Verilog

Harris & Harris, Digital Design and Computer Architecture

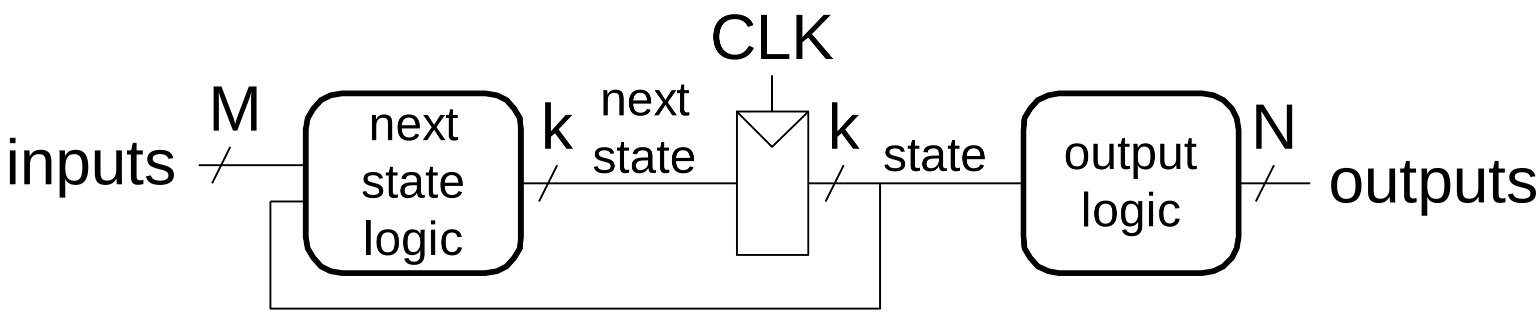
Ch. 4 - Hardware Description Languages

4.6 FINITE STATE MACHINES

4.9 TESTBENCHES

Finite State Machines (FSMs)

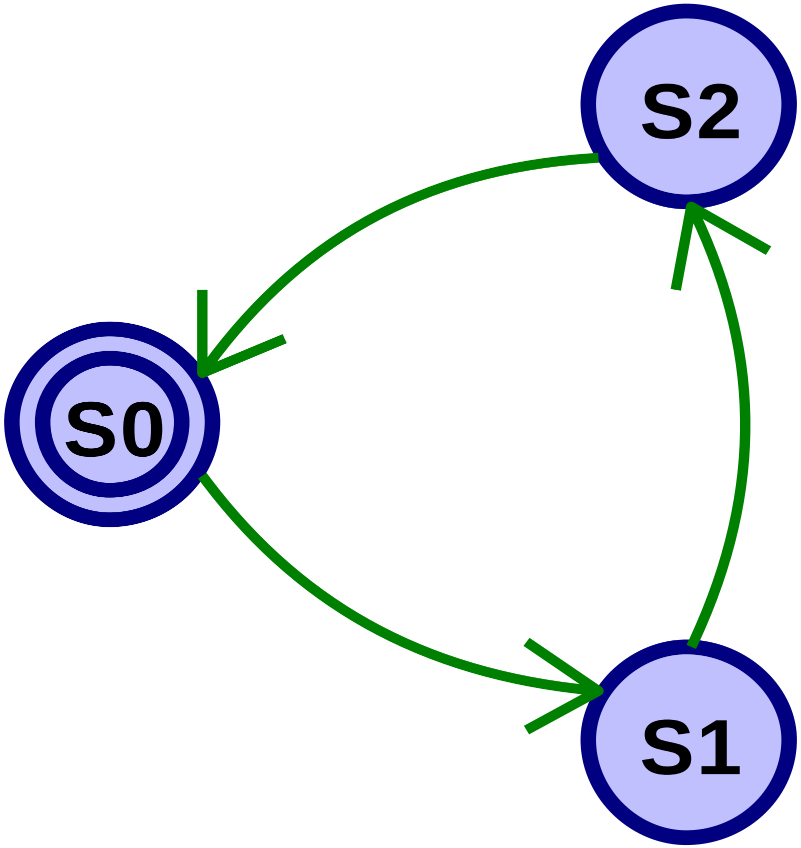
A finite state machine (FSM) consists of a **state register** and **two blocks of combinational logic** to compute the **next state** and the **output** given the current state and the input



