ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E
COMPUTER ORGANIZATION PROJECT REPORT

PROJECT NO	:	2
PROJECT DATE	:	09.04.2020 - 30.04.2020
GROUP NO	:	29

GROUP MEMBERS:

150160802	:	MEHMET CAN GÜN
150170054	:	ZAFER YILDIZ
150180012	:	MUHAMMED SALİH YILDIZ
150180080	:	BAŞAR DEMİR

SPRING 2020

Contents

FRONT COVER

CONTENTS

1 INTRODUCTION **1**

. . . . 1

1.1 MATERIALS 1

2 Applications of Circuits **2**

2.1 PART 1 2

. . . . 2

2.1.1 Calculations 3

. . . . 3

. . . . 3

. . . . 3

. . . . 3

. . . . 4

2.1.2 OutFlags 5

. . . . 5

. . . . 5

. . . . 5

. . . . 6

. . . . 6

2.1.3 Test Cases 8

2.2 PART 2 9

. . . . 9

3 Conclusion **10**

. . . . 10

1 INTRODUCTION

In this project, we are expected to design an Arithmetic Logic Unit which can operate basic calculations on binary numbers. It is responsible for arithmetic calculations on a computer. In the second part, we have designed a circuit that does fundamental operations of the processor.

1.1 MATERIALS

In this project we used following items:

- 8-bit Parallel Adder
- 8-bit Parallel Subtractor
- 16-bit Parallel Adder
- 16-bit Parallel Subtractor
- 8-bit register
- 16-bit register
- D Type Flip-Flop
- Multiplexers
- Splitter
- Hex Digit Display
- Logic Gates
- Tunnel

2 Applications of Circuits

2.1 PART 1

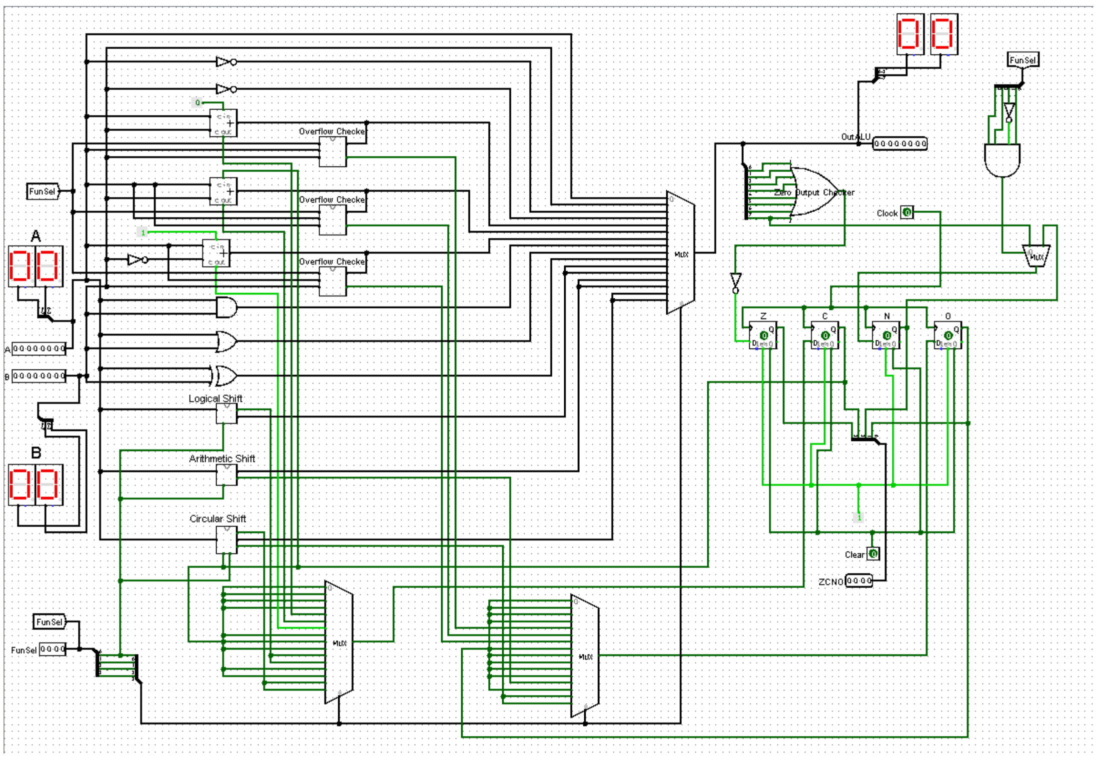
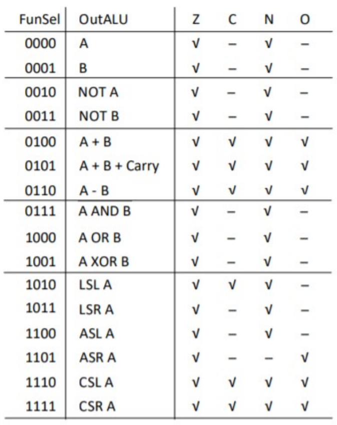


