



Instituto Politécnico Nacional
Escuela Superior de Cómputo



Arquitectura de Computadoras

“Testbench para Contador”

Alumno:

Malagón Baeza Alan Adrian

Profesor:

Alemán Arce Miguel Ángel

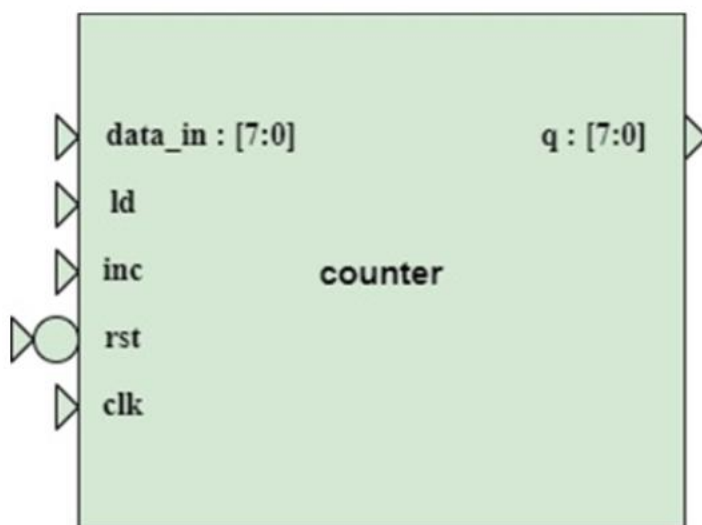
Grupo: 5CV1

Introducción

En esta práctica de laboratorio, se entrega un dispositivo y una especificación. Su trabajo es asegurarse de que el dispositivo coincida con la especificación utilizando un Testbench. El el Testbench se ha montado para usted. Solo se necesita proporcionar la funcionalidad de prueba.

El dispositivo bajo prueba (Device Under Test, DUT)

El dispositivo bajo prueba (DUT) es un simple contador de 8 bits. Tiene los siguientes símbolos y comportamiento de señal:



clk – El reloj contador.

rst: un reinicio síncrono.

data_in[7:0]: un bus de entrada de 8 bits.

q[7:0] – Un bus de salida de 8 bits.

inc – q pasa a 1 q+1 en el siguiente ciclo de reloj si inc es 1. Si q es 8'hFF, vuelve a 8'h00.

ld – q se establece igual a data_in si ld es 1. ld tiene prioridad sobre inc.

Tarea de laboratorio

1. Cree un banco de pruebas que demuestre que este diseño funciona como se esperaba. Debe poder demostrar que el diseño funciona a un tercero.

2. El banco de pruebas ya se ha conectado y hay un script de compilación en su lugar. Hay tres archivos en el directorio de laboratorio:

a) run.do: este script compila el DUT y el banco de pruebas.

b) DUT.sv: este archivo SystemVerilog contiene el dispositivo bajo prueba. Estamos tratando este archivo como una caja negra, por lo que no es legible por humanos.

c) top.sv – Este es el banco de pruebas. Instancia el DUT.

Crearás tu banco de pruebas en top.sv. El archivo top.sv se ve así:

```
module top;
  logic [7:0] data_in;
  bit        clk, inc, ld, rst;
  wire [7:0] q;

  counter DUT(.data_in(data_in), .q(q), .clk(clk), .inc(inc), .ld(ld), .rst(rst));

endmodule // top
```

El DUT se instancia en el archivo top y puede asignar valores a las señales clk, inc, ld, rst y data_in utilizando cualquier combinación de bloques always e initial que pondrán a prueba el diseño.

Este diseño se puede compilar y simular en ModelSim o Questa usando el script run.do. Utilice este banco de pruebas para demostrar que el diseño funciona como se describe.

