Computer Organization

<BLG 222E>

Project Report

Group:

Fatih Uzunoğlu – 150170725

Uzay Dalyan – 150170041

Berdan Çağlar Aydın – 150170068

Yiğit Emre Yılmaz – 150170071

---GROUP NR. #**26**---

Project Number: #2

Date: 29/04/2020

Introduction

In the second project we have successfully built an Arithmetic Logic Unit (ALU) which is capable of doing arithmetic, logic and shift operations on 8-bit / 1-byte numbers. We have also implemented 4-bit FLAG register and an ALU system for the purpose of using it later on.

An ALU is a building block / a built-in component of a CPU. It works by combinational logic therefore it does not cause any lag or sacrifice CPU cycles for the operations it does.

Following are designed and implemented in this project as part of the ALU and ALU-related implementations:

- 4-bit FLAG register
- 8-bit Arithmetic Logic Unit (ALU)
    - Shift unit (LSL, LSR, ASL, ASR, CSL, CSR)
    - Arithmetic unit (ADD, SUB)
    - Logic unit (BUFFER, NOT, AND, OR, XOR)
- ALU Testbed
- ALU System

Logisim (v3.3.4 / Logisim-Evolution) was used for the implementations since the original logisim caused problems in our computers because of it being very old and not maintained for a very long time. Note that it requires the latest version of Java runtimes to run. Please read README.txt if you have troubles opening the project.

Part-1

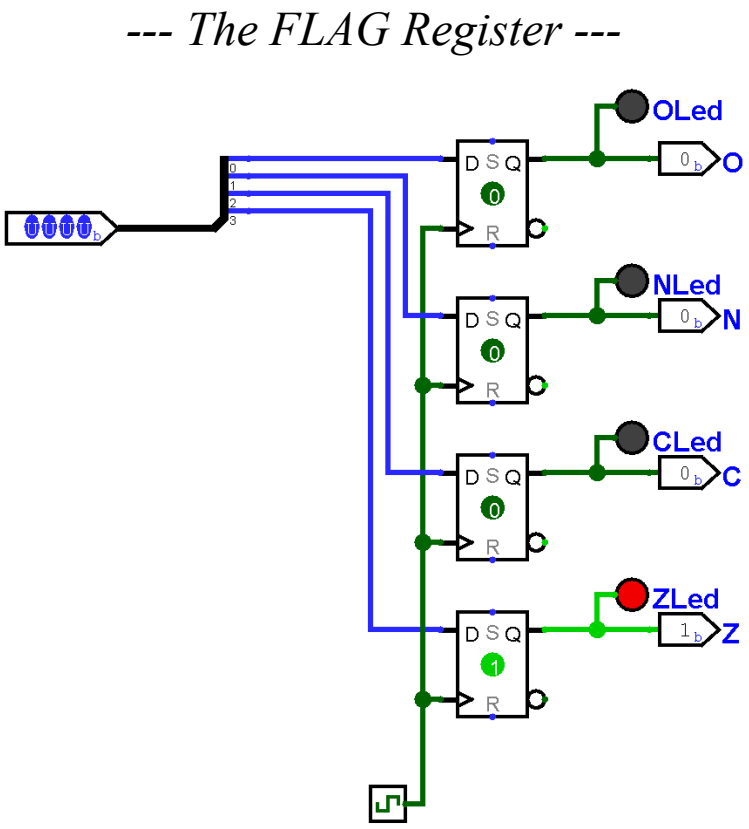


Figure 1 Flag Register Circuit

A 4-bit flag register was built as an initial step. The circuit can be seen in *Figure 1.* It uses 4 D-type flip flops for bit storage purposes. When it receives U (Unknown / High-Z) information in the input bits, the stored flag bits remain (it accepts tri-state / U input). It has 4 different LEDs to show the stored FLAG bits.

Part-1a

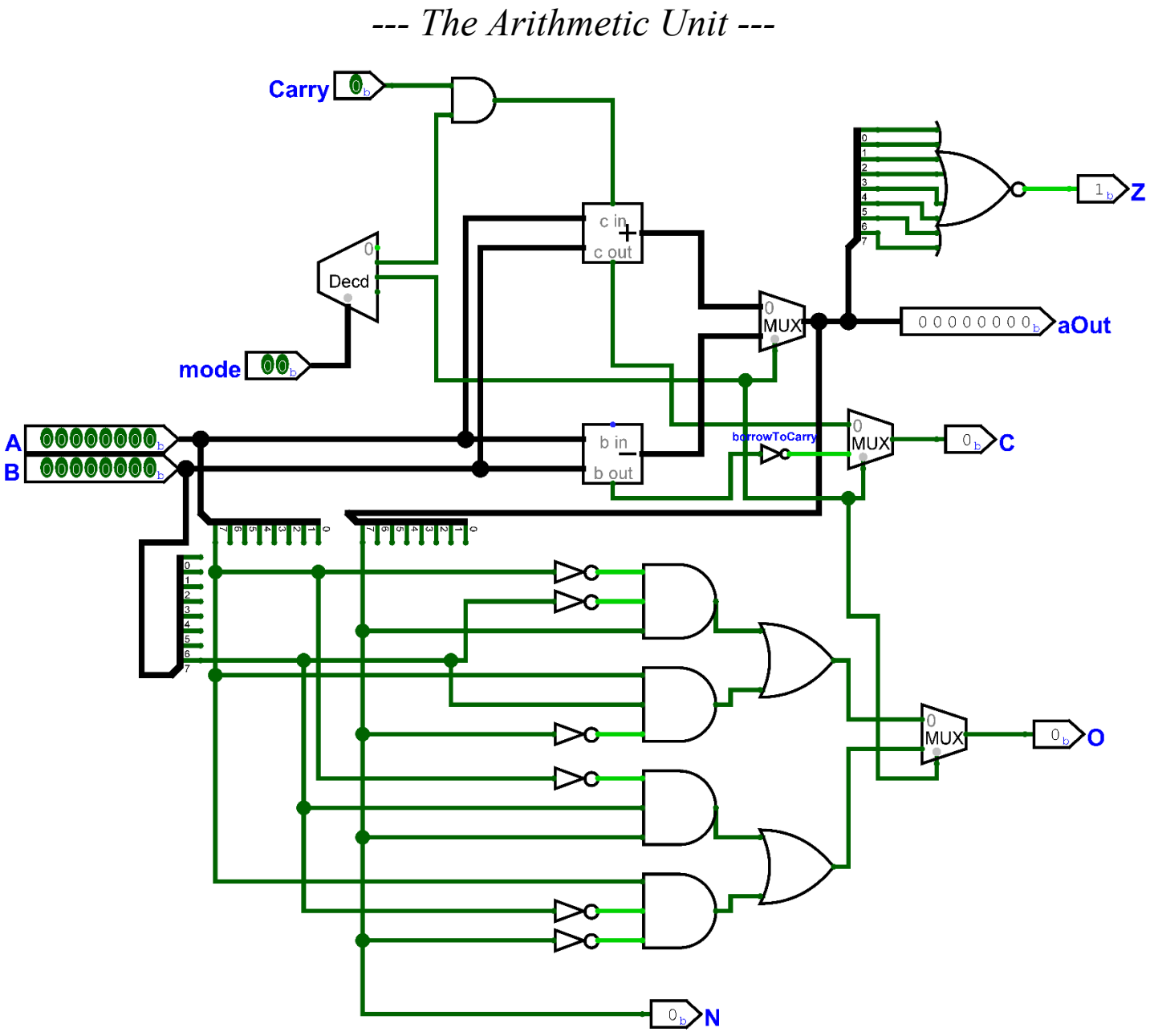


Figure 2 Arithmetic Unit of the ALU

After building the FLAG register, we then continued with implementing the *Arithmetic Unit* for our ALU. We designed our ALU with a modular approach and every functionality that the ALU is capable of doing is actually done by the modules or so-called components that we built before. Figure 2 shows how Arithmetic Unit was designed by us.

