Estructura de Computadores (2016-2017)
Subgrupo C3
Grado en Ingeniería Informática
Universidad de Granada

Práctica 3: Programación mixta C-asm x86 Linux

Mario Rodríguez Ruiz

23 de noviembre de 2016

Índice

1 Diario de trabajo **4**
1.1 Primera sesión 4
1.2 Segunda sesión 4
1.3 Tercera sesión 4

2 Ejercicio 4.0: Preguntas de autocomprobación (suma_01-suma_09) **5**
2.1 Sesión de depuración suma_01_S_cdecl 5
2.2 Sesión de depuración suma_02_S_libC 5
2.3 Sesión de depuración suma_03_SC 5
2.4 Sesión de depuración suma_04_SC 5
2.5 Sesión de depuración suma_05_SC 5
2.6 Sesión de depuración suma_07_Casm 6
2.7 Sesión de depuración suma_08_Casm 6

3 Ejercicio 4.1: Calcular la suma de bits de una lista de enteros sin signo **7**
3.1 Primera versión 7
3.2 Segunda versión 7
3.3 Tercera versión 8
3.4 Cuarta versión 8
3.5 Quinta versión 9
3.6 Sexta versión 10
3.7 Mediciones: cronometrar las distintas versiones con -O0, -O1 y -O2 11
3.8 Representación de los resultados 12

4 Ejercicio 4.2: Calcular la suma de paridades de una lista de entero sin signo **13**
4.1 Primera versión 13
4.2 Segunda versión 13
4.3 Tercera versión 14
4.4 Cuarta versión 14
4.5 Quinta versión 15
4.6 Sexta versión 16
4.7 Mediciones: cronometrar las distintas versiones con -O0, -O1 y -O2 17
4.8 Representación de los resultados 18

Índice de figuras

3.1. Ejecución del programa **popcount** con **-O0** 11
3.2. Ejecución del programa **popcount** con **-O1** 11
3.3. Ejecución del programa **popcount** con **-O2** 11
3.4. Resultado de las ejecuciones de **popcount** 12
3.5. Resumen de los promedios 12
3.6. Gráficas de las ejecuciones de **popcount** 12

4.1. Ejecución del programa **parity** con **-O0** 17
4.2. Ejecución del programa **parity** con **-O1** 17
4.3. Ejecución del programa **parity** con **-O2** 17
4.4. Resultado de las ejecuciones de **parity** 18
4.5. Resumen de los promedios 18
4.6. Gráficas de las ejecuciones de **parity** 18

1. Diario de trabajo

1.1. Primera sesión

Finalizar el tutorial de la práctica 3. Ejecutar las distintas versiones del programa **suma**:
La primera versión consistía en modificar el código para que funcionara según **cdecl**.
La segunda añadía la llamada **printf()** para sacar por pantalla el resultado en decimal y hexadecimal que sustituye la llamada directa al kernel Linux.
La tercera eliminaba cualquier rastro de ensamblador en **suma()** creando una nueva equivalencia completa en **C**.
La cuarta dejaba como única instrucción un salto a la subrutina suma.
La quinta era dejar completamente el código en **lenguaje C**.
La sexta consistía en volver a pasar la función suma a un módulo ensamblador separado.
En la séptima se volvía a incorporar el código ensamblador de suma como ensamblador en-linea.
La octava versión reducía el código ensamblador en-linea al cuerpo del bucle **for**.
Y por último, en la novena se creaban **tres alternativas a suma** cronometrando sus tiempos de ejecución.

1.2. Segunda sesión

Realización de las preguntas de autocomprobación desde suma_01 hasta suma_09.
Estudio de las distintas versiones de **popcount** a realizar así como una pequeña estructuración y organización de cada uno de ellos.
Desarrollo de cada uno de los códigos correspondientes a **popcount**, por medio del guión de prácticas con la ayuda de las transparencias de clase y del libro de la asignatura para llegar a sus soluciones.
Toma de los resultados obtenidos en las ejecuciones y elaboración de un estudio por medio de una hoja de cálculo con sus correspondientes gráficas finales.

