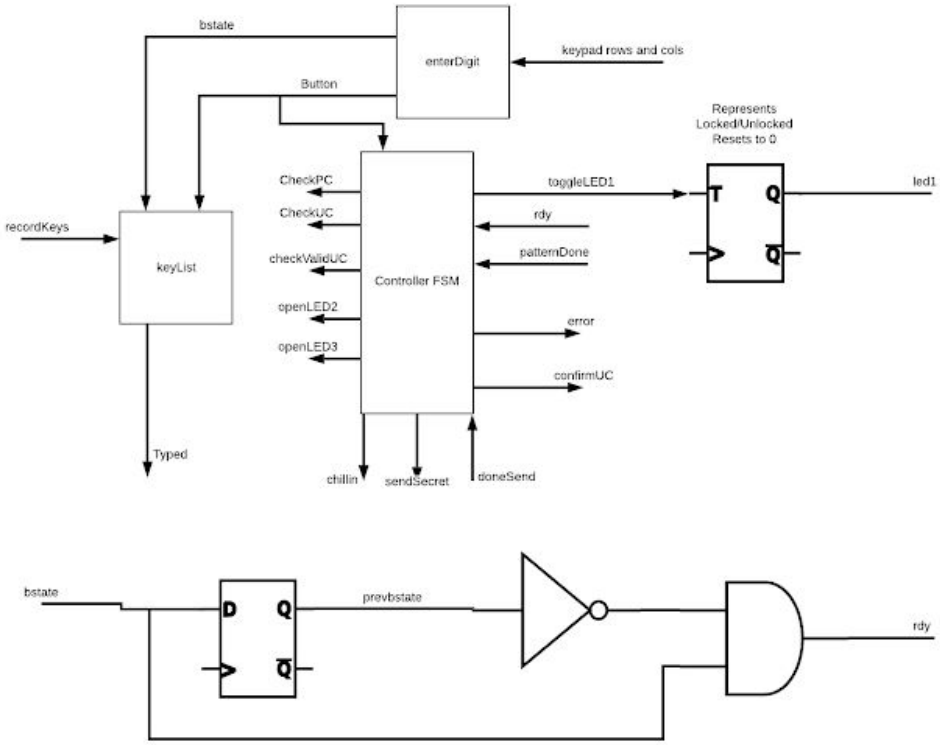
Blake Lazarine and Sahil Dani's VeriLOCK

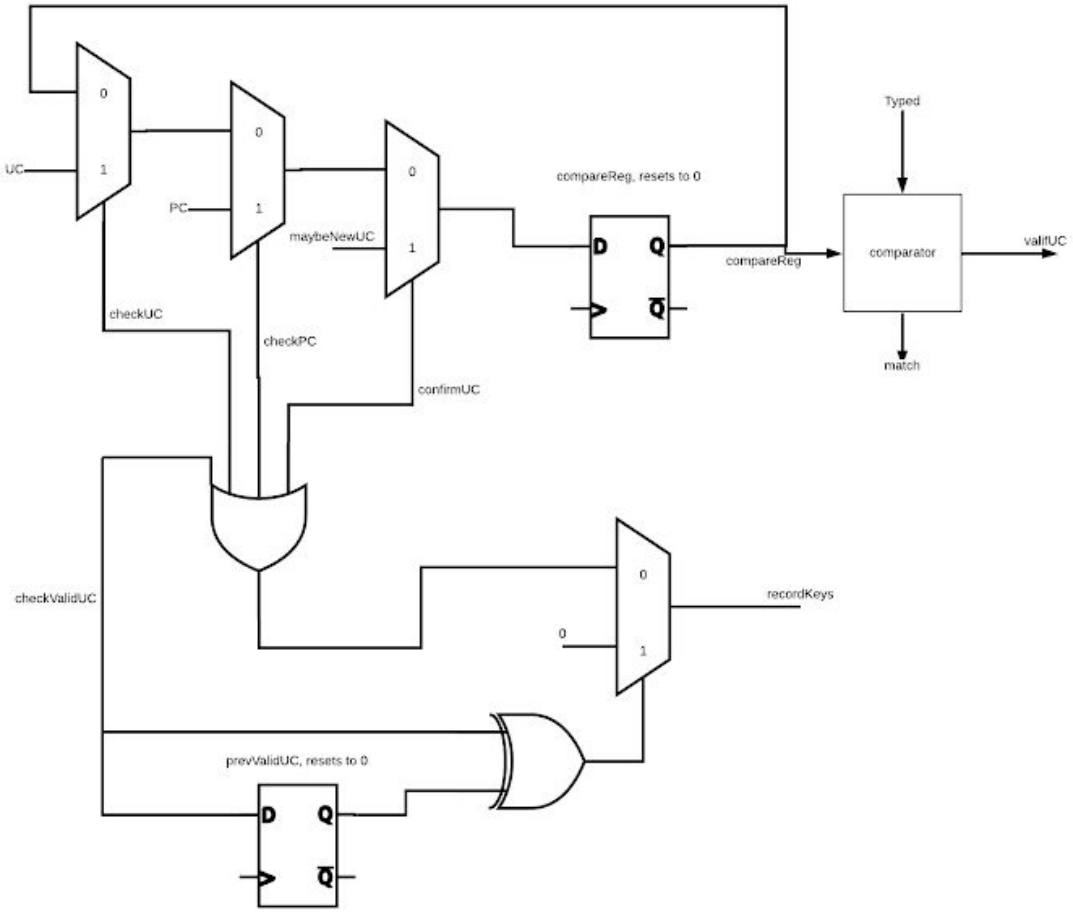
Top

The top module is used to act as an interface for all other modules. It processes the outputs of various modules and feeds the result into other modules as inputs or uses them to determine the state of lights.

