

8. Ficheros de entrada / salida

En cualquier sistema de proceso de datos es fundamental poder disponer de ficheros para almacenamiento de los mismos. La necesidad de estos ficheros en VHDL es evidente ya que en cualquier proceso de simulación de tamaño mediano se partirá de ficheros de datos o estímulos y se obtendrán ficheros de resultados para un análisis posterior.

En VHDL existen dos tipos de ficheros para archivo de datos: ficheros formateados y ficheros de texto. Los primeros tienen un formato que depende del sistema *host*, es decir, no pueden ser leídos o escritos por el diseñador. Primeramente deben ser escritos por el *host* en un proceso de simulación y posteriormente leídos por el sistema para su aplicación a otro proceso de simulación o conversión a datos interpretables por el usuario. Los ficheros de texto pueden ser preparados por el diseñador en un editor de texto. Su manejo es sencillo y son fácilmente transportables de un sistema a otro.

8.1 FICHEROS FORMATEADOS

Para procesar estos ficheros se necesitan dos declaraciones preliminares : una que se refiere al tipo base de los datos que componen el fichero y la otra, la propia declaración del fichero. Por ejemplo, si los datos de un fichero son bytes que representan el nivel de gris de los pixeles de una imagen, se puede declarar un tipo base :

```
TYPE imagen IS FILE OF BIT_VECTOR;
```

y a continuación se declara el fichero que contiene esos datos :

```
FILE escritura : imagen IS OUT "C:\abc\imag.pix" ;
```

donde *escritura* es el nombre lógico del fichero de salida - OUT - y *abc\imag.pix* sería el nombre físico del fichero en el sistema *host*.

El proceso de escritura de ese fichero en el sistema sería ejecutado con la sentencia de llamada al procedimiento WRITE, cuyo formato es :

```
WRITE ( escritura, varlocal);
```

donde *varlocal* se refiere a una variable local a declarar en la región al efecto del proceso o cuerpo desde donde se llama al procedimiento WRITE.

Una vez que el fichero ha sido escrito, con el formato que imponga el *host*, podrá ser leído por sentencias similares, y para ello se declara previamente el fichero de entrada como

```
FILE lectura : imagen IS IN "C:\abc\imag.pix" ;
```

que al ser de modo IN admite o implica que pueda ser referenciado en llamadas al procedimiento READ y la función ENDFILE con los formatos siguientes :

```
READ ( lectura, localvar, [longitud] );
ENDFILE ( lectura );
```

en donde cada llamada a READ lee el byte en cabeza del fichero lectura y lo asigna a la variable localvar, declarada de forma similar a como lo fue varlocal en la región donde se llamaba al procedimiento WRITE. La siguiente operación de READ leerá el siguiente byte, que ha pasado a ser la cabecera del fichero secuencial de lectura. El parámetro opcional denominado longitud solo se utiliza cuando el tipo base es un array de tipo abierto y, en tal caso, devuelve la dimensión del array.

ENDFILE es una función que devuelve el valor booleano TRUE cuando se ha leído el último elemento del fichero, y será FALSE mientras no se alcance el fin del fichero.

Un aspecto importante de los ficheros de entrada / salida en VHDL es que su tipo, como se ha visto, puede ser de modo IN o OUT, pero no de modo INOUT. Esto implica que en una misma simulación no se puede escribir y leer el mismo fichero. Ambas operaciones pueden efectuarse, pero en simulaciones diferentes.

