Lab 6: Dice game

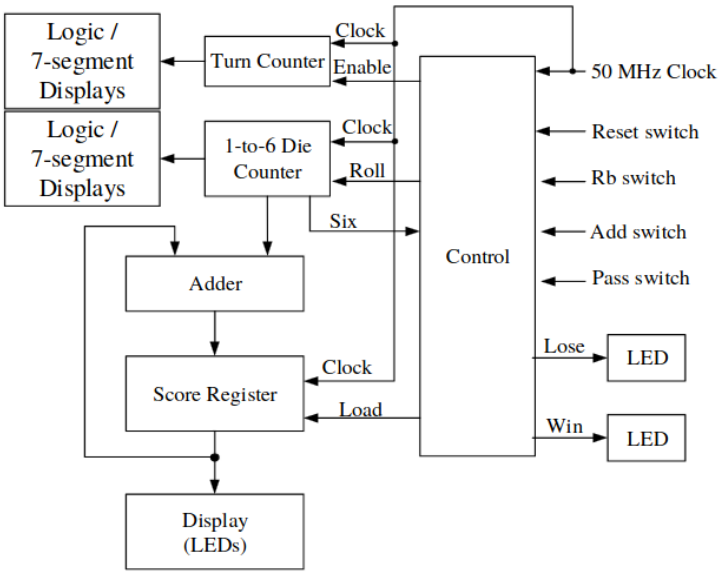
Eric Liu

Objectives

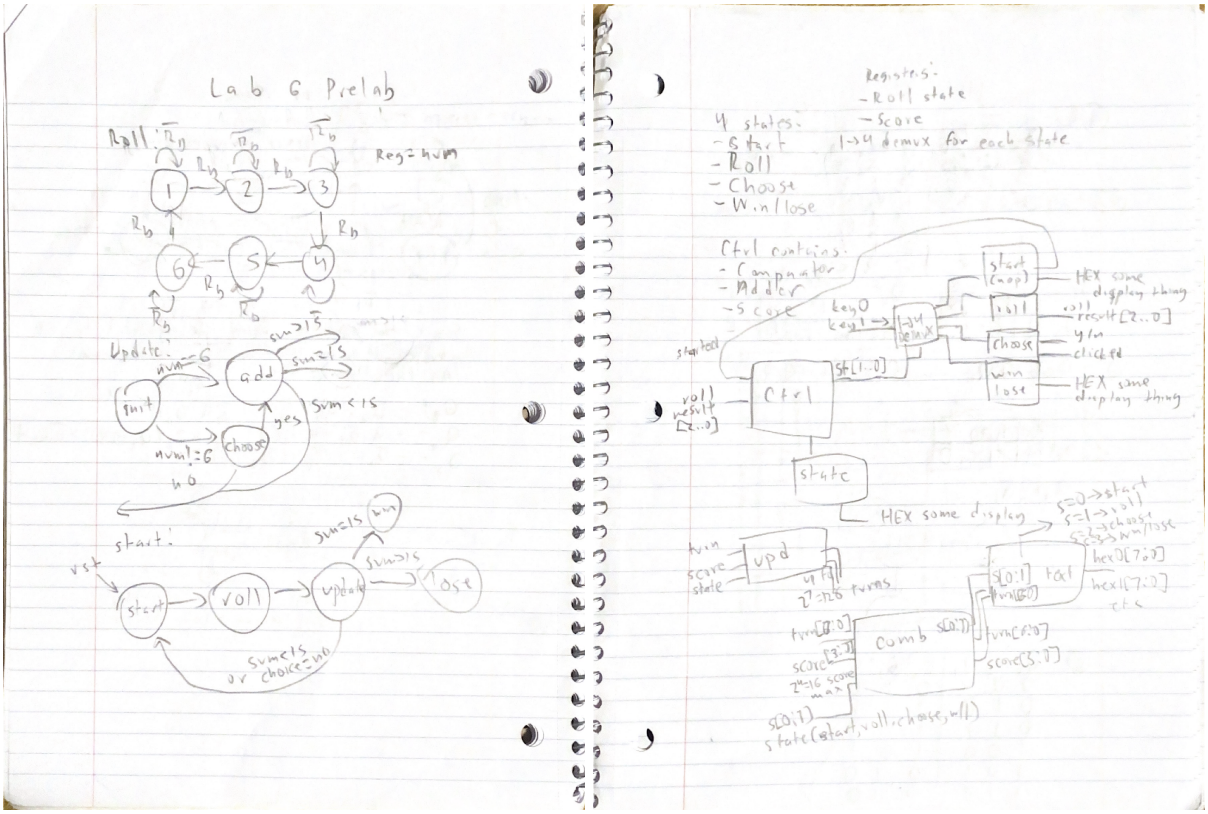
In this lab, we will implement the dice game 15. The game consists of rolling a 6-sided dice and trying to get the exact score of 15. You can choose to keep or discard the roll for strategic purposes, but a 6 is automatically added for an element of randomness. This lab combines basically all of the skills we learned in the previous labs: designing finite state machines, creating components, optimizing k-maps, testing our circuits with ModelSim, etc.

Design and Test Procedure

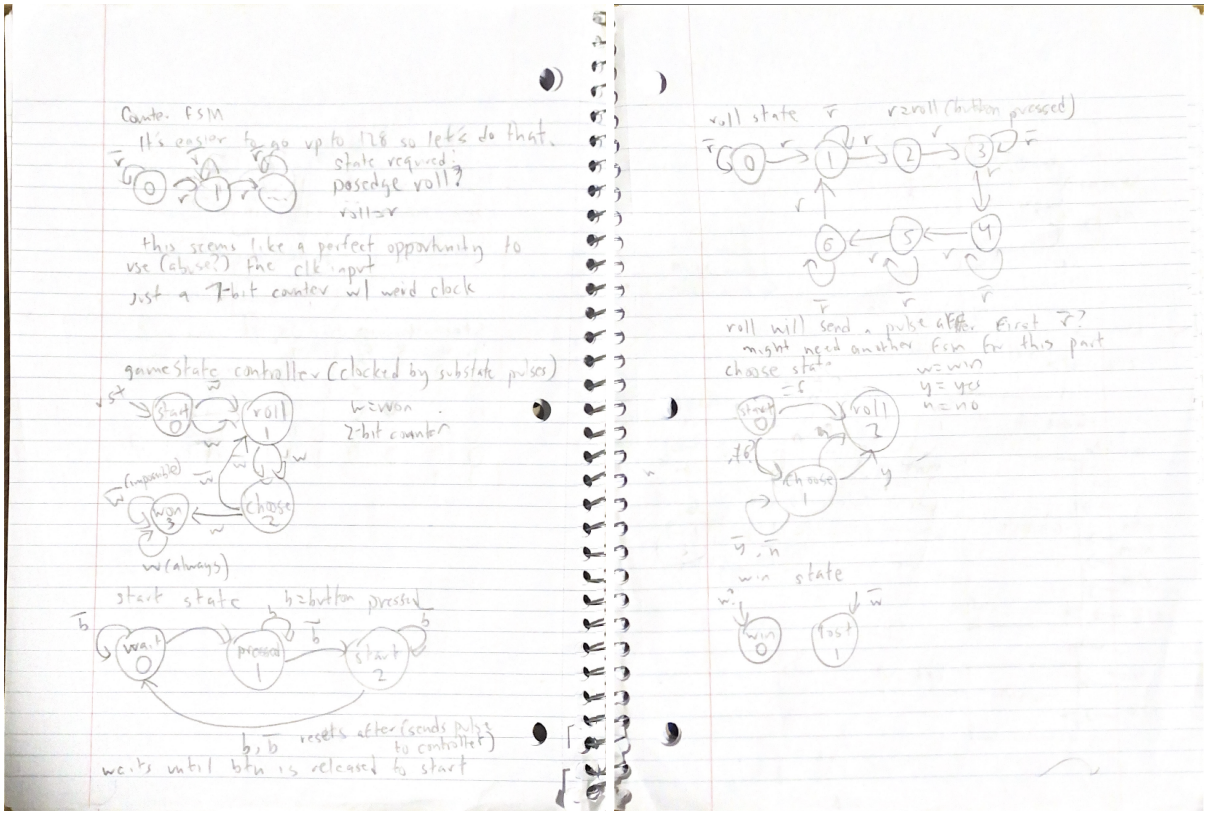
I ended up approaching this lab slightly differently than the block diagram specified in the lab manual:



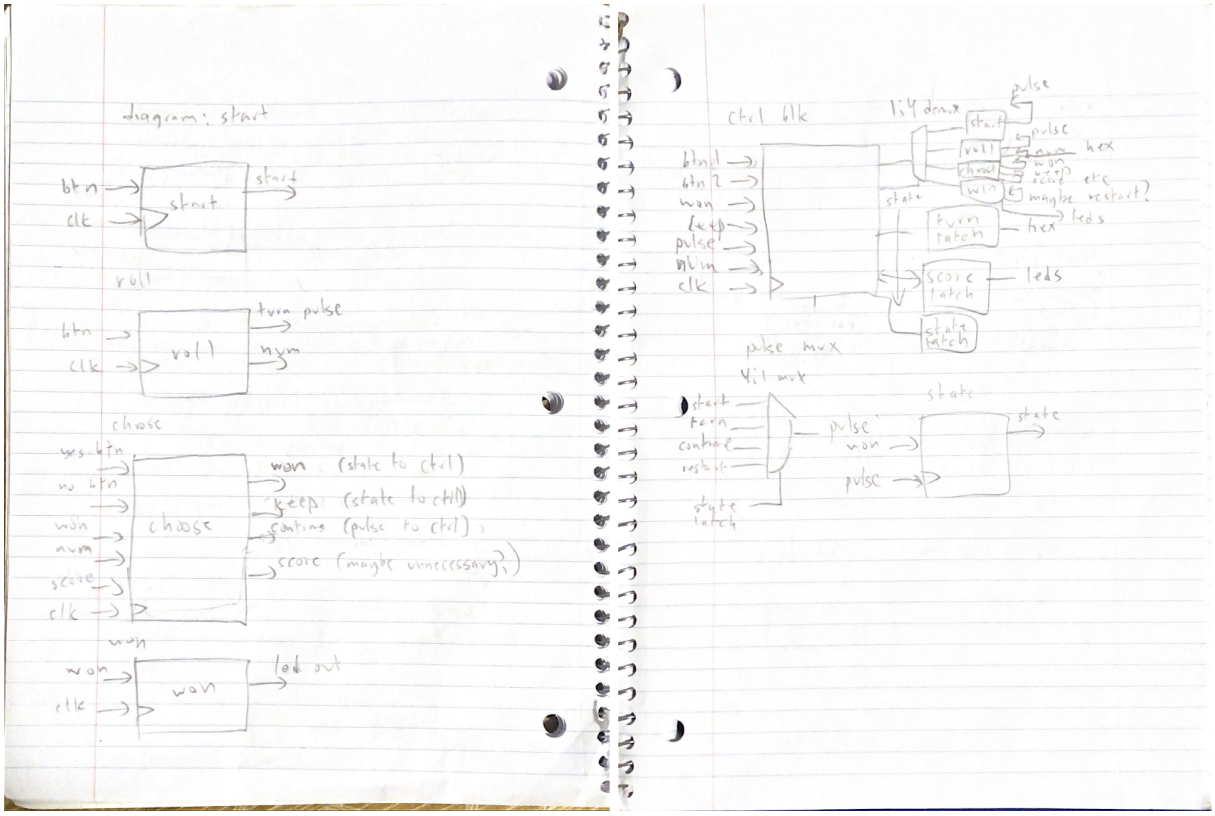
I thought this diagram was a bit unorganized, and instead I broke things up into smaller state-specific logic blocks:



This is the initial prototype block diagram along with some fragments of a state diagram (for each separate component)

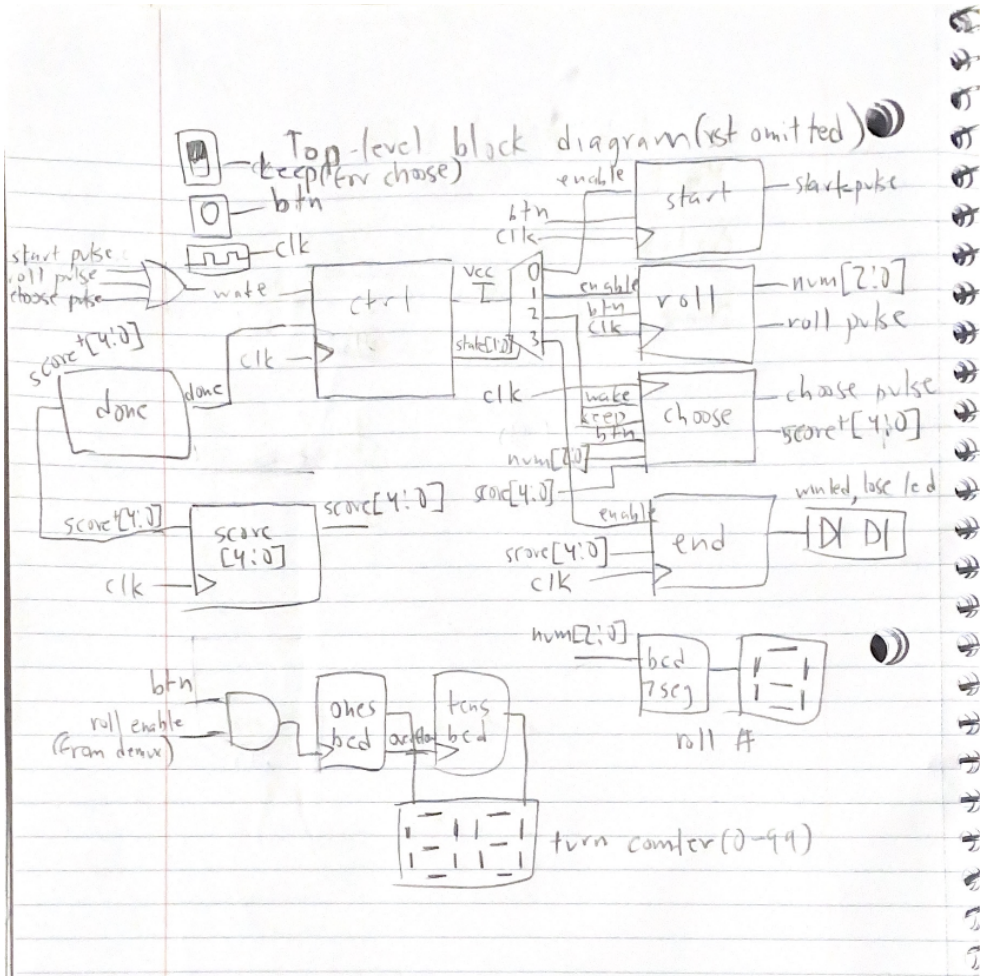


These are more detailed plans for each FSM.



