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

EECS 150 — R. H. Katz, Instructor
Spring 2004 — Greg Gibeling, Head TA

SECOND MIDTERM EXAMINATION
Tuesday, 30 March 2004

INSTRUCTIONS—READ THEM NOW! This examination is CLOSED BOOK/CLOSED NOTES. There is no need for calculations, and so you will not require a calculator, Palm Pilot, laptop computer, or other calculation aid. Please put them away. You MAY use one 8.5" by 11" double-sided crib sheet, as densely packed with notes, formulas, and diagrams as you wish. The examination has been designed for 50 minutes/50 points (1 point = 1 minute, so pace yourself accordingly). All work should be done on the attached pages.

In general, if something is unclear, write down your assumptions as part of your answer. If your assumptions are reasonable, we will endeavor to grade the question based on them. If necessary, of course, you may raise your hand, and a TA or the instructor will come to you. Please try not to disturb the students taking the examination around you.

Please refrain from discussing the examination after you have taken it. A small number of students are taking a late examination due to special circumstances.

We will post solutions to the examination as soon as possible, and will grade the examination as soon as practical, usually within a week. Requests for regrades should be submitted IN WRITING, explaining why you believe your answer was incorrectly graded, within ONE WEEK of the return of the examination in class. We try to be fair, and do realize that mistakes can be made during the regarding process. However, we are not sympathetic to arguments of the form "I got half the problem right, why did I get a quarter of the points?"

_____ SID: _____
(Signature)

_____
(Name—Please Print!)

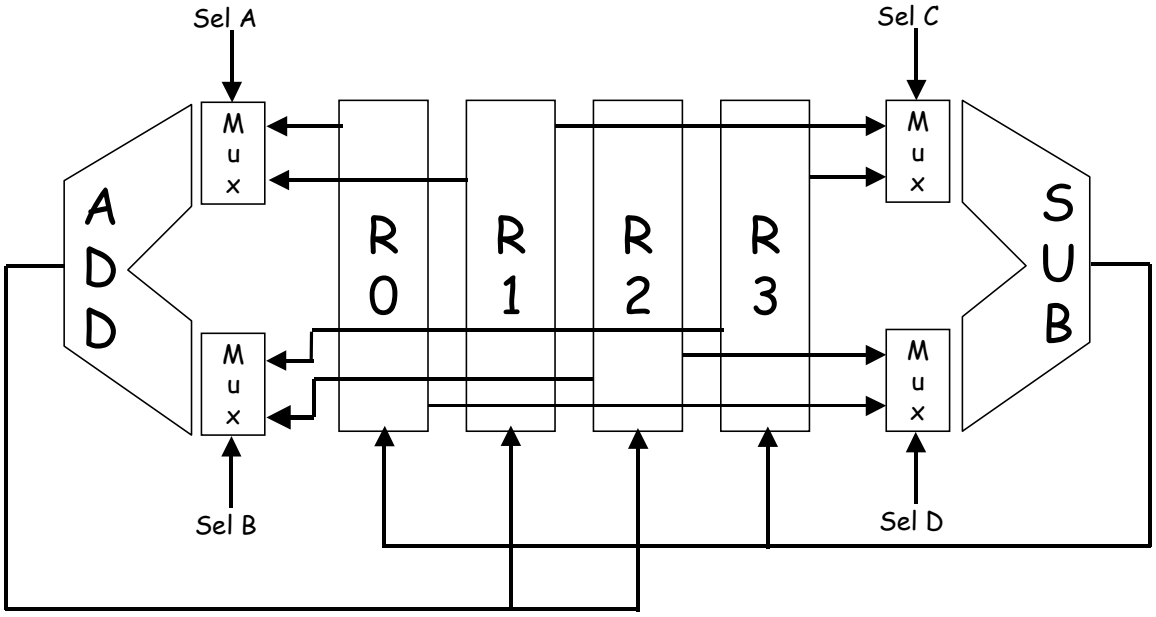
QUESTION	POINTS ASSIGNED	POINTS OBTAINED
1	10	-3
2	20	$\pi^2 + e$
3	20	i
TOTAL	50	$\pi^2 + e - 3 + i$

