

Organización del Computador II  
TP2  
Grupo A Nightmare on Elm Street / Family Game

15 de julio de 2015

| Integrante      | LU     | Correo electrónico       |
|-----------------|--------|--------------------------|
| Federico Beuter | 827/13 | federicobeuter@gmail.com |
| Juan Rinaudo    | 864/13 | jangamesdev@gmail.com    |
| Mauro Cherubini | 835/13 | cheru.mf@gmail.com       |

**Reservado para la cátedra**

| Instancia       | Docente | Nota |
|-----------------|---------|------|
| Primera entrega |         |      |
| Segunda entrega |         |      |

# Índice

|                                 |           |
|---------------------------------|-----------|
| <b>1. Introducción</b>          | <b>3</b>  |
| <b>2. Filtro 1: Blur</b>        | <b>4</b>  |
| 2.1. Cambios . . . . .          | 4         |
| 2.2. Explicacion . . . . .      | 4         |
| 2.3. Implementacion 1 . . . . . | 4         |
| 2.4. Implementacion 2 . . . . . | 5         |
| 2.5. Resultados . . . . .       | 8         |
| 2.6. Conclusion . . . . .       | 9         |
| <b>3. Filtro 2: Merge</b>       | <b>10</b> |
| 3.1. Cambios . . . . .          | 10        |
| 3.2. Explicacion . . . . .      | 10        |
| 3.3. Implementacion 1 . . . . . | 10        |
| 3.4. Implementacion 2 . . . . . | 11        |
| 3.5. Resultados . . . . .       | 12        |
| 3.6. Conclusion . . . . .       | 13        |
| <b>4. Filtro 3: HSL</b>         | <b>14</b> |
| 4.1. Explicacion . . . . .      | 14        |
| 4.2. Implementacion 1 . . . . . | 14        |
| 4.3. Implementacion 2 . . . . . | 15        |
| 4.3.1. RGB a HSL . . . . .      | 15        |
| 4.3.2. HSL a RGB . . . . .      | 17        |
| 4.4. Resultados . . . . .       | 20        |
| 4.5. Conclusion . . . . .       | 21        |

## 1. Introducción

El objetivo del trabajo practico es utilizar el set de instrucciones SIMD para el procesamiento de imagenes. Para esto se nos pidio implementar 2 versiones en ASM (x86\_64) de 3 filtros diferentes (Blur, Merge, HSL), ademas se nos brindo una version en C para usar como guia.

Es importante destacar que las imagenes que vamos a procesar son multiplo de 4 pixeles y con un tamaño minimo de 16 pixeles, esto nos permite en nuestras implementaciones cargar de a 4 pixeles en los registros XMM sin tener que preocuparnos por los casos borde. Ademas el procesamiento de las imagenes se hace unicamente usando instrucciones SSE y durante el procesamiento de los mismos tratamos de mantenernos adentro de los margenes de error brindados por los test de la catedra.

Una vez implementado los 3 filtros y sus diferentes versiones se iniciara con la etapa de experimentacion, donde buscamos responder las preguntas brindadas por la catedra y formar un mayor entendimiento de los algoritmos implementados para sacar nuestras propias conclusiones.

## 2. Filtro 1: Blur

### 2.1. Cambios

Para el recuperatorio decidimos hacer cambios en el código.

La lista de cambios esta a continuacion, con la aclaracion de por que se hicieron los mismos:

- Se remplazaron las operaciones que realizaban las copias por operaciones de SIMD para aumentar la velocidad a la que se copian los pixeles, tambien se modifico el código el cual antes copiaba las dos filas de pixeles superiores para que solo copie una (Swapeando los punteros de R13 con R12 y luego solo copiando en R13 los datos de la fila de pixeles del medio).

### 2.2. Explicacion

El filtro *blur* consiste en para cada pixel (exceptuando los ubicados en el borde), tomar sus 8 vecinos y hacer un promedio de cada uno de sus componentes R, G y B entre los 9 pixeles, este nuevo valor reemplaza los que teniamos en el pixel actual. El promedio debe ser calculado respecto a los datos originales, es decir, que si al procesar un pixel y alguno de los vecinos ya fue procesado, debemos utilizar los datos del mismo antes de la modificacion. Nuestro algoritmo se basara en el implementado por la catedra en C pero aprovechando de las operaciones vectoriales de SIMD para mejorar su performance tratando de trabajar con la mayor cantidad de datos al mismo tiempo.

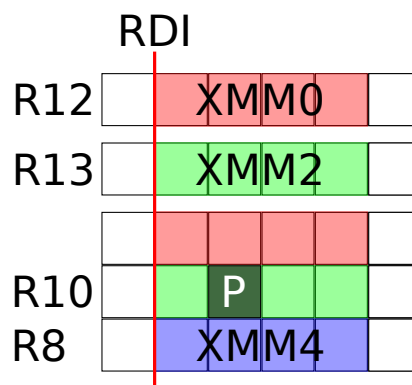
### 2.3. Implementacion 1

La primera implementacion nos pide trabajar de a 1 pixel por iteracion.

Los pixeles estan compuestos por 4 bytes: A R G B, esto nos permite cargar 4 pixeles en un registro XMM, como nosotros necesitamos 9 pixeles, ubicados de a 3 en 3 filas diferentes vamos a precisar 3 registros XMM para cargarlos (Para esto usamos los registros XMM0, XMM2 y XMM4). Ademas se utilizaron los siguientes 6 registro de proposito general:

- RDI que lo usamos para iterar sobre el eje X
- R9 que lo usamos para iterar sobre Y
- R12 que es un puntero a la copia de la fila superior
- R13 que es un puntero a la copia de la fila actual
- R8 es un puntero a la fila inferior
- R10 es un puntero a la fila actual de la imagen (fila que estoy modificando)

Esto se puede visualizar en el siguiente grafico:



Donde P es el pixel a procesar.

Antes de comenzar el ciclo inicializamos RDI en 0, R9 en 2, movimos R8 a R10, aumentamos R8 en una fila y posicionamos el puntero a la imagen en memoria en la segunda fila de pixeles.

Al principio de cada ciclo copiamos en los registros los 3 grupos de pixeles y quedan de la siguiente manera:

```

XMM0 ← [R12 + RDI] = - | p2 | p1 | p0
XMM1 ← [R13 + RDI] = - | p5 | p4 | p3
XMM2 ← [R8 + RDI] = - | p8 | p7 | p6

```

Despues desempaquetamos los pixeles de *byte* a *word* para poder sumar los 9 pixeles sin saturacion, hicimos una copia de cada uno para poder desempaquetar la parte inferior en un registro y la superior en otro ( $XMM1 = XMM0$ ,  $XMM3 = XMM2$ ,  $XMM5 = XMM4$ ) y ademas llenamos un registro ( $XMM12$ ) con ceros para expandir cada una de las componentes sin alterar el numero original. Una vez desempaquetados nos quedan los registros con los siguientes valores:

```

XMM0 ← p1 | p0
XMM1 ← - | p2
XMM2 ← p4 | p3
XMM3 ← - | p5
XMM4 ← p7 | p6
XMM5 ← - | p8

```

Luego sumamos los registros e hicimos la division.

