Circuitos Secuenciales - Sincronismo

HDLs: Verilog

Contenido

- Circuitos Secuenciales
  - Inferencia de Componentes
- Circuitos Síncronos vs. Circuitos Asíncronos

Objetivos de la Clase

- Aprender a diseñar Circuitos Secuenciales en FPGA.
- Comprender una de las reglas básicas del diseño en FPGA: **Sincronismo**.

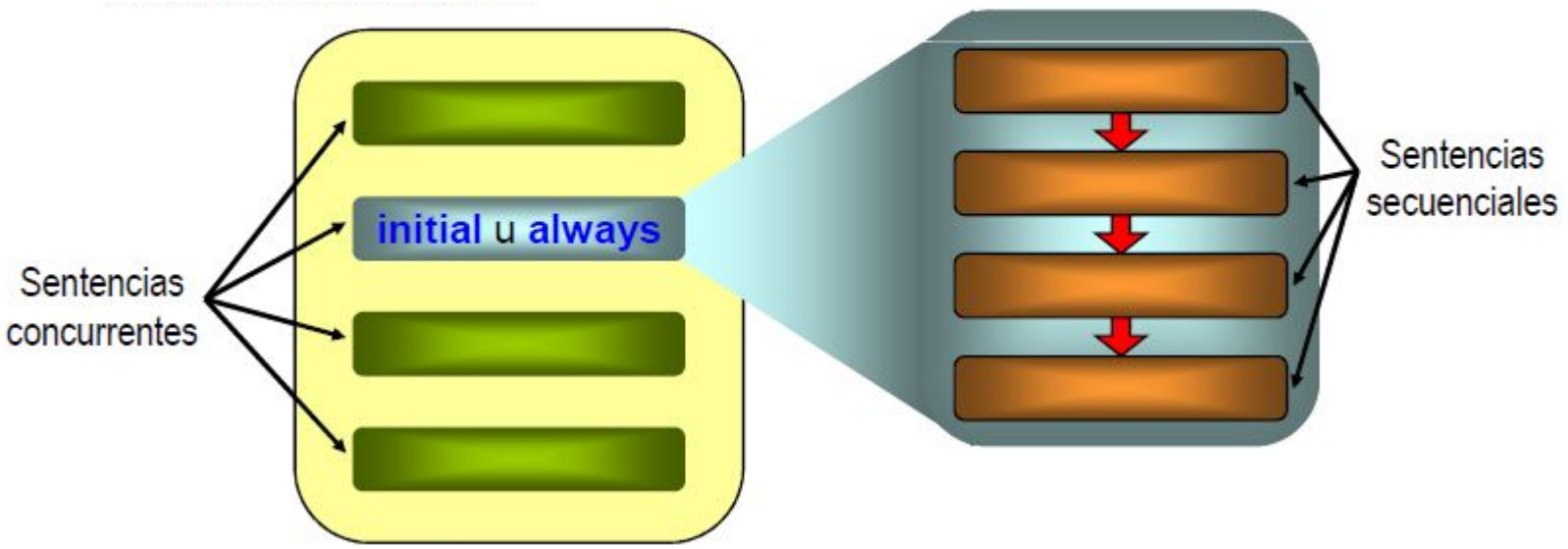
Circuitos Secuenciales

Circuitos Secuenciales

- Los circuitos secuenciales son aquellos que poseen memoria
- La memoria representa el estado interno del circuito
- A diferencia de un circuito combinatorio, en donde la salida es función únicamente de la entrada, la salida de un circuito secuencial es función de la entrada y el estado interno del circuito.

