Lenguajes de descripción de Hardware (Verilog)

David Alejandro González Márquez

Programación de softcores en FPGAs
Programa de Profesoras/es Visitantes
Departamento de computación
Universidad de Buenos Aires

Clase disponible en: `https://github.com/fokerman/fpgaSoftcoreProgrammingCourse`

Hardware Description Lenguajes

HDL

Lenguajes de programación que nos permiten **describir** el comportamiento o la estructura del hardware de sistemas digitales.

Permiten especificar circuitos con **complejos diseños**, **simular** su comportamiento y **sintetizar** su implementación.

Son lenguajes específicamente diseñados para describir hardware, **no son de propósito general**.

Existen dos lenguajes bien conocidos de HDL.

- **VHDL** (Estándar ANSI/IEEE 1076-1993)
  Desarrollado en 1981 por el Departamento de Defensa de los EEUU.
- **Verilog** (Estándar IEEE 1364-2001)
  Desarrollado en 1984 por Gateway Design Automation (Now Cadence Design Systems).

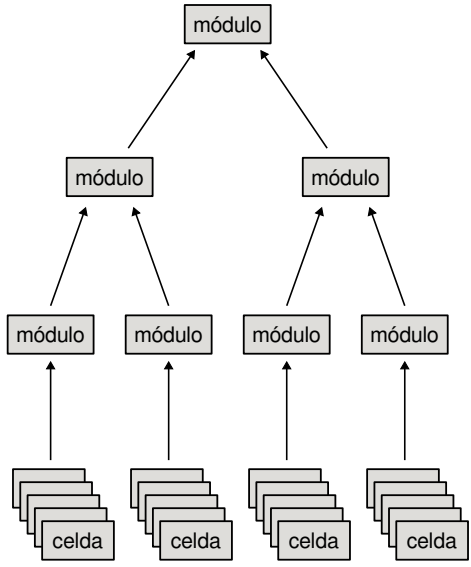
Diseño Jerárquico

El diseño jerárquico consiste en construir una jerarquía de módulos. Desde niveles más bajos de abstracción a niveles más altos.

Construir un diseño jerárquico nos permite:

- **Tener control** sobre el diseño y sobre las modificaciones o instanciaciones particulares.
- **Reducir la complejidad** del diseño, reduciendo comportamientos complejos a módulos.

Sin embargo, un mal diseño puede generar **complejidad accidental** en la sintesis y por lo tanto construir una implementación **ineficiente** o que **no respeta el comportamiento** esperado.



Estrategias de diseño

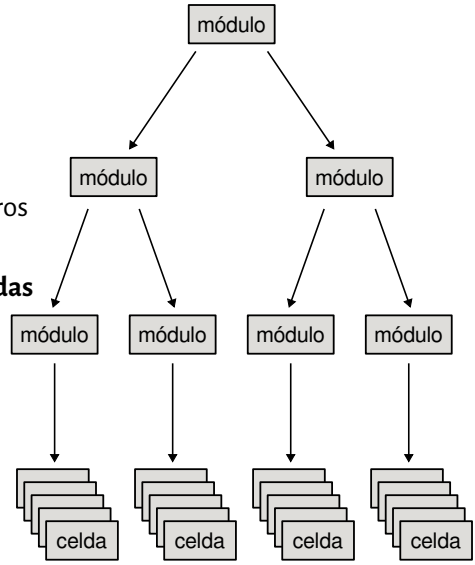
Top Down Design

- Definimos el **módulo principal** (*Top module*)
- Construimos los **Sub-Módulos** que permiten modelar el modulo principal.
- Diseñamos cada sub-módulo implementando otros módulos que **simplifican** el problema.
- Por último llegamos al **nivel de compuertas o celdas** (*leaf cell*, tanto *logic gates* o *cell libraries*).

Botton-Up Design

- Comenzamos construyendo módulos básicos.
- Hasta llegar a construir módulos más complejos.

Los buenos diseños suelen ser afrontados con las dos estrategias.



¿Cómo se utiliza el código de descripción de hardware?

Sintetización (`hardware Syntesis`)

- Herramientas mapean el código a librerías de **celdas de bajo nivel**.
  Construyen una *netlist* de compuertas (gates) y cables (wires).
- Sobre la *netlist* se resuelven los algoritmos de ***placement*** **(ubicación)** y ***routing*** **(ruteo)**.
- El código debe ser escrito para que sea fácil de sintetizar, ya que no se puede garantizar encontrar una solución óptima, ni siquiera una solución.