Sumando  $XMM0$ ,  $XMM2$  y  $XMM3$  en  $XMM15$ :

```

XMM15 ← p1 + p4 + p7 | p0 + p3 + p6

```

Sumando  $XMM1$ ,  $XMM3$  y  $XMM4$  en  $XMM14$

```

XMM14 ← - | p2 + p5 + p8

```

Luego hicimos una copia de  $XMM15$  en  $XMM13$  y la shifteamos 8 bytes a la derecha

```

XMM13 ← - | p1 + p4 + p7

```

Por ultimo sumamos  $XMM15$ ,  $XMM14$  y  $XMM13$  en  $XMM15$

```

XMM15 ← - | p0 + p1 + p2 + p3 + p4 + p5 + p6 + p7 + p8

```

Para hacer la division optamos por multiplicar por  $(2^{16}/9)+1$  y luego shiftear 16 bits a la derecha cada componente. Para eso tengo el valor por el cual voy a multiplicar en memoria precalculado, al principio del programa decidimos guardarlo en  $XMM11$ . Luego, hicimos una copia de  $XMM15$  en  $XMM14$  y multiplicamos por  $XMM12$  guardando la parte superior de la mutiplicacion en  $XMM15$  y la inferior en  $XMM14$ . Luego empaquetamos los dos valores juntos como doubleword y los guardamos en  $XMM14$  y shifteamos 16 bits a la derecha. Por ultimo empaquetamos los valores obtenidos a words y luego a bytes, posteriormenent los guardamos en la posicion de memoria del pixel actual.

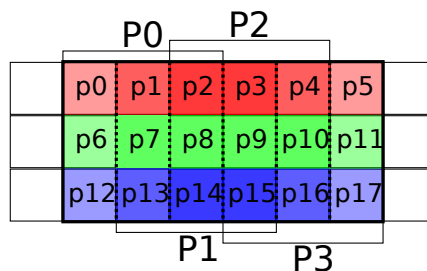
Aumentamos en 4 el iterador en  $X$  y comparamos con el tamaño en bytes de una linea de pixeles, si es menor iteramos nuevamente. En caso de que fuese mayor o igual, copiamos cambiamos los punteros de  $R12$  y  $R13$  y luego copiamos en  $R13$  los pixeles de la siguiente fila (las cuales seran necesarias para poder operar la siguiente fila).

Para terminar copiamos  $R8$  en  $R10$ , movemos  $R8$  una fila de pixeles, reseteamos el iterador  $X$   $RDI$ , incrementamos el iterador en  $Y$   $R9$  y si el iterador de  $Y$  es menos a la altura de la imagen iteramos nuevamente.

## 2.4. Implementacion 2

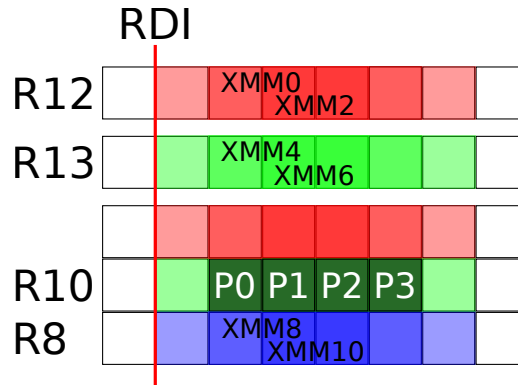
Para la segunda implementacion se nos pidio trabajar de a 4 pixeles por iteracion.

Para eso calculamos primero los pixeles  $P0$  y  $P1$ , y despues el  $P2$  y  $P3$  tratando de minimizar la cantidad de accesos a memoria.



Para hacer los calculos tomamos 6 set de 4 pixeles que nos permiten sumar en un solo registro 2 pixeles al mismo tiempo usando solo sumas, cada set ira adentro de un registro **XMM**. Ademas necesitamos 6 registros **XMM** mas para poder desempaquetar los bytes a words, y asi no tener saturacion.

Al igual que la primer implementacion, necesitaremos los mismos 6 registros de proposito general.



Antes de comenzar el ciclo inicializamos RDI en 0, R9 en 2, movimos R8 a R10, aumentamos R8 en una fila y posicionamos el puntero a la imagen en memoria en la segunda fila de pixeles.

Luego cargamos los 6 pixeles a sus respectivos registros:

|       |   |     |  |     |  |     |  |     |
|-------|---|-----|--|-----|--|-----|--|-----|
| XMM0  | ← | p3  |  | p2  |  | p1  |  | p0  |
| XMM2  | ← | p4  |  | p3  |  | p2  |  | p1  |
| XMM4  | ← | p9  |  | p8  |  | p7  |  | p6  |
| XMM6  | ← | p10 |  | p9  |  | p8  |  | p7  |
| XMM8  | ← | p15 |  | p14 |  | p13 |  | p12 |
| XMM10 | ← | p16 |  | p15 |  | p14 |  | p13 |

De la misma forma que en la primer implementacion desempaquetamos los registros haciendo una copia y desempaquetando la parte superior en el registro original y la inferior en la copia. Para desempaquetar usamos un registro con ceros (XMM15)

|       |   |     |  |     |
|-------|---|-----|--|-----|
| XMM0  | ← | p1  |  | p0  |
| XMM1  | ← | p3  |  | p2  |
| XMM2  | ← | p2  |  | p1  |
| XMM3  | ← | p4  |  | p3  |
| XMM4  | ← | p7  |  | p6  |
| XMM5  | ← | p9  |  | p8  |
| XMM6  | ← | p8  |  | p7  |
| XMM7  | ← | p10 |  | p9  |
| XMM8  | ← | p13 |  | p12 |
| XMM9  | ← | p15 |  | p14 |
| XMM10 | ← | p14 |  | p13 |
| XMM11 | ← | p16 |  | p15 |

Con todos los registros cargados, procedimos a sumar cada uno de ellos con XMM15, el resultado final fue el siguiente:

$$\text{XMM15} \leftarrow p1 + p2 + p3 + p7 + p8 + p9 + p13 + p14 + p15 \mid p0 + p1 + p2 + p6 + p7 + p8 + p12 + p13 + p14$$

Como podemos apreciar, el mismo responde al grafico presentado anteriormente, particularmente a los pixeles P0 y P1. Luego hicimos la division de cada pixel de la misma manera que en la primer implementacion, antes de dividir preservamos una copia de XMM15 en XMM9, dividimos por 9, shifteamos 8 bytes y dividimos nuevamente. Finalmente movimos los pixeles procesados a P0 y P1 a la memoria en la posicion correcta. Despues hubo que procesar los pixeles P2 y P3, para hacer esto movimos 3 grupos mas de pixeles de la memoria hacia los registros. En esta etapa tenemos:

|      |   |     |  |     |  |     |  |     |
|------|---|-----|--|-----|--|-----|--|-----|
| XMM0 | ← | p5  |  | p4  |  | p3  |  | p2  |
| XMM4 | ← | p11 |  | p10 |  | p9  |  | p8  |
| XMM8 | ← | p17 |  | p16 |  | p15 |  | p14 |

Luego hicimos copias en `XMM1`, `XMM5` y `XMM9` y desempaquetamos de byte a word:

```

XMM0 ← p3 | p2
XMM1 ← p5 | p4
XMM4 ← p9 | p8
XMM5 ← p11 | p10
XMM8 ← p15 | p14
XMM9 ← p17 | p16

```

Nuevamente sumamos los registros de la misma forma que antes, es decir, uno a uno con `XMM15`, el resultado final es:

```

XMM15 ← p3 + p4 + p5 + p9 + p10 + p11 + p15 + p16 + p17 | p2 + p3 + p4 + p8 + p9 + p10 + p14 + p15 + p16

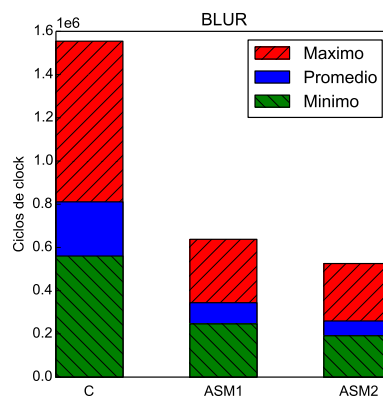
```

Como podemos apreciar, en este caso las cuentas tambien responden a lo presentado en los graficos, en este caso a los pixeles P2 y P3. De la misma forma que antes hacemos las divisiones apropiadas, y guardamos los pixeles procesados en memoria.

