Estructura de computadores

Práctica 3:Popcount

Federico García Garrido.
fedeggj@correo.ugr.es
Grupo A2
Noviembre 2018.

Universidad de Granada.
Ingeniería Informática.
Estructura de computadores

INTRODUCCIÓN

En esta práctica hemos realizado diez versiones del popcount (contar los unos que tiene un número en binario) para ver las diferencias de optimización midiendo los tiempos y diferenciando las diferentes optimizaciones de cada opción de compilación.

Lo primero que hemos hecho ha sido estructurar nuestro código con lo dado en suma09_Casm.c y lo dicho en el guión para poder programar nuestros popcount en c y hacer las mediciones del tiempo para cada versión.

Cada versión está contenida en una función que tiene como parámetros una lista de números enteros y su longitud. Las pruebas se han realizado con un script para medir los tiempos dependiendo del compilador y con varias listas para ver el resultado con diferentes variables. El resultado es una variable global que se inicializa a 0 cada vez que empezamos una versión y devuelve el resultado una vez pasado por la función.

Implementaciones de las diferentes versiones

1.Lenguaje C- for:

```
45  #define WSIZE 8*sizeof(int)
46  int popcount1(unsigned* array, size_t len)
47  {
48      resultado = 0;
49      size_t i , j=0;
50      unsigned elemento;
51      for ( i=0; i < len; i++){
52          elemento = array[i];            //cogemos cada elemento de la lista
53          for (j=0; j < WSIZE; j++){      //recorremos cada bit
54              resultado += elemento & 0x1; //binario 0001
55              elemento >>= 1;             //desplazamos un bit a la derecha
56          }
57      }
58
59      return resultado;
60  }
```

En esta versión cogemos cada elemento y repasamos bit a bit el elemento desplazando a la derecha , con 0x1 (0001 en binario) y la funcion AND vamos sumando en resultado si hay o no un uno a la derecha (el bit desplazado). Este popcount es muy poco óptimo ya que recorremos bit a bit cada elemento.

2. Lenguaje C- while:

```
62  int popcount2(unsigned* array, size_t len)
63  {
64      resultado = 0;
65      size_t i =0;
66      unsigned elemento;
67      for ( i=0; i < len; i++){
68          elemento = array[i]; //cogemos cada elemento
69          while (elemento){       //mientras haya elemento que analizar
70              resultado += elemento & 0x1;        //acumulamos si hay un uno
71              elemento >>= 1;             //desplazamos el bit
72          }
73      }
74
75      return resultado;
76  }
```

Esta versión es casi igual que la primera hecha con el for solo que en vez de usar un for para recorrer cada bit del elemento usamos un while que mira si se ha acabado ya el elemento a analizar o no.

3. ASM-body-while 4i:

```
78  int popcount3(unsigned* array, size_t len)
79  {
80      size_t i;
81      unsigned x;
82      resultado = 0;
83
84      for (i=0; i<len; i++) {
85      x = array[i];
86      asm("\n"
87     "ini3:   \n\t"
88      "shr %[x]       \n\t"    //orden de desplazamiento del bit
89      "adc $0, %[r]    \n\t"    //Acumulamos en resultado
90      "test %[x], %[x] \n\t"    //Hace una conjuncion bit a bit y cambia el flag ZF
91      "jnz ini3        \n\t"    //Salta si el flag de cero está inactivo
92      : [r]"+r" (resultado)
93      : [x] "r" (x)   );
94      }
95
96      return resultado;
97  }
```

En esta versión ya implementamos código ensamblador dentro de la implementación en C. Esta versión coge tambien cada elemento de la lista, dentro de las instrucciones primero con la instrucción shr desplaza el bit a la derecha, si el bit menos significativo es 1 se activa el flag CF (acarreo), luego con la instrucción adc acumulamos en resultado 0+CF por lo tanto sumamos uno si el bit menos significativo es uno si no lo es sumará 0.

Por último para el bucle usamos la instrucción test con el elemento que nos va quedando de desplazar el bit, usamos esta instrucción test que es parecida a la función AND pero en vez de guardar el resultado en el registro destino lo que hace es activar el flag ZF (zero flag).

