Taller No. 3: TUTORIAL DE CONSTRUCCIÓN DE COMPONENTES EN Qsys

Qsys es una herramienta que permite automatizar el proceso de interconexión de componentes para general sistemas complejos (SPoC). El sistema será capaz de interconectar cualquier combinación de componente en la medida de que estos nuevos módulos cumplan la especificación de las interfaces Avalon en cualquiera de sus versiones. El proceso de incluir un componente en Qsys sigue los pasos:

1. Diseñar e implementar un sistema digital
2. Añadir el **circuito envolvente** – *wrapping circuit* - para acomodar las entradas y salidas del sistema a los requerimientos de la interfase Avalon.
3. Utilizar el *Editor de Componentes* de Qsys que incluye la creación de interfaces y la especificación de las propiedades temporales del componente (*timming*).

Para la presente práctica se utilizará un circuito divisor de números enteros con el fin de demostrar el proceso de incluir componentes descritos en HDL en Qsys. El circuito divisor puede considerarse como un acelerador en hardware que incrementará el rendimiento en operaciones de división. Recuerde que la versión económica del procesador NIOS II no incluye multiplicadores ni mucho menos divisores (la versión completa si incluye tales circuitos), por tanto, estas operaciones son realizadas por software. El compilador de C incluirá las rutinas aritméticas respectivas en el momento en que las operaciones son invocadas en el código. Funciones aritméticas, procesamiento de matrices, funciones DSP (MACC, filtros FIR/IIR), procesamiento de señales (transformada de Fourier, filtros de Kalman) procesamiento de imágenes, procesamiento de video, etc., son funciones susceptibles a ser aceleradas utilizando módulos en hardware, que, utilizadas en conjunto con partes del algoritmo escritas en software, generarán una mejora sustancial en el rendimiento global de la ejecución de tales tareas. A continuación, se ilustrarán los tres pasos anteriores siguiendo el proceso de diseño completo.

1. Diseñar e implementar un sistema digital

Debido a su complejidad, la división es una operación aritmética que no puede ser sintetizada directamente sobre código HDL, distinto al caso de la suma, '+', que si permite síntesis utilizando las librerías apropiadas. El algoritmo mostrado a continuación en la Figura 1 ilustra la operación de división entera de 4 bits sin signo.

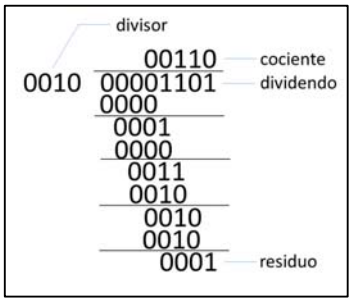

Figura 1. Algoritmo de división de 4-bits sin signo

El algoritmo de división puede resumirse en los siguientes pasos:

1. Doblar el tamaño del dividendo concatenando 0's a la izquierda.
2. Si el correspondiente dividendo es mayor que los cuatro primeros bits del divisor, realizar la resta del dividendo menos el divisor y anotar en el cociente el correspondiente '1'. De lo contrario, mantenga el dividendo original y añada un '0' en el cociente.
3. Añadir un bit adicional en el divisor al resultado previo, y desplace el divisor hacia la derecha en una posición.
4. Repetir los pasos 2 y 3 hasta que todos los bits del dividendo sean usados.

Para implementar el algoritmo de división se utiliza una máquina de estados finitos (FSM) que contiene los siguientes estados: *idle*, *op*, *last*, y *done*. De acuerdo con el código que se muestra a continuación, el divisor es almacenado en el registro *d*, y el dividendo en los registros *rh* y *rl.* En cada iteración, los registros *rl* y *rh* son desplazados a la izquierda una posición. Esta operación corresponde al desplazamiento del divisor a la derecha en el algoritmo descrito anteriormente. Es posible ahora comparar el registro *rh* con el registro *d*. Cuando *rh* y *rl* son desplazados hacia la izquierda, el bit más a la derecha del registro *rl* se hace disponible. Este puede ser utilizado para almacenar el bit actual del cociente. Una vez se han iterado todos los bits del dividendo, el resultado de la última resta se almacena en el registro *rh* y se convierte en el residuo de la división, y todos los cocientes son desplazados en el registro *rl.*

El cómputo más grande se realiza en el estado *op*, en donde los bits del dividendo y el divisor son comparados, sustraídos y finalmente desplazados a la izquierda en una posición. Es importante notar que el residuo no debe ser desplazado en la última iteración. De esta manera se genera un estado adicional, *last*, para acomodarse a este requerimiento especial. Finalmente, el estado *done*, es utilizado para generar la señal *done_tick*, que hace las veces de bandera de finalización del proceso. La señal *done_tick* es levantada por un solo ciclo de reloj.

Ejercicio: Cree un proyecto con nombre "*divisor*". *Escriba y entienda el siguiente código de un divisor entero parametrizable. Escriba un archivo testbench en VHDL para verificar el funcionamiento del circuito divisor (puede utilizar como guía el diagrama de tiempos dela Figura 2).*

```
LIBRARY IEEE;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
--------------------------------------------------------------------------
ENTITY divisor IS
   GENERIC (   W           :  INTEGER  :=8;
               CBIT        :  INTEGER  :=4); --CBIT=log2(W)+1
   PORT    (   clk         :  IN    STD_LOGIC;
               reset       :  IN    STD_LOGIC;
               start       :  IN    STD_LOGIC;
               dvnd        :  IN    STD_LOGIC_VECTOR(W-1 DOWNTO 0);
               dvsr        :  IN    STD_LOGIC_VECTOR(W-1 DOWNTO 0);
               ready       :  OUT   STD_LOGIC;
               done_tick   :  OUT   STD_LOGIC;
               quo         :  OUT   STD_LOGIC_VECTOR(W-1 DOWNTO 0);
               rmd         :  OUT   STD_LOGIC_VECTOR(W-1 DOWNTO 0));
END ENTITY;
--------------------------------------------------------------------------
```

```
ARCHITECTURE rtl OF divisor IS
   TYPE state IS (idle,op,last,done);
   SIGNAL state_reg, state_next  : state;

   SIGNAL rh_reg, rh_next        : UNSIGNED(W-1 DOWNTO 0);
   SIGNAL rl_reg , rl_next       : STD_LOGIC_VECTOR(W-1 DOWNTO 0);
   SIGNAL rh_tmp                 : UNSIGNED(W-1 DOWNTO 0);
   signal d_reg, d_next          : UNSIGNED(W-1 DOWNTO 0);
   signal n_reg, n_next          : UNSIGNED(CBIT-1 DOWNTO 0);
   signal q_bit                  : STD_LOGIC;
BEGIN
   --=======================================
   --    fsmd state and data registers
   --=======================================
   PROCESS(clk,reset)
   BEGIN
      IF (reset='1') THEN
         state_reg   <= idle;
         rh_reg      <= (OTHERS=>'0');
         rl_reg      <= (OTHERS=>'0');
         d_reg       <= (OTHERS=>'0');
         n_reg       <= (OTHERS=>'0');
      ELSIF (rising_edge(clk)) THEN
         state_reg   <= state_next;
         rh_reg      <= rh_next;
         rl_reg      <= rl_next;
         d_reg       <= d_next;
         n_reg       <= n_next;
      END IF;
   END PROCESS;
```

```
   --=========================================
   -- fsmd next-state logic and data path logic
   --=========================================
   PROCESS( state_reg, n_reg, rh_reg, rl_reg, d_reg,
            start, dvsr, dvnd, q_bit, rh_tmp , n_next)
   BEGIN
      CASE state_reg IS
         -----------------------------------------------------
         WHEN idle =>
            ready       <= '1';
            done_tick   <= '0';
            IF (start = '1') THEN
               rh_next     <= (OTHERS=>'0');
               rl_next     <= dvnd;                  --dividend
               d_next      <= UNSIGNED(dvsr);        --divisor
               n_next      <= to_unsigned(W+1, CBIT); --index
               state_next  <= op;
            ELSE
               rh_next     <= rh_reg;
               rl_next     <= rl_reg;
               d_next      <= d_reg;
               n_next      <= n_reg;
               state_next  <= state_reg;
            END IF;
         -----------------------------------------------------
         WHEN OP  =>
            ready       <= '0';
            done_tick   <= '0';
            d_next      <= d_reg;
            -- shift rh and rl left
            rl_next  <= rl_reg(W-2 downto 0) & q_bit;
            rh_next  <= rh_tmp(W-2 downto 0) & rl_reg(W-1);
            -- decrease index
            n_next   <= n_reg - 1;
            IF (n_next = 1) THEN
               state_next <= last ;
            ELSE
               state_next <= state_reg;
            END IF;
         -----------------------------------------------------
```

```
         WHEN last   => --last iteratrion
            ready       <= '0';
            done_tick   <= '0';
            rl_next     <= rl_reg(W-2 downto 0) & q_bit;
            rh_next     <= rh_tmp;
            d_next      <= d_reg;
            n_next      <= n_reg;
            state_next  <= done;
         ------------------------------------------------------
         WHEN done   =>
            ready       <= '0';
            done_tick   <= '1';
            rh_next     <= rh_reg;
            rl_next     <= rl_reg;
            d_next      <= d_reg;
            n_next      <= n_reg;
            state_next  <= idle;
      END CASE;
   END PROCESS;
   --=======================================
   --       Compare and substract
   --=======================================
   PROCESS(rh_reg, d_reg)
   BEGIN
      IF(rh_reg >= d_reg) THEN
         rh_tmp   <= rh_reg - d_reg;
         q_bit    <= '1';
      ELSE
         rh_tmp   <= rh_reg;
         q_bit    <= '0';
      END IF;
   END PROCESS;

   --=======================================
   --                Output
   --=======================================
   quo   <= rl_reg;
   rmd   <= std_logic_vector(rh_reg);

END ARCHITECTURE;
```

