

Diseño Digital y Verificación

usando

Verilog / System Verilog

Roberto Pereira Arroyo
Alfonso Chacón Rodríguez



Diseño Digital utilizando HDLs

- Los lenguajes de descripción de hardware (HDLs) permiten modelar sistemas digitales completos.
- Mediante herramientas de software estos modelos pueden luego “sintetizarse” para implementarlos como circuitos reales.
- La utilización de HDLs y su posterior síntesis puede tener como objetivo la creación de un circuito integrado de propósito específico (ASICs) o la implementación del circuito en algún dispositivo de lógica programable.



Diseño Digital utilizando HDLs

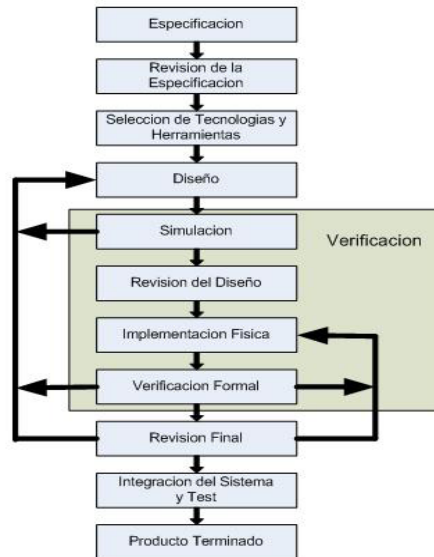
- Al utilizar un HDL es importante tener en cuenta que **se está modelando hardware, y no escribiendo software.**
 - El software se caracteriza por ser secuencial, un procesador ejecutará una instrucción después de otra.
 - En el hardware hay varias tareas que se ejecutan en forma concurrente.
 - Los HDL poseen la habilidad de representar el paso del tiempo y secuenciar los eventos acordemente.
 - Los HDL proveen tipos de datos orientados a hardware. Ej.: variables para representar Hi-Z



Diseño Digital utilizando HDLs

- Existen varios HDLs pero dos son los que predominan: **Verilog** y **VHDL**.
- Casi todo los fabricantes de PLDs proveen soporte para Verilog y VHDL
- Se está investigando en nuevos HDLs que incorporen mayores facilidades y prestaciones a las nuevas tecnologías y necesidades de los desarrolladores. Ej.: SystemC, SystemVerilog

Flujo de diseño de un circuito integrado



VHDL

- Quiere decir VHSIC Hardware Description Language, a su vez, VHSIC proviene de “Very High Speed Integrated Circuit”.
- Surge en 1980 a partir del apoyo del departamento de defensa de Estados Unidos y la IEEE.
- Estandarizado en 1987 (IEEE 1076) conocido como VHDL-87. Extendido y modificado en 1993 (VHDL-93) y en el 2002 (VHDL-2002)



Verilog

- Desarrollado por una empresa privada (*Gateway Design Automation*) en 1984.
- Esta empresa es después adquirida por *Cadence Design Systems*.
- En 1990 Verilog se abre al dominio público.
- En 1995 es estandarizado por la IEEE.

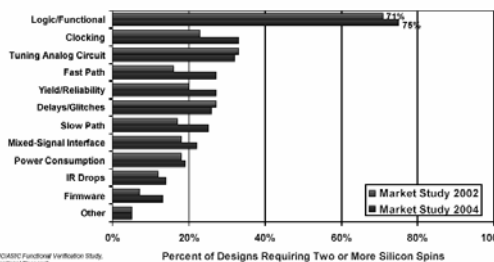


Verificación

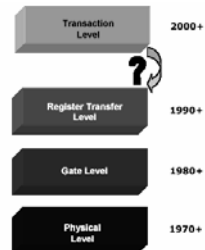
- ¿Por qué se evoluciona hacia los lenguajes de verificación, como SystemVerilog?

```
always @(decade)
abstraction.raise();
```

IC/ASIC Designs Requiring Re-Spins by Type of Flaw

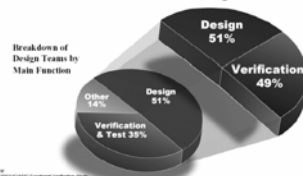


Source:
IC/ASIC Functional Verification Study
© 2004 International Research,
Inc. All rights reserved.



Design Engineers Are Becoming...

Verification Engineers



Source:
© 2004 International Research,
Inc. All rights reserved.



Verificación

- La complejidad creciente de los diseños ya no permiten asegurar diseños funcionales con las limitantes impuestas por los antiguos HDL como VHDL-Verilog
- Los HDL se transforman en HVL (Hardware Verification Languages). Se heredan las propiedades más avanzadas de la programación de software:
 - Verificación basada en aserciones
 - Cobertura funcional
 - Verificación a nivel de transferencias
 - Verificación dirigida por cobertura
 - Test restringido aleatorio
 - Etc.



Plan para las charlas

- Introducción al Verilog HDL
- Introducción a las metodologías de verificación avanzadas
- Introducción al SystemVerilog

Diseño de sistemas digitales por medio de Verilog HDL



Consideraciones de Diseño

- Recordar **siempre** que se está modelando hardware.
- Aplicar la metodología de diseño de ingeniería, recordar siempre en dividir el problema.
- Tener en mente si se va a usar un PLD (y cual) o diseñar un ASIC.
- Definir el sistema en los diferentes niveles de abstracción y en los tres dominios diferentes.

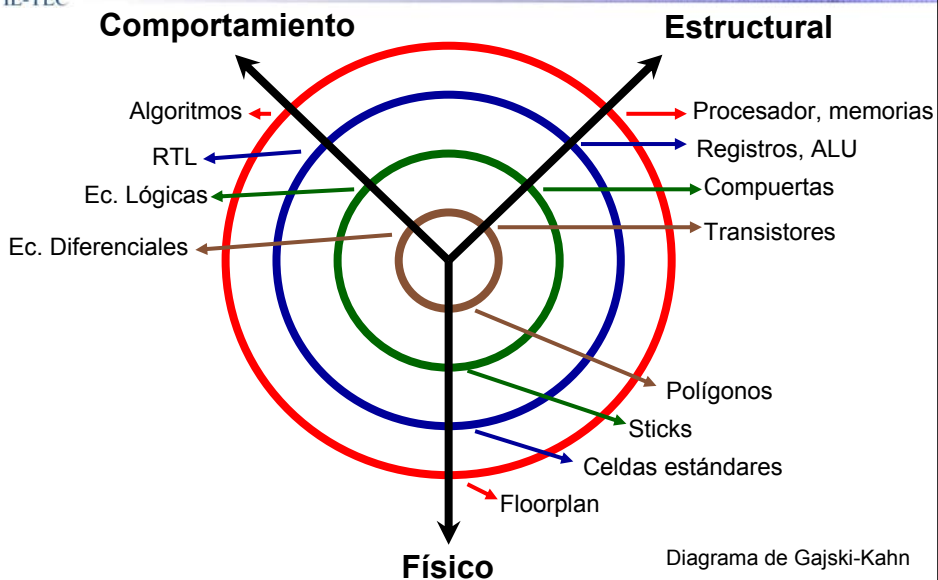


Niveles de abstracción y dominios

- El sistema digital puede definirse en distintos niveles de abstracción y en tres dominios diferentes: **comportamiento, estructural y físico**.
- El diseño debe ser integral, se debe tener presente lo anterior en todo momento.
- Para interpretar estos conceptos se usa el diagrama “Y” de Gajski y Kahn.

