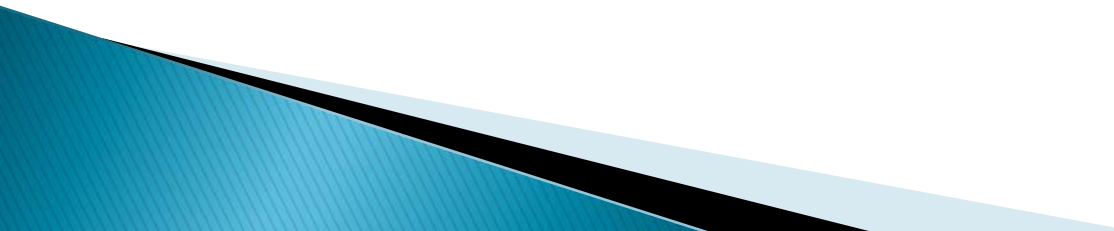


Lenguajes de Descripción de Hardware

Verilog

Contenido

- ▶ HDLs
 - ▶ Verilog
 - Módulos
 - Interfaz
 - Variables
 - Comportamiento
 - ▶ Flujo de Diseño
 - ▶ Niveles de Abstracción
 - Descripción Estructural
 - Descripción Algorítmica
 - ▶ Test Benches
- 

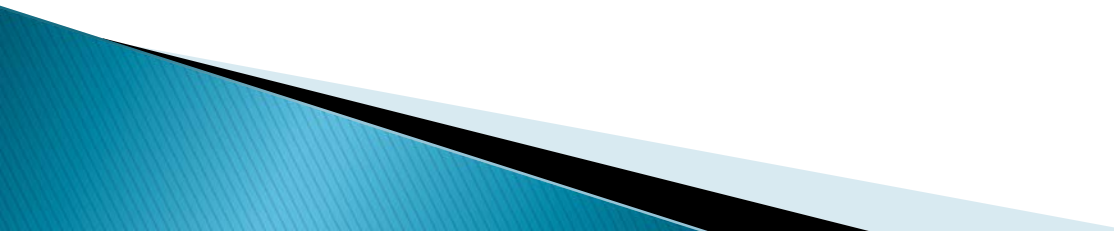
Objetivos de la Clase

- ▶ Conocer que son los Lenguajes de Descripción de Hardware.
- ▶ Aprender la sintaxis básica de Verilog y su metodología de desarrollo.
- ▶ Describir Circuitos Combinacionales en Verilog.
- ▶ Realizar primera etapa de testing a un módulo:
 - Test Benches
- ▶ Primer Trabajo Práctico: ALU

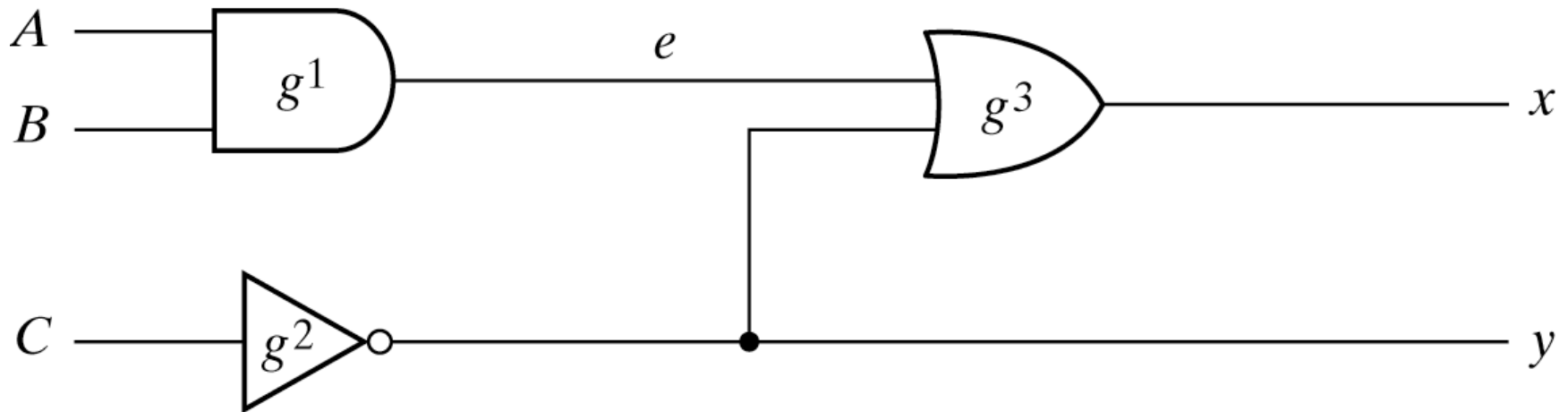
HDLs



HDLs

- ▶ Languages de Descripción de Hardware
 - ▶ Inicialmente creados para documentar y simular circuitos.
 - ▶ El código es interpretado por un Sintetizador.
 - ▶ Mas populares: **Verilog**, VHDL.
- 

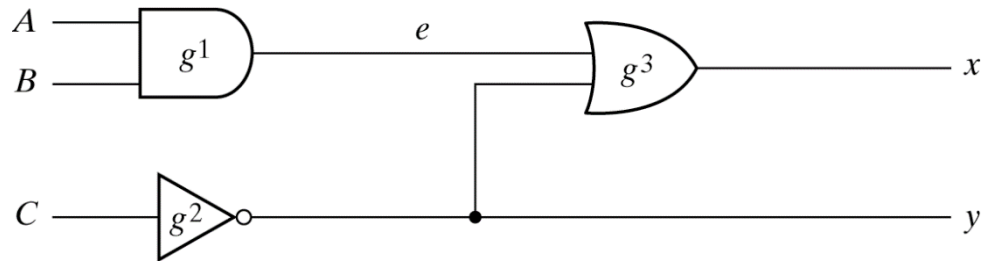
Ejemplo Modulo Simple



Verilog



Ejemplo Modulo Simple



```
module modulo_simple(A,B,C,x,y);  
    input A,B,C;  
    output x,y;  
    wire e;  
    assign e = A & B;  
    assign y = ~ C;  
    assign x = e | y;  
endmodule
```

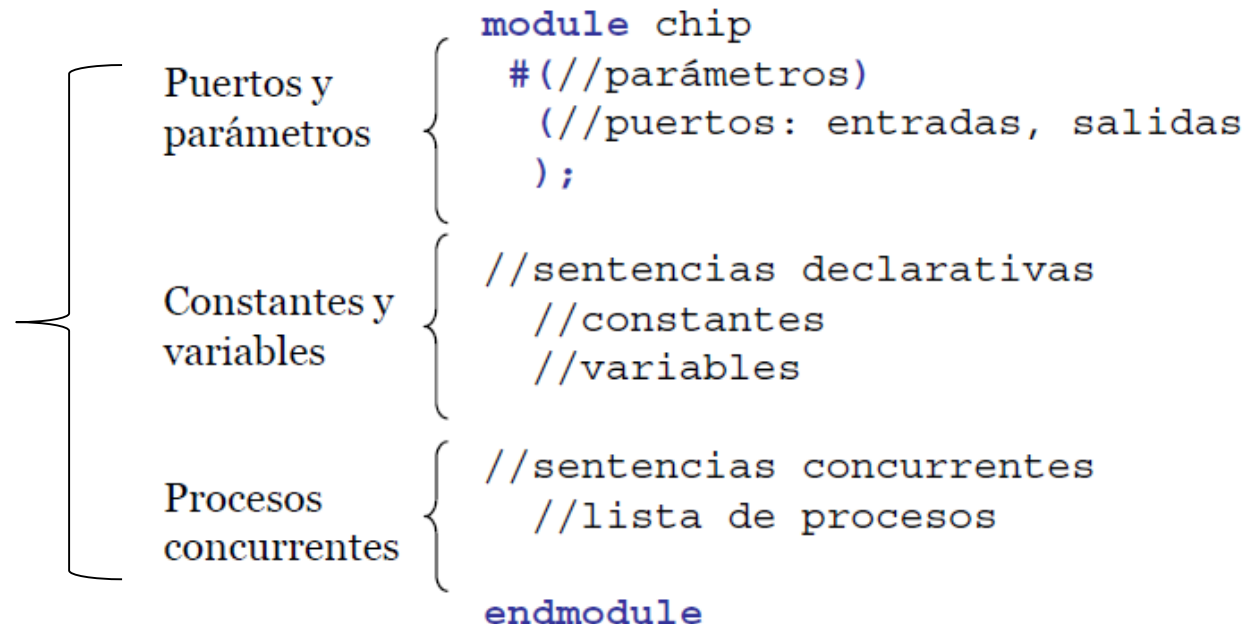

Diseño de caja negra

- ▶ Según el flujo de diseño, todo hardware se especifica primero como una «caja negra» que define sus entradas y salidas. En verilog esta caja negra se denomina **module** (módulo).