Una vez terminado el procesamiento de pixeles, comparamos el iterador con la cantidad de pixeles en una fila y repetimos hasta llegar hasta los ultimos 16 bytes de la fila, estos ultimos 16 bytes los cargamos en solo 3 registros `XMM` y los procesamos al igual que la primera parte del ciclo para asegurarme de no pasarnos. Al final del ciclo hacemos las copias pertinentes, incrementamos el iterador de filas y revisamos si llegamos al final.

## 2.5. Resultados

Para la experimentacion vamos a correr las 3 implementaciones (La version de C compilada con optimizaciones de nivel 3) para poder comparar la performance, vamos a usar la imagen de *lena* brindada por la catedra (Resolucion 160x160). Se ejecutara cada implementacion del algoritmo 100 veces y luego se calculara el tiempo (Usando el time stamp counter del procesador y la funcion de C brindada por la catedra) minimo, maximo y promedio (Sumando todos los tiempos tomados y diviendo por el la cantidad de iteraciones). A continuacion graficamos los tiempos en un grafico de barras que esta a continuacion:

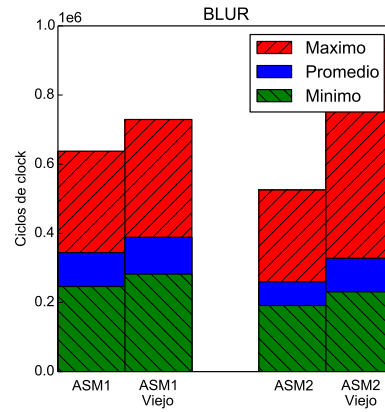


Con esto podemos ver que la implementacion ASM2 es un poco mas rapida que las otras implementaciones.

Revisando el codigo podemos ver que el tamaño de la imagen solo deberia modificar cantidad de pixeles a procesar aumentando el tiempo de ejecucion solo por que se necesita hacer mas iteraciones, tambien podemos ver que el algoritmo es independiente del color de los pixeles a procesar.

Para la reentrega se hicieron algunos cambios en el codigo que se supone que deberian mejorar la performance (Ver lista de cambios), decidimos correr el test explicado anteriormente el codigo nuevo y el codigo viejo para confirmar que los cambios propuestos cumplan con su objetivo. En el siguiente grafico graficaron los tiempos maximos, minimo y promedio de ejecucion de la implementaciones de ASM.





## 2.6. Conclusion

Podemos concluir que en el caso de la utilizacion de instrucciones SIMD para la aplicacion del algoritmo de blur se puede ver que operar de a varios pixeles mejora un poco con respecto a operar de a un pixel pero al mismo tiempo tambien complica considerablemente el codigo que se debio implementar en nuestro caso.

Igual la simple implementacion de instrucciones SIMD comparandola con el codigo C compilado se puede notar una diferencia conciderable en cuanto a la cantidad de ciclos de reloj usados por el mismo.

### 3. Filtro 2: Merge

#### 3.1. Cambios

Para el recuperatorio decidimos hacer cambios en el código.

La lista de cambios esta a continuacion, con la aclaracion de por que se hicieron los mismos:

- Se elimino uno de los iteradores de ciclo para reducir la cantidad de saltos condicionales y mejorar la performance del código.
- En la primera implementacion cambio la utilizacion de CALL addPixels por un macro debido a que los calls rompen el pipeline y requieren operaciones extra.
- Se cambio la escritura a disco de usar PEXTRD, PINSRD y luego hacer MOVDQU de un solo registro XMM a hacer MOVD del registro **XMM0** luego de cada llamada del macro addPixels

En la experimentacion se ve que los cambios no producen una mejora de performance notable pero decidimos dejarlos por que ayudaban a hacer el código mas claro para leer y por que a pesar de que fueron optimizaciones fallidas consideramos que los resultados inesperados que devolvio la experimentacion son importantes.

#### 3.2. Explicacion

El filtro *merge* consiste en tomar 2 imagenes del mismo tamaño y un parametro (llamdo **value** en el enunciado) de tipo *float* entre 0 y 1, multiplicar las componetes de la primer imagen por **value** y los de la segunda por  $1 - \text{value}$  y sumar los pixeles de ambas para crear una nueva imagen.

#### 3.3. Implementacion 1

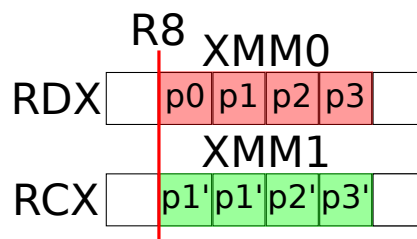
Para la primer implementacion se nos pidio trabajar con valores de tipo *float*, procesando la mayor cantidad de pixeles posibles por iteracion.

Ya que la cantidad de pixeles por imagen siempre va a ser un multiplo de 4 segun el enunciado, decidimos tomar de a 4 pixeles de la memoria para trabajar con los mismos, esto nos permitio simplificar el ciclo y operar sin tener que preocuparnos por excedernos del area de memoria alocada a la imagen.

Para procesar los pixeles del merge utilizamos 2 registros **XMM** para guardar los 4 pixeles como bytes que obtenemos de la memoria, y 2 registros mas para guardar copias de los mismos. Ademas como utilizamos un macro tambien precisamos de 4 registros **XMM** mas para poder guardar los resultados de la funcion. Por ultimo, tenemos **value** y  $1 - \text{value}$  guardados en los registros **XMM15** y **XMM14** respectivamente. Ademas de estos registros, se emplearon tambien los siguientes de proposito general:

- **RDX** puntero a la primera imagen
- **RCD** puntero a la segunda imagen
- **R9** cantidad de bytes totales de la imagen
- **R8** contador

Esta distribucion de registros reponde al siguiente grafico:



Antes de empezar a recorrer la imagen, para asignar los valores correctos de **XMM14** y **XMM15** utilizamos el dato **value** original almacenado en **XMM0**. Esto lo hacemos mediante una operacion de shuffle hacia **XMM15**, la misma copia **value**

a todas las componentes de `XMM15`. Para poder calcular `XMM14` alcanzo con mover al mismo una constante almacenada en memoria que tuviese 1.0 en los 4 *float* y posteriormente restarle el contenido de `XMM15`. Finalmente tenemos en `XMM14` y `XMM15`:

```
XMM14 ← 1 - value | 1 - value | 1 - value | 1 - value
XMM15 ← value | value | value | value
```

Para poder movernos por la imagen empleamos un ciclo, iteramos sobre el la imagen de a 16 bytes usando `R8` como contador. Antes de comenzar a iterar inicializamos `R8` en 0. Al comienzo de cada ciclo, movemos hacia los registros `XMM0` y `XMM1` los datos de los pixeles a procesar de la primer y segunda imagen respectivamente, ademas copiamos cada una en `XMM2` y `XMM3`, luego de esto procedimos a llamar al macro `addPixels` 4 veces. Al finalizar los llamados, tenemos en `XMM4` los 4 pixeles procesados, lo unico restante fue volcarlos en memoria. Una vez que los pixeles se encuentran en memoria, tenemos que aumentar el contador 16 bytes, tambien aumentamos los punteros a la imagenes por el mismo valor y luego comparo `R8` con `R9`. Si ya llegue al final de la imagen, termino el ciclo de la implementacion y procedo a retornar de la funcion.

