecha (ambas con el cuerpo de la p.29), para ir extrayendo y acumulando los bits (ver [1]). Compararíamos los tiempos para comprobar que a veces se pueden obtener buenas ganancias pensando bien las cosas en C, sin necesidad de usar ASM.

```
#define WSIZE 8*sizeof(long)
long pcount_for(unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Figura 13: CS:APP [1], T-2.2, tr.30 cuerpo como tr.29

```
long pcount_while(unsigned long x)
{

  long result = 0;
  while (x)
  {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Figura 14: CS:APP [1], Tema 2.2, transparencia 29

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

Notar que la variable `result` vista en clase puede continuar usándose para seguir sumando los bits de otro elemento. Notar que preferimos llamar "1ª versión" a la del bucle `for` (tr.30), que previsiblemente tendrá peores prestaciones que el bucle `while` (incluso usando el mismo desplazamiento a derecha), porque debe iterar siempre `8*sizeof(int)` veces independientemente del nº de bits activados.

Realizar una **tercera** y **cuarta** versiones traduciendo el bucle interno `while` por un tramo de unas 4-5 ó 5-7 líneas ensamblador respectivamente (sin contar-contando etiquetas) que incluyan la instrucción ADC que ya utilizamos en la práctica anterior. La idea consiste en que como el bit desplazado acaba en el acarreo (consultar el manual de SHR), de ahí mismo lo podemos sumar y nos ahorramos aplicar la máscara. En principio, se pensaría que debe suponer alguna mejora sobre la 2ª versión. El resultado nos decepcionará. Por facilitar el desarrollo de este primer ejemplo, propuesto como ejercicio preparatorio, indicamos unas posibles restricciones:

```
    for (i=0; i<len; i++) {
      x = array[i];
      asm("\n"
"ini3:                  \n\t"          // seguir mientras que x!=0
       "shr %[x]        \n\t"          // LSB en CF
       ...
       : [r]"+r" (result)              // e/salida: añadir a lo acumulado por el momento
       : [x] "r" (x)       );          // entrada:  valor elemento
    }
```

La **tercera** versión consta de otras 3 instrucciones ASM (aparte de la etiqueta e instrucción mostradas, 4-5 líneas en total), ya hemos mencionado que se usa ADC, y es obvio que si hay etiqueta `ini3:` es porque debe haber un salto hacia atrás. Ya hemos estudiado en clase cómo hace `gcc` para comparar con $0, y recomendamos encarecidamente recordar que no se usa CMP. *¿Por qué?* *(ver Tema 2.2 tr.7)*

La **cuarta** versión consta de 5 instrucciones ASM y 1 (ó 2) etiqueta(s) `ini4:` (y `fin4:`), (5-6-7 líneas en total), que se obtienen al intentar evitar la instrucción TEST aprovechando que SHR también afecta al flag ZF. En un primer intento se podría saltar condicionalmente al final tras SHR, y aprovechando que los saltos no modifican los flags (particularmente CF), acumular el acarreo y volver al principio. Eso produce un bucle de 4 instrucciones algo menos eficiente que el anterior. En un segundo intento, podría recordarse la instrucción CLC para comenzar el segmento ASM de la siguiente forma:

```
      asm("\n\t"
       "clc             \n"            // CLC para poder empezar por ADC
"ini4:                  \n\t"
       "adc $0, %[r]    \n\t"          // ahora sale bucle de 3 instrucciones
       ...
      );
```

La sentencia ASM tiene las mismas restricciones que la tercera versión. Tanto el primer como el segundo intento necesitan una última instrucción ADC repetida para capturar el último bit. Aunque se obtenga un bucle con un 25% menos de instrucciones, comprobaremos que las prestaciones no mejoran proporcionalmente. De hecho, seguimos sin vencer a `gcc`.

Implementar como **quinta** versión la solución que aparece en el libro de clase [1] 2ª Ed., problema 3.49, resuelto en la página 364 (lenguaje C). Viene resuelto para 64bits (y un único elemento), bastaría con adaptarlo a 32bits (y a un *array* completo). Recordar que preferimos usar `size_t` para los índices.

```
long fun_c(unsigned long x) {
    long val = 0;
    int i;
    for (i = 0; i < 8; i++) {
        val += x & 0x0101010101010101L;
        x >>= 1;
    }
    val += (val >> 32);
    val += (val >> 16);
    val += (val >> 8);
    return val & 0xFF;
}
```

Figura 15: CS:APP2e [1], Problema 3.49, resuelto p.364

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

El código se basa en aplicar sucesivamente (8 veces) la máscara 0x0101… a cada elemento, para ir acumulando los bits de cada byte en una nueva variable `val` (no podemos acumular los bits de uno en uno en `result` como antes) y sumar en árbol los 4B. Seguramente uno no se espera una mejora tan drástica en las prestaciones, lo cual nos debe hacer reflexionar en lo importante que es escoger bien la idea básica de un algoritmo. Es muy difícil ganar a `gcc` cuando compila un programa C bien pensado. *¿Qué significa "acumular los bits de cada byte"? ¿Qué significa "sumar en árbol"? ¿Por qué hace falta la nueva variable* `val`*?*

