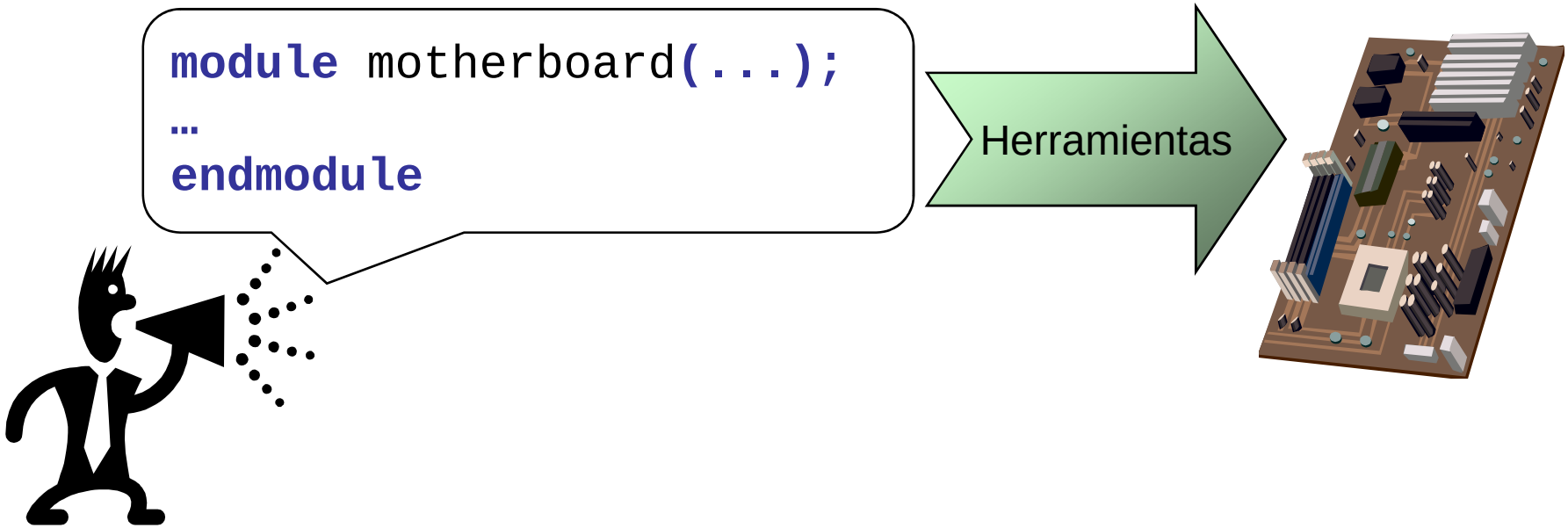
INTRODUCCIÓN A VERILOG

Ing. Pablo Cayuela
pablo.cayuela@gmail.com
Ing. Federico Paredes
fedesor@gmail.com

¿Qué es Verilog?

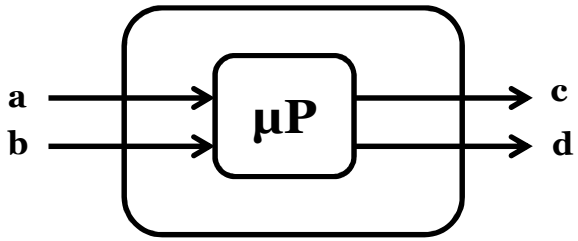
Verilog es un HDL (Hardware Description Language), es decir, un lenguaje de descripción de hardware.
Es un lenguaje desarrollado a principios de los ‘80 en la industria electrónica de los EEUU, para la documentación, modelado y simulación de circuitos digitales. Con el tiempo, también adquirió importancia en la síntesis de estos circuitos, siendo hoy una de sus principales aplicaciones.

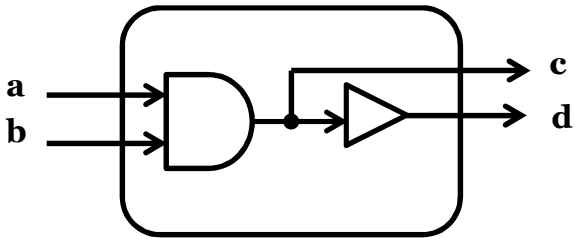


Algunas características importantes de Verilog

- Verilog es un lenguaje concurrente, sus sentencias se ejecutan en paralelo.
- Es un lenguaje extenso con más de 100 palabras reservadas.
- Posibilita la integración de diferentes herramientas de CAD.
- Posee una sintaxis amplia y flexible, permitiendo distintos tipos de descripción.
- Permite dividir un diseño en unidades más pequeñas, incrementando la modularidad de éste.
- Se encuentra estandarizado por el IEEE Std 1364-1995/2001/2005.
- Permite diseñar, modelar y simular un sistema desde altos niveles de abstracción hasta niveles muy cercanos al hardware final.

Concurrencia Vs. Secuencia





`void function (bool a, bool b)`
`{ ...`
`  c = a & b;`
`  d = !c;`
`}`
	`module(input a,b; output c,d);`
	
	`  c = a & b;`
	`  d = ~c;`
	`endmodule`

`void function (bool a, bool b)`
`{ ...`
`  d = !c;`
`  c = a & b;`
`}`
	`module(input a,b; output c,d);`
	
	`  d = ~c;`
	`  c = a & b;`
	`endmodule`

Importancia de las Herramientas

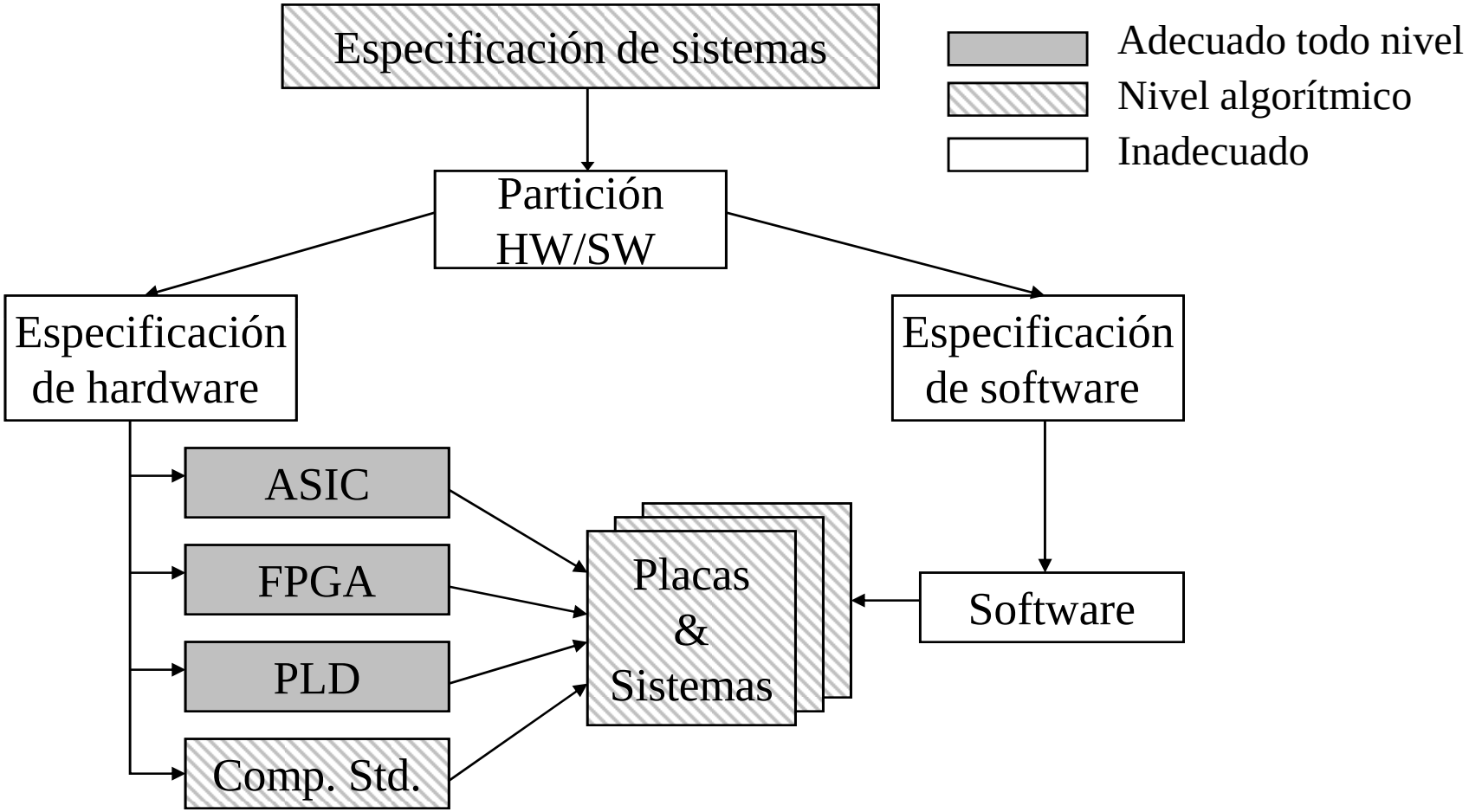
Flujos de diseño

- Especificación
- Simulación
- Implementación

Herramientas

- Compilador
- Simulador
- Sintetizador

Áreas de aplicación de Verilog



Introducción a Verilog

Verilog para síntesis

No siempre una descripción en Verilog es sintetizable.
Sólo un subconjunto reducido del lenguaje se utiliza para sintetizar circuitos.

Introducción a Verilog

Verilog para simulación

El lenguaje Verilog fue diseñado inicialmente como medio para el modelado y la simulación de sistemas digitales, por lo cual no impone limitaciones en cuanto a la simulación.

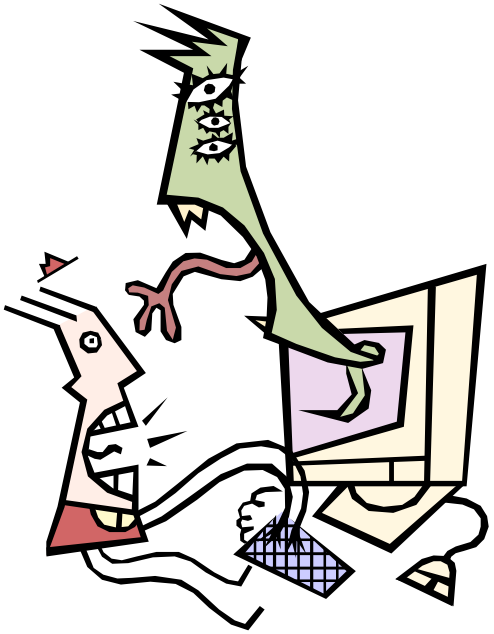
Adelantos para ir perdiendo el miedo:

• Las sentencias terminan en punto y coma (**;**), con la excepción de la finalización de estructuras.

• Los comentarios son tanto como en *C* `/* que pueden extenderse por varias líneas */` o como en *C++* `// que solo ocupan una línea.`

• Los espacios en blanco, tabulaciones y fines de línea son ignorados.

• Las sentencias que componen un bloque se agrupan entre **`begin`** y **`end`**.



Adelantos para ir perdiendo el miedo:

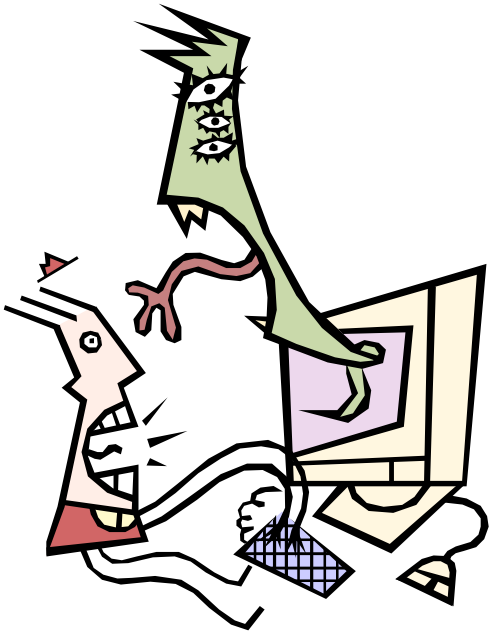
• Los números se consideran escritos en base 10.

• Pueden especificarse otras bases y longitudes. El formato es <*long. en bits*>'<*base*><*valor*>.

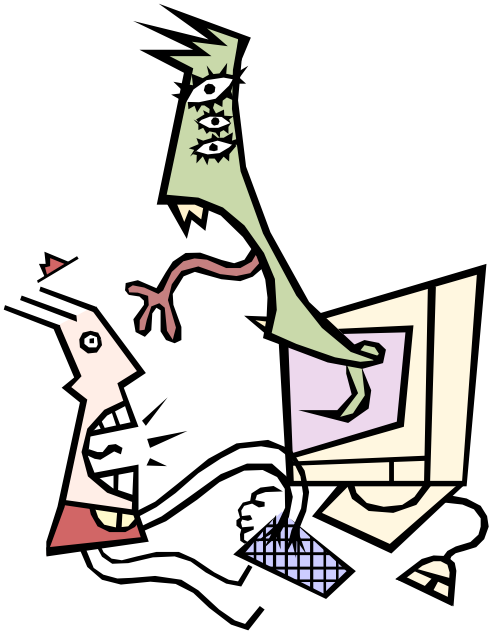
- **4'o13** octal, long. 4 bits
- **5'b10111** binario, long. 5 bits
- **5'd23** decimal, long. 5 bits
- **5'h17** hexadecimal, long. 5 bits

• Se puede emplear guión bajo para ayudar con la lectura de números largos.

- **12'b000_111_010_100**



Adelantos para ir perdiendo el miedo:



• Todo identificador debe contener solo letras, números y signo *pesos* (**$**) y *guión de subrayado* (**_**). Sólo pueden comenzar con *letras* o *guión de subrayado*.

- **id_bueno**
- **i_d_bueno**
- **_id_bueno**
- **id$bueno**
- **3id_errado**
- **$id_errado**

• Mayúsculas y minúsculas no son equivalentes.

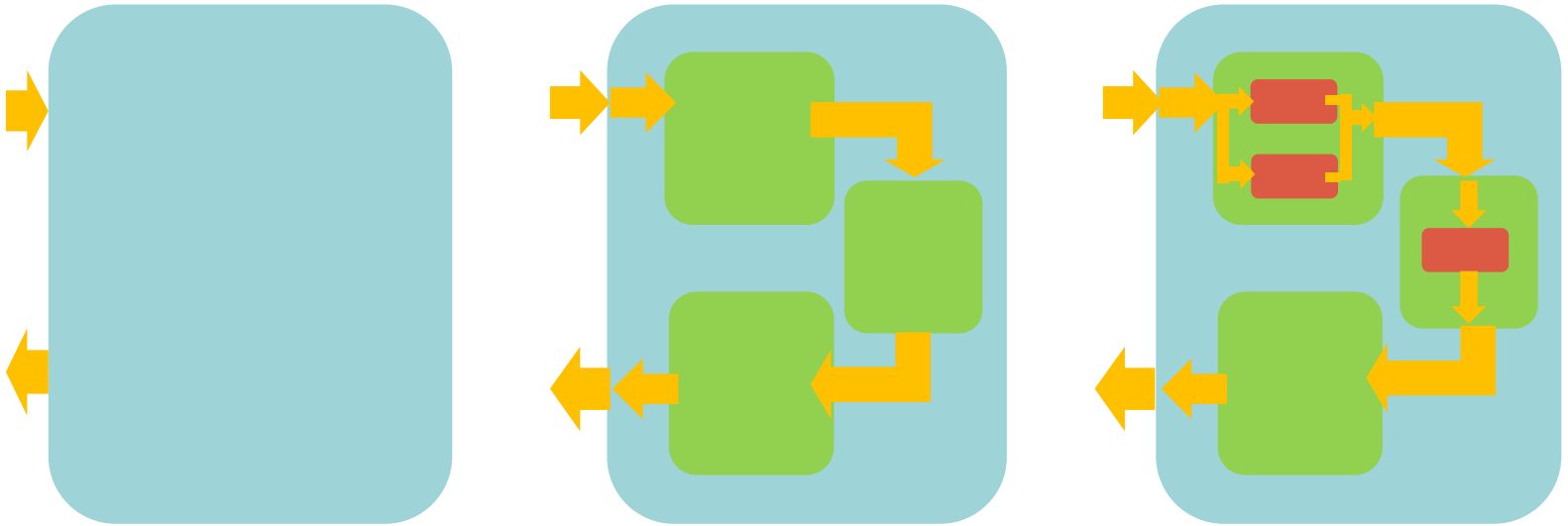
- **data ≠ Data**
- **CLOCK ≠ clock**

Diseño de hardware: El concepto de caja negra

Todo diseño de hardware comienza como un bloque definido por sus entradas, sus salidas y la relación entre estas. En Verilog esa caja negra se denomina **`module`** (módulo).

Diseño Top Down

Consiste en comenzar con la caja negra que representa al sistema y refinar la implementación hasta describir el nivel más bajo.



