Group-20
Signature Design Final Report
CS281R

Team Members
Jonathan Benson
Stephen Holman
Jens Kenneth

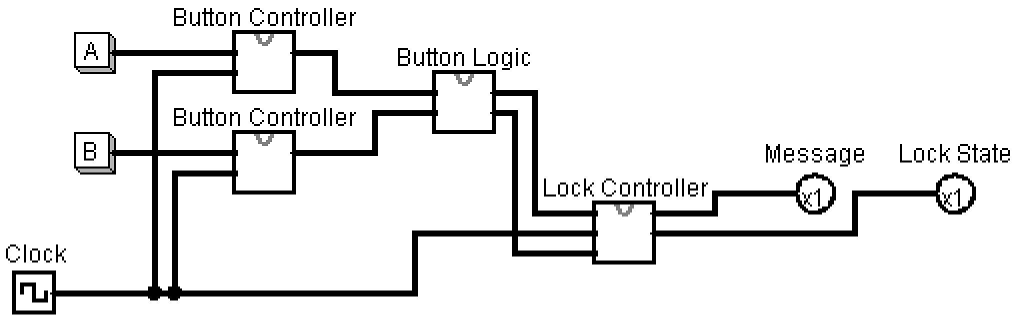
Introduction

The following document serves as our final report for the design of a digital lock system that recognizes the pattern "BAABBA". Contained in the document are truth tables, state diagrams, component analyses, and discussions on social and ethical issues. We have also prepared a fully-functional simulation of our digital lock in Logisim entitled "lock.circ" in the project submission folder. How-to simulation instructions and report references will be at the end.

Bird's Eye View

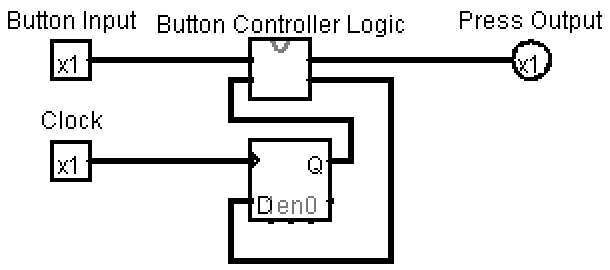
From a bird's eye view, the system consists mainly of two button controllers, some button logic, and a central lock controller. The button controllers and button logic block ensure that the lock controller only loads its state register on single, valid button presses. In other words, it will ignore when two buttons are pressed down at the same time or when there is inactivity. If a button press is recognized out of sequence, the lock controller will relay an alarm message. On the other hand, if the valid sequence of button presses is encountered, the lock controller will unlock the lock. In all other cases, the lock will remain locked and there will be no alarm message. When the lock is finally unlocked, the system will wait for another button press to take the system back to its initial state.

Component Diagram of Digital Lock System



Button Controller

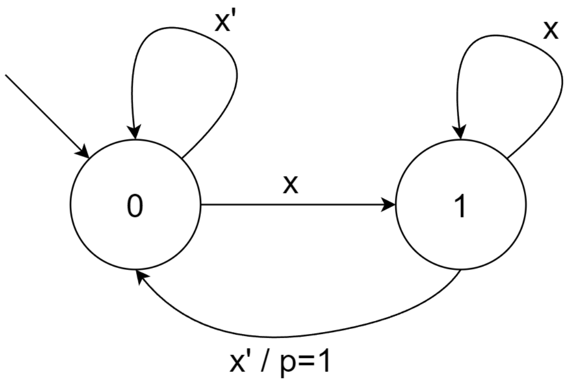
The purpose of the button controller is to provide button-press pulses only when the valid button-activation sequence is recognized. In other words, the button controller will output 1 only if the button's state sequence is deactivated, activated, and then deactivated. In all other cases, it will output 0. This sequence ensures that only valid button presses are sent as input to the lock controller. A higher clock frequency is ideal to allow for faster, smoother button presses. Since the buttons only have two states (activated and deactivated), only one D-Flip-Flop is required to keep track of the state.



