FPGA implementation of Image Convolution Filter

Partiendo del código VHDL proporcionado se han hecho una serie de correcciones y modificaciones que faciliten la comprensión del código original y la implementación en la tarjeta ZYNQ Cora Z7.

Como modificaciones generales se han cambiado las líneas en las que se declaran las señales de forma agrupada a individual, como por ejemplo:

```
clk, reset          : in std_logic
```

→

```
clk       : in std_logic;
reset     : in std_logic;
```

También se han sustituido los reset asíncronos por reset síncronos y se han eliminado las señales sobrantes de las listas de sensibilidad y los usos simultáneos de rising_edge y falling_edge, en la siguiente figura se encuentran unos ejemplos:

Original

```
81        --Sum process
82        --Register
83        process(clk, reset)
84        begin
85            if(reset = '1') then
86                r_reg <= (others => '0');
87            elsif(rising_edge(clk)) then
88                r_reg <= r_next;
89            end if;
90        end process;
```

Corregido

```
--Sum process
--Register
process (clk)
begin
    if rising_edge(clk) then
        if reset = '1' then
            r_reg <= (others => '0');
        else
            r_reg <= r_next; -- añadir
        end if;
    end if;
end process;
```

```
25        --Register
26        process(clk, reset, enable)
27        begin
28            if(falling_edge(reset)) then
29                r_reg <= (others => '0');
30            elsif(rising_edge(clk)) then
31                r_reg <= r_next;
32            end if;
33        end process;
```

```
--Register
process (clk)
begin
    if rising_edge(clk) then
        if reset = '1' then
            r_reg <= (others => '0');
        elsif clk_ena = '1' then
            r_reg <= r_next;
        end if;
    end if;
end process;
```

Se han modificado los puertos de entrada y salida del *system_conv*.

```
entity system_conv is
        port(
                filter_sel: in std_logic_vector(1 downto 0);
                reset, global_clock: in std_logic;
                data_read: in std_logic_vector(7 downto 0);
                image_width: in std_logic_vector(12 downto 0);
                image_height: in std_logic_vector(12 downto 0);
                data_write: out std_logic_vector(7 downto 0);
                enable: out std_logic;
                data_ready: out std_logic;

                --Test signals
                clk_test_div: out std_logic;
                pixel_1, pixel_2, pixel_3, pixel_4, pixel_5,
                pixel_6, pixel_7, pixel_8, pixel_9: out std_logic_vector(7 downto 0));

end system_conv;
```

```
entity system_conv is
    generic(
        IMG_WIDTH   : integer; --  1-512
        IMG_HEIGHT  : integer; --  1-512
        IMG_FILTER  : integer -- 0-3
    );
    port(

        reset               : in std_logic;
        global_clock        : in std_logic;
        init_conv           : in std_logic;
        rd_complete         : in std_logic;
        start_of_frame      : in std_logic;
        data_read           : in std_logic_vector(PIXEL_WIDTH-1 downto 0);

        processing          : out std_logic;
        data_write          : out std_logic_vector(PIXEL_WIDTH-1 downto 0);
        enable              : out std_logic;
        data_ready          : out std_logic;
        end_of_frame        : out std_logic
        );
```

Se ha suprimido las entradas *filter_sel, image_width, image_height* y las señales de test ya que con la finalidad de empaquetar el IP Core se ha implementado el tipo de filtro, el ancho y el alto con genéricos. Las señales de test, por su parte, simplemente no son necesarias.

Por otro lado, se han añadido *start_of_frame, init_conv, rd_complete, processing y end_of_frame.*

- *Start_of_frame:* se activa durante un ciclo de reloj tras la recepción del primer pixel para indicar que este pertenece a una imagen nueva.
- *Init_conv:* activa durante un ciclo de reloj cada vez que se recibe de manera correcta un pixel a través de la interfaz AXI e inicia el proceso de cálculo del pixel de salida.
- *Rd_complete:* se activa durante un ciclo de reloj cuando ha terminado el proceso de lectura en la interfaz AXI, permitiendo al sistema ponerse a la espera del siguiente pixel.
- *Processing:* proviene de la señal *enable_tmp* del fichero *enable_write.vhd* y es de utilidad en el wrapper AXI para indicar cuándo se ha llenado el buffer y los píxeles se empiezan a procesar.
- *End_of_frame:* también activa durante un ciclo de reloj cuando ha terminado el proceso de convolución de una imagen completa. También es de gran ayuda durante el test bench para automatizar el fin del mismo.

