Kahan Patel's Technical Memo

To: Dr. Christopher Peters, College of Engineering, Drexel University

From: Kahan Patel

Date: 3/16/2024

Re: Adders & Subtractors (Chapters 25-26)

Summary

In this final laboratory the concepts of binary addition and subtraction were explored. Furthermore, the subtraction method applied was the concept of 2's complement. This laboratory was Verilog only, meaning it was completed in a two step approach. The first half of the laboratory consisted of designing the half and full-adder (1 bit). Once the adders were designed, they were "daisy-chained" together to create a multi-bit adder (ripple-adder). Once the design was created, it was simulated in Verilog, where different conditions were tested to see if the circuit behaved correctly.

Purpose

The laboratory delved into the fundamental concept of binary addition and subtraction, illustrating their importance in digital logic. These calculations are intricately intertwined with modern devices, emphasizing the critical need for a deep comprehension to effectively navigate digital logic design principles. It became evident upon the laboratory's completion that, contrary to common belief, logical systems do not execute subtraction directly. Instead, the subtrahend undergoes a process of being complemented twice before addition occurs. This key realization shows the method behind binary subtraction (2's complement). In essence, this laboratory

sheds light on the intricate workings of basic to logic adders and subtractors, unraveling the mechanisms behind fundamental binary calculation operations.

Methodology/Approach Section

The way the first part of this laboratory was tackled was by understanding the logic of binary addition and subtraction. When adding two binary numbers, the sum of the binary numbers must allocate the same amount of memory as the largest added number. If the sum is greater, it leads to an overflow. This overflow needs to be handled, to ensure the sum is properly read. By first understanding the different cases involved in binary addition, then a circuit could be designed to execute the proper behavior. When assessing binary subtraction, other problems arise such as determining if a number is positive or negative after and before the subtraction. In this case, the concept of signed magnitude must be utilized to determine if a number is going to be positive or negative. Once the concept of how the mathematical operations will be implemented, a circuit was designed to represent the behavior. The first thing that was considered when designing the circuit were the inputs and outputs. When performing both addition or subtraction between two numbers there are always two numbers of n length, these will be represented as A and B. Moreover, there are always two outputs, the binary sum and the overflow/COUT, lastly a CIN/SUB to determine the mathematical operation. Now, the only concept to implement is 2's complement. However, this will only apply during subtraction.

After all variables have been properly assessed, the basic 1 bit full adder was constructed. The way the full adder was created was by checking all possible conditions of A, B, and CIN. After doing so, it leads to the derivation of four boolean equations that completely describe the behavior of a 1-bit adder. Each equation is then implemented in Verilog with the proper variable name to simulate a behavior description of a 1-bit adder. Since the operation of addition is done in a cyclical fashion, multiple adders can be chained together to drive the inputs and outputs of each module. This allows for an n amount of adders to be used to simulate a theoretically infinitely long binary addition. However, for the purposes of this laboratory, four 1-bit adders will drive one another to create a "4-bit ripple carry adder." This was done by creating a new module, defining the intermediary connections between the four adders, and instantiating four 1-bit adders inside the new module.

The final part of this laboratory involves creating a single circuit that can simulate both addition and subtraction operations. The fundamental concepts used to allow for this behavior were the usage of toggling CIN (0, 1) and XOR gates for the subtrahend. First, by toggling CIN as either 0 or 1, it simulates behavior of signed binary numbers, and the toggled values are imputed into an XOR gate which determines if the number should be complemented or not. By including these simple additions, the circuit can now handle both subtraction and addition operations. To test the circuit's ability to add and subtract different number combinations, 10 different 4-bit binary numbers were added and subtracted. The results were checked by hand and noted. Final results can be found in the appendix below as well as the complete adder/subtractor circuit.

Results

After creating the final circuit that can add and subtract in Verilog and checking the results to see if the outputs were correct, the circuit was able to accurately give the sum and difference of the 10 different combinations with their respective COUT/OVRF. By passing these relatively randomly selected inputs, it proves the circuit's ability to perform addition and subtraction arithmetic. An important note to mention is that this design can still be improved upon by incorporating a way to determine which of the two inputs (A, B) is larger. By including this additional circuit it would allow for a negative number to be entered first and the result would still be correct after applying 2's complement. Another modification that can be made is storing sum/difference in a register so it can be used again or in a different circuit. By doing this, it would build off the basic 4-bit ripple-carry adder and allow for it to have functionality in more complex circuits.

Overall, after the completion of this laboratory, the concepts of binary addition and subtraction using 2's complement were able to be executed in digital logic to create a full adder. By understanding the basics of adders and subtractors, binary arithmetic can now be applied in future circuits/projects to create more complex and applicable designs.

Kahan Patel's Technical Memo

To: Dr. Christopher Peters, College of Engineering, Drexel University

From: Kahan Patel

Date: 3/08/2024

Re: Finite State Machines (Chapters 21-24)

Summary

In this laboratory, the concept of finite state machines was explored, particularly the Moore Finite State Machine and the detection of "101" binary sequences. The lab was tackled in a three-part sequence, initially, the state transition diagram was made and the electrical drawing was eventually constructed. Based on the drawing of the circuit, it was then tested in Verilog to observe the behavior and see if the proposed equations were correct. Lastly, after completing both parts, the physical circuit was created and the results were compared between each part to verify the validity of the results.

Purpose

Finite state machines fall under the study of finite automata. Finite automata allow for the creation of pattern recognition machines that allow for the simulation of abstract conditions with a nth amount of states. Each state represents a possible condition the finite machine can be in. The reason why this topic was thoroughly studied was because of finite automata's ability to help simulate input and output detection based on a given input sequence. Since finite state machines can be used to detect this behavior, they are essential when creating circuits that solve sequence recognition problems. Moreover, finite state machines' graph-like structure allows for the creation of infinitely complex conditions with a finite nth amount of states. This property allows for the simulation of real-world tasks with a set amount of conditions. So,

cumulatively, the modularity and versatility of FSM make them an important topic to further analyze.

Methodology/Approach Section

The first part of this laboratory involved creating a theoretical solution to the 101 detection circuit with overlap. The way this was accomplished was by first defining each possible state. By doing this first, the overall complexity of the machine can be assessed, and the next steps that follow. For this particular FSM (Moore machine) there were 4 defined states. Each state was denoted with S_n, with n being the state number. Next, the logical traversal of each state was carefully drawn in a state transition diagram. This part is vital to the construction of the FSM, as everything following depends on the drawing. Once the drawing was complete, a state table was made to map the behavior of the graph. To allow for the creation of a boolean equation each state was mapped with an arbitrary binary number. The way this was determined was by using the basic combinatorics equation $2^n = 4$, where n is the number of states. In this case, n = 2, meaning each state will be represented by a 2-bit binary number ranging from 00_b-11_b. After mapping all the states with their binary correspondents, K-maps were used to derive equations for F, A*, and B*. Lastly, a proposed electrical drawing was made to simulate the behavior based on the equations derived from the k-map. The design procedure for the sequence detector 101 with overlap is in the appendix below.

After creating both the electrical drawing and deriving the boolean equations from the k-map. The results need to be tested in a virtual environment before they are implemented with physical hardware. This was done by using Verilog. When simulating in Verilog there were 6 key signals that needed to be created to properly observe the correct behavior: CLK, X, A, B, A^*, and B^*. CLK, X, A, and B were input signals and A^* and B^* were output signals, followed by F which was the AND'd logic between A & B. Using behavioral description a 101 sequence detector module was created, and the boolean conditions were applied using the logical operators rather than the built in logic gate functions. Another condition inside this module was that the boolean equations derived would only be applied if there was a rising edge of a clock. Next, a testbench file was created to simulate the behavior of the FSM. The x input values were taken from the provided spreadsheet and stored in a register. This register was iterated over until all cases were checked. Finally, the variables were dumped and saved as a waveform which showed the graphical depiction of the FSM. The Verilog code and waveform are in the appendix below.

After verifying the results of the design, the final part of this laboratory involved creating the actual 101 sequence detector with overlap. As seen by the equations proposed, logical conditions of AND OR and NOT will be needed as well as a memory component (74HC595) to store the current and next states. After acquiring all ICs, three LEDs were used, each representing A, B, and F. Next, each ICs datasheet needed to be consulted to understand which pins correspond as inputs and outputs. Lastly, using the drawing created each IC was wired according to the proposed design; all the while, checking iteratively, to see if the wiring so far was correct by using the provided C++ code. Once the circuit was complete each input bit was compared with the provided excel sheet to validate the results. A picture of the final circuit is attached in the appendix below.

Results

When checking to validate each part of the laboratory, it was done differently for each part. For the design process, it was verified by the Verilog simulation. When simulating the proposed boolean equations in Verilog to see if identical results were present, each random input sequence in Verilog behaved identically to how the state transition diagram was made. Thus, the Verilog simulation proved that the equations derived from the k-maps are a correct representation of the state transition diagram. Next, when constructing the hardware portion of the laboratory each input entered into the serial monitor was compared with the provided excel file. If the hardware behaved as the excel spreadsheet expected, then the circuit was correctly wired and programmed. In the case of this laboratory, the results of the hardware and the excel sheet were identical proving the circuit was correctly made.

Overall, by understanding the design and implementation process of finite state machines, the concepts of digital logic design can all be applied. For example, first a graph needs to be created to represent the flow of data. By understanding the core logic behind the “flow” of data that logic can be implemented through the different techniques taught throughout the course (i.e truth tables, boolean algebra, k-maps, etc.). Ultimately, by having FSMs be one of the final labs, it allows students to better understand the entire design process when creating solutions in the world of digital logic design.

Kahan Patel’s Technical Memo

To: Dr. Christopher Peters, College of Engineering, Drexel University

From: Kahan Patel

Date: 2/23/2024

Re: Registers (Chapters 18-20)

Summary

In this lab, the concept of registers was explored through the design and assembly of the 74HC595 module and a seven-segment LED. This laboratory is a detailed expository study on how the 74HC595 module behaves and how it can be used to expand the total number of ports on an Arduino board. The module that is being used is a serial in and parallel out IC, meaning that it takes in a single serialized input and outputs to different pins in a parallel fashion. This structure is what allows this IC to be used in unison with a microcontroller's pins to expand the total amount available.

Purpose

The goal is to use this principle of SIPO to cycle through different binary combinations that activate different LEDs on the seven-segment LED module. By utilizing the shifting behavior of the 74HC595 module, different serialized inputs can be sent through the module to cycle through different LED combinations displaying the numbers and letters in a sequential pattern. By understanding the concept of how the 74HC595 module works, in particular the different registers inside, the more complex topics of finite state machines discussed in the lecture can be designed and implemented with ease.

Methodology/Approach Section

The first part of this laboratory consisted of understanding how the 74HC595 module works. To do this the structural description of the module was implemented in Verilog. This simulated module consisted of the submodules found inside the 74HC595 module. These modules are the 8 stage shift register, 8 bit storage register, and the 3 state output. When designing the module, it was important that each of these submodules were understood thoroughly. To begin, the 8 stage shift register consists of 3 of the 74HC595 module's inputs: SER, SRCLK, (SRCLR)'. SER is the serialized signal that is inputted to the IC. SRCLK is the clock signal that oscillates continuously as long as the IC has power, and the (SRCLR)' is the active LOW clear signal that clears all data currently stored in the shift register. After understanding how these three signals behave the actual shift register can be simulated. It is known that one bit of data is inputted into the shift register during the rising edge of SRCLK and this is only true if (SRCLR)' is HIGH. So, by using an *always* block this logic can be simulated if all the following conditions are met. Furthermore, each time there is a rising edge the least significant bit should be shifted to the right and concatenated with the most recent serial input (SER). Continuing, if (SRCLR)' is LOW then the shift register should be cleared. Lastly, the most significant bit should be sent an overflow handler (in this it is QH').

The next submodule that needs to be simulated is the storage register. This register receives its input from the shift register after 8 rising edge clock cycles (1 byte of data). However, this submodule has a dedicated RCLK signal that clears the current state of the storage register. Once data is stored in the storage register it needs to be passed to output (Q). However, there needs to be a signal that is enabled before the stored values are outputted. In this case, this value is (OE)'. So, knowing this, using an always block like before, if there is a rising edge in RCLK then the shift register values are passed to the storage register. And if (OE)' is LOW the values in the storage register are outputted. After creating both of these submodules the behavioral description of SN74HC595 module is complete and it can be implemented with other modules to create actual systems.

The next part of the laboratory consisted of using the developed module to simulate a 7-segment LED. When simulating the LED, the lookup table needed to be considered to see each binary representation of the LED (used table provided in lab manual); this was done because each number/letter needs a particular binary combination to be correctly displayed by the 7-segment LED. Each value was stored in a vector with a depth of 16 bits(16 different

combinations). Then in the main initial block in the testbench file (where the SN74HC595 is simulated) all the different combinations were iterated over and assigned to SER. These values will now be assigned to the shift register and they will eventually be outputted sequentially. The code and waveform showing this is attached in the appendix below.