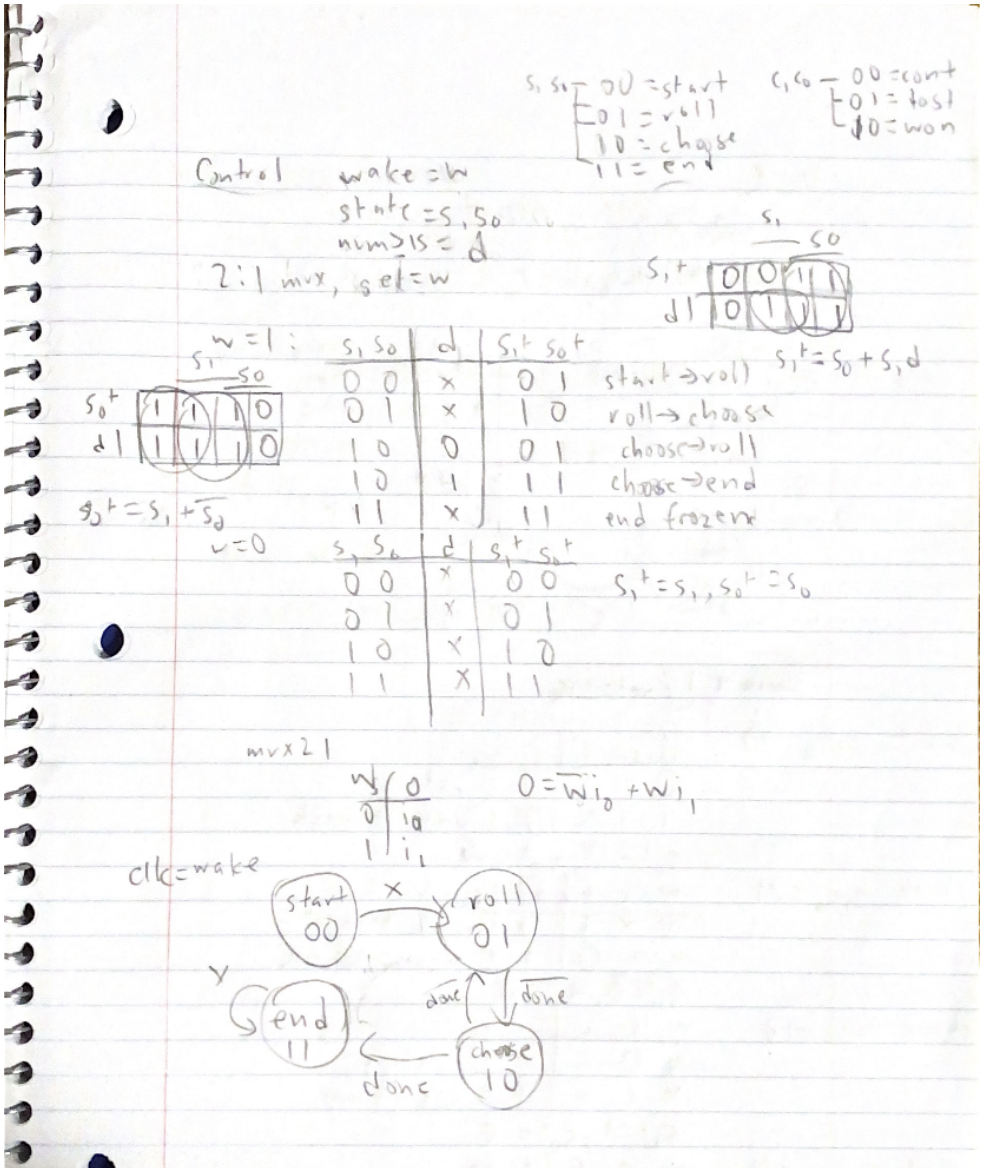
These are more detailed initial block diagrams.

These initial diagrams are a little weirdly coupled, which I discovered during the design process. I ended up changing the inputs to a lot of the modules and such in order to make the design process easier. Here's the final design:

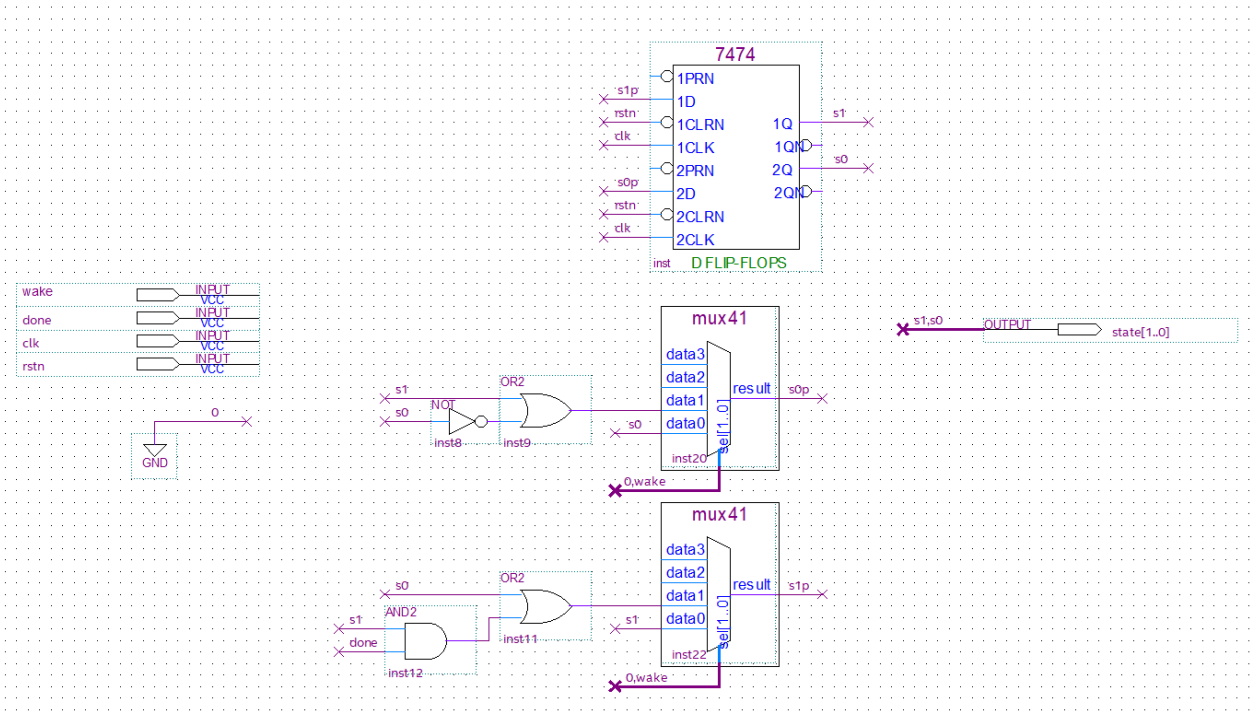


I thought it would be more organized to break down the logic into four different submodules, and implement a controller block that only transitioned from one state to the next based on the outputs of the substrate FSMs.

For the controller:

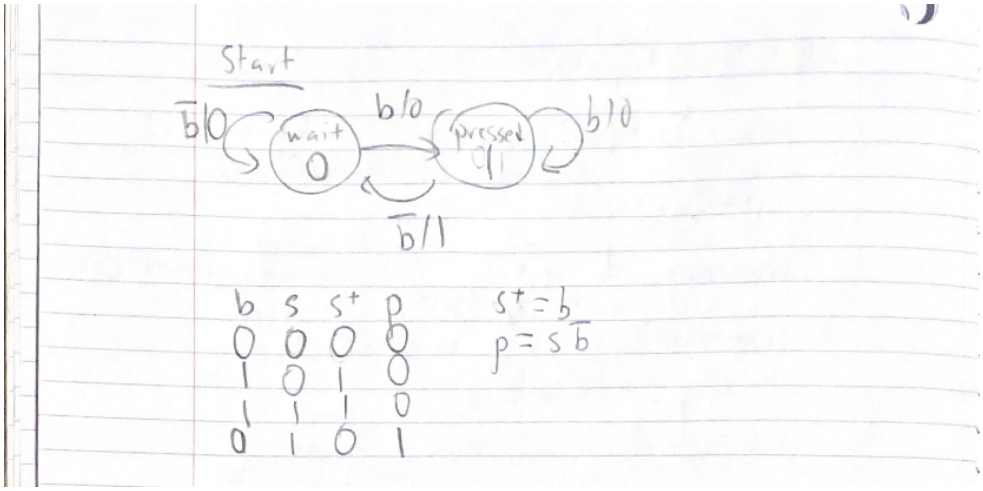


The controller is effectively “clocked” by the wake input on the schematic. I did this by using a 2:1 multiplexer, with the 0 input simply set to the current register state. This makes the FSM really easy to reason about: it only has to worry about whether or not the game has finished. This is the resulting schematic:

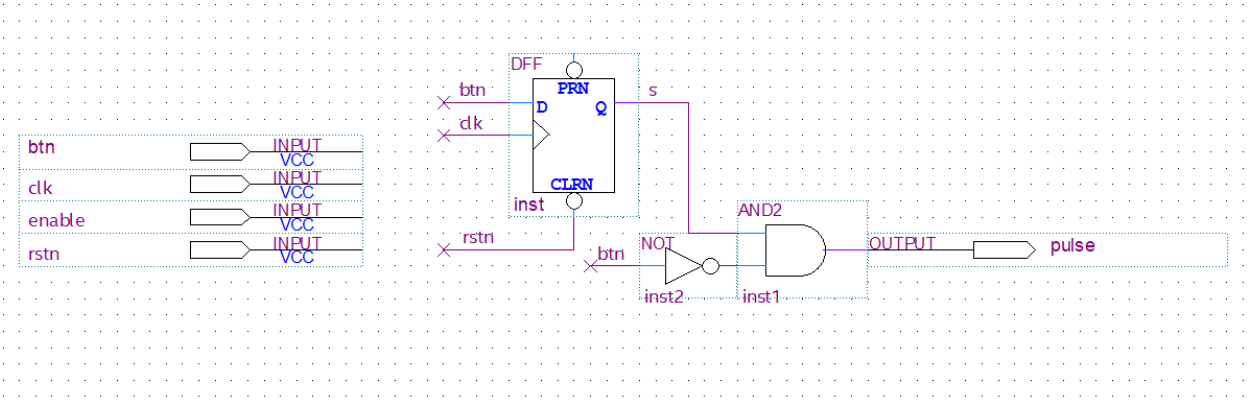


You can see that the logic is really quite simple, and derived from the K-maps in the design document.

