Lecture 3: Continuation of SystemVerilog

UC San Diego
JACOBS SCHOOL OF ENGINEERING
Electrical and Computer Engineering

Last Lecture

- Talked about combinational logic always statements. e.g.,

```
module ex2(input logic a, b, c,
           output logic f);

  logic t; // internal signal

  always_comb
  begin
    t = a & b;
    f = t | c;
  end

endmodule
```

should use "**=**" (called "**blocking**" assignment) in comb. logic always statements. RHS just takes output from the previous equation.

The order of statements matters!

This Lecture

- Talk about "**clocked always statements**", which generate combinational logic gates **and flip-flops**
- Unfortunately, SystemVerilog **does not** have well-defined semantics for describing flip-flops and finite state machines (FSMs)
- Instead, SystemVerilog relies on **idioms** to describe flip-flops and FSMs (i.e., the use of coding templates that synthesis tools will interpret to mean flip-flops and FSMs)
- If you do not follow these "templates", your code may still simulate correctly, but may produce incorrect hardware

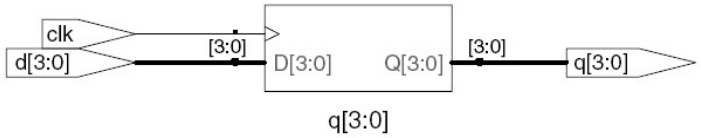
D Flip-Flop

```
module flop(input  logic       clk,
            input  logic [3:0] d,
            output logic [3:0] q);

  always_ff @(posedge clk)
    q <= d;                  // pronounced "q gets d"

endmodule
```

- Verilog calls "**<=**" a "**non-blocking**" assignment.
- It means "**wait until next clk tick**" **before** updating "q"
- This is why synthesis will produce a positive edge-trigged D-FF
- "always_ff" indicates that this is a "clocked always statement



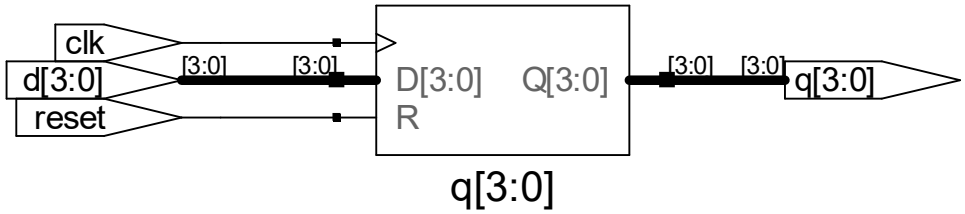
Resettable D Flip-Flop

```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // synchronous reset
  always_ff @(posedge clk)
    if (reset) q <= 4'b0;
    else       q <= d;

endmodule
```

- Using this "template", the "**if**" part specifies the "reset" condition, and the "**else**" part specifies what gets stored "q" after next clk tick
- Synthesis tools will recognize this "template"



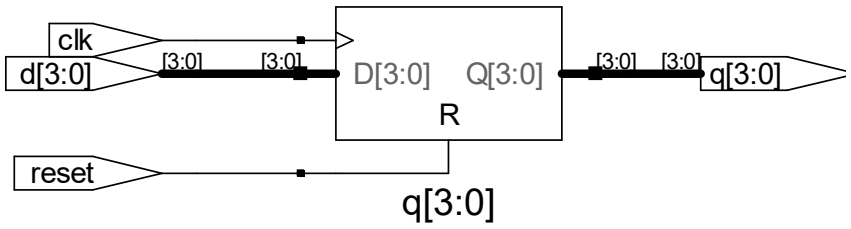
Resettable D Flip-Flop

```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else       q <= d;

endmodule
```

- By specifying the "reset" signal here, synthesis tools will understand the **if-then-else** template matches to an asynchronously resettable D-FF
- Again this works because synthesis tools recognize this template.

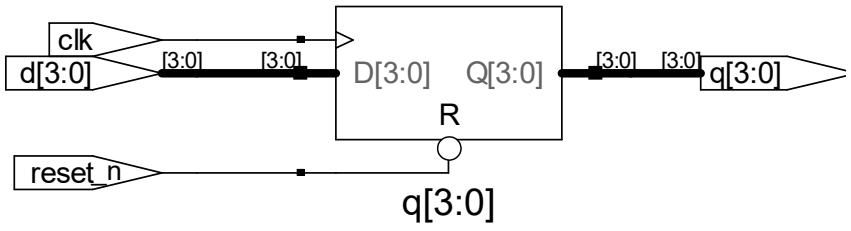


Resettable D Flip-Flop

```
module flopr_n(input  logic       clk,
               input  logic       reset_n,
               input  logic [3:0] d,
               output logic [3:0] q);

  // asynchronous reset
  always_ff @(posedge clk, negedge reset_n)
    if (!reset_n) q <= 4'b0;
    else          q <= d;

endmodule
```