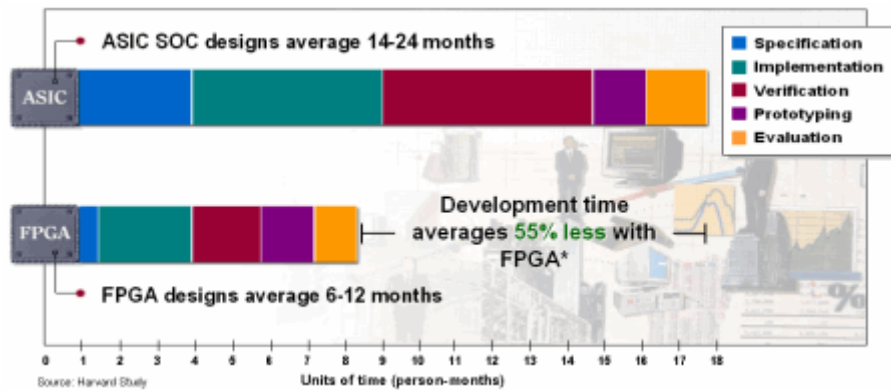


Niveles de abstracción y dominios





ASICs vrs PLDs



Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica

Sintaxis de Verilog



Variables

Estructuras utilizadas para el almacenamiento y manejo de datos en Verilog.

Existen 3 tipos fundamentales de variables:

1. `reg`: Variables que almacenan un valor. Solo puede ser manejada por un proceso. Asignación posible solo dentro de procesos.
2. `wire`: Variables de interconexión. Puede manejarse por varios procesos o asignaciones, y sirve para intercomunicar módulos.
3. `tri`: Nombre alternativo de variable `wire`. Recomendada cuando las señales a inferir son de alta impedancia

Declaración:

Tipo [msb:lsb] nombre ;

Ejemplo:



Otros tipos de datos

- Integers
 - 4'd3, 4'b0100, 6'h20
- Array (para modelar memorias)
 - reg [8:0] ram [0:9];
- Parameter
 - parameter size = 16;
 - wire [size-1:0] bus;
- Preprocessor Directive
 - ``define BEQ 4'b0100`

No “=”



Módulos

Bloque constructivo básico en Verilog.

Sintaxis:

Declaración de módulo ⇒ `module Nombre (Entrada, Salida);`

Declaración de puertos ⇒ `input Entrada;`
`output Salida;`

Instancias ⇒ `Original Copia(Puertos);`

Procesos ⇒ `always @ (ListaDeSensibilidad) begin`
`//Código`
`end`

`endmodule`



Declaración del módulo

Bloque de construcción básico, en Verilog un sistema digital está compuesto por la interconexión de un conjunto de módulos.

Sintaxis

```
module <nombre del modulo> (<señales>);  
    //Código  
endmodule
```



Declaraciones de Puertos

Los puertos son los argumentos del módulo que le permiten comunicarse con el exterior, pueden ser de tres tipos:

- *input*
- *output*
- *inout*



input

Son las entradas al módulo, son variables de tipo `wire` (mismo comportamiento de un cable).

Sintaxis:

```
input [MSB:LSB] Entrada;
```

Ejemplos:

```
input [4:0] Entrada1; //entrada de 5 bits
input Entrada2; //entrada de 1 bit
wire Entrada2;
```



output

Son las salidas del módulo, pueden ser variables de tipo wire, cuando es combinacional o tipo reg (guarda el valor) cuando es secuencial.

Sintaxis:

```
output [MSB:LSB] Salida;
```

Ejemplos:

```
output [3:0] Salida1; //salida de 4 bits
```

```
output Salida2; //salida de 1 bit  
reg Salida2;
```



inout

Son puertos bidireccionales, de tipo wire.

Sintaxis:

```
inout [MSB:LSB] Dididireccional;
```

Ejemplos:

```
inout [4:0] Bider1; // Dididireccional de 5 bits
```

```
inout [7:0] Bider; // Dididireccional de 8 bit  
wire [7:0] Bider;
```



Instancias

Proceso por el cual se crean objetos a partir de un módulo base.

Se tiene un módulo como el que sigue:

```
module Original (CLK, Reset, Variable);  
    //Código  
endmodule
```

Instancia por orden de este módulo sería:

```
Original Copia (localCLK, RESETlocal, VarLocal);
```

Puertos sin conectar se designan dejando vacío el espacio respectivo entre comas.



Instancias

El formato por nombre es preferible:

```
Original Copia (.CLK(localCLK),  
    .Reset(RESETlocal),  
    .Variable(VarLocal));
```

El orden es irrelevante:

```
Original Copia (.Reset(RESETlocal),  
    .Variable(VarLocal),  
    .CLK(localCLK));
```



Procesos

- Son estructuras que ejecutan código definido en ellas.
- Se ejecutan paralelamente, es decir que los procesos se puede ejecutar simultáneamente.
- Los más utilizados son:
 - *initial (no sintetizable)*
 - *always*



initial

Solo se ejecutan una vez, este proceso no es sintetizable. Muy útil para simulación.

Formato:

```
initial begin
    //Código
end
```

Ejemplo:

```
initial begin
    A = 0;
    C = D && E;
end
```



always

- Se ejecutan siempre que ocurra algún cambio en la lista de sensibilidad.
- La lista de sensibilidad son variables, las cuales, al momento que cambian de valor, activan el `always`.
- El `always` es utilizado para definir tanto procesos combinacionales como secuenciales.



always

Formato:

```
always @ ( <lista de sensibilidad> ) begin
    //Código
end
```

Ejemplos

```
always @ ( A or B or C ) begin
    if(B) begin
        E = C;
        D = 1
    end else
        E = 0;
        D = 0;
    end
end
```



always

Ejemplos

```
always @ (posedge CLK) begin
    if(Reset) Var <= 0;
    else Var <= E;
end

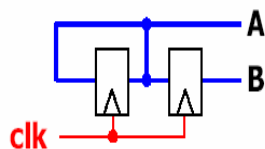
always @ * begin // Al solo poner '*' es sensible
    if(B) begin // a todas las entradas del
módulo
        E = C;
        D = 1
    end else
        E = 0;
        D = 0;
    end
end
```



Asignaciones: Blocking y Non-blocking

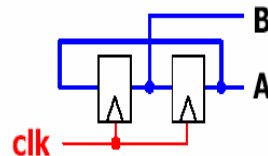
Bad: Circuit from blocking assignment.

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



Good: Circuit from nonblocking assignment.

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





If-else y case

- If (condition 1) begin

.....

end

- else if (condition 2) begin

.....

end

- else begin

.....

end

- reg [1:0] state;

case (state)

2'b00:

2'b01:

2'b10:

2'b11:

default:

endcase



if statement

- Igual que en C
- Solo dentro de sentencias `always`

reg out;

always @(sel or a or b)

begin

if(sel == 1'b1) out = a;

else out = b;

end



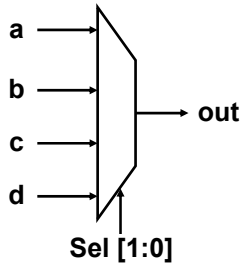
wire out;

assign out = (sel)? a : b;



Uso de case y casex

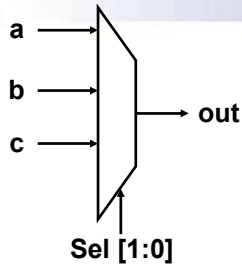
- Muxes y selecciones
- Dentro de un proceso `always`