Lo que hacemos con esta instrucción es poner como parámetros el elemento con el mismo asi que el zero flag se pondra a 0 solo cuando el bit mas significativo (cuando se hayan desplazado todos los demás) sea igual a 0 asi también nos ahorramos una iteración, por último con la instrucción de salto jnz (jump if not equal zero) saltamos cuando no haya saltado el flag de cero con la orden test.

Luego guardamos el resultado y lo devolvemos en C.

4. ASM-body-while 3i:

```
100 int popcount4(unsigned* array, size_t len)
101 {
102     size_t i;
103     unsigned x;
104     resultado = 0;
105 
106     for (i=0; i<len; i++) {
107     x = array[i];
108     asm("\n"
109     "clc    \n\t"           //CLC para poder empezar por adc (Restaura el flag de acarreo)
110    "ini4:            \n\t"
111     "adc $0, %[r]    \n\t"  //acumulamos el bit de acarreo en resultado
112     "shr %[x]        \n\t"  //Desplazamiento del bit hacia CF
113     "jnz ini4        \n\t"  //Saltamos si ZF != 0
114     "adc $0, %[r]    \n\t"  //Acumulamos el ultimo bit restante
115     : [r]"+r" (resultado)
116     : [x] "r" (x)   );
117     }
118 
119     return resultado;
120 }
```

Esta versión es muy parecida a la anterior con leves diferencias de optimización ahorrándonos una instrucción, primero con CLC restauramos el bit de acarreo para la primera iteración, dentro del bucle usamos adc para acumular el resultado 0+CF, ahora con SHR es cuando viene el cambio ya que SHR también provoca cambios en el flag ZF para poder luego saltar a ini4. Una vez terminado el bucle hacemos otra instrucción adc para acumular el último movimiento de SHR del elemento.

5. CS:APP2e 3.49-group 8b:

```
122 int popcount5(unsigned* array, size_t len)
123 {
124     long val;
125     resultado = 0;
126     size_t i, j;
127     unsigned elemento;
128 
129     for (i = 0; i < len; i++){
130         elemento = array[i];    //Cogemos cada elemento
131         val = 0;                //Restauramos el valor auxiliar
132         for (j = 0;j < 8; j++){  //En este bucle acumulamos en cada byte el numero de bits
133             val += elemento & 0x01010101; //Acumulamos en el val auxiliar el and del elemento y la máscara
134             elemento >>= 1;
135         }
136         val += (val >> 16); //Desplazamos los bits
137         val += (val >> 8);
138         resultado += (val & 0xFF);  //Miramos todos los bits totales
139     }
140     return resultado;
141 }
```

En esta versión utilizamos una máscara (00000001 x4) para ver en cada byte cuantos bits hay, desplazando en un bit a la derecha conseguimos esto si usamos la función AND en cada una de las 8 iteracciones viendo los 8 bits de cada byte.

Luego movemos los unos a los dos bytes menos significativos y con 0xFF vemos cuantos unos tiene ese elemento.

6. Wikipedia – naive – 32b:

```
147 int popcount6(unsigned* array, size_t len)
148 {
149     const uint64_t m1 = 0x5555555555555555; //binary: 0101...
150     const uint64_t m2 = 0x3333333333333333; //binary: 00110011..
151     const uint64_t m4 = 0x0f0f0f0f0f0f0f0f; //binary: 4 zeros, 4 ones ...
152     const uint64_t m8 = 0x00ff00ff00ff00ff; //binary: 8 zeros, 8 ones ...
153     const uint64_t m16 = 0x0000ffff0000ffff; //binary: 16 zeros, 16 ones ...
154     const uint64_t m32 = 0x00000000ffffffff; //binary: 32 zeros, 32 ones
155 
156     resultado = 0;
157     size_t i;
158     uint64_t x;
159 
160     for (i = 0; i < len; i++){
161 
162         x = array[i];
163 
164         x = (x & m1 ) + ((x >> 1) & m1 ); //put count of each 2 bits into those 2 bits
165         x = (x & m2 ) + ((x >> 2) & m2 ); //put count of each 4 bits into those 4 bits
166         x = (x & m4 ) + ((x >> 4) & m4 ); //put count of each 8 bits into those 8 bits
167         x = (x & m8 ) + ((x >> 8) & m8 ); //put count of each 16 bits into those 16 bits
168         x = (x & m16) + ((x >> 16) & m16); //put count of each 32 bits into those 32 bits
169         x = (x & m32) + ((x >> 32) & m32); //put count of each 64 bits into those 64 bits
170         resultado += x;
171 
172     }
173     return resultado;
174 }
```

