Estructura de Computadores: Práctica 2 (Preguntas)

Patricia Córdoba Hidalgo

Cuestiones sobre popcount.c

1.

En el peor de los casos todos los elementos tienen todos sus bits activados, es decir, cada elemento está formado por 32 unos, ya que nuestro array está formado por `unsigned int` de 32 bits. Por lo tanto, el número total de unos es 32*N, siendo N el nº de elementos de la lista. Como un `int` puede almacenar 32767 bits (2^15-1), N puede ser a lo sumo la parte entera de (2^15 - 1)/32, es decir N puede llegar a valer 1023 sin desbordamiento.

En el caso de un `unsigned int`, que puede almacenar 65535 bits, que equivale a 2^16-1, N puede ser, a lo sumo, la parte entera de (2^16-1)/32, es decir, N puede llegar a valer 2047 sin que se produzca overflow.

2.

Para calcular el peso de Hamming (número de unos) de una lista de números sucesivos, del 0 a `SIZE-1`, siendo `SIZE` una potencia de 2, usamos el siguiente argumento:

Si tenemos una lista desde el cero a una potencia de 2 menos un elemento, podemos observar que cada bit esta activo sólo la mitad de las veces. Por ejemplo, en una lista de 8 elementos, obtenemos:

```
000
001
010
011
100
101
110
111
```

Como podemos observar, cada bit está activo la mitad de las veces, es decir, 4, por lo que el número de bits activos es (8*3)/2 = 4*3 =12.

Si SIZE=2^k, la fórmula que obtenemos es: **(SIZE*k)/2**.

3.

En el desplazamiento de n bits de números negativos (en complemento a 2) hacia la derecha, todos los bits se mueven n bits hacia la derecha y se van insertando por la parte más significativa del número el bit de signo, en caso de números negativos, un 1. Ejemplo, para desplazar 1010, al ser negativo, queda

1101, lo cual introduce un bit activo más. Debido a esto, al desplazar todos los bits de dicho número, te quedarían tantos unos como tamaño tenga el elemento, y eso corresponde a N bits activos, luego el número nunca es cero. Es por esto que declaramos la lista como unsigned, para que al desplazar a la derecha, se introduzcan 0s en la parte izquierda.

4.

Cambiando “+r” por “+m”, el tiempo medio aumenta considerablemente, siendo los resultados:

```
Media con "+r":1802.6 us
Media con "+m":5432,4 us
```

Esto se debe a que con “+r” los datos se guardan en un registro del procesador y con “+m”, se guardan en memoria, luego el procesadir tiene que pedir los datos a la memoria y el proceso de solicitud y envío de datos lleva más tiempo.

Cambiando “r” por “m”, no compila el programa, dando el error: `"no se dio un sufijo mnemónico de instrucción y ningún operando de registro; no se puede determinar el tamaño de la instrucción"` Se debe a que no puede determinar el tamaño de x en memoria. Esto se arregla cambiando la instrucción `shr` por `shrl`.

```
Media con "m":5226,1 us
Media con "+m" y "m":6165,7 us
```

5.

Porque los elementos guardados en memoria son arrays y no caben en un registro.

6.

La operación `pshufb x1, x2` coge un par de bits de x1, digamos que está en la posición j-ésima y j+1-ésima, calcula el valor de este par, llamemoslo i, y mira el valor almacenado en las posiciones 2*i-ésima y 2*i+1-ésima del segundo elemento. Después, coloca en la posición j de x2 el valor almacenado en la posición i de este. Como esta operación se hace en paralelo con todas las parejas de bits de x1, no hay problema con que se cojan datos de x1 y no de x2.

7.

Usando el comando `gcc -O0 -S popcount.c`, `gcc -O1 -S popcount.c` y `gcc -O2 -S popcount.c` vemos los códigos ensamblador generados, y observamos que son muy parecidos, por lo que la mejora de tiempo es muy sutil. Sin embargo, mi versión 3 es mejor que mi versión 2.

8.

El mejor resultado se obtiene con la versión 5, creada a partir de código ASM.

Cuestiones sobre parity.c

1.

Dados los 16 primeros elementos, veamos sus paridades:

```
Número:   Paridad:
0000          0
0001          1
0010          1
0011          0
0100          1
0101          0
0110          0
0111          1
1000          1
1001          0
1010          0
1011          1
1100          0
1101          1
1110          1
1111          0
```

Dado una lista de tamaño `SIZE`, donde `SIZE=2^k` para un cierto k perteneciente a los naturales, y una lista de números consecutivos de 0 a `SIZE-1`, podemos ver que se repite la secuencia 0110-1001, donde la mitad de los números tienen un nº par de unos, y la otra mitad un número impar de unos.

