Introducción al VERILOG

Ing. Marcelo Casasnovas
Técnicas Digitales I Año 2018 Centro CUDAR

UTN

Indice

- Lenguajes descriptores de hardware
- Verilog
  - Introducción
  - Entidad de diseño (declaración de entidad y cuerpo de arquitectura)
  - Tipos
  - Objetos
  - Operadores
  - Expresiones concurrentes y secuenciales
  - Subprogramas
  - Funciones de resolución
  - Instanciación de componentes
- Bancos de prueba
- Funciones de conversión
- Máquinas de estado

Lenguajes de descripción de hardware

• Introducción

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

Verilog es un lenguaje de descripción de hardware con similitudes a C, pero no es un lenguaje de programación de software.

Cada línea de código Verilog significa que uno o más componentes de hardware se incluyen en el diseño.

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

Lenguajes de descripción de hardware

- **Objetivos**

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

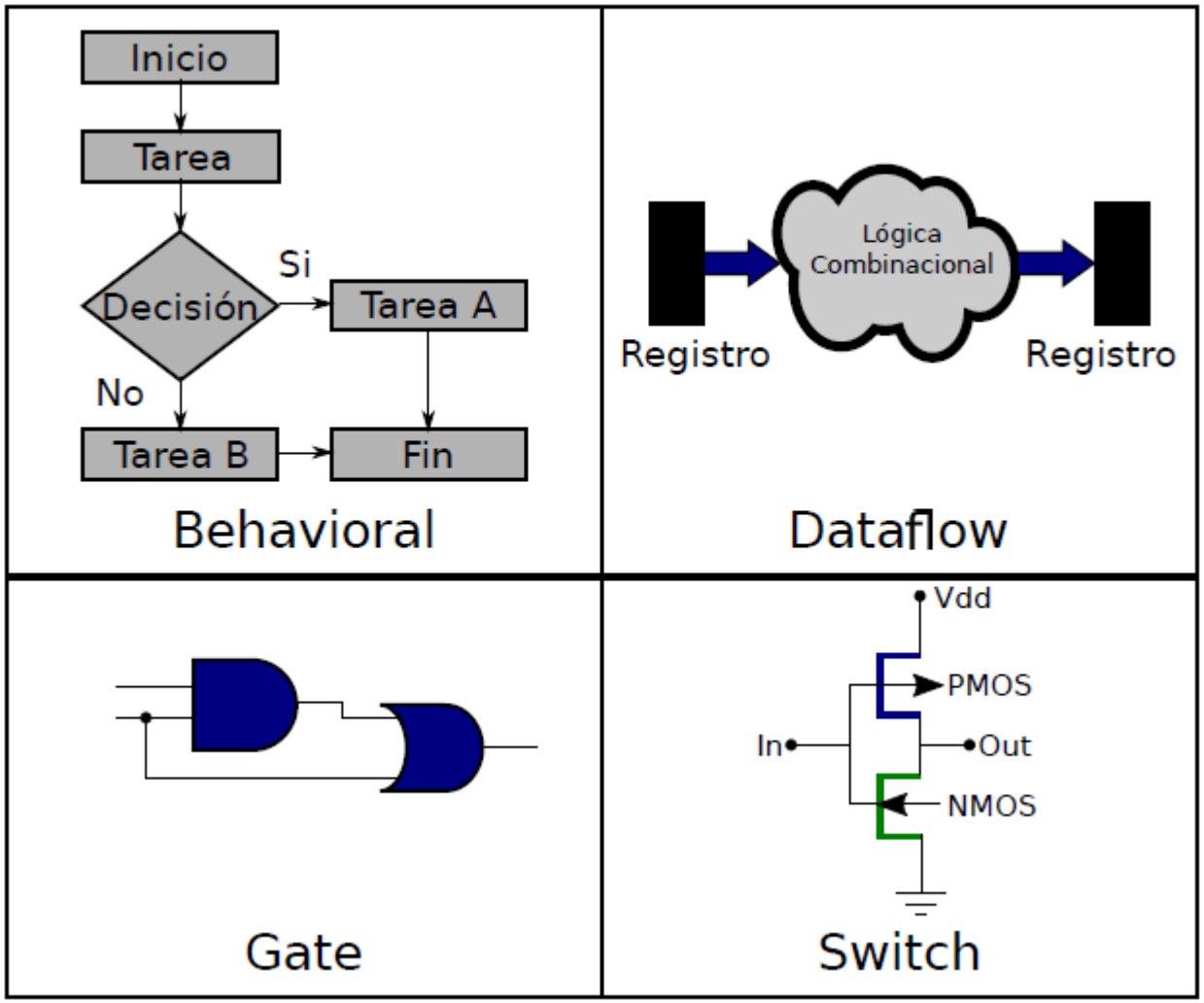
Los 4 niveles de abstracción en Verilog

Verilog es un lenguaje de descripción de hardware (HDL). El hardware(HW) puede ser descripto en varios niveles de detalle. Para capturar estos detalles, Verilog provee al diseñador de los siguientes cuatro niveles de abstracción:

- Nivel Interruptores
- Nivel de compuertas
- Nivel flujo de datos
- Nivel comportamental o algorítmico

Un diseño en Verilog puede estar formado por una mezcla de niveles, desde el nivel de abstracción más bajo como nivel de interruptores, hasta el más alto como el nivel comportamental.

Los 4 niveles de abstracción en Verilog

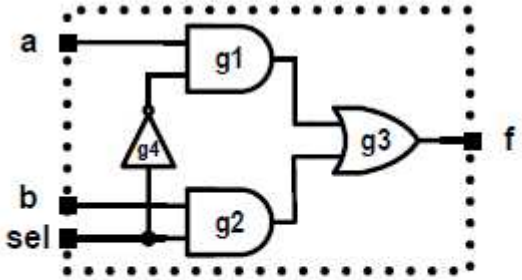


VERILOG

- **Niveles de abstracción en Verilog**

  - **Nivel de interruptores**: es el nivel más bajo de abstracción.

  - **Nivel de compuerta o Estructural**: descripción a bajo nivel del diseño, también denominada modelo estructural. El modelo estructural describe como esta construido un modulo partiendo de bloques simples o primitivas del Verilog, es una aplicación en la que hay jerarquías.