Button Controller State Diagram

Inputs = {x}
x' = 0 -> button inactive (up)
x = 1 -> button active (pressed down)

Outputs = {p}
p = 0 -> no button press (implicit)
p = 1 -> valid button press



In the following truth table, the input variables p and x represent the current state and button activation input respectively. The output variable y represents the button press output, while n represents the next state.

Button Controller Logic Truth Table

p	x	y	n
0	0	0	0
0	1	0	1
1	0	1	0
1	1	0	1

From the button controller logic's truth table, we derived the following equivalent boolean expressions for each output variable.

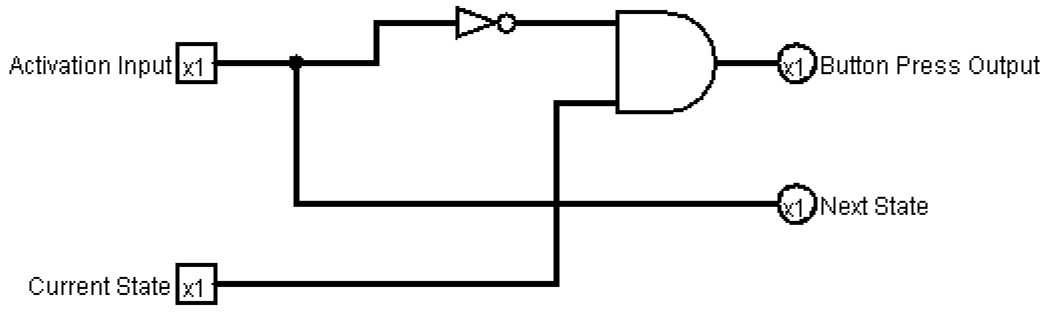
Button Controller Logic Expressions

$$y = px'$$

$$n = x$$

With the above boolean functions, we designed its equivalent logic circuit to be used in the simulation. We determined that the gate delay for this circuit is 1 assuming NOT gates are not counted.

Button Controller Logic Circuit



Button Logic

The purpose of the button logic block is to ensure that a signal is only registered by the lock controller upon a single, valid button press via an enable output. It also acts like a 2x1 encoder in that it outputs a 0 or 1 depending on which button is pressed. On the other hand, it utilizes don't cares in the invalid cases that both buttons are pressed or inactive because its enable output will prevent the lock controller from registering a button press.

In the following truth table, variables a and b represent the inputs of button A and B respectively. The output variable p represents the button output (A=0, B=1), while the variable e is an enable for the lock controller.

Button Logic Truth Table

a	b	p	e
0	0	x	0
0	1	1	1
1	0	0	1
1	1	x	0

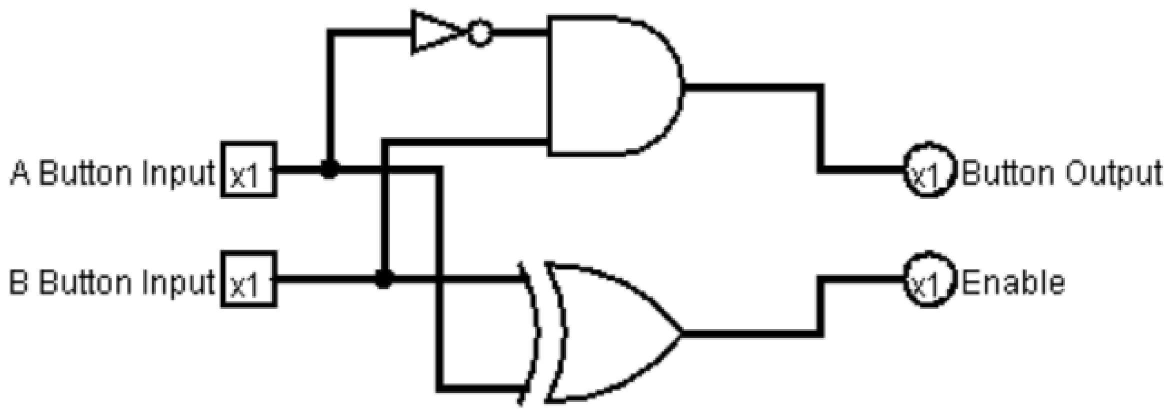
From the button logic's truth table, we derived the following boolean expressions for each output variable.

