

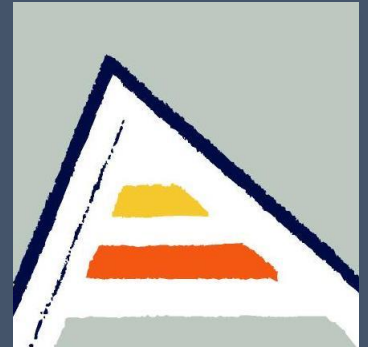
# ARQUITECTURA DE LOS COMPUTADORES

## PRÁCTICA 3

*FASE 3: Implementación de una rutina para  
comparación de arquitecturas SISD y SIMD*

### INTEGRANTES DEL GRUPO

Pavel Razgovorov | Y0888160Y  
María Rico Martínez | 48775095F (Directora)  
Eddie Rodríguez Pastor 74391601X  
Miguel Sánchez Moltó | 20096774H (Secretario)  
César Sarrión Posas | 20456867T (Controlador)



07.02.2016

Grupo: 04

# ÍNDICE

- Introducción.....Página 2
- Ejercicio seleccionado.....Página 4
- Algoritmo.....Página 6
- Ejecución del algoritmo.....Página 12
- Referencias.....Página 22

## Introducción

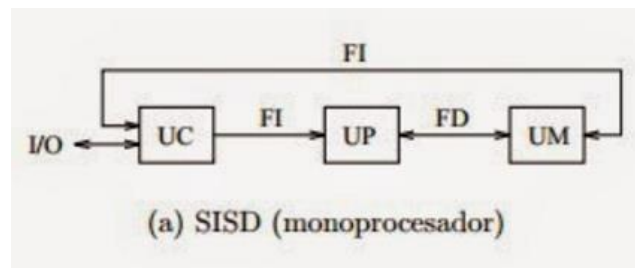
Para la correcta realización de esta fase, procederemos a utilizar dos tipos distintos de arquitectura clasificados según la “Taxionomía de Flynn”: SISD y SIMD. Esta clasificación, dependerán del número de instrucciones concurrentes y los flujos de datos disponibles que posea la arquitectura en cuestión. Así pues, de esta manera encontramos además de las dos formas empleadas anteriormente mencionadas, otras dos maneras adicionales:

- **Arquitecturas empleadas en la práctica**
- **Arquitecturas adicionalmente comentadas**

### Un dato

- **SISD (Una instrucción, un dato):**

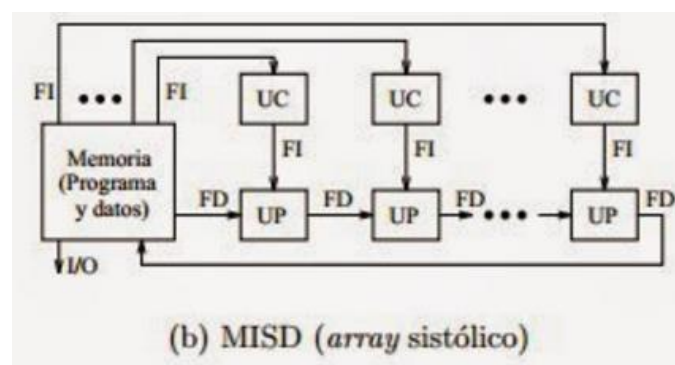
Se trata de un computador secuencial que no explota el paralelismo en las instrucciones ni en flujos de datos y, cuyo término se corresponde con la arquitectura de Von Neumann.



Un ejemplo de esta arquitectura serían las máquinas con un monoprocesador tradicional como el PC o los antiguos mainframe (computadora central).

- **MISD (Múltiples instrucciones, un dato):**

Se trata de un tipo que resulta ser poco común, puesto que la efectividad de los múltiples flujos de instrucciones, suelen precisar de múltiples flujos de datos y será principalmente utilizado en situaciones de paralelismo redundante.

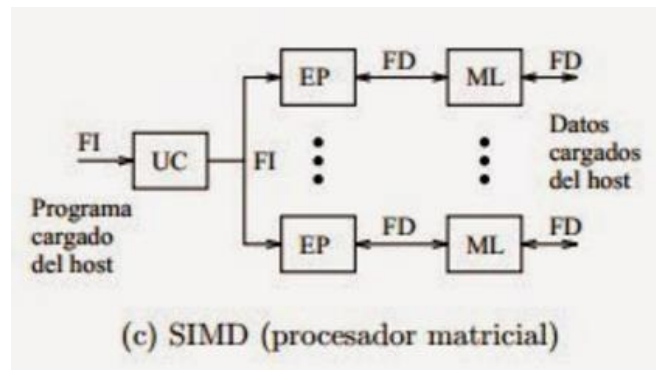


Un ejemplo de esta arquitectura sería cualquier situación en las que se necesitara de varios sistemas de respaldo, con el fin de salvaguardar el sistema en caso de que uno fallara.

## Múltiples datos

### ■ SIMD (Una instrucción, múltiples datos):

Se trata de explotar diversos flujos de datos dentro de un mismo flujo de instrucciones bajo la supervisión de una unidad de control común, cuyo fin resulta ser el de realizar operaciones que tendrán la posibilidad de ser paralelizadas de manera natural (paralelismo a nivel de datos).

