

# Verilog

Microarquitecturas y softcores



Laboratorio de  
**Sistemas Embebidos**



- **Lenguajes descriptores de hardware**

Un lenguaje descriptor de hardware es un lenguaje para la descripción formal y el diseño de circuitos electrónicos. Puede ser utilizado para describir el funcionamiento de un circuito, su diseño y organización, y para realizar las pruebas necesarias para verificar el correcto funcionamiento.

Ejemplos de este tipo de lenguaje son VHDL, **Verilog**, ABEL, AHDL, SystemC, SystemVerilog).

- **Objetivos de los HDL's**

- Especificación de circuitos electrónicos
- Documentación
- Simulación/Verificación de los circuitos antes de ser implementados

## • Introducción

- Verilog es un HDL (Hardware Description Language)
- Fue desarrollado en 1985 por AIDS (Automated Integrated Design Systems), más tarde renombrada como Gateway Design Automation.
- Esta empresa creció fuertemente y fue adquirida por Cadence Design Systems en 1989.
- Verilog fue desarrollado como un **lenguaje de simulación**. La idea de que fuera sintetizable fue muy posterior.
- A finales de los 80's se observó que los diseñadores empezaron a elegir el estándar del DoD de USA, llamado VHDL.

## • Introducción

- En ese momento Cadence decidió abrir el lenguaje, y hacerlo público en 1990.
- Así nació OVI (Open Verilog International)
- A partir de ese momento varias compañías comenzaron a diseñar simuladores para el lenguaje. El primero dio a luz en 1992.
- En 1993 se estableció un grupo de trabajo en IEEE para establecer el estándar del lenguaje
- En 1995 fue formalmente lanzado bajo la denominación **IEEE-1364**
- En 2002 se lanzó una revisión, conocida como **IEEE 1364-2001**.
- Debido a numerosos errores se lanzó en 2003 una nueva revisión denominada **IEEE 1364-2001 Revision C**.

- **Elementos sintácticos básicos del lenguaje**

- **Identificador** (Identifier). Debe comenzar con (a-z, A-Z, \_). Puede contener (a-z, A-Z, 0-9, \_, \$). Largo máximo de 1024 caracteres.
- **Palabra reservada** (Keyword). Siempre van en minúsculas.
- **Espacio en blanco** (incluye los caracteres de espacio, tabulación y nueva línea)
- **Comentario** (// para comentarios por línea y /\* \*/ para comentarios en bloque)

- **Tipos de datos**

En la mayoría de los tipos de datos se utilizan 4 valores básicos:

- **0**: representa un '0' lógico
- **1**: representa un '1' lógico
- **z**: representa el estado de alta impedancia
- **x**: representa un valor desconocido

- **Grupos de tipos de datos**

Verilog tiene dos grupos principales de tipos de datos



**Net**

**Variable**



- Grupos de tipos de datos: Net

**Net:** los tipos de datos en este grupo representan conexiones físicas entre componentes de hardware. Se usan como salidas de **asignaciones continuas** y como conexiones entre diferentes módulos.

Ej: **wire**, supply0, supply1, tri, triand, trior, tri0, tri1, wand, wor.

- Grupos de tipos de datos: Net

El tipo wire representa una señal de 1 bit

```
wire a0, a1;           // senales de un bit
```

Si agrupamos varios bits se tiene:

```
wire [3:0] dato1; (1)           wire [7:0] mem [31:0]  
wire [0:7] dato2; (2)
```

El rango del vector puede ser descendente (como en 1) o ascendente (como en 2). Para la representación de datos se prefiere el primer caso, ya que los pesos binarios quedarían alineados. El MSB correspondería a *dato1*[3].

- Grupos de tipos de datos: Net

Volviendo al arreglo bidimensional, cómo se accede a cada parte?

```
wire [7:0] mem [31:0]
```

Acceso a una palabra determinada del vector:

```
// acceso a la tercera palabra del vector
```

```
mem[2] = 8'b0011_1101;
```

Acceso a un bit de una palabra determinada del vector:

```
// acceso al segundo bit de la tercera
```

```
// palabra del vector
```

```
mem[2][1] = 1'b0;
```

- Grupos de tipos de datos: **Variable**

**Variable:** los tipos de datos en este grupo representan almacenamiento abstracto y son usados en las salidas de las **asignaciones procedurales**.

Ej: integer, real, realtime, reg, time.

## • Representación de números

La forma general de representación de una constante numérica es:

[signo] [tamaño] ' [base] [valor]

- [signo] = signo del número
- [tamaño] = número de bits en el número (opcional)  
Si se encuentra presente el número se denomina **sized**, de lo contrario **unsized**
- [base] = base del número  
b (binario) , o (octal), h (hexadecimal), d (decimal)
- [valor] = valor del número en la base correspondiente

- **Representación de números**

La forma general de representación de una constante numérica es:

[signo] [tamaño] ' [base] [valor]

Ejemplos:

6'b011101	→	011101
8'b0111_0100	→	01110100
5'h1E	→	11110
5'd12	→	01100

- **Representación de números**

**Sized number:** el número de bits que representa el número se encuentra presente de manera explícita. Si el tamaño del número es menor que [tamaño] se efectúa un agregado de ceros al inicio. En algunos casos especiales el caracter agregado puede variar:

- Si el valor es z, entonces se agregan z's
- Si el valor es x, entonces se agregan x's
- Si el tipo de dato usado es ***signed***, entonces se extiende el MSB

- **Representación de números**

**Sized number (Ejemplos)**

6'b0	→	000000
6'b1	→	000001
6'bx	→	xxxxxx
6'bz	→	zzzzzz
6'bx011	→	xxx011
-6'b000001	→	111111





- Representación de números

## Ejercicio

```
module prueba_numeros;  
    reg [7:0] f;  
    reg signed [7:0] g;  
    reg [7:0] h;  
  
    initial begin  
        f = 8'b10000000;  
        g = 8'b1000_0000;  
        h = -8'd2;  
        #100;  
        f = f >>> 2;  
        g = g >>> 2;  
        h = h >>> 4;  
    end  
endmodule
```

- **Estructura de un código en Verilog**

**Declaración de puertos de entrada y salida:** especifica el modo, el tipo y el nombre de los puertos de entrada/salida

**Declaración de señales:** declaración de las señales que se utilizan en la descripción del hardware.

**Cuerpo del módulo:** se describe la manera en que se interconectan las entradas con las salidas

- Estructura de un código en Verilog

## Declaración de puertos de entrada / salida

```
module [nombre_del_modulo]
(
    [modo] [tipo_de_dato] [nombre_de_puerto],
    [modo] [tipo_de_dato] [nombre_de_puerto],
    ...
    [modo] [tipo_de_dato] [nombre_de_puerto]
);
```

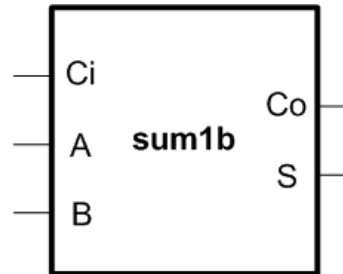
*[modo] = **input**, **output** o **inout***

*[tipo\_de\_dato] = si es **wire** puede omitirse*

- Estructura de un código en Verilog

## Declaración de puertos de entrada / salida

```
module sum1b  
(  
    input wire a, b, ci,  
    output wire s, co  
);
```



- Estructura de un código en Verilog

### Declaración de puertos de entrada / salida (Verilog-1995)

```
module [nombre_del_modulo] ([nombre_de_puertos])  
    [modo] [nombre_de_puerto_1];  
    [modo] [nombre_de_puerto_2];  
    ...  
    [modo] [nombre_de_puerto_n];  
    [tipo_de_dato] [nombre_de_puerto_1];  
    [tipo_de_dato] [nombre_de_puerto_2];  
    ...  
    [tipo_de_dato] [nombre_de_puerto_1];
```

- Estructura de un código en Verilog

### Declaración de puertos de entrada / salida (Verilog-1995)

```
module sum1b (a, b, ci, s, co)
    // declaracion del modo
    input a, b, ci;
    output s, co;
    // declaracion del tipo de dato
    wire a, b, ci;
    wire s, co;
```

- **Estructura de un código en Verilog**

## **Cuerpo del módulo**

El cuerpo de un módulo puede ser visto como un conjunto de partes de un circuito conectadas entre sí. Todas estas partes operan concurrentemente.

Existen diferentes formas de describir cada una de esas partes:

- **Asignación continua**
- **Bloque always**
- **Instanciación de módulo**



- Estructura de un código en Verilog

## Cuerpo del módulo

- Asignación continua

Muy útil para describir circuitos combinacionales simples.

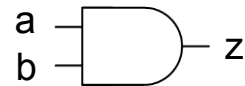
Su sintaxis es:

***assign*** *[nombre\_senal]* = *[expresión]*;

Cada asignación continua puede pensarse como una parte del circuito.

Ejemplo:

***assign*** *z* = *a* & *b*;



- Estructura de un código en Verilog

## Cuerpo del módulo

### - Bloque always

Dentro de este bloque se pueden utilizar asignaciones procedurales más abstractas, por lo que se puede usar para describir circuitos más complejos.

Su sintaxis es:

```
always @ (lista_de_sensibilidad) begin  
    [deklaración de variables]  
  
    [sentencia procedural 1];  
    [sentencia procedural 2];  
    ...  
end
```

- **Estructura de un código en Verilog**

## **Cuerpo del módulo**

- **Bloque always**

- Los elementos dentro del bloque se setean o actualizan cuando se satisface la lista de sensibilidad.
- Los elementos dentro del bloque se setean o actualizan de manera secuencial o paralela.
- Existen dos tipos de asignaciones:
  - non-blocking (se utiliza el símbolo <=)
  - blocking (se utiliza el símbolo =)

- Estructura de un código en Verilog

## Cuerpo del módulo

- Bloque **always**

- Asignaciones non-blocking

Ocurren en paralelo

Si se tienen varias asignaciones de este tipo, dentro de un bloque **always**, todas son establecidas al mismo tiempo.

Ejemplo:

```
always @ (lista_de_sensibilidad) begin  
    b <= a;  
    c <= b;  
    d <= c;  
end
```

b toma el valor de a, c toma el valor  
“viejo” de b y d el “viejo” de c

- Estructura de un código en Verilog

## Cuerpo del módulo

- Bloque **always**

- Asignaciones non-blocking

Ocurren en paralelo

Si se tienen varias asignaciones de este tipo, dentro de un bloque **always**, todas son establecidas al mismo tiempo.

Ejemplo:

```
always @ (lista_de_sensibilidad) begin  
    b <= a;  
    c <= b;  
    d <= c;  
end
```

Se utiliza para la descripción de comportamientos secuenciales

- Estructura de un código en Verilog

## Cuerpo del módulo

- Bloque **always**

- Asignaciones **blocking**

Ocurren en secuencia

Si se tienen varias asignaciones de este tipo, dentro de un bloque **always**, todas son establecidas una detrás de la otra.

Ejemplo:

```
always @ (lista_de_sensibilidad) begin
```

```
    b = a;
```

```
    c = b;
```

```
    d = c;
```

```
end
```

b toma el valor de a, c toma el valor de b y d el de c. Finalmente b, c y d toman el valor de a

- Estructura de un código en Verilog

## Cuerpo del módulo

- Bloque **always**

- Asignaciones **blocking**

Ocurren en secuencia

Si se tienen varias asignaciones de este tipo, dentro de un bloque **always**, todas son establecidas una detrás de la otra.

