Ejercicio 5.4

a. En la versión menos optimizada, el registro %xmm0 se usa como un valor temporal. Es decir, en cada iteración, el registro se carga con el producto almacenado en dest, se multiplica por el elemento actual del vector data[i], y el resultado se almacena en dest. En este caso, el registro se inicializa en 1.0 fuera del bucle y se multiplica por el elemento actual del vector data[i], y el resultado se almacena en dest. De esta manera, en cada iteración, el registro %xmm0 siempre almacena el producto acumulado.

b. Ambas versiones de la función combine3 tienen una funcionalidad idéntica, porque el realizado en la versión más optimizada es segura y no cambia el comportamiento del programa.

c. En la primera iteración, se escribe el valor de acc en dest para inicializarlo. Sin embargo, a partir de la segunda iteración, el valor almacenado en dest es el mismo que el valor almacenado en el registro %xmm0 al final de la iteración anterior. Por lo tanto, el valor leído desde dest al principio de la iteración será el mismo valor que se escribió en este registro al final de la iteración anterior. No se produce ninguna dependencia de datos entre el valor actual de dest y el valor almacenado en el registro %xmm0, lo que significa que no se pierde información.

Ejercicio 5.5

double poly(double a[], double x, long degree) {

long i;

double result = a[0];

double xpwr = x; /\* Equals x^i at start of loop \*/

for (i = 1; i <= degree; i++) {

result += a[i] \* xpwr;

xpwr = x \* xpwr;

}

return result;

}

a. Para determinar el número de adiciones y multiplicaciones realizadas por el código, podemos examinar el bucle que se ejecuta n veces. Dentro del bucle, se realizan una suma y una multiplicación en la línea 7 y una multiplicación en la línea 8 en cada iteración. Por lo tanto, el número total de adiciones es n, y el número total de multiplicaciones es 2n.

b. La operación de multiplicación requerida para calcular xpwr en cada iteración es la que limita el rendimiento de la función. La dependencia de datos creada por la operación de multiplicación significa que hasta que el cálculo de la iteración anterior no haya terminado el cálculo de una iteración no puede comenzar . Esto contribuye aún más al tiempo de ejecución total de la función. Por otro lado, la operación de suma requerida para actualizar el resultado de la función solo tarda 3 ciclos de reloj en la máquina de referencia. Como resultado, el costo de esta operación es menor en comparación con el costo de la operación de multiplicación.

Ejercicio 7.1

/\* m.c \*/

void swap();

int buf[2]={1,2};

int main()

{

swap();

return0

}

/\* swap.c \*/

extern int buf[];

int \*bufp0=&buf[0];

int \*bufp1;

void swap()

{

int temp;

bufp1=&buf[1];

temp=\*bufp0;

\*bufp0=\*bufp1

\*bufp1=temp;

}

Symbol .symtab entry? Symbol type Module where defined Section

buf sí externo m.o .data

bufp0 sí global swap.o .data

bufp1 sí global swap.o COMMON

swap sí global swap.o .text

temp no

Los símbolos buf, bufp0, bufp1 y swap son entradas .symbtab, mientras que temp no lo es ya que constituye una variable local. El símbolo buf es externo mientras que bufp0, bufp1 y swap son globales. A .data pertenecen las variables globales y estáticas inicializadas, por lo que buf y bufp0 (variables globales inicializadas) pertenecen a esta sección. El símbolo bufp1 pertenece a COMMON por ser una variable global no inicializada. La función swap() pertenece a la sección .text porque es una función que contiene código de máquina ejecutable: la sección .text es la sección del archivo ejecutable o de objeto que contiene el código del programa compilado en lenguaje de máquina.

Ejercicio 7.2

a. /\* Module 1 \*/

int main(){}

a) REF(main.1) → DEF(main.1)

b) REF(main.2) → DEF(main.1)

El main del módulo 1 es un símbolo fuerte ya que es una función, mientras que el main del módulo 2 es una variable sin inicializar y por tanto un símbolo débil. Entonces se escoge el símbolo fuerte por la regla 2.

b. /\* Module 1 \*/

void main(){}

a) REF(main.1) → DEF(ERROR)

b) REF(main.2) → DEF(ERROR)

Se usa la regla1 y resulta error ya que ambos módulos definen main como símbolo fuerte, en el módulo 1 como una función y en el 2 como una variable inicializada.

c. /\* Module 1 \*/

int x;

void main(){}

a) REF(x.1) → DEF(x.2)

a) REF(x.2) → DEF(x.2)

Se usa nuevamente la regla 2 ya que x es un símbolo débil en el módulo 1 y un símbolo fuerte en el módulo 2

/\* Module 2 \*/

int main;

int p2(){}

/\* Module 2 \*/

int main=1;

int p2(){}

/\* Module 2 \*/

double x = 1.0;

int p2(){}

Ejercicio 7.3

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

a. gcc p.o libx.a

En este caso el archivo de objeto p.o depende de la biblioteca estática libx.a. Hay que otorgar al enlazador el archivo de objeto p.o y la biblioteca estática libx.a para enlazar bien el programa.

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

El archivo de objeto p.o depende de la biblioteca estática libx.a, que a su vez depende de la biblioteca estática liby.a. Para enlazar adecuadamente el programa, se necesita el archivo de objeto p.o, la biblioteca estática libx.a y la biblioteca estática liby.a.

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

En este escenario, se presenta una dependencia circular entre la biblioteca estática libx.a y liby.a. Por lo tanto, hay que proporcionar el archivo de objeto p.o, la biblioteca estática libx.a, la biblioteca estática liby.a y la biblioteca estática libx.a para enlazar el programa.