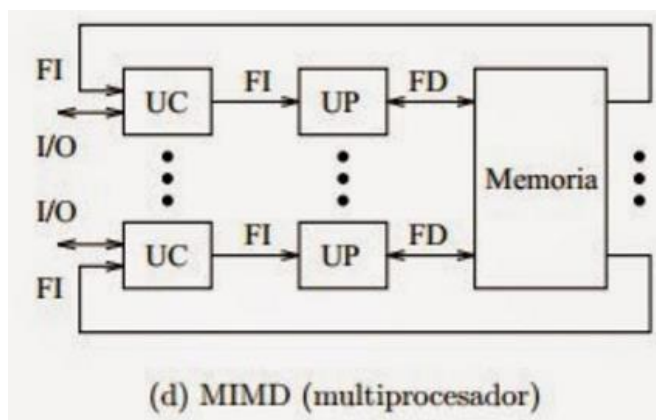


Además, su repertorio de instrucciones de aplican desde una única unidad de control recibida por todos los procesadores que operan con conjuntos distintos de datos, es decir, una misma instrucción se ejecuta simultáneamente por todas las unidades de procesamiento.

Un ejemplo de esta arquitectura sería un procesador vectorial, el cual se encarga de ejecutar operaciones matemáticas sobre múltiples datos de forma simultánea.

### ■ MIMD (Múltiples instrucciones, múltiples datos):

Se trata de varios procesadores autónomos, los cuales se encargan de ejecutar de manera simultánea instrucciones diversas sobre datos distintos.



Un ejemplo de esta arquitectura serían los sistemas distribuidos, bien sea explotando un único espacio compartido de memoria o uno distribuido.

## Ejercicio seleccionado

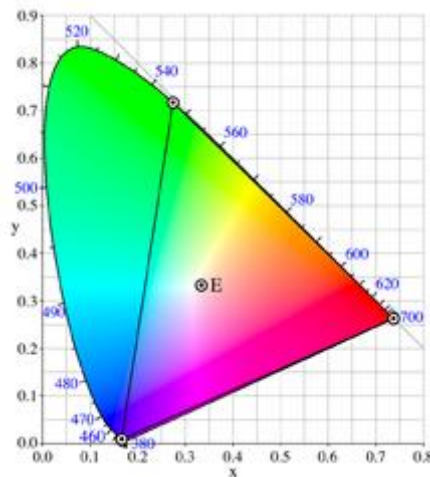
### La codificación de color RGB

Para la realización de nuestro algoritmo, hemos decidido trabajar sobre “la codificación de color RGB”.

#### ¿Qué es la codificación de color RGB?

RGB es un modelo de color basado en la síntesis aditiva. Con él, es posible representar un color mediante la mezcla por adición de los tres colores de luz primarios.

El modelo de color RGB no define por sí mismo lo que significa exactamente rojo, verde o azul, por lo que los mismos valores RGB pueden mostrar colores notablemente diferentes en distintos dispositivos que usen este modelo de color. Aunque utilicen un mismo modelo de color, sus espacios de color pueden variar considerablemente.



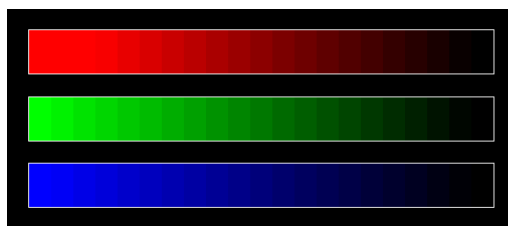
De esta manera, el color RGB se usa para mostrar imágenes en pantallas y monitores (nunca para impresión).

En el monitor de un equipo, la sensación de color se produce por la mezcla aditiva de rojo, verde y azul. Hay una serie de puntos minúsculos denominados “píxeles”, los cuales representan cada uno de los puntos de la pantalla, siendo cada píxel en sí un conjunto de tres subpíxeles (uno rojo, uno verde y uno azul), cada uno brillando con una determinada intensidad.

Esta intensidad se establece por una gama que va del 0 al 255. De esta forma cada color en modo RGB tiene tres valores, cada uno del 0-255 (por ejemplo R164 G25 B25), estableciéndose así más de 16 millones de colores diferentes.

### Funcionalidad de la propuesta

Las funcionalidades principales de nuestro algoritmo, consisten en la obtención de los colores RGB de una imagen cualquiera en formato BMP para, posteriormente, modificar el brillo y el contraste de la misma realizando una suma (el brillo se incrementa) o una resta (el brillo disminuye) a cada uno de estos tres colores RGB (rojo, verde y azul).



Así pues, el funcionamiento del algoritmo resulta ser el siguiente:

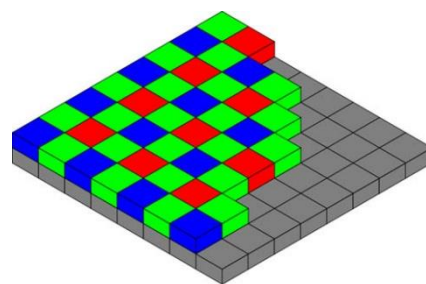
En primer lugar, mediante el uso de la librería "**CImg.h**", procederemos a la apertura de la imagen, obteniendo a la par un array con todos y cada uno de los píxeles de la misma.

Se recibe un porcentaje con rango establecido entre los valores -100 y 100, mediante la fórmula "**increment = (255 \* percent) / 100**", el cual se transformará posteriormente en el incremento a aplicar en el RGB de la imagen.

De esta manera, si por ejemplo el porcentaje recibo resulta ser del 10%, aplicando el mismo al valor 255, obtendremos un incremento de 25.

Este resultado, posteriormente será sumado al valor de cada uno de los píxeles anteriormente recogidos, siempre controlando y/o evitando el desbordamiento, pues debe estar comprendido entre 0 y 255.

