Lab 4: Sequential Logic

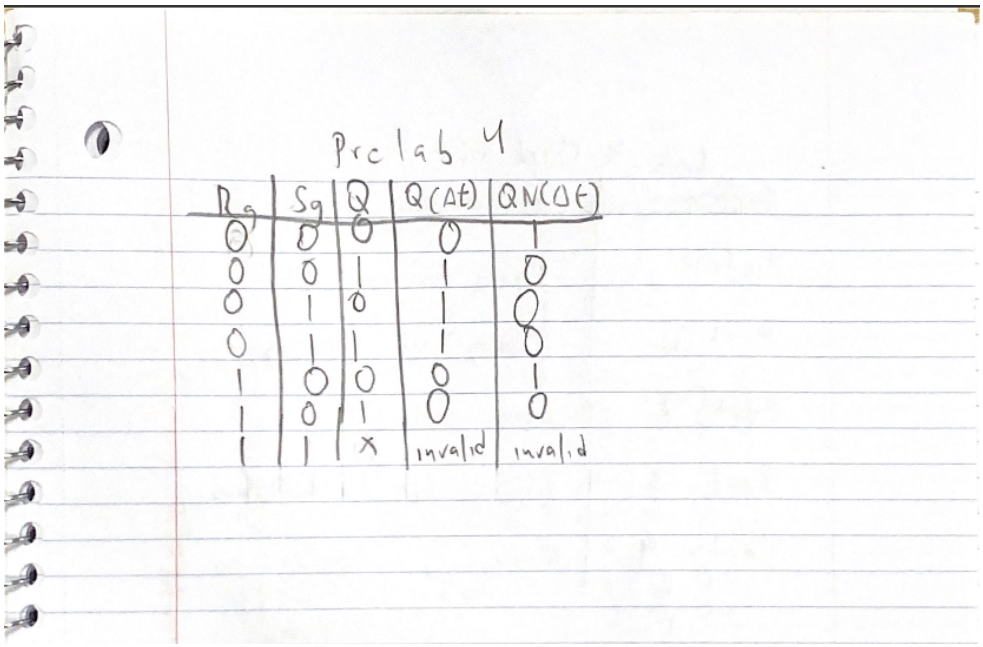
Eric Liu

Objectives

In this lab, our goal is to understand how latches, flip-flops, and registers work: the basic building blocks of sequential logic. We will achieve this by creating two variants of latches (one made with NOR gates and one made with NAND gates), a flip-flop, and finally a register that connects to a 7-segment display so we can “view its contents”.

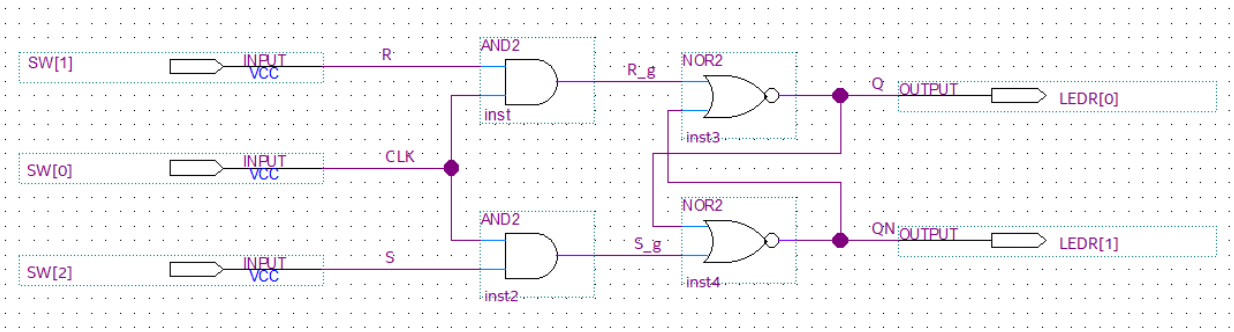
Design and Test Procedure

Prelab:



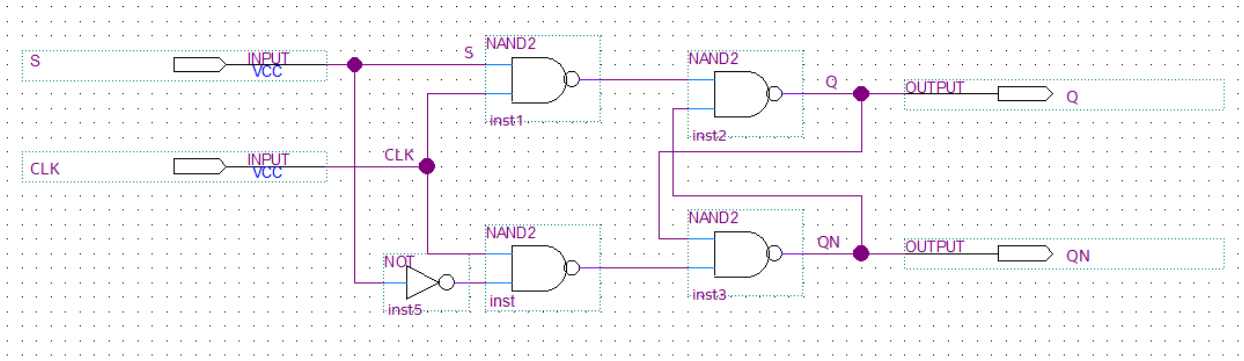
This simply describes the behavior of the RS latch we will be building in part 1. Note that R and S both being driven high should be invalid because R is reset and S is set, so you shouldn’t reset *and* set the memory at the same time.

