TRABAJO PRÁCTICO DE ARQUITECTURA DEL COMPUTADOR

FACULTAD: Ciencias Exactas, Ingeniería y Agrimensura.

CARRERA: Licenciatura en Cs de la Computacion.

INTEGRANTES: Diego Chiodi : C/3847/4
Denise Marzorati : M/6219/7
Luis Dzikiewicz : D/3850/4

FECHA: 20 de Febrero de 2016.

TRABAJO SELECCIONADO: Fractal.

El trabajo práctico se dividió en varias etapas:

1. Recopilación de información sobre fractales.
2. Investigación acerca del formato de mapas de bits.
3. Realización de la estructura principal del trabajo en C.
4. Familiarización (como extensión a lo visto en clase) de las instrucciones y uso del coprocesador de coma flotante de x86.
5. Desarrollo de la función que realiza la transformación lineal en assembler.
6. Adición de opciones extra tales como “ESCALA” y “OFFSET” que modifican el resultado final del fractal.

1.Recopilación de información sobre fractales

En esta etapa aumentamos la noción que teníamos sobre el tema a tratar.

Existen distintos tipos de fractales que se pueden clasificar sobre la **forma en que se generan**:

• ***Lineales:*** a partir de conceptos y algoritmos lineales (rectas, triángulos, etc).

• ***Complejos:*** son generados mediante algoritmos de escape (para cada punto se repite un proceso hasta cumplir cierta condición).

• ***Órbitas caóticas:*** este tipo de modelo nació con un estudio sobre órbitas caóticas desarrollado por Edward Lorenz en 1963.

• ***Autómatas celulares:*** desarrollado por John von Neumann y Stanislaw Ulam a fines de la década de 1940 para representar la reproducción en algunos sistemas biológicos.

Aplicaciones de los fractales en el mundo real:

• ***Comunicaciones:*** modelado del tráfico en redes.

• ***Informática:*** técnicas de compresión (audio y vídeo).

• ***Robótica:*** robots fractales.

• ***Infografía:*** paisajes fractales y otros objetos.

• ***Biología:*** crecimiento tejidos, organización celular, Evolución de poblaciones, Depredador-presa.

• ***Matemáticas:*** convergencia de métodos numéricos.

• ***Música:*** composición musical.

• ***Física:*** transiciones de fase en magnetismo.

• ***Química:*** agregación por difusión limitada (DLA).

•***Geología:*** análisis de patrones sísmicos, fenómenos de erosión, modelos de formaciones geológicas.

• ***Economía:*** análisis bursátil y de mercado.

En este trabajo se abordó el enfoque de la generación de ***fractales lineales***. Los fractales lineales son aquellos que se construyen con un cambio en la variación de sus escalas. Esto implica que los fractales lineales son idénticos en sus escalas hasta el infinito. La generación de fractales a través de ecuaciones iteradas consiste en repetir y volver sobre sí mismo una cierta cantidad de veces.

2. Investigación acerca del formato de mapas de bits.

El formato BMP (Bits Maps Pixels) es un archivo con pixeles almacenados en forma de tabla de puntos. Se lo utiliza para almacenar imágenes independientes del dispositivo e independientes de la aplicación.

Existen distintas versiones, las más conocidas son:

•***bmp v2:*** tamaño de la cabecera: 12 bytes (BITMAPCOREHEADER).

•***bmp v3:*** tamaño de la cabecera: 40 bytes (BITMAPINFOHEADER). Esta es la versión más utilizada.

•***bmp v4:*** tamaño de la cabecera: 108 bytes(BITMAPV4HEADER).

•***bmp v5:*** tamaño de la cabecera: 124 bytes (BITMAPV5HEADER).

Así mismo hay distintas profundidades de colores posibles: 1, 4, 8, 16, 24, paleta personalizada, e incluso las últimas versiones permiten compresión RLE y profundidad de 32 bits.

En este caso, a los fines prácticos se desarrolló la imagen BMP en la versión 2, con una profundidad de color de 24 bit (no requiere paleta de colores), sin compresión (de hecho esta versión no la admite).

Especificaciones de la versión BMP v2:

Todas las versiones contienen en el comienzo del archivo, una cabecera de archivo (File Header), que consta de 14 bytes, compuestos por:

•***FileType (2 bytes):*** se utiliza para identificar el tipo de archivo. En este caso: “BM”.