Arithmetic Unit is capable of doing addition and subtraction operations. It also feeds FLAG register with certain flags (Z, C, O, N). In our design, when a flag is not changed, components break the regarding FLAG bit

connection with High-Z (high impedance) technique (Tri-State). And as mentioned before, FLAG register does not alter itself if any input is in High-Z / U state.

Our arithmetic unit does the operations with the help of a binary adder and a subtractor. For operation selection, it uses the information coming from the 2-bit input pin: *mode.* There is an underlying mapping mechanism for mode selection within every component we built. The mapping for Arithmetic Unit is shown below.

FunSel	OutALU	Z	C	N	O
0000	A	√	–	√	–
0001	B	√	–	√	–
0010	NOT A	√	–	√	–
0011	NOT B	√	–	√	–
0100	A + B	√	√	√	√
0101	A + B + Carry	√	√	√	√
0110	A - B	√	√	√	√
0111	A AND B	√	–	√	–
1000	A OR B	√	–	√	–
1001	A XOR B	√	–	√	–
1010	LSL A	√	√	√	–
1011	LSR A	√	–	√	–
1100	ASL A	√	–	√	–
1101	ASR A	√	–	–	√
1110	CSL A	√	√	√	√
1111	CSR A	√	√	√	√

Arithmetic unit does the highlighted operations. ALU implementation redirects FunSel inputs to module *modes*. For Arithmetic Unit, the mapping is as such: (see Part-1d for the mapping circuitry)

ALU: *FunSel*	**Arithmetic Unit: *mode***
0100	00
0101	01
0110	10

With help of decoder and multiplexer, Arithmetic Unit sets the correct mode (redirects regarding outputs properly).

For Carry flag, it feeds Carry in when needed (0101 / 01) to the binary adder. And uses the reverse of borrow out of the subtractor as the carry out.

For the Zero flag, all output bits are fed into a NOR gate to check if any of the bits is 1. If any of the bits are 1, the Z flag becomes 1 as well.

For the Negative flag, MSB of the output is directly connected to the N out pin.

For the Overflow flag, input and output numbers are compared against each other. It is implemented based on the table given below. Note that the signs of the numbers are determined by their corresponding MSBs.

Cases	O out (Overflow flag out)
Positive + Positive = Negative	1
Negative + Negative = Positive	1
Positive – Negative = Negative	1
Negative – Positive = Positive	1
*** ANY OTHER CASE ***	0

The lower part of the arithmetic unit circuit (Fig. 2) implements the functionality of this table.

Part-1b

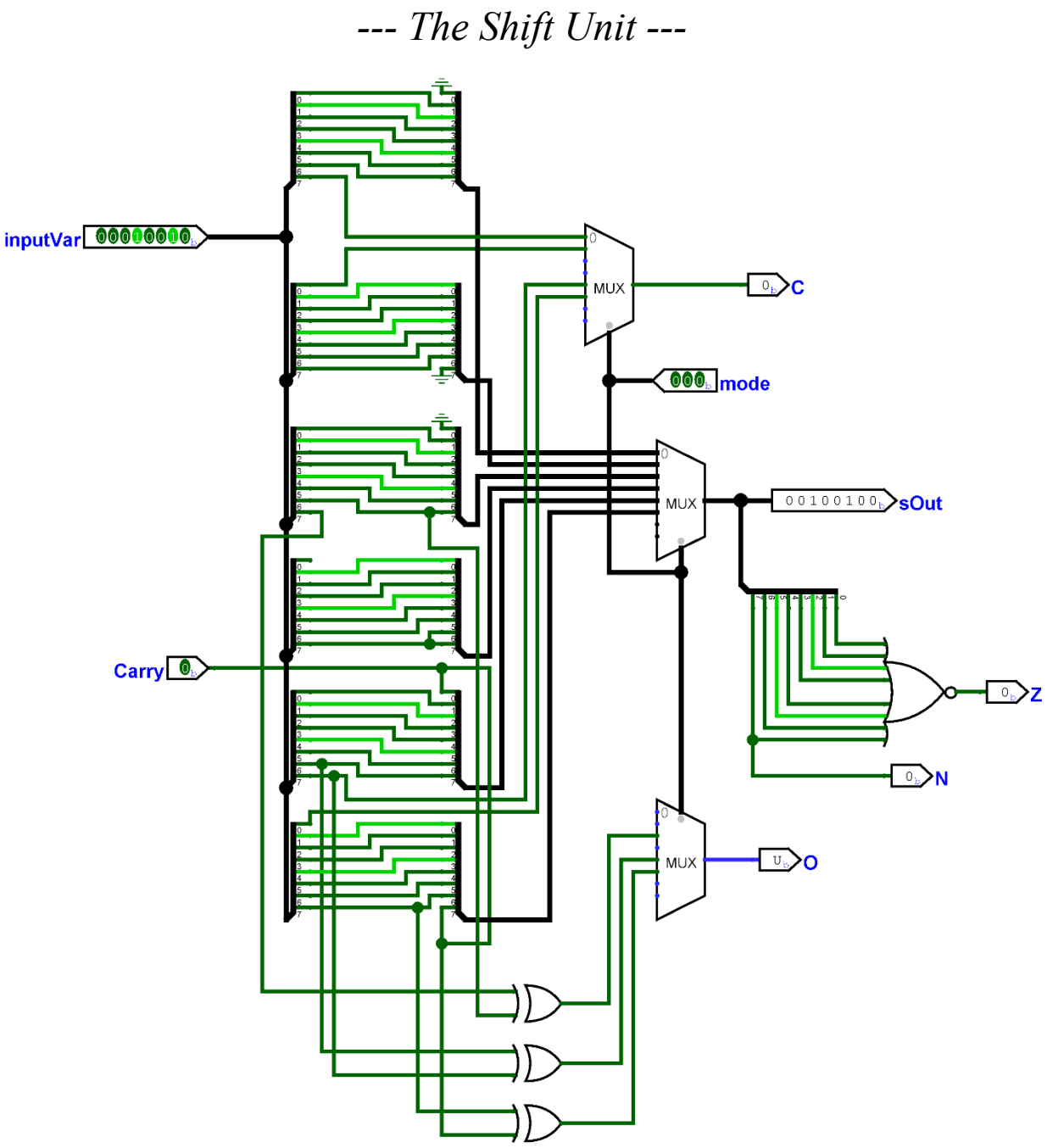


Figure 3 Shift Unit of the ALU

We followed bit-by-bit redirecting approach for the shift unit. It required hardwiring a lot of cables but since it is only 8-bit we concluded that it's the best way to implement a shifter. Otherwise we would have needed to use a lot of decoders with a complex underlying logic.