Part 1:

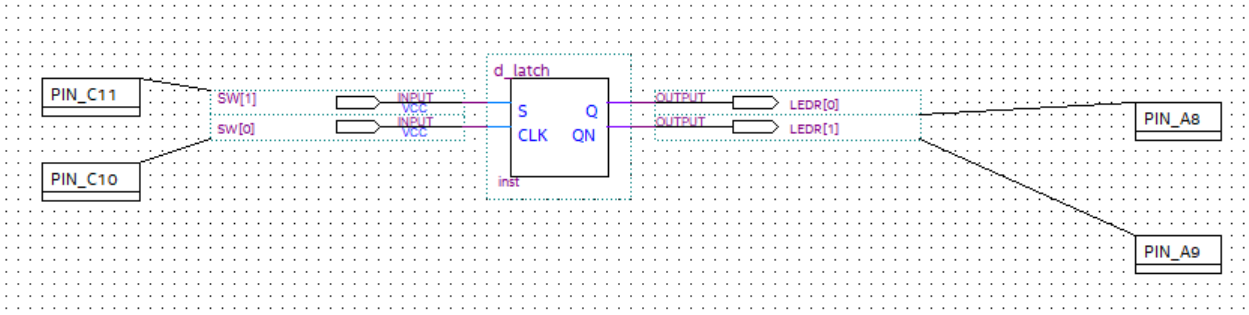


This is simply a replica of the schematic given in the lab manual. This is an RS latch: the R input is a “reset” input that sets Q to 0 after enough time delay, and the S input is a “set” input that sets Q to 1 after the same amount of time delay.

Part 2:

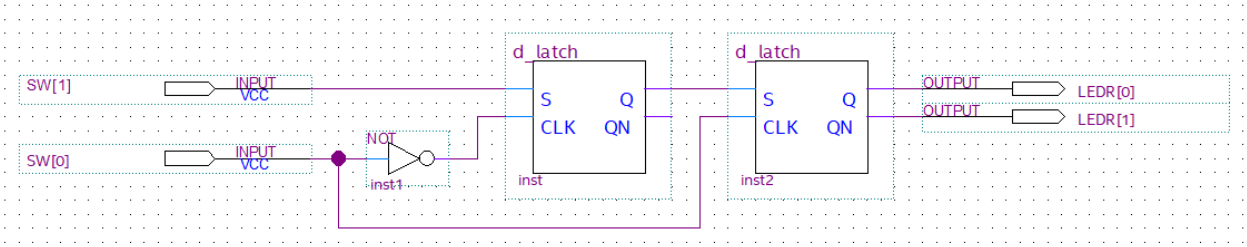


D latch component.



This is a D latch. It’s an improvement over the RS latch because it removes the possibility of invalid state: R and S both being driven high. Since there’s only a “set” input, there’s no possibility of reset and set being requested at the same time. Instead, S being driven low “resets” or “clears” the input, while S being driven high sets the input.

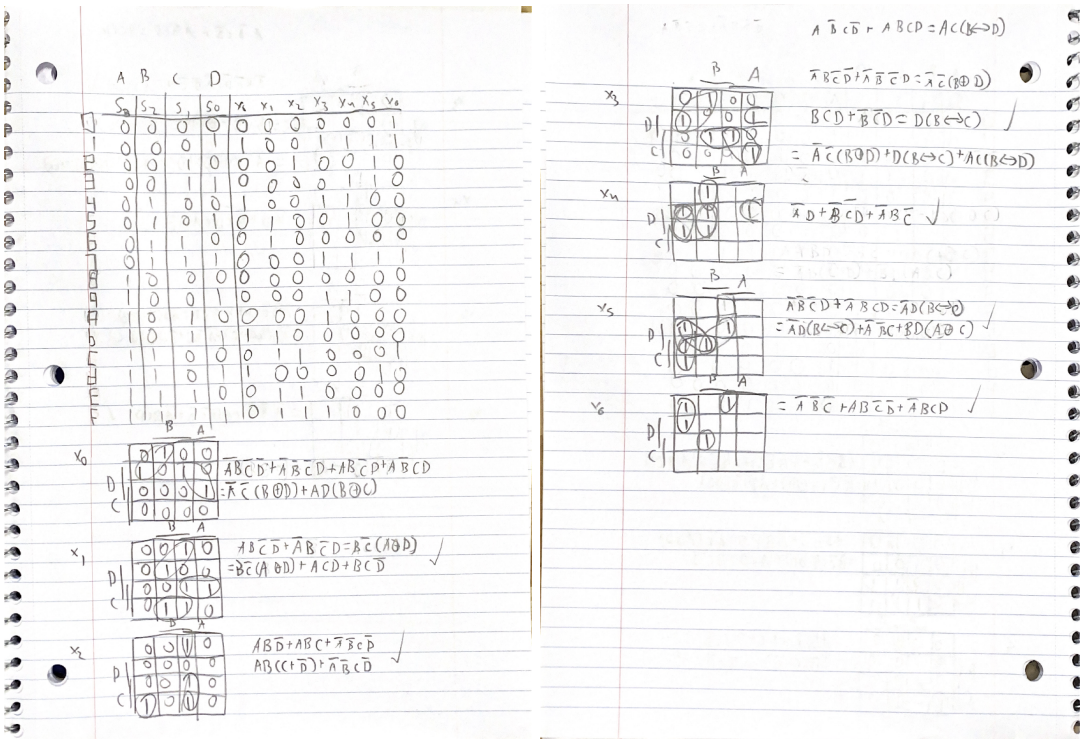
Part 3:



