3 - DISEÑO RTL

Un diseño a nivel funcional o de comportamiento (*behavioral description*) permite describir circuitos digitales atendiendo únicamente a las relaciones existentes entre las entradas y salidas del circuito, pero sin hacer referencia a la estructura real o hardware del que se compone el circuito.

Existen dos tipos de descripciones a nivel de comportamiento en Verilog:

- Descripción RTL (r*egister transfer level*): Es un nivel de abstracción mayor que el diseño estructural. Trabaja sobre variables tipo net, es decir, para implementar lógica combinacional. Utiliza la “asignación continua” (assign continuous assignment) para crear las relaciones lógicas existentes entre las entradas y las salidas. Esta descripción es siempre sintetizable, es decir, los programas de síntesis permiten crear los esquemáticos equivalente.

- Descripción basada en algoritmos: Este nivel de abstracción es mayor que el RTL. Trabaja sobre variables de tipo “reg”. Puede ser utilizado para describir circuitos combinacionales y circuitos secuenciales. Utiliza principalmente las construcciones “initial” y “always” como veremos más adelante. Ejecuta instrucciones de control de flujo comunes a muchos lenguajes de programación de alto nivel. Esta descripción no siempre es sintetizable, aún cuando se pueda simular el comportamiento especificado sin ningún problema.

Ahora estudiaremos la descripción RTL de circuitos digitales. Comenzaremos lo primero de todo estudiando cómo se representan los números en Verilog, ya que será necesario más adelante.

REPRESENTACIÓN NÚMERICA EN VERILOG

Verilog permite representar información numérica utilizando diferentes códigos. Por defecto los valores aparecen escritos en código decimal, aunque también se puede utilizar el código binario, el código octal y el código hexadecimal.
Por defecto se utilizan 32 bits para cada valor, pero es interesante indicar el número exacto de bits del que se compone cada variable para, de este modo, economizar hardware a la hora de la síntesis.

El formato de un valor es el siguiente: *nº_bits'base valor* El número de bits y la base son opcionales. La base por defecto es la decimal. La base se especifica de acuerdo a la siguiente tabla:

Índice de la base	**Base**
'b	Binario
'd	Decimal
'h	Hexadecimal
'o	Octal

Se pueden escribir números con signo y también en notación científica. Ejemplos:

Índice de la base	**Decimal equivalente**	**Almacenado**
2'b10	2	10
3'd5	5	101
7'o5	5	0000101
8'ha	10	00001010
8'had	173	10101101

Cuando se quiere asignar a una señal el valor x ó z también se puede hacer, introduciéndola como si fuese un número. Por ejemplo, el valor 1'bz es un bit en alta impedancia.

Cuando hay que extender el número de bits para representar el número, Verilog no extiende el signo, lo que hace es rellenar con 0's las posiciones más significativas, excepto cuando aparece una x o una z en el lugar más significatico. En este caso se extiende con x o con z respectivamente.

CONTINUOUS ASSIGNMENT

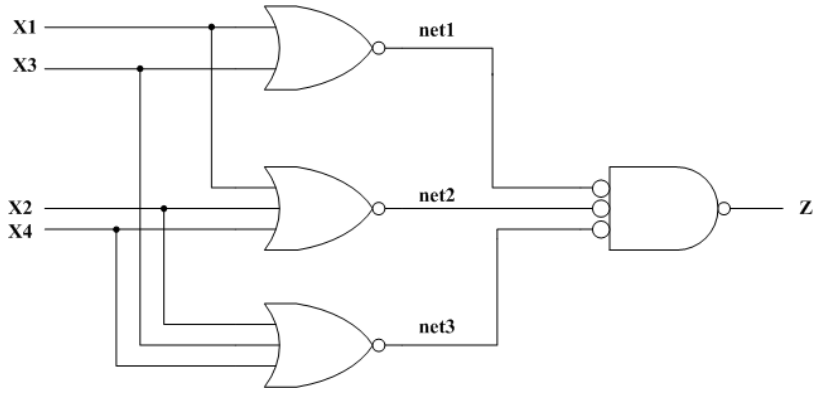
La “asignación continua” (*continuous assignment*) se utiliza para el diseño de lógica combinacional sin utilizar puertas lógicas y sus interconexiones. Se utiliza para asignar valores a una variable net (wire) escalar o vector, pero nunca a una variable registro.