For the start state:

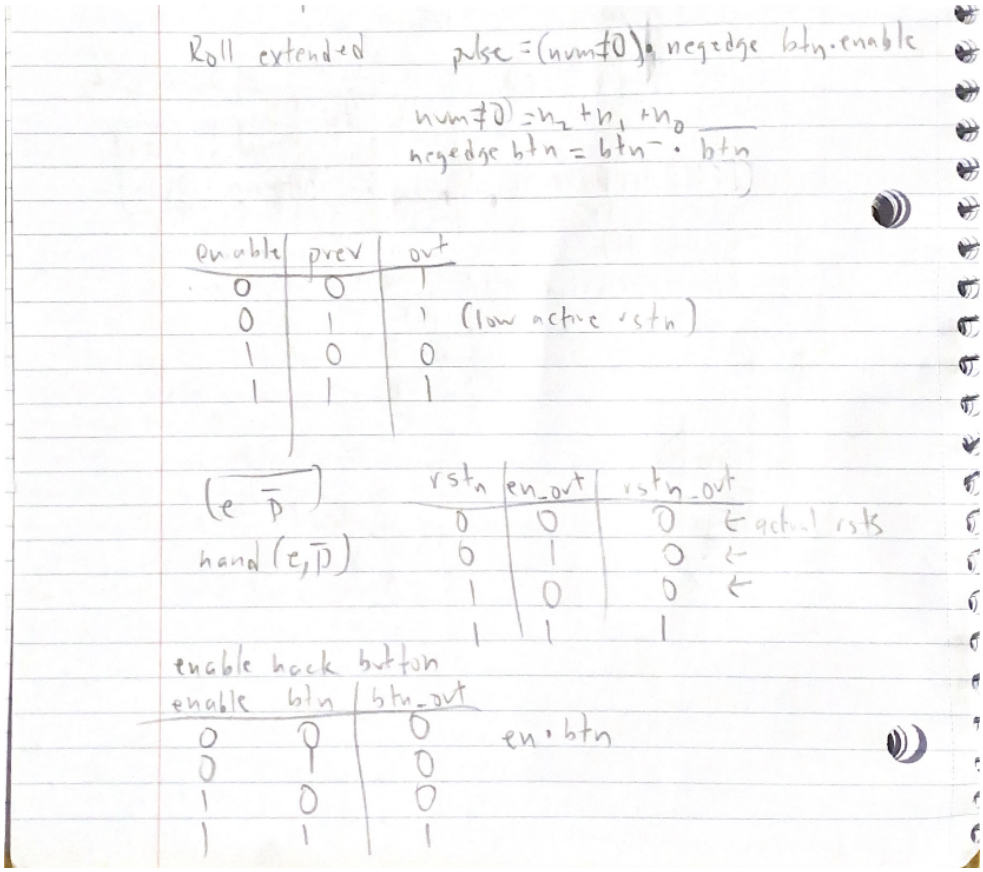


This is not complicated at all, and inspired by the level pulse circuit in the lab supplements. It basically just waits for the button on the FPGA to be pressed down, then sends a pulse once the button is released.
I decided to make this a Mealy machine because it only requires 2 states and it looked easier to implement than a Moore machine, and I was pretty certain such simple logic wasn't going to glitch up too much. This is the resulting schematic:



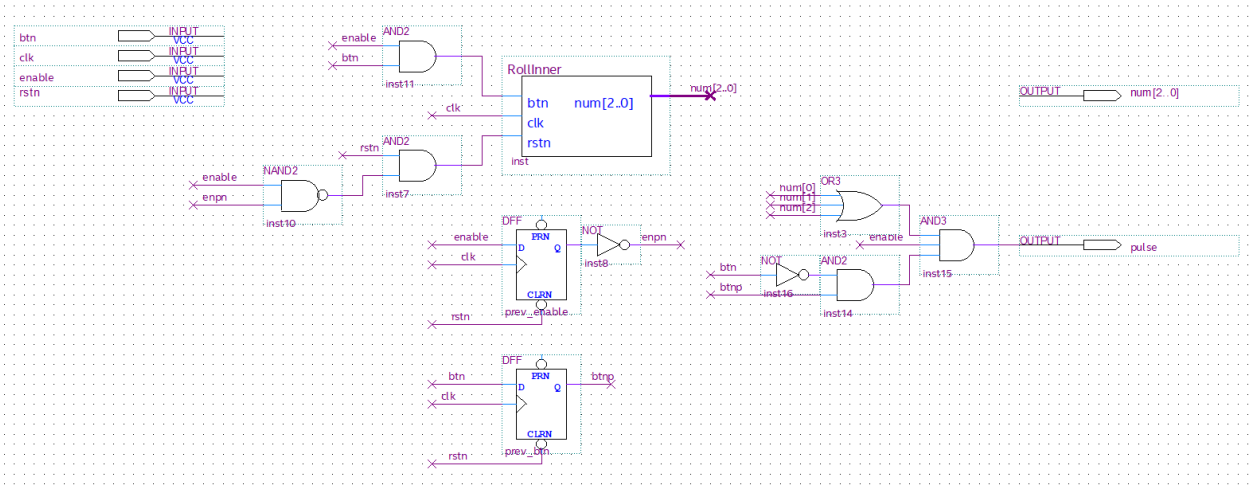
Not complicated at all.

For the roll state:



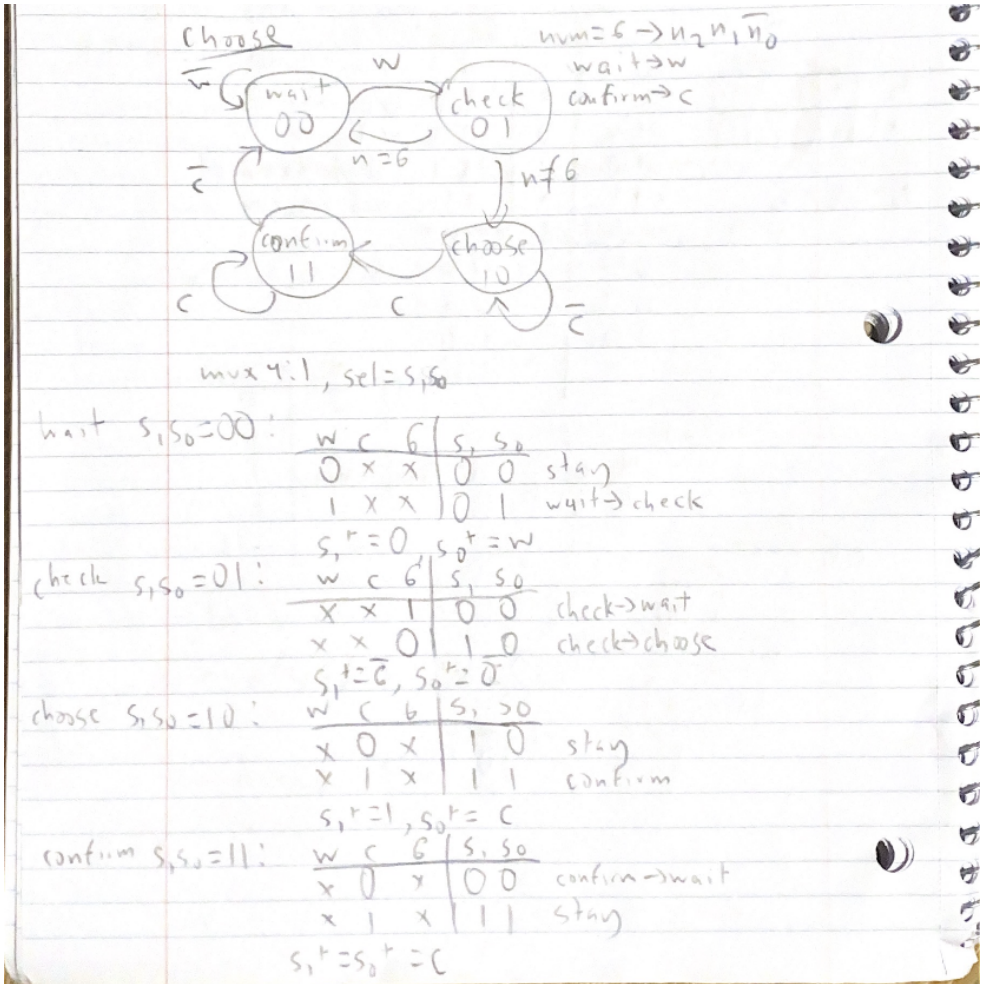
Because this is adapted from the roll circuit in lab 5, all the logic here is just clamped onto the old circuit. The roll circuit was designed to send a pulse to the control circuit after the button is released, so I just created a tiny FSM to capture the release of the button for generating it. I also created another FSM to reset the roll number to 0 when the enable bit flips to positive, so that taking more than one turn doesn't end up with some weird behavior (because if the number isn't reset back to 0, the number will either

