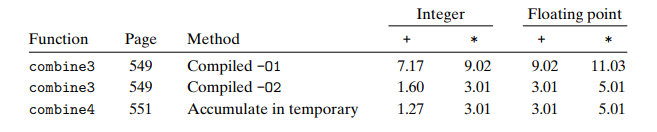
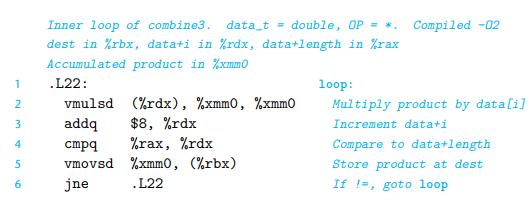
**Ejercicios 5.4, 5.5, 7.1, 7.2 y 7.3 del libro**

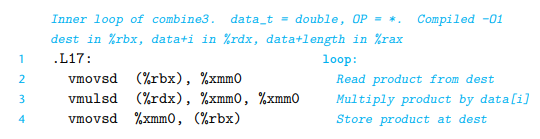
**5.4** Cuando usamos gcc para compilar combine3 con la opción de línea de comandos -O2, obtenemos código con un rendimiento de CPE sustancialmente mejor que con -O1:



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):

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

En el código menos óptimo, el registro %xmm0 se emplea como un valor temporal, fijándose y empleándose en cada vuelta del bucle. En contraste, en el código más eficiente, este registro se utiliza como una variable acc en combine4, acumulando el producto de los elementos vectoriales. No obstante, la distinción con combine4 radica en que la ubicación de dest se actualiza en cada iteración por medio de la segunda instrucción vmovsd.

La optimización realizada en el bucle mejorado permite que %xmm0 desempeñe la función de acumulador, manteniendo el producto acumulado de los elementos del vector. Para lograrlo, se evita la lectura del valor de dest en cada iteración, en su lugar se utiliza el valor previamente almacenado en %xmm0 al inicio del bucle. Al eliminar la instrucción vmovsd que lee el valor de dest, se evita una operación de carga de memoria innecesaria en cada iteración.

Esta versión optimizada funciona de manera muy similar al siguiente Código 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

}

}

1. ¿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?

Las dos versiones de combine3 tendrán una funcionalidad idéntica, incluso en presencia de aliasing de memoria. La optimización realizada en el bucle no afectará el comportamiento esperado del código. Aunque exista 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.

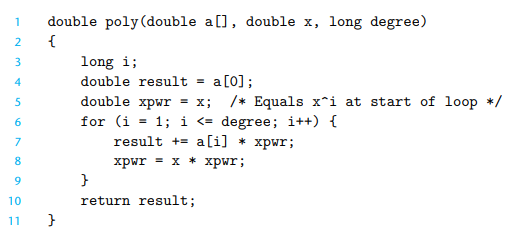
1. Explique por qué esta optimización conserva el comportamiento deseado, o dé un ejemplo en el que produciría resultados diferentes en códigos menos optimizados.

Al eliminar la instrucción vmovsd que lee el valor de dest en cada iteración, se reduce el número de operaciones de memoria y se 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

**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:

C:\Users\danie\OneDrive\Imágenes\Capturas de pantalla\Captura de pantalla (267).png

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:



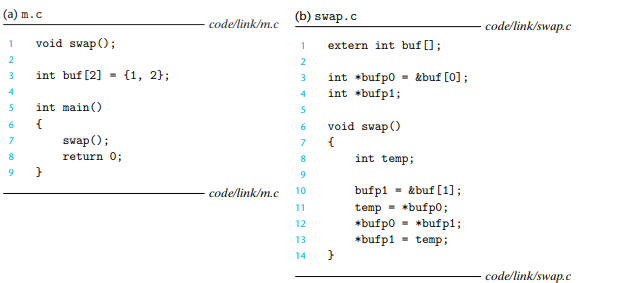
1. Para el grado n, ¿cuántas sumas y cuántas multiplicaciones realiza este código?