Figure 1: 8-Bit Arithmetic Logic Unit

In this part, we have implemented 8-bit Arithmetic Logic Unit, which operates intended calculations which are given 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	√	√	√	√

Figure 2: FunSel and Operation Table

2.1.1 Calculations

To calculate the result of operations, we implemented different circuit elements for each of them. For every clock cycle, all elements calculate own data output and we choose one of them by a multiplexer that is controlled by FunSel. Finally, the chosen output is given to the output pin.

For the FunSel inputs that are in 0000-1001 interval, we used already implemented Logisim circuit elements. For other inputs, we implemented new circuits which are given below.

- Logic Shift

We designed Logic Shift Unit using several multiplexers. We added left or right selector to determine which type of logic shift is applied. We gave carry as an output and we registered it to the carry Flip-Flop.

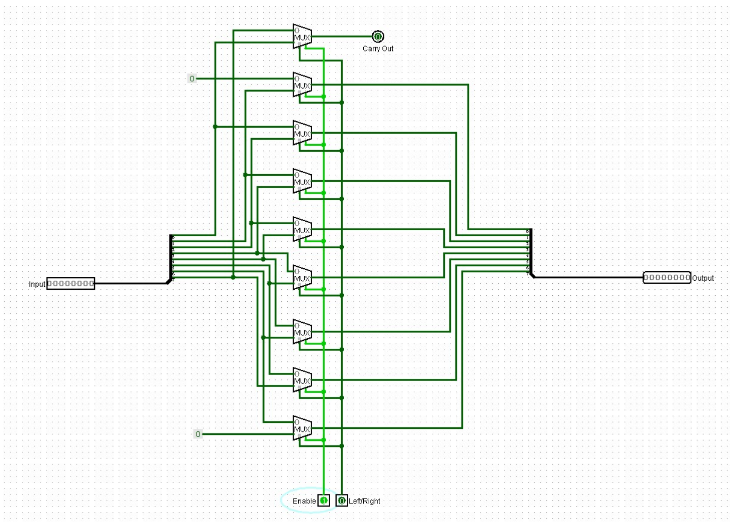


Figure 3: Logic Shift Unit

- Arithmetic Shift

We designed Arithmetic Shift Unit using several multiplexers. We added left or right selector to determine which type of arithmetic shift is applied. To check the overflow, we used XOR to compare the most significant bits of input and output. If there is a difference, this means that there is an overflow.

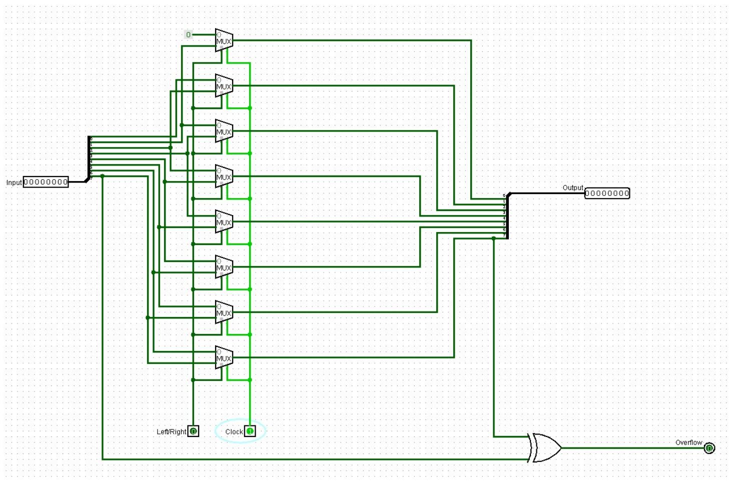


Figure 4: Arithmetic Shift Unit

- Circular Shift

We designed Circular Shift Unit using several multiplexers. We added left or right selector to determine which type of circular shift is applied. In a circular shift, additional input is needed to fill most or least significant bits. We have taken this input from carry register and we update carry register with our new carry out. To check the overflow, we used XOR to compare the most significant bits of input and output. If there is a difference, this means that there is an overflow.

