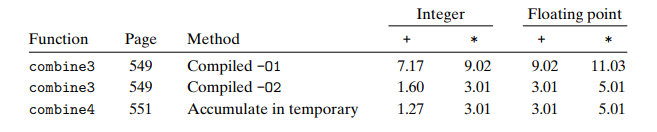
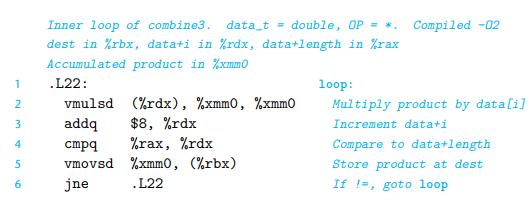
# Osliani Abel Figueiras Saucedo

## Ejercicio 5.4

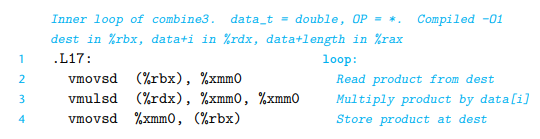
Cuando usamos gcc para compilar combine3 con la opción de línea de comandos -O2, obtenemos código con un rendimiento de CPE sustancialmente mejor que con -O1:



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):

**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.

**Solución:**

1. En el código menos eficiente, el registro %xmm0 se usa como un valor temporal que se establece y utiliza en cada iteración del bucle. Por otro lado, en el código más optimizado, se utiliza como una variable de acumulación acc en la función combine4, donde almacena el producto acumulado de los elementos vectoriales. A diferencia de combine4, en esta versión optimizada, 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, evitando así la necesidad de leer el 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 para leer el valor de dest, se reduce una operación de carga de memoria innecesaria en cada iteración. Esta versión optimizada tiene un funcionamiento muy similar al siguiente código en lenguaje 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

}

}

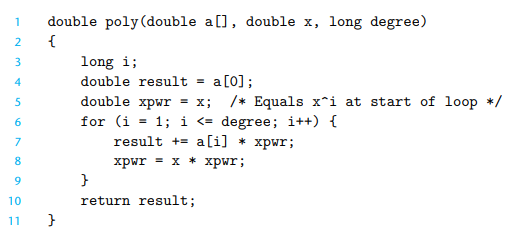
1. Ambas versiones de la función combine3 tienen la misma funcionalidad, incluso en presencia de aliasing de memoria. La optimización realizada en el bucle no afectará el comportamiento esperado del código. Incluso si existe aliasing de memoria entre dest y el vector de datos, el producto acumulado se actualizará correctamente y se almacenará en dest al final de cada iteración.
2. La eliminación de la instrucción vmovsd para leer el valor de dest en cada iteración permite reducir el número de operaciones de memoria y mejorar la eficiencia del código. Esto es posible gracias a que el valor de dest permanece constante dentro del bucle, por lo que se puede confiar en el valor almacenado en %xmm0 como el producto acumulado actualizado. Esta optimización se puede aplicar sin cambiar el comportamiento del programa, ya que, excepto en la primera iteración, el valor leído de dest al inicio de cada iteración será el mismo valor escrito en el registro al final de la iteración anterior. Por lo tanto, la instrucción de combinación puede utilizar simplemente el valor que ya se encuentra en %xmm0 al inicio del ciclo.

## Ejercicio 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:



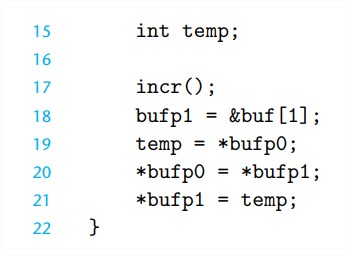
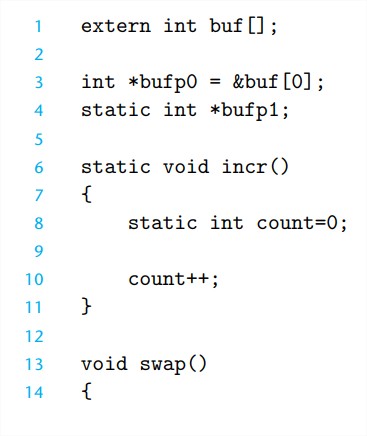
**A.** Para el grado n, ¿cuántas sumas y cuántas multiplicaciones realiza este código?

