R·I·T | Kate Gleason College of ENGINEERING

Engineering Lab Report

From: **Daniel Pak**

Date: **28 April 2023**

Subject: **Bit Serial Adder Subtractor FSM**

Abstract

This lab demonstrates the designing of a Bit Serial Adder Subtractor with an Accumulator System which consists of a Control Finite State Machine (FSM) for a Bit Serial Adder Subtractor with 5 Bit operands through Quartus. Quartus is a software developed by Intel, formerly known as Altera Corporation, which provides a complete set of tools for design entry, synthesis, simulation, and verification, as well as for programming and configuring FPGAs (Field Programmable Gate Arrays). First, the Finite State Machine (Figure 1: Finite State Machine Block Diagram) was made as it served as the "brain" of the whole circuits function. Inputs for this FSM were N, the data coming in, Reset, nADD, and Clk (Clock). Outputs were y [2..0], Done, Sh (Shift), and Sub (Subtraction).

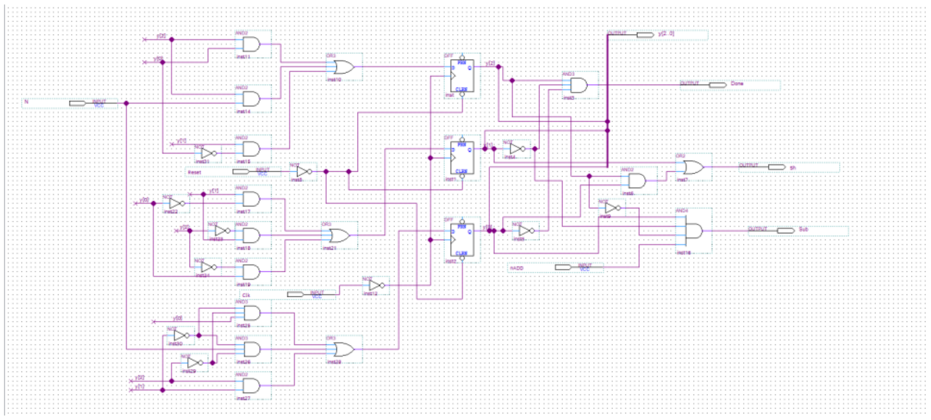

Figure 1: Finite State Machine Block Diagram

Next, Shift Registers were made for the X and Y operands. Since the outputs for the Finite State Machine feeds into the inputs for the X and Y Registers, the inputs of the Registers are Done, Shift, Clock, and Reset. The only additional inputs are the bussed inputs of each respective register, X [4..0] and Y[4..0], as well as the Si (Sum in). The X Register (Figure 2: X Register Block Diagram) and Y Register used 5 cascading D Flip Flops (DFF) with 5 corresponding 4 to 1 MUX gates to make the X operand. The same circuit was used for the Y Register (Figure 3: Y Register Block Diagram), apart from a XOR gate with inputs of a Subtract input and a Bus input of the Y operand (Y[4..0]). Its outputs are simply the bussed outputs, Xout [4..0, Yout [4 ..0], and its corresponding output from the least significant bit (LSB).

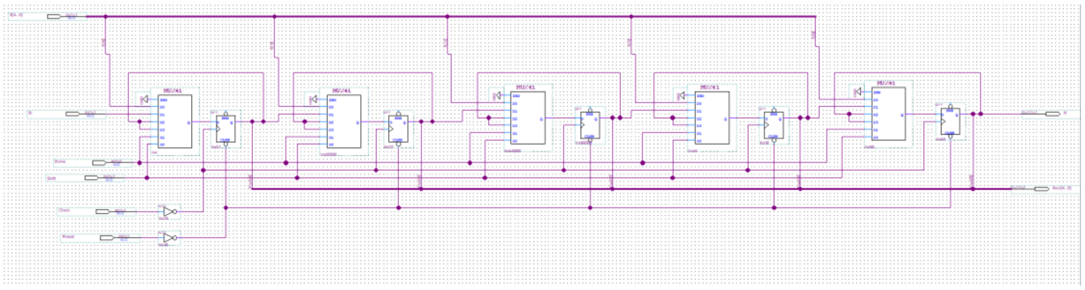
Figure 2: X Register Block Diagram

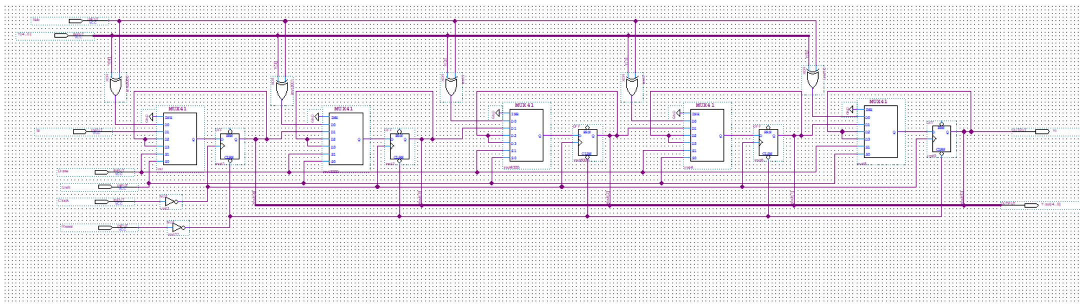
Figure 3: Y Register Block Diagram

Next, a Bit Serial Adder (Figure 4: Bit Serial Adder Block Diagram) was made using a full adder and D Flip Flop to then feed the sum of the X and Y operands to feed back into the most significant bit of the X Register, while the Y Register simply took the output and fed it back into its Most Significant Bit input. Additionally, the Cout (Carry Out) for the Full adder fed into the D while the Q fed into the Cin (Carry in). The D Flip Flop took the Done output from the FSM to feed into its Clear, while the Sub output for the FSM was fed into the Preset of the DFF. And finally, the clock and shift were NANDed together to then feed into the clock for the D Flip Flop. Also note that the Done and Sub outputs which went into the Preset and Reset were NOT gated. After synthesizing all components into one coherent circuit, the Bit Serial Adder Subtractor with Accumulator is complete (Figure 5) . All that was needed was the provided top level schematic of the circuit, to then use ModelSim, a waveform simulator which verifies the circuits' function. Results reflected a successful functioning Bit Serial Adder Subtractor with an Accumulator System with a Finite State Machine.