Ejemplo:

```
always @ (lista_de_sensibilidad) begin  
    b = a;  
    c = b;  
    d = c;  
end
```

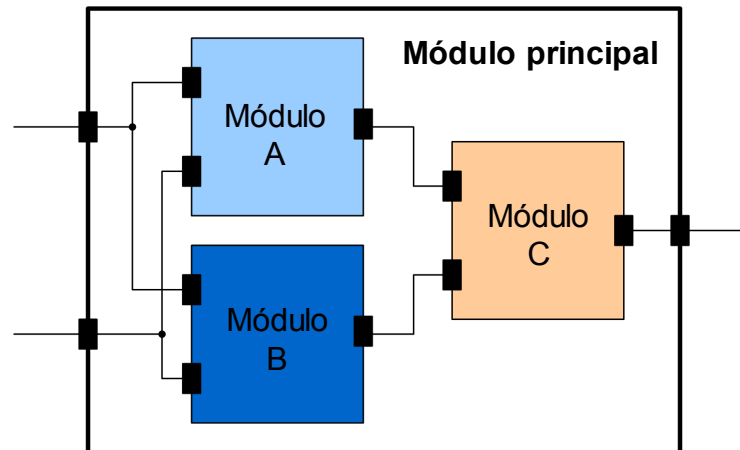
Se utiliza para la descripción de circuitos combinacionales

- Estructura de un código en Verilog

## Cuerpo del módulo

- Instanciación de módulo

- Se crea una instancia de un módulo dentro de otro.
- Permite al diseñador agregar un módulo predefinido, como un subsistema del módulo principal.





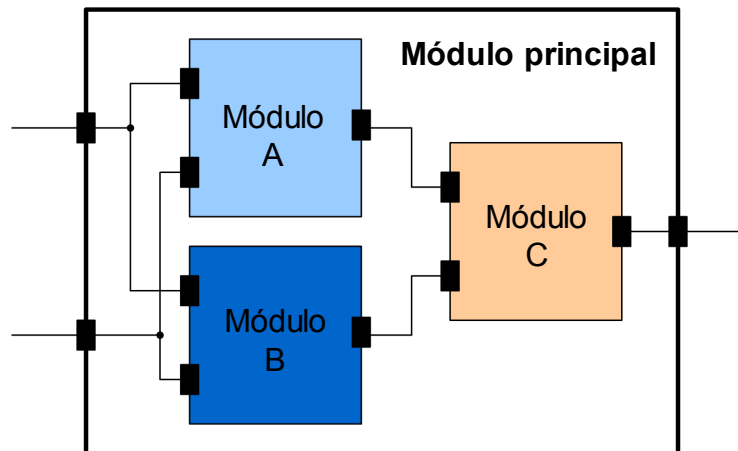
- Estructura de un código en Verilog

## Cuerpo del módulo

- Instanciación de módulo

La inclusión de un módulo se realiza de la siguiente manera:

*nombre\_modulo nombre\_instancia (lista\_de\_puertos)*



## • Estructura de un código en Verilog

### Cuerpo del módulo

#### - Instanciación de módulo

- Los módulos se pueden conectar especificando esas conexiones de dos modos:

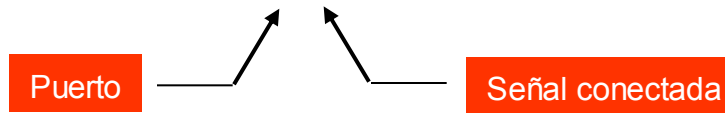
**Asociación posicional:** las conexiones a los puertos del módulo deben respetar el orden en que fueron definidos estos últimos.

```
sum1b sum1b_inst (a, b, ci, s, co);
```

← No se indica el puerto

**Asociación explícita:** las conexiones a los puertos pueden ser realizadas en cualquier orden.

```
sum1b sum1b_inst (.b(b), .a(a), .ci(ci), .s(s), .co(co));
```



## • Estructura de un código en Verilog

### Declaración de señales

- En esta parte se especifican las señales internas y los parámetros que son usados en el módulo.
- Las señales internas pueden pensarse como cables que interconectan las diferentes partes del circuito a describir.
- La sintaxis simplificada de una declaración de señal es la siguiente:

[tipo\_de\_dato] [nombre\_de\_la\_señal]

Ejemplos:

```
wire a0, a1, a2;
```

```
reg y1;
```

- Estructura de un código en Verilog

## Declaración de señales

- En Verilog, un identificador no debe necesariamente haber sido declarado antes de su uso.

En el caso en que se encuentre un identificador con esta característica se lo considera del tipo **wire**. Esto se conoce como *implicit net*.

Ejemplos:

- Estructura de un código en Verilog

## Declaración de señales

Declaración explícita

```
module eq1 (  
    // Puertos de entrada/salida  
    input wire i0, i1,  
    output wire eq  
);  
  
// Declaracion de senales  
wire p0, p1;  
  
// Cuerpo  
assign eq = p0 | p1;  
assign p0 = ~i0 & ~i1;  
assign p1 = i0 & i1 ;  
endmodule
```

```
module eq1 (  
    // Puertos de entrada/salida  
    input wire i0, i1,  
    output wire eq  
);  
  
// Declaracion de senales  
  
// Cuerpo  
assign eq = p0 | p1;  
assign p0 = ~i0 & ~i1;  
assign p1 = i0 & i1 ;  
endmodule
```

Declaración implícita  
(implicit net)

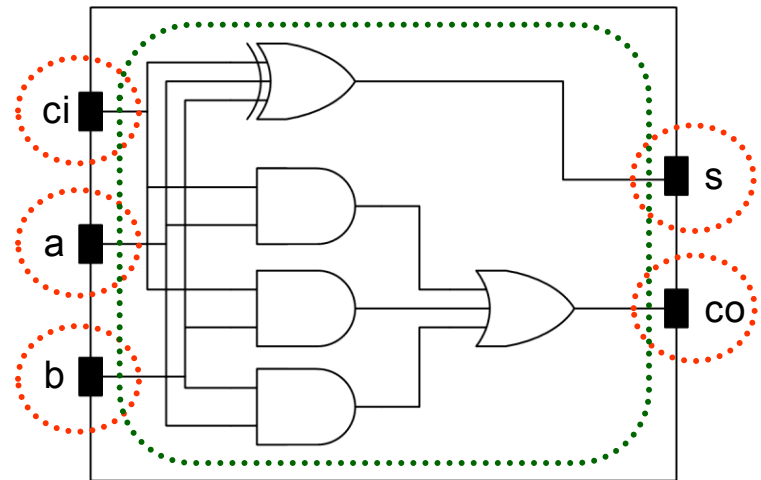
- Estructura de un código en Verilog

### Ejemplo: Sumador de 1 bit

Declaración de puertos

```
module sum1b (  
    input wire a, b, ci,  
    output wire s, co  
);  
  
    assign s = a ^ b ^ ci;  
    assign co = (a & b) | (a & ci) | (b & ci);  
  
endmodule
```

Cuerpo del módulo



- Estructura de un código en Verilog

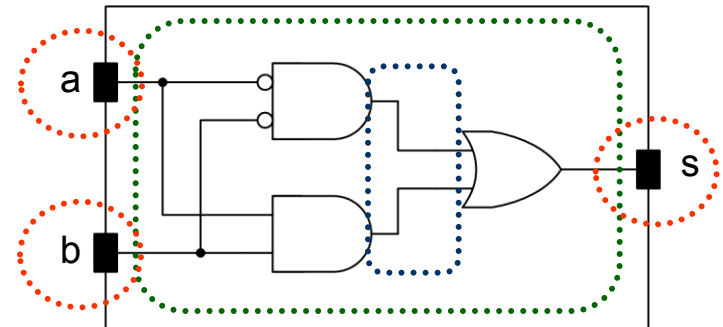
### Ejemplo: Comparador de 1 bit

Declaración de componentes

```
module comp1b (  
    input wire a, b,  
    output wire eq  
);  
    wire i0, i1;  
    assign eq = i0 | i1;  
    assign i0 = ~a & ~b;  
    assign i1 = a & b;  
endmodule
```

Declaración de señales

Cuerpo del módulo



- Operadores en Verilog

### Operadores aritméticos

+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo
**	Exponenciación



- Operadores en Verilog

## Operadores de desplazamiento

>>	Desplazamiento a derecha lógico
<<	Desplazamiento a izquierda lógico
>>>	Desplazamiento a derecha aritmético
<<<	Desplazamiento a izquierda aritmético

Ej: Sea  $a = 4'b1100$

$y = x \gg 1;$                       // y resulta  $4'b0110$

$y = x \ll 1;$                       // y resulta  $4'b1000$

$y = x \ll 2;$                       // y resulta  $4'b0000$

- Operadores en Verilog

## Operadores de relación

>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que

Ej: Sea  $a = 4$ ,  $b = 3$ ,  $x = 4'b1010$ ,  $y = 4'b1101$ ,  $z = 4'b1xxx$

$a <= b$       // se evalúa como 0 lógico

$a > b$       // se evalúa como 1 lógico

$y >= x$       // se evalúa como 1 lógico

$y < z$       // se evalúa como x

- Operadores en Verilog

## Operadores de igualdad

==	Igualdad
!=	Desigualdad
===	Igualdad, incluyendo x y z
!==	Desigualdad, incluyendo x y z

Ej: Sea  $a = 4$ ,  $b = 3$ ,  $x = 4'b1010$ ,  $y = 4'b1101$ ,  $z = 4'b1xxz$ ,

$m = 4'b1xxz$ ,  $n = 4'b1xxx$

$a == b$  // se evalúa como 0 lógico

$x != y$  // se evalúa como 1 lógico

$x == z$  // se evalúa como x

- Operadores en Verilog

## Operadores de igualdad

==	Igualdad
!=	Desigualdad
===	Igualdad, incluyendo x y z
!==	Desigualdad, incluyendo x y z

Ej: Sea  $a = 4$ ,  $b = 3$ ,  $x = 4'b1010$ ,  $y = 4'b1101$ ,  $z = 4'b1xxz$ ,

$m = 4'b1xxz$ ,  $n = 4'b1xxx$

$z === m$             // se evalúa como 1 lógico

$z === n$             // se evalúa como 0 lógico

$m !== n$             // se evalúa como 1 lógico

- Operadores en Verilog

### Operadores de bit a bit

~	Negación (un operando)
&	And
	Or
^	Xor

Ej: Sea  $a = 4'b1010$ ,  $b = 4'b0000$

$a | b$  // se evalúa como  $4'b1010$

$a \& b$  // se evalúa como  $4'b0000$

$a \wedge b$  // se evalúa como  $4'b1010$

$\sim a$  // se evalúa como  $4'b0101$

- Operadores en Verilog

## Operadores de reducción

&	And de reducción (un operando)
	Or de reducción (un operando)
^	Xor de reducción (un operando)

Ej: Sea  $a = 4'b1010$

$\&a$  // se evalúa como 0 lógico

$|a$  // se evalúa como 1 lógico

$^a$  // se evalúa como 0 lógico

- Operadores en Verilog

## Operadores lógicos

!	Negación lógica (un operando)
&&	And lógica
	Or lógica

Ej: Sea a = 3, b = 0

(a && b)      // se evalúa como 0

(a || b)      // se evalúa como 1

!a            // se evalúa como 0

!b            // se evalúa como 1

- Operadores en Verilog

## Operadores de concatenación

{ }	Concatenación
{ { } }	Replicación

Ej: Sea  $a = 1'b1$ ,  $b = 2'b00$ ,  $c = 2'b10$ ,  $d = 3'b110$

$y = \{b, c\};$  // y resulta  $4'b0010$

$y = \{a, b, c, d, 3'b001\};$  // y resulta  $11'b10010110001$

$y = \{a, b[0], c[1]\}$  // y resulta  $3'b101$

$y = \{4\{a\}\};$  // y resulta  $4'b1111$

$y = \{4\{a\}, 2\{b\}\};$  // y resulta  $8'b11110000$

$y = \{4\{a\}, 2\{b\}, c\}$  // y resulta  $10'b1111000010$



- Operadores en Verilog

## Operadores de condición

?	Condicional
---	-------------

Ej: Cálculo del máximo entre dos números

```
assign max = (a>b) ? a : b;
```

Ej: Cálculo del máximo entre tres números

```
assign max = (a>b) ? ((a>c) ? a : c) :  
              ((b>c) ? b : c);
```

## • Operadores en Verilog

### Precedencia de operadores

Operador	Precedencia
! ~ + - (unario)	Mayor
**	
* / %	
+ - (binario)	
>> << >>> <<<	
< <= > >=	
== != === !==	
&	
^	
&&	
?:	Menor

## • Operadores en Verilog

### **Ajuste del tamaño en bits de una expresión**

En una sentencia Verilog los operandos pueden tener diferentes tamaños.

