CSE370 Final Exam (16 March 2005)

Please read through the entire examination first! This exam was designed to be completed in 110 minutes (one hour and 50 minutes) and, hopefully, this estimate will be reasonable.

There are 3 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. If you have difficulty with part of a problem, move on to the next one. They are mostly independent of each other.

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 break.

Name: J. Doe

ID#: 0537099

Problem	Max Score	Score
1	25	***25***
2	40	***40***
3	35	***35***
TOTAL	**100**	***100***

1. Combinational Logic (25 points)

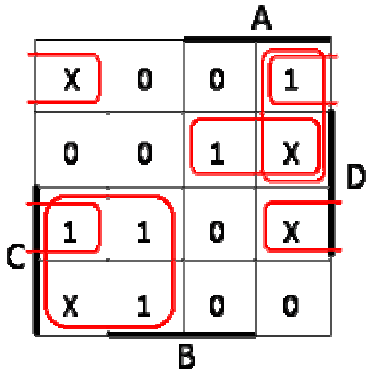
(a – 5 pts) Write the following function in canonical sum-of-products form. Feel free to choose to write terms either with four variables each (e.g., A'B'CD) or in minterm notation (e.g., m_3).

Z = (AC') (B xor D)' + (A'C) (B'D')'

Z = AC'(BD + B'D') + (A'C) (B + D)
Z = ABC'D + AB'C'D' + A'BC + A'CD
Z = ABC'D + AB'C'D' + A'BCD + A'BCD' + A'B'CD
$Z = m_{13} + m_8 + m_7 + m_6 + m_3$
$Z = m_3 + m_6 + m_7 + m_8 + m_{13}$

(b – 5pts) Minimize the function via the K-map method with the addition of don't cares for AB'D and A'B'D'.

Don't cares = $d_0 + d_2 + d_9 + d_{11}$



Z = A'C + AC'D + AB'C'
or
Z = A'C + AC'D + B'C'D'

(c – 5pts) List all the prime implicants for the function of part (b). Which are essential prime implicants?

There are 5 prime implicants: A'C, AC'D, AB'C', B'C'D', B'CD

Two of these are essential to any cover: A'C, AC'D

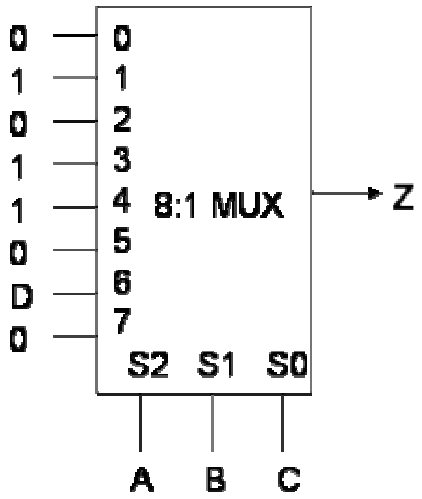
(d – 10pts) Implement your result for part (b) using an 8:1 multiplexer (provided below) and at MOST one inverter. S2 is the most significant bit of the control signals, S0 is the least significant, therefore, S2=1, S1=1, S0=0 selects input #6.

Using Z = A'C + AC'D + AB'C', we can rewrite Z to match the multiplexer equation,