Seguidamente, se prosigue calculando el número total de píxeles que posee la imagen tratada. Puesto que cada píxel en sí, resulta ser un conjunto de tres subpíxeles (uno rojo, uno verde y uno azul), procederemos a multiplicar el total de píxeles obtenidos tras el producto entre el ancho y el alto de la imagen (matriz de píxeles), por la cantidad de subpíxeles que posee cada uno de los mismos (3), obteniendo de esta manera el total final.



### Edición brillo

De igual forma que en el apartado anterior, tras los cálculos y consideraciones anteriores sobre el rango de valores y el incremento a aplicar, procedemos a calcular el valor RGB del nuevo píxel.

El valor de brillo por lo general se encontrará en el rango de -255 a 255 para una paleta de 24 bits. Los valores negativos oscurecerán la imagen y, a la inversa, los valores positivos aclararán la imagen.

Así pues, en el caso de que se diera un desbordamiento de brillo, se seleccionaría el color blanco, mientras que si se diera una insuficiencia de brillo, quedaría seleccionado el color negro, ya que de otro modo entraría en los valores normales establecidos.

### Edición contraste

En esta función, además de haber realizado las consideraciones y cálculos anteriormente mencionados, se procederá al cálculo del factor del nuevo contraste para, posteriormente, aplicarlo en la fórmula del ajuste.

La primera etapa es calcular un factor de corrección de contraste que se da por la siguiente fórmula:

$$F = \frac{259(C + 255)}{255(259 - C)}$$

Para que el algoritmo funcione correctamente el valor para el factor de corrección de contraste ( F ) necesita ser almacenada como un número de coma flotante y no como un número entero. El valor C en la fórmula indica el nivel deseado de contraste.

El siguiente paso es para realizar el ajuste de contraste en sí. La siguiente fórmula muestra el ajuste de contraste que se realiza para el componente rojo de un color y, procediendo de igual forma para el resto de colores componentes.

$$R' = F(R - 128) + 128$$

## Algoritmo

Los algoritmos principales de nuestro programa encargados de la edición del brillo y del contraste, aparecen organizados en dos funciones principales: “EditContrast” y “EditBrightness”. Así pues, los códigos para cada uno de los lenguajes empleados quedarían de la siguiente manera:

- **Código en C**

La implementación en el lenguaje C es la más sencilla de todas y, además de ello, resulta ser la base principal empleada para las otras dos con ensamblador. Dado que es un lenguaje de alto nivel, las operaciones aritméticas son mucho más sencillas de implementar enfocándolo desde el ámbito algorítmico-matemático.

```
void EditConstrastC(int percent) {
    int increment, length;
    float factor, newPixel;

    increment = (255 * percent) / 100; //Calculamos el incremento a partir de
                                        //un porcentaje [-100,100], haciendo
                                        //que el incremento sea [-255,255]

    length = width * height * 3;

    //Formula del factor para el nuevo contraste
    factor = (float)(259 * (increment + 255)) / (255 * (259 - increment));

    for (int i = 0; i < length; ++i) {
        newPixel = factor * (pixels[i] - 128) + 128; //Formula del ajuste
        if (newPixel > 255) pixels[i] = 255;

        else if (newPixel < 0) pixels[i] = 0;
        else pixels[i] = (unsigned char)newPixel;
    }
}

//Editar el brillo usando el array pixels, que son para implementar en ASM y
luego en SSE
void EditBrightnessC(int percent) {
```

```

int increment, length, newPixel;

//Los valores RGB van del 0 al 255. El incremento se realizará por cada
//píxel P.E. Si tengo un porcentaje de brillo de -10%, haremos un
//incremento de -25'5 (25)
increment = (255 * percent) / 100;

length = width * height * 3; //ancho * alto = matriz de pixeles y 3 = cada
                             //color del RGB

for (int i = 0; i < length; ++i) {
    newPixel = (int)pixels[i] + increment; //Calculamos el valor RGB
                                           //del nuevo pixel

    if (newPixel > 255) pixels[i] = 255; //Si hay overflow de brillo,
                                           //me quedo en el blanco

    else if (newPixel < 0) pixels[i] = 0; //Si hay underflow de brillo,
                                           //me quedo en el negro

    else pixels[i] = newPixel; //Si no, entra en los valores normales
}
}

```

- **Código en ASM**

En contraposición con el anterior lenguaje empleado, ensamblador resulta ser un lenguaje de bajo nivel, cuya representación principal se basa en las instrucciones, los registros del procesador y las posiciones de memoria, siendo de esta manera un lenguaje con mayor complejidad.

```

void EditBrightnessASM(int percent) {
    int increment, tam;
    _asm {

        //increment = (255 * percent) / 100;

        imul eax, percent, 255; //Signed Multiplication percent*255 y lo
                                //guardamos en EAX

        cdq; //Extiende el signo negativo a los registros EDX:EAX para que
              //podamos dividir con signo

        mov ecx, 100; //Movemos el inmediato 100 a un registro porque la
                      //division no permite dividir con immediatos

        idiv ecx; //Dividimos EAX entre 100 y el resultado va a EAX

        mov increment, eax; //Copiamos el resultado de toda esta operacion
                             //aritmética en la variable increment

        //length = width * height * 3;

        mov esi, this; //Para acceder a los miembros de la clase se hace
    }
}

