

A photograph of a misty forest path. The path is a narrow, winding trail covered in brown leaves and moss, leading into the distance. On the left, there are large, dark tree trunks. On the right, there is a dense thicket of green ferns and other foliage. The background is filled with tall, slender trees and a thick mist or fog, creating a serene and somewhat mysterious atmosphere.

Secuenciales (1)

Diseño de Sistemas con FPGA

Patricia Borensztein

Sistemas Secuenciales

- Es un circuito con memoria.
- La memoria forma el estado del circuito.
- Las salidas son función de las entradas y del estado interno.
- La metodología mas común de diseño es la síncrona: todos los elementos de memoria son controlados por un reloj global y los datos son muestreados y almacenados en el flanco ascendente o descendente de la señal del reloj.

Esta metodología permite separar los elementos de almacenamiento en un diseño grande, simplificando las tareas de diseño y testing.

Diseño Sistema Secuencial

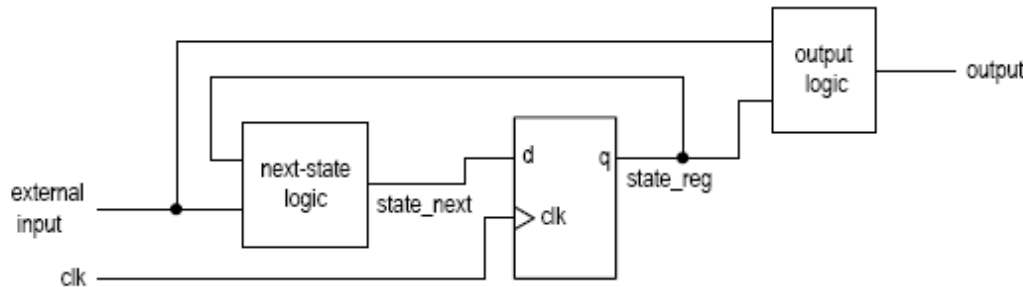
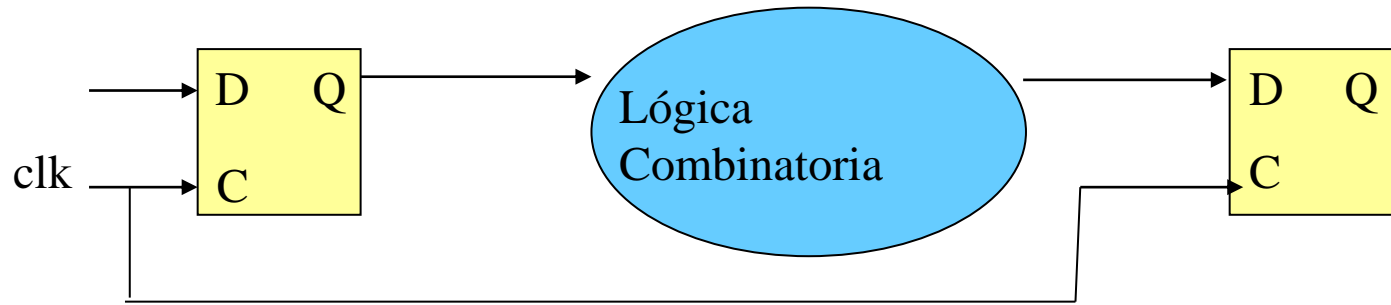


Figure 4.2 Block diagram of a synchronous system.

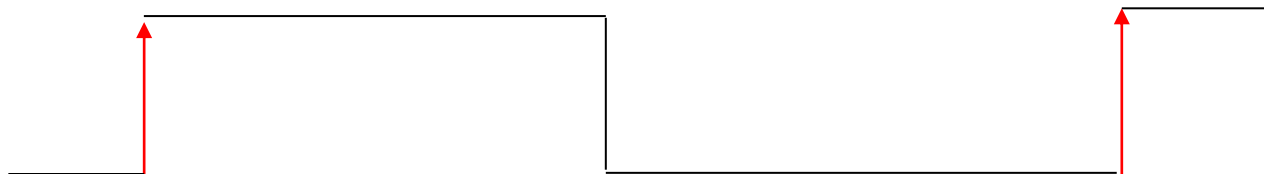
- Registro de Estado: colección de D FFs controlados por la misma señal
- Lógica de Siguiente Estado: lógica combinatorial que utiliza la entrada externa y el estado interno para determinar el nuevo valor del registro
- Lógica de Salida: lógica combinatorial que genera la señal de salida.
- Clave del diseño: separar la parte de memoria del resto del sistema.

Máxima Frecuencia de Operación

- El sistema secuencial está caracterizado por f_{\max} , que es la máxima frecuencia de reloj con la que se puede trabajar.
- Su inversa, T_{clk} , es el tiempo entre dos flancos de reloj.



$$T_{\text{clock}} > T_{\text{cq}} + T_{\text{comb}} + T_{\text{setup}}$$



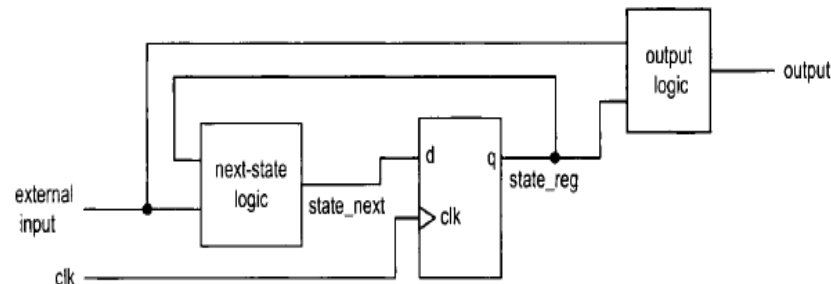
Máxima Frecuencia de Operación

- XST reporta fmax después de sintetizar el circuito.
- ISE permite que le especifiquemos la frecuencia de operación. Lo hacemos en el archivo de constraints (.ucf)
- Xilinx va a intentar satisfacer estos requerimientos y luego en el Design Summary podemos ver si fueron o no alcanzados.

```
NET "clk" TNM_NET = "clk";  
TIMESPEC "TS_clk" = PERIOD "clk" 20 ns HIGH 50 %;
```

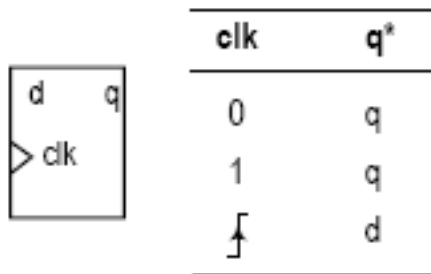

Desarrollo de Código

- La clave del diseño es separar los elementos de memoria.
- Los bloques de la lógica del siguiente estado y lógica de salida son combinacionales.
- Según las características de la lógica del siguiente estado, podemos caracterizar a los circuitos secuenciales en:
 1. Regulares: las transiciones entre estados exhiben un patrón regular que permite implementarse con componentes sencillos prediseñados (incrementador, por ej). Ejemplo: un contador mod n
 2. FSM: Máquina de Estados Finita. No hay patrón regular. Las transiciones entre estados siguen un orden “random”. Ejemplo: reconocedor de secuencia. Hay que sintetizarlas mediante un circuito específico.
 3. FSMD: Formado por un circuito regular (Data Path) y un FSM (control path). Usado para sintetizar algoritmos escritos en la metodología RT.