El módulo

- ▶ Tiene un nombre, puertos de I/O y parámetros de configuración (la interfaz externa);
- ▶ Declaración de Constantes y variables;
- ▶ Sentencias y/o procesos concurrentes (funcionalidad);



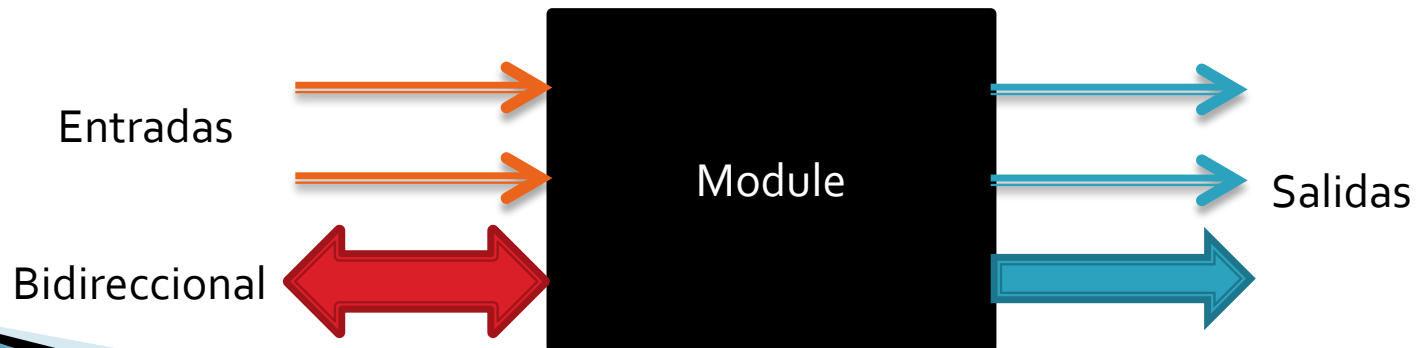
Verilog



Modulos: Interfaz

Interfaz: Puertos

- ▶ Las conexiones externas pueden ser puertos de entrada, de salida o bidireccionales
- ▶ El tipo de puerto determina la dirección de los datos:
 - A través de los puertos de entrada entran datos y señales, es decir, se leen. (no se pueden escribir). Se especifican como **input**
 - A través de los puertos de salida se envían datos y señales, es decir, se escriben (pueden leerse!). Se especifican como **output**
 - A través de los puertos bidireccionales se envían y reciben datos y señales. Se especifican como **inout**



Interfaz: Parámetros



```
module megachip
  #(parameter fmax=100)
  (input entrada,
   output salida)
  ...
endmodule
```

Con **parameter** se definen los parámetros del módulo.

Con **input** y **output** declaramos los puertos de nuestro módulo.

Interfaz: Parámetros

- ▶ Los parametros son constantes.

```
parameter msb = 7;
```

```
parameter e = 25, f = 9;
```

```
parameter average_delay = (r + f) / 2;
```

```
parameter byte_size = 8, byte_mask =  
    byte_size - 1;
```

Verilog



Modulos: Variables

Variables

- ▶ Las variables se llaman registros. Los registros a su vez pueden poseer distintos tipos de datos: **reg**, **integer**, **real**, **time** y otras.
- ▶ Las variables se utilizan para almacenar valores, lo que no siempre implica la síntesis de memoria en la implementación de hardware
- ▶ Ejemplo:

```
reg unRegistro; //1-bit reg  
integer a; // 32 bit integer
```


Variables

- ▶ Los números por defecto son enteros de 32 bits (base 10)
- ▶ Pueden especificarse otras bases y longitudes:

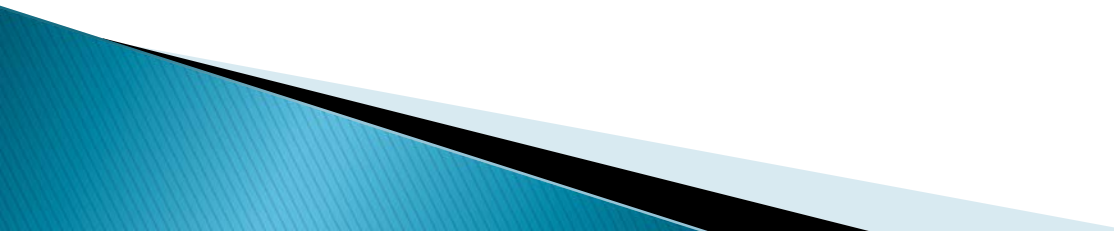
number of bits	'	radix	value
----------------	---	-------	-------

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	000000000000000000000000000011010	extended to 32 bits
'hee	000000000000000000000000000011101110	extended to 32 bits
1	000000000000000000000000000000000001	extended to 32 bits
-1	111111111111111111111111111111111111	extended to 32 bits

Table 2-1 Radix Specifiers

Radix Mark	Radix
'b 'B	Binary
'd 'D	Decimal (default)
'h 'H	Hexadecimal
'o 'O	Octal

Variables: Lógica de 4 estados

- ▶ El 0 y 1 lógico no son suficientes para representar todos los estados de un sistema digital. Verilog tiene una lógica de 4 estados que permite que los *reg* y los *wires* sean:
 - ▶ x: desconocido
 - ▶ 0: false o nivel cero
 - ▶ 1: true o nivel 1
 - ▶ z: alta impedancia
- 

Variables: Ej. lógica 4 estados

Table 2-2 Numbers and Their Values

Number	Value	Number	Value
8 'b0	00000000	8 'b1	00000001
8 'bx	xxxxxxxx	8 'hz1	zzzz0001
8 'b1x	0000001x	8 'x1	xxxxxxxx1
8 'b0x	0000000x	8 'bx0	xxxxxxxx0
8 'hx	xxxxxxxx	8 'hz	zzzzzzzz
8 'hzx	zzzzxxxx	8 'h0z	0000zzzz

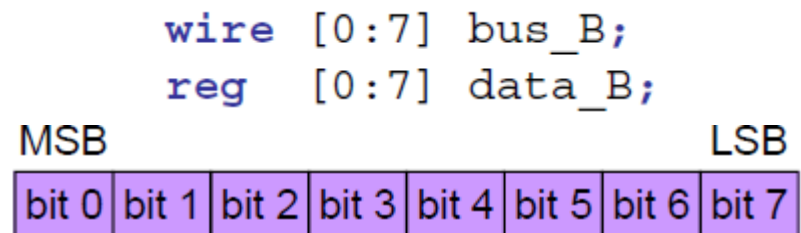
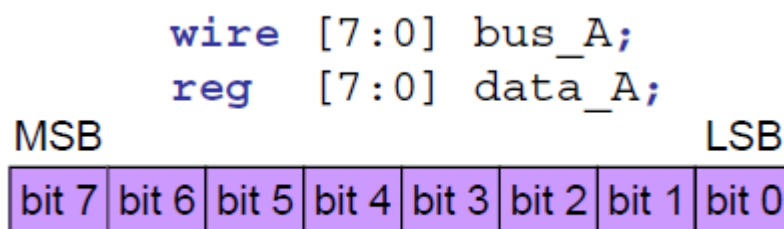
Variables: Buses

- ▶ Los buses se declaran como elementos de varios bits. Se pueden concatenar bits para obtener un bus.

```
module compuerta
(  input wire [3:0] entrada1,
    input wire [3:0] entrada2,
    output wire [3:0] salida);
```

Variables: Buses

- ▶ Los buses pueden realizarse tanto con conexiones (wire) como con variables (reg).
- ▶ En la declaración de un bus, el valor y el orden de los índices de los bits, determina el tamaño del bus y la ubicación del bit más significativo.