Question 1. *Verilog* (10 points)
Identify ALL of the errors in the following Verilog description of a two-bit up-counter and the FSM that uses it. This includes errors in the actual text as well as omitted text. Annotate the text with the error type and your correction to it: Each found error is worth 1 point; there are a total of 10 separate errors.

```
module Counter(Clock, Reset, Count);
        input           Clock, Reset;
        output  [1:0]   Count;

        reg     [1:0]   Count;                                          // 1

        always @ (posedge Clock) begin                                  // 2
                if (Reset) Count <=             0;                      // 3
                else Count <=                   Count + 1;              // 4
        end
endmodule

module FSM(In, Out, Clock, Reset);
        input           In, Clock, Reset;
        output          Out;

        wire    [1:0]   Count;

        reg     [1:0]   CurrentState, NextState;                        // 5
        reg             CounterReset;

        Counter         ACounter(Clock, CounterReset, Count);           // 6

        // This capitalization was introduced as the 11th error by MS Word.
        Parameter       STATE_Idle =            2'h0,                   // 7
                        STATE_A =               2'h1,
                        STATE_B =               2'h2;

        always @ (posedge Clock) begin
                if (Reset) CurrentState <=      STATE_Idle;
                else CurrentState <=            NextState;
        end

        always @ (CurrentState or Count or In) begin                    // 8
                NextState =                     CurrentState;
                Out =                           1'b0;
                CounterReset =                  1'b0;                   // 9

                case (CurrentState)
                        STATE_Idle: begin
                                if (In) begin
                                        NextState =     STATE_A;
                                        CounterReset = 1'b1;
                                end
                        end
                        STATE_A: begin
                                if (Count == 2'h3) begin
                                        NextState =     STATE_B;
                                        CounterReset = 1'b1;            // 10
                                end
                        end
                        STATE_B: begin
                                Out =           1'b1;
                                if (Count == 2'h1)
                                        NextState =     STATE_Idle;
                        end
                        default: begin
                                NextState =     2'bxx;
                                Out =           1'bx;
                                CounterReset =  1'bx
                        end                                             // 11
                endcase
        end

endmodule
```

Question 2. *Datapath Design and Register Transfer* (20 Points)

(i) Given the following datapath schematic, write down ALL of the architectural level register transfer operations (e.g., Reg ← Reg op Reg) that can be executed in a *single* processor cycle. The functional unit at the right of the datapath is a subtractor; at the left is an adder. (10 Points)



Write your register transfer operations below:

R0 <- R1 – R2
R0 <- R1 – R0
R0 <- R3 – R2
R0 <- R3 – R0

R3 <- R1 – R2
R3 <- R1 – R0
R3 <- R3 – R2
R3 <- R3 – R0

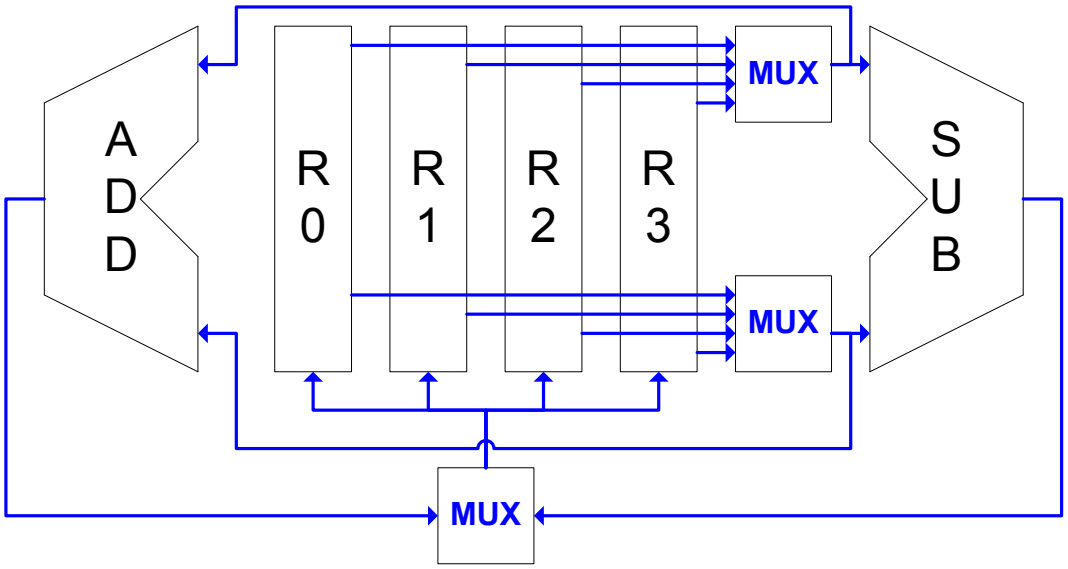
R1 <- R0 + R2
R1 <- R0 + R3
R1 <- R1 + R2
R1 <- R1 + R3

R2 <- R0 + R2
R2 <- R0 + R3
R2 <- R1 + R2
R2 <- R1 + R3

For this problem we were operating under the assumption that we would only ever write to a single register per cycle. Since this is a microprocessor like structure that is a reasonable assumption. We also assumed that each register had a separate write enable signal controlled by whatever (not shown) controller that controlled the muxs.

