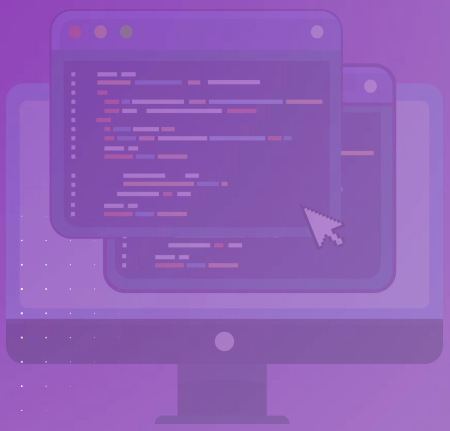
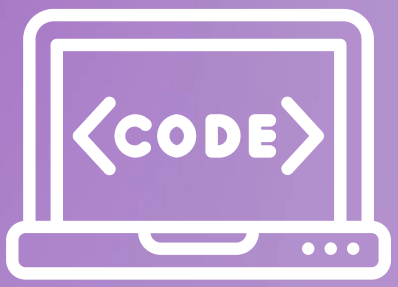




VERILOG

EXAMPLES

JAIRAJ MIRASHI
DESIGN VERIFICATION ENGINEER

Verilog Hello World

It's always best to get started using a very simple example, and none serves the purpose best other than "Hello World !".

```
// Single line comments start with double forward slash "//"
// Verilog code is always written inside modules, and each module represents a
digital block with some functionality
module tb;

                    // Initial block is another construct typically used to initialize signal nets and
variables for simulation
initial
// Verilog supports displaying signal values to the screen so that
designers can debug whats wrong with their circuit
// For our purposes, we'll simply display "Hello World"
$display ("Hello World !");
endmodule
```

A module called tb with no input-output ports act as the top module for the simulation. The initial block starts and executes the first statement at time 0 units. $display is a Verilog system task used to display a formatted string to the console and cannot be synthesized into hardware. Its primarily used to help with testbench and design debug. In this case, the text message displayed onto the screen is "Hello World !".

Simulation Log

```
ncsim> run
Hello World !
ncsim: *W,RNQUIE: Simulation is complete.
```

D Latch

A latch has two inputs : data(D), clock(clk) and one output: data(Q). When the clock is high, D flows through to Q and is transparent, but when the clock is low the latch holds its output Q even if D changes.

A flip-flop on the other hand captures data at its input at the positive or negative edge of a clock and output does not reflect changes in the input until the next clock edge.

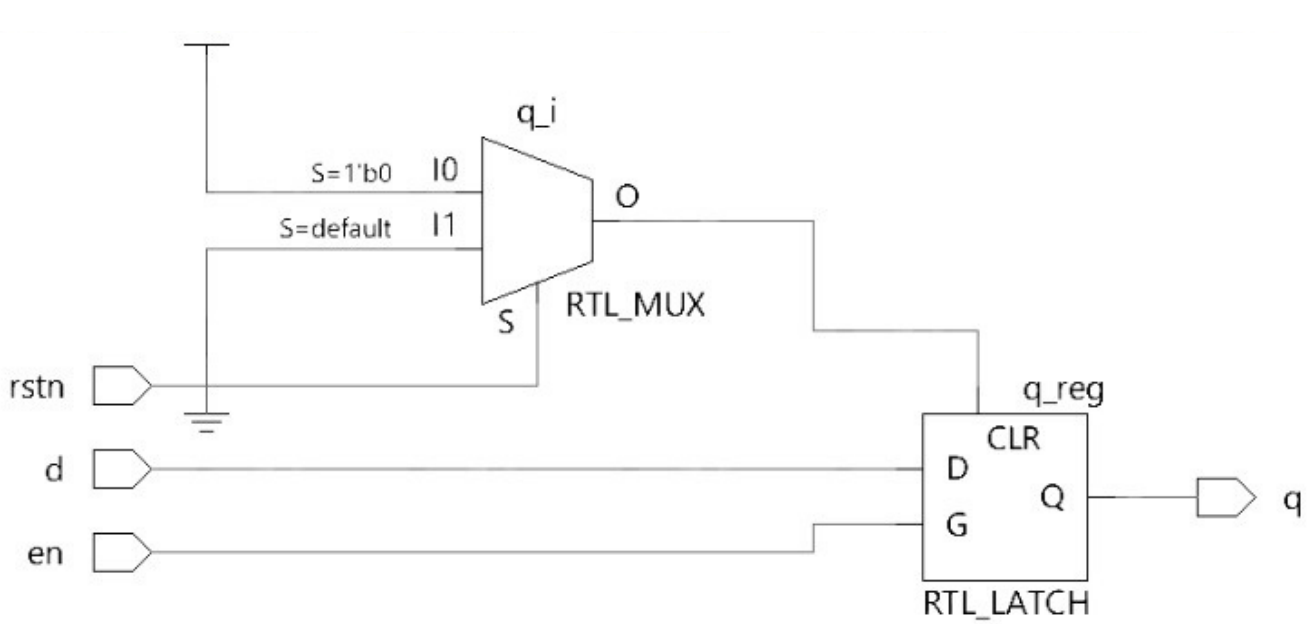
Design

In this example, we'll build a latch that has three inputs and one output. The input d stands for data which can be either 0 or 1, rstn stands for active-low reset and en stands for enable which is used to make the input data latch to the output. Reset being active-low simply means that the design element will be reset when this input goes to 0 or in other words, reset is active when its value is low. The value of output q is dictated by the inputs d, en and rstn.

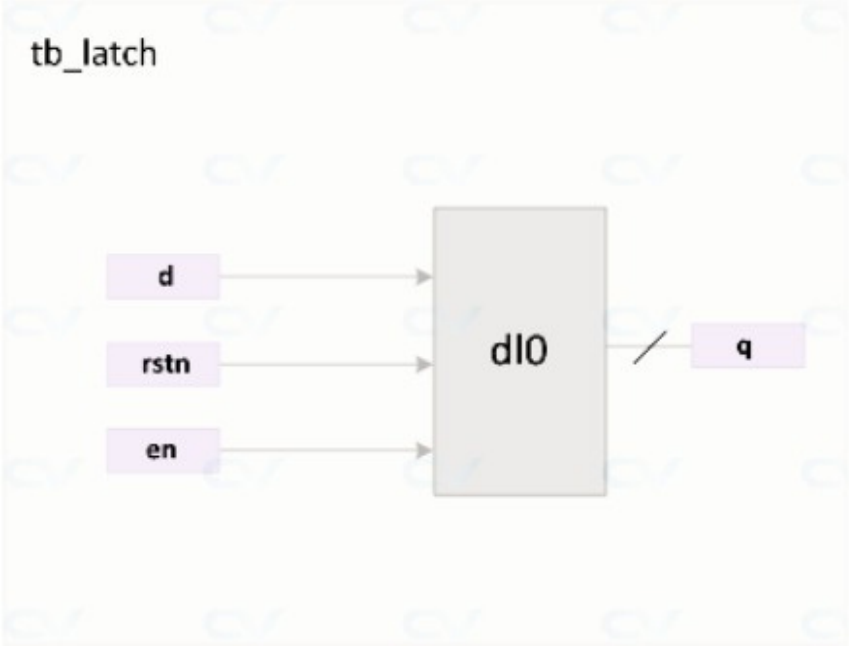

```
module d_latch ( input d, // 1-bit input pin for data
                                        input en, // 1-bit input pin for enabling the latch
input rstn, // 1-bit input pin for active-low reset
output reg q); // 1-bit output pin for data output

// This always block is "always" triggered whenever en/rstn/d changes
// If reset is asserted then output will be zero
// Else as long as enable is high, output q follows input d
always @ (en or rstn or d)
if (!rstn)
q <= 0;
else
if (en)
q <= d;
endmodule
```

Note that the sensitivity list to the always block contains all the signals required to update the output. This block will be triggered whenever any of the signals in the sensitivity list changes its value. Also q will get the value of d only when en is high, and hence is a *positive* latch.

Schematic



Testbench



```
module tb_latch;
            // Declare variables that can be used to drive values to the design
reg d;
reg en;
reg rstn;
reg [2:0] delay;
reg [1:0] delay2;
integer i;

// Instantiate design and connect design ports with TB signals
d_latch dl0 ( .d (d),
.en (en),
.rstn (rstn),
.q (q));

// This initial block forms the stimulus to test the design
initial begin
$monitor ("[%0t] en=%0b d=%0b q=%0b", $time, en, d, q);

// 1. Initialize testbench variables
d <= 0;
en <= 0;
rstn <= 0;

// 2. Release reset
#10 rstn <= 1;

// 3. Randomly change d and enable
for (i = 0; i < 5; i=i+1) begin
delay = $random;
delay2 = $random;
#(delay2) en <= ~en;
#(delay) d <= i;
end
end
endmodule
```

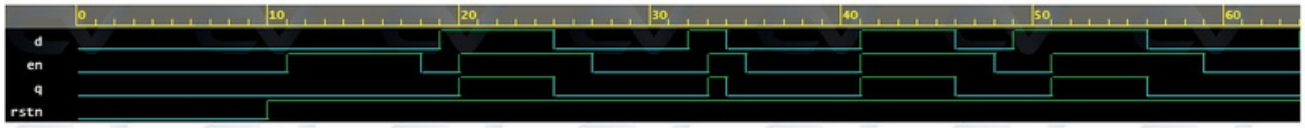
To make our testbench assert and deassert signals in a more random manner, we have declared a reg variable called delay of size 3 bits so that it can take any value from 0 to 7. Then the delay variable is used to delay the assignment of d and en to get different patterns in every loop.

Output

Simulation Log

```
ncsim> run
[0] en=0 d=0 q=0
[11] en=1 d=0 q=0
[18] en=0 d=0 q=0
[19] en=0 d=1 q=0
[20] en=1 d=1 q=1
[25] en=1 d=0 q=0
[27] en=0 d=0 q=0
[32] en=0 d=1 q=0
[33] en=1 d=1 q=1
[34] en=1 d=0 q=0
ncsim: *W,RNQUIE: Simulation is complete.
```

Click on the image to make it larger.



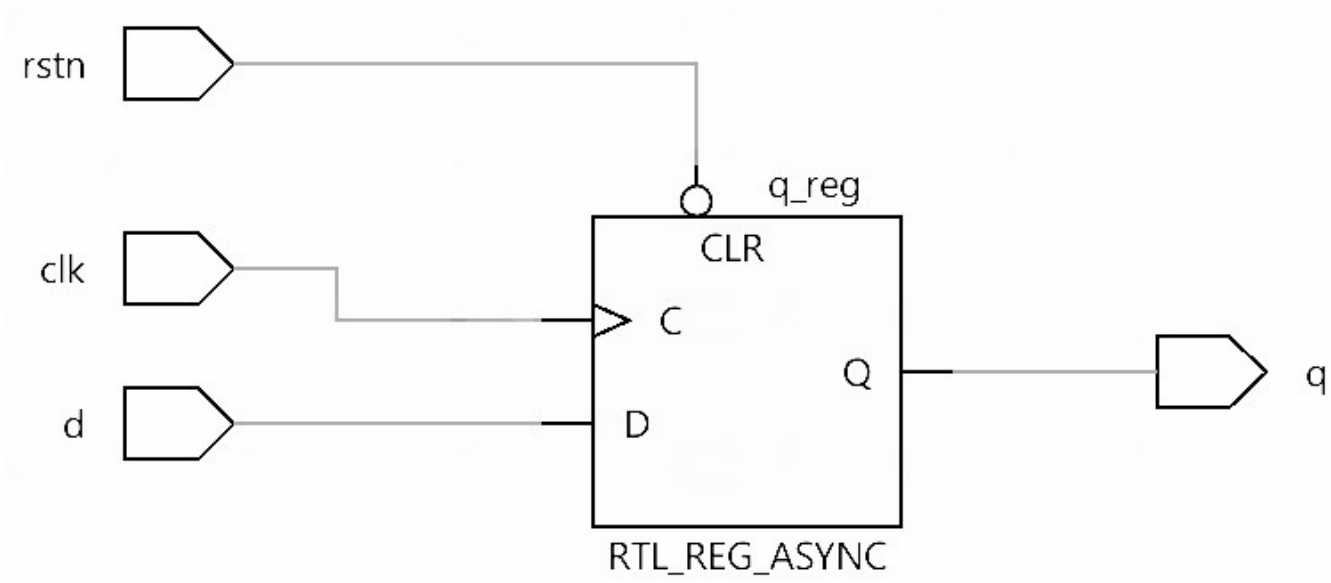
D Flip-Flop Async Reset

A *D flip-flop* is a sequential element that follows the input pin *d* at the given edge of a clock.

