# UNIT 2

## Modeling Recommendations

María José Avedillo

Esta presentación asume un conocimiento básico de Verilog y se centra en exponer recomendaciones para la escritura de códigos, con especial énfasis en la codificación para síntesis.

# Modeling Recommendations. General

- Define specifications as tightly as practical possible in terms of input, output, associated timing and functionality before writing HDL models
- Determines a sound architecture and partition accordingly before attempting to code a model

- Write structure code (functions, tasks ..)
- Document code using comments: module functionality, signal use ….
- Make use of abstract data types to make models easier to read and maintain. Use define directive to represent data values
- Use meaningful signal names

- When writing for synthesis, keep in mind the hardware intent and the synthesis modeling style (see Coding for Synthesis Section)
- Be careful with Verilog characteristic
  - Case sensitivity
  - Unintentional bit width mismatched since they are not detected by compilers
- Be careful with blocking and non-blocking assignments

MHCAD. UNIT 2 2

Comenzamos por algunas recomendaciones de carácter muy general, algunas de ellas, aplicables no sólo a los lenguajes de descripción de hardware.

En primer lugar y para evitar diseñar correctamente algo incorrecto debemos definir, tan ajustadamente como sea posible desde un punto de vista práctico, las especificaciones del diseño en términos de entradas, salidas, funcionalidad y temporización.

También es importante desde esta misma perspectiva, determinar una arquitectura y un particionamiento adecuados antes de comenzar a codificar un modelo.

Desde el punto de vista del código en sí, es recomendable:

Dotarlo de estructura (por ejemplo, usando funciones y tareas).

Documentarlo utilizando comentarios para describir funcionalidad, uso de las señales,

Utilizar nombres simbólicos para representar valores de datos mediante por ejemplo la directica define

Y usar para los identificadores de señales, de modulos etc.. nombres con significado

Como recomendaciones muy generales de los HDLs es importante, cuando se escriben códigos que van a ser sintetizados, tener en mente el harware que se persigue y adoptar un estilo de modelado particular sobre el que volveremos en esta presentación.

Por último, ya en concreto para Verilog, recordamos, sobre todo para quienes han utilizado otros lenguajes que es sensible a minúsculas y mayúsculas y que hay que ser cuidadoso con desajustes no intencionados entre el número de bits de las señales involucradas en una sentencia, puesto que el compilador no las detecta

Y la importancia de distinguir entre asignamientos blocking y no blocking

# Blocking and non-Blocking assignments

- It is important to understand the semantics of the language and their timing implications. With this understanding and detailed analysis of the behaviour of a model, decisions about the proper construct should be made
- For example next code exhibit timing problems that can be correct substituting blocking by non blocking assignments

```
// Value taken by q correspond to value of d 4 time units
// after the event (change of c or change of d) that activates
// execution. Value taken by q_b corresponds to value taken by
// d 7 time units after that event. If d changes between  both readings
//  results are erroneous
Module latch (input d, c, output reg q, q_b);
Always @ (c or d)
If (c) begin
q = #4 d;
q_b = #3 ~d;
end
endmodule
```

MHCAD. UNIT 2 3

Respecto a estas últimas construcciones, es importante comprender la semántica y las implicaciones temporales de ambos tipos de asignamientos, de forma que se pueda elegir en función del comportamiento deseado la construcción adecuada en cada caso

El ejemplo que se muestra exhibe problemas temporales que pueden corregirse substituyendo asignamientos tipo blocking por asignamientos no blocking.

De acuerdo con este código, *q* toma el valor de *d* cuatro unidades de tiempo después de que se produzca el cambio de *c* o de *d* que activa la ejecución.

El valor que toma *q_b* corresponde al valor de *d* siete unidades de tiempo después de dicho cambio. Si *d* cambia entre ambos asignamientos, el resultado es erróneo puesto que la salida del latch q y su salida complementada podrían ser iguales y no sólo transitoriamente.

Usando asignamientos de tipo nonblocking se pueden modelar retrasos distintos para ambas salidas que evitan los problemas anteriores.

# Modeling Recommendations. Simulation

- (Accuracy) Model to control order of concurrent assignments (Example 1)
- (Accuracy) Ensure event list of always statement are complete (Example 2)
- (Speed) Loop invariant signals should not be contained in a loop
- (Speed) Do not model many small always is possible. If there are many registers being clocked from the same clock source, put them in one process
- (Speed) Reduce the number of signals the simulator must monitor
  - Write models to minimize signals in the event list of an always (resorting to continuous procedural assignments Example 3)

MHCAD. UNIT 2 4

Tanto desde en punto de vista de la corrección de los resultados como desde el punto de vista de la velocidad de las simulaciones debemos seguir algunas normas.

El orden en él que se ejecuten un conjunto de sentencias concurrentes, esto es correspondientes al mismo tiempo de simulación, puede afectar a los resultados. Este orden depende del simulador utilizado. El que un modelo describa el comportamiento deseado no debe depender de dicho orden. Por ello, hay que modelar de forma que controlemos el orden de ejecución de los asignamientos concurrentes como ilustramos en el siguiente ejemplo.

También para evitar que el hardware sintetizado no se corresponda con el comportamiento modelado, hay que asegurarse de que las listas de eventos que disparan la ejecución de un *always* estén completas.

Debe contener todas las señales que aparecen únicamente a la derecha de los operadores de asignamiento en las sentencias asociadas al proceso. Así …

Respecto a la velocidad es conveniente:

Sacar de los bucles todas las señales que no varíen.

No modelar muchos pequeños procesos *always*. Por ejemplo, registros que comparten la misma señal de reloj deben ir en un mismo *always*.

Y reducir el número de señales que el simulador debe monitorizar. El siguiente ejemplo muestra la utilización del asignamiento procedural continuo con este fin.

# Modeling Recommendations. Simulation. Example 1

```
always @ (posedge clock) Y1 = A;
always @ (posedge clock) begin
If (Y1 == 1) Y2 = B; else Y2 = 0;
end
```

- Depending the order of execution of the two always, the pre-synthesis simulation match the post-synthesis simulation or not. If second executes first, they match

Solution: only one always. Both simulations will match

```
always @ (posedge clock) begin
If (Y1 == 1) Y2 = B; else Y2 = 0; Y1 <= A; end
```

MHCAD. UNIT 2 5

Back to Modeling Recommendations. Simulation

Para el código que se muestra en la parte superior de la transparencia las sentencias de asignamiento Y1 = A e Y2 = B ó Y2 = 0 son concurrentes.

Dependiendo del orden de ejecución de los dos procesos always activados por el flanco ascendente de la señal clock, que depende del simulador, podemos tener un valor distinto para Y2.

Si se ejecutan en el orden escrito, el valor de Y1 que se compara con 1 es el actual de A. Si el que está escrito en segundo lugar se ejecuta antes, el valor de Y1 que determina el valor que se asigna a B es el anterior (el que se asignó a Y1 en el anterior flanco de *clock*).

Sin embargo, independientemente del orden en el que se escriban, un sintetizador genera el mismo hardware. En este caso, sólo si la segunda se ejecuta primero, la simulación del código y del circuito sintetizado producen los mismos resultados.

Es posible reescribir el código de forma que aseguremos que efectivamente la segunda se ejecuta primero, que es lo que podemos sintetizar

En este caso se ha utilizado un único always y se ha hecho uso del operador non blocking

# Modeling Recommendations. Simulation. Example 2

Differences between pre-synthesis simulation and post-synthesis simulation due to incomplete event list

- In pre-synthesis simulation *z* does not change if *y* changes with *x* constant
- From this code an AND gate is synthesized which responds to *y* changes with *x* = 1

```
module and_problema(x, y, z);
output z; reg z; input x, y;
always @(x) z <= x & y;
endmodule
```

Back to Modeling Recommendations. Simulation


También en el código aquí mostrado encontramos diferencias entre la simulación antes y después de la síntesis.

Esto significa diferencias entre el comportamiento que describe este código y el comportamiento del circuito sintetizado a partir de él.

En la simulación antes de la síntesis, la señal *z* no cambia si *y* cambia con *x* constante.

Sin embargo, a partir de este código, se sintetiza una puerta AND y por lo tanto, su salida *z* responde a los cambios de *y* para *x* = 1.

# Modeling Recommendations. Simulation. Example 3

```
module mux4_case(a, b, c, d, select, y_out);
input a, b, c, d; input [1:0] select;
output y_out; reg y_out
always @(a or b or c or d or select).
      case (select)
0: y_out = a;
1: y_out = b;
2: y_out = c;
3: y_out = d;
default y_out = 1'bx; endcase
endmodule
```

Two descriptions of MUX

Differences in event list and type of assignments

Second better for simulation

```
module mux4_PCA(a, b, c, d, select, y_out);
input a, b, c, d; input [1:0] select;
output y_out; reg y_out;
            always @ (select)
                        if (select == 0) assign y_out = a; else
                        if (select == 1) assign y_out = b; else
                        if (select == 2) assign y_out = c; else
                        if (select == 3) assign y_out = d; else
                        y_out = 1'bx;
endmodule
```

MHCAD. UNIT 2

Back to Modeling Recommendations. Simulation

7

Estos dos códigos describen multiplexores 4:1 con el mismo comportamiento.

En el primero el *always* está activado por eventos en *a*, *b*, *c*, *d* y *select*, esto es, en cualquiera de las entradas de datos o de selección.

El segundo sólo contiene *select* en la lista de activación. Para que el comportamiento descrito no cambie con respecto al anterior y por lo tanto se describa correctamente la operación de un multiplexor, los asignamientos que se utilizan ahora son de tipo procedural continuo.

Este segundo sería más eficiente para simulación.

# Coding for Synthesis

- When using an HDL as input to a synthesis tool, a coding style which guarantee that the description can be synthesized and that the synthesized circuit behaves as required must be adopted
- Basic rules are:
  - Be aware that not all algorithms are synthesizable
  - Avoid using given constructs since not all of them are supported by synthesis tools
  - Be aware that there are constructs that are ignored by synthesis tools (Delays and Initial)
  - Avoid referencing the same variable in more than one cyclic behaviour so that there are not software races which could result in pre-synthesis and post-synthesis behaviour mismatching
  - Observe basic rules to distinguish combinational and sequential logic (Combinational resources, Sequential resources)
  - It is advisable a synchronous design methodology (resettable synchronous flip-flops as memory elements)
  - Be aware that the HDL description style influences the final result. Anticipate the result produced by the applied synthesis tool. The designer should understand the mappings embedded in the tool (Unit 4)

MHCAD. UNIT 2 8

Nos centramos ahora en la codificación para síntesis.

Ya en el contexto de los ejemplos anteriores, hemos introducido la necesidad de adoptar un determinado estilo de codificación, cuando el código se va a utilizar como entrada a una herramienta de síntesis.

Dicho estilo persigue garantizar que la descripción puede ser sintetizada y que el circuito sintetizado a partir de ella se comporta como queremos.

En última instancia, el estilo concreto viene determinado por la herramienta de síntesis lógica que se use. Sin embargo, hay unas reglas básicas que debemos seguir:

Es importante ser consciente de que no todos los algoritmos se pueden sintetizar.

Además, las herramientas de síntesis lógica no soportan todas las construcciones del lenguaje por lo que hay que evitar determinadas construcciones.

En particular hay algunas que son ignoradas.

Muy relacionado con el control del orden en el que se ejecutan las sentencias concurrentes, hay que evitar referenciar la misma señal en más de un proceso. De nuevo podrían producirse carreras en el software, de forma que el resultado dependa del simulador utilizado y se obtengan discrepancias entre la simulación pre y post síntesis.

Es fundamental aplicar reglas básicas para distinguir la lógica combinacional de la secuencial.

Finalmente es recomendable utilizar una metodología de diseño síncrona, con flip-flops como elementos de memoria, y ser conscientes de que la descripción inicial influye en el circuito final. A este respecto, al codificar debemos anticipar el resultado producido por la herramienta de síntesis aplicada, para lo que el diseñador debe conocer la herramienta. Volveremos a ello en la Unidad 4.

## Coding for Synthesis. Ignored constructs

- Delays
  - The descriptions should not  try to model technological characteristic such as gate delays
  - Do not use the delay control operator (#) for timing (example 4)
- Initial
  - An initial behaviour can be used to initialize a sequential element within a simulation but not for synthesis

MHCAD. UNIT 2 9

Por ejemplo, las construcciones que modelan retrasos y las construcciones *initial* son ignoradas.

La síntesis de un código que las contiene se puede realizar, pero se ignoran. Esto significa que si son críticas para que el código describa el comportamiento deseado, el circuito sintetizado no va implementar la funcionalidad deseada.

Un código para síntesis no debe tratar de modelar características tecnológicas, como son los retrasos que va a a tener una determinada componente.

Ni utilizar el operador de control de retrasos para establecer la temporización como se muestra en el siguiente ejemplo.

De la misma forma, en simulación puede resolverse la inicialización de un elemento de memoria con una construcción initial, pero no en síntesis.

## Coding for Synthesis. Example 4

- A signal *led* which switches every 10 μs

```
‘timescale 1us/ns;
module led_parpadea(led);
output led; reg led;
always begin
#10 led = 0;
#10 led = 1;
end
endmodule
```

•Pre-synthesis simulation shows desired behaviour for signal *led*
•In synthesized circuit signal *led* is constant at 1
•Control operator (#) is ignored

```
module led_parpadea(clk, led);
output led; reg led; input clk;
always @(posedge clk)
led = !led ;
endmodule

// clk clock with a period of 10 us
// a frequency divisor is used if faster clock availa-
ble
```

For synthesis

MHCAD. UNIT 2 10

Este ejemplo ilustra lo anterior

Supongamos que queremos controlar el encendido y apagado de un led, de forma que parpadee cada 10 microsegundos. Para ello necesitamos generar una señal, que hemos denominado *led*, que cambie cada 10 microsegundos.

El código de la izquierda describe este comportamiento utilizando dos sentencias de asignamiento, precedidas de #10, que con la directiva timescale utilizada, temporizan la ejecución de una cada 10 microsegundos

Sin embargo, si sintetizamos este código los operadores de control de retrasos son ignorados y se sintetiza una señal *led* que es constante. Siempre está a 1.

Para síntesis, tenemos que recurrir a un reloj para medir el tiempo. En el código de la derecha, led cambia con cada flanco de la señal clk. Por lo tanto si el periodo del reloj es 10 μs tenemos el comportamiento deseado.

Al sintetizar se obtiene un elemento de memoria controlado por clk, que cambia de estado con cada flanco de su señal de reloj. Si el reloj disponible no es de la frecuencia adecuada, lo generaríamos, usando un divisor de frecuencia, a partir de otro más rápido.

# Coding for Synthesis. Combinational resources

- Combinational logic can be synthesized from:
  - Netlist of Verilog primitives or combinational modules (Example 5)
  - Continuous assignment (signals should not appear both in LHS and RHS of =)
  - Cyclic behaviour (*always) adopting following rules*
    - The sensitivity list should include only level sensitive signals
    - signals should not appear both in LHS and RHS of assignments
    - Completely specified the output for all cases of the input. Failure to do so will produce a design with unwanted latches:
      - Initialize signal before complex loops (Example_6)
      - Avoid incomplete *case* constructions (Example 7)
      - Avoid incomplete conditional constructions (Example 8)
  - Functions and Tasks *adopting following rules*
    - Avoid incomplete *case* constructions
    - Avoid incomplete conditional constructions
    - Do not use timing control construct in tasks

MHCAD. UNIT 2 11

Se sintetiza lógica combinacional a partir de los siguientes recursos:

Un *netlist* que contiene primitivas Verilog o módulos combinacionales

Otro recurso posible es utilizar asignamientos continuos, siempre que la señal que se asigna no se utilice a la derecha del operador asignamiento

En tercer lugar también se sintetiza lógica combinacional a partir de Un *always* que cumple determinadas reglas:
La lista de eventos incluye únicamente señales sensibles a niveles.
Ninguna señal aparece en la parte derecha y en la parte izquierda de los asignamientos.
La salida está completamente especificada para todos las entradas.

Funciones y tareas, adoptando también determinadas reglas similares a las enumeradas para las construcciones *always*. Por supuesto sin incluir construcciones de control de retrasos en las tareas.

Veamos algunos ejemplos:
Ejemplo 5

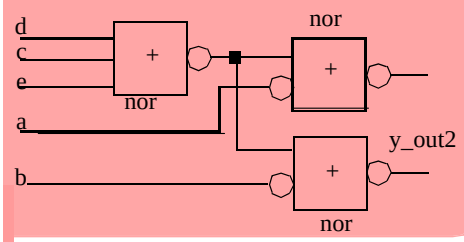
Efectivamente, cuando estamos describiendo un comportamiento combinacional (tanto con un *always* como con otros recursos), es muy importante que la salida este especificada para todas las entradas, De lo contrario se producen circuitos con *latches*.
En los siguientes ejemplos se muestras situaciones en las que es fácil dejar combinaciones de entrada sin asociarles salidas y que hay que evitar.

# Coding for Synthesis. Example 5

Even if the starting point is a gate-level representation (we already know how to connect gates to realize a given functionality) advantages can be obtained from logic synthesis:

- Less complex implementation. Redundant logic is eliminated
- Map to available gates in the library

```
module boole_opt (y_out1, y_out2, a, b, c, d, e);
   input a, b, c, d, e;
   output y_out1, y_out2;
   and (y1, a, c);
   and (y2, a, d);
   and (y3, a, e);
   or (y4, y1, y2);
   or (y_out1, y3, y4);
   and (y5, b, c);
   and (y6, b, d);
   and (y7, b, e);
   or (y8, y5, y6);
   or (y_out2, y7, y8);
endmodule
```




MHCAD. UNIT 2 12

Incluso si ya sabemos como conectar puertas para implementar una determinada funcionalidad; esto es, el punto de partida del diseño es una representación a nivel de puertas, la utilización de una herramienta de síntesis lógica es útil.

Permite obtener realizaciones menos complejas ya que durante el proceso de síntesis se pueden eliminar lógica redundante, generándose implementaciones más simples. De la misma forma se optimiza el mapeado a la librería de celdas disponibles.

El ejemplo mostrado ilustra esto. La red generada por el sintetizador, a la derecha, únicamente utiliza tres puertas, un número muy inferior a las que contiene el netlist que se utiliza como entrada a la herramienta.

# Coding for Synthesis. Example 6

```
module and4_compor (y, x_in);
   parameter word_length = 4;
   input [word_length -1:0] x_in;
   output y;
   reg y;
  integer k;
  always @ x_in
   begin: check_for_0
       y=1;
       for(k= 0; k <= word_length -1; k = k + 1)
          if(x_in[k] == 0)
           begin
               y = 0;
               disable check_for_0;
           end
   end
endmodule
```

Algorithmic description of a four input AND. If the initialization before the for loop is not included:

- The output is not completely specified for the case *x_in* all ones
- Thus, signal y should conserve its value (the one corresponding to previous x_in value) resulting in a sequential behaviour

MHCAD. UNIT 2 13

Cuando se utilizan bucles complejos puede ser relativamente común que no se especifique la salida para alguna condición o condiciones de las entradas.

En el bucle for de este ejemplo se analizan sucesivamente los bits de la señal *x_in*. Cuando se encuentra un cero, la salida se asigna y el bucle finaliza debido a la sentencia *disable*.

La salida no está especificada para el caso correspondiente a que todos los bits de x-in sean 1. Esto significa que en esta situación, la señal *y* debe conservar su valor. Esto es, no sólo depende de las entradas presentes *y* por lo tanto describe un comportamiento secuencial.

Obsérvese que en este código se ha inicializado antes de comenzar el bucle y por lo tanto no resulta en un comportamiento secuencial

# Coding for Synthesis. Example 7

```
Si no se asignan todas las salidas se infieren
latches no deseados
module alu_incomplete(opcode, a, b, alu_out);
output [3:0] alu_out; reg [3:0] alu_out;
input [3:0] a, b;
input [2:0] opcode
always @(a or b or opcode)
case (opcode)
3'b001: alu_out = a | b;
3'b010: alu_out = a ^ b;
3'b110: alu_out = ~ b; endcase
endmodule
```

Latches are inferred in synthesis

Should be complete with default

```
module muxonehot (oneHot, a, b, c, y);
input a, b, c;
input [2:0] oneHot; output y; reg y;
always (oneHot or a or b or c )
begin case (oneHot)
3'b001: y <= a;
3'b010: y <= b;
3'b100: y<= c;
default: y<= 'bx;
//  latches are not inferred
//don´t cares for synthesis
endcase
```

OK

MHCAD. UNIT 2 14

En el código de la izquierda, debido a que no todos los casos de opcode están especificados (esto es lo que llamamos una construcción case incompleta), la herramienta de síntesis lógica infiere la necesidad de utilizar latches para mantener el valor de *alu_out* para aquellas combinaciones de entrada no especificadas.

La solución es completar siempre esa construcción con el *default* como se muestra en el código de la derecha.

Para cualquier combinación de *opcode* distinta de 001, 010 y 100, la salida se ha definido, en este caso, como inespecificaciones. Esto es, estamos indicando que no importan lo que valga, que es distinto de que tenga que conservar su valor.

## Coding for Synthesis. Example 8

**always @ (D or clk) if( clock ) Q_latch = D;**

- **This simple incomplete conditional describes a latch**

MHCAD. UNIT 2 15

También el uso de construcciones condicionales incompletas implica un comportamiento secuencial y deben evitarse al describir lógica combinacional.

De hecho, la manera más simple de describir un latch es utilizando precisamente una construcción condicional incompleta.

La ejecución de la sentencia condicional del código mostrado se dispara si *D* cambia o si *clk* cambia, pero sólo si *clk* está a 1 se asigna a *Q_latch* el valor de *D*.

Si D cambia con clk a cero, la señal *Q-latch* conserva su valo, lo cual se corresponde con el comportamiento de un elemento de memoria tipo *latch*.

Si al modelar lógica combinacional utilizamos condicionales incompletos, la herramienta infiere *latches*

# Coding for Synthesis. Sequential Resources *i*

- When *edge qualifiers* are included in the event control list of an *always*, edge triggered flip-flops are inferred
  - **always @(posedge clk) Q_ff <= D; // flip-flop**
- Other circuits with flip-flops
  - **always @(posedge clk) Count <= Count + 1; //Counter**
  - always @ (posedge clk) y <= a & b  // and gate driving a D flip-flop
- Registered combinational logic
  always @ ( a or b) y <= a & b // and gate
  always @ (posedge clk) y <= a & b  // and gate driving a D flip-flop
- FSMs (later described)

MHCAD. UNIT 2 16

Cuando se incluyen los cualificadores *posedge* o *negedge* en la lista de control de un *always* se sintetizan circuitos con *flip-flops* disparados por flanco.

El ejemplo más simple es la línea de código mostrada de la que se infiere un flip-flop D disparado por flanco ascendente, si la señal *Q_ff* es escalar, o un registro con carga en paralelo si está declarada con más de un bit.

Otros circuitos secuenciales síncronos más complejos utilizan también una construcción always que se activa con uno de los flancos de una señal y describen en su conjunto de sentencias asociado la evolución del estado de sus elementos de memoria.

Así, por ejemplo, la línea de código en rojo describe un contador binario ascendente. En cada flanco, el estado del contador cambia incrementándose en uno.

O por ejemplo la línea azul describe una puerta and cuya salida se conecta a un flip-flop.

De hecho, la manera de sintetizar lógica registrada, esto es, bloques combinacionales con registros en sus salidas, es substituir la lista de sensibilidad del always combinacional por la activación del always por flancos, tal como se muestra.

Por, último, una componente secuencial fundamental son las máquinas de estado finito, FSM, a las que dedicaremos una sección específica.

# Coding for Synthesis. Blocking and non-blocking assignments

- Non-blocking assignments capture better register transfer operations in digital synchronous systems. Use of non-blocking assignment for sequential circuits is recommended. (any case be aware of general recommendation on the topic)

  This example illustrates the differences

```
always @ (posedge clk)
  begin
     // shift register
     Z = Y; Y = X;
     y = x ; z = y; // parallel flip-flops
```

```
always @ (posedge clk)
  begin
     //both are shift registers
     Z <= Y; Y <= X;
     y <= x ; z <= y;
```

MHCAD. UNIT 2 17

Volvamos de nuevo a las diferencias entre los asignamientos blocking y no blocking.

Como ya dijimos, es fundamental conocer las implicaciones temporales de cada uno de ellos para utilizarlos adecuadamente.

Cuando escribimos para síntesis es inportante ser conscientes de que los asignamientos tipo non-blocking capturan mejor las operaciones de transferencias de registros de los sistemas digitales síncronos.

Cuando tenemos un conjunto de asignamientos non-blocking concurrentes, las expresiones o términos a la derecha del operador se evalúan antes de que se ejecute ningún asignamiento. En un sistema digital síncrono, con un flanco se almecena en cada registro lo que hay en sus entradas. La operación síncrona asume que estos valores, en ningún caso, están determinados por el contenido adquirido por los registros con ese mismo flanco. Este comportamiento se corresponde con el de los asignamientos non-blocking

Los siguientes ejemplos clarifican estas diferencias.

Al utilizar asignamientos tipo blocking, el orden en el que se escriban las sentencias de asignamiento influye en el comportamiento descrito. Por ejemplo, si queremos describir un registro de desplazamiento utilizando este tipo de operador, tenemos que ser muy cuidadosos con el orden en el que escribimos las asignaciones a las señales que representan cada una de las etapas.

Solo la primera opción en el código de la izquierda está describiendo un registro de desplazamiento. En la segunda se describen en realidad dos flip-flops (*z* e *y*) a cuya entrada de datos se conecta *x*.

A nivel de circuito, la actualización de cada etapa se produce con el flanco, y no tiene sentido hablar de un orden. De la misma forma, el orden en el que se escriban un conjunto de sentencias de asignamiento non-blocking concurrentes tampoco influye en los resultados. Las dos opciones para la ordenación de las sentencias de asignamiento describen registros de desplazamiento en el código de la derecha.

# Coding for Synthesis. Sequential resources *ii*

- An always without edges can also implied sequential logic (Example 8)
  - **always @(D or clk) if( clock ) Q_latch <= D; else Q_latch <= Q_latch;**
  - **always @(D or clk) if( clock ) Q_latch <= D;**
- Reset signals are important for sequential modules. Not all possible ways of modeling initialization are understood by synthesis tools. Use:

```
always @(posedge clk or negedge reset) /* asynchronous, low level
 active*/
 if (reset==0) Q<=0; else Q<=D;

always @(posedge clk) // synchronous, low level active
 if (reset==0) Q<=0; else Q<=D;
```

MHCAD. UNIT 2 18

Un *always* sin flancos en su lista de eventos también puede implicar lógica secuencial, como ya adelantábamos al analizar el modelado de comportamientos combinacionales.

Las señales de inicialización son fundamentales para los módulos secuenciales. Sin embargo, no todas las formas posibles de modelar una inicialización son entendidas por las herramientas de síntesis

Los códigos mostrados indican como modelar un reset, tanto síncrono como asíncrono, para síntesis.

Si el reset es asíncrono, la señal de reset debe aparecer en la lista de activación del *always*, pero cualificada con un flanco. El flanco es el negativo para señales de reset activas en bajo y el positivo para señales activas en alto. Si aparecen sin cualificar, al desactivarse se ejecutaría el código y el valor de *D* se asignaría a *Q*. Esto no se corresponde con el comportamiento de un flip-flop en el que al desactivarse la señal de reset, el estado almacenado se mantiene hasta el siguiente flanco de la señal de reloj.

Para modelar un reset síncrono, la única diferencia es que reset no aparece en la lista de activación. De esta forma, se captura que la inicialización se produce con los flancos de la señal de reloj.

# Coding for Synthesis. Sequential resources *iii*

- Flip-flops (previously described)
- Latches (previously described)
- Reset signals (previously described)
- Registered combinational logic (previously described)
- FSMs
  - Explicit FSMs (
    - Only one clock-synchronized event control expression
    - Represents the state of the machine by a declared register
    - Specify the sequence of states explicitly
  - Implicit FSMs (usually less code)
    - explicitly-declared state registers is not required
    - explicitly enumerated states are not required
    - Can use multiple event controls to implicitly describe an evolution of states
    - Synthesis tools require that the clocks must be all synchronized to the same clock edge

MHCAD. UNIT 2 19

Por último, para finalizar esta descripción sobre modelado de circuitos secuenciales y con ello esta presentación, abordamos la descripción de máquinas de estado finito.

Cuando se describe una FSM es posible hacerlo de forma explícita o implícita.

En una descripción explícita hay una declaración del registro de estado y una especificación de las transiciones de estado, ambas explícitas. En este tipo de codificación, el código describe explícitamente el diagrama de estados de la máquina.

Sin embargo, está no es la única forma en la que es posible modelar una FSM. De hecho, anteriormente en esta presentación, hemos utilizado como ejemplo un contador binario ascendente. Pero en dicha descripción, consistente en la sentencia always @ (posedge clk) Count = Count +1; no se enumeran explícitamente los estados, ni las transiciones entre ellos contenidas en su diagrama, a pesar de que el contador es una FSM y podríamos utilizar un diagrama o tabla de estados para representarlo. En este caso el estilo de descripción es implícito.

En general, a diferencia de las descripción explícitas, puede haber múltiples construcciones de control por eventos para describir de forma implícita la evolución de estados.

El estilo implícito suele requerir menos código. Esto es, se trata de descripciones más compactas, aunque no siempre es fácil capturar el comportamiento de una FSM de esta forma.

Utilizando el estilo explícito modelar una FSM a partir de su diagrama o tabla de estados es directo. A continuación mostramos distintas formas de hacerlo.

# CfS. Sequential resources. Explicit FSM

```
module FSM_estilo1(....);
input ...;
output ....;
parameter size = ...;
reg [size-1:0] state, next_state;
assign the_outputs = ...
// una funcion de las entradas y el estado
assign next_state = ...
// una funcion de las entradas y el estado
always @(negedge reset or posedge clk)
 if (reset == 1'b0) state <= start_state;
else state <= next_state;
endmodule
```

```
module FSM_estilo2(....);
input ...;
output ....;
parameter size = ...;
reg [size-1:0] state, next_state;
assign the_outputs = ...
// una funcion de las entradas y el estado
always @ (state or entradas)
begin
// decodifica next_state con case o if
end
always @(negedge reset or posedge clk)
 if (reset == 1'b0) state <= start_state;
else
state <= next_state;
endmodule
```

MHCAD. UNIT 2 20

En esta transparencia se muestran dos, podríamos llamar “plantillas”, para modelar máquinas de estado finito explícitamente.

En ambos casos, la descripción de la componente combinacional y de la parte de memoria es independiente.

Esta última se ha resuelto con un *always* síncrono, activado por el flanco activo del reloj o la activación de la señal de reset, en el que se modela la inicialización y la transferencia del próximo estado a estado presente.

La descripción de la componente combinacional requiere describir las salidas primarias y las funciones de próximo estado como funciones combinacionales de las entradas primarias y el estado. Puede hacerse utilizando distintos recursos. En el código de la izquierda que se ha denominado FSM_estilo 1, se utilizan asignamientos continuos. En el de la derecha, FSM_estilo2, se utiliza un always asíncrono (no hay cualificadores en la lista de eventos) para las funciones de próximo estado. Esto permite recurrir a sentencias de control de flujo tipo case, condicionales etc.. que simplifican la enumeración de las transiciones de estado.

Además de las opciones aquí mostradas, hay otras posibilidades: usar también un always asíncrono para las salidas, o incluso un único always para las salidas y las funciones de próximo estado.

Por supuesto la inicialización puede ser síncrona o no existir.

# CfS. Sequential resources. Explicit FSM. Example

```
// explícita
module speed_machine (clock, acelarador, freno, velocidad, dyn_ind)
input clock, acelerador, freno;
output [1:0] velocidad, dyn_ind;
reg [1:0] state, next_state;
parameter parado = 2'b00;
parameter V_baja = 2'b01, desacelarando = 0;
parameter V_media = 2'b10, velocidad_constante = 1;
parameter V_alta  = 2'b11, acelerando = 2;
assign velocidad = state;
assign dyn_ind = freno ? (state == parado) ? velocidad_constante : desacelerando:
acelerador ? (state == V_alta) ? velocidad_constante: acelerando: velocidad_constante;
always @(posedge clock) // síncrono gobierna las transiciones de estado
state <= next_state;
always @ (state or acelerador or freno) // asíncrono combinacional
if (freno == 1'b1) case (state)
parado: next_state <= parado;
V_baja: next_state <= parado;
V_media: next_state <= V_baja;
V_alta: next_state <= V_media;
default: parado;
endcase
else if (acelerador == 1'b1) case (state)
parado: next_state <= V_baja;
V_baja: next_state <= V_media;
V_media: next_state <= V_alta;
V_alta: next_state <= V_alta;
default: parado;
endcase
else next_state <= state;
endmodule
```

| freno/ acelerador | 1- | 00 | 01 |
|---|---|---|---|
| parado | parado, 1 | parado, 1 | V_low, 2 |
| V_baja | parado, 0 | V_baja, 1 | V_media, 2 |
| V_media | V_baja, 0 | V_media, 1 | V_alta, 2 |
| V_alta | V_media, 0 | V_alta, 1 | V_alta, 1 |

MHCAD. UNIT 2 21

El código mostrado describe la tabla de estados de la derecha. Se trata de una descripción explícita. Observamos:

Se enumeran los estados y se utilizan nombres simbólicos para ellos mediante la construcción *parameter.*

Un *always* síncrono gobierna las transiciones de estado

Dos asignamientos continuos modelan la salidas. Uno la salida *velocidad*, que se corresponde con el estado de la FSM y otro la salida *dyn_ind,* a cuyos posibles valores también se han asignado nombres simbólicos.

La función de próximo estado se describe con un always asíncrono. En él, una construcción condicional decodifica las entradas primarias y una construcción case, en dos de las tres ramas del condicional, el estado presente, de forma que se define el próximo estado en función del estado presente y las entradas primarias.

Observamos que el código es una traslación directa de la tabla de estados.

Este estilo de modelado es muy adecuado para controladores. En el caso de lógica estructurada, registros con distintas funcionalidades, contadores etc.., es posible utilizar un estilo más compacto, en el que capturamos su comportamiento sin pasar por un diagrama o tabla de estados como se ilustra en los siguientes ejemplos

# CfS. Sequential resources. Implicit FSM. Example

```
// implicit FSM no explicit state register
module contador_1 (clock, up_down, reset, count)
input clock, reset;
input [1:0] up_down;
output [2:0] count;
reg [2:0] count;
always @(negedge clock or negedge reset)
      if (reset == 0) count = 3'b0; else
            if (up_down == 2'b00 || up_down == 2'b11)
            count = count;
            else if (up_down == 2'b01) count = count + 1;
                  else if (up_down == 2'b10) count = count - 1;
endmodule
```

```
// implicit FSM. States are not explicitly enumerated
module contador_2 (clock, up_down, reset, count)
input clock, reset;
input [1:0] up_down;
output [2:0] count;
reg [2:0] count, next_count;
always @(negedge clock or negedge reset)
      if (reset == 0) count = 3'b0; else count = next_count;
always @ (count or up_down) begin
            if (up_down == 2'b00 || up_down == 2'b11)
            next_count = count;
            else if (up_down == 2'b01) next_count = count + 1;
                  else if (up_down == 2'b10) next_count = count - 1;
                        else next_count = count;
end
endmodule
```

Ambos códigos describen un contador binario, de tres etapas, con reset asíncrono activo en bajo y una señal de control de dos bits que permite seleccionar: inhibición, códigos 00 y 11, cuenta ascendente, código 01, o descendente, código 10.

En ninguno de los dos casos hay una enumeración explícita de los estados y por tanto tampoco del diagrama de estados. De hecho, los contadores aquí descritos tienen ocho estados, pero el cuerpo de la descripción no varía si estamos modelando contadores con un número distintos de etapas y por lo tanto de estados. La única difereencia estaría en la declaración de la señal *count*.

La diferencia entre ambos códigos es que, en el superior, hay un único *always*, esto es, no se separa la parte de memoria de la combinacional, lo que es muy frecuente en el modelado implícito, pero no en el explícito.