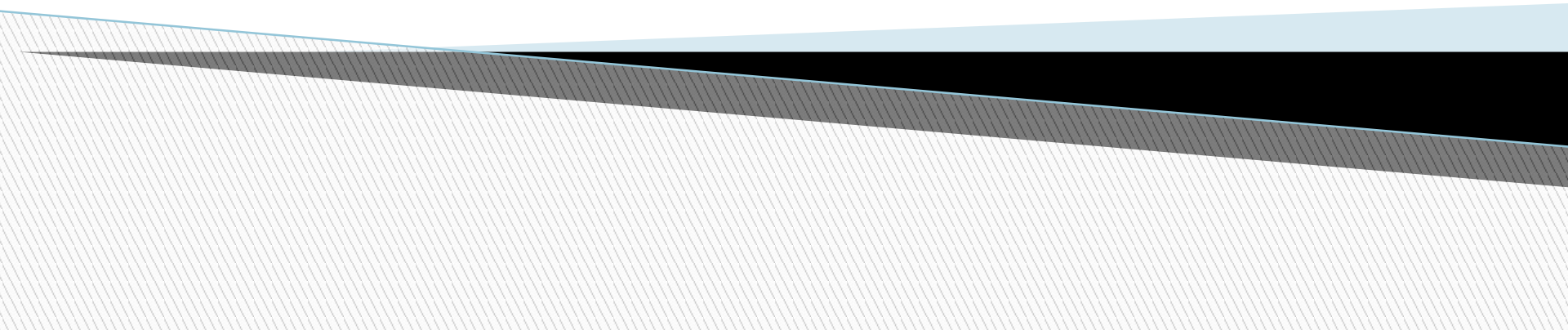
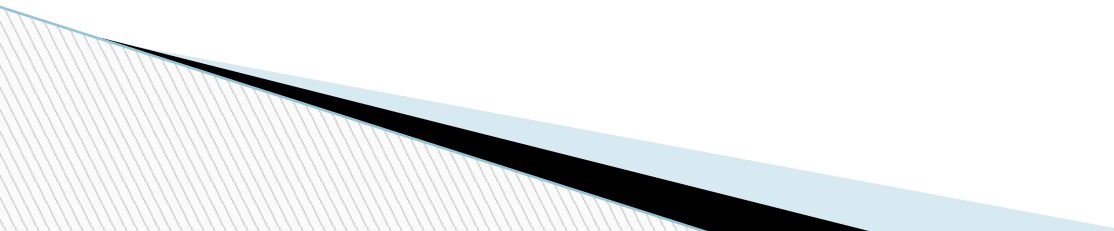


Lenguajes de Descripción de Hardware

Verilog



Contenido

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

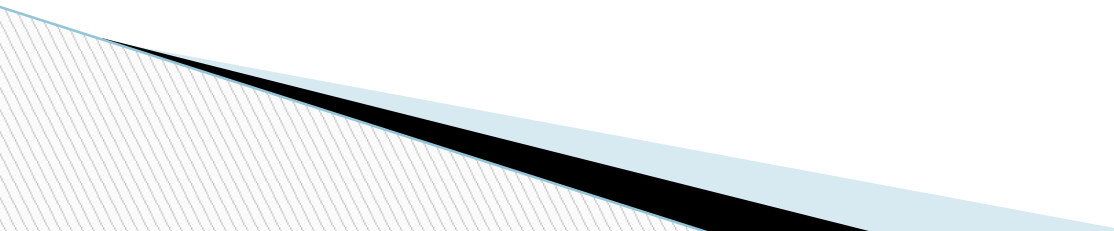
Objetivos de la Clase

- Conocer qué 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

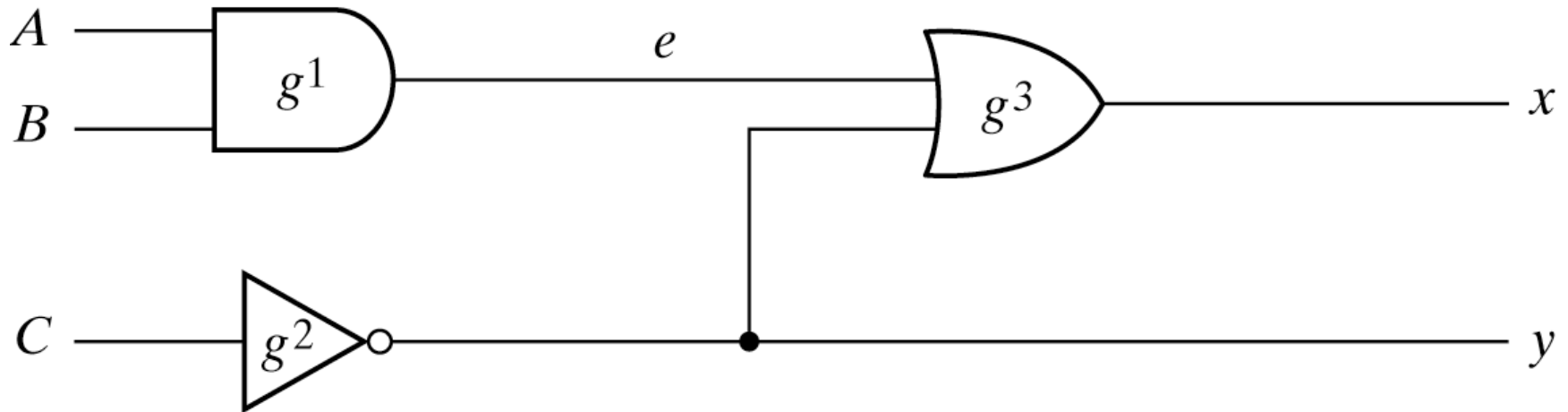
HDLs



HDLs

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

Ejemplo Módulo Simple



Verilog



Ejemplo Módulo Simple

```
module modulo_simple
(
  input wire A,
  input wire B,
  input wire C,
  output wire x,
  output wire y
);

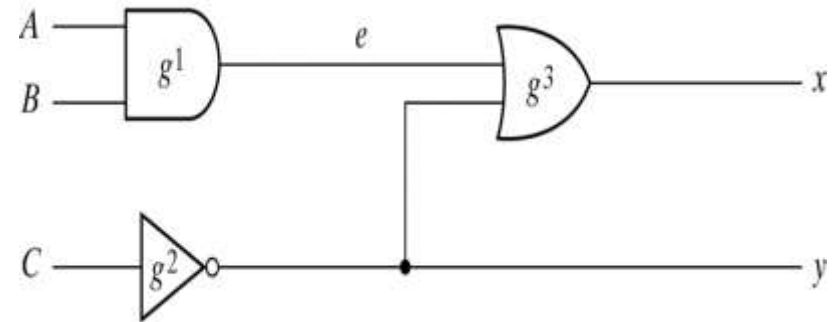
  wire e;

  assign e = A & B;

  assign y = ~ C;

  assign x = e | y;

endmodule
```