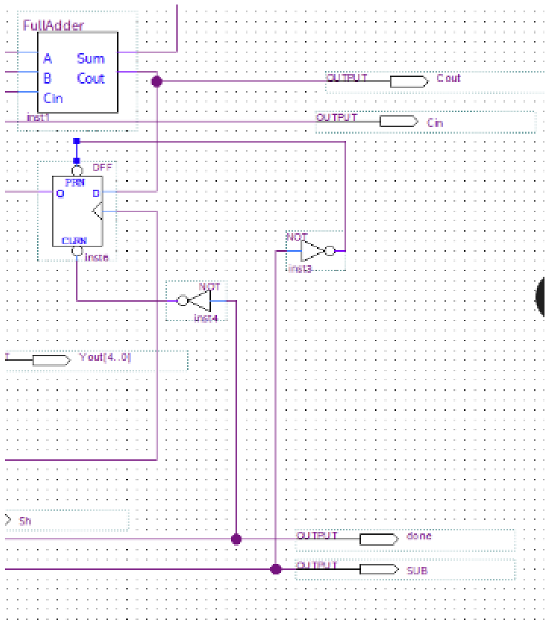

Figure 4: Bit Serial Adder Block Diagram

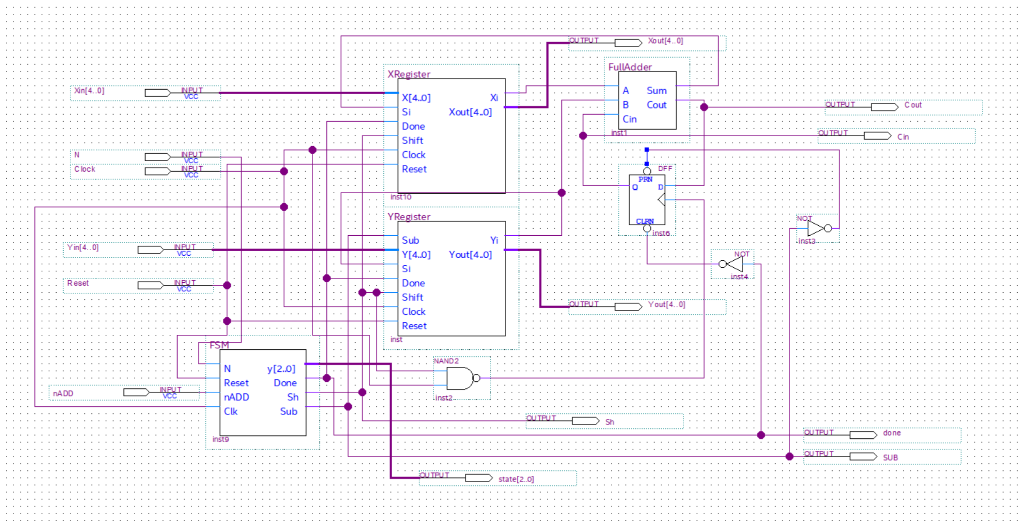

Figure 5: Bit Serial Adder Subtractor with Control Finite State Machine Block Diagram

Theory

In digital electronics, circuit gates are the building blocks for what makes a circuit. These gates take inputs and convert them into outputs based on what type of gate it is. Common gates include AND gate, OR gate, NOT gate, NAND gate, NOR gate, and XOR gate, these gates are the building blocks of digital circuits and can be combined in numerous ways to perform more complex logic functions. However, to understand how the logic gate functions, a truth table is used to graphically show it. The truth table lists all possible input combinations and the resulting output for a given logical function. The inputs and outputs are represented by binary digits, typically 0 for false or low and 1 for true or high. Table 10 shows a truth table for an AND gate. In this truth table, the first column represents the input A, the second column represents input B, and the third column represents the output of the AND gate. For each combination of inputs, the table shows the resulting output of the AND gate. The truth table shows that the output of the AND (Table 1) gate is only 1 when both inputs A and B are 1, otherwise, the output is 0. The OR (Table 2) gate Truth Table shows that when at least one of the inputs is 1, the output will result in a 1. A 0 0 input will result in a 0. A NOT (Table 3) gate takes in only one input and outputs the OPPOSITE. For example, an input of 0 will output a 1 and an input of 1 will output a zero. NOT gates can be used as logical negation, also known as inversion of logic. A NAND (Table 4: NAND Gate Truth Table) Gate stands for "NOT AND" which essentially uses a not gate in such a way that its "inverts" the output of the AND gate. In other words, it functions opposite of a traditional AND gate. The only combination of inputs in which an output of 0 is asserted is 1 1 for inputs A and B respectively. A NOR (Table 5: NOR Gate Truth Table) gate stands for "NOT OR" gate and again, it functions reverse to the OR gate where a NOT gate takes in the output of a NOR gate. Note how NAND and NOR gates both have proprietary circuits for costly benefits. A NOR gate will output a 0 for all combinations except for a 0 0 input combo. Finally, there is the (XOR Table 6: XOR Gate Truth Table) gate, which stands for an "Exclusive OR" gate. This logic gate will detect an output of 1 if and only if there is one value of 1 in either input. Contrary to the OR gate, where it detects an output of 1 when both inputs have a value of 1, the XOR gate will output a zero for that combination.

