Práctica 3. Estructura de Computadores
Noelia Escalera Mejías
Grupo A3

En esta práctica se nos ha pedido desarrollar diferentes versiones del programa **popcount**, el cuál consiste en, dada una lista de enteros, devolver el número de bits puestos a 1 en ellos. Hemos desarrollado un total de 10 versiones del programa, con el objetivo de averiguar cuál de ellas es más eficiente y si en alguna de ellas podemos ganar en eficiencia al compilador gcc.

Primera versión

Nuestra primera versión consiste en recorrer un array de enteros de 4 bits con dos bucles for, uno para recorrer el propio array, y otro para recorrer cada entero.

```
int popcount1(unsigned* array, size_t len)
{
        size_t i, j;
        int result = 0;
        unsigned x;

        for (i = 0; i < len; i++){ // Recorrer el vector
                x = array[i];
                for (j = 0; j < sizeof(int) * 8; j++){ // Recorremos cada entero del array por sus bits
                        unsigned bit = (x >> j) & 0x1; // Desplazamos los bits necesarios a la derecha y
                                                       // aplicamos la máscara
                        result += bit; // Añadimos el bit al resultado
                }
        }

        return result;
}
```

Segunda versión

La segunda versión es prácticamente idéntica a la primera, solo que el bucle del interior será un while en vez de un for. Esta versión es más eficiente, ya que nos saldremos el bucle while en cuanto el entero que estamos recorriendo sea 0 debido a los desplazamientos de bits.

```
int popcount2(unsigned* array, size_t len)
{
        size_t i;
        int result = 0;
  unsigned x;

        for (i = 0; i < len; i++){ // Recorremos el vector
    x = array[i];
                while (x){    // Recorremos cada entero del array por sus bits, nos salimos del bucle cuando sea 0
                        result += x & 0x1; // Añadimos al resultado el entero con la máscara aplicada
                        x >>= 1; // Desplazamos un bit a la derecha
                }
        }

        return result;
}
```

Tercera versión

Ahora vamos a sustituir el código del bucle por asm inline. Básicamente programaremos el bucle while, pero con instrucciones en ensamblador.

```
int popcount3(unsigned* array, size_t len)
{
	int result = 0;
	unsigned x;
	size_t i;

	for (i = 0; i < len; i++){ // Recorremos el vector
		x = array[i];
		asm("\n"
	"ini3:                                  \n\t"
			"shr %[x]           \n\t"     // Desplaza un bit a la derecha
			"adc $0, %[r] \n\t"       // Sumamos el último bit a result
			"test %[x], %[x] \n\t"
			"jnz ini3           \n\t" // Si x no es 0, salta a ini3

			: [r]"+r" (result)
			: [x] "r" (x)                   );
	}
	return result;
}
```

Cuarta versión

En la cuarta versión vamos a prescindir de la operación test, aprovecharemos que la propia operación shr también modifica los flags que necesitamos.

```
int popcount4(unsigned* array, size_t len)
{
	int result = 0;
	unsigned x;
	size_t i;

	for (i = 0; i < len; i++){
		x = array[i];
		asm("\n"
			"clc                    \n\t" // Limpiamos el flag de acarreo
	"ini4:                                  \n\t"
			"adc $0, %[r] \n\t" // Sumamos el último bit a result
			"shr %[x]           \n\t" // Desplaza un bit a la derecha
			"jnz ini4\n\t" // Si x no es cero, vuelve a ini4
	"fin4:                                  \n\t"
			"adc $0, %[r]     \n\t" // Si x es cero, añade el último bit a result

			: [r] "+r" (result)
			: [x] "r" (x)                   );

	}
	return result;
}
```

Noelia Escalera Mejías
Grupo A3

Quinta versión

En esta versión, adaptaremos la solución que aparece en el libro CS:APPa nuestro array de enteros de 32 bits. Esta versión vuelve a estar íntegramente en lenguaje C. Consiste en aplicar 8 veces la máscara 0x0101… a cada elemento e ir acumulando los bits en una variable local, para luego sumar los bytes en árbol.

```
int popcount5(unsigned* array, size_t len)
{
	int result = 0, val = 0;
	size_t i, j;
	unsigned x;

	for (i = 0; i < len; i++){ // Recorremos el vector
		x = array[i];
		val = 0; // Variable local para acumular los bits
		for (j = 0; j < 8; j++){ // Recorremos cada entero
			val += x & 0x01010101; // Aplicamos la máscara (para 32 bits)
			x >>= 1; // Desplazamos un bit a la derecha
		}
		val += (val >> 16); // Sumamos los bits
		val += (val >> 8);
		result += val & 0xFF;
	}
	return result;
}
```