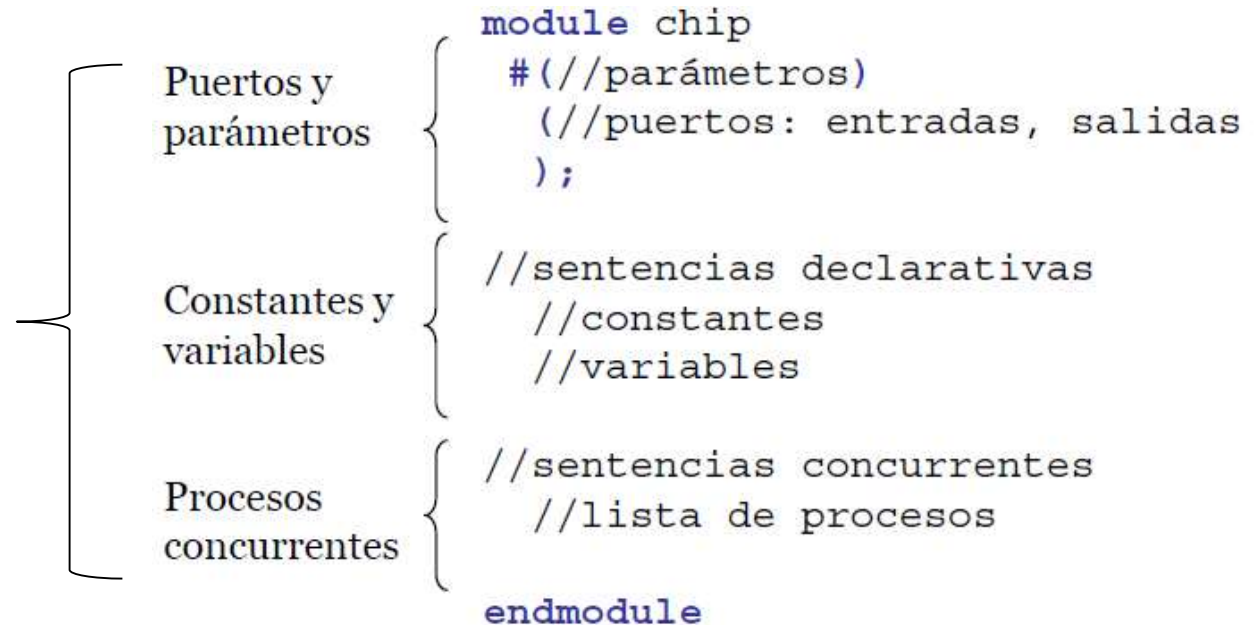
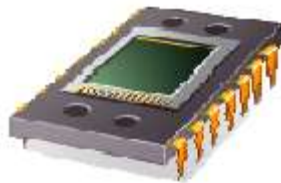
Diseño de caja negra

- Según el flujo de diseño, todo hardware se especifica 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);



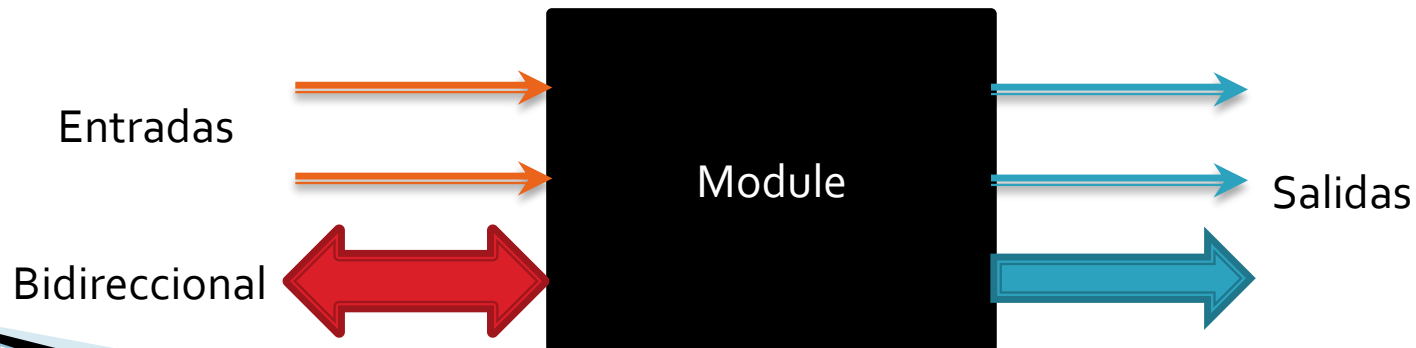


Verilog

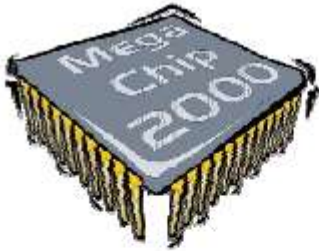
Módulos: Interfaz

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 parámetros son constantes.

```
parameter MSB = 7;
```

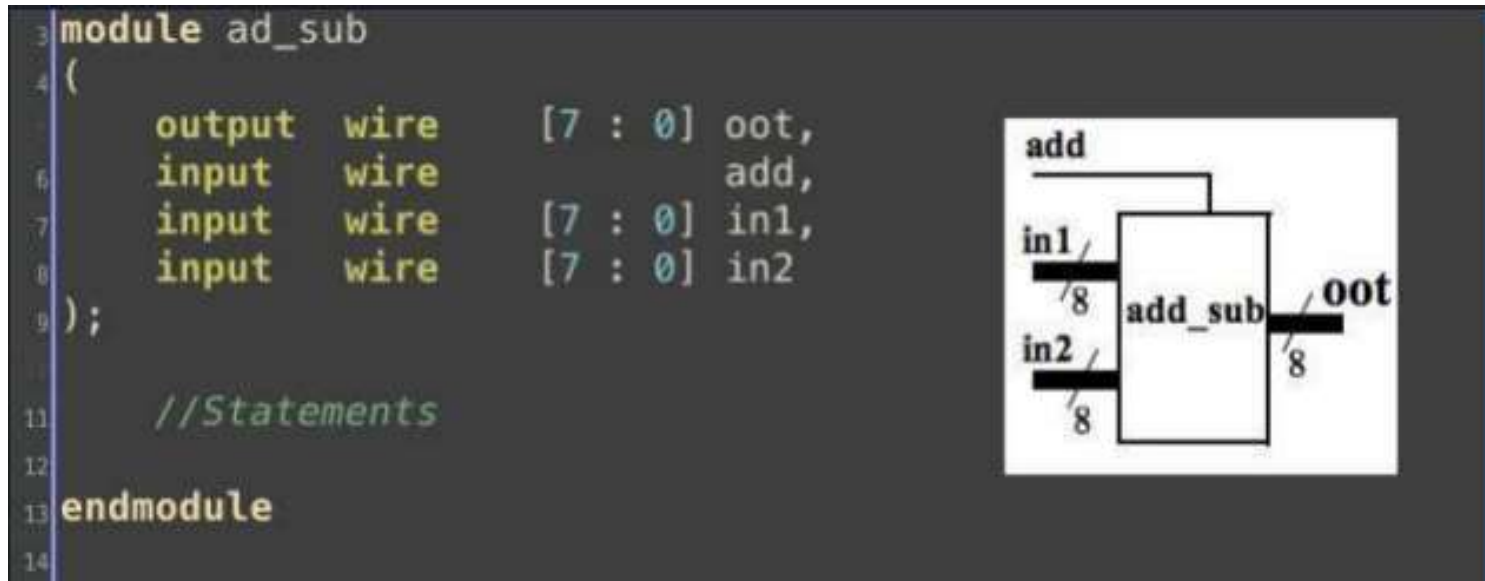
```
parameter R = 25, F = 9;
```

```
parameter AVERAGE_DELAY = (R + F) / 2;
```

```
parameter BYTE_SIZE = 8, BYTE_MASK = BYTE_SIZE - 1;
```