The shifter determines the sOut result according to the input pin: *mode*. The logic is the same as Arithmetic Unit. Multiplexers maps the operation to the corresponding output according to mode. The mapping is described below.

FunSel	OutALU	Z	C	N	O
0000	A	√	–	√	–
0001	B	√	–	√	–
0010	NOT A	√	–	√	–
0011	NOT B	√	–	√	–
0100	A + B	√	√	√	√
0101	A + B + Carry	√	√	√	√
0110	A - B	√	√	√	√
0111	A AND B	√	–	√	–
1000	A OR B	√	–	√	–
1001	A XOR B	√	–	√	–
1010	LSL A	√	√	√	–
1011	LSR A	√	–	√	–
1100	ASL A	√	–	√	–
1101	ASR A	√	–	–	√
1110	CSL A	√	√	√	√
1111	CSR A	√	√	√	√

Shift unit does the highlighted operations. The mapping table is given below. (see Part-1d for the mapping circuitry)

ALU: *FunSel*	**Shift Unit: *mode***
1010	000
1011	001
1100	010
1101	011
1110	100
1111	101

For the FLAG bits, it follows the same approach of the Arithmetic Unit except Carry and Overflow.

Shifter assigns Carry out and Overflow out flags according to theoretical definitions of these shifting operations. Note that the missing inputs of the Multiplexers show that the flag does not change for that mode. It means High-Z / Unknown and FLAG register does not change its FLAG bits if the input is High-Z. Overflow check is basically comparing sign bit change of the output and input numbers. If there is a difference O becomes 1.

Part-1c

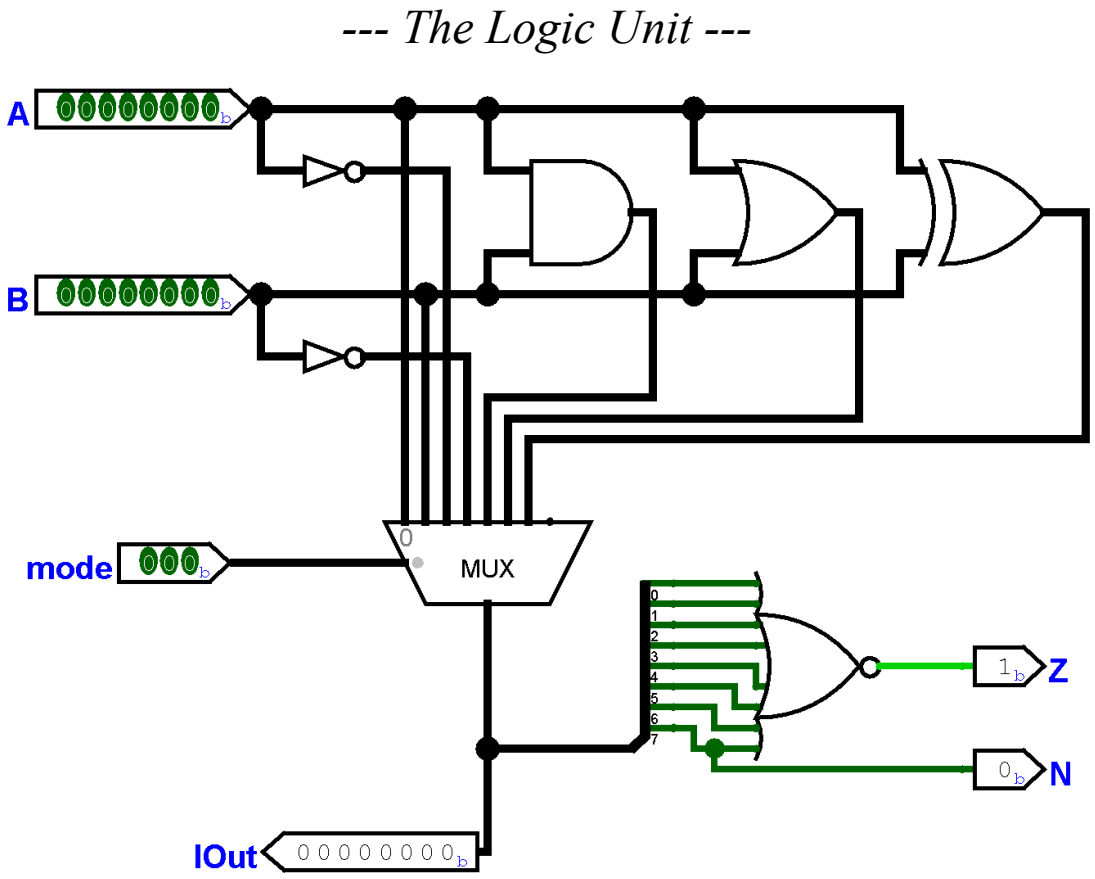


Figure 4 The Logic Unit of the ALU

The logic unit was rather simpler to build. It only requires a few distinct logic gates. Just like the others, it follows a mapping approach for operation/output determination.

FunSel	OutALU	Z	C	N	O
0000	A	√	–	√	–
0001	B	√	–	√	–
0010	NOT A	√	–	√	–
0011	NOT B	√	–	√	–
0100	A + B	√	√	√	√
0101	A + B + Carry	√	√	√	√
0110	A - B	√	√	√	√
0111	A AND B	√	–	√	–
1000	A OR B	√	–	√	–
1001	A XOR B	√	–	√	–
1010	LSL A	√	√	√	–
1011	LSR A	√	–	√	–
1100	ASL A	√	–	√	–
1101	ASR A	√	–	–	√
1110	CSL A	√	√	√	√
1111	CSR A	√	√	√	√

The logic unit does the operations that are highlighted in the table. Note that since operations 0000 and 0001 are not necessarily logic operations, we implemented them inside our Logic Unit because it was the simplest unit to modify. We simply connected input lines to the output multiplexer to act as a BUFFER for these operations.