The next part of his laboratory involved using the simulated circuit to create the “live” circuit that will cycle through 0-9 and a-f. The way this was done was by following the simulation in Verilog and the specification sheet of the 7-segment LED. Each pin on the LED corresponds to a particular diode found in the LED module. If the pins on the LED are provided the same signal that is shown on the lookup table by the SN74HC595 module, the lights will display a character from 0-f until there is no more power. The input signals to the SN74HC595 module were provided by the Arduino using the given C++ code to shift between each LED combination. The given C++ code was similar to the Verilog module file, since it was describing how the SN74HC595 would behave based on the different binary signals sent to the module. A picture showing the activation of the circuit is in the appendix below.

Results

The results for part 5A of the Verilog were identical to how the 74HC595 signals were displayed on the official datasheet. Thus, it proved that the 74HC595 was simulated correctly. Next, when doing 5B the output of each storage register was outputted to Q with the same delay between each update. Moreso, the binary combination that was being shifted was identical to the combination found on the lookup table. Since this was the case, it indicates the correct binary values were being sent as output to each pin. Now, to test to see if the Verilog can be replicated in reality the 7-segment LED circuit was made and the same pattern of input was sent to the LED from the Arduino using the 74HC595. When observing the results, both the 7-segment LED and the Verilog produced identical results cycling through all the different combinations to out the LED from 0-f. Since this was the case, it showed that both systems were correctly implemented.

Kahan Patel's Technical Memo

To: Dr. Christopher Peters, College of Engineering, Drexel University

From: Kahan Patel

Date: 2/16/2024

Re: Latches & Flip Flops (Chapters 15-17)

Summary

In this lab, flip-flops and latches were explored through the design and assembly of both circuits. Both of the following circuits introduce the concept of sequential circuits, which are a way to design circuits to store the memory of a previously executed signal. By retaining a previously stored signal, the concept of a register can be further explored. By having a register, the building blocks of the initial design of volatile memory architecture can be constructed to create systems like JK flip flops and Toggle flop flops. These volatile memory systems are essential for electronic devices like computers, phones, etc which need to access different signals in rapid succession. The way flip flops and latches receive their signals is from a clock. This clock is an oscillating DC voltage that alternates between HIGH and LOW states. Both circuits utilize either the edge or level of a signal to relay the appropriate signals to connected outputs. Moreover, by having a regulated clock (specific amplitude and frequency) the signals received can be more predictable leading to less erratic behavior in the circuit.

Purpose

Before this week's lab, all the circuits made were combinatorial, meaning they couldn't store past signal values. Even though there is still potential with combinatorial circuits, having a way to store past inputs allows for the creation of systems that can quickly manipulate data and execute specific operations based on inputs that may not be actively being sent to the system. As discussed this week in the lecture, by knowing sequential circuits, systems such as finite state machines can be created to understand the behavior of circuits that may have clock signals inputted.

Methodology/Approach Section

In the first part of this laboratory both the D-Latch and Flip Flop circuits needed to be implemented in Verilog. To begin, the D-Latch was first simulated by creating both a structural and behavioral model. Both models have identical outputs; thus, in a testbench both modules were instantiated and iterated in a for loop for 20 transitions. For each transition there were also random nonperiodic EN and D signals simulated to test the level-triggered behavior of the D-latch. Next, after successful simulation of the D-Latch the D-Flip-Flop was simulated. Similar to the D-Latch, first the structural module for the D-latch was made. After making this initial module the module for the flip flop was made. The flip flop was made using both the structural and behavioral descriptions (same as lab 4A). However, important to note, when making the behavioral model for the flip flop the module for the latch did not need to be instantiated, since the behavioral model only addresses how the module behaves when there is a rising edge. After creating both modules, a test bench was made and the outputs were recorded. For the final part of the simulation, both the D-Latch and D-Flip Flop need to be simulated. The way this was done was by first taking the behavioral module for the latch and flip flop and inserting the code into the design.sv file. Then in the testbench file both modules are instantiated with clock, data, and output parameters. To simulate different instances the random keyword is used to get nonperiodic signals. These random signals allow different cases to be tested without hardcoding the particular signals. The files are then dumped as a vcd and the waveform is analyzed. When looking at the waveform both modules behave differently to high signals (to be expected). When testing, this behavior was identical to how both modules should behave theoretically. So, the results prove successful. The Verilog code and waveform files are attached in the appendix below.

The final part of this laboratory involved the building of the D-Latch. This required referencing the electrical diagram for a D-Latch, which contains both an SR-Latch part with NOR gates and a gated part with both AND and NOT gates. It's important to note that the D-Latch has two inputs, so selecting the correct Integrated Circuits (ICs) becomes crucial. Once the appropriate ICs are selected, the next step is to wire the buttons to either the NOT gate or AND gate as needed. The output of the AND gate then becomes one of the inputs for the NOR gate, while the output of the NOR gate serves as the second input for either gate. Finally, the positive end of the LED is connected to the output of the NOR gate, and the EN, Data, and Q are wired to the Arduino for proper functioning. This entire process ensures that the D-Latch is built according to the required specifications and functionality outlined in the electrical diagram.

After using the provided C++ code, the serial plotter in the Arduino IDE was inspected to see different signals sent for different button press combinations. Since the D-Latch is level triggered, if that behavior is captured in the plotter then the wiring is correct. The results of the Arduino and the physical circuit are in the appendix below.

Results

The results outputted by the Arduino serial plotter and the Verilog waveform behave identically when the signal is at a HIGH level. It is worth noting that both the Arduino and Verilog simulation utilize the same EN, Data, and Q pins, which ensures that the state tables generated by both are also identical. This identical behavior between the simulation and physical execution demonstrates the successful implementation of the logic. Additionally, in the case of the Verilog code, both the structural and behavioral modules of both the D-Latch and D-Flip Flops produce identical outputs. This further highlights the capability of both methods to accurately represent the same logic. Considering this reliable consistency, one can confidently expect that if the flip flop were to be physically implemented, it would behave identically to the Verilog results. Therefore, the alignment between the simulation and physical execution validates the functionality and reliability of both the Arduino and Verilog approaches.

Kahan Patel's Technical Memo

To: Dr. Christopher Peters, College of Engineering, Drexel University

From: Kahan Patel

Date: 2/09/2024

Re: Decoders (Chapters 12-14)

Summary

In this lab, decoders were explored, and the many applications the device potentially has. Decoders are a combinational logic circuit that takes n amount of inputs and outputs 2^n outputs. By having this modular nature, decoders can be used to switch between multiple output connections in rapid succession, allowing for more precise control over different electrical systems. In particular, this lab will explore the design and implementation of decoders to simulate all outputs on a given decoder and then implement this concept using LEDs to visualize the different conditions of an 8-bit decoder.

Purpose

In the first two labs, logic gates were used to create small circuits to test the different combinations different gates can produce when they are used separately or combined. However, lab 3 is a blend between both by introducing decoders. Decoders have a combination of AND, OR, and even NOT gates. So, by having a thorough understanding of basic logic gates, understanding more complex modules like decoders is intuitive and more digestible. Furthermore, recently, concepts of memory and sequential circuits were introduced, and both of the concepts involve using multiple different gates to execute more complex logic, so by understanding decoders, their logic can be implemented into future circuits.

Methodology/Approach Section

First, this laboratory involved simulating the oscillating light pattern that appears on the "Nightrider" car. The simulation was done in Verilog using both a 3x8 decoder and a 4x16 decoder. This was done by first modeling the decoder after the SN74HC138 integrated circuit. When simulating the behavior of this decoder it was important to note that it is an active LOW decoder. This meant that at ~0V it would be a logical 1 and at ~5V it would be a logical 0. Knowing this, every pin on the SN74HC138 was defined corresponding to its signal type (input or output). These were then stored using the reg type and assigned the appropriate amount of bits (3-bit input) (8-bit output). Next, all the case flags were initialized depending on whether the output was all HIGH or LOW. Lastly, the behavioral structure was made using a case block that sequentially executed each input condition for the decoder and its corresponding binary output. Now that the module was designed it was implemented in the testbench file. When iterating each condition in the decoder, a for loop was used to increment each bit by one place. This incrementation represented the bit that corresponded to the LED's open position. The bits would shift left until position 7. After position 7, the bit was decremented by 1 until the binary output was the same as the beginning (00000000). Then the variables were dumped in a VCD file and the waveform was analyzed.

The next part of the laboratory involved using this initial 3x8 to implement a 4x16 decoder. The way a 4x16 decoder was made was by the concept of cascading decoders. Essentially, every decoder has an enabler (EN1), if the enabler is off the entire decoder is off, and vice versa. Thus, the enabler can be used as the most significant bit to switch between two different decoders. This switching behavior can be done by using a NOT gate. So, whenever the enabler is 0 the complement is 1 leading at least 1 out of the 2 decoders to be on. Moreover, whenever a decoder is in the off state, its output is concatenated with the EN = 1 decoder. This leads to the output now being 16 bits. However, since there are more possible states now, the total amount of interactions in the for loops needs to increase and G1 needs to be assigned to the iterator (i) after each iteration in each loop. Lastly, this new 4x16 decoder is no longer active LOW, rather it is active HIGH. So, the output signal needs to be inverted so that now all bits are 0 except for 1. In short, by toggling G1 after each 8th sequence, concatenating the outputs of each decoder, and inverting the output, two decoders can be instantiated and combined using the method of cascading decoders to simulate the behavior of a 4x16 decoder using only 2 3x8 decoders. The Verilog code for the 3x8 and 4x16 coder is attached in the Appendix below.

The final part of this laboratory involved taking this simulation and implementing it into a working physical design. The physical design was based on the single 3x8 decoder; however, since the SN74HC138 is an active LOW decoder a NOT gate had to be used to invert the signal. The setup involved using 8 LEDs, connecting the anode to the NOT gate output and the NOT gate input to the SN74HC138 output. This process was repeated for each LED (VCC, GND, G'2A, G’2B, and G1 were either wired to positive or negative). Lastly, the 3 inputs were supplied by the Arduino. Each of the possible signals and the oscillating behavior were compiled in the provided C++ script. Then, when the Arduino was given a voltage the LEDs began alternating between on and off. The oscillating behavior of the LEDs lasted forever until there was no longer a voltage source being applied. A picture of the final circuit is attached in the appendix below.

Results

The Verilog simulation of the 3x8 decoder and the actual implementation had identical outputs. Thus, the results prove that hardware implementation has been wired correctly. Moreover, by having identical results it shows the ability the Verilog language has to simulate potentially extremely complex circuits before physical execution. This can be greatly beneficial, since time and resources can be properly allocated during the design of a particular system before it is actually implemented. This shows the real world and practical benefits of first simulating the circuit to see if it can be physically made before actually creating it.

By having Verilog as a robust simulation tool future projects can be first, like this one, be simulated in a testing environment to see the behavior of a particular circuit before its structure is actually made.

Kahan Patel's Technical Memo

To: Dr. Christopher Peters, College of Engineering, Drexel University

From: Kahan Patel

Date: 2/02/2024

Re: Sum of Minterms and K-maps (Chapters 8-11)

Summary

In this lab, it explored the concepts of sum of minterms and how they can be simplified using karnaugh maps (k-maps). These maps are a powerful tool for minimizing the expression of sum of minterms. Before introducing this lab, the sum of minterms expression was simplified using boolean algebra simplification. However, now k-maps will be used to validate the simplification and ensure its accuracy. Once the logical equation has been simplified, a circuit diagram is created based on it. This circuit diagram is a visual representation of the logical expression and serves as a blueprint for constructing the actual circuit. To ensure the correctness of the design, all possible conditions are simulated using both TinkerCAD and Verilog. This rigorous testing process allows the identification of any potential issues or discrepancies in the circuit implementation. Finally, a comparison between the results obtained from the truth tables generated in all the tests with the original given table is executed. This step is crucial in validating the accuracy of the simplified logical equation and circuit. By comparing the results, the different tests can confidently confirm that the circuit is functioning correctly and accurately representing the given truth table. Overall, this lab provides a comprehensive exploration of sum of minterms, karnaugh maps, logical equation simplification, circuit design, simulation, and result validation.

Purpose

This laboratory is designed to expand on the principles of the basic logic gate circuits created in the first lab by combining different gates to result- in more complex circuits. Furthermore, the teachings of the boolean algebra techniques taught prior are expanded upon to manage the complexity of the newer, more complex circuits explored in this laboratory section. In addition to the increase in complexity in this laboratory, more complex topics recently discussed in the lecture, such as decoders and multiplexers are reliant upon the combination of different logical gates to create their internal architecture. Thus, if the concepts in this laboratory are first introduced in an approachable manner, understanding more complex modules becomes easier, undermining the intricate nature of the new concepts.