Table 1: AND Gate Truth Table

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

Table 2: OR Gate Truth Table

A	B	Output
0	0	0
0	1	1

1	0	1
1	1	1

Table 3: NOT Gate Truth Table

Input	Output
0	1
1	0

Table 4: NAND Gate Truth Table

A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

Table 5: NOR Gate Truth Table

A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

Table 6: XOR Gate Truth Table

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

The significance of the output is that it provides various ways to create circuits with different types of functions. For example, a D flip flop circuit. Based on the truth table and schematic, a D flip takes in two inputs: D (data) and “Clk” (Clock). In digital electronics, a clock is an electronic logic signal which oscillates between two states 1 (high) and 0 (low). Clocks can be seen as a “metronome” for a circuit as it operates at a constant frequency. For D Flip Flops, and other types of circuits, a clock can have either a “Rising Edge trigger” clock or “Falling Edge trigger” clock, where certain areas of the clock cycle will be activated and will assert an output. A rising edge will allow assertions ONLY when the clock is transitioning from a 0 to 1, while falling edge clocks only allow for assertions during a 1 to 0 transition of the clock. In a D flip flop, the “Q” or output line will transition from a low to high or high to low based on the “D” input state and Clock state. For example, when D has a 0 to 1 transition right after a rising edge clock cycle, Q output will not detect a change in D until the next Rising Edge is encountered. In simple terms, transitions in Q output will only be determined by the rising or falling edge clock cycle. D-Flip Flops are a very versatile component in electronics, with some applications of a D Flip Flop being Counters, Shift Registers, Finite State Machines, and Frequency Dividers.

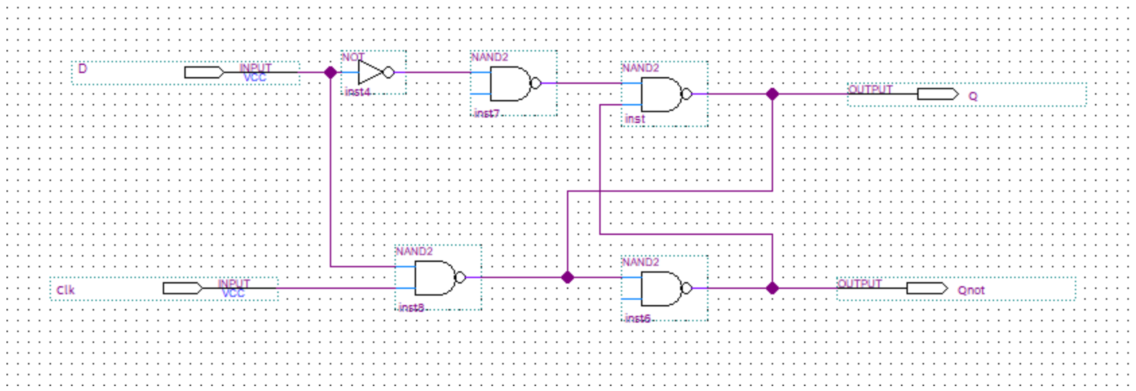

Figure 6: D Flip Flop Block Diagram Schematic

Table 7: D Flip Flop Truth Table

D	Clk	Q	Qnot
0	1	0	1
1	1	1	0
X	0	Q	Qnot

A finite state machine, in theory, is in one of “n” states at a given time and can transition from one state or another in response to different inputs. There are two types of state machines: Moore FSM and Mealy FSM. Although the two Finite State Machines function the same, the differences between the two lies in the number of states. A Moore Finite State Machine will feature one more state than a Mealy Finite State Machine because Mealy FSMs depend on both the current state and the inputs. This means that the output can change more frequently and can convey more info. The type of FSM used in the project was Moore. To make a Finite State Machine, a state diagram is necessary to get a general overview on how the state machine will function. This is made by drawing x number of cells which correspond to each state in the machine. In the case of the project, 8 states were made as cells. Figure 7: Finite State Machine State Diagram showcases the state diagram drawing, with nADD being relevant only in the second state. Each arrow represents a pathway to go to depending on if n (shift) is equal to a value of 1 or 0. In the case of the project, if n equals 1, B will be the next state. Then, nADD is taken in as either a 1 or 0. This is dependent on the FSM. nADD will tell the rest of the circuit whether to add or subtract on this FSM. After State B, the shifting will occur regardless of a 1 or 0 up until the last state H is met. Then when "Done" is equal to 1, the shifting will turn to zero. The state diagram then returns to its original state A.

