ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E
COMPUTER ORGANIZATION
PROJECT 1 REPORT

CRN : 21334

LECTURER : Doç. Dr. Gökhan İnce

GROUP MEMBERS:

150210116 : YASİN SAYMAZ

150220031 : ZELİHA MELEK BEKDEMİR

SPRING 2024

Contents

1 INTRODUCTION [10 points]

We started by designing a 16-bit multi-purpose register. This register is controlled by 3-bit control and an enable input. Later, we continued by designing an Instruction Register(IR) that will be used in our system later. By giving an input, We can load either the lower or higher half by giving LH signal. We used our register implementation to develop a Register File system which consist of 8 registers, 4 general purpose registers and 4 scratch registers. İt chooses its outputs depending an given output choosing signals and it also decides what to do based on given function selection signal.

In the third part, we designed an Arithmetic Logic Unit(ALU) that is capable of doing arithmetic, logic and also shifting operations. From given inputs, it provides an output, and, also it updates the FlagOut array.

Finally, we pieced together every individual circuit device to make them into a system. We observed every input and output of the devices and how they connect. Using multiplexers, we created a bus system to keep the system from collapsing or to keep the data from being corrupted.

1.1 TASK DISTRIBUTION

We completed the project by getting together and discussing through every question. All the parts has been constructed after discussions and researches. After finishing general structure, while one of us was dealing with debugging Verilog codes, the other created test conditions for edge cases.

2 MATERIALS AND METHODS [40 points]

2.1 PART 1 / REGISTER

This 16-bit register has 8 functionalites that are controlled by 3-bit control signals which are called **FunSel** and an enable input **E**.

- Inputs
  - I[15:0] = 16-bit Input
  - FunSel[3:0] = 3-bit Control Signal Input. It will be used to decide which operation the resister will be doing.
  - E = Enable. If it is equal to 0, register retains its value. Otherwise, choosen operation is performed.
  - Clock

- Outputs
  - Q[15:0] = 16-bit register output.

Enable input, E, is used to enable or disable the register.

If Enable is 0, this means the system is disabled and does nothing, it retains its value.

E	FunSel	Q^+
0	$\emptyset$	Q (Retain Value)

If Enable is 1, the register can operate depending on the FunSel value.

000 == Q^+ = Q - 1. Next state is the decrementation of the current state.

000 == Q^+ = Q + 1. Next state is the inrementation of the current state.

001 == Q^+ = I. It writes the given input into register.

010 == Q^+ = 0. It writes all zeros into the register.

011 == Q^+ = Q(15 - 8) ¡– Clear, Q(7 - 0). The bits between 15 and 8 are all zeroes, only the remaining 8 bits are same.

100 == Q^+ = Q(7 - 0) ¡– I(7 - 0).The input bits between 0 and 7 are written into the register's bit between 0 and 7.

101 == Q^+ = Q(15 - 8) ¡– I(7 - 0). The input bits between 0 and 7 are written into the register's bit between 15 and 8.

111 == Q^+ = Q(15 - 8) ¡– Sign Extend (I(7)), Q(7 - 0) ¡– I(7 - 0). The input bits between 0 and 7 are written into the register's bit between 7 and 0. The sign is extended.

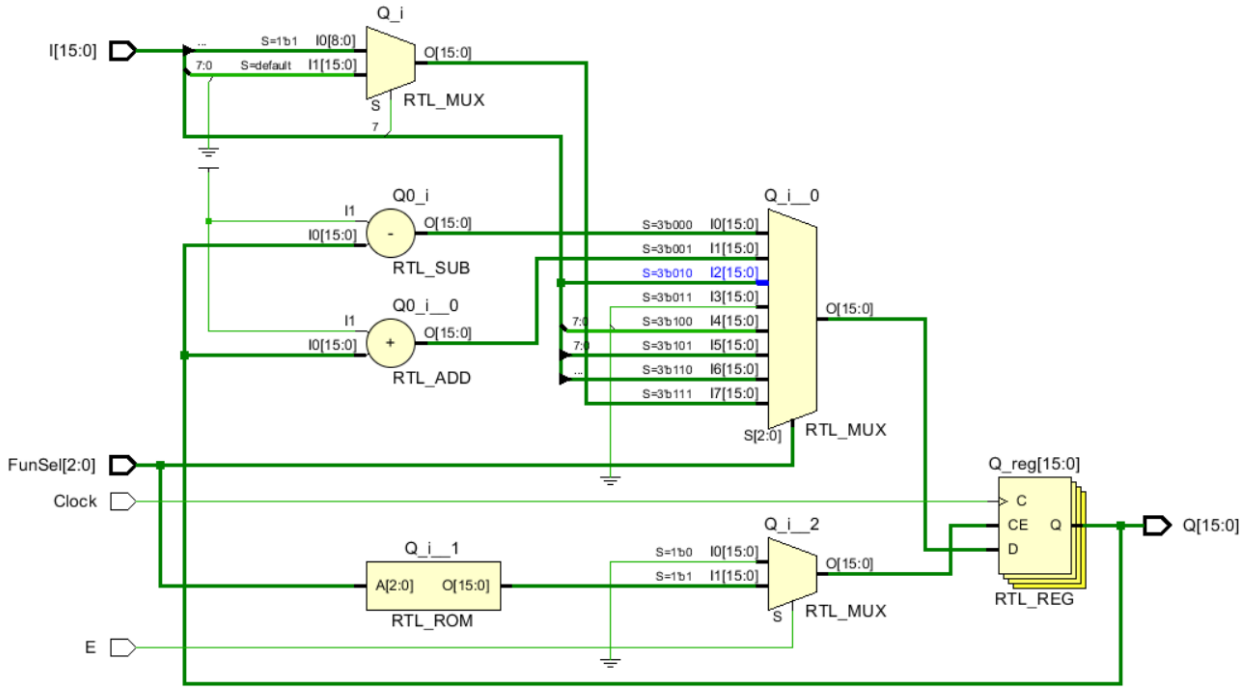


Figure 1: Register Representation

2.2 PART 2

2.2.1 PART 2a / INSTRUCTION REGISTER

To design 16-bit output Instruction Register with 8 bit input is possible by using 8-bit input like bus. According to L'H signal Instruction Register can load both lower(7-0) or higher(15-8) bits.

