Documentación PRÁCTICA 3

Estructura de computadores

UNIVERSIDAD DE GRANADA

Fernández Mertanen, Jonathan

Índice

En esta práctica vamos a realizar la implementación de distintos algoritmos que calculan el Hamming weight o Population count que consiste en el número de 1's que aparecen en un número escrito en binario. Realizaremos una suma de los pesos de los números de una lista.

Tests

```
#ifndef TEST
#define TEST 5
#endif

#if TEST==1
    #define SIZE 4
    unsigned lista[SIZE]={0x80000000, 0x04000000, 0x00000200,
0x00000001};
    #define RESULT 4
#elif TEST==2
    #define SIZE 8
    unsigned lista[SIZE]={0x7fffffff, 0xffbfffff, 0xfffffdff,
0xfffffffe,
                          0x01000023, 0x00456700, 0x8900ab00,
0x00cd00ef};
    #define RESULT 156
#elif TEST==3
    #define SIZE 8
    unsigned lista[SIZE]={0x0 , 0x01020408, 0x35906a0c, 0x70b0d0e0,
                          0xffffffff, 0x12345678, 0x9abcdef0,
0xdeadbeef};
    #define RESULT 116
#elif TEST==4 || TEST==0
    #define NBITS 20
    #define SIZE (1<<NBITS)
    unsigned lista[SIZE];
    #define RESULT NBITS*(1 << NBITS-1)
#else
    #error "Definir TEST entre 0..4"
#endif
```

El test más usado es el número 4, el cual ofrece unos números de tiempo razonables para su manejo

Función crono

```
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("|%s:\n|\n|_____Tiempo: %9ld us\n", msg, tv_usecs);
    printf("|_____resultado = %u\n\n", resultado);
#endif
}
```

Implementaciones POPCOUNT

popcount1 (bucle for c)

```
int popcount1(unsigned *array, size_t len)
{
    size_t i;
    int result = 0;
    unsigned x;
    for(i = 0; i < len; i++){
        x = array[i];
        for(; x != 0;){
            result += x & 0x1;
            x >>= 1;
        }
    }
    return result;
}
```

popcount2 (bucle while c)

```
int popcount2(unsigned *array, size_t len)
{
    size_t i;
    int result = 0;
    unsigned x;
    for(i = 0; i < len; i++){
        x = array[i];
        while(x != 0){
            result += x & 0x1;
            x >>= 1;
        }
    }
    return result;
}
```

popcount3 (ASM)

```
int popcount3(unsigned *array, size_t len)
{
    size_t i;
    unsigned x;
    int result = 0;
    for(i = 0; i < len; i++){
        x = array[i];
        asm("\n"
            "ini3:                  \n\t"
                "shr    %[x]        \n\t"
                "adc    $0, %[r]    \n\t"
                "test   %[x], %[x]  \n\t"
                "jne    ini3"
                : [r] "+r" (result)
                : [x] "r"  (x)
        );
    }
    return result;
}
```

popcount4 (ASM CLC)

```
int popcount4(unsigned *array, size_t len)
{
    size_t i;
    int result = 0;
    unsigned x;
    for(i = 0; i < len; i++){
        x = array[i];
        asm("\n"
            "clc                    \n\t"
            "ini4:                  \n\t"
```

```
                "adc    $0, %[r]    \n\t"
                "shr    %[x]        \n\t"
                "jne    ini4        \n\t"
                "adc    $0, %[r]    \n\t"
                : [r] "+r" (result)
                : [x] "r"  (x)
        );
    }
    return result;
}
```

popcount5 (Tree)

```
int popcount5(unsigned *array, size_t len)
{
    size_t i;
    int val, result = 0;
    unsigned x;
    for(i = 0; i < len; i++){
        x = array[i];
        val = 0;
        for(size_t j=0; j<8; j++){
            val += x & 0x01010101;
            x >>= 1;
        }
        val += (val >> 16);
        val += (val >> 8);
        result += val & 0xFF;
    }
    return result;
}
```

popcount6 (Naive)

```
int popcount6(unsigned *array, size_t len)
{
    size_t i;
    int result = 0;
    unsigned x;
    const unsigned m1 = 0x55555555;
    const unsigned m2 = 0x33333333;
    const unsigned m4 = 0x0f0f0f0f;
    const unsigned m8 = 0x00ff00ff;
    const unsigned m16 = 0x0000ffff;
    for(i = 0; i < len; i++){
        x = array[i];

        x = (x & m1 ) + ((x >> 1) & m1 );
        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;
}
```

popcount7 (Naive 128bits)

```
int popcount7(unsigned *array, size_t len)
{
    size_t i;
    int result = 0;
    unsigned long x1, x2;
    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("leyendo 128b pero len no múltiplo de 4\n");

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


        x1 = (x1 & m1 ) + ((x1 >> 1) & m1 );
        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;
}
```

popcount8 (SSE3 PSHUFB 128bits)

```
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     1110 9 8    15141312
    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;
}
```

popcount9 (SSE4 POPCNT)

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

    for(i=0; i<len; i+)
    {
        x = array[i];
        asm("popcnt %[x], %[val]"

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

popcount10 (SSE4 POPCNT 128bits)

```
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 multiplo de 4\n");

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

        asm("popcnt %[x1], %[val]   \n\t"
            "popcnt %[x2], %[x1]   \n\t"
            "add %[x1], %[val]     \n\t"

            :[val] "=&r" (val)
            :  [x1] "r"  (x1),
               [x2] "r"  (x2)
        );
        result += val;
    }
    return result;
}
```