El ajuste se realiza siguiendo ciertas reglas:

- Se determina el máximo tamaño de los operandos (tanto la sentencia a la derecha como la señal a la izquierda)
- Se extiende la cantidad de bits de los operandos de la derecha al máximo y se evalúa la expresión
- Se asigna el resultado a la señal de la izquierda. Si el tamaño de la señal es menor, se truncan los bits más significativos



- Operadores en Verilog

## Ajuste del tamaño en bits de una expresión

Algunos ejemplos:

```
wire [7 : 0] a, b;  
wire [7 : 0] sum8;  
wire [8 : 0] sum;
```

```
assign sum8 = a + b;
```

```
assign sum = a + b;
```

← Se pierde el bit de carry

← Se obtiene bit de carry  
en sum[8]

- Operadores en Verilog

## Ajuste del tamaño en bits de una expresión

Algunos ejemplos:

```
wire [7 : 0] a, b;  
wire [7 : 0] sum8;  
wire [8 : 0] sum;
```

```
assign sum8 = a + b;
```

```
assign sum = a + b;
```

← Se pierde el bit de carry

← Se obtiene bit de carry  
en sum[8]

- Operadores en Verilog

## Ajuste del tamaño en bits de una expresión

Algunos ejemplos:

```
wire [7 : 0] a, b;  
wire [7 : 0] sum1, sum2;  
  
assign sum1 = (a + b) >> 1;  
assign sum2 = (0 + a + b) >> 1;
```

- Operadores en Verilog

## Ajuste del tamaño en bits de una expresión

Algunos ejemplos:

```
wire [7 : 0] a, b;  
wire [7 : 0] sum1, sum2;  
  
assign sum1 = (a + b) >> 1;  
assign sum2 = (0 + a + b) >> 1;
```

Si  $a + b$  no genera carry las dos sentencias generarán el mismo valor.

Si  $a + b$  genera carry las dos sentencias generarán distintos valores.



## • Operadores en Verilog

### Procesos

- Son bloques que se ejecutan en paralelo
- Toda descripción de comportamiento debe declararse dentro de un proceso
- Existen dos tipos de procesos (también son llamados bloques concurrentes):
  - . Initial: se ejecuta una sola vez al inicio (en tiempo 0). **No es sintetizable**. Su uso está asociado a un testbench
  - . Always: su ejecución está controlada por eventos. **Es sintetizable**.

- Operadores en Verilog

## Procesos

- Algunos ejemplos ...



```
initial  
begin  
    clk = 0;  
    rst = '1';  
    ena = '1';  
end;
```

```
always @(a or b or sel)  
begin  
    if (sel == 1)  
        y = a;  
    else  
        y = b;  
end;
```

## • Operadores en Verilog

### Procesos

- Todas las asignaciones que se realizan dentro de un proceso (initial o always) se deben realizar sobre variables tipo **reg**, nunca sobre wire

```
wire clk,rst;  
reg ena;  
Initial  
begin  
    clk = 0;  Error  
    rst = 0;  Error  
    ena = 0;  
end
```

```
reg clk,rst;  
reg ena;  
Initial  
begin  
    clk = 0;  
    rst = 0;  
    ena = 0;  
end
```

- **Descripción de circuitos combinacionales**

### **Bloque Always para diseño de circuitos combinacionales**

Las sentencias procedurales son bastantes. Nos centraremos en tres de las más importantes, debido a que tienen una implementación en hardware clara.

- Asignación procedural bloqueante

- Sentencia If

- Sentencia Case

← Inferen estructuras de ruteo

- **Descripción de circuitos combinacionales**

**Bloque Always para diseño de circuitos combinacionales**

- Asignación procedural bloqueante
  - Sólo puede ser usada dentro de un bloque always (también en un bloque initial)
  - La expresión es evaluada e inmediatamente asignada a la variable, antes de la ejecución de la sentencia siguiente
  - Una expresión puede ser asignada a una salida del tipo reg, integer, real, time, y realtime (las últimas tres no son sintetizables).

- Descripción de circuitos combinacionales

## Bloque Always para diseño de circuitos combinacionales

- Asignación procedural bloqueante
- Ejemplo

```
module eq1 (  
    input wire i0, i1,  
    output reg eq  
);  
reg p0, p1;  
always @(i0, i1)  
begin  
    p0 = ~i0 & ~i1;  
    p1 = i0 & i1;  
    eq = p0 p1;  
end  
endmodule
```

Verilog-1995

**always** @(i0 or i1)

- **Descripción de circuitos combinacionales**

**Bloque Always para diseño de circuitos combinacionales**

- Asignación procedural bloqueante

En un circuito combinacional todas las entradas deben estar en la lista de sensibilidad.

Para evitar el olvido de alguna de ellas, en Verilog-2001 se puede utilizar @\*

El ejemplo del comparador de un bit quedaría:

- Descripción de circuitos combinacionales

## Bloque Always para diseño de circuitos combinacionales

- Asignación procedural bloqueante

```
module eq1 (  
    input wire i0, i1,  
    output reg eq  
);  
    reg p0, p1;  
    always @*  
    begin  
        p0 = ~i0 & ~i1;  
        p1 = i0 & i1;  
        eq = p0 p1;  
    end  
endmodule
```



- Descripción de circuitos combinacionales

## Bloque Always para diseño de circuitos combinacionales

```
module and3i (  
    input wire i0, i1, i2,  
    output reg y  
);  
always @*  
begin  
    y = i0;  
    y = y & i1;  
    y = y & i2;  
end  
endmodule
```

Utilización de sentencia procedural

```
module and3i (  
    input wire i0, i1, i2,  
    output wire y  
);  
    assing y = i0;  
    assing y = y & i1;  
    assing y = y & i2;  
endmodule
```

Utilización de asignación continua

- **Descripción de circuitos combinacionales**

### **Bloque Always para diseño de circuitos combinacionales**

- Sentencia IF

Es una sentencia procesural que sólo puede ser utilizada dentro de un bloque always.

La síntesis genera una estructura de ruteo

- Descripción de circuitos combinacionales

## Bloque Always para diseño de circuitos combinacionales

- Sentencia IF

```
if [expresion_booleana]  
  begin  
    [sentencia procedural 1];  
    [sentencia procedural 2];  
    ...  
  end  
else  
  begin  
    [sentencia procedural 1];  
    [sentencia procedural 2];  
    ...  
  end
```

- Descripción de circuitos combinacionales

## Bloque Always para diseño de circuitos combinacionales

- Sentencia IF

```
if [expresion_booleana]
  begin
    [sentencia_procedural 1];
    [sentencia_procedural 2];
    ...
  end
```

```
if [expresion_booleana_1]
  ...
else if [expresion_booleana_2]
  ...
else if [expresion_booleana_3];
  ...
else
  ...
```

- Descripción de circuitos combinacionales

### Bloque Always para diseño de circuitos combinacionales

- Sentencia IF

Ejemplo: Codificador de prioridad

Entrada	Salida
r	Y
1 - - -	100
01 - -	011
001 -	010
0001	001
0000	000

*Tabla de verdad*

- Descripción de circuitos combinacionales

## Bloque Always para diseño de circuitos combinacionales

- Sentencia IF

Ejemplo: Codificador de prioridad

```
module cod_prio (  
    input wire [4:1] r,  
    output reg [2:0] y  
);  
    always @*  
        if (r[4] == 1'b1)  
            y = 3'b100;  
        else if (r[3] == 1'b1)  
            y = 3'b011;  
        else if (r[2] == 1'b1)  
            y = 3'b010;  
        else if (r[1] == 1'b1)  
            y = 3'b001;  
        else  
            y = 3'b000;  
endmodule
```

- Descripción de circuitos combinacionales

## Bloque Always para diseño de circuitos combinacionales

- Sentencia IF

Ejemplo: Decoder 2 a 4

Entrada			Salida
ena	a(1)	a(0)	y
0	-	-	0000
1	0	0	0001
1	0	1	0010
1	1	0	0100
1	1	1	1000

*Tabla de verdad*

- Descripción de circuitos combinacionales

## Bloque Always para diseño de circuitos combinacionales

- Sentencia IF

Ejemplo: Decoder 2 a 4

```
module deco_2_a_4 (  
    input wire [1:0] a,  
    input wire [1:0] ena,  
    output reg [3:0] y  
);  
    always @*  
        if (ena == 1'b0)  
            y = 4'b0000;  
        else if (a == 2'b00)  
            y = 4'b0001;  
        else if (a == 1'b01)  
            y = 4'b0010;  
        else if (a == 1'b10)  
            y = 4'b0100;  
        else  
            y = 4'b1000;  
endmodule
```



- **Descripción de circuitos combinacionales**

### **Bloque Always para diseño de circuitos combinacionales**

- Sentencia CASE

Es una sentencia de decisión múltiple que compara una expresión contra otras tantas.

Si existe más de un matcheo se ejecuta la primera.

Puede tener un item default (optativo)

- Descripción de circuitos combinacionales

## Bloque Always para diseño de circuitos combinacionales

### - Sentencia CASE

```
case [expresion_case]
  [item]:
    begin
      [sentencia procedural 1];
      [sentencia procedural 2];
      ...
    end
  [item]:
    begin
      [sentencia procedural 1];
      [sentencia procedural 2];
      ...
    end
  [item]:
    begin
      [sentencia procedural 1];
      [sentencia procedural 2];
      ...
    end
end
```

```
...
[item]:
  begin
    [sentencia procedural 1];
    [sentencia procedural 2];
    ...
  end
default:
  begin
    [sentencia procedural 1];
    [sentencia procedural 2];
    ...
  end
endcase
```

- Descripción de circuitos combinacionales

## Bloque Always para diseño de circuitos combinacionales

- Sentencia CASE

Ejemplo: Codificador de prioridad

```
module cod_prio (  
    input wire [4:1] r,  
    output reg [2:0] y  
);  
    always @*  
        case (r)  
            4'b1000, 4'b1001, 4'b1010, 4'b1011,  
            4'b1100, 4'b1101, 4'b1110, 4'b1111:  
                y = 3'b100;  
            4'b0100, 4'b0101, 4'b0110, 4'b0111:  
                y = 3'b011;  
            4'b0010, 4'b0011:  
                y = 3'b010;  
            4'b0001:  
                y = 3'b001;  
            4'b0000:  
                y = 3'b000;  
        endcase  
endmodule
```

- Descripción de circuitos combinacionales

## Bloque Always para diseño de circuitos combinacionales

- Sentencia CASE

Ejemplo: Decoder 2 a 4

```
module deco_2_a_4 (  
    input wire [1:0] a,  
    input wire [1:0] ena,  
    output reg [3:0] y  
);  
    always @*  
        case ({ena, a})  
            3'b000, 4'b001, 4'b010, 4'b011: y = 4'b0000;  
            3'b100: y = 4'b0001;  
            3'b101: y = 4'b0010;  
            3'b110: y = 4'b0100;  
            3'b111: y = 4'b1000;  
        endcase  
endmodule
```

- **Descripción de circuitos combinacionales**

### **Bloque Always para diseño de circuitos combinacionales**

- Sentencia CASEZ

Es una variante de la sentencia CASE.

Permite el uso de “z” y “?” como don’t cares al momento de la comparación. Pueden presentarse tanto en la expresión del case como en los diferentes items a matchear

- Descripción de circuitos combinacionales

## Bloque Always para diseño de circuitos combinacionales

- Sentencia CASEZ

Ejemplo: Codificador de prioridad

```
module cod_prio (  
    input wire [4:1] r,  
    output reg [2:0] y  
);  
    always @*  
        casez (r)  
            4b1??? : y = 3b100;  
            4b01?? : y = 3b011;  
            4b001? : y = 3b010;  
            4b0001: y = 3b001;  
            4b0000: y = 3b000;  
        endcase  
endmodule
```

- **Descripción de circuitos combinacionales**

### **Bloque Always para diseño de circuitos combinacionales**

- Sentencia CASEX

Es una variante de la sentencia CASE.

