1 - INTRODUCCIÓN

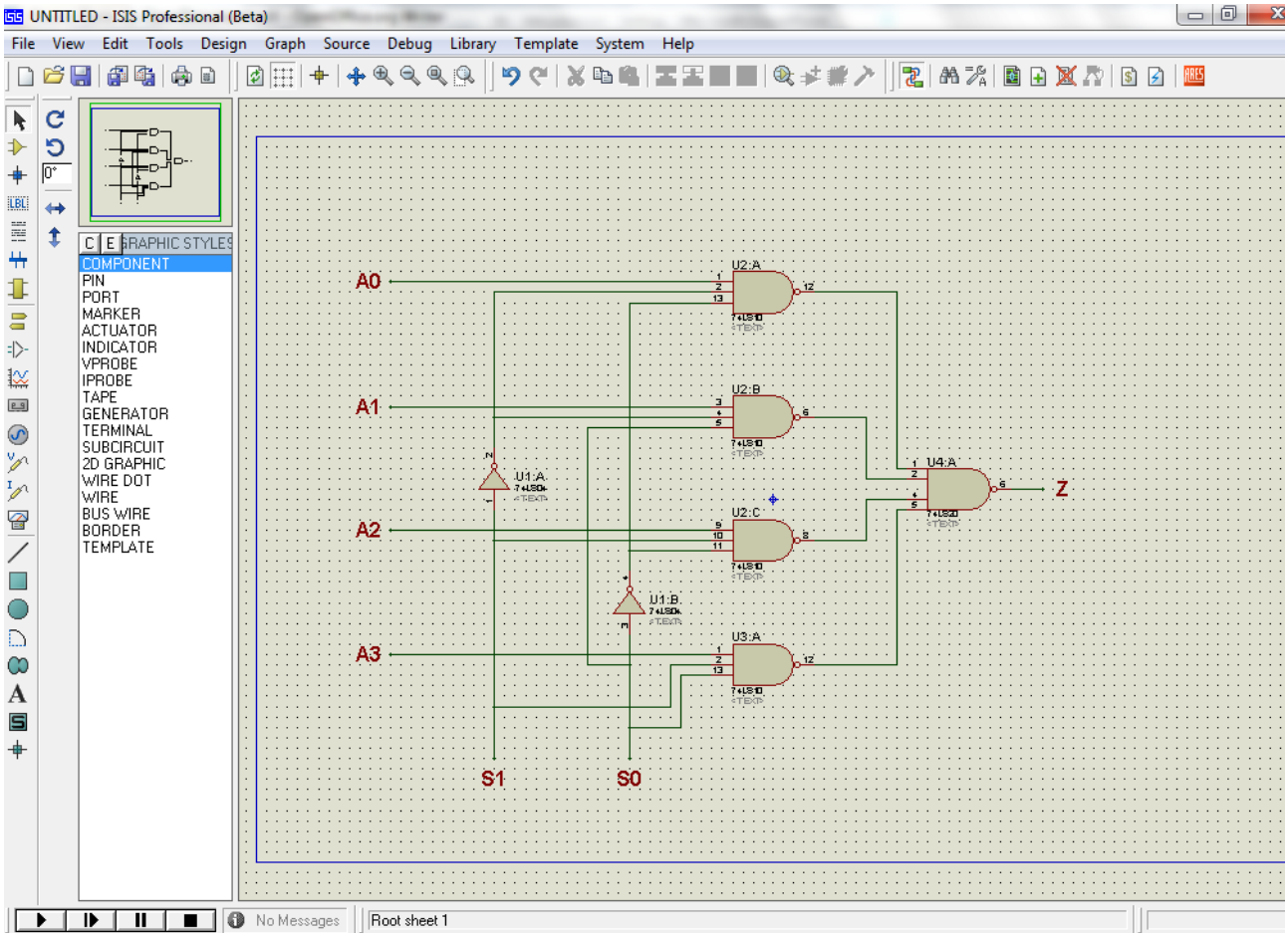
La creación de circuitos integrados utilizando herramientas CAD (*Computer Aided Design*) conlleva una serie secuencial de pasos, comenzando con el diseño de entrada y finalizando con la generación de una base de datos que contenga los detalles geométricos de las fotomáscaras que se utilizarán para el diseño sobre el silicio. En los pasos intermedios habría, entre otras cosas, que comprobar el correcto funcionamiento del circuito mediante la simulación del diseño introducido.

