What: Mid-Term Exam (For Spring 2012)

Details: Please answer the following questions to the best of your ability. If you need to make any assumptions or you do something that is not explicitly mentioned in the question, please note it on your paper and the rationale for your answer (Failure to do so will result in missing points). This exam should take approximately 75 minutes. The only items allowed during the test are your class notes (hand written), slides from the class, or the textbook that was required for this course (I will only make exceptions for students if they have spoken with me beforehand). No computers are allowed, at all, during the test.

One last note, since you will be hand-writing the answers, your neatness matters. So, if I cannot read your answer, I will not give you points.

Note to Online Students: Please fax (480-965-2929) or email homework (cpd.hwexam@asu.edu) to the GOEE office and we will deliver it to my office (BYENG450) or mailbox.

Question 1 (30 of 160 points possible) (15 minutes): Consider the following Verilog module that uses Euclid’s algorithm to iteratively compute the greatest common divisor of two 16-bit unsigned integer values \( A \) and \( B \) where \( A > B \).

```verilog
module gcd (clk, start, Ain, Bin, answer, done);

input clk, start;
input [15:0] Ain, Bin;
output reg [15:0] answer;
output reg done;

reg [15:0] a, b;

always @(posedge clk) begin
  if (start) begin
    a <= Ain;
    b <= Bin;
    done <= 0;
  end
  else if (b == 0) begin
    answer <= a;
    done <= 1;
  end
  else if (a > b)
    a <= a - b;
  else
    b <= b - a;
end
endmodule
```
Please neatly complete the timing diagram below as the module computes the gcd of 21 (Ain) and 15 (Bin). Use "???” to indicate values that cannot be determined from the information given.

For the “Ain”, “Bin”, and “answer” signals, please represent the values in Verilog’s Hexadecimal format. Make sure to the size, in bits, of the port (not the minimum number of bits required to represent the number) when representing how many bits make up the number.

For the “a” and “b” signals, please represent the number in Verilog’s decimal format. Please use the width of the registers for determining the number of bits required to describe the numbers appropriately.

---

**Question 2 (30 of 160 points possible) [15 minutes]**: Draw a Finite State Machine diagram that detects a sequence within a stream data. The sequence we will be searching for is “1101” in this stream; where the first bit seen is ‘1’, the second bit seen is ‘0’, and so on. When the sequence is detected, we will drive a value of ‘1’ on the output ‘seq_fnd’.
Using formalism Semantics, describe the State-Machine:

\[ S = \{ \text{IDLE}, \text{SXXX1}, \text{SXX10}, \text{SX101}, \text{S1101} \} \]

\[ I = \{ \text{DIN} \} \]

\[ O = \{ \text{seq\_fnd} \} \]

\[ \delta: S \times I \rightarrow S = \{ \]
\[ \text{IDLE} \times 0 \rightarrow \text{IDLE}, \text{IDLE} \times 1 \rightarrow \text{SXXX1} \\
\[ \text{SXXX1} \times 0 \rightarrow \text{SXX10}, \text{SXXX1} \times 1 \rightarrow \text{SXXX1} \\
\[ \text{SXX10} \times 0 \rightarrow \text{IDLE}, \text{SXX10} \times 1 \rightarrow \text{SX101} \\
\[ \text{SX101} \times 0 \rightarrow \text{SXX10}, \text{SX101} \times 1 \rightarrow \text{S1011} \\
\[ \text{S1011} \times 0 \rightarrow \text{SXX10}, \text{S1011} \times 1 \rightarrow \text{SXXX1} \} \]

\[ \lambda: S \times I \rightarrow O = \{ \text{S1011} \times 1 \rightarrow \text{seq\_fnd}, \text{S1011} \times 0 \rightarrow \text{seq\_fnd} \} \]

Using the following module header; complete the design of this Finite-State-Machine. Use the reverse side of this sheet to complete the code.

```verilog
module FSM (input wire clk, input wire rst, input wire din, output wire seq_fnd);

reg seq_fnd;
reg [02:00] state, next_state = IDLE;

parameter IDLE=3'd0, SXXX1= 3'd1, SXX10=3'd2, SX101=8’d3, S1101=3’d4;

always@(posedge clk, posedge rst)
  seq_found <= rst ? (state == S1101);

always@(posedge clk) state <= next_state;

always@
  case(state)
    IDLE : next_state = (din == 1) ? SXXX1 : IDLE;
    SXXX1 : next_state = (din == 1) ? SXX10 : SXXX1;
    SXX10 : next_state = (din == 1) ? SX101 : IDLE;
    SX101 : next_state = (din == 1) ? S1011 : SXX10;
    S1011 : next_state = (din == 1) ? SXXX1 : IDLE;
  endcase
endmodule
```

**Bonus Question (1% of Mid-Term Grade)**

Explain what non-determinism means in terms of Finite-State-Machines. In addition, draw a brand new state diagram illustrating the changes required to make your state-machine deterministic.