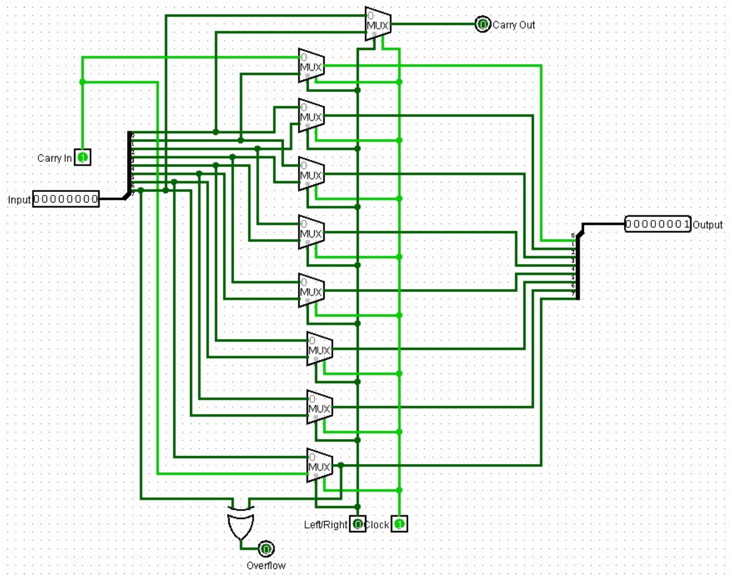


Figure 5: Circular Shift Unit

2.1.2 OutFlags

Storing some properties of calculations and results is important for evaluating output value. Because of this reason zero output, negative output, carry, and overflow flags are integrated into the circuit.

- Zero Output Flag

To determine output is zero or not, we used OR and NOT gate. If all bits of the output is 0, we updated zero flag as 1. On the other hand, if there is at least one bit which is equal to 1, that means the zero flag will be updated as 0, because the result is not equal to 0. We used an 8-bit OR gate whose inputs are bits of the result to implement this logic to the circuit. If there is at least one bit which is equal to 1, that means the result is not 0, but the output of the OR gate is 1. So, we loaded inverse of OR gate output to zero flag.

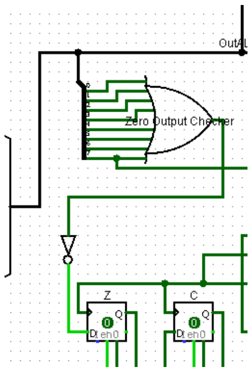


Figure 6: Zero Output Checker

- Negative Output Flag

To determine whether the output is negative or positive, we observed the most significant bit of the output for all operations except ASR operation. We registered the value of the most significant bit to the Negative Output Flag and if the selected operation is ASR, we protected the value in the Negative Output Flag. We used an AND gate and 2:1 MUX which is shown below to implement this logic to the circuit.

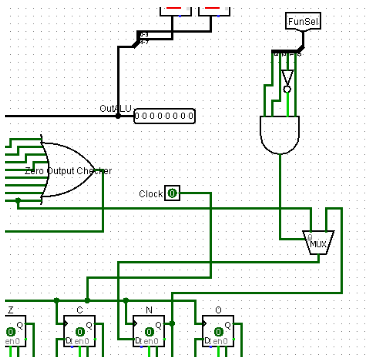


Figure 7: Negative Output Checker

- Carry Output Flag

In some calculations, carry output occurs. To hold and give these values as an output, Carry Output Flag is used. We take all carry-outs of circuit elements and we choose value that is output of intended calculation by FunSel. We connected current carry value which is kept in Carry Flag for calculations that do not affect the result.

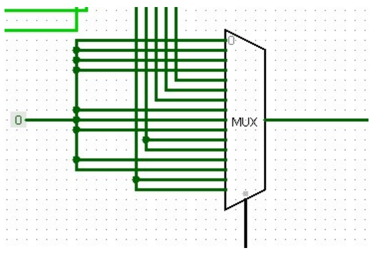


Figure 8: Carry Output Multiplexer

- Overflow Output Flag

In some calculations, overflow occurs. To hold and give these values as an output, Overflow Output Flag is used. We took all overflow outputs of shift operations which can produce overflow as an output. And for arithmetic calculations, we implemented an overflow checker. Then, we connected all these overflow outputs to a MUX whose output is controlled by FunSel. We connected current overflow value which is kept in Overflow Flag for calculations that do not affect the result to protect the current value.