La asignación se realiza comenzando una sentencia con la palabra ***assign***. La sentencia assign continuamente monitoriza la parte derecha de la ecuación. Si una variable cambia de valor, la expresión es evaluada y se asigna el valor al destino (parte izquierda de la ecuación).

OPERADORES BIT A BIT (*BITWISE OPERATORS*)

Operador	nº de operandos	Descripción
~	1	NOT
&	2	AND
|	2	OR
^	2	X-OR
~&	2	NAND
~|	2	NOR
~^ ó ^~	2	X-NOR

Ejemplo:




```
module  circuit (Z, X1, X2, X3, X4);
output  Z;
input   X1, X2, X3;
wire    net1, net2, net3;

assign  net1 = ~(X1 | X3);
assign  net2 = ~(X1 | X2 | X4);
assign  net3 = ~(X2 | X3 | X4);

assign Z = ~((~net1) & (~net2) & (~net3));

endmodule
```

También se pueden aplicar estos operadores a vectores en lugar de escalares. El resultado es que se aplican bit a bit a cada elemento de los vectores. Por ejemplo, si a= 010101 y b = 001100, entonces:

~a = 101010

a & b = 000100

a | b = 011101

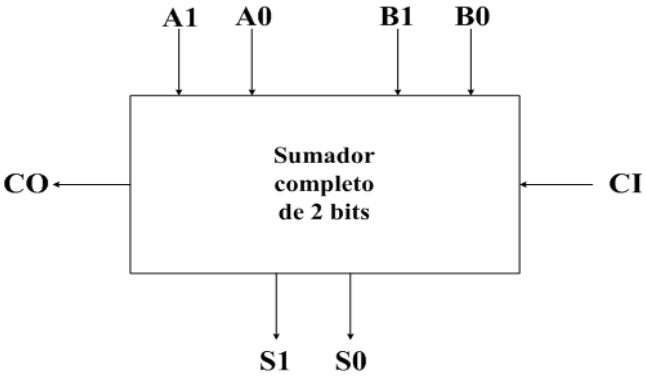
a ^ b = 011001

OPERADORES ARITMÉTICOS

Operador	nº de operandos	Descripción
+	2	Suma
-	2	Resta
*	2	Multiplicación
/	2	Cociente (entero)
%	2	Resto de división

Los operandos siempre se suponen como números positivos. En caso de que el resultado sea negativo, se almacenará en complemento a 2, pero se interpretará como una cantidad sin signo cuando se utilice en una expresión.

Ejemplo:




```
module  sumador (CO, S, A, B, CI);

output   CO;
output   [1:0] S;
input    CI;
input    [1:0] A, B;

wire     [2:0] result;

assign  result = A+B+CI;
assign  CO = result[2];
assign  S = result[1:0];

endmodule
```

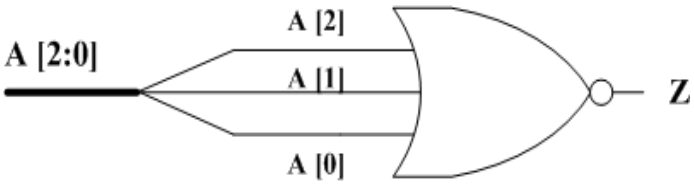
Cuando veamos el operador concatenación, se podrá hacer esto de un modo más sencillo.

OPERADORES DE REDUCCIÓN

Los operadores de reducción son operadores que actúan sobre un único operando, y devuelven un único bit, resultado de la operación lógica sobre los bits del operando.

