Generación de números aleatorios para aplicaciones en criptografía cuántica.

Catalina M. Rivera Mesías [*] Andrés Camilo Barajas Ramírez [**]
Asesor: David Guzman
Departamento de Física
Universidad de los Andes

Bogotá, Noviembre 15 de 2011

Resumen

La generación de números aleatorios es muy importante en el cifrado de comunicación y en especial, en la encriptación cuántica de información. Como parte de un proyecto de gran envergadura que tiene ésto como su eje principal, hemos desarrollado dos posibles métodos para la obtención de secuencias binarias que servirán para la codificación de mensajes: la implementación de un generador de secuencias pseudoaleatorias en una FPGA, y la adaptación de este mismo dispositivo como memoria y receptor de eventos ocasionados en la medición de estados entrelazados de probabilidad de 1/2. Presentamos los avances y logros obtenidos, y las dificultades y posibles soluciones que deben ser sobrepasadas para la implementación efectiva de estos sistemas en el proyecto general.

1. Marco Teórico

1.1. Generador de secuencias pseudoaleatorias

Un generador de números pseudoaleatorios es un algoritmo que permite obtener secuencias de números cuyas propiedades se asemejan en alto grado a la de los números aleatorios. Estas secuencias sin embargo son determinadas por condiciones iniciales y por el algoritmo en si mismo. Se utilizan por ejemplo en simulaciones de sistemas físicos en donde algunas de sus propiedades son de carácter aleatorio (como por ejemplo las velocidades de las partículas en un gas).

Una de las características mas particulares de estas secuencias es su carácter reproducible: con el conocimiento del algoritmo utilizado y las condiciones iniciales, se puede obtener la misma secuencia de datos. Esta propiedad es deseable cuando se realizan simulaciones pues permite rastrear fácilmente la presencia de errores, pero es de gran vulnerabilidad cuando es utilizada para encriptar flujos de información.

El carácter pseudoaleatorio de estas secuencias, las hace susceptibles a presentar periodicidad, falta de uniformidad o la presencia de patrones de correlación.

[*] cm.rivera164@uniandes.edu.co
[**] custore@gmail.com

1.2. Generador cuántico de números aleatorios

Es posible utilizar los principios de la mecánica cuántica para crear secuencias aleatorias. [2]Un q-bit es un sistema cuántico de dos niveles que en general puede presentarse en un estado de superposición $|\varphi\rangle = \cos\left(\frac{\theta}{2}\right)|\uparrow\rangle + e^{i\varphi}\sin\left(\frac{\theta}{2}\right)|\downarrow\rangle$ con $0 \leq \theta \leq \pi$ y $0 \leq \varphi < 2\pi$. Cuando se logran obtener experimentalmente y de manera sistemática q-bits cuyas amplitudes de probabilidad sean ambas iguales a $\frac{1}{\sqrt{2}}$ se pueden generar secuencias aleatorias por medio de la medición del observable en el cuál se tiene descrito el estado. Esto presenta una gran ventaja con respecto al método mencionado anteriormente ya que cada uno de los valores que toma la secuencia es impredecible hasta el instante en que se realiza la medición.

Para llevar éstas ideas a la práctica se hace necesario programar una tarjeta electrónica FPGA (Field- Programable Gate Array) que reciba la información de los detectores de fotones y a partir de ella sea capaz de generar la secuencia binaria correspondiente.

El lenguaje de programación que permite programar la tarjeta es VHDL. Una de las ventajas de éste lenguaje es que permite describir el comportamiento del sistema que se quiere modelar, a través de simulaciones sin tener que sintetizar el programa en el hardware real. Esto hace más fácil la verificación y optimización del programa. Una de las diferencias con respecto a los lenguajes de programación estándar como C, Matlab, etc; es que la ejecución del código no es secuencial sino que por el contrario las señales que van siendo modificadas se van evaluando constantemente. Como consecuencia de esto se tiene que por lo general el uso de ciclos como for y while entre otros, no son sintetizables. Usualmente para ejecutar una orden repetidas veces se hace uso de los cambios del reloj que viene incluido en la FPGA.

Ahora bien, para hacer la conexión entre la FPGA y el computador, es decir el usuario, utilizamos [4] LabVIEW (Laboratory Virtual Instrumentation Engineering Workbench); una plataforma y entorno de desarrollo para diseñar sistemas, con un lenguaje de programación visual gráfico. En LabVIEW los programas se diseñan gráficamente, implementando bloques prediseñados con cierta funcionalidad que también pueden ser creados para objetivos particulares. Se deben programar dos partes: el panel frontal, es decir, el interfaz con el usuario que permite visualizar el estado de las variables del problema durante la ejecución del programa; y el diagrama de bloques; que es el programa propiamente dicho, donde se define su funcionalidad. Aquí se colocan íconos que realizan una determinada función y se interconectan entre sí. Para nuestro caso, es importante notar que en LabVIEW los ciclos sí están permitidos.