Button Logic Boolean Expressions

$$p = a'b$$
$$e = a'b + ab'$$

We noticed that the expression for the e variable matched that of an XOR gate and so we incorporated that into the design of the logic circuit to lower the gate delay of the circuit.

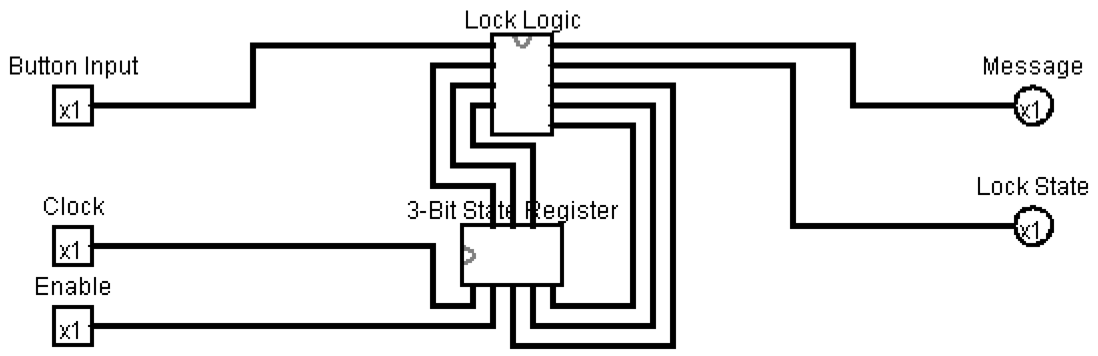
Button Logic Circuit



Assuming NOT gates are not taken into consideration with regards to gate delay, we determined that the gate delay of the button logic circuit was 1.

<u>Lock Controller</u>

The lock controller keeps track of state with a 3-bit register made with 3 D-Flip-Flops. Every valid button press, the lock controller's logic block will output an alarm message (or no message at all), the lock state (locked or unlocked), and the next state. The next state is then immediately loaded into the 3-bit state register in preparation for the next valid button press. On invalid button presses or inactivity, loading to the state register is disabled provided the button logic's enable output.

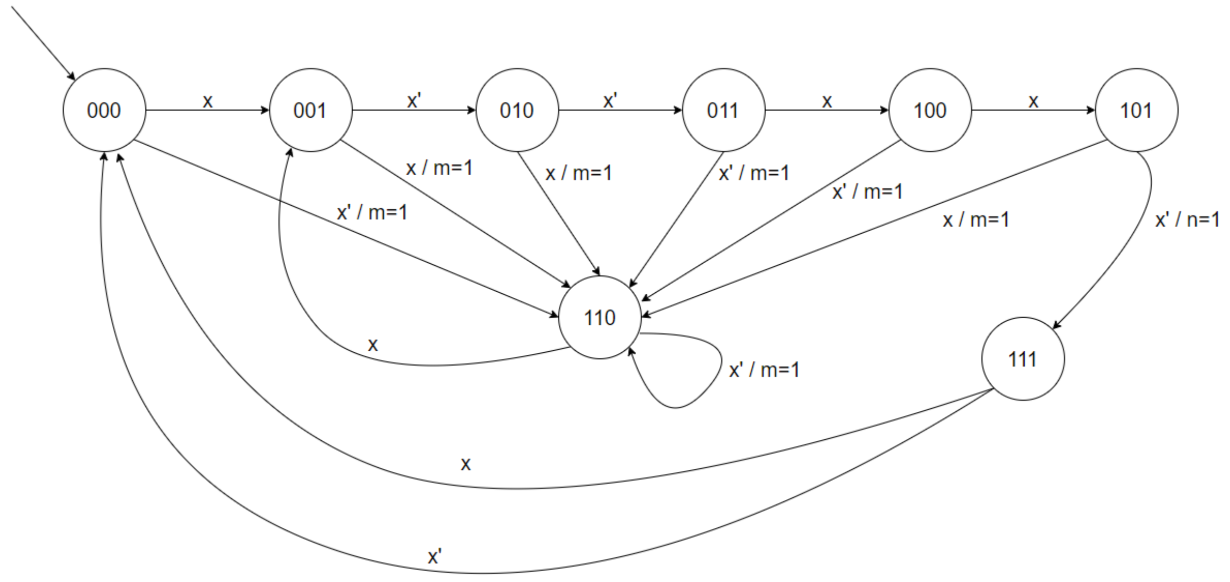


.