Descripción Algorítmica

Todos estos bloques corren en paralelo, la diferencia es q en el always todo es de manera secuencial



Descripción Algorítmica

```
module test
(
    input  wire i_a, input  wire i_b, input  wire i_clk,
    output reg o_1, output reg o_2
);
    reg tmp1; reg tmp2;

   always @(posedge i_clk)
   begin
        tmp1 <= tmp1 + 1b'1;
        o_1 <= i_a;
   end

   always @(posedge i_clk)
   begin
        tmp2 <= tmp2 + 1b'1;
        o_2 <= i_b;
   end
endmodule // end of Verilog code
```

hacemos sumas que las vamos asignando a o_1 y o_2 que son las que se van para afuera

Descripción Algorítmica: Lista de Sensibilidad

```
always @(lista_de_sensibilidad)

begin [: nombre_bloque]

  /* sentencias
    secuenciales*/

end
```

todo lo que esta dentro de un always se ejecuta en un ciclo de reloj

Descripción Algorítmica: Lista de Sensibilidad

- Eventos regulares
  - Cambio de valor de una señal
  - A la transición de uno a cero (posedge)
  - A la transición cero a uno (negedge)
  - Permite sintetizar lógica secuencial (sincronizada a un clock)
  - **SOLO USAR POSEDGE O NEGEDGE EN SEÑALES DE CLOCK**

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

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

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

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

- No se recomienda mezclar tipos de eventos en la lista sensitiva

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

usar solo algun flanco del reloj

Inferencia de Componentes: Bloques Always

Always usado para lógica combinacional

- La lista de sensibilidad debe ser tipo por nivel.
- Se sintetiza lógica combinacional si la lista de sensibilidad es equivalente a **@(*).**

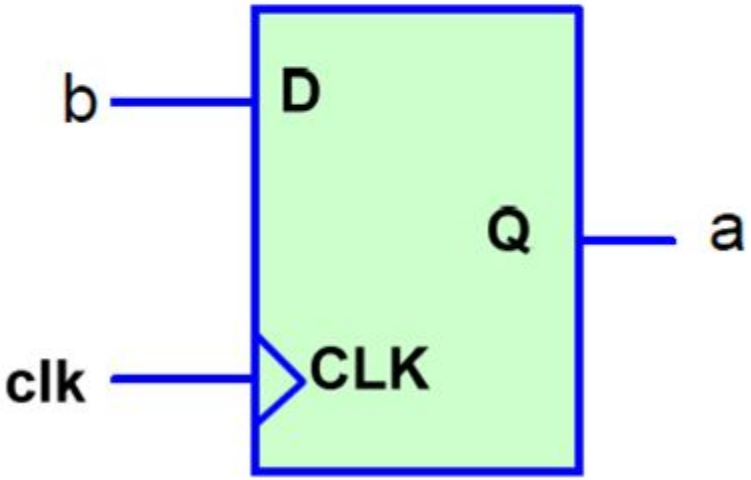
Always usado para lógica secuencial

- Si la lista de sensitividad es por flanco, **flip-flops** son generados.
- Si la lista de sensibilidad es por nivel, y hay ejecuciones del bloque en las que el valor de una variable no es explicitamente asignado, se genera un **latch** para dicha variable.

Inferencia de Componentes: Flip-Flop

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

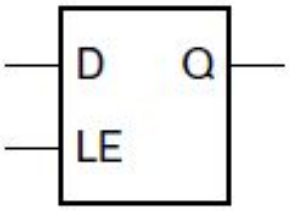
asigno en a lo que tengo en b luego de cada flanco de reloj



Flip-Flops

Inferencia de Componentes: Latch

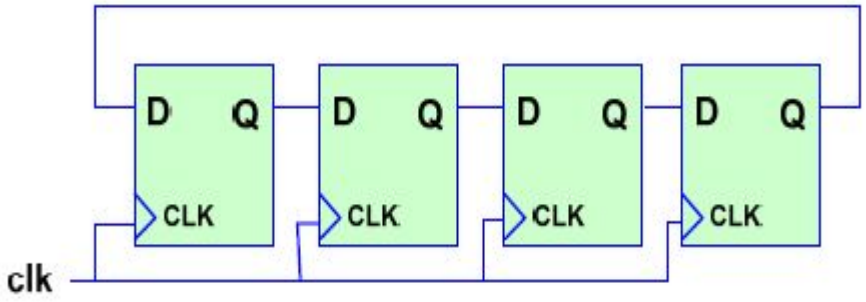
```
always @(LE or D)
  if(LE) Q <= D;
```