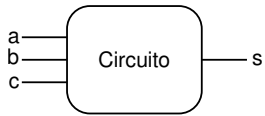
Simulación

- Permite verificar el comportamiento del circuito **sin que sea manufacturado**.
- Valida el comportamiento tanto del **código escrito**, ya sea de forma *structural* o *behavioral*.
- La simulación es esencial para verificar el funcionamiento y ***timming*** **del circuito**.

Los HDL buscan describir *hardware*, **NO se deben pensar como código ejecutable.**
Para poder escribir esté código, se debe conocer que es lo que se busca construir.

Definiendo módulos en Verilog

Un `module` es el bloque de construcción más básico de Verilog.



Se define con:

- **Nombre**: Identificación del modulo, debe ser única.
- **Puertos**: Define cada una de las entradas y salidas.
- **Cuerpo**: Descripción de su funcionalidad.

Ejemplo:

```
module example(a, b, c, s);
    input a;
    input b;
    input c;
    output s;
    ....
endmodule
```

La dirección de los puertos se puede definir como parte de la declaración.

```
module example(input a, input b,
               input c, output s);
    ....
endmodule
```

Definición de entradas y salidas con múltiples bits

La sintaxis para definir múltiples bits es la siguiente:

```
[start:end]
```

Donde `start` corresponde al bit numerado como más significativo y end al bit numerado como menos significativo. Tanto `start` como end **pueden tomar cualquier número** entero positivo, incluso pueden estar a la inversa.

```
input  [15:0] a; // --> a[15], a[14], a[13], ..., a[1], a[0]
input  [1:8]  b; // --> b[1], b[2], ..., b[6], b[8]
input  [3:2]  c; // --> c[3], c[2]
output        d; // --> d (only one signal)
```

Preferimos la forma `[15:0]` para definir multi-señales, donde el comienzo es un entero positivo y final siempre es cero.

Solo para arreglos de señales podemos usar la declaración a la inversa para evitar confusiones.

Tipos de datos

En `Verilog` las variables pueden ser de dos tipos:

- Cable (*net data type*)
  Representan cables que pueden conectar distintos circuitos lógicos.

  - **`wire`**: Cable bidireccional, se puede usar para completar múltiples entradas y salidas.
  - **`wor`**: Se agrega una operación lógica OR entre todas las entradas al cable.
  - **`wand`**: Se agrega una operación lógica AND entre todas las entradas al cable.
  - **`supply0`**: Cable siempre conectado a cero lógico.
  - **`supply1`**: Cable siempre conectado a uno lógico.

- Registro (*register data type*)
  Representa una variable que guarda información. No necesariamente será sintetizado como un conjunto de flip-flops.

  - **`reg`**: Declara un conjunto de bits de forma explicita como variable.
  - **`integer`**: Representa un número en complemento a 2 de como máximo 32 bits.

Si no se indica la cantidad de bits en cada caso, el default es 1.

Valores lógicos

En `Verilog` se define un conjunto de estados posibles.

- **`0`** → Valor lógico cero.
- **`1`** → Valor lógico uno.
- **`z`** → Alta impedancia (*high-impedance*).
- **`z`** → Valor sin importancia estructuras de casos (*dont-care*).
- **`x`** → Valor sin importancia (*dont-care*).
- **`x`** → Valor desconocido (*unknown*).

Notar que x puede significar desconocido o sin importancia, ya que su significado depende del contexto.

Para asignar valores a variables en `Verilog` se utiliza la palabra reservada **`assign`**.

Ejemplo:

```
assign x = 1;
assign z = y + x;
```

Definición de constantes

Simple decimal

Número decimal signado de 32 bits como máximo.

```
410      // numero signado de 32 bits
-22      // numero signado de 32 bits
```

Base format

Formato *S* ' *F* *x* . . . *x*, donde *S* es el tamaño en bits; *F* el formato: binario (b), decimal (d);
y *x* . . . *x* los bits del número representado.

```
3'b010   // numero de 3 bits, representa el numero 2
7'd-4    // numero de 7 bits, representa el -4 en complemento a dos
'd-10    // numero de 32 bits, representa el -10 en complemento a dos
```

Parameters

Es una constante con nombre. Puede no tener tamaño específico, este depende de su uso.
Es asignado al momento de instanciar el módulo y queda fijo en tiempo de simulación o síntesis.

```
parameter RED = -1, BLUE = 2;
parameter READY = 2'b00, BUSY = 2'11, INVALID = 2'01;
```