•***FileSize (4 bytes):*** especifica el tamaño del archivo (útil en la compresión). Sin compresión, en este caso: 0.

•***Reserved (4 bytes):*** es utilizado por las aplicaciones que manipulan el archivo. En este caso: 0.

•***BitmapOffset (4 bytes):*** indica el comienzo de los pixeles de la imagen. En este caso: 26.

Luego continúa una segunda cabecera de 12 bytes, propia de la versión 2:

•***Size (4 bytes):*** indica el tamaño de los datos de esta cabecera. El tamaño correspondiente a la versión 2 es 12.

•***Ancho (2 bytes):*** ancho en pixeles de la imagen.

•***Alto (2 bytes):*** alto en pixeles de la imagen.

•***Planos (2 bytes):*** cantidad de planos de colores. Bmp admite solo uno.

•***Prof (2 bytes):*** indica la cantidad de bits por pixeles en cada plano. Se seleccionó 24 bits.

Nota: si la altura es un número positivo, entonces la imagen es un mapa de bits "de abajo hacia arriba" con el origen en la esquina inferior izquierda; y si es negativo, la imagen es un mapa de bits "de arriba hacia abajo" con el origen en la esquina superior izquierda.

Aquí se decidió utilizar números positivos, “invirtiendo” la imagen, pues es la manera más normal, o clásica, de trabajar con archivos bmp (evitando así posibles incompatibilidades con alguna aplicación).

Tras estas cabeceras, se colocan los datos de la paleta de colores. Innecesaria en el caso de una profundidad de color de 24 bits.

Finalmente, a continuación se colocan los datos de la imagen propiamente dicha (“de abajo hacia arriba”). La imagen se construye pixel por pixel. Cada pixel está compuesto por 3 bytes, uno por cada canal de color (RGB), los valores en cada byte varían entre 0 y 255. Al escribir los pixeles, la información esta invertida: BLUE | GREEN | RED.

3.Realización de la estructura principal del trabajo en C

Para una mejor organización del trabajo y para facilitar la comprensión del mismo, se trabajó con dos ficheros, que son los que se detallan a continuación:

•***Fichero cabecera “tp.h”:*** en este fichero se incluyeron los siguientes datos:

a. Declaración de cuatro tipos de estructuras. Detallando:

• datos_vista: utilizada para contener la información adicional sobre la presentación de la imagen a generar.

• bmp_header: utilizada para organizar los datos requeridos en la cabecera del bmp.

• datos: en esta estructura se agrupó la información pertinente a la generación de los fractales.

• punto: utilizada para guardar las componentes en el eje de las x y el eje de las y de un punto del plano.

b. Definición de diez macros. Dichas macros son las que se enumeran en las siguientes líneas:

• cant(l), iter(l): puesta en práctica para el acceso a los datos contenidos en la estructura 'l' del tipo 'datos'.

• escala(l), of_x(l), of_y(l): dichas macros brindan acceso a los valores en la estructura 'l' del tipo' datos_vista'.

• list_ecs(l, n): esta macro permite acceder al sistema de ecuaciones ubicado en la posición 'n' del arreglo de ecuaciones contenido en la estructura 'l' del tipo 'datos_vista'.

• coef_ec(l, n, k): mediante el macro anterior, accedemos al coeficiente 'k', del sistema de ecuaciones 'n' en la estructura 'l'.

• ANCHO, ALTO: estas macros permitieron delimitar las dimensiones de la imagen que ha de ser generada.

• OFFSET: con esta macro se definió a partir de que byte comienza la imagen (26 para la versión de bmp que elegimos)

c. Definición de los prototipos de las nueve funciones utilizadas en el desarrollo del programa. A continuación se brinda una breve descripción de las mismas:

• leer_datos_vista(): se encarga de leer los datos adicionales sobre la imagen a generar.

• leer_datos(): utilizada para leer los datos básicos necesarios para generar la imagen fractal.

• base_image(): crea el fichero bmp básico con un color de fondo.

• pto_random(): genera un punto de coordenadas aleatorias (acotadas).

• ec_random(): selecciona un sistema de ecuaciones aleatoriamente desde la lista de sistema de ecuaciones.

• despl(): calcula el desplazamiento necesario para marcar el píxel.

• marcar_pxl(): graba un píxel en la imagen.