- By specifying the "negedge reset_n" signal here, synthesis tools will asynchronously resettable D-FFs that are reset on the "falling" edge of reset_n
- The template requires the if-then-else to use

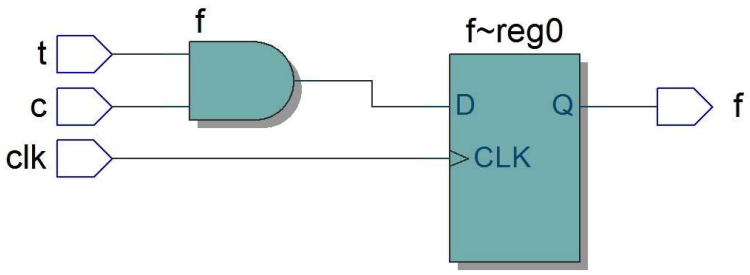
  if (!reset_n) … else ….

Creating logic gates + FFs

```
module ex4(input  logic clk,
           input  logic t, c,
           output logic f);

  always_ff @(posedge clk)
  begin
    f <= t & c;
  end
endmodule
```

- Positive edge triggered register (D-FF) produced for "f".
- "<=" means LHS stores"**new**" value after next clk tick.



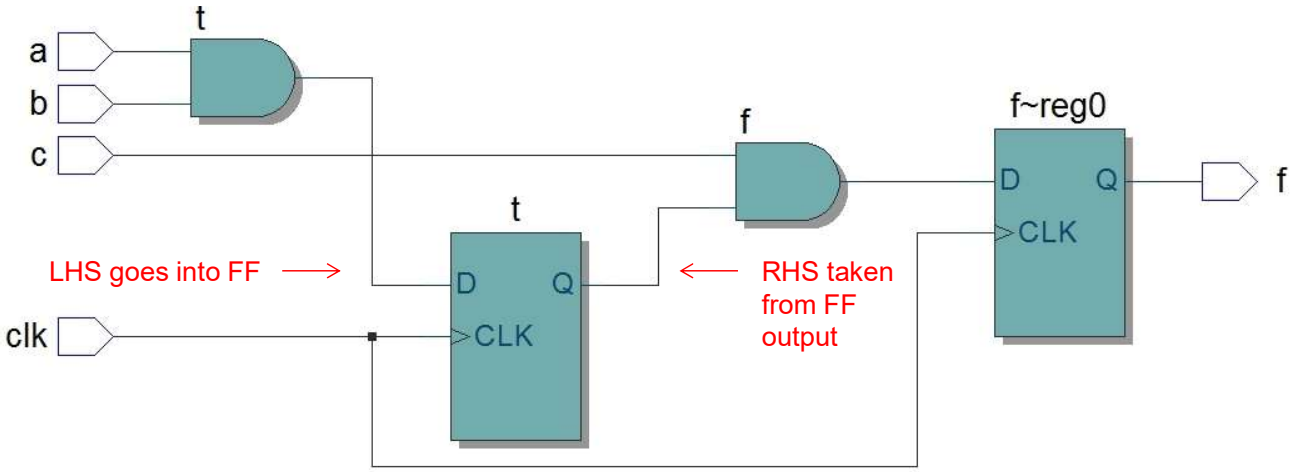
Clocked always statements

```
module ex5(input  logic clk,
           input  logic a, b, c,
           output logic f);
  logic t;
  always_ff @(posedge clk)
  begin
    t <= a & b;
    f <= t & c;
  end
endmodule
```

D-FF produced for "t" and "f"

← LHS stores output of AND-gate to "t" register

← RHS reads from "t" register



Clocked always statements

```
module ex6(input  logic clk, reset_n, a, b, c,
           output logic f);
  logic t;
  always_ff @ (posedge clk, negedge reset_n)
  begin
    if (!reset_n) begin
      t <= 0;
      f <= 0;
    end else begin
      t <= a & b;
      f <= t & c;
    end
  end
endmodule
```