Design #1: With async active-low reset

```
module dff ( input d,
input rstn,
input clk,
output reg q);

            always @ (posedge clk or negedge rstn)
if (!rstn)
q <= 0;
else
q <= d;
endmodule
```

Hardware Schematic



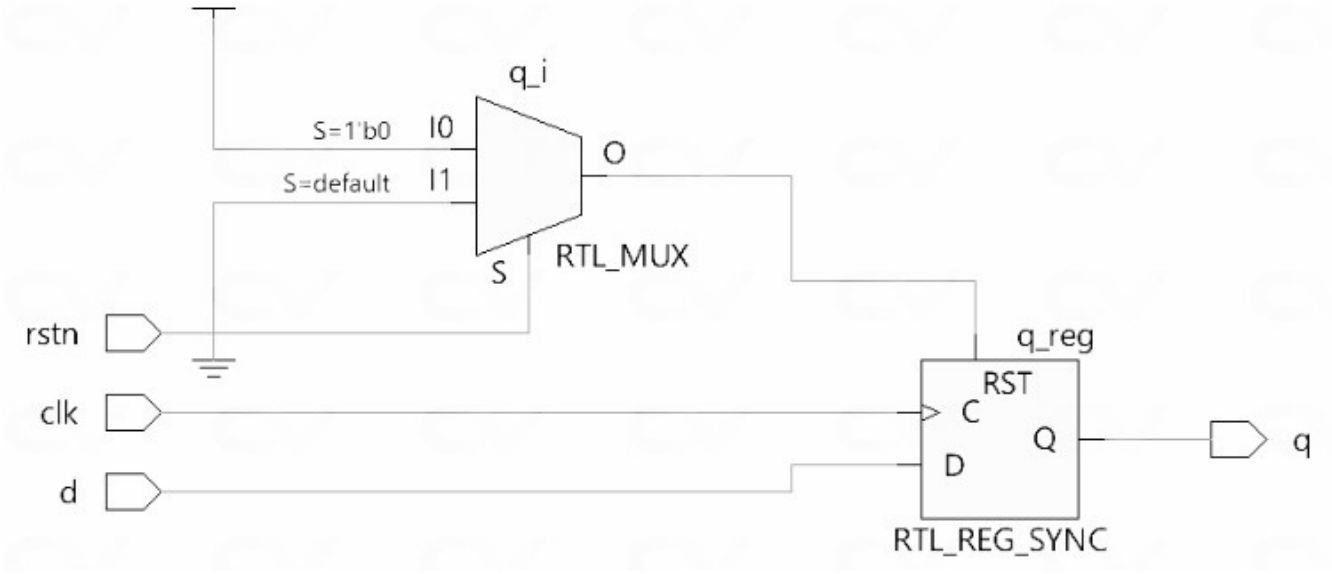
Testbench

```
module tb_dff;
reg clk;
reg d;
reg rstn;
reg [2:0] delay;

dff dff0 ( .d(d),
.rsnt (rstn),
                              .clk (clk),
.q (q));

// Generate clock
always #10 clk = ~clk;

// Testcase
initial begin
clk <= 0;
d <= 0;
rstn <= 0;

#15 d <= 1;
#10 rstn <= 1;
                          for (int i = 0; i < 5; i=i+1) begin
delay = $random;
#(delay) d <= i;
end
end
endmodule
```

Design #1: With sync active-low reset

```
module dff ( input d,
input rstn,
input clk,
output reg q);

                always @ (posedge clk)
if (!rstn)
q <= 0;
else
q <= d;
endmodule
```

Hardware Schematic



Testbench

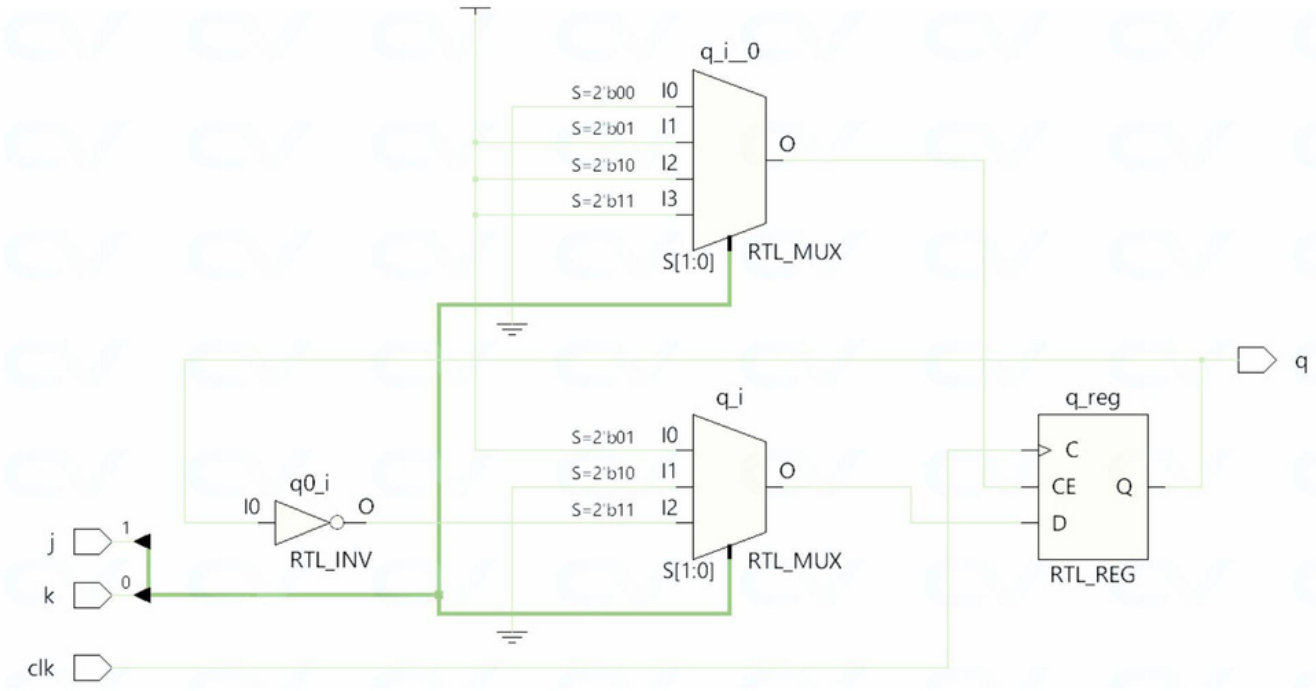
```
module tb_dff;
reg clk;
reg d;
reg rstn;
reg [2:0] delay;

dff dff0 ( .d(d),
.rsnt (rstn),
                        .clk (clk),
.q (q));

// Generate clock
always #10 clk = ~clk;

// Testcase
initial begin
clk <= 0;
d <= 0;
rstn <= 0;

#15 d <= 1;
#10 rstn <= 1;
                    for (int i = 0; i < 5; i=i+1) begin
delay = $random;
#(delay) d <= i;
end
end
endmodule
```

JK Flip Flop

Design

```
module jk_ff ( input j,
input k,
input clk,
output q);

reg q;

always @ (posedge clk)
case ({j,k})
2'b00 : q <= q;
2'b01 : q <= 0;
2'b10 : q <= 1;
                    2'b11 : q <= ~q;
endcase
endmodule
```

Hardware Schematic



Testbench

```
module tb_jk;
reg j;
reg k;
reg clk;

always #5 clk = ~clk;

jk_ff jk0 ( .j(j),
.k(k),
                                .clk(clk),
.q(q));

initial begin
j <= 0;
k <= 0;

#5 j <= 0;
k <= 1;
#20 j <= 1;
k <= 0;
#20 j <= 1;
k <= 1;
#20 $finish;
end

initial
                $monitor ("j=%0d k=%0d q=%0d", j, k, q);
endmodule
```

Verilog T Flip Flop

Design

```
module tff ( input clk,
input rstn,
input t,
output reg q);

    always @ (posedge clk) begin
if (!rstn)
q <= 0;
else
if (t)
q <= ~q;
else
q <= q;
end
endmodule
```

Testbench

```
module tb;
reg clk;
reg rstn;
reg t;

tff u0 ( .clk(clk),
.rstn(rstn),
.t(t),
.q(q));

always #5 clk = ~clk;

initial begin
{rstn, clk, t} <= 0;

            $monitor ("T=%0t rstn=%0b t=%0d q=%0d", $time, rstn, t, q);
repeat(2) @(posedge clk);
rstn <= 1;

for (integer i = 0; i < 20; i = i+1) begin
reg [4:0] dly = $random;
#(dly) t <= $random;
end
#20 $finish;
end
endmodule
```

Simulation Log

```
ncsim> run
T=0 rstn=0 t=0 q=x
T=5 rstn=0 t=0 q=0
T=15 rstn=1 t=0 q=0
T=19 rstn=1 t=1 q=0
T=25 rstn=1 t=1 q=1
T=35 rstn=1 t=1 q=0
T=43 rstn=1 t=0 q=0
T=47 rstn=1 t=1 q=0
T=55 rstn=1 t=0 q=1
T=59 rstn=1 t=1 q=1
T=65 rstn=1 t=1 q=0
T=67 rstn=1 t=0 q=0
T=71 rstn=1 t=1 q=0
T=75 rstn=1 t=0 q=1
T=79 rstn=1 t=1 q=1
T=83 rstn=1 t=0 q=1
T=87 rstn=1 t=1 q=1
T=95 rstn=1 t=0 q=0
Simulation complete via $finish(1) at time 115 NS + 0
```

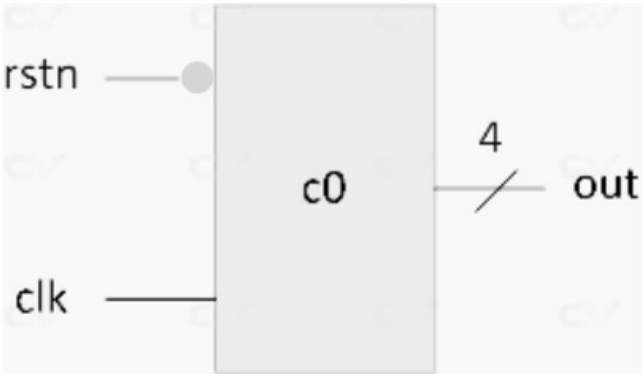
4-bit counter

The 4-bit counter starts incrementing from 4'b0000 to 4'h1111 and then rolls over back to 4'b0000. It will keep counting as long as it is provided with a running clock and reset is held high.

The rollover happens when the most significant bit of the final addition gets discarded. When counter is at a maximum value of 4'b1111 and gets one more count request, the counter tries to reach 5'b10000 but since it can support only 4-bits, the MSB will be discarded resulting in 0.

```
0000
0001
0010
...
1110
1111
          rolls over
0000
0001
...
```

The design contains two inputs one for the clock and another for an active-low reset. An active-low reset is one where the design is reset when the value of the reset pin is 0. There is a 4-bit output called out which essentially provides the counter values.