Si asignamos a degree el valor de n, podemos observar que la función realiza un bucle de n iteraciones, en el que se efectúan dos multiplicaciones y una suma en cada iteración. Por lo tanto, la función realiza un total de 2n multiplicaciones y n sumas.

1. 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.

El cálculo que limita el rendimiento de la función es la expresión xpwr = x \* xpwr, la cual requiere una multiplicación en coma flotante (5 ciclos de reloj) y la iteración actual no puede comenzar hasta que se haya completado la iteración anterior. Por otro lado, la actualización del resultado solo necesita una adición en punto flotante (3 ciclos de reloj) entre iteraciones sucesivas. Debido a estas dependencias de datos, no se puede ejecutar en paralelo las operaciones de multiplicación y adición en diferentes iteraciones del bucle, lo que resulta en una limitación en el rendimiento de la función. 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.

****

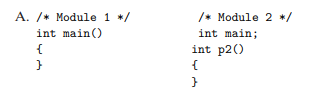
**Respuesta:**

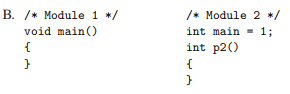
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Symbol | .symtab entry ? | Symbol type | Module where defined | Section |
| buf | Sí | extern | m.o | .data |
| bufp0 | Sí | global | swap.o | .data |
| bufp1 | Sí | global | swap.o | COMMON |
| swap | Sí | global | swap.o | .text |
| temp | No | - |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

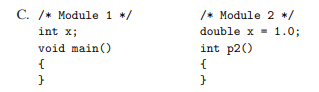
En el caso de buf, bufp0, bufp1 y swap, son incluidos en la tabla de símbolos (.symtab) ya que buf, bufp0 y bufp1 son variables globales y swap es una función global. Mientras que temp es una variable local y por lo tanto no se incluye en la tabla.

En cuanto a la sección a la que pertenecen, buf y bufp0 pertenecen a la sección .data ya que se inicializan en el momento en que se declaran. En cambio, bufp1 pertenece a la sección COMMON ya que es una variable global no inicializada. Por último, swap pertenece a 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 que:

Símbolos fuertes: Funciones y variables globales inicializadas.

Símbolos débiles: Las variables globales no inicializadas.

Y las siguientes reglas:

\* 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

Podemos determinar que:

**A.** El enlazador elige el símbolo fuerte definido en el módulo 1 sobre el símbolo débil definido en el módulo 2 (regla 2):

(a) REF(main.1) → DEF(main.1)

(b) REF(main.2) → DEF(main.1)

**B.** Esto es un error, porque cada módulo define un símbolo fuerte principal (regla 1).

**C.** El enlazador elige el símbolo fuerte definido en el módulo 2 sobre el símbolo débil definido en el módulo 1 (regla 2):

(a) REF(x.1) → DEF(x.2)

(b) REF(x.2) → DEF(x.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, el archivo objeto p.o tiene una dependencia 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, la línea de comando mínima sería "gcc p.o libx.a", en la que se especifica el archivo de objeto p.o y la biblioteca libx.a.

linux> gcc p.o libx.a

B. En este otro caso, el archivo objeto 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, la línea de comando mínima sería "gcc p.o libx.a liby.a", en la que se especifica el archivo de objeto p.o y las bibliotecas libx.a y liby.a.

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

C. Finalmente, en el caso C existe una dependencia circular entre las bibliotecas libx.a y liby.a. Esto implica que libx.a depende de liby.a y, a su vez, liby.a depende de libx.a. Para resolver todas las referencias de símbolos, se deben incluir tanto libx.a como liby.a en el comando de enlace, y libx.a debe incluirse dos veces debido a la dependencia circular. Por lo tanto, la línea de comando mínima sería "gcc p.o libx.a liby.a libx.a", en la que se 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