2. Circuito envolvente: Interfase Avalon

El circuito divisor tiene dos puertos de entrada (***dvnd*** y ***dvsr***), dos puertos de salida (***quo*** y ***rmd***), una señal de control (***start***) y dos señales de estatus (***ready*** y ***done_tick***). El circuito externo *master* debe ubicar los datos del divisor y el dividendo en los puertos *dvsr* y *dvdn* respectivamente, mientras que al mismo tiempo debe poner en '1' la señal de control ***start*** por un ciclo de reloj. Una vez el cálculo es terminado, el cociente y el residuo serán enviados a los puertos ***quo*** y ***rmd***, mientras que al mismo tiempo se pondrá en '1' la señal ***done_tick*** por un ciclo de reloj. La figura 2 presenta una simulación funcional del circuito divisor donde se ilustra el estado de las distintas señales.

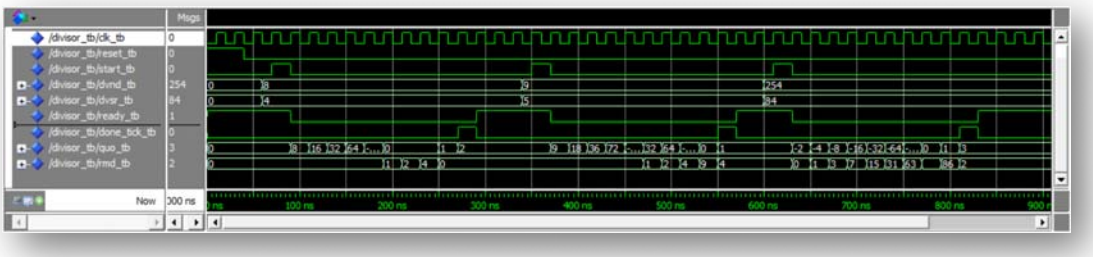

Figura 2. Simulación funcional del circuito divisor entero parametrizable

Adicionalmente a las salidas originales, se pretende conectar los ocho bits del residuo a ocho LEDs en la tarjeta DE2. Este último requerimiento no será utilizado para demostrar el uso de la interfase *Conduit*.

Para diseñar una interfase Avalon utilizando un circuito envolvente (*wrapping circuit*), es necesario analizar las características de los periféricos E/S y determinar los puertos Avalon que son requeridos. A continuación, se listan los siguientes requerimientos para el circuito envolvente del divisor:

- Una interfase de reloj del sistema.
- Una interfase Avalon *MM slave* para acceder a los datos.
- Un puerto de envío interrupción para generar una señal *interruption request* cuando el cálculo haya sido terminado.
- Una interfase *Conduit* para conectar ocho LEDs de la tarjeta DE0.

Dado que el tamaño del bus de datos del procesador Nios II es de 32 bits, resulta razonable instanciar el circuito de división igualmente de 32 bits. Sin embargo, para demostrar el procedimiento de configuración de parámetros en Qsys, se dejarán los valores de W=8 y CBIT=4 al instanciar el divisor en el circuito envolvente.

2.1 Mapa de Memoria

La función principal del circuito envolvente es la interfase Avalon MM *slave*. El primer paso para integrar la nueva *IP* al mapa de memoria de sistema Nios II, consiste en determinar las direcciones para los puertos E/S más relevantes. Desde la perspectiva del procesador Nios II (con su respectiva interfase Avalon MM *master*), estas direcciones corresponderán a una serie de *offsets* sobre la dirección base del periférico. Después de determinar el mapa de memoria, se añadirán una serie de *buffers* necesarios para registrar los datos y una lógica de decodificación y multiplexación.

No existe una manera única de definir los offsets. Por claridad, se acostumbra a utilizar registros individuales, y por tanto *offsets*, para cada señal. Una posible asignación del mapa de memoria se describe a continuación:

- *Write addresses (data from cpu)*
  - *offset 0 (dividend register)*
    - *bits W-1 to 0: dividend data*
  - *offset 1 (divisor register)*
    - *bits W-1 to 0: divisor data*
  - *offset 2 (start register)*
    - *Dummy data used to generate an enable pulse*
  - *offset 6 (**done_tick** register)*
    - *Dummy data used to clear the **done_tick** flag*
- *Read addresses (data to cpu)*
  - *offset 3 (quotient register)*
    - *bits W-1 to 0: quotient data*
  - *offset 4 (remainder register)*
    - *bits W-1 to 0: remainder data*
  - *offset 5 (ready register)*
    - *bit 0: ready status*
  - *offset 6 (done-tick register)*
    - *bit 0: **done_tick** flag*

Los registros del divisor y del dividendo almacenarán los datos de entrada y de hecho se requieren dos registros físicos en el circuito envolvente. El registro *star* es lo que se denomina registro "dummy", lo que significa que existirá un registro físico asociado a este *offset* y los datos de entrada son irrelevantes. Se incluye para mapear la señal *write* proveniente del decodificador que corresponde a un pulso de un ciclo de reloj, que será utilizada como señal *start* para el circuito divisor. Cuando el maestro escribe en esta dirección, la señal de *write* será decodificada y al estar conectada a la señal *start,* el circuito iniciará su operación. Los registros de las señales del cociente, residuo y *ready* almacenarán los datos de salida y la señal de estatus. Dado que el circuito de división contiene registros para almacenar los datos resultado del cálculo, y estos datos no cambian hasta la siguiente operación (ver figura 2), no es necesario incluir registros de salida en el circuito envolvente. El registro *done_tick* será utilizado como registro bandera. Este último, será puesto en ´1´por la señal *done_tick* del divisor y puesto en '0' por el pulso de *write* proveniente del decodificador. El maestro podrá leer este registro para verificar si el cálculo ha sido terminado y escribir en el registro para limpiarlo una vez ha extraído los datos. Para alinear los datos con el sistema Nios II, se acostumbra a definir el tamaño de los registros mencionados en 32 bits. Los bits sin utilizar serán removidos automáticamente durante el proceso de síntesis y no utilizarán hardware adicional.

