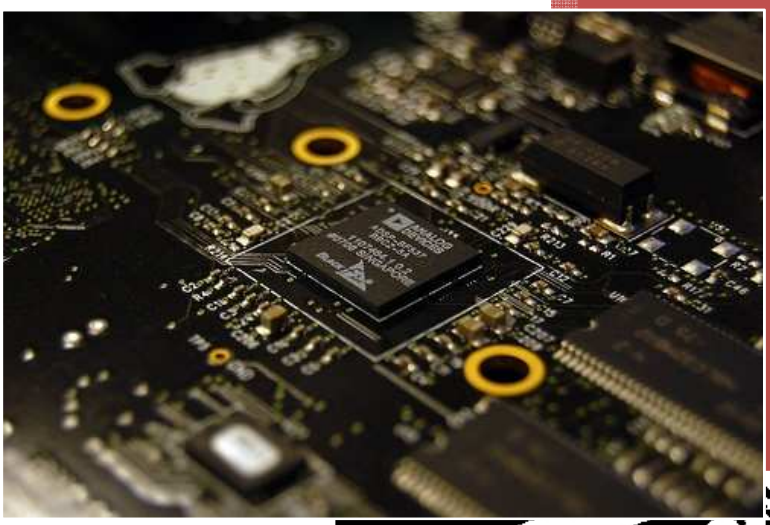
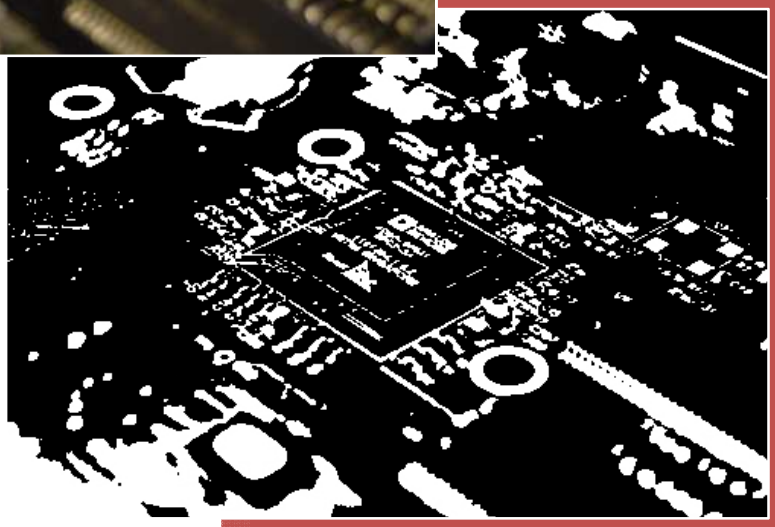
Universidad Tecnológica Nacional
Facultad Regional Buenos Aires

“Procesamiento de Imágenes en Tiempo Real”





Alumno: Pagani, Santiago José

Legajo: 07-117667-5

Índice

1 Introducción

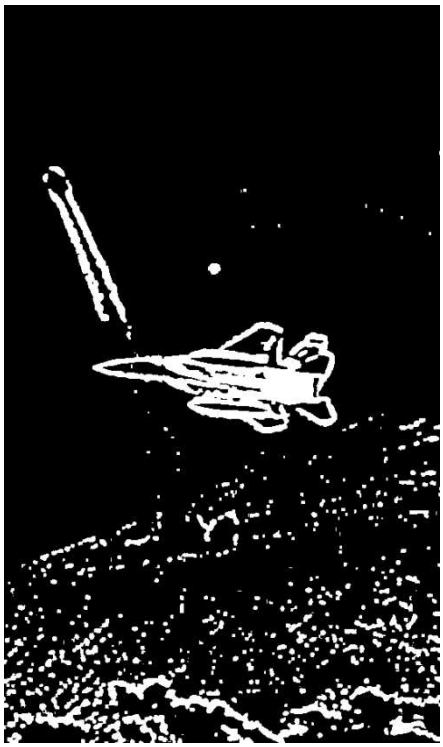
Podemos asegurar que hoy en día existen diversidad de aplicaciones en medicina o sistemas militares que trabajan con imágenes de video y necesitan un procesamiento de las mismas en tiempo real, ya sea con filtros de media para quitar ruido o con filtros de detección de bordes.

Al procesador de imágenes ingresa una señal de video compuesto proveniente de una cámara, se realiza el filtrado y egresa la señal de video compuesta que se puede visualizar en un televisor o proyector, o también enviar la información a otro sistema computacional; como por ejemplo a un sistema de angiografía digital.

Podemos explicar un uso práctico con el siguiente ejemplo:

Un sistema director de tiro de un buque militar posee un radar que detecta un avión acercándose, entonces apunta con la cámara a esa posición con una aproximación mínima y visualizando en la pantalla tan sólo como un punto. Luego el sistema centra al blanco, adiciona zoom, y corrige el centrado recursivamente hasta obtener la mayor precisión en cuanto a la posición del avión con el máximo acercamiento posible. Mediante la aplicación de un algoritmo de detección de bordes, se obtiene el contorno del avión permitiéndole a al sistema calcular el centroide del mismo para saber exactamente adonde apuntar.

Como este ejemplo hay muchos otros donde podemos apreciar la importancia y la necesidad de realizar filtros de video en tiempo real.



2 Teoría General de Filtros de Video

Como una señal de video es en realidad una secuencia de imágenes moviéndose a una frecuencia especifica, los filtros de video necesitan operar a una velocidad lo suficientemente rápida para seguirle el ritmo a la sucesión de imágenes de entrada. Por esto, es imperativo que los algoritmos de filtros de imágenes sean optimizados para ejecutarse en la menor cantidad posible de ciclos.

La convolución en dos dimensiones es una de las operaciones fundamentales en el procesamiento de imágenes. En este tipo de convoluciones, el cálculo realizado para filtrar un pixel dado es la suma de los valores de intensidad de los pixeles que lo rodean, previa multiplicación con una matriz (kernel) que le da un peso distinto a cada uno según su ubicación. Este kernel generalmente es pequeño y una elección común es que sea de 3 x 3, ya que es lo suficientemente grande para detectar bordes en una imagen, pero de un tamaño razonable para utilizar en cálculos computacionales.

La estructura básica del kernel de 3 x 3 se muestra en la siguiente figura:

H11	H12	H13
H21	H22	H23
H31	H32	H33

Algunos ejemplos de kernels con distintos efectos de filtrado:

0	1	0
1	1	1
0	1	0

Desenfoque

0	-1	0
-1	5	-1
0	-1	0

Nitidez

-2	-1	0
-1	1	1
0	1	2

Relieve

La descripción de alto nivel del algoritmo de convolución, puede resumirse en los siguientes pasos:

1- Ubicar el centro del kernel sobre un elemento de la imagen de entrada a procesar.
2- Multiplicar cada elemento del kernel de la matriz con el pixel correspondiente de la matriz de entrada.
3- Sumar el resultado de cada multiplicación en un único valor.
4- Ubicar el resultado de la suma en la matriz de imagen de salida.