1.3. Tercera sesión

Estudio de las distintas versiones de **parity** a realizar así como una pequeña estructuración y organización de cada uno de ellos.
Desarrollo de cada uno de los códigos correspondientes a **parity**, por medio del guión de prácticas con la ayuda de las transparencias de clase y del libro de la asignatura para llegar a sus soluciones.
Toma de los resultados obtenidos en las ejecuciones y elaboración de un estudio por medio de una hoja de cálculo con sus correspondientes gráficas finales.

2. Ejercicio 4.0: Preguntas de autocomprobación (suma_01-suma_09)

2.1. Sesión de depuración suma_01_S_cdecl

4.¿Qué modos de direccionamiento usa la instrucción add (%ebx, %edx,4), %eax?

La instrucción usa el modo de direccionamiento inmediato.
Los componentes son: add(rt, rs, imm),escala.

2.2. Sesión de depuración suma_02_S_libC

1.¿Qué error se obtiene si no se añade –lc al comando de enlazar? ¿Qué tipo de error es? (en tiempo de ensamblado, enlazado, ejecución...)

Se obtienen varios errores de referencias a funciones que no están definidas.
Se trata de un error en tiempo de enlazado.

2.3. Sesión de depuración suma_03_SC

2.¿Qué diferencia hay entre los comandos Next y Step?

La diferencia está en que, por ejemplo, si se encuentra en una llamada a función y se ejecuta el comando **Next**, la función se ejecutará y volverá. En cambio con **Step** se irá a la primera línea de esa función.

2.4. Sesión de depuración suma_04_SC

3. Otras diferencias están en el manejo de pila. Explicar dichas diferencias.

La función **__printf_chk flag** comprobará el desbordamiento de pila antes de calcular un resultado, dependiendo del valor del parámetro *flag*. Si se prevé un desbordamiento, la función se cancelará y el programa se saldrá del programa que la llama.

2.5. Sesión de depuración suma_05_SC

5. Se aplica una máscara al punte ero de pila. ¿Cuál, y qué efecto produce? (Pista: alineamiento). Nosotros usam mos .int para de clarar enteros y arrays. ¿Qué usa gcc? Nosotros usamos el contador de posiciones y aritmé ética de etiquetas para calcular la longitud del array. ¿Qué usa gcc? Nosotros hemos usado push para introducir argumentos en pila, ¿Cuál usa gcc?

Máscara de pila: **leaq 0(,%rax,4),%rdx**
Para declarar enteros y arrays gcc usa **.long**
gcc para introducir argumentos en pila utiliza **pushq**

2.6. Sesión de depuración suma_07_Casm

Por motivos estéticos, a veces se terminan las líneas ASM con "$\backslash n$" ***y otras con*** "$\backslash n\backslash t$"***. ¿Por qué en este caso apenas se ha usado*** "$\backslash t$"***?***

No se ha usado apenas "$\backslash t$" para que cuando se imprima el código para su depuración o para su lectura resulte más cómodo.

2.7. Sesión de depuración suma_08_Casm

Si res es variable de salida, ¿por qué se le ha indicado restricción "$+r$"***, en lugar de*** "$=r$"***?***

Se le ha indicado la restricción "$+r$" porque ésta le dice a gcc qué necesita para cargar un registro con el valor, es decir, significa que este operando es leído y escrito por la instrucción.
Cuando el compilador arregla los operandos para satisfacer las restricciones, necesita saber qué operandos son leídos por la instrucción y que son cuáles por ella. '=' Identifica un operando que solamente se encuentra escrito, sin embargo '+' identifica un operando que es escrito y a la vez leído.

3. Ejercicio 4.1: Calcular la suma de bits de una lista de enteros sin signo

3.1. Primera versión

```
1  // Primera versión, previsiblemente con peores prestaciones.
2  int popcount1(unsigned* array, int len)
3  {
4    int  i,j,  res=0;
5    for (i=0; i<len; i++){
6      unsigned x =array[i];
7      for (j=0; j<WSIZE;j++){      // Recorre los bits
8        res += x & 0x1;            // Aplica la máscara 0x1
9        x >>=1;                    // Desplaza a la derecha
10     }
11   }
12   return res;
13 }
```