Sexta versión

Ahora adaptaremos una solución propuesta por Wikipedia otra vez a un array de enteros de 32 bits. Lo que propone esta versión es sustituir el bucle for interior por más sumas en árbol.

```
int popcount6(unsigned* array, size_t len)
{
	const unsigned m1 = 0x55555555;
	const unsigned m2 = 0x33333333;
	const unsigned m4 = 0x0f0f0f0f;
	const unsigned m8 = 0x00ff00ff;
	const unsigned m16 = 0x0000ffff;

	int result = 0;
	size_t i;
	unsigned x;

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

		x = (x & m1) + ((x >> 1) & m1);  // Sumamos en árbol los bits
		x = (x & m2) + ((x >> 2) & m2);
		x = (x & m4) + ((x >> 4) & m4);
		x = (x & m8) + ((x >> 8) & m8);
		x = (x & m16) + ((x >> 16) & m16);

		result += x;
	}
	return result;
}
```

Noelia Escalera Mejías
Grupo A3

Séptima versión

Nuestra siguiente versión es prácticamente igual que la anterior, pero extendida para tamaños más grandes.

```
int popcount7(unsigned* array, size_t len)
{
	size_t i;
	unsigned long x1, x2;
	int result = 0;

	const unsigned long m1 = 0x5555555555555555;
	const unsigned long m2 = 0x3333333333333333;
	const unsigned long m4 = 0x0f0f0f0f0f0f0f0f;
	const unsigned long m8 = 0x00ff00ff00ff00ff;
	const unsigned long m16 = 0x0000ffff0000ffff;
	const unsigned long m32 = 0x00000000ffffffff;

	if (len & 0x3) printf("leyendo128b pero len no múltiplo de 4\n");

	for (i = 0; i < len; i +=4) // Recorremos el vector
	{
		x1 = *(unsigned long*) &array[i];
		x2 = *(unsigned long*) &array[i+2];

		x1 = (x1 & m1) + ((x1 >> 1) & m1); // Sumas en árbol
		x1 = (x1 & m2) + ((x1 >> 2) & m2);
		x1 = (x1 & m4) + ((x1 >> 4) & m4);
		x1 = (x1 & m8) + ((x1 >> 8) & m8);
		x1 = (x1 & m16) + ((x1 >> 16) & m16);
		x1 = (x1 & m32) + ((x1 >> 32) & m32);

		x2 = (x2 & m1) + ((x2 >> 1) & m1);
		x2 = (x2 & m2) + ((x2 >> 2) & m2);
		x2 = (x2 & m4) + ((x2 >> 4) & m4);
		x2 = (x2 & m8) + ((x2 >> 8) & m8);
		x2 = (x2 & m16) + ((x2 >> 16) & m16);
		x2 = (x2 & m32) + ((x2 >> 32) & m32);

		result += x1+x2;
	}
	return result;
}
```

