150117509 - Mustafa Abdullah Hakkoz
150117052 - Berk Yıldız
150118508 - Çağrı Hodoğlugil
150115841 - Erman Kundakçıoğlu

04.01.2020

DIGITAL LOGIC DESIGN PROJECT REPORT

- ## Subject

In this project, we are supposed to design a basic processor which has the 18 bits address width, again 18 bits data width, 16 registers and supports 16 instructions: ADD, AND, OR, XOR, ADDI, ANDI, ORI, XORI, LD, ST, JUMP, BEQ, BLT, BGT, BLE, BGE.

- ADD, AND, OR and XOR have the same form:
  OPCODE DST,SRC1,SRC2
  They do corresponding operation(sum, add, or, xor) on 2 values that fetched from source registers(SRC1, SRC2) and write the result to destination register(DST).

- ADDI, ANDI, ORI and XORI have the same form:
  OPCODE DST,SRC1,IMM
  They resemble to previous instructions with 1 difference. Instead of using second source register, they use an immediate value that comes from input. Rest is same; they do corresponding operation(sum, add, or, xor) on 2 values that fetched from source register(SRC1) and immediate value (IMM), then write the result to destination register(DST).

- Structure of JUMP instruction is:
  JUMP ADDR
  Jump instruction sets Program Counter (PC) to the given value (ADDR). ADDR is supposed to be maximum width available so we set it to 15 bits. ADDR is relative address and can be negative or positive.

- Structure of LD instruction is:
  LD DST,ADDR
  LD instruction loads a value from data memory (ADDR) in to destination register (DST).

- Structure of STR instruction is:
  ST SRC,ADDR
  STR instruction stores a value from source register (SRC) to data memory (ADDR).

- Branch operations have the same form:
  OPCODE OP1,OP2,ADDR
  They compares 2 operands that can be fetched from registers (OP1,OP2) and sets PC to the relative address (ADDR), if result is true. Branch operations also sets binary n,z,p values for corresponding operations.

- **Part – 1: ISA and Assembler**

  For the first iteration of the project, we were supposed to design an ISA table and optimize IMM and ADDR sizes to the maximum available. So we submitted a table *(“ISA.xlsx”)* like that:

	OPCODES																	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
ADD	0	0	0	0	DST				SRC1				SRC2					
ADDI	0	0	0	1	DST				SRC1				IMM					
AND	0	0	1	0	DST				SRC1				SRC2					
ANDI	0	0	1	1	DST				SRC1				IMM					
OR	0	1	0	0	DST				SRC1				SRC2					
ORI	0	1	0	1	DST				SRC1				IMM					
XOR	0	1	1	0	DST				SRC1				SRC2					
XORI	0	1	1	1	DST				SRC1				IMM					
JUMP	1	0	0	0	ADDR													
LD	1	0	0	1	DST				ADDR									
ST	1	0	1	0	SRC				ADDR									
BEQ	1	0	1	1	n	z	p		OP1				OP2				ADDR	
BLT	1	1	0	0	n	z	p		OP1				OP2				ADDR	
BGT	1	1	0	1	n	z	p		OP1				OP2				ADDR	
BLE	1	1	1	0	n	z	p		OP1				OP2				ADDR	
BGE	1	1	1	1	n	z	p		OP1				OP2				ADDR	

But we figured out that we could have done more optimization on it like lowering opcode bits from 4 to 3 with cost of adding decider bit to ALU operations. But in the end, we had an extra bit for addresses. So we updated our ISA *(“ISA-updated.xlsx”)* and relevant parts of Assembler *(“AssemblerUpdated.java”)* in our final submit. Our final instruction design is:

	OPCODES																	
	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD	0	0	0	DST REG				SRC1 REG				0	SRC2 REG				*00*	
ADDI	0	0	0	DST REG				SRC1 REG				1	IMM VALUE					
AND	0	0	1	DST REG				SRC1 REG				0	SRC2 REG				*00*	
ANDI	0	0	1	DST REG				SRC1 REG				1	IMM VALUE					
OR	0	1	0	DST REG				SRC1 REG				0	SRC2 REG				*00*	
ORI	0	1	0	DST REG				SRC1 REG				1	IMM VALUE					
XOR	0	1	1	DST REG				SRC1 REG				0	SRC2 REG				*00*	
XORI	0	1	1	DST REG				SRC1 REG				1	IMM VALUE					
JUMP	1	0	0	PC RELATIVE ADDR														
LD	1	0	1	DST REG				DATA MEMORY ADDR										
ST	1	1	0	SRC REG				DATA MEMORY ADDR										
BEQ	1	1	1	0	1	0		OP1 REG				OP2 REG					PC RELATIVE ADDR	
BLT	1	1	1	1	0	0		OP1 REG				OP2 REG					PC RELATIVE ADDR	
BGT	1	1	1	0	0	1		OP1 REG				OP2 REG					PC RELATIVE ADDR	
BLE	1	1	1	1	1	0		OP1 REG				OP2 REG					PC RELATIVE ADDR	
BGE	1	1	1	0	1	1		OP1 REG				OP2 REG					PC RELATIVE ADDR	
				n	z	p												

