Lab 9 Report

Kiran Nadkarni

Elizabeth

11/1/19

Hailstone Sequence

Objectives:

Our goal in this lab is to be able to understand and implement the Hailstone number sequence in LogicWorks using only two 4-bit adder chips and two Quadruple multiplexer chips. We want to correctly be able to represent the two specific features of the Hailstone sequence to be able to create a circuit that, when we input a number, will tell us the next number for that specific hailstone sequence.

Introduction:

The hailstone sequence is a sequence of numbers that has two main characteristics as previously mentioned. To generate the sequence, you have to know if the number you are currently working with is even or odd. If the number is even, it must then be divided by two, if the number is odd it must then be multiplied by three and then have a 1 added to it. The sequence will always end up decreasing, and usually starts by increasing as well, which is the inspiration for the sequence's name.

Implementing this in LogicWorks is somewhat tricky, and because our operation changes due to what input number we are given, we use the multiplexers to switch between operations. We also are using the adders to multiply the input too. The division operation is taken care of manually, with us just needing to shift the bits down by 1 because that is a convenient way to represent division in binary. Aside from this and the multiplication, the other operation that we need to perform is the addition of 1 after multiplying an input by 3. Again, because we are using binary this is fairly simple because we are using binary. Because this addition only happens for odd numbers, we know that all odd numbers in binary end with a 1. This means that we can just use this last bit to determine if the number is odd or even to perform the division or multiplication and addition of 1.

The specific chips we will be using are the 7483 Adder chips and the 74157 Quad-multiplexer chips. We have previously worked with this same adder chip before and the operation is quite simple so I will not go into detail explaining it, but the chip adds two 4-bit binary numbers and includes another input for a bit that might be carried into the addition as well as the 4 output bits and a 5^{th} carry-out bit which will only function if an output is big enough. We have also used multiplexers before, but this chip is a little bit more complex. These multiplexers will support 4 bits each, and will have two inputs and a single output for each of those bits (8 inputs total). There is a selector input as well to choose between these two inputs, which is what makes this a multiplexer. As mentioned before, we are using 2 of each of these sets of chips because we want to represent numbers larger than 15 in this hailstone sequence, so doubling the amount of chips will result in twice the amount of output bits, allowing us to calculate much larger numbers.

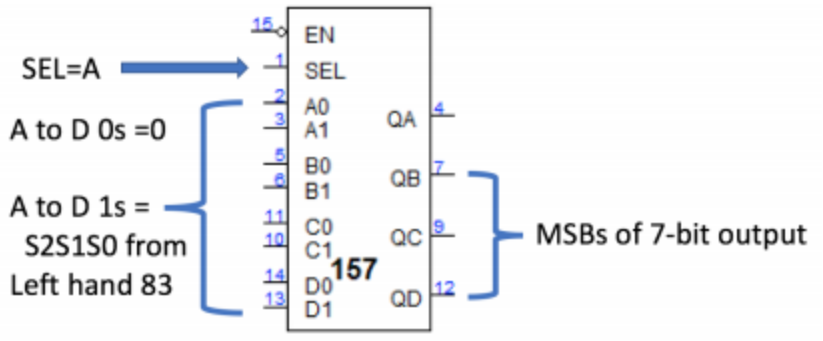

Figure 1: this is the diagram of the 157 chip. Notice the 4 bit output as well as the two 4-bit inputs and the selector input.