2. Metodología y Resultados

2.1. Generador de Congruencia lineal

2.1.1. Algoritmo

El generador escogido para realizar la secuencia pseudoaleatoria es el generador de congruencia lineal el cual se caracteriza por ser un algoritmo de sencilla implementación. Este es el generador interno implementado por diversos lenguajes de programación y se define por medio de la siguiente relación de recurrencia:

$$x_{n+1} = \mathrm{mod}(ax_n + c, m) \tag{2.1}$$

en donde a, c y m son valores constantes dentro del generador; mod representa a la función módulo que se define como

$$\mathrm{mod}(r, s) = r - \left\lfloor \frac{r}{s} \right\rfloor s \tag{2.2}$$

tal que $\lfloor . \rfloor$ representa la función parte entera. De allí que se deba cumplir que los valores de c y m sean menores que los de a. La escogencia de estos valores repercute en características del desarrollo de la secuencia, como por ejemplo, su periodicidad: ésta puede ser como máximo el valor de m, y pueden presentarse casos con periodos mucho menores que este valor. También es importante el valor inicial x_n, el cual es denominado como la semilla de la secuencia.En particular, para la implementación realizada se tomaron los siguiente valores

Constante	Valor
m	2^{24}
a	1140671485
c	12820163
x_0	327680

Estos valores fueron tomados del generador de Microsoft Visual Basic, por tener constantes cuyos valores no sobrepasan las capacidades de memoria de la FPGA, y por generar un buen desarrollo de la secuencia [3].

2.1.2. Implementación

El código utilizado para la implementacion de este algoritmo se encuentra en el anexo 4.1. Mediante este código, la FPGA tiene 3 posibles estados determinados por la variable lógica de entrada *reset*, y por el valor de la variable entera interna *counter*, los cuales son resumidos a continuación

Reset	Counter	Resultado
1	0	En espera
0	Menor que *long*	Llenando la secuencia
0	Igual o mayor que *long*	Secuencia llena: en espera

Este proceso se activa con el cambio de las variable *xor_inicie_y_reset* o de la variable *clock*. Cada vez que una de estas se activa, se evalúa el valor de las variables detalladas en la tabla anterior. En particular, si las variables dan para llenar la secuencia, la FPGA determina el siguiente valor de la formula de recurrencia; es decir,en este caso cada vez que se activa el proceso, se halla un nuevo valor en la secuencia. Por este motivo, el proceso se halla atado al reloj de la FPGA, lo cual asegura que el llenado se realizara de forma ininterrumpida. Visto de esta manera, la FPGA va a estar siempre llenando la secuencia, a menos de que *reset* se encuentre activado.

Otro detalle a observar, es la condición del tercer caso, que toma como posibilidad que *counter* sea mayor que la cantidad *long*. Aunque en principio, esto no debería ser posible observando la estructura del programa, se debe tener en cuenta que es posible que el reloj de la FPGA active el proceso, aun cuando las operaciones definidas en su interior no se hayan

realizado completamente. Esto podría ser la causa de que la secuencia obtenida presente bloques grandes de ceros o de unos y por lo tanto se disminuya en gran medida su grado de aleatoriedad. En la figura 1 se muestra el diagrama utilizado para la activación, lectura y escritura de la secuencia obtenida en LabVIEW