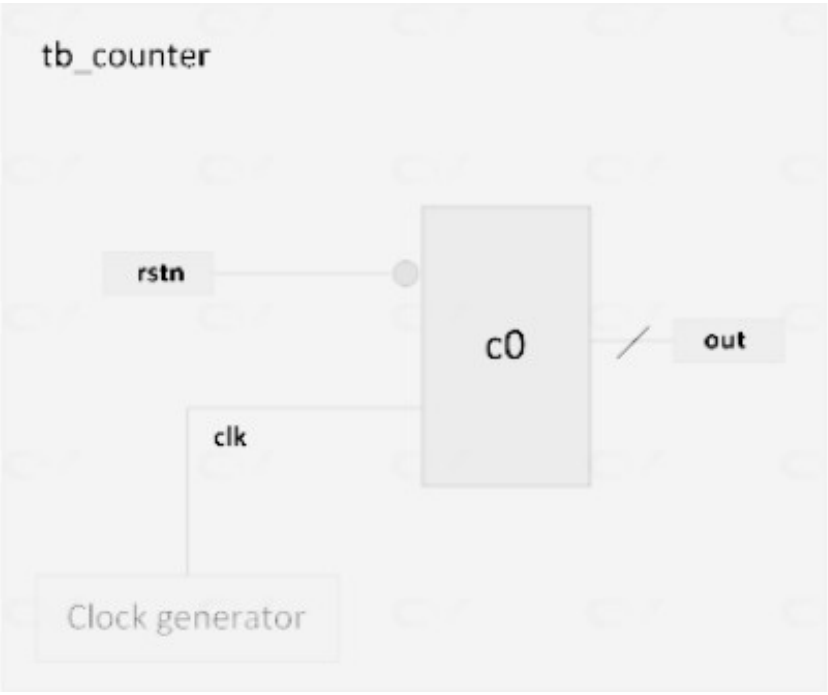
Electronic Counter Design

```
module counter ( input clk, // Declare input port for clock to allow counter to count up
                                                    input rstn, // Declare input port for reset to allow
the counter to be reset to 0 when required
output reg[3:0] out); // Declare 4-bit output port to get the
counter values

// This always block will be triggered at the rising edge of clk (0->1)
                                // Once inside this block, it checks if the reset is 0, if yes then change out to
zero
                            // If reset is 1, then design should be allowed to count up, so increment counter
always @ (posedge clk) begin
if (! rstn)
out <= 0;
else
out <= out + 1;
end
endmodule
```

The module counter has a clock and active-low reset (denoted by n) as inputs and the counter value as a 4-bit output. The always block is *always* executed whenever the clock transitions from 0 to 1 which signifies a rising edge or a positive edge. The output is incremented only if reset is held high or 1, achieved by the if-else block. If reset is found to be low at the positive edge of clock, then output is reset to a default value of 4'b0000.

Testbench

We can instantiate the design into our testbench module to verify that the counter is counting as expected.



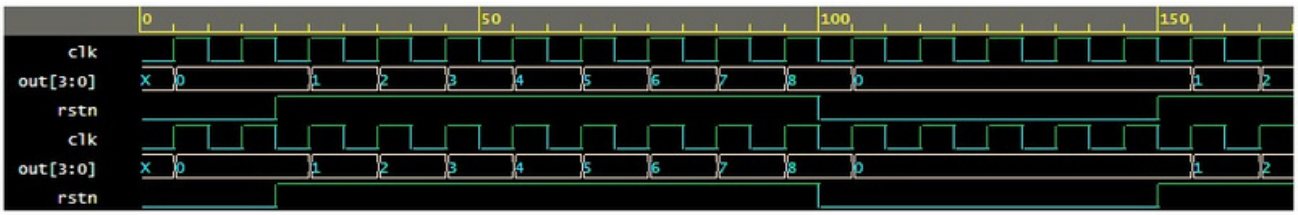
The testbench module is named tb_counter and ports are not required since this is the top-module in simulation. However we do need to have internal variables to generate, store and drive clock and reset. For that purpose, we have declared two variables of type reg for clock and reset. We also need a wire type net to make the connection with the design's output, else it will default to a 1-bit scalar net.

Clock is generated via always block which will give a period of 10 time units. The initial block is used to set initial values to our internal variables and drive the reset value to the design. The design is *instantiated* in the testbench and connected to our internal variables, so that it will get the values when we drive them from the testbench. We don't have any $display statements in our testbench and hence we will not see any message in the console.

```
module tb_counter;
                        reg clk; // Declare an internal TB variable called clk to
drive clock to the design
reg rstn; // Declare an internal TB variable called rstn to
drive active low reset to design
wire [3:0] out; // Declare a wire to connect to design output

// Instantiate counter design and connect with Testbench variables
counter c0 ( .clk (clk),
.rstn (rstn),
.out (out));

// Generate a clock that should be driven to design
                        // This clock will flip its value every 5ns -> time period = 10ns -> freq = 100
MHz
always #5 clk = ~clk;

// This initial block forms the stimulus of the testbench
initial begin
// 1. Initialize testbench variables to 0 at start of simulation
clk <= 0;
rstn <= 0;

// 2. Drive rest of the stimulus, reset is asserted in between
#20 rstn <= 1;
#80 rstn <= 0;
#50 rstn <= 1;

// 3. Finish the stimulus after 200ns
#20 $finish;
end
endmodule
```

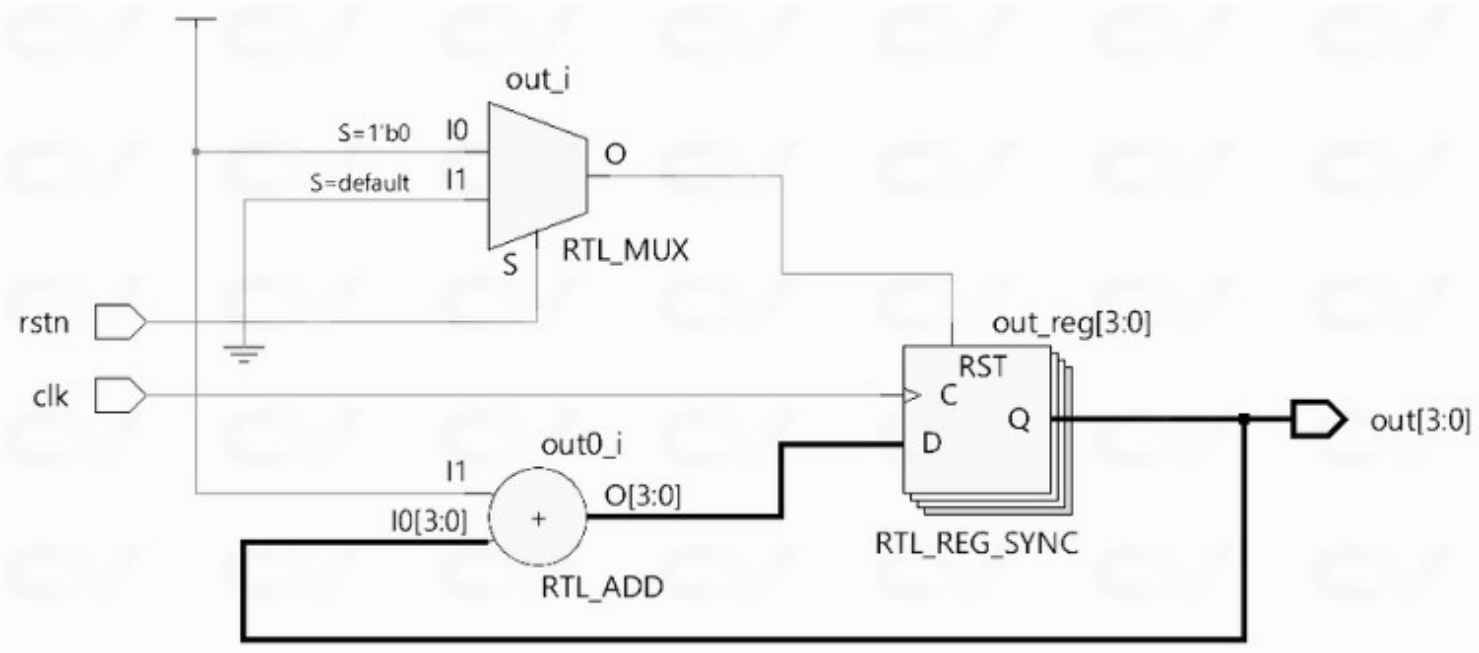
Simulation Log

```
ncsim> run
[0ns] clk=0 rstn=0 out=0xx
[5ns] clk=1 rstn=0 out=0x0
[10ns] clk=0 rstn=0 out=0x0
[15ns] clk=1 rstn=0 out=0x0
[20ns] clk=0 rstn=1 out=0x0
[25ns] clk=1 rstn=1 out=0x1
[30ns] clk=0 rstn=1 out=0x1
[35ns] clk=1 rstn=1 out=0x2
[40ns] clk=0 rstn=1 out=0x2
[45ns] clk=1 rstn=1 out=0x3
[50ns] clk=0 rstn=1 out=0x3
[55ns] clk=1 rstn=1 out=0x4
[60ns] clk=0 rstn=1 out=0x4
[65ns] clk=1 rstn=1 out=0x5
[70ns] clk=0 rstn=1 out=0x5
[75ns] clk=1 rstn=1 out=0x6
[80ns] clk=0 rstn=1 out=0x6
[85ns] clk=1 rstn=1 out=0x7
[90ns] clk=0 rstn=1 out=0x7
[95ns] clk=1 rstn=1 out=0x8
[100ns] clk=0 rstn=0 out=0x8
[105ns] clk=1 rstn=0 out=0x0
[110ns] clk=0 rstn=0 out=0x0
[115ns] clk=1 rstn=0 out=0x0
[120ns] clk=0 rstn=0 out=0x0
[125ns] clk=1 rstn=0 out=0x0
[130ns] clk=0 rstn=0 out=0x0
[135ns] clk=1 rstn=0 out=0x0
[140ns] clk=0 rstn=0 out=0x0
[145ns] clk=1 rstn=0 out=0x0
[150ns] clk=0 rstn=1 out=0x0
[155ns] clk=1 rstn=1 out=0x1
[160ns] clk=0 rstn=1 out=0x1
[165ns] clk=1 rstn=1 out=0x2
Simulation complete via $finish(1) at time 170 NS + 0
```



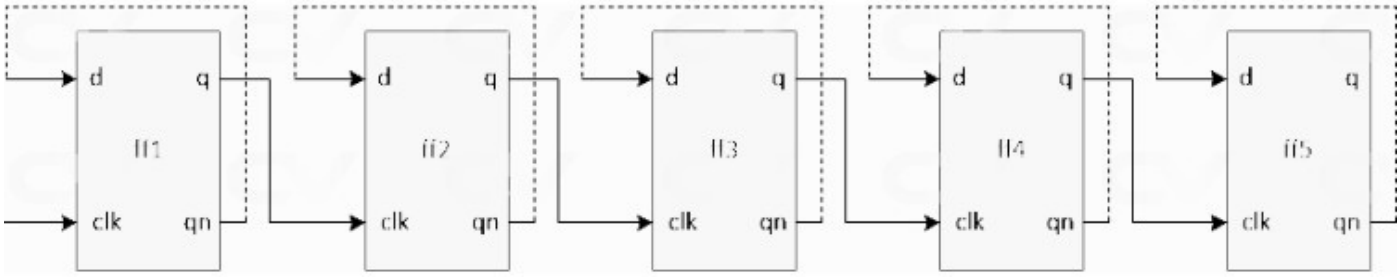
Note that the counter resets to 0 when the active-low reset becomes 0, and when reset is de-asserted at around 150ns, the counter starts counting from the next occurence of the positive edge of clock.

Hardware Schematic



Verilog Ripple Counter

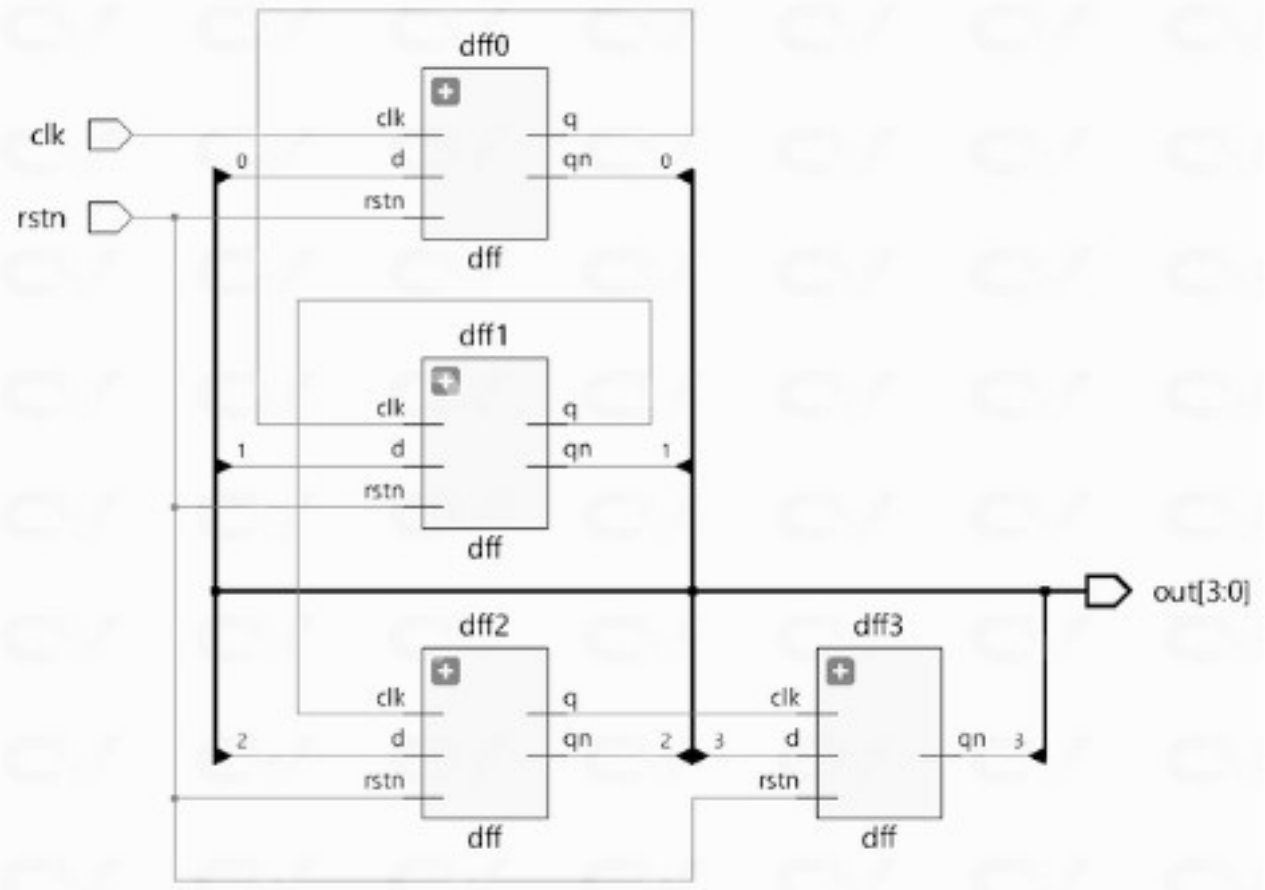
A *ripple* counter is an asynchronous counter in which the all the flops except the first are clocked by the output of the preceding flop.



