Verilog HDL

66.17 / 86.41 SISTEMAS DIGITALES

TA147 TALLER DE SISTEMAS DIGITALES

PEDRO IGNACIO MARTOS – PMARTOS@FI.UBA.AR

Temario

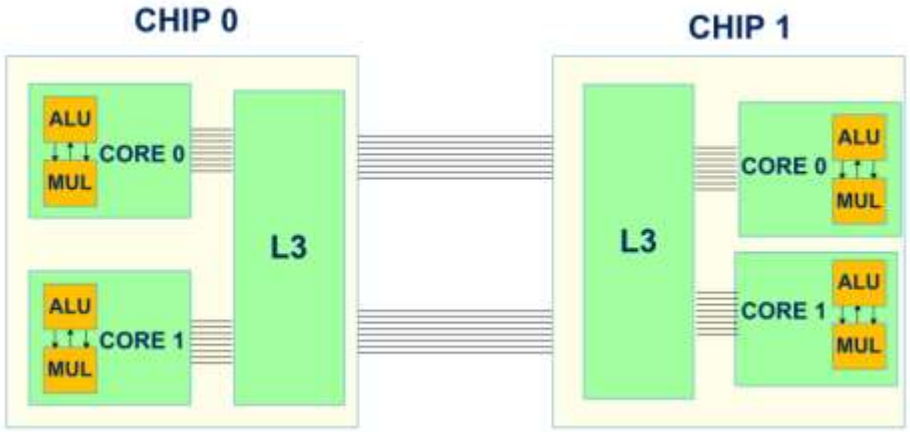
- **Introducción**
- Sintaxis
- Módulos
- Estructuras IF y CASE
- Circuitos Secuenciales
- Errores Comunes
- Bibliografía

Introducción

- Verilog, al igual que VHDL, es un lenguaje de descripción de hardware (HDL – Hardware Description Languaje).
- Verilog originariamente fue diseñado para verificar y simular diseños de hardware, con el surgimiento de los dispositivos FPGA, empezó a ser utilizado para implementar (sintetizar) circuitos digitales.
- No todos los elementos del lenguaje se pueden utilizar para generar descripciones que pueden sintetizarse, hay descripciones no sintetizables que se utilizan para simulación y verificación.
- Utilizamos el standard Verilog IEEE 1364-2001

Introducción – Estructura

- Un diseño en Verilog es una estructura jerárquica de Módulos interconectados entre sí.
- Cada Módulo define una interface con las señales que se utilizarán para interconectarse con otros Módulos
- Internamente un Módulo es un conjunto de Submódulos, flip-flops y lógica combinacional interconectados entre sí para implementar una función lógica



Introducción – Similitudes y Diferencias con un lenguaje de programación

Verilog	C/C++
Declaración de Variables	
`wire [7:0] sum;`	`int sum;`
Bloque Procedural	

`if (a == 8'b0) begin`
`…`
`end`
`else begin`
`…`
`end`
	`if (a == 0) {`
	`…`
	`}`
	`else {`
	`…`
	`}`

Introducción – Similitudes y Diferencias con un lenguaje de programación

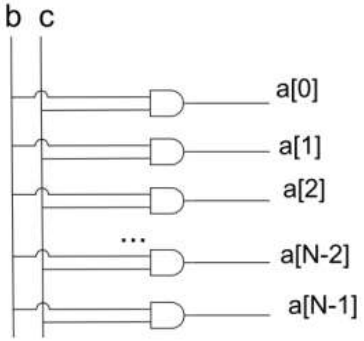
Verilog

Ciclo **For**

```
wire [N-1:0] a;
generate
  for (i=0; i<N; i=i+1)
begin
      assign a[i] = b & c;
  end
endgenerate
```

Describe

Hardware Concurrente



C/C++

```
for (i=0; i<N; i++) {
      a[i] = b + c;
}
```

```
a[0] = b+c;
a[1] = b+c;
a[2] = b+c;
...
a[N-2] = b+c;
a[N-1] = b+c;
```

Describe

Una secuencia de instrucciones

Temario

- Introducción
- **Sintaxis**
- Módulos
- Estructuras IF y CASE
- Circuitos Secuenciales
- Errores Comunes
- Bibliografía

Sintaxis – Elementos del Lenguaje

- Identificadores: Son nombres únicos para distintos elementos (señales, variables, módulos, etc.) Se componen de letras, números, el guion bajo (“_”) y el signo (“$”). Un identificador que comienza con “$” suele ser el nombre de una función de sistema, por ejemplo $Display es una función de sistema para generar salida en una consola
- Palabras Clave (“Keywords”): son palabras reservadas para elementos del lenguaje, por ejemplo “wire”, “Module”, etc.
- Espacio en blanco: incluye también tabulaciones y nueva línea. Se usan para separar identificadores y palabras clave. Se pueden utilizar libremente para facilitar la legibilidad del código.
- Comentarios: Hay de dos tipos: comentarios de una línea, que comienzan con “//” y se ignora todo lo que sigue hasta el fin de la línea; y comentarios multilínea, se ignora todo lo que hay entre “/*” y “*/”

Sintaxis – Estructura

- Ejemplo de un Módulo

```
1  module eq1( // declaracion de puertos de i/o
2      input wire i0, i1,
3      output wire eq
4      );
5      // declaracion de señales
6      wire p0, p1;
7      // Cuerpo del modulo
8      // suma de dos productos de terminos
9      assign eq = p0 | p1;
10     // producto de terminos
11     assign p0 = ~i0 & ~i1;
12     assign p1 = i0 & i1;
13 endmodule
```

Tipos de Datos

- Los tipos de datos pueden tener 4 valores:
  - '0': Valor lógico 0 (Falso)
  - '1': Valor lógico 1 (Verdadero).
  - 'X': Valor indefinido, para tipos de datos sin valor definido o cuando hay un conflicto de valores.
  - 'Z': Alta impedancia, el tipo de datos esta desconectada del resto del circuito, usada a la salida de buffers tri-state.
- La diferencia entre 'X' y 'Z' es que X representa a un tipo de dato que tiene un valor, pero que no puede determinarse, y 'Z' representa un tipo de datos que no tiene un valor asignado

Tipos de Datos – Dos tipos de datos básicos

- "Redes" (NET): representan conexiones físicas entre distintos bloques de hardware, se usan como salida de asignaciones *continuas* y como señales de conexión entre distintos módulos.
  - WIRE: (sintetizable) Es el tipo más común y representa una conexión física. Dentro de una función o bloque solo puede leerse. No se almacena su valor, por lo que debe ser la salida de un módulo o tener una asignación continua.
  - WAND y WOR: (no sintetizables) representan una conexión wired-and y wired-or respectivamente
  - TRI: (no sintetizable) representa una conexión three-state
  - SUPPLY0 y SUPPLY1: (no sintetizable) representan conexiones a masa o a la alimentación del circuito

Tipos de Datos – Dos tipos de datos básicos

- "Variables": representan el almacenamiento de un valor y son la salida de asignaciones *procedurales*.
  - REG: (sintetizable) Es el más común de este tipo y almacena su valor lógico entre asignaciones procedurales, solo se puede usar en funciones y bloques.
  - INTEGER: (no sintetizable) variable de propósito general para constantes o índices de ciclos
  - REAL: (no sintetizable) variable de propósito general que puede almacenar valores con decimales
  - TIME: (no sintetizable) variable para almacenamiento de tiempos durante una simulación (solo puede tener valores enteros)
  - REALTIME: (no sintetizable) variable para almacenamiento de tiempos durante una simulación (puede tener valores decimales)

Tipos de Datos – Representación de números

- La estructura de la representación de un número es

<signo> <Tamaño> ’ <Base> <Valor>

- Signo: (opcional) Para números signados, se indica si es positivo o negativo, si no está presente, el número es no signado.
- Tamaño: (opcional) Indica la cantidad de bits del número, si no está presente, se asume 32 bits de tamaño para almacenar el número, independientemente de la base. Si <Valor> tiene menos dígitos que <Tamaño> para números no signados se completa con ‘0’ y para números signados se extiende el bit de signo
- Base: indica la base del número, puede ser b/B (Binario), o/O (Octal), h/H (hexadecimal) o d/D (Decimal)
- Se puede utilizar “_” para clarificar el numero (Por ejemplo, agrupar de a 4 dígitos binarios)

Representación de números - Ejemplos

number	stored value	comment
5'b11010	11010	
5'b11_010	11010	_ ignored
5'o32	11010	
5'h1a	11010	
5'd26	11010	
5'b0	00000	0 extended
5'b1	00001	0 extended
5'bz	zzzzz	z extended
5'bx	xxxxx	x extended
5'bx01	xxx01	x extended
-5'b00001	11111	2's complement of 00001
'b11010	00000000000000000000000000011010	extended to 32 bits
'hee	00000000000000000000000011101110	extended to 32 bits
1	00000000000000000000000000000001	extended to 32 bits
-1	11111111111111111111111111111111	extended to 32 bits

