# EECS150: Spring 2009, Midterm Solutions

UC Berkeley College of Engineering
Department of Electrical Engineering and Computer Science

## 1 Processor Implementation $(6+3+6=15)$

This problem was worth 15 points. 6 points were given for hex instruction representations (2 points for each instruction). 6 points were given for the datapath addendum (`beq`). 3 points were given for the `beq` hex representation. Solutions which had enough errors to yield negative points were given 0 (instead of a negative score).

### 1.1 Hex Instructions $(2+2+2=6)$

There were three instructions that you had to decode into hex. Each instruction was worth 2 points for a total of 6. Most mistakes lost a single point. Regardless of how many points you lost on one of the three subparts, you could only lose 2 points per subpart (so errors were localized to each subpart).

The subparts, their solutions, and common mistakes were:

`nor $2, $1, $0` $\Longrightarrow$ `0x4440_0000`
Common mistakes:

> `0x40C0_0000`: Register swap (-1).
>
> `0x2140_0000`: Register swap (-1).

`sw $4, 3($5)` $\Longrightarrow$ `0x1618_0003`
Common mistakes:

> `0x1298_0003`: Register swap (-1).
>
> `0x9418_0003`: Register swap & write to `RF[`$RC$`]` (-2).
>
> `0x9488_0003`: Register swap & write to `RF[`$RC$`]` (-2).

`lw $6, 5($7)` $\Longrightarrow$ `0xDC10_0005`
Common mistakes:

> `0xDB10_0005`: Register swap (-1).
>
> `0xF810_0005`: Register swap (-1).
>
> `0x1B88_0005`: Register swap & write to data memory (-2).

Register swapping errors were most common. This datapath sent the contents of $RB$ to the data input port on the data memory and always wrote back to the register file at the address given by $RC$. Bear in mind that solutions which swapped registers and wrote back to the register file at address $RC$ received 0 credit (this was quite common).

As a general rule of thumb, an error in the first 2 hex letters was likely a register swapping issue. A problem with the third hex letter could have been register swap related but was more likely a problem with the `ALUOp`. A problem with the fourth hex letter was most likely writing to data memory when you should not (see the `lw` instruction). **Note that regardless of the value on the input of the write port of the register file, the data will always get written (`WE = 1'b1` always)!** Also note that solutions which set the `*` bit were **not** deducted points.

Aside from the common mistakes listed above, a complete point-loss guide is given below:

**-1** Wrong offset/immed.

**-1** Register swap (some but not all registers were incorrect or swapped).

**-1** Writing bad data to the register file.

**-1** Writing bad data to the data memory.

**-1** `ALUOp` error.

**-2** Shifted bits (garbage hex word).

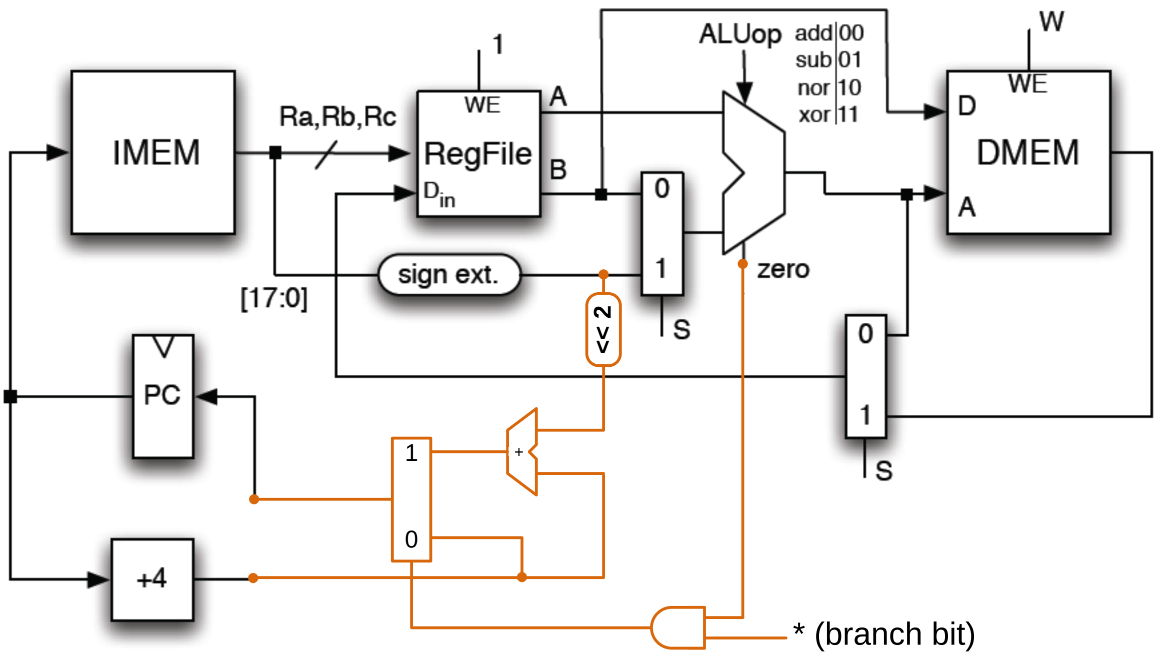
**-2** All registers incorrect (or swapped).