Esta versión es una versión sacada de wikipedia que utiliza varias máscaras para ir sacando el numero de unos en dos bits y guardar el resultado en los dos bits, el numero de unos cada 4 bits y guardarlo en 4 bits… asi hasta sacar los numeros de unos en los 64 bits y guardarlo en 64 bits sacando asi el numero de unos que existen en el elemento de 64 bits.

7. Wikipedia – naive-128b:

```
176 int popcount7(unsigned* array, size_t len)
177 {
178     size_t i;
179     unsigned long x1,x2;
180     resultado=0;
181 
182     const unsigned long m1 = 0x5555555555555555; //binary: 0101...
183     const unsigned long m2 = 0x3333333333333333; //binary 0011...
184     const unsigned long m4 = 0x0f0f0f0f0f0f0f0f; // 00001111...
185     const unsigned long m8 = 0x00ff00ff00ff00ff; // 0000 0000 1111 1111
186     const unsigned long m16 = 0x0000ffff0000ffff; // 16 zeros, 16 ones
187     const unsigned long m32 = 0x00000000ffffffff; //binary: 32 zeros, 32 ones
188 
189     if (len & 0x3) printf("leyendo 128b pero len no múltiplo de 4\n");
190 
191     for (i=0; i<len; i+=4)
192     {
193         x1 = *(unsigned long*) &array[i];
194         x2 = *(unsigned long*) &array[i+2];
195 
196         x1 = (x1 & m1 ) + ((x1 >> 1) & m1 ); //put count of each 2 bits into those 2 b
197         x1 = (x1 & m2 ) + ((x1 >> 2) & m2 ); //put count of each 4 bits into those 4 bits
198         x1 = (x1 & m4 ) + ((x1 >> 4) & m4 ); //put count of each 8 bits into those 8 bits
199         x1 = (x1 & m8 ) + ((x1 >> 8) & m8 ); //put count of each 16 bits into those 16 bits
200         x1 = (x1 & m16) + ((x1 >> 16) & m16); //put count of each 32 bits into those 32 bits
201         x1 = (x1 & m32) + ((x1 >> 32) & m32); //put count of each 64 bits into those 64 b
202         x2 = (x2 & m1 ) + ((x2 >> 1) & m1 );    //put count of each 2 bits into those 2 b
203         x2 = (x2 & m2 ) + ((x2 >> 2) & m2 ); //put count of each 4 bits into those 4 bits
204         x2 = (x2 & m4 ) + ((x2 >> 4) & m4 ); //put count of each 8 bits into those 8 bits
205         x2 = (x2 & m8 ) + ((x2 >> 8) & m8 ); //put count of each 16 bits into those 16 bits
206         x2 = (x2 & m16) + ((x2 >> 16) & m16); //put count of each 32 bits into those 32 bits
207         x2 = (x2 & m32) + ((x2 >> 32) & m32); //put count of each 64 bits into those 64 b
208 
209         resultado += x1+x2;
210     }
211 
212     return resultado;
213 }
```

Esta versión es parecida a la anterior pero la mejoramos aumentando el tamaño y desenrollamos el bucle para un menor número de iteraciones.