Desarrollo

Código SystemVerilog

```
C:/Users/alanm/Documents/GitHub/ArquitecturaDeComputadoras/ModelSim/counter/DUT.sv (/top/DUT) - Default
Ln#
1  module counter (
2      input wire [7:0] data_in,
3      output reg [7:0] q,
4      input wire clk,
5      input wire inc,
6      input wire ld,
7      input wire rst);
8  `protected
9
10     MII!#j='0;. {l25c=Qm=p?_|^73!O>$+!GRi]3!{N9?{'$~si[aa7e=<]sG)#To]QO[IszB~rlo
11     7MZY_k2}5!F!|ax[B37zr[O,v{IusQCJT{B;[Ok_H<ITT!e2E<r5\u'O<]^AaML-5\-<{KnH~{}}?
12     _9=3^>$+vO,=r_+<>[P=QI]PjeeT2<)Wv3@{'^bdp]vn3a]-
13 `endprotected
14 endmodule // counter
```

Código SystemVerilog Testbench (Banco de pruebas)

```
C:/Users/alanm/Documents/GitHub/ArquitecturaDeComputadoras/ModelSim/counter/top.sv - Default *
Ln#
1  module top;
2      logic [7:0] data_in;
3      bit        clk, inc, ld, rst;
4      wire [7:0] q;
5      logic [7:0] q_beh;
6
7
8      counter DUT(.data_in(data_in),.q(q),.clk(clk),.inc(inc),.ld(ld),.rst(rst));
9
10     always #10 clk = ~clk;
11
12     initial begin : reset_block
13         clk = 0;
14         rst = 0;
15         @(posedge clk);
16         @(negedge clk);
17         rst = 1;
18         data_in = $urandom;
19         ld = 1'b1;
20         inc = 1'b0;
21         @(negedge clk);
22         @(negedge clk);
23     repeat(50) begin : do_mult_stim
24         {ld, inc} = $urandom;
25         data_in = $random;
26         @(negedge clk);
27         @(negedge clk);
28     end
29     $stop;
30 end
31
32     always @(posedge clk) begin: beh_cntr;
33         if (!rst)
34             q_beh <= 0;
35         else
36             if (ld)
37                 q_beh <= data_in;
38             else if (inc)
39                 q_beh++;
40         end
41
42     always @(negedge clk) begin : test_dut
43         assert(q == q_beh);
44     end
45
46 endmodule // top
47
```

El código anterior es una descripción de un módulo de nivel superior llamado "top" en lenguaje de descripción de hardware (HDL). Este módulo actúa como un entorno de prueba para el módulo de contador llamado "DUT" que se instancia en él. El módulo "top" también contiene lógica adicional para generar señales de reloj, controlar la entrada del contador y verificar la salida del contador.

A continuación, se explica el código por secciones:

```
module top;
    logic [7:0] data_in;
    bit        clk, inc, ld, rst;
    wire [7:0] q;
    logic [7:0] q_beh;
```

En esta sección se definen las señales que se utilizan en el módulo "top". data_in es una señal de 8 bits utilizada para cargar datos en el contador. clk es una señal de reloj que se utiliza para sincronizar las operaciones del contador. inc y ld son señales de control que se utilizan para incrementar el contador y cargar datos en el contador, respectivamente. rst es una señal de reinicio que se utiliza para restablecer el contador a un valor inicial. q es una señal de salida que representa el valor actual del contador. q_beh es una señal interna utilizada para simular el comportamiento esperado del contador.

```
counter DUT(.data_in(data_in),.q(q),.clk(clk),.inc(inc),.ld(ld),.rst(rst));
```

En esta línea se instancia el módulo de contador llamado "DUT" y se conectan las señales del módulo "top" con las señales del módulo "DUT".

```
always #10 clk = ~clk;
```

Esta línea define un bucle siempre activo que cambia el valor de la señal clk cada 10 unidades de tiempo. Esto genera una señal de reloj con un período de 20 unidades de tiempo (10 unidades de tiempo en nivel alto y 10 unidades de tiempo en nivel bajo).

```
initial begin : reset_block
    clk = 0;
    rst = 0;
    @(posedge clk);
    @(negedge clk);
    rst = 1;
    // ...
end
```

En este bloque inicial se establece el valor inicial de clk y rst en 0. Luego, se espera por un flanco de subida (posedge clk) y un flanco de bajada (negedge clk) de la señal clk. Después, rst se establece en 1 para realizar un reinicio del contador. A continuación, se genera un valor aleatorio para data_in, se establece ld en 1 y inc en 0. Luego, se esperan dos flancos de bajada de clk. Después de eso, se repite un bloque de estimulación (do_mult_stim) 50 veces, donde se generan valores aleatorios para ld, inc y data_in, y se esperan dos flancos de bajada de clk. Finalmente, se utiliza \$stop para finalizar la simulación.

```
always @(posedge clk) begin: beh_cntr;
    if (!rst)
        q_beh <= 0;
    else if (ld)
        q_beh <= data_in;
    else if (inc)
        q_beh++;
end
```