Methodology/Approach Section

Similar to the last laboratory, this lab is distributed into a three part procedure. The first part of the laboratory involves representing the truth table provided into its sum of minterms form. It is important to mention that the sum of the minterm equation can be represented in multiple ways. The way it was done in this laboratory was, first, in its sigma notation. The sigma notation denotes each row of the truth table, beginning its indexing at 0. Each time the table's output (F) is 1 the row is noted. This is done for all sixteen combinations of the four bit system. After noting each significant row, a boolean expression using alphabetical terms can be easily created for each row that outputs 1. If the input signal is 0 the alphabetical term is represented by its complement (ex. A'). Thus, after iterating through the entire sum, each row's alphabetical representation is added (or'd) with each resulting expression. After the entire loop a maximal sum of minterms can be created. However, when approaching this design it would be inefficient because of the large size the circuit would need to occupy to output the result. In this case, next, a simplified expression needs to be created that can capture the entire logic of the max term expression. The way this was done was by using both properties of boolean algebra and k-maps. After achieving the simplified equation, a minimal sum of products, and a simplified circuit was constructed. This circuit was then designed in TinkerCAD. The maximal and minimal SOPs are shown in the appendix below, as well as the simplified and TinkerCAD circuits.

The next part of the laboratory involved validating the circuit created. Verilog was used to validate the signals for each possible condition sent. This was done by first assigning each input (A, B, C, D) with the *reg* type and output (F) as a *wire* type. Then, using the *assign* identifier the simplified boolean equation was imputed into Verilog using the previously assigned inputs and output. Sequentially, Verilog tested all possible conditions for the circuit and created a VCD file

to view the waveform. After viewing the created waveform, all output signals are compared to the truth table to check if the results match. The Verilog code and the waveform are attached in the appendix below.

The final part of this laboratory is taking the results of both the Verilog simulation and TinkerCAD circuit to create “the live” circuit. The way the actual circuit was implemented was similar to the TinkerCAD simulation. However, instead of having actual buttons, the logic was controlled by the Arduino’s microcontroller. First, three different IC’s were needed (74HC08, 74HC32, 74HC04). Both the 74HC08 and 74HC32 have four two-input AND and OR gates. The 74HC04 has six NOT gates. Since the ICs are made using DIP both ends of the gates are electrically isolated (parallel pins). Thus, it was vital when building the circuit to inspect the datasheet to ensure each pin was correctly wired. The input signals were directly connected to Arduino’s GPIO pins. The signals from the pins were either sent to the AND or NOT gates. These gates would first perform the “product ” part of the expression checking the conditions of the first two inputs. In terms of sequential wiring, if a signal needed to be complemented it would first go to the NOT gate then the inverted output would be an input for the AND gate. In total, there were two NOT gates and two AND gates used. The final part of the wiring involved taking the outputs of both AND gates and putting them as inputs for the OR gate. The final output of the gate was then inserted into the Arduino. This output signal from the OR gate was what generated the truth table on the computer. The table was generated by taking the output of the OR gate and putting it back into the GPIO pin of the Arduino. Then, using the C++ script provided, a truth table was generated in the IDE’s serial monitor. Ultimately, the truth table generated is compared to the results of the Verilog simulation and the original table provided. The final circuit and truth table are in the appendix below.

Results

After the completion of the laboratory, the results were identical to the truth table given initially. By having such results, it proves the numerous ways a system can be described. Moreover, each different representation has its positives and negatives. For example, the maximal sum of product expression is long and not practical in a final solution; however, the maximal solution is able to completely describe the system, showing every possible combination of inputs explicitly. By having different ways to express the same system the ambiguity and complexity of more complex systems can be easily managed since there are many different representations of one system. Furthermore, since any of these ways can be

used to represent the system, if only one of the items is given, that sole item can be used to create all other representations.

In conclusion, the theoretical concepts of Boolean algebra simplification and truth tables can be directly used to help understand the behavior of an entire system. This lab emphasizes this behavior on three different platforms. Ultimately, by understanding the nature of more complex logical circuits in this laboratory, these circuits can be joined together to create more complex modules to execute more intricate and complex computational operations in future laboratories.

Kahan Patel’s Technical Memo

To: Dr. Christopher Peters, College of Engineering, Drexel University

From: Kahan Patel

Date: 1/19/2024

Re: Basic Logic Gates Lab 1 (Chapters 3-6)

Summary

In this lab, several basic logic gates were explored, and their fundamental characteristics were analyzed through practical laboratory-designed tests. These tests ranged from digitally designed basic electrical systems by using Verilog to simulate logic gate outputs over given intervals to “hands-on” designing and building of basic logic gate circuits using Arduino and TinkerCAD. The following memo outlines the purpose of the laboratory and how the different procedures were accomplished.

Purpose

This laboratory introduces students to the world of digital logic design through the analysis of both physical and digital circuits. By assembling different logic circuits, the theory learned in lecture and recitation can be further strengthened in students' understanding of how logic circuits input and output different values. It is important to remember that in this lab, only two inputs were given to each circuit, lowering the complexity of the truth tables generated by the different logic gates. However, after understanding the basic structure of a logic circuit, students can use the attained skills to increase the complexity of future circuits to create projects with more complex computational potential. Furthermore, topics such as binary numbers and the sum of minterms were discussed recently. This is important because the topics mentioned above involve further complexity; so, by having a better basis for understanding digital logic,

students can find newer concepts more intuitive, undermining the complex nature of these new topics and allowing for better implementation in future labs.

Methodology/Approach Section

The laboratory is designed as a three-part procedure utilizing Verilog, TinkerCAD, and Arduino. When approaching the Verilog portion of the laboratory, each logic gate and its complement needed to be simulated. The way this was accomplished was by using the built-in Verilog functions for each gate to create an *all_gates_s* module. The function parameters included inputs (a, b) and the output variable for each gate. Then a testbench is created to instantiate the module. The testbench then sequentially inputs different values into the function parameters to simulate all the possible combinations for the two input logic gates. Lastly, the results are dumped as a vcd file in which the waveform can be observed on graphical software. (ex. gtkwave). By having graphs for all the different logic gates, the truth tables can be easily created and used to compare the results of the other parts of the lab.

However, because many students could not open the GTKwave software on their Mac computers, the graphical output has been omitted from this memo. Nonetheless, the code needed to output the waveform is attached in the appendix.

The next part of the laboratory involved TinkerCAD to model the physical circuit before its eventual assembly. TinkerCAD incorporates graphical components that allow for easy assembly of circuits. However, it was vital to reference the datasheet for each logic gate. The data sheet included crucial information regarding the pin configuration of each gate. For example, when assembling the NOR gate logical circuit, pin 13 changed from being input to an output, and pins 12 & 11 became input. Thus, by consulting the appropriate datasheet during the building of the circuit the correct pins would be allocated as input and output. The two inputs for each gate were executed by two tactile buttons, and the output was received by a single LED. Each button and the LED needed to have the appropriate resistor wired in series to ensure nothing would burn out or break when the circuit was live. To test all gates, the logic gate was replaced each time, and the outputs were recorded. An example picture of a NAND gate is shown in the appendix.

Next, using the TinkerCAD simulated circuit, a physical circuit can be identically created using the parts in the Arduino kit. This assembly was similar to the design process on TinkerCAD, but it was done physically. There were small factors to consider, such as ensuring that the polarity was

not inverted and that all components were electrically connected. Nonetheless, when each physical gate was constructed with reference to the TinkerCAD simulation, the circuits behaved identically to TinkerCAD. The physical NAND gate circuit is shown in the appendix.

The final portion of the lab required the simulation of the logic gates with the Arduino itself. The approach to this part of the lab involved connecting the positive end of the diode to the digital GPIO pin on the Arduino. This involved wiring four LEDs in parallel to represent AND and OR gates and their complements (NAND and NOR). Moreover, the part that changed the most was the removal of the logic gates. Now, all logical inputs and outputs are directly controlled by the Arduino board. To create the digital logic for the LEDs, a modified version of C++ was used to program the Arduino to execute specific outputs based on different button combinations. In turn, this new circuit can generate all the truth tables for all four logic gates simultaneously without using any physical logical gate as before. Overall, by using the Arduino last, all four logical circuits can be simplified into one circuit. This concept summarizes the teachings of the lectures in the past two weeks while showing the practical benefits of creating a simplified circuit. The picture of the circuit is shown in the appendix below.

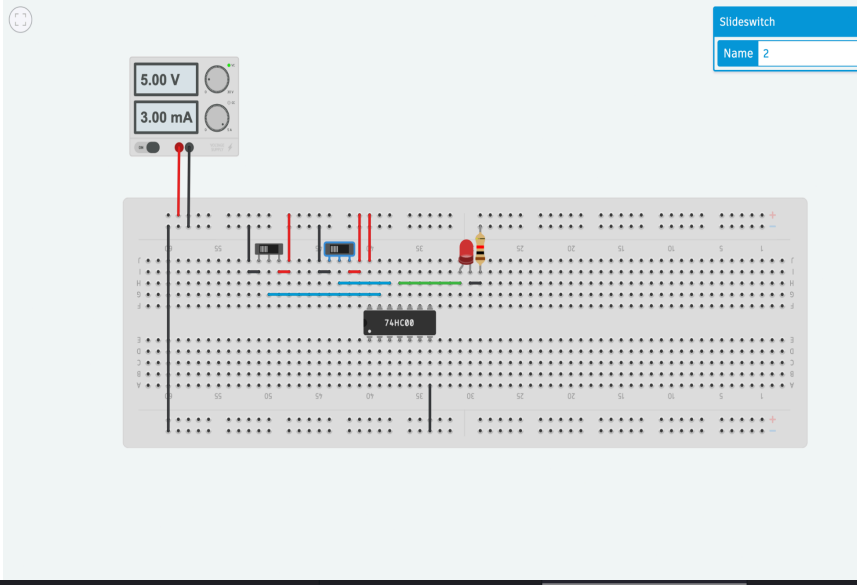
Results

After completing each part of the lab, different truth tables for each logic gate were created. Since this lab used the same logic gates (with the same number of inputs...excluding xor) the truth tables on every medium were identical. Since all the truth tables were the same, the results of the laboratory can be proven as accurate to what each gate should output under identical conditions. Furthermore, the results highlight the modular nature of the different logic gates. For example, if more inputs were added to each gate, the complexity of the new truth tables would be far more complex than the results of this lab. Because of this, it highlights the fascinating nature of digital logic and the many applications of logical gates in real-world circuits and systems.

In conclusion, the theoretical concepts of Boolean algebra and truth tables can be directly used to help understand the logical nature of logic gates. This lab emphasizes this behavior on three different platforms. Ultimately, by understanding the nature of logic gates in this laboratory, circuits of more complexity and computation potential can be created to solve more abstract problems.

Appendix: Basic Logic Gates Lab 1 (Chapters 3-6)

TinkerCAD Simulation (74HC00 NAND).....



Verilog Code.....

verilog_lab_1c.v

```
//All gates using Structural modeling
module all_gates_s(
        input a,
        input b,
        output not_out,
        output and_out,
        output or_out,
        output nand_out,
        output nor_out,
        output xor_out);

    not(not_out,a);
    and(and_out,a,b);
    or(or_out,a,b);
    nand(nand_out,a,b);
    nor(nor_out,a,b);
    xor(xor_out,a,b);

endmodule
```

verilog_lab_1c_tb.v

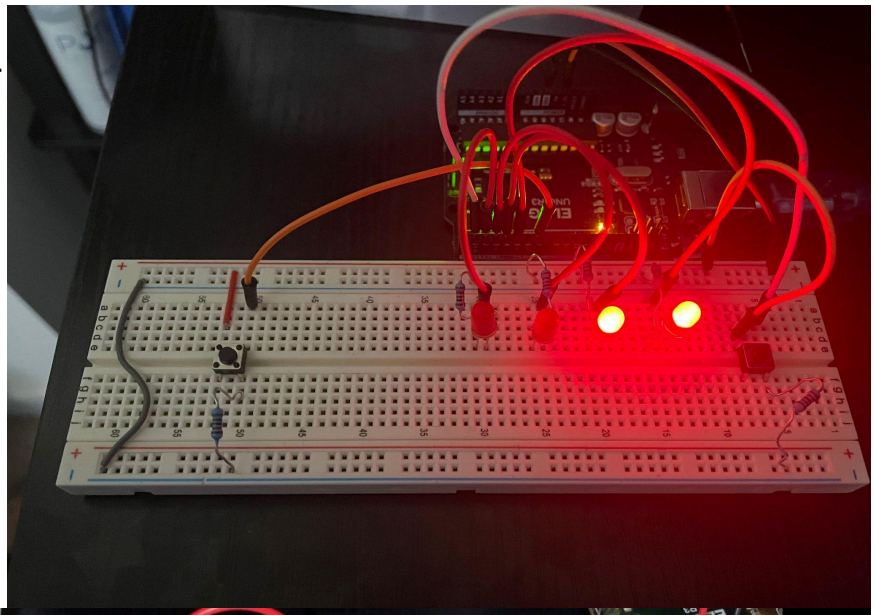
```
`timescale 10ns/1ns
module all_gates_tb;

reg A, B;
wire OUT_NOT, OUT_AND, OUT_OR, OUT_NAND, OUT_NOR, OUT_XOR;

all_gates_s uut(A, B, OUT_NOT, OUT_AND, OUT_OR, OUT_NAND, OUT_NOR, OUT_XOR);

initial begin
 A = 0; B = 0;
 #1
 A = 0; B = 1;
 #1
 A = 1; B = 0;
 #1
 A = 1; B = 1;
 #1
 $finish();
end

initial begin
 $dumpfile("verilog_lab_1c.vcd");
 $dumpvars(1,A,B,OUT_NOT, OUT_AND,OUT_OR,OUT_NAND,OUT_NOR,OUT_XOR);
 end
endmodule
```

Arduino Code……..