The `d_latch` component is the same as the one used in part 2.
This is a D flip-flop. Flip flops are different from latches because they require a specific change in clock signal (edge trigger) to set the input. Here, the memory can only be set when the CLK input flips from low to high: no other state can change the memory cell’s value.

Part 4:

K-maps, formulas, and truth tables:



Final simplified functions:

$$x_0 = \neg A \neg C(B \oplus D) + AD(B \oplus C)$$

$$x_1 = B \neg C(A \oplus D) + ACD + BC \neg D$$

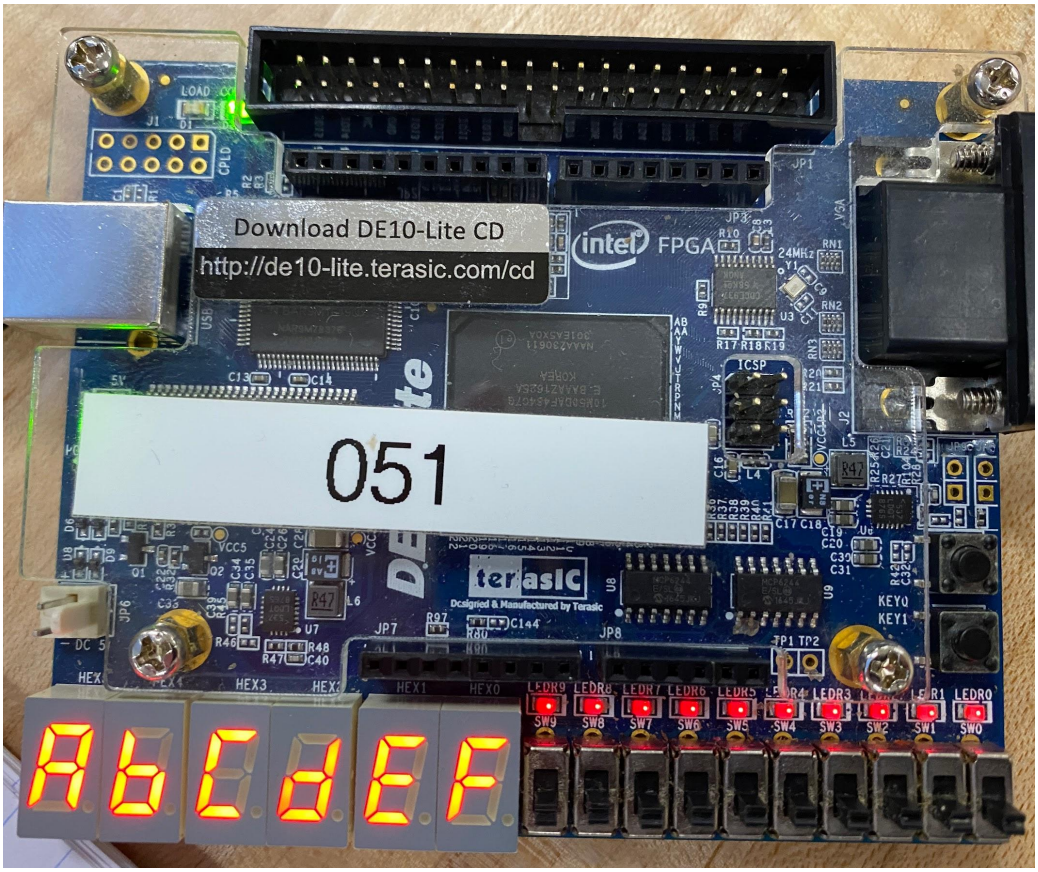
$$x_2 = AB(C + \neg D) + \neg A \neg BC \neg D$$

$$x_3 = \neg A \neg C(B \oplus D) + D(B \leftrightarrow C) + AC(B \leftrightarrow D)$$