Un ejemplo del tratamiento en VHDL de ficheros de entrada / salida formateados es el que se muestra a continuación :

Se desea grabar secuencialmente en un fichero 256 bytes, generados en un grabador provisto de señales de reloj e inhibición, - clk , inhibe - y con una salida de bytes, -pixdat- según indica la declaración de la entidad. Para ello se utilizarán subprogramas que se supone definidos en el empaquetamiento “modelos”:

```
USE WORK.modelos.ALL;
ENTITY grabador IS
    PORT( clk, inhibe : IN BIT; pixdat : OUT BIT_VECTOR(7 DOWNT0 0));
END grabador;

ARCHITECTURE formateante OF grabador IS
    TYPE imagen IS FILE OF BIT_VECTOR;
    FILE escritura : imagen IS OUT "C:\abc\imag.pix";
    SIGNAL pixbyte : BIT_VECTOR(7 DOWNT0 0);
    SIGNAL sincro : BIT;
    BEGIN
        sincro <= clk AND NOT(inhibe);
    grabacion: PROCESS(clk, inhibe)
        VARIABLE cuenta : INTEGER := 0;
        BEGIN
            IF inhibe'EVENT AND inhibe = '1' THEN
                cuenta := (2**8) -1;
            END IF;
            pixbyte <= entero_a_bin(cuenta,8) AFTER 10 ns;
            pixdat <= pixbyte AFTER 5 ns;
            IF ( clk'EVENT AND clk = '1') AND ( inhibe = '0') THEN
                cuenta := cuenta -1 ;
            END IF;
        END PROCESS grabacion;
    escribir: PROCESS(sincro)
        VARIABLE membyte : BIT_VECTOR( 7 DOWNT0 0 );
        BEGIN
            membyte := pixbyte;
            WRITE(escritura,membyte);
        END PROCESS escribir;
END formateante;
```

cuyo comportamiento se puede simular con el probador siguiente:

```
ENTITY test_graber IS
END test_graber;

ARCHITECTURE probador OF test_graber IS
  COMPONENT graber
    PORT( clk, inhibe : IN BIT; pixdat : OUT BIT_VECTOR(7 DOWNT0 0));
  END COMPONENT;
  FOR ejemplo : graber USE ENTITY WORK.grabador(formateante);
  SIGNAL clk      : BIT;
  SIGNAL inhibe   : BIT;
  SIGNAL pixdat   : BIT_VECTOR( 7 DOWNT0 0);
BEGIN
  ejemplo : graber PORT MAP (clk,inhibe,pixdat);
  reloj : PROCESS
  BEGIN
    clk <= '0', '1'    AFTER 50 ns;
                      WAIT FOR 100 ns;
  END PROCESS reloj;
  inhib : PROCESS
  BEGIN
    inhibe <= '0', '1'  AFTER 640 ns;
                      WAIT FOR 1280 ns;
  END PROCESS inhib;
END probador;
```

Otra forma de comprobar si el modelo anterior funciona correctamente es modelar un “lector” con el que se pueda ver lo que se ha grabado en el fichero C:\abc\imag.pix. Así, de forma similar al grabador, el modelo de lector podría ser :

```
ENTITY lector IS
  PORT( clk : IN BIT; pixdat : OUT BIT_VECTOR(7 DOWNT0 0));
END lector;

ARCHITECTURE leyente OF lector IS
  TYPE imagen IS FILE OF BIT_VECTOR;
  FILE lectura : imagen IS IN "C:\abc\imag.pix";
  BEGIN
  extraccion: PROCESS
    VARIABLE longi : INTEGER := 8;
    VARIABLE varlocal : BIT_VECTOR(7 DOWNT0 0);
  BEGIN
    WAIT ON clk UNTIL clk = '1';
    LOOP
      EXIT when ENDFILE(lectura);
      READ(lectura, varlocal, longi);
      pixdat <= varlocal;
      WAIT for 10 ns;
    END LOOP;
  END PROCESS extraccion;
END leyente;
```

cuyo funcionamiento puede comprobarse con el probador siguiente :

```
ENTITY test_lector IS
END test_lector;

ARCHITECTURE probador OF test_lector IS
  COMPONENT leedor
    PORT( clk : IN BIT; pixdat: OUT BIT_VECTOR(7 DOWNTO 0));
  END COMPONENT;
  FOR leer: leedor USE ENTITY WORK.lector; -- usa la ultima arquitectura
  SIGNAL clk      : BIT;
  SIGNAL pixdat    : BIT_VECTOR(7 DOWNTO 0);
BEGIN
  leer : leedor PORT MAP(clk,pixdat);
  reloj : PROCESS
  BEGIN
    clk <= '0', '1'   AFTER   50 ns;
                      WAIT FOR 100 ns;
  END PROCESS reloj;
END probador;
```

En este punto resultará interesante ver el formato y tamaño del fichero que se grabó .