Este bloque siempre activo (always @(posedge clk)) se encarga de simular el comportamiento esperado del contador. Si rst es igual a 0, se establece q_beh en 0. Si ld es igual a 1, se carga el valor de data_in en q_beh. Si inc es igual a 1, se incrementa el valor de q_beh en 1.

```
always @(negedge clk) begin : test_dut
    assert(q == q_beh);
end
```

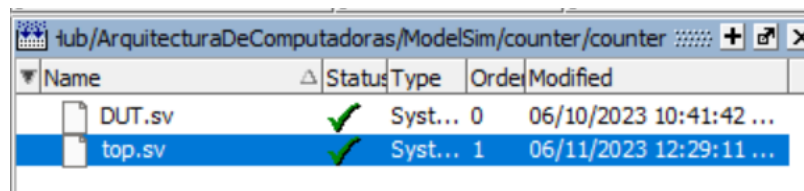
Este bloque siempre activo (always @(negedge clk)) se encarga de verificar la salida del contador q con respecto al valor simulado q_beh. La declaración assert verifica si q es igual a q_beh y genera una advertencia si la afirmación es falsa.

```
endmodule // top
```

Esta línea indica el final del módulo "top".

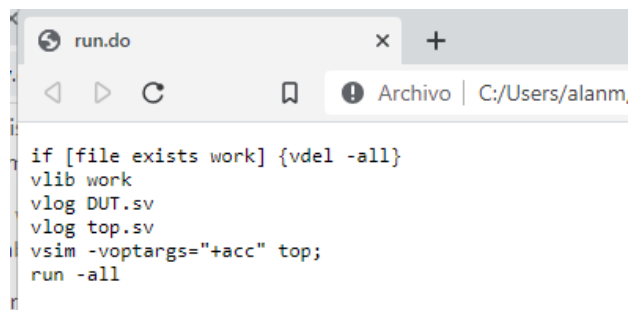
En resumen, el código proporciona un entorno de prueba para el módulo de contador "DUT". Genera señales de reloj, controla la entrada del contador y verifica la salida del contador utilizando una señal simulada q_beh. Este código puede ser utilizado para simular y verificar el comportamiento del módulo de contador en un entorno de simulación.

Programas compilados



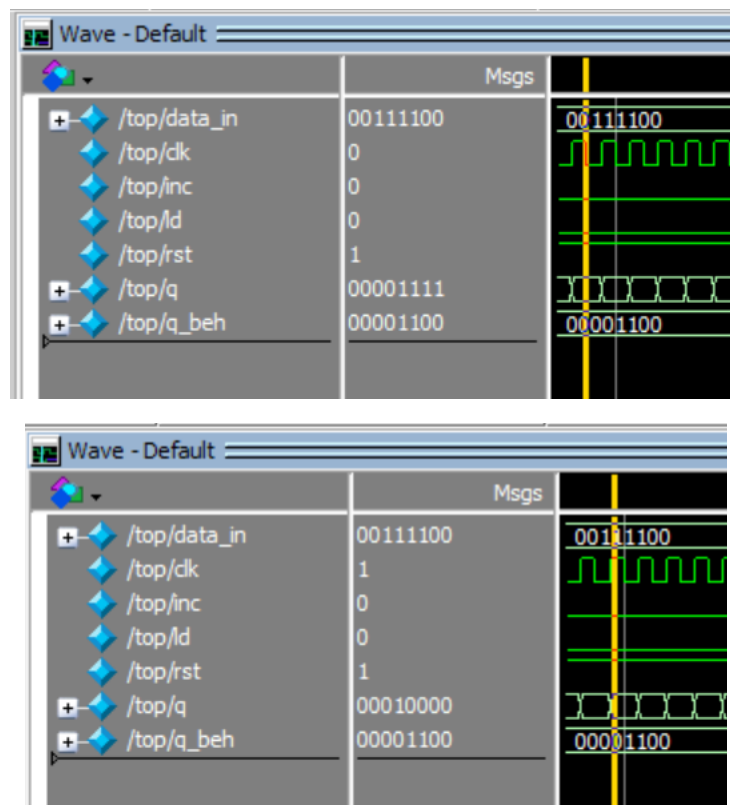
Name	Status	Type	Order	Modified
DUT.sv	✓	Syst...	0	06/10/2023 10:41:42 ...
top.sv	✓	Syst...	1	06/11/2023 12:29:11 ...