HDL descriptions of state machines are correspondingly divided into **three parts** to model:

- the state register
- the next state logic
- the output logic

FSM Example: Divide by 3



- A **divide-by-N counter** has one output and no inputs
- The output Y is HIGH for one clock cycle out of every N → the output divides the frequency of the clock by N

The double circle indicates the reset state

FSM in SystemVerilog

```
module divideby3FSM (input  logic clk,
                     input  logic reset,
                     output logic q);

 typedef enum logic [1:0] {S0, S1, S2} statetype;
 statetype [1:0] state, nextstate;

   // state register
   always_ff @ (posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;

   // next state logic
   always_comb
      case (state)
         S0:      nextstate = S1;
         S1:      nextstate = S2;
         S2:      nextstate = S0;
         default: nextstate = S0;
      endcase

   // output logic
   assign q = (state == S0);
endmodule
```

- The `typedef` statement defines `statetype` to be a two-bit `logic` value with three possibilities: `S0`, `S1`, or `S2`
- `state` and `nextstate` are `statetype` signals
- The enumerated **encodings** default to numerical order: `S0` = 00, `S1` = 01, and `S2` = 10
- Notice that the **states** are named with an (easier, more readable) **enumeration data type instead of binary values**
- The **encodings** can be explicitly set by the user

FSM in SystemVerilog

```
module divideby3FSM (input  logic clk,
                     input  logic reset,
                     output logic q);

 typedef enum logic [1:0] {S0, S1, S2} statetype;
 statetype [1:0] state, nextstate;

   // state register
   always_ff @(posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;

   // next state logic
   always_comb
      case (state)
         S0:      nextstate = S1;
         S1:      nextstate = S2;
         S2:      nextstate = S0;
         default: nextstate = S0;
      endcase

   // output logic
   assign q = (state == S0);
endmodule
```

- The **output**, y, is 1 when the **state is S0**
- The **equality comparison** **a == b** evaluates to 1 if **a** equals **b** and 0 otherwise

- The code provides an asynchronous reset to initialize the FSM
- The **state register** uses the ordinary idiom for flip-flops
- The **next state** and **output** logic blocks are combinational

FSM in SystemVerilog

```
module divideby3FSM (input  logic clk,
                     input  logic reset,
                     output logic q);

 typedef enum logic [1:0] {S0, S1, S2} statetype;
 statetype [1:0] state, nextstate;

   // state register
   always_ff @ (posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;

   // next state logic
   always_comb
      case (state)
         S0:      nextstate = S1;
         S1:      nextstate = S2;
         S2:      nextstate = S0;
         default: nextstate = S0;
      endcase

   // output logic
   assign q = (state == S0);
endmodule
```

- If we had wanted the output to be HIGH in states S0 and S1, the output logic would be modified as follows:

```
// output logic
assign y = (state== S0 |
state== S1);
```

Testbenches

- A **testbench** is an HDL module that is used to test another module, called the **device under test** (DUT)
- The testbench contains statements:
  - to apply inputs to the DUT and
  - to check that the correct outputs are produced
- The input and desired output patterns are called ***test vectors***

- Types of testbench:
  - Simple
  - Self-checking
  - Self-checking with testvectors

Testbench Example

- Consider the SystemVerilog code to implement the following function in hardware:

$$y = \overline{b}\,\overline{c} + a\overline{b}$$

```
module sillyfunction(input  logic a, b, c,
                     output logic y);
  assign y = ~b & ~c | a & ~b;
endmodule
```

Simple Testbench

```
module testbench1();
  logic a, b, c;
  logic y;
  // instantiate device under test
  sillyfunction dut(a, b, c, y);
  // apply inputs one at a time
  initial begin
    a = 0; b = 0; c = 0; #10;
    c = 1;               #10;
    b = 1; c = 0;        #10;
    c = 1;               #10;
    a = 1; b = 0; c = 0; #10;
    c = 1;               #10;
    b = 1; c = 0;        #10;
    c = 1;               #10;
  end
endmodule
```

simple testbench
It instantiates the DUT, then applies the inputs
Blocking assignments and delays are used to apply the inputs in the appropriate order