LH	Write	IR+
∅	0	IR (retain value)
0	1	IR (7-0) ← I (Load LSB)
1	1	IR (15-8) ← I (Load MSB)

To control Instruction Register (IR):

- Inputs
  - I[7:0] = 8-bit Input
  - LH = Low/High signal. It will be used to decide whether the load will be on MSB or LSB bits. If LH=0 I loads for MSB of IR. IR (7-0) ← I (Load LSB) Otherwise, LH=1, I loads for LSB of IR. IR (7-0) ← I (Load LSB)
  - Write = Acts like Enable. If it is equal to 0, IR retains the value. Otherwise IR can loadable.
  - Clock
- Outputs
  - IROut[15:0] = 16-bit IR output.

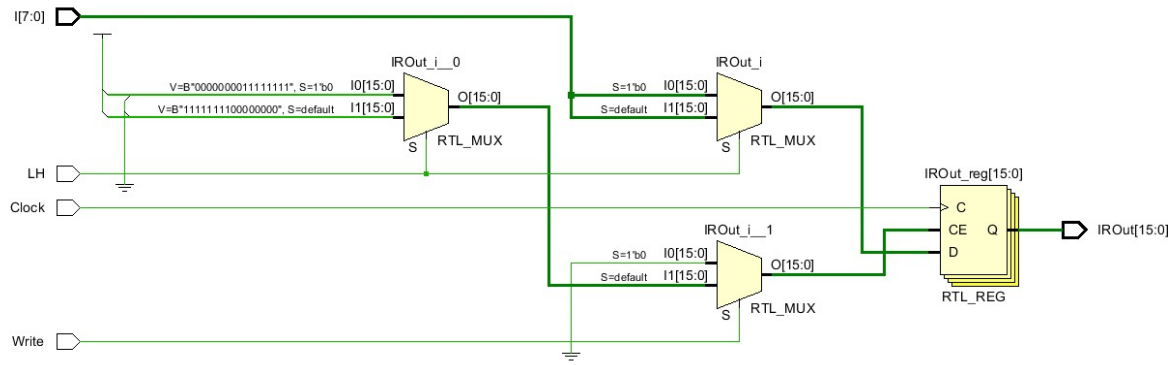


Figure 2: IR Representation

2.2.2 PART 2b / REGISTER FILE

Register File has 2 16-bit outputs, 1 16-bit input and control inputs. According to control inputs, some functions are performed based on input in selected Registers. Register file is a consist of 8 16-bit register that we designed previously. The 4-bit Regsel and ScrSel are signals that select which registers the function selected by the 3-bit FunSel will be applied to. In addition to this selectors, 4-bit OutASel and OutBSel selects the registers which will be assigned to OutA and OutB in order.

- Inputs
  - I[15:0] = 16-bit Input
  - OutASel[2:0] = It selects which registers will be assigned to output OutA. From 000 to 011 is equal R1 to R4 in order. And 100 to 111 is equal to S1 to S4.
  - OutBSel[2:0] = It selects which registers will be assigned to output OutB.
  - FunSel[2:0] = FunSel is function selector. Selects the which function will be performed to selected registers.
  - RegSel[3:0] = RegSel is 4-bit Purpose Register selector. Whether the function to be applied by FunSel will be applied to R1, R2, R3, R4 represents 16 possibilities, corresponding one by one to all possibilities between bits 0000, 1111. 0000 represents that it will be applied to all registers, while 1111 represents that it will not be applied to any registers.
    For ex. 1011 means Only R2 is enabled. R1, R3 ,R4 will be enabled.
  - ScrSel[3:0] = ScrSel is 16-bit Scratch Register selector. It has the same logic as FunSel.
    For ex. 1001 means S2 and S3 are enabled. S1, S4 will be enabled.
- Outputs
  - OutA[15:0] = 16-bit IR output. OutA is directly connected to ALU's input.
  - OutB[15:0] = 16-bit IR output. OutB is also directly connected to ALU's other inputs. We can say that the main inputs of the ALU are basically the outputs from the register file.

We can examine FunSel, which performs the most important function of Register File, in a little more detail.

FunSel	Next State of R
000	R-1 (Decrement)
001	R+1 (Increment)
010	I (Load)
011	0 (Clear)
100	R(15:8) ← (Clear) and R(7:0) ← I(7:0) (Write Low)
101	R(7:0) ← I(7:0) (Only Write Low))
110	R(15:8) R(7:0) ← I(7:0) (Only Write High))
111	R(15:8) ← (Sign Extend (Write 8 times I(7)) and R(7:0) ← I(7:0) (Write Low)

This is also how we implemented Register File:

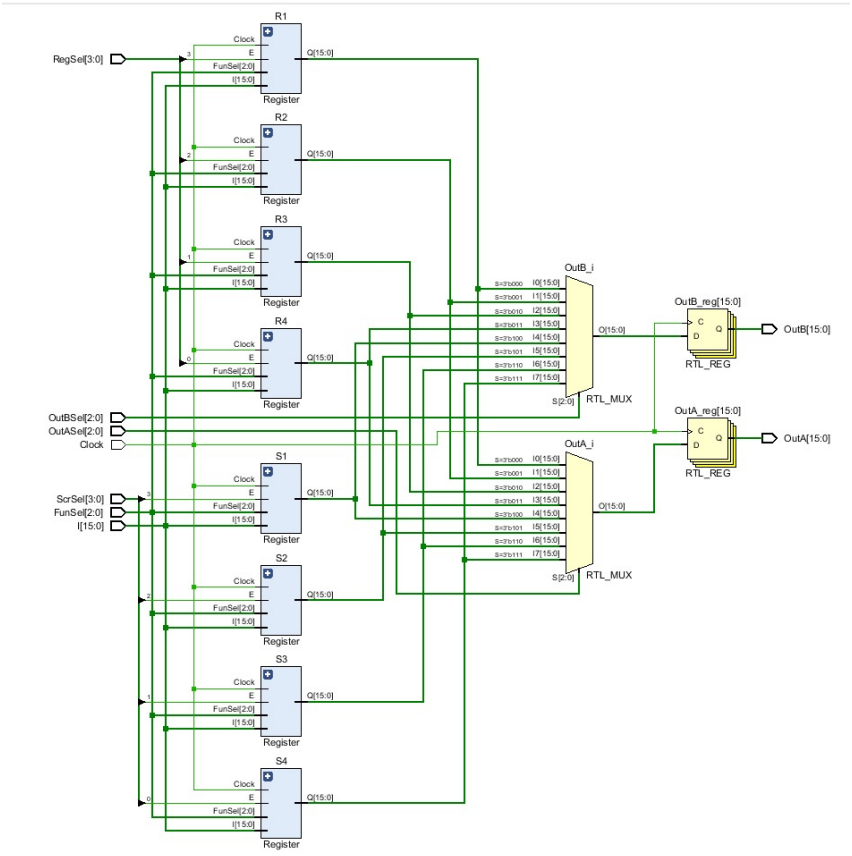


Figure 3: Register File Representation

2.2.3 PART 2c / ADDRESS REGISTER FILE

Adress Register File works in a similar logic to Adress file. The information coming from Input is assigned to 3 16-bit registers that we defined before. The registers we implemented before were receiving funsel as input. Likewise, ARF also has a funsel input. This funsel will be applied with the register selected from RegSel.