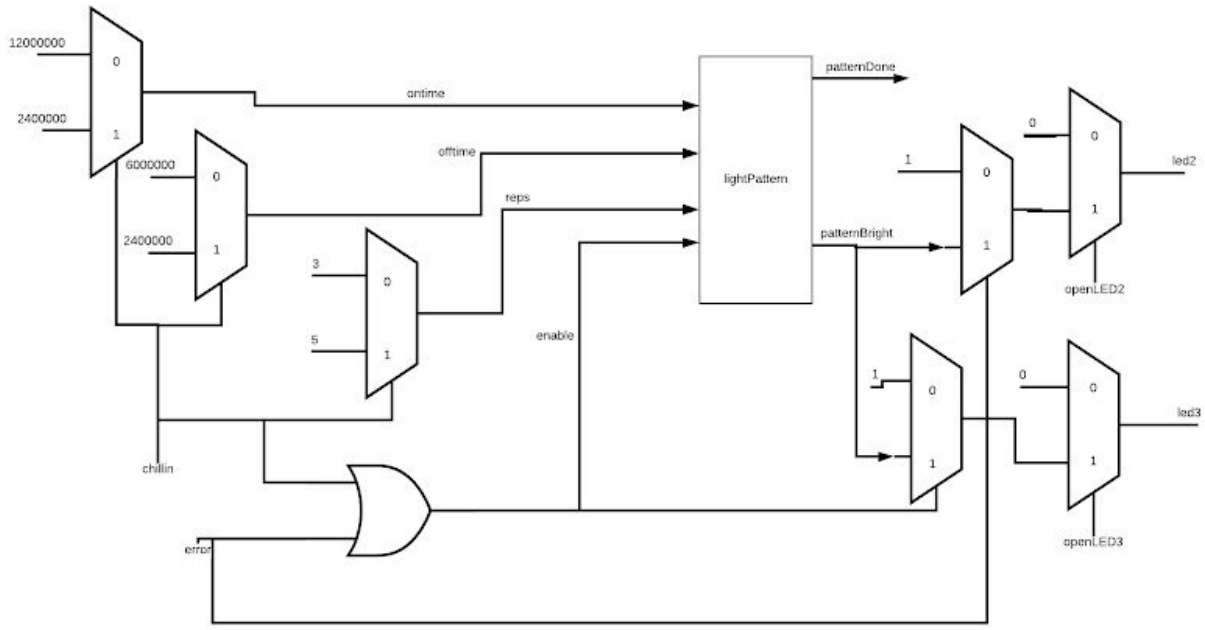


Above is a part of the top module which contains the controller, the enterDigit module, and the keyList module, along with all their inputs and outputs (designated by direction of arrow). The rdy value is intended to be 1 for exactly one clock cycle on the posedge of state. Rather than use the posedge feature, we store a register for the previous value of bstate and set rdy to 1 only if bstate is currently 1 and was previously 0. The values sendSecret and doneSend are used in our extra credit module, described later.

The value of led1 represents if the system is locked. The toggleLED1 signal that comes out of the controller will be high for exactly 1 clock cycle, allowing it to be fed into a TFF to toggle the locked status.

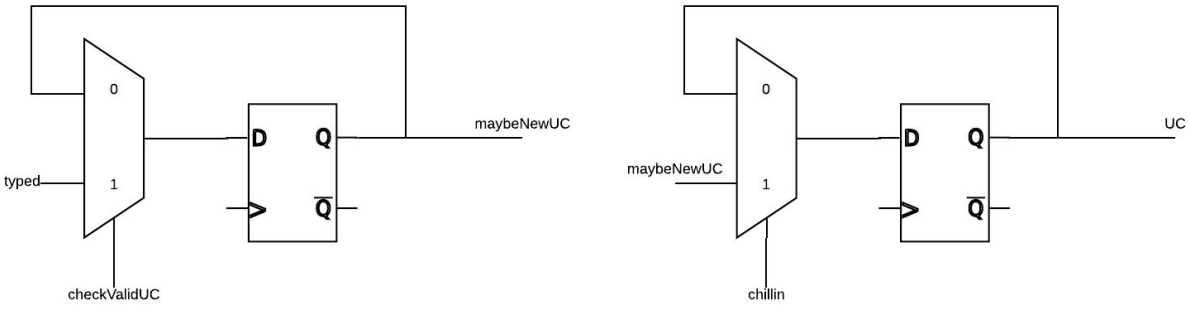


compareReg is compared to the entered value to check for matches. We set its value to the UC, PC, or maybeNewUC based on a 1-hot input based on checkUC, checkPC, and confirmUC, accordingly. The value of recordKeys is generally equal to 1 when any of checkUC, checkPC, confirmUC, or checkValidUC is equal to 1, and 0 otherwise. The exception is for a single clock cycle when the value of checkValidUC changes in either direction. In this case, recordKeys will be 0 to reset in the two intermediary steps while in the reprogramming sequence.

