CSE370 Final Exam (12 December 2003)

Please read through the entire examination first! This exam was designed to be completed in 100 minutes (one hour and 40 minutes) and, hopefully, this estimate will be reasonable. There are 4 problems for a total of 100 points. The point value of each problem is indicated in the table below.

Each problem and sub-problem is on a separate sheet of paper. Write your answer neatly in the space provided. If you need more space (you shouldn't), you can write on the back of the sheet where the question is posed, but please make sure that you indicate clearly the problem to which the comments apply. Do NOT use any other paper to hand in your answers.

The exam is CLOSED book and CLOSED notes. Please do not ask or provide anything to anyone else in the class during the exam. Make sure to ask clarification questions early so that both you and the others may benefit as much as possible from the answers.

Good luck and have a great winter break.

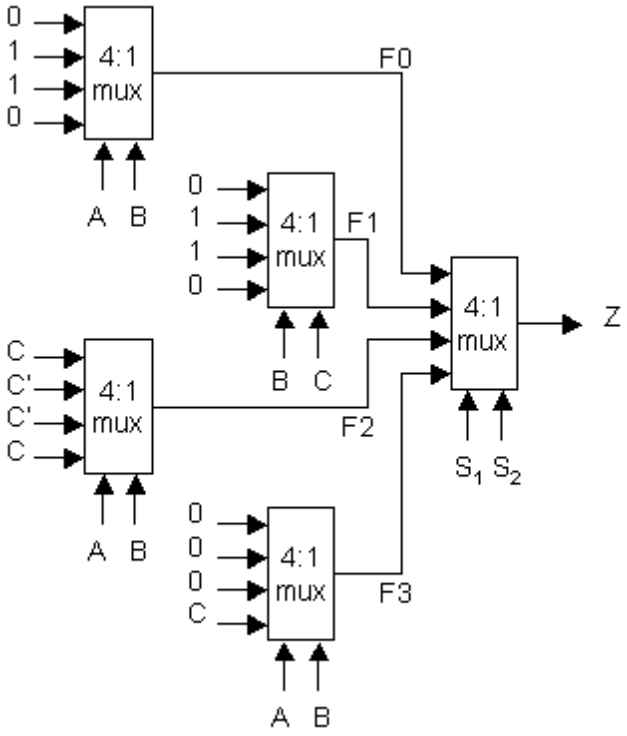
Name:

ID#:

Problem	Max Score	Score
1	15	
2	20	
3	35	
4	30	
TOTAL	100	

1. Boolean Algebra (15 points)

(a – 5 pts) The following circuit implements a Boolean function of 5 variables (A, B, C, S1, and S2). It uses five 4:1 multiplexers. Note that S1 and S2 are simply used to select one of four functions of A, B, and C. Write the equations for these four functions. We'll call them F0, F1, F2, and F3.