Two 16-bit outputs are selected by OutCSel and OutDSel.

Thats how OutCSel and OutDSel selects which registers informations will be assigned to OutC and OutD.

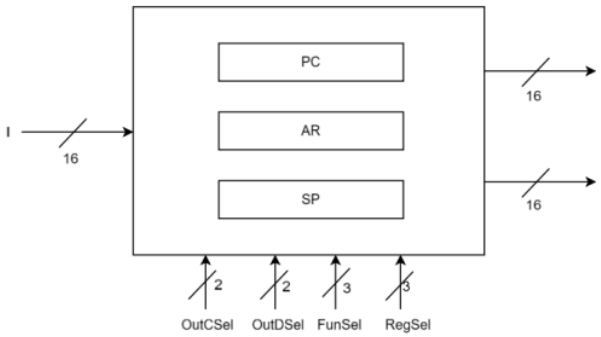


Table 5: OutCSel and OutDSel controls.

OutCSel	OutC
00	PC
01	PC
10	AR
11	SP

OutDSel	OutD
00	PC
01	PC
10	AR
11	SP

Figure 4: Address Register File

Thats the implementation of ARF:

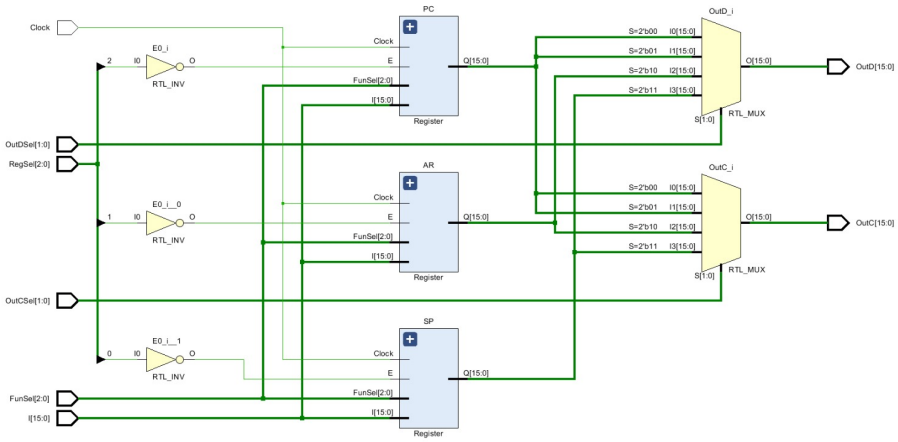


Figure 5: Implementation of Adress Register File

2.3 PART 3

Arithmetic Logic Unit(ALU) has two 16-bit inputs and two outputs as one 16-bit ALU output and one 4-bit output which updates some flags about zero, negative, carry and overflow, respectively Z,C,N,O.

It requires a FunSel signal to choose which operation to perform. It is capable of doing both arithmetic and logic operations.

The general operation of the ALU is shown below:

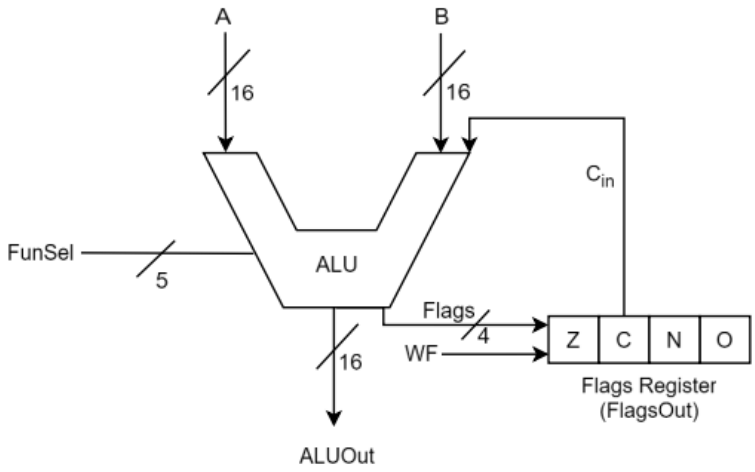


Figure 6: ALU

- Inputs
  - A[15:0] - 16-bit input
  - B[15:0] - 16-bit input
  - FunSel[4:0] - 5-bit control signals
  - WF - 1 bit signal which enables the writing into the FlagsOut
  - Clock
- Outputs
  - FlagsOut[3:0] - 4-bit flags output which keeps the Z,C,N,O values.
  - ALUOut[15:0] - 16-bit operation output
- Wires and Reg Type Wires
  - TEMP, A_TEMP, B_TEMP, NOTB_TEMP, A8, B8, TEMP8, A_TEMP8, B_TEMP8 - makes it easy to manipulate and learn the flag informations

As we said earlier, ALU works according to a given signal called Funsel.