• lib_dat(): función destinada a liberar la memoria utilizada.

• solve(): función en lenguaje ensamblador (AT&T) que se encarga de resolver el sistema de ecuaciones.

• ***Fichero principal "fractales.c":***

la primer función en ser llamada es leer_datos_vista(), encargada de tomar por la entrada estándar los parámetros adicionales referentes al escalado, y reposicionamiento de la imagen, y retornar la dirección de memoria donde fueron guardados dichos parámetros. Acto seguido guarda dicha dirección en vista, que es un puntero a 'datos_vista'.

A continuación en ret, a través de leer_datos(), reservamos la memoria necesaria y tomamos desde la entrada estándar la cantidad de sistemas de ecuaciones, números de iteraciones, y los sistemas de ecuaciones. La manera de guardar dichos sistemas, es mediante una tabla (lis_ecs) con tantas filas como sistemas y 6 columnas correspondientes cada una a cada coeficiente (los 3 primeros pertenecientes a la primer ecuación y los 3 restantes pertenecientes a la segunda ecuación).

Luego generamos fichero bmp base, mediante la función base_image(); el nombre del fichero está dado por la cadena guardada en nombre, sus dimensiones por los macros ALTO y ANCHO, y el color de fondo por pxb[] (privado de base_image()).

Una vez obtenida la imagen que servirá de base para "dibujar" el fractal, se procede a generar un punto aleatoriamente, dentro de ciertos parámetros (para no intentar marcar puntos que están por "fuera" del lienzo), a través la función pto_random().

Terminados los preparativos preliminares, abrimos el fichero con la imagen base y procedemos a realizar las iteraciones necesarias, mediante un bucle.

Dicho bucle realiza el siguiente bloque:

a. Un llamado a marcar_pxl(): en la primer iteración actúa sobre el punto obtenido aleatoriamente, luego actúa sobre el punto obtenido por la función solve.

b. Seleccionar un sistema de ecuaciones de manera aleatoria: esto se lleva a cabo a través del puntero 'ecuación' y la función ec_random().

c. Generar uno de los puntos que construirá la imagen fractal: mediante la aplicación de la transformación lineal al punto, a través de la función solve().

Una vez concluidas las iteraciones del bucle, se procede a cerrar el fichero.

Finalmente, se libera la memoria reservada de manera dinámica durante la ejecución del programa.

4.Familiarización (como extensión a lo visto en clase) de las instrucciones y uso del coprocesador de coma flotante de x86.

Para mejorar el rendimiento del programa en lo referido al tiempo de procesamiento, se implementó operaciones del tipo packed cuando fuera posible. Dados los requerimientos observados, fueron necesarias las siguientes operaciones:

•***shufps:*** esta función toma como argumentos un byte de control y dos registros de 128 bits, o un registro de 128 bits y un espacio de memoria de 128 bits (xmm1 y xmm2/m128). De acuerdo al valor que se encuentre en el byte de control, la función copiará a la parte baja de xmm2/m128 8 bytes provenientes de xmm1 (pudiendo seleccionar de a cuatro bytes) y 8 bytes provenientes de xmm2/m128 (pudiendo seleccionar de a cuatro bytes) a la parte alta de xmm2/m128.

En particular, fue utilizada en este trabajo para rotar los bits correspondientes a los registros de 128 bits.

•***movddup:*** toma como argumento un registro de 128 bits o bien un espacio de memoria del de 64 bits xmm2/m64, y un registro de 128 bits xmm1. Acto seguido, copia en los 64 bits menos significativos de xmm1 los 64 bits menos significativos de xmm2 o los 64 bits de m64, y realiza la misma operación en los 64 bits más significativos de xmm1.

•***movhps:*** toma como argumentos un espacio de memoria de 64 bits m64, y un registro de 128 bits xmm1. Si el destino es xmm1, copia los 64 bits de m64 en los 64 bits

más significativos de xmm1. Caso contrario, copia los 64 bits más significativos de xmm1 a m64.

•***addps:*** esta función suma flotantes de precisión simple empaquetados.

•***mulps:*** esta función multiplica flotantes de precisión simple empaquetados.