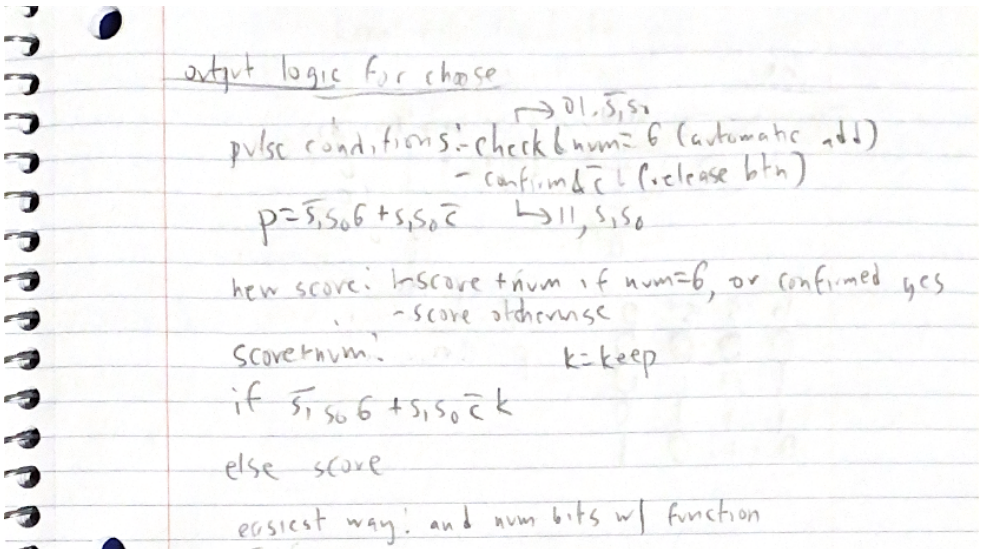
start changing without any input, or the pulse to the controller will occur without the button being pressed). Here's the resulting schematic:



The inner portion is effectively the same as the circuit from lab 5 and is omitted. You can see the two D flip flops for the two tiny FSMs, and a little combinational logic. The logic for the button input forces the button to only work when the roll module is enabled, and the logic for the rstn input allows a reset of the roll number when we detect a positive edge of the enable bit *along* with when an actual reset occurs. There's also some logic to determine when to generate the pulse output, which should only occur when the number is *not* zero, and the button is released.

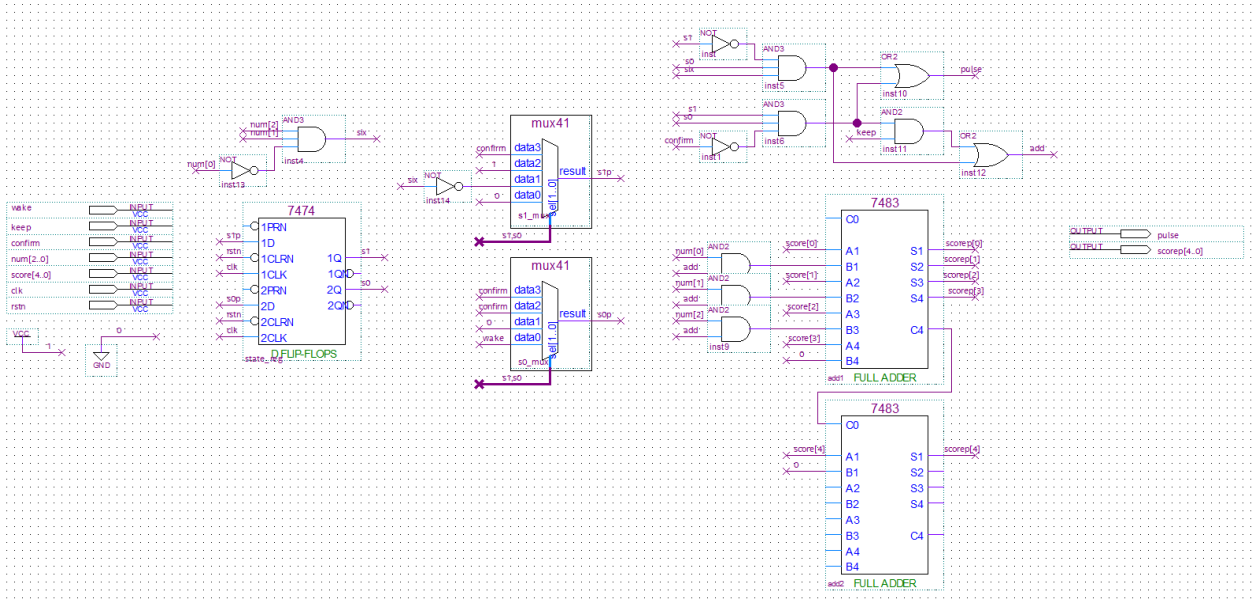
For the choose state:





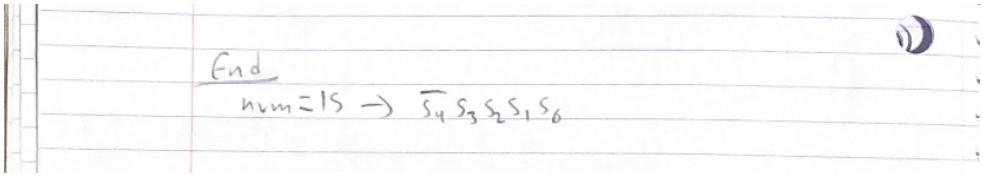
The states for this module are a bit different than the other three submodules, but the idea is that waiting for the enable bit is actually a state. In retrospect, I'm not sure how good of an idea this was, but the K-maps were pretty easy to derive because I used a 4:1 multiplexer, so it didn't end up being a horrible decision anyways. After the FSM is