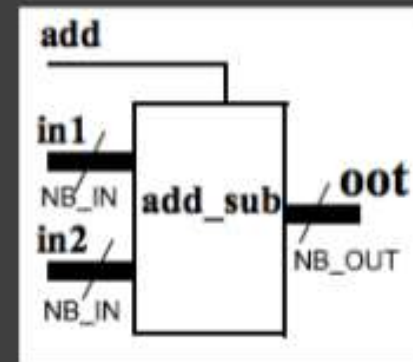
Módulos

- Principal entidad de diseño de Verilog.
- Las primeras líneas de la declaración especifican el nombre del módulo y sus puertos (dirección, tipo, tamaño y nombre)



Módulos Parametrizados

```
3 module add_sub
4 #
5 (
6     parameter                NB_IN    = 8,
7     parameter                NB_OUT   = 8
8 )
9 (
10
11     output  wire    [NB_OUT - 1 : 0]    oot,
12     input   wire    [NB_OUT - 1 : 0]    add,
13     input   wire    [NB_IN  - 1 : 0]    in1,
14     input   wire    [NB_IN  - 1 : 0]    in2
15 );
16
17     //Statements
18
19 endmodule
```





Verilog

Módulos: 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
----------------	---	-------	-------

Table 2-1 Radix Specifiers

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

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

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

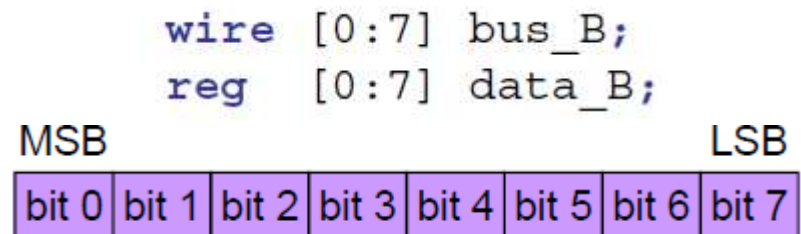
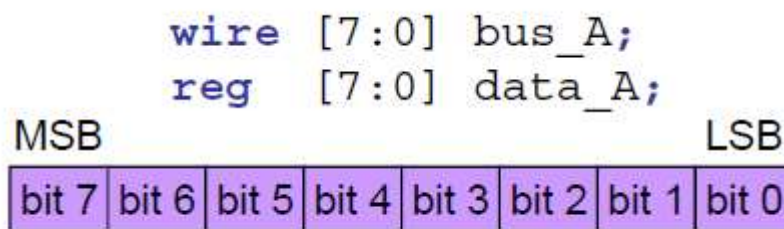
Variables: Buses

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

```
module compuerta
#(
    parameter NB_IN  = 4,
    parameter NB_OUT = 4
)
(
    input wire  [NB_IN-1:0]  i_entrada1,
    input wire  [NB_IN-1:0]  i_entrada2,
    output wire [NB_OUT-1:0] o_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.



Variables: Tipos de datos

- Wire: Representa una conexión física, utilizada para conectar compuertas o módulos. El valor de un wire puede ser leído en un bloque o una función, pero no asignado.
 - `wire [1:0] wire_name;`
- Registers: Representan variables que guardan información.
 - `reg [1:0] reg_name;`

Verilog



Módulos: 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 asignar un valor:

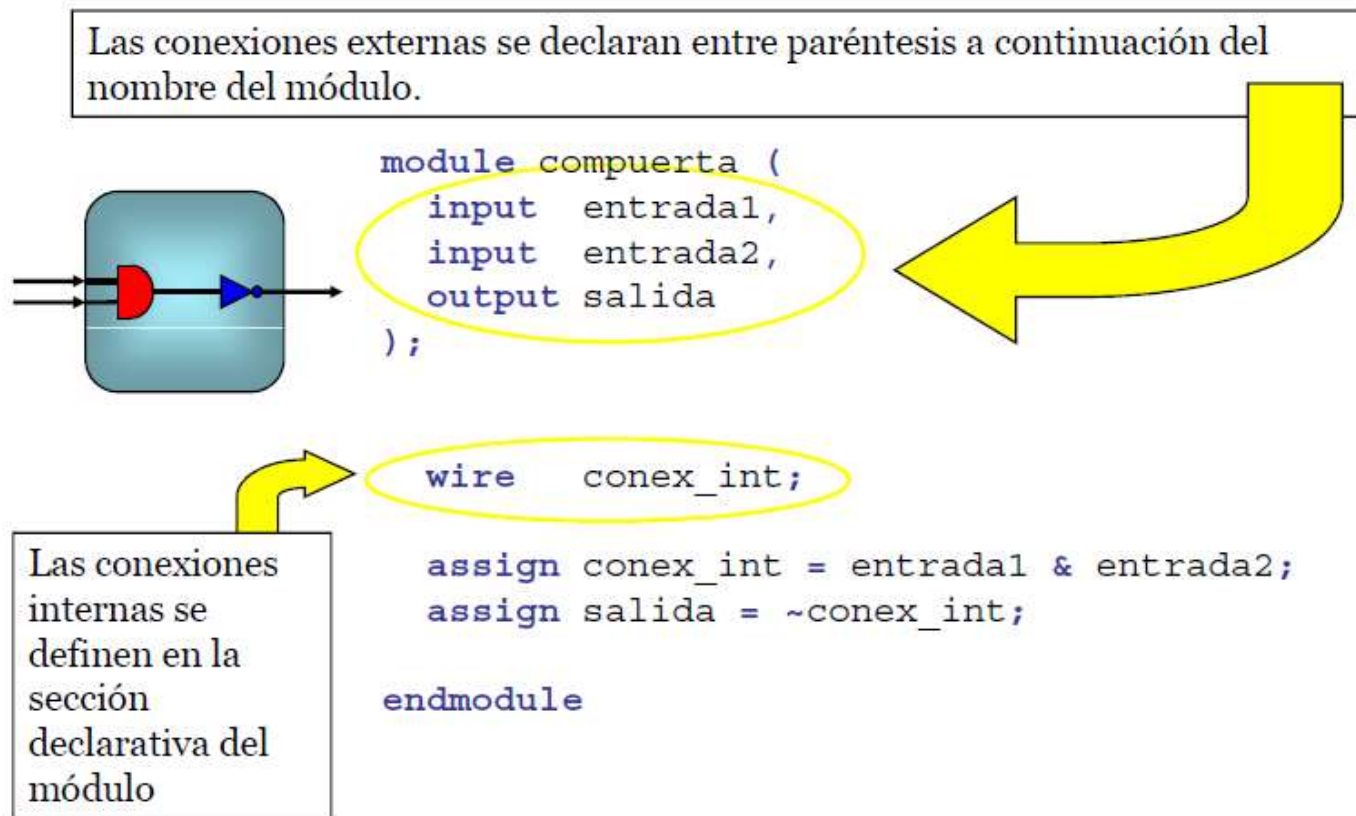
```
assign AB = A & B;
```

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

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

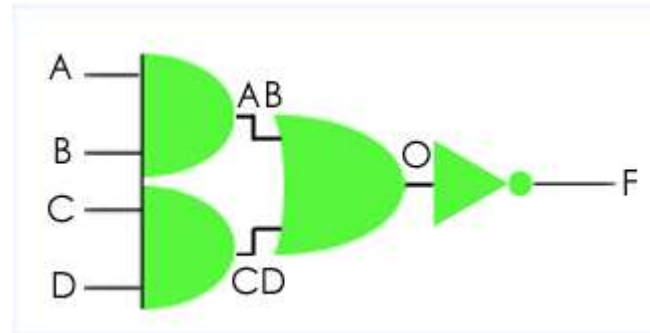
Comportamiento: Conexiones

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



Comportamiento: Ejemplo wires