Arreglos

- Wire s1, s2; // dos señales de un bit
- Wire [31:0] addr; // Arreglo de 1D de 32 bits. Si bien el orden puede ser ascendente [0:31] o descendente [31:0], en general se debe usar orden descendiente para que el número se lea de izquierda a derecha.
- Wire [31:0] mem1 [3:0]; // Arreglo para representar una memoria de 4 palabras de 32 bits. Ej. mem1 [2] = 32'hABCD
- Wire [31:0] mem2 [1:0][2:0]; // Arreglo para representar una matriz de 2 filas y 3 columnas de elementos de 32bits. Ej. mem2 [1] [1] = 32'hAAFF
- Seleccionar una parte: addr[7:0] hace referencia al Byte menos significativo de "addr"
- Concatenacion: {addr[31:24], addr[23:16], addr[15:8], addr[7:0]} representa a addr[31:0]

Operadores

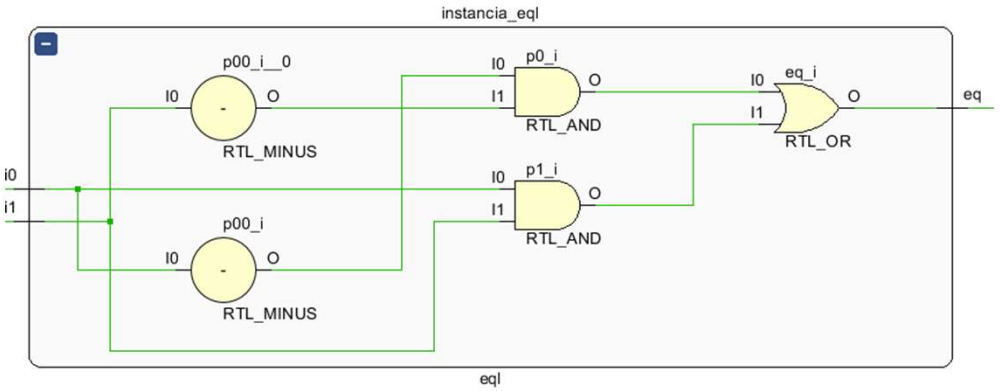
Operator Type	Symbols	Example
Bitwise	`~ & | ^`	`4'b1010 & 4'b0100`
Logical	`! && ||`	`4'b1010 && 4'b0100`
Reduction	`& ~& | ~| ^ ~^`	`|4'b0001`
Arithmetic	`+ - * / ** %`	`4'b1110 - 1`
Relational	`> < >= <= == != === !===`	`4'd5 < 4'd3`
Shift	`>> << >>> <<<`	`4'0110 << 1`
Concatenation	`{ , }`	`{2'b10, 2'b01}`
Replication	`{n{m}}`	`{8{1'b1}}`
Conditional	`? :`	`empty ? 1'b1 : 1'b0`

Temario

- Introducción
- Sintaxis
- **Módulos**
- Estructuras IF y CASE
- Circuitos Secuenciales
- Errores Comunes
- Bibliografía

Estructura de un Módulo

```
1  module eq1( // declaracion de puertos de i/o
2      input wire i0, i1,
3      output wire eq
4      );
5      // declaracion de señales
6      wire p0, p1;
7      // Cuerpo del modulo
8      // suma de dos productos de terminos
9      assign eq = p0 | p1;
10     // producto de terminos
11     assign p0 = -i0 & -i1;
12     assign p1 = i0 & i1;
13 endmodule
```



Estructura de un Módulo

- Declaración de puertos

```
1  module eql( // declaracion de puertos de i/o
2      input wire i0, i1,
3      output wire eq
4      );
```

- En general se declara el modo (entrada-in, salida-out, o entrada/salida-inout), el tipo de dato (si no se declara, se asume WIRE) y el nombre del puerto

```
module [module_name]
   (
    [mode] [data_type] [port_names],
    [mode] [data_type] [port_names],
    . . .
    [mode] [data_type] [port_names]
   );
```

Estructura de un Módulo

- Declaración de Señales: se declaran las señales que se van a usar dentro del módulo y que realizan las interconexiones internas

```
5       // declaracion de señales
6       wire p0, p1;
```

- La forma general de declarar señales es:

```
[data_type] [port_names];
```

- Es posible hacer una declaración implícita de las señales, pero no es recomendable porque puede dar lugar a errores.

Estructura de un Módulo

- Cuerpo: todas las partes se ejecutan en paralelo y en forma concurrente. Dentro del cuerpo puede haber una asignación continua, el instanciado de un submódulo, y/o un bloque “always”
- Asignación contínua:

```
9        assign eq = p0 | p1;
```

- Asignación continua en general: `assign [signal_name] = [expression];`
- En una asignación continua solo se pueden usar tipos de datos WIRE.
- La asignación continua siempre es concurrente, no secuencial

Instanciado de submódulos

- Ejemplo: generar un comparador de 2 bits a partir del comparador de 1 bit

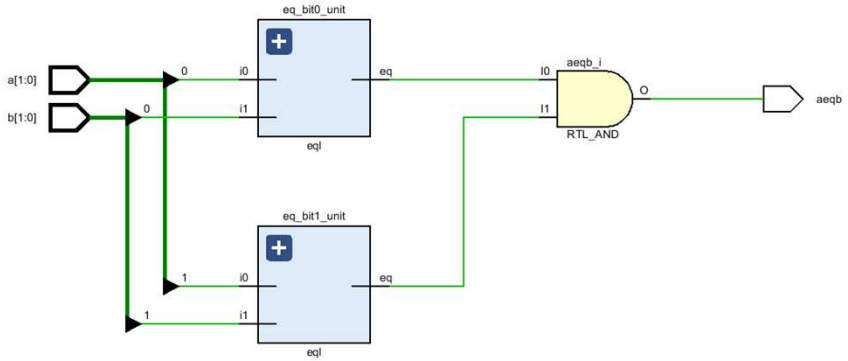
```
1  module eq2(
2      input [1:0] a, b,
3      output aeqb
4      );
5      wire e0, e1;
6
7      eql eq_bit0_unit (.i0(a[0]), .i1(b[0]), .eq(e0));
8
9      eql eq_bit1_unit (.i0(a[1]), .i1(b[1]), .eq(e1));
10     assign aeqb = e0 & e1;
11 endmodule
```

Modulo a Instanciar

Nombre de la Instancia

Nombre del parámetro del submodulo

Nombre de la señal del módulo actual



- La forma general de instanciar un submodulo es

<modulo a instanciar> <nombre de la instancia del submódulo> (.<nombre del parámetro del submódulo> (<nombre de la señal del módulo>, … , .<nombre del parámetro del submódulo> (<nombre de la señal del módulo>). Se puede asociar los parámetros del submodulo a señales del módulo por ubicación, pero no es recomendable

- Si el parámetro del submódulo es tipo “reg”, se le puede asignar una señal tipo “wire”. Esto se usa cuando la salida de un submódulo es tipo reg por tener elementos de memoria y hay que conectarla a una señal del módulo que está instanciándolo.

Constantes

- Las constantes se declaran mediante la palabra clave “localparam” (abreviatura de “local parameter”)
- Se declara la lista de constantes separada por ‘,’

```
localparam DATA_WIDTH = 8,
           DATA_RANGE = 2**DATA_WIDTH - 1;
```

```
localparam UART_PORT  = 4'b0001,
           LCD_PORT   = 4'b0010,
           MOUSE_PORT = 4'b0100;
```

Parámetros

- En algunos casos es necesario que las constantes definidas en un módulo puedan ser fijadas desde afuera del mismo.
- Para fijar las constantes desde afuera de un módulo se utilizan parámetros (“parameters”), que forman parte del encabezado del módulo y se declaran igual que la lista de puertos de I/O, pero comenzando con el símbolo “#”:

```
module [module_name]
   #(
    parameter [parameter_name]=[default_value],
              . . .
              [parameter_name]=[default_value];
   )
   (
    . . .  // I/O port declaration
   );
```

- Los parámetros pueden tener un valor por defecto, que se utilizan cuando al instanciar el módulo no se fija ningún valor para el parámetro

