Go to **wooclap.com** and use the code **EYETQC**
Any questions? Send them in now by

1 Go to wooclap.com
2 Enter the event code in the top banner

Event code
EYETQC

Writing complete Verilog code for a RISC-V CPU, including all the modules and their interconnections, is a complex and extensive task that would go beyond the scope of a single response. Implementing a full RISC-V CPU in Verilog is typically a project that requires significant time and expertise. However, I can provide a simplified example of Verilog code for a basic RISC-V core to give you a starting point.

Please note that this is a very basic example and may not cover all features and optimizations of a real-world RISC-V CPU. You would need to consult detailed CPU designs and possibly a CPU microarchitecture textbook for a more comprehensive implementation. Here's a basic structure of a 3-stage pipelined RV32I RISC-V core in Verilog:

verilog Copy code

```verilog
module RISC_V_CPU (
    input wire clk,
    input wire reset,
    input wire [31:0] instruction,
    output wire [31:0] result
);

// Define the registers
reg [31:0] pc;
reg [31:0] if_id_instruction;
reg [31:0] id_ex_instruction;
reg [31:0] ex_mem_instruction;

// Fetch stage
always @(posedge clk or posedge reset) begin
    if (reset) begin
        pc <= 32'h0;
        if_id_instruction <= 32'h0;
    end else begin
        pc <= pc + 4;
        if_id_instruction <= instruction;
    end
end
```

Contact Information

Anupam Chattopadhyay

Email: anupam@ntu.edu.sg

Office: N4-02c-105

Plan for the 2nd half of the semester

NANYANG TECHNOLOGICAL UNIVERSITY

- Full-Time Course

Week	Pre-Recorded Lectures	Monday (LT19A) 830-920	Thursday (Zoom) 1630-1720	Tutorial	Lab
7	L13-L14				
Recess Week					
8	L15-L16	L13-L14 Summary	Online Consultation (Zoom)	Tutorial 6	+ quiz 3
9	L17-L18	L15-L16 Summary		Tutorial 7	Experiment 4 + quiz 4
10	L19-L20	L17-L18 Summary		Tutorial 8	
11	L21-L22	L19-L20 Summary (Zoom)		Tutorial 9	Experiment 5 + quiz 5
12	L23	L21-L22 Summary		Tutorial 10	
13		*Public holiday*			

Plan for the 2nd half of the semester *(contd.)*

NANYANG TECHNOLOGICAL UNIVERSITY

- Part-Time Course

Week	Pre-Recorded Lectures	Tuesday (LT11) 1830-2130		Lab
7	L13-L14			
Recess Week				
8	L15-L16	L13-L14 Summary	Tutorial 6	
9	L17-L18	L15-L16 Summary	Tutorial 7, 8	
10	L19-L20	L17-L19 Summary	Tutorial 9	
11	L21-L22			Experiment 4 + quiz 4
12	L23	L20-L22 Summary	Tutorial 10	
13				Experiment 5 + quiz 5

Plan for the 2nd half of the semester *(contd.)*

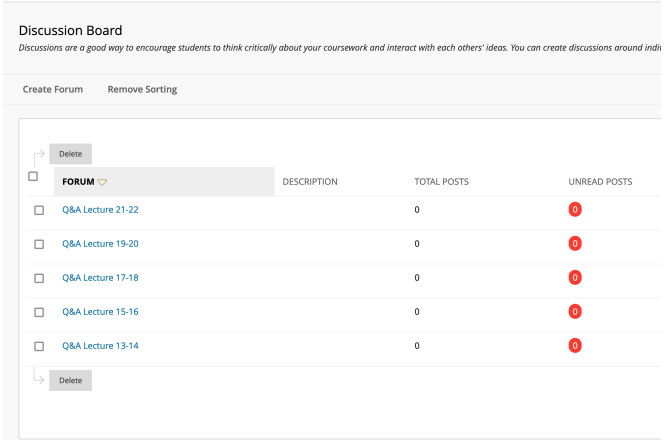
NANYANG TECHNOLOGICAL UNIVERSITY

- Online Tasks for L13 to L22
  - Will not be graded

- Discussion lectures (Monday, 8:30-9:20 AM, LT19A)
  - You are required to view the pre-recorded lectures
  - Recap and discussion (slides to be uploaded afterwards)
  - Additional examples and exercises
  - Polls through Wooclap (QR code in respective slides)

Plan for the 2nd half of the semester *(contd.)*

- Participation in Course
  - Use NTULearn Discussion Forum to ask follow-up questions



- Consultation Slots on (Thursday, 4:30-5:20 PM, Zoom)
  - Limit yourself to 3 questions
  - Avoid the clarification on tutorial
  - https://ntu-sg.zoom.us/j/81954891824
    Meeting ID: 819 5489 1824
    Passcode: 364003

NANYANG TECHNOLOGICAL UNIVERSITY

SC1005
Digital Logic

Recap and Discussion

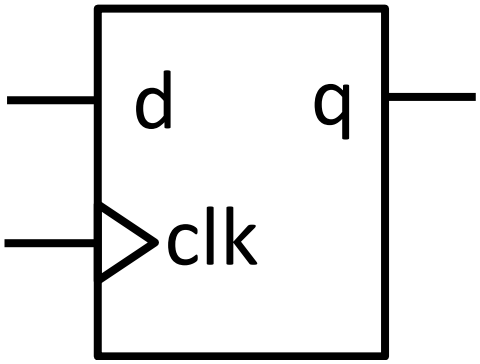
Lecture 19

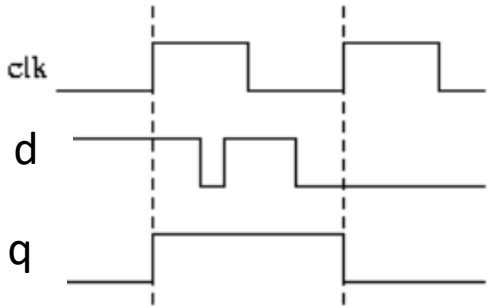
Sequential Circuits in Verilog

Summary of Lecture 19

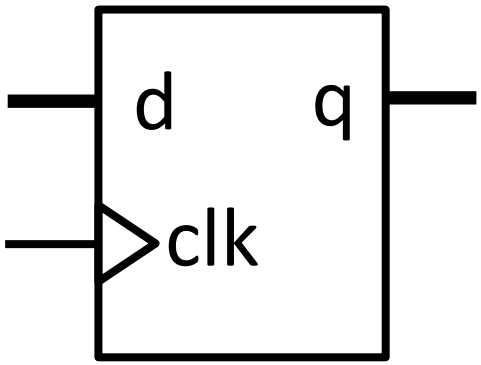
- Sequential Circuits in Verilog
  - Registers in Verilog
  - Clock and Reset
  - Synchronous always block
  - Counters