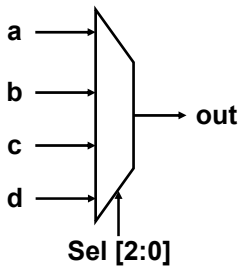
```
always @(sel or a or b or c or d)
begin
  case (sel[1:0] )
    2'b00 : out <= a;
    2'b01 : out <= b;
    2'b10 : out <= c;
    2'b11 : out <= d;
  endcase
end
```



Uso de case y casex (cont.)



```
always @(sel or a or b or c or d)
begin
  case (sel[1:0] )
    2'b00, 2'b11 : out <= a;
    2'b01 : out <= b;
    2'b10 : out <= c;
  endcase
end
```



```
always @(sel or a or b or c or d)
begin
  casex (sel[2:0] )
    3'b011 : out <= a;
    3'b00x : out <= b;
    3'b100 : out <= c;
    default : out <= d;
  endcasex
end
```

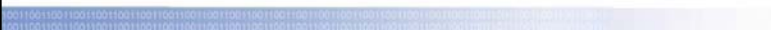
← All others



Otras sentencias comunes



Funciones y tareas





Uso de puertos de un módulo

```
module incremter(go,out,clk,rst);  
  input go,clk,rst;  
  output[11:0] out; ← Si un puerto de salida y una variable  
  reg[11:0] out; ← reg tienen mismo nombre, Verilog infiere un cable entre ellos, "cable implicito"  
  always @ (posedge clk or negedge rst) begin  
    if(!rst)  
      out = 12'b0;  
    else if(go) ← Un puerto de entrada se puede usar directamente en el código pues Verilog infiere un cable (wire) también  
      out = out + 1;  
  end  
endmodule
```



assign (continuous assignment)

- Otra estructura muy usada es el assign, usada para declarar estructuras combinacionales.

– Ejemplo:

```
assign A = B | C;
```

La variable A debe ser de tipo wire

- No se utiliza dentro de un always o initial, pues el assign implica ejecución concurrente, no secuencial

```
always (posedge CLK) begin  
  assign A = B | C;  
end
```



assign

- El assign también se puede usar para construir un "if":

```
assign A = (Condición)? B: C;
```

Si la condición es verdadera: $A = B$, sino $A = C$.

Ejemplos:

```
assign A = (Bandera == 1)? B: C;
```

```
assign A = (Bandera)? B: C;
```

```
assign Salida = (Sel == 0)? Entrada1: Entrada2;
```

```
assign Salida = (!Sel)? Entrada1: Entrada2;
```



Resumen de operaciones sintetizables

- Pueden usarse en el código RTL

<code>`define</code>	<code>`include</code>	<code>module/endmodule</code>
<code>parameter</code>	<code>input</code>	<code>output</code>
<code>wire</code>	<code>reg</code>	<code>begin/end</code>
<code>always</code>	<code>assign</code>	<code>posedge/negedge</code>
<code>if/else</code>	<code>case/casex</code>	<code>function</code>



Operaciones no sintetizables

IE-TEC **No** pueden usarse en el código RTL

- Pueden usarse en los testbench

Delay	initial	repeat
forever	wait	forke
join	event	Time
deassign	force	relace
primitive	task	



Operadores sintetizables

IE-TEC

- Binarios (2 operands)
 - & AND
 - ~& NAND
 - | OR
 - ~| NOR
 - ^ XOR
 - ~^ XNOR
- Lógicos (1 o 2 op.)
 - !A not A
 - A&&B A and B
 - A || B A or B
- Ejemplo
 - assign a= 1;
 - assign b= 0;
 - luego
 - a && b = 0
 - a || b = 1
- Ejemplo
 - a = 4'b0010;
 - luego
 - &a = 1'b0;
 - ^a = 1'b1;



Más ejemplos

- reg [3:0] a;
 always @() begin
 a=b & c;
 end



- wire [3:0] a;
 assign a=b & c

b = 4'b0011
c = 4'b0101

a = 4'b0001

- wire [3:0] a = b | c;



- wire [3:0] a;
 assign a=b | c

b = 4'b0011
c = 4'b0101

a = 4'b0111



Más operadores sintetizables

Operadores de bit

- ~ inverse
- & and
- | or
- ^ XOR
- ~^ XNOR

Ejemplo

- assign a= ~b;
- si b = 4'b0010;
- entonces
- a = 4'b1101;

Shift

- << desplaza izquierda
- >> desplaza derecha

Ejemplo

- a= 4'b0010;
- a >>1 = 4'b0001;
- a <<2 = 4'b1000;

Condicionales

- A = (condition) ? B : C;

Concatenar

- A=2'b10; B=3'b110;
- then {A, B} = 5'b10110



Números

- (tamaño) ' (base)(número)
- $a = 1'b0, 4'b0001, 8'hFF, 10'd700$
- Números negativos
- $-8'd34 = (-34)_{10}, 5'd-4$ (error)
- Underscore (_)
- $a = 20'b00001110110101101010;$
- $a = 20'b00001_11011_01011_01010;$

Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica

Implementación de Circuitos Combinacionales con Verilog



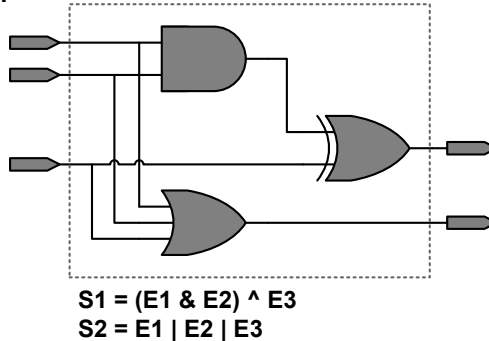


Circuitos Combinacionales

- “Un circuito combinacional consiste en compuertas lógicas cuyas **salidas** en cualquier momento están determinadas por la combinación actual de **entradas**”.

Morris, Mano, M. *Diseño Digital*. 3a. ed, Prentice-Hall, 2003

- Ejemplo:



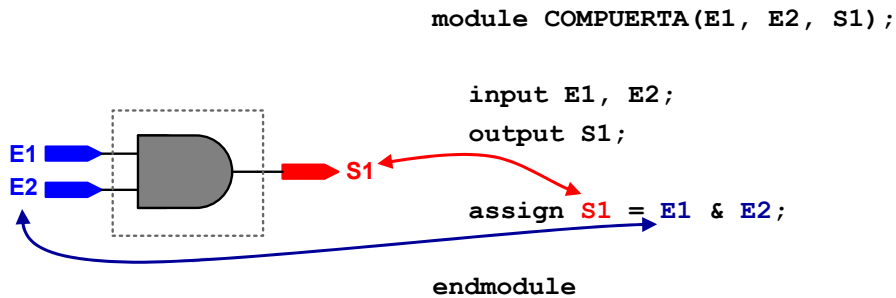
Circuitos Combinacionales

- En el modelo en Verilog de un circuito combinacional se identifican claramente las entradas y salidas.
- Para modelar se utilizan asignaciones (`assign`) y procesos (`always`)



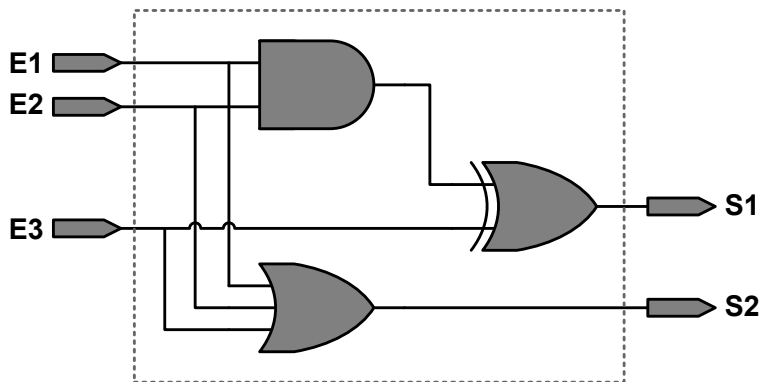
Declaración de operaciones con asignaciones

