PRÁCTICA 3

Jose Miguel Hernández García
2º C

En la primera semana de esta práctica (la última semana de octubre) he tratado de entender los diversos ejemplos que aparecía en el tutorial de esta mediante el uso de herramientas como gcc para compilarlos, ddd para depurarlos y comprobar su funcionamiento y además tratando de entender su código en ensamblador, pues se trataban de ejemplos escritos en C y en Ensamblador y al crear el .o (archivo objeto) he podido ver el código completo en ensamblador, donde he ido comprobando la función de cada instrucción para así comprenderlos.

En esta segunda semana de la práctica he empezado con el programa pop_count realizando las tres primeras versiones de este, tratándose respectivamente de un doble ciclo for, un ciclo for-while y un for donde implementamos algunas instrucciones en ensamblador. También he tratado de resolver mis dudas a cerca de las dos versiones posteriores a esta gracias a mis compañeros.

Esta tercera semana he terminado el ejercicio pop_count comprobando si ha funcionado bien y si los resultados obtenidos han sido los correctos, posteriormente he empezado el siguiente ejercicio, parity.

En la cuarta semana de la práctica he continuado trabajando sobre el ejercicio parity, tratando de entenderlo y continuando con lo que había empezado la anterior semana.

Finalmente, la última semana he terminado el parity, y he hecho las tablas requeridas para cada grado de optimización aplicado tanto en el ejercicio pop_count como en parity, además de terminar de poner a punto ambos ejercicios.

A continuación incluyo los códigos de cada uno de los ejercicios:

4.1 Calcular la suma de bits de una lista de enteros sin signo

```
#include <stdio.h>                  // para printf()
#include <stdlib.h>                 // para exit()
#include <sys/time.h>               // para gettimeofday(), struct timeval

#define SIZE (1<<16)                // tamaño suficiente para tiempo apreciable
// unsigned lista[SIZE]={0x80000000, 0x00100000, 0x00000800, 0x00000001};
// unsigned lista[SIZE]={0x7fffffff, 0xffefffff, 0xfffff7ff, 0xfffffffe, 0x01000024, 0x00356700, 0x8900ac00,
0x00bd00ef};
// unsigned lista[SIZE]={0x0          , 0x10204080, 0x3590ac06, 0x70b0d0e0, 0xffffffff 0x12345678,
0x9abcdef0, 0xcafebeef};

unsigned lista[SIZE];
#define WSIZE (8*sizeof(int))
```

```
// Primera versión del pop_count, donde recorremos el array con un bucle for
// y recorremos los bits con otro bucle for
unsigned pop_count1(int* array, int len){

   int i;
   int j;
   unsigned result = 0;
   unsigned x;

   for(i = 0; i < len; i++) {                    // Para todos los números del vector:
      x = array[i];                              // Extraemos un número

      for(j = 0; j < WSIZE; j++){     // Recorremos todos los bits del número.
         unsigned mascara = 1 << j;           // Extraemos los bits con una máscara 0x1
          result += (x & mascara) != 0;       // Vamos acumulando el resultado
      }
   }
   return result;
}

// Segunda versión del pop_count, donde recorremos el array con un bucle for
// y recorremos los bits con un bucle while
unsigned pop_count2(int* array, int len){

   int i;
   unsigned x;
   unsigned result = 0;

   for(i = 0; i < len; i++) {              // Recorremos el vector
      x = array[i];                        // Extraemos un número

      while (x) {                          // Recorremos todos los bits de dicho número
         result += x&0x1;    // Extraemos los bits con una máscara 0x1 y
                                           // los acumulamos en el contador "resultado"
         x >>= 1;                          // Realizamos el desplazamiento de bits a la dcha
       }
   }
   return result;
}

// Tercera versión del pop_count, donde recorremos el array con un bucle for
// y para recorrer los bits no utilizaremos un ciclo while, sino recurriremos
// al ensamblador, aplicando la orden adc
unsigned pop_count3(unsigned *array, int len){

   int i;
   unsigned result = 0;
   unsigned x;
```

```
    for(i = 0; i < len; i++){
        x = array[i];
       asm(                              "\n"
            "ini3:                       \n\t"          // Seguiremos mientras num != 0
            "shr  %[x]                   \n\t"          // LSB en CF
            "adc  $0,   %[result]        \n\t"          // Acumulamos el acarreo
            "test %[x], %[x]             \n\t"          // Comprobamos si num != 0
            "jnz  ini3                   \n\t"          // Hacemos el salto si quedan bits a 1
            : [result]"+r" (result)                     // E/S: añadir lo acumulado por el momento
            : [x] "r" (x)                               // entrada: valor elemento
            );
    }
    return result;
}

// Cuarta versión del pop_count, donde recurrimos a un bucle for anidado,
// y realizamos una suma en árbol
unsigned pop_count4(unsigned *array, int len){

    int i;
    int j;
    unsigned val = 0;
    unsigned result = 0;
    unsigned x;

    for(i = 0; i < len; i++){               // Recorremos el vector
        x = array[i];                       // Escogemos un número del vector

        for(j = 0; j < 8; j++){             // Recorremos los bytes
            val += x & 0x01010101;          // Acumulamos los bits de cada byte recorrido
            x >> 1;                         // Realizamos el desplazamiento a la dcha
         }

         val += (val >> 16);                // Volvemos a acumular en "valor" mientras desplazamos
         val += (val >> 8);                 // Volvemos a acumular en "valor" mientras desplazamos

         result += (val & 0xFF);           // Acumulamos todo en "resultado"
         val = 0;                           // Volvemos a poner a cero la variable valor
                                            // y volvemos a empezar el bucle
    }
    return result;
}

// Quinta versión del pop_count, implementando la instrucción SSS3
unsigned pop_count5 (unsigned *array, int len){

    int i;
    unsigned val;
    unsigned result = 0;
    int SSE_mascara[] = {0x0f0f0f0f, 0x0f0f0f0f, 0x0f0f0f0f, 0x0f0f0f0f};
    int SSE_LUTb[] = {0x02010100, 0x03020201, 0x03020201, 0x04030302};
```

```
    if (len & 0x3)
        printf("leyendo 128b pero len no multiplo de 4?\n");

    for (i = 0; i < len; i += 4){
       asm(
        "movdqu  %[x],         %%xmm0  \n\t"
        "Movdqa  %%xmm0, %%xmm1  \n\t"      // Creamos dos copias de x
        "movdqu  %[m],         %%xmm6  \n\t"      // Creamos la máscara
        "psrlw       $4,            %%xmm1 \n\t"
        "pand        %%xmm6, %%xmm0 \n\t"      // xmm0 - nibbles inferiores
        "pand        %%xmm6, %%xmm1 \n\t"      // xmm1 - nibbles superiores

        "movdqu   %[l],          %%xmm2 \n\t"      // ...como pshufb sobreescribe LUT
        "movdqa   %%xmm2, %%xmm3 \n\t"      // ...queremos 2 copias
        "pshufb     %%xmm0, %%xmm2 \n\t"      // xmm2 = vector popcount inferiores
        "pshufb     %%xmm1, %%xmm3 \n\t"      // xmm3 = vector popcount superiores

        "paddb      %%xmm2, %%xmm3 \n\t"      // xmm2 += xmm3 - vector popcount bytes
        "pxor        %%xmm0, %%xmm0 \n\t"      // xmm0 = 0,0,0,0
        "psadbw   %%xmm0, %%xmm3  \n\t"      // xmm3 = [pcnt bytes0..7|pcnt bytes8..15]
        "movhlps  %%xmm3, %%xmm0  \n\t"      // xmm0 = [        0                       |pcnt bytes0..7]
        "paddd      %%xmm3, %%xmm0 \n\t"      // xmm0 = [        no usado  |pcnt bytes0..15]
        "movd       %%xmm0, %[val]\n\t"
        : [val]"=r" (val)
        : [x]  "m"  (array[i]),
        [m]  "m"   (SSE_mascara[0]),
        [l]  "m"     (SSE_LUTb[0])
        );
        result += val;
    }
    return result;
}

// Funcion cronometro para calcular los tiempos
void cronometro(unsigned (*func)(), char* msg){

    struct timeval tv1,tv2;      // gettimeofday() secs-usecs
    long tv_usecs;               // y sus cuentas

    gettimeofday(&tv1, NULL);
    unsigned resultado = func(lista, SIZE);
    gettimeofday(&tv2, NULL);

    tv_usecs = (tv2.tv_sec - tv1.tv_sec )*1E6 + (tv2.tv_usec - tv1.tv_usec);

    printf("resultado = %d\t", resultado);
    printf("%s:%9ld us\n", msg, tv_usecs);
}
```

```
int main(){

   int i;

   for (int i = 0; i < SIZE; i++)
      lista[i] = i;

   cronometro(pop_count1, "pop_count1, con doble ciclo for");
   cronometro(pop_count2, "pop_count2, con ciclo for y while");
   cronometro(pop_count3, "pop_count3, con ciclo for y asm");
   cronometro(pop_count4, "pop_count4, ejercicio del libro, suma en árbol");
   cronometro(pop_count5, "pop_count5, aplicando la instrucción SSS3");

   exit(0);
}
```