La ecuación matemática que corresponde a la convolución entonces es:

$$g(x,y) = \sum_{i=-1}^{i=1} \sum_{k=-1}^{k=1} h(i,k) \cdot f(x-i, y-k)$$

Donde:

x: $Columna\ del\ pixel\ a\ filtrar$

y: $Fila\ del\ pixel\ a\ filtrar$

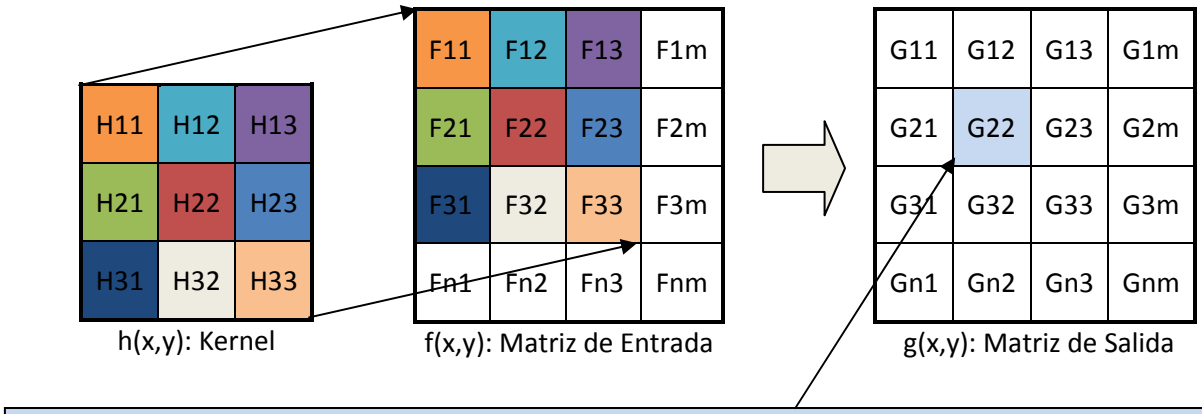
g: $Matriz\ de\ la\ imagen\ de\ salida$

$g(x,y)$: $Pixel\ de\ salida\ de\ la\ imagen\ filtrada, ubicado\ en\ la\ columna\ 'x', fila\ 'y'$

h: $Kernel\ de\ 3x3$

f: $Matriz\ de\ la\ imagen\ de\ entrada$

Representación del proceso de una convolución en dos dimensiones:



$$G22 = H11 \cdot F11 + H12 \cdot F12 + H13 \cdot F13 + H21 \cdot F21 + H22 \cdot F22 + H23 \cdot F23 + H31 \cdot F31 + H32 \cdot F32 + H33 \cdot F33$$

Luego de obtener el resultado de un pixel y ubicarlo en la imagen de salida, la matriz se mueve un elemento a la derecha. Al llegar a la última columna, se vuelve a ubicar la matriz en la primera columna, pero en la fila siguiente. Como resultado, la parte útil de la imagen de salida es reducida un elemento a lo largo de los bordes de la imagen.

Teniendo en cuenta que una imagen NTSC (parte activa sin sincronismo) tiene 720 x 507 pixeles a 30 cuadros por segundo, tenemos un total de 10.951.200 pixels/segundo (en blanco y negro). Considerando que deben realizarse 9 multiplicaciones y 8 acumulaciones, y olvidándose momentáneamente del acomodamiento de los buffers de memoria utilizados, se requiere procesar los datos a: 9 x 10.951.200 p/s = 98,56 MIPS.

Los filtros que necesiten trabajar con el color de la imagen, serán implementados con instrucciones que trabajen en paralelo, aprovechando las prestaciones del DSP y así no duplicar el tiempo de procesamiento.

3 Filtro de Detección de Bordes

Muchos efectos de video pueden obtenerse con una única convolución de dos dimensiones: algunas que solo convolucionan la información de color y otros que solo procesan la de luminancia.

Otros tipos de filtros necesitan realizar dos convoluciones por cada pixel: una convolución que trabaja con el color y otra con la luminancia.

Existen también casos, como ser la detección de bordes, donde se requieren más de una convolución: siempre trabaja con la luminancia, pero utiliza más de un kernel para cada pixel de salida.

La idea básica detrás de la detección de bordes es encontrar lugares en una imagen donde la intensidad cambia rápidamente.

Basándose en esta idea, un detector de bordes puede basarse en una técnica que localice lugares donde la primera derivada de la intensidad es mayor en magnitud que un nivel mínimo determinado. Si no, puede basarse en el criterio de buscar lugares donde la segunda derivada de la intensidad tenga un cruce por cero.

Para la detección de bordes de nuestro Procesador de Imágenes se ha elegido el algoritmo de Prewitt, gracias a la simplicidad de su implementación computacional, como también a sus buenos resultados.

Cuando se usa el algoritmo de Prewitt, la imagen es convolucionada con un set de kernels de 3 x 3. Existe un kernel para detectar bordes en cada dirección, siendo lo más común el tener 8 (cada 45°).

1	1	1
0	0	0
-1	-1	-1

Det. 0°

0	1	1
-1	0	1
-1	-1	0

Det. 45°

-1	0	1
-1	0	1
-1	0	1

Det. 90°

-1	-1	0
-1	0	1
0	1	1

Det. 135°

-1	-1	-1
0	0	0
1	1	1

Det. 180°

0	-1	-1
1	0	-1
1	1	0

Det. 225°

1	0	-1
1	0	-1
1	0	-1

Det. 270°

1	1	0
1	0	-1
0	-1	-1

Det. 315°

Sin embargo, esto requeriría de 8 convoluciones para cada pixel y no se obtienen resultados notoriamente más satisfactorios que utilizando tan solo algunos de los kernels. En consecuencia, se utilizan dos kernels, siendo uno sensible a los ejes verticales y otro sensible a los ejes horizontales.

-1	0	1
-1	0	1
-1	0	1

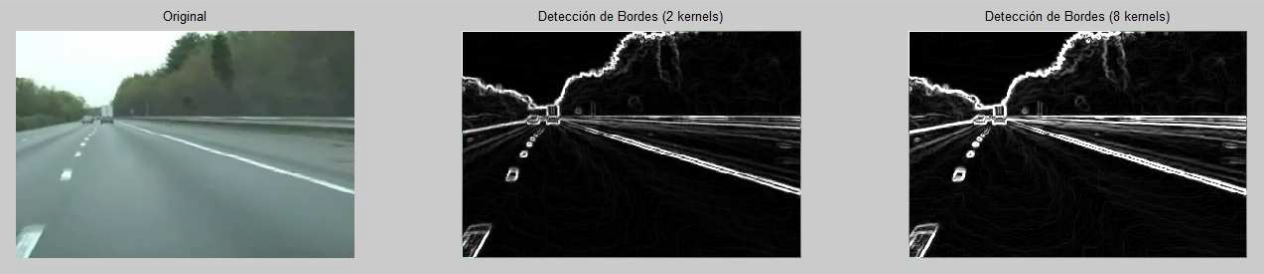
Detección Horizontal

-1	-1	-1
0	0	0
1	1	1

Detección Vertical

Lo correcto es ahora realizar la suma cuadrática del resultado de estas dos convoluciones, pero nuevamente verificando los resultados se concluye que la suma de los módulos es una aproximación lo suficientemente acertada, y puede realizarse en muchos menos ciclos de máquina.

A continuación se muestra una detección de bordes realizada con 2 kernels y la suma de sus módulos, y otra detección de la misma imagen realizada con los 8 kernels y la suma cuadrática. Observando los resultados, puede aceptarse que las diferencias en ambas imágenes son despreciables para la percepción del ojo humano.



La ecuación del filtro de detección de bordes de Prewitt resulta entonces:

$$g1(x,y) = \sum_{i=-1}^{i=1} \sum_{k=-1}^{k=1} h1(i,k) \cdot f(x-i, y-k)$$

$$g2(x,y) = \sum_{i=-1}^{i=1} \sum_{k=-1}^{k=1} h2(i,k) \cdot f(x-i, y-k)$$

$$g(x,y) = |g1(x,y)| + |g2(x,y)|$$

4 Implementación en Matlab

Para verificar la operación de los algoritmos de los filtros de video y los resultados obtenidos con distintos kernels, se procedió a escribir los mismos en Matlab y probarlos con varios videos.