Un registro E/S es usualmente utilizado bien como registro de entrada o bien como registro de salida, pero no cumple ambas funciones. De esta manera, es posible que un registro de lectura tenga el mismo *offset* de un registro de escritura irrelevante (registro *dummy*). Por ejemplo, el *offset* del registro del cociente puede ser asignado a 0. Para efectos de claridad en la implementación, no se utilizan *offsets* similares para operaciones de lectura y escritura a menos que estas operaciones estén relacionadas, como es el caso del registro *done_tick*. La salida del registro *donde_tick* será utilizada igualmente como señal de *interruption request*. Si se utiliza la función de interrupciones, deberá existir una rutina en el procesador para limpiar la bandera de interrupción, escribiendo datos a esta dirección. Tales datos son irrelevantes y por tanto será un registro *dummy* de escritura.

2.2 Circuito envolvente

A continuación, se presenta el código en VHDL del circuito envolvente para el divisor, el cual incluye una instancia de la *IP* y lógica adicional para almacenamiento de datos, codificación y multiplexación. Los nombres de los puertos E/S corresponden a los requeridos en el estándar de Avalon, pero incluirán un prefijo ***div_***.

Ejercicio: Cree un nuevo proyecto con nombre "*nios_divisor*". Incluya el archivo *"divisor.vhd". Escriba y entienda el siguiente código correspondiente a la interfase Avalon MM Slave para el circuito divisor entero parametrizable. Escriba un archivo testbench en VHDL para verificar el funcionamiento del circuito de la interfase en donde se tengan en cuenta ciclos de escritura de los registros, ciclos de lectura de resultados y limpieza de banderas.*

```vhdl
LIBRARY IEEE;
USE ieee.std_logic_1164.all;
-------------------------------------------------------------------------
ENTITY divisor_avalon IS
   GENERIC (   W              :  INTEGER  :=32;
               CBIT           :  INTEGER  :=6); --CBIT=log2(W)+1
    PORT   (    -- TO BE CONNECTED TO AVALON CLOCK INPUT INTERFACE
               clk            :  IN    STD_LOGIC;
               reset          :  IN    STD_LOGIC;
               -- TO BE CONNECTED TO AVALON MM SLAVE INTERFACE
               div_address    :  IN    STD_LOGIC_VECTOR(2 DOWNTO 0);
               div_chipselect :  IN    STD_LOGIC;
               div_write      :  IN    STD_LOGIC;
               div_writedata  :  IN    STD_LOGIC_VECTOR(W-1 DOWNTO 0);
               div_readdata   :  OUT   STD_LOGIC_VECTOR(W-1 DOWNTO 0);
               -- TO BE CONNECTED TO AVALON INTERRUPT SENDER INTERFACE
               div_irq        :  OUT   STD_LOGIC;
               -- TO BE CONNECTED TO AVALON CONDUIT INTERFACE
               DIV_LED        :  OUT   STD_LOGIC_VECTOR(7 DOWNTO 0));
END ENTITY divisor_avalon;
-------------------------------------------------------------------------
```

```vhdl
ARCHITECTURE avalonMMslave OF divisor_avalon IS
   CONSTANT ZEROS          :  STD_LOGIC_VECTOR(W-2 DOWNTO 0):= (OTHERS => '0');
   SIGNAL   div_start      :  STD_LOGIC;
   SIGNAL   div_ready      :  STD_LOGIC;
   SIGNAL   set_done_tick  :  STD_LOGIC;
   SIGNAL   clr_done_tick  :  STD_LOGIC;
   SIGNAL   dvnd_reg       :  STD_LOGIC_VECTOR(W-1 DOWNTO 0);
   SIGNAL   dvsr_reg       :  STD_LOGIC_VECTOR(W-1 DOWNTO 0);
   SIGNAL   done_tick_reg  :  STD_LOGIC;
   SIGNAL   quo            :  STD_LOGIC_VECTOR(W-1 DOWNTO 0);
   SIGNAL   rmd            :  STD_LOGIC_VECTOR(W-1 DOWNTO 0);
   SIGNAL   wr_en          :  STD_LOGIC;
   SIGNAL   wr_dvnd        :  STD_LOGIC;
   SIGNAL   wr_dvsr        :  STD_LOGIC;
BEGIN
```

```vhdl
   --=======================================
   --       DIVISOR INSTANTIATION
   --=======================================
   div_unit:   ENTITY work.divisor
   GENERIC MAP(   W          => W,
                  CBIT       => CBIT)
   PORT MAP  (    clk        => clk,
                  reset      => '0',
                  start      => div_start,
                  dvnd       => dvnd_reg,
                  dvsr       => dvsr_reg,
                  ready      => div_ready,
                  done_tick  => set_done_tick,
                  quo        => quo,
                  rmd        => rmd);
```

```vhdl
   --=======================================
   --             REGISTERS
   --=======================================
   PROCESS( clk, reset)
   BEGIN
      IF (reset = '1')THEN
         dvnd_reg       <= (OTHERS => '0');
         dvsr_reg       <= (OTHERS => '0');
         done_tick_reg  <= '0';
      ELSIF (rising_edge(clk)) THEN
         ------------------------------------
         IF (wr_dvnd = '1') THEN
            dvnd_reg    <= div_writedata;
         END IF;
         ------------------------------------
         IF (wr_dvsr = '1') THEN
            dvsr_reg    <= div_writedata;
         END IF;
         ------------------------------------
         IF (set_done_tick = '1') THEN
            done_tick_reg  <= '1';
         ELSIF (clr_done_tick = '1' ) THEN
            done_tick_reg  <= '0';
         END IF;
      END IF;
   END PROCESS;
```

```
   --=======================================
   --          WRITE DECODING LOGIC
   --=======================================
   wr_en          <= '1'   WHEN  (div_write='1' AND div_chipselect='1') ELSE '0';
   wr_dvnd        <= '1'   WHEN  (div_address="000" AND wr_en='1')      ELSE '0';   -- offset 0 (dividend register)
   wr_dvsr        <= '1'   WHEN  (div_address="001" AND wr_en='1')      ELSE '0';   -- offset 1 (divisor register)
   div_start      <= '1'   WHEN  (div_address="010" AND wr_en='1')      ELSE '0';   -- offset 2 (start register)
   clr_done_tick  <= '1'   WHEN  (div_address="110" AND wr_en='1')      ELSE '0';   -- offset 6 (done_tick register)

   --=======================================
   -- READ MULTIPLEXING LOGIC (assume W=32)
   --=======================================
   div_readdata   <= quo                     WHEN  div_address="011" ELSE  -- offset 3 (quotient register)
                     rmd                     WHEN  div_address="100" ELSE  -- offset 4 (remainder register)
                     ZEROS & div_ready       WHEN  div_address="101" ELSE  -- offset 5 (ready register)
                                                                           --    bit 0: ready status
                     ZEROS & done_tick_reg;                                -- offset 6 (done-tick register)
                                                                           --    bit 0: done_tick flag

   --=======================================
   --          CONDUIT SIGNAL
   --=======================================
   div_led <= rmd(7 downto 0);   -- assume that W > 7

   --=======================================
   --        INTERRUPT REQUEST SIGNAL
   --=======================================
   div_irq <= done_tick_reg;

END ARCHITECTURE;
```

