Centrale Nantes

MAC : 1^{st} Lab's Report
Build an 8-bit ALU with Logisim

1^{st} year Embedded Systems Engineering

By

EL KHAYDER Zakaria

SAOUTI Rayan

Professor

BRIDAY Mikael

October 22, 2023

Session

2023-2024

Made with LaTeX

Contents

Contents **I**

List of Figures **III**

Table of Tables **III**

1 Adder **1**

1.1 1-Bit Adder 1

1.1.1 Truth Table 1

1.1.2 Karnaugh Map 1

1.1.3 Equations 1

1.1.4 Circuit 2

1.2 8-Bit Ripple-Carry Adder 3

1.2.1 Logic 3

1.2.2 Circuit 3

2 ALU **4**

2.1 Operations 4

2.1.1 Output Control / Selected Operation 4

2.1.2 Addition 5

2.1.3 Subtraction 6

2.1.4 Adder/Subtractor 6

2.1.5 And, Or, Not, XOR 7

2.1.6 Left and Right Shifting 8

2.2 Status 9

2.2.1 Negative 9

2.2.2 Zero 10

2.2.3 oVerflow 10

2.2.4 Carry 11

2.3 One Step Further! 11

2.3.1 Arithmetic right shift 11

3 Conclusion **12**

Resources **12**

List of Figures

1 1-Bit Adder Karnaugh Map 1

2 1-Bit Adder Circuit 2

3 8 Bits Ripple-Carry Adder Circuit 3

4 ALU Output Control 5

5 ALU Adder 5

6 ALU Subtractor - With two adders 6

7 ALU Subtractor - With one adder 6

8 OP = 1 Checker 7

9 ALU Adder/Subtractor 7

10 ALU Bit-by-Bit Operations 8

11 ALU shifting Operations 8

12 ALU Status: BUS 9

13 ALU Status: N Flag 9

14 ALU Status: Z Flag 10

15 ALU Status: V Flag 11

16 Arithmetic Right Shift 12

List of Tables

1 1-Bit Adder Truth Table 1

1 Adder

1.1 1-Bit Adder

1.1.1 Truth Table

c_n	a_n	b_n	c_{n+1}	s_n
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 1: 1-Bit Adder Truth Table

a_n**:** Input A (1-Bit).

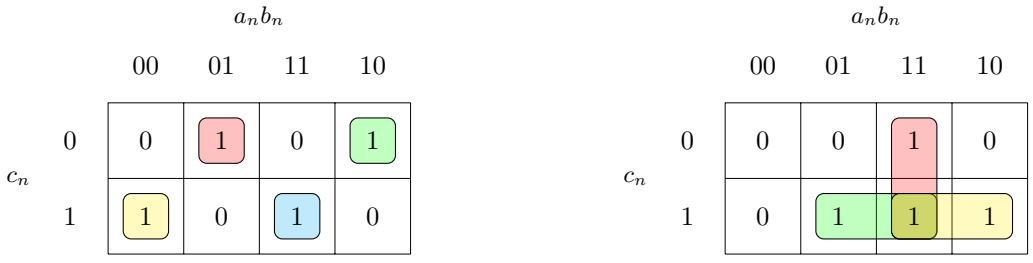
b_n**:** Input B (1-Bit).

c_n**:** The Carry from the previous stage (1-Bit).

s_n**:** This stage's result (1-Bit).

c_{n+1}**:** This stage's Carry (to be used in the next stage) (1-Bit).