$$Z = A'B'C'(I0) + A'B'C(I1) + A'BC'(I2) + A'BC(I3) + AB'C'(I4) + AB'C(I5) + ABC'(I6) + ABC(I7)$$

$$Z = A'B'C'(0) + A'B'C(1) + A'BC'(0) + A'BC(1) + AB'C'(1+ D) + AB'C(0) + ABC'(D) + ABC(0)$$

$$Z = A'B'C'(0) + A'B'C(1) + A'BC'(0) + A'BC(1) + AB'C'(1) + AB'C(0) + ABC'(D) + ABC(0)$$



2. Finite State Machines (40 points)

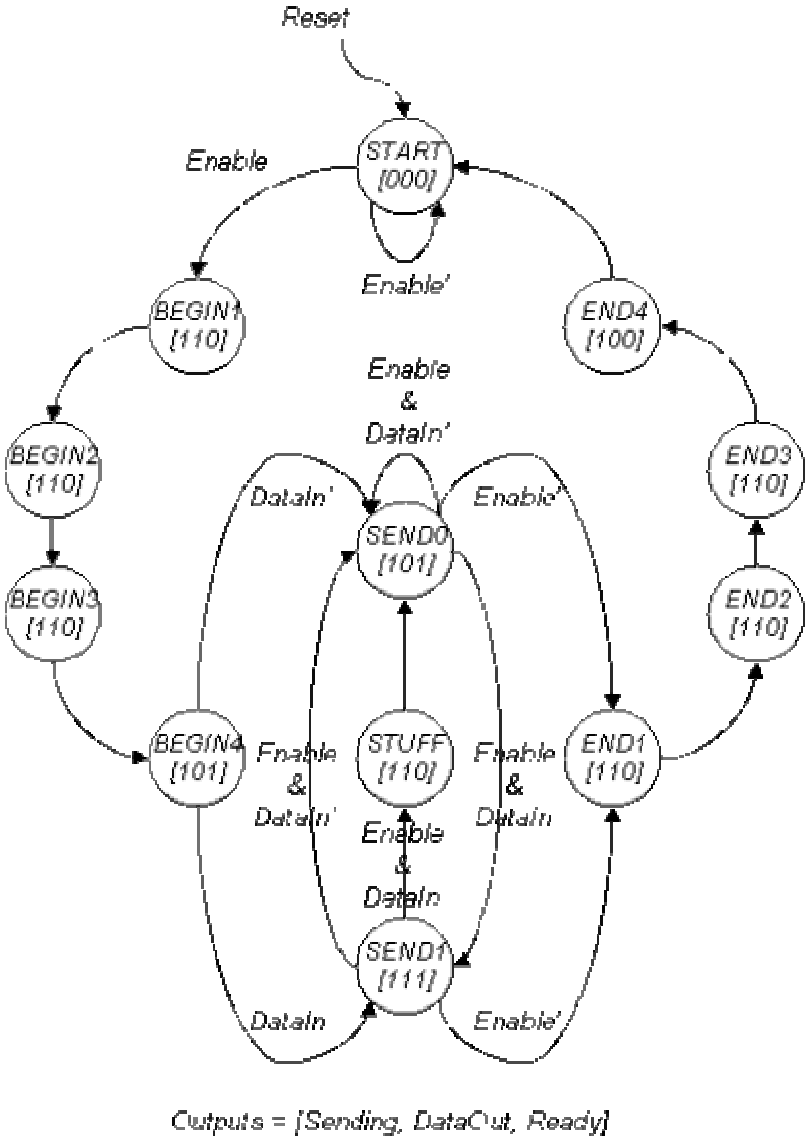
The following Verilog was found among old papers in a dusty drawer of a now defunct dot-com company. Unfortunately, there were no comments in the code.

```
module StuffFrame(Enable, Ready, DataIn, DataOut, Sending, Clk, Reset);
  input      Enable, DataIn, Clk, Reset;
  output     Ready, DataOut, Sending;

  reg  [6:0]  state;
  reg  [6:0]  next_state;

  parameter START   = 7'b0000000;
  parameter BEGIN1  = 7'b1100001;
  parameter BEGIN2  = 7'b1100010;
  parameter BEGIN3  = 7'b1100011;
  parameter BEGIN4  = 7'b1010100;
  parameter SEND0   = 7'b1010101;
  parameter SEND1   = 7'b1110110;
  parameter STUFF   = 7'b1100111;
  parameter END1    = 7'b1101000;
  parameter END2    = 7'b1101001;
  parameter END3    = 7'b1101010;
  parameter END4    = 7'b1001011;

  always @(posedge Clk) begin
       if (Reset) begin state = START;      end
       else       begin state = next_state; end
  end

  always @(Enable or DataIn or state) begin
       case(state)
         START:     if (Enable) next_state = BEGIN1; else next_state = START;
         BEGIN1:    next_state = BEGIN2;
         BEGIN2:    next_state = BEGIN3;
         BEGIN3:    next_state = BEGIN4;
         BEGIN4:    if (DataIn) next_state = SEND1; else next_state = SEND0;
         SEND0:     if (~Enable) next_state = END1;
                    else if (DataIn) next_state = SEND1; else next_state = SEND0;
         SEND1:     if (~Enable) next_state = END1;
                    else if (DataIn) next_state = STUFF; else next_state = SEND0;
         STUFF:     next_state = SEND0;
         END1:      next_state = END2;
         END2:      next_state = END3;
         END3:      next_state = END4;
         END4:      next_state = START;
       endcase
  end

  assign Sending = state[6];
  assign DataOut = state[5];
  assign Ready   = state[4];

endmodule
```