Operador	nº de operandos	Descripción
&	1	AND reductora
|	1	OR reductora
^	1	X-OR reductora
~&	1	NAND reductora
~|	1	NOR reductora
~^ ó ^~	1	X-NOR reductora

Ejemplo: Supongamos que tenemos un vector de dimensión 3, y queremos comprobar si todos sus bits están a 0 lógico. Un modo sería utilizando una puerta NOR de 3 entradas, tal y como se muestra en la figura:




```
module chk_zero (Z, A);

output  Z;
input  [2:0] A;

assign  Z = ~|A;

endmodule
```

OPERADORES DE DESPLAZAMIENTO

Los operadores de desplazamiento desplazan el contenido de un vector hacia la derecha o izquierda un número determinado de posiciones. Son operaciones de desplazamiento lógico, no aritmético, ya que las posiciones de los bits desplazados se rellenan con 0 (no extiende el signo en desplazamientos a la derecha). Los bits desplazados se pierden. Los operadores son los siguientes:

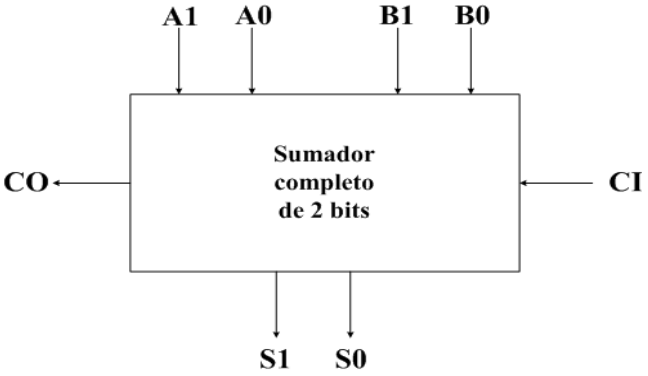
- A << B Se desplaza el vector A a la izquierda el número de posiciones que indique B.
- A >> B Se desplaza el vector A a la derecha el número de posiciones que indique B.

Estos operadores resultan útiles a la hora de multiplicar o dividir un número binario por una potencia de 2.

OPERADOR CONCATENACIÓN

El operador combina dos o más operandos para formar un vector de un mayor número de bits. Se puede utilizar la concatenación tanto a la derecha como a la izquierda de una sentencia de asignación. La concatenación se indica con llaves {} y se separa cada uno de los vectores que se concatenan con comas.

Por ejemplo, podemos escribir el código del sumador completo de dos bits visto en los operadores aritméticos del siguiente modo:




```
module  sumador (CO, S, A, B, CI);

output   CO;
output   [1:0] S;
input    CI;
input    [1:0] A, B;

assign  {CO,S} = A+B+CI;

endmodule
```

Otro ejemplo:

```
.....
wire [1:0] a, b;
wire [2:0] x;
wire [3:0] y, Z;

assign x = {1'b0, a}; //x[2]=0, x[1] = a[1], x[0] = a[0]

assign y = {a, b};  // y[3] = a[1], y[2] = a[0], y[1] = b[1], y[0] = b[0]
.....
```

Es ilegal tener un número sin tamaño definido en una concatenación. Esto es debido a que se utiliza la concatenación para crear un número específico de bits, y es por lo tanto absurdo no definir el tamaño de alguno de sus elementos. Es una buena práctica definir siempre el tamaño de las constantes en el código Verilog para evitar problemas en la concatenación.

OPERADOR CONDICIONAL

Este operador es similar al utilizado en C. Requiere tres operandos y su sintaxis es la siguiente:

sintaxis ::= conditional_expression ? true_expression : false_expression;

Es equivalente a una instrucción del tipo *if-then-else*. El primer operando (*conditional_expression*) debe ser una condición cuyo resultado sea 1 (*true*) ó 0 (*false*). Si el resultado de la expresión condicional es verdadero se devuelve el valor del operando *true_expression*, y si el resultado de la expresión condicional es falso se devuelve el valor del operando *false_expression*.

Hay que tener en cuenta lo siguiente:

- El valor Z (alta impedancia) no está permitido como resultado de evaluar la condición.
- Los operandos de diferente longitud se rellenan automáticamente con 0's pada poder ser comparados.