Noelia Escalera Mejías
Grupo A3

Octava versión

```
int popcount8(unsigned *array, size_t len){
        size_t i;
 int val, result=0;
 int SSE_mask[] = {0x0f0f0f0f, 0x0f0f0f0f, 0x0f0f0f0f, 0x0f0f0f0f};
 int SSE_LUTb[] = {0x02010100, 0x03020201, 0x03020201, 0x04030302};
 //                    3 2 1 0    7 6 5 4   11 10 9 8   15 14 13 12

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

 for (i=0; i<len; i+=4){
        asm("movdqu %[x], %%xmm0   \n\t"
        "movdqa %%xmm0, %%xmm1              \n\t"  //  x: two copies xmm0-1
        "movdqu          %[m], %%xmm6                 \n\t"  // mask: xmm6
        "psrlw $4 , %%xmm1                \n\t"
        "pand %%xmm6, %%xmm0                \n\t"  //; xmm0 – lower nibbles
        "pand %%xmm6, %%xmm1                \n\t"  //; xmm1 – higher nibbles

        "movdqu %[l], %%xmm2                \n\t"  //; since instruction pshufb modifies LUT
        "movdqa %%xmm2, %%xmm3              \n\t"  //; we need 2 copies
        "pshufb %%xmm0, %%xmm2              \n\t"  //; xmm2 = vector of popcount lower nibbles
        "pshufb %%xmm1, %%xmm3              \n\t"  //; xmm3 = vector of popcount upper nibbles

        "paddb %%xmm2, %%xmm3               \n\t"  //; xmm3 - vector of popcount for 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 = [  not needed  |pcnt bytes0..15]
        "movd %%xmm0, %[val]"

        : [val]"=r" (val)
        : [x] "m" (array[i]),
        [m] "m" (SSE_mask[0]),
        [l] "m" (SSE_LUTb[0])
        );

        result += val;
 }

 return result;

}
```

Novena versión

Esta versión consiste en sustituir el bucle interno while por la instrucción popcnt en asm.

```
int popcount9(unsigned* array, size_t len)
{
	size_t i;
	unsigned x;
	int val, result = 0;

	for (i = 0; i < len; i++) // Recorremos el vector
	{
		x = array[i];
		asm("popcnt %[x],%[val]" // Guardamos el popcount de x en val

			:[val] "=r" (val)
			:  [x]  "r" (x)
		);
		result += val;
	}
	return result;
}
```

Décima versión

Por último, la décima versión es prácticamente igual a la anterior, pero realizando dos lecturas de 64 bits y dos popcount

```
int popcount10(unsigned* array, size_t len)
{
	 size_t i;
	 unsigned long x1, x2;
	 long val = 0;
	 int result = 0;

	 if (len & 0x3) printf("leyendo 128b pero len no múltiplo de 4\n");
		 for (i=0; i<len; i+=4){ // Recorremos el vector con un paso de 4
			x1 = *(unsigned long*) &array[i];
			x2 = *(unsigned long*) &array[i+2];
			asm("popcnt %[x1], %[val] \n\t" // Realizamos el popcount de x1 y lo guardamos val
				"popcnt %[x2], %[x1] \n\t"  // Realizamos el popcount de x2 y lo guardamos en
                                                                // x1
				"add     %[x1], %[val] \n\t" // Sumamos x1 y val y guardamos el resultado en val
			: [val]"=&r" (val)
			:  [x1]  "r" (x1),
			  [x2]  "r" (x2)
			);
			result += val;
		}
		return result;
}
```

Medición de tiempos

Finalmente, con las 10 versiones de popcount, nos queda el siguiente programa:

Noelia Escalera Mejías
Grupo A3

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

int resultado = 0;

#ifndef TEST
#define TEST 5
#endif

/* ------------------------------------------------------------------------ */
        #if TEST==1
/* ------------------------------------------------------------------------ */
        #define SIZE 4
        unsigned lista[SIZE]={0x80000000, 0x00400000, 0x00000200, 0x00000001};
        #define RESULT 4
/* ------------------------------------------------------------------------ */
#elif TEST==2
/* ------------------------------------------------------------------------ */
        #define SIZE 8
        unsigned lista[SIZE]={0x7fffffff, 0xffbfffff, 0xffffdfff, 0xfffffffe,
                0x01000023, 0x00456700, 0x8900ab00, 0x00cd00ef};
        #define RESULT 8
/* ------------------------------------------------------------------------ */
#elif TEST==3
/* ------------------------------------------------------------------------ */
        #define SIZE 8
        unsigned lista[SIZE]={0x0       , 0x01020408, 0x35906a0c, 0x70b0d0e0,
                0xffffffff, 0x12345678, 0x9abcdef0, 0xdeadbeef};
        #define RESULT 8
/* ------------------------------------------------------------------------ */
#elif TEST==4 || TEST==0
/* ------------------------------------------------------------------------ */
        #define NBITS 20
        #define SIZE (1<<NBITS)                 // tamaño suficiente para tiempo apreciable
        unsigned lista[SIZE];       // unsigned para desplazamiento derecha lógico
        #define RESULT ( NBITS * ( 1 << NBITS-1 ) )
/* ------------------------------------------------------------------------ */
#else
                #error "Definir TEST entre 0..4"
#endif
/* ------------------------------------------------------------------------ */