Verilog



Modulos: Comportamiento

Comportamiento: Sentencias Concurrentes

- Luego de completar la declaración de puertos, parámetros, constantes y variables, el paso que sigue es describir la funcionalidad del módulo. Esto se realiza mediante sentencias concurrentes.

```
module chip
  #(//parámetros)
    (//puertos: entradas, salidas
    );
```

...

Procesos
concurrentes



```
endmodule
```

Todos simultáneos



Comportamiento: Conexiones

- ▶ Para crear señales internas que modelan conexiones eléctricas se usan los wires.
- ▶ Los puertos de I/O son wires por defecto.
- ▶ Se usa el keyword **wire** para declararlos:

```
wire AB;
```

- ▶ Se usa el keyword **assign** para asignarle un valor:

```
assign AB = A & B;
```

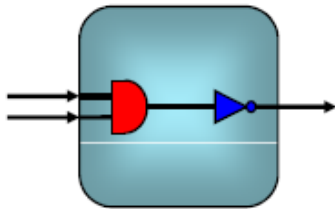
- ▶ Se puede declarar un wire y asignarlo en la misma línea:

```
wire AB = A & B;
```

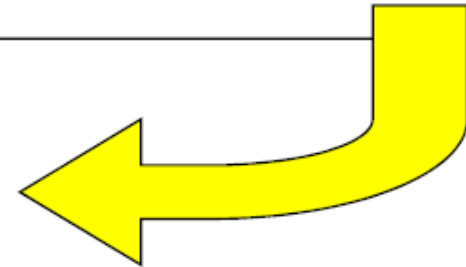

Comportamiento: Conexiones

- ▶ Todo wire debe declararse previamente a su uso (lectura o asignación).
- ▶ Hay wires externos (puertos) e internos al módulo:

Las conexiones externas se declaran entre paréntesis a continuación del nombre del módulo.



```
module compuerta (  
    input  entrada1,  
    input  entrada2,  
    output salida  
);
```



```
    wire  conex_int;
```

```
    assign conex_int = entrada1 & entrada2;  
    assign salida = ~conex_int;
```

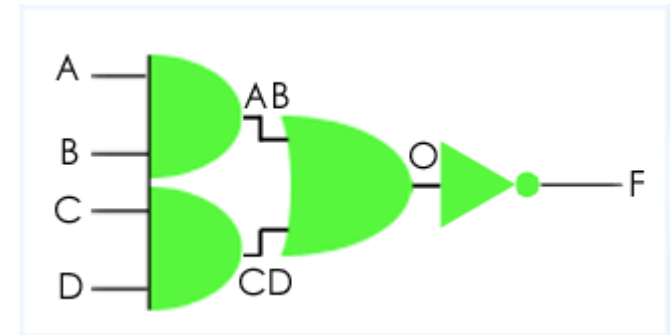
```
endmodule
```

Las conexiones internas se definen en la sección declarativa del módulo



Comportamiento: Ejemplo wires

```
// Verilog code for AND-OR-INVERT gate
module AOI (input A, B, C, D, output F);
  wire F;
  wire AB, CD, O;
  assign AB = A & B;
  assign CD = C & D;
  assign O = AB | CD;
  assign F = ~O;
endmodule // end of Verilog code
```



Comportamiento: Op. Aritmeticos

Operadores aritméticos



Suma

$a = b + c;$



Resta

$a = b - c;$



Multiplicación

$a = b * c;$



División

$a = b / c;$



Módulo

$a = b \% c;$



Potenciación

$a = b ** c;$

Comportamiento: Op. Binarios

Operadores binarios a nivel de bits y de reducción



not (negación, inversión)

`x = ~ y; // solo a nivel de bits, arg. único`



and (y)

`x = y & z; //nivel bits`
`w = & u; //reducción`



nand (no y)

`x = ~ & z; //solo reducción`



or (o)

`x = y | z; //nivel bits`
`w = | u; //reducción`



nor (no o)

`x = ~ | z; //solo reducción`



xor (o exclusivo)

`x = y ^ z; //nivel bits`
`w = ^ u; //reducción`



xnor (no o exclusivo)

`x = y ~ ^ z; //nivel bits`
`w = ~ ^ u; //reducción`

Comportamiento: Op. binarios de desplazamiento



Right shift (desplazamiento a la derecha)

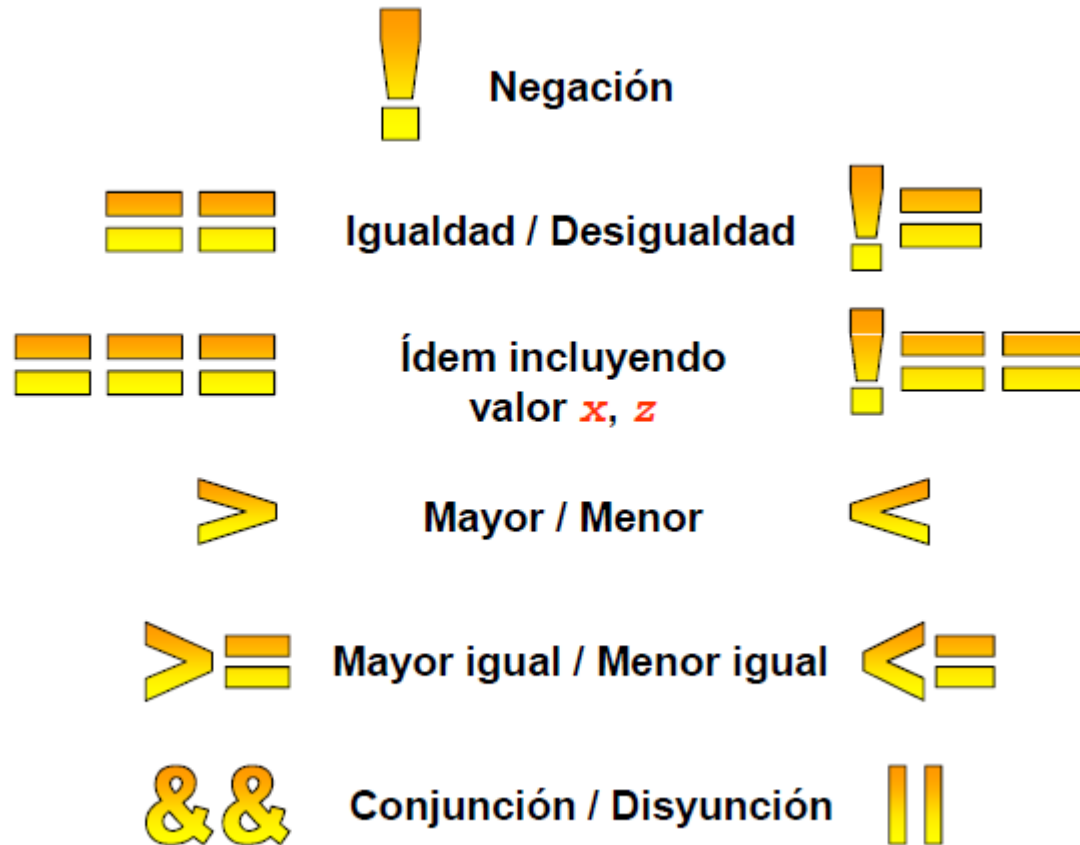
```
x = z >> 3; /* si z es 1011_0011  
              x será 0001_0110  
              es decir rellena  
              con ceros*/
```



Left shift (desplazamiento a la izquierda)

```
x = z << 3; /* si z es 1011_0011  
              x será 1001_1000  
              ídem anterior */
```

Comportamiento: Op lógicos y relacionales con resultado booleano



Comportamiento: Op. de concatenacion y replicacion



Concatenación de argumentos

```
x = { a, b }; /* si a es 011 y b es 110  
              x será 011_110 */
```

Operador de replicación



Replicación de un argumento

```
x = { a {b} }; /* si a es 3 y b es 110  
                x será 110_110_110 */
```

Comportamiento: Extensiones Aritméticas para enteros

- Los tipos de datos **reg** y **wire** pueden declararse como **signed**.
`reg signed [63:0] data;`
`wire signed [11:0] address;`
- Los números pueden declararse como **signed**.
`16'shC501 //hexadecimal long. 16 bits con signo`
- Los operadores aritméticos `<<<` y `>>>` mantienen el signo del operando.
- Las funciones del sistema **\$signed()** y **\$unsigned()** permiten convertir sus argumentos a **signed** o **unsigned**.