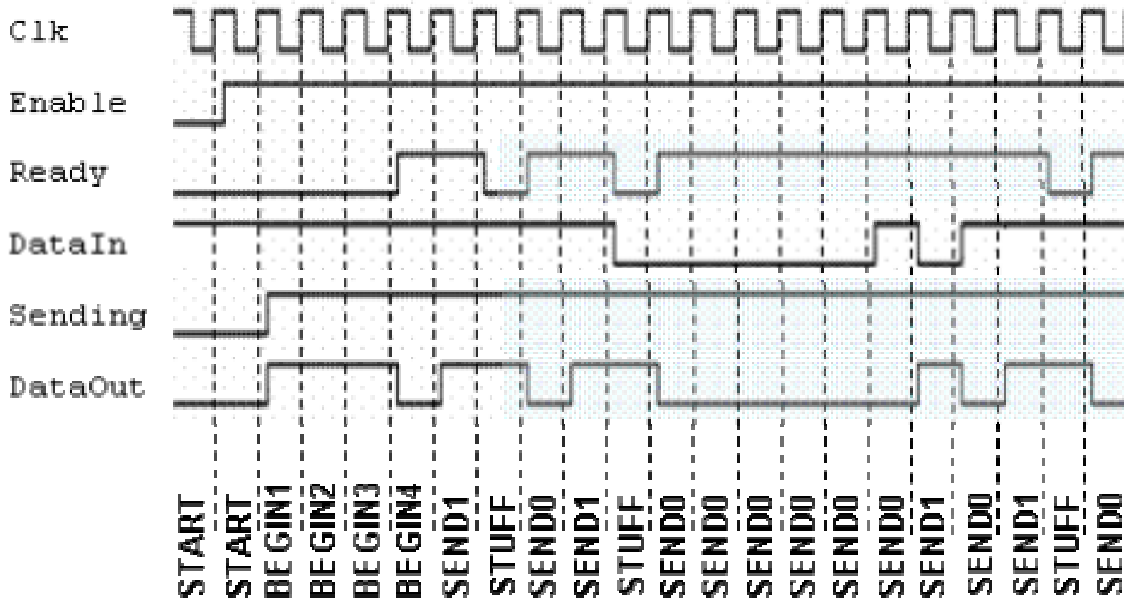
(a – 10pts) What type of state machine is being described (Mealy, Moore, synchronous Mealy)? How many states does it have? Derive its state diagram and clearly label all transitions and the values for the 3 outputs in each state.

This is a Moore machine as the outputs depend only on the current value of the state bits as evidenced by the three assign statements at the bottom of the module.

There are 12 states as evidenced by the 12 parameter statements defining their binary codes and the 12-way case statement in the body of the always block triggered by changes in the inputs or state bits.