Manipulación de bits

Supongamos las siguientes variables

```
wire [15:0] bus; wire [7:0] data; wire [7:0] addr;
```

Bit slicing

```
assing data = bus[15:8]
```

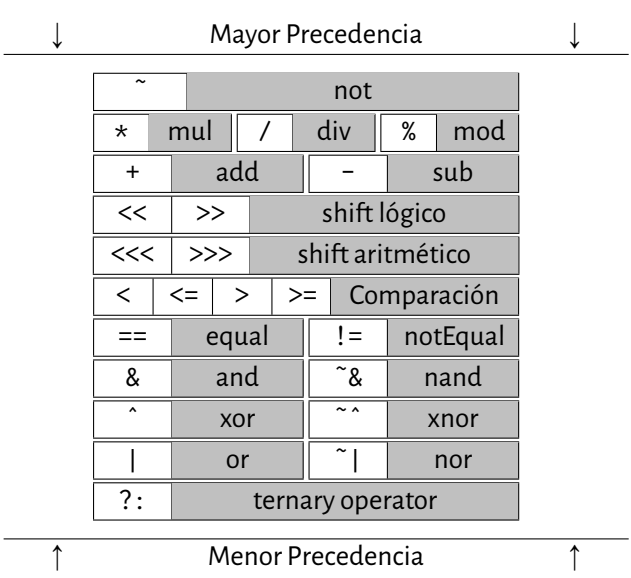
Concatenación

```
assing bus = { data, addr }
assing bus = { data[7:4], addr, data[3:0] }
assing bus = { data[7:4], addr[7:4], data[3:0], addr[3:0] }
```

Duplicación

```
assing bus = { 4'd0, data[7], data[7], data[7], data[7], data[7:0] }
assing bus = { 4'd0, 4{data[7]}, data[7:0] }
```

Operadores Aritméticos y Lógicos



Las expresiones en `verilog` se pueden construir utilizando **cualquier combinación de operadores**.

Los operadores binarios deben ser utilizados con datos de la **misma cantidad de bits**.

Las operaciones aritméticas se realizan considerando que son operaciones **en complemento a 2**.

`Verilog` es *case sensitive*.
Comentarios una línea `//···`
Comentarios multi-línea `/*··· */`

Estilos de descripción de hardware

Structural (Gate-Level)

- Cada modulo contiene una descripción a **nivel de compuertas** del circuito.
- Describe como se **interconectan** los módulos en módulos que los contienen.
- Define una **jerarquía de módulos** a partir de compuertas básicas.

Behavioral (if-else-operators)

- Cada modulo contiene una **descripción funcional** del circuito, utilizando operadores lógicos y aritméticos.
- Presenta un **más alto nivel de abstracción** que usando compuertas.
- A partir de una descripción de comportamiento se pueden construir **múltiples diseños usando diferentes compuertas**.

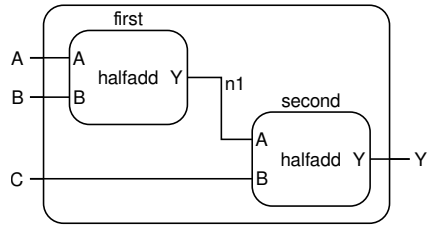
La mayoría de los diseños utilizan una combinación de los dos enfoques o estilos.

Ejemplo: Estilo Structural

Se definen módulos y sus conexiones, por medio del nombre de la entrada del sub-modulo y del módulo.

Ejemplo: `.submoduloName(localVaribleModuleName)`

```
module adderMod1(A, B, C, Y);
    input A, B, C; output Y; wire n1;
    halfadd first( .A(A), .B(B), .Y(n1) );
    halfadd second( .A(n1), .B(C), .Y(Y) );
endmodule
```



o se puede hacer sin indicar la entrada del sub-modulo,
pero se debe respetar el orden de los parámetros.

```
    ...
    halfadd first(A, B, n1);
    halfadd second(n1, C, Y);
    ...
```

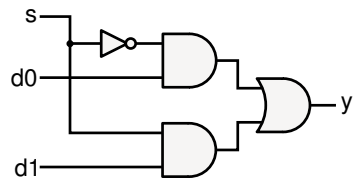
Ejemplo: Estilo Structural

Las compuertas en Verilog están definidas como **primitivas**,
y pueden ser **instancias** y utilizadas como módulos.

```
module mux2(d0, d1, s, y);
    input d0;
    input d1;
    input s;
    output y;
    wire ns, y1, y2;

    not g1 (ns, s);
    and g2 (y1, d0, ns);
    and g2 (y2, d1, s);
    or  g4 (y, y1, y2);
endmodule
```

En el ejemplo se está instanciando una compuerta not, una compuerta or y dos compuertas and.