8. asm SSE3 -pshufb 128b:

```
215 int popcount8(unsigned* array, size_t len)
216 {
217
218     size_t i;
219     resultado = 0;
220     int val = 0;
221     int SSE_mask[] = { 0x0f0f0f0f, 0x0f0f0f0f, 0x0f0f0f0f, 0x0f0f0f0f };
222     int SSE_LUTb[] = { 0x02010100, 0x03020201, 0x03020201, 0x04030302 };
223
224     if (len & 0x3)
225         printf("leyendo 128b pero len no múltiplo de 4?n");
226     for (i = 0; i < len; i += 4) {
227         asm("movdqu        %[x], %%xmm0 \n\t"
228             "movdqa  %%xmm0, %%xmm1 \n\t"       //dos copias de [x]
229             "movdqu    %[m], %%xmm6 \n\t"       // máscara
230             "psrlw          $4, %%xmm1 \n\t"
231             "pand    %%xmm6, %%xmm0 \n\t"       //mordiscos inferiores
232             "pand    %%xmm6, %%xmm1 \n\t"       //mordiscos superiores
233
234             "movdqu    %[l], %%xmm2 \n\t"       //; ...como pshufb sobrescribe LUT
235             "movdqa  %%xmm2, %%xmm3 \n\t"       //; ...queremos 2 copias
236             "pshufb  %%xmm0, %%xmm2 \n\t"       //; xmm2 = vector popcount inferiores
237             "pshufb  %%xmm1, %%xmm3 \n\t"       //; xmm3 = vector popcount superiores
238
239             "paddb   %%xmm2, %%xmm3 \n\t"       //; xmm3 - vector popcount bytes
240             "pxor    %%xmm0, %%xmm0 \n\t"       //; xmm0 = 0,0,0,0
241             "psadbw  %%xmm0, %%xmm3 \n\t"       //;xmm3 = [pcnt bytes0..7|pcnt bytes8..15]
242             "movhlps %%xmm3, %%xmm0 \n\t"       //;xmm3 = [             0           |pcnt bytes0..7 ]
243             "paddd   %%xmm3, %%xmm0 \n\t"       //;xmm0 = [ no usado        |pcnt bytes0..15]
244             "movd    %%xmm0, %[val] \n\t"
245             : [val]"=r" (val)
246             : [x] "m" (array[i]),
247               [m] "m" (SSE_mask[0]),
248               [l] "m" (SSE_LUTb[0])
249             );
250         resultado += val;
251     }
252
253     return resultado;
254
255 }
```

Esta versión SSE3 (pshubf) tiene registros XMM de 128 bits que pueden contener 4 enteros de 32 bits, la operación pshubf permite barajar esos elementos indicando el tipo de baraje y el segundo argumento los datos a barajar. La idea es acelarar el proceso del conteo haciendo una tabla de cuantos bits tiene activado cada número hasta un límite dado por lo tatno se tarda menos en acceder al elemento. Para poder realizar esto no lo podemos meter en un registro XMM pero si lo limitamos a medio byte si que podemos (nibble:mordisco). Por lo tanto

podemos recorrer el array de 4 en 4 elementos cargando 4 enteros en un registro de XMM de 128 bits repartiendo los nibbles en dos registros XMM (en este caso XMM1 y XMM2). Una vez tenemos esto usamos máscaras para cada uno de los nibbles y sumamos cuantos bits tiene activados cada uno. La última parte sirve para acumular todos esos popcount en val.

9. asm SSE4 -popcount 32b

```
257 int popcount9(unsigned* array, size_t len)
258 {
259     resultado = 0;
260     size_t i;
261     int val = 0;
262     unsigned x;
263 
264     for (i=0; i < len; i++){
265         x = array [i];
266 
267         asm(
268             "popcnt %[x], %[val] \n\t"
269 
270             : [val]"=r"(val)
271             : [x] "r" (x)
272            );
273 
274         resultado += val;
275     }
276 
277     return resultado;
278 }
```

Esta versión tiene una instrucción SSE4 POPCNT ya integrada dentro de las instrucciones del procesador, veremos cuando comentemos los resultados de tiempo que pasar del repertorio SSE3 al SSE4 nos da ganancia de tiempo, pero esta versión solo lee en 32 bits por lo que tendremos que ir leyendo los elementos de la lista uno por uno.