El código anterior se divide en seis partes. El primer segmento del código corresponde a la instancia del circuito divisor. El segundo segmento de código corresponde a los registros encargados de almacenar las entradas del dividendo y el divisor y de registrar la bandera *done_tick* proveniente del circuito divisor. El tercer segmento del código consiste en la definición de la lógica de decodificación de escritura. El proceso de escritura será controlado por una señal *wr_en* común que estará en '1' cuando la interfase Avalon Master indique *div_write* y *div_chipselect* en '1'. Las demás señales de escritura *wr_dvnd*, *wr_dvsr*, *div_start* y *clr_done_tick* serán puestas en '1' cuando exista *wr_en* y la dirección de escritura corresponda al *offset* correspondiente a los diferentes registros. En el momento en que *wr_dvnd* y *wr_dvsr* son puestos en '1', los correspondientes registros *dvnd_reg* y *dvsr_reg* serán escritos con el dato de entrada correspondiente. La señal *div_start* será conectada a la señal *start* del divisor para indicar el comienzo de la operación. Finalmente, la señal *clr_done_tick* será enviada por el procesador al atender la rutina de atención de interrupción cumpliendo la función de limpieza de la bandera de *IRQ*.

Por otro lado, el cuarto segmento del código corresponde a la lógica de multiplexación en la operación de lectura. Esta lógica utiliza la señal *div_address* como selector para establecer la ruta de los posibles registros que se podrán leer en la señal *div_readdata* correspondiente al bus de datos de lectura en Avalon. Es importante observar, que se han añadido una serie de ceros en los bits más significativos de los registros *ready register* (con *offset* 5) y *done_tick register* (*offset* 6). Por último, el sexto segmento del código conecta el registro *done_tick_reg* con la señal *interuption request*.

2.3 Creación de componentes en Qsys

Es posible añadir componentes creados en HDL en Qsys utilizando la herramienta de configuración llamada como *Component Editor*. Que puede ser llamada directamente desde el Qsys. El procedimiento para incluir un nuevo componente es el siguiente:

1. Abrir el *Component Editor* con el fin de incluir un nuevo componente
2. Especifique los archivos HDL que deben ser incluidos.
3. Crear el mapa de interfases y señales.
4. Configurar las interfases.
5. Definir los parámetros HDL
6. Editar la información de la librería

7. Guardar la información del nuevo componente.

Abrir el programa Component Editor

- Cree un nuevo proyecto llamado a*valon_div*
- Abrir el programa Qsys desde la interfase gráfica de Quartus II
- En Qsys, utilice la opción "*New*" en el panel izquierdo. Puede utilizar de forma alternativa la ruta *File>> New Component.*

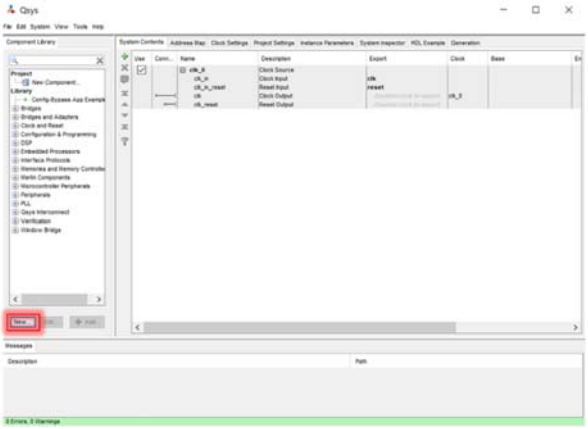

Figura 3. Instanciar nuevo componente en Qsys

- La ventana del editor de componentes se abrirá. Esta ventana tiene varias pestañas de configuración que se describirán a continuación.

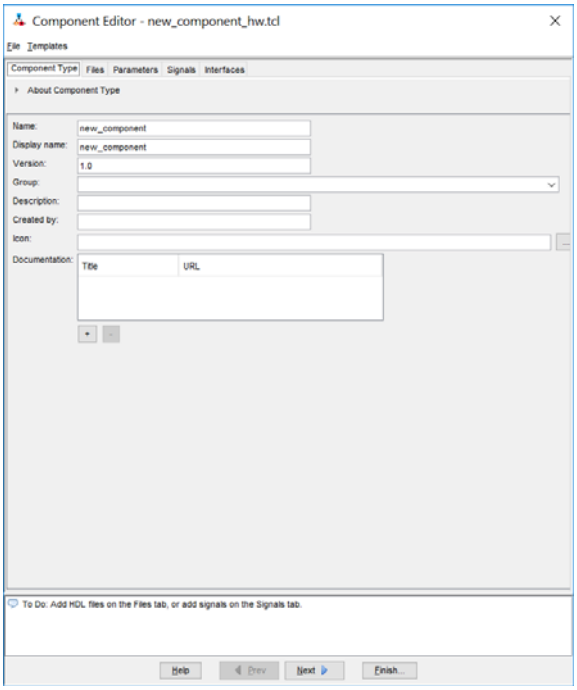

Figura 4. Ventana inicial del editor de componentes

- Seleccione la pestaña "*Files*" para agregar los archivos que componen el nuevo componente.

Especificar los archivos HDL del nuevo componente

- Seleccione el botón "***+***" para navegar en el directorio del proyecto.

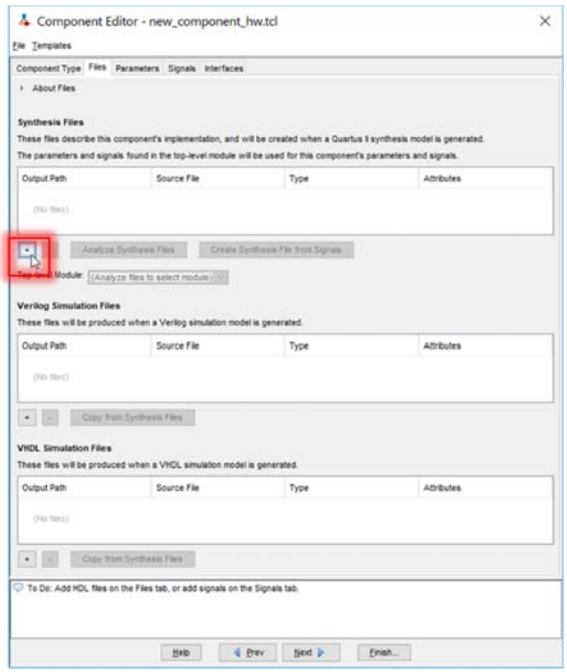

Figura 5. Añadir archivos HDL del nuevo componente.

- Seleccione los dos archivos que componen el divisor.

