Manual de Verilog
ver 0.4

Jorge Chávez

Marzo 1999

Índice General

En este documento no se pretende describir exhaustivamente todas y cada una de las funcionalidades de Verilog , sino dar una visión superficial sobre el lenguaje y de las diferentes descripciones que se pueden realizar de un sistema. El autor agradece cualquier sugerencia o comentario sobre cualquiera de los aspectos de este documento:
Jorge Chávez
`chavez@esi.us.es`

1 Introducción

Verilog es un lenguaje para la descripción de sistemas digitales (**HDL**: *Hardware Description Language*). Los sistemas pueden ser descritos:

- **Nivel estructural** empleando elementos de librería o bien elementos previamente creados, se realiza la interconexión de unos con otros. Sería similar a una captura esquemática donde la función del diseñador es instanciar bloques y conectarlos entre sí.
- **Nivel de comportamiento** el diseñador describe la transferencia de información entre registros (nivel **RTL**: *Register Transfer Level*).

Estos dos niveles de descripción pueden mezclarse, dando lugar a los denominados diseños mixtos[1]. Existen multitud de lenguajes HDL en el mercado (de hecho inicialmente cada fabricante disponía de su propio lenguaje), sin embargo la necesidad de unificación ha hecho que en la actualidad sólo existan dos grandes lenguajes: VHDL y Verilog . Ambos están acogidos a estándares IEEE (VHDL en 1987 y Verilog en 1995). Existen defensores y detractores de cada uno de ellos. Con carácter general se dice que es más fácil aprender Verilog al ser un lenguaje más compacto[2].
Verilog nació en 1985 como un lenguaje propietario de una compañía (Cadence Design System), pero en 1990 se formó **OVI** (*Open Verilog International*) haciendo dicho lenguaje de dominio público, permitiendo a otras empresas que pudieran emplear Verilog como lenguaje, con objeto de aumentar la difusión de dicho lenguaje.

2 El lenguaje

Uno de los aspectos que salta a la vista al contemplar un código Verilog es su similitud con el lenguaje C. Una de las mayores diferencias que presenta este lenguaje es que permite modelar sistemas digitales reales, que funcionan de forma paralela a diferencia de la ejecución secuencial, típica de un sistema computacional.

2.1 Comentario

Existen dos alternativas: desde los caracteres "//" hasta el final de la línea es considerado comentario. Otra alternativa es la zona comprendida entre "`/*`" y "`*/`".

2.2 Números

La expresión general de una cantidad es:

tamaño	base	número

Donde:

[1] Según el contexto, un diseño mixto también podría ser aquel que mezcla analógico con digital
[2] un mismo diseño ocupa menos líneas de código

- tamaño es el número de bits (expresado en decimal) de la cantidad que viene a continuación. Es opcional.
- base indica la base en la que se va a expresar `número`. Es opcional, si no se indica es decimal.

  `'b` base binaria
  `'d` base decimal
  `'h` base hexadecimal
  `'o` base octal

- número es la cantidad. Para facilitar la lectura se pueden colocar "_". Además de las cifras permitidas en cada base (0-9 y A-F), se pueden emplear en el caso de base binaria:

  x para indicar que el valor de un bit es desconocido
  z para indicar que el valor de un bit es alta-impedancia

Ejemplos:

187	número decimal
8'h0a	número hexadecimal de 8 bits
3'b1	número binario de 3 bits
'o73	número octal
2'b1x	número binario de dos bits cuyo bit menos significativo es desconocido
4'bz	número binario de 4 bits en alta impedancia
-4'b10	número binario de 4 bits complemento a 2 de 10 (1110)
'b1000_0001	número binario de 8 bits

2.3 Identificadores

Un identificador está formado por una letra o "_" seguido de letras, números y los caracteres "$" o "_". Se distingue entre mayúsculas y minúsculas.

2.4 Variables

Existen 2 tipos fundamentales de variables:

- `reg` es un registro y permite almacenar un valor
- `wire` es una red que permite la conexión

Por defecto dichas variables son de un único bit, pero pueden declararse variables con un mayor número de bits:

tipo [msb : lsb] varname;

Ejemplo:

```
reg [5:0] C; // Es un registro de 6 bits, siendo C[0] el bit menos significa-
tivo.
```

También es posible también definir memorias, así por ejemplo:

```
reg [7:0] mem [0:1023]; // Es la declaración de una memoria de 1024 bytes.
```

Además de estos tipos existen otros tipos de datos más abstractos como: `integer` (registro de 32 bits), `real` (registro capaz de almacenar un número en coma flotante) o `time` (registro unsigned de 64 bits).

2.5 Estructura general

En Verilog un sistema digital está compuesto por la interconexión de un conjunto de **módulos**.

```
module <nombre del modulo> ( <señales> );
<declaraciones de señales>
<funcionalidad del módulo>
endmodule
```

Algunos aspectos a recordar son:

- Cada módulo dispone de una serie de entradas y salidas a través de las que se interconecta con otros módulos, aunque puede no tener entradas ni salidas.
- No existen variables globales.
- Fuera de los módulos sólo puede haber directivas de compilador, que afectan al fichero a partir del punto en el que aparecen.
- A pesar de que se pueden realizar varias simulaciones concurrentes (funcionando paralelamente con la misma base de tiempos), en general se suele tener un único módulo superior que emplea módulos previamente definidos
- Como se ha comentado con anterioridad cada módulo se puede describir de forma **arquitectural** o de **comportamiento**.

Los argumentos del módulo (que le permiten comunicarse con el exterior) pueden ser de tres tipos:

- `input`: entradas del módulo. Son variables de tipo `wire`
- `output`: salidas del módulo. Según el tipo de asignación que las genere serán: `wire` si proceden de una asignación continua o `reg` si proceden de una asignación procedural.
- `inout`: entrada y salida del módulo. Son variables de tipo `wire`. En la sección 5.5 comentaremos más detalladamente este tipo de argumentos.

En la figura 1 se muestran dos módulos. El primero se denomina `sumador3` y tiene 4 señales para comunicarse con otros bloques. Estas señales son dos buses de 3 bits de entrada, 1 bus de 3 bits de salida y una señal de un bit de salida. Este bloque es una descripción de comportamiento, ya que generamos las salidas empleando construcciones que relacionan los registros disponibles.
El segundo módulo es el nivel superior. Obsérvese que este módulo no dispone de entradas salidas, ya que no necesita comunicarse con ningún otro bloque. Desde este módulo instanciamos al bloque `sumador3` previamente definido, conectando las señales locales `S1`, `S2`, `RES` y `C`. Esta descripción es una descripción estructural, ya que haciendo uso de bloques previamente definidos, los conectamos de la forma deseada.
En este ejemplo hemos visto como se instancia otro módulo:

```
module_name instance_name (args);
```

Donde `module_name` es el nombre de un módulo definido, `instance_name` el nombre que se tiene la instancia (debe de ser único dentro del módulo).
Respecto a los argumentos existen dos alternativas:

- Argumentos implícito: cuando se tiene en cuenta el orden de los argumentos en la llamada para la asignación dentro del módulo. Es el caso de la figura 2
- Argumento explícito: cuando se especifica qué variable del módulo se asocia a cada variable de la llamada. En el ejemplo de la figura 2 se muestra un ejemplo de argumentos explícitos.

```
‘timescale 1ns/1ns
module sumcarry3( A, B, R, C );
input  [2:0] A, B;
output [2:0] R;
output C;

reg    [2:0] R;
reg    C;

always @(A or B) begin
  {C,R}=A+B;
end

endmodule


module test;
reg  [2:0] S1, S2;
wire [2:0] RES;
wire C;

sumcarry3 mysum(S1, S2, RES, C);

parameter T=100;

initial begin
  S1=0;
  S2=0;
  $display("Comienza la simulacion");
  #(T) S1=4;
  #(T) S2=3;
  $display("Termina la simulacion");
  $finish;
end

endmodule
```

Figura 1: Ejemplo con dos módulos

```
module test;
reg  [2:0] S1, S2;
wire [2:0] RES;
wire C;

sumcarry3 mysum( .C(C), .R(RES), .A(S1), .B(S2));

endmodule
```

Figura 2: Ejemplo con argumentos explícitos

2.6 Procesos

El concepto de procesos que se ejecutan en paralelo es una de las características fundamentales del lenguaje, siendo ese unos de los aspectos diferenciales con respecto al lenguaje procedural como es el C.
Los procesos comienzan con el inicio de la simulación y secuencialmente van procesando cada una de las líneas que aparecen. Se distinguen dos tipos de procesos:

- `initial` cuando ejecutan la última línea dejan de ejecutarse.
- `always` cuando ejecutan la última línea comienzan de nuevo a ejecutar la primera.

Veamos algunos ejemplos para clarificar:

```
initial begin
  A=5;
  B=0;
  C='bz;
end
```

Este proceso asigna los valores a las variables A, B y C en el instante inicial.

```
always begin
   #(10) CLK=0;
   #(10) CLK=1;
end
```

En este caso tenemos un proceso que después de 10 unidades de tiempo asigna el valor de la variable $CLK = 0$, después de otras 10 unidades de tiempo asigna $CLK = 1$, comenzando de nuevo su ejecución una vez acabado.

```
initial begin
   A=1;
   wait (B==1);
   #(5) A=0;
end
```

En este proceso se asigna inicialmente $A = 1$, se espera a que la variable B tome el valor 1 y una vez garantizado esto se asigna $A = 0$.

2.7 Operadores

2.7.1 Binarios aritméticos

El operador aparece entre los dos operandos. Si algún bit es `x` el resultado es `x`

`+`	suma
`-`	diferencia
`*`	multiplicación
`/`	división
`%`	resto

2.7.2 Relacionales

Permiten comparar dos operandos, retornando el valor cierto(1) o falso(0). Si algún bit es x el resultado es x

>	mayor que
>=	mayor o igual que
<	menor que
<=	menor o igual que
==	igual que
!=	diferente a

2.7.3 Lógicos

Aparece entre dos operandos lógicos y proporciona un valor lógico cierto (1) o falso (0)

!	negado (único argumento)
&&	AND lógica
||	OR lógico

2.7.4 Lógica de bit

Permiten efectuar operaciones lógicas con los bits de los operandos:

~	negación bit a bit
&	AND bit a bit
|	OR bit a bit
^	XOR bit a bit
~&	NAND bit a bit
~|	NOR bit a bit
~^ o ^~	NOT XOR bit a bit

2.7.5 Lógica de reducción

Tienen un único argumento, siendo su resultado un único bit

&	reducción AND
|	reducción OR
^	reducción XOR
~&	reducción NAND
~|	reducción NOR
~^ o ^~	reducción NOT XOR

2.7.6 Otros

{,}	concatenación
<<	desplazamiento izquierda, con adición de ceros
>>	desplazamiento derecha, con adición de ceros
?:	condicional

2.8 Estructuras de control

El lenguaje dispone de una elevada cantidad de estructuras de control, similares a las disponibles en otros lenguajes de programación.

2.8.1 if

```
if ( expresión )  command1;
else              command2;
```

Si `expresión` se evalúa como cierto(1) se ejecuta `command1` en caso contrario se ejecuta `command2`.

2.8.2 case

```
case( expresión )
val1: command1;
val2: command2;
...
default: commandN;
endcase
```

Se evalúa `expresión`, en caso de resultar `val1` se ejecuta `command1`, si resulta `val2` se ejecuta `command2`. En caso de no estar reflejado el resultado se ejecuta `commandN`.

2.8.3 for

```
for( init ; cond ; rep ) command;
```

Inicialmente se ejecuta el comando `init`, ejecutándose a continuación `command` mientras que la condición `cond` resulte un valor cierto(1), al final de cada ejecución se ejecuta `rep`.

2.8.4 while

```
while(cond) command;
```

Mientras que la condición `cond` sea cierta (1), se ejecuta el comando `command`.

2.8.5 repeat

```
repeat(Ntimes) command;
```

Repite el comando `command` tantas veces como indique `Ntimes`

2.8.6 wait

```
wait(cond) command;
```

Mientras que la condición `cond` sea falsa (0), se ejecuta el comando `command`.
A menudo se emplea para detener la ejecución secuencial del proceso hasta que se verifique una condición.

2.9 Asignaciones

Existen dos maneras de asignar valores:

- **asignación continua**

  ```
  assign var=expr;
  ```

  Donde la variable `var` debe de ser de tipo `wire`. Una asignación continua debe realizarse fuera de un procedimiento.

- **asignación procedural**

```
var=expr;
```

  Donde la variable `var` debe de ser de tipo `reg`. Una asignación procedural debe de estar siempre dentro de un proceso.

A menudo se dice que una variable de tipo `wire` no puede contener un valor. Sería sin embargo más correcto decir que es una variable dependiente (de la expresión asignada de forma continua).

```
wire A;

assign A=B&C;
```

En este ejemplo la variable `A` contiene la operación AND entre los registros `B` y `C`
Por contra las variables de tipo `reg` son capaces de contener un valor (el último valor asignado proceduralmente), esto es, se comportan como variables independientes.

```
reg A;

initial begin
   A=0;
   . . .
   A=B+C;
end
```

En este ejemplo, la variable `A` se inicializa con el valor 1, pero más adelante se modifica su valor para que sea $B + C$
(NOTA):
En algunas implementaciones del lenguaje es posible realizar una asignación continua de una variable `reg`. Esto sin embargo no suele ser una buena costumbre. Por ejemplo:

```
reg A;

initial begin
  . . .
  assign A=B+C;
  . . .
  A=5;
end
```

Obsérvese también que la asignación continua es interna al procedimiento (en contra de lo que hemos comentado anteriormente). Este código no va a dar errores simulando (en determinadas implementaciones), pero el valor de `A` no esta gobernado por la asignación procedural sino por la asignación continua, esto es la línea `A=5` no tiene ningún efecto en dicho código.

2.10 Temporizaciones

A pesar de que las asignaciones procedurales se ejecutan secuencialmente, es posible modificar el instante en el que se producen. Este retraso se especifica empleando el carácter # seguido de las unidades de tiempo.
Cabe distinguir dos tipos de asignaciones procedurales:

- con bloqueo (*blocking procedure*). La asignación se realiza antes de proceder con la siguiente.

- sin bloqueo (*non-blocking procedure*). El término derecho se evalúa en el instante actual, pero no se asigna al derecho hasta finalizar dicho instante.

```
module test;

reg [7:0] a,b;

initial begin
   a=5;

   #1  a=a+1;
       b=a+1;

   #1 $display("a=%d b=%d",a,b);

   a=5;

   #1  a<=a+1;
       b =a+1;

   #1 $display("a=%d b=%d",a,b);

end

endmodule
```

Figura 3: Ejemplo de proceso con y sin bloqueo

En el ejemplo de la figura 3 se tiene inicialmente una asignación con bloqueo, donde el resultado que imprimirá será: $a = 6$ y $b = 7$, ya que el valor de a para calcular b está actualizado. Sin embargo en la asignación sin bloqueo obtendríamos: $a = 6$ y $b = 6$, ya que el valor de a para calcular b aún no ha sido actualizado.

2.11 Eventos

Una asignación procedural puede ser controlada por el cambio de una variable, denominándose control por evento. Para ello se emplea el carácter @ seguido del evento que permite la ejecución de la asignación procedural.
Se distinguen dos tipos de eventos:

- Eventos de nivel: el cambio de valor de una o un conjunto de variables controla el acceso.
- Eventos de flanco: la combinación de flanco/s de subida ($0 \rightarrow 1$) y de bajada ($1 \rightarrow 0$)

Veamos algunos ejemplos:

```
always
   @A B=C+D;
```

Cada vez que A cambie de valor, se realizará la asignación: $B = C + D$.

```
always
   @(posedge CLK) B=C+1;
```

Cada vez que la señal CLK experimente un flanco positivo, se realizará la asignación: $B = C + 1$.
También es posible tener el evento flanco negativo empleando `negedge`.
Es posible tener condiciones de activación que dependan de más de una variable:

```
always @(A or B) C=D&3;

always @(posedge A or negedge B) C=0;
```

Se denomina **lista de sensibilidad** al conjunto que forman el evento.

2.12 Parámetros

```
parameter const=value;
```

Permite disponer de una constante. La definición de un parámetro sólo puede realizarse en el interior de un módulo.
Una aplicación frecuente suele ser la definición del periodo de un reloj:

```
parameter T=10_000;
reg CLK;

always begin
#(0.5*T) CLK=0;
#(0.5*T) CLK=1;
end
```

También se suele emplear para la definición del tamaño de un bus

```
parameter N=8;

reg [N-1:0] A;
```

Otra utilidad que tienen los parámetros es la *redefinición externa*, para ello veamos el ejemplo de la figura 4. Primero definimos el módulo `cont` que dispone de dos parámetros `N1` (tamaño del bus de salida `Q`) y `N2` (tamaño del registro interno `P`). En el módulo `test2` instanciamos la celda `cont` pero modificando sus dos argumentos: $N1 = 4$ y $N2 = 32$.
Obsérvese que los argumentos se definen implícitamente según el orden de aparición dentro del módulo.

2.13 Directivas para el compilador

Permiten modificar el procesamiento del fichero de entrada. Comienzan por un acento grave ‘ y tienen efecto desde su aparición hasta el fin de todos los ficheros de entrada[3].

2.13.1 define

```
‘define MACRO value
```

Define una macro, de forma que cada vez que aparezca `‘MACRO` en el texto será sustituida por `value`

[3] al ser un lenguaje de dos pasadas, en la primera pasada se sustituyen las directivas de compilador y a continuación se efectúa la segunda pasada

```
module cont(BCLK,BnRES,Q);
parameter N1=2;
parameter N2=8;

input  BCLK, BnRES;
output [N1-1:0] Q;

reg    [N2-1:0] P, nP;

assign Q=P[N2-1:N2-N1-1];

always @(posedge BCLK or negedge BnRES)
if(BnRES==0) P=0;
else         P=nP;

always @(P)  nP=P+1;

endmodule

module test2;

reg  clk, reset;
wire [3:0] A;

cont #(4,32) micont(clk, reset, A);

endmodule
```

Figura 4: Instancia con modificación de parámetros internos

2.13.2 include

```
‘include "filename"
```

Permite insertar el fichero `filename` en la posición donde aparezca esta directiva, veamos algunas de las más empleadas.

2.13.3 ifdef

```
‘ifdef VAR
  code
‘endif
```

Si la macro `VAR` está definida, el código `code` es incluido, en caso contrario es ignorado.

2.13.4 timescale

```
‘timescale utime / prec
```

Permite especificar la unidad de tiempo empleada (`utime`) para tiempos y retrasos, así como la precisión `prec` que se empleará en los retrasos a partir de ese instante.
Debe de verificarse que $utime \geq prec$.
Los enteros permitidos como argumentos son 1, 10 y 100. A continuación debe colocarse una cadena con la unidad de medida, que puede ser: ”s”, ”ms”, üs”, ”ns”, ”ps^{o} ”fs”

```
‘timescale 1ns / 100ps

module test1;
   parameter T=100.5;
   initial begin

   #(T/2) $display($time," test1a");
   #(T/2) $display($time," test1b");
   end
endmodule

‘timescale 1ns / 1ns

module test2;
   parameter T=100.5;
   initial begin

   #(T/2) $display($time," test2a");
   #(T/2) $display($time," test2b");
   end
endmodule
```

Figura 5: Ejemplo de uso de `timescale`

En este caso, a partir de la primera directiva, todos los retrasos se expresarán en base a `1 ns`. Debido a la precisión especificada, cualquier retraso fraccional se redondeará al primer decimal, los tiempos que se imprimirán serán: 50 y 101.
A continuación se disminuye la precisión, de forma que cualquier retraso expresado no tiene cifras decimales, por lo que se imprimirá: 50 y 100.

La utilidad que tiene es emplear módulos desarrollados con diferentes unidades de tiempo, sin necesidad de modificar los retrasos de dicho fichero. No es aconsejable modificarlo sin tener una buena razón.

2.14 Funciones de sistema

Existen acciones de bajo nivel disponibles denominadas funciones de sistema, suelen variar de una implementación a otra. Comentaremos algunas de las más empleadas:

2.14.1 `$finish`

Si no se indica lo contrario la simulación es indefinida, como esto no suele ser deseable, esta función indica el final de la simulación.

2.14.2 `$time`

Esta función retorna el instante de tiempo en el que se encuentra la simulación.
(NOTA)
Debido a que cada módulo puede tener su escala de tiempos(ver sección 2.13.4), el valor retornado es el tiempo escalado al módulo que invoca dicha función.

```
‘timescale 1ns/1ns
module kk(a);
input a;

always @(a) $display($time," modulo kk");

endmodule

‘timescale 100ns/100ns
module test;
reg A;

kk kk0(A);

initial begin
  A=0;
  #(10) A=1; $display($time," modulo test");
end
endmodule
```

Figura 6: Ejemplo del efecto de cambio de escala de tiempos

En este caso desde el módulo `test` se imprimiría 10 que es el número de unidades transcurridas. El tiempo impreso desde el módulo `kk` tiene una escala de tiempos 100 inferior, la impresión resultaría 1000.

2.14.3 `$random`

Esta función retorna un valor aleatorio entero de 32 bits cada vez que se invoca.
Se le puede suministrar como argumento una variable con la semilla que controla la generación de la misma secuencia aleatoria en el caso de repetir la simulación. Esto es, los valores obtenidos de la invocación repetida de esta función son aleatorios entre sí, pero será la misma secuencia si se ejecuta de nuevo.

2.14.4 `$display` y `$write`

```
$display( P1, P2, P3, ...);
$write( P1, P2, P3, ...);
```

Estas dos funciones de sistema permiten imprimir por la salida estándar, mensajes y variables del sistema. Ambas tienen una funcionalidad similar salvo porque `$display` coloca un salto de línea al final.
Si el argumento es una variable se imprime (en formato decimal), si es una cadena se imprime tal cual salvo que tenga caracteres de escape:

`\n`	salto de líneas
`\t`	tabulador
`\\`	carácter \
`%%`	carácter %

o indicadores de formato:

`%d`	formato decimal (defecto)
`%h`	formato hexadecimal
`%b`	formato binario
`%o`	formato octal
`%t`	formato tiempo
`%c`	carácter ASCII

En este caso, a continuación de la cadena, deben de aparecer tantos argumentos como formatos se especifiquen.
Por omisión el tamaño del formato es el máximo de la variable correspondiente, pudiendo ajustarse al tamaño actual de la variable colocando un cero después del %. Así por ejemplo:

```
reg [15:0] A;

initial begin
   A=10;
   $display("%b %0b %d %0d",A,A,A,A);
end
```

Imprimiría:

```
0000000000001010 1010 □□□10 10
```

Veamos a continuación el caso de bits indeterminados o alta impedancia

- si todos son indeterminados se imprime x
- si todos son alta impedancia se imprime z
- si existen algunos bits indeterminados se imprime X según el formato
- si existen algunos bits alta impedancia se imprime Z según el formato

Así por ejemplo:

```
reg [15:0] A;

initial begin
   A=A|31;
   $display("%b %h %d",A,A,A);
end
```

Imprimiría (al no haber sido inicializada A):

```
xxxxxxxxxxx11111 xxXf X
```

2.14.5 `$fdisplay` y `$fwrite`

```
$fdisplay( fd, P1, P2, P3, ...);
$fwrite( fd, P1, P2, P3, ...);
```

Estas dos comandos son similares a los del apartado anterior salvo que permiten almacenar el resultado en un fichero, cuyo descriptor (`fd`) se le da como primer argumento.
Las funciones para manipular la apertura y cierre del fichero son:

```
fd=$fopen( nombre_del_fichero);
$fclose( fd );
```

Veamos un ejemplo:

```
integer fd;

initial fd=$fopen("resultados.dat");

initial begin
   for(i=0;i<100;i=i+1) begin
     #(100) $fdisplay(fd,"%d %h",A,B);
   end
   $fclose(fd);
   $finish;
end
```

(NOTA)
Al emplear descriptores de canal es posible escribir simultáneamenente en varios canales empleando un descriptor que sea la OR de los descriptores a los que se desee acceder.

2.14.6 `$monitor` y `$fmonitor`

```
$monitor( P1, P2, P3, ...);
$fmonitor( fd, P1, P2, P3, ...);
```

Estas funciones permiten la monitorización continua: se ejecutan una única vez, registrando aquellas variables que deseamos ver su cambio. De forma que cada vez que se produzca un cambio en una variable registrada, se imprimirá la línea completa.

2.14.7 `$dumpfile y $dumpvars`

Un formato de almacenamiento de los datos de una simulación Verilog es el **VCD** (*Verilog Change Dump*). Este formato se caracteriza porque se almacena el valor de un conjunto de variables cada vez que se produce un cambio. Suele ser empleado por los visualizadores y post-procesadores de resultados.
Para crear este fichero son necesarias dos acciones: abrir el fichero donde se almacenarán los resultados y decir qué variables se desea almacenar.

```
initial begin
  $dumpfile("file1.vcd");
  $dumpvars;
end
```

En el ejemplo anterior se desea crear un fichero cuyo nombre es `file1.vcd`, donde se desean almacenar todas las variables del diseño (que es lo que se almacena en el caso de no proporcionar parámetros).
Pueden restringirse las variables almacenadas empleando:

```
$dumpvars(levels,name1,name2,...)
```

Siendo `levels` el número de niveles de profundidad en la jerarquía que se desean almacenar, y `name1, name2, ...` las partes del diseño que se desean almacenar.
Si el primer argumento es 0, significa que se desean todas las variables existentes por debajo de las partes indicadas. Un 1 significa las variables en esa parte, pero no el contenido de módulos instanciados en su interior.

2.14.8 `$readmemb` y `$readmemh`

Permite leer la información contenida en un fichero en una memoria.

```
$readmemh(fname,array,start_index,stop_index);
$readmemb(fname,array,start_index,stop_index);
```

Donde `fname` es el nombre de un fichero que contiene:

- datos binarios (`$readmemb`) o hexadecimales (`$readmemh`) (sin especificadores de tamaño o base). Pudiendo emplearse _, x o z
- separadores: espacios, tabulaciones, retornos de carro

La información se debe de almacenar en una memory `array`, como por ejemplo:

```
reg [7:0] mem [0:1023];
```

El resto de argumentos son opcionales:

```
initial $readmemh(file.dat,mem,0);
initial $readmemh(file.dat,mem,5);
initial $readmemh(file.dat,mem,511,0);
```

En el primer ejemplo se comienza a rellenar desde el índice 0, en el segundo se comienza desde el índice 5. En el último caso se comienza por 511 y descendiendo hasta 0.
Si el número de datos del fichero es diferente al tamaño reservado o especificado para rellenar el simulador da un *warning*.

2.15 funciones y tareas

Una función es similar a una rutina en cualquier lenguaje de programación: tiene argumentos de entrada y puede retornar un valor, pero dentro de Verilog tiene un par de limitaciones:

- no se puede invocar a otra función ni tarea
- no puede contener control de tiempo (retrasos, control por eventos ni sentencias `wait`)

Su sintaxis es:

```
function <range> <fname>;
   <argumentos>
   <declaraciones>
   <funcionalidad>
endfunction
```

En la figura 7 se tiene un ejemplo de función y de su llamada. Como se puede observar la función retorna un valor asignando un nombre a una variable cuyo nombre es el de la función y cuya dimensión es `range`

```
module func_test;

function [7:0] add8;
   input [7:0] a,b;

   reg   [7:0] res;

   begin
     res=a+b;
     add8=res;
   end
endfunction

reg  [7:0]  P;

initial begin
   #(10) P=add8(100,'b101);
   $display(P);
end

endmodule
```

Figura 7: Ejemplo de función

Una tarea puede tener argumentos, pero no retorna ningún valor. Se diferencia de una función en que:

- puede invocar a otras tareas y funciones
- puede contener control de tiempo

Su sintaxis es:

```
task <tname>;
   <argumentos>
   <declaraciones>
   <funcionalidad>
endtask
```

En la figura 8 se tiene un ejemplo de tarea

```
module task_test;

parameter T=100;

reg S, R;
reg [7:0] Data;

task Send;
   input [7:0] a;

   begin
     Data=a;
     #(T) S=1;
     wait(R!=0);
     #(T) S=0;
   end
endtask

initial begin
   R=100;
   #(4*T) R=0;
   #(4*T) $finish;
end

initial begin
   #T Send(150);
end

endmodule
```

Figura 8: Ejemplo de tarea

3 Sintetizabilidad

El lenguaje Verilog tiene una doble funcionalidad:

- La realización de simulaciones de sistemas digitales: empleando dicho lenguaje para describir los estímulos que se van a aplicar, la interconexión entre los bloques, . . .
- La descripción de un sistema para su posterior síntesis.

Tener un diseño en Verilog que simule correctamente es "relativamente fácil", existiendo múltiples alternativas para conseguirlo. Sin embargo si lo que se pretende es realizar un diseño que no sólo se comporte como está previsto, sino que además sea sintetizable, deben de respetarse ciertas normas, algunas de las cuales pasamos a enumerar:

1. No emplear retrasos, dado que ello conduciría a diseños poco o imposiblemente portables de una tecnología a otra

```
‘timescale 1ns / 1ns
module kk(a,b);
input  [2:0] a;
output [2:0] b;
reg    b;

always @(a)
  #(100_000) ~b=a;

endmodule
```

   Los sintetizadores suelen ignorar estos retrasos, pudiendo haber diferencias notables entre la simulación y la síntesis.

2. No modificar una misma variable en dos procesos diferentes.

```
module kk(a,b,c);
input a,b;
output c;
reg c;

always @(b)
  if(b==0) c=0;
  else     c=1;

always @(a)
   if(a==1) c=1;

endmodule
```

   En este caso la modificación de la misma variable en dos procesos diferentes puede dar lugar a resultados diferentes de la simulación y de la síntesis (en caso de que se pueda sintetizar).

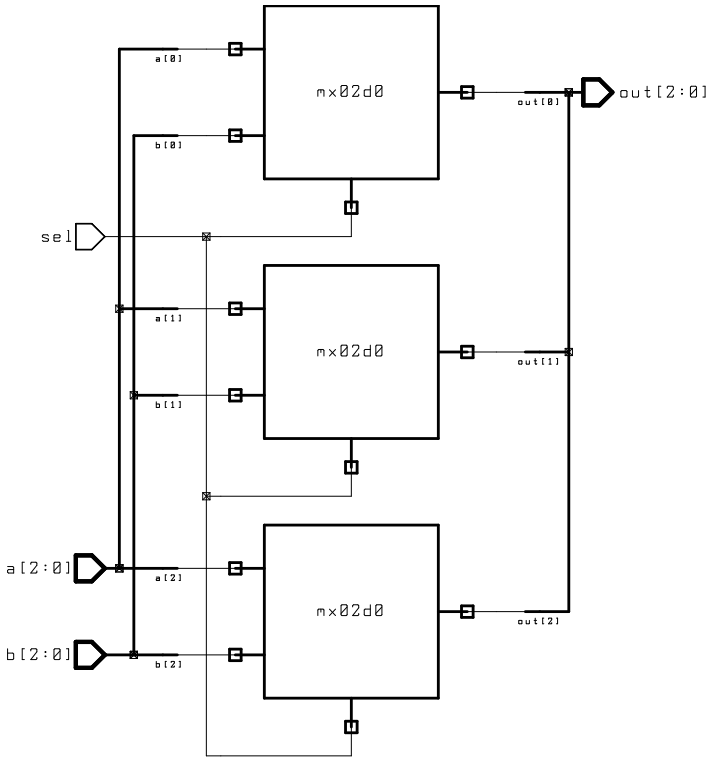
3. En una asignación procedural, todas las variables de las que dependa la asignación deben de aparecer en la lista de sensibilidad.

```
module mux(a,b,sel,out);
input  [2:0] a,b;
input  sel;
output [2:0] out;
reg    [2:0] out;

always @(sel) // incorrecto
   case (sel)
   0: out=a;
   1: out=b;
   endcase

endmodule
```

Como la lista de sensibilidad sólo contiene a sel, el sintetizador suele suponer que se desea muestrear el valor de a y b cuando se activa sel, por lo que añade biestables controlados por el nivel de sel que muestrean la salida de los multiplexores.



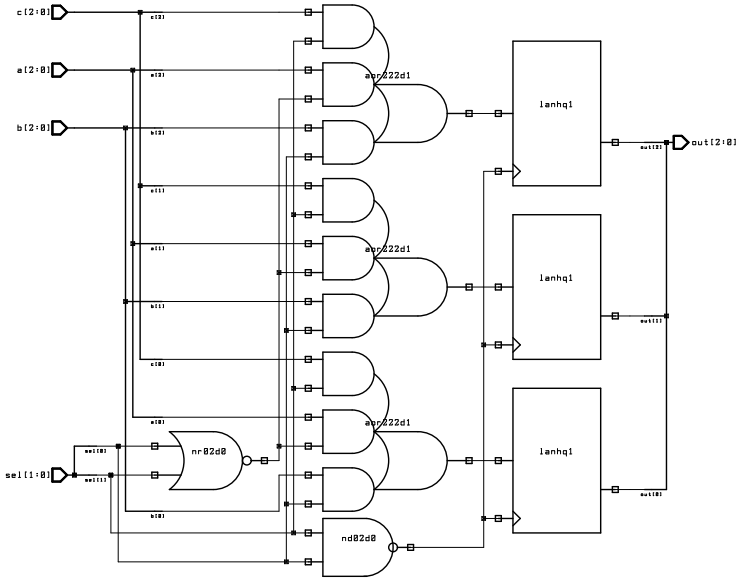
4. Si no se define completamente una variable, el lenguaje supone que conserva el último valor asignado, sintetizándose un biestable que almacene su estado.

```
module mux(a,b,c,sel,out);
input  [2:0] a,b,c;
input  [1:0] sel;
output [2:0] out;
reg    [2:0] out;

always @(sel or a or b or c)
   case (sel)
   0: out=a;
   1: out=b;
   2: out=c;
   endcase

endmodule
```

En este caso se han sintetizado unos biestables que mantengan la salida en el caso de que $sel = 3$



5. Los sintetizadores no suelen admitir procesos initial.

6. Los sintetizadores no suelen admitir los operadores división y resto.

7. Debe tenerse precaución con el problema de carrera (*race condition*), que se produce cuando dos asignaciones se realizan en el mismo instante pero una depende de la otra, y por tanto el orden de ejecución es importante. Supongamos que queremos retrasar una señal `C` dos ciclos de reloj, una posibilidad sería:

```
module race(CLK,C,ddC);
input  CLK, C;
output ddC;

reg    dC, ddC;


always @(posedge CLK) ddC=dC;

always @(posedge CLK) dC=C;

endmodule
```

Como los dos procesos se ejecutan en paralelo y la variable `dC` es modificada y empleada, el código se ve influenciado por el orden. Una alternativa sería:

```
module race(CLK,C,ddC);
input  CLK, C;
output ddC;

reg    dC, ddC;


always @(posedge CLK) begin
   ddC=dC;
   dC=C;
end

endmodule
```

Donde la asignación con bloqueo fija el orden deseado.

4 Máquinas de estado

Una de las aplicaciones más frecuentes es la realización de una máquina de estados. Propondremos un método para la estructuración de una máquina de estado, empleando tres procesos:

- preparación asíncrona del siguiente estado, partiendo de las entradas asíncronas de nuestro sistema y del estado en el que nos encontremos.
- asignación síncrona, donde en el flanco de reloj hacemos que el estado actual sea el que hemos estado preparando.
- asignación de las salidas. En este apartado debemos de distinguir dos tipos: salidas que sólo dependen del estado y salidas que dependen del estado y de las variables de entrada

  Nosotros nos ceñiremos a las primeras, aunque es fácil ver cómo sería el segundo caso.

(EJEMPLO):
Tengamos una señal S que queremos detectar su flanco de subida y generar un pulso P de un periodo de reloj.

```
module risepulse(BCLK, BnRES, S, P);
input BCLK;
input BnRES;

input  S;
output P;

reg P;

reg  [1:0] Estado, nEstado;

always @(Estado or S)
  case(Estado)
  0: // Valor 0
     if(S==1) nEstado=2;
     else     nEstado=0;
  1: // Valor 1
     if(S==0) nEstado=3;
     else     nEstado=1;
  2: // Flanco positivo
     nEstado=1;
  3: // Flanco negativo
     nEstado=0;
  endcase

always @(posedge BCLK or negedge BnRES)
  if(BnRES==0) Estado=0;
  else         Estado=nEstado;

always @(Estado)
   if(Estado==2) P=1;
   else          P=0;

endmodule
```

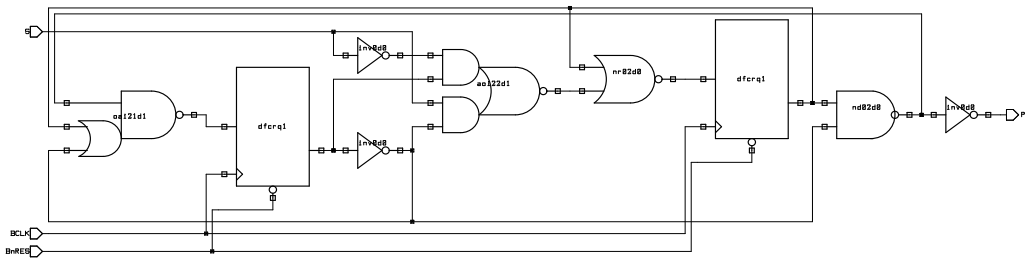


Figura 9: Máquina de estados para generar pulso con cada flanco de subida de la señal S

5 Ejemplos

5.1 De-Multiplexor

Veamos como ejemplo un de-multiplexor que está decodificando un bus de direcciones:

- $A \in (`h00\_00, `h0f\_ff)$ activa el bit de selección 0
- $A \in (`h10\_00, `h1f\_ff)$ activa el bit de selección 1
- $A \in (`h20\_00, `h2f\_ff)$ activa el bit de selección 1

Obsérvese que el proceso tiene en su lista de sensibilidad sólo la variable A, debido a que la asignación sólo depende de ella.

```
module demuxA(A,SEL);
input  [15:0] A;
output [ 2:0] SEL;

reg    [ 2:0] SEL;

always @(A)
   case(A[15:12])
   0: SEL='b001;
   1: SEL='b010;
   2: SEL='b100;
   default: SEL=0;
   endcase

endmodule
```

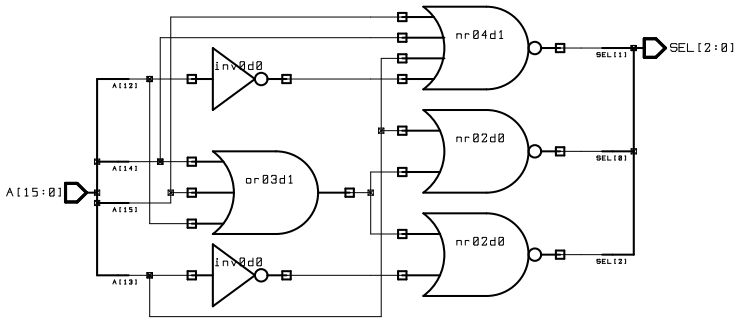


Figura 10: Decodificador de un bus de direcciones

5.2 Multiplexor

La potencialidad del lenguaje permite que un multiplexor de buses sea fácilmente de describir. Como ejemplo tenemos un multiplexor de tres buses de 24 bits, donde hemos supuesto que la opción de selección `2'b11` da una salida con todos los bits a cero.

5.3 Registro de desplazamiento

En este ejemplo se puede comprobar la compacidad del código, obsérvese que en una única asignación estamos insertando el bit de entrada y realizando el desplazamiento de todos los bits.

```
module muxA(Out,A,B,C, SEL);
input  [ 1:0] SEL;
input  [23:0] A, B, C;
output [23:0] Out;

reg    [23:0] Out;

always @(SEL or A or B or C)
  case(SEL)
  'b00: Out=A;
  'b01: Out=B;
  'b10: Out=C;
  default: Out=0;
  endcase

endmodule
```

Figura 11: Multiplexor de 3 buses de 24 bits

```
module regdes(BCLK, BnRES, A, B);
input BCLK;
input BnRES;
input  A;
output B;

reg [3:0] C, nC;
assign B=C[3];

always @(posedge BCLK or negedge BnRES)
   if(BnRES==0) C=0;
   else         C=nC;

always @(C or A)
   nC={C,A};

endmodule
```

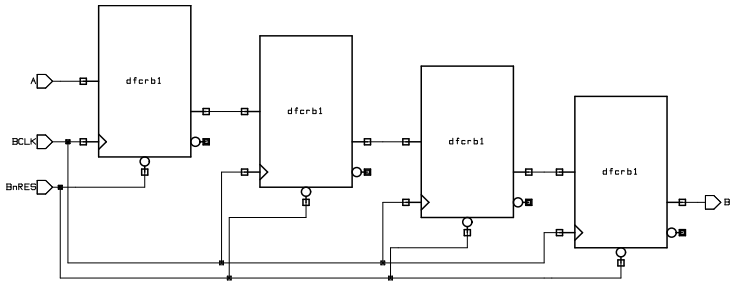


Figura 12: Registro de desplazamiento

5.4 Contador

Comencemos por un simple contador que incrementa el valor de un registro interno con cada flanco de reloj. Obsérvese que existen dos procesos:

- uno que asigna el valor del registro interno en cada flanco de reloj así como realiza la inicialización asíncrona.
- otro que prepara el siguiente valor que tendrá el registro

```
module contA(BCLK,BnRES,VAL);
input BCLK;
input BnRES;
output [3:0] VAL;

reg [3:0] VAL, nVAL;

always @(posedge BCLK or negedge BnRES)
   if(BnRES==0) VAL=0;
   else         VAL=nVAL;

always @(VAL)
   nVAL=VAL+1;

endmodule
```

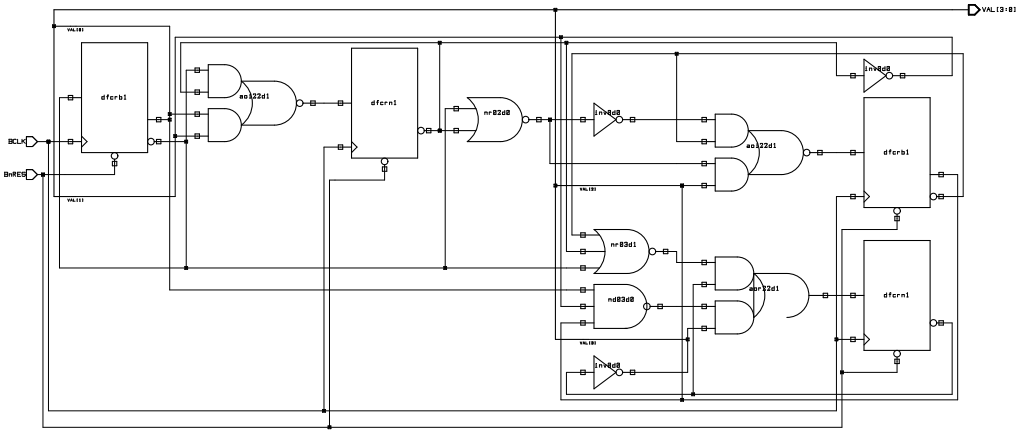


Figura 13: Contador Ascendente

Veamos las modificaciones para que permita que la cuenta pueda ser ascendente/descendente:
Obsérvese que tanto en un caso como en el otro se produce desbordamiento (`8'b1111` al incrementar pasa a `8'b0000`, mientras que `8'b0000` decrementado pasa a `8'b1111`)

5.5 Acceso bidireccional

Anteriormente hemos comentado que una señal de entrada/salida debe de ser de tipo `wire`, por lo que si queremos que la salida sea una señal asignada proceduralmente, deberemos disponer de una variable intermedia de tipo `reg` y realizar una asignación continua.
En el ejemplo de la figura 15 se observa que la variable `BD` es de entrada/salida[4]. `BDO`, que es la

[4] siendo de tipo `wire` que es el tipo por omisión

```
module contB(BCLK,BnRES,UD,VAL);
input BCLK;
input BnRES;
input UD;
output [3:0] VAL;

reg [3:0] VAL, nVAL;

always @(posedge BCLK or negedge BnRES)
   if(BnRES==0) VAL=0;
   else         VAL=nVAL;

always @(VAL or UD)
if(UD==1) nVAL=VAL+1;
else      nVAL=VAL-1;

endmodule
```

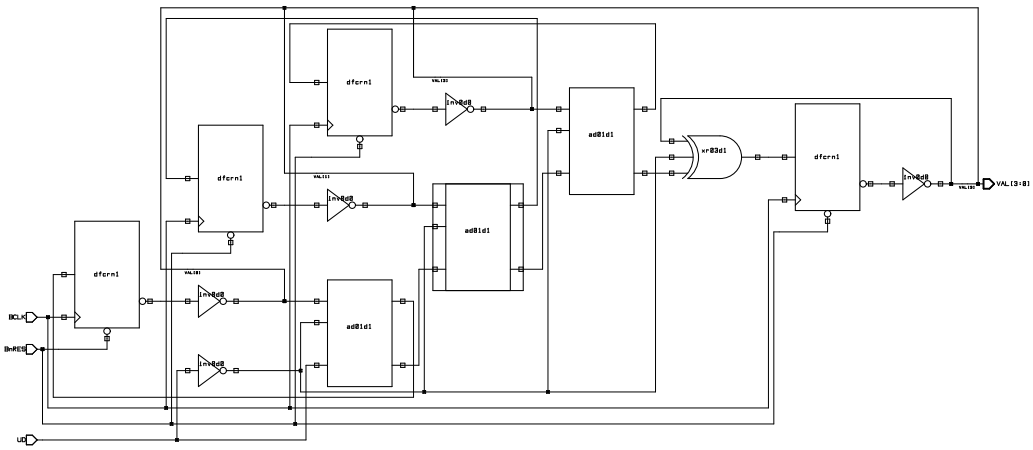


Figura 14: Contador Ascendente/Descendente

salida del bus, es una variable de tipo `reg`, que a continuación asignamos continuamente con la señal `BD`.
Obsérvese que la señal `WRITE` controla la dirección de funcionamiento del bus bidireccional: $WRITE = 1$ significa que el exterior quiere escribir en mi módulo, por lo que debe de colocarse el bus de salida en triestado, mientras que $WRITE = 0$ es una petición de lectura del registro interno, por lo que: $BDO = C$
Cuando se quiere emplear el bus como entrada se emplea directamente BD, como es el caso de la precarga síncrona del ejemplo: $nC = BD$.

```
module iocont(BCLK,BnRES,WRITE,BD);
input BCLK;
input BnRES;
input WRITE;
inout [7:0] BD;

reg   [7:0] BDO;  assign BD=BDO;
reg   [7:0] C, nC;


always @(WRITE or C)
if(WRITE==1) BDO='bz;
else         BDO=C;

always @(posedge BCLK or negedge BnRES)
if(BnRES==0) C=0;
else         C=nC;

always @(WRITE or C or BD)
if(WRITE==1) nC=BD;
else         nC=C+1;

endmodule
```

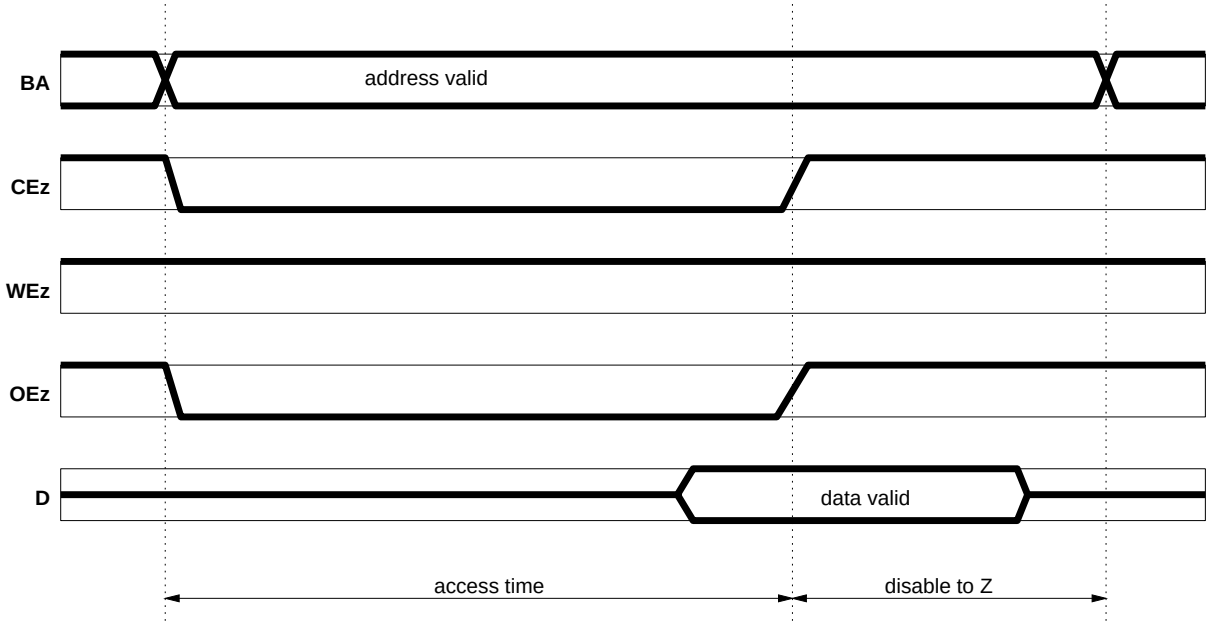
Figura 15: Contador con entrada salida bidireccional. Cuando se lee ($WRITE = 0$) se obtiene el valor actual, cuando se escribe ($WRITE = 1$) se precarga un determinado valor.

5.6 Memorias

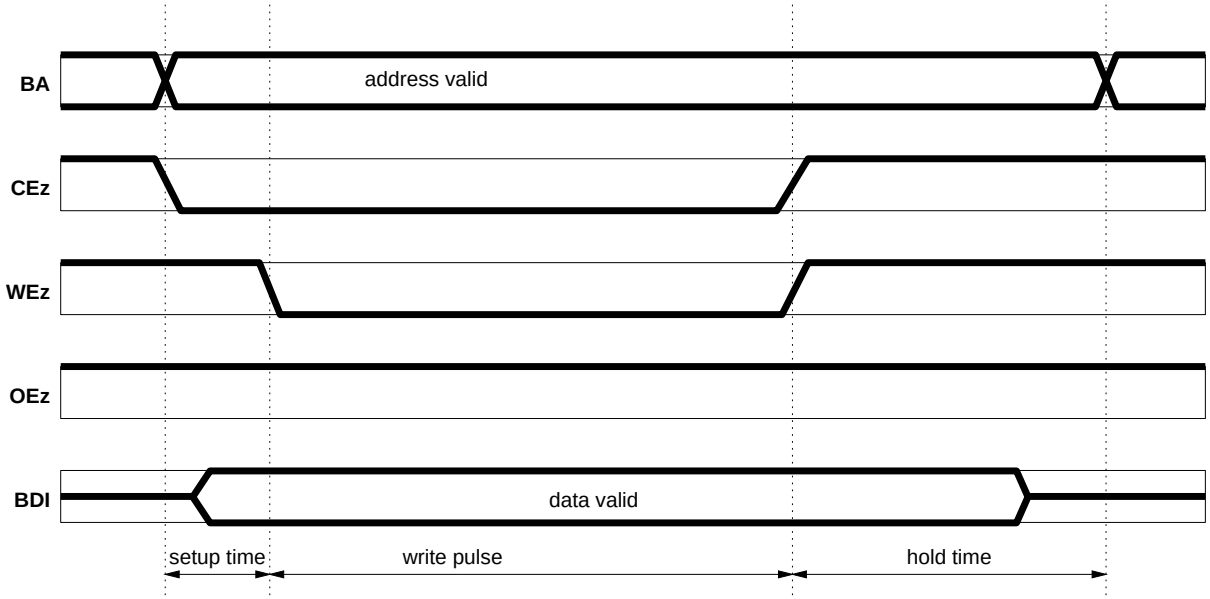
Veamos a continuación la descripción funcional de una memoria. Dispondremos de las siguientes señales:

- **A** Bus de direcciones
- **D** Bus de datos.
- **OEz** Habilitación de la salida (*Output Enable*)
- **CEz** Habilitación del dispositivo (*Chip Enable*)
- **WEz** Habilitación de escritura (*Write Enable*)

Las señales de habilitación suelen funcionar con lógica negada, estando la última señal (WEz) disponible sólo en memorias que permitan escritura.
En la figura 16 se muestra los cronogramas de los diferentes accesos.
Veamos como ejemplo una memoria RAM de 32kx8 bits (figura 17(a)).



(a) Acceso en lectura



(b) Acceso en escritura

Figura 16: Cronogramas de los accesos a memorias en lectura/escritura

Cuando las condiciones de lectura se verifican (`((CEz==0)&&(OEz==0)&&(WEz==1))`), se presenta el dato solicitado en el bus de datos, tras un retraso:

```
regD <= #(AT) memo[A];
```

En cuanto dejan de verificarse las condiciones para la lectura (tras un cierto retraso), el bus de datos vuelve a ser triestado.
En cuanto las condiciones de escritura se verifican (`((CEz==0)&&(OEz==0)&&(WEz==0))`), se almacena el dato en la memoria interna tras un retraso:

```
#(AT) memo[A]=D;
```

El modelo de una ROM (figura 17(b)) es muy parecido al de la RAM, salvo que no tiene acceso en escritura, y inicialmente se precarga la memoria con valores previamente almacenados en un fichero: `ini.dat`

```
module RAM8 (A, D, CEz, OEz, WEz);

parameter   ST= 1_000;  // Setup time  (1ns)
parameter   AT=10_000;  // Access time (10ns)
parameter   HT= 1_000;  // Hold time   (1ns)

input [14:0] A;
inout [ 7:0] D;

input CEz, OEz, WEz;

reg   [ 7:0] memo [0:32767];

reg   [ 7:0] regD;   assign D=regD;
reg   access;

initial begin
  access=0; regD='bz;
end

// Lectura
always @(CEz or OEz or WEz or A)
if ((CEz==0)&&(OEz==0)&&(WEz==1)) begin
   regD <= #(AT) memo[A]; access=1;
end
else begin
   if(access==1) begin
     regD<= #(HT) 'bz; access=0;
   end
end

// Escritura
always @(CEz or OEz or WEz or A or D)
if((CEz==0)&&(OEz==0)&&(WEz==0))
    #(AT) memo[A]=D;

endmodule
```

(a) Código de una RAM 16k x 8

```
module ROM8 (A, D, CEz, OEz);

parameter   ST= 1_000;  // Setup time  (1ns)
parameter   AT=10_000;  // Access time (10ns)
parameter   HT= 1_000;  // Hold time   (1ns)
parameter   Data="ini.dat";

input [14:0] A;
inout [ 7:0] D;

input CEz, OEz;

reg   [ 7:0] memo [0:32767];

reg   [ 7:0] regD;   assign D=regD;
reg   access;

initial begin
  access=0; regD='bz;
  $readmemh(Data,memo,0);
end

// Lectura
always @(CEz or OEz or A)
if ((CEz==0)&&(OEz==0)) begin
   regD <= #(AT) memo[A]; access=1;
end
else begin
   if(access==1) begin
     regD<= #(HT) 'bz; access=0;
   end
end

endmodule
```

(b) Código de una ROM 16k x 8

Figura 17: Códigos de memorias

Índice de Materias