Other operations such as AND and OR are done by their respective gates. Z and N flags are determined the same as previously described units. Mapping table is given below. (see Part-1d for the mapping circuitry)

ALU: *FunSel*	**Logic Unit: *mode***
0000	000
0001	001
0010	010
0011	011
0111	100
1000	101
1001	110

Part-1d

--- The Arithmetic Logic Unit ---

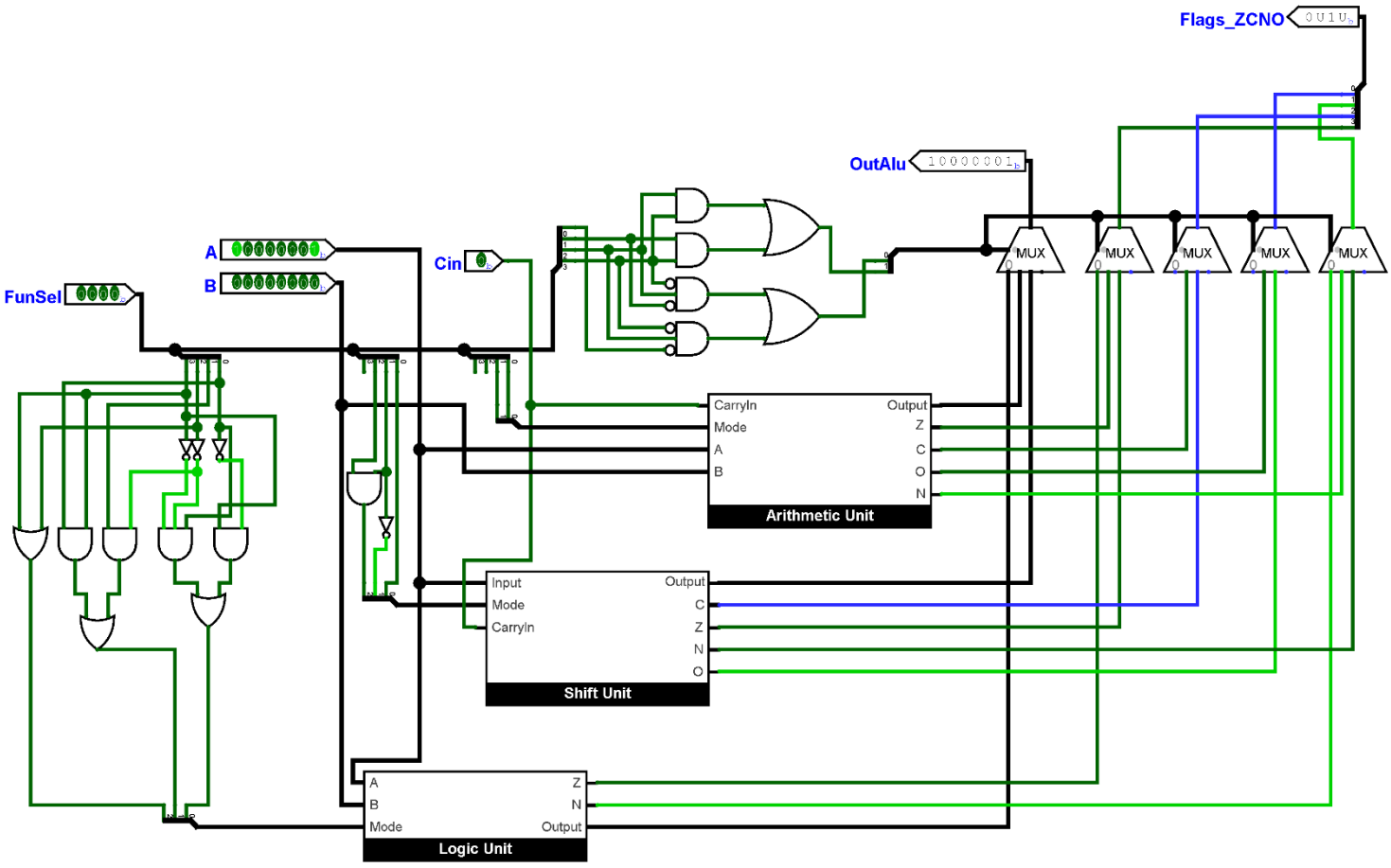
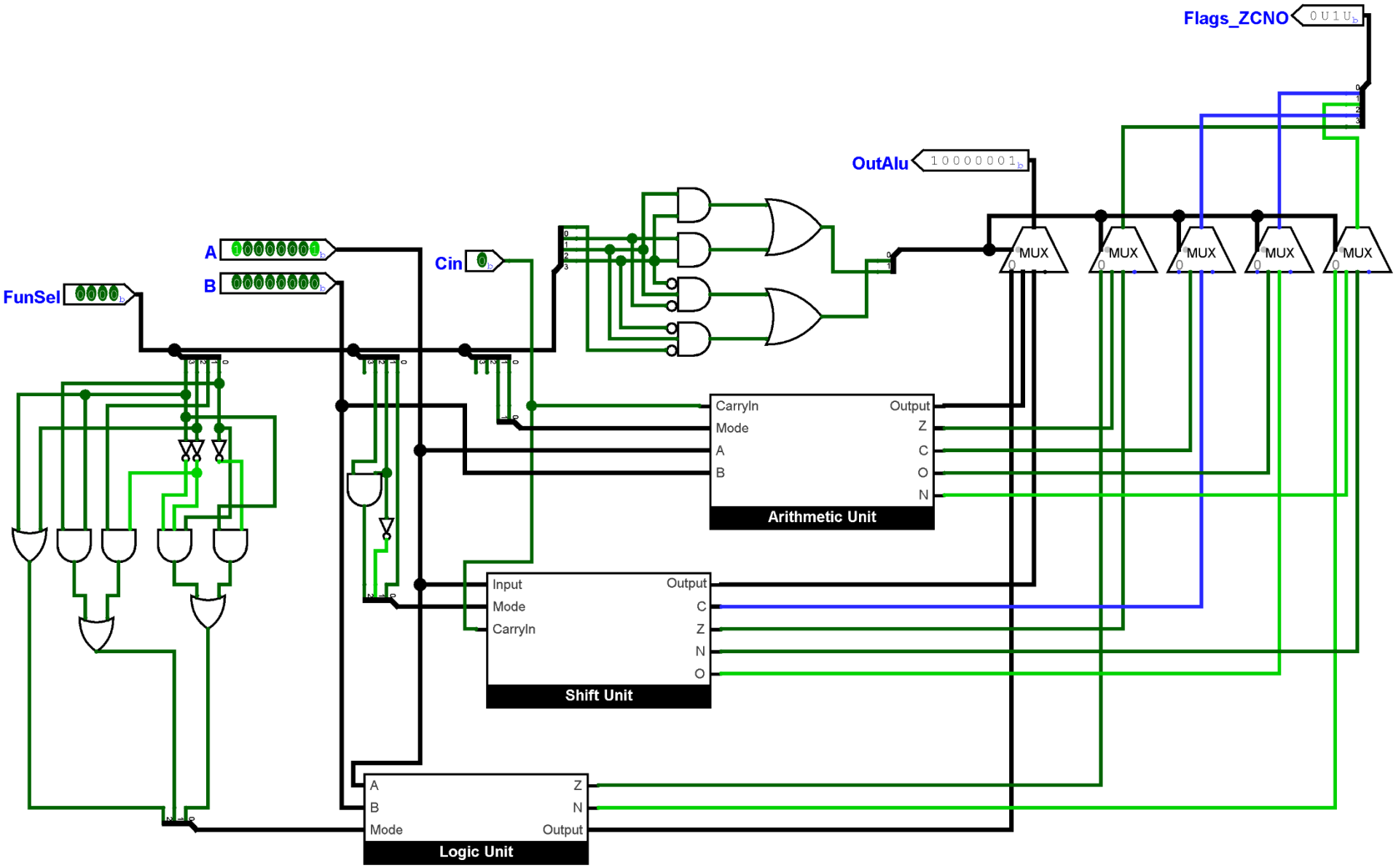
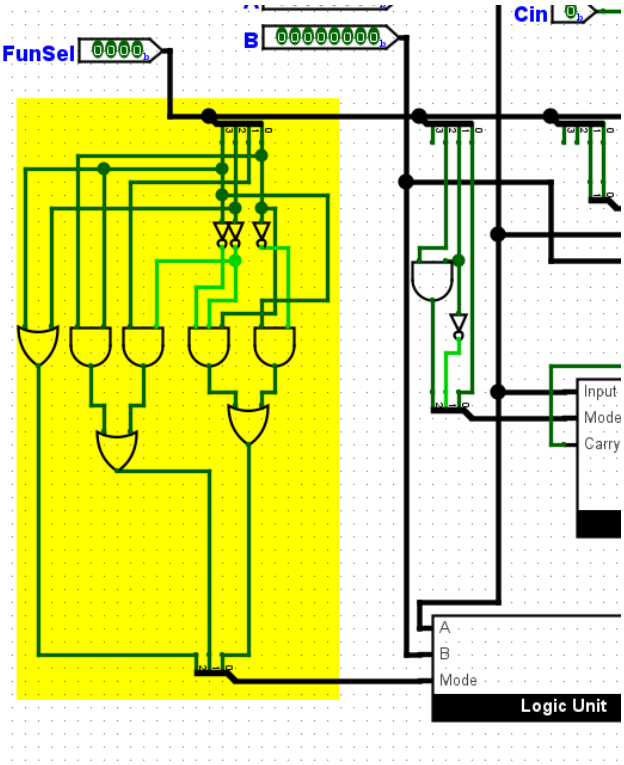