El Matlab nos proporciona muchas herramientas para este propósito, incluida la facilidad de decodificar archivos .avi y permitirnos visualizar los mismos luego del procesamiento adecuado.

En el script de Matlab puede observarse la definición de varios kernels, ya que como resultado de esta prueba lo que se observara será un mismo video filtrado con varios efectos distintos, todas en simultaneo. Para la elección de los kernels se tomaron los más comunes encontrados en las bibliografías populares. Hay que tener en cuenta que los coeficientes de los mismos se seleccionan básicamente en forma empírica, experimentando con valores hasta obtener los efectos visuales deseados.

El proceso de filtrado de video, utilizando las herramientas del entorno que nos permiten trabajar con archivos .avi, se realiza en varios pasos:

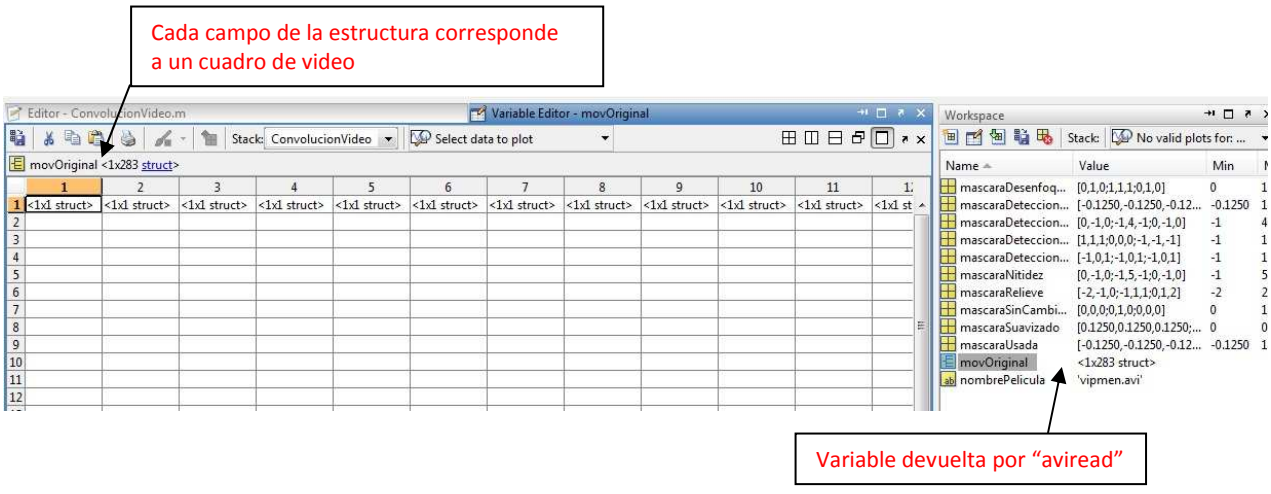
- Se definen los filtros que se van a utilizar.

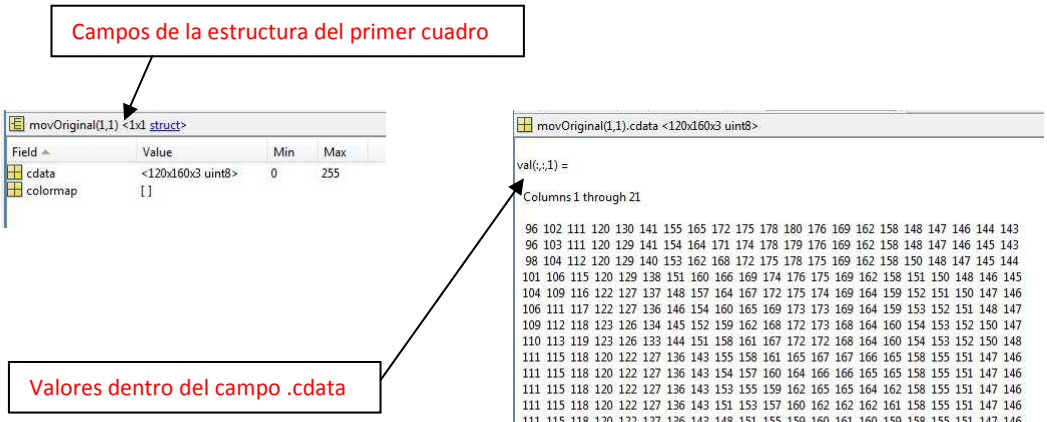
```
% Defino las mascaras de los filtros
mascaraSinCambios = [0 0 0; 0 1 0; 0 0 0];
mascaraDeteccionBordesVerticales = [-1 0 1; -1 0 1; -1 0 1];
mascaraDeteccionBordesHorizontales = [1 1 1; 0 0 0; -1 -1 -1];
```

- Se utiliza la función “aviread” para leer todos los cuadros del archivo.

```
% Defino los parámetros del video a mostrar
nombrePelicula = 'vipmen.avi';
movOriginal = aviread(nombrePelicula);
```

- El resultado de esta función es un array de estructuras. El largo del array equivale a la cantidad de cuadros en el video, y cada estructura contiene la información de cada campo. En el campo .cdata se encuentra la información de cada pixel, ordenada en forma de un array de dos dimensiones con un tamaño igual al del cuadro.





- Se itera por todos los campos del video, recorriendo todos los valores del array resultado de “aviread”.
- Según el tipo de archivo leído, la información en .cdata tiene un formato u otro. Si el formato es RGB, se deja como esta. Caso contrario, se realiza una conversión a RGB.

```
% Itero todos los cuadros del video
for cuadro = 1:CANTIDADcAMPOS
    % Según el tipo de imagen, obtengo el cuadro en formato RGB
    if movInfo.ImageType(1) == 't'
        frameRGB = movOriginal(cuadro).cdata;
    else
        % Paso el formato de los cuadros de color indexado a RBG
        frameRGB = ind2rgb(movOriginal(cuadro).cdata,movOriginal(cuadro).colormap);
    end
```

- A continuación se transforma de RGB a NTSC, para poder discriminar la información de luminancia de la de crominancia.

```
% Paso el formato de RGB a NTSC para obtener el valor de luminancia
frameNTSC = rgb2ntsc(frameRGB);
```

- Luego, se itera a través de todas las filas y columnas del cuadro realizando la convolución de dos dimensiones para todas las mascaras distintas, y los promedios para los filtros de media.