Parámetros – Ejemplo

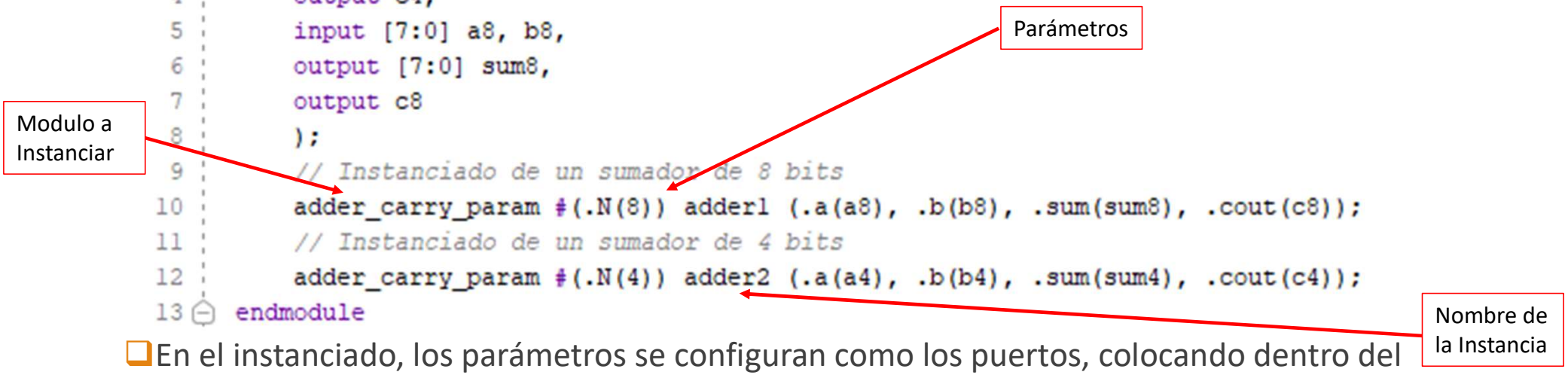
- Sumador definido con parámetros con un tamaño por defecto de 4 bits:

```
1  module adder_carry_param
2      #(parameter N=4) // sumador de 4 bits por defecto
3      (
4          input wire [N-1:0] a, b,
5          output wire [N-1:0] sum,
6          output wire cout
7      );
8      localparam N1 = N-1; // parametro local con la cantidad de bits menos uno para indice
9      wire [N:0] sum_ext; // señal interna extendida con un bit para carry
10     assign sum_ext = {1'b0, a} + {1'b0, b}; // realizo la suma extendida con 0 para guardar el carry
11     assign sum = sum_ext[N1:0]; // el resultado en la cantidad de bits del parametro
12     assign cout = sum_ext[N]; // el MSB es el carry
13 endmodule
```

Parámetros – Ejemplo

- Instanciado del sumador con parámetros para fijar su tamaño (8 y 4 bits):

```
1  module adder_insta (
2      input [3:0] a4, b4,
3      output [3:0] sum4,
4      output c4,
5      input [7:0] a8, b8,
6      output [7:0] sum8,
7      output c8
8      );
9      // Instanciado de un sumador de 8 bits
10     adder_carry_param #(.N(8)) adder1 (.a(a8), .b(b8), .sum(sum8), .cout(c8));
11     // Instanciado de un sumador de 4 bits
12     adder_carry_param #(.N(4)) adder2 (.a(a4), .b(b4), .sum(sum4), .cout(c4));
13 endmodule
```



- En el instanciado, los parámetros se configuran como los puertos, colocando dentro del paréntesis el valor del parámetro

Bloques @initial y @always

- Además de la asignación continua, se pueden implementar bloques procedurales, dentro de los cuales la asignación es secuencial.
- El uso de bloques procedurales está asociado a una descripción "de comportamiento", mientras que el uso de asignaciones continuas está asociado a descripciones a nivel compuertas lógicas.
- Hay dos tipos de bloques procedurales
  - @Initial: No sintetizable, se ejecuta una vez al inicio de simulaciones, solo puede haber un bloque @initial
  - @Always: Sintetizable, se ejecuta permanentemente, puede haber distintos bloques @always, todos ejecutándose concurrentemente.

Bloques procedurales

- El contenido de los bloques procedurales no tiene una relación directa con elementos de hardware y puede utilizarse tanto para circuitos secuenciales como combinacionales.
- Una mala codificación genera implementaciones innecesariamente complejas o no sintetizables.
- En circuitos combinacionales se utilizan
  - Asignaciones procedurales bloqueantes y no bloqueantes
  - Estructuras IF
  - Estructuras CASE

Bloque @always - Estructura

- Los bloques con una “lista de sensibilidad” o “expresión de control de evento” tienen la siguiente estructura:

```
always @([sensitivity_list])
begin [optional name]
   [optional local variable declaration];

   [procedural statement];
   [procedural statement];
  . . .
end
```

Bloque @always - Estructura

- La lista de sensibilidad “[sensitivity_list]” es una lista de las señales para las cuales el bloque @always debe ser evaluado, representa las señales para las cuales el bloque “es sensible”
- Para un circuito combinacional, es la lista de señales de entrada del circuito
- Un bloque @always puede considerarse un circuito digital que puede estar suspendido o activado, cuando una señal en la lista de sensibilidad cambia, el circuito se activa y se ejecuta secuencialmente. Al llegar al final del bloque, el circuito vuelve a estar suspendido
- El bloque @always se ejecuta en un ciclo infinito cuyo inicio está controlado por un cambio en una señal de la lista de sensibilidad

Asignaciones procedurales

- Las asignaciones procedurales pueden ser bloqueantes o no bloqueantes:

```
[variable_name] = [expression];    // blocking assignment
[variable_name] <= [expression];   // nonblocking assignment
```

- En las asignaciones bloqueantes la expresión se evalúa y se asigna inmediatamente, antes de ejecutar la siguiente sentencia del bloque. Durante la asignación se bloquean otras asignaciones, de forma similar a la asignación de variables en “C”
- En las asignaciones no bloqueantes, la expresión se evalúa, pero se asigna recién al finalizar la ejecución del bloque, es decir no se bloquean las demás asignaciones, sino que se realizan todas en forma concurrente. En caso de que el resultado de una asignación sea necesario, se utiliza el valor anterior de la variable.
- Las asignaciones bloqueantes se utilizan para circuitos combinacionales
- Las asignaciones no bloqueantes se utilizan para circuitos secuenciales, ya que se infiere un elemento de memoria para “recordar” el valor anterior de la variable.

Ejemplo

```
1  module eq1_always(
2      input wire i0, i1,
3      output reg eq // eq se declara como registro
4      );
5      // p0 y p1 tambien se declaran como reg
6      reg p0, p1;
7
8      always @(i0, i1) // i0 y i1 estan en la listga de sensibilidad
9      begin
10         // el orden es importante porque la asignacion es bloqueante
11         // por ser asignacion bloqueante se usa reg en vez de wire
12         p0 = ~i0 & ~i1;
13         p1 = i0 & i1;
14         eq = p0 | p1;
15     end
16 endmodule
```

Ejemplo

- Las señales eq, p0 y p1 se asignan dentro del bloque, por eso son del tipo REG.
- La lista de sensibilidad son las señales de entrada: i0 y i1.
- La ejecución es secuencial, por eso es importante el orden de las sentencias.
- Como es un circuito combinacional, las asignaciones son bloqueantes (“=“).
- Si no se incluyen todas las señales de entrada en la lista de sensibilidad, puede haber diferencias entre la simulación (que responde a la lista de sensibilidad) y el circuito sintetizado (que responde a todas las señales). Por eso las herramientas suelen generar advertencias cuando no todas las señales de entrada están en la lista de sensibilidad.

Temario

- Introducción
- Sintaxis
- Módulos
- **Estructuras IF y CASE**
- Circuitos Secuenciales
- Errores Comunes
- Bibliografía

Estructura IF

- La sintaxis de la estructura IF es:

```
if [boolean_expr]
   begin
      [procedural statement];
      [procedural statement];
      . . .
   end
else
   begin
      [procedural statement];
      [procedural statement];
      . . .
   end
```

- Se sintetiza como un multiplexor de dos entradas cuya salida es controlada por la expresión booleana y sus entradas son los bloques procedurales

Estructura IF

- La expresión booleana se evalúa primero, si su valor es “true” se ejecutan las sentencias que siguen a continuación, si su valor es “false” se ejecutan las sentencias a continuación de ELSE.
- La parte de ELSE es opcional, pero para generar circuitos combinacionales debe estar presente.
- Las estructuras IF pueden anidarse en cascada para establecer prioridades en las expresiones booleanas:

```
if [boolean_expr_1]
   . . .
else if [boolean_expr_2]
   . . .
else if [boolean_expr_3]
   . . .
else
   . . .
```

- Estas expresiones infieren multiplexores en cascada para el ruteo de la señal de salida