Figure 5 The Arithmetic Logic Unit

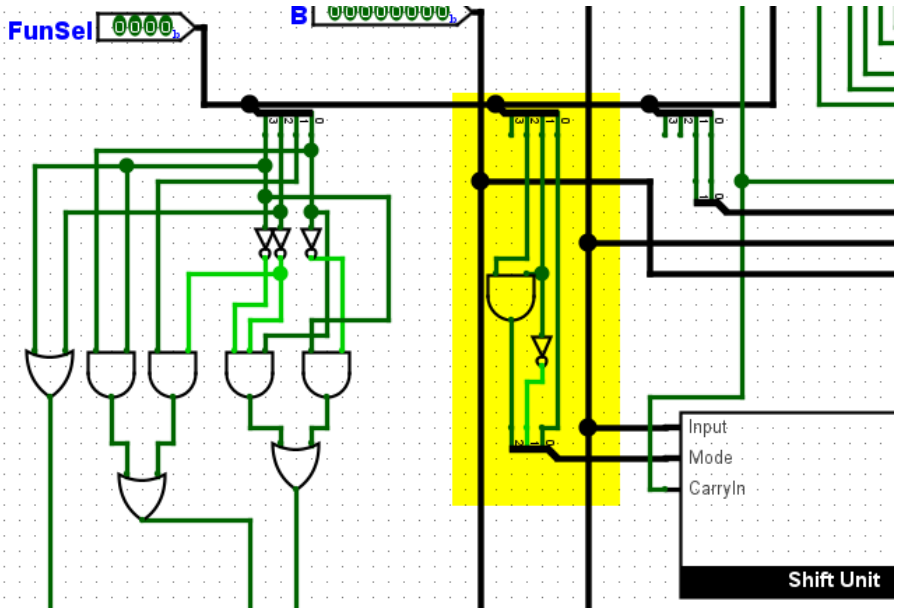

Flags_ZCNO
0 U 1 U b
OutAlu
1 0 0 0 0 0 0 1 b
A
B
Cin
FunSel
MUX
MUX
MUX
MUX
MUX
CarryIn
Mode
A
B
Output
Z
C
O
N
Arithmetic Unit
Input
Mode
CarryIn
Output
C
Z
N
O
Shift Unit
A
B
Mode
Z
N
Output
Logic Unit

The Arithmetic Logic Unit contains all the modules that we designed and described above to perform operations. It uses mapping for all three components to map FunSel to their internal modes. In Figure 5, one can see that all the circuitry implemented for Mode inputs of components as well as Flag selector Multiplexer inputs are actually implemented just for the mapping (redirecting i/o properly). They do not carry out any operations. All these mapping circuitry are designed by doing several times of Karnaugh map reduction calculation.

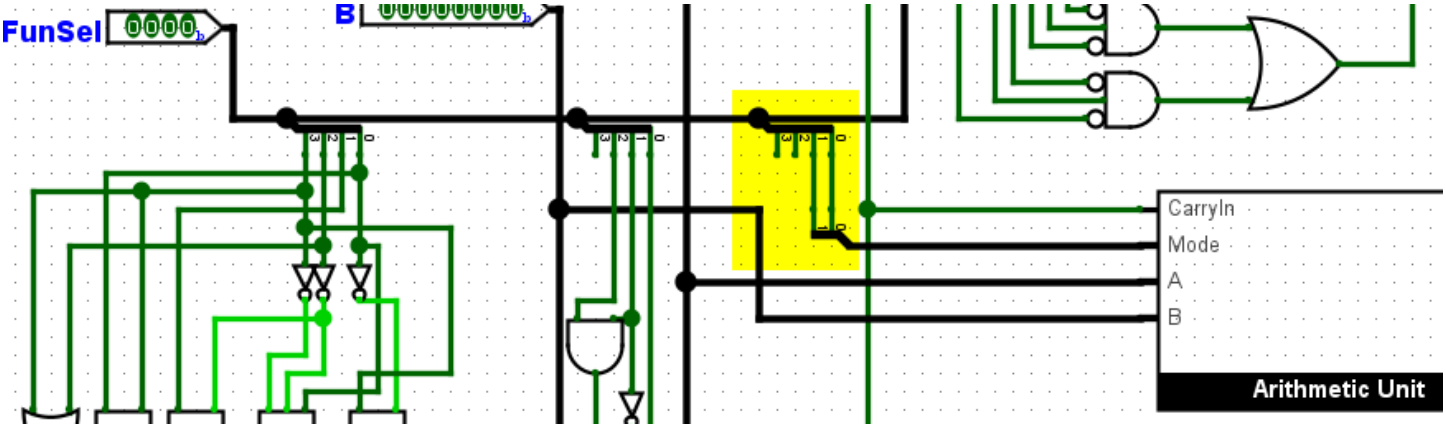
Mapping parts are separately given below.