Recap: Registers in Verilog



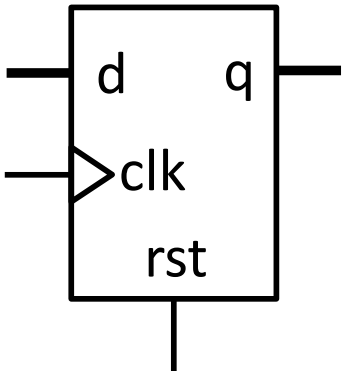


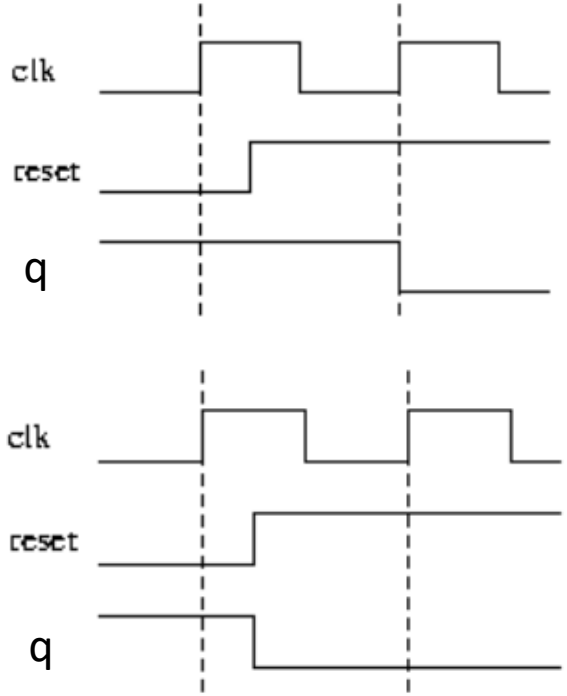

```
module simplereg (input d, clk,
                  output reg q);

always@(posedge clk)
    q <= d;

endmodule
```




```
module simplereg (input [7:0] d, input clk,
                  output reg [7:0] q);

always@(posedge clk)
    q <= d;

endmodule
```

Recap: Synchronous vs Asynchronous Reset



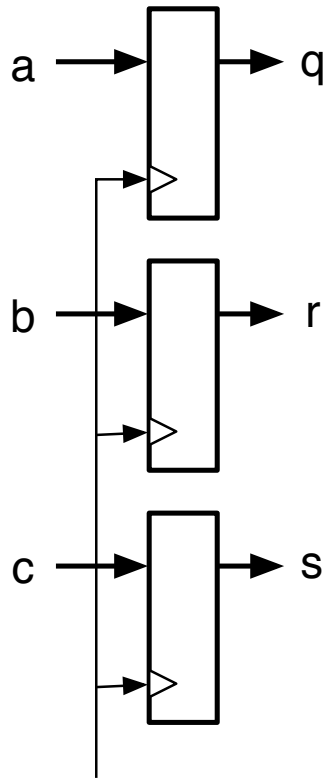



```
module simplereg (input [7:0] d,
                  input clk, rst,
                  output reg [7:0] q);

always@(posedge clk)
begin
    if(rst) // same as (rst==1'b1)
        q <= 8'b0000_0000;
    else
        q <= d;
end

endmodule
```

```
module simplereg (input [7:0] d,
                  input clk, rst,
                  output reg [7:0] q);

always@(posedge clk or posedge rst)
begin
    if(rst)
        q <= 8'b0000_0000;
    else
        q <= d;
end

endmodule
```

Registers in Verilog

- Each *assignment* in a synchronous always block results in a *register*



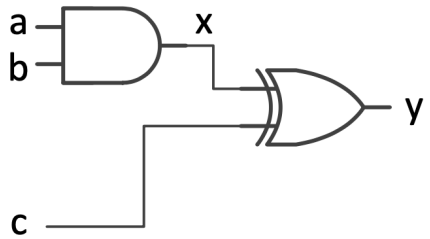

```
module multireg (input [7:0] a, b, c,
                 input clk, rst,
                 output reg [7:0] q, r, s);

always@(posedge clk)
begin
    if(!rst) begin
        q <= 8'b0000_0000;
        r <= 8'b0000_0000;
        s <= 8'b0000_0000;
    end else begin
        q <= a;
        r <= b;
        s <= c;
   end
end

endmodule
```

Recap: Assignments in Always Blocks

- For ***combinational*** always blocks, we *always* use a blocking assignment (=), and **order matters**
- For ***synchronous*** always blocks, we *always* use non-blocking assignments (<=), and **order does not matter**

Combinational always block

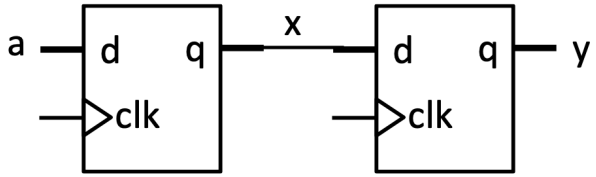
```
reg x, y;

always @ *
begin
   x = a & b;
   y = x ^ c;
end
```



Synchronous always block

```
reg x, y;

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

Each *assignment* in a synchronous always block results in a *register*

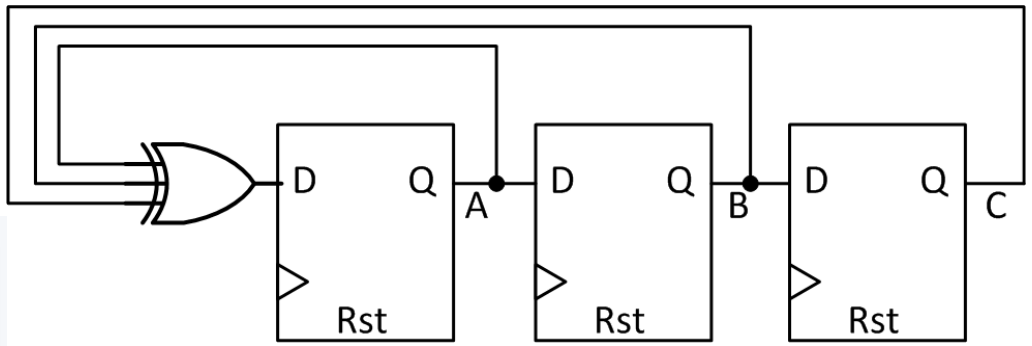


Exercise 1

For the Verilog code shown, if the current state of the circuit is *ABC* = 001, what is the next state?