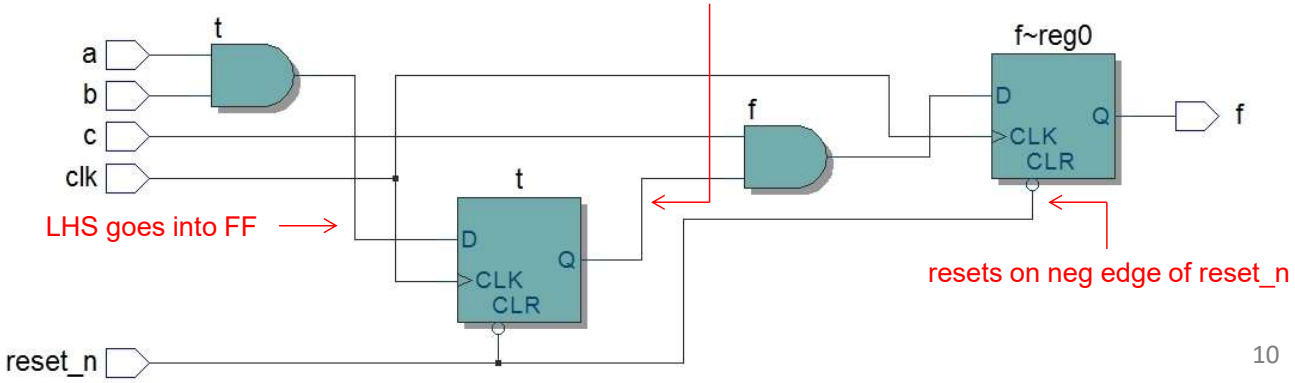
this **if** part specifies how registers should be initialized

Template specifies async reset edge-triggered D-FFs

Positive edge-triggered on "clk"

Async reset **negative** edge on "reset_n"

RHS taken from FF output



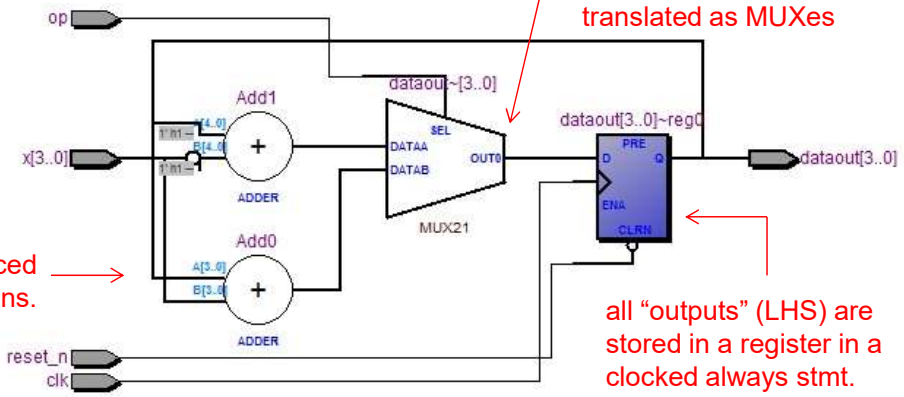
Clocked always statements

```
module ex7 (input logic       clk, reset_n, op,
            input logic [3:0] x,
           output logic [3:0] dataout);
  always_ff @ (posedge clk, negedge reset_n)
  begin
    if (!reset_n)
        dataout <= 4'b0000;
    else
        if (op)
            dataout <= dataout + x;
        else
            dataout <= dataout - x;
  end
endmodule
```

this **if** part specifies how registers should be initialized

- this **else** part specifies how registers should be updated at each cycle.
- all "outputs" (LHS) are stored in a register in a clocked always stmt. LHS specifies "**new**" value that will be stored after next clk tick.
- should use "**<=**" instead of "**=**", which means on the RHS, the value is the "**current**" value of the register.

If-then-else & case stmts translated as MUXes



Arithmetic operators (e.g. "+") are replaced with library logic implementations.

all "outputs" (LHS) are stored in a register in a clocked always stmt.

