Informe de Pasantía: Diseño y Simulación de una Unidad de Conteo de Coincidencias

Santiago Bustamante*
Grupo de Física Atómica y Molecular, Universidad de Antioquia,
Instituto de Física, Medellín, Antioquia, Colombia

Durante la última década, la implementación de sistemas de adquisición de datos basados en FPGAs ha sido de gran interés para la física experimental moderna debido a las características favorables de estos dispositivos. En esta pasantía buscamos solventar un problema de adquisición de datos fundamental en el contexto de la óptica cuántica experimental utilizando este tipo de sistemas. Con este propósito diseñamos una Unidad de Conteo de Coincidencias utilizando el lenguaje de descripción de hardware VHDL. Posteriormente ponemos a prueba el diseño a través de una simulación utilizando el software ModelSim. La simulación muestra que la unidad funciona de la manera esperada.

Keywords: sistemas de adquisición de datos, unidad de conteo de coincidencias, VHDL

I. INTRODUCCIÓN

En los últimos años las demandas experimentales de los grandes equipos modernos destinados al estudio de diversos campos de la ciencia han impulsado el crecimiento de la línea de desarrollo de sistemas encargados de la adquisición y manipulación de datos (DAQ - Data Acquisition Systems). Estos no solo deben adaptarse a la constante evolución de la tecnología, sino también cumplir las demandas de registro de información experimental en tiempo real. En este sentido, factores como la resolución, la precisión, la conectividad de alto rendimiento y la reconfigurabilidad de los DAQs se vuelven de vital importancia [1]. Una solución llamativa a estos retos modernos es el diseño de DAQs basados en FPGAs (Field-Programmable Gate Arrays), dispositivos lógicos programables usualmente comercializados de forma tal que permiten realizar computación heterogénea [2]. Gracias a su alto rendimiento, adaptabilidad, escalabilidad y velocidad, las FPGAs se han vuelto de gran interés para la comunidad científica, especialmente en el área de la física de altas energías [3].

Con el fin de aprender a programar e implementar FPGAs en DAQs, durante una breve pasantía con el Grupo de Instrumentación Científica y Microelectrónica buscamos resolver un problema de adquisición y manipulación de datos que es fundamental en el contexto de la óptica cuántica experimental. Este problema consta del conteo de detecciones simultáneas de pares de fotones distinguibles en un par de detectores de fotones individuales basados en fotodiodos de avalancha. Como en principio cada fotón es convertido en una señal digital, el problema electrónico se reduce al conteo de coincidencias de dos señales lógicas de entrada. En ese sentido buscamos diseñar una Unidad de Conteo de Coincidencias (UCC) de señales lógicas, para lo cuál usamos el lenguaje de descripción de hardware VHDL. Luego, utilizando el software ModelSim, simulamos el sistema de adquisición de datos para verificar que la UCC funciona de forma correcta.

* santiago.bustamanteq@udea.edu.co

II. DISEÑO

Como se mencionó anteriormente, para el diseño de la UCC se utiliza el lenguaje de descripción de hardware VHDL. Creamos entonces un archivo con el nombre *signal_ counter.vhd* que contendrá todo el diseño de la UCC en el lenguaje VHDL. En este archivo escribimos las siguientes líneas de código. Primero, empezamos importando las librerías necesarias y declarando las señales de entrada y de salida del sistema:

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity signal_counter is
6    port (
7      i_clock      : in  std_logic;
8      i_signal1    : in  std_logic;
9      i_signal2    : in  std_logic;
10     o_counter1   : out  std_logic_vector(15 downto 0);
11     o_counter2   : out  std_logic_vector(15 downto 0);
12     o_counter12  : out  std_logic_vector(15 downto 0)
13     );
14 end signal_counter;
```

Aquí, usamos el prefijo `i_` para las señales de entrada y `o_` para las señales de salida. La señal lógica `i_clock` corresponde a aquella asociada al reloj de la FPGA, cuya frecuencia es usualmente de alrededor de los 100MHz.

Las señales lógicas `i_signal1` e `i_signal2` representan las señales provenientes de los detectores de fotones individuales 1 y 2 respectivamente. Las señales de 16 bits `o_counter1` y `o_counter2` dan cuenta del número de fotones individuales registrados por los detectores 1 y 2, mientras que la señal de 16 bits `o_counter12` da cuenta del número de coincidencias entre las señales `i_signal1` y `i_signal2`. Una vez declaradas las señales de entrada y salida, procedemos a definir las señales y constantes necesarias para la arquitectura de la entidad `signal_counter`:

```
architecture arch of signal_counter is

  constant max_count : natural := 65535;
  signal toggle1: std_logic := '1';
  signal toggle2: std_logic := '1';
  signal toggle12: std_logic := '1';
  signal counter1 : natural range 0 to max_count;
  signal counter2 : natural range 0 to max_count;
  signal counter12 : natural range 0 to max_count;
```

La constante `max_count` es el número entero máximo de conteo posible (número entero positivo máximo de 16 bits), las señales `toggle` indicarán cuándo la UCC esté lista para contar los siguientes eventos (registro de fotones individuales o coincidencias) y las señales `counter` serán naturales que llevarán dichas cuentas. Notemos que las señales `toggle` son inicializadas en `'1'`, indicando que la UCC está lista para contar los siguientes eventos. Ahora pasamos a definir la arquitectura de `signal_counter`:

```
begin

  counting : process (i_clock) is
  begin
  if rising_edge(i_clock) then
```

En este bloque de código definimos el proceso de conteo de eventos de forma sincronizada con la señal de entrada del reloj. Es decir que a partir de este momento las siguientes líneas de código se ejecutan de forma paralela cada vez que la señal `i_clock` pasa de `'0'` a `'1'`. Dentro de este proceso añadimos el siguiente bloque de código:

```
    if i_signal1 = '1' and toggle1 = '1' then
      toggle1 <= '0';
      if counter1 = max_count then
        counter1 <= 0;
      else
        counter1 <= counter1 + 1;
      end if;
    end if;
    if i_signal1 = '0' and toggle1 = '0' then
      toggle1 <= '1';
    end if;

    if i_signal2 = '1' and toggle2 = '1' then
      toggle2 <= '0';
      if counter2 = max_count then
        counter2 <= 0;
      else
        counter2 <= counter2 + 1;
      end if;
    end if;
    if i_signal2 = '0' and toggle2 = '0' then
      toggle2 <= '1';
    end if;
```

En este bloque definimos el proceso de conteo de eventos en las señales 1 y 2 de manera individual. En esencia lo que hacemos es programar la UCC de forma que si una de las señales de entrada se encuentra en el valor `'1'` y la UCC está lista para contar dicho evento, entonces se suma un uno a la señal de conteo `counter` correspondiente y la señal `toggle` respectiva pasa a `'0'`, indicando que la UCC deja de estar lista para contar el siguiente evento asociado a dicha señal. Si posteriormente la señal de entrada vuelve a tomar el valor `'0'`, entonces la UCC vuelve a prepararse para contar el siguiente evento en el siguiente ciclo del reloj. Esta arquitectura está específicamente diseñada para contar el número de señales rectangulares individuales de entrada sin importar su ancho.

Para contar las coincidencias se hace un proceso similar, el cual se muestra en el siguiente bloque de código:

```
    if toggle12 = '1' then
      if i_signal1 = '1' and i_signal2 = '1' then
        toggle12 <= '0';
        if counter12 = max_count then
          counter12 <= 0;
        else
          counter12 <= counter12 + 1;
        end if;
      end if;
    else
      if (i_signal1 = '0' or i_signal2 = '0') then
        toggle12 <= '1';
      end if;
    end if;
```

En este caso, la UCC chequea que ambas señales de entrada se encuentren simultáneamente en `'1'`, y si la unidad está lista para contar dicho evento entonces suma un uno a la señal `counter12` y le asigna un `'0'` a `toggle12`. Para que la unidad vuelva a prepararse para

contar el siguiente evento, esta chequea que al menos una de las dos señales tome el valor `'0'`. Finalmente terminamos el diseño de la arquitectura con la transformación de las señales enteras `counter` a señales de 16 bits, las cuales son enviadas a los canales de salida tal y como se muestra en el siguiente bloque de código.