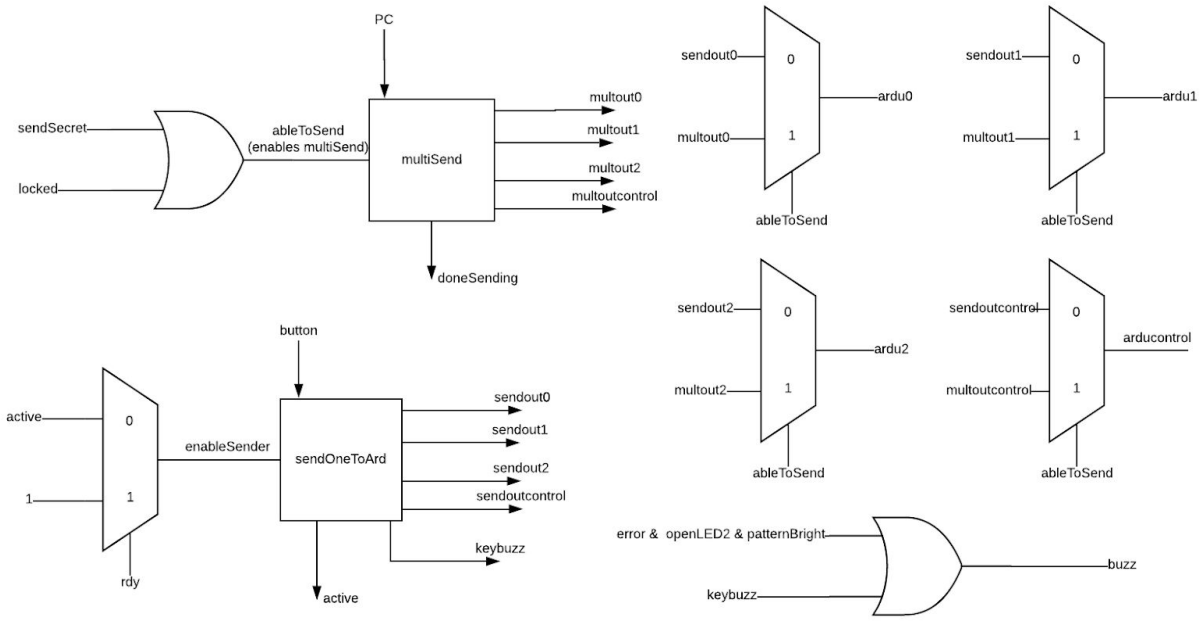


Above is the part of the top module which handles the outputs of leds 2 and 3. The controller uses the outputs of “chillin” (indicating a successful reprogramming) and “error” (which represents an error) to signify that the lightPattern should be enabled and with particular settings. The values are conversions of specified time amounts into clock cycles. The lightPattern module is enabled if either chillin or error is 1.

Led2 can only be in pattern mode with error, so the first multiplexer is controlled by the value of error. If error is 0, a solid 1 gets passed onto the next multiplexer. The next multiplexer allows the value to pass only if openLED2 is 1, otherwise 0 is sent to the value of led2. LED3, however, can be in pattern state with either chillin or error, so the first mux is controlled by the OR of these values.



Above is the final part of the top module for the main project, which assists in the reprogramming procedure. The left component sets maybeNewUC to what the user typed if the system is in the state when entering UCnew1. The component on the right handles setting UC to its new value if reprogramming is successful.



