Universidad de Cantabria

Ampliación de Sistemas Electrónicos

Ejercicio de evaluación de los Temas 1.1 y 1.2

Autor : Betegón garcía, Miguel

E-mail : mbg51@alumnos.unican.es

Diciembre 2018

Índice

1. Introducción

Esta primera práctica de la asignatura *Ampliación de Sistema Electrónicos* trata de diseñar un sistema de procesado de video en tiempo real, que tome imágenes de una cámara en color y obtenga como resultado una imagen en escala de grises que muestre los bordes o contornos que se observan en la imagen.

En este caso, los parámetros proporcionados en los que basar el proyecto son los mostrados en el cuadro 1

NBYTES_COLOR_DATA_MEM	NBYTES_GRAY_DATA_MEM	SOBEL_G_EQUATION	**Objetivo en sobel**
4	1	4	Tiempo

Cuadro 1: Parámetros.

Esta práctica esta estructurada en dos partes, que corresponden con las dos partes del primer módulo de la asginatura, a realizar:

(i) Diseño a nivel algoritmo y síntesis del módulo sobel.

(ii) Diseño a nivel de transferencia de registros y síntesis del módulo color2gray.

En cada una de las partes se modificará un fichero, `sobel.c` y `color2gray.vhd` respectivamente. Dichos códigos se encuentran en el Anexo A. También se puede acceder a los códigos a través del repositorio de github, [1].

El primer paso es realizar el código correspondiente a `sobel.c`, que una vez desarrollado, permitirá simular y sintetizar el proyecto global en el programa *Vivado HLS*. La versión utilizada ha sido concretamente *Vivado HLS 18.3* correspondiente a la tercera actualización del año 2018. No debiera haber problemas en utilizar este código en versiones anteriores.

Antes de comenzar con la primera parte, se deben configurar los valores conforme a los parámetros proporcionados en el cuadro 1. El fichero donde han de configurarse es `config.h`, se cambian las siguientes líneas tal que:

```
1 #define NBYTES_COLOR_DATA_MEM 4
2 #define NBYTES_GRAY_DATA_MEM  1
3 #define SOBEL_G_EQUATION  4
```

Una vez configurados los parámetros, se procede a realizar el código Sobel.

2. Diseño a nivel algoritmo y síntesis del módulo sobel

Esta parte de la práctica se realiza en *Vivado HLS*. El único fichero que se ha de modificar es `sobel.c`, lo que constituye el componente *sobel* utilizando síntesis de alto nivel.

Dentro del fichero, se desarrolla una función C, sobel, que describe la funcionalidad del componente. Una vez desarrollada esta función, se simula y sintezia utilizando el entorno *Vivado HLS*.

2.1. Sobel

Esta función implementa el cálculo de los gradientes que permiten obtener los bordes de la imagen. A partir de una imagen en escala de grises, `imgGray`, el operador sobel genera 2 gradientes: $Grad_x$ y $Grad_y$.

Dichos gradientes se obtienen mediante la operación de convolución de matrices dada por las siguientes ecuaciones:, y han de ser calculados para cada pixel en la imagen:

$$Grad_x(x,y) = \sum_{i=-1}^{+1} \sum_{j=-1}^{+1} \{G_x(i+1, y+1) \times imgGray(x+i, y+j)\} \tag{1}$$

$$Grad_y(x,y) = \sum_{i=-1}^{+1} \sum_{j=-1}^{+1} \{G_y(i+1, y+1) \times imgGray(x+i, y+j)\} \tag{2}$$

Donde:

$\boldsymbol{x}$ — $0 < x < width - 1$ — Columna.

$\boldsymbol{y}$ — $0 < t < height - 1$ — Fila.

$$\boldsymbol{G_x} — \left\{ \begin{array}{ccc} \text{-1} & 0 & \text{-1} \\ \text{-2} & 0 & +2 \\ \text{-1} & 0 & +1 \end{array} \right\}$$

$$\boldsymbol{G_x} — \left\{ \begin{array}{ccc} \text{-1} & \text{-2} & \text{-1} \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{array} \right\}$$

Estás ecuaciones se expresan en las siguientes líneas del código de `sobel.c` que figura en el anexo A.1:

```
29 Gradx+=leePixelGray(imgGray,col+i-1,fila+j-1,width)*Gx[i][j];
30 Grady+=leePixelGray(imgGray,col+i-1,fila+j-1,width)*Gy[i][j];
```

Hay que tener en cuenta que aunque estas ecuaciones definen el valor de los gradientes, faltan los siguientes casos:

- Primera fila de la imagen, $y = 0$.
- Primera columna de la imagen, $x = 0$.
- Última fila de la imagen $y = height - 1$.
- Última columna de la imagen $x = width - 1$.

Para estos casos aislados los gradientes toman el valor 0 y por tanto el gradiente final vale 0. Se puede comprobar esto observando las siguientes línes de `sobel.c`, en las que se asigna el valor del gradiente total:

```
19 if(fila==0 || fila==height-1 || col==0 || col==width-1){
20 Grad= 0;
21 }
```

Una vez se ha llegado a este punto, se hace uso de la ecuación brindada por el parámetro `SOBEL_G_EQUATION`:

$$G = |Grad_x| + |Grad_y| \tag{3}$$

Esta ecuación se implementa en el código de la siguiente manera:

```
33 Grad = abs(Gradx) + abs(Grady);
```

El último paso es comprobar el valor del gradiente, si es mayor o igual que el *threshold típico* entonces ese pixel tendrá un valor de 255 y se escribirá en la memoria `imgEdges`. Si es menor, su valor será 0. De forma más precisa:

$$imgEdges(x, y) = \begin{cases} 0 & \text{si } G(x, y) < threshold \\ 255 & \text{si } G(x, y) \geq threshold \end{cases} \tag{4}$$

esta operaciones se realizan para cada pixel, por lo que se implementan dos ciclos anidados, el primero para recorrer las filas y el anidado para para recorrer las columnas:

```
14 lazoprincipal: for(fila=0;fila<height;fila++) {
15 #pragma HLS LOOP_TRIPCOUNT min=100 max=140 avg=120
16   lazoanidado: for(col=0; col<width; col++) {
17   #pragma HLS LOOP_TRIPCOUNT min=160 max=200 avg=176
```

Una vez desarrollado el código, se procede a realizar la simulación y síntesis del mismo. A continuación se muestran las imágenes conseguidas implementando Sobel, mostradas en la figura 1:

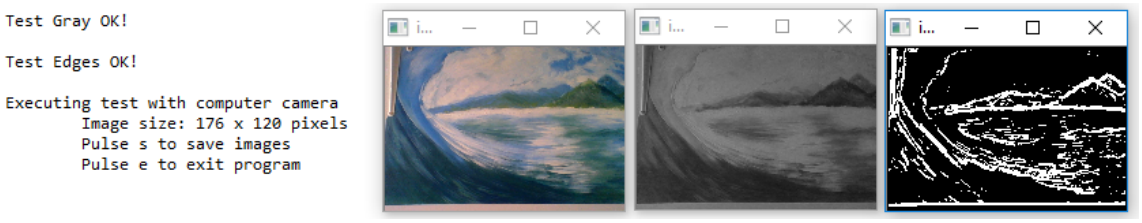
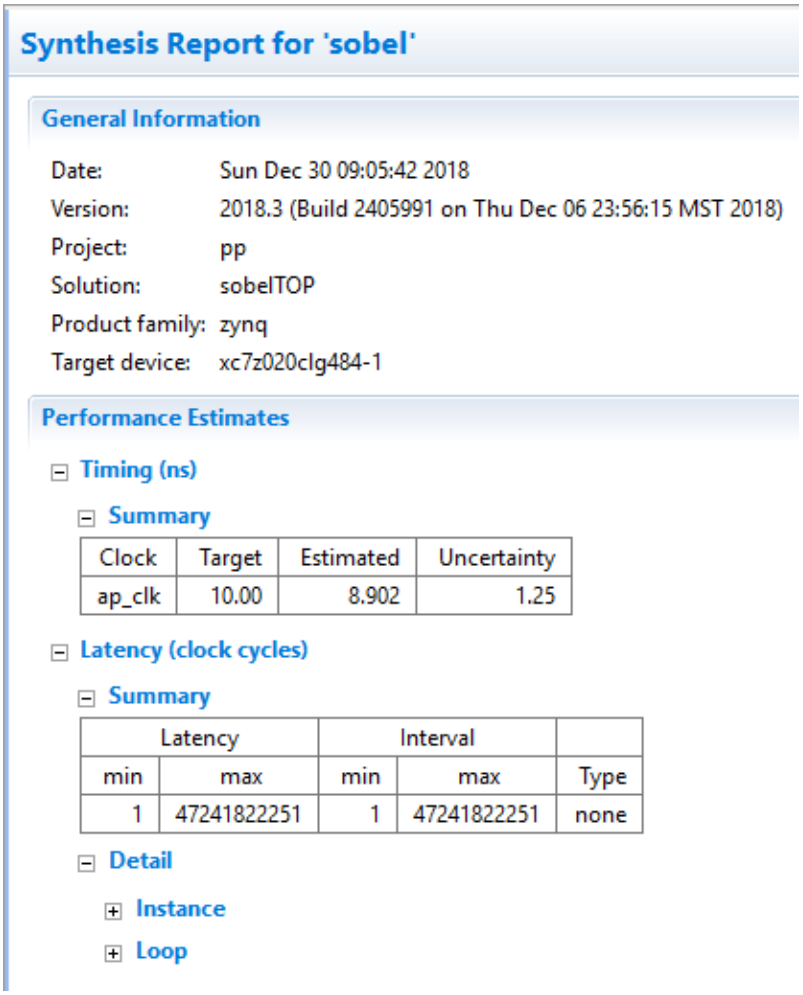


Figura 1: Imagenes obtenidas.

Se puede ver perfectamente que se realiza el paso de escala de grises a obtener los bordes de la imagen sin ningún problema. A la izquierda de la figura 1 se observa como la consola de *Vivado HLS*, *Vivado HLS Console*, confirma que se han pasado los testbench ofrecidos para la comprobación.

Después de realizar la síntesis, se presta atención a la información obtenida en (*Synthesis Report for 'Sobel'*). Lás metricas que se pueden obtener en esta pantalla son varias: latencia, frecuencia del reloj, LUTS y demás.

La figura 2, dispuesta en la siguiente página, es una captura del informe conseguido a partir de la síntesis de sobel (sin el uso de directivas):



Synthesis Report for 'sobel'

General Information

Date:	Sun Dec 30 09:05:42 2018
Version:	2018.3 (Build 2405991 on Thu Dec 06 23:56:15 MST 2018)
Project:	pp
Solution:	sobelTOP
Product family:	zynq
Target device:	xc7z020clg484-1

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.902	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1	47241822251	1	47241822251	none

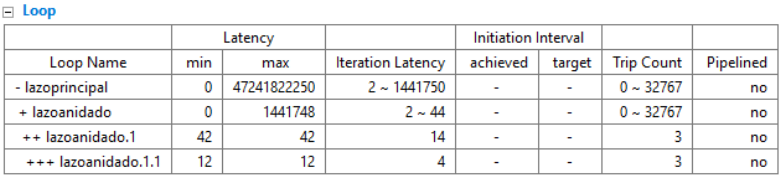
Detail

Instance

Loop

Figura 2: Imagenes obtenidas.

En la figura 2 se infiere que la latencia máxima conseguida es 4724182251, un valor alto. Se puede además conseguir información acerca de la latencia de cada ciclo, haciendo click en *loop*, de donde surge lo mostrado en la figura 3:



Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- lazoprincipal	0	47241822250	2 ~ 1441750	-	-	0 ~ 32767	no
+ lazoanidado	0	1441748	2 ~ 44	-	-	0 ~ 32767	no
++ lazoanidado.1	42	42	14	-	-	3	no
+++ lazoanidado.1.1	12	12	4	-	-	3	no

Figura 3: Imagenes obtenidas.

Como es lógico, el ciclo con mayor latencia es el `lazoprincipal`, el que contiene

al resto de lazos. Estas latencias conseguidas no son todo lo buenas que pueden llegar a ser. El objetivo de esta parte consiste también en disminuir la latencia lo máximo posible. Los resultados obtenidos , pueden ser mejorados mediante métodos de optimización como son las directivas.

2.2. Optimización

Se pretende mininimar la latencia, para ello, se hace uso de directivas.

Las directivas que se utilizan en este caso son directivas de código fuente, que se pueden asociar a una solución o si se insertan en el código se aplican a todas las soluciones.

Se ha elegido insertarlas en el código. De esta forma, se aplican a todas las soluciones. Puesto que el objetivo es reducir la latencia, hay que tener en cuenta que consiguiendolo seguramente se pierdan otras prestaciones del sistema, como puede ser el área efectiva.

Este tipo de directivas, son insertadas dentro del código con la siguiente estructura: `#pragma HLS <directiva>`

Una directiva muy interesante es la que lleva el nombre de `LOOP_FLATTEN` en *Vivado HLS*, y es que permite combinar los lazos. Esto quiere decir, que si se tienen dos ciclos anidados, se convierten en uno solo meidante está directiva. `LOOP_FLATTEN` está activada por defecto, y si por algún motivo se quisiera deshabilitar basta con ponerla en modo *off* dentro de las opciones de la directiva. En este caso, se quedará activa puesto que ayuda a reducir la latencia.