```
module Ex1 (input  Clk, Rst,
            output reg A, B, C);

  always@(posedge Clk)
  begin
    if (Rst) begin
      C <= 1'b1;
      B <= 1'b0;
      A <= 1'b0;
    end
    else begin
      C <= B;
      B <= A;
      A <= A ^ B ^ C;
    end
  end
endmodule
```

111
001
100
010



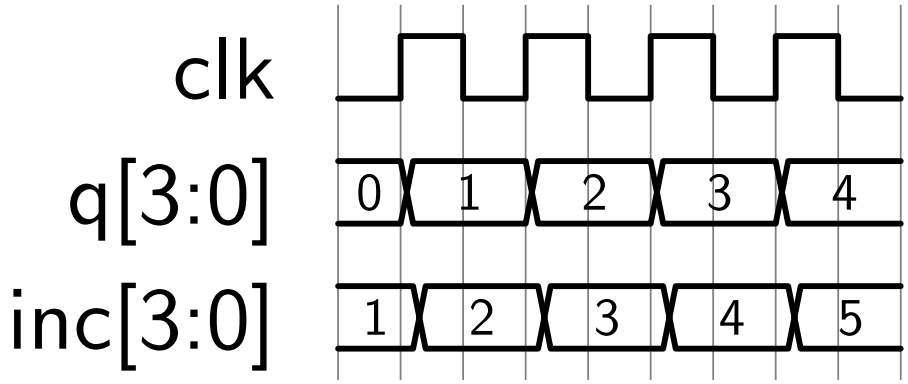
Sequential Verilog

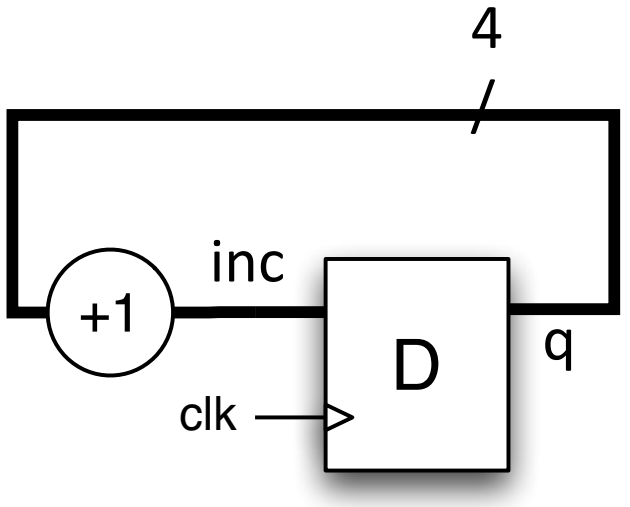
- There are a number of other basic sequential blocks we can use within our designs.
- Generally *edge-triggered components* are used.
  - Counters
  - Shift Registers
  - Serial-to-Parallel and Parallel-to-Serial Converters
  - Memories
  - FIFO buffer

- Synthesis tools will convert designs to D-type flip-flops/registers

Synchronous Counters

- Timing diagram helps:





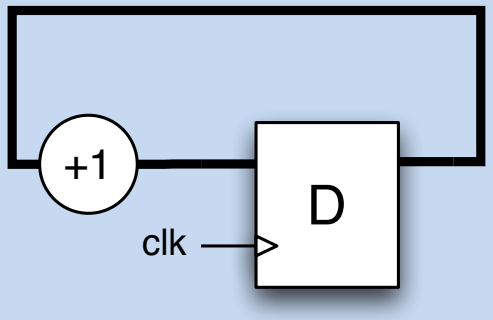
- At each clock edge, the incremented value, *inc*, derived from the current output, is passed to *q*

Recap: Binary Counters

- At each rising edge, we pass through the incremented value of the current output

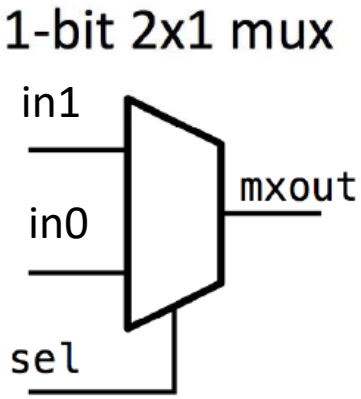
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

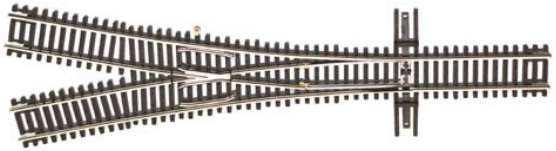
```
module simplecnt (input clk, rst,
                  output reg [3:0] q);

always@(posedge clk)
begin
    if(rst)
        q <= 4'b0000;
    else
        q <= q + 1'b1;
end