## 1.2 `beq` Instruction (6)

The datapath modifications needed for the `beq` instruction are shown in Figure 1. **Your implementation should have followed the semantics of the `beq` instruction given by the MIPS specification.** Many solutions tried to arbitrarily simplify the solution by assuming that branches didn't take shifted immediates, sign-extended immediates, an implied +4 to the PC, etc. Some other implementations tried to change the semantics of when a branch was supposed to be taken. Because too many of these assumptions overly simplified the problem, they were disregarded. All implementations were graded based on branches performing in the following manner:

$$\text{if } (\$RA == \$RB)\ PC = PC + 4 + (\text{signext}(\text{imm}) << 2)$$

**Figure 1** Problem 1.b: Datapath with `beq` addendum.



Points were deducted as follows:

**-1** Doesn't sign extend the immediate.

**-1** Uses $PC + \ldots$ instead of $PC + 4 + \ldots$.

**-1** Doesn't branch using the `zero` flag out of the `ALU`. For example, some solutions added a comparator circuit to the datapath.

**-2** No use of the `*` bit.

**-2** Branches under incorrect conditions (aside from not using the `*` bit).

**-2** Doesn't shift branch offset by 2 to account for byte addressing.

**-2** Shifts the offset right instead of left.

**-2** Shifts the branch target address after the $PC = PC + 4 + \ldots$ addition.

**-2** No `mux` into the `PC` register to avoid loading the branch target into the `PC` register when no branch happens.

**-2** Includes the `mux` (mentioned directly above) but places it after the `PC` register and after the fork to the +4 block. This error causes the next instruction after the branch to get loaded correctly, however, every instruction after the next instruction will be incorrect (think about why this is the case!).

**-2** No adder to add the offset to the current `PC` (loads the `PC` register with the offset directly).

### 1.3 `beq` Hex Instruction (3)

Your final task was to encode the `beq` instruction from Section 1.2 as a hex word (like in Section 1.1). The instruction, along with the solution (given the datapath shown in Figure 1) is as follows:

`beq $1, $2, skip` $\Longrightarrow$ `0x0524_0002`

Common mistakes/notes:

`0x08A4_0002`: Register swap (-0). Either ordering for $RA$ and $RB$ was accepted for this problem.

`0x2XXX_0002`: Register swap such that you write to `RF[`$RC$`]` (-1).

`0x4XXX_0002`: Register swap such that you write to `RF[`$RC$`]` (-1).

**Important:** Since many people made assumptions in Section 1.2 that may have overly simplified that Section, but not degraded from the difficulty of this Section, no points were deducted for an incorrect hex word that matched your solution to Section 1.2. For example, if your solution to Section 1.2 didn't add +4 to the offset, your hex word may have been `0xXXXX_XXX3` or similar. No points were deducted for this type of error, assuming your implementation matched in Section 1.2. **If your solution to Section 1.2 did not match your solution to this Section, points were deducted based on the rubric given in Section 1.1.**

## 2 LUT Implementation $(10 + 2 = 12)$

This problem was worth 12 points. 10 points were given for the LUT implementation. 2 points were given for labeling the appropriate nodes in the circuit with the correct configurational value.

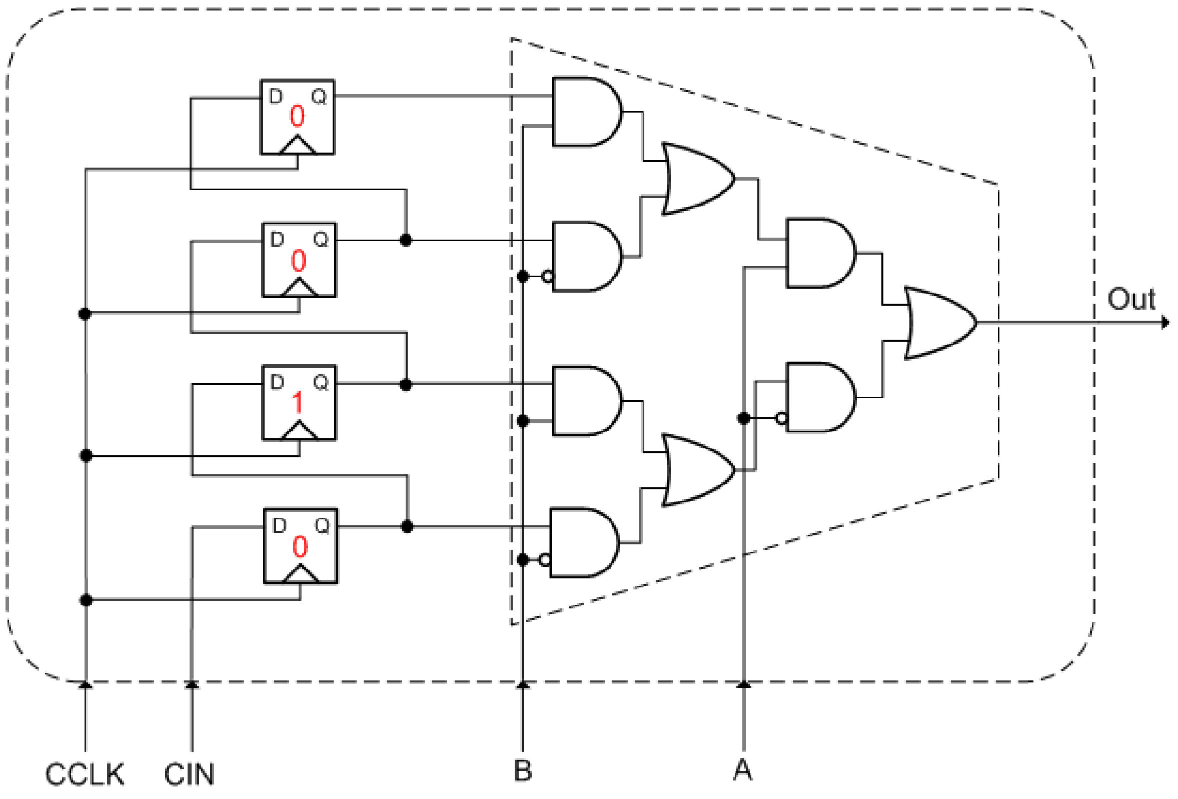
### 2.1 LUT Implementation (10)

