Se completa con esta actividad el modelado de la segunda clase de sistemas, los sistemas secuenciales, esenciales para comprender el funcionamiento interno del procesador. En los siguientes apartados, se repasan los conceptos de máquinas secuenciales y se dan indicaciones sobre su modelado en Verilog. Se verá también un ejemplo de circuito secuencial típico, una unidad de control, cuyo esquema puede adoptarse como plantilla para diseñar un circuito secuencial genérico. Al final del documento se les propone una actividad que, si bien no es obligatoria y no incidirá en la nota, es muy recomendable su realización para entender el funcionamiento de los circuitos secuenciales.

INTRODUCCIÓN: MODELADO DE SISTEMAS SECUENCIALES

Un circuito lógico secuencial es que tiene la propiedad de que su salida no depende solamente de sus entradas presentes sino también de la secuencia de sus entradas pasadas. Esa dependencia se debe a que el circuito contiene un "estado" almacenado en elementos de memoria (que depende de las entradas pasadas y en último término de su valor inicial). Dicho de otra forma, el estado es la colección de todos los elementos de almacenamiento cuyos valores en un momento dado contienen toda la información necesaria sobre el pasado para dar cuenta del comportamiento futuro del circuito. Dado que las componentes del estado son variables binarias (por ejemplo, k biestables), tendremos un conjunto finito de posibles estados, limitado por todas sus posibles combinaciones (2^k posibilidades de estados diferentes). Por ello, a los circuitos secuenciales se les suele llamar también "autómatas o máquinas de estados finitos".

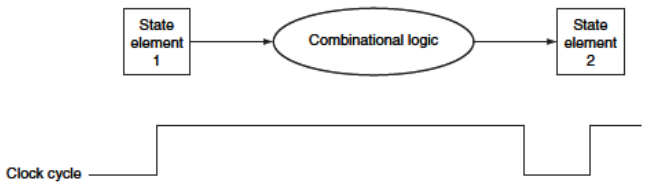
LA SEÑAL DE RELOJ

Los estados en la mayoría de diseños de circuitos secuenciales sólo cambian en instantes específicos que vienen determinados por una señal de reloj periódica. Por ello, este tipo de sistemas reciben el nombre de síncronos. La señal de reloj está caracterizada por su frecuencia (la inversa es su periodo o tiempo de ciclo) y por su ciclo de trabajo (porcentaje del periodo total en el que la señal está en estado 1). Un ciclo de trabajo diferente al 50% hará que la forma de onda no sea estrictamente cuadrada. Hay que crear esta señal de reloj en los *testbench* para introducirla como entrada en nuestros módulos secuenciales. El siguiente fragmento de código modela una señal de reloj de 80 ns de periodo y ciclo de trabajo del 25%. Esto se debe a que el bloque **always** se repite indefinidamente al no haber condiciones de cambio en sus argumentos y la señal de reloj está a alta 20ns y después a baja 60ns. En total, tenemos un ciclo de 80ns (20 + 60) de duración, y de esos 80ns está a alta 20ns, luego 20/80=1/4, en porcentaje un 25%.

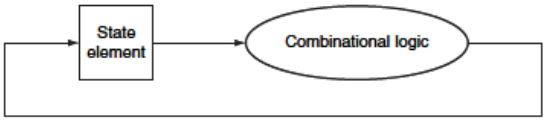
```
`timescale 1 ns / 10 ps
reg clk;

// generación de reloj clk
always //siempre activo, no hay condición de activación
begin
  clk = 1;
  #20;
  clk = 0;
  #60;
end
```

Los momentos particulares en los que los estados pueden cambiar suelen ser los flancos de la señal de reloj (es muy común usar el flanco de subida para ello). Los sistemas que presentan esta propiedad se denominan síncronos activados por flanco y son los que trataremos fundamentalmente aquí. En Verilog podemos usar los operadores lógicos **posedge** y **negedge** aplicados a la señal de reloj para detectar los instantes en que se dan los flancos de subida y de bajada, respectivamente, en las listas de sensibilidad de las construcciones **always**, por ejemplo. La ventaja de usar los flancos es que el proceso de lectura de las entradas en los elementos con estado es esencialmente instantáneo, eliminando las variaciones que se producirían si ese proceso ocurriera en tiempos ligeramente diferentes. El principal requerimiento de diseño de este tipo de sistemas es que las señales que se memorizan en los elementos de estado tienen que ser válidas en el momento del flaco de subida. Se entiende por válidas el que sean estables (no estén cambiando) y que no cambien mientras que las entradas de los bloques que las generan a su vez no cambien. Un bloque combinacional que genere

estas señales a memorizar cumple este requisito, ya que dará señales válidas siempre que esperemos el tiempo suficiente.



La figura ilustra como un elemento de estado 1 que puede cambiar su valor en el flanco de subida de la señal de reloj durante el resto del ciclo se mantiene estable, proporcionando la entrada a un bloque combinacional que a su vez dispone del resto del periodo para dar su salida a otro elemento de estado 2. Este elemento 2 muestreará sus entradas en el nuevo flanco de subida. Siempre que el retardo del bloque combinacional no supere el valor del tiempo de ciclo el funcionamiento será correcto. Basar los cambios en los flancos permitiría incluso la situación de la siguiente figura, en la que el elemento combinacional procesa el estado y genera las señales de control del mismo elemento de estado. En la práctica hay que tener cuidado con esta situación para evitar "carreras" o cambios transitorios indeseados. En nuestros modelos Verilog en los que por simplicidad, no hemos introducido retardos en los elementos combinacionales y responderían instantáneamente, podríamos tener problemas de múltiples cambios de estado indeseados o indeterminación en los estados de los elementos de almacenamiento. Veremos cómo evitar esos problemas más adelante.



Como ejemplo de elemento de almacenamiento, veamos un biestable activado por flanco. Este biestable denominado D puede ser usado fácilmente como unidad básica para implementar registros u otros módulos secuenciales. Posee una entrada D y un estado Q interno que también es su salida. El estado o salida Q dependerá de la entrada y de su valor previo. La tabla de verdad del biestable es la siguiente:

Reloj	**D**	**Q**	**Q_{previo}**
Flanco Subida	0	0	X
Flanco Subida	1	1	X
0	X	Q_{previo}	
1	X	Q_{previo}	

El estado Q debe modelarse con una variable de tipo **reg** (con almacenamiento). Las asignaciones a variables que representan estados deben realizarse usando una construcción nueva que es el operador '**<=**' (*asignación no bloqueante*), por las razones que veremos en el siguiente apartado. Una implementación en Verilog de un módulo de este tipo (con una entrada de reset asíncrona añadida) sería:

```
module ffd (input wire clk, input wire reset, input wire d, output reg q);
//reset asíncrono, ya que es independiente del flanco del reloj

always @(posedge clk, posedge reset)   //--> cuando se de el flanco en clk o reset
  if (reset)
    q <= 1'b0; //asignación no bloqueante q = 0
  else
    q <= d; //asignación no bloqueante q = d

endmodule
```

La entrada de reset es importante porque nos permite llevar los estados almacenados en los biestables (y por tanto en nuestro circuito secuencial) a un valor inicial bien determinado mediante una señal de control común para todo el sistema.

En el biestable anterior, el estado 'sigue' la entrada D, cambiando a este valor en cada flanco de subida del reloj. En algunas ocasiones, no deseamos este cambio con cada ciclo de reloj, sino que se realice bajo ciertas condiciones (cuando modelemos que un registro deba ser actualizado sólo en determinadas ocasiones, por ejemplo). En ese caso, dispondremos de una señal explícita de escritura para esos elementos de estado. De igual forma que anteriormente, combinaremos esa señal de escritura con el flanco de subida del reloj para que el cambio tenga lugar únicamente en el flanco de subida.

```
module ffdc (input wire clk, reset, carga, d, output reg q);

always @(posedge clk, posedge reset)
  if (reset)
    q <= 1'b0; //asignación no bloqueante q=0
  else
    if (carga)
      q <= d; //asignación no bloqueante q=d

endmodule
```

IMPLEMENTACIÓN DE SISTEMAS SECUENCIALES: ASIGNACIONES BLOQUEANTE Y NO BLOQUEANTE

La principal novedad en el Verilog de los ejemplos anteriores es la llamada *asignación no bloqueante o concurrente* "<=". La asignación procedural convencional "=" que hemos usado hasta ahora la denominaremos *asignación bloqueante*. Como regla general, usaremos la nueva asignación no bloqueante dentro de bloques procedurales (**always**, **initial**,…) en sistemas secuenciales cuando le asignemos valores a variables de estado como la **q** del ejemplo. En cambio, la asignación bloqueante la reservaremos para las asignaciones en las partes combinacionales de nuestros sistemas como hemos hecho hasta ahora. Ambas operaciones asignan valores a variables que han de ser de tipo **reg**, la diferencia está en que las variables que son asignadas mediante la no bloqueante no cambian inmediatamente, sino que se determina el valor a asignarles en ese instante y las asignaciones propiamente dichas se realizan simultáneamente en todas las variables antes de avanzar el tiempo de simulación al siguiente instante. Por ejemplo, el siguiente código produce resultados indeterminados, dependientes de la implementación del intérprete Verilog

```
a = b;
b = a;
```

Son dos asignaciones bloqueantes que ocurren en el mismo tiempo de simulación (no hay operador retardo # entre ellas). Representarían una 'carrera' o riesgo dinámico en hardware real. En cambio, el código

```
a <= b;
b <= a;
```

haría que `a` y `b` intercambiasen sus valores iniciales en el momento de la ejecución de esas líneas y sin usar ninguna variable intermedia.

Otra variación en la sintaxis de las asignaciones procedurales (bloqueantes o no) es el retardo intra-asignación. Un operador retardo colocado entre la asignación y el valor a asignar causa que se calcule el valor a asignar en ese momento de la simulación, pero que no se realice la asignación hasta que pase el tiempo de simulación dado por el retardo. Si en particular la asignación es no bloqueante, el sistema pasaría a las siguientes construcciones (sin espera en el tiempo de simulación, de ahí su nombre) pero teniendo presente que en cuanto el tiempo de simulación alcance el retardo previsto la variable a asignar cambiará al valor previamente determinado. Por ejemplo,

```
a <= #3 b;
//Sentencias siguientes
…
```

Causará que se le asigne a la variable 'a' el valor de la 'b' del instante actual, pero dentro de tres unidades de tiempo. Entretanto, las sentencias siguientes se ejecutarán (no se bloquean) hasta que el tiempo avance a ese momento. En cambio,

```
a = #3 b;
//Sentencias siguientes
…
```

Aquí ocurre que se determina el valor de 'b' en el momento actual, se esperan las tres unidades de tiempo de simulación y sólo entonces se procede con las sentencias siguientes (se han bloqueado).

Ahora tenemos los elementos adecuados para proponer una solución al problema visto anteriormente de usar módulos combinacionales de retardo cero en los sistemas secuenciales. El uso de asignaciones no bloqueantes garantiza que el cambio de estado se produzca completo en el momento adecuado (el del flanco de subida de reloj, normalmente), sin transitorios temporales. Supongamos además que introducimos un pequeño retardo (una pequeña fracción del ciclo de reloj) en el cambio de estado de un elemento de almacenamiento. Este retardo nos permitirá visualizar mejor los cambios en el sistema debidos a las transiciones de un estado a otro.

En Verilog, lo haremos introduciendo de paso una nueva notación en la declaración de un módulo. Esta notación sirve para expresar parámetros o constantes que podamos cambiar de valor en cada instanciación del módulo:

```
module nombremodulo #(parameter nombreparam = valorpordefecto) (input wire a, input wire b, …);
```

en la instanciación del módulo, podremos cambiar el valor del parámetro a un nuevo valor deseado

```
nombremodulo #(valor_real) pepemodulo(test_a, test_b, …);
```

En el ejemplo del biestable:

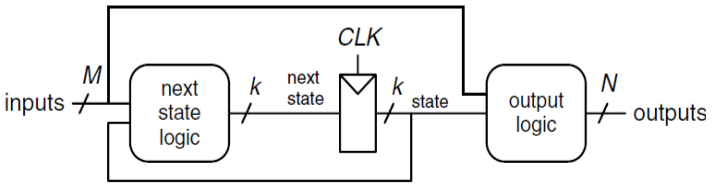
```
module ffd #(parameter retardo = 1)(input wire clk, reset, d, output reg q);
//reset asíncrono, es independiente del flanco del reloj

always @(posedge clk, posedge reset)
  if (reset)
    q <= #retardo 1'b0; //asignación no bloqueante q = 0 con retardo
  else
    q <= #retardo d; //asignación no bloqueante a q = d con retardo
endmodule
```

y ahora instanciamos el módulo con valor 3 para el retardo de la siguiente forma, poniendo el símbolo lista de parámetros "#" y el valor del parámetro (o parámetros, separados por comas y en orden de su declaración) **antes** del nombre de la instancia concreta:

```
ffd #(3) miffd(t_clock, t_reset, t_d, t_q);
```

MÁQUINAS DE ESTADOS FINITOS

La figura muestra un ejemplo de estructura general de máquina de estados, que en este caso corresponde a una máquina de Mealy. En este tipo de máquina, las salidas dependen de las entradas y los estados (en las máquinas de Moore, las salidas dependen sólo de los estados).



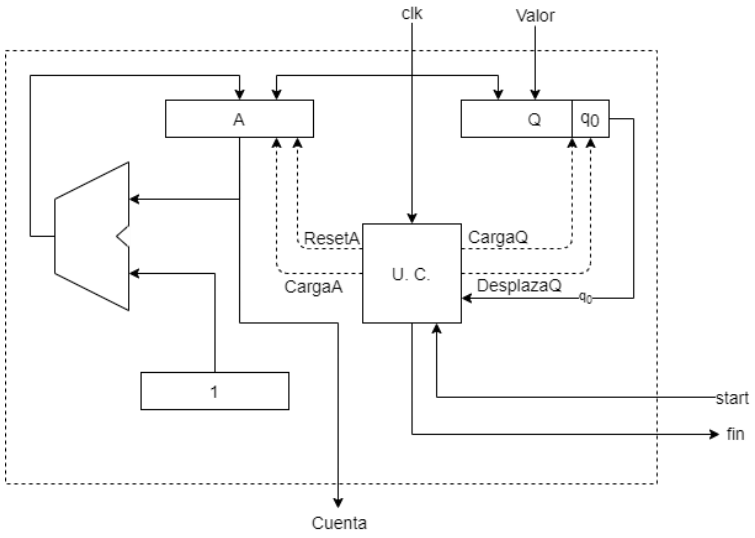
En general, una máquina de estados se describe como un *conjunto de estados* y dos funciones: la *función de transición de estado* y la *función de salida*. El conjunto de estados representa todas las posibilidades que se pueden formar con los elementos de almacenamiento (con k biestables como elementos de almacenamiento, tendremos 2^k estados posibles). La función de transición de estado cambia al estado siguiente a partir del estado actual y las entradas. En la figura, esta tarea se reparte en dos bloques, uno combinacional que a partir de las entradas y el estado actual determina cuál será el estado siguiente y un bloque secuencial que almacena el estado actual y realiza el cambio de estado en sí usando la información del bloque anterior cuando llega el flanco de subida del reloj. Por otro lado, la función de salida produce las salidas a partir del estado actual y las entradas en un bloque puramente combinacional. Como sabemos, los estados cambiarán sólo en los flancos de subida, momento a partir del cual las funciones de transición y de salida calcularán sus nuevos valores hasta el nuevo flanco.

EJEMPLO DE MÁQUINA DE ESTADOS

Veamos como ejemplo de máquina de estados la Unidad de Control de un circuito que realiza la función de contar los bits puestos a '1' que tenga un registro de 3 bits.

El módulo representado en la figura tendría la siguiente declaración

module cuenta1(**input wire** [2:0] Valor, **input wire** start, clk, **output wire** [3:0] Cuenta, **output wire** fin);



Se dispone de un registro Q de 3 bits donde inicialmente pondremos el número cuyos bits a 1 pretendemos contar. Para ello, el registro cuenta con una señal de CargaQ que lo inicializa y con la habilidad de desplazar sus bits a la derecha controlada por la señal DesplazaQ. Disponemos también de un sumador y de un registro A con la posibilidad de ser reseteado y cargado donde quedará el resultado final. Todo ello lo controlaremos mediante la unidad de control que será una máquina de estados finitos con la siguiente declaración de módulo

module uc(**input wire** q0, start, clk, **output wire** CargaQ, DesplazaQ, ResetA, CargaA, Fin)

La unidad de control tiene como entradas el bit menos significativo q0 de Q, la señal de start que reinicia al estado inicial la unidad de control y el reloj clk. Como salidas posee CargaQ para inicializar el registro Q de un valor externo, DesplazaQ que provoca su desplazamiento, ResetA para inicializar a cero el registro A, CargaA para inicializar el valor de A (en esta ocasión del valor entregado por el sumador) y la señal Fin para indicar a un dispositivo externo la finalización de la operación de conteo. El algoritmo es muy simple, se debe inicializar el registro Q con el valor cuyos bits a 1 queremos contar y resetear el contenido de A, a continuación, comprobamos el valor de q0 y si es 1 debemos sumar a A una unidad y desplazar Q. Esto se repite tantas veces como bits tenga Q y al acabar debemos activar la señal de Fin.

A partir de este algoritmo, se pueden proponer los siguientes estados para la unidad de control y las transiciones entre ellos, que coinciden con la secuencia de pasos elementales a llevar a cabo para 3 bits en Q:

Estado	Salidas a activar	Observaciones	Siguiente Estado
S0	CargaQ, ResetA	Estado inicial de carga del operando en Q e inicialización de A	S1
S1	Si q0==1, CargaA	Carga A con la salida del sumador	S2
S2	DesplazaQ	Actualizar q0	S3
S3	Si q0==1, CargaA	Carga A con la salida del sumador	S4
S4	DesplazaQ	Actualizar q0	S5
S5	Si q0==1, CargaA	Carga A con la salida del sumador	S6
S6	Fin	Estado final de la operación	S6

Los estados S1, S3, y S5 por un lado, y los estados S2 y S4 por otro, son muy similares entre sí, pero necesarios para contar las etapas necesarias para Q de tres bits. Las transiciones entre estados (primera y última columna) son muy simples en este caso particular ya que no dependen de las entradas. Básicamente encadenan un estado con el siguiente hasta llegar al S6 final, del que ya no se sale hasta una nueva operación (indicada con un nuevo start).

Deberemos implementar la de transición de estados y la de salida para la unidad de control de acuerdo a la tabla anterior. Una posible implementación de la máquina de estados en Verilog, estructurada en los bloques antes descritos sería

```
module uc(input wire q0, start, clk, output wire CargaQ, DesplazaQ, ResetA, CargaA, Fin)

reg [2:0] state, nextstate; //Variables para los estados actual y siguiente
                           //Tres bits son suficientes para los siete estados
//Codificación de los estados

parameter S0 = 3'b000;      //declaracion de constantes que representan estados
parameter S1 = 3'b001;
parameter S2 = 3'b010;
parameter S3 = 3'b011;
parameter S4 = 3'b100;
parameter S5 = 3'b101;
parameter S6 = 3'b110;

// Registro de estado, cambia en cada flanco ciclo de reloj por el nuevo estado o
// se inicia en caso de flanco de subida de start al estado inicial

always @ (posedge clk, posedge start)
  if (start)
    state <= S0;
  else
    state <= nextstate;

//Función de Transición

always @(*)      // (*) significa cualquier cambio en alguna variable del bloque
  case (state)
    S0: nextstate = S1;
    S1: nextstate = S2;
    S2: nextstate = S3;
```

```
    S3: nextstate = S4;
    S4: nextstate = S5;
    S5: nextstate = S6;
    S6: nextstate = S6;
    default: nextstate = S0;
  endcase

// Función de Salida

assign CargaQ = (state == S0)? 1:0; //a 1 si el estado es S0
assign DesplazaQ = ((state == S2)|(state == S4))? 1:0;
assign ResetA = (state == S0)? 1:0;
assign CargaA = (q0 & ((state == S1)|(state == S3)|(state == S5)))? 1:0;
assign Fin = (state == S6)? 1:0;

endmodule
```

En el código anterior la única novedad es el uso de la sentencia **parameter**, ahora para nombrar las configuraciones de bits que representan cada estado y trabajar más cómodamente con ese nombre simbólico. Es importante comprobar que tenemos el suficiente número de bits en la variable de estado para que variando sus configuración podamos representar todos los estados que deseamos. En la sentencia **case** es importante poner un caso por defecto de forma que si por alguna perturbación o transitorio en un circuito real, el estado contuviera una configuración inválida, en la siguiente transición la hagamos desaparecer y volver a un estado conocido. El bloque Función de Transición, que determina el siguiente estado, se ha implementado de forma procedural a pesar de que la forma preferida de este tipo de elementos es el **assign**, debido a que resulta mucho más claro hacerlo con la sentencia procedural **case** dentro de un **always**. En la Función de Salida se hace uso de **assign**s debido a que es un bloque combinacional y que resulta relativamente compacta expresarlas así. Hay que destacar el uso frecuente del operador condicional, similar al del C: (exp_condicional **?** exp1 **:** exp2) que evalúa exp_condicional y si es verdadera devuelve exp1 y en caso contrario devuelve exp2. Es importante que las salidas queden determinadas para cualquier combinación de estado y entradas. En resumen, lo que realiza este código es: activar CargaQ y ResetA sólo en el estado S0 inicial, activar DesplazaQ sólo en los estados S2 y S4, activar CargaA (que toma el A incrementado de la salida del sumador) únicamente si q0 es 1 y estamos en los estados S1, S3 ó S5 y activar Fin al llegar al estado final S6.

ACTIVIDAD RECOMENDADA

A partir de los ficheros suministrados con esta actividad, realizar un módulo testbench cuenta1_tb.v que produzca las entradas al módulo global: la señal de reloj como se ha visto antes, un breve pulso inicial en la señal de start (ponerla a alta y un breve retardo después, a baja), la entrada del valor a contar y observar la salida de Fin y de Cuenta. En alguno de los ficheros de los módulos principales se ha incluido el código de los módulos de los que éstos dependen para reducir el número de ficheros.

Observar también las señales de la unidad de control, usando el gtkwave que permite 'abrir' el módulo global cuenta1 y acceder a sus señales internas. Suponiendo que en la sentencia timescale el valor del retardo (primer parámetro) es 1ns, intentar decidir experimentalmente cuál sería la frecuencia máxima de reloj (o el periodo mínimo) al que podría funcionar este circuito.