(b – 15pts) Simulate the state machine for the following sample input waveforms. Fill in the details for the signals Ready, Sending, and DataOut. Also, please indicate the state the FSM is in for each clock cycle. Assume that the FSM is initially in state "START".



(c – 15pts) We are now faced with the problem of implementing this circuit (the state diagram of part (a)) and our company has a special deal on 4-bit counters and we have to use one in our realization. We'll represent the last four bits of the state variable of our FSM using a counter – note that this state assignment (using only those 4 bits) is enough to uniquely identify each state. Verilog code describing the internals of the counter is shown below. Your job is to use its "clear", "enable", and "load" signals to control the counter's sequence so that it matches the state diagram's transitions. Don't worry about the outputs of the FSM, they will just be functions of the state bits realized by the counter.

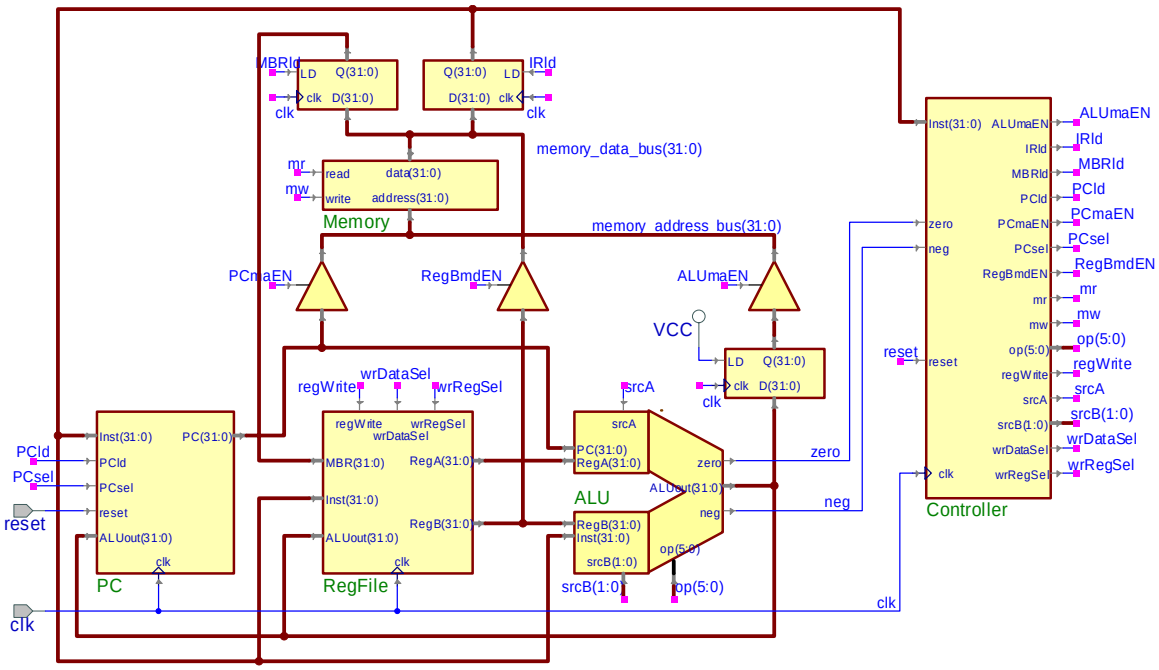
```
module counter(clk, clear, enable, load, datain, value);
  input  clk, clear, enable, load;
  input  [3:0] datain;
  output [3:0] value;
  reg    [3:0] value;
  always @(posedge clk) begin
         if (clear)  value = 0;
    else if (load)   value = datain;
    else if (enable) value = value + 1;
  end
endmodule
```

Complete the following state table for the finite state machine. Make sure to use don't cares as much as possible. The first couple of rows are already filled in as an example.