8.2 FICHEROS DE TEXTO Y EMPAQUETAMIENTO TextIO

Los ficheros formateados tienen el inconveniente de estar fuera del control del diseñador. Generalmente no pueden ser creados fuera del entorno del sistema *host* y su tamaño, a veces, puede ser sorprendentemente grande por la complejidad del formato que, además, dificulta la transportabilidad de los ficheros a otros sistemas *host*.

Para evitar estos inconvenientes, se pueden utilizar ficheros con formato de texto, que en VHDL se denominan ficheros TextIO, por su relación con el empaquetamiento de ese nombre de la biblioteca STD, donde se recogen los procedimientos de lectura y escritura de este tipo de ficheros. Dado que el uso de este paquete no es necesario en muchas descripciones VHDL, cuando sea necesario hay que visualizarlo con la cláusula

```
USE STD. TEXTIO.ALL;
```

Como puede verse en el Apéndice B donde se presenta el empaquetamiento TextIO, se tienen declarados los tipos :

```
TYPE LINE IS ACCESS STRING;           -- A LINE is a pointer to a STRING value
TYPE TEXT IS FILE OF STRING;          -- a file of variable-length ASCII records
```

que son los que van a utilizarse en los procedimientos TextIO. Ambos hacen referencia a cadenas cuyos elementos tienen tipo base indefinido pero que, por la sobrecarga de tipos y subprogramas del empaquetamiento, serán los tipos que utilice el modelo del usuario los que determinen en el contexto de la descripción qué procedimientos se utilizarán. No obstante, existe una limitación a los tipos :

BOOLEAN, BIT, BIT_VECTOR, CHARACTER, INTEGER, REAL, STRING y TIME

lo que obliga a definir funciones de conversión entre tipos si el usuario utiliza tipos diferentes a esos disponibles en procedimientos TextIO.

A partir de los tipos LINE y TEXT se declaran los tipos fichero del usuario, por ejemplo, para escribir en un fichero :

FILE escritura : TEXT IS **OUT** " C:\abc\imagen.txt"

donde escritura es el nombre lógico del fichero y \abc\imagen.txt el nombre físico en el *host*.

Dado que los ficheros están organizados en líneas - LINE -, el proceso de escritura se inicia con la llamada a procedimiento WRITE para preparar una línea con los elementos que la integran y después se llama al procedimiento WRITELINE para grabar la línea en el fichero, con los formatos siguientes

```
WRITE( línea, variable_local);  
WRITELINE(escritura,línea );
```

y de forma similar, cuando se vaya a leer un fichero, se declara el fichero

FILE lectura : TEXT IS **IN** " C:\abc\imagen.txt"

y después se llama a los procedimientos de lectura de línea, READLINE, con el formato

```
READLINE ( lectura, línea )
```

y se continúa con llamadas múltiples del tipo

```
READ (línea, elemento)
```

para extraer los elementos secuencialmente dispuestos en la línea leída con READLINE. Nótese que cada línea leída con READLINE puede tener un número de elementos que dependerá del formato con que fue escrita, es decir, de los elementos que se grabaron con WRITE en una línea antes de llamar al procedimiento WRITELINE.