A correct answer is shown in Figure 2.

The points were assigned as follows. Out of the 10 points, 4 points were allocated for the multiplexer, 4 points were allocated for the configuration shifter, and 2 points were given for having the correct overall structure of the LUT.

**+4 Shifter** 2 points were given for some kind of a flip-flop structure able to load and store configuration bits. The other 2 points were for having the correct shift register implementation.

**Figure 2** Problem 2: LUT implementation. A shift register structure on the left is used to shift in the configuration bits and the multiplexer on the right selects between which register to output. The LUT inputs A and B are connected to the selector inputs of that mux, which "looks up" what the output should be.



**+4 Multiplexer** 2 points were given for implementing a functionally correct multiplexer. The other 2 points were given if your implementation was optimal.

**+2 Structure** Given for a correct overall LUT architecture, with no extra circuitry besides just the shifter to load in configuration inputs and the multiplexer to select the output

A common mistake was that many people implemented a functionally correct multiplexer, but was clearly suboptimal. For this, they would only receive 2 of the 4 points allocated for the multiplexer implementation. Another common mistake was that many people added another set of flip-flops in addition to the configuration shifter and thus did not receive the 2 points allocated for the overall LUT strucure.

## 2.2 LUT Programming (2)

The answer is marked in red in Figure 2. To program a LUT, all you need to do load the flip flops with the output values of the truth table. So, in the case of $\overline{A} \cdot B$, you will need to load the flip flops with the values 0, 1, 0, 0.

Points were assigned as follows:

**+1** For labeling the diagram in some meaningful way representative of the truth table but made a mistake somewhere

**+2** For labeling the diagram correctly

Many people switched the location of the "1" and received only 1 point for this part.

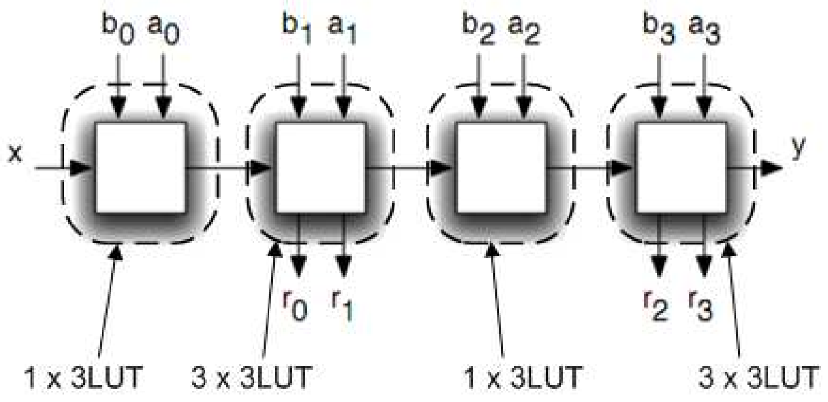
## 3 LUT Mapping $(6 + 6 = 12)$

This problem was worth 12 points, divided equally between the mapping for cost and mapping for delay sections.

### 3.1 Mapping for Cost (6)

The optimal choice here was to use **3 input LUTs**. See Figure 3 for how the design was partitioned into 3-LUTs.

**Figure 3** Problem 3a: Mapping the design into 3-LUTs. Remember that each LUT only has 1 output. So, the blocks with 3 outputs take 3 3-LUTs to map.



We can see that it takes a total of 8 3-LUTs to implement this design. The cost of each 3-LUT is $2^3 + 3 = 11$. The total cost of this then is just $8 \cdot 11 = 64$.

Points were assigned as follows:

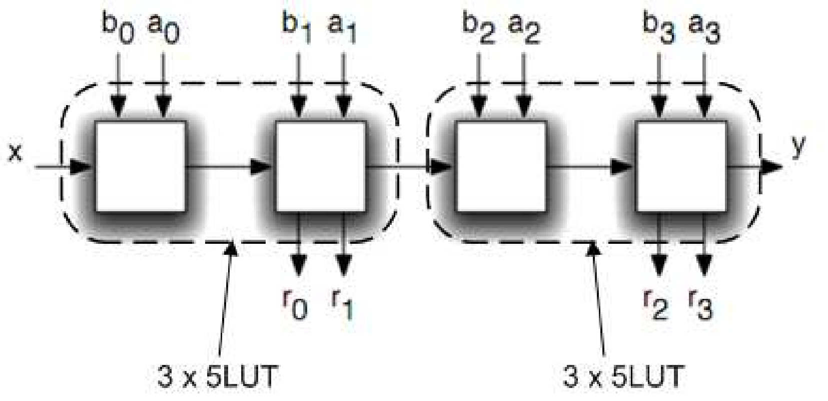
**+3** For mapping the design into 3-LUTs correctly

**+3** For calculating the total cost correctly

### 3.2 Mapping for Delay (6)

The optimal choice here was to use **5 input LUTs**. See Figure 4 for how the design was partitioned into 5-LUTs.