si LE no es 1 no se que voy a tener en la salida, no recomendable

Inferencia de Componentes: Registros de Desplazamiento

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



Circuitos Síncronos vs. Asíncronos

Flip-Flops - Latches

Circuito Asíncrono

- La respuesta (salida) del circuito cambia al cambiar sus entradas

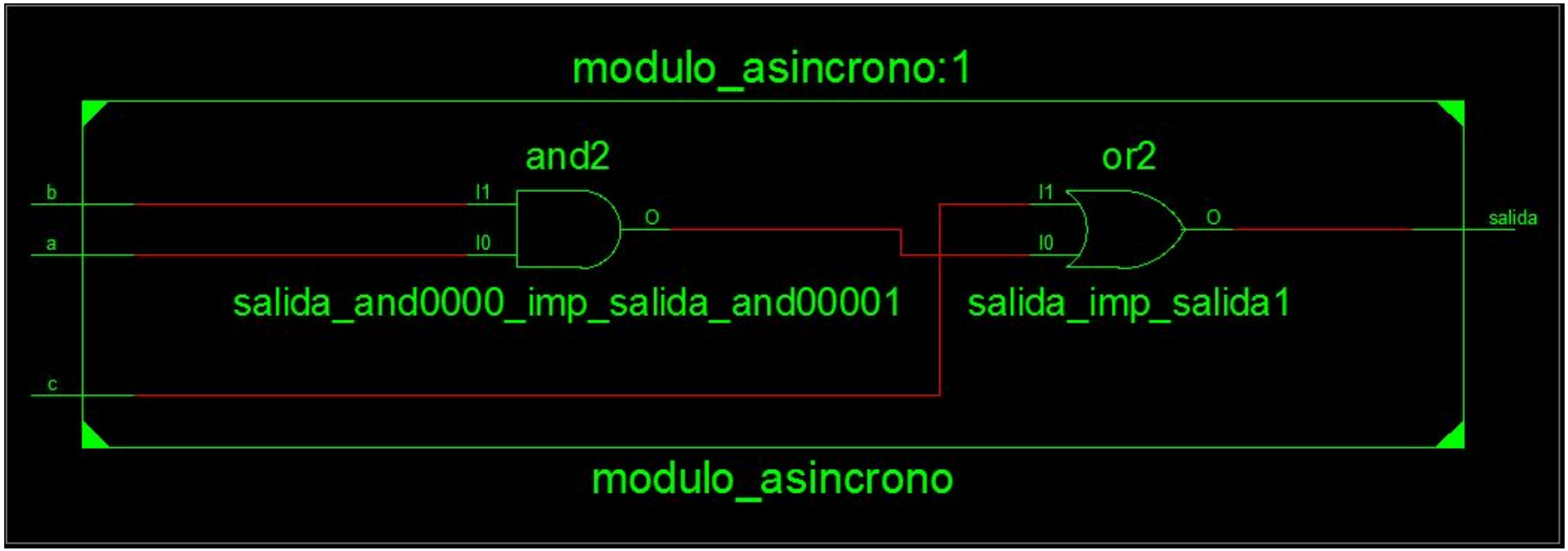
```
always @(a, b, c)
begin
    salida = (a & b) | c;
end
```

el assign lo usamos solamente para concatenar datos o para condiciones simples, sino always

esto es lo mismo que hacer un assign pero lo podemos ver de una forma mas ordenada, y cuando tenemos mas variables se ve esto mas claramente