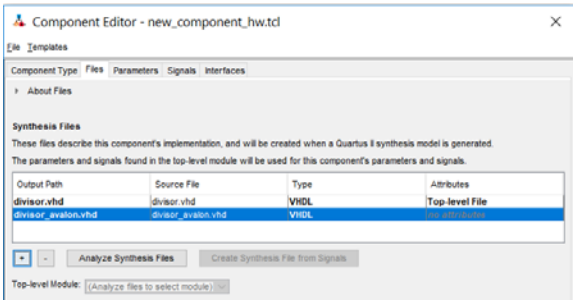

Figura 6. Archivos HDL incluidos

- Qsys es capaz de detectar el archivo que contiene la entidad superior del módulo. Utilice el botón "*Analyze Synthesis Files*". Verifique que la entidad superior corresponda al módulo "*divisor_avalon"*.

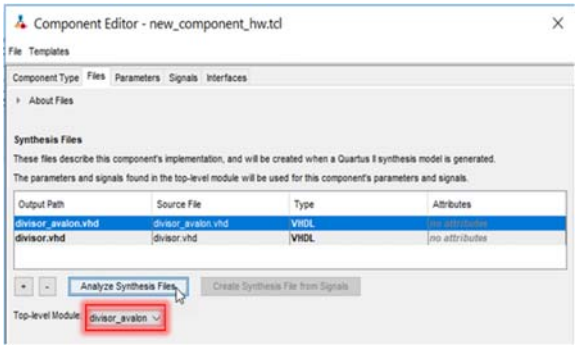

Figura 7. Selección de la entidad superior

- Seleccione la pestaña "*Signals*" para agregar definir las señales de entrada y salida del nuevo módulo.

Crear el mapa de interfases y señales.

Al realizar el análisis de los archivos HDL, el editor de componentes decide automáticamente la función de cada una de las señales E/S de la entidad superior, crea las interfases requeridas y mapea las señales del módulo a las interfases creadas. Esta "decisión" puede no ser siempre correcta, en esos casos es necesario ajustar el mapeo y crear las interfaces que se requieran.

La página inicial de la pestaña de señales se observa en la figura 8. La columna "*Name*" describe las señales de entrada y salida del módulo *divisor_ avalon*. Las columnas "*Interface*" y "*Signal Type*" muestran las interfases asignadas a estos puertos y el mapeo entre las señales E/S y las señales de la interface Avalon MM slave.

Es importante recordar que, para este caso, se intentan incluir cuatro interfases Avalon en el circuito envolvente. El editor de componentes ha detectado algunas de ellas correctamente, que corresponden a la interface del reloj y la interfase Avalon MM *slave*, nombrada como *div* en la columna "*Interface*". Esto sucede debido a que se ha diseñado el circuito evolvente siguiendo de cerca los requerimientos de las señales y los nombres se han utilizado de tal forma que correspondan a los nombres de las señales en Avalon.

Por otro lado, el editor de componentes ha mapeado equivocadamente las señales *div_irq* y la señal *div_led* a otra interface Avalon MM *slave*, nombrada como *avalon_slave_0* en la columna "*Interface*". Para corregir el presente error, es necesario crear una interfase de interrupciones y otra del tipo *Conduit*. El procedimiento es el siguiente:

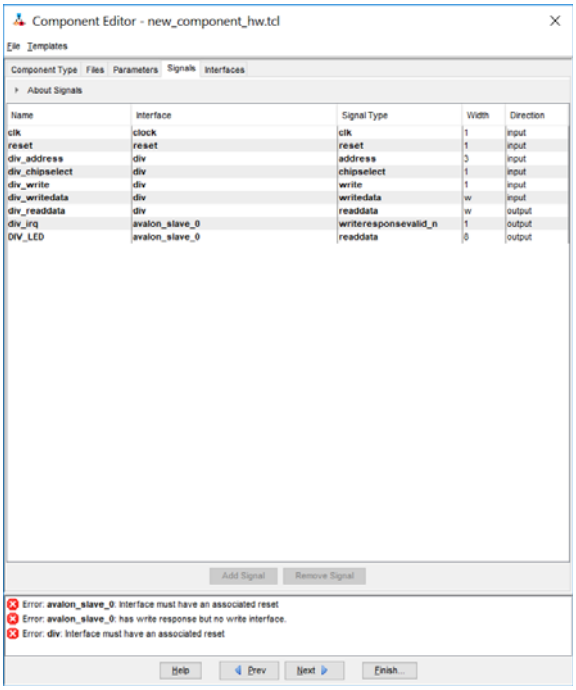

Figura 8. Pestaña de señales

- Seleccione el campo *Interface* en la señal *div_irq* y abra el menú desplegable.
- Seleccione la opción *New Interrupt Sender*, para integrar esta señal a la interfase de interrupciones.

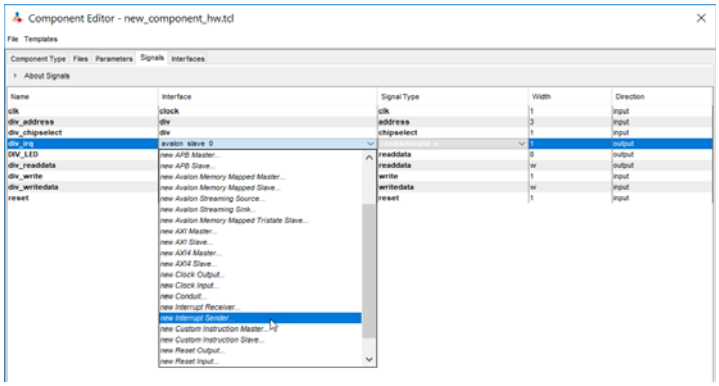

Figura 9. Nueva interface de envío de interrupción

- Seleccione el campo "*Signal Type*" de la señal *div_irq*, y en el menú desplegable seleccione la opción *irq*.

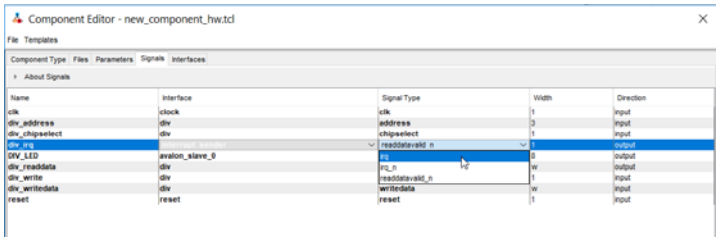

Figura 10. Selección de tipo de señal de interrupción

- Repita el procedimiento anterior para la señal *div_led*, añadiendo una nueva interface tipo C*onduit* y mapeada a una señal a exportar como se observa en la figura 11.

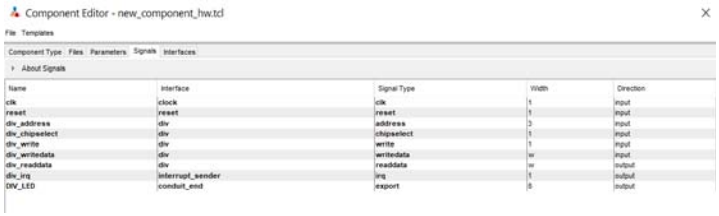

Figura 11. Mapeo de las señales E/S a las diferentes interfaces Avalon.

- Seleccione la pestaña "*Interfaces*" para editar los parámetros de las diferentes interfases seleccionadas.

Configurar las interfases

Cuando una nueva interfase es añadida, Qsys configura los parámetros por defecto. Algunas interfases deberán ser modificadas de acuerdo a los mensajes de error observados en el panel inferior del editor de componentes. En la pestaña "*Interfaces*" es posible verificar en detalle y corregir los errores de configuración que aún se presenten. Es importante prestar especial atención a la interfase Avalon MM *slave* para garantizar una comunicación correcta de los datos entre el procesador (Avalon MM master) y el nuevo módulo.

