LENGUAJE DE DESCRIPCIÓN DE HARDWARE VERILOG

FACULTAD DE INGENIERÍA
DIVISIÓN DE INGENIERÍA ELÉCTRICA
DEPARTAMENTO DE INGENIERÍA EN COMPUTACIÓN
E INGENIERÍA EN ELECTRÓNICA

ING. JORGE VALERIANO ASSEM
ING. NORMA ELVA CHÁVEZ RODRÍGUEZ

CIUDAD UNIVERSITARIA, JUNIO 2001

PRESENTACIÓN

La Facultad de Ingeniería ha decidido realizar una serie de ediciones provisionales de obras recientemente elaboradas por académicos de la institución, como material de apoyo para sus clases, de manera que puedan ser aprovechadas de inmediato por alumnos y profesores. Tal es el caso del Manual de consulta *Lenguaje de descripción de Hardware Verilog*, de los Ingenieros Norma Elva Chávez Rodríguez y Jorge Valeriano Assem.

Se invita a los estudiantes y profesores a que comuniquen a los autores las observaciones y sugerencias que mejoren el contenido de la obra, con el fin de que se incorporen en una futura edición definitiva.

ÍNDICE

1. INTRODUCCIÓN

Con las complejidades de los chips actuales, cerca de 50 millones de dispositivos activos, ya no es posible el diseño de hardware usando métodos basados en esquemáticos. Así inevitablemente los diseños deben ahora utilizar exclusivamente especificaciones de alto nivel y procesos de síntesis.

De esta forma actualmente se dispone de los llamados HDL, que son lenguajes de descripción de hardware los cuales nos permiten describir hardware utilizando especificaciones de alto nivel; es decir utilizando un lenguaje. Haciendo una analogía los HDLs son lenguajes de alto nivel que nos sirven para crear hardware; así como el lenguaje "C" nos sirve para generar software.

Los HDLs disponibles actualmente nos permiten diferentes formas de especificar un diseño; a estas formas se les conoce como *estilos de escritura*, los cuales van desde especificaciones puramente funcionales hasta especificaciones estructurales (lo mismo que un esquemático).

Así entonces tenemos que **Verilog** es uno de estos lenguajes de descripción de hardware. *Verilog* es preferido por la mayoría de los diseñadores comerciales en América y en Japón. La popularidad de Verilog se puede deber a que muchos usuarios comerciales se sienten más en casa con la sintaxis de Verilog, la cual deriva o es muy parecida a la sintaxis de "C" que es ampliamente conocida. También se puede deber a que Verilog es un lenguaje más pequeño y sencillo que por ejemplo VHDL.

Inicialmente Verilog fue creado por Gateway Technologies, posteriormente fue apoyado por la empresa Cadence y actualmente se ha convertido en un estándar de IEEE 1364. Debemos destacar dos aspectos importantes, en primer lugar Verilog nos sirve para diseñar solamente circuitos digitales y en segundo lugar las herramientas comerciales que existen y aceptan Verilog, sintetizan solo un subconjunto del lenguaje. Estas ideas se muestran en las Fig. 1 y Fig. 2.

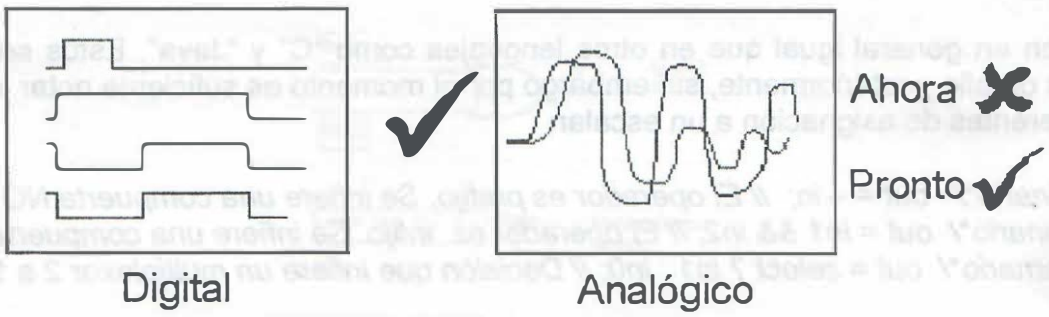


Fig. 1. Hoy podemos diseñar solamente circuitos digitales con Verilog.

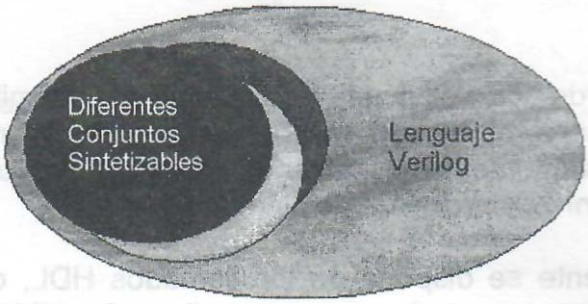


Fig. 2. Las herramientas actuales sintetizan diferentes conjuntos del lenguaje Verilog.

2. SINTAXIS DE LOS CONSTRUCTORES BÁSICOS DEL LENGUAJE VERILOG.

La sintaxis de Verilog es muy similar a la de "C" aunque la semántica es diferente. Verilog tiene muchos mecanismos para concurrencia indispensable para el modelado de hardware.

2.1 Elementos básicos.

Verilog tiene un conjunto de elementos básicos en su sintaxis que son muy parecidos a lenguajes de alto nivel, en especial a "C" y "Java".

2.2 Identificadores.

Los identificadores en Verilog son los nombres dados a objetos de hardware en un diseño, los cuales pueden ser alambres (buses), registros, memorias ó módulos. Estos nombres son creados de caracteres numéricos y alfabéticos. El primer carácter no debe ser numérico o "$" que tiene un significado especial en Verilog. Igual que en "C" los comentarios de una línea se pueden hacer utilizando "//" y los comentarios multilínea se pueden hacer utilizando el par " /* */" para inicio y fin del comentario.

2.3 Operadores.

Estos son en general igual que en otros lenguajes como "C" y "Java". Estos se verán con más detalle posteriormente, sin embargo por el momento es suficiente notar los tres tipos diferentes de asignación a un escalar:

```
/*Ejm. Unario*/  out = ~ in;  // El operador es prefijo. Se infiere una compuerta NOT
/*Ejm. Binario*/ out = in1 && in2; // El operador es  infijo. Se infiere una compuerta AND
/*Ejm Ternario*/ out = select ? in1 : in0; // Decisión que infiere un multiplexor 2 a 1
```

Cabe hacer notar que todas las asignaciones hechas en Verilog terminan con ";" al igual que en "C".

2.4 Valores.

El conjunto de valores disponibles en Verilog son [0,1,x,z] los cuales corresponden a los valores de señal que pueden ser soportados en forma escalar. Estos valores evidentemente tienen que ver con los fenómenos eléctricos en los circuitos considerándolos a bajo nivel.

Eléctricamente "0" lo utilizamos para representar un valor lógico falso o cero volts. El "1" lo utilizamos para representar un valor lógico verdadero o 5 volts TTL. La "z" eléctricamente corresponde a un nodo o alambre que está aislado o desconectado de cualquier otra región del circuito, es decir se trata de un circuito abierto. La "x" eléctricamente corresponde a un nodo o alambre del cual no se tiene información para decidir si es manejado por un valor alto (1) o bajo (0).

2.5 Expresiones.

Como en el lenguaje "C", las expresiones son creadas a partir de identificadores, operadores y constantes.

2.6 Tipos de datos.

Los tipos de datos disponibles en Verilog son **net, reg, integer, real, time, parameter** y **event**, estos tipos se explican a continuación.

2.6.1 Nets (Redes o alambres).

Se representan por la palabra reservada "**wire**". Para propósitos de síntesis el único importante es el **wire** (alambre). Como su contraparte física un alambre puede tener un valor sobre él solamente si es manejado continuamente por algo. Cabe hacer notar que la palabra clave aquí es "continuamente". De cualquier otra forma adquiere "z" por default . De esta manera el tipo "net" nos representa físicamente alambres que para tener algún valor deben ser manejados continuamente por ejemplo por la salida de una compuerta lógica. Esto se ilustra en la Fig. 3.

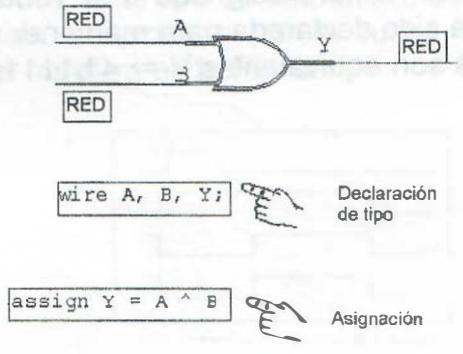


Fig. 3. Tipo de dato NET

2.6.2 Registros.

Se representan por la palabra reservada "**reg**". Este tipo corresponde con el concepto de almacenamiento estático en circuitos VLSI. Un tipo "**reg**" no es necesariamente un registro controlado por un reloj en respuesta a una declaración "**reg**", simplemente debe ser entendido como un tipo de dato que es capaz de recibir y mantener un cierto valor, es decir a diferencia del tipo "**wire**" no debe ser manejado continuamente para que tenga un valor.

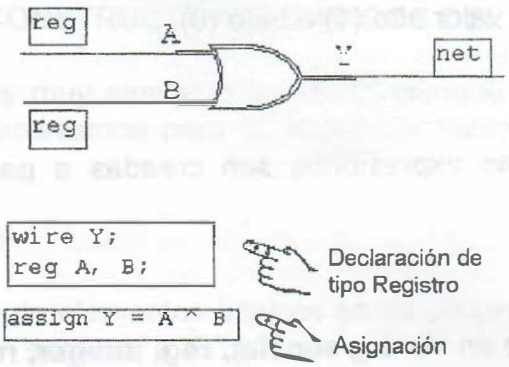


Fig. 4. Tipo de dato REG

2.6.3 Enteros.

Los enteros son especificados de la forma más general como:

Size ' base number

La palabra "size" indica el número de bits utilizados para la representación del número entero. El apóstrofo es parte de la sintaxis. "base" indica la base numérica en la que será representado el número entero, esta puede ser decimal, octal, hexadecimal o binaria (d,o,h,b) y "number" es el número asignado a la variable entera en la "base" especificada. Por ejemplo "V" ha sido declarada para mantener cuatro bits de forma que las siguientes tres asignaciones son equivalentes:V = 4'b1111; V = 4'hf; V = 4'd15. La Fig. 5 muestra esta idea.

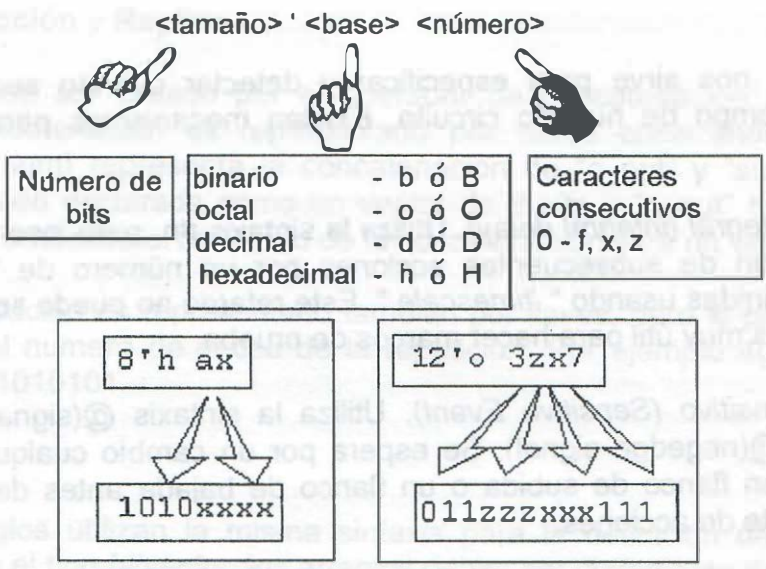


Fig. 5. Representación del tipo entero.

2.6.4 Reales.

El tipo real aunque puede ser declarado e inicializado no es posible sintetizarlo en hardware razón por la cual no se detalla este tipo.

2.6.5 Time.

Este es un tipo de dato especial que al igual que el tipo "real" no es posible sintetizarlo en hardware, sin embargo es un tipo de dato utilizado por el simulador para almacenar el tiempo actual. Las unidades del tiempo utilizado son especificadas con la sentencia:

'timescale 100 ns / 100 ps

El primer número indica las unidades utilizadas en este caso 100 nanosegundos donde un nanosegundo=10^{-9} segundos, y el segundo número indica el dígito menos significativo a ser impreso en este caso 100 picosegundos, donde un picosegundo=10^{-12} segundos. Este tipo de dato utiliza 64 bits para su representación. El tiempo puede ser referido o invocado por la rutina "$time" en la mayoría de los simuladores. Esta idea se muestra en la Fig. 6.

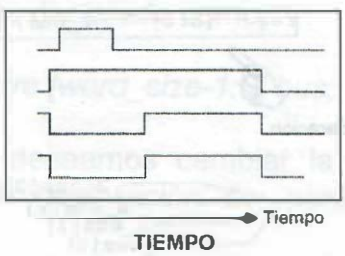


Fig. 6. Representación del tipo Time.

2.6.6 Eventos.

Es un tipo que nos sirve para especificar y detectar cuando sucede algo en la dimensión del tiempo de nuestro circuito. Existen mecanismos para especificar los eventos.

- *Retardo integral (integral delay).* Utiliza la sintaxis #n, para insertar un retraso en la activación de subsecuentes acciones por un número de "n" unidades de tiempo definidas usando " *'timescale* ". Este retardo no puede ser sintetizado, sin embargo es muy útil para hacer marcos de prueba.

- *Evento sensitivo (Sensitive Event).* Utiliza la sintaxis @(signal) ó @(posedge signal) ó @(negedge signal). Se espera por un cambio cualquiera en la señal indicada, un flanco de subida o un flanco de bajada antes de activar la serie subsiguiente de acciones.

2.6.7 Bitvector.

Los tipos "wire" y "reg" son escalares por default, sin embargo pueden ser declarados en versiones multibit agregando un intervalo antes del identificador. El intervalo es declarado en la forma:

[expresion1 : expresion2]

donde las expresiones son valores constantes no negativos. No importando cual de las dos expresiones tiene el número más grande, Verilog siempre considera el valor izquierdo como el bit más significativo y el valor derecho como el bit menos significativo. En la Fig. 7 se muestra esta idea, y a continuación mostramos el siguiente código:

```
wire [3:1] tribit;      //declara un vector de tres bits tipo wire
wire onebit;            //declara un bit tipo wire
wire twobit;            //declara un bit tipo wire
wire wobit;             //declara un bit tipo wire

onebit = tribit[2];     //selecciona el bit de en medio
twobit = tribit[1:0];   //ilegal porque trata de elegir un bit fuera del intervalo de declaración
wobit = tribit[1:2];    //ilegal porque trata de seleccionar un subcampo indexado al revés
```

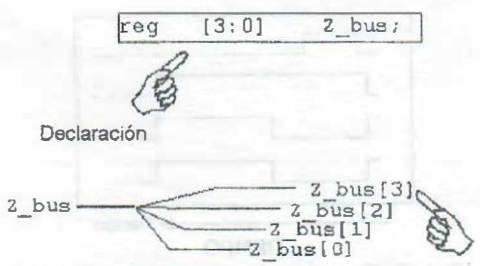


Fig. 7. Declaración de vectores.

2.6.8 Concatenación y Replica.

Un *bitvector* puede ser creado por el operador de concatenación o replicación. El operador de concatenación es representado por llaves encerrando una lista, por ejemplo: {c_out, sum} representa la concatenación de "c_out" y "sum" donde "sum" previamente ha sido declarada como un vector de 8 bits y "c_out" ha sido declarado como un escalar; entonces el resultado de la concatenación será un vector de 9 bits.

El operador replicación es representado también por llaves, pero le precede un entero que representa el numero de veces de la repetición. Por ejemplo 4{2'b01} genera el vector de 8 bits 01010101.

2.6.9 Arreglos.

Aunque los arreglos utilizan la misma sintaxis para la definición de los índices del intervalo como en el tipo *bitvector*, los arreglos deben ser claramente distinguidos de los vectores. Para propósitos de síntesis los arreglos serán utilizados únicamente con el tipo de dato *reg*. Tal tipo de arreglo es equivalente a un bloque de memoria. Por ejemplo:

```
reg [15:0] ram [0:255]
```

Declara una memoria de 256 palabras cada una de 16 bits. La invocación *w = ram[27];* adquiere la palabra veintisiete de la memoria ram. Efectivamente un decodificador es inferido además del arreglo para el almacenamiento de la información. Como en el mundo real la extracción de un subcampo de una palabra de *ram* requiere la invocación de pasos sucesivos:

```
tmp = ram[27];
y = tmp[3:0];
```

2.6.10 Parámetros.

Los parámetros son utilizados para definir constantes en los módulos. Por ejemplo:

```
parameter word_size = 16, pulse_width = 20, capacity = 256;
```

después de esto un bus puede ser declarado como:

```
wire [word_size-1:0] bus;
```

De esta manera si más tarde deseamos cambiar la dimensión solo es necesario modificar una sola línea (la de declaración del parámetro) en vez de todas las declaraciones.

2.6.11 Directivas del Preprocesador.

Verilog tiene un preprocesador similar al de "C", solo que en Verilog la sintaxis inicia con el apóstrofo. Un ejemplo de una instrucción al preprocesador es:

'define word_size 16

esto le indica al preprocesador que cada que encuentre *'word_size* en las subsecuentes líneas de texto lo reemplazará por un 16.

La directiva al preprocesador *'include* también está disponible. En Verilog nos sirve para incluir el código fuente de submódulos frecuentemente utilizados, por ejemplo:

'include submodulo.v

incluye en el módulo actual el código fuente del submódulo.

2.7 Módulos.

Los sistemas de hardware al igual que otros tipos de sistemas se pueden particionar en módulos. En Verilog un módulo nos permite expresar una pieza de hardware independiente que va a interactuar con otros módulos en un nivel de jerarquía más elevado. En Verilog un módulo es expresado en el siguiente formato. La Fig. 8 muestra el concepto gráfico de un módulo.

module modulename(port list);
Parámetros
declaración de puertos
declaración de buses
declaración de registros
instanciación de submódulos
...cuerpo del modulo ...
endmodule

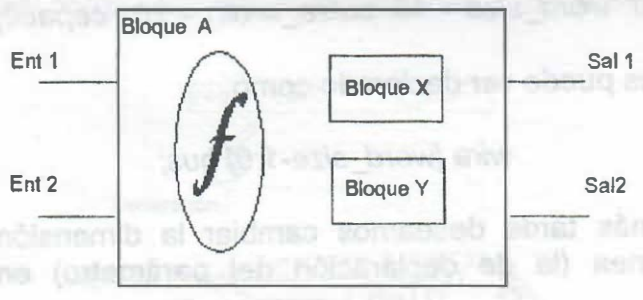


Fig. 8. Representación gráfica del módulo.

Un ejemplo simple de un modulo es la definición del sumador completo (full-adder) como se muestra a continuación:

```
module fulladder(a, b, c_in, sum, c_out); // suma operandos de un solo bit
  input a, b, c_in;
  output sum, c_out;
    assign {c_out, sum} = a + b + c_in;    //cuerpo del programa
endmodule
```

El nombre del módulo debe ser el mismo que el nombre del archivo que contiene el módulo con extensión ".v". En el ejemplo anterior después del nombre del módulo aparece la lista de todo el conjunto de entradas y salidas del mismo. Es decir estamos definiendo la interfase del módulo. A continuación mediante las palabras reservadas *"input"* y *"output"* definimos cuales son entradas y cuales son salidas del módulo. Lo que sigue es el cuerpo del modulo, constituido por todas las sentencias que definen el funcionamiento del mismo. Todo modulo finaliza con la palabra reservada "endmodule".

El siguiente ejemplo es un sumador en cascada de 4 bits el cual crea cuatro instancias del módulo sumador completo definido anteriormente. El diagrama se ilustra en la Fig. 9 y su código es el siguiente:

```
module add_4_r (A, B, C_in, SUM, C_out);
  input [3:0] A, B;
  input C_in;
  output [3:0] SUM;
  output C_out;
  wire [3:0]  A, B, C_in, SUM, C_out;
  fulladder FA3(A[3], B[3], C2, SUM[3], C_out);
  fulladder FA2(A[2], B[2], C1, SUM[2], C2);
  fulladder FA1(A[1], B[1], C0, SUM[1], C1);
  fulladder FA0(A[0], B[0], C_in, SUM[0], C0);
endmodule
```

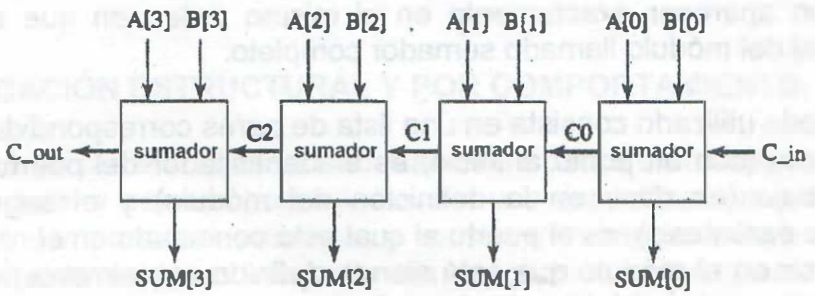


Fig. 9. Sumador en cascada de cuatro bits.

Las instanciaciones del sumador completo son físicamente piezas de hardware diferentes con la misma funcionalidad, por lo tanto requerimos identificadores diferentes para cada uno de ellos tales como FA3, FA2, FA1 y FA0. Los alambres C0, C1 y C2 sirven para indicar como estos submódulos están interconectados como se muestra en la Fig. 9. Finalmente se debe notar que todos los submódulos son concurrentes en activación, es decir, trabajan en paralelo no importando el orden en el cual fueron escritos. Recuerde que estamos escribiendo hardware no software.

2.7.1 Reglas para la conexión de módulos

Una salida debe ser manejada por un tipo **net** proveniente de un **wire** o la salida de una compuerta o por un tipo **reg** proveniente de alguna salida de un flip-flop. Por otra parte los puertos de entrada es recomendable que siempre sean del tipo **nets**. La situación es ilustrada en la Fig.10. Otra forma de expresar esta regla es que aunque se pueden conectar varias entradas juntas, en general es mala idea conectar salidas juntas. La excepción a esta regla es cuando la salidas pueden tener estado de alta impedancia en cuyo caso estamos construyendo por ejemplo buses tres estados.

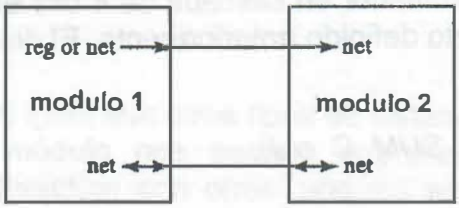


Fig. 10. Conexión de módulos.

2.7.2 Lista de puertos.

Verilog tiene dos maneras para expresar los argumentos de la lista de puertos en las instanciaciones del módulo. La primera es simplemente ordenar la lista como se uso en el ejemplo anterior. Este método es económico y útil para módulos desde mediana hasta pequeña complejidad en los pines de entrada-salida. Sin embargo este método tiene la desventaja de que los identificadores de los puertos de las instancias FA3, FA2, FA1, FA0 deben aparecer exactamente en el mismo orden en que aparecen en la definición original del módulo llamado sumador completo.

El segundo método utilizado consiste en una lista de pares correspondidos. En cada par el primer elemento (con un punto al inicio) es el identificador del puerto en el nivel de jerarquía más bajo (es decir en la definición del módulo) y el segundo elemento (encerrado entre paréntesis) es el puerto al cual está conectado en el nivel de jerarquía más alto (es decir en el módulo que está siendo definido actualmente). Por ejemplo la última instanciación en le módulo *add_4_r* definido anteriormente puede ser como:

FA0(.a(A[0]), .b(B[0], .c_out(C0), .sum(SUM[0]), .c_in(C_in));

Este método es mejor para módulos que tienen muchos puertos porque de esta manera el orden en el que aparezcan los pares correspondidos no importa.

2.7.3 Nombres Jerárquicos.

Los módulos utilizados en un diseño pueden ser pensados como una jerarquía ordenada de niveles. Por ejemplo, si la especificación de un marco de prueba (testbench) es utilizado para estimular el módulo *add_4_r* definido anteriormente, entonces la jerarquía estará compuesta de tres niveles como se ilustra en la Fig. 11.

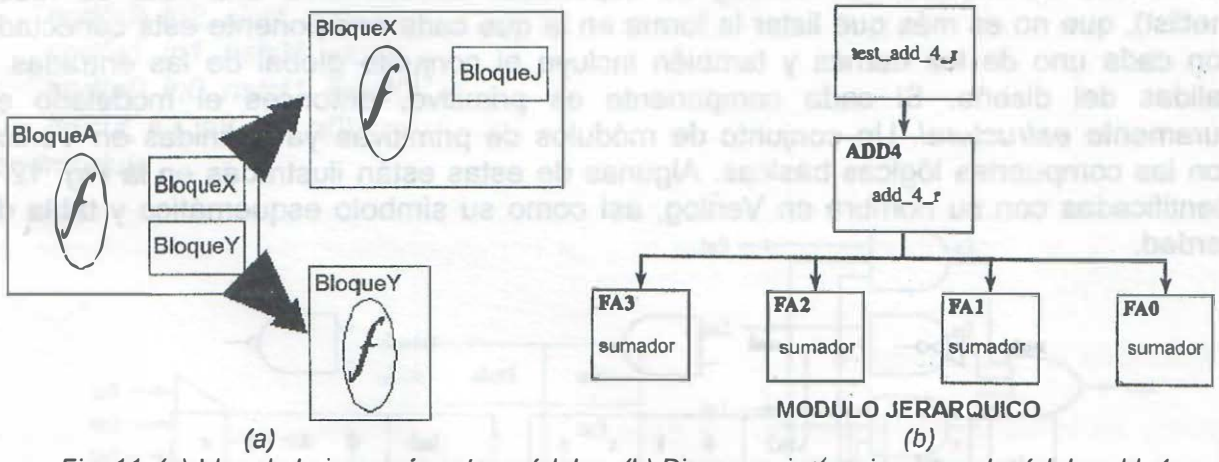


Fig. 11. (a) Idea de la jerarquía entre módulos. (b) Diagrama jerárquico para el módulo add_4_r

Sobre el nivel más alto de la jerarquía se puede utilizar la notación de punto para permitirle a Verilog comunicarse con los objetos varios en los diversos módulos a diferentes niveles de jerarquía. Para el ejemplo actual del sumador la identificación del acarreo de entrada a diferentes niveles puede ser caracterizado como sigue:

Nivel 3 (más alto): *C_IN*
Nivel 2 (más alto para la síntesis): *ADD4.C_in*
Nivel 1 (más bajo): *ADD4.FA0.c_in*

3. ESPECIFICACIÓN ESTRUCTURAL Y POR COMPORTAMIENTO.

En esta sección se van a analizar dos formas de introducir un diseño en Verilog que son la forma estructural y por comportamiento. En una especificación *estructural* se listan únicamente los componentes primitivos y como están interconectados conjuntamente, es decir únicamente estamos haciendo una descripción del esquemático del diseño y técnicamente no hay diferencia entre esto y un diseño gráfico. En el método de especificación por *comportamiento* solo se muestra o se describen los pasos *funcionales* a través de los cuales las salidas son calculadas en términos de las entradas. La descripción por comportamiento puede ser hecha usando expresiones de asignación y expresiones lógicas. De esta manera cualquier diseño puede ser descrito

estrictamente en forma *estructural* o por *comportamiento*. Por supuesto es posible escribir diseños con una mezcla de estos dos métodos.

En esta sección se comienza con módulos completamente combinacionales para mostrar el método de especificación estructural. A la mitad de la sección se continua con algo llamado especificación *"dataflow"* la cual es un tipo simple del método de descripción por comportamiento.

3.1 Compuertas básicas.

Una forma de modelar en Verilog es especificar directamente una lista de redes (netlist), que no es más que listar la forma en la que cada componente está conectado con cada uno de los demás y también incluye el conjunto global de las entradas y salidas del diseño. Si cada componente es primitivo, entonces el modelado es puramente *estructural*. Un conjunto de módulos de primitivas ya definidas en Verilog son las compuertas lógicas básicas. Algunas de estas están ilustradas en la Fig. 12 e identificadas con su nombre en Verilog, así como su símbolo esquemático y tabla de verdad.

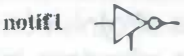
in	out
0	1
1	0
x	x
z	x

\in2 in1\	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

\in2 in1\	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

\in2 in1\	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

\in2 in\	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x



\ctrl in\	0	1	x	z
0	z	1	?	?
1	z	0	?	?
x	z	x	x	x
z	z	x	x	x

COMPUERTAS BASICAS

Fig. 12. Primitivas básicas de las compuertas en Verilog.

3.2 Módulos estructurales utilizando compuertas básicas.

Se muestra como ejemplo la construcción de un módulo que representa el multiplexor 4 a 1 ilustrado en la Fig. 13. El código Verilog que lo representa es:

```
module mux4(slct1, slct0, in3, in2, in1, in0, out);
  input slct1, slct0;
  input in3, in2, in1, in0;
  output out;
    not(nslctc1, slct1); not(nslct0, slct0);
    and(a3, in3, slct1, slct0);
    and(a2, in2, slct1, nslct0);
    and(a1, in1, nslct1, slct0);
    and(a0, in0, nslct1, nslct0);
    or(out, a3, a2, a1, a0);
endmodule
```

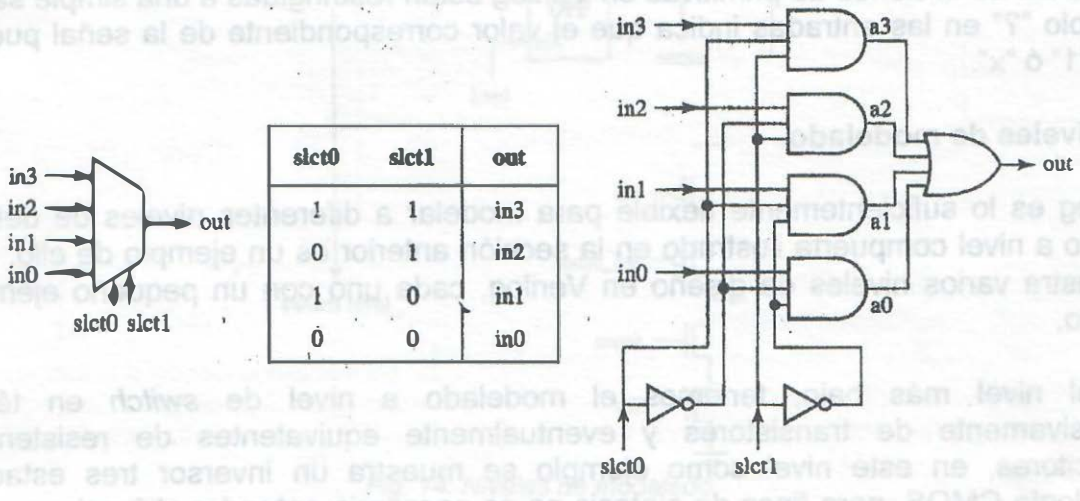


Fig. 13. Ejemplo de un multiplexor 4 a 1

Observe que en la utilización de las compuertas, Verilog espera primero el nombre de la salida y después el nombre de todas las entradas separadas por comas. Los *wires "nslct1, nslct0, a3, a2, a1 y a0"* representan interconexiones internas en el módulo. Las declaraciones para ellas son obligatorias únicamente si son vectores, en nuestro caso son escalares por tanto no se requieren, aunque su declaración hace el código más legible.

3.3 Primitivas definidas por el usuario.

Además de las primitivas ya definidas por Verilog, también se permiten primitivas definidas por el usuario, generalmente abreviadas como UDPs. En estas el usuario puede especificar ya sea una función combinacional o una función secuencial por

medio de una tabla. La forma de definir una primitiva para una función combinacional se ilustra en seguida y es para el caso de un multiplexor de dos entradas:

```
primitive prim_mux2(out, in1, in0, slct);
  output out;
  Input in1, in0, slct;
  table
    //in1  in0  slct  :  out
      0    0    ?    :  0;
      1    1    ?    :  1;
      0    ?    1    :  0;
      1    ?    1    :  1;
      ?    0    0    :  0;
      ?    1    0    :  1;
  endtable
```

Todas las definiciones de primitivas en Verilog están restringidas a una simple salida. El símbolo "?" en las entradas indica que el valor correspondiente de la señal puede ser "0" , "1" ó "x".

3.4 Niveles de modelado.

Verilog es lo suficientemente flexible para modelar a diferentes niveles de detalle, el diseño a nivel compuerta ilustrado en la sección anterior es un ejemplo de ello. La Fig. 14 ilustra varios niveles de diseño en Verilog, cada uno con un pequeño ejemplo de diseño.

En el nivel más bajo, tenemos el modelado a nivel de *switch* en términos exclusivamente de transistores y eventualmente equivalentes de resistencias y capacitores, en este nivel como ejemplo se muestra un inversor tres estados en tecnología CMOS, para fines de síntesis no es necesario entender el funcionamiento a este nivel. El siguiente nivel de la jerarquía es el *nivel compuerta* (y flip-flops), como ejemplo se muestra un flip-flop conectado a un circuito inversor tres estados. El siguiente nivel se le conoce como *RTL* (nivel de transferencia entre registros), formalmente hablando este nivel consiste en una secuencia de transferencia entre registros y otros bloques de tamaño similar, representado como una serie de asignaciones entre registros. El lado derecho de estas asignaciones puede contener expresiones del tipo descrito en este manual; en el ejemplo se muestra un registro conectado a un bus, al cual se puede transferir la información en cualquier sentido. Finalmente la cima de la jerarquía está constituida por el conocido como *nivel arquitectura* el cual utiliza bloques grandes para modelar cosas tales como una memoria, pequeños subsistemas etc. Esta forma generalmente es especificada por módulos compuestos por sentencias en el nivel RTL.

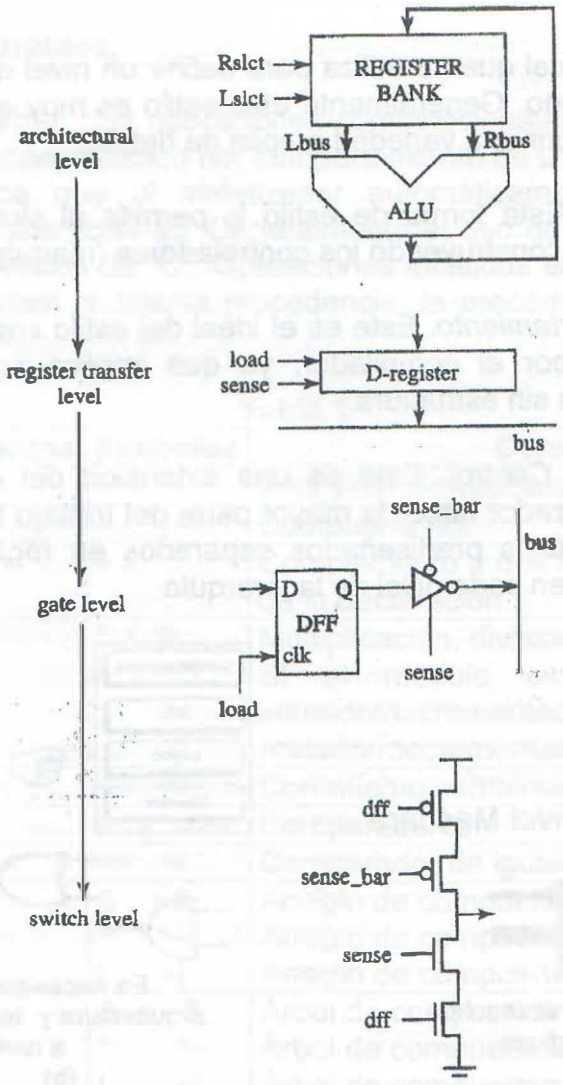


Fig. 14. Niveles de modelado

3.5 Estilos de escritura.

En adición a la pregunta de los niveles de modelado se encuentra la otra dimensión que está entre la especificación estructural o por comportamiento. La especificación a nivel de compuerta dada para el multiplexor en las secciones anteriores es un ejemplo de estilo puramente estructural. La misma funcionalidad puede ser expresada en el estilo puramente de comportamiento como una sentencia condicional:

assign out = select ? in1 : in0;

Para un módulo de cualquier complejidad usualmente se acepta una mezcla de estos dos estilos extremos. Verilog permite fácilmente una mezcla arbitraria de estos dos extremos. Aquí hay algunos ejemplos de estilos de escritura, representados gráficamente en la Fig. 15.

- *Estilo RTL.* Al igual que se utiliza para definir un nivel de diseño, también define un estilo de diseño. Generalmente este estilo es muy estructural y explícito, por lo que puede cubrir una variedad amplia de detalle.

- *Estilo Implícito.* Esta forma de estilo le permite al sintetizador hacer la mayor parte del trabajo construyendo los controladores (maquinas de estado).

- *Estilo de comportamiento.* Este es el ideal del estilo implícito, y depende de que sea soportado por el compilador, ya que implica solamente descripción de comportamientos sin estructura.

- *Estilo Punto de Control.* Este es una extensión del estilo implícito. También permite al sintetizador hacer la mayor parte del trabajo al construir controladores y utiliza submódulos prediseñados separados en regiones estructurales y de comportamiento en cada nivel de la jerarquía.

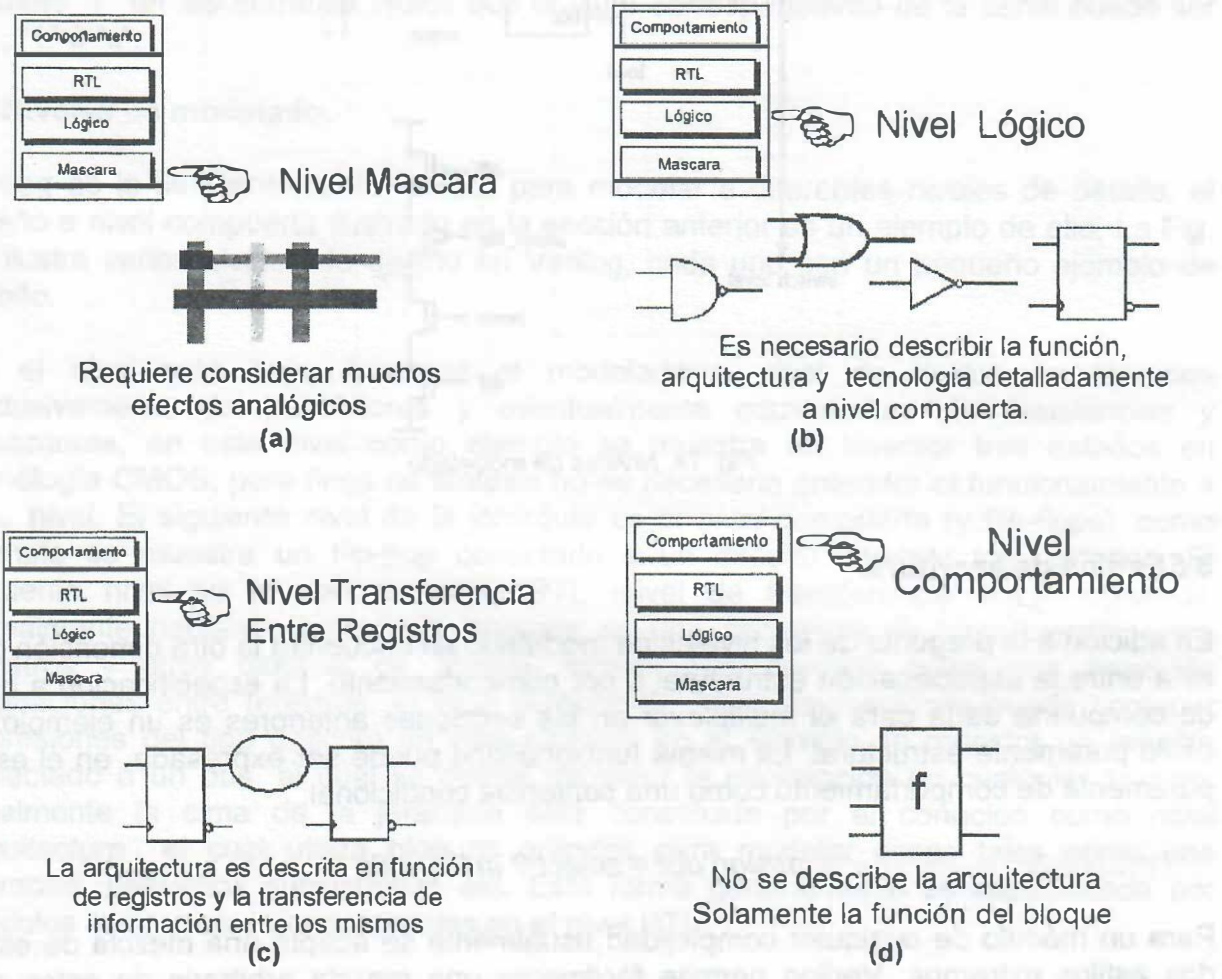


Fig. 15. Estilos de escritura y niveles de modelado .

3.6 Operaciones sintetizables.

En la Tabla 1, se muestran las operaciones combinacionales de Verilog que pueden aparecer en cualquier especificación por comportamiento de un diseño y que es posible sintetizar. Esto significa que el sintetizador automáticamente elegirá un módulo funcional adecuado de sus librerías. Generalmente el uso de los símbolos para estos operadores siguen la sintaxis de "C". Operaciones incluidas en el mismo renglón de la tabla se asume que tienen la misma precedencia, la precedencia disminuye hacia el final de la tabla.

Tabla 1

Tipo	*Argumentos*	*Símbolos*	*Comentarios*
Nivel bit	Unario	~	Inversión complemento a 1
Lógicos	Unario	!	Compuerta not
Aritmético	Unario	+ -	Complemento a dos con respecto a la longitud de la declaración.
Aritmético	Binario	* / % + -	Multiplicación, división, modulo. Si el módulo está disponible, también sumador/incrementador y restador/decrementador.
Corrimiento	Binario	<< >>	Corrimiento combinacional (barrel).
Relacional	Binario	<<= >>=	Comparadores.
Igualdad	Binario	== !=	Comparador de igualdad.
Nivel bit	Binario	& ~& ^ ^~ | ~|	Arreglo de compuertas *and* separadas. Arreglo de compuertas *xor* separadas. Arreglo de compuertas *or* separadas.
Reducción	Unario	& ~& ^ ^~ | ~|	Árbol de compuertas realizando función *And*. Árbol de compuertas realizando función *Xor*. Árbol de compuertas realizando función *Or*.
Lógicos	Binario	&& ||	Compuerta *And*. Compuerta *Or*.
Condicionales	Ternario	? :	Multiplexor de dos entradas.

Tabla 1. Operadores Sintetizables

3.7 Asignaciones Continuas.

El más simple constructor de comportamiento que utiliza los operadores de la Tabla 1 es la sentencia de asignación continua. Se llama continua porque el simulador y el hardware generado continuamente examina el lado derecho de la asignación para detectar algún cambio y reevaluar la expresión para actualizar el lado izquierdo de la asignación. La sintaxis es:

assign net = expresión;

Por ejemplo:

assign out = dff && mask;

Una forma simplificada de hacerlo utilizando la declaración es:

wire out = dff && mask;

En ambos casos la sintaxis modela una situación de hardware en la cual un circuito AND maneja continuamente la NET. El siguiente es otro ejemplo con una variante que usa concatenación en el lado izquierdo:

assign {C_out, SUM} = A+B+C_in;

Sin *C_out*, asumiendo que *SUM, A y B* tienen las mismas dimensiones, *SUM* podría ser truncada.

La utilización de asignamientos continuos aún cuando se utilizan constructores como *if...elseif...case* (descritos más adelante) garantiza que estos módulos serán sintetizados con lógica puramente combinacional más que con circuitos secuenciales. A continuación se listan dos códigos que definen multiplexores 4 a 1 pero en estilo de comportamiento y no estructural. Ambos casos están escritos utilizando el estilo de comportamiento, difieren en que el segundo considera que está conectado a un bus utilizado por otros dispositivos y por tanto tiene la posibilidad de tres estados.

```
module mux4(select, Din0, Din1, Din2, Din3, Dout);
  parameter n=16;
  input [1:0] select;
  input [n-1:0] Din0, Din1, Din2, Din3;
  output [n-1:0] Dout;
  assign Dout = select[1] ?
          (select[0] ? Din3 : Din2) :
          (select[0] ? Din1 : Din0) ;
endmodule


module mux4(select, Din0, Din1, Din2, Din3, Dout);
  parameter n=16;
  input [1:0] select;
  input [n-1:0] Din0, Din1, Din2, Din3;
  output [n-1:0] Dout;
  assign Dout = (select==3) ? Din3 : {n{1'bz}},
         Dout = (select==2) ? Din2 : {n{1'bz}},
         Dout = (select==1) ? Din1 : {n{1'bz}},
         Dout = (select==0) ? Din0 : {n{1'bz}};
endmodule
```

3.8 Ejemplo de diseño. Unidad Lógica y Aritmética (ALU).

Se presenta el diseño de una ALU que puede ser utilizada por ejemplo en un micro procesador. Esta ALU es definida en estilo de flujo de datos que es parte del diseño por comportamiento. Esta ALU tiene un conjunto de instrucciones signadas y otro de no signadas. La condición de overflow en operaciones no signadas es igual que la del carry; sin embargo la condición de overflow en operaciones signadas es más compleja y se da cuando el bit de acarreo entra al bit de signo y nunca sale o cuando nunca entra pero si sale. A continuación se muestra el código Verilog para esta ALU.

```
// Conjunto de operaciones del ALU, elegidas por función

'define ADD  4'b0111    //suma signada con complemento a dos
'define ADDU 4'b0001    //suma no signada
'define SUB 4'b0010     //resta signada con complemento a dos
'define SUBU 4'b0100    //resta no signada
'define AND 4'b0100     //And lógica a nivel de bits
'define OR 4'b0101      //Or lógica a nivel de bits
'define XOR 4'b0110     //Xor lógica a nivel de bits
'define SLT 4'b1010     //envía resultado = 1 si es menor que el 2's complemento
'define SLTU 4'b1011    //envía resultado = 1 si es menor que no signado
'define NOP 4'b0000     //No hace nada

module alu(func, operand0, operand1, result, ovf1);
   parameter n=16;
   input [3:0] func;
   input [n-1:0] operand0, operand1;
   output [n-1:0] result;
   output ovf1;
   wire carry_out, ovf1;
   wire [n-1:0] operand0, operand1, tmp, result;

   assign
   {carry_out, tmp} = ((func=='ADDU) || (func=='ADD)) ? operand1+operand0 : {n{1'bz}},
   {carry_out, tmp} = ((func=='SUBU) || (func=='SUB)) ? operand1-operand0 : {n{1'bz}},
   {carry_out, tmp} = ((func=='SLTU) || (func=='SLT)) ? operand1-operand0 : {n{1'bz}};

   assign  // calculo del overflow
   ovf1 = ((func == 'ADDU)  || (func == 'SUBU) || (func == 'SLTU)) ? carry_out : 1'bz,
   ovf1 = (func == 'ADD) ? operand0[n-1] & operand1[n-1] & ~tmp[n-1] | ~operand0[n-1]
          & ~operand1[n-1] & tmp[n-1] : 1'bz,
   ovf1 = ((func == 'SUB) || (func =='SLT)) ? operand0[n-1] & ~operand1[n-1] & tmp[n-1]
          | ~operand0[n-1] & operand1 [n-1] & ~tmp[n-1] : 1'bz;
```

```
assign
   result = ((func == 'ADDU)  || (func == 'ADD)) ? tmp : {n{1'bz}},
   result = ((func == 'SUBU)  || (func == 'SUB)) ? tmp : {n{1'bz}},
   result = (func == 'OR)  ?  operand0 | operand1 : {n{1'bz}},
   result = (func == 'AND)  ?  operand0 & operand1 : {n{1'bz}},
   result = (func == 'XOR)  ?  operand0 ^ operand1 : {n{1'bz}},
   result = (func == 'NOP)  ?  operand0 : {n{1'bz}},
   result = (func == 'SLTU)  ?  carry_out : {n{1'bz}},
   result = (func == 'SLT)  ?  tmp[n-1] ^ovf1 : {n{1'bz}};
endmodule
```

La primera parte define las diez operaciones posibles para el ALU. Dentro del módulo la primera parte de las asignaciones continuas coloca el resultado de la operación en la variable temporal *temp* y en el acarreo *carry_out*. La segunda parte calcula el overflow de la operación. La tercera asigna el resultado de la operación a la variable de salida.

4. ESPECIFICACIÓN PROCEDURAL.

El estilo de asignación continua descrito en la sección anterior están limitados en poder expresivo. Los estilos de descripción por comportamiento utilizan constructores procedurales. Este tipo de especificación procedural es típica y natural de los lenguajes de programación de software como "C", de forma tal que este estilo es fácil de seguir. El precio que se paga por ganar poder expresivo, es que las asignaciones se tienen que hacer hacia registros.

4.1 El bloque *always*

La sintaxis del bloque *always* es:

Always @(expresión o expresiones de eventos)
(asignaciones) ó (bloques)

El funcionamiento de este bloque es que la ejecución comienza en el tiempo cero, es decir cuando el sistema se enciende, y todas las sentencias de asignación o bloques se repiten indefinidamente hasta que el sistema se apaga. La expresión de eventos es opcional y puede ser una de las siguientes:

- *Tipo nivel.* Por ejemplo always @(a or b or c), este tipo de bloque se dispara si algún cambio ocurre en los valores de "a" ó "b" ó "c". Esta forma usualmente resulta en la síntesis de redes combinacionales con la declaración de registros para el lado izquierdo de las asignaciones.

- *Tipo flanco.* Por ejemplo always @(posedge a or negedge b), en este tipo de bloque al menos una de las variables será utilizada como reloj y por tanto un sistema síncrono secuencial será sintetizado.

4.1.1 Bloques

Un bloque secuencial es una serie de asignaciones encerradas entre sentencias **begin** y **end.** Las sentencias que aparecen en un bloque se ejecutan en forma secuencial. Hay varias alternativas para construir y modelar concurrencia:

- *Primero.* Es posible tener múltiples bloques *always* todos ejecutándose en tiempo cero. Es decir los bloques *always* son paralelos y se ejecutan por tanto al mismo tiempo. La desventaja de este procedimiento es que máquinas secuenciales separadas son generadas para cada bloque *always.*
- *Segundo.* El efecto de concurrencia a nivel de sentencias puede ser obtenido con un bloque secuencial utilizando asignamientos no bloqueados como se discurrirá más adelante.
- *Finalmente.* Invocaciones estructurales fuera de los bloques *always* se activan concurrentemente.

4.2 Funciones y Tareas

Las funciones en Verilog trabajan en forma muy similar a las funciones en lenguaje "C", y son introducidas por la misma razón, para hacer el software más entendible por modularización. En Verilog una función es declarada de la siguiente forma:

```
function  [range]
func_name;
  parameters
  input declarations
  reg declarations
  ... text body ...
endfunction
```

El orden y rango de la declaración debe casar con los argumentos en la invocación. Como en "C" la función retorna un *bitvector* de anchura correspondiente a *[range]* que por default es de un bit. Si se desea retornar más de un argumento se puede hacer construyéndolo internamente en la función utilizando el operador de concatenación.

Las funciones tienen las limitaciones de que se ejecutan en tiempo cero y no pueden contener sentencias de control. Por estas razones los diseñadores de Verilog eligieron para relajar estas limitaciones un nuevo tipo de subrutina llamada **task** (tarea). Las tareas pueden tener la opción de no entradas, múltiples salidas o entradas y constructores de control. Las salidas deben ser registradas y el formato para la invocación es:

```
task_name(argumentos);
```

La sintaxis para el cuerpo de la tarea es:

```
task task_name;
```

```
   parámetros
   input declaración
   output declaración
   reg declaración
   .... cuerpo de texto ....
endtask
```

Por supuesto el orden y rango de los argumentos en la invocación debe casar la declaración de entradas salidas en el cuerpo de la tarea.

Como ejemplo se presenta el diseño de un sumador de 4-bits donde el componente principal llamado full-adder se definirá en termino funciones y no de submódulos.

```
module add_4_r (A, B, C_in, SUM, C_out);
   input [3:0] A, B;
   input C_in;
   output [3:0] sum;
   output C_out;

   function fulladd;
      input a, b, c_in;
      {c_out, sum} = a + b + c_in;
   endfunction

   always @(A or B or C_in)
   begin
      {C0, SUM[0]}=fulladd(A[0], B[0], C_in);
      {C2, SUM[2]}=fulladd(A[2], B[2], C1);
      {C_out, SUM[3]}=fulladd(A[3], B[3], C2);
   end
endmodule
```

En esta especificación primero se declara el cuerpo de la función y después se hacen las cuatro invocaciones necesarias dentro del bloque always. La función tiene solo una salida permitida, de manera que las dos salidas del fulladder deben ser concatenadas. Es importante notar que las cuatro invocaciones a la función ocurren en secuencia. Actualmente la utilización del símbolo "=" indica que los asignamientos son *bloqueados*, es decir cada asignación debe completarse antes de que la siguiente asignación comience.

Como otro ejemplo se presenta el diseño de un sumador de 4-bits donde el componente principal llamado full-adder se definirá en termino de tareas que permiten múltiples salidas como las requerimos para este ejemplo:

```
module add_4_r(A, B, C_in, SUM, C_out);
   input [3:0] A, B;
   input C_in;
```

```
    output [3:0] SUM;
    output C_out;

    task fulladd;
      input a, b, c_in;
      output sum, c_out;
      {c_out, sum} = a + b + c_in;
    endtask

    always @(A or B or C_in)
    begin
      fulladd (A[0], B[0], C_in, SUM[0], C0);
      fulladd (A[1], B[1], C0, SUM[1], C1);
      fulladd (A[2], B[2], C1, SUM[2], C2);
      fulladd (A[3], B[3], C2, SUM[3], C_out);
    end
endmodule
```

Las propiedades de las tareas que permiten múltiples salidas pueden ser utilizadas para este ejemplo del sumador de 4 bits.

4.3Asignamientos bloqueados y no bloqueados

Los asignamientos se indican por el signo "=", que se llama *asignación bloqueada*, tal como:

```
begin
   A = B;
   B = A;
end
```

En este tipo, la asignación se debe completar antes de continuar con la siguiente acción aún cuando no exista tiempo de retraso explícitamente indicado. El resultado en este caso será que tanto A como B tendrán al final el valor original de B.

La *asignación no bloqueada* se indica por el símbolo "<=", tal como:

```
begin
   A <= B;
   B <= A;
end
```

En este tipo, todos los lados derechos de las asignaciones son evaluados y colocados en localidades temporales después de esto se ponen los valores en los lados izquierdos de las asignaciones. Esto da el efecto de concurrencia, de modo que para el ejemplo el resultado será el intercambio de los valores en A y B. En la Fig. 16 se muestran los posibles resultados para la síntesis de los correspondientes ejemplos de las

asignaciones bloqueadas y no bloqueadas. Para el caso de las asignaciones no bloqueadas, lado derecho de la Fig. 16, la variable "B" recibe el valor previo de la variable "A", un resultado de la síntesis es el registro de corrimiento mostrado en la figura. Para los asignamientos bloqueados por el otro lado el resultado son dos latches terminando ambos con el mismo valor como se muestra en el lado izquierdo de la Fig. 16. Cabe hacer notar que no es permitido mezclar los dos tipos de asignamientos en el mismo bloque *always* ni tampoco en asignamientos al mismo registro.

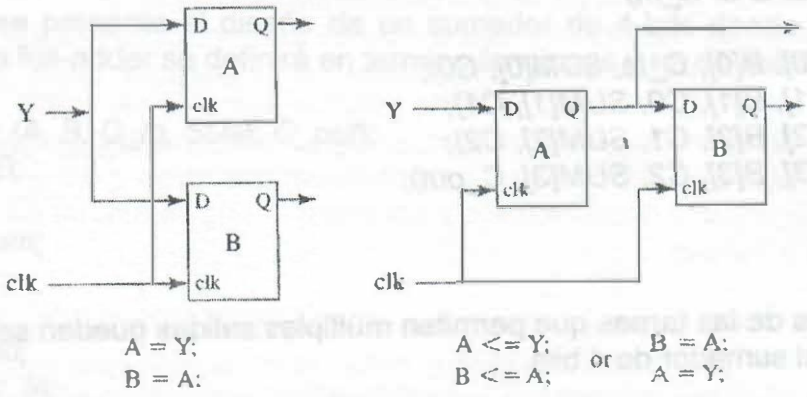


Fig. 16. Asignaciones bloqueadas y no bloqueadas.

4.4Constructores de Control

Los constructores de control son muy parecidos a las estructuras de control utilizadas en los lenguajes de alto nivel como "C" para definir programación estructurada. Verilog dispone de este tipo de constructores que nos permiten tomar decisiones así como modificar el flujo de control.

4.4.1 Sentencias IF

En especificación procedural el operador condicional "?:" es reemplazado en forma más general por el constructor **IF**, el cual puede tener las siguientes formas:

If (expresión) bloque;
If (expresión) bloque_1 else bloque_2;
If (expresión) bloque_1 elseif (expresión) bloque_2 elseif ... else bloque_n;

En la primera forma el bloque se ejecuta solamente si la expresión se evalúa a verdadera. En la segunda forma el bloque_1 se ejecuta si la expresión se evalúa a verdadera sino se ejecuta el bloque_2. En la tercera forma el bloque_1 se ejecuta si la expresión se evalúa a verdadero, sino entonces se prueba otra condición si se cumple se ejecuta el bloque_2, sino se prueba otra expresión y así sucesivamente hasta el bloque_n.

4.4.2 Sentencias de Iteración

Verilog tiene cuatro tipos de sentencias de iteración que pueden ser mezcladas:

- ***repeat <n> <bloque>*** : Repite "n" veces las sentencias del bloque.
- ***for (índice inicial; índice terminal; paso) bloque*** : Usualmente utilizado para iterar sobre un espacio más que sobre tiempo. Sirve por ejemplo para modelar hardware paralelo sobre vectores o arreglos. En este caso un entero debe ser declarado para el índice, no un registro. En los casos que el ciclo for itera sobre el tiempo, las consideraciones de romper con el ciclo son aplicables.
- ***while (expresión) bloque*** : Ejecuta el bloque hasta que la expresión se evalúa a falsa. Puede no ejecutarse si al inicio la expresión es evaluada a falsa.
- ***Forever*** : Ejecuta un bloque siempre. Es muy parecido al always.

Como ejemplo vamos a mostrar el diseño de un comparador en varios estilos para demostrar como se pueden utilizar los ciclos para diseñar un mismo ejemplo. A continuación se presenta la especificación del comparador en estilo de flujo de datos:

```
module comparador(A, B, Cgt, Clt, Cne);
   parameter n = 4;
   input [n-1:0] A, B;
   output Cgt, Clt, Cne;
   assign Cgt = (A>B);
   assign Cgt = (A<B);
   assign Cgt = (A!=B);
endmodule
```

Superior a la forma de especificación anterior se muestra la siguiente especificación que utiliza un estilo procedural como el descrito en esta sección:

```
module comparador(A, B, Cgt, Clt, Cne);
   parameter n = 4;
   input [n-1:0] A, B;
   output Cgt, Clt, Cne;
   reg Cgt, Clt, Cne;
   always @(A or B)
   begin
      Cgt <= (A > B);
      Clt  <= (A < B);
      Cne <=(A != B);
   end
endmodule
```

Es importante notar que en la definición de una función puramente combinacional como esta, las asignaciones no bloqueadas no son realmente necesarias, sin embargo tener el hábito es buena costumbre.

La tercera alternativa para definir el mismo comparador utiliza un ciclo *for* para lograr una comparación por propagación. Cual método utilizar depende de las librerías y el poder de síntesis de nuestra herramienta, pero como regla general es conveniente comenzar por las definiciones más sencillas:

```
module comparador(A, B, Cgt, Clt, Cne);
  parameter n= 4;
  input [n-1:0] A, B;
  output Cgt, Clt, Cne;
  reg Cgt, Clt, Cne;
  integer i;
    always @(A or B)
      begin : compare
        for (i=n-1; i>=0; i=i-1)
          begin
            If(A[i] ^ B[i])
              begin
                Cgt <= A[i]; Clt <= ~A[i]; Cne <=1;
                disable compare; //detiene la comparación
              end
          end
          // Si este punto es alcanzado entonces A==B
          // Si el disable se ejecuta, la siguiente línea no se ejecuta
          Cgt <= 0;  Clt <= 0; Cne <= 0;
      end
endmodule
```

Note que en este caso se comparan iterativamente dentro del ciclo *for* cada uno de los bits de "A" y "B" utilizando una operación XOR, si se encuentra algún bit diferente se establece el estado de las salidas y se termina la ejecución del ciclo *for* mediante el uso de la cláusula *disable* que es idéntica al *break* de "C".

4.4.3 Síntesis de Constructores Condicionales

Los siguientes dos ejemplos de constructores **if** son llamados **completos** debido a que todas las posibilidades son definidas:

```
If (selector)  out = A;          out = B;
Else           out = B;          if (selector)  out = A;
```

La síntesis esperada para una sentencia **if** completa resultará en un multiplexor. La siguiente sentencia: *if (selector) out = A;* resultará en un latch tipo "D" con la variable "A" conectada a la entrada "D" y la variable *selector* conectada a la entrada de nivel del reloj.

4.4.4 Sentencias Case

Para probar más de tres alternativas, aunque es posible utilizar **if...elseif** en forma anidada, en términos de claridad es más conveniente utilizar la sentencia **case.** La semántica es muy parecida a la utilizada en lenguajes de software como "C" y es como sigue:

```
case (expresión)  // valor de n bits
  // item del caso : acción del caso
  n'd0 : block_0;
  n'd1 : block_1;
  ...
  n'dm : block_m;
  [default : block_d;]
endcase
```

La cláusula *default* es opcional y se ejecuta cuando ninguno de los casos se ajusta a la *expresión.* El bloque correspondiente al primer caso se ejecuta si la *expresión* toma el valor previsto en el primer caso. La sentencia *case* compara su *expresión* literalmente bit por bit con los *items* de los *casos* y considera los valores de "x" y "z" como distintas alternativas. Por tanto existen dos variaciones de la sentencia *case* que son:

- **Casez** que trata el símbolo "z" como "no importa (don't-cares)"
- **Casex** que trata a los símbolos "x" o "z" como "no importa (don't-cares)"

El constructor **case** tiene características similares a las sentencias **if**, de modo que el *case* puede ser *completo* cuando todos los posibles casos son considerados y puede ser *paralelo* si los casos de las alternativas son mutuamente excluyentes. De esta forma existen cuatro posibles caos:

Completo y Paralelo: Resulta en una simple lógica paralela, en este caso un multiplexor, como lo muestra el código siguiente y la Fig. 17.

```
module completo-paralelo (slct, A, B, C, D, out);
  input [1:0] select;
  input A, B, C, D;
  output out;
  reg out
    always @(slct or A or B or C or D)
      case (slct)
        2'b11 : out <= A;
        2'b10 : out <= B;
        2'b01 : out <= C;
        default : out <= D;  //2'b00
      endcase
endmodule
```

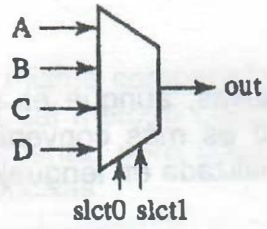


Fig. 17. Caso completo y paralelo.

No Completo y Paralelo: En este caso no se prueban todas las condiciones y produce que se sintetice un latch, como lo muestra el código siguiente y la Fig. 18.

```
module no-completo-paralelo (slct, A, B, C, out);
   input [1:0] select;
   input A, B, C, D;
   output out;
   reg out
   always @(slct or A or B or C)
      case (slct)
         2'b11 : out <= A;
         2'b10 : out <= B;
         2'b01 : out <= C;
      endcase
endmodule
```

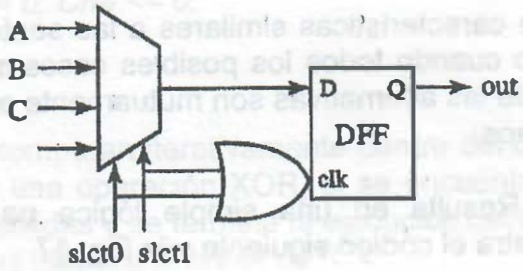


Fig. 18. Caso No completo y paralelo.

Note que aquí no se considera el caso 2'b00, de manera que cuando se presente no debe suceder nada, por ello se sintetiza la compuerta OR conectada a la entrada de nivel del latch.

Completo y no Paralelo: En este caso la especificación de los casos es completa, es decir, se consideran todos los casos, sin embargo no es paralelo, dado que los casos pueden traslaparse entre si, es decir, puede haber diferentes acciones para el mismo caso, como se muestra en el código siguiente y en la Fig. 19.

```
module completo-no-paralelo (slct, A, B, C, out);
   input [1:0] select;
   input A, B, C, D;
   output out;
   reg out
   always @(slct or A or B or C)
      casez (slct)
         2'b1? : out <= A;
         2'b?1 : out <= B;
         default : out <= C;
      endcase
endmodule
```

Note el uso de la sentencia *casez* en lugar de *case*.

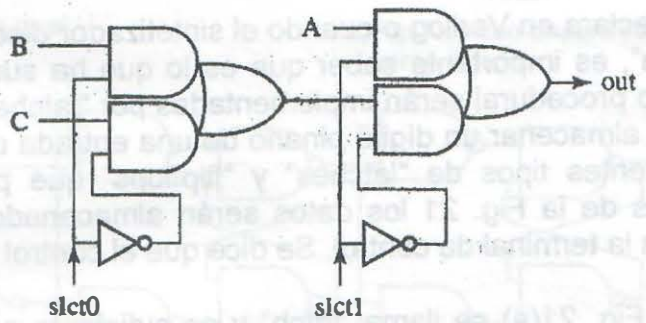


Fig. 19. Caso completo y No paralelo.

No completo y No Paralelo: En este caso la especificación de los casos no es completa porque falta la cláusula *default* y no es paralelo, dado que los casos pueden traslaparse entre si, es decir, puede haber diferentes acciones para el mismo caso, como se muestra en el sig. código y en la Fig. 20.

```
module no-completo-no-paralelo (slct, A, B, out);
   input [1:0] select;
   input A, B;
   output out;
   reg out
   always @(slct or A or B )
      casez (slct)
         2'b1? : out <= A;
         2'b?1 : out <= B;
         // es incompleto si cualquier item de caso falta como el "default"
         // o si alguna de las variables es no asignada en algún caso
         default : ;
      endcase
endmodule
```

Note el uso de la sentencia *casez* en lugar de *case*.

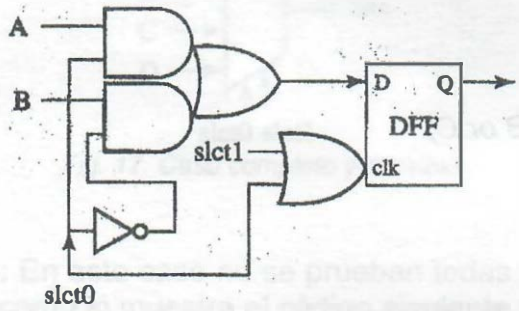


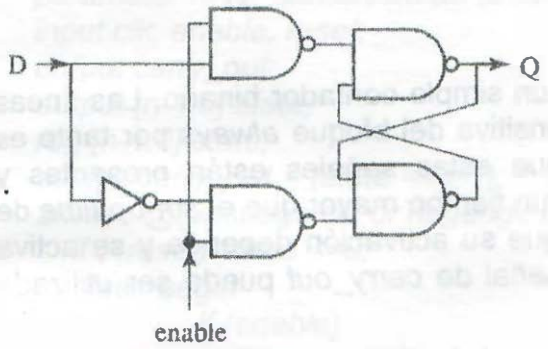
Fig. 20. Caso No completo y No paralelo.

4.5 Flip-Flops contra Latches

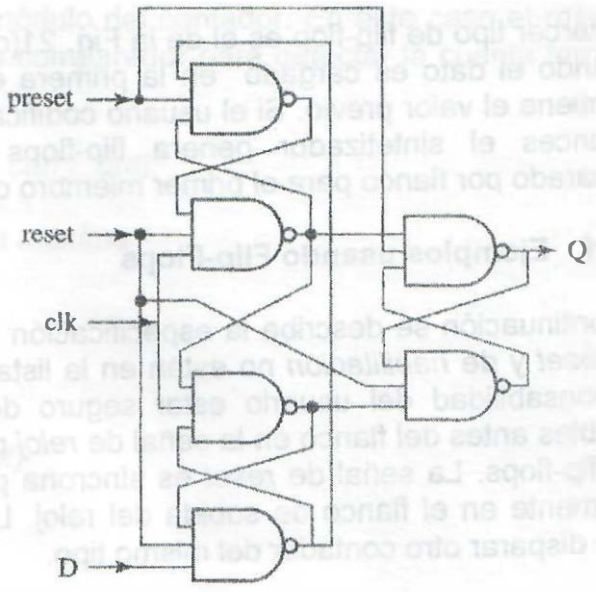
Cuando un **reg** se declara en Verilog o cuando el sintetizador dice que se ha inferido un "registro" o un "latch", es importante saber que es lo que ha sucedido. Sentencias de asignación en código procedural serán implementadas por "latches" o "flipflops" tipo "D" que son capaces de almacenar un dígito binario de una entrada de datos. En la Fig. 21 se revisan los diferentes tipos de "latches" y "flipflops" que pueden resultar de la síntesis. En los tipos de la Fig. 21 los datos serán almacenados en respuesta a un ascenso del pulso en la terminal de control. Se dice que el control es *activo alto*.

El primer tipo en la Fig. 21(a) se llama "latch" y es suficiente para la mayoría de las aplicaciones que requieren retención de datos, pero tienen la desventaja que si el valor de la entrada de datos cambia mientras el habilitador de la entrada de control se mantiene en alto, el cambio se almacena y se refleja en el valor almacenado. Este efecto puede causar problemas si tal elemento es utilizado en el lazo de retroalimentación de una máquina secuencial esto debido a que existirían problemas de alcance de resultado final debido a que la actualización seria continua. Para prevenir este problema se utiliza el dispositivo de la Fig. 21 (b) que se le denomina flip-flop disparado por flanco, este dispositivo ciertamente toma una instantánea de la entrada de datos en el instante que la señal de control ocurre el flanco de subida (cambio de cero a uno). Este tiempo derivado del flanco de subida es caracterizado por las siguientes propiedades de tiempos:

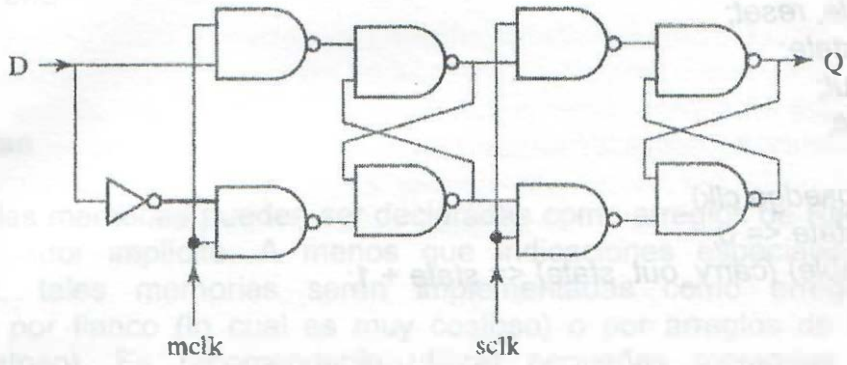
- **Set-up time:** Es el tiempo que la entrada "D" debe permanecer estable antes del flanco del reloj.
- **Hold time:** Es el tiempo que la entrada "D" debe permanecer en estable después del flanco del reloj.



a) Latch, funciona por nivel



b)Flip-Flop con disparo por frente de onda.



c) Flip - Flop maestro esclavo

Fig. 21. Latch contra Flip-flop.

El flip-flop disparado por flanco puede ser construido con preset y/o reset asíncrono. Estas entradas, activas bajas, causan que el flip-flop sea fijado o reseteado respectivamente en cualquier tiempo, sobrescribiendo los datos y líneas de control si estas son activadas simultáneamente. Si un reset asíncrono es requerido las especificaciones deben ser escritas exactamente como:

```
module ff_ar(d, clk, reset, Q);
   input d, clk, reset;
   output Q;
   reg Q;
     always @(posedge clk or negedge reset)
       if (!reset)  Q<=0;
        else        Q<=d;
endmodule
```

Un tercer tipo de flip-flop es el de la Fig. 21(c) llamado *master-slave* (maestro-esclavo). Cuando el dato es cargado en la primera etapa, llamada *maestro* la etapa *esclava* mantiene el valor previo. Si el usuario codifica explícitamente registros *maestro-esclavo* entonces el sintetizador genera flip-flops *maestro-esclavo* utilizando un flip-flop disparado por flanco para el primer miembro del par y para el otro miembro un "latch".

4.5.1 Ejemplos usando Flip-Flops

A continuación se describe la especificación de un simple contador binario. Las líneas de *reset* y de *habilitación* no están en la lista sensitiva del bloque *always* por tanto es responsabilidad del usuario estar seguro de que estas señales están presentes y estables antes del flanco en la señal de *reloj* por un tiempo mayor que el *set-up time* de los flip-flops. La señal de *reset* es síncrona porque su activación depende y se activa solamente en el flanco de subida del reloj. La señal de *carry_out* puede ser utilizada para disparar otro contador del mismo tipo.

```
module counter(clk, enable, reset, state, carry_out);
  parameter n=4;
  input clk, enable, reset;
  output [n-1:0] state;
  output carry_out;
  reg [n-1:0] state;
  reg carry_out;
    always @(posedge clk)
      if (reset) state <= 0;
      elseif (enable) {carry_out, state} <= state + 1;
endmodule
```

Este contador que acabamos de describir es sintetizado utilizando un incrementador y un circuito registro con flip-flops disparados por flanco como se ilustra en la Fig. 22.

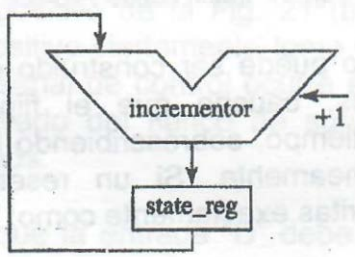


Fig. 22. Incrementador.

El siguiente código es una especificación de un contador un poco más complejo. Este contador tiene una parámetro de cuenta terminal en lugar de esperar a que el contador de la vuelta completa y reinicie desde cero después de alcanzar la cuenta máxima. Este parámetro es estático porque es fijo en la especificación del diseño y no puede variar durante la operación del contador como lo haría un parámetro dinámico el cual formaría

parte de los argumentos de entrada del módulo del contador. En este caso el *reset* es asíncrono. El sintetizador va a insertar un comparador para detectar la cuenta terminal *r-1*.

```
module countertc(clk, reset, enable, state, carry_out);
  parameter n=4;  //numero de bits
  parameter r=12;  //intervalo de la cuenta máxima
  input clk, enable, reset;
  output carry_out;
  output [n-1:0] state;
  reg [n-1:0] state;
  assign carry_out = (state == r-1);
  always @(posedge clk or negedge reset)
    if (~reset)  state <=0;
    else  begin
            if (enable)
              if (carry_out)  state <=0;
              else  state <= state + 1;
          end
endmodule
```

4.6 Memorias

En Verilog las memorias pueden ser declaradas como arreglos de **Registros (reg)** con un decodificador implícito. A menos que indicaciones especiales sean dadas al sintetizador, tales memorias serán implementadas como arreglos de flip-flops disparados por flanco (lo cual es muy costoso) o por arreglos de latches (un poco menos costoso). Es recomendable utilizar pequeñas memorias para propósitos específicos, dado que la implementación de estas en CPLDs o FPGAs es bastante costoso en términos de recursos de espacio. Nosotros veremos dos tipos de memorias RAM, asíncrona y síncrona.

En la RAM asíncrona constantemente se lee la dirección actual hasta que la línea de habilitación de escritura WE es habilitada, en este momento la escritura es inicializada. El siguiente código implementa una RAM asíncrona:

```
module aram(addr, data_in, we, data_out);
  parameter n=7, a=3, w=4; //número de palabras, tamaño de la dirección, tamaño de la palabra
  input we;
  input [a-1:0]  addr;
  input [w-1:0] data_in;
  output [w-1:0] data_out;
  reg [w-1:0] data_out;
  reg [w-1:0] ram_data [n-1:0]; //memoria RAM
  always @(addr  or  data_in)
    if (we)  ram_data[addr] <= data_in;
```

```
    else data_out <= ram_data[addr];
endmodule
```

En contraste en una RAM síncrona existe definido un ciclo de control proveniente de una señal de reloj externa. Las señales de habilitación de escritura WE, la selección de memoria (slct), la dirección de entrada (addr) y la entrada de datos (data_in) deben mantenerse estables para el *set-up time* antes del la llegada del flanco en la señal de reloj. En esta forma de RAM la palabra leída de la memoria debe ser guardada utilizando la señal de control de reloj en un registro buffer a la salida de la RAM. El código de esta RAM se especifica a continuación:

```
module sram(addr, clk, cslct, data_in, we, data_out);
   parameter n=7, a=3, w=4; //número de palabras, tamaño de la dirección, tamaño de la palabra
   input clk, cslct, we; // reloj, selector del chip, habilitador de escritura
   input [a-1:0]  addr;
   input [w-1:0] data_in;
   output [w-1:0] data_out;
   reg [w-1:0] data_out;
   reg [w-1:0] ram_data [n-1:0]; //memoria RAM

   always @(posedge clk)
      if (cslct)   if (we) ram_data[addr] <= data_in;
                   else   data_out <= ram_data[addr];
endmodule
```

Otro tipo de memoria RAM que es útil es la llamada RAM *de doble puerto*. Esta memoria tiene dos puertos de selección para indicar la dirección a acceder. Uno de los puertos es generalmente restringido a propósitos de solo lectura (por supuesto dos contenidos diferentes no pueden ser escritos en la misma localidad jamás). El otro puerto puede permitir tanto lectura como escritura. En nuestro ejemplo deseamos implementar esta RAM con latches en lugar de filp-flops lo que nos obliga a que en la lista sensitiva deba haber solo señales de nivel y no de flanco. El código que define esta RAM asíncrona de doble puertos:

```
module aramdp(D_in, raddr, waddr, we, D1_out, D2_out);
   parameter n=7, a=3, w=4; //número de palabras, tamaño de la dirección, tamaño de la palabra
   input we; // habilitador de escritura
   input [a-1:0] raddr,  waddr;
   input [w-1:0] D_in;
   output [w-1:0] D1_out, D2_out;
   reg [w-1:0] D1_out, D2_out;
   reg [w-1:0]  tpr_data [n-1:0];

   always @(D_in  or  we  or  waddr  or  raddr)
      if (we)  begin  if (waddr !=0)  tpr_data[waddr] <=  D_in; end
      else
```

```
    begin
      if (raddr == 0)  D1_out <= 0;  else  D1_out <= tpr_data[raddr];
      if (waddr == 0)  D2_out <= 0;  else  D2_out <= tpr_data[waddr];
    end
endmodule
```

5. BLOQUES DE FUNCIONES BÁSICAS.

En los días de la lógica TTL, cuando los microprocesadores de 4 bits eran considerados el estado del arte, la familia 74xx fue muy popular. Hoy en día estos bloques son funciones básicas que se siguen utilizando pero como parte de las librerías en las nuevas herramientas de diseño con Dispositivos Lógicos Programables. En esta sección mostraremos los códigos en Verilog para estas funciones básicas que podemos utilizarlas como bloques funcionales en nuestros diseños más complejos.

5.1 Flip-Flop J-K.

El flip-flop JK tiene dos entradas de control llamada "J" y "K". El popular 7473 funciona con reloj controlado por flanco de bajada, el diseño presentado aquí trabaja con flanco de subida por la razón de que este manual está enfocado para trabajar con las herramientas de la empresa ALTERA. Dependiendo de los valores de las entradas "J" y "K" el flip-flop puede tener los estados *set, reset, toggle o de nuevo dato*. El dispositivo es mostrado en la Fig. 23, y su tabla de verdad es la siguiente:

ENTRADAS				**SALIDAS**	
CLRn	**CLK**	**J**	**K**	**Q**	**Qn**
0	X	X	X	0	
1	↑	0	0	No cambia	
1	↑	1	0	1	0
1	↑	0	1	0	1
1	↑	1	1	Complementa (toggle)	

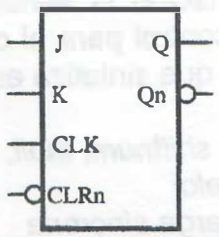


Fig. 23. Flip-flop JK.

El código Verilog que sintetiza este bloque básico conocido como flip-flop JK es el siguiente:

```
module jk_flipflop(clk, clr_n, j, k, q, q_n);
  input clk, clr_n, j, k;  //reloj, limpieza, entradas de control
  output  q, q_n;  // salidas del flipflop
  wire clk, clr_n, j, k;  //entradas tipo alambre
  reg q;  // salida registrada
  wire q_n;

  parameter [1 :0]  HOLD=0, RESET=1, SET=2, TOGGLE=3 ; //funciones del flipflop

  assign q_n=~q;  //calculo de la salida negada
  always @(posedge clk or negedge clr_n)
    begin
      if (~clr_n) begin  q<= 1'b0; end  // limpia el estado del flipflop
      else
        begin
          case ({j,k})
            RESET   :  q <= 1'b0;
            SET     :  q <= 1'b1;
            TOGGLE :  q <= ~q;
          endcase
        end
    end
endmodule
```

Note que la señal de *clear* es asíncrona ya que no requiere de la señal de control de reloj para limpiar el estado del flip-flop.

5.2 Registro de Corrimiento.

Un registro de corrimiento es otro bloque básico común utilizado para muchos diseños. En esta sección se verá un registro de corrimiento de 8 bits conocido como *barrel shifter*, el cual puede recorrer cualquier numero de bits a la izquierda o derecha. Esta implementación utiliza un solo bloque *always* para probar cada flanco de reloj. La entrada de carga es examinada al igual que las demás entradas de control durante este bloque *always*. En esta implementación la señal de *load* (carga) tiene la más alta prioridad, seguida por la señal de control para el corrimiento a la derecha y finalmente corrimiento a la izquierda. El código que sintetiza este registro en Verilog es:

```
module shifter(clk, load, rshift, lshift, shiftnum, inbit, in, out);
  input       clk;       //señal de reloj
  input       load;      //señal de carga síncrona
  input       rshift;    //señal de control de corrimiento a la derecha en forma síncrona
  input       lshift;    //señal de control de corrimiento a la izquierda en forma síncrona
  input [2:0] shiftnum;  //numero de bits a recorrer
  input       inbit;     //bit a introducir en el lugar vacío
  input [7:0] in;        //palabra de entrada para cargar el registro usando la señal de control load
```

```
output [7:0] out;       //palabra de salida del registro

wire        clk, load, rshift, lshift, inbit;
wire  [2:0]  shiftnum;
wire [7:0]   in;
reg  [7:0]   out;

always @(posedge clk)
  begin
    if (load)  begin  out <= in;  end  //load tiene la más alta prioridad
    else  if(rshift)
          begin
            case (shiftnum)
              3'h0:  out <= out;
              3'h1:  out <= {inbit, out[7:1]};
              3'h2:  out <= {inbit, inbit, out[7:2]};
              3'h3:  out <= {inbit, inbit, inbit, out[7:3]};
              3'h4:  out <= {inbit, inbit, inbit, inbit, out[7:4]};
              3'h5:  out <= {inbit, inbit, inbit, inbit, inbit, out[7:5]};
              3'h6:  out <= {inbit, inbit, inbit, inbit, inbit, inbit, out[7:6]};
              3'h7:  out <= {inbit, inbit, inbit, inbit, inbit, inbit, out[7]};
            endcase
          end
    else  if(lshift)
          begin
            case (shiftnum)
              3'h0:  out <= out;
              3'h1:  out <= { out[6:0], inbit};
              3'h2:  out <= { out[5:0], inbit, inbit};
              3'h3:  out <= { out[4:0], inbit, inbit, inbit};
              3'h4:  out <= { out[3:0], inbit, inbit, inbit, inbit};
              3'h5:  out <= { out[2:0], inbit, inbit, inbit, inbit, inbit};
              3'h6:  out <= { out[1:0], inbit, inbit, inbit, inbit, inbit, inbit};
              3'h7:  out <= { out[0], inbit, inbit, inbit, inbit, inbit, inbit};
            endcase
          end
  end
endmodule
```


5.3 Contador.

El contador es otra función muy utilizada en el diseño de sistemas digitales. Aquí presentamos un contador de 8 bits de tipo genérico. Este contador tiene dos entradas asíncronas (reset y preset). Tiene tres entradas síncronas (control up/down, control enable, control load). También existe un acarreo de salida. La representación esquemática del contador se ilustra en la Fig. 24. Cuando las señales de control son

asíncronas y definidas en bloques *always* diferentes la prioridad de las mismas queda indeterminada. A continuación mostramos el código de nuestro contador:

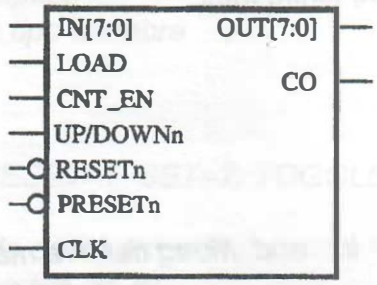


Fig. 24. Contador genérico.

```
module counter (clk, in, reset_n, preset_n, load, up_down, count_en, out, carry_out);
   input clk;                          //señal de reloj
   input [7:0] in;                     //entrada para la carga de datos al contador
   input reset_n;                      //reset asíncrono activo bajo
   input preset_n;                     //preset asíncrono activo bajo
   input load;                         //carga síncrona para el contador
   input up_down;                      //entrada síncrona para cuenta arriba/abajo
   input count_en;                     //habilitador síncrono para cuenta del contador

   output [7:0] out;                   //salida del contador
   output carry_out;                   //acarreo de salida

   wire       clk;
   wire [7:0] in;
   wire       reset_n, preset_n, load, up_down, cout_en;

   reg [7:0]  out;
   reg carry_up, carry_dn, carry_out; // el acarreo de salida se genera a partir de dos registros

   assign carry_out = up_down ? carry_up : carry_dn;

   always @(posedge clk or negedge reset_n or negedge preset_n)
   begin
      carry_up <= 1'b0;
      carry_dn <= 1'b0;

      if (~reset_n) //condición de reset
      begin
         out <= 8'h0;
         carry_dn <= 1'b1;
      end
      else If (~reset_n) //condición de preset, note que el reset tiene prioridad sobre el preset
      begin
         out <= 8'h0;
```

```
      carry_up <= 1'b1;
    end
    else if (load)  //condición de carga. Load tiene prioridad sobre count_enable
    begin
      out <= in;
      if (in == ~8'h0)
        carry_up <= 1'b1;
      else if (in ==  8'h0)
        carry_dn <= 1'b1;
    end
    else if (count_en)  begin
      if (up_down)  begin
        out <= out + 1;
        if (out == ~8'h1)
          carry_up <= 1'b1;
       end
       else  begin
         out <= out – 1;
         if (out == 8'h1)
           carry_dn <= 1'b1
       end
     end
   end
endmodule
```

5.4 Sumador.

El sumador es una función muy utilizada por un ingeniero que hace diseño digital. Cada implantación particular es levemente diferente. En esta sección se presenta el diseño de un sumador síncrono de 32 bits. La representación esquemática de este sumador es mostrada en la Fig. 25. Note que existe una salida llamada *valid*. Esta señal es habilitada cuando la salida del sumador es valida. En la Fig. 26 se muestra el diagrama de tiempo de este sumador.

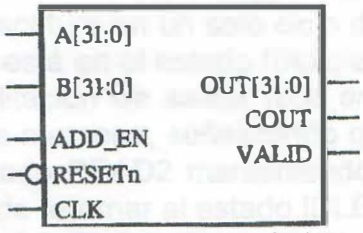


Fig. 25. Sumador de 32 bits.

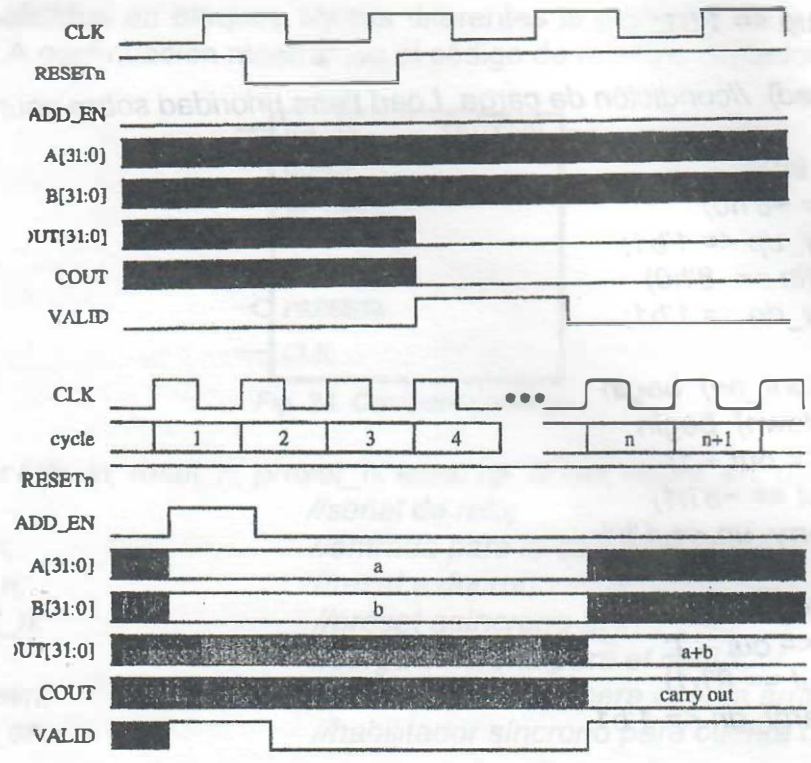


Fig. 26. Diagramas de tiempo del sumador de la Fig. 25.

A continuación se muestra el código para este sumador:

```
module sumador(clk, a, b, reset_n, add_en, out, cout, valid);
  input         clk;      //señal de reloj
  input [31:0]  a;        //entrada operando A
  input [31:0]  b;        //entrada operando B
  input reset_n;          //reset síncrono activo bajo
  input add_en;           //habilitador síncrono del sumador

  output [31:0] out;      //salida
  output cout;            //acarreo de salida
  output valid;           //indica si la salida es válida

  wire          clk;
  wire  [31:0]  a;
  wire  [31:0]  b;
  wire  reset_n;
  wire add_en;
  reg  [31:0]  out;
  reg          cout;
  reg          valid;

  always @(posedge clk) begin
```

```
    if (~reset_n)  begin
        out <= 32'h0;
        cout <= 1'b0;
    end
    else
      if(add_en)
        {cout, out} <= a + b;
    valid <= ~reset_n | add_en | valid;
  end
endmodule
```

6. MÁQUINAS DE ESTADOS.

Las máquinas de estados son parte esencial de cualquier diseño secuencial. De manera que cualquier controlador secuencial es una máquina de estados. En términos de hardware, una máquina de estados consiste de elementos de retraso, lógica combinacional y retroalimentación de las salidas de los elementos de retraso a través de la lógica combinacional, hacia las entradas de los elementos de retraso. Los retrasos de los elementos están controlados por una señal de reloj. Más específicamente los elementos de retraso pueden ser flip-flops tipo "D". Los tipos de máquinas de estados que vamos a estudiar aquí son la máquina Moore y la Mealy.

6.1 La máquina de estados Moore.

Una máquina de estados tipo "Moore" es aquella en la cual las salidas son determinadas únicamente por el estado actual del dispositivo. De esta manera cada estado en la máquina de estados corresponde a un valor específico para cada salida. En la Fig. 27 se muestra una simple máquina de estados para leer y escribir en un dispositivo de memoria. La máquina de estados comienza en el estado IDLE después de una señal de reset. Esta entonces permanece en IDLE hasta que reciba una señal de lectura o escritura (*rd* or *wr*). Si se recibe una señal de escritura, la máquina se mueve hacia el estado WRITE, establece la señal de (*write_en*) y la señal de conocimiento (*ack*) para un ciclo. Después se retorna al estado IDLE, habiendo completado la operación de escritura en un solo ciclo de reloj. Si en vez de esto recibe una señal de lectura mientras está en el estado IDLE, entonces se va al estado READ1 y establece la señal de habilitación de salida (*out_en*). Espera por la señal de listo (*ready*) desde el dispositivo de memoria, señalizando que son correctos los datos en la salida. Entonces se va al estado READ2 manteniendo la salida habilitada y dando la señal de conocimiento, antes de retornar al estado IDLE.

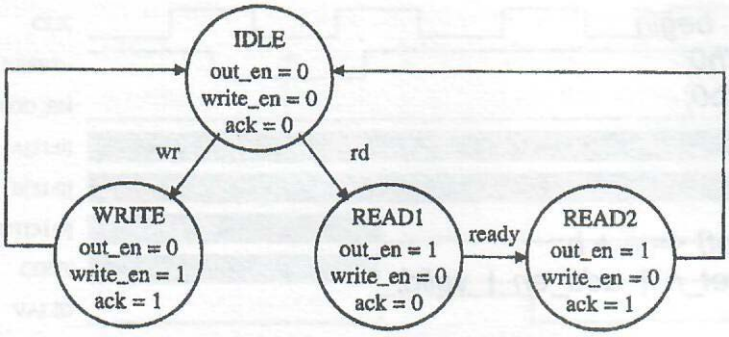
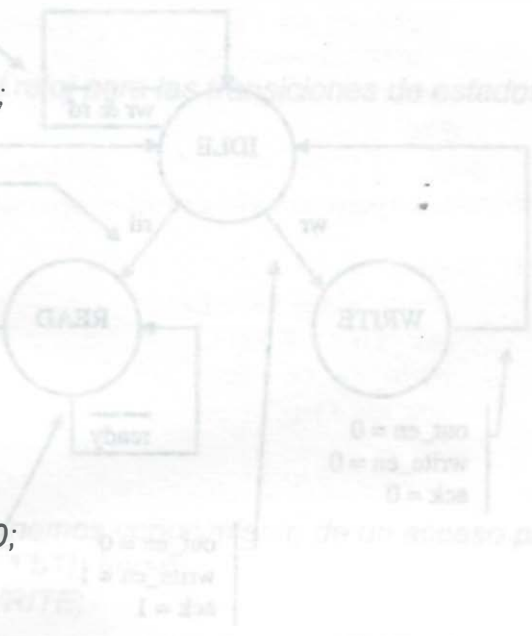


Fig. 27. Diagrama de estados de una máquina Moore.

A continuación se muestra el código que sintetiza esta máquina de estados Moore:

```
module state_machine(clock, reset_n, wr, rd, ready, out_en, write_en, ack);
  input  clock;      //reloj de la maquina de estados
  input  reset_n;    //reset síncrono activo bajo
  input  wr;         //comando de escritura para el procesador
  input  rd;         //comando de lectura para el procesador
  input  ready;      //señal de "listo" proveniente del dispositivo de memoria

  output  out_en;    //salida habilitada para la memoria
  output  write_en;  //Habilita la escritura en memoria
  output  ack;       //Señal de conocimiento del procesador

  wire clock;
  wire reset_n;
  wire wr;
  wire rd;
  wire ready;
  reg out_en;
  reg wite_en;
  reg ack;
  reg [1:0]  mem_state;  //estado de memoria de la máquina de estados

  parameter [1:0]  IDLE = 0, WRITE = 1, READ1 = 2, READ2 = 3;

  //Espera el flanco de subida del reloj para definir las transiciones

  always @(posedge clock)
  begin
    if (~reset_n)
      mem_state = IDLE;
    else
      case (mem_state)
        IDLE:  begin
                 if (wr == 1'b1)
```

```
                mem_state <= WRITE;
              else if (rd == 1'b1)
                mem_state <= READ1;
            end
        WRITE: begin
                 mem_state <= IDLE;
               end
        READ1: begin
                 If (ready == 1'b1)
                   mem_state <= READ2;
               end
        READ2: begin
                 mem_state <= IDLE;
               end
      endcase
  end
//Espera cambios en los estados para determinar el valor de las salidas
  always @(mem_state)
  begin
    case (mem_state)
      WRITE : begin
                write_en = 1'b1;
                ack = 1b'1;
              end
      READ1 : begin
                out_en = 1'b1;
              end
      READ2 : begin
                out_en = 1'b1;
                ack = 1'b1;
              end
       IDLE :  begin
                 out_en = 1'b0;
                 write_en = 1'b0;
                 ack = 1'b0;
               end
    endcase
  end
endmodule
```



6.2La máquina de estados Mealy.

Una máquina de estados Mealy es aquella en la cual las salidas cambian sobre las transiciones del dispositivo. En otras palabras, las salidas para el siguiente estado son determinadas por el estado actual y por las entradas actuales. En una máquina Mealy, las salidas durante un estado particular pueden ser diferentes a diferentes tiempos,

dependiendo de como la máquina entro al estado actual. Esencialmente la especificación de los requerimientos de la máquina de estados determinará cual método (Moore o Mealy) es óptimo para el diseño. En la Fig. 28 se muestra una máquina de estados para el control de lectura y escritura a una memoria. Se trata del mismo problema que se resolvió en el apartado anterior con la máquina Moore, pero en este caso la solución se ha redefinido como una máquina Mealy.

Estando en el estado IDLE y con las entradas *wr* y *rd* no habilitadas, la máquina se mantiene en el estado IDLE y el valor de las salidas *out_en, write_en* y *ack* valen "0". Estando en el estado IDLE pero con la entrada *wr* habilitada se pasa al estado WRITE y el valor de las salidas es consiste en habilitar *write_en* y *ack*. Estando en el estado IDLE pero con la entrada *rd* habilitada se pasa al estado READ y se habilita solo la señal de salida *out_en*. Estando en el estado WRITE sin verificar ninguna entrada en el sig. Ciclo de reloj se pasa al estado IDLE y se deshabilitan todas las salidas. Estando en el estado READ, si la señal de entrada *ready* no es verificada por el dispositivo, se mantiene en el estado READ y la única señal de salida habilitada es *out_en*. Estando en el estado READ, si la señal de entrada *ready* es verificada por el dispositivo, entonces el siguiente estado es IDLE y las variables de salida habilitadas son *out_en* y *ack*.

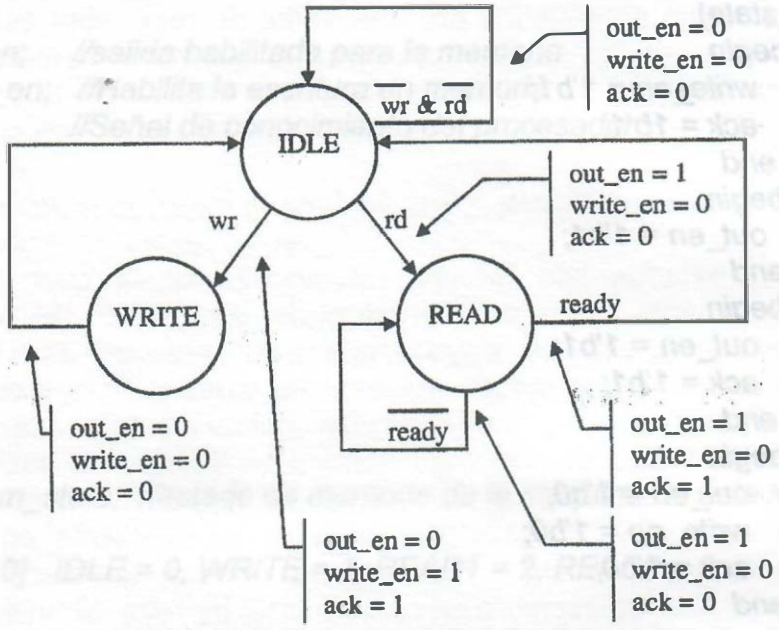


Fig. 28. Diagrama de estados de una máquina Mealy.

El código Verilog de esta máquina Mealy es el siguiente:

```
module state_machine(clock, reset_n, wr, rd, ready, out_en, write_en, ack);
  input  clock;      //reloj de la maquina de estados
  input  reset_n;    //reset síncrono activo bajo
  input  wr;         //comando de escritura al procesador
```

```
input  rd;          //comando de lectura al procesador

output  out_en;    //habilitador de la salida de la memoria
output  write_en;  //habilitador de escritura a la memoria
output  ack;       //señal de conocimiento del procesador

wire  clock;
wire  reset_n;
wire  wr;
wire  rd;
wire  ready;
reg   out_en;
reg   write_en;
reg   ack;

reg   [1:0]   mem_state;  //estados de la máquina de estados

parameter [1:0]   //nombres de los estados
  IDLE  = 0,
  WRITE  = 1,
  READ  = 2;

// Espera el flanco de subida del reloj para las transiciones de estados y salidas
always @(posedge clock)
begin
  if (~reset_n) begin
    mem_state <= IDLE;
    out_en <= 1'b0;
    write_en <= 1'b0;
    ack <= 1'b0;
  end
  else begin
    case (mem_state)
      IDLE:  begin
             //No hace nada si tenemos conocimiento de un acceso previo
             if (~ack  &&  (wr == 1'b1)) begin
               mem_state <= WRITE;
               out_en <= 1'b0;
               write_en <= 1'b1;
               ack <= 1'b1;
             end
             else if (~ack  &&  (rd == 1'b1)) begin
               mem_state <= READ;
               out_en <= 1'b1;
               write_en <= 1'b0;
               ack <= 1'b0;
             end
```

```
                else  begin
                  out_en <= 1'b0;
                  write_en <= 1'b0;
                  ack <= 1'b0;
                end
                 end
            WRITE:  begin
                      mem_state <= IDLE;
                      out_en <= 1'b1;
                      write_en <= 1'b0;
                      ack <= 1'b0;
                     end
            READ:   begin
                      if (ready == 1'b1)) begin
                        mem_state <= IDLE;
                        out_en <= 1'b1;
                        write_en <= 1'b0;
                        ack <= 1'b1;
                      end
                      else  begin
                         out_en <= 1'b1;
                        write_en <= 1'b0;
                        ack <= 1'b0;
                      end
                    end
          //como no todos los estados están definidos, agregar default permite optimizar la
        //síntesis
            default:   begin
                        mem_state <= 2'bxx;
                        out_en <= 1'bx;
                        write_en <= 1'bx;
                        ack <= 1'bx;
                       end
        endcase
    end
  end
endmodule
```

7 BIBLIOGRAFÍA

A) A Verilog HDL Primer. J. Bhasker. Star Galaxy. Second Edition. 1999.
B) Verilog Styles for Synthesis of Digital Systems. David Richard Smith, Paul D. Franzon. Prentice Hall. 2001.
C) Verilog Designer's Library. Bob Zeidman, Robert M. Zeidman. Prentice Hall. 1999.
D) Logic and Computer Design Fundamentals. M. Morris Mano, Charles R. Kime. Prentice Hall. 1999.
E) Modern Digital Systems Design". John Y. Cheung. Ed. West. 1991.