Ejemplo: Priority Encoder

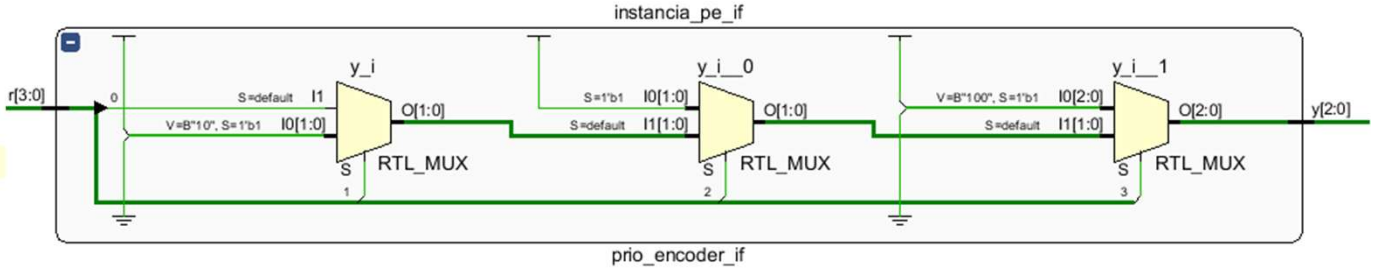
- Un circuito “Priority Encoder” genera una salida en función de cuál de sus entradas esta activa, con una prioridad, es decir, algunas entradas tienen más prioridad para generar la salida.
- La tabla de verdad del circuito es la siguiente:

input r	**output** pcode
1 – – –	100
0 1 – –	011
0 0 1 –	010
0 0 0 1	001
0 0 0 0	000

Ejemplo: Priority Encoder

- La descripción como módulo es la siguiente:

```
1  module prio_encoder_if(
2      input wire [3:0] r,
3      output reg [2:0] y
4      );
5      always @*
6          if (r[3] == 1'b1)
7              y = 3'b100;
8          else
9              if (r[2] == 1'b1)
10                 y =  3'b011;
11             else
12                 if (r[1] == 1'b1)
13                     y =  3'b010;
14                 else
15                     if (r[0] == 1'b1)
16                         y =  3'b001;
17                     else
18                         y =  3'b000;
19 endmodule
```



- Always @* es una forma resumida para indicar que la lista de sensibilidad debe incluir todas las señales de entrada (señales a la derecha de una asignación y señales que forman parte de una condición boolena

Estructura CASE

- Es una estructura de decisión múltiple que evalúa numéricamente [case_expr] y la compara con los valores [item] y se evalúa el bloque para el cual su valor de [item] coincide con el valor de [case_expr].
- Si varios [item] coinciden con [case_expr], se evalúa el primer bloque para el que coinciden
- Opcionalmente el ultimo [item] puede ser **default**, que es el bloque que se evalúa cuando ningún [item] coincide con [case_expr]
- Si cada bloque tiene una sola línea procedural, se puede no utilizar los delimitadores begin/end
- Se sintetiza como un multiplexor con tantas entradas como bloques [item] haya, y la salida es controlado por [case_expr]

```
case [case_expr]
   [item]:
      begin
         [procedural statement];
         [procedural statement];
         . . .
      end
   [item]:
      begin
         [procedural statement];
         [procedural statement];
         . . .
      end
   [item]:
      begin
         [procedural statement];
         [procedural statement];
         . . .
      end
   . . .
   default:
      begin
         [procedural statement];
         [procedural statement];
         . . .
      end
   endcase
```

Ejemplo: Priority Encoder

input r	output pcode
1 – – –	100
0 1 – –	011
0 0 1 –	010
0 0 0 1	001
0 0 0 0	000

```
1  module prio_encoder_case(
2      input wire [3:0] r,
3      output reg [2:0] y
4      );
5      always @*
6          case (r)
7              4'b1000, 4'b1001, 4'b1010, 4'b1011,
8              4'b1100, 4'b1101, 4'b1110, 4'b1111:
9                  y = 3'b100; // r = 1XXX
10             4'b0100, 4'b0101, 4'b0110, 4'b0111:
11                 y = 3'b011; // r = 01XX
12             4'b0010, 4'b0011:
13                 y = 3'b010; // r = 001X
14             4'b0001:
15                 y = 3'b001; // r = 0001
16             default: // tambien podria usarse 4'b0000
17                 y = 3'b000; // r = 0000
18         endcase
19 endmodule
```

CASEZ y CASEX

- También se pueden usar los valores X y Z en estructuras CASE.
- En CASEZ los valores Z y los caracteres '?' se tratan como bits "don't care", es decir, no se evalúa si coinciden.
- En CASEX los valores X, Z y los caracteres '?' se tratan como bits "don't care", es decir, no se evalúa si coinciden
- Como en las simulaciones pueden aparecer valores X o Z, se prefiere el caracter '?'

Ejemplo: Priority Encoder

```
1  module prio_encoder_casez(
2      input wire [3:0] r,
3      output reg [2:0] y
4      );
5      always @*
6          casez (r)
7              4'b1???:
8                  y = 3'b100; // r = 1XXX
9              4'b01??:
10                 y = 3'b011; // r = 01XX
11             4'b001?:
12                 y = 3'b010; // r = 001X
13             4'b0001:
14                 y = 3'b001; // r = 0001
15             4'b0000: // tambien podria usarse default
16                 y = 3'b000; // r = 0000
17         endcase
18 endmodule
```

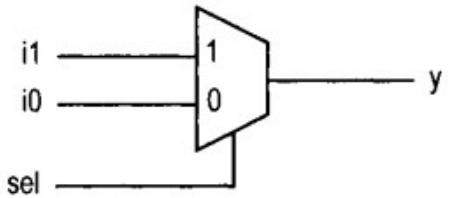
“Full” CASE y “Parallel” CASE

- Cuando en una estructura CASE los bloques [item] contemplan todos los casos posibles de [case_expr], se trata de un “Full” CASE y se corresponde a un circuito combinacional, ya que tengo definida la salida para cada posible entrada.
- Cuando esto no sucede, la salida debe preservar el valor anterior para las entradas no contempladas, por lo que se infiere un registro y el circuito pasa a ser secuencial.
- Se puede utilizar “Default” para que el CASE sea combinacional
- Cuando en una estructura CASE los bloques [item] son mutuamente excluyentes, es decir, cada [item] corresponde a un único valor, se trata de un “Parallel” CASE
- Un Parallel CASE infiere una estructura multiplexada (multiplexed routing network). Cuando esto no sucede, se infiere una estructura en cascada (priority routing network)
- Se pueden utilizar Atributos en el código HDL o directivas en las herramientas para forzar la implementación de los CASE como Full y/o Parallel CASE, pero esto genera diferencias entre la simulación funcional y la simulación de implementación, por lo que no es recomendable su uso

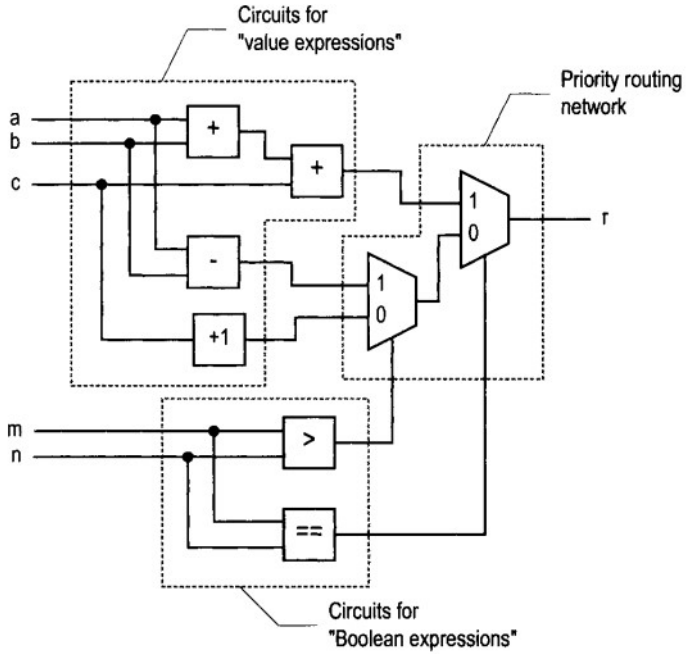
Estructura en cascada “Priority Routing Network”

- Esta estructura se implementa como una secuencia de multiplexores 2 a 1
- Una expresión IF-ELSE implementa este tipo de estructura

```
if (m==n)
   r = a + b + c;
else if (m > n)
   r = a - b;
else
   r = c + 1;
```



sel	y
0 (false)	i0
1 (true)	i1



Estructura en cascada “Priority Routing Network”

- Una expresión CASE que no sea “Full” también implementa este tipo de estructura porque se busca la primera equivalencia entre [case_expr] e [item]:

```
reg [2:0] s
. . .
casez (s)
   3'b111: y = 1'b1;
   3'b1??: y = 1'b0;
   3'b000: y = 1'b1;
endcase
```