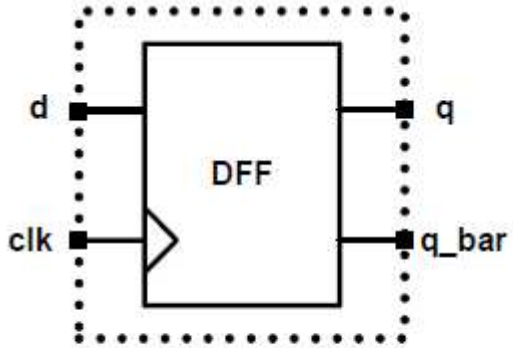



```
module mux(f,a,b,sel);
input          a,b,sel;
output         f;

and     #5     g1(f1,a,nsel),
               g2(f2,b,sel);
or      #5     g3(f,f1,f2);
not            g4(nsel,sel);
endmodule
```

VERILOG

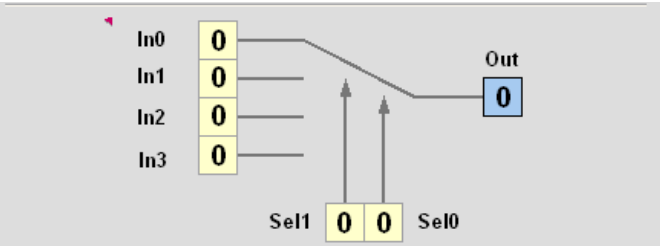
- Nivel de Transferencia de Registro o nivel RTL:

Es un nivel de abstracción superior al nivel de compuertas. Especifican las características de un circuito mediante operaciones y la transferencia de datos entre registros. Expresiones, operandos y operaciones son utilizados con frecuencia en este nivel.




```
module ff(d,clk,q,q_bar);
input          d,clk;
output         q,q_bar;
reg            q,q_bar;
always @(posedge clk)
begin
q      <= #1 d;       // Retardo de 1 unidad
q_bar  <= #1 !d;      // Retardo de 1 unidad
end
endmodule
```

-<u>Nivel de comportamiento (Behavioral level).</u> Es el nivel más alto de abstracción. Proporciona construcciones de lenguaje de alto nivel como **for**, **while**, **repeat**, **if-else** y **case**. La principal característica de este nivel es su total independencia de la estructura del diseño. El diseñador, más que definir la estructura, define el comportamiento del diseño. En este nivel, el diseño se define mediante algoritmos en paralelo. Cada uno de estos algoritmos consiste en un conjunto de instrucciones que se ejecutan de forma secuencial. Este modelo describe lo que el modulo hace. Verilog restringe todas las declaraciones de comportamiento para ser encerrado en un **bloque de procedimiento**. En un bloque de procedimiento todas las variables en el lado izquierdo de las sentencias deben declararse como del tipo reg, mientras que los operandos del lado derecho en las expresiones pueden ser del tipo **reg** o **wire**. Hay dos tipos de bloqueo de procedimiento, **always** e **initial**




```
module mux_4_to_1 (Out,In0,In1,In2,In3,Sel1,Sel0);
output Out;
input In0, In1, In2, In3, Sel0, Sel1;
reg Out;

always @(Sel1 or Sel0 or In0 or In1 or In2 or In3)
  begin
    case ({Sel1, Sel0})
    2'b00 : Out = In0;
    2'b01 : Out = In1;
    2'b10 : Out = In2;
    2'b11 : Out = In3;
    default : Out = 1'bx;
    endcase
  end

endmodule
```

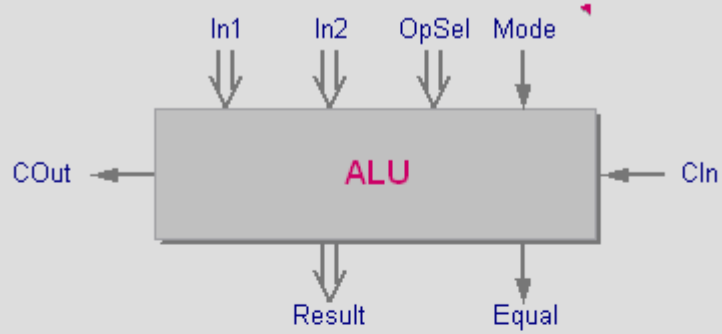
Módulo: un bloque de hardware con entradas/salidas se llama MODULO

```
module function_logica(input logic a, b, c,
output logic y);
assign y = ~a & ~b & ~c |
a & ~b & ~c |
a & ~b & c;
endmodule
```

Un modulo comienza con el listado de puertos de entradas y salidas

Interface del Módulo

- LISTA DE PUERTOS
- DECLARACIÓN DE PUERTOS




```
module ALU (Result, COut, Equal, In1, In2,
            OpSel, CIn, Mode);

output [3:0] Result;   // operation result
output       COut;     // carry out
output       Equal;    // when 1, In1 = In2
input  [3:0] In1;      // first operand
input  [3:0] In2;      // second operand
input  [3:0] OpSel;    // operation select
input        CIn;      // carry in
input        Mode;     // mode arithm/logic;
                       // arithm when 0
 . . .

endmodule
```

Operadores y precedencia en Verilog

Similar a cualquier otro lenguaje de programación

	Op	Meaning
Highest	~	NOT
	*, /, %	MUL, DIV, MOD
	+, -	PLUS, MINUS
	<<, >>	Logical Left / Right Shift
	<<<, >>>	Arithmetic Left / Right Shift
	<, <=, >, >=	Relative Comparison
	==, !=	Equality Comparison
	&, ~&	AND, NAND
	^, ~^	XOR, XNOR
	|, ~|	OR, NOR
Lowest	?:	Conditional

Números

Numbers	Bits	Base	Val	Stored
3'b101	3	2	5	101
'b11	?	2	3	000...0011
8'b11	8	2	3	00000011
8'b1010_1011	8	2	171	10101011
3'd6	3	10	6	110
6'o42	6	8	34	100010
8'hAB	8	16	171	10101011
42	?	10	42	00...0101010
'1	?	n/a		11...111

Entero	Almacenado como	Descripción
1	00000000000000000000000000000001	Unzised 32 bits
8'hAA	10101010	Sized hex
6'b10_0011	100011	Sized binary
'hF	00000000000000000000000000001111	Unzised hex 32 bits
6'hCA	001010	Valor truncado
6'hA	001010	Relleno de 0's a la izquierda
16'bz	zzzzzzzzzzzzzzzz	Relleno de z's a la izquierda
8'bx	xxxxxxxx	Relleno de x's a la izquierda
8'b1	00000001	Relleno de 0's a la izquierda

Operadores

Tipo	Operadores							
Arithmetic	+	−	=	*	/	%	**	
Binary Bitwise		&	&	|	|	^	^	^
Unary Reduction	&	&	|	|	^	^	+	−
Logical	!	&&	||	==	! =	! ===		
Rational	<	>	<=	>=				
Logical Shift	>>	<<						
Arithmetic Shift	>>>	<<<						
Conditional	? :							
Concatenation	{}							
Replication	{{}}							

Ejemplo

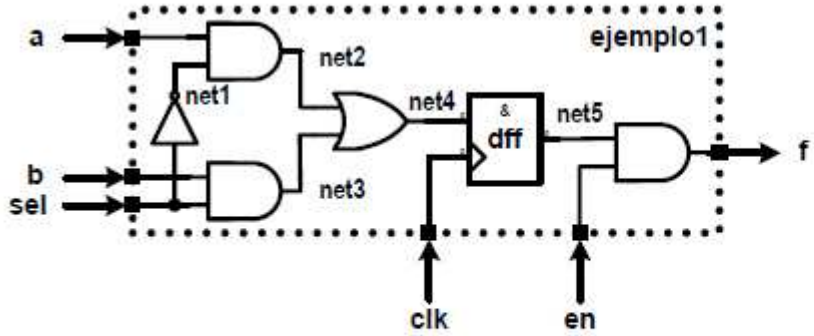
```
// Suma
assign c = a + b;

// Condicional
assign out = sel ? a : b;

if(a ==1'b1)
  out = 1'b1;
else
  out = 1'b0;

// Repeticion
A = 2'b01;
B = {4{A}}; // B = 8'b01010101
```

```
// Logical Shift
A = 6'b101111;
B = A >> 2; // B = 6'b001011

// Arithmetic Shift
A = 6'b101111;
B = A >>> 2; // B = 6'b111011

A = 6'b101111;
B = A <<< 2; // B = 6'b111110

// Reduction
A = 4'b1011;
assign out = &A; // out = 1'b0
```