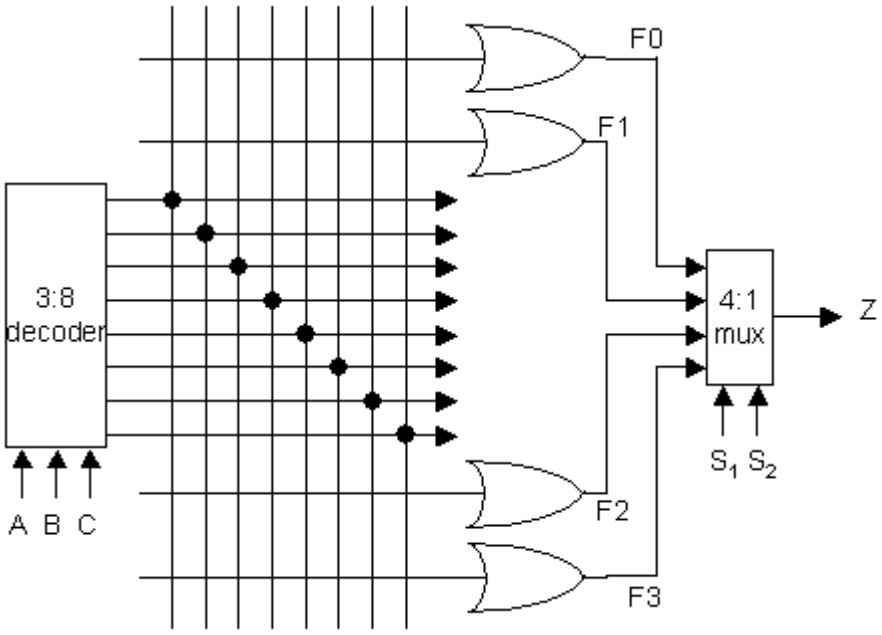
F0 =

F1 =

F2 =

F3 =

(b – 5 pts) We will now use a new kind of programmable logic called a DAL (Decoder Array Logic) to re-implement our logic function. The schematic of a very useful DAL370V4 component is given below. Show which connections need to be made to the OR gates to implement Z (show the connections as an X at the intersection of the OR input line and the decoder outputs).



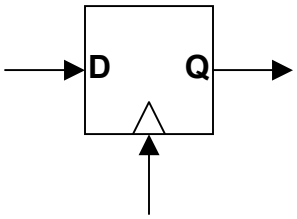
(c – 5 pts) On the other hand, it might be more efficient to implement this function in a ROM. Let's assume we have a 32 word by 1-bit ROM available. Complete the table of ROM contents below (note that only 12 of the 32 words are shown – only complete these 12).

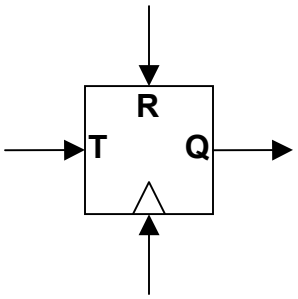
S_1	S_2	A	B	C	Z
0	0	0	0	0	
0	0	0	0	1	
0	0	0	1	0	
0	0	0	1	1	
0	0	1	0	0	
0	0	1	0	1	
0	0	1	1	0	
0	0	1	1	1	
0	1	0	0	0	
0	1	0	0	1	
0	1	0	1	0	
0	1	0	1	1	
…					
1	1	1	0	0	
1	1	1	0	1	
1	1	1	1	0	
1	1	1	1	1	

2. Sequential Logic (20 points)

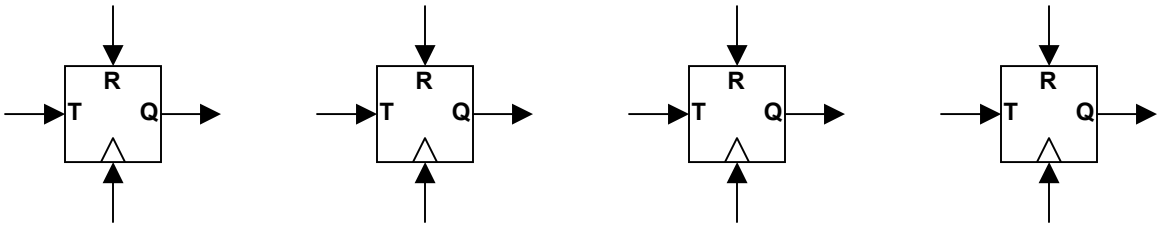
(a – 5pts) You are given a basic edge-triggered D-type flip-flop. Your task is to turn it into a new kind of flip-flop that we'll call a toggle (or T-type) flip-flop. It will have two inputs in addition to the clock signal: R, a synchronous reset signal; and T, the toggle input. When T is high at a positive clock edge, the value of Q will become the opposite of what it was previously. Using only simple gates (OR, AND, NOT, XOR – with as many inputs as you need) around the D-FF, create a T-FF. Fill in the equation for D as well as drawing a circuit around the D-FF.

D = *f*(R, T, Q) =





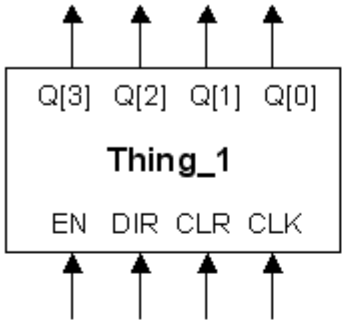
(b – 10 pts) By now, you have your T-FF ready to go. Design a reset-able 4-bit binary up-counter with enable using your new T-FF as the replicated component. Four of them are provided below. Draw your logic around them. Again, you should only need to use simple gates (with any number of inputs). Clearly label: the reset (RESET) signal for the entire counter; 4 output bits (labeled with LSB and MSB from left to right); and the counter's enable (ENABLE) signal. When enable is low, the counter should not change value. When ENABLE is high, it should count to the next binary value.



