

6. Modelos estructurales

Consisten en una descripción de los componentes que forman el sistema y de las conexiones entre ellos, reflejando de manera precisa la estructura del circuito que describen. Así, las sentencias básicas de este tipo de descripciones son las sentencias de colocación de componentes, utilizadas para incluir o poner un componente en una arquitectura, las declaraciones de componentes, bien locales o en un empaquetamiento al que se accede y las sentencias de especificación de configuración, identificándolos con *entidades de diseño*.

6.1 ESPECIFICACIONES DE CONFIGURACIÓN

Consisten en asociar componentes con entidades de diseño previamente compiladas.

La especificación permite asociar diferentes modelos a un componente según la funcionalidad que deba tener en sus diferentes colocaciones en el circuito, así como seleccionar para cada uno de esos modelos o *entidades de diseño* la arquitectura más idónea de aquellas que tenga. Las sentencias de especificación se colocan en las áreas declarativas de las arquitecturas con el formato siguiente :

FOR etiqueta : componente USE ENTITY biblioteca . entidad [(arquitectura)];

Los significados o referencias de cada uno de los campos anteriores son como sigue:

etiqueta	:	Se refiere a la usada en la sentencia de colocación del componente. Puede usarse ALL si todas las colocaciones utilizan el mismo modelo.
componente	:	Nombre del componente en su declaración.
biblioteca	:	Aquella donde está la entidad de diseño. Por defecto es WORK.
entidad	:	Nombre de la entidad asociada al componente.
arquitectura	:	Optativa. Una de las posibles de la entidad, si se desea especificar. Por defecto se configura la última arquitectura compilada.

En el ejemplo siguiente, se tiene una descripción estructural de un comparador de bits utilizando puertas cuyos modelos se adjuntan para referencia. La declaración de entidad del comparador solo describe su interfaz, mientras que la arquitectura detalla la estructura del diseño concreto basado en los componentes de las figuras adjuntas:

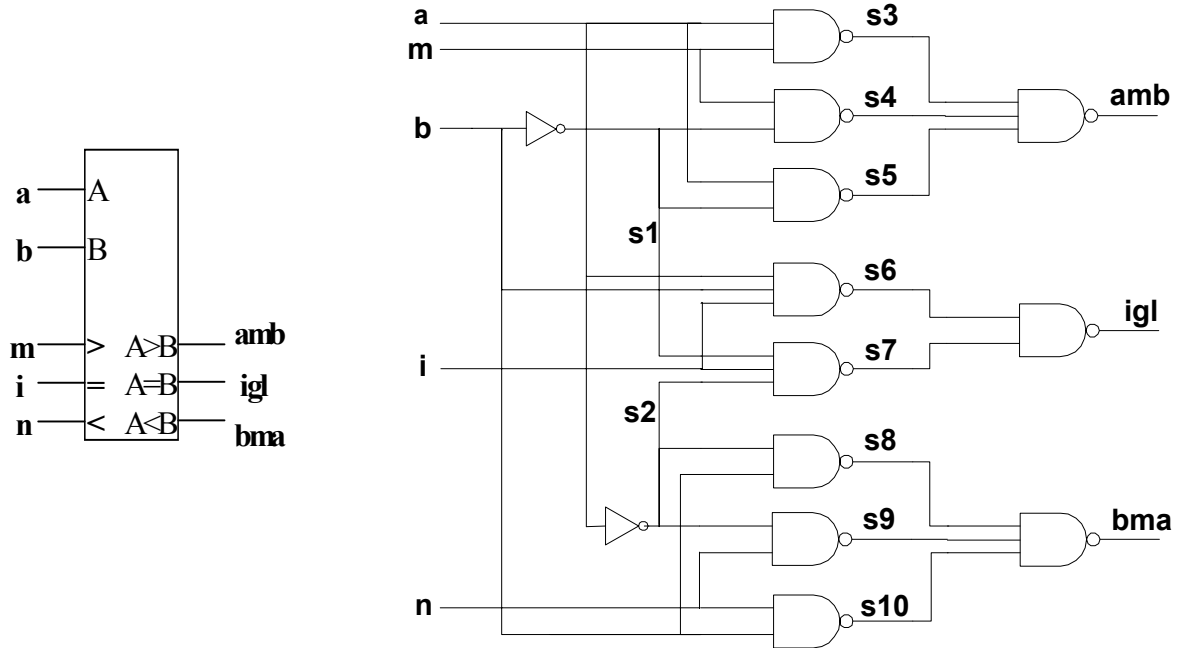
```
ENTITY inv IS
  PORT ( e1: IN BIT; sal: OUT BIT);
END INV;
ARCHITECTURE normal OF inv IS
BEGIN
  sal <= NOT e1 AFTER 10 NS;
END normal;
```

```
ENTITY nand2 IS
  PORT ( e1,e2 : IN BIT; sal: OUT BIT);
END NAND2;
ARCHITECTURE lenta OF nand2 IS
BEGIN
  sal <= e1 NAND e2 AFTER 20 NS;
END lenta;
```

```

ENTITY nand3 IS
    PORT ( e1,e2,e3 : IN BIT; sal : OUT BIT);
END nand3;
ARCHITECTURE veloz OF nand3 IS
BEGIN
    sal <= NOT ( e1 and e2 and e3 ) AFTER 5 NS;
END veloz ;