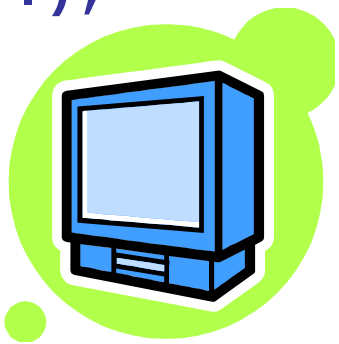
1. 2. 3.

El módulo

Todo diseño en Verilog comienza con un módulo:

```
module celular (...);
…
endmodule
```


```
module televisor (...);
…
endmodule
```



```
module procesador (...);
…
endmodule
```

El módulo: sus partes

El primer paso en la construcción de un módulo, es colocarle un nombre y definir los puertos de entrada/salida y sus parámetros.

Puertos y parámetros

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

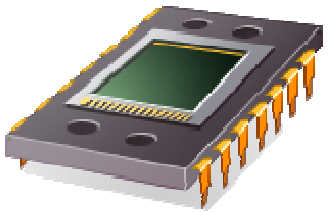
Constantes y variables

```
//sentencias declarativas
  //constantes
  //variables
```

Procesos concurrentes

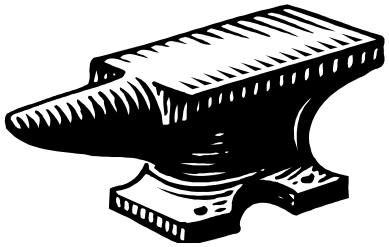
```
//sentencias concurrentes
  //lista de procesos
```

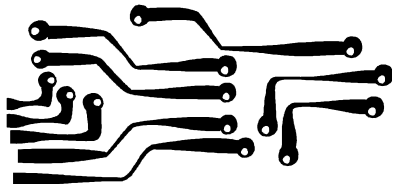
```
endmodule
```

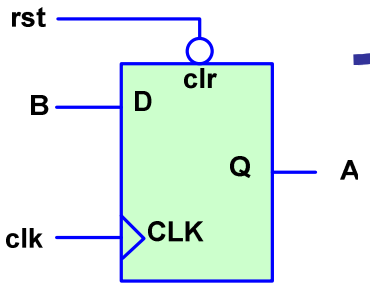


El módulo: parámetros y declaraciones

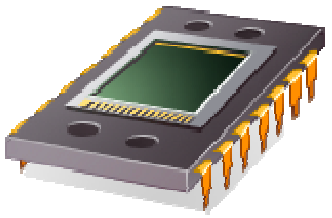
A continuación se muestra un ejemplo de declaración de parámetros, constantes, conexiones y variables.





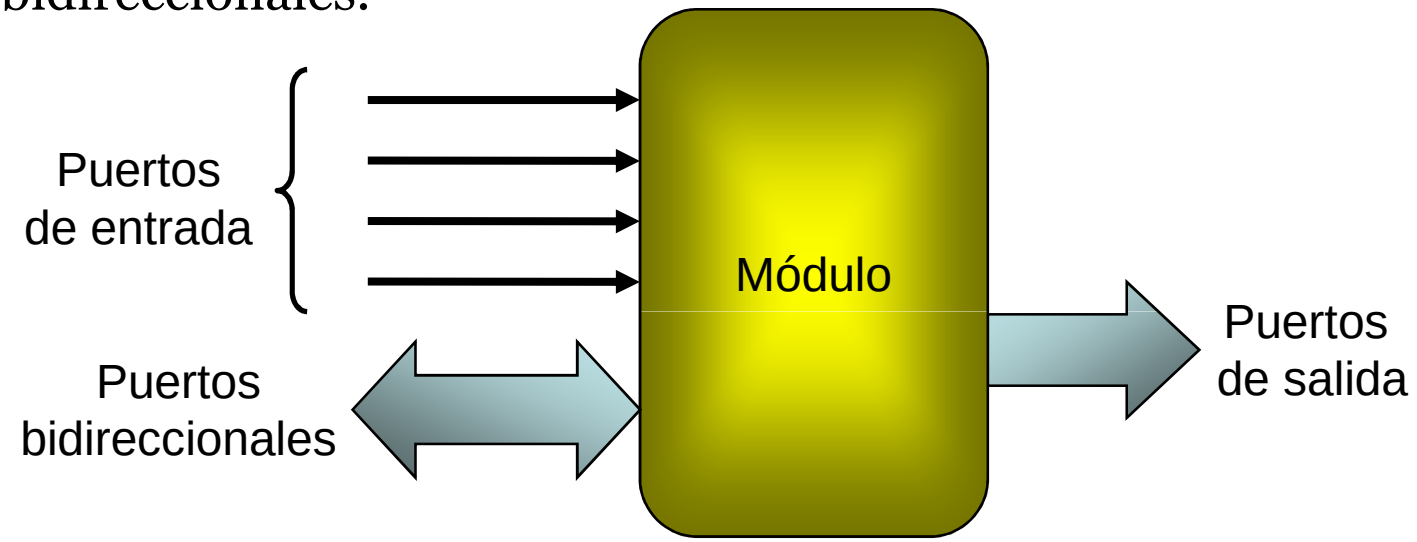



```
module chip
 #(parameter package = "dip")
  ( //puertos E/S
  );

 localparam constante_1 = 3.14;
 wire conexion_1;
 reg  variable_x;

 ...

endmodule
```



El módulo: Puertos

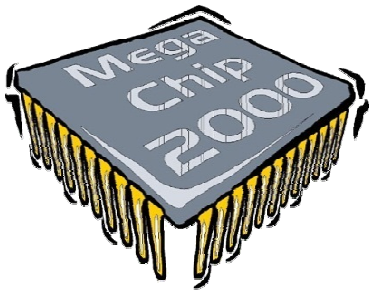
Las conexiones externas pueden ser puertos de entrada, de salida o bidireccionales.



- En los puertos de entrada no se puede escribir (son de solo lectura), y se especifican como **`input`**.
- De los puertos de salida no se puede leer (son de solo escritura), y se especifican como **`output`**.
- Los puertos bidireccionales se pueden leer y escribir, y se especifican como **`inout`**.

El módulo: Puertos

Junto con los parámetros también se declaran los puertos del módulo.




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

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

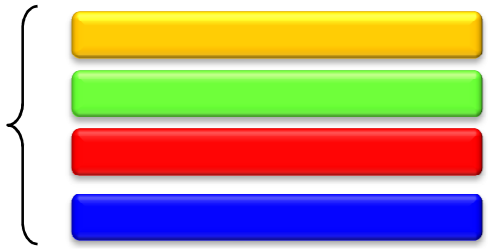
Con **input** y **output** declaramos los puertos de nuestro módulo.

El módulo: 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
  );

  ...

endmodule
```

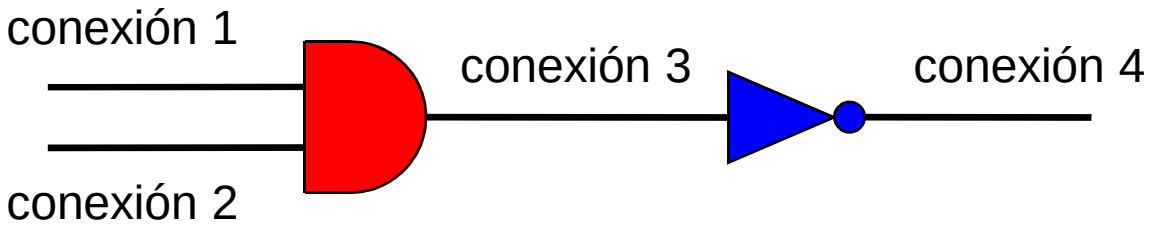
Procesos concurrentes



Todos simultáneos

Conexiones (*Nets*)

Las conexiones (*Nets*) son un tipo de declaración que modela en Verilog las conexiones eléctricas en un circuito esquemático.

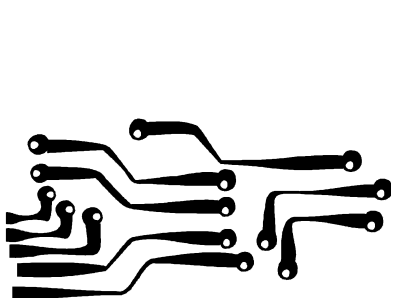


Existe en Verilog una gran variedad de *Nets*: **`wire`**, **`tri`**, **`wand`**, **`wor`** y otras. Cada una de estas modela situaciones de circuitos a nivel de compuertas en situaciones específicas.
Nosotros emplearemos solamente la conexión de tipo **`wire.`**

Conexiones (*Nets*): declaración

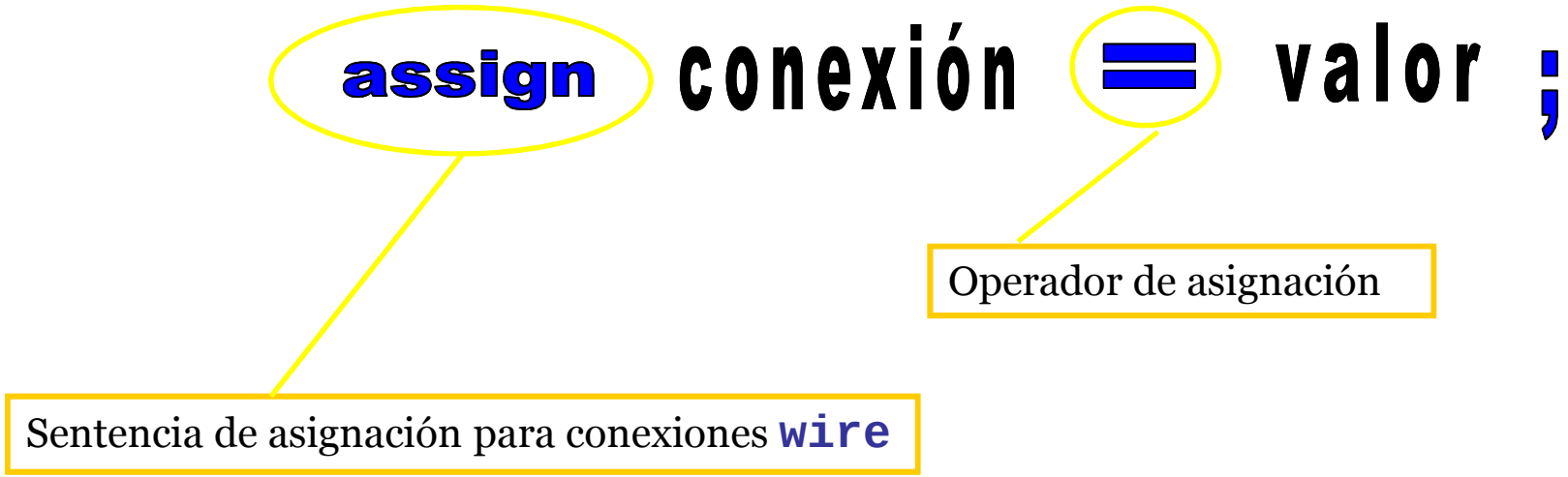
Toda *Net* debe declararse dentro del módulo previamente a su uso :

```
module chip (...);

 wire conexion_3;
 wire conexion_4;

 ...

endmodule
```



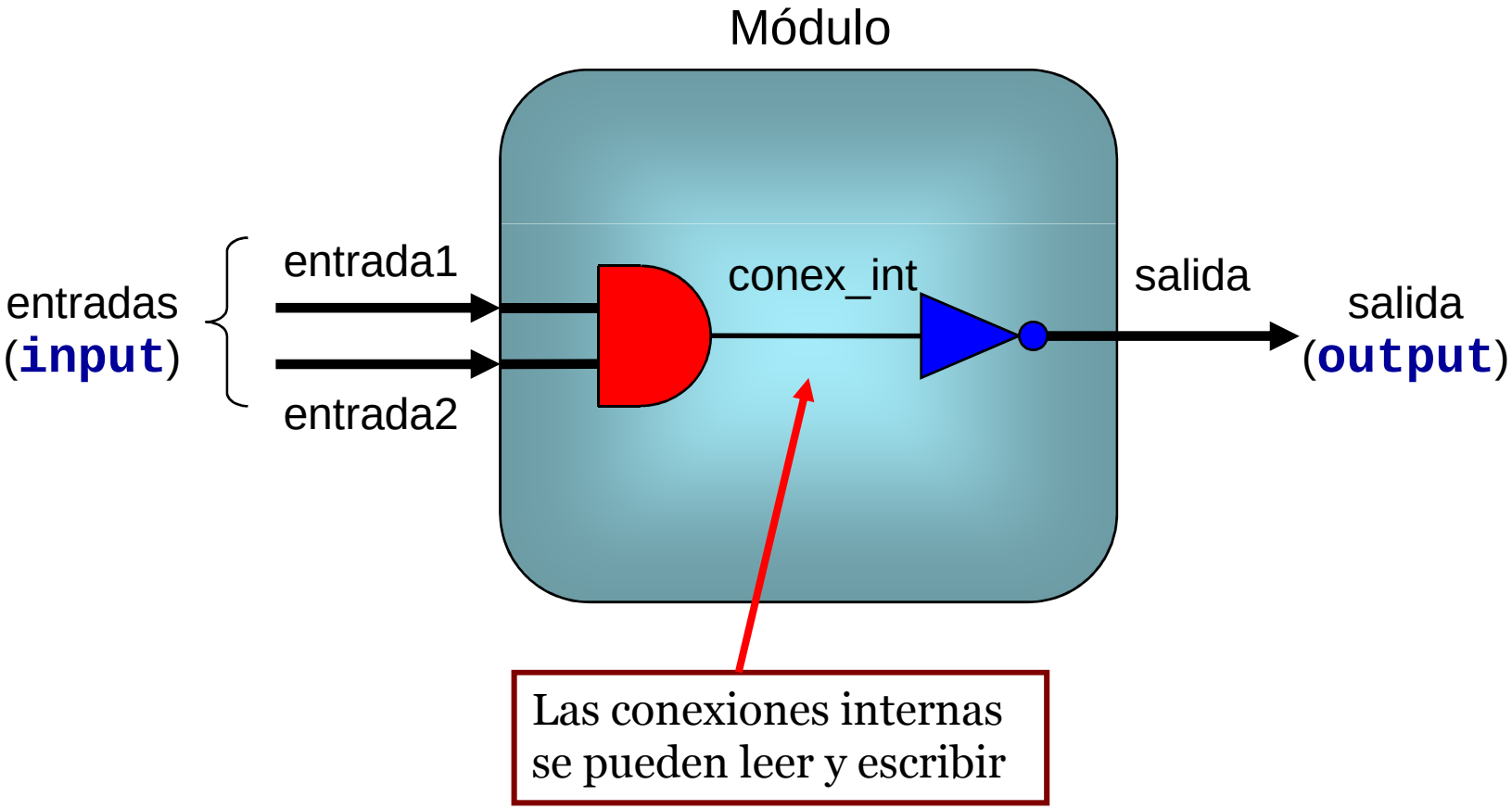
Conexiones (*Nets*): asignación de fuentes de valor

Una vez declarada una *Net* de tipo `wire`, debe realizarse una asignación conocida como *asignación continua*, que sea la fuente de su valor (es decir que posea un *driver*).



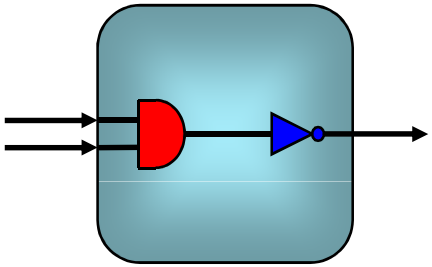
Conexiones (*Nets*): externas e internas

Las conexiones se pueden clasificar en internas y externas. Las externas constituyen puertos del módulo: **`input`** - **`output`** - **`inout`**.



Conexiones (*Nets*): externas e internas

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

Introducción a Verilog

Variables

En Verilog también existen variables y se denominan registros (*Register variables*). 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.

- reg

- integer 0 1 2 3 ...

- time 10 ns 5 ms

- real π = 3,14159265358979...

Introducción a Verilog

Variables

Una variable con tipo de dato **reg**, puede reflejar en ciertos casos un registro o memoria y en otros, solo una conexión eléctrica.

La declaración de variables de tipo **reg** es similar a la de **wire**.

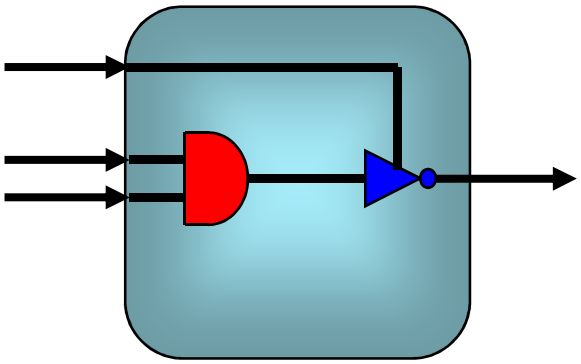
```
module chip (...);
...

 reg conexion_4;
 reg variable_2;
 reg registro_1;

...