1.1.2 Karnaugh Map



(a) s_n (b) c_{n+1}

Figure 1: 1-Bit Adder Karnaugh Map

1.1.3 Equations

Using the two Karnaugh Maps extracted from the truth table above, we can find the two simplified equations for s_n and c_{n+1}.

$$c_n = a_n \cdot b_n + a_n \cdot c_n + b_n \cdot c_n$$

We notice that the 4 implicants in s_n are made up of only one single bit each, which does not simplify any of the input, making each group equation depend on all three inputs.

$$s_n = \overline{a_n} \cdot b_n \cdot \overline{c_n} + a_n \cdot \overline{b_n} \cdot \overline{c_n} + \overline{a_n} \cdot \overline{b_n} \cdot c_n + a_n \cdot b_n \cdot c_n$$

However, given the definition of XOR: $a \oplus b = \overline{a} \cdot b + a \cdot \overline{b}$, we can see that the previous equation can be simplified, and with some experimentation, we found that s_n can be written as

$$s_n = (a_n \oplus b_n) \oplus c_n$$

We are not sure how this could be reached, as the first equation contains both XOR and XAND gates, and they are kinda confusing to deal with, nevertheless, this worked 😄

1.1.4 Circuit

We won't be pointlessly explaining how to implement the circuit in `Logisim Evolution` (connections, circuits' placement), as this is not the point of the Lab Session, so we will insert screenshots of the circuits directly.

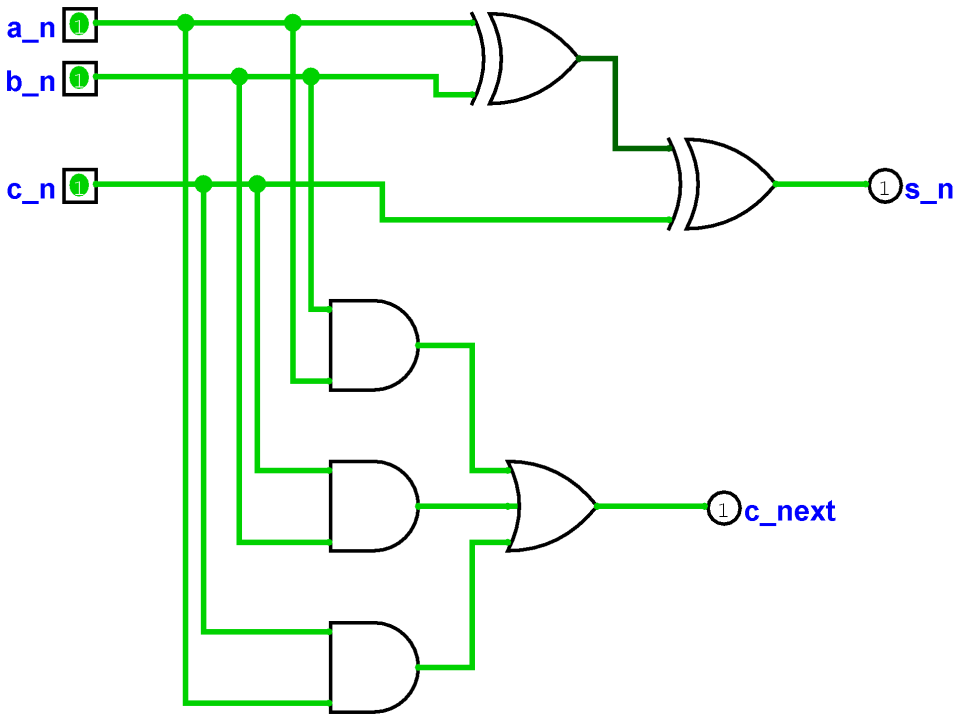


Figure 2: 1-Bit Adder Circuit

1.2 8-Bit Ripple-Carry Adder

1.2.1 Logic

An 8-bit binary adder can be simply achieved by chaining 8 1-bit adders and connecting each c_{n+1} to c_n of the next stage (as the name of the circuit suggests).
This actually can be done for any size n.

1.2.2 Circuit

To simplify the usage, and make the circuit more concise, we used Logisim's built-in splitters and bus input/output.

