**Respuesta a los ejercicios 5.4... 5.5... 7.2... 7.3...**

**Javier Enrique Casares Chavarri**

**Solución al Problema 5.4**

A. En el código menos optimizado, el registro %xmm0 se utiliza simplemente como un valor temporal, tanto para su configuración como para su uso en cada iteración del bucle. En cambio, en el código más optimizado, se utiliza de manera similar a la variable "acc" en "combine4", acumulando el producto de los elementos del vector. Sin embargo, la diferencia con "combine4" radica en que la ubicación "dest" se actualiza en cada iteración mediante la segunda instrucción "vmovsd".

Podemos observar que 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 si la longitud es <= 0 \*/

\*dest = acc;

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

acc = acc OP data[i];

\*dest = acc;

}

}

B. Ambas versiones de "combine3" tendrán la misma funcionalidad, incluso en casos de aliasing de memoria.

C. Esta transformación se puede realizar sin cambiar el comportamiento del programa, ya que, excepto en la primera iteración, el valor leído desde "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 utilizar el valor ya presente en %xmm0 al comienzo del bucle.

**Solución al Problema 5.5**

A. La función realiza n adiciones y n multiplicaciones en el bucle principal. Cada iteración del bucle realiza una adición (línea 7) y una multiplicación (línea 8).

B. El CPE (Cycles Per Element) de 5.00 se debe a las dependencias de datos que se forman entre las iteraciones debido a las operaciones de las líneas 7-8 de la función. En cada iteración, se necesita el valor calculado de xpwr (que se obtiene multiplicando x por el valor anterior de xpwr) para realizar la multiplicación en la línea 7. Sin embargo, el cálculo de xpwr no puede comenzar hasta que la multiplicación anterior haya finalizado, lo que crea una dependencia de datos entre las iteraciones. Esto resulta en un retraso en la ejecución de cada iteración y, por lo tanto, en un mayor CPE. El rendimiento se ve limitado por la latencia de las multiplicaciones y la necesidad de esperar a que se completen antes de poder realizar las siguientes operaciones en las iteraciones sucesivas.

**Solución al Problema 7.1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Símbolo | ¿Entrada en .symtab? | Tipo de símbolo | Módulo donde se define | Sección |
| buf | Sí | Externo | m.o | .data |
| bufp0 | Sí | Global | swap.o | .data |
| bufp1 | Sí | Global | swap.o | .data |
| swap | Sí | Global | swap.o | .text |
| temp | No | - | - | - |

**Solución al Problema 7.2**

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 "main" (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)

Estos ejemplos ilustran las reglas utilizadas por el enlazador de Unix para resolver símbolos globales definidos en más de un módulo. Comprender estas reglas es importante para evitar errores de programación. En el caso de (a), se aplica la regla 2 y se elige el símbolo fuerte definido en el módulo 1. En el caso de (b) en A y (a) en C, hay un error porque ambos módulos definen un símbolo fuerte con el mismo nombre. En (b) en C, se aplica la regla 2 y se elige el símbolo fuerte definido en el módulo 2.

**Solución al Problema 7.3**

A. Para resolver las referencias de símbolos en el escenario A, donde p.o depende de libx.a, el siguiente comando sería suficiente:

gcc p.o libx.a

B. En el escenario B, donde p.o depende de libx.a y libx.a depende de liby.a, el orden de los argumentos en el comando es importante. Deberías especificar p.o, seguido de libx.a y liby.a:

gcc p.o libx.a liby.a

C. En el escenario C, donde hay una dependencia circular entre p.o, libx.a y liby.a, necesitas especificar los archivos en el orden correcto para resolver las referencias correctamente. Una posible solución sería:

gcc p.o libx.a liby.a libx.a

Incluir libx.a dos veces en el comando (al final) es necesario para garantizar que todas las referencias se resuelvan adecuadamente en presencia de la dependencia circular.