Archivo run.do



```
run.do
if [file exists work] {vdel -all}
vlib work
vlog DUT.sv
vlog top.sv
vsim -voptargs="+acc" top;
run -all
```

Resultado de la simulación:



$$Q1 = 1111 + 1$$

$$Q2 = 10000 \checkmark \text{ Incremento en 1}$$

Resultado consola:

```
VSIM 2> run
# ** Error: Assertion error.
#   Time: 2080 ps  Scope: top.test_dut File: top.sv Line: 43
# ** Error: Assertion error.
#   Time: 2100 ps  Scope: top.test_dut File: top.sv Line: 43
# ** Error: Assertion error.
#   Time: 2120 ps  Scope: top.test_dut File: top.sv Line: 43
# ** Error: Assertion error.
#   Time: 2140 ps  Scope: top.test_dut File: top.sv Line: 43
# ** Error: Assertion error.
#   Time: 2160 ps  Scope: top.test_dut File: top.sv Line: 43
```

Manda error porque $q \neq q_beh$

Modificación al programa

Se agregó un monitor para observar mejor de donde proviene el error.

```
47 initial begin
48     $monitor("clk=%0d rst=%0d ld=%0d inc=%0d data_in=%0h q=%0h q_beh=%0h", clk, rst, ld, inc, data_in, q, q_beh);
49 end
50

# clk=1 rst=0 ld=0 inc=0 data_in=x q=x q_beh=x
# clk=1 rst=1 ld=1 inc=0 data_in=24 q=0 q_beh=0
# clk=1 rst=1 ld=1 inc=0 data_in=24 q=24 q_beh=24
# clk=1 rst=1 ld=0 inc=1 data_in=9 q=24 q_beh=25
# clk=1 rst=1 ld=0 inc=1 data_in=9 q=25 q_beh=26
# clk=1 rst=1 ld=1 inc=1 data_in=d q=26 q_beh=26
# clk=1 rst=1 ld=1 inc=1 data_in=d q=d q_beh=d
# clk=1 rst=1 ld=0 inc=1 data_in=65 q=d q_beh=e
# clk=1 rst=1 ld=0 inc=1 data_in=65 q=e q_beh=f
# clk=1 rst=1 ld=1 inc=0 data_in=1 q=f q_beh=f
# clk=1 rst=1 ld=1 inc=0 data_in=1 q=1 q_beh=1
# clk=1 rst=1 ld=0 inc=1 data_in=76 q=1 q_beh=2
# clk=1 rst=1 ld=0 inc=1 data_in=76 q=2 q_beh=3
# clk=1 rst=1 ld=0 inc=1 data_in=ed q=3 q_beh=4
# clk=1 rst=1 ld=0 inc=1 data_in=ed q=4 q_beh=5
# clk=1 rst=1 ld=0 inc=0 data_in=f9 q=5 q_beh=5
# ** Error: Assertion error.
#   Time: 320 ps  Scope: top.test_dut File: top.sv Line: 43
# clk=1 rst=1 ld=0 inc=0 data_in=f9 q=6 q_beh=5
# ** Error: Assertion error.
#   Time: 340 ps  Scope: top.test_dut File: top.sv Line: 43
# clk=1 rst=1 ld=1 inc=0 data_in=c5 q=7 q_beh=5
# clk=1 rst=1 ld=1 inc=0 data_in=c5 q=c5 q_beh=c5
```