```



```

        //con la direccion this + miembro

        mov ebx, [esi + width];
        imul ebx, [esi + height];
        imul ebx, 3;
        mov tam, ebx; //Se guarda el resultado final en la variable tam

        mov esi, [esi + pixels]; //Cambiamos el puntero de this a
                                //this->pixels. Hacer [esi + pixels
                                // + edx] no funciona

        mov edx, 0; //EDX = Indice del bucle

bucle:
        cmp edx, tam;

        je fin_bucle; //Si son iguales salta al fin_bucle

        movzx edi, [esi + edx]; //Movemos a edi pixels[i]
        mov ecx, increment; //ECX = increment
        add ecx, edi; //ECX += pixels[i]
        cmp ecx, 255; //Comprobamos overflow
        jg blanco; //Si es mayor, truncamos a blanco
        cmp ecx, 0; //Comprobamos underflow
        jl negro; //Si es menor, truncamos a negro
        jmp nuevoColor; //En cualquier otro caso, vamos a nuevo color

blanco:
        mov ecx, 255;
        jmp nuevoColor;

negro:
        mov ecx, 0;
        jmp nuevoColor;

nuevoColor:
        mov [esi + edx], cl; //Mover a pixels[i] la parte baja del
                            //registro ECX

        inc edx;
        jmp bucle;

fin_bucle:
}
}

void EditContrastASM(int percent) {
    int increment, tam, factorTemp, aux, number;
    float factor, newPixel;

    _asm {
        //increment = (255 * percent) / 100;

        imul eax, percent, 255; //Signed Multiplication percent*255 y lo
                                //guardamos en EAX

        cdq; //Extiende el signo negativo a los registros EDX:EAX para que
            //podamos dividir con signo

        mov ecx, 100; //Movemos el inmediato 100 a un registro porque la
            //division no permite dividir con immediatos
    }
}

```

```

    idiv ecx; //Dividimos EAX entre 100 y el resultado va a EAX

    mov increment, eax; //Copiamos el resultado de toda esta operación
                        //aritmética en la variable increment

    //length = width * height * 3;

    mov esi, this; //Para acceder a los miembros de la clase se hace
                  //con la direccion this + miembro

    mov ebx, [esi + width];
    imul ebx, [esi + height];
    imul ebx, 3;
    mov tam, ebx; //Se guarda el resultado final en la variable tam

    //factor = (float)(259 * (increment + 255)) / (255 * (259 -
    //increment))
    add eax, 255; //Sumar incremento al 255 se guarda en eax
    mov ebx, 259;
    sub ebx, increment; //Restar 259 a incremento se guarda en ebx
    mov ecx, 259;
    imul ecx, eax; //(259 * (increment + 255)) se guarda en ecx
    mov edx, 255;
    imul edx, ebx; //(255 * (259 - increment)) se guarda en edx

    mov factorTemp, ecx;
    fild factorTemp;
    mov factorTemp, edx;
    fild factorTemp;
    fdivp st(1), st;
    fstp factor;

    mov eax, 0;
    mov ebx, tam;

    mov esi, [esi + pixels]; //Cambiamos el puntero de this a
                            //this->pixels. Hacer [esi + pixels +
                            //edx] no funciona
bucle:
    fstp aux;
    fstp aux;
    fstp aux;
    cmp eax, ebx; //Compara tam y la i
    je fin_bucle; //Si son iguales salta al final

    //newPixel = factor * (pixels[i] - 128) + 128;

    movzx ecx, [esi + eax]; //Movemos a ecx pixels[i]
    sub ecx, 128; //Restamos 128 a pixels[i] y se guarda en ecx

    fld factor;
    mov aux, ecx;
    fild aux;
    fmul st(0), st(1);

    //fst newPixel;

    mov number, 128;
    fild number;
    fadd st(0), st(1);
    fisttp newPixel;

```

```

        mov ecx, newPixel;

        cmp ecx, 255; //Comprobamos overflow
        jg blanco; //Si es mayor, truncamos a blanco
        cmp ecx, 0; //Comprobamos undeflow

        jl negro; //Si es menor, truncamos a negro
        jmp nuevoColor; //En cualquier otro caso, vamos a nuevo color

blanco:
    mov ecx, 255;
    jmp nuevoColor;

negro:
    mov ecx, 0;
    jmp nuevoColor;

nuevoColor:

    mov[esi + eax], cl; //Mover a pixels[i] la parte baja del registro
                        //ECX

    inc eax;
    jmp bucle;

fin_bucle:
}
}
};

```

- **Código en SSE**

De la misma forma que con el algoritmo anterior, al tratarse también de un programa en ensamblador, se ha dividido el programa en diferentes funciones, pues ello nos facilitará la tarea a la hora de trabajar con este tipo de ensamblador.

```

void EditBrightnessSSE(int percent) {

    int increment;

    int tam = width * height * 3;

    int numBytes = sizeof(unsigned char) * tam;

    unsigned char* aux = (unsigned char*)_aligned_malloc(numBytes, 16);

    for (int i = 0; i < tam; i++) {

        aux[i] = pixels[i];

    }

    unsigned char* incr = (unsigned char*)_aligned_malloc(16, 16);

    int signo = 0;