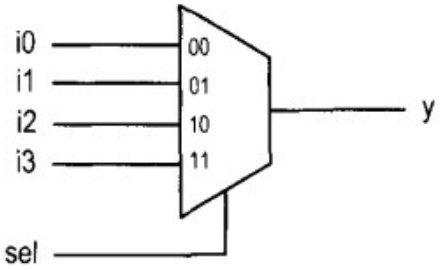
- En este caso, el valor 111 coincide en 2 ítems, por lo que se traslada a una estructura if-else como la siguiente:

```
if [expr==item1]
   statement1;
else if [expr==item2]
   statement2;
else if [expr==item3]
   statement3;
else
   statement4;
```

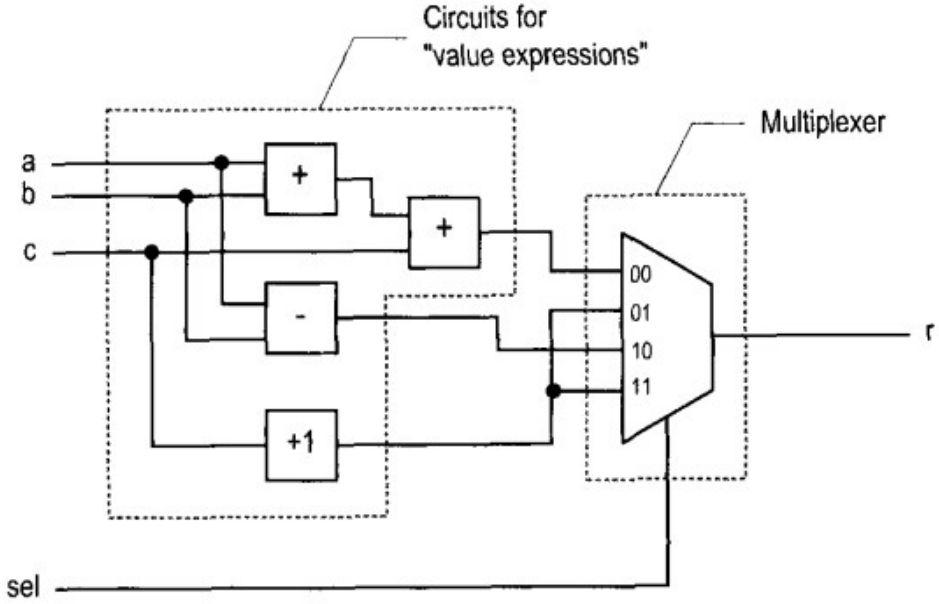
Estructura multiplexada “Multiplexed Routing Network”

- Esta estructura se implementa con un multiplexor n a 1
- Una expresión Parallel CASE implementa este tipo de estructura:

```
wire [1:0] sel;
. . .
case (sel)
   2'b00: r = a + b + c;
   2'b10: r = a - b;
   default: r = c + 1; // 2'b01, 2'b11
endcase
```



sel	y
00	i0
01	i1
10	i2
11	i3



Codificación de bloques Always

- Al escribir descripciones, se debe tener en cuenta como las distintas estructuras del lenguaje se implementan en hardware.
- Es importante recordar que la finalidad del código HDL es describir hardware, no describir algoritmos en forma secuencial como si fueran sentencias de un lenguaje de programación.
- Describir hardware en forma algorítmica puede producir código no sintetizable, implementaciones excesivamente complejas, o diferencias entre la simulación funcional y la simulación de implementación.
- Al describir circuitos combinacionales, los errores más comunes son: Asignar una misma variable en múltiples bloques Always; Que la lista de sensibilidad el bloque este incompleta; Que no estén contempladas todas las posibilidades; o que no se asigne la salida en todos los casos

Guía para describir circuitos combinacionales

- Asignar una variable en un solo bloque always.
- Usar asignaciones bloqueantes (=) en vez de no bloqueantes (<=).
- Usar always @* para incluir todas las entradas.
- Implementar todas las bifurcaciones de las estructuras IF y CASE.
- Asignar todas las salidas en todas las bifurcaciones.
- Si no son necesarias todas las bifurcaciones, asignar valores por defecto de las salidas al principio del bloque always.
- Generar las estructuras Full CASE y Parallel CASE en código, no mediante atributos
- Tener en cuenta los tipos de estructura que implementan las construcciones IF y CASE (priority o multiplexed routing network).
- Pensar los bloques always como descripciones de hardware, no como algoritmos .

Temario

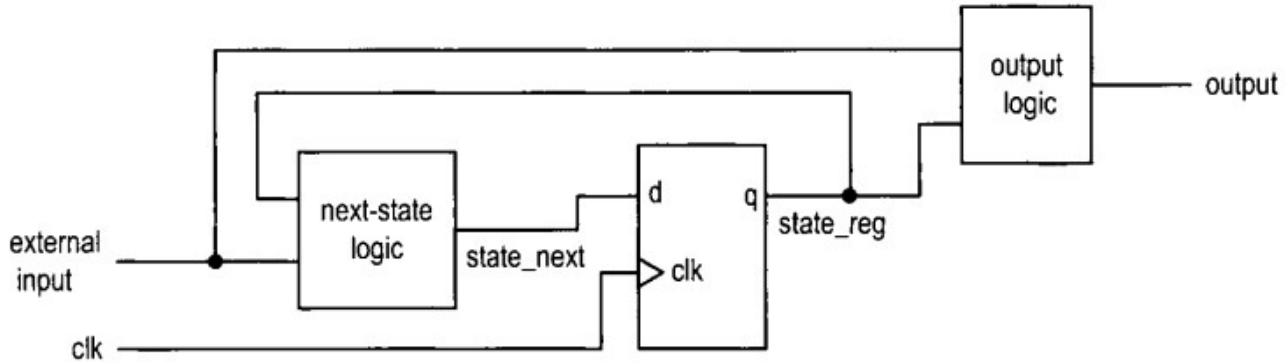
- Introducción
- Sintaxis
- Módulos
- Estructuras IF y CASE
- **Circuitos Secuenciales**
- Errores Comunes
- Bibliografía

Circuitos Secuenciales

- Un circuito secuencial contiene un elemento de memoria cuya función es almacenar el "estado interno" del circuito.
- La entrada afecta a la salida y/o al estado interno.
- La salida es función de la entrada y del estado interno.
- En los circuitos secuenciales *sincrónicos* todos los elementos de memoria están controlados por una misma señal de temporización (se dice que están "sincronizados").
- Los cambios en la entrada durante el estado alto o bajo de la señal de temporización no afectan al circuito.
- El circuito cambia su estado interno y su salida de acuerdo al valor presente en su entrada en el flanco ascendente o descendente de la señal de temporización

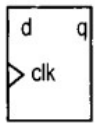
Elementos de un circuito secuencial sincrónico

- Registro de estado (State Register): grupo de flip flops con una señal de temporización común (clk).
- Lógica de siguiente estado (Next state logic): circuito combinacional que usa la entrada (externa input) y el valor actual del registro de estado para generar el nuevo valor del registro de estado.
- Lógica de salida (Output logic): circuito combinacional que usa la entrada y el valor actual del registro de estado para generar el nuevo valor de la salida (output).



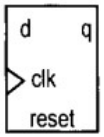
Flip Flop tipo D

- El elemento de memoria de los circuitos secuenciales sincrónicos es el flip flop tipo “D”
- El valor de la entrada “d” es muestreado en el flanco de la señal de temporización “clk” y almacenado en el flip flop.
- En el siguiente flanco de clk el valor almacenado aparece en la salida “Q”
- Un flip flop almacena 1 bit. Un grupo de flip flops almacenan múltiples bits y forman un “registro”
- Opcionalmente puede haber una señal asincrónica de inicialización “reset” que fija la salida “q” en “0” y una señal de habilitación “enable” (en) que permite la actualización de la salida “q” en el flanco de clk.