$$x_4 = \neg AD + \neg B \neg CD + \neg AB \neg C$$

$$x_5 = \neg AD(B \leftrightarrow C) + \neg A \neg BC + BD(A \oplus C)$$

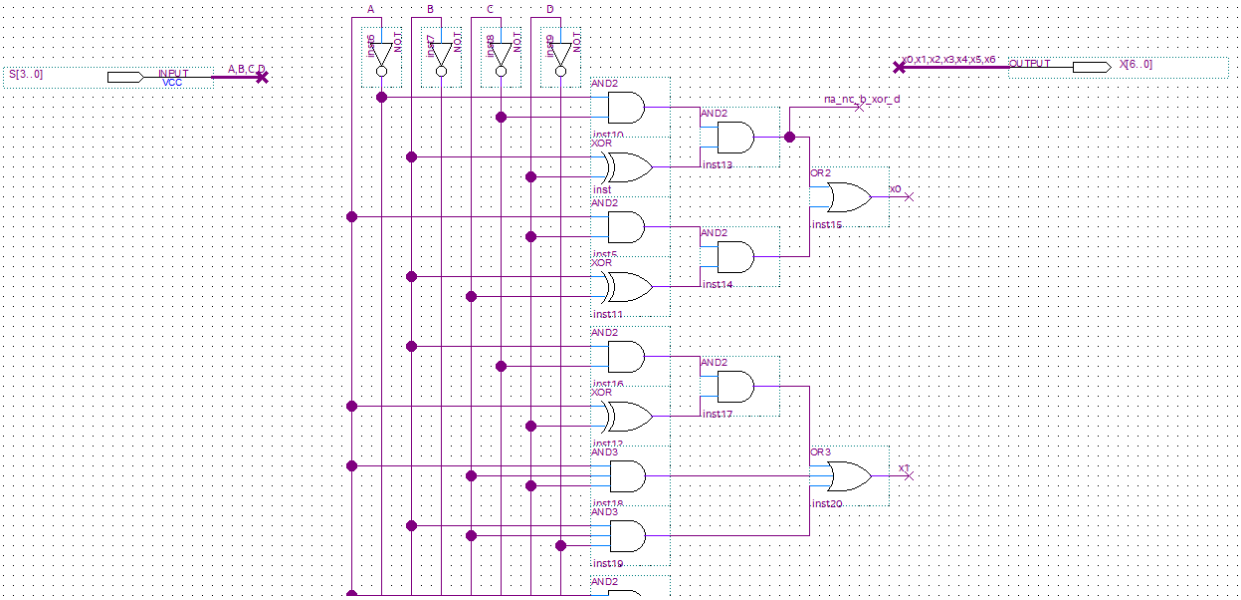
$$x_6 = \neg A \neg B \neg C + AB \neg C \neg D + \neg ABCD$$



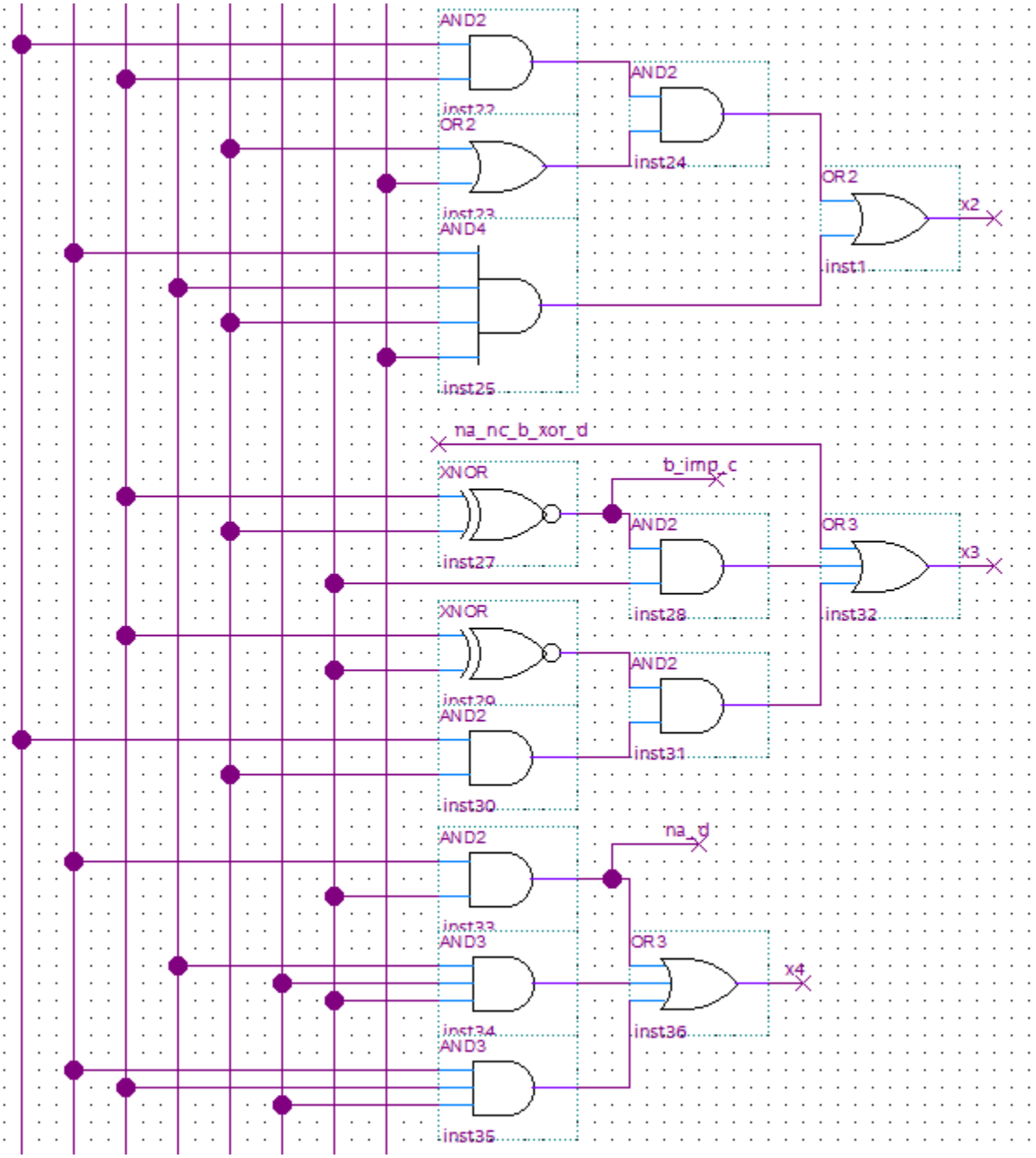
Circuit demoing A-F hex literals.

