4 Trabajo a realizar

Nos interesaría experimentar con ejemplos que permitan obtener ventaja sobre gcc, y que al mismo tiempo sean lo suficientemente sencillos como para estudiarlos y programarlos en pocas sesiones de prácticas. O aún mejor, que no requieran estudio adicional.

Estas condiciones las cumplen por ejemplo: el cálculo del peso Hamming o *"population count"*, ya visto en clase de teoría, para el cual existe una instrucción SSE4.2 cuyo uso gcc no podrá deducir a partir de nuestro código C; y el cálculo de la paridad, que también se postula en el libro de teoría (p.300) como buen candidato para ello, siendo esta vez el bit PF la característica que gcc no aprovecha; en ambos casos daremos pistas sobre las instrucciones a utilizar, la idea general que debe implementar del tramo de código ASM, y las restricciones a utilizar, al objeto de guiar, orientar y acelerar tanto la lectura del manual del repertorio de instrucciones como la programación de los tramos `asm()`.

Se trata por tanto de programar varias versiones de estas dos funciones:

- Sumar los pesos Hamming (nº de bits activados) de todos los elementos de un array
- Sumar las paridades de todos los elementos de un array

…con y sin ensamblador en-línea, y comprobando siempre la corrección del resultado calculado. Para ello, podemos consensuar algunos ejemplos pequeños de prueba, cuyo resultado correcto pueda calcularse a mano. Pero para que el tiempo de medición sea apreciable tendremos que usar arrays de mayor tamaño, y para conocer el resultado correcto hará falta una fórmula aplicable a los datos de entrada utilizados.

Se deben cronometrar de forma justa y equitativa todas las versiones. Una vez desarrollado el programa y comprobados los ejemplos de tamaño pequeño, repetiremos 10 veces la ejecución 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 de paquete ofimático (Calc o Excel).

Se propondrá comenzar con un programa normal en C (la versión más inmediata posible), y continuar con mejoras que no requieran ASM, si las hubiera. Cuando no se pueda mejorar más en lenguaje C, pasar a ASM en-línea. A veces también propondremos versión ASM de versiones C no óptimas, como ejercicio preparatorio, especialmente si la versión óptima ASM se basa en SSE4.

En el laboratorio disponemos de procesadores con SSE3 (no SSE4), mientras que muchos estudiantes disponen de portátiles con SSE4. Se pedirá por tanto realizar alguna versión ASM que se pueda probar en el laboratorio, y se dejará sugerida alguna otra que aproveche las capacidades superiores de los portátiles, para los entusiastas que siempre quieren probar lo más avanzado.

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, responder a las preguntas de autocomprobación, 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 ddd, 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: marcos de pila (Tema 2.1, transparencias 31-39), códigos de condición (Tema 2.2, tr. 18-25), bucles (tr. 37-49), estructura de la pila (Tema 2.3, tr. 1-32), convenciones de llamada (tr. 33-36), punteros y variables locales (tr. 45-48), declaración y acceso a arrays (Tema 2.4, tr. 15-37). Se recomienda su estudio detallado.

4.0 Contestar las preguntas de autocomprobación (suma_01-suma_09)

El objetivo es comprender con detalle el proceso de ensamblado, compilación y enlace de los programas mixtos C-ASM, utilizar con soltura las herramientas implicadas (incluyendo `objdump` y `nm`), entender cuándo y por qué hace falta enlazar la librería C, el *runtime* C y el enlazador dinámico, entender cuándo conviene usar `as`/`ld` y cuándo `gcc`, comprender la convención `cdecl`, ser capaz de leer (comprender) y redactar código ASM en convención `cdecl` y en-línea (*inline* ASM), y adquirir habilidad en el manejo de las herramientas usadas (compilador, ensamblador, enlazador y depurador).

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

Utilizar el programa `suma_09` de la Figura 12 como esqueleto para cronometrar diversas versiones de una función que sume los bits (peso Hamming, *popcount*) de los elementos de una lista de N números. Notar que la suma puede llegar a ser 32*N (en modo 32bits, donde un entero ocupa 4B), 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, 0x00100000, 0x00000800, 0x00000001};`
- `unsigned lista[SIZE]={0x7fffffff, 0xffefffff, 0xfffff7ff, 0xfffffffe, 0x01000024, 0x00356700, 0x8900ac00, 0x00bd00ef};`
- `unsigned lista[SIZE]={0x0 , 0x10204080, 0x3590ac06, 0x70b0d0e0, 0xffffffff, 0x12345678, 0x9abcdef0, 0xcafebeef};`

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.

Para poder comparar los tiempos de ejecución de distintos programas (de distintos usuarios) en el laboratorio (con los mismos ordenadores), 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 bits intermedios?

Realizar una **primera** y **segunda** versiones C como las vistas en clase (Tema 2.2, tr.43 y 38), recorriendo el array con un bucle `for`, y recorriendo los bits con bucle `for` (1ª versión, p.43) o con un bucle `while` (2ª versión, p.38), aplicando en ambos máscara 0x1 y desplazamiento a la derecha (p.38), 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.

Notar que la variable `result` puede continuar usándose para seguir sumando los bits de otro elemento. Notar que preferimos llamar "1ª versión" a la del bucle `for` (tr.43), 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** versión traduciendo el bucle interno `while` por un tramo de unas 4-5 líneas ensamblador 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, debería suponer alguna mejora sobre la 2ª versión. El resultado podría sorprendernos. 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/s:     añadir a lo acumulado por el momento
       : [x] "r" (x)      );           // entrada: valor elemento
    }
```

Implementar como **cuarta** versión la solución que aparece en el libro de clase [1], problema 3.49, resuelto en la página 364 (lenguaje C). Viene resuelto para 64bits (y un único elemento), bastaría con adaptarlo a 32bits (y un array completo). 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. Esta cuarta versión nos demostraría, caso de ser mejor, lo difícil que es ganar a un programa C bien pensado.

Para una **quinta** versión, podemos buscar con Google qué otros métodos han usado algunos entusiastas para calcular el *popcount*, e implementar alguno de ellos (ver [5], 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 13 y los párrafos tras ella se dedican a explicar el método [5].

Una **sexta** versión consistiría en sustituir todo el bucle interno `while` por la instrucción SSE4 popcount. Compararíamos con el crono de la versión 5 para ver cuánto se gana por pasar del repertorio SSSE3 a SSE4 (y a lo mejor nos volveríamos a sorprender). Atendiendo a que esta versión no se podría ejecutar en el laboratorio, o en un portátil que no tenga SSE4, se deja tan sólo como sugerencia.

Para los entusiastas que siempre desean algo más, sugerimos una **séptima** y última versión para mejorar prestaciones, consistente en realizar dos lecturas y dos *popcount*. En modo de 32bits no podemos usar el *popcount* de 64bits, siendo ésto lo más parecido que se puede conseguir. Ambos ejemplos se ofrecen resueltos, aunque no se puedan ejecutar en el laboratorio.