And for our assembler, we write a script in java language. You can find it in our final submission as *“AssemblerUpdated.java”.*

- In our main method, we read input file line by line, split it to words and send the resulting word array to strToBinary() method. After converting instructions to first binary, then hex, we extended string to 5 characters if it’s lower than 5 and finally write it to output file.

```
//write hexadecimal results to output file
try {
    FileWriter fileWriter = new FileWriter("output.txt");
    fileWriter.write("v2.0 raw\n");
    //get lines from inputList and clean them
    for (int i = 0; i < inputList.size(); i++) {
        String[] cleanStr = inputList.get(i).toUpperCase().split("[\\s,]+");
        //convert the string to binary then to decimal
        String binString = strToBinary(cleanStr);
        int decString = Integer.parseInt(binString, 2);
        //Now convert decimal to hexadecimal
        String hexString = Integer.toString(decString, 16).toUpperCase();
        //extend hexString to 5 digit
        int len=hexString.length();
        if(len<5)
            for (int j = 0; j < 5-len; j++)
                hexString = "0"+hexString;
        System.out.printf("%d. line of input: \"%s\", binary form: \"%s\", hex form (output): \"%s\"\n", i+1, i
        //Write the result in a txt file
        fileWriter.write(hexString+"\n");
    }
    fileWriter.close();
}
```

- In our strToBinary() method, we implemented all cases for opcodes in switch-case and send them to relevant methods.

```
public static String strToBinary(String[] str){
    String binaryStr;

    switch (str[0]) {
        case "ADD":
        case "AND":
        case "OR":
        case "XOR":
            binaryStr = arithmeticOps(str);
            break;
        case "ADDI":
        case "ANDI":
        case "ORI":
        case "XORI":
            binaryStr = arithmeticOps_Imm(str);
            break;
        case "JUMP":
            binaryStr = jump(str);
            break;
        case "LD":
        case "ST":
            binaryStr = load_store(str);
            break;
        case "BEQ":
        case "BLT":
        case "BGT":
        case "BLE":
        case "BGE":
           binaryStr = branchOps(str);
            break;
        default:
            throw new IllegalArgumentException("Error - Invalid Instruction");
    }
    return binaryStr;
```

- For example for arithmetic operations, we produced registers for dest, src1 and src2 then returned a string that contains them in proper structure.

```
//////////////////
//ARITHMETIC OPS//
//////////////////
public static String arithmeticOps(String[] str){
    String opCode = "";
    String result;

    switch (str[0]){
        case "ADD":
            opCode = "000";
            break;
        case "AND":
            opCode = "001";
            break;
        case "OR":
            opCode = "010";
            break;
        case "XOR":
            opCode = "011";
            break;
        default:
    }

    //construct binary result
    String dest = produceRegister(str[1]);
    String src1 = produceRegister(str[2]);
    String src2 = produceRegister(str[3]);
    result = opCode + dest + src1 + "0" + src2 + "00";
    return result;
}
```

- In produceRegister() methods we removed the letter “R”, converted string from decimal to binary and extend it to 4. These are the corresponding methods:

```java
//make valid register codes
public static String produceRegister(String str){
    String validRegister = removeLetter_R(str);   //remove letter R
    validRegister = decToBin(validRegister);    //convert dec to bin
    validRegister = extend(validRegister, limit: 4);  //extend to 4 bits
    return validRegister;
}

//remove letter "R" to convert register name to a relevant number
public static String removeLetter_R(String str) {
    String number = str.replaceAll("R", "");
    return number;
}

//converts decimal to binary
public static String decToBin(String str) {
    int integer = Integer.parseInt(str);
    String binary = Integer.toBinaryString(integer);
    return binary;
}

//extends given binary string to a limit
public static String extend(String str,int limit) {
    int len = str.length();
    if (len < limit)
        for (int i = 0; i < limit-len; i++)
            str = "0"+str;
    else if (limit < len) {          //if len > limit (overflow),
        str = str.substring(len-limit,len);
    }
    return str;
}
```