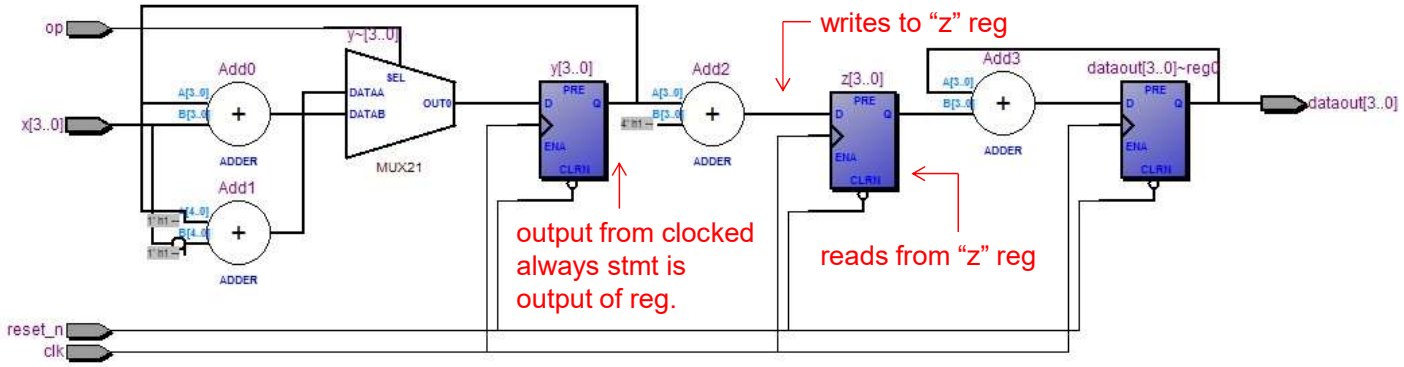
Clocked always statements

```
module ex8 (input logic clk, reset_n, op,
            input logic [3:0] x,
           output logic [3:0] dataout);
  logic [3:0] y, z;

  always_ff@(posedge clk,negedge reset_n)
  begin
  if (!reset_n)
      y <= 4'b0000;
  else
    if (op)
      y <= y + x;
    else
      y <= y - x;
  end
```

```
always_ff@(posedge clk,negedge reset_n)
if (!reset_n) begin
  dataout <= 4'b0000;
  z <= 4'b0000;
end else begin
  z <= y + 1;          ←writes to "z" reg
  dataout <= dataout + z;←reads from "z" reg
end
endmodule
```

1st always stmt **2nd always stmt**

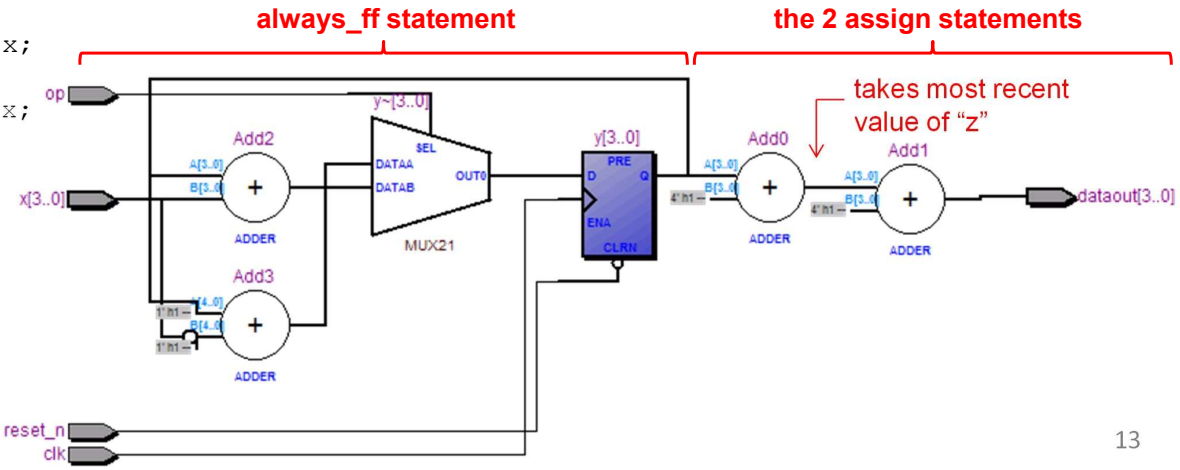


Mixing statements

```
module ex9 (input logic clk, reset_n, op,
            input logic [3:0] x,
           output logic [3:0] dataout);
 logic [3:0] y, z;

 assign z = y + 1;
 assign dataout = z + 1;

 always_ff@(posedge clk, negedge reset_n)
 begin
 if (!reset_n)
  y <= 4'b0000;
 else
  if (op)
    y <= y + x;
  else
    y <= y - x;
 end
endmodule
```