4.2 Calcular la suma de paridades de una lista de enteros sin signo

```
#include <stdio.h>              // para printf()
#include <stdlib.h>             // para exit()
#include <sys/time.h>           // para gettimeofday(), struct timeval

#define SIZE (1<<16)        // tamaño suficiente para tiempo apreciable
// unsigned lista[SIZE]={0x80000000, 0x00100000, 0x00000800, 0x00000001};
// unsigned lista[SIZE]={0x7fffffff, 0xffefffff, 0xfffff7ff, 0xfffffffe, 0x01000024, 0x00356700, 0x8900ac00,
0x00bd00ef};
// unsigned lista[SIZE]={0x0              , 0x10204080, 0x3590ac06, 0x70b0d0e0, 0xffffffff 0x12345678,
0x9abcdef0, 0xcafebeef};

unsigned lista[SIZE];
#define WSIZE (8*sizeof(int))

// Primera versión de parity, donde recorremos el array con un bucle for,
// y posteriormente recorremos los bits con un segundo bucle for
unsigned parity_1(unsigned *array, int len){

   unsigned i;
   int j;
   unsigned result = 0;
   unsigned val = 0;
   unsigned x;

   for(i = 0; i < len; i++){                  // Recorremos el vector
      x = array[i];                           // Seleccionamos un número

      for(j = 0; j < WSIZE; j++){             // Recorremos los bits
         unsigned mascara = 0x1 << j; // Aplicamos la máscara y el desplazamiento
         val ^= (x & mascara) != 0;       // Acumulamos en val los bits con XOR
```

```
        }

        result += val;                  // Acumulamos en result lo que tenemos en val
        val = 0;                        // Ponemos val a 0
    }
    return result;
}

// Segunda versión de parity, donde recorremos el array con un bucle for,
// y tras ello recorreremos los bits mediante un ciclo while
unsigned parity_2(unsigned *array, int len){

    int i;
    unsigned result = 0;
    unsigned val = 0;
    unsigned x;

    for(i = 0; i < len; i++){ // Recorremos el vector
        x = array[i];          // Seleccionamos un número del vector

        while(x >>= 1){       // Recorremos los bits mediante el ciclo while
            val ^= x&0x1;     // Acumulamos dichos bits en la variable val con XOR
                              // Hemos utilizado aquí la máscara 0x1 y el desplazamiento
        }

        result += val;        // Acumulamos en result lo que tenemos en val
        val = 0;              // Ponemos val a 0
    }
    return result;
}

// Tercera versión del parity, implementada a partir del ejercicio 3.22 del libro
// de clase, adaptada para un array completo. En esta versión acumulamos la mascara
// en el resultado final
unsigned parity_3(unsigned *array, int len){

    int i;
    unsigned result = 0;
    unsigned val = 0;
    unsigned x;

    for(i = 0; i < len; i++){      // Recorremos el vector
        x = array[i];              // Seleccionamos un número del vector

        while(x){
            val ^= x;              // Aplicamos XOR
            x >>= 0x1;             // Aplicamos el desplazamiento y la máscara
        }

        result += (val & 0x1);  // Acumulamos en result lo anterior
        val = 0;                // Ponemos val a 0
```

```
    }
    return result;
}

// Cuarta versión del parity, donde sustituimos el ciclo while utilizado anteriormente
// implementando en su lugar varias instrucciones de ensamblador
unsigned parity_4(unsigned *array, int len){

    int i;
    unsigned result = 0;
    unsigned val = 0;
    unsigned x;

    for(i = 0; i < len; i++){       // Recorremos el vector
        x = array[i];               // Escogemos un número del vector
        val = 0;                    // Ponemos la variable val a 0

        asm("\n"
        "ini3:                  \n\t"    // Seguir mientras que x!=0
        "xor  %[x], %[val] \n\t"    // Realmente solo nos interesa LSB
        "shr  %[x]             \n\t"    // LSB en CF
        "test %[x], %[x]    \n\t"    // Hacemos una comparación si x != x
        "jnz  ini3              \n\t"    // Saltamos a ini3 si quedan bits a 1
        "and  $1,   %[val] \n\t"
        : [val] "+r" (val)              // E/S: entrada 0, salida paridad elemento
        : [x] "r" (x)                   // Entrada: valor elemento
        );

        result += val;                // Acumulamos en result lo que teníamos en val
    }
    return result;
}

// Quinta versión del parity donde recurrimos a la suma en arbol, al igual
// que se hizo en el ejercicio pop_count4, utilizando XOR en el array y haciendo
// desplazamientos sucesivos a mitad de distancia hasta finalizar en x ^= x >> 1
unsigned parity_5(unsigned *array, int len){

    int i, j;
    unsigned x;
    unsigned result = 0;

    for (i = 0; i < len; i++){            // Recorremos el vector
        x = array[i];                       // Seleccionamos un número del vector

        for (j = 16; j >= 1; j = j / 2)
            x ^= x >> j;                    // Aplicamos XOR y desplazamiento

        result += (x&0x1);              // Acumulamos en result lo anterior
    }
    return result;
```