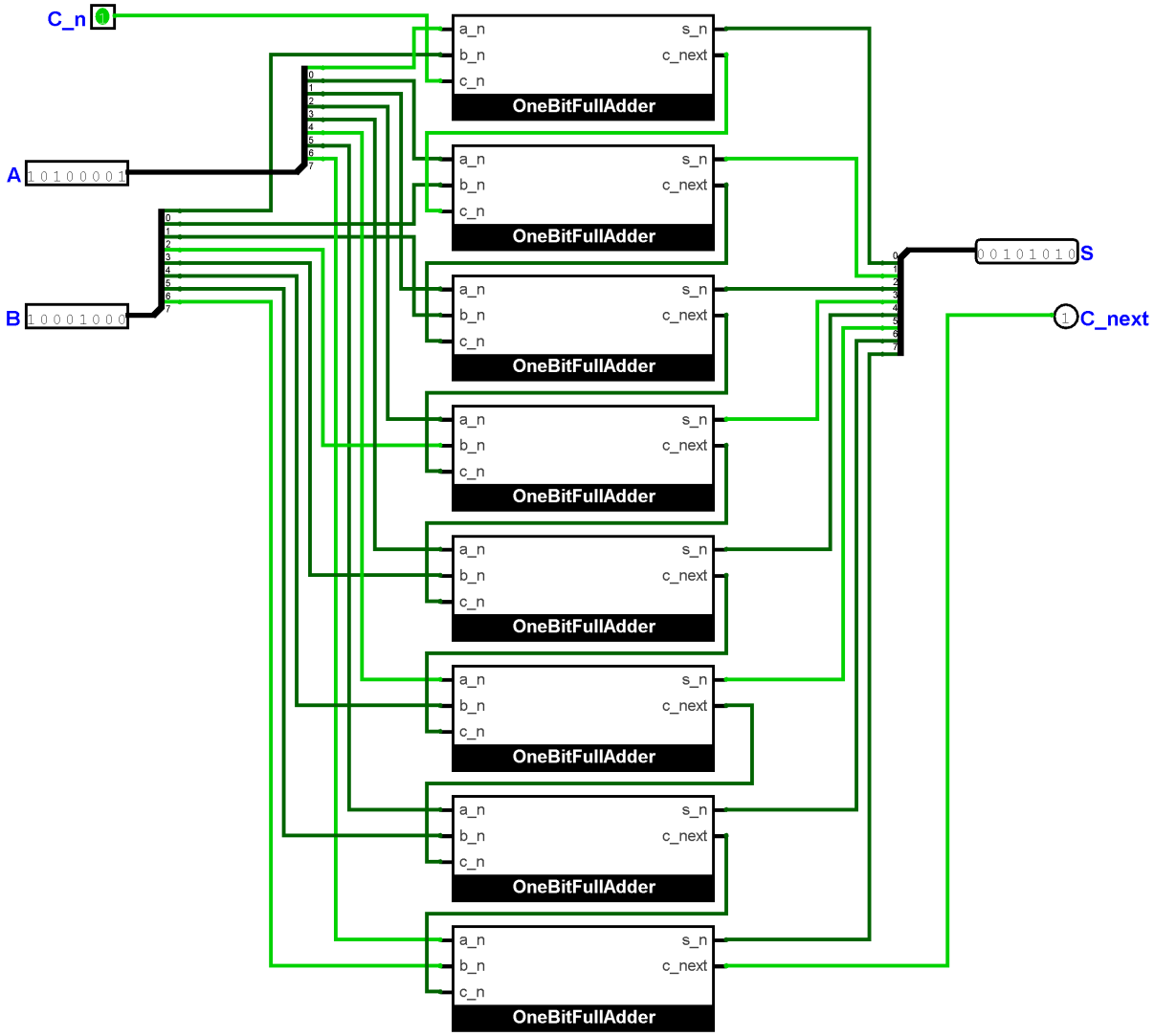


Figure 3: 8 Bits Ripple-Carry Adder Circuit

The circuit shows the correct result of the addition of

$$1010\,0001\ (161) + 1000\,1000\ (136) + 1\ (1) = 1\,0010\,1010\ (298)$$

The result is on 9 bits, with the Carry being the MSB.