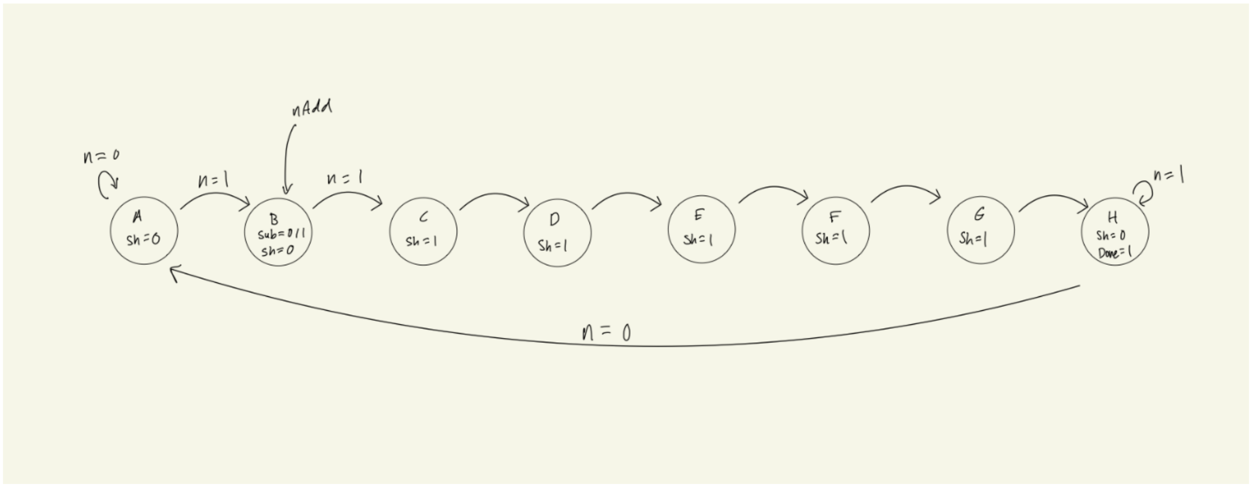

Figure 7: Finite State Machine State Diagram

Next, a State Table (Table 8) is made from the State Diagram to visualize the concept in table form. This is made by using the states from A to H and putting them in an array on the "Present" column. The next two columns are represented by "Next" state depending on whether the input is a 1 or 0. As previously mentioned, it won't matter what input is received after B because Sh is equal to 1, therefore both w = 0 and w = 1 have the same state up until H where Sh = 0. This can be verified by the next two column taken up by Shift. Additionally, the "Done" columns show a value of 1 on the last state, akin to the design made on the state diagram. Finally, there is "Sub" which is the output of nADD. On the state diagram, it showed an nADD arrow which was only relevant to state B. This is why when w = 1, it is only 1 on state B. On the contrary, when Sub is 0, no value of 1 will ever be detected. Next, the state assignment table is made (Table 9Table 9: Finite State Machine State Assignment Table). A state assignment table is essentially the State Table with 3-bit numbers replacing its corresponding state using encoding. In this case, grey code was used. For example, A is now represented by 000, and B is represented by 001.

Table 8: Finite State Machine State Table

Present	Next State		Shift		Done		Sub	
	w=0	w=1	w=0	w=1	w=0	w=1	w=0	w=1
A	A	B	0	0	0	0	0	0
B	C	C	0	0	0	0	0	1
C	D	D	1	1	0	0	0	0
D	E	E	1	1	0	0	0	0
E	F	F	1	1	0	0	0	0
F	G	G	1	1	0	0	0	0
G	H	H	1	1	0	0	0	0
H	A	H	0	0	1	1	0	0

Table 9: Finite State Machine State Assignment Table

Present	Next State		Shift		Done		Sub	
	w=0	w=1	w=0	w=1	w=0	w=1	w=0	w=1
y3 y2 y1	Y3 Y2 Y1	Y3 Y2 Y1						
0 0 0	0 0 0	0 0 1	0	0	0	0	0	0
0 0 1	0 1 1	0 1 1	0	0	0	0	0	1
0 1 1	0 1 0	0 1 0	1	1	0	0	0	0
0 1 0	1 1 0	1 1 0	1	1	0	0	0	0
1 1 0	1 1 1	1 1 1	1	1	0	0	0	0
1 1 1	1 0 1	1 0 1	1	1	0	0	0	0
1 0 1	1 0 0	1 0 0	1	1	0	0	0	0
1 0 0	0 0 0	1 0 0	0	0	1	1	0	0

Finally, Karnaugh Maps are made to get a derived equation to make circuitry for Next State, Shift, Done, and Sub. The purpose of making Karnaugh Maps is to simplify Boolean expressions and minimize the number of logic gates required in digital logic circuits. It also serves as an intuitive way to visualize how equations can be derived from truth tables in order to make circuits with certain logic gates. For example, let's look at a Karnaugh Map representative of the Done output (**Table 9**) and Shift output (**Table 10**). The way the two Karnaugh Maps were made was by using the which runs along the two ends of the box. Then, two inputs are represented by one number on each row or column. For example, for column 01, the 0 will be in respect to y3 while the 1 will be in respect to y2. With this, each present state is used in conjunction with the w input. For example, present state A or 000 with a w input of 1 will be assigned the leftmost cell on the 2nd row as that's how the table was made. A present state of 110 with a w of 1 will be on the 2nd row but on the 3rd column. To derive an equation from the Karnaugh map, group all the 1's values in either a 2, 4, 8, or 16 orientations. Overlapping is allowed in Karnaugh maps, as shown in Table 4. From this, an equation can be made. As shown in **Table 10: Karnaugh Map for Done Output**, the equation of $Y_3\overline{Y_2Y_1}$ was derived. The reasoning as to why Y2 and Y1 are "barred" is because for the high values that are on the Karnaugh map, those inputs take on low values. To account for this, the inputs are "barred", which symbolizes a "NOT" gate which is to be used in the circuit making. Additionally, it is evident that the w output was omitted from the final expression, and this is because w takes values of both 0 and 1, which means that that input does not affect the function of that specific output. In other words, the purpose of a Karnaugh map is to simplify gates through means of omitting certain inputs. Karnaugh Maps for Sub (**Table 12**), Y3 (**Table 13**) , Y2 (**Table 14**), and Y1 (**Table 15**) are shown below and show corresponding equations to be used in its circuitry.

