Name: _____

MIDTERM

*Closed book except for one 8½ x 11 sheet. No calculators or electronics devices. All work is to be your own - **show your work** for maximum partial credit.*

1) (15pts) Multiple Choice. Circle the most correct choice:

a. If it is critical that an output of your design not glitch, the best way to accomplish this is:
   i. Generate it with a Moore machine instead of a Mealy machine
   ii. Ensure the logic ends with a gate with long inertial delay to filter glitiches
   iii. Ensure the logic ends with a flop
   iv. Any of the above

b. For the verilog shown to the side, choose the most correct statement.

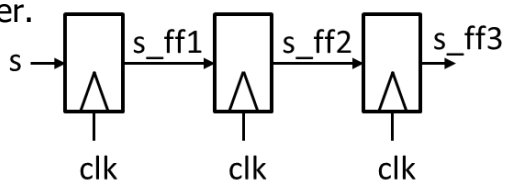
```
always @(posedge clk)
  if (en)
    Q <= #2 D;
```

   i. Will infer a latch because there is no *else* to the *if*
   ii. Models a conditionally enabled flop with a 2 time unit clk2q delay
   iii. Should have *en* in the sensitivity list
   iv. Models a flop with a 2 time unit setup time.

c. An unconnected input signal to a module will:
   i. be assigned "x" by the simulator
   ii. be assigned "z" by the simulator
   iii. likely cause increased current consumption
   iv. ii & iii above

d. Meta-stability:
   i. Occurs when alpha particles impart charge on a node of a latch or flop
   ii. Only exists in Hoffman's mind, and I wish he would shut up about it.
   iii. Is a hazard to be aware of for a UART receiver design
   iv. Can occur when the input of a flop changes coincident with the clock
   v. Can be solved by using a high gain inverter.
   vi. Both iii & iv above

e. Pipeline Flops: (as shown)
   i. Code 1 simulates/synthesizes correct
   ii. Code 2 simulates/synthesizes correct
   iii. Both would simulate/synthesize fine
   iv. Neither are good, should have used blocking statements



code1

```
always_ff @(posedge clk) begin
  s_ff1 <= s;
  s_ff2 <= s_ff1;
  s_ff3 <= s_ff2;
end
```

code2

```
always_ff @(posedge clk) begin
  s_ff3 <= s_ff2;
  s_ff2 <= s_ff1;
  s_ff1 <= s;
end
```

(Multiple Choice continued next page)

Multiple Choice…continued…circle the best answer)

f. A synthesis constraint we need to provide Synopsys for a circuit **output** is:
   i. The drive strength of the gate driving the output
   ii. The capacitive load present at the output
   iii. The input delay of the signal
   iv. All of the above
   v. ii & iii above

g. Verilog modules:
   i. Should be coded only one module per file
   ii. Require at least one input or output signal
   iii. Can use multiple styles (structural, dataflow, behavioral) in a given module
   iv. Should always be coded using the synthesizable subset of Verilog

h. What is the output of this code:

```
initial begin
  $strobe("fstrb = %b",f);
  f = 0;
  f <= 1;
  $display("fdisp = %b",f);
end
```

   i. fstrb = x
      fdisp = 1
   ii. fdisp = 0
      fstrb = 1
   iii. fdisp = 1
      fstrb = 1
   iv. fstrb = 0
      fdisp = 1

i. Regarding reset
   i. Flops with asynch reset are considerably larger than flops without reset.
   ii. The flops that hold current state of a SM don't need to be reset because they will be written to before being used.
   iii. Unlike clock which needs a buffer tree, reset can simply be routed to all flops in a large design.
   iv. Reset should be deasserted on the opposite edge as the active edge of the flops in your design.

j. #0 delays
   i. Force an assignment into the inactive event queue
   ii. Allow a way of making assignments to a signal from multiple **always** blocks
   iii. Are an accepted way of solving race conditions.
   iv. All of the above

2) (4pts) We are to implement a UART transmitter using a 50MHz system clock (**clk**). The baud rate will be 25,000.

a. What value will our ***baud_cnt*** count up to before shifting our ***tx_reg*** shift register?