- Rest of the opcodes are quite similar to above only with minor differences. For example we used produceImmediate() and produceAddress() methods too, when we need them. In these methods, we removed characters “#” and “H” if used, to support different synthaxes for immediate values.

```
//make valid immediate codes
public static String produceImmediate(String str){
    String validImmediate, validImmediate2;
    String validImmediate_bin="", extended;

    if (str.contains("#") || str.contains("H")){
        if (str.contains("#") && !str.contains("H")){          //i.e. ADDI, R01, #12
            validImmediate = removeSymbol_Hash(str);           //remove symbol "#" the
            validImmediate_bin = decToBin(validImmediate);
        }
        if(str.contains("H") && !str.contains("#")){            //i.e. ADDI, R01, 12H
            validImmediate = removeLetter_H(str);               //remove letter "H" th
            validImmediate_bin = hexToBin(validImmediate);
        }
        if (str.contains("#") && str.contains("H")){           //i.e. ADDI, R01, #12H
            validImmediate = removeSymbol_Hash(str);           //remove symbols "#" an
            validImmediate2 = removeLetter_H(validImmediate);
            validImmediate_bin = decToBin(validImmediate2);
        }
    }
    else{
        validImmediate_bin = decToBin(str);             //if string includes neither "H
    }
    extended=extend(validImmediate_bin, limit: 6);
    return extended;      //extend to 6 bits
}
```

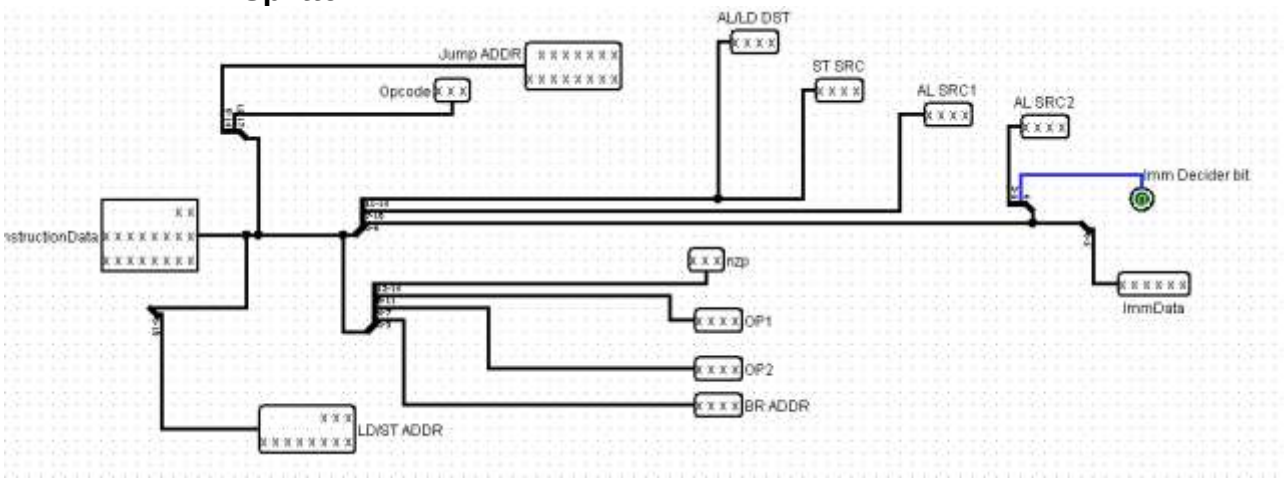
- When we run the code with the file *("input.txt")* in final submit folder we had the these results *("InstructionMemory.txt"):*

	v2.0 raw
ADD R4,R7,R1	02384
LD R0,#13	2800D
LD R1,9	28809
LD R3,2	29802
LD R4,6	2A006
LD R5,3	2A803
LD R6,19	2B013
LD R7,11H	2B811
LD R8,1BH	2C01B
LD R9,13	2C80D
ADD R3,R5,R7	01A9C
BGT R7,R8,1	39781
BLE R1,R0,-1	3E10F
JUMP 2	20002
ADDI R0,R0,-3	0007D
ANDI R1,R2,5	08945
BLT R3,R4,2	3C342
ST R1,2	30802
XORI R2,R3,8	191C8
BEQ R2,R6,1	3A261
ST R3,5	31805
ADDI R5,R5,25	02AD9
ADDI R2,R4,#25	01259
BGE R2,R0,1	3B201
AND R5,R4,R6	0AA18
OR R6,R1,R3	1308C
XOR R7,R3,R2	1B988
ADD R7,R5,R6	03A98
JUMP -8	27FF8