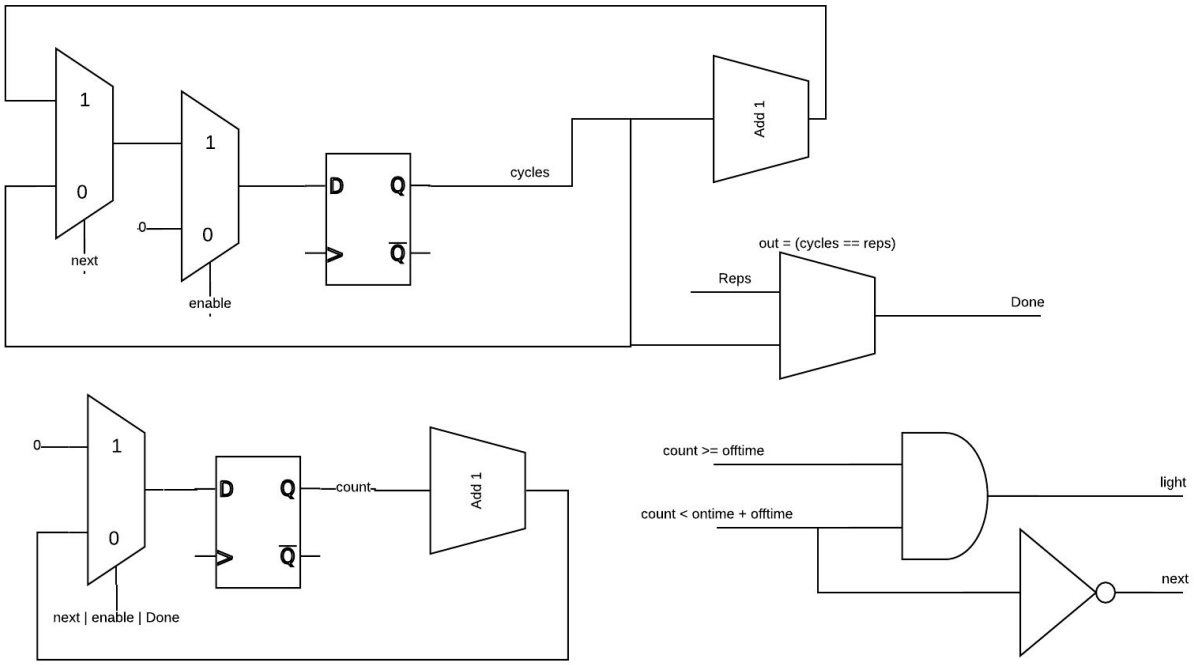
This shows the additions made to the top module to account for our extra credit communication with an arduino. The details are discussed later, but essentially this allows us to send either one digit or send the entire PC to a connected arduino and enables a buzzer for audio feedback.

Enter Digit

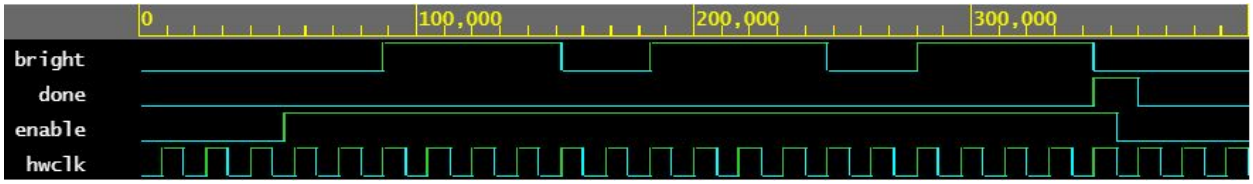
The enter Digit module is used to detect when a button is pressed and determine which button is pressed. The code for it comes directly from the Button_Code.v file that Noyan posted in the Piazza resources page. This module is used to send signals to the controller on when and how to proceed and to send information to the keylist module which takes in a series of button presses to produce a multi-digit entered value.

Light Pattern

The light pattern module is used to create adaptable blink patterns. The blink pattern is different for the error state and for the successful reprogramming state. We decided to make a general case pattern generator which takes in the ontime, offtime, number of repetitions, and an enable signal. It outputs a signal for whether the light should be on or off and a signal for if the pattern is done. Since different LEDs can use the pattern module, we handle setting LEDs outside this module.

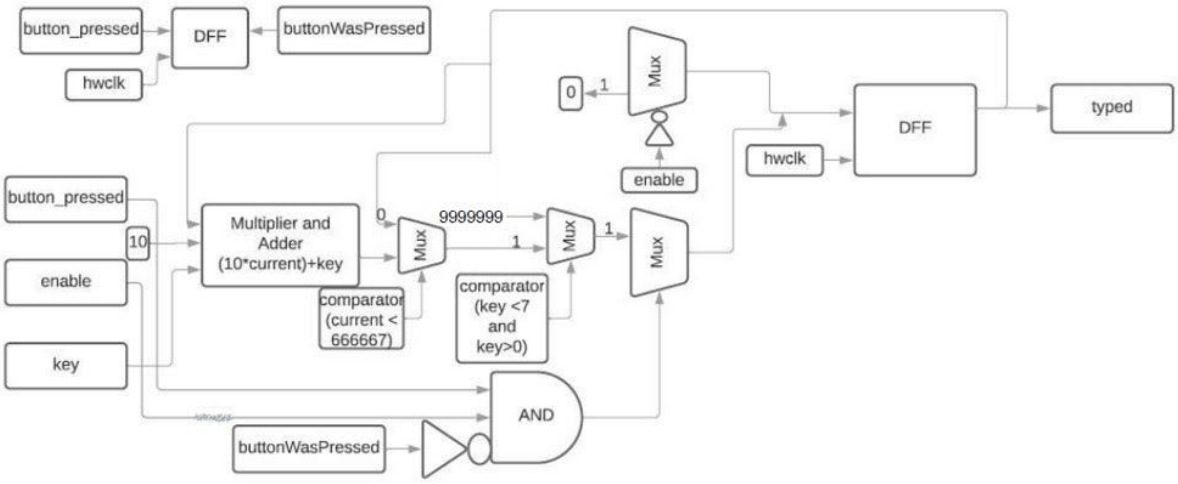


Our system keeps a running timer by counting clock cycles since the last repetition. If this value is under the offtime input value, then the light output will be 0 otherwise it will be 1. If the count value is greater than the offtime plus the ontime, then the next register is set to 1, causing the clock count to reset to 0 and incrementing the number of cycles performed. If the cycles register equals the input reps, then the Done output gets set to 1. If enable is 0, values get reset. If Done is 1, then clock count is set to 0;