```
// Verilog code for AND-OR-INVERT gate
module AOI
(
    input  wire A,
    input  wire B,
    input  wire C,
    input  wire D,
    output 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. Aritméticos

Operadores aritméticos



Suma

$a = b + c;$



Resta

$a = b - c;$



Multiplicación

$a = b * c;$



División

$a = b / c;$



Módulo

$a = b \% c;$

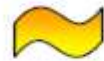


Potenciación

$a = b ** c;$

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)

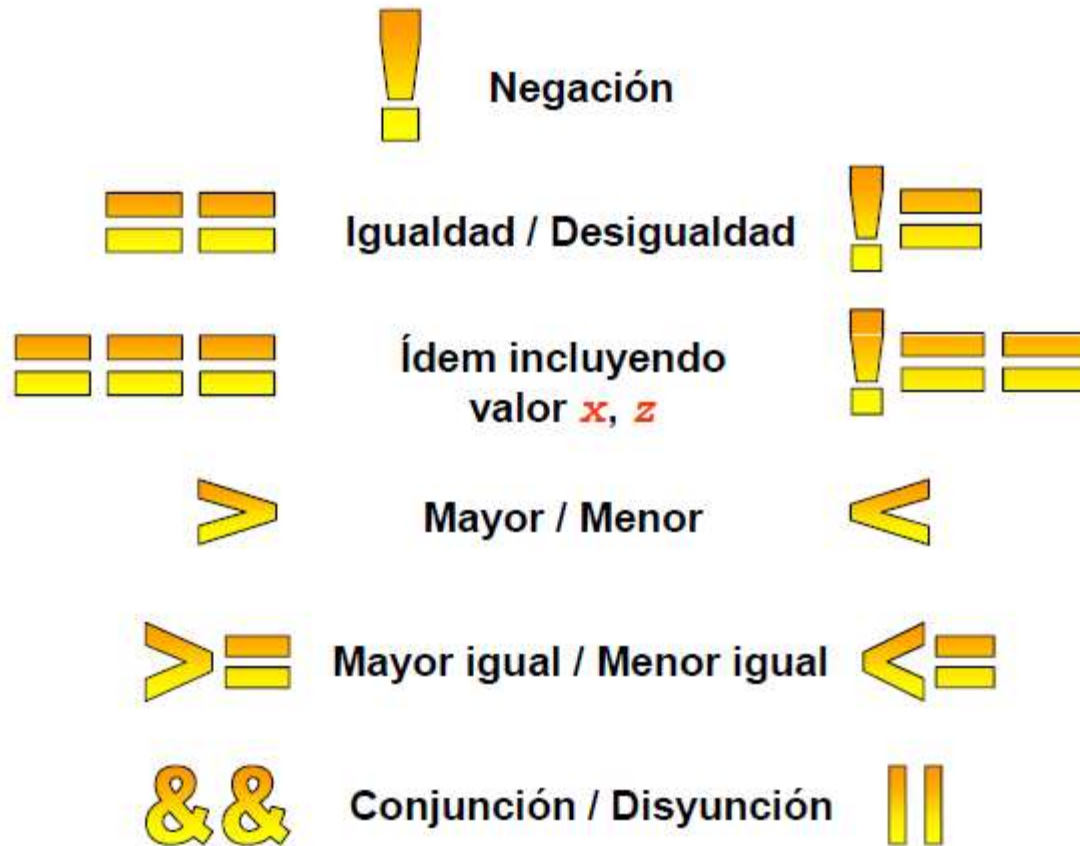
```
x = z >> 3; /* si z es 1011_0011  
              x será 0001_0110  
              es decir rellena  
              con ceros*/
```



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 concatenación y replicación