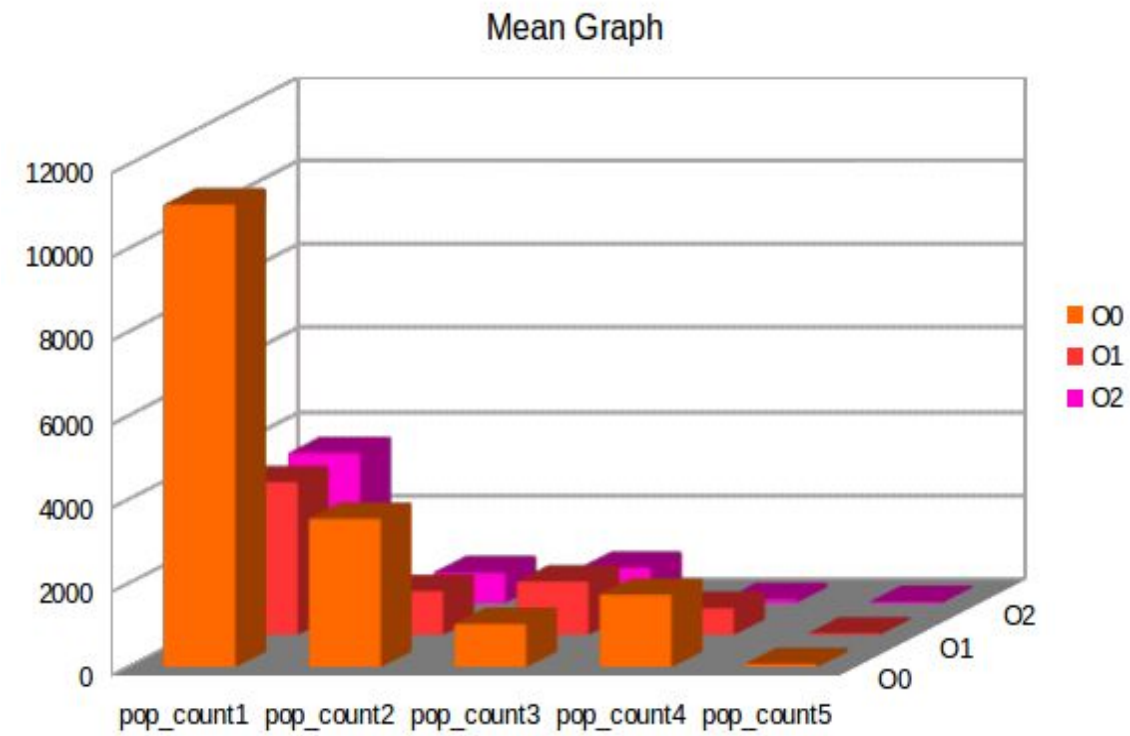
```
}

// Sexta versión del parity, sustituyendo de nuevo, el ciclo for interno
// por instrucciones en ensamblador, incluyendo XOR y SHR, ademas de SETcc y
// MOVZx
unsigned parity_6(unsigned *array, int len){

   int i;
   unsigned result = 0;
   unsigned x;

   for (i = 0; i < len; i++){
      x = array[i];

      asm(
                                    "\n"
      "mov     %[x],    %%edx  \n\t"   // Sacar copia para XOR. Controlar el registro edx
      "shr     $16,     %[x]       \n\t"
      "xor     %[x],    %%edx  \n\t"  // Aplicamos XOR
      "xor     %%dh, %%dl    \n\t"   // Aplicamos XOR
      "setpo   %%dl                \n\t"
      "movzx %%dl,  %[x]       \n\t"   // devolvemos en 32b
      : [x]"+r" (x)                          //e/s: entrada valor elemento, salida paridad
      :
      :"edx"                                 //clobber
      );

      result += x;
   }
   return result;
}

// Funcion cronometro para calcular los tiempos
void cronometro(unsigned (*func)(), char* msg){

         unsigned resultado;
   struct timeval tv1,tv2;              // gettimeofday() secs-usecs
   long tv_usecs;                       // y sus cuentas

   gettimeofday(&tv1, NULL);
   resultado = func(lista, SIZE);
   gettimeofday(&tv2, NULL);

   tv_usecs = (tv2.tv_sec - tv1.tv_sec )*1E6 + (tv2.tv_usec - tv1.tv_usec);

                  printf("resultado = %d\t", resultado);
                  printf("%s:%9ld us\n", msg, tv_usecs);
}

int main(){
```

```
    int i;

    for(i = 0; i < SIZE; i++)
        lista[i] = i;

    cronometro(parity_1, "parity_1, con doble ciclo for");
    cronometro(parity_2, "parity_2, con ciclo for y while");
    cronometro(parity_3, "parity_3, con ciclo for y while (libro)");
    cronometro(parity_4, "parity_4, con ciclo for y asm");
    cronometro(parity_5, "parity_5, con doble ciclo for y suma en árbol");
    cronometro(parity_6, "parity_6, con suma en árbol en asm");

    printf("N * (N + 1) / 2 = %d\n", (SIZE - 1) * (SIZE / 2));  /*OF*/

    exit(0);
}
```