• Part – 2: Logisim

In this part, we designed our processor in Logisim software. We had 8 main components: A **register file** unit to hold 16 registers, a read only **instruction memory** to hold hex instructions, a read-write **data memory** to hold to store data, an **arithmetic logic unit (ALU)** to compute arithmetic operations, a **branch** unit to make comparison operations, a **program counter (PC)** to decide next intruction, a **splitter** to split instruction string in to words with all kind of possibilities and finally a **control unit** to produce signals for all datapath components. Besides them, we used sign and zero **extenders** and several multiplexers.

- **Splitter:**



To make everything simpler we defined a component named “Splitter” in the control unit.

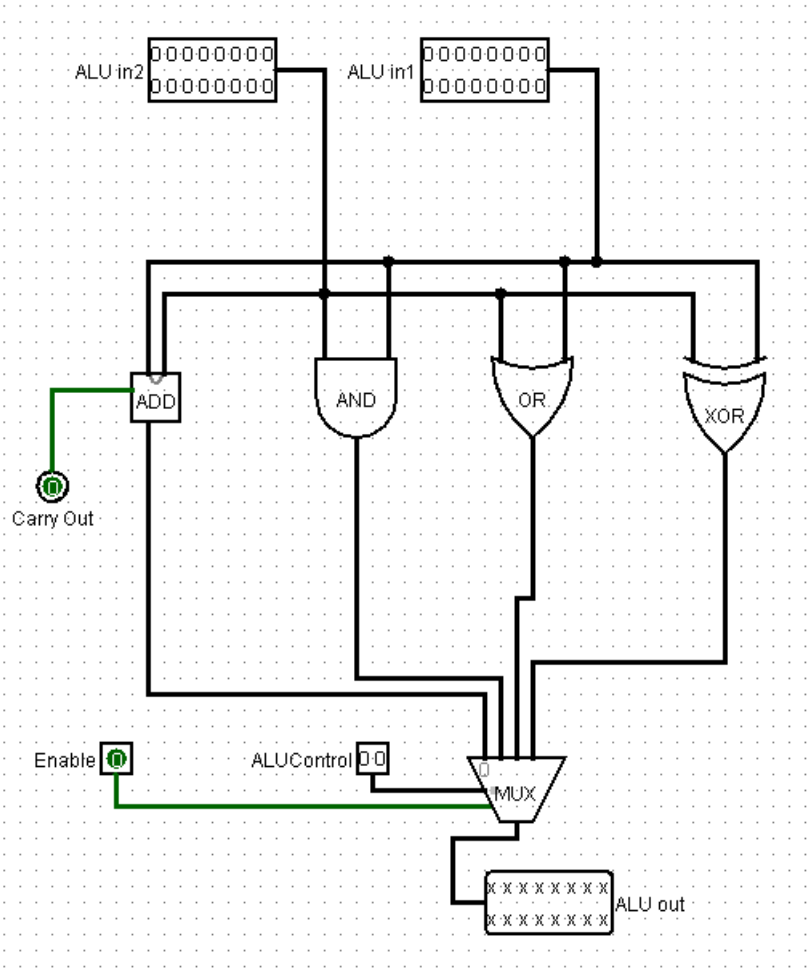
It has 1 input:

InstructionData: 18 bit data according to our ISA.