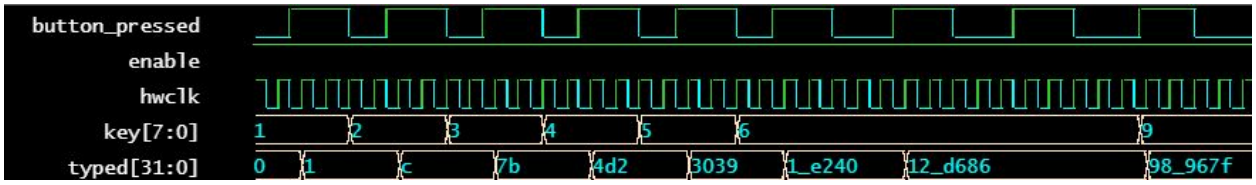


This is a timing diagram for the inputs and outputs of the pattern module. In this example, ontime, offtime, and reps were set to 3, 2, and 3 respectively for the sake of visibility.

Keylist

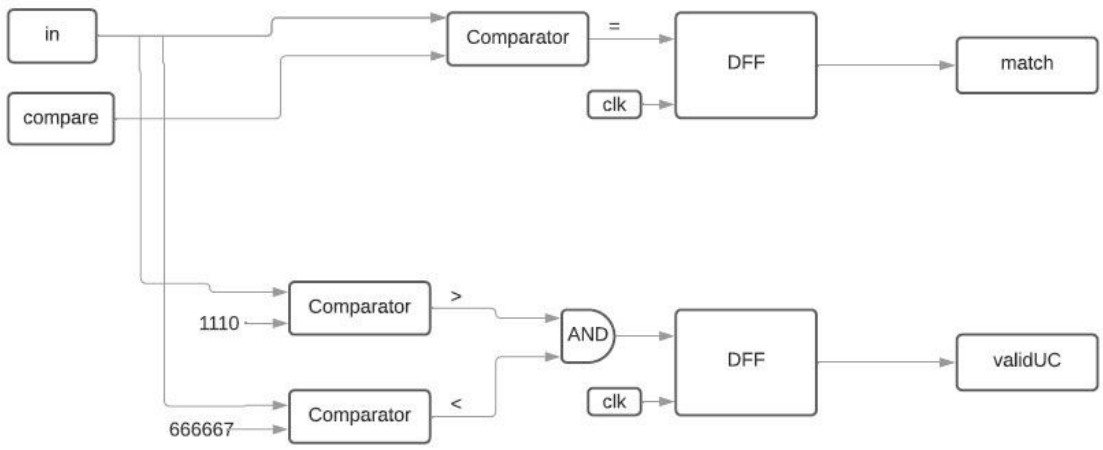


The keylist module is used to read in the digits pressed by the user and calculate the decimal value of the code entered. If the enable signal is low, then it assigns 0 to the output since this means that the system is not ready to receive inputs. As long as the button is pressed and the enable (which acts as the RDY signal) is high, and the input satisfies the requirement for a valid input (contains only digits 1-6 and has length between 4 to 6 digits), the output is assigned to the entered user code.

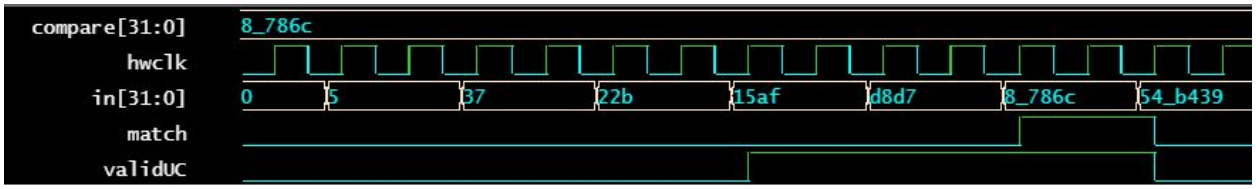


It might be kind of hard to tell, but if you convert the values into decimal you see that the value in key is added to the end of typed whenever button_pressed turns 1. Also notice how the value stops altering after 7 values have been entered (prevents overflow) and when 9 is entered the value is set to 9999999.

Compare



This module detects whether the user input for a UC is valid or not. The keylist module is built such that only keypresses from 1 to 6 are counted as valid. So, this module checks whether the inputted UC is between 1110 and 666667, since these are the lower and upper bounds on an acceptable length for a UC, and assigns validUC to 1 if the user inputted UC is between 1110 and 666667. It also checks the equality of in and compare and assigns match accordingly.