En la pestaña de "*interfaces*" cada módulo es instanciado de forma individual y una serie de propiedades son listadas para cada una. El procedimiento de configuración es el siguiente:

- El editor de componentes incluyó inicialmente de forma equivocad la interfase "*avalon_slave_0*". Una vez se mapearon correctamente estas señales, la interface

aparece vacía. Para eliminar esta interfase utilice el botón "*Remove Interfaces With No Signals*"

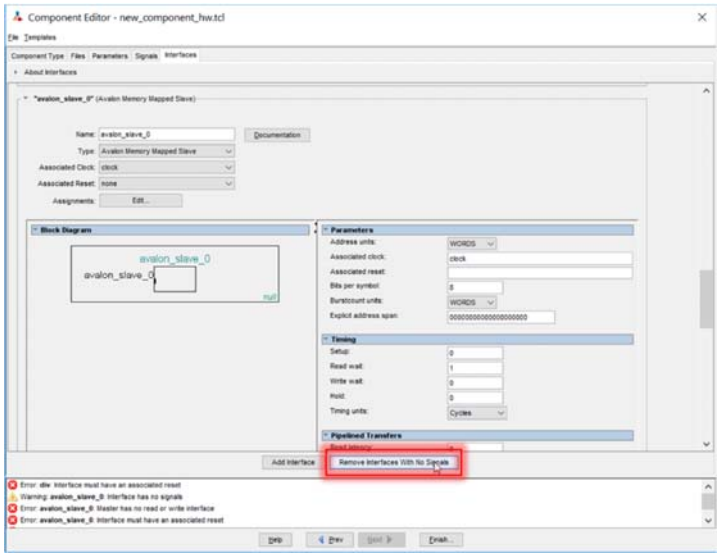

Figura 12. Eliminar las interfaces vacías

- En este caso, las interfaces *clock* y *reset* están correctas y no deben ser modificadas.
- Encuentre la interfase Avalon MM *slave* y cambie el nombre de la misma por *div_cpu* para representar la conexión entre el circuito de división y el procesador.
- En el campo "*Associated reset*" escriba "*reset*" que corresponde al circuito de incialización global. En el campo "*Read wait*" escriba '0' (por defecto viene configurado en '1'). Los datos de lectura están disponibles en los registros del divisor y la operación de lectura no requiere ciclos adicionales de lectura.

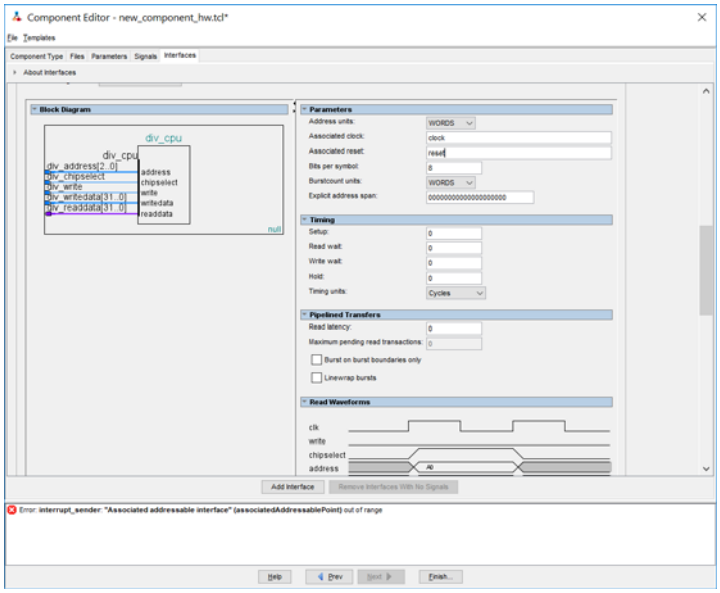

Figura 13. Configuración de la interfase Avalon MM slave

- Cambie el nombre de la interface "*Interrupt Sender*" por *div_intr.*

- Seleccione la opción "*div_cpu*" en el campo "*Associated addressable interface*" para asociar la interrupción generada por el circuito divisor a la interface Avalon MM *slave*.

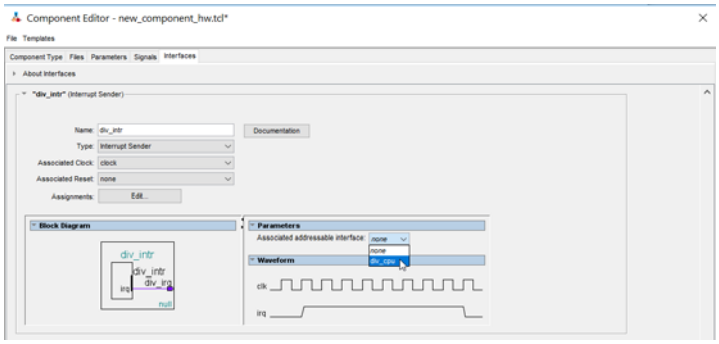

Figura 14. Configuración de la interfase Interrupt Sender

- Cambie el nombre de la interfase tipo *Conduit* a "*div_ledg*". Al final de este proceso, no debe haber errores en el panel inferior del editor de componentes.

Definir los parámetros HDL

Es posible que la entidad superior contenga elementos parametrizables (*generic*) y estos pueden ser configurados en la pestaña "*Paramaters*". En este caso, el módulo *divisor_avalon.vhd* contiene dos parámetros genéricos: *CBIT* y *W*. El editor extrae los valores por defecto instanciados en el HDL original, sin embargo, es posible modificarlos al llamar el componente. Observe en la figura 15, que la opción de editar está habilitada en ambos parámetros. También es posible desactivar esta opción para dejar los parámetros fijos cada vez que se instancie el nuevo componente en Qsys. En este caso se dejarán editables.

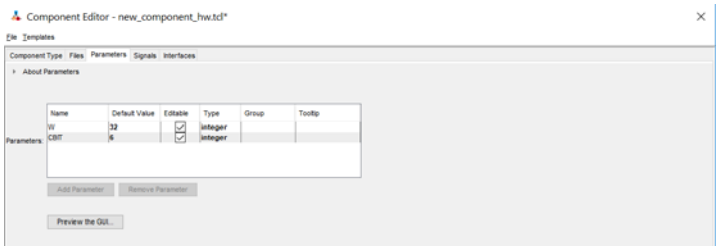

Figura 15. Pestaña de parámetros

Editar la información de la librería

Es posible incluir información de la librería en la pestaña "*Component Type*". Los componentes en Qsys están organizados por categorías. Es posible incluir el nuevo módulo en alguna de las categorías existentes o crear una nueva. En este caso, se incluirá el nuevo componente en la categoría "*others*".

Guardar la información del nuevo componente

La información del componente Qsys es almacenada en un archivo *.tcl*. El nombre del archivo será similar al nombre de la entidad superior en los archivos HDL pero con extensión *.tcl*, siendo este último almacenado en el directorio donde el HDL está localizado. Por ejemplo, en este caso, la entidad superior corresponde al archivo *divisior_avalon.vhd* y el archivo llamado *divisior_avalon.tcl* será creado por Qsys. Es posible salvar ahora el nuevo componente como se observa en la figura 16, pulsando el botón *finish* y salvando el nuevo componente. Es una buena práctica, mantener todos los archivos HDL y TCL en el mismo directorio puesto que se podrá mover el directorio entero del proyecto de un PC a otro.