2 ALU

2.1 Operations

2.1.1 Output Control / Selected Operation

Given that the ALU should execute many operations, we need a way to control which of the operations to execute. This is done through the 3-bit `OP` input, which allows us to select between $2^3 = 8$ different operations, mapped as the following:

0. addition
1. subtraction
2. and
3. or
4. not: ~ a (Changed to arithmetic shift right later-on)
5. xor
6. shift left a « b
7. shift right: a » b

We used an 8 to 1 Multiplexer (provided by Logisim) to easily control the selected output, with the selection bus being coded on 3-bits (hence the 8 inputs) and each of the inputs and the output being an 8-bit data bus.

We used Tunnels instead of wiring everything to make the circuit cleaner and less crowded, and to avoid dumb wiring mistakes.

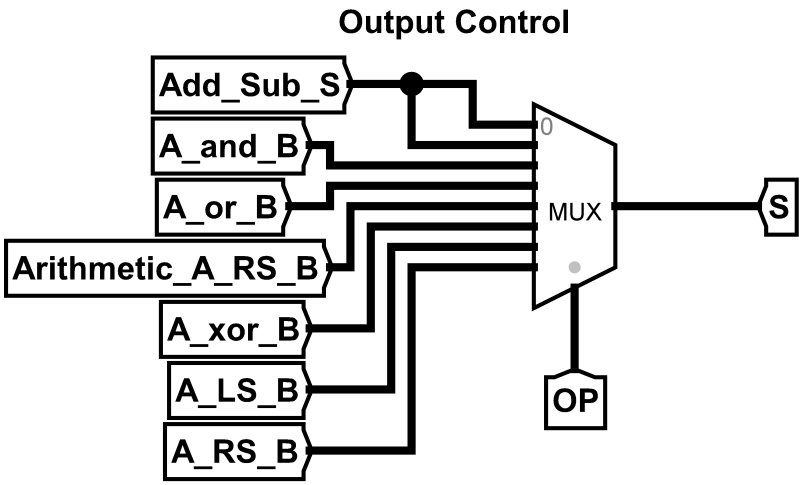


Figure 4: ALU Output Control

You notice that instead of having two tickets for both addition and subtraction, we have only one for both, connected to the two first inputs of the multiplexer. We will come back to this in the subtraction paragraph.
Also, the `not` is already not there, as when this image was exported, we have already replaced it with the arithmetic right shift.

2.1.2 Addition

Implementing Addition is very straightforward, as we already have built an 8-bit adder, only left to connect the inputs and output to their proper correspondence.

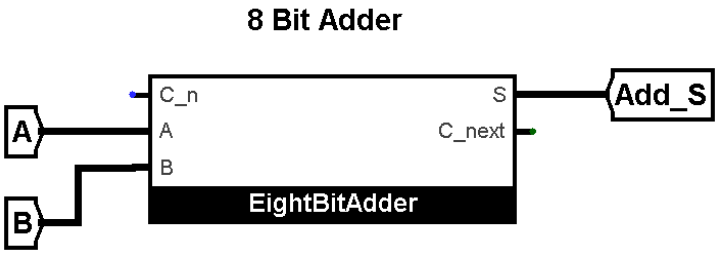


Figure 5: ALU Adder

Nevertheless, this circuit will be soon changed when we implement the subtraction.

2.1.3 Subtraction

With some 6^{th} grade's math knowledge, we can deduct that $a - b = a + (-b)$, and with that, we can use the same 8-bit adder that we have built before to do the subtraction, the only thing left is inverting the sign of b.
We will be using 2's complement to represent negative numbers in our binary system, so $-b = \sim b + 1$.

Our first solution, involved first inverting B, then adding 1, and only then summing it with A.