Además, el always me deja utilizar if else, case, etc

Circuito Asíncrono

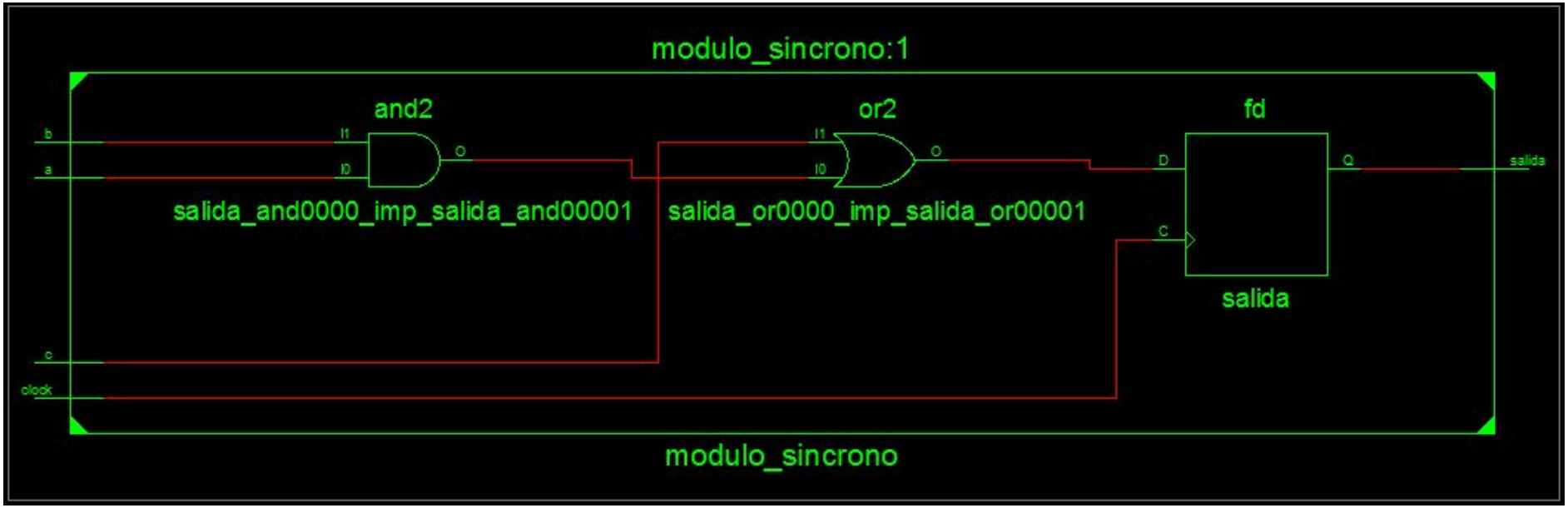


Circuito Síncrono

- La respuesta (salida) del circuito cambia al cambiar sus entradas y cumplir con una condición de reloj

```
always @(posedge clock)
begin
    salida = (a & b) | c;
end
```