We see that the input is considered a valid UC when 4-6 decimal digits are entered, but that match is only true when it is exactly equal to the value in compare.

Extra Credit Component

For the extra credit component of the project, we had our FPGA communicate with an Arduino to display numbers on an LCD screen. It would communicate using 4 pins on each device from which the FPGA sends information and the arduino receives. These pins consist of 3 for data and 1 ready signal. This process had two parts to it: one in verilog on the FPGA and the other in C on the Arduino. Running the board without attachments causes no problems.

Verilog:

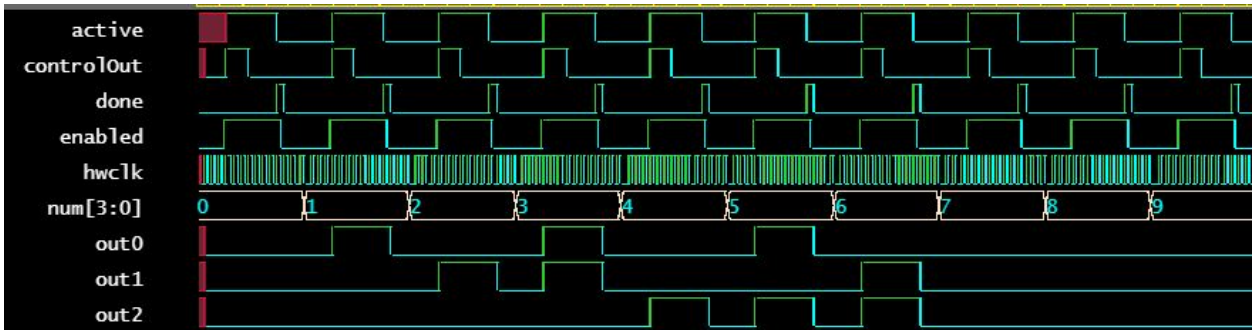
We made a “sendOneToArd” module which sends one digit to the arduino to display as the number is pressed by the user. It uses a counter for how long various outputs should be 1 and 0 and sends out a done signal. When the digit sent is 0, the arduino clears the LCD screen. To have it clear regularly, 0 is sent whenever the user presses 7, 8, or 9.

The other module built in verilog was one that deals with displaying a “secret” (the PC) when the system is unlocked and the user presses 7. We use known properties of the PC, such as that it is exactly 6 digits ranging from 1-6. The multiSend module, as it is called, isolates each digit in the PC and calls its own “sendOneToArd” module to output them all in rapid fashion. The sendOneToArd module takes 0.2 seconds to send each digit, so the entire PC transfers in 1.2 seconds. This could probably be improved

through the use of interrupts in the arduino, but the size of our messages is so small that it does not matter very much.

We also added a buzzer portion of extra credit. We had the buzzer play a different note based on which button is pressed by the user and we also made it play an angry noise when someone tries to unlock the device with the wrong UC. We had to add another output to the FPGA and the top module for this.

We added 5 output pins on the pinmap to handle the outputs. The value of these is generally controlled by the top module's sendOneToArd module, but if the user presses 7 in the idle state while the system is unlocked, the multisend is allowed to take over and print out the PC. To our controller, we had to add another state, input, and output to handle the extra credit portion.



Above is a timing diagram for the sendOneToArd module when numbers 0 through 9 are sent in one after the other. Notice how for numbers 0, 7, 8, and 9 out0, out1, and out2 are all 0. This is to signal the arduino to clear the lcd screen whenever these buttons are pressed. The controlOut is high for half of the total time, so the arduino could read numbers sent contiguously. The holdTime was small for readability.