3.2. Segunda versión

```
1  int popcount2(unsigned* array, int len)
2  {
3    int  i, res=0;
4    unsigned x;
5    for (i=0; i<len; i++){
6      x=array[i];
7      do{
8        res += x & 0x1;          // Aplica la máscara 0x1
9        x >>=1;                  // Desplaza a la derecha
10     }while(x);                 // Recorre los bits
11   }
12   return res;
13 }
```

3.3. Tercera versión

```
1  // Traduce el bucle interno while por código ASM.
2  int popcount3(unsigned* array, int len){
3    int res=0,i;
4    unsigned x;
5
6    for(i=0; i<len; i++){
7      x=array[i];
8      asm("\n"
9        "ini3:    \n\t"          // seguir mientras que x!=0
10       "shr %[x] \n\t"          // LSB en CF
11       "adc $0, %[r] \n\t"      // suma con acarreo
12       "cmp $0,%[x] \n\t"       // compara
13       "jne  ini3 \n\t"         // salto, modificando CF y ZF
14       : [r]"+r" (res)          // e/s: añadir a lo acumulado
             por el momento
15       : [x]"r" (x)             // entrada: valor elemento
16     );
17   }
18   return res;
19 }
```

3.4. Cuarta versión

```
1  int popcount4(unsigned* array, int len){
2    int i,j;
3    int result = 0;
4    unsigned n;
5
6    for(i = 0; i < len; i++){
7      int res = 0;
8      n = array[i];
9      for(j = 0; j < 8; j++){      // Aplica la máscara a cada
           elemento.
10       res += n & 0x01010101;     // Acumula los bits de cada
             byte.
11       n >>= 1;
12     }
13     // Suma en árbol los 4B
14     res += (res >> 16);
15     res += (res >> 8);
```

```
16      result += (res & 0xff);
17    }
18    return result;
19  }
```

3.5. Quinta versión

```
1  // Versión SSSE3 (pshufb)
2  int popcount5(unsigned* array, int len) {
3    int i;
4    int res, result = 0;
5    int SSE_mask[] = { 0x0f0f0f0f, 0x0f0f0f0f, 0x0f0f0f0f, 0
       x0f0f0f0f };
6    int SSE_LUTb[] = { 0x02010100, 0x03020201, 0x03020201, 0
       x04030302 };
7
8    if (len & 0x3)
9      printf("leyendo 128b pero len no múltiplo de 4?n");
10   for (i = 0; i < len; i += 4) {
11     asm("movdqu         %[x], %%xmm0 \n\t"
12     "movdqa   %%xmm0, %%xmm1 \n\t"        // dos copias de x
13     "movdqu     %[m], %%xmm6 \n\t"        // máscara
14     "psrlw            $4, %%xmm1 \n\t"
15     "pand     %%xmm6, %%xmm0 \n\t"        // xmm0 - nibbles
         inferiores
16     "pand     %%xmm6, %%xmm1 \n\t"        // xmm1 - nibbles
         superiores
17
18     "movdqu     %[l], %%xmm2 \n\t"        // como pshufb
         sobrescribe LUT
19     "movdqa   %%xmm2, %%xmm3 \n\t"        // queremos 2 copias
20     "pshufb   %%xmm0, %%xmm2 \n\t"        // xmm2 = vector
         popcount inferiores
21     "pshufb   %%xmm1, %%xmm3 \n\t"        // xmm3 = vector
         popcount superiores
22
23     "paddb    %%xmm2, %%xmm3 \n\t"        // xmm3 - vector
         popcount bytes
24     "pxor     %%xmm0, %%xmm0 \n\t"        // xmm0 = 0,0,0,0
25     "psadbw   %%xmm0, %%xmm3 \n\t"        // xmm3 = [pcnt
         bytes0..7|pcnt bytes8..15]
```

```
26      "movhlps %%xmm3, %%xmm0 \n\t"        // xmm3 = [        0
                |pcnt bytes0..7 ]
27      "paddd   %%xmm3, %%xmm0 \n\t"        // xmm0 = [   no
           usado   |pcnt bytes0..15]
28      "movd    %%xmm0, %[res] \n\t"
29      : [res]"=r" (res)
30      : [x]  "m"  (array[i]),
31      [m]  "m"  (SSE_mask[0]),
32      [l]  "m"  (SSE_LUTb[0])
33      );
34      result += res;
35    }
36    return result;
37  }
```