Inputs				Outputs				
StuffFrame Inputs			Counter Value		Counter Controls			
Reset	**Enable**	**DataIn**	**state[3:0]**	**next_state[3:0]**	**clear**	**enable**	**load**	**datain[3:0]**
1	*X*	*X*	*XXXX*	*0000*	*1*	*X*	*X*	*XXXX*
0	*0*	*X*	*0000*	*0000*	*1*	*X*	*X*	*XXXX*
0	*1*	*X*	*0000*	*0001*	*0*	*1*	*0*	*XXXX*
0	*X*	*X*	*0001*	*0010*	*0*	*1*	*0*	*XXXX*
0	*X*	*X*	*0010*	*0011*	*0*	*1*	*0*	*XXXX*
0	*X*	*X*	*0011*	*0100*	*0*	*1*	*0*	*XXXX*
0	*X*	*0*	*0100*	*0101*	*0*	*1*	*0*	*XXXX*
0	*X*	*1*	*0100*	*0110*	*0*	*X*	*1*	*0110*
0	*0*	*X*	*0101*	*1000*	*0*	*X*	*1*	*1000*
0	*1*	*0*	*0101*	*0101*	*0*	*0*	*0*	*XXXX*
0	*1*	*1*	*0101*	*0110*	*0*	*1*	*0*	*XXXX*
0	*0*	*X*	*0110*	*1000*	*0*	*X*	*1*	*1000*
0	*1*	*0*	*0110*	*0101*	*0*	*X*	*1*	*0101*
0	*1*	*1*	*0110*	*0111*	*0*	*1*	*0*	*XXXX*
0	*X*	*X*	*0111*	*0101*	*0*	*X*	*1*	*0101*
0	*X*	*X*	*1000*	*1001*	*0*	*1*	*0*	*XXXX*
0	*X*	*X*	*1001*	*1010*	*0*	*1*	*0*	*XXXX*
0	*X*	*X*	*1010*	*1011*	*0*	*1*	*0*	*XXXX*
0	*X*	*X*	*1011*	*0000*	*1*	*X*	*X*	*XXXX*

3. Computer Organization (35 points)

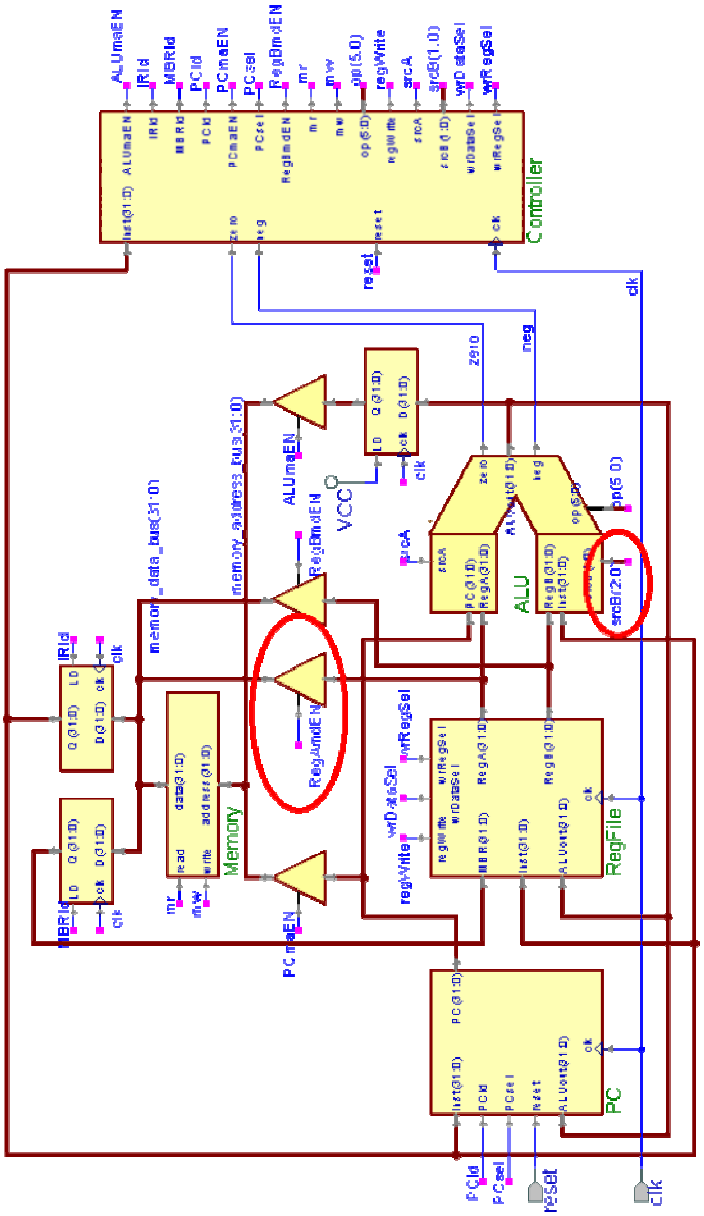
Below is the architecture diagram for the processor of Assignment #9 and #10.