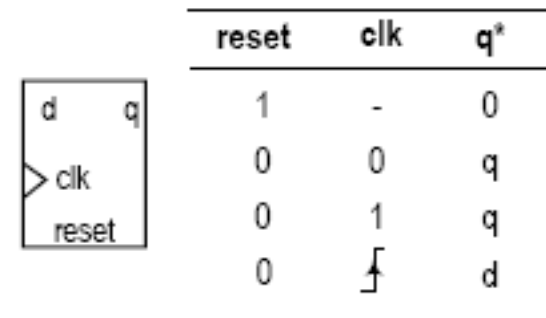


Flip-Flop D (D FF)

- Es el elemento de estado más elemental en un circuito secuencial.
- Funcionamiento: El valor de la señal d es muestreado en el flanco ascendente de la señal clk y almacenado en el FF.
- También puede tener una señal asíncrona “reset”
- Un D FF permite almacenar un bit. Una colección de D FF pueden agruparse para almacenar varios bits: esta colección se llama registro.



(a) D FF



(b) D FF with asynchronous reset

Códigos Verilog que infieren elementos de estado

- Se utiliza la estructura “always block”.
- Las asignaciones para generar elementos de estado deben ser “no bloqueantes”

```
[variable_name] <= [expression];
```

- Veamos las diferencias entre los tres tipos de asignaciones:
 - Bloqueantes, dentro de un always block (=)
 - No Bloqueantes, dentro de un always block (=>)
 - Continuas (ASSIGN)

Comportamiento de las asignaciones no bloqueantes

El always block es pensado como un hardware que se ejecutará en una unidad de tiempo. Al activarse el always block se evalúan todas las asignaciones no bloqueantes en paralelo, al finalizar el always block se asignan esos valores a todas las variables del lado izquierdo.

```
module and_nonblock
(
  input wire a, b, c,
  output reg y
5  );

  always @*
  begin
    y <= a;           // yentry = y
                      // yexit = a
10   y <= y & b;       // yexit = yentry & b
    y <= y & c;       // yexit = yentry & c
  end                // y = yexit

endmodule
```

Diferentes tipos de asignaciones generan diferentes circuitos

```
module and_block_assign
(
  input wire a, b, c,
  output reg y
5 );

  always @*
  begin
    y = a;
10    y = y & b;
    y = y & c;
  end

endmodule
```

(a)

bloqueante

No bloqueante

```
module and_nonblock
(
  input wire a, b, c,
  output reg y
5 );

  always @*
  begin
10    y <= a;
    y <= y & b;
    y <= y & c;
  end

endmodule
```

```
module and_cont_assign
(
  input wire a, b, c,
  output wire y
5 );

  assign y = a;
  assign y = y & b;
  assign y = y & c;
10

endmodule
```

©

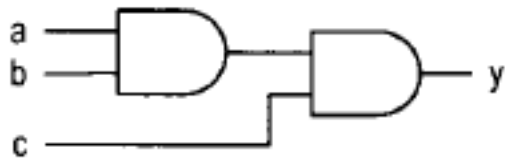
concurrente



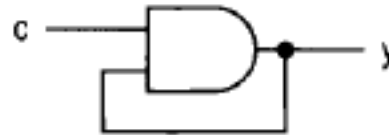
```
always @*
  y <= y & c;
```

(b)

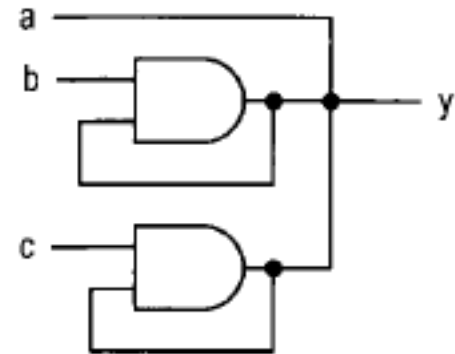
Diferentes tipos de asignaciones generan diferentes circuitos



(a)



(b)

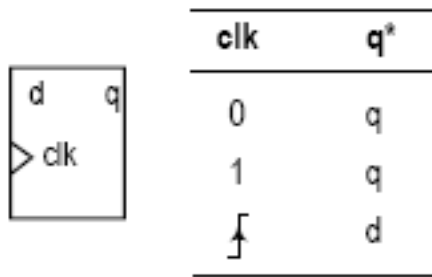


(c)

Para inferir elementos de memoria...

- Usaremos =>
- Las razones por las cuales esto es así tiene que ver con la semántica definida en el lenguaje Verilog. (VHDL no tiene esta complicación)
- Esto quiere decir que si usamos bloqueantes o bien mezclamos bloqueantes con no bloqueantes en un always block.... El resultado no está claro que sea el que deseamos.
- Por simplicidad, PARA INFERIR ELEMENTOS DE MEMORIA USAREMOS SIEMPRE ASIGNACIONES NO BLOQUEANTES.

Código Verilog para el Flip-Flop D



(a) D FF

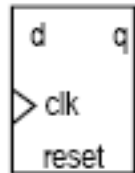
```
module d_ff
(
    input wire clk,
    input wire d,
    output reg q
);


// body
always @(posedge clk)
    q <= d;

endmodule
```

- La señal d no está en la lista de sensibilidad indicando que, aunque su valor cambie, el always block solo se activará en el flanco ascendente del clk.