3.6. Sexta versión

```
1  // Versión SS4.2
2  int popcount6(unsigned* array, int len) {
3    int i;
4    unsigned x;
5    int val, result=0;
6
7    for(i=0; i<len; i++){
8      x=array[i];
9      asm("popcnt %[x], %[val]"
10     : [val] "=r" (val)
11     :   [x]  "r" (x)
12     );
13     result += val;
14   }
15   return result;
16 }
```

3.7. Mediciones: cronometrar las distintas versiones con -O0, -O1 y -O2

```
[mario@manjario fuentes]$ make popcount
cc -m32 -O0 -Wall popcount.c -o popcount
[mario@manjario fuentes]$ for((i=0;i<11;i++)); do echo $i ; ./popcount ; done | pr -11 -l 20 -w 100


2016-11-15 19:21                                                                                    Página 1


0       1       2       3       4       5       6       7       8       9       10
134526  102265  134611  132743  103006  102800  133786  102241  135439  103115  132505
 58108   58070   58068   58270   58083   58012   58052   57968   58029   58098   57978
 17140   17172   17158   17203   17162   17154   17164   17151   17174   17162   17142
 29254   29292   29317   29384   30427   29361   29279   29271   29339   29289   29291
  1332    1336    1311    1331    1336    1326    1313    1317    1339    1324    1339
  3869    3871    3855    3873    3866    3880    3865    3870    3862    3870    3866
```

Figura 3.1: Ejecución del programa **popcount** con **-O0**

```
[mario@manjario fuentes]$ make popcount
cc -m32 -O1 -Wall popcount.c -o popcount
4[mario@manjario fuentes]$ for((i=0;i<11;i++)); do echo $i ; ./popcount ; done | pr -11 -l 20 -w 100


2016-11-15 19:17                                                                                    Página 1


0       1       2       3       4       5       6       7       8       9       10
 58936   55534   68341   42253   69306   69118   45043   42329   42502   61942   42243
 18965   18830   22635   17127   21512   21504   17220   17158   17144   20047   17134
 17037   16997   18677   17045   18940   18960   16812   16712   16687   17867   16699
  8161    8023    7893    7925    8521    8682    7796    8119    7788    7869    7941
   645     643     675     648     675     672     656     648     666     669     647
   931     925     965     952     959     986     941     957     942     945     934
```

Figura 3.2: Ejecución del programa **popcount** con **-O1**

```
[mario@manjario fuentes]$ make popcount
cc -m32 -O2 -Wall popcount.c -o popcount
[mario@manjario fuentes]$ for((i=0;i<11;i++)); do echo $i ; ./popcount ; done | pr -11 -l 20 -w 100


2016-11-15 19:23                                                                                    Página 1


0       1       2       3       4       5       6       7       8       9       10
 49385   52650   36218   50129   28535   28561   40783   50163   50106   38306   28658
 23457   23739   19076   23840   17180   17127   19789   23823   23777   19850   17193
 20317   20481   17105   20538   16689   16683   17724   20530   20566   17865   16767
 12594   12522   11133   12698   11066   11128   11231   14315   12630   11339   11371
   664     662     603     659     610     602     612     666     667     615     606
  1031    1016     961    1020     942     955     969    1025    1020     945     967
```

Figura 3.3: Ejecución del programa **popcount** con **-O2**

3.8. Representación de los resultados

/proc/cpuinfo	Intel(R) Pentium(R) CPU 2020M @ 2.40GHz cache size: 2048 KB
POPCOUNT:	gcc -m32 -O<n> -Wall popcount.c -o popcount for((i=0;i<11;i++)); do echo $i ; ./popcount ; done | pr -11 -l 20 -w 100