The `initial` statement executes the statements in its body at the start of simulation and is used in testbenches

The user must view the results and **verify the correct outputs** are produced → tedious and error prone

Self-checking Testbench

```
module testbench2();
  logic  a, b, c;
  logic y;
  sillyfunction dut(a, b, c, y);  // instantiate dut
  initial begin // apply inputs, check results one
   at a time
    a = 0; b = 0; c = 0;      #10;
    if (y !== 1) $display("000 failed.");
    c = 1;                    #10;
    if (y !== 0) $display("001 failed.");
    b = 1; c = 0;             #10;
    if (y !== 0) $display("010 failed.");
    c = 1;                    #10;
    if (y !== 0) $display("011 failed.");
    a = 1; b = 0; c = 0;      #10;
    if (y !== 1) $display("100 failed.");
    c = 1;                    #10;
    if (y !== 1) $display("101 failed.");
    b = 1; c = 0;             #10;
    if (y !== 0) $display("110 failed.");
    c = 1;                    #10;
    if (y !== 0) $display("111 failed.");
  end
endmodule
```

self-checking testbench

Also in this case, you have to write code for each test vector, that becomes especially tedious for modules with a large number of vectors

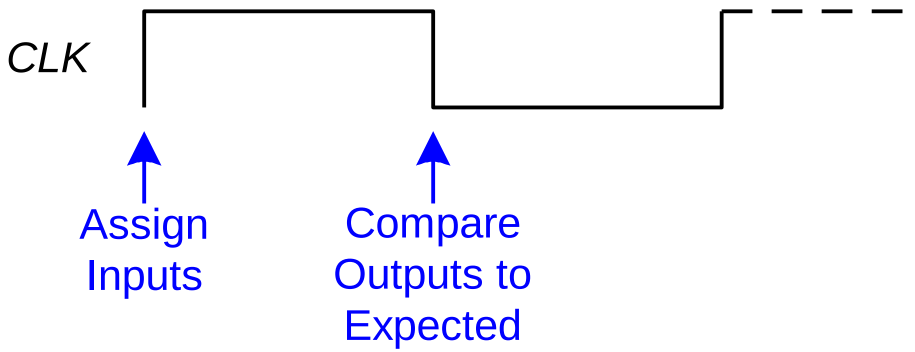
Testbench with Testvectors

- A better approach is to place the **test vectors in a separate file**: inputs and expected outputs

- Testbench:
  1. Generate clock for assigning inputs, reading outputs
  2. Read testvectors file into array
  3. Assign inputs, expected outputs
  4. Compare outputs with expected outputs and report errors

Testbench with Testvectors

- Testbench clock:
  - assign inputs (on rising edge)
  - compare outputs with expected outputs (on falling edge)



- Testbench clock also used as clock for synchronous sequential circuits

Testvectors File

- File: `example.tv`
- Contains vectors of the inputs and expected output written in binary in the form `abc_y`

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

1. Generate Clock

```
module testbench3();
  logic        clk, reset;
  logic        a, b, c, yexpected;
  logic        y;
  logic [31:0] vectornum, errors;    // bookkeeping variables
  logic [3:0]  testvectors[10000:0]; // array of testvectors

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // generate clock
  always     // no sensitivity list, so it always executes
    begin
      clk = 1; #5; clk = 0; #5;
    end
```

2. Read Testvectors into Array

```
// at start of test, load vectors and pulse reset

 initial
    begin
      $readmemb("example.tv", testvectors);
      vectornum = 0; errors = 0;
      reset = 1; #27; reset = 0;
    end


// Note: $readmemh reads testvector files written in
// hexadecimal
```

3. Assign Inputs & Expected Outputs

```
// apply test vectors on rising edge of clk

always @(posedge clk)
  begin
    #1; {a, b, c, yexpected} = testvectors[vectornum];
  end
```