**Figure 4** Problem 3b: Mapping the design into 5-LUTs



We can see that the critical path through our design is from inputs `x`, `b0`, `a0`, `b1`, `or` `a1` to the output `y`, with a delay of two 5-LUTs. Each 5-LUT has a delay of 5, so the total delay is just $5 \cdot 2 = 10$.

Points were assigned as follows:

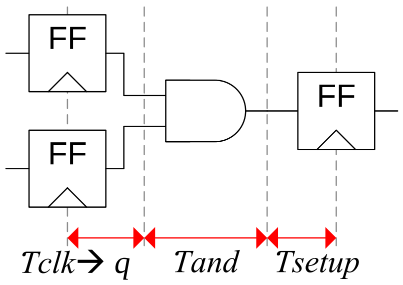
**+3** For mapping the design into 5-LUTs correctly

**+3** For calculating the total delay correctly

## 4 Pipelining (12)

The problem called for the minimization of $(ClockPeriod * NumberOfFlip - Flops)$. Consider first how to reduce the clock period to a minimum. Injecting a flip-flop between each gate clearly provides the shortest critical path (One AND gate, surrounded by flip-flops).

**Figure 5** Problem 4: Minimal Critical Path.

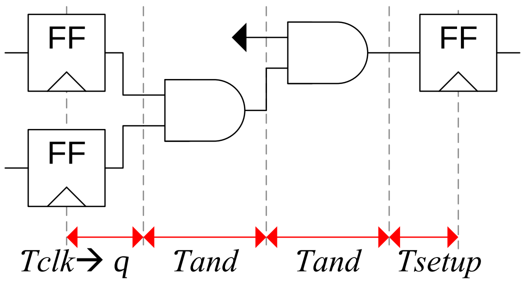


To compute the maximum clock period given the arrangement of flip-flops described above, we must consider the given timing characteristics of the system: **clock to Q delay**, **AND gate delay**, and cenbflip-flop setup time. Clearly, given the critical path described above, our clock period is limited by the sum of **clock to Q delay**, a single **AND gate delay**, and the cenbflip-flop setup time. The minimum clock period achievable via pipelining is thus 3 units of time.

This arrangement, however, comes at a cost of a large number of added flip flops. In fact, the total cost of the system is 31 flip-flops, resuling in a high flip-flop count and clock period product (93).

We can lengthen the critical path to reduce the number of flip-flops required to pipeline the system. Consider a the addition of 4 flip-flops throught the middle column of the ciruict.

**Figure 6** Problem 4: Correct Critical Path.

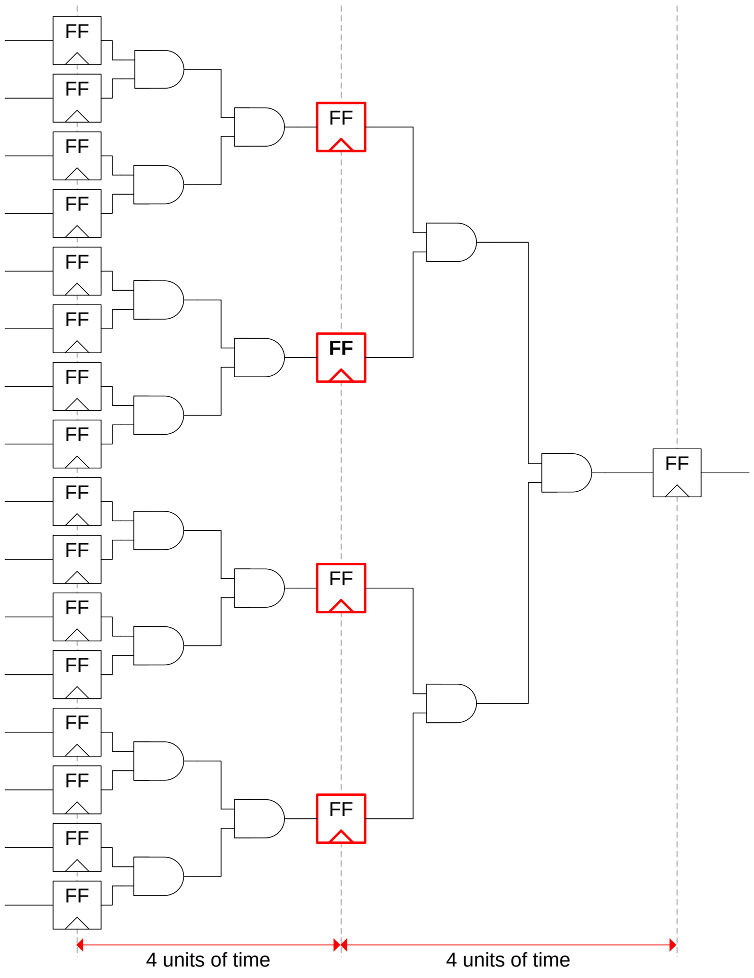


This pipeline has a critical path of two levels of AND gates, surrounded by flip-flops. The clock period must be increased to 4 units of time, but the overhead in flip flop count is reduced. Only 21 flip-flops are needed, reducing the product of the flip-flop count and clock period to 84.