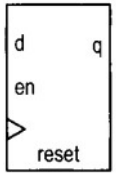
clk	q*
0	q
1	q
↓	d

(a) D FF



reset	clk	q*
1	-	0
0	0	q
0	1	q
0	↓	d

(b) D FF with asynchronous reset

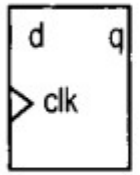


reset	clk	en	q*
1	-	-	0
0	0	-	q
0	1	-	q
0	↓	0	q
0	↓	1	d

(c) D FF with synchronous enable

Codificación Flip Flop tipo D sin señal asincrónica de inicialización

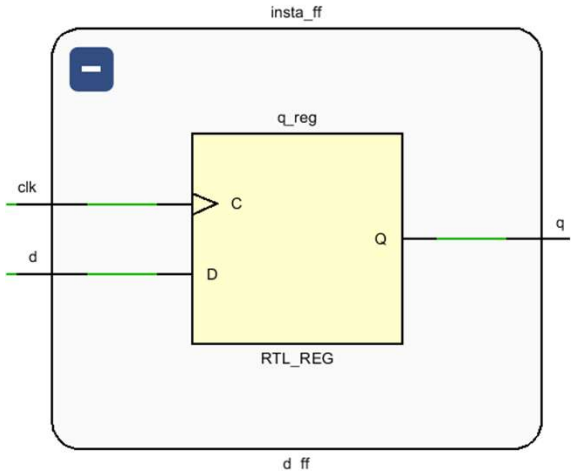
- La salida se actualiza en el flanco positivo (ascendente) de clk, por eso en la lista de sensibilidad se utiliza “posedge” (de “positive Edge”).
- Los cambios solo suceden en el flanco ascendente de clk, por eso “d” no forma parte de la lista de sensibilidad del bloque always



clk	q*
0	q
1	q
↓	d

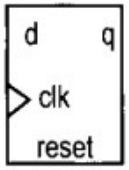
Codificación Flip Flop tipo D sin señal asincrónica de inicialización

```
1  module d_ff(
2       input clk,
3       input d,
4       output reg q
5       );
6       always @(posedge clk)
7           q <= d;
8  endmodule
```



Codificación Flip flop tipo D con señal asincrónica de inicialización

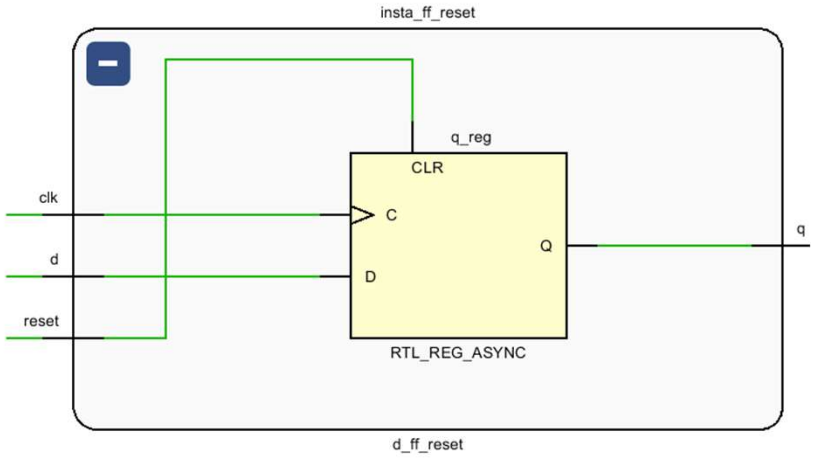
- La salida se actualiza solo en el flanco positivo (ascendente) de clk
- Si la seña de reset está en estado activo, la salida queda en cero



reset	clk	q*
1	-	0
0	0	q
0	1	q
0	↓	d

Codificación Flip flop tipo D con señal asincrónica de inicialización

```
1  module d_ff_reset(
2      input clk,
3      input reset,
4      input d,
5      output reg q
6      );
7      //  No es posible mezclar señales activadas por flanco
8      //  y por nivel al mismo tiempo en la lista de sensibilidad
9      always @(posedge clk, posedge reset)
10         if (reset)
11             q <= 1'b0;
12         else
13             q <= d;
14 endmodule
```



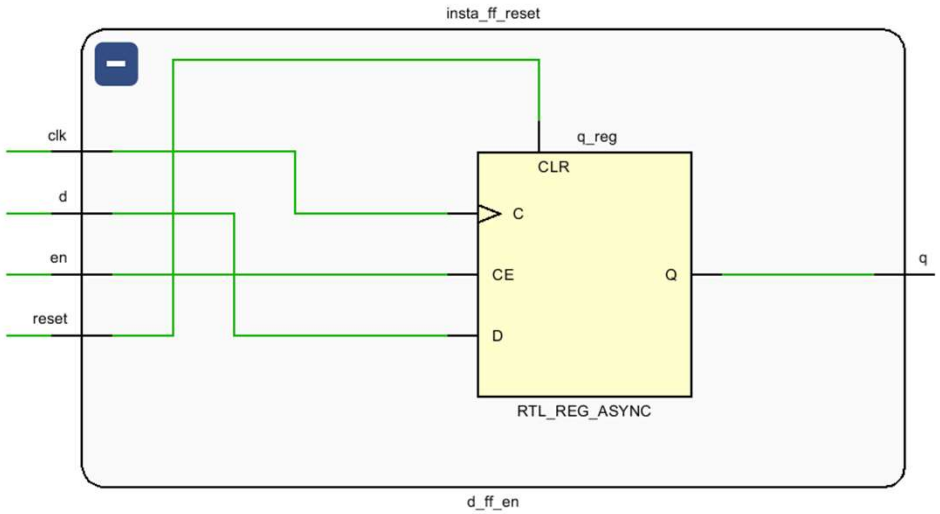
Codificación Flip flop tipo D con señales asincrónicas de inicialización y de habilitación

- La salida se actualiza solo en el flanco positivo (ascendente) de clk
- Si la seña de reset está en estado activo, la salida queda en cero
- La señal de inicialización (reset) tiene prioridad por sobre la señal de habilitación (en).

reset	clk	en	q*
1	-	-	0
0	0	-	q
0	1	-	q
0	↓	0	q
0	↓	1	d

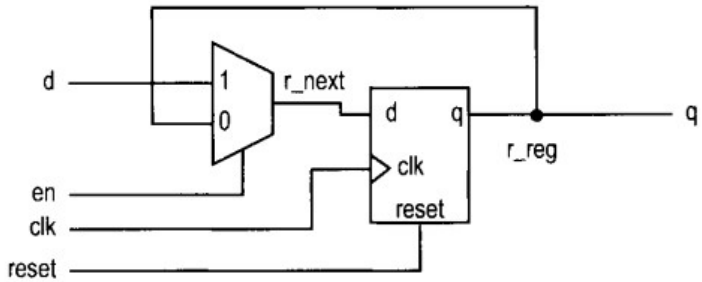
Codificación Flip flop tipo D con señales asincrónicas de inicialización y de habilitación

```
1  module d_ff_en(
2      input clk,
3      input reset,
4      input en,
5      input d,
6      output reg q
7      );
8      always @(posedge clk, posedge reset)
9          if (reset)
10             q <= 1'b0;
11         else
12             if (en)
13                 q <= d;
14 endmodule
```



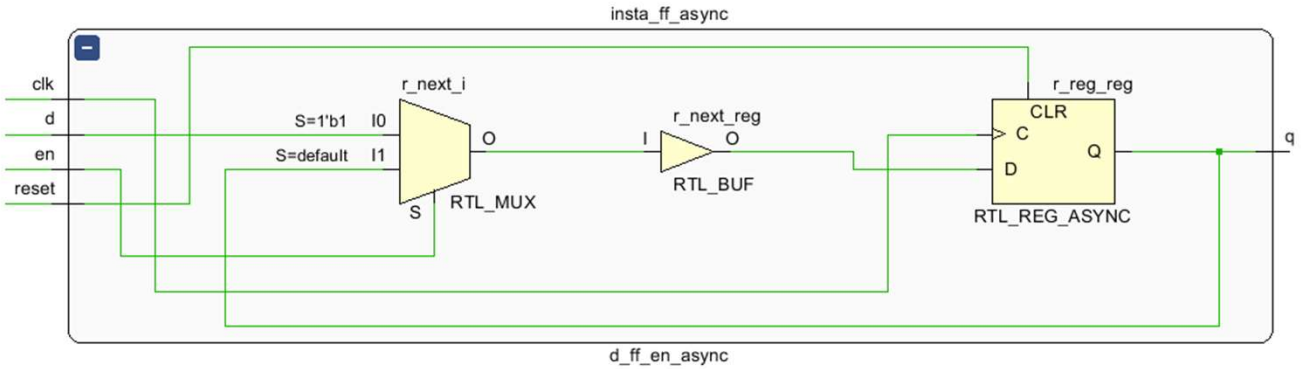
Codificación Flip flop tipo D con señales asincrónicas de inicialización y de habilitación a partir de un flip flop D con solo una señal asincrónica de inicialización

- La salida se actualiza solo en el flanco positivo (ascendente) de clk
- Si la seña de reset está en estado activo, la salida queda en cero
- La señal de inicialización (reset) tiene prioridad por sobre la señal de habilitación (en).
- Las señales de inicialización (reset) y habilitación (en) son asincrónicas
- Se implementa un circuito secuencial sincrónico con la lógica de siguiente estado cuya salida es el estado actual o la entrada en función de la entrada asincrónica “en”