Comportamiento: Op. Aritméticos con signo



Right shift (desplazamiento a la derecha)

```
x = z >>> 3; /* si z es 1011_0011  
               x será 1111_0110 es decir  
               rellena con el signo para  
               variables signed, sino con ceros */
```

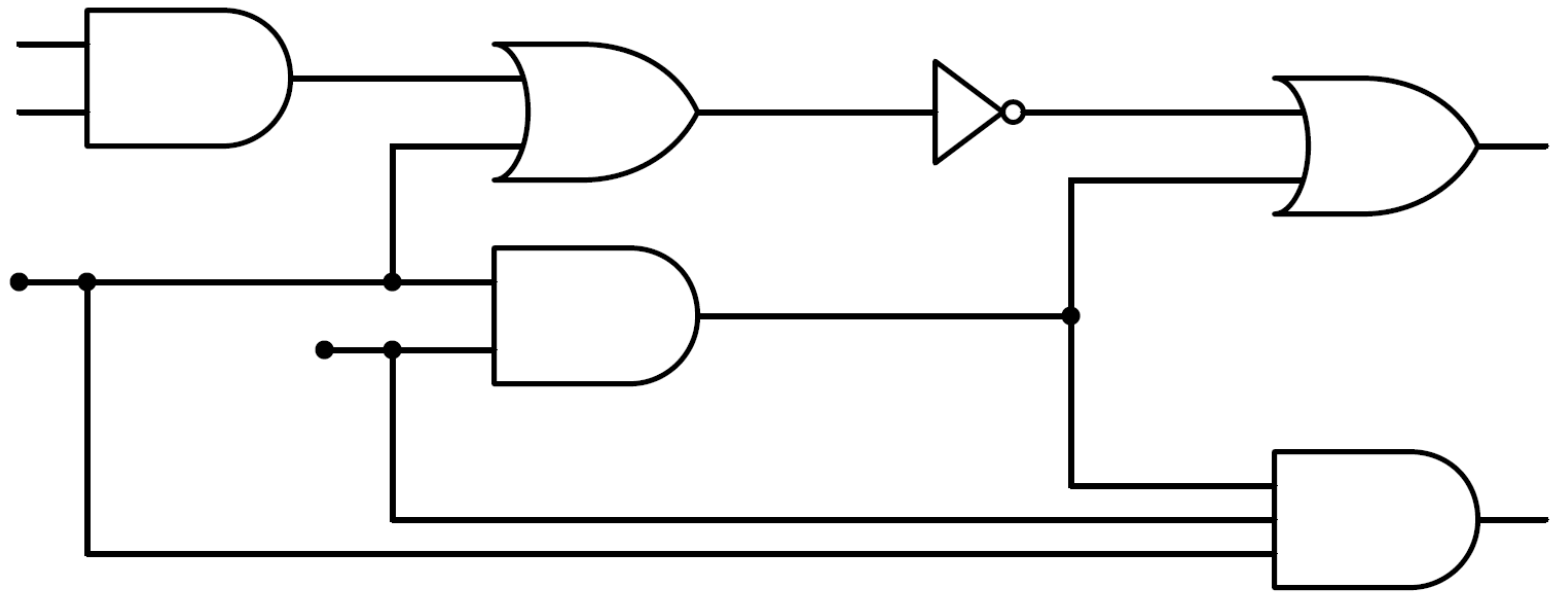


Left shift (desplazamiento a la izquierda)

```
x = z <<< 3; /* si z es 0011_0011  
               x será 1001_1000  
               rellena con ceros  
               y el resultado sigue siendo  
               signed */
```

Ejercicio

- ▶ Con lo visto desarrollar en Verilog un módulo llamado `multi_compuerta`, según el diagrama siguiente, sintetizar con el ISE, sin warnings, y mostrar el esquemático RTL y de tecnología generado.

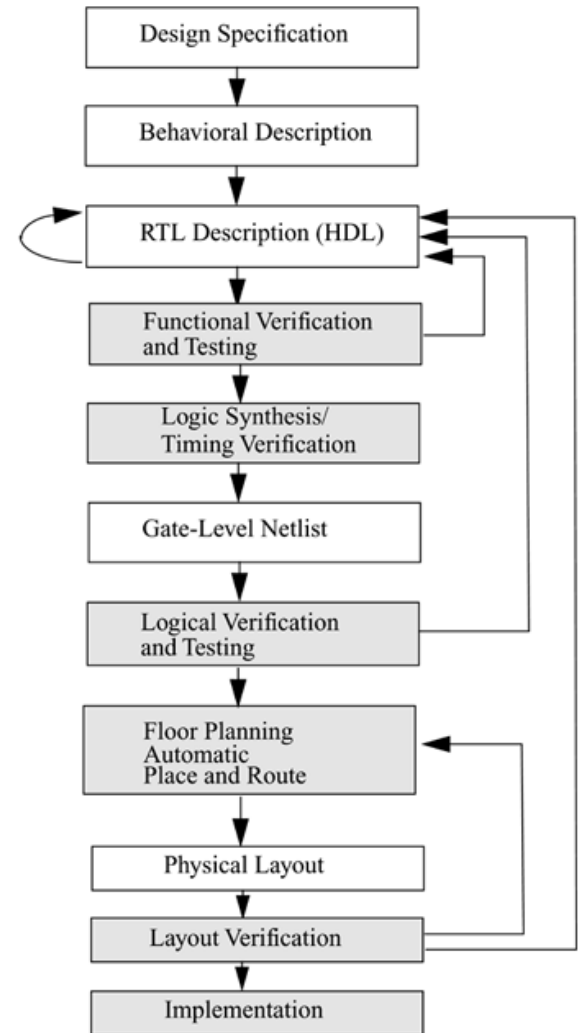


Flujo de Diseño



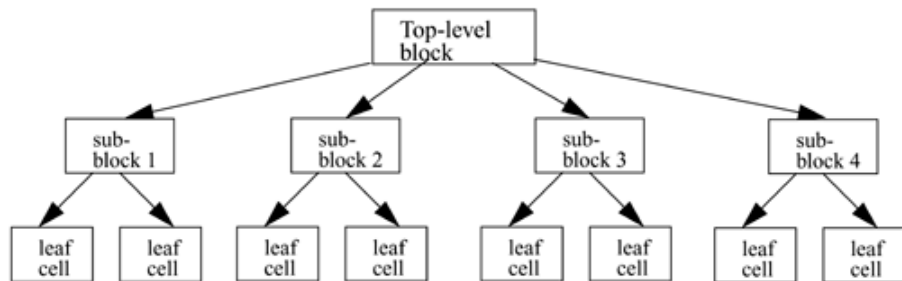
Flujo de diseño

- ▶ Se escriben especificaciones del circuito/sistema (funcionalidad, interfaces y arquitectura general).
- ▶ Con una descripción de comportamiento se analiza el diseño a alto nivel (se modela con lenguaje de alto nivel ej C++, Matlab).
- ▶ El modelo comportamental se convierte a una descripción RTL (dataflow) en un HDL.
- ▶ De ahí en más el proceso continúa con la ayuda de herramientas EDA: se sintetiza el circuito a un "netlist", se crea un layout del mismo a través del "place and route", se verifica y fabrica el chip.
- ▶ A la par se verifican los resultados de cada etapa para asegurar que cumple con las restricciones de timing, potencia y función.



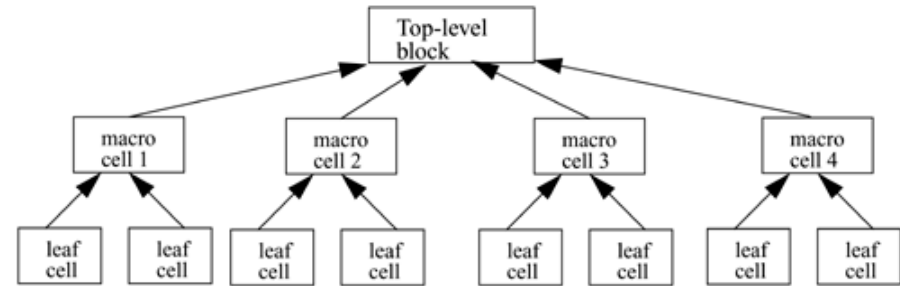
Metodologías de diseño

TOP-DOWN



- ▶ Definimos el bloque "top" e identificamos los subbloques que lo conforman y lo vamos componiendo de lo general a lo específico.