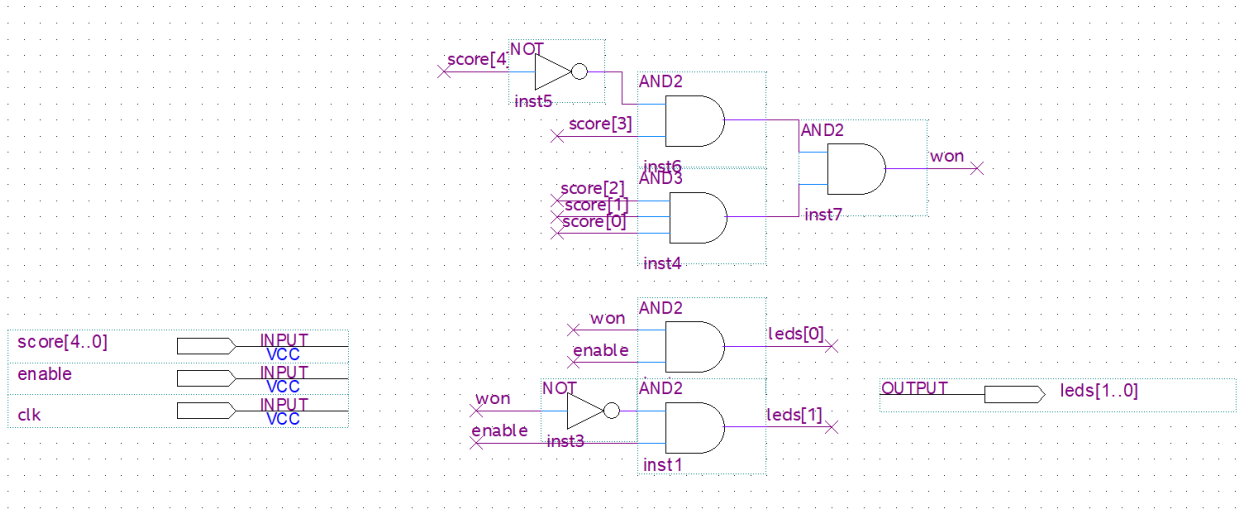
“woken up”, then it spends a cycle checking if the number is equal to 6 (and if it does, it sends a pulse and uses a full adder to output the new score), and then waits for the user to choose whether or not to keep it. Once the button is pressed down, the FSM waits for the button to be released before confirming the choice. Here’s what the schematic looks like:



You can see I used two 4:1 multiplexers for the state transition logic, and some simple combinational logic to determine whether or not to send a pulse and add the number to the score. I didn’t put too much thought into the combinational logic because I thought it was pretty trivial, so the output logic page doesn’t actually have any K-maps, just inferred expressions I assumed.

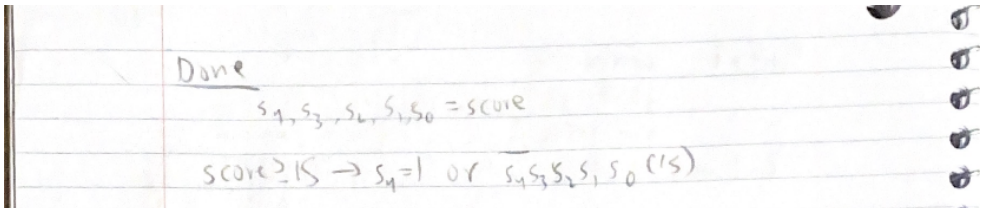
For the end state:

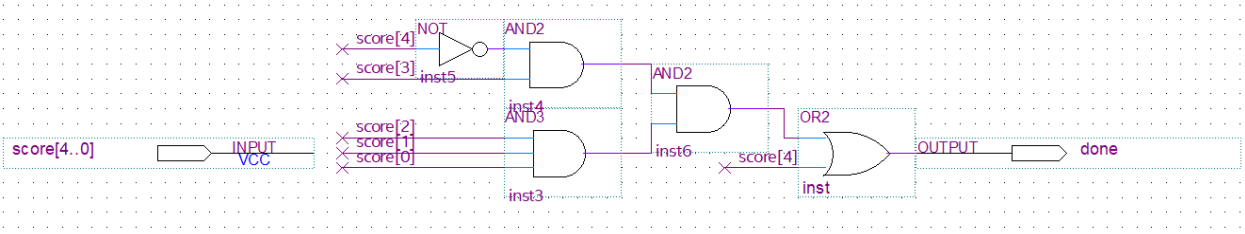




Very straightforward; this is just some combinational logic that checks if the score is 15 and outputs to the appropriate LED accordingly. No K-maps or anything because it's trivial to derive.

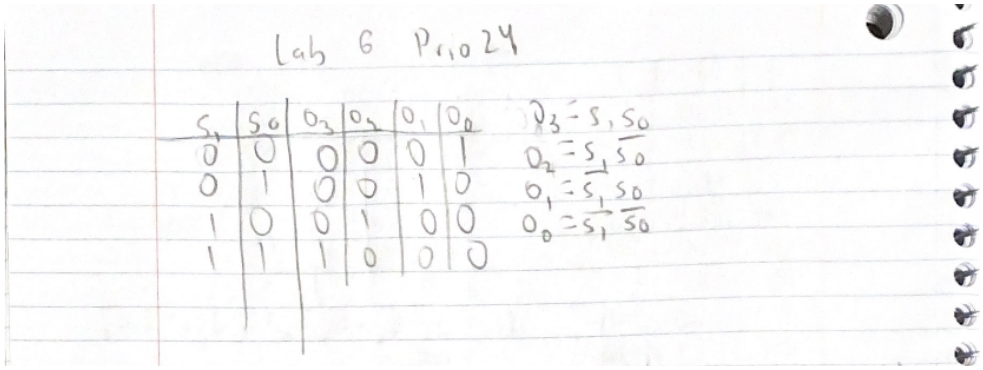
I also had a combinational module to determine if the game should be over (as an input to the control block):

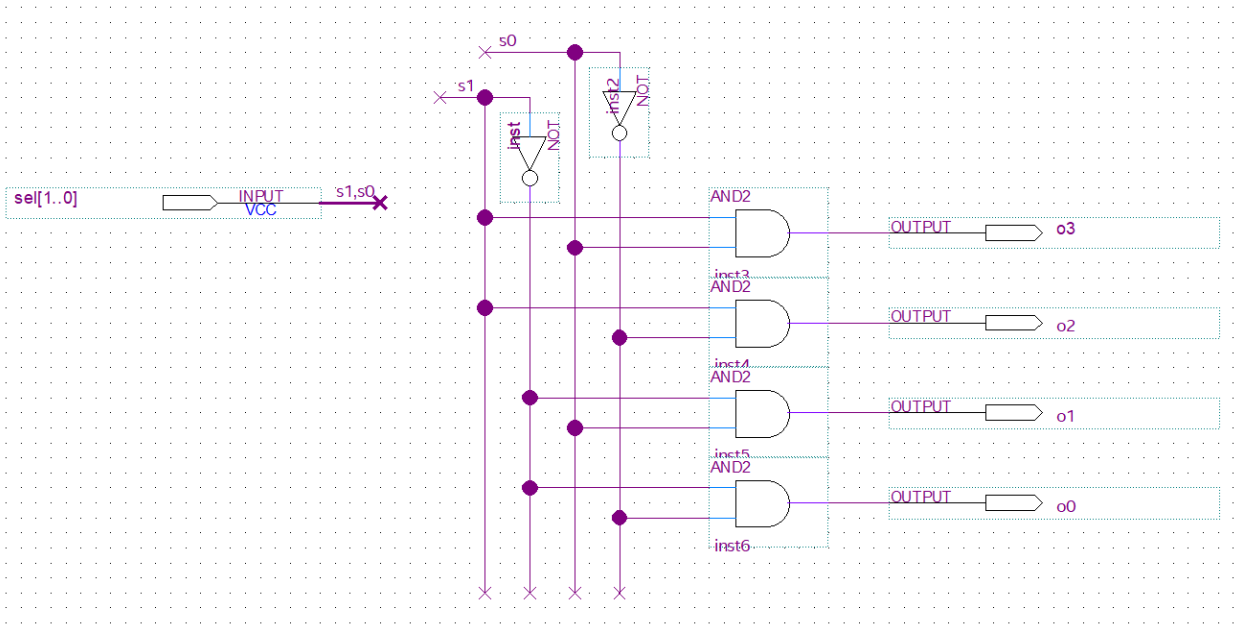




Again, trivial. I knew that values greater than or equal to 16 would mean the top bit is set, and then there's a single exception, 15, which can just be derived from observation. The reason why I didn't incorporate this into the control block was simply because I wanted the input and output wires to be as simple as possible, and a bus is definitely more input than a single wire.