int popcount1(unsigned* array, size_t len)
{
        size_t i, j;
        int result = 0;
        unsigned x;

        for (i = 0; i < len; i++){ // Recorrer el vector
                x = array[i];
                for (j = 0; j < sizeof(int) * 8; j++){ // Recorremos cada entero del array por sus bits
                        unsigned bit = (x >> j) & 0x1; // Desplazamos los bits necesarios a la derecha y
                                                       // aplicamos la máscara
                        result += bit; // Añadimos el bit al resultado
                }
        }

        return result;
}
```

```
/* ------------------------------------------------------------------------ */

int popcount2(unsigned* array, size_t len)
{
	size_t i;
	int result = 0;
  unsigned x;

	for (i = 0; i < len; i++){ // Recorremos el vector
    x = array[i];
		while (x){   // Recorremos cada entero del array por sus bits, nos salimos del bucle cuando sea 0
			result += x & 0x1; // Añadimos al resultado el entero con la máscara aplicada
			x >>= 1; // Desplazamos un bit a la derecha
		}
	}

	return result;
}

/* ------------------------------------------------------------------------ */

int popcount3(unsigned* array, size_t len)
{
	int result = 0;
	unsigned x;
	size_t i;

	for (i = 0; i < len; i++){ // Recorremos el vector
		x = array[i];
		asm("\n"
	"ini3:                                  \n\t"
			"shr %[x]           \n\t"     // Desplaza un bit a la derecha
			"adc $0, %[r] \n\t" // Sumamos el último bit a result
			"test %[x], %[x] \n\t"
			"jnz ini3           \n\t" // Si x no es 0, salta a ini3

			: [r]"+r" (result)
			: [x] "r" (x)                   );
	}
	return result;
}

/* ------------------------------------------------------------------------ */

int popcount4(unsigned* array, size_t len)
{
	int result = 0;
	unsigned x;
	size_t i;
```

```
	for (i = 0; i < len; i++){
		x = array[i];
		asm("\n"
			"clc			\n\t" // Limpiamos el flag de acarreo
	"ini4:					\n\t"
			"adc $0, %[r] \n\t" // Sumamos el último bit a result
			"shr %[x]		\n\t" // Desplaza un bit a la derecha
			"jnz ini4\n\t" // Si x no es cero, vuelve a ini4
	"fin4:					\n\t"
			"adc $0, %[r]	\n\t" // Si x es cero, añade el último bit a result

			: [r] "+r" (result)
			: [x] "r" (x)			);

	}
	return result;
}

/* ---------------------------------------------------------------------- */

int popcount5(unsigned* array, size_t len)
{
	int result = 0, val = 0;
	size_t i, j;
	unsigned x;

	for (i = 0; i < len; i++){ // Recorremos el vector
		x = array[i];
		val = 0; // Variable local para acumular los bits
		for (j = 0; j < 8; j++){ // Recorremos cada entero
			val += x & 0x01010101; // Aplicamos la máscara (para 32 bits)
			x >>= 1; // Desplazamos un bit a la derecha
		}
		val += (val >> 16); // Sumamos los bits
		val += (val >> 8);
		result += val & 0xFF;
	}
	return result;
}

/* ---------------------------------------------------------------------- */

int popcount6(unsigned* array, size_t len)
{
	const unsigned m1 = 0x55555555;
	const unsigned m2 = 0x33333333;
	const unsigned m4 = 0x0f0f0f0f;
	const unsigned m8 = 0x00ff00ff;
	const unsigned m16 = 0x0000ffff;