10. asm SSE4 -popcount 128b

```
280  int popcount10(unsigned* array, size_t len)
281  {
282      size_t i;
283      unsigned long x1,x2;
284      long val;
285      resultado = 0;
286
287      if (len & 0x3)
288          printf("leyendo 128b len no multiplo de 4\n");
289
290      for (i=0; i<len; i+=4){
291          x1 = *(unsigned long*) &array[i];
292          x2 = *(unsigned long*) &array[i+2];
293
294          asm("popcnt %[x1],  %[val]  \n\t"
295              "popcnt %[x2],  %%rdi  \n\t"
296              "add    %%rdi, %[val] \n\t"
297          : [val] "=&r" (val)
298          : [x1]  "r" (x1),
299            [x2]  "r" (x2)
300          : "rdi"
301          );
302          resultado += val;
303      }
304
305      return resultado;
306  }
```

Esta versión es una versión que usa un repertorio SSE4 y a la vez lee en 128 bits (4 enteros) con lo que podemos desenrollar el bucle for leyendo valores de 4 en 4 por lo que funcionará 4 veces más rápido aproximadamente que el anterior, lo que hacemos es usar un registro %rdi de 64 bits que funcionará de registro auxiliar para poder leer los 128bits.

COMPARACIONES DE TIEMPO

Para todas las versiones de popcount hemos realizado pruebas de tiempo para varias listas, utilizando un script que nos compilase y ejecutase los diferentes TEST con diferentes opciones de compilación. Hemos obtenido los datos de todas las diferentes pruebas y las hemos introducido en una hoja de cálculo:

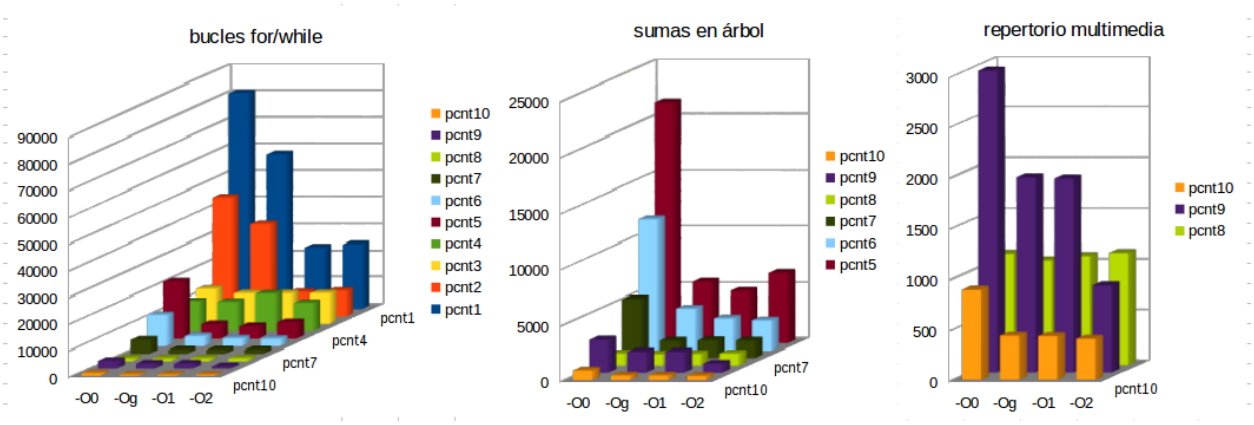
Optimización -O0	0	1	2	3	4	5	6	7	8	9	10	media
popcount1 (lenguaje C - for):	80959	81359	80748	80684	80760	80753	80687	80721	80861	81351	81699	80962
popcount2 (lenguaje C - while):	47070	44061	42352	42365	49119	46958	42347	42340	42613	42171	51343	44567
popcount3 (leng.ASM-body while 4i):	12256	12156	12228	12186	12137	12194	12197	15843	16863	12086	12203	13009
popcount4 (leng.ASM-body while 3i):	11442	11350	11325	11332	11335	11377	11336	11333	11333	11190	11324	11324
popcount5 (CS:APP2e 3.49-group 8b):	21483	21421	21436	21621	21451	21412	22016	21413	21397	21524	21421	21511
popcount6 (Wikipedia- naive - 32b):	11582	11681	11548	11623	11614	11567	13192	11574	11550	11629	11633	11761
popcount7 (Wikipedia- naive -128b):	5128	5122	5120	5144	5129	5129	6855	5093	5139	5102	5048	5288
popcount8 (asm SSE3 - pshufb 128b):	1071	1070	1068	1072	1076	1071	1390	1067	1081	1118	1037	1105
popcount9 (asm SSE4- popcount 32b):	2955	2998	2940	2953	2923	2923	3398	2917	2918	2931	2969	2987
popcount10(asm SSE4- popcount128b):	887	913	887	929	880	885	868	884	884	864	942	894