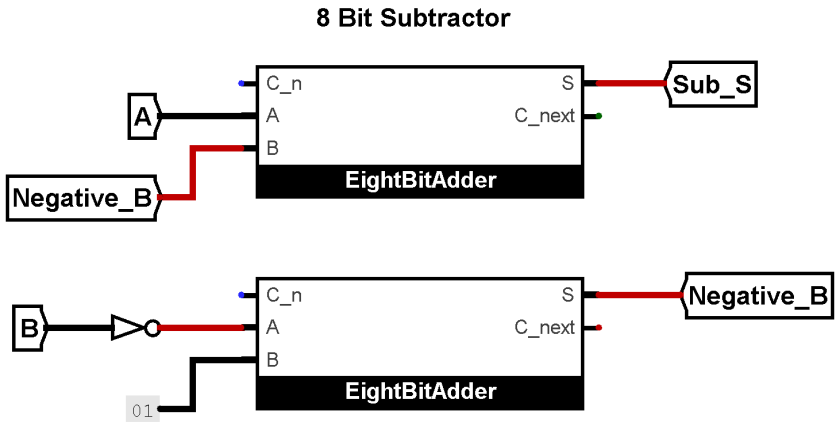


Figure 6: ALU Subtractor - With two adders

This solution works, however, we use two 8-bit adders to achieve this, and that is a waste of components since we can use the C_n input for the $+1$ in our 2's complement equation, therefore reducing our subtractor to half its size.

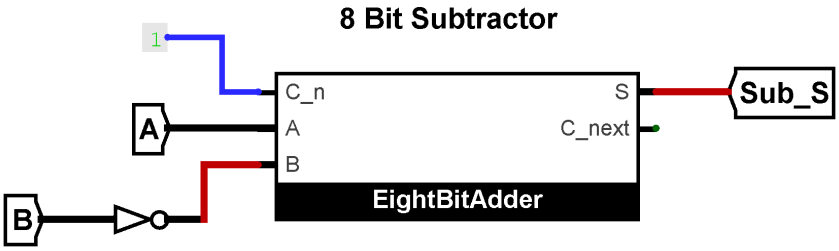


Figure 7: ALU Subtractor - With one adder

This is a more optimized solution for our subtractor, but let's not forget that we already have an 8-bit adder in our ALU, wouldn't be great to use it for both addition and subtraction? **Well, WE CAN!**

2.1.4 Adder/Subtractor

As hinted above, we can use a single adder to do both addition and subtraction in our ALU, reducing the used number of adders by two-thirds, from 3 to 1.

We can check if the `OP` input is set to subtraction (= 1), if so, we can run invert the B input and set $C_n = 1$ of our adder to make it calculate $S = A + \sim B + 1 = A + (\sim B + 1) = S = A + (-B) = A - B$, otherwise we can run normal addition of A and B.
We will have a single output for both addition and subtraction, that we connect to both addition and subtraction result inputs on our Output Control Multiplexer (exactly what we have in the image before).

For OP to be set to Subtraction (= 1), $OP = 001_{(2)} \Rightarrow \overline{OP_2} \cdot \overline{OP_1} \cdot OP_0$.
We can add a simple AND gate for the check.

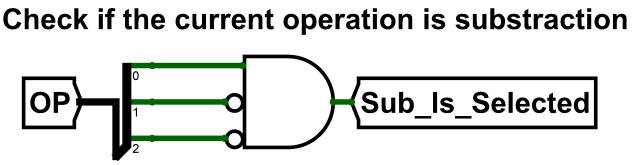


Figure 8: OP = 1 Checker

Given this information, we can decide whether we have to inverse the B sign or not, we can use a 2 to 1 multiplexer for this.