Se detallan a continuación las directivas elegidas y la razón o razones por las cuáles se han implementado:

`PIPELINE` Reduce el intervalo de inicio, *Initiation Interval* (II), para la función o bucle que se aplique, permitiendo la ejecución concurrente/en paralelo de las iteraciones. Se utiliza para los dos ciclos que tienen un número de iteraciones fijas, ya que si el número de iteraciones es variable, compensa utilizar otra directiva.

También se utilizó en primera instancia la directiva `UNROLL` aunque, sustituyendo esta directiva por las que se comentan a continuación se consiguió mejorar la latencia.

En el código se implementa esta directiva dos veces, como se puede observar en las siguientes líneas de `sobel.c`:

```
25 lazoanidado2: for(i=0;i<3;i++){
26 #pragma HLS PIPELINE
27   lazoanidado3: for(j=0;j<3;j++){
28   #pragma HLS PIPELINE
```

`LOOP_TRIPCOUNT` Esta directiva se aplica un bucle para especificar manualmente el número total de iteraciones realizadas por dicho bucle. Es muy interesante, porque permite calcular la latencia de un ciclo cuando antes no se podía, ayudando a determinar optimizaciones apropiadas para el diseño.

Se implementa en `sobel.c` de la siguienter forma:

```
25 lazoprincipal: for(fila=0;fila<height;fila++) {
26 #pragma HLS LOOP_TRIPCOUNT min=100 max=140 avg
     =120
27   lazoanidado: for(col=0; col<width; col++) {
28   #pragma HLS LOOP_TRIPCOUNT min=160 max=200
       avg=176
```

Usandola una vez por cada ciclo, sabiendo que las imágenes tienen unas dimensiones de 176×20.

Añadiendo estas directivas al código se consigue mejorar notablemente la latencia. En la figura 4 se muestra el informe de la síntesis de sobel, *Synthesis Report for 'Sobel'*, correspondiente al uso de las directivas comentadas anteriormente.

Para comprender cuanto ha mejorado la latencia, se realiza un rápido cálculo en el que se involucran las dos latencias máximas de las síntesis, con y sin directivas. Dicho calculo es:

$$\#Bigger = \frac{latency_{max}(nodirectives)}{latency_{max}(directives)} \tag{5}$$

Haciendo uso de la figura 4 se conoce la latencia máxima usando directivas y con la figura 1 la latencia sin usar ningunt tipo de directiva.

Synthesis Report for 'sobel'

General Information

Date:	Sun Dec 30 10:24:25 2018
Version:	2018.3 (Build 2405991 on Thu Dec 06 23:56:15 MST 2018)
Project:	pp
Solution:	sobelTOP
Product family:	zynq
Target device:	xc7z020clg484-1

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.510	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
51202	390002	51202	390002	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- lazoprincipal_lazoanidado	51201	390001	4 ~ 13	-	-	12800 ~ 30000	no
+ lazoanidado2	8	8	5	2	1	3	yes

Figura 4: *Synthesis Report for 'Sobel'* con el uso de directivas.

Se observa a simple vista, sin necesidad de hacer ningún cálculo que la latencia ha mejorado considerablemente. Aún así, se aplica la fórmula anterior para conocer cuantas veces se ha reducido la latencia en comparación a no usar directivas:

$$\#Bigger = \frac{latency_{max}(nodirectives)}{latency_{max}(directives)} = \frac{4724182251}{390001} = 12113.25 \qquad (6)$$

Este resultado es aclaratorio, se ha reducido la latencia 12113.25 veces la anterior. Consiguiendo el objetivo marcado por la práctica.

Aunque se han sacrificado otros factores, como son el número de LUTS necesarias. en las figuras 5 y 6 se muestran el número total de LUTS necesarias sin usar directivas y usándolas, respectivamente.

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	3	-	-
Expression	-	-	0	681
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	6	2
Multiplexer	-	-	-	155
Register	-	-	436	-
Total	0	3	442	838
Available	280	220	106400	53200
Utilization (%)	0	1	~0	1

Figura 5: Número de LUTS necesarias sin el uso de directivas.

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	5	-	-
Expression	-	10	0	1132
FIFO	-	-	-	-
Instance	-	-	0	60
Memory	-	-	-	-
Multiplexer	-	-	-	158
Register	-	-	841	-
Total	0	15	841	1350
Available	280	220	106400	53200
Utilization (%)	0	6	~0	2

Figura 6: Número de LUTS necesarias con el uso de directivas.

Se observa en las figuras anteriores que para el uso de directivas se necesitan un total de 1350 LUTS mientras que sin usar directivas solo son necesarias 838

Estos resultados dejan claro que, al querer máximizar un aspecto o métrica del sistema, generalmente se debe sacrificar otro.

3. Diseño a nivel de transferencia de registros y síntesis del módulo color2gray

En esta segunda parte de la práctica se va utiliza *Vivado* para desarrollar el código y una vez está implementado, se copia en *Vivado HLS*.

El objetivo es implementar el módulo `color2gray` realizando una descripción RTL FSMD. Para ello, se crea un fichero llamado `color2gray.vhd` que se sustituye una vez desarrollado por el que se encuentra en el proyecto de *Vivado HLS* bajo la ruta `Solution1` → `syn` → `vhdl` → `color2gray.vhd`.

Se ha de cambiar la función top en *Vivado HLS* y poner `color2gray.c`. El código VHDL existirá siempre y cuando se haya sintetizado la solución deseada, con la función top comentada anteriormente.

Se aprovecha el código presente en dicha ruta, copiando las librerías que usa y los puertos al fichero que se desarrollará, esto, resulta en el siguiente código, presente en el Anexo A.2:

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4
5  entity color2gray is
6  port (
7      ap_clk : IN STD_LOGIC;
8      ap_rst : IN STD_LOGIC;
9      ap_start : IN STD_LOGIC;
10     ap_done : OUT STD_LOGIC;
11     ap_idle : OUT STD_LOGIC;
12     ap_ready : OUT STD_LOGIC;
13     imgColor_address0 : OUT STD_LOGIC_VECTOR (14 downto 0);
14     imgColor_ce0 : OUT STD_LOGIC;
15     imgColor_q0 : IN STD_LOGIC_VECTOR (31 downto 0);
16     imgGray_address0 : OUT STD_LOGIC_VECTOR (14 downto 0);
17     imgGray_ce0 : OUT STD_LOGIC;
18     imgGray_we0 : OUT STD_LOGIC;
19     imgGray_d0 : OUT STD_LOGIC_VECTOR (7 downto 0);
20     width : IN STD_LOGIC_VECTOR (15 downto 0);
21     height : IN STD_LOGIC_VECTOR (15 downto 0) );
22 end color2gray;
```

El VHDL presente en la ruta comentada anteriormente, ha sido generado mediante el fichero `color2gray.c` proporcionado a los alumnos.

Aunque el código que se desarrolla en este apartado se realice después de `sobel.c`, en realidad es la primera parte del sistema en completo puesto que se encarga de convertir las imágenes tomadas por la cámara en su equivalente en escala de grises (que posteriormente sobel convertirá esa escala en bordes).

Con este objetivo, se realiza un cambio de coordenadas de color, pasando de cordenadas RGB a coordenadas YUV. La componente Y (luminancia) proporciona la imagen en escala de grises.

La conversión en escala de girses viene dada por la ecuación:

$$Y = 0.2627R + 0.678G + 0.0593B \tag{7}$$

Que para evitar operaciones en punto flotante, se aproxima de la siguiente forma:

$$Y = \frac{(67R + 174G + 15B)}{256} \tag{8}$$

Una vez creado un proyecto en *Vivado* y añadido el fichero VHDL con el contenido de los puertos, se procede a desarrollar el resto del fichero para que cumpla los requisitos necesarios. La descripción VHDL debe realizar la ecuación de arriba.

Antes de escribir el código, se crea un diagrama de estados que facilita el proceso de desarrollar el código. Este diagrama de estados se encuentra en la siguiente página, en la figura 7.

En el diagrama se pueden observar 8 estados, siendo estos las bolas de colores del diagrama. Estos estados se implementan en el código en forma de una variable de tipo *state_type*. Se podría haber implementado como una señal, pero por motivos educaciones se ha utilizado una variable, así se comprenden las diferencias usando variables y señales. El código de los estados correspondiente a `color2gray.vhd` se encuentra en el anexo A.2, y son las que siguen:

```
41 type state_type is (init, read,grad1,grad2,grad3,grad4,
     write, endd);
42 variable state: state_type;
```

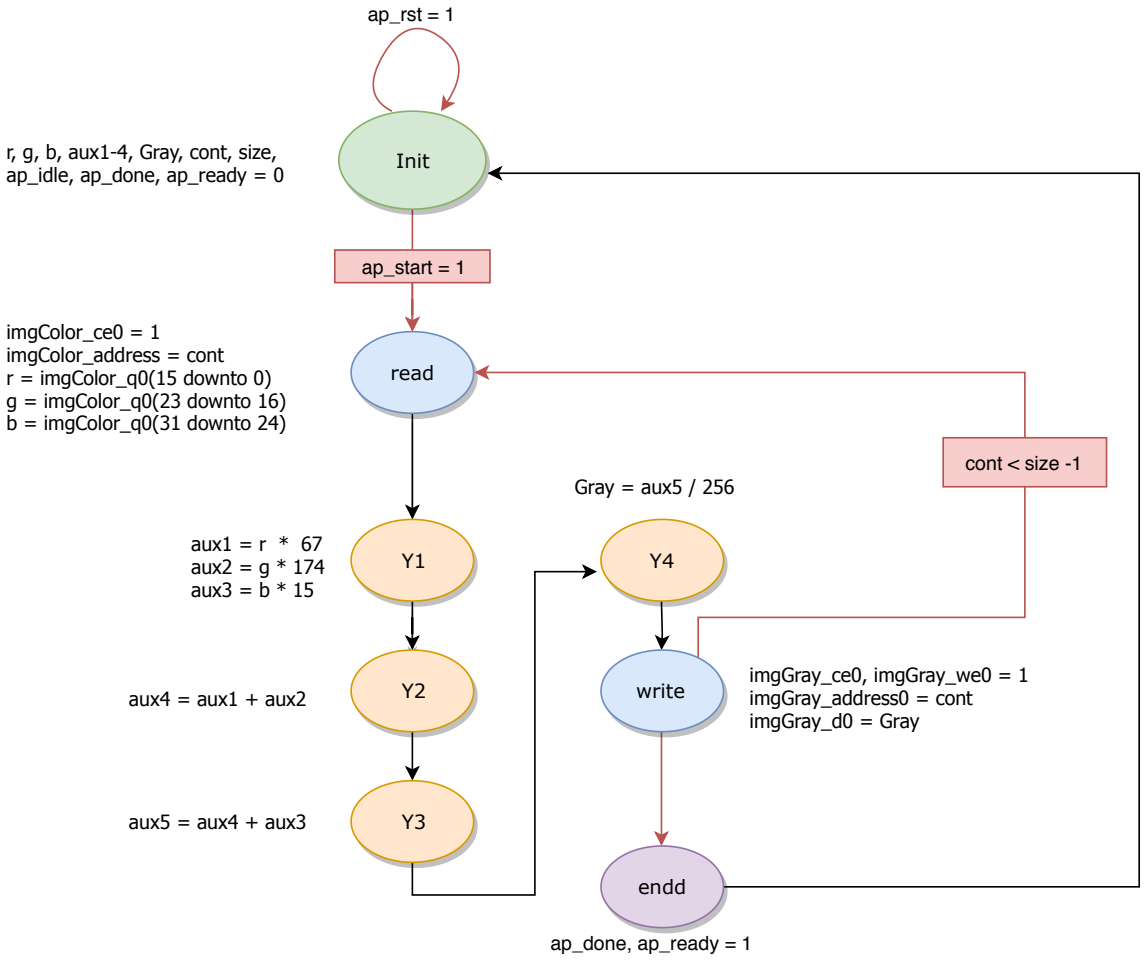


Figura 7: *Throughput* frente a *Retardo* con $M = 12$

Además de los estados, en el diagrama se incorporan líneas rojas, que son condiciones necesarias para ir de uno a otro estado. El resto de estados no tienen varias posibilidades, de ellos se irá a otro estado salvo que `ap_rst = 1` en cuyo caso se volverá al estado inicial. En cada estado, se han añadido las operaciones y asgnaciones de valores que se han de realizar.

Con el diagrama de estados se facilita la escritura de la descripción VHDL, que se hace por estados y se detallan cada uno de ellos a continuación. Se añade el código VHDL correspondiente a cada estado:

`init` El primer estado es el llamado `init`. En él se inicializan los valores de las señales, variables y puertos necesarios. Además, se multipla se realiza la siguiente operación: $size = width \times height$ con la que se

obtiene el número de píxeles de la imagen. De este estado sólo se sale cuando el puerto de entrada `ap_start='1'`. Mientras tanto, se permanece a la espera. En dicho valor de ap_start se pasa al siguiente estado, `read`, como se observa en el diagrama de estados. El código correspondiente a este estado:

```
25 if (ap_start='1') then
26     size<= std_logic_vector(unsigned(width)*
          unsigned(height));
27     state:=read;
```

`read` En el segundo estado, `read`, se pone `imgColor_ce0 = '1'` lo que activa la memoria `imgColor` para poder leerla. Se lee la memoria en la dirección `imgColor_address0` que está controlada por la variable `cont`, cuyo valor es 0 en la primera iteración al venir del estado inicio y después va aumentando su valor para poder leer las siguientes direcciones. En `read`, se lee la memoria para conseguir los valores RGB. Las siguientes líneas de código muestran la lectura de dichos valores. Estas acciones se traducen en código VHDL:

```
25 imgColor_address0<=std_logic_vector(cont);
26 imgColor_ce0<='1';
27 r<=imgColor_q0(31 downto 24);
28 g<= imgColor_q0(23 downto 16);
29 b<= imgColor_q0(15 downto 8);
```

Notar que no se leen los primeros bits de la palabra de la memoria, puesto que están vacios, al tratarse de una memoria de 32 bits y cada valor R,G y B ocupa 8 bits, por lo que esos 8 bits restantes quedan vacíos en este caso. De este estado se pasa directamente al siguiente, `Y1`

`Y1` El tercer estado, `Y1`, corresponde ya a la computación de la luminancia. En él, se realiza la multiplicación de cada valor RGB p or su correspondiente integer, expresado en la ecuación de luminiscencia. Esto se describe en VHDL de la siguiente forma:

```
25 aux1 <= std_logic_vector(unsigned(r)*67);
26 aux2 <= std_logic_vector(unsigned(g)*174);
27 aux3 <= std_logic_vector(unsigned(b)*15);
28 state:=grad2;
```

De este estado se pasa a `Y2`.

`Y2` En `Y2` se realiza la siguiente operación en aras de obtener la limunancia. Esta operación es la suma de los dos de los valores obtenido en el estado anterior. De este estado se pasa a `Y3` directamente.

```
25 aux4 <= std_logic_vector(unsigned(aux1)+unsigned(
     aux2));
26 state:=grad3;
```

`Y3` En este estado se realiza la suma del valor obtenido en el estador anterior y el valor RGB que faltaba de añadir. Se avanza de este estado a `Y4`.

```
25 aux5 <= std_logic_vector(unsigned(aux4)+unsigned(
     aux3));
26 state:=grad4;
```

`Y4` Este estado es el último en relación al cálculo de la luminancia, se divide el valor resultante de las sumas entre 256. Se ha elegido hacerlo en forma de desplazamiento aunque no hay problema en dividirlo entre el integer 256. Una vez realizada esta operación, la señal `Gray` contiene el valor que será escrito en la memoria `imgGray`. Esta escritura se lleva a cabo en el siguiente estado, `write`.

```
25 Gray <= aux5/256;
26 state:=write;
```

`write` De forma análoga a como se hizo en el estado `read`, en `write` también se activa la memoria `imgGray`, esta vez poniendo a 1 `imgGray_ce0` y `imgGray_we0`. Se escribe `Gray` en la memoria en la dirección `imgGray_address0`. De este estado se puede partir hacia `read` o hacia `endd`, esto depende de que el contador `cont` sea menor que el numero de pixeles (ya que se inicializó en 0 el contador). Si se cumple esta condición, entonces se pasa al estado `read` para realizar todas las operaciones con el siguiente pixel. En cambio, si el pixel analizado era el último, se pasa al estado `endd`.

```
25 imgGray_ce0 <='1';
26 imgGray_we0 <='1';
```

```
27 imgGray_address0<=std_logic_vector(cont);
28 imgGray_d0<=std_logic_vector(Gray);
29 if(unsigned(cont)<unsigned(size)-1) then
30     cont<=std_logic_vector(unsigned(cont)+1);
31     state:=read;
32  else
33     state:= endd;
34 end if;
```

Este código ilustra las operaciones que se llevan a cabo en el estado `write`

`endd` En lo que se considera el último etado de la descripción VHDL, se llega a él tras obtener los valores y escrbirlos en la memoria de todos los píxeles de la imagen. Em este estado, se ponen los puertos de salida `ap_done` y `ap_ready`. De este estado se parte nuevamente a `init`.

```
25 ap_done<='1';
26 ap_ready<='1';
27 state:=init;
```

No siendo requisito de la práctica, se ha desarrollado un testbench que permite comprobar el funcionamiento de `color2gray.vhd`. El nombre del fichero de testbench es `tbcolor2gray.vhd` y se encuentra en en el Anexo A.3.

Una vez comprobado con el testbench que el código desarrollado es correcto, se procede a poner este código dentro del proyecto de *Vivado HLS*, en la ruta mencionada con anterioridad. Una vez hecho esto, se realiza la Co-simulación y se comprueba que se han pasado todos los tests correctamente.

Con este último paso, se da por finalizada la práctica 1 de la asginatura.

CREAR GITHUB COLOR2GRAY.VHD SOBEL.C tbcolor2gray.vhd CAMBIAR GITHUB DE LA BIBLIOGRAFIA

CAMBIAR GITHUB DE LA BIBLIOGRAFIA

CAMBIAR GITHUB DE LA BIBLIOGRAFIA

CAMBIAR GITHUB DE LA BIBLIOGRAFIA

CAMBIAR GITHUB DE LA BIBLIOGRAFIA

CAMBIAR GITHUB DE LA BIBLIOGRAFIA

CAMBIAR GITHUB DE LA BIBLIOGRAFIA

CAMBIAR GITHUB DE LA BIBLIOGRAFIA

CAMBIAR GITHUB DE LA BIBLIOGRAFIA

CAMBIAR GITHUB DE LA BIBLIOGRAFIA

CAMBIAR GITHUB DE LA BIBLIOGRAFIA

CAMBIAR GITHUB DE LA BIBLIOGRAFIA

CAMBIAR GITHUB DE LA BIBLIOGRAFIA

ANEXOS

A. Códigos de la práctica

Se muestran a continuación los ficheros utilizados para realizar la práctica:

`sobel.c`	En este archivo se encuentra el código C de la función sobel.
`color2gray.vhd`	Descripción VHDL correspondiente a las segunda parte de la práctica.
`tbcolor2gray.vhd`	Testbench de `color2gray.vhd`

A.1. `sobel.c`

```
1  #include "typeDefinition.h"
2  #include "stdlib.h"
3
4  void sobel(memGray imgGray[MEMGRAYSIZE],memGray imgEdges[MEMGRAYSIZE], short width,
       short height, unsigned int threshold) {
5
6    int Gx[3][3]={{-1,0,1},{-2,0,2},{-1,0,1}};
7    int Gy[3][3]={{-1,-2,-1},{0,0,0},{1,2,1}};
8    int i,j;
9    short fila;
10   short col;
11   int Gradx,Grady;
12   int Grad;
13
14   lazoprincipal: for(fila=0;fila<height;fila++) {
15   #pragma HLS LOOP_TRIPCOUNT min=100 max=140 avg=120
16     lazoanidado: for(col=0; col<width; col++) {
17     #pragma HLS LOOP_TRIPCOUNT min=160 max=200 avg=176
18
19        if(fila==0 || fila==height-1 || col==0 || col==width-1){
20        Grad= 0;
21        }
22       else{
23         Gradx=0;
24         Grady=0;
25         lazoanidado2: for(i=0;i<3;i++){
26         #pragma HLS PIPELINE
27           lazoanidado3: for(j=0;j<3;j++){
28           #pragma HLS PIPELINE
29               Gradx+=leePixelGray(imgGray,col+i-1,fila+j-1,width)*Gx[i][j];
30               Grady+=leePixelGray(imgGray,col+i-1,fila+j-1,width)*Gy[i][j];
31             }
32           }
33          Grad = abs(Gradx) + abs(Grady);
34        }
35            if(Grad<threshold){
```