Ejemplo: vamos a ver cómo se podría implementar un buffer triestado de este modo.

```
module  buffer_triestado (out, in, enable);

output   out;
input    in, enable;

assign  out = enable ? in : 1'bz;

endmodule
```

Podemos necesitar los llamados operadores de relación para poder escribir la condición del operador condicional. Los operadores de relación son los que se muestran en la siguiente tabla:

Operador	nº de operandos	Descripción
==	2	Comprueba si dos operandos son iguales
!=	2	Comprueba si dos operandos son diferentes
<	2	Comprueba si Op1 es menor que Op2 (Op1<Op2)
>	2	Comprueba si Op1 es mayor que Op2 (Op1>Op2)
<=	2	Comprueba si Op1 es menor o igualque Op2 (Op1<=Op2)
>=	2	Comprueba si Op1 es mayor o igualque Op2 (Op1>=Op2)

Una vez vistos estos operadores de relación y los operadores lógicos, vamos a ver cómo se podría implementar un multiplexor de 4 líneas a 1 con entrada de habilitación en baja, utilizando el operador condicional. El código Verilog sería el siguiente:

```
module  mux_4_1 (out, in, enable, select);

output   out;
input     enable;
input  [1:0] select;
input  [3:0] in;

assign  out = enable ? 0 :
                  (select == 2'b00) ? in[0] :
                  (select == 2'b01) ? in[1] :
                  (select == 2'b10) ? in[2] : in[3];

endmodule
```

Se utiliza el operador condicional equivalente a sentencias *if-then-else* anidadas. Se está suponiendo que el valor de los bits de selección siempre estará definido correctamente.

RETARDOS

podemos introducir retardos en las asignaciones continuas del siguiente modo:

assign #2 y_out = a ^ b;

En este caso se introducen dos unidades de retardo en la salida y_out desde el cambio de alguna de las entradas.

4 - PRIMITIVAS DEFINIDAS POR EL USUARIO

Las primitivas predefinidas en Verilog son útilies, pero limitadas. El usuario puede crear primitivas adicionales que simulen comportamientos combinacionales o secuenciales más complicados. Cuando el usuario define una primitiva, no hace llamadas a otras primitivas (predefinidas o no) ni tampoco a módulos. Las primitivas han de ser declaradas fuera del módulo en el cual son referenciadas.

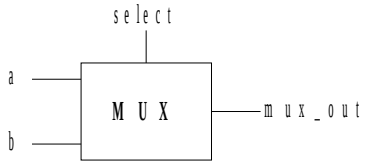
Las primitivas predefinidas por el usuario requieren menos memoria que los módulos, y se simulan más rápidamente que estos últimos. Las llamadas a primitivas definidas por el usuario se hacen del mismo modo que las llamadas a primitivas predefinidas en Verilog. Exactamente igual, se pueden definir con o sin retardos de propagación.

Otras características de las primitivas definidas por el usuario son las siguientes:

- Pueden tener un máximo de diez entradas.
- Han de tener una única salida.
- Tanto las entradas como las salidas han de ser magnitudes escalares, es decir no pueden ser buses.
- No soportan puertos bidireccionales.

PRIMITIVAS COMBINACIONALES

Vamos a ver ahora cómo se define una primitiva que implemente un circuito combinacional. Utilizaremos como ejemplo un multiplexor de dos líneas a 1.



El código en Verilog que implementa dicha primitiva será el siguiente:

```
primitive   mux_prim (mux_out, select, a, b);
output  mux_out;
input    select, a, b;

table

//  select        a       b       :  mux_out

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

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

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


endtable

endprimitive
```

Para definir una primitiva, se comienza con *primitive*, y finalizamos de definirla con *endprimitive*. Debemos dar un nombre a la primitiva que definimos, que en este caso es *mux_prim*. A continuación se define la única salida (*mux_out*) y las entradas existentes (*select*, *a*, *b*).
La función que realiza la primitiva se define mediante una tabla de verdad, comenzando por *table*, y cerrando con *endtable*. Hay que tener cuidado con el orden de las entradas en la tabla, ya que Verilog supone que es la misma que en la definición de los puertos de entrada. La salida se define al final.

Para definir la tabla de verdad, además de 0 y 1 lógicos, se ha utilizado el símbolo *?*. Mediante este símbolo queremos representar la condición "no importa" en la entrada. Los dos últimos estados de entrada definidos, donde *select* toma el valor *?* se han definido para "reducir pesimismo", ya que en el caso en que no sepamos qué canal de entrada estemos seleccionando, está claro que si ambos canales toman el mismo valor (0 ó 1 lógico), dicho valor será el que se muestre en la salida.