Homework 5

Please turn in all verilog code and a hardcopy of your simulation results. You can demonstrate proper operation by either a printout of the "monitor" output (the table printed once the simulation has been run) or by a printout of the waveforms (the results of some problems are easier to see one way and the results of other problems are easier to see another). **Be sure to annotate your simulation results telling me how your results prove that you have met all specifications.**

The Problem:

Envision a stream of 4-bit data packets that is traveling across an asynchronous channel. The source of the information has two 1-bit communication ports, **READY** and **DONE**. This configuration is illustrated in figure 1:

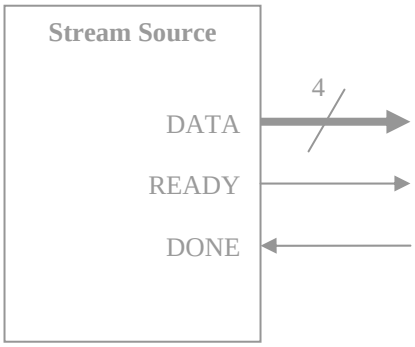


Figure 1: Interface of Stream Source Module.

When the Stream Source Module (SSM) has a new packet of data available, it asserts the active-high **READY** port. It *continues* to assert **READY** until the **DONE** port is asserted, telling the SSM that the data has been received and that it is ok to produce a new packet of data.

Your job is design, implement, and test a Stream Receiving Buffer Module (SRBM) that

- Communicates with the SSM via **READY** and **DONE**
- Stores up to two packets of data
- Outputs a stored packet of data on an output line **OUT**
- Communicates with a third device through the ports **NEW** and **UPDATE**

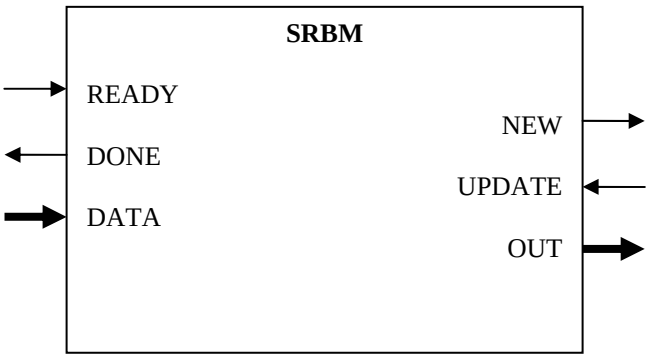
The SRBM asserts **DONE** when it has successfully stored the new data packet from the SSM. It outputs the next packet in sequential order of arrival on the 4-bit output **OUT**, asserting **NEW** whenever new data is available. When the third, unknown device has

collected the data from **OUT**, it will assert **UPDATE**, signaling to the SRBM that it can output the next packet of data.

Note that at any time the SRBM could be in one of the following situations:

- Both packets are stored, waiting for the third device to signal **UPDATE**
- Both packets are empty, waiting for the SSM to signal **READY**
- One packet stored, transmitting to third device while storing a second packet from the SSM
- etc.

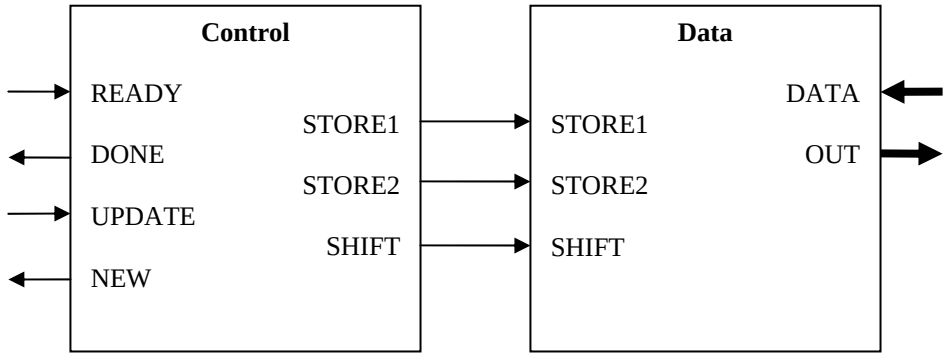
1. Draw a top-level interface diagram of the SRBM



2. Divide your SRBM into two sub-units: a data partition and a controller partition
3. Outline the responsibilities of each of your two sub-units