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

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

		x = (x & m1) + ((x >> 1) & m1);  // Sumamos en árbol los bits
		x = (x & m2) + ((x >> 2) & m2);
		x = (x & m4) + ((x >> 4) & m4);
		x = (x & m8) + ((x >> 8) & m8);
		x = (x & m16) + ((x >> 16) & m16);

		result += x;
	}
	return result;
}

/* ---------------------------------------------------------------------- */

int popcount7(unsigned* array, size_t len)
{
	size_t i;
	unsigned long x1, x2;
	int result = 0;

	const unsigned long m1 = 0x5555555555555555;
	const unsigned long m2 = 0x3333333333333333;
	const unsigned long m4 = 0x0f0f0f0f0f0f0f0f;
	const unsigned long m8 = 0x00ff00ff00ff00ff;
	const unsigned long m16 = 0x0000ffff0000ffff;
	const unsigned long m32 = 0x00000000ffffffff;

	if (len & 0x3) printf("leyendo128b pero len no múltiplo de 4\n");

	for (i = 0; i < len; i +=4) // Recorremos el vector
	{
		x1 = *(unsigned long*) &array[i];
		x2 = *(unsigned long*) &array[i+2];

		x1 = (x1 & m1) + ((x1 >> 1) & m1); // Sumas en árbol
		x1 = (x1 & m2) + ((x1 >> 2) & m2);
		x1 = (x1 & m4) + ((x1 >> 4) & m4);
		x1 = (x1 & m8) + ((x1 >> 8) & m8);
		x1 = (x1 & m16) + ((x1 >> 16) & m16);
		x1 = (x1 & m32) + ((x1 >> 32) & m32);

		x2 = (x2 & m1) + ((x2 >> 1) & m1);
		x2 = (x2 & m2) + ((x2 >> 2) & m2);
		x2 = (x2 & m4) + ((x2 >> 4) & m4);
		x2 = (x2 & m8) + ((x2 >> 8) & m8);
		x2 = (x2 & m16) + ((x2 >> 16) & m16);
		x2 = (x2 & m32) + ((x2 >> 32) & m32);

		result += x1+x2;
	}
	return result;
}
```

```
/* ---------------------------------------------------------------------- */
int popcount8(unsigned *array, size_t len){
        size_t i;
  int val, result=0;
  int SSE_mask[] = {0x0f0f0f0f, 0x0f0f0f0f, 0x0f0f0f0f, 0x0f0f0f0f};
  int SSE_LUTb[] = {0x02010100, 0x03020201, 0x03020201, 0x04030302};
  //                    3 2 1 0    7 6 5 4    11 10 9 8   15 14 13 12

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

  for (i=0; i<len; i+=4){
        asm("movdqu %[x], %%xmm0   \n\t"
        "movdqa %%xmm0, %%xmm1              \n\t"  //  x: two copies xmm0-1
        "movdqu          %[m], %%xmm6                 \n\t"  // mask: xmm6
        "psrlw $4 , %%xmm1                \n\t"
        "pand %%xmm6, %%xmm0                \n\t"  //; xmm0 – lower nibbles
        "pand %%xmm6, %%xmm1                \n\t"  //; xmm1 – higher nibbles

        "movdqu %[l], %%xmm2                \n\t"  //; since instruction pshufb modifies LUT
        "movdqa %%xmm2, %%xmm3              \n\t"  //; we need 2 copies
        "pshufb %%xmm0, %%xmm2              \n\t"  //; xmm2 = vector of popcount lower nibbles
        "pshufb %%xmm1, %%xmm3              \n\t"  //; xmm3 = vector of popcount upper nibbles

        "paddb %%xmm2, %%xmm3               \n\t"  //; xmm3 - vector of popcount for 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 = [  not needed  |pcnt bytes0..15]
        "movd %%xmm0, %[val]"

        : [val]"=r" (val)
        : [x] "m" (array[i]),
        [m] "m" (SSE_mask[0]),
        [l] "m" (SSE_LUTb[0])
        );

        result += val;
  }

  return result;

}

/* ---------------------------------------------------------------------- */

int popcount9(unsigned* array, size_t len)
{
        size_t i;
        unsigned x;
        int val, result = 0;

        for (i = 0; i < len; i++) // Recorremos el vector
        {
                x = array[i];
                asm("popcnt %[x],%[val]" // Guardamos el popcount de x en val