**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.

**Solución:**

1. Si consideramos que degree es equivalente a n, podemos decir que la función se compone de un ciclo con n iteraciones en el que se realizan dos multiplicaciones y una suma en cada iteración. Por lo tanto, la función realiza un total de 2n multiplicaciones y n sumas.
2. Se puede observar que el cálculo que limita el rendimiento en esta función es la expresión xpwr = x \* xpwr. Este cálculo requiere una multiplicación en coma flotante, que tiene un costo de 5 ciclos de reloj, y no se puede comenzar a calcular una iteración hasta que se haya completado la iteración anterior. Por otro lado, la actualización del resultado solo requiere una adición en punto flotante, que tiene un costo de 3 ciclos de reloj, entre iteraciones consecutivas. Debido a estas dependencias de datos, no es posible ejecutar en paralelo las operaciones de multiplicación y adición en diferentes iteraciones del bucle, 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.

## Ejercicio 7.1



Para cada símbolo definido y referenciado en swap.o, indique si se tiene una entrada en la tabla de símbolos en la sección .symtab en el módulo swap.o. Si es así, indique

el módulo que define el símbolo (swap.o o m.o), el tipo de símbolo (local, global, o extern), y la sección (.text, .data, o .bss) que ocupa en ese módulo.

**Solución:**

**Símbolo .symtab entry? Tipo de símbolo Módulo donde se definió Sección**

buf Si extern m.o .data

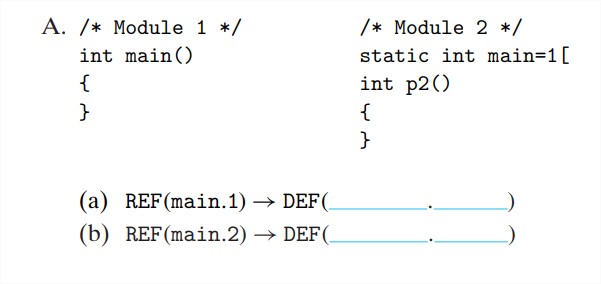
bufp0 Si global swap.o .data

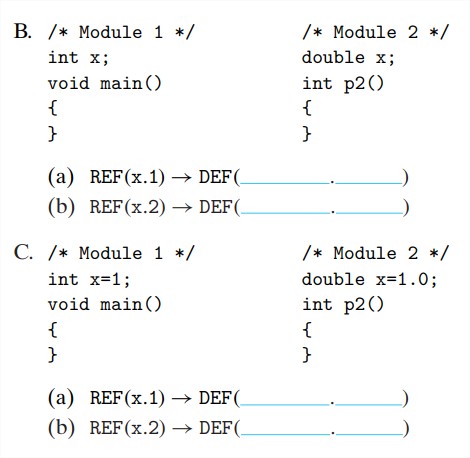
bufp1 Si global swap.o COMON

swap Si global swap.o .text

temp No - - -

## Ejercicio 7.2





**Solución:**

A. El enlazador elige el símbolo fuerte definido en el módulo 1 sobre el débil

símbolo definido en el módulo 2 (regla 2):

(a) REF(principal.1) → DEF(main.1)

(b) REF(principal.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 débil

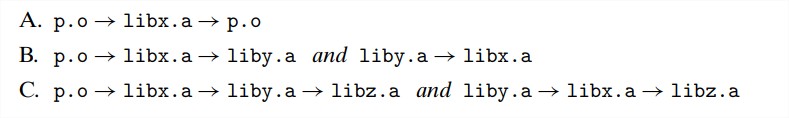
símbolo definido en el módulo 1 (regla 2):

(a) REF(x.1) → DEF(x.2)

(b) REF(x.2) → DEF(x.2)

## Ejercicio 7.3

Para cada uno de los siguientes escenarios, muestre el comando mínimo (es decir, uno con el menor número de argumentos de biblioteca y archivo de objeto) que permita que el enlazador estático resuelva todas las referencias de símbolos:



**Solución:**

A. linux> gcc p.o libx.a

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

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

En la primera línea, el programa "p.o" se enlaza con la biblioteca estática "libx.a" y se genera un archivo ejecutable. En la segunda línea, el programa "p.o" se enlaza con las bibliotecas estáticas "libx.a" y "liby.a", y se genera un archivo ejecutable. En la tercera línea, el programa "p.o" se enlaza con las bibliotecas estáticas "libx.a", "liby.a" y "libx.a", en ese orden, y se genera un archivo ejecutable.