endmodule
```

Lógica de 4 estados

¿Son suficientes los valores ***0*** y ***1*** para modelar los sistemas digitales?

Supongamos que deseamos agregar a nuestro circuito la capacidad de poner su salida en alta impedancia.

¿Cómo modelamos tal situación?



¿y Z?

Introducción a Verilog

Lógica de 4 estados

En Verilog se contempla el modelado de un sistema digital en función de 4 valores o situaciones posibles de funcionamiento de entradas, salidas y variables internas.

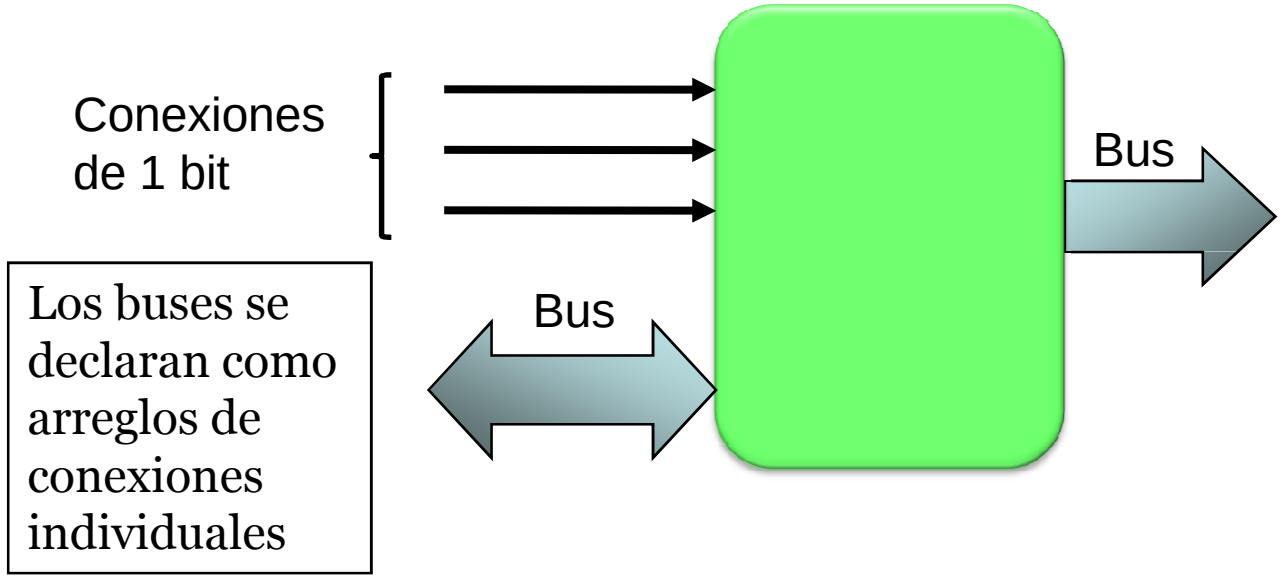
Valores de tipos de datos `wire` y `reg`

`x`	**Desconocido**
`0`	**Nivel lógico bajo o condición falsa**
`1`	**Nivel lógico alto o condición verdadera**
`z`	**Alta impedancia**

Introducción a Verilog

Buses

Se pueden emplear buses declarándolos como conexiones de varios bits.



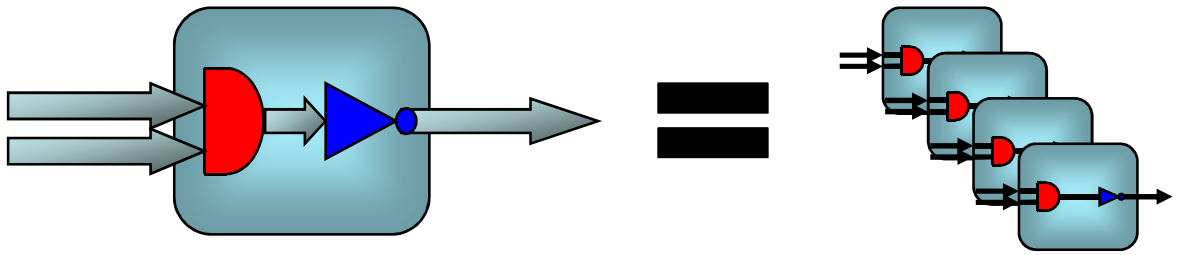
Varias líneas de conexión de 1 bit se pueden unir (concatenar) para formar un bus y éste también se puede descomponer en conexiones de un bit.

Buses

En la declaración de nuestra compuerta, utilizamos conexiones externas e internas de 1 bit de ancho. Si quisiéramos aumentar la cantidad de bits de nuestra compuerta, podríamos utilizar un bus en la declaración de conexiones:

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

  wire        [3:0] conex_int;
```

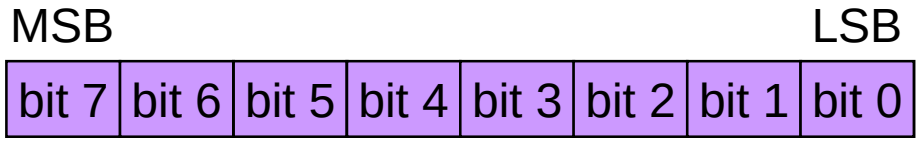
Nota: si no se especifica otra cosa, **input** y **output** generan conexiones de tipo **wire** de 1 bit.



Buses

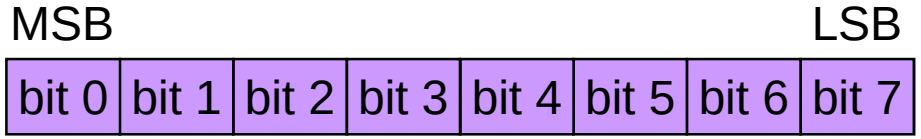
Los buses pueden realizarse tanto con conexiones (**wire**) como con variables (**reg**). En la declaración de un bus, el valor y el orden de los índices de los bits, determina el tamaño del bus y la ubicación del bit más significativo.

```
wire [7:0] bus_A;
reg  [7:0] data_A;
```



Tenga cuidado al emplear ambos ordenamientos en un mismo diseño

```
wire [0:7] bus_B;
reg  [0:7] data_B;
```



Operadores

Al igual que en otros lenguajes, existe una importante cantidad de operadores. A continuación se detallan algunos.

Operadores aritméticos

Suma

```
a = b + c;
```

División

```
a = b / c;
```

Resta

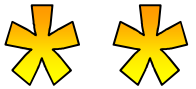
```
a = b - c;
```

Módulo

```
a = b % c;
```

Multiplicación

```
a = b * c;
```



Potenciación

```
a = b ** c;
```

Introducción a Verilog

Operadores binarios a nivel de bits y de reducción

not (negación, inversión)

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

and (y)

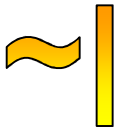
```
x = y & z; //nivel bits
w = & u; //reducción
```

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)

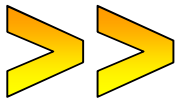
```
x = y ^ z; //nivel bits
w = ^ u; //reducción
```

xnor (no o exclusivo)

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

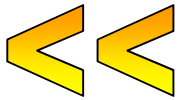
Introducción a Verilog

Operadores binarios de desplazamiento



Right shift (desplazamiento a la derecha)

```
x = z >> 3; /* si z es 1011_0011
               x será 000 1_0110
               es decir rellena
               con ceros*/
```

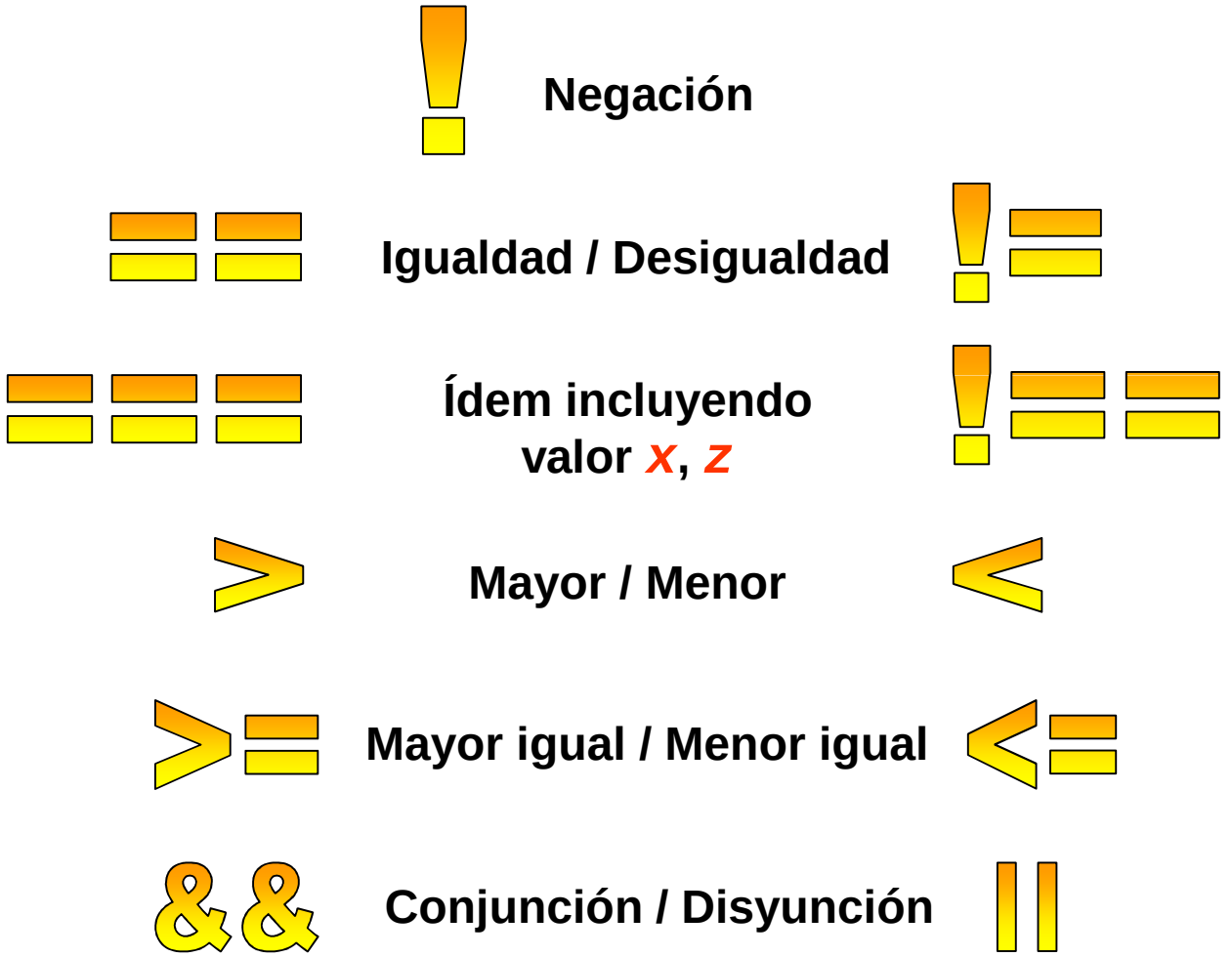


Left shift (desplazamiento a la izquierda)

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

Introducción a Verilog

Operadores lógicos y relacionales de resultado booleano



Operador de concatenación

{ , } **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 */
```

El operador de concatenación y el de replicación pueden anidarse sin problemas combinando sus dos funciones.

Extensiones aritméticas para enteros (signed variables)

- Los tipos de datos **reg** y **wire** pueden declararse como **signed**.

```
reg signed [63:0] data;
wire signed [11:0] address;
```

- Los retornos de función pueden declararse como **signed**.

```
function signed [128:0] alu;
```

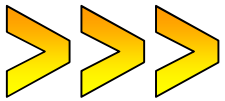
- Los números pueden declararse como **signed**.

```
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**.

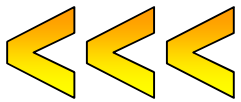
Introducción a Verilog

Operadores aritméticos de desplazamiento con signed



Right shift (desplazamiento a la derecha)

```
x = z >>> 3; /* si z es 1011_0011
                x será 111 1_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 */
```

Introducción a Verilog

Sumergiéndose en Verilog

En Verilog existen tres estilos principales de descripción que nos interesan:

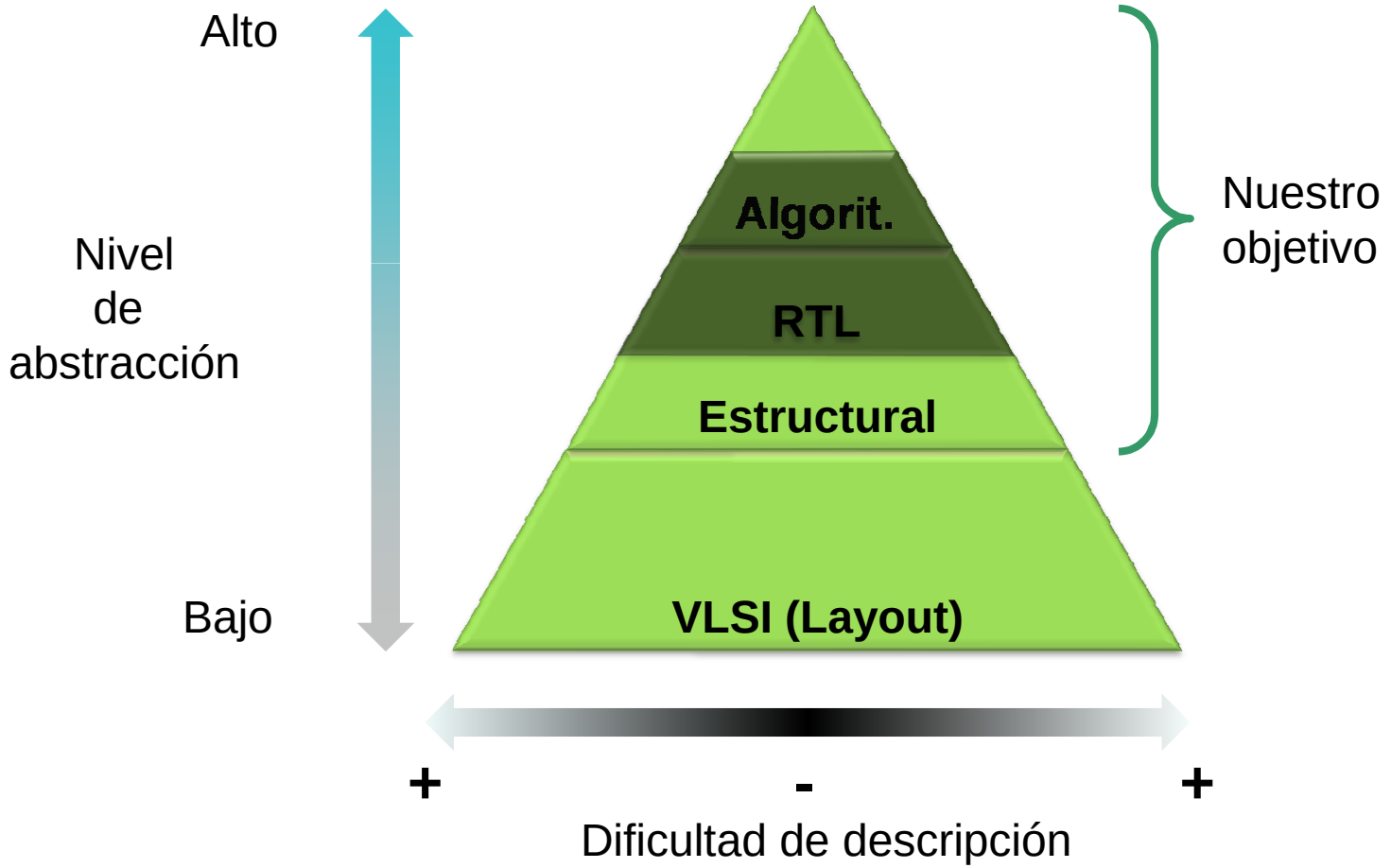
- Descripción Estructural
- Descripción Algorítmica
- Descripción de Flujo de datos (RTL)

Es importante destacar que esta distinción de estilos es de carácter informal. En la práctica es común el utilizar una combinación de estilos.