Reflexionando sobre la quinta versión se llega a la solución "*naive*" propuesta por la Wikipedia [5], consistente en reemplazar el bucle lineal inicial `for(i<8)` por más sumas en árbol de granularidad más fina, empezando por bits sueltos. Nuestra **sexta** versión será esta propuesta "*naive*", adaptada de nuevo a *array* de enteros. *¿Qué significa "bucle lineal"? ¿Por qué funciona este algoritmo?*

```
//types and constants used in the functions below
//uint64_t is an unsigned 64-bit integer type (defined in C99 version of C language)
const uint64_t m1  = 0x5555555555555555; //binary: 0101...
const uint64_t m2  = 0x3333333333333333; //binary: 00110011..
const uint64_t m4  = 0x0f0f0f0f0f0f0f0f; //binary:  4 zeros,  4 ones ...
const uint64_t m8  = 0x00ff00ff00ff00ff; //binary:  8 zeros,  8 ones ...
const uint64_t m16 = 0x0000ffff0000ffff; //binary: 16 zeros, 16 ones ...
const uint64_t m32 = 0x00000000ffffffff; //binary: 32 zeros, 32 ones

//This is a naive implementation, shown for comparison,
int popcount64a(uint64_t x)
{
    x = (x & m1 ) + ((x >>  1) & m1 ); //put count of each  2 bits into those  2 bits
    x = (x & m2 ) + ((x >>  2) & m2 ); //put count of each  4 bits into those  4 bits
    x = (x & m4 ) + ((x >>  4) & m4 ); //put count of each  8 bits into those  8 bits
    x = (x & m8 ) + ((x >>  8) & m8 ); //put count of each 16 bits into those 16 bits
    x = (x & m16) + ((x >> 16) & m16); //put count of each 32 bits into those 32 bits
    x = (x & m32) + ((x >> 32) & m32); //put count of each 64 bits into those 64 bits
    return x;
}
```

Figura 16: Wikipedia, popcount, C "*naive*" implementation [5]

Por un conocimiento de arquitectura que adquiriremos posteriormente (la versión 8 con múltiples instrucciones SSE3 gana a la versión 9 con una única instrucción SSE4) se nos ocurre pensar que también se puede mejorar esta versión leyendo en tamaños superiores (y desenrollando el bucle en un factor 4x). Nuestra **séptima** versión tiene por tanto el siguiente aspecto:

```
// Wikipedia popcount "naive" implementation *** 128b ***
int popcount7(unsigned* array, size_t len)
{
  size_t i;
  unsigned long x1,x2;
  int result=0;

  const unsigned long m1  = 0x5555555555555555; //binary: 0101...
  ...
  const unsigned long m32 = 0x00000000ffffffff; //binary: 32 zeros, 32 ones

  if (len & 0x3) printf("leyendo 128b pero len no múltiplo de 4\n");

  for (i=0; i<len; i+=4)
  {
    x1 = *(unsigned long*) &array[i  ];
    x2 = *(unsigned long*) &array[i+2];

    x1 = (x1 & m1 ) + ((x1 >>  1) & m1 ); //put count of each  2 bits into those  2 b
    ...
    x1 = (x1 & m32) + ((x1 >> 32) & m32); //put count of each 64 bits into those 64 b

    x2 = (x2 & m1 ) + ((x2 >>  1) & m1 );
    ...
    x2 = (x2 & m32) + ((x2 >> 32) & m32);

    result+= x1+x2;
  }
  return result;
}
```

Figura 17: aumento de tipo y desenrollado de bucle para mejorar prestaciones

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

Para una **octava** versión, podemos buscar con Google qué otros métodos han usado algunos entusiastas para calcular el *popcount*, e implementar alguno de ellos (ver [6], instrucción SSSE3 PSHUFB). Compararíamos con el crono de la versión anterior para ver cuánto se gana por pasar del repertorio normal a SSSE3. La Figura 18 y los párrafos tras ella se dedican a explicar el método [6].

Una **novena** versión consistiría en sustituir todo el bucle interno `while` por la instrucción SSE4 POPCNT. Compararíamos con el crono de la versión 8 para ver cuánto se gana por pasar del repertorio SSSE3 a SSE4. Como hemos comentado anteriormente, los resultados nos hacen sospechar de leer sólo 32 bits.