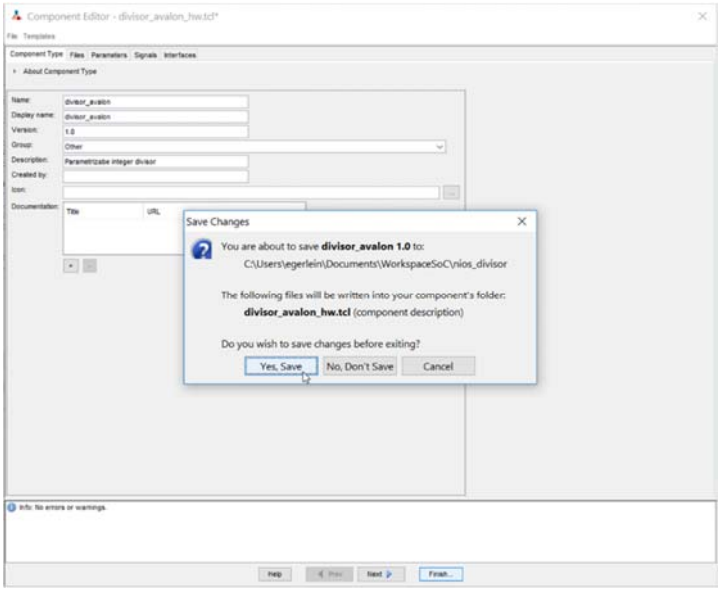

Figura 16. Guardar la información del nuevo componente

3. Sistema NIOS II

Es posible ahora construir un sistema SPoC que incluya el nuevo componente diseñado. En este caso, el circuito divisor será utilizado como acelerador en hardware de la operación de división entera ya que la versión económica del procesador no incluye multiplicadores ni divisores, teniéndose que realizar estas operaciones por software.

Una vez el componente *divisor_avalon* es creado, este puede ser instanciado y procesado de la misma forma que cualquier otro *IP core* de los que Qsys dispone en su librería. La figura 17 muestra el sistema NIOS II en Qsys.

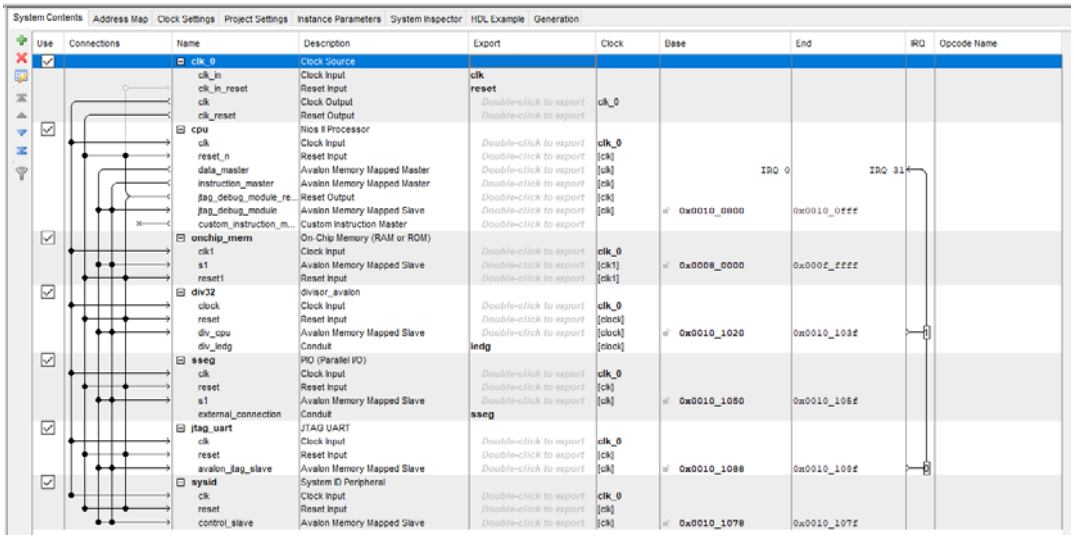

Figura 17. Sistema Nios II que incluye el circuito divisor entero parametrizable

Ejercicio: *Cree un nuevo sistema NIOS II tal y como se observa en la figura 17. Note que aunque el presente taller corresponde a la integración de nuevos componentes con la interfase Avalon MM, aparecen nuevos conceptos los cuales serán profundizados más adelante en el curso. En particular se observa el uso del módulo jtag_uart y la configuración de interrupciones,*

aunque no serán utilizadas en el software desarrollado para el presente taller. Los módulos observados en la figura son los siguientes:

- *Procesador Nios II/e.*
- *Módulo divisor_avalon.*
- *Controlador on_chip memory. Según el rango mostrado en el mapa memorias del módulo SRAM, ¿cuál debería ser el tamaño del módulo de memoria?*
- *Una interfase JTAG UART para obtener los parámetros de entrada y visualizar los resultados. Deje los parámetros definidos por defecto al instanciar el módulo.*
- *Un módulo PIO como interfase de los cuatro displays 7-segmentos. Por facilidad de implementación, cree el PIO como salida con 32 bits.*
- *Módulo system_id.*

4. Verificación del funcionamiento del acelerador de división entera

El procedimiento de creación de un sistema Nios II que incluya el nuevo módulo de división y el desarrollo del software para verificar su operación, se describe en los siguientes pasos:

1. Crear un sistema Nios II que incluya el módulo *divisor_avalon* y los periféricos de adicionales (este punto fue desarrollado en el ejercicio anterior).
2. Crear la entidad superior en VHDL que instancie el sistema Nios II y compilación del proyecto.
3. Desarrollo del software de prueba.
4. Compilar y correr el software.

A continuación, se describen los pasos 2 al 4, dado que la creación del sistema Nios II fue realizada como ejercicio al final de la sección 3.

Crear la entidad superior en VHDL

Una vez Qsys genera los archivos HDL requeridos, es posible generar la entidad superior que instancie el sistema Nios II. El código de la entidad superior se observa a continuación.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.all;
---------------------------------------------------------
ENTITY nios_divisor_top IS
   PORT (
      clk      :  IN    STD_LOGIC;
      ledg     :  OUT   STD_LOGIC_VECTOR(7 DOWNTO 0); --  ledg
      hex3     :  OUT   STD_LOGIC_VECTOR(6 downto 0);
      hex2     :  OUT   STD_LOGIC_VECTOR(6 downto 0);
      hex1     :  OUT   STD_LOGIC_VECTOR(6 downto 0);
      hex0     :  OUT   STD_LOGIC_VECTOR(6 downto 0)  --  sseg
   );
END ENTITY;
---------------------------------------------------------
ARCHITECTURE top OF nios_divisor_top IS
   SIGNAL   sseg_sig :  STD_LOGIC_VECTOR(31 downto 0);
BEGIN

   --Nios II system instantiation
   nios_unit:  ENTITY WORK.nios_divisor
   PORT MAP (  clk_clk        => clk,
               ledg_export    => ledg,
               sseg_export    => sseg_sig,
               reset_reset_n  => '1');