Flip-Flop D con reset asíncrono



reset	clk	q*
1	-	0
0	0	q
0	1	q
0		d

(b) D FF with asynchronous reset

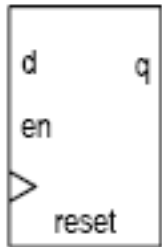
```
module d_ff_reset
(
    input wire clk, reset,
    input wire d,
    output reg q
5   );

    // body
    always @(posedge clk, posedge reset)
10    if (reset)
        q <= 1'b0;
    else
        q <= d;

15 endmodule
```

- La señal reset es asíncrona y tiene mayor prioridad que el flanco del reloj.

Flip-Flop D con reset asíncrono y enable síncrono



reset	clk	en	q*
1	-	-	0
0	0	-	q
0	1	-	q
0	⌋	0	q
0	⌋	1	d

(c) D FF with synchronous enable

```

module d_ff_en_1seg
(
    input wire clk, reset,
    input wire en,
    input wire d,
    output reg q
);

// body
10 always @(posedge clk, posedge reset)
    if (reset)
        q <= 1'b0;
    else if (en)
        q <= d;

15 endmodule

```

- Cuando la señal *en* (síncrona) no está activa, q no cambia su valor sino que mantiene el que tenía.
- Cada LC en Spartan tiene un DFF con reset asíncrono y enable síncrono

Flip-Flop D con reset asíncrono y enable síncrono

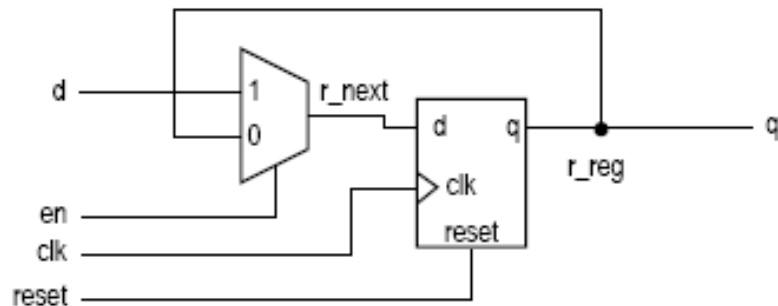


Figure 4.3 D FF with synchronous enable.

```

module d_ff_en_2seg
(
    input wire clk, reset,
    input wire en,
    input wire d,
    output reg q
);

// signal declaration
reg r_reg, r_next;

// body
// D FF
always @(posedge clk, posedge reset)
    if (reset)
        r_reg <= 1'b0;
    else
        r_reg <= r_next;

// next-state logic
always @*
    if (en)
        r_next = d;
    else
        r_next = r_reg;

// output logic
always @*
    q = r_reg;

endmodule

```

ISE: Language Templates

FDCPE: Single Data Rate D Flip-Flop with Asynchronous Clear, Set and Clock Enable (negedge clock).

```
// FDCPE_1: Single Data Rate D Flip-Flop with Asynchronous Clear, Set and
//      Clock Enable (negedge clock).
//      All families.
// Xilinx HDL Language Template, version 11.1

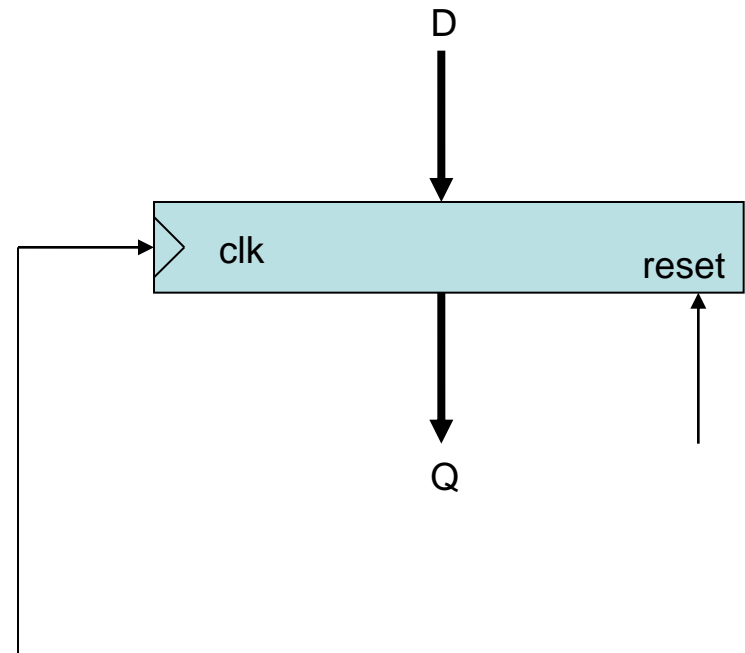
FDCPE_1 #(
    .INIT(1'b0) // Initial value of register (1'b0 or 1'b1)
) FDCPE_1_inst (
    .Q(Q),      // Data output
    .C(C),      // Clock input
    .CE(CE),    // Clock enable input
    .CLR(CLR),  // Asynchronous clear input
    .D(D),      // Data input
    .PRE(PRE)   // Asynchronous set input
);
```

Registros

```
// Listing 4.5
module reg_reset
(
    input wire clk, reset,
    input wire [7:0] d,
    output reg [7:0] q
);

// body
always @(posedge clk, posedge reset)
    if (reset)
        q <= 0;
    else
        q <= d;

endmodule
```



