ECE 351
Verilog and FPGA Design

Week 7_1: Blocking and non-blocking assigns (revisited)
Modeling sequential logic

Roy Kravitz
Electrical and Computer Engineering Department
Maseeh College of Engineering and Computer Science

What did you think of the midterm?

Problem 1 solution: ..\..\exams\midterm_exam\solution_code\sync_ctr_mt

Blocking and non-blocking assignments (Revisited)

Blocking vs. non-blocking assignment

- The SystemVerilog standard allows always blocks to be scheduled in any order
  - If the 1st block executes first, the 2nd block will calculate Out based on the new value of Sel
  - If the 2nd block executes first, it will use the old value of Sel to calculate Out
- Solutions include using non-blocking assignment or putting both functions in the same block

```
always_ff @(posedge clk)
  begin
    Sel = In0;
  end

always_ff @(posedge clk)
  begin
    if (Sel == 1)
      Out = PB;
    else
      Out = 1'b0;
  end
```

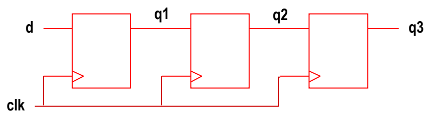
Portland State UNIVERSITY

Blocking vs. non-blocking assignment (cont'd)

- Examples from Cliff Cummings' excellent paper on non-blocking assignments (used with permission of the author)

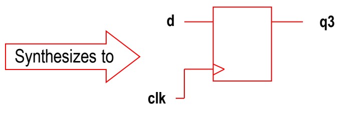
  Download the paper from the Course Content/Other Readings Interesting Articles/CummingsSNUG2000SJ_NBA

- Suppose you want to write SystemVerilog for the three-stage pipeline register below



Blocking assignment: Example #1

```
module pipeb1 (q3, d, clk);
   output logic [7:0] q3;
   input logic  [7:0] d;
   input logic  clk;
   logic [7:0] q3, q2, q1;
   always_ff @(posedge clk) begin
      q1 = d;
      q2 = q1;
      q3 = q2;
   end
endmodule
```

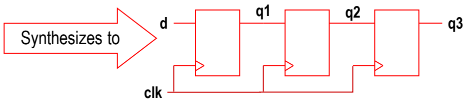


intermediate stages are optimized out because they are within a single always block and even though the statements are blocking, the minimized equation is q3 = d;

On every clock rising edge the value of d is transferred directly to q3 without delay!

Blocking assignment: Example #2

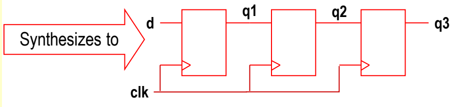
```
module pipeb2 (q3, d, clk);
   output logic [7:0] q3;
   input logic  [7:0] d;
   input logic  clk;
   logic [7:0] q3, q2, q1;
   always_ff @(posedge clk) begin
      q3 = q2;
      q2 = q1;
      q1 = d;
   end
endmodule
```



Changing the order of blocking assignments within a clocked always block yields dramatically different results!

Blocking assignment: Example #3

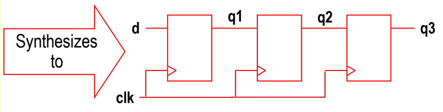
```
module pipeb3 (q3, d, clk);
   output logic [7:0] q3;
   input logic  [7:0] d;
   input logic  clk;
   logic [7:0] q3, q2, q1;
   always_ff@ (posedge clk) q1=d ;
   always_ff@ (posedge clk) q2=q1;
   always_ff@ (posedge clk) q3=q2;
endmodule
```



…but simulation results are unpredictable because the three always blocks can be executed in any order

Blocking assignment: Example #4

```
module pipeb4 (q3, d, clk);
   output [7:0] q3;
   input  [7:0] d;
   input  clk;
   reg [7:0] q3, q2, q1;
   always@ (posedge clk) q3=q2;
   always@ (posedge clk) q2=q1;
   always@ (posedge clk) q1=d;
endmodule
```



Simulation results are unpredictable just as before

Non-blocking assignment: Examples #1, #2

When each of the four blocking-assignment examples are rewritten with non-blocking assignments, all four simulate correctly and synthesize correctly

```
module pipen1 (q3, d, clk);
   output logic [7:0] q3;
   input  logic [7:0] d;
   input logic  clk;
   logic [7:0] q3, q2, q1;
   always_ff @(posedge clk) begin
      q1 <= d;
      q2 <= q1;
      q3 <= q2;
   end
endmodule
```

```
module pipen2 (q3, d, clk);
   output logic [7:0] q3;
   input logic  [7:0] d;
   input logic  clk;
   logic [7:0] q3, q2, q1;
   always_ff @(posedge clk) begin
      q3 <= q2;
      q2 <= q1;
      q1 <= d;
   end
endmodule
```