For signals that have 1 in their 5'th bit:

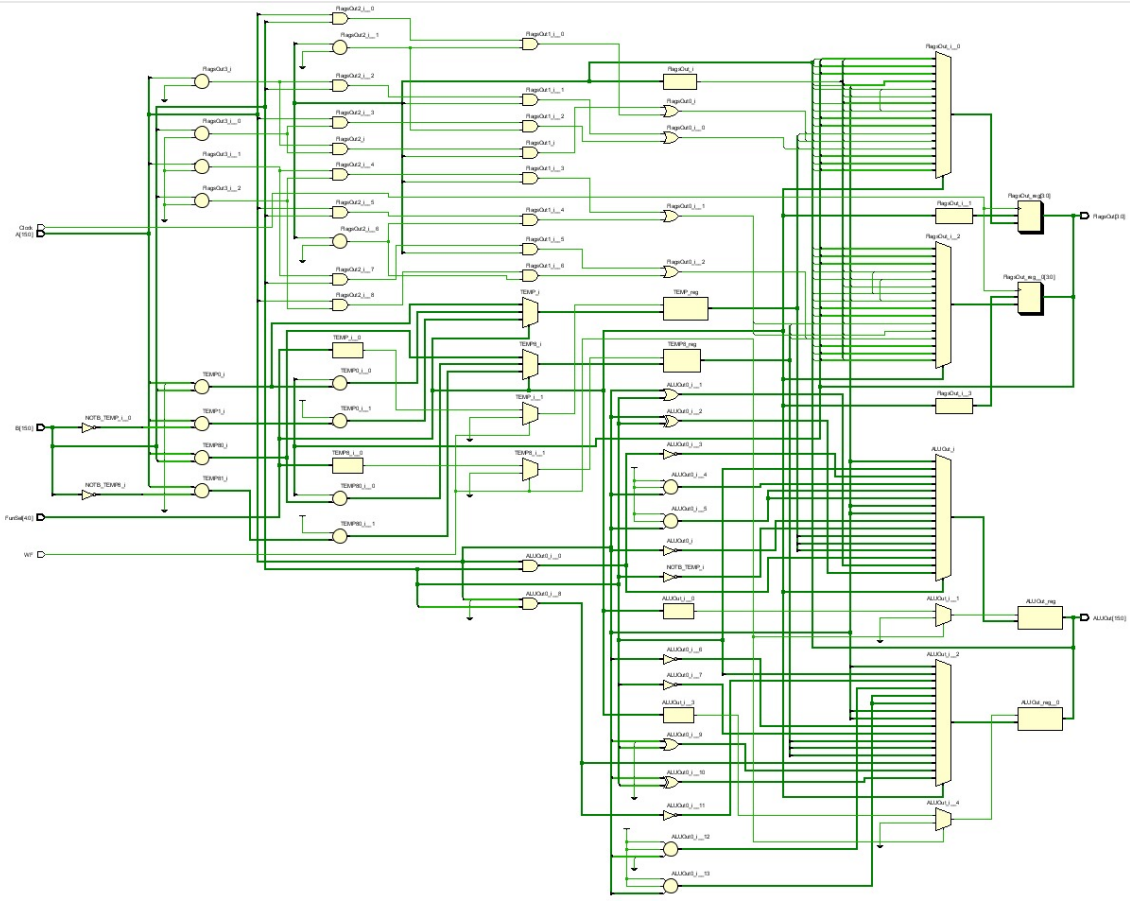
FunSel	ALUOut & ZCNO
10000	**A (16-bit) = Assignment operation is carried out.** • If the number is 0, FlagsOut[3](Z flag) is 1. • This operation does not produce carry nor overflow. • If the number is negative, FlagsOut[1](N flag) is 1.
10001	**B (8-bit) = Assignment operation is carried out.** • If the number is 0, FlagsOut[3](Z flag) is 1. • This operation does not produce carry nor overflow. • If the number is negative, FlagsOut[1](N flag) is 1.
10010	**NOT A (16-bit) = ALU takes the complement of A. 0's turn into 1's, 1's turn into 0's.** • If the number is 0, FlagsOut[3](Z flag) is 1. • This operation does not produce carry nor overflow. • If the number is negative, FlagsOut[1](N flag) is 1.
10011	**NOT B (16-bit) = ALU takes the complement of B. 0's turn into 1's, 1's turn into 0's.** • If the number is 0, FlagsOut[3](Z flag) is 1. • This operation does not produce carry nor overflow. • If the number is negative, FlagsOut[1](N flag) is 1.
10100	**A + B (16-bit) = Summation operation.** • If the number is 0, FlagsOut[3](Z flag) is 1. • This operation may produce carry. After checking, if there is a carry, FlagsOut[2] is updated. • If the number is negative, FlagsOut[1](N flag) is 1. • If two positive inputs are summed and the result is negative or the result of the negative number being summed is positive, there is an overflow. So, FlagsOut[0] is updated.

10101	**A + B + Carry(16-bit) = Summation operation with carry.** • If the number is 0, FlagsOut[3](Z flag) is 1. • This operation may produce carry. After checking, if there is a carry, FlagsOut[2] is updated. • If the number is negative, FlagsOut[1](N flag) is 1. • If two positive inputs are summed and the result is negative or the result of the negative number being summed is positive, there is an overflow. So, FlagsOut[0] is updated.
10110	**A - B (16-bit) = Subtraction operation.** • If the number is 0, FlagsOut[3](Z flag) is 1. • This operation may produce carry. After checking, if there is a carry, FlagsOut[2] is updated. • If the number is negative, FlagsOut[1](N flag) is 1. • If a negative number is subtracted from a positive number and the result is negative or a positive number is subtracted from a negative number and the result is positive, there is an overflow. So, FlagsOut[0] is updated.
10111	**A and B (16-bit) = Logical AND operation. The bits are compared with each other and result is determined according to the rules of AND operation.** • If the number is 0, FlagsOut[3](Z flag) is 1. • This operation does not produce carry nor overflow. • If the number is negative, FlagsOut[1](N flag) is 1.
11000	**A or B (16-bit) = Logical OR operation.The bits are compared with each other and result is determined according to the rules of OR operation.** • If the number is 0, FlagsOut[3](Z flag) is 1. • This operation does not produce carry nor overflow. • If the number is negative, FlagsOut[1](N flag) is 1.

11001	**A xor B (16-bit) = Logical XOR operation.The bits are compared with each other and result is determined according to the rules of XOR operation.** • If the number is 0, FlagsOut[3](Z flag) is 1. • This operation does not produce carry nor overflow. • If the number is negative, FlagsOut[1](N flag) is 1.
11010	**A nand B (16-bit) = Logical NAND operation.The bits are compared with each other and result is determined according to the rules of NAND operation.** • If the number is 0, FlagsOut[3](Z flag) is 1. • This operation does not produce carry nor overflow. • If the number is negative, FlagsOut[1](N flag) is 1.
11011	**LSLA (16-bit) = A is being shifted 1 bit to the left and A[0] is 0.** • After shifting A[15] is written to the FlagsOut[2] as carry. • If the number is 0, FlagsOut[3](Z flag) is 1. • If the number is negative, FlagsOut[1](N flag) is 1. • Overflow will not be observed in this operation.
11100	**LSRA (16-bit) = A is being shifted 1 bit to the right and A[15] is 0.** • After shifting A[0] is written to the FlagsOut[2] as carry. • If the number is 0, FlagsOut[3](Z flag) is 1. • If the number is negative, FlagsOut[1](N flag) is 1. • Overflow will not be observed in this operation.
11101	**ASRA (16-bit) = A is being shifted 1 bit to the right and A[15] is again A[15].** • After shifting A[0] is written to the FlagsOut[2] as carry. • If the number is 0, FlagsOut[3](Z flag) is 1. • If the number is negative, FlagsOut[1](N flag) is 1. • Overflow will not be observed in this operation.