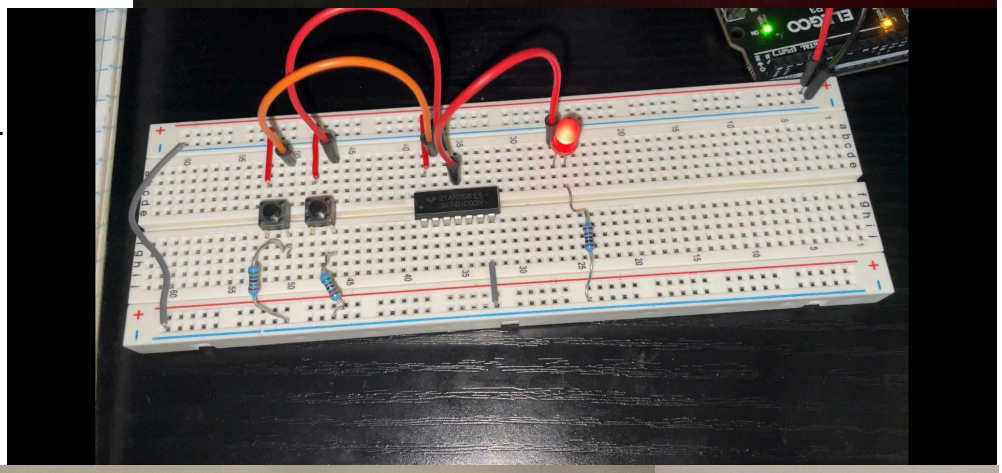
src > main.cpp > loop()

```
#include <Arduino.h>
#include <Adafruit_Sensor.h>

int button1 = 7;
int button2 = 1;

int led1 = 2;
int led2 = 3;
int led3 = 4;
int led4 = 5;

void setup() {
 // Each digital pin used must be declared as
 // INPUT or OUTPUT before being used

 pinMode(button1, INPUT); // Button A input
 pinMode(button2, INPUT); // Button B input
 pinMode(led1, OUTPUT); // AND pin
 pinMode(led2, OUTPUT); // OR pin
 pinMode(led3, OUTPUT); // NAND pin
 pinMode(led4, OUTPUT); // NOR pin
 }

 void loop() {
 // Read in both buttons
 int buttonA_value = digitalRead(button1);
 int buttonB_value = digitalRead(button2);

 digitalWrite(led1, buttonA_value && buttonB_value);
 digitalWrite(led2, buttonA_value || buttonB_value);
 digitalWrite(led3, !(buttonA_value && buttonB_value));
 digitalWrite(led4, !(buttonA_value || buttonB_value));
 }
```

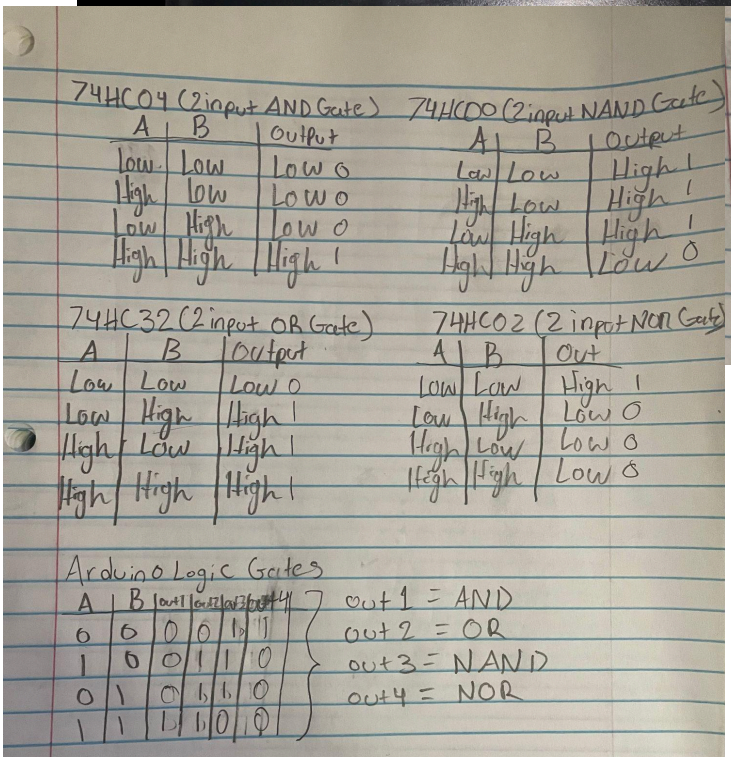
Arduino Circuit……..



74HC00N Circuit (NAND)……



Logic Gate Output……



74HC04 (2 input AND Gate)

A	B	Output
Low	Low	Low 0
High	Low	Low 0
Low	High	Low 0
High	High	High 1

74HC00 (2 input NAND Gate)

A	B	Output
Low	Low	High 1
High	Low	High 1
Low	High	High 1
High	High	Low 0

74HC32 (2 input OR Gate)

A	B	Output
Low	Low	Low 0
Low	High	High 1
High	Low	High 1
High	High	High 1

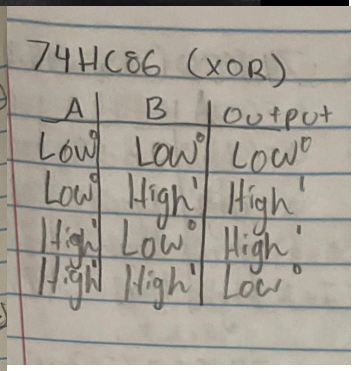
74HC02 (2 input NOR Gate)

A	B	Out
Low	Low	High 1
Low	High	Low 0
High	Low	Low 0
High	High	Low 0

Arduino Logic Gates

A	B	out1	out2	out3	out4
0	0	0	0	1	1
1	0	0	1	1	0
0	1	0	1	1	0
1	1	1	1	0	0

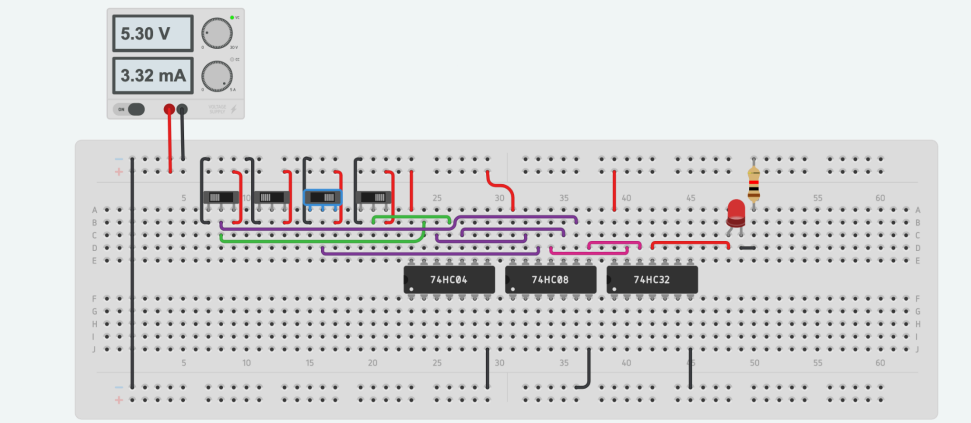
out 1 = AND
out 2 = OR
out 3 = NAND
out 4 = NOR



74HC86 (XOR)

A	B	Output
Low	Low	Low
Low	High	High
High	Low	High
High	High	Low

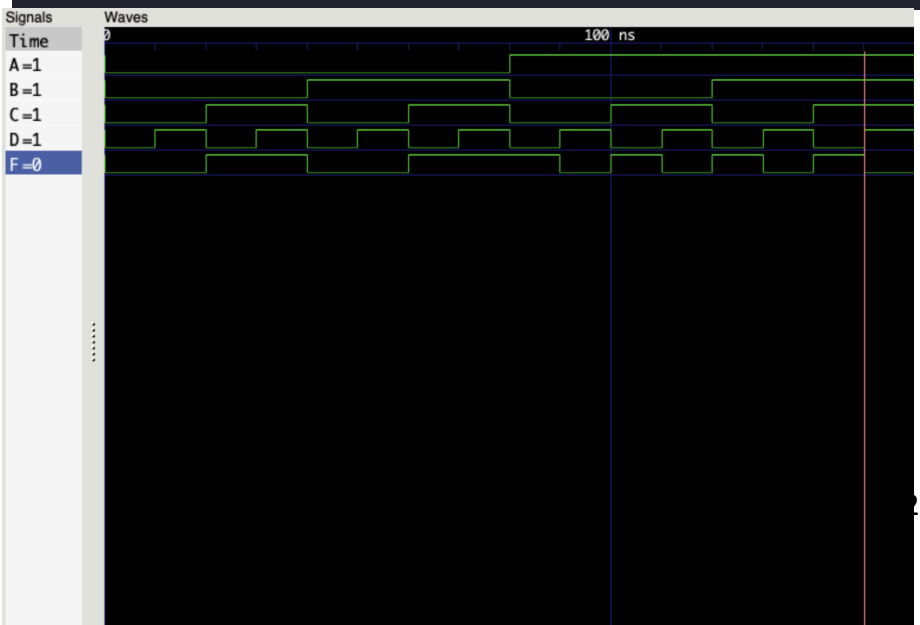
TinkerCAD Simulation……



Verilog Code……

```
lab_2b.v
1   `timescale 10ns/1ns
2   module project_2A;
3
4
5       reg A=0, B=0, C=0, D = 0;
6       wire F;
7
8       assign F = (!A && C) || (A && !D);
9
10      initial begin
11          $dumpfile ("project_2B.vcd");
12          $dumpvars(1,A,B,C,D,F);
13          #16;
14          $finish();
15      end
16
17      always #8 A = ~A;
18      always #4 B = ~B;
19      always #2 C = ~C;
20      always #1 D = ~D;
21  endmodule
22
23
24
25
26
```