Circuito Síncrono

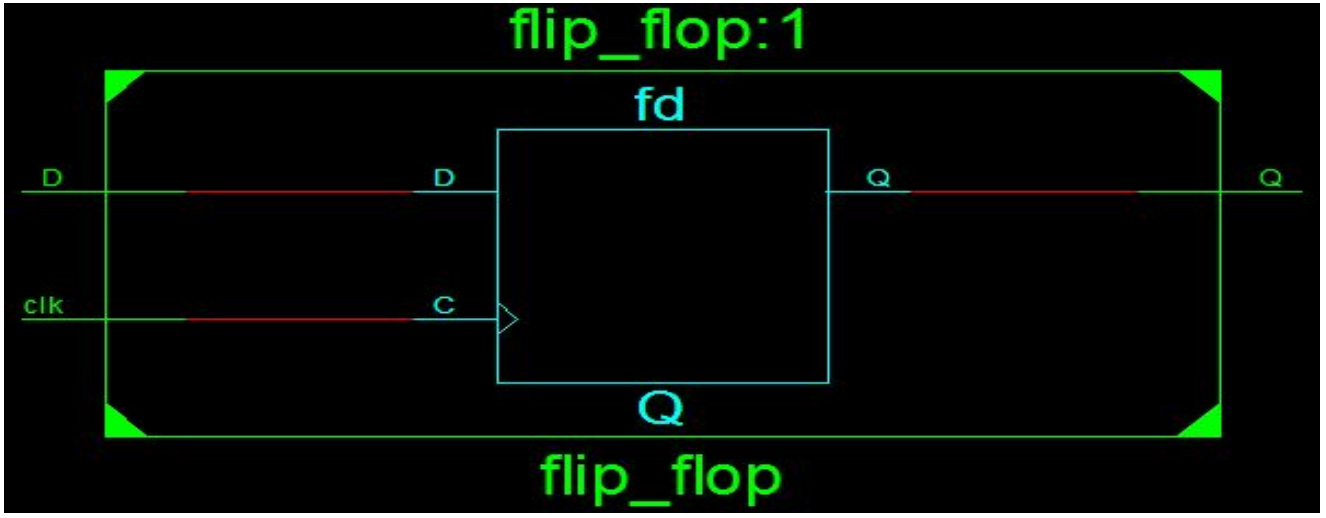


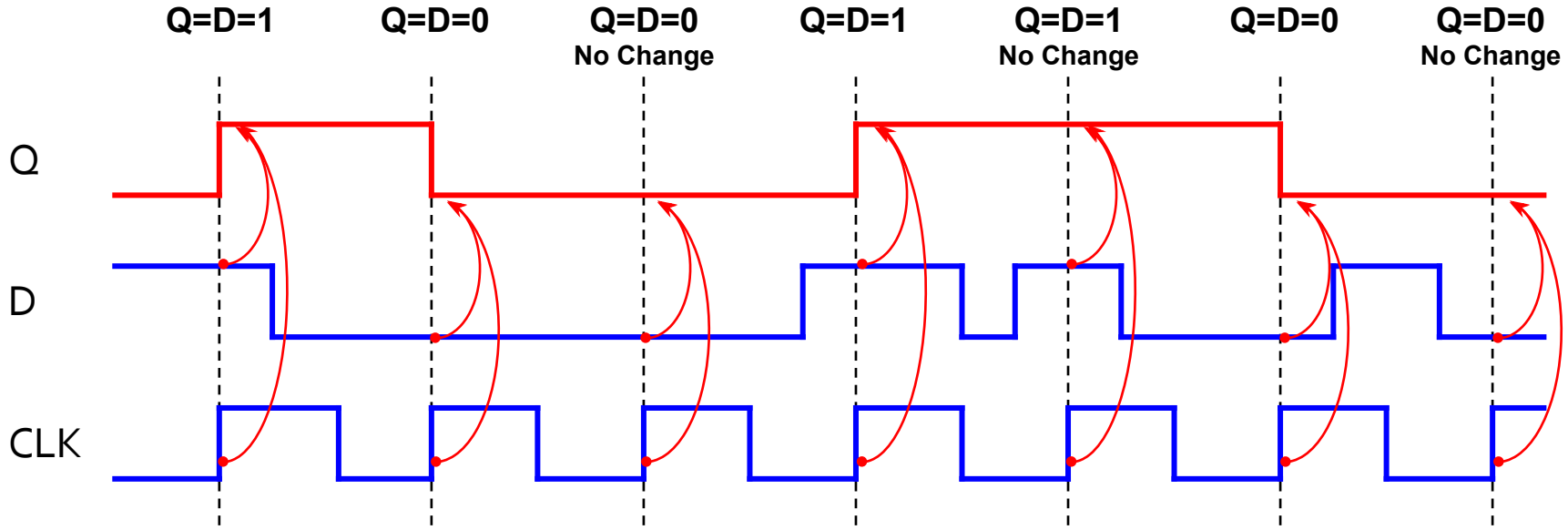
Flip-Flop D

- Dispositivo secuencial que responde a flanco

- `always @(posedge clock)`
- `begin`
  - `salida <= entrada;`
- `end`

Flip-Flop D


flip_flop:1
fd
D
D
Q
Q
clk
C
Q
flip_flop


Q=D=1
Q=D=0
Q=D=0
No Change
Q=D=1
Q=D=1
No Change
Q=D=0
Q=D=0
No Change
Q
D
CLK

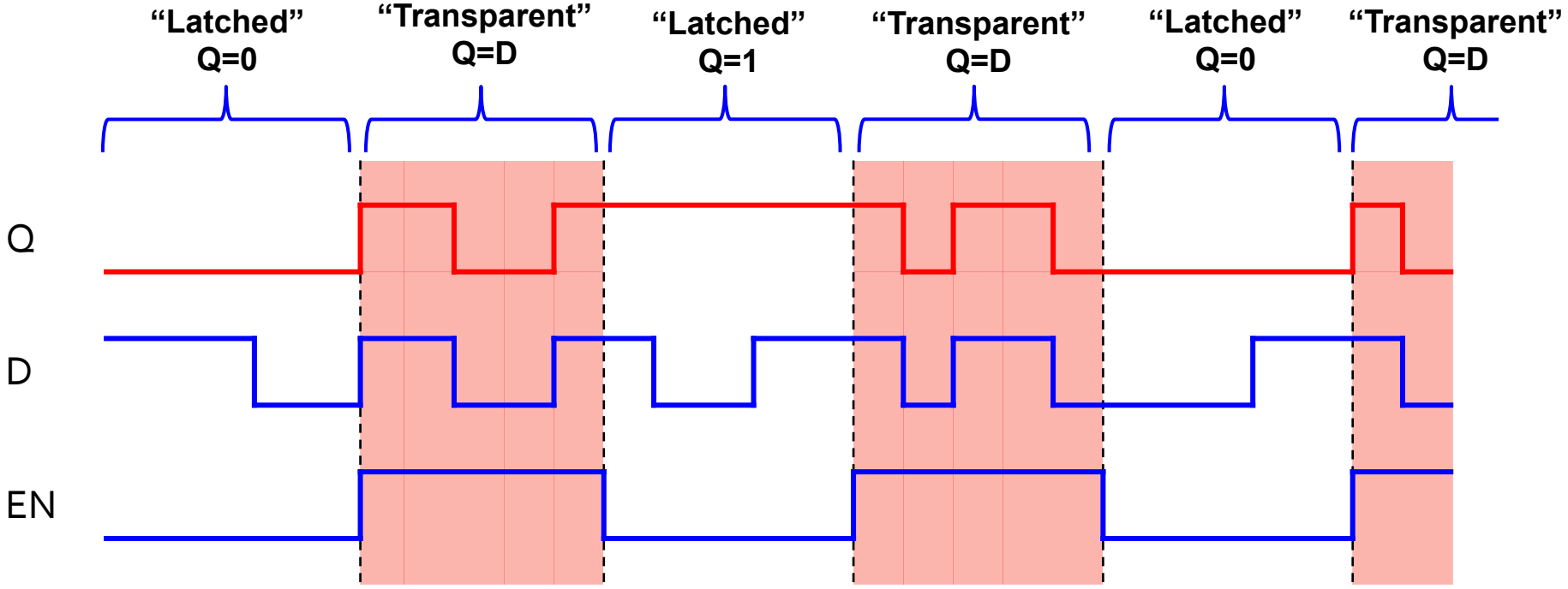
Latch D

- Dispositivo secuencial que responde a nivel

- `always @(enable, entrada)`
- `begin`
  - `if(enable == 1)`
    - `salida = entrada;`
- `end`

Latch D


LPM_LATCH_1
entrada
D
Q
salida
enable
Gate
salida


"Latched"
Q=0
"Transparent"
Q=D
"Latched"
Q=1
"Transparent"
Q=D
"Latched"
Q=0
"Transparent"
Q=D
Q
D
EN

Circuitos Secuenciales: Sistema Sincrónico

- La *Metodología de Diseño Sincrónico* es la técnica de diseño más utilizada para crear circuitos secuenciales:
  - Todos los elementos de memoria se sincronizan con una señal de reloj global;
  - Los datos se muestrean y guardan en el flanco positivo o negativo de la señal de reloj.
- Esta técnica permite separar los elementos de memoria del circuito simplificando el proceso de desarrollo.
- Este es el principio más importante para desarrollar un sistema digital complejo.

Sincronismo de Entradas

Entradas Asíncronas, Circuito Síncrono

Sincronizar Entradas

- `always @(posedge clock)`
- `begin`
  - `if(boton == 1)`
    - `salida <= entrada;`
- `end`

flip flop pero con una condición

Testbench Secuencial

```
module test_MUX2TO1; // No tiene puertos
   //DUT I/Os
   reg A, B, SEL, CLK;
   wire F;
   // DUT instantiation
    DUT my_mux(.A(A), .B(B), .SEL(SEL), .F(F));
   //Stimulus
   initial begin
      A = 0; B = 1; SEL = 0; CLK = 0;
      #20 SEL = 1;
   end

   always begin
      #(PERIOD/2) CLK = ~CLK;
   end
endmodule
```

instancio el bloque

En este caso el CLK no influye en mi módulo porque no esta como entrada

Ejercicio

- Sincronizar entradas y salidas del ejercicio combinacional de la clase anterior. Realizar Testbench