- Utilizar asignaciones para modelar circuitos combinacionales es sumamente fácil:



Declaración de operaciones con asignaciones

- Como ejemplo, se muestra el modelo en Verilog del siguiente circuito combinacional (se usa paréntesis para separar niveles):





Declaración de operaciones con asignaciones

```
module COMPUERTA(E1, E2, E3, S1, S2);  
    input E1, E2, E3;        output S1, S2;  
  
    assign S1 = (E1 & E2) ^ E3;  
    assign S2 = E1 | E2 | E3;  
  
endmodule
```

Otra forma, declarando una señal intermedia:

```
module COMPUERTA(E1, E2, E3, S1, S2);  
    input E1, E2, E3;        output S1, S2;  
    wire TEMP;  
  
    assign TEMP = E1 & E2;  
    assign S1 = TEMP ^ E3;  
    assign S2 = E1 | E2 | E3;  
  
endmodule
```



Declaración de operaciones con asignaciones

- El `assign` se puede utilizar para realizar operaciones condicionales de la siguiente forma:

```
assign VARIABLE = (CONDICION)? OPCION1: OPCION2;
```

Si CONDICION ES '1', VARIABLE = OPCION1

Si CONDICION ES '0', VARIABLE = OPCION2

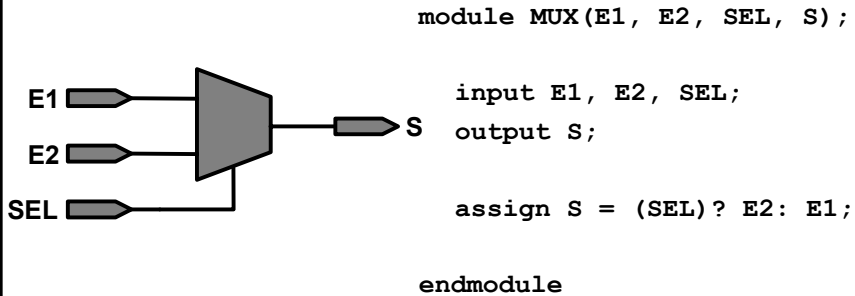
- Se puede anidar esta estructura de selección, ejemplo:

```
assign VARIABLE = (COND0)? ((COND1)? OP0: OP1):  
    ((COND2)? OP2: OP3);
```



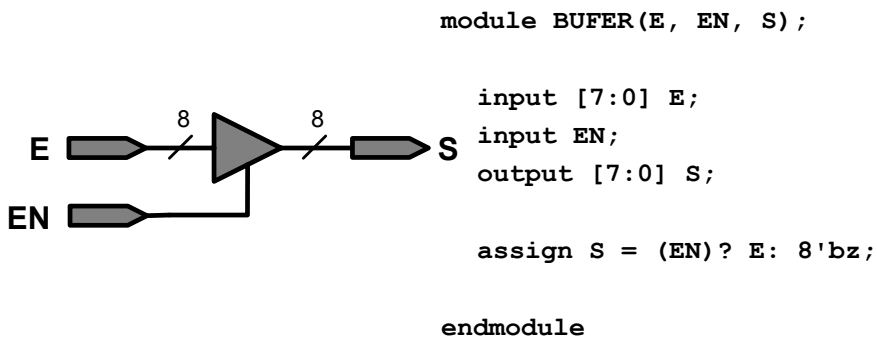
Declaración de operaciones con asignaciones

- Como ejemplo, se muestra el modelo para un multiplexor de 2 a 1, de 1 bit:



Declaración de buffers de alta impedancia

- Los buffers de alta impedancia se modelan de una forma muy sencilla, se utilizan `assign` condicionales.
- Ejemplo: Buffer Hi-Z de 8 bits.





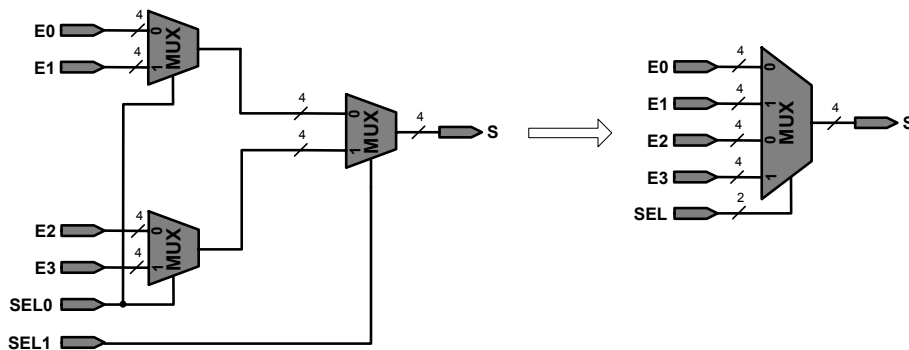
Ejemplo: buffer 16 bits a partir de 2 de 8 bits

```
module BUFER_16(E, EN, S);  
  
    input [15:0] E;  
    input EN;  
    output [15:0] S;  
  
    BUFER BUFER_0(E[7:0], EN, S[7:0])  
  
    BUFER BUFER_1(E[15:8], EN, S[15:8])  
  
endmodule
```



Ejercicio: Mux 4 a 1

- Implementar un Mux de 4 a 1 (4 bits) a partir de Muxes de 2 a 1 (4 bits).





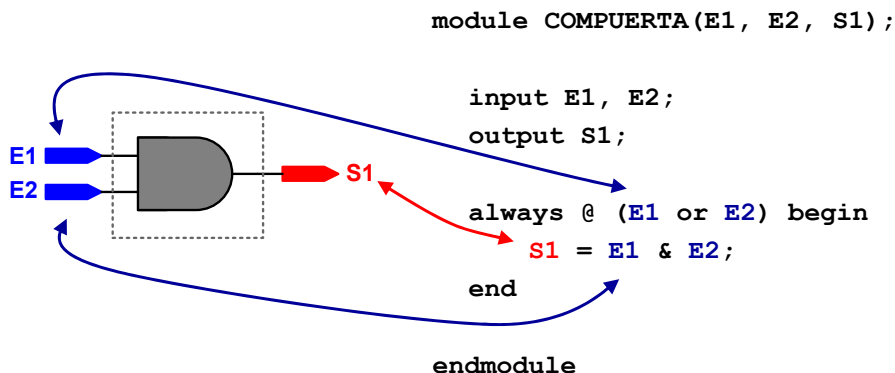
Declaración de operaciones con procesos

- Como se mencionó anteriormente, los `always` pueden ser utilizados para modelar circuitos combinacionales.
- Se deben colocar todas las entradas en la lista de sensibilidad, y declarar todas las opciones. Por ejemplo, si se utiliza un `case`, declarar todas las variables en todas las opciones, o, si se utiliza un `if`, seguirlo con un `else`, etc.
- A partir de Verilog 2001, se puede omitir la lista de sensibilidad, utilizando la sentencia `always@*`. El compilador genera automáticamente la lista, a partir de las variables usadas para tomar decisiones, invocadas por instanciaciones de módulos internos, o que se encuentran a la derecha de una asignación.