```
36                escribePixelGray(0,imgEdges,col,fila,width);
37              }
38              else{
39                escribePixelGray(255,imgEdges,col,fila,width);
40              }
41          }
42      }
43  }
```

A.2. color2gray.vhd

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4
5  entity color2gray is
6  port (
7      ap_clk : IN STD_LOGIC;
8      ap_rst : IN STD_LOGIC;
9      ap_start : IN STD_LOGIC;
10     ap_done : OUT STD_LOGIC;
11     ap_idle : OUT STD_LOGIC;
12     ap_ready : OUT STD_LOGIC;
13     imgColor_address0 : OUT STD_LOGIC_VECTOR (14 downto 0);
14     imgColor_ce0 : OUT STD_LOGIC;
15     imgColor_q0 : IN STD_LOGIC_VECTOR (31 downto 0);
16     imgGray_address0 : OUT STD_LOGIC_VECTOR (14 downto 0);
17     imgGray_ce0 : OUT STD_LOGIC;
18     imgGray_we0 : OUT STD_LOGIC;
19     imgGray_d0 : OUT STD_LOGIC_VECTOR (7 downto 0);
20     width : IN STD_LOGIC_VECTOR (15 downto 0);
21     height : IN STD_LOGIC_VECTOR (15 downto 0) );
22 end color2gray;
23
24 architecture behav of color2gray is
25
26 signal r: std_logic_vector(7 downto 0);
27 signal g: std_logic_vector(7 downto 0);
28 signal b: std_logic_vector(7 downto 0);
29 signal aux1: std_logic_vector(15 downto 0);
30 signal aux2: std_logic_vector(15 downto 0);
31 signal aux3: std_logic_vector(15 downto 0);
32 signal aux4: std_logic_vector(15 downto 0);
33 signal aux5: std_logic_vector(15 downto 0);
34 signal Gray: unsigned(7 downto 0);
35 signal cont: std_logic_vector(14 downto 0);
36 signal size: std_logic_vector(31 downto 0);
37
38 begin
39
40 main_synch_process: process(ap_clk,ap_rst)
41 type state_type is (init, read,Y1,Y2,Y3,Y4,write, endd);
42 variable state: state_type;
43
44 begin
45     if (ap_rst='1') then
46         r<=(others=>'0');
47         b<=(others=>'0');
48         g<=(others=>'0');
49         Gray<=(others=>'0');
50         aux1<=(others=>'0');
51         aux2<=(others=>'0');
52         aux3<=(others=>'0');
53         aux4<=(others=>'0');
54         aux5<=(others=>'0');
55         cont <=(others=>'0');
56         size <=(others=>'0');
57         ap_idle <='0';
58         ap_done <='0';
```

```
59          ap_ready <='0';
60
61        state:= init;
62
63      elsif (ap_clk='1' and ap_clk'EVENT) then
64          case state is
65              when init =>
66                  r<=(others=>'0');
67                  b<=(others=>'0');
68                  g<=(others=>'0');
69                  Gray<=(others=>'0');
70                  aux1<=(others=>'0');
71                  aux2<=(others=>'0');
72                  aux3<=(others=>'0');
73                  aux4<=(others=>'0');
74                  aux5<=(others=>'0');
75                  size <=(others=>'0');
76                  cont <=(others=>'0');
77                  ap_idle <='0';
78                  ap_done <='0';
79                  ap_ready <='0';
80                  if (ap_start='1') then
81                      size<= std_logic_vector(unsigned(width)*unsigned(height));
82                      state:=read;
83
84                  else
85                      state:=init;
86                  end if;
87
88              when read =>
89                  imgColor_address0<=std_logic_vector(cont);
90                  imgColor_ce0<='1';
91      r<=imgColor_q0(31 downto 24);
92      g<= imgColor_q0(23 downto 16);
93      b<= imgColor_q0(15 downto 8);
94
95                  state:=Y1;
96
97              when Y1 =>
98                  aux1 <= std_logic_vector(unsigned(r)*67);
99                  aux2 <= std_logic_vector(unsigned(g)*174);
100                 aux3 <= std_logic_vector(unsigned(b)*15);
101                 state:=Y2;
102
103             when Y2 =>
104                 aux4 <= std_logic_vector(unsigned(aux1)+unsigned(aux2));
105                 state:=Y3;
106             when Y3 =>
107                 aux5 <= std_logic_vector(unsigned(aux4)+unsigned(aux3));
108                 state:=Y4;
109
110             when Y4 =>
111                 Gray <= aux5/256;
112                 state:=write;
113
114             when write =>
115                 imgGray_ce0<='1';
116                 imgGray_we0<='1';
117                 imgGray_address0<=std_logic_vector(cont);
118                 imgGray_d0<=std_logic_vector(Gray);
119                 if(unsigned(cont)<unsigned(size)-1) then
```

```
120                     cont<=std_logic_vector(unsigned(cont)+1);
121                     state:=read;
122                  else
123                     state:= endd;
124                 end if;
125
126             when endd =>
127             ap_done<='1';
128             ap_ready<='1';
129             state:=init;
130         end case;
131      end if;
132
133    end process;
134 end behav;
```