Estas están conectadas de forma de construir un multiplexor de 2 entradas con 1 bit de ancho.



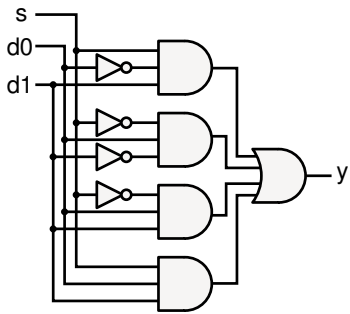
¿Cómo podemos construir un multiplexor de 2 bits de ancho?

Ejemplo: Estilo Behavioral

Utilizando operaciones aritméticas, lógicas, asignaciones y cambios en el flujo de control, podemos modelar un comportamiento específico.

```
module mux2(d0, d1, s, y);
    input d0;
    input d1;
    input s;
    output y;

    assing y = ~d0 &  d1 &  s
             | d0 & ~d1 & ~s
             | d0 &  d1 & ~s
             | d0 &  d1 &  s;
endmodule
```

En este caso utilizamos las operaciones lógicas and, or y not, para definir el comportamiento.



Si bien podemos traducir el código a compuertas de forma directa, **su representación no es la misma.**

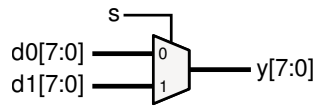
Entonces, ¿Cómo podemos construir un multiplexor de 2 bits de ancho?

Ejemplo: Estilo Behavioral

Para definir comportamientos más complejos tenemos el `if-then-else` *inline*.

```
module mux2(d0, d1, s, y);
    input [7:0] d0, d1;
    input s;
    output [7:0] y;
    assign y = s ? d1 : d0;
endmodule
```

Se asigna d0 a la salida si s es 0 y d1 si s vale 1.



Incluso se pueden anidar.

```
module mux4(d0, d1, d2, d3, s, y);
    input [7:0] d0, d1, d2, d3;
    input s;
    output [7:0] y;
    assign y = s[1] ?
                   (s[0] ? d1: d0)
               : (s[0] ? d3: d2);
endmodule
```

Como las entradas y salidas tiene ancho 8 bits, el circuito se comportará como un multiplexor de 2 entradas de 8 bits de ancho.

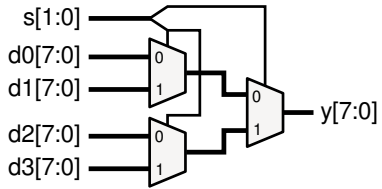
En este caso el operador de asignación condicional describe un comportamiento, **no indica como este debe ser implementado**.

Ejemplo: Estilo Behavioral

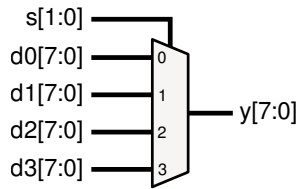
Al definir un comportamiento este puede implementarse de múltiples formas, dependiendo de las **celdas disponibles**.

```
module mux4(d0, d1, d2, d3, s, y);
    input [7:0] d0, d1, d2, d3;
    input s;
    output [7:0] y;
    assign y = s[1] ?
                  (s[0] ? d1: d0)
               : (s[0] ? d3: d2);
endmodule
```

O bien utilizando varios multiplexores en cascada.



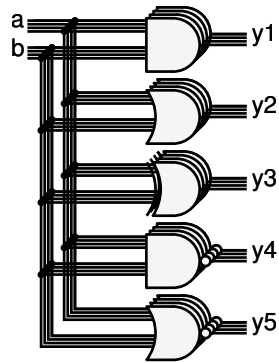
O bien utilizando un solo multiplexor de cuatro entradas.