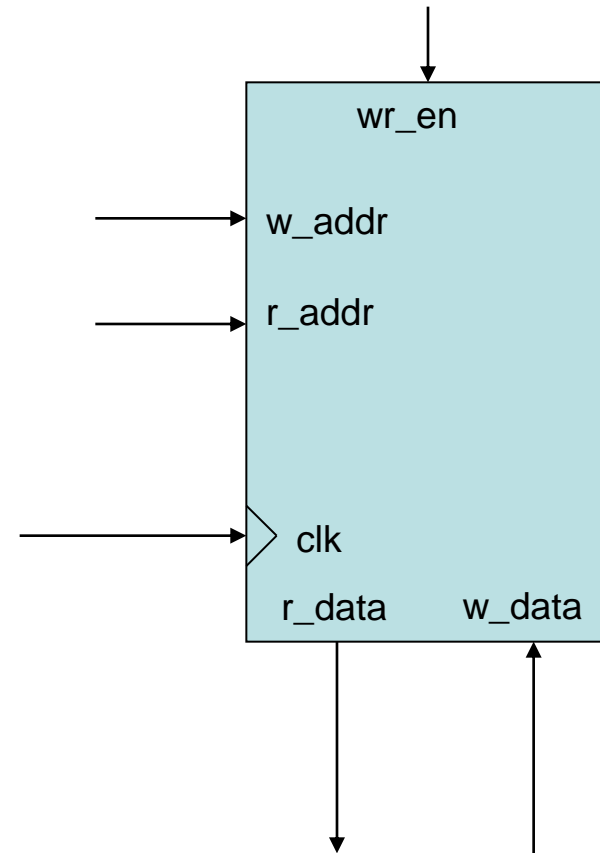
Banco de Registros

```
module reg_file
#(
    parameter B = 8, // number of bits
           W = 2 // number of address bits
)
(
    input wire clk,
    input wire wr_en,
    input wire [W-1:0] w_addr, r_addr,
    input wire [B-1:0] w_data,
    output wire [B-1:0] r_data
);

// signal declaration
reg [B-1:0] array_reg [2**W-1:0];

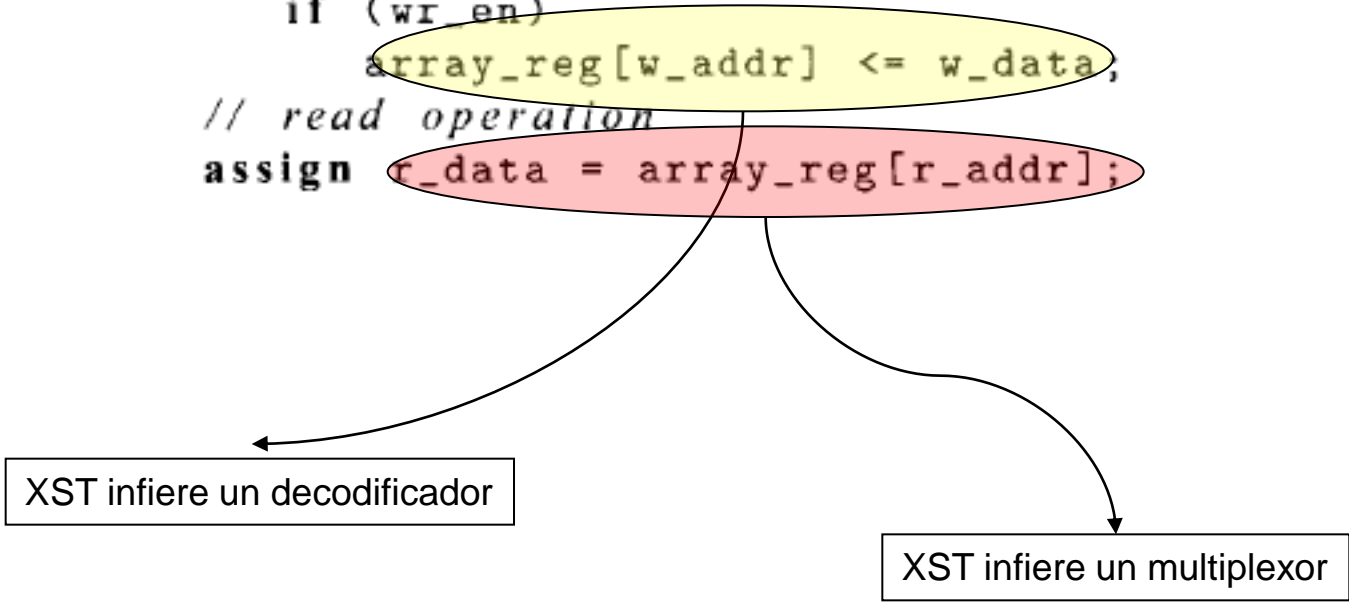
// body
// write operation
always @(posedge clk)
    if (wr_en)
        array_reg[w_addr] <= w_data;
// read operation
assign r_data = array_reg[r_addr];

endmodule
```



Banco de Registros

```
// body
// write operation
always @(posedge clk)
    if (wr_en)
        array_reg[w_addr] <= w_data;
// read operation
assign r_data = array_reg[r_addr];
```



XST infiere un decodificador

XST infiere un multiplexor

!!! En realidad, esto infiere una RAM!!!!

Instanciación: 16 (w=4) registros de 2 bits (B=2)

```
// RAM16X2S: 16 x 2 posedge write distributed (LUT) RAM
//      Spartan-3E
// Xilinx HDL Language Template, version 13.2

RAM16X2S #(
    .INIT_00(16'h0000), // Initial contents of bit 0 of RAM
    .INIT_01(16'h0000) // Initial contents of bit 1 of RAM
) RAM16X2S_inst (
    .O0(O0),    // RAM data[0] output
    .O1(O1),    // RAM data[1] output
    .A0(A0),    // RAM address[0] input
    .A1(A1),    // RAM address[1] input
    .A2(A2),    // RAM address[2] input
    .A3(A3),    // RAM address[3] input
    .D0(D0),    // RAM data[0] input
    .D1(D1),    // RAM data[1] input
    .WCLK(WCLK), // Write clock input
    .WE(WE)     // Write enable input
);

// End of RAM16X2S_inst instantiation
```