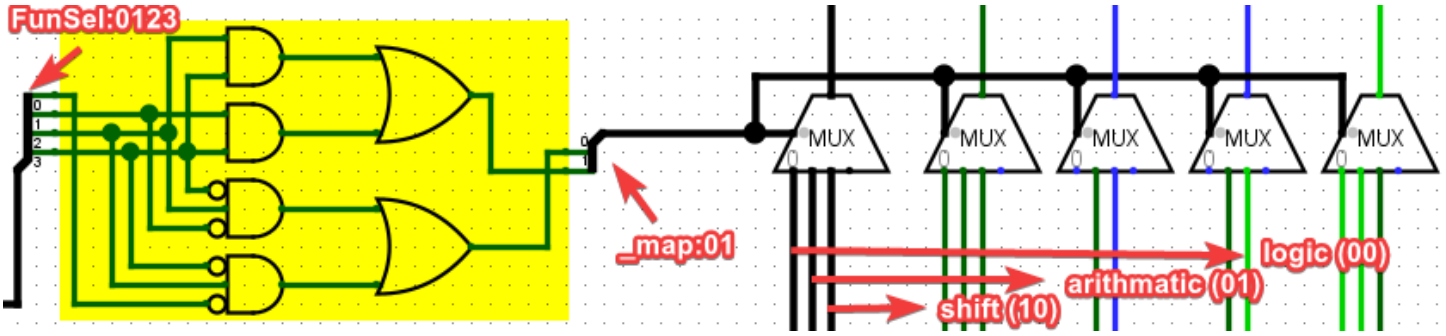
The circuitry highlighted does the **FunSel -> Mode** mapping for **logic unit**. (See Part-1c)



The circuitry highlighted does the **FunSel -> Mode** mapping for **shift unit**. (See Part-1b)



The circuitry highlighted does the **FunSel -> Mode** mapping for **arithmetic unit**. (See Part 1-a)



The circuitry highlighted does the **selection** mapping for **FLAG register i/o** and **ALUOut**. This part of mapping helps which **FLAG** information that the logic units computed to be forwarded into the FLAG register as well as what to feed **ALUOut** with. The highlighted circuitry does the mapping given below in the table.

FunSel	OutALU	_map – Mapped Unit	_map:01
0000 0001	A B	Logic	00
0010 0011	NOT A NOT B	Logic	00
0100 0101 0110	A + B A + B + Carry A – B	Arithmetic	01
0111 1000 1001	A AND B A OR B A XOR B	Logic	00
1010 1011 1100 1101 1110 1111	LSL A LSR A ASL A ASR A CSL A CSR A	Shift	10

Part-1e

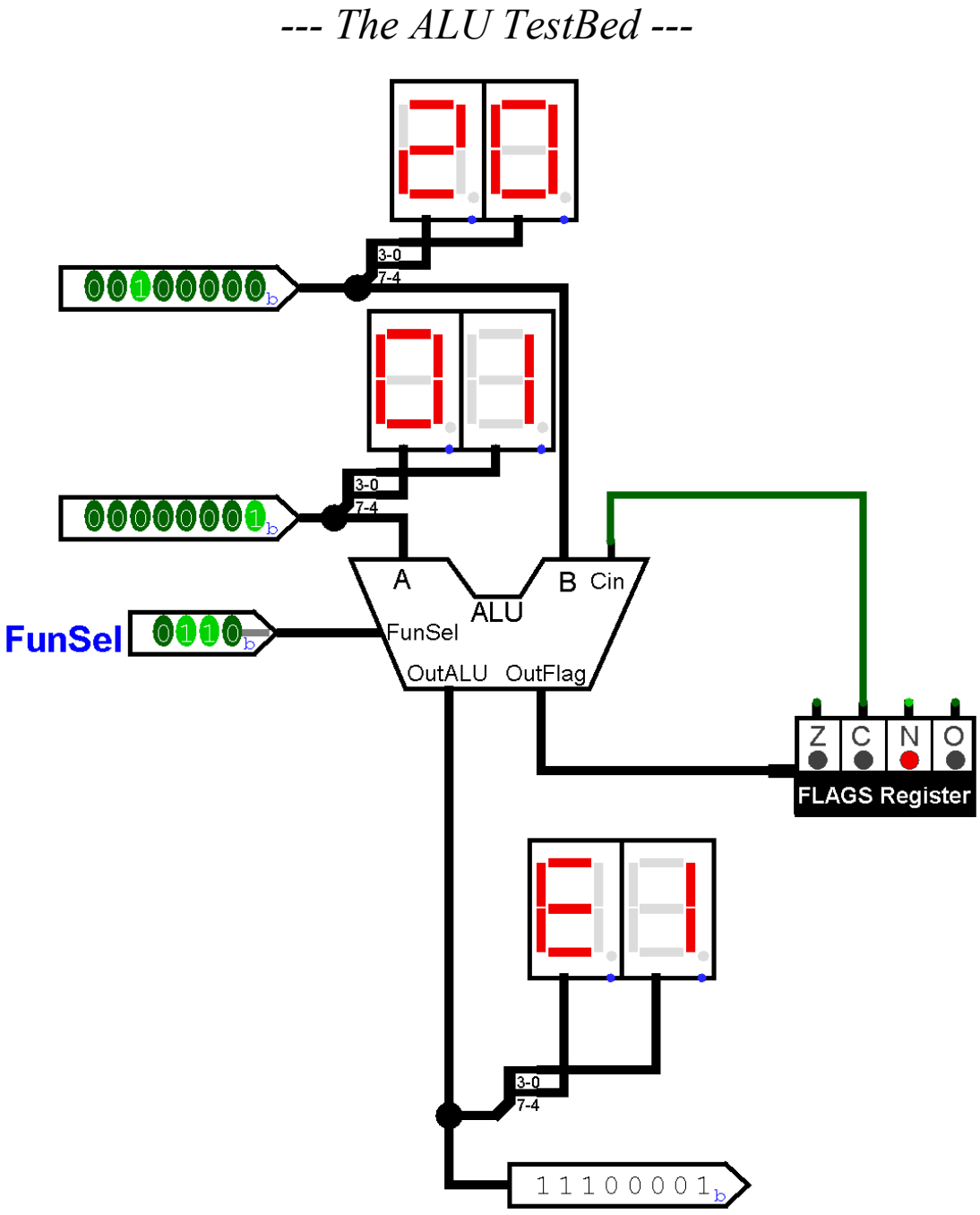


Figure 6 The ALU TestBed circuit.

The ALU TestBed circuit was designed to encapsulate all ALU related circuits into a master circuit to test its behavior.