It can be easily shown that a longer critical path results in an increase in the product goal of optimization. The solution is given below:

Clock Period [4]

**Figure 7** Problem 4: Solution.



Flip Flop Cost [21]

A solution could earn up to a maximum of 12 points.

**+3** Clock Period reflects the placement of flip-flops on the diagram.

**+3** Clock Period is correct (4).

**+2** Clock Period is **incorrect** only due to **visible** arithmetic error.

**+3** The diagram clearly shows the addition of 4 flip flops to the diagram, all in correct locations.

**+3** Flip-flop cost is correct (21).

**+2** Flip-flop cost is **incorrect** only due to **visible** arithmetic error.

## 5 Memory Cascade (12)

The problem calls for the construction of a 3K x 2 single-port memory using gates and 1K x 1 blocks. The problem allows the use of flip-flops, but these primitives are not necesary for the task at hand.

Hierarchy is a powerful tool useful in almost any design problem. Consider a lesser task of constructing a 3K x 2 block using 3K x 1 blocks (we will go over the implementation of one next). We are making a wide memory out of a narrow memory of equal depth. To accomplish the task, we simply write to both memories, read from both memories, and concatenate the result. In other words, we use the two memories side-by-side to effecively create a wider memory. Some optimization is possible via sharing of the address decoding logic used by the two 3K x 1 memories.

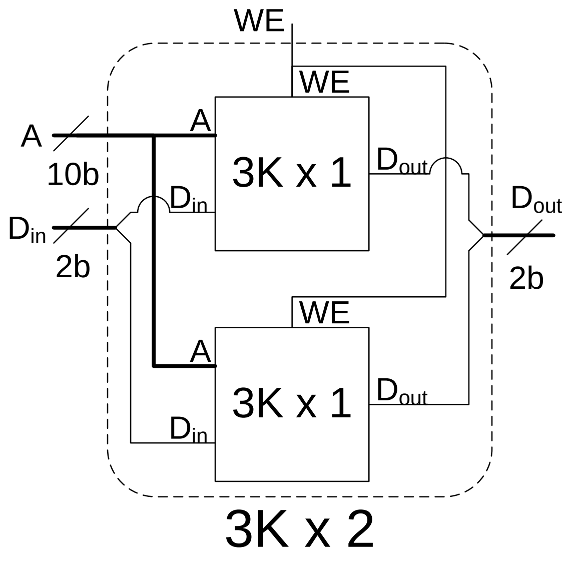
To make a 3K x 1 memory using 1K blocks, we must do a bit more work. In odred to increase the depth of memory, we must introduce a decoder an a multiplexer. Due to the fact that any given address corresponds to a location in one, not all, of the smaller memories, we must decode the write enable signal based on additional address bits. Twelve address bits are needed in total to index the 3K x 1 memory. The top 2 bits of address decode the write enable, and select the output. An optimal mux used the decode signals to drive a tristate output from each memory block.

The problem statement guarantees that unimplemented addresses never occur. We can optimize the decoding logic by treating the occurance of A[11:10] = 2'b11 as a don't care. We can also share the decoding logic used in the decoder and multiplexer for optimality.

A common error was a discontinuity in the address space caused by incorrect address decoding. In particular, the range of **(12'h800 : 12'hBFF)** in **(20'h00000 : 20'hFFFFF)** was often not mapped to physical memory. This discontinuity corresponds to the top address bits (2'b10) While it is true that a 3K x 2 memory does not occupy all of the 12b address space, it is essential that the memory is contiguous.

Due to the relative complexity of this problem, the grader may have taken notes on your exam. Just because there's red writing doesn't mean you've done something wrong!

**Figure 8** Problem 5: Composing a 3K x 2 Memory from 3K x 1 Blcoks.



A solution could earn up to a maximum of 12 points.

**+2** Low address bits **(A[9:0])** are wired to each memory block.

**+2** **Din** is wired to each memory block.

**+2** A total of 6 (Six) 1K x 1 memories used to implement the 3K x 2 memory.

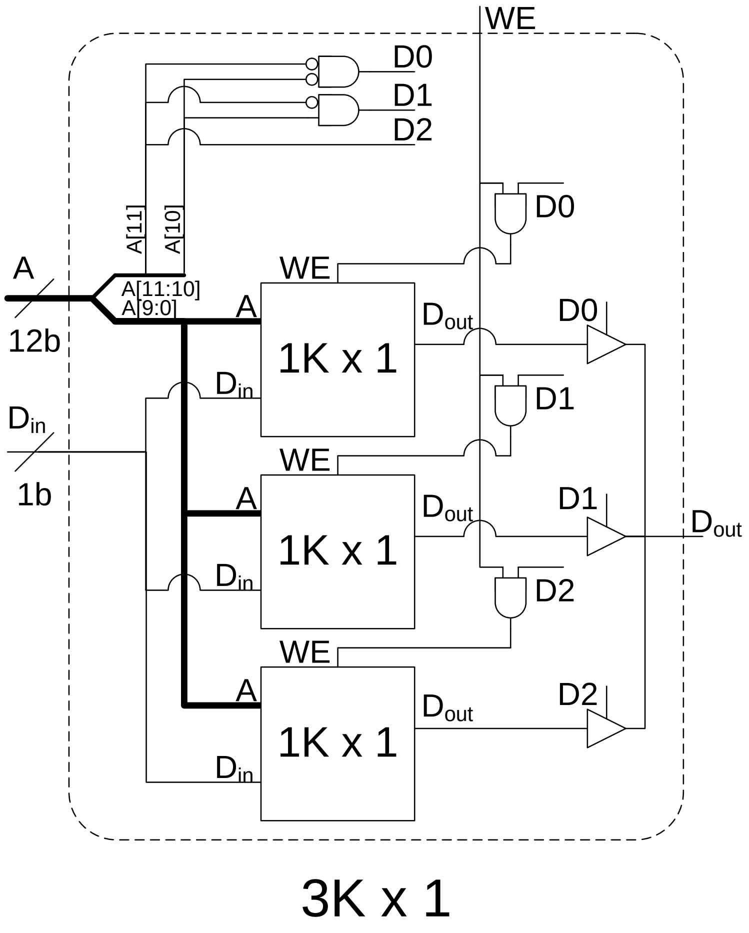
**+2** **Dout** of each block is muxed correctly based on A[11:10].