Tipos de Datos

Existen en Verilog dos tipos de datos principalmente:

- **Nets**. Representan conexiones estructurales entre componentes. No tienen capacidad de almacenamiento de información. De los diferentes tipos de nets sólo utilizaremos el tipo **wire**.
- **Registers**. Representan variables con capacidad de almacenar información. De los diferentes tipos de registeros sólo utilizaremos el tipo **reg** y el tipo **integer**.

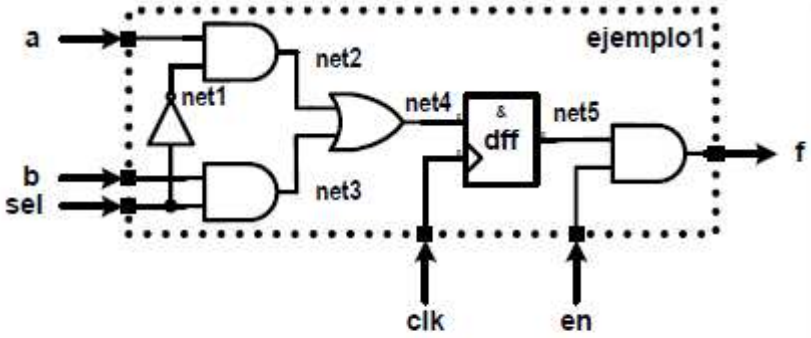



```
module ejemplo1(a,b,sel,clk,en,f);
//Inputs-Ouputs
input          a,b,sel,clk,en;
output         f;
wire           f;
//Descripción de los nodos internos
reg            net5;
wire           net1,net2,net3,net4;
// Descripción del diseño

endmodule
```

Tipos de Datos

Las señales de interfaz y los nodos internos se declaran de la siguiente forma:

- **Inputs**. El tipo de las señales de entrada NO SE DEFINEN, por defecto se toman como **wire**.
- **Outputs**. Las salidas pueden ser tipo **wire** o **reg**, dependiendo si tienen capacidad de almacenamiento de información. En Verilog, un nodo tipo **wire** puede afectar a una salida.
- **Nodos internos**. Siguen la misma filosofía que las salidas.

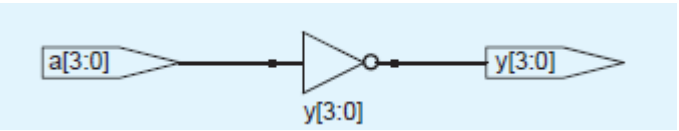



```
module ejemplo1(a,b,sel,clk,en,f);
//Inputs-Ouputs
input          a,b,sel,clk,en;
output         f;
wire           f;
//Descripción de los nodos internos
reg            net4,net5;
wire           net1,net2,net3;
// Descripción del diseño

endmodule
```

VECTORES

```
module adder(input  logic [31:0] a,
             input  logic [31:0] b,
             output logic [31:0] y);

  assign y = a + b;
endmodule
```

```
module inv(input  logic [3:0] a,
           output logic [3:0] y);

  assign y = ~a;
endmodule
```



Notar que las entradas y salidas son buses de 32 bits en el primer ejemplo y de 4 en el segundo

Operaciones Bit a Bit

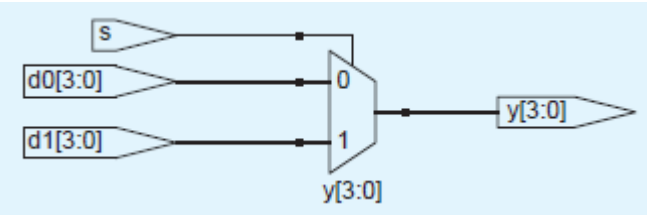
```
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2,
                                y3, y4, y5);

   /* Five different two-input logic
      gates acting on 4 bit busses */
   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
```

~, ^, y | son ejemplos de operadores , mientras que a, b, y y1 son operandos. Una combinación de operadores y operandos, como a & b, o ~ (a | b) se llaman expresiones. Un comando completo como asignar y4 = ~ (a & b); se llama una declaración.

El operador condicional ?

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);

   assign y = s ? d1 : d0;
endmodule
```

If s = 1, then y = d1. If s = 0, then y = d0.



El operador condicional ?: Elige, basado en una primera expresión, entre una segunda y tercera expresión. La primera expresión es llamada la condición. Si la condición es 1, el operador elige la segunda expresión. Si la condición es 0, el operador elige la tercera expresión.

?: es especialmente útil para describir un multiplexor porque, basado en una primera entrada, selecciona entre otros dos. El seguimiento El código muestra la expresión idiomática de un multiplexor 2: 1 con entradas de 4 bits y salidas usando el operador condicional.

Instanciación de componentes

- Los módulos son declarados e instanciados como las clases en C++, pero las declaraciones de módulos no se pueden anidar.

-Las instancias de módulos de bajo nivel están interconectadas y los módulos tienen puertos para estas conexiones.

Sumador completo

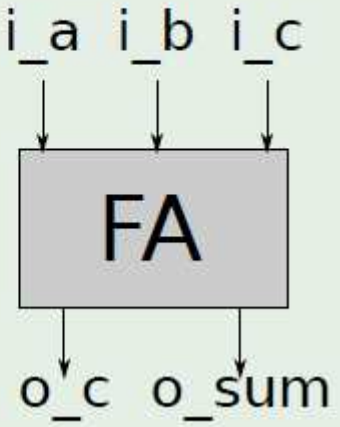
- Modulo