Permite el uso de “z”, “x” y “?” como don’t cares al momento de la comparación. Pueden presentarse tanto en la expresión del case como en los diferentes items a matchear

- Descripción de circuitos combinacionales

## Bloque Always para diseño de circuitos combinacionales

- Sentencias CASE, CASEZ y CASEX

**FULL CASE:** Están especificados todos los posibles valores de la expresión del case.

**PARALLEL CASE:** Todos los valores de la expresión del case son mutuamente excluyentes

**NON PARALLEL CASE:** No todos los valores de la expresión del case son mutuamente excluyentes

En la síntesis de circuitos combinacionales se debe utilizar **FULL CASE**, ya que todas las entradas deben tener asociada una salida. En el caso de faltar algún valor de entrada se deberá utilizar *default*.



- **Descripción de circuitos combinacionales**

**Bloque Always para diseño de circuitos combinacionales**

Las sentencias condicionales se sintetizan mediante redes de ruteo (routing networks)

- **Priority Routing Network** (if - else)
- **Multiplexing Network** (parallel case)

- Descripción de circuitos combinacionales

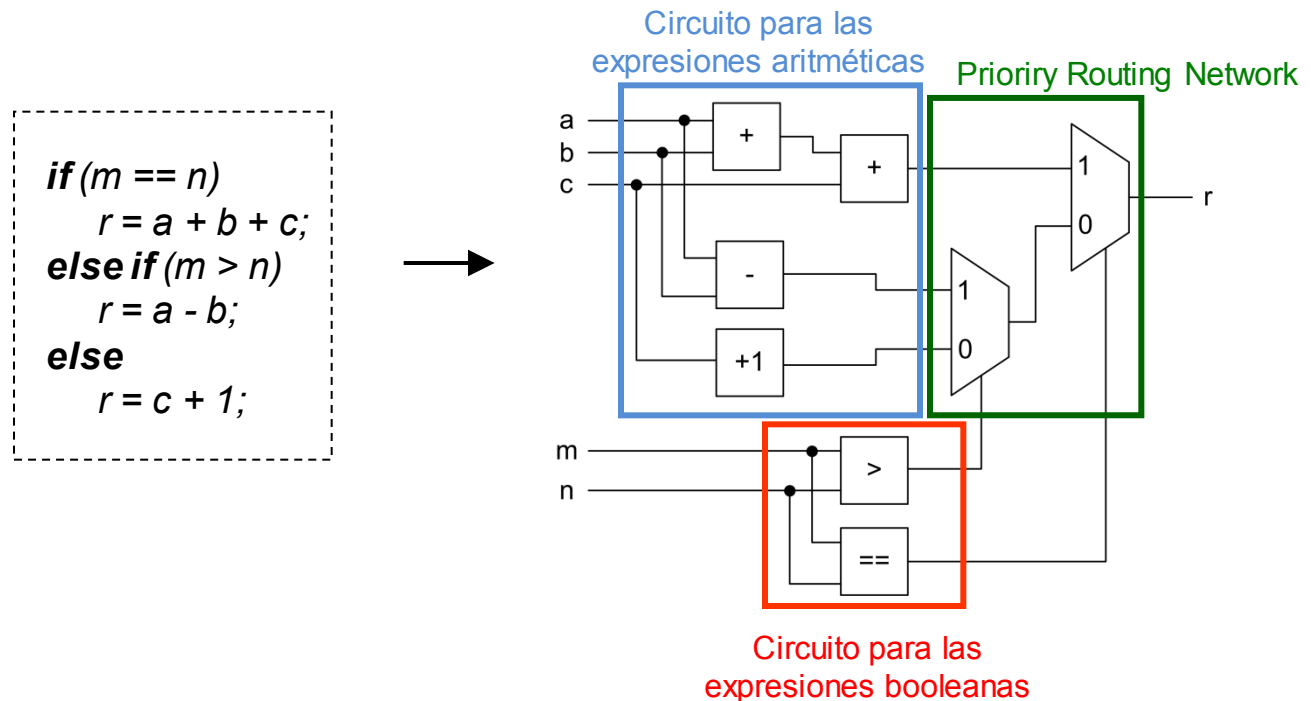
## Bloque Always para diseño de circuitos combinacionales

### - Priority Routing Network (if - else)

Es una red implementada por una secuencia de multiplexores 2 a 1.

Una sentencia if-else infiere este tipo de red.

Un código como el siguiente resulta en una red como la mostrada.



- Descripción de circuitos combinacionales

## Bloque Always para diseño de circuitos combinacionales

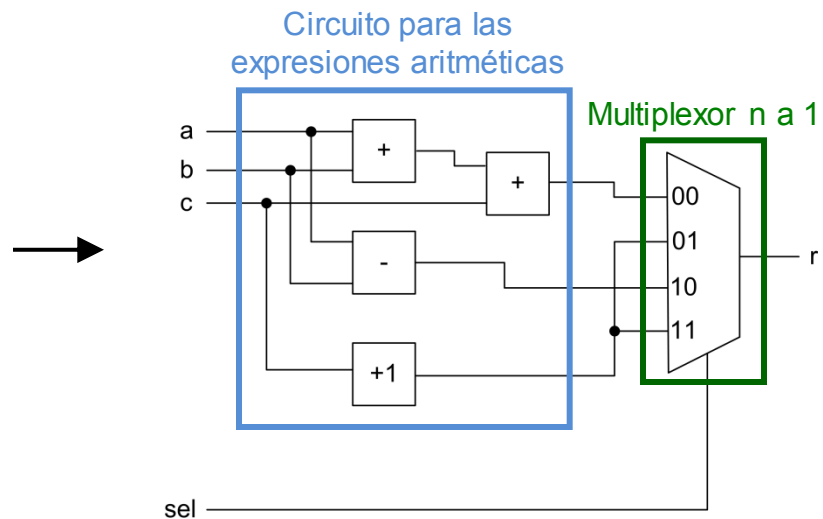
- Multiplexing Network (parallel case)

Es una red implementada por un multiplexor n a 1.

Cada valor de la expresión del case se mapea a una entrada del multiplexor.

Un código como el siguiente resulta en una red como la mostrada.

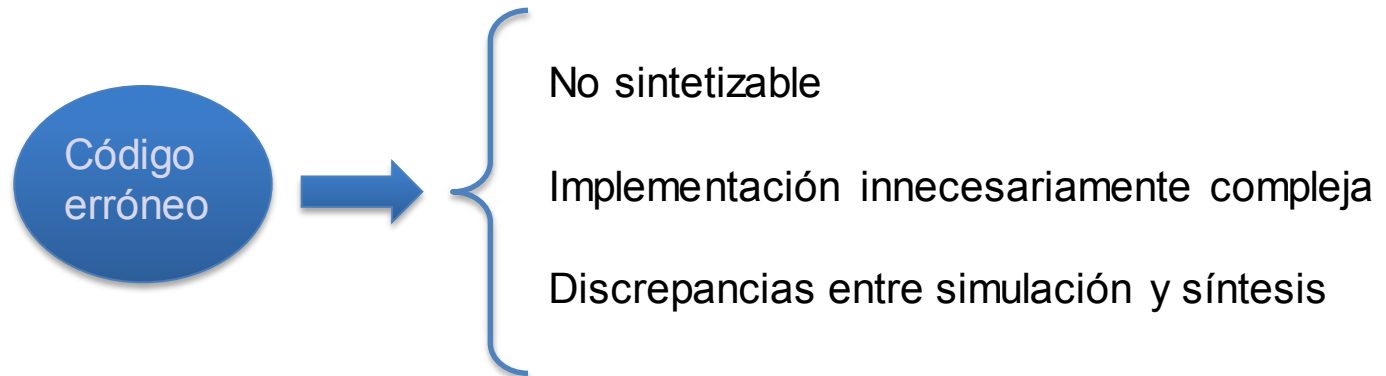
```
wire [1:0] sel;  
...  
case (sel)  
  2'b00: r = a + b + c;  
  2'b10: r = a - b;  
  default: r = c + 1;  
endcase
```



- Descripción de circuitos combinacionales

## Bloque Always para diseño de circuitos combinacionales

- Mejores prácticas en la codificación



- **Descripción de circuitos combinacionales**

**Errores comunes al describir circuitos combinacionales**

- Asignación de variables en más de un bloque always
- Lista de sensibilidad incompleta
- Rama incompleta y asignación de salida incompleta

- Descripción de circuitos combinacionales

## Errores comunes al describir circuitos combinacionales

- Asignación de variables en más de un bloque always

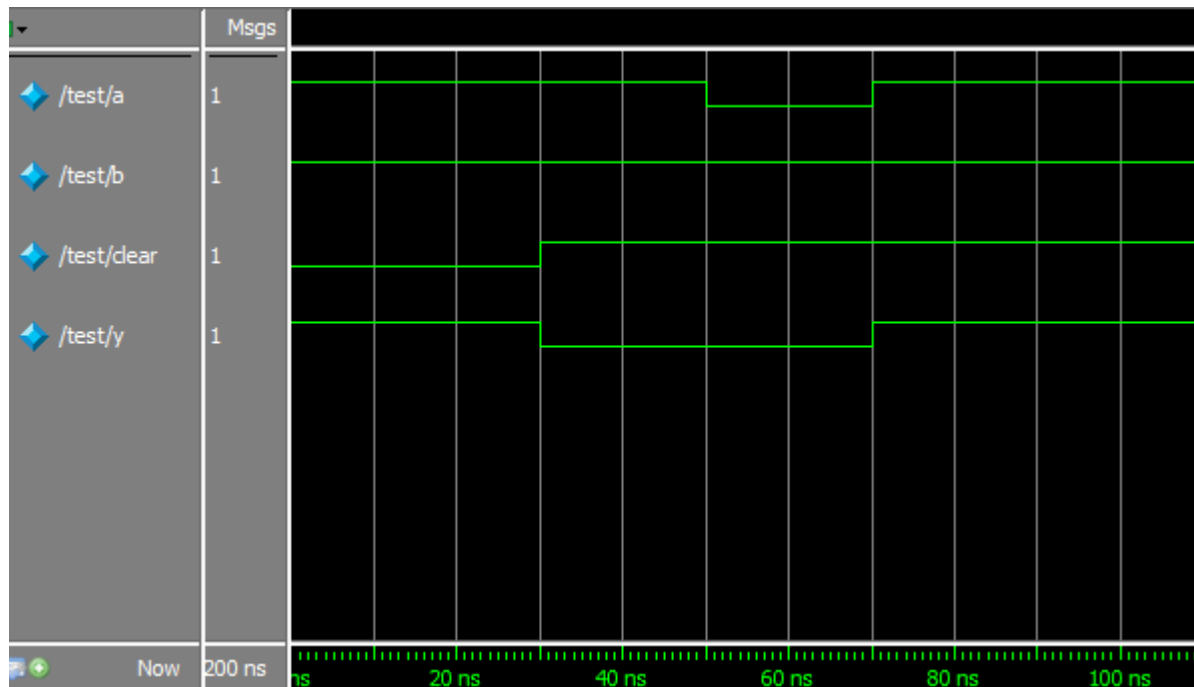
En Verilog las variables pueden asignarse en más de un bloque always (sintácticamente no existe ninguna restricción), por lo que se puede llegar a generar un código **no sintetizable**.

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

- Descripción de circuitos combinacionales

## Errores comunes al describir circuitos combinacionales

- Asignación de variables en más de un bloque always



- Descripción de circuitos combinacionales

### Errores comunes al describir circuitos combinacionales

- Asignación de variables en más de un bloque `always`

Se podría reescribir el código:

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



Código sintetizable



- Descripción de circuitos combinacionales

## Errores comunes al describir circuitos combinacionales

- Lista de sensibilidad incompleta

En un circuito combinacional la salida es función de la entrada. Cualquier cambio en la entrada debe activar al circuito, por lo tanto todas las entradas deben estar en la lista de sensibilidad.

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

Compuerta and con  
entradas a y b

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

Sintácticamente correcto. El sintetizador puede detectar el problema y corregirlo generando un warning. Si esto ocurre se habrá generado una discrepancia entre lo simulado y el hardware obtenido.

- Descripción de circuitos combinacionales

## Errores comunes al describir circuitos combinacionales

- Lista de sensibilidad incompleta

Para evitar este problema Verilog-2001 cuenta con @\* que incluye automáticamente todas las entradas relevantes.