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 */
```

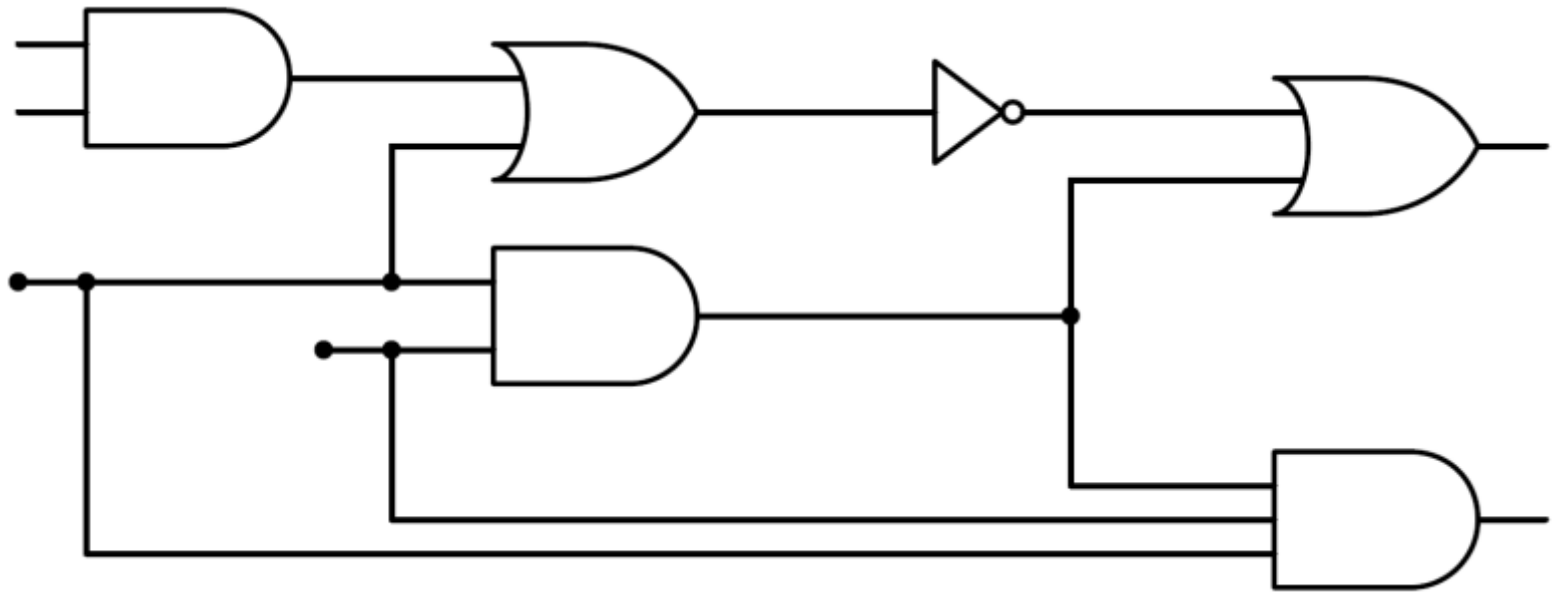


Left shift (desplazamiento a la izquierda)

```
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.



Niveles de Abstracción



Niveles de abstracción

A diagram illustrating the hierarchy of abstraction levels in computer engineering. The levels are listed vertically within a black rectangular border, starting from the highest level of abstraction at the top and descending to the most physical level at the bottom. The levels are: System, Architectural, Behavioral, Algorithmic, Register Transfer Level (RTL), Boolean Equations, Structural, Gates, Switches, Transistors, Polygons, and Masks. The text is centered within the box.

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.

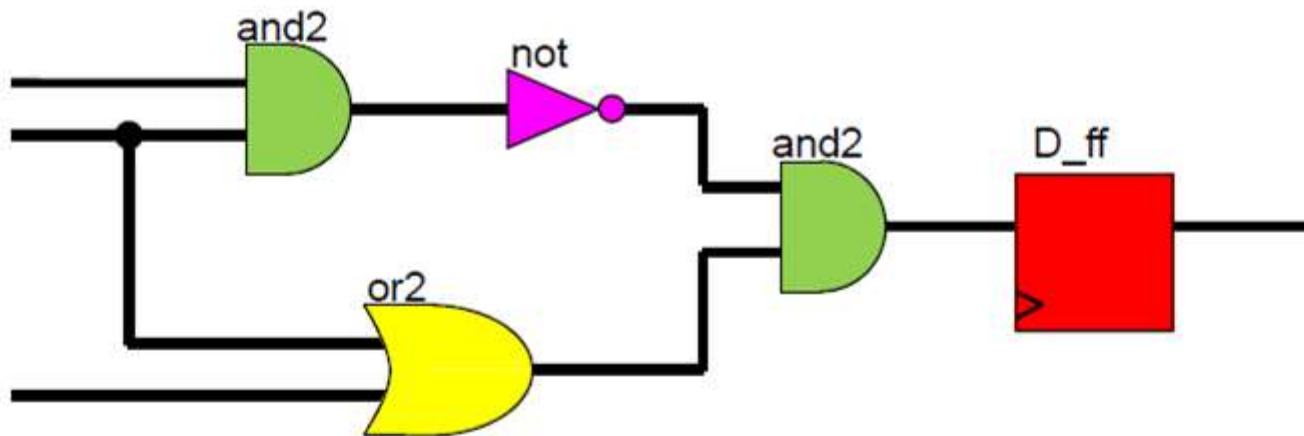


Niveles de Abstracción

» Descripción Estructural

Descripción Estructural

También llamada Procedural



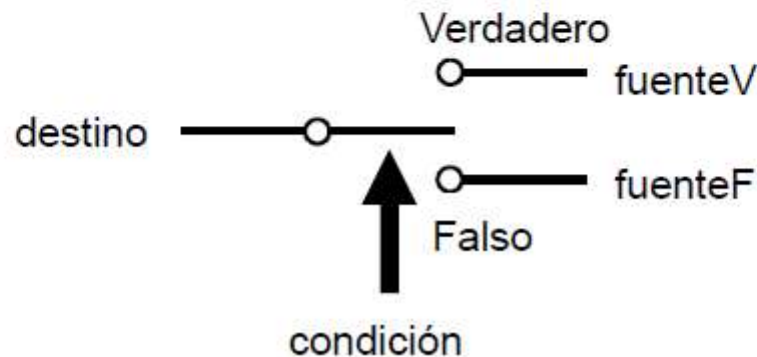
Una descripción estructural de un diseño emplea componentes previamente definidos y los interconecta de manera adecuada.

Descripción Estructural

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

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

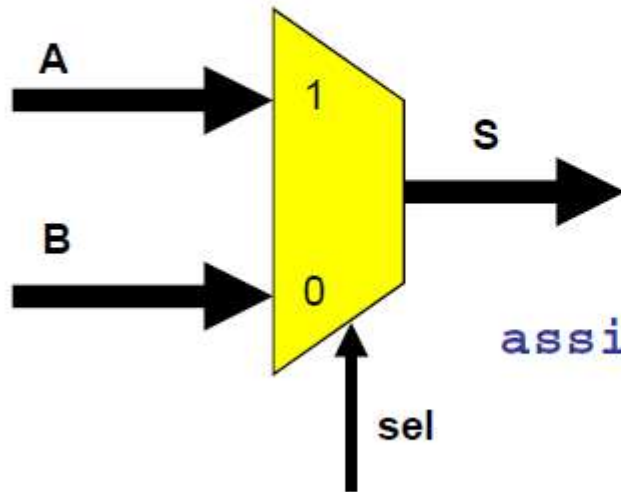
fuenteV y *fuenteF* pueden ser expresiones, variables o constantes
condición es una expresión con resultado booleano.