La **décima** y última versión intenta mejorar la anterior realizando dos lecturas de 64 bits (como en la versión 7) y dos *popcount*, realizando por tanto la cuarta parte de iteraciones. Todos los ejemplos con repertorio multimedia se ofrecen prácticamente resueltos.

```
// Versión SSE4.2 (popcount)
int popcount9(unsigned* array, size_t len){
    size_t i;
    unsigned x;
    int val, result=0;


    for (i=0; i<len; i++)
    {
      x = array[i];
      asm("popcnt %???, %???"


        : [val] "=r" (val)
        :   [x]  "r" (x)


      );
      result += val;
    }
    return result;
}
```

```
// popcount 128bit p/mejorar prestaciones
int popcount10(unsigned* array, size_t len){
    size_t i;
    unsigned long x1,x2;
    long val; int result=0;
    if (len & 0x3) printf(
"leyendo 128b pero len no múltiplo de 4\n");
    for (i=0; i<len; i+=4) {
      x1 = *(unsigned long*) &array[i  ];
      x2 = *(unsigned long*) &array[i+2];
      asm("popcnt %????, %???? \n"
          "popcnt %[x2], %%r10 \n"
          "add    %????, %???? \n"
       : [val]"=&r" (val)
       :  [x1]  "r" (x1),
          [x2]  "r" (x2)
       :  "r10"
      );
      result += val;
    }
    return result;
}
```

```
// Versión SSSE3 (pshufb) web http:/wm.ite.pl/articles/sse-popcount.html
int popcount8(unsigned* array, size_t len){
    size_t i;
    int val, result=0;
    int SSE_mask[] = {0x0f0f0f0f, 0x0f0f0f0f, 0x0f0f0f0f, 0x0f0f0f0f};
    int SSE_LUTb[] = {0x02010100, 0x03020201, 0x03020201, 0x04030302};
                   //    3 2 1 0     7 6 5 4    1110 9 8    15141312
    if (len & 0x3) printf("leyendo 128b pero len no múltiplo de 4\n");
    for (i=0; i<len; i+=4)     {
    asm("movdqu    %[x], %%????\n\t"
        "movdqa  %%xmm0, %%???? \n\t"  //    x: two copies xmm0-1
        "movdqu    %[m], %%???? \n\t"  // mask: xmm6
        "psrlw      $4 , %%???? \n\t"
        "pand    %%xmm6, %%xmm0 \n\t"  //; xmm0 – lower  nibbles
        "pand    %%xmm6, %%xmm1 \n\t"  //; xmm1 – higher nibbles

        "movdqu    %[l], %%???? \n\t"  //; since instruction pshufb modifies LUT
        "movdqa  %%xmm2, %%???? \n\t"  //; we need 2 copies
        "pshufb  %%xmm0, %%xmm2 \n\t"  //; xmm2 = vector of popcount lower nibbles
        "pshufb  %%xmm1, %%xmm3 \n\t"  //; xmm3 = vector of popcount upper nibbles

        "paddb   %%????, %%xmm3 \n\t"  //; xmm3 – vector of popcount for bytes
        "pxor    %%????, %%xmm0 \n\t"  //; xmm0 =  0,0,0,0
        "psadbw  %%????, %%xmm3 \n\t"  //; xmm3 = [pcnt bytes0..7|pcnt bytes8..15]
        "movhlps %%????, %%xmm0 \n\t"  //; xmm0 = [      0      |pcnt bytes0..7 ]
        "paddd   %%????, %%xmm0 \n\t"  //; xmm0 = [  not needed  |pcnt bytes0..15]
        "movd    %%xmm0, %[val]     "
        : [val]"=r" (val)
        :  [x]  "m" (array[i]),
           [m]  "m" (SSE_mask[0]),
           [l]  "m" (SSE_LUTb[0])
      );
        result +=  val;
    }
    return result;
}
```

Figura 18: popcount8: función para cálculo SSSE3 del peso Hamming (algunos registros XMM omitidos)

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

Para comprender el método SSSE3 propuesto en la web [6] conviene consultar el dibujo que acompaña a la página de manual de PSHUFB, la operación de ***baraje*** más corta del repertorio SSSE3. Los registros XMM (XMM0-XMM7) son de 128bits, y están pensados para almacenar en paralelo varios elementos, por ejemplo 4 enteros de 32bits (4ints x 2^5 bits/int = 2^7 bits = 128bits), 8 shorts, o 16 chars (2^4 x 2^3). La operación de ***baraje*** permite "barajar" esos elementos (como si fueran cartas de una baraja), indicando en un primer argumento el baraje deseado (en cada posición se indica el nº del dato deseado en esa posición) y en un segundo argumento los datos a barajar. Es *fundamental* advertir que en el baraje no se indica a qué posición va cada elemento (podríamos equivocarnos y dejar huecos), sino qué elemento termina en esa posición. Por fijar conceptos, la instrucción `pshufb %xmm1, %xmm2` baraja los 16 bytes de XMM2, colocando el byte i (i=0..15) en todos los bytes de XMM1 donde ponga i. Esto nos permite repetir elementos y que otros se queden fuera del resultado, lo cual puede parecer anti-intuitivo y poco relacionado con barajas de cartas. Notar también que los datos de baraje (XMM2) son sobrescritos. Conviene re-leer este párrafo junto con el dibujo del manual hasta comprender la operación de baraje.

La idea para acelerar el cálculo del *popcount* consiste en pre-calcular cuántos bits tiene activados cada número (hasta un límite dado, por ejemplo de 8 bits: 0 tiene 0bits, 1 y 2 tienen 1bit, 3 tiene 2bits… hasta 255, que tiene 8 bits activados), y usar el propio número como índice en una tabla (más o menos grande según el límite impuesto) en donde se almacenan esos resultados pre-calculados. El *popcount* de un elemento `x=array[i]` (supongamos `x`=255) es entonces `Tabla[x]` (=8). A este tipo de tabla, donde el dato disponible indexa el resultado deseado, se les suele llamar *Tabla de Consulta* (*Look-Up Table, LUT*). Por ejemplo, una paleta de colores indexados es una LUT, porque el código del color se usará como índice.

