Apéndice 5

Uso de verilog

A5.1 Introducción

Verilog es uno de los lenguajes de descripción de hardware (HDL) más utilizados.

Las herramientas CAD soportan el ingreso de diseños digitales en base a esquemáticos, los cuales se arman conexionando componentes básicas o de bibliotecas en un ambiente visual.
Sin embargo a medida que el sistema aumenta su tamaño, cada vez es más difícil la descripción por esquemáticos. Los esquemáticos describen la conectividad y por lo tanto el armado del sistema, pero no destacan o enfatizan la funcionalidad de éste.

Los lenguajes descriptivos de hardware permiten diseñar, en forma abstracta, complejos sistemas digitales que luego de ser simulados podrán ser implementados en dispositivos programables como FPGA o CPLD, lo cual se logra describiendo el sistema digital mediante código HDL.

Las herramientas computacionales de ayuda al diseño (CAD) permiten diseñar, verificar códigos de alto nivel, simular el circuito representado y crear los códigos binarios que serán cargados en los dispositivos programables (síntesis).

Debido a que los módulos pueden reutilizarse, y los subsistemas más frecuentemente utilizados pueden estar prediseñados, la programación a través de lenguajes toma cada vez más importancia en el diseño de sistemas digitales.

Los lenguajes HDL deben permitir la **simulación**, cuyos objetivos son: verificar que el diseño es funcionalmente correcto; es decir que se cumplan las **especificaciones lógicas**; y además que se cumplan las **especificaciones temporales**; es decir que los tiempos de propagación a través de las componentes y de los caminos que permiten la interconexión cumplan las especificaciones de setup y hold de los registros, en caso de sistemas secuenciales sincrónicos. Como deben considerarse todos lo casos, es difícil asegurar que la simulación entregará la solución definitiva, ya que el conjunto de estímulos, para los cuales se realiza la simulación, puede no ser completo. Al mismo tiempo los lenguajes deben realizar la **síntesis lógica**; es decir traducir el programa a una serie de compuertas y flip-flops; también deben efectuar minimizaciones, y permitir **mapear a los bloques tecnológicos disponibles**; para finalmente **colocar y enrutar** de forma conveniente los bloques en el dispositivo programable.

Verilog existe desde el año 1984, luego de varios años de uso se ha estandarizado y su última versión es del año 2001. Es un lenguaje similar a C, y dentro de los HDL, además de las

descripciones abstractas permite representaciones en bajo nivel; es decir puede describir sistemas digitales en base a compuertas, e incluso en base a transistores.

Verilog permite que en un diseño se puedan usar diferentes niveles de descripción de sistemas digitales en un mismo ambiente; las diferentes descripciones pueden ser **simuladas** para verificar el funcionamiento y además pueden ser **sintetizadas**; es decir traducidas a la interconexión de componentes básicas de un dispositivo programable.

Verilog permite la descripción **estructural** del diseño en base a componentes básicas, y descripciones más abstractas que se enfocan en la **conducta** del sistema. La conducta puede describirse mediante expresiones lógicas y también empleando procedimientos.

Un diseño basado en descripciones funcionales o de comportamiento puede resultar lento y de gran costo en área. Las descripciones en niveles estructurales permiten optimizar los circuitos lógicos para maximizar la velocidad y minimizar el área.

Se expondrán las formas de descripción para sistemas combinacionales y secuenciales.

La Figura A5.1 resume las principales etapas del diseño digital.

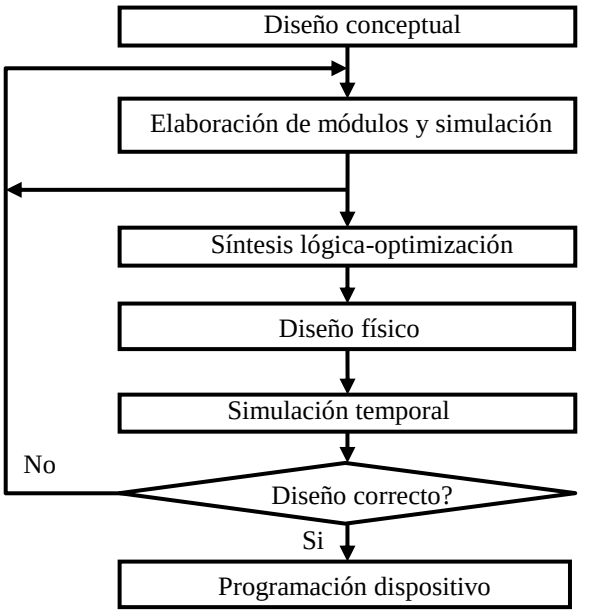

Figura A5.1

A5.2 Descripción estructural.

A5.2.1. Nivel compuertas.

Permite representar una red lógica mediante sus ecuaciones. Para esto se emplean **funciones lógicas** básicas para representar compuertas.

Los operadores and, or not, en minúsculas, son palabras reservadas, y se emplean como nombres de funciones.

La función and [nombre de compuerta] (salida, entrada1, entrada2); describe un and de dos entradas. Donde el nombre de compuerta puede ser opcional. Importante es destacar que estas compuertas pueden tener "muchas" entradas.

La función or(x, ~a, b, ~c); representa a la compuerta or de tres entradas que puede describirse por la ecuación lógica: $x = \overline{a} + b + \overline{c}$

Las compuertas definidas son: and, nand, or, nor, xor, xnor, buf, not. La salida es el primer argumento de la lista.

El inversor y el buffer pueden tener varias salidas, el último elemento de la lista es la entrada.

Para representar un sistema se requiere definir un **módulo**, el que se describe como una función del lenguaje C; en la cual los argumentos son los nombres de las variables de entrada, de salida y bidireccionales del módulo.

Luego deben colocarse **listas de variables** de entrada, de salida y bidireccionales. El módulo debe estar terminado con la sentencia **endmodule**.

Pueden emplearse comentarios, para mejorar la legibilidad.

Los identificadores de variable deben comenzar con una letra o el carácter '_'. Cuando se definen identificadores se consideran distintas las letras minúsculas y mayúsculas.

La Figura A5.2 representa una red lógica con dos salidas y cuatro entradas. Su descripción puede efectuarse mediante un módulo.

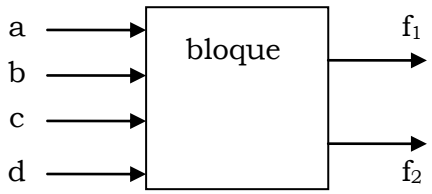

Figura A5.2

```
module  bloque(a, b, c, d, f1, f2);      //note que termina en punto y coma.
  input a, b, c, d;   // comentario de fin de línea
```

```
  output  f1, f2;
/*
    Aquí se describen las funciones lógicas mediante compuertas o expresiones.
*/
endmodule
```

Ejemplo A5.1.

Se tiene la función de cuatro variables:

$$Z(A,B,C,D) = \sum m(0,3,5,12,13) + \sum d(1,2,15)$$

Empleando alguna herramienta de minimización (ver A3.2 en Apéndice 3) se obtiene la forma minimizada en suma de productos:

$$Z = B\bar{C}D + \bar{A}\bar{B} + AB\bar{C}$$

La ecuación se puede traducir en el módulo siguiente, notar el empleo de los nodos internos: n1, n2 y n3 para formar la salida Z.

```
module ejemploA5_1(A, B, C, D, Z);
input A, B, C, D;
output Z;
wire n1, n2, n3;
  and(n1, B, ~C, D);
  and(n2,  ~A, ~B);
  and(n3, A, B, ~C);
  or(Z, n1, n2, n3);
endmodule
```

Con fines de simulación puede asociarse un **retardo de propagación** a cada compuerta. Por ejemplo, la descripción: and #10 (x, y, z) indica que esa compuerta and tiene asociado un retardo de 10 unidades de tiempo de simulación. También puede darse un nombre a cada compuerta, escribiéndolo antes de la lista de las señales.

En un proceso de síntesis no es recomendable asociar retardos a las compuertas, ya que los algoritmos pueden aumentar su tiempo de ejecución buscando satisfacer los requerimientos de tiempo de las compuertas. Algunos programas de síntesis no consideran estos retardos en el proceso de síntesis.

Ejemplo A5.2.

Mux de dos vías a una, implementado mediante compuertas.

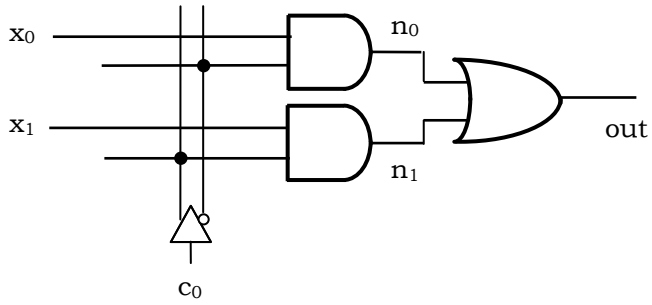

Figura A5.3

La ecuación del mux de la Figura A5.3, como suma de productos, puede expresarse según:

out(c0, x1, x0) = c0 x1 + c0' x0

El módulo puede escribirse directamente a partir de la ecuación.

```
// Multiplexer nivel compuertas. multiplexer 2-1, vías de 1-bit
// Cuando c0=1, out=x1; y cuando c0=0, out=x0

module mux21_gt (out, c0, x1, x0);
  output out;              // mux output
  input  c0, x1, x0;   // mux inputs
  wire n1, n0;         // nodos internos
  and(n0, x0, ~c0);
  and(n1,  x1, c0);
  or(out, n1, n0);

endmodule
```

Las conexiones o **nets** permiten ir uniendo componentes de un diseño digital. En el ejemplo anterior; n1 y n0, los nodos internos, son declarados **wire** (en español alambres) y son un tipo de net. Las señales tipificadas con wire, no tienen capacidad de almacenamiento; se consideran continuamente alimentadas por la salida de un módulo o por una asignación continua assign. Si las señales de tipo wire de entrada se dejan desconectadas, toman el valor x, que significa desconocido. Una variable de tipo wire es simplemente un rótulo sobre un alambre.

Una señal con **tipo de datos** de modo wire representa la actualización continua, en circuitos combinacionales, de las salidas respecto a cambios de las entradas. Otro tipo de datos, que suele estar asociado a salidas de flip-flops es **reg**, que es abreviatura de registro.

A5.2.2. Nivel transistores.

Verilog permite descripciones a nivel de conmutación de transistores, lo cual permite modelar compuertas.

Ejemplo A5.3.

Descripción de inversor CMOS en nivel de conmutación.

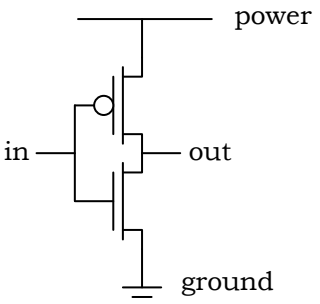

Figura A5.4

La Figura A5.4, muestra las señales empleadas en el módulo. Los tipos de conexiones o **nets**: supply1 y supply0 conectan señales a un voltaje de polarización y a tierra, respectivamente.

Los transistores cmos se especifican con tres argumentos (salida, entrada, control), y se dispone de las formas nmos y pmos.

El siguiente módulo implementa al inversor cmos de la Figura A5.4.

```
module inv_sw (out, in);
  output  out;          // salida inversor
  input   in;           // entrada inversor

  supply1 power;        // "power" conexión a Vdd .  Tipos de net.
  supply0 ground;       // "ground" conexión a Gnd

  pmos (out, ground, in);   // instancia switch pmos
  nmos (out, power, in);    // instancia switch nmos

endmodule
```

Ejemplo A5.4.

Multiplexor basado en compuertas de transmisión.

La Figura A5.5 muestra a la izquierda un símbolo para la compuerta de transmisión y, a la derecha, su implementación basada en un par de transistores. Cuando ctrl=0 la salida está en alta impedancia; cuando ctrl=1 la salida es igual a la entrada.

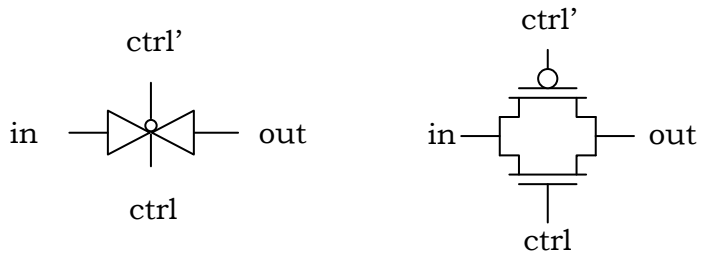

Figura A5.5

Verilog dispone de un modelo intrínseco para la compuerta de transmisión. Para instanciar una compuerta de transmisión se emplea:

cmos [nombreinstancia]([output],[input],[nmosgate],[pmosgate])

La Figura A5.6 muestra un multiplexor de dos vías a una. Cuando ctrl=1, se tiene que out=in2; y cuando ctrl=0 se tendrá out=in1.

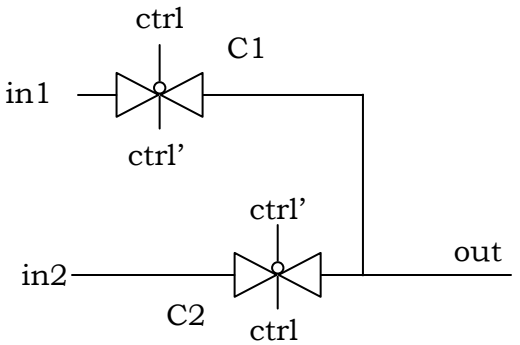

Figura A5.6

La implementación de este multiplexor se logra con el siguiente módulo, en el cual se ha generado la señal ctrl' a partir de ctrl y un inversor; el cual se instancia según el modelo anterior y con nombre I1. Se definen dos compuertas de transmisión con nombres C1 y C2.

```
// Multiplexer nivel conmutación. 1-bit 2-1 multiplexer
// Cuando ctrl=0, out=in1; y cuando ctrl=1, out=in2

module mux21_sw (out, ctrl, in1, in2);

  output out;              // mux output
  input  ctrl, in1, in2;   // mux inputs
  wire         w;          // nodo interno de tipo wire

  inv_sw I1 (w, ctrl);        // instancia inversor

  cmos C1 (out, in1, w, ctrl);  // instancias  switches cmos
```

```
  cmos C2 (out, in2, ctrl, w);

endmodule
```

A5.3 Descripción del comportamiento (Behavior).

Cuando son numerosas las ecuaciones de la red puede ser muy laboriosa la descripción de la estructura mediante compuertas.

Los lenguajes permiten una descripción más abstracta, y a la vez compacta, de las redes booleanas, ya que puede representarse el comportamiento o la conducta de la red.

Se describe lo que debe efectuar el sistema, empleando sentencias del lenguaje; es decir, la red booleana se describe como un programa.

Lo que el programa describe en este nivel son los registros y las transferencias y transformaciones de vectores de información entre los registros; este nivel se denomina **RTL** (Register Transfer Level).

También podría decirse que la descripción del comportamiento es una descripción de la **arquitectura** del sistema digital.

El diseñador se puede concentrar en el análisis de arquitecturas alternativas, mediante la simulación, determinando las partes del diseño que requerirán un estudio más detallado.

Este lenguaje permite **describir con precisión la funcionalidad** de cualquier diseño digital.

La descripción del comportamiento puede efectuarse de dos modos: Mediante **expresiones** lógicas o mediante **procedimientos**.

El compilador del lenguaje traduce estas representaciones abstractas a ecuaciones lógicas y éstas a su vez son mapeadas a los bloques lógicos del dispositivo programable.

Ejemplo A5.5. Expresiones.

Mediante la **asignación continua**, y escribiendo expresiones lógicas, empleando los operadores lógicos al bit del lenguaje C, puede obtenerse la misma funcionalidad de la descripción de Z mediante compuertas, vista en el Ejemplo A5.1. La expresión del lado derecho se evalúa continuamente a medida que cambian arbitrariamente las entradas; el lado derecho es un alambre que es la salida del sistema combinacional.

```
module ejemploA5_5(input A, B, C, D, output Z);

   assign Z =  (B & ~C & D) | ( ~A & ~B) | (A & B & ~C );

endmodule
```

Ejemplo A5.6. Mux mediante Expresiones.

El mux visto antes, en el Ejemplo A5.2, puede describirse:

```
// Multiplexer nivel expresiones . 2-1 multiplexer de 1-bit.
// Cuando c0=1, out=x1; y cuando c0=0, out=x0
module mux21_exp (output out, input c0, x1, x0);

  assign out= (c0&x1) | (~c0&x0);

endmodule
```

Operadores.

Los **operadores al bit** del lenguaje C: ~ para el complemento a uno, & para el and, | para el or, y ^ para el xor, efectúan la operación bit a bit del o los operandos. Además se dispone de los siguientes operandos: ~& para el nand, ~| para el nor, y: ~^ o ^~ para la equivalencia o not xor.

Se dispone de operadores **unarios** que efectúan la **reducción del operando a un bit**. Lo que efectúan es aplicar el operador a todos los bits de la señal. Se tienen: & para la reducción AND, | para la reducción OR, ^ para la reducción XOR, ~& para la reducción NAND, ~| para la reducción NOR, y ~^ para la reducción XNOR.

Verilog dispone de dos operadores adicionales, no incluidos en el lenguaje C, que son la igualdad y desigualdad, considerando los valores no conocido x, y alta impedancia z. En A5.6 se describe la forma de escribir valores. Se anotan: (a === b) y (a !== b), en estas operaciones si se incluyen bits con valores x ó z, también deben los correspondientes bits ser iguales para que la comparación sea verdadera, valor 1; o valor 0, en caso contrario.

En los test de igualdad o desigualdad (a == b) y (a != b), que están presentes en C, si existen operandos con valores x ó z, el resultado es desconocido, es decir x.

Ejemplo A5.7. Procesos.

La descripción de la conducta mediante **procedimientos** está basada en la definición de un bloque **always**. Esta construcción está acompañada de una **lista de sensibilidad**, cuyo propósito es evaluar las acciones dentro del procedimiento **siempre que** una o más de las señales de la lista cambie.

Dentro del bloque pueden usarse las sentencias de control: if-then-else, case, while, for.

Si se produce un evento de la lista de sensibilidad y se desea que la salida cambie inmediatamente, se efectúa una asignación **bloqueante** (se usa el signo igual); esto modela la conducta de un proceso combinacional. El calificativo bloqueante deriva del modelo de simulación, orientado a eventos, del lenguaje Verilog; en este sentido se dice bloqueante, ya que todos los otros eventos quedan pendientes hasta que se produzca la renovación del valor; es decir, esta asignación no puede ser **interrumpida** por otras órdenes Verilog concurrentes.

Si se produce un evento de la lista de sensibilidad y se desea que la salida cambie luego de un tiempo, se efectúa una asignación **nobloqueante** (se usa el signo <=), esto modela el proceso de escritura en un flip-flop disparado por cantos. Apenas producido el evento, se recalculan los lados derechos de las asignaciones nobloqueantes; y una vez que se han actualizado todos los valores, correspondientes a ese paso de simulación, se copian en las variables a la izquierda.

Importa el orden en que se efectúan una serie de asignaciones bloqueantes; ya que las acciones dentro de un bloque de un proceso se ejecutan secuencialmente. Cuando se tiene una serie de asignaciones nobloqueantes, no importa el orden en que son escritas.

Cada bloque always define un **hebra de control**. Si hay varios bloques always en un módulo, se considera que están corriendo **concurrentemente**.

Las acciones dentro del bloque o proceso son evaluadas en orden secuencial. En esta construcción puede tenerse uno o más bloques always dentro del módulo; y por lo menos una acción en uno de los bloques always. Si dentro del procedimiento se asigna valores a una de las señales, ésta debe ser declarada de tipo **reg**.

```
module ejemploA5_7(A, B, C, D, Z);
input A, B, C, D;
output Z;
   reg Z;      //se marca como registrada pero es combinacional.

   always @(A or B or C or D)
     begin
       Z =  (B & ~C & D) | ( ~A & ~B) | (A & B & ~C );
     end
endmodule
```

Cuando se efectúa la síntesis de esta descripción, Z será la salida de compuertas, y no se usará un flip-flop para guardar el valor de Z.

Las entradas deben estar en la lista de sensibilidad, las salidas deben ser asignadas en cada una de las alternativas controladas por sentencias.

En forma similar se dispone de **initial** que crea un proceso que se realiza sólo una vez, lo cual permite inicializar variables y generar estímulos para simulaciones. El proceso always se ejecuta **siempre**.

La diferencia fundamental entre el tipo net y reg es que a este último se le asignan valores en forma explícita. El valor se mantiene hasta que se efectúe una nueva asignación. Las salidas de flip-flops serán de tipo reg.

No debe confundirse el tipo reg de Verilog, con un registro formado por flip-flops.

Dentro de un módulo se pueden tener procesos y asignaciones continuas.

Ejemplo A5.8. Mux mediante Procesos.

El mux visto en los Ejemplos A5.2 y A5.6, puede describirse mediante un proceso, según:

```
// Multiplexer nivel proceso . 2-1 multiplexer 1-bit de ancho.
// Cuando c0=1, out=x1; y cuando c0=0, out=x0

module mux21_proc (out, c0, x1, x0);
  output out;          // mux output
  input  c0, x1, x0;   // mux inputs
  reg out;
    always @(*)  //asterisco representa todas las entradas que pueden modificar la salida
     out= (c0&x1) | (~c0&x0);
endmodule
```

Ejemplo A5.9. Mux mediante lenguaje.

La introducción de elementos similares a los empleados en lenguajes de programación permite describir procesos en forma cada vez más abstracta. Como veremos esto facilita la generalización.

```
// Multiplexer nivel proceso abstracto. 1-bit 2-1 multiplexer
// Cuando c0=1, out=x1; y cuando c0=0, out=x0

module mux21_procAbstracto (out, c0, x1, x0);
  output out;          // mux output
  input  c0, x1, x0;   // mux inputs
  reg out;
  always @(*)
    if (c0==1) out=x1; else out=x0;
 endmodule
```

En descripciones del comportamiento, mediante lenguaje, se representa la funcionalidad del módulo, independiente de la implementación lógica en base a compuertas. La herramienta CAD debería resolver el descenso o **compilación a compuertas**.

Sentencias.

Los modelos que describen el comportamiento contienen procesos que controlan la simulación y que manipulan variables. Cada proceso tiene su propio flujo de acciones. Las acciones de cada proceso se ejecutan concurrentemente, permitiendo modelar sistemas digitales que son inherentemente concurrentes o paralelos.

Se dispone de similares construcciones de **secuenciación de acciones** que en el lenguaje C. Algunos detalles son diferentes: si se emplea una variable índice entera en un for debe ser declarada de tipo integer (no int); no se emplea ++ para incrementar variables, sino var=var+1; se emplea begin .. end en lugar de {..}.

En la sentencia case no son necesarios los break; en los casos pueden colocarse valores con x ó z, y sólo si sintonizan exactamente, se realiza la acción asociada al caso.

Existen adicionalmente casex y casez que permiten comparación con condiciones superfluas con valores que contengan x ó z respectivamente.

La ejecución de una acción puede ser retardada hasta que una condición se haga verdadera. Esto se logra con la construcción: **wait** (condición) acción;

Se dice que la sentencia wait es sensible a los **cambios de nivel**, a diferencia de los eventos controlados por @, que disparan la acción con los **cantos** de las señales de la lista.

Se evalúa la condición del wait, y si es falsa la acción se detiene; es decir se espera hasta que la condición sea verdadera, momento en el cual se realiza la acción.

Ejemplo de uso:
Cuando se habilita la escritura, con WEnable=1, se transfiere el contenido del busA al registro regA, puede escribirse:

```
wait (!WEnable)  regA <= busA;
```

Sincronización entre procesos.
A los eventos por cantos puede dárseles un nombre.
Por ejemplo la transferencia: @ trigger rega <= regb;
Puede ser realizada desde otro proceso, mediante: -> trigger;

Acciones en paralelo.
Entre las palabras **fork** y **join** las acciones se consideran de ejecución paralela. Los separadores **begin**, **end** encierran sentencias secuenciales. Si las sentencias tienen retardos, en un conjunto de acciones en paralelo, éstos se consideran respecto del inicio del bloque.

A5.4 Diseños jerárquicos.

Verilog permite descripciones **jerárquicas** de redes complejas. Por ejemplo un sumador completo de 4 bits puede describirse mediante la instanciación de cuatro módulos con las variables actuales de un sumador completo de un bit.

Si se tiene en un archivo, denominado sumadorcompleto.v, el siguiente módulo combinacional definido por asignaciones continuas:

```
module sumadorcompleto(cin, a, b, sum, cout);
    input cin, a, b;
    output sum, cout;

    assign sum = a ^ b ^ cin,          //  Operador or exclusivo
    assign cout = (a & b) | (a & cin) | (b & cin);
endmodule
```

El cual puede representarse en el esquema de la Figura A5.7.

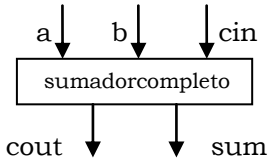

Figura A5.7

Alternativamente podría haberse efectuado una descripción estructural:

```
module sumadorcompleto (Cin, a, b, sum, Cout);
    input Cin, a, b;
    output sum, Cout;
    wire n1, n2, n3;
        xor (sum, a, b, Cin);
        and (n1, a, b);
        and (n2, a, Cin);
        and (n3, b, Cin);
        or (Cout, n1, n2, n3);
endmodule
```

El sumador de 4 bits, puede estar en otro archivo, en el cual se incluye la definición del módulo básico. Se le da un nombre a cada instancia, $etapa_i$ en el ejemplo, el que debe estar precedido por el nombre del módulo básico.

En la Figura A5.8, se ilustran las instancias y las correspondencias de los nombres de las puertas de cada etapa con los nombres del módulo básico. Con este diagrama resulta sencillo efectuar las instancias dentro del módulo.

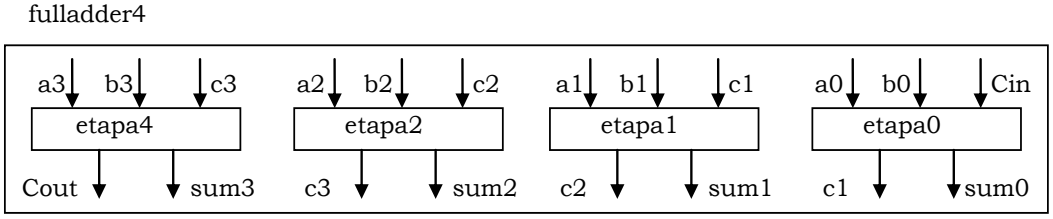

Figura A5.8

```
`include " sumadorcompleto.v"

module fulladder4 (Cin, a3, a2, a1, a0, b3, b2, b1, b0, sum3, sum2, sum1, sum0, Cout);
    input Cin, a3, a2, a1, a0, b3, b2, b1, b0;
    output sum3, sum2, sum1, sum0, Cout;
    wire c1, c2, c3;
    sumadorcompleto etapa0 (Cin, a0, b0, sum0, c1);
    sumadorcompleto etapa1 ( c1, a1, b1, sum1, c2);
    sumadorcompleto etapa2 ( c2, a2, b2, sum2, c3);
```

```
sumadorcompleto etapa3 ( c3, a3, b3, sum3, Cout);

endmodule
```

Los módulos no deben ser considerados procedimientos o funciones, en el sentido que no son invocados. Se dice que son instanciados al inicio del programa y permanecen hasta el final del programa. En el ejemplo anterior, fulladder4 crea 4 sumadorescompletos, denominados etapa0 hasta etapa3.

A5.5. Fundamentos de simulación.

El lenguaje debe proveer facilidades para generar estímulos, medir respuestas, y efectuar despliegues de los datos.

Empleando programación jerárquica un esquema que mantiene separado el módulo del diseño con el módulo de test, se visualiza en la Figura A5.9. La ventaja de esto es que el módulo con el diseño debe luego sintetizarse; de esta forma todas las sentencias que tienen que ver con la simulación quedan definidas dentro del módulo test; la finalidad de éste es generar los estímulos y grabar los resultados de los test. El módulo test podría describir un elemento de hardware, por ejemplo si se generaran los estímulos posibles mediante un contador o una memoria.

Debe notarse que las salidas del módulo test (estímulos) son las entradas del módulo que se está diseñando; y que las entradas del módulo test (respuestas) son las salidas del módulo que se está diseñando.

La activación del diseño y test se logra con un tercer módulo denominado Banco de prueba ubicado en el tope de la jerarquía, en el diagrama de la Figura A5.9.

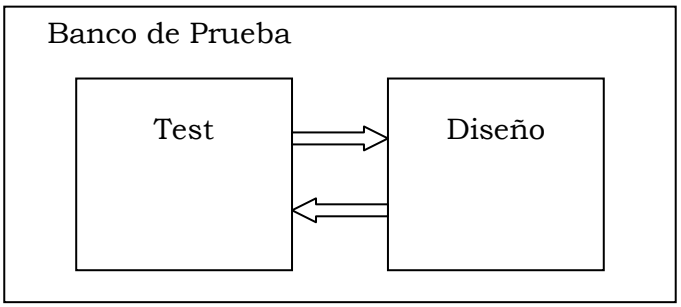

Figura A5.9.

Ejemplo A5.10. Banco de prueba para semisumador binario.

Efectuaremos un test de un módulo de semisumador escrito en forma estructural:

```
module semisumador(a, b, suma, cout);
input a, b;
output suma, cout;
  xor(suma, a, b); and(cout, a, b);
```

```
endmodule
```

En la confección del test del semisumador, cuyo texto se encuentra a continuación, se define un bloque que sólo se efectuará una vez. Esto se logra con la palabra reservada **initial**. El símbolo: #10 b=1; indica efectuar un **retardo** de 10 (unidades de tiempo de simulación) y luego efectuar la asignación. Con la secuencia dada se generan los estímulos para las entradas del semisumador, con valores para ab: 00, 01, 11, 10. La palabra **finish** termina la generación de estímulos.

La palabra $time muestra el tiempo en unidades de simulación.

El comando **monitor** muestra en la salida estándar, en formato similar a printf, los valores de la lista de variables; las comas adicionales se interpretan como espacios.

Cuando cambia uno de los valores de la lista del comando monitor, se produce impresión de una línea. Los valores se imprimen cuando todos los eventos del paso se han producido; es decir se imprime al final de un intervalo de simulación. El formato %b despliega en binario, %d en decimal, %h en hexadecimal. Sólo un monitor puede estar activo.

```
module testsemisumador(suma, cout, a, b);
input suma, cout;
output a, b;  //genera estímulos de entrada para el diseño
reg a, b;

initial
  begin
    $monitor("t=",$time,,"a=%b, b=%b, suma=%b, cout=%b", a, b, suma, cout);
    a=0; b=0;
    #10 b=1;
    #10 a=1;
    #10 b=0;
    #10 $finish;
  end
endmodule
```

EnVerilog se forman bloques de instrucciones que se ejecutan secuencialmente, encerrándolas entre las palabras reservadas: begin, end; esto es una diferencia con el lenguaje C que usa paréntesis cursivos.

El módulo BancoDePrueba, en el tope de la jerarquía, define las señales que se emplearán, las declara de tipo **wire**.

```
module BancoDePrueba;
wire a, b, su, co;
 semisumador      sm(a, b, su, co);
 testsemisumador  ts(su, co, a, b);
endmodule
```

Efectuando la simulación, se obtiene el listado:

```
t=        0 a=0, b=0, suma=0, cout=0
t=       10 a=0, b=1, suma=1, cout=0
t=       20 a=1, b=1, suma=0, cout=1
t=       30 a=1, b=0, suma=1, cout=0
```

Listado que permite verificar, en forma exhaustiva, el funcionamiento del semisumador. La prueba es completa, en este ejemplo, ya que se generan todas las combinaciones de estímulos posibles; en este caso se está verificando la tabla de verdad.

Funcionamiento del simulador.

En forma simplificada el funcionamiento del simulador es como sigue:

Initial y always comienzan a ejecutarse en el tiempo t=0, y continúan hasta encontrar un retardo (#); cuando esto ocurre, la condición que lo produce se almacena en la cola de los que esperan, por el tiempo indicado; luego del cual vuelve a colocarse el bloque en la cola de eventos activos para atender; es decir reasume la ejecución del bloque desde donde fue detenido.

Los siguientes **eventos activos** se itineran en cualquier orden: Salidas de compuertas, asignaciones continuas y bloqueantes, lectura de entradas y actualización de salidas, ejecución de comandos display, cálculo de lados derechos de asignaciones nobloqueantes.

Cuando, para un intervalo de tiempo de simulación, no quedan eventos activos por ser atendidos, se procesa la **cola de eventos de asignaciones nobloqueantes**, pasando al lado izquierdo los valores ya calculados.

Luego de lo cual se procesan los **eventos de monitor**, los comandos strobe y monitor. Después de esto se pasa a procesar los eventos asociados al siguiente intervalo de tiempo.

Los comados $monitor, $display y $strobe operan en forma similar a printf.

El comando **$monitor** muestra los valores actualizados de todas las variables de su lista, al final de un paso de simulación y después que todas las asignaciones, para ese paso de simulación, hayan sido efectuadas.

El comando **$display** muestra los cambios que se producen antes de que las asignaciones nobloqueantes sean actualizados, ya que se considera evento activo.

El comando **$strobe** permite mostrar las asignaciones nobloqueantes una vez realizadas.

Cuando un proceso está en ejecución sólo se detiene al encontrar un retardo (#) o un evento de una lista de sensibilidad (@) o el comando wait, este último comando espera hasta que su variable argumento sea verdadera.

Un bloque always o proceso debe contener alguno de los elementos de control de tiempo para que la simulación se detenga. El bloque continúa su ejecución: cuando transcurre el retardo, o cuando cambia una de las señales de la lista de sensibilidad o cuando ocurre el evento que espera wait. Un evento produce otros eventos. Son éstas las únicas instancias en que se

asume que el tiempo transcurre; el resto de los comandos (if-else, case, etc.) se considera que son instantáneos.

Los procesos detienen la simulación en determinados instantes, y también lo hacen las activaciones de salidas de compuertas.

Los eventos son itinerados para tiempos específicos y son colocados en una cola de espera, ordenados en el tiempo, para ser atendidos. Los primeros eventos quedan al frente, los últimos al fondo. Estando en el intervalo presente pueden procesarse varios eventos, no importando el orden; una vez atendidos todos los eventos asociados al intervalo presente, se pasa al siguiente. Durante el procesamiento pueden generarse nuevos eventos, que son ubicados en los lugares temporales que les corresponden para su posterior procesamiento.

Verilog emplea la sintaxis de C, pero es un lenguaje que permite describir **procesos concurrentes**, los eventos descritos en el párrafo anterior no están implementados en C, que es un lenguaje procedural.

La simulación en Verilog se produce por eventos que ocurren en determinados instantes; es una **simulación por eventos discretos**.

En una asignación bloqueante, las evaluaciones y asignaciones son inmediatas. En una no bloqueante, las asignaciones se postergan hasta que todos los lados derechos hayan sido evaluados, lo que efectúa al final del intervalo de simulación.

Si el módulo es combinacional se recomienda emplear asignaciones bloqueantes; y si el módulo es secuencial sincronizado por eventos, emplear asignaciones no bloqueantes.

Ejemplo A5.11. Simulación nivel transistores.

```
// Estímulos para verificar el inversor CMOS nivel de transistores del Ejemplo A5.3.
module stimulus;
  reg       in;          // se escribe en in. Se le da tipo reg.
  wire      out;

  inv_sw I0 (out, in);   // instancia inversor

  initial begin
    in=0; $display("%b, %b, t=%0d", in, out, $time);
    #1 in=1'b1; $display("%b, %b,t=%0d",in, out, $time);
    #4 in=1'b0; $display("%b, %b,t=%0d",in, out, $time);
    #5 in=1'b1; $display("%b, %b,t=%0d",in, out, $time);
    #5 in=1'b0;
    #5 $finish;
  end
endmodule
```

Generando la siguiente traza:

```
0, x, t=0       //Al inicio, en t=0, aún no hay salida. No se conoce es x.
```

1, 0, t=1
0, 1, t=5
1, 0, t=10

El descriptor 0 antes de d, en el string de control del display, especifica no generar espacios, e imprimir el tiempo en ancho variable, de acuerdo a la magnitud.

El siguiente proceso da un máximo tiempo a la simulación.
initial begin: DetenerSimulacion
// Detiene la simulación después de 400 unidades de tiempo de simulación.
#400; $stop;
end

A5.6. Buses, registros, vectores, arreglos. Valores constantes.

En numerosas situaciones de sistemas digitales, las variables de entrada y salida son multivaluadas, sin embargo ha sido tradicional codificarlas como **vectores booleanos**.

En general cuando se agrupan una serie de señales con un nombre común se origina un **bus** o vector booleano.

En Verilog, los **tipos de datos** net y reg pueden ser declarados como vectores. Existen mecanismos de acceso a partes o a la totalidad del vector

Puede convenirse en establecer el bit más significativo del vector como el que se escribe primero; además importa el orden con el que se declara el ancho del bus.

El siguiente ejemplo define out como la salida de un registro de 8 bits.
reg [7:0] out;

Se tiene que el bit 7 es el más significativo, el 0 el menos significativo. Convenio little endian.

Valores.

Para describir **constantes** o valores se emplea la notación: <largo>'<base><valor>. Donde la base suele ser binaria o hexadecimal; si no se coloca '<base> se asume decimal. A los valores binarios 1 y 0, se agregan los valores **z** y **x** que representan **alta impedancia** y **desconocido**, respectivamente.

Con estos nuevos valores, la operación que realiza una compuerta and se rige por la siguiente tabla de verdad.

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

Figura A5.10.

Para una compuerta or se tiene:

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

Figura A5.11.

Los bits de un vector o bus pueden ser referenciados empleando la notación:
[<bit de partida>:<bit de término>]

De esta forma puede inicializarse un registro en forma parcial, asignado un valor a los primeros cuatro bits de la variable out, mediante:
out[7: 4] = 4'b1010;

La operación **concatenación**, agrupa los elementos separados por comas en una lista definida entre paréntesis cursivos.
out = {A[3:0], 4'b0101};

Adicionalmente puede emplearse el comando de **repetición** que especifica el número de veces que se repite una concatenación: {2{4'b1010}} repite dos veces la secuencia 1010.
Una constante en complemento dos se anota con el signo precediendo la secuencia: -4d'6 define el complemento dos de 6 en un largo de 4 bits.

Los operandos al bit realizan la operación bit a bit de los operandos.
4'b0101 & 4'b0011 = 4'b0001

El operador monádico, al bit, se aplica a cada uno de los bits, formando el complemento uno.
~(4'b0101) = {~0,~1,~0,~1} = 4'b1010

Los operadores lógicos retornan un bit de resultado.
!(4'b0101) = ~1 = 1'b0

Los operadores de reducción realizan la operación sobre cada uno de los elementos del vector.
&(4'b0101) = 0 & 1 & 0 & 1 = 1'b0

Si la expresión es con signo (se agrega s antes de la base) el número se extiende con signo.
3'sb1 se extiende a 3'sb111; 3'sb0 se extiende a 3'sb000
Si es unsigned se extiende con ceros. 3'b1 se extiende a 3'sb001

El nombre de una variable puede tener componentes. En este caso se define un **arreglo.**
reg datos[31:0]; // define a datos como un arreglo de 32 bits.
Puede tenerse acceso individual a los bits, por ejemplo: datos[5]=1'bz;
Notar que en la definición del arreglo los corchetes van después del nombre del arreglo.

Una **memoria** es un arreglo de registros.
reg [15:0] mem16_1024 [1023:0]; //1024 palabras de 16 bits cada una.

Ejemplo A5.12. Mux con dos buses de entrada.

El esquema en base a lenguajes permite generalizaciones.
Una de las generalizaciones más comunes es cuando en lugar de tener una señal se tienen varias con funcionalidad similar. Puede efectuarse generalizaciones relativas al ancho del bus.

El siguiente módulo modifica el del Ejemplo A5.12, cambiando las entradas y salidas, de un alambre, por un bus de 8 señales.

```
// Multiplexer nivel proceso abstracto. Multiplexer 2 vías a una de 8 bits cada una.
// Cuando c=1, out=x1; y cuando c=0, out=x0

module mux21_buses (out, c, x1, x0);

  output [7:0] out;          // mux outputs de 8 bits
  input  c;
  input [7:0] x1, x0;   // mux inputs de 8 bits cada una

  wire c;
  wire [7:0] x1, x0;
  reg [7:0] out;

  always @(*)
    if (c==1) out=x1; else out=x0;

endmodule
```

Debe notarse que una descripción estructural, en base a compuertas, sería muy costosa en términos de representación. Nótese que el segmento anterior puede ser generalizado a buses de mayor ancho.
Para esto puede emplearse una declaración de constantes:
parameter <constante>=<valor>;

El siguiente es un ejemplo de uso:
parameter ancho=16;
output [ancho-1: 0] out; //bus de 16 bits

Ejemplo A5.13. Mux con varios buses de entrada.

Puede generalizarse aún más el concepto de multiplexor, cambiando el número de vías de entrada. Se diseña un mux de 4 buses de entrada con 8 bits cada uno.

```
// Multiplexer nivel proceso abstracto. 4-1 multiplexer, buses de 8 bits.
```

```
// Cuando c=3, out=x3; cuando c=2, out=x2; cuando c=1, out=x1; cuando c=0, out=x0
module mux4_1 (out, c, x3, x2, x1, x0);
  output [7:0] out;          // mux output
  input  [1:0] c;
  input [7:0]  x3, x2, x1, x0;   // mux inputs

  wire [1:0] c;
  wire [7:0] x3, x2, x1, x0;
  reg  [7:0] out;

  always @(*)
  if (c==3) out=x3;
  else  if (c==2) out=x2;
  else  if (c==1) out =x1;
  else out= x0;
 endmodule
```

Cada vez más la descripción de la funcionalidad o del comportamiento se va alejando de las tradicionales representaciones de sistemas digitales mediante compuertas o esquemáticos.

Adicionalmente puede mencionarse que la entrada de diseños en base a una biblioteca de esquemáticos de componentes tiene sus limitaciones. Es posible que vengan bloques de mux, prediseñados de un ancho fijo; sin embargo si un diseñador requiere uno de otro ancho, tendrá dificultades; en diseños mediante lenguajes, para cambios como los que mencionamos, sólo basta cambiar una constante.

El proceso anterior también podría describirse mediante una sentencia switch case.

Ejemplo A5.14. Mux con operador condicional.

El siguiente es un ejemplo de uso de sentencia condicional.

```
module mux4to1 (input x0, x1, x2, x3, c1, c0, output out);
      assign out = c1 ? (c0 ? x3 : x2) : (c0 ? x1 : x0);
endmodule
```

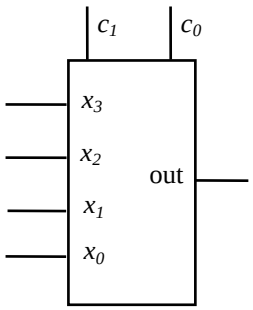

Figura A5.12. mux4to1.

Nótese que el módulo podría haberse desarrollado mediante vectores:

```
module mux4to1 (X, c, out);
    input [0:3] X;
    input [1:0] c;
    output out;
    reg out;

    always @(X or c)
        if (c = = 0)      out = X[0];
        else if (c = = 1) out = X[1];
        else if (c = = 2) out = X[2];
        else if (c = = 3) out = X[3];

endmodule
```

Dibujar un esquemático, empleando como componente al mux de 4 vías a una, es una buena manera de representar la arquitectura de un mux de 16 vías a una.
El diagrama ayuda a escribir el módulo de un mux de 16 vías a 1, en base a cinco muxes de 4 vías a una, mediante el empleo de vectores, puede escribirse:

```
module mux16to1 (X, C, out);
    input [0:15] X;
    input [3:0] C;
    output out;
    wire [0:3] n;     //nodos

    mux4to1 Mux1 (X[0:3],     C[1:0], n[0]);
    mux4to1 Mux2 (X[4:7],     C[1:0], n[1]);
    mux4to1 Mux3 (X[8:11],    C[1:0], n[2]);
    mux4to1 Mux4 (X[12:15], C[1:0], n[3]);
    mux4to1 Mux5 (n[0:3],     C[3:2], out);

endmodule
```

Sin embargo una representación compacta del mux de 16 a 1, mucho más simple, puede verse a continuación.

```
module mux16a1 (out, c, x);
output  out;        // mux output de 1 bit
input  [3:0] c;     // 4 líneas de control
input [15:0] x;   // mux input de 16 bits
wire [1:0] c;
wire [3:0] x;
wire out;

 assign out=x[c];
```

```
endmodule
```

Ejemplo A5.15. ALU con operandos vectores.

```
// 74381 ALU
module alu(s, A, B, F);
      input [2:0] s;
      input [3:0] A, B;
      output [3:0] F;
      reg [3:0] F;

      always @(s or A or B)
            case (s)
                  0: F = 4'b0000;
                  1: F = B - A;
                  2: F = A - B;
                  3: F = A + B;
                  4: F = A ^ B;
                  5: F = A | B;
                  6: F = A & B;
                  7: F = 4'b1111;
            endcase
endmodule
```

Ejemplo A5.16. Sumador completo con operandos vectores.

Empleando vectores se reduce apreciablemente la lista de argumentos en las puertas del módulo. Por ejemplo el módulo que representa a un sumador completo de 4 bits resulta:

```
module fulladder4 (Cin, a, b, sum, Cout);
      input carryin;
      input [3:0] a, b;
      output [3:0] sum;
      output carryout;
      wire [3:1] c;

      sumadorcompleto etapa0 (Cin,  a[0], b[0], sum[0], c[1]);
      sumadorcompleto etapa1 (c[1], a[1], b[1], sum[1], c[2]);
      sumadorcompleto etapa2 (c[2], a[2], b[2], sum[2], c[3]);
      sumadorcompleto etapa3 (c[3], a[3], b[3], sum[3], Cout);

endmodule
```

Comparar con A5.4.

Ejemplo A5.17. Estructuras repetitivas. Lazos for dentro de procesos.

Una de las ventajas de las descripciones mediante lenguajes, es que las estructuras que son repetitivas pueden ser planteadas dentro de un lazo for dentro de un proceso.

```
module AdderBitSlice (Cin, a, b, sum, Cout);
    parameter n=32;          //n es una constante
    input Cin;
    input [n-1:0] a, b;
    output [n-1:0] sum;
    output Cout;

// Las señales que son escritas dentro de un procedimiento deben ser declaradas reg.
    reg [n-1:0] sum;
    reg Cout;
    reg [n:0] c;       //reservas intermódulos
    integer k;         //variable entera para índice de for

    always @(a or b or Cin)
    begin
      c[0] = Cin;
      for (k = 0; k < n; k = k+1)
        begin
          sum[k] = a[k] ^ b[k] ^ c[k];
          c[k+1] = (a[k] & b[k]) | (a[k] & c[k]) | (b[k] & c[k]);
        end
      Cout = c[n];
    end

endmodule
```

La compilación o la traducción a compuertas del dispositivo programable puede que de origen a un tiempo considerable para efectuar la suma. Una de las soluciones es cambiar la arquitectura de la propagación ondulada de la reserva por algún esquema de generación adelantada y agrupada de las reservas. Esto también puede describirse mediante lenguajes.

Ejemplo A5.18. Descripción algorítmica, empleando operadores del lenguaje.

Empleando el operador que realiza la suma puede describirse un sumador binario en forma muy compacta. La operación de concatenación de registros permite generar Cout correctamente.
La señal overflow considera operandos con signo.

```
module addern (Cin, a, b, sum, Cout, overflow);
    parameter n = 32;
    input Cin;
    input [n-1:0] a, b;
    output [n-1:0] sum;
```

```
    output Cout, overflow;
    reg [n-1:0] sum;
    reg Cout, overflow;

    always @(a or b or Cin)
    begin
            {Cout, sum} = a + b + Cin;     //descripción funcional
            overflow = Cout ^ a[n-1] ^ b[n-1] ^ sum[n-1];
    end

endmodule
```

Las dos últimas columnas de la tabla de la Figura A5.13, muestran la generación de la señal overflow, sin emplear la entrada C31. Se ilustra la señal LT (less than) para operandos con signo.

A31	B31	C31	Co	S31	Ov	LT	A31^B31	C0^S31
0	0	0	0	0	0	0	0	0
0	0	1	0	1	1	0	0	1
0	1	0	0	1	0	1	1	1
0	1	1	1	0	0	0	1	1
1	0	0	0	1	0	1	1	1
1	0	1	1	0	0	0	1	1
1	1	0	1	0	1	1	0	1
1	1	1	1	1	0	1	0	0

Figura A5.13.

Ejemplo A5.19. Decodificador, empleando operadores del lenguaje.

Un decodificador de 2 líneas a 4, cuyas señales se muestran en el diagrama en bloques de la Figura A5.14, puede describirse funcionalmente según:

```
module dec2a4 (x, y, Enable);
    input [1:0] x;
    input Enable;
    output [0:3] y;
    reg [0:3] y;
    integer k;

    always @(x or Enable)
      for (k = 0; k <= 3; k = k+1)
        if ((x == k) && (Enable == 1))
          y[k] = 1;
        else
          y[k] = 0;
```

endmodule

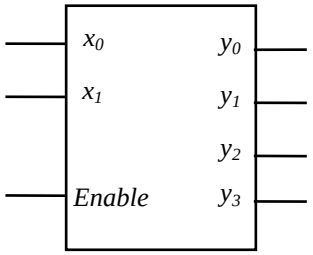

Figura A5.14.

Para la siguiente descripción jerárquica, de un decodificador de 4 líneas a 16, en base al módulo anterior, resulta útil desarrollar la arquitectura; es decir, un esquema de las conexiones entre los cinco bloques básicos, antes de escribir el código.

```
module dec4to16 (x, y, Enable);
    input [3:0] x;
    input Enable;
    output [0:15] y;
    wire [0:3] nodosenables;

    dec2to4 Dec1 (x[3:2], nodosenables [0:3], Enable);
    dec2to4 Dec2 (x[1:0], y[0:3],     nodosenables [0]);
    dec2to4 Dec3 (x[1:0], y[4:7],     nodosenables [1]);
    dec2to4 Dec4 (x[1:0], y[8:11],    nodosenables [2]);
    dec2to4 Dec5 (x[1:0], y[12:15], nodosenables [3]);

endmodule
```

El siguiente segmento ilustra una descripción abstracta del comportamiento de un decodificador, el cual se considera un sistema combinacional básico. Debe notarse lo compacto de la representación.

```
wire [15:0] Out;
wire [3:0] In;
assign Out = 1'b1 << In; //decodificador 4-a-16
```

Ejemplo A5.19a. Generador de paridad empleando lazo for.

El siguiente módulo genera la paridad de la entrada.

```
`define input_width 8
module paritygen (in, par);
input [`input_width - 1:0] in;
output reg par;
integer i;
```

```
  always @(in)
    begin
      par = 0;
      for ( i=0; i < `input_width; i=i+1)
            par = par ^ in[i];
    end
endmodule
```

Sin embargo la siguiente descripción es más compacta y simple.

```
module paridad (par, in);
parameter size = 8;
input [size-1:0] in;
output par;
  assign par = ^in;   //calcula paridad. Operador de reducción.
endmodule
```

Ejemplo A5.20. Buffer de tercer estado

La Figura A5.15 muestra un buffer de tercer estado, de 8 bits.

```
module tristate (X, OE, Y);
      parameter n = 8;
      input [n-1:0] X;
      input OE;
      output [n-1:0] Y;
      wire [n-1:0] Y;

      assign Y = OE ? X : 'bz;    //se rellena con n valores de alta impedancia.

endmodule
```

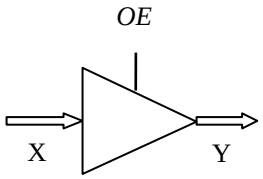

Figura A5.15.

No es usual encontrar compuertas de tercer estado en FPGA.

Ejemplo A5.21. Relación entre simulación y síntesis.

El siguiente segmento ilustra una situación de síntesis no combinacional. Esto se advierte ya que W y Z no son asignadas en cada una de las alternativas de control. Por ejemplo cuando A es 1, se cambia Z al valor de B; pero si A es 0, Z no cambia. Similar situación se tiene para W. Esta descripción plantea que en caso de ser A igual a 0, el sistema recuerda el valor anterior de

Z. Esta conducta no es de tipo combinacional, ya que los sistemas combinacionales no recuerdan valores previos. En esta situación al efectuar la síntesis se emplearán latchs para las variables Z y W.

```
input A, B;
output W, Z;
reg W, Z;
   always @(A or B )
     begin
         if (A= =1) Z=B;
         else W=A;
     end
```

En el caso del ejemplo, es preferible emplear Z<=B y W<=A, para las asignaciones, de esta forma la simulación y la síntesis darán iguales resultados.

En el proceso de **simulación**, del Ejemplo A5.21, se considera a Z y W como variables combinacionales, pero en el proceso de **síntesis** se las considerará secuenciales.

Si el diseño es de tipo combinacional, un riesgo importante de mal emplear el lenguaje Verilog, es no especificar una de las salidas en una "pasada" por un bloque always; en este caso se recuerda el valor anterior. Esto implica que se agregará una salida registrada al bloque combinacional. En caso de usar case, debe revisarse que exista una asignación, antes del case; o que esté especificada la opción por defecto, dentro del case. En caso de usar if, asegurarse que existe especificación else.

Otro riesgo es la interpretación literal de las condiciones. Por ejemplo: Se tienen 4 líneas en la entrada y se desea obtener el código binario de la línea que tenga valor uno. En caso de tener dos o más líneas activas la salida debe generar valor desconocido "x".

```
module binary_encoder(i, e);
input [3:0] i;
output [1:0] e;
reg e;
  always @(i)
    begin
      if (i[0]) e = 2'b00;
      else if (i[1]) e = 2'b01;
      else if (i[2]) e = 2'b10;
      else if (i[3]) e = 2'b11;
      else e = 2'bxx;
    end
endmodule
```

La síntesis lleva a una cascada de muxs, ya que las condiciones de los if, se forman con los bits individuales.

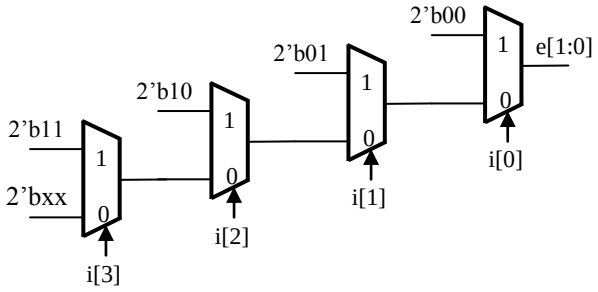

Figura A5.15a.

Mejores resultados se obtienen generando condiciones con evaluación "en paralelo" de las cuatro entradas.

```
module binary_encoder(i, e);
input [3:0] i;
output [1:0] e;
reg e;
  always @(i)
     begin
       if (i == 4'b0001) e = 2'b00;
       else if (i == 4'b0010) e = 2'b01;
       else if (i == 4'b0100) e = 2'b10;
       else if (i == 4'b1000) e = 2'b11;
       else e = 2'bxx;
     end
endmodule
```

Lo que debería llevar a un diseño combinacional de menor costo. En la Figura A5.15b, se indican dos diseños alternativos, que emplean eficientemente las condiciones superfluas.

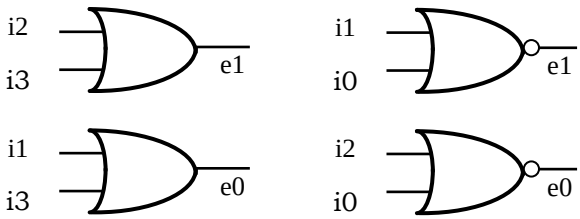

Figura A5.15b.

En algunos casos es conveniente utilizar alguna herramienta de minimización, para asegurar que el proceso de síntesis sea eficiente.

El siguiente código minimizado por espresso (ver Apéndice 3, sobre Uso de espresso), permite obtener los diseños de la Figura A5.15b.

```
.i 4
```

```
.o 2
.ilb i3 i2 i1 i0
.ob e1 e0
0001 00
0010 01
0100 10
1000 11
0000 --
0011 --
0101 --
0110 --
0111 --
1001 --
1010 --
1011 --
1100 --
1101 --
1110 --
1111 --
.e
```

En este caso también podría haberse realizado la minimización, empleando un mapa de Karnaugh.

La descripción mediante compuertas asegura la síntesis eficiente.

Ejemplo A5.22. Interconexión de módulos y descripciones del comportamiento.

En grandes diseños, la modularidad es esencial. Habiendo sido simulados y verificados los módulos componentes, la interconexión de las instancias de los módulos, resulta simple.

Se desea implementar una unidad aritmética con las siguientes funcionalidades:

F2	F1	F0	Función
0	0	0	A+B
0	0	1	A+1
0	1	0	A-B
0	1	1	A-1
1	0	x	A*B

Figura A5.15c.

Donde A y B son operandos de 32 bits, el resultado se deja en R, también de 32 bits. En el caso de la multiplicación se consideran sólo los 16 bits menos significativos de los dos operandos, de este modo el producto será también de 32 bits.

Una implementación directa, de la especificación:

```
module alu(a, b, f, out);
```

```
input [31:0] a, b;
input [2:0] sel;
output [31:0] out;
reg [31:0] out;
  always @ (a or b or f)
    begin
     case (f)
        3'b000: out = a + b;
        3'b001: out = a + 1;
        3'b010: out = a - b;
        3'b011: out = a - 1;
        3'b10x: out = a[15:0]*[15:0];
        default: out = 32'bx;
      endcase
    end
endmodule
```

El siguiente diagrama RTL, ilustra una solución posible, quizás no la de menor costo, pero directamente basada en la descripción abstracta anterior.

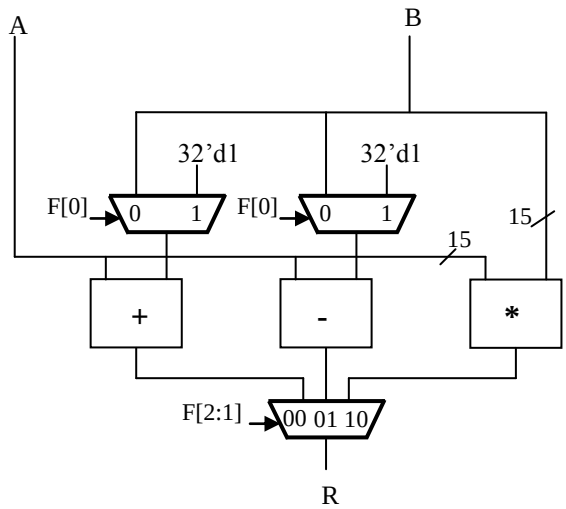

Figura A5.15d.

En este diagrama se han utilizado los siguientes módulos:
Dos mux de dos entradas de 32 bits cada una y con una salida de 32 bits.
Un mux de tres entradas de 32 bits cada una y con una salida de 32 bits.
Un sumador, y un restador con operandos de 32 bits.
Un multiplicador con operandos de 16 bits, y producto de 32 bits.

Basados en la arquitectura de la Figura A5.15d, se procede al diseño de los módulos:

```
module mux32two(i0, i1, sel, out);
```

```
input [31:0] i0, i1;
input sel;
output [31:0] out;
  assign out = sel ? i1 : i0;
endmodule

module mux32three(i0, i1, i2, sel, out);
input [31:0] i0, i1, i2;
input [1:0] sel;
output [31:0] out;
reg [31:0] out;
  always @ (i0 or i1 or i2 or sel)
    begin
     case (sel)
        2'b00: out = i0;
        2'b01: out = i1;
        2'b10: out = i2;
        default: out = 32'bx;
     endcase
    end
endmodule

module add32(i0,i1,sum);
input [31:0] i0,i1;
output [31:0] sum;
    assign sum = i0 + i1;        //descripción abstracta de comportamiento
endmodule

module sub32(i0,i1,diff);
input [31:0] i0,i1;
output [31:0] diff;
   assign diff = i0 - i1;
endmodule

module mul16(i0, i1, prod);
input [15:0] i0, i1;
output [31:0] prod;
    // en caso de que el hardware tenga unidades de multiplicación entera de 16 bits.
   assign prod = i0 * i1;
endmodule
```

Finalmente el módulo que implementa la unidad aritmética, empleando submódulos:

```
module alu(a, b, f, r);
input [31:0] a, b;
input [2:0] f;
output [31:0] r;
```

```
wire [31:0] addmux_out, submux_out;
wire [31:0] add_out, sub_out, mul_out;
  mux32two adder_mux (b, 32'd1, f[0], addmux_out);
  mux32two sub_mux (b, 32'd1, f[0], submux_out);
  add32 mod_adder(a, addmux_out, add_out);
  sub32 mod_subtracter(a, submux_out, sub_out);
  mul16 mod_multiplier(a[15:0], b[15:0], mul_out);
  mux32three output_mux(add_out, sub_out, mul_out, f[2:1], r);
endmodule
```

Debe notarse el orden de los argumentos en las instancias de los módulos.

Ejemplo A5.23. Módulos con operadores relacionales.

```
Se emplean descripciones de comportamiento.
module GeneraCondicion(gt, A, B);
parameter n=4;
output gt;
input [n-1:0] A, B;
    assign gt = (A > B);
endmodule

module ObtieneElMayor (mayor, A, B);
parameter n=4;
output [n-1:0] mayor;
input [n-1:0] A, B;
   assign mayor = (A > B) ? A : B;
endmodule
```

Ejemplo A5.24. Módulos con sentencias case.

Una forma de estructurar acciones de acuerdo a los valores que toma una expresión es la sentencia case. A diferencia con el lenguaje C, las sentencias break son implícitas; si se encuentra un valor de entre los casos igual al valor de la expresión, se realiza la acción asociada, y termina la "pasada" por el case.

Permite la representación tabular de operaciones mutuamente excluyentes como las que ocurren en máquinas de estados finitos. Existen versiones casez y casex, pero no se recomiendan para la síntesis.

La construcción siguiente:

```
case (expresión)
  case_item1 : acción1;
  case_item2 : acción 2;
  case_item3 : acción 3;
  case_item4 : acción 4;
  default : acción_por_defecto;
```

```
endcase
```

es equivalente a la desarrollada con if then else anidados.

```
if (expresión === case_item1) acción 1;
else if (expresión === case_item2) acción 2;
else if (expresión === case_item3) acción 3;
else if (expresión === case_item4) acción 4;
else acción_por_defecto;
```

La expresión y los valores de los ítems pueden ser cualquier expresión de tipo wire o reg. En general la expresión se evalúa a un valor en bits, el cual es comparado con los especificados enlos casos. No conviene emplear enteros para los valores de los casos, se recomienda especificar los bits de los valores. La expresión puede ser una constante, por ejemplo: 1'b1.

Las acciones asociadas pueden ser compuestas, en este caso deben estar entre un begin y un end. La acción por defecto se realiza si el valor de la expresión es diferente de los casos especificados; ésta no es necesaria si los valores de los casos cubren todos los posibles.

Si todos los posibles valores de la expresión sintonizan con a lo menos un valor de los casos o con el caso por defecto siempre que en éste se asignen todas las salidas, se denomina un caso completo. Si no se especifica un caso en forma completa, la síntesis empleará latchs. Esto último no es deseable en la descripción de situaciones combinacionales.

En el siguiente segmento, falta la especificación para el valor 2'b11. Cuando la expresión tenga este valor, el proceso no cambia el valor de y; es decir recuerda el valor anterior, y esto implica un latch para memorizar el valor anterior.

```
wire [1:0] sel;
reg y;
always @(*)
  case (sel)
    2'b00 : y = a;      //casos con valores binarios
    2'b01 : y = b;
    2'b10 : y = c;
  endcase
```

Si los valores de los casos son mutuamente excluyentes se tendrá que la expresión sólo puede tomar uno de los valores asociados a los casos. Se dice que es un caso paralelo; una entre varias vías posibles.

Si el estado actual puede ser S0, S1, S2, S3, el siguiente segmento que genera la salida asociada al estado, sintetizará con un latch la salida z, ya que el segmento establece que si el estado actual es S3, el sistema recuerda el valor anterior de z.

```
always @(EstadoActual)
case (EstadoActual)
```

```
S0, S2: z=0; //representación de la tabla de salida.
S1: z=4;
end
```

A5.7 Sistemas secuenciales.

Ejemplo A5.25. Latch transparente. (D gated latch)

La Figura A5.17 muestra las formas de ondas de un latch transparente, su principal característica, es que desde el canto de subida del reloj se copia la entrada hacia la salida (este es el modo transparente); y cuando ocurre el canto de bajada del reloj el flip-flop la salida queda capturada.

Un esquemático basado en compuertas, se muestra en la Figura A5.16.

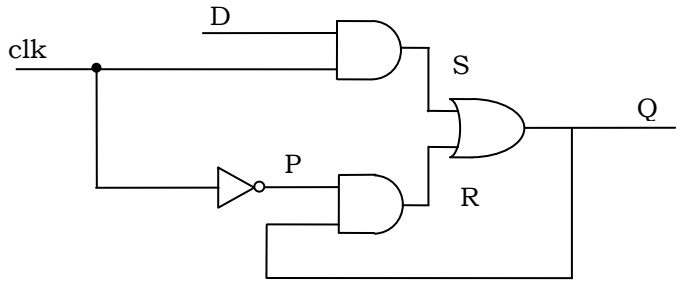

Figura A5.16.

Un módulo que describe la estructura usando compuertas, es el siguiente:

```
//latch con compuertas.
module latchc(input D, clk, output Q);
  wire P, S, R;

  and #1 (S, D, clk);
  not #1 (P, clk);
  and #1 (R,P,Q);
  or  #2 (Q, S, R); //retardo mayor es necesario para correcta simulación
endmodule
```

Para simular correctamente circuitos con compuertas realimentadas, es necesario definirles el retardo asociado a cada una de ellas.

La Figura A5.17, describe funcionalmente el latch transparente, empleando un multiplexor.

```
//latch proceso.
module latchp(input D, clk, output Q);
      reg Q;
      always @(D or clk)
        if (clk==1)  Q = D;
         else Q = Q;
```

endmodule

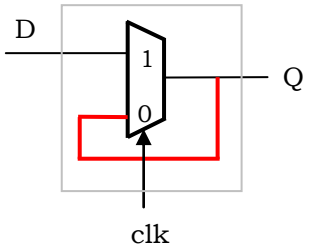

Figura A5.17.

Note que las asignaciones son bloqueantes, ya que se está representando un sistema cuya salida puede cambiar entre cantos del reloj.

No importando la arquitectura, un símbolo lógico del latch transparente se muestra en la Figura A5.18.

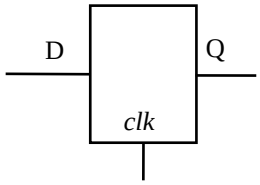

Figura A5.18.

La tabla característica del latch, que muestra el próximo estado en función de las entradas, se muestra en la Figura A5.19.

clk	D	Q(k+1)
0	x	Q(k)
1	0	0
1	1	1

Figura A5.19.

La Figura A5.20, muestra diferentes casos de secuencias de las entradas D y clk. Cuando el reloj tiene un canto de subida se deja pasar la entrada hacia la salida, cuando el reloj está bajo los cambios de la entrada no afectan al latch. Si ocurren cambios de la entrada con el reloj en alto, éstos son considerados.

Note en el intervalo t_3, la caída de Q cuando baja D. También debe advertirse que en el intervalo t_4, estando el reloj bajo, un pulso en D no alcanza a ser registrado por el latch. Luego de t4, estando el reloj alto, un canto de subida de D es capturado por el latch.

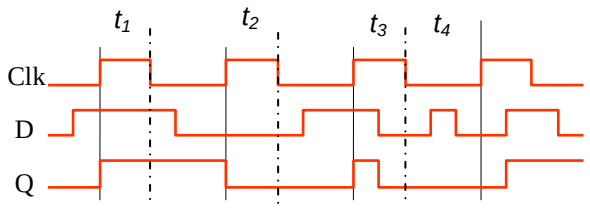

Figura A5.20.

El siguiente módulo genera estímulos y despliega los resultados al ser aplicados los estímulos de la Figura A5.20 a las implementaciones del latch transparente basadas en compuertas y en un mux.

```
module testlatch(input Q, output clk, D);

reg clk, D;
initial          //proceso que permite grabar en archivo las formas de ondas.
begin
    $dumpfile("latcht.vcd");
    $dumpvars(0, $time, clk, D, Q);
end

initial          //proceso que monitorea los cambios.
  begin
    $monitor("clk=%b, D=%b, Q=%b t=%d", clk, D, Q, $time);
    clk=0; D=1;   //D=1 para iniciar latch con compuertas.
    // clk=0; D=0;   //D=0 para iniciar latch basado en mux.
    #1 clk=1;
    #1 clk=0;
    #1 D=0;  //hasta aquí son inicializaciones para simulación.
    #10 D=1;
    #10 clk=1;
    #20 clk=0;
    #10 D=0;
    #20 clk=1;
    #20 clk=0;
    #10 D=1;
    #20 clk=1;
    #10 D=0;//efectúa cambio con reloj alto.
    #10 clk=1;
    #20 clk=0;
    #10 D=1;
    #10 D=0;
    #10 clk=1;
    #10 D=1;
    #10 clk=0;
```

```
    #20 D=0;
    #10 $finish;
  end
endmodule

module BancoDePrueba;
wire D, clk, Q, co;
  latchc    l1(D, clk, Q);
// latchp    l1(D, clk, Q);
  testlatch t1(Q, clk, D);
endmodule
```

Para el latch basado en compuertas, puede obtenerse la siguiente salida, empleando un simulador verilog, de los varios disponibles como software libre en la red.

```
clk=0, D=1, Q=x t=     0   //Al inicio Q es desconocido
clk=1, D=1, Q=x t=     1
clk=0, D=1, Q=x t=     2
clk=0, D=0, Q=x t=     3   //Aquí puede considerarse iniciada la simulación.
clk=0, D=0, Q=0 t=     4
clk=0, D=0, Q=0 t=     5
clk=0, D=1, Q=0 t=    13
clk=1, D=1, Q=0 t=    23
clk=1, D=1, Q=1 t=    26
clk=0, D=1, Q=1 t=    43
clk=0, D=1, Q=1 t=    46
clk=0, D=1, Q=1 t=    47
clk=0, D=0, Q=1 t=    53
clk=1, D=0, Q=1 t=    73
clk=1, D=0, Q=0 t=    77
clk=0, D=0, Q=0 t=    93
clk=0, D=1, Q=0 t=   103
clk=1, D=1, Q=0 t=   123
clk=1, D=1, Q=1 t=   126
clk=1, D=0, Q=1 t=   133
clk=1, D=0, Q=0 t=   136
clk=0, D=0, Q=0 t=   163
clk=0, D=1, Q=0 t=   173
clk=0, D=0, Q=0 t=   183
clk=1, D=0, Q=0 t=   193
clk=1, D=1, Q=0 t=   203
clk=1, D=1, Q=1 t=   206
clk=0, D=1, Q=1 t=   213
clk=0, D=1, Q=1 t=   216
clk=0, D=1, Q=1 t=   217
clk=0, D=0, Q=1 t=   233
```

Figura A5.21.

Para el basado en mux, puede obtenerse una visualización de las formas de ondas, a partir de un archivo en formato estándar vcd; existen aplicaciones de software libre para visualizar estos archivos. Ver referencias.

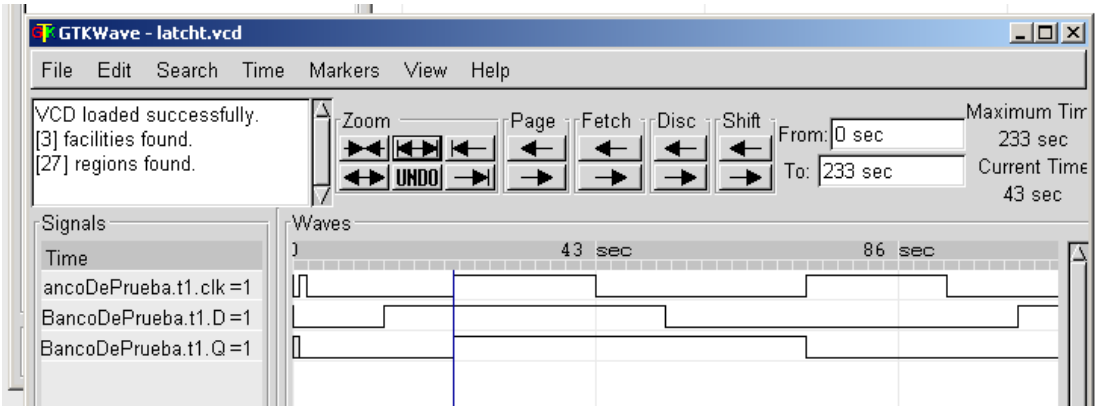

Figura A5.22.

En diseños complejos en dispositivos programables, no suelen emplearse latchs, ya que introducen señales no sincrónicas debido al modo transparente. Si se emplean FPGAs, la abundancia de flip-flops hace innecesario el uso de latchs.

Ejemplo A5.26. Flip-flop D. Disparado por canto de subida.

La Figura A5.23 muestra un flip-flop D disparado por cantos de subida del reloj. El módulo en Verilog, emplea la condición posedge para definir el evento canto de subida. La asignación a Q es no bloqueante.

El nombre de la función posedge recuerda **pos**itive edge (canto de subida, en español).

```
module flipflop (input D, clk, output Q);
      reg Q;
      always @(posedge clk)
            Q <= D;
endmodule
```

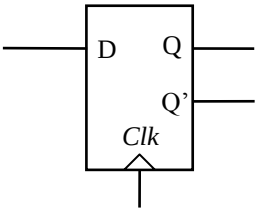

Figura A5.23.

La Figura A5.24 muestra el funcionamiento, ahora la señal Q permanece constante entre ciclos de reloj, notar la diferencia con la salida del latch de la Figura A5.20. La salida registra el valor de la entrada en el momento de ocurrir el canto. En un dispositivo real es preciso que no ocurran cambios de la entrada un intervalo antes del canto de subida del reloj (set-up); y un intervalo después del canto (hold).

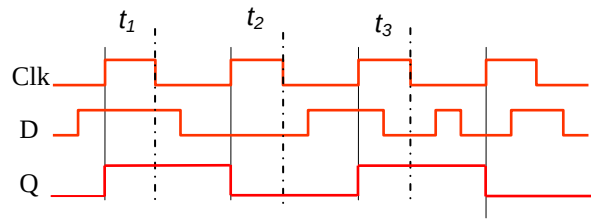

Figura A5.24.

Ejemplo A5.27. Flip-flop D. Disparado por canto de bajada.

En la descripción Verilog, basta cambiar el evento a negedge. En el símbolo lógico, de la Figura A5.25, se coloca un círculo que simboliza un inversor en la entrada del reloj.

```
module flipflop (input D, clk, output Q);
      reg Q;
      always @(negedge clk)
            Q <= D;
endmodule
```

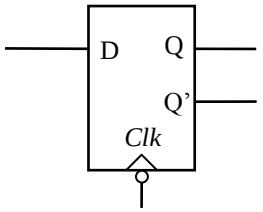

Figura A5.25.

Ejemplo A5.28. Flip-flop T.

T	Q(k+1)
0	Q(k)
1	Q'(k)

Figura A5.26.

```
module tff(input T, clk, output Q);
      reg q;
      always @ (posedge clk)
         if(T) Q<=~Q; else Q<= Q;
endmodule
```

Los sistemas secuenciales sincronizados por cantos se caracterizan por tener procesos always cuya lista de sensibilidad contiene negedge o posedge. Notar además, que si una de las señales de la lista contiene posedge o negedge, el resto de las señales de la lista también debe tenerlas.

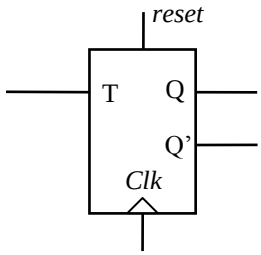

Figura A5.27.

Ejemplo A5.29. Controles de reset asincrónicos y sincrónicos.

Los siguientes segmentos ilustran un flip-flop D, con controles de reset sincrónico y asincrónico.

Si en el proceso (always) que describe al flip-flop D, la señal reset no está en la lista de sensibilidad, sólo se activarán los cambios de la salida cuando se produzca un canto de subida del reloj. Entonces el reset se producirá en un canto del reloj, por esto se dice que es sincrónico.

```
module flipflop (input D, clk, reset, output Q);
      reg Q;
      always @(posedge clk)
         if (reset) Q <= 0;       //reset sincrónico
         else  Q <= D;
endmodule
```

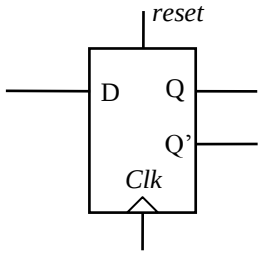

Figura A5.28.

El siguiente proceso produce el reset asincrónico del flip-flop D. Ya que se producirá el cambio de estado siempre que cambie la señal reset, que puede ser asincrónica con el reloj.

```
module flipflop (input D, clk, reset, output Q);
      reg Q;
      always @(posedge clk or posedge reset)
         if (reset) Q = 0;        //reset asincrónico
         else  Q <= D;
endmodule
```

El reset asincrónico se asigna en forma bloqueante, de esta manera la salida irá a cero apenas reset suba, incluso entre cantos de reloj.

En el ejemplo se emplean asignaciones bloqueantes y nobloqueantes a una misma señal; sin embargo no es recomendable mezclarlas en el mismo proceso o bloque always.

Ejemplo A5.30. Flip-flop D con habilitamiento de reloj.

```
input D, clk, enable ;
reg Q ;

always @ (posedge clk)
if (enable) Q < = D;
```

Los flip-flops de los bloques xilinx tienen habilitamiento del reloj.

Ejemplo A5.31. Registro n bits.

La descripción de la arquitectura de un registro con habilitación de escritura, teniendo como parámetro el ancho del registro, puede escribirse:

```
module registro (R, WE, Clk, Q);
    parameter n = 8;
    input [n-1:0] R;
    input WE, Clk;
    output [n-1:0] Q;
    reg [n-1:0] Q;

    always @(posedge Clk)
        if (WE)  Q <= R;

endmodule
```

Un diagrama lógico del registro se muestra en la Figura A5.29.

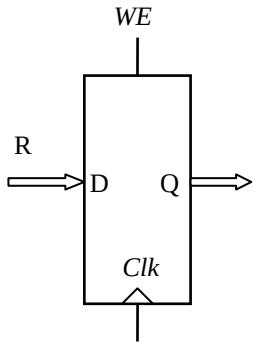

Figura A5.29.

Ejemplo A5.32. Carreras en simulación.

No es recomendable efectuar asignaciones a la misma variable en dos procesos diferentes, con el fin de evitar carreras.

Como se define que en la simulación que realiza Verilog, los procesos pueden ser itinerados en cualquier orden. Luego de un pulso de reset, las variables y1 e y2 podrían estar ambas en cero o ambas en uno. Situación que define una carrera.

```
reg y1, y2;

always @(posedge clk or posedge reset)
if (reset) y1 = 0; // reset
else y1 = y2;

always @(posedge clk or posedge reset)
if (reset) y2 = 1; // preset
else y2 = y1;
```

Si las asignaciones, dentro de los procesos, se cambian a nobloqueantes, luego de un reset, y1 quedará en cero e y2 en 1.

Ejemplo A5.33. Registro de desplazamiento.

La descripción de la arquitectura de un registro de desplazamiento lógico hacia la derecha de 4 bits, con la función adicional de carga paralela, puede plantearse:

```
module shift4 (C, Load, x, Clk, Q);
      input [3:0] C;    //carga paralela
      input Load, x, Clk;
      output [3:0] Q;
      reg [3:0] Q;
      always @(posedge Clk)
          if (Load) Q <= C;    //carga sincrónica
          else
            begin
               Q[0] <= Q[1]; Q[1] <= Q[2];
               Q[2] <= Q[3]; Q[3] <= x;
            end
endmodule
```

Debido a que las asignaciones son nobloqueantes, no importa el orden en que están escritos los corrimientos individuales de los bits.

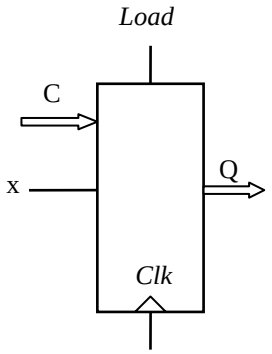

Figura A5.30.

El esquema anterior permite generalizar a un registro de ancho n, empleando una sentencia for.

```
//Registro de desplazamiento ancho n.
module shiftn (C, Load, x, Clk, Q);
    parameter n = 16;
    input [n-1:0] C;
    input Load, x, Clk;
    output [n-1:0] Q;
    reg [n-1:0] Q;
    integer k;

    always @(posedge Clk)
        if (Load) Q <= C;
        else
        begin
          for (k = 0; k < n-1; k = k+1) Q[k] <= Q[k+1];
          Q[n-1] <= x;
        end
endmodule
```

Sin embargo la notación preferida, por ser simple y compacta, para describir registros de desplazamiento es una descripción del corrimiento a la derecha, empleando la operación concatenación:

```
{Q, Q[0]} = {x, Q};
```

Ejemplo A5.34. Contador ascendente con reset de lógica negativa.

Una señal de lógica negativa se considera activada por cantos de bajada. En los diagramas lógicos suele denominarse en forma complementada, y su lugar de entrada al bloque pasa por un pequeño círculo. En la Figura A5.31 se muestra la señal con nombre complementado.

```
module upcounter (input reset, Clk, Enable, output [3:0] Q);
    reg [3:0] Q;
```

```
    always @(negedge reset or posedge Clk)
        if (!reset) Q <= 0;
        else if (Enable) Q <= Q + 1;
endmodule
```

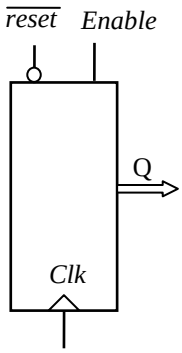

Figura A5.31.

Ejemplo A5.35. Contador con reset, carga paralela y habilitación para contar.

La Figura A5.32 muestra el contador con sus funciones.

```
module upcount (C, reset, Clock, Enable, Load, Q);
    input [3:0] C;
    input reset, Clk, Enable, Load;
    output [3:0] Q;
    reg [3:0] Q;

    always @(posedge reset or posedge Clk)
      if (!reset) Q <= 0;
      else if (Load)
          Q <= C;
      else if (Enable)
          Q <= Q + 1;
endmodule
```

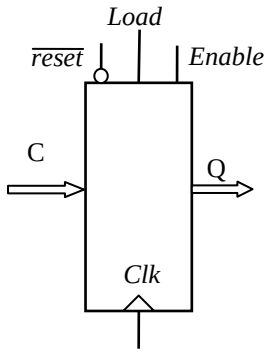

Figura A5.32.

Ejemplo A5.37. Divisor de frecuencia por n.

En un contador interno Cnt va contando, desde 0 hasta (Divisor-1) veces el reloj de alta frecuencia clk. Cuando cuenta el pulso Divisor mantiene durante un pulso, del reloj de alta frecuencia, un uno en el reloj de baja frecuencia; en el resto de las cuentas de cnt el reloj de baja frecuencia permanece en cero.

```
module divisor(input Clk, reset, output Clkd);
      parameter Divisor = 4;
      parameter Bitcnt = 1;  // 2 elevado a (Bitcnt+1) debe ser >= Divisor
      reg       Clkd;
      reg       [Bitcnt:0] Cnt;  //debe poder contar desde 1 a Divisor.

  always @ (posedge Clk or negedge reset)
      if(!reset)  begin Clkd <= 0; Cnt <= 0; end
      else if (Cnt == Divisor-1) begin Clkd <= 1; Cnt <=0; end
      else begin Clkd <= 0; Cnt <= Cnt+1; end

endmodule

module testdivisor(input clkd, output clk, reset);
reg clk, reset;
integer k;
initial
begin
    $dumpfile("divisor.vcd");
    $dumpvars(0, $time, reset, clk, clkd);
end

initial begin: DetenerSimulacion
// Detiene la simulación después de 100 unidades de tiempo de simulación.
//Es necesaria si se emplea simulador de reloj
#100 $stop;
end

initial
  begin
    $monitor("reset=%b, clk=%b, clkd=%b t=%d", reset, clk, clkd, $time);
    reset=0; clk=0;
    #5 reset=1;
    //alternativa para generar una cantidad finita de pulsos de reloj.
    //for(k=0;k<30;k=k+1) #10 clk<=~clk; $finish;      End

always @(clk)  #10 clk <=~clk;        //genera infinitos pulsos
endmodule

module BancoDePrueba;
```

```
wire reset, clk, clkd;
  divisor     d1(clk, reset, clkd);
  testdivisor t1(clkd, clk, reset);
endmodule
```

La Figura A5.33 muestra el período del reloj de baja frecuencia, se cumplen las relaciones, con T el período del reloj de alta frecuencia.

$$Clkd = \frac{Clk}{Divisor}$$

$$Td = Divisor \cdot T$$

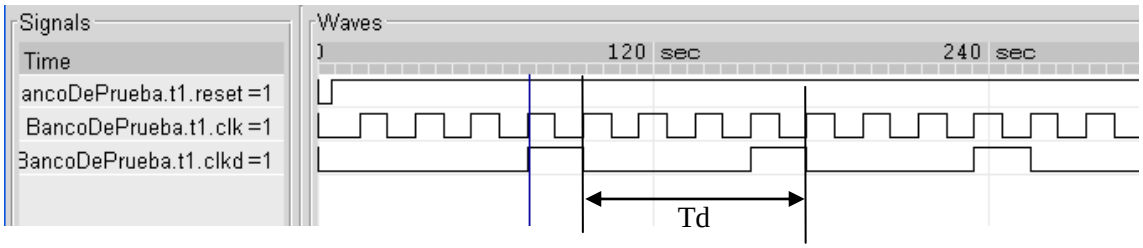

Figura A5.33.

Las formas de ondas fueron generadas con el archivo divisor.vcd, producido en la simulación.

El siguiente módulo emplea una asignación bloqueante para actualizar la señal clk. Este cambio se produce dentro del paso de simulación y no generará nuevos eventos; por lo tanto no oscila. La construcción correcta es con una asignación nobloqueante.

```
module oscilador (output clk);
reg clk;
  initial #10 clk = 0;
  always @(clk) #10 clk = ~clk;   //debe cambiarse a:  clk <= ~clk;
endmodule
```

Ejemplo A5.38. Transferencias entre registros de segmentación en pipeline.

En las transferencias entre registros en secuencia, de la Figura A5.34, deben emplearse asignaciones no bloqueantes. De este modo no importa el orden de las asignaciones y además cada una de las asignaciones podría estar en procesos always diferentes. Con esta elección, que refleja la naturaleza secuencial del proceso, la simulación y la síntesis dan iguales resultados.

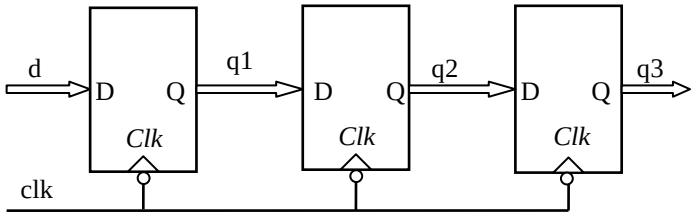

Figura A5.34.

```
module segmentos (q3, d, clk);
output [7:0] q3;
input [7:0] d;
input clk;
reg [7:0] q3, q2, q1;
  always @(negedge clk)
    begin
       q3 <= q2;
       q2 <= q1;
       q1 <= d;
   end
endmodule
```

Ejemplo A5.39. Registro de desplazamiento con realimentaciones lineales.

Un Linear Feedback Shift-Register (LFSR) es una máquina secuencial que posee realimentaciones. Uno de sus principales usos es en encriptación.

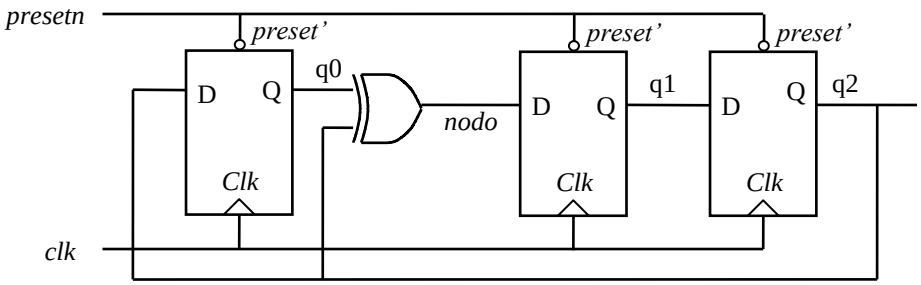

Figura A5.35.

```
module lfsr (output q2, input clk, presetn);
reg q2, q1, q0;
wire nodo;
assign nodo = q0 ^ q2;
always @(posedge clk or negedge presetn)
  if (!presetn) begin q2 <= 1'b1; q1 <= 1'b1; q0 <= 1'b1; end
  else begin q2 <= q1; q1 <= nodo; q0 <= q2; end
endmodule
```

Una alternativa para el cuerpo del proceso es emplear la construcción de concatenación.

```
always @(posedge clk or negedge presetn)
if (!presetn) {q2,q1,q0} <= 3'b111; else {q2,q1,q0} <= {q1,(q0^q2),q2};
```

El siguiente diseño considera un registro con carga paralela y notación vectorial.

```
module lfsr3(c, Load, clk, q);
      input [0:2] c;
      input Load, clk;
      output [0:2] q;
      reg [0:2] q;

      always @ (posedge clk)
      if(Load) q <= c; else q <= {q[2], q[0]^q[2], q[1]};
endmodule
```

Los siguientes módulos simulan el funcionamiento del módulo anterior.

```
module testlfsr(q, clk, Load, carga);
input [0:2] q;
output clk;
output Load;
output [0:2] carga;

reg clk, Load;
reg [0:2] carga;

initial
begin
    $dumpfile("lfsr3.vcd");
    $dumpvars(0, $time, Load,carga, clk, q);
end

initial begin: DetenerSimulacion
// Detiene la simulación después de 100 unidades de tiempo de simulación.
#400 $stop;
end

initial
  begin
    $monitor("Load=%b carga=%b clk=%b salida=%b t=%d", Load, carga, clk, q, $time);
    Load=0; clk=0;carga=0;
    #10 Load=1;
    #10 carga=1;
    #10 Load=0;
```

```
    #10 carga=0;
  end
always @(clk) #5 clk <=~clk;
endmodule

module BancoDePrueba;
wire Load, clk;
wire [0:2] c;
wire [0:2] q;

 lfsr3   lf1(c, Load, clk, q);
 testlfsr t1(q, clk, Load, c);
endmodule
```

Se obtienen las siguientes formas de ondas:

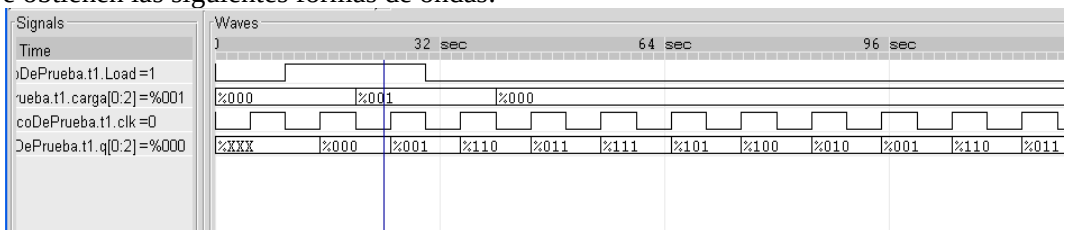

Figura A5.36.

Ejemplo A5.40. Contador de anillo

Se realimenta un registro de desplazamiento a la izquierda, formando un anillo. Se inicia el bit menos significativo en 1, y se lo va desplazando. Las cuentas son potencias de dos. El uso de la operación concatenación, permite describir la arquitectura en función de n, el número de bits del registro.

La señal reset', al activarse coloca en 1 el flip-flop q_0, mediante la señal S (set); y en cero el resto de los flip-flops, mediante la señal C (clear).

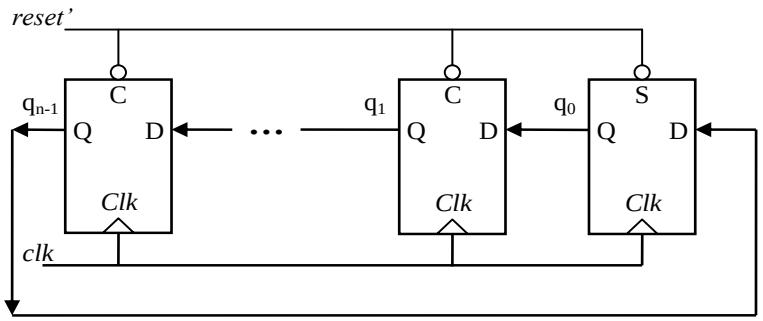

Figura A5.37

```
module cntanillo(resetn, clk, Q);
```

```
        parameter n = 8;
        input resetn, clk;
        output [n-1:0] Q;
        reg [n-1:0] Q;
        always @(posedge clk)
                if (!resetn) begin Q[n-1:1] <= 0; Q[0] <= 1; end
                else Q <= {{Q[n-2:0]}, {Q[n-1]}};
    endmodule
```

Ejemplo A5.41. Contador BCD de dos cifras.

El módulo siguiente contador BCD de dos cifras, módulo 100.
Con funciones de Clear, para dejar en cero y Enable para habilitar las cuentas.
La función Clear opera sincrónica con el reloj.

```
    module BCDcount (Clk, Clear, Enable, BCD1, BCD0);
        input Clk, Clear, Enable;
        output [3:0] BCD1, BCD0;
        reg [3:0] BCD1, BCD0;

        always @(posedge Clk)
        begin
           if (Clear)  begin BCD1 <= 0; BCD0 <= 0; end
          else
          if (Enable)
              if (BCD0 == 4'b1001)
              begin        //rebalse unidades
                  BCD0 <= 0;
                  if (BCD1 == 4'b1001) BCD1 <= 0; //rebalse decenas, módulo100.
                   else BCD1 <= BCD1 + 1;
               end
                  else  BCD0 <= BCD0 + 1;
             end
    endmodule
```

Ejemplo A5.42. Contador ondulado.

Se tiene el contador ondulado, de 4 bits, empleando flip-flops Ds.

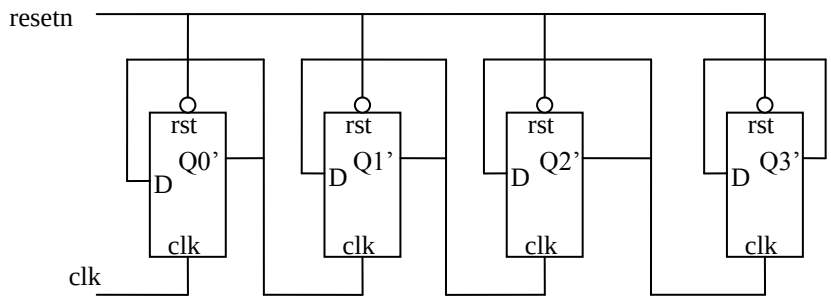

Figura A5.37a. Contador ondulado en base a flip-flops Ds.

La cuenta se toma de las salidas Q de cada flip-flop. Q0 es la menos significativa, la que varía más rápido.

Al estar todos los flip-flops en cero, al inicio, las salidas Q' son unos; lo cual divide por dos, la frecuencia de los diferentes relojes.

```
//Flip-flop D con reset asincrónico.
module dar (input reloj, clearn, d, output q, qprima);
reg q;
  always @ (posedge reloj or negedge clearn)
    begin
      if (!clearn) q <= 1'b0; else begin # 2 q <= d; end  //se agrega retardo para simulación.
     end
  assign qprima = ~q;  //combinacional, dentro del módulo.
endmodule
```

Para identificar los parámetros actuales en cada instancia, se precede con un punto el parámetro formal, y luego entre paréntesis el actual. En esta situación, no importa el orden en que se pasen los argumentos. Esta modalidad resulta útil en sistemas con varios módulos y con listas de varios argumentos.

```
module ripple_counter (input clk, resetn, output [3:0] cuenta);
wire [3:0] cuenta, cntprima;
dar bit0(.reloj(clk),              .clearn(resetn), .d(cntprima[0]), .q(cuenta[0]), .qprima(cntprima[0]));
dar bit1(.reloj(cntprima[0]), .clearn(resetn), .d(cntprima[1]), .q(cuenta[1]), .qprima(cntprima[1]));
dar bit2(.reloj(cntprima[1]), .clearn(resetn), .d(cntprima[2]), .q(cuenta[2]), .qprima(cntprima[2]));
dar bit3(.reloj(cntprima[2]), .clearn(resetn), .d(cntprima[3]), .q(cuenta[3]), .qprima(cntprima[3]));
endmodule
```

Para similar se genera el siguiente banco de prueba:

```
module test (output clk, resetn, input [3:0] cuenta);
reg clk,resetn;
integer j;
 initial
   begin
```

```
      $dumpfile("ripple.vcd");
      $dumpvars(0, clk, resetn, cuenta);
    end
initial
  begin
     $monitor("resetn=%bclk=%b q0=%b q1=%b q2=%b q3=%b", resetn,
                  clk,cuenta[0],cuenta[1],cuenta[2],cuenta[3],$time);
     resetn=1;clk=0;
       #5 resetn=0;
       #5 resetn=1;
     for (j=1;j<20;j=j+1)
       begin   #5 clk=~clk;  #5 clk=~clk; end
               #5 $finish;
       end
endmodule

module BancoDePrueba;
   wire clk,resetn;
   wire [3:0] cuenta;
   ripple_counter rc ( clk, resetn, cuenta);
   test t1 (clk, resetn, cuenta);
endmodule
```

Se generan las formas de ondas siguientes, las que muestran la naturaleza asincrónica del contador.

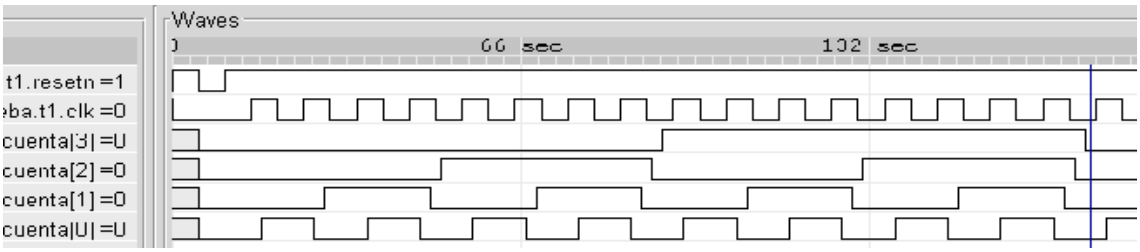

Figura A5.37b. Cuentas asincrónicas.

Ejemplo A5.43. Contador 74163.

El contador MSI 74163 puede ser descrito, por el siguiente módulo. Debe notarse que se dispone de señales clear y load de tipo sincrónicas y de lógica negativa, y que el AND de P y T habilitan el contador. Se dispone de una salida RCO ripple carry out, que favorece la implementación de contadores en cascada.

```
module contador163(Loadn, Clrn, P, T, Clk, D,
count, RCO);
input Loadn, Clrn, P, T, Clk;
input [3:0] D;
output [3:0] count;
output RCO;
```

```
reg [3:0] Q;
always @ (posedge Clk)
  begin
    if (!Clrn) Q <= 4'b0000;
    else if (!Loadn) Q <= D;
    else if (P && T) Q <= Q + 1;
  end
  assign count = Q;
  assign RCO = Q[3] & Q[2] & Q[1] & Q[0] & T;
endmodule
```

En la práctica las salidas de los diferentes flip-flops no cambian simultáneamente, esto lleva a que el circuito combinacional que genera RCO tendrá salidas espúreas debidas a carreras en las entradas. El módulo Verilog, podría evitar estos "glitches" sincronizando la salida RCO.

A5.8. Máquinas secuenciales de estados finitos.

Las herramientas de apoyo al diseño permiten ingresar un diagrama de estados en forma gráfica; el cual luego es traducido a un programa Verilog.

Una alternativa es describir directamente mediante lenguajes. La situación puede plantearse mediante dos o tres procesos.

A5.8.1. Diagramas de Moore.

Para una máquina de Moore, se tendrán tres procesos, los cuales se describen en la Figura A5.38.

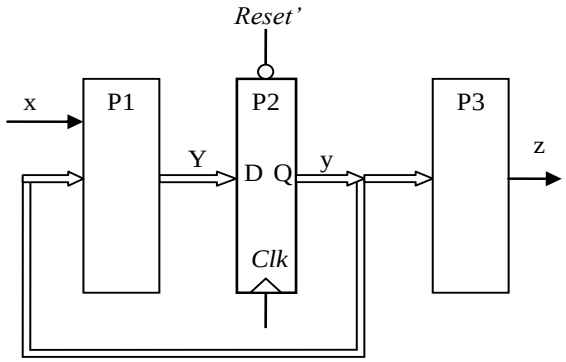

Figura A5.38

El diagrama de estados, de la Figura A5.39, describe una máquina que cuando detecta dos o más unos seguidos en la entrada x entrega salida z=1; en caso contrario la salida será z=0.

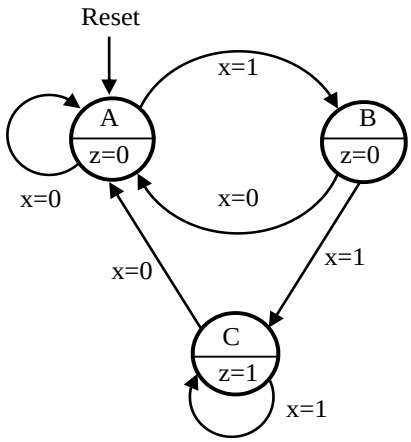

Figura A5.39

El siguiente módulo describe el diagrama de la Figura A5.39, siguiendo el esquema de la Figura A5.38.

```
//Detecta dos o más unos seguidos de la entrada x con salida z=1.
module moore (input Clk, Resetn, x, output z);
    reg [1:0] y, Y;
    parameter [1:0] A = 2'b00, B = 2'b01, C = 2'b10;
//Asignación de estados. Codificación binaria

    // Red combinacional de próximo estado. P1
    always @(x or y)
        case (y)
            A: if (x) Y = B;   else   Y = A;
            B: if (x) Y = C;   else   Y = A;
            C: if (x) Y = C;   else   Y = A;
            default: Y = 2'bxx;
        endcase
    // Proceso secuencial del registro. P2
    always @(negedge Resetn or posedge Clk)
        if (Resetn == 0) y <= A;
        else     y <= Y;          //estado presente<=próximo estado

    // Bloque combinacional de salida. P3
    assign z = (y = = C);
endmodule
```

La generación de estímulos en máquinas secuenciales es difícil. En casos sencillos es posible generar todas las secuencias que recorran las diferentes opciones que da el diagrama. Existen herramientas para generar estímulos, en forma gráfica e interactiva.

Como ejemplo se muestran algunos estímulos para revisar el funcionamiento, mediante simulación.

```
module test(output Clk, Resetn, x, input z);
reg  Clk, Resetn, x;
initial
  begin
    $monitor("clk=%b reset=%b x=%b z=%d", Clk, Resetn, x, z, $time);
    Clk=0; Resetn=1; x=0;
    #10 Clk=1; Resetn=0;
    #10 Clk=0; Resetn=1;

    #10 Clk=0;x=1;//un pulso de reloj => tres vectores.
    #10 Clk=1;x=1;//Cambios de señal con reloj bajo.
    #10 Clk=0;x=1;

    #10 Clk=0;x=0;//un pulso de reloj => tres vectores.
    #10 Clk=1;x=0;
    #10 Clk=0;x=0;

    #10 Clk=0;x=1;//un pulso de reloj => tres vectores.
    #10 Clk=1;x=1;
    #10 Clk=0;x=1;
    #10 Clk=0;x=1;//un pulso de reloj => tres vectores.
    #10 Clk=1;x=1;
    #10 Clk=0;x=1;

    #10 Clk=0;x=0;//un pulso de reloj => tres vectores.
    #10 Clk=1;x=0;
    #10 Clk=0;x=0;
    #10 $finish;
  end
endmodule

module BancoDePrueba;
wire Clk, Resetn, x, z;
  moore m1 (Clk, Resetn, x, z);
  test t1  (Clk, Resetn, x, z);
endmodule
```

A5.8.2. Esquema de Moore con salidas registradas.

El esquema general de la Figura A5.38, tiene ciertas dificultades prácticas y es que es posible que se produzcan perturbaciones (glitches), debidas a carreras en las entradas de la red combinacional de salida. Esto se debe a que los tiempos de propagación, entre el canto del reloj y el cambio de las salidas, no puede ser idéntico para los diferentes flip-flops. Una forma segura de tener salida libre de glitches, es generar la función combinacional de salida a partir del próximo estado, y registrar la salida. El esquema general se muestra en la Figura A5.40.

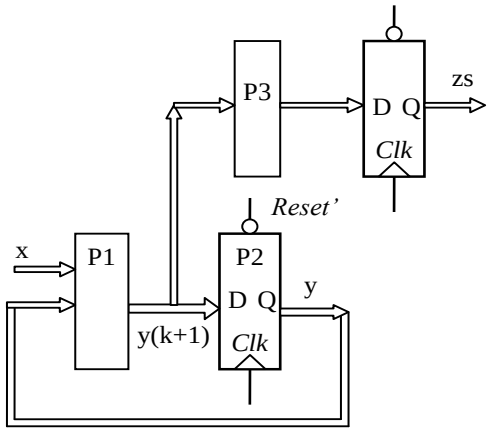

Figura A5.40

Básicamente se mueve el código asociado a P3, que calcula la salida, dentro del proceso always P2. Adicionalmente se calculan las salidas respecto al próximo estado.

Se ilustra el ejemplo anterior, con las modificaciones para obtener salida registrada libre de perturbaciones, debidas a las carreras ocasionadas por el cambio de estado.

```
//Detecta dos o más unos seguidos de la entrada x con salida z=1.
module MooreSalidaRegistrada (input Clk, Resetn, x, output zs);
    reg [1:0] y, Y, zs;
    parameter [1:0] A = 2'b00, B = 2'b01, C = 2'b10;
//Asignación de estados. Codificación binaria

    // Red combinacional de próximo estado. P1
    always @(x or y)
        case (y)
            A: if (x) Y = B;   else    Y = A;
            B: if (x) Y = C;    else   Y = A;
            C: if (x) Y = C;    else   Y = A;
            default: Y = 2'bxx;
        endcase
    // Proceso secuencial del registro. P2
    always @(negedge Resetn or posedge Clk)
      begin
        if (Resetn == 0) begin y <= A; zs<=0; end
        else      y <= Y;          //estado presente<=próximo estado
       // Salida registrada. P3
      zs < = (Y = = C);   //se efectúa el test con el próximo estado
      end
endmodule
```

A5.8.3. Diagramas de Mealy.

El programa Verilog, está basado en dos procesos, P1 es combinacional y P2 secuencial. Note que P2 es igual al de la máquina similar de Moore.

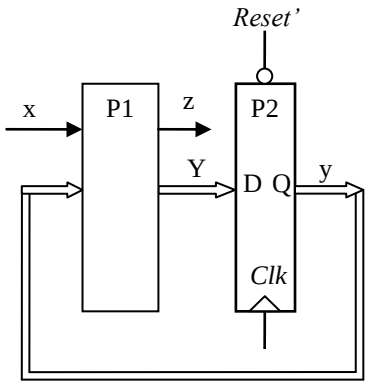

Figura A5.41

El diagrama de estados de Mealy, se muestra en la Figura A5.42, con un estado menos que el similar de Moore.

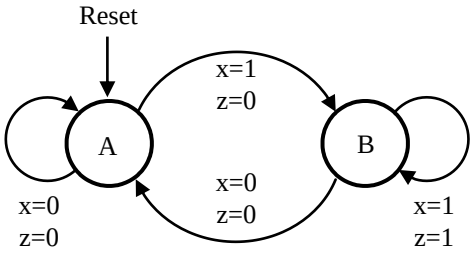

Figura A5.42

```
//Detecta dos o más unos seguidos de la entrada x con salida z=1.
module mealy (input Clk, Resetn, x, output z);
    reg y, Y, z;
    parameter A = 0, B = 1;
    // Redes combinacionales de próximo estado y de salida. P1
    always @(x or y)
      case (y)
        A: if (x) begin z = 0;  Y = B; end
           else   begin z = 0;  Y = A; end
        B: if (x) begin z = 1;  Y = B; end
           else   begin z = 0;  Y = A; end
      endcase

  // Proceso registro. P2
     always @(negedge Resetn or posedge Clk)
```

```
        if (Resetn == 0)    y <= A;
        else                y <= Y;

endmodule
```

Ejemplo A5.44. Máquina de Mealy.

Compara las entradas x1 e x2, si durante cuatro pulsos de reloj, se tienen secuencias iguales se produce salida z=1; en caso contrario la salida es cero.

Para las siguientes secuencias, se tiene la salida:

```
x1:  0 1 1 0 1 1 1 0 0 0 1 1 0
x2:  1 1 1 0 1 0 1 0 0 0 1 1 1
  z: 0 0 0 0 1 0 0 0 0 1 1 1 0
```

```
module ejemploMealy(input Clk, Resetn, x1, x2, output z);
    reg z;
    reg [1:0] y, Y;
    wire x;
    parameter [1:0] A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11;  //asignación simbólica
    // Define el próximo estado y el circuito combinacional de salida
    assign x = x1 ^ x2;
    always @(x or y)
            case (y)
            A: if (x) begin   Y = A; z = 0;    end
               else begin     Y = B; z = 0;    end
            B: if (x) begin   Y = A; z = 0;    end
               else begin     Y = C; z = 0;    end
            C: if (x) begin   Y = A; z = 0;    end
               else begin     Y = D; z = 0;    end
            D: if (x) begin   Y = A; z = 0;    end
               else begin     Y = D; z = 1;    end
    endcase

// Define el bloque secuencial
    always @(negedge Resetn or posedge Clk)
            if (Resetn == 0) y <= A;
                    else y <= Y;
endmodule
```

Ejemplo A5.45. Moore.

Árbitro de bus.
Las Figuras A5.42 y A5.43, ilustran una máquina secuencial para arbitrar un bus.

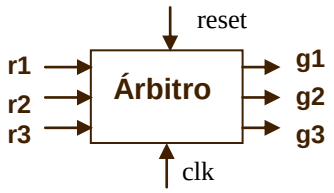

Figura A5.42

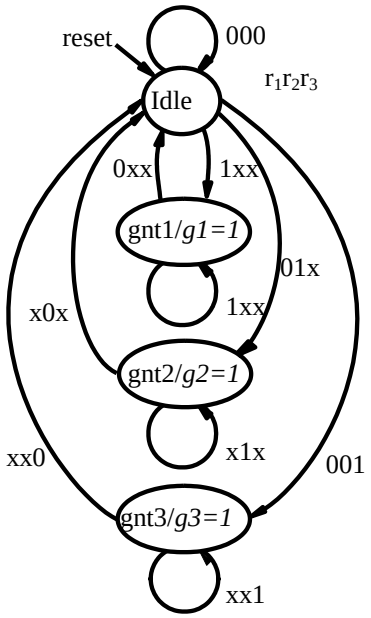

Figura A5.43

A5.9. Memorias.

Un arreglo de registros se sintetiza en un arreglo de flip-flops o en RAM, dependiendo de los primitivos que disponga el dispositivo.

```
//Memoria: 4 palabras de 16 bits.
module mem4x16(addr, din, dout, write, clk);
input [1:0] addr;
input [15:0] din;
input write, clk;
output [15:0] dout;
reg [15:0] mem[1:0]; // 4 x 16 bits
wire [15:0] dout;
    assign dout = mem[addr];  //lectura asincrónica
    always @ (posedge clk)
          if (write == 1'b1) mem[addr] <= din;
endmodule
```

A5.9.1. Accesos a RAM

Puede emplearse doble indexación.

```
reg [31:0] mem [0:1023]; // 1Kx32 RAM
wire [5:0] opcode;
wire [9:0] addr;
assign opcode = mem[addr][31:26] // selecciona un elemento del arreglo
                                 // y de éste los primeros 6 bits
```

La última versión de Verilog (2001), permite seleccionar cualquier elemento, bit o parte de cualquier arreglo. Los cuales pueden ser de varias dimensiones.

```
//Arreglo bidimensional de palabras de 32 bits.
reg [31:0] ArregloBid [0:255][0:15];

integer n, m;
wire [31:0] PalabraA = ArregloBid [n][m];  // selecciona una palabra del arreglo

wire [7:0] ByteA = ArregloBid [n][m][31:24];//selecciona un byte de una palabra del arreglo

wire Bit0 = ArregloBid [n][m][0]; // selecciona un bit de una palabra del arreglo
```

A5.9.2. Simular RAM.

Las tareas del sistema: $readmemb y $readmemh, leen y cargan datos desde un archivo de texto hacia la memoria. Estas tareas pueden ser invocadas en cualquier instante dentro de la simulación. La sintaxis del comando es:

$readmemh("<NombreArchivo>", <NombreMemoria>, <DirecciónInicial>, <DirecciónFinal>);

El archivo de texto debe contener solamente: blancos (espacios, tabs, fin de línea), comentarios, y los datos en forma de números binarios (para readmemb) o hexadecimales (para readmemh).

Para una RAM de 8x8, el siguiente segmento, inicia el contenido, y luego define la ram.

```
initial
$readmemb("contenido.out", mem, 0, 7); // inicia memoria. Lee archivo ascii
always @(posedge clk)
   begin
      if (write) mem[addr] <= din; // escritura sincrónica de la memoria
   end
assign # 5 dout = mem[addr]; // Lectura asincrónica de la memoria.

//contenido.out  formato binario
00000000
11000001
00010000
11000100
```

```
00010001
11000110
00010000
11001010
```

//contenido.out formato hexadecimal, equivalente al binario anterior.
00 C1 10 C4
11 C6 10 CA

A5.9.3. Parameter para instanciar.

Se emplea parameter para asignar un valor a un nombre simbólico. Suele emplearse para parametrizar módulos. Por ejemplo puede emplearse como valor por defecto cuando se instancia un módulo y puede cambiarse en el momento de instanciación.

```
module paridad (y, in);
parameter size = 8;
input [size-1:0] in;
output y;
  assign y = ^in;   //calcula paridad
endmodule

module top();
reg [15:0] data;
// instancia de paridad para palabras de 16 bits.
paridad  #(16) p0 (y, data);   //se cambia size a 16.
```

Un **localparam** no puede ser redefinido fuera del módulo.

```
module paramram(d, a, wr, q);
parameter AnchoDatos = 8, AnchoDireccion = 8;
localparam palabras = (2** AnchoDireccion)-1;
input [AnchoDatos -1:0] d;
input [AnchoDireccion -1:0] a;
input wr;
output wire [AnchoDatos -1:0] q;
reg [AnchoDatos -1:0] mem [palabras:0]; // declara el arreglo de memoria
  always @(wr) mem[a] = d; // write
  assign q = mem[a];            //read
endmodule
```

Si la RAM es de mayores dimensiones se sintetiza en bloques si el dispositivo tecnológico los posee.

```
module RamPuertoSimple (clk, we, addr, data_in, data_out);
parameter addr_width = 8;
parameter data_width = 8;
input clk, we;
```

```
input [addr_width - 1:0] addr;
input [data_width - 1:0] data_in;
output[data_width - 1:0] data_out;
reg [addr_width - 1:0] addri;
reg [data_width - 1:0] mem [(32'b1 << addr_width):0];  //2**8 =256 direcciones
  always @(posedge clk)
    begin
      if (we) mem[addr] = data_in;
      addri = addr;   //actualiza dirección i
    end
  assign data_out = mem[addri];   //lee cuando se cambia addri
endmodule
```

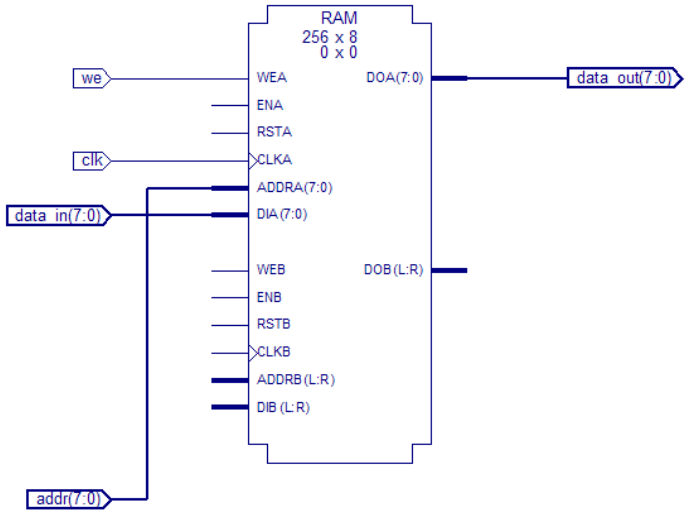

Figura A5.43a. Bloque de RAM 256*8

A5.10. Sistemas asincrónicos.

Se dan algunos ejemplos de sistemas asincrónicos. Mostrando las capacidades del lenguaje para describirlos y simularlos.

Ejemplo A5.46. Latch de nand.

```
module latchnand(q, qprima, setn, resetn);
output q, qprima;
input setn, resetn;

nand #1 G1(q, qprima, setn);
nand #1 G2 (qprima, q, resetn);

endmodule
```

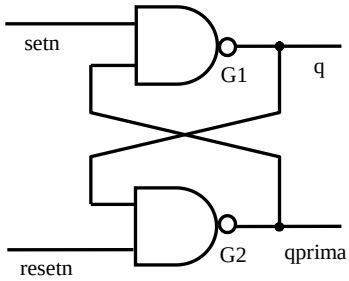

Figura A5.44

La siguiente serie de estímulos genera diferentes recorridos. Si ambos controles pasan por cero, no puede asegurarse el estado del latch.

```
initial
  begin
    $monitor("s=%b r=%b q=%b q'=%b", setn, resetn, q, qprima, $time);
    setn=1;resetn=1;
    #2 setn=0;
    #2 setn=1;
    #4 setn=0;
    #4 setn=1;
    #12 resetn=0;
    #4  resetn=1;
    #10 setn=0;resetn=0;
    #4  setn=1;
    #4  setn=0;
    #4  setn=1;
    #4  setn=0;
    $finish;

  end

module latchNAND (SN, RN, Q, QN);
input SN, RN;
output Q, QN;
wire Q, QN;
  assign Q = ~(SN & QN);
  assign QN = ~(RN & Q);
endmodule
```

Ejemplo A5.46. Descripción estructural flip-flop D, en base a nand.

```
//Descripción conducta flip-flop D
module dffc(output q, input clk, d);
reg q;
  always @(negedge clk) q = #10 d;
```

```
endmodule

//Descripción estructural flip-flop D
module dffe(output q, input clk, d);
wire q, qprima, r, s , r1, s1;
  nor G1 (q, qprima, r);
  nor G2 (qprima, q, s);
  nor G3 (s, r, clk, s1);
  nor G4 (s1, s, d);
  nor G5 (r, r1, clk);
  nor G6 (r1, s1, r);
endmodule
```

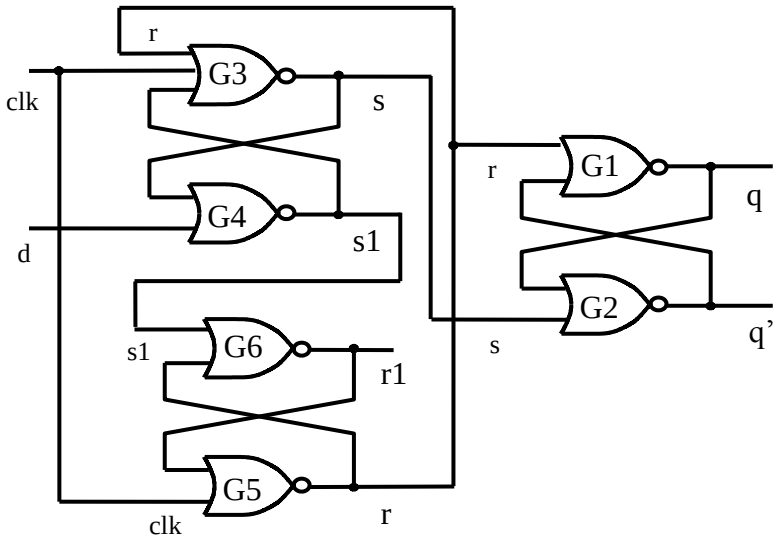

Figura A5.45

Ejemplo A5.46. Flip-flop RS.

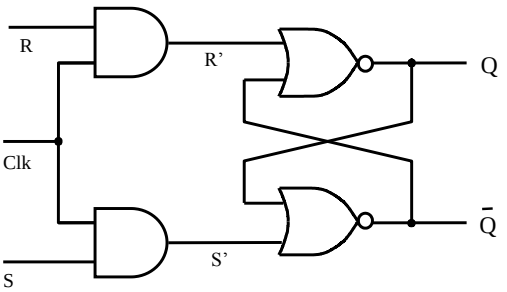

Figura A5.46

A5.11. Síntesis.

El proceso de síntesis describe las etapas para convertir, la descripción HDL (Verilog o VHDL) de un sistema digital, a una forma que pueda ser armada, manufacturada o descargada a un dispositivo.

El o los elementos en los que será implementado el diseño, se denomina sistema tecnológico objetivo (target), normalmente es de tecnología semiconductora. Se denomina ASIC (Application-Specific Integrated Circuit), a los dispositivos que se construyen especialmente para soportar un diseño particular. En los dispositivos programables como FPGAs y CPLDs el diseño es descargado al chip, que suele ser reprogramable.

Una metodología que puede emplearse es efectuar una descripción del comportamiento lo más abstracta posible, luego preparar una serie de estímulos que al ser aplicados al diseño, en una simulación, confirmen su funcionalidad. La preparación cuidadosa de un banco de pruebas suele ser más difícil que el diseño mismo.

Puede ser que algunas descripciones abstractas no puedan ser sintetizadas, ya que el programa de síntesis reconoce sólo las estructuras que se le han almacenado en su inteligencia artificial. Si por ejemplo puede reconocer: registros, sumadores, multiplexores, comparadores, contadores, etc., es porque se ha almacenado en bibliotecas la descripción (en Verilog) de estas unidades en términos de compuertas y flip-flops, o en términos de los bloques básicos del dispositivo (muxs, luts).

Si a pesar de lo abstracta de la descripción, ésta puede ser sintetizada, es decir traducida a la interconexión de compuertas y flip-flops, entonces se podría pasar a la siguiente etapa. En caso que esto no sea posible, el diseñador debe descomponer en partes su idea original, posiblemente mediante descripciones RTL de la arquitectura. Lo cual implica crear nuevos módulos, que deben ser sintetizables, cada uno de los cuales debe ser sometido a simulaciones funcionales cuidadosas.

Cuando ya se dispone de una red booleana, la siguiente etapa es minimizarla, normalmente en área, considerando que se tendrá multifunciones y multiniveles, esto es realizado por complejas heurísticas programadas en la herramienta CAD. Luego de esto viene el mapeo tecnológico, en el cual las configuraciones de compuertas y flip-flops se hacen calzar con los arreglos de multiplexores o tablas de búsqueda con que está formado el dispositivo tecnológico. Luego es necesario colocar e interconectar los bloques (en caso de dispositivos programables). En estas condiciones es preciso evaluar, mediante simulaciones temporales, si se cumplen los requerimientos de velocidad de los sistemas combinacionales y los tiempos de hold y set-up de los flip-flops. En caso de no cumplirse, deben rediseñarse partes y volver al lazo de diseño (ver Figura A5.1).

Mientras más abstracto el diseño, más rápido es el desarrollo.

Debido a la incorporación de nuevos algoritmos y heurísticas para efectuar la síntesis en términos de compuertas y flip-flops, debería evaluarse la necesidad de enfrentar un diseño en términos de más bajo nivel. Una forma de ir controlando el descenso de nivel, es llevar registro del número de elementos usados en el diseño y de los tiempos de ejecución, para cada una de las

arquitecturas que se vayan desarrollando; esto permitirá evaluar si es preciso confeccionar módulos con más nivel de detalle. Conocer los subsistemas que la herramienta de ayuda al diseño que se está empleando, sintetiza eficientemente, también permite decidir hasta donde descender en el nivel de descripción.

Una de las ventajas de Verilog, es que, por un lado, permite describir módulos en términos de compuertas y flip-flops, lo cual posibilita llegar a niveles muy bajos de abstracción. Por el otro extremo, a medida que se tienen más experiencias, se han desarrollado por expertos, algunas bibliotecas o núcleos que pueden ser reutilizados, entre ellos: memorias, multiplicadores, microcontroladores embebidos, lo cual facilita la realización de diseños abstractos sintetizables.

La forma más básica de programar o diseñar es mediante descripciones estructurales explícitas. Es decir construir módulos solamente con compuertas, sin emplear procesos (always, initial) ni asignaciones continuas (assign). Son dificultosas de escribir pero fáciles y seguras de sintetizar.

Las descripciones estructurales implícitas, emplean los operadores al bit del lenguaje y el comando assign para describir expresiones. Tampoco se emplean procesos. Estas descripciones son más fáciles de escribir y también puede asegurarse que son sintetizables.

Algunos operadores más complejos (como la suma) suelen ser reconocidos por las herramientas de síntesis; el operador condicional también es sintetizable ya que suele traducirse a multiplexores.

En el mapeo tecnológico se traducen las compuertas genéricas (por ejemplo compuertas de 8 entradas) a los bloques lógicos del dispositivo; también algunas operaciones pueden ser mapeadas tecnológicamente (por ejemplo: multiplicadores, registros de desplazamiento, memorias, si existen en el dispositivo), empleando este tipo de chips suele mapearse tecnológicamente con software que provee el mismo fabricante.

Una de las dificultades que tiene trabajar con herramientas de ayuda al diseño es que se pierde contacto rápidamente con el problema, en la forma en que fue escrito por el diseñador. Esto se debe a varias causas. Los compiladores suelen proceder a través de la generación de numerosos archivos intermedios y además usan innumerables archivos auxiliares con información; de los archivos generados para la compilación del diseño, algunos de ellos son binarios, otros con formatos adecuados a los datos y tablas del programa y no son legibles con facilidad por un humano; además si existen archivos de texto, los rótulos e identificadores son elegidos por un autómata, y son de difícil asociación para el programador. Es decir, prácticamente se pierde control del proceso de síntesis, sólo puede verse lo que la aplicación está diseñada para mostrar: resúmenes del proceso de síntesis, esquemas, etc.

Veremos el proceso de síntesis a través de algunos ejemplos.

A5.11.1. Síntesis de descripciones estructurales implícitas.

```
// Módulo combinacional que genera:  f= ab + ac + ad
module simple(output x, input a, b, c, d);
```

```
  assign x = a & b  |  a & c  |  a & d;
endmodule
```

La síntesis rtl, realizada por ISE de xilinx, produce.

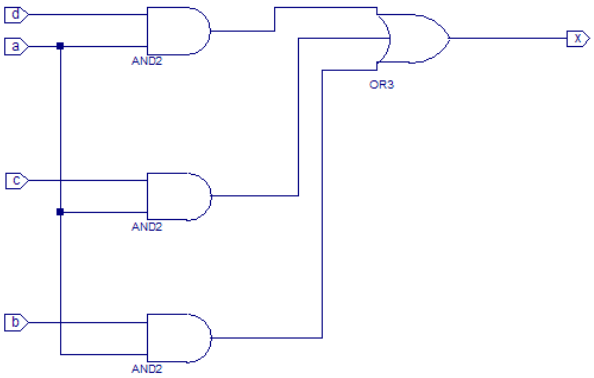

Figura A5.47. Síntesis RTL de ISE

La síntesis rtl genera un módulo en base a compuertas o módulos de biblioteca. La Figura A5.47, se describe a continuación en Verilog. Los nombres de las compuertas y los nodos internos han sido elegidos por un ser humano, y por esto el programa resulta fácil de leer.

```
 module simpleRtl(output x, input a, b, c, d);
  wire   n1, n2, n3;
    and a1  (n1, a, b);
    and a2  (n2, a, c);
    and a3  (n3, a, d);
    or     o1 (x, n1, n2, n3);
endmodule
```

Sin embargo es posible que diferentes programas de síntesis generen implementaciones diferentes, por ejemplo usando sólo compuertas de fan-in 2, se obtendría:

```
module simpleRtlFanin2(output x, input a, b, c, d);
  wire   n1, n2, n3, n4;
  and  a1  (n1, a, b);
  and  a2  (n2, n1, n2);
  or    o1  (n3, n1, n2);
  and  a3  (n4, a, d);
  or     o1 (x, n3, n4);
endmodule
```

Para ilustrar la rapidez con que la información detallada, de lo que se está realizando, se vuelve cada vez más incomprensible, se muestra la misma información del programa anterior, representada en un formato netlist xnf de xilinx (formato ya obsoleto), y generado por el compilador icarus, mediante el comando: iverilog –o simple.xnf –t xnf simple.v

```
LCANET,6
PROG,verilog,$Name: v0_8 $,"Icarus Verilog"
PART,
SIG, a, PIN=a
SIG, b, PIN=b
SIG, c, PIN=c
SIG, d, PIN=d
SIG, x, PIN=x
SYM, _s1, AND, LIBVER=2.0.0
   PIN, O, O, _s0
   PIN, I0, I, a
   PIN, I1, I, b
END
SYM, _s3, AND, LIBVER=2.0.0
   PIN, O, O, _s2
   PIN, I0, I, a
   PIN, I1, I, c
END
SYM, _s5, OR, LIBVER=2.0.0
   PIN, O, O, _s4
   PIN, I0, I, _s0
   PIN, I1, I, _s2
END
SYM, _s7, AND, LIBVER=2.0.0
   PIN, O, O, _s6
   PIN, I0, I, a
   PIN, I1, I, d
END
SYM, _s9, OR, LIBVER=2.0.0
   PIN, O, O, x
   PIN, I0, I, _s4
   PIN, I1, I, _s6
END
EOF
```

También se producen formatos netlist estandarizados, pero que son de difícil lectura para un ser humano.

Si el dispositivo físico, en que será depositado el diseño, tiene celdas básicas or y and de cuatro entradas, el módulo mapeado tecnológicamente y optimizado, resulta:

```
module simpleOptMapeado(output x, input a, b, c, d);
  wire   n1;
  or4    o1 (n1, b, c, d, 1'b0);
  and4  a1  (x, a, n1, 1'b1, 1'b1);
endmodule
```

Esto debido a que $f= ab + ac + ad$, puede (y debería) ser minimizada a: $f=a(c + b +d)$.

Si el dispositivo sólo tiene compuertas inversoras y nor de 2 y 3 entradas, se obtendría lo siguiente, asumiendo que el sintetizador optimiza las ecuaciones boolenas:

```
module simpleOptMapeadoOr(output x, input a, b, c, d);
  wire   n1, n2;
  inv    i1 (n1, a);
  nor3  no1  (n2, b, c, d);
  nor2  no2  (x, n1, n2);
endmodule
```

Si el sistema físico es una fpga, en base a LUTs de 4 entradas, basta una para implementar la tabla de verdad de la función f de cuatro variables. El resultado de la síntesis de mapeo tecnológico que realiza ISE de Xilinx, se muestra en la Figura A5.48.

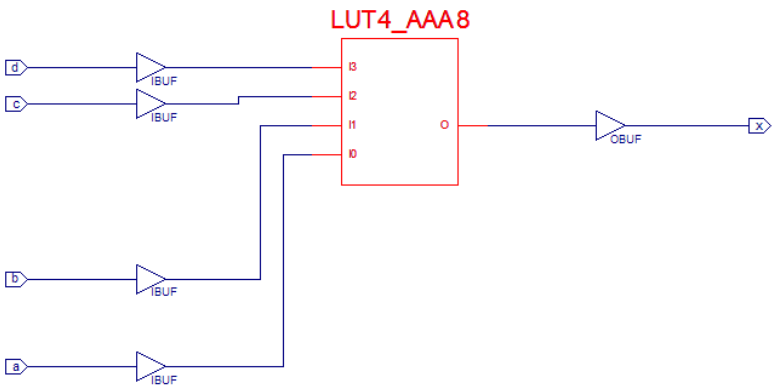

Figura A5.48 Síntesis con LUT. Mapeo tecnológico.

La ecuación implementada en la LUT, puede escribirse en términos de las entradas, según:

*Eqn= [(I0*I2) + (I0*I3) + (I0*I1)]*

El código AAA8 es el patrón hexadecimal almacenado en los 16 latchs o flip-flops de la LUT. Puede leerse el patrón o estado inicial: AAA8, en la última columna de la siguiente tabla.

I3	I2	I1	I0	O
1	1	1	1	1
1	1	1	0	0
1	1	0	1	1
1	1	0	0	0
1	0	1	1	1
1	0	1	0	0
1	0	0	1	1
1	0	0	0	0
0	1	1	1	1
0	1	1	0	0
0	1	0	1	1

0	1	0	0	0
0	0	1	1	1
0	0	1	0	0
0	0	0	1	0
0	0	0	0	0

Figura A5.49.

El modelo conceptual de una LUT, puede visualizarse como una ROM de 16x1.

```
module ROM16X1 (output O, input [3:0] addr);
   parameter INIT = 16'hAAA8;       //16 bits almacenados en ROM
   reg [15:0] mem;
     initial   mem = INIT;
     assign O = mem[addr];
endmodule
```

A5.11.2. Algunas referencias para el proceso de síntesis.

Todos los operadores que operan al bit (bitwise) son sintetizables, mediante grupos de compuertas.

Los operadores de reducción son sintetizables, mediante compuertas que tienen salida de un bit.

Los operadores aritméticos: +, - y * son sintetizables.

Los tipos reg y wire son sin signo a menos que se los declare con signo, mediante:

```
wire signed [31:0] suma;
```

Se realiza aritmética con signo sólo si ambos operandos son declarados con signo.

El operador aritmético para la potencia es sintetizable si la base es una constante potencia de dos, o si el exponente es 2.

```
assign resultado = 8 ** exponente;
assign resultado = base ** 2;
```

El operador división es sintetizable si el divisor es una constante que es potencia de 2.

```
assign result = divisor/1024;
```

El operador módulo es sintetizable si el operando del lado derecho es una constante potencia de 2.

```
assign result = a % 4;
```

Los bloques initial no son sintetizados. No se consideran los retardos en la síntesis.

La asignación continua assign se sintetiza a bloques combinacionales que alimentan alambres (wires).

Las asignaciones procedurales de procesos always se sintetizan a bloques combinacionales, o mediante latchs, o flip-flops, dependiendo del contexto.

Bloques always con lista completa de eventos o que empleen @(*) producen lógica combinacional.

Bloques always con lista de sensibilidad que emplee posedge o negedge se sintetizan empleando flip-flops disparados por cantos.

Bloques always con lista de eventos incompleta (if sin else, case incompleto, asignaciones a no todas las señales de la lista) se sintetizan con latchs.

No debería efectuarse asignaciones bloqueantes y no bloqueantes a una misma señal. O dos asignaciones a la misma señal, dentro de un bloque always.

Cuando se empleen operadores aritméticos debe considerarse el ancho de los operandos de entrada y de salida. Por ejemplo en una suma el resultado ocupa un bit adicional a los operandos. En caso de sumas de números sin signo, el bit más significativo del resultado es la reserva de salida.

```
module SumaConReserva(a, b, suma);
input [15:0] a, b;
output [16:0] suma;  //incluye reserva de salida
assign suma = a + b;
endmodule
```

Operadores lógicos son sintetizables mediante redes combinacionales.

```
module LogicalAnd (a, b, c);
input [3:0] a, b;
output c;
assign c = (a == 2) && (b == 3);
endmodule
```

La operación también puede describirse mediante decodificadores del número 2 y del 3. Esta forma muestra la interpretación del operador &&.

```
assign c =  (~a[3])&(~a[2])&(a[1])&(~a[0])&(~b[3])&(~b[2])&(b[1])&(b[0]);
```

El operador condicional ?, implica multiplexores.

```
module condicional (sel, a, b, y);
input sel, a, b;
output y;
assign y = sel ? a : b;
endmodule

//mux 4 a 1, con condicionales anidados.
input s1, s0;
input i3, i2, i1, i0;
assign y = s1? (s0 ? i3: i2) : (s0 ? i1: i0);
```

El operador if-then-else, controla la ejecución de funciones combinacionales alternativas. Se comporta de manera similar al operador condicional. Se sintetiza empleando multiplexores.

```
always @*
if (Control)
  Z = A & B;
else
  Z = A | B;
```

Si se utiliza un if sin un else, se sintetiza con latchs.

```
reg Y;
always @(sel or A)
 if (sel) Y = A;  //si sel=0  Y conserva su valor anterior.
```

A5.11.3. Formación y creación de buses.

La creación de buses se logra con asignaciones continuas con índices constantes.

```
reg [31:0] InstReg; // Registro de instrucción de 31 bits
wire [5:0] OpCod; // Los primeros 6 bits (los más significativos).
wire [4:0] Rs;      // Especificación bus Rs con 5 bits
wire [4:0] Rt;      // Especificación bus Rt con 5 bits
wire [15:0] Inm16;      // Especificación bus Inm16 con 16 bits

      assign OpCod = InstReg[31:26];  // El código de operación
      assign Rs = InstReg[25:21];
      assign Rt = InstReg[20:16];
      assign Inm16 = InstReg[15:0];
```

La operación de concatenación permite reunir una serie de señales, o partes de un conjunto de cables o buses, en un nuevo bus. En el ejemplo siguiente se extiende con signo el bus Inm16 de 16 bits, a uno de 32, el cual forma un operando de 32 bits, denominado Op32. La operación extiende con signo y sin signo, de acuerdo al valor de signed.

```
reg [31:0] Op32;
always @(*)
      if(signed)  Op32 = {Inm16[15] ? 16'hffff : 16'h0, Inm16};
      else Op32 = { 16'h0, Inm16 };
```

A5.11.4. Inferencia de flip-flops.

El procedimiento de síntesis infiere la presencia de flip-flops si encuentra bloques always que tengan posedge o negedge en la lista de sensibilidad.

En el siguiente ejemplo, se detecta un bloque lógico combinacional y un flip-flop d.

```
wire a, b, clk;
```

```
reg q;
  always @(posedge clk)
     q <= a & b;
```

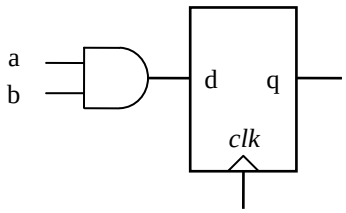

Figura A5.50.

Si en la biblioteca de funciones se dispone de un flip-flop D, con reset sincrónico, el proceso de síntesis podría implementar la lógica empleando la señal de reset, lo cual se muestra en la Figura A5.51.

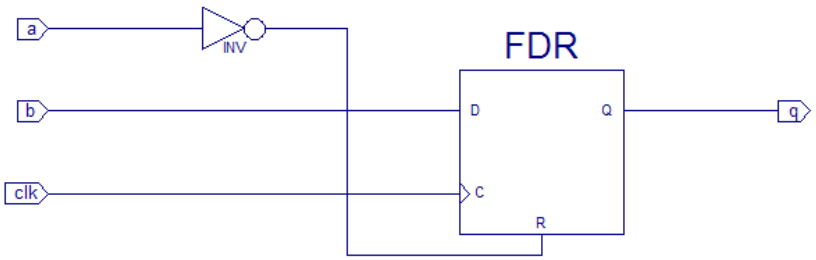

Figura A5.51. Síntesis ISE

También se detecta la presencia de múltiples flip-flops, cuando se emplea notación de vectores o buses:

```
wire clk;
wire [1:0] d;
reg [1:0] q;
   always @(posedge clk)
       q <= d;
```

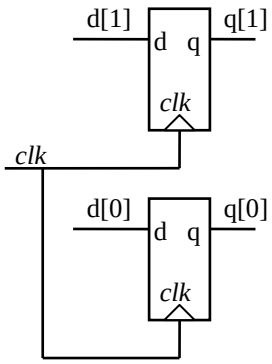

Figura A5.52.

A5.11.5. Tablas de búsqueda. Look-up tables.

Una tabla de búsqueda puede ser tratada conceptualmente como una ROM.

Se analiza la descripción Verilog de una tabla de búsqueda de dos entradas.
Esto implica una ROM 4x1. Es decir cuatro direcciones, con celdas de 1 bit.

Con dirección dada por: I1, I0, el espacio podría visualizarse según:

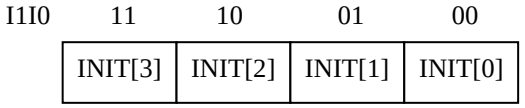

Figura A5.53.

Un esquema que muestra el bus de direcciones (I1, I0) y el bus de salida (O), se muestra en la Figura A5.54.

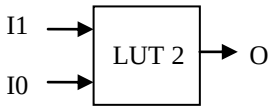

Figura A5.54.

Se emplea el parameter INIT para inicializar el contenido de la ROM. Cuando se crea una instancia del modelo, puede iniciarse con los datos específicos.

A5.11.5.1. Modelos de simulación de LUTs.

Se analiza primero una descripción por primitivos y luego por una función equivalente. Ambos modelos corresponden a las suministradas en las bibliotecas para simulación Verilog de Xilinx.

```
`timescale 1 ps/1 ps
//2-input Look-Up-Table. Definida como primitiva
module LUT2p (output O, input I0, I1);
  parameter INIT = 4'h0;  //especifica 4 bits con el contenido de la LUT
  wire O;
  x_lut2_mux4 (O, INIT[3], INIT[2], INIT[1], INIT[0], I1, I0);//  instancia primitivo
endmodule

primitive x_lut2_mux4 (output o, input d3, d2, d1, d0, s1, s0);

  table
  // d3  d2 d1 d0 s1 s0 : o;
      ?   ?   ?   1   0   0  : 1;  //lee de la rom si están definidas las direcciones
      ?   ?   ?   0   0   0  : 0;
      ?   ?   1   ?   0   1  : 1;
```

```
    ?  ?  0  ?  0  1 : 0;
    ?  1  ?  ?  1  0 : 1;
    ?  0  ?  ?  1  0 : 0;
    1  ?  ?  ?  1  1 : 1;
    0  ?  ?  ?  1  1 : 0;

    ?  ?  0  0  0  x : 0;  //si s1==0 y d0==d1 lee d0 (o d1)
    ?  ?  1  1  0  x : 1;
    0  0  ?  ?  1  x : 0; //si s1==1 y d3==d2 lee d2 (o d3)
    1  1  ?  ?  1  x : 1;

    ?  0  ?  0  x  0 : 0; //s0==0 y d2==d0
    ?  1  ?  1  x  0 : 1;
    0  ?  0  ?  x  1 : 0; //s0==1 y d3==d1
    1  ?  1  ?  x  1 : 1;

    0  0  0  0  x  x : 0;  //los cuatro contenidos iguales.
    1  1  1  1  x  x : 1;
  endtable
endprimitive
```

La definición mediante procesos implementa funcionalidades similares a las de la descripción del primitivo.

```
`timescale  100 ps / 10 ps
//2-input Look-Up-Table. Definida como función
module LUT2f (output O, input I0, I1);

   parameter INIT = 4'hA;  //1010  minimizada equivale a O=I0
   reg  O;
   wire [1:0] s;

   assign s = {I1, I0};

   always @(s)
     if ((s[1]^s[0] ==1) || (s[1]^s[0] ==0))   //si están especificadas las direcciones
        O = INIT[s];                          //lee la rom
      else if ((INIT[0] == INIT[1]) && (INIT[2] == INIT[3]) && (INIT[0] == INIT[2]))
        O = INIT[0];  //si los contenidos de las cuatro celdas son iguales
      else if ((s[1] == 0) && (INIT[0] == INIT[1]))
        O = INIT[0];
      else if ((s[1] == 1) && (INIT[2] == INIT[3]))
        O = INIT[2];
      else if ((s[0] == 0) && (INIT[0] == INIT[2]))
        O = INIT[0];
      else if ((s[0] == 1) && (INIT[1] == INIT[3]))
        O = INIT[1];
```

```
        else
          O = 1'bx;
Endmodule
```

Modelos de LUTs de mayores capacidades son implementadas como instancias de LUT2. Por ejemplo una LUT 4 se construye con 4 LUT2 y 3 MUX2a1.

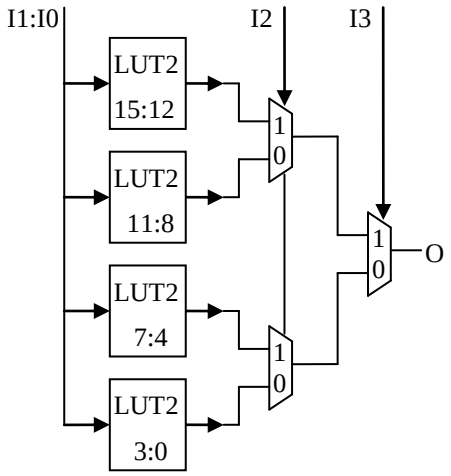

Figura A5.55. LUT 4 en base a LUTs 2 y muxs.

A5.11.5.2. Diseños en base a LUTs 4.

Si el módulo básico de una FPGA es una LUT4, se analiza cuántas de éstas se requieren para implementar una LUT M, esto con el objetivo de tener una medida del crecimiento de empleo de LUTs 4 para implementar funciones complejas.

Una LUT5 se construye con 2 LUT4 y 1 MUX2a1 = 2^{5-4}LUT4 + (2^{5-4}-1) MUX2a1
1 nivel de mux

Una LUT6 se construye con 2 LUT5 y 1 MUX2a1 = 4 LUT4 + 3 MUX2a1
2 niveles de muxs = 2^{6-4}LUT4+ (2^{6-4}-1) MUX2a1
Una LUT7 se construye con 2LUT6 y 1 MUX2a1 = 8 LUT4 y 7 MUX2a1
3 niveles de muxs = 2^{7-4} LUT4+ (2^{7-4}-1) MUX2a1
Una LUT8 se construye con 2LUT7 y 1 MUX2a1 = 16 LUT4 y 15 MUX2a1
4 niveles de muxs = 2^{8-4} LUT4+ (2^{8-4}-1) MUX2a1

Entonces una LUTM se construye con 2^{M-4} LUT4+ (2^{M-4}-1) MUX2a1.
(M-4) niveles de muxs

Puede encontrarse una cota superior de este número, mediante:
2^{M-4} LUT4+ (2^{M-4}-1) MUX2a1 < $2*2^{M-4}$ LUT4 = 2^{M-3} LUT4

Entonces el número n de LUT 4 necesarias para implementar una LUT M es:

$$n < 2^{M-3}$$

Por otro lado, como una LUT M es una memoria de 2^M bits, pueden implementarse 2^{2^M} funciones diferentes. Con una LUT4 pueden implementarse $2^{2^4} = 2^{16}$ funciones diferentes de 4 variables. Con 2 LUT 4, el número de funciones es: $2^{32} = (2^{16})(2^{16}) = (2^{2^4})^2$. Entonces puede plantearse el número necesario de LUT 4 como aquel que implementa igual o superior número de funciones que la LUT M.

$$(2^{2^4})^n >= 2^{2^M}$$

Sacando logaritmos en base 2, en ambos lados, se obtiene:

$$n \log_2(2^{2^4}) \geq \log_2(2^{2^M})$$
$$n 2^4 \log_2(2) \geq 2^M \log_2(2)$$
$$n \geq 2^{M-4}$$

Finalmente, juntado los dos resultados, se obtiene:

$$2^{M-3} > n \geq 2^{M-4}$$

Lo cual muestra un crecimiento exponencial.

En una FPGA el área dedicada a la interconexión de LUTs, es mucho mayor que el área dedicada a las LUTs. Intentos de minimizar el área de interconexión y los retardos han conducido a la elección de LUTs 4 como el módulo básico.

A5.11.5.3. Comparación con uso de memorias.

Una memoria estática de 64Kx1 equivale a 4K de LUTs 4, ya que cada LUT 4 almacena 16 bits. Sin embargo, disponer de LUTs con capacidad de interconectarlas, es muchísimo más eficiente que emplear memorias.

La capacidad de interconectar LUTs permite que la estructura de las aplicaciones sea eficientemente mapeada tecnológicamente.

Algunos diseños de funciones que en su estructura presentan cierta regularidad muestran que es preferible emplear LUTs que una memoria.

Ejemplo A5.47. Generador de paridad

Un generador de paridad para palabras de 16 bits, requiere usar la memoria anterior en forma completa, ya que el bus de direcciones de 16 bits se ocupa en su totalidad. Sin embargo sólo se requieren 5 LUTs 4 de una FPGA para implementar el mismo diseño.

El diseño en Verilog del generador de paridad para palabras de 16 bits, se implementa usando 5 LUTs 4:

```
module paridad (y, in);
parameter size = 16;
input [size-1:0] in;
```

```
output y;
   assign y = ^in;   //calcula paridad. Operador de reducción.
endmodule
```

La Figura A5.56 muestra la síntesis realizada por ISE. Mostrando 5 bloques iguales para la implementación. Esta modularidad de la estructura se planteó antes como regularidad.

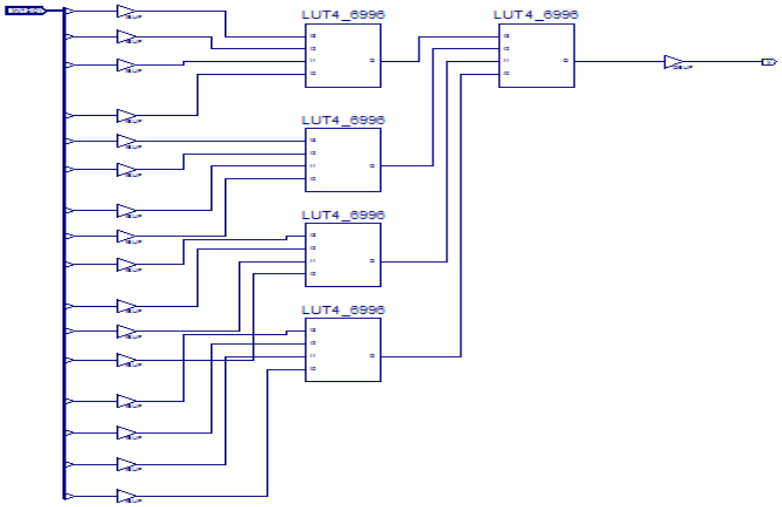

Figura A5.56.

A5.11.6. Síntesis de bloques combinacionales, basados en LUTs.

A5.11.6.1. LUT4

Una LUT 4 puede ser empleada para generar cualquier función de cuatro variables, en este caso se comporta como una ROM 16x1. La lectura de los bits individuales se logra con un multiplexor de 16 a 1. La Figura A5.57, muestra una arquitectura interna posible de la LUT, en la cual los elementos de almacenamiento se muestran como latchs; esta elección implementa la celda de memoria con menos transistores que empleando un flip-flop disparado por cantos, y no emplea un reloj. En el proceso de configuración inicial, usando un decodificador se van colocando en uno o en cero los diferentes latchs; los cuales mantienen sus valores, mientras se tenga polarización aplicada, por esto se dice que se comporta como una ROM. Una vez grabada la LUT con la tabla de verdad correspondiente, los bits individuales se leen mediante el multiplexor que alimenta la salida.

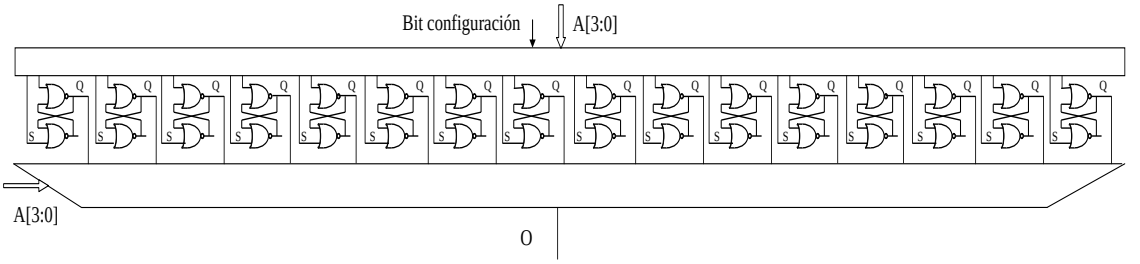

Figura A5.57

Un esquema simplificado mostrando las entradas y la salida se ilustra en la Figura A5.58.

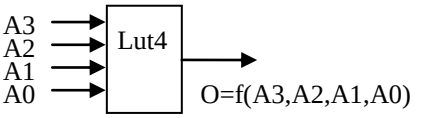

Figura A5.58

A5.11.6.2. Dos LUT4.

Función de cinco variables.

Si dentro del bloque, donde están las LUTS, se dispone de un multiplexor de dos vías a una, puede lograrse, combinando dos LUTs como se indica en la Figura A5.59, cualquier función de cinco entradas. Se emplea el teorema de expansión de Shannon.

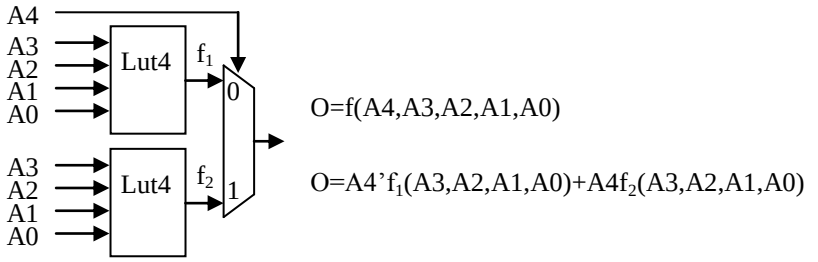

Figura A5.59

Mux 2 a 1 ancho 2.

La forma de ir componiendo funciones más complejas, en base a los componentes existentes se denomina mapeo tecnológico. Por ejemplo en una LUT 4, puede programarse una función de tres entradas: A0A2+A0'A1, que se comporta como una LUT 3. Si se forma una tabla de verdad con las variables ordenadas según A2A1A0, con valores binarios descendentes, los 8 bits de la función leídos en hexadecimal forman el patrón E4 (11100100).

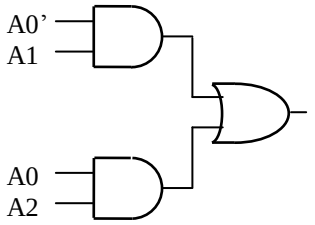

Figura A5.60. LUT 3, iniciada con E4.

Uniendo dos de estas estructuras se logra un multiplexor de 2 vías a 1 de 2 bits de ancho. El diagrama RTL del mux, se ilustra a la derecha de la Figura A5.61.

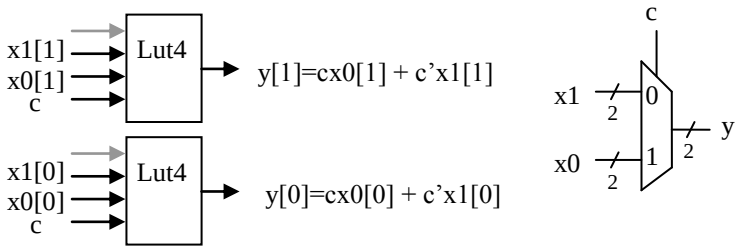

Figura A5.61. Mux 2 a 1, 2 bits de ancho.

```
module mux2a1w2 (out, c, x1, x0);
  output [1:0] out;          // mux output de 2 bits
  input  c;
  input [1:0] x1, x0;   // 2 mux inputs de 2 bits cada una
  wire c;
  wire [1:0] x1,x0;
  reg [1:0] out;

always @(*)
   if (c==0) out=x0;  else if (c==1) out=x1;
endmodule
```

La expansión del ancho del bus se logra agregando tantas LUT3 (E4), como se requiera. La estructura es repetitiva, lo que implica una regularidad en la arquitectura.

Un multiplexor de 4 a 1 ancho 1.

Empleando la configuración LUT 3 (E4) se tienen:

$$y1= c[0]x3 + c[0]'x2$$
$$y0=c[0]x1 + c[0]'x0$$

Y con un mux adicional de 2 vías a 1, controlado por c[1], se obtiene la ecuación del mux 4 a 1, de ancho 1 bit:

$$out=c[1](c[0]x3 + c[0]'x2) + c[1]'(c[0]x1 + c[0]'x0)$$

El diagrama con las interconexiones se muestra en la Figura A5.62. A la derecha se muestra el diagrama RTL del mux 4 a 1 de ancho 1.

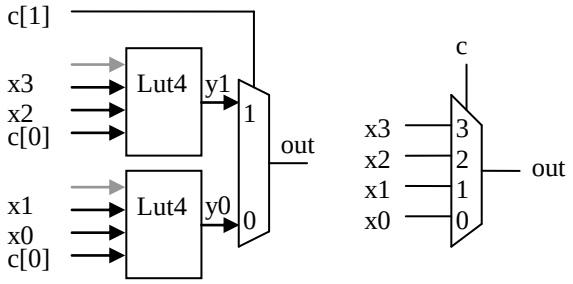

Figura A5.62. Mux 4 a 1, ancho de 1 bit.

```
module mux4a1w1 (out, c, x);
  output out;     // mux output de 1 bit
  input  c;
  input [3:0] x;   // 4 mux inputs de 1 bit cada una
  wire c;
  wire [3:0] x;
    assign out=x[c];
endmodule
```

Funciones particulares de hasta 9 variables.

En el mejor de los casos, con dos LUTs 4 pueden lograrse algunas funciones de nueve entradas. Esto es posible cuando la función tenga la forma que se muestra en la Figura A5.63.

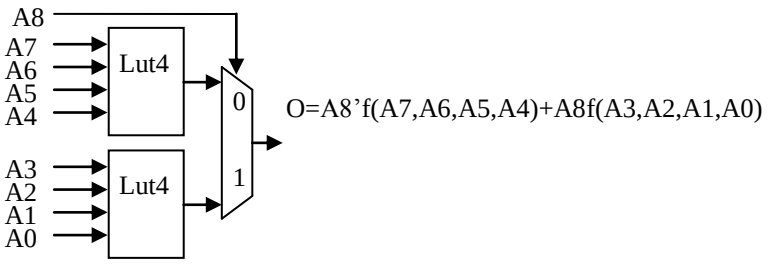

Figura A5.63. Funciones de 9 variables de entrada.

A5.11.6.3. Cuatro LUTs 4

Si existiese otro mux de dos vías a una, disponible en las inmediaciones de las LUTs, la interconexión de 4 LUTS, en la configuración que se indica en la Figura A5.64, permite:

Cualquier función de seis variables (LUT6).

Algunas funciones de hasta 19 entradas.

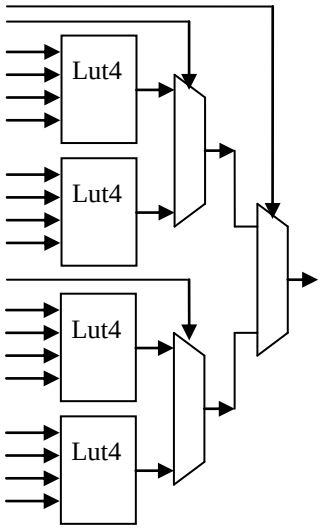

Figura A5.64.

Con la misma configuración puede implementarse un multiplexor 8 a 1.

```
module mux8a1w1 (out, c, x);
  output out;          // mux output de 1 bit
  input  [2:0] c;
  input [7:0] x;   // 8 mux inputs de 1 bits cada una
  wire [2:0] c;
  wire [7:0] x;
  assign out=x[c];
endmodule
```

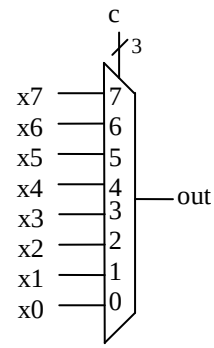

Figura A5.65. Mux 8 a 1, ancho de 1 bit.

Un mux de dos vías a uno de 4 bits de ancho.

```
module mux2a1w4 (out, c, x1,x0);
  output [3:0] out;          // mux output de 4 bits
  input  c;
  input [3:0] x1,x0;   // 2 mux inputs de 4 bits cada una
  wire c;
```

```
  wire [3:0] x1,x0;
  reg [3:0] out;

always @(*)
    if (c==0) out=x0;  else if (c==1) out=x1;
endmodule
```

Las LUTs están en la configuración LUT 3, y el conjunto puede simbolizarse por el diagrama RTL, a la derecha de la Figura A5.66.

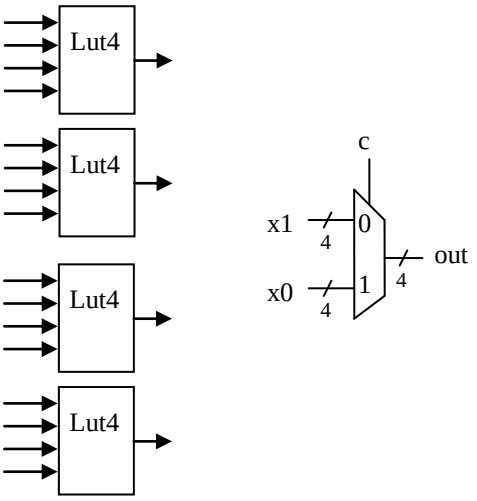

Figura A5.66. Mux 2 a 1, 4 bits de ancho.

A5.11.6.4. 8 LUTs

Si se unen dos bloques similares al de la Figura A5.64, mediante un tercer mux, de dos vías a una, se logran: Un mux de 16 a 1, cualquier función de 7 entradas, algunas funciones de hasta 39 entradas. Con las LUTs, sin multiplexores, se logra un mux de 2 vías a 1 de ancho 8 bits.

A5.11.6.5. 16 LUTs

Un cuarto nivel de multiplexores, permite el diseño de un multiplexor de 32 bits a 1, de cualquier función de 8 variables, y algunas hasta de 79 entradas. Con las LUTs, sin multiplexores, se logra un mux de 2 vías a 1 de ancho 16 bits.

A5.11.6.6. Operadores al bit.

La síntesis de operadores al bit, se realiza empleando una LUT 4 para cada pareja de los operandos de entrada. Por ejemplo, la síntesis de: assign z = x & y; si los operandos son de n bits requiere n LUTs 4.

La síntesis de operadores de reducción, se efectúa en niveles. La síntesis de: assign z= &x; donde el bus x tiene más de 7 y menos de 13 bits, debería sintetizarse según:

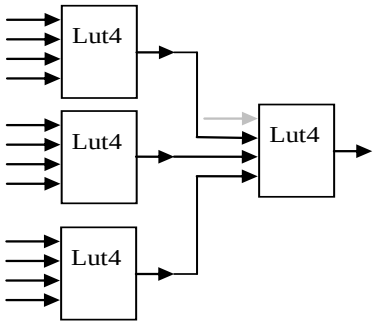

Figura A5.66.a Reducción en niveles.

A5.11.6.7. Potencias.

Una potencia de dos elevada a un entero x, por ejemplo 8^x, puede interpretarse como 2^{3x}. A su vez puede implementarse como $2^x 2^x 2^x$. Pero 2^x es (1<<x). Entonces: assign z= 8**x; se logra con el diagrama RTL:

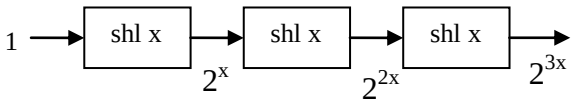

Figura A5.66.b Corrimientos a la izquierda en x bits.

Sin embargo, si x es un bus de ancho 2, el resultado estará formado por solamente un 1, en posiciones 0, 3, 6, 9, la síntesis puede efectuarse sin necesidad de registros, en términos solamente combinacionales.

x	2^{3x}
0	1
1	1000
2	1000000
3	1000000000

Figura A5.66.c Posiciones de los unos en 2^{3x}.

Las posiciones con ceros se logran con conexiones a tierra. El uno en la posición cero se obtiene con: z[0]=x[1]' x[0]'. El uno en posición 3, z[3]= x[1]' x[0].
Y también: z[6]=x[1] x[0]'; z[9]= x[1] x[0]. Lo anterior se puede implementar con 4 LUTs 4.

La síntesis de: assign z= 2**x; se logra con una unidad de desplazamiento a la izquierda. Observando que el resultado z, sólo tiene un uno de acuerdo al valor de x, la síntesis puede efectuarse en forma combinacional. Con x de ancho 4, se tiene que el bit de z que debe estar alto se logra decodificando x. Por ejemplo: z[3]= x[3]'x[2]'x[1]x[0].

x	2^{x}
0	1
1	10
2	100
15	1000000000000000

Figura A5.66.d Posiciones de los unos en 2^{x}.

La implementación anterior requiere 16 LUTs 4, una para cada decodificador.

A5.11.6.8. División entera y corrimiento a la derecha.

Las divisiones por una potencia de dos, ya sea en la forma: assign z=x/8; o bien: assign z=x>>3; se realizan mediante conexiones solamente. Esto debido a que implican un corrimiento a la derecha en tres posiciones, en el ejemplo planteado.
Si x es de 5 bits, se requiere un espacio de 2 bits para z. El diseño se obtiene con z[1]=x[4], y z[0]=x[3], y no se requieren los datos x[2:0].

A5.11.6.9. Operación módulo.

La operación módulo sólo puede sintetizarse si el segundo operando es una potencia de dos. En este caso la implementación se logra a través de conexiones solamente.

Por ejemplo: assign z = x % 8; con x de ancho 5 bits, requiere 3 bits para el resultado z. Y no se emplean los dos bits más significativos x[4:3]. Las conexiones: z[2]=x[2], z[1]=x[1], z[0]=x[0].

A5.11.6.10. Corrimiento a la izquierda.

Los corrimientos a la izquierda por una constante son implementados por conexiones.

Los corrimientos a la izquierda en una variable son implementados con lógica combinacional si se emplea: assign z= x << y. Si x es de 5 bits e y de 4 bits, el resultado z requiere 20 bits. La implementación ISE emplea 36 LUTs 4.

A5.11.6.11. Corrimientos aritméticos.

Se dispone de corrimientos aritméticos con los operadores >>> y <<<. Debe notarse que el símbolo <<< es equivalente con el corrimiento lógico << a la izquierda.

A5.11.6.12. Operadores lógicos.

Antes se planteó, para operandos de largo 4, que la red combinacional:

```
assign c = (a == 2) && (b == 3);
```

También puede plantearse según:

```
assign c =  (~a[3])&(~a[2])&(a[1])&(~a[0])&(~b[3])&(~b[2])&(b[1])&(b[0]);
```

Esta última relación muestra que se requieren 3 LUTs 4; dos para generar las decodificaciones, ya que los operandos son de 4 bits, y una adicional para el &&, de las salidas de los decodificadores.

A5.11.6.13. Salidas registradas.

Si un generador de paridad, se coloca dentro de un proceso always controlado por un reloj, la salida combinacional se conectará a un flip-flop.

```
module paridad (y, in, clk);
parameter size = 16;
input [size-1:0] in;
input clk;
output y;
reg y;
   always @(posedge clk)
     begin
       y = ^in;     //salida combinacional conectada a flip-flop D.
     end
endmodule
```

El siguiente diagrama RTL, muestra el diseño anterior.

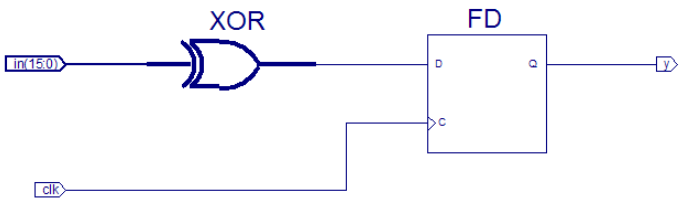

Figura A5.66.e Salida registrada.

A11.7. Bloques secuenciales.

A11.7.1. RAM

Si la estructura interna de la LUT está basada en flip-flops, en lugar de latchs, puede emplearse como memoria RAM.

Una LUT 4 equivale a una RAM de 16x1. Con escritura sincrónica y lectura asincrónica. Si se emplea el flip-flop asociado a la LUT, puede emplearse lectura sincrónica.

Un esquema que muestra una estructura posible para una LUT que funcione como RAM, se muestra en la Figura A5.67. Se dispone de un reloj común, y cada flip-flop tiene una habilitación de escritura (WE) que está conectada a un decodificador de las cuatro líneas de entrada; lo cual permite escribir en uno de los flip-flops, en forma sincrónica. El dato de entrada es común a todos los flip-flops. Las salidas de éstos están conectadas a un multiplexor de 16 vías a una; del tal modo que se puede efectuar una lectura asincrónica (sin reloj), del bit de la memoria direccionado por las líneas de entrada, que constituyen el bus de dirección de la RAM.

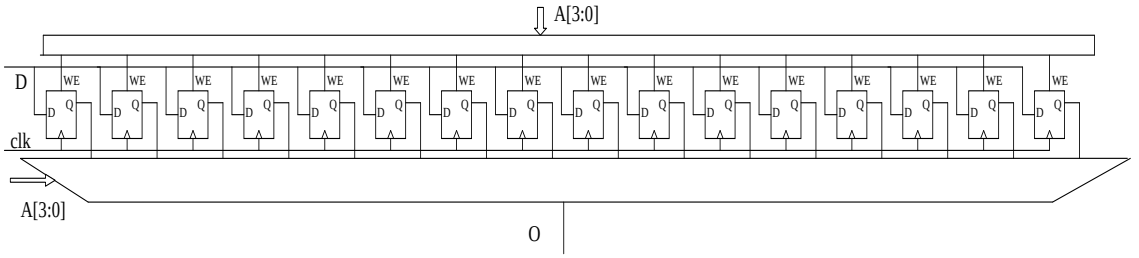

Figura A5.67. RAM 16x1

Un esquema simplificado de la memoria se muestra en la Figura A5.68.

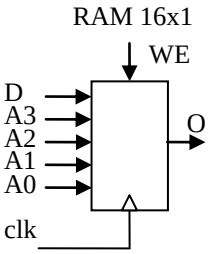

Figura A5.68.

Si se emplean dos LUTs como memoria RAM, pueden configurarse una RAM32x1 empleando un mux adicional; o una RAM16x2 empleándolas en paralelo.

También empleando dos LUTs, puede implementarse una RAM 16x1 con dos puertas duales. Se escribe en ambas memorias los mismos datos; una de las RAMs es read/write, la otra de sólo lectura controlada por la dirección adicional DPRA. Se pueden leer dos direcciones simultáneamente, en los dos puertos de salida SPO y DPO.

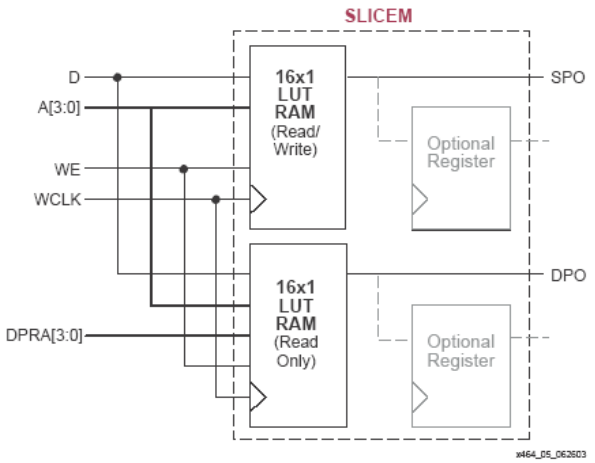

Figure 5: **RAM16X1D Placement**

A11.7.2. Registro de desplazamiento.

Una LUT, en base a flip-flops, puede emplearse como registro de 16 bits de desplazamiento lógico a la derecha, con entrada y salida serial, y con salida direccionable; el reloj es común. Si se deja fija la entrada al multiplexor, puede programarse un registro de largo variable.

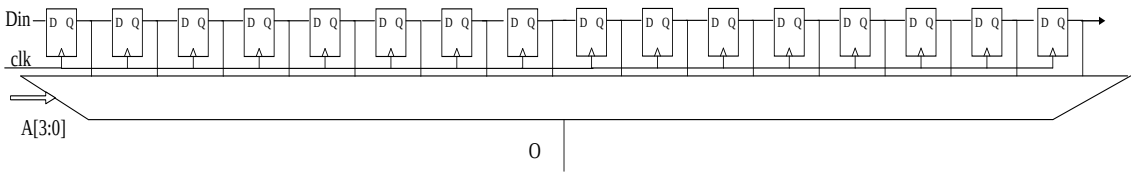

Figura A5.69. Registro de desplazamiento.

El siguiente código Verilog es sintetizable como un registro de desplazamiento de 16 bits.

```
always @ (posedge clk)
  begin
    srl <= {srl[14:0], Din}; //desplaza a la derecha.
  end
always @(srl)
  begin
    Q <= srl[15];  //registra la salida.
  end
```

El siguiente es un un modelo para simulación de un registro de desplazamiento.

```
`timescale  1 ps / 1 ps
module SRL16 (Q, A0, A1, A2, A3, CLK, D);
    parameter INIT = 16'h0000;
    output Q;
    input  A0, A1, A2, A3, CLK, D;
    reg  [15:0] data;
```

```
   assign Q = data[{A3, A2, A1, A0}];   //mux de salida

   initial
   begin
        assign  data = INIT;
        while (CLK === 1'b1 || CLK===1'bX)
          #10;
        deassign data;
   end

   always @(posedge CLK)
   begin
     {data[15:0]} <= #100 {data[14:0], D};
   end

endmodule
```

Un diagrama simplificado del registro anterior:

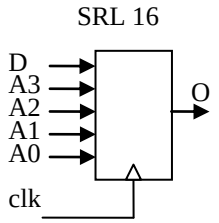

Figura A5.70

A11.8. Síntesis de operaciones aritméticas en base a sumadores y muxs.

Se tienen dos buses de entrada de 8 bits cada uno, y se desea generar una salida cuando la suma de los operandos de entrada sea menor que 128 decimal.

El siguiente módulo describe con precisión una operación aritmética sencilla, mostrando la capacidad de descripción empleando lenguajes. Sin embargo el problema no es sencillo de sintetizar con métodos de papel y lápiz, ya que se tienen 16 entradas, lo cual implica una tabla de verdad con 65536 entradas, o mapas de Karnaugh de 16 variables.

Se ha empleado el macro `define para parametrizar el módulo. Su ventaja, en lugar de usar parameter, es que la definición puede ser empleada en varios módulos.

```
`define N 7
`define MAX 255
module aritmetica(x, a, b);
   input [`N:0] a, b;
   output x;
        assign x= a+b < 128;
endmodule
```

El desarrollar un banco de prueba, para generar estímulos, básicamente consiste en generar todas las combinaciones posibles de las entradas; esto se logra con dos for anidados.

Puede notarse que el diseño del módulo test es muy similar a un programa escrito en C. Como el programa debe generar datos, posiblemente en la pantalla con fines de depuración y también en archivos para generar formas de ondas, gran parte del código es para lograr este objetivo.

Es en estos programas que se debe usar comandos que generen eventos, para que la simulación pueda ir avanzando.

```
module test(a, b, result);
output [`N:0] a, b;
input  result;
reg [`N:0] a, b;
integer j, k;
integer errores, vectores;

initial
begin  $dumpfile("mul.vcd");   $dumpvars(0,a,b,result); end

initial
  begin
    //$monitor("a=%d b=%d result=%b", a, b, result, $time);    //para depuración
    a=0; b=0;
    errores=0; vectores=0;
     for(j=0;j<`MAX;j=j+1)
       begin
         #10 a<=j;
         for(k=0;k<`MAX;k=k+1)
           begin
             #10 b<=k;
             vectores=vectores+1;
             if(result!=((a+b)<128) )
               begin
                 errores=errores+1;
                 $display("Error: a=%d b=%d result=%d Correcto=%d", a, b, result,(a+b)<128);
               end
           end
       end

    $display("Se probaron %d vectores. Errores=%d", vectores, errores);
       #10 $finish;
  end

endmodule

module BancoDePrueba;
```

```
wire [`N:0] a, b;
wire x;
 aritmetica a1 (x, a, b);
 test       t1 (a, b, x); //las salidas de test alimentan al módulo aritmético.
endmodule
```

Debe notarse que los módulos para la prueba tienen bastantes más líneas que el módulo que se desea probar.

Con la ayuda de este test, se visualiza que hay problemas con el diseño, ya que se producen demasiados errores. Si se analizan se verá que la suma se está realizando en 8 bits, de este modo si los operandos son mayores que 128, se tendrá rebalse del sumador. Este puede corregirse si se especifica un sumador que entregue el resultado en 9 bis, de esta manera la suma máxima: 255+255=510 no produce rebalse; es decir el máximo valor de la suma se puede representar en 9 bits, ya que 510= 1'b111111110. Esto asumiendo números sin signo.

Nos aseguramos que la suma se efectúe en 9 bits, definiendo el bus temp, esto implica cambios en el módulo:

```
module aritmetica(x, a, b);
   input [`N:0] a, b;
   output x;
      wire [9:0] temp;
      assign temp=a+b;  //el resultado de la suma se calcula con 9 bits
      assign x= temp < 128;
endmodule
```

La herramienta de síntesis podría haber reconocido lo anterior, y emplear dos módulos de sus bibliotecas: un sumador de 9 bits, y un circuito comparador slt (por set less than) que genere la señal de salida.

```
module aritmeticaSint(x, a,b);
   input [`N:0] a, b;
   output x;
   wire [9:0] temp;
      sumador  s1 (temp, a, b);   //operandos de 8 bits, suma con reserva de salida.
      slt   sl1 (x, temp, 9'd128);   //operandos de 9 bits
endmodule
```

También este camino podría seguir un diseñador, permitiendo descomponer el diseño, en dos módulos más simples. Según muestra el esquemático de la Figura A5.71.

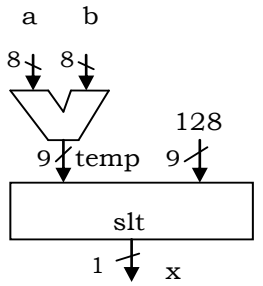

Figura A5.71.

Ahora deben desarrollarse dos nuevos módulos, los cuales pueden simularse por separado hasta verificar su funcionalidad. El sumador es posible que sea mejor sintetizado por la herramienta CAD, ya que los sumadores son un recurso fundamental en sistemas digitales, y se ha trabajado suficiente es su desarrollo para optimizar la velocidad de éstos.

El comparador también debe estar en las bibliotecas del sintetizador; pero podría ser bastante más económico en espacio, el realizarlo como un módulo combinacional, ya que en este caso sólo se tienen 9 entradas, en lugar de 18. Además el número 128=1'b010000000, es una potencia de dos, y se podría diseñar el módulo combinacional, considerando que los dos primeros bits de temp deben ser ceros, para que éste sea menor que 128. Emplearemos esta idea, que difícilmente sería realizada por un autómata de síntesis digital, definiendo un módulo combinacional slt128 de sólo dos entradas, los dos bits más significativos del bus temp.

```
module aritmeticaSint2(x, a, b);
   input [`N:0] a, b;
   output x;
   wire [9:0] temp;
      assign temp=a+b;   //operandos de 8 bits, suma con reserva de salida en 9 bits.
      slt128  s1 (x, temp[8:7]);
endmodule
```

El esquemático con la arquitectura de aritmeticaSint2, se muestra en la Figura A5.72.

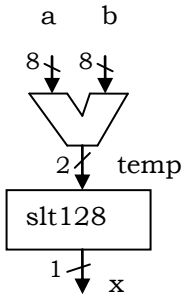

Figura A5.72.

Asumiendo que el sintetizador es eficiente, se le podría delegar la responsabilidad de generar slt128, mediante una descripción conceptual de la operación: los dos bit de entrada deben ser ceros.

```
module slt128(output x, input [1:0] val);
  assign x= ( (val[1] | val[0]) ==0);
endmodule
```

Evaluando el costo de este módulo, puede no ser necesario seguir descendiendo en el nivel de detalle. Pero si se desea asegurar un diseño mínimo, puede escribirse a nivel de compuertas, según:

```
module slt128(output x, input [1:0] val);
  and(x, ~val[1], ~val[0]);
endmodule
```

A5.12. Conclusiones.

Conviene conocer el diseño en bajo nivel de los subsistemas típicamente usados, de esta manera se conocerá su funcionalidad, sus alcances, la forma de generalizarlos, y su costo. Luego las descripciones de la arquitectura pueden realizarse basada en esos módulos.

Un curso básico en sistemas digitales debe profundizar en el **diseño de los bloques básicos**, y también capacitar en **descomponer un problema en términos de esos bloques**. Esto garantiza plantear arquitecturas con partes que serán sintetizadas eficientemente.

La Figura A5.73, muestra el proceso de síntesis como un nivel intermedio, entre las descripciones de arquitecturas mediante módulos sintetizables (top-down), y el desarrollo de módulo básicos implementados mediante compuertas y flip-flops (bottom-up). Los lenguajes de descripción de hardware, como Verilog, permiten las descripciones y simulaciones en los diferentes niveles, dando homogeneidad al proceso de diseño.

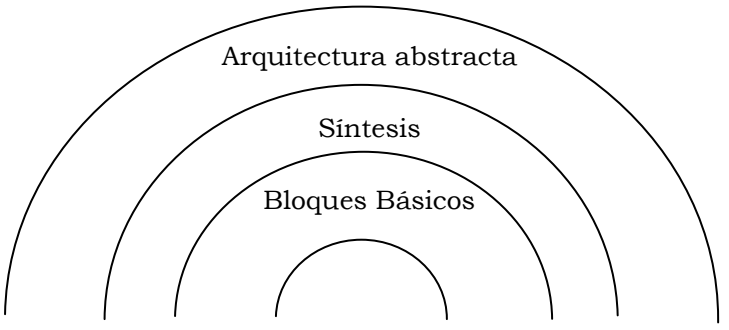

Figura A5.73.

Referencias.

Tutoriales y conceptos sobre Verilog.

Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!. Clifford E. Cummings. SNUG 2000.
"full_case parallel_case", the Evil Twins of Verilog Synthesis. Clifford E. Cummings. SNUG 1999.

http://www.see.ed.ac.uk/~gerard/Teach/Verilog/manual/Intro/gatetypes.html

http://www.doulos.com/knowhow/verilog_designers_guide/

Cursos universitarios sobre Verilog.

El profesor David M. Koppelman, de Louisiana State University, Baton Rouge, tiene excelentes materiales para aprender Verilog, aplicado a sistemas digitales y estructuras de computadores en: http://www.ece.lsu.edu/ee3755/ln.html.

El profesor Weidong Kuang, The University of Texas, desarrolló un curso básico de sistemas digitales en: http://www.engr.panam.edu/~kuangw/courses/ELEE4303/index.html.

Software libre.

Herramienta de simulación y síntesis.
Icarus Verilog
http://www.icarus.com/eda/verilog/index.html

Visor de formas de ondas.
GTKWave visor de archivos VCD/EVCD. Anthony J Bybell .
http://www.geocities.com/SiliconValley/Campus/3216/GTKWave/gtkwave-win32.html

Desgraciadamente las referencias a la red son volátiles y no permanentes.

Índice general.

Índice de Figuras.