Part-2

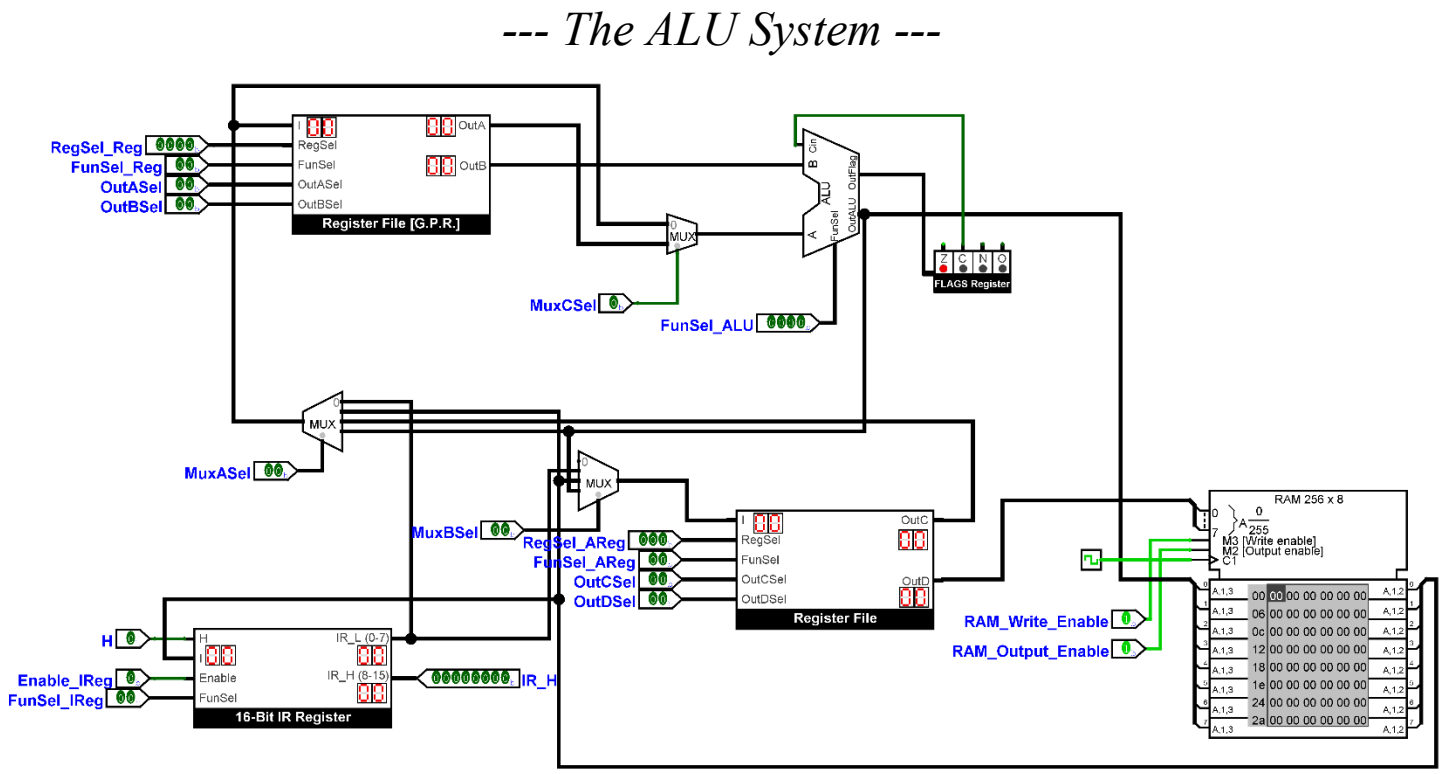


Figure 7 ALU System Implementation

We implemented the ALU system based on the schematic provided with the project documentation. We used components that we designed earlier in Project #1 as well as the current project to design such a circuit as requested.

Tests

Arithmetic Unit

ARITHMETIC LOGIC	Substraction	0110 (Mode 10)						
A	B	AB Decimal	aOut	aOut Decimal	Z	C	N	O
11111111	11111111	1 - 1	00000000	0	1	1	0	0
11111111	00000001	-1 -1	11111110	-2	0	1	1	0
11111101	01111111	-3 - 127	01111110	126	0	1	0	1
01001000	00010000	72 - 16	00111000	56	0	1	0	0
00000101	00000001	5 - 1	00000100	4	0	1	0	0
00000000	11111111	0 - (-1)	00000001	1	0	0	0	0
00000000	00000001	0 - 1	11111111	-1	0	0	1	0
00001111	00011110	15 - 30	11110001	-15	0	0	1	0
10000000	00000001	-128 - 1	01111111	127	0	1	0	1

0100 (mode: 00): In mode 10, the unit makes subtraction with given inputs. According to the system, if aOut is zero Z flag, if negative N flag and if overflow occur O flag opens. If First input (A) is smaller than second

input (B) Carry Output will be 0 according to digital systems operation rules. If First input (A) is bigger than or equal to the second input (B) Carry Output will be 1. And carry input is meaningless for this operation.

A	B	AB Decimal	Carry	aOut	aOut Decimal	Z	C	N	O
11111111	00000001	Calc -1 + 1	0	00000000	0	1	1	0	0
11111111	00000001	Calc -1 + 1	1	00000000	0	1	1	0	0
01111111	00000001	Calc 127 + 1	0	10000000	-128	0	0	1	1
01111111	00000001	Calc 127 + 1	1	10000000	-128	0	0	1	1
01101100	00010011	Calc 108 + 19	0	01111111	127	0	0	0	0
01101100	00010011	Calc 108 + 19	1	01111111	127	0	0	0	0
11001110	01100100	Calc -50 + 100	0	00110010	50	0	1	0	0
11001110	01100100	Calc -50 + 100	1	00110010	50	0	1	0	0
10000000	01111111	Calc -128 + 127	0	11111111	-1	0	0	1	0
10000000	01111111	Calc -128 + 127	1	11111111	-1	0	0	1	0

0100 (mode: 00): In this mode, the unit simply makes addition with given inputs. As seen in the test cases, the difference from 0101 (mode: 01) is that carry bit does not effect the output. We used an half adder and a full adder to separate these modes. In addition, four flags (Z, C, N, O) are enable and working properly.

A	B	A+B Decimal	Carry	aOut	aOut Decimal	Z	C	N	O
00000001	11111111	1 - 1	0	00000000	0	1	1	0	0
00000001	11111111	1 - 1	1	00000001	1	0	1	0	0
11111111	11111101	-1 - 3	0	11111100	-4	0	1	1	0
11111111	11111101	-1 - 3	1	11111101	-3	0	1	1	0
11111111	10000000	-1 - 128	0	01111111	127	0	1	0	1
11111111	10000000	-1 - 128	1	10000000	-128	0	1	1	0
01111111	01010101	127 + 85	0	11010100	-44	0	0	1	1
01010101	11100010	85 - 30	0	00110111	55	0	1	0	0
10001101	00010010	-115 + 18	0	10011111	-97	0	0	1	0

0101 (mode: 01): This mode is the sum of A and B. It adds two 8 bit number and displays the result in aOut.

As we see from the table, when the sum of A and B is bigger than 127 or smaller than -128,

aOut shows a wrong result and O(overflow) flag becomes "1". When aOut becomes 0, Z flag becomes "1".

When the sum of A and B is a 9-bit number, the first bit of that number sets C flag "1".

Shift Unit

InputVar	sOut	Z	C	N	O
11110010	11100100	0	1	1	X
10011011	00110110	0	1	0	X
01010010	10100100	0	0	1	X
10100101	1001010	0	1	0	X

1010 (mode: 000): This mode makes the operation **LSR** (Logical Shift Right). It puts a zero bit to the rightmost side of the InputVar (least significant bit) and gives leftmost bit (most significant bit) as the carry flag.