Por último, también se ha creado el fichero *conv_pkg.vhd* con el objetivo de sustituir los valores *hardcodeados* por constantes más descriptivas del valor en cuestión.

Dando paso a los módulos individuales del sistema de convolución:

1. ***Sel_counter:*** las modificaciones que se le han hecho han sido en pro de que funcione como elemento que controle el proceso general de convolución, y que tanto el comienzo del procesado de un nuevo pixel como la indicación de cuándo está disponible el de salida estén basados en su estado de cuenta. Es por esto que su entidad tiene los siguientes puertos:

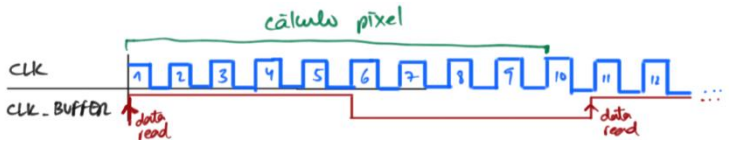
```
entity sel_counter is
    port(

        clk             : in std_logic;
        reset           : in std_logic;
        sof             : in std_logic;
        init_count      : in std_logic;
        rd_complete     : in std_logic;
        in_process      : in std_logic;
        ena_clk_buffer  : out std_logic;
        q               : out std_logic_vector(3 downto 0));
end sel_counter;
```

   La señal *init_count* procede de la señal *init_conv* y se propaga al módulo buffer por medio de *ena_clk_buffer*.

```
process (clk)
begin
    if rising_edge(clk) then
        if reset = '1' then
            r_reg <= (others => '0');
            ena_clk_buffer <= '0';
        else
            r_reg <= r_next;
            ena_clk_buffer <= init_count;
        end if;
    end if;
end process;
```

```
--Next-state logic
r_next <= (others => '0') when ((r_reg = NINE) and (rd_complete = '1')) or sof = '1' else -- reset sum
          (others => '0') when r_reg = NINE and  init_count = '1' else
          r_reg           when ((r_reg = NINE) and (rd_complete = '0')) or (r_reg = 0 and init_count = '1') else -- hold sum
          r_reg + 1; -- update sum
```

El motivo de hacerlo por medio de un registro es que se busca que el estado de cuenta de *r_reg* sea *0* y el de *q* sea *1* cuando se recibe el *init_count* para que queden sincronizados, al igual que lo estarían si se mantuviera la dinámica anterior (con *clock_divider)*:



La lógica combinacional que da valor a *r_next* controla que *sel_counter* no se reinicie hasta que se haya leído el valor del píxel de salida (tras llenar el buffer), que no se reinicie hasta que se haya completado con éxito la operación de escritura AXI, que se mantenga el último estado de cuenta (por lo tanto también el valor del pixel) hasta ser leído o que se mantenga en el primer estado de cuenta hasta que reciba un pixel válido a la entrada, y por último, que en cualquier otro caso se procese el píxel en cada ciclo ya que el pixel de entrada es válido y no se leerá hasta que *r_reg = 9.*