BOTTOM-UP



- ▶ Primero identificamos los bloques que tenemos a disposición y a partir de estos construimos bloques mas grandes, componiendo hacia lo general.

- ▶ Normalmente se diseña mediante una combinación de ambos.

Niveles de Abstracción

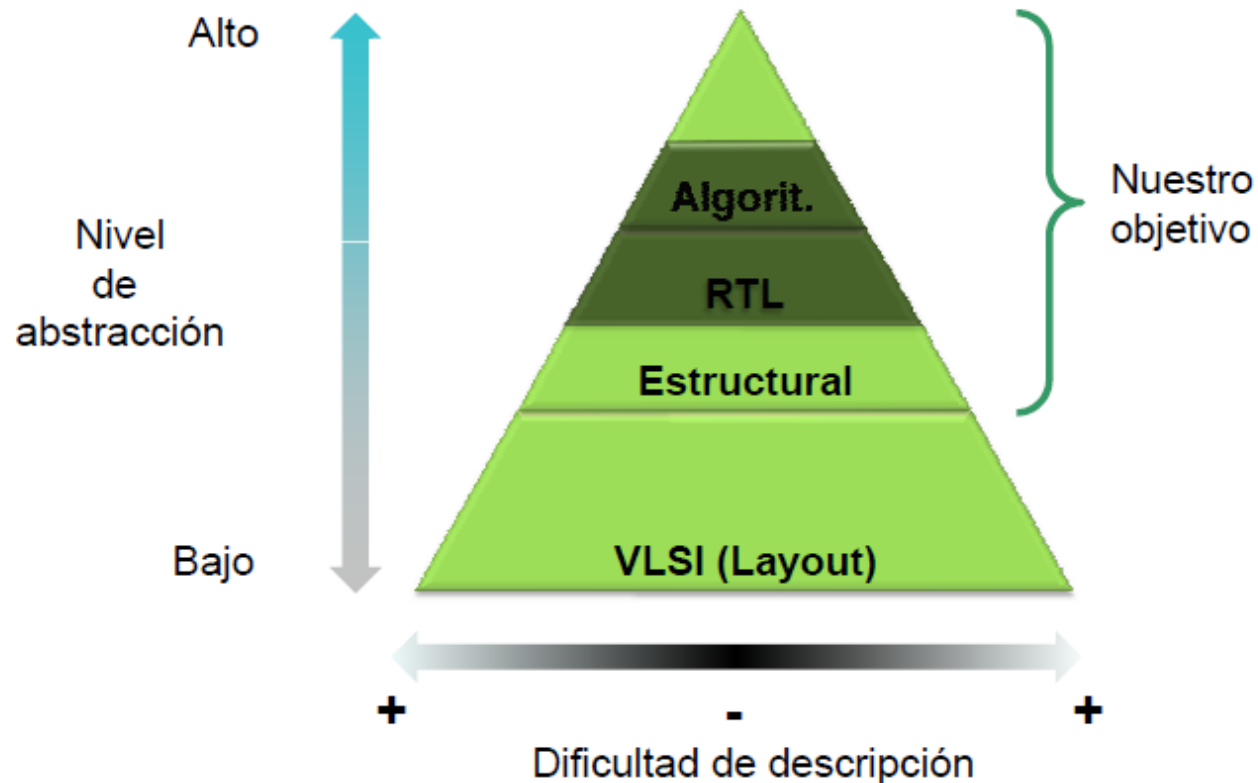


Niveles de abstraccion

System
Architectural
Behavioral
Algorithmic
Register Transfer Level (RTL)
Boolean Equations
Structural
Gates
Switches
Transistors
Polygons
Masks

Tipos de descripciones

Cada estilo de descripción posee un grado de abstracción y dificultad diferente.

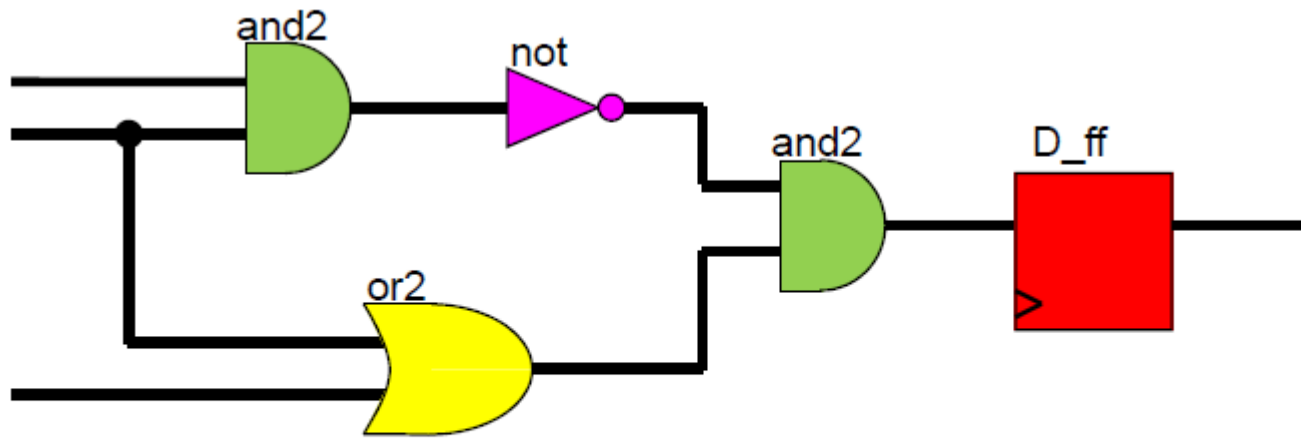


Niveles de Abstracción

» Descripción Estructural

Descripción Estructural

También llamada Procedural



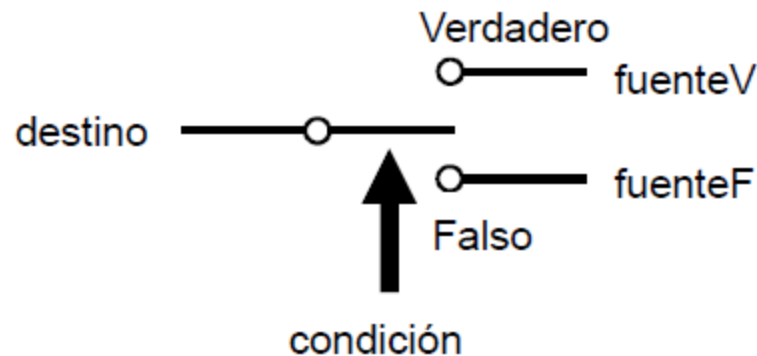
Una descripción estructural de un diseño emplea componentes previamente definidos y los interconecta de manera adecuada.

Descripción Estructural

Sentencias utilizadas por la descripción flujo de datos: asignación condicional.

```
assign destino = (condición) ? fuenteV : fuenteF;
```

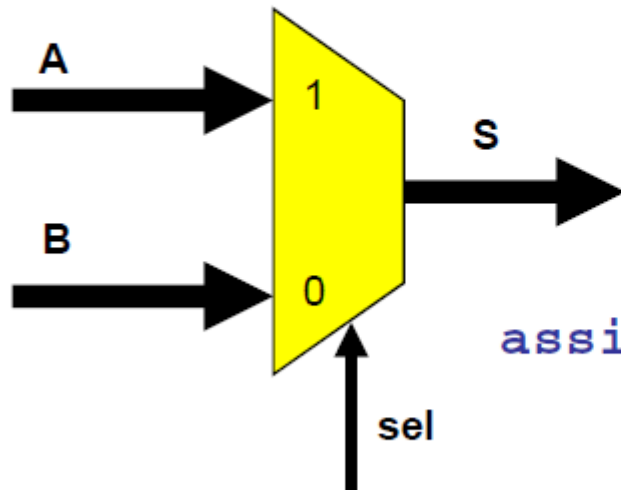
fuenteV y *fuenteF* pueden ser expresiones, variables o constantes
condición es una expresión con resultado booleano.



Descripción Estructural

Sentencias utilizadas por la descripción flujo de datos: asignación condicional.