Introducción a Verilog

Sumergiéndose en Verilog

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

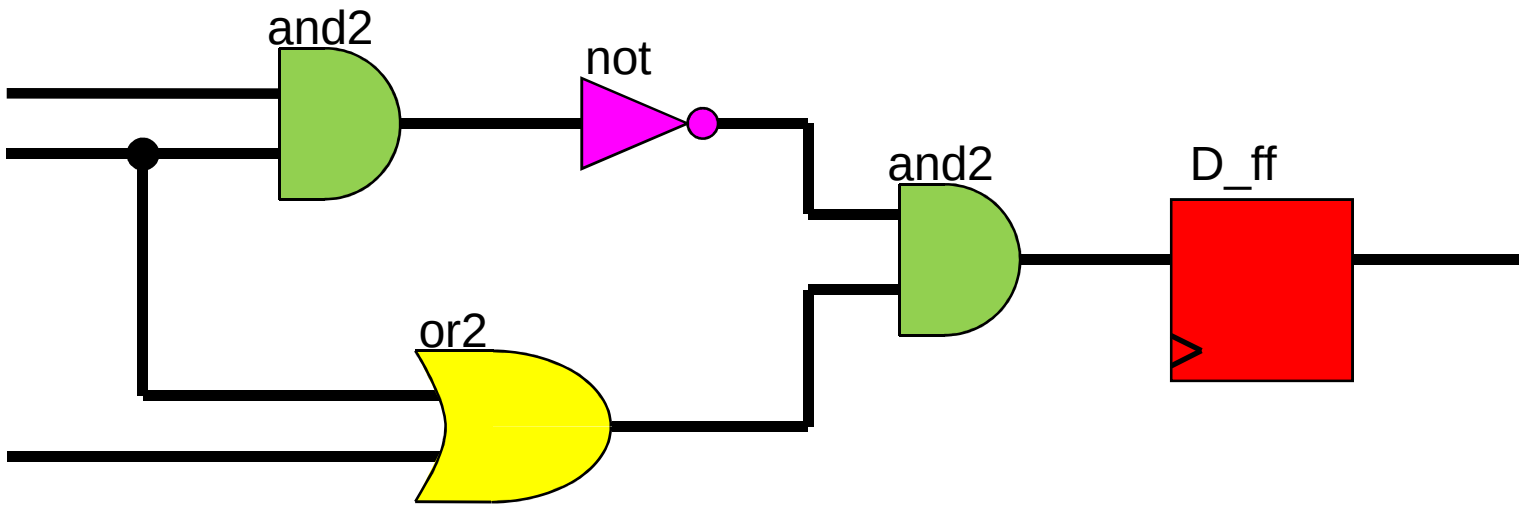


Introducción a Verilog

Sumergiéndose en Verilog

Tipos de descripciones de hardware

- Descripción Estructural



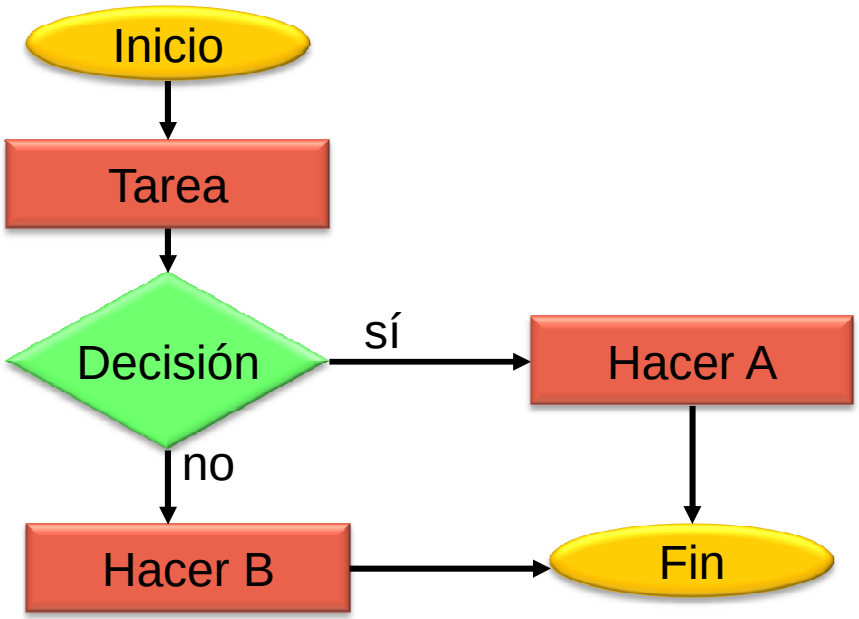
Una descripción estructural de un diseño emplea componentes previamente definidos y los interconecta de manera adecuada.

Introducción a Verilog

Sumergiéndose en Verilog

Tipos de descripciones de hardware

- Descripción Algorítmica



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

Introducción a Verilog

Sumergiéndose en Verilog

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
```

Introducción a Verilog

Sumergiéndose en Verilog

Tipos de descripciones de hardware

- 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.

Descripción Estructural

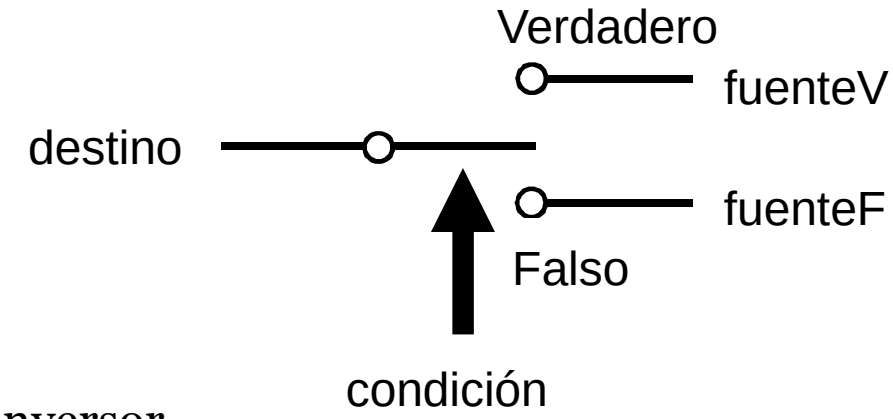
Descripción Estructural

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

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

fuente1 y ***fuente2*** pueden ser expresiones, variables o constantes
condición es una expresión con resultado booleano.

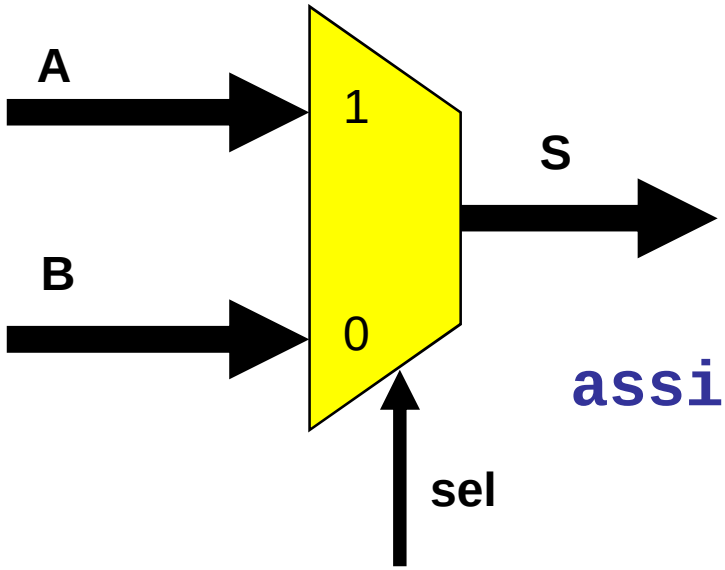
Funciona como un interruptor inversor



Descripción Estructural

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

Es ideal para describir multiplexores 2x1:




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

Instanciación de módulos

- Nombre de instancia de módulo comienza con "**u**_", y luego el nombre del módulo
- Parámetros y puertos instanciados con lista nominada (explícita)
- Un solo puerto por renglón, con sangrado idéntico al primer paréntesis:

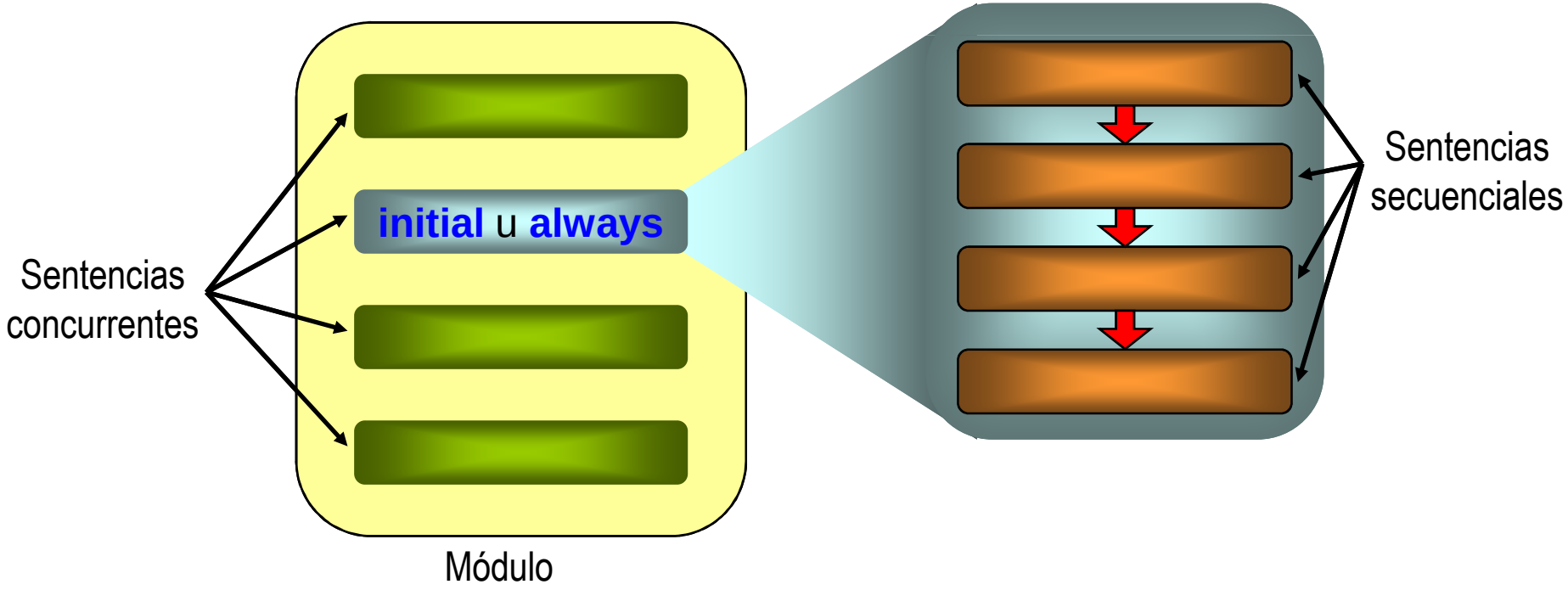
```
nombre_modulo
#(
 .parametro_modulo      (valor_param_instancia)
)
u_nombre_instancia
(
 .i_entrada1_modulo     ( i_ent1_instancia ),
 .i_entrada2_modulo     ( i_ent2_instancia ),
 .o_salida_modulo       ( o_sal_instancia )
);
```

Descripción Algorítmica

Descripción Algorítmica - Sentencias Secuenciales

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.



Bloques `initial` y `always`

En estos bloques se pueden escribir sentencias secuenciales solamente.

```
initial
begin
 /*sentencias
    secuenciales*/
end
```

- Inicia cuando arranca la simulación.
- Termina cuando se alcanza el fin del bloque (**end**).
- Ideal para la generación de estímulos (simulación).

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

- Inicia cuando arranca la simulación.
- Reinicia cuando se alcanza el fin del bloque (**end**).
- Ideal para el modelado y la especificación de hardware (síntesis).

Descripción Algorítmica - Sentencias Secuenciales

El bloque `always`*: su estructura*

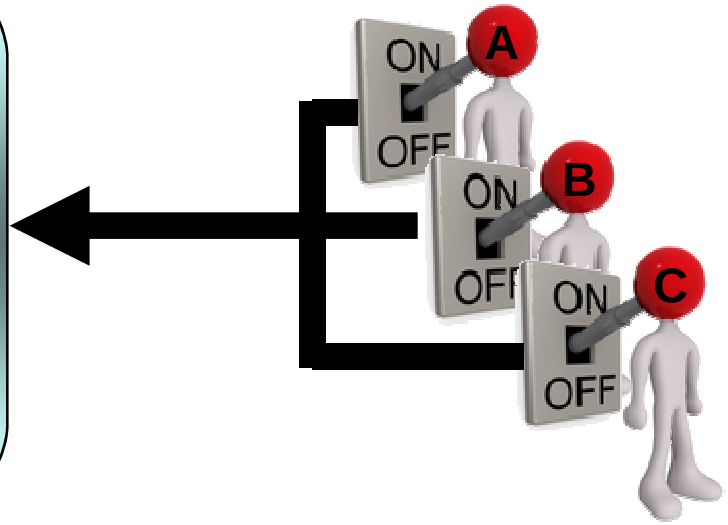
```
always @(lista_de_sensibilidad)


begin [: nombre_bloque]

  /* sentencias
    secuenciales*/

end
```

Descripción Algorítmica - Sentencias Secuenciales

El bloque `always`

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.

```
always @ (A, B, C)
begin

...

end
```



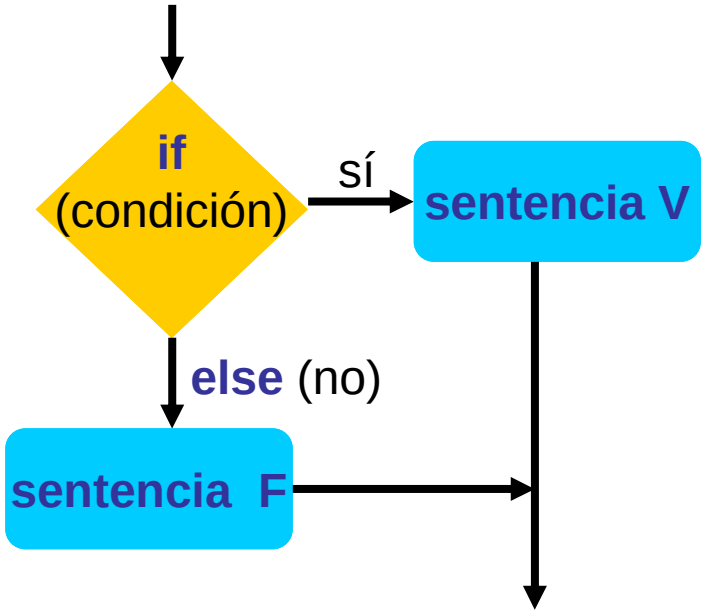
Descripción Algorítmica - Sentencias Secuenciales

El bloque `always`

Sentencia condicional **if ... else**

condición es una expresión con resultado booleano

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

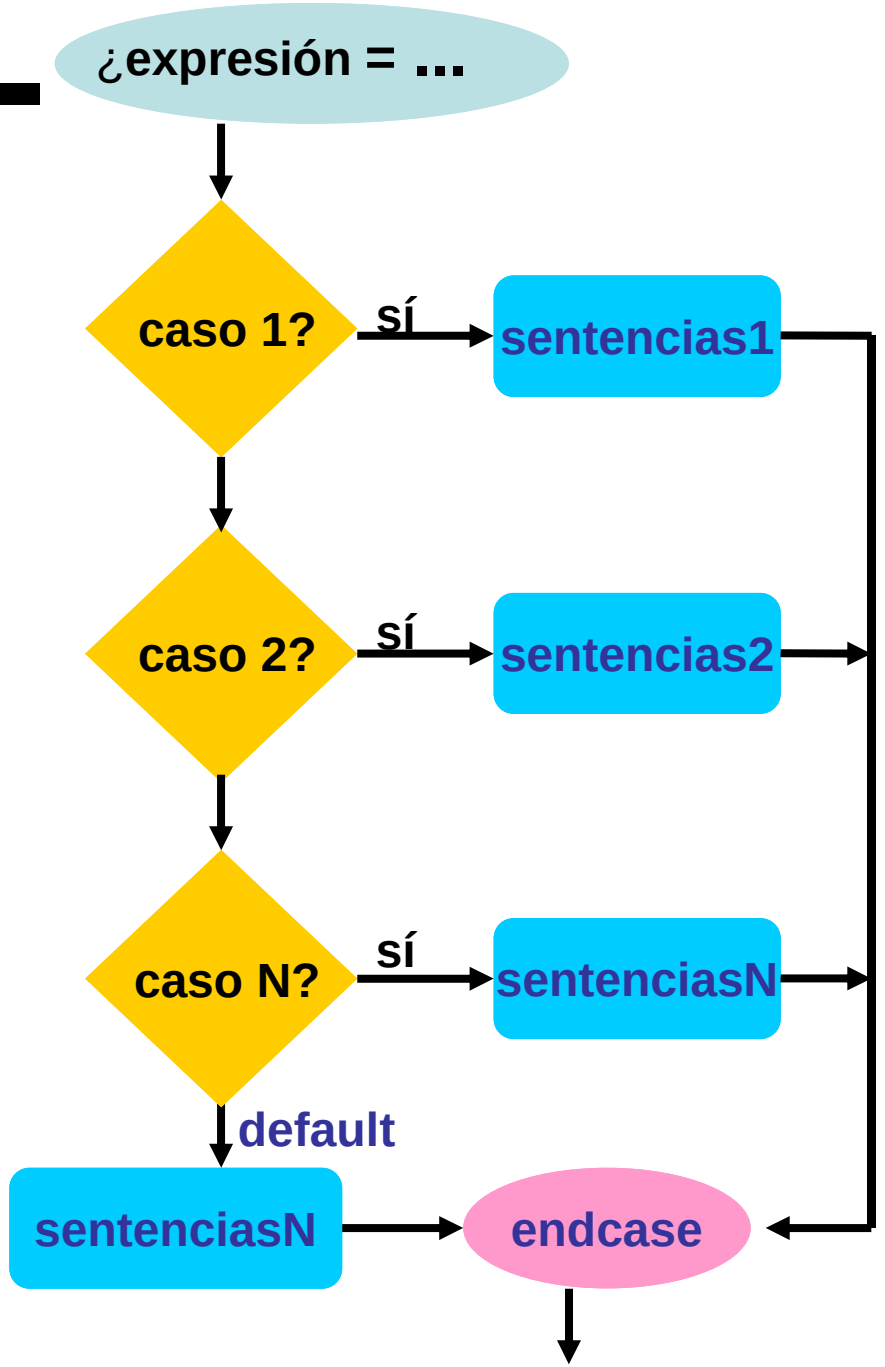


Descripción Algorítmica: Sentencias Secuenciales

El bloque `always`

El condicional múltiple `case`

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



Descripción Algorítmica - Sentencias Secuenciales

El bloque `always`

El 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

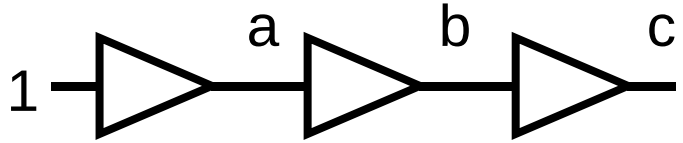
- variable **<=** sentencia
- Las entradas se almacenan cuando se activa el bloque secuencial.
- Las sentencias y asignaciones se ejecutan en paralelo.
- Modela *flip-flops*, *latches* y registros.

NO MEZCLE AMBAS ASIGNACIONES EN UN BLOQUE

Descripción Algorítmica: asignación secuencial

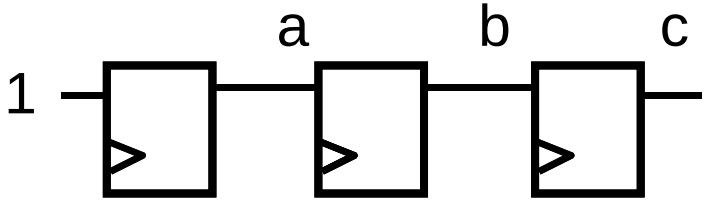
- RHS de bloqueantes tomada de conexiones (wires)