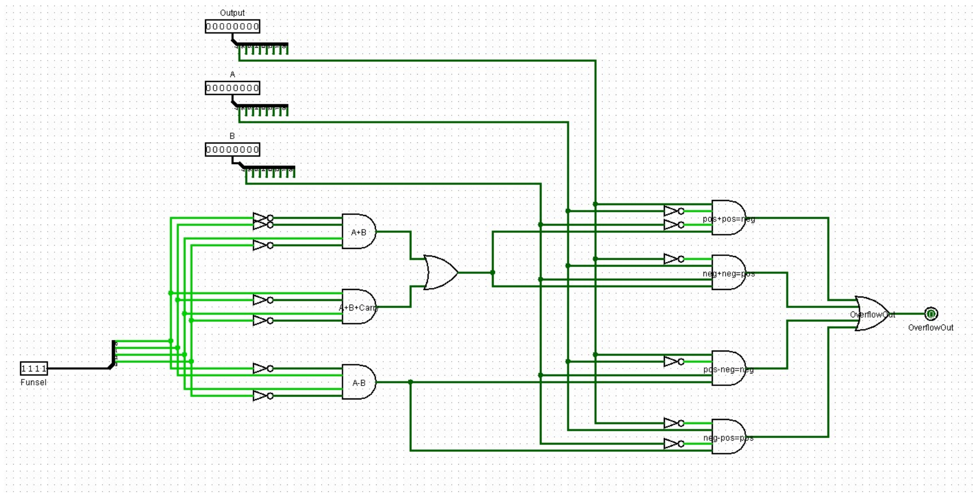


Figure 9: Overflow Checker for Arithmetic Calculations

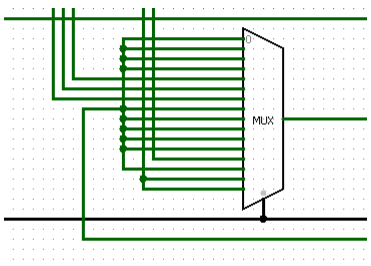


Figure 10: Overflow Multiplexer

2.1.3 Test Cases

Testcase for Functionalities				
Funsel	**A**	**B**	**OutAlu**	**ZCNO**
0000	11010101	01110111	11010101	0010
0001	11010101	01110111	01110111	0000
0010	11010101	01110111	00101010	0000
0011	11010101	01110111	10001000	0010
0100	11010101	01110111	01001100	0100
0101	11010101	01110111	01001101	0101
0110	11010101	01110111	01011110	0001
0111	11010101	01110111	01010101	0001
1000	11010101	01110111	11110111	0011
1001	11010101	01110111	10100010	0011
1010	11010101	01110111	10101010	0111
1010	11010101	01110111	10101010	0111
1011	11010101	01110111	01101010	0101
1011	11010101	01110111	01101010	0101
1100	11010101	01110111	10101010	0110
1100	11010101	01110111	10101010	0110
1101	11010101	01110111	11101010	0110
1101	11010101	01110111	11101010	0110
1110	11010101	01110111	10101011	0110
1110	11010101	01110111	10101011	0110
1111	11010101	01110111	11101010	0110
1111	11010100	01110111	11101010	0010
0000	00000000	11111111	00000000	1000
0011	00000000	11111111	00000000	1000
0101	11111111	01111111	01111111	0101
1101	11111111	01111111	11111111	0101

2.2 PART 2

In this part, we are expected to implement given circuit. This circuit is a simple form of processor without input.