Optimización -Og	0	1	2	3	4	5	6	7	8	9	10	media
popcount1 (lenguaje C - for):	57203	58181	58043	58086	58122	58020	58231	58143	57108	58024	58127	58009
popcount2 (lenguaje C - while):	34825	34150	34724	34586	34577	34375	34595	34586	34540	34536	34815	34548
popcount3 (leng.ASM-body while 4i):	11331	11265	11648	11263	11260	11289	11290	11264	11262	11283	12129	11395
popcount4 (leng.ASM-body while 3i):	10844	10751	12494	10806	10770	11499	10840	10875	10829	10887	11164	11092
popcount5 (CS:APP2e 3.49-group 8b):	5491	5583	5534	5494	5564	5578	5481	5503	5504	5497	5562	5530
popcount6 (Wikipedia- naive - 32b):	3802	3811	3742	3790	3790	3742	3816	3798	3821	3829	3737	3788
popcount7 (Wikipedia- naive -128b):	1615	1580	1546	1592	1598	1591	1595	1595	1577	1588	1595	1586
popcount8 (asm SSE3 - pshufb 128b):	1023	1073	1022	1022	1026	1046	1035	1025	1064	1038	1021	1037
popcount9 (asm SSE4- popcount 32b):	1940	1897	1874	1921	1920	1967	1938	1942	1917	1930	1946	1925
popcount10(asm SSE4- popcount128b):	435	423	440	436	449	421	423	438	452	418	528	443

Optimización -O1	0	1	2	3	4	5	6	7	8	9	10	media
popcount1 (lenguaje C - for):	23716	23697	23388	22904	22565	23446	23621	22650	22936	22589	22975	23077
popcount2 (lenguaje C - while):	12947	9728	9639	8870	9098	8963	8894	8906	8967	8877	9403	9135
popcount3 (leng.ASM-body while 4i):	11878	11941	11791	11387	11391	11303	11322	11291	11332	11276	11302	11434
popcount4 (leng.ASM-body while 3i):	13688	14632	15302	13717	14010	14535	13694	13667	14363	13639	13629	14119
popcount5 (CS:APP2e 3.49-group 8b):	4591	4677	4603	4970	4880	4648	4692	4655	4674	4709	4647	4716
popcount6 (Wikipedia- naive - 32b):	2791	3550	2851	2845	2843	2789	2841	2832	2853	2842	2839	2909
popcount7 (Wikipedia- naive -128b):	1518	1546	1520	2046	1541	1524	1545	1610	1541	1551	1575	1600
popcount8 (asm SSE3 - pshufb 128b):	1118	1051	1045	1097	1016	1043	1064	1066	1071	1105	1246	1080
popcount9 (asm SSE4- popcount 32b):	1927	1885	1889	1948	1882	1881	1950	1888	1936	1881	1996	1914
popcount10(asm SSE4- popcount128b):	460	441	434	475	445	443	422	426	437	434	424	438