2. ***Buffer module:*** en este caso, las modificaciones que se han hecho son para adaptar el uso de genéricos y del *ena_clk_buffer* proveniente de *sel_counter* al código original.
   Este módulo funciona con la misma señal de reloj que el resto y únicamente aceptará un pixel cuando *sel_counter* se lo indique mediante *ena_clk_buffer.* El tamaño del buffer dependerá del ancho de la imagen que se le haya configurado mediante el genérico, lo cual se verá implementado en *pixel_buffer.vhd.*
   a. ***Pixel_buffer.vhd***
      Se crea la constante *PIXEL_BUFFER_WIDTH* que calcula el tamaño del pixel buffer en función del genérico del ancho de la imagen *IMG_WIDTH* y del ancho de la ventana del kernel *SHIFT_REG_WIDTH.*
      Adicionalmente, se ha eliminado tanto la lógica como los registros correspondientes a *r_next* y se ha implementado directamente dentro del process.
   b. ***Shift_register.vhd***
      Del mismo modo que en el fichero anterior, se ha eliminado todo lo relacionado con *r_next* para implementarlo dentro del process. También se han sustituido las 8 líneas que se utilizaban para asignar valor a *q* por una concatenación.

3. ***Convolution Module*****:** el reset del process que gobierna este módulo se cambió de asíncrono a síncrono. Se añadieron las señales *sof, init_conv, rd_process, in_process* y *ena_clk_buffer* para comunicar *sel_counter* con el resto de módulos, ya que este es un componente de *convolutionModule.* Finalmente se añadió también la implementación de *sof (start_of_frame)* como método de reinicio de *r_next* (registro que almacena el valor del pixel de salida durante los 10 ciclos de procesado.

4. ***Módulo enable_write:*** el cambio más sustancial que se ha realizado en este módulo es, por un lado que ahora es síncrono y está gobernado por la señal de reloj global y ya no es habilitado únicamente por un combinacional, y por otro, la generación de *end_of_frame* (generado con el nombre de *s_strobe_out* en este módulo) y de *edge_data_ready* (detección del flanco de subida de la señal *data_ready*).

   Por la parte de *edge_data_ready,* se genera por medio de un proceso síncrono y un combinacional.

```
process(clk)
begin
    if rising_edge(clk) then
        if reset = '1' then
            data_ready_reg <= '0';
        else
            data_ready_reg <= data_ready; -- FIXME: ask if two flipflops are needed
        end if;
    end if;
end process;

edge_data_ready <= '1' when data_ready = '1' and data_ready_reg = '0' else
                   '0';
```

   Se registra la señal de entrada *data_ready*, la cual indica que *sel_counter* tiene un estado de cuenta *q = 10* y que, por tanto, el valor del pixel de salida es válido. El motivo de detectar únicamente el flanco es porque *sel_counter* se puede mantener en ese estado de cuenta un número indefinido de ciclos hasta que el dato sea leído, por tanto, para evitar que todos los ciclos de reloj en los que *data_ready* se encuentra activa se interpreten como pixeles de salida independientes válidos, se tiene en cuenta únicamente el primer ciclo en el que *data_ready = '1'.* Dado que todos los módulos se encuentran bajo el mismo dominio de reloj, se ha decidido registrar la señal con un único flipflop, si *data_ready* hubiera sido generada bajo un dominio de reloj distinto, habría que pasarla por dos flipflops y dependiendo de la velocidad de cada reloj.

   Por la parte de *s_strobe_out,* se genera por medio de un proceso síncrono y un combinacional y el valor del registro únicamente se actualiza cuando se actualiza este módulo, es decir, con el primer flanco de *data_ready.*

```
process(clk) -- NOTE: added to generate strobe_out
begin
    if rising_edge(clk) then
        if reset = '1' then
            reg_ena <= '0';
        elsif edge_data_ready = '1' then
            reg_ena <= enable_tmp; -- FIXME: ask if two flipflops are needed
        end if;
    end if;
end process;

s_strobe_out <= '1' when reg_ena = '1' and enable_tmp = '0' else -- NOTE: added to generate strobe_out
    '0';
```

   Lo que se detecta en este caso es el flanco de bajada de la señal *enable_tmp* que es la que indica cuándo se está realizando el procesado de la imagen como tal, siendo '0' cuando solo se está llenando el buffer.