Siguiendo con el ejemplo `Tabla[array[i]]`, se tarda menos en acceder al elemento 255 de la tabla (obteniendo resultado=8bits) que hacer 8 desplazamientos, máscaras y acumulaciones. El inconveniente es que una tabla tan grande no cabe en un registro XMM. Pero si la limitamos a elementos de 4bits (medio byte, un *nibble*), sí que podemos almacenarla en un registro XMM, en donde caben 16B. De hecho nos sobra más de la mitad de cada byte, porque la LUT sólo necesita 16 elementos de 3bits: 16 porque calcularemos *popcount* de 4bits, y 3 porque el máximo son 4bits activados (0b100). Pero en SSSE3 no existe operación de baraje con 32 nibbles. La operación de baraje más corta opera sobre 16B, y nosotros aprovecharemos sólo la mitad de cada byte.

Se puede recorrer por tanto el *array* de 4 en 4 elementos, cargando 4 enteros en un registro XMM de 128bits (16B), repartiendo sus nibbles entre dos registros XMM (para que todos los índices salgan entre 0..15), barajando con la tabla pre-calculada (LUT) para obtener cuántos bits hay activados en cada nibble, y sumando todos esos números. La máscara y tabla se pueden consultar en la Figura 18.

Explicado paso a paso, el tramo ASM carga 4 enteros en un registro XMM y saca copia en otro XMM. Carga una máscara para quedarse con nibbles inferiores. Desplaza 4b una de las copias, de manera que al aplicar la máscara a ambas copias, resulten separados los nibbles inferiores y superiores en su correspondiente registro XMM. Se cargan entonces dos copias de la LUT y se barajan usando como índices los nibbles, obteniendo los popcount respectivos, como se explicó anteriormente.

El último tramo sirve para acumular todos esos popcount en `val`. PADDB es una suma de bytes, que se usa para reunir las cuentas de nibbles inferiores y superiores. Sumar horizontalmente esas cuentas es más complicado, debiéndose usar PSADBW (instrucción pensada para sumar valores absolutos de diferencias), que produce 2 resultados de 16b, uno en la parte menos significativa y otro en el centro del registro XMM. MOVHLPS sirve para llevar el resultado central a la parte inferior de otro registro XMM, y PADDD sirve para sumar ambos. El resultado final se puede mover a un registro de 32b con MOVD.

Notar que casi todas las restricciones se han indicado en memoria, encargándonos nosotros del movimiento explícito a registros (con MOVDQU, para evitar problemas si los *arrays* resultan no estar alineados a 16B). De esta forma el tramo ASM es virtualmente idéntico al de la web [6]. Se puede usar MOVDQA para mover entre registros XMM. Notar por último que el *array* se recorre de 4 en 4 elementos, y que dicho recorrido y la acumulación son las únicas tareas que se realizan en lenguaje C. Sólo la restricción para `val` se ha indicado en registro de 32b, para optimizar su suma con `result`. El movimiento de los 32b inferiores de un registro XMM a uno de 32b se puede realizar con MOVD.

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

Mediciones: cronometrar las distintas versiones con –O0, –O1 y –O2

Como también nos interesa saber cómo se comporta `gcc` según el nivel de optimización (`-O0`, `-Og`, `-O1`, `-O2`), repetiremos 10 mediciones de tiempo para esos 4 niveles. Se trata por tanto de recompilar 4 veces, repetir 10 mediciones, y organizar los resultados en una gráfica de paquete ofimático (Calc), mostrando los promedios de cada versión (1ª - 10ª) para cada nivel de optimización (0 - 2), tal vez con un gráfico de barras con abscisas bidimensionales versión-optimización (Calc lo denomina *“columnas 3D”*). Conviene que cambie el color de las columnas con la versión de la función, no con la optimización.

En principio se esperaría que cada versión fuera progresivamente mejor, y dentro de cada una, se mejorara con el nivel de optimización. Si alguna versión no siguiera esta tendencia, convendría probar con otro modelo de CPU para ver si es una característica propia del modelo usado. Si varios modelos reprodujeran la anomalía, sería conveniente cambiar de orden las mediciones o realizar otras modificaciones en el código al objeto de descubrir si somos nosotros quienes estamos creando la anomalía. Si no se descubre ningún motivo, se debería consultar el código ensamblador generado para intentar explicar dicho comportamiento.

Recomendaciones

Recordar que siempre se debe comprobar que el resultado es correcto. Una optimización que produce un resultado distinto sólo tiene tres explicaciones: o está mal el programa optimizado, o está mal el original, o están mal ambos.

