**Lab 5 Report**

**Part One: Flip Flop Implementations**

To start the implementation I copied the design of the Master-Slave Flip Flop configuration as shown in the lab. This lead to a correctly functioning D Flip-Flop and an SR Flip-Flop with the corner case unhandled. In order to transform the D Flip-Flop into a T Flip-Flop I determined that since T = D’Q + DQ’ which really is just T = DxorQ. Since this is the case, all I had to do was use an xor gate using my D flip flop and my output, Q of the slave latch as the inputs to the circuit where my original input D, used to be for the D flip flop. This modification made a successful conversion from a D flip-flop to a T flip-flop.

Transforming the SR flip flop into a JK flip flop took a little more logic. Once again I had to replace the inputs. With additional glue logic the S needs to transform to the behavior for a J and the R needs to transform to the behavior of a K to go from an SR flip flop to a JK flip flop. The glue logic I added to make this was to have my new S from the SR flip flop transform to

S = (J+Qslave)(K’+Q’slave­) and R = (J’ + Qslave)(K + Q’slave)

These equations mostly mimic the behavior of an SR flip flop, except for the case when J=1 and K=1, In this case S really just depends on the value of Q’slave which forces S to be 0 is Qslave is on, and S to be 1 if Qslave is off. On that same note J=1 and K=1 will also mean that R really just depends on Qslave­. So this means that in the case of J=1 and K=1, S=R’ which effectively forces the logic for a toggle.

For the asynchronous set and reset functionality of the SR and JK flip flops, I examined the NOR gates of each master and slave latches, as the inputs of those gates are what propagated value to the output. I had to add logic independent of the clock, so adding extra inputs to these NOR gates to accept the values of the asynchronous set and reset input seemed the most efficient way to change the outputs of the latches independent of the clock. When testing the flip-flop I noticed that the input to the NOR gate of master slave latch for Q was the compliment of the NOR gate input for Q’ in the slave latch. The reason why this is comes from the nature of the propagation of values depending on the clock edge. Because of this I had the inputs go to the opposite NOR gate in the slave latch from whatever input of the NOR gate I used in the master latch.

The corner case in the SR flip flop lied within that fact that when we toggle the set when the clock edge is low and then set it to S=R=0 before flipping to the positive edge of the clock, the values in the master latch end up being different than the value in the slave latch. The set stays on in the master when S=R=0 but these changes never propagated to the slave on the set during the low clock cycle. So when the clock is flipped these new values are set in the master latch instead of preserving state like it should have. In order to overcome this I added extra AND gates in the outputs of the master latch which checked the case of S=R=0 and also the state of Q and Q’ for each of the respective extra AND gates. When these AND gates noticed the S=R=0 condition it would then ensure that any output in the master latch would be overridden with the current values in the slave latch, which effectively preserved the start of the flip-flop.

**Part Two: Mealy and Moore Machines**

The state tables for each Mealy and Moore FSM are as follows (one can trivially draw the FSM picture from these tables)

For the Mealy FSM:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Present State, Q | Next State, Q+  I = 00 I = 01 I = 10 I = 11 | | | | Output, Z  I = 00 I = 01 I = 10 I = 11 | | | |
| S0 | S0 | S0 | S0 | S1 | 0 | 1 | 1 | 0 |
| S1 | S0 | S1 | S1 | S1 | 1 | 0 | 0 | 1 |

Adjacency encoding was very trivial here. Since there are only two possible states, I used one bit to name them which automatically made them adjacent. In this table the state had the encodings of S0 = 0 and S1 = 1

For the Moore FSM:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Present State, Q | Next State, Q+  I = 00 I = 01 I = 10 I = 11 | | | | Output, Z |
| S0 | S0 | S2 | S2 | S1 | 0 |
| S1 | S2 | S1 | S1 | S3 | 0 |
| S2 | S0 | S2 | S2 | S1 | 1 |
| S3 | S2 | S1 | S1 | S3 | 1 |