Table 10: Karnaugh Map for Done Output

$y_1w \backslash y_3y_2$	00	01	11	10
00	0	0	0	1
01	0	0	0	1
11	0	0	0	0
10	0	0	0	0

$$Done = y_3\overline{y_2y_1}$$

Table 11: Karnaugh Map for Shift Output

$y_1w \backslash y_3y_2$	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	0	1	1	1
10				

0	1	1	1

$$Shift = y_2 + y_1 y_3$$

Table 12: Karnaugh Map for Sub Output

y_1 nAdd \ $y_3 y_2$	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	0	0	0
10	0	0	0	0

$$Sub = nAdd\,\overline{y_3\,y_2}\,y_1$$

Table 13: Karnaugh Map for Output Y3

y_1 w \ $y_3 y_2$	00	01	11	10
00	0	1	1	0
01	0	1	1	1
11	0	0	1	1
10	0	0	1	1

$$Y_3 = \overline{y_1}\,y_2 + y_3 w + y_1 y_3$$

Table 14: Karnaugh Map for Output Y2

$y_1 w \backslash y_3 y_2$	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	1	1	1	0
10	1	1	1	0

$$Y_2 = \overline{y_1} y_2 + y_3 \overline{y_2} + \overline{y_3} y_1$$

Table 15: Karnaugh Map for Output Y1

$y_1 w \backslash y_3 y_2$	00	01	11	10
00	0	0	1	0
01	1	0	1	0
11	1	0	1	0
10	1	0	1	0

$$Y_1 = \overline{y_2 y_3} w + \overline{y_2 y_3} y_1 + y_3 y_2$$

The benefit of using grey as opposed to binary is mainly the fact that grey code results in simpler equations, which leads to less logic gates to be used in the final circuit. The equations now may look somewhat complicated but with binary, the equations could have been 3 times more complex, which wastes time and makes the cost higher which can cause delays. Because the 1's were circled rather than the 0's, this Karnaugh Map has derived equations which use the Sum of Products, or SOP. A Sum of Products expression will cause the circuit to use more "AND" gates with the outputs feeding into an "OR" gate. Figure 1 showcases exactly that. On the left side of the circuit shows an array of "AND" and "NOT" gates being used depending on the derived equation, to then be "OR" gated. The "OR" gate output is then fed into its corresponding D Flip Flop. The output of the D flip flop then outputs a y1, y2, or y3 to feed into two things: 1. A bus output represented by y[2..0], and 2. The remaining outputs Done, Shift, and Clock. To verify the function of this circuit, all that needs to be done is to manually add signals to each input with clock having its clock cycle, of course.

Next, the Shift Registers for X and Y are made to serve as the Accumulator for the circuit. A Shift Register is a sequential data circuit that can store and shift binary data, consisting of a cascading number of D Flip Flops in which the output of the first DFF becomes the Input for the next one. This project had D Flip Flops which went from Most Significant Bit to Least. A Most Significant Bit (MSB) is the leftmost bit, having the highest magnitude of value. The Least Significant Bit (LSB) is the least in magnitude in terms of value, being 1. In binary computing, bits are expressed in the base 2 numeral system, using two digits 0 and 1. For instance, in a 5-bit number, assuming it is 11111, the value will be as follows. 2^4 + 2^3 + 2^2 + 2^1 +2^0, adding to 31. D Flip Flops (D standing for data) a type of circuit that can store a bit of data with two stable states, 0 and 1. It can be set to either state by applying a Clock signal to its input. The output of the D Flip Flop changes depending on the input D value at the rising edge of the clock signal. However, this project requires a falling edge clock cycle which is done by NOT gating the clock. While a traditional shift register only consists of a cascading number of D Flip Flops, this project required a 4 to 1 Multiplexer (MUX 41) to be attached to the D Flip Flop in such a way that the output of the MUX gate feeds into the input of the DFF. The MUX gate in this case serves as a LOAD, SHIFT, and DONE signal to be given into the D Flip Flops. Table 16 shows the function of the MUX gate in the context of this project. S1 and S0 both represent select lines and Y represents the output of the MUX gate, which goes into the D Flip Flop. S1 represents Done and S0 represents Shift. Essentially, when "Done" is 0 and Shift is 1, the output is shift. When "Done" is 1 and shift is 0, output done will be asserted. However, when both "Shift" and "Done" have values of 0, it is telling the MUX to "Load" in new bits, signaling to the MUX gate that if Shifting is not happening and "Done" is not asserted, signaling a finished job, it will Load in new bits.

Table 16: 4 to 1 MUX Table

S1	S0	Y
0	0	Load
0	1	Shift
1	0	Done
1	1	Done

To translate this logic in circuit form, the D2 and D3 lines were connected by the output of the D Flip Flop, which signals a "Done". D0 was done by wiring the input to the X[4..0] directly. Finally, D1 gets its signal from an input pin named Sh, which comes from the FSM. The difference between the X and Y registers mainly lies within the Load input. Instead of a direct wiring of the X[4..0] bus line to the input, the bus line is XOR gated with a Sub input, which was made on the FSM, to then be wired into the D1 input of the 4 to 1 MUX gate. In short, a XOR gate will output a 1 if the two inputs are different. When applying that logic to the Y Register, when the incoming bit is 0, Sub is 1, and when the incoming bit is 1, Sub is 0. XOR will output a zero when both inputs are 1 or 0. This is known as a two's complement arithmetic. In the two's complement subtraction method, the XOR gate is utilized to add the binary digits of the minuend and the two's complement of the subtrahend. This is done in a manner where the bits in the same positions are added without any carry from lower positions. However, in order to fulfill the functionality of a two's complement, a one needs to be added after inverting. This is done in the Bit Serial Adder, which will be later explained. To get the bits shifted in, however, the Bit Serial Adder is used to do this. The X Register output named Xi and Yi for the Y Register are inputted into the Full Adders A and B inputs. Because it is a "Serial" Adder, it adds bits one by one. The adder outputs a sum in which the output feeds into the Si (Shift In) of the X Register while the Yi feeds into the Si.

Finally, the Bit Serial Adder. As mentioned previously, the main functionality of this circuit is to "Serially" add bits from the X and Y Register one at a time. It adds bits through the Full Adder Component.