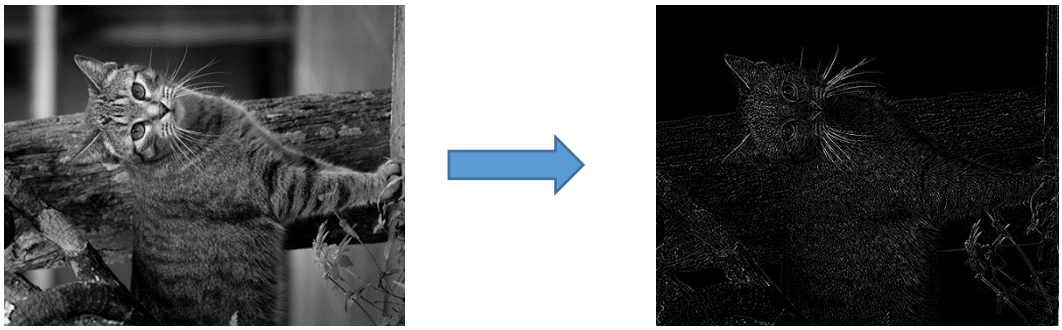
En cuanto a sus módulos internos, tanto en *counter* como en *counter_write* se ha modificado la lista de sensibilidad para que únicamente se incluya *clk*, se han cambiado los reset asíncronos de los process por síncronos y se ha eliminado el uso de *falling_edge(reset)*. Adicionalmente, se ha implementado la señal *start_of_frame* para que sirva también como reinicio de los contadores.

MODIFICACIONES EN EL TEST BENCH

Si bien los test bench individuales han sido modificados durante el desarrollo de la práctica para ir comprobando el funcionamiento de cada uno de los módulos frente a los cambios introducidos, este apartado se verá centrado en el fichero *system_conv_tb.vhd,* el cual sirve como banco de pruebas para el sistema en su totalidad.
En la carpeta *P2\Image-Conv-VHDL-main\vhdl\testbench_files\scripts* se pueden encontrar una serie de scripts para probar los diferentes módulos, siendo ***runsimu_all.do*** el más actualizado ya que es el de todo el sistema y ejecuta el tb del que se habla en este apartado. Cabe mencionar que en el tb han de modificarse los path de los ficheros.
Se ha incluido el *generic map* y se ha modificado el *port map* de la siguiente manera:

```
unit: system_conv
    generic map(
        IMG_WIDTH   => 512,
        IMG_HEIGHT  => 410,
        IMG_FILTER  => 1
    )
    port map(
        global_clock    => clk,
        reset           => reset,
        --filter_sel        => filter_sel,
        init_conv       => init_conv,
        rd_complete     => rd_complete,
        start_of_frame  => start_of_frame,
        --image_width    => image_width,
        --image_height   => image_height,
        processing      => processing,
        data_read       => data_read,
        end_of_frame    => end_of_frame,
        data_write      => data_write,
        enable          => enable,
        data_ready      => data_ready
        );
```

En este test bench se ha eliminado lo relacionado con las señales que ya no se usan, se ha generado un pulso de *start_of_frame* en cuanto se ha leído un dato válido del txt de entrada, se ha generado el primer *init_conv* justamente en el pulso siguiente para comprobar la respuesta a cuando no llegan en el mismo instante y posteriormente, esta señal se activa cada 10 ciclos de reloj.
En el process que maneja la escritura en el fichero de salida se ha añadido una variable para indicar qué número de línea se ha escrito en todo momento, esto ayuda a depurar de una manera rápida ya que si no empieza a contar es que no se ha llegado a escribir, y si se queda en 510 es que solo se ha procesado la primera línea de la imagen.
Por último, se ha añadido un process que atiende a la señal *end_of_frame* y que para el test automáticamente una vez se ha procesado la imagen completa.
Durante las diferentes iteraciones en el proceso de modificación de este proyecto se han testeado de forma exitosa las esperas de *sel_counter* y se ha comparado visualmente las imágenes de entrada con la de salida, siendo en este caso:



En las carpetas *P2\Image-Conv-VHDL-main\data* y *P2\Image-Conv-VHDL-main\outputs* se puede encontrar un script de Python configurado para convertir de txt a png los ficheros *img_3.txt* y *result_cat_3.txt* respectivamente mediante el comando ***py text_2_image.py***.