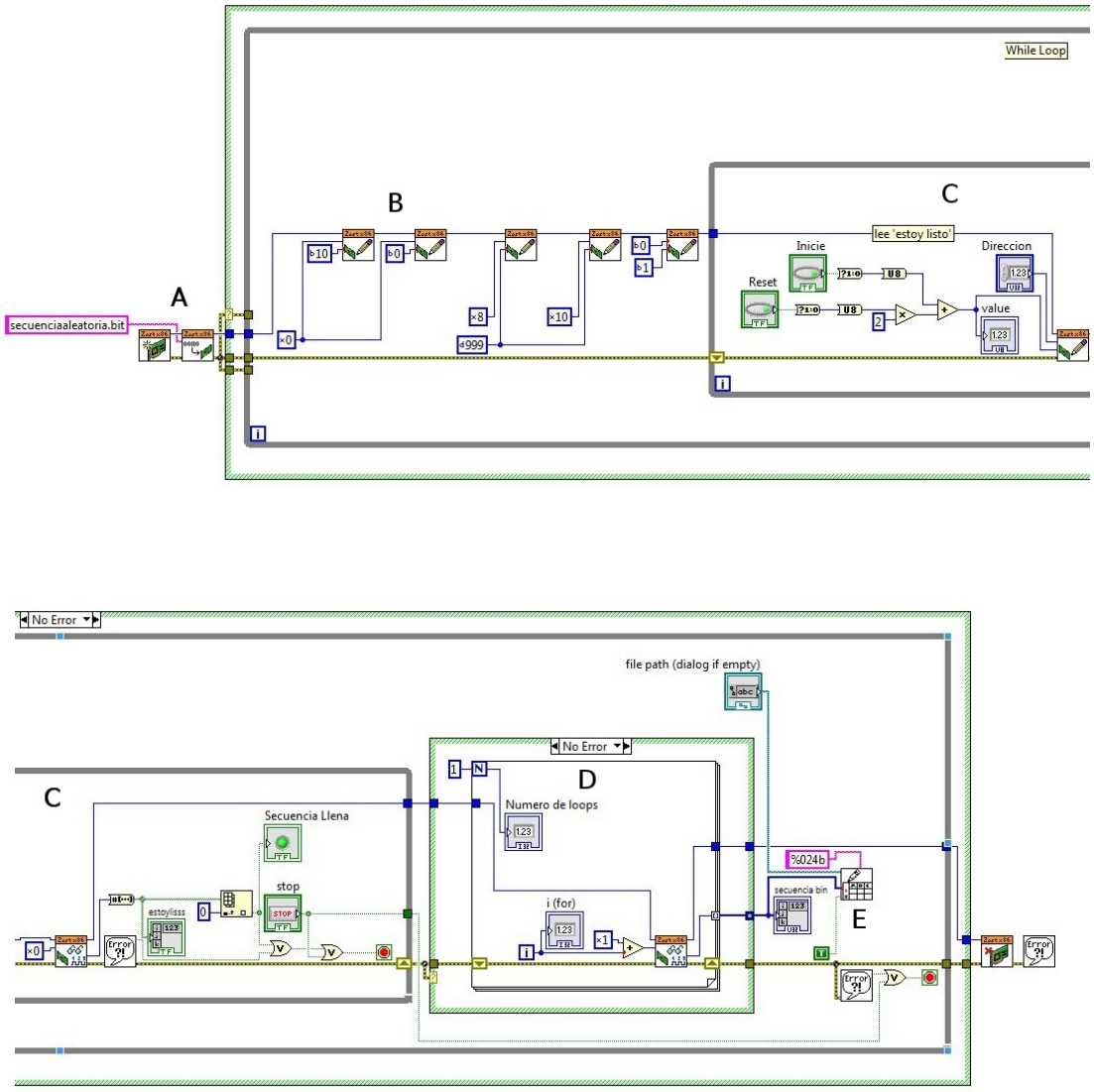

Figura 1: *Diagrama del programa utilizado en LabVIEW*

Este diagrama realiza de forma secuencial el siguiente proceso

A. LabVIEW establece contacto con la FPGA e ingresa el programa que esta ejecutara. Luego de esto entra en el primer bucle general

B. Inicialmente se activa y desactiva *reset* para asegurar que todas las variables comienzan de forma adecuada

C. El programa entra en un bucle interno, en el cual consulta continuamente a la FPGA si la secuencia se lleno, por medio del bloque *ReadCountZest* . De ser así, el programa sale de esta rutina y pasa el siguiente bloque

D. Se realiza la lectura de la secuencia de los datos por medio del bloque *ReadCountZest*. Este bloque en particular, lee segmentos de 24 bits de la secuencia, por lo que el procedimiento debe ser repetido varias veces si la secuencia es mayor tiene mayor longitud; por esto, este bloque es un *for* que repite la operación por el numero de veces que el usuario determine

E. Finalmente, se procede a escribir el valor leído en el archivo escogido en la interfaz.

Este procedimiento se repite de forma continua hasta que se detiene el programa con el botón *Stop* de la interfaz.

2.2. Generación cuántica de secuencias aleatorias

El montaje propuesto consistía en la medición de fotones en estados entrelazados mediante el montaje de la figura 2

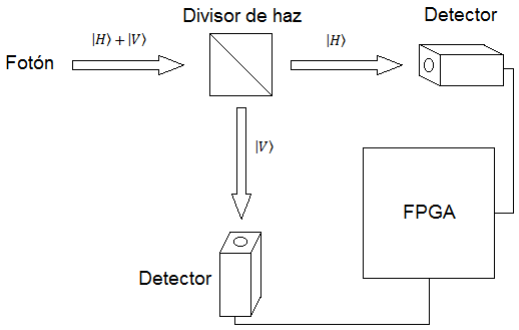

Figura 2: *Montaje propuesto*

En particular, el objetivo era programar la FPGA de forma que esta percibiera y registrara los fotones que llegan a los detectores y lograr establecer comunicación entre la FPGA y el computador (por medio de LabVIEW) para guardar las secuencias obtenidas en este ultimo. Esto implicaba la necesidad de realizar un programa que se activara unicamente cuando se presentaran eventos en los detectores con el objetivo de no perder información y de obtener secuencias aleatorias de alta resolución. Para ello, se decidió programar la FPGA de forma asincrónica, lo cual no solo satisface la necesidad anterior, sino que ofrece un componente adicional particularmente importante en este experimento: velocidad. Por otro lado, dada la velocidad de la llegada de los fotones que se espera, se deseaba que la FPGA guardara secuencias de gran longitud, lo cual minimizaba el tiempo perdido que se produce cuando se realiza la comunicación o la transmisión de datos con el computador.