```

```

_asm {

    //increment = (255 * percent) / 100;
    imul eax, percent, 255; //Signed Multiplication percent*255 y lo
                            //guardamos en EAX

    cdq; //Extiende el signo negativo a los registros EDX:EAX para que
        //podamos dividir con signo

    mov ecx, 100; //Movemos el inmediato 100 a un registro porque la
                //division no permite dividir con immediatos

    idiv ecx; //Dividimos EAX entre 100 y el resultado va a EAX

    mov increment, eax; //Copiamos el resultado de toda esta operación
                    //aritmética en la variable increment

    mov edx, 0; // en el bucle, i=0
    mov edi, incr; // movemos el puntero de incr a edi

    cmp eax, 0; //comparamos si nuestro incremento es positivo
    jg cargarIncremento; //si es asi dejamos la variable signo a 0
    mov signo, 1; //si no lo es, signo=1
    imul eax, eax, -1; // y hacemos el incremento positivo

    cargarIncremento: //cargamos 16 bytes con el incremento en el
                    //vector auxiliar incr

    cmp edx, 16; // mientras i<16
    je cont; //saltamos si i=16
    mov[edi + edx], eax; // incr[i]=incremento;
    inc edx; // i++
    jmp cargarIncremento;

cont:

    movapd xmm1, [edi]; //cargamos nuestro incrementox16 en xmm1

    //length = width * height * 3;

    mov esi, this; //Para acceder a los miembros de la clase se hace
                    //con la direccion this + miembro

    mov ebx, [esi + width];
    imul ebx, [esi + height];
    imul ebx, 3;
    mov tam, ebx; //Se guarda el resultado final en la variable tam

    mov esi, [aux]; // Copiamos el puntero a aux en esi
    mov edx, 0; //EDX = Indice del bucle

    mov eax, signo; //movemos el contenido de signo a eax

bucle:
    cmp edx, tam; //Comparamos si edx es < tam
    jge fin_bucle; //Si son iguales salta al fin_bucle

    movapd xmm0, [esi + edx]; //Movemos a xmm0 las i+15 posiciones
                            //pixels[i]

```

```

        //mov ecx, increment; //ECX = increment

        cmp eax, 1; //comparamos el signo del incremento
        je resta; //si es negativo restamos, en caso contrario sumamos

suma:
    paddusb xmm0, xmm1; //xmm0 = pixeles + incremento
    jmp continuarBucle;

resta:
    psubusb xmm0, xmm1; //xmm0 = pixeles - incremento
    jmp continuarBucle;

continuarBucle:

    movapd[esi + edx], xmm0; //Mover a pixels[i] la parte baja del
                                //registro ECX

    add edx, 16;
    jmp bucle;

fin_bucle:
}

for (int i = 0; i < tam; i++) {
    pixels[i] = aux[i];
}
}

void EditConstrastSSE(int percent) {

    int increment, tam, factorTemp, tmp;
    float factor;

    float* factorArray = new float[4];
    float* unoDosOcho = new float[4];

    float* newPixelArray = new float[width * height * 3];
    for (int i = 0; i < width * height * 3; i++) newPixelArray[i] = pixels[i];

    for (int i = 0; i < 4; i++) unoDosOcho[i] = 128;

    _asm {

        imul eax, percent, 255; //Signed Multiplication percent*255 y lo
                                //guardamos en EAX
        cdq; //Extiende el signo negativo a los registros EDX:EAX para que
            //podamos dividir con signo

        mov ecx, 100; //Movemos el inmediato 100 a un registro porque la
            //division no permite dividir con immediatos

        idiv ecx; //Dividimos EAX entre 100 y el resultado va a EAX

        mov increment, eax; //Copiamos el resultado de toda esta operacion
            //aritmética en la variable increment

        mov esi, this; //Para acceder a los miembros de la clase se hace
            //con la direccion this + miembro

        mov ebx, [esi + width];
        imul ebx, [esi + height];
    }
}

```

```

    imul ebx, 3;
    mov tam, ebx; //Se guarda el resultado final en la variable tam

    add eax, 255; //Sumar incremento al 255 se guarda en eax
    mov ebx, 259;
    sub ebx, increment; //Restar 259 a incremento se guarda en ebx
    mov ecx, 259;
    imul ecx, eax; //(259 * (increment + 255)) se guarda en ecx
    mov edx, 255;
    imul edx, ebx; //(255 * (259 - increment)) se guarda en edx
    mov factorTemp, ecx;
    fild factorTemp;
    mov factorTemp, edx;
    fild factorTemp;
    fdivp st(1), st;
    fstp factor;

    mov eax, 0;
    mov ebx, factorArray;
    mov ecx, factor;

bucleFactor: //rellenamos un array de floats con 4 factores para trabajar
              //con ellos
    mov [ebx + eax*4], ecx; //Array[i] = factor;
    inc eax; //i++
    cmp eax, 4; //i<4
    jnl bucleFactor;

    mov eax, 0;
    mov ebx, tam;
    mov esi, newPixelArray; //Ponemos el puntero a newPixelArray
    mov ecx, unoDosOcho;
    movups xmm0, [ecx]; //xmm0 almacena 4 valores 128
    mov ecx, factorArray;
    movups xmm1, [ecx]; //xmm1 almacena 4 factores

    add ebx, -4; //Restamos 4 para evitar salirnos

bucle:
    movups xmm2, [esi + eax * 4]; //muevo pixeles a xmm3
    subps xmm2, xmm0; //-128
    mulps xmm2, xmm1; //*factor
    addps xmm2, xmm0; //+128
    movups[esi + eax * 4], xmm2; //Guardamos el resultado

    add eax, 4; //Vamos de 4 en 4 asi que i+=4
    cmp eax, ebx; //Si i<tam, saltamos
    jnl bucle;

    // continuamos
    mov ebx, tam; //i<tam
    add ebx, -4; //Evitamos salirnos
    mov eax, 0; // i=0
    mov edi, this;
    mov edi, [edi + pixels]; //Copiamos el puntero a pixels
    finit; //Iniciamos la pila de la FPU

comprobar_overflow:
    cmp eax, ebx;
    jge fin_bucle; //i<tam