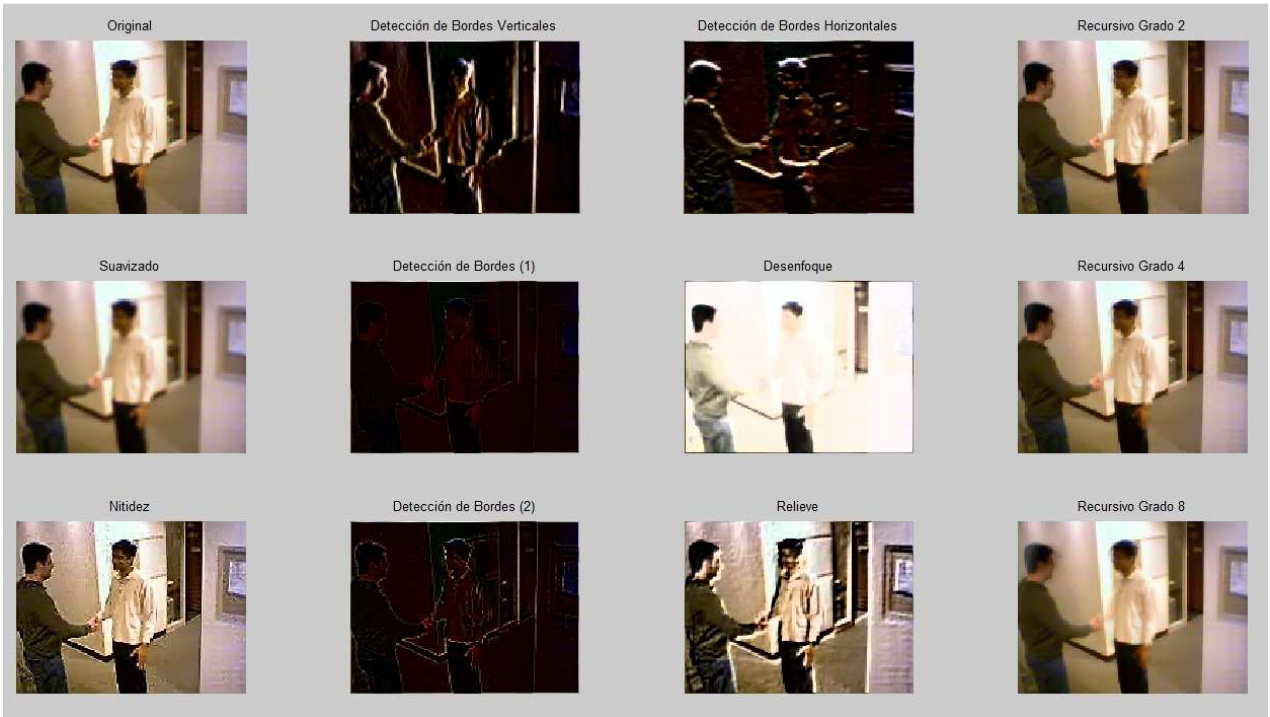
```
% Algoritmo de convolución de dos dimensiones
for fila = 2:CANTIDADfILAS-1
    for columna = 2:CANTIDADcOLUMNAS-1
        actual = zeros(3,3,'double');
        filtrado_Suavizado = 0;
        for wMasc = 1:3
            for hMasc = 1:3
                actual(wMasc,hMasc) = frameNTSC(fila + wMasc - 2,columna + hMasc - 2);
                filtrado_Suavizado = filtrado_Suavizado + actual(wMasc,hMasc)*mascaraSuavizado(wMasc,hMasc);
            end
        end
        frameNTSC_Suavizado(fila,columna) = filtrado_Suavizado;
    end
end
```

- Como paso siguiente, vuelve a pasarse el resultado a RGB y a formato .avi, para poder visualizarse en la pantalla.

```
% Luego de realizar el filtrado, vuelvo a pasar el formato a RGB
 frameRGB_DeteccionBordesVerticales = ntsc2rgb(frameNTSC_DeteccionBordesVerticales);
 % Y vuelvo a pasarlo a formato de película
 movFiltrado_DeteccionBordesVerticales(cuadro).cdata = cast((frameRGB_DeteccionBordesVerticales*256),'uint8');
```

- Por último, se entra en un bucle que muestra el video original y todas las salidas de video filtradas con las distintas variantes.

```
subplot(3,4,1); h1 = image(movOriginal(1).cdata); title('Original'); axis off; axis image;
subplot(3,4,2); h2 = image(movFiltrado_DeteccionBordesVerticales(1).cdata); title('Bordes Verticales'); axis off; axis image;
subplot(3,4,3); h3 = image(movFiltrado_DeteccionBordesHorizontales(1).cdata); title('Bordes Horizontales'); axis off; axis image;
subplot(3,4,4); h4 = image(movRecursivo2(1).cdata); axis off; title('Recursivo Grado 2'); axis image;
subplot(3,4,5); h5 = image(movFiltrado_Suavizado(1).cdata); title('Suavizado'); axis off; axis image;
subplot(3,4,6); h6 = image(movFiltrado_DeteccionBordes1(1).cdata); title('Detección de Bordes (1)'); axis off; axis image;
subplot(3,4,7); h7 = image(movFiltrado_Desenfoque(1).cdata); title('Desenfoque'); axis off; axis image;
subplot(3,4,8); h8 = image(movRecursivo4(1).cdata); axis off; title('Recursivo Grado 4'); axis image;
subplot(3,4,9); h9 = image(movFiltrado_Nitidez(1).cdata); title('Nitidez'); axis off; axis image;
subplot(3,4,10); h10 = image(movFiltrado_DeteccionBordes2(1).cdata); title('Detección de Bordes (2)'); axis off; axis image;
subplot(3,4,11); h11 = image(movFiltrado_Relieve(1).cdata); axis off; title('Relieve'); axis image;
subplot(3,4,12); h12 = image(movRecursivo8(1).cdata); axis off; title('Recursivo Grado 8'); axis image;
while 1 == 1
    for fr = 1 : length(movOriginal)
        set(h1, 'Cdata', movOriginal(fr).cdata);
        set(h2, 'Cdata', movFiltrado_DeteccionBordesVerticales(fr).cdata);
        set(h3, 'Cdata', movFiltrado_DeteccionBordesHorizontales(fr).cdata);
        set(h4, 'Cdata', movRecursivo2(fr).cdata);
        set(h5, 'Cdata', movFiltrado_Suavizado(fr).cdata);
        set(h6, 'Cdata', movFiltrado_DeteccionBordes1(fr).cdata);
        set(h7, 'Cdata', movFiltrado_Desenfoque(fr).cdata);
        set(h8, 'Cdata', movRecursivo4(fr).cdata);
        set(h9, 'Cdata', movFiltrado_Nitidez(fr).cdata);
        set(h10, 'Cdata', movFiltrado_DeteccionBordes2(fr).cdata);
        set(h11, 'Cdata', movFiltrado_Relieve(fr).cdata);
        set(h12, 'Cdata', movRecursivo8(fr).cdata);
        pause(1/movInfo.FramesPerSecond);
    end
end
```



Resultados de filtros realizados en MATLAB

5 Configuración de Hardware

Para cumplir con las prestaciones requeridas por el proyecto se debe:

- Realizar una conversión analógica/digital del video.
- Interpretar correctamente la información y ubicación de cada pixel de la imagen.
- Realizar el algoritmo del filtro de video deseado.
- Realizar una conversión digital/analógica del video procesado.

El procesamiento de imágenes se implementa mediante un kit de evaluación de la familia Blackfin de Analog Devices, principalmente gracias a la facilidad de contar con varios de ellos en el departamento de electrónica de la facultad.

Los kits mencionados utilizan para la entrada de video el decodificador ADV7183, y se comunican con el mismo por medio de la interfaz “Parallel Peripheral Interface” (Interfaz de periféricos paralela o PPI) configurada como entrada.

A su vez, para la salida de video utilizan el codificador ADV7171, comunicándose con este por medio de la interfaz PPI configurada como salida.

Tanto el codificador como el decodificador necesitan un tiempo de arranque luego de su habilitación. Además, el decodificador requiere sincronizarse con la señal de sincronismo vertical del video entrante.

Estas necesidades obligan a utilizar un kit que cuente con un procesador que tenga dos PPI, como ser el ADSP-BF561. Debido a que este kit no se encuentra en el departamento de la facultad, se opto por utilizar una alternativa de compromiso: usar dos kits con el ADSP-BF533 y comunicarlos mediante un puerto serie sincrónico de alta velocidad (SPORT).