Declaración de operaciones con procesos

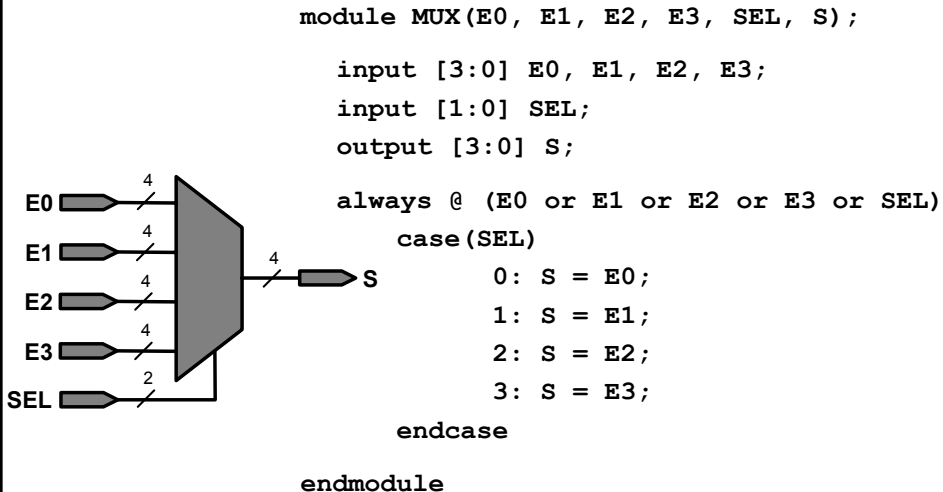
- Por ejemplo, la sintaxis para modelar una compuerta seria:





Declaración de operaciones con procesos

- Ejemplo: Multiplexor de 4 a 1, de 4 bits:



Declaración de operaciones con procesos

- El ejemplo anterior también se puede modelar de la siguiente forma:

```
module MUX(E0, E1, E2, E3, SEL, S);  
  
    input [3:0] E0, E1, E2, E3;  
    input [1:0] SEL;  
    output [3:0] S;  
  
    always @ (E0 or E1 or E2 or E3 or SEL)  
        if (SEL == 0) S = E0;  
        else if (SEL == 1) S = E1;  
        else if (SEL == 2) S = E2;  
        else S = E3;  
  
endmodule
```



Declaración de operaciones con procesos

- **NOTA IMPORTANTE:**

Si llegáramos a omitir alguna opción de case o un if de los dos modelos anteriores, el sintetizador, al notarlo, inferirá un elemento de memoria (en estos casos un latch), para poder guardar el valor anterior, por lo tanto, **YA NO SERÍA COMBINACIONAL**.

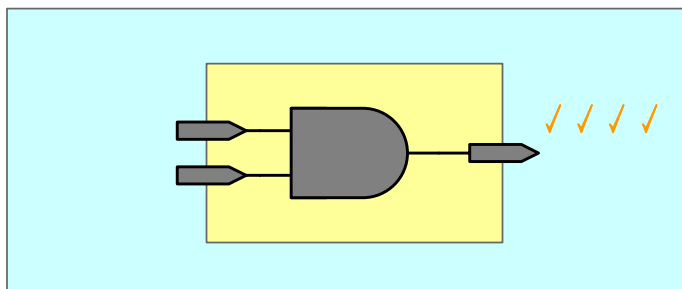
Lo anterior también se presenta si no escribimos todas las entradas en la lista de sensibilidad.

Los sintetizadores muestran un aviso de precaución cuando se genera un LATCH por esta causa.



Creación de bancos de prueba para circuitos combinacionales

- El banco de prueba (test bench) es un módulo el cual instancia a otro el cual deseamos simular para verificar su comportamiento





Ejemplo de un banco de prueba

```
timescale 1ns / 1ps

module PRUEBA_COMPUERTA;

    reg E0, E1; //Entradas
    wire S;      //Salida

    //Instancia del modulo a simular
    COMPUERTA UUT(E0, E1, S);

    initial begin
        E0 = 0; E1 = 0;
        #10 E0 = 0; E1 = 1;
        #10 E0 = 1; E1 = 0;
        #10 E0 = 1; E1 = 1;
    end

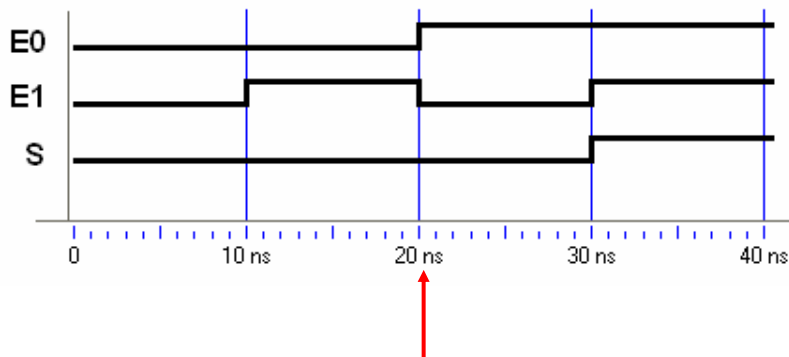
endmodule
```

Tiempo	Eventos
0	E0 = 0; E1 = 0; S = 0
10	E0 = 0; E1 = 0; S = 0
20	E0 = 0; E1 = 0; S = 0
30	E0 = 0; E1 = 0; S = 0



Ejemplo de un banco de prueba (cont)

- Al simular obtenemos la siguiente respuesta:



- Nota: La directiva de “timescale” sirve para definir las unidades de tiempo.

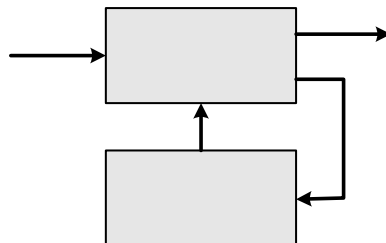
Implementación de Circuitos Secuenciales con Verilog



Circuitos Secuenciales

- “El circuito secuencial recibe información binaria de entradas externas. Estas entradas, junto con el estado presente de los elementos de memoria, determinan el valor binario en las terminales de salida”.

Morris, Mano, M. *Diseño Digital*. 3a. ed, Prentice-Hall, 2003





Circuitos Secuenciales (cont)

- Al modelar circuitos secuenciales en Verilog se utilizan procesos (`always`), las pautas son diferentes en comparación con los enteramente combinacionales.
- En la lista de sensibilidad colocamos el temporizador (el reloj) y opcionalmente señales que queramos darle un comportamiento asíncrono.
- Si no se declaran todas las opciones se infiere un elemento de memoria, por ejemplo:

```
if(condicion) var = entrada;
```

no declaramos un `else`, entonces si “condicion” es ‘0’ el sintetizador entenderá que tiene que guardar el valor anterior y para ello inferirá un elemento de memoria



Sentencias bloqueadoras y no bloqueadoras

- Analice los siguientes procesos, los cuales utilizan sentencias bloqueadoras:

```
always @ (posedge CLK)
```

```
  A = B;
```

```
always @ (posedge CLK)
```

```
  B = A;
```

**Sentencia
bloqueadora**

- No se tiene seguridad de cual se ejecutará primero, si se ejecuta el primer proceso se pierde el valor inicial de ‘A’, en caso contrario ‘B’
- Si se ejecutará el primero, lo ideal sería que el valor de ‘A’ inicial se guardará antes de asignarle el valor de ‘B’.



Sentencias bloqueadoras y no bloqueadoras (cont)

- Esto se puede arreglar utilizando sentencias no bloqueadoras:

```
always @ (posedge CLK)
  A <= B;
```

```
always @ (posedge CLK)
  B <= A;
```

**Sentencia no
bloqueadora**

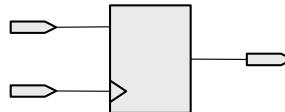
- A la variable de 'A' se le asigna el valor que tenía 'B' al inicio, y a 'B' el que tenía 'A'.
- Las sentencias no bloqueadoras solo se utilizan en procesos.