50000000/25000 = 2000

b. How many bits wide does ***baud_cnt*** need to be?

2^{11} = 2048 which covers 2000

3) (7pts) Design generalized saturation logic that can saturate a M bit number to an N-bit number *(finish off the module below).*

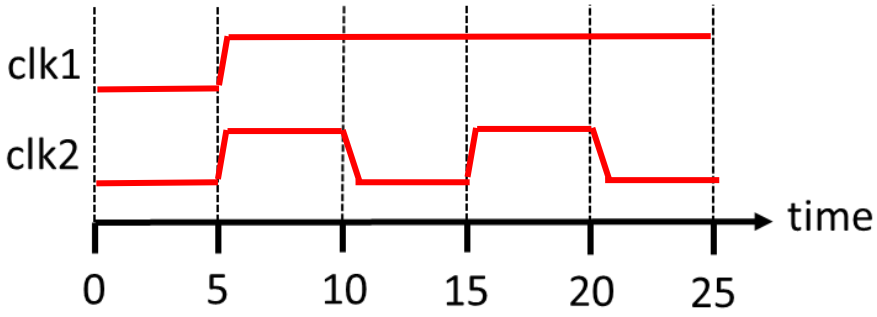
```
module saturation #(parameter M=10, N=5)(data_in, data_out);
  input [M-1 :0] data_in;
  output [N-1:0] data_out;

  assign data_out = (data_in[M-1] & ~&data_in[M-2:N-1]) ? {1'b1,{{N-1{1'b0}}} :
                    (~data_in[M-1] & |data_in[M-2:N-1]) ? {1'b0,{{N-1{1'b1}}} :
                    data_in[N-1:0];

  endmodule
```

4) (5pts) Given the **initial** and two **always** blocks below draw clk1 & clk2 vs time.

```
initial begin
  clk1 = 0;
  clk2 = 0;
end
```

```
always @(clk1)
  #5 clk1 = ~clk;
```

```
always @(clk2)
  clk2 <= #5 ~clk2;
```



5) (5pts) Write a code snippet *(full module syntax not needed)* to infer a circuit which synchronizes an incoming signal to clk and performs both rising and falling edge detection.

(might have to write kinda small)

```
module sync_edge(
  input clk, async_sig,
  output sig_edge
);

  /// Need to double flop for synchronizing, then flop again for
  edge detect ///
  always_ff @(posedge clk) begin
    sig_ff1 <= asynch_sig;
    sig_ff2 <= sig_ff1;
    sig_ff3 <= sig_ff2;
  end

  /// either edge would mean last two samples differ ///

  assign sig_edge = sig_ff2 ^ sig_ff3;


endmodule
```

6) (8pts) Write a self checking test bench for your design of problem 5. Test at least a rising or falling edge.

```
module sync_edge_tb();
```

Did you instantiate your DUT

Initial begin
  Did you initialize stimulus signals
  Did you use events instead of # delays
  Did you create a change on async_sig in?
  Did you wait at least 2 clocks after that for the edge detect to occur?
  Did you perform a selfcheck using === or !==
End

Always
  Did you toggle the clk signal

```
endmodule
```

7) (5pts) Answer the questions posed for the two **case** statements are shown below:

```
casex (exp)
  3'b0xx : $display("case 1");
  3'b10x : $display("case 2");
  3'b111 : $display("case 3");
  default : $display("default case");
endcase
```

What is displayed when **exp** = 3'bxx0 ?

case 1 (x's in case exp are don't care (wildcard) for casex)

What is displayed when **exp** = 3'b1xx ?

case 2 (x's in case exp are don't care (wildcard) for casex)

```
case (exp) inside()
  3'b0xx : $display("case 1");
  3'b10x : $display("case 2");
  3'b111 : $display("case 3");
  default : $display("default case");
endcase
```

What is displayed when **exp** = 3'bxx0 ?

default (x's in case exp are not wildcards for case inside)

What is displayed when **exp** = 3'b1xx ?

default (x's in case exp are not wildcards for case inside)