A full adder uses a series of XOR gates, AND gates, and OR gates. Figure 8 showcases the way in which a Full Adder is made. It takes in 3 inputs, A and B being the outputs of the X and Y Registers, and Carry in. The two outputs, Sum represents the sum of A and B, and carry out is the carry out of the addition. For example, if a 1 and 1 are being added, the sum will be zero, with a carry out of 1 and will be carried into the next. If a 1 and 1 are being added with the addition of a carry in of 1, the sum will be 1 with a carry out of 1. This explanation can be verified through the Full Adder Truth table shown in Table 17. The Block diagram shown in Figure 11 provides the exact logic that is being used to perform addition. For example, A is 0 and B is 1. Based on previously explained theory on a XOR gate, it will only output a 1 with a combination of 0 1 or 1 0. In this case it will output a 1. A Carry in of 0 will be asserted since this is the first instance of addition. The sum will be equal to 1 since a 1 from the A B XOR gate and 0 from input Carry in are being used. The Carry out will also be zero because both AND gates will output a zero because no 1 1 combination was detected.

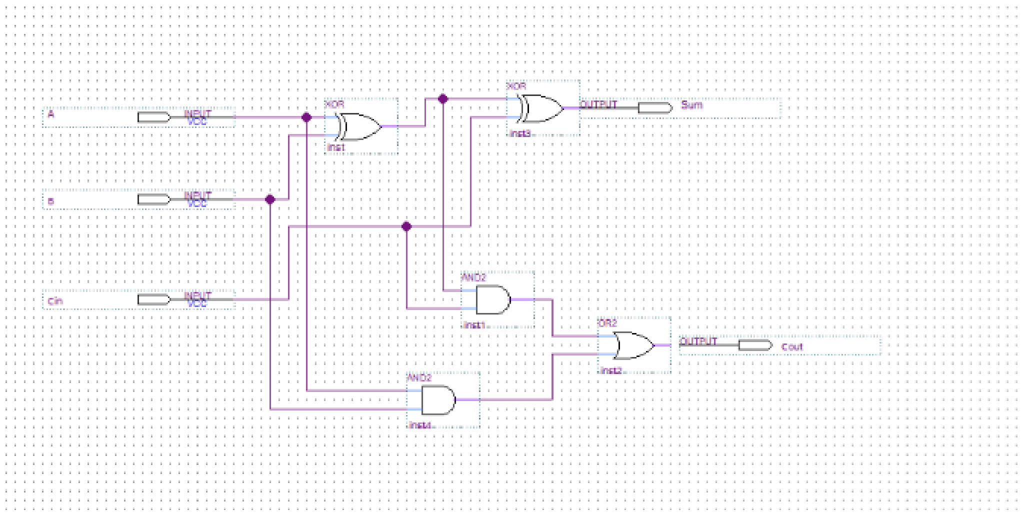

Figure 8: Full Adder Block Diagram

Table 17: Full Adder Truth Table

Inputs			Outputs	
A	B	Carry In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

In the context of the project, the full adder is just a component to a Bit Serial Adder, which behaves differently in the fact that a DFF is used in conjunction to the Full Adder. This is done to account for the potential subtractions that can happen due to the nADD/Sub signal from the Finite State Machine. However, subtraction is not explicitly done in digital electronics. Instead, a 2's complement addition is done. This is when the subtracted bit is inverted, and 1 is added to the least significant bit after. The full adder produces a sum and a carry out as output. However, the added one after inversion cannot be done by the full adder. This is where the D Flip Flop is used, using the reset, and preset of the DFF is to use Sub and Done outputs as the inputs respectively. The output of the Sub signal is NOT gated into the Clear. This will always result in a 1 when sub is 1. This is because the preset signal has a built-in inverter. Additionally, the done signal can be asserted through the Clear input for the D flip flop. Again, the wire line is inverted to get the intended value. Because of the negative edge triggered clock, the DFF takes a NAND gated output with inputs of Shift and Clock, to ensure that both inputs are synchronized.

Results and Discussion

Results for this lab assignment involve the use of Waveform Diagrams. Waveforms are used to verify the function of a circuit because they provide a visual representation of the electrical signals within the circuit. A waveform is a graph of the voltage or current over a period of time (nanoseconds in this case), and it can be used to analyze various aspects of the circuit's behavior, such as its frequency, amplitude, and rising and falling triggers. By comparing the expected waveform with the actual waveform produced by the circuit, engineers can determine if the circuit is functioning properly.

Waveform analysis is widely used in electrical engineering, particularly in the design and testing of electronic circuits, as it can provide valuable insights into the circuit's behavior. It can determine if the circuit is generating the expected output signals and detect any undesirable oscillations or distortions in the signal. Additionally, waveform analysis can assist in identifying and resolving issues with the circuit, which is why it is an essential technique for circuit testing and verification.

In Quartus, Waveform Diagrams are provided under the name “VWF Waveform Diagram”. Before a Waveform Diagram can be made for a specific block diagram, that specific block diagram must be “compiled” and set as “Top Level Entity”. It essentially lets Quartus know that the specific block diagram is what should be relevant regarding verification and testing. Waveform Diagrams are made through right clocking the left side margin and using the “Node Finder”. There will be all inputs and outputs for the top-level entity project (the most recently compiled file). Once all inputs and outputs are shown on the left side margin, specific signals are given to each individual input. For example, Figure 8 shows a clock cycle input being asserted fixed intervals of 25 nanoseconds, while N and nADD inputs are given the same behavior. Once all inputs are given signals, testing and verification can be done.