A continuación incluyo las gráficas en las que se comparan los tiempos de cada una de las versiones de los dos programas que se han realizado:

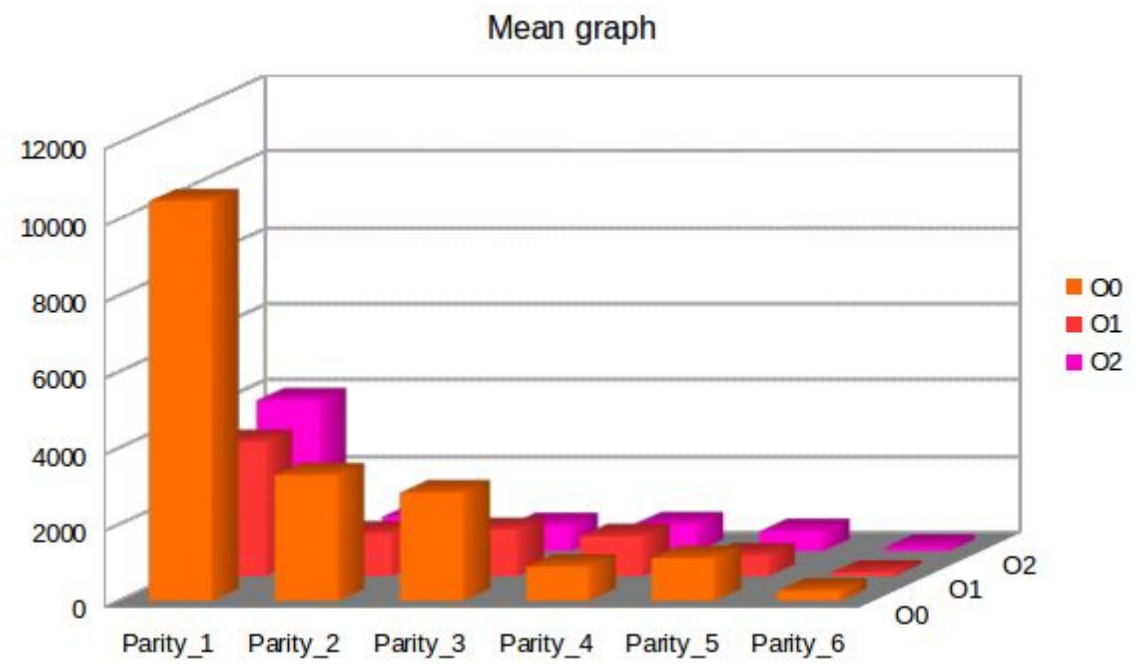
En primer lugar, la gráfica del primer programa:

4.1 Calcular la suma de bits de una lista de enteros sin signo



Donde vemos que la versión pop_count1 es la que más tiempo tarda en completarse, en los 3 niveles de optimización estudiados.

4.2 Calcular la suma de paridades de una lista de enteros sin signo



En esta gráfica es posible apreciar también como la primera versión del Parity es la más lenta de las 6 estudiadas.