                        :[val] "=r" (val)
                        :  [x]  "r" (x)
                );
```

```
result += val;
        }
        return result;
}

/* ---------------------------------------------------------------------- */

int popcount10(unsigned* array, size_t len)
{
        size_t i;
        unsigned long x1, x2;
        long val = 0;
        int result = 0;

        if (len & 0x3) printf("leyendo 128b pero len no múltiplo de 4\n");
                for (i=0; i<len; i+=4){ // Recorremos el vector con un paso de 4
                        x1 = *(unsigned long*) &array[i];
                        x2 = *(unsigned long*) &array[i+2];
                        asm("popcnt %[x1], %[val] \n\t" // Realizamos el popcount de x1 y lo guardamos
                                                        // val
                                "popcnt %[x2], %[x1] \n\t"  // Realizamos el popcount de x2 y lo
                                                            // guardamos en x1
                                "add     %[x1], %[val] \n\t" // Sumamos x1 y val y guardamos el resultado
                                                            // en val
                        : [val]"=&r" (val)
                        :  [x1]  "r" (x1),
                           [x2]  "r" (x2)
                        );
                        result += val;
                }
                return result;
}

/* ---------------------------------------------------------------------- */

void crono(int (*func)(), char* msg){
   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);
#if TEST==0
        printf( "%ld" "\n",     tv_usecs);
#else
   printf("resultado = %d\t", resultado);
   printf("%s:%9ld us\n", msg, tv_usecs);
#endif
}
```

```
int main()
{
        #if TEST==0 || TEST==4
  size_t i;                                             // inicializar array
  for (i=0; i<SIZE; i++)                        // se queda en cache
         lista[i]=i;
        #endif

  crono(popcount1 , "popcount1 (lenguaje C -      for)");
        crono(popcount2 , "popcount2 (lenguaje C -     while)");
        crono(popcount3 , "popcount3 (leng.ASM-body while 4i)");
        crono(popcount4 , "popcount4 (leng.ASM-body while 3i)");
        crono(popcount5 , "popcount5 (CS:APP2e 3.49-group 8b)");
        crono(popcount6 , "popcount6 (Wikipedia- naive - 32b)");
        crono(popcount7 , "popcount7 (Wikipedia- naive -128b)");
        crono(popcount8 , "popcount8 (asm SSE3 - pshufb 128b)");
        crono(popcount9 , "popcount9 (asm SSE4- popcount 32b)");
        crono(popcount10, "popcount10(asm SSE4- popcount128b)");

        #if TEST != 0
  printf("calculado %d\n", RESULT);
        #endif

  exit(0);
}
```

popcount.c

Para compilar y ejecutar el programa hemos usado el siguiente script de bash:

```
#!/bin/bash

for i in 0 g 1 2; do
  printf "__OPTIM%1c__%48s\n" $i "" | tr " " "="
  for j in $(seq 1 4); do
    printf "__TEST%02d__%48s\n" $j "" | tr " " "-"
    rm popcount
    gcc popcount.c -o popcount -O$i -D TEST=$j -g
    ./popcount
  done
done