```
// Versión SSE4.2 (popcount)
int popcount6(unsigned* array, int len)
{
    int i;
    unsigned x;
    int val, result=0;

    for (i=0; i<len; i++)
    {
      x = array[i];
      asm("popcnt %[x], %[val]"
       : [val] "=r" (val)
       :   [x]  "r" (x)
      );
      result += val;
    }
    return result;
}
```

```
// popcount 64bit p/mejorar prestaciones
int popcount7(unsigned* array, int len){
    int i;
    unsigned x1,x2;
    int val,result=0;
    if (len & 0x1)
      printf("leer 64b y len impar?\n");
    for (i=0; i<len; i+=2)    {
      x1 = array[i]; x2 = array[i+1];
      asm("popcnt %[x1], %[val] \n\t"
          "popcnt %[x2], %%edi  \n\t"
          "add    %%edi, %[val] \n\t"
        : [val]"=&r" (val)
        :  [x1]  "r" (x1),
           [x2]  "r" (x2)
        : "edi");
      result += val;    }
    return result;    }
```

```
// Versión SSSE3 (pshufb) web http:/wm.ite.pl/articles/sse-popcount.html
int popcount5(unsigned* array, int len)
{
    int 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"  // dos copias de x
        "movdqu    %[m], %%     \n\t"  // máscara
        "psrlw      $4 , %%     \n\t"
        "pand    %%xmm6, %%xmm0 \n\t"  //; xmm0 - nibbles inferiores
        "pand    %%xmm6, %%xmm1 \n\t"  //; xmm1 - nibbles superiores

        "movdqu    %[l], %%     \n\t"  //; ...como pshufb sobrescribe LUT
        "movdqa  %%xmm2, %%     \n\t"  //; ...queremos 2 copias
        "pshufb  %%xmm0, %%xmm2 \n\t"  //; xmm2 = vector popcount inferiores
        "pshufb  %%xmm1, %%xmm3 \n\t"  //; xmm3 = vector popcount superiores

        "paddb   %%    , %%xmm3 \n\t"  //; xmm3 - vector popcount 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 = [    no usado  |pcnt bytes0..15]
        "movd    %%xmm0, %[val] \n\t"
```

```
        : [val]"=r" (val)
        :  [x]  "m" (array[i]),
           [m]  "m" (SSE_mask[0]),
           [l]  "m" (SSE_LUTb[0])
      );
         result +=  val;
    }
    return result;
}
```

Figura 13: popcount5: función para cálculo SSSE3 del peso Hamming (algunos registros XMM omitidos)

Para comprender el método SSSE3 propuesto en la web [5] 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 13.

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 [5]. 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.

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

Como también nos interesa saber cómo mejora gcc según el nivel de optimización (`-O0`, `-O1`, `-O2`), repetiremos 10 mediciones de tiempo para esos 3 niveles. Se trata por tanto de recompilar 3 veces, repetir 10 mediciones, y organizar los resultados en una gráfica de paquete ofimático (Calc o Excel), mostrando los promedios de cada versión (1ª - 5ª) para cada nivel de optimización (O - 2), tal vez con un gráfico de barras con abscisas bidimensionales versión-optimización (Excel 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, y si no lo es, 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.

Se puede usar compilación condicional para facilitar tanto la comprobación de los ejemplos pequeños que hemos sugerido, como la realización de las mediciones y su incorporación a una hoja Calc. Considerar el siguiente esquema: sólo si no se activa TEST se define e inicializa normalmente el array; si se activa, se define un array más corto y no se inicializa en el programa principal.

```
#define TEST          0
#define COPY_PASTE_CALC      1

#if ! TEST
  #define NBITS 20
  #define SIZE (1<<NBITS)     // tamaño suficiente para tiempo apreciable
  unsigned lista[SIZE];
  #define RESULT (...)        // fórmula
#else
/* ------------------------------------------------------------------- */
  #define SIZE 4
  unsigned lista[SIZE]={0x80000000, 0x00100000, 0x00000800, 0x00000001};
                    // 1 ^       + 1   ^     + 1      ^  + 1        ^ = 4
  #define RESULT 4
/* ------------------------------------------------------------------- */
#endif
...
int main()
{
#if ! TEST
    int i;                    // inicializar array
    for (i=0; i<SIZE; i++)
       lista[i]=i;
#endif

    crono(popcount1, "popcount1 (lenguaje C -       for)");
    ...
#if ! COPY_PASTE_CALC
    printf("calculado = %d\n", RESULT);
#endif
    exit(0);
}
```

Figura 14: uso de compilación condicional para facilitar la comprobación de ejemplos pequeños

De la misma forma, en la función `crono()` de la Figura 12 se puede programar un `printf()` alternativo para cuando estemos haciendo mediciones de tiempo con intención de incorporarlas a una hoja Calc, uno que sólo imprima el tiempo, sin mensajes adicionales, que funcionaría cuando no se active el símbolo COPY_PASTE_CALC. Podríamos entonces lanzar la ejecución de las 10 mediciones con un simple comando Shell:

```
for (( i=0 ; i<11; i++ )); do echo $i ; ./popcount; done | pr -11 -l 20 -w 80
```

En realidad ese comando lanza 11 ejecuciones, por si la primera (o alguna) sale claramente peor que el resto, y pagina los 55 números (5 versiones x 11 mediciones) en 11 columnas para poder hacer *copy-paste* fácilmente a la hoja Calc. Los comandos para recompilar y lanzar la medición se podrían anotar en la propia hoja Calc como recordatorio para cuando se desee repetir el experimento.

En el Apéndice 2 se recuerdan algunas preguntas de autocomprobación.

4.2 Calcular la suma de paridades de una lista de enteros sin signo

Utilizar el programa `suma_09` de la Figura 12 como esqueleto para cronometrar diversas versiones de una función que sume las paridades de los elementos de una lista de N números, calculadas como el XOR (lateral) de los bits de cada elemento. Notar que la suma puede llegar a ser N, si todos tienen paridad impar (y producen XOR lateral 1). Concluir que basta calcular la suma en un entero, para cualquier valor práctico de N.

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

- `unsigned lista[SIZE]={0x80000000, 0x00100000, 0x00000800, 0x00000001};`
- `unsigned lista[SIZE]={0x7fffffff, 0xfeffffff, 0xfffff7ff, 0xfffffffe,`
  `0x01000024, 0x00356700, 0x8900ac00, 0x00bd00ef};`
- `unsigned lista[SIZE]={0x0 , 0x10204080, 0x3590ac06, 0x70b0d0e0,`
  `0xffffffff, 0x12345678, 0x9abcdef0, 0xcafebeef};`

Los resultados deberían ser 4, 8, 2, respectivamente. Notar que la lista se declara sin signo para que los desplazamientos (que previsiblemente tendremos que utilizar) no dupliquen ningún bit de signo.

Para poder comparar los tiempos de ejecución de distintos programas (de distintos usuarios) en el laboratorio (con los mismos ordenadores), necesitaremos acordar algún ejemplo de tamaño mayor, como por ejemplo SIZE=2^{20} elementos, inicializados en secuencia desde 0 hasta SIZE-1. Calcular qué suma de paridades 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 a los dos primeros elementos? ¿Y a los siguientes 2? ¿Y a los siguientes 4?

Realizar una **primera** y **segunda** versiones C como las vistas en clase para *popcount*, recorriendo el array con un bucle `for`, y recorriendo los bits con bucle `for` (1ª versión) o con un bucle `while` (2ª versión), aplicando en ambos máscara 0x1 y desplazamiento a la derecha para ir extrayendo y acumulando los bits. Vamos a necesitar otra variable auxiliar (como `val` en la versión `popcount4`) para acumular lateralmente los bits con XOR (^=) en lugar de con suma normal (+=), y esa suma acumularla normalmente (+=) con `result`. Compararíamos los tiempos para comprobar que a veces se pueden obtener buenas ganancias pensando bien las cosas en C, sin necesidad de usar ASM.

Implementar como **tercera** versión la solución que aparece en el libro de clase [1], problema 3.22, resuelto en la página 352 (lenguaje C), adaptándola para array completo, en lugar de un solo elemento. La segunda versión no estaba tan bien pensada como nos imaginábamos: la máscara se puede aplicar al acumular con `result`, ahorrándose todas las máscaras del bucle `while`. Cuando realicemos las mediciones de tiempo comprobaremos si esta mejora es más o menos importante con los distintos niveles de optimización.

Realizar una **cuarta** versión traduciendo el bucle interno `while` por un tramo de unas 4-5 líneas ensamblador que incluyan la instrucción XOR (y SHR, que ya utilizamos en el ejemplo anterior). Notar que SHR afecta al flag ZF, pudiéndose hacer una traducción casi literal del código C de la 3ª versión, incluyendo la máscara final con 0x1. En principio, debería suponer alguna mejora sobre la 3ª versión.

Por facilitar también el desarrollo de este ejemplo, indicamos unas posibles restricciones:

```
    for (i=0; i<len; i++) {
      x = array[i];
      val = 0;
      asm("\n"
"ini3:                \n\t"          // seguir mientras que x!=0
       "xor %[x], ... \n\t"          // realmente sólo nos interesa LSB
       ...
       : [v]"+r" (val)               // e/s:     entrada 0, salida paridad elemento
       : [x] "r" (x)                 // entrada: valor elemento
      );
      result += val;
    }
```

Para una **quinta** versión, podemos recuperar la idea de sumar en árbol usada en `popcount4` (Libro de clase, Problema 3.49, p.364). La idea es someter al elemento del array a XOR y desplazamientos sucesivos cada vez a mitad de distancia (16, 8, 4, 2, 1) hasta que finalmente se hace sencillamente `x^=x>>1`. En lugar de hacerlo "a mano" (muy poco elegante, 5 líneas de código idénticas cambiando sólo la distancia de desplazamiento), se puede programar un bucle `for (j=…)` en el cual la distancia vaya cambiando, y el cuerpo del bucle sería `x^=x>>j`. De nuevo, se puede aplicar la máscara al valor final antes de acumular con `result`. Razonar por qué funciona (en qué propiedades de XOR se basa) este método.

Realizar una **sexta** versión traduciendo el bucle interno `for` por un tramo de unas 6 líneas ensamblador que incluyan las instrucciones XOR y SHR ya conocidas, y las instrucciones SETcc y MOVZx (también estudiadas en clase). La ventaja a explotar en este caso es que XOR afecta al flag PF señalando la paridad par de los 8 bits inferiores (ver libro de clase, p.300, o manual de Intel, vol.1, sección 3.4.3.1, p.3-21), pudiendo obtener el resultado a partir de PF (mediante SETcc, consultar el manual de Intel para escoger el mnemotécnico requerido) tras llegar a 8bits. En este caso no consideramos poco elegante realizar "a mano" los desplazamientos requeridos (porque sólo se necesitará 16), así que la traducción C→ASM será un poco "libre". Compararíamos con el crono de la versión 5ª para ver la ganancia por usar ASM, cuando existen instrucciones o características que gcc no sabe aprovechar.

En principio recomendaríamos las siguientes restricciones para poder operar con facilidad en 16 y 8 bits