EMPAQUETADO DEL IP CORE

Una vez codificado y testeado el *system_conv*, se ha hecho uso de Vivado para añadir un AXI wrapper que permita la comunicación entre PS y PL. El proyecto de Vivado que permite editar el IP Package es *P2\wrapper\ip_repo\edit_convolution_ip_v1_0.xpr*.

En este se ha incluido el mapa de genéricos, se ha añadido una salida LED que deje encendido un LED cuando se haga la primera escritura y que deje deje encendido los dos cuando se complete el procesado de la imagen.

Se configuró con 4 registros ya que inicialmente no se habían usado genéricos, sin embargo en la aplicación entregada las señales slv_reg1, slv_reg2, slv_reg3 y slv_reg4 no se usan.

Adicionalmente a las señales generadas por Vivado, se han añadido las siguientes:

```
-- USER SIGNALS
signal sof              : std_logic;
signal eof              : std_logic;
signal first_addr0_wr   : std_logic; -- lo pone a 1 la primera transaccion de escritura a addr0 y lo resetea a 0 el eof
signal rd_complete      : std_logic;
signal s_reset          : std_logic;
signal s_enable         : std_logic;
signal s_data_ready     : std_logic;
signal s_data_write     : std_logic_vector(7 downto 0);
signal s_processing     : std_logic;
```

De las cuales *s_reset* es la señal S_AXI_ARESETN negada, *s_enable* y *s_data_ready* son para asignar esas salidas del *system*_conv a algún lado. Unas señales más interesantes son:

- *First_addr0_wr:* indica si ya se ha hecho la primera escritura en slv_reg0.

```
capture_data: process (S_AXI_ACLK)
variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
begin
  if rising_edge(S_AXI_ACLK) then
    if S_AXI_ARESETN = '0' then
      slv_reg0 <= (others => '0');
      slv_reg1 <= (others => '0');
      slv_reg2 <= (others => '0');
      slv_reg3 <= (others => '0');
      slv_reg4 <= (others => '0');
      first_addr0_wr <= '0';
    else
      loc_addr := axi_awaddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
      if (slv_reg_wren = '1') then
        case loc_addr is
          when b"000" =>
            first_addr0_wr <= '1';
            for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
              if ( S_AXI_WSTRB(byte_index) = '1' ) then
```

- *Sof:* se activa durante un ciclo de reloj cuando se realiza la primera escritura de slv_reg0, que es el registro donde se almacenará el valor del pixel de entrada.

```
sof <= '1' when (slv_reg_wren = '1') and (first_addr0_wr = '0') else
       '0';
```

- *Rd_complete:* se activa durante un ciclo de reloj cuando ha terminado la transacción de lectura AXI.

```
gen_rvalid: process (S_AXI_ACLK)
begin
  if rising_edge(S_AXI_ACLK) then
    if S_AXI_ARESETN = '0' then
      axi_rvalid <= '0';
      axi_rresp  <= "00";
      rd_complete <= '0';
    else
      rd_complete <= '0'; -- se mantiene este siempre a no ser que se cumpla (a
      if (axi_arready = '1' and S_AXI_ARVALID = '1' and axi_rvalid = '0') then
        -- Valid read data is available at the read data bus
        axi_rvalid <= '1';
        axi_rresp  <= "00"; -- 'OKAY' response
      elsif (axi_rvalid = '1' and S_AXI_RREADY = '1') then -- tras esto puede c
        -- Read data is accepted by the master
        axi_rvalid <= '0';
        -- GENERAR FLAG AQUÍ PARA INDICAR SIGUIENTE ESCRITURA
        rd_complete <= '1';
      end if;
    end if;
  end if;
end process;
```