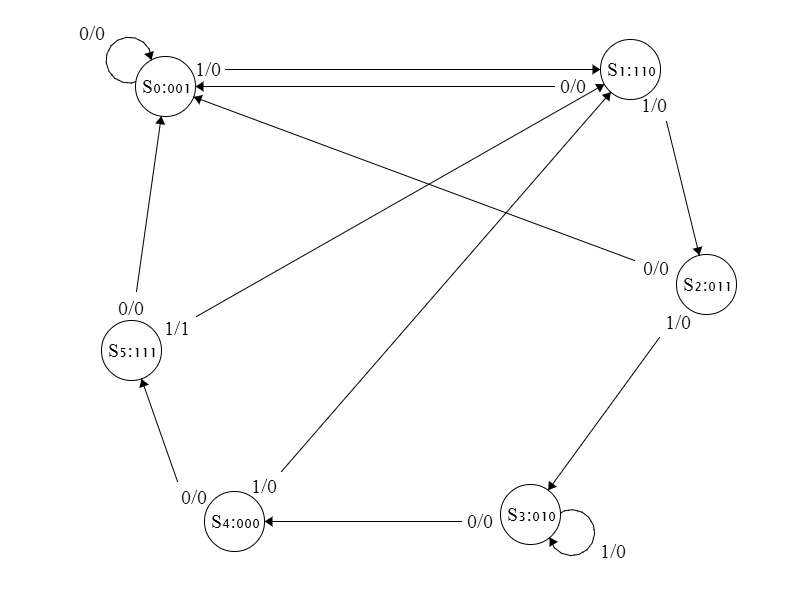
In order to calculate adjacency we can notice that S0 ­and S2 have S1 as the next state on the same input, so they have priority one adjacency assignment. The same condition applies for S1 and S3. For priority two, S2 andS1 are next states of S3 on hamming adjacent inputs. We also find that S2 andS1 are next states of S0 on hamming adjacent inputs. Given these heuristics, we find that S0 ­and S2 must be hamming adjacent and also S1 and S3 both in priority one. Given this a valid encoding scheme is S0 = 00, S2 = 10, S1 = 01, S3 = 11 which also happens to be a binary representation of the decimal value indicating the state number.

State minimization for the mealy and moore FSM’s were done using a state equivalence table. I first looked for states with different inputs. If they did, we can determine that they can’t be equivalent. I then check to see if the same input corresponded to the same outputs for each state, if they did not then I would also determine those states to be nonequivalent. Any states which I did find to have the same inputs corresponding to the same outputs going to the same next states I marked as being equivalent states.

**Part Three: Sequence Detector Implementations**

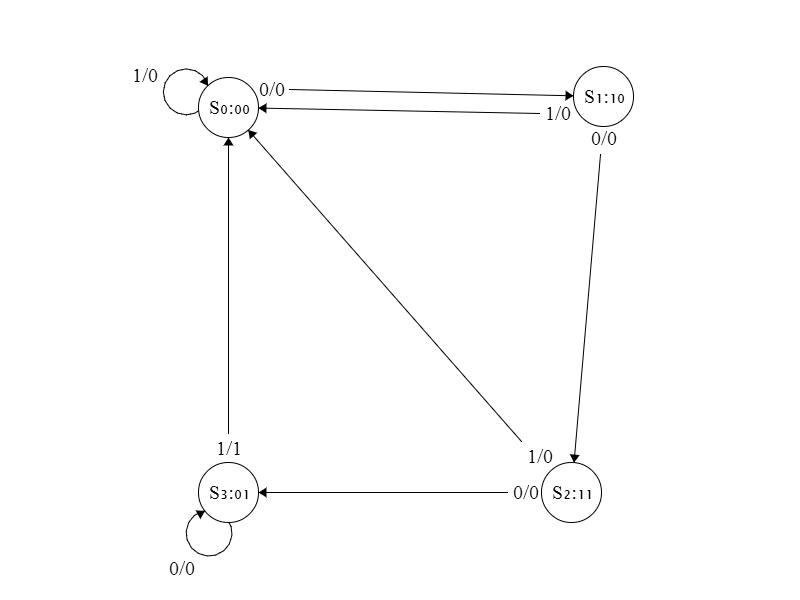
(Images courtesy of an online FSM maker <http://madebyevan.com/fsm/> using HTML5 Canvas)

**Part 3A: Sequence Detector for “111001”**



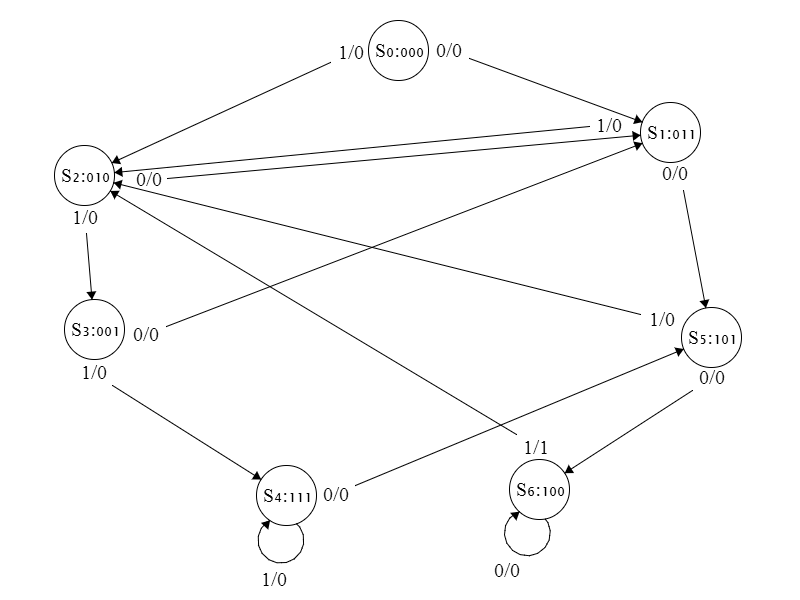
For priority one adjacency we find that S5, S2, S1 all have the same next state S0 on the same inputs and thus have a pair which we want adjacent. We also can see that that S0, S5, S4 all have the same next state S1 and therefore also have an adjacent pair. For priority two we can see that S0, S2 are next states of S1 on hamming adjacent inputs and are a possible pairing. The same also goes for S0, S3 being next states of S2, S1, S5 being next states of S4, S0 and S1 being next states of S5. For priority three we see that S0 thru S4 may all be hamming adjacent. Since S5 and S1 are pairs in both priority one they get the most precedence for adjacency. Also since S5 in priority one is not in priority three we want the other two state in priority one to be adjacent since they are also in priority three. This means that S0 and S4 must be adjacent. This leaves S2 and S3 left over, which should be adjacent because of priority three. Given these heuristics, we may derive the state assignments as expressed in the diagram above.