Por ultimo, como medida de control tanto en la elaboración del programa, como en su posterior utilización, se requería la existencia de dos variables de control, una que indicara el inicio de la toma de datos (*inicie*), y otra que la detuvieran y que la reiniciara (*reset*)

2.2.1. Implementación

El código utilizado para la implementación de lo propuesto descrito anteriormente se encuentra en el anexo 4.2. Este programa consta de dos procesos, uno dirigido por el XOR

entre los receptores y el otro por el XOR entre *inicie* y *reset*. La utilización de esta operación lógica se halla sustentada en la necesidad de que estímulos simultáneos no sean tenidos en cuenta; es decir, si a los detectores llegan al tiempo dos fotones, se desea que no se marque ninguno de los dos en la secuencia. Por otro lado, por razones de funcionamiento, también es útil exigir el mismo comportamiento entre las variables de control *inicie* y *reset*, excluyendo así las posibles combinaciones simultaneas de estas variables de control.

El primer proceso, es el que se activa por los detectores. El comportamiento de este, en el primer nivel, depende de dos variables *borretodo*, la cual es una variable de tipo lógica y *counter* la cual es una variable que toma valores enteros. A continuación se resumen las posibles opciones del programa de acuerdo con estas variables

Borretodo	Counter	Resultado
0	Menor que *long*	Llenando la secuencia
1	0	Valores internos reiniciados. En espera
0	Igual o mayor que *long*	Secuencia llena: en espera

Si las condiciones se dan para que la secuencia sea llenada, se evaluaran los valores de *polh* y *polv*. Dependiendo de esto, se guardará un 0 o un 1 en el vector llamado *seq* y se dejará aumentar el contador para que el próximo valor sea escrito en la próxima posición del vector. Si en cambio, la secuencia ya esta llena, no se realizara ninguna operación: esto con el objetivo de que los datos puedan ser leídos desde el computador. Por ultimo, si el *borretodo* se encuentra en 1, todos las variables internas estarán en sus valores iniciales. Sin embargo, en este estado la secuencia no se llenara y permanecerá en espera.

El segundo proceso esta orientado a controlar al reinicio e inicializacion de la toma de datos, y por tanto depende de valores que el usuario maneja. Si la variable *reset* se activa, *borretodo* tomara el valor de 1 y por tanto toda la secuencia sera borrada y todos los valores reiniciados. En este estado, solo activando la variable *inicie*, se logrará comenzar la toma de datos. Este proceso se puede realizar en cualquier momento, independientemente de que la secuencia haya sido llenada o no.

Vale la pena mencionar un aspecto particular relacionado con el proceso de reseteo e inicio: si inmediatamente después de activar el *reset* se activa el *inicie* (es decir, sin que haya mediado entre ellos un pulso de los detectores), la reinicializacion de los valores no se dara. Esto sucede porque a pesar de que el valor de *borretodo* se convierte en 1, nunca se da la oportunidad de que este active el borrado de las señales en el primer proceso. En un principio se incluyo a *borretodo* dentro de la lista de sensibilidad del primer de proceso, de forma tal que cuando este cambie, se activara seguido el primer proceso. Sin embargo, en las pruebas esto no funciono lo que indicaba que no era una solución viable. No obstante, debe tenerse en cuenta, que para el tipo de experimento al cual esta orientado este programa, no es mayor problema que se presente un evento en los detectores entre la activación de las dos variables de control.

2.2.2. Simulación en Xilinxs

Para evaluar el funcionamiento de este código realizamos varias simulaciones en VHDL, en las cuales queríamos observar el comportamiento del programa para diferentes configuraciones de las señales provenientes de los detectores. El código estaba diseñado para llenar la secuencia con cuatro datos. En la primera simulación (Ver Figura 3) observamos el caso en el cual el pulso del detector dos empieza durante el pulso del detector uno. Aquí se obtiene

lo deseado, es decir, las dos señales aportan por separado, la secuencia se llena con cuatro pulsos y reinicia cuando la señal de reset es uno.

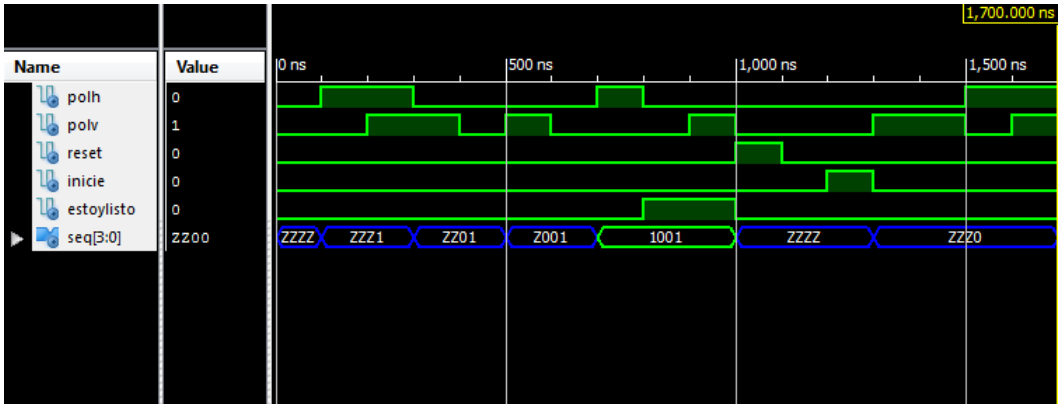

Figura 3: *Simulación con Xilinxs pulsos intersectados.*

Entre las otras configuraciones simuladas, tenemos el caso en el cual las señales llegan al tiempo y un pulso es más largo que el otro; en cuyo caso solo el pulso largo aporta a la secuencia. Otra configuración como el de la Figura 4 en donde la señal para el detector dos viene con un pequeño ruido durante el pulso proveniente del detector uno. Estas dos configuraciones resultan ser problemáticas, ya que, por ejemplo en el último caso se cuenta dos veces el mismo pulso.