11110	**CSL A (16-bit) = A is being shifted 1 bit to the left and A[0] is replaced with FlagsOut[2] which is carry.** • A[15] is the new carry. • If the number is 0, FlagsOut[3](Z flag) is 1. • If the number is negative, FlagsOut[1](N flag) is 1. • Overflow will not be observed in this operation.
11111	**CSR A (16-bit) = A is being shifted 1 bit to the right and A[15] is replaced with FlagsOut[2] which is carry.** • A[0] is the new carry. • If the number is 0, FlagsOut[3](Z flag) is 1. • If the number is negative, FlagsOut[1](N flag) is 1. • Overflow will not be observed in this operation.

For function signals that start with 0, our inputs are considered as 8 bits. We take the 8 bits starting from LSB and carry the operations out this way. After performing the operations, while writing into ALUOut, we do **sign extension**.
Our result design for Arithmetic Logic Unit System:



2.4 PART 4 / ARITHMETIC LOGIC UNIT SYSTEM

Arithmetic Logic Unit System will be the final part in which we will combine the Register File, Address Register File, Instructor Register, and ALU we have previously made with the help of various MUX's and Memory and turn it into a system.

As we said system works with the help of MUX's and Memory.

MuxA and MuxB have 4 inputs with a 2-bit selector input.

MuxC has a 1-bit input and an 8-bit output and is connected to the output of the ALUout. It selects the ALUOut, which comes as a 16-bit input, as L ([7:0]) if the selector is 0 and H ([15:8]) if it is 1.

ALUOut also connected to MuxA and MuxB as a first input. Seconds input of MuxA and MuxB is OutC which 16-bit output of ARF. Third one is 8-bit MemOut which output of memory. We pad it with 8 zeros to extend 16-bit. And the last inputs of Mux A and B output of Low IROut[7:0] which is output of Instructor Register.

The general operation of the system is shown below:

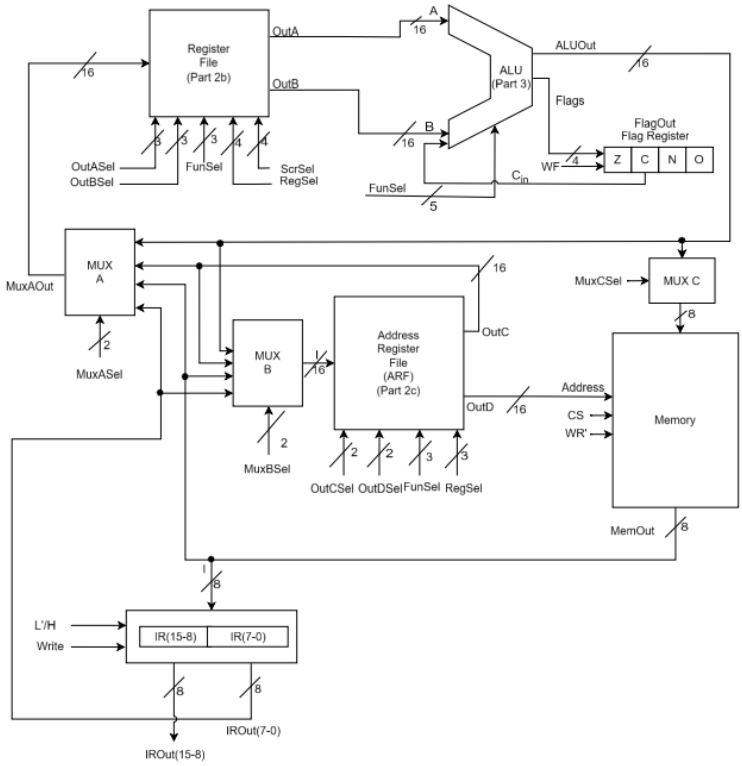


Figure 7: ALU SYSTEM

- Inputs
  - Belong to Register File:
    - RF_OutASel, RF_OutBSel, RF_FunSel, RF_RegSel, RF_ScrSel
  - Belong to Adress Register File:
    - ARF_OutCSel, RF_OutDSel, ARF_FunSel, ARF_RegSel
  - MUX selectors:
    - MuxASel(2-bit), MuxASel(2-bit), MuxASel(1-bit)
  - ALU_WF=
  - IR_LH= Low/High signal for Instructor Register. It will be used to decide whether the load will be on MSB or LSB bits. If LH=0 I loads for MSB of IR. IR (7-0) ← I (Load LSB) Otherwise, LH=1, I loads for LSB of IR. IR (7-0) ← I (Load LSB)
  - IR_Write= Instructor Register write signal. Acts like Enable. If it is equal to 0, IR retains the value. Otherwise IR can loadable.

- Mem_WR= Memory is writable when it equals 1 , otherwise it can not be writeable.
- Mem_CS= Acts like Enable. Chip is enable when CS=0
- Clock