```



```

USE WORK.modelos.ALL;
ENTITY compbit IS
    PORT( a,b,m,i,n : IN BIT; amb, igl, bma:OUT BIT);
END compbit;
ARCHITECTURE puertas OF compbit IS
    FOR ALL : inver      USE ENTITY WORK. inv (normal);      -- Especificaciones
    FOR ALL : nand2      USE ENTITY WORK. nand2 (lenta);     -- Internas
    FOR ALL : nand3      USE ENTITY WORK. nand3 (veloz);     -- de Configuración
    SIGNAL s1,s2,s3,s4,s5,s6,s7,s8,s9,s10 : BIT;
BEGIN
    p0 : inver          PORT MAP (b,s1);      -- Notar que los identificadores de los
    p1 : inver          PORT MAP (a,s2);      -- componentes y las entidades no es
    p2 : nand2          PORT MAP (a,m,s3);    -- necesario que sean iguales .
    p3 : nand2          PORT MAP (m,s1,s4);
    p4 : nand2          PORT MAP (a,s1,s5);
    p5 : nand3          PORT MAP (s3,s4,s5,amb);
    p6 : nand3          PORT MAP (a,b,i,s6);
    p7 : nand3          PORT MAP (s1,i,s2,s7);
    p8 : nand2          PORT MAP (s6,s7,igl);  -- Los identificadores de componentes
    p9 : nand2          PORT MAP (s2,b,s8);    -- son aquellos con los que figuran en
    p10 : nand2         PORT MAP (s2,n,s9);    -- el empaquetamiento donde estan.
    p11 : nand2         PORT MAP (b,n,s10);
    p12 : nand3         PORT MAP (s8,s9,s10,bma);
END puertas;

```

6.2 DECLARACIÓN DE CONFIGURACIÓN

Al ser las especificaciones de configuración del ejemplo del comparador internas a la arquitectura, cada cambio de esas especificaciones supone recompilar la arquitectura, por lo que ese estilo de configuración solo se emplea en diseños de pequeño tamaño, siendo preferible en diseños grandes mantener una arquitectura genérica que solo se compila una vez y configurar *externamente* aquellas combinaciones de interés, que serán las *unidades de diseño* que se compilen separadamente.

La CONFIGURATION es una unidad de diseño *primaria*, pero deberá ser analizada y compilada después de la arquitectura a que configura.

Aunque existen otros estilos, la forma más conveniente de configuración externa consiste en una estructura que permite anidamientos de configuraciones hasta el nivel jerárquico más bajo que se desee, con el formato siguiente entre dos niveles sucesivos:

```
CONFIGURATION ejemplo_de_config OF entidad_configurada IS
  FOR arquitectura_configurada
    .....
    FOR etiqueta : componente USE ENTITY WORK. entidad [(arquitectura)];
    END FOR;
    .....
  END FOR;
END ejemplo_de_config;
```

donde las líneas FOR, iguales a las especificaciones de configuración internas y las END FOR

se repiten con el mismo criterio que las especificaciones de configuración internas.