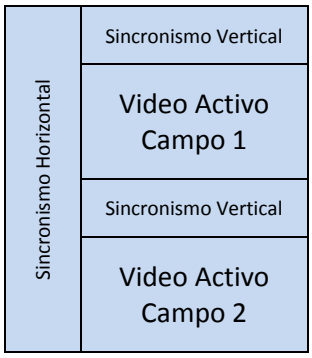
En resumen, el sistema realizará las siguientes operaciones:

- Kit de Entrada de Video y Procesamiento de Imágenes:
  - Configura el decodificador de video para que se comunique por la interfaz PPI, configurando la misma como entrada.
  - Interpreta la información recibida y la ubicación de cada pixel de la imagen, ya que el decodificador de video nos entrega el video en forma entrelazada (tal cual lo requiere un televisor o proyector para su visualización).
  - Realiza el algoritmo del filtro de video deseado.
  - Envía el video procesado por el SPORT al otro kit.

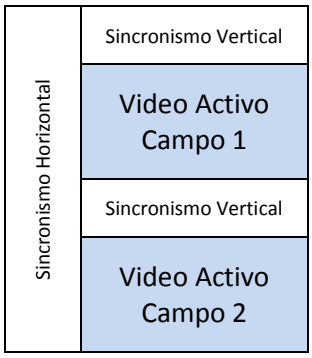
- Kit de Salida de Video:
  - Recibe por el SPORT el video procesado.
  - Reacomoda en memoria el video recibido para su correcta visualización, ya que para bajar los tiempos de transmisión se envío solo la imagen activa sin sincronismos.
  - Lo envía al codificador de video para su visualización en tiempo real.

6 Interpretación de una señal de video

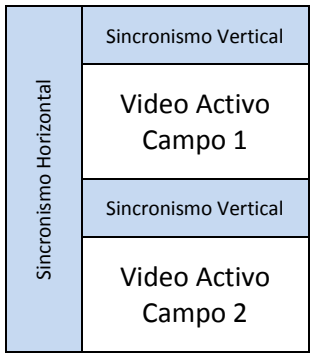
El decodificador de video ADV7183 es capaz de proporcionar tres modos de salida: uno en el que se recibe toda la información de video (señal completa), otro en el que se recibe solo los campos activos, y otro en el que se recibe solamente la información de sincronismo.



Señal Completa



Campos Activos



Sincronismos

En formato NTSC la señal completa tiene un tamaño de 858 x 525 (columnas x filas), mientras que solo los campos activos ocupan 720 x 507 pixeles. Cada pixel cuenta con dos bytes de información empaquetados en 16 bits: un byte para la información de luminancia y un byte para la información de crominancia.

Un punto muy importante a tener en cuenta, es que sí bien la tasa de video es de 30 cuadros por segundo, cualquier televisor o proyector recrea la imagen en forma entrelazada. O sea, se dibujan en orden las líneas impares dejando un espacio entre líneas hasta llegar al final de la pantalla, para luego volver al principio y dibujar las líneas pares en los espacios libres. Cada una de estas pasadas es lo que se denomina “campo”, y puede observarse que eso es exactamente lo que nos entrega la PPI: un campo con las líneas impares y otro con las líneas pares.

El entrelazado del video es una circunstancia muy importante que debe tenerse en cuenta a la hora de realizar algoritmos de filtrado, ya que las líneas impares se ubicaran en posiciones consecutivas de memoria, pero los algoritmos deben analizar el video de forma desentrelazada debido a la necesidad de operar con líneas consecutivas.

Para poder simplificar la implementación de los filtros se optó por configurar el PPI para recibir solo los campos activos. De esta forma, no es necesario ignorar el sincronismo y se puede filtrar los cuadros completos.

Esto genera una pequeña problemática, pero de fácil solución: el DSP que se encarga de entregar la señal de video ya filtrada debe generar los sincronismos verticales y horizontales, así como también ubicar correctamente en memoria la información recibida por el otro DSP.

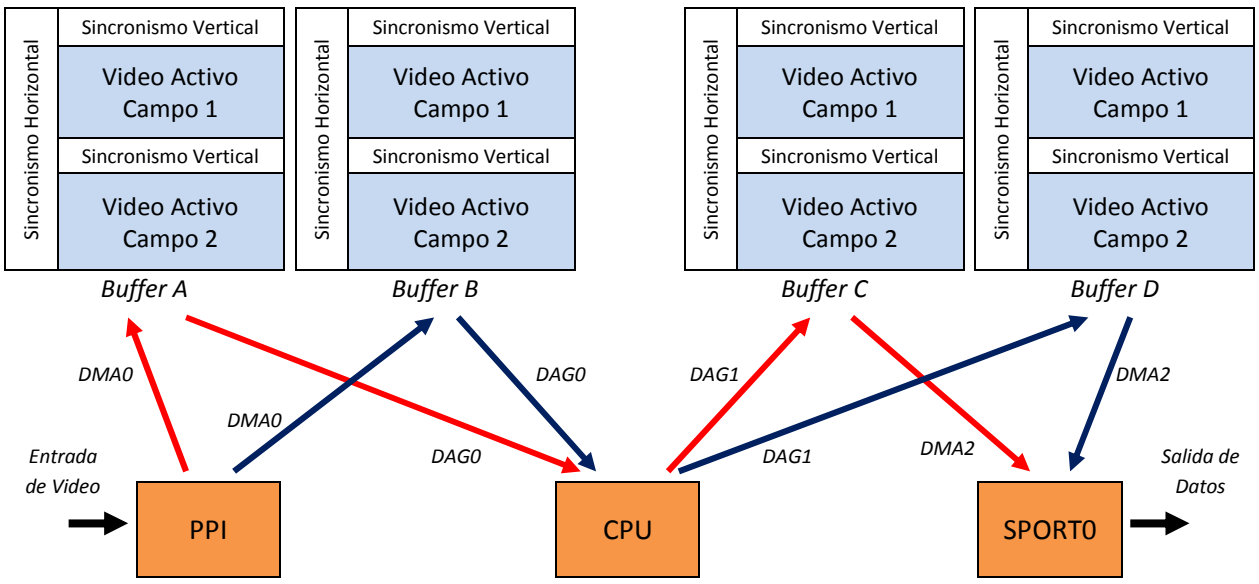
7 Manejo de Buffers de Entrada y Salida

Para la implementación del programa del DSP se utilizan cuatro buffers de memoria en el kit de entrada y cuatro buffers en el kit de salida.

En el kit de entrada, los buffers se manejan de la siguiente manera:

- El DMA0 maneja los buffers de entrada del PPI y el DMA2 los buffers de salida del SPORT.
- Se obtiene un cuadro y se almacena el mismo en el buffer A, y se le indica al DMA0 que el siguiente cuadro lo almacene en el buffer B.
- En este momento entra en funcionamiento el filtro de video, el cual accede a la imagen a través de los "Generadores de Direcciones de Datos" (DAGs). El algoritmo almacena el resultado del filtrado (también vía DAG) en el buffer C y se le indica entonces al DMA2 que utilice este buffer para enviarlo por el SPORT.
- Cuando llegue un nuevo cuadro al buffer B, se volverá a indicarle al DMA0 que ahora almacene el siguiente cuadro en el buffer A, y así sucesivamente.
- El filtro de video almacenara el resultado del filtrado en el buffer D y se le indicará al DMA2 que ahora utilice este buffer para enviarlo por el SPORT.