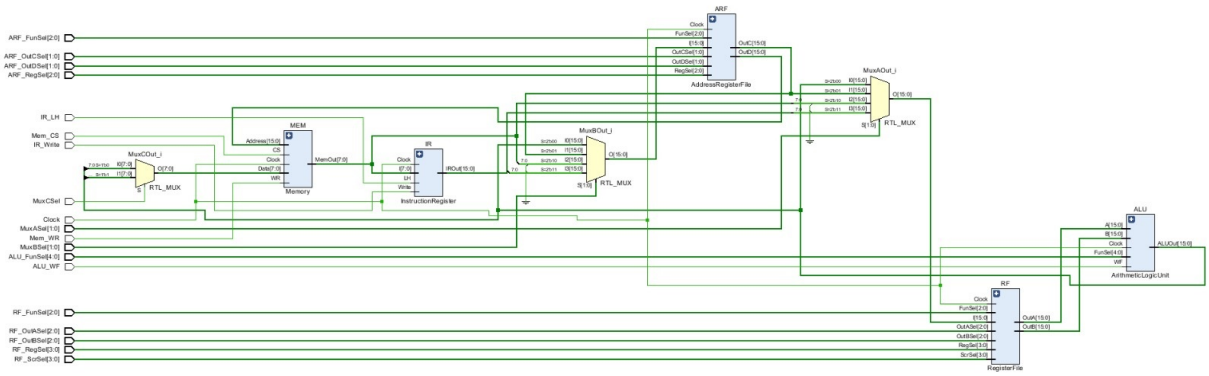
- Wire's And Reg's
  - To carry inputs and bits inside of the ALUSystem from one part the other part we use wires.It's like a physical wire in a circuit, allowing information to flow between different modules or blocks. According the this information we used wire for:

    ALUOut[15:0], OutA[15:0], OutB[15:0], OutC[15:0], OutD[15:0], IROut[15:0], Address[15:0], FlagOut[3:0], Memout[7:0]

    However, on the other hand, we defined the outputs from the MUX as reg. They retain their value till next value is assigned to them. Thus, their use with Memory becomes very functional. So we defines reg's as below:

    MuxAOut[15:0], MuxBOut[15:0], MuxCOut[15:0].

And our result design for Arithmetic Logic Unit System:



3 RESULTS [15 points]

During our experiments, we used test cases to evaluate our codes correctness. These test cases has different scenarios implemented in them that supply different inputs, different function signals and different FlagsOut values. After finishing our codes, by uploading these simulation files to Vivado environment and running the simulation, we obtained some rules. We also tested our results using cmd. It basically does the same thing with these simulations and we saw our passed or failed codes and we realized where we had a mistake and fixed it.

Using the simulations on Vivado rather than cmd has an advantage such that the results are printed as function signals which changes values according to the inputs, flags and clock. We can see the changes in the outputs and flags directly from these results.

Here our results for the project separately:

3.1 PART 1

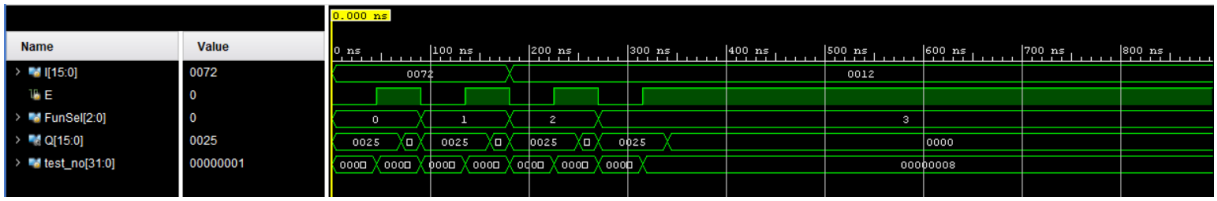


Figure 8: Register

```
----------------------------------
Register Simulation Started
[PASS] Test No: 1, Component: Q, Actual Value: 0x0025, Expected Value: 0x0025
[PASS] Test No: 2, Component: Q, Actual Value: 0x0024, Expected Value: 0x0024
[PASS] Test No: 3, Component: Q, Actual Value: 0x0025, Expected Value: 0x0025
[PASS] Test No: 4, Component: Q, Actual Value: 0x0026, Expected Value: 0x0026
[PASS] Test No: 5, Component: Q, Actual Value: 0x0025, Expected Value: 0x0025
[PASS] Test No: 6, Component: Q, Actual Value: 0x0012, Expected Value: 0x0012
[PASS] Test No: 7, Component: Q, Actual Value: 0x0025, Expected Value: 0x0025
[PASS] Test No: 8, Component: Q, Actual Value: 0x0000, Expected Value: 0x0000
Register Simulation Finished
----------------------------------
```

Figure 9: Register Cmd

3.2 PART 2

3.2.1 PART2-A

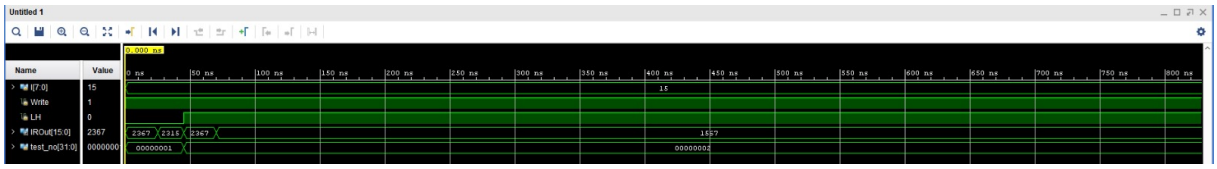


Figure 10: IR

```
------------------------------------
InstructionRegister Simulation Started
[PASS] Test No: 1, Component: IROut, Actual Value: 0x2315, Expected Value: 0x2315
[PASS] Test No: 2, Component: IROut, Actual Value: 0x1567, Expected Value: 0x1567
InstructionRegister Simulation Finished
------------------------------------
```

3.2.2 PART2-B

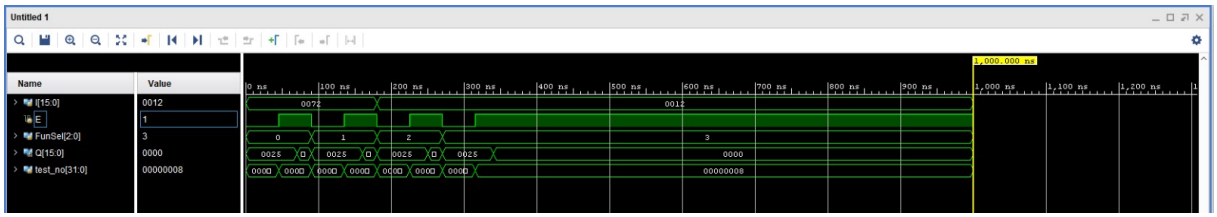


Figure 11: Register File

```
------------------------------------
RegisterFile Simulation Started
[PASS] Test No: 1, Component: OutA, Actual Value: 0x1234, Expected Value: 0x1234
[PASS] Test No: 1, Component: OutB, Actual Value: 0x5678, Expected Value: 0x5678
[PASS] Test No: 2, Component: OutA, Actual Value: 0x1234, Expected Value: 0x1234
[PASS] Test No: 2, Component: OutB, Actual Value: 0x3548, Expected Value: 0x3548
RegisterFile Simulation Finished
------------------------------------
exit
```