Suponiendo que en el ejemplo anterior de comparador existieran para las tres puertas arquitecturas "lenta", "normal", y "veloz", se podría modelar con una única arquitectura:

```
USE WORK.modelos.ALL;
ARCHITECTURE config_exter OF compbit IS
  -- No existen Especificaciones internas de Configuración
  SIGNAL s1,s2,s3,s4,s5,s6,s7,s8,s9,s10 : BIT;
BEGIN
  p0 : inver      PORT MAP (b,s1);
  p1 : inver      PORT MAP (a,s2);
  p2 : nand2      PORT MAP (a,m,s3);
  p3 : nand2      PORT MAP (m,s1,s4);
  p4 : nand2      PORT MAP (a,s1,s5);
  p5 : nand3      PORT MAP (s3,s4,s5,amb);
  p6 : nand3      PORT MAP (a,b,i,s6);
  p7 : nand3      PORT MAP (s1,i,s2,s7);
  p8 : nand2      PORT MAP (s6,s7,igl);
  p9 : nand2      PORT MAP (s2,b,s8);
  p10 : nand2     PORT MAP (s2,n,s9);
  p11 : nand2     PORT MAP (b,n,s10);
  p12 : nand3     PORT MAP (s8,s9,s10,bma);
END puertas;
```

y para esta arquitectura se podrán definir tantas declaraciones de configuración como permitan los componentes empleados y las arquitecturas disponibles de éstos, por ejemplo :

```
CONFIGURATION externa OF compbit IS
  FOR config_exter
    FOR ALL      : inver      USE ENTITY WORK. inv (normal);
  END FOR;
  FOR P2        : nand2      USE ENTITY WORK. nand2 (lenta);
  END FOR;
  FOR OTHERS    : nand2      USE ENTITY WORK. nand2 (veloz);
  END FOR;
  FOR P5,P6     : nand3      USE ENTITY WORK. nand3 (lenta);
  END FOR;
  FOR P7        : nand3      USE ENTITY WORK. nand3 (normal);
  END FOR;
  FOR P12       : nand3      USE ENTITY WORK. nand3 (veloz);
  END FOR;
  END FOR;
END externa;
```

donde se tienen diferentes formatos de describir configuración del par entidad-arquitectura.

Cuando se utiliza un par entidad-arquitectura en que los puertos han cambiado sus identificadores y no se cambia la declaración de un componente que los asocia, quizá porque éste está declarado en un empaquetamiento que probablemente no se desea alterar para no afectar a otros diseños que lo utilicen, y dado que el mapeo se hace referenciando la etiqueta del componente al que se asocia la nueva entidad, será necesario hacer una asociación de puertos interna en la configuración, por ejemplo, si la situación fuese :

ENTITY nand2 IS	COMPONENT nand2
PORT (a,b : IN BIT; f : OUT BIT);	PORT (x,y : IN BIT; z: OUT BIT);
END nand2;	END COMPONENT;

en el ejemplo anterior la colocación P2 usando la nueva entidad nand2, se modificaría a :

```
FOR P2 : nand2 USE ENTITY WORK. nand2 (nueva);
  PORT MAP ( x => a, y => b, z => f );      ----- componente => entidad
END FOR;
```

La declaración previa del componente no es un modelo de dispositivo, sino simplemente una referencia a su interfaz, muy similar a la de ENTITY. Sin embargo, para que el modelo sea simulable se requiere asociarle una entidad de diseño formada por una pareja Entidad / Arquitectura que encaje con la interfaz con que ha sido declarado el componente en su colocación.

El mapeo recuerda a la colocación de un dispositivo en un zócalo puesto en una posición etiquetada de un circuito impreso. En el zócalo -componente- se pueden colocar distintos dispositivos -entidades- y los terminales del zócalo-componente aplican sus señales al dispositivo-entidad que se inserte en el zócalo. Así, el zócalo es el que tiene una

etiqueta diferente para cada referencia o posición, un nombre común a todas las referencias del mismo componente-zocalo en el circuito impreso y una orientación de sus terminales a la que debe adaptarse la entidad-arquitectura que se inserte en él, con una asociación posicional, que no es necesaria si se hace nominalmente, terminal a terminal.