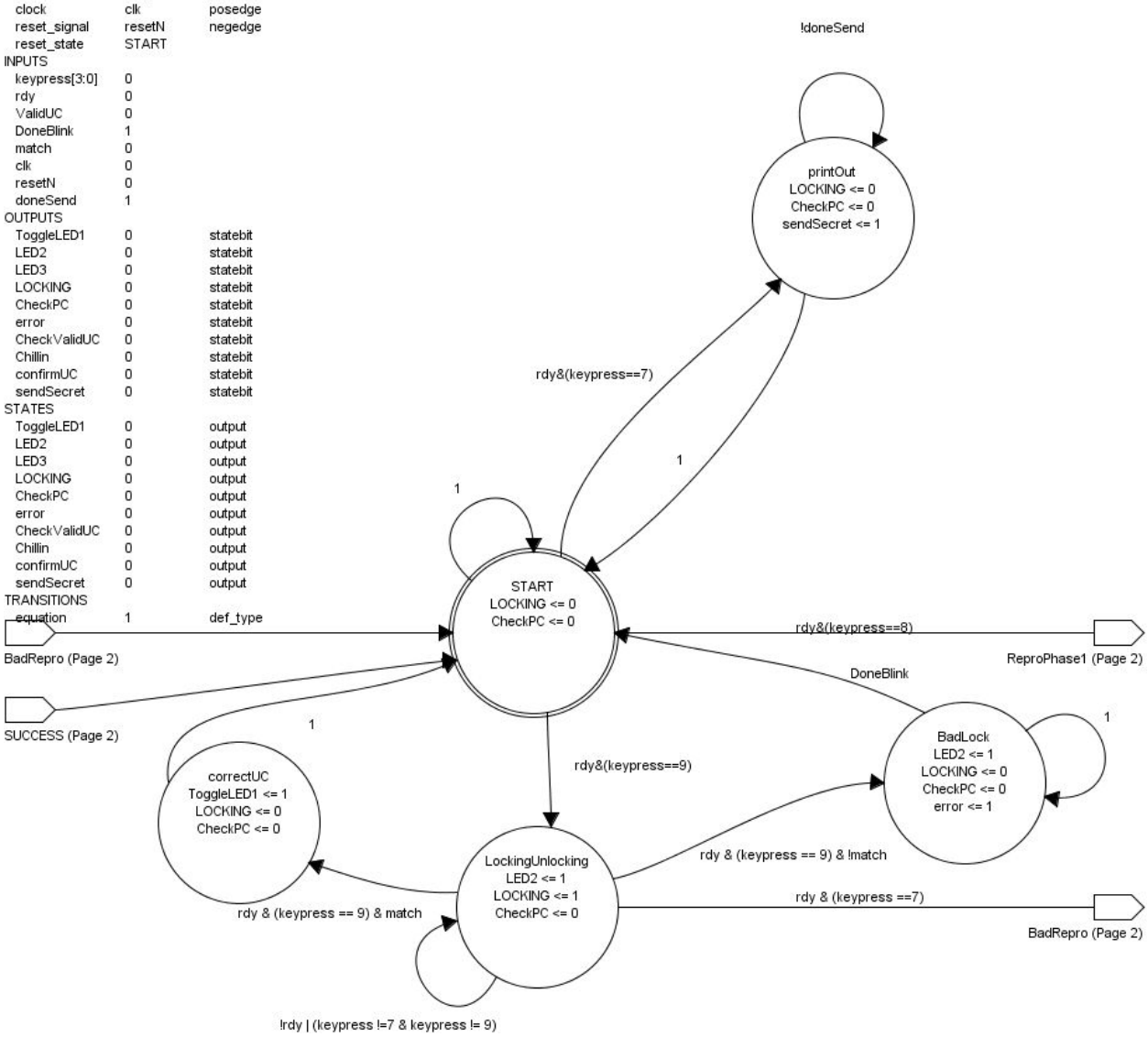
Arduino:

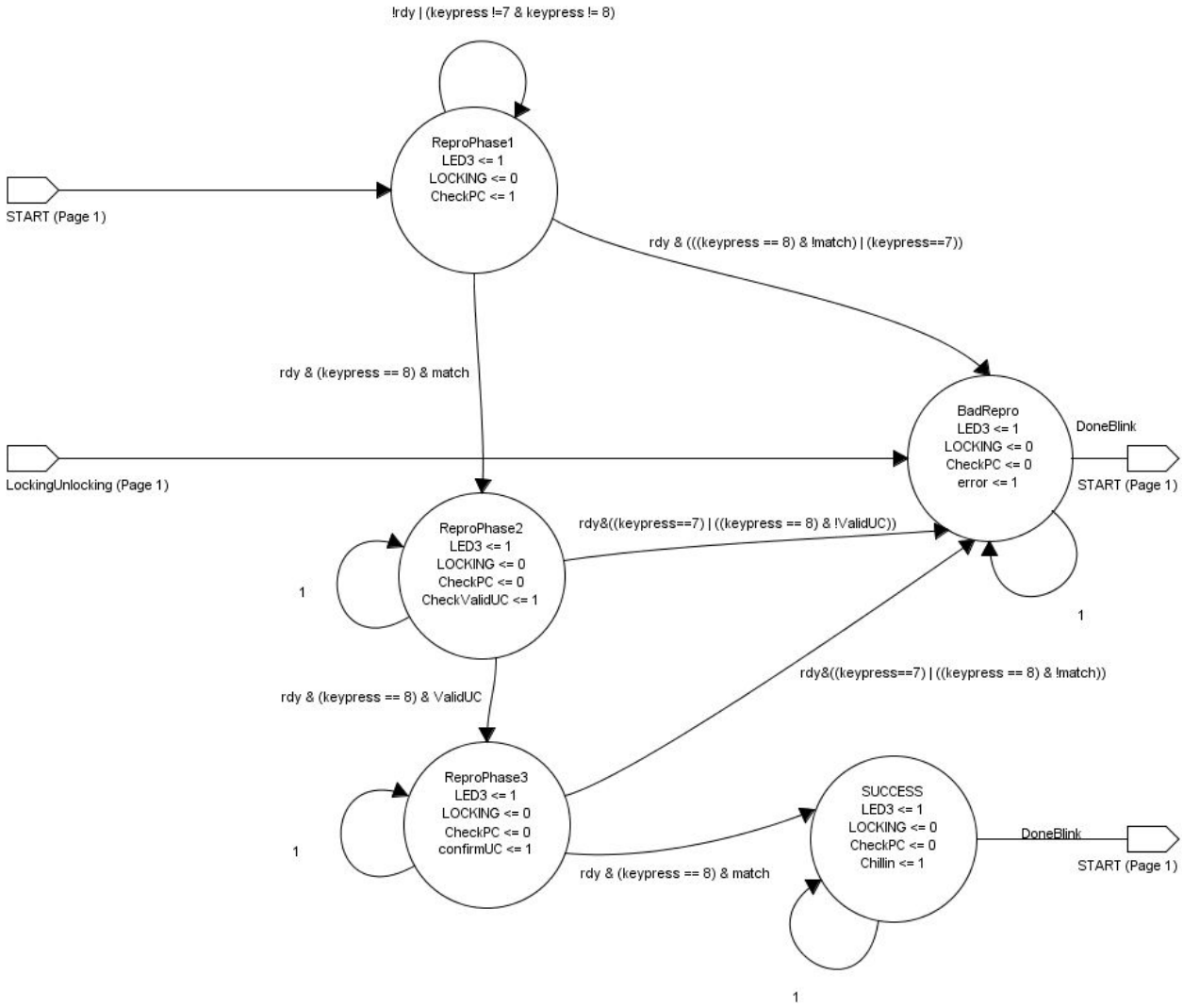
The analog pins read data from the FPGA board and the digital pins print onto the LCD display screen. Because the FPGA operates on 3.3V and the Arduino on 5V, we manually set an analog threshold to read in as HIGH. We used 3 analog pins for the binary representation of the digit being typed, and 1 for the RDY signal. Once the RDY is high, the Arduino program reads in the value on the other 3, and calculates the decimal value of the entered digit. Then, the value is displayed on the LCD screen. Once done printing, the program moved the cursor to the side, so that subsequently entered digits get printed adjacent to the previously displayed digit instead of overwriting it. The LCD is cleared if RDY is HIGH and all other pins are LOW. Once RDY goes HIGH, we need to wait for it to go back down to low before considering more inputs. Essentially we only act on the posedge of RDY.