El esquema de compilación condicional que sugerimos en la Figura 12 nos facilita tanto la comprobación de los ejemplos pequeños definidos al principio, como la realización de las mediciones y su incorporación a una hoja Calc. Considerar los siguientes comandos *shell bash* añadidos al principio del código fuente:

```
// gcc popcount.c -o popcount -Og -g -D TEST=1
/*
  === TESTS === ____________________________________
  for i in 0 g 1 2; do
      printf "__OPTIM%1c__%48s\n" $i "" | tr " " "="
    for j in $(seq 1 4); do
      printf "__TEST%02d__%48s\n" $j "" | tr " " "-"
      rm  popcount
      gcc popcount.c -o popcount -O$i -D TEST=$j -g
        ./popcount
    done
  done
  === CRONOS === ____________________________________
  for i in 0 g 1 2; do
      printf "__OPTIM%1c__%48s\n" $i "" | tr " " "="
      rm  popcount
      gcc popcount.c -o popcount -O$i -D TEST=0
    for j in $(seq 0 10); do
      echo $j; ./popcount
    done | pr -11 -l 22 -w 80
  done
  ___________________________________________________
 */

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

int resultado=0;

#ifndef TEST
#define TEST 5
#endif

...
```

Figura 19: popcount.c: autodocumentando el sistema de test y medición

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

El comando que aparece como primer comentario sería útil para depurar alguna versión que produjera resultado incorrecto con el primer ejemplo (TEST=1). Pudiera incluso suceder que alguna versión sólo produjera resultados erróneos en algún nivel de optimización concreto. Se debería entonces recompilar con ese nivel de optimización (e información de depuración `-g`), comprobar que el error sigue manifestándose, y proceder a depurar con `gdb -tui`.

El segundo comando (etiquetado TESTS en el listado de la Figura 19) barre con dos bucles anidados los cuatro niveles de optimización y los cuatro ejemplos de test, ejecutando para cada combinación el programa recompilado con esas opciones. Haber cuidado la alineación de los resultados al redactar el código fuente nos permite ahora revisar a una formidable velocidad la corrección de todas las versiones de *popcount*, con los diversos tests que habíamos acordado, para los distintos niveles de optimización.

Considerar ahora el siguiente commando *shell bash* para un ejecutable recompilado con TEST=0.

```
for i in $(seq 0 10); do echo $i ; ./popcount; done | pr -11 -l 22 -w 80
```

Consultando la página de manual de `pr`, se deduce que este comando permite realizar las deseadas 10 mediciones y dejarlas escritas en pantalla en un formato listo para hacer *copy-paste* fácilmente a la hoja de cálculo mostrada en el Apéndice 1. En realidad ese comando lanza 11 ejecuciones, por si la primera (o alguna) sale claramente peor que el resto, y pagina los 110 números (10 versiones x 11 mediciones) en las mismas 11 columnas de la hoja Calc. Se han escogido 22 líneas de 80 caracteres como tamaño de página. El comando etiquetado CRONOS en el listado de la Figura 19 permite realizar todas las mediciones a la misma formidable velocidad.

Los comandos para recompilar y lanzar la medición se podrían anotar también en la propia hoja Calc como recordatorio para cuando se desee repetir el experimento.

5 Entrega del trabajo desarrollado

Los distintos profesores de teoría y prácticas acordarán las normas de entrega para cada grupo, incluyendo qué se ha de entregar, cómo, dónde y cuándo.

Por ejemplo, puede que en un grupo el profesor habilite que los estudiantes que cumplan el régimen de asistencia puedan entregar en SWAD, hasta medianoche del domingo de la última semana de esta práctica, el código fuente realizado (`popcount.c`), listo para compilar y reproducir las mediciones usando el comando CRONOS incluido en el propio fuente con la técnica mostrada más arriba, y otro fichero (`popcount.pdf`) con las gráficas de las mediciones de tiempo, con el formato indicado en este guión de prácticas. Puede que en otro grupo el profesor de prácticas proporcione otras instrucciones distintas.

En cualquier caso, **el examen de prácticas** a final del curso (Test TP de 4 puntos) **es el mismo para todos los grupos**, lo cual significa que independientemente del profesor de prácticas, cada estudiante es responsable de comprender todo lo explicado en estos **guiones de prácticas (que son comunes a todos los grupos y son materia de examen)** y desarrollar las habilidades que se ejercitan en los mismos (que también son materia de examen).

6 Bibliografía

[1] Apuntes y presentaciones de clase, y particularmente
Programación Máquina II: Control
sección "Códigos de condición", instrucciones `test/setcc`, p.7-11
sección "Bucles", p.22-33
Libro CS:APP 2ª Ed., Problema 3.49, p.364.
Randal E. Bryant, David R. O'Hallaron: "Computer Systems: A Programmer's Perspective", 2nd-3rd Ed., Pearson, 2011-2016. http://csapp.cs.cmu.edu/

[2] Manuales de Intel sobre IA-32 e Intel64, en concreto el volumen 2: "Instruction Set Reference"
https://software.intel.com/sites/default/files/managed/a4/60/325383-sdm-vol-2abcd.pdf