4. Compare with Expected Outputs

```
// check results on falling edge of clk

   always @(negedge clk)
     if (~reset) begin       // skip during reset
       if (y !== yexpected) begin
         $display("Error: inputs = %b", {a, b, c});
         $display("  outputs = %b (%b expected)",y,yexpected);
         errors = errors + 1;
       end

// Note: to print in hexadecimal, use %h. For example,
//       $display(“Error: inputs = %h”, {a, b, c});
```

4. Compare with Expected Outputs

```
// increment array index and read next testvector

      vectornum = vectornum + 1;
       if (testvectors[vectornum] === 4'bx) begin
           $display("%d tests completed with %d errors",
                 vectornum, errors);
           $finish;
       end
     end
endmodule


// === and !== can compare values that are 1, 0, x, or z
```

ROM

```
module rom(input logic [1:0] adr,
           output logic [2:0] dout):

always_comb
   case(adr)
        2'b00: dout = 3'b011;
        2'b01: dout = 3'b110;
        2'b10: dout = 3'b100;
        2'b11: dout = 3'b010;
   endcase
endmodule
```

ADDER

```
module adder #(parameter N = 8)
              (input logic [N-1:0] a, b,
               input logic cin,
               output logic [N-1:0] s,
               output logic cout);

  assign {cout, s} = a + b + cin;
endmodule
```

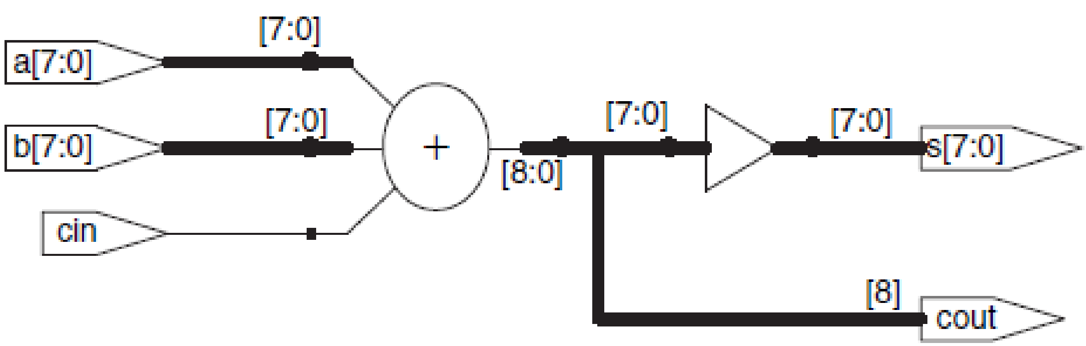
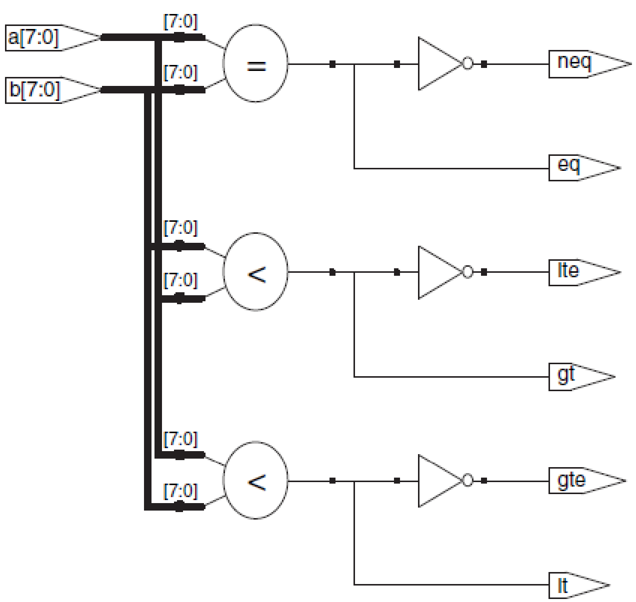


Figure 5.8 Synthesized adder

COMPARATORS

```
module comparator #(parameter N = 8)
                   (input logic [N-1:0] a, b,
                    output logic eq, neq, lt, lte, gt, gte);

   assign eq = (a == b);
   assign neq = (a != b);
   assign lt = (a < b);
   assign lte = (a <= b);
   assign gt = (a > b);
   assign gte = (a >= b);
endmodule
```