Design

```
module dff ( input d,
input clk,
                        input rstn,
output reg q,
                       output qn);
      always @ (posedge clk or negedge rstn)
if (!rstn)
q <= 0;
else
q <= d;

assign qn = ~q;
endmodule

module ripple ( input clk,
input rstn,
output [3:0] out);
wire q0;
wire qn0;
wire q1;
wire qn1;
wire q2;
wire qn2;
wire q3;
wire qn3;

dff dff0 ( .d (qn0),
.clk (clk),
.rstn (rstn),
.q (q0),
.qn (qn0));

dff dff1 ( .d (qn1),
.clk (q0),
.rstn (rstn),
.q (q1),
.qn (qn1));

dff dff2 ( .d (qn2),
.clk (q1),
.rstn (rstn),
.q (q2),
.qn (qn2));

dff dff3 ( .d (qn3),
.clk (q2),
.rstn (rstn),
.q (q3),
.qn (qn3));

assign out = {qn3, qn2, qn1, qn0};

endmodule
```

Hardware Schematic

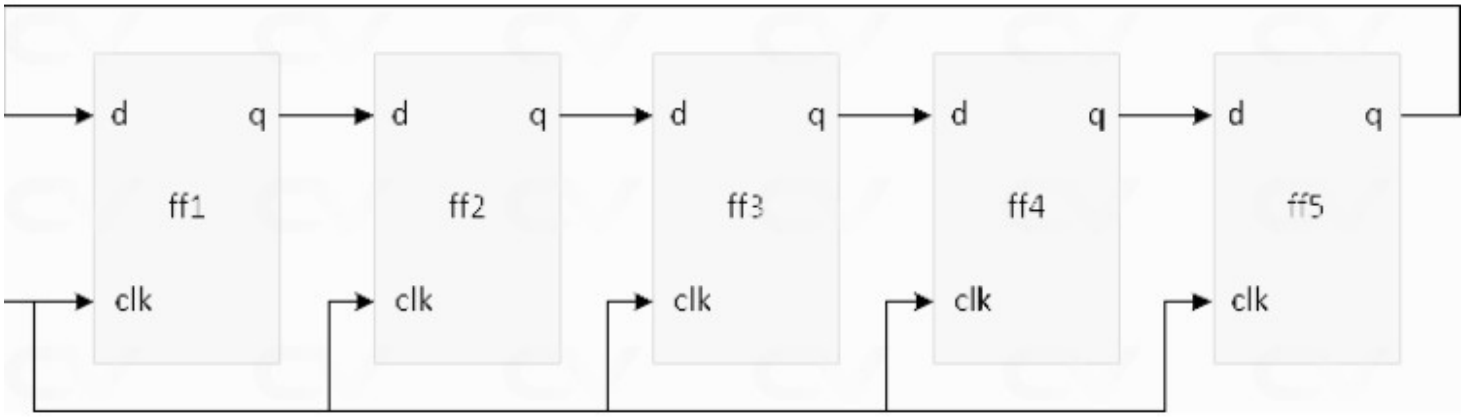


Testbench

```
module tb_ripple;
reg clk;
reg rstn;
wire [3:0] out;

ripple r0 ( .clk (clk),
.rstn (rstn),
.out (out));

always #5 clk = ~clk;

initial begin
rstn <= 0;
clk <= 0;

repeat (4) @ (posedge clk);
rstn <= 1;

                repeat (25) @ (posedge clk);
$finish;
end
endmodule
```

Verilog Ring Counter

chipverify.com/verilog/verilog-ring-counter



Design

```
module ring_ctr #(parameter WIDTH=4)
(
input clk,
input rstn,
output reg [WIDTH-1:0] out
);

always @ (posedge clk) begin
if (!rstn)
out <= 1;
else begin
out[WIDTH-1] <= out[0];
                                for (int i = 0; i < WIDTH-1; i=i+1) begin
out[i] <= out[i+1];
end
end
end
endmodule
```

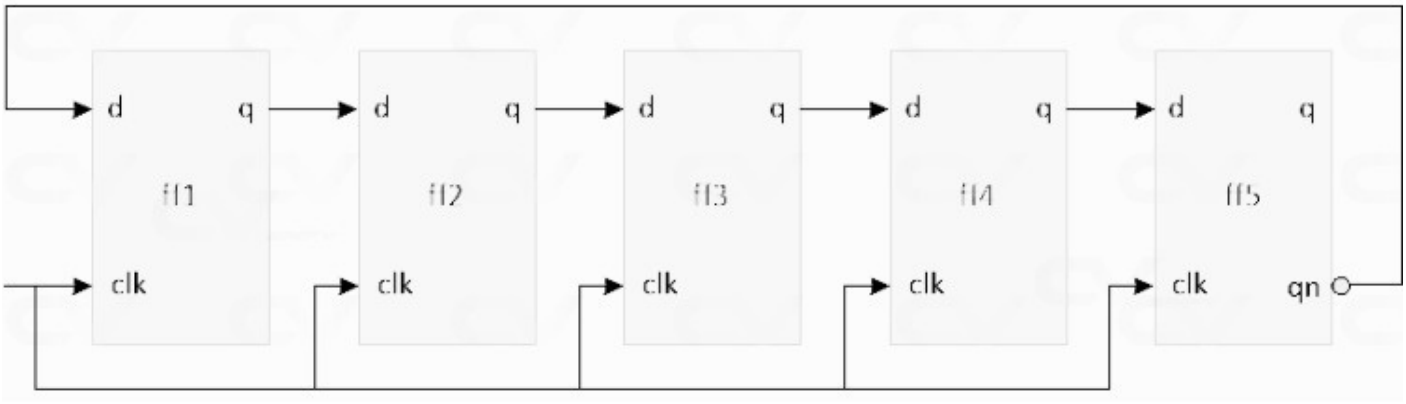
Testbench

```
module tb;
parameter WIDTH = 4;

reg clk;
reg rstn;
wire [WIDTH-1:0] out;

ring_ctr u0 (.clk (clk),
.rstn (rstn),
.out (out));

always #10 clk = ~clk;

initial begin
{clk, rstn} <= 0;

                $monitor ("T=%0t out=%b", $time, out);
repeat (2) @(posedge clk);
rstn <= 1;
repeat (15) @(posedge clk);
$finish;
end
endmodule
```

Simulation Log

```
ncsim> run
T=0 out=xxxx
T=10 out=0001
T=50 out=1
000
T=70 out=01
00
T=90 out=001
0
T=110 out=0001
T=130 out=1
000
T=150 out=01
00
T=170 out=001
0
T=190 out=0001
T=210 out=1000
T=230 out=0100
T=250 out=0010
T=270 out=0001
T=290 out=1000
T=310 out=0100
Simulation complete via $finish(1) at time 330 NS + 0
```

Verilog Johnson Counter



Design

```
module johnson_ctr #(parameter WIDTH=4)
(
input clk,
input rstn,
output reg [WIDTH-1:0] out
);

always @ (posedge clk) begin
if (!rstn)
out <= 1;
else begin
out[WIDTH-1] <= ~out[0];
                                    for (int i = 0; i < WIDTH-1; i=i+1) begin
out[i] <= out[i+1];
end
end
end
endmodule
```

Testbench

```
module tb;
parameter WIDTH = 4;

reg clk;
reg rstn;
wire [WIDTH-1:0] out;

johnson_ctr u0 (.clk (clk),
.rstn (rstn),
.out (out));

always #10 clk = ~clk;

initial begin
{clk, rstn} <= 0;

                $monitor ("T=%0t out=%b", $time, out);
repeat (2) @(posedge clk);
rstn <= 1;
repeat (15) @(posedge clk);
$finish;
end
endmodule
```

Simulation Log

```
ncsim> run
T=0 out=xxxx
T=10 out=0001
T=50 out=0000
T=70 out=1000
T=90 out=1100
T=110 out=1110
T=130 out=1111
T=150 out=0111
T=170 out=0011
T=190 out=0001
T=210 out=0000
T=230 out=1000
T=250 out=1100
T=270 out=1110
T=290 out=1111
T=310 out=0111
Simulation complete via $finish(1) at time 330 NS + 0
```

Verilog Mod-N counter

Design

```
module modN_ctr
# (parameter N = 10,
parameter WIDTH = 4)

( input clk,
input rstn,
                        output reg[WIDTH-1:0] out);

always @ (posedge clk) begin
if (!rstn) begin
out <= 0;
end else begin
if (out == N-1)
out <= 0;
else
out <= out + 1;
end
end
endmodule
```

Testbench

```
module tb;
parameter N = 10;
parameter WIDTH = 4;

reg clk;
reg rstn;
wire [WIDTH-1:0] out;

modN_ctr u0 ( .clk(clk),
                                .rstn(rstn),
.out(out));

always #10 clk = ~clk;

initial begin
{clk, rstn} <= 0;

        $monitor ("T=%0t rstn=%0b out=0x%0h", $time, rstn, out);
repeat(2) @ (posedge clk);
rstn <= 1;

repeat(20) @ (posedge clk);
$finish;
end
endmodule
```

Simulation Log

```
ncsim> run
T=0 rstn=0 out=0xx
T=10 rstn=0 out=0x0
T=30 rstn=1 out=0x0
T=50 rstn=1 out=0x1
T=70 rstn=1 out=0x2
T=90 rstn=1 out=0x3
T=110 rstn=1 out=0x4
T=130 rstn=1 out=0x5
T=150 rstn=1 out=0x6
T=170 rstn=1 out=0x7
T=190 rstn=1 out=0x8
T=210 rstn=1 out=0x9
T=230 rstn=1 out=0xa
T=250 rstn=1 out=0x0
T=270 rstn=1 out=0x1
T=290 rstn=1 out=0x2
T=310 rstn=1 out=0x3
T=330 rstn=1 out=0x4
T=350 rstn=1 out=0x5
T=370 rstn=1 out=0x6
T=390 rstn=1 out=0x7
T=410 rstn=1 out=0x8
Simulation complete via $finish(1) at time 430 NS + 0
```

Verilog Gray Counter

Design

```
module gray_ctr
# (parameter N = 4)

( input clk,
input rstn,
output reg [N-1:0] out);

reg [N-1:0] q;

always @ (posedge clk) begin
if (!rstn) begin
q <= 0;
out <= 0;
end else begin
q <= q + 1;

`ifdef FOR_LOOP
for (int i = 0; i < N-1; i= i+1) begin
out[i] <= q[i+1] ^ q[i];
end
out[N-1] <= q[N-1];
`else
                                        out <= {q[N-1], q[N-1:1] ^ q[N-2:0]};
`endif
end
end
endmodule
```