```
  end if;
  end process counting;

  o_counter1 <= std_logic_vector(to_unsigned(counter1,
  ↪  o_counter1'length));
  o_counter2 <= std_logic_vector(to_unsigned(counter2,
  ↪  o_counter2'length));
  o_counter12 <= std_logic_vector(to_unsigned(counter12,
  ↪  o_counter12'length));

end arch;
```

III. SIMULACIÓN

Para hacer la simulación creamos un archivo de testbench con el nombre *signal_ counter_ tb.vhd*. Este archivo nos permitirá crear las señales de entrada deseadas para la simulación. Para esto, primero, importamos las librerías necesarias e inicializamos las señales de entrada con el prefijo `r_` y de salidas con el prefijo `w_`.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.uniform;

entity signal_counter_tb is
end signal_counter_tb;

architecture sim of signal_counter_tb is

  constant c_clock_period : time := 10 ns;

  signal r_clock     :  std_logic := '0';
  signal r_signal1   :  std_logic := '0';
  signal r_signal2   :  std_logic := '0';
  signal w_counter1  :  std_logic_vector(15 downto 0);
  signal w_counter2  :  std_logic_vector(15 downto 0);
  signal w_counter12 :  std_logic_vector(15 downto 0);
```

Notemos que también definimos una constante `c_clock_period` que indica el periodo del reloj de la FPGA. En este caso escogemos un periodo de 10 ns, correspondiente a una frecuencia de 100 MHz. Luego procedemos a definir un puerto que permitirá conectar el diseño *signal_ counter.vhd* con el testbench:

```
  component signal_counter is
    port (
      i_clock     :  in std_logic;
      i_signal1   :  in std_logic;
      i_signal2   :  in std_logic;
      o_counter1  :  out std_logic_vector(15 downto 0);
      o_counter2  :  out std_logic_vector(15 downto 0);
      o_counter12 :  out std_logic_vector(15 downto 0)
      );
  end component signal_counter;
```

Ahora definimos la arquitectura del testbench. Lo primero que debemos hacer es establecer un mapeo entre las señales del diseño con las del testbench, tal y como se muestra en el siguiente bloque de código:

```
begin

  UUT : signal_counter
    port map (
      i_clock => r_clock,
      i_signal1 => r_signal1,
      i_signal2 => r_signal2,
      o_counter1 => w_counter1,
      o_counter2 => w_counter2,
      o_counter12 => w_counter12
      );
```

Luego definimos un proceso `p_clock_gen` que se encargará de simular la señal del reloj.

```
  p_clock_gen : process is
  begin
    wait for c_clock_period/2;
    r_clock <= not r_clock;
  end process p_clock_gen;
```

Para finalizar simulamos las señales de entrada 1 y 2. Como el proceso de conteo de fotones es naturalmente estocástico, buscamos entonces generar señales de entrada aleatorias. Para esto creamos un proceso `p_signal_gen` y utilizamos la función `uniform` de la libreria `ieee.math_real`, la cuál permite generar números reales pseudo-aleatorios distribuidos de manera uniforme entre 0 y 1 a partir de dos semillas naturales. Con esta herramienta simulamos señales aleatorias de la siguiente forma. Primero se deja pasar un intervalo de tiempo aleatorio distribuido uniformemente entre 20 ns y 30 ns, luego del cual la señal lógica `r_signal1` se invierte con una probabilidad de 50 %. Finalmente, luego de otro intervalo de tiempo aleatorio distribuido uniformemente entre 0 ns y 10 ns la señal lógica `r_signal2` se

invierte también con una probabilidad de 50 %. Una vez definido este proceso, queda completada la arquitectura del testbench.

```
  p_signal_gen : process
    variable r : real;
    variable seed1 : positive := 2;
    variable seed2 : positive := 2;
  begin
    uniform(seed1,seed2,r);
    wait for c_clock_period*2 + c_clock_period*r;
    uniform(seed1,seed2,r);
    if r < 0.5 then
      r_signal1 <= not r_signal1;
    end if;
    uniform(seed1,seed2,r);
    wait for c_clock_period*r;
    uniform(seed1,seed2,r);
    if r < 0.5 then
      r_signal2 <= not r_signal2;
    end if;
  end process p_signal_gen;

end sim;
```

