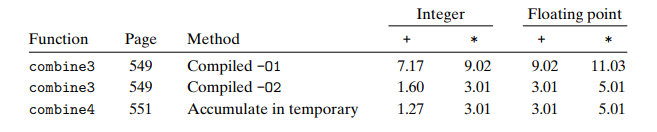
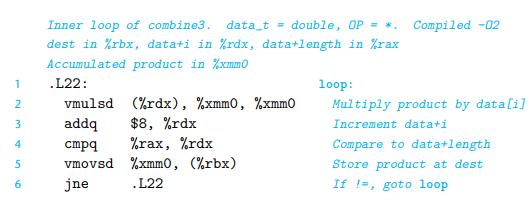
**Ejercicios 5.4, 5.5, 7.1, 7.2 y 7.3 del libro**

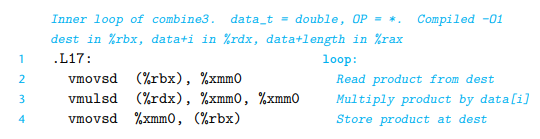
**5.4** Cuando usamos gcc para compilar combine3 con la opción de línea de comandos -O2, obtenemos código con un rendimiento de CPE sustancialmente mejor que con -O1:



Logramos un rendimiento comparable al de combine4, excepto en el caso de suma entera, pero incluso mejora significativamente. Al examinar el código ensamblador generado por el compilador, encontramos una variante interesante para el ciclo interno:



Podemos comparar esto con la versión creada con el nivel de optimización 1:





Vemos que, además de cierto reordenamiento de instrucciones, la única diferencia es que la versión más optimizada no contiene el vmovsd que implementa la lectura desde la ubicación designada por dest (línea 2):

Respuesta:

**A.** ¿Cómo difiere el papel del registro %xmm0 en estos dos bucles?

**B**. ¿La versión más optimizada implementará fielmente el código C de combine3, incluso cuando haya un alias de memoria entre el destino y los datos del vector?

**C.** Explique por qué esta optimización conserva el comportamiento deseado, o dé un ejemplo en el que produciría resultados diferentes en códigos menos optimizados.

Respuesta:

A. En el código menos eficiente, el registro %xmm0 se utiliza como un valor temporal que se establece y utiliza en cada iteración del bucle. En el código más optimizado, se utiliza como una variable "acc" en "combine4", acumulando el producto de los elementos vectoriales. Sin embargo, la diferencia con "combine4" es que la ubicación de "dest" se actualiza en cada iteración mediante la segunda instrucción "vmovsd".

La optimización en el bucle optimizado permite que %xmm0 actúe como un acumulador, manteniendo el producto acumulado de los elementos del vector. Esto se logra al no leer el valor de "dest" en cada iteración, sino utilizando el valor previamente almacenado en %xmm0 al comienzo del bucle. Al eliminar la instrucción "vmovsd" para leer el valor de "dest", se ahorra una operación de carga de memoria innecesaria en cada iteración.

Esta versión optimizada funciona de manera similar al siguiente código en C:

/\* Asegurarse de que dest se actualice en cada iteración \*/

void combine3w(vec\_ptr v, data\_t \*dest)

{

long i;

long length = vec\_length(v);

data\_t \*data = get\_vec\_start(v);

data\_t acc = IDENT;

/\* Inicializar en caso de que length <= 0 \*/

\*dest = acc;

for (i = 0; i < length; i++) {

acc = acc OP data[i]; // Acumular el producto

\*dest = acc; // Actualizar dest con el valor acumulado

}

}

B. Ambas versiones de "combine3" tendrán la misma funcionalidad, incluso con aliasing de memoria. La optimización realizada en el bucle no afectará el comportamiento esperado del código. Incluso si hay aliasing de memoria entre "dest" y el vector de datos, el producto acumulado se actualizará correctamente y se almacenará en "dest" al final de cada iteración.

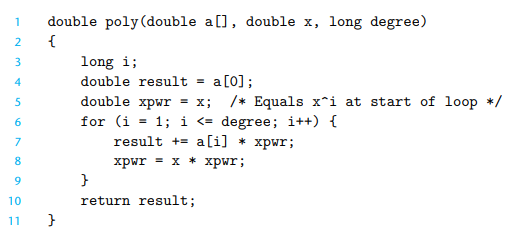
C. La eliminación de la instrucción "vmovsd" para leer el valor de "dest" en cada iteración reduce el número de operaciones de memoria y mejora la eficiencia del código. Esto es posible porque el valor de "dest" no cambia dentro del bucle y se puede confiar en el valor almacenado en %xmm0 como el producto acumulado actualizado.