endmodule
```

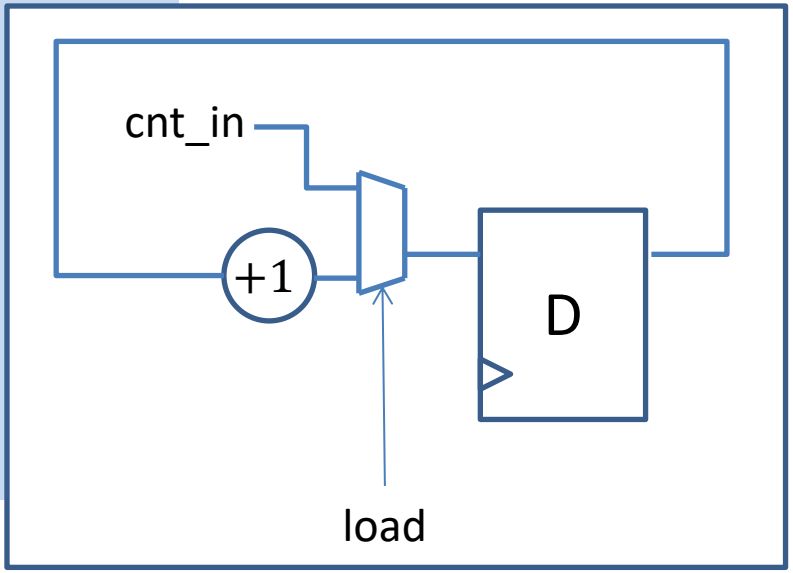


Synchronous Counters in Verilog

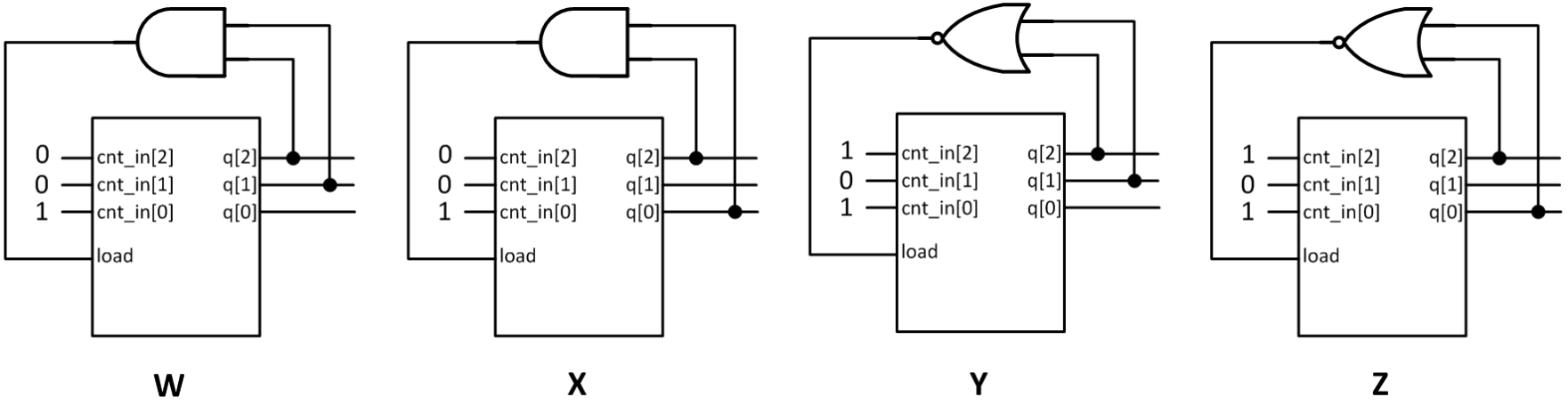
```
module simplecnt (input clk, rst,
                  input down, load,
                  input [3:0] cnt_in,
                  output reg [3:0] q);

always@(posedge clk)
begin
    if(rst)
        q <= 4'b0000;
    else
        if(load)
            q <= cnt_in;
        else
            q <= q + 1'b1;
end
endmodule
```





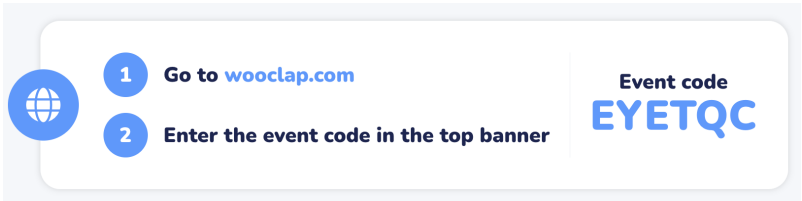


Exercise 2

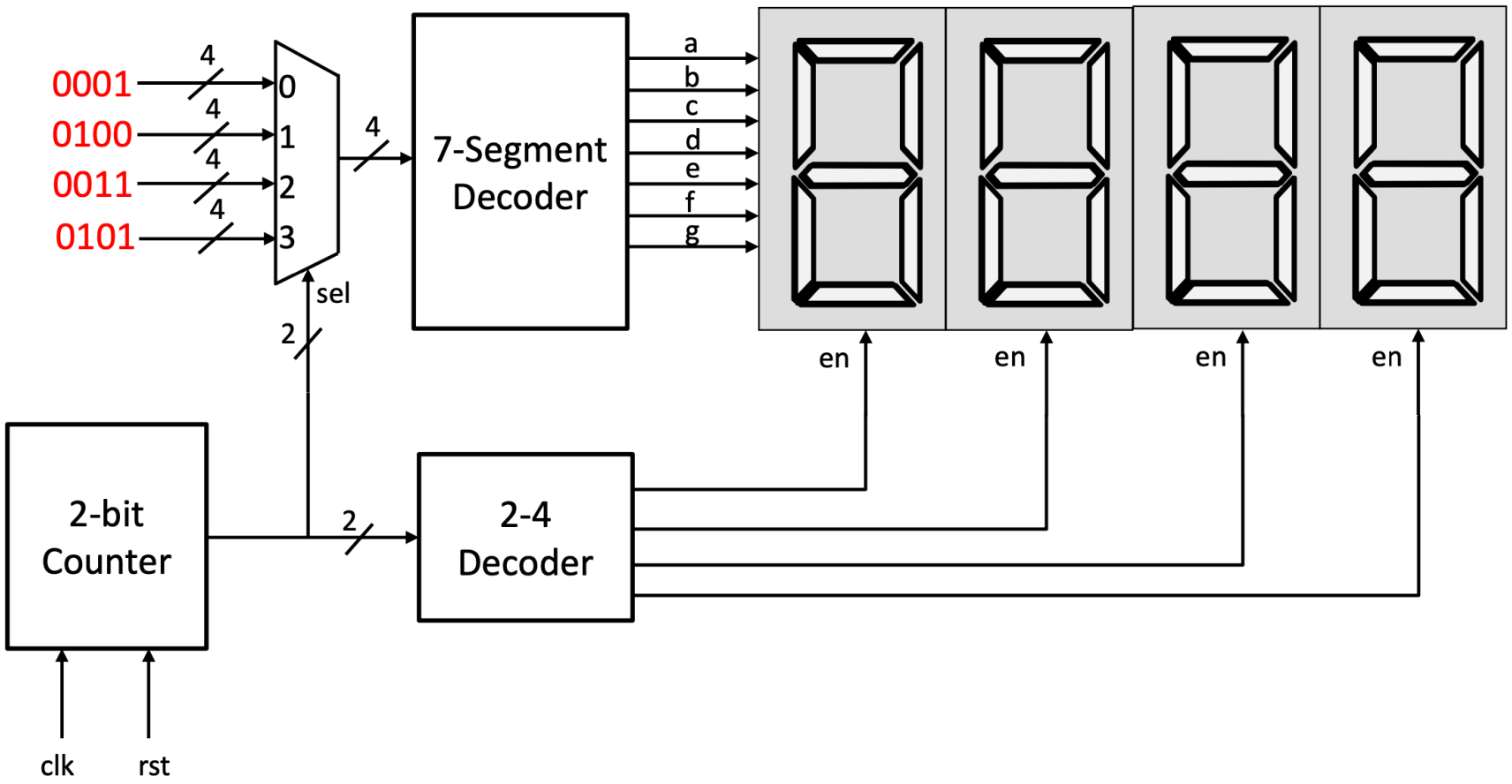


- Using an up-counter with load capability, which circuit will generate a counting sequence of 1, 2, 3, 4, 5, 6? You can assume that the initial state of the counter is 001_2.

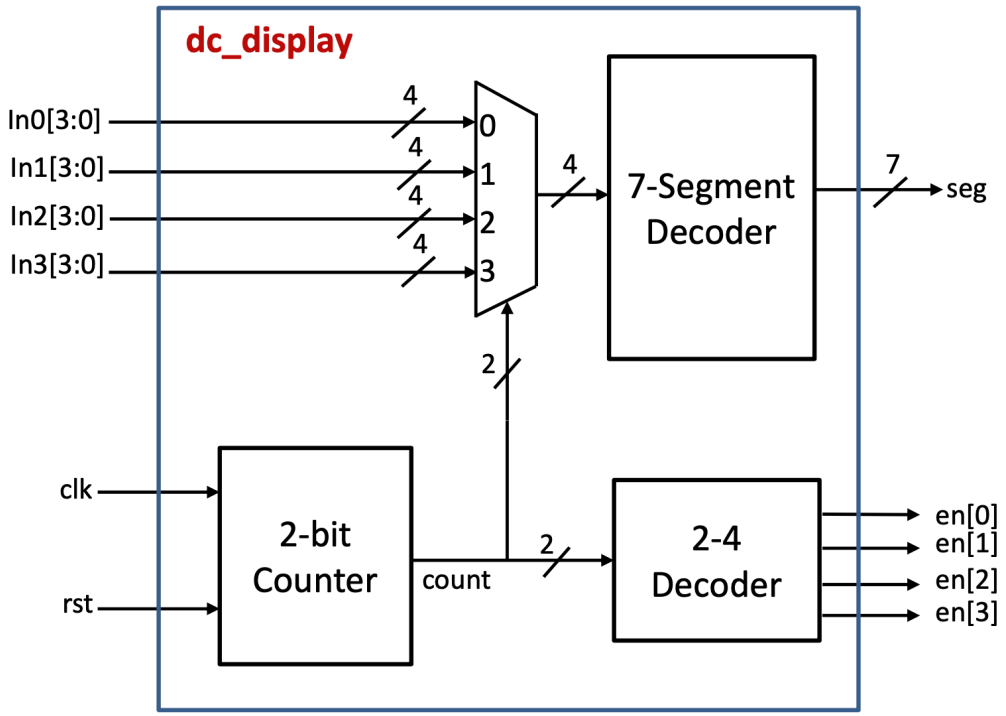
W
X
Y
Z
None of the above



Synchronous Circuit Design



Synchronous Circuit Design