Consider adding a new instruction to those that you implemented: SWI – "store word to immediate". SWI has a 6-bit op-code followed by a register specification (5-bits) and 21 bits that are to be used as the index into memory. The operation is:

```
Memory[21-bit offset] ← RegFile[rs]
```

(a – 10pts) What changes need to be made to the data-path to realize this instruction? Consider changes to RegFile and ALU or even adding additional paths, if needed. You can also feel free to add additional control signals, if needed. Make any changes to the diagram below (just a large version of the diagram on the previous page) with clear annotations. Recall that register "rs" is read into RegA every clock cycle and that the multiplexer on the B input of the ALU can choose between a sign-extended 16-bit offset, 0, 1, or RegB controlled by srcB[1:0].



Add a new tristate driver (controlled by RegAmdEN) to get the output of RegA, where the contents of rs will be delivered, to the memory data bus. Enable the B-side multiplexer on the ALU to extract the 21-bit memory address from the instruction register and pad the upper 11 bits with zeros.

(b – 10pts) Given the changes you specified in part a.1 and ignoring instruction fetch and instruction decode cycles, how many additional cycles do you need to execute this instruction? Specify the value of the control signals for each of these cycles in the table below (including any control signals you may have added or modified). Make sure to specify any don't cares. Using symbolic signal names is fine (as shown in the one entry below).

Control Signal	Value in 1st Execute Cycle	Value in 2nd Execute Cycle (if needed)	Value in 3rd Execute Cycle (if needed)
IRld	*0*	*0*	
MBRld	*X*	*X*	
PCld	*0*	*0*	
PCsel	*X*	*X*	
mr	*0*	*0*	
mw	*0*	*1*	
PCmaEN	*0*	*0*	
ALUmaEN	*0*	*1*	
RegBmdEN	*0*	*0*	
srcA	*X*	*X*	
~~srcB[1:0]~~			
Op[5:0]	"pass B"	*X*	
regWrite	*0*	*0*	
wrDataSel	*X*	*X*	
wrRegSel	*X*	*X*	
RegAmdEN	*0*	*1*	
srcB[2:0]	*"padded 21-bit offset"*	*X*	

(c – 15pts) Consider adding two flip-flops on the “zero” and “neg” signals between the ALU and the controller. How would this affect the processor? Consider the controller size (number of states), performance (minimum clock period), and any changes that may be required of the data-path. Which of the instructions: ADD, SUB, AND, OR, SLT, LW, SW, ADDI, BEQ, J, HALT would be affected by this change?

Adding a flip-flop to each of the “zero” and “neg” lines delays when the values of those signals get to the Controller. This will make certain instructions (SLT and BEQ) that require the result of an ALU operation to wait an extra cycle. No changes are needed in the data-path.

However, since the ALU and Controller delays are no longer in the same cycle, we may be able to run the processor clock with a shorter period and execute more instructions in the same amount of time.