Non-blocking assignment: Examples #3, #4

```
module pipen3 (q3, d, clk);
   output logic [7:0] q3;
   input logic  [7:0] d;
   input logic clk;
   logic [7:0] q3, q2, q1;
   always_ff @(posedge clk) q1<=d ;
   always_ff @(posedge clk) q2<=q1;
   always_ff @ posedge clk) q3<=q2;
endmodule
```

```
module pipen4 (q3, d, clk);
   output logic [7:0] q3;
   input logic [7:0] d;
   input logic  clk;
   logic [7:0] q3, q2, q1;
   always_ff @(posedge clk) q3<=q2;
   always_ff @(posedge clk) q2<=q1;
   always_ff @(posedge clk) q1<=d;
endmodule
```

Blocking vs. non-blocking scoreboard

- Which of the 4 blocking and 4 nonblocking examples are guaranteed to simulate and synthesize correctly?

	Blocking		Nonblocking	
Example	Sim	Synth	Sim	Synth
1	No	No	Yes	Yes
2	Yes	Yes	Yes	Yes
3	No	Yes	Yes	Yes
4	No	Yes	Yes	Yes

Which assignment should I use?

- Recommended:
  - Use non-blocking assignments for modeling clocked processes in sequential logic.
  - Use blocking assignments for modeling combinational logic

Modeling sequential logic

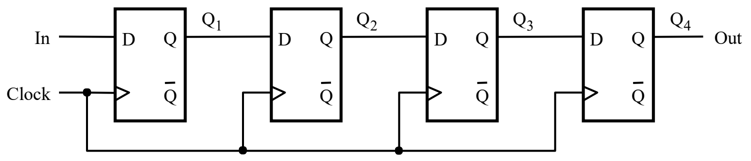
Source material drawn from:
- Roy’s ECE 351 and ECE 571 lecture material
- *RTL Modeling with SystemVerilog* by Stuart Sutherland
- *Logic Design and Verification Using SystemVerilog* by Donald Thomas

Shift registers and counters

Sources:
- Roy's lecture notes from days long past (2003 – 2004)
- *Fundamentals of Digital Logic with Verilog 1st Edition*, Stephen Brown and Zvonko Vranesic, McGraw-Hill, 2003

17

Shift registers



A Shift register using 4 D flip-flops

```
module shiftreg4 (
     input logic In, Clock
     output Out
);
     logic [1:4] Q;

     assign Out = Q[4];
     always_ff @(posedge Clock)
         Q <= {In, Q[1:3]};

endmodule
```

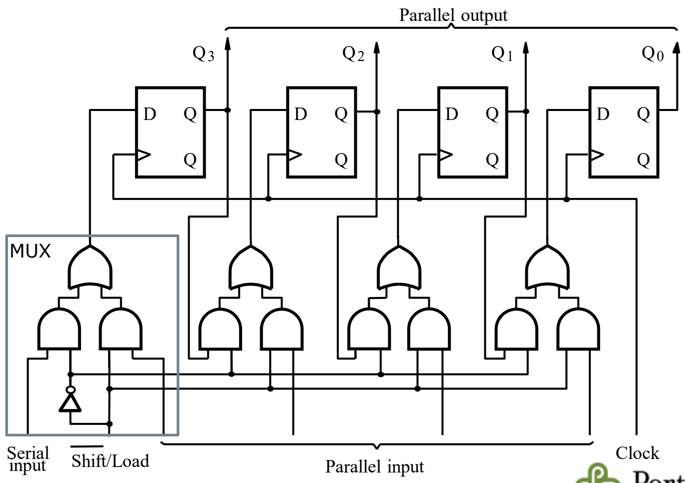
	In	Q_1	Q_2	Q_3	Q_4 = Out
t_0	1	0	0	0	0
t_1	0	1	0	0	0
t_2	1	0	1	0	0
t_3	1	1	0	1	0
t_4	1	1	1	0	1
t_5	0	1	1	1	0
t_6	0	0	1	1	1
t_7	0	0	0	1	1

Portland State UNIVERSITY

18

Parallel access shift register

Loads input or shifts on clock edge



Parallel access shift register (cont'd)