Ejemplo:

- Al lado del HDL se muestra el circuito inferido:

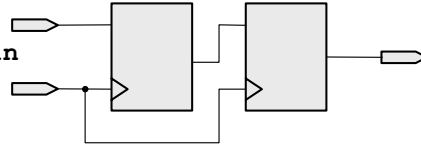
```
module REGISTRO(CLK, X, A);
  input CLK, X;
  output A;
  reg A, B;
  always @ (posedge CLK)begin
    B = X;
    A = B;
  end
endmodule
```





Ejemplo: (cont)

```
module REGISTRO(CLK, X, A);  
  
  input CLK, X;  
  output A;  
  
  reg A, B;  
  always @ (posedge CLK)begin  
    B <= X;  
    A <= B;  
  end  
  
endmodule
```



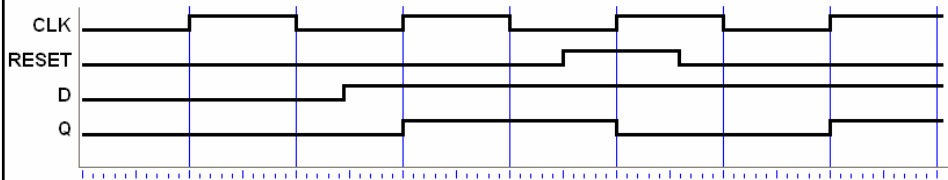
Consejos:

- Utilice sentencias bloqueadoras cuando modele un circuito combinacional en un `always`. Si llegara a usar no bloqueadoras, corre el riesgo de inferir latches no deseados.
- Utilice sentencias no bloqueadoras cuando modele un circuito secuencial.



Ejemplo: Registro con RESET sincrónico

- El registro de este ejemplo es de 1 bit, cuya entrada es 'D' y salida 'Q', sensible al flanco positivo y con RESET sincrónico, el comportamiento puede entenderse con la siguiente figura:



Ejemplo: Registro con RESET sincrónico (cont)

- El HDL sería:

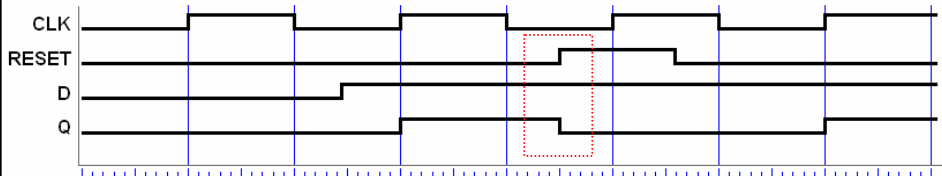
```
module REG_SINC(CLK, RESET, D, Q);  
    input CLK, RESET, D;  
    output reg Q;  
    always @ (posedge CLK)  
        if(RESET)  
            Q <= 0;  
        else  
            Q <= D;  
endmodule
```

posedge indica que es sensible al flanco positivo, para flanco negativo Indicamos negedge



Ejemplo: Registro con RESET asincrónico

- El registro de este ejemplo es de 1 bit, cuya entrada es 'D' y salida 'Q', sensible al flanco positivo y con RESET sincrónico, el comportamiento puede entenderse con la siguiente figura:



Ejemplo: Registro con RESET asincrónico (cont)

- El HDL sería:

```
module REG_ASINC (CLK, RESET, D, Q);  
    input CLK, RESET, D;  
    output reg Q;  
  
    always @ (posedge CLK or posedge RESET)  
        if (RESET)  
            Q <= 0;  
        else  
            Q <= D;  
  
endmodule
```

Consejo: La señal asincrónica debe ser la primera por chequear en los if anidados



Ejemplo: Registro con RESET sincrónico (cont)

- Un posible banco de prueba sería (daría como resultado la figura anterior):

```
`timescale 1ns / 1ps
```

```
module PRUEBA_REG_ASINC(CLK, RESET, D, Q);
```

```
  reg CLK = 0, RESET = 0, D = 0;
```

```
  wire Q;
```

```
  REG_ASINC uut(CLK, RESET, D, Q);
```

```
  initial
```

```
    forever #10 CLK = !CLK;
```

```
  initial begin
```

```
    #20 RESET = 1;
```

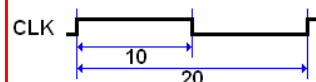
```
    #20 RESET = 0;
```

```
  end
```

```
endmodule
```

Se le asignan valores iniciales a las entradas

Se simula un reloj cuyo periodo es de $2 * 10$:



Ejemplo: Registro bidireccional

- En este ejemplo utilizamos puertos bidireccionales y buffers de alta impedancia.

```
module REG_BIDIRR(CLK, RESET, RD, WR, D_OUT);
```

```
  input CLK, RESET, RD, WR;
```

```
  inout D_OUT;
```

```
  reg VAR;
```

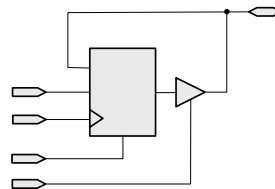
```
  assign D_OUT = (RD)? VAR: 1'bz;
```

```
  always @ (posedge CLK)
```

```
    if(RESET) VAR <= 0;
```

```
    else if(WR) VAR <= D_OUT;
```

```
endmodule
```





Ejemplo: Registro bidireccional (cont.)

- Estimular un puerto bidireccional es algo diferente a una entrada o una salida, prácticamente en el banco de prueba se debe modelar un buffer para poder manejar el flujo de datos.



Ejemplo: Registro bidireccional (cont.)

```
timescale 1ns / 1ps

module PRUEBA_REG_BIDIRR_v;

    //entradas
    reg CLK = 0, RESET = 0, RD = 0, WR = 0;

    //BIDIRR
    reg D_TEMP = 0;
    wire D_OUT;

    assign D_OUT = (WR) ? D_TEMP : 1'bz;

    REG_BIDIRR uut(CLK, RESET, RD, WR, D_OUT);

    initial forever #10 CLK = !CLK;

    initial begin
        #20 RESET = 1;
        #20 RESET = 0;
        #20 RD = 1;
        #20 RD = 0;
        #20 D_TEMP = 1;
        #20 WR = 1;
        #20 D_TEMP = 0;
        #20 WR = 0;
        #20 RD = 1;
    end

endmodule
```

Con estas declaraciones evitamos el conflicto del flujo de datos



Bancos de memoria

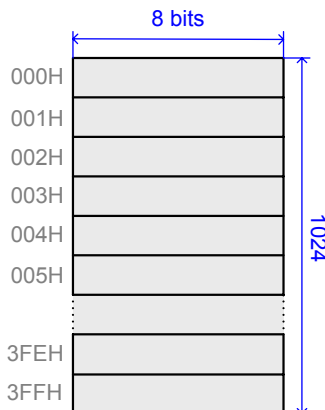
- Para modelar una banco de memoria se usa una variable tipo "reg" en dos dimensiones:

Variable de 8 bits por 1024:

```
reg [7:0] BANCO [1023:0]
```



2^{10} byte = 1024 byte = 1 kbyte



Ejemplo: Bancos de 4 bits x 2k, bidireccional

```
module BANCO_4_X_2K(CLK, WR, RD, DIRR, DATO);  
  
    input CLK, WR, RD;  
    input [10:0] DIRR;  
    inout [3:0] DATO;  
  
    reg [3:0] BANCO [2047:0];  
  
    assign DATO = (RD)? BANCO[DIRR]: 4'bz;  
  
    always @ (posedge CLK)  
        if (WR)  
            BANCO[DIRR] = DATO;  
  
endmodule
```

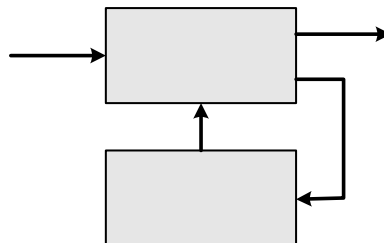
Implementación de Máquinas de Estados Finitos con Verilog



Concepto de Máquina de estados

- “El circuito secuencial recibe información binaria de entradas externas. Estas entradas, junto con el estado presente de los elementos de memoria, determinan el valor binario en las terminales de salida”.