Based on the waveform for the Finite State Machine, shown in Figure 8, when nADD has a value of 1, SUB will have an output value of 1 ONLY during the second state. Additionally, the Y outputs will have 5 instances of shifting after the Sub until the Shift output is low. The Done signal is then activated, and all bits are reset to 0. This behavior shown on the experimental waveform behaves identically to the State Assignment Table (Table 9: Finite State Machine State Assignment Table) as well as the State Diagram for the Finite State Machine (Figure 7). Results show that the Finite State machine is successful in asserting correct Sub values in response to nADD in correct states, as well as shifting the correct number of times before Shift is asserted to 0, while “Done” is shifted to 1 in the next clock cycle.

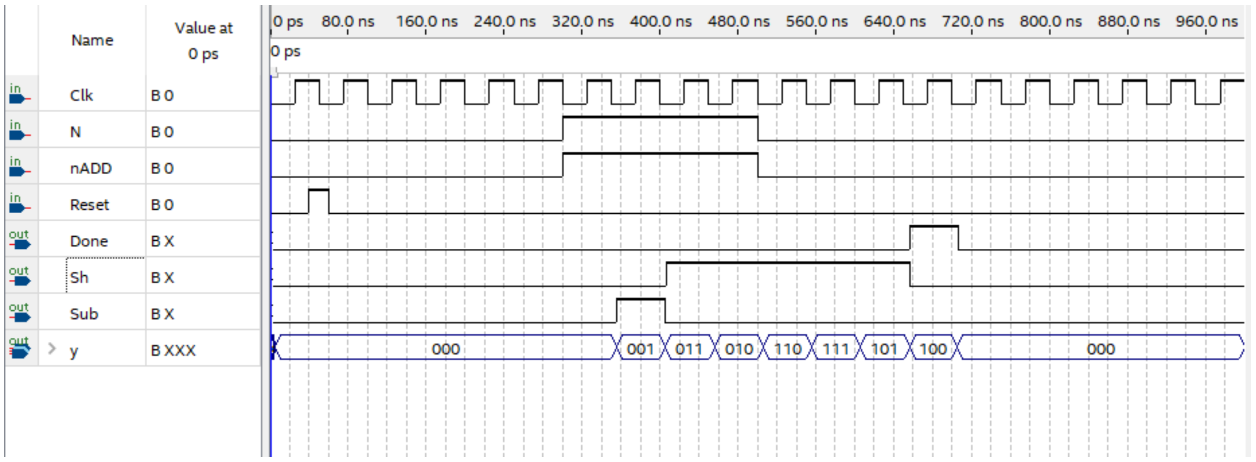

Figure 9: Finite State Machine Waveform

Based on the theory of the Y Register (Table 16: 4 to 1 MUX Table), it shows that when both inputs Sub and Done are 0, the Y Register should constantly load in bits. Additionally, if Shift is 1 and Done is 0, simply shift the bits in a way that the value of the least significant bit gets moved to the most significant bit, shifting every bit to the right 1 unit. Finally, when Done is asserted, do not Shift or Load. Essentially, freeze the register until done is low. This behavior can be shown in Figure 10, where new bits are constantly being given by the input Y up until Shift is 1 and Done is 0. Once Done is asserted 1, the Yout value is frozen at 00010 until Done transitions from 1 to 0. Then, new bits are loaded in. Additionally, Yi is functioning as theorized, taking the value of the least significant bit in Y out and asserting that value as its Yi. Subsequently, when shifting, the value of the least significant bit now becomes the most significant bit. Finally, results show that the Y Register is successful in performing a correct inversion and adding 1 to fulfill the function of a two's complement arithmetic when SUB input Is asserted 1.

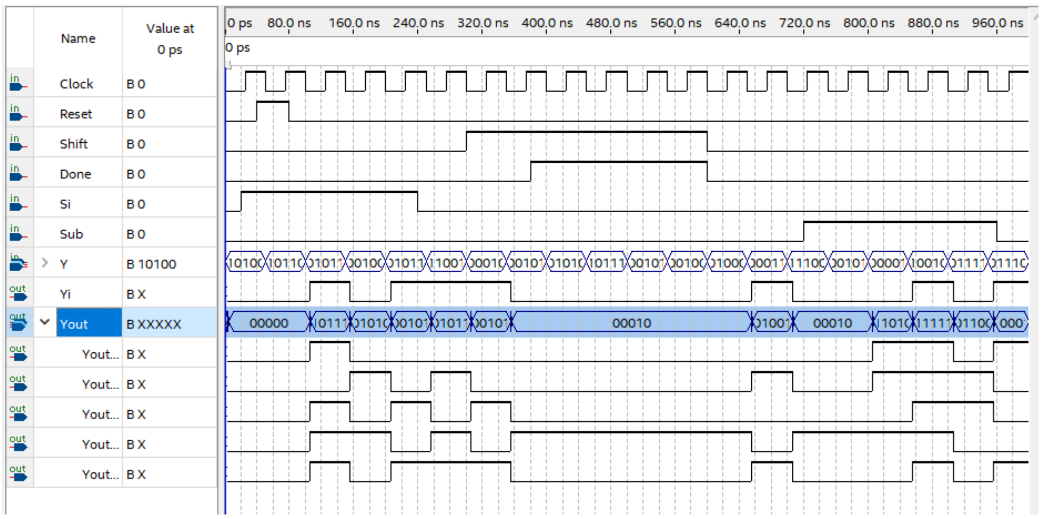

Figure 10: Y Register Waveform

Similar to the Y Register, the X Register also takes in 5-bit operands and has an Si, Clock, Shift, and Reset inputs with outputs being X, Xi, and X out, with the only input "SUB", thus the absence of a XOR gate altogether. The same theory applies to the X Resister, with the only functional difference being that when shifting, the most significant will depend on the value of Sum of the Full Adder rather than Yi in the Y Register. Because the Full Adder Is not included in this test, manual values will be given to Si. Results show that correct bits are being loaded in when Shift and Done are 0. Additionally, When Shift is High and Done is 0, correct shifting occurs with the values of Si being inputted into the X Registers' most significant bit. Finally, values of the Xi waveform is correct in showing its dependance on the Xout's Least Significant Bit, asserting a 1 when the Xout LSB is 1 and asserting a 0 when Xout LSB is 0.