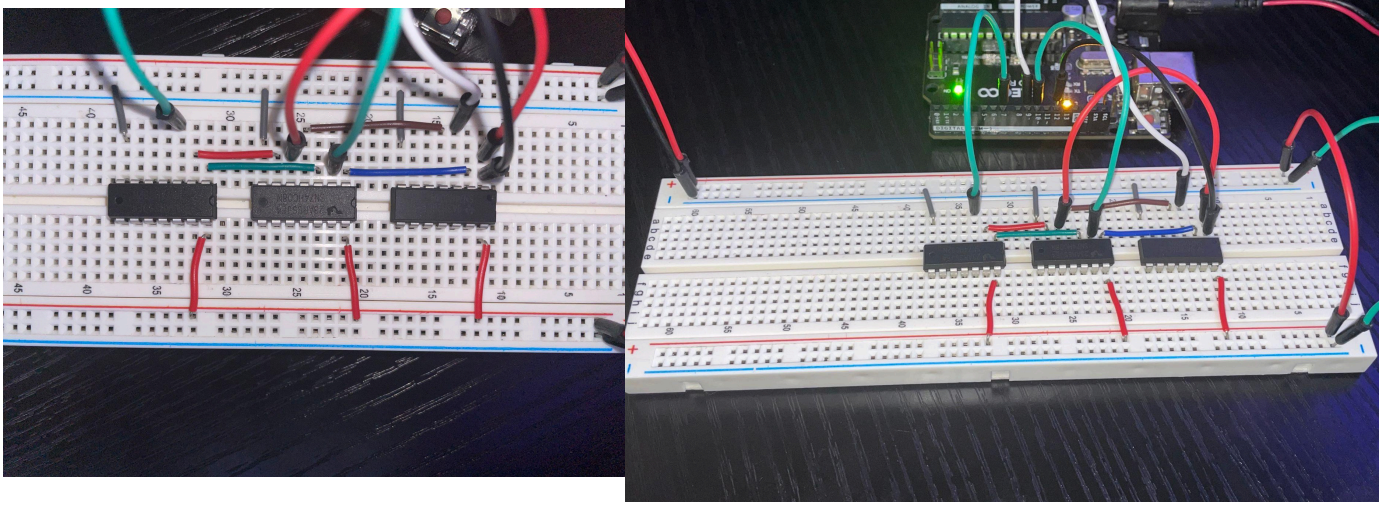
Verilog Simulation Results……



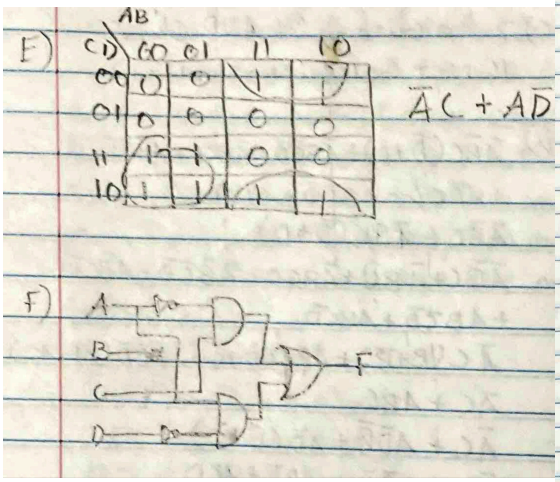
Arduino Truth Table……

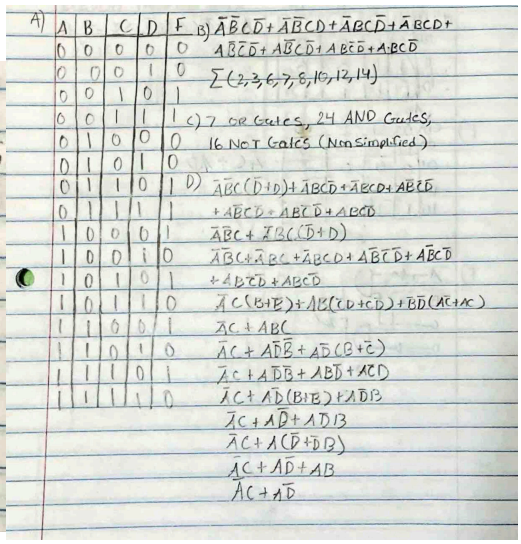
A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

Arduino Circuit……..

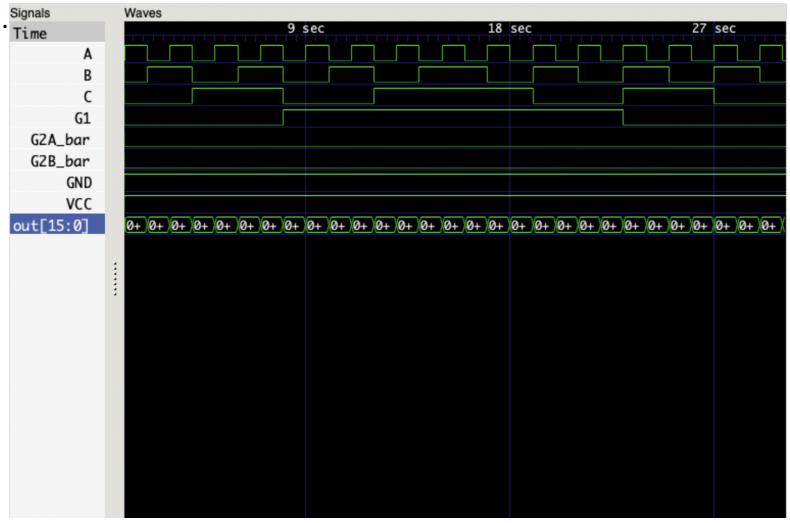


Simplification Of Table & Electrical Schematic…..





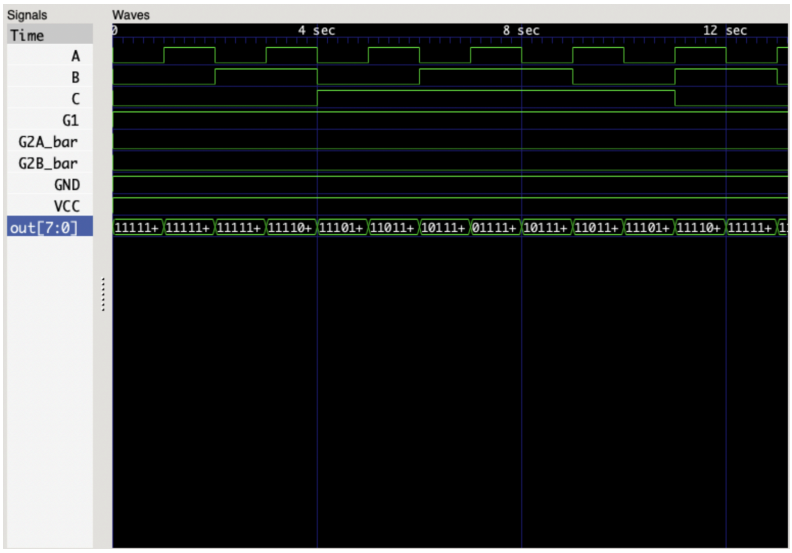
4x16 Decoder (Active HIGH) Waveform..



4x16 Decoder (Active HIGH) Terminal Output……..

```
(base) kahanpatel@Kahans-MacBook-Pro verilog % iverilog -o decoder_tb.vvp decoder_tb.v
(base) kahanpatel@Kahans-MacBook-Pro verilog % vvp decoder_tb.vvp
VCD info: dumpfile verilog_lab_3A.vcd opened for output.
VCC=1, GND=1, A=0, B=0, C=0, G1=0, G2A_bar=0, G2B_bar=0, out=0000000000000001
VCC=1, GND=1, A=1, B=0, C=0, G1=0, G2A_bar=0, G2B_bar=0, out=0000000000000010
VCC=1, GND=1, A=0, B=1, C=0, G1=0, G2A_bar=0, G2B_bar=0, out=0000000000000100
VCC=1, GND=1, A=1, B=1, C=0, G1=0, G2A_bar=0, G2B_bar=0, out=0000000000001000
VCC=1, GND=1, A=0, B=0, C=1, G1=0, G2A_bar=0, G2B_bar=0, out=0000000000010000
VCC=1, GND=1, A=1, B=0, C=1, G1=0, G2A_bar=0, G2B_bar=0, out=0000000000100000
VCC=1, GND=1, A=0, B=1, C=1, G1=0, G2A_bar=0, G2B_bar=0, out=0000000001000000
VCC=1, GND=1, A=1, B=1, C=1, G1=0, G2A_bar=0, G2B_bar=0, out=0000000010000000
VCC=1, GND=1, A=0, B=0, C=0, G1=1, G2A_bar=0, G2B_bar=0, out=0000000100000000
VCC=1, GND=1, A=1, B=0, C=0, G1=1, G2A_bar=0, G2B_bar=0, out=0000001000000000
VCC=1, GND=1, A=0, B=1, C=0, G1=1, G2A_bar=0, G2B_bar=0, out=0000010000000000
VCC=1, GND=1, A=1, B=1, C=0, G1=1, G2A_bar=0, G2B_bar=0, out=0000100000000000
VCC=1, GND=1, A=0, B=0, C=1, G1=1, G2A_bar=0, G2B_bar=0, out=0001000000000000
VCC=1, GND=1, A=1, B=0, C=1, G1=1, G2A_bar=0, G2B_bar=0, out=0010000000000000
VCC=1, GND=1, A=0, B=1, C=1, G1=1, G2A_bar=0, G2B_bar=0, out=0100000000000000
VCC=1, GND=1, A=1, B=1, C=1, G1=1, G2A_bar=0, G2B_bar=0, out=1000000000000000
VCC=1, GND=1, A=0, B=1, C=1, G1=1, G2A_bar=0, G2B_bar=0, out=0100000000000000
VCC=1, GND=1, A=1, B=0, C=1, G1=1, G2A_bar=0, G2B_bar=0, out=0010000000000000
VCC=1, GND=1, A=0, B=0, C=1, G1=1, G2A_bar=0, G2B_bar=0, out=0001000000000000
VCC=1, GND=1, A=1, B=1, C=0, G1=1, G2A_bar=0, G2B_bar=0, out=0000100000000000
VCC=1, GND=1, A=0, B=1, C=0, G1=1, G2A_bar=0, G2B_bar=0, out=0000010000000000
VCC=1, GND=1, A=1, B=0, C=0, G1=1, G2A_bar=0, G2B_bar=0, out=0000001000000000
VCC=1, GND=1, A=0, B=0, C=0, G1=1, G2A_bar=0, G2B_bar=0, out=0000000100000000
VCC=1, GND=1, A=1, B=1, C=1, G1=0, G2A_bar=0, G2B_bar=0, out=0000000010000000
VCC=1, GND=1, A=0, B=1, C=1, G1=0, G2A_bar=0, G2B_bar=0, out=0000000001000000
VCC=1, GND=1, A=1, B=0, C=1, G1=0, G2A_bar=0, G2B_bar=0, out=0000000000100000
VCC=1, GND=1, A=0, B=0, C=1, G1=0, G2A_bar=0, G2B_bar=0, out=0000000000010000
VCC=1, GND=1, A=1, B=1, C=0, G1=0, G2A_bar=0, G2B_bar=0, out=0000000000001000
VCC=1, GND=1, A=0, B=1, C=0, G1=0, G2A_bar=0, G2B_bar=0, out=0000000000000100
VCC=1, GND=1, A=1, B=0, C=0, G1=0, G2A_bar=0, G2B_bar=0, out=0000000000000010
VCC=1, GND=1, A=0, B=0, C=0, G1=0, G2A_bar=0, G2B_bar=0, out=0000000000000001
(base) kahanpatel@Kahans-MacBook-Pro verilog %
```

3x8 Decoder (Active LOW) Waveform……..



Appendix: Decoders Lab 3 (Chapters 12-14) 6

3x8 Decoder (Active LOW) Terminal Output……

```
(base) kahanpatel@n3-27-22 verilog % iverilog -o 3x8_tb.vvp 3x8_tb.v
(base) kahanpatel@n3-27-22 verilog % vvp 3x8_tb.vvp
VCD info: dumpfile verilog_lab_3A.vcd opened for output.
VCC=1, GND=1, A=0, B=0, C=0, G1 = 1, G2A_bar = 0, G2B_bar = 0, out=11111110
VCC=1, GND=1, A=1, B=0, C=0, G1 = 1, G2A_bar = 0, G2B_bar = 0, out=11111101
VCC=1, GND=1, A=0, B=1, C=0, G1 = 1, G2A_bar = 0, G2B_bar = 0, out=11111011
VCC=1, GND=1, A=1, B=1, C=0, G1 = 1, G2A_bar = 0, G2B_bar = 0, out=11110111
VCC=1, GND=1, A=0, B=0, C=1, G1 = 1, G2A_bar = 0, G2B_bar = 0, out=11101111
VCC=1, GND=1, A=1, B=0, C=1, G1 = 1, G2A_bar = 0, G2B_bar = 0, out=11011111
VCC=1, GND=1, A=0, B=1, C=1, G1 = 1, G2A_bar = 0, G2B_bar = 0, out=10111111
VCC=1, GND=1, A=1, B=1, C=1, G1 = 1, G2A_bar = 0, G2B_bar = 0, out=01111111
VCC=1, GND=1, A=0, B=1, C=1, G1 = 1, G2A_bar = 0, G2B_bar = 0, out=10111111
VCC=1, GND=1, A=1, B=0, C=1, G1 = 1, G2A_bar = 0, G2B_bar = 0, out=11011111
VCC=1, GND=1, A=0, B=0, C=1, G1 = 1, G2A_bar = 0, G2B_bar = 0, out=11101111
VCC=1, GND=1, A=1, B=1, C=0, G1 = 1, G2A_bar = 0, G2B_bar = 0, out=11110111
VCC=1, GND=1, A=0, B=1, C=0, G1 = 1, G2A_bar = 0, G2B_bar = 0, out=11111011
VCC=1, GND=1, A=1, B=0, C=0, G1 = 1, G2A_bar = 0, G2B_bar = 0, out=11111101
VCC=1, GND=1, A=0, B=0, C=0, G1 = 1, G2A_bar = 0, G2B_bar = 0, out=11111110
(base) kahanpatel@n3-27-22 verilog %
```