There are 13 outputs in this component:

1- Jump ADDR (15 bit)
2- Opcode (3 bit)
3- AL/LD DST (4 bit)
4- ST SRC (4 bit)
5- AL SRC1 (4 bit)
6- AL SRC2 (4 bit)
7- Imm Decider Bit (1 bit)
8- ImmData (6 bit)
9- LD/ST ADDR (11 bit)
10- nzp (3 bit)
11- OP1 (4 bit)
12- OP2 (4 bit)
13- BR ADDR (4 bit)

- **Arithmetic Logic Unit:**



Our ALU has 4 operations, ADD , AND, OR and XOR and 4 inputs:
1- ALU in1: 16 bit data

2- ALU in2: 16 bit data

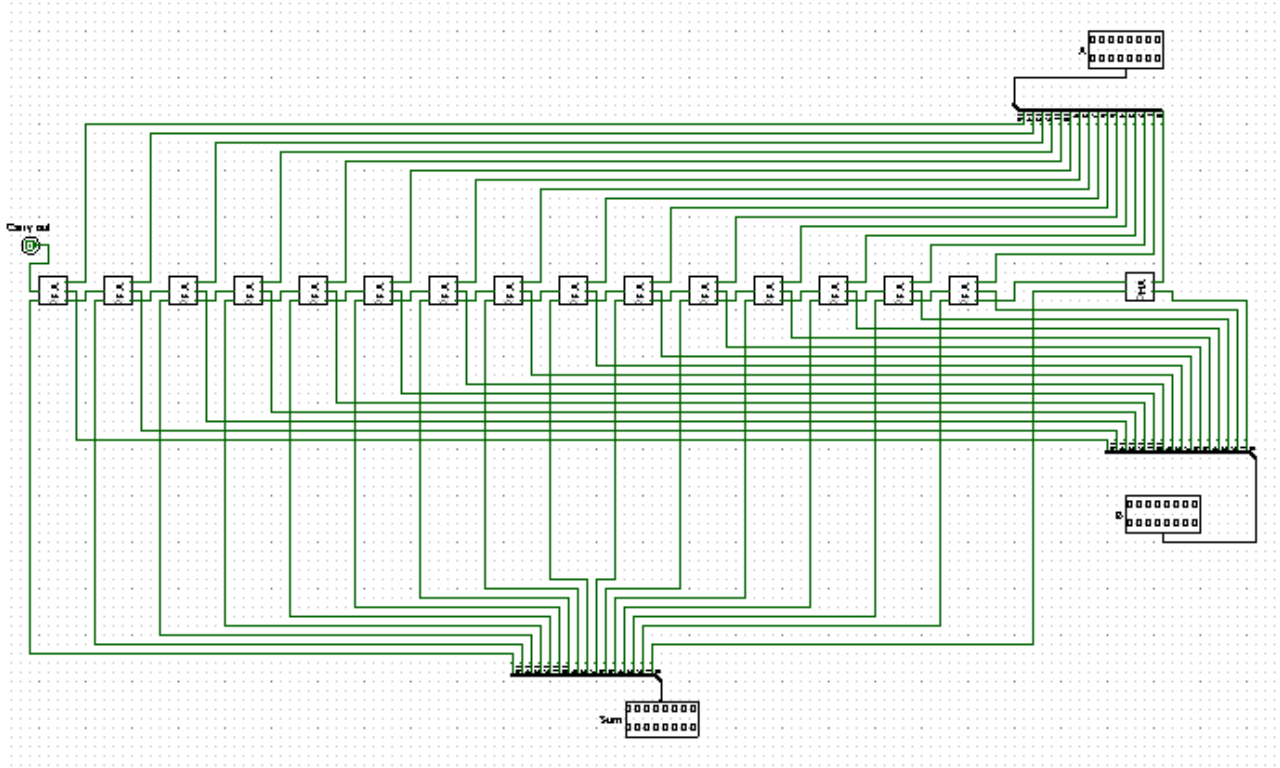
3- ALUControl: Select bit for operation multiplexer. If it's 00, ADD output will be selected. If it's 01, AND output will be selected. If it's 10, OR output will be selected. If it's 11, XOR output will be selected as output.
4-Enable: Enable ALU.

There are 2 outputs in this component:
1- Result: Output of desired operation.

2- Carry out: Carry out bit coming from adder.