```
      asm(
       "mov   %[x], %%edx    \n\t "  // sacar copia para XOR. Controlar el registro...
       ...                           // (EDX) nos permite usar nombres registros 8bits
       "movzx %%dl,  %[x]    \n\t"   // devolver en 32bits
       : [x]"+r" (x)                 // e/s: entrada valor elemento, salida paridad
       :
       : "edx"                       // clobber
      );
```

Mediciones y recomendaciones

Se hacen los mismos comentarios que para el problema anterior: repetir 10 mediciones para los 3 niveles de optimización, y presentar los resultados en *"columnas 3D"* de Calc o Excel, mostrando los promedios de cada versión (1ª -6ª) para cada nivel de optimización (0 - 2). 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, intentar explicar dicho comportamiento.

Recordar que siempre se debe comprobar que el resultado es correcto. Como mínimo se debe comprobar que todas las versiones dan el resultado correcto para los ejemplos pequeños y el ejemplo grande propuestos.

Recordar que se puede usar compilación condicional para facilitar tanto la comprobación de los ejemplos como la realización de las mediciones y su incorporación a una hoja Calc. Seguramente convendría anotar en la propia hoja Calc el comando Shell usado para lanzar la ejecución de las 10 mediciones. También convendría anotar el modelo de CPU donde se hizo la medición.

En el Apéndice 2 se recuerdan algunas preguntas de autocomprobación.

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 se deban entregar los listados de los programas realizados (`parity.c`, `popcount.c`), y las gráficas de las mediciones de tiempo, y un documento PDF con las respuestas a las preguntas de autocomprobación, subiéndolo al SWAD hasta 3 días después de la última sesión de prácticas, con penalización creciente por entrega tardía hasta 1 semana posterior.

Puede que en otro grupo se pueda trabajar y entregar por parejas, pero que el profesor de prácticas visite cada puesto al final de cada sesión comprobando si ambos estudiantes saben responder a las preguntas de autocomprobación, programar en ensamblador en-línea y utilizar las herramientas, permitiendo que se suba al SWAD el trabajo en caso afirmativo.

Los profesores de teoría y prácticas de cada grupo acordarán cómo entregará ese grupo el trabajo desarrollado.

6 Bibliografía

[1] Apuntes y presentaciones de clase, y particularmente
Programación Máquina II: Aritmética y Control
sección "Bucles", p.38 y siguientes
sección "Códigos de condición", instrucciones `test/setcc`, p.22-25
Libro CS:APP, Problema 3.49, p.364.
Randal E. Bryant, David R. O'Hallaron: "Computer Systems: A Programmer's Perspective", 2nd Ed., Pearson, 2011. http://csapp.cs.cmu.edu/

[2] Manuales de Intel sobre IA-32 e Intel64, en concreto el volumen 2: "Instruction Set Reference"
http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-software-developer-vol-2a-2b-instruction-set-a-z-manual.pdf

[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] 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

[6] GAS manual http://sourceware.org/binutils/docs/as/index.html
9.13: 80386 depend.features http://sourceware.org/binutils/docs/as/i386_002dDependent.html

[7] GCC manual v.4.4 (la del laboratorio) http://gcc.gnu.org/onlinedocs/gcc-4.4.6/gcc/
5: Extensions to C Language http://gcc.gnu.org/onlinedocs/gcc-4.4.6/gcc/index.html#toc_C-Extensions
5.33: Variable attributes http://gcc.gnu.org/onlinedocs/gcc-4.4.6/gcc/Variable-Attributes.html
5.37: Assembler with C operands http://gcc.gnu.org/onlinedocs/gcc-4.4.6/gcc/Extended-Asm.html
5.38: Constraints http://gcc.gnu.org/onlinedocs/gcc-4.4.6/gcc/Constraints.html#Constraints

[8] 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

[9] 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

[10] 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

Apéndice 1. Ejemplo de gráficas

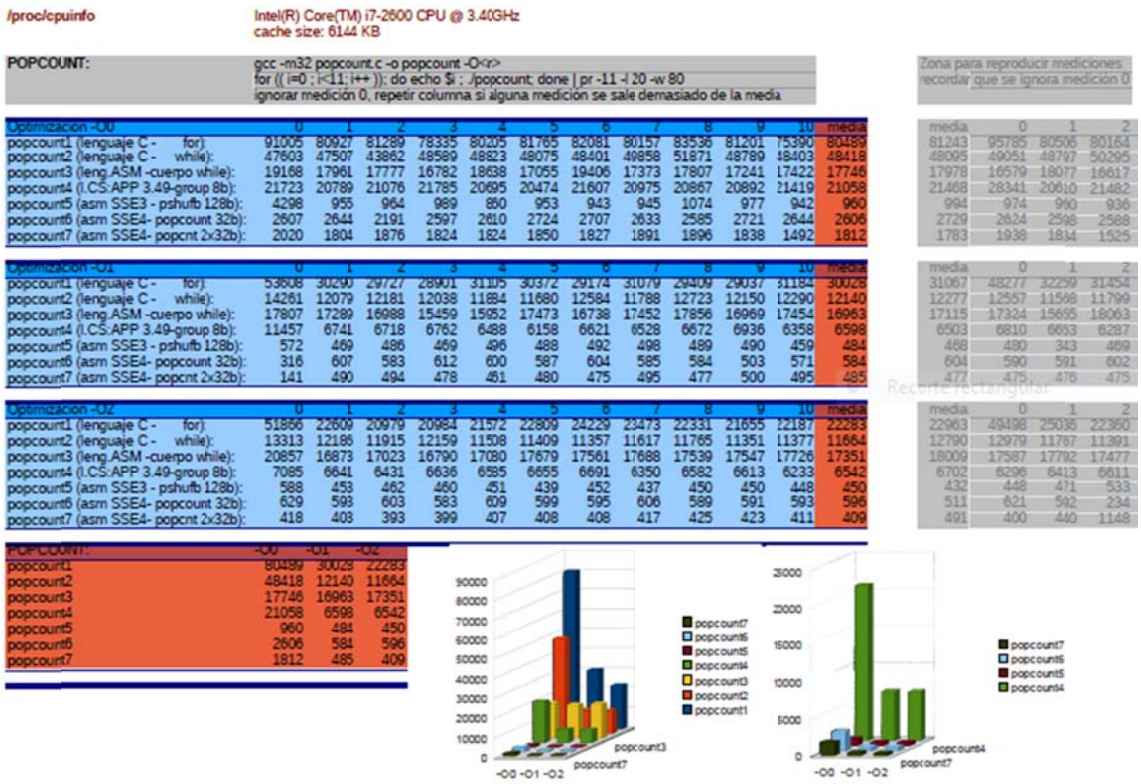
A continuación se muestran ejemplos del tipo de hoja de cálculo deseada, en donde se anotan las mediciones (incluso repetidas, para poder corregir mediciones erróneas), el comando usado para compilar el programa, el comando usado para lanzar la medición, el modelo de CPU usada, e incluso un texto recordatorio de que se están realizando 11 mediciones por si la primera suele salir mal.

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.

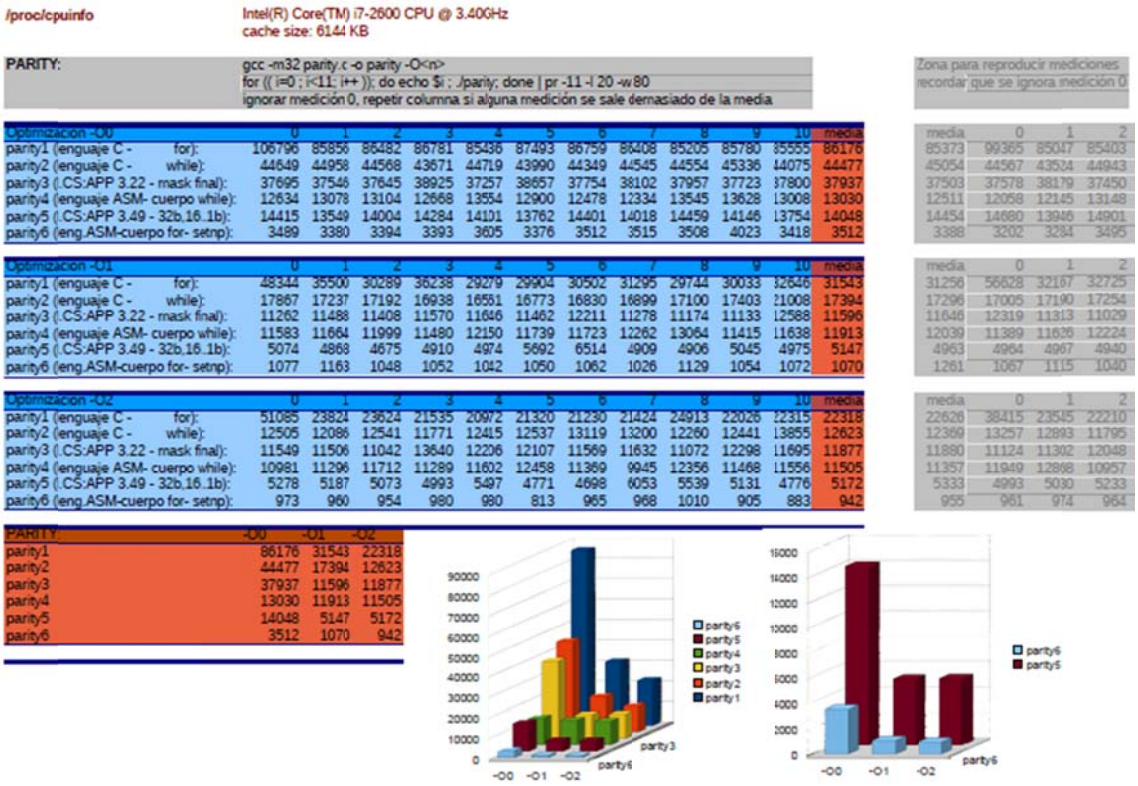
Notar que las tablas grises son otra medición, por si acaso algún número estuviera claramente mal y necesitáramos sustituirlo por otra medición correcta. La media se hace de las mediciones 1-10, excluyendo la medición 0, que estadísticamente suele salir peor que el resto. Notar también que ha sido necesario 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.

Se añade otra gráfica, copia de la primera, eliminando todas las versiones salvo la última de lenguaje C y las últimas ASM (que mejoran el tiempo), para apreciar mejor el factor de mejora que supone utilizar ASM.

Notar por último que se resalta el modelo de CPU usado. En esta CPU, a `popcount3` no le afecta el nivel de optimización ni supone una mejora sobre la versión C. Las versiones SSE3 y SSE4 son muy superiores. Por cuantificar numéricamente la mejora que supone cada nueva versión, podemos centrarnos en el nivel de optimización –O2 y concluir que pasar de `for` a `while` supuso un factor de mejora 1.91x (=22283/11664), pasar a reducir en árbol un factor 1.78x (=11664/6542), y pasar a los repertorios SSSE3 y SSE4 un 14.5x - 16x respectivamente (=6542/450 - 409). Los inversos de dichos cocientes son la fracción de tiempo que tarda en ejecutarse el programa con dicha mejora, que son 52.3%, 56.1%, 6.88% y 6.25%, respectivamente.