Shift Register

```
module free_run_shift_reg
  #(parameter N=8)
  (
    input wire clk, reset,
    input wire s_in,
    output wire s_out
  );

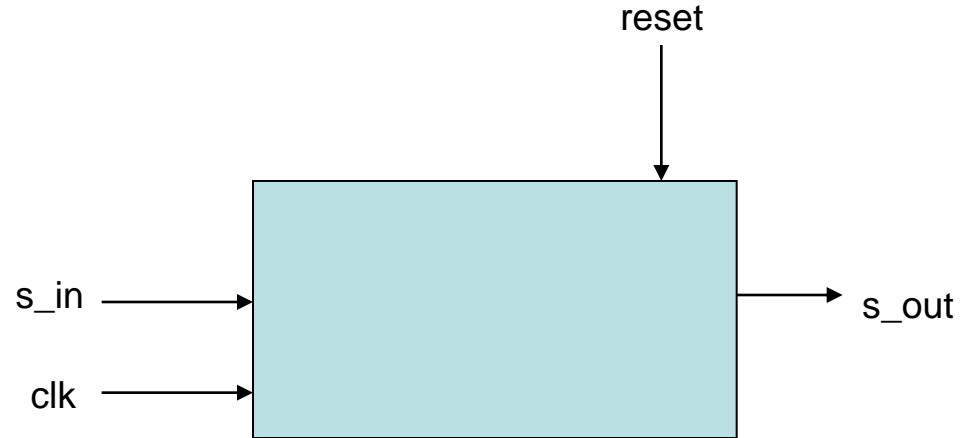
  //signal declaration
  reg [N-1:0] r_reg;
  wire [N-1:0] r_next;

  // body
  // register
  always @(posedge clk, posedge reset)
    if (reset)
      r_reg <= 0;
    else
      r_reg <= r_next;

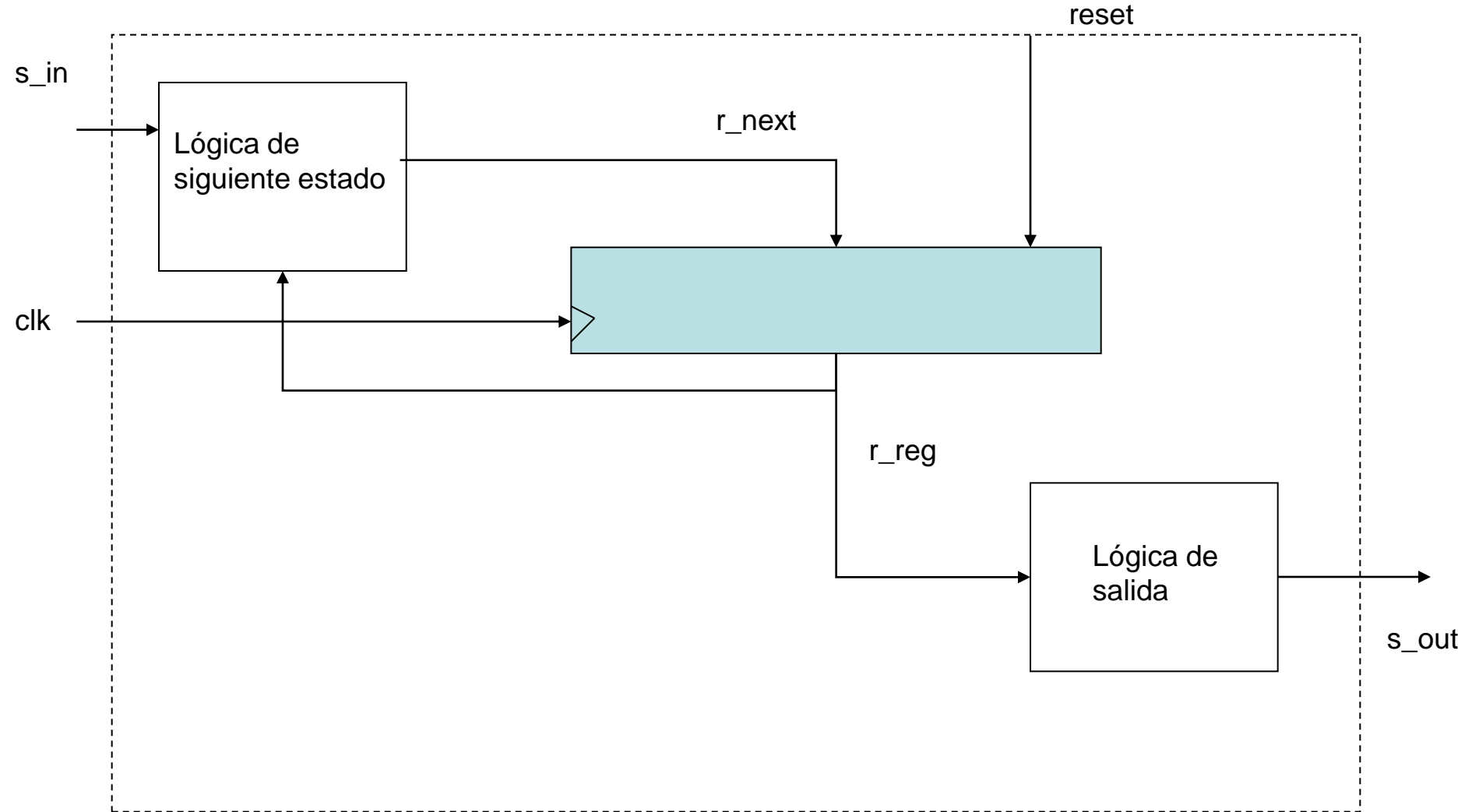
  // next-state logic
  assign r_next = {s_in, r_reg[N-1:1]};
  // output logic
  assign s_out = r_reg[0];

endmodule
```

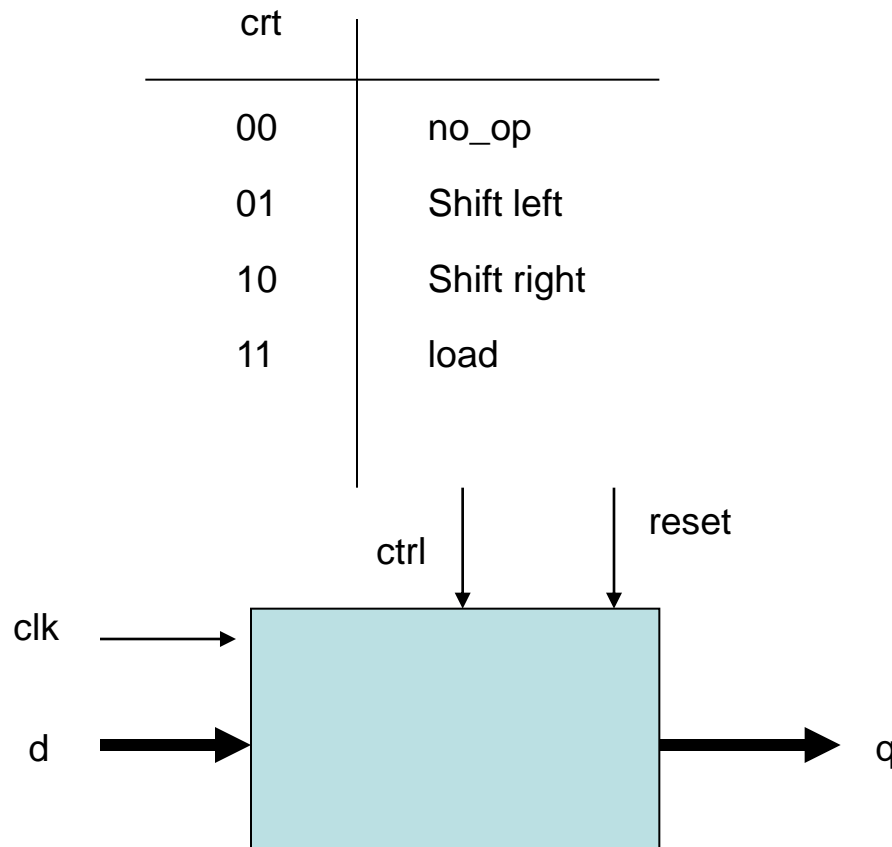
- En cada ciclo, desplaza hacia derecha (o izquierda) una posición.