This is the design for the 7-segment display circuit that we will use to display 4-bit binary numbers in decimal (0-9, A-F).

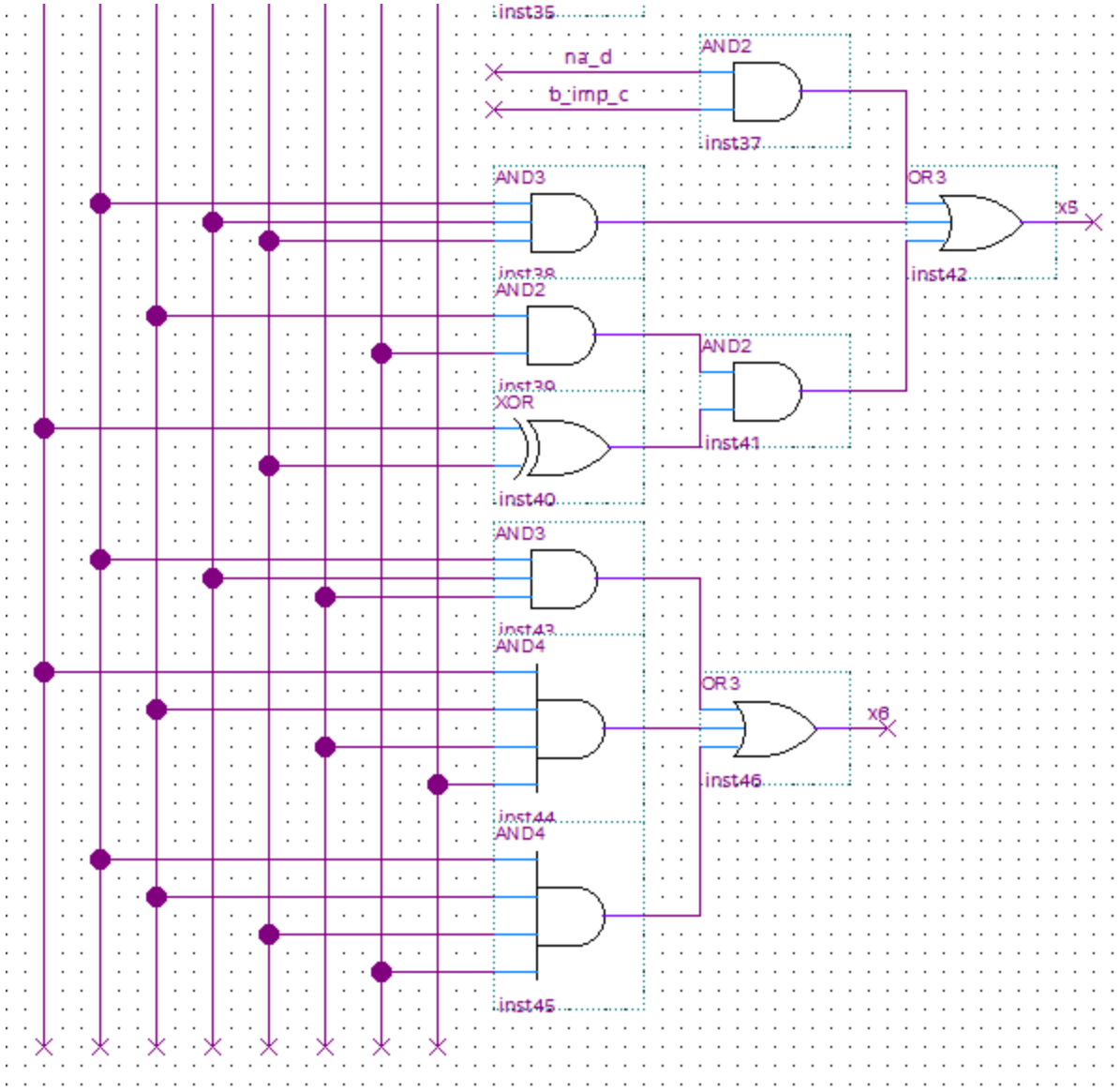
Below is the block diagram for the whole circuit component, split into parts because it's so long:



7-Segment display circuit, segments X0 and X1.



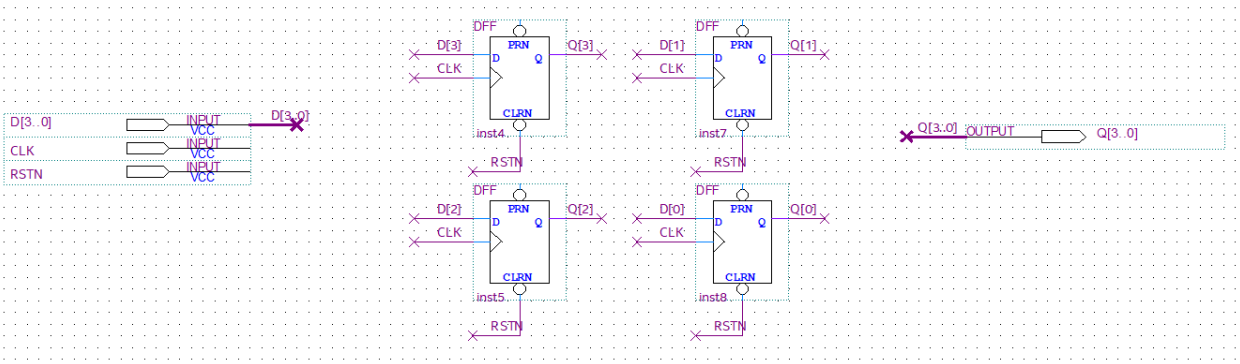
7-Segment display circuit, segments X2 to X4.



7-Segment display circuit, segments X5 and X6.

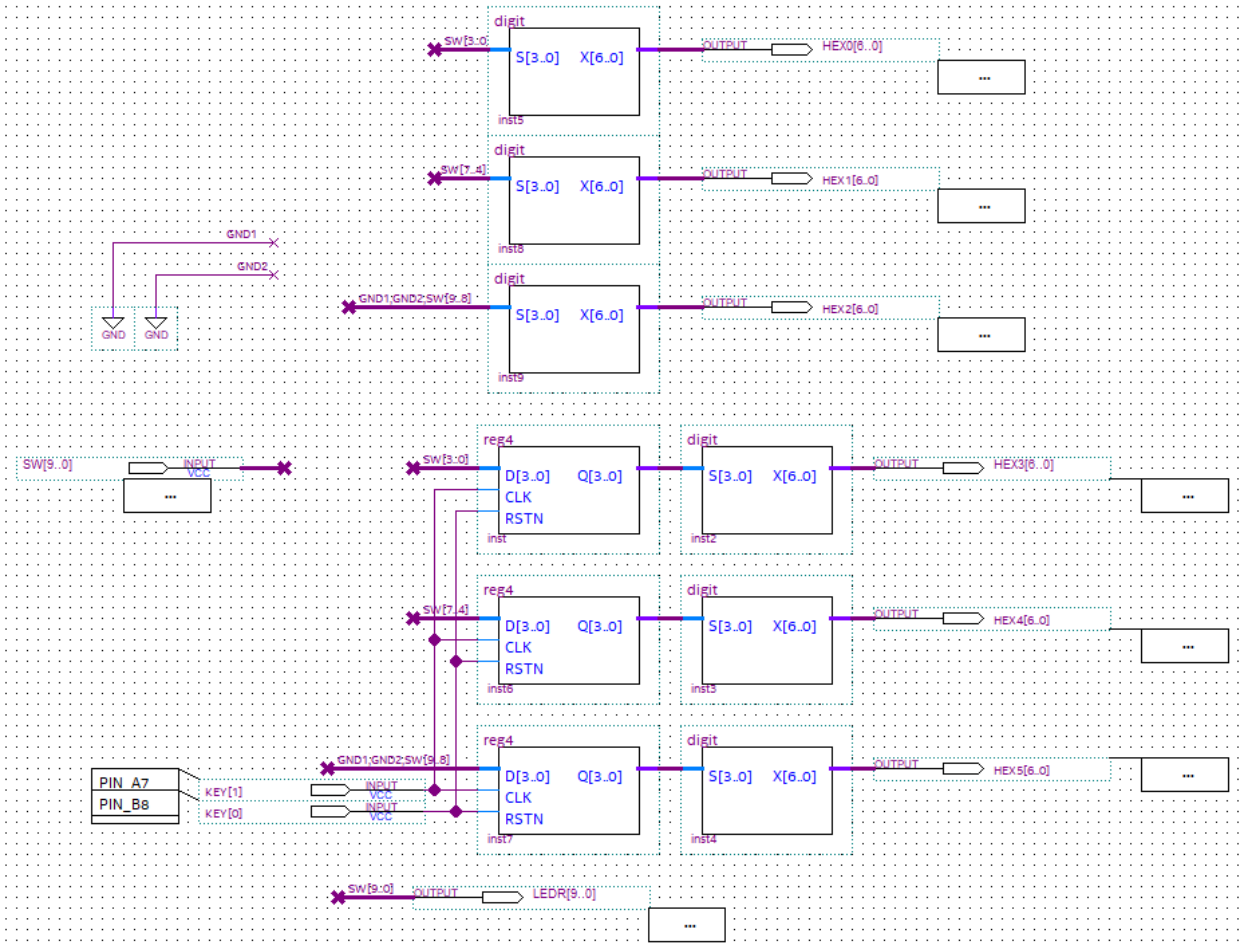
Not much to say about these schematics; they mirror the functions derived from the K-maps.