El macro `addPixels` es una funcion auxiliar la cual hace la aritmetica necesaria para poder mergear las imagenes, esta funcion toma los valores de `XMM2` y `XMM3` y los copia hacia `XMM0` y `XMM1` respectivamente, luego procede a desempaquetarlos a doubleword utilizando el registro `XMM10` el cual contiene 0 en todas sus componentes. Una vez desempaquetados los valores, se procede a convertirlos a tipo *float* para poder realizar el producto con `XMM14` (1-value) y `XMM15` (value). Entonces tenemos:

```
XMM0 ← a * value | r * value | g * value | b * value
XMM1 ← a * (1 - value) | r * (1 - value) | g * (1 - value) | b * (1 - value)
```

Con estos valores en los registros, lo unico restante es sumarlos, convertirlos nuevamente a entero, y empaquetarlos a byte. Antes de finalizar la funcion, shifteamos los registros `XMM2` y `XMM3` 4 bytes a la derecha, asi en el siguiente llamado al macro ya van a estar los siguientes pixeles a procesar en el lugar correcto.

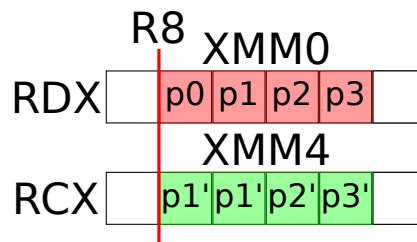
### 3.4. Implementacion 2

Para la segunda implementacion se no pidio trabajar con enteros procesando la mayor cantidad de enteros posibles por iteracion. Al igual que en la primera implementacion trabajamos de a 4 pixeles al mismo tiempo por que nos permite iterar de manera segura. Al igual que antes vamos a emplear los registros `XMM14` y `XMM15` para guardar los valores por los cuales vamos a multiplicar los pixeles. Para calcularlos movimos `value` a `XMM15`, luego aplicamos un shuffle para que los 4 floats de `XMM15` `value`, luego movimos a `XMM14` una constante de *float* almacenada en memoria la cual contenia 8192.0 en cada una de sus componentes, este registro fue multiplicado por `XMM15` almacenando el resultado del producto en `XMM15`, para obtener el 1 - value lo unico necesario fue restarle a `XMM14` el contenido de `XMM15`. Finalmente tenemos:

```
XMM15 ← 8192.0 * value | 8192.0 * value | 8192.0 * value | 8192.0 * value
XMM14 ← 8192.0 - 8192.0 * value | 8192.0 - 8192.0 * value | 8192.0 - 8192.0 * value | 8192.0 - 8192.0 * value
XMM14 ← 8192.0 * (1 - value) | 8192.0 * (1 - value) | 8192.0 * (1 - value) | 8192.0 * (1 - value)
```

El numero 8192, que es  $2^{13}$  fue elegido de manera empirica, ya que era el numero mas pequeño que nos permitia que el margen de error fuese lo suficientemente pequeño para poder cumplir con el enunciado.

Ademas de estos registros vamos a emplear los mismos registros de proposito general que la primer implementacion, junto con los registros `XMM0` a `XMM7` para tomar los valores de la imagen de la memoria y desempaquetar.

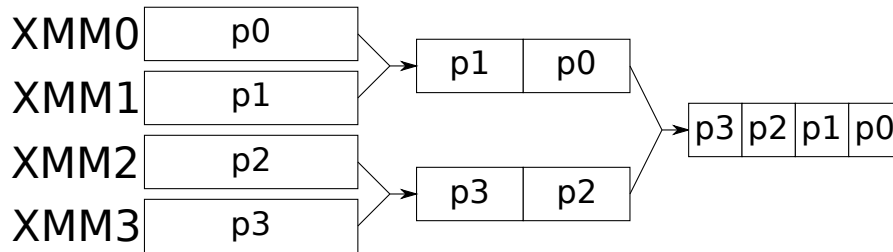


La estructura de la primera implementacion se mantuvo igual, es decir, se preservó el ciclo y el uso del contador, esto significa que las misma precauciones respecto a los mismos tomadas anteriormente aplican nuevamente en ese

caso. Una vez dentro del ciclo procedimos a mover hacia XMM0 y XMM4 los grupos de pixeles de la primera y segunda imagen respectivamente. Estos fueron desempquetados a doubleword, quedando de la siguiente forma:

```
XMM0 ← p0
XMM1 ← p1
XMM2 ← p2
XMM3 ← p3
XMM4 ← p0'
XMM5 ← p1'
XMM6 ← p2'
XMM7 ← p3'
```

Despues se multiplico a XMM0, XMM1, XMM2 y XMM3 por XMM15, el resultado de la operacion fue shifteado 14 bits hacia la derecha, obteniendo  $(p * 8192 * v) / 8192 = p * v$ . Esta misma logica se aplico con los otro cuatro registros, salvo que fueron multiplicados por XMM14, obteniendo  $(p * 8192 * (1 - v) / 8192 = p * (1 - v)$ . Una vez completado esto, finalmente tenemos cada pixel multiplicado por su value correspondiente, los mismos luego fueron empaquetados en un solo registro como muestra el grafico:



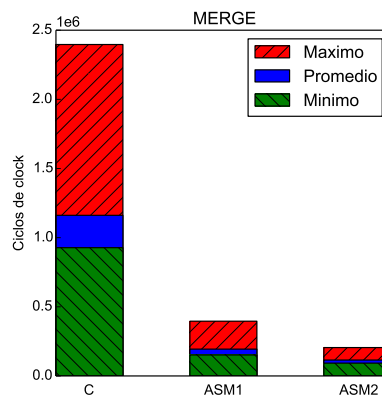
Al final tenemos:

```
XMM0 ← p3 * value | p2 * value | p1 * value | p0 * value
XMM4 ← p3' * (1 - value) | p2' * (1 - value) | p1' * (1 - value) | p0' * (1 - value)
```

Para terminar solo fue necesario sumarlos y moverlos a memoria.

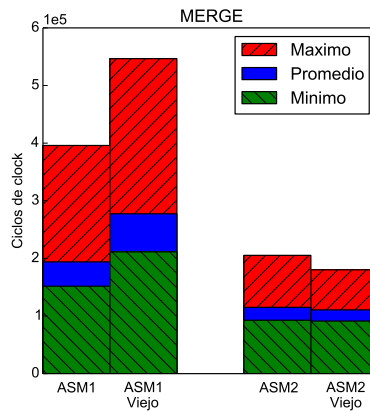
### 3.5. Resultados

Para la experimentacion vamos a correr las 3 implementaciones (La version de C compilada con optimizaciones de nivel 3) para poder comparar la performance, vamos a usar la imagenes de *lena* y *colores* brindadas por la catedra (Ambas con una resolucion 160x160). Se ejecutara cada implementacion del algoritmo 100 veces y luego se calculara el tiempo (Usando el time stamp counter del procesador y la funcion de C brindada por la catedra) minimo, maximo y promedio (Sumando todos los tiempos tomados y diviendo por el la cantidad de iteraciones). Luego pasamos a graficar los resultados para ver la performance de las mismas.



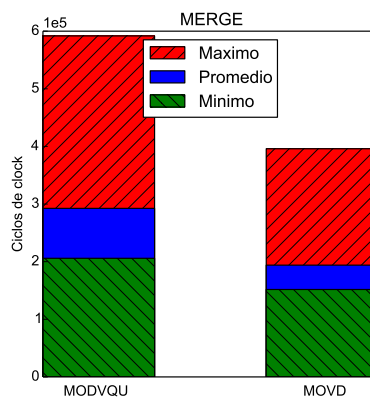
Con esto podemos ver claramente que la implementacion ASM2 corre mas rapida que las otras.

Tambien decidimos correr la implementacion de la primera entrega y compararla con la nueva para ver si los cambios propuestos mejoraron la performance.



Esta experimentacion nos sorprendio ya que la eliminacion de branches tanto con los calls como con los jumps practicamente no cambio el tiempo de ejecucion del programa, en la primera implementacion el cambio que vemos es principalmente por la manera en la que ahora hacemos el pasaje a memoria.