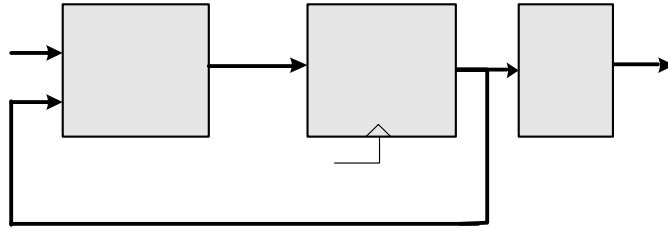
Morris, Mano, M. *Diseño Digital*. 3a. ed, Prentice-Hall, 2003





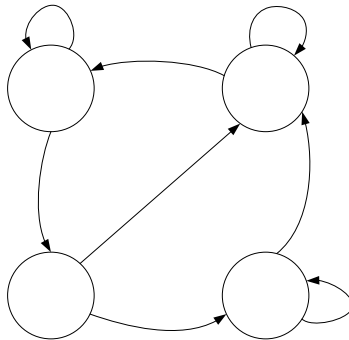
Máquina de Moore

- Las salidas solo dependen del estado actual



Máquina de Moore (cont.)

- Diagrama de estados
- Salidas se asignan en los estados
- Entradas controlan transiciones entre estados



- Tabla de estados

Estado Actual		Entrada X	Siguiete Estado		Salida Z
A	B		A	B	
0	0	0	0	1	0
0	0	1	0	0	0
0	1	0	1	1	1
0	1	1	1	0	1
1	0	0	1	1	0
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	1	1

A DEL
ADO
ENTE

ES
SI

ESTADO A



Máquina de Moore: Implementación en HDL

```
module maq_Moore(X, CLK, RST, Z);
    input X;
    input CLK;
    input RST;
    output reg Z;

    reg[1:0] state;
    parameter S0=2'b00, S1=2'b01, S2=2'b10, S3=2'b11;

    always @ (posedge CLK or negedge RST)
        if (~RST) state <= S0;
        else
            case (state)
                S0: begin
                    Z <= 0;
                    if (~X) state <= S1;
                    else state <= S0;
                end
                S1: begin
                    Z <= 1;
                    if (X) state <= S2;
                    else state <= S3;
                end
                S2: begin
                    Z <= 0;
                    if (~X) state <= S3;
                    else state <= S2;
                end
                S3: begin
                    Z <= 1;
                    if (~X) state <= S0;
                    else state <= S3;
                end
            endcase
    endmodule
```



Otra forma de implementar máquina de Moore

```
module moore_Mejor(X, CLK, RST, Z);
    input X;
    input CLK;
    input RST;
    output reg Z;
    reg [1:0] ESTADO_ACTUAL, ESTADO_SIGUIENTE;
    parameter [1:0] S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;

    always @ (posedge CLK)
        if(RST) ESTADO_ACTUAL <= S0;
        else ESTADO_ACTUAL <= ESTADO_SIGUIENTE;

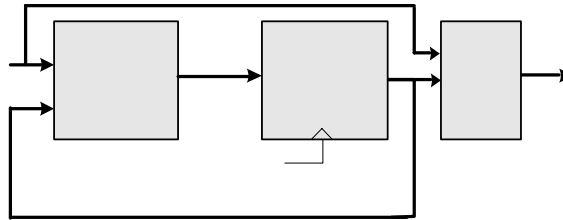
    always @ (ESTADO_ACTUAL or X)
        case(ESTADO_ACTUAL)
            S0:
                if(X) ESTADO_SIGUIENTE = S0;
                else ESTADO_SIGUIENTE = S1;
            S1:
                if(X) ESTADO_SIGUIENTE = S2;
                else ESTADO_SIGUIENTE = S3;
            S2:
                if(~X) ESTADO_SIGUIENTE = S3;
                else ESTADO_SIGUIENTE = S2;
            S3:
                if(~X) ESTADO_SIGUIENTE = S0;
                else ESTADO_SIGUIENTE = S3;
        endcase

    always @ (ESTADO_ACTUAL)
        case(ESTADO_ACTUAL)
            S0:
                Z = 0;
            S1:
                Z = 1;
            S2:
                Z = 0;
            S3:
                Z = 1;
            default:
                Z = 0;
        endcase
    endmodule
```



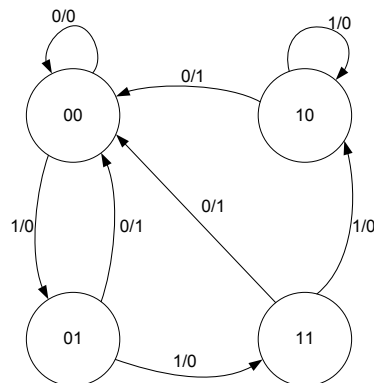
Máquina de Mealy

- Las salidas dependen tanto del estado actual como de las entradas



Máquina de Mealy (cont.)

- Las salidas se activan en las transiciones entre los estados
- Salidas responden inmediatamente a las entradas



Estado Actual		Entrada X	Siguiete Estado		Salida Z
A	B		A	B	
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

A DEL
ADO
ENTE

EST.
SIG.

ESTADO ACT



Máquina de Mealy: Implementación en HDL

```
module maq_Mealy(X, RST, CLK, Z);
    input X;
    input RST;
    input CLK;
    output reg Z;

    reg [1:0] ESTADO_ACTUAL, ESTADO_SIGUIENTE;
    parameter [1:0] S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;

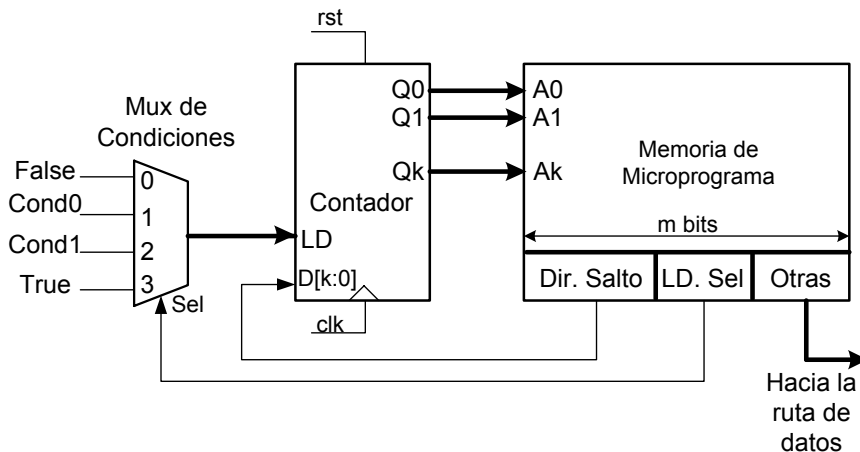
    always @ (posedge CLK)
        if (RST) ESTADO_ACTUAL <= S0;
        else ESTADO_ACTUAL <= ESTADO_SIGUIENTE;

    always @ (ESTADO_ACTUAL or X)
        case (ESTADO_ACTUAL)
            S0:      if (X) ESTADO_SIGUIENTE = S1;
                     else ESTADO_SIGUIENTE = S0;
            S1:      if (X) ESTADO_SIGUIENTE = S3;
                     else ESTADO_SIGUIENTE = S0;
            S2:      if (~X) ESTADO_SIGUIENTE = S0;
                     else ESTADO_SIGUIENTE = S2;
            S3:      if (X) ESTADO_SIGUIENTE = S2;
                     else ESTADO_SIGUIENTE = S0;
        endcase

    always @ (ESTADO_ACTUAL or X)
        case (ESTADO_ACTUAL)
            S0:      Z = 0;
            S1:      if (X) Z = 0;
                     else Z = 1;
            S2:      if (X) Z = 0;
                     else Z = 1;
            S3:      if (X) Z = 0;
                     else Z = 1;
            default:  Z = 0;
        endcase
endmodule
```