SHIFT REGISTER WITH PARALLEL LOAD

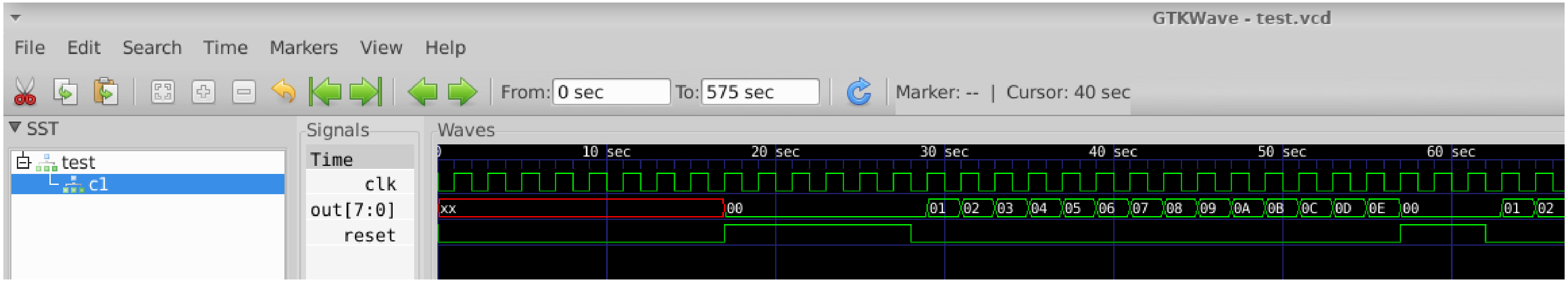
```
module shiftreg #(parameter N = 8)
              (input logic clk,
               input logic reset, load,
               input logic sin,
               input logic [N-1:0] d,
               output logic [N-1:0] q,
               output logic sout);

  always_ff @(posedge clk, posedge reset)
      if (reset) q <= 0;
      else if (load) q <= d;
      else q <= {q[N-2:0], sin};

  assign sout = q[N-1];
endmodule
```

ESEMPI

Esempio contatore: simulazione

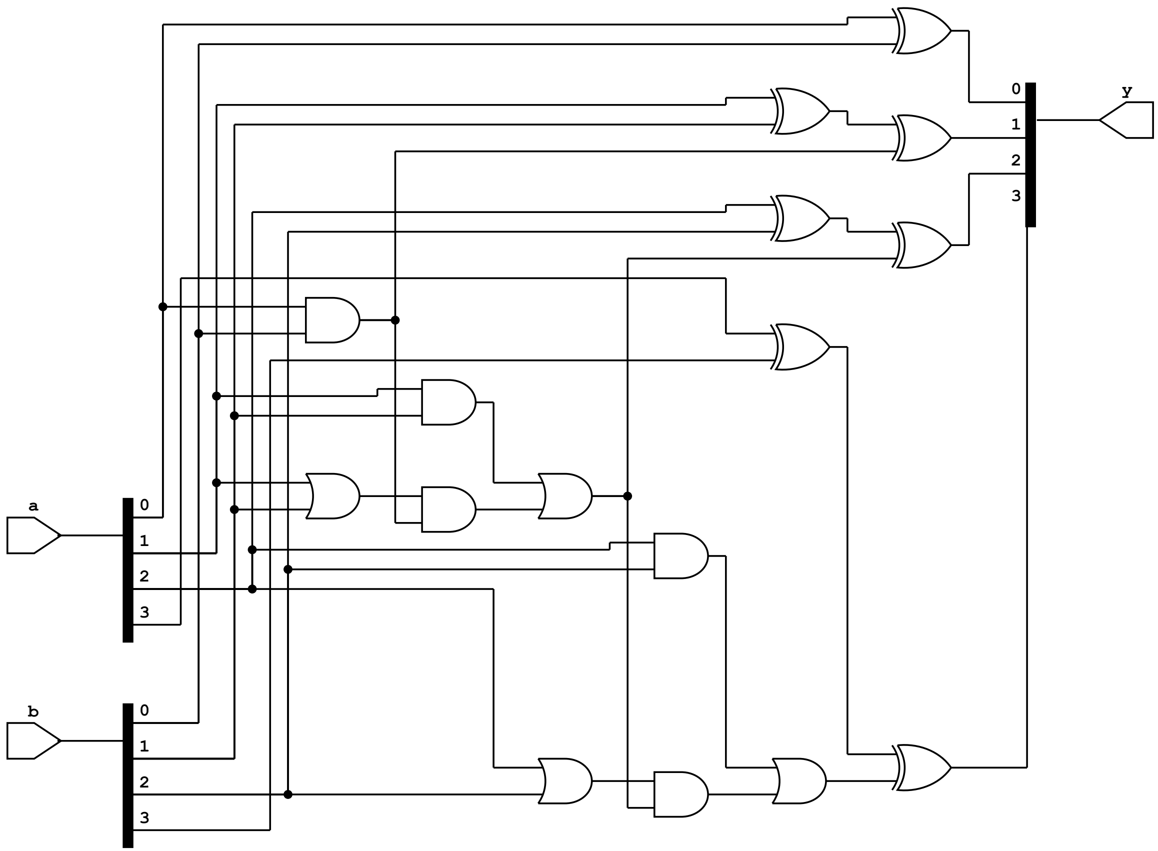
- GTKWave is a fully featured GTK based wave viewer (GTK is a cross-platform widget toolkit for creating graphical user interfaces (GUIs))
- Waveform dumps are written by the Icarus Verilog runtime program **vvp**
- The **vvp command** is the simulation run-time engine

```
$ iverilog -o tb counter_tb.v counter.v
$ vvp tb
$ gtkwave test.vcd
```