```
module dc_display (...) ...
```

```
// 2-bit counter
always@(posedge clk)
begin
    if(rst)
        count <= 2'b00;
    else
        count <= count + 1'b1;
end
```

```
// 2-to-4 decoder
always @ *
case (count)
  2'b00 : en = 2'b0001;
  2'b01 : en = 2'b0010;
  2'b10 : en = 2'b0100;
  2'b11 : en = 2'b1000;
endcase
```

```
// 4-bit 4x1 multiplexer
always @ *
case (count)
  2'b00 : mx_o = In0;
  2'b01 : mx_o = In1;
  2'b10 : mx_o = In2;
  2'b11 : mx_o = In3;
endcase
```

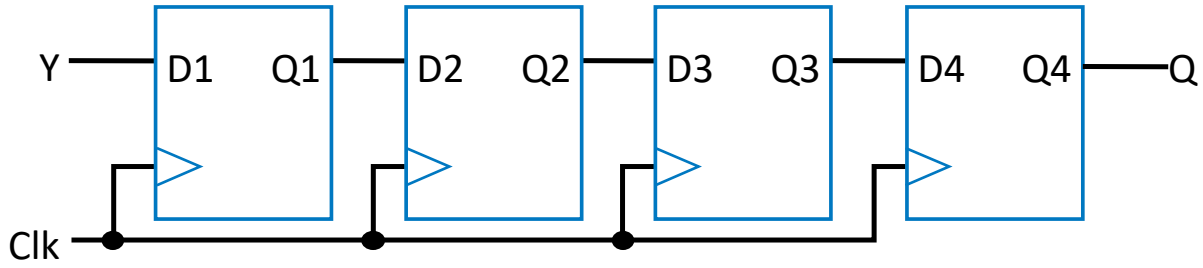
```
// 7-segment decoder
always @ *
case (mx_o)
  4'b0000 : seg = ...;
  4'b0001 : seg = ...;
  ...
  4'b1111 : seg = ...;
endcase
```

```
... endmodule
```

Summary of Lecture 20

- Sequential Circuits in Verilog
  - Shift Registers
  - Serial Data Transfer
  - Memory
  - What gets synthesized

Recap: Shift Registers in Verilog