Máquina de estados microprogramada





Mux de condiciones: tabla de funcionamiento

LD. Sel	Cond0	Cond1	Función
0	X	X	Incremente contador
1	T	X	Carga paralela de contador
1	F	X	Incremente contador
2	X	T	Carga paralela de contador
2	X	F	Incremente contador
3	X	X	Carga paralela de contador



Características de máquina microprogramada

- Se sustituyen los bloques combinacionales por memoria tipo PROM (Programmable Read Only Memory)
- Con una sola arquitectura se pueden tener muchas máquinas de estados o una muy grande almacenadas en la PROM
- Facilita realizar cambios en un diseño, simplemente debemos cambiar el microprograma almacenado.

Ejemplo de Implementación en Verilog

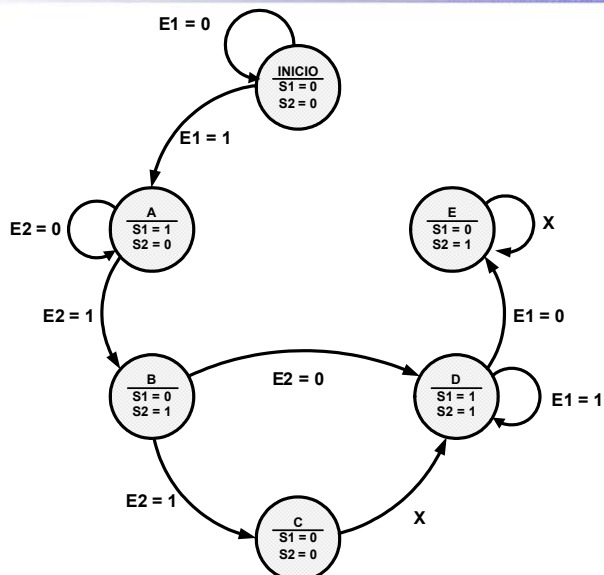
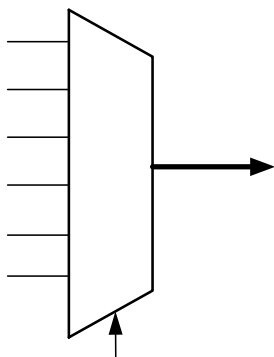


Tabla de microprograma y configuración Mux



Estado Actual	Dir. Salto	L.D. Sel	Salidas
000	000	010	00
001	001	100	10
010	100	100	01
011	000	000	00
100	100	001	11
101	101	101	01



Código Verilog

//Top level

```
module FSM_uPROGRAMADA(CLK, RST, E1, E2, S1, S2);

    input CLK, RST;
    input E1, E2;
    output S1, S2;

    wire [2:0] SEL_MUX;
    wire LD;
    wire [2:0] DIR_SALTO, ESTADO_ACTUAL;

    MUX_8_A_1 MUX CONDICIONES({1'b0, E1, !E1, E2, !E2, 1'b1,
    1'b0, 1'b0}, SEL_MUX, LD);

    COUNT_FSM CONTADOR(CLK, RST, DIR_SALTO, LD, ESTADO_ACTUAL);

    MEM_uPROGRAMA uPROGRAMA(ESTADO_ACTUAL, {DIR_SALTO, SEL_MUX,
    S1, S2});

endmodule
```



Código Verilog (cont.)

//Mux de condiciones y contador de microprograma

```
module MUX_8_A_1(ENTRADAS, SEL, SALIDA);

    input [0:7] ENTRADAS;
    input [2:0] SEL;
    output SALIDA;

    assign SALIDA = ENTRADAS[SEL];

endmodule

module COUNT_FSM(CLK, RST, IN, LD, OUT);

    input CLK, RST;
    input [2:0] IN;
    input LD;
    output reg [2:0] OUT;

    always @ (posedge CLK or posedge RST)
        if(RST)
            OUT <= 0;
        else if(LD)
            OUT <= IN;
        else
            OUT <= OUT + 1;

endmodule
```



Código Verilog (cont.)

```
//memoria de microprograma
module MEM_uPROGRAMA(DIR, SALIDAS);

    input [2:0] DIR;
    output reg [7:0] SALIDAS;

    always @ (DIR)
        case(DIR)
            3'b000:SALIDAS = 8'b000_010_00;
            3'b001:SALIDAS = 8'b001_100_10;
            3'b010:SALIDAS = 8'b100_100_01;
            3'b011:SALIDAS = 8'b000_000_00;
            3'b100:SALIDAS = 8'b100_001_11;
            3'b101:SALIDAS = 8'b101_101_01;
            default:      SALIDAS = 8'b000_101_00;
        endcase

endmodule
```

Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica

Introducción a la verificación / SystemVerilog

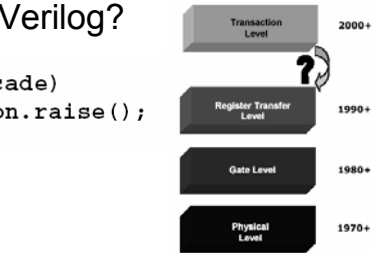
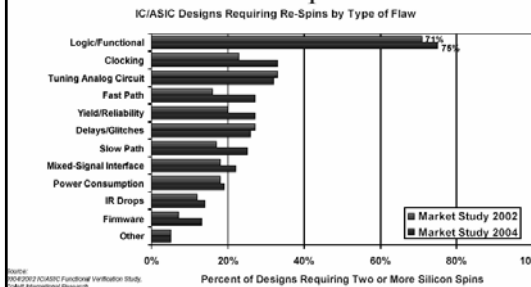




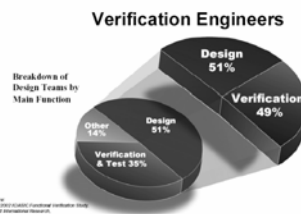
Lenguaje de Verificación

- ¿Por qué se evoluciona hacia los lenguajes de verificación, como SystemVerilog?

always @(decade)
abstraction.raise();



Design Engineers Are Becoming...



Lenguaje de Verificación

- La complejidad creciente de los diseños ya no permiten asegurar diseños funcionales con las limitantes impuestas por los antiguos HDL como VHDL-Verilog
- Los HDL se transforman en HVL (Hardware Verification Languages). Se heredan las propiedades más avanzadas de la programación de software:
 - Verificación basada en aserciones
 - Cobertura funcional
 - Verificación a nivel de transferencias
 - Verificación dirigida por cobertura
 - Test restringido aleatorio
 - Etc.