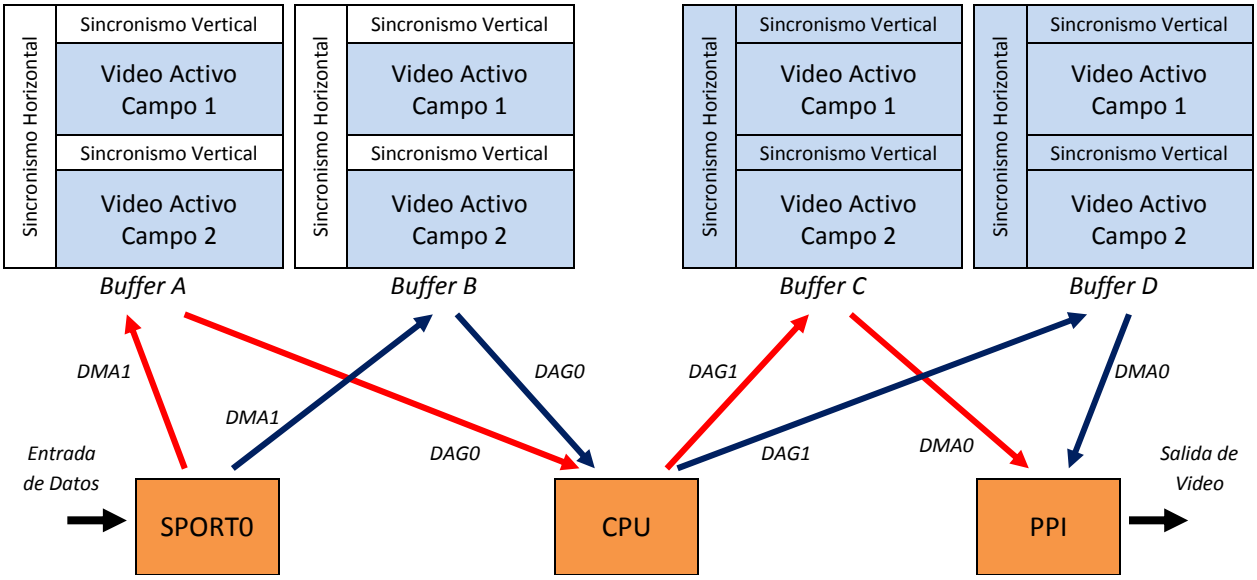
Gráficamente, este manejo puede interpretarse de la siguiente manera:



En el kit de salida el manejo es similar:

- El DMA1 maneja los buffers de entrada del SPORT y el DMA0 los buffers de salida del PPI.
- Al inicio del programa se arma una imagen patrón en los buffer C y D. La misma es una señal de calibración de barras de colores que ya incluye los sincronismos verticales y horizontales.
- Se obtiene un cuadro por el SPORT en el buffer A.
- Por medio de los DAGs se ubica la imagen recibida en el buffer C, teniendo en cuenta que se recibió una imagen de solo campos activos, y en el buffer C se tenía una imagen con la señal completa.
- Se le indica entonces al DMA1 que el siguiente cuadro lo almacene en el buffer B, y al mismo tiempo se le indica al DMA0 que utilice el buffer C como salida para enviar al codificador de video.
- El ciclo vuelve a repetirse intercambiando los buffers.

Gráficamente, este manejo puede interpretarse de la siguiente manera:



Por el tamaño de la imagen a procesar, los buffers deben ubicarse necesariamente en la memoria externa con la que cuenta el kit. La ubicación de los mismos debe elegirse con cuidado, para que no se generen demoras excesivas entre lecturas y escrituras, ya que el controlador de la SDRAM tiene cuatro bancos internos y el acceso simultáneo solo puede realizarse entre distintos bancos.

De esta manera, ubicamos cada buffer en un banco interno de la SDRAM.

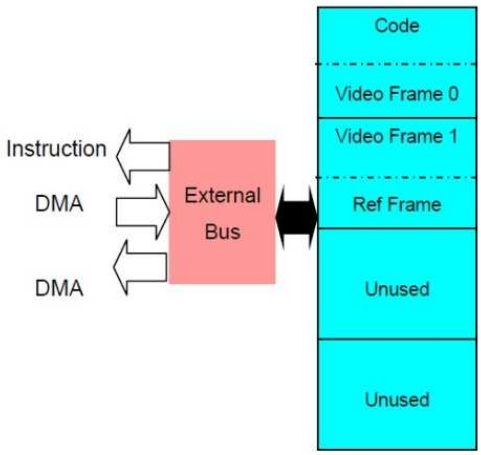


Figure 3. Un-Optimized SDRAM Memory Allocation

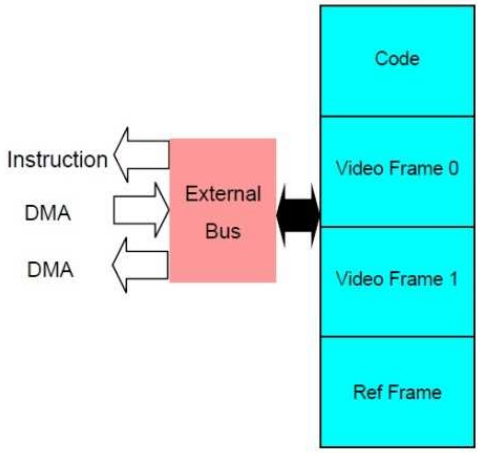


Figure 4. Optimized SDRAM Memory Allocation

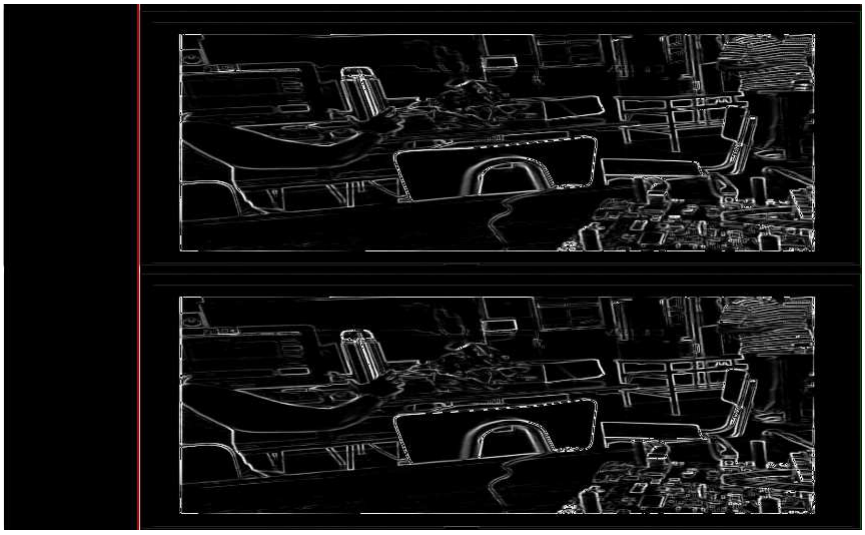
8 Simulación en Visual DSP++

El entorno de desarrollo Visual DSP++ proporciona la herramienta “Image Viewer” que es de vital importancia para la implementación de aplicaciones que realizan filtros de video.

Gracias al mismo, podemos visualizar en la PC la imagen que ingresa al decodificador de video, procesar la misma y observar cómo se vería el resultado en una televisión.

Lo único que debe tenerse en cuenta, es que la imagen se visualiza igual como se recibe: entrelazada. Por este motivo, se observará el campo 1 en la parte superior, y el campo 2 en la inferior.

