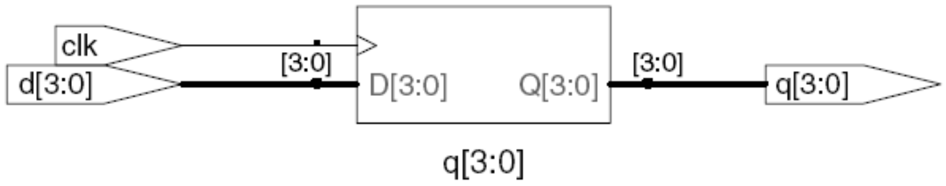
Sequential Logic

- Verilog uses certain idioms to describe latches, flip-flops and FSMs
- Other coding styles may simulate correctly but produce incorrect hardware

D Flip-Flop

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

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

endmodule
```



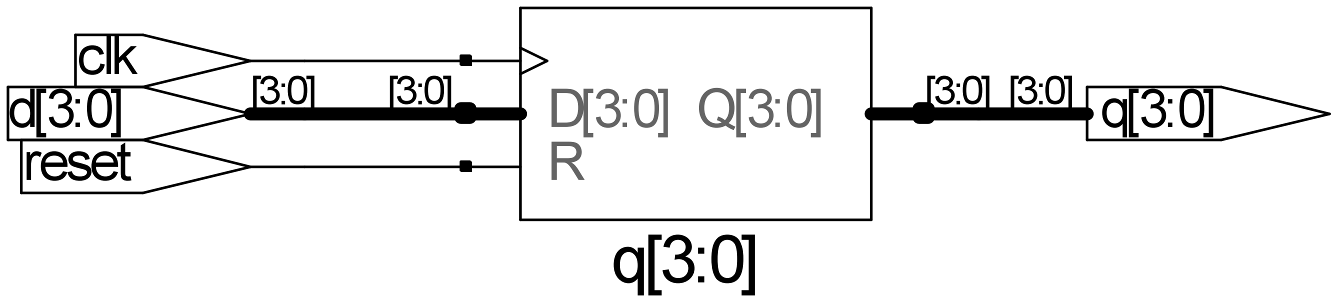
Any signal assigned in an `always` statement must be declared `reg`. In this case `q` is declared as `reg`

Beware: A variable declared `reg` is not necessarily a registered output. We will show examples of this later.

Note the use of the <= assignment operator

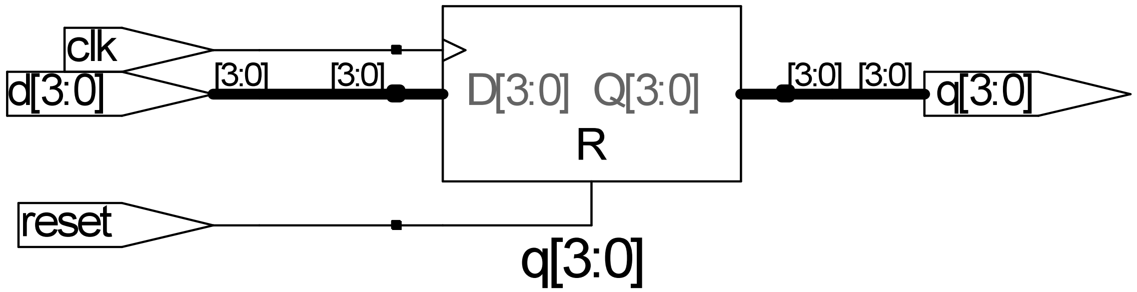
Resettable D Flip-Flop

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

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

endmodule
```