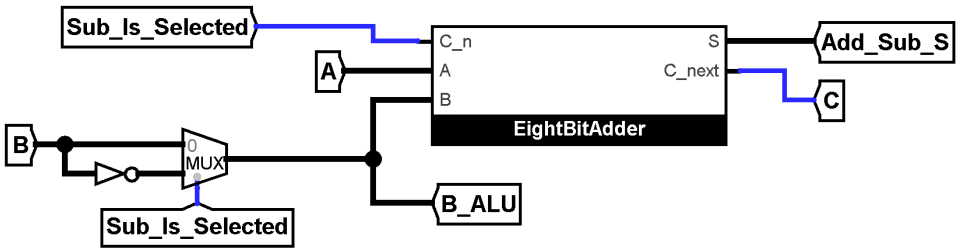
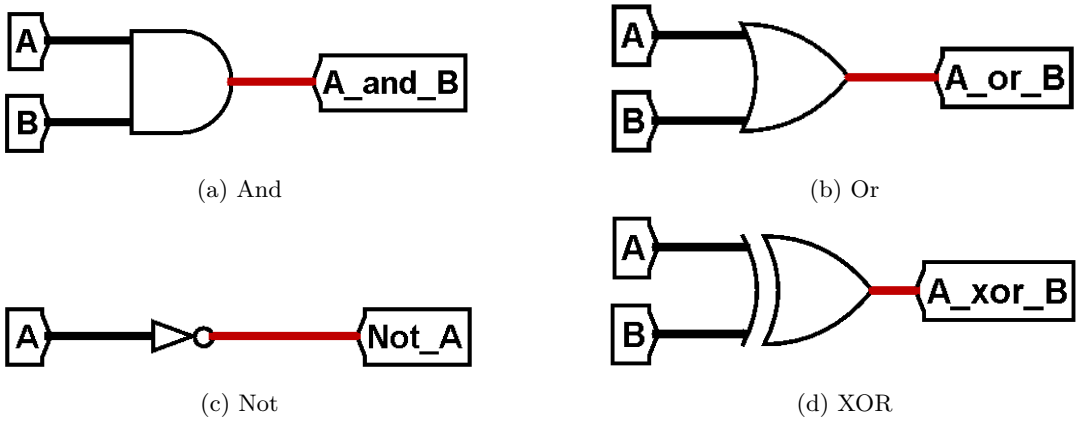


Figure 9: ALU Adder/Subtractor

We add a Tunnel at the output of the multiplexer to get the value of B depending on the selected operation, this will come in handy later on when we implement the Status output.

2.1.5 And, Or, Not, XOR

Implementing these operations is very easy since Logisim already provides us with plug-and-play components that do the exact tasks, we only need to change the data size from a single bit to 8 bits.



(a) And

(b) Or

(c) Not

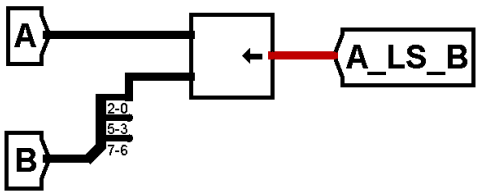
(d) XOR

Figure 10: ALU Bit-by-Bit Operations

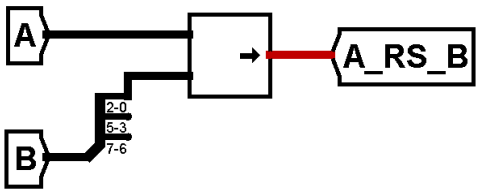
2.1.6 Left and Right Shifting

Logisim already provides us with Left and Right shift components that we use directly, however, given our A and B inputs are 8 bits each, the maximum value that we can shift is 7, beyond that, we would only get 0s, which is useless, so we should limit the input of B from 255 to 7, by only taking the first 3 bits (B_2, B_1, and B_0).

We can achieve that with the build-in Splitter, by setting the Fan Out to 3 and the Bit In Width to 8, which will do $\lceil 8/3 \rceil = 3$, and outputting bits 0 to 2 in the first output, 3 to 5 in the second, and the remaining 6 and 7 in the third. We then take the first output as our shifting amount.



(a) Left Shift



(b) Right Shift

Figure 11: ALU shifting Operations

2.2 Status

Our ALU also has a 4-bit output as Status, each bit is a flag for a specific status of our result, mapped as follows:

0. **N**egative
1. **Z**ero
2. o**V**erflow: the result is not correct with signed numbers, i.e. the sum of 2 positive numbers is a negative number.
3. **C**arry: the result is not correct with unsigned numbers: this is the 9th bit of an operation.

We can use a Splitter in reverse to transform the single flags into a single 4-bit Status BUS.