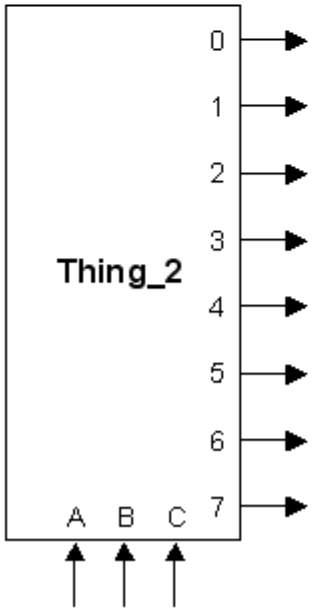
(c – 5 pts) Do you foresee any issues with making this type of counter into a longer counter (say, 32 bits long)? Write a couple of sentences at most.

3. Finite State Machine Implementation (35 points)

For this problem, you will implement a finite-state machine using the parts provided below. However, you'll have to figure out what the parts can do for you by reading their Verilog models. Read these Verilog descriptions carefully.




```
module thing_1 (EN, DIR, CLR, CLK, Q);

input  EN, DIR, CLR, CLK;
output [3:0] Q;
reg    [3:0] Q;

always @(posedge CLK) begin
      if (CLR) Q <= 4'b0000;
      else if (EN && !DIR) Q <= Q + 4'b0001;
      else if (EN && DIR) Q <= Q - 4'b0001;
      else Q <= Q;
end

endmodule
```



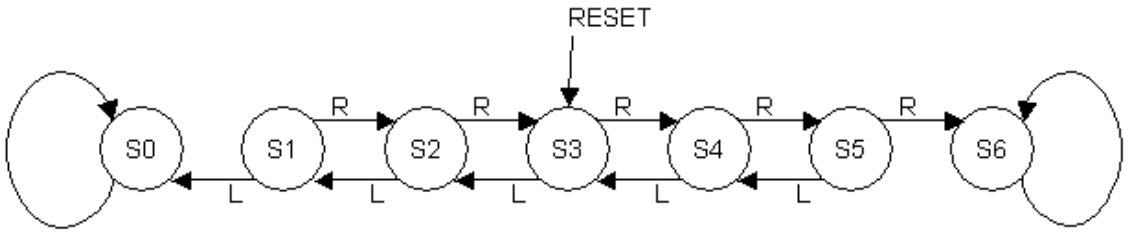

```
module thing_2 (A, B, C, OUT);

input  A, B, C;
output [7:0] OUT;
reg    [7:0] Q;

always @(A or B or C) begin
      case ({A, B, C})
            3'b000: OUT <= 8'b00000001;
            3'b001: OUT <= 8'b00000010;
            3'b010: OUT <= 8'b00000100;
            3'b011: OUT <= 8'b00001000;
            3'b100: OUT <= 8'b00010000;
            3'b101: OUT <= 8'b00100000;
            3'b110: OUT <= 8'b01000000;
            3'b111: OUT <= 8'b10000000;
      endcase
end

endmodule
```

(a – 10 pts) What are the two components you've been provided? Describe the functions implemented by Thing_1 and Thing_2 with at most a couple of sentences for each. Be specific, complete, and relate them to parts you already know.

Thing_1 is a ….

Thing_2 is a ….

(b – 10pts) In one of the lab assignments this quarter, you implemented a finite-state machine to play a simple "tug-of-war" game using two push buttons and seven LEDs. The state diagram for that FSM is given below. You can assume you have the two inputs, L and R, they are high for exactly one cycle whenever the left and right push buttons are pressed.



For this problem, you will re-implement the state machine using the parts provided in part (a). Begin by completing the symbolic state table below. You will need to use Thing_1 to realize your state. First, choose an encoding for state S1 through S6 that will be compatible with using Thing_1, then complete the state table below (using symbolic states). Start by entering the next state for each possible state transition. Then, fill in the values for EN and DIR that you will need to make that state change occur in Thing_1 (make sure to use don't cares). RESET will be connected to the CLR input of Thing_1.