**Part 3B: Sequence Detector for “0001”**

****

For the priority one adjacency we can see that S1, S2 have the same next state S0 on the same inputs and that’s all we have for priority one. In priority two, according to the same adjacency logic as described in part 3A of the FSM we find that S0 and S2 are adjacent pairs because of S1 and S0 and S3 are adjacent pairs because of S2. For priority three, since S0, S1 and S­2 all have the same outputs they also have adjacent pairs. From this information we can find two unique adjacent pairs, one pair in priority one, namely S1 and S2 and another pair in priority two, namely S­0 and S3. These pairs are given the proper adjacent encodings as shown in the above diagram.

**Part 3C: Double Sequence Detector for “0001” and “111001”**

****

For the priority one encoding we can see that S0, S2 and S3 can pair up somehow since they are both going to S1 on the same input. We can also see that S0, S1, S5 and S6 have some pair combination since they are all going to S2 on the same input. Lastly we see that S1, S4 go to S3 on the same input. We can’t know exactly how to pair these until we analyze priorities two and three. For priority two we need to look at what states are next states of the same state on hamming adjacent inputs. This happens with S1 and S­2 from S0, S5 and S­2 from S1, S1 and S­3 from S2, S1 and S­4 from S3 and finally, S2 and S­6 from S0. For priority three we need to consider all states pointing to next states on the same outputs. We can notice that the all of the states have the same output except for S6 which may output a 1. Since this is the case for priority three we may say that all states from S0 to S5 may be adjacent. Given this pattern we find that it does not matter which pair we choose for S0,S2, S3 in priority one since we don’t find any of these terms paired in priority two and they all exist in priority three together. The terms I chose were S0 and S2 to be hamming adjacent pairs as seen in the above diagram leaving S3 as a loner to take whatever state encodings are left over after all of the other pairs have been encoded. We can also see that in priority one S5 and S6 should have hamming adjacent state encodings. Lastly, S1 and S4 need hamming adjacent encodings as they exist both in priority one and priority two. This exhausts all of the possible pairings and they now may be assigned their corresponding hamming adjacent states as shown in the diagram above.

To design this third sequence detector, instead of merging the FSM’s from part 3A and 3B I decided to start the design from scratch and come up with a viable solution. I did so with the mentality that there should be no more than seven states in the final machine realization. We know there should be no more than seven states in the final FSM due to the subset relationship of the two sequences being detected. The sequences 111001 and 0001 have a 3-bit sequence in common. Namely the sequence 001 is a subset of both sequences. Since this is the case we need exactly 3 states to recognize the sequence 001. After we figure out these states we still need to detect the 111 in the sequence 111001 and the first 0 in the sequence 0001. It takes another four states to realize these remaining bits creating a total of 7 states that are needed to recognize both of these sequences in the same FSM. Once that is done, one can then check whether or not the functioning FSM using a state equivalence table, which will show that none of the states in part 3C can be merged.

**Part Four: Verilog Code Modification**

In order to properly modify the circuit so as to reduce the number of Shift cycles, first we had to overload the Load cycle with a shift whenever there is a no operation step. In multiplier.v there is a variable called doShift which was set equal to a Shift step. Noticing this, I knew we needed to shift whenever there was a no operation so I OR’d the Shift logic with the not of opDone to indicate that it should shift on a no operation step. Once this was implemented, in Controller.v we still needed to override the Load step. In order to do this the program checked whether or not a shift was done, if a shift was done then it will know to go to the load step and if a shift was not done, meaning that it was not on a shift step and not an a no operation step, then the load cycle would ensue. There was also a final modification that had to deal with in the last step of the process which was to check if a load was done, for if on the last step a load was done it needed to do one final shift to obtain the correct value. The average number of cycles in the test bench was 17 clock cycles.