Shift Register



Ejemplo: Universal Shift Register



- Carga Paralela de la entrada d.
- Shift de una posición a la izquierda. El bit mas bajo se carga con el bit mas bajo de la entrada d.
- Shift de una posición a la derecha. El bit mas alto se carga con el bit mas alto de la entrada d.

Ejemplo: Universal Shift Register

```

module univ_shift_reg
  #(parameter N=8)
  (
    input wire clk, reset,
    input wire [1:0] ctrl,
    input wire [N-1:0] d,
    output wire [N-1:0] q
  );

```

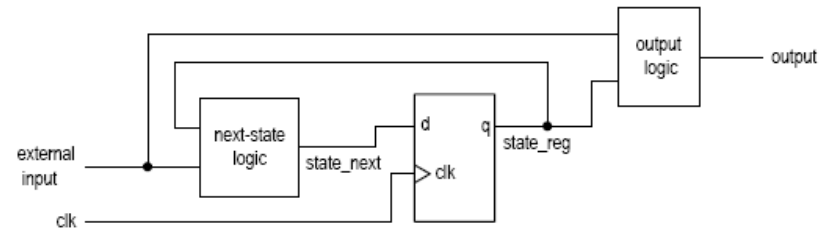
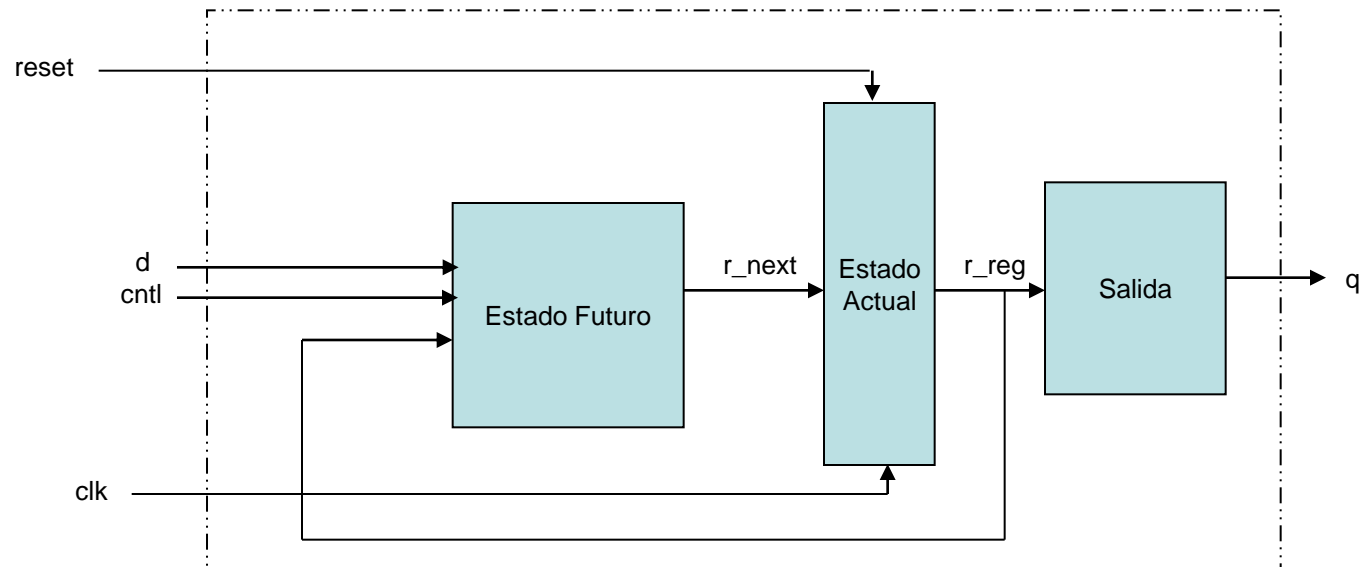


Figure 4.2 Block diagram of a synchronous system.

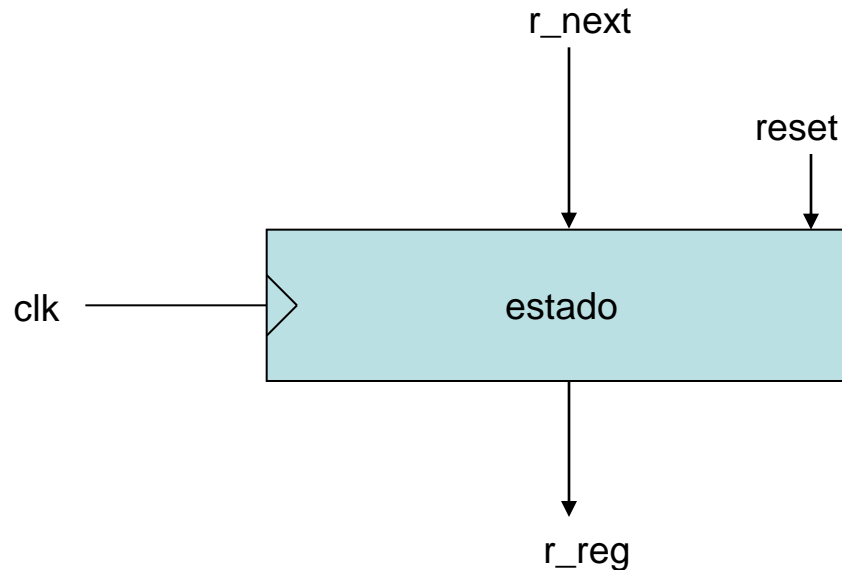


Universal Shift Register: Estado Futuro

```
// next-state logic
always @*
  case(ctrl)
    2'b00: r_next = r_reg;           // no op
    2'b01: r_next = {r_reg[N-2:0], d[0]}; // shift left
    2'b10: r_next = {d[N-1], r_reg[N-1:1]}; // shift right
    default: r_next = d;           // load
  endcase
```

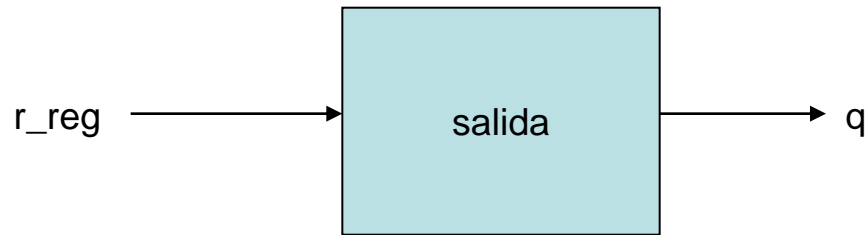
00	nada
01	shift left
10	shift right
11	carga paralela