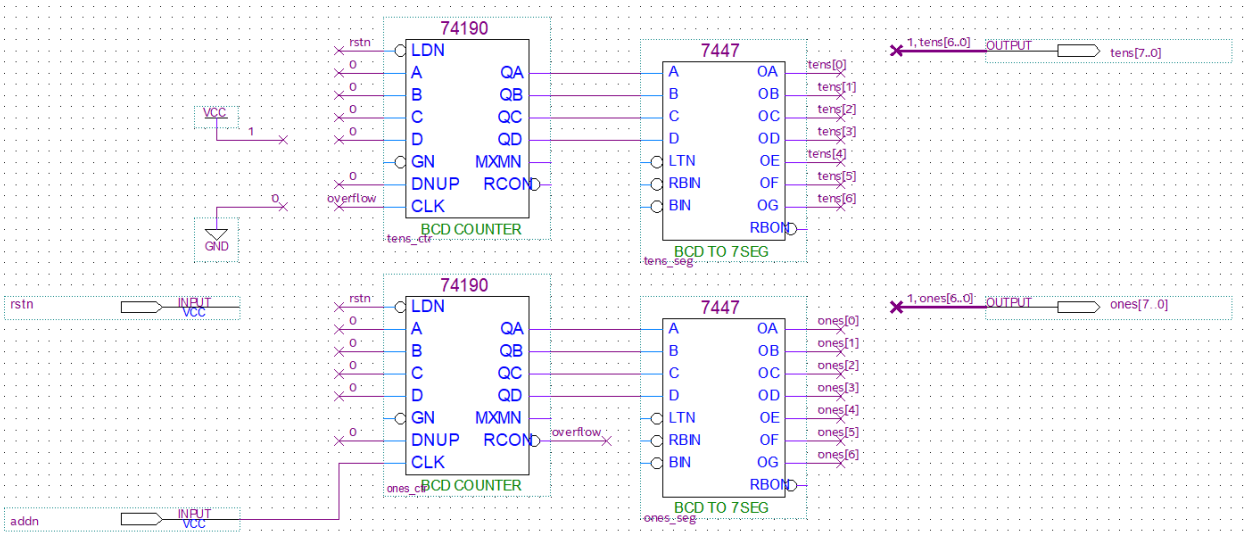
To connect everything together, I implemented a "priority decoder" for the state output:





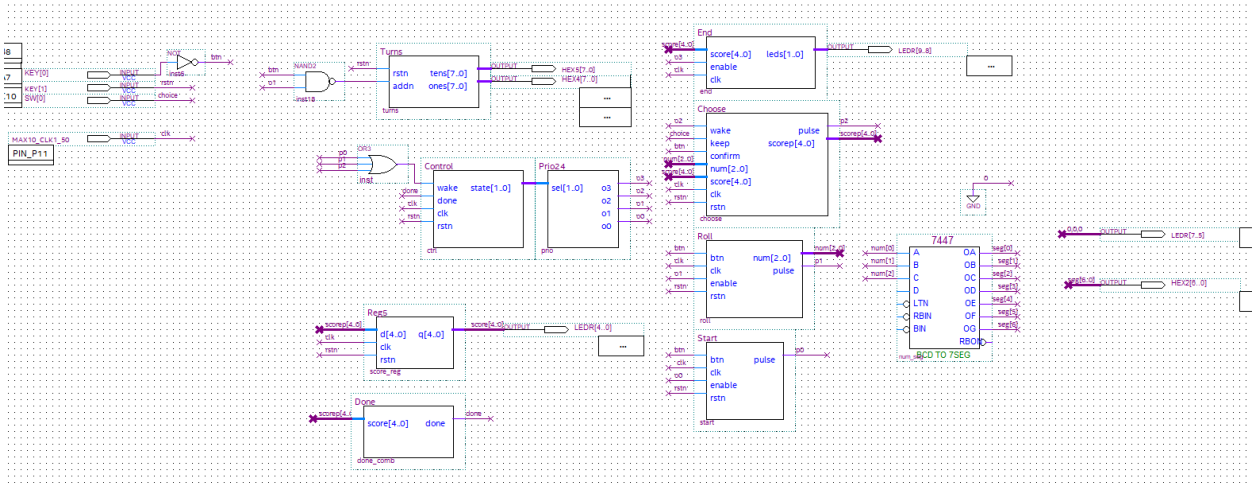
I'm not particularly sure if this is the correct terminology for this type of circuit. I originally wanted to implement a 4:1 demultiplexer, but then I realized the input would always be 1, so I just removed it and created this instead. Not sure what the correct name for this type of circuit would be. 2 bit binary to 4 bit wire output?

I then added the turn counter:



The combinational logic here is trivial. This isn't actually a synchronous module, which kind of breaks the rules of the lab manual, but this will never have glitches (nor have any effect on the rest of the circuit) anyway so it's not particularly important. Basically this just connects the button to the clock of the "ones digit", which makes it increment each time the button is released (since the button is low active, posedge clk = negedge btn). Then, the RCON output of the ones digit is connected to the clock of the tens digit. The RCON output pulses on a cycle when the digit overflows (9 to 0), so this simply increments the tens digit every 10 button presses, as we expect. Then, all the output wires are connected to BCD to 7-Segment display converters, and finally, the rstn input is connected to LDN with the load wires driven to ground, meaning that if the reset button is activated, the counters will load 0 (effectively resetting the turns).

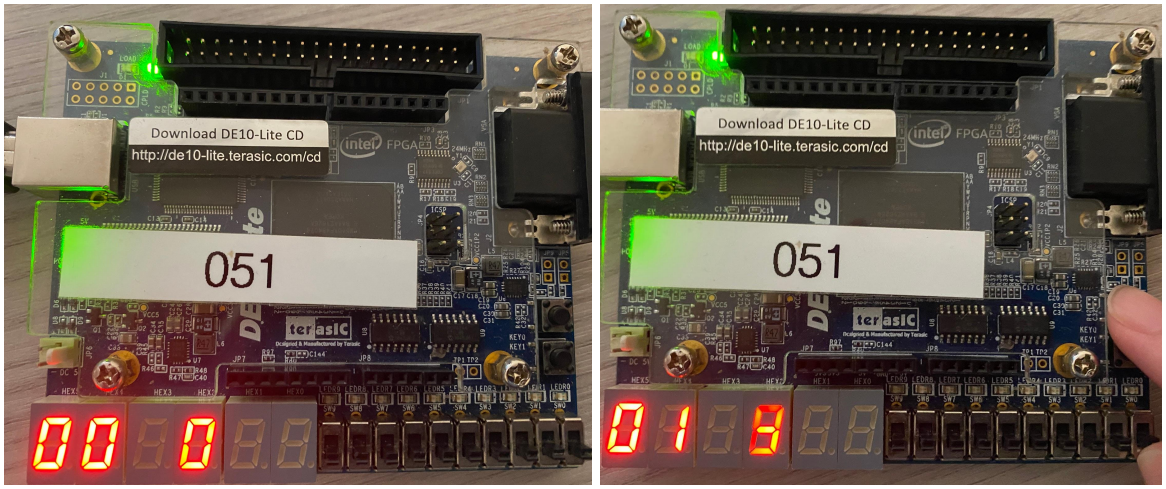
Finally, the top-level circuit schematic:



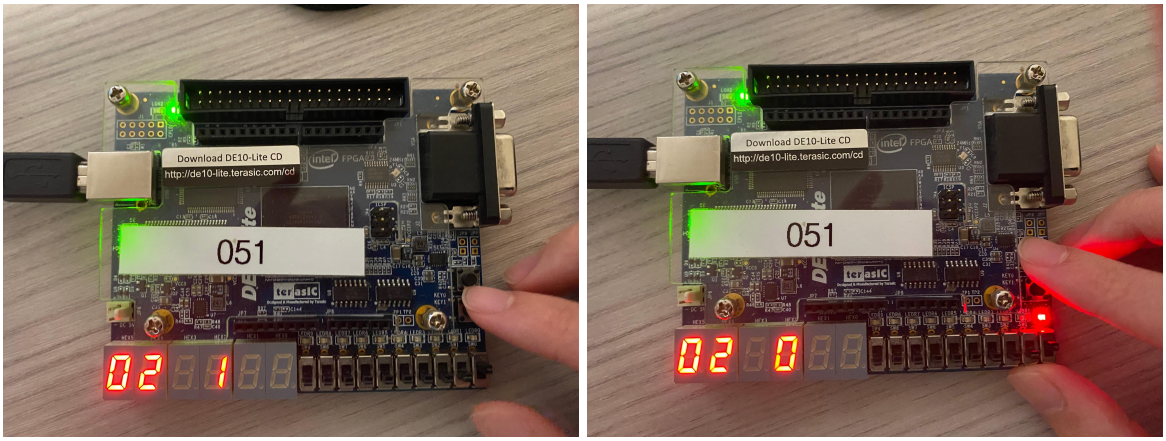
This simply connects everything together. The only notable things are that there's a 5-bit register for the score, and a BCD to 7-Segment converter to output the number more readably.