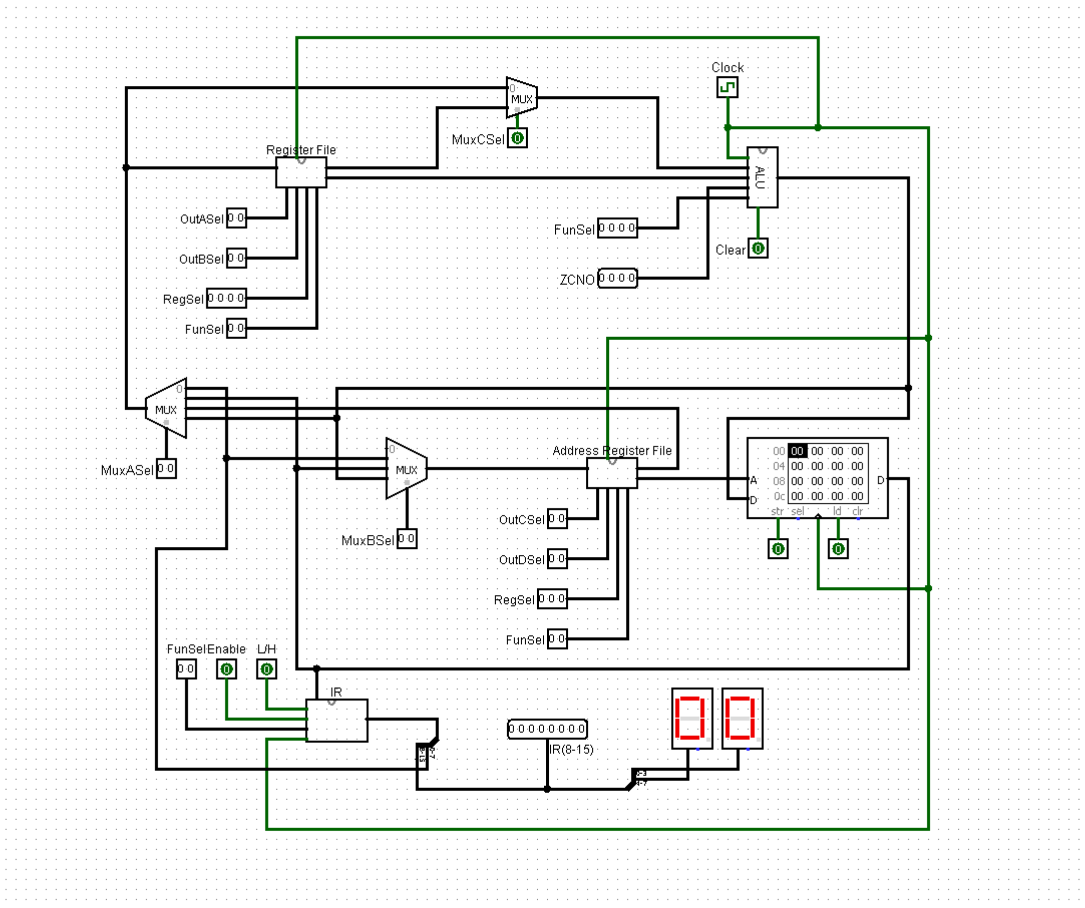


Figure 11: Part 2 Circuit Implementation

Part-2 Test																	
Register				**ALU**	**Address Register**				**IR**			**MUX**			**RAM**	**Case**
RegSel	**FunSel**	**OutASel**	**OutBSel**	**Funsel**	**RegSel**	**FunSel**	**OutCSel**	**OutDSel**	**Funsel**	**Enable**	**L/H**	**ASel**	**BSel**	**CSel**	**S/L**	
0001	01	00	00	0011	000	00	00	00	00	0	0	11	00	00	00	Load R0 with FF
0010	01	00	00	1010	000	00	00	00	00	0	0	11	00	1	00	Load R1 with LSR(R0)
0100	01	01	00	1110	000	00	00	00	00	0	0	11	00	1	00	Load R2 with CSL(R1)
1000	01	10	00	0110	000	00	00	00	00	0	0	11	00	1	00	Load R3 with (R2-R0)
0000	01	00	00	0000	000	00	00	00	00	0	0	11	00	1	10	Store R0 in RAM(0x00)
0000	01	00	00	0000	001	10	00	00	00	0	0	11	00	1	00	RAM points(0x01)
0000	01	01	00	0000	001	10	00	00	00	0	0	11	00	1	10	Store R1 in (0x01) and point to (0x02)
0000	01	10	00	0000	001	10	00	00	00	0	0	11	00	1	10	Store R2 in (0x02) and point to (0x03)
0000	01	11	00	0000	001	10	00	00	00	0	0	11	00	1	10	Store R3 in (0x03) and point to (0x04)
0000	01	10	00	0000	010	01	00	01	00	0	0	11	10	1	01	Load from 0x00 and go to address in it
0000	01	11	00	0110	001	10	00	01	00	0	0	01	00	0	11	Load from 0xFF and substract R0. Store result.
0000	01	11	00	0110	100	01	00	01	11	1	1	01	10	0	01	Load from RAM as an adress to SP and store value in IR(8-15)
0000	01	11	00	0101	000	01	00	10	11	1	0	00	10	0	11	Load from RAM to IR(0-7) and perform A+B+carry operation.Store result

3 Conclusion

In this experiment, we have learned principles of implementing Arithmetic Logic Unit and the key points of mathematical calculations in computer systems. In the second part of the homework, we have learned the relationship between RAM and other circuit elements. We have found a chance to observe, how computers read data from RAM, and after processing this data, how they write the results to the RAM. As a final comment, we had the opportunity to learn and observe the background of calculations and memory relationship in computers.