Ejemplo: Estilo Behavioral

Las operaciones pueden ser aplicadas sobre múltiples bits simultáneamente.
Es decir, utilizarlos como buses de datos.

```
module gate(a, b, y1, y2, y3, y4, y5);
    input [3:0] a, b;
    output [3:0] y1, y2, y3, y4, y5;

    assign y1 = a & b;      // AND
    assign y2 = a | b;      // OR
    assign y3 = a ^ b;      // XOR
    assign y4 = ~(a & b);   // NAND
    assign y5 = ~(a | b);   // NOR

endmodule
```



Ejemplo: Estilo Behavioral

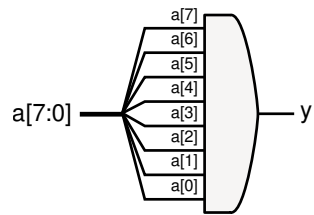
Algunas de las operaciones se pueden usar como **operadores de reducción**.
Estas actúan siempre sobre un conjunto de bits.

```
module and8(a, y);
    input [7:0] a;
    output y;

    assign y = &a;
endmodule
```

Equivalente a la siguiente sentencia,

```
    ...
    assign y = a[7] & a[6] & a[5]
             & a[4] & a[3] & a[2]
             & a[1] & a[0];
    ...
```

En el ejemplo, el resultado en **y** es equivalente a realizar un and entre todos los bits de **a**.

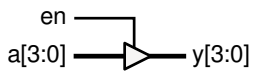


Ejemplo: Buffer de tres estados

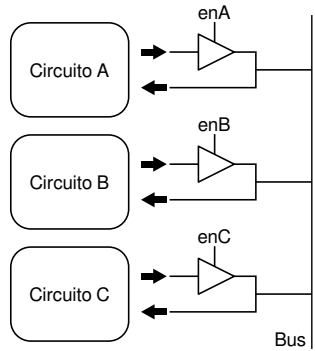
Este es uno de los circuitos más simples que podemos implementar utilizando el valor z.

```
module bufferTresEstados(a, en, y);
    input [3:0] a;
    input en;
    output [3:0] y;

    assign y = en ? a : 4'bzzzz;

endmodule
```

Si la señal **en** está en uno deja pasar la señal **a**, sino deja el circuito en alta impedancia (z).



Este comportamiento se utiliza para compartir un bus entre varias señales.



Respuesta completa de compuertas

Las compuertas como entradas pueden recibir también los estados Z y X.

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

nor	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

xnor	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

Módulos Parametrizados

Permite incluir variables dentro de módulos que pueden ser definidas al momento de la instanciación.

```
module mux2 #(parameter width = 8) (d0, d1, s, y);
    input  [width-1:0] d0, d1; input s;
    output [width-1:0] y;
    assign y = s ? d1 : d0;
endmodule
```

Si no se completa el parámetro, este toma su valor por defecto. Para el ejemplo 8.

```
mux2 i_mux_a(d0, d1, s, out);
```

Es posible instanciarlo indicando los argumentos de forma posicional.

```
mux2 #(12) i_mux_b(d0, d1, s, out);
```

O indicando el nombre de cada uno, incluyendo el parámetro.

```
mux2 #(.width(12)) i_mux_b(.d0(d0), .d1(d1), .s(s), .out(y));
```

Circuitos Secuenciales

Para definir circuitos secuenciales necesitamos una nueva construcción del lenguaje `Verilog`.

```
always @ (sensitivity list)
    statement;
```

La sentencia `always` permite definir un **bloque de expresiones** que serán "ejecutadas" cada vez que alguna variable dentro de la **lista de sensibilidad** (*sensitivity list*) sea modificada.

La lista de sensibilidad debe contener una lista de variables y una indicación de tipo de cambio esperado. Este puede ser:

- `posedge`: Cambia en el flanco ascendente de reloj (Cambio de `0` a 1).
- `negedge`: Cambia en el flanco descendente de reloj (Cambio de 1 a `0`).

Se puede utilizar `*` para indicar que la sentencia es sensible a cualquier cambio, o no indicar cambio y se toma por nivel.

Importante Incluir mal las variables implica un comportamiento final no esperado.

Ejemplo registro

```
module register(clk, d, q);
    input  clk;
    input  [3:0] d;
    output [3:0] q;
    reg [3:0] q;

    always @ (posedge clk)
    begin
        q <= d;
    end

endmodule
```

Las palabras reservadas `begin` y `end` indican donde comienza y termina el bloque `always`.
Si solo se tiene un *statement* dentro del bloque se puede presindir de estas indicaciones.

El valor de q solo se actualizará cuando el `clk` cambie de `0` a 1. Entonces, el valor de d será copiado a q.

Toda variable que será asignada o no dependiendo del tiempo, debe ser declarada como registro (`reg`).
El nombre `reg` no implica directamente una memoria. Dependiendo del contexto se puede materializar como memoria (`flip-flop/latch`) o como cables.