**+1** The multiplexer is correctly implemented with tri-state buffers for optimality. The decoding logic is shared with the decoder.

**+1** Two additional address bits are used (Address is 12 bits wide).

**+2** Decoder correctly decodes the write enable signal. Solutions with a discontinuous address space lost points.

**Figure 9** Problem 5: Implementation of a 3K x 1 Memory.



**-1** Significant lack of labeling of port names, bus bit widths, etc.

**-1** Significant lack of clarity in wiring (unclear intersections of wires, etc). Solutions that posed a significant challenge in the grading process lost both points.

## 6 Verilog Circuit Implementation (6+8+6=20)

This problem was worth 20 points.

### 6.1 Part A (6)

See Program 1.

### 6.2 Part B (8)

1 point for each output, and 1 point for each state's transitions
See Figure 10.

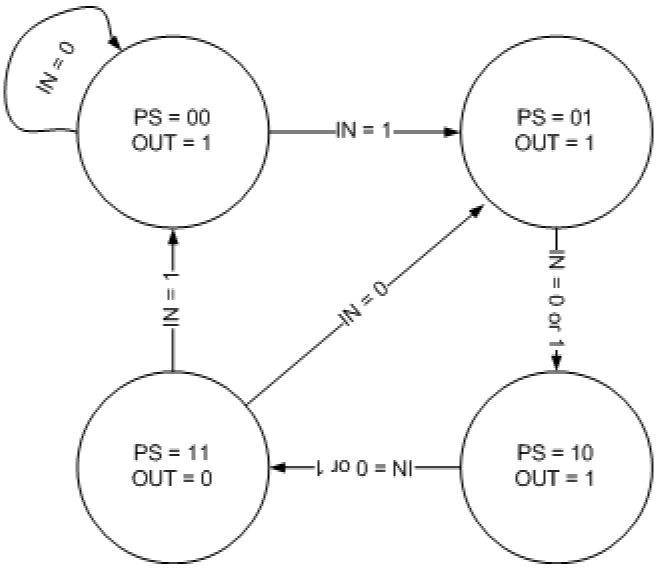
### 6.3 Part C (6)

See Program 2.

**Program 1** Problem 6a

```
module foo(
        input wire Clock,
        input wire IN,
        input wire Reset,

        output wire OUT
        );

        wire[1:0]       NS;
        reg[1:0]        PS;

        always@(posedge Clock ) begin
                if (Reset) PS <= 2'b00;
                else PS <= NS;
        end

        assign NS[0] = (~PS[0] & IN) | (~IN & PS[1]);
        assign NS[1] = PS[1] ^ PS[0];
        assign OUT = ~PS[1] | ~PS[0];

        /*
        2 points - declaration of wires and regs.
        2 points - always block.
        2 points - assign statements.
        */

endmodule
```

**Figure 10** State transition diagram for Probelm 6b



**Program 2** Problem 6c

```
module foo(
        input wire Clock,
        input wire IN,
        input wire Reset,

        output wire OUT
        );

        localparam      STATE_0 = 2'b00,
                        STATE_1 = 2'b01,
                        STATE_2 = 2'b10,
                        STATE_3 = 2'b11;

        reg[1:0]        NextState;
        reg[1:0]        CurrentState;

        always@(posedge Clock ) begin
                if (Reset) CurrentState <= 2'b0;
                else CurrentState <= NextState;
        end

        always@( * ) begin
                NextState = CurrentState;
                case (CurrentState)
                        STATE_0: begin
                                if(IN)
                                        NextState = STATE_1;
                                else
                                        NextState = STATE_0;
                        end

                        STATE_1: begin
                                NextState = STATE_2;
                        end

                        STATE_2: begin
                                NextState = STATE_3;
                        end

                        STATE_3: begin
                                if(IN)
                                        NextState = STATE_0;
                                else
                                        NextState = STATE_1;
                        end
        end

        assign OUT = ~(CurrentState = STATE_3);

        /*
        2 points - wire/reg declaration and OUT assignment
        2 points - always@(posedge Clock) block
        2 points - always@(*) block
        */

endmodule
```