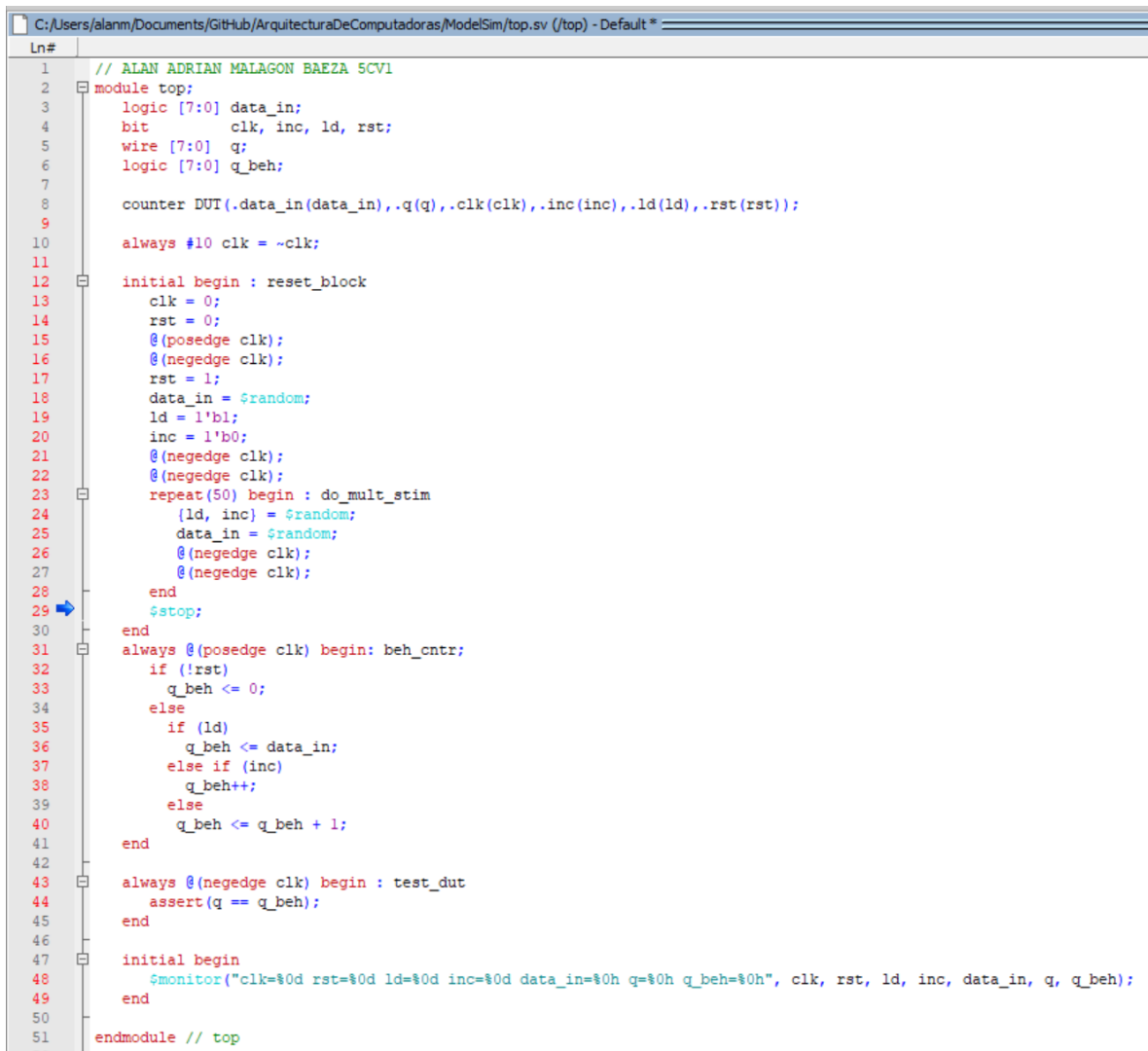
Al observar los resultados con el monitor se puede concluir que $q \neq q_beh$ cuando ld e inc son iguales a 0, ya que q sigue aumentando pero q_beh no. Para arreglar esto se modificó el código agregando un `else` para incrementar q_beh cuando $ld=inc=0$.

```

always @(posedge clk) begin: beh_cntr;
    if (!rst)
        q_beh <= 0;
    else
        if (ld)
            q_beh <= data_in;
        else if (inc)
            q_beh++;
        else
            q_beh <= q_beh + 1;
    end