8) (4pts) Circle the best implementation for a UART shift register ***(2 of 4pts)***

```
always_ff @(posedge clk, negedge rst_n)
  if (!rst_n)
    tx_shft_reg <= '1;
  else if (init)
    tx_shft_reg <= {tx_data,1'b0};
  else if (shft)
    tx_shft_reg <= (tx_shft_reg>>>1);
```

```
always_ff @(posedge clk, negedge rst_n)
  if (!rst_n)
    tx_shft_reg <= '1;
  else if (init)
    tx_shft_reg <= {1'b0,tx_data};
  else if (shft)
    tx_shft_reg <= {tx_shft_reg[7:0],1'b1};
```

```
always_ff @(posedge clk, negedge rst_n)
  if (!rst_n)
    tx_shft_reg <= '1;
  else if (init)
    tx_shft_reg <= {tx_data,1'b0};
  else if (shft)
    tx_shft_reg <= {1'b1,tx_shft_reg[8:1]};
```

```
always_ff @(posedge clk, negedge rst_n)
  if (!rst_n)
    tx_shft_reg <= 9'h00;
  else if (init)
    tx_shft_reg <= {tx_data,1'b0};
  else if (shft)
    tx_shft_reg <= {1'b1,tx_shft_reg[8:1]};
```

For **one** of the options you did not choose explain what is wrong with it ***(2 of 4pts)***
Top left: >>> is arithmetic so could shift in 1 or 0 depending
Top right: shifting wrong direction
Bottom right: We need to initialize tx_shft_reg to all 1's

9) (5pts) Write a line of dataflow verilog that will divide **oper [7:0]** by 8 if **div8** is set, otherwise it will multiply operand by 8. Consider **oper** to be a **signed** number, and result to be 8-bits wide.

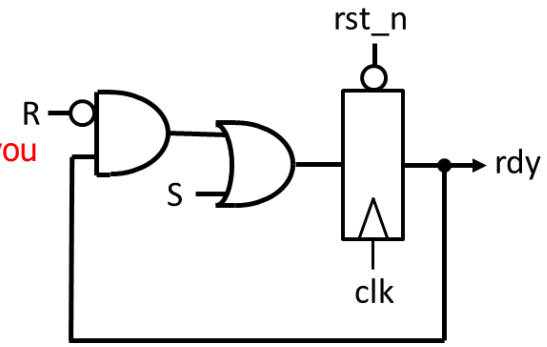
assign result = is <<< a real operation? What is an arithmetic shift left? If you use / or * you had better used $signed to cast 8 to be a signed number.

10) (5pts) Code the circuit shown like a rebel engineer:

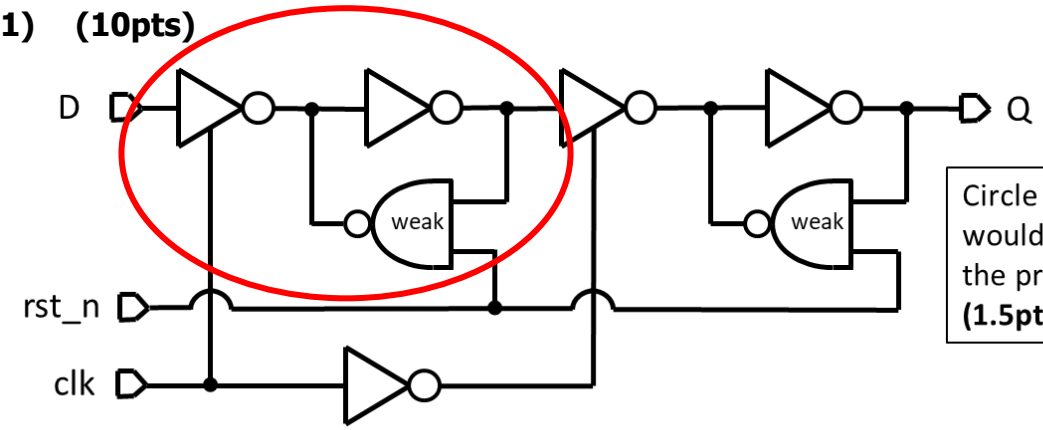
People!! What is this? It is the set/reset flop we Have used several time for done/rdy signals.