InputVar	sOut	Z	C	N	O
11110010	01111001	0	x	0	X
10011011	01001101	0	x	0	X
01010010	00101001	0	x	0	X
10100101	01010010	0	x	0	x

1011 (mode: 001): Similar to the mode: 000, this mode makes the operaiton **LSL** (Logical Shift Left). It puts a zero bit to the leftmost side of the InputVar (most significant bit). As a result, flag N always gives the output 0.

inputVar	Carry	sOut	Z	N	O	C
00000001	0	00000000	1	0	x	0
10000000	0	11000000	0	1	x	0
11000011	0	11100001	0	1	x	0
00001111	0	00000111	0	0	x	0
00000001	1	00000000	1	0	x	1
10000000	1	11000000	0	1	x	1
11000011	1	11100001	0	1	x	1
00001111	1	00000111	0	0	x	1

1101 (mode: 011): This mode shifts each digit of a 8-bit number to the right. The leftmost digit remains the same. When sOut is "0", Z becomes "1". When sOut is a negative number, N becomes "1". Overflow is not an issue in this mode.

inputVar	Carry	sOut	Z	N	O	C
01000000	0	10000000	0	1	1	0
10000000	0	00000000	1	0	1	0
10101010	0	11111100	0	0	1	0
00111111	0	01111110	0	0	0	0
00000001	0	00000010	0	0	0	0
01000000	1	10000000	0	1	1	1
10000000	1	00000000	1	0	1	1
10101010	1	11111100	0	0	1	1
00111111	1	01111110	0	0	0	1
00000001	1	00000010	0	0	0	1

1100 (mode: 010): This mode shifts each digit of a 8-bit number to the left. The rightmost digit becomes "0". If first 2 digit of the input are different O becomes "1". When sOut becomes a negative number, N becomes "1". When sOut becomes "0", Z becomes "1".

SHIFT UNIT	CSL	1110 (Mode 100)						
Input	Input Decimal	Carry Input	aOut	aOut Decimal	Z	C	N	O
01111111	127	1	11111111	-1	0	0	1	1
11111111	-1	0	11111110	-2	0	1	1	0
11111111	-1	1	11111111	-1	0	1	1	0
00000000	0	1	00000001	1	0	0	0	0
01000000	64	0	10000000	-128	0	0	1	1
01000000	64	1	10000001	-127	0	0	1	1
10000000	-128	1	00000001	1	0	1	0	1
10000000	-128	0	00000000	1	1	1	0	1

1110 (mode: 100): In the shift unit at the mode "100" (FunSel 1110) CSL(Circular Shift Left) operation will occur. Acording to the system last bit of input (A7) will be carry output and every bit of input will go to the next left position to near it. First bit of input (A0) will be carry input.

SHIFT UNIT	CSR	1111(Mode 101)						
Input	Input Decimal	Carry Input	aOut	aOut Decimal	Z	C	N	O
11111110	-2	1	11111111	-1	0	0	1	0
11111110	-2	0	01111111	127	0	0	0	1
11111111	-1	1	11111111	-1	0	1	1	0
11111111	-1	0	01111111	127	0	1	0	1
00000000	0	1	10000000	-128	0	0	1	1
00000000	0	0	00000000	0	1	0	0	0
00000010	2	1	10000001	-127	0	0	1	1
00000010	2	0	00000001	1	0	0	0	0
00000001	1	1	10000000	-128	0	1	1	1
00000001	1	0	00000000	0	1	1	0	0

1111 (mode: 101): At the mode 101 (FunSel 1111) CSR (Circular Shift Right) operation will occur. According to the system first bit of input (A0) will be carry output and every bit of input will go to the next right position to near it. Last bit of input (A7) will be carry input.

Logic Unit

A	B	A Decimal	B Decimal	IOut	IOut Decimal	Z	N
11111111	00000001	-1	1	11111111	-1	0	1
01101100	00010011	108	19	01101100	108	0	0
00000000	01111111	0	127	00000000	0	1	0
01111111	00000000	127	0	01111111	127	0	0
00000000	11111111	0	-1	00000000	0	1	0

0000 (mode: 000): In this mode, the output is simply equal to **A**. We used a multiplexer to choose a specific output. On the table above, you can see that Input B does not affect the output.

A	B	A Decimal	B Decimal	IOut	IOut Decimal	Z	N
00000000	00000000	0	0	00000000	0	1	0
00000000	00000001	0	1	00000000	0	1	0
00000001	00000000	1	0	00000000	0	1	0
00000001	00000001	1	1	00000001	1	0	0

0111 (mode: 100) : This mode gives the output **A.B**. This output simply obtained using an AND gate and chosen with the multiplexer.

A	B	A Decimal	B Decimal	IOut	IOut Decimal	Z	N
11111111	00000001	-1	1	11111110	-2	0	1
01101100	00010011	108	19	11101100	-20	0	1
00000000	01111111	0	127	10000000	-128	0	1
01111111	00000000	127	0	11111111	-1	0	1
00000000	11111111	0	-1	00000000	0	1	0

0011 (mode: 011): On this mode, output is **NOT B**. Similar to the other modes, desired output was achieved with the help of the multiplexer and a NOT gate.

A	B	IOut	Z	N
00000000	10000000	10000000	0	1
10101010	00000000	00000000	1	0
01010101	10101010	10101010	0	1
10101010	01010101	01010101	0	0

0001 (mode: 001): This mode takes two input A and B, and displays B in IOut. If IOut is "0", Z becomes "1", and if IOut is negative, N becomes "1".

LOGIC UNIT	XOR	1001(Mode 110)			
A	B	aOut	aOut Decimal	Z	N
11111111	11111111	00000000	0	1	0
11111111	00000001	11111110	-2	0	1
00000000	00000000	00000000	0	1	0
01001000	00010000	01011000	88	0	0

1001 (mode: 110): In the logic unit at the mode “110” (FunSel 1001) XOR operation will occur. This mode gives the output A ⊕ B. This output is simply obtained using an XOR gate.

LOGIC UNIT	OR	1000(Mode 101)			
A	B	aOut	aOut Decimal	Z	N
11111111	11111111	11111111	-1	0	1
11111111	00000001	11111111	-1	0	1
00000000	00000000	00000000	0	1	0
01001000	00010000	01011000	88	0	0

1000 (mode: 101): In the logic unit at the mode “101” (FunSel 1000) OR operation will occur. This mode gives the output A + B. This output is simply obtained using an OR gate.

LOGIC UNIT	NOT A	0010(Mode 010)		
A	aOut	aOut Decimal	Z	N
11111111	00000000	0	1	0
00000000	11111111	-1	0	1
01001000	10110111	-73	0	1

0010 (mode: 010): In the logic unit at the mode “010” (FunSel 0010) NOT A operation will occur. This mode gives the output $(A)\overline{\ }$. This output simply obtained using an NOT gate and chosen with the multiplexer.

Conclusion

Throughout the development phase, we did not encounter to a problem. Initially original Logisim caused a lot of problems (mainly display scaling and crash issues) so we decided to proceed with its updated fork Logisim-Evolution.

We designed our circuits from scratch but we tested them thoroughly and got promising results. Therefore we are confident that our implementation adheres to the project design specifications.

In the .circ file, please note that all implementations regarding the second project do contain a prefix of “**proj2_**”. This was done to properly separate implementations which belong to different projects.

In the report, some tables available in the project document are not put (Ex. ALU System MUX tables). It can be assumed that our implementation works according to the information available in those tables since the circuits were designed according to them. We did not put those tables in the report to avoid repetition.