En el ejemplo siguiente se ha modelado un grabador textual de bytes. Con él se desea grabar una imagen sintética en un fichero. Cada byte -pixdat- representa el nivel de gris de un pixel de dicha imagen, en la escala de 0 a 255. Se desea una imagen de 256 x 256 pixels, compuesta por cuatro cuadrantes iguales, de 128 x 128 pixels cada uno. En cada cuadrante se inicia un vértice con 0 de nivel de gris, mínimo o negro, y se va elevando en cada pixel contiguo, horizontal y vertical, hasta llegar al vértice opuesto al de nivel negro, donde se tendrá un pixel de nivel blanco 254.

```
USE WORK.modelos.ALL;  
ENTITY generador IS  
    PORT( clk : IN BIT; pixdat: OUT BIT_VECTOR(7 DOWNT0 0));  
END generador;  
USE STD.TEXTIO.ALL;                                -- Clausula necesaria para ficheros TextIO
```

```
ARCHITECTURE escribtextio OF generador IS
BEGIN
grabarbytes: PROCESS
    VARIABLE numbyte: INTEGER := 0;
    VARIABLE pixbyte: BIT_VECTOR(7 DOWNTO 0);
    VARIABLE línea: LINE;
    FILE escritura : TEXT IS OUT "c:\abc\imagen.txt";
    BEGIN
        WAIT ON clk UNTIL clk = '1';
        FOR H IN 0 TO 1 LOOP                                -- índice de cuadrantes en horizontal
            FOR I IN 0 TO 1 LOOP                              -- índice de cuadrantes en vertical
                FOR J IN 0 TO 127 LOOP                        -- rango 0 a 127
                    FOR K IN 0 TO 127 LOOP                    -- rango 0 a 127
                        numbyte := K + J;                    -- rango 0 a 254
                        pixbyte := entero_a_bin(numbyte,8);  -- Medite si se podría eliminar la variable
                        WRITE(línea, pixbyte);                -- local pixbyte y operar solo con pixdat.
                        WAIT FOR 3 NS;
                        WRITELINE(escritura,línea);           -- Grabación de solo un byte por línea
                        pixdat <= pixbyte;
                    END LOOP;
                END LOOP;
            END LOOP;
        END LOOP;
    END PROCESS grabarbytes;
END escribtextio;
```

y con el modelo de probador siguiente podemos simular lo que sale por el puerto denominado pixdat, lo que será un indicio de lo que puede haberse grabado en el fichero, pero sin comprobarlo, ya que la grabación se hace con las sentencias de llamada a WRITE y WRITELINE, mientras que pixdat es otra sentencia posterior, que no indica que el formato de los procedimientos de grabación fuese el apropiado :

```
ENTITY test_bytegrab IS
END test_bytegrab;

ARCHITECTURE probador OF test_bytegrab IS
    COMPONENT grabador
        PORT( clk: IN BIT; pixdat : OUT BIT_VECTOR(7 DOWNTO 0));
    END COMPONENT;
    FOR grabante : grabador USE ENTITY WORK.generador(escribtextio);
        SIGNAL pixdat      : BIT_VECTOR(7 DOWNTO 0);
        SIGNAL clk          : BIT;
    BEGIN
        reloj : PROCESS
            BEGIN
                clk <= '0', '1' AFTER 50 ns;
                WAIT FOR 100 ns;
            END PROCESS reloj;
        grabante : grabador PORT MAP( clk , pixdat);
    END probador;
```

Como un ejemplo de lectura de ficheros TextIO, se desarrolla a continuación un modelo para leer el fichero que debería haber grabado el modelo identificado “grabador”.

```

USE WORK.modelos.ALL;
ENTITY lector IS
    PORT( clk : IN BIT; pixdat: OUT BIT_VECTOR(7 DOWNTO 0));
END lector;
USE STD.TEXTIO.ALL;                                -- Cláusula necesaria para ficheros TextIO
ARCHITECTURE leertextio OF lector IS
BEGIN
leerbytes: PROCESS
    VARIABLE brillo      : BIT_VECTOR(7 DOWNTO 0);
    VARIABLE pixbyte     : BIT_VECTOR(7 DOWNTO 0);
    VARIABLE abc,ord      : INTEGER RANGE 0 to 128;
    VARIABLE línea       : LINE;
    FILE lectura          : TEXT IS IN "c:\abclimagen.txt";
BEGIN
    abc:= 0 ;   ord := 0;
    WHILE ord < 128 LOOP
        WHILE abc < 128 LOOP
            WHILE NOT ENDFILE(lectura) LOOP
                WAIT UNTIL clk'EVENT AND clk = '1';
                READLINE(lectura,línea);    -- Como se grabó solo un byte por línea
                READ(línea,brillo);         -- se hace solo un READ por READLINE
                pixdat <= brillo;
            END LOOP;
            abc := abc + 1;                 -- Particular de los WHILE LOOPS
        END LOOP;
        ord := ord + 1;
    END LOOP;
END PROCESS leerbytes;
END leertextio;

```

cuyo movimiento de datos puede verificarse con el probador siguiente:

```

ENTITY test_bytelect IS
END test_bytelect;

ARCHITECTURE probador OF test_bytelect IS
    COMPONENT leedor
        PORT( clk: IN BIT; pixdat : OUT BIT_VECTOR(7 DOWNTO 0));
    END COMPONENT;
    FOR leyente : leedor USE ENTITY WORK.lector(leertextio);
        SIGNAL pixdat      : BIT_VECTOR(7 DOWNTO 0);
        SIGNAL clk         : BIT;
    BEGIN
        reloj : PROCESS
            BEGIN
                clk <= '0', '1' AFTER 50 ns;
                WAIT FOR 100 ns;
            END PROCESS reloj;
        leyente : leedor PORT MAP( clk, pixdat);
    END probador;

```

8.2.1 DESARROLLO DE PROBADORES CON FICHEROS TextIO

En numerosas aplicaciones de prueba de circuitos, se manejan como vectores de pruebas conjuntos de datos que engloban los estímulos y las respuestas a ellos del sistema en prueba. Este tipo de programas de prueba tiene entre otras ventajas la capacidad de determinar fácilmente los vectores que provocan fallos y, de esta forma, facilitan el proceso de diagnóstico o depuración, bien del sistema o del programa de pruebas.

En VHDL, una parte fundamental del desarrollo de modelos es la verificación del mismo, su simulación, para ver si el comportamiento del modelo corresponde al esperado. En esta tarea, a veces, la generación de vectores de prueba puede conseguirse de diversas formas, siendo una de ellas por medio de vectores de prueba en los que se incluyen, entre otros datos, los estímulos y la respuesta del sistema correcto. Para ello se tiene la posibilidad de utilizar los ficheros de texto TextIO, ya que los formateados no son generables con facilidad fuera del entorno del *host*.

En el ejemplo que sigue se expone el desarrollo de un modelo de probador para el sumador de 1 bit, al que se le proporcionan un juego de vectores de test con el formato :

duración estímulos espera respuestas

que están contenidos en el fichero “sumavec” y que son :

```
10ns 0 0 0 5ns 0 0
20ns 0 0 1 5ns 1 0
30ns 0 1 0 5ns 1 0
40ns 0 1 1 5ns 0 1
50ns 1 0 0 5ns 1 0
60ns 1 0 1 5ns 0 1
70ns 1 1 0 5ns 0 1
80ns 1 1 1 5ns 1 1
```

El objeto de este ejemplo es mostrar el uso repetido de llamadas READ para extraer los datos obtenidos con una llamada READLINE.

El modelo de probador sigue las pautas descritas en el apartado 6.4, pero se completa con llamadas a procedimientos TextIO y se hace uso de sentencias ASSERT para poner de manifiesto posibles errores al hacer la simulación.

La entidad del probador y la sección declarativa de su arquitectura son:

```
ENTITY sumatest IS
END sumatest;

USE STD.TEXTIO.ALL;
USE WORK.modelos.ALL;

ARCHITECTURE probador OF sumatest IS
    COMPONENT sumador
    PORT (x,y,ci : IN BIT; sum,co : OUT BIT);
    END COMPONENT;
    FOR xyz : sumador USE ENTITY WORK.sumador;
    SIGNAL x,y,ci,sum,co : BIT;
```

-- no hace falta declarar si está
-- en el paquete de "modelos"
-- Se incluye para referencia.

y la parte ejecutable de la arquitectura del probador puede ser como sigue :

```

BEGIN
    xyz : sumador PORT MAP( x,y,ci,sum,co ) ;
simdatos : PROCESS
    FILE testvect          : TEXT IS IN "C:\abc\sumavec" ;
    VARIABLE vector        : LINE;
    VARIABLE valor         : BIT;
    VARIABLE espacio       : CHARACTER;
    VARIABLE tiempo        : TIME;

    BEGIN
        WHILE ( NOT ENDFILE( testvect)) LOOP
            READLINE ( testvect, vector ) ;
            READ ( vector, tiempo ) ;
                WAIT FOR (tiempo - NOW ) ;           -- Ver función NOW en Apéndice A
            READ ( vector, valor );
                x <= valor;
            READ ( vector, valor );
                y <= valor;
            READ ( vector, valor );
                ci <= valor;
            READ ( vector, tiempo );
                WAIT FOR tiempo;
            READ ( vector, valor );
                ASSERT ( sum = valor )                --respuesta real = respuesta esperada?
                REPORT "suma incorrecta"
                SEVERITY WARNING;
            READ ( vector, valor );
                ASSERT ( co = valor )                  --respuesta real = respuesta esperada ?
                REPORT "carry co incorrecto"
                SEVERITY WARNING;
            WAIT for 3ns;
        END LOOP;
    END PROCESS ;
END probador;

```

en donde se pueden ver las repeticiones de READ por cada READLINE, según el formato de vectores de prueba y las declaraciones de variables de la arquitectura.

En las sentencias finales, se lee las respuestas que indican los vectores de prueba como esperadas y se hacen comprobaciones para ver si coinciden con las que proporciona el sumador como entidad en prueba. Si no hay coincidencia, las sentencias ASSERT se activarán al ser falsas las condiciones, identificando el vector o vectores de prueba en los que falla el sumador.

De forma similar a como se han manejado los procedimientos READ y READLINE en el ejemplo anterior, se puede llamar repetidamente a los procedimientos WRITE del empaquetamiento TextIO , para ir preparando una línea de texto que posteriormente se escribe en un fichero de salida por medio de una llamada al procedimiento WRITELINE.

Como ejemplo se presenta a continuación un probador del LFSR del apartado 6.5.

```
USE STD.TEXTIO.ALL;
USE WORK.all;
ENTITY test_lfsr IS
    GENERIC (N: POSITIVE := 4);
END test_lfsr;
ARCHITECTURE probadora OF test_lfsr IS
    COMPONENT lfsr
        PORT clk : IN BIT;
              qd : OUT BIT_VECTOR (0 TO (N-1));
              pr, poly : IN BIT_VECTOR (0 TO (N-1)) );
    END COMPONENT;
    FOR all:lfsr USE ENTITY WORK.lfsr(cuad);
        SIGNAL clk : BIT;
        SIGNAL qd : BIT_VECTOR (0 TO (N-1)) ;
        SIGNAL pr : BIT_VECTOR (0 TO (N-1)) ;
        SIGNAL poly : BIT_VECTOR (0 TO (N-1)) := "1000" ; -- x3, x2, x1
    BEGIN
        comp:lfsr PORT MAP (clk, qd, pr, poly);
        pr <= "1000" AFTER 3 ns, -- Inicialización de LFSR
              "0000" AFTER 6 ns;

        reloj: PROCESS
            BEGIN
                clk <= '0', '1' AFTER 10 ns;
                WAIT FOR 20 ns;
            END PROCESS reloj;

        datos: PROCESS
            FILE salida : TEXT IS OUT "texto";
            VARIABLE li : LINE;
            VARIABLE espacios : STRING (1 TO 5) := " --- ";
            VARIABLE NUM : INTEGER := 0 ;
            VARIABLE titulo : STRING (1 to 14) := " POLINOMIO = ";

        BEGIN

            IF NOW < 10 ns THEN
                WRITE (li, titulo, LEFT, 16); -- escribir STRING
                WRITE (li, poly, RIGHT, 6); -- escribir BIT_VECTOR
                WRITELINE(salida, li);
                WAIT FOR 5 ns;
            END IF;

            WRITE (li, NUM, RIGHT, 5); -- escribir INTEGER
            WRITE (li, NOW, right, 8, ns); -- escribir TIME
            WRITE (li, espacios, right, 5); -- escribir STRING
            WRITE (li, qd); -- escribir BIT_VECTOR
            WRITELINE (salida, li);
            NUM := NUM + 1;
            WAIT UNTIL qd'EVENT;
        END PROCESS datos;
    END probadora;
```

Las cuatro sentencias WRITE dentro del proceso datos, tienen el formato previsto para escribir INTEGER, TIME, STRING y BIT_VECTOR. como se puede ver en el empaquetamiento TextIO. Posteriormente, correspondiendo a la declaración :

FILE salida : TEXT IS OUT "texto";

las sentencias

WRITELINE (salida, li);

escriben en el fichero de salida - OUT- con nombre físico “texto” y nombre lógico “salida” los datos alineados por las cuatro sentencias WRITE, y que aparecen en el fichero “texto” con el contenido y formato siguiente, dependiendo de la configuración fijada para el LFSR:

POLINOMIO = 1000

1	3	NS	---	1000
2	10	NS	---	0100
3	30	NS	---	0010
4	50	NS	---	0001
5	70	NS	---	1100
6	90	NS	---	0110
7	110	NS	---	0011
8	130	NS	---	1101
9	150	NS	---	1010
10	170	NS	---	0101
11	190	NS	---	1110
12	210	NS	---	0111
13	230	NS	---	1111
14	250	NS	---	1011
15	270	NS	---	1001
16	290	NS	---	1000
17	310	NS	---	0100
18	330	NS	---	0010
19	350	NS	---	0001
20	370	NS	---	1100
21	390	NS	---	0110

.....

POLINOMIO = 0010

1	3	NS	---	1000
2	10	NS	---	0100
3	30	NS	---	0010
4	50	NS	---	0001
5	70	NS	---	1001
6	90	NS	---	1101
7	110	NS	---	1111
8	130	NS	---	1110
9	150	NS	---	0111
10	170	NS	---	1010
11	190	NS	---	0101
12	210	NS	---	1011
13	230	NS	---	1100
14	250	NS	---	0110
15	270	NS	---	0011
16	290	NS	---	1000
17	310	NS	---	0100
18	330	NS	---	0010
19	350	NS	---	0001
20	370	NS	---	1001
21	390	NS	---	1101

.....

que corresponden a un LFSR exhaustivo -salvo “0000”- configurado con polinomios característicos $x^4 + x^3 + 1$ y $x^4 + x + 1$ respectivamente por las sentencias :

poly	<= "1000" ;	-- x3,x2,x1	mostrada en página anterior
poly	<= "0010" ;	-- x3,x2,x1	

y que, por tanto, para esas dos configuraciones y no para otras, contiene 15 estados diferentes que se repiten cíclicamente como muestra la secuencia obtenida de las salidas qd del LFSR, pudiendo compararse la pseudoaleatoriedad y diferencias en ambas secuencias.