Dentro del bloque `always` no se permite usar `assign`.
Se utilizan *Non-Blocking assingment* (`<=`) y *Blocking assingment* (`=`)

Asignaciones Non-Blocking vs Blocking

NonBlocking (<=)

```
always @ (a)
begin
    a <= 2'b01;
    b <= a;
end
```

- Todas las asignaciones se hacen al final del bloque, **todas al mismo tiempo**.
- Se realizan **en paralelo**, con los valores que YA tenían las variables que son usadas del lado derecho.
- En el ejemplo b toma el valor que a tenia en el pasado y no 2'b01

Blocking (=)

```
always @ (a)
begin
    a = 2'b01;
    b = a;
end
```

- Cada asignación es ejecutada **inmediatamente**.
- Se ejecutan en orden, es similar a la **ejecución secuencial**. Hasta que no se termina de ejecutar la asignación anterior, no se continua con la siguiente.
- En el ejemplo b toma el valor 2'b01

Ejemplos de asignaciones bloqueantes y no bloqueantes

```
always @ (*)
begin
    p <= a & b;
    q <= a ~^ b;
    r <= p | q;
end
```

NonBlocking (<=)

Suponer a=0, b=1. Tanto p, q y r valen cero. Si cambia b a 0. Entonces p será 0, q será 1, **pero r utilizará los valores anteriores de p y q, resultando en cero.**

→ Se asigna en paralelo.

```
always @ (*)
begin
    p = a & b;
    q = a ~^ b;
    r = p | q;
end
```

Blocking (=)

Suponer a=0, b=1. Tanto p, q y r valen cero. Si cambia b a 0. Entonces p será 0, q será 1. **r tomará los nuevos valores de p y q, resultando en uno.**

→ Se asigna secuencialmente.

Reglas de asignación de señales

Usar `always @(posedge clk)` y asignaciones no bloqueantes (<=), implica modelar lógica secuencial.

Ejemplo: `always @ (posedge clk) q <= d;`

Usar asignaciones (`assign`), implica modelar lógica combinacional.

Ejemplo: `assign y = a & b;`

Usar `always @(*)` y asignaciones bloqueantes (=), implica modelar lógica combinatoria más compleja.

Además no es posible hacer asignaciones a la misma señal desde múltiples bloques `always` o simultáneamente en un `assign`.

Ejemplo:

(No se puede)

```
always @(*)
    a = b;
always @(*)
    a = c;
```

```
always @(*)
    a = b;
assign a = c;
```

Sentencia `if`

La sentencia `if` es ejecutada de **forma paralela**.

Se debe pensar como que describe simultáneamente dos flujos diferentes y que ambos se ejecutan todo el tiempo.

Para utilizar más de una sentencia dentro de `if/else` se pueden usar las palabras reservas `begin` y `end` para agrupar el conjunto de sentencias.

Ejemplo:

```
module abs(data, result);
    input [7:0] data;
    output [7:0] result;
    reg [7:0] result;

    always @ (*)
      if (result[7])
        result <= {1'b0, data[6:0]};
      else
        result <= data;
endmodule
```

Sentencia case

La sentencia case permite decidir entre un conjunto de posibilidades.

Su implementación es equivalente a una serie de `if` anidados uno detras del otro.

Existen además las sentencias casez y casex que permiten incluir valores en las etiquetas valores no definidos.

Ejemplo:

```
module GraySeq(seq, gray);
    input [3:0] seq;
    output [3:0] gray;
    reg [3:0] gray;
    always @ (*)
    begin
        case (seq)
        4'd0:    gray = 4'b0000;
        4'd1:    gray = 4'b0001;
        4'd2:    gray = 4'b0011;
        4'd3:    gray = 4'b0010;
        4'd4:    gray = 4'b0110;
        4'd5:    gray = 4'b0111;
        4'd6:    gray = 4'b0101;
        4'd7:    gray = 4'b0100;
        default: gray = 4'b0000;
        endcase
    end
endmodule
```

Sentencia casez y casex

Ejemplo: casez

```
module deco(opcode, command);
    input [4:0] opcode;
    output [2:0] command;
    reg [2:0] command;
    always @ (*)
    begin
        casez (opcode)
        4'b0???1: command = 1;
        4'b0??10: command = 2;
        4'b0?100: command = 3;
        4'b01000: command = 4;
        4'b11000: command = 5;
        default:  command = 7;
        endcase
    end
endmodule
```

casez: No considera como parte de la comparación los bits marcados como ? o z.