Similarmente sucede en esta CPU con `parity4`, cuyas prestaciones vuelven a ser independientes del nivel de optimización ni tampoco mejora significativamente la versión C equivalente (aunque tampoco la empeora). Los tres factores de mejora importantes han sido pasar de `for` a `while` (22318/12623 = 1.77x, reduciendo el tiempo de ejecución al 56.6%), pasar a reducir en árbol (12623/5172 = 2.44x, 41%), y por último usar el bit PF (5172/942 = 5.49x, 18.2%), quedando así cuantificada numéricamente la importancia relativa de cada mejora.



Apéndice 2. Preguntas de Autocomprobación

Para evitar inquietudes sobre si es están comprendiendo bien los tutoriales, se proporcionan a continuación una serie de preguntas sobre los mismos, que pueden considerarse como ejercicios de autocomprobación.

Las siguientes preguntas se refieren al programa suma_01_S_cdecl de la Figura 2. Su propósito es que cada uno pueda comprobar por sí mismo la correcta comprensión de los ejemplos del tutorial.

	Sesión de depuración suma_01_S_cdecl
1	Se puede realizar un volcado de la pila, usando Data->Memory->Examine 8 hex words(4B) $esp, en donde se ha escogido 8 por tener margen de sobra (en línea de comandos gdb sería **x/8xw $esp**). Comprobar que coincide el volcado así obtenido con la Figura 3. *El programa principal no tiene marco de pila (EBP=0), pero el S.O. le deja algo anotado en pila. Si el 1 indicara int argc=1 y el segundo argumento fuera un array de char argv[]… ¿cómo se volcaría su valor? (Examine 1 **<qué>** bytes **<argv[0]>**) ¿Qué podría ser ese argumento?*[1] Pista: Si se desea, se puede indagar más ajustando los argumentos en **ddd** con **set args <arg1> <arg2>**
2	El volcado de pila es útil para ir viendo la pila durante la ejecución del programa, conforme va cambiando ESP. Tras llamar a **suma**, se puede realizar un volcado de memoria para comprobar que el argumento #2 es nuestra lista de 9 enteros, usando Data->Memory->Examine **<cuántos>** hex words(4B) **<qué>**. ¿De dónde sacamos **<cuántos>** y **<qué>**?

[1] *Preguntar a los estudiantes si han visto int main(int argc, char*argv[]). Si no, eliminar esa pregunta.*

3	¿Por qué la función **suma** preserva ahora %ebx y no hace lo mismo con %edx?
4	¿Qué modos de direccionamiento usa la instrucción **add (%ebx,%edx,4), %eax**? ¿Cómo se llama cada componente del primer modo? El último componente se denomina escala. ¿Qué sucedería si lo eliminásemos?
5	Es posible eliminar el factor de escala y conseguir que el programa siga funcionando correctamente sin añadir instrucciones adicionales, sino simplemente modificando las que hay. ¿Cómo? (pista: dec %ecx)
6	También es posible conseguir lo mismo dejando únicamente un puntero, **add (%edx), %eax**. ¿Cómo?
7	La instrucción **jne** en el programa original se podría cambiar por alguno de entre otros tres saltos condicionales (uno de ellos es sencillamente otro mnemotécnico para el mismo código de operación) y el programa seguiría funcionando igual. ¿Cuáles son esos 3 mnemotécnicos? ¿Qué tendría que suceder para que se notaran diferencias con el original?
8	Según la Figura 3, si hubiésemos necesitado añadir una variable local **.int** (entero de 4B) a la función suma, hubiéramos restado 4 a ESP… ¿cuándo? (entre cuáles dos instrucciones). A la salida, deberíamos sumarle 4 a ESP… ¿cuándo?
9	Si hubiésemos reservado sitio para 3 variables locales **.int** (enteros de 4B), ¿qué dos instrucciones cambiarían respecto a la pregunta anterior, y en qué cambiarían? ¿Cómo se direccionaría la segunda variable local respecto al marco de pila? Por ejemplo, ¿cómo sería la instrucción ensamblador para poner esa variable a 0?
10	Volviendo a la Figura 3, cuando una función no tiene registros salvados, sino sólo variables locales, es posible eliminarlas de otra forma alternativa, más directa que sumar el tamaño a ESP. ¿Cuál? Pista: las siguientes instrucciones serían recuperar el antiguo EBP y retornar, así que… ¿qué otra cosa se podría hacer para que POP EBP funcionara bien?

Tabla 3: preguntas de autocomprobación (suma_01_S_cdecl)

Las siguientes preguntas se refieren al programa suma_02_S_libC de la Figura 4. Aunque sea posible responder acertadamente algunas de ellas sin necesidad de realizar la sesión de depuración (ejecutando paso a paso usando ddd), se debe recordar que el objetivo de las preguntas de autocomprobación es que cada uno pueda comprobar por sí mismo su correcta comprensión de los ejemplos del tutorial.

	Sesión de depuración suma_02_S_libC
1	¿Qué error se obtiene si no se añade **–lc** al comando de enlazar? ¿Qué tipo de error es? (en tiempo de ensamblado, enlazado, ejecución…)
2	¿Qué error se obtiene si no se añade la especificación del enlazador dinámico al comando de enlazar? ¿Y si se indica como enlazador un fichero inexistente, p.ej. **<…>.so.3** en lugar de **<…>.so.2**? ¿Qué tipo de error es? (en tiempo de ensamblado, enlazado, ejecución…)
3	Proporcionar Instrucciones paso a paso para obtener un volcado como el primero de la Figura 5 (argumentos de **suma**): - en modo gráfico **ddd** (Examine <**cant**> <**fmt**> <**tam**> desde <**dir**>) y - en modo comando **gdb** (x/<**fmt**> <**addr**>)
4	En ese momento, antes de llamar a **suma**, podríamos modificar memoria con el siguiente comando gdb: **set * (int*) $esp=suma**. ¿Qué efecto tendría eso sobre nuestro programa? Para precisar la respuesta, también podemos ejecutar el comando gdb **set * (int*) ($esp+4)=2**. ¿Qué resultado se obtiene? ¿Deberían obtener todos los compañeros ese mismo resultado? ¿De qué depende que suceda eso? Dicho de otro modo… ¿qué se está sumando, al hacer esas alteraciones? (Pista: **objdump –d suma_02_S_libC** y Accesories->Calculator Hex)