D flip-flop with a 2-to-1 multiplexer on the D input

```
module muxdff (
    input logic D0, D1, Sel, Clock,
    output logic Q
);
    always_ff @(posedge Clock)
        if (!Sel)
            Q <= D0;
        else
            Q <= D1;
endmodule: muxdff
```

Four-bit shift register

```
module shift4 (
    input logic [3:0] R,
    input logic L, w, Clock,
    output logic [3:0]
);

    muxdff Stage3 (w,    R[3], L, Clock, Q[3]);
    muxdff Stage2 (Q[3], R[2], L, Clock, Q[2]);
    muxdff Stage1 (Q[2], R[1], L, Clock, Q[1]);
    muxdff Stage0 (Q[1], R[0], L, Clock, Q[0]);

endmodule: shift4
```

Parallel access shift register (cont'd)

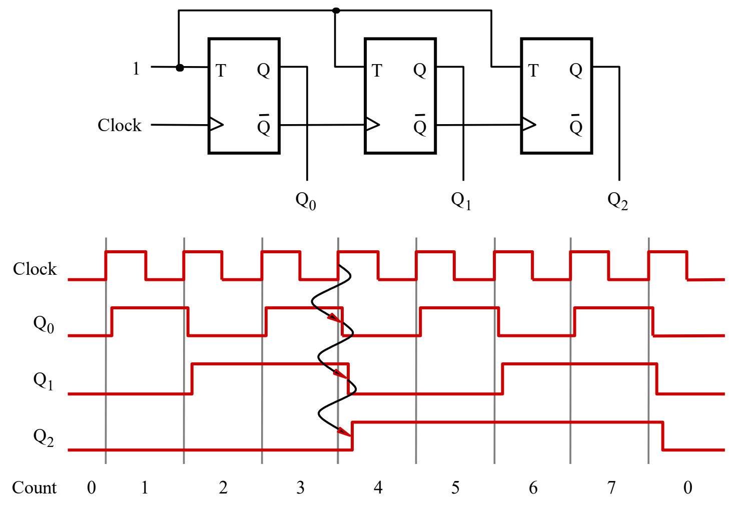
Alternate code for a four-bit shift register

```
module shift4 (
    input logic [3:0] R,
    input logic L, w, Clock,
    output logic [3:0] Q
);

    always_ff @(posedge Clock)
        if (L)
          Q <= R;
        else begin
          Q[0] <= Q[1];  // Could also be Q <= {w, Q[3:1]};
          Q[1] <= Q[2];
          Q[2] <= Q[3];
          Q[3] <= w;
        end

endmodule: shift4
```

An *n*-bit parameterized shift register

```
module shiftn
#(  parameter n = 16)
(
    input logic [n-1:0] R,
    input logic L, w, Clock,
    output logic [n-1:0] Q
);

    logic [n-1:0] Q;

    always_ff @(posedge Clock)
        if (L)
          Q <= R;
        else begin
            for (int k = 0; k < n-1; k = k+1) begin
                Q[k] <= Q[k+1];
                Q[n-1] <= w;
            end
        end

endmodule: shiftn
```

Asynchronous Up-Counter



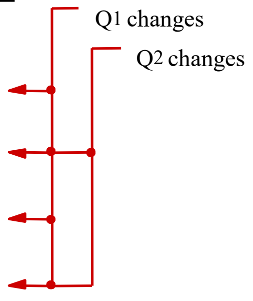
Synchronous Up-counter

A large asynchronous counter can be very slow as the toggle propagates through all the bits

A synchronous counter provides a much Cleaner design

Bit *n* toggles if all lower significant bits are 1

Clock cycle	Q2	Q1	Q0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
8	0	0	0



Synchronous Up-counter (cont'd)

```
module upcount (
    input logic Resetn, Clock, E
    output logic [3:0] Q
);
    logic [3:0] Q;

    always_ff @(negedge Resetn or posedge Clock)
       if (!Resetn)
          Q <= 4'b0;
       else if (E)
          Q <= Q + 1'b1;
       else
          Q <= Q;

endmodule: upcount
```

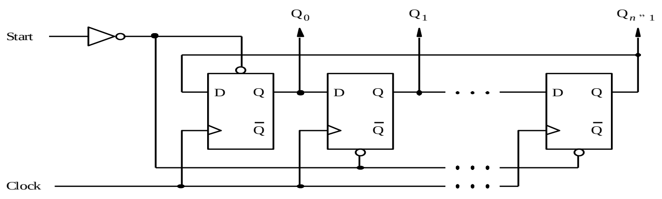
Synchronous counter with parallel load

```
module upcount (
    input logic [3:0] R,
    input logic Resetn, Clock, E, L,
    output logic [3:0] Q
);
    logic [3:0] Q;

    always_ff @(negedge Resetn or posedge Clock)
       if (!Resetn)
          Q <= 0;
       else if (L)
          Q <= R;
       else if (E)
          Q <= Q + 1;   // for a down counter Q <= Q - 1;
       else
          Q <= Q;

endmodule
```