There are no changes required. You must identify that this State-Diagram is always deterministic, i.e. for every input, there is a defined transition.
Question 3 (20 of 160 points possible) (10 minutes): Show the values that will be assigned in each assignment to \( r \). Variables \( a \), \( c \), and \( r \) are six-bit registers.

<table>
<thead>
<tr>
<th>and</th>
<th>0</th>
<th>1</th>
<th>( x )</th>
<th>( z )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( x )</td>
<td>( x )</td>
</tr>
<tr>
<td>( x )</td>
<td>0</td>
<td>( x )</td>
<td>( x )</td>
<td>( x )</td>
</tr>
<tr>
<td>( z )</td>
<td>0</td>
<td>( x )</td>
<td>( x )</td>
<td>( x )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>or</th>
<th>0</th>
<th>1</th>
<th>( x )</th>
<th>( z )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( x )</td>
<td>( x )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( x )</td>
<td>( x )</td>
<td>1</td>
<td>( x )</td>
<td>( x )</td>
</tr>
<tr>
<td>( z )</td>
<td>( x )</td>
<td>1</td>
<td>( x )</td>
<td>( x )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>xor</th>
<th>0</th>
<th>1</th>
<th>( x )</th>
<th>( z )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( x )</td>
<td>( x )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>( x )</td>
<td>( x )</td>
</tr>
<tr>
<td>( x )</td>
<td>( x )</td>
<td>( x )</td>
<td>( x )</td>
<td></td>
</tr>
<tr>
<td>( z )</td>
<td>( x )</td>
<td>( x )</td>
<td>( x )</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{a} & = 6'b101010; \\
\text{c} & = 6'b1x0x1; \\
\text{r} & = \& \ a; \to \ 0 \\
\text{r} & = \mid \ a; \to \ 1 \\
\text{r} & = ^\ a; \to \ 1 \\
\text{r} & = \& \ c; \to \ 0 \ (1_{c[0]} \ & x_{c[1]} \ & 0_{c[2]} \ & x_{c[3]} \ & 1_{c[4]} \ & x_{c[5]} ) \\
\text{r} & = \mid \ c; \to \ 1 \ (1_{c[0]} \mid x_{c[1]} \mid 0_{c[2]} \mid x_{c[3]} \mid 1_{c[4]} \mid x_{c[5]} ) \\
\text{r} & = ^\ c; \to \ x \ (1_{c[0]} \ ^\ x_{c[1]} \ ^\ 0_{c[2]} \ ^\ x_{c[3]} \ ^\ 1_{c[4]} \ ^\ x_{c[5]} ) \\
\end{align*}
\]

Question 4 (10 of 160 points) (3 minutes): Explain the difference between a Blocking Statement and a Non-Blocking Statement.

- **A Blocking** statement must be executed before the execution of the statements that follow it in a sequential block.
- **A Non-Blocking** statement allows you to schedule assignments without blocking the procedural flow. You can use the No-Blocking procedural statement whenever you want to make several register assignments within the same time step without regard to order or dependence upon each other. It means that Non-Blocking statements resemble actual hardware more than blocking assignments.

Question 5 (10 of 160 points) (3 minutes): What is the difference between wire and reg?

The wire is used as a means to connect different resources such as modules or combinational logic. It can only be using inside of a module, but never inside of a process. The reg data-type is used to represent synchronous and combinational logic. It can only be used inside of a process.

Question 6 (10 of 160 points) (3 minutes): How do you detect if two 8-bit signals are same (What is the difference between \( == \), \( != \) and \( === \), \( !== \))? Which one would you use for building hardware and which one would you use for simulation?

You use a comparator. This can be done using combinational statements like

**Option 1:** if (A == B)

**Option 2:** if (A === B)

If you are interested in building hardware, Option 1 is the approach that you should use. If you are only simulating the states and checking whether or not a signal has gone to \( x \) or \( z \), you should use Option 2.
Question 7 (10 of 160 points) [3 minutes]: What is meant by inferring latches, how to avoid it?
Latches, in general, refer to an undesired storage element. This usually occurs when combinational logic fails to describe an output value for all possible inputs. To avoid this, we can make sure to use a default state in case statements and always supply an else condition.