El código anterior quedaría:

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

- **Descripción de circuitos combinacionales**

### **Errores comunes al describir circuitos combinacionales**

- **Rama incompleta y asignación de salida incompleta**

El estándar Verilog especifica que una variable mantendrá su valor anterior si no se le asigna valor dentro de un bloque always. Si ocurre esto, en la síntesis se inferirá un elemento de memoria no deseado.

Regla práctica para evitar este problema:

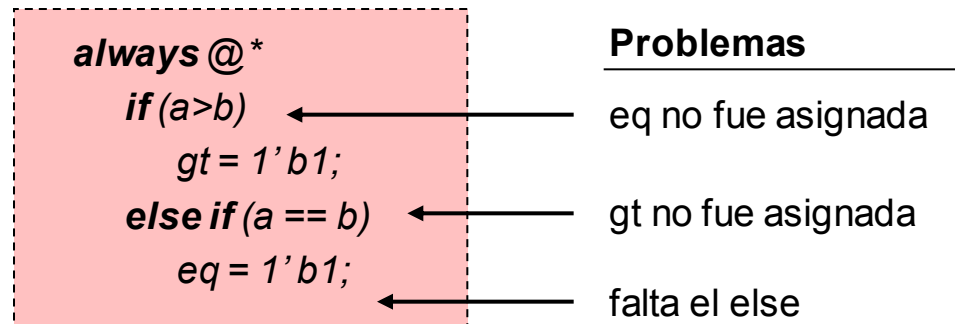
- Incluir todas las ramas de las sentencias if y case.
- Asignar un valor a cada señal de salida en cada rama.

- Descripción de circuitos combinacionales

## Errores comunes al describir circuitos combinacionales

- Rama incompleta y asignación de salida incompleta

Veamos el siguiente ejemplo:



- Descripción de circuitos combinacionales

## Errores comunes al describir circuitos combinacionales

- Rama incompleta y asignación de salida incompleta

Existen dos maneras de resolver el problema:

```
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  
    begin  
      gt = 1'b0;  
      eq = 1'b0;  
    end;
```

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

- Descripción de circuitos combinacionales

## Errores comunes al describir circuitos combinacionales

- Rama incompleta y asignación de salida incompleta

Para el caso de la sentencia case también pasa lo mismo:

```
reg [1:0] sel;  
...  
always @*  
  case (sel)  
    2'b00: y = 1'b1;  
    2'b10: y = 1'b0;  
    2'b11: y = 1'b1;  
  endcase
```

### Problema

falta evaluar la opción  
sel == 2'b01

- Descripción de circuitos combinacionales

## Errores comunes al describir circuitos combinacionales

- Rama incompleta y asignación de salida incompleta

Tres posibles manera de resolver el problema:

```
reg [1:0] sel;  
...  
always @*  
  case (sel)  
    2'b00: y = 1'b1;  
    2'b10: y = 1'b0;  
    default: y = 1'b1;  
  endcase
```

```
reg [1:0] sel;  
...  
always @*  
  case (sel)  
    2'b00: y = 1'b1;  
    2'b10: y = 1'b0;  
    2'b11: y = 1'b1;  
    default: y = 1'bx;  
  endcase
```

```
reg [1:0] sel;  
...  
always @*  
  y = 1'b0;  
  case (sel)  
    2'b00: y = 1'b1;  
    2'b10: y = 1'b0;  
    2'b11: y = 1'b1;  
  endcase
```

- Descripción de circuitos combinacionales

## Constantes

Las constantes se utilizan frecuentemente en expresiones y en los límites de los vectores.

En Verilog una constante se declara por medio de la palabra reservada **localparam**.

Por ejemplo:

```
localparam DATA_W = 16,  
            MAX_NUM = 2**DATA_W - 1;
```



- Descripción de circuitos combinacionales

## Parámetros

Los parámetros son utilizados para pasar información a los módulos que son instanciados en un diseño mayor.

Su uso hace que los módulos sean parametrizables facilitando su reutilización.

Un parámetro no puede ser modificado dentro del módulo, por lo tanto se comportan como constantes.

```
module[nombre_del_modulo]
  #(
    parameter [nombre_parametro] = [valor_por_defecto],
    ...
    [nombre_parametro] = [valor_por_defecto]
  )
  (
    [declaracion_puertos]
  );
```

- Descripción de circuitos combinacionales

## Parámetros

Ejemplo: full-adder de N bits

```
module full_adder
  #(
    parameter N = 4
  )
  (
    input wire [N-1:0] a, b,
    input wire ci,
    output wire [N-1:0] sum,
    output wire cout
  );

  wire [N+1:0] sum_aux;

  assign sum_aux = {1'b0, a, ci} + {1'b0, b, 1'b1};
  assign sum = sum_aux[N:1];
  assign cout = sum_aux[N+1];

endmodule
```

- Descripción de circuitos combinacionales

## Parámetros

Ejemplo: Testbench para el full-adder de N bits

```
module full_adder_tb;

    localparam N = 3;

    // declaración de las señales de prueba
    reg [N-1:0] a, b;
    reg ci;
    wire [N-1:0] sum;
    wire co;

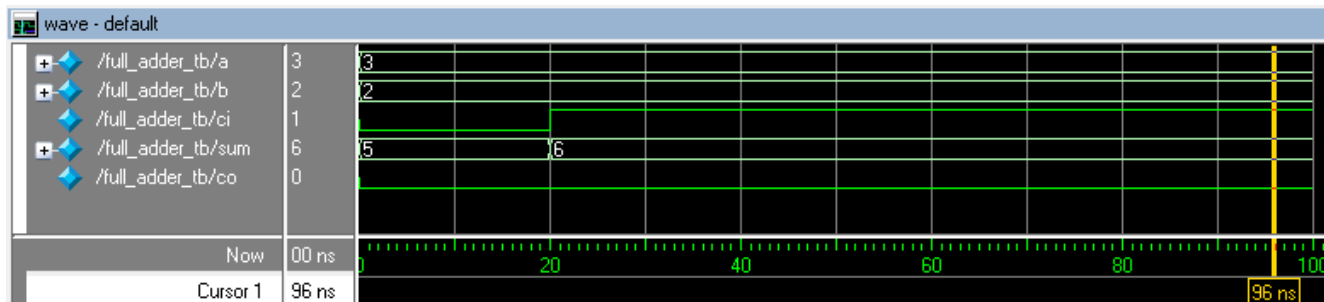
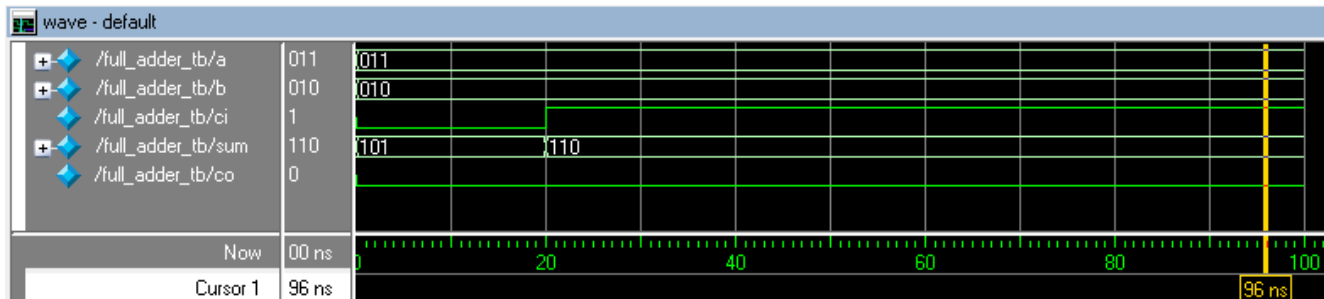
    // instancia del DUT
    full_adder #(.N(N)) DUT (.a(a), .b(b), .ci(ci), .sum(sum), .co(co));

    // creación de las señales de prueba
    initial
    begin
        a = 3'b011;
        b = 3'b010;
        ci = 1'b0;
        #20;
        ci = 1'b1;
    end
endmodule
```

- Descripción de circuitos combinacionales

## Parámetros

Ejemplo: Testbench para el full-adder de N bits



## • Descripción de circuitos secuenciales

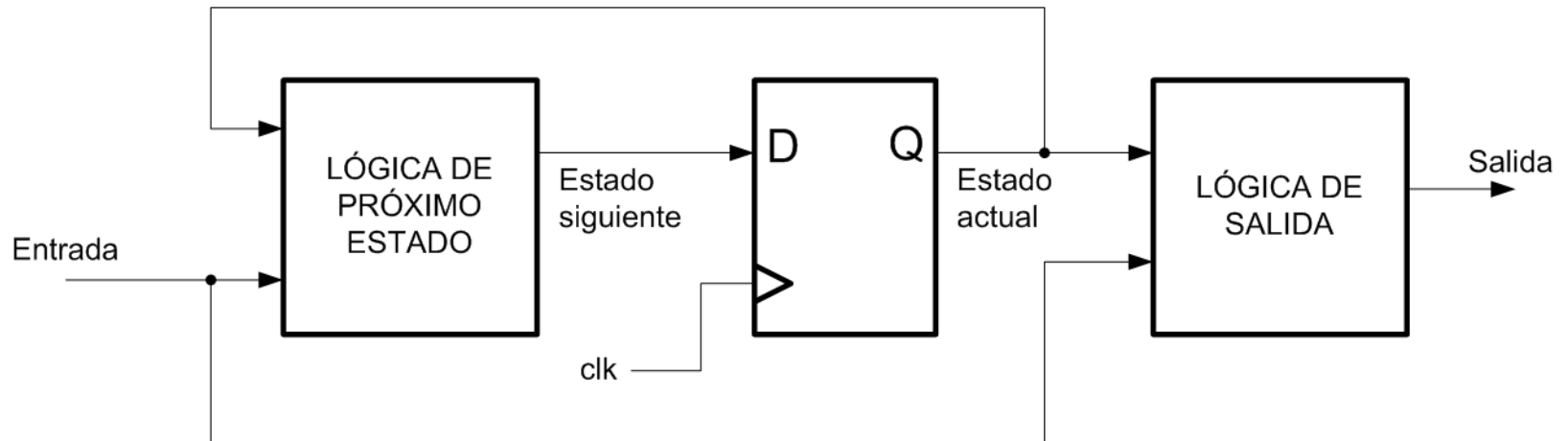
### Introducción

- Un circuito secuencial es un circuito con memoria. Esta memoria forma el estado interno del circuito.
- La salida depende de las **entradas** y del **estado interno** del circuito
- Existen circuitos secuenciales sincrónicos y asincrónicos.  
En el curso vamos a trabajar con los sincrónicos.
- En este tipo de circuitos todos los elementos de memoria son controlados por un reloj global haciendo que los datos sean muestreados y almacenados en cada flanco ascendente/descendente de reloj.
- El elemento de memoria más común en un circuito secuencial es el flip-flop D.

- Descripción de circuitos secuenciales

## Introducción

Diagrama en bloques de un sistema secuencial síncrono



- **Descripción de circuitos secuenciales**

## **Introducción**

Teniendo en cuenta la lógica de próximo estado, los circuitos secuenciales se pueden dividir en:

- Circuitos secuenciales comunes
- FSM (Finite State Machines)
- FSMD (FSM con ruta de datos)

- **Circuitos secuenciales comunes**

## **Introducción**

- Las transiciones entre estados exhiben un patrón regular (Ej: contadores y desplazadores).
- La lógica de estado siguiente se construye, principalmente, en base a componentes comunes, tales como incrementadores o desplazadores.



- **Circuitos secuenciales comunes**

### **Componentes de memoria**

Descripción en Verilog de los siguientes elementos de memoria:

- Flip-flop D
- Registro
- Register file


- Las descripciones contendrán **bloques always**.
- Para lograr que se infieran elementos de memoria se usarán asignaciones **no bloqueantes**.