Results and Answers to Questions

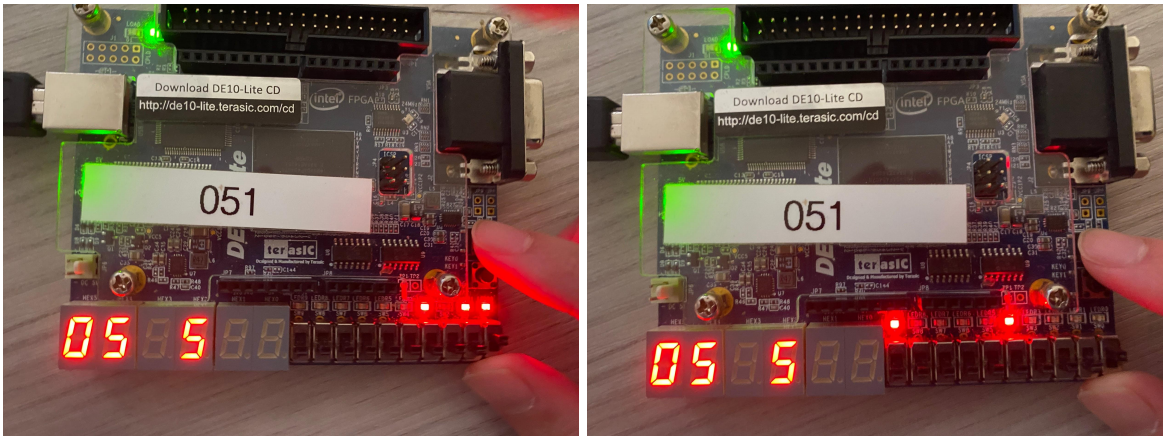
Here's some images of playing the game:



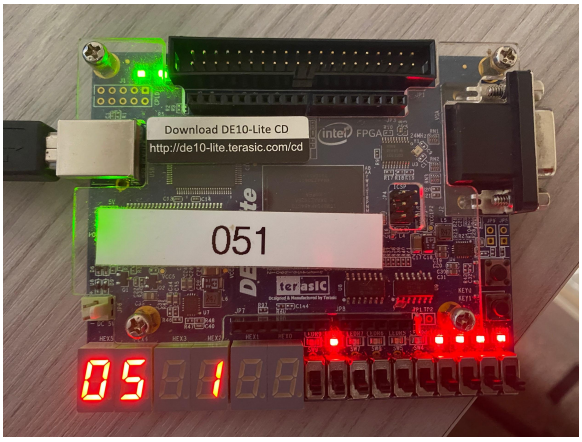
Starting the game and rolling a number.



Keeping a number by pressing the button. Note that the roll button resets to 0 and the score is now 1 (in binary)



Accepting a losing number (11 + 5 = 16, which is higher than 15)! Note that the lose light (LEDR9) is on now.



Winning. Note that the win light (LEDR8) is on now.

Q1: I had multiple FSMs. For most of them I used Moore machines (because they sent pulses and I wanted those to be as consistent and glitchless as possible to spare myself from the headache), but I used Mealy machines for a couple since it wasn't critical for them to glitch and it was easier to implement, such as the start button pulse.

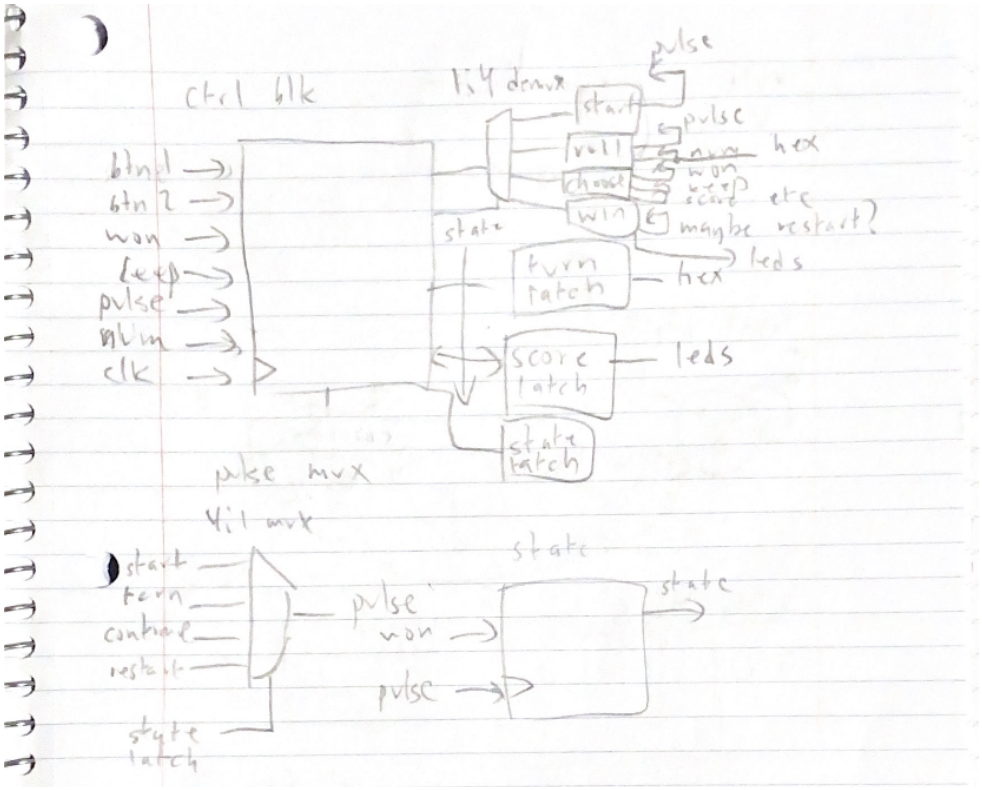
Q2: My design makes it impossible for both switches to be both activated, because it involved a switch determining whether or not to keep the roll and a button that confirmed the decision. You can't have the switch output both low and high at the same time, so I didn't have to deal with any invalid state where both add and pass are clicked.

Conclusions

I learned a lot from this lab: I got to piece together all the knowledge that we had learned previously, and also I got a lot more experience with creating components that interact nicely with each other. I also wrote a lot of test scripts and got a lot more

experience on how to use ModelSim correctly to make sure my circuits worked properly. Good, modular design and correctness are both very important to both EE and CS, so both these skills are super important to possess. There wasn't actually many new concepts at play here, it was mostly just choosing how to design a circuit and figuring out the correct process to prototype and test them.

A lot of things went wrong in this lab. When I was designing individual components, I initially guessed what the inputs and outputs of each component would be:



This turned out to be a pretty bad design. Specifically, the choose module had way too much to deal with (it was originally supposed to determine if you had won or not), so I had to go and rework it to make it less difficult to implement and easier to reason about.
I also had some troubles testing the circuit. I forgot that the rstn input had to be driven high when testing the top-level schematic, so I was confused as to why the state of the game was stuck at 0. Turns out, I had set rstn to 0 so it would continuously drive all the registers in my circuit to 0 and no inputs would end up working. I quickly discovered my mistake though, so overall this wasn't a huge hindrance.
I also had some issues when I finalized my design: the control module wasn't switching to the end state when the score exceeded 14. I realized later on that feeding the score input to the done logic was incorrect, because the score wouldn't be updated until the next clock cycle, and the controller would already have determined whether or not the game should end, and you would end up being able to roll an extra, unintended time. I

ended up connecting the new score output of the choose module instead, which worked as expected.

Although I think I learned a lot, I also think I over complicated the design of this circuit quite a bit. I added a “start” submodule because I thought it would be cool to output some text and also to break the game into different parts, but in retrospect this wasn’t a super good idea because this didn’t end up being super useful as I didn’t end up implementing any cool text to be added. Overall though, I think my circuit turned out to be quite readable just from looking at the top-level schematic, which I am pretty happy about.