Podemos introducir el diseño de entrada del circuito en el ordenador utilizando una herramienta gráfica de captura esquemática. Dos conocidos ejemplos (aunque no los únicos) serían:

- *Proteus ISIS Schematic Capture*, de la empresa LabCenter Electronics.
- *OrCAD Capture*, de la empresa Cadence Design Systems.

En estos casos, una vez que el diseño ha sido introducido al ordenador utilizando la herramienta gráfica, estos programas son capaces de generar un listado de conexiones o *netlist* donde se describe el circuito diseñado, y dicho fichero puede ser leído por herramientas SPICE (*Simulation Program with Integrated Circuit Emphasis*) para la simulación.

Por ejemplo, si queremos diseñar un circuito sencillo, un multiplexor de 4 líneas a 1 utilizando puertas NAND e inversores, el circuito en la herramienta gráfica de diseño esquemático tomaría un aspecto como éste:



Sin embargo, los circuitos integrados digitales actuales pueden contener cientos de miles de puertas lógicas, por lo que resulta prácticamente imposible introducir el diseño de dichos circuitos al ordenador utilizando herramientas gráficas de captura esquemática. Incluso el diseño *manual* de circuitos puede ser desalentador para circuitos más sencillos, de algunos miles de puertas lógicas.

Otra forma de introducir el diseño del circuito al ordenador, es mediante un fichero de texto escrito en un lenguaje de descripción de hardware (HDL *Hardware Description Language*). Los lenguajes de descripción de hardware se utilizan para modelar la arquitectura y el comportamiento de sistemas electrónicos, especialmente digitales, y también para la simulación del comportamiento temporal. La descripción del circuito en HDL no necesita ninguna herramienta especial para ser leída o modificada, a diferencia de las herramientas gráficas de diseño esquemático. Por lo tanto, el código es mucho más portátil.

Los lenguajes de descripción de hardware tomaron importancia en la década de los 80 debido a la creciente complejidad de los diseños electrónicos. En la actualidad, los dos principales lenguajes de descripción de hardware son:

- VHDL (*VHSIC Hardware Decription Language*, donde *VHSIC = Very High Speed Integrated Circuit*).
- *Verilog* HDL. Es más utilizado en la industria, y tiene una sintaxis similar al lenguaje de programación C.

Los lenguajes HDL pueden describir un diseño a nivel de puertas, conmutadores e incluso de transistores. A este tipo de diseños se los denomina estructurales, ya que definimos la estructura real a nivel de puertas, conmutadores o transistores del circuito. En el caso del multiplexor de 4 líneas a 1 anterior, el código en lenguaje Verilog a nivel de puertas sería el siguiente:

```
module  MUX_4_1 (Z, S1, S0, A3, A2, A1, A0);

input   S1, S0, A3, A2, A1, A0;
output  Z;
wire    W3, W2, W1, W0, NS1, NS0;

not     (NS0, S0);
not     (NS1, S1);
nand    (W3, A3, S1, S0);
nand    (W2, A2, S1, NS0);
nand    (W1, A1, NS1, S0);
nand    (W0, A0, NS1, NS0);
nand    (Z, W3, W2, W1, W0);

endmodule
```

Sin embargo, la principal característica de los lenguajes de descripción de hardware es que permiten utilizar mayores niveles de abstracción en los que el diseñador no se ocupa de los detalles a nivel de puertas o de transistores. **El diseñador puede especificar** el comportamiento que ha de tener un circuito en lugar de su estructura, es decir, **cómo ha de comportarse el circuito en lugar de cómo ha de estar hecho el circuito.**

En la siguiente figura podemos ver los distintos niveles de abstracción a la hora de describir un circuito electrónico. Verilog permite descripciones desde el nivel de sistema hasta el de conmutadores, aunque comúnmente se utiliza desde el nivel funcional (*behavioral*) hasta el nivel de puertas lógicas.

Por ejemplo, si queremos diseñar un flip-flop tipo D, con una entrada de datos (data_in), una entrada para la señal de reloj (clk), una entrada de reset síncrono (rst) y una salida (q), un posible código en Verilog sería el siguiente:

```
module  flip-flop (q, data_in, clk, rst);
input   data_in, clk, rst;
output  q;
reg     q;

always @ (posedge clk)
  begin
    if     (rst==0) q=0;
    else   q=data_in;
  end
endmodule
```

Básicamente, lo que nos dice el código es que el circuito está continuamente esperando un flanco positivo en la señal de reloj. Cuando eso ocurra, si el reset está activado (en baja), la salida se resetea, y si no estuviera activado, la salida se hace igual a la entrada.

En este caso, hemos utilizado un nivel de abstracción superior, ya que hemos definido el comportamiento del circuito (*behavioral*) pero no hemos especificado absolutamente nada sobre su estructura interna. Existen herramientas de síntesis que pueden crear los esquemáticos a partir de la descripción hardware del circuito. Por lo tanto, el diseñador puede ahorrar tiempo evitando el diseño completo del circuito a nivel de puertas, ya que de ello se encargará una herramienta de síntesis. Tampoco se necesitará inicialmente conocer los componentes o tecnología que se utilizarán para construir el circuito.

Las simulaciones eléctricas de los diseños en HDL son eficientes computacionalmente. Son simulaciones discretas o controladas por eventos (*event-driven*) frente a las simulaciones continuas basadas en la solución de un conjunto de ecuaciones diferenciales que representan el comportamiento eléctrico de cada transistor que forma parte del circuito. En las simulaciones controladas por eventos es un cambio en alguna de las entradas lo que origina un cambio en las salidas tras un retardo definido en el programa. En otras palabras, las simulaciones controladas por eventos implican una cadena de causas y efectos. Perdemos precisión en los resultados a cambio de ganar velocidad en el cálculo de la solución.

2 - DISEÑO ESTRUCTURAL

El diseño estructural consiste en la interconexión de objetos de diseño creando una estructura que contiene el comportamiento deseado. Se podría interpretar como la traducción “literal” de la captura esquemática al lenguaje Verilog. El diseño estructural se utiliza para implementar circuitos combinacionales. Es lo que veremos en este capítulo.

MÓDULOS EN VERILOG

La unidad básica de diseño en Verilog se denomina **módulo**. Un módulo en Verilog está formado por una información que contiene la estructura o el comportamiento de un determinado circuito, así como una descripción de las características temporales.

La estructura de un módulo es la siguiente:

```
module  nombre_modulo (z0, z1, ..., zp-1, x0, x1, ..., xn-1);
output    z0, z1, ..., zp-1;
input     x0, x1, ..., xn-1;


// Aquí vendría la descripción estructural
// o funcional del circuito

endmodule
```

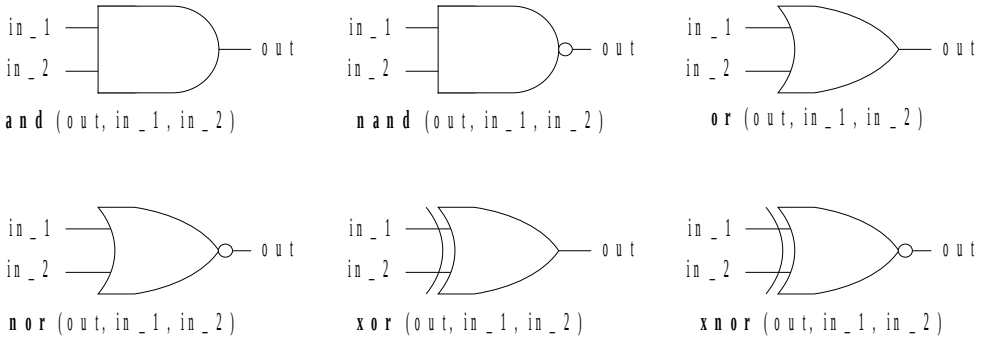
En la definición de módulo debemos, lo primero de todo, asignar un nombre a dicho módulo (en este caso *nombre_modulo*). A continuación debemos de indicar entre paréntesis todos los puertos del módulo, tanto de entrada como de salida. En las siguientes líneas se especifica qué puertos son de entrada (*input*) y cuáles de salida (*output*). Se procedería entonces a definir la función que ha de realizar el módulo, o su diseño estructural, y una vez hecho esto se finaliza el módulo (*endmodule*). Podemos fijarnos que en Verilog los comentarios se hacen comenzando con // y siguen hasta nueva línea. Lógicamente, los comentarios los saltará el compilador de Verilog.

Además de los puertos de entrada y de salida, un módulo puede contener puertos bidireccionales (de entrada-salida). Estos puertos de declaran en Verilog como *inout*.
Por ahora, supondremos que tanto las entradas como las salidas son simples cables de conexión, realmente un tipo de variable llamado *wire* en Verilog. Veremos luego otros tipos de variables.

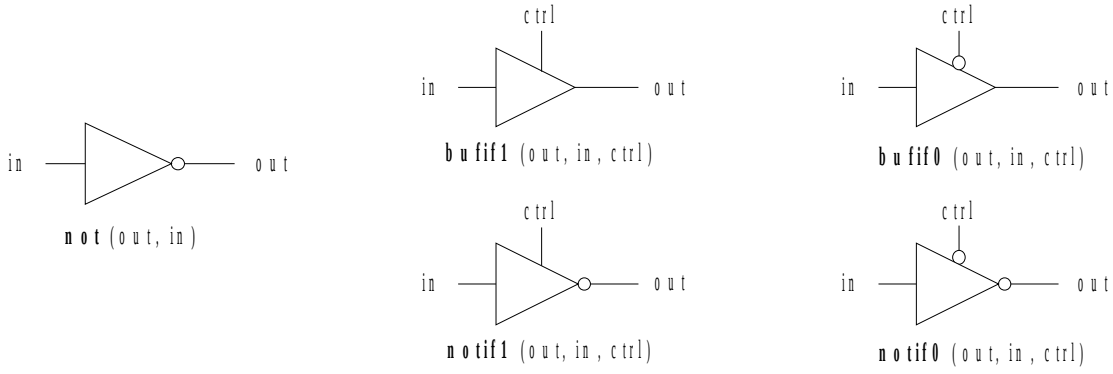
PRIMITIVAS PREDEFINIDAS EN VERILOG

Verilog tiene predefinidas una serie de elementos estructurales llamados primitivas. Estas primitivas son esencialmente puertas lógicas. Las principales primitivas predefinidas son las siguientes:

1. **Puertas multi-entrada.** Todas estas puertas lógicas pueden tener un número mayor de entradas, pero sólo una salida.



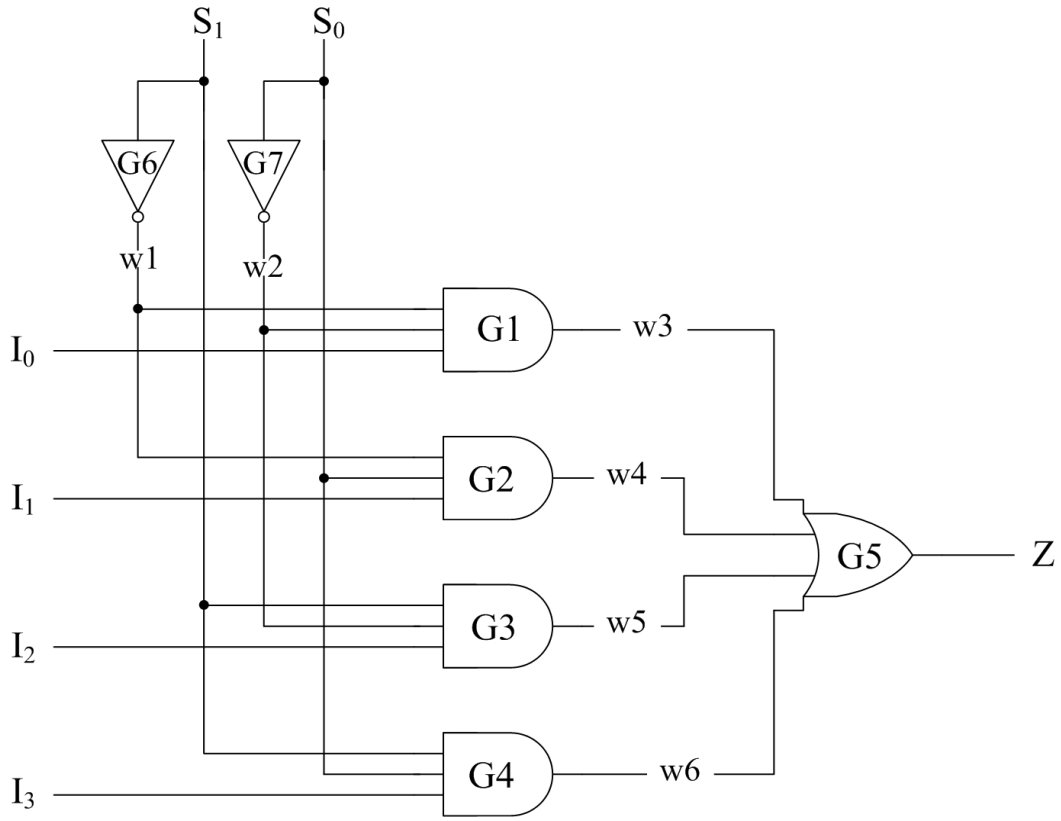
2. **Puertas de una única entrada y buffers triestado.**



Supondremos ahora que el retardo de propagación de todas estas puertas lógicas y buffers triestado es nulo. Más adelante veremos cómo se pueden incluir los retardos a estas primitivas predefinidas.

EJEMPLO DE DISEÑO ESTRUCTURAL

Vamos a ver un sencillo ejemplo de diseño estructural en Verilog. Supongamos que queremos implementar un multiplexor de 4 líneas a 1. Su esquema de puertas es el que se muestra en la siguiente figura.



Hemos dado un nombre a cada una de las puertas (G1, G2, G3, G4, G5, G6, G7). Además, hemos dado un nombre también a cada una de las conexiones internas (w1, w2, w3, w4, w5, w6). El código en Verilog de un módulo que contenga el diseño estructural de este multiplexor sería el siguiente:

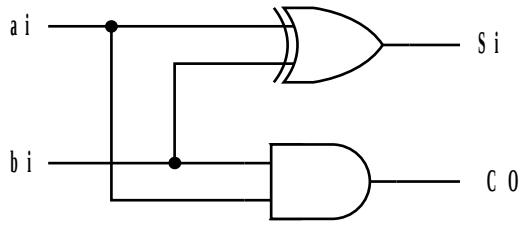
```
module  MUX4_1 (Z, I0, I1, I2. I3, S1, S0);
output  Z;
input   I1, I2, I3, I4, S1, S0;
wire    w1, w2, w3, w4, w5, w6;

and  G1   (w3, w1, w2, I0);
and  G2   (w4, w1, S0, I1);
and  G3   (w5, S1, w2, I2);
and  G4   (w6, S1, S0, I3);
not  G6   (w1, S1);
not  G7   (w2, S0);
or   G5   (Z, w3, w4, w5, w6);

endmodule
```

Vemos cómo el nombre del módulo es MUX4_1, y hemos definido los puertos de entrada y los puertos de salida del módulo. Hemos definido también las conexiones internas, mediante un tipo de variables que en Verilog se denomina *wire*. Hemos dado un nombre a cada una de las primitivas, aunque en realidad esto no es realmente necesario.

LLAMADAS A MÓDULOS

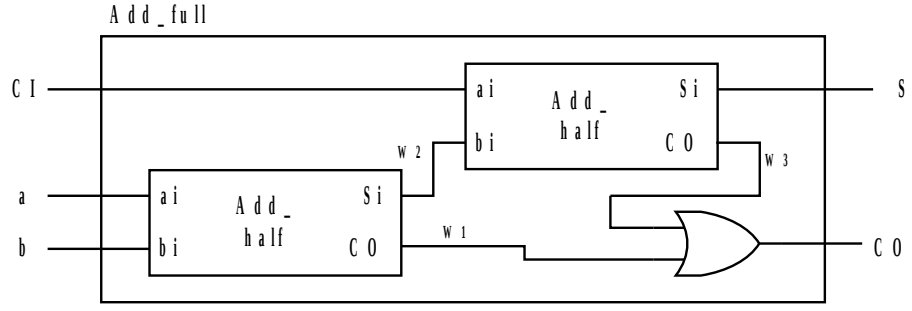
Como hemos visto, cuando diseñamos un módulo, le damos un nombre y definimos todos sus puertos de entrada y de salida. El módulo que hemos creado, puede ser utilizado posteriormente llamando a dicho módulo. Mediante la llamada a un módulo (*instantiation* en inglés), utilizamos módulos en un niver jerárquico menor (módulos hijos) para la construcción de módulos de nivel jerárquico mayor (módulos padre).

Vamos a ver un ejemplo: cómo construir un sumador completo de 1 bit utilizando semisumadores de 1 bit. En la siguiente figura podemos ver un semisumador de 1 bit, junto con su código en Verilog.




```
module  Add_half (Si, CO, ai, bi);
output  Si, CO;
input   ai, bi;

xor     (Si, ai, bi);
and     (CO, ai, bi);

endmodule
```

Utilizando dos semisumadores y una puerta OR podemos crear un sumador completo de 1 bit, y por lo tanto, necesitaremos hacer dos llamadas al módulo semisumador, que hemos llamado *Add_half*. Las llamadas al módulo se hacen simplemente por el nombre. El esquema es el mostrado en la figura.



El código Verilog del módulo sumador completo sería el siguiente:

```
module   Add_full (S, CO, a, b, CI);
output   S, CO;
input    a, b, CI;
wire     W1, W2, W3;

Add_half   M1   (W2, W1, a, b);
Add_half   M2   (S, W3, CI, W2);
or              (CO, W3, W1);

endmodule
```

La llamada a un módulo debe llevar siempre un identificador, a diferencia de las primitivas predefinidas, que es opcional (y por ejemplo aquí no se ha utilizado). El diseño en este caso es jerárquico: el módulo *Add_full* sería el módulo padre, que incluye dos módulos *Add_half*, que serían en este caso los módulos hijos.

Un módulo nunca ha de ser declarado dentro de otro módulo!!!

TIPOS DE DATOS EN DISEÑO ESTRUCTURAL

Existen diferentes tipos de datos que pueden ser utilizados en el diseño estructural. Los más importantes (y los que utilizaremos nosotros) son los siguientes:

- ***wire***: representa un cable físico en un circuito y se utiliza para conectar primitivas o módulos. Representa un cable sin más, sin propiedades especiales. Adopta alguno de los valores básicos de la tabla siguiente:

Valor	Descripción
0	0 lógico, condición falsa
1	1 lógico, condición verdadera
X	Valor lógico desconocido
Z	Estado de alta impedancia

- ***tri***: este tipo de dato tiene la misma funcionalidad que *wire*, pero indica explícitamente que será tri-estado.

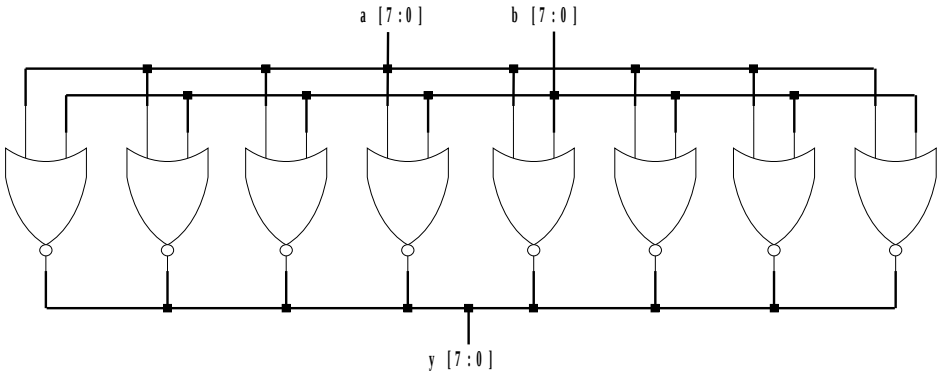
- ***supply0***: conexión a tierra del circuito, es decir, 0 lógico.

- ***supply1***: conexión a alimentación del circuito, es decir, 1 lógico.

BUSES O VECTORES

Hasta ahora, hemos supuesto que la dimensión de los puertos de un módulo o de las variables era 1 (1 bit), que es la dimensión que por defecto usa Verilog. Sin embargo, podemos cambiar la dimensión tanto de los puertos de un módulo como de los datos, lo cual es útil para crear buses. Verilog permite definir buses de hasta 1000000 de líneas. Normalmente a los buses se los llama **vectores**. También podemos incluso definir un vector de primitivas e incluso de llamadas a módulos.

Por ejemplo, en el siguiente circuito hemos supuesto que la entrada consta de dos buses de 1 byte cada uno de ellos, y la salida es otro bus de 1 byte.



Su código en verilog sería el siguiente:

```
module   array_of_nor (y, a, b);
output  [7:0]    y;
input    [7:0]    a, b;

nor   G1 [7:0]  (y, a, b);

endmodule
```

En este caso, podemos ver cómo, además de definir un vector en los puertos de entrada y salida del módulo, hemos definido un vector o array de primitivas, en este caso de puertas NOR.

En la declaración de un bus, la línea cuyo nombre aparece a la izquierda será siempre la más significativa (*msb*) y la de la derecha la menos (*lsb*) independientemente de con qué número se las nombre. Esto es importante a la hora de conectar terminales de distintos circuitos.

Podemos referirnos también a un elemento del bus. Por ejemplo, en el caso del conjunto de puertas NOR del último ejemplo, la expresión a[2] indicaría el elemento dos del bus a. También podemos seleccionar un conjunto de bits dentro del bus. Por ejemplo a[5:1] seleccionaría los bits 5,4,3,2,1 del bus a.

TIEMPO DE RETARDO DE PUERTAS LÓGICAS

Como ya sabemos, las tensiones que representan valores lógicos en los circuitos no se propagan instantáneamente desde la entrada a la salida de las puertas lógicas, limitando la velocidad a la que los circuitos digitales pueden funcionar correctamente. También existen retardos en la propagación de las señales a través de los cables del circuito, aunque nosotros no estudiaremos este último caso.

Definiremos tres retardos distintos para las puertas lógicas:

- *Rising delay:* retardo en la salida cuando ésta hace una transición 0→1, Z→1 ó X→1.
- *Falling delay:* retardo en la salida cuando ésta hace una transición 1→0, Z→0 ó X→0.
- *Turn-off delay:* retardo en la salida cuando ésta hace una transición 0→Z, 1→Z, ó X→Z.

Ejemplos:

```
and   (y_out, x1, x2); // Retardo nulo
and   #1 (y_out, x1, x2); // Todos los retardos son de 1 unidad
and   #(2,3) (y_out, x1, x2); // rising delay de dos unidades y falling delay de 3 unidades
bufif1   #(1,2,3) (out, in, ctrl); // rising, falling y turn-off delays de 1, 2 y 3 unidades respectivamente.
```

Al comienzo del programa en Verilog, hemos de definir la escala de tiempos que se utilizará durante la simulación. El comando es **`timescale**. Por ejemplo, **`timescale 1 ns / 1 ps** querría decir que la unidad de tiempo es de 1 ns, mientras que el paso del simulador sería de 1 ps. El paso del simulador nunca puede ser mayor que la unidad de tiempo.

Hay que tener cuidado con lo siguiente:
Por ejemplo, si **`timescale 10 ns / 100 ps**, y en el código aparece #4,629, el simulador lo tomará como 46,3 ns, debido a que la resolución del simulador está limitada a 100 ps.
Por lo tanto, conviene tomar un paso del simulador adecuado. De todos modos, cuanto menor sea este valor, má tiempo tardará en simularse el circuito.

El comando **`timescale** puede cambiar a lo largo del programa, y su validez sería desde que aparece hasta el final del programa o hasta que aparezca de nuevo.