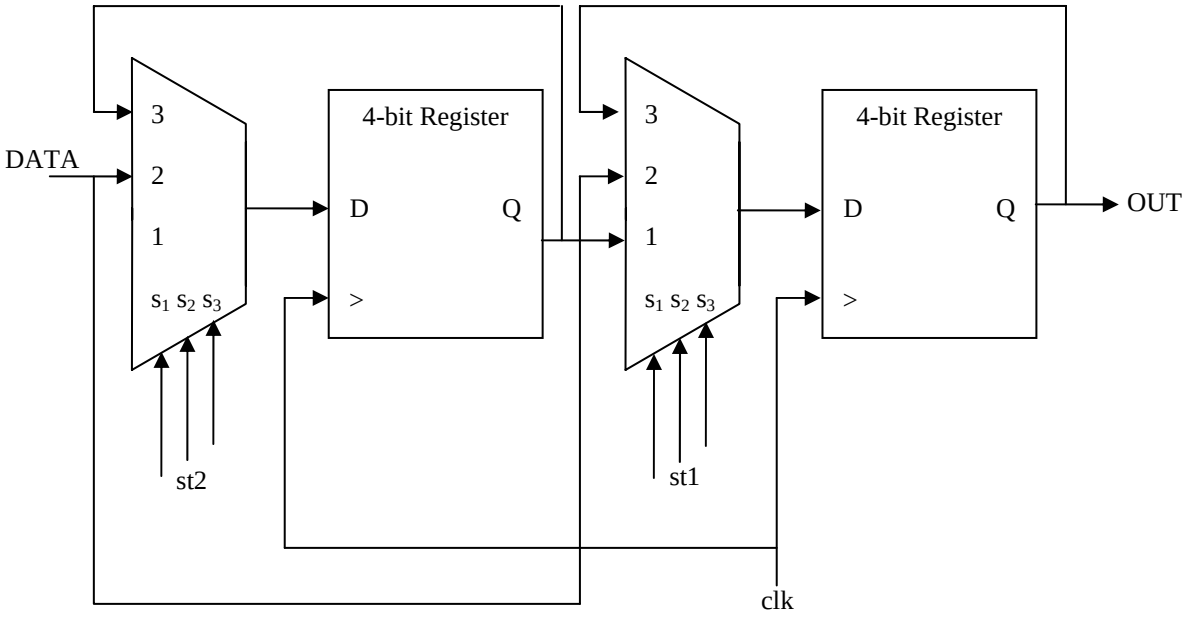
Control Unit	Data Unit
• Interact with SSM via READY and DONE • Assert DONE after DATA is loaded (implies that storage had room for the new DATA) • Interact with "third device" via NEW and UPDATE • Assert NEW when a new DATA is available on OUT • Instruct Data Unit to shift when UPDATE is asserted	• Store new DATA in one of two storage units • Shift DATA out when instructed by the controller

4. Draw interface diagrams for each of your sub-units

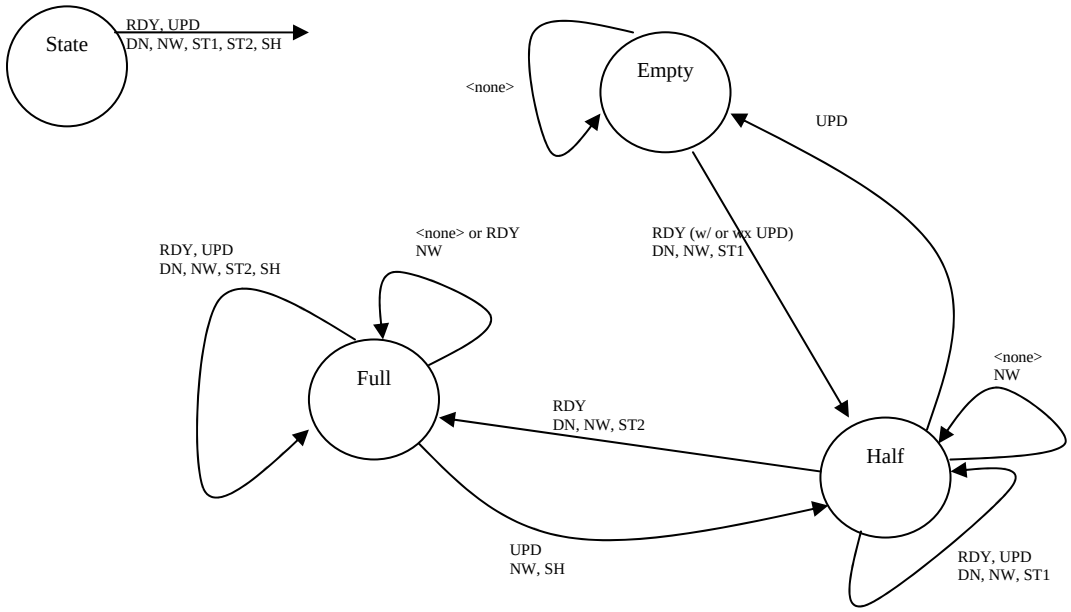


5. Draw lower-middle level schematics of your sub-units. If you plan to use a finite state machine, include a block called FSM in your design – you don't have to go deeper into your schematic.

The data unit is a pair of two 4-bit registers that has shifting capability. The mux's select between feedback and data in.



The controller can be implemented with just a finite state machine. I chose a Meely machine because it has fewer states.



YOU WILL NOT RECEIVE CREDIT FOR ANY OF THE FOLLOWING PARTS IF YOU HAVE NOT FINISHED THE ABOVE PARTS

6. Implement your sub-units in Verilog. Use multi-module design whenever convenient.

srbm.v

```
module srbm( clk, reset, READY, UPDATE, DONE, NEW, DATA, OUT );

input clk, reset, READY, UPDATE;
input [3:0] DATA;

output NEW, DONE;
output [3:0] OUT;

wire [3:0] OUT;
wire STORE1, STORE2, SHIFT;

cu controller( clk, reset, READY, UPDATE, DONE, NEW, STORE1, STORE2,
SHIFT );

du datapath( clk, reset, DATA, STORE1, STORE2, SHIFT, OUT );


endmodule
```

cu.v

```
module cu( clk, reset, READY, UPDATE, DONE, NEW, STORE1, STORE2,
SHIFT );

input clk, reset, READY, UPDATE;
output DONE, NEW, STORE1, STORE2, SHIFT;

reg DONE, NEW, STORE1, STORE2, SHIFT;
reg [1:0] STATE, NEXT_STATE;

parameter EMPTY = 2'b00,
          HALF = 2'b01,
          FULL = 2'b10,
          ERROR = 2'b11;


// state register
always @ ( posedge clk, posedge reset )
   if( reset )
      STATE <= EMPTY;
   else
      STATE <= NEXT_STATE;

// next state decoder
always @ ( STATE, READY, UPDATE )
   case( STATE )
      EMPTY : if( READY )
                 NEXT_STATE <= HALF;
              else
                 NEXT_STATE <= EMPTY;
      HALF  : case( {READY, UPDATE} )
                 2'b01 : NEXT_STATE <= EMPTY;
                 2'b10 : NEXT_STATE <= FULL;
                 default : NEXT_STATE <= HALF;
              endcase
      FULL  : case( {READY, UPDATE} )
                 2'b01 : NEXT_STATE <= HALF;
                 default : NEXT_STATE <= FULL;
              endcase
      default : NEXT_STATE <= ERROR;
   endcase

// output decoder
// (Meely, so a function of state and inputs)
always @ ( STATE, READY, UPDATE )
   case( STATE )
      EMPTY : case( {READY, UPDATE} )
                 2'b10 : {DONE, NEW, STORE1, STORE2, SHIFT} = 5'b11100;
                 default:{DONE, NEW, STORE1, STORE2, SHIFT} = 5'b00000;
              endcase
      HALF  : case( {READY, UPDATE} )
                 2'b00 : {DONE, NEW, STORE1, STORE2, SHIFT} = 5'b01000;
                 2'b01 : {DONE, NEW, STORE1, STORE2, SHIFT} = 5'b00000;
                 2'b10 : {DONE, NEW, STORE1, STORE2, SHIFT} = 5'b11010;
                 2'b11 : {DONE, NEW, STORE1, STORE2, SHIFT} = 5'b11100;
```

```
            endcase
      FULL  : case( {READY, UPDATE} )
                 2'b01 : {DONE, NEW, STORE1, STORE2, SHIFT} = 5'b01001;
                 2'b11 : {DONE, NEW, STORE1, STORE2, SHIFT} = 5'b11011;
                 default:{DONE, NEW, STORE1, STORE2, SHIFT} = 5'b01000;
            endcase
      default : {DONE, NEW, STORE1, STORE2, SHIFT} = 5'b00000;
   endcase

endmodule
```

du.v

```
module du( clk, reset, DATA, STORE1, STORE2, SHIFT, OUT );

input clk, reset, STORE1, STORE2, SHIFT;
input [3:0] DATA;

output [3:0] OUT;

reg [3:0] q2, q1, OUT, muxout1, muxout2;

always @ ( q1 )
   OUT <= q1;

// mux1
always @ ( STORE1, SHIFT, DATA, q2, q1 )
   case( {STORE1, SHIFT} )
      2'b01 : muxout1 <= q2;
      2'b10 : muxout1 <= DATA;
      default : muxout1 <= q1;
   endcase

// mux2
always @ ( STORE2, SHIFT, DATA, q2, q1 )
   casex( {STORE2, SHIFT} )
      2'b01 : muxout2 <= 4'b0000;
      2'b1x : muxout2 <= DATA;
      default : muxout2 <= q2;
   endcase

// reg1
always @ ( posedge clk, posedge reset )
   if( reset )
      q1 <= 0;
   else
      q1 <= muxout1;

// reg2
always @ ( posedge clk, posedge reset )

   if( reset )
      q2 <= 0;
   else
      q2 <= muxout2;

endmodule
```

7. Write a test-bench for your Verilog.

```
module srbm_TB;

reg clk, reset, READY, UPDATE;
wire DONE, NEW;
reg [3:0] DATA;
wire [3:0] OUT;

srbm testUnit( clk, reset, READY, UPDATE, DONE, NEW, DATA, OUT );

always #5 clk <= ~clk;

initial begin
   $shm_open( "srbm.waves" );
   $shm_probe( "AC" );

   clk = 0;
   READY = 0;
   UPDATE = 0;
   DATA = 0;
   reset = 1;

   #7          // takes it off the clock
   reset = 0;

   DATA = 5;
   READY = 1;  // trace path from EMPTY to FULL through HALF

   #50
   DATA = 7;   // test shifting (it should not)

   #50
   UPDATE = 1; // test shifting (it should)

   #50
   READY = 0;  // should go back to empty
               // etc.

   #50
   $finish;

end

endmodule
```

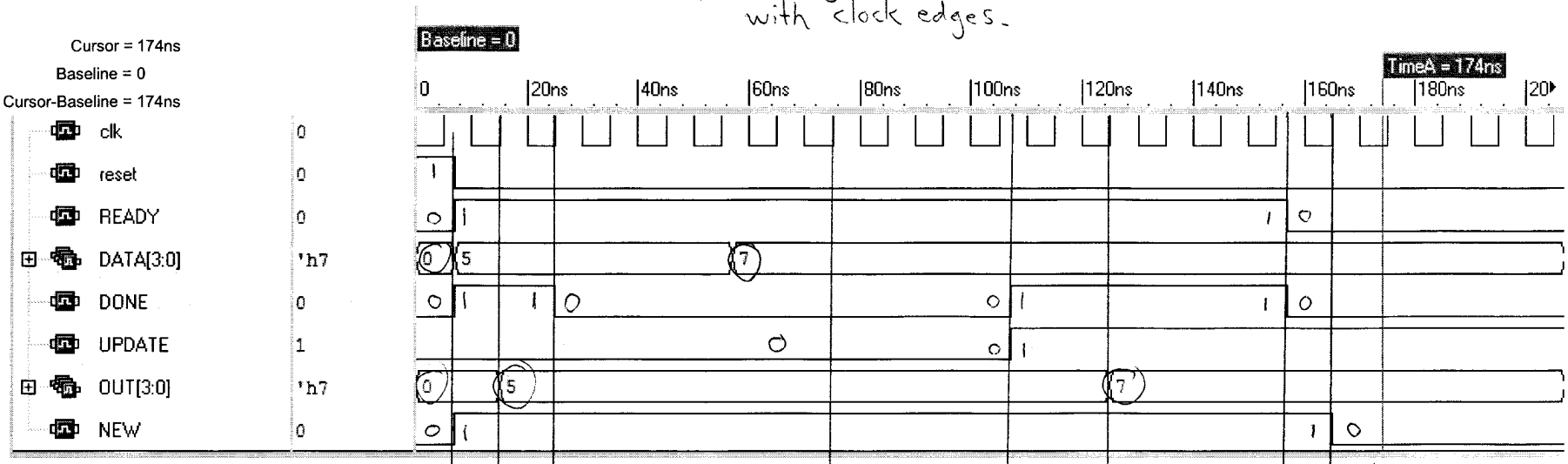
8. Simulate and annotate your work.

next page

ECE 333 Winter 2004 Homework #5 Simulation

Laflen
RHIT

Notice that input changes ARE NOT ALIGNED with clock edges.



reset clears everything

done stops asserting because buffer is full

first clock edge after reset, 5 becomes evident on output

although input has changed 5→7, 7 is not evident on output because UPDATE is LOW

with both UPDATE & READY High, output should shift

at 2nd clock cycle after UPDATE, The 2 stored 5's are shifted out and 7 is now evident on output

ready is LOW, so Done stops (no more data loading)

buffer has emptied, NEW is LOW