Universal Shift Register: elemento de estado



```
// register  
always @(posedge clk, posedge reset)  
    if (reset)  
        r_reg <= 0;  
    else  
        r_reg <= r_next;
```

Ejemplo: Universal Shift Register



```
// output logic  
assign q = r_reg;
```

Ejemplo: Universal Shift Register

```
module univ_shift_reg
  #(parameter N=8)
  (
    input wire clk, reset,
    input wire [1:0] ctrl,
    input wire [N-1:0] d,
    output wire [N-1:0] q
  );

  //signal declaration
  reg [N-1:0] r_reg, r_next;

  // body
  // register
  always @(posedge clk, posedge reset)
    if (reset)
      r_reg <= 0;
    else
      r_reg <= r_next;
```

```
// next-state logic
always @*
  case(ctrl)
    2'b00: r_next = r_reg;           // no op
    2'b01: r_next = {r_reg[N-2:0], d[0]}; // shift left
    2'b10: r_next = {d[N-1], r_reg[N-1:1]}; // shift right
    default: r_next = d;           // load
  endcase

// output logic
  assign q = r_reg;

endmodule
```


Registro implementado usando LUT (xilinx)

- Recordamos que una LUT de 4 entradas se implementa mediante 16 celdas de SRAM de 1 bit.
- Xilinx permite configurar la LUT como un registro de desplazamiento.
- La restricción es:
 - No hay set ni reset
 - La carga es en serie
 - La salida es en serie

For this example, XST infers an SRL16.

```
//  
// 8-bit Shift-Left Register with Positive-Edge Clock,  
// Serial In, and Serial Out  
//  
  
module v_shift_registers_1 (C, SI, SO);  
    input C,SI;  
    output SO;  
    reg [7:0] tmp;  
  
    always @(posedge C)  
    begin  
        tmp <= tmp << 1;  
        tmp[0] <= SI;  
    end  
  
    assign SO = tmp[7];  
  
endmodule
```

```

module free_run_bin_counter
  #(parameter N=8)
  (
    input wire clk, reset,
    output wire max_tick,
    output wire [N-1:0] q
  );

  //signal declaration
  reg [N-1:0] r_reg;
  wire [N-1:0] r_next;

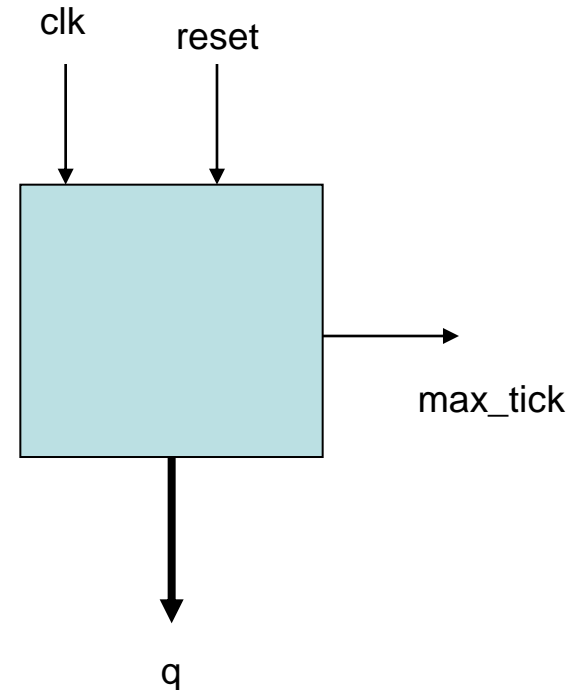
  // body
  // register
  always @(posedge clk, posedge reset)
    if (reset)
      r_reg <= 0; // {N{1b'0}}
    else
      r_reg <= r_next;

  // next-state logic
  assign r_next = r_reg + 1;
  // output logic
  assign q = r_reg;
  assign max_tick = (r_reg==2**N-1) ? 1'b1 : 1'b0;
  //can also use (r_reg=={N{1'b1}})

endmodule

```

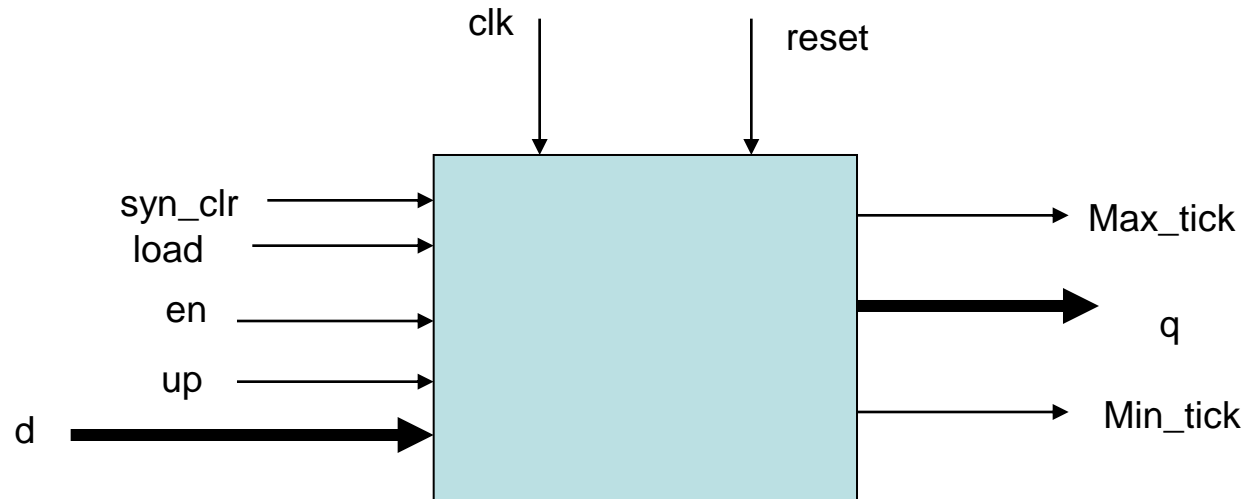
Contador Binario



Contador Binario Universal

Table 4.1 Function table of a universal binary counter

syn_clr	load	en	up	q*	Operation
1	—	—	—	00...00	synchronous clear
0	1	—	—	d	parallel load
0	0	1	1	q+1	count up
0	0	1	0	q-1	count down
0	0	0	—	q	pause