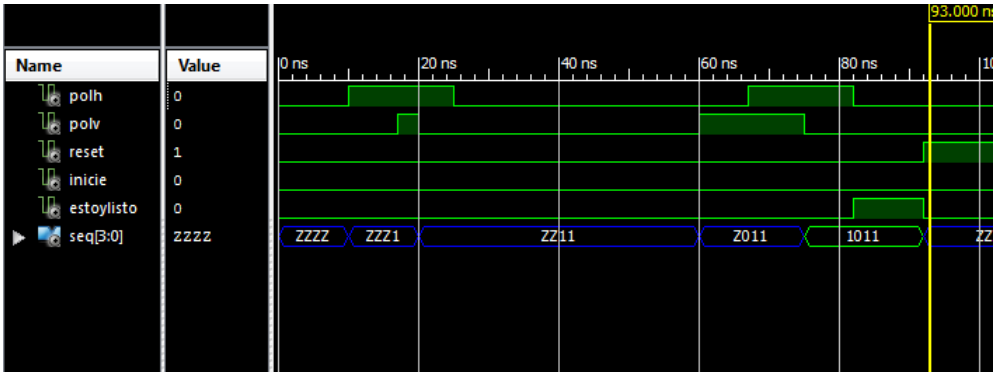

Figura 4: *Simulación con Xilinxs pulsos intersectados.*

2.2.3. Simulación en LabVIEW

Una vez sintetizado el código en VHDL, generamos el archivo.bit que es el que LabVIEW utiliza para establecer la conexión con la FPGA. Antes de generarlo cambiamos la longitud de la secuencia a 24. Utilizamos entonces, el código de LabVIEW (Ver Figura 1). Para hacer una simulación manual, tuvimos que modificar el archivo principal del código en VHDL. Así, las señales que deberían provenir de los detectores las asignamos manualmente a través del llamado a las direcciones de memoria 3008, 3009, 3010, (Ver anexos sección 4.3). Por ejemplo, si desde la interfaz el usuario llamaba la dirección 3008, las señales Detectorh y Detectorv tomaban los valores 1 y 0 respectivamente. Posteriormente cuando el usuario

llamaba la dirección 3010 ambas señales volvían a 0. De ésta manera y utilizando la dirección de memoria 3009 para caracterizar las señales de manera opuesta a cómo se activaron en la dirección 3008, pudimos llenar manualmente la secuencia binaria. Es importante notar que el ciclo externo que posee el código de LabVIEW inicializa las variables del código en VHDL para que una vez llena la secuencia de longitud 24 se pueda volver a crear otra secuencia y poder concatenar secuencias de tamaño 24 formando así una sola secuencia de gran longitud.

En ésta simulación los resultados fueron positivos, el llenado de la secuencia se realizaba acorde a lo requerido por el usuario desde la interfaz, la variable estoylisto se prendía cuando la secuencia estaba llena y se lograba concatenar varias secuencias de 24 sin problema.

2.2.4. Prueba con detectores

Finalmente realizamos una prueba conectando las señales de entrada a los detectores de fotones cuando éstos registraban conteos de fondo, es decir, cuando registraban únicamente entre 50-100 conteos por segundo.
En éste punto, tuvimos problemas bastante extraños y el programa no registró lo que esperábamos sino que se creaban secuencias de sólo ceros sin hacer distinción de si el pulso recibido venía de un detector o de otro. Llegamos a considerar como un posible problema, el hecho de que la longitud de los pulsos fuera de unos 15 ns, es decir, anchos muy cortos y la FPGA no alcanzaba a leer el pulso. Sin embargo, para identificar el problema hicimos asignaciones de una de las señales que venía de los detectores y del primer valor que tomaba la variable secuencia (donde se guardaban los datos), para que se conectarán a puertos de salida y poder monitorearlos desde el osciloscopio. La figura 5 muestra el pulso y los dos valores entre los que oscilaba la segunda señal, demostrando que aparentemente la variable secuencia en algunas ocasiones sí tomaba valores de uno, pero éstos no estaban siendo registrados correctamente en el archivo de escritura. Las figuras 6 y 7 muestran la señal para un sitio de la secuencia a lo largo del tiempo, donde aparentemente se observa de nuevo que su valor sí cambia de cero a uno constantemente. Intentamos encontrar la fuente del error de múltiples formas, pero no lo logramos. Por ahora,la buena noticia es que al parecer en algún momento la secuencia sí se crea de ceros y unos de manera aleatoria.