4x16 Decoder (Active HIGH) Verilog Code……..

```
`include "decoder.v"
module SN74HC138_tb;
  reg VCC, GND, A, B, C, G1, G2A_bar, G2B_bar;
  wire [15:0] out;
  reg [3:0] in;
  integer i;

  SN74HC138 uut1 (.VCC(VCC), .GND(GND), .G1(~G1), .G2A_bar(G2A_bar), .G2B_bar(G2B_bar), .A(A), .B(B), .C(C), .out(out[7:0]));
  SN74HC138 uut2 (.VCC(VCC), .GND(GND), .G1(G1), .G2A_bar(G2A_bar), .G2B_bar(G2B_bar), .A(A), .B(B), .C(C), .out(out[15:8]));

  initial begin
    VCC = 1;
    GND = 1;
    G1 = 0;
    G2A_bar = 0;
    G2B_bar = 0;

    $monitor("VCC=%b, GND=%b, A=%b, B=%b, C=%b, G1=%b, G2A_bar=%b, G2B_bar=%b, out=%b", VCC, GND, A, B, C, G1, G2A_bar, G2B_bar, out);


    for (i = 0; i < 16; i = i + 1) begin
      {G1, C, B, A} = i;
      #1;
    end


    for (i = 14; i >= 0; i = i - 1) begin
      {G1, C, B, A} = i;
      #1;
    end

    $finish();
  end

  initial begin
    $dumpfile("verilog_lab_3A.vcd");
    $dumpvars(1, VCC, GND, A, B, C, G1, G2A_bar, G2B_bar, out);
  end
endmodule
```

……

```
module SN74HC138(VCC, GND, G1, G2A_bar, G2B_bar, A, B, C, out);
  input VCC, GND, A, B, C, G1, G2A_bar, G2B_bar;
  output [7:0] out;
  reg normal_ops, all_HIGH, all_LOW;
  reg [7:0] out;
  reg [2:0] in;

  always @(in or VCC or GND or A or B or C or G1 or G2A_bar or G2B_bar)
  begin
    normal_ops = VCC && GND && G1 && !G2A_bar && !G2B_bar;
    all_LOW = !VCC || !GND;
    all_HIGH = VCC && GND && (G2A_bar || G2B_bar || !G1);

    in = {C, B, A};
    if (normal_ops)
    begin
      out = 8'b00000000;
      case (in)
        3'b000: out[0] = 1'b1;
        3'b001: out[1] = 1'b1;
        3'b010: out[2] = 1'b1;
        3'b011: out[3] = 1'b1;
        3'b100: out[4] = 1'b1;
        3'b101: out[5] = 1'b1;
        3'b110: out[6] = 1'b1;
        3'b111: out[7] = 1'b1;
        default: out = 8'b11111111;
      endcase
    end

    if (all_HIGH)
      out = 8'b00000000; // b00000000

    if (all_LOW)
      out = 8'b11111111; // b11111111
  end
endmodule
```

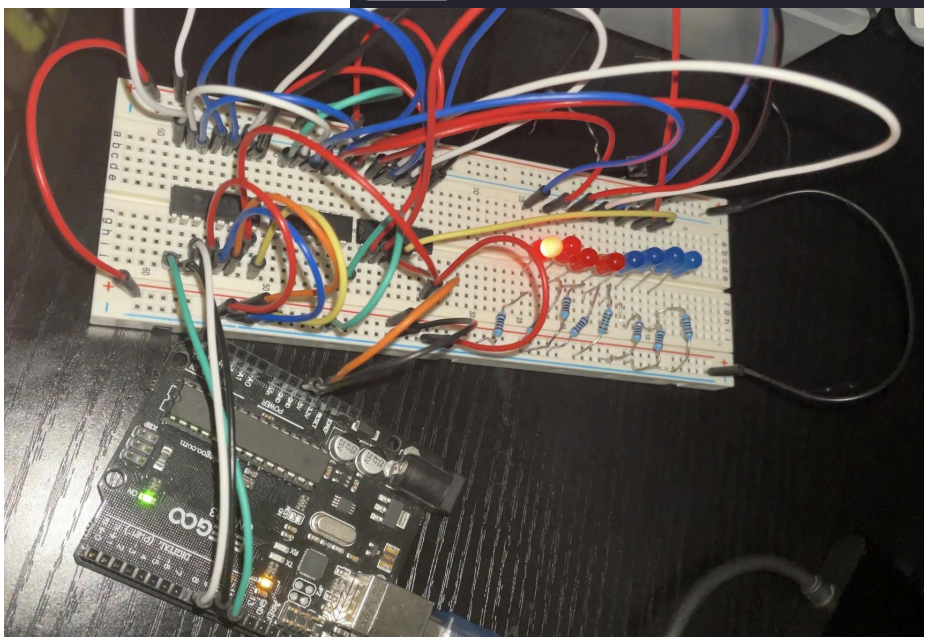
Appendix: Decoders Lab 3 (Chapters 12-14) 7

3x8 Decoder (Active LOW) Verilog Code……

```
`include "3x8.v"
module SN74HC138_tb;
 wire [7:0] out;
 reg VCC, GND, A, B, C, G1, G2A_bar, G2B_bar;
 reg [2:0] in;
 integer i;

 SN74HC138 dut(VCC, GND, G1, G2A_bar, G2B_bar, A, B, C, out);

 initial begin
 VCC = 1;
 GND = 1;
 G1 = 1;
 G2A_bar = 0;
 G2B_bar = 0;
 $monitor( "VCC=%b, GND=%b, A=%b, B=%b, C=%b, G1 = %b, G2A_bar = %b, G2B_bar = %b, out=%b", VCC, GND, A, B, C, G1, G2A_bar, G2B_bar,out);
 for (i=0; i<8; i=i+1)
 begin
 {C,B,A} = i;
 #1;
 end
 for (i=6; i>=0; i=i-1)
 begin
 {C,B,A} = i;
 #1;
 end
 $finish();
 end

 initial begin
 $dumpfile("verilog_lab_3A.vcd");
 $dumpvars(1,VCC, GND, G1, G2A_bar, G2B_bar, A, B, C, out);
 end
 endmodule
```

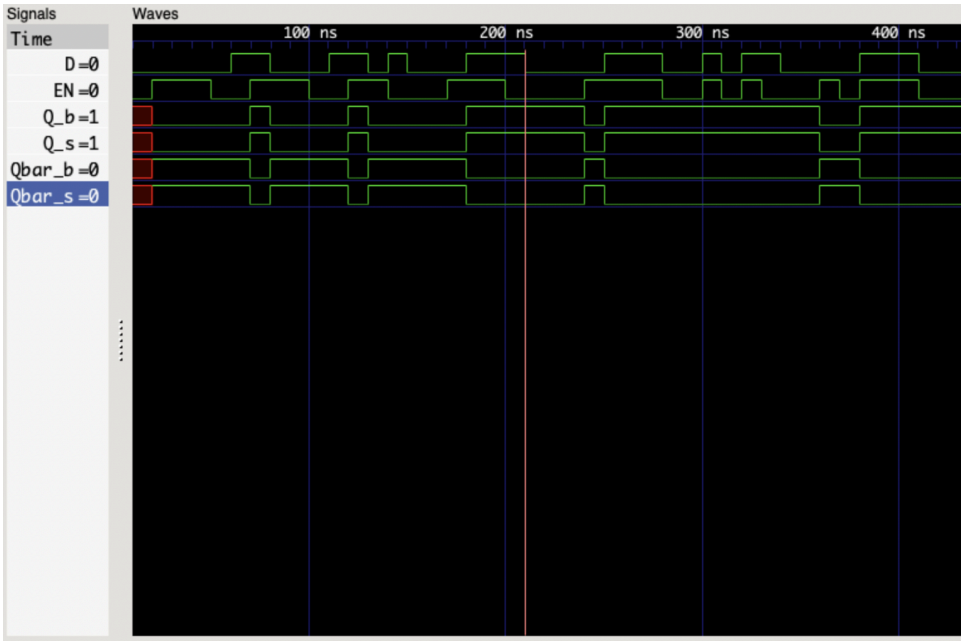
```
module SN74HC138(VCC, GND, G1, G2A_bar, G2B_bar, A, B, C, out);
 input VCC, GND, A, B, C, G1, G2A_bar, G2B_bar;
 output [7:0] out;
 reg normal_ops, all_HIGH, all_LOW;
 reg [7:0] out;
 reg [2:0] in;

 always @(in or VCC or GND or A or B or C or G1 or G2A_bar or G2
 begin
 normal_ops = VCC && GND && G1 && !G2A_bar && !G2B_bar;
 all_LOW = !VCC || !GND;
 all_HIGH = VCC && GND && (G2A_bar || G2B_bar || !G1);

 in = {C, B, A};
 if (normal_ops)
 begin
 out=8'b11111111;
 case (in)
 3'b000: out[0]=1'b0;
 3'b001: out[1]=1'b0;
 3'b010: out[2]=1'b0;
 3'b011: out[3]=1'b0;
 3'b100: out[4]=1'b0;
 3'b101: out[5]=1'b0;
 3'b110: out[6]=1'b0;
 3'b111: out[7]=1'b0;
 default: out=8'b00000000;
 endcase
 end
 if (all_HIGH)
 out = 8'b11111111;
 if (all_LOW)
 out = 8'b00000000;
 end
 endmodule
```

Arduino Circuit……

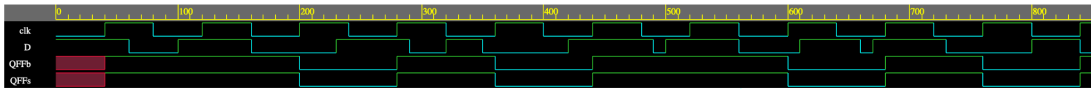


Appendix: Decoders Lab 3 (Chapters 12-14) 8

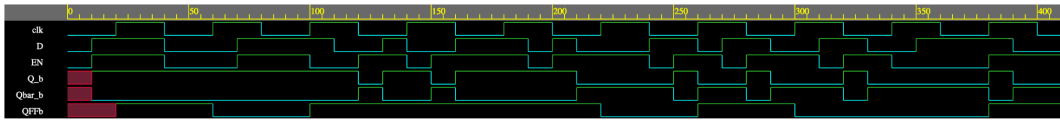
Verilog 4A results……..



Verilog 4B results……



Verilog 4C results……

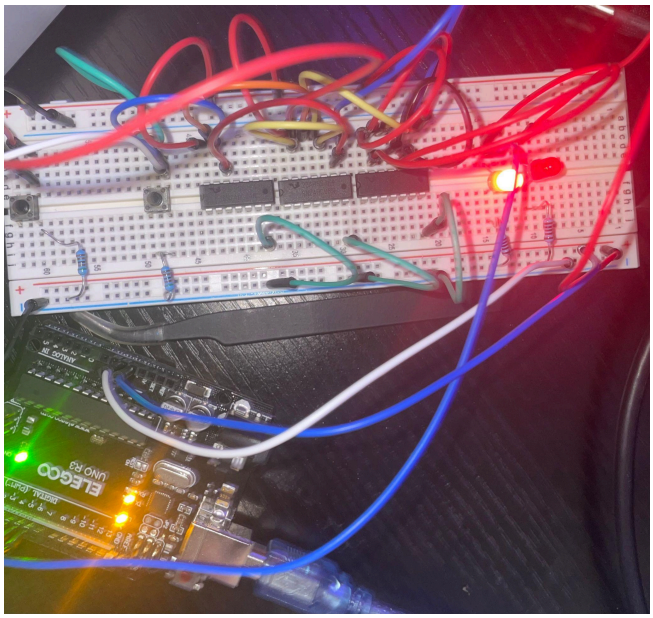


Verilog 4C Code……

```
// Behav for D-Latch
module d_latch_b(input EN, input D, output reg Q, output reg Qbar);
always @ (EN or D)
    if(EN) begin
        Q <= D;
        Qbar <= ~D;
    end
endmodule

// Behav for D-Flip Flop
module DFFb(input clk, input D, output reg Q);
 always @(posedge clk)
    begin
        Q <= D;
    end
 endmodule
```

```
`include "lab5/project_4c.v"
`timescale 10ns/1ns

module Project_5b_tb;
reg clk, EN, D;
reg delay;
reg [1:0] delay2;
wire Q_b, Qbar_b, QFFb;
integer i, NUM_TRANSITIONS;
integer seed = 1;

d_latch_b dlb(.EN(EN), .D(D), .Q(Q_b), .Qbar(Qbar_b));
DFFb dffb(.clk(clk), .D(D), .Q(QFFb));

initial begin
    NUM_TRANSITIONS = 20;

    $monitor("[@%t], clk=%0b, EN=%0b, D=%0b, Q_b=%0b, QFFb=%0b, Qbar_b=%0b",$time, clk, EN, D, Q_b, QFFb, Qbar_b);
    D <= 0;
    EN <= 0;
    #1;

    for(i = 0; i < NUM_TRANSITIONS; i = i + 1) begin
        delay = $random(seed);
        delay2 = $random(seed);
        #(delay2) EN <= ~EN;
        #(delay) D <= i;
    end
    $finish();
end

initial begin
    $dumpfile("project_4c.vcd");
    $dumpvars(1, clk, EN, D, Q_b, Qbar_b, QFFb);
end

always #2 clk = ~clk;
initial clk = 0;
endmodule
```

Arduino Circuit……

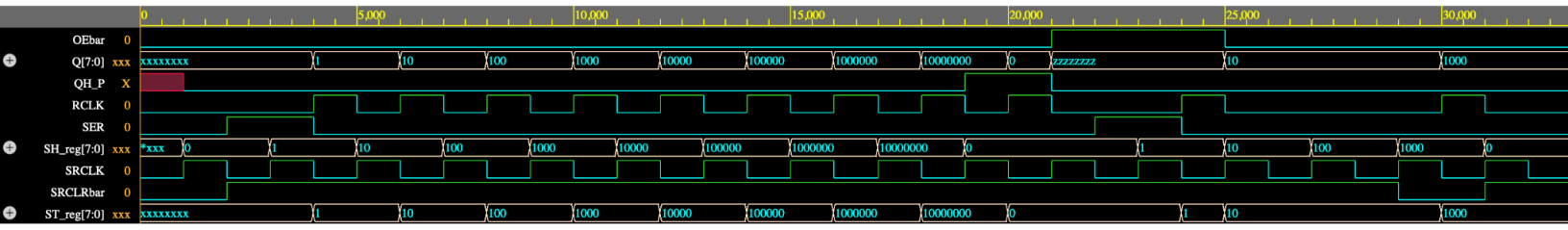


Arduino Output……..