```
module fulladder(<port declaration>);
...
endmodule
```

- Descripción

```
module fulladder(
    output o_sum,
    output o_c,
    input  i_a,
    input  i_b,
    input  i_c);
    assign {o_c,o_sum}=i_a+i_b+i_c;
endmodule
```

- Esquema



Instanciación de componentes

Diseño jerárquico

Jerarquías:

- Verilog funciona de manera eficiente con un concepto de modelado jerárquico.
- El código Verilog contiene un modulo de nivel superior y puede o no tener mas módulos instanciados.
- El modulo de nivel superior (top level) no se instancia en ningún lugar.
- Pueden existir varias instancias de un modulo de nivel inferior.
-Verilog es un HDL y, a diferencia de otros lenguajes de programación, una vez sintetizado cada instanciación infiere una copia física del hardware con sus propias compuertas lógicas, registros y cables.

Instanciación de componentes

- Instanciar significa incluir un modulo dentro de otro.
- La instancia consiste en:
  - -Nombre del modulo (**FA**)
  - -Nombre de la instancia (**u_FA**)
  - -Declaración de puertos
- El orden en la declaración de los puertos es importante dependiendo el estilo que utilicemos.
  - - Verilog 95 solo permite declarar los puertos según el orden que fueron definidos en la declaración del modulo.
  - - Verilog 2001 incorpora la lista de puertos en la instancia sin tener en cuenta el orden que fueron definidos en la declaración del modulo.

Declaración de puertos

- Verilog 95

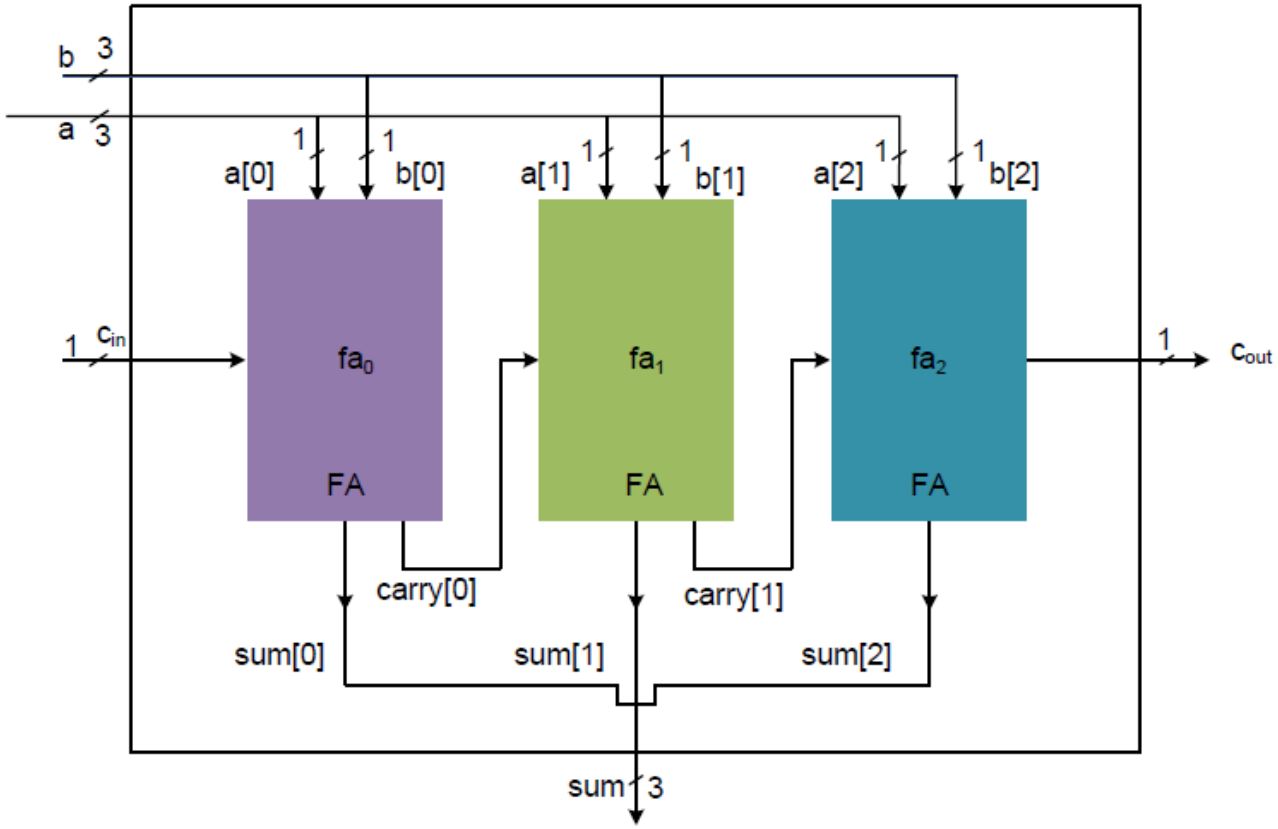
```
module fulladder(
    o_sum, o_c, i_a, i_b, i_c);

   output o_sum;
   output o_c;
   input  i_a;
   input  i_b;
   input  i_c;

   assign {o_c,o_sum}=i_a+i_b+i_c;
endmodule
```

- Verilog 2001

```
module fulladder(
   output o_sum,
   output o_c,
   input  i_a,
   input  i_b,
   input  i_c);
   assign {o_c,o_sum}=i_a+i_b+i_c;
endmodule
```

Instanciación, diseño jerárquico

- **Sumador de dos números de 3 bits**



Instanciación, diseño jerárquico

- **Sumador de dos números de 3 bits**

- Verilog 95

```
module rca(
            input [2:0]  a, b,
            input        c_in,
            output [2:0] sum,
            output       c_out);
   wire                  carry[1:0];
   // module instantiation
   FA u_fa0(a[0], b[0], c_in,
            sum[0], carry[0]);
   FA u_fa1(a[1], b[1], carry[0],
            sum[1], carry[1]);
   FA u_fa2(a[2], b[2], carry[1],
            sum[2], c_out);
endmodule
```

- Verilog 2001

```
module rca(
            input [2:0]  a, b,
            input        c_in,
            output [2:0] sum,
            output       c_out);
   wire                  carry[1:0];
   // module instantiation
   FA u_fa0(.a(a[0]),.b( b[0]),
            .c_in(c_in),
            .sum(sum[0]),
            .c_out(carry[0]));
   FA u_fa1(.a(a[1]), .b(b[1]),
            .c_in(carry[0]),
            .sum(sum[1]),
            .c_out(carry[1]));
   FA u_fa2(.a(a[2]), .b(b[2]),
            .c_in(carry[1]),
            .sum(sum[2]),
            .c_out(c_out));
endmodule
```

Nivel Comportamental, lógica Secuencial, Procesos

El concepto de procesos que se ejecutan en paralelo es una de las características fundamentales del lenguaje, siendo ese uno de los aspectos diferenciales con respecto al lenguaje C.
Toda descripción de comportamiento en lenguaje Verilog debe declararse dentro de un proceso. Existen en Verilog dos tipos de procesos, también denominados bloques concurrentes.

Initial. Este tipo de proceso se ejecuta una sola vez comenzando su ejecución en tiempo cero. Este proceso NO ES SINTETIZABLE, es decir no se puede utilizar en una descripción RTL. Su uso está íntimamente ligado a la realización del testbench.

Always. Este tipo de proceso se ejecuta continuamente a modo de bucle. Tal y como su nombre indica, se ejecuta siempre. Este proceso es totalmente sintetizable. La ejecución de este proceso está controlada por una temporización (es decir, se ejecuta cada determinado tiempo) o por eventos. En este último caso, si el bloque se ejecuta por más de un evento, al conjunto de eventos se denomina lista sensible. La sintaxis de este proceso es pues:
always <temporización> o <@(lista sensible)>