Es ideal para describir multiplexores 2x1:



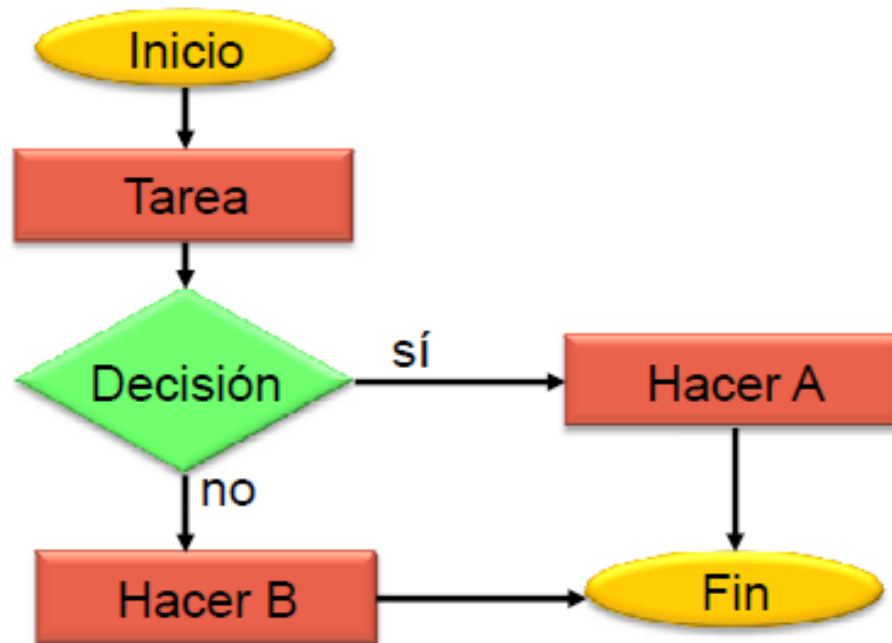
```
assign S = ( sel ) ? A : B;
```

Niveles de Abstracción

» Descripción Algorítmica

Descripción Algorítmica

También llamada Behavioral

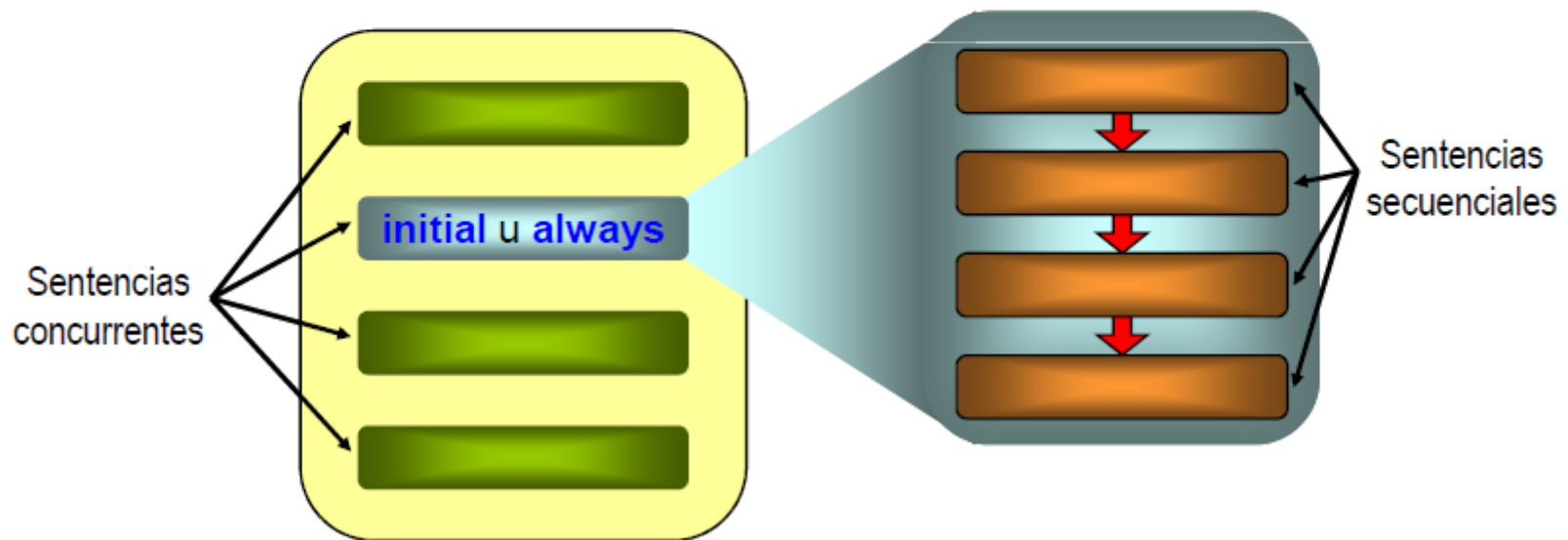


Define un diseño mediante algoritmos secuenciales similares a los utilizados en lenguajes de programación convencionales. Por ende, utiliza sentencias secuenciales.

Descripción Algorítmica

La base de las descripciones secuenciales: los bloques `initial` y `always`.

Los bloques `initial` y `always`, son construcciones que permiten, dentro de un lenguaje concurrente como Verilog, la declaración de sentencias secuenciales.



Descripción Algorítmica: Bloques Always

En estos bloques se pueden escribir sentencias secuenciales solamente.

```
always...  
begin  
    /*sentencias  
       secuenciales*/  
end
```

- Inicia cuando arranca la simulación.
- Reinicia cuando se alcanza el fin del bloque (**end**).

Descripción Algorítmica: Lista de Sensibilidad

El bloque `always`: su estructura

```
always @(lista_de_sensibilidad)
```

```
begin [: nombre_bloque]
```

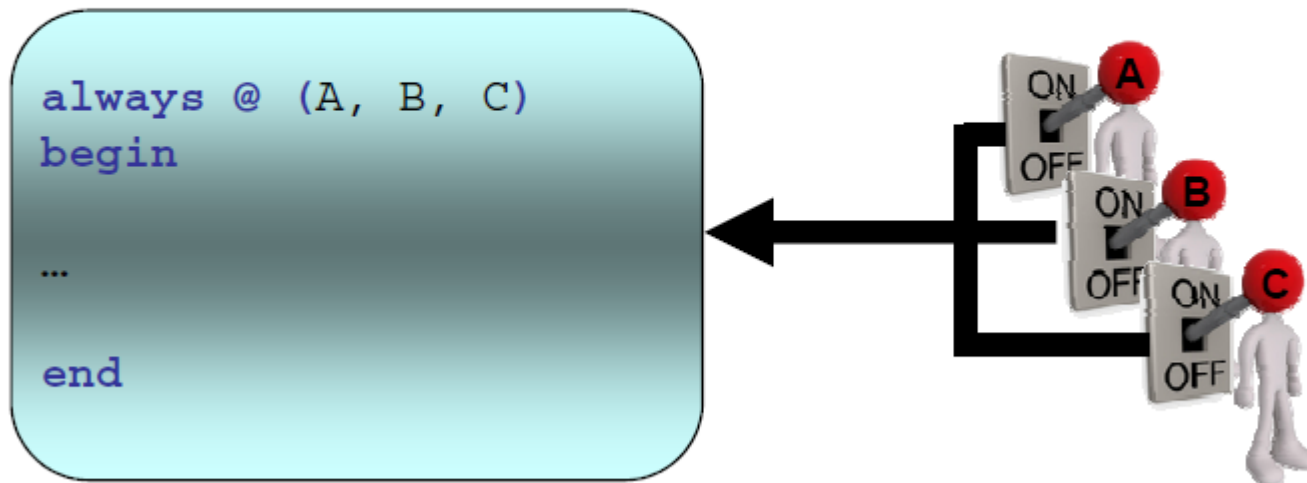
```
    /* sentencias  
       secuenciales*/
```

```
end
```



Descripción Algorítmica: Lista de Sensibilidad

La lista de sensibilidad define a través de las variables listadas en ella, el momento de evaluación del bloque. Cuando alguna de estas variables cambie de valor, éste se ejecutará.