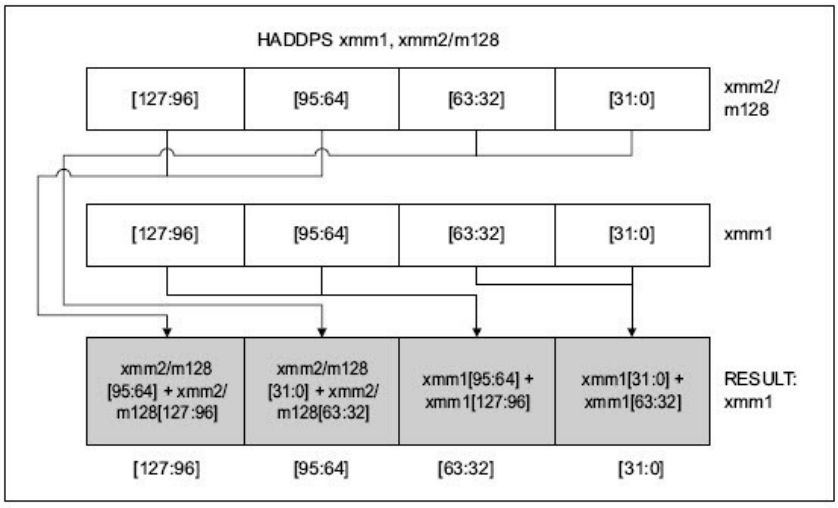
•***haddps:*** esta función toma como argumentos a un registro de 128 bits (xmm1) y otro registro de 128 bits o bien un espacio en memoria de 128 bits (xmm2/m128). Realiza lo siguiente:

-Realiza la suma entre el byte menos significativo de xmm1 y el segundo byte menos significativo de xmm1, y guarda el resultado en el byte menos significativo de xmm1.

-Realiza la suma entre el segundo byte más significativo de xmm1 y el byte más significativo de xmm1, y guarda el resultado en el segundo byte menos significativo de xmm1.

-Realiza la suma entre el byte menos significativo de xmm2/m128 y el segundo byte menos significativo de xmm2/m128, y guarda el resultado en el segundo byte más significativo de xmm1.

-Realiza la suma entre el segundo byte más significativo de xmm2/m128 y el byte más significativo de xmm2/m128, y guarda el resultado en el byte más significativo de xmm1.



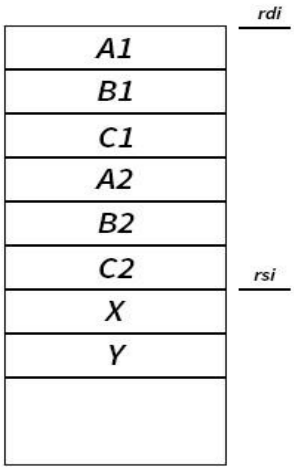
Observaciones: en el desarrollo del código en assembler se sacó provecho de las formas particulares en que operan movsd y movss de acuerdo al tipo de sus operandos; ya que por ejemplo si el operando de destino de movss es un registro y la fuente un espacio de memoria los bits 127-32 del registro son seteados a cero, mientras que al trabajar con dos registros dichos bits permanecen iguales a como se encontraban antes de realizar la operación.

5.Desarrollo de la función que realiza la transformación lineal en assembler.

Se considera que al momento de realizar esta transformación: se ha seleccionado una ecuación de coeficientes A1, B1, C1, A2, B2, C2; se tienen las coordenadas de un punto de componentes X e Y. Se desea realizar la transformación: X<-A1*X+B1*Y+C1; Y<-A2*X+B2*Y+C2.