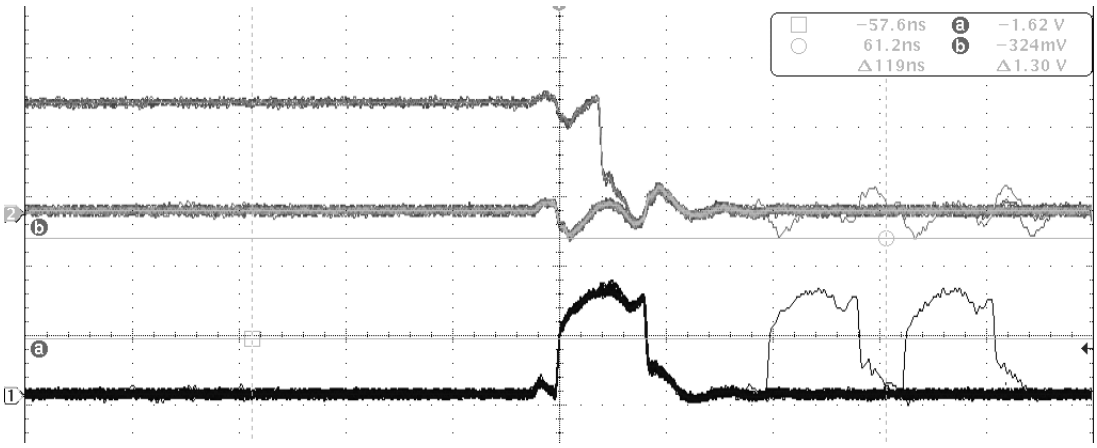

Figura 5: *Imagen del osciloscopio usando trigger respecto a la señal del pulso proveniente de uno de los detectores(inferior). Señal del primer sitio del vector de variables que forma la secuencia(abajo)*

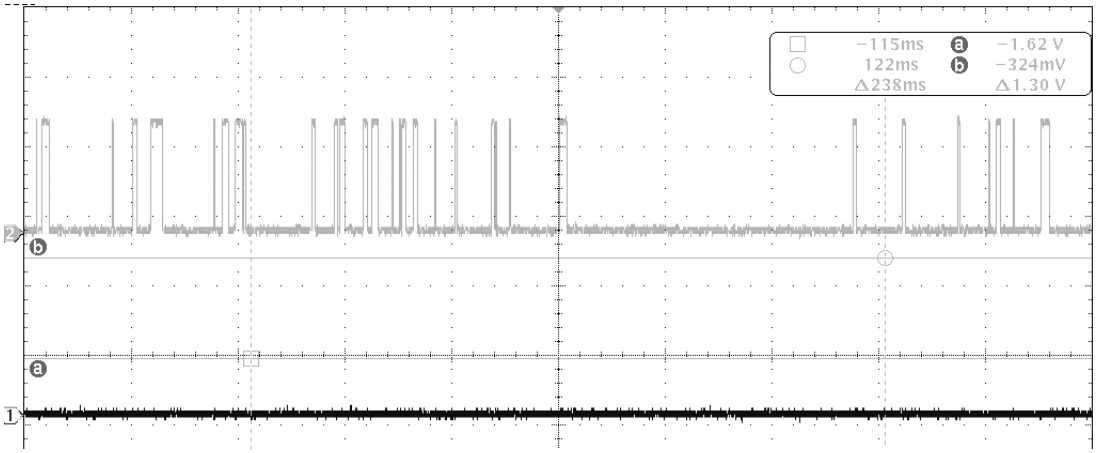

Figura 6: *Imagen del osciloscopio de la señal de un sitio de la secuencia en el tiempo.*

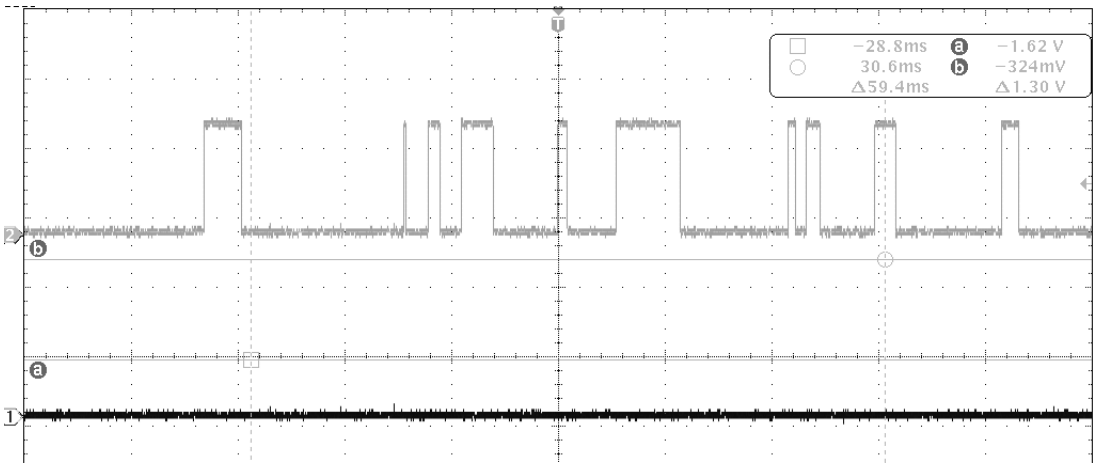

Figura 7: *Imagen del osciloscopio de la señal de un sitio de la secuencia en el tiempo.*

Debido a que la fuente de error no se ha identificado aún, el programa no pudo ser evaluado en el montaje experimental 2 y las secuencias aleatorias no se pudieron generar; sin embargo consideramos que no estamos muy lejos de hallar la solución al problema.