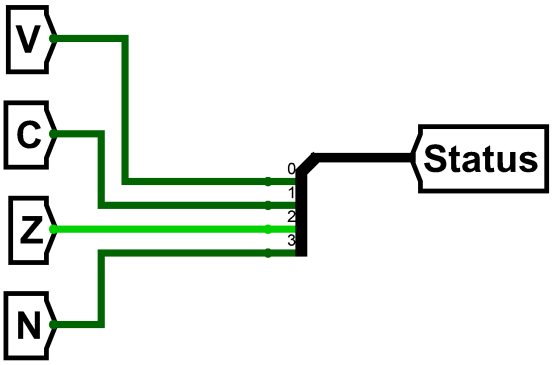


Figure 12: ALU Status: BUS

2.2.1 Negative

This bit is straightforward to extract. Since we are using 2's complement to represent negative values, we can just check the MSB of our ALU output if it is equal to 1 or 0, 1 being negative, and positive otherwise.
We can use a splitter for the bit extraction.

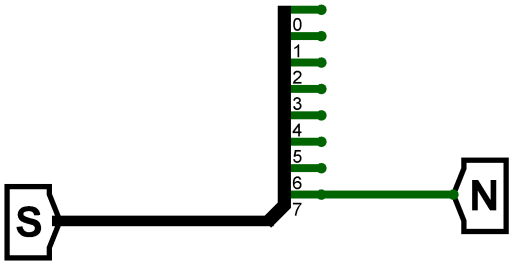


Figure 13: ALU Status: N Flag

2.2.2 Zero

We can find the Z flag by checking that every bit of our ALU output is equal to 0, giving us the following equation:

$$Z = \overline{S_7} \cdot \overline{S_6} \cdot \overline{S_5} \cdot \overline{S_4} \cdot \overline{S_3} \cdot \overline{S_2} \cdot \overline{S_1} \cdot \overline{S_0} \;=\; \overline{S_7 + S_6 + S_5 + S_4 + S_3 + S_2 + S_1 + S_0}$$

We can implement the second equation with a single 8-input NOR gate.

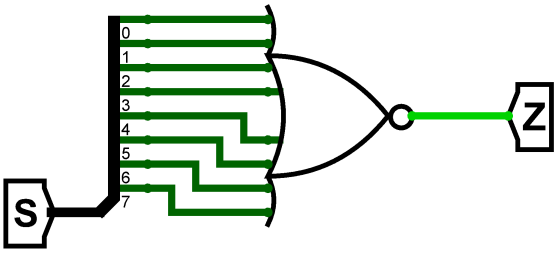


Figure 14: ALU Status: Z Flag

2.2.3 oVerflow

As explained above, the oVerflow Flag is set when the result of addition or subtraction overflows into the sign bit (MSB), which renders the result incorrect.

If the result of an addition between two positive numbers is negative (MSB of S is 1) we know that we have an overflow, similarly, if the addition of two negative numbers is positive (MSB of S is 0) we know that there is an overflow. This also applies if A + B if both A and B are negative, or A - B if B is negative.

This can be irritating to implement for all use-cases, but we can heavily simplify our check if we take the B from the multiplexer that we have already set up in our Adder/Subtractor, this will take care of all the use-cases, and the only thing remaining is to check if the sign of A and B is coherent with the sign of the result S (the N flag previously implemented).

$$V = MSB(A) \cdot MSB(B) \cdot \overline{N} + \overline{MSB(A)} \cdot \overline{MSB(B)} \cdot N$$