for i in 0 g 1 2; do
  printf "__OPTIM%1c__%48s\n" $i "" | tr " " "="
  rm popcount
  gcc popcount.c -o popcount -O$i -D TEST=0
  for j in $(seq 0 10); do
    echo $j; ./popcount
  done | pr -11 -l 22 -w 80
done
```

script.sh

Hemos obtenido los siguientes resultados de las mediciones:

Optimización -O0	0	1	2	3	4	5	6	7	8	9	10	media
popcount1 (lenguaje C - for):	86355	76347	78809	78167	78122	77730	77539	75855	75732	77466	75988	77176
popcount2 (lenguaje C - while):	44828	39793	39919	39997	39889	39849	39555	39701	39814	39626	39786	39793
popcount3 (leng.ASM-body while 4i):	13229	11972	12130	11983	12071	11943	11918	11983	11977	11848	11915	11974
popcount4 (leng.ASM-body while 3i):	11396	11044	11260	11181	11175	11120	10967	11116	11017	10990	10980	11085
popcount5 (CS:APP2e 3.49-group 8b):	20014	19906	20009	20038	19917	20064	19801	20172	19935	19821	19820	19948
popcount6 (Wikipedia- naive - 32b):	9165	9059	9078	9185	9063	9134	9002	9026	9065	9018	9039	9067
popcount7 (Wikipedia- naive -128b):	4675	4685	4775	4695	4815	4742	4703	4660	4745	4672	4679	4717
popcount8 (asm SSE3 - pshufb 128b):	838	882	891	881	910	915	810	810	913	808	808	863
popcount9 (asm SSE4- popcount 32b):	2799	2793	2801	2791	2794	2753	2753	2731	2798	2753	2730	2770
popcount10(asm SSE4- popcount128b):	852	908	921	917	938	845	845	844	915	843	843	882

Optimización -Og	0	1	2	3	4	5	6	7	8	9	10	media
popcount1 (lenguaje C - for):	23000	24004	20696	23174	20608	22509	22450	22009	22344	21796	22106	22170
popcount2 (lenguaje C - while):	8330	11222	8016	8103	8061	8008	8077	8017	9569	8104	8135	8531
popcount3 (leng.ASM-body while 4i):	13030	13058	12519	12526	12503	12428	12508	12500	12486	12488	12523	12554
popcount4 (leng.ASM-body while 3i):	13579	12214	12174	12098	12069	12117	12124	12074	12138	12122	12124	12125
popcount5 (CS:APP2e 3.49-group 8b):	6441	5386	5303	5272	5266	5340	5288	5276	5299	5282	5294	5301
popcount6 (Wikipedia- naive - 32b):	3245	2612	2690	2635	2615	2611	2689	2611	2711	2693	2696	2656
popcount7 (Wikipedia- naive -128b):	1906	1601	1551	1553	1548	15911	1577	1558	1567	1571	1585	3002
popcount8 (asm SSE3 - pshufb 128b):	575	401	396	392	392	394	423	390	419	427	428	406
popcount9 (asm SSE4- popcount 32b):	618	464	487	455	452	455	484	451	482	482	489	470
popcount10(asm SSE4- popcount128b):	378	317	315	312	311	314	318	310	318	320	327	316

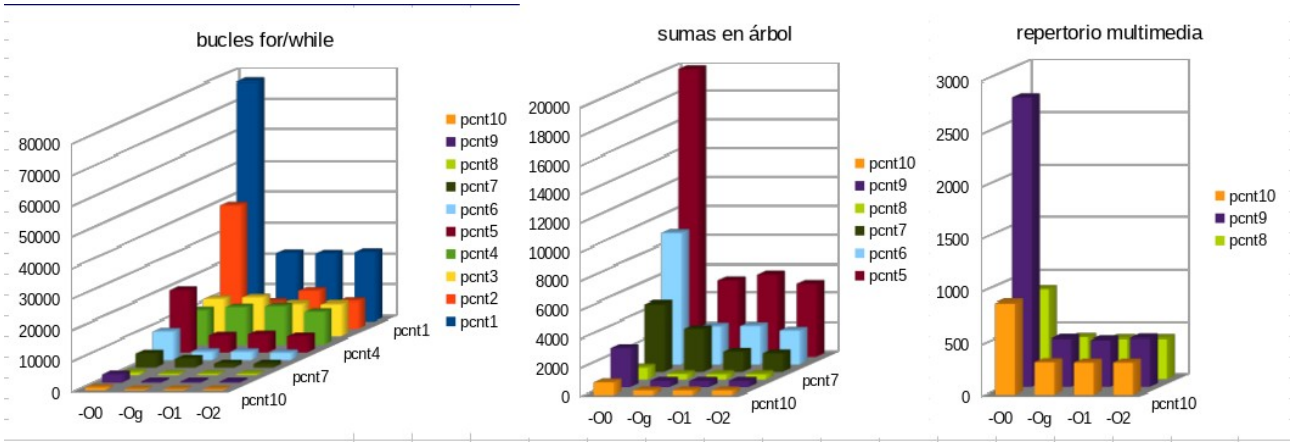
Optimización -O1	0	1	2	3	4	5	6	7	8	9	10	media
popcount1 (lenguaje C - for):	30444	24087	20783	20441	21301	22385	21981	21714	22286	22032	22510	21952
popcount2 (lenguaje C - while):	16491	13054	12185	12176	12176	12170	12156	12178	12176	12178	12156	12261
popcount3 (leng.ASM-body while 4i):	10741	13054	10255	10320	10243	10209	10325	10327	10319	10285	10322	10566