```

```

        fld [esi + eax * 4]; //Cargamos el float
        fistp tmp; //Y lo sacamos en forma de int
        mov ecx, tmp; //Lo movemos a un registro

        cmp ecx, 255; //Si es mayor de 255, pondremos 255
        jg overflow;

        cmp ecx, 0; //Si es menor de 0, pondremos 0
        jl underflow;

        mov [edi + eax], ecx; //Si esta entre 0 y 255 lo almacenamos
                                //directamente

        inc eax;
        jmp comprobar_overflow;

overflow:
        mov[edi + eax], 255; //Almacenamos un 255 en su lugar
        inc eax; //i++
        jmp comprobar_overflow; //Continuamos con el bucle

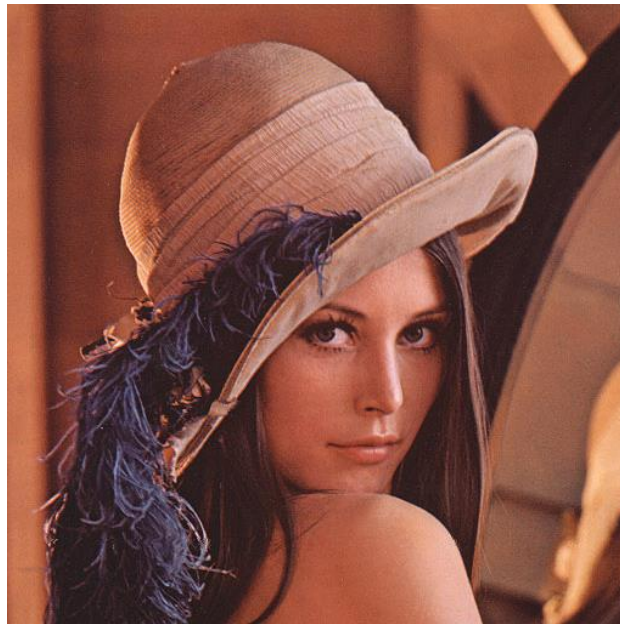
underflow:
        mov[edi + eax], 0; //Almacenamos un 0 en su lugar
        inc eax; //i++
        jmp comprobar_overflow; //Continuamos con el bucle
fin_bucle:
    }
}

```

## Ejecución del algoritmo

Una vez implementado el algoritmo principal, procederemos a probar la ejecución del programa para cada uno de los lenguajes empleados (C, ASM y SSE):

En primer lugar, realizaremos una modificación sobre el contraste de la imagen seleccionada para este ejemplo, la cual resulta ser una ilustración muy conocida con el nombre de Lenna (o Lena), pues es una de las imágenes de prueba estándar más utilizada para los algoritmos de compresión.



Para ello, realizaremos las siguientes llamadas a los módulos en lenguaje C y ASM, que aparecen definidos para editar este componente de la imagen, especificando además el porcentaje a aplicar para dicha modificación:

- **Llamada a función en C**

```
int percent = 40;  
img3.EditContrastC(percent);  
img3.save("pruebaContrasteC.bmp");
```

- **Llamada a función en ASM**

```
int percent = -25;  
img2.EditContrastASM(percent);  
img2.save("pruebaContrasteASM.bmp");
```

Tras ello, se generarán los respectivos archivos de salida en formato “.bmp”, con la imagen inicial ya finalmente modificada según los valores que hayamos especificado para el contraste de la misma.



pruebaContrasteC.bmp



Tal y como podemos observar, al haber especificado porcentajes tan diferentes entre sí, el nivel de contraste es notablemente distinto entre ambas imágenes, pues la prueba realizada en ASM, al presentar un menor porcentaje, nos genera un aspecto en la imagen grisáceo y/o apagado.

pruebaContrasteASM.bmp



En cambio, la misma imagen modificada en C con mayor porcentaje, presenta colores más vivos e intensificados.

En segundo lugar, procederemos a llevar a cabo modificaciones sobre el brillo de la imagen anteriormente seleccionada como ejemplo.

Mediante la función implementada en los lenguajes C, ASM y SSE, procederemos pues a modificar este otro componente de la imagen. Para ello, se realizarán las siguientes llamadas a función, especificando nuevamente el porcentaje con el cual queremos modificar dicho valor:

- **Llamada a función en C**

```
int percent = 20;  
img3.EditBrightnessC(percent);  
img3.save("pruebaBrilloC.bmp");
```