```
initial
begin
  clk = 0;
  reset = 0;
  enable = 0;
  data = 0;
end
```

```
always @(a or b or sel)
begin //Sobra en este caso
  if (sel == 1)
      y = a;
  else
      y = b;
  end   //Sobra en este caso
```

Begin, end. Si el proceso engloba más de una asignación procedural (=) o más de una estructura de control (if-else, case, for, etc…), estas deben estar contenidas en un bloque delimitado por begin y end.

Initial. Se ejecuta a partir del instante cero y, en el ejemplo, en tiempo 0 (no hay elementos de retardo ni eventos, ya los trataremos), si bien las asignaciones contenidas entre begin y end se ejecutan de forma secuencial comenzando por la primera. En caso de existir varios bloques initial todos ellos se ejecutan de forma concurrente a partir del instante inicial.

Always. En el ejemplo, se ejecuta cada vez que se produzcan los eventos variación de la variable **a** o variación de **b** o variación de **sel** (estos tres eventos conforman su lista de sensibilidad) y en tiempo 0. En el ejemplo, el proceso always sólo contiene una estructura de control por lo que los delimitadores begin y end pueden suprimirse.

Asignación bloqueante y no bloqueante

Asignación bloqueante

- Una asignación bloqueante es una asignación regular dentro de un bloque de procedimiento.
- Estas asignaciones se llaman bloqueante porque cada asignación bloquea la ejecución de las asignaciones posteriores en la secuencia.
- En el código RTL Verilog, estas asignaciones se utilizan para modelar la lógica combinacional.
- Para que el código RTL deduzca la lógica combinatoria, las asignaciones de procedimiento de bloqueo se colocan en un bloque mismo *always*.
- Los bloques contienen lista de sensibilidad. Sólo ejecuta un bloque siempre si cambia una de las variables de la lista de sensibilidad.
- El método clásico de listado de sensibilidad es escribir todas las entradas en el bloque en un corchete, donde cada entrada está separada por una etiqueta "or". En Verilog-2001 admite listas de sensibilidad separadas por comas. También admite escribir un "*" para la lista de sensibilidad.

Asignación no-bloqueante

- Las asignaciones de procedimientos no bloqueantes no bloquean otras declaraciones en el bloque y estas sentencias se ejecutan en paralelo.
- El simulador ejecuta esta funcionalidad asignando la salida de estas sentencias a variables temporales y al final de la ejecución del bloque estas variables temporales se asignan a variables reales.
- Las variables del lado izquierdo son del tipo *reg*.
- Se utiliza para inferir lógica sincrónica.
- Todas las asignaciones se realizan en paralelo.

Asignación bloqueante y no bloqueante

```
reg sum, carry;

always @ (x or y)

begin

   sum = x^y;

   carry = x&y;

end
```

```
reg sum, carry;

always @ (x, y)

begin

   sum = x^y;

   carry = x&y;

end
```

```
reg sum, carry;

always @ (*)

begin

   sum = x^y;

   carry = x&y;

end
```

Asignación procedural bloqueante con tres métodos de escribir la lista de sensibilidad.

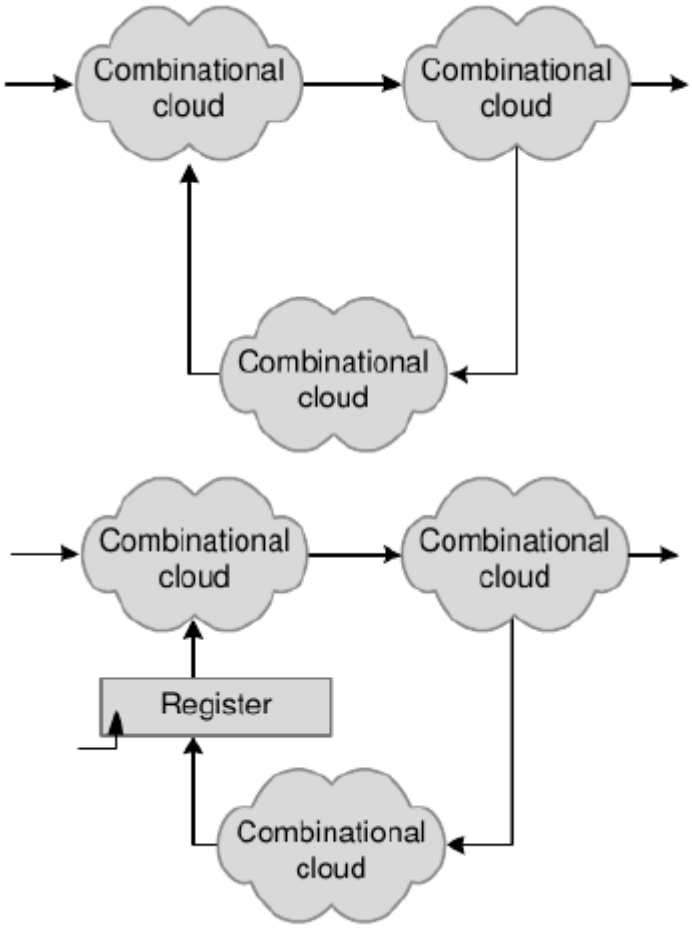
El lado izquierdo de la asignación no bloqueante debe ser del tipo reg. Las asignaciones procedurales no bloqueantes son usadas para inferir lógica sincrónica, el ejemplo de abajo infiere dos registros: sum y carry.

```
reg sum reg, carry reg;
always @ (posedge clk)
begin
   sumreg <  x^y;
   carry reg <  x&y;
end
```

Sistema realimentado

Sistema Realimentado (Feedback)

- Un sistema realimentado, como por ejemplo un acumulador, tienen que ser sistema que incluyan registros entre la lógica combinacional.
- Cualquier circuito que sea solamente combinacional no será sintetizable correctamente por la herramienta.
- Este tipo de diseño no es aceptado en la síntesis.
- Importante: Utilizar una señal de inicialización (reset) en estos registros.
- El reset puede ser síncrono o asíncrona y en ambos casos activo por alto o bajo. Usualmente es activo por bajo porque de esta forma se encuentra disponible en la librerías de tecnologías.



Ejemplos

```
// Registro con reset asincrono activo por
     bajo
always @ (posedge clk or negedge rst_n)
  begin
     if(!rst_n)
       acc_reg <= 16'b0;
     else
       acc_reg <= data+acc_reg;
  end

// Registro con reset asincrono activo por
     alto
always @ (posedge clk or posedge rst)
  begin
     if(rst)
       acc_reg <= 16'b0;
     else
       acc_reg <= data+acc_reg;
  end
```

```
// Registro con reset sincrono activo por
     bajo
always @ (posedge clk)
  begin
     if(!rst_n)
       acc_reg <= 16'b0;
     else
       acc_reg <= data+acc_reg;
  end