Video visualizado con el Image Viewer antes de filtrar



Video visualizado con el Image Viewer luego de aplicar filtro de detección de bordes

9 Implementación en el kit de desarrollo

Como primer paso y para acotar la posibilidad de errores se escribió un programa con toda la configuración del procesador y DMA, logrando así visualizar video en tiempo real pero sin filtrar. En este punto surgió el primer impedimento: la comunicación por el SPORT no es lo suficientemente rápida, teniendo como resultado un flujo de video en el cual se pierden muchos cuadros dando un efecto de una secuencia de fotos en lugar de video. Esto puede apreciarse matemáticamente:

La tasa de recepción de video es de:

$$720\ [columnas]x\ 507\ [líneas]x\ 16[bits] =\ 5.840.640\ {}^{bits}/_{cuadro}$$

Sabiendo que en NTSC se reciben 30 cuadros por segundo, la información de video que se maneja es de:

$$5.840.640\ {}^{bits}/_{cuadro}\ x\ 30\ {}^{cuadros}/_{seg} = 175.219.200\ {}^{bits}/_{seg} = 21.902.400\ {}^{bytes}/_{seg}$$

Pero el SPORT se opera en su máximo bitrate, siendo el mismo: $xxxxxx\ {}^{bits}/_{seg}$

Lo que nos da una tasa máxima de transferencia de datos de:

$$xxxxxx\ {}^{bits}/_{seg}\ x\ 5.840.640\ {}^{bits}/_{cuadro} = yyyyy\ {}^{cuadros}/_{seg}$$

Se realizó un programa de prueba cuya función es recibir un único cuadro y retransmitirlo indefinidamente por el SPORT. De esta manera logró asilarse el periférico para evaluar en forma práctica la velocidad de transmisión del mismo.

Grafico del osciloscopio con tiempo de transmisión del SPORT

El algoritmo de filtrado es el punto crucial de todo el sistema. Por cada pixel recibido debe realizarse una doble convolución: luminancia y color para filtros generales, y luminancia con dos kernels para detección de bordes).

A continuación se describe la implementación del algoritmo de detección de bordes:

- Se configuran los DAGs para acceder a los buffer de entrada y salida, así como también al kernel.

```
Deteccion_De_Bordes:
        // Configuro el buffer circular I0 para apuntar al frame original
        // Verifico cual es el buffer a utilizar
        P2.L = LO(bufferActual);
        P2.H = HI(bufferActual);
        R1.L = W[P2];
        CC = bittst(R1,0);
        R1.H = HI(FRAME_ORIGINAL_1);
        R1.L = LO(FRAME_ORIGINAL_1);
        IF !CC jump ELEGIR_BUFFER_ENTRADA_DETECCION_BORDES;
        R1.H = HI(FRAME_ORIGINAL_0);
        R1.L = LO(FRAME_ORIGINAL_0);
ELEGIR_BUFFER_ENTRADA_DETECCION_BORDES:
        R2.H = HI(PIXELS_X_FRAME);
        R2.L = LO(PIXELS_X_FRAME);
        I0 = R1; B0 = R1; L0 = R2; M0 = 2;

        // Configuro el buffer circular I1 para apuntar al frame resultado
        // Verifico cual es el buffer a utilizar
        P2.L = LO(bufferActual);
        P2.H = HI(bufferActual);
        R1.L = W[P2];
        CC = bittst(R1,0);
        R1.H = HI(FRAME_FILTRADO_1);
        R1.L = LO(FRAME_FILTRADO_1);
        IF !CC jump ELEGIR_BUFFER_SALIDA_DETECCION_BORDES;
        R1.H = HI(FRAME_FILTRADO_0);
        R1.L = LO(FRAME_FILTRADO_0);
ELEGIR_BUFFER_SALIDA_DETECCION_BORDES:
        R2.H = HI(PIXELS_X_FRAME);
        R2.L = LO(PIXELS_X_FRAME);
        I1 = R1; B1 = R1; L1 = R2; M1 = LINE_INCREMENT;

        // Configuro el buffer circular I2 para apuntar a la matriz de
        // convolucion XY
        R1.H = HI(mascaraDeteccionBordes);
        R1.L = LO(mascaraDeteccionBordes);
        R2.H = HI(MASCARA_BORDES_LENGTH);
        R2.L = LO(MASCARA_BORDES_LENGTH);
        I2 = R1; B2 = R1; L2 = R2; M2 = 4;

        // Lo uso para poder recorrer la matriz de dos dimensiones, campo a campo
        M3.H = HI(FRAME_INCREMENT);
        M3.L = LO(FRAME_INCREMENT);
```

- Se copia la primera línea tal cual está, ya que el algoritmo no modifica los pixel del borde del cuadro debido a que necesita el entorno de cada pixel y los bordes no lo tienen.

```
        // Se usan para hacer las mascaras de luminancia y color
        R3.L = 0x00FF;
        R4.L = 0x00FF;
        R5.L = 0x0080; // Si no se toca mas R5, no se le agrega el color

        // Se copia la primer linea tal cual esta
        P2.H = HI(Line_Length/2);
        P2.L = LO(Line_Length/2);
        LSETUP(First_Line_Loop_Start, First_Line_Loop_End) LC1 = P2;
        First_Line_Loop_Start:
                R1 = [I0++];
        First_Line_Loop_End:
                [I1++] = R1;
```

- Se inicia el bucle que recorre cada una de las 507 líneas, y se precargan algunos valores necesarios para el procesamiento de cada línea.

```
I1 += M3;
I1 -= M1;
I0 += M3;
I0 -= M1;
// Al salir de aca, dejo a I0 apuntando a F21 y a I1 apuntado a G21

//Carga H11x en R2.H, carga H11y en R2.L
R2 = [I2++];

// Se inicia el loop que recorre las lineas del frame
P2.H = HI(Frame_Length-2);
P2.L = LO(Frame_Length-2);
LSETUP(Frame_Loop_Start, Frame_Loop_End) LC0 = P2;
Frame_Loop_Start:
        // Copia el primer pixel de la linea tal cual esta
        R1.L = W[I0++];
        W[I1] = R1.L;
        I1+= M0;
        I0 += M3;
        I0 -= M1;
        I0 -= M0;
        // Aca tengo a I0 apuntando al pixel 138 de la primer linea de la imagen de entrada,
        // y a I1 apuntando al pixel 139 de la segunda linea de la imagen de salida.
        // O sea, F11 y G22 del campo activo, respectivamente.

        // Carga F11 en R1.H
        R1.L = W[I0++];
        R1 >>= 8; R1 = R1 & R3;
```