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
(cc) BY-NC-ND

[3] Wikipedia, convenciones de llamada http://en.wikipedia.org/wiki/Calling_convention
X86 calling conventions http://en.wikipedia.org/wiki/X86_calling_conventions
WikiBook http://en.wikibooks.org/wiki/X86_Disassembly/Calling_Conventions
[4] Wikipedia, extensiones x86 MMX/SSE: http://en.wikipedia.org/wiki/X86#Extensions
MMX http://en.wikipedia.org/wiki/X86#MMX
SSE, SSE2, SSE3, SSSE3, SSE4 http://en.wikipedia.org/wiki/X86#SSE
[5] Wikipedia, popcount, C naive implem. https://en.wikipedia.org/wiki/Hamming_weight#Efficient_implementation
[6] Código SSSE3 para fast popcount http://0x80.pl/articles/sse-popcount.html
Copia rescatada de http://web.archive.org/web/20100701222327/http://wm.ite.pl/articles/sse-popcount.html
[7] GAS manual http://sourceware.org/binutils/docs/as/index.html
9.13: 80386 depend.features http://sourceware.org/binutils/docs/as/i386_002dDependent.html
[8] GCC manual v.7.3 (la del laboratorio) https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc/
6: Extensions to C Language https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc/#toc-Extensions-to-the-C-Language-Family
6.32.1: Common Variable attributes https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc/Common-Variable-Attributes.html
6.45 How to Use Inline Assembly https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc/Using-Assembly-Language-with-C.html
6.45.2 Extended Asm https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc/Extended-Asm.html
6.45.3 Constraints https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc/Constraints.html
[9] GCC Inline Assembly HOWTO http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html
6: More about constraints http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html#s6
[10] Linux Assembly HOWTO http://tldp.org/HOWTO/Assembly-HOWTO/index.html
3.1: GCC inline assembly http://tldp.org/HOWTO/Assembly-HOWTO/gcc.html
Brennan's Guide to inline asm http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html
5.1: Linux calling conventions http://tldp.org/HOWTO/Assembly-HOWTO/linux.html
[11] Sourceforge tutorials http://asm.sourceforge.net/resources.html#tutorials
Using asm in Linux http://asm.sourceforge.net/articles/linasm.html#InlineASM
Inline asm x86 – IBM http://www.ibm.com/developerworks/linux/library/l-ia
Otra Brennan's – SETI@Home http://setiathome.ssl.berkeley.edu/~korpela/djgpp_asm.html
Miyagi's intro – texto http://asm.sourceforge.net/articles/rmiyagi-inline-asm.txt

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

Apéndice 1. Ejemplo de gráfica

A continuación se muestra un ejemplo del tipo de hoja de cálculo deseada, en donde se anota el modelo de CPU y el comando usado para recompilar el programa, lanzar las mediciones y preparar los números para poder hacer *copy-paste* fácilmente. En la parte dedicada a anotar las mediciones, la media se calcula sobre las columnas 1-10, ignorando la columna 0 como se explica en el texto recordatorio. La medición 0 suele salir claramente mal. Si fuera otra la que sale mal, se podría intercambiar con la 0.

lscpu:	CPU(s): 4 Nombre del modelo: Intel(R) Core(TM) i5-4460 CPU @ 3.20GHz Virtualización: VT-x Caché L3: 6144K

POPCOUNT:

```
for i in 0 g 1 2; do
   printf "__OPTIM%1c__%48s\n" $i "" | tr " " "="
   rm popcount
   gcc popcount.c -o popcount -O$i -D TEST=0
 for j in $(seq 0 10); do
   echo $j; ./popcount
 done | pr -11 -l 22 -w 80
done
```

Ignorar medición 0, repetir columna si alguna medición se sale demasiado de la media

Zona para reprodu... recordar que se ig...

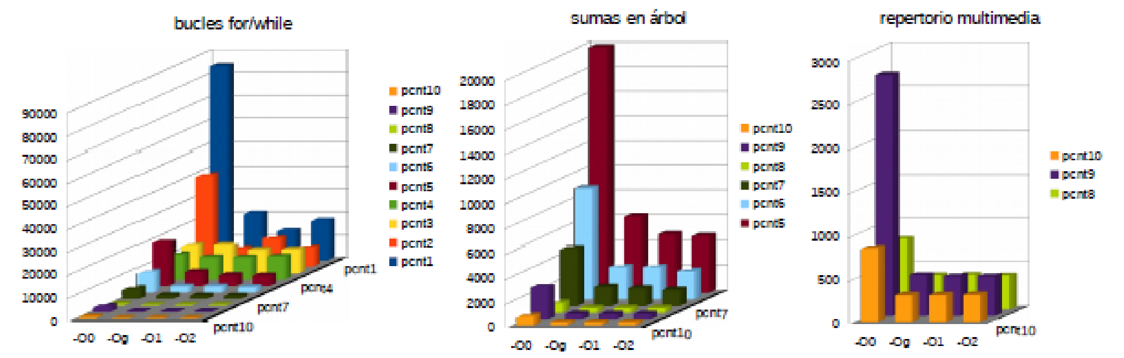
Optimizacion -O0	0	1	2	3	4	5	6	7	8	9	10	media	media	0	1
popcount1 (lenguaje C - for):	88237	74662	81400	88886	82354	88627	86607	83895	85103	88977	85817	84633	87532	88724	82991
popcount2 (lenguaje C - while):	41198	40171	39885	39797	39798	39754	39746	39849	39684	39715	39824	39822	39966	40976	39744
popcount3 (leng.ASM-body while 4i):	12618	12458	12462	12442	12459	12409	12473	12429	12422	12424	12465	12444	12477	12781	12648
popcount4 (leng.ASM-body while 3i):	11165	11039	10986	10989	10973	10984	11051	10967	10966	10984	11010	10995	10992	11050	11024
popcount5 (CS:APP2e 3.49-group 8b):	20001	19986	19993	19963	19880	19881	19941	19892	19930	19863	19920	19925	19898	19987	19966
popcount6 (Wikipedia- naive - 32b):	9050	9072	9053	9042	9027	9032	9025	9015	9112	9030	9052	9046	9044	9075	9082
popcount7 (Wikipedia- naive -128b):	4706	4710	4719	4702	4712	4676	4704	4729	4707	4695	4706	4706	4710	4703	4725
popcount8 (asm SSE3 - pshufb 128b):	830	809	809	834	810	811	835	810	810	814	832	817	810	827	811
popcount9 (asm SSE4- popcount 32b):	2733	2760	2772	2730	2733	2730	2774	2733	2753	2753	2773	2751	2746	2732	2755
popcount10(asm SSE4- popcount128b):	842	869	868	844	844	844	844	843	847	846	845	849	855	845	844