Lock Controller State Diagram

Inputs = {x}
x' = 0 -> "A" button press
x = 1 -> "B" button press

Outputs = {m, n}
m = 0 -> no message (implicit)
m = 1 -> signal alert message
n = 0 -> locked (implicit)
n = 1 -> unlocked



In the following truth table, the variables p2, p1, and p0 represent the current state, while the variables n2, n1, and n0 represent the next state of the lock controller. The variable x represents which button has been pressed (B=0, A=1). For the outputs, m is the alarm message (silent = 0, alarm = 1) and n is the lock state (locked = 0, unlocked = 1).

Lock Controller Truth Table

	p2	p1	p0	x	m	n	n2	n1	n0
m0	0	0	0	0	0	0	1	1	0
m1	0	0	0	1	0	0	0	0	1
m2	0	0	1	0	0	0	0	1	0
m3	0	0	1	1	0	0	1	1	0
m4	0	1	0	0	0	0	0	1	1
m5	0	1	0	1	0	0	1	1	0
m6	0	1	1	0	0	0	1	1	0
m7	0	1	1	1	0	0	1	0	0
m8	1	0	0	0	0	0	1	1	0
m9	1	0	0	1	0	0	1	0	1
m10	1	0	1	0	0	0	1	1	1
m11	1	0	1	1	0	0	1	1	0
m12	1	1	0	0	1	0	1	1	0
m13	1	1	0	1	1	0	0	0	1
m14	1	1	1	0	0	1	0	0	0
m15	1	1	1	1	0	1	0	0	0

With the truth table formulated from the lock controller's logic block, we constructed another truth table to conveniently show the flow of states in the state diagram. This table made it much easier to create an implication table that we used to fully reduce the number of states in our lock controller.

State Table for Lock Controller

Present State	Next State, x=0	Next State, x=1	Present Output
A (000)	G	B	00
B (001)	C	G	00
C (010)	D	G	00
D (011)	G	E	00
E (100)	G	F	00
F (101)	H	G	00
G (110)	G	B	10
H (111)	A	A	01

Interestingly, the states that seemed to be the most likely to be reduced were along the top chain in the state diagram. Unfortunately, however, after completing our implication table, no states could be reduced. There ended up being three rounds of non-equivalent state-elimination until no reducible states were left. The first round only consisted of the states that could not be equivalent due to their different outputs (represented in dark red). Then the second (mild red) and third round (light red) states were found to be not equivalent because they contained state-pairs that were discarded from a previous round.

Implication Table for Lock Controller

B	G-C, B-G						
C	G-D, B-G	C-D					
D	B-E	C-G, G-E	D-G, G-E				
E	B-F	C-G, G-F	D-G, G-F	E-F			
F	G-H, B-G	C-H	D-H	G-H, E-G	G-H, F-G		
G	X	X	X	X	X	X	
H	X	X	X	X	X	X	X
	A	B	C	D	E	F	G

Gate Minimization of Lock Controller's Combinational Logic with Karnaugh Maps

After obtaining a minimized state diagram for our lock controller, we continued on to design its combinational logic. We constructed a 4-variable Karnaugh map (using minterms) for each output value and derived for each of them a simplified boolean function.

