5 – DESCRIPCIÓN BASADA EN ALGORITMOS

Como hemos visto anteriormente, el diseño a nivel funcional o de comportamiento (*behavioral description*) permite describir circuitos digitales atendiendo únicamente a las relaciones existentes entre las entradas y salidas del circuito, pero sin hacer referencia a la estructura real o hardware del que se compone el circuito. Ya se ha estudiado la descripción RTL, y ahora se estudiará la descripción basada en algoritmos.

Como ya dijimos, la descripción basada en algoritmos es un nivel de abstracción mayor. Trabaja sobre variables de tipo *reg*. Puede ser utilizado para describir circuitos combinacionales y circuitos secuenciales. Ejecuta instrucciones de control de flujo comunes a muchos lenguajes de programación de alto nivel. Esta descripción no siempre es sintetizable, aún cuando se pueda simular el comportamiento especificado sin ningún problema.

Llamaremos procedimiento (*procedure*) a construcciones que generalmente comienzan con las sentencia *initial* o con la sentencia *always*, donde se realizan una serie de operaciones con el objetivo de cambiar el valor de variables tipo *reg*. Por ejemplo, como ya vimos anteriormente, se muestra el código de un flip-flop tipo D:

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

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

endmodule
```

Lo que vemos en color verde sería un procedimiento en el cual se asigna un valor a la variable *q*, que es de tipo *reg*.

TIPOS DE DATOS EN VERILOG

En la siguiente tabla aparecen los principales tipos de datos en Verilog:

Nets (Connectivity)		**Registers** (Storage)	
wire	supply0	reg	integer
tri	supply1		

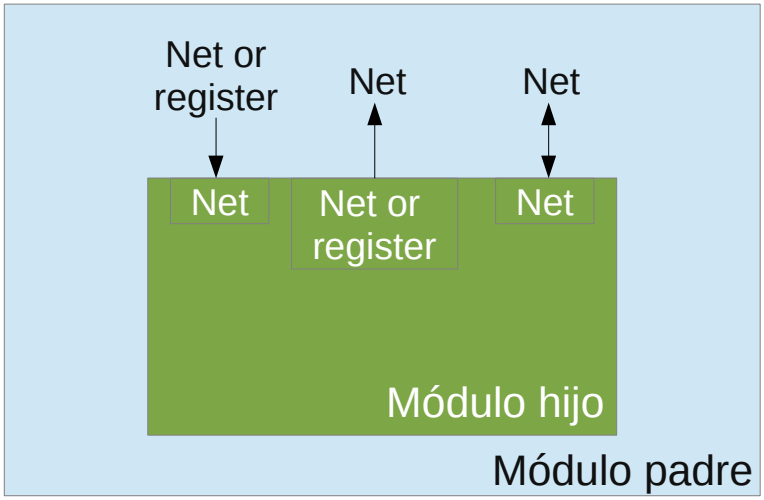
Los datos tipo red (*net*) ya han sido estudiados. Se utilizan simplemente como conexiones, por ejemplo, para realizar la conexión entre la salida de una puerta lógica y la entrada de otra. Se utilizan dentro de la descripción estructural y dentro del nivel de comportamiento RTL. No pueden representar salidas de dispositivos secuenciales, como por ejemplo flip-flops. El valor de una variable de red puede ser asignado utilizando la "asignación continua" (*continuous assignment*), o como la salida de una puerta lógica predefinida, pero no utilizando un procedimiento.

Nunca se puede asignar un valor a una variable tipo *net* dentro de un procedimiento.

Las variables tipo registro, se suelen utilizar para representar salidas de dispositivos secuenciales, como pueden ser las salidas de los flip-flops. Es una abstracción de un elemento hardware de almacenamiento que no se corresponde directamente con ningún elemento de memoria; de hecho, puede representar perfectamente salidas de circuitos combinacionales. Las variables tipo registro, al igual que las tipo red, pueden contener 1 o más bits.
El valor de una variable tipo registro sólo puede ser cambiada dentro de un procedimiento, pero nunca utilizando la asignación continua. Tampoco puede ser declarada como la salida de una puerta lógica predefinida.

Las variables entero (*integer*) son similares a las tipo *reg*, con la diferencia que pueden tener signo. Los enteros negativos se almacenan en complemento a 2. Hay que tener mucho cuidado, porque con las variables enteras se extiende el signo. Si queremos que esa variable se represente en decimal, por ejemplo, no tendrá en cuenta el signo, y se convertiría como una variable sin signo.

Es importante también tener en cuenta que mientras que una variable de red puede actuar como una variable de entrada, salida o bidireccional de un módulo, una variable registro sólo puede actuar como variable de salida de un módulo. En la siguiente figura se puede ver un esquema de esto.



En cuanto al tipo de datos *reg*, es importante saber que también se pueden definir estructuras bidimensionales. Esto es útil a la hora de simular memorias de acceso aleatorio. Por ejemplo, en la siguiente sentencia hemos definido una memoria de 1024 direcciones, y la longitud de la palabra es de 32 bits.

```
reg [31:0] memory_name [1023:0];
```

Si queremos hacer referencia a una dirección completa de la memoria o a un bit concreto de la memoria, se haría por ejemplo del siguiente modo:

```
assign b = memory[6]   // el registro b de 32 bits se haría igual a la dirección 6 de la memoria.
assign c = memory[1000][0] //el bit c se hace igual al bit 0 de la dirección 1000 de la memoria.
```

LA SENTENCIA *INITIAL*

Se puede comenzar un procedimiento mediante la sentencia *initial*. **La sentencia *initial* se ejecuta una sola vez comenzando en tiempo cero.** Es muy útil para inicializar el valor de variables y de memorias. Se pueden utilizar dentro de un módulo el número de procedimientos basados en la sentencia *initial* que se deseen. Todos ellos se ejecutarán en paralelo un sola vez comenzando en tiempo cero.

Por ejemplo, si queremos inicializar el valor que toma el registro *A* en el inicio, se podría hacer utilizando el siguiente procedimiento:

```
initial
A = 32'h00000000;   //suponemos que el registro A es de 32 bits
```

Ya que la variable *A* está cambiando dentro de un procedimiento, debe ser obligatoriamente de tipo registro.

Dentro de un procedimiento comenzado con la sentencia *initial* se puede inicializar el valor de más de una variable. En este caso, llevaría asociada una instrucción compuesta, y se debería comenzar con ***begin*** y acabar con ***end***. Esto es similar a las llaves { } utilizadas para instrucciones compuestas en el lenguaje de programación C. Si por ejemplo queremos inicializar el valor de dos registros en lugar de uno solo se haría con el siguiente procedimiento:

```
initial
begin
A = 32'h00000000;   //suponemos que el registro A es de 32 bits
B = 32'h00000000;   //suponemos que el registro B es de 32 bits
end
```

Hay que tener en cuenta que las instrucciones que se encuentran dentro del procedimiento, entre *begin* y *end*, se ejecutan de forma secuencial, es decir,hasta que no se completa una no se empieza

con la siguiente. A continuación tenemos un ejemplo:

```
initial
begin
a = 5;
b = 6;
c = 3;
a = b + c;  // a toma el valor 9
b = a;    // finalmente b toma el valor 9
end
```

LA SENTENCIA *ALWAYS*

Se puede comenzar un procedimiento mediante la sentencia *always*. **La sentencia *always* se ejecuta de forma repetitiva comenzando en tiempo cero y finalizando cuando termine la simulación.** Se pueden utilizar dentro de un módulo el número de procedimientos basados en la sentencia *always* que se deseen. Todos ellos se ejecutarán en paralelo.

Por ejemplo, se puede generar una señal de reloj para ser utilizada en un testbench utilizando el siguiente procedimiento:

```
always
#10 clk = ~clk;
```

Cada 10 unidades de tiempo, el valor de *clk* se complementaría, por lo que su periodo sería de 20 unidades de tiempo. Comenzaría al inicio de la simulación y finalizaría al terminar la simulación. Ya que la variable *clk* está cambiando dentro de un procedimiento, debe ser obligatoriamente de tipo registro.

Al igual que con la sentencia *initial*, si el procedimiento consta de más de una instrucción se comenzaría con ***begin*** y se terminaría con ***end***. Lo que haya entre estas dos palabras se ejecutaría de forma secuencial.

Opcionalmente se puede escribir una condición de control con la sentencia *always*. La sintaxis sería la siguiente:

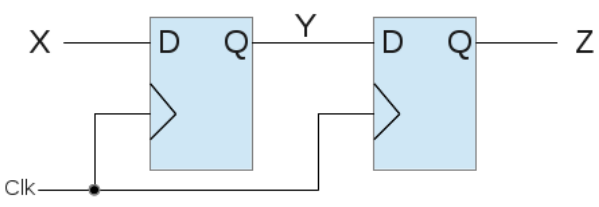
always @ (*condition*)

El símbolo @ significaría "esperar hasta que se cumpla la condición".La condición puede ser alguna de las siguientes:

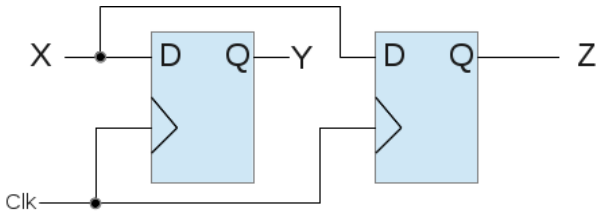
- El cambio de una variable: **@ (a)** Se ejecutaría el procedimiento cuando cambie la variable *a*.
- El flanco positivo de una variable: **@ (posedge clk)** Se ejecutaría el procedimiento en un flanco positivo del reloj.
- El flanco negativo de una variable **@(negedge clk)** Se ejecutaría el procedimiento en el flanco negativo del reloj.
- Función or lógica de algunas de las anteriores condiciones **@ (a or b)** Se ejecutaría el procedimiento cuando cambie la variable *a* o la *b*.

Ejemplos:

```
// flip-flop tipo D activado en flanco positivo


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

always @ (posedge clk)  //siempre que se de un flanco positivo en el reloj

    q=data_in;

endmodule
```




```
// registro de desplazamiento


module  reg (Z, X, clk);
input   X, clk;
output  Z;
reg     Z, Y;

always @ (posedge clk)
        Z=Y;
        Y=X;
 endmodule
```




```
// flip flops en paralelo

module  ff (Z, X, clk);
input   X, clk;
output  Z;
reg     Z, Y;

always @ (posedge clk)
        Y=X;
        Z=Y;
endmodule
```

IF.....ELSE

Las sentencias ***if....else*** ejecutan una operación o bloque de operaciones dependiendo del resultado de la expresión entre paréntesis escrita tras *if*. Si el resultado es verdadero las operaciones en ese bloque se ejecutan. Si el resultado es falso se ejecutarían las operaciones del bloque que comienza por *else* (en caso de existir).

*sintaxis ::**if** (expression)*

begin

.....statements.....

end

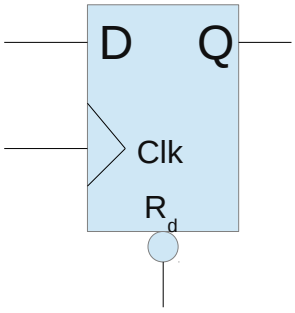
else

begin

.....statements.....

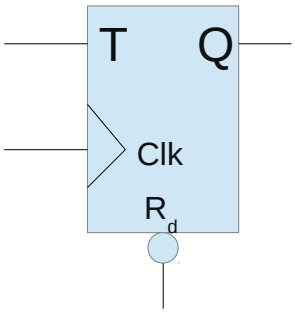
end

Ejemplo:




```
// flip flop tipo D con reset asíncrono

module  ff_D (Q, D, clk, Reset);
input   D, clk, Reset;
output  Q;
reg     Q;

always @ (posedge clk or negedge Reset)
        if (Reset == 0)
          Q=0;
        else
          Q=D;

endmodule
```

Las sentencias ***if....else*** pueden también aparecer anidadas el número de veces que sea necesario, como en el siguiente ejemplo:




```
// flip flop tipo T con reset asíncrono


module  ff_T (Q, T, clk, Reset);
input   T, clk, Reset;
output  Q;
reg     Q;

always @ (posedge clk or negedge Reset)
        if (Reset == 0)
          Q=0;
        else if (T == 1)
          Q=~Q;

endmodule
```

Para escribir la condición que aparece entre paréntesis tras *if*, pueden ser necesarios los operadores de relación (ya vistos en el diseño RTL), y los operadores lógicos que se muestran a continuación.

Operador	nº de operandos	Descripción
!	1	NOT lógico
||	2	OR lógico
&&	2	AND lógico

CASE, CASEX

La sentencia ***case*** permite bifurcar en varias opciones el flujo de ejecución en función de los valores de una expresión escrita entre paréntesis tras la palabra *case*.

El bloque *case* puede opcionalmente terminar con un bloque *default* que se ejecutará si el valor de la expresión no coincide con ninguno de los valores analizados. Es un buen hábito especificar siempre el bloque *default* se necesite o no. Si los *default* son irrelevantes, es interesante definirlos explícitamente como *x*, porque de ese modo el minimizador de lógica los tratará como tales a la hora de sintetizar el circuito.

Una vez ejecutado uno de los casos el programa salta hasta *endcase*. Si se quiere que una sentencia o grupo de sentencias se ejecuten para varios valores de la condición, se colocan todos ellos

separados por comas.

Los valores que puede tomar la expresión pueden ser simples constantes o expresiones o una lista de valores separados por comas.

*sintaxis ::**case** (expression)*

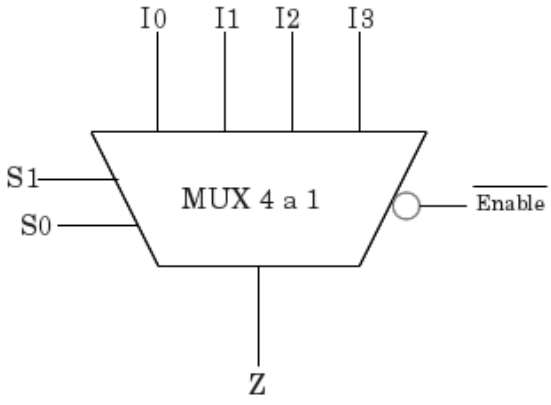
case_choice1:
begin
.....statements.....
end

case_choice2:
begin
.....statements.....
end

.....more case choices blocks.....

default:
begin
.....statements.....
end

Ejemplo: multiplexor de 4 líneas a 1 con entrada de habilitación activa en baja.




```
// MUX 4 a 1 con habilitación en baja
module  mux_4_a_1 (Z, I, S, Enable);
input   Enable;
input [3:0] I;
input [1:0] S;
output Z;
reg      Z;

always @ (I or S or Enable)
       begin
         case ({S, Enable})
            3'b001, 3'b011, 3'b101, 3'b111: Z = 1'b0;
            3'b000: Z = I[0];
            3'b010: Z = I[1];
            3'b100: Z = I[2];
            3'b110: Z = I[3];
            default: Z = 1'bx;
         endcase
       end
endmodule
```

La diferencia entre ***case*** y ***casex*** es la siguiente: en *case* la condición debe cumplirse exactamente. Por lo tanto cuando aparece el valor *x*, la condición debe tomar exactamente el valor *x* para que se ejecute en ese caso. Sin embargo, en caso de *casex* se puede utilizar *x* para indicar cualquier valor. Si por ejemplo queremos escribir de nuevo el código Verilog del multiplexor anterior de 4 líneas a 1, utilizando el comando *casex*, se podría hacer del siguiente modo (únicamente escribo el procedimiento, lo demás es igual):

```
always @ (I or S or Enable)
        begin
          casex ({S, Enable})
             3'bxx1: Z = 1'b0;
             3'b000: Z = I[0];
             3'b010: Z = I[1];
             3'b100: Z = I[2];
             3'b110: Z = I[3];
             default: Z = 1'bx;
          endcase
        end
```

Podemos ver que se ha cambiado el caso para el que el multiplexor está deshabilitado. Como utilizamos *casex*, entonces el caso 3'bxx1 se cumple cuando el bit menos significativo es 1 lógico, independientemente del valor de los dos primeros. Si hubiésemos utilizado *case*, entonces 3'bxx1 se cumpliría cuando el bit menos significativo valiese 1 lógico y los dos más significativos valiesen *x*.

BUCLES: *REPEAT, WHILE, FOR, FOREVER, DISABLE*

Verilog tiene 4 formas distintas de realización de bucles: *repeat*, *while*, *for* y *forever*. *Disable* se utiliza para la salida de bucles.

Repeat.

Sintaxis:: **repeat** (loop count expression) statment

Cuando el flujo del programa llega a *repeat*, se evalúa *loop count expression* para conocer el número de veces que se ha de ejecutar *statement*. Si el resultado de *expression* es *x* ó *z*, el resultado será tratado como 0.

Ejemplo: resetear el contenido de una memoria tras aplicar una señal de reset activa en alta.

```
always @ (posedge reset)
        begin
          word_address = 0;
              repeat (memory_size)
                    begin
                      memory [word_address] = 0;
                      word_address = word_address+1;
                    end
          end
```

Sólo se muestra el procedimiento, no el módulo completo. Se debería haber declarado previamente un dato tipo *integer* que sirve de índice para el bucle (*word_address*).

While.

Sintaxis:: **while** (expression) statment

Bucle similar a repeat, pero ahora no se conoce el número exacto de repeticiones del bucle, por lo que se repetirá *statement* hasta que el valor de *expression* sea falso.

Ejemplo: contar el número de 1 que contiene un registro llamado *reg_a*:

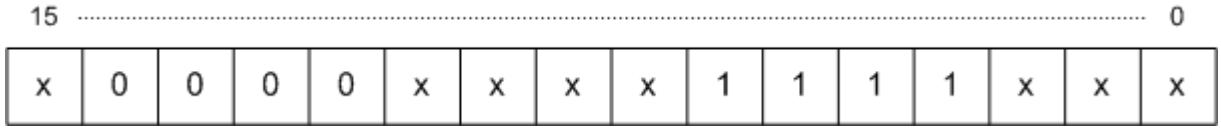
```
initial
       begin
          temp_reg = reg_a;
          count = 0;
          while (temp_reg)
                    begin
                      if (temp_reg[0]) count = count+1;
                       temp_reg = temp_reg<<1;
                    end
      end
```

For.

Sintaxis:: **for** (reg_assignment_1; reg_assignment_2; expression) statment

Se repite *statement* desde un primer valor de un registro (*reg_assignment_1*) hasta un segundo valor del registro (*reg_assignment_2*), y el valor del registro cambia en cada ciclo del bucle del modo que indica *expression*.

Ejemplo: tenemos un registro de 16 bits, y queremos poner a 0 lógico los bits 11, 12, 13 y 14, y a 1 lógico los bits 6, 5, 4 y 3.



Podríamos utilizar el siguiente procedimiento:

```
reg [15:0] demo_register;
integer k;
.....
initial
        for (k=4; k; k = k-1)
          begin
           demo_register [k+10] = 1'b0;
           demo_register[k+2] = 1'b1;
          end
```

Forever.

El bucle *forever* causa una ejecución repetitiva de alguna sentencia. La peculiaridad es que se ejecuta de forma incondicional.

Ejemplo: vamos a generar una señal de reloj, cuyo semiperiodo se haya definido como 50 unidades:

```
initial
        begin
        clock = 0;
        forever
                begin
                #50 clock = 1;
                #50 clock = 0;
                end
        end
```

Disable.

La sentencia **disable** se puede utilizar para la terminación de bucles.

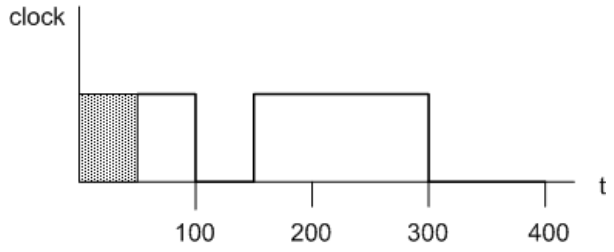
Ejemplo: supongamos que tenemos un registro de 15 bits, y queremos conocer la posición, comenzando por el bit menos significativo, en la que aparece el primer 1 lógico en ese registro. Se

podría utilizar el siguiente módulo:

```
module  find_first_1 (A_word, trigger, index_value);
input    trigger;
input [15:0] A_word;
output [3:0] index_value;
reg [3:0] index_value;

always @ (posedge trigger)
        begin
             index_value = 3'b000;
              for (index_value = 0; index_value<=15; index_value=index_value+1)
              if (A_word[index_value] == 1) disable;
        end
endmodule
```

FORK.....JOIN

Cuando un procedimiento o una sentencia lleva asociado una instrucción compuesta, ya hemos dicho que ésta ha de comenzar con *begin* y terminar con *end*. La instrucción compuesta se ejecuta de modo secuencial, es decir por orden. Por ejemplo, si queremos generar la siguiente señal de reloj para un testbench, tendríamos que escribir el procedimiento que se muestra.




```
initial
        begin
        #50   clock = 1'b1;
        #50   clock = 1'b0;
        #50   clock = 1'b1;
        #150 clock = 1'b0;
        end
```

La construcción ***fork***... ***join*** puede sustituir a *begin* .... *end*, aunque su significado es diferente. Todas las instrucciones simples encerradas entre *fork* y *join* se ejecutan de forma simultánea, no secuencial. Por ejemplo, la anterior señal de reloj se podría haber generado con el siguiente procedimiento:

```
initial
        fork
        #50   clock = 1'b1;
        #100 clock = 1'b0;
        #150  clock = 1'b1;
        #300  clock = 1'b0;
        join
```

La construcción *fork...join* no es soportada por las herramientas de síntesis, pero es útil a la hora de realizar *testbenches*.

PROCEDURAL ASSIGNMENTS

Las asignaciones a las variables tipo registro que se realizan dentro de procedimientos como hemos visto hasta aquí, se denominan asignaciones funcionales (*procedural assignments*), a diferencia de las asignaciones continuas (*continuous assignment*) que hemos estudiado anteriormente para asignar valores a variables tipo *wire*.

Ya hemos dicho que una asignación funcional siempre ha de asignar un valor a una variable tipo registro, nunca a una variable de red. Sin embargo, todavía no hemos visto cómo asignar un retraso a este tipo de asignaciones. Es lo que veremos ahora. Lo dividiremos en dos bloques: asignaciones bloqueantes y asignaciones no bloqueantes.

Asignaciones bloqueantes (*blocking assignments*).

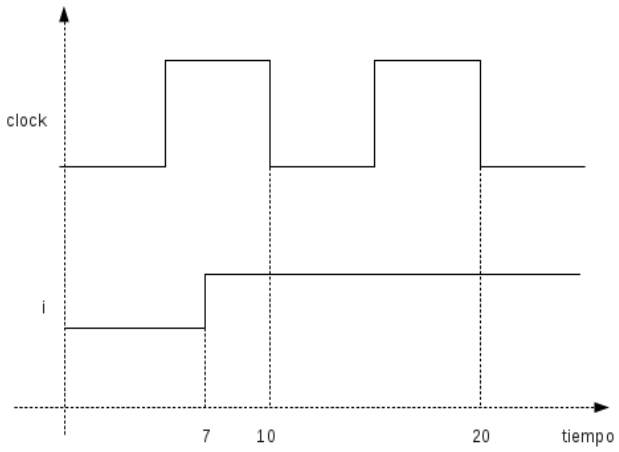
Las asignaciones bloqueantes son aquellas bloquean el flujo del procedimiento hasta que se terminan de ejecutar.

Existen dos formas de asignaciones bloqueantes:

- **Inter-statement delay**:

```
initial
        begin
        clock = 0;
        i=0;
        end
always
        #5 clock = ~ clock;

initial
        #7   i=clock;
```

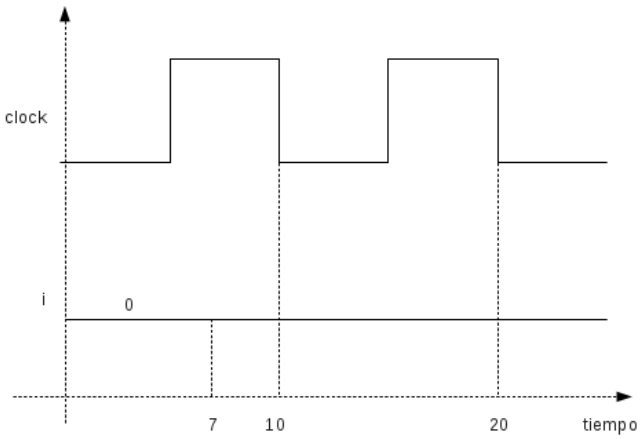
Lo que ocurriría en este caso es lo siguiente: en t=0, los valores de *i* y de la señal de reloj valen 0 lógico. Respecto a la señal de reloj, se hace que su periodo sea de 10 unidades de tiempo (o su semiperiodo 5 unidades de tiempo). En el último procedimiento ocurre lo siguiente: primero se esperan 7 unidades de tiempo, y se asigna a *i* el valor que tome la señal de reloj justo en ese instante, en las 7 unidades de tiempo. La salida que produciría esto se muestra en la siguiente figura:



- **Intra-statement delay**:

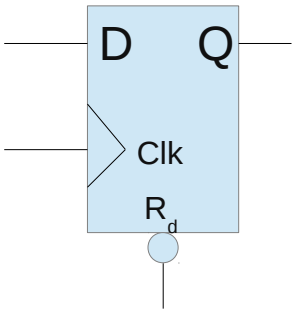
```
initial
        begin
        clock = 0;
        i=0;
        end
always
        #5 clock = ~ clock;

initial
        i = #7 clock;
```

Este caso de retardo es el que se ha escrito para la variable *i*. El retardo se pone después del signo = en lugar de antes. Al igual que antes, la variable *i* vale 0 en el instante inicial y generamos la misma señal de reloj. Al llegar al último procedimiento, lo primero que se hace es evaluar en ese instante el valor de *i*. El valor de *i* en ese instante es el valor de la señal de reloj en el instante inicial, es decir, 0 lógico. Sin embargo, se asigna dicho valor 7 unidades de tiempo después, por lo que el cronograma en este caso sería el siguiente:



En cualquiera de los dos casos de retardo vistos, la asignación es bloqueante, ya que hasta que no pasa el tiempo definido en el retardo, no se ejecutaría la siguiente instrucción dentro del procedimiento.

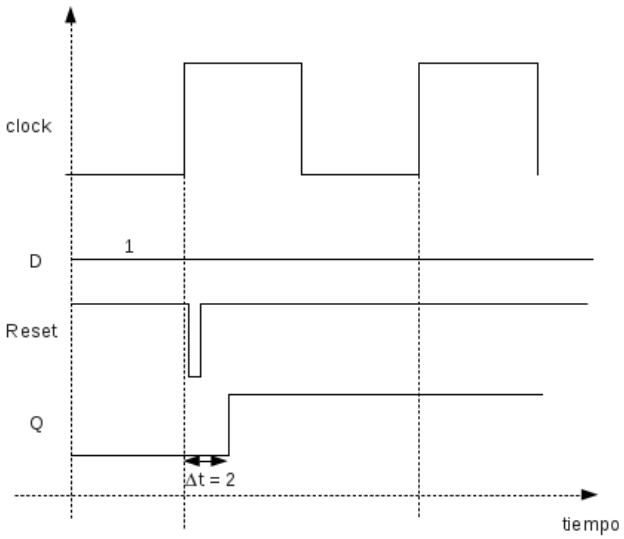
Por ejemplo, supongamos que queremos definir un flip-flop tipo D con una entrada de reset asíncrono activa en baja. Supongamos también que el tiempo que tarda en cambiar la salida desde que se produce un flanco activo de reloj es el mismo tiempo que tarda la salida en cambiar desde que se activa el reset. Dicho tiempo es de 2 unidades de tiempo. Si escribiésemos el siguiente código sería incorrecto:




```
// flip flop tipo D con reset asíncrono


module  ff_D (Q, D, clk, Reset);
input   D, clk, Reset;
output  Q;
reg     Q;

always @ (posedge clk or negedge Reset)
        if (Reset == 0)
          Q= #2 0;
        else
          Q= #2 D;

endmodule
```

Según vemos en el código, supongamos que se da un flanco positivo que hace que la salida tenga que cambiar desde 0 hasta 1 lógico (estaríamos en el caso *else*). En el instante que se da el flanco positivo del reloj, se leería el dato D, y dos unidades de tiempo más tarde se asignaría ese valor a la

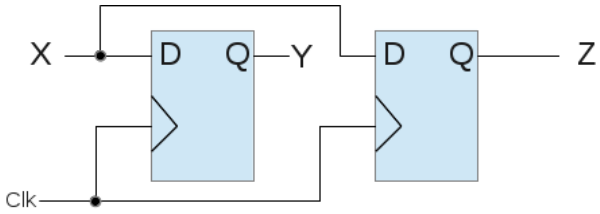
salida Q. Pero, ¿qué ocurre durante esas dos unidades de tiempo? Pues el procedimiento quedaría bloqueado (de ahí el nombre de asignaciones bloqueantes), y aunque hubiese otro flanco activo del reloj o se activase el reset durante ese tiempo, el compilador lo saltaría. El procedimiento queda bloqueado durante esas dos unidades de tiempo. Aunque para la mayoría de casos prácticos el funcionamiento del flip-flop sería correcto, podría haber situaciones en las que no funcionase correctamente. Por ejemplo, según vemos en la siguiente figura, la activación del reset que aparece no la detectaría.



Dicho problema aparece igualmente si se introduce el comportamiento de un circuito combinacional. La forma de evitarlo, es utilizando las asignaciones funcionales no bloqueantes.

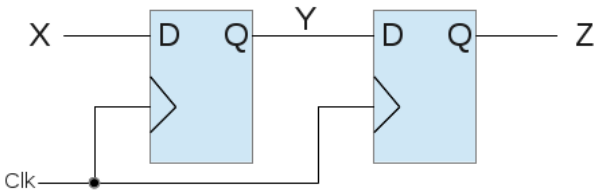
Asignaciones no bloqueantes (*non-blocking assignments*).

Las asignaciones no bloqueantes utilizan un operador diferente. El operador es "<=" en lugar de "=". Las asignaciones no-bloqueantes se ejecutan en dos pasos: para un tiempo dado, la parte de la derecha de todas las asignaciones no bloqueantes dentro de un procedimiento son evaluadas. Una vez que todas y cada una de ellas han sido evaluadas, su valor es asignado a la variable que aparece a la izquierda de la asignación.




```
// flip flops en paralelo
//asignación bloqueante

module  ff (Z, X, clk);
input   X, clk;
output  Z;
reg     Z, Y;

always @ (posedge clk)
        Y=X;
        Z=Y;
endmodule
```

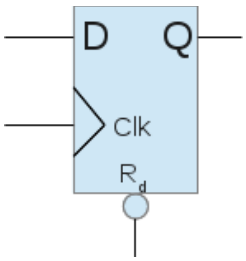
En este caso la asignación es bloqueante. Dentro del procedimiento, lo primero que se hace es asignar a *Y* el valor de *X*. Una vez hecho esto se asigna a *Z* el valor de *Y* actualizado (que es *X*).




```
// registro de desplazamiento
// asignación no bloqueante

module  reg (Z, X, clk);
input   X, clk;
output  Z;
reg     Z, Y;

always @ (posedge clk)
        Y<=X;
        Z<=Y;
endmodule
```

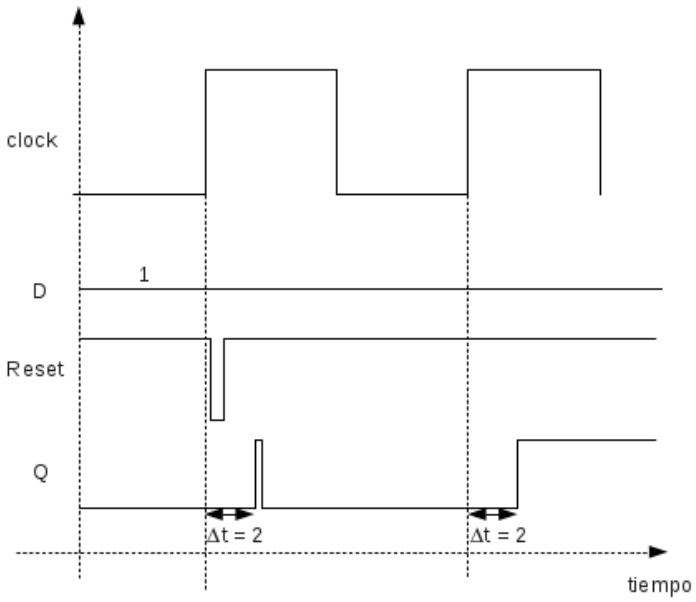
En este segundo caso la asignación es no bloqueante. Lo que ocurriría es lo siguiente: cuando se llega a un procedimiento, en este caso al único que existe y que comienza por *always*, lo primero que se hace es evaluar la parte derecha de las asignaciones, en este caso se evalúa el valor de *X* y de *Y*, pero se hace en paralelo, sin haber actualizado todavía el valor de *Y*. Una vez que los valores de *X* y de *Y* han sido evaluados, se actualizan de forma simultánea las variables *Y* y *Z* que aparecen a la izquierda de las asignaciones.

Con la asignación no-bloqueante, el *intra-statement delay* no bloquea. Por ejemplo, volvamos al anterior caso de flip-flop tipo D




```
// flip flop tipo D con reset asíncrono
//asignación no bloqueante

module  ff_D (Q, D, clk, Reset);
input   D, clk, Reset;
output  Q;
reg     Q;

always @ (posedge clk or negedge Reset)
        if (Reset == 0)
          Q<= #2 0;
        else
          Q<= #2 D;

endmodule
```

En este caso, si se da un flanco activo del reloj la salida tarda dos unidades de tiempo el cambiar exactamente igual que antes. Sin embargo, durante esas dos unidades de tiempo el procedimiento no quedaría bloqueado, si no que si se da otro flanco activo del reloj o se activa el reset asíncrono, el flip-flop lo detectaría. Podemos ver que el cronograma que se muestra a continuación es distinto al anterior.



Como regla general, a la hora de utilizar asignaciones funcionales para describir módulos sintetizables, es buena idea seguir las siguientes reglas:

- No mezclar "<=" y "=" dentro de un mismo procedimiento.
- "<=" es más adecuado para describir flip-flops y siempre que se quieran asignar retardos a la salida de los dispositivos.
- "=" es más adecuado en procedimientos que describen lógica combinacional sin asignar retardos.