We followed this guide on setting up and using the LCD https://www.arduino.cc/en/Tutorial/HelloWorld

State Diagrams of Controller

STATE MACHINE		
name	def_name	
clock	clk	posedge
reset_signal	resetN	negedge
reset_state	START	
INPUTS		
keypress[3:0]	0	
rdy	0	
ValidUC	0	
DoneBlink	1	
match	0	
clk	0	
resetN	0	
doneSend	1	
OUTPUTS		
ToggleLED1	0	statebit
LED2	0	statebit
LED3	0	statebit
LOCKING	0	statebit
CheckPC	0	statebit
error	0	statebit
CheckValidUC	0	statebit
Chillin	0	statebit
confirmUC	0	statebit
sendSecret	0	statebit
STATES		
ToggleLED1	0	output
LED2	0	output
LED3	0	output
LOCKING	0	output
CheckPC	0	output
error	0	output
CheckValidUC	0	output
Chillin	0	output
confirmUC	0	output
sendSecret	0	output
TRANSITIONS		
equation	1	def_type





The first of these two diagrams contains the procedure for locking and unlocking as well as the start state. The second diagram handles the procedure for reprogramming. The controller has many inputs and outputs, which are used to communicate with various modules as described in the **Top** section. Starting in the START state, the system waits for signal from the keypad, which it receives as the rdy and keypress signals. Signals such as match are adapted and reused in multiple locations, all with similar purpose. Light handling can be seen via the BadRepro state, which sets error to 1 and openLED3 to 1. This means that the error blink should occur on LED3. While in states that take a certain amount of time to process (errors, success, and printOut), the controller waits until signalled to proceed.

How to Change Programmer Code (PC)

Near the top of the file top.v (currently line 17), there is a 32-bit parameter called PC. To change the PC, simply modify this value to the new value. The code does not check for validity in the PC because there is no way of modifying the PC without altering the source code.

Ways to make our project better

-Do all LED control inside the pattern module and have multiple pattern modules inside top.
-Revise the names of registers in the code and FSM to improve standardization.
-Compress the various “check input” registers to reduce total outputs of the FSM.
-Incorporate the compare module into the keylist module.

-The lightPattern module can be improved in its function while “disabled” in the verilog. The diagram in this report follows the general logic of the verilog but is not a perfect representation.
-Have multiple comparators in the top module to eliminate some of the multiplexers for compareReg.

Important Reflection
Debugging is hard in verilog since it doesn’t tell you many syntax errors so we had to use LEDs that changed state on certain conditions. We would do this for like an hour and then discover that it was just a spelling/capitalization mistake. This happened more than we would like to admit. Plus it takes like 30 seconds to compile and send the code so each iteration takes a while. This was a little rant to basically say that we really wish that our naming conventions were better.

Division of Labor
Sahil worked on the arduino code for the EC, and on the keylist and compare modules. Blake worked on the top module, light pattern module, and verilog code for the EC. The controller FSM was designed by both partners. Design choices were made collaboratively as was some testing and debugging. In the report, both partners wrote about and made architecture diagrams for the components they specialized in.