Mixing statements

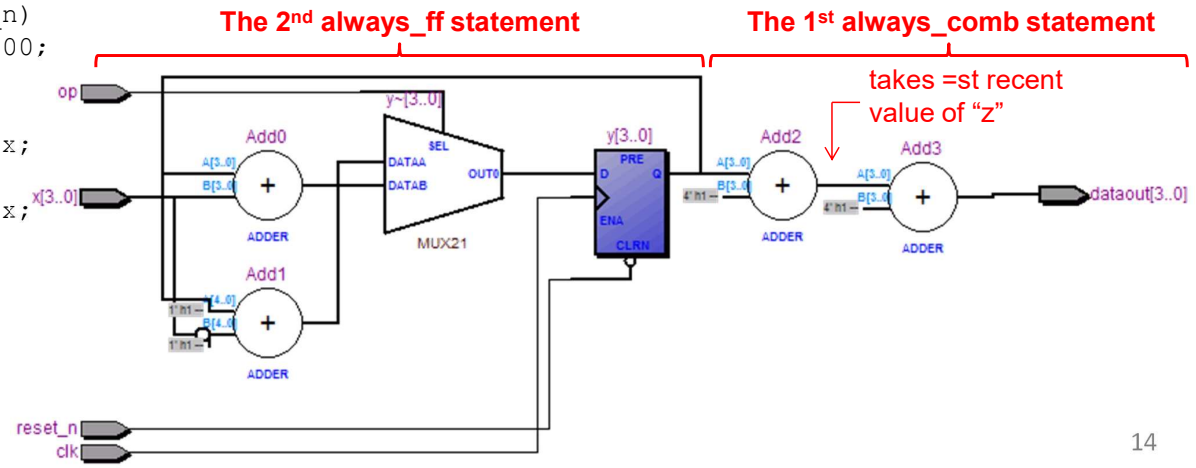
```
module ex10 (input logic clk, reset_n, op,
             input logic [3:0] x,
            output logic [3:0] dataout);
 logic [3:0] y, z;

 always_comb
 begin
   z = y + 1;
   dataout = z + 1;
 end

 always_ff@(posedge clk, negedge reset_n)
 begin
 if (!reset_n)
  y <= 4'b0000;
 else
  if (op)
   y <= y + x;
  else
   y <= y - x;
 end
endmodule
```

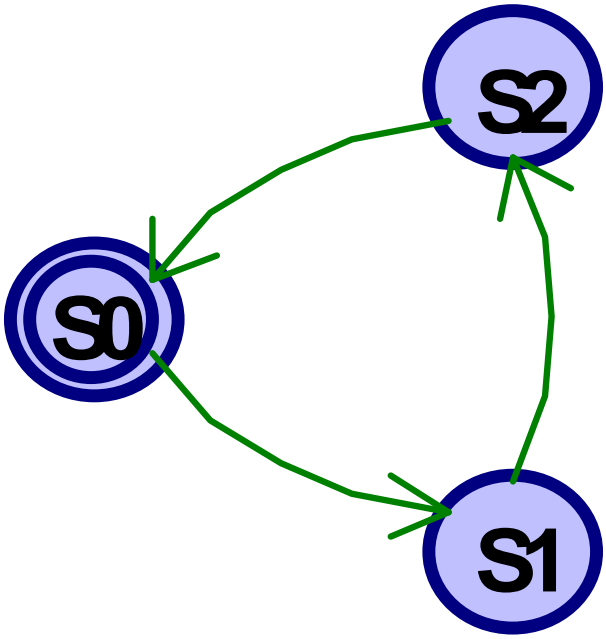
same behavior as ex9, but using always_comb instead of 2 assign statements

should use "=" in comb. logic always stmts. no reg. RHS just takes output from the previous eqn.



Last Lecture: Divide by 3 FSM

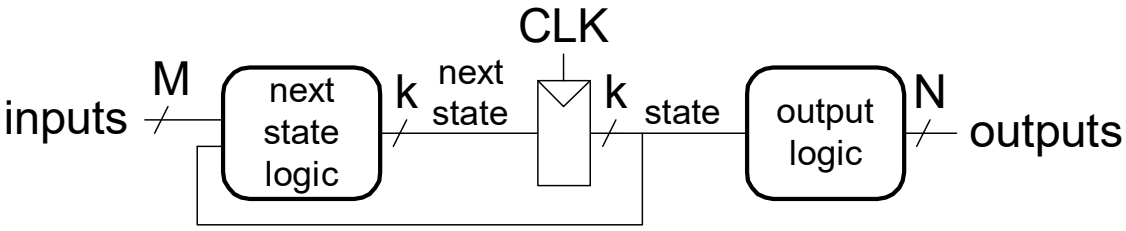
- Output should be “1” every 3 clock cycles



The double circle indicates the reset state

Finite State Machines (FSMs)

- A simple Moore machine looks like the following



FSM Example in SystemVerilog

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

  enum logic [1:0] {S0=2'b00, S1=2'b01, S2=2'b10} state; // declare states as enum

  // next state logic and state register
  always_ff @(posedge clk, negedge reset_n)
  begin
    if (!reset_n)
      state <= S0;
    else begin
      case (state)
        S0: state <= S1;
        S1: state <= S2;
        S2: state <= S0;
      endcase
    end
  end

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

state transition graph is the same thing as a state transition table, which can be specify as a case statement

← output is "1" every clock cycles when we are in state S0