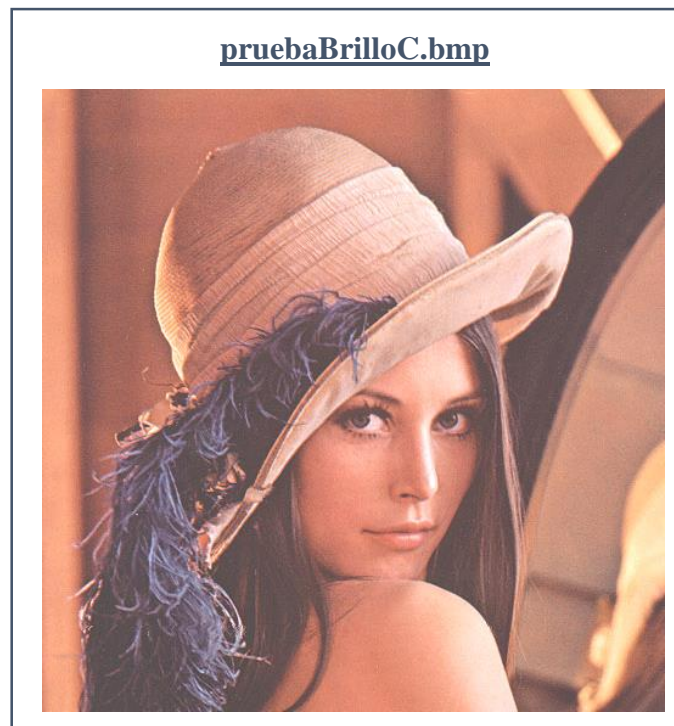
- **Llamada a función en ASM**

```
int percent = 50;  
img3.EditBrightnessASM(percent);  
img3.save("pruebaBrilloASM.bmp");
```

- **Llamada a función en SSE**

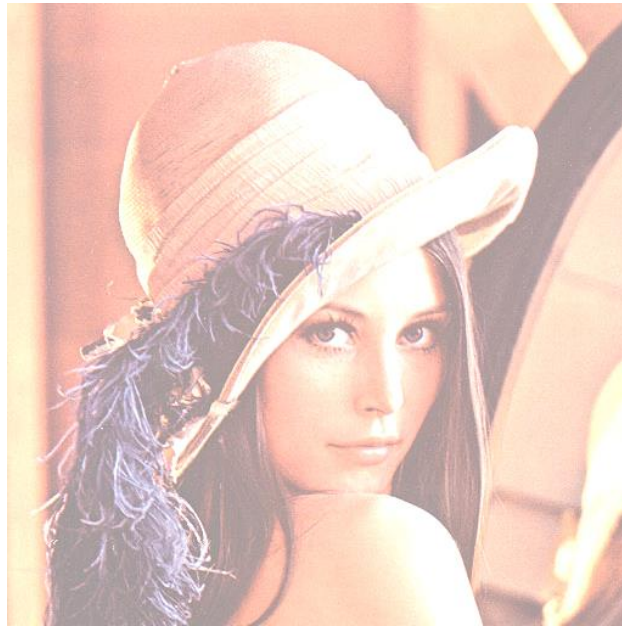
```
int percent = -30;  
img2.EditBrightnessSSE(percent);  
img2.save("pruebaBrilloSSE.bmp");
```

De igual manera que en el ejemplo anterior, se generarán los respectivos archivos de salida en formato “.bmp” con la imagen inicial ya finalmente modificada según los valores que hayamos especificado para el brillo de la misma.



En cambio, la misma imagen modificada en C con un porcentaje algo inferior al anterior, presenta una luminosidad más suave y similar a la imagen original.

**pruebaBrilloASM.bmp**



Tal y como podemos observar, al haber especificado porcentajes tan diferentes entre sí, el nivel de brillo es notablemente distinto entre las tres imágenes, pues la prueba realizada en ASM, al presentar un mayor porcentaje, nos genera un aspecto en la imagen con una destacable luminosidad.

**pruebaBrilloSSE.bmp**



Finalmente, en tercer lugar podemos observar que la prueba realizada en SSE, resulta ser totalmente opuesta a las dos anteriores, pues al presentar un porcentaje bastante inferior, presenta una oscuridad muy notable en la misma.

## Comparativa entre lenguajes y conclusiones

El algoritmo anteriormente comentado se ha implementado en tres lenguajes diferentes C, ensamblador 8086 y SSE.

Así pues, a continuación, procederemos a realizar una comparativa sobre el tiempo de ejecución que han experimentado tres equipos escogidos, distintos entre sí, para el algoritmo diseñado encargado de la edición del brillo,, en cada uno de los lenguajes anteriormente mencionados, modificando una imagen de amplias dimensiones y/o resolución.

De esta manera, las características de los equipos empleados son las siguientes:



### PC 1

- Sistema operativo: Windows 10 ~ 32 bits
- Procesador: Intel® Core™ 2 Duo @ 2.16 GHz
- Número de núcleos: 2
- Memoria RAM: 4GB



### PC 2


- Sistema operativo: Windows 10 ~ 64 bits
- Procesador: Intel® Core™ i5-3317U @ 1.70 GHz
- Número de núcleos: 2
- Memoria RAM: 4GB



### PC 3

- Sistema operativo: Windows 8.1 ~ 64 bits
- Procesador: Intel® Core™ i7-4700HQ @ 2.40 GHz
- Número de núcleos: 8
- Memoria RAM: 8GB

Para realizar esta comprobación, procederemos a ejecutar primeramente en cada equipo un ejecutable generado, el cual nos permitirá obtener dichos resultados.

 [Vectorization] Images Editor.exe

Un ejemplo de ejecución para esta comprobación, sería la siguiente:

```
C = 329 ms
ASM = 131 ms
SSE = 13 ms
INC C/ASM = 151.145 %
INC C/SSE = 2430.77 %
INC ASM/SSE = 907.692 %
Presione una tecla para continuar . . .
```

Así pues, tal y como se observa en la imagen, además de obtener el coste temporal que ha supuesto la ejecución de nuestro algoritmo para cada uno de los lenguajes, obtenemos además las ganancias obtenidas dos a dos entre los mismos.