El objetivo de esta capa es que, a partir de las señales de la interfaz AXI, se pueda controlar el flujo de escrituras y lecturas para que no se empiece a procesar el siguiente pixel hasta que se haya leído el resultado del anterior.
Para conseguir esto, utilizaremos el siguiente process para manejar la señal *aw_en*, que es la encarga de preparar al AXI slave para recibir la siguiente transacción de escritura. Por tanto, tenemos dos contextos:
Si *s_processing = '0',* es decir, el buffer se está llenando pero no se está procesando, la única espera que habrá que realizar será a que pasen los 10 ciclos de *sel_counter,* lo cual se traduce en un ciclo a nivel alto de *data_ready.*
Si *s_processing = '1',* es decir, ya se están sacando píxeles procesados, habrá que esperar a que se realice la lectura de los mismos antes de que el slave esté preparado para la siguiente escritura.
Todo esto se ve reflejado en el siguiente process:

```
gen_aw_en: process (S_AXI_ACLK)
begin
   if rising_edge(S_AXI_ACLK) then
       if S_AXI_ARESETN     = '0' then
           aw_en <= '1'; -- ok ya puedo recibir una escritura, axi haz lo tuyo
       else
           if (axi_awready = '0' and S_AXI_AWVALID = '1' and S_AXI_WVALID = '1' and aw_en = '1') then
               aw_en <= '0'; -- no puedo aceptar otra porque ya estoy en proceso de recibir una
           elsif (S_AXI_BREADY = '1' and axi_bvalid = '1'and s_processing = '0' and s_data_ready = '1') then
               aw_en <= '1'; -- vuelve a 1 cuando esta listo para intentar aceptar la siguiente. 100% genera
           -- condicion para que aw_en='1' cuando se llena el buffer y tambien para cuando se procesa
           elsif s_processing = '1' then
               if rd_complete = '1' then
                   aw_en <= '1';
               end if;
           end if;
       end if;
   end if;
end process;
```

Por la parte de la lectura, el slave solo podrá ser leído cuando se haya procesado un pixel, por lo que solo estará preparado para recibir una operación de lectura cuando parta de no estarlo, el master quiera realizar una lectura y haya un pixel válido a la salida para ser leído. Esto se ve reflejado en el siguiente process:

```
gen_arready: process (S_AXI_ACLK)
begin
  if rising_edge(S_AXI_ACLK) then
    if S_AXI_ARESETN = '0' then
      axi_arready <= '0'; -- slave determina si puede ser leido o no
      axi_araddr  <= (others => '1'); -- inicializa señal interna
    else
      if (axi_arready = '0' and S_AXI_ARVALID = '1' and s_data_ready = '1' and s_enable = '1') then
        axi_arready <= '1'; -- aqui slave acepta ser leído MODIFICAR
        axi_araddr  <= S_AXI_ARADDR;     -- se captura que addr quiere leer el master
      else
        axi_arready <= '0';
      end if;
    end if;
  end if;
end process;
```

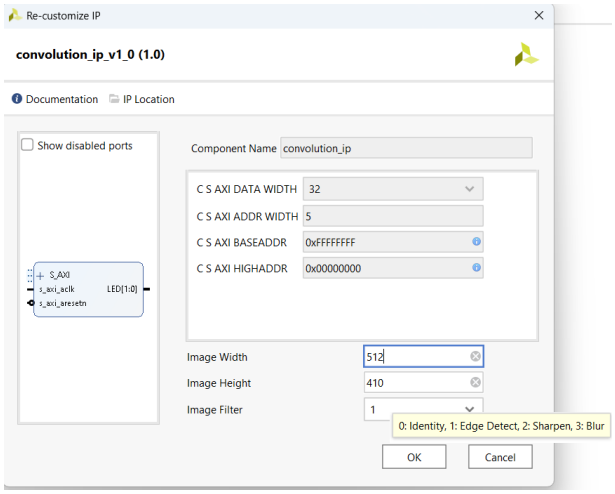
En cuanto al dato que se usa para la lectura, aunque se haga sobre el registro 4 y se indique así en la operación, se tomará el dato que se encuentre en ese momento en el puerto *s_data_write* que es una copia directa de *data_write.*