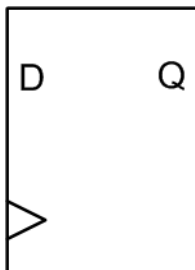
```
[nombre_de_variable] <= [expresion];
```

- Circuitos secuenciales comunes


## Flip-flop D

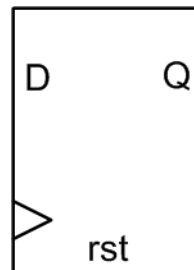
Sin reset

clk	Q*
0	Q
1	Q
	D





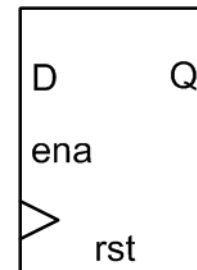
Con reset asincrónico

rst	clk	Q*
1	-	0
0	0	Q
0	1	Q
0		D



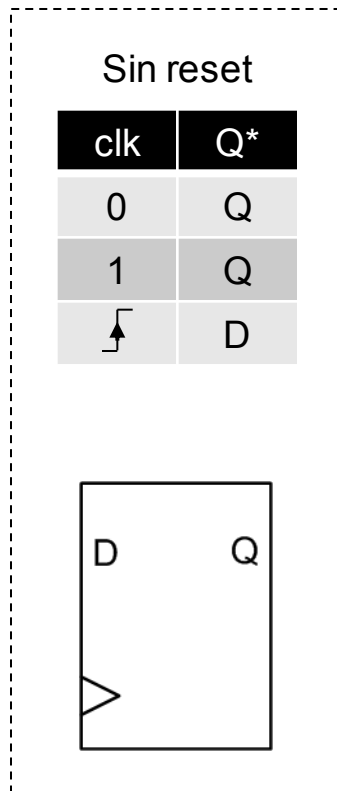
Con reset asincrónico y  
habilitación sincrónica

rst	clk	ena	Q*
1	-	-	0
0	0	-	Q
0	1	-	Q
0		0	Q
0		1	D



- Circuitos secuenciales comunes

## Flip-flop D sin reset




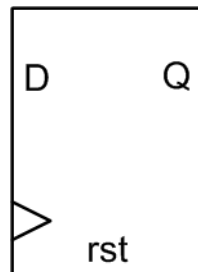
```
module ffd(  
    input wire clk, d,  
    output reg q  
);  
  
    // cuerpo del FFD  
    always @(posedge clk)  
        q <= d;  
  
endmodule
```

- Circuitos secuenciales comunes

## Flip-flop D con reset asincrónico

Con reset asincrónico

rst	clk	Q*
1	-	0
0	0	Q
0	1	Q
0		D





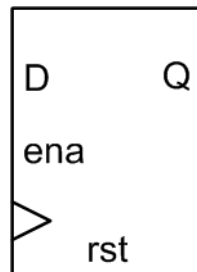
```
module ffd(  
    input wire clk, rst, d,  
    output reg q  
);  
  
    // cuerpo del FFD  
    always @(posedge clk, posedge rst)  
    begin  
        if (rst)  
            q <= 1'b0;  
        else  
            q <= d;  
    endmodule
```

- Circuitos secuenciales comunes

## Flip-flop D con reset asincrónico y habilitación sincrónica

Con reset asincrónico y  
habilitación sincrónica

rst	clk	ena	Q*
1	-	-	0
0	0	-	Q
0	1	-	Q
0		0	Q
0		1	D



```
module ffd(  
    input wire clk, rst, ena, d,  
    output reg q  
);  
  
    // cuerpo del FFD  
    always @(posedge clk, posedge rst)  
    begin  
        if (rst)  
            q <= 1'b0;  
        else if (ena)  
            q <= d;  
    endmodule
```

- Circuitos secuenciales comunes

## Registro

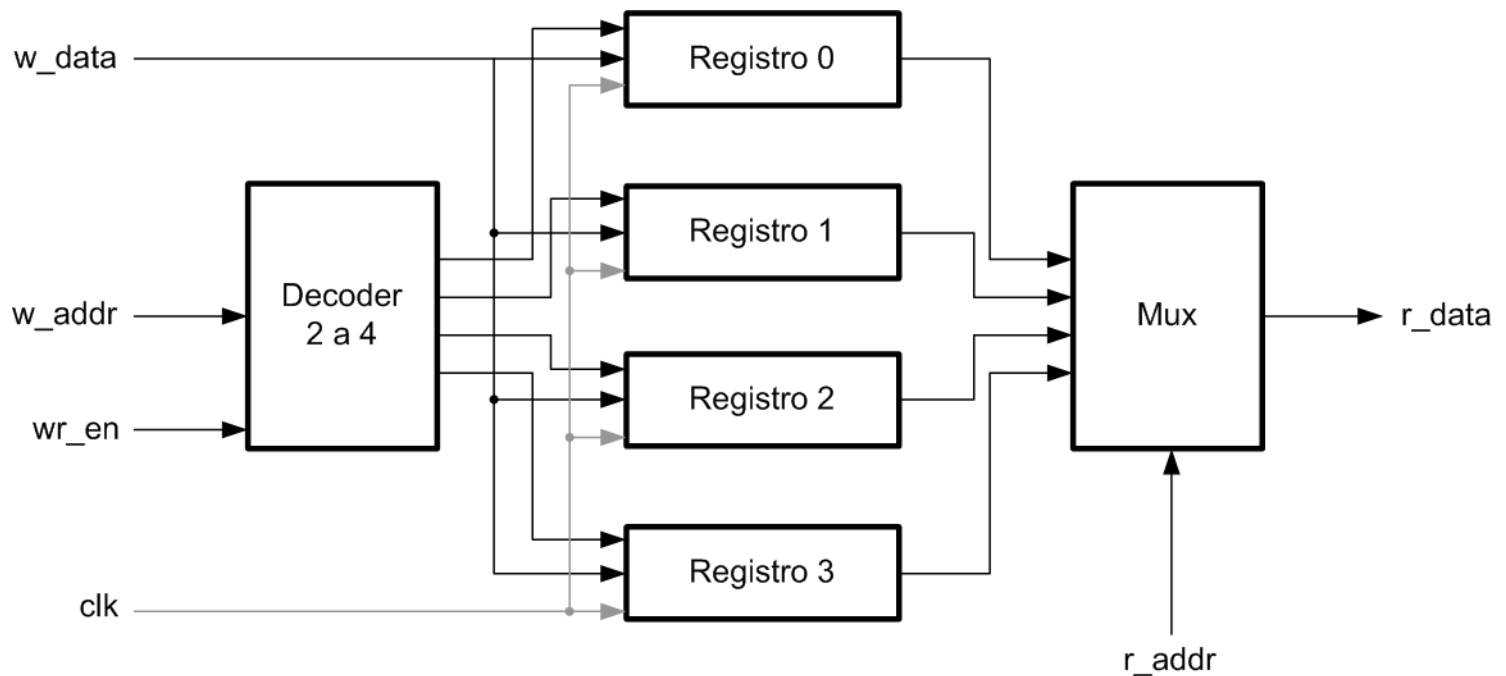
Es un arreglo de flip-flops D, controlados por el mismo clock y reset

```
module ffd(  
    input wire clk, rst, ena,  
    input wire [7:0] d;  
    output reg [7:0] q  
);  
  
// cuerpo del FFD  
always @(posedge clk, posedge rst)  
    if (rst)  
        q <= 0;  
    else if (ena)  
        q <= d;  
  
endmodule
```

- Circuitos secuenciales comunes

## Register file

Es un agrupamiento de registros con un puerto de entrada y uno o más puertos de salida



- Circuitos secuenciales comunes

## Register file

```

module reg_file
  #(
    parameter N = 8, // número de bits
               W = 2 // cant. de bits de la dirección
  )
  (
    input wire clk, wr_en,
    input wire [N-1:0] w_data,
    input wire [W-1:0] w_addr, r_addr,
    output wire [N-1:0] r_data
  );

  reg [N-1:0] array_reg [2**W-1:0];

  // operación de escritura
  always @(posedge clk)
    if (wr_en)
      array_reg[w_addr] <= w_data;
  // operación de lectura
  assign r_data <= array_reg[r_addr];

endmodule

```



- Circuitos secuenciales comunes

## Error común al describir un secuencial síncrono

Se busca crear un registro de desplazamiento de 3 posiciones

### CODIGO ERRONEO

```
module shift_register
(
    inputwire clk,
    inputwire a,
    outputreg d
);

reg b;
reg c;

always @(posedge clk)
begin
    b = a;
    c = b;
    d = c;
end;

endmodule
```

- Circuitos secuenciales comunes

## Error común al describir un secuencial síncronico

Se busca crear un registro de desplazamiento de 3 posiciones

### CODIGO ERRONEO

```
=====
*                               HDL Synthesis                               *
=====

Performing bidirectional port resolution...

Synthesizing Unit <shift_register>.
  Related source file is "../codigo_erroneo.v".
⚠️WARNING:Xst:646 - Signal <c> is assigned but never used. This unconnected
⚠️WARNING:Xst:646 - Signal <b> is assigned but never used. This unconnected
  Found 1-bit register for signal <d>.
  Summary:
    inferred    1 D-type flip-flop(s).
Unit <shift_register> synthesized.
```

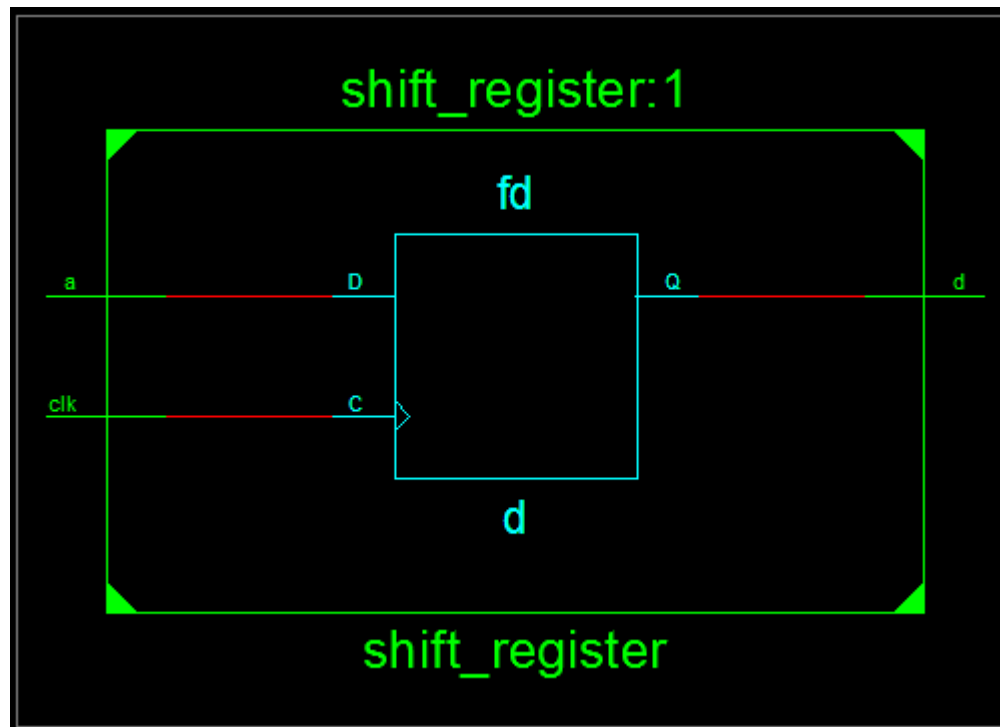
*Reporte emitido por el ISE (Synthesis Report)*

- Circuitos secuenciales comunes

## Error común al describir un secuencial síncrono

Se busca crear un registro de desplazamiento de 3 posiciones

### CODIGO ERRONEO



Se infiere un solo flip-flop D!!!