Testbench

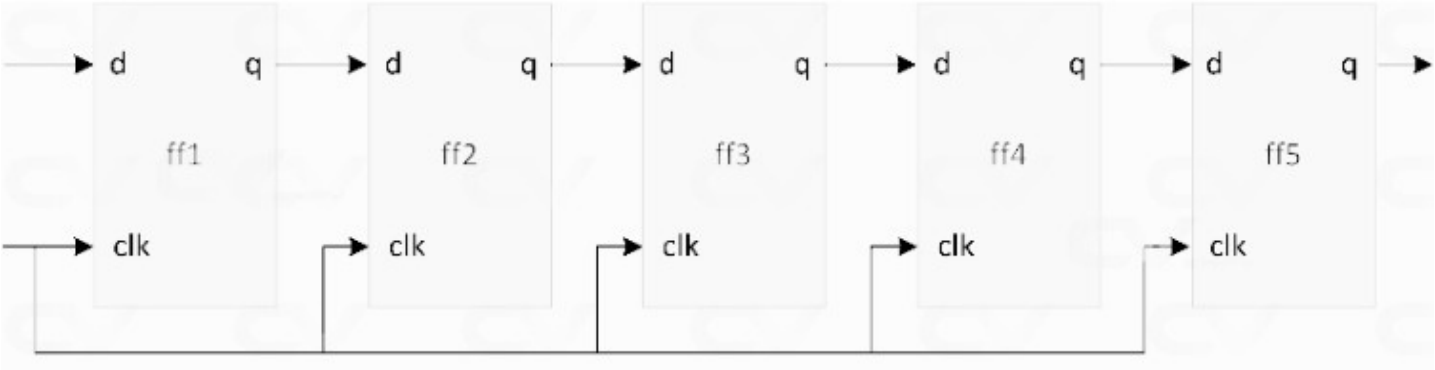
```
module tb;
parameter N = 4;

reg clk;
reg rstn;
wire [N-1:0] out;

gray_ctr u0 ( .clk(clk),
.rstn(rstn),
.out(out));

always #10 clk = ~clk;

initial begin
{clk, rstn} <= 0;

                $monitor ("T=%0t rstn=%0b out=0x%0h", $time, rstn, out);

repeat(2) @ (posedge clk);
rstn <= 1;
repeat(20) @ (posedge clk);
$finish;
end
endmodule
```

Simulation Log

```
ncsim> run
T=0 rstn=0 out=0xx
T=10 rstn=0 out=0x0
T=30 rstn=1 out=0x0
T=50 rstn=1 out=0x1
T=70 rstn=1 out=0x3
T=90 rstn=1 out=0x2
T=110 rstn=1 out=0x6
T=130 rstn=1 out=0x7
T=150 rstn=1 out=0x5
T=170 rstn=1 out=0x4
T=190 rstn=1 out=0xc
T=210 rstn=1 out=0xd
T=230 rstn=1 out=0xf
T=250 rstn=1 out=0xe
T=270 rstn=1 out=0xa
T=290 rstn=1 out=0xb
T=310 rstn=1 out=0x9
T=330 rstn=1 out=0x8
T=350 rstn=1 out=0x0
T=370 rstn=1 out=0x1
T=390 rstn=1 out=0x3
T=410 rstn=1 out=0x2
Simulation complete via $finish(1) at time 430 NS + 0
```

Verilog n-bit Bidirectional Shift Register

In digital electronics, a *shift register* is a cascade of flip-flops where the output pin *q* of one flop is connected to the data input pin (d) of the next. Because all flops work on the same clock, the bit array stored in the shift register will shift by one position. For example, if a 5-bit right shift register has an initial value of 10110 and the input to the shift register is tied to 0, then the next pattern will be 01011 and the next 00101.



Design

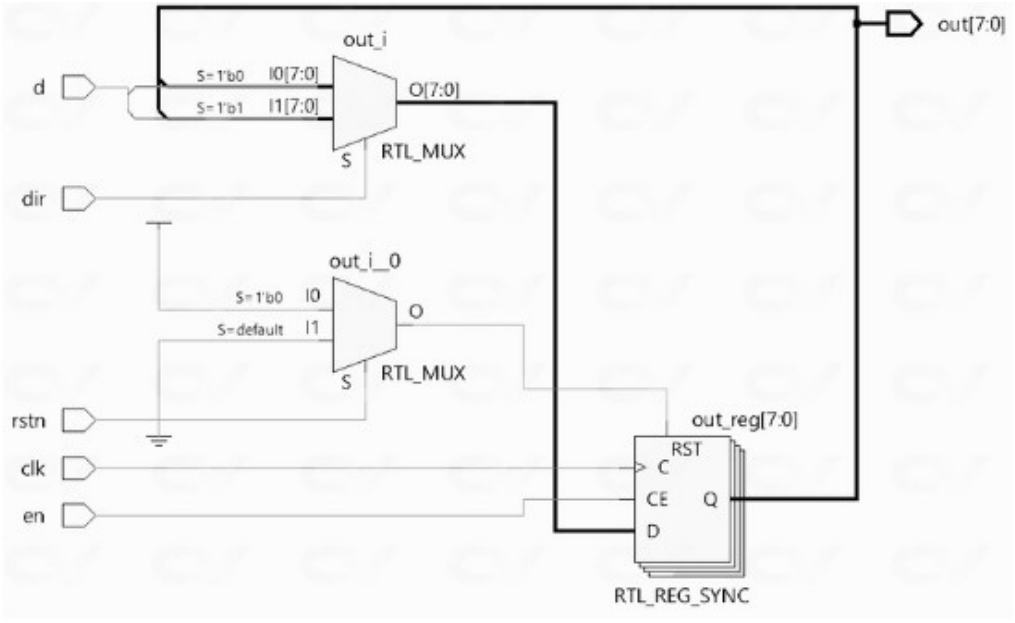
This shift register design has five inputs and one n-bit output and the design is parameterized using parameter MSB to signify width of the shift register. If n is 4, then it becomes a 4-bit shift register. If n is 8, then it becomes an 8-bit shift register.

This shift register has a few key features:

- Can be enabled or disbled by driving en pin of the design
- Can shift to the left as well as right when dir is driven
- If rstn is pulled low, it will reset the shift register and output will become 0
- Input data value of the shift register can be controlled by d pin

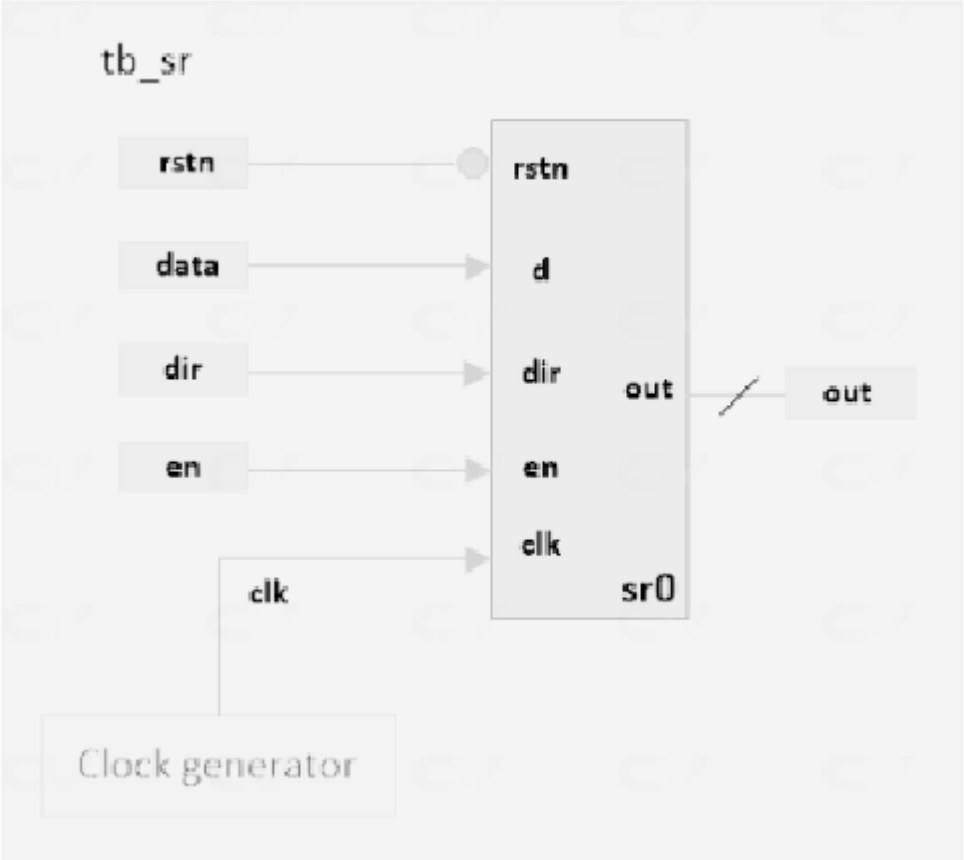
```
module shift_reg #(parameter MSB=8) ( input d, // Declare input for data to the first flop in the
shift register
                                                                                    input clk, // Declare
input for clock to all flops in the shift register
                                                                                    input en, // Declare
input for enable to switch the shift register on/off
                                                                                    input dir, // Declare
input to shift in either left or right direction
                                                                                    input rstn, // Declare
input to reset the register to a default value
                                                                                    output reg [MSB-1:0] out); // Declare
output to read out the current value of all flops in this register


 // This always block will "always" be triggered on the rising edge of clock
                                                    // Once it enters the block, it will first check to see if reset is 0 and if yes
then reset register
// If no, then check to see if the shift register is enabled
// If no => maintain previous output. If yes, then shift based on the requested
direction
always @ (posedge clk)
if (!rstn)
out <= 0;
else begin
if (en)
case (dir)
0 : out <= {out[MSB-2:0], d};
1 : out <= {d, out[MSB-1:1]};
endcase
else
out <= out;
end
endmodule
```

Hardware Schematic



Testbench

The testbench is used to verify the functionality of this shift register. The design is instantiated into the top module and the inputs are driven with different values. The design behavior for each of the inputs can be observed at the output pin out.



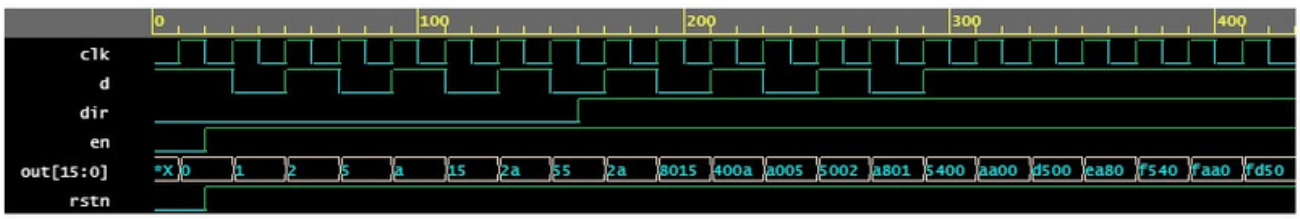
```
module tb_sr;
                                        parameter MSB = 16; // [Optional] Declare a parameter to represent number
of bits in shift register

reg data; // Declare a variable to drive d-input of design
                                    reg clk; // Declare a variable to drive clock to the design
reg en; // Declare a variable to drive enable to the design
reg dir; // Declare a variable to drive direction of shift
registe
                                   reg rstn; // Declare a variable to drive reset to the design
wire [MSB-1:0] out; // Declare a wire to capture output from the design

                                    // Instantiate design (16-bit shift register) by passing MSB and connect with TB
signals
shift_reg #(MSB) sr0 ( .d (data),
.clk (clk),
.en (en),
.dir (dir),
.rstn (rstn),
.out (out));

// Generate clock time period = 20ns, freq => 50MHz
always #10 clk = ~clk;

// Initialize variables to default values at time 0
initial begin
clk <= 0;
en <= 0;
dir <= 0;
rstn <= 0;
data <= 'h1;
end

// Drive main stimulus to the design to verify if this works
initial begin

// 1. Apply reset and deassert reset after some time
rstn <= 0;
#20 rstn <= 1;
en <= 1;

// 2. For 7 clocks, drive alternate values to data pin
repeat (7) @ (posedge clk)
data <= ~data;

                        // 4. Shift direction and drive alternate value to data pin for another 7
clocks
#10 dir <= 1;
repeat (7) @ (posedge clk)
data <= ~data;

// 5. Drive nothing for next 7 clocks, allow shift register to simply shift
```

```
based on dir
repeat (7) @ (posedge clk);

// 6. Finish the simulation
$finish;
end

                                        // Monitor values of these variables and print them into the logfile for debug
initial
                                      $monitor ("rstn=%0b data=%b, en=%0b, dir=%0b, out=%b", rstn, data, en, dir,
out);
endmodule
```

The time when shift register is enabled is highlighted in green in the log given below. The time when it shifts its direction is highlighted in yellow. The time when data input pin remains constant is highlighted in blue.