5	Repetir **3** para el segundo volcado de la Figura 5 (argumentos de **printf**).
6	En ese momento, antes de llamar a **printf**, podemos modificar el puntero de pila con este comando gdb: **set $esp=$esp-4**, y modificar el tope con **set * (int*) $esp=&formato**. ¿Qué resultado se obtiene? ¿Deberían obtener todos ese mismo resultado? ¿De qué depende que suceda eso? Dicho de otro modo… ¿qué se imprime, al hacer esas alteraciones?
7	Repetir **3** para el tercer volcado de la Figura 5 (argumentos de **exit**).
8	En ese momento, justo antes de llamar a **exit**, podemos modificar el puntero de pila con el comando gdb: **set $esp=$esp+4**. ¿Qué pasa entonces? ¿En qué afecta eso a nuestro programa? ¿Deberían obtener todos ese resultado? ¿De qué depende el resultado?
9	Repetir **8** poniendo -4 en lugar de +4

Tabla 4: preguntas de autocomprobación (suma_02_S_libC)

Las siguientes preguntas se refieren al programa suma_03_SC de la Figura 6. Aunque sea posible responder acertadamente algunas de ellas sin necesidad de realizar la sesión de depuración, el objetivo es que cada uno pueda comprobar por sí mismo su correcta comprensión de los ejemplos del tutorial.

	Sesión de depuración suma_03_SC
1	¿Qué comando **gdb** se puede usar para ver el punto de entrada? Si no conocemos ese comando… ¿qué problemas tendríamos para depurar un programa como éste? Reconocer la importancia de dominar no sólo el modo gráfico **ddd**, sino también la línea de comandos **gdb**.
2	¿Qué diferencia hay entre los comandos **Next** y **Step**? (Pista: texto de ayuda) ¿Y entre esos comandos y su versión **<…>i**? Aprovechando que por primera vez incorporamos lenguaje C, poner un breakpoint en **call suma** y probar las cuatro variantes, anotando a dónde lleva exactamente cada una de ellas, y por qué. (Pista: Machine Code Window)
3	Editar el **formato** para que sea idéntico al de suma_02 (acabado en \n), reconstruir el programa y ejecutarlo. Explicar con precisión por qué se obtiene GCC: (Ubuntu 4.4 .3-4ubuntu5) 4.4.3. (Pista: objdump -s)
4	Obtener el código ensamblador generado para suma (no con ddd->Machine Code Window, sino con gcc) y compararlo con nuestra suma_01. ¿Qué diferencias hay? Sugerencia: quitar información de depuración para simplificar el listado.
5	Probar las opciones ddd-> Data-> Display Local Variables/Display Arguments. ¿Qué significa "value optimized out"? Pista: ir avanzando en **suma** con **Stepi** hasta que desaparezca el mensaje "optimized out". ¿Cuándo desaparece? Es posible poner un breakpoint sobre la Machine Code Window, no tiene por qué ser en la Source Window.
6	También se puede compilar el módulo C sin optimización. No hace falta re-ensamblar el módulo ASM, basta con re-enlazar el ejecutable. Comprobar si sigue saliendo el mensaje. ¿Qué direcciones tienen las variables locales **i** y **res** tras dicho cambio? Realizar un dibujo del marco de pila de **suma** sin optimización.

Tabla 5: preguntas de autocomprobación (suma_03_SC)

Las siguientes preguntas se refieren al programa suma_04_SC de la Figura 7. Se espera que cada uno pueda comprobar por sí mismo su correcta comprensión de los ejemplos del tutorial, realizando la sesión de depuración (ejecutando paso a paso usando ddd), y respondiendo estas preguntas.

	Sesión de depuración suma_04_SC
1	Obtener el código ensamblador generado para **suma** (no con ddd->Machine Code Window, sino con gcc) y compararlo con el anterior **suma_03**. ¿Qué diferencias hay? Sugerencia: quitar información de depuración para simplificar el listado. Otra sugerencia: comparar también con el **suma.s** original de la Figura 1.
2	Una de esas diferencias nos hace pensar que nuestra versión ensamblador de **suma** no implementa exactamente un bucle **for**, y podría producir un resultado incorrecto para cierto tamaño de la lista… ¿Cuál? ¿Por qué?
3	Otras diferencias están en el manejo de pila. Explicar dichas diferencias. Los curiosos pueden buscar **__printf_chk flag** con Google.
4	Ejecutar **nm** sobre ambos objetos C/ASM y sobre el ejecutable, indicando qué significa cada letra y fijándose en los valores (direcciones) asociados con cada símbolo. Notar que las direcciones en el ejecutable son definitivas. ¿Qué símbolos carecen de dirección? ¿Cómo es posible que **_start** y **lista** tengan la misma dirección en los objetos? ¿Por qué no tienen la misma dirección en el ejecutable?
5	¿Cómo es posible que aún queden símbolos sin definir (U) en el ejecutable?
6	Ejecutar **nm** sobre los objetos y ejecutable del ejemplo anterior **suma_03**, y explicar las diferencias con **suma_04**: ¿Por qué el módulo C tiene ahora símbolos indefinidos? ¿Cuál podría ser el motivo de que los símbolos anteriormente (d) sean ahora (D)? (Pista: comparar fuentes) ¿Por qué varía el nombre del símbolo **printf**?
7	En relación con **5**, ¿por qué cambian los nombres de los símbolos **printf** y **exit** del objeto al ejecutable, tanto en **suma_03** como en **suma_04**?
8	¿Cómo se podría comprobar la afirmación de que los símbolos no resueltos se rellenan a cero? (se afirma en la sección Ejercicio 4: suma_04_SC) (Pista: objdump). Localizar los 6 símbolos indefinidos y comprobar si se rellenan a cero. ¿Hay alguno que se rellene a valor distinto de cero? Comparar el objeto con el ejecutable.

Tabla 6: preguntas de autocomprobación (suma_04_SC)

Las siguientes preguntas se refieren al programa suma_05_C de la Figura 8. Se espera que cada uno pueda comprobar por sí mismo su correcta comprensión de los ejemplos del tutorial, realizando la sesión de depuración (ejecutando paso a paso usando ddd), y respondiendo estas preguntas.