- Circuitos secuenciales comunes

## Error común al describir un secuencial síncrono

Se busca crear un registro de desplazamiento de 3 posiciones

### CODIGO CORRECTO

```
module shift_register
(
    inputwire clk,
    inputwire a,
    outputreg d
);

reg b;
reg c;

always @(posedge clk)
begin
    b <= a;
    c <= b;
    d <= c;
end;

endmodule
```

- Circuitos secuenciales comunes

## Error común al describir un secuencial síncronico

Se busca crear un registro de desplazamiento de 3 posiciones

### CODIGO CORRECTO

```
=====
*                               HDL Synthesis                               *
=====

Performing bidirectional port resolution...

Synthesizing Unit <shift_register>.
  Related source file is "../codigo_erroneo.v".
  Found 1-bit register for signal <d>.
  Found 1-bit register for signal <b>.
  Found 1-bit register for signal <c>.
  Summary:
    inferred   3 D-type flip-flop(s).
Unit <shift_register> synthesized.
```

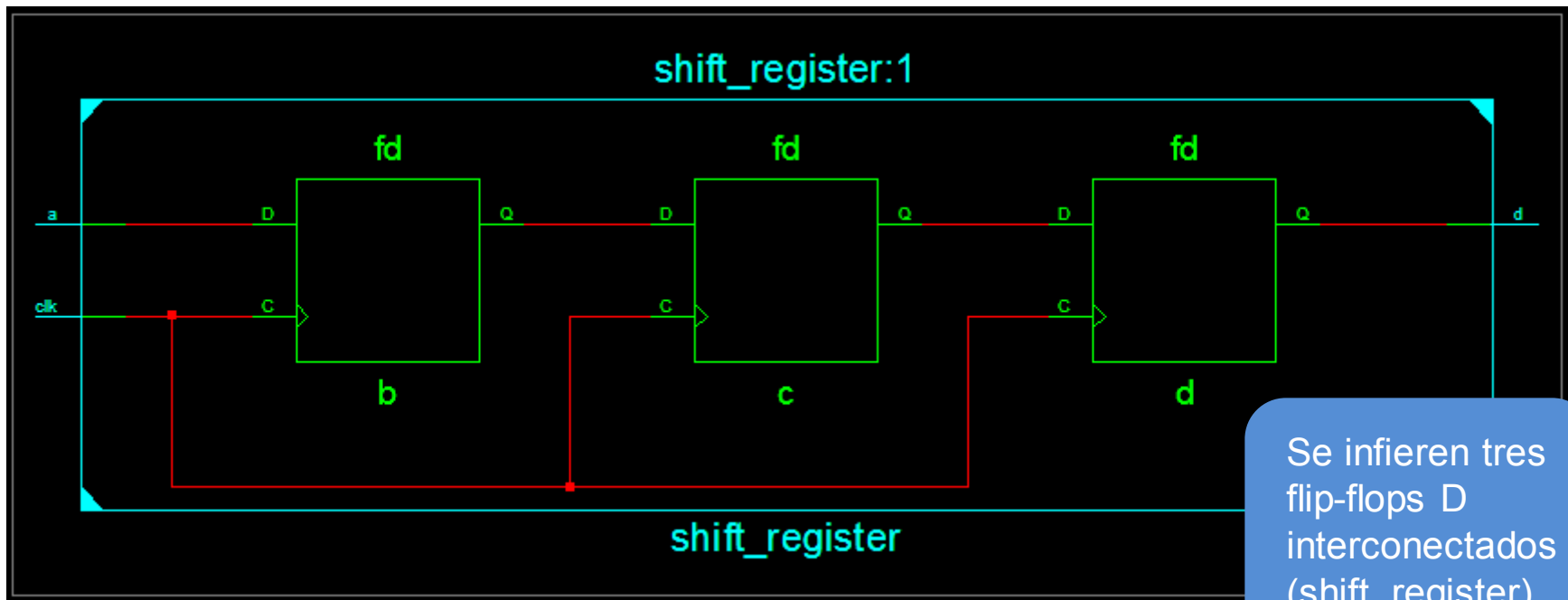
*Reporte emitido por el ISE (Synthesis Report)*

- Circuitos secuenciales comunes

## Error común al describir un secuencial síncrono

Se busca crear un registro de desplazamiento de 3 posiciones

### CODIGO CORRECTO



Se infieren tres flip-flops D interconectados (shift\_register)

Vista del RTL Schematic (Synthesis - XST)

- Banco de pruebas para circuitos secuenciales

## Dispositivo a probar (contador binario universal)

```

module univ_bin_counter
#(
  parameter N = 8
)
(
  inputwire clk, reset,
  inputwire syn_clr, load, en, up,
  inputwire [N-1:0] d,
  outputwire max_tick, min_tick,
  outputwire [N-1:0] q
);

// declaración de señales
reg [N-1:0] r_reg, r_next;

// cuerpo del módulo
// registro
always @(posedge clk, posedge reset)
  if (reset)
    r_reg <= 0;
  else
    r_reg <= r_next;

```

Parte 1

```

// lógica de próximo estado
always @*
  if (syn_clr)
    r_next = 0;
  else if (load)
    r_next = d;
  else if (en & up)
    r_next = r_reg + 1;
  else if (en & ~up)
    r_next = r_reg - 1;
  else
    r_next = r_reg;

// lógica de salida
assign q = r_reg;
assign max_tick = (r_reg == 2**N-1) ? 1'b1 : 1'b0;
assign min_tick = (r_reg == 0) ? 1'b1 : 1'b0;

```

endmodule

Parte 2

- Banco de pruebas para circuitos secuenciales

## Banco de pruebas para el contador binario universal

```
`timescale 1 ns / 10 ps

module bin_counter_tb;
    // declaracion de puertos
    localparam T = 20; // clock period
    reg clk, reset;
    reg syn_clr, load, en, up;
    reg [2:0] d;
    wire max_tick, min_tick;
    wire [2:0] q;

    // Instanciacion del contador a probar
    univ_bin_counter#(N(3)) DUT (
        .clk(clk), .reset(reset),
        .syn_clr(syn_clr), .load(load), .en(en),
        .up(up), .d(d), .max_tick(max_tick),
        .min_tick(min_tick), .q(q)
    );
```

Parte 1

```
// clock de 20 ns, corriendo por siempre
always
begin
    clk = 1'b1;
    #(T/2);
    clk = 1'b0;
    #(T/2);
end

// reset en el primer ciclo
initial
begin
    reset = 1'b1;
    #(T/2);
    reset = 1'b0;
end
```

Parte 2



- Banco de pruebas para circuitos secuenciales

## Dispositivo a probar (contador binario universal)

```
// otros estímulos
initial
begin
    syn_clr = 1'b0;
    load = 1'b0;
    en = 1'b0;
    up = 1'b1; // count up
    d = 3'b000;
    // esperar al flanco desc. del reset
    @(negedge reset);
    // esperar por un ciclo de reloj
    @(negedge clk);
    // ==== prueba de load ====
    load = 1'b1;
    d = 3'b011;
    // esperar por un clock
    @(negedge clk);
    load = 1'b0;
```

Parte 3

```
repeat(2) @(negedge clk);
// ==== prueba de syn_clear ====
syn_clr = 1'b1; // poner a 1 clear
@(negedge clk);
syn_clr = 1'b0;
// === prueba de cuenta y pausa ===
en = 1'b1; // cuenta
up = 1'b1; // ascendente
repeat(10) @(negedge clk);
en = 1'b0; // pausa
repeat(2) @(negedge clk);
en = 1'b1; // reinicio
repeat(2) @(negedge clk);
// ==== prueba de cuenta desc. ====
up = 1'b0;
repeat(10) @(negedge clk);
```

Parte 4

- Banco de pruebas para circuitos secuenciales

## Dispositivo a probar (contador binario universal)

```
// continuar hasta que q=2
wait(q == 2);
@(negedge clk);
up = 1'b1; // cambio a cuenta asc.
// continuar hasta que min_tick = 1
@(negedge clk);
wait(min_tick);
@(negedge clk);
up = 1'b0;
// ==== absolute delay ====
#(4*T); // wait for 80 ns
en = 1'b0; // pause
#(4+T); // wait for 80 ns
// ==== stop simulation ====
$stop;
end
endmodule
```

- **Sentencia generate**

- El **verilog 1995** tenía limitaciones en la generación de diseños escalables
- El **verilog 2001** introdujo la sentencia **generate** que permite la generación de múltiples instancias de módulos y primitivas, como así también la generación de múltiples ocurrencias de variables, nets, tasks, funciones, asignaciones continuas, procedimientos iniciales y procedimientos always
- La generación puede ser controlada por sentencias if-else y case

## • Sentencia generate

### Template

```
generate { generate_item } endgenerate
```

```
generate_item_or_null := generate_item | ;
```

```
generate_item :=
```

```
    generate_conditional_statement |
```

```
    generate_case_statement |
```

```
    generate_loop_statement |
```

```
    generate_block |
```

```
    module_or_generate_item
```

```
generate_conditional_statement :=
```

```
    if (constant_expression) generate_item_or_null
```

```
    [else generate_item_or_null]
```

```
generate_case_statement := case (constant_expression)
```

```
    genvar_case_item { genvar_case_item } endcase
```

```
genvar_case_item := constant_expression { , constant_expression } ;
```

```
generate_item:or_null | default [ : ] generate_item_or_null
```

- Sentencia generate

### Template

```
generate { generate_item } endgenerate
```

```
generate_loop_statement :=
```

```
    for (genvar_assignment ; constant_expression ; genvar_assignment )
```

```
        begin : generate_block_identifier { generate_item } end
```

```
genvar_assignment := genvar_identifier = constant_expression
```

```
generate_block := begin [ : generate_block_identifier ] { generate_item } end
```

```
genvar_declaration := genvar list_of_genvar_identifiers ;
```

```
list_of_genvar_identifiers := genvar_identifier { , genvar_identifier }
```

- **Sentencia generate**

### Template

```
genvar i;  
generate  
    for (genvar_assignment ; constant_expression ; genvar_assignment )  
        begin : generate_block_identifier  
            { generate_item}  
        end  
endgenerate
```

```
genvar i;  
generate  
    for (i=0 ; i<5; i=i+1)  
        begin : bloque_de_prueba  
            ...  
        end  
endgenerate
```

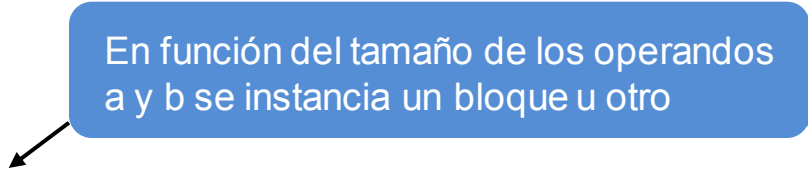
- Sentencia generate

### Template

```
generate
  if (constant_expression)
    generate_item_or_null
  else
    generate_item_or_null
endgenerate
```

```
generate
  if (N<8)
    bloque1 #(N) bloque1_inst (a, b, resultado);
  else
    bloque2 #(N) bloque2_inst (a, b, resultado);
endgenerate
```

En función del tamaño de los operandos a y b se instancia un bloque u otro



- Sentencia generate

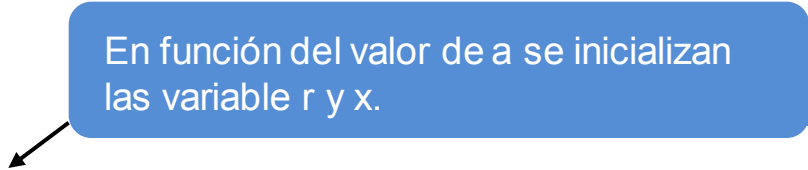
### Template

```
generate  
  case (constant_expression)  
    item1: ...  
    item2: ...  
    ...  
    default: ...
```

```
endgenerate
```

```
generate  
  case (a)  
    1: initial r = 2;  
    2: initial begin  
        x = 1; r = 4;  
    end  
  default:  
    initial begin  
        x = 0; r = p;  
    end  
  endcase  
endgenerate
```

En función del valor de a se inicializan las variable r y x.