// Registro con reset asincrono activo por
     alto
always @ (posedge clk)
  begin
     if(rst)
       acc_reg <= 16'b0;
     else
       acc_reg <= data+acc_reg;
  end
```

Sentencia CASE

La sentencia case evalúa una expresión y *en función de su valor ejecuta la sentencia o grupos de sentencias agrupadas en el primer caso* en que coincida. El uso de múltiples sentencias o asignaciones requiere el uso de begin – end. En caso de no cubrir todos los posibles valores de la expresión a avaluar, es necesario el uso de un caso por defecto (default). Este caso se ejecutará siempre y cuando no se cumplan ninguno de los casos anteriores. En SystemVerilog, CASE debe aparecer dentro de la sentencia ALWAYS.

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);

  always_comb
    case (data)
      //               abc_defg
      0: segments = 7'b111_1110;
      1: segments = 7'b011_0000;
      2: segments = 7'b110_1101;
      3: segments = 7'b111_1001;
      4: segments = 7'b011_0011;
      5: segments = 7'b101_1011;
      6: segments = 7'b101_1111;
      7: segments = 7'b111_0000;
      8: segments = 7'b111_1111;
      9: segments = 7'b111_1011;
      default: segments = 7'b000_0000;
    endcase
endmodule
```

```
module mux4_1(in1,in2,in3,in4,sel,out);
   input  [1 :0] sel;
   input  [15:0] in1, in2, in3, in3;
   output [15:0] out;
   reg    [15:0] out;
   always @(*) begin
      case (sel)
        2'b00: out = in1;
        2'b01: out = in2;
        2'b10: out = in3;
        2'b11: out = in4;
        default: out = 16'bx;
      endcase
   end
endmodule
```

```
always @(op_code)
  begin
     casez (op_code)
       4'b1???: alu_inst(op_code);
       4'b01??: mem_rd(op_code);
       4'b001?: mem_wr(op_code);
     endcase
  end
```

Sentencia IF

La sentencia condicional if – else controla la ejecución de otras sentencias y/o asignaciones (estas últimas siempre procedurales). El uso de múltiples sentencias o asignaciones requiere el uso de begin – end.
Las sentencias always/process pueden contener sentencias IF.
Cuando todas las combinaciones de entrada han sido analizadas, la sentencia implica una lógica combinacional, de otra forma produce una salida secuencial.

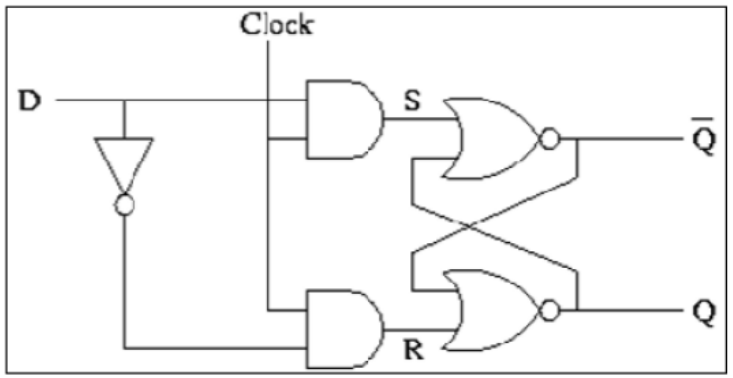
```
module priorityckt(input  logic [3:0] a,
                   output logic [3:0] y);

  always_comb
    if      (a[3]) y = 4'b1000;
    else if (a[2]) y = 4'b0100;
    else if (a[1]) y = 4'b0010;
    else if (a[0]) y = 4'b0001;
    else           y = 4'b0000;
endmodule
```

```
if (brach_flag)
  PC = brach_addr;
else
  PC = next_addr;
```

```
always @(op_code) begin
    if (op_code == 2'b00)
      cntr_sgn = 4'b1011;
    else if(op_code == 2'b01)
      cntr_sgn = 4'b1110;
    else
      cntr_sgn = 4'b0000;
end
```

Evitar latches en el diseño

Un diseñador debe evitar cualquier sintaxis RTL que infiera latches.
Un latch es un dispositivo de almacenamiento que almacena un valor sin el uso de un clock.



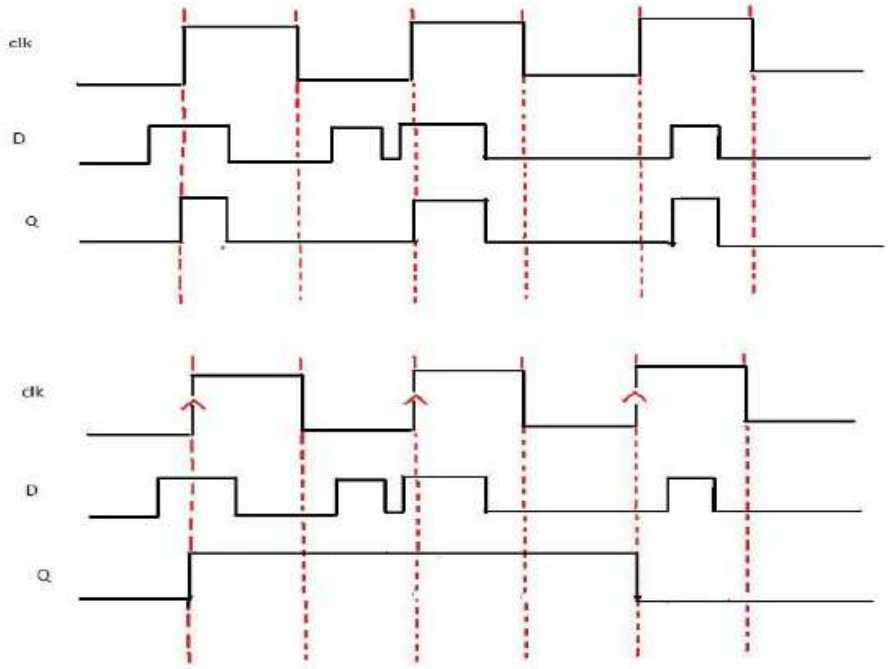


Diagrama de tiempo D-Latch y D-Flip-Flop.

Evitar latches en el diseño

Caso 1

```
input [1:0] sel;
reg   [1:0] out_a,out_b;
always @(*) begin
   if (sel == 2'b00) begin
      out_a = 2'b01;
      out_b = 2'b10;
   end
   else
     out_a = 2'b01;
end
```

```
input [1:0] sel;
reg   [1:0] out_a,out_b;
always @(*) begin
   out_a = 2'b00;
   out_b = 2'b00;
   if (sel=2'b00) begin
      out_a = 2'b01;
      out_b = 2'b10;
   end
   else
     out_a = 2'b01;
end
```

Caso 2

```
always @(*) begin
   out_a = 2'b00;
   out_b = 2'b00;
   if (sel==2'b00) begin
      out_a = 2'b01;
      out_b = 2'b10;
   end
   else if (sel == 2'b01)
     out_a = 2'b01;
end
```

```
always @(*) begin
   out_a = 2'b00;
   out_b = 2'b00;
   if (sel==2'b00) begin
      out_a = 2'b01;
      out_b = 2'b10;
   end
   else if (sel == 2'b01)
     out_a = 2'b01;
   else
     out_a = 2'b00;
end
```

Parámetros (parameter)

- Los parámetros son constantes que son locales a un módulo.
- A un parámetro se le asigna un valor por defecto en el modulo y para cada instancia de este modulo se le puede asignar un valor diferente.
- Los parámetros son muy útiles para mejorar la reutilización de los módulos desarrollados.
- Un modulo se llama parametrizado (parametered) si esta escrito de una manera que el mismo modulo se puede instanciar para diferentes anchos de puertos de entrada y salida.