Verilog 5A Code & Results……

```
module SN74HC595 (
    input SRCLK,
    input RCLK,
    input SER,
    input SRCLRbar,
    input OEbar,
    output [7:0] SH_reg,
    output [7:0] ST_reg,
    output [7:0] Q,
    output QH_P);

    reg [7:0] SH_reg, ST_reg, Q;
    reg QH_P;
    integer i;

    // Shift Register
    always @(posedge SRCLK) begin

        if(SRCLK) begin
            QH_P <= SH_reg[7];
            SH_reg[7:0] <= {SH_reg[6:0], SER};
        end

        if(!SRCLRbar) begin
        SH_reg[7:0] <= 8'b00000000;
        QH_P <= 0;
        end
    end

    always @(posedge RCLK or OEbar) begin
        ST_reg[7:0] <= SH_reg[7:0];
        if(!OEbar) begin
            Q[7:0] <= SH_reg[7:0];
        end
        else begin
            Q[7:0] <= 8'bzzzzzzzz;
        end
    end
endmodule
```

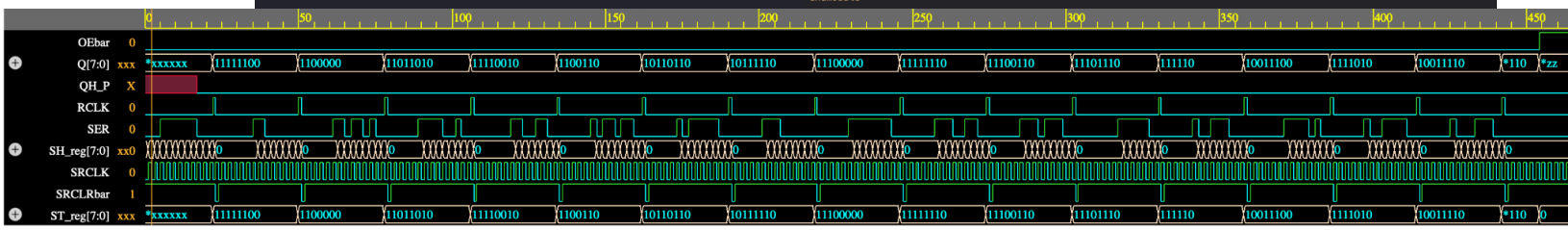
```
`include "project_5A.v"
`timescale 1s/100ms

module project_6_tb;
    reg SRCLK, RCLK, SER, SRCLRbar, OEbar;
    wire [7:0] SH_reg, ST_reg, Q;
    wire QH_P;
    integer i;

   SN74HC595 dut (
    .SRCLK(SRCLK),
    .RCLK(RCLK),
    .SER(SER),
    .SRCLRbar(SRCLRbar),
    .OEbar(OEbar),
    .SH_reg(SH_reg),
    .ST_reg(ST_reg),
    .Q(Q),
    .QH_P(QH_P));

    initial begin
        SER <= 0;
        SRCLRbar <= 0;
        OEbar <= 0;
        RCLK <= 0;
        #2;
        SER <= 1;
        SRCLRbar <= 1;
        #2;
        SER <= 0;
        RCLK <= 1;
        #1;
        RCLK <= 0;
        #1;
        RCLK <= 1;
        for(i = 0; i < 7; i = i + 1) begin
            #1 RCLK <= 0;
            #1 RCLK <= 1;
        end
        #1;
        RCLK <= 0;
        OEbar <= 1;
        #1;
        SER <= 1;
        #2;
        SER <= 0;
        RCLK <= 1;
        #1;
        RCLK <= 0;
        OEbar <= 0;
        #4;
        SRCLRbar <= 0;
        #1;
        RCLK <= 1;
        #1;
        RCLK = 0;
        SRCLRbar <= 1;
        #2;
        $finish();
    end
    initial begin
        $dumpfile("project_5A.vcd");
        $dumpvars(1,SRCLK, SER, RCLK, SRCLRbar, OEbar, SH_reg, ST_reg, Q, QH_P);
    end

    always #1 SRCLK = ~SRCLK;
    initial SRCLK = 0;
endmodule
```

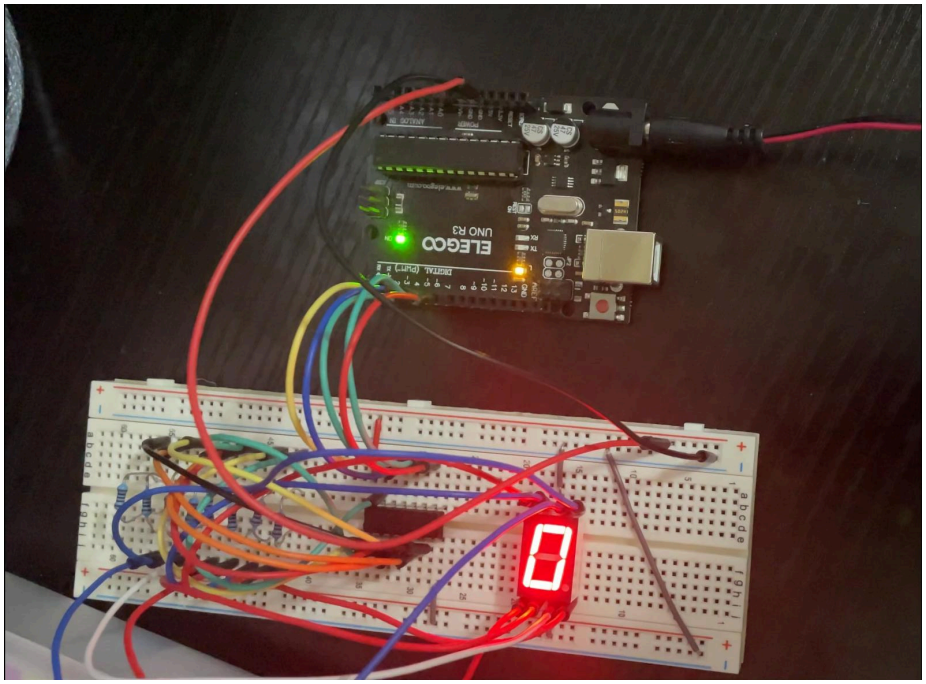


Verilog 5B Code & Results.....

```
module SN74HC595 (
    input SRCLK,
    input RCLK,
    input SER,
    input SRCLRbar,
    input OEbar,
    output [7:0] SH_reg,
    output [7:0] ST_reg,
    output [7:0] Q,
    output QH_P);

    reg [7:0] SH_reg, ST_reg, Q;
    reg QH_P;
    integer i;

    // Shift Register
    always @(posedge SRCLK) begin

        if(SRCLK) begin
            QH_P <= SH_reg[7];
            SH_reg[7:0] <= {SH_reg[6:0], SER};
        end

        if(!SRCLRbar) begin
        SH_reg[7:0] <= 8'b00000000;
        QH_P <= 0;
        end
    end

    always @(posedge RCLK or OEbar) begin
        ST_reg[7:0] <= SH_reg[7:0];
        if(!OEbar) begin
            Q[7:0] <= SH_reg[7:0];
        end
        else begin
            Q[7:0] <= 8'bzzzzzzzz;
        end
    end
endmodule
```

```
`timescale 1ns/1ns
module project_5_tb;
    reg SRCLK, RCLK, SER, SRCLRbar, OEbar;
    wire [7:0] SH_reg, ST_reg, Q;
    wire QH_P;
    integer i, j;
    reg [7:0] hex_values [0:15];

    SN74HC595 dut (
        .SRCLK(SRCLK),
        .RCLK(RCLK),
        .SER(SER),
        .SRCLRbar(SRCLRbar),
        .OEbar(OEbar),
        .SH_reg(SH_reg),
        .ST_reg(ST_reg),
        .Q(Q),
        .QH_P(QH_P)
    );

    initial begin
      hex_values[0] = 8'b11111100; // 0
      hex_values[1] = 8'b01100000; // 1
      hex_values[2] = 8'b11011010; // 2
      hex_values[3] = 8'b11110010; // 3
      hex_values[4] = 8'b01100110; // 4
      hex_values[5] = 8'b10110110; // 5
      hex_values[6] = 8'b10111110; // 6
      hex_values[7] = 8'b11100000; // 7
      hex_values[8] = 8'b11111110; // 8
      hex_values[9] = 8'b11100110; // 9
      hex_values[10] = 8'b11101110; // A
      hex_values[11] = 8'b00111110; // B
      hex_values[12] = 8'b10011100; // C
      hex_values[13] = 8'b01111010; // D
      hex_values[14] = 8'b10011110; // E
      hex_values[15] = 8'b10001110; // F
    end

    initial begin

        SRCLK <= 0;
        RCLK <= 0;
        SER <= 0;
        SRCLRbar <= 1;
        OEbar <= 0;
        #5;

        for (j = 0; j < 16; j = j + 1) begin
            for(i = 7; i >= 0; i = i - 1) begin
              SER <= hex_values[j][i];
                #1 SRCLK <= 0;
                #1 SRCLK <= 1;
            end

            #1 RCLK <= 1;
            #1 RCLK <= 0;
            #10; // Wait some time before moving on to the next value
        end

        #1;
        OEbar <= 1;
        #10;
        $finish();
    end

    initial begin
        $dumpfile("project_5B.vcd");
        $dumpvars(1,SRCLK, SER, RCLK, SRCLRbar, OEbar, SH_reg, ST_reg, Q, QH_P);
    end

always #1 SRCLK = ~SRCLK;
always #1 SRCLRbar = ~RCLK;
initial SRCLK = 0;

endmodule
```



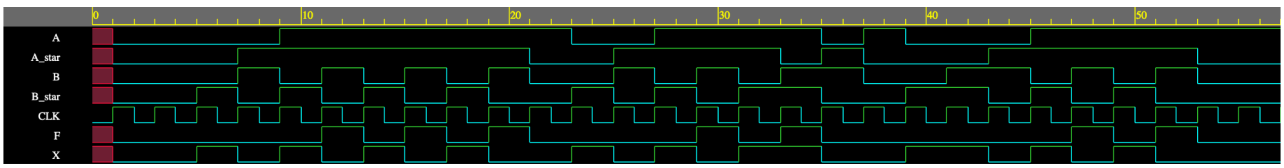


Verilog Code.....