Descripción Algorítmica:

Lista de Sensibilidad

- ▶ Cuando la lista sensitiva de variables de entrada a un bloque es muy grande se puede usar el símbolo @* que implica que el bloque se ejecuta con el cambio de CUALQUIERA de las señales.

```
//Combination logic block using the or operator
//Cumbersome to write and it is easy to miss one input to the block
always @(a or b or c or d or e or f or g or h or p or m)

begin
out1 = a ? b+c : d+e;
out2 = f ? g+h : p+m;
end

//Instead of the above method, use @(*) symbol
//Alternately, the @* symbol can be used
//All input variables are automatically included in the
//sensitivity list.
always @(*)
begin
out1 = a ? b+c : d+e;
out2 = f ? g+h : p+m;
end
```

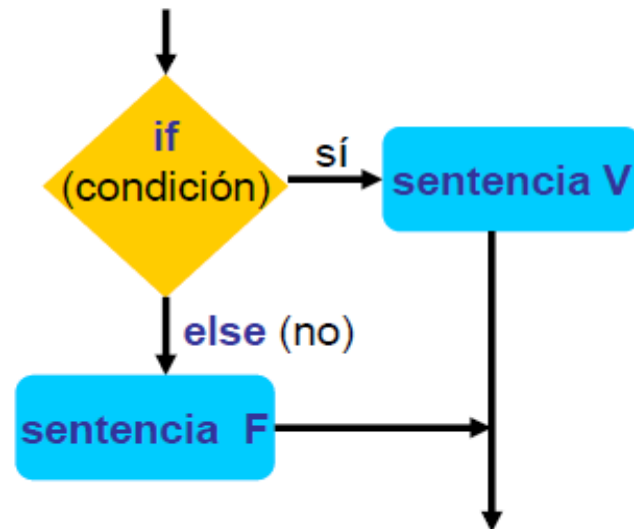
Descripción Algorítmica: If... else

El bloque always

Sentencia condicional **if ... else**

condición es una
expresión con
resultado booleano

```
if ( condición )  
    sentenciaV ;  
[ else  sentenciaF ; ]
```



Descripción Algorítmica: Case

Descripción Algorítmica: Sentencias Secuenciales

El bloque always

El condicional múltiple *case*

```
case (expresión)
  caso1:   sentencias1;
  caso2:   sentencias2;
  ...
  casoN:   sentenciasN;
  default: sentenciasDef;
endcase
```

Descripción Algorítmica:

Bucle for

```
integer j;  
  
for (j=0;j<=7;j=j+1)  
begin  
    c[j] = a[j] + b[j];  
end
```

Descripción Algorítmica: Asignación Secuencial

Asignación bloqueante

- `variable = sentencia`
- Similar a código en C.
- La siguiente asignación espera hasta que la actual termina. Las asignaciones se ejecutan en secuencia.
- Modela lógica combinacional.

Asignación antibloqueante

- `variable <= sentencia`
- Las entradas se almacenan cuando se activa el bloque secuencial.
- Las sentencias y asignaciones se ejecutan en paralelo.
- Modela *flip-flops*, *latches* y registros.



NO MEZCLE AMBAS ASIGNACIONES EN UN BLOQUE

Descripción Algorítmica:

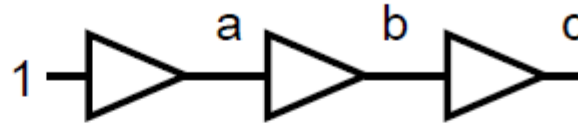
Asignaciones bloqueantes y no bloqueantes

- RHS de bloqueantes tomada de conexiones (wires)

`a = 1;`

`b = a;`

`c = b;`

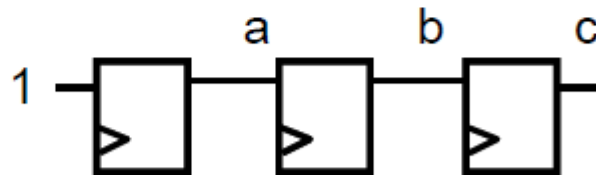


- RHS de antibloqueante tomada a las salidas de *latches*

`a <= 1;`

`b <= a;`

`c <= b;`



El resultado de una asignación antibloqueante
luce como un *latch*

Ejemplo: Comparación

Estructural

- Piense en la implementación
- El orden de las sentencias no importa
- Se usan sentencias **assign** o **generate**
- Descripción más compleja
- Se debe construir el circuito digital

```
wire c, d;  
assign c = a & b;  
assign d = c | b;
```

Algorítmica

- Piense en el resultado
- El orden de las sentencias sí importa
- Se usan sentencias **initial** u **always**
- Descripción más sencilla
- Pueden emplearse sentencias de control de flujo: **if**, **case**, **for**.

```
reg c, d;  
always@ (a, b, c)  
begin  c = a & b;  
       d = c | b;  
end
```

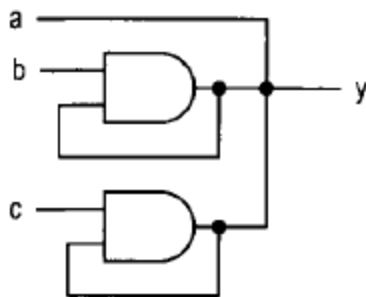
Estructural (assign) vs. Algorítmica (always)

► Estructural (modelo incorrecto)

Listing 3.3 Incorrect code for a reduced and circuit

```
module and_cont_assign
(
  input wire a, b, c,
  output wire y
5 );

  assign y = a;
  assign y = y & b;
  assign y = y & c;
10
endmodule
```



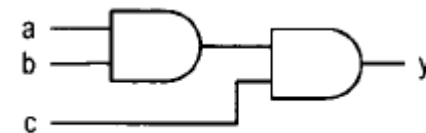
► Algorítmica (modelo correcto)

Listing 3.2 Behavioral reduced and circuit using a variable

```
module and_block_assign
(
  input wire a, b, c,
  output reg y
5 );

  always @*
  begin
    y = a;
    y = y & b;
10    y = y & c;
  end

endmodule
```



Ejemplos

Listing 3.5 Binary decoder using an if statement

```
module decoder_2_4_if
(
    input wire [1:0] a,
    input wire en,
5    output reg [3:0] y
);

    always @*
        if (en==1'b0)                // can be written as (~en)
10            y = 4'b0000;
        else if (a==2'b00)
            y = 4'b0001;
        else if (a==2'b01)
            y = 4'b0010;
15        else if (a==2'b10)
            y = 4'b0100;
        else
            y = 4'b1000;

20 endmodule
```

Ejemplos

Listing 3.6 Binary decoder using a case statement

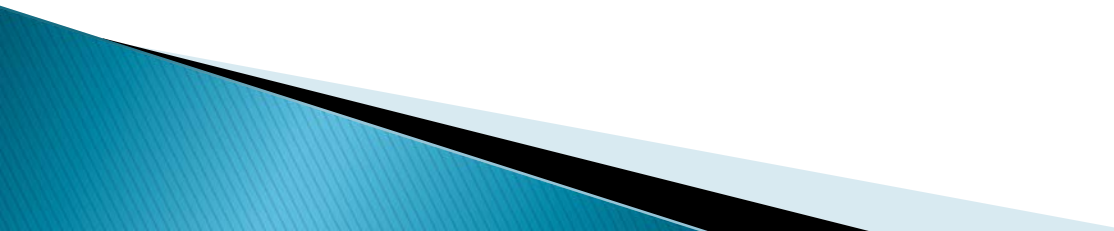
```
module decoder_2_4_case
(
    input wire [1:0] a,
    input wire en,
5    output reg [3:0] y
);

    always @*
        case({en,a})
10        3'b000, 3'b001, 3'b010, 3'b011: y = 4'b0000;
           3'b100: y = 4'b0001;
           3'b101: y = 4'b0010;
           3'b110: y = 4'b0100;
           3'b111: y = 4'b1000; // default can also be used
15        endcase

endmodule
```

Instanciación de Módulos

```
nombre_modulo
#(
    .parametro_modulo    (valor_parametro)
)
nombre_instancia
(
    .nombre_puerto_1    (conexion),
    .nombre_puerto_2    (conexion)
);
```



Descripción RTL

- Descripción RTL: Register Transfer Level



Este estilo de descripción establece una serie de pautas de descripción que aseguran que tanto el diseñador como las herramientas puedan identificar claramente la ubicación de los elementos de memoria.