Optimización con -O0	0	1	2	3	4	5	6	7	8	9	10	Media
popcount1 (lenguaje C – for)	134526	102265	134611	132743	103006	102800	133786	102241	135439	103115	132505	119731
popcount2 (lenguaje C – while)	58108	58070	58068	58270	58083	58012	58052	57968	58029	58098	57978	58067
popcount3 (ASM– cuerpo while)	17140	17172	17158	17203	17162	17154	17164	17151	17174	17162	17142	17162
popcount4 (l.CS:APP 3,49 – 8b)	29254	29292	29317	29384	30427	29361	29279	29271	29339	29289	29291	29409
popcount5 (asm SSE3)	1332	1336	1311	1331	1336	1326	1313	1317	1339	1324	1339	1328
popcount6 (asm SSE4 – 32b)	3869	3871	3855	3873	3866	3880	3865	3870	3862	3870	3866	3868

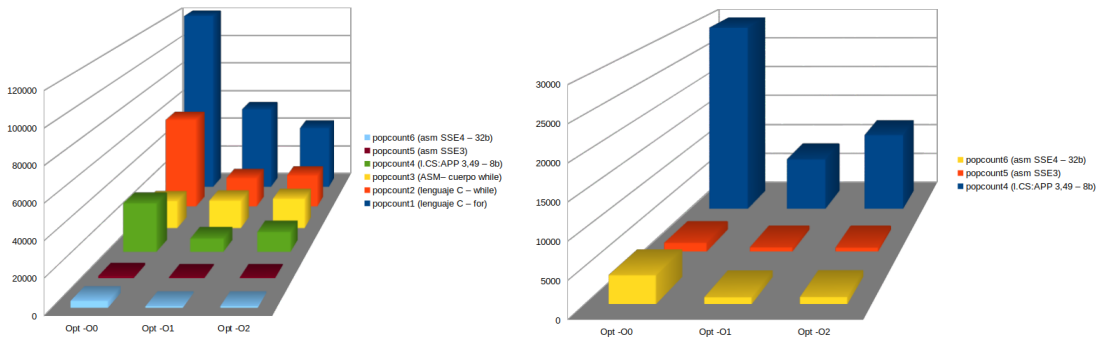
Optimización con -O1	0	1	2	3	4	5	6	7	8	9	10	Media
popcount1 (lenguaje C – for)	58936	55534	68341	42253	69306	69118	45043	42329	42502	61942	42243	54322
popcount2 (lenguaje C – while)	18965	18830	22635	17127	21512	21504	17220	17158	17144	20047	17134	19025
popcount3 (ASM– cuerpo while)	17037	16997	18677	17045	18940	18960	16812	16712	16687	17867	16699	17494
popcount4 (l.CS:APP 3,49 – 8b)	8161	8023	7893	7925	8521	8682	7796	8119	7788	7869	7941	8065
popcount5 (asm SSE3)	645	643	675	648	675	672	656	648	666	669	647	659
popcount6 (asm SSE4 – 32b)	931	925	965	952	959	986	941	957	942	945	934	949

Optimización con -O2	0	1	2	3	4	5	6	7	8	9	10	Media
popcount1 (lenguaje C – for)	49385	52650	36218	50129	28535	28561	40783	50163	50106	38306	28658	41227
popcount2 (lenguaje C – while)	23457	23739	19076	23840	17180	17127	19789	23823	23777	19850	17193	20805
popcount3 (ASM– cuerpo while)	20317	20481	17105	20538	16689	16683	17724	20530	20566	17865	16767	18660
popcount4 (l.CS:APP 3,49 – 8b)	12594	12522	11133	12698	11066	11128	11231	14315	12630	11339	11371	12002
popcount5 (asm SSE3)	664	662	603	659	610	602	612	666	667	615	606	633
popcount6 (asm SSE4 – 32b)	1031	1016	961	1020	942	955	969	1025	1020	945	967	986

Figura 3.4: Resultado de las ejecuciones de **popcount**

POPCOUNT:	Opt -O0	Opt -O1	Opt -O2
popcount1 (lenguaje C – for)	119731	54322	41227
popcount2 (lenguaje C – while)	58067	19025	20805
popcount3 (ASM– cuerpo while)	17162	17494	18660
popcount4 (l.CS:APP 3,49 – 8b)	29409	8065	12002
popcount5 (asm SSE3)	1328	659	633
popcount6 (asm SSE4 – 32b)	3868	949	986