Codificación Flip flop tipo D con señales asincrónicas de inicialización y de habilitación a partir de un flip flop D con solo una señal asincrónica de inicialización

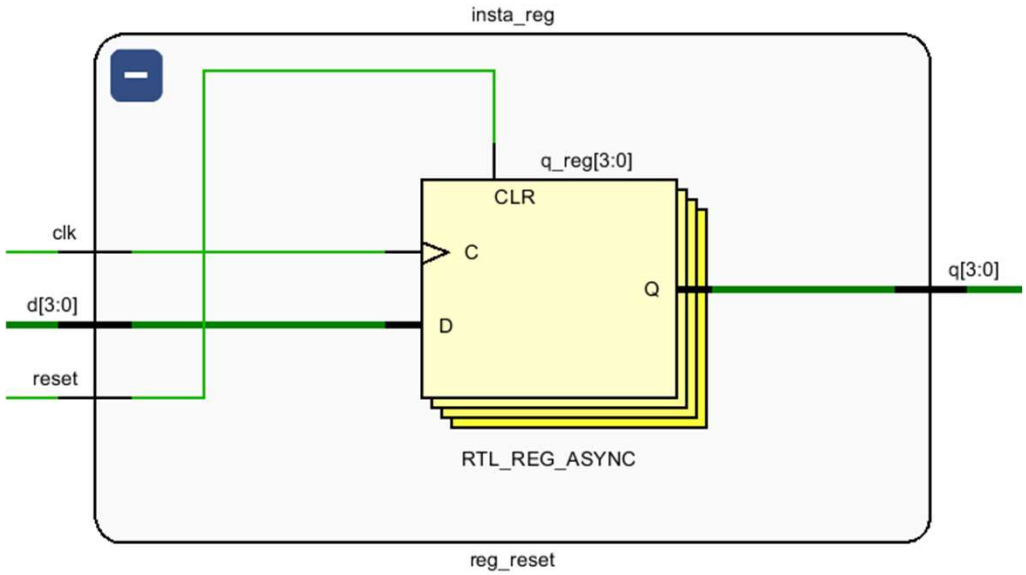
```
1  module d_ff_en_async(
2      input clk,
3      input reset,
4      input en,
5      input d,
6      output reg q
7      );
8      // keep mantiene las señales, y se infiere
9      // un FF tipo D, sino se infiere un FF
10     // con CE directamente como en d_ff_en
11     (* keep = "true" *) reg r_reg, r_next;
12     // FF
13     always @(posedge clk, posedge reset)
14         if (reset)
15             r_reg <= 1'b0;
16         else
17             r_reg <= r_next;
18      // logica de siguiente estado
19      always @*
20         if (en)
21             r_next = d;
22         else
23             r_next = r_reg;
24      // logica de salida
25      always @*
26         q = r_reg;
27 endmodule
```



Codificación – Registros

- Para formar un Registro a partir de un grupo de flip flops, se codifica de forma similar utilizando arreglos. Ejemplo con un registro de 4 bits a base de flip flops tipo D con reset:

```
1  module reg_reset(
2       input clk,
3       input reset,
4       input [3:0] d,
5       output reg [3:0] q
6       );
7       always @(posedge clk, posedge reset)
8           if (reset)
9               q <= 0;
10          else
11              q <= d;
12 endmodule
```

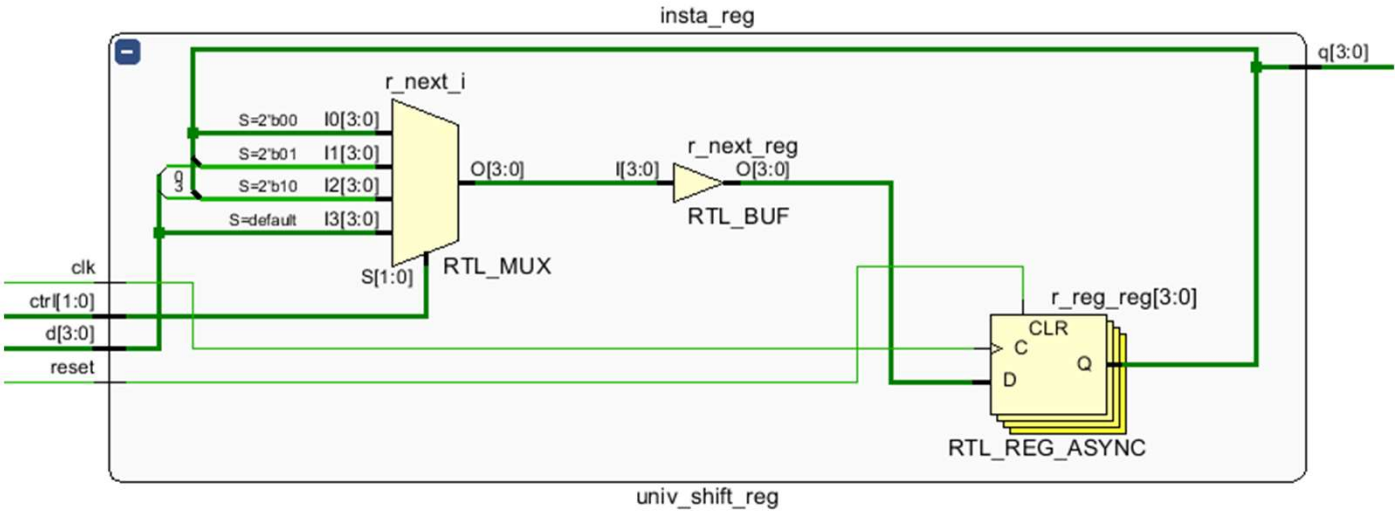


Ejemplos – Registro de desplazamiento universal

- El registro de desplazamiento universal carga datos en forma paralela, los desplaza un bit a derecha o izquierda o mantiene la salida sin cambios.
- Permite hacer conversiones serie-paralelo o paralelo-serie
- La operación deseada se determina mediante una señal de control de 2 bits (ctrl).

Ejemplos – Registro de desplazamiento universal

```
1  module univ_shift_reg
2      #(parameter N=8)
3      (
4      input clk,
5      input reset,
6      input [1:0] ctrl,
7      input [N-1:0] d,
8      output [N-1:0] q
9      );
10     reg [N-1:0] r_reg, r_next;
11     // registros
12     always @(posedge clk, posedge reset)
13         if (reset)
14             r_reg <= 0;
15         else
16             r_reg <= r_next;
17     // logica de siguiente estado
18     always @*
19         case (ctrl)
20             2'b00: r_next = r_reg; // no hacer nada
21             2'b01: r_next = {r_reg[N-2:0], d[0]}; // desp. izquierda
22             2'b10: r_next = {d[N-1], r_reg[N-1:1]}; // desp. derecha
23             default: r_next = d; // carga paralela
24         endcase
25     // logica de salida
26     assign q = r_reg;
27 endmodule
```



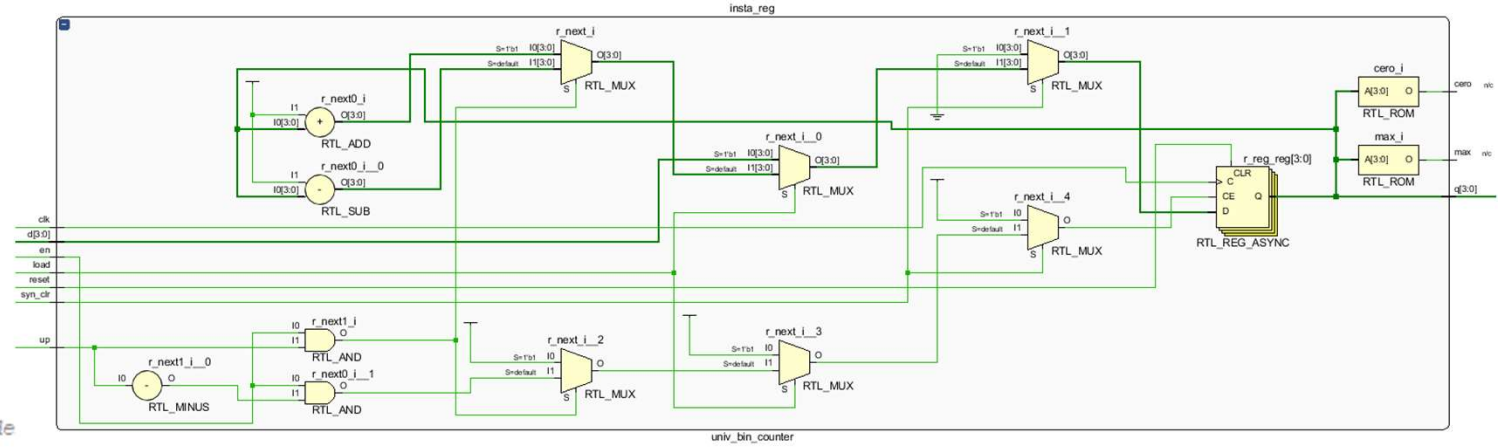
Ejemplos – Contador binario universal

- El contador binario universal puede contar en forma ascendente o descendente, pausar la cuenta, contar a partir de un número determinado o reinicializar desde cero.
- Las salidas son q (contador), max=1 cuando se alcanza el máximo de conteo y cero=1 cuando el contador está en 0.
- Tabla de funcionamiento:

syn_clr	load	en	up	q*	**Operation**
1	–	–	–	$00\cdots00$	synchronous clear
0	1	–	–	d	parallel load
0	0	1	1	q+1	count up
0	0	1	0	q−1	count down
0	0	0	–	q	pause