Optimización -O2	0	1	2	3	4	5	6	7	8	9	10	media
popcount1 (lenguaje C - for):	26630	23290	23288	23294	23295	23275	23517	24474	25146	24451	28040	24207
popcount2 (lenguaje C - while):	8720	8682	8878	8686	8893	8643	12832	9949	10053	9434	9719	9577
popcount3 (leng.ASM-body while 4i):	11339	11056	11121	11073	11181	11059	13455	11455	11718	11495	11075	11469
popcount4 (leng.ASM-body while 3i):	10457	10426	10453	10440	10440	10488	11908	10979	11155	10763	10526	10758
popcount5 (CS:APP2e 3.49-group 8b):	6213	6169	6465	6148	6157	6162	6134	7265	6196	6186	6189	6307
popcount6 (Wikipedia- naive - 32b):	2656	2653	3605	2638	2630	2631	2670	2644	2612	2650	2682	2742
popcount7 (Wikipedia- naive -128b):	1452	1450	2339	1417	1437	1442	1923	1426	1410	1447	1492	1578
popcount8 (asm SSE3 - pshufb 128b):	1028	1028	1155	1042	1030	1027	1724	1024	1009	1043	1031	1111
popcount9 (asm SSE4- popcount 32b):	792	761	1195	758	759	764	1229	768	742	883	777	864
popcount10(asm SSE4- popcount128b):	359	412	399	429	382	383	504	361	377	474	377	410

Haciendo varios test con la misma versión podemos obtener su media para obtener un valor más general de cada versión, podemos observar como las opciones de compilación se van optimizando hasta el punto de mejorar el tiempo en ¾ en la optimización entre el -O0 y el -O2 pasando de 80962 a 24207 en el ejemplo del popcount1 o lo mismo a la mitad con el popcount10 pasando de 894 a 410 con esto concluimos que las optimizaciones de los compiladores hoy dia es muy potente.



POPCOUNT:	-O0	-Og	-O1	-O2		Ganancias:	-O0	-Og	-O1	-O2
pcnt1 (lenguaje C - for):	80962	58009	**23077**	24207		pcnt1			1,00	
pcnt2 (lenguaje C - while):	44567	**34548**	9135	9577		pcnt2		**0,67**		
pcnt3 (leng.ASM-body while 4i):	13009	11395	**11434**	11469		pcnt3			2,02	
pcnt4 (leng.ASM-body while 3i):	11324	11092	**14119**	10758		pcnt4			1,63	
pcnt5 (CS:APP2e 3.49-group 8b):	21511	5530	4716	**6307**		pcnt5				**3,66**
pcnt6 (Wikipedia- naive - 32b):	11761	3788	2909	**2742**		pcnt6				**8,42**
pcnt7 (Wikipedia- naive -128b):	5288	1586	1600	**1578**		pcnt7				**14,62**
pcnt8 (asm SSE3 - pshufb 128b):	1105	1037	1080	**1111**		pcnt8				**20,77**
pcnt9 (asm SSE4- popcount 32b):	2987	1925	1914	**864**		pcnt9				26,72
pcnt10 (asm SSE4- popcount128b):	894	**443**	**438**	**410**		pcnt10		**52.12**	52.68	56.31

Pasando a las distintas versiones de popcount vamos a ir comparando con el for más rápido (bit a bit) que es la opcion -O1 con las diferentes opciones de los otros popcount.

La segunda versión del while es entre un 65% y un 70% mas rápida, las dos versiones con un cuerpo de bucle de ensamblador es de dos veces mas rápida para 4 instrucciones y solo 1,5 veces más rapido en la versión de 3 instrucciones en la opción de compilador -O1. Vemos también que la opción while sin ensamblador en opciones de compilador mas potentes es mucho mejor nuestra código en ensamblador.

Si seguimos comparando con el for vemos que sumar en grupos de 8b sale 3,6 veces más rápido y que sumar en árbol sale aproximadamente unos 8 veces y media más rápido y que leer en 128b sale 14 veces y media más rápido aproximadamente.

Pasando a los SSE vemos que pasa a ir 20 veces más rápido para SSE3 en lectura de 128b, cambiando de repositorio a SSE4 va 26 veces más rápido y ya si leemos en 128b y usamos SSE4 va 52 veces más rápido.