popcount4 (leng.ASM-body while 3i):	15928	12480	12442	12416	12424	12425	12377	12428	12450	12376	12401	12422
popcount5 (CS:APP2e 3.49-group 8b):	7517	5753	5614	5901	5727	5622	5742	5666	5764	5665	5600	5705
popcount6 (Wikipedia- naive - 32b):	3299	2642	2670	2766	2656	2648	2694	2643	2656	2705	2659	2674
popcount7 (Wikipedia- naive -128b):	2016	1456	1422	1540	1418	1426	1418	1419	1422	1441	1418	1438
popcount8 (asm SSE3 - pshufb 128b):	431	390	393	426	391	388	391	389	394	392	392	395
popcount9 (asm SSE4- popcount 32b):	637	453	452	482	451	453	452	450	459	452	453	456
popcount10(asm SSE4- popcount128b):	345	312	313	317	317	310	311	321	314	310	313	314

Optimización -O2	0	1	2	3	4	5	6	7	8	9	10	media
popcount1 (lenguaje C - for):	21969	23050	26141	22884	22710	20661	20521	20671	22576	22639	22663	22452
popcount2 (lenguaje C - while):	8383	8590	13331	8587	8401	8401	8432	8377	8250	8294	8482	8915
popcount3 (leng.ASM-body while 4i):	10638	10851	12478	10336	10231	10212	10194	10326	10217	10212	10169	10523
popcount4 (leng.ASM-body while 3i):	10157	10560	16031	9523	9539	9514	9570	9568	9589	11811	9499	10520
popcount5 (CS:APP2e 3.49-group 8b):	6007	5566	5000	4764	4625	5185	4828	5395	5131	4873	5083	5045
popcount6 (Wikipedia- naive - 32b):	2669	2411	2342	2393	2304	2338	2334	2327	2308	2314	2302	2337
popcount7 (Wikipedia- naive -128b):	1541	1318	1316	1422	1339	1316	1318	1316	1314	1314	1332	1331
popcount8 (asm SSE3 - pshufb 128b):	587	403	394	421	392	391	390	388	388	389	389	395
popcount9 (asm SSE4- popcount 32b):	657	554	492	488	463	456	457	462	455	457	457	474
popcount10(asm SSE4- popcount128b):	406	316	318	321	314	312	310	313	311	312	310	314

POPCOUNT:	-O0	-Og	-O1	-O2
pcnt1	77176	22170	**21952**	22452
pcnt2	39793	**8531**	12261	8915
pcnt3	11974	12554	**10566**	10523
pcnt4	11085	12125	**12422**	10520
pcnt5	19948	5301	5705	**5045**
pcnt6	9067	2656	2674	**2337**
pcnt7	4717	3002	1438	**1331**
pcnt8	863	406	395	**395**
pcnt9	2770	470	456	**474**
pcnt10	882	**316**	**314**	**314**

Ganancias:	-O0	-Og	-O1	-O2
pcnt1			1,00	
pcnt2		**2,57**		
pcnt3			2,08	
pcnt4			1,77	
pcnt5				**4,35**
pcnt6				**9,39**
pcnt7				**16,50**
pcnt8				**55,65**
pcnt9				46,30
pcnt10		**69,42**	69,96	69,98

Comentario
comparado con el for más rápido
el while es un 70% más rápido
ASM se queda en un 35%
o en un 43%
sumar en grupos 8b sale 3x más rápido
sumar en árbol 6x
lectura 128b sube a 10x
SSSE3 sube a 35x más rápido
SSE4 sólo 30x por leer 32b
SSE4 128b sube a 44x



Noelia Escalera Mejías
Grupo A3

Equipo usado para la práctica:

Host: HP ENVY 17 Notebook PC
OS: KDE neon User Edition 5.14 x86_64
Kernel: 4.15.0-38-generic
CPU: Intel i7-4700MQ (8 núcleos) 3.4 GHz
RAM: 4 GB

Noelia Escalera Mejías
Grupo A3