Figura 3.5: Resumen de los promedios



(a) Completa (b) Reducida para apreciar el factor mejora

Figura 3.6: Gráficas de las ejecuciones de **popcount**

4. Ejercicio 4.2: Calcular la suma de paridades de una lista de entero sin signo

4.1. Primera versión

```
1  // Primera versión
2  int parity1(unsigned* array, int len)
3  {
4    int  i,j,  res=0,val;
5    for (i=0; i<len; i++){           // Recorre el array
6      val=0;
7      unsigned x =array[i];
8      for (j=0; j<WSIZE;j++){      // Recorre los bits
9        val ^= x & 0x1;           // Aplica la máscara y acumula
             lateralmente los bits
10       x >>=1;                   // Desplaza a la derecha para
             extraer y acumular bits
11     }
12     res+=val;
13   }
14   return res;
15 }
```

4.2. Segunda versión

```
1  // Segunda versión
2  int parity2(unsigned* array, int len)
3  {
4    int  i,res=0,val;
5    unsigned x;
6    for (i=0; i<len; i++){
7      val=0;
8      x=array[i];
9      do{
10       val ^= x & 0x1;      // Aplica la máscara y acumula
             lateralmente los bits
11       x >>=1;              // Desplaza a la derecha para
             extraer y acumular bits
12     }while(x);               // Recorre el array
13     res+=val;
14   }
```

```
15   return res;
16 }
```

4.3. Tercera versión

```
1  // Tercera versión: Adapta la segunda versión para el array
       completo
2  int parity3(unsigned* array, int len) {
3    int i;
4    unsigned x;
5    int result = 0;
6    for (i = 0; i < len; i++) {
7      x = array[i];
8      int res=0;
9      while (x) {
10       res ^= x;             // Acumula lateralmente los bits
11       x >>= 1;              // Desplaza a la derecha para
             extraer y acumular bits
12     }
13     result += res & 0x1;      // Aplicar la máscara al
          acumular con result.
14   }
15   return result;
16 }
```

4.4. Cuarta versión

```
1  // Cuarta versión: Traduce el bucle intero por ASM
2    int parity4(unsigned* array, int len) {
3    int i,res;
4    unsigned x;
5    int result = 0;
6    for (i = 0; i < len; i++) {
7      x = array[i];
8      res = 0;
9      asm(
10       "ini3:         \n\t"    // seguir mientras que x!=0
11       "xor %[x], %[v]    \n\t"
12       "shr $1, %[x]    \n\t"    // LSB en ZF
```

```
13        "test %[x], %[x]  \n\t"
14        "jnz ini3     \n\t"   // salto, modificando CF y ZF
15        : [v]"+r"(res)              // e/s: entrada 0, salida
             paridad elemento
16
17        : [x]"r"(x)                 // entrada: valor elemento
18      );
19      result += res & 0x1;            // Aplicar la máscara al
           acumular con result.
20    }
21    return result;
22  }
```

4.5. Quinta versión

```
1  // Quinta versión: Suma en árbol
2  int parity5(unsigned* array, int len) {
3    int i, j;
4    int result = 0;
5    unsigned x;
6    for (i = 0; i < len; i++) {
7      x = array[i];
8      // Somete a cada elemento a XOR y desplazamientos
9      // sucesivos a mitdad de la distancia cada vez
10     for (j = 16; j > 0; j /= 2)
11       x ^= x >> j;
12     result += x & 0x01;      // Aplica la máscara al valor
          final.
13   }
14   return result;
15 }
```

4.6. Sexta versión

```
1  // Sexta versión: Traduce el bucle for por ASM
2  int parity6(unsigned* array, int len) {
3    int j,result = 0;;
4    unsigned x = 0;
5
6    for (j = 0; j < len; j++) {
7      x = array[j];
8      asm(
9        "mov   %[x], %%edx       \n\t" // Sacar copia para XOR
10       "shr   $16,  %%edx   \n\t"
11       "xor   %[x], %%edx   \n\t" // PF, señalando la paridad
             par de los 8 bits inferiores
12       "xor   %%dh, %%dl    \n\t"
13       "setpo  %%dl             \n\t"
14       "movzx  %%dl, %[x]    \n\t" // Devuelve en 32 bits
15       : [x] "+r" (x)                  // e/s entrada valor
             elemento, salida paridad
16       :
17       : "edx"                         // clobber
18     );
19     result += x;
20   }
21   return result;
22 }
```