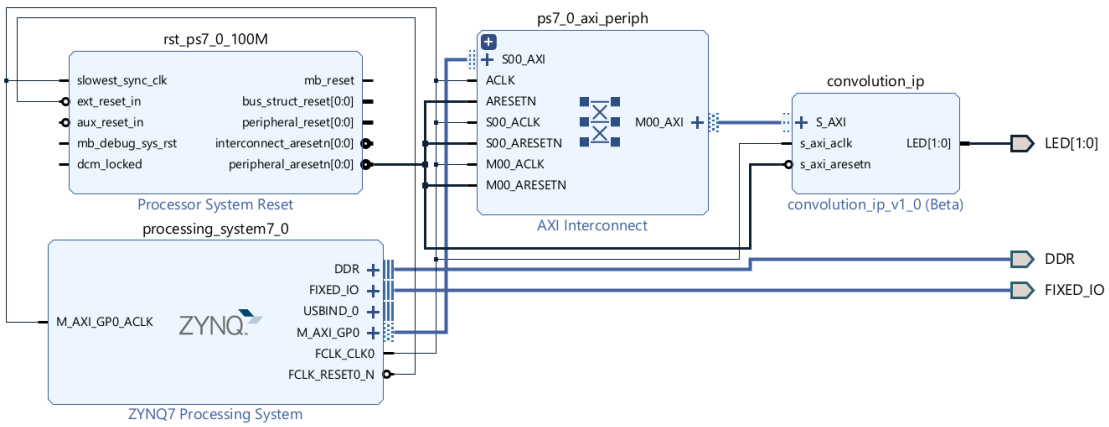
```
process (slv_reg0, slv_reg1, slv_reg2, slv_reg3, slv_reg4,axi_araddr, S_AXI_ARESETN, slv_reg_rden)
variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
begin
    -- Address decoding for reading registers
    loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
    case loc_addr is
      when b"000" =>
        reg_data_out <= slv_reg0;
      when b"001" =>
        reg_data_out <= slv_reg1;
      when b"010" =>
        reg_data_out <= slv_reg2;
      when b"011" =>
        reg_data_out <= slv_reg3;
      when b"100" =>
        reg_data_out <= (23 downto 0 => '0') & s_data_write;
      when others =>
        reg_data_out  <= (others => '0');
    end case;
end process;
```

Por último, las señales que indican cuando se han producido de manera correcta los procesos de escritura y lectura son *slv_reg_wren* y *rd_complete* respectivamente, quedando de la siguiente forma el puerto de conexionado de *system_conv:*

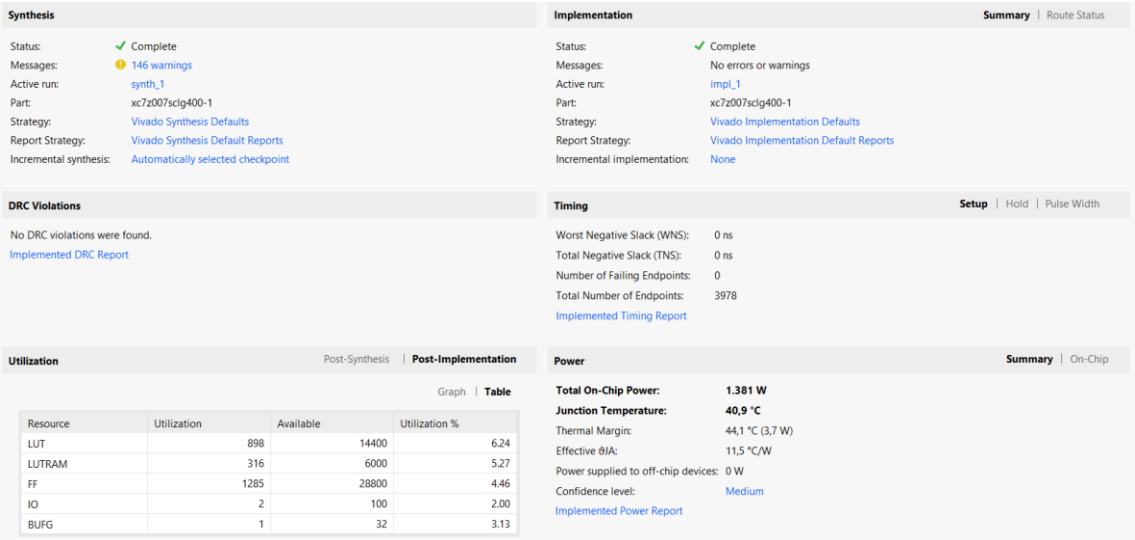
```
-- Add user logic here
app: system_conv
generic map(
        IMG_WIDTH    => IMG_WIDTH,
        IMG_HEIGHT   => IMG_HEIGHT,
        IMG_FILTER   => IMG_FILTER
    )
   port map(
        global_clock     => S_AXI_ACLK,
        reset            => s_reset,
        --filter_sel         => slv_reg1(1 downto 0), -- CHANGED FOR GENERIC
        init_conv        => slv_reg_wren,
        rd_complete      => rd_complete,
        start_of_frame   => sof,
        --image_width     => slv_reg2(12 downto 0), -- CHANGED FOR GENERIC
        --image_height    => slv_reg3(12 downto 0), -- CHANGED FOR GENERIC
        processing       => s_processing,
        data_read        => slv_reg0(7 downto 0),
        end_of_frame     => eof,
        data_write       => s_data_write,
        enable           => s_enable,
        data_ready       => s_data_ready
    );

s_reset <= not S_AXI_ARESETN;
-- User logic ends
```