Esta transformación se puede realizar sin cambiar el comportamiento del programa porque, excepto en la primera iteración, el valor leído de "dest" al comienzo de cada iteración será el mismo valor escrito en este registro al final de la iteración anterior. Por lo tanto, la instrucción de combinación puede simplemente usar el valor que ya está en %xmm0 al comienzo del ciclo.

**5.5** Supongamos que deseamos escribir una función para evaluar un polinomio, donde un polinomio de grado n se define para tener un conjunto de coeficientes a0, a1, a2,...,an. Para un valor x, evaluamos el polinomio calculando:



Esta evaluación puede ser implementada por la siguiente función, teniendo como argumentos un arreglo de coeficientes a, un valor x, y el grado polinomial grado (el valor n en la Ecuación 5.2). En esta función, calculamos tanto los términos sucesivos de la ecuación como las potencias sucesivas de x dentro de un solo bucle:



**A.** Para el grado n, ¿cuántas sumas y cuántas multiplicaciones realiza este código?

**B.** En nuestra máquina de referencia, con operaciones aritméticas que tienen las latencias que se muestran en la Figura 5.12, medimos el CPE para esta función como 5.00. Explique cómo surge este CPE en función de las dependencias de datos formadas entre iteraciones debido a las operaciones que implementan las líneas 7 y 8 de la función.

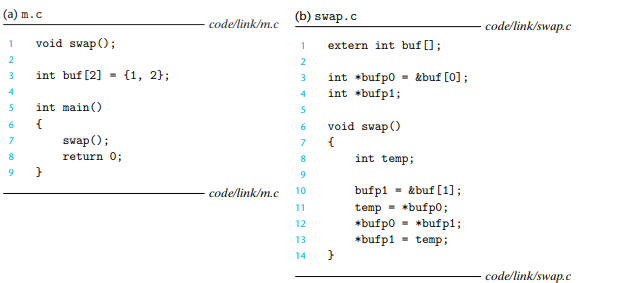
**Respuesta:**

A. Si consideramos degree como n, podemos observar que la función, al tener un ciclo de n iteraciones en el que se realizan dos multiplicaciones y una suma, llevará a cabo 2n multiplicaciones y n sumas.

B. Se puede notar que el cálculo que afecta el rendimiento es la expresión xpwr = x \* xpwr. Esta operación implica una multiplicación de coma flotante que toma 5 ciclos de reloj, y la siguiente iteración no puede comenzar hasta que se haya completado la anterior. Por otro lado, la actualización del resultado solo requiere una adición de punto flotante que toma 3 ciclos de reloj entre iteraciones consecutivas.

Debido a estas dependencias de datos, la ejecución de las operaciones de multiplicación y adición en diferentes iteraciones del bucle no puede hacerse en paralelo, lo que resulta en una limitación en el rendimiento. Además, la multiplicación en punto flotante tiene un mayor costo en términos de latencia en comparación con la adición en punto flotante.

**7.1** Este problema afecta a los módulos m.o y swap.o de la figura 7.5. Para cada símbolo definido o referenciado en swap.o, indique si tendrá o no una entrada en la tabla de símbolos en la sección .symtab del módulo swap.o. En caso afirmativo, indique el módulo que define el símbolo (swap.o o m.o), el tipo de símbolo (local, global o externo) y la sección (.text, .data, .bss o COMMON) a la que está asignado en el módulo.

****

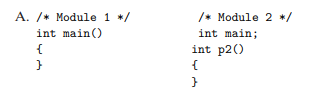
**Respuesta:**

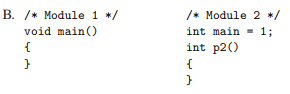
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Symbol | .symtab entry ? | Symbol type | Module where defined | Section |
| buf | Sí | extern | m.o | .data |
| bufp0 | Sí | global | swap.o | .data |
| bufp1 | Sí | global | swap.o | COMMON |
| swap | Sí | global | swap.o | .text |
| temp | No | - | - | - |

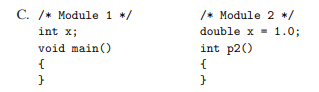
En lo que respecta a las entradas en la tabla de símbolos (.symtab), podemos decir que buf, bufp0, bufp1 y swap se encuentran en ella. Esto se debe a que buf, bufp0 y bufp1 son variables globales, mientras que swap es una función global. En cambio, temp es una variable local, por lo que no aparece en la tabla de símbolos.