```
module shiftreg (input clk, y,
                 output reg q);

reg q1, q2, q3;

always@(posedge clk)
begin
    q1 <= y;
    q2 <= q1;
    q3 <= q2;
    q  <= q3;
end

endmodule
```

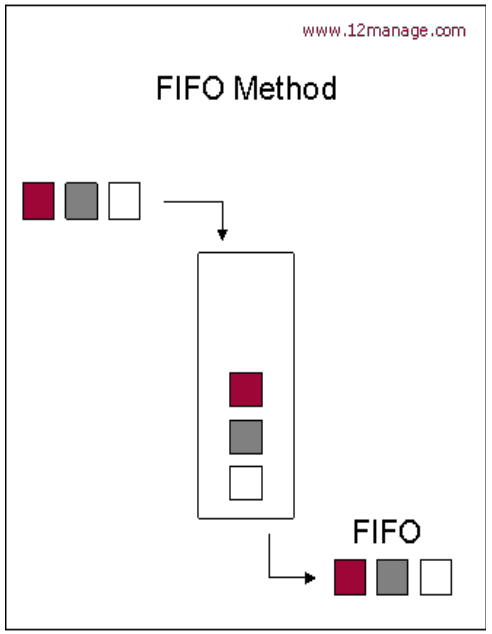
```
module shiftreg (input clk, y,
                 output reg q);

reg q1, q2, q3;

always@(posedge clk)
begin
    q2 <= q1;
    q1 <= y;
    q  <= q3;
    q3 <= q2;
end

endmodule
```

Order does not matter

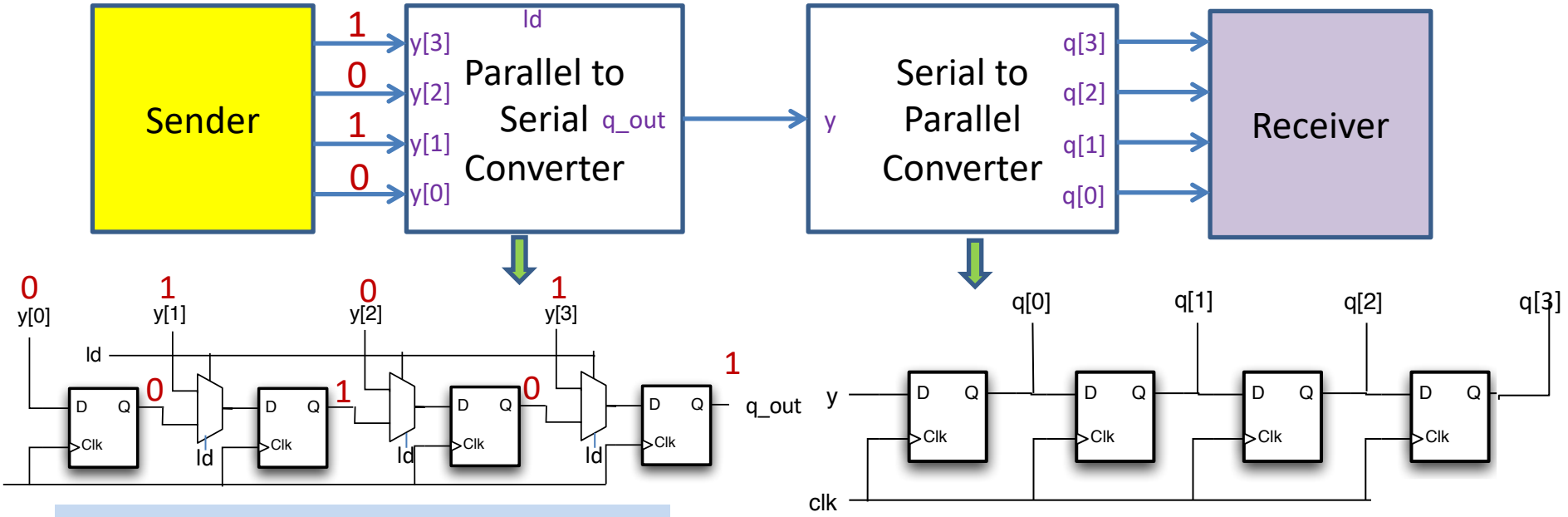
First-In-First-Out Buffers

- FIFOs (First-In-First-Out) are useful for managing dataflow
- The structure is identical to shift registers, but now each register is multi-bit
- The input, output, and internal signal widths should match

```
module fifo4 (input clk, input [3:0] y,
              output reg [3:0] q);

reg [3:0] q1, q2, q3;

always@(posedge clk)
begin
    q1 <= y;
    q2 <= q1;
    q3 <= q2;
    q  <= q3;
end

endmodule
```



Serial to Parallel; Parallel to Serial

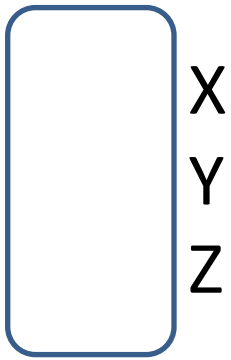



```
module piso4 (input clk, ld,
              input [3:0] y,
              output q_out);
reg [3:0] q;

always@(posedge clk)
begin
    if (ld) q <= y;
    else begin
        q[0]   <= y[0];
        q[3:1] <= q[2:0];
    end
end
assign q_out = q[3];

endmodule
```

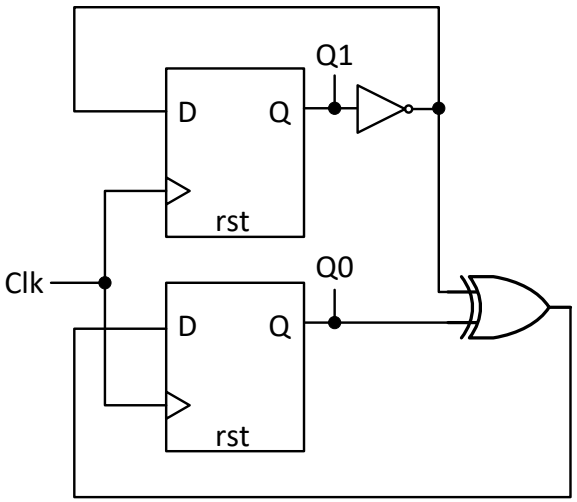
```
module sipo4 (input clk, y,
              output reg [3:0] q);
always@(posedge clk)
begin
    q[0]   <= y;
    q[3:1] <= q[2:0];
end