```
a = 1;
b = a;
c = b;
```



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

```
a <= 1;
b <= a;
c <= b;
```

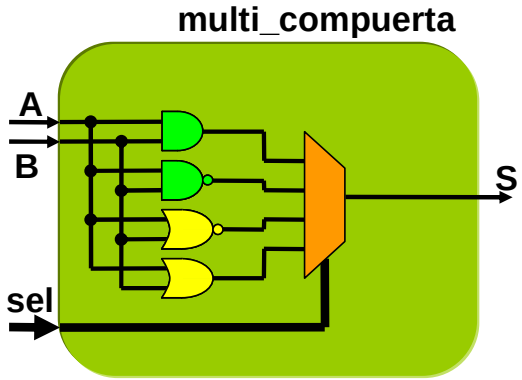


El resultado de una asignación antibloqueante luce como un *latch*

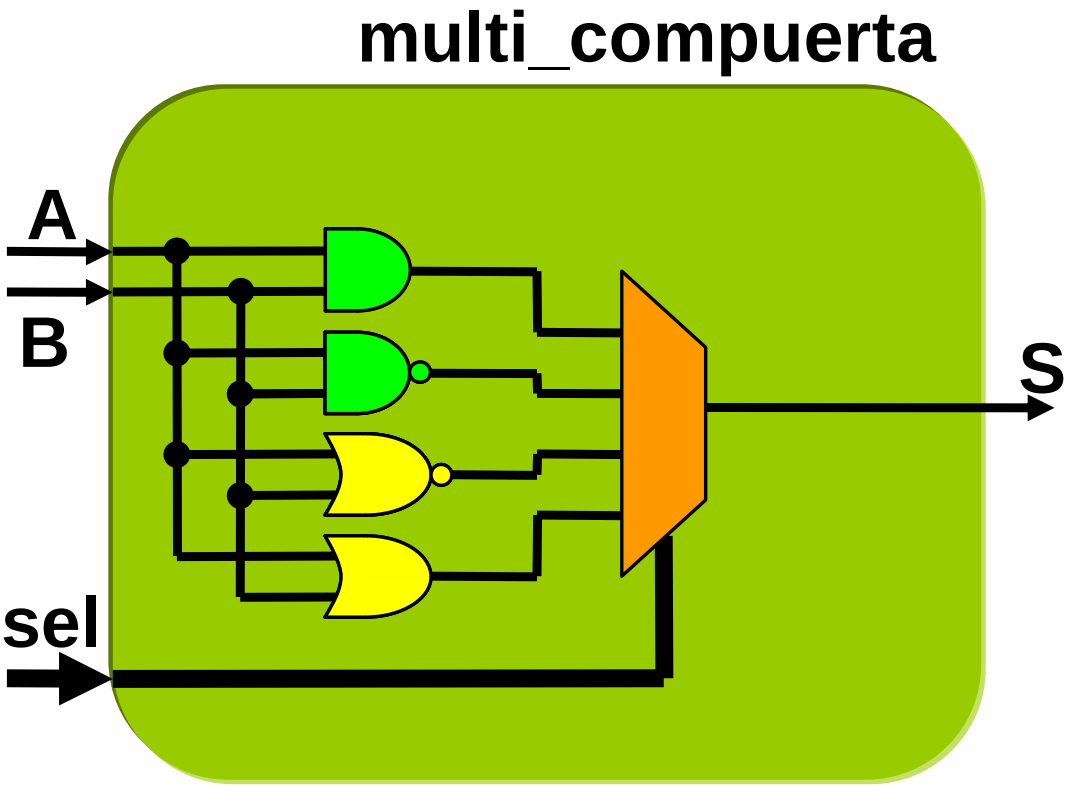
Introducción a Verilog

Trabajos prácticos

1. Multi-compuerta.



2. Half adder y Full adder.
3. Generador y verificador de paridad.
4. Multiplicador con saturación y redondeo.
5. Slicer.


multi_compuerta
A
B
sel
S

Inferencia de componentes

Descripción Algorítmica: circuitos secuenciales

```
always @(variable1 or variable2 or ..)
 begin
    ...
 end
```

se ejecuta cada vez que una variable cambia

```
always @(posedge clk)
 begin
    ...
 end
```

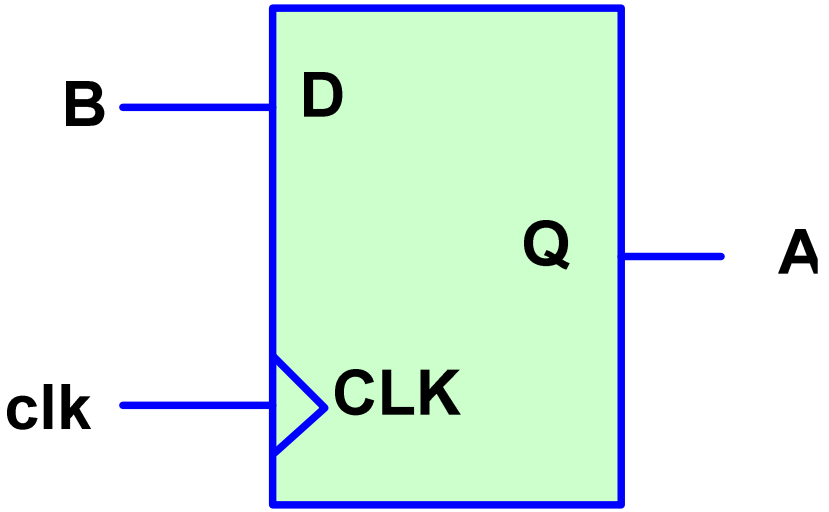
se ejecuta cada vez que clk cambia de *0* a *1*

```
always @(negedge clk)
 begin
    ...
 end
```

se ejecuta cada vez que clk cambia de *1* a *0*

Descripción Algorítmica: circuitos secuenciales

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

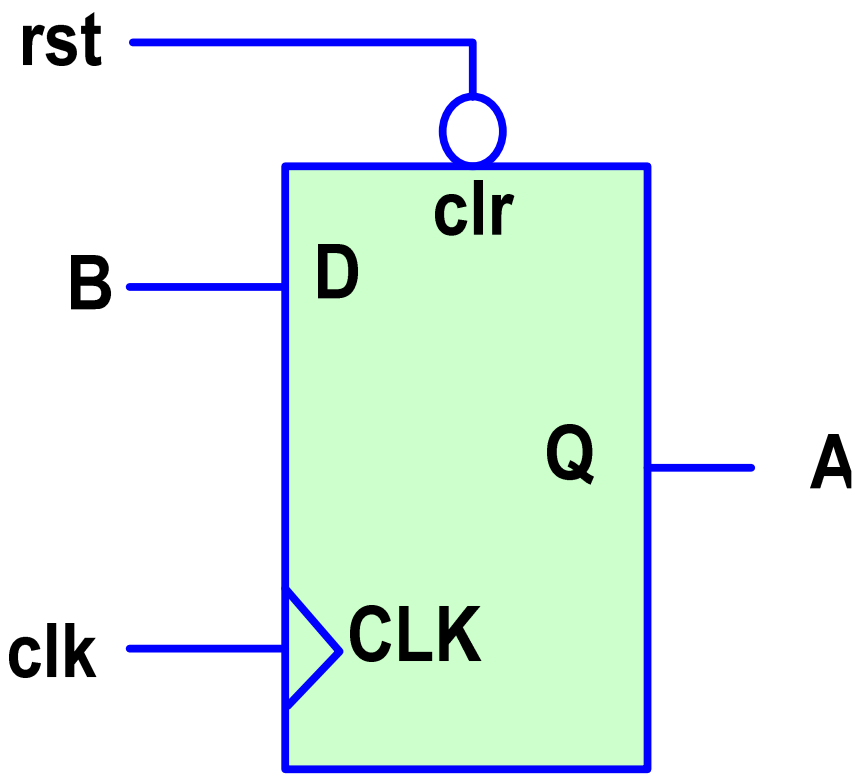


Flip-Flops

Descripción Algorítmica: circuitos secuenciales

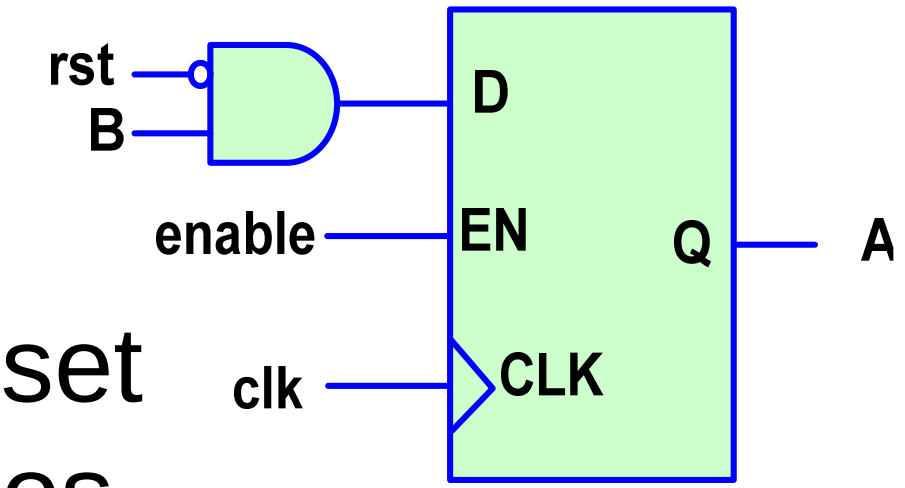
```
always@(posedge clk or negedge rst)
begin
  if (!rst) a<=0;
  else a<=b;
end
```

Flip-Flop D con reset asíncrono



Descripción Algorítmica: circuitos secuenciales

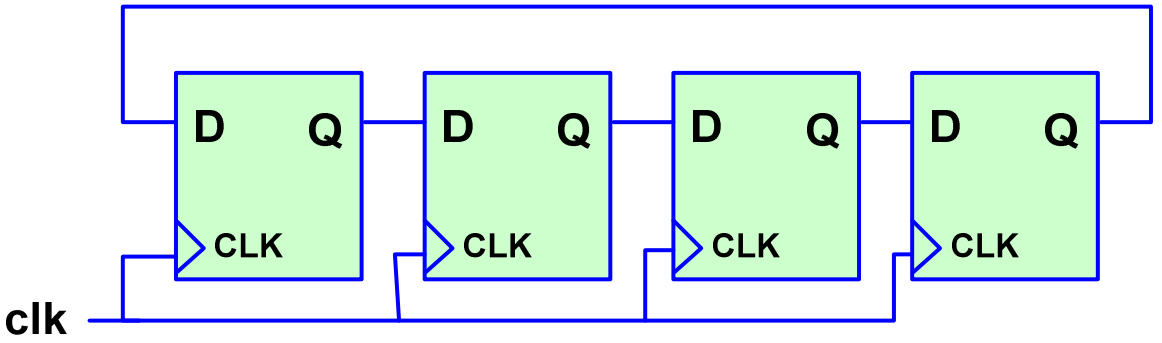
```
always@(posedge clk)
begin
  if (rst) a<=0;
  else if (enable) a<=b;
end
```



Flip-flop D con reset y enable síncronos

Descripción Algorítmica: circuitos secuenciales

```
reg [3:0] Q;
always@(posedge clk or posedge rst )
begin
  if (rst) Q<=0;
  else
   begin
    Q <= Q << 1;
    Q[0]<=Q[3];
   end
```



Registros de desplazamiento

Descripción Algorítmica: circuitos secuenciales

Método 1

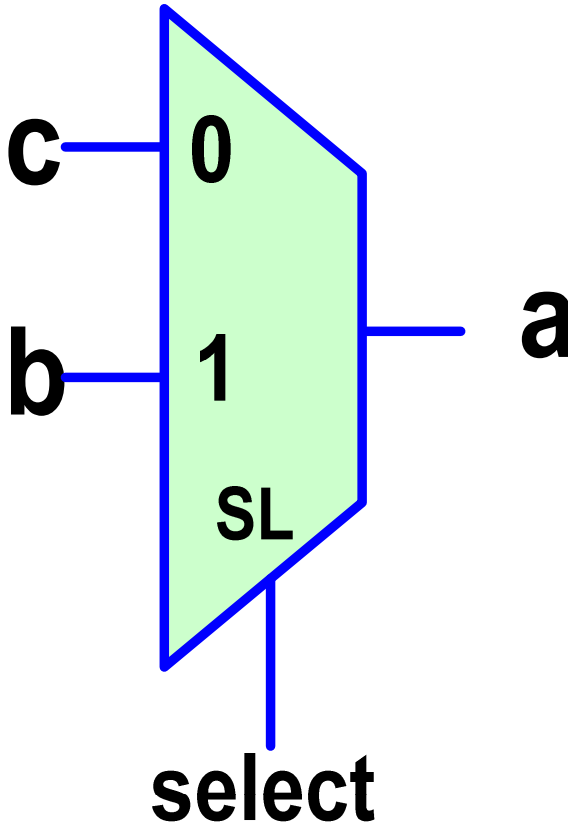
```
assign a = (sel ? b : c);
```

Método 2

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

Método 2b

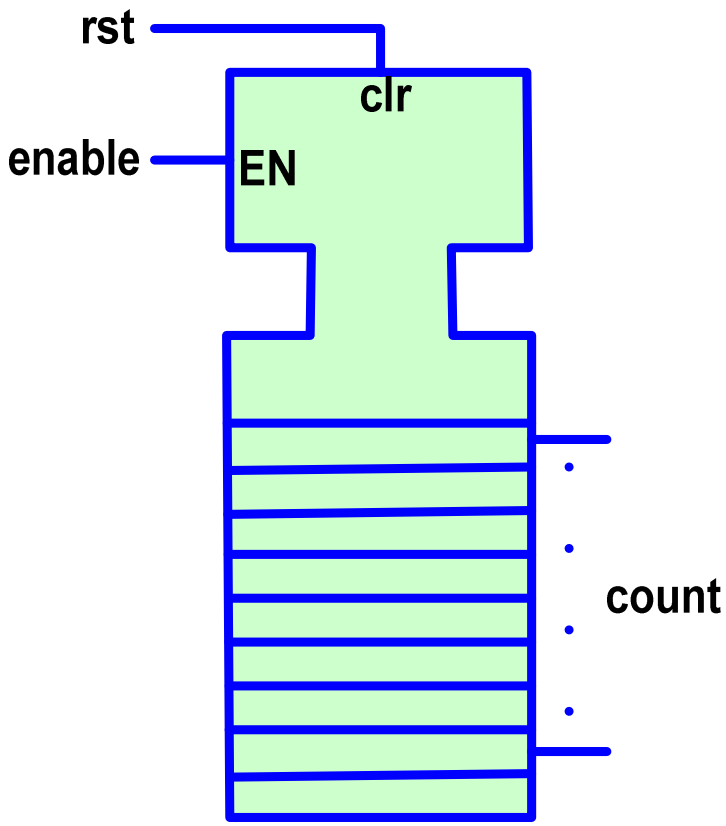
```
case(sel)
  1'b1: a=b;
  1'b0: a=c;
endcase
```



Multiplexores

Descripción Algorítmica: circuitos secuenciales

