University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Sciences

EECS150 J. Wawrzynek
Spring 2010 May 10, 2010

Final Exam

Name: ______________________________

ID number: ______________________________

This is a *closed-book, closed-note* exam. No calculators or any other electronic devices, please.

Read all the questions **before** you begin. Each question is marked with its number of points (one point per expected minute of time). Although you might not need it, you have 3 hours.

You can tear off the spare pages at the end of the booklet and/or use the backs of the pages to work out your answers. Neatly copy your answer to the places allocated for them.

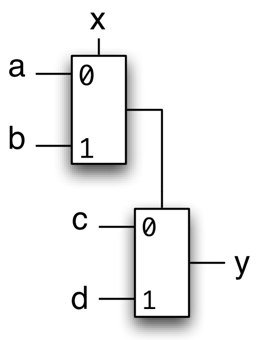
Neatness counts. We will deduct points if we need to work hard to understand your answer. **Simplicity also counts.** In the design problems, correct simpler designs with fewer components will be awarded a higher score than more complex designs with more components.

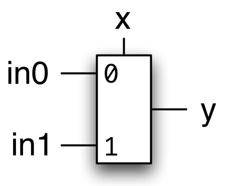
Put your name and SID on each page.

problem	maximum	score
1	6pts	
2	6pts	
3	8pts	
4	22pts	
5	16pts	
6	12pts	
7	10pts	
8	6pts	
9	20pts	
10	8pts	
11	6pts	
Total	120pts	

1. Cascading Multiplexors [6pts].

   Consider the multiplexor circuits show below. Using algebraic manipulation, derive simplified expressions for `in0` and `in1`, that guarantee equivalence of `y` in both circuits. Show your work.





2. Three-value Logic Elements [6pts].

   In circuit simulation, often the value "x" is used to represent unknown signal values. Therefore signal values can take on one of *three* values, "0", "1", or "x". Circuit simulators attempt to propagate 1's and 0's whenever possible, minimizing the propagation of x's.

   Fill in the truth table below to indicate how a circuit simulator should treat each logic gate, given each set of input values. Remember, your goal is to minimize the number of x's propagated.

a b	a OR b	a AND b	NOT(a)	a XOR b
1 1				
1 x				
1 0				
x 1				
x x				
x 0				
0 1				
0 x				
0 0				

3. K-maps [8pts].

Below is the truth table for function f. The symbol "-" is used to represent "don't care".

abcd	f
0000	1
0001	1
0010	-
0011	0
0100	-
0101	1
0110	1
0111	1
1000	1
1001	1
1010	-
1011	0
1100	0
1101	1
1110	0
1111	0

(a) Using a K-map, write a reduced expression for f in products-of-sums (POS) form:

(b) Using a K-map, write a reduced expression for f in sum-of-products (SOP) form:

4. Tree Engine [22pts].

In this problem, you are asked to design a hardware engine for traversing a tree data structure stored in memory. The tree is represented in memory as a collection of nodes. Each node contains three n-bit fields: value holds a two's complement integer, left holds a pointer to the left subtree, and right holds a pointer to the right subtree. All 0's in a pointer field is used to indicate a "null" pointer (no subtree). Trees are stored in a n-bit wide memory block. The fields of a node, value, left, right, are stored in consecutive memory locations. The tree root node always begins at address 0.

Your task is to design the datapath and specify the controller for a hardware block that traverses a tree and outputs the maximum node value (the max over all the value fields of all the tree nodes). Your circuit should output the final max value, (max), and a done signal (done). Your design objective is to minimize the number of clock cycles required for a tree traversal, then the cost.

You are allowed to make instances of the following design blocks:

- Asynchronous read memory with single read port, n-bit wide address (addr), and n-bit wide data output (dout). This block is given to you preinitialized storing a tree.
- n-bit wide register with clock enable (ce), and reset (rst) inputs.
- n-bit wide 2-to-1 multiplexor.
- n-bit wide adder/substractor block with function control (SUB).
- n-bit wide "equal zero" comparator. Takes an n-bit word and outputs a 1 if the input is =0.
- n-bit wide *stack* block, with clock enable (ce), synchronous push/pop control (PUSH), and a empty flag (EMPTY). (For a push operation, set PUSH=1, for pop, set PUSH=0.) The top of the stack is always available on the output dout. The data input din is used for the pushing data onto the stack (when PUSH = 1). EMPTY is an output signal that is =1 when the stack contains no elements.
- Simple logic gates.