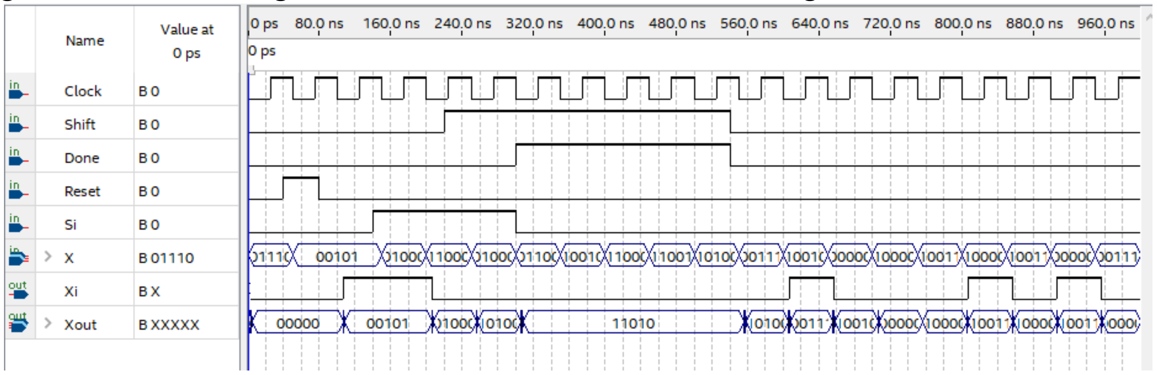

Figure 11: X Register Waveform

Finally, a software within Quartus called "ModelSim", is used to verify the whole circuit's function. This ModelSim waveform (Figure 12) is only possible through compiling the provided top level schematic for the file, as it correlates with the coded Model Sim behavior. The main waveform to pay attention to in this case is Equal and Done. If both waveforms have an identical pattern, that is enough to verify that the whole circuit is functioning as theorized. Results show that both Equal and Done are both identical in pattern, thus confirming that Finite State Machine is providing correct information to the X and Y registers. Additionally, the MUX gates are functioning as told, shifting when only Shift Is asserted 1, performing Done when at least Done is asserted 1, and loading in new bits when both Shift and Done are 0. Additionally, when Sub outputs a 1, the Bit Serial Adder correctly inverts all bits in the Y Register and presets a 1 to account for the 2's complement arithmetic. Finally, X is successfully taking in the Sum of Both X and Y and carrying that value over to the Most Significant Bit when shifting.

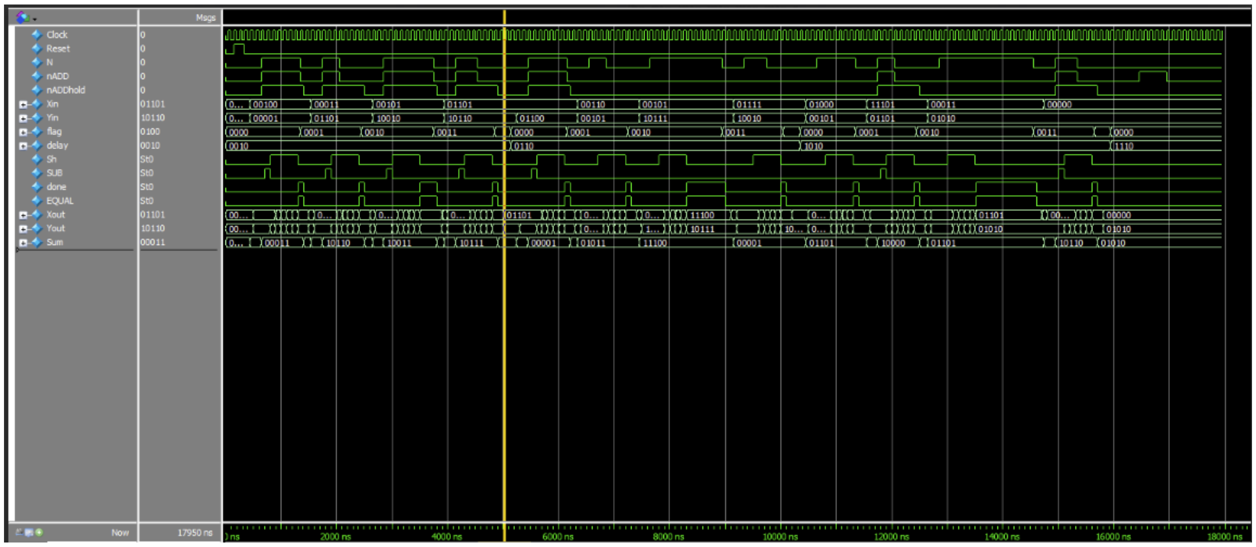

Figure 12: ModelSim Waveform Simulation

Conclusions

The objective of this assignment is to synthesize all topics learned throughout the semester to create a cohesive circuit that uses performs correct Addition and Subtraction given the outputs of the Finite State Machine. From the basics of simple gates to complex state machines and serial adders, this project required the knowledge and application skills to create a project of this magnitude. The approach taken to complete the assignment was to first start with the Finite State Machine, as it is considered the main point of this project. Additionally, when completing the Finite State Machine, it provides as the brain for every other component, giving a better visualization and conceptualization on how all the components will function into one complete circuit. In conclusion, this project provided an invaluable experience in creating a complex circuit much more advanced than previous assignments, using all concepts learned in Digital systems I to create a practical circuit which adds and subtracts 5 bit operands using Boolean Algebra.