### Tiempo de ejecución

Siguiendo este mismo procedimiento con cada uno de los tres equipos a testear, los resultados obtenidos de tiempo de ejecución para el algoritmo encargado de la edición del brillo de la imagen, resultan ser los siguientes:

Equipos	Algoritmo en C (ms)	Algoritmo en ASM (ms)	Algoritmo en SSE (ms)
PC 1	850	478	13
PC 2	1666	1017	64
PC 3	329	131	13

De esta manera, la gráfica comparativa para el tiempo de ejecución resulta ser la siguiente:



Tras observar la gráfica generada, podemos concluir claramente que el algoritmo más rápido resulta ser el implementado en Ensamblador con instrucciones SSE, pues presenta una importante optimización.

Seguido del mismo, encontramos al lenguaje Ensamblador 8086, y, finalmente, el que presenta una mayor lentitud resulta ser el desarrollado en C (Lenguaje de alto nivel).

El algoritmo en C es el más sencillo de implementar, ya que nos ofrece una mayor comodidad al resultar un lenguaje de alto nivel.

En pocas líneas de código hemos podido llevar a cabo la tarea de modificar los valores de la imagen, mientras que en los algoritmos desarrollados en ensamblador (tanto 8086 como en SSE) el tamaño del código aumenta considerablemente, aunque al precio de un menor rendimiento.

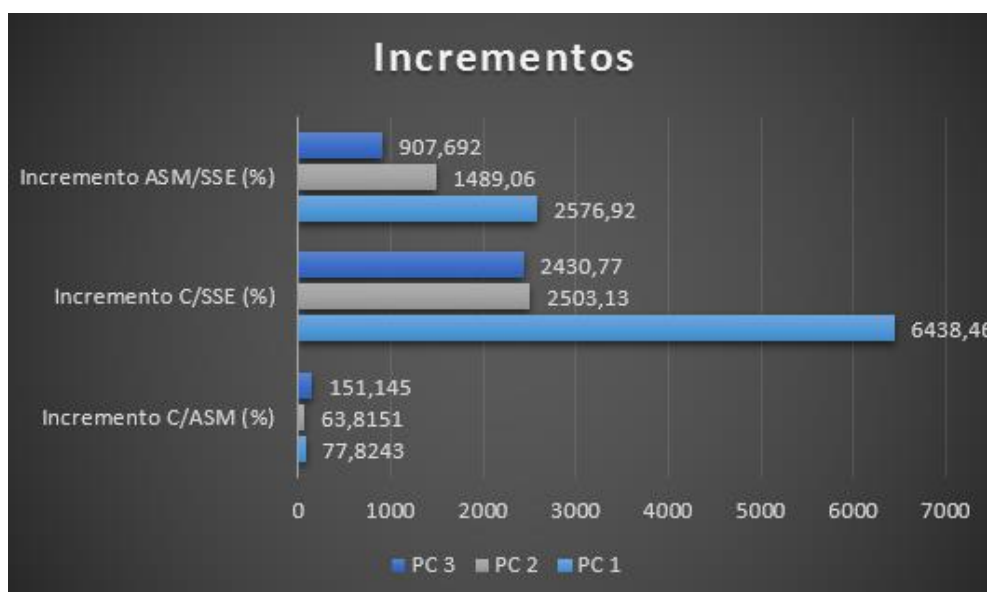


## Ganancia

Así pues, las ganancias en la velocidad de ejecución (programación paralela) obtenidas entre lenguajes para el algoritmo encargado de la edición del brillo de la imagen, quedarían resumidas en la siguiente tabla:

Equipos	Incremento C/ASM (%)	Incremento C/SSE (%)	Incremento ASM/SSE (%)
PC 1	77,8243	6438,46	2576,92
PC 2	63,8151	2503,13	1489,06
PC 3	151,145	2430,77	907,692

De esta manera, la gráfica para dichos resultados resulta ser la siguiente:



Tras ello, podemos observar que el mayor incremento experimentado resulta darse entre los lenguajes C y SSE. Seguido a este, encontramos el incremento propiciado entre los lenguajes ASM y SSE y, finalmente, en último lugar, encontramos el resultado entre los lenguajes C y ASM, el cual presenta un incremento ínfimo entre ambos.

Por ejemplo, si analizamos los resultados obtenidos para el equipo número 1, el incremento entre C y SSE resulta ser de un 6438,46%, mientras que para el mismo equipo pero calculando la ganancia entre C y ASM, el porcentaje tan sólo llega a un 77,82%, estando así 83 veces por debajo de la ganancia mayor.

## Conclusión

Así pues, podemos concluir afirmando que para la obtención de una mayor velocidad de ejecución en un equipo, se ha de optar y/o apostar por una mayor paralelización, ya que cuantos más procesadores se empleen en realizar una misma tarea, menor será el tiempo de ejecución requerida.

## **Referencias**

### ■ Taxionomía de Flynn

[https://es.wikipedia.org/wiki/Taxonom%C3%ADa\\_de\\_Flynn](https://es.wikipedia.org/wiki/Taxonom%C3%ADa_de_Flynn)

<http://2013.es.pycon.org/media/programacion-paralela.pdf>

### ■ Color RGB

<https://es.wikipedia.org/wiki/RGB>

### ■ Ajuste brillo

<http://www.dfstudios.co.uk/articles/programming/image-programming-algorithms/image-processing-algorithms-part-4-brightness-adjustment/>

### ■ Ajuste contraste

<http://www.dfstudios.co.uk/articles/programming/image-programming-algorithms/image-processing-algorithms-part-5-contrast-adjustment/>

### ■ Imagen Lena

<http://www.cs.cmu.edu/~chuck/lennapg/lenna.shtml>