6.3 CONFIGURACIÓN CON GENÉRICOS

Como se ha visto, los genéricos son parámetros constantes cuyo valor determina ciertos comportamientos o aspectos de los modelos. Dado que los posibles valores por defecto son sustituidos por los que se asignen en una configuración o colocación particular, si se considera el modelo de puerta AND2 y el componente AND2 con genéricos :

ENTITY and2 IS	COMPONENT and2
GENERIC(tphl, tplh:TIME; fanout:POSITIVE);	GENERIC(tphl,tplh:TIME;fanout:POSITIVE);
PORT (e1,e2 : IN BIT; sal: OUT BIT);	PORT (x,y : IN BIT; z: OUT BIT);
END and2;	END COMPONENT;

colocados en la estructura del multiplicador visto en 5.1 podrían modelarse como

```

ARCHITECTURE congeneric OF multunbit IS                                -- Cláusula USE...se supone
  SIGNAL p1, s1, s2, s3 : BIT;
BEGIN
  C1: and2      GENERIC MAP (15 ns, 18 ns, 5)    -- No hay ni “,” ni “;”
               PORT MAP (x,y, p1);
  C2: semisum   PORT MAP (p1,z,s1,s2);
  C3: or2       PORT MAP (s1,s3,co);
  C4: semisum   PORT MAP (s2,w,s3,pr);
END congeneric;

CONFIGURATION externa OF multunbit IS
  FOR congeneric
    FOR all:semisum      USE ENTITY WORK.semisum;
    FOR all:and2         USE ENTITY WORK.and2;
    FOR all:or2          USE ENTITY WORK.or2;
  END FOR;
END externa;
```

pero que también podrían modelarse configurando los genéricos

```

ARCHITECTURE singeneric_2 OF multunbit IS                             -- Cláusula USE...se supone
  SIGNAL p1, s1, s2, s3 : BIT;
BEGIN
  C1: and2      PORT MAP (x,y, p1);
  C2: semisum   PORT MAP (p1,z,s1,s2);
  C3: or2       PORT MAP (s1,s3,co);
  C4: semisum   PORT MAP (s2,w,s3,pr);
END singeneric;
CONFIGURATION externa OF multunbit IS
  FOR singeneric
```

```
FOR all:semisum USE ENTITY WORK.semisum;
FOR all:and2    USE ENTITY WORK.and2
                GENERIC MAP (15 ns, 18 ns, 5);
FOR all:or2     USE ENTITY WORK.or2;
END FOR;
END externa;
```

y si se desea considerar el nivel jerárquico inferior al semisumador

```
ENTITY semisum IS
PORT ( a,b: IN BIT; su, ca: OUT BIT);
END semisum;

ARCHITECTURE simple OF semisum IS
BEGIN
    su <= a XOR b AFTER 20 ns;
    ca <= a AND b AFTER 30 ns;
END simple;
COMPONENT semisum
PORT ( a,b: IN BIT; su, ca: OUT BIT);
END COMPONENT;
```

A partir de este componente se describe otra nueva arquitectura para el multiplicador como

```
ARCHITECTURE semisumadora OF multunbit IS
    SIGNAL p1, s1, s2, s3 : BIT;
BEGIN
    C1: and2      PORT MAP (x,y, p1);
    C2: semisum   PORT MAP (p1,z,s1,s2);
    C3: or2       PORT MAP (s1,s3,co);
    C4: semisum   PORT MAP (s2,w,s3,pr);
END semisumadora;
```

cuya configuración de dos niveles jerárquicos podría ser :

```
CONFIGURATION dosniveles OF multunbit IS
    FOR semisumadora
        FOR ALL semisum
            FOR ALL : xr2    USE ENTITY WORK. xr2;
            END FOR;
            FOR ALL : and2   USE ENTITY WORK. and2
                            GENERIC MAP (15 ns, 18 ns, 5);
            END FOR;
        END FOR;
        FOR all:and2        USE ENTITY WORK.and2
                            GENERIC MAP (15 ns, 18 ns, 5);
        END FOR;
        FOR all:or2         USE ENTITY WORK.or2;
        END FOR;
    END FOR;
END dosniveles;
```

Esta estructura sería la que se aplicaría en caso de tener una estructura con BLOCKs.

Como se puede ver, aunque sea posible, desde un punto de visto de desarrollo de configuraciones de modelos, es conveniente tener simulados y configurados “componentes” de dos o tres niveles como máximo, que serían utilizados para configurar a niveles jerárquicos más altos como componentes de un nivel. Así los anidamientos tendrán menos niveles, la estructura general será más modular y, probablemente, optimizada entre niveles.