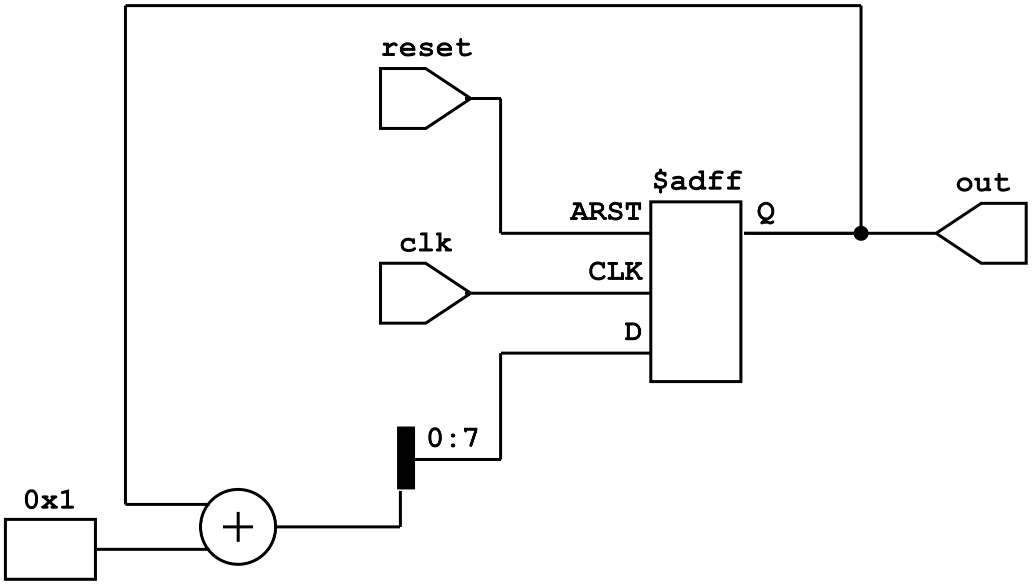
Esempio sommatore: sintesi

```
$ yosys synth.ys
$ netlistsvg carryadd.js -o carryadd.svg
$ ristretto carryadd.svg
```