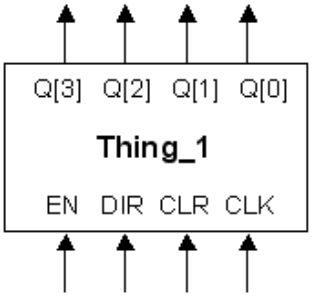
	L	R	Current State	Next State	EN	DIR
	0	0	S0			
	0	1	S0			
	1	0	S0			
	1	1	S0			
S0 =	0	0	S1			
	0	1	S1			
	1	0	S1			
S1 =	1	1	S1			
	0	0	S2			
	0	1	S2			
S2 =	1	0	S2			
	1	1	S2			
	0	0	S3			
S3 =	0	1	S3			
	1	0	S3			
	1	1	S3			
S4 =	0	0	S4			
	0	1	S4			
	1	0	S4			
S5 =	1	1	S4			
	0	0	S5			
	0	1	S5			
S6 =	1	0	S5			
	1	1	S5			
	0	0	S6			
	0	1	S6			
	1	0	S6			
	1	1	S6			

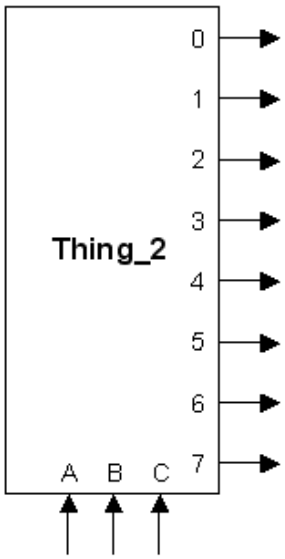
(c – 5 pts) Derive equations for EN and DIR from the table in part (c). You do NOT need to use K-maps, in fact, it is NOT recommended you do so as they have high dimensionality. Instead, exploit the don't cares to come up with the simplest functions you can of L, R, and the current state of Thing_1 (Q3, Q2, Q1, and Q0).

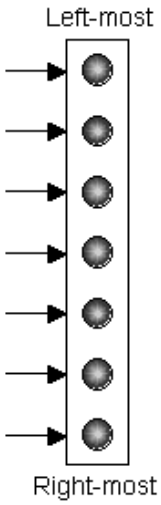
EN =

DIR =

(d – 10 pts) Now, it is time to wrap up our tug-of-war game. Show, how you would wire up the components provided below: Thing_1, Thing_2, and the bank of seven LEDs. Use whatever simple gates you need to realize the functions for EN and DIR you derived in part (c). Clearly label the L, R, RESET, and CLK inputs to your completed circuit. You should only need wires to connect to the inputs and outputs of Thing_2 and the LEDs.