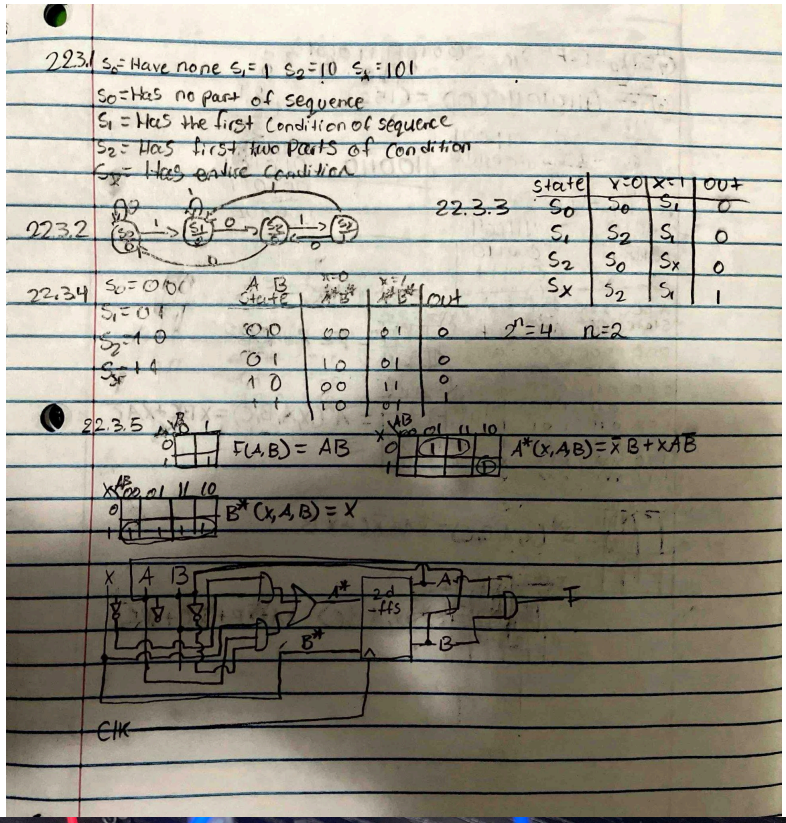
```
`include "fsm.v"
`timescale 1ns/1ns

module Sequence_Detector_tb;
  reg[0:27] vector ='b00101010100101011100110101000;
  reg X, A, B, CLK;
  wire A_star, B_star, F;
  integer i;

  Sequence_Detector uut(CLK, X, A, B, A_star, B_star, F);

  initial begin
    #1;
    X = vector[0];
    A = 0;
    B = 0;
    #2;

    for (i=1; i<27; i = i +1) begin
      X = vector[i];
      A = A_star;
      B = B_star;
      #2;
    end
    #2
    $finish();
  end

  initial begin
    $dumpfile("FSM.vcd");
    $dumpvars(1,F,A,B,X,A_star,B_star,CLK);
  end

  always #1 CLK = ~CLK;
  initial CLK = 0;
endmodule
```

```
module Sequence_Detector(
    input wire CLK,
    input wire X,
    input wire A,
    input wire B,
    output A_star,
    output B_star,
    output F);

  reg A_star, B_star, F;


  always @(posedge CLK) begin
    A_star = (~X & B) | (X & A & (~B));
    B_star = (X);
    F = A & B;
  end

endmodule
```

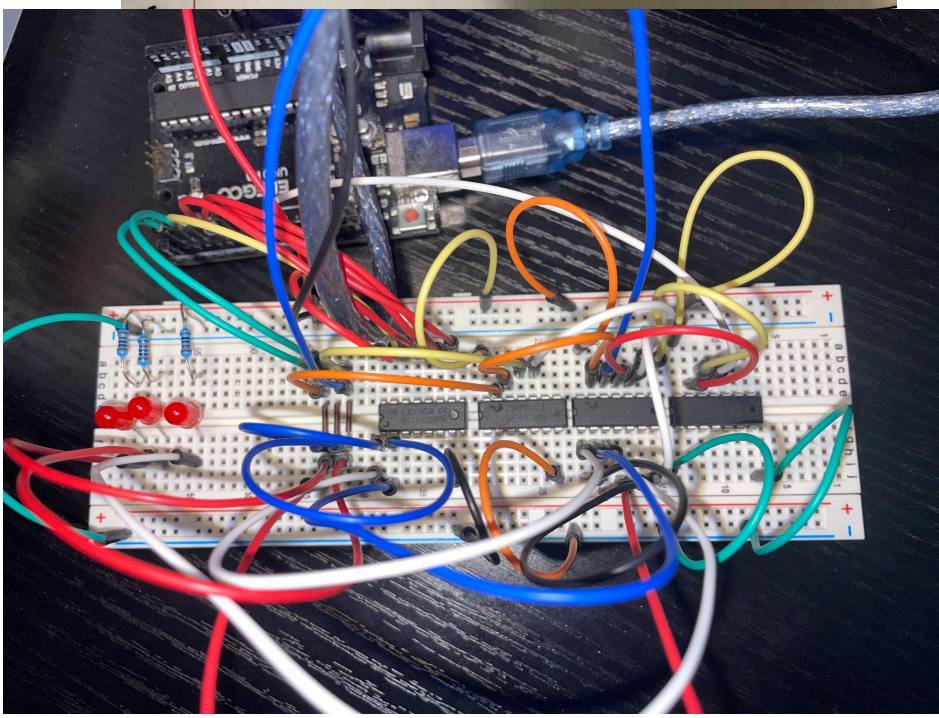
FSM Moore Waveform.....



Equations & Tables & Graphs……..



Arduino Circuit……..



Verilog Code……

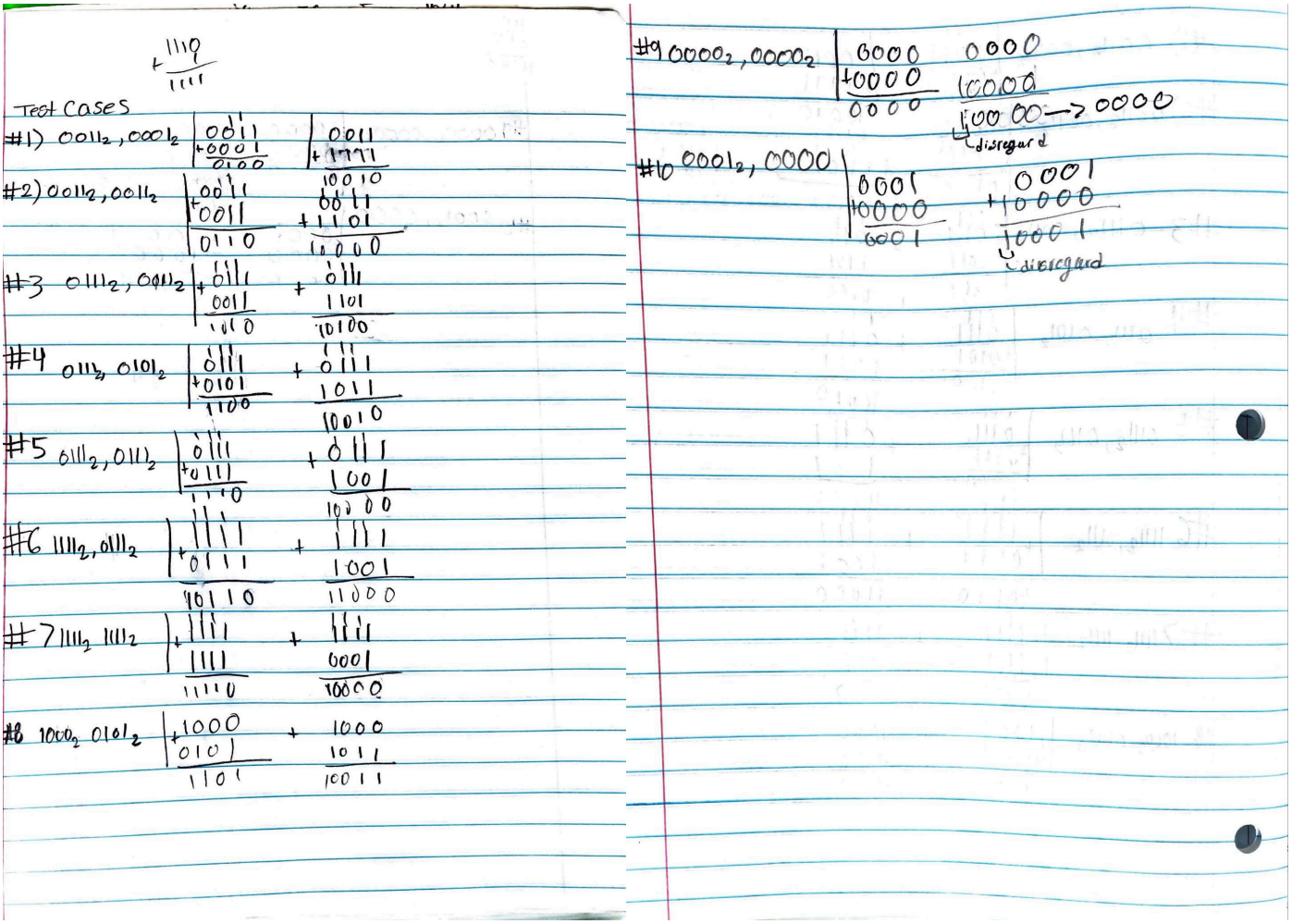
```
module ripple_carry_adder_tb;
  reg [3:0] A;
  reg [3:0] B;
  reg CIN;
  wire [3:0]S;
  wire COUT;

ripple_carry_adder dut(A, B, CIN, S, COUT);

initial begin
A <= 'b1111;
B <= 'b0001;
CIN <= 0;
#1;
$display("A = %4b, B = %4b, CIN = %b, S=%4b, COUT = %b", A, B, CIN, S, COUT);
$finish();
end

endmodule


module ripple_carry_sub_tb;
  reg [3:0] A;
  reg [3:0] B;
  reg SUB;
  wire [3:0]S;
  wire OVRF;

  ripple_carry_sub dut(A, B, SUB, S, OVRF);


  initial begin
     A <= 'b0000;
     B <= 'b0000;
     SUB <= 1; // Change to toggle between Add or Sub
     #1;
     $display("A = %4b, B = %4b, SUB = %b, S=%4b, OVRF = %b", A, B, SUB, S, OVRF);
     $finish();
  end

endmodule
```

```
module full_adder(A, B, CIN, S, COUT);
 input A;
 input B;
 input CIN;
 output S;
 output COUT;

 wire A, B, CIN, X, Y, S, COUT;

 assign X = A ^ B;
 assign Y = A && B;
 assign S = CIN ^ X;
 assign COUT = Y || (CIN && X);

 endmodule

 module ripple_carry(A, B, SUB, S, OVRF);
  input [3:0] A, B;
  input SUB;
  output [3:0] S;
  output OVRF;
  wire w1, w2, w3;

  full_adder adder0(A[0], B[0] ^ SUB, SUB, S[0], w1);
  full_adder adder1(A[1], B[1] ^ SUB, w1, S[1], w2);
  full_adder adder2(A[2], B[2] ^ SUB, w2, S[2], w3);
  full_adder adder3(A[3], B[3] ^ SUB, w3, S[3], OVRF);

endmodule
```

Verilog Output(test cases).....

```
[2024-03-15 02:14:50 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
A = 0011, B = 0001, SUB = 0, S=0100, OVRF = 0
A = 0011, B = 0001, SUB = 1, S=0010, OVRF = 1
Done
[2024-03-15 02:15:34 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
A = 0011, B = 0011, SUB = 0, S=0110, OVRF = 0
A = 0011, B = 0011, SUB = 1, S=0000, OVRF = 1
Done
[2024-03-15 02:16:01 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
A = 0111, B = 0011, SUB = 0, S=1010, OVRF = 0
A = 0111, B = 0011, SUB = 1, S=0100, OVRF = 1
Done
[2024-03-15 02:16:28 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
A = 0111, B = 0101, SUB = 0, S=1100, OVRF = 0
A = 0111, B = 0101, SUB = 1, S=0010, OVRF = 1
Done
[2024-03-15 02:16:50 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
A = 0111, B = 0111, SUB = 0, S=1110, OVRF = 0
A = 0111, B = 0111, SUB = 1, S=0000, OVRF = 1
Done
[2024-03-15 02:17:06 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
A = 1111, B = 0111, SUB = 0, S=0110, OVRF = 1
A = 1111, B = 0111, SUB = 1, S=1000, OVRF = 1
Done
[2024-03-15 02:17:33 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
A = 1111, B = 1111, SUB = 0, S=1110, OVRF = 1
A = 1111, B = 1111, SUB = 1, S=0000, OVRF = 1
Done
[2024-03-15 02:18:27 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
A = 1000, B = 0101, SUB = 0, S=1101, OVRF = 0
A = 1000, B = 0101, SUB = 1, S=0011, OVRF = 1
Done
[2024-03-15 02:19:52 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
A = 0000, B = 0000, SUB = 0, S=0000, OVRF = 0
A = 0000, B = 0000, SUB = 1, S=0000, OVRF = 1
Done
[2024-03-15 02:14:07 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
A = 0001, B = 0000, SUB = 0, S=0001, OVRF = 0
A = 0001, B = 0000, SUB = 1, S=0001, OVRF = 1
Done
```

Test Cases.....


Test Cases
#1) 0011₂, 0001₂
#2) 0011₂, 0011₂
#3 0111₂, 0011₂
#4 0111₂, 0101₂
#5 0111₂, 0111₂
#6 1111₂, 0111₂
#7 1111₂ 1111₂
#8 1000₂ 0101₂
#9 0000₂, 0000₂
disregard
#10 0001₂, 0000
disregard