If you had a different set of assumptions, your answer may well be correct too, provided that you wrote down your assumptions.

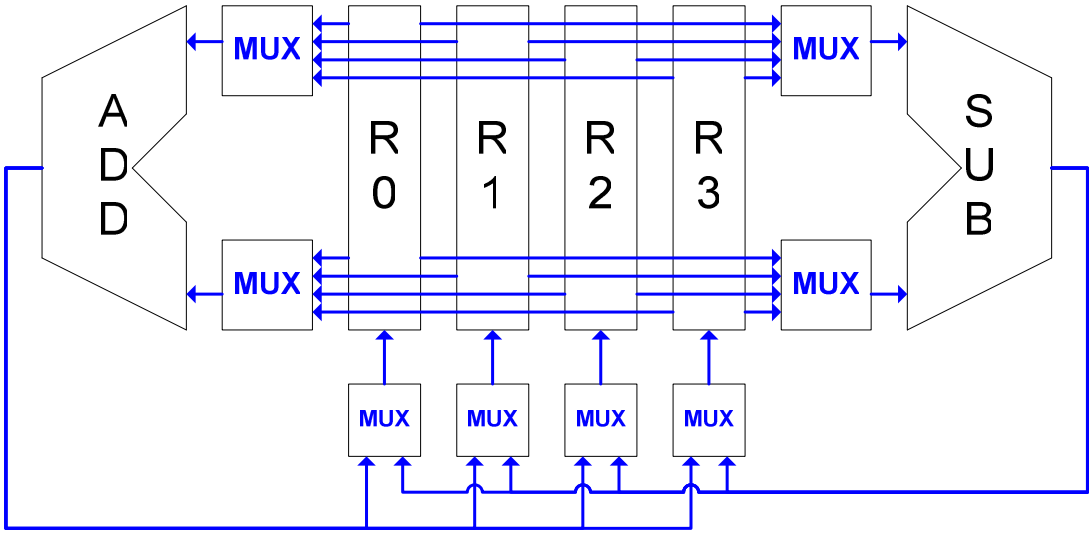
(ii) Starting with the four registers and two functional units (ADD, SUB), design a bussing structure for the datapath so you can implement ANY register-to-register ADD or register-to-register SUB (including the same register used as both sources and the destination). It is *never* the case that both ADD and SUB take place at the same time. Your bussing design must use the fewest possible additional wires to accomplish this task. Draw your bussing structure below. (5 Points)



In this problem, both addition and subtraction actually take place on the same cycle. However, only one of those two results would be used. The problem was loosely worded.

(iii) Revise your solution to (ii) for the case where ADD and SUB *can* occur simultaneously, but never have the same target register (it wouldn't make sense to write something into the same register from two different places at the same time!). Your design must use the fewest possible wires! (5 Points)

Draw your bussing structure below:



Question 3. *Controller Design* (20 Points)
Your task is to design the control for a sequential multiplier. The two operands are called the *multiplier* and the *multiplicand*, and the result is the *product*. It is possible to implement a sequential multiplier using the successive addition method. This is illustrated with the example below for unsigned 4-bit magnitude operands and an unsigned 8-bit magnitude product (e.g., 3 times 4 is 12 in decimal):