Optimizacion -Og	0	1	2	3	4	5	6	7	8	9	10	media	media	0	1
popcount1 (lenguaje C - for):	21631	21455	21008	21000	20995	21095	21022	20923	20923	20945	21030	21040	21000	21214	21257
popcount2 (lenguaje C - while):	8015	8379	7995	8064	8010	8584	8008	8002	7986	7988	7963	8097	8146	7990	8384
popcount3 (leng.ASM-body while 4i):	12472	12745	12486	12516	12494	12493	12516	12526	12459	12510	12452	12520	12525	12507	12807
popcount4 (leng.ASM-body while 3i):	9610	9781	9558	9664	9621	9605	9565	9572	9974	9621	9594	9656	9618	9639	9719
popcount5 (CS:APP2e 3.49-group 8b):	6219	6219	6232	6227	6214	6248	6223	6219	6241	6227	6228	6228	6245	6254	6397
popcount6 (Wikipedia- naive - 32b):	2641	2660	2617	2688	2640	2618	2645	2625	2617	2661	2638	2640	2638	2615	2651
popcount7 (Wikipedia- naive -128b):	1547	1549	1573	1550	1549	1572	1551	1573	1586	1553	1576	1563	1557	1582	1548
popcount8 (asm SSE3 - pshufb 128b):	395	392	391	437	391	393	417	391	392	393	392	399	396	390	394
popcount9 (asm SSE4- popcount 32b):	466	502	463	469	462	467	466	465	464	514	465	474	465	467	467
popcount10(asm SSE4- popcount128b):	313	313	312	314	312	313	313	313	313	315	313	313	313	313	315

Optimizacion -O1	0	1	2	3	4	5	6	7	8	9	10	media	media	0	1
popcount1 (lenguaje C - for):	13697	14204	13640	13634	13667	13719	13694	13719	13675	13618	13642	13721	13671	14236	13651
popcount2 (lenguaje C - while):	12531	13102	12565	12523	12545	12540	12531	12544	12623	12553	12560	12607	12552	12763	12549
popcount3 (leng.ASM-body while 4i):	10111	10370	10032	10150	10090	10074	10151	10165	10221	10103	9930	10129	10159	10344	10215
popcount4 (leng.ASM-body while 3i):	9649	9716	9631	9640	9584	9622	9642	9579	9603	9559	9648	9612	9656	9604	10077
popcount5 (CS:APP2e 3.49-group 8b):	4571	5148	5199	4663	4357	4815	4726	4630	4481	5513	4487	4802	4913	5184	4371
popcount6 (Wikipedia- naive - 32b):	2650	2658	2631	2633	2636	2638	2656	2615	2616	2633	2615	2633	2632	2614	2654
popcount7 (Wikipedia- naive -128b):	1418	1419	1438	1418	1417	1419	1419	1442	1600	1461	1443	1448	1456	1465	1637
popcount8 (asm SSE3 - pshufb 128b):	392	428	416	391	390	393	393	391	450	394	391	404	406	390	514
popcount9 (asm SSE4- popcount 32b):	452	455	453	453	451	473	473	451	461	454	462	458	458	453	462
popcount10(asm SSE4- popcount128b):	314	311	312	313	312	313	326	311	319	314	312	314	317	312	323

Optimizacion -O2	0	1	2	3	4	5	6	7	8	9	10	media	media	0	1
popcount1 (lenguaje C - for):	18567	18047	17847	18173	17391	18290	17996	18097	18314	17820	18076	18005	17748	18399	18186
popcount2 (lenguaje C - while):	8245	8545	8439	8253	8589	8577	8576	8601	8531	8564	8368	8504	8464	8044	8517
popcount3 (leng.ASM-body while 4i):	10268	10395	10252	10252	10288	10263	10264	10316	10238	10315	10215	10280	10320	10223	10761
popcount4 (leng.ASM-body while 3i):	9543	9835	9554	9587	9544	9651	9551	11301	9572	11709	9631	9994	9838	9581	9816
popcount5 (CS:APP2e 3.49-group 8b):	4622	4937	4443	4544	4486	4475	4692	4760	4617	4875	4707	4654	4790	4711	4850
popcount6 (Wikipedia- naive - 32b):	2340	2345	2327	2305	2325	2303	2303	2346	2302	2327	2348	2323	2319	2327	2304
popcount7 (Wikipedia- naive -128b):	1330	1320	1315	1360	1317	1338	1360	1316	1315	1314	1315	1327	1335	1316	1318
popcount8 (asm SSE3 - pshufb 128b):	392	395	392	393	391	389	391	390	414	390	391	394	395	393	431
popcount9 (asm SSE4- popcount 32b):	462	461	475	455	454	454	453	455	455	455	456	457	461	457	458
popcount10(asm SSE4- popcount128b):	313	317	311	312	312	312	312	311	330	350	312	318	312	336	313