- Se inicia el bucle que recorre los pixeles activos de cada linea y se ejecuta el corazón del algoritmo.

```
        // Se inicia el loop que recorre los pixeles activos dentro de la linea
        P2.H = HI(Active_Line_Length);
        P2.L = LO(Active_Line_Length);
        LSETUP(Active_Line_Loop_Start, Active_Line_Loop_End) LC1 = P2;
        Active_Line_Loop_Start:

                // En A0 se va a acumular G22x y en A1 se acumula G22y.
                // Cuando entra al loop F11, H11x y H11y ya estan cargados en R1 y R2

                // Hago: G22x = F11 * H11x, G22y = F11 * H11y
                // Carga F12 en R1.H: carga H12x en R2.H,carga H12y en R2.L
                A0 = R1.L * R2.H, A1 = R1.L * R2.L || R1.L = W[I0++] || R2 = [I2++];
                // Esto se hace por cada byte leido, ya que en el empaquetado se recibe la
                // información luminancia en el byte alto, pero se necesita en el bajo para
                // su correcto calculo matemático
                R1 >>= 8; R1 = R1 & R3; NOP;

                // Hago: G22x += F12 * H12x, G22y += F12 * H12y
                // Carga F13 en R1.H: carga H13x en R3.H,carga H13y en R2.L
                A0 += R1.L * R2.H, A1 += R1.L * R2.L || R1.L = W[I0++] || R2 = [I2++];
                R1 >>= 8; R1 = R1 & R3;

                // Hago: G22x += F13 * H13x, G22y += F13 * H13y
                // Carga F21 en R1.H: carga H21x en R3.H,carga H21y en R2.L
                I0 += M3;
                I0 -= M0;
                I0 -= M2;
                A0 += R1.L * R2.H, A1 += R1.L * R2.L || R1.L = W[I0++] || R2 = [I2++];
                R1 >>= 8; R1 = R1 & R3; NOP;

                // Hago: G22x += F21 * H21x, G22y += F21 * H21y
                // Carga F22 en R1.H: carga H22x en R3.H,carga H22y en R2.L
                A0 += R1.L * R2.H, A1 += R1.L * R2.L || R1.L = W[I0++] || R2 = [I2++];
                // Toma el valor de F22 para luego sumar el color
                //R5 = R1 & R4; // Si no comento esto, se le agrega el color que tenia
                R1 >>= 8; R1 = R1 & R3; NOP;
```

```
                // Hago: G22x += F22 * H22x, G22y += F22 * H22y
                // Carga F23 en R1.H: carga H23x en R3.H,carga H23y en R2.L
                A0 += R1.L * R2.H, A1 += R1.L * R2.L || R1.L = W[I0++] || R2 = [I2++];
                R1 >>= 8; R1 = R1 & R3;

                // Hago: G22x += F23 * H23x, G22y += F23 * H23y
                // Carga F31 en R1.H: carga H31x en R3.H,carga H31y en R2.L
                I0 -= M3;
                I0 += M1;
                I0 -= M0;
                I0 -= M2;
                A0 += R1.L * R2.H, A1 += R1.L * R2.L || R1.L = W[I0++] || R2 = [I2++];
                R1 >>= 8; R1 = R1 & R3; NOP;

                // Hago: G22x += F31 * H31x, G22y += F31 * H31y
                // Carga F32 en R1.H: carga H32x en R3.H,carga H32y en R2.L
                A0 += R1.L * R2.H, A1 += R1.L * R2.L || R1.L = W[I0++] || R2 = [I2++];
                R1 >>= 8; R1 = R1 & R3; NOP;

                // Hago: G22x += F32 * H32x, G22y += F32 * H32y
                // Carga F33 en R1.H: carga H33x en R3.H,carga H33y en R2.L
                A0 += R1.L * R2.H, A1 += R1.L * R2.L || R1.L = W[I0++] || R2 = [I2++];
                R1 >>= 8; R1 = R1 & R3;

                // Hago: G22x += F33 * H33x, G22y += F33 * H33y
                // Carga F12 en R1.H: carga H11x en R3.H,carga H11y en R2.L (para
                // el calculo del siguiente pixel: G23.
                I0 -= M1;
                I0 -= M2;
                A0 += R1.L * R2.H, A1 += R1.L * R2.L || R1.L = W[I0++] || R2 = [I2++];
                A0 = A0 >>> 1; A1 = A1 >>>1;
                R1 >>= 8; R1 = R1 & R3;

                // Se obtiene el modulo de las dos convoluciones y se suman ambos.
                // Estrictamente deberia ser la suma cuadratica, pero analizando el
                // algoritmo, no se obtiene tanto error haciendo la suma de los modulos
                A1 = ABS A1, A0 = ABS A0;
                R6 = (A0 += A1);
                // Se quita el carry, para que no se mezcle con el color
                R6 = R6 & R3;
                R6 <<= 8;
                // Se le suma el color, (dejando comentada la linea de arriba es negro)
                R6 = R6 | R5;
                // Se almacena en el buffer de salida
                W[I1] = R6.L;
        Active_Line_Loop_End:
                I1+= M0;
```

- Se copia el último pixel de la línea tal cual está, y se precarga lo que se necesita para comenzar a filtrar la línea siguiente.

```
        // Cuando termino de hacer el algoritmo para toda la linea G2x, me queda copiar el
        //ultimo pixel de la liena tal cual esta, para luego pasar a la linea siguiente
        I0 += M3;
        R1.L = w[I0++];
        W[I1] = R1.L;
        I1+= M0;

        // Dejo preparado I0 para que apunte al princio de la siguiente linea del frame
        // original, e I1 para que apunte al principio de la siguiente linea del frame
        // destino
        I1 += M3;
        I1 -= M1;

        I0 += M3;
Frame_Loop_End:
        I0 -= M1;
```

- Se copia la última línea tal cual está, por el mismo motivo que se copiaba la primera.

```
        // Se copia la ultima linea tal cual esta
        P2.H = HI(Line_Length/2);
        P2.L = LO(Line_Length/2);
        LSETUP(Last_Line_Loop_Start, Last_Line_Loop_End) LC1 = P2;
        Last_Line_Loop_Start:
                R1 = [I0++];
        Last_Line_Loop_End:
                [I1++] = R1;

Deteccion_De_Bordes.END:
        RTS;
```

De la misma manera que se midió el tiempo de transmisión del SPORT, se realizó un programa de prueba que aísle al algoritmo de filtrado para así poder medir su performance.

Grafico del osciloscopio con tiempo de procesamiento del algoritmo

10 Conclusiones

La implementación de procesamiento de video en tiempo real no es una tarea trivial. Cada cuadro a procesar cuenta con más de 360 KBytes de datos si se trabaja en blanco y negro, y más de 720 KBytes si se trabaja en color.

A su vez, ya realizamos el cálculo que indica que el caudal de datos a procesar es de 10,95 MBytes por segundo para imágenes NTSC en blanco y negro, y que realizando los 9 productos y acumulaciones por pixel de imagen para cada convolución, la velocidad de procesamiento debe ser superior a 100 MIPS. Como la organización de la memoria no es lineal, en realidad cada convolución lleva más de 9 instrucciones por lo que se necesita una velocidad de procesamiento aún más rápida.

Todo este análisis nos lleva a la conclusión de que los algoritmos y el manejo de memoria deben estar sumamente optimizados para poder procesar el video en tiempo real y no perder cuadros.

Analizando detalladamente el algoritmo implementado, se observa que la mayor cantidad de tiempo se pierde en los accesos a SDRAM a través de los DAGs. Como trabajo a futuro y con vistas a mejorar la performance de los filtros, queda pendiente hacer copias parciales de la SDRAM a memoria interna por medio del DMA. Esto, si bien tendrá como consecuencia un algoritmo más complicado, permitirá trabajar a una velocidad mucho más rápida.

11 Bibliografía

[1] *ADSP-BF533 Blackfin Processor Hardware Reference.* Revision 3.4 Apr 2009. **Analog Devices, Inc.**

[2] *ADSP-BF533 EZ-KIT Lite EvaluationSystem Manual.* Revision 3.1 Sep 2007. **Analog Devices, Inc.**

[3] *Blackfin Processor Programming Reference.* Revision 1.3 Sep 2008. **Analog Devices, Inc.**

[4] *Video Filtering Considerations for Media Processors.* David Katz, Rick Gentile. **Analog Devices, Inc.**

[5] *VideoFramework Considerations for Image Processing on Blackfin Processors.* Revision 1 Sep 2005. **Kunal Singh and Ramesh Babu**

[6] *Edge Detection.* Chapt 4 and Chapt 5. **Trucco**