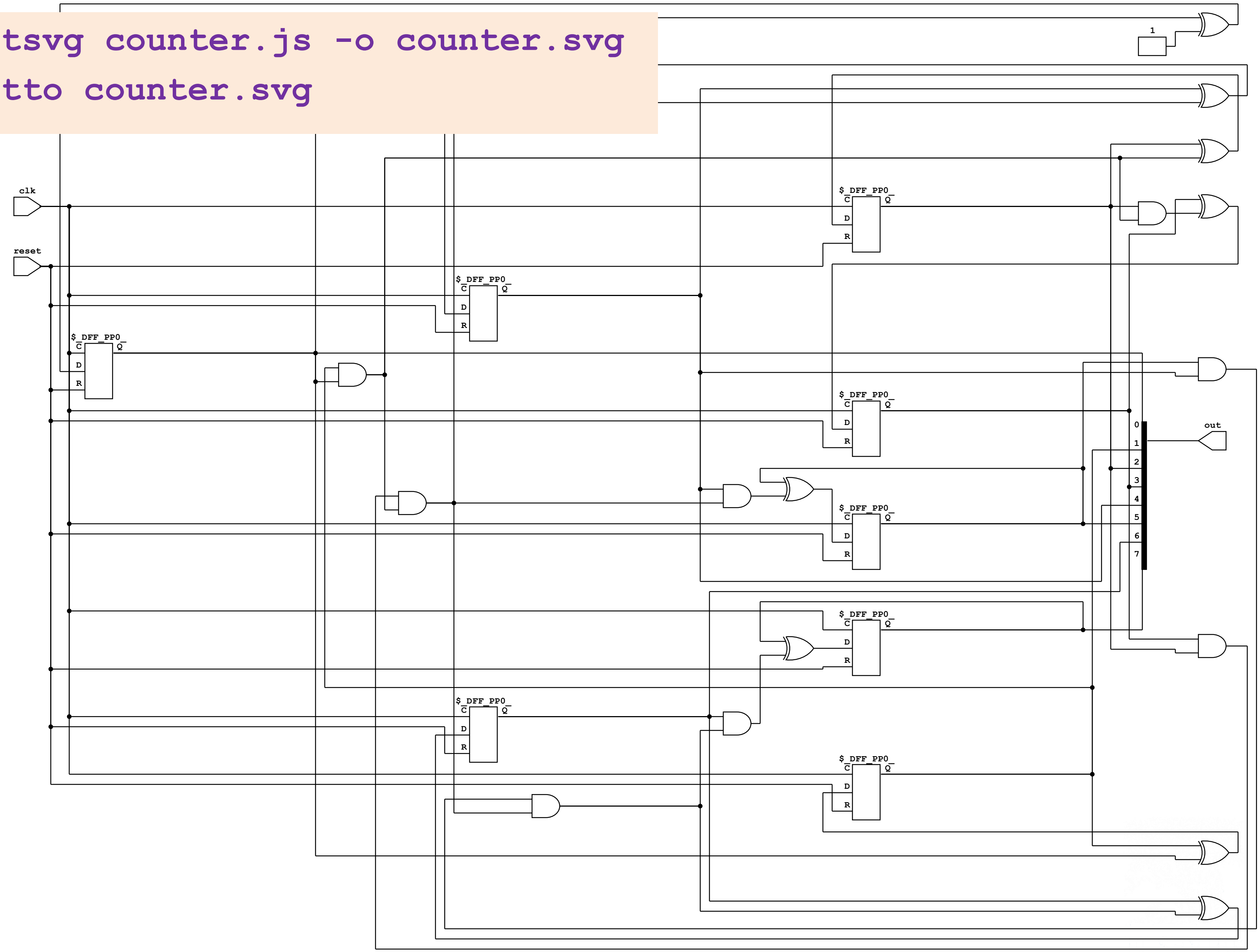
Esempio contatore: sintesi (RTL)

```
$ yosys synth_count.ys
$ netlistsvg counter_rtl.js -o counter_rtl.svg
$ ristretto counter_rtl.svg
```



Esempio contatore: sintesi (gates)

```
$ netlistsvg counter.js -o counter.svg
$ ristretto counter.svg
```