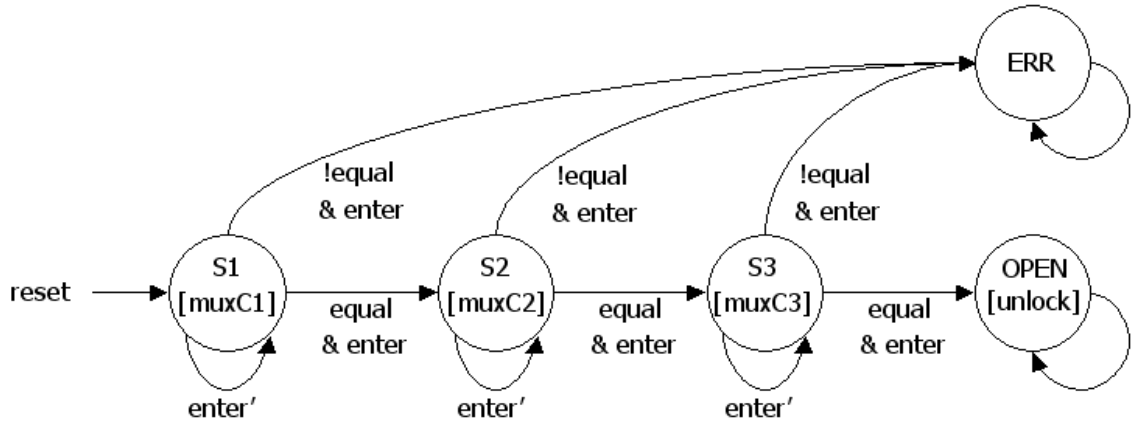




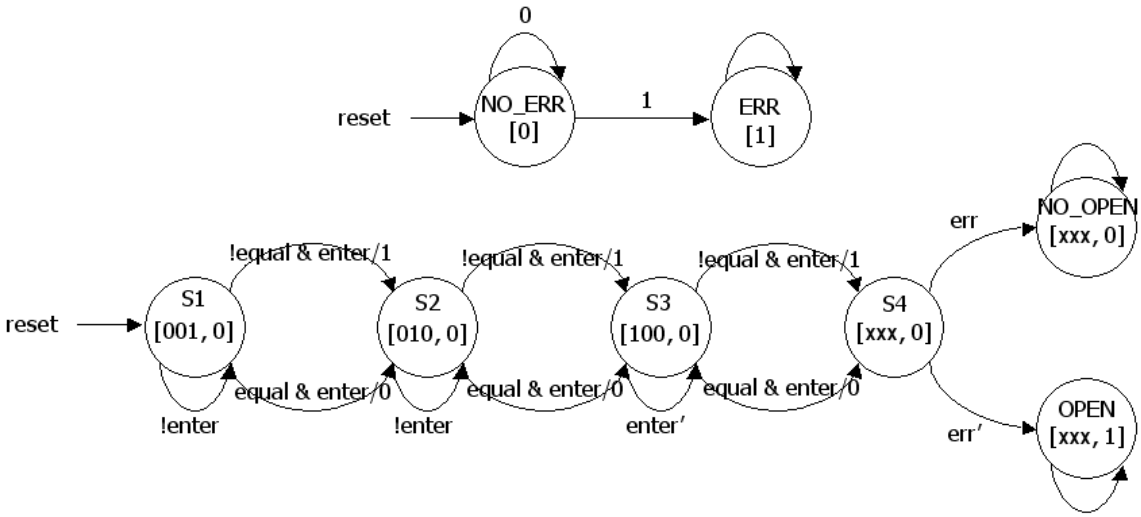


4. Finite State Machine Description (30 points)

The state diagram below is the one we used in lecture to describe a 3-value-key combinational lock.



However, the program code we used did not correspond to this state diagram exactly. A more accurate rendition is the state diagram shown below. Note that it consists of two parallel finite state machines that communicate through two signals (set_err and err). The machine with two states is a Moore machine. Its single bit output is "err". The machine with 6 states is a hybrid Moore/asynchronous-Mealy. Note the Moore outputs consisting of 3 bits to control the comparison "mux" and a single bit for opening the lock ("unlock"). The Mealy output is "set_err".



(a – 5 pts) Below are two symbolic state tables. The one for the 6-state FSM needs to be completed. The one for the 2-state FSM is already completed. Complete the first state table – use symbolic states.

Symbolic state table for the 6-state FSM:

equal	enter	err	Current State	Next State	set_err	mux	unlock
X	0	X	S1	S1	0	001	0
0	1	X	S1	S2	1	001	0
1	1	X	S1	S2	0	001	0
			S2				
			S2				
			S2				
			S3				
			S3				
			S3				
			S4				
			S4				
X	X	X	NO_OPEN	NO_OPEN	X	XXX	0
			OPEN				

Symbolic state table for the 2-state FSM:

set_err	Current State	Next State	err
0	NO_ERR	NO_ERR	0
1	NO_ERR	ERR	0
X	ERR	ERR	1

(b – 10 pts) Select a state encoding for the two state machines:

ERR =

NO_ERR =

S1 =

S2 =

S3 =

S4 =

OPEN =

NO_OPEN =

Explain in a few sentences why you chose the state encoding above.

(c – 5 pts) Complete the Verilog description for the 2-state FSM provided below:

```
module FSM_2 (clk, reset, set_err, err);

input  clk, reset, set_err;
output err;

reg    [ ___ : ___ ] state;

parameter ERR = _______ ;
parameter NO_ERR = _______ ;
```

(d – 5 pts) Complete the Verilog description for the 6-state FSM provided below:

```
module FSM_6 (clk, reset, equal, enter, set_err, mux, unlock);

input  clk, reset, equal, enter;
output set_err, unlock;
output [2:0] mux;

reg    [ ___ : ___ ] state;

parameter S0 = _______ ;
parameter S1 = _______ ;
parameter S2 = _______ ;
parameter S3 = _______ ;
parameter OPEN = _______ ;
parameter NO_OPEN = _______ ;
```

(e – 5 pts) Why was the set_err output of the 6-state FSM made to be an asynchronous Mealy output? Would our combinational lock still function properly if this FSM were realized as a synchronous Mealy machine?