Simulation Log

```
ncsim> run
rstn=0 data=1, en=0, dir=0, out=xxxxxxxxxxxxxxxx
rstn=0 data=1, en=0, dir=0, out=0000000000000000
rstn=1 data=1, en=1, dir=0, out=0000000000000000
rstn=1 data=0, en=1, dir=0, out=0000000000000001
rstn=1 data=1, en=1, dir=0, out=0000000000000010
rstn=1 data=0, en=1, dir=0, out=0000000000000101
rstn=1 data=1, en=1, dir=0, out=0000000000001010
rstn=1 data=0, en=1, dir=0, out=0000000000010101
rstn=1 data=1, en=1, dir=0, out=0000000000101010
rstn=1 data=0, en=1, dir=0, out=0000000001010101
rstn=1 data=0, en=1, dir=1, out=0000000001010101
rstn=1 data=1, en=1, dir=1, out=0000000000101010
rstn=1 data=0, en=1, dir=1, out=1000000000010101
rstn=1 data=1, en=1, dir=1, out=0100000000001010
rstn=1 data=0, en=1, dir=1, out=1010000000000101
rstn=1 data=1, en=1, dir=1, out=0101000000000010
rstn=1 data=0, en=1, dir=1, out=1010100000000001
rstn=1 data=1, en=1, dir=1, out=0101010000000000
rstn=1 data=1, en=1, dir=1, out=1010101000000000
rstn=1 data=1, en=1, dir=1, out=1101010100000000
rstn=1 data=1, en=1, dir=1, out=1110101010000000
rstn=1 data=1, en=1, dir=1, out=1111010101000000
rstn=1 data=1, en=1, dir=1, out=1111101010100000
rstn=1 data=1, en=1, dir=1, out=1111110101010000
Simulation complete via $finish(1) at time 430 NS + 0
```



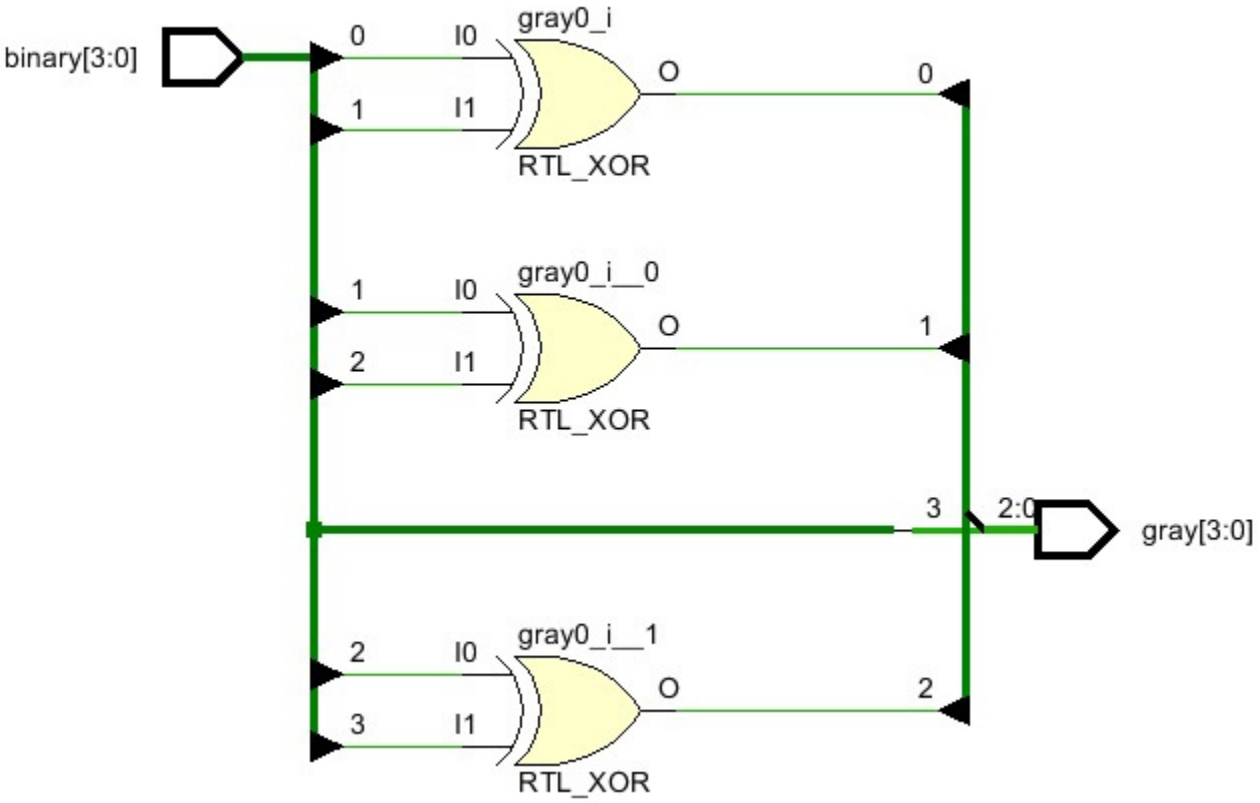
Verilog Binary to Gray

Gray code is a binary code where each successive value differs from the previous value by only one bit.

Implementation #1

```
module bin2gray #(parameter N=4) ( input [N-1:0] bin,
                                   output [N-1:0] gray);

genvar i;
generate
for(i = 0; i < N-1; i = i + 1) begin
assign gray[i] = bin[i] ^ bin[i+1];
end
endgenerate

assign gray[N-1] = bin[N-1];
endmodule
```

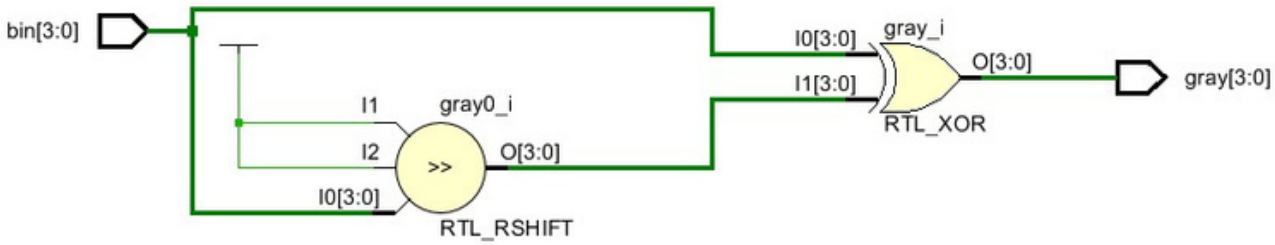
Hardware Schematic



Implementation #2

```
module bin2gray #(parameter N=4) ( input [N-1:0] bin,
                                   output [N-1:0] gray);

assign gray = bin ^ (bin >> 1);

endmodule
```

Hardware Schematic



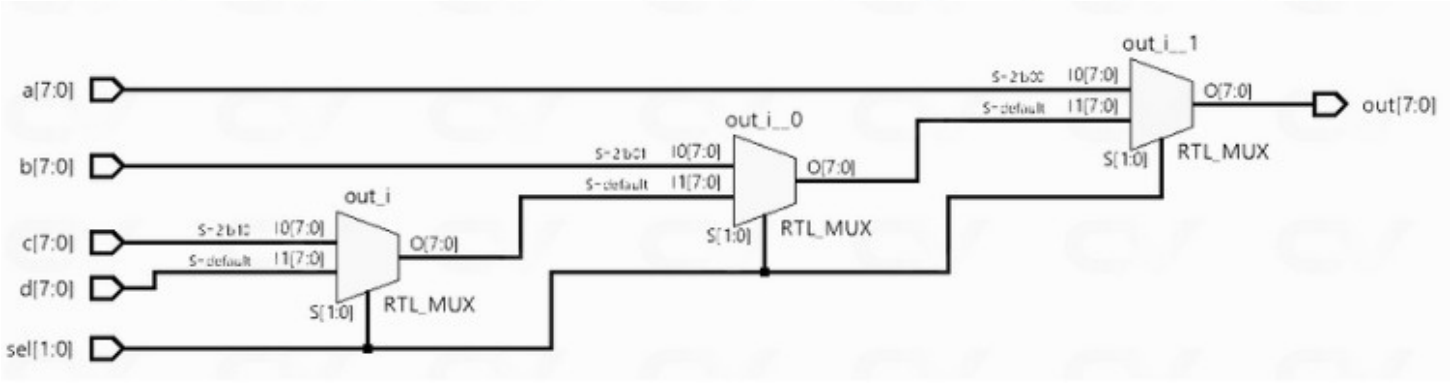
Note that the second implementation resulted in the synthesis of a shift register as implied by the >> operator and will occupy more area and power.

Verilog Priority Encoder

Design

```
module pr_en ( input [7:0] a,
input [7:0] b,
input [7:0] c,
input [7:0] d,
input [1:0] sel,
output reg [7:0] out);

                        always @ (a or b or c or d or sel) begin
if (sel == 2'b00)
out <= a;
else if (sel == 2'b01)
out <= b;
else if (sel == 2'b10)
out <= c;
else
out <= d;
end
endmodule
```

Hardware Schematic



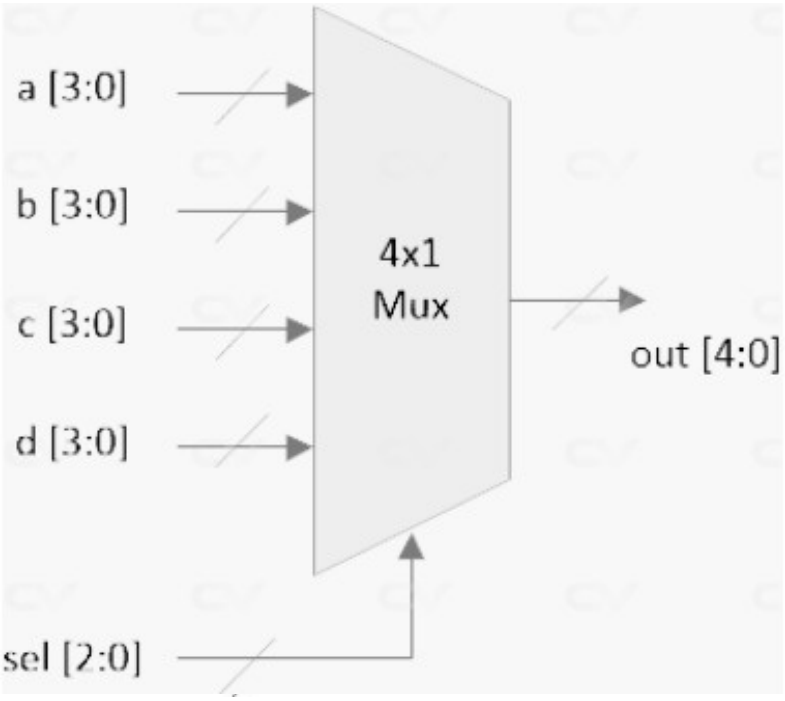
Testbench

```
module tb_4to1_mux;
reg [7:0] a;
reg [7:0] b;
reg [7:0] c;
reg [7:0] d;
wire [7:0] out;
reg [1:0] sel;
integer i;

pr_en pr_en0 ( .a (a),
.b (b),
.c (c),
.d (d),
.sel (sel),

.out (out));


initial begin
sel <= 0;
a <= $random;
b <= $random;
c <= $random;
d <= $random;

for (i = 1; i < 4; i=i+1) begin
#5 sel <= i;
end

#5 $finish;
end
endmodule
```

Verilog 4 to 1 Multiplexer/Mux

What is a mux or multiplexer ?