m = (m12, m13)
m = p2p1p0'

p2, p1 \ p0, x	00	01	11	10
00				
01				
11	1	1		
10				

	p2	p1	p0	x	
m12	1	1	0	0	-> p2p1p0'
m13	1	1	0	1	

n = (m15, m14)
n = p2p1p0

p2, p1 \ p0, x	00	01	11	10
00				
01				
11			1	1
10				

	p2	p1	p0	x	
m15	1	1	1	1	-> p2p1p0
m14	1	1	1	0	

n0 = (m1, m9) + (m13, m9) + (m4) + (m10)
n0 = p1'p0'x + p2p0'x + p2'p1p0'x' + p2p1'p0x'

p2, p1 \ p0, x	00	01	11	10
00		1		
01	1			
11		1		
10		1		1

	p2	p1	p0	x	
m1	0	0	0	1	-> p1'p0'x
m9	1	0	0	1	
m13	1	1	0	1	-> p2p0'x
m9	1	0	0	1	
m4	0	1	0	0	-> p2'p1p0'x'
m10	1	0	1	0	-> p2p1'p0x'

n1 = (m0, m4, m12, m8) + (m0, m2, m4, m6) + (m2, m3, m10, m11) + (m4, m5)
n1 = p0'x' + p2'x' + p1'p0 + p2'p1p0'

p2, p1 \ p0, x	00	01	11	10
00	1		1	1
01	1	1		1
11	1			
10	1		1	1

	p2	p1	p0	x	
m0	0	0	0	0	-> p0'x'
m4	0	1	0	0	
m12	1	1	0	0	
m8	1	0	0	0	
m0	0	0	0	0	-> p2'x'
m2	0	0	1	0	
m4	0	1	0	0	
m6	0	1	1	0	
m2	0	0	1	0	-> p1'p0
m3	0	0	1	1	
m10	1	0	1	0	
m11	1	0	1	1	
m4	0	1	0	0	-> p2'p1p0'
m5	0	1	0	1	

n2 = (m8, m9, m11, m10) + (m0, m8) + (m12, m8) + (m3, m7) + (m5, m7) + (m6, m7)
n2 = p2p1' + p1'p0'x' + p2p0'x' + p2'p0x + p2'p1x + p2'p1p0

p2, p1 \ p0, x	00	01	11	10
00	1		1	
01		1	1	1
11	1			
10	1	1	1	1

	p2	p1	p0	x	
m8	1	0	0	0	-> p2p1'
m9	1	0	0	1	
m11	1	0	1	1	
m10	1	0	1	0	
m0	0	0	0	0	-> p1'p0'x'
m8	1	0	0	0	
m12	1	1	0	0	-> p2p0'x'
m8	1	0	0	0	
m3	0	0	1	1	-> p2'p0x
m7	0	1	1	1	
m5	0	1	0	1	-> p2'p1x
m7	0	1	1	1	
m6	0	1	1	0	-> p2'p1p0
m7	0	1	1	1	

It turned out that a sum of products provided the least number of gates for each output value except the next state's most significant bit (n2). Interestingly, deriving the simplified product of sums for n2 provided a combinational circuit with one less gate than it's sum of products version.