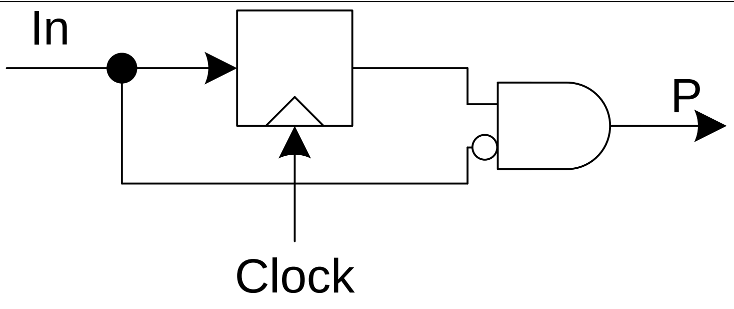
## 7 Clocked Circuits $(6 + 6 = 12)$

This problem was worth 12 points, divided equally between the falling edge detector and the pulse widener. For both parts of this problem, points were deducted for each error. Implementations which had enough errors to yield negative points were given 0 (instead of a negative score).

### 7.1 Falling Edge Detector (6)

A correct falling edge detector is shown in Figure 11. Points were assigned as follows:

**Figure 11** Problem 7.a: Falling edge detector.



**-1** Detects falling edges correctly but is slightly suboptimal in that it uses more gates than necessary. Redundant logic (such as an `and` gate that has one of its inputs tied to `1'b1`) is an example of this.

**-2** Detects positive edge instead of negative edge. This amounts to an inverter on the wrong input of the `and` gate.

**-2** Output changes synchronously.

**-2** Too many/few flip-flop delay elements. This deduction usually implied that your circuit's output changed synchronously (as you added an *extra* flip-flop at the output of the circuit). In this case, you lost 2 points only (instead of 4).

**-3** Overly complicated implementation that works correctly. For example, 3 flip-flops and enough logic gates to do floating point calculations is too complicated!

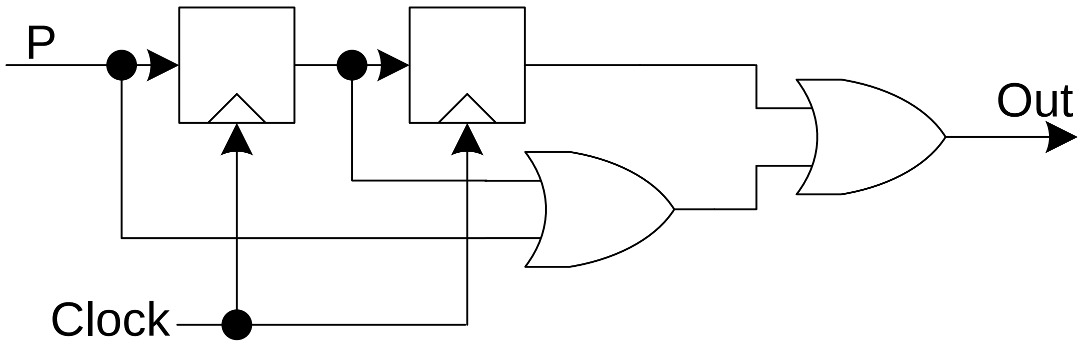
**-4** Doesn't detect positive or negative edges but has an idea.

**-6** A circuit that goes against design principles taught in class. An example of such a circuit is one where the `clock` signal is sent to the `D` input of a flip-flop, the `D` input is sent to the `clock` input, etc. **Note:** Designs that fell into this category typically suffered in the above categories as well. As a result, because no one got less than 0 points, you may have fallen into this category and lost the points in other categories.

### 7.2 Pulse Widener (6)

A correct pulse widener is shown in Figure 12. Points were deducted in the same fashion as enumerated in Section 7.1. There were several counter/FSMs implementations that counted up until 3 cycles had passed before returning the output to 0. In general, think simple gates and registers! It is amazing what you can do in the realm of signal conditioning with just several gates/flip-flops. If you limit yourself to only gates and registers, you will most likely design more optimally. As soon as you start building FSMs, you start thinking in software (serial) terms, which has the tendancy to lead to suboptimal performance and area consumption.

**Figure 12** Problem 7.b: Pulse widener.



## 8 Short Answers $(6+3+2+3+2+2+2+3+2=25)$

### 8.1 Quick Answers (6)

a. **False**. Increasing performance does **not always** increase cost. For example, you can raise the supply rails of your circuit to get it to run at a higher clock speed.

b. **False**. The keyword here is energy consumption **per operation**, which is just $\frac{1}{2}CV^2$ and is independent of the clock frequency.

c. The answer we are looking for here is that the general programmable interconnect is somewhat slow. If we used it to route our clock we will run into **clock skew** problems.

d. The easiest way to do this problem is to just draw out the two input waveforms and the output waveform, then see how many times the output transitions over some given period. If you assumed that the two input signals were synchronized to each other, you will get 2 transitions per 10 ns (period of the 100 MHz signal). If you did not make that assumption, then you will get 4 transitions per 10 ns. We accepted both answers. You can then plug these values into our equation:

$$P = \frac{1}{2}CV^2(2)\frac{1}{10ns} = CV^2(100 \cdot 10^6)$$

OR

$$P = \frac{1}{2}CV^2(4)\frac{1}{10ns} = 2CV^2(100 \cdot 10^6)$$

Points were assigned as follows:

**+1 part a** For answering False.

**+1 part b** For answering False.

**+1 part c** For talking about clock skew.

**+3 part d** 1 point was awarded for writing down the $\frac{1}{2}CV^2\alpha f$ equation down in part d. All 3 points were given only if you arrived at one of the two final answers.