endmodule
```

Exercise 3

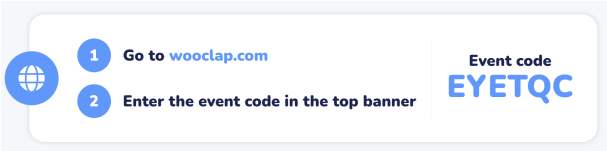
Which of the following is the correct Verilog description of the circuit?



X



Y



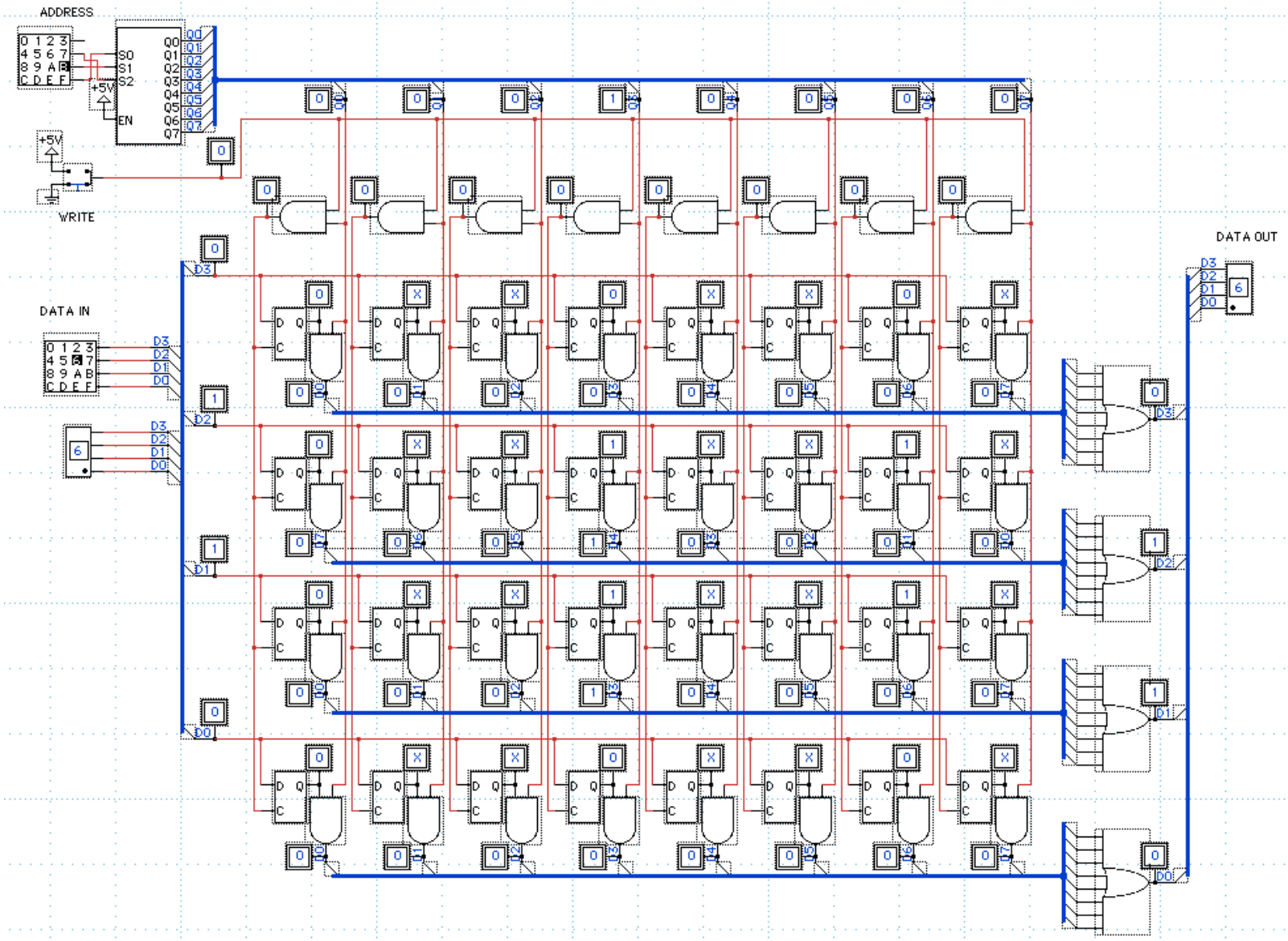
Z

```
  always@(posedge Clk)
  begin
    if (rst) begin
      Q1 <= 1'b0;
      Q0 <= 1'b0;
    end
    else begin
      Q1 <= ~(Q1);
      Q0 <= ~(Q0) ^ Q1;
     end
   end
endmodule
```