En cuanto a la sección a la que pertenecen, podemos decir que buf y bufp0 se encuentran en la sección .data, ya que se inicializan en el momento en que se declaran. Por otro lado, bufp1 se encuentra en la sección COMMON, ya que es una variable global no inicializada. En cuanto a swap, se encuentra en la sección .text, ya que se trata de una función.

**7.2** En este problema, REF(x.i) → DEF(x.k) denota que el enlazador asociará una referencia arbitraria al símbolo x en el módulo i a la definición de x en el módulo k. Para cada ejemplo que sigue, use esta notación para indicar cómo el enlazador resolvería las referencias al símbolo definido de forma múltiple en cada módulo. Si hay un error de tiempo de enlace (regla 1), escriba "ERROR". Si el enlazador elige arbitrariamente una de las definiciones (regla 3), escriba "UNKNOWN".







**Respuesta:**

Teniendo en cuenta las definiciones de símbolos fuertes y débiles, y las reglas establecidas, podemos llegar a las siguientes conclusiones:

\* Regla 1. No se permiten múltiples símbolos fuertes con el mismo nombre.

\* Regla 2. Dado un símbolo fuerte y múltiples símbolos débiles con el mismo nombre, elige el símbolo fuerte.

\* Regla 3. Dados múltiples símbolos débiles con el mismo nombre, elija cualquiera de los símbolos débiles

1. En este caso, el enlazador elegirá el símbolo fuerte definido en el módulo 1 sobre el símbolo débil definido en el módulo 2, siguiendo la regla 2:

* (a)REF(main.1) → DEF(main.1)
* (b) REF(main.2) → DEF(main.1)

Esto se debe a que el símbolo mencionado en (a) hace referencia a un símbolo fuerte, mientras que el mencionado en (b) hace referencia a un símbolo débil. Por lo tanto, el enlazador elegirá el símbolo fuerte definido en el módulo 1 para satisfacer ambas referencias.

B. En este caso, se produce un error ya que cada módulo define un símbolo fuerte principal, violando la regla 1. Es decir, hay múltiples símbolos fuertes con el mismo nombre.

1. En este caso, el enlazador elegirá el símbolo fuerte definido en el módulo 2 sobre el símbolo débil definido en el módulo 1, siguiendo la regla 2:

* (a)REF(x.1) → DEF(x.2)
* (b) REF(x.2) → DEF(x.2)

Esto se debe a que ambos símbolos mencionados hacen referencia a símbolos débiles, por lo que el enlazador elegirá cualquiera de los dos para satisfacer las referencias. En este caso, se elegirá el símbolo fuerte definido en el módulo 2.

**7.3** Sean*a* y *b* módulos de objetos o bibliotecas estáticas en el directorio actual, y *a*→*b* denote que *a* depende de *b*, en el sentido de que *b* define un símbolo al que *a* hace referencia. Para cada uno de los siguientes escenarios, muestre la línea de comando mínima (es decir, una con la menor cantidad de argumentos de archivo de objeto y biblioteca) que permitirá que el vinculador estático resuelva todas las referencias de símbolos.

**A**. p.o → libx.a

**B.** p.o → libx.a → liby.a

**C.** p.o → libx.a → liby.a and liby.a → libx.a → p.o

**Respuesta:**

A. En este caso, se puede afirmar que el programa objetivo p.o depende de la biblioteca estática libx.a. Para resolver las referencias de símbolos en p.o, es necesario incluir la biblioteca libx.a en el comando de enlace. Por lo tanto, se puede usar la línea de comando "gcc p.o libx.a" que especifica el archivo de objeto p.o y la biblioteca libx.a.

linux> gcc p.o libx.a

B. En este caso, se puede decir que el programa objetivo p.o depende de dos bibliotecas estáticas: libx.a y liby.a. Para resolver todas las referencias de símbolos, ambas bibliotecas deben incluirse en el comando de enlace. Por lo tanto, se puede usar la línea de comando "gcc p.o libx.a liby.a" que especifica el archivo de objeto p.o y las bibliotecas libx.a y liby.a.

linux> gcc p.o libx.a liby.a

C. En este caso, se presenta una dependencia circular entre las bibliotecas libx.a y liby.a, lo que significa que libx.a depende de liby.a y liby.a depende de libx.a. Para resolver todas las referencias de símbolos, es necesario incluir tanto libx.a como liby.a en el comando de enlace, y se debe incluir libx.a dos veces debido a la dependencia circular. Por lo tanto, se puede usar la línea de comando "gcc p.o libx.a liby.a libx.a" que especifica el archivo de objeto p.o y las bibliotecas libx.a y liby.a, incluyendo libx.a dos veces para satisfacer la dependencia circular.

linux> gcc p.o libx.a liby.a libx.a