A multiplexer or *mux* in short, is a digital element that transfers data from one of the N inputs to the output based on the select signal. The case shown below is when N equals 4. For example, a 4 bit multiplexer would have N inputs each of 4 bits where each input can be transferred to the output by the use of a select signal.



sel is a 2-bit input and can have four values. Each value on the select line will allow one of the inputs to be sent to output pin out.

```
sel a b c d out
0 3 7 1 9 3
1 3 7 1 9 7
2 3 7 1 9 1
3 3 7 1 9 9
```

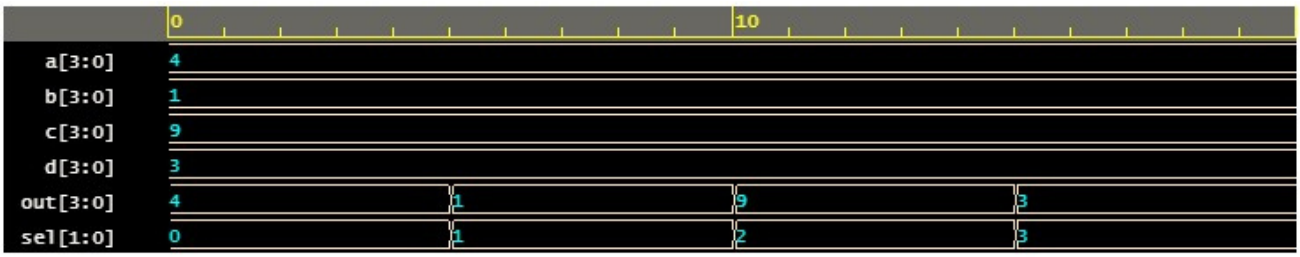
A 4x1 multiplexer can be implemented in multiple ways and here you'll see two of the most common ways:

- Using an assign statement
- Using a case statement

Using assign statement

```
module mux_4to1_assign ( input [3:0] a, // 4-bit input called a
input [3:0] b, // 4-bit input called b
input [3:0] c, // 4-bit input called c
input [3:0] d, // 4-bit input called d
                                                    input [1:0] sel, // input sel used to select
between a,b,c,d
output [3:0] out); // 4-bit output based on
input sel

// When sel[1] is 0, (sel[0]? b:a) is selected and when sel[1] is 1, (sel[0] ?
d:c) is taken
                                        // When sel[0] is 0, a is sent to output, else b and when sel[0] is 0, c is sent
to output, else d
assign out = sel[1] ? (sel[0] ? d : c) : (sel[0] ? b : a);

endmodule
```

The module called *mux_4x1_assign* has four 4-bit data inputs, one 2-bit select input and one 4-bit data output. The multiplexer will select either a , b, c, or d based on the select signal sel using the assign statement.



Using case statement

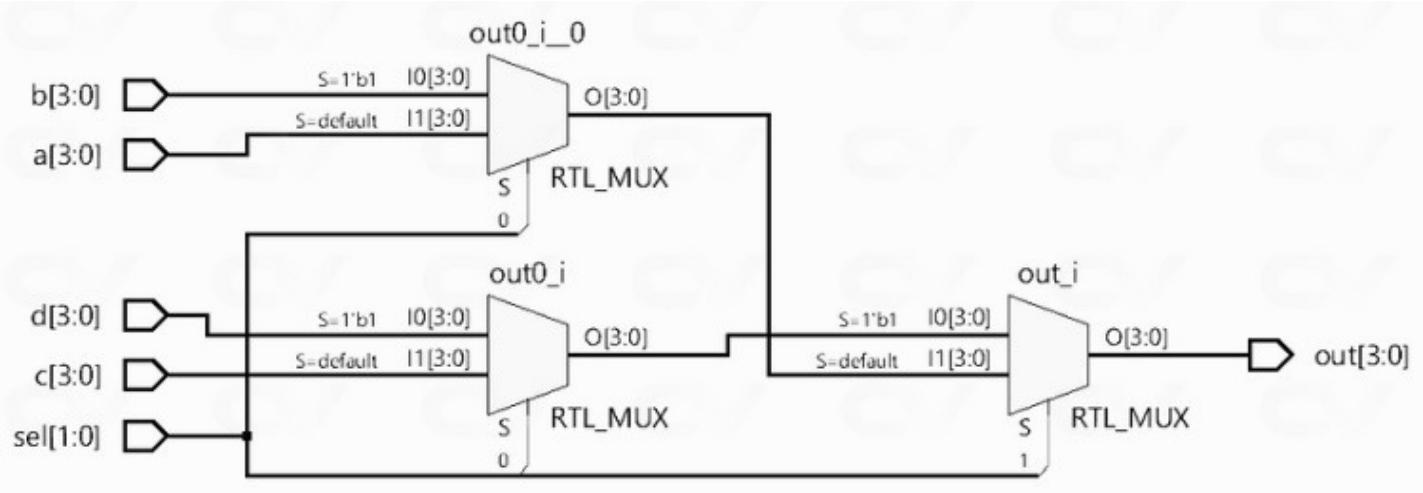
Note that the signal out is declared as a reg type because it is used in a *procedural* block like always.

```
module mux_4to1_case ( input [3:0] a, // 4-bit input called a
input [3:0] b, // 4-bit input called b
input [3:0] c, // 4-bit input called c
input [3:0] d, // 4-bit input called d
input [1:0] sel, // input sel used to select
between a,b,c,d
                                                output reg [3:0] out); // 4-bit output based on input
sel

// This always block gets executed whenever a/b/c/d/sel changes value
                                    // When that happens, based on value in sel, output is assigned to either a/b/c/d
always @ (a or b or c or d or sel) begin
case (sel)
2'b00 : out <= a;
2'b01 : out <= b;
2'b10 : out <= c;
2'b11 : out <= d;
endcase
end
endmodule
```

The module called *mux_4x1_case* has four 4-bit data inputs, one 2-bit select input and one 4-bit data output. The multiplexer will select either a , b, c, or d based on the select signal sel using the case statement.

Hardware Schematic

Both types of multiplexer models get synthesized into the same hardware as shown in the image below.

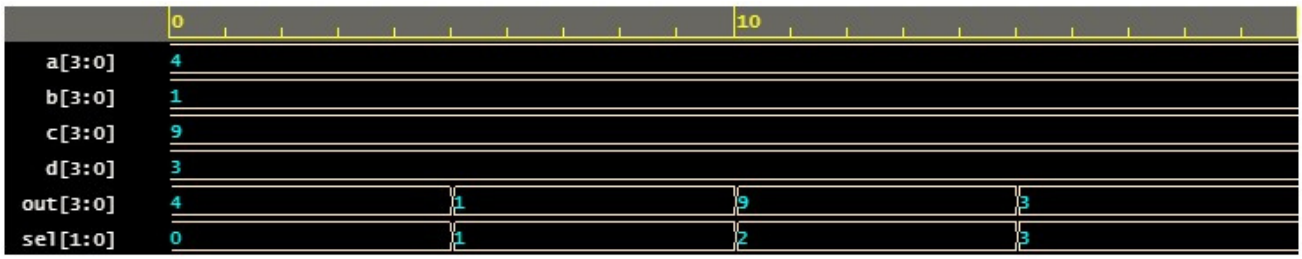


Testbench

```
module tb_4to1_mux;

// Declare internal reg variables to drive design inputs
// Declare wire signals to collect design output
// Declare other internal variables used in testbench
reg [3:0] a;
reg [3:0] b;
reg [3:0] c;
reg [3:0] d;
wire [3:0] out;
reg [1:0] sel;
integer i;

                    // Instantiate one of the designs, in this case, we have used the design with
case statement
// Connect testbench variables declared above with those in the design
mux_4to1_case mux0 ( .a (a),
.b (b),
.c (c),
.d (d),
.sel (sel),
.out (out));

// This initial block is the stimulus
initial begin
// Launch a monitor in background to display values to log whenever
a/b/c/d/sel/out changes
                        $monitor ("[%0t] sel=0x%0h a=0x%0h b=0x%0h c=0x%0h d=0x%0h out=0x%0h", $time,
sel, a, b, c, d, out);

// 1. At time 0, drive random values to a/b/c/d and keep sel = 0
sel <= 0;
a <= $random;
b <= $random;
c <= $random;
d <= $random;

// 2. Change the value of sel after every 5ns
for (i = 1; i < 4; i=i+1) begin
#5 sel <= i;
end

// 3. After Step2 is over, wait for 5ns and finish simulation
#5 $finish;
end
endmodule
```

Simulation Log

```
ncsim> run
[0] sel=0x0 a=0x4 b=0x1 c=0x9 d=0x3 out=0x4
[5] sel=0x1 a=0x4 b=0x1 c=0x9 d=0x3 out=0x1
[10] sel=0x2 a=0x4 b=0x1 c=0x9 d=0x3 out=0x9
[15] sel=0x3 a=0x4 b=0x1 c=0x9 d=0x3 out=0x3
Simulation complete via $finish(1) at time 20 NS + 0
```



Verilog Full Adder

An adder is a digital component that performs addition of two numbers. Its the main component inside an ALU of a processor and is used to increment addresses, table indices, buffer pointers and in a lot of other places where addition is required.

A full adder adds a carry input along with other input binary numbers to produce a sum and a carry output.

Truth Table

A	B	Cin	Cout	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Design

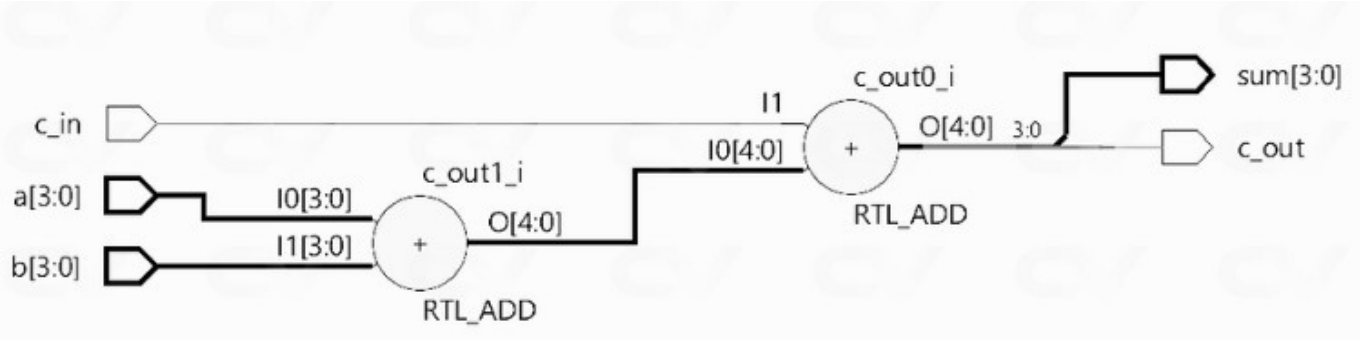
An example of a 4-bit adder is shown below which accepts two binary numbers through the signals a and b which are both 4-bits wide. Since an adder is a combinational circuit, it can be modeled in Verilog using a continuous assignment with assign or an always block with a sensitivity list that comprises of all inputs. The code shown below is that of the former approach.

```
module fulladd ( input [3:0] a,
input [3:0] b,
input c_in,
output c_out,
output [3:0] sum);

                    assign {c_out, sum} = a + b + c_in;
endmodule
```

The code shown below uses an always block which gets executed whenever any of its inputs change value.

```
module fulladd ( input [3:0] a,
input [3:0] b,
input c_in,
output reg c_out,
                                    output reg [3:0] sum);

always @ (a or b or c_in) begin
{c_out, sum} = a + b + c_in;
 end
endmodule
```

Hardware Schematic



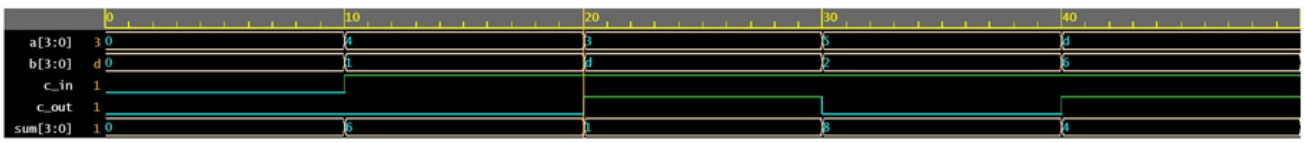
Testbench

```
module tb_fulladd;
// 1. Declare testbench variables
reg [3:0] a;
reg [3:0] b;
reg c_in;
wire [3:0] sum;
integer i;

                                    // 2. Instantiate the design and connect to testbench variables
fulladd fa0 ( .a (a),
.b (b),
.c_in (c_in),
.c_out (c_out),
.sum (sum));

// 3. Provide stimulus to test the design
initial begin
a <= 0;
b <= 0;
c_in <= 0;

                                        $monitor ("a=0x%0h b=0x%0h c_in=0x%0h c_out=0x%0h sum=0x%0h", a, b, c_in,
c_out, sum);

// Use a for loop to apply random values to the input
for (i = 0; i < 5; i = i+1) begin
#10 a <= $random;
b <= $random;
c_in <= $random;
end
end
endmodule
```

Note that when a and b add up to give a number more than 4 bits wide, the sum rolls over to zero and c_out becomes 1. For example, the line highlighted in yellow adds up to give 0x11 and the lower 4 bits get assigned to sum and bit#4 to c_out .