```
  always@(posedge Clk)
  begin
    if (rst) begin
      Q1 <= 1'b0;
      Q0 <= 1'b0;
    end
    else begin
      Q1 <= ~(Q1);
      Q0 <= ~(Q1) ^ Q0;
     end
   end
endmodule
```

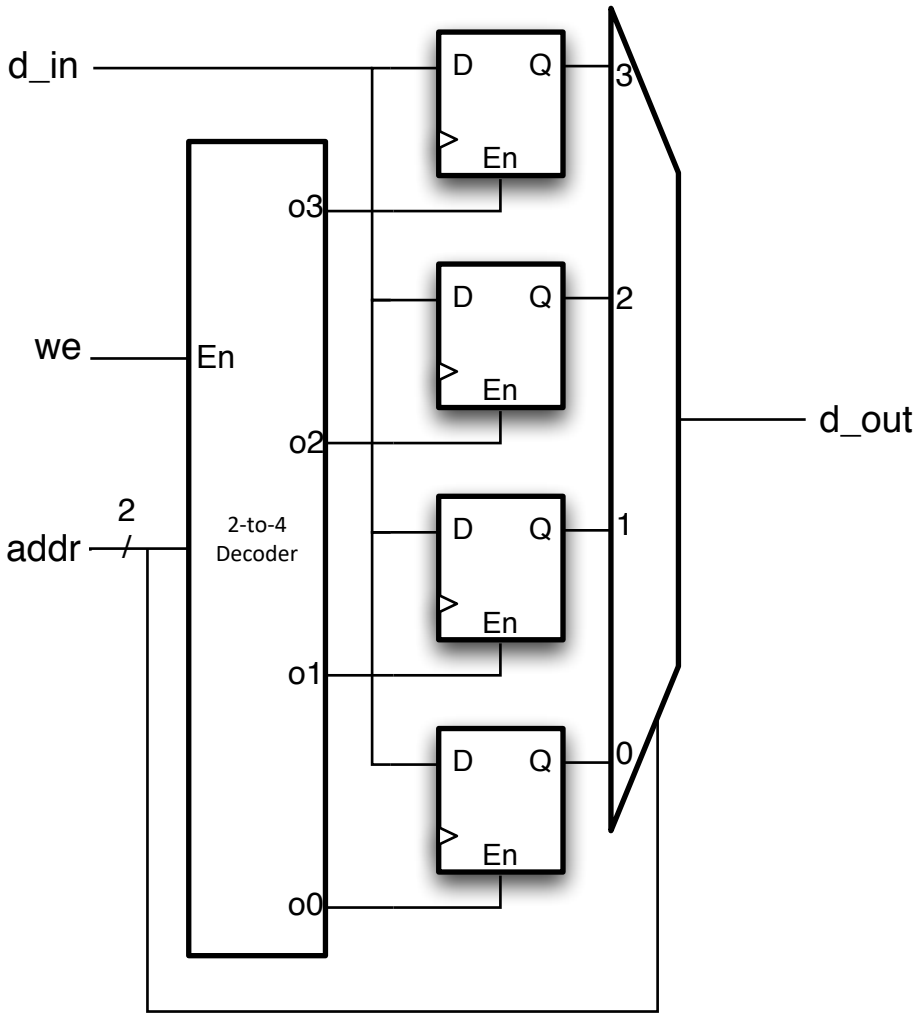
```
  always@(posedge Clk)
  begin
    if (rst) begin
      Q1 <= 1'b0;
      Q0 <= 1'b0;
    end
    else begin
      Q1 <= ~(Q0);
      Q0 <= ~(Q0) ^ Q1;
     end
   end
endmodule
```

Memories



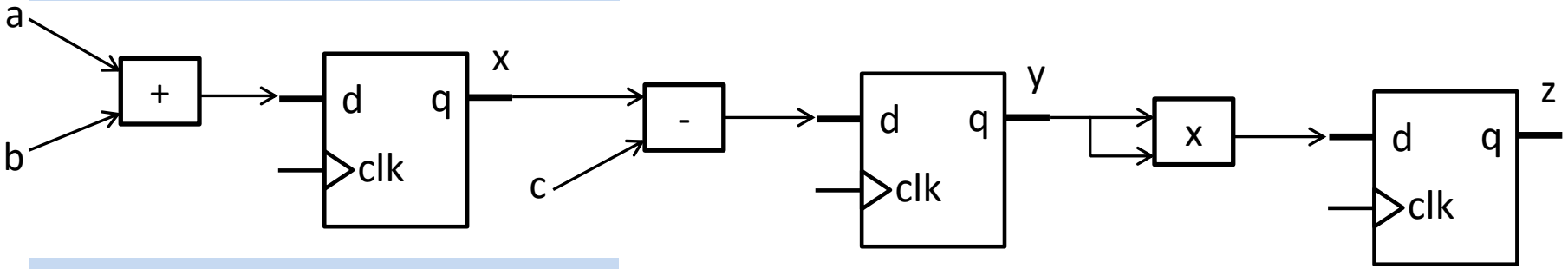
Memories

- In order to store a value, we must assert *we* (write enable) and provide an address
- For reading, a mux connects the corresponding register to the *d_out* output

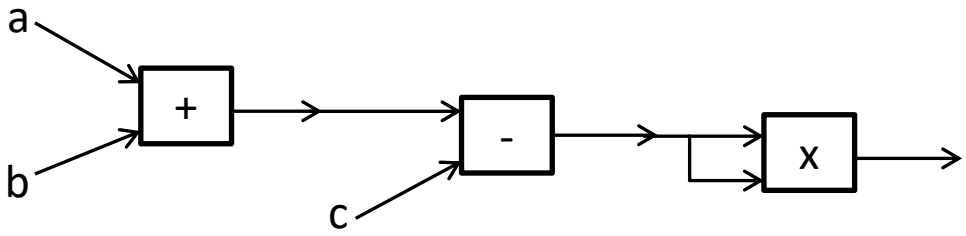


Recap: What Gets Synthesized?

```
always@(posedge clk)
begin
    x <= a + b;
    y <= x - c;
    z <= y * y;
end
```

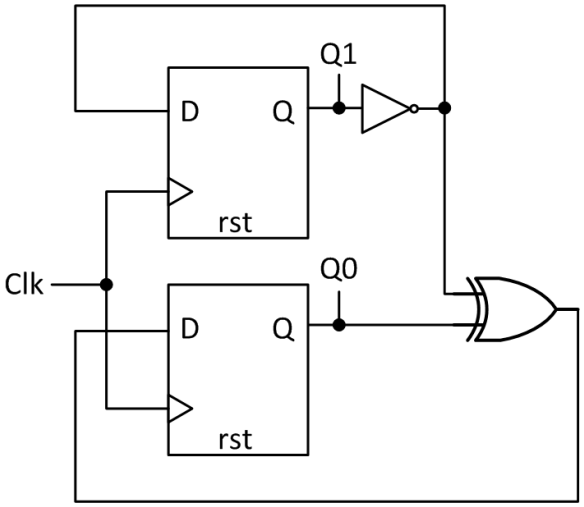


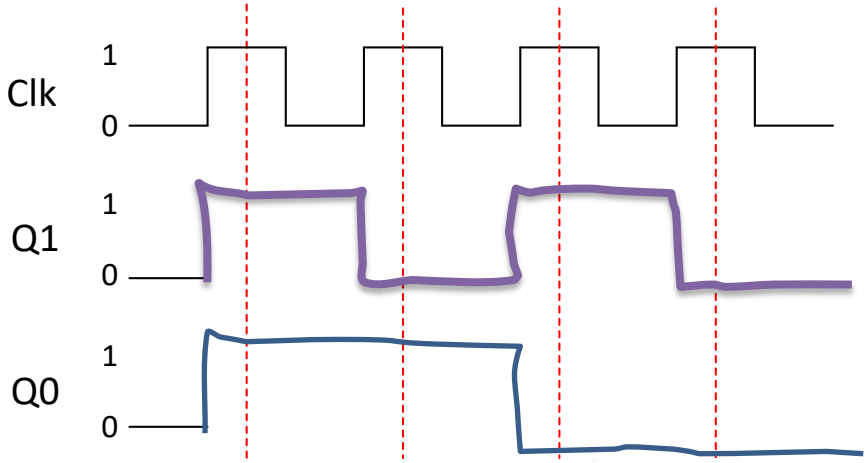

```
always@ *
begin
    x = a + b;
    y = x - c;
    z = y * y;
end
```



Exercise 4

What is the correct output sequence of Q1 and Q0 observed at the dash-red lines?



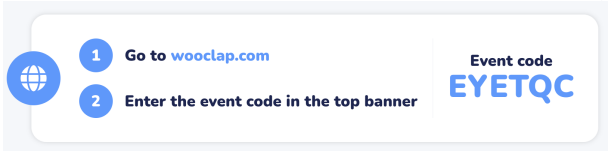


Q1: 1010; Q0: 0011

Q1: 0101; Q0: 1010

Q1: 1010; Q0: 1100

None of the above



END OF L19,L20 SUMMARY