Esercizio

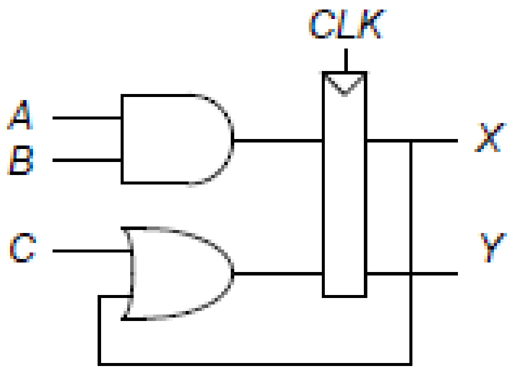
Exercise 4.26 Write an HDL module for an SR latch.

```
module srlatch(input logic s, r,
               output logic q, qbar);

always @(*)
      case ({s,r})
            2'b01: {q, qbar} = 2'b01;
            2'b10: {q, qbar} = 2'b10;
            2'b11: {q, qbar} = 2'b00;
      endcase
endmodule
```

Esercizio

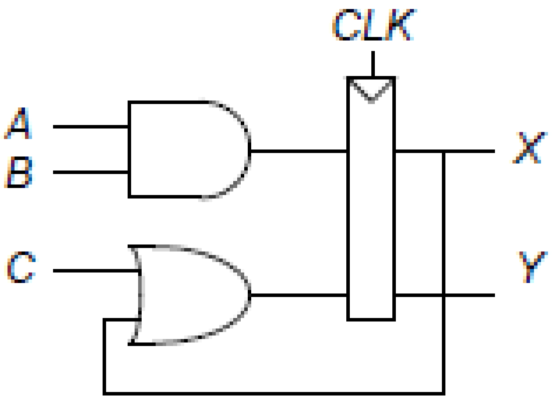
(Exercise 4.48)

Dato il seguente circuito scrivere il modulo Verilog che lo produce



Esercizio

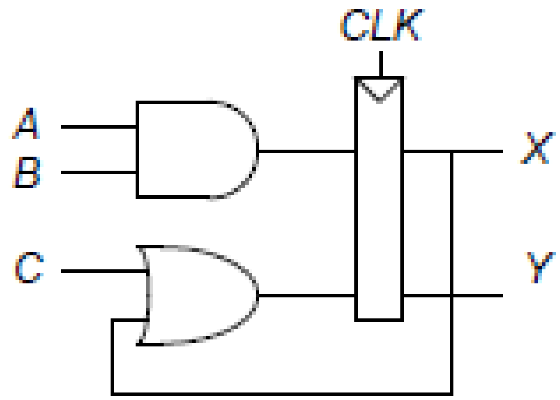
```
module code1(input  logic clk, a, b, c,
               output logic y);
  logic x;

  always_ff @(posedge clk) begin
    x <= a & b;
    y <= x | c;
  end
endmodule
```



Esercizio

```
module code1(input  logic clk, a, b, c,
                output logic y);
  logic x;

  always_ff @(posedge clk) begin
    x <= a & b;
    y <= x | c;
  end
endmodule
```

```
module code2 (input  logic a, b, c, clk,
                output logic y);
  logic x;

  always_ff @(posedge clk) begin
    y <= x | c;
    x <= a & b;
  end
endmodule
```



Esempio contatore: simulazione

```
module counter(
        output logic [WIDTH-1 : 0] out,
        input logic clk, reset);

parameter WIDTH = 8;

always @(posedge clk, posedge reset)

        if (reset)
                out <= 0;
        else
                out <= out + 1;

endmodule // counter
```

```
module test;
/* Make a reset that pulses once. */
reg reset = 0;
initial begin
        $dumpfile("test.vcd");
        $dumpvars(0,test);
        # 17 reset = 1;
        # 11 reset = 0;
        # 29 reset = 1;
        # 5  reset =0;
        # 513 $finish;
end
/* Make a regular pulsing * clock. */
     reg clk = 0;
     always #1 clk = !clk;
     wire [7:0] value;
     counter c1 (value, clk, reset);
/*   initial
     $monitor("At time %t, value = %h
     (%0d)",$time, value, value); */
endmodule // test
```