Respuestas a los ejercicios 5.4, 5.5, 7.1, 7.2 y 7.3.

Estudiante: Ana Isabel Portal Díaz.

5.4

A. En ambos ciclos de código, el registro `%xmm0` se utiliza para realizar operaciones de punto flotante de 64 bits. Sin embargo, hay una diferencia clave en cómo se utiliza este registro en cada ciclo.

En el primer ciclo:

```
.L22:
vmulsd (%rdx), %xmm0, %xmm0
addq $8, %rdx
cmpq %rax, %rdx
vmovsd %xmm0, (%rbx)
jne .L22
```

El registro `%xmm0` se utiliza para almacenar el resultado de la multiplicación realizada en cada iteración del bucle. Cada vez que se ejecuta la instrucción `vmulsd`, el resultado se guarda en `%xmm0` y luego se mueve a la memoria en la dirección apuntada por `%rbx` mediante la instrucción `vmovsd`. Esto significa que en cada iteración del bucle se realizan dos movimientos de datos desde y hacia la memoria.

En el segundo ciclo:

```
.L17:
vmovsd (%rbx), %xmm0
vmulsd (%rdx), %xmm0, %xmm0
vmovsd %xmm0, (%rbx)
addq $8, %rdx
cmpq %rax, %rdx
jne .L17
```

El registro `%xmm0` se utiliza de manera más eficiente en este ciclo. Antes de realizar la multiplicación en cada iteración con la instrucción `vmulsd`, se mueve un valor de la

memoria en la dirección apuntada por `%rbx` a `%xmm0` utilizando la instrucción `vmovsd`. Esto significa que se utiliza `%xmm0` como una especie de acumulador en lugar de mover datos entre registros y memoria en cada iteración. Luego, el resultado se guarda nuevamente en la memoria.

En resumen, la diferencia clave en el papel del registro `%xmm0` entre ambos ciclos es que en el segundo ciclo se utiliza como un acumulador eficiente, evitando movimientos innecesarios entre registros y memoria en cada iteración. Esto puede llevar a un rendimiento mejorado en comparación con el primer ciclo, especialmente si el acceso a la memoria es más costoso en términos de tiempo y recursos.

B. Las dos versiones de combine3 tendrán una funcionalidad idéntica, incluso con *aliasing* de memoria.

El *aliasing* de memoria se refiere a cuando dos o más punteros apuntan a la misma ubicación de memoria. En el caso de combine3, si existe *aliasing* de memoria, significa que los registros %rbx y %rdx pueden apuntar a la misma ubicación de memoria en algún momento.

Las operaciones realizadas en ambos ciclos y el orden en que se realizan son equivalentes, aunque el segundo ciclo puede ser más eficiente debido al uso más optimizado del registro %xmm0, ambas versiones producirán el mismo resultado final.

C. La transformación del primer código al segundo se puede llevar a cabo sin alterar cómo funciona el programa. La razón es que, exceptuando la primera iteración, el valor leído desde la dirección de memoria apuntada por `%rbx` al inicio de cada iteración en el segundo código es el mismo valor que fue escrito en esa misma dirección al finalizar la iteración anterior.

En el primer código (`.L22`), se realiza la multiplicación en `%xmm0`, se guarda el resultado en la memoria en la dirección apuntada por `%rbx` y luego se compara y se salta a `.L22` si se cumple una condición. Esto significa que en cada iteración se lee el valor anteriormente escrito en la memoria antes de realizar la siguiente multiplicación.

En el segundo código (`.L17`), antes de realizar la multiplicación en `%xmm0`, se mueve el valor de la memoria en la dirección apuntada por `%rbx` a `%xmm0`. Luego, se guarda el resultado de la multiplicación nuevamente en la misma dirección de memoria antes de comparar y saltar a `.L17`. Esto implica que se utiliza el valor que ya se encuentra en el registro `%xmm0` al inicio de cada iteración para realizar la siguiente multiplicación.

Dado que ambas versiones del código realizan las mismas operaciones y utilizan el mismo registro `%xmm0` para almacenar el resultado de la multiplicación, la transformación del primer código al segundo puede llevarse a cabo sin alterar cómo funciona el programa. El valor almacenado en `%xmm0` se mantiene consistente a lo largo de las iteraciones, y el resultado final será el mismo en ambos códigos.

5.5

A. En el código proporcionado, en el cuerpo del ciclo *for* se realizan dos multiplicaciones y una suma, por tanto, si se realizan n iteraciones, el número total de multiplicaciones será 2n y el número total de sumas será n.

B. Podemos ver que el cálculo que limita el rendimiento aquí es el cálculo repetido de la expresión `xpwr = x * xpwr` , ya que cada vez que se ejecuta esta expresión se requiere una multiplicación de punto flotante, que generalmente lleva más tiempo de ejecución que otras operaciones aritméticas (según la tabla es de 5 ciclos de reloj) y el cálculo de una iteración no puede comenzar hasta que se haya completado el de la iteración anterior (esta naturaleza secuencial impide una ejecución paralela o concurrente de las iteraciones). La actualización del resultado solo requiere una adición de punto flotante (3 ciclos de reloj) entre iteraciones sucesivas.

7.1

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	-	-	-

Los símbolos *buf, bufp0, bufp1* y *swap* sí son entradas de la tabla de símbolos, por el contrario, *temp* no tiene entrada en la tabla de símbolos ya que es una variable local.

El símbolo *buf* está declarado como externo mientras que *bufp0, bufp1* y *swap* son globales.

La sección .data es una parte de la memoria de un programa donde se almacenan las variables globales y estáticas que tienen valores predefinidos. Estas variables se inicializan

antes de que el programa comience a ejecutarse. Debido a lo anterior explicado, *buf* y *bufp0* pertenecen a esta sección.

El símbolo *bufp1* pertenece a COMMON por ser una variable global no inicializada.

El símbolo *swap* pertenece a la sección .text debido a que es una función.

7.2

A. El enlazador escoge el símbolo fuerte definido en el módulo 1, sobre el símbolo débil definido en el módulo 2, siguiendo lo planteado en la regla 2 que establece que dado un símbolo fuerte y varios débiles con el mismo nombre, se elige el fuerte.

(a) REF(main.1) → DEF(main.1)

(b) REF(main.2) → DEF(main.1)

B. Es un Error, porque cada módulo define un símbolo fuerte principal y la regla 1 establece que múltiples símbolos fuertes con el mismo nombre no son permitidos.

C. El enlazador escoge el símbolo fuerte definido en el módulo 2, sobre el símbolo débil definido en el módulo 1, siguiendo lo establecido por la regla 2 que ya fue explicado en **A**.

(a) REF(x.1) → DEF(x . 2)

(b) REF(x.2) → DEF(x . 2)

7.3

A. En este escenario, tenemos al archivo de objeto p.o que depende de la biblioteca estática libx.a. La solución sería simplemente enlazar p.o con libx.a:

```
gcc p.o libx.a
```

B. En este escenario el archivo de objeto p.o depende de las bibliotecas estáticas libx.a y liby.a. La solución sería enlazar p.o, libx.a y liby.a en ese orden para resolver todas las referencias de símbolos, como a continuación:

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

C. En este escenario, tenemos que p.o depende de la biblioteca estática libx.a y esta depende de liby.a y a la vez liby.a depende de libx.a (una dependencia circular), debido a ello debemos indicarle al enlazador resolver las referencias circulares entre liby.a y libx.a en el siguiente orden específico:

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