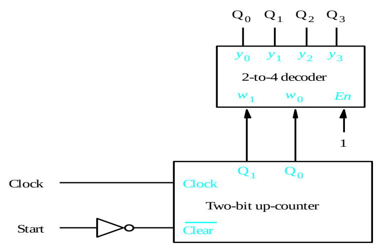
Up/Down counter w/ parallel load

```
module updowncount
#(parameter n = 8)
(
   input logic [n-1:0] R,
   input logic Clock, L, E, up_down,
   output logic [n-1:0] Q
);
   logic [n-1:0] Q;
   logic direction;

   always_ff @(posedge Clock) begin
      direction = up_down ? 1 : -1;

      if (L)
        Q <= R;
      else if (E)
        Q <= Q + direction;
   end

endmodule: updowncount
```

Other counters – Ring counter



(a) An *n*-bit ring counter

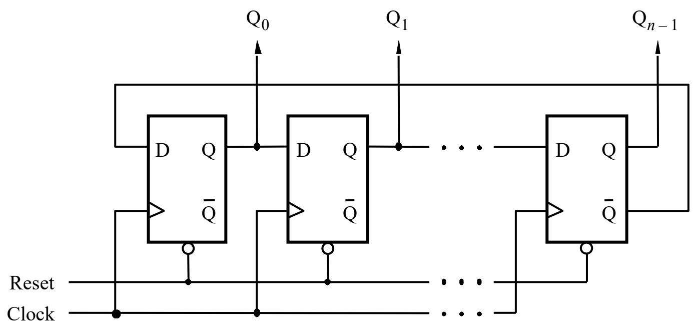


(b) A four-bit ring counter

A ring-counter moves a 1 around a "ring" of latches:
1000->
0100->
0010->
0001->
1000 …

A "one-hot" encoding

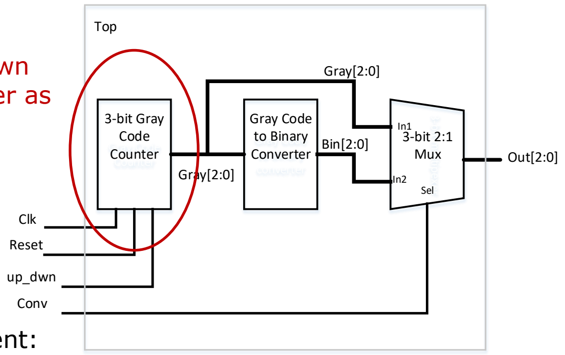
Other counters – Johnson counter



Johnson counter: 0000 -> 1000 -> 1100 -> 1110 -> 1111 -> 0111 -> 0011 -> 0001 -> 0000 ->

Gray Code counter/mux revisited on last time

Implement up/dwn gray code counter as an FSM



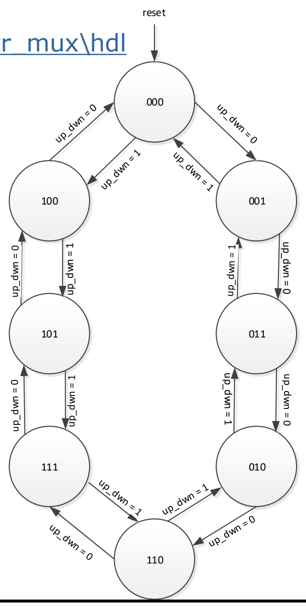
Problem statement:

Implement a gray-code counter w/ a selectable binary and gray code output

When Conv == 0 you want Out[2:0] to be the unconverted Gray Code from the counter. When Conv == 1 you want the Out[2:0] to be the Binary code from the converter. The 3-bit Up/Down Gray Code counter will be implemented as a FSM.

Gray Code counter state transition diagram

Example:
..\examples\gray_code_cntr_mux\hdl



Next Time

- Topics:
  - Modeling sequential logic (wrap-up)
  - Avoiding unintentional latches
- You should:
  - Read Sutherland Ch 9
- Homework, projects and quizzes
  - Homework #3 will be released by Thu, 13-May. Due to D2L by 10:00 PM on Mon, 24-May