OPTIMIZED n2 Karnaugh Map using product of sums

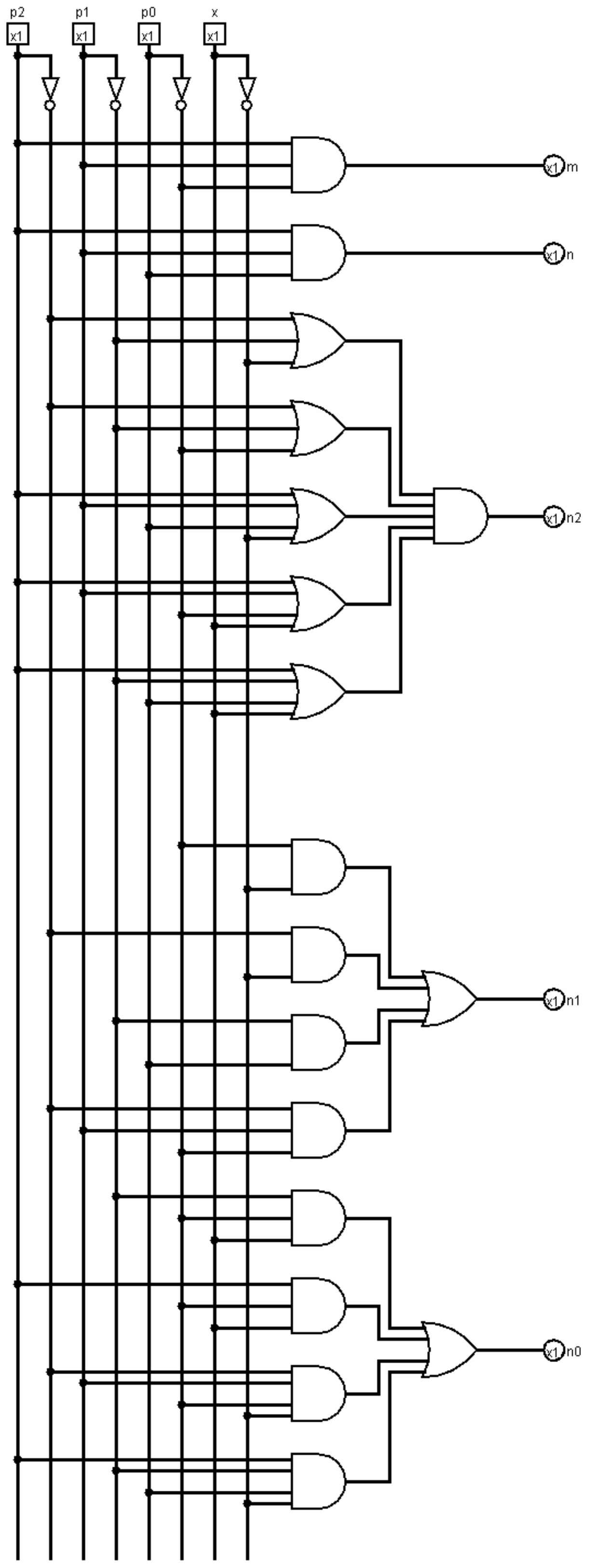
n2 = (M4) (M1) (M13, M15) (M15, M14) (M2)
n2 = (p2' + p1 + p0' + x') (p2' + p1' + p0' + x) (p2 + p1 + x) (p2 + p1 + p0) (p2' + p1' + p0 + x')

p2, p1 \ p0, x	00	01	11	10
00		0		0
01	0			
11		0	0	0
10				

	p2	p1	p0	x	
M4	0	1	0	0	-> p2' + p1 + p0' + x'
M1	0	0	0	1	-> p2' + p1' + p0' + x
M13	1	1	0	1	-> p2 + p1 + x
M15	1	1	1	1	
M15	1	1	1	1	-> p2 + p1 + p0
M14	1	1	1	0	
M2	0	0	1	0	-> p2' + p1' + p0 + x'

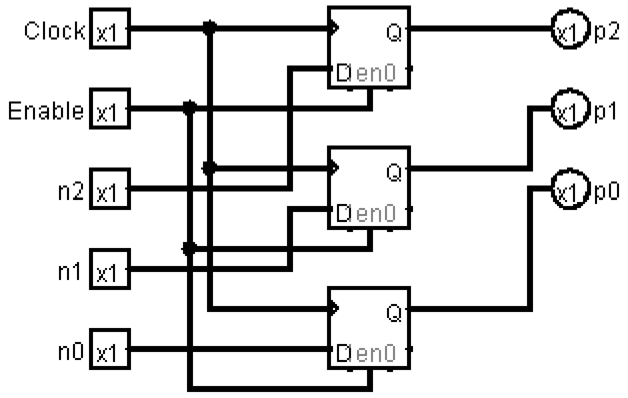
With all of the simplified boolean expressions for each output, we were able to design the logic circuit for the lock controller.

Lock Controller Logic Circuit



We determined that the gate delay for the lock controller circuit is 2, again assuming that NOT gates do not factor into the gate delay.

3-bit Lock Controller State Register



We implemented the lock controller's 3-bit state register with 3 D-flip-flops. It's job is to keep track of the state of the lock controller. The lock controller updates its state on rising clock edges, but will not register a new state if the enable (supplied by the button logic circuit) is off. We also determined the gate delay of the state register to be 4 because each D-flip-flop contains 2 D-latches that each contain a gate delay of 2.