Simulation Log

```
ncsim> run
a=0x0 b=0x0 c_in=0x0 c_out=0x0 sum=0x0
a=0x4 b=0x1 c_in=0x1 c_out=0x0 sum=0x6
a=0x3 b=0xd c_in=0x1 c_out=0x1 sum=0x1
a=0x5 b=0x2 c_in=0x1 c_out=0x0 sum=0x8
a=0xd b=0x6 c_in=0x1 c_out=0x1 sum=0x4
a=0xd b=0xc c_in=0x1 c_out=0x1 sum=0xa
ncsim: *W,RNQUIE: Simulation is complete.
```



Verilog Single Port RAM

What is a single port RAM ?

A single-port RAM (Random Access Memory) is a type of digital memory component that allows data to be read from and written to a single memory location (address) at a time. It is a simple form of memory that provides a basic storage mechanism for digital systems. Each memory location in a single-port RAM can store a fixed number of bits (usually a power of 2, such as 8, 16, 32, etc.).

During a read operation, the data stored at a specific address is retrieved. During a write operation, new data is stored at a specific address, replacing the previous data.

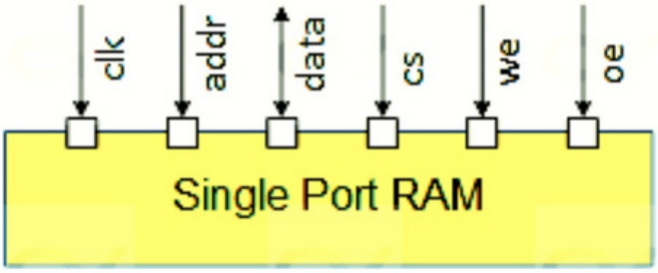
Why is it called single port ?

A single-port RAM has only one data port, which means that read and write operations cannot occur simultaneously at different addresses. If a write operation is in progress, a read operation must wait, and vice versa.

Signals

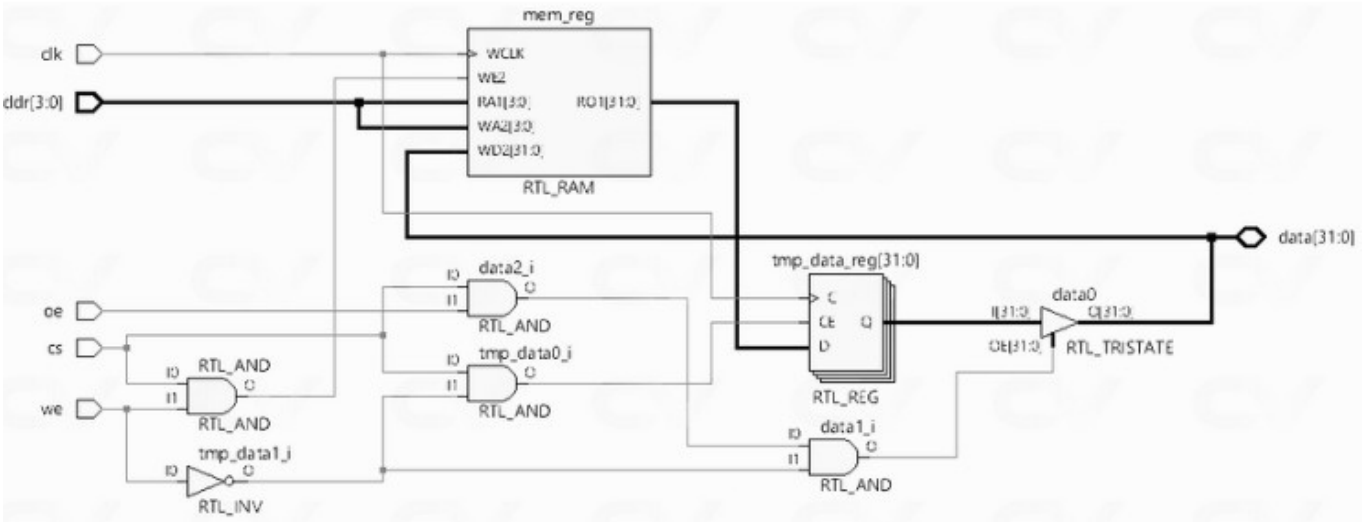
Single-port RAMs have *address lines* that are used to select the memory location to be accessed. The number of address lines determines the maximum number of memory locations that the RAM can hold.

Data lines are used to carry the actual data to be read from or written to the memory location.
Control signals, such as read enable (read request) and write enable (write request), are used to initiate specific memory operations.



Verilog Design

```
module single_port_sync_ram
# (parameter ADDR_WIDTH = 4,
parameter DATA_WIDTH = 32,
parameter DEPTH = 16
)
                                                    ( input clk,
input [ADDR_WIDTH-1:0] addr,
inout [DATA_WIDTH-1:0] data,
input                  cs,
input                  we,
input                  oe
);

reg [DATA_WIDTH-1:0] tmp_data;
reg [DATA_WIDTH-1:0] mem [DEPTH];

always @ (posedge clk) begin
if (cs & we)
mem[addr] <= data;
end

always @ (posedge clk) begin
if (cs & !we)
tmp_data <= mem[addr];
end

assign data = cs & oe & !wr ? tmp_data : 'hz;
endmodule
```



Verilog Testbench

```
module tb;
parameter ADDR_WIDTH = 4;
parameter DATA_WIDTH = 16;
parameter DEPTH = 16;

reg clk;
reg cs;
reg we;
reg oe;
reg [ADDR_WIDTH-1:0] addr;
wire [DATA_WIDTH-1:0] data;
reg [DATA_WIDTH-1:0] tb_data;

        single_port_sync_ram #(.DATA_WIDTH(DATA_WIDTH)) u0
( .clk(clk),
.addr(addr),
.data(data),
.cs(cs),
.we(we),
.oe(oe)
                                        );


 always #10 clk = ~clk;
assign data = !oe ? tb_data : 'hz;

 initial begin
{clk, cs, we, addr, tb_data, oe} <= 0;

repeat (2) @ (posedge clk);

for (integer i = 0; i < 2**ADDR_WIDTH; i= i+1) begin
                                repeat (1) @(posedge clk) addr <= i; we <= 1; cs <=1; oe <= 0; tb_data <=
$random;
end

for (integer i = 0; i < 2**ADDR_WIDTH; i= i+1) begin
repeat (1) @(posedge clk) addr <= i; we <= 0; cs <= 1; oe <= 1;
end

#20 $finish;
end
endmodule
```

Single-port RAMs are commonly used in various digital systems for tasks such as data storage, temporary buffering, and data manipulation.

Verilog Sequence Detector

A very common example of an FSM is that of a sequence detector where the hardware design is expected to detect when a fixed pattern is seen in a stream of binary bits that are input to it.

Example

```
module det_1011 ( input clk,
input rstn,
input in,
output out );

parameter IDLE = 0,
                                                    S1 = 1,
S10 = 2,
S101 = 3,
S1011 = 4;

reg [2:0] cur_state, next_state;

        assign out = cur_state == S1011 ? 1 : 0;

always @ (posedge clk) begin
if (!rstn)
cur_state <= IDLE;
else
cur_state <= next_state;
end

always @ (cur_state or in) begin
case (cur_state)
IDLE : begin
if (in) next_state = S1;
else next_state = IDLE;
end

S1: begin
if (in) next_state = IDLE;
else next_state = S10;
 end

S10 : begin
if (in) next_state = S101;
else next_state = IDLE;
end

S101 : begin
if (in) next_state = S1011;
else next_state = IDLE;
end

S1011: begin
next_state = IDLE;
end
endcase
end
endmodule
```

Testbench

```
module tb;
reg clk, in, rstn;
wire out;
reg [1:0] l_dly;
reg tb_in;
integer loop = 1;

always #10 clk = ~clk;

                det_1011 u0 ( .clk(clk), .rstn(rstn), .in(in), .out(out) );

initial begin
clk <= 0;
rstn <= 0;
in <= 0;

repeat (5) @ (posedge clk);
rstn <= 1;

// Generate a directed pattern
@(posedge clk) in <= 1;
@(posedge clk) in <= 0;
@(posedge clk) in <= 1;
                                @(posedge clk) in <= 1; // Pattern is completed
@(posedge clk) in <= 0;
@(posedge clk) in <= 0;
@(posedge clk) in <= 1;
@(posedge clk) in <= 1;
@(posedge clk) in <= 0;
@(posedge clk) in <= 1;
                @(posedge clk) in <= 1; // Pattern completed again

                        // Or random stimulus using a for loop that drives a random
// value of input N times
for (int i = 0 ; i < loop; i ++) begin
l_dly = $random;
repeat (l_dly) @ (posedge clk);
tb_in = $random;
in <= tb_in;
end

// Wait for sometime before quitting simulation
#100 $finish;
end
endmodule
```

Simulation Log

```
ncsim> run
T=10 in=0 out=0
T=30 in=0 out=0
T=50 in=0 out=0
T=70 in=0 out=0
T=90 in=0 out=0
T=110 in=1
out=0
T=130 in=0
out=0
T=150 in=1
out=0
T=170 in=1
out=0
T=190 in=0 out=1
T=210 in=0 out=0
T=230 in=1 out=0
T=250 in=1
out=0
T=270 in=0
out=0
T=290 in=1
out=0
T=310 in=1
out=0
T=330 in=1 out=1
T=350 in=1 out=0
T=370 in=1 out=0
T=390 in=1 out=0
Simulation complete via $finish(1) at time 410 NS + 0
```

There is a bug in the design. Can you find it ?

Verilog Pattern Detector

A previous example explored a simple sequence detector. Here is another example for a pattern detector which detects a slightly longer pattern.

Design

```
module det_110101 ( input clk,
input rstn,
input in,
output out );

parameter IDLE = 0,
                                                S1 = 1,
S11 = 2,
S110 = 3,
S1101 = 4,
S11010 = 5,
S110101 = 6;

reg [2:0] cur_state, next_state;

            assign out = cur_state == S110101 ? 1 : 0;

always @ (posedge clk) begin
if (!rstn)
cur_state <= IDLE;
else
cur_state <= next_state;
end

always @ (cur_state or in) begin
case (cur_state)
IDLE : begin
if (in) next_state = S1;
else next_state = IDLE;
end

S1: begin
if (in) next_state = S11;
else next_state = IDLE;
end

S11: begin
if (!in) next_state = S110;
else next_state = S11;
 end

S110 : begin
if (in) next_state = S1101;
else next_state = IDLE;
end

S1101 : begin
if (!in) next_state = S11010;
else next_state = IDLE;
end

S11010: begin
```

```
                    if (in) next_state = S110101;
else next_state = IDLE;
end

S110101: begin
if (in) next_state = S1;
else next_state = IDLE;                                    // Bug 2
end
endcase
end
endmodule
```

Testbench

```
module tb;
reg clk, in, rstn;
wire out;
integer l_dly;

always #10 clk = ~clk;

                        det_110101 u0 ( .clk(clk), .rstn(rstn), .in(in), .out(out) );

initial begin
clk <= 0;
rstn <= 0;
in <= 0;

repeat (5) @ (posedge clk);
rstn <= 1;

@(posedge clk) in <= 1;
@(posedge clk) in <= 1;
@(posedge clk) in <= 0;
@(posedge clk) in <= 1;
@(posedge clk) in <= 0;
@(posedge clk) in <= 1;
@(posedge clk) in <= 1;
@(posedge clk) in <= 1;
@(posedge clk) in <= 0;
@(posedge clk) in <= 1;
@(posedge clk) in <= 0;
@(posedge clk) in <= 1;

#100 $finish;
end
endmodule
```

Simulation Log

```
ncsim> run
T=10 in=0 out=0
T=30 in=0 out=0
T=50 in=0 out=0
T=70 in=0 out=0
T=90 in=0 out=0
T=110 in=1
out=0
T=130 in=1
out=0
T=150 in=0
out=0
T=170 in=1
out=0
T=190 in=0
out=0
T=210 in=1
out=0
T=230 in=1
out=1
T=250 in=1
out=0
T=270 in=0
out=0
T=290 in=1
out=0
T=310 in=0
out=0
T=330 in=1
out=0
T=350 in=1 out=1
T=370 in=1 out=0
T=390 in=1 out=0
T=410 in=1 out=0
Simulation complete via $finish(1) at time 430 NS + 0
```