### 8.2 CMOS Logic (3)

To do this, we can look just at our pull-down network and write an AND function whenever we see two transistors in series and OR whatever is in parallel. From the circuit, we can see that inputs $A$ and $C$ are connected to the gates of two NMOS transistors in series with each other, meaning we have a $AC$ term in our final boolean expression. Similarly, we see inputs $B$ and $D$ going to two NMOS in series with each other, meaning we have $BD$ term in our final boolean expression. We then note that the part of the circuit responsible for $AC$ is in parallel with the circuit responsible for $BD$ and we arrive at a circuit implementing $AC + BD$. Since we just looked at the pull down network, we must invert everything to arrive at our final answer: $\overline{AC + BD}$.

We also accepted other equivalents of this such as $\overline{AB} + \overline{AD} + \overline{CB} + \overline{CD}$.

Points were assigned as follows:

**+1** For getting started in the right direction, but was not able to get much further

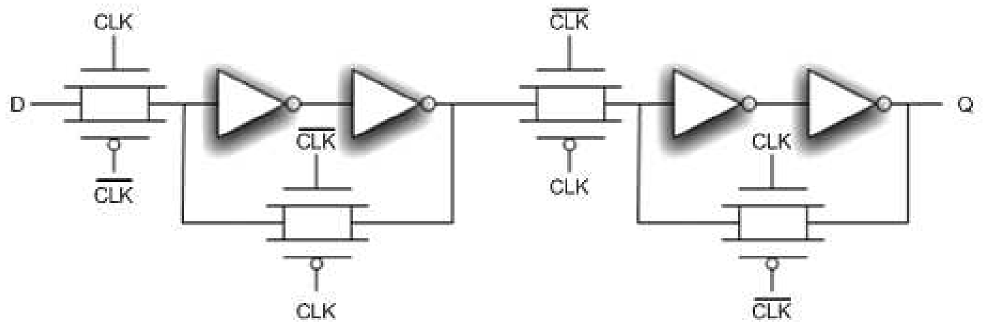
**+2** For showing a truth table but unable to come up with a correct boolean expression

**+3** For arriving at the correct boolean expression.

## 8.3 Negative Edge-Triggered Flip-Flop (2)

We have seen this structure many times in class; it is just a positive level-sensitive latch followed by a negative level-sensitive latch. See Figure 13.

**Figure 13** Problem 8f: Negative edge-triggered flip-flop



**+2** For labeling this correctly

## 8.4 Mystery Flip-Flop (3)

When X = 1, the output of the flip-flop is immediately forced to 1, regardless of what the Clock is at that instant. Thus, the signal X is used as an **asynchronous preset**.

**+1** For choosing either the asynchronous clear or synchronous set

**+3** For choosing asynchronous preset

## 8.5 Multiplexer mapping using LUTs (2)

The most optimal mux structure using a single 6-LUT is the 4:1 mux (4 data inputs, 2 selector inputs, 1 output). The most optimal mux structure using a single 4-LUT is the 2:1 mux (2 data inputs, 1 selector input, 1 output). Note that you cannot build a 3:1 mux using a 4-LUT, since it has 3 data inputs and 2 selector inputs.

**+1** For answering 4:1 mux for 6-LUTs

**+1** For answering 2:1 mux for 4-LUTs

## 8.6 Why Is Building Multiple Write Port RAM using SDP RAMs more expensive Than Building Multiple Read Port RAM? (2)

In addition to duplicating SDP RAMs (which is required to add more read ports as well), adding more write ports requires additional logic for the implementation of a residency table, which keeps track of where you wrote things is also needed.

**+1** For only mentioning something relatively vague such as "more logic" or "add a decoder"

**+2** For mentioning that the additional cost is in additional logic needed to implement the residency table

## 8.7 Architectural Features of FPGAs Responsible for Greater Power Consumption than Custom ICs (2)

Several possible answers include:

1. Long wires, presenting extra capacitances that increase the energy per transition
2. In most designs, large portions of the FPGA are unused, but are still powered-on and leaking current
3. Lots of hardware used for programmable logic, LUTs needs the extra hardware to implement any function, straight wires become switch matrices, etc.

**+1** For Making two very vague reasons

**+1** For Making only one valid reason

**+2** For Making two valid reasons

Many people put down ambiguous answers (i.e. "FPGAs have latches") without really mentioning why they would cause an FPGA to consume more power. These types of answers without any additional justification fell into the vague category and did not receive full credit.

## 8.8 FPGAs vs. ASICs (3)

i. FPGAs give you better time to market, since you can program it instantly whereas an ASIC chip takes months to come back from the fabrication plant.

ii. FPGAs give you less engineering costs, since it is easier to test and debug due to its reprogrammable nature.

iii. ASICs give you less per unit cost because it is custom-tailored for the specific purpose the chip is designed for. FPGAs are more general and programmable, and thus contain a lot of other logic that is unnecessary and cost more per unit. However, we also accepted the answer of "FPGAs for low volumes, ASICs for high volumes."

**+1** For each part

## 8.9 Write Your Name On Every Page (2)

Everyone got these two points, even those who missed a single page or two (I was lenient).

| Rev. | Name | Date | Description |
|---|---|---|---|
| A | John Wawrzynek<br>Chris Fletcher<br>Ilia Lebedev<br>Chen Sun | 4/10/2009 | Initial release. |