Ejemplo: casex

```
module PriorityEncoder(sel, bit);
    input [5:0] sel;
    output [2:0] bit;
    reg [2:0] bit;
    always @ (*)
    begin
        casex (sel)
        6'bxxxxx1: bit = 1;
        6'bxxxx1x: bit = 2;
        6'bxxx1xx: bit = 3;
        6'bxx1xxx: bit = 4;
        6'bx1xxxx: bit = 5;
        6'b1xxxxx: bit = 6;
        default:   bit = 0;
        endcase
    end
endmodule
```

casex: No considera como parte de la comparación los bits marcados como ? o z o x.

Señales sincrónicas y asincrónicas

Los circuitos reciben señales que respetan alguna semántica temporal.

- **Señales Sincrónicas**
  Dependen del reloj, se leen en un momento preciso del tiempo.
- **Señales Asincrónicas**
  Se pueden dar en cualquier momento y afectan al circuito instantáneamente.

Señal de reset

Se utiliza para inicializar el hardware. Setea los circuitos a un estado conocido, o de inicio.

- **Reset Asincrónico**: Independiente del clock.
  Actuá con máxima prioridad (antes que todo).
  Es sensible a *glitches* que pueden generar problemas de estabilidad del circuito.
- **Reset Sincrónico**: Respeta los cambios del clock.
  La activación se debe prolongar por el tiempo necesario para el cambio de clock.
  El resultado es sincrónico con el circuito, no genera problemas de estabilidad.

Ejemplo: Reset asincrónico

```
module register_AR(clk, reset, d, q);
  input  clk;
  input  reset;
  input  [3:0] d;
  output [3:0] q;
  reg [3:0] q;

  always @ (posedge clk, negedge reset)
  begin
      if (reset == 1)
          q <= 0;
      else
          q <= d;
  end
endmodule
```

El reset se chequea al comienzo del bloque, si es 1 entonces se setea q a 0.

La lista de sensibilidad incluye el cambio de la señal de reset a cero.

Es un reset asincronico, porque el cambio puede suceder independiente al cambio del reloj.

Ejemplo: Reset sincrónico

```
module register_SR(clk, reset, d, q);
  input  clk;
  input  reset;
  input  [3:0] d;
  output [3:0] q;
  reg [3:0] q;

  always @ (posedge clk)
  begin
      if (reset == 1)
          q <= 0;
      else
          q <= d;
  end
endmodule
```

El reset se chequea al comienzo del bloque, si es 1 entonces se setea q a 0.

La lista de sensibilidad **solo** incluye al reloj. Por lo tanto solo va a actuar si se altera el reloj.

Es un reset sincronico, porque el cambio por la señal de reset solo puede actuar ante un cambio en la señal de reloj.

Ejemplo: Enable sincrónico y reset asincrónico

```
module reg_SR(clk, reset, en, d, q);
  input  clk;
  input  reset;
  input  en;
  input  [3:0] d;
  output [3:0] q;
  reg [3:0] q;

  always @ (posedge clk, negedge reset)
  begin
      if (reset == 1)
          q <= 0;
      else
          if (en)
              q <= d;
  end
endmodule
```

El registro tiene dos señales adicionales *enable* y *reset*.

Notar que la señal de ***enable*** **no figura en la lista de sensibilidad**.

El comportamiento respeta que q toma el valor de d, solo cuando hay un flanco ascendente del reloj y el enable está en 1.

Ejemplo: Latch

```
module latch(clk, d, q);
    input  clk;
    input  [3:0] d;
    output [3:0] q;
    reg [3:0] q;

    always @ (clk, d)
        if (clk) q <= d;

endmodule
```

El *latch*, a diferencia de un registro, funciona con el valor de nivel del reloj.

Si el nivel es alto toma el valor, si el nivel es bajo no toma el valor.

Con dos latch se puede construir un flip-flop activado por flanco, como *master-slave*.

Lógica combinacional vs lógica secuencial

```
module seq(clk, d, q);
    input clk; input [3:0] d;
    output [3:0] q;
    reg [3:0] q;
    always @ (posedge clk)
        q <= d;
endmodule
```

El bloque *always* describe la señal q.

Esta solo se modifica cuando hay un flanco ascendente de reloj.

Durante el resto del tiempo el circuito expone el valor anterior.

El resultado es un circuito secuencial, genera una memoria, porque **recuerda**.

```
module comb(inv data, result);
    input inv; input [3:0] data;
    output [3:0] result;
    reg [3:0] result;
    always @ (inv, data)
        if (inv)
            result <= ~data;
        else
            result <= data;
endmodule
```

Si `inv` vale 1, expone en `result` el valor de `data` negado bit a bit. Si `inv` vale `0`, no invierte.