3.2.3 PART2-C

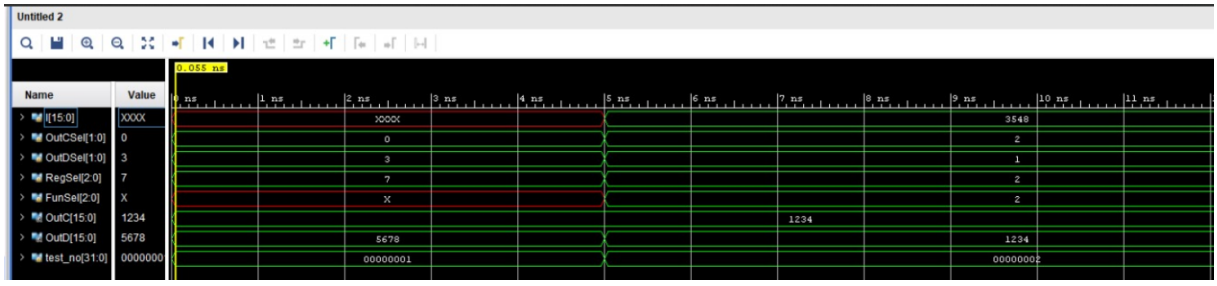


Figure 12: Adress Register File

```
------------------------------------
AddressRegisterFile Simulation Started
[PASS] Test No: 1, Component: OutC, Actual Value: 0x1234, Expected Value: 0x1234
[PASS] Test No: 1, Component: OutD, Actual Value: 0x5678, Expected Value: 0x5678
[PASS] Test No: 2, Component: OutA, Actual Value: 0x1234, Expected Value: 0x1234
[PASS] Test No: 2, Component: OutB, Actual Value: 0x3548, Expected Value: 0x3548
AddressRegisterFile Simulation Finished
------------------------------------
```

3.3 PART 3

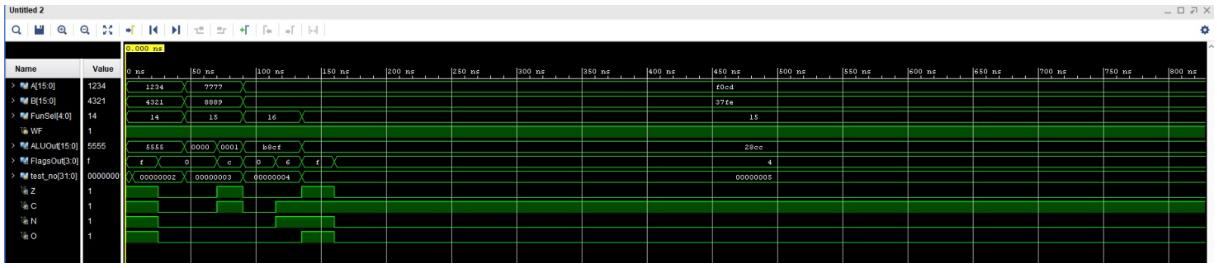


Figure 13: ALU

```
------------------------------------
ArithmeticLogicUnit Simulation Started
[PASS] Test No: 1, Component: ALUOut, Actual Value: 0x5555, Expected Value: 0x5555
[PASS] Test No: 1, Component: Z, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 1, Component: C, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 1, Component: N, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 1, Component: O, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 2, Component: ALUOut, Actual Value: 0x5555, Expected Value: 0x5555
[PASS] Test No: 2, Component: Z, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 2, Component: C, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 2, Component: N, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 2, Component: O, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 3, Component: ALUOut, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 3, Component: Z, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 3, Component: C, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 3, Component: N, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 3, Component: O, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 4, Component: ALUOut, Actual Value: 0xb8cf, Expected Value: 0xb8cf
[PASS] Test No: 4, Component: Z, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 4, Component: C, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 4, Component: N, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 4, Component: O, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 5, Component: ALUOut, Actual Value: 0x28cc, Expected Value: 0x28cc
[PASS] Test No: 5, Component: Z, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 5, Component: C, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 5, Component: N, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 5, Component: O, Actual Value: 0x0000, Expected Value: 0x0000
ArithmeticLogicUnit Simulation Finished
------------------------------------
```