Descripción Estructural

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

Es ideal para describir multiplexores 2x1:



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

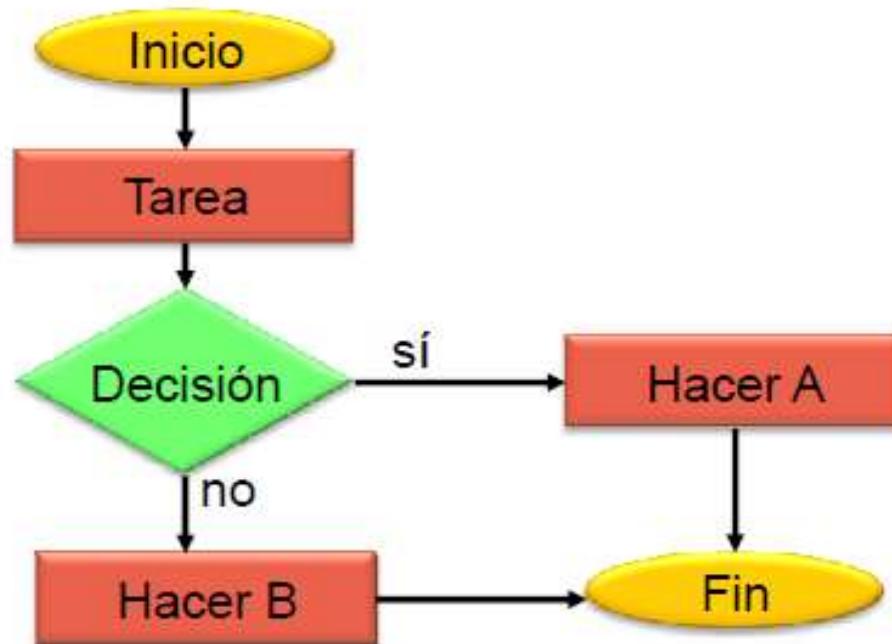
Niveles de Abstracción



Descripción Algorítmica

Descripción Algorítmica

También llamada Behavioral

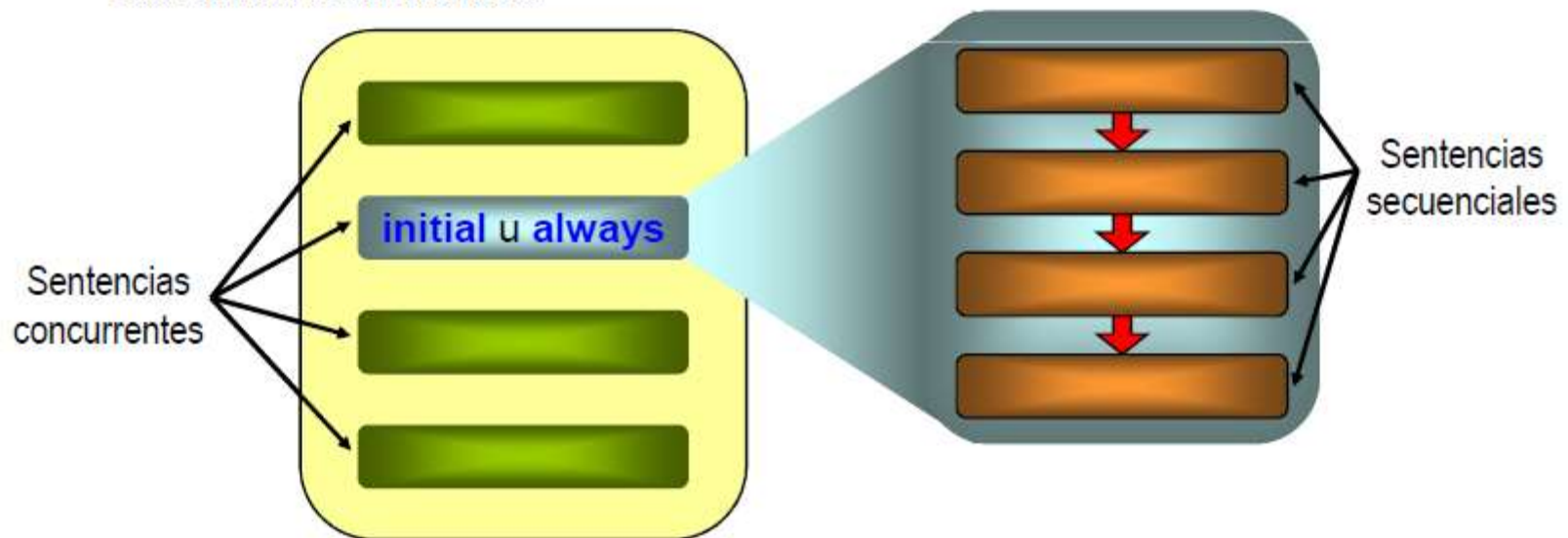


Define un diseño mediante algoritmos secuenciales similares a los utilizados en lenguajes de programación convencionales. Por ende, utiliza sentencias secuenciales.

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]
```

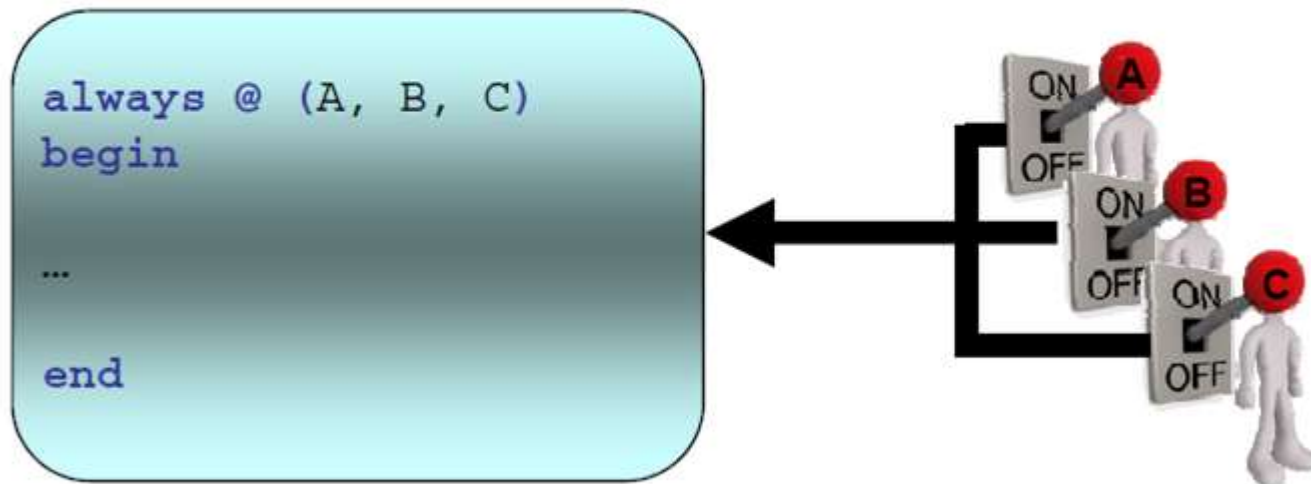
```
    /* sentencias  
       secuenciales*/
```