Luego de descubrir que la operacion PEXTRD y PINSRD son operaciones caras (Requieren varios ciclos de clock) decidimos probar si era mas rapido usar estas instrucciones o hacer 4 copias a memoria directamente desde el registro XMM usando MOVD.



Con esto podemos ver que las operaciones de extraccion e incersion de codigo son mas caras que hacer los 4 accesos a memoria.

### 3.6. Conclusion

Finalmente podemos concluir que la implementacion de ASM2 es la mas rapida de todas ya que al trabajar con bytes en vez de floats nos permite aprovecharnos mucho mas de las instrucciones SIMD reduciendo la cantidad de operaciones logradas.

## 4. Filtro 3: HSL

### 4.1. Explicacion

La idea de este filtro consiste en transformar una imagen del espacio RGB a HSL, sumarle un valor recibido por parametro a cada una de las componentes y luego convertir la imagen de HSL a RGB, esto debe hacerse sobre la totalidad de los pixeles. Para las conversiones de espacio y la suma, la catedra proveyo las ecuaciones apropiadas junto con una implementacion en *C* de las tres etapas.

### 4.2. Implementacion 1

Para la primer implementacion habia que implementar unicamente la etapa de *suma* en *Assembler*, utilizando las funciones de *C* provistas por la catedra para la conversion de RGB a HSL y viceversa. La ecuaciones de *suma* son las siguientes:

$$\begin{aligned} suma_h(h, HH) &= \begin{cases} h + HH - 360 & \text{si } h + HH \geq 360 \\ h + HH + 360 & \text{si } h + HH < 0 \\ h + HH & \end{cases} \\ suma_s(s, SS) &= \begin{cases} 1 & \text{si } s + SS \geq 1 \\ 0 & \text{si } s + SS < 0 \\ s + SS & \end{cases} \\ suma_l(l, LL) &= \begin{cases} 1 & \text{si } l + LL \geq 1 \\ 0 & \text{si } l + LL < 0 \\ l + LL & \end{cases} \end{aligned}$$

Para las operaciones con punto flotante se empleo el tipo *float*. Para comenzar las operaciones de *suma*, partimos de los siguientes registros *XMM* con sus respectivos valores:

```
XMM0 ← LL | SS | HH | -  
XMM1 ← 1 | s | h | -
```

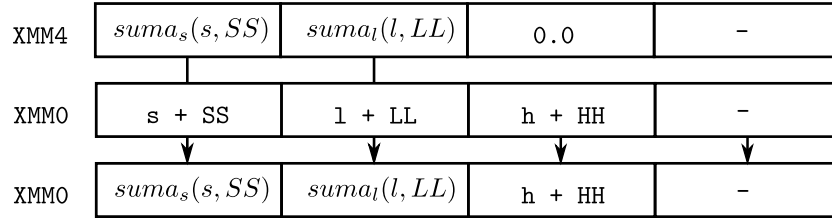
Tambien se cargaron en las siguientes mascarar:

```
XMM2 ← 1.0 | 1.0 | 0.0 | -  
XMM3 ← 0.0 | 0.0 | 0.0 | -  
XMM6 ← 0.0 | 0.0 | 360.0 | -  
XMM7 ← 0.0 | 0.0 | 0.0 | -  
XMM8 ← 0.0 | 0.0 | 360.0 | -  
XMM9 ← 0.0 | 0.0 | 360.0 | -
```

Con estos valores en los registros, se procedio a realizar la suma de la componentes de *XMM0* y *XMM1* con la instruccion *ADDPS* guardando el resultado en *XMM0*, y luego se almacenaron dos copias del mismo en *XMM4* y *XMM5* mediante la instruccion *MOVQDU*

La copia de *XMM4* se utilizo junto con las mascarar *XMM2* y *XMM3*, para poder satisfacer *suma\_s* y *suma\_l*. Primero se tomo el minimo entre *XMM4* y *XMM2*, almacenando el resultado en *XMM4*, luego se procedio a tomar el maximo entre *XMM4* y *XMM3*, nuevamente guardando el resultado en *XMM4*. Esto nos sirve para respetar la funcion de *suma\_s* y *suma\_l*, ya que en caso de que nuestros valores de *s* o *l* se excedan de 1.0, el mismo pasaria a ser 1.0, el mismo razonamiento aplica en el caso de que alguna de esas dos componentes sea menor a 0.0.

Una vez que tenemos los valores correspondientes en **XMM4**, procedemos a hacer un shuffle con el registro **XMM0**. El mismo lo hacemos con la instruccion **SHUFPS**, este responde al comportamiento mostrado en el siguiente grafico:



Con los valores correctos de  $s + SS$  y  $l + LL$ , solo queda analizar  $h + HH$ . Para hacer esto empleamos las mascaras de **XMM6** y **XMM7**, procedimos a hacer una comparacion entre estas y el registro **XMM5** para obtener las nuevas mascaras que nos permitieron filtrar los condicionales de  $suma_h$ . La secuencia de operaciones fue la siguiente:

```

XMM6 ← XMM5 ≥ 360.0
XMM7 ← XMM5 ≥ 0
XMM6 ← XMM5 AND XMM8
XMM7 ← NOT(XMM7) AND XMM9

```

Para las comparaciones se utilizo **CMPPS** y para las ultimas dos se utilizo **ANDPS** y **ANDNPS** respectivamente. Una vez finalizadas estas operaciones, si  $h + HH$  se excede o iguala a  $360.0$ , en el tercer float de **XMM6** tendríamos el valor  $360.0$  mientras que en el caso que  $h + HH$  sea menor a  $0.0$ , tendríamos el valor  $360.0$  en el tercer float de **XMM7**, si no se cumplen esas condiciones tendríamos el valor  $0.0$  en el tercer float del registro que no haya cumplido con el condicional. Con estos valores, lo unica operacion faltante fue a **XMM0** restarle **XMM6**, a este luego sumarle **XMM7** y guardar el resultado en **XMM0**, esto nos permite satisfacer el condicional de  $suma_h$ , y como originalmente **XMM6** y **XMM7** tenia  $0.0$  en los primeros dos *float*, esto hace que la ultima suma y resta no afecte los valores de  $s$  y  $l$ , ya que tanto **XMM6** y **XMM7** siguen teniendo  $0.0$  en los primeros dos *float*. Teniendo en cuenta esto, tenemos finalmente en **XMM0** un registro que cumple con:

```

XMM0 ←  $suma_l(l, LL) \mid suma_s(s, SS) \mid suma_h(h, HH) \mid -$ 

```

Con esto concluimos la etapa *suma* de la primer implementacion, solo resta llamar a la funcion *HSLtoRGB* provista por la catedra y volcar el resultado de la conversion en memoria.

### 4.3. Implementacion 2

Para la segunda implementacion se nos pidio implementar las 3 etapas del filtro en *Assembler*. Para la etapa *suma* se utilizo la misma implementacion que la primera.

#### 4.3.1. RGB a HSL

Al igual que el caso de *suma*, la catedra proveyo de las ecuaciones apropiadas. Las mismas estan presentadas a continuacion:

$$\begin{aligned}
 h(r, g, b) &= \begin{cases} 0 & \text{si } cmax = cmin \\ 60 * (g - b) / d + 6 & \text{si } cmax = r \\ 60 * (b - r) / d + 2 & \text{si } cmax = g \\ 60 * (r - g) / d + 4 & \text{si } cmax = b \end{cases} \quad (\text{mód } 360) \\
 l(r, g, b) &= \frac{cmax + cmin}{510} \\
 s(r, g, b) &= \begin{cases} 0 & \text{si } cmax = cmin \\ (d / (1 - \text{fabs}(2 * l(r, g, b) - 1))) / 255,0001 & \text{si } cmax \neq cmin \end{cases}
 \end{aligned}$$

Donde  $cmax = \max(r, g, b)$ ,  $cmin = \min(r, g, b)$  y  $d = cmax - cmin$ .