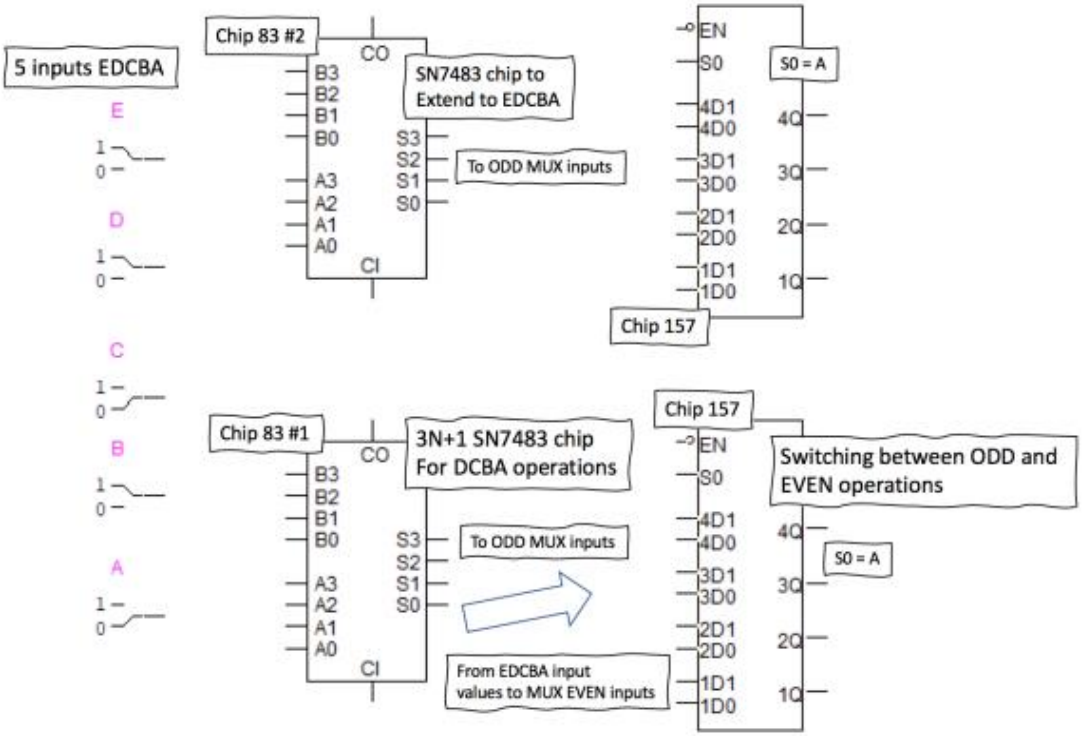

Figure 2: This is the diagram of every circuit component except the binary switches. We are restricted to using two adders and two MUXs, as well as the 5 input bits because more than that would exceed 8 output bits.

Procedure:

1- We first want to set up our inputs. We are going to use 5 bits, EDCBA to represent our input, which will give us a maximum input value of 31 in decimal. We will set up

these 5 bits, and then look to attach them to our first chips, the adders. Since we are multiplying by 3 and adding 1, we first attach the 4-bit adders to each other, through one's carry-out to the others carry-in. this allows us to represent a large output. We then take the rightmost set of 4-bit inputs A3-A0 and attach DCBA inputs to them. The free carry-in input will be attached to Vcc here because we are always adding 1 to the number we multiply. The second set of inputs on this first adder B3-B0, are shifted over to the left 1 bit with the least significant bit grounded. We do this to represent the binary multiplication, where we are adding the number to itself, so we need to shift it up one to represent the second partial product.

2- The second adder is also fairly straightforward. This is supposed to finish our representation of the multiplication by 3, so we just attach the D switch to the second circuit's B0 input and the E switch to the second circuit's A0 and B1 inputs. We then ground the other 5 inputs for the same representation of the previous ground. Our outputs and inputs to the MUX will reflect only one of the possibilities of the either even or odd inputs operations.

3- For our first set of inputs to the MUXs, we will be using the 0 inputs to represent the multiplication operation and the 1 input to represent the division. For the multiplication, we will have a 1 to 1 map for both the right and left MUX to the right and left adders, where A0, B0, C0, D0 for both MUXs are all taken up by adder outputs. The EN inputs for both MUX and A1B1C1D1 inputs for the left MUX are all grounded too, because the divided number will not need this second MUX. The A1-D1 inputs for the right MUX are also 1 to 1 mapped to the E-B input switches from the top of the circuit. We do not include the A switch because of the division, where we just shift the set of bits to the right and "delete" the A bit.

4- The SEL inputs to the MUX are determined by the value of the A input bit, so we connect them up to there. I also included a NOT gate between the A bit and the SEL connection because I made this connection after choosing to use 0 input for the multiplication and 1 input for the division on the MUX, if the reverse assignments were made, the NOT gate would not be required.

5- We finally connect all of the outputs of the MUXs to binary probes and these eight probes will display the second part of this sequence for us. This circuit is easy to test by just inputting number into the hailstone sequence and comparing the circuit's result to a hand calculated result.

Results:

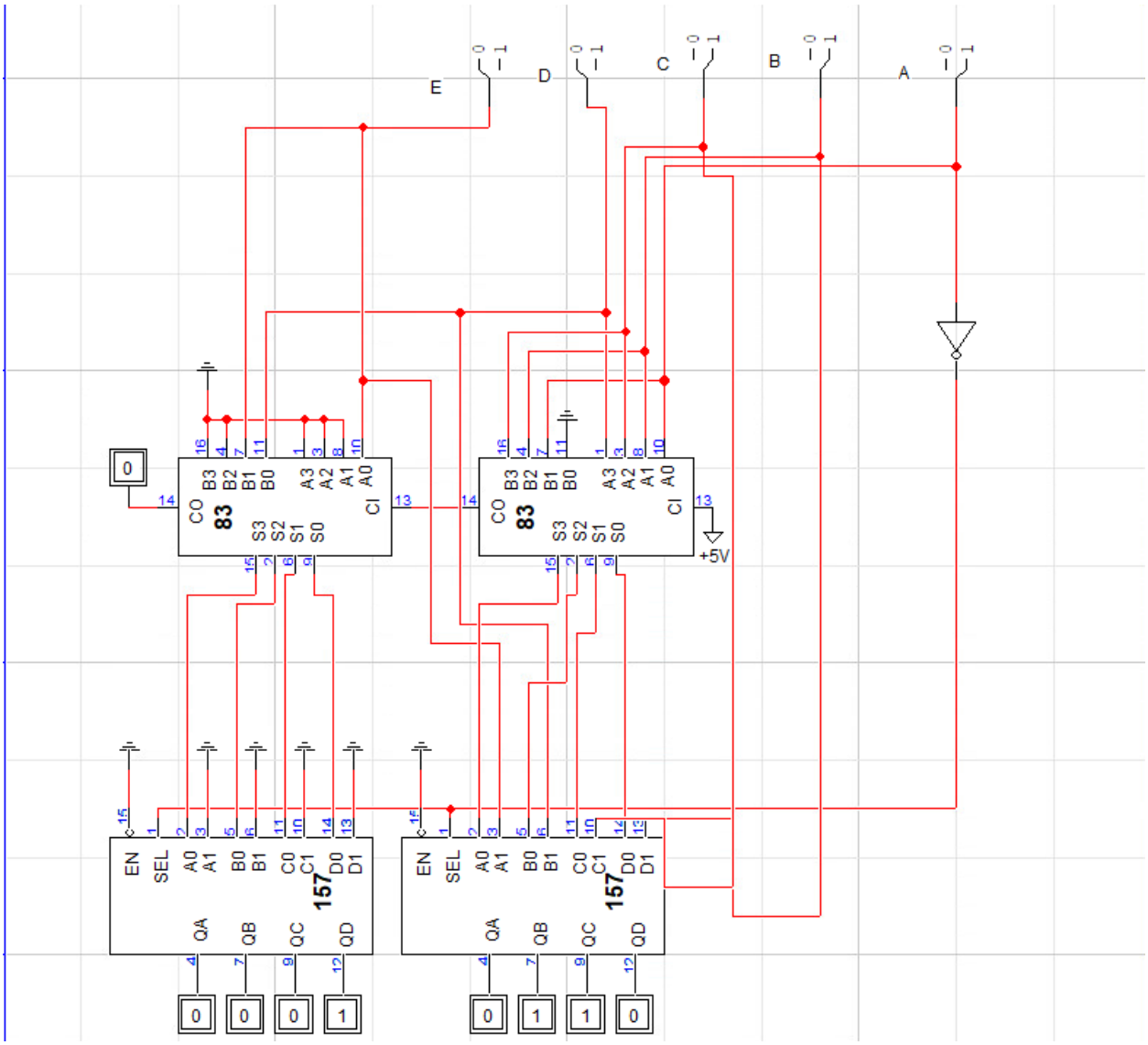

Figure 3: This is the diagram of my completed circuit. The 8 output bits are on the bottom, with the MSB on the left as always. The switches are labeled in the same fashion with the MSB on the left. The NOT gate attached to the A switch is from my own wiring mistake which I fixed with the gate instead of rewiring many of the 157 inputs.

E	D	C	B	A	Output	Even or Odd
0	0	0	0	0	0	
0	0	0	0	1	100	Odd
0	0	0	1	0	1	Even
0	1	0	0	1	11100	Odd
0	1	1	0	0	110	Even
1	0	0	0	0	1000	Even
1	0	0	0	1	110100	Odd
1	1	1	1	1	1011110	Odd
1	1	1	1	0	1111	Even
1	0	1	1	0	1011	Even
1	1	0	0	1	1001100	Odd

Figure 4: This is the Data from the circuit, as you can see the odd values are all tripled and have 1 added to them, while the even values are divided/shifted down a bit in their output.

Discussion:

My results were very accurate, and it was easy to check them as I mentioned before, due to the simple formula to continue the sequence. I also think that because of the variety of possible numbers I could have found, I made a good decision by trying to find values around the bottom, middle and high end of possible values for the output of the circuit. This allowed me to gauge the effectiveness of my circuit without having to test it exhaustively, since there would be 2^5 possible input combinations.

Conclusion:

This lab tested my ability to perform arithmetic operations on a string of bits, specifically the operations of multiplication and division, which are not able to be explicitly done with a simple adder setup. We also had the equivalent of a conditional statement where the input value was tested to be odd or even and depending on the result, this input went through a different operation. The specific 157 MUX that we used was very interesting and a chip that I know I will find useful in future logic endeavors because of how straightforward its functionality is, despite how daunting the number of inputs may seem. One specific thing that I regret in this lab is my choices for the 0 and 1 inputs on the MUXs. By choosing the 0 input for multiplication and the 1 input for division, I made it so that when a number is even, the number will be multiplied, which is incorrect. To fix this, as previously mentioned, I added a NOT gate to reverse the input signal, instead of rewiring everything which would take significantly more time.

Question:

To continuously generate the Hailstone sequence, I would need to add a similar circuit on the output of the MUXs. I would need to expand the circuit to accommodate more input bits, because there could be up to 7 based upon my input, and if I was to continue it further from there, I would need to add even more chips on further layers to accommodate those numbers, up until the number came “crashing down” as the hailstone sequence tends to do.