6.4 DESARROLLO DE PROBADORES

Una vez desarrollado el modelo VHDL del circuito, es necesario probarlo o simularlo para ver si su funcionamiento es el esperado. El procedimiento que se sigue es desarrollar una entidad de nivel superior al modelo bajo prueba, donde éste será englobado como un componente. El conjunto formado por la unidad bajo prueba y los elementos o descripciones adicionales para probarla constituyen lo que se denomina en la literatura americana como *test_bench*, que podríamos llamar *banco de pruebas* o simplemente *probador*.

El probador es una entidad que genera internamente los estímulos que necesita para probar el modelo y donde se generan como señales, también internas, las respuestas del modelos a los estímulos. Por tanto, el conjunto probador es una entidad sin señales de interfaz, que se declara como :

```
ENTITY nombre_de_probador IS  
END nombre_de_probador ;
```

y para la cual se describe la arquitectura necesaria para generar los estímulos apropiados al modelo bajo prueba.

Las pautas básicas para desarrollar arquitecturas de probadores son las siguientes:

1. Si van a utilizarse entidades declaradas en algún empaquetamiento, hacer uso de cláusulas USE en la forma conveniente.
2. El modelo a probar se asocia a un componente que se declara localmente y cuya interfaz será igual que la del modelo a probar. La denominación de las señales de interfaz del componente no es necesario que sea igual a las de la entidad a que se va a asociar, pero suelen asignarse nombres iguales para facilitar la descripción.
3. Hacer una especificación de configuración para el componente, referenciando la entidad y arquitectura que se desean probar.
4. Declarar todas las señales internas, es decir, las que van a constituir los estímulos o entradas al componente, las que vayan a ser entradas fijas o conectadas a un nivel fijo, como masa o alimentación y aquellas que sean las salidas del componente.

5. Describir el área ejecutable de la arquitectura, comenzando por una colocación del componente, mapeando las señales de estímulos declaradas a las entradas del componente declarado.
6. Generar la forma de onda o vectores de test necesarios, por medio de sentencias de asignación de señal, procesos o algún procedimiento apropiado al modelo.
7. Realizar una simulación del probador para examinar las señales relevantes en relación a los estímulos aplicados al modelo.

Veamos un ejemplo de probador aplicado al modelo del decodificador siguiente :

```
ENTITY decoder_3_a_8 IS
    PORT ( adr : IN BIT_VECTOR ( 2 DOWNTO 0 );
          sal : OUT BIT_VECTOR ( 7 DOWNTO 0 ) );
END decoder_3_a_8;

ARCHITECTURE directa OF decoder_3_a_8 IS
BEGIN
    WITH adr SELECT
        sal <= "00000001" AFTER 15 NS WHEN "000",
              "00000010" AFTER 15 NS WHEN "001",
              "00000100" AFTER 15 NS WHEN "010",
              "00001000" AFTER 15 NS WHEN "011",
              "00010000" AFTER 15 NS WHEN "100",
              "00100000" AFTER 15 NS WHEN "101",
              "01000000" AFTER 15 NS WHEN "110",
              "10000000" AFTER 15 NS WHEN "111";
END directa;
```

El banco de pruebas o probador podría describirse como:

```
ENTITY prob_decoder IS
END prob_decoder;
```

y una posible arquitectura de la misma puede ser :

```
ARCHITECTURE probadora OF prob_decoder IS

    COMPONENT decodificador          -- nombre a repetir al especificar configuracion
    PORT ( adr : IN BIT_VECTOR ( 2 DOWNTO 0 );
          sal : OUT BIT_VECTOR ( 7 DOWNTO 0 ) );
    END COMPONENT;

    FOR comp1 : decodificador USE ENTITY WORK. decoder_3_a_8 (directa);
        SIGNAL adr : BIT_VECTOR ( 2 DOWNTO 0 );
        SIGNAL sal : BIT_VECTOR ( 7 DOWNTO 0 );
    BEGIN
        comp1 : decodificador PORT MAP ( adr, sal );
    END;
```



```

dir_2 : PROCESS
BEGIN
    adr(2) <= '0', '1' AFTER 40 ns;
    WAIT FOR 80 ns;
END PROCESS dir_2;

dir_1 : PROCESS
BEGIN
    adr(1) <= '0', '1' AFTER 20 ns;
    WAIT FOR 40 ns;
END PROCESS dir_1;

dir_0 : PROCESS
BEGIN
    adr(0) <= '0', '1' AFTER 10 ns;
    WAIT FOR 20 ns;
END PROCESS dir_0;
END probadora;