#### 4.3.1.1. Calculo de h

Primero procedimos a levantar las componentes del pixel de la memoria y copiarlo al registro **XMM1**, dicha operacion la hicimos con la instruccion **MOVD**. Despues desempaquetamos los datos de tipo *uint8* a *int*, esto fue hecho mediante **PUNPCKLBW** y **PUNPCKLWD** en ese orden, aplicando las instruccion sobre **XMM1** junto con algun registro **XMM** que contenga 0 en todos sus bits. Una vez con los valores en su tamaño correspondiente, procedimos a calcular **cmax** y **cmin**, esto lo logramos utilizando tres copias de **XMM1**, aplicando shifts a dos de ellas y tomando el minimo y maximo vertical almacenando el resultado en **XMM5** y **XMM6** respectivamente, los shifts fueron logrados mediante las instrucciones **PSRLDQ** y **PSLLDQ** segun el sentido en el cual deseamos shiftear. Hasta este momento tenemos en los registros:

```

XMM0 ← B | G | R | A
XMM1 ← B | G | R | 0
XMM5 ← cmin | - | - | -
XMM6 ← cmax | - | - | -

```

El registro **XMM0** contiene una copia con el canal *alpha* preservado, esta sera utilizada unicamente al final de la conversion. Con **cmin** y **cmax** en sus respectivos registros, procedimos a limpiarlos con una mascara y a hacer un shuffle con **XMM6**, copiando la componente **cmax** en las cuatro componentes de **XMM2**. Luego, hicimos una nueva copia de **XMM1** en **XMM3** y mediante un shift y un OR con **XMM5**, logramos juntar las componentes **cmin**, **R**, **G** y **B** en un solo registro. Entonces tenemos:

```

XMM5 ← cmin | 0 | 0 | 0
XMM6 ← cmax | 0 | 0 | 0
XMM2 ← cmax | cmax | cmax | cmax
XMM3 ← cmin | B | G | R

```

Los registros **XMM2** y **XMM3** fueron empleados para obtener la mascara para poder calcular la funcion  $h(r, g, b)$ . La misma fue obtenida utilizando la instruccion **PCMPEQD** entre **XMM3** y **XMM2**, almacenando el resultado de la comparacion de igualdad en **XMM3**. Una vez obtenida esta mascara, la misma fue reordenada via un shuffle, para seguir el mismo orden que el condicional de la implementacion *C*, el nuevo orden fue guardado en el registro **XMM4**. En esta etapa tenemos:

```

XMM3 ← cmin == cmax | B == cmax | G == cmax | R == cmax
XMM4 ← R == cmax | G == cmax | B == cmax | cmin == cmax

```

Uno de los problemas encontrados durante la implementacion fueron los casos donde existian 2 o mas componentes que eran iguales a **cmax**, esto hacia que la mascara tuviese mas de una componente con **0xFFFFFFFF**, lo cual hacia que los resultados al filtrar fuesen incorrectos. Para solucionar esto propusimos armar una mascara, la cual tuviese **0xFFFFFFFF** en las componentes despues la primer componente distinta de cero de **XMM4**, lo unico necesario seria hacer XOR bit a bit entre la nueva mascara y **XMM4**. El grafico a continuacion ilustra la idea:

|         |            |            |            |            |
|---------|------------|------------|------------|------------|
| XMM4    | 0x00000000 | 0xFFFFFFFF | 0x00000000 | 0xFFFFFFFF |
|         | XOR        | XOR        | XOR        | XOR        |
| mascara | 0x00000000 | 0x00000000 | 0xFFFFFFFF | 0xFFFFFFFF |
|         | =          | =          | =          | =          |
| XMM4    | 0x00000000 | 0xFFFFFFFF | 0x00000000 | 0x00000000 |

Una vez que tenemos esta mascara, preparamos los registros **XMM7**, **XMM8**, **XMM9** y **XMM11** con los siguientes valores y tambien disponiamos de una serie de constantes en memoria. Los valores de los registros y constantes son los siguientes:

```

XMM7 ← g | b | r | 0
XMM8 ← b | r | g | 0
XMM9 ← cmax | cmax | cmax | cmax
XMM11 ← cmin | cmin | cmin | cmin
cte_suma ← 6.0 | 2.0 | 4.0 | 0.0
cte_60 ← 60.0 | 60.0 | 60.0 | 60.0

```

Se procedio a realizar las operaciones verticales apropiadas para poder satisfacer la funcion  $h(r, g, b)$ . La unica salvedad es que luego de efectuar las operaciones de resta entre **XMM7** y **XMM8**, y la de **XMM11** y **XMM9**, el resultado de ambas



fue convertido a *float* mediante la instruccion `CVTDQ2PS` y almacenado en los registros `XMM8` y `XMM10` respectivamente. Posteriormente se procedio con el resto de las operaciones con las instrucciones de punto flotante apropiadas, y se guardo el resultado en `XMM8`, a este luego se le aplico la mascara guardada en `XMM4`.

Otro problema que encontramos fue el caso `cmin == cmax`, aqui tenemos que `d` es igual a cero, con lo cual al proceder con las divisiones nos topariamos con un NaN, para solucionar esto decidimos armar una nueva mascara que contenga `cmin == cmax` en cada una de sus componentes. Para hacerlo alcanzamos con utilizar el registro `XMM9` y `XMM2` (el mismo no fue modificado y mantiene los mismos valores que arriba) para hacer la comparacion de igualdad, el resultado de la misma fue guardado en `XMM9`. Con esto se procedio a hacer la operacion `NOT(XMM9) AND XMM8`, y para poder calcular el mód 360 de  $h(r, g, b)$  se aplico la misma logica que en caso de *suma* ya que los calculos para *h* no van a excederse de 720. Finalmente tenemos en `XMM10`:

`XMM10 ← - | - | - | h`

#### 4.3.1.2. Calculo de l y s

Para el calculo de *l* no hubo ninguna particularidad, el mismo se hizo en base a los siguientes registros y constantes:

`XMM7 ← cmax | 0 | 0 | 0`  
`XMM5 ← cmin | 0 | 0 | 0`  
`cte_510 ← 510.0 | 510.0 | 510.0 | 510.0`

Primero se realizo la resta entre `XMM7` y `XMM5` almacenando el resultado en `XMM7`, a esta se la convirtio a *float* mediante `CVTDQ2PS` previo a la multiplicacion con `cte_510`. Tenemos finalmente en resultado en `XMM8`, el mismo tiene la pinta:

`XMM8 ← 1 | 0 | 0 | 0`

Para el calculo de *s* se utilizaron los siguientes registros, mascaras y constantes:

`XMM7 ← cmax | 0 | 0 | 0`  
`XMM5 ← cmin | 0 | 0 | 0`  
`XMM5 ← 1 | 0 | 0 | 0`  
`cte_1 ← 1.0 | 1.0 | 1.0 | 1.0`  
`cte_2 ← 2.0 | 2.0 | 2.0 | 2.0`  
`cte_255 ← 255.0001 | 0 | 0 | 0`  
`masc_abs ← 0x7FFFFFFF | 0 | 0 | 0`  
`limpiar_msb ← 0xFFFFFFFF | 0 | 0 | 0`

En este caso hay dos puntos a destacar, estos son el condicional de  $s(r, g, b)$  y el *fabs*. Para poder filtrar segun si `cmax == cmin` alcanza con hacer una comparacion de igualdad entre `cmax` y `cmin` mediante `PCMPEQD`, guardar el resultado de la misma en un registro, y luego de hacer las operaciones correspondientes aplicar el resultado de la comparacion mediante un `AND`. En el caso del *fabs*, al estar operando con numeros de tipo *float* codificados bajo la norma IEEE 754, basta con colocar el bit mas significativo en cero, esto lo podemos hacer mediante un `AND` con el numero al cual deseamos aplicar *fabs* y la *masc\_abs*. Finalmente tenemos en el registro `XMM7`:

`XMM7 ← s | 0 | 0 | 0`

Con los valores de *h*, *s* y *l* calculados, lo unico que falta por hacer es juntar todo en un solo registro junto con el dato del canal *alpha*, el cual se encuentra en el registro `XMM0`. Para hacer esto, vamos a shiftear los registros que contienen a *s* y a *h*, 4 y 8 bytes hacia la derecha respectivamente. Una vez que tenemos esto, lo vamos a juntar a todos en un unico registro mediante un `OR` y vamos a procede a aplicarle a `XMM0` la mascara `limpiar_msb`, convertir el resultado a *float* y juntarlo junto con lo que ya teniamos utilizando nuevamente un `OR`. Con esto podemos concluir la etapa *RGBtoHSL*.