   --Seven segment assigment
   hex3  <= sseg_sig(30 DOWNTO 24);
   hex2  <= sseg_sig(22 DOWNTO 16);
   hex1  <= sseg_sig(14 DOWNTO 8);
   hex0  <= sseg_sig(6  DOWNTO 0);
END ARCHITECTURE;
```

Nótese que los puertos E/S del circuito divisor están conectados al bus Avalon y ya no son visibles al exterior. Únicamente las señales definidas como puertos en la entidad superior serán visibles. Es posible compilar el diseño para obtener el archivo de configuración *.sof*.

Ejercicio: *Incluya la entidad superior nios_divisor utilizando el código anterior. Verifique los puertos expuestos en el archivo nios_divisor.vhd generado por Qsys. Realice la conexión apropiada de pines para los LEDs verdes y los displays 7-segmentos. Compile el proyecto y descargue el bitstream generado en la FGPA.*

Desarrollo del software de prueba

El módulo JTAG UART (*universal asynchronous receiver and transmitter*) es similar a un Puerto serial. Sin embargo, en vez de utilizar la interfase RS-232, los datos son enviados y transmitidos a través del puerto JTAG de la FPGA. Esto elimina el uso de una conexión serial cableada adicional entre la tarjeta de desarrollo y el computador. Desde el punto de vista del procesador, el módulo JTAG UART es manejado como un puerto serial regular que comunica un tren de caracteres entre el PC y la tarjeta. De esta manera, el periférico tiene dos registros de 32 bits, uno de datos y otro de control, como se observa en la figura 17. El registro de datos contiene los siguientes campos:

- ***Data***: este campo contiene el byte que será transmitido o recibido. Durante una operación de escritura, el campo *data* contiene el caracter a ser escrito al buffer de envío. Durante una operación de lectura, el registro *data* contiene el carácter recibido por la USAT.
- ***rv***: campo de 1 bit que determina si el dato es válido.
- ***ravail***: contiene el número de datos almacenados en el buffer de la USART (después de la lectura actual)

Por otro lado, el registro de control contiene los siguientes campos:

- ***re:*** Campo de 1 bit que debe ser puesto en1 para habilitar la función de *read interruption request.*
- ***we***: Campo de 1 bit que debe ser puesto en1 para habilitar la función de *write interrupt request*.
- ***ri***: Campo de 1 bit que indica si existe una interrupción de escritura pendiente.
- ***wi***: Campo de 1 bit que indica si existe una interrupción de escritura pendiente.
- ***ac***: Campo de 1 bit que indica si ha habido actividad en el Puerto JTAG desde la útima vez que el bit fue puesto en '0'.
- ***wspace***. Campo que contiene el número de espacios disponibles en el buffer de la USART.

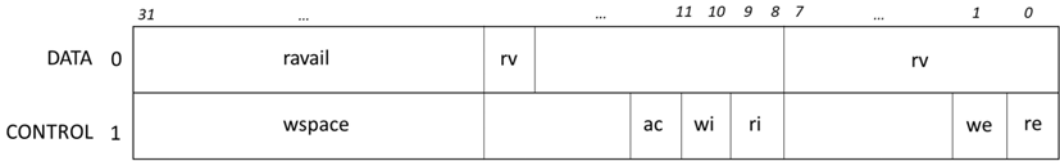

Figura 18. Registros del módulo JTAG UART

Desde el punto de vista de la aplicación de software, la interacción con el módulo JTAG UART involucra la lectura y recepción de datos del buffer. La rutina que se encargará de esta función requiere desplegar en consola los datos obtenidos del circuito divisor.

El código utilizado para la verificación del módulo de división entera, se observa a continuación.

```c
#include <stdio.h>
#include "system.h"

/* register offset definitions */
#define DVND_REG_OFT 0 // dividend register address offset
#define DVSR_REG_OFT 1 // divisor register address offset
#define STRT_REG_OFT 2 // start register address offset
#define QUOT_REG_OFT 3 // quotient register address offset
#define REMN_REG_OFT 4 // remainder register address offset
#define REDY_REG_OFT 5 // ready signal register address offset
#define DONE_REG_OFT 6 // done-tick register address offset

/*****************************************************
            function prototypes
*****************************************************/
alt_u8 sseg_conv_hex(int hex);
void sseg_disp_ptn(alt_u32 base, alt_u8 *ptn);

/* main program */
int main()
{
    alt_u32 a, b, q, r, ready, done;
    alt_u8 dil_msg[4]={sseg_conv_hex(13),0xfb,0xff,sseg_conv_hex(2)};
    sseg_disp_ptn(SSEG_BASE, dil_msg); // display "di 2"
    printf("Division accelerator test: \n\n");
    while (1){
        printf("Perform division a / b = q remainder r\n");
        printf("Enter a: " );
        scanf("%d", &a);
        printf("Enter b: " );
        scanf("%d", &b) ;

        /* send data to division accelerator */
        IOWR(DIV32_BASE, DVND_REG_OFT, a );
        IOWR(DIV32_BASE, DVSR_REG_OFT, b );

        /* wait until the division accelerator is ready */
        while(1) {
            ready = IORD(DIV32_BASE , REDY_REG_OFT) & 0x00000001;
            if (ready==1) { break; }
        }

        /* generate a 1-pulse */
        printf( "Start... \ n " ) ;
        IOWR(DIV32_BASE, STRT_REG_OFT, 1);

            if (ready==1) { break; }
        }

        /* generate a 1-pulse */
        printf( "Start... \ n " ) ;
        IOWR(DIV32_BASE, STRT_REG_OFT, 1);

        /* wait for completion */
        while (1) {
            done = IORD(DIV32_BASE , DONE_REG_OFT) % 0x00000001;
            if (done==1) {break;}
        }

        /* clear done-tick register */
        IOWR(DIV32_BASE, DONE_REG_OFT , 1);

        /* retrieve results from division accelerator */
        q = IORD(DIV32_BASE, QUOT_REG_OFT);
        r = IORD(DIV32_BASE, REMN_REG_OFT);
        printf( "Hardware: %u / %u = %u remainder %u\n", a, b, q, r ) ;

        /* compare results with built-in C operators */
        printf( "Software: %u / %u = %u remainder %u\n\n\n" , a, b, a / b , a%b) ;
    } // end while
}

/*****************************************************************************
function: sseg_conv_hex()
convert a hex digit to a 7-segment pattern
Argument:
    hex :   hex digit (0-15)
Return:
    7-segment display pattern
Note: blank pattern is returned if hex>15
*****************************************************************************/
alt_u8 sseg_conv_hex(int hex){
    static const alt_u8 SSEG_HEX_TABLE[16] = {
        0x40, 0x79, 0x24, 0x30, 0x,19, 0x92, 0x20, 0x78x 0x78, 0x00, 0x10 //0-9
        0x88, 0x03, 0x46, 0x21, 0x06, 0x0E};
    alt_u8 ptn;

    if (hex < 16) {
        ptn = SSEG_HEX_TABLE[hex];
    }
    else{
        ptn = 0xFF;
    }
    return ptn;
}
```

```
/*****************************************************************************
function: sseg_disp_ptn()
display patter in four 7-segment displays
Argument:
    base :  base address of 7-segment display
    ptn  :  pointer to a 4-element array that holds the patterns
Return:
    void
Note:
*****************************************************************************/
void sseg_disp_ptn(alt_u32 base, alt_u8 *ptn){
    alt_u32 sseg_data;
    int i;

    /* form a 32-bit data*/
    for(i=0; i<4; i++){
        sseg_data = (sseg_data<<4) | *ptn;
        ptn++;
    }
    IOWR(base, 0, sseg_data);
}
```

Ejercicio: Genere un proyecto de software nuevo en la herramienta ***Nios II Software Build Tools for Eclipse*** que incluya el código descrito anteriormente. Lea y entienda cada una de las rutinas generadas.

Compilar y correr el software

Ejercicio: *Compile y corra el software desarrollado en el sistema Nios II. Interactúe con el sistema utilizando la consola de Eclipse. Anote los resultados obtenidos.*