Preguntas de autoevaluación

Estructura de Computadores - Práctica 2

José Antonio Álvarez Ocet

Preguntas de autocomprobación: suma_08_Casm

1. Comparar el código ensamblador generado por *gcc* para el ejemplo anterior (suma_07_Ca asm) y para éste. ¿Hay alguna diferencia?

El código generado para suma_08 se diferencia del generado para suma_07 en que comprueba el caso en el que longlista = 0. Por otra parte hace uso de la instrucción addl $1, %edx en lugar de usar inc.

2. Comparar el código generado comentando y descomentando "cc" de la list ta clobber. ¿Hay alguna diferencia?

Compilando con las opciones -fno-omit-frame-pointer -m32 -O1 -S en gcc versión gcc (Ubuntu 5.2.1-22ubuntu2) 5.2.1 20151010 no se obtiene ninguna diferencia. La utilidad diff reporta: *Los archivos suma_08.s y suma_08cc.s son idénticos.*

3. No necesitamos s declarar ningún otro sobrescrito, pero por un motivo distinto que en el ejemplo anterio r. ¿Por qué?

No necesitamos declarar ningún sobrescrito porque todos los registros que utilizamos son puestos por gcc en la compilación. Lo hacemos así ya que necesitamos utilizar los registros que guardan las variables locales de la función y los argumentos y no podemos (salvo para estos últimos) saber en qué registros estarán guardados.

4. Si *res* es variable e de salida, ¿po or qué se le ha i indicado restric cción "+r", en lu ugar de "=r"?

Necesitamos indicar que %eax es de entrada salida para que se efectúe la operación:
movl $0, %eax
Es decir, que el registro %eax (que se corresponde con res) sea inicializado con un valor por defecto. Si no indicamos que también es de entrada (ya que en la instrucción add se utiliza su valor para la entrada), gcc interpreta que no importa su valor por defecto y el programa da valores no deseados.

5. Volver a explica r por qué en es este caso se prefiere acabar la l ínea con \n en lugar de \n \t

Se prefiere acabar con \n por si se añaden nuevas instrucciones, ya que si se pusiera \n\t al añadir una instrucción nueva esta quedaría desalineada del resto.
No obstante es una preferencia estética que no tiene efecto en la correcta ejecución del programa.

Preguntas de autocomprobación: suma_09_Casm

1. Repasar el código ensamblador generado por gcc para las tr es versiones. ¿Hay alguna diferencia?

El código de suma2 es idéntico al de suma_08 mientras que el código de suma3 es idéntico al de suma_07. Por otra parte, el código de suma1 es la versión implementada completamente en C, idéntica en este caso a la de suma_05.

2. En la versión 3 s se ha añadido un clobber que antes no estaba (ver Figura 10). ¿Acaso no sirve para nada ese clobber? ¿No h hay diferencias en el código ensamblador generado?

El código generado es idéntico (salvo el uso del sinónimo pushl en lugar de push). Sin embargo, el declarar el registro %ebx como clobber tiene una clara utilidad: el compilador gestiona el guardado del valor de este registro en la pila y su posterior restauración, ahorrando código que podría dar fallos si no lo implementaramos correctamente.

3. En la versión 3 s se han escrito los registros con n dos símbolos %, en lugar de uno (como en la Figura 10). ¿Qué pasa si se escriben como antes? ¿Por qué no pasaba eso antes?

Los escribirmos con dos % para indicar a gcc que nos referimos a registros y no a los argumentos que especificaríamos en la lista de entradas y salidas. Al especificar el nuevo clobber necesitamos utilizar los dos % % para evitar que gcc confunda los nombres de registros con nombres de variables.

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 este tipo de sumas?

El array tiene $1 >> 16$ elementos, es decir, $2^{16} = 65536$ elementos. Cada entero ocupa 4 bytes, es decir que en total el array ocupará $65536 * 4 \simeq 262kB$. La suma vale 2147450880. Puede utilizarse la fórmula para la suma de una sucesión aritmética:

$$\sum_{i=1}^{n} a_i = \frac{n(a_1 + a_n)}{2}$$

5. El código C imprime un mensaje diciendo $\frac{N*(N+1)}{2}$, pero luego calcula $(SIZE - 1) * \frac{SIZE}{2}$. ¿Cuál es la fórmula correcta?

Ambas fórmulas son correctas ya que en este caso $SIZE = N + 1$. Partiendo de la fórmula anterior y sabiendo que a i = i tenemos:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Donde n será el número de elementos sumados, que, en este caso, coincide con SIZE-1. Sustituyéndolo obtenemos la fórmula implementada.