POPCOUNT:	-O0	-Og	-O1	-O2	Ganancias	-O0	-Og	-O1	-O2	Comentario
pcnt1	84633	21040	13721	18005	pcnt1			1.00		comparado con el for más rápido
pcnt2	39822	8097	12607	8504	pcnt2		1.69			el while es un 70% más rápido
pcnt3	12444	12520	10129	10280	pcnt3			1.35		ASM se queda en un 35%
pcnt4	10995	9656	9612	9994	pcnt4			1.43		o en un 43%
pcnt5	19925	6228	4802	4654	pcnt5				2.95	sumar en grupos 8b sale 3x más rápido
pcnt6	9046	2640	2633	2323	pcnt6				5.91	sumar en árbol 6x
pcnt7	4706	1563	1448	1327	pcnt7				10.34	lectura 128b sube a 10x
pcnt8	817	399	404	394	pcnt8				34.86	SSSE3 sube a 35x más rápido
pcnt9	2751	474	458	457	pcnt9				30.00	SSE4 sólo 30x por leer 32b
pcnt10	849	313	314	318	pcnt10		43.82	43.66	43.16	SSE4 128b sube a 44x



Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/

Notar que las tablas grises son otra medición completa, por si acaso varios números de una fila estuvieran claramente mal y necesitáramos sustituirlos por otros correctos. Si sólo uno estuviera mal, se podría intercambiar con la columna 0. La media se hace de las mediciones 1-10, excluyendo la medición 0, que estadísticamente suele salir peor que el resto.

Más habitualmente, repetiremos dos veces el comando, anotando unos resultados a la izquierda y otros a la derecha (tablas grises). Las gráficas se calculan a partir de la información a la izquierda, y se ha procurado que las medias de ambas partes queden enfrentadas para poderlas comparar sin esfuerzo. Si salen medias muy parecidas, ambas mediciones está bien hechas y ni siquiera nos molestaremos en comprobar si alguna medición está subiendo demasiado la media y querríamos sustituirla por la medición 0, o por alguna de la parte derecha.

Notar que ha sido necesario copiar y etiquetar las medias (para que queden adyacentes, ver tabla roja) de forma que sea fácil generar la gráfica a partir de la tabla roja. El asistente de Calc para generar gráficas necesita que los datos estén de partida dispuestos bidimensionalmente, y puede aprovechar las cabeceras para etiquetar los ejes de la gráfica y la leyenda adecuadamente.

Para que los gráficos de columnas 3D resulten intuitivos, se ha procurado que cambie el color de columna con la versión, y se mantenga para los distintos niveles de optimización. También se ha procurado que al fondo aparezcan las mediciones más lentas, para que no tapen a las más rápidas. Otro detalle para facilitar la comprensión consiste en añadir un texto que describa aproximadamente lo que se hacía en cada versión, de manera que se pueda recordar de qué versión estamos hablando. El texto descriptivo no se incluye a la hora de etiquetar los ejes.

Se añaden otras gráficas de detalle (*zoom*), copias de la primera, eliminando las versiones anteriores para apreciar mejor las diferencias entre distintas versiones posteriores. La mejora combinada es tan grande que las últimas versiones parecen tardar todas 0 cuando se visualizan a la escala de la primera versión.

Notar que se resalta el modelo de CPU usado, y si se estaba usando virtualización. En un equipo de sobremesa como los disponibles en el L-2.9 (como el usado para esta gráfica) las mediciones saldrán seguramente bien a la primera. Se tarda más en hacer *copy-paste* que en realizar la medición propiamente dicha. En un equipo portátil seguramente se tendrán activadas en la BIOS diversas opciones de protección termal y ahorro de energía que justamente reducen la velocidad del reloj cuando el procesador empieza a trabajar fuerte y calentarse, impidiendo obtener mediciones fiables. En un portátil con Windows ejecutando Linux en una máquina virtual, las mediciones son adicionalmente impedidas. Aun así, es posible que cualitativamente se obtengan resultados sensatos.

En el laboratorio L-2.9, como muestra la gráfica, se han obtenido ganancias (comparando con el mejor `for`) de alrededor de 1.7 para el bucle `while`, alrededor de 3, 6 y 10 para algoritmos de suma en árbol (no se obtuvo ganancia intentando hacer el `while` mejor que `gcc`), y alrededor de 35, 30 y 44 para el repertorio SSSE3 y SSE4.

Recordar siempre que no se debe medir la velocidad de un programa sin haber comprobado previamente que produce el resultado correcto. Si se siguió la recomendación de autodocumentar el método de test y cronometraje (Figura 19), la comprobación se puede realizar en segundos.

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND