Bloque I. METODOLOGÍAS Y HERRAMIENTAS DE DISEÑO

TEMA 2 VERILOG

(2ª parte)

Organización Bloque I

Bloque I. Metodologías y Herramientas de diseño

- **Teoría**
  - Tema 1: Introducción.
  - Tema 2: Verilog
    - Revisión del lenguaje
    - Verilog para síntesis y simulación

      Recomendaciones/directrices de codificación:

      » Generales

      » Para describir **diseños que se van a sintetizar**

      » Para escribir **testbenches**

Recomendaciones generales

- Definir de forma precisa las especificaciones (del circuito, o del testbench) antes de escribir los códigos HDL
- Documentar el código usando comentarios: funcionalidad del módulo, uso de señales ….
- Escribir códigos estructurados (utilizando funciones, tareas, …)
- Utilizar nombres de señales que ayuden a su identificación
- Utilizar nombres simbólicos para representar valores de forma que se mejora la legibilidad y se facilitan las modificaciones

Verilog y la síntesis lógica

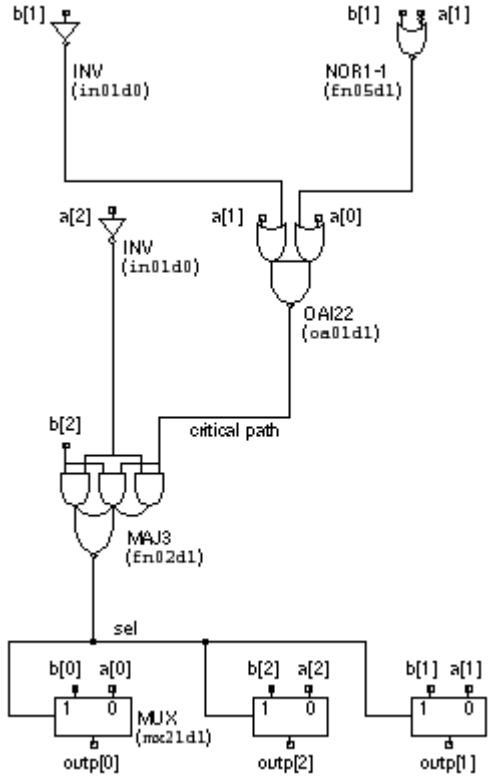
Síntesis RT (o lógica)

✓ Escribimos la funcionalidad, se genera una red lógica que la implementa

```
// la salida es la entrada menor
module comp_mux(a, b, outp);
input [2:0] a, b;
output [2:0] outp;
function [2:0] compara;
      input [2:0] ina, inb;
      begin
      if (ina <= inb) compara = ina;
      else compara = inb;
     end
 endfunction
assign outp = compara(a, b);
endmodule
```

Descripción comportamiento

Síntesis



Circuito sintetizado (los bloques utilizados pertenecen a una determinada librería de la tecnología) netlist

Verilog y la síntesis lógica

Ventajas de usar HDLs y herramientas de síntesis RTL

- Lleva mucho menos tiempo escribir una descripción comportamiento en un HDL y sintetizar automáticamente el circuito que desarrollar directamente la realización a nivel de puertas
- Las descripciones HDL son más sencillas de verificar, ya que hay menos detalles de los que ocuparse al intentar diagnosticar problemas en la funcionalidad de un sistema
- Las descripciones HDL son independientes de la tecnología
- Favorece la exploración arquitectural puesto que es fácil modificar o sustituir las descripciones HDL (probar varias soluciones RT distintas)

Verilog y la síntesis lógica. Directrices

- Cuando se utiliza un HDL como entrada de una herramienta de síntesis hay que adoptar un “estilo” que asegure que la descripción sea sintetizable y evite que el circuito sintetizado se comporte de manera no deseada:
  - ser conscientes de que no todos los algoritmos son sintetizables
  - evitar usar determinadas construcciones. No todas las construcciones están soportadas por las herramientas de síntesis lógica. Algunas no se sintetizan (mensaje de error) y otras son ignoradas (#, operador control de retrasos)
    - Por ejemplo, desde el punto de vista de la síntesis es muy distinto usar el operador multiplicación (*, sintetizable) que la división (/, no sintetizable en general)
  - observar reglas básicas que distinguen la lógica combinacional y la secuencial
  - no hacer asignamientos a la misma variable en más de un always
  - usar el operador de asignamiento non_blocking cuando se modela el comportamiento de registros hardware
  - ser consciente de que el estilo de descripción puede influir en el resultado de la síntesis (esto es, en cómo es el circuito resultante)

Verilog y la síntesis lógica. Recursos combinacionales

- ✓ Para sintetizar lógica combinacional se puede utilizar:
  - Netlist de primitivas verilog o de modulos combinacionales (ejemplo 2)
  - Asignamientos continuos (no procedurales)
  - Procedimientos *always @()*
    - *Incluir todas las entradas en la lista de o utilizar @* (lista de sensibilidad impícita, recomendado)*
    - *La lista de sensibilidad debe incluir únicamente señales sensibles a niveles*
    - Especificar la salida para todas las combinaciones de entrada. Si no se hace se generan diseños con latches no deseados:
      - Inicializar señales antes de bucles complejos
      - Evitar construcciones case incompletas (ejemplos 3 y 4)
      - Evitar construcciones incondicionales incompletas (ejemplo 5)
  - Funciones *y* Tareas
    - También debe especificarse la salida para todas las combinaciones de entrada

Verilog y la síntesis lógica. Recursos combinacionales

❑ Salidas completamente especificadas. Case incompleto

Ejemplo 3

```
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
```

Ejemplo 4

```
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;
default: alu_out <= 'bx; endcase
endmodule
```

- a: hay combinaciones de entrada para las que no está definida la salida. Se está describiendo un comportamiento secuencial y en síntesis se infieren latches
- Deben completarse las construcción case con un default
- b: no se infieren latches. Las combinaciones "incluidas" en el default se interpretan como inespecificaciones para la síntesis

Verilog y la síntesis lógica. Recursos secuenciales

- Un latch se describe con un condicional incompleto
  - **always @ * if(clock) Q_latch = D;**
- Cuando se incluyen los cualificadores posedge o negedge en la lista de sensibilidad de un always @ se sintetizan flip-flops (o registros paralelos si las señales Q_ff y D son de más de un bit)
  - **always @(posedge clk) Q_ff <= D;**

- Las señales de reset son importantes para los módulos secuenciales. No todas las formas de modelar inicializaciones son entendidas por las herramientas de síntesis. Utilizar:

always @(posedge clk or negedge reset) // asíncrono, activo en bajo
if (reset==0) Q<=0; **else** Q<=D; // equivalente if (!reset)

always @(posedge clk) // síncrono, activo en alto
if (reset==1) Q<=0; **else** Q<=D;

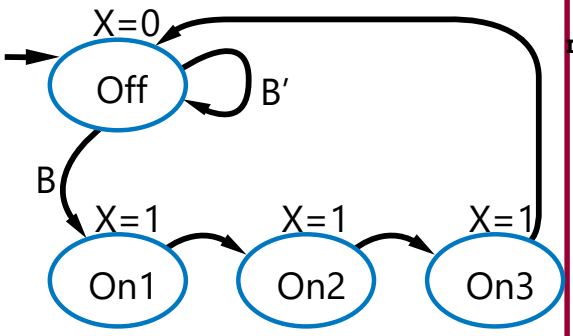
- Lógica combinacional con registros en sus salidas
  always @ (a or b) y = a & b // se sintetiza una puerta and
  always @ (posedge clk) y <= a & b // se sintetiza una and conectada a un flip-flop D

Notad el uso del operador non-blocking cuando modelamos registros hardware

Verilog y la síntesis lógica. FSMs. Modelado explícito

FSM con 2 procedimientos always

Inputs: B; Outputs: X

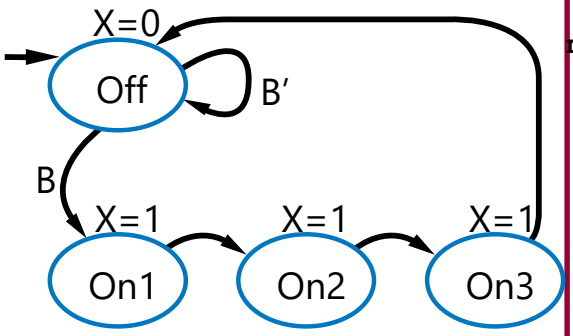



```
`timescale 1 ns/1 ns

module ejeplo8(B, X, Clk, Rst);

   input B;
   output reg X;
   input Clk, Rst;

   parameter S_Off = 0, S_On1 = 1,
             S_On2 = 2, S_On3 = 3;

   reg [1:0] State, StateNext;

   // CombLogic
   always @(State, B) begin
      case (State)
         S_Off: begin
            X = 0;
            if (B == 0)
               StateNext = S_Off;
            else
               StateNext = S_On1;
         end
         ...
         S_On1: begin
            X = 1;
            StateNext = S_On2;
         end
         S_On2: begin
            X = 1;
            StateNext = S_On3;
         end
         S_On3: begin
            X <= 1;
            StateNext = S_Off;
         end
          default: begin
            X = 0;
            StateNext = S_Off;
         end
      endcase
   end

   // StateReg
   always @(posedge Clk) begin
      if (Rst == 1 )
         State <= S_Off;
      else
         State <= StateNext;
   end   endmodule
```

Verilog y la síntesis lógica. FSMs. Modelado explícito

FSM sin declaración de StateNext

Inputs: B; Outputs: X



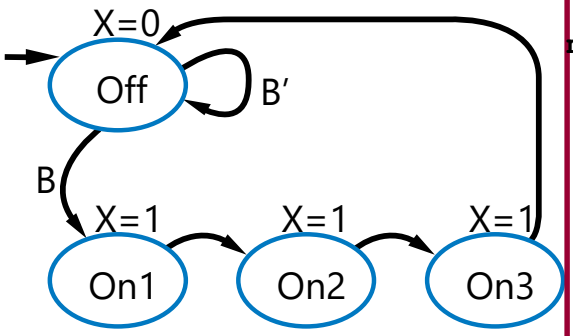

```
`timescale 1 ns/1 ns

module ejeplo9(B, X, Clk, Rst);

   input B;
   output X;
   input Clk, Rst;

   parameter S_Off = 0, S_On1 = 1,
             S_On2 = 2, S_On3 = 3;

   reg [1:0] State;

always @(posedge Clk) begin
      if (Rst == 1 )
         State <= S_Off;
      else
      case (State)
         S_Off: begin
            if (B == 0)
               State <= S_Off;
            else
               State <= S_On1;
         end
```

```
         ...
         S_On1: begin
            State <= S_On2;
         end
         S_On2: begin
            State <= S_On3;
         end
         S_On3: begin
            State <= S_Off;
         end
          default: begin
            State <= S_Off; end
      endcase
   end

assign X = (State != S_Off);
endmodule
```

Verilog y la síntesis lógica. FSMs. Modelado explícito

FSM modelada sólo con un always (incluida la salida)

Inputs: B; Outputs: X

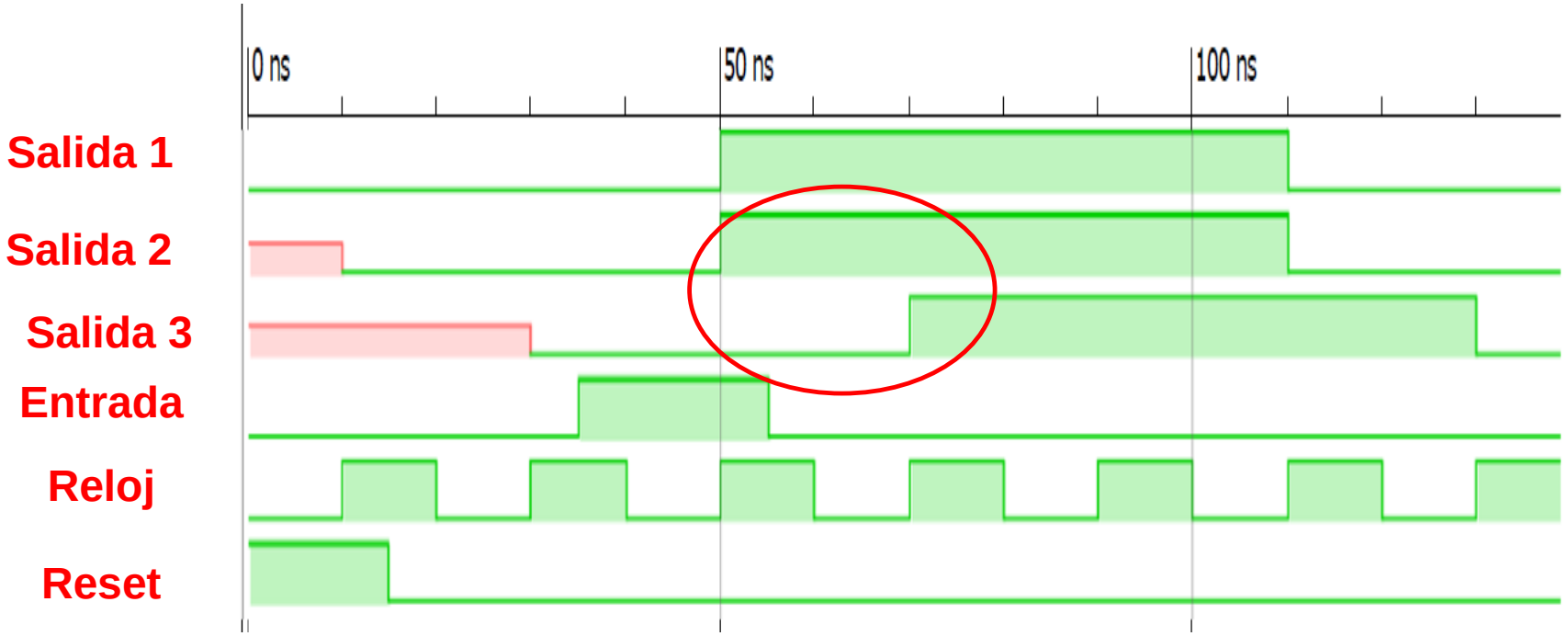



```
`timescale 1 ns/1 ns

module ejeplo10(B, X, Clk, Rst);

   input B;
   output reg X;
   input Clk, Rst;

   parameter S_Off = 0, S_On1 = 1,
             S_On2 = 2, S_On3 = 3;

   reg [1:0] State;

   always @(posedge Clk) begin
      if (Rst == 1 )
         State <= S_Off;
      else
      case (State)
         S_Off: begin
            X <= 0;
            if (B == 0)
               State <= S_Off;
            else
               State <= S_On1;
         end
```

```
         ...
         S_On1: begin
            X <= 1;
            State <= S_On2;
         end
         S_On2: begin
            X <= 1;
            State <= S_On3;
         end
         S_On3: begin
            X <= 1;
            State <= S_Off;
         end
          default: begin
            X <= 0;
            State <= S_Off;
         end
      endcase
   end

endmodule
```

Verilog y la síntesis lógica. FSMs. Modelado explícito

✓ Comparación de los tres estilos



La 3 sintetiza un flip-flop extra en las salidas. Esto explica el retraso

Verilog y la síntesis lógica. FSMs. Modelado implícito

- No requiere una enumeración explícita de los estados
- Se utiliza para componentes típicas de las unidades de datos como registros, y contadores

Ejemplo 10. Contador ascendente/descendente

```
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
```

Inhibición

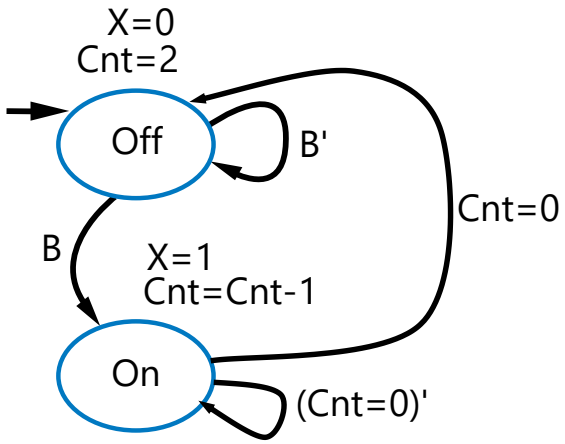
Cuenta ascendente

Cuenta descendente

Verilog y la síntesis lógica. HLSMs

- FSM modelo matemático a nivel lógico de hardware (circuitos secuenciales síncronos)
  - Diagrama de estados
  - Carta ASM
- HLSM (High Level State Machine) modelo matemático a nivel RT de hardware (sistemas digitales síncronos)
  - Diagrama de estados en el que aparecen operaciones de transferencias de registros
  - Carta ASM en la que aparecen operaciones de transferencias de registros

Inputs: B; Outputs: X; Register: Cnt(2)



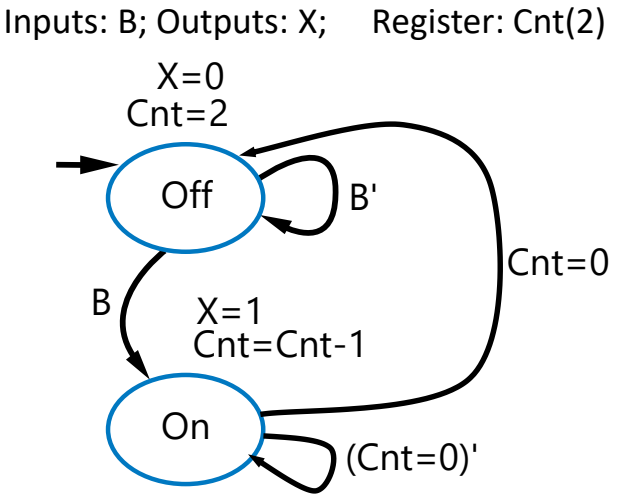
- Declaración de un registro de 2 bits Cnt
- Descripción de operaciones sobre estos registros (como “salidas” de la FSM)
- Estos registros aparecen en las condiciones asociadas a las transiciones

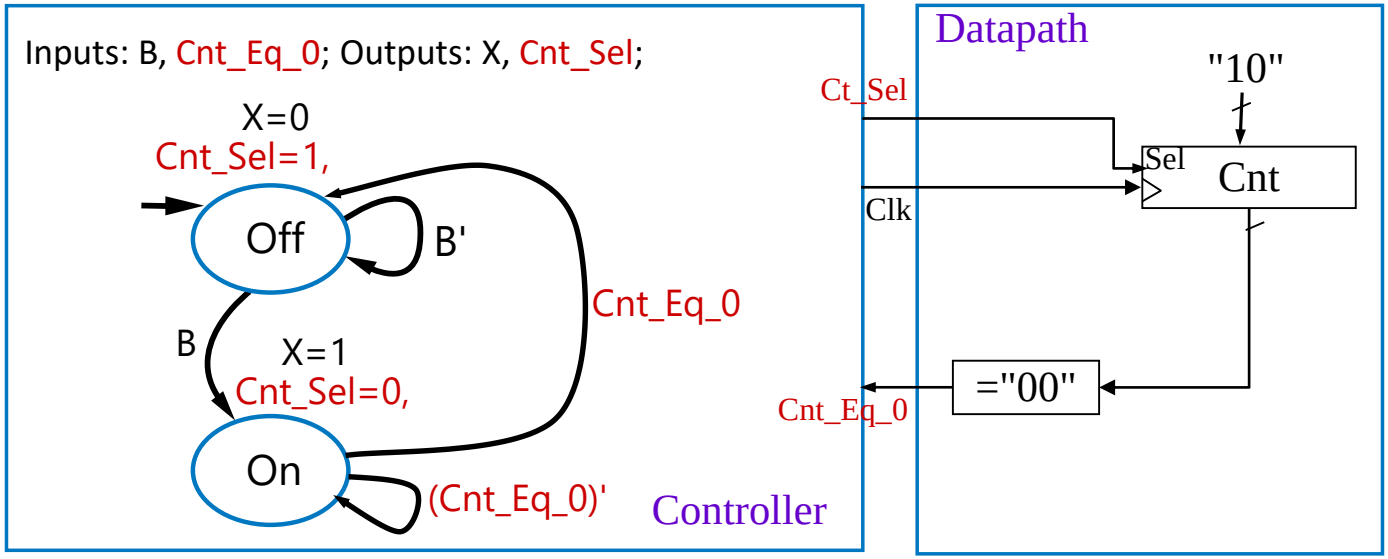
Verilog y la síntesis lógica. HLSMs

- Modelado jerárquico (varios modules)
  - Escribir descripciones comportamiento para cada componente de la unidad de datos (registros y unidades funcionales) y para el controlador
  - Escribir una descripción estructural del sistema instanciando y conectando los modules anteriores
  - Otra opción es escribir una descripción comportamiento para la unidad de datos, en lugar de para cada una de sus componentes
- Modelado no jerárquico (un único module)
  - Generalizando el modelado de FSMs
    - Distintas opciones como el caso de las FSMs
    - En las descripciones en las que no se declara explícitamente la lógica de próximo estado hay que ser muy cuidadoso con los asignamientos *blocking* y *non-blocking*

Verilog y la síntesis lógica. HLSMs. Jerárquico

- La implementación de esta HLSM requiere una unidad de datos y un controlador
- La unidad de datos está compuesta por un contador (operaciones de carga paralela, cuenta descendente) y un comparador
- La FSM que describe el controlador se deriva de la HLSM sustituyendo las operaciones de transferencia de registro por la activación de las señales de control correspondientes a dichas operaciones





Verilog y la síntesis lógica. HLSMs. Jerárquico

Ejemplo HLSM_1. modelado jerárquico

```
module HLSM1(input Rst, Clk, B, output X);
 wire [1:0] Cnt;
wire Cnt_Sel, Cnt_Eq_0;
Comparador C1( Cnt,  Cnt_Eq_0 );
control C2( Clk, B, Rst, Cnt_Eq_0,  X, Cnt_Sel);
contador C3( Clk, Cnt_Sel, Rst,  Cnt);
endmodule
```

```
module Comparador(input [1:0] Cnt, output Cnt_Eq_0 );
assign Cnt_Eq_0 = (Cnt==0)?1:0;
endmodule
```

```
module contador(input Clk, Cnt_Sel, Rst, output reg [1:0] Cnt);
always @ (posedge Clk) begin
                if (Rst == 1 )
      Cnt <= 0;
    else if (Cnt_Sel==1)
      Cnt <= 2;
    else
      Cnt <= Cnt - 1;
  end
endmodule
```

```
module, control(Clk, B, Rst, Cnt_Eq_0 ,X, Cnt_Sel);
input Clk, B, Rst, Cnt_Eq_0;
output X, Cnt_Sel;
reg  State;

parameter S_Off = 0,   S_On  = 1;
assign X = (State == S_On);
assign Cnt_Sel = (State == S_Off);

  always @(posedge Clk) begin
    if (Rst == 1 ) begin
      State <= S_Off;  end
    else begin
          case (State)
      S_Off: begin
        if (B == 0)
          State <= S_Off;
        else
          State <= S_On; end
      S_On: begin
        if (Cnt_Eq_0 == 1)
          State <= S_Off;
        else
          State <= S_On; end
      default:  State <= S_Off;
    endcase
    end
  end endmodule
```

Verilog y la síntesis lógica. HLSMs. No jerárquico

Ejemplo HLSM_2. Generalizando modelos FSMs

```
module HLSM2(B, X, Clk, Rst);
  input B;
  output  X;
  input Clk, Rst;

  parameter S_Off = 0,
            S_On  = 1;

  reg [0:0] State;
  reg [1:0] Cnt;

  always @(posedge Clk) begin
     if (Rst == 1 ) begin
        State <= S_Off;
        Cnt <= 0;
     end
     else begin
```

```
case (State)
            S_Off: begin
                  Cnt <= 2;
               if (B == 0)
                  State <= S_Off;
               else
                  State <= S_On;
            end
            S_On: begin
               Cnt <= Cnt - 1;
               if (Cnt == 0)
                  State <= S_Off;
               else
                  State <= S_On;
            end
          default: State <= S_Off;
endcase
      end
   end
assign X = (State != S_Off);
endmodule
```

Índice Tema 2.2.

- Recomendaciones generales
- Verilog y la síntesis lógica
- Verilog y la simulación (colección de ejemplos)
  - TB_1**. Revisión de concepto de testbench**
  - **TB_2. Generación de relojes**
  - **TB_3. Generación de señales con pocos cambios**
  - **TB_4. Generación con otras construcciones procedurales**
  - TB_5. Lectura desde array
  - **TB_6. Datos sincronizados con la señal de reloj**
  - TB_7. Lectura desde un fichero
  - **TB_8. Testbench autochequeante I**
  - TB_9. Testbench autochequeante II

Verilog y la Simulación. Testbench

HDL *para:*

- Instanciar el **modelo** del diseño que se quiere validar
- generar **estímulos** y aplicarlos al modelo
- generar la respuesta esperada (**vectores de referencia**) y compararlos con la salida del modelo (opcional)

```
`timescale 1ns/10ps
module testbenchexample();
reg in1, in2;
ejemploTB1 dut (in1, in2, sal); // se instancia
initial
  // aplico las cuatro combinaciones de entrada. Una cada 10ns
begin
 in1 = 0; in2 = 0;
#10 in1 = 1;
#10 in2 = 1;
#10 in1 = 0;
end
endmodule
```

Verilog y la Simulación. Testbench. Generación de estímulos

Ejemplo TB_2. Relojes

```
`timescale 1ns/100ps
module clocks();
// genera clock1 100MHz 50% duty cycle y clock2 50MHz 60% duty cycle
reg clock1, clock2;
real Clock1Period_ns;
parameter Clock1Freq_GHz = 0.1; // frecuencia de clock1 en GHz
initial
Clock1Period_ns = 1/Clock1Freq_GHz; // calcula el periodo de clock1
parameter Clock2Period_ns= 20; //define el periodo de clock2
initial clock1 = 0;
always # (Clock1Period_ns/2) clock1 = !clock1;
initial clock2 = 0;
always begin
# (Clock2Period_ns *0.60) clock2 = 0;
# (Clock2Period_ns *0.40) clock2 = 1;
end
endmodule
```

Verilog y la Simulación. Testbench. Generación de estímulos

Ejemplo TB_3. Señales con pocas transiciones

```
`timescale 1ns/100ps
module few_transition();
// Genera dos señales idénticas usando tiempos relativos (signal1) y
// absolutos (signal2)
reg signal1, signal2;
initial begin
signal1 = 0;
#20 signal1 = 1;
#20 signal1 = 0;
end
initial fork
signal2 = 0;
#20 signal2 = 1;
#40 signal2 = 0;
join
endmodule
```

signal1 y signal son idénticas

Verilog y la síntesis lógica. Material adiccional

Este Material profundiza en la codificación para síntesis.
Incluye los ejemplos que se referencian en ese apartado

Verilog y la síntesis lógica. Recursos combinacionales

Ejemplo 6. Modelado algorítmico

```
module align(data_in, data_out);
 input [7:0] data_in;
 output reg [7:0] data_out;
 integer j; reg flag;
always @ (data_in) // always @ *
begin
  data_out = data_in;
  flag = 0;
   for (j= 7; j >= 0; j = j - 1)
       if ((data_in[j] != 1)&& (flag == 0))
                data_out = data_out <<1;
       else flag = 1;
end
endmodule
```

Describe un bloque hardware que como salida (data_out) proporciona una versión de su entrada (data_in) en la que su primer "1" ocupa la posición más significativa y las posiciones vacantes se completan con ceros

La descripción es de tipo comportamiento. Se ha codificado un algoritmo que resuelve el problema Usando las construcciones propias de los lenguajes de programación de Verilog. Capturar esta Funcionalidad en una ecuación booleana no es directo

La descripción sólo captura funcionalidad. Si lo simulamos: en el mismo tiempo de simulación en el que data_in cambia su valor, se obtiene el data_out correspondiente

Si lo usamos como entrada a un sintetizador RTL obtenemos una red de puertas/bloques lógicos (netlist) que implementa un circuito combinacional con el comportamiento descrito (y cuyo retraso dependerá de la tecnología)

Verilog y la síntesis lógica. Recursos combinacionales

❑ Salidas completamente especificadas. Condicional incompleto

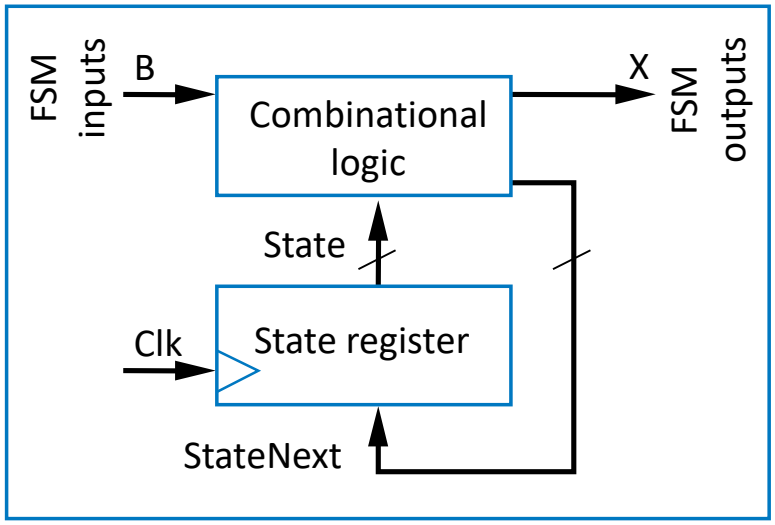
Ejemplo 5

```
module latch(D, clock, Q_latch);
input D, clock;
output Q_latch;
reg Q_latch
always @ (D or clock)
 if( clock ) Q_latch = D;
endmodule
```

No describe un circuito combinacional pues no está especificado ningún asignamiento a *Q_latch* para *clock* = 0. Por lo tanto conserva su valor incluso si D cambia.
Este código describe precisamente un elemento de memoria tipo latch

Verilog y la síntesis lógica. FSMs.

❑ Modelado explícito



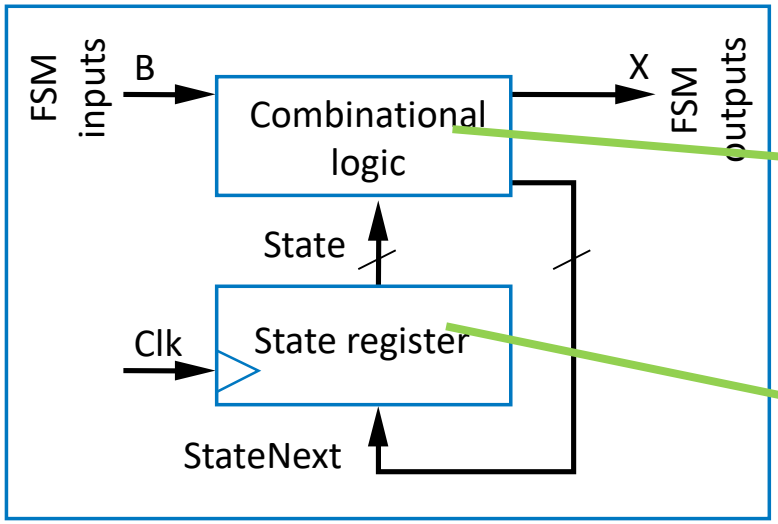
- Partimos de un diagrama de estados o de una tabla de estados
- Nos apoyamos en el modelo estructural de una FSM (combinacional + registro)
- Enumera los estados

❑ Modelado implícito

- No requiere una enumeración explícita de los estados
- Se utiliza para componentes típicas de las unidades de datos como registros, y contadores

Verilog y la síntesis lógica. FSMs. Modelado explícito

❑ Modelando separadamente la componente combinacional y los elementos de memoria




```
module FSM_1(B, X, Clk,);
   .....
   reg [1:0] State, StateNext;

   always @* begin
      ...describes X y StateNext
como función de state y B
   end

   always @(posedge Clk) begin
      describe el registro de estado
   end
endmodule
```

Ejemplo 7

❑ Sin declaración de StateNext

```
module FSM_1(B, X, Clk,)
   always @(posedge Clk) begin
     describe state como función de B y state
   end
endmodule
```

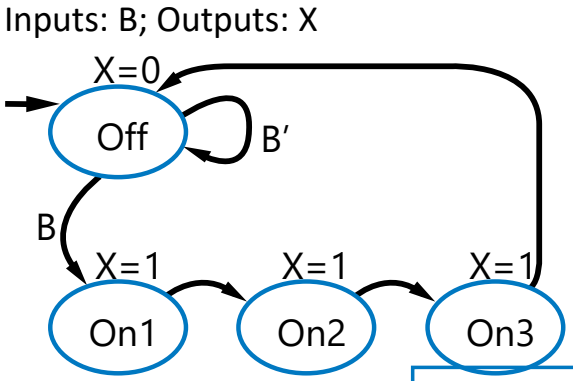
Logica combinacional con registro a la salida !

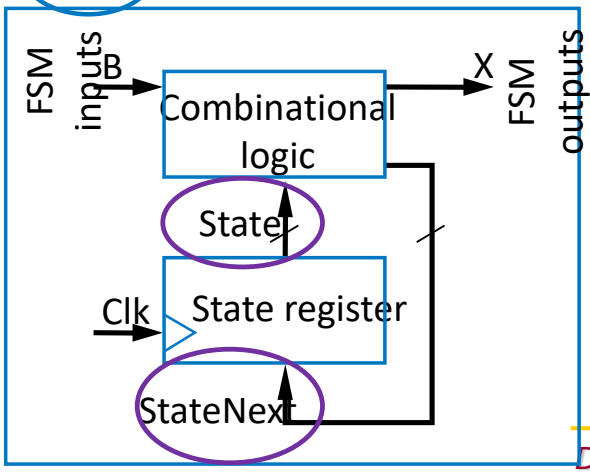
Ejemplo 8 y 9

Verilog y la síntesis lógica. FSMs. Modelado explícito

Ejemplo 7

- Declaración parameter
  - Nombres simbólicos para los estados
- Declara State y StateNext
  - Variables tipo reg de 2 bits



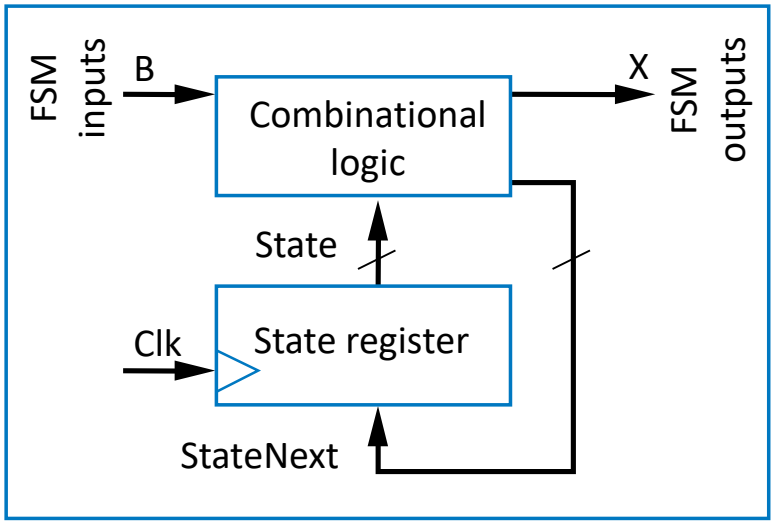



```
`timescale 1 ns/1 ns

module Ejemplo8(B, X, Clk, Rst);

   input B;
   output reg X;
   input Clk, Rst;

   parameter S_Off = 0, S_On1 = 1,
             S_On2 = 2, S_On3 = 3;

   reg [1:0] State, StateNext;

   // CombLogic
   always @(State, B) begin
      ...
   end

   // StateReg
   always @(posedge Clk) begin
      ...
   end
endmodule
```

Verilog y la síntesis lógica. FSMs. Modelado explícito

Ejemplo 7

- Dos procedimientos
  - uno para la C. combinacional
  - otro para el registro de estado

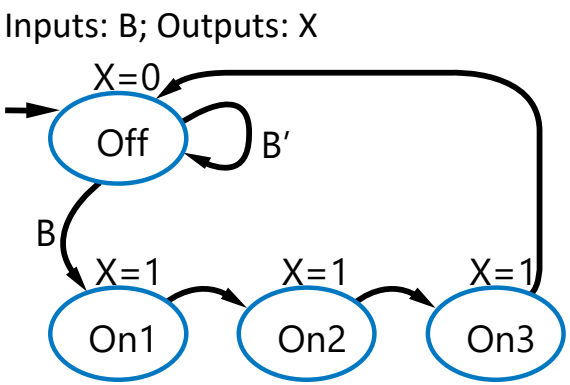



```
`timescale 1 ns/1 ns

module ejemplo8(B, X, Clk, Rst);

   input B;
   output reg X;
   input Clk, Rst;

   parameter S_Off = 0, S_On1 = 1,
             S_On2 = 2, S_On3 = 3;

   reg [1:0] State, StateNext;

   // CombLogic
   always @(State, B) begin
      ...
   end

   // StateReg
   always @(posedge Clk) begin
      ...
   end
endmodule
```

Verilog y la síntesis lógica. FSMs. Modelado explícito

Ejemplo 7

- Componente combinacional
  - Controlado por eventos en State y B (o @*)
  - Describe el próximo estado StateNext y la salida X
  - Usa case y condicionales if …else para describirlos
  - Usa default de la construcción case para que la FSM sea segura (desde cualquier estado no utilizado (ilegal, no alcanzable) hay una transición a uno legal)




```
always @(State, B) begin
      case (State)
         S_Off: begin
            X = 0;
            if (B == 0)
               StateNext = S_Off;
            else
               StateNext = S_On1;
         end
         S_On1: begin
            X = 1;
            StateNext = S_On2;
         end
         S_On2: begin
            X = 1;
            StateNext = S_On3;
         end
         S_On3: begin
            X = 1;
            StateNext = S_Off;
         end
        default: begin
            X = 0;
            StateNext = S_Off; end
      endcase end
```

Verilog y la síntesis lógica. FSMs. Modelado explícito

Ejemplo 7

- Registro de estado
  - Sincronizado con flanco ascendente Clk
  - Puesta a cero sincrona
  - Conversión StateNext - State

```
...
   parameter S_Off = 0, S_On1
= 1,
             S_On2 = 2, S_On3
= 3;

   reg [1:0] State,
StateNext;

...

   // StateReg
   always @(posedge Clk)
begin
      if (Rst == 1 )
         State <= S_Off;
      else
         State <= StateNext;
   end
...
```

Verilog y la simulación. Material adiccional

Más ejemplos de testbenches

Verilog y la Simulación. Testbench. Generación de estímulos

Ejemplo TB_4. Otras construcciones procedurales

```
// Genera una secuencia que sigue un código Gray para un bus de 16 bits
`timescale 1ns/100ps
module bit_gray();
reg [15:0] databus;
integer N;
initial begin
databus = 0;
for (N = 0; N < 65535; N = N + 1) begin
databus = (N ^ (N >> 1));
# 20;
end
end
endmodule
```

avance del tiempo, sino todo ocurre en tsim = 0!!

Verilog y la Simulación. Testbench. Generación de estímulos

Ejemplo TB_5. Lectura desde un array

```
`timescale 1ns/100ps
module array_test();
parameter test_number = 8;
parameter test_period = 10;
reg [15:0] entradas; // estímulos para dut
reg [15:0] arraytest [test_number -1:0];
integer N;
initial begin // se asignan los estímulos a un array
arraytest [0] = 37; arraytest [1] = 'b0001011100110110;
arraytest [2] = 45; arraytest [3] = 1037; arraytest [4] = 30765; arraytest [5] = 4001;
arraytest [6] = 102; arraytest [7] =52346; end
initial  begin // cada test_period se aplica un patrón del array al dut
for (N = 0; N < test_number; N = N + 1)
# test_period  entradas = arraytest[N];
end
……
endmodule
```

Verilog y la Simulación. Testbench. Generación de estímulos

Ejemplo TB_6. Datos sincronizados

```
// taken from Verilog Digital System Design by Z. Navabi
// sincroniza los vectores de test a un reloj
// si la frecuencia de la señal de reloj cambia, no es
// necesario cambiar la temporización de los patrones de test
`timescale 1ns/100ps
module testsynchronized ();
reg clock;
reg [3:0] in1;
initial repeat (13) #5 clock = ~clock;
initial
 clock = 0;
 always @ (posedge clock) #3 in1 = $random;
// retrasa el asignamientode in1 en 3 unidades de tiempo para garantizar
// que el cambio del reloj y de los datos no coincide
endmodule
```

Verilog y la Simulación. Testbench. Generación de estímulos

Ejemplo TB_7. Lectura desde un fichero

```
// Lee los estímulos de un fichero
// Estímulos en testdata.v_vec
// in1 in2 in3[1] in3[2]
//0010
//1110
//1100
//1010
//1111
`timescale 1ns/100ps
module tb_from_file();
reg in1, in2; reg [1:0] in3;
reg [3:0] array [0:4]; integer N;
parameter test_cycles =5;
initial begin
$readmemb("testdata.v_vec", array); // lee el contenido del fichero en array
for (N = 0; N < test_cycles; N = N + 1)
// cada 20 ns un vector de test de array se aplica al  dut
# 20 {in1, in2, in3} = array[N];  end   endmodule
```

Verilog y la Simulación. Testbench. Generación de estímulos

- Generándolos en el código HDL

- señales que cambian pocas veces (por ejemplo, resets)
  - construcción initial y bloque secuencial (begin .. end) o paralelo (fork ... join) y construcción #
- Relojes:
  - separadas del resto de estímulos
  - construcciones always o forever (en un initial) para el reloj

- Otros estímulos
  - usando construcciones procedurales (bucles etc..)
  - No olvidar el avance del tiempo

- Datos aleatorios usando $random (función del sistema)

- Leyéndolos de un array o de un fichero

Verilog y la Simulación. Testbench. Generación de referencias

- usando el propio HDL se calcula la respuesta esperada (Ejemplo TB_8)
- instanciando más de un modelo del diseño. Uno funciona como referencia y el otro es el que se valida. Por ejemplo, el modelo comportamiento como referencia del modelo post-síntesis (Ejemplo TB_9)

Verilog y la Simulación. Testbench. Generación de referencias

Ejemplo TB_8. Testbench autochequeante 1

```
`timescale 1ns/1ns
module testbenchAa();
reg [3:0] A, B;
wire [4:0] SUMAcir;
reg [4:0] SUMAcal;
integer i, j;

adder1 dut(A, B, SUMAcir);

initial
for (i = 0; i <= 15; i = i + 1)
begin
for (j = 0; j <= 15; j = j + 1)
begin
A = i; B = j;
SUMAcal = A + B; #5 if (SUMAcal != SUMAcir) $display ("error");
End end
endmodule
```

Verilog y la Simulación. Testbench. Generación de estímulos

Ejemplo TB_9. Testbench autochequeante 2

```
`timescale 1ns/1ns
module testbenchAb();
reg [3:0] A, B;
wire [4:0] SUMAcir, SUMAcomp;
integer i, j;

adder1 dut(A, B, SUMAcir);
adder1ref ref(A, B, SUMAcomp);

initial
for (i = 0; i <= 15; i = i + 1)
begin
for (j = 0; j <= 15; j = j + 1)
begin
A = i; B = j;
#5 if (SUMAcir != SUMAcomp) $display ("error");
end
end
endmodule
```