## • Sentencia generate

Ejemplo: full-adder de N bits (utilizando primitivas)

```

module full_adder_generate_1
  #(parameter N = 4)
  (
    input [N-1:0] a, b,
    input ci,
    output [N-1:0] sum,
    output co
  );

  wire [N:0] aux; // señal auxiliar
  genvar i;
  generate
    for(i=0; i<N; i=i+1)
      begin: suma1bit
        wire n1,n2,n3; // conexiones internas

        xor g1 (n1, a[i], b[i]);
        xor g2 (sum[i], n1, aux[i]);
        and g3 (n2, a[i], b[i]);
        and g4 (n3, n1, aux[i]);
        or g5 (aux[i+1], n2, n3);
      end
    endgenerate

    assign aux[0] = ci;
    assign co = aux[N];
endmodule

```

## • Sentencia generate

Ejemplo: full-adder de N bits (utilizando asignación continua)

```

module full_adder_generate_2
  #(parameter N = 4)
  (
    input [N-1:0] a, b,
    input ci,
    output [N-1:0] sum,
    output co
  );

  wire [N:0] aux; // señal auxiliar
  genvar i;
  generate
    for(i=0; i<N; i=i+1)
      begin: suma1bit
        wire n1,n2,n3; // conexiones internas

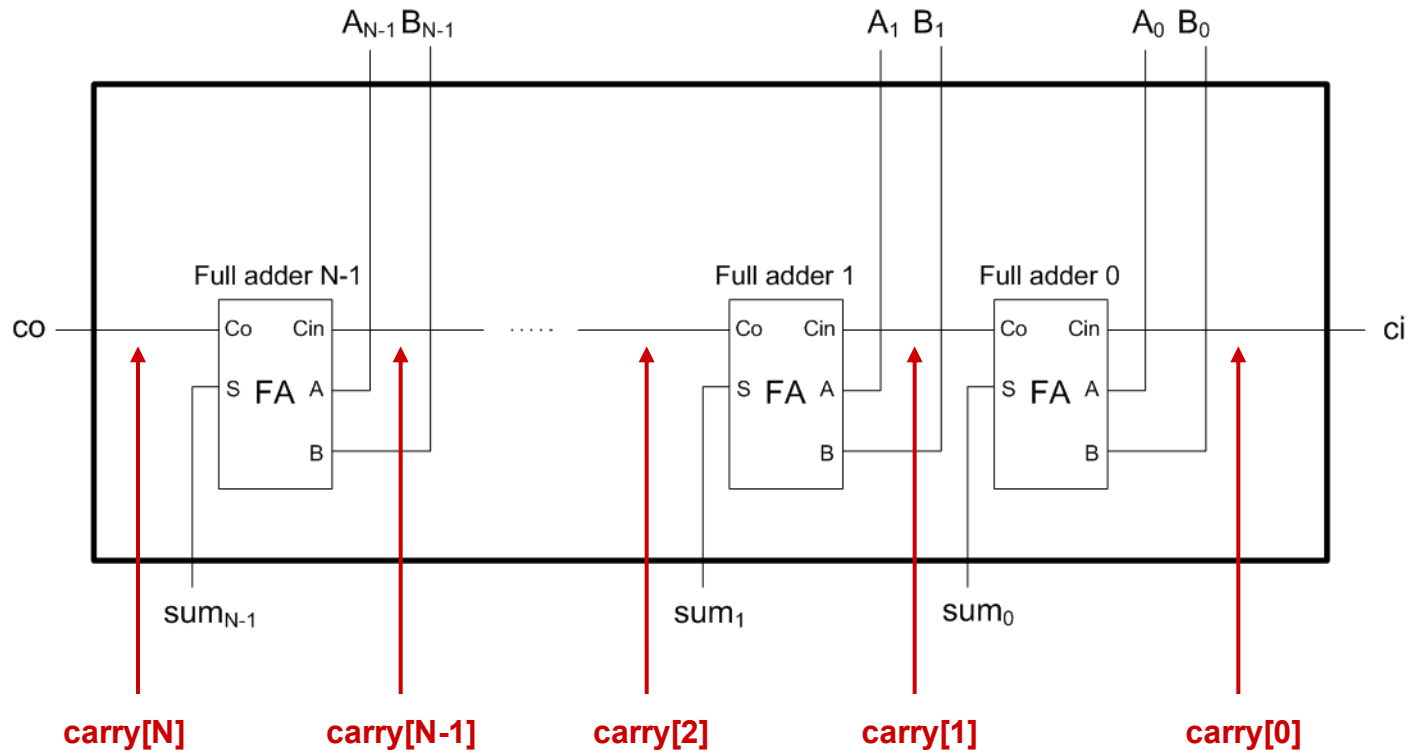
        assign n1 = a[i] ^ b[i];
        assign sum[i] = n1 ^ aux[i];
        assign n2 = a[i] & b[i];
        assign n3 = n1 & aux[i];
        assign aux[i+1] = n2 | n3;
      end
    endgenerate

    assign aux[0] = ci;
    assign co = aux[N];
  endmodule

```

- Sentencia generate

Ejemplo: full-adder de N bits (utilizando un full-adder de 1 bit)



- Sentencia generate

Ejemplo: full-adder de N bits (utilizando un full-adder de 1 bit)

```
module full_adder_generate_3
#(
  parameter N = 4
)
(
  input [N-1:0] a,b,
  input ci,
  output co,
  output [N-1:0] sum );

  wire [N:0] carry;

  assign carry[0] = ci;
  assign co = carry[N];

  genvar i;
  generate for ( i = 0; i < N; i = i+1 )
    begin : full_adder_1b
      FA fa( op1[i], op2[i], carry[i], carry[i+1] );
    end
  endgenerate

endmodule
```

- **Funciones y tasks**

- Verilog permite crear subprogramas usando funciones y tasks.
- Son usados para facilitar la lectura y reusabilidad del código.
- Las funciones no pueden contener operadores de control de eventos ni retardos (como los usados en la descripción de circuitos secuenciales).
- Los tasks son más generales.

## • Funciones

Las funciones son declaradas dentro de un módulo con las palabras clave **function** y **endfunction**.

Dentro de una función se puede llamar a otra, pero no a un task.

Para crear una función se deben cumplir las siguientes condiciones:

- No existen estructuras de control de evento, timing, ni retardo.
- Sólo se retorna un valor
- Existe al menos un argumento de entrada
- No existen argumentos de salida (output) ni inout
- No existen asignaciones no bloqueantes

En pocas palabras las funciones deben implementar sólo lógica combinacional

- Funciones

### Template

```
function [tipo_del_resultado] [nombre_funcion] ([argumentos]);  
    [variables];  
begin  
    [sentencia1];  
    [sentencia2];  
    ...  
    nombre_funcion = expresion;  
end  
endfunction
```

- Funciones

### Ejemplo 1: Cálculo del log2

```
function integer log2 (input integer n);  
    integer i;  
    begin  
        log2 = 1;  
        for (i=0; 2**i < n; i = i + 1)  
            log2 = i + 1;  
    end  
endfunction
```



- Funciones

### Ejemplo 2: Reversión de bits

```
function [N-1:0] reverse_bits (input [N-1:0] data_in);  
    integer i;  
    begin  
        for (i=0; i < N; i = i + 1)  
            reverse_bits[N-i-1] = data_in[i];  
    end  
endfunction
```

- Funciones

### Ejemplo 2: Reversión de bits (uso de la función)

```
module prueba_funcion
  #(parameter N = 8)
  (
    input[N-1:0] x_in, // palabra de entrada
    output reg[N-1:0] rev_x // palabra de salida
  );

  function[N-1:0] reverse_bits (input[N-1:0] data_in);
    integer i;
    begin
      for (i=0; i < N; i = i + 1)
        reverse_bits[N-i-1] = data_in[i];
      end
    endfunction

  always @ (x_in)
    rev_x = reverse_bits(x_in); // funcion llamada

endmodule
```

## • Tasks

Un task es como un procedure que provee la habilidad de ejecutar pedazos de código comunes desde diferentes lugares dentro de un módulo. Son declarados entre las palabras clave **task** y **endtask**

- Son definidos en el módulo en el cual son usados
- Pueden incluir retardos de tiempo, tales como posedge, negedge, # delay y wait.
- Pueden llamar otros tasks y funciones
- Pueden tener cualquier número de entradas y de salidas (argumentos de modo input, output y inout)
- Pueden ser utilizados para modelar tanto circuitos combinacionales como secuenciales
- Debe ser llamado como una sentencia (no puede ser usado dentro de una expresión, como puede serlo una función)

- Tasks

### Template

```
task task_id;  
    [declaraciones];  
begin  
    [sentencia1];  
    [sentencia2];  
    ...  
    [sentenciaN];  
end  
endtask
```

- Tasks

### Ejemplo 1: Reversión de bits

```
task reverse_bits;  
  input [N-1:0] din;  
  output [N-1:0] dout;  
  
  integer k;  
  
  begin  
    for (k=0; k < N; k = k + 1)  
      dout[N-k-1] = din[k];  
    end  
  endtask
```

- Tasks

### Ejemplo 1: Reversión de bits (uso del task)

```
module prueba_task
  #(parameter N = 8)
  (
    input [N-1:0] data_in,
    output reg [N-1:0] data_out
  );

  task reverse_bits;
    input [N-1:0] din;
    output [N-1:0] dout;

    integer k;

    begin
      for (k=0; k < N; k = k + 1)
        dout[N-k-1] = din[k];
      end
    endtask

    always @ (data_in)
      reverse_bits(data_in,data_out); // llamada del task

endmodule
```

## • System Tasks

Verilog provee también system tasks.

Los nombres de estos tasks van precedidos del signo \$

**\$display**("<formato>", exp1, exp2, ...) Emite texto formateado cada vez que se ejecuta.

formatos: %b %B → binario

%c %C → carácter

%d %D → decimal

%g %G → punto flotante (formato general)

%h %H → hexadecimal

%o %O → octal

%s %S → string

%t %T → tiempo de simulación (la expresión es \$time)

%c %C → carácter

## • System Tasks

- \$write** → Igual que \$display, salvo que no existe la inclusión del newline.
- \$monitor** → Igual que \$display, salvo que se dispara cada vez que cambia alguno de sus argumentos.
- \$time, \$stime, \$realtime** → Devuelven el tiempo actual de simulación como un entero de 64 bits, un entero de 32 bits y un número real, respectivamente
- \$reset** → Resetea la simulación (vuelve a cero)
- \$stop** → Detiene el simulador y lo pone en modo interactivo
- \$finish** → Sale del simulador
- \$random** → Genera un entero aleatorio cada vez que es llamada
- \$dumpfile** → Crea un archivo en el que se van a volcar variables
- \$dumpvar** → Vuelca sobre el archivo creado todas las variables del diseño



## • Directivas de compilación

- Son usadas para controlar la compilación de un código Verilog.
- Se indican con el carácter `
- Una directiva es efectiva desde el punto en que es declarada hasta el punto en que otra directiva la sobrescribe, aún entre varios archivos
- Puede aparecer en cualquier parte del código, pero es recomendable que se encuentre fuera de la descripción del módulo
- Algunas directiva son:

**`include**

**`define**

**`undef**

**`ifdef**

**`timescale**

- **Directivas de compilación**

- **``include`**

Permite incluir el código de un archivo en otro durante la compilación.

La compilación procede como si el código del archivo incluido apareciese en el lugar del comando ``include`.

Se puede usar esta directiva para incluir definiciones globales, o comúnmente utilizadas, y tasks.

- **Directivas de compilación**

- **`define**

Es usada para definir macros (son definidas normalmente en archivos verilog nombrados como xxxx.vh)

Puede ser usada a través de múltiples archivos

- **`undef**

Permite remover la definición de macros creadas por la directiva `define

- **Directivas de compilación**

- **``ifdef`**

Permite incluir, de acuerdo a una condición, líneas de código durante la compilación.

Verifica que una macro haya sido definida, y si lo está, se compila el código que sigue. En caso contrario se compila el código que sigue a la directiva ``else`.

La directiva ``endif` indica el final del código condicional.

- Directivas de compilación

- Ejemplo

```
module ifdef ();  
  initial begin  
    `ifdef FIRST  
      $display("First code is compiled");  
    `else  
      `ifdef SECOND  
        $display("Second code is compiled");  
      `else  
        $display("Default code is compiled");  
      `endif  
    `endif  
    $finish;  
  end  
endmodule
```

- Directivas de compilación

- Ejemplo

```
`define SECOND

module ifdef ();
  initial begin
    `ifdef FIRST
      $display("First code is compiled");
    `else
      `ifdef SECOND
        $display("Second code is compiled");
      `else
        $display("Default code is compiled");
      `endif
    `endif
  $finish;
end
endmodule
```

Console
ISim P.20131013 (signature 0x8ef4fb42) This is a Full version of ISim.
# run 1000 ns Simulator is doing circuit initialization process.
<b>Second code is compiled</b>
Finished circuit initialization process.
ISim>

**FIN**