Cualquier cambio de `inv` o de `data` va a cambiar `result`, no importa que cambie.

El resultado es un circuito combinacional, no genera memoria, porque **siempre cambia**.

Bloque `always` para circuitos combinatorios

Si bien el bloque `always` nos permite generar circuitos secuenciales,
el resultado de la sintesis será un **ciruito combinatorio** si:

- Todas las señales asignadas en el bloque son **siempre asignadas**,
  en todos los caminos tenemos un valor para la señal (*continuously assignment*).

- Todas las señales del lado derecho **pertenecen a la lista de sensibilidad**.
  Se puede usar `always @*` para implicar a todas las señales.

- Todas las señales en el lado izquierdo **son asignadas para cualquier caso** dentro de todos los bloques. Ojo con asignar parcialmente un bus de señales por error.

Warning: Es muy facil comenter errores y generar memorias (latch) no intencionales.

Bloque always para circuitos combinatorios

Ejemplo: Errores en bloques always para construir circuitos combinatorios

```
...
wire enable, data;
reg a, b;
always @ (*)
begin
    a = 0;

    if(enable)
    begin
        a = ~data;
        b = data;
    end
end
...
```

Error

b no está siendo asignado para cuando enable es false.

```
...
wire enable, data;
reg a, b;
always @ (data)
begin
    a = 1;
    b = 0;
    if(enable)
    begin
        a = ~data;
        b = data;
    end
end
...
```

Error

enable no esta en la lista de sensibilidad.

Bloque `always`: No siempre es práctico

Para cualquier operatoria que requiera código simple de condiciones y asignaciones, donde siempre se asignan datos. Lo ideal es utilizar asignaciones.

Opción 1

```
...
reg  [31:0] result;
wire [31:0] a, b, comb;
wire sel;

always @ (a, b, sel)
begin
    if (sel)
        result <= a;
    else
        result <= b;
end
...
```

Opción 2

```
...
reg  [31:0] result;
wire [31:0] a, b, comb;
wire sel;

assign comb = sel? a : b;
...
```

La opción 2 es más simple y no expone errores de creación de memorias no deseadas.

Definición de tiempos *(Timing relations)*

- Podemos simular tiempos pero no son objetivo para la sintetización.
- No podemos hacer que la sintetización reproduzca los tiempos indicados.
  La temporización está **definida por las celdas** que tengamos en el hardware.
- Los tiempos definidos se usan como **retardos dentro de la simulación**.

```
timescale 1ns/1ps
module simple(input a, output z1, z2);
    assign #5 z1 = ~a;
    assign #9 z2 = a;
endmodule
```

En el ejemplo:

- `timescale 1ns/1ps`
  Definimos la escala de tiempo para la simulación.
- `#5 z1 = ~a;`
  La asignación de z1 será resuelta en 5 unidades de tiempo.

Buenas practicas

A diferencia de escribir código en lenguajes de programación.
En los HDL **debemos** ser muy ordenados y eficientes en el código.
Para esto hay un conjunto de reglas que ayudan en la tarea.

Usar un estilo de nombres consistente

Los nombres de señales, entradas salidas y de cables deben ser claros, evitando ambigüedades, y reduciendo la posibilidad de confusiones.

Orden de los bits

Si bien es posible utilizar cualquier orden para los bits en un bus se recomienda:
`a[31:0]`: Para cualquier número o bus de datos.
`a[0:31]`: Solo para arreglos.

Diseño jerarquico

Cada archivo debe tener un modulo particular, o conjunto de módulos relacionados.
Respetar que el nombre del archivo siga una convención con el nombre del modulo.

Bibliografía

- **“Digital Design and Computer Architecture”**, Second Edition
  David Money Harris, Sarah L. Harris - Morgan Kaufmann - 2013
  - Chapter 4 - Hardware Description Languages → Pag. 173-220

- **“Diseño Digital”**, Tercera Edicion
  M. Morris Mano - Pearson - 2003
  - Capitulo 4-11 HDL para circuitos combinacionales → Pag. 147-160
  - Capitulo 5-5 HDL para circuitos secuenciales → Pag. 190-197

- **“Digital Design and Verilog HDL Fundamentals”**
  Joseph Cavanagh - CRC Press, Taylor & Francis Group - 2008
  - Chapter 4 - Combinational Logic Design Using Verilog → Pag. 231-412

¡Gracias!