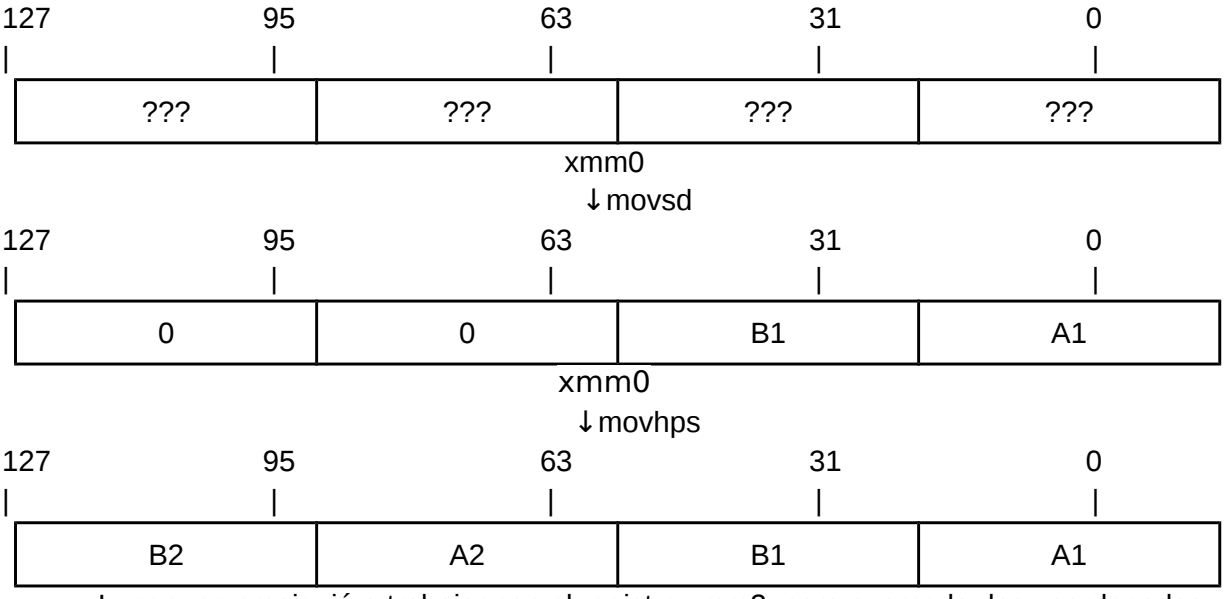
El primer problema que surgió en esta etapa fue el desconocimiento acerca del guardado de datos en memoria. Puesto que se trabajó con estructuras, fue necesario realizar varias pruebas previas para observar cómo se guardaban los datos, y se concluyó que se guardan en memoria del siguiente modo:

Como se ha mencionado en el apartado anterior, se decidió trabajar en lo posible

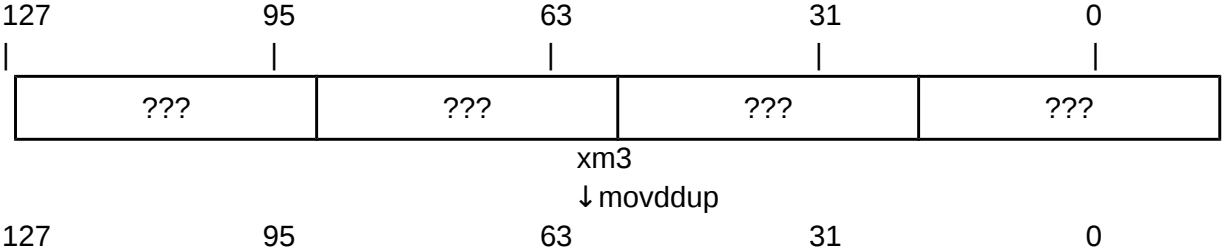


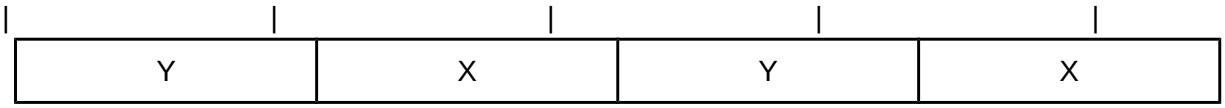
con instrucciones packed para optimizar el rendimiento. Por eso mismo se elaboró el siguiente esquema que explica en términos generales el código desarrollado en assembler:

•***Paso 1:*** obtener A1*X,B1*Y,A2*X,B2*Y. Para ello, fue necesario primero acomodar en un registro de 128 bits los valores |A1|B1|A2|B2| y en otro registro también de 128 bits los valores |X|Y|X|Y|. Esto ocurre:



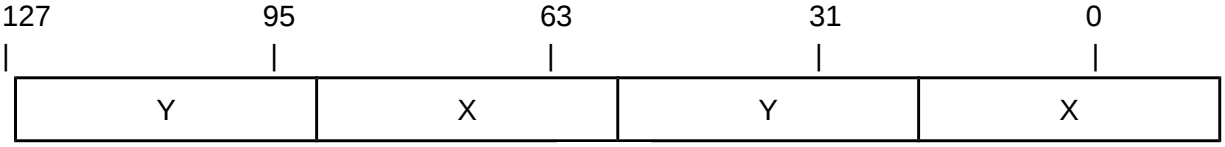
Luego, se prosiguió a trabajar con el registro xmm3, para acomodar los coordenadas del punto:



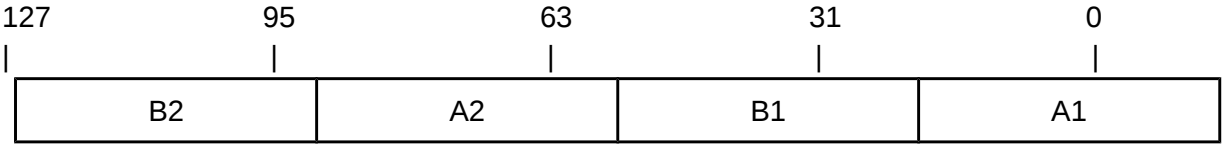


xmm3

Para finalizar este paso se realizó una sencilla operación de multiplicación:

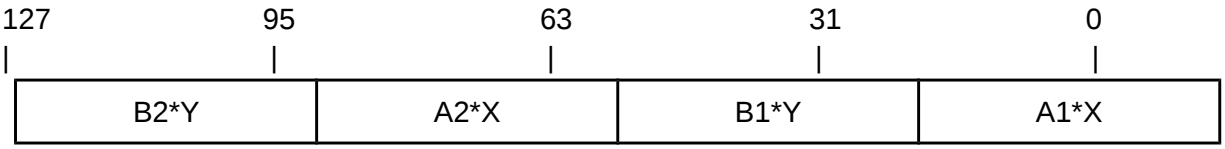


xmm3



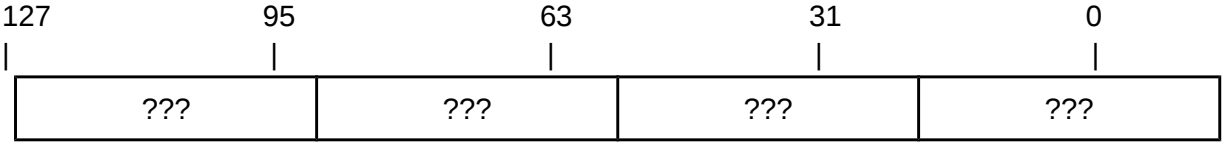
xmm0

↓mulps



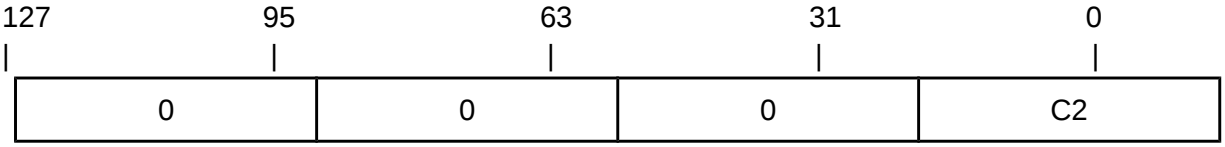
xmm0

•***Paso 2:*** obtener A1*X+C1, B1*Y, A2*X, B2*Y+C2. Para ello primero fue necesario acomodar los valores de C1 y C2 en un registro:



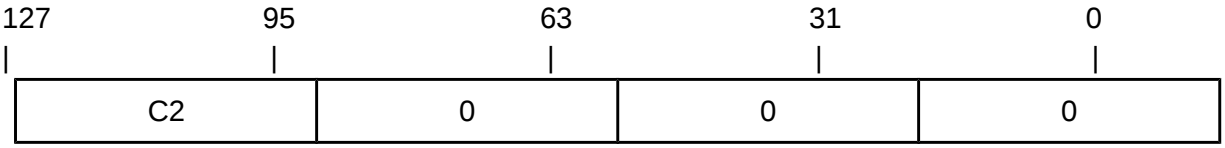
xmm4

↓movss



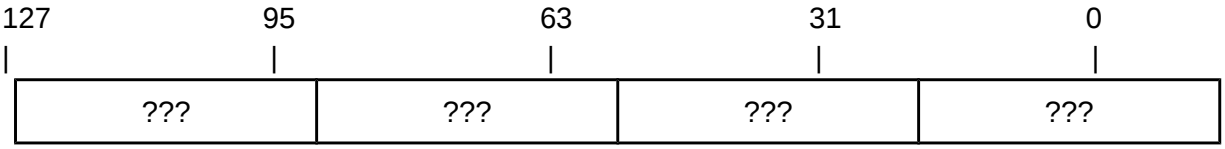
xmm4

↓shufps



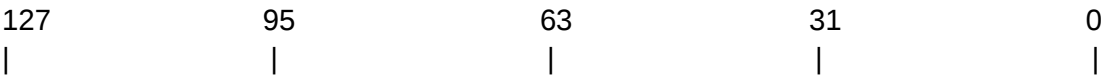
xmm4

Y en otro registro aparte:



xmm5

↓movss



0	0	0	C1

xmm5

Para guardar los valores en un único registro se operó como sigue:

127 95 63 31 0

0	0	0	C1

xmm5

127 95 63 31 0

C2	0	0	0

xmm4

↓movss

127 95 63 31 0

C2	0	0	C1

xmm4

Para finalizar este paso:

127 95 63 31 0

C2	0	0	C1

xmm4

127 95 63 31 0

B2*Y	A2*X	B1*Y	A1*X

xmm0

↓addps

127 95 63 31 0

B2*Y+C2	A2*X	BY*Y	A1*X+C1

xmm0

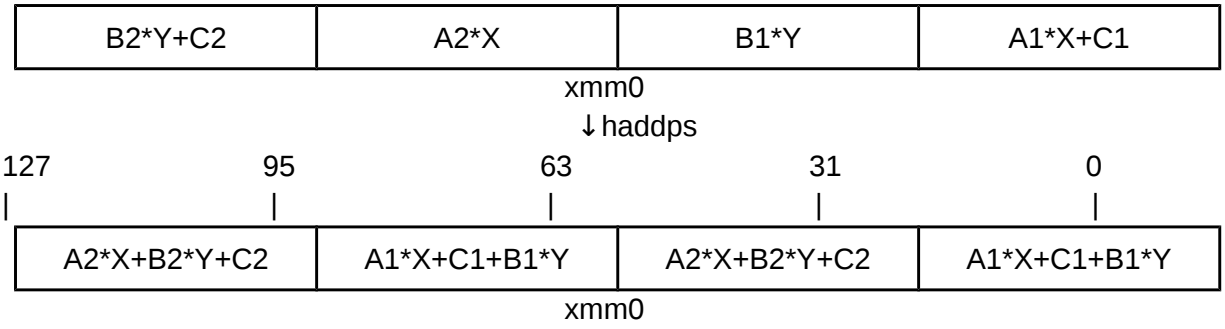
•***Paso 3:*** obtener el resultado final. El esquema a continuación detalla lo que sucede en esta etapa:

127 95 63 31 0

B2*Y+C2	A2*X	B1*Y	A1*X+C1

xmm0

127 95 63 31 0



•***Paso 4:*** devolver el nuevo valor de X e Y a la dirección de memoria que corresponde. Esto se completó mediante la utilización de la instrucción movsd.

6.Adición de opciones extra tales como “ESCALA” y “OFFSET” que modifican el resultado final del fractal.

Debido a que la no utilización de un escalado, obteníamos valores con diferencias muy pequeñas, y al redondearlos para obtener las coordenadas del píxel a marcar, se perdía la imagen característica del fractal. Por ello, Utilizamos en primera instancia una misma escala para todas los fractales. El resultado fueron imágenes fractales de distintos tamaños, y no centradas o cortadas. De allí surgió la necesidad de escalar y centrar las imágenes de manera individual, para una visualización óptima.

Dado que cada imagen emplea una entrada distinta, con la cantidad de iteraciones y sus ecuaciones iteradas respectivas, optamos por incluir la información de escalado y desplazamientos individuales, al comienzo de la entrada.

Esta nueva linea contiene la información: “escala” “desplazamiento horizontal” y “desplazamiento vertical”.

Estos valores fueron obtenidos por prueba y error, estimando que los mejores valores son:

input1: 40 200 10;
input2: 300 50 100;
input3: 350 25 25;

CONCLUSIÓN:

Como finalización de este informe podemos concluir que la temática dada en clase sobre instrucciones packed es solo una introducción la gran cantidad de instrucciones existentes; y es llamativo ver la formación de imágenes fractales a partir de la iteración de ecuaciones, y como estas van aumentando su calidad a medida que se incrementan las iteraciones.