```
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: Bloque Always

- Eventos regulares
 - Cambio de valor de una señal
 - A la transición de uno a cero (posedge)
 - A la transición cero a uno (negedge)
 - Permite sintetizar lógica secuencial (sincronizada a un clock)
 - SOLO USAR POSEDGE O NEGEDGE EN SEÑALES DE CLOCK
- No se recomienda mezclar tipos de eventos en la lista sensitiva

```
always @(posedge a or b or c) //MAL!
```

tratar de que se ejecute el always bajo una condicion

Descripción Algorítmica:

Bloque Always, combinacionales y secuenciales

- Always usado para lógica combinacional
 - La lista de sensibilidad debe ser tipo por nivel.
 - Se sintetiza lógica combinacional si la lista de sensibilidad es equivalente a `@(*)`.
- Always usado para lógica secuencial
 - Si la lista de sensibilidad es por flanco, flip-flops son generados.
 - Si la lista de sensibilidad es por nivel, y hay ejecuciones del bloque en las que el valor de una variable no es explícitamente asignado, se genera un latch para dicha variable.

Descripción Algorítmica:

Bloque Always, asignaciones bloqueantes

- La asignación se realiza con “=”.
- Las asignaciones se realizan secuencialmente en el orden en el que aparecen. Una asignación no se realiza hasta que se completa la anterior.
- Usar este tipo de asignaciones para bloques combinacionales.

por más de que sea combinacional las cosas se siguen ejecutando en orden

```
36 always@(*)
37 begin
38     ids = {N_LANES * NB_ID{1'b0}};
39     for( i = 0; i < N_LANES; i = i + 1 )
40         if( locked_ids[i] )
41             ids[(i+1) * NB_ID - 1 -: NB_ID] = i[NB_ID - 1 : 0];
42 end
```

Descripción Algorítmica:

Bloque Always, asignaciones no bloqueantes

- La asignación se realiza con "<=".
- Todas las asignaciones dentro de un bloque suceden en paralelo.
- Se utiliza para modelar lógica secuencial (Flip-Flops).
- Utilizar siempre en bloques always @(posedge clock) flanco positivo del clock

```
43 always@(posedge i_clock)
44     if( i_reset )
45         shift_register <= {2*Nb_LANE{1'b0}};
46     else if( i_valid )
47         shift_register <= { shift_register[Nb_LANE - 1 : 0] , i_data_bus};
```

si es un reset pongo todo en 0 y si es un dato lo shifteo, como buffer, y permito que entre un nuevo registro en la parte baja

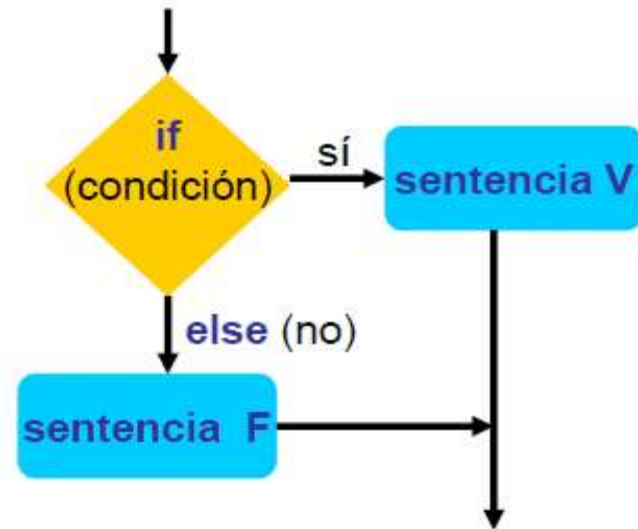
Descripción Algorítmica: If... else

El bloque always

Sentencia condicional **if ... else**

condición es una
expresión con
resultado booleano

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



Descripción Algorítmica: Case 1/2

Descripción Algorítmica: Sentencias Secuenciales

El bloque always

El condicional múltiple *case*

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


Descripción Algorítmica:

Case 2/2

```
1 case (case_expression) // case statement header
2   case_item_1 : begin
3     case_statement_1a;
4     case_statement_1b;
5   end
6   case_item_2 : case_statement_2;
7   default     : case_statement_default;
8 endcase
```

```
1 always @(irq) begin
2   {int2, int1, int0} = 3'b000;
3   casez (irq)
4     3'b1?? : int2 = 1'b1;
5     3'b?1? : int1 = 1'b1;
6     3'b??1 : int0 = 1'b1;
7     default: {int2, int1, int0} = 3'b000;
8   endcase
9 end
```

Descripción Algorítmica: Bucle for

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

Descripción Algorítmica: Bloques Procedurales

- Dos tipos de bloques: initial y always.
- Solo puedo asignar variables del tipo reg en estos bloques. NO WIRE.
- Initial: Se ejecuta una sola vez al inicio de la simulación. En general este bloque **NO ES SINTETIZABLE**. Solo se utiliza en testbenches. **simulación**
- Always: Bloque concurrente que se ejecuta continuamente. Todos los always dentro de un módulo se ejecutan simultáneamente.

```
wire      reset;
reg       flop, sec;

initial
begin
    reset = 1'b1;
    #5
    reset = 1'b0;
end

always @(posedge i_clock)
    if( reset )
        flop <= 1'b0;
    else
        flop <= input_value;

always @(*)
    sec = input_value ^ 1'b0;
```