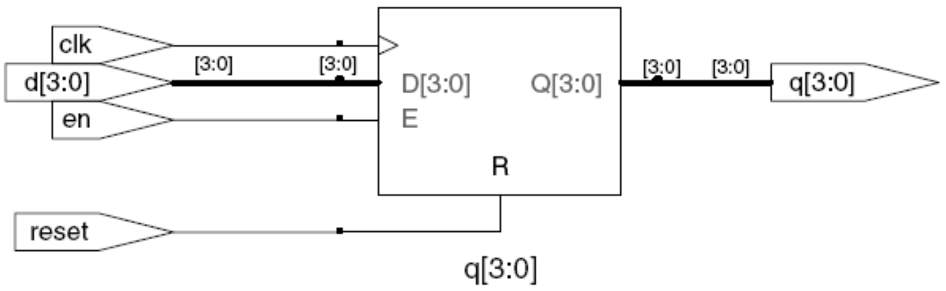
Resettable D Flip-Flop

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

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

endmodule
```



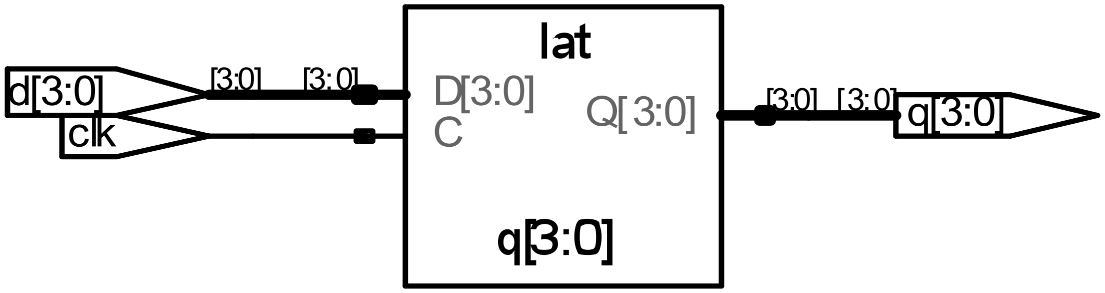
D Flip-Flop with Enable

```
module flopren(input              clk,
               input              reset,
               input              en,
               input        [3:0] d,
               output reg   [3:0] q);

  // asynchronous reset and enable
  always @ (posedge clk, posedge reset)
    if      (reset) q <= 4'b0;
    else if (en)    q <= d;

endmodule
```



Latch

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

  always @ (clk, d)
    if (clk) q <= d;

endmodule
```



Warning: We won't use latches in this course, but you might write code that inadvertently implies a latch. So if your synthesized hardware has latches in it, this indicates an error.

8-bit Register with Synchronous Reset

```
module reg8
  (input reset,
   input CLK,
   input [7:0] D,
   output reg [7:0] Q);

   always @(posedge CLK)
     if (reset)
       Q <= 0;
     else
       Q <= D;
endmodule    // reg8
```

N-bit Register with Asynchronous Reset

```
module regN
  #(parameter N=8)
   (input reset,
    input CLK,
    input [N-1:0] D,
    output reg [N-1:0] Q);

   always @(posedge CLK or posedge reset)
     if (reset)
       Q <= 0;
     else if (CLK == 1)
       Q <= D;
endmodule    // regN
```

Shift Register Example

```
// 8-bit register can be cleared, loaded, shifted left
// Retains value if no control signal is asserted

module shiftReg
  (input  CLK,
   input clr,              // clear register
   input shift,            // shift
   input ld,               // load register from Din
   input [7:0] Din,        // Data input for load
   input  SI,              // Input bit to shift in
   output reg [7:0] Dout);

  always @(posedge CLK) begin
    if (clr)               Dout <= 0;
    else if (ld)           Dout <= Din;
    else if (shift)        Dout <= { Dout[6:0], SI };
  end

endmodule                  // shiftReg
```

Counter Example

- Simple components with a register and extra computation
  - Customized interface and behavior, e.g.
    - counters
    - shift registers

```
// 8-bit counter with clear and count enable controls
module count8
  (input             CLK,
   input             clr,  // clear counter
   input             cntEn,// enable count
   output reg [7:0] Dout);// counter value

  always @(posedge CLK)
    if (clr)                Dout <= 0;
    else if (cntEn)         Dout <= Dout + 1;

endmodule
```

Rules for Signal Assignment

- Use `always @(posedge clk)` and nonblocking assignments (<=) to model synchronous sequential logic

```
always @ (posedge clk)
  q <= d; // nonblocking
```

- Use continuous assignments (assign …) to model simple combinational logic.

```
 assign y = a & b;
```

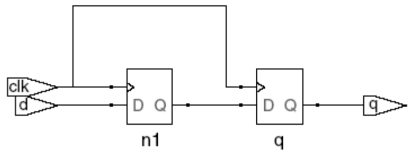
- Use `always @ (*)` and blocking assignments (=) to model more complicated combinational logic if, case, for, etc. statements are useful
- Do not make assignments to the same signal in more than one `always` statement or continuous assignment statement.
    - Equivalent to driving one wire with multiple outputs
    - Assignment to Z is the only exception