Luego el número de elementos con un número impar de unos es **SIZE/2**, siendo SIZE el tamaño de la lista.

2.

En el desplazamiento de n bits de números negativos (en complemento a 2) hacia la derecha, todos los bits se mueven n bits hacia la derecha y se van insertando por la parte más significativa del número el bit de signo, en caso de números negativos, un 1. Ejemplo, para desplazar 1010, al ser negativo, queda 1101, lo cual introduce un bit activo más. Debido a esto, al desplazar todos los bits de dicho número, te quedarían tantos unos como tamaño tenga el elemento, y eso corresponde a N bits activos, luego el número nunca es cero. Es por esto que declaramos la lista como unsigned, para que al desplazar a la derecha, se introduzcan 0s en la parte izquierda.

3.

Hay una pequeña mejora en el tiempo para todos los niveles de optimización de la versión 3 respecto de la versión 2 (mirar gráficas).

En la cuarta versión, con las restricciones a memoria, la media de los tiempos son:

```
Media con "+r": 2424
Media con "+m": 3454.2
```

Esto se debe a que con “+r” los datos se guardan en un registro del procesador y con “+m”, se guardan en memoria, luego el procesadir tiene que pedir los datos a la memoria y el proceso de solicitud y envío de datos lleva más tiempo.

Cambiando “r” por “m”, no compila el programa, dando el error: `"no se dio un sufijo mnemónico de instrucción y ningún operando de registro; no se puede determinar el tamaño de la instrucción"` Se debe a que no puede determinar el tamaño de x en memoria. Esto se arregla cambiando la instrucción `shr` por `shrl`.

```
Media con "m": 5005.6
```

Al cambiar ambos registros por “+m” y “m”, da el siguiente error: `demasiadas referencias a memoria para xor"`.Este error no puede corregirse.

4.

Esta versión tiene una media de tiempo mejor que la versión 4 con las optimizaciones `-O0` y `-O1`. Además hay una ligera mejora al usar la segunda optimización. Sin embargo, con la optimización `-O2` la media de tiempo aumenta considerablemente y mayor, no solo que sus anateriores optimizaciones, sino también es superior a la versión 2 y 3.

5.

En la cuarta versión, con las restricciones a memoria, la media de los tiempos son:

```
Media con "+r": 2424
Media con "+m": 3454.2
```

Esto se debe a que con “+r” los datos se guardan en un registro del procesador y con “+m”, se guardan en memoria, luego el procesadir tiene que pedir los datos a la memoria y el proceso de solicitud y envío de datos lleva más tiempo.

Cambiando “r” por “m”, no compila el programa, dando el error: `"no se dio un sufijo mnemónico de instrucción y ningún operando de registro; no se puede determinar el tamaño de la instrucción"` Se debe a que no puede determinar el tamaño de x en memoria. Esto se arregla cambiando la instrucción `shr` por `shrl`.

```
Media con "m": 5005.6
```

Al cambiar ambos registros por “+m” y “m”, da el siguiente error: `demasiadas referencias a memoria para xor"`.Este error no puede corregirse.

6.

Esto funciona ya que, al hacer la operación `XOR` bit a bit entre dos datos, si tengo dos 1s, o dos 0s, guarda un 0, es decir, el número de 1s es par. Sin embargo, si tengo un sólo 1s, se guarda un 1, es decir, el número de 1s es impar. Como podemos observar, la paridad final del número se conserva.

En cada iteración reducimos el número a la mitad y hacemos `XOR` con el número antes de reducirse, con esto obtenemos un número que conserva la paridad del original. El proceso se repite hasta que sólo queda un bit, o un 0, o un 1.

7.

En la versión 6, reducimos el dato sin usar el último bucle for, ya que de este modo nos ahorramos las 4 últimas iteraciones de dicho bucle, a pesar de que quede menos elegante y haya que hacer dos desplazamientos “a mano”. Esto se debe a que, una vez que tenemos el dato en un registro de 8 bits, `%dl`, podemos usar la orden `setpo`, que calcula su paridad.

8.

En los tres niveles de compilación se obtienen resultados muy similares cuando se usa clobber y cuando no.

9.

El mejor resultado se obtiene con la versión 6, creada a partir de código ASM.