(a) In "register transfer language" style pseudo-code, write out the control algorithm. (Remember to use "," to separate transfers that occur on the same clock cycle and ";" for transfers on consecutive clock cycles.)

(b) In the space below, neatly draw your datapath of the tree engine. Circle the control signals—the signals you intend to generate from a controller external to the datapath, or signals from the datapath back to the controller. Label all circuit inputs and outputs.

5. Stack Design [16pts].

   In this problem you will design two versions of the internal details of a *stack* block, also known as a LIFO (last in first out) data structure. This is a synchronous block that on each clock cycle, either *pushes* a word, *pops* a word, or performs *no action*. The block has the following interface:

   - clk: Clock input.
   - Din: Data input.
   - Dout: Data output.
   - push: Active high control for push operation.
   - pop: Active high control for pop operation, take priority over push.

   Stacks are usually built using one of two primary methods. For relatively small stacks, one method is to use a *shift-register* like structure that moves data in one direction on a push operation and the opposite direction on a pop operation. The second method, used for relatively large stacks, uses a *memory block*, similar to how FIFOs are often implemented.

   For parts a) and b) below, you may make instances of the following blocks. Remember to keep things as simple as possible.

   - Simple logic gates,
   - n-bit wide 2-to-1 multiplexor,
   - n-bit wide register with clock enable (ce) and reset (rst).
   - n-bit wide adder/substractor block with function control (SUB).

   (a) In the space below, neatly draw your circuit that uses the *shift register* approach described above, for a 4 element stack.

(b) In the space below, neatly draw your stack circuit using the *memory block* approach described above.

For this part you may used any of the design blocks from part a), plus you must additionally use:

- A single-ported memory block with asynchronous read, n-bit wide address (Addr), and n-bit wide data output (Dout), n-bit wide data input (Din), and a write enable input (we).

6. Finite State Machine [12pts].

Consider the design of a *Moore style* FSM with the following specification. In addition to the clock input (clk), the machine has two inputs (in) and reset (rst), and a single output (out).

When the machine is reset, the output is driven to a 0 and remains at a 0 until the machine recognizes an input sequence of the form:

$$1 * * 1 * * 1$$

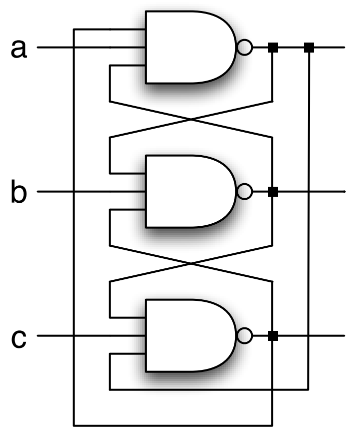
where $*$ could be either 0 or 1. When the desired sequence is seen, on the rising edge after the final 1 of the sequence, the machine sets its output to a 1 for one cycle, then sets it to 0 until another sequence is recognized. *Sequences recognized are non-overlapping.* (Therefore, a sequence such as 1001001001 would be recognized as 1 occurence of the pattern, followed by the first 1 of a new pattern.)

(a) Draw the state transition diagram from the FSM. Label all circles and arcs. Label your states S0, S1, S2, ...