I don't care if your code is functionally correct if you Used something like rdy <= S | (~R & rdy)

You were asked to code it like an engineer. Only 15% of the class got this right. Many got -2 for Using: S | (~R & rdy)



11) (10pts)



Circle the gates that would be considered the primary *(first)* latch **(1.5pts)**

Was not too picky, but you should have had the something close to the correct primitive name for the tristate. You should have made some form of attempt to specify the NAND as weak.

Implement the gates of just the primary latch in structural verilog **(4pts)** (assign gate names and node names as necessary)

You know it is a flop right?

```
always_ff @(negedge clk, negedge rst_n)
  if (!rst_n)
    Q <= 0;
  else
    Q <= D;
```

Implement the **behavior** of the entire circuit in behavioral Verilog **(4.5pts)**

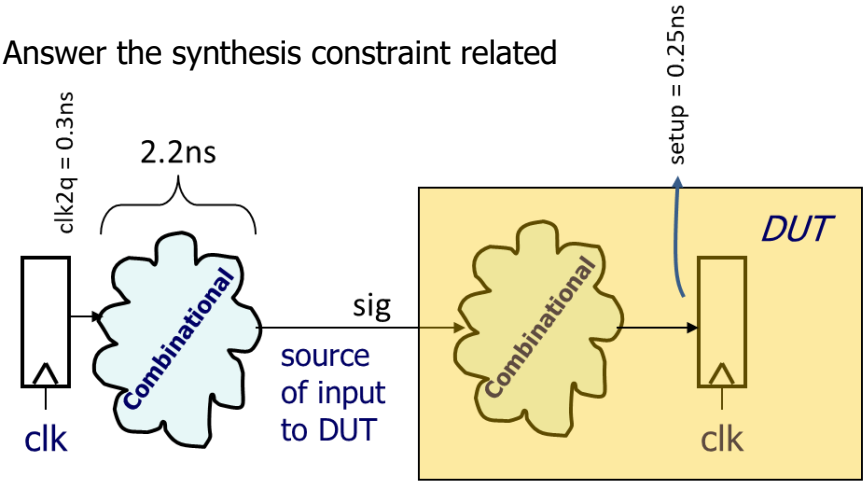
12) (5pts) Draw a schematic of how you think the following Verilog code segment would synthesize

```
module hmm(a,b,f);

  input a,b;
  output f;

  reg f;

always @(a,b)
  if (a&b) f=1;
  else if (a^b) f=0;

endmodule
```

Many drew logic with flops with constants. Valid, but we know that simplifies. Small penalty for not simplifying.

Many saw the fact that if neither a or b was high it had to maintain but drew it as a combinational feedback loop instead of as a latch.

13) (6pts) Answer the synthesis constraint related questions:



Circle the Synopsys command you would use to define a 250MHz system clock:

create_clock –name "sysclk" –period 4 clk

create_clock –name "clk" –period 4 –source_to [find pin sysclk]

create_clock –name "sysclk" –frequency 250M –source_to [find pin clk]

create_clock –name "clk" –frequency 250M clk

Write the **set_input_delay** statement you would for the **sig** input to "*DUT*"