2.3. Pruebas de aleatoriedad

Para probar la calidad de las secuencias obtenidas, se pueden realizar pruebas sobre las mismas que evalúen diferentes parámetros como por ejemplo, la frecuencia con la que se presentan ceros y unos, o las frecuencias que se presentan en la transformada de Fourier de la secuencia. El Instituto Nacional de Estándares y Tecnología de los Estados Unidos (NIST), tiene una división de seguridad informática la cual ha desarrollado un conjunto de pruebas estadísticas para secuencias de datos, para aplicaciones en criptografía. El documento que detalla las características de cada prueba puede ser descargado de su pagina web[1]. Inicialmente se planeaba programar las pruebas para su posterior uso, y para complementar algunas que ya habían sido desarrolladas por estudiantes de laboratorio intermedio [5]; sin embargo, de ésta página web puede descargarse un software (sts-2.1) que permite realizar

todas las pruebas simultáneamente, o cada una por aparte. La instalación y el manejo de este software se detalla en el capitulo 5 del documento mencionado. Nosotros realizamos la instalación y posterior utilización de este paquete de software sobre secuencias de prueba dentro del mismo. En general, su utilización es relativamente sencilla. Sin embargo vale la pena mencionar que éste esta diseñado para sistemas operativos basados en UNIX (como linux o Mac OS).

3. Conclusiones

- Se logró sintetizar el código para la generación de secuencias pseudoaleatorias e implementarlo en el programa de LAbVIEW. Sin embargo la secuencias obtenidas no eran aleatorias debido a problemas relacionados con la velocidad de lectura de la FPGA, que no se han podido solucionar.

- Se logró sintetizar el código para la generación cuántica de secuencias aleatorias. Se realizaron simulaciones internas (en Xilinxs)para evaluar el funcionamiento del código y los resultados obtenidos fueron los deseados. Es importante notar que las simulaciones se hicieron usando las escalas de tiempo razonables para la descripción de los pulsos provenientes de los detectores.

- Se logro generar el archivo.bit para implementar el código en LabVIEW y se realizaron simulaciones de carácter manual para evaluar el acoplamiento de los dos códigos (VHDL y LabVIEW). Los resultados fueron positivos, se generaban las secuencias como se esperaba.

- Las pruebas con los detectores de fotones cuando éstos solo registraban cuentas oscuras no fueron del todo exitosas. A pesar de que se logró comprobar por medio de un osciloscopio que efectivamente se estaba leyendo la información proveniente de los detectores como era debido, el archivo de escritura solo está produciendo secuencias de ceros. Este problema aún no ha sido resuelto. Creemos que tiene que ver con la velocidad con la que la FPGA escribe en las direcciones de memoria.

- Se redifinió el objetivo relacionado con el desarrollo de las pruebas de aleatoriedad en vista de la existencia de un software desarrollado por el NIST, el cual realiza estas pruebas. Se instaló y se realizaron pruebas con este paquete, sin mayor dificultad.

4. Anexos

4.1. Código del generador de la secuencia pseudoaleatoria

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY secuencia IS
generic ( long : integer);
PORT(
                polh                : IN    STD_LOGIC; --uno
                polv                : IN    STD_LOGIC; --cero
                reset           : IN    STD_LOGIC;
                inicie           : IN    STD_LOGIC;
                estoylisto  : OUT   STD_LOGIC := '0';
                seq             : OUT   STD_LOGIC_VECTOR(long-1 downto 0):= (others => 'Z')
                );
END secuencia;

ARCHITECTURE registro OF secuencia IS
        SHARED VARIABLE counter        :  NATURAL := 0;
        SIGNAL xor_h_y_v               :  STD_LOGIC;
        SIGNAL xor_inicie_y_reset      :  STD_LOGIC;
        SIGNAL borretodo               :  STD_LOGIC := '0';

        BEGIN

        xor_h_y_v <= polh XOR polv;
        xor_inicie_y_reset <= inicie XOR reset;

        PROCESS(xor_h_y_v,borretodo)
                BEGIN
                IF(borretodo = '0' AND counter < long) THEN
                        IF(rising_edge(xor_h_y_v)) THEN --tam-vec
                                IF (polh = '1') THEN
                                        seq(counter) <= '1';
                                        counter := counter + 1;
                                ELSIF (polv = '1') THEN
                                        seq(counter) <= '0';
                                        counter := counter + 1;
                                END IF;
                        END IF;
                ELSIF(borretodo = '1') THEN
                        estoylisto <= '0';
                        counter := 0;
                        seq <= (others => '0');
                ELSIF (counter = long) THEN --tam-vec
                        estoylisto <= '1';
                END IF;
```

