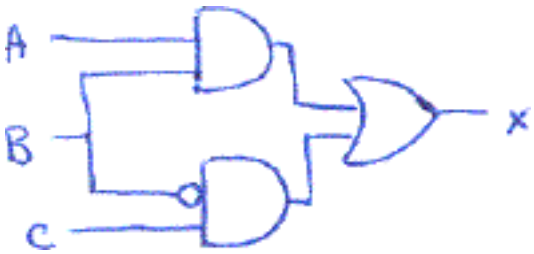
COS 126 Architecture Review Answers

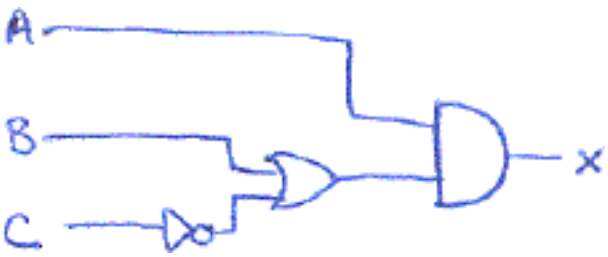
I. Combinational Logic

1.

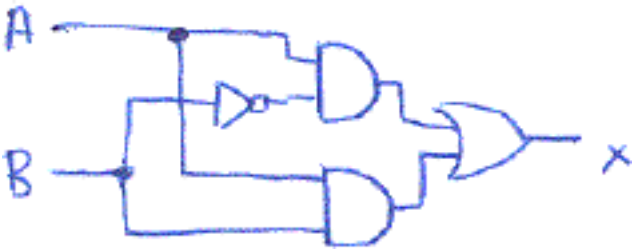
(a) x = AB + B'C



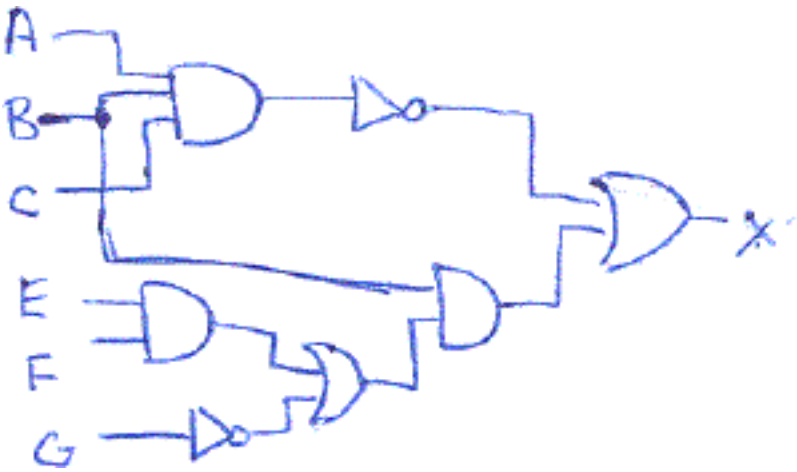
(b) x = A(B + C')



(c) x = A B' + AB



(d) x = (ABC)' + B(EF + G')



2. Assume that X consists of 3 bits, x2x1x0. Write three logic functions that are true if and only if

(a) X contains only one 1

This function has the following truth-table:

x2	x1	x0	f(x)
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

The corresponding sum-of-products boolean function is:

$f(x2, x1, x0) = x'_2 x'_1 x_0 + x'_2 x_1 x'_0 + x_2 x'_1 x'_0$

(b) X when interpreted as an unsigned binary number is less than 3

This function has the following truth-table:

x2	x1	x0	f(x)
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

The corresponding sum-of-products boolean function is:

$f(x2, x1, x0) = x'_2 x'_1 x'_0 + x'_2 x'_1 x_0 + x'_2 x_1 x'_0$

(c) X when interpreted as a signed (two's complement) number is less than -1

This function has the following truth-table:

x2	x1	x0	f(x)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

The corresponding sum-of-products boolean function is:

$f(x2, x1, x0) = x_2 x'_1 x'_0 + x_2 x'_1 x_0 + x_2 x_1 x'_0$

3. Assume that X consists of 2 bits, x1x0, and Y consists of 2 bits, y1y0. Write logic functions that are true if and only if