A.3. tbcolor2gray.vhd

```
1
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.ALL;
4  USE ieee.numeric_std.ALL;
5
6
7  ENTITY color2graytb IS
8  END color2graytb;
9
10 ARCHITECTURE behavior OF color2graytb IS
11
12     -- Component Declaration for the Unit Under Test (UUT)
13
14     COMPONENT color2gray
15     PORT(
16     ap_clk : IN STD_LOGIC;
17     ap_rst : IN STD_LOGIC;
18     ap_start : IN STD_LOGIC;
19     ap_done : OUT STD_LOGIC;
20     ap_idle : OUT STD_LOGIC;
21     ap_ready : OUT STD_LOGIC;
22     imgColor_address0 : OUT STD_LOGIC_VECTOR (14 downto 0); -- este valor y varios
           mas cambian en funcion del trabajo que sean. estan puestos los mios (mispe)
           .
23     imgColor_ce0 : OUT STD_LOGIC;
24     imgColor_q0 : IN STD_LOGIC_VECTOR (31 downto 0);
25     imgGray_address0 : OUT STD_LOGIC_VECTOR (14 downto 0);
26     imgGray_ce0 : OUT STD_LOGIC;
27     imgGray_we0 : OUT STD_LOGIC;
28     imgGray_d0 : OUT STD_LOGIC_VECTOR (7 downto 0); -- memoria donde voy a escribir
           (palabra = 1Byte)
29     width : IN STD_LOGIC_VECTOR (15 downto 0);
30     height : IN STD_LOGIC_VECTOR (15 downto 0) );
31     END COMPONENT;
32
33
34    --Inputs
35     signal ap_clk : std_logic:='0';
36     signal ap_rst : std_logic:='0';
37     signal ap_start : std_logic:='0';
38     signal imgColor_q0 : std_logic_vector(31 downto 0) := (others => '0');
39     signal width : std_logic_vector(15 downto 0) := (others => '0');
40     signal height : std_logic_vector(15 downto 0) := (others => '0');
41
42
43
44    --Outputs
45     -- signal D : std_logic; -- de soda.
46     signal imgGray_address0 : STD_LOGIC_VECTOR (14 downto 0);
47     signal imgGray_ce0 :  STD_LOGIC;
48     signal imgGray_we0 : STD_LOGIC;
49     signal imgGray_d0 : STD_LOGIC_VECTOR (7 downto 0); -- memoria donde voy a
           escribir (palabra = 1Byte)
50     signal ap_done :  STD_LOGIC;
51     signal ap_idle :  STD_LOGIC;
52     signal ap_ready : STD_LOGIC;
53     signal imgColor_address0 : STD_LOGIC_VECTOR (14 downto 0); -- este valor y
           varios mas cambian en funcion del trabajo que sean. estan puestos los mios (
```

```
        mispe).
54    signal imgColor_ce0 : STD_LOGIC;
55
56
57
58
59    -- Clock period definitions
60    constant CLK_period : time := 10 ns;
61
62  BEGIN
63
64    -- Instantiate the Unit Under Test (UUT)
65    uut: color2gray PORT MAP (
66         ap_clk => ap_clk,
67         ap_rst => ap_rst,
68         ap_start  =>ap_start,
69         ap_done  =>ap_done,
70         ap_idle  =>ap_idle,
71         ap_ready  =>ap_ready,
72         imgColor_address0  => imgColor_address0,
73         imgColor_ce0  =>imgColor_ce0,
74         imgColor_q0 => imgColor_q0,
75         imgGray_address0  =>imgGray_address0,
76         imgGray_ce0  =>imgGray_ce0,
77         imgGray_we0  =>imgGray_we0,
78         imgGray_d0  =>imgGray_d0,
79         width  =>width,
80         height  =>height
81         );
82
83    -- Clock process definitions
84    CLK_process :process
85    begin
86     ap_clk <= '0';
87     wait for CLK_period/2;
88     ap_clk <= '1';
89     wait for CLK_period/2;
90    end process;
91
92
93    -- Stimulus process
94    stim_proc: process
95    begin
96       ap_rst<='1';
97       ap_start<='0';
98
99       -- hold reset state for 100 ns.
100      wait for 100 ns;
101      ap_rst<='0';
102      wait for 50 ns;
103      ap_start<='1';
104      width<= std_logic_vector(to_unsigned(18, 16));
105      height<= std_logic_vector(to_unsigned(12, 16));
106
107      wait for 50 ns;
108      imgColor_q0 <= std_logic_vector(to_unsigned(11111111,32));
109      wait for 50 ns;
110      imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
111      wait for 50 ns;
112      imgColor_q0 <= std_logic_vector(to_unsigned(11111111,32));
113      wait for 50 ns;
```

```
114         imgColor_q0 <= std_logic_vector(to_unsigned(500,32));
115         wait for 50 ns;
116         imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
117         wait for 50 ns;
118         imgColor_q0 <= std_logic_vector(to_unsigned(11111111,32));
119         wait for 50 ns;
120         imgColor_q0 <= std_logic_vector(to_unsigned(500,32));
121         wait for 50 ns;
122         imgColor_q0 <= std_logic_vector(to_unsigned(1255555,32));
123         wait for 50 ns;
124          imgColor_q0 <= std_logic_vector(to_unsigned(11255555,32));
125         wait for 50 ns;
126          imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
127         wait for 50 ns;
128          imgColor_q0 <= std_logic_vector(to_unsigned(55550000,32));
129         wait for 50 ns;
130          imgColor_q0 <= std_logic_vector(to_unsigned(255555000,32));
131         wait for 50 ns;
132          imgColor_q0 <= std_logic_vector(to_unsigned(1234568,32));
133         wait for 50 ns;
134          imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
135         wait for 50 ns;
136         imgColor_q0 <= std_logic_vector(to_unsigned(11111111,32));
137         wait for 50 ns;
138         imgColor_q0 <= std_logic_vector(to_unsigned(500,32));
139         wait for 50 ns;
140         imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
141         wait for 50 ns;
142         imgColor_q0 <= std_logic_vector(to_unsigned(11111111,32));
143         wait for 50 ns;
144         imgColor_q0 <= std_logic_vector(to_unsigned(500,32));
145         wait for 50 ns;
146         imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
147         wait for 50 ns;
148         imgColor_q0 <= std_logic_vector(to_unsigned(11111111,32));
149         wait for 50 ns;
150         imgColor_q0 <= std_logic_vector(to_unsigned(500,32));
151         wait for 50 ns;
152         imgColor_q0 <= std_logic_vector(to_unsigned(1255555,32));
153         wait for 50 ns;
154          imgColor_q0 <= std_logic_vector(to_unsigned(11255555,32));
155         wait for 50 ns;
156          imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
157         wait for 50 ns;
158          imgColor_q0 <= std_logic_vector(to_unsigned(55550000,32));
159         wait for 50 ns;
160          imgColor_q0 <= std_logic_vector(to_unsigned(255555000,32));
161         wait for 50 ns;
162          imgColor_q0 <= std_logic_vector(to_unsigned(1234568,32));
163         wait for 50 ns;
164          imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
165         wait for 50 ns;
166         imgColor_q0 <= std_logic_vector(to_unsigned(11111111,32));
167         wait for 50 ns;
168         imgColor_q0 <= std_logic_vector(to_unsigned(500,32));
169         wait for 50 ns;
170         imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
171         wait for 50 ns;
172         imgColor_q0 <= std_logic_vector(to_unsigned(11111111,32));
173         wait for 50 ns;
174         imgColor_q0 <= std_logic_vector(to_unsigned(500,32));
```

```
175         wait for 50 ns;
176         imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
177         wait for 50 ns;
178         imgColor_q0 <= std_logic_vector(to_unsigned(11111111,32));
179         wait for 50 ns;
180         imgColor_q0 <= std_logic_vector(to_unsigned(500,32));
181         wait for 50 ns;
182         imgColor_q0 <= std_logic_vector(to_unsigned(1255555,32));