```
        END PROCESS;

        PROCESS(xor_inicie_y_reset)
                BEGIN
                IF (reset = '1') THEN
                        borretodo <= '1';
                ELSIF (inicie = '1') THEN
                        borretodo <= '0';
                END IF;
        END PROCESS;
END registro;
```

4.2. Código del receptor

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY secuencia IS
generic ( long : integer);
PORT(
                polh        : IN    STD_LOGIC; --uno
                polv        : IN    STD_LOGIC; --cero
                reset       : IN    STD_LOGIC;
                inicie      : IN    STD_LOGIC;
                estoylisto  : OUT   STD_LOGIC := '0';
                seq         : OUT   STD_LOGIC_VECTOR(long-1 downto 0):= (others => 'Z')
                );
END secuencia;

ARCHITECTURE registro OF secuencia IS
        SHARED VARIABLE counter        :  NATURAL := 0;
        SIGNAL xor_h_y_v               :  STD_LOGIC;
        SIGNAL xor_inicie_y_reset      :  STD_LOGIC;
        SIGNAL borretodo               :  STD_LOGIC := '0';

        BEGIN

        xor_h_y_v <= polh XOR polv;
        xor_inicie_y_reset <= inicie XOR reset;

        PROCESS(xor_h_y_v)
                BEGIN
                IF(borretodo = '0' AND counter < long) THEN
                        IF(rising_edge(xor_h_y_v)) THEN
                                IF (polh = '1') THEN
                                        seq(counter) <= '1';
                                        counter := counter + 1;
                                ELSIF (polv = '1') THEN
```

```
                                        seq(counter) <= '0';
                                        counter := counter + 1;
                                END IF;
                        END IF;
                ELSIF(borretodo = '1') THEN
                        estoylisto <= '0';
                        counter := 0;
                        seq <= (others => '0');
                ELSIF (counter >= long) THEN
                        estoylisto <= '1';
                END IF;
        END PROCESS;

        PROCESS(xor_inicie_y_reset)
                BEGIN
                IF (reset = '1') THEN
                        borretodo <= '1';
                ELSIF (inicie = '1') THEN
                        borretodo <= '0';
                END IF;
        END PROCESS;
END registro;
```

4.3. Manejo de los registros de memoria

```
process (Addr,CLK)
  begin
    if (CLK'event and CLK='1') then
      case Addr is
        --Lectura de datos
        when X"2001" =>
           RegDataOut(0) <= Estoylistoo;
           RegDataOut(7 downto 1) <= "1100101";
        when X"2002" =>
           RegDataOut <= secuenciaa(7 downto 0);
        when X"2003" =>
           RegDataOut <= secuenciaa(15 downto 8);
        when X"2004" =>
           RegDataOut <= secuenciaa(23 downto 16);

        --Escritura de datos
        when X"3000" =>
           LEDVal (1) <= RegDataIn(1);
           LEDVal (0) <= RegDataIn(0);
           Resett <= RegDataIn(1);
           Innicie <= RegDataIn(0);
        when X"3001" =>
           LEDVal (7 downto 0) <= Addr(7 downto 0);
        when X"3002" =>
```

```
            LEDVal (7 downto 0) <= Addr(7 downto 0);
         when X"3003" =>
            LEDVal (7 downto 0) <= Addr(7 downto 0);

         --Activar por registros a polh y a polv
         when X"3008" =>
            LEDVal (7 downto 0) <= "11000011";
            Detectorh <= '1';
            Detectorv <= '0';
         when X"3009" =>
            LEDVal (7 downto 0) <=  "00011000";
            Detectorh <= '0';
            Detectorv <= '1';

         --apagar detectores y mirar como va la secuencia
         when X"3010" =>
            LEDVal (7 downto 0) <= Secuenciaa(7 downto 0);
            Detectorh <= '0';
            Detectorv <= '0';
         when X"3011" =>
            LEDVal (7 downto 0) <= Secuenciaa(15 downto 8);
            Detectorh <= '0';
            Detectorv <= '0';
         when X"3012" =>
            LEDVal (7 downto 0) <= Secuenciaa(23 downto 16);
            Detectorh <= '0';
            Detectorv <= '0';

         when others =>
            LEDVal (7 downto 0) <= LEDVal (7 downto 0);
            Detectorh  <= Detectorh;
            Detectorv  <= Detectorv;
            Resett     <= Resett;
            Innicie    <= Innicie;
            RegDataOut <= RegDataOut;
      end case;
    end if;
end process;
```

Referencias

[1] Andrew Rokhin et al. *A statistical test suit for random and pseudorandom number generators for cryptographic applications.* National Institute of Standards and Technology, first edition, 2010.

[2] IsaacL. Chuang Michael A. Nielsen. *Quantum Computation and Quantum Information.* Cambridge, first edition, 2000.

[3] Microsoft. How Visual Basic Generates Pseudo-Random Numbers for the RND Function.

[4] National Instruments. LabVIEW.

[5] Paul Diaz y Nicolas Barbosa. *Obtencion de numeros aleatorios*. Universidad de los Andes, 2012.