```
reg [7:0] count;
wire enable;
always@(posedge clk or
    negedge rst)
begin
  if (rst) count<=0;
  else if (enable)
    count<=count+1;
end
```



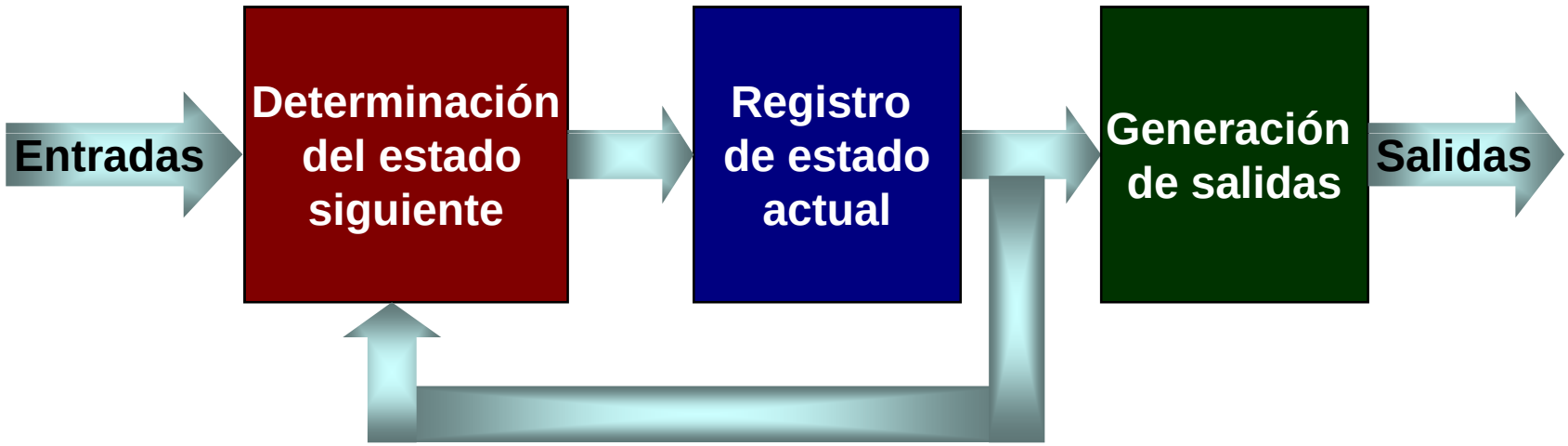
Contadores

Descripción Algorítmica: FSM

Máquinas de estado finito (FSM)

Descripción Algorítmica: FSM

El esquema de una máquina de estados de Moore es el siguiente:



Descripción Algorítmica: FSM

```
// registro de estado
reg [2:0] estado, sig_estado;
// definición de estados
localparam  E0=0, E1=1, E2=2,
   E3=3,...;



// DETERMINACIÓN ESTADO SIG.
always@(estado or entradas)
begin
  case (estado)
     E0: case (entradas)
                  sig_estado =
   ...
   …
end
```

```
// REGISTRO DE ESTADO ACTUAL
always@(posedge clk)
begin
  if (rst == 1)
     estado <= E0;
  else
     estado <= sig_estado;
end

// GENERACIÓN DE SALIDAS
always@(estado or entradas)
begin
  case (estado)
     E0: case (entradas)
                  salidas = ...
   …
end
```

Descripción Algorítmica: FSM

```
module peqFSM (clk, rst, x, z)
 input clk, rst, x; output z;
// registro de estado
 reg [2:0] estado, sig;
// estados
 localparam
   E0=0,E1=1,E2=2,E3=3,E7=7;

// registro estado actual
 always @(posedge clk)
 begin
  if (rst == 1) estado <= E0;
  else estado <= sig;
 end
// Salidas
 assign z = (estado == E7);
```

```
// Det. estado siguiente
 always @(estado, x)
 begin
  case (estado)
   E0: if(x)  sig = E1;
         else sig = E0;
   E1: if(x)  sig = E3;
         else sig = E2;
   E2: if(x)  sig = E0;
         else sig = E7;
   E3: if(x)  sig = E2;
         else sig = E7;
   E7:        sig = E0;
   default:   sig = E0;
  endcase
 end
endmodule
```

Funciones

```
function [rango] nombre_de_la_función;
  [entradas y declaraciones]
  sentencia_asignación_a_la_función;
endfunction
```

Ejemplo:

```
function [7:0] batido;
input [7:0] a;
input [2:0] control;
integer i;
begin
  for (i = 0; i <= 7; i = i + 1)
        batido[i] = a[ i ^ control ];
end
endfunction
```

Cómo evitar *latches*

Regla #1

Si hay varias alternativas, cada una debe valuar todas las salidas

- **Método1**:

Inicialice cada salida al principio del bloque. Luego sobrescriba las que cambian.

```
always @(...
begin
  x=0;y=0;z=0;
  if (a) x=2; else if (b) y=3; else z=4;
end
```

- **Método2**:

Genere valores para todas las salidas en cada caso evaluado.

```
always @(...
begin
  if (a) begin       x=2; y=0; z=0; end
  else if (b) begin x=0; y=3; z=0; end
  else begin       x=0; y=0; z=4; end
end
```

Regla #2

Todas las entradas deben figurar en la lista de sensibilidad

- **Operandos (Right-hand side variables):**

No incluya aquellas calculadas y empleadas solo dentro del bloque.

```
always @(a or b or c or x or y)
  begin
    x=a; y=b; z=c;
    w=x+y;
  end
```

- **Variables de control de flujo**:

Genere valores para todas las salidas en cada caso evaluado.

```
always @(a or b)
begin
  if (a)       begin    x=2; y=0; z=0; end
  else if (b) begin    x=0; y=3; z=0; end
  else         begin    x=0; y=0; z=4; end
end
```

Regla #3

Verifique todas las sentencias de control de flujo

- Termine cada case con default siempre.

```
case(estado)
  ...
  default: sig_estado = reset;
endcase
```

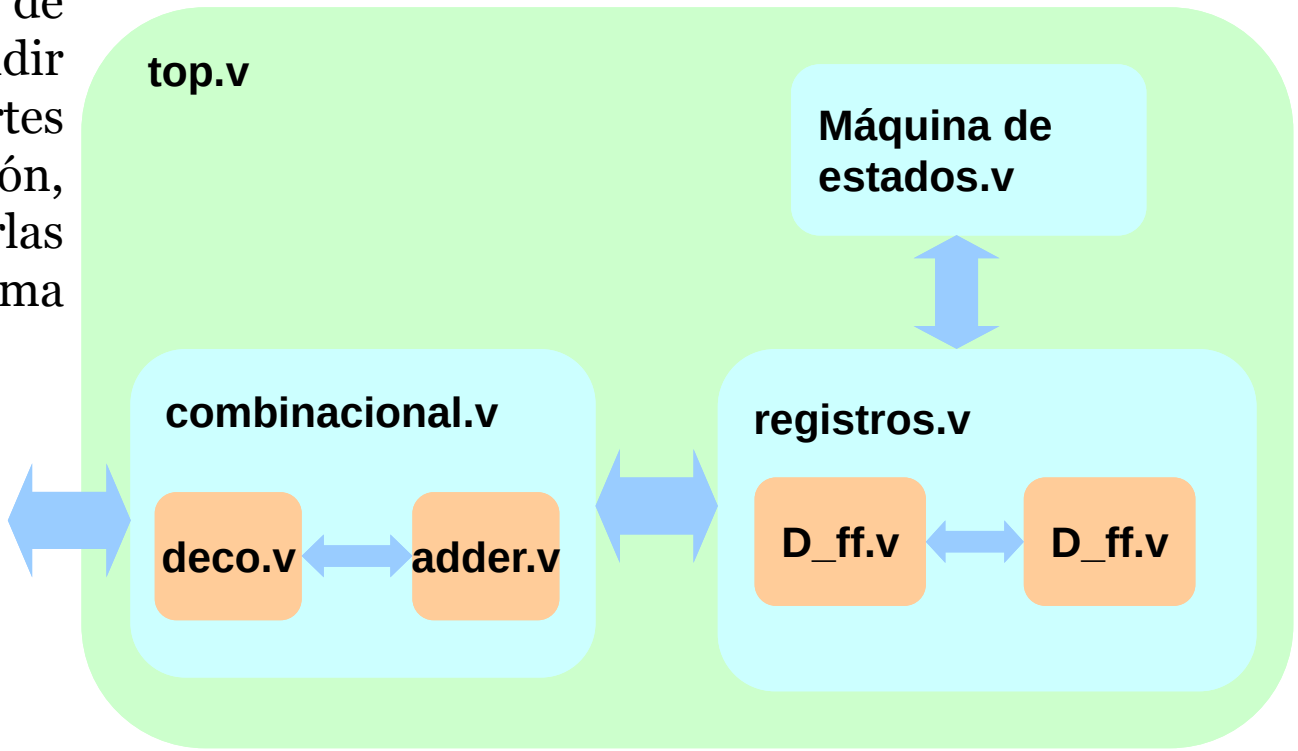
- Cuidado con los bucles de su FSM

```
if       (a|b&c) sig_estado=E1;
else if(c&d)     sig_estado=E2;
else             sig_estado=reset;
```

Descripción Estructural Avanzada

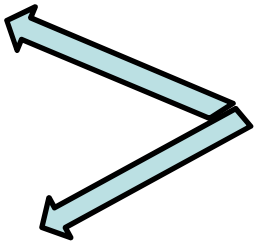
Descripción Estructural: Diseño jerárquico

Mediante este estilo de diseño se puede dividir un sistema en partes de fácil resolución, para luego integrarlas y formar el sistema completo.

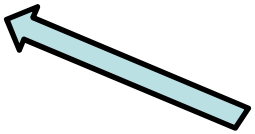


Descripción Estructural: bloque `generate`

```
genvar nombre_genvar, ... ;

generate

        genvar nombre_genvar, ... ;

        items_generate

endgenerate
```



Declaración de variables generate

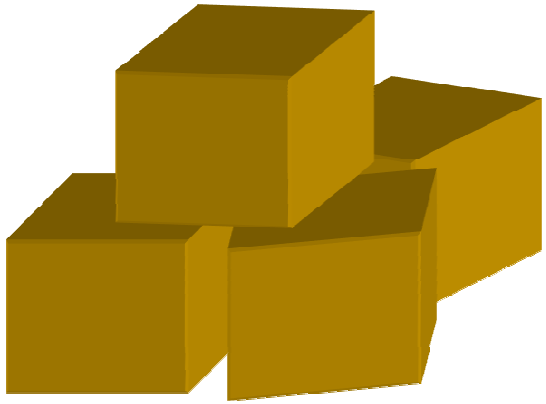


Sentencias de replicación

Descripción Estructural: bloque `generate`

Ítems generate disponibles:

- *Asignación de genvars*
- *Declaración de nets*
- *Declaración de variable*
- *Instanciación de módulos*
- *Instanciación de primitivas*
- *Asignación continua*
- *Procesos*
- *Definición de tareas*
- *Definición de funciones*



Descripción Estructural: bloque `generate`

if (*condición_expr_cte_o_genvar***)**

item_generate ó *grupo_items*

else

item_generate ó *grupo_items*

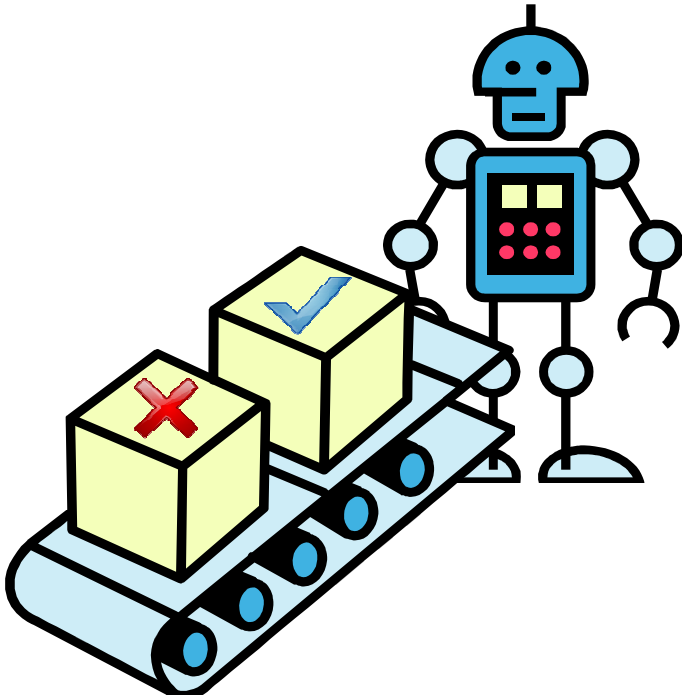
La condición debe ser una expresión de ctes. o genvars. Permite decidir cuando se instancia (o no) un elemento

grupo_items es:

begin: *nombre_grupo*

item_generate
item_generate

...
end



Descripción Estructural: bloque `generate`

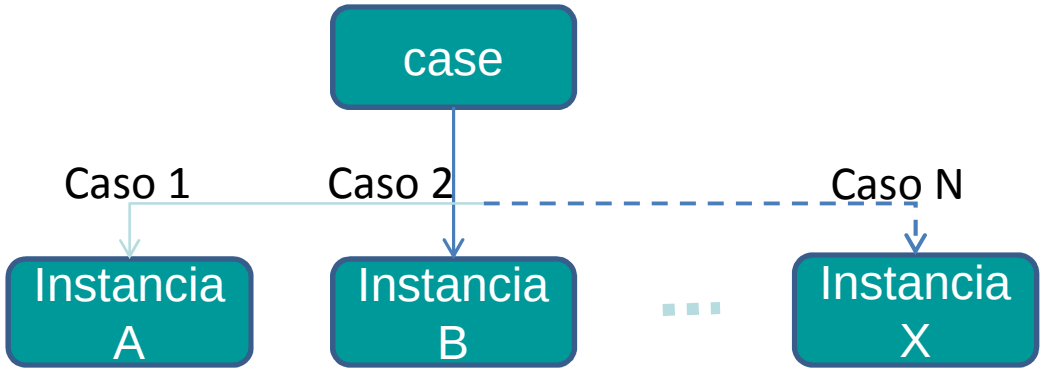
case (*expresión_constante_o_genvar*)

valor_genvar **:** *item_generate* ó *grupo_items*

valor_genvar **:** *item_generate* ó *grupo_items*

...
default: *item_generate* o *grupo_items*

endcase



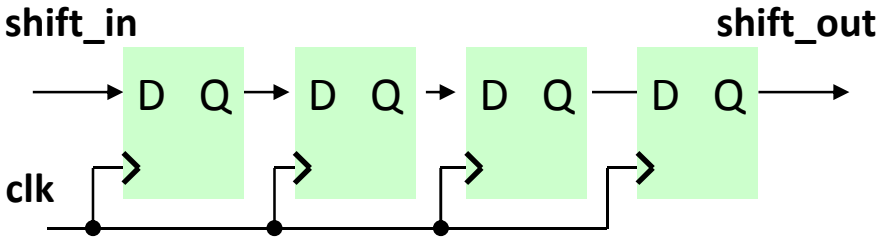
Descripción Estructural: bloque `generate`

for (*nombre_genvar* = *expresión_constante_o_genvar*;
expresión_constante_o_genvar;
nombre_genvar = *expresión_constante_o_genvar*)

item_generate ó *grupo_items*

¡Similar al `for` de C!