183         wait for 50 ns;
184          imgColor_q0 <= std_logic_vector(to_unsigned(11255555,32));
185         wait for 50 ns;
186          imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
187         wait for 50 ns;
188          imgColor_q0 <= std_logic_vector(to_unsigned(55550000,32));
189         wait for 50 ns;
190          imgColor_q0 <= std_logic_vector(to_unsigned(255555000,32));
191         wait for 50 ns;
192          imgColor_q0 <= std_logic_vector(to_unsigned(1234568,32));
193         wait for 50 ns;
194          imgColor_q0 <= std_logic_vector(to_unsigned(1234568,32));
195         wait for 50 ns;
196          imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
197         wait for 50 ns;
198         imgColor_q0 <= std_logic_vector(to_unsigned(500,32));
199         wait for 50 ns;
200         imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
201         wait for 50 ns;
202         imgColor_q0 <= std_logic_vector(to_unsigned(11111111,32));
203         wait for 50 ns;
204         imgColor_q0 <= std_logic_vector(to_unsigned(500,32));
205         wait for 50 ns;
206         imgColor_q0 <= std_logic_vector(to_unsigned(1255555,32));
207         wait for 50 ns;
208          imgColor_q0 <= std_logic_vector(to_unsigned(11255555,32));
209         wait for 50 ns;
210          imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
211         wait for 50 ns;
212          imgColor_q0 <= std_logic_vector(to_unsigned(55550000,32));
213         wait for 50 ns;
214          imgColor_q0 <= std_logic_vector(to_unsigned(255555000,32));
215         wait for 50 ns;
216          imgColor_q0 <= std_logic_vector(to_unsigned(1234568,32));
217         wait for 50 ns;
218          imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
219         wait for 50 ns;
220          imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
221         wait for 50 ns;
222         imgColor_q0 <= std_logic_vector(to_unsigned(11111111,32));
223         wait for 50 ns;
224         imgColor_q0 <= std_logic_vector(to_unsigned(500,32));
225         wait for 50 ns;
226         imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
227         wait for 50 ns;
228         imgColor_q0 <= std_logic_vector(to_unsigned(11111111,32));
229         wait for 50 ns;
230         imgColor_q0 <= std_logic_vector(to_unsigned(500,32));
231         wait for 50 ns;
232         imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
233         wait for 50 ns;
234         imgColor_q0 <= std_logic_vector(to_unsigned(11111111,32));
235         wait for 50 ns;
```

```
236         imgColor_q0 <= std_logic_vector(to_unsigned(500,32));
237         wait for 50 ns;
238         imgColor_q0 <= std_logic_vector(to_unsigned(1255555,32));
239         wait for 50 ns;
240          imgColor_q0 <= std_logic_vector(to_unsigned(11255555,32));
241         wait for 50 ns;
242          imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
243         wait for 50 ns;
244          imgColor_q0 <= std_logic_vector(to_unsigned(55550000,32));
245         wait for 50 ns;
246          imgColor_q0 <= std_logic_vector(to_unsigned(255555000,32));
247         wait for 50 ns;
248          imgColor_q0 <= std_logic_vector(to_unsigned(1234568,32));
249         wait for 50 ns;
250          imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
251         wait for 50 ns;
252          imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
253         wait for 50 ns;
254         imgColor_q0 <= std_logic_vector(to_unsigned(11111111,32));
255         wait for 50 ns;
256         imgColor_q0 <= std_logic_vector(to_unsigned(500,32));
257         wait for 50 ns;
258         imgColor_q0 <= std_logic_vector(to_unsigned(500,32));
259         wait for 50 ns;
260         imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
261         wait for 50 ns;
262         imgColor_q0 <= std_logic_vector(to_unsigned(11111111,32));
263         wait for 50 ns;
264         imgColor_q0 <= std_logic_vector(to_unsigned(500,32));
265         wait for 50 ns;
266         imgColor_q0 <= std_logic_vector(to_unsigned(1255555,32));
267         wait for 50 ns;
268          imgColor_q0 <= std_logic_vector(to_unsigned(11255555,32));
269         wait for 50 ns;
270          imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
271         wait for 50 ns;
272          imgColor_q0 <= std_logic_vector(to_unsigned(55550000,32));
273         wait for 50 ns;
274          imgColor_q0 <= std_logic_vector(to_unsigned(255555000,32));
275         wait for 50 ns;
276          imgColor_q0 <= std_logic_vector(to_unsigned(1234568,32));
277         wait for 50 ns;
278          imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
279         wait for 50 ns;
280          imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
281         wait for 50 ns;
282         imgColor_q0 <= std_logic_vector(to_unsigned(11111111,32));
283         wait for 50 ns;
284         imgColor_q0 <= std_logic_vector(to_unsigned(500,32));
285         wait for 50 ns;
286         imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
287         wait for 50 ns;
288         imgColor_q0 <= std_logic_vector(to_unsigned(11111111,32));
289         wait for 50 ns;
290         imgColor_q0 <= std_logic_vector(to_unsigned(500,32));
291         wait for 50 ns;
292         imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
293         wait for 50 ns;
294         imgColor_q0 <= std_logic_vector(to_unsigned(11111111,32));
295         wait for 50 ns;
296         imgColor_q0 <= std_logic_vector(to_unsigned(500,32));
```

```
297         wait for 50 ns;
298         imgColor_q0 <= std_logic_vector(to_unsigned(1255555,32));
299         wait for 50 ns;
300          imgColor_q0 <= std_logic_vector(to_unsigned(11255555,32));
301         wait for 50 ns;
302          imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
303         wait for 50 ns;
304          imgColor_q0 <= std_logic_vector(to_unsigned(55550000,32));
305         wait for 50 ns;
306          imgColor_q0 <= std_logic_vector(to_unsigned(255555000,32));
307         wait for 50 ns;
308          imgColor_q0 <= std_logic_vector(to_unsigned(1234568,32));
309         wait for 50 ns;
310          imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
311         wait for 50 ns;
312          imgColor_q0 <= std_logic_vector(to_unsigned(1234568,32));
313         wait for 50 ns;
314          imgColor_q0 <= std_logic_vector(to_unsigned(255555,32));
315         wait for 50 ns;
316         imgColor_q0 <= std_logic_vector(to_unsigned(11111111,32));
317         wait;
318     end process;
319
320 END;
```

Referencias

[1] M. Betegón. RTL Color to Luminance Images.
, 2018.

[2] Xilinx contents. Vivado HLS Optimization Methodology Guide. *Xilinx, 2017.*

[3] V. Fernández. Ampliación de Sistemas Electrónicos. Módulo 1. Parte 2: Nivel de Transferencia entre Registros. *Grupo de Ingeniería Microelectrónica, Universidad de Cantabria, 2018.*

[4] P.P. Sanchez. Ampliación de Sistemas Electrónicos. Módulo 1. Parte 1 - Implementación de algoritmos descritos en C/C++: Síntesis de Alto Nivel. *Grupo de Ingeniería Microelectrónica, Universidad de Cantabria, 2018.*