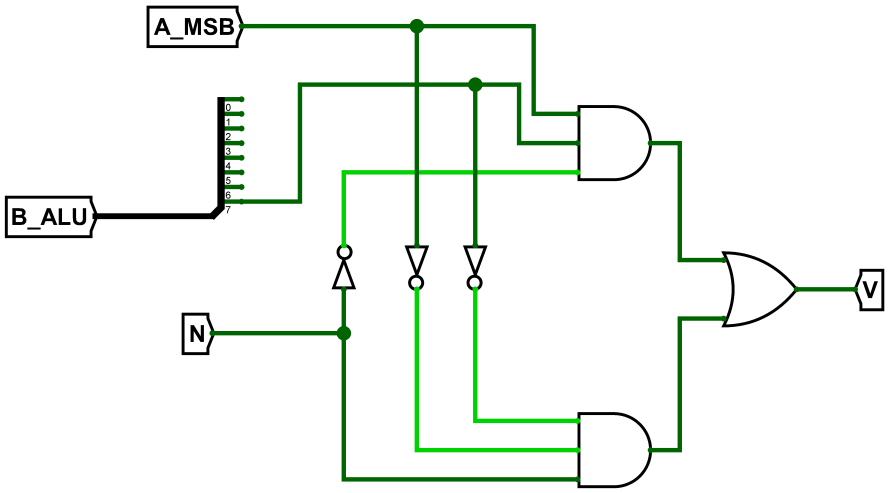


Figure 15: ALU Status: V Flag

2.2.4 Carry

Similar to the N flag, this is straightforward to find, it is equal to C_n of our Adder/Subtractor. (See Figure 9)

2.3 One Step Further!

2.3.1 Arithmetic right shift

An arithmetic right shifter is an ordinary right shifter, but it is sign-aware!!.
If the input is negative (MSB is 1), instead of filling the gap with 0s, it will fill it with 1s instead, keeping the same sign of the result.

We think our solution is not the best, or the most efficient. We have simply extended the A input from 8 bits to 16 bits. the extra 8 bits are either 0s (0x00) if A is positive, or 1s (0xFF) if A is negative. With this in mind, we have to also extend the shifting amount to be coded on 4 bits instead of the previous 3 ($16 = 2^4$), so we just forced the 3^{rd} bit to always be 0 by hooking it to ground.

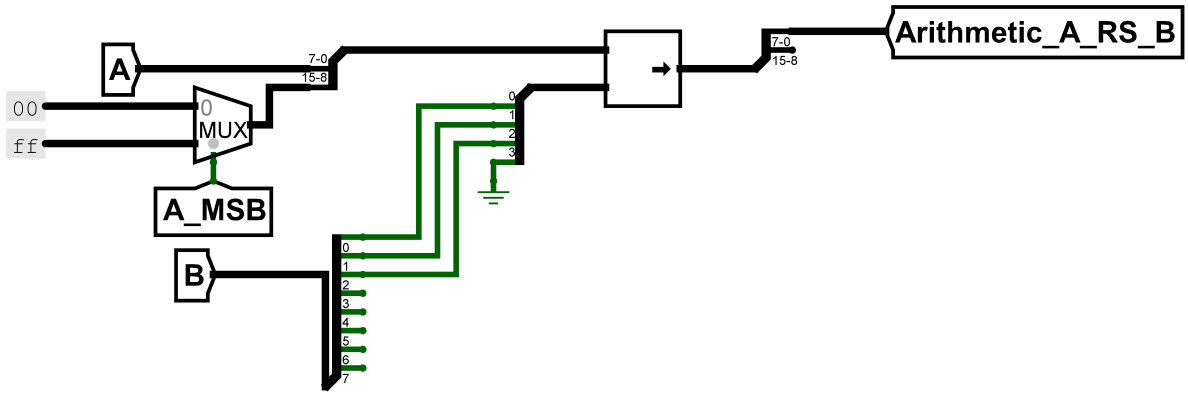


Figure 16: Arithmetic Right Shift

3 Conclusion

In conclusion, the successful implementation of an 8-bit Arithmetic Logic Unit (ALU) in Logisim has provided us with valuable hands-on experience in digital logic design. This project allowed us to apply our knowledge of combinational circuits, logical operations, and circuit simulation to create a functional ALU that can perform a variety of arithmetic and logic operations.

We would have loved it if we were able to implement the Multiplier circuit, but unfortunately, due to its complexity, our humble knowledge, and the limited time we had (traveling to different cities where our companies are), we couldn't find enough time to understand and implement the circuit.

Nevertheless, through this project, we gained a deeper understanding of the complexities and intricacies of digital circuits and the importance of careful design and testing. Overall, our work on the 8-bit ALU in Logisim has been an insightful and rewarding journey into the world of digital design.

Resources

The Logisim Circuit File, and this report's source code, are available on this GitHub repository: elkhayder/sec1-tp-mac