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?

Esta línea puede indicarnos que, escribiendo la fórmula de otra forma puede producirse *overflow*. La fórmula está escrita de tal manera que se evita realizar el producto: $SIZE * (SIZE - 1)$. Esto se debe a que este producto puede superar 2^{32} , lo que produciría overflow.
Se podría haber evitado la incomodidad utilizando un tipo de dato que tenga más capacidad, como es long unsigned. De esta forma la instrucción printf podría escribirse:
printf("N*(N+1)/2 = %lu\n", ((SIZE-1)*SIZE)/2);

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 immprimir 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 uusa %9d, o sencillamente %d?

El primer tipo del argumento es un puntero a una función que devuelve un int. El formato para imprimir el segundo argumento es %s: %9ld us, es decir, se imprime la string y luego el tiempo con un espacio de al menos 9 caracteres.
La función gettimeofday modifica el struct apuntado por el puntero que le pasamos para añadir el tiempo. Se pone NULL al segundo porque no nos interesa el valor que se nos devolvería (timezone). Se multiplica para que todo esté en la misma unidad, ya que se devuelve en y microsegundos (10^{-6}s).
El formato acaba con p̂ara introducir un espacio. Se utilizar %9ld para imprimir un long con 9 espacios.

8. ¿Hay alguna esperanza de ganar a gcc haciendo el tipo de cosas que venimos haciendo con suma? (pregunta retórica)

Utilizando asm podemos "ganar a gcc.[en] caunto a eficiencia de cómputos.

Cuestiones sobre popcount.c

1. Dar una respuesta precisa a la primera pregunta (primer pár rrafo) de la Sec ción 4.1: en el peor casos, cuando todos los elementos tienen todos los bits activados... ¿cómo de grande puede ser N sin q que haya overflow, si acumulamos la suma de bits en un int? ¿Y si se acumulara en un unsigned?

Basta resolver la ecuación:

$$2^{31} - 1 = 32 * n$$

De la que obtenemos $n \simeq 2^{26}$. En el caso de que sea unsigned tendríamos un bit más para el almacenaje, por lo que tendríamos que resolver la ecuación:

$$2^{32} - 1 = 32 * n$$

Obteniendo como resultado $n \simeq 2^{27}$.

2. Diseñar la fórmula sugerida en el cuarto párrafo. ¿Cómo se ha razonado ese cálculo?

El siguiente razonamiento es exacto unicamente para potencias de 2.
El bit 0 de cada número vale 1 cada 2 números. En general, el bit n vale 1 en grupos de 2 n+1 unos cada 2 n+1 números. Aprovechando este razonamiento vemos que en toda cifra que no sea nula tendremos un total de 16 unos. Podemos entonces aplicar la siguiente fórmula, con la que calculamos el número de cifras significativas:

$$P_c = \frac{log_2(SIZE)}{2} * SIZE$$

3. ¿Por qué necesi itaremos declar rar la lista de en nteros como unnsigned? (comentado en el ter rcer párrafo) ¿Qué problema habría si se de clarara como int? ¿Notaríamos en nuestro programa la diferencia? En caso negativo... ¿Qué tendría que suceder para notar la diferencia?

Para que en el caso de que hubiera bits de signo estos no se dupliquen al hacer el desplazamientohacia la derecha. Que se podrían duplicar bits caso de que hubiera un número negativo.
No, ya que todos los números con los que trabajamos son positivos. Tendríamos que poner algún bit negativo.

4. En la tercera 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 s. Si se ha usado la primera instrucción ASM sugerida, también surge un error r. ¿Cuál? ¿Por qué?

Usando registros para ambas variables, 10946 us; mientras que si usamos memoria para ambas variables, 45647 us. Al intentar poner la segunda en memoria se obtiene el error:

popcount.c:67: Error: no se dio un sufijo mnemónico de instrucción y ningún operando

Que podemos solucionar indicando la instrucción shrl en vez shr. Esto sucede ya que no puede determinarse el tamaño de x en memoria. Los nuevos tiempos: almacenando sólo la segunda varibable en memoria, 27895 us; mientras que si almacenamos ambas en memoria, 42265 us.

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 quinta versión, y las demás a memoria?

Porque son arrays. Su contenido no cabe en un registro.

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.

El tiempo es similar en todos los niveles de optimización pero es peor que el de la segunda versión a partir de O2. Si observamos los códigos ensamblador generados veremos que son muy similadres. Sin embargo, el código de la versión 3 realiza una comparación que la versión 2 no realiza.