	Sesión de depuración suma_05_C
1	Volver a probar las diferencias entre **Next/Step** y sus variantes **<...>i** aprovechando la llamada a **suma**. Poniendo un breakpoint en la primera línea de **main**, ¿qué efecto tiene cada uno de los comandos? Seguramente conviene visualizar Machine Code Window, Display->Locals/Args, y un volcado de pila, y pulsar varias veces cada comando a partir del breakpoint.
2	Pulsando una vez **Next** tras el breakpoint, identificar en el volcado de pila los argumentos (su valor se indica en el volcado Data->Display Args), y partiendo de ahí, identificar los componentes del marco de pila. Puede ser interesante utilizar Status->Backtrace (o **disas main**, en línea de comandos) para comprobar la dirección de retorno.
3	Comprobar la respuesta anterior reiniciando la ejecución y avanzando con **Stepi**. Esto nos permite comprobar dos valores del marco de pila que antes sólo podíamos suponer, basándonos en el desensamblado. ¿Cuáles?
4	Como hemos comprobado, **main** sí que tiene marco de pila (**_start** no tenía, recordar EBP=0). Recordando la primera pregunta de comprobación del guión, ¿cómo se volcarían los argumentos de main? (Pista: esta vez, el segundo argumento es char*argv[], y convendría usar Examine <n> **<qué>** bytes ***<argv>**. Recordar que se pueden ajustar los argumentos con **set args**.)
5	Con **gcc –S** (y quitando depuración) podemos consultar el código ensamblador generado por **gcc** para este programa. Nos debería sonar todo, salvo algunos detalles: Se aplica una máscara al puntero de pila. ¿Cuál, y qué efecto produce? (Pista: alineamiento). Nosotros usamos **.int** para declarar enteros y arrays. ¿Qué usa **gcc**? Nosotros usamos el contador de posiciones y aritmética de etiquetas para calcular la longitud del array. ¿Qué usa **gcc**? Nosotros hemos usado **push** para introducir argumentos en pila, aunque en transparencias en clase hemos visto otros métodos. ¿Cuál usa **gcc**?

Tabla 7: preguntas de autocomprobación (suma_05_C)

Las siguientes preguntas se refieren al programa suma_07_Casm de la Figura 10. Se ofrecen para que cada uno pueda comprobar por sí mismo su correcta comprensión de los ejemplos del tutorial.

	Preguntas de autocomprobación: suma_07_Casm
1	Comparar el código ensamblador generado por **gcc** para el ejemplo anterior (suma_06_CS) y para éste. ¿Hay alguna diferencia?
2	No necesitamos declarar como sobrescrito ninguno de los registros usados, aunque por distintos motivos. ¿Cuántos motivos distintos hay, y a qué registros se aplica cada uno? (Notar que la sentencia **asm()** implementa la función completa).
3	Por motivos estéticos, a veces se terminan las líneas ASM con "\n" y otras con "\n\t". ¿Por qué en este caso apenas se ha usado "\t"? Explicar qué edición estética realiza la sentencia **asm()** sobre la línea ASM insertada. (Pista: hacer pruebas con más/menos líneas, con/sin "\n\t", y consultar el ensamblador generado por **gcc**).
4	Esa edición estética delata que la sentencia **asm()** está pensada inicialmente para una única línea ASM. ¿En qué se nota?

Tabla 8: preguntas de autocomprobación (suma_07_Casm)

Las siguientes preguntas se refieren al programa suma_08_Casm de la Figura 11. Se ofrecen para que cada uno pueda comprobar por sí mismo su correcta comprensión de los ejemplos del tutorial.

	Preguntas de autocomprobación: suma_08_Casm
1	Comparar el código ensamblador generado por **gcc** para el ejemplo anterior (suma_07_Casm) y para éste. ¿Hay alguna diferencia?
2	Comparar el código generado comentando y descomentando "cc" de la lista clobber. ¿Hay alguna diferencia?
3	No necesitamos declarar ningún otro sobrescrito, pero por un motivo distinto que en el ejemplo anterior. ¿Por qué?
4	Si **res** es variable de salida, ¿por qué se le ha indicado restricción "+r", en lugar de "=r"?
5	Volver a explicar por qué en este caso se prefiere acabar la línea con "\n" en lugar de "\n\t"

Tabla 9: preguntas de autocomprobación (suma_08_Casm)

Las siguientes preguntas se refieren al programa suma_09_Casm de la Figura 12. Se ofrecen para que cada uno pueda comprobar por sí mismo su correcta comprensión de los ejemplos del tutorial.

	Preguntas de autocomprobación: suma_09_Casm
1	Repasar el código ensamblador generado por **gcc** para las tres versiones. ¿Hay alguna diferencia?
2	En la versión 3 se ha añadido un clobber que antes no estaba (ver Figura 10). ¿Acaso no sirve para nada ese clobber? ¿No hay diferencias en el código ensamblador generado?
3	En la versión 3 se han escrito los registros con dos símbolos %, en lugar de uno (como en lla Figura 10). ¿Qué pasa si se escriben como antes? ¿Por qué no pasaba eso antes?
4	¿Cuántos elementos tiene el array? ¿Cuánta memoria ocupa? ¿Cuánto vale la suma? ¿Qué fórmula se usa para calcular una suma como esa? ¿Cómo se llaman ese tipo de sumas?
5	El código C imprime un mensaje diciendo N*(N+1)/2=, pero luego calcula (SIZE-1)*(SIZE/2). ¿Cuál es la fórmula correcta?
6	Esa línea viene comentada con /* OF */. ¿Qué puede significar ese comentario? ¿Qué se puede decir acerca de la forma de escribir esa fórmula? Si es por "incomodidad para calcular la fórmula", ¿qué se podría haber hecho para evitar de golpe cualquier incomodidad? ¿Cómo se escribiría entonces, más cómodamente, la fórmula, y toda la instrucción **printf**?