Testbench para el circuito contador universal

// Listing 4.12

```
`timescale 1 ns/10 ps
```

```
// The `timescale directive specifies that  
// the simulation time unit is 1 ns and  
// the simulator timestep is 10 ps
```

```
module bin_counter_tb();
```

```
// declaration
```

```
localparam T=20; // clock period
```

```
reg clk, reset;
```

```
reg syn_clr, load, en, up;
```

```
reg [2:0] d;
```

```
wire max_tick, min_tick;
```

```
wire [2:0] q;
```

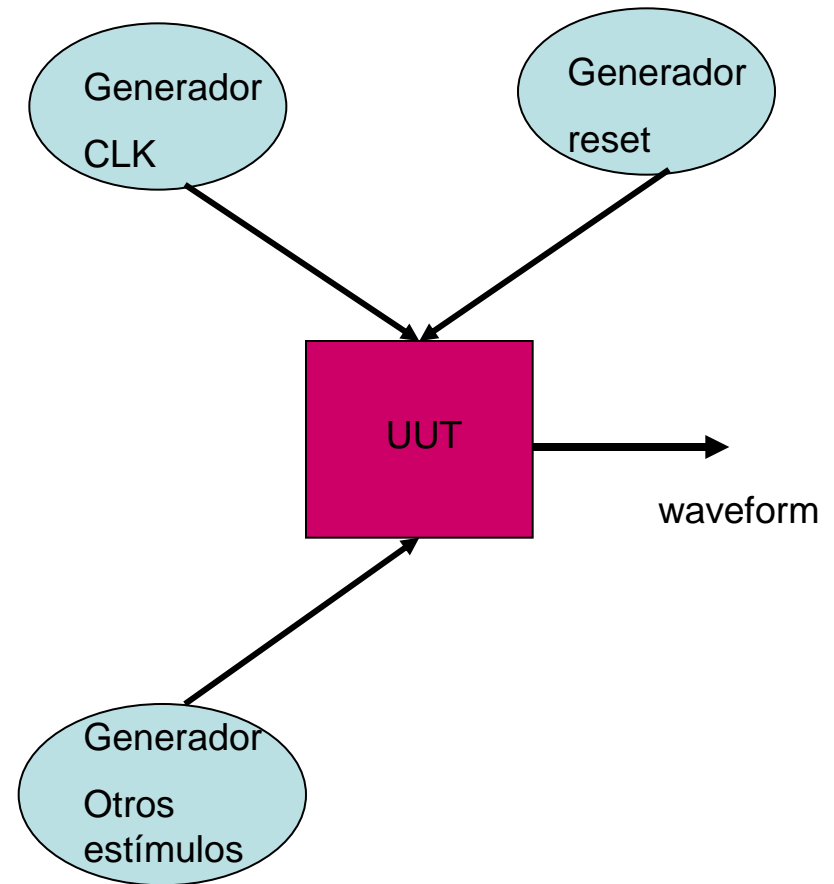
```
// uut instantiation
```

```
univ_bin_counter #(N(3)) uut
```

```
(.clk(clk), .reset(reset), .syn_clr(syn_clr),
```

```
.load(load), .en(en), .up(up), .d(d),
```

```
.max_tick(max_tick), .min_tick(min_tick), .q(q));
```



Testbench para el circuito contador universal

```
// clock  
// 20 ns clock running forever
```

```
always  
begin  
  clk = 1'b1;  
  #(T/2);  
  clk = 1'b0;  
  #(T/2);  
end
```

```
// reset for the first half cycle
```

```
initial  
begin  
  reset = 1'b1;  
  #(T/2);  
  reset = 1'b0;  
end
```

- El generador de reloj se realiza con un always block sin lista de sensibilidad... se repite por siempre.

- El generador de reset se realiza con un initial block que se ejecuta una única vez, al principio de la simulación.

Testbench para el circuito contador universal

```
// other stimulus
initial
begin
    // ===== initial input =====
    syn_clr = 1'b0;
    load = 1'b0;
    en = 1'b0;
    up = 1'b1; // count up
    d = 3'b000;
    @(negedge reset); // wait reset to deassert
    @(negedge clk); // wait for one clock
    // ===== test load =====
    load = 1'b1;
    d = 3'b011;
    @(negedge clk); // wait for one clock
    load = 1'b0;
    repeat(2) @(negedge clk);
    // ===== test syn_clear =====
    syn_clr = 1'b1; // assert clear
    @(negedge clk);
    syn_clr = 1'b0;
```

- En un circuito síncrono por flanco ascendente, las entradas deben estar estables cuando el flanco del reloj asciende. Para garantizar esto, podemos esperar al flanco descendente para cambiar las entradas. Esto se hace con la sentencia: `@(negedge clk)`.

- Cada sentencia `@(negedge clk)` representa un nuevo flanco descendente. Si queremos esperar varios ciclos de reloj podemos usar: `repeat(2) @(negedge clk);`

Testbench para el circuito contador universal

```
// ==== test up counter and pause ====
en = 1'b1; // count
up = 1'b1;
repeat(10) @(negedge clk);
en = 1'b0; // pause
repeat(2) @(negedge clk);
en = 1'b1;
repeat(2) @(negedge clk);
// ==== test down counter ====
up = 1'b0;
repeat(10) @(negedge clk);
// ==== wait statement ====
// continue until q=2
wait(q==2);
@(negedge clk);
up = 1'b1;
// continue until min_tick becomes 1
@(negedge clk);
wait(min_tick);
@(negedge clk);
up = 1'b0;
// ==== absolute delay ====
#(4*T); // wait for 80 ns
en = 1'b0; // pause
#(4*T); // wait for 80 ns
// ==== stop simulation ====
// return to interactive simulation mode
$stop;
end
endmodule
```

- También podemos esperar un valor específico para una señal: `wait(q==2);` o bien el cambio de una señal : `wait(min_tick);` A continuación, para asegurarnos que el cambio posterior de una entrada no ocurre en el flanco ascendente del reloj, esperamos un ciclo.

- También podemos esperar un tiempo absoluto medido en unidades del periodo del reloj.

- La sentencia : `$stop;` detiene la simulación

Ejercicio Secuenciales

- Construir un sistema usando el módulo contador universal, que permita visualizar el resultado del contador en los Leds
- Hacer que el contador funcione a una frecuencia suficientemente baja como para que lo podamos verlo.