4.7. Mediciones: cronometrar las distintas versiones con -O0, -O1 y -O2

```
[mario@manjario fuentes]$ make parity
cc -m32 -O0 -Wall parity.c -o parity
[mario@manjario fuentes]$ for((i=0;i<11;i++)); do echo $i ; ./parity ; done | pr -11 -l 20 -w 100


2016-11-22 19:08                                                                Página 1


0       1       2       3       4       5       6       7       8       9       10
147481  160056  159204  127820  128491  156093  158320  133220  157947  127809  128465
 60753   60704   61127   61091   61162   60744   60856   60731   60771   61252   61014
 60129   69934   61639   54701   59969   62052   54677   67878   59959   54741   54688
 18648   24705   18774   24920   18627   24587   18655   18630   18729   18760   18664
 18315   18305   18400   18479   18326   18305   18374   18315   18404   18418   18407
  4422    4418    4419    4447    4413    4426    4567    4424    4423    4428    4429
```

Figura 4.1: Ejecución del programa **parity** con **-O0**

```
[mario@manjario fuentes]$ make parity
cc -m32 -O1 -Wall parity.c -o parity
[mario@manjario fuentes]$ for((i=0;i<11;i++)); do echo $i ; ./parity ; done | pr -11 -l 20 -w 100


2016-11-22 19:12                                                                Página 1


0       1       2       3       4       5       6       7       8       9       10
 36235   35500   36717   36055   37098   37065   62567   35890   39097   62799   60118
 16784   16942   16868   16842   17561   16702   27801   21971   16768   21372   22093
 16754   16772   16752   16755   16745   16702   18860   21980   21153   19340   19686
 17186   17200   17202   17218   21459   17129   17724   18169   17130   18171   19661
  7463    7457    7440    7482   10517    7437    7444    7431    7435    7435    7512
  1360    1361    1388    1360    1800    1371    1361    1373    1415    1350    1368
```

Figura 4.2: Ejecución del programa **parity** con **-O1**

```
[mario@manjario fuentes]$ make parity
cc -m32 -O2 -Wall parity.c -o parity
[mario@manjario fuentes]$ for((i=0;i<11;i++)); do echo $i ; ./parity ; done | pr -11 -l 20 -w 100


2016-11-22 19:15                                                                Página 1


0       1       2       3       4       5       6       7       8       9       10
 69023   76429   76583   66068   76486   51333   48548   75644   50005   48569   49977
 19266   20594   20600   17901   20765   17120   28394   17180   17188   17186   17183
 17702   24955   22831   17209   19453   17276   24310   17179   17187   17172   17179
 16705   16802   16788   16691   18637   17881   20989   16738   16807   16729   16773
  4903    4821    4813    4832    5082    4812    5651    4885    4790    4808    4864
  1363    1352    1356    1353    1739    1350    1566    1366    1364    1352    1388
```

Figura 4.3: Ejecución del programa **parity** con **-O2**

4.8. Representación de los resultados

/proc/cpuinfo	Intel(R) Pentium(R) CPU 2020M @ 2.40GHz
	cache size: 2048 KB
POPCOUNT:	gcc -m32 -O<n> -Wall parity.c -o parity
	for((i=0;i<11;i++)); do echo $i ; ./parity ; done | pr -11 -l 20 -w 100

Optimización con -O0	0	1	2	3	4	5	6	7	8	9	10	Media
parity1 (lenguaje C – for)	147481	160056	159204	127820	128491	156093	158320	133220	157947	127809	128465	144082
parity2 (lenguaje C – while)	60753	60704	61127	61091	61162	60744	60856	60731	60771	61252	61014	60928
parity3 (I.CS:APP 3.22-mask final)	60129	69934	61639	54701	59969	62052	54677	67878	59959	54741	54688	60033
parity4 (Lenguaje ASM-cuerpo while)	18648	24705	18774	24920	18627	24587	18655	18630	18729	18760	18664	20336
parity5 (I.CS:APP 3.49 – 32b)	18315	18305	18400	18479	18326	18305	18374	18315	18404	18418	18407	18368
parity6 (ASM – cuerpo for)	4422	4418	4419	4447	4413	4426	4567	4424	4423	4428	4429	4438