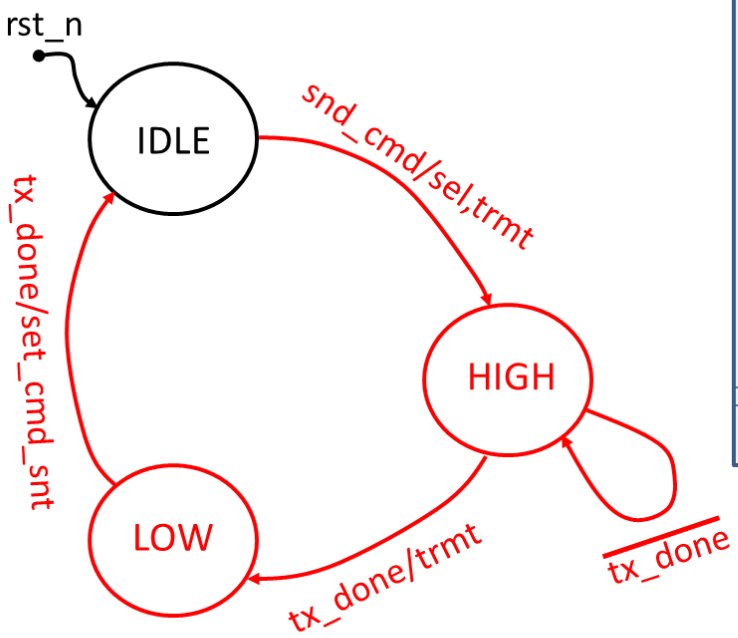
set_input_delay -clock clk 2.5 sig

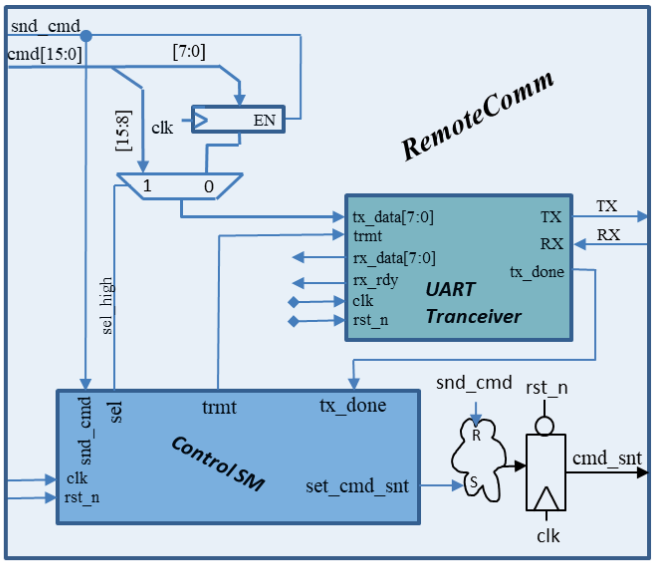
Was really looking for the 2.5 to be correct

What other timing related constraint needs to be applied to input **sig**?

Drive strength of the input. Remember…you are telling it what it does not know.

14) **(16pts)** Design Problem (stay calm…its really not that hard). Look at the image of ***RemoteComm***. This is a unit that will employ an 8-bit UART transmitter to send a 16-bit command as 2-bytes. From the diagram one can see it has an 8-bit register to buffer the lower byte of the 16-bit command. The upper byte does not need to be buffered because it is transferred immediately to the UART upon reception of the **snd_cmd** signal. Once the 2nd byte of the command is transferred **cmd_snt** is asserted. It should be kept high until the next occurrence of **snd_cmd** *(assume we are using the Set/Reset flop style to produce* **cmd_snt***)*





SM Output:	Default:
sel	1'b0
trmt	1'b0
set_cmd_snt	1'b0

(7 of 16 pts) Choose default values for the SM outputs and indicate them in the table. Complete the bubble diagram using proper notation, and only indicate SM output signal when they differ from your chosen defaults.

(9pts of 16) Complete the Verilog for the state machine. (Make Cummings (& Hoffman) Proud)

```
module SM(clk, rst_n, snd_cmd, tx_done, sel, trmt, set_cmd_snt);

input clk, rst_n, snd_frm, tx_done;
output _logic_ sel;
output _logic_ trmt, set_cmplt, clr_cmplt;

typedef enum reg[1:0] {IDLE, HIGH, LOW} state_t;

state_t state, nxt_state;

always_ff @(posedge clk, negedge_rst_n)
  if (!rst_n)
    state <= IDLE;
  else
    state <= nxt_state;

always_comb begin
  sel = 1'b0;
  trmt = 1'b0;
  set_cmd_snt = 1'b0;
  nxt_state = state;

  case (state)
    IDLE : if (snd_cmd) begin
      trmt = 1'b1;
      sel = 1'b1;
      nxt_state = HIGH;
    end
    HIGH : if (tx_done) begin
      trmt = 1'b1;
      nxt_state = LOW
    end
    default : if (tx_done) begin   // same as LOW
      set_cmd_snt = 1'b1;
      nxt_state = IDLE;
    end
  endcase
end
```