Con esto, obtenemos el siguiente block design (el cual se puede encontrar en *P2\coraz7_conv_proy\ coraz7_conv_proy.xpr*) una vez hemos empaquetado el ip core:





La síntesis e implementación arroja el siguiente resultado:



Synthesis

Status:	✓ Complete
Messages:	146 warnings
Active run:	synth_1
Part:	xc7z007sclg400-1
Strategy:	Vivado Synthesis Defaults
Report Strategy:	Vivado Synthesis Default Reports
Incremental synthesis:	Automatically selected checkpoint

Implementation — Summary | Route Status

Status:	✓ Complete
Messages:	No errors or warnings
Active run:	impl_1
Part:	xc7z007sclg400-1
Strategy:	Vivado Implementation Defaults
Report Strategy:	Vivado Implementation Default Reports
Incremental implementation:	None

DRC Violations

No DRC violations were found.
Implemented DRC Report

Timing — Setup | Hold | Pulse Width

Worst Negative Slack (WNS):	0 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	3978

Implemented Timing Report

Utilization — Post-Synthesis | Post-Implementation

Graph | Table

Resource	Utilization	Available	Utilization %
LUT	898	14400	6.24
LUTRAM	316	6000	5.27
FF	1285	28800	4.46
IO	2	100	2.00
BUFG	1	32	3.13

Power — Summary | On-Chip

Total On-Chip Power:	**1.381 W**
Junction Temperature:	**40,9 °C**
Thermal Margin:	44,1 °C (3,7 W)
Effective θJA:	11,5 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Implemented Power Report

Finalmente, podemos importar el XSA a la aplicación de Vitis (disponible en *P2\vivado_app*) donde lo que se intenta es realizar una serie de escrituras, sin embargo, esto no se ha conseguido por medio de la lectura de un fichero de entrada y el uso de las librerías *stdio.h* y *stdlib.h* ya que el código se compila y se carga en la placa así que no se puede acceder directamente a un fichero del ordenador.

En respuesta a esto se decidió probar como fichero de origen *img_data.h*, sin embargo, esto no es práctica ya que no se puede tener un .h para cada imagen en todo momento y no arregla el problema para la salida.

Una posible solución sería hacer uso de los puertos USB o microSD, sin embargo, la simple conexión no era suficiente ya que probablemente hubiera que controlarlos de forma manual.

Finalmente, para probar si la escritura se hacía de forma correcta se ha hecho mediante una operación manual de un dato determinado, encendiéndose el primer LED como era de esperar.