Declaración

```
module adder(a, b, c_in, sum, c_out);
   parameter SIZE = 4;
   input [SIZE-1: 0] a, b;
   output [SIZE-1: 0] sum;
   input              c_in;
   output             c_out;
   assign {c_out, sum} = a + b + c_in;
endmodule
```

```
module adder
  #(parameter SIZE = 4)
   (input [SIZE-1: 0] a, b,
    output [SIZE-1: 0] sum,
    input              c_in,
    output             c_out);
   assign {c_out, sum} = a + b + c_in;
endmodule
```

Instancia

```
module stimulus;
   reg [7:0] in1, in2;
   wire [7:0] sum_byte;
   reg        c_in;
   wire       c_out;
   adder #(8)
     u_add_byte(in1,
                in2,
                c_in,
                sum_byte,
                c_out);
endmodule
```

```
module stimulus;
   reg [7:0] in1, in2;
   wire [7:0] sum_byte;
   reg        c_in;
   wire       c_out;
   adder #(.SIZE(8))
     u_add_byte(.a(in1),
                .b(in2),
                .c_in(c_in),
                .sum(sum_byte),
                .c_out(c_out));
endmodule
```

Flips Flops: FF D

```
// FPGA usando Verilog
//Código Verilog para Flip FLop D
// con flanco ascendente
module
RisingEdge_DFlipFlop(D,clk,Q);
input D; // Dato de entrada
input clk; // entrada de clock
output Q; // salida Q
always @(posedge clk)
begin
      Q <= D;
end endmodule
```

```
// Código Verilog para para Flip Flop D
// flip flop D con entrada de reset sincrono
module
RisingEdge_DFlipFlop_SyncReset(D,clk,sync_reset,Q);
input D; // Dato de entrada
input clk; // entrada de clock
input sync_reset; // reset sicrónico
output reg Q; // output Q
always @(posedge clk)
 begin
    if(sync_reset==1'b1)
       Q <= 1'b0;
    else Q <= D;
end endmodule
```

```
//Código Verilog para Flip FLop D
// con flanco ascendente y reset asíncrono por alto
Module FlancAsc_DFF_AsyncResetHigh(D,clk,async_reset,Q);
input D; // Dato de entrada
input clk; // entrada de clock
input async_reset; // reset asíncrono por alto
output reg Q; // salida Q
always @(posedge clk or posedge async_reset)
begin
   if(async_reset==1'b1)
     Q <= 1'b0;
   else
     Q <= D;
end endmodule
```

Flips Flops: FF D

```
// FPGA usando Verilog
//Código Verilog para Flip FLop D
// con flanco descendente
module
RisingEdge_DFlipFlop(D,clk,Q);
input D; // Dato de entrada
input clk; // entrada de clock
output Q; // salida Q

always @(negedge clk)
begin
     Q <= D;
end
endmodule
```

```
//Código Verilog para Flip FLop D
// con flanco descendente y reset asíncrono por alto
module FlancAsc_DFF_AsyncResetHigh (D,clk,async_reset,Q);
input D; // Dato de entrada
input clk; // entrada de clock
input async_reset; // reset asíncrono por bajo
output reg Q; // salida Q

always @(negedge clk or posedge async_reset)
begin
  if(async_reset==1'b0)
    Q <= 1'b0;
  else
    Q <= D;
end
endmodule
```

```
// Código Verilog para para Flip Flop D
// flip flop D flanco desc con entrada de reset sincrono
module
RisingEdge_DFlipFlop_SyncReset(D,clk,sync_reset,Q);
input D; // Dato de entrada
input clk; // entrada de clock
input sync_reset; // reset sicrónico
output reg Q; // output Q
always @(negedge clk)
 begin
   if(sync_reset==1'b1)
     Q <= 1'b0;
   else Q <= D;
end endmodule
```

Flips Flops: FF D Testbench

```
`timescale 1ns/1ps;
// Testebench para verificación de flip flop D

module tb_DFF();
reg D;
reg clk;
reg reset;
wire Q;

RisingEdge_DFlipFlop_SyncReset dut(D,clk,reset,Q);
 initial begin
clk=0;
      forever #10 clk = ~clk;
end
initial begin
  reset=1;
  D <= 0;
  #100;
  reset=0;
  D <= 1;
  #100;
  D <= 0;
  #100;
  D <= 1;
 end
endmodule
```

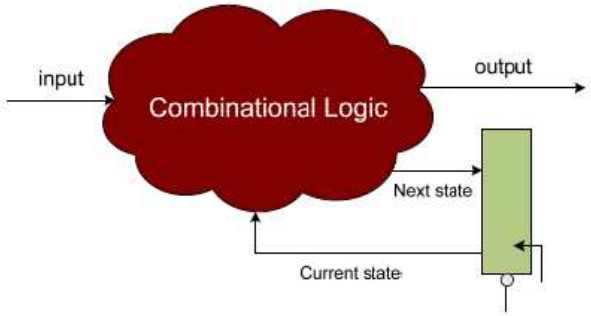
FSM Máquinas de Estado Finitas

Una FSM es un circuito digital secuencial con un registro de estado de N bits, para almacenar el estado actual de la secuencia de operaciones es finita.
Un registro de estado de N bits puede tener 2^N valores posibles, por lo que el número de estados posibles en la secuencia es finito.
Una FSM se puede describir utilizando un diagrama de burbujas(estado), o de forma algorítmica(ASM).
La implementación en hardware de una FSM tiene dos componentes, una nube combinacional y una lógica secuencial.
La nube combinacional calcula la salida y el estado siguiente basada en el estado actual y las entradas, mientras que la parte secuencial posee el registro de estado reseteable.

GLITCHES: Debido a que los tiempos de propagación desde el flanco hasta la salida de los flip-flops pueden tener pequeñas variaciones, el diseño presentará glitches o perturbaciones, debido a las carreras de las entradas al circuito combinacional.
Se denomina glitch (con el significado de deslizar) a un pulso de corta duración, que generalmente es el resultado de una falla en el diseño.