```

Código SystemVerilog TestBench (Banco de pruebas) Modificado:



```

C:/Users/alanm/Documents/GitHub/ArquitecturaDeComputadoras/ModelSim/top.sv (/top) - Default *
Ln#
1 // ALAN ADRIAN MALAGON BAEZA SCV1
2 module top;
3     logic [7:0] data_in;
4     bit clk, inc, ld, rst;
5     wire [7:0] q;
6     logic [7:0] q_beh;
7
8     counter DUT(.data_in(data_in),.q(q),.clk(clk),.inc(inc),.ld(ld),.rst(rst));
9
10    always #10 clk = ~clk;
11
12    initial begin : reset_block
13        clk = 0;
14        rst = 0;
15        @(posedge clk);
16        @(negedge clk);
17        rst = 1;
18        data_in = $random;
19        ld = 1'b1;
20        inc = 1'b0;
21        @(negedge clk);
22        @(negedge clk);
23        repeat(50) begin : do_mult_stim
24            {ld, inc} = $random;
25            data_in = $random;
26            @(negedge clk);
27            @(negedge clk);
28        end
29        $stop;
30    end
31    always @(posedge clk) begin: beh_cntr;
32        if (!rst)
33            q_beh <= 0;
34        else
35            if (ld)
36                q_beh <= data_in;
37            else if (inc)
38                q_beh++;
39            else
40                q_beh <= q_beh + 1;
41        end
42    end
43    always @(negedge clk) begin : test_dut
44        assert(q == q_beh);
45    end
46
47    initial begin
48        $monitor("clk=%0d rst=%0d ld=%0d inc=%0d data_in=%0h q=%0h q_beh=%0h", clk, rst, ld, inc, data_in, q, q_beh);
49    end
50
51 endmodule // top

```

Resultado simulación del programa modificado:

```
# clk=1 rst=1 ld=0 inc=1 data_in=76 q=3 q_beh=3
# clk=0 rst=1 ld=0 inc=1 data_in=ed q=3 q_beh=3
# clk=1 rst=1 ld=0 inc=1 data_in=ed q=4 q_beh=4
# clk=0 rst=1 ld=0 inc=1 data_in=ed q=4 q_beh=4
# clk=1 rst=1 ld=0 inc=1 data_in=ed q=5 q_beh=5
# clk=0 rst=1 ld=0 inc=0 data_in=f9 q=5 q_beh=5
# clk=1 rst=1 ld=0 inc=0 data_in=f9 q=6 q_beh=6
# clk=0 rst=1 ld=0 inc=0 data_in=f9 q=6 q_beh=6
# clk=1 rst=1 ld=0 inc=0 data_in=f9 q=7 q_beh=7
# clk=0 rst=1 ld=1 inc=0 data_in=c5 q=7 q_beh=7
# clk=1 rst=1 ld=1 inc=0 data_in=c5 q=c5 q_beh=c5
# clk=0 rst=1 ld=1 inc=0 data_in=c5 q=c5 q_beh=c5
```

Como podemos observar ya no tenemos errores de assert cuando ld=inc=0 gracias a la modificación anterior.

Conclusión

En este código, se ha implementado un entorno de prueba para un módulo de contador en lenguaje de descripción de hardware (HDL). El entorno de prueba, definido en el módulo "top", genera señales de reloj, controla la entrada del contador y verifica la salida del contador.

El módulo de contador, llamado "DUT", se instancia en el módulo "top" y se conectan las señales de entrada y salida correspondientes. El módulo de contador tiene la capacidad de cargar datos, incrementar el contador y realizar un reinicio.

El bloque "reset_block" en la sección "initial" del código establece los valores iniciales de las señales de reloj y reinicio. Luego, se generan estimulaciones aleatorias para las señales de entrada del contador ("ld" y "inc") y se esperan flancos de bajada de la señal de reloj para realizar las operaciones correspondientes.

Además, se simula el comportamiento esperado del contador mediante el bloque "beh_cntr". Este bloque se activa en cada flanco de subida de la señal de reloj y se encarga de cargar datos en el contador, incrementarlo o restablecerlo según corresponda.

Finalmente, se verifica la salida del contador utilizando el bloque "test_dut", que se activa en cada flanco de bajada de la señal de reloj. Se utiliza la declaración "assert" para verificar si la salida del contador coincide con el valor simulado esperado.

En conclusión, este código proporciona un entorno de prueba funcional para el módulo de contador. Permite simular y verificar el comportamiento del contador en un entorno de simulación.

Referencias

1. Brock J. LaMeres, Introduction to Logic Circuits & Logic Design with Verilog, Springer, 1st Edition, USA, 2017.
2. Harris, D., & Harris, S. (2013). Digital design and computer architecture (2nd ed.). Morgan Kaufmann Publishers.
3. Palnitkar, S. (2003). Verilog HDL: A guide to digital design and synthesis (2nd ed.). Prentice Hall.
4. Weste, N. H. E., & Harris, D. (2011). CMOS VLSI design: A circuits and systems perspective (4th ed.). Addison-Wesley.