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
```

Ejemplos

```
module sumador
#(
    // PARAMETERS.
    parameter                NB_DATA    = 160,
    parameter                NB_CNT     = 64
)
(
    // OUTPUTS.
    output wire    [NB_CNT - 1 : 0]    o_counter,
    output wire    o_alarm_cnt,
    // INPUTS.
    input  wire    i_valid,
    // CLOCKS & RESETS.
    input  wire    i_reset,
    input  wire    i_clock
);

    localparam                MAX_COUNT = 8;

    wire    [NB_CNT - 1:0]    next_count;
    reg     [NB_CNT - 1:0]    count;

    assign next_count = count + {NB_CNT-1{1'b0},1'b1};

    always @(posedge i_clock) begin:counter_valid
        if(i_reset || o_alarm_cnt)
            count <= {NB_CNT{1'b0}};
        else if(i_valid)
            count <= next_count;
    end//counter_valid

    assign o_alarm_cnt = (count == MAX_COUNT);

endmodule//sumador
```

Ejemplos: Generate

```
parameter                N_FRAMERS      = 2
parameter                N_FRAMER_LANES = 2
//----- Local parameters
localparam              N_LANES        = N_FRAMERS*N_FRAMER_LANES;
localparam              L0              = 0
localparam              L1              = 1
localparam              L2              = 2
localparam              L3              = 3

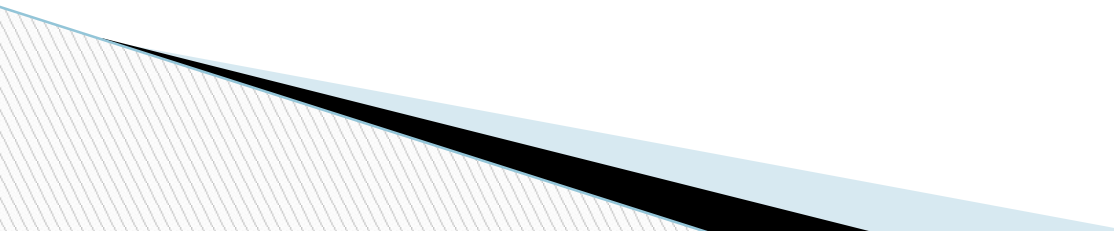
//----- Local variables
genvar                  ii
wire                    [NB_DATA-1:0]  data_g

//----- 200G deinterleave
generate
  for( ii=0; ii < NB_LANE_DATA; ii=ii+1 )
  begin: forgen_interleave
    assign data_g[ii + L0*NB_LANE_DATA] = i_data[ii*N_LANES + L0];
    assign data_g[ii + L1*NB_LANE_DATA] = i_data[ii*N_LANES + L1];
    assign data_g[ii + L2*NB_LANE_DATA] = i_data[ii*N_LANES + L2];
    assign data_g[ii + L3*NB_LANE_DATA] = i_data[ii*N_LANES + L3];
  end // forgen_interleave
endgenerate
```

Instanciación de Módulos

si tengo un modulo que depende del resultado de otro módulo, debo instanciarlo dentro del módulo

```
nombre_modulo
# (
    .parametro_modulo      (valor_parametro)
)
nombre_instancia
(
    .nombre_puerto_1      (conexion_1),
    .nombre_puerto_2      (conexion_2)
);
```



Test Benches



instancia lo que hicimos nosotros, excita las entradas y produce una salida, que esperamos que sea la que programamos nosotros

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)

clock, reset, variables internas, variables y un monitor para corroborar las salidas

- 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 `initial` para generar el stimulus y evaluar las salidas

podemos utilizar todo lo que queramos, debemos excitar con variables y ver los resultados en el monitor

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 estímulo del componente a testear.

```
initial
```

```
// Stimulus
```

```
begin
```

```
    SEL = 0;
```

```
    A = 0;
```

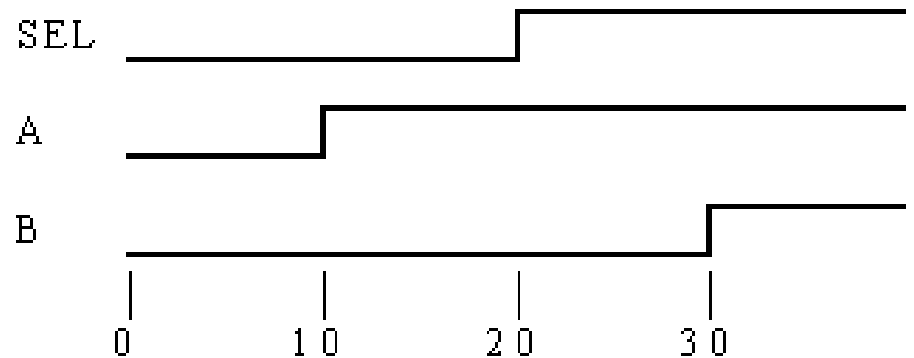
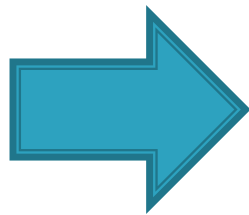
```
    B = 0;
```

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

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

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

```
end
```

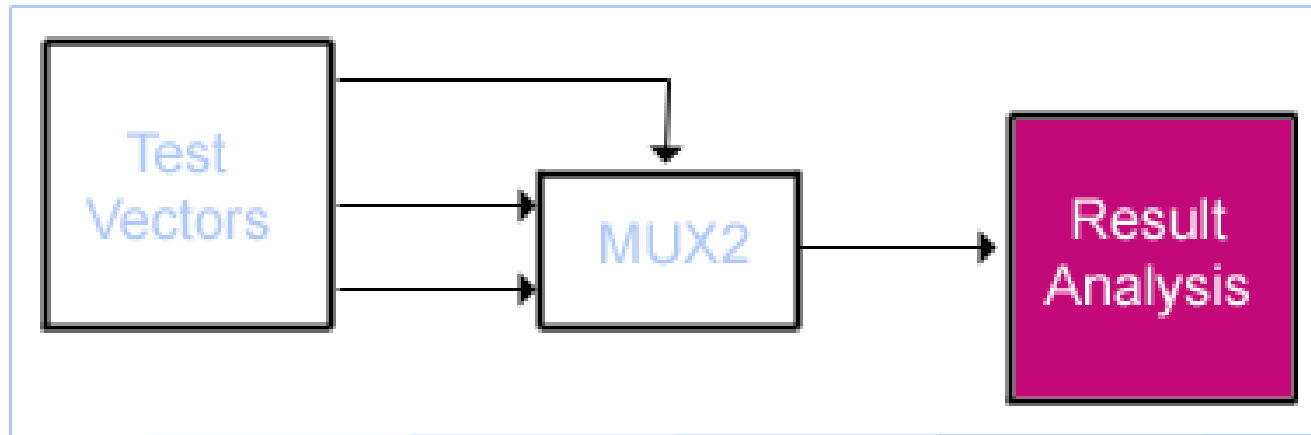


Test Benches:

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 análisis 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