Question 8 (10 of 160 points) [3 minutes]: Define the following terms $T_{\text{setup}}$, $T_{\text{hold}}$, and $T_{\text{Clock-2-Q}}$:

$T_{\text{setup}}$ refers to the setup timing requirement of a flip-flop. This is the period of time that the input to the flip-flop must be stable before the clock edge.

$T_{\text{hold}}$ refers to the hold timing requirement of a flip-flop. This is the period of time that the input to the flip-flop must be stable after the clock edge.

$T_{\text{Clock-2-Q}}$ refers to the duration of time that it takes for the input $D$ to be seen on the output $Q$ after the clock edge.

Explain what these terms have to do with metastability:
If we fail to meet setup of hold timing checks, we will encounter metastability. The closer we are to the clock edge, either before or after the clock edge, when we violate the setup and hold timing checks, the longer it will take for the output to stabilize, i.e. clock-to-$Q$ will take longer.
Design a switch-level (PMOS or NMOS) XOR. Now, how do you convert it to XNOR? (Without inverting the output)

\[
Z = (XY) + (\overline{XY})
\]

```
module inv_cmos (input wire A, output wire C_b);
    pmos p1 (C_b, supply1, A);
    nmos n1 (C_b, supply0, A);
endmodule

module nand_cmos (output wire C_b, input wire A, B);
    wire wp;
    pmos p1 (C_b, supply1, A);
    pmos p2 (C_b, supply1, B); wp
    nmos n1 (C_b, wp, A);
    nmos n2 (wp, supply0, B);
endmodule

module and_cmos (output wire C, input wire A, B);
    wire C_b;
    nand_cmos c1 (A, B, C_b);
    inv_cmos c2 (C, C_b);
endmodule

module nor_cmos (output wire C, input wire X, Y,
    wire wp; // connect the series [n|p]mos switches
    pmos p1 (wp, supply1, X);
    pmos p2 (F, wp, Y);
    nmos n1 (F, supply0, X);
    nmos n2 (F, supply0, Y);
endmodule

module or_cmos (output wire C, input wire A, B);
    wire C_b;
    nor_cmos c1 (A, B, C_b);
    inv_cmos c2 (C, C_b);
endmodule

module xor_cmos (output wire out, input wire X, Z);
    wire Y_b, X_b, X and Y_b, X_b_and_Y;
    inv_cmos u1 (Y_b, Y);
    inv_cmos u2 (X_b, X);
    and_cmos u3 (X and Y_b, X, Y_b);
    and_cmos u4 (X_b_and_Y, X_b, Y);
    or_cmos u5 (Z, X and Y_b, X_b_and_Y);
endmodule
```
Question 10 (20 of 160 points) (10 minutes): Write a Verilog model of an SRAM memory device with the following specifications:

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Direction</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Input</td>
<td>9-bits</td>
<td>Address bus</td>
</tr>
<tr>
<td>D</td>
<td>In/Out</td>
<td>8-bits</td>
<td>Data Bus</td>
</tr>
<tr>
<td>CE</td>
<td>Input</td>
<td>1</td>
<td>Active Low Chip-Enable Control Signal</td>
</tr>
<tr>
<td>OE</td>
<td>Input</td>
<td>1</td>
<td>Active Low Read-Enable operation. A read operation occurs when CE and OE valid.</td>
</tr>
<tr>
<td>WE</td>
<td>Input</td>
<td>1</td>
<td>Active Low Write-Enable operation. A write operation occurs when CE and WE valid.</td>
</tr>
</tbody>
</table>
Note that the D pin (Data) is specified as “inout.” We didn’t spend more than a few seconds talking about this. An SRAM device has a data-line that takes inputs with WE and CE are valid. When CE and OE are valid, the SRAM device drives data onto the D pin.

Please examine the following sample code that shows how to create a bi-directional interface:

```verilog
module bidirec (  
  input wire oe, clk,  
  input wire [07:00] inp,  
  output wire [07:00] outp,  
  inout wire [07:00] bidir);  

  reg [07:00] a;  
  reg [07:00] b;  

  assign bidir = oe ? a : 8’bZ;  
  assign outp = b;  

always@(posedge clk) begin  
  b <= bidir;  
  a <= inp;  
end
endmodule
```

```verilog
module SRAM (  
  input wire [08:00] A,  
  input wire CE,  
  input wire OE,  
  input wire WE,  
  inout wire [07:00] D);  

  reg [07:00] memory [511:0];  

assign D = ((OE & CE) & ~WE) ? memory[A] : 8’bz;  

always@(WE, CE, OE)  
endmodule
```