Multiplier: 0011
Multiplicand: 0100
Product: 0000 1100

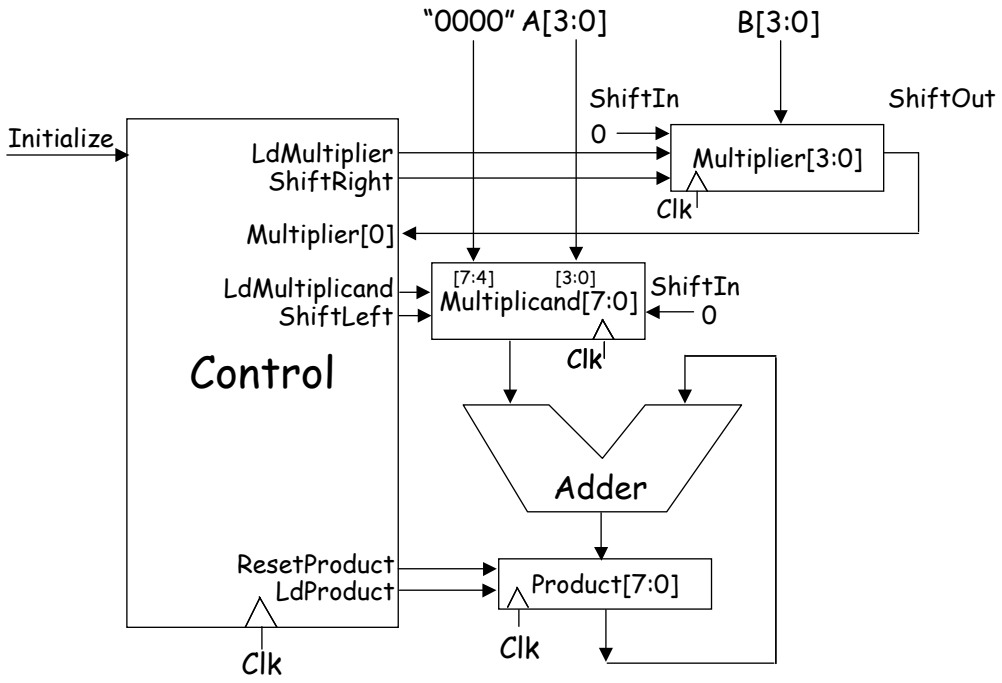
Start with the Product set to 0. The basic strategy is to examine the low order bit of the multiplier. If it is 1, then add the multiplicand to the running Product. Otherwise, don't do any addition. Shift the multiplier one position to the right, and the multiplicand one position to the left. This process repeats four times for a 4-bit Multiplier and Multiplicand. See the cycle-by-cycle results in the table below:

Cycle	Multiplier	Multiplicand	Product
Initialize	0011	0000 0100	0000 0000
Cycle 0, Multiplier[0]=1	0001	0000 1000	0000 0100
Cycle 1, Multiplier[0]=1	0000	0001 0000	0000 1100
Cycle 2, Multiplier[0]=0	0000	0010 0000	0000 1100
Cycle 3, Multiplier[0]=0	0000	0100 0000	0000 1100

High-level pseudocode for the multiplier is as follows:

```
Product = 0
For i = 0 to 3 do
     If Multiplier[0] = 1 then Product = Product + Multiplicand
     Shift right the Multiplier
     Shift left the Multiplicand
```

Given the following datapath, write the *verilog* description for a Moore Machine implementation of the multiplier control on the next page.



(i) Write your Verilog below. When the Initialize signal is true, load A and B into the Multiplicand and the Multiplier, and set the Product to zero. When Initialize is no longer true, commence the computation of the product (15 Points). Please note: you may need or want a counter for your implementation. You may either build one yourself or instantiate one such as the counter given out in lab.

```
module     Control(Clock,     Initialize,     LdMultiplier,     ShiftRight,
                   Multiplier0, LdMultiplicand, ShiftLeft, ResetProduct,
                   LdProduct);
       input       Clock, Initialize, Multiplier0;
       output      LdMultiplier, ShiftRight, Multiplier0, ;
       output      LdMultiplicand, ShiftLeft, ResetProduct, LdProduct;

       wire        Enable;
       wire  [2:0] Count;
       reg         Start;

       Counter     #(3)  ACounter(    .Clock(     Clock),
                                      .Reset(     Start),
                                      .Count(     Count),
                                      .Enable(    Enable));

       always @ (posedge Clock) Start <=         Initialize;

       assign      Enable =                      ~Count[2];

       assign      LdMultiplier =                Start;
       assign      LdMultiplicand =              Start;
       assign      ResetProduct =                Start;

       assign      ShiftRight =                  Enable;
       assign      ShiftLeft =                   Enable;

       assign      LdProduct =                   Multiplier0;
endmodule
```

Of course there are an infinite number of answers to this question, and naturally we expect to see the normal two-always-block format answer on most of the midterms. In fact given that this is such a small counter (only 2 or 3 bits) you might not have used a counter at all, that's just fine too.

Anything that works…

(ii) Determine one way to accelerate the multiplication by taking advantage of special case values of the inputs. Briefly describe how you would change your control to take advantage of the special case you identified. (5 Points).

Any time the multiplier register is all 0s, the multiplication is over. Since none of the subsequent steps would actually involve an addition, the product register already contains the answer. A signal 4-input NOR gate could save you a lot of cycles for some multiplications.

Other valid answers include multiplications by 0 (at the start), or multiplications by 1, both of which are easy to optimize.