Test Benches



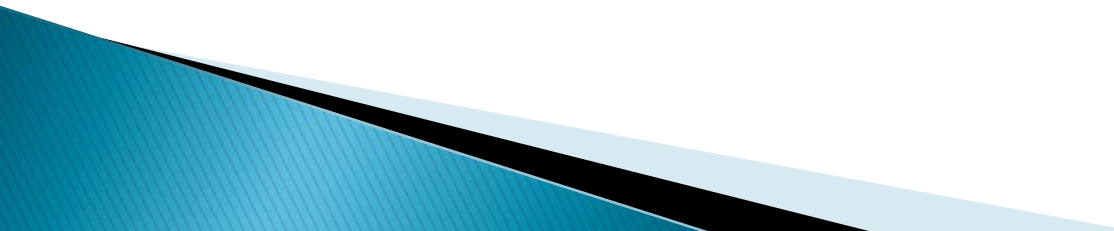
Test Benches

- ▶ Es un programa especial escrito en Verilog para verificar el diseño (DUT, Design Under Test)
- ▶ Imita un laboratorio físico para probar el circuito
- ▶ Se generan las señales de estímulo de entrada del diseño (test vector)
- ▶ Se evalúan las salidas del circuito (análisis)

Test Bench



Test Benches

- Es un módulo que no tiene puertos de I/O
 - Instancia el módulo a probar (DUT)
 - Utiliza variables (regs) para crear el test vector (stimulus)
 - Conecta el test vector a las entradas del DUT
 - Utiliza wires para conectar las salidas del DUT
 - Utiliza bloques inicial para generar el stimulus y evaluar las salidas
- 

Test Benches

```
module test_DUT; // No tiene puertos
    //DUT I/Os
    reg A, B, SEL;
    wire F;
    // DUT instantiation
    DUT my_dut(.A(A), .B(B), .SEL(SEL), .F(F));
    //Stimulus
    initial begin
        A = 0; B = 1; SEL = 0;
        #20 SEL = 1;
    end
    //Analysis
    initial $monitor($time, A, B, SEL, F);
endmodule
```

Test Benches: Stimulus

- ▶ En un bloque initial creamos el estimulo del componente a testear.

```
initial
```

```
// Stimulus
```

```
begin
```

```
    SEL = 0;
```

```
    A = 0;
```

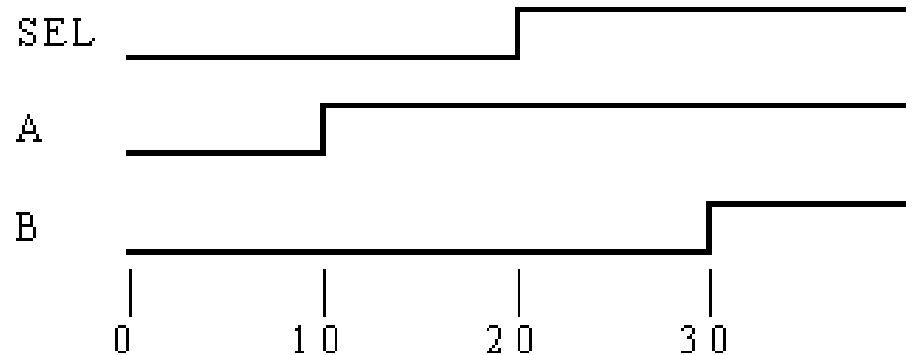
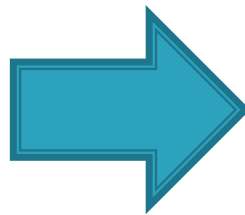
```
    B = 0;
```

```
    #10 A = 1;
```

```
    #10 SEL = 1;
```

```
    #10 B = 1;
```

```
end
```

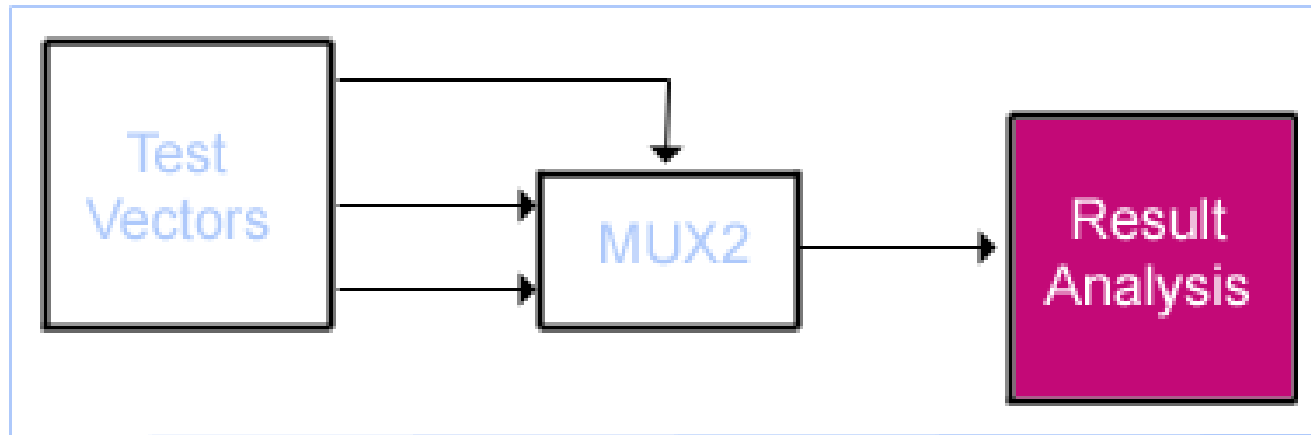


Test Benches:

Captura de las respuestas

```
// Analysis
```

```
initial $monitor($time, SEL, A, B, F);
```



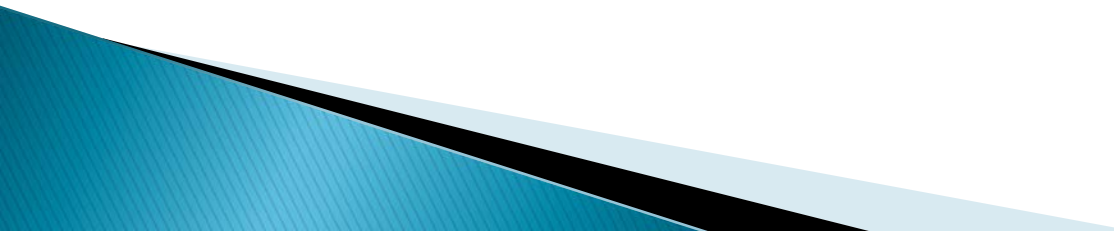
Test Benches:

Funciones y Tareas del Sistema

- ▶ Existen tareas y funciones predefinidas en Verilog.
- ▶ Sintácticamente todas las tareas y funciones del sistema comienzan con \$
- ▶ Proveen funcionalidad para:
 - Input-output desde archivos, la pantalla y el teclado
 - Control de simulación y debugging
 - Chequeos de tiempo, y analisis de probabilidades
 - Funciones de conversión entre los diferentes tipos

Test Benches:

Funciones y Tareas del Sistema

- ▶ **\$display** display values
 - ▶ **\$monitor** trace value-changes
 - ▶ **\$fopen, \$fclose** open, close a file
 - ▶ **\$readmem** memory read tasks
 - ▶ **\$time** simulation time
 - ▶ **\$finish, \$stop \$end** stop simulation
 - ▶ **\$dumpvars** dump data to file for waveform display
 - ▶ **\$setup, \$hold** setup and hold timing checks
- 

Test Benches:

Mostrar por Consola: `$display`

- ▶ Muestra los valores en el formato elegido por el usuario (parecido a un `printf` de C):

`$display $displayb $displayh $displayo`

```
reg [7:0] A;  
initial begin  
    A = 8b0000_1111 ;  
    $display ("%d %b %0b %h %0h", A, A, A, A, A);  
end
```

Test Benches:

FILE I/O

- ▶ La función `$fopen` abre un archivo y le asigna el file descriptor. `$swrite` escribe las salidas formateadas a un string.
- ▶ Verilog también provee tareas para la entrada de datos desde archivos o strings. `$fgetc`, `$fscanf`, `$sscanf` son para obtener caracteres desde un archivo, otras input tasks sirven para leer datos de memoria directamente. `$fread`, `$readmemh`.

Referencia

- ▶ **FPGA Prototyping By Verilog Examples:**
Xilinx Spartan-3 Version, Pong P. Chu