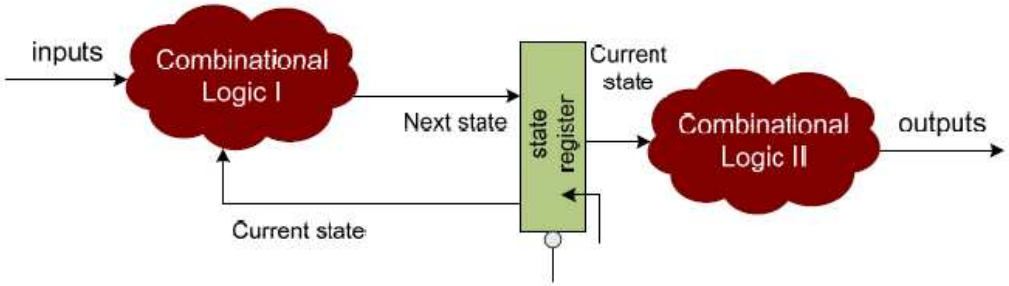
FSM Tipos

Máquina de Mealy



- La salida es función del estado presente y las entradas
- Tiene menos estados que Moore
- Salidas asíncronas
- Si las entradas tienen glitches las salidas también
- Salida disponible inmediatamente
- La salida puede no ser estable el tiempo que es necesario

Máquina de Moore



- La salida es función solo del estado presente
- Tiene mas estados que Mealy
- Salidas síncronas
- No se producen glitches
- Retardo de un ciclo
- Ciclo completo de salida estable

FSM Ejemplo de codificación en Verilog

Diseño de una FSM que cuenta cuatro 1's de una entrada serial y genera un 1 a la salida.

```
always @(current_state or in)
begin
case (current_state)
    S0:
    begin
          if (in)
                    begin next_state = S1; out = 1'b0; end
          else
                    begin next_state = S0; out = 1'b0; end
    end
    S1:
    begin…
    end
endcase
```

Parte combinacional

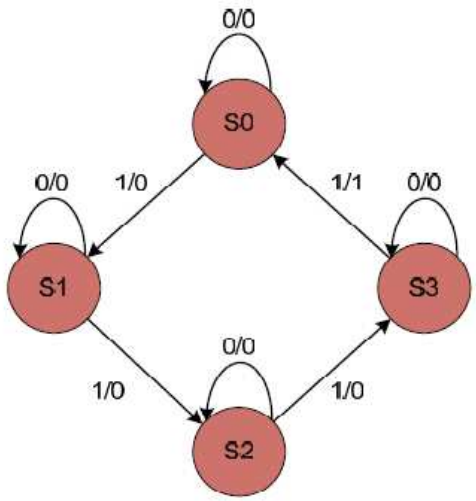
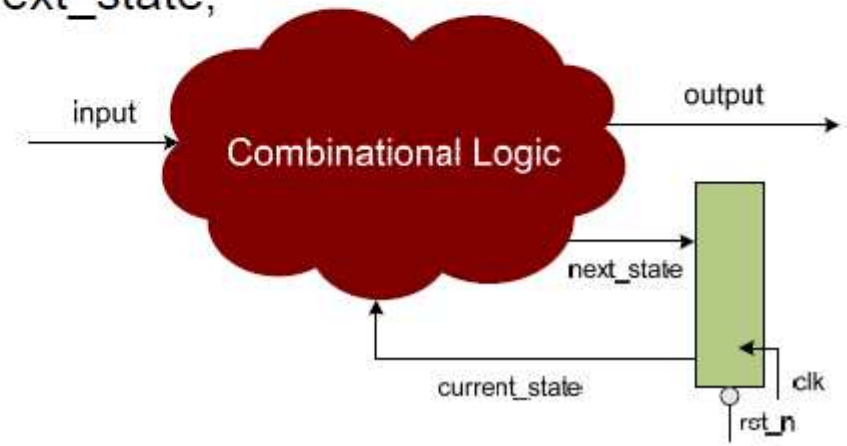


Diagrama de estados

FSM

```
always @(posedge clk or negedge res_n)
begin
if (!res_n)
   current_state <= S0;
else
   curret_state <= next_state;
end
```



Parte Secuencial

FSM Descipción del ejemplo en Verilog

FSM ejemplo de diseño de un semáforo en Verilog

Se va a diseñar el funcionamiento de un semáforo que tiene dos comportamientos diferentes, dependiendo del valor de una entrada externa **on_off** , si está en ‘0’ el semáforo está fuera de servicio y se mostrará un patrón que alternará entre todas las luces apagadas y todas las luces encendidas.
Si está en ‘1’ inicia con la luz verde, luego de un tiempo pasará a la luz amarilla y finalmente a la roja, reiniciando el ciclo hasta que la entrada **on_off** vuelva a estar en ‘0’.

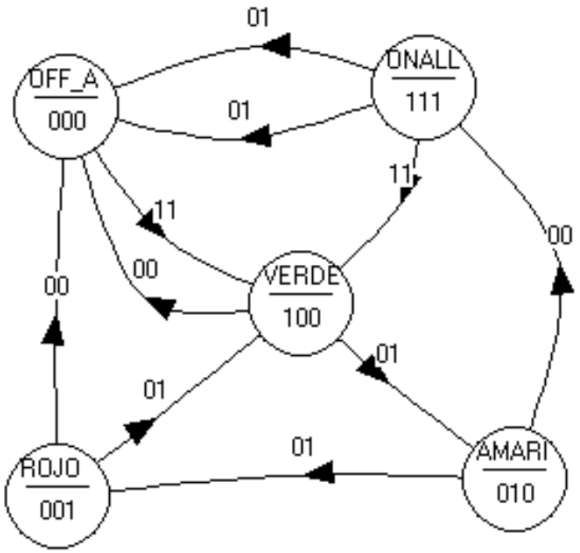


Diagrama de estados

S0 = 00
S1 = 11
CLK = 01

Tabla de codificación de estados

Estado	Luz Verde	Luz Amarilla	Luz Roja
Verde	1	0	0
Amarillo	0	1	0
Rojo	0	0	1
Off_all	0	0	0
On_all	1	1	1

Estado actual	Salida	Entrada SW[0]	Estado siguiente
Off_all	000	0	On_all
On_all	111	0	Off_all
Off_all	000	1	Verde
On_all	111	1	Verde
verde	100	1	Amarillo
amarillo	010	1	Rojo
rojo	001	1	Verde
verde	100	0	Off_all
amarillo	010	0	Off_all
rojo	001	0	Off_all

```
module semaforo(
    input on_off,// DEFINICION DE ENTRADAS Y SALIDAS
    output luz_verde,
    output luz_amarilla,
    output luz_roja,
    input clk
    );

//DEFINICION DE ESTADOS

parameter on_all = 3'b111;
parameter off_all = 3'b000;
parameter verde = 3'b100;
parameter amarillo = 3'b010;
parameter rojo = 3'b001;

reg [2:0] estado = off_all;
```

```
// TRANSICIÓN DE LA MÁQUINA
always @ (posedge clk)
begin
   case(estado)
         on_all: if (on_off)
                     estado <= verde;
                     else
                     estado <= off_all;
         off_all: if (on_off)
                     estado <= verde;
                     else
                     estado <= on_all;
         verde:   if (on_off)
                     estado <= amarillo;
                     else
                     estado <= off_all;
         amarillo: if (on_off)
                     estado <= rojo;
                     else
                     estado <= on_all;
         rojo:     if (on_off)
                     estado <= verde;
                     else
                     estado <= off_all;
         default: estado <= verde;
   endcase
end

// ASIGNACIÓN DE LAS SALIDAS EN FUNCIÓN DEL ESTADO
assign luz_roja = estado[0];
assign luz_amarilla = estado[1];
assign luz_verde = estado[1];


endmodule
```

Comentario: el apunte no está completo, se entrega para estudio del último parcial.

FIN