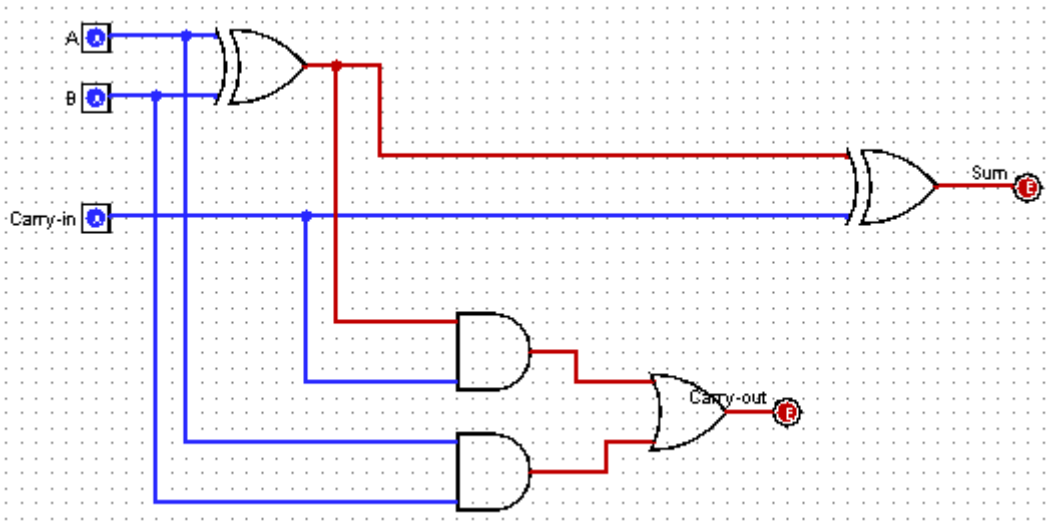
16 bit AND gate, 16 bit OR gate, 16 bit XOR gate and 16 bit adder (which we implemented) were used in this component.

- **16 Bit Adder:**



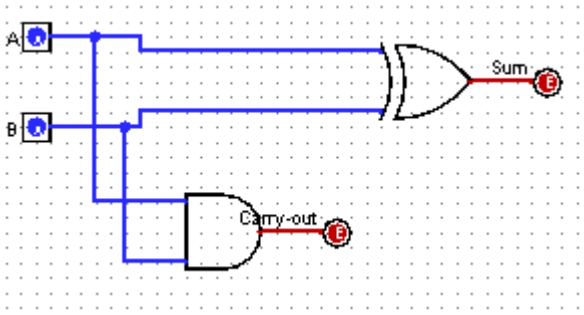
16 bits adder contains 15 Full adder and a half adder. It takes two 16 bit inputs (A and B), sums them then returns a 16 bits data (Sum) as result.

- **Full Adder:**



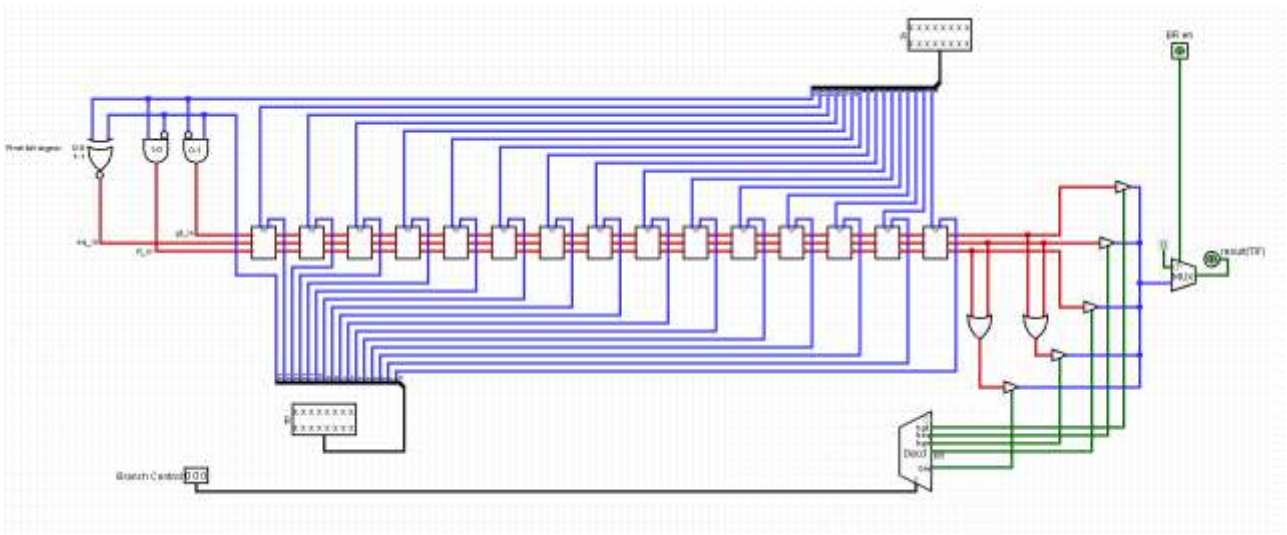
Full adder takes two 1 bit input (A and B) and a carry-in input and sums three of them. Output is 1 bit result and 1 bit carry-out.