```
wire [N_bits-1:0 ]interna;
generate
genvar i;
for (i=0;i<N_bits;i=i+1)
   begin
      if (i==0)
         begin : primer_ff
            dff P_dff(  .D(shift_in),
                        .Q(interna[0]),
                        .clk(clk));
         end
       if (( i != 0) && ( i != Nbits-1 ))
         begin : ff_medio
             dff M_dff(  .D(interna[i-1]),
                         .Q(interna[i]),
                         .clk(clk));
         end
       if (i == (Nbits-1))
         begin : ultimo_ff
             dff U_dff(   .D(interna[i-1]),
                          .Q(shift_out),
                          .clk(clk));
         end
    end
endgenerate
```



Verilog para simulación

Funciones

```
function [rango] nombre_de_la_función;
  [entradas y declaraciones]
  sentencia_asignación_a_la_función;
endfunction
```

Ejemplo:

```
function [7:0] batido;
input [7:0] a;
input [2:0] control;
integer i;
begin
  for (i = 0; i <= 7; i = i + 1)
        batido[i] = a[ i ^ control ];
end
endfunction
```

Tareas

```
task nombre_de_la_tarea;
   [entradas, salidas y declaraciones]
   sentencia;
endtask
```

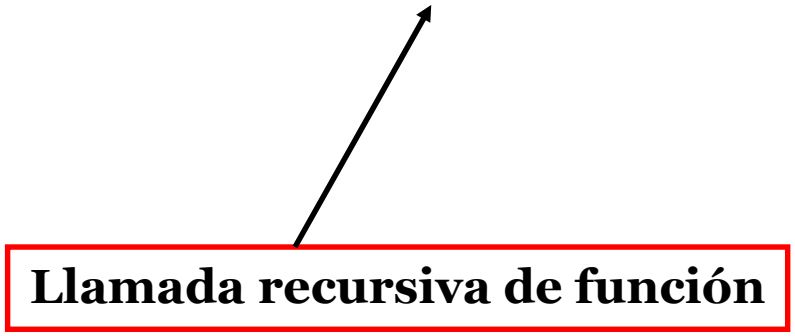
```
module ejemplo_tarea (a,b,c);
input [7:0] a,b;
output [7:0] c;
reg [7:0] c;

task adder;

input [7:0] a,b;
output [7:0] adder;
reg c;
integer i;

begin
   c = 0;
   for (i = 0; i <= 7; i = i+1)
    begin
     adder[i] = a[i] ^ b[i] ^ c;
      c = (a[i] & b[i]) | (a[i]
           &c) | (b[i] & c);
   end
end
endtask
always
 adder (a,b,c); //c es reg
endmodule
```

Tareas

```
//DEFINICIÓN DE TAREA (dentro de un módulo)
task read_mem (input [15:0] address, output [31:0]
  data );
begin
  read_request = 1;
  wait (read_grant) addr_bus = address;
  data = data_bus;
  #5 addr_bus = 16'bz; read_request = 0;
end
endtask

//USO DE TAREA (LLAMADA)
always @(posedge clock)
  read_mem(PC, IR);
```

Funciones y tareas recursivas: uso de automatic

```
function automatic [63:0] factorial;
input [31:0] n;

  if (n == 1) factorial = 1;
  else        factorial = n*factorial(n-1);

endfunction
```

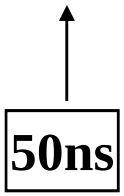


Verilog para simulación: Directivas del compilador

- `include "nombre de archivo" → incluye como referencia de búsqueda, los contenidos del archivo en el actual; debe escribirse en dentro del módulo.

- `define <texto1> <texto2> → texto1 substituye o define a texto2;
  - ej.: `define BUS reg [31:0]`
  - en porción declarativa: `BUS data; /* define a data como reg [31:0] */`

- `timescale <unidad de tiempo>/<precisión>
  - ej.: `timescale 10ns/1ns` luego: `#5 a = b;`



Verilog para simulación: Tareas del sistema

- Se emplean en la simulación para generar entradas y salidas. Comienzan con el signo **$**.

- Control de tiempo de simulación

  $stop; → suspende la simulación, permite al usuario ingreso de comandos

  $finish; → finaliza la simulación

- Generador de números aleatorios: **$random** (*entero_semilla*)
- Para consultar tiempo actual de simulación: **$time**

Verilog para simulación: Tareas del sistema, archivos

```
arch_datos = $fopen("ruta y nombre archivo", "r");
```
→ retorna descriptor de archivo (integer)

```
cod = $fscanf (arch_datos,"%b ", datos);
```

```
$fdisplay(arch_datos, "formato cadena", arg2,
arg3, ..);
```
→ escribe en el archivo

```
$fmonitor(arch_datos, "formato cadena", arg2,
arg3, ..);
```
→ escribe en el archivo

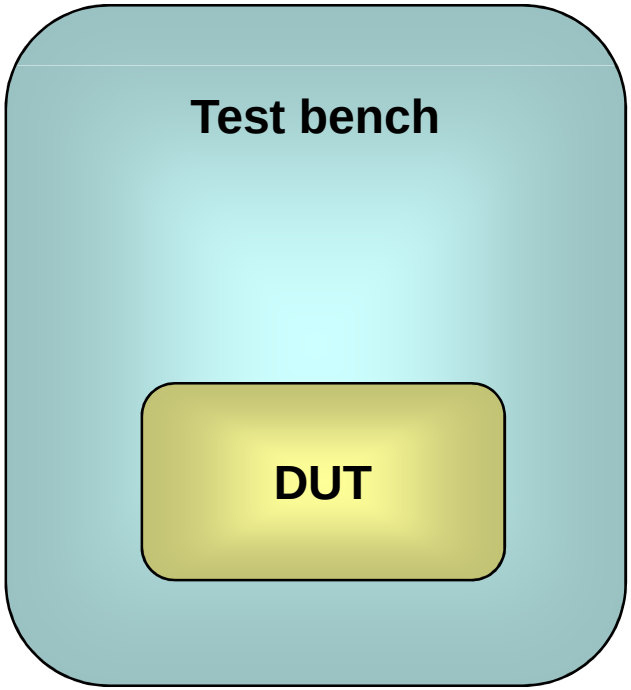
```
$fclose(arch_datos);
```
→ cierra el archivo

Verilog para simulación: Formato de cadena para $display, $monitor y $fscanf

Format	Display
%d or %D	Display variable in decimal
%b or %B	Display variable in binary
%s or %S	Display string
%h or %H	Display variable in hex
%c or %C	Display ASCII character
%m or %M	Display hierarchical name
%v or %V	Display strength
%o or %O	Display variable in octal
%t or %T	Display in current time format
%e or %E	Display real number in scientific format
%f or %F	Display real number in decimal format
%g or %G	Display scientific or decimal, whichever is shorter

Verilog para simulación: *Test benches*

Un *test bench* (banco de pruebas) se encarga de generar los estímulos necesarios para comprobar un circuito y corroborar los resultados de la simulación.



Un test bench es un módulo sin puertos que se utiliza para probar un diseño

DUT: Design Under Test

Verilog para simulación: *Test benches*

Pasos para crear un test bench:

1. Creación de un módulo sin puertos.

2. Declaración de una variable `reg` por cada puerto de entrada y una `wire` por cada salida del diseño bajo prueba, con el mismo nombre (e inicialización de las entradas).

3. Instanciación del diseño bajo prueba con asignación de señales homónimas.

4. Generación de estímulos de reloj (clock) y de reset.

5. Generación de los estímulos o vectores de prueba (generalmente en un bloque `initial` u `always`).

6. Escritura de las sentencias que comprobarán los resultados o lectura y almacenamiento de archivo para contrastación.

Verilog para simulación: *Test bench* de ejemplo

```
‘timescale 1 ns /100 ps
/* unidad de tiempo = 1ns;
   precisión = 1/10ns; */
module peqFSM_tb;
reg clk, rst, x;
wire z;

// Instanciación de peqFSM
peqFSM dut1(clk, rst, x, z);

// Generación de CLOCK y RESET
initial
begin
  clk=0;
  rst=0;
  #1 rst=1; /*La demora asegura que
   haya flanco positivo de reset.*/
  #200 rst=0; /*Desactiva reset
   luego de dos ciclos de reloj.*/
end
always #50clk=~clk; /* reloj 10MHz
   (50*1ns*2) con 50% duty-cycle */

/* Generación de estímulos */
initial
begin
  #1 x=0;
  #400 x=1;
  $display("Salida z: %b", z);
  #100 x=0;
  @(posedge clk) x=1;

  #1000  $finish; /*fin de simulación
   sin esto, no se detiene */
end
endmodule
```

Verilog para simulación: Test bench de ejemplo

```
...
initial
begin
  clock = 1 'b0 ;
  input_data = $fopen
   ({'VECTOR_DIR,'FILENAME_IN},
   "r" ) ;
  reset = 1 'b1 ;
  #20 reset = 1 'b0 ;
end

always #0.776 clock = ~clock ;
```

```
always @( posedge clock or
   negedge reset )
begin
 if (reset)
 begin
  for ( k=0; k<3; k=k+1)
  temp_data [ k ]=0;
 end
 else
 begin
  for (k=0; k<3; k=k+1)
  begin
   code = $fscanf (input_data,
   "%b ", carrier);
   temp_data [ k ] <= carrier ;
  end
 if (code == -1)
 #10 $stop ;
 end
end
```

Reglas de estilo

Reglas de estilo: módulos

- Nombre del archivo idéntico al del módulo que contiene, más extensión ".v"
- Todos los nombres de identificadores deberán estar en inglés, y ser autoexplicativos mientras sea posible, para evitar comentarios extras
- Sangrado (*indentation*) de dos espacios, nunca tabulaciones (*tab*)
- Nombres de módulos en minúsculas, palabras separadas por guión bajo: **`top_adder`**, **`hex_decoder`**

Reglas de estilo: puertos

- Orden de puertos: salidas, entradas, bidireccionales, controles, *reset*, *clock*
- Puerto de la señal de reloj del módulo se llama **`clock`**. Si hay más de una, se identifica con una etiqueta: `[label]_`**`clock`**
- Puerto de la señal de *reset* del módulo se llama **`reset`**
- Nombres de puertos de entrada y salida comienzan con "**`i`**_" y "**`o`**_" respectivamente y longitud no menor a 7 caracteres en total, y deben ser autoexplicativos, mientras sea posible
- Nombres de puertos y señales idénticos en todo nivel de jerarquía, mientras sea aplicable
- Solo las señales de control del ***fast control signal generator*** se escriben en mayúsculas, separando las palabras con guión bajo

Reglas de estilo: puertos

- Estilo ANSI C para la declaración de puertos:

```
module half_adder
  #(
  parameter nb_in
  )
  (
  out wire [nb_in - 1] o_sum_result,
  out wire [nb_in - 1] o_carry,
  in  wire [nb_in - 1] i_data_a,
  in  wire [nb_in - 1] i_data_b
  );
```

Reglas de estilo: constantes y números

- Defina siempre las constantes y valores para parámetros externos e internos (locales) mediante `` `define `` (se usa el tilde o acento grave, inverso, al principio) y siempre en mayúsculas, con nombres autoexplicativos, mientras sea posible:

```
`define INTEGER_SAMPLE_WIDTH 7
`define FRACTIONAL_SAMPLE_WIDTH 9
`define DATA_PATH_WIDTH 32
`define SAMPLES_NUMBER 156
```

- Especifique siempre la longitud y la base de los números que emplee en asignaciones y comparaciones (estos emplean el apóstrofo vertical no tilde, como separador):
  - `1b'1` no `1`
  - `1b'0` no `0`
  - ...

Reglas de estilo: variables

- Nombres de variables `reg` y `wire` al menos de 5 caracteres, y deben ser autoexplicativos mientras sea posible
- Solo las señales de control del ***fast control signal generator*** se escriben en mayúsculas, separando las palabras con guión bajo
- Si son variables de generación (`genvar` de bloques `generate`) pueden emplearse índices clásicos cortos: **`i`**, **`j`**, **`k`**, etc.

Reglas de estilo: parámetros

- Parámetros de definición de número de bits, comienzan con "**nb**_", y luego el nombre de variable al que se aplica el parámetro: **nb**_[nombre]
- Para variables en punto fijo, se definen número de bits fraccionales (**nbf**_[nombre]) y número de bits enteros (**nbi**_[nombre]), mediante:

```
localparam nb_[nombre] = nbi_[nombre] + nbf_[nombre]
```

- Use siempre para los parámetros una constante previamente definida con `define

```
parameter  nbi_sample = `INTEGER_SAMPLE_WIDTH;
parameter  nbf_sample = `FRACTIONAL_SAMPLE_WIDTH;
localparam nb_sample  = nbi_sample + nbf_sample; // format s16.9
```

Reglas de estilo: instanciación de módulos

- Nombre de instancia de módulo comienza con "**u**_", y luego el nombre del módulo
- Parámetros y puertos instanciados con lista nominada (explícita), no por lista ordenada (implícita por posición)
- Un solo puerto por renglón, con sangrado idéntico al primer paréntesis:

```
bit_sync
#(
.nb_samples     (nb_samples)
)
u_bit_sync
(
.i_reset_nd     ( i_hdwrst_nd ),
.i_dest_d_clk   ( i_644MHz_d_clk ),
.i_bit_in       ( i_lck2ref_n ),
.o_bit_out_d    ( o_bit_out_d )
);
```

Reglas de estilo: comentarios

- Coméntese un algoritmo, sobre todo, cuando la funcionalidad no sea trivial
- En la declaración de puertos y variables, descríbase su uso, función o significado, si no es claro en su nombre
- Puede comentarse en castellano, pero se prefiere en inglés
- Mantenga actualizados los comentarios cuando modifica el código, para no inducir a malinterpretación del nuevo código

Reglas de estilo: bloques `begin end`

- Coméntese un algoritmo, sobre todo, cuando la funcionalidad no sea trivial
- En la declaración de puertos y variables, descríbase su uso, función o significado, si no es claro en su nombre
- Etiquete cada bloque de código delimitado por **`begin`** y **`end`**; idéntico sangrado que el **`always`**, **`initial`**, **`if`**, **`for`**, etc. La etiqueta definida en el **`begin`**, va como comentario en el **`end`** correspondiente:

```
always @(...)
begin : decoder
   ...;
end // decoder
```

Reglas de estilo: bloques `always` para combinacionales

- Use coma como separador en la lista de sensibilidad, no la palabra "`or`" para evitar confusiones con el comportamiento esperado:

```
always @ (i_data_a, i_data_b, i_opcode)
```

- Asegúrese de que la lista de sensibilidad esté completa para evitar *latches* y comportamiento indeseados
- Sino, utilice el comodín `*` para la lista de sensibilidad (equivalente a una lista completa); esto evita *latches* indeseados y errores
- Para asignaciones, emplee siempre el signo igual "`=`" bloqueante:

```
always @(*)
begin : add_samples
  added_samples = sample1 + sample2 + samples3;
end // add_samples
```

Reglas de estilo: bloques `always` para secuenciales

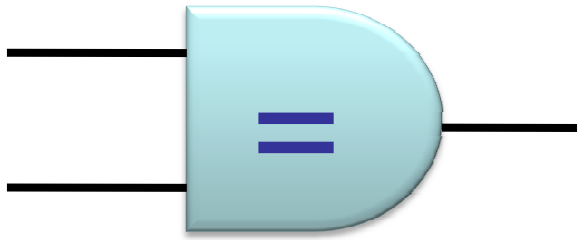
- Si el bloque es puramente secuencial, utilice `(posedge clock)` en lista de sensibilidad
- Solo emplee diseños sincronizados y todas las señales de control sincrónicas, nunca realice operaciones sobre la señal de reloj (ni *gated clock*, ni *clock gating*). Emplee en estos casos, señales de habilitación sincrónicas (ej.: *clock_enable*)
- Emplee *reset* sincrónico, siempre que sea posible
- Para asignaciones, emplee siempre el doble signo "**<=**" antibloqueante:

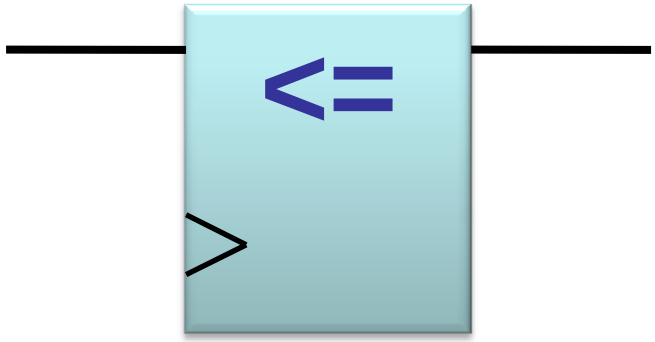
```
always @(posedge clock)
begin : D_type_flipflop
  if (reset)
    o_flipflop_d <= 1b'0;
  else
    o_flipflop_d <= i_flipflop_d;
end // D_type_flipflop
```

Reglas de estilo: asignaciones bloqueantes y antibloqueantes

En bloques `initial` u `always`:

1. Para comportamiento secuencial, emplee siempre el doble signo "<=" de la asignación antibloqueante
2. Para comportamiento combinacional, emplee siempre el signo "=" de la asignación bloqueante:
3. Si debe combinar comportamiento secuencial y combinacional en un único bloque, use únicamente asignaciones antibloqueantes (<=)
4. Nunca mezcle asignaciones bloqueantes (=, combinacionales) y antibloqueantes (<=, secuenciales) en un mismo bloque
5. No asigne una variable en más de un bloque de un mismo módulo

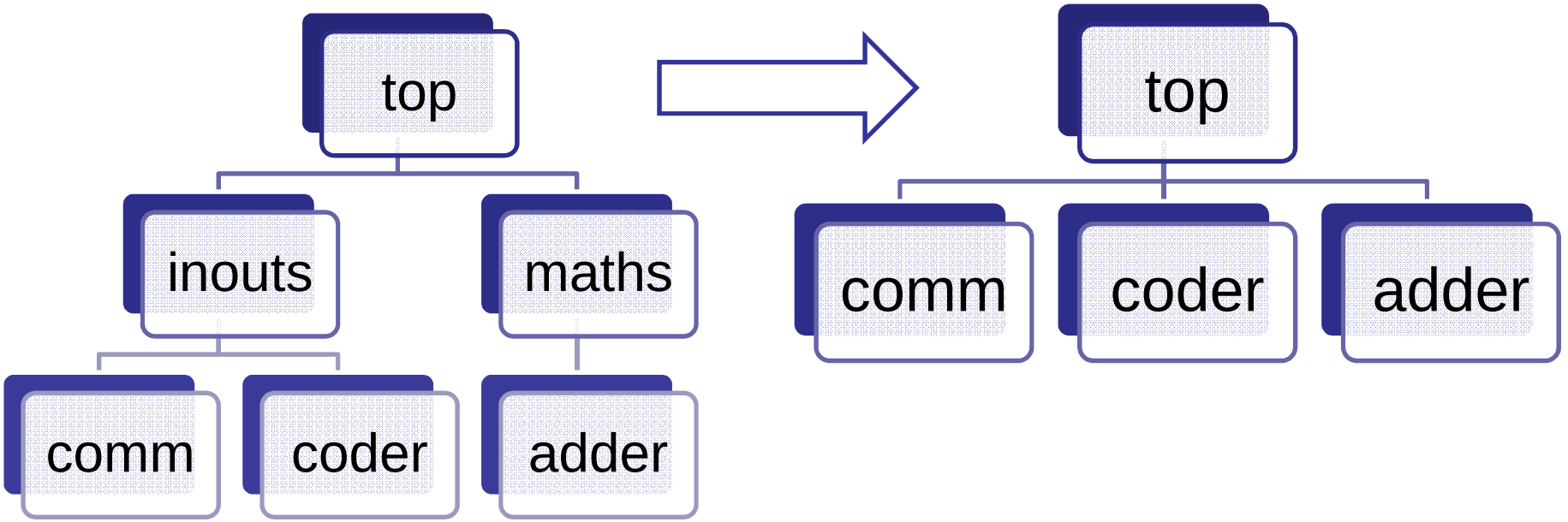




Sugerencias para síntesis

Sugerencias de síntesis: reglas generales

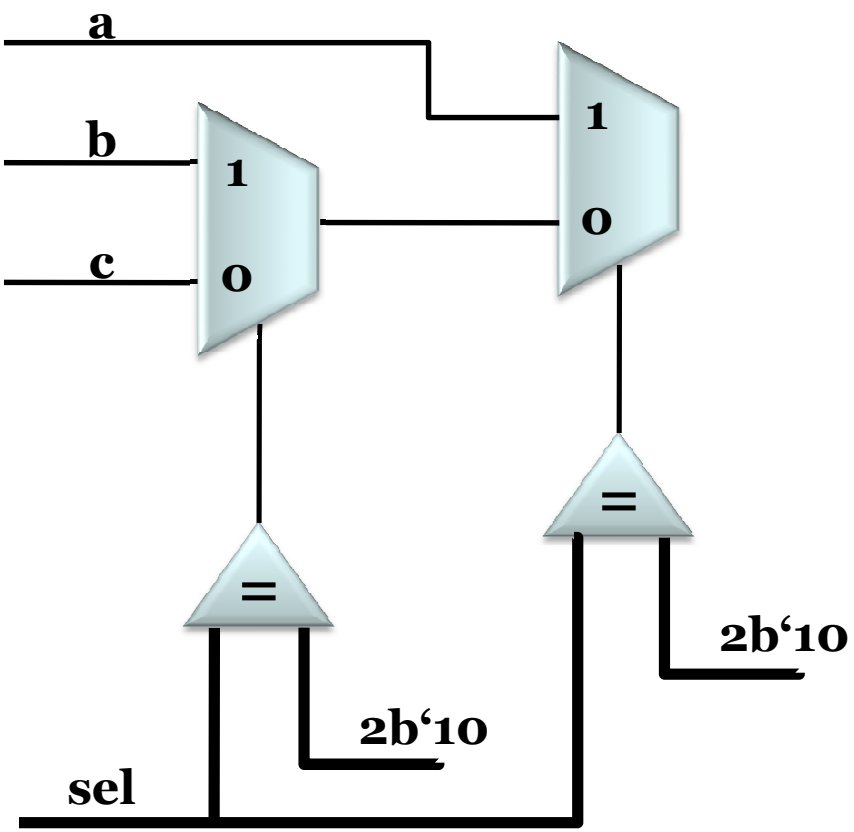
- Reestructure un diseño que emplea repetidamente grande s componentes, para minimizar la cantidad de instancias
- En un diseño, que durante su operación, necesita algunas variables almacenadas pero no todas, minimice el número de *latches* o *flip-flops* requeridos
- Considere la reducción de estructuras jerárquicas para contribuir a un diseño más eficiente



Sugerencias de síntesis: codificadores de prioridad

- Si usa el operador de asignación condicional (o asignación condicional), o construcciones **`if else`**, tenga en cuenta que por lo general, el compilador deducirá la estructura de un codificador de prioridad
- En un case, si el compilador no puede garantizar que las condiciones sean equiprobables, genera también un codificador de prioridad

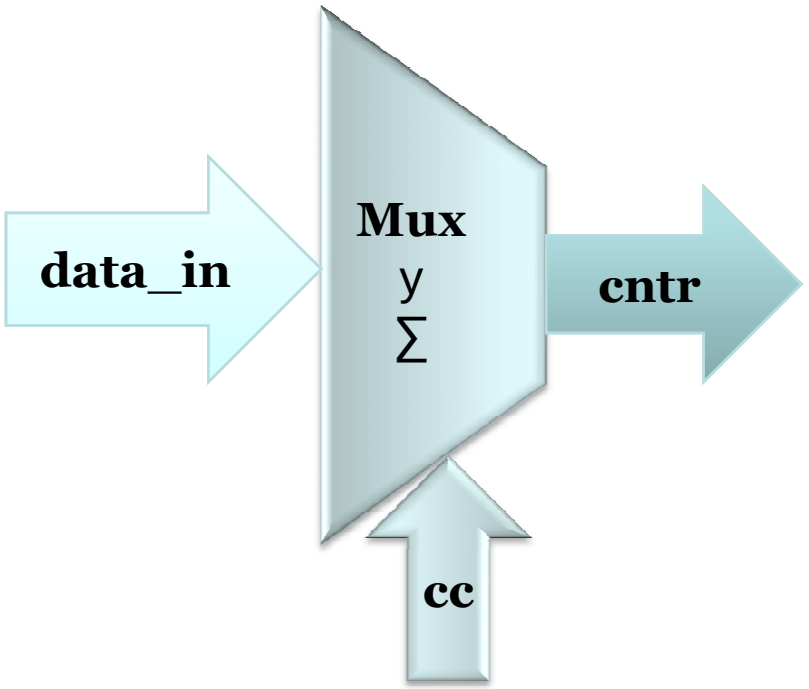
```
// codificador de prioridad
always @ (sel, a, b, c)
  if      (sel == 2'b11)
    d = a;
  else if (sel == 2'b10)
    d = b;
  else
    d = c;
```



Sugerencias de síntesis: multiplexores

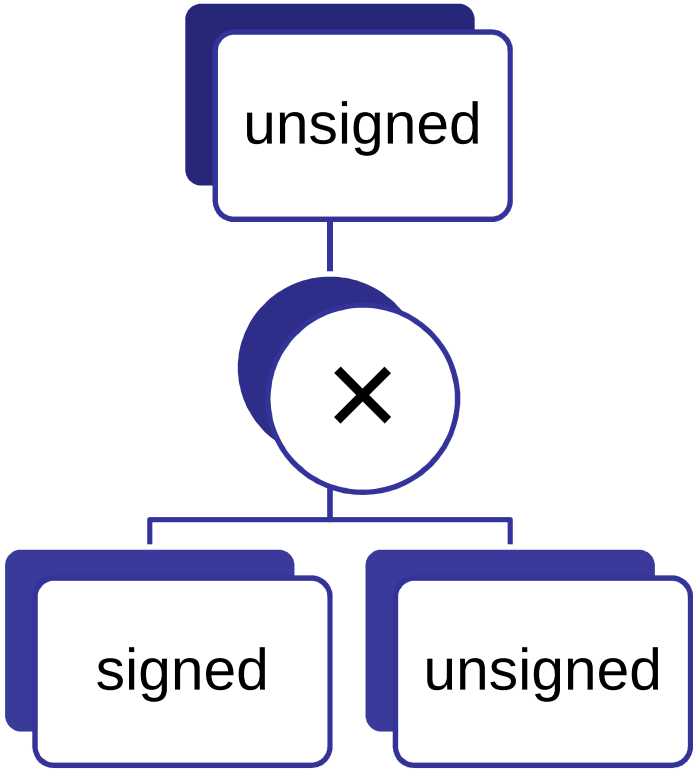
- Para generar multiplexores con condiciones equiprobables debe emplearse la sentencia case, lo que evita la generación de codificadores de prioridad
- Si se emplean `if else` o `case` y se quiere forzar la generación de un multiplexor debe emplearse el *pragma* `parallel_case`:

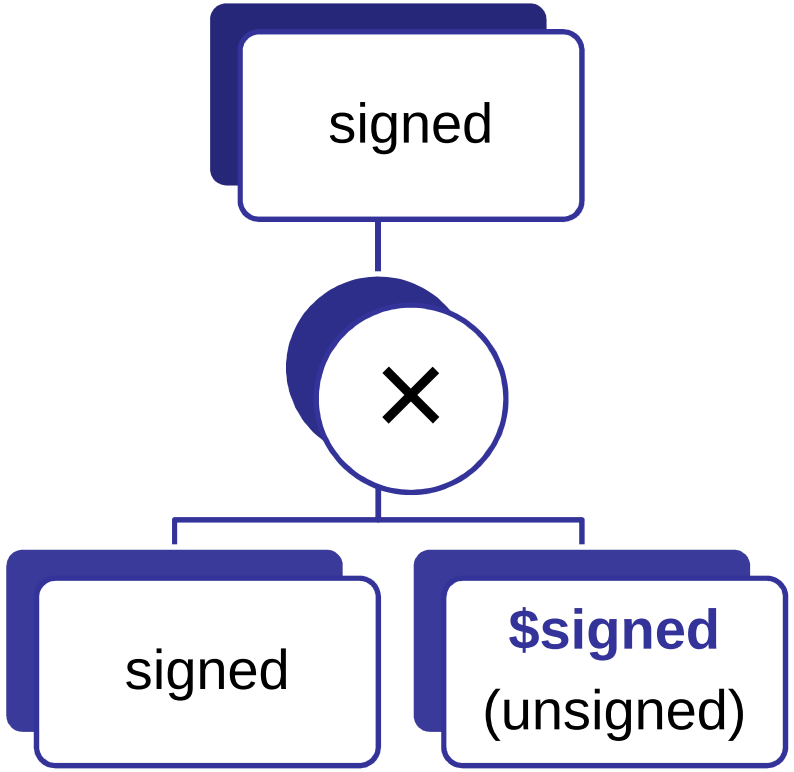
```
module case_parallel (clk, cc, data_in, cntr);
  input clk;
  input [3:0] cc;
  input [2:0] data_in;
  output [2:0] cntr;
  reg [2:0] cntr;
  always @(posedge clk) begin
  case (1'b1) // cadence parallel_case
    cc[0] : cntr = 0 ;
    cc[1] : cntr = data_in ;
    cc[2] : cntr = data_in - 1 ;
    cc[3] : cntr = data_in + 1 ;
  endcase
  end
endmodule
```



Sugerencias de síntesis: operaciones con `signed`

- No mezcle operaciones entre variables **`signed`** y **`unsigned`**, pues el resultado es siempre **`unsigned`**
- Si es necesario, entonces emplee funciones de conversión **`$signed`** y **`$unsigned`**

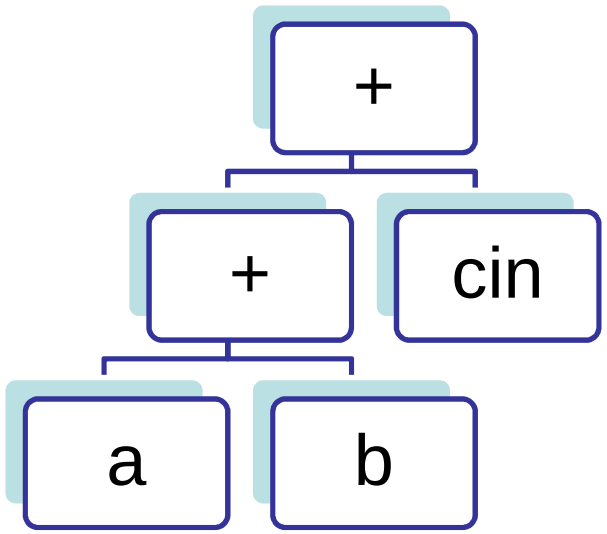




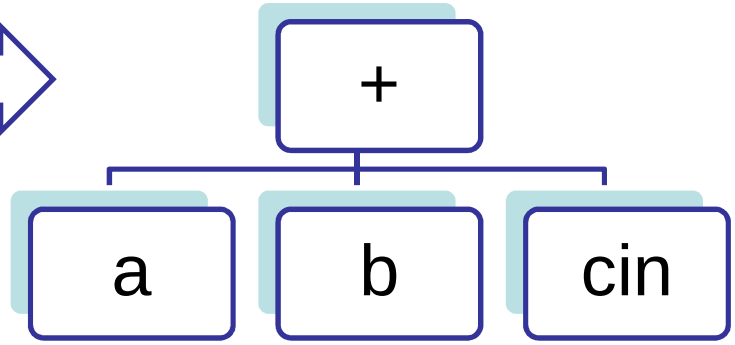
Sugerencias de síntesis: inferencia de sumadores con acarreo

- El compilador puede deducir que una suma puede implementarse con un sumador con acarreo

```
z = (a + b) + cin
```



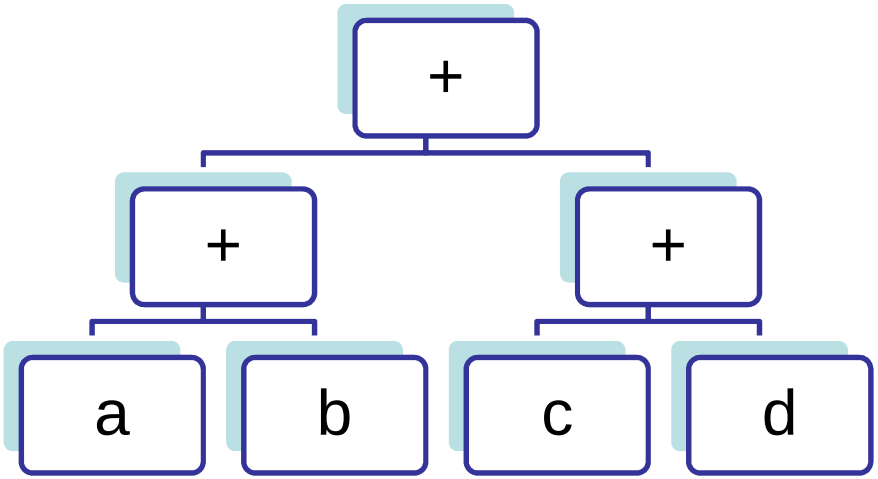

```
z = a + b + cin
```



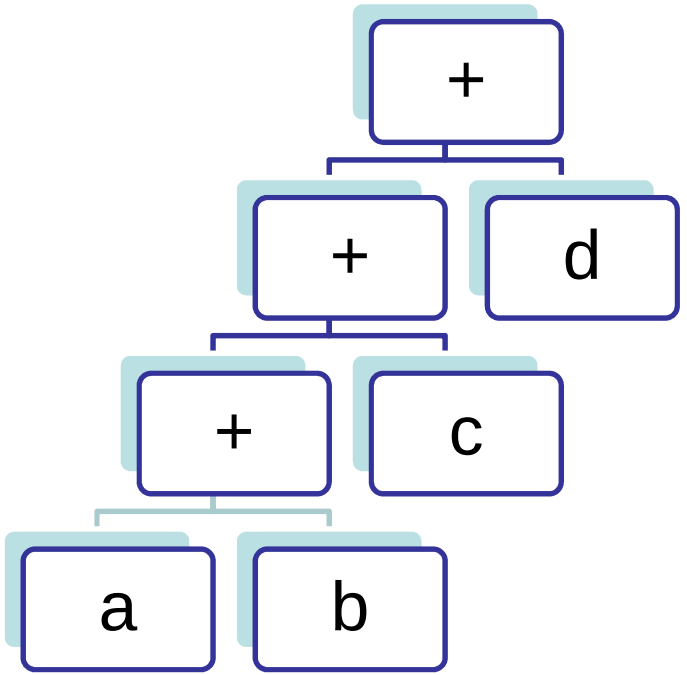
Sugerencias de síntesis: paréntesis y balanceo de árboles

- Los paréntesis pueden ayudar al compilador a deducir la estructura de árbol que uno necesita:

```
z = (a + b) + (c + d)
```




```
z = ((a + b) + c) + d
```



Sugerencias de síntesis: bibliografía de consulta

1. HDL Modeling in Encounter RTL Compiler v9.1, April 2010, Cadence
2. Datapath Synthesis in Encounter RTL Compiler v9.1, April 2010, Cadence