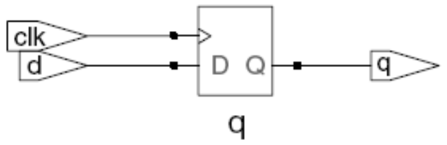
Blocking vs. Nonblocking Assignments

- <= is a "nonblocking assignment"
    - Occurs simultaneously with others
- = is a "blocking assignment"
    - Occurs in the order it appears in the file

```
// Good synchronizer using
// nonblocking assignments
module syncgood
  (input      clk,
   input      d,
   output reg q);
  reg n1;
  always @(posedge clk)
    begin
      n1 <= d;  // nonblocking
      q  <= n1; // nonblocking
    end
endmodule
```

```
// Bad synchronizer using
// blocking assignments
module syncbad
  (input      clk,
   input      d,
   output reg q);
  reg n1;
  always @(posedge clk)
    begin
      n1 = d;  // blocking
      q  = n1; // blocking
    end
endmodule
```





Blocking and Non-Blocking Assignments

- Blocking assignments (Q = A)
  - variable is assigned immediately before continuing to next statement
  - new variable value is used by subsequent statements
- Non-blocking assignments (Q <= A)
  - variable is assigned only after all statements already scheduled are executed
    - value to be assigned is computed here but saved for later
  - usual use: register assignment
    - registers simultaneously take their new values after the clock tick
- Example: swap

```
always @(posedge CLK)
    begin
        temp = B;
        B = A;
        A = temp;
    end
```

```
always @(posedge CLK)
    begin
        A <= B;
        B <= A;
    end
```

ELSEVIER

Swap (continued)

- The real problem is parallel blocks
  - one of the blocks is executed first
  - previous value of variable is lost

```
always @(posedge CLK)        always @(posedge CLK)
    begin                        begin
        A = B;                       B = A;
    end                          end
```

- Use delayed assignment to fix this
  - both blocks are scheduled by posedge CLK

```
always @(posedge CLK)        always @(posedge CLK)
    begin                        begin
        A <= B;                      B <= A;
    end                          end
```

Non-Blocking Assignment

- Non-blocking assignment is also known as an RTL assignment
  - if used in an always block triggered by a clock edge
  - mimic register-transfer-level semantics – all flip-flops change together
- My rule: ALWAYS use <= in sequential (posedge clk) blocks

```
// this implements 3 parallel flip-flops
always @(posedge clk)
   begin
      B = A;
      C = B;
      D = C;
   end
```

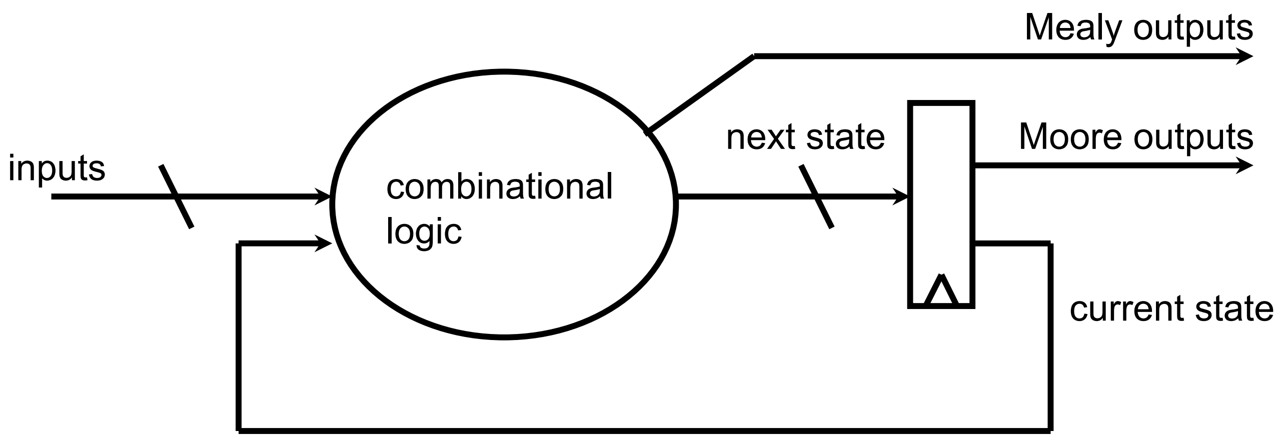
```
// this implements a shift register
always @(posedge clk)
   begin
      {D, C, B} = {C, B, A};
   end
```

```
// this implements a  shift register
always @(posedge clk)
   begin
      B <= A;
      C <= B;
      D <= C;
   end
```