#### 4.3.2. HSL a RGB

Tanto como en el caso de *suma* y *RGBtoHSL*, la catedra proveyo las ecuaciones necesarias para la conversion, las mismas son:

$$RGBAux(h, s, l) = \begin{cases} (c, x, 0) & \text{si } 0 \leq h < 60 \\ (x, c, 0) & \text{si } 60 \leq h < 120 \\ (0, c, x) & \text{si } 120 \leq h < 180 \\ (0, x, c) & \text{si } 180 \leq h < 240 \\ (x, 0, c) & \text{si } 240 \leq h < 300 \\ (c, 0, x) & \text{si } 300 \leq h < 360 \end{cases}$$

$$RGB(h, s, l) = (RGBAux(h, s, l)_r * 255, RGBAux(h, s, l)_g * 255, RGBAux(h, s, l)_b * 255)$$

Donde  $c = (1 - fabs(2 * l - 1)) * s$ ,  $x = c * (1 - fabs(fmod(h/60, 2) - 1))$  y  $m = 1 - c/2$ . Se tuvo que hacer un cambio, ya que en el enunciado las componentes B y G se encontraban invertidas, imposibilitando la conversión correcta.

Para poder hacer el cálculo de R, G, B tuvimos que hacer el de c, x y m.

#### 4.3.2.1. Cálculo de c

Para el cálculo de c, tenemos los siguientes registros, mascarar y constantes:

|          |   |            |  |            |  |            |  |            |
|----------|---|------------|--|------------|--|------------|--|------------|
| XMM1     | ← | 1          |  | 1          |  | 1          |  | 1          |
| XMMn     | ← | s          |  | s          |  | s          |  | s          |
| cte_1    | ← | 1.0        |  | 1.0        |  | 1.0        |  | 1.0        |
| cte_2    | ← | 2.0        |  | 2.0        |  | 2.0        |  | 2.0        |
| mask_abs | ← | 0x7FFFFFFF |  | 0x7FFFFFFF |  | 0x7FFFFFFF |  | 0x7FFFFFFF |

Con estos podemos hacer las operaciones correspondientes para satisfacer el cálculo de c. El único caso que requiere atención particular es el de **fabs**, debido a que los números de punto flotante están codificados con la norma IEEE 754, lo único que hace falta para poder obtener el valor absoluto es colocar el bit más significativo de cada *float* en cero, para hacer esto alcanza con aplicar la máscara **mask\_abs** mediante la instrucción **ANDPS** al registro apropiado. Tras finalizar las operaciones tenemos en **XMM1**:

$$XMM1 \leftarrow c \mid c \mid c \mid c$$

Posteriormente procedimos a limpiar el registro, dejando únicamente el *float* de la posición menos significativa. Entonces obtenemos:

$$XMM1 \leftarrow 0 \mid 0 \mid 0 \mid c$$

Esto lo hacemos para después poder armar todas las posibilidades de  $RGBAux(h, s, l)$ .

#### 4.3.2.2. Cálculo de x

Al igual que con c, vamos a empezar por los registros:

|          |   |            |  |            |  |            |  |            |
|----------|---|------------|--|------------|--|------------|--|------------|
| XMM0     | ← | 1          |  | s          |  | h          |  | a          |
| XMM1     | ← | 0          |  | 0          |  | 0          |  | c          |
| XMM3     | ← | h          |  | h          |  | h          |  | h          |
| cte_1    | ← | 1.0        |  | 1.0        |  | 1.0        |  | 1.0        |
| cte_2    | ← | 2.0        |  | 2.0        |  | 2.0        |  | 2.0        |
| cte_60   | ← | 60.0       |  | 60.0       |  | 60.0       |  | 60.0       |
| mask_abs | ← | 0x7FFFFFFF |  | 0x7FFFFFFF |  | 0x7FFFFFFF |  | 0x7FFFFFFF |

Con estos valores podemos hacer fácilmente los cálculos de x, para **fabs** aplicamos la misma lógica que antes, el único caso que requiere atención particular es el de **fmod**. La función **fmod** realiza las siguientes operaciones:

$$fmod(a, b) = a - \left\lfloor \frac{a}{b} \right\rfloor * b$$

Para poder implementar esta función con instrucciones SSE, tenemos que encontrar una forma de tomar la parte entera de la división, para hacer esto decidimos utilizar la instrucción **ROUNDPS**, la misma si es utilizada con el inmediato **0x03** nos permite guardar una versión redondeada con cero en la parte decimal del operando fuente. Una vez

que disponimos de este resultado, solo queda aplicar la multiplicacion y resta correspondiente. Una vez finalizadas las operaciones requeridas, tenemos en **XMM2**:

$$\text{XMM2} \leftarrow 0 \mid 0 \mid 0 \mid x$$

Los primeros tres *float* quedan en cero puesto a que al hacer la multiplicacion con **XMM1**, el mismo tenia *c* unicamente en la ultima componente.

#### 4.3.2.3. Calculo de *m*

Para el calculo de *m* no hubo mayor particularidad, el mismo se llevo a cabo con los siguientes registros y mascaras:

$$\text{XMM3} \leftarrow 1 \mid 1 \mid 1 \mid 1$$

$$\text{XMM4} \leftarrow c \mid c \mid c \mid c$$

$$\text{cte\_2} \leftarrow 2 \mid 2 \mid 2 \mid 2$$

El resultado final fue guardado en el registro **XMM3**, el mismo quedo asi:

$$\text{XMM3} \leftarrow m \mid m \mid m \mid m$$

#### 4.3.2.4. Calculo de RGB

Para el calculo de *RGBAux*, se tuvieron que armar seis mascaras, las mismas filtraban los 6 casos de la funcion. Para armarlas se conto con los siguientes registros y constantes:

$$\text{XMM9} \leftarrow h \mid h \mid h \mid h$$

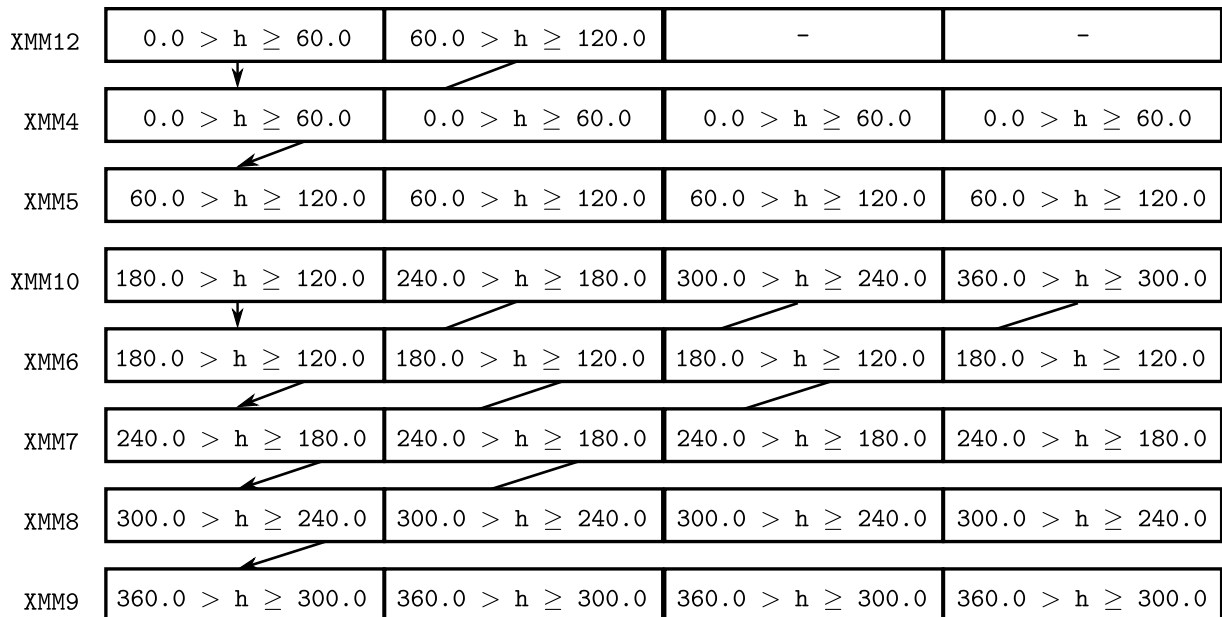
$$\text{cte\_cmp1} \leftarrow 180.0 \mid 240.0 \mid 300.0 \mid 360.0$$

$$\text{cte\_cmp2} \leftarrow 120.0 \mid 180.0 \mid 240.0 \mid 300.0$$

$$\text{cte\_cmp3} \leftarrow 0.0 \mid 0.0 \mid 60.0 \mid 120.0$$

$$\text{cte\_cmp4} \leftarrow 0.0 \mid 0.0 \mid 0.0 \mid 60.0$$

La idea consistio en comparar si  $\text{XMM9} < \text{cte\_cmp1}$  y  $\text{XMM9} \geq \text{cte\_cmp2}$ , luego se procedio a unificar estas dos comparaciones con un AND, es decir, nos quedamos unicamente con los casos donde una componente de **XMM9** sea menor a su respectiva componente de **cte\_cmp1** y mayor o igual a la de **cte\_cmp2**, luego se almaceno el resultado en **XMM10**. Este mismo proceso fue repetido pero con **cte\_cmp3** y **cte\_cmp4**, y el resultado fue almacenado en **XMM12**. Estos resultados luego fueron expandidos a 6 registros **XMM** mediante un shuffle, el proceso responde al siguiente grafico:



Con estas mascarar armadas, se procedio a armar las 6 posibles combinaciones de R, G y B. Las mismas son:

```
XMM10 ← 0 | x | c | 0
XMM11 ← 0 | c | x | 0
XMM12 ← x | c | 0 | 0
XMM13 ← c | x | 0 | 0
XMM14 ← c | 0 | x | 0
XMM15 ← x | 0 | c | 0
```

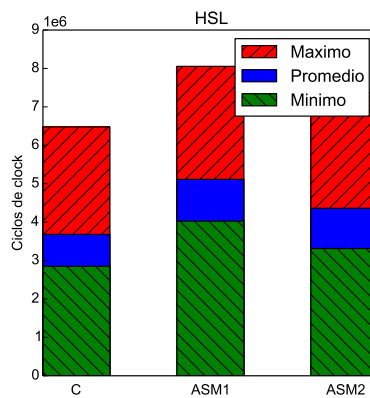
Lo unico restante fue aplicar cada una de las mascarar formadas anteriormente a su correspondiente registro de la lista anterior mediante un AND, y luego sumar todo en un solo registro. Con eso ya teniamos el valor final de *RGBAux* en el registro XMM4, ademas de este teniamos tambien el siguientes registro y constante:

```
cte_255 ← 255.0 | 255.0 | 255.0 | 255.0
XMM3 ← m | m | m | m
```

Con estos registros, lo unico faltante fue realizar las operaciones aritmeticas indicadas en la funcion *RGB*, mencionada al comienzo del esta seccion, y unificar el resultado con el canal *alpha* del registro XMM0. Para hacer esto alcanzo con quedarnos unicamente con la componente menos signifactiva y juntarla mediante un OR con el resultado final de *RGB*. Esto nos da el resultado final de la conversion, el siguiente paso fue convertir los numeros de tipo *float* de 32 bits a *uint* de 8 bits, para hacer esto basto con convertir de *float* a *int* de 32 bits via la instruccion CVTQSDQ y empaquetar el resultado de 32 bits a 8 bits mediante las instrucciones PACKUSDW y PACKUSWB, en ese orden. Una vez concluido el empaquetado, procedimos a volcar dicho resultado a la misma posicion de memoria a partir de la cual iniciamos el ciclo.

#### 4.4. Resultados

Para la experimentacion vamos a correr las 3 implementaciones (La version de C compilada con optimizaciones de nivel 3) para poder comparar la performance, vamos a usar la imagen de *lena* brindada por la catedra (Resolucion 160x160). Se ejecutara cada implementacion del algoritmo 100 veces y luego se calculara el tiempo (Usando el time stamp counter del procesador y la funcion de C brindada por la catedra) minimo, maximo y promedio (Sumando todos los tiempos tomados y diviendo por el la cantidad de iteraciones). Luego pasamos a graficar los resultados para ver la performance de las mismas.



Luego decidimos probar el algoritmo con diferentes imagenes para testear si los saltos condicionales impactan en el rendimiento de las implementaciones.

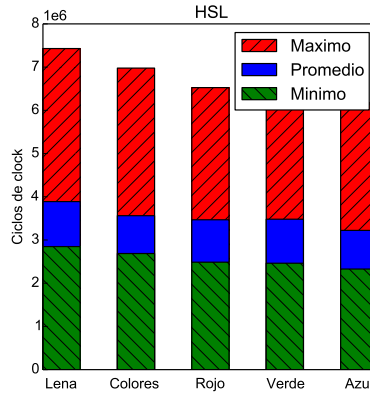


Figura 1: C

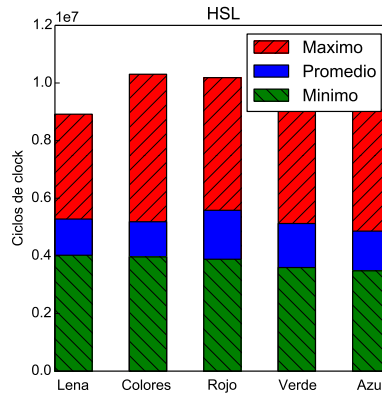


Figura 2: ASM1

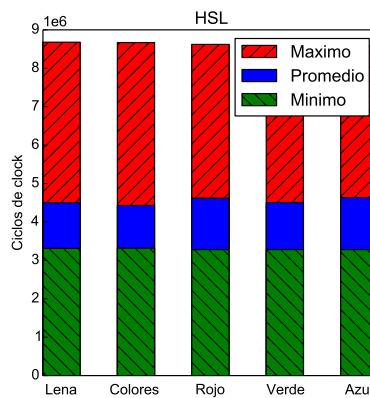


Figura 3: ASM2

## 4.5. Conclusion

En conclusion podemos ver que la version de C es la mas rapida, suponemos que esto es asi debido a que la prediccion de saltos del procesador es lo suficientemente buena como para que sea mas rapido que hacer las cosas usando SIMD y procesando todo usando mascarar.

Tambien podemos ver tambien que los saltos afectan la consistencia del algoritmo ya que la implementacion ASM2 posee tiempos de ejecucion mucho mas consistentes, en cambio la de C y ASM1 (La cual posee funciones con saltos) poseen variaciones en los tiempos de ejecucion.