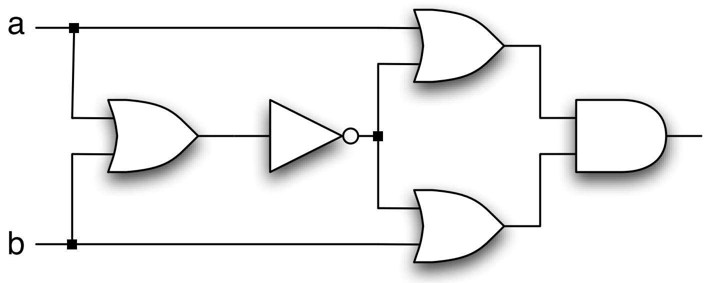
(b) Based on your state transition diagram, draw the gate level circuitry for the FSM, with a *one-hot encoding* of the states. Assume that the flip-flops available to you have only data in (d) and clock (clk) inputs, and data out (q).

7. NAND/NOR circuits [10pts].

(a) Convert the circuit shown below from all NANDs to all NORs (simplicity counts).



(b) Convert the circuit shown below to all NANDs (simplicity counts).



8. Gray-code Counter [6pts].

   A 2-bit Gray counter generates the following sequence of values:

$$..., 10, 00, 01, 11, 10, 00, 01, 11, ...$$

   Derive a counter circuit based on flip-flops and simple logic gates that generates the Gray sequence. A reset signal is not necessary. Show your work and final result.

9. Unsigned Equality Checker Circuit [20pts].

   Consider the design of a circuit that compares two unsigned integers. Specifically, the circuit takes two N-bit unsigned integers inputs, A and B, and generates a single output bit (f) equal to 1 iff $\mathsf{A} > \mathsf{B}$.

   As you know, a substractor circuit can be used for this function. However, a circuit optimized specifically for comparison is simplier. In this problem you will investigate different performance–cost tradeoffs for this optimized comparison function.

   (a) Derive the simplest circuit to achieve this function. This circuit (similar to a ripple adder) will have $O(N)$ cost and delay $O(N)$.

       Draw an instance of this circuit for $N = 4$. Show your work.

(b) Derive a strategy for improving the performance of your circuit to have delay $O(\sqrt(N))$ and $O(N)$ cost. Draw a circuit to illustrate the structure of your circuit for $N = 16$.

(c) Derive a strategy for improving the performance of your circuit from part a) to have $O(log(N))$ delay and $O(N)$ cost. Draw a circuit to illustrate the structure of your circuit, for $N = 8$.

10. Constant Multiplier [8pts].

Consider the design of a circuit for multiplying a constant, C, with a signed two-complement variable, X, such that $Y = C \times X$.

In this problem, let $C = 13_{10}$, and assume X is a 6-bit variable. Using only *full-adder blocks* (1-bit adders), in the space below, draw a multiplier circuit. Your design objective is to minimize the total number of full-adder blocks, as well as the delay from input to output. Give priority to cost over delay.

11. Scheduling [6pts].

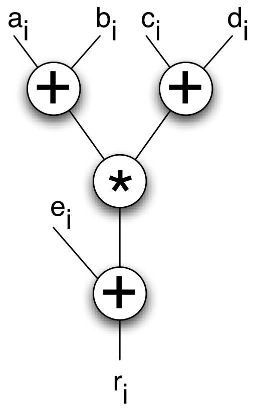
Assume we have a datapath with three arithmetic blocks: two adders (`add1`), and (`add2`), and a single pipelined multiplier (`mult`). The multiplier has two pipeline stages.

The arithmetic blocks are interconnected with a set of switch elements, so that any block can send its output to any other block. Furthermore, the blocks are separated with pipeline registers.

The datapath is connected to a memory block that holds multiple arrays of data values, A, B, C, D and E. Up to 4 data values can be read from the memory on a single clock cycle.

Your goal is to generate the schedule for iterative execution of the computation graph shown below on the datapath described above using as few cycles as possible.

The result value r_i for each iteration of the graph is sent out of the datapath as an output signal (not written to the memory).



Without applying any algebraic manipulation, or modifying the structure of the graph, derive the information needed for scheduling the computation, and illustrate your solution by filling in the schedule chart for four iterations of the computation (generating $r_0, ..., r_3$).

resource	cycle																			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Spare page. *Will not be graded.*

Spare page. *Will not be graded.*

Spare page. *Will not be graded.*

Spare page. *Will not be graded.*

Spare page. *Will not be graded.*