ELSEVIER

Finite State Machines

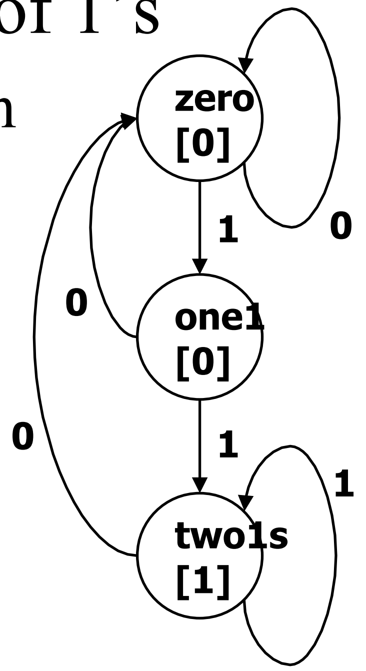
- Recall FSM model



- Recommended FSM implementation style
  - Implement combinational logic using a one always block
  - Implement an explicit state register using a second always block

Verilog FSM - Reduce 1s example

- Change the first 1 to 0 in each string of 1's
  - Example Moore machine implemenation



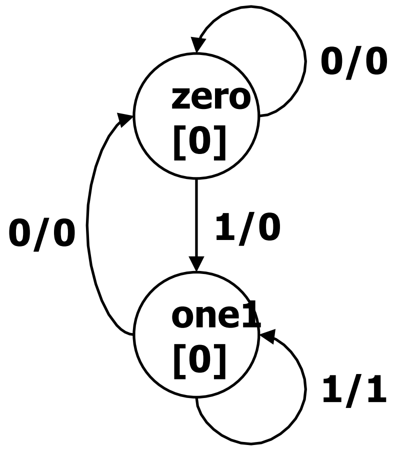
```
module reduce
  (input clk, reset, in,
   output reg out);

// State assignment
  localparam ZERO = 0, ONE1 = 1, TWO1s = 2;
  reg [1:0] state, next_state;     // state register

// Implement the state register
  always @(posedge clk)
    if (reset) state <= ZERO;
    else       state <= next_state;
```

Moore Verilog FSM (cont’d)

```
  always @(*) begin
    out = 0;     // defaults
    next_state = state;
    case (state)
      ZERO: begin         // last input was a zero
        if (in) next_state = ONE1;
      end

      ONE1: begin         // we've seen one 1
        if (in) next_state = TWO1S;
        else    next_state = ZERO;
      end

      TWO1S: begin        // we've seen at least 2 ones
        out = 1;
        if (~in) next_state = ZERO;
      end
      // Don’t need case default because of default assignments
    endcase
  end
endmodule
```

Mealy Verilog FSM for Reduce-1s example

```
module reduce
  (input clk, reset, in,
   output reg out);

// State register
  localparam ZERO = 0, ONE = 1;
  reg state, next_state; // state register
  always @(posedge clk)
    if (reset) state <= ZERO;
    else       state <= next_state;

  always @(*)
    out = 0;
    next_state = state;
    case (state)
      ZERO:                // last input was a zero
        if (in) next_state = ONE;
      ONE:                 // we've seen one 1
      if (in) begin
        out = 1;
      end else begin
         out = 0;
         next_state = ZERO;
       end
    endcase
endmodule
```



Restricted FSM Implementation Style

- Mealy machine requires two always blocks
  - register needs posedge CLK block
  - input to output needs combinational block
- Moore machine can be done with one always block
  - e.g. simple counter
  - Not a good idea for general FSMs
    - Can be very confusing (see example)
- Moore outputs
  - Share with state register, use suitable state encoding