7	En la función crono… ¿cómo se lee el tipo del primer argumento? (se puede consultar libro CS:APP [1], pág.287) ¿Qué formato **printf** se usa para imprimir el segundo? ¿Por qué se pasa por referencia el primer argumento de **gettimeofday**? ¿Por qué se pone a NULL el segundo? ¿Por qué se multiplica por 1E6 una de las restas, y la otra no? ¿Por qué el primer formato **printf** acaba con \t, en lugar de con \n? ¿Qué significa el formato %9ld usado en el segundo **printf**, por qué no se usa %9d, o sencillamente %d?
8	¿Hay alguna esperanza de ganar a **gcc** haciendo el tipo de cosas que venimos haciendo con **suma**? (pregunta retórica)

Tabla 10: preguntas de autocomprobación (suma_09_Casm)

Las siguientes preguntas se refieren al programa *popcount* de la Sección 4.1.

	Cuestiones sobre popcount.c
1	Dar una respuesta precisa a la primera pregunta (primer párrafo) de la Sección 4.1: en el peor caso, cuando todos los elementos tienen todos los bits activados… ¿cómo de grande puede ser N sin que haya *overflow*, si acumulamos la suma de bits en un **int**? ¿Y si se acumulara en un **unsigned**?
2	Diseñar la fórmula sugerida en el cuarto párrafo. ¿Cómo se ha razonado ese cálculo?
3	¿Por qué necesitaremos declarar la lista de enteros como **unsigned**? (comentado en el tercer párrafo) ¿Qué problema habría si se declarara como int? ¿Notaríamos en nuestro programa la diferencia? En caso negativo… ¿qué tendría que suceder para notar la diferencia?
4	En la 3ª versión (ASM) se han escogido restricciones en registros, "+r" y "r". ¿Cómo afectaría a las prestaciones que la primera restricción y/o la segunda fueran memoria "+m"/"m"? Comprobarlo con –O2, cambiando primero una de ellas, luego otra, luego ambas y midiendo 10 tiempos. Si se ha usado la primera instrucción ASM sugerida, también surge un error. ¿Cuál? ¿Por qué?
5	Si las restricciones a registro pueden ser mucho mejores que las restricciones a memoria… ¿Por qué entonces usamos sólo una restricción a registro en la versión 5ª, y las demás a memoria?
6	Realizar un dibujo de cómo funciona una iteración del algoritmo SSSE3 (5ª versión), con valores de elemento que causen que se use toda la tabla LUT, preferiblemente no en orden (porque entonces no quedaría clara la operación de baraje).
7	La versión 3 probablemente producirá resultados extraños, porque no sea mejor que la anterior (versión 2, incluso usando restricciones a registros) y/o porque tarde lo mismo independientemente del nivel de optimización. Intentar buscar explicación a ambas características, comparando los códigos ASM generados.
8	Realizar dos gráficas Calc o Excel del tipo "columnas 3D", una mostrando todos los resultados y otra mostrando la mejor versión C y las versiones ASM que le superan. ¿Qué ha tenido más impacto, mejorar la programación C o usar ASM?

Tabla 11: preguntas de autocomprobación (popcount.c)

Las siguientes preguntas se refieren al programa *parity* de la Sección 4.2.

	Cuestiones sobre parity.c
1	Diseñar la fórmula sugerida en el cuarto párrafo de la sección 4.2. ¿Cómo se ha razonado ese cálculo?
2	¿Por qué necesitaremos declarar la lista de enteros como **unsigned**? (comentado en el tercer párrafo) ¿Qué problema habría si se declarara como int? ¿Notaríamos en nuestro programa la diferencia? En caso negativo… ¿qué tendría que suceder para notar la diferencia?
3	En la 3ª versión (aplicar máscara al final) se dejó pendiente comparar con la 2ª para ver si la mejora es tan importante. ¿Lo es? ¿Para todos los niveles de optimización? En caso de que no lo sea, comparar los códigos ASM generados para encontrar una explicación (o comentar lo extraño del caso, si los fuentes no sirven para defender la explicación) ¿Cómo afectaría a las prestaciones que la primera restricción o la segunda fueran memoria "+m"/"m"? Comprobarlo con –O2, cambiando primero una de ellas, luego otra, luego ambas y midiendo 10 tiempos. Si se ha usado la primera instrucción ASM sugerida, también surge un error. ¿Cuál? ¿Por qué?
4	La 4ª versión (paso a ASM de la 3ª) probablemente sea la versión "extraña" de este ejemplo, porque no sea mejor que la anterior (incluso usando restricciones a registros) y/o porque tarde lo mismo independientemente del nivel de optimización. Intentar buscar explicación a ambas características, comparando los códigos ASM generados.
5	En la 4ª versión (ASM) se han escogido restricciones en registros, "+r" y "r". ¿Cómo afectaría a las prestaciones que la primera restricción o la segunda fueran memoria "+m"/"m"? Comprobarlo con –O2, cambiando primero una de ellas, luego otra (no ambas), y midiendo 10 tiempos. Si se ha usado la primera instrucción ASM sugerida, probablemente no puedan ponerse ambas restricciones a memoria, porque surja un error que no puede corregirse (a diferencia de la pregunta 4 de popcount). ¿Cuál error? ¿Por qué?
6	En la 5ª versión (XOR en árbol) se pidió razonar por qué funciona ese algoritmo (x^=x>>j para j=16,8,4,2,1), y en qué propiedades de XOR se basa. Responder.
7	En la 6ª versión (aprovechar PF) se indica que ahora no nos parece poco elegante hacer la reducción en árbol "a mano". Se indica como razón que sólo hay dos desplazamientos. Comentar sobre la elegancia del código ASM final, sobre que gcc no sea capaz de generar ese tipo de código a partir del código C, y sobre que la traducción que hemos hecho ha sido algo "libre".
8	En la 6ª versión se usa en *clobber* EDX. Probar a quitarlo y recompilar con los tres niveles de optimización. ¿Pasa algo? ¿Para cuáles niveles? ¿Por qué? La explicación debe basarse en el código ASM generado.
9	Realizar dos gráficas Calc o Excel del tipo "columnas 3D", una mostrando todos los resultados y otra mostrando la mejores versiones, C y ASM. ¿Qué ha tenido más impacto, mejorar la programación C o usar ASM?

Tabla 12: preguntas de autocomprobación (parity.c)