Figure 14: ALU Cmd

3.4 PART 4

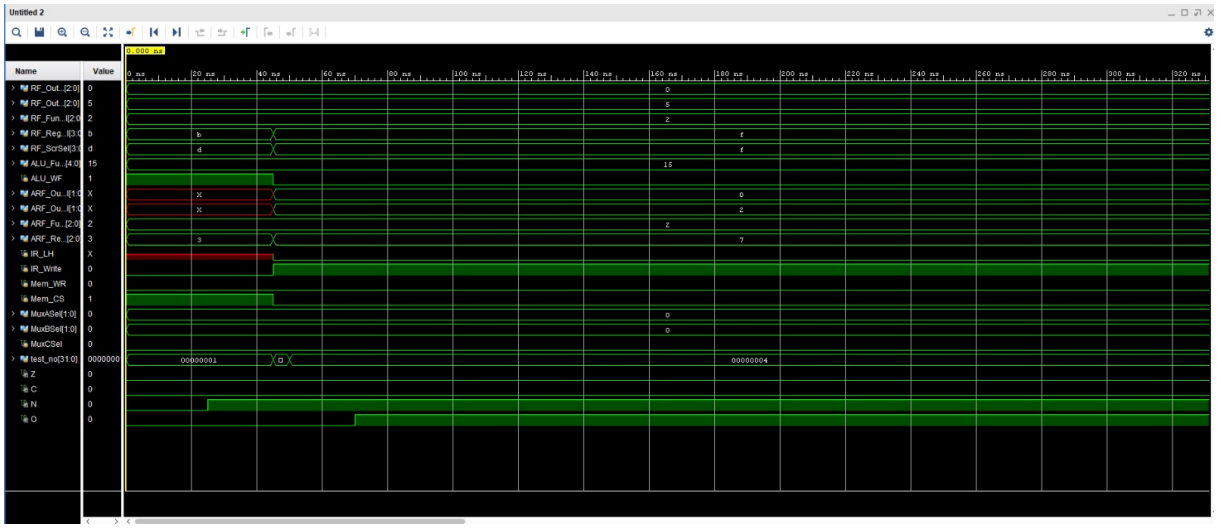


Figure 15: ALU System

```
----------------------------------
ArithmeticLogicUnitSystem Simulation Started
[PASS] Test No: 1, Component: OutA, Actual Value: 0x7777, Expected Value: 0x7777
[PASS] Test No: 1, Component: OutB, Actual Value: 0x8887, Expected Value: 0x8887
[PASS] Test No: 1, Component: ALUOut, Actual Value: 0xfffe, Expected Value: 0xfffe
[PASS] Test No: 1, Component: Z, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 1, Component: C, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 1, Component: N, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 1, Component: O, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 1, Component: MuxAOut, Actual Value: 0xfffe, Expected Value: 0xfffe
[PASS] Test No: 1, Component: MuxBOut, Actual Value: 0xfffe, Expected Value: 0xfffe
[PASS] Test No: 1, Component: MuxCOut, Actual Value: 0x00fe, Expected Value: 0x00fe
[PASS] Test No: 1, Component: R2, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 1, Component: S3, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 2, Component: OutA, Actual Value: 0x7777, Expected Value: 0x7777
[PASS] Test No: 2, Component: OutB, Actual Value: 0x8887, Expected Value: 0x8887
[PASS] Test No: 2, Component: ALUOut, Actual Value: 0xfffe, Expected Value: 0xfffe
[PASS] Test No: 2, Component: Z, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 2, Component: C, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 2, Component: N, Actual Value: 0x0001, Expected Value: 0x0001
[PASS] Test No: 2, Component: O, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 2, Component: MuxAOut, Actual Value: 0xfffe, Expected Value: 0xfffe
[PASS] Test No: 2, Component: MuxBOut, Actual Value: 0xfffe, Expected Value: 0xfffe
[PASS] Test No: 2, Component: MuxCOut, Actual Value: 0x00fe, Expected Value: 0x00fe
[PASS] Test No: 2, Component: R2, Actual Value: 0xfffe, Expected Value: 0xfffe
[PASS] Test No: 2, Component: S3, Actual Value: 0xfffe, Expected Value: 0xfffe
[PASS] Test No: 2, Component: PC, Actual Value: 0xfffe, Expected Value: 0xfffe
[PASS] Test No: 3, Component: OutC, Actual Value: 0x1254, Expected Value: 0x1254
[PASS] Test No: 3, Component: Address, Actual Value: 0x0023, Expected Value: 0x0023
[PASS] Test No: 3, Component: Memout, Actual Value: 0x0015, Expected Value: 0x0015
[PASS] Test No: 3, Component: IROut, Actual Value: 0x0000, Expected Value: 0x0000
[PASS] Test No: 4, Component: OutC, Actual Value: 0x1254, Expected Value: 0x1254
[PASS] Test No: 4, Component: Address, Actual Value: 0x0023, Expected Value: 0x0023
[PASS] Test No: 4, Component: Memout, Actual Value: 0x0015, Expected Value: 0x0015
[PASS] Test No: 4, Component: IROut, Actual Value: 0x0015, Expected Value: 0x0015
ArithmeticLogicUnitSystem Simulation Finished
----------------------------------
```

4 DISCUSSION [25 points]

In our project, we used modules to define the circuit elements. We determined the necessary parameters that these elements take and defined them respectfully. Register uses function signals to operate on positive edge of clock, so, we used **always(posedge Clock)** block to ensure that. We used **case()** block to evaluate different scenarios of FunSel.

For the second part, first we designed a specific purpose IR similar to the register we designed earlier. We used assignments in case scenarios. Then, by calling the **Register** module in **RegisterFile** module, we implemented a system that has different output channels and different registers. According to given functions, a register is chosen and some operations are performed and transferred to output wires. We designed this module in **always(*)** and give the same clock to the registers we called so that the operations are done according to the clock but function and input giving can be done in any moment. Even though, it is possible to change inputs anytime, since registers works with clock, operations are not effected.

Address Register File uses the same system. It consist 3 registers we call with parameters and according to the functions we choose which registers output is displayed.

ALU has 4 **always()** blocks but 2 of them works with ***** and two of them works with **posedge Clock**. The ones using the * are the ones that are irrelevant from the clock. The operations are done under this clock. Posedge clock block is for flag writing. The system waits for clock to write Z,C,N,O values to the FlagsOut.

It checks if the number is zero for Z flag, and if it is zero, it writes 1 to the flag.

It checks the n+1'th bit for carry, if it is 1, then there is a carry and it is written into the part of carry part of FlagsOut as 1.

It checks if the number is negative by looking at the nth bit, and if it is negative, then 1 is written into the N flag.

Overflow is occurred under certain conditions such as **pos - neg = neg — neg - pos = pos — pos + pos = neg — neg + neg = pos**. İf any of these conditions are satisfied, O flag is one. For this conditions we check the sign of the numbers and sign of the result and see if the situation fits any of the above.

Lastly, we connected our modules by identifying every input and output and how they connect. We called our modules in our code by giving them parameters. We were careful about input and output parameters. And we use function signals for multiplexers to avoid using more than one module or data at a time.

5 CONCLUSION [10 points]

Throughout this assignment, we learned how to start from the most basic structures of Verilog, progress to ALU design and turn it into a system.

In each design, we created an even more complex system by using the structures we created before and stacking them on top of each other. First, we learned the Register design and the Instruction Register design using this design. Then, we created a system by grouping the Registers we created in the Register File and Address Register Files we would install into blocks. Then, with the help of additional structures and MUXes, we designed the Arithmetic Logic Unit and finally the overall ALUSystem.

Despite all this, there were a few challenges we encountered and a few things we learned while doing the project. For example:
We learned difference between *wire* and *reg* keywords. While *wire* can be treated as physical wires, like they can be read or assigned and no values get stored in them, *reg* can store data. They retain their value till next value is assigned to them.

Moreover, we learned while creating a large piece system, we learned how to break it down into smaller pieces and start from there, for example we used Helper, Register, IR register etc. modules while building more complex systems from small parts.

Additionally, we understood how clock works and what is the difference between
always @(posedge Clock)
always @(*)
these two clocks. While @posedge Clock represent the always at the positive edge of the clock that we implemented in a Helper.v, @(*) blocks are used to describe Combinational Logics.