<u>Optimal Clock Frequency of Digital Lock</u>

Table of Components and their Gate Delays

Component	Gate Delay
Button Controller	1
"^" State Register	4
Button Logic	1
Lock Controller	2
"^" State Register	4
Total	12

We computed a total gate delay of 12 by adding up all the gate delays from each component. Assuming a propagation delay of 1 nanosecond for each gate, we calculated the system's critical path to be 12 nanoseconds. This critical path gives an absolute maximum clock frequency of about 80 megahertz. In practical design, however, we believe that a clock frequency of about 50 megahertz would be ideal. Our system is not high-speed dependent, but higher speeds do allow for a smoother user experience. Unfortunately, the highest clock frequency Logisim allows is 4.1 kilohertz, but we think that is good enough for testing.

Social and Ethical Issues

In order to provide a thorough social and ethical analysis of our digital lock system, we utilized the principles contained in the ACM Code of Ethics [1]. There are a wide range of ethical principles that are applicable to the design of our digital lock, but those most notable are competency evaluation, professional review, and meaningful work.

According to Section 2.6 of the Code, professionals are required to perform work only in their respective areas of competence. We adhered to this principle by dividing up the work of the project based on each team member's areas of competence. For example, I have strong written-communication skills and professional leadership experience. My primary roles were to lead project development, organize and divide work between team members, manage the project's Discord server and Google Drive folder, and prepare the final submission for all reports. Jens, on the other hand, is a strong conceptual thinker. Jens' primary roles were to design all of the state machines, truth tables, and implication tables. Stephen is great at interpreting and editing code but has poor creativity. Therefore, Stephen's role was to design the Logisim circuit and test its functionality.

The Code also states that professionals are required to provide each other with constructive criticism according to Section 2.4. Early on in our design, we realized our lock's state diagram was not producing the desired functionality when its circuit was implemented in Logisim. This required us to completely redesign our lock's state diagram from the ground up in order to correct its functionality. Without constructive criticism between team members, we would have been left with a faulty state diagram which would inevitably lead to the design of a broken lock system that could potentially harm the end user.

Additionally, the ACM Code of Ethics requires professionals to respect the amount of work required to foster the safe, efficient, and robust design of systems according to Section 2.9. At first, the design of a sequential lock seemed trivial. In fact, we had a fully functional design in Logisim very early on in the process. It was not until we performed extensive design analysis and wrote thorough documentation that we realized producing a quality product entails a tremendous amount of work and collaboration. For this reason, we learned that even the simplest designs demand thorough examination and testing before they are turned into an end-product.

Instructions on How to Run the Lock Simulation

Downloading Logisim

Our digital simulation was designed in Logisim, so interacting with the simulation will require the Logisim program. We will also assume the use of the Windows operating system to run the program. Logisim can be downloaded for Windows at the following link:

https://sourceforge.net/projects/circuit/

Opening the simulation

Upon downloading the executable file, drag it to an acceptable folder on the destination computer where it can be found. To open the simulation, simply drag the simulation file entitled “lock.circ” in the project folder right over to the Logisim executable file. If all goes well, the simulation should open without any errors.

Interacting with the simulation

With the “lock.circ” file open in Logisim, double click the “Lock” circuit over on the left-hand side of the screen to make sure the buttons can be pressed. Then navigate to the “Simulate” tab at the top of the screen and make sure “Simulation Enabled” is checked. Before running the simulation, reset the simulation and set the “Tick Frequency” to 4.1 kHz all within the “Simulate” drop-down menu. Finally, check the “Ticks Enabled” in the “Simulation” menu and the simulation will begin to run. To test the simulations, simply press the buttons and watch the Message and Lock State outputs change as the lock system keeps track of the button sequence.

References

[1] ACM Code of Ethics and Professional Conduct. Association for Computing Machinery, 7 June 2016, ethics.acm.org/code-of-ethics/. Accessed 5 Dec. 2021.