Para llevar a cabo la simulación incluimos los archivos de diseño y testbench dentro de un proyecto en el software ModelSim, el cual permite compilar los códigos y simular la UCC con las señales generadas en el testbench. La simulación es realizada durante un tiempo total de simulación de 500 ns, correspondiente a 50 períodos del reloj. En la Figura 1 se pueden apreciar las señales simuladas en el tiempo. La primera señal de arriba hacia abajo corresponde a la señal del reloj del sistema, mientras que las siguientes dos representan las señales de entrada aleatorias que imitan aquellas provistas por los detectores de fotones individuales. Las últimas 9 señales corresponden a los primeros tres bits de las tres señales de salida. Al final de la simulación, las señales de salida son `w_counter1 = '0000000000000011'`, `w_counter2 = '0000000000000100'` y `w_counter12 = '0000000000000011'`, las cuales corresponden a un total de 3 eventos individuales en la señal 1, 4 eventos individuales en la señal 2 y 3 eventos de coincidencia. Por la forma de las señales de entrada simuladas, estos son los resultados esperados del diseño.

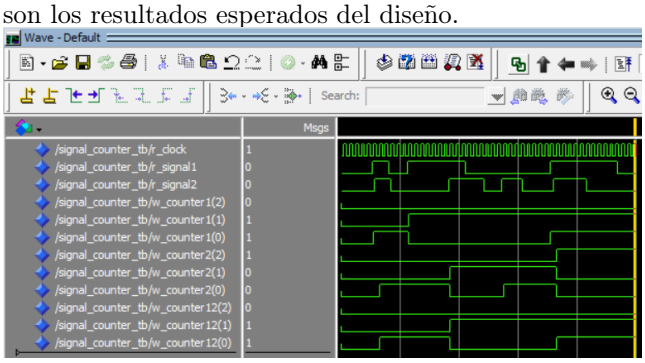

Figura 1. Resultados de la simulación. Las tres primeras señales de arriba hacia abajo son las señales de entrada. El resto son señales de salida. Al final, las señales de salida muestran un conteo de 3 eventos individuales en la señal 1, 4 eventos individuales en la señal 2 y 3 eventos de coincidencia de señales.

IV. CONCLUSIONES Y PERSPECTIVAS

Los resultados obtenidos en la simulación muestran que el diseño utilizado para la UCC funciona de la manera esperada, permitiendo hacer un conteo en paralelo de eventos individuales y de coincidencia. Este diseño fue construido de la forma más simple posible, sin tener en cuenta posibles parámetros experimentales reales de las señales provenientes de los detectores de fotones individuales tales como los tiempos muertos o el ancho de los pulsos. En este sentido el diseño puede ser mejorado para una situación experimental real dada, permitiendo un mejor filtrado de eventos y aumentando así la fidelidad de los datos adquiridos. Como perspectiva con respecto a este trabajo, queda tan solo poner a prueba este diseño a través de la implementación del mismo en una FPGA real. Debido a las limitaciones de tiempo, esto no pudo realizarse durante la pasantía.

AGRADECIMIENTOS

Agradezco al profesor Fabián Castaño y a mi compañero Daniel Estrada por recibirme amablemente en el Grupo de Instrumentación Científica y Microelectrónica y ayudarme con el desarrollo de esta pasantía.

[1] in *Structure of the Stratosphere and Mesosphere*, International Geophysics, Vol. 9, edited by W. L. Webb (Academic Press, 1966) pp. 23–56.

[2] I. Kuon, R. Tessier, and J. Rose, Foundations and Trends® in Electronic Design Automation **2**, 135 (2008).

[3] K. T. Pozniak, Measurement Science and Technology **21**, 062002 (2010).