Ejemplos – Contador binario universal

```
1  module univ_bin_counter
2      #(parameter N=8)
3      (
4      input clk, reset, syn_clr, load, en, up,
5      input [N-1:0]d,
6      output max, cero,
7      output [N-1:0] q
8      );
9      reg [N-1:0] r_reg, r_next;
10     // FF
11     always @(posedge clk, posedge reset)
12         if (reset)
13             r_reg <= 0;
14         else
15             r_reg <= r_next;
16     // logica de siguiente estado
17     // if anidado para las prioridades
18     // de señales de control
19     always @*
20         if (syn_clr) // puesta a cero
21             r_next = 0;
22         else
23             if (load) // carga de valor
24                 r_next = d;
25             else
26                 if (en & up) // conteo ascende
27                     r_next = r_reg + 1;
28                 else
29                     if (en & -up) // conteo descendente
30                         r_next = r_reg - 1;
31                     else
32                         r_next = r_reg; // pausa
33      // logica de salida
34      assign q = r_reg;
35      assign max = (r_reg == 2**N-1) ? 1'b1 : 1'b0;
36      assign cero = (r_reg == 0) ? 1'b1 : 1'b0;
37 endmodule
```



Temario

- Introducción
- Sintaxis
- Módulos
- Estructuras IF y CASE
- Circuitos Secuenciales
- **Errores Comunes**
- Bibliografía

Errores comunes: Asignación múltiple

- Es sintácticamente correcto que una misma variable pueda aparecer en más de un bloque always:

```
reg y;
reg a, b, clear;
. . .
always @*
   if (clear) y = 1'b0;

always @*
   y = a & b;
```

- Este código puede simularse, pero no implementarse, ya que generaría una señal controlada por dos circuitos distintos. La forma correcta es tener ambas asignaciones en un solo bloque always

```
always @*
   if (clear)
      y = 1'b0;
   else
      y = a & b;
```

Errores comunes: lista de sensibilidad incompleta

- En los circuitos combinacionales la salida es función de todas las entradas, por lo que cualquier cambio en las entradas debe reflejarse en las salidas.
- Ejemplo para una compuerta AND de dos entradas:

```
always @(a, b)
   y = a & b;
```

- Si se omite una de las entradas (p.ej. b):

```
always @(a)
   y = a & b;
```

- El código es sintácticamente correcto, pero el comportamiento es muy distinto. Los cambios de 'a' generan la operación deseada, pero los cambios de 'b' no activan el bloque y por lo tanto no se actualiza la salida, conservando su valor anterior, lo que implica el instanciado en forma indirecta de un elemento de memoria, pasando a ser un circuito secuencial.
- Para evitar este problema es recomendable usar always @* en los circuitos combinacionales para asegurar que todas las señales de entrada generen una actualización de la señal de salida.

Errores comunes: no contemplar todas las posibilidades o no asignar todas las salidas en todos los casos

- Verilog establece que una variable debe mantener su valor si no se le asigna un valor nuevo al evaluar un bloque always.
- Por esta razón se infiere un elemento de memoria (latch) en los casos en que es necesario mantener un valor previo de salida al evaluar el bloque always.
- Para evitar esto en los circuitos combinacionales, se debe asignar un valor a todas las salidas en todos los casos que se evalúa el bloque always.
- La forma de asignar un valor a todas las salidas en todos los casos es incluir todas las ramificaciones en estructuras IF o CASE y asignar un valor a todas las salidas en cada una de las ramificaciones

Errores comunes: no contemplar todas las posibilidades o no asignar todas las salidas en todos los casos – Ejemplo IF

- Se busca un circuito que compare dos números y genere las salidas “mayor que”(gt) y “igual a”(eq)

```
always @*
   if (a > b)
      gt = 1'b1;
   else if (a == b)
      eq = 1'b1;
```

- Si a < b (ambos IF se evalúan falso), ninguna de las salidas tiene un valor asignado, por lo que se infiere un latch para almacenar el valor previo de las salidas gt y eq (no contemplar todas las posibilidades).
- Por otra parte, si, por ejemplo, a > b se evalúa verdadero, se asigna un valor a gt, pero no a eq, por lo que se infiere un latch para almacenar el valor previo de eq (no asignar todas las salidas en todos los casos)

Errores comunes: no contemplar todas las posibilidades o no asignar todas las salidas en todos los casos – Ejemplo IF

- Solución A: agregar el ELSE faltante y asignar todas las salidas en todas las ramificaciones

```
always @*
   if (a > b)
      begin
         gt = 1'b1;
         eq = 1'b0;
      end
   else if (a == b)
      begin
         gt = 1'b0;
         eq = 1'b1;
      end
   else    // i.e., a < b
      begin
         gt = 1'b0;
         eq = 1'b0;
      end
```

Errores comunes: no contemplar todas las posibilidades o no asignar todas las salidas en todos los casos – Ejemplo IF

- Solución B: asignar un valor por defecto a las salidas al principio del bloque always

```
always @*
begin
    gt = 1'b0;   // default value for gt
    eq = 1'b0;   // default value for eq
    if (a > b)
        gt = 1'b1;
    else if (a == b)
        eq = 1'b1;
end
```

Errores comunes: no contemplar todas las posibilidades o no asignar todas las salidas en todos los casos – Ejemplo CASE

- Cuando no se implementa un Full CASE, se infieren elementos de memoria

```
reg [1:0] s
case (s)
   2'b00: y = 1'b1;
   2'b10: y = 1'b0;
   2'b11: y = 1'b1;
endcase
```

- La entrada 2'b01 no está cubierta por ninguna rama del CASE, por lo que se infiere un latch para almacenar el valor anterior de 'y' para esta entrada.
- Una solución es usar un caso por defecto (default)

```
case (s)
   2'b00: y = 1'b1;
   2'b10: y = 1'b0;
   default: y = 1'b1;
endcase
```

Errores comunes: no contemplar todas las posibilidades o no asignar todas las salidas en todos los casos – Ejemplo CASE

- Una segunda solución, si sabemos que la combinación 2'b01 nunca va a estar presente, es agregar una rama con una salida "don't care"

```
case (s)
   2'b00: y = 1'b1;
   2'b10: y = 1'b0;
   2'b11: y = 1'b1;
   default: y = 1'bx;
endcase
```

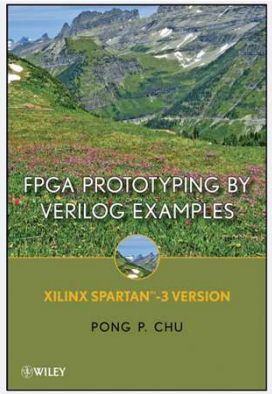
- Otra posibilidad es asignar un valor por defecto al principio del bloque always

```
y = 1'b0;    // can also use y = 1'bx for don't-care
case (s)
   2'b00: y = 1'b1;
   2'b10: y = 1'b0;
   2'b11: y = 1'b1;
endcase
```

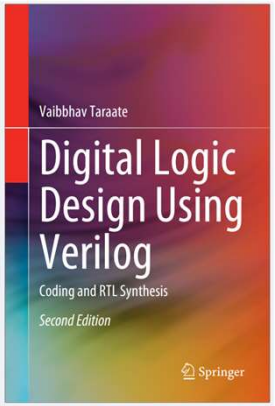
Temario

- Introducción
- Sintaxis
- Módulos
- Estructuras IF y CASE
- Circuitos Secuenciales
- Errores Comunes
- **Bibliografía**

Bibliografía



- "FPGA Prototyping by Verilog Examples", Pong P. Chu, Wiley (2008)



- "Digital Logic Design Using Verilog: Coding and RTL Synthesis 2nd Ed.", Vaibbhav Taraate, Springer (2022)