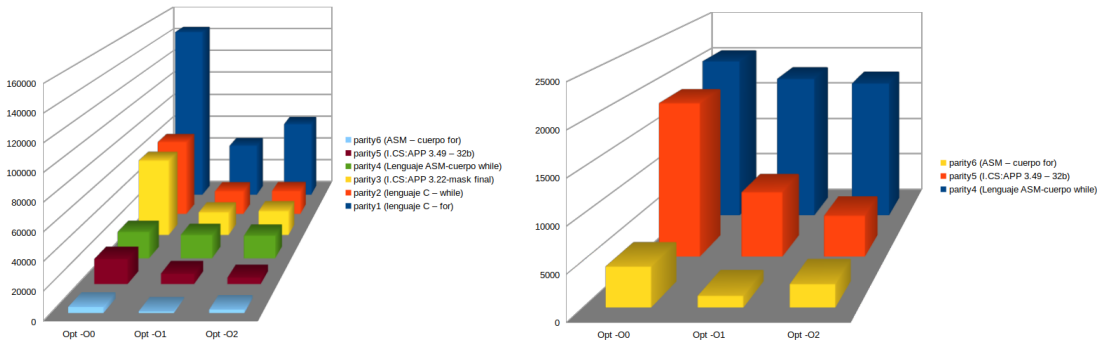
Optimización con -O1	0	1	2	3	4	5	6	7	8	9	10	Media
parity1 (lenguaje C – for)	36235	35500	36717	36055	37098	37065	62567	35890	39097	62799	60118	43558
parity2 (lenguaje C – while)	16784	16942	16868	16842	17561	16702	27801	21971	16768	21372	22093	19246
parity3 (I.CS:APP 3.22-mask final)	16754	16772	16752	16755	16745	16702	18860	21980	21153	19340	19686	18318
parity4 (Lenguaje ASM-cuerpo while)	17186	17200	17202	17218	21459	17129	17724	18169	17130	18171	19661	18023
parity5 (I.CS:APP 3.49 – 32b)	7463	7457	7440	7482	10517	7437	7444	7431	7435	7435	7512	7732
parity6 (ASM – cuerpo for)	1360	1361	1388	1360	1800	1371	1361	1373	1415	1350	136	1298

Optimización con -O2	0	1	2	3	4	5	6	7	8	9	10	Media
parity1 (lenguaje C – for)	69023	76429	76583	66068	76486	51333	48548	75644	50005	48569	49977	62606
parity2 (lenguaje C – while)	19266	20594	20600	17901	20765	17120	28394	17180	17188	17186	17183	19398
parity3 (I.CS:APP 3.22-mask final)	17702	24955	22831	17209	19453	17276	24310	17179	17187	17172	17179	19314
parity4 (Lenguaje ASM-cuerpo while)	16705	16802	16788	16691	18637	17881	20989	16738	16807	16729	16773	17413
parity5 (I.CS:APP 3.49 – 32b)	4903	4821	4813	4832	5082	4812	5651	4885	4790	4808	4864	4933
parity6 (ASM – cuerpo for)	1363	1352	1356	1353	1739	1350	1566	1366	1364	1352	13888	2550

Figura 4.4: Resultado de las ejecuciones de **parity**

POPCOUNT:	Opt -O0	Opt -O1	Opt -O2
parity1 (lenguaje C – for)	144082	43558	62606
parity2 (lenguaje C – while)	60928	19246	19398
parity3 (I.CS:APP 3.22-mask final)	60033	18318	19314
parity4 (Lenguaje ASM-cuerpo while)	20336	18023	17413
parity5 (I.CS:APP 3.49 – 32b)	18368	7732	4933
parity6 (ASM – cuerpo for)	4438	1298	2550

Figura 4.5: Resumen de los promedios



(a) Completa (b) Reducida para apreciar el factor mejora

Figura 4.6: Gráficas de las ejecuciones de **parity**