- **Half Adder:**



Half adder takes two 1 bit input (A and B) only and sums them. Output is 1 bit result and 1 bit carry-out.

- **Branch:**



Our Branch has 5 operations, BGT , BEQ, BGE, BLT and BLE and 4 inputs:

1- A: 16 bit data

2- B: 16 bit data

3- Branch Control: Select bit for operation multiplexer. If it's 001, BGT output will be selected. If it's 010, BEQ output will be selected. If it's 011, BGE output will be selected. If it's 100, BLT output will be selected. If it's 110, BLE output will be selected as output.

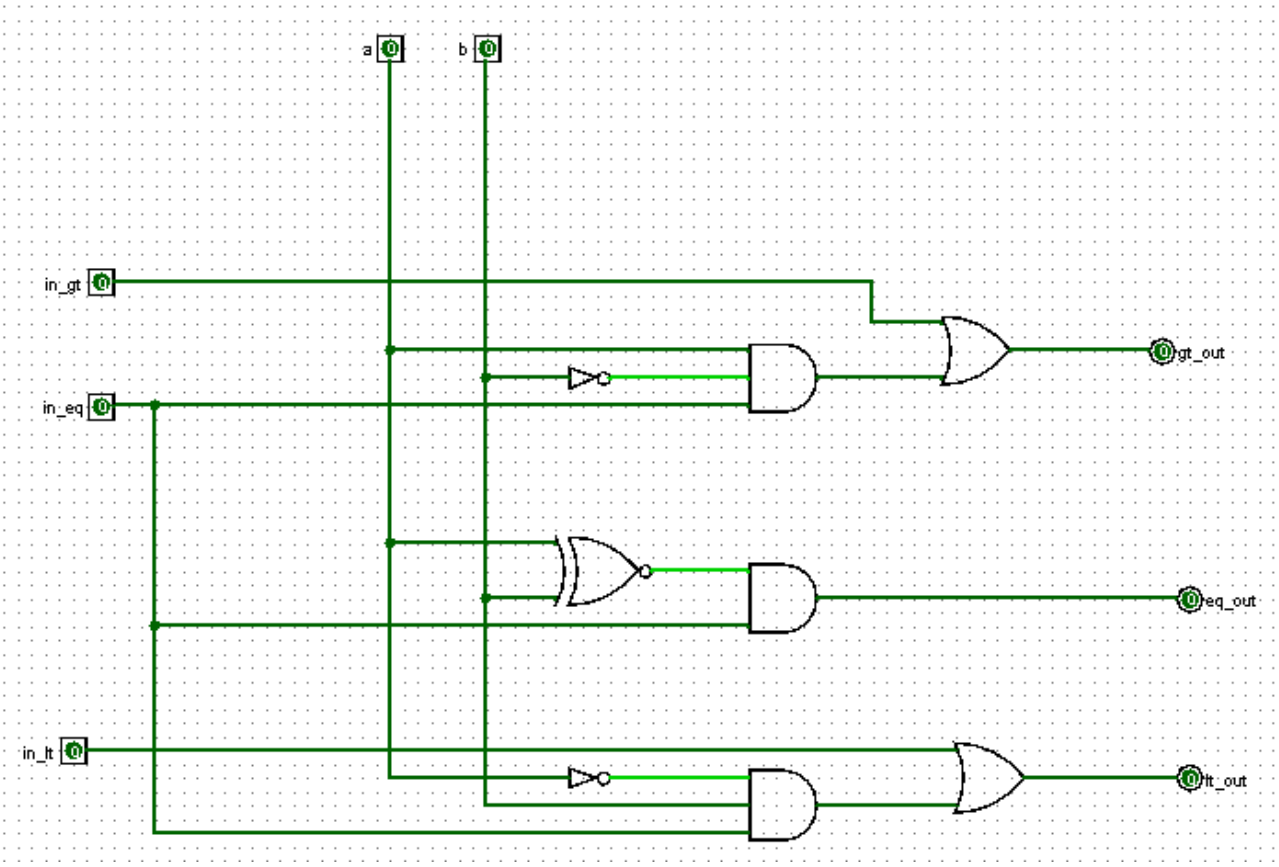
4-Enable: Enable Branch.

There is 1 output in this component:

1- Result: Output of desired operation.