(a) X < Y, where X and Y are thought of as unsigned binary numbers
This function has the following truth-table:

x1	x0	y1	y0	f(x)
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

The corresponding sum-of-products boolean function is:

$$f(x2, x1, x0) = x'_1\, x'_0\, y'_1\, y_0 + x'_1\, x'_0\, y_1\, y'_0 + x'_1\, x'_0\, y_1\, y_0 + x'_1\, x_0\, y_1\, y'_0 + x'_1\, x_0\, y_1\, y_0 + x_1\, x'_0\, y_1\, y_0$$

(b) X < Y, where X and Y are thought of as signed (two's complement) numbers
This function has the following truth-table:

x1	x0	y1	y0	f(x)
0	0	0	0	0
0	0	0	1	1
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	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

The corresponding sum-of-products boolean function is:

$$f(x2, x1, x0) = x'_1\, x'_0\, y'_1\, y_0 + x_1\, x'_0\, y'_1\, y'_0 + x_1\, x'_0\, y'_1\, y_0 + x_1\, x'_0\, y_1\, y_0 + x_1\, x_0\, y'_1\, y'_0 + x_1\, x_0\, y'_1\, y_0$$

(c) X = Y

This function has the following truth-table:

x1	x0	y1	y0	f(x)
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

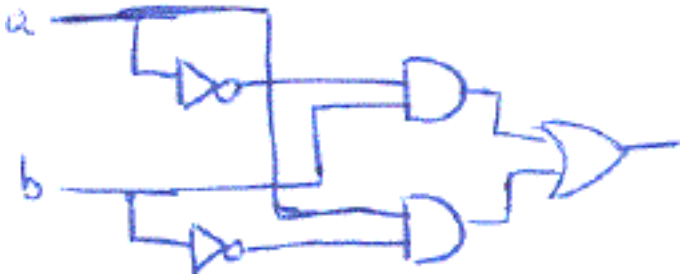
The corresponding sum-of-products boolean function is:

$f(x2, x1, x0) = x'_1\, x'_0\, y'_1\, y'_0 + x'_1\, x_0\, y'_1\, y_0 + x_1\, x'_0\, y_1\, y'_0 + x_1\, x_0\, y_1\, y_0$

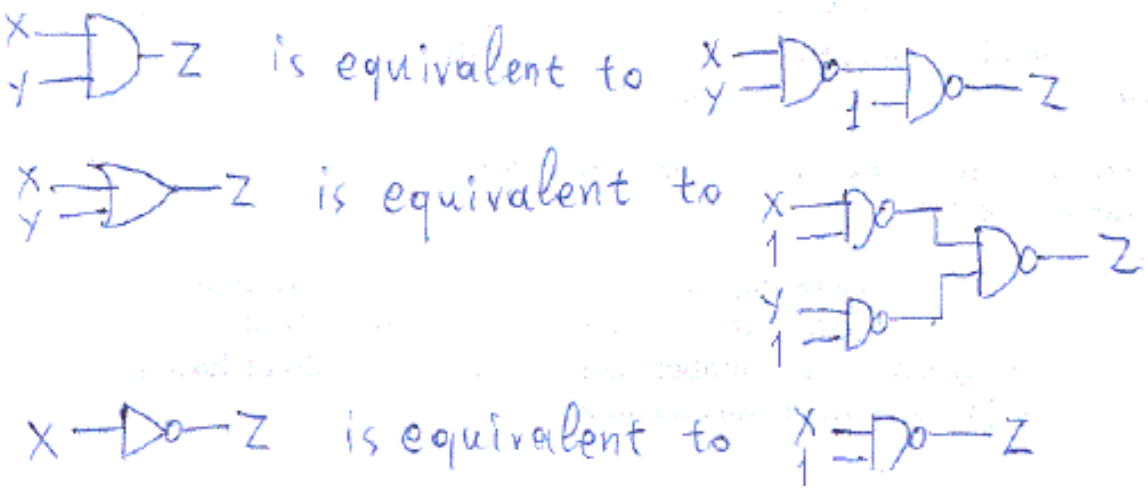
4.

The XOR function has the following truth-table:

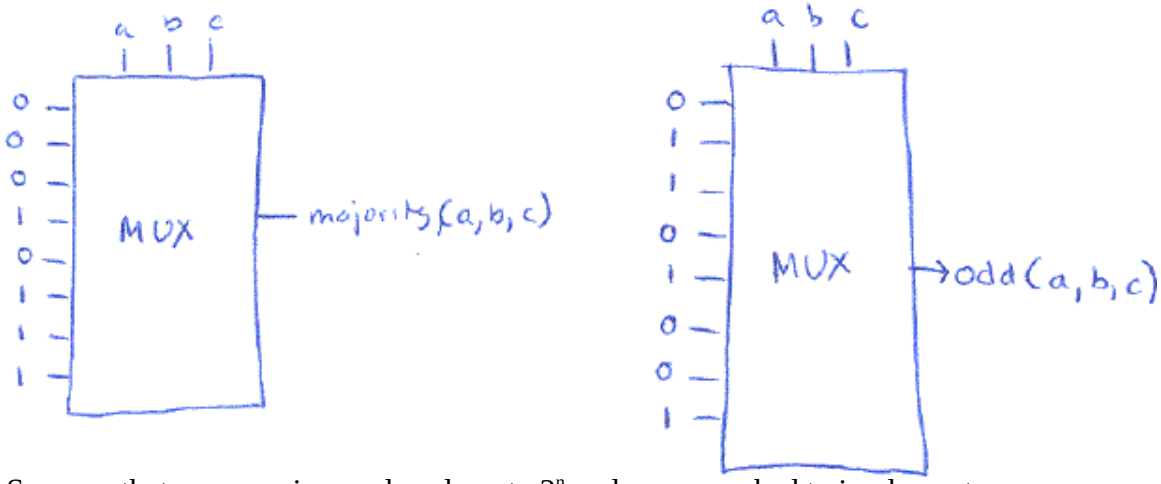
a	b	XOR
0	0	0
0	1	1
1	0	1
1	1	0



5. Show that the NAND gate is universal by constructing AND, OR, and NOT functions using a two-input NAND gate.



6. Suppose you have a multiplexer with 8 inputs (and 3 address lines). If you want to use the multiplexer to compute the majority function, treating the 3 address lines as the inputs to the majority function, what values should be assigned to the input lines of the multiplexer? How could you similarly adapt the multiplexer to compute the odd parity function?



7. Suppose that you are given a decoder n to 2^n and you are asked to implement one or more boolean functions with n inputs. What other component(s) do you need?

> The decoder implements all possible products between its *n* input variables. Since we can implement any boolean function using sum-of-products form, the only other component we need are OR gates. By passing the appropriate decoder outputs through an OR gate we can implement any boolean function we want. We use the decoder instead of a level of AND gates.

8. Implement a switching network that has two data inputs (A and B), two data outputs (C and D), and a control input (S). If S equals 1, the network is in pass-through mode, and C should equal A, and D should equal B. If S equals 0, the network is in crossing mode, and C should equal B, and D should equal A.

   This function has the following truth-table:

A	B	S	C	D
0	0	0	0	0
0	0	1	0	0
0	1	0	1	0
0	1	1	0	1
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	1	1

   C = A'BS' + AB'S + ABS' + ABS
   D = A'BS + AB'S' + ABS' + ABS

9. Here's how to approach this question, in four steps:

   1. Suppose we encode the functions as such:

      F0: f0f1 = 00
      F1: f0f1 = 01
      F2: f0f1 = 10
      F3: f0f1 = 11

      We can construct a truth table that has three inputs (X, f0, f1) and one output (F). From this truth table, using the sum-of-product method, we can construct the output function:

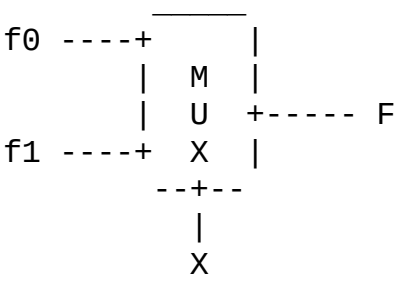
      F = f0' * f1 * X + f0 * f1' * X' + f0 * f1 * X' + f0 * f1 * X

   2. But there's a much easier way without going through the truth table. Let's think about the very meaning of f0 and f1. When we define a one-input function, we need to specify whether we want the output to be 1 when the input is X=0, and whether we want the output to be 1 when the input is X=1. In other words, we have two independent choices (controls) for each of the two possible values of X. The meaning of f0 and f1 is to encode these two independent choices for the two possible values of X. So the output is simply:

      F = f0 * X' + f1 * X

      (We can prove that this result is equivalent to the one above, but you don't have to worry about the proof.)

3. The problem is even simpler if we realize that the value of the input X is "choosing" which one of the two controls will have an effect on the final output. So to solve the problem, we can simply use a single 2-1 MUX:

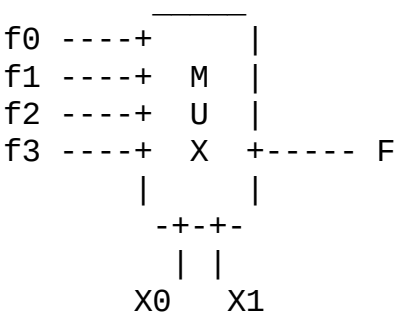


4. Think about how to extend the solution to this problem when we have two input variables X0 and X1 (page A3-2 of course packet). The total number of functions of two inputs is 2^2^2=16 so we need four bits to encode all those possible functions: f0 f1 f2 f3

Here are the solutions:

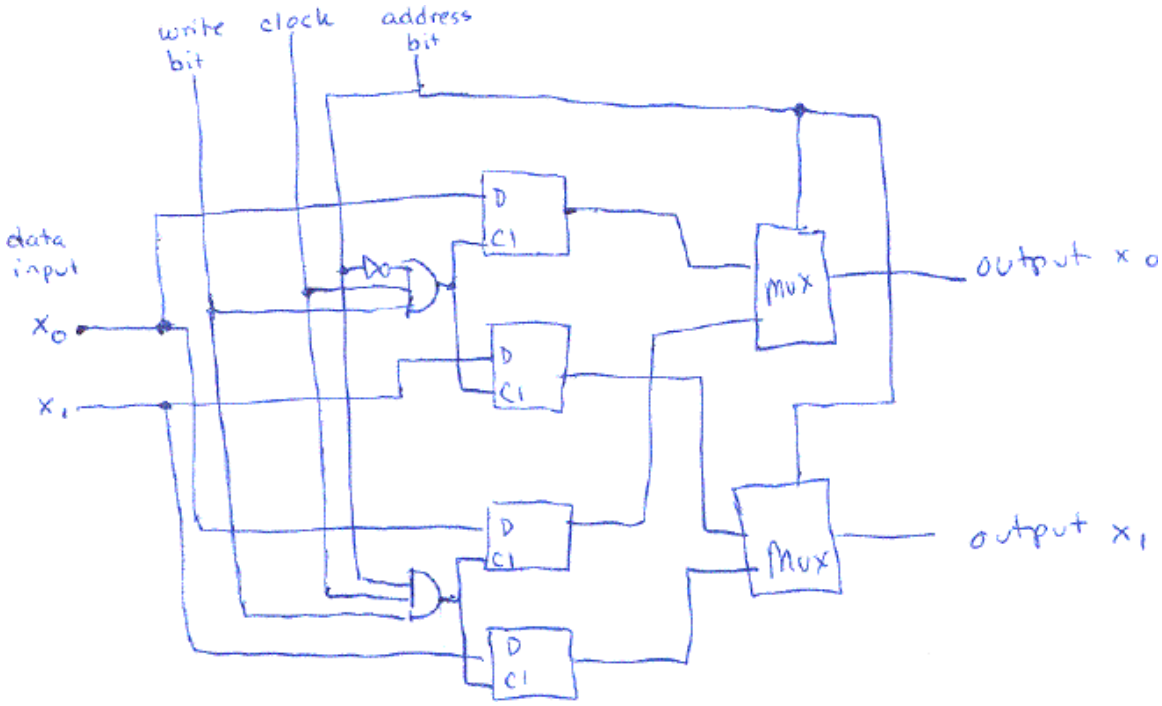
$$F = f0 * X0' * X1' + f1 * X0' * X1 + f2 * X0 * X1' + f3 * X0 * X1$$

Or using a 4-1 MUX:



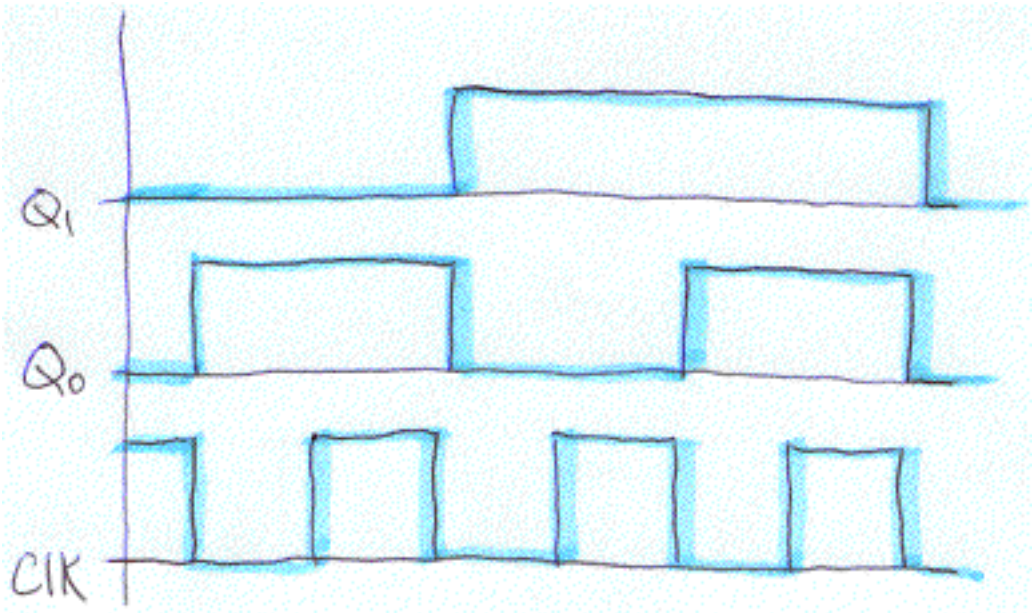
II. Sequential Circuits

10.



11.

Answer:



Comment: This is an alternative way of implementing a 2-bit counter.

12. a) From the truth table we can derive:

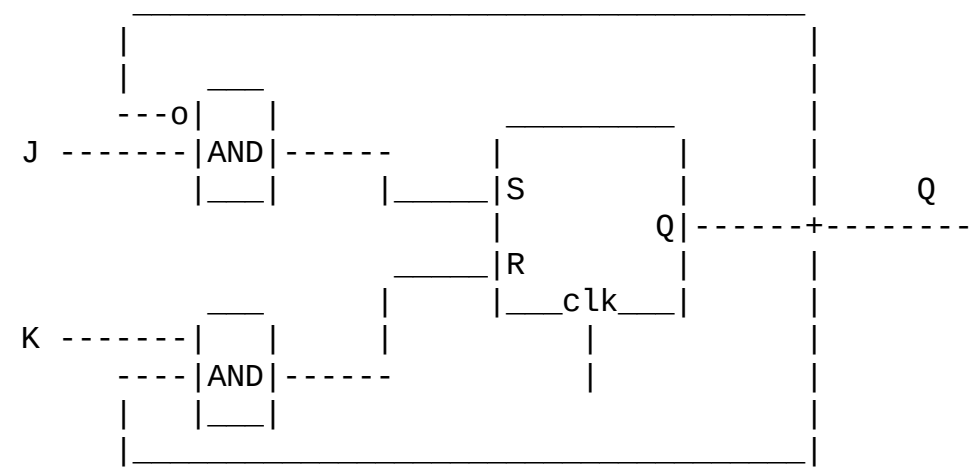
$Q^+ = J'K'Q + JK' + JKQ'$
which can be simplified (don't worry too much if you don't see how, but it can be done) to:

$Q^+ = K'Q + JK' + JQ'$

From here the implementation with logic gates is trivial, as long as we remember to AND the two inputs with the clock signal, as in the clocked SR flip-flop in the lecture notes. However, the circuit has a problem: during the time when the clock is 1 and J=K=1, the value of the output keeps changing from 0 to 1 and back at really high speed, and when the clock goes back to 0 the output can stop at any of the two values, while we wanted it to be exactly Q'. To ensure that the JK flip-flop operates correctly, we must carefully control the clock pulse width to ensure that the state of the flip-flop changes at most once per cycle.

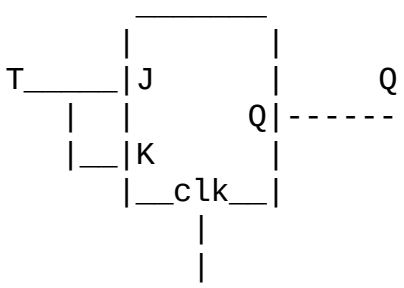
12. b)

We use an RS flip-flop and add some logic gates to make it behave like a JK. We also need to make sure S and R will never be set to 1 at the same time. A good way to do this is to keep S and R on zero all times except when it is necessary to set one of them to 1.
We only need S=1 when Q^+ goes from 0 to 1, and in the case of a JK that happens if and only if J=1 and Q=0. Therefore we hook up S to J*Q'. Similarly, Q^+ only goes from 1 to 0 when K=1 and Q=1, so we hook up R to K*Q. You can verify we'll never have S=R=1.
The circuit looks like:



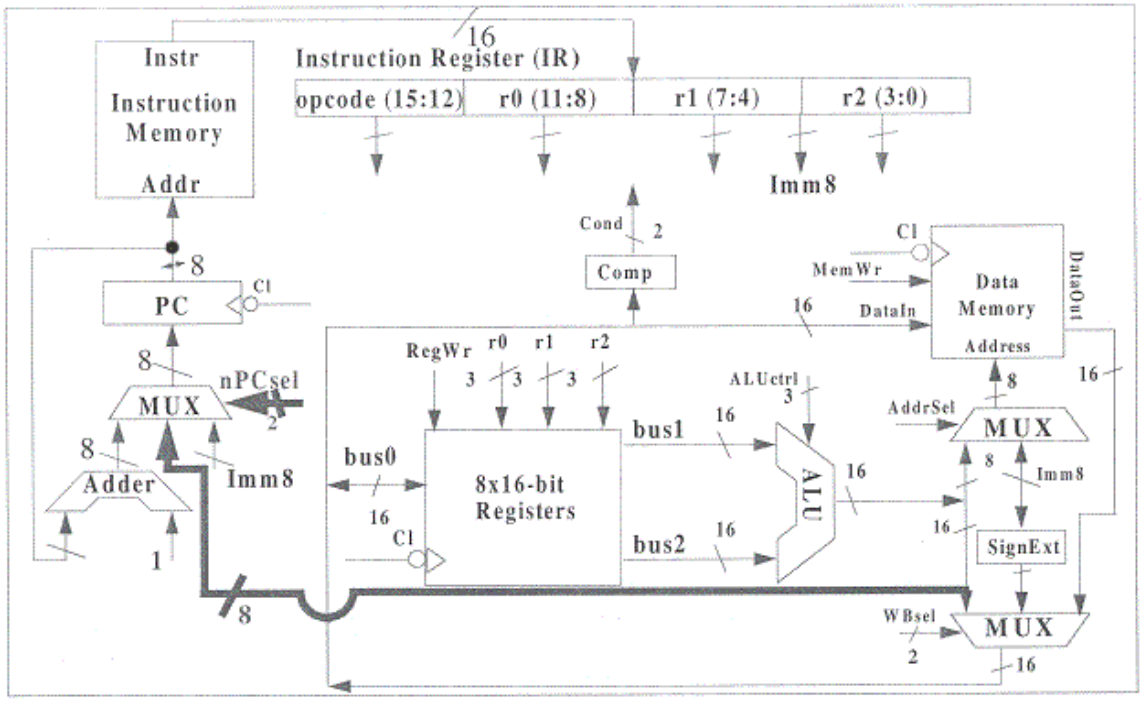
If the RS flip-flop we use is the standard one, we have the same problem as in part a), In addition to the fragile solution of controlling clock width, we can use a master-slave RS flip-flop to break the feedback loop and solve the problem. Now, when the clock is 1 only the master will be updated and will save the new input values, then when the clock goes down to 0 only the slave will be on and will update the output value Q.
A master-slave SR flip-flop is similar to the master-slave D in the lecture notes, except slave S is connected to master Q, and slave R to the inverse of master Q.

12. c) No need to write truth tables.
We know the JK toggles its output whenever J=K=1, and preserves it when J=K=0. So all we need to do is to connect the J and K inputs together:



III. TOY Architecture

13. Here's what the updated datapath should look like. The thick black line near the bottom is the difference between this version and the one in the original lecture slides.



14. What values should be assigned to each of the control lines in the Single Cycle TOY Datapath given in lecture when the instruction AF10 is executed?

    memWr: 1
    AddrSel: 0 (because we want to select the result of the ALU as output of the MUX)
    WBsel: doesn't matter because we're not writing to the register file
    RegWr: 0
    ALUCtr: depends on the encoding of the ALU functions. If ADD is, say, the second function implemented by the ALU, ALUCtr should get the value: 001
    nPCsel: 0 (because we want to set the value PC+1 as the next PC

15. Consider the TOY instruction set, suppose we decided to add the division operation, how many control lines (ALUctrl) do we need?

    b. three
    With one control, we can select between 2 functions (setting 1 to 0 & 1 to 1). With two controls, we are able to encode 4 different functions (00, 01, 10, 11). The TOY instruction set has 7 functions that need to be executed by the ALU: add, subtract, multiply, and, xor, shift right, and shift left. Adding division increases this number to 8. Since we need to specify one of 8 functions, we need 3 controls: (000, 001, 010, 011, 100, 101, 110, 111). Note that if we wanted to add another function to our ALU, we would need to add an additional control line.

16. Consider the TOY datapath. If we read and write the same register in a single instruction (such as R1=R1+R2) under the single cycle design, how is the TOY datapath able to avoid repeatedly fetching and incrementing the same register (R1) multiple times in a cycle?

    A write to the register file can only happen at the end of a clock cycle (triggered by the falling clock edge).

17. a) add:
    instr. fetch + reg. file read + add + reg. file write
    = 3+1+2+1 = 7 ns

17. b) multiply:
    instr. fetch + reg. file read + multiply + reg. file write
    = 3+1+4+1 = 9 ns

17. c) store:
    instr. fetch + reg. file read + add + memory store
    = 3+1+2+3 = 9 ns

17. d) load:
    instr. fetch + reg. file read + add + memory load + reg. file write
    = 3+1+2+3+1 = 10 ns

18. a) The maximum of instruction delay, load instruction, 10 ns.

18. b) The maximum of any single stage, multiply ALU op, 4 ns.

19. a) The maximum of instruction delay, still load, still 10 ns.
19. b) The maximum of any single stage, now it's memory, 3 ns.

20. a) The maximum of instruction delay, multiply instruction:
instr. fetch + reg. file read + multiply + reg. file write
= 3+1+8+1 = 13ns

20. b) The maximum of any single stage, multiply ALU op, 8ns.

· The goals of these last four exercises are to help you:

a) Visualize the different functional units an instruction needs to "touch" in the datapath in order to do what it's supposed to do;

b) Understand the concepts of single cycle design vs. multicycle design;

c) Get a taste of why the computer architect's job can be complex. Changes to the organization are seldom isolated. For example, changing the multiplication part of the ALU affects the cycle time which in turn affects performance of instructions that have nothing to do with multiply at all! As another example of complex interactions, improving the performance of one part can have zero effect on the final result because of other slow parts of the chip that's becoming the bottleneck (look at case 19 a).