I reused some input combinations (like `na_d`, `b_imp_c`, and `na_nc_b_xor_d`) in order to reduce the likelihood of error and also reduce the number of gates I used to implement the circuit.



4-bit register component.

This is the actual register we are using to store values. It's simply 4 D flip-flops but wired to the same CLK and RST inputs.



Top-level circuit for part 4.

This is the final circuit. It contains 3 4 bit binary number inputs (technically 2 4 bit and 1 2 bit), along with 3 4-bit registers that can be set and reset accordingly. All 3 inputs and registers are also displayed on the 7-segment displays. The lower 3 simply reflect the input values, and the upper 3 show the current register contents. Registers can be

changed by using the KEY0 and KEY1 buttons. KEY1 triggers a positive edge of the clock signal and prompts the registers to store the values currently provided by the switches, and KEY0 resets all registers to 0.

This part was quite long. For this reason, I actually made a lot of separate small circuits to test each part individually on the FPGA before I combined them all to get the final result. I initially started with only 1 7-segment display connected to 4 switches so I could make sure that I implemented my converter correctly. I also tested the register logic separately so I could understand what each input and output did individually, so that I could more easily debug the circuit later on if the output wasn't what I expected. Finally, I combined everything together, which luckily worked almost flawlessly.

Results and Answers to Questions

Part 1:

1. Yes. The CLK input has to be high in order for the R and S inputs to take any effect because both inputs are ANDed with CLK, meaning that if CLK is low, neither will drive any wire high.
2. No, for the same reason as stated in question 1. The R and S inputs won't drive any wire high if CLK is low because the inputs are ANDed together with CLK.
3. When R and S are both high, Q = QN = 0. This is considered illegal because you shouldn't be driving both the reset and set inputs at the same time so it shouldn't come up as a real issue in properly designed circuits.
4. The output stays the same as what was last driven to it with CLK enabled. If R was last driven, LED 1 stays on (QN), and if S was last driven, LED 0 stays on (Q).
5. LED 0 turns on (Q is driven high and QN is driven low). I think the reason why this happens is a race condition: both Q and QN should be low from R and S being equal to 1, but then QN propagates to the gate that produces Q, which drives it high (0 NOR 0 is 1), which then feedbacks into QN (1 NOR 0 is 0), steadily propagating Q as 1 and QN as 0. There's not really a reason why QN propagates first; it's simply random or an artifact of Quartus's synthesizer. It could very well be that someone else with an identical circuit would observe LED 1 turning on (Q driven low and QN driven high), completely arbitrarily.

Part 2:

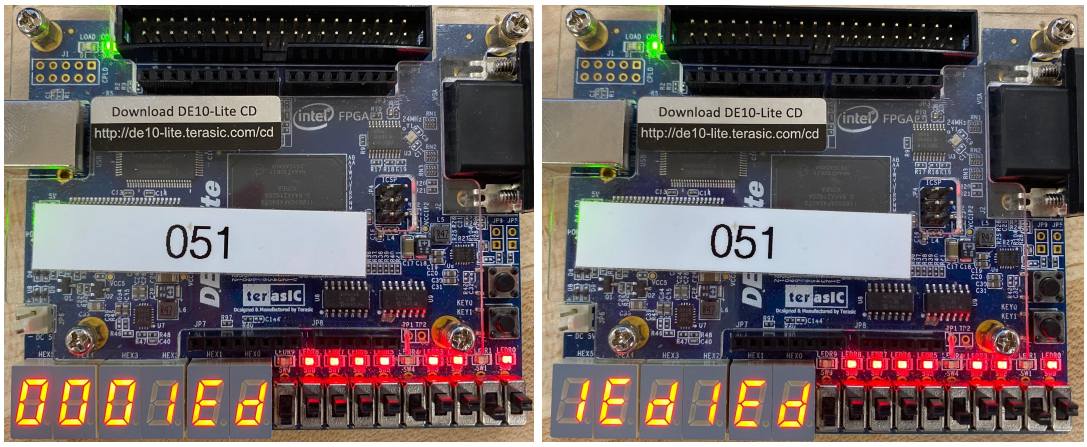
1. Yes. Here, the CLK input is NANDed with S and passed to additional NAND gates, so CLK has to be high in order for the wires to propagate any signal from S.