```

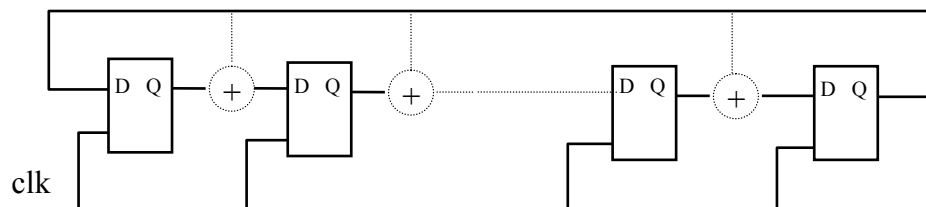
Para mejor seguimiento se ha mantenido la descripción completa, incluyendo los componentes. Tal como se indicaba en el punto 1º de las pautas a seguir para desarrollo de los probadores, podría haberse eliminado la declaración local del componente a probar si éste se hubiese declarado en el paquete de “modelos” y visualizado con USE.

6.5 GENERACIÓN DE VECTORES DE TEST PSEUDOALEATORIOS

En el ejemplo del último apartado se han generado tres señales periódicas por medio de procesos muy simples. Sin embargo, se puede decir que la simulación de modelos puede requerir formas de onda o vectores de test específicos y adecuados a las características funcionales del modelo a probar.

Un tipo de circuitos de amplio uso en sistemas de autoverificación son los que se basan en LFSRs - *Linear Feedback Shift Registers* - que, por su fácil reconfiguración, permiten utilizarlos como generadores pseudoaleatorios, o como compresores- analizadores de firma, con entrada serie o con entradas paralelas.

Sin entrar en el análisis de este tipo de circuitos, en la figura adjunta se muestra su estructura como un ejemplo interesante a modelar con una arquitectura estructural que, en este caso, será más reducida que la que resultaría con una arquitectura de comportamiento, por la funcionalidad pseudoaleatoria de estos circuitos y la configurabilidad que se pretende dar al modelo de LFSR que se utiliza como generador reconfigurable.



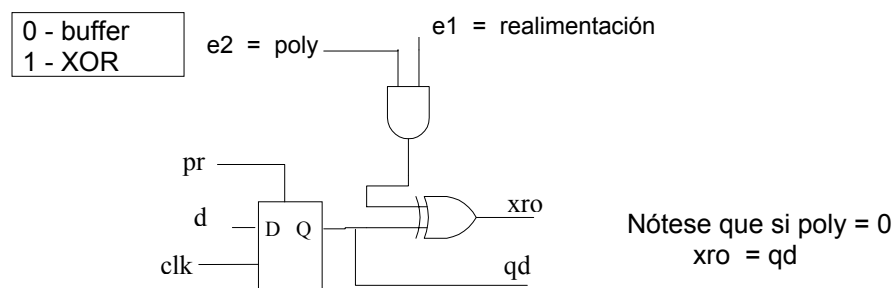
El circuito de la figura se conoce como “*LFSR con XOR internas*” y está compuesto físicamente por DFFs y puertas XOR en serie entre las salidas Q de los DFFs y las entradas de los siguientes. Nótese que las puertas XOR son opcionales.

Es interesante mencionar de estos circuitos que :

- No permiten la combinación o estado “0000..00”, conocido como *estado de bloqueo*, ya que si se produjera, el circuito no puede evolucionar o salir de él. Existen varias alternativas para que ese estado sea posible, añadiendo dos o tres puertas a la estructura genérica de la figura. En tal caso se les llama *generadores pseudoaleatorios exhaustivos*, porque la secuencia de estados, aparte de completa, reúne características de aleatoriedad.
- Dependiendo de las puertas XOR insertadas, la transición entre estados y el número de estos será diferente. Existen algunas configuraciones que permiten obtener los $2^n - 1$ estados posibles, exceptuado el “000...00”, siendo n el número de DFFs. Se dispone de una teoría matemática por la que se asocian *polinomios característicos* de grado n a cada configuración o estructura de puertas XOR. Dicha teoría se utiliza como alternativa para determinar el estado final o firma obtenida cuando los LFSRs se usan como analizadores.

Para modelar el circuito genérico del LFSR con XOR internas, partimos de la función de los componentes que se utilizan - XORs y DFFs - y definimos un componente especial en el que se engloba la funcionalidad del conjunto y, además, se incluye una puerta AND de dos entradas con una de las cuales a ‘1’ se controlará la existencia de la conexión de realimentación a la puerta XOR, o con ‘0’ se transforma la función XOR en la de un buffer.

El componente especial de la figura se modela con la descripción de comportamiento



La entidad que se define para el componente especial es :

```
ENTITY xrf IS
    PORT(d,clk,pr,e1,e2: IN BIT; qd,xro : OUT BIT);
END xrf;
```

Los DFFs son inicializados a ‘1’ por activación de la entrada PRESET ($pr = '1'$). La entrada **e2** de la puerta AND se usa para fijar el polinomio característico del LFSR, mientras que la entrada **e1** está permanentemente conectada al lazo de realimentación.

La arquitectura de comportamiento de este componente específico puede ser :

```

ARCHITECTURE algorit OF xrf IS
    SIGNAL inter : BIT;      -- señal interna para evitar lectura de señal qd
                                -- y el uso de modos INOUT o BUFFER.
BEGIN
    PROCESS (d,clk,pr)
    BEGIN
        IF ( pr = '1' )      THEN      inter <= '1';
        ELSIF (clk'EVENT AND clk = '1' ) THEN      inter <= d;
        END IF;
    END PROCESS;
                                xro <= inter XOR ( e1 AND e2);
                                qd <= inter;
END algorit;

```

A partir del componente XRF se puede modelar un LFSR de cuatro XRFs como :

```

ENTITY lfsr IS
    PORT(    clk      :    IN BIT;
            pr, poly   :    IN BIT_VECTOR (0 TO 3) ;
            qd         :    OUT BIT_VECTOR (0 TO 3) ) ;
END lfsr;

```

La señal poly define los coeficientes del polinomio característico citado anteriormente, en el que los coeficientes de x^n y x^0 son siempre '1', indicando que se realimenta siempre desde la última salida ($1.x^0$) a la primera entrada ($1.x^n$).

```

ARCHITECTURE cuad OF lfsr IS
    COMPONENT xrf
        PORT(d,clk,pr,e1,e2: IN BIT; q,xro : OUT BIT);
    END COMPONENT;
    FOR ALL : xrf USE ENTITY WORK.xrf;
    SIGNAL xr0,xr1,xr2,xr3: BIT;
BEGIN
    ff0 : xrf PORT MAP (xr3, clk, pr(0), xr3, poly(0), qd(0), xr0); -- poly(3) será = '0' siempre
    ff1 : xrf PORT MAP (xr0, clk, pr(1), xr3, poly(1), qd(1), xr1); -- para hacer xr3 = qd(3) .
    ff2 : xrf PORT MAP (xr1, clk, pr(2), xr3, poly(2), qd(2), xr2); -- y evitar la lectura de qd(3)
    ff3 : xrf PORT MAP (xr2, clk, pr(3), xr3, poly(3), qd(3), xr3);
END cuad;

```

que puede generalizarse para LFSRs con N XRFs usando GENERIC y sentencias GENERATE por medio de la descripción siguiente :

```

ENTITY lfsr IS
    GENERIC ( N : POSITIVE := 4 );      -- particularización al caso anterior
    PORT(    clk      :    IN BIT;
            pr, poly   :    IN BIT_VECTOR (0 TO (N-1)) ;
            qd         :    OUT BIT_VECTOR (0 TO (N-1)) ) ;
END lfsr;

```

para la cual se puede definir la arquitectura siguiente :

```
ARCHITECTURE cuad OF lfsr IS
  COMPONENT xrf
    PORT(d,ck,pr,e1,e2: IN BIT;  qd,xro : OUT BIT);
  END COMPONENT;
  FOR ALL : xrf USE ENTITY WORK.xrf;          -- Especificación de configuración
  SIGNAL xr : BIT_VECTOR(0 TO (N-1));
BEGIN
  ff0 : xrf PORT MAP(xr(N-1), clk, pr(0), xr(N-1), poly(0), qd(0), xr(0));
  ffi : FOR i IN 1 TO (N-2) GENERATE
    xrf PORT MAP( xr(i-1), clk, pr(i), xr(N-1), poly(i), qd(i), xr(i));
    END GENERATE;
  ffN : xrf PORT MAP(xr(N-2), clk, pr(N-1), xr(N-1), poly(N-1), qd(N-1), xr(N-1));
END cuad;
```

La arquitectura contiene especificaciones de configuración que normalmente serán aceptadas por casi todas las herramientas VHDL. Sin embargo, como una excepción que confirma la regla de las particularidades de cada herramienta, en el entorno del Simulador de Sistemas de Synopsys - VSS - se requiere configurar los componentes internos a sentencias GENERATE de una forma especial que, por supuesto, debe ser una expresión sintácticamente correcta en VHDL . El formato requerido se basa en el uso de sentencias BLOCK internas al bucle FOR de la sentencia GENERATE como se indica a continuación :

```
ARCHITECTURE cuad OF lfsr IS
  COMPONENT xrf
    PORT(d,ck,pr,e1,e2: IN BIT;  qd,xro : OUT BIT);
  END COMPONENT;
  FOR ALL : xrf USE ENTITY WORK.xrf(algorit);
  SIGNAL xr: BIT_VECTOR(0 TO (N-1));
BEGIN
  ff0: xrf PORT MAP(xr(N-1), clk, pr(0), xr(N-1), poly(0), qd(0), xr(0));

  ffi: FOR i IN 1 TO (N-2) GENERATE
    especial: BLOCK
      FOR all: xrf USE ENTITY WORK.xrf(algorit);
      BEGIN
        nx: xrf PORT MAP(xr(i-1), clk, pr(i), xr(N-1), poly(i), qd(i), xr(i));
      END block especial;
    END GENERATE;

  ffN: xrf PORT MAP(xr(N-2), clk, pr(N-1), xr(N-1), poly(N-1), qd(N-1), xr(N-1));
END cuad;
```

Para estos modelos de LFSRs podrán describirse probadores de acuerdo con lo visto en el apartado anterior, y que para el caso de un LFSR de cuarto orden, como el modelado en los ejemplos anteriores, podría describirse como :

```
ENTITY test_lfsr IS
END test_lfsr;
```

para la que puede describirse una arquitectura como sigue:

```

ARCHITECTURE probadora OF test_lfsr IS

    COMPONENT lfsr
    PORT(  clk    : IN BIT;
          qd     : OUT BIT_VECTOR (0 TO 3);
          pr     : IN BIT_VECTOR (0 TO 3);
          poly   : IN BIT_VECTOR (0 TO 3));
    END COMPONENT;

    FOR all:lfsr USE ENTITY WORK.lfsr(cuad);

        SIGNAL    clk    : BIT;
        SIGNAL    qd     : BIT_VECTOR (0 TO 3) ;
        SIGNAL    pr     : BIT_VECTOR (0 TO 3) ;
        SIGNAL    poly   : BIT_VECTOR (0 TO 3) := "1000" ;      -- x3, x2, x1
                                                                --Configuraciónde LFSR

    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;

END probadora;

```

En la descripción anterior, podría haberse declarado el componente XRF en el paquete de modelos y haber utilizado la cláusula `USE WORK.modelos.ALL;` para visualización de biblioteca y componentes del paquete.

Nótese que por medio del array de `PRESET` se inicializa la cadena de DFFs con una determinada combinación y poco después se desactiva `PRESET` para que el sistema pueda evolucionar libremente con las señales de reloj que conmutan a los DFFs. Esta descripción ha de ser acorde con el modelo de los DFFs descrito por la entidad XRF.

En cuanto a la señal `poly` utilizada para fijar el polinomio característico del LFSR, es decir, las XOR que actúan como tales o como *buffers* y, por tanto, se supone que está prefijada y que no cambiará durante el funcionamiento del LFSR, ya que no tendría sentido desde un punto de vista funcional. Por tal razón, así se ha previsto en la lista de sensibilidad del proceso que describe los DFFs.