2. No, since CLK is NANDed with S and passed to additional NAND gates, so CLK being low means the wires don't propagate any net signal from S.
3. No, there is no case where Q = QN, meaning there is no invalid state in a D latch. This makes sense because the "reset" input is now derived from the set input, which makes it impossible for S and "R" to be the same value.

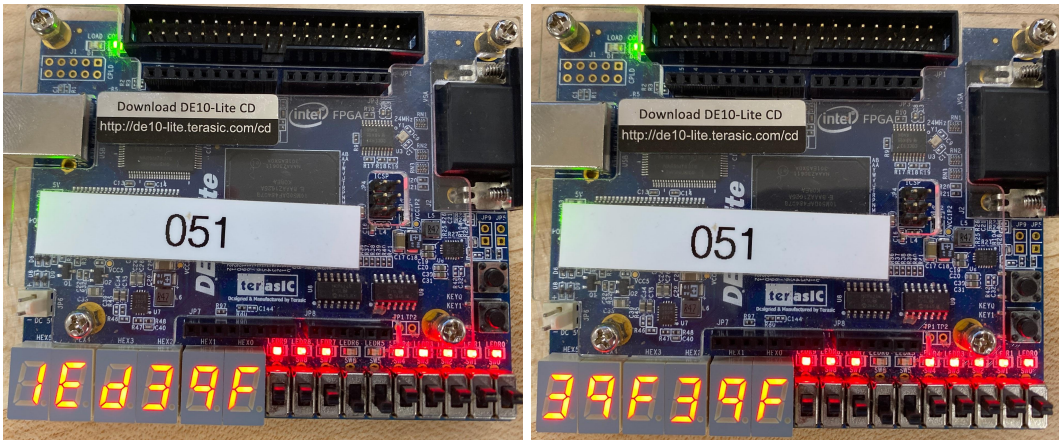
Part 3:

1. The output is only affected when the CLK *changes* from low to high. Keeping the CLK as high does not affect the output by itself, but if you change the D input and then switch the CLK on, the output LED (and Q) changes accordingly.
2. No, the D input doesn't affect the output when the CLK is low. Only when the CLK is *switched* from low to high does the output LED change state.
3. The output only changes when the CLK input switches from low to high. This is similar to the `always @(posedge clk)` construct that's very common in SystemVerilog. And of course, this is different from the behavior of a D latch because a D latch changes the memory cell whenever the clock is high, instead of when the clock jumps from low to high.

Part 4:



Resetting the register then setting the register to 0x1ED, before and after.



Changing the inputs without the register changing, then setting them again with KEY1.

Conclusions

In this lab, we learned about the basic elements that make up sequential logic: latches, flip-flops, and registers. We first looked at the RS latch, and then moved on to the D latch, which is better because it can't represent invalid state. Then, we looked at a flip-flop constructed from the D latches, to see how edge triggering can be beneficial, and then created registers from adding multiple flip-flops together. This was very educational because now we will be able to implement circuits that retain state, rather than only responding to the current input, which is a lot more powerful in terms of what can be created.

Only a couple things went wrong in this lab. When I was implementing the 7-segment display converter, I accidentally messed up one of the segments by using an AND gate rather than an OR gate, which led to incorrect output. I simply fixed it by figuring out which segment was incorrect and analyzed the gates to make sure they lined up with my boolean formula, and I caught the error pretty early.
I also had a bit of trouble understanding how the register's RSTN input worked. I originally inverted the RSTN to connect it to the D flip-flop provided by Altera, but then I wasn't able to set it on the final circuit. I realized that the N in both the RSTN and CLRN inputs meant that both inputs were inverted, and the circuit worked properly after I removed the inverter.

I think I could've improved upon my design by merging more gate sequences together. In my converter I had a lot of redundant sequences that could've been shared between outputs, but I didn't really want to bother trying to optimize this because I felt that it would be much easier to mess up. I still added a couple of shared wires for elements of functions that were very obviously the same, but for smaller elements (like A AND B), I simply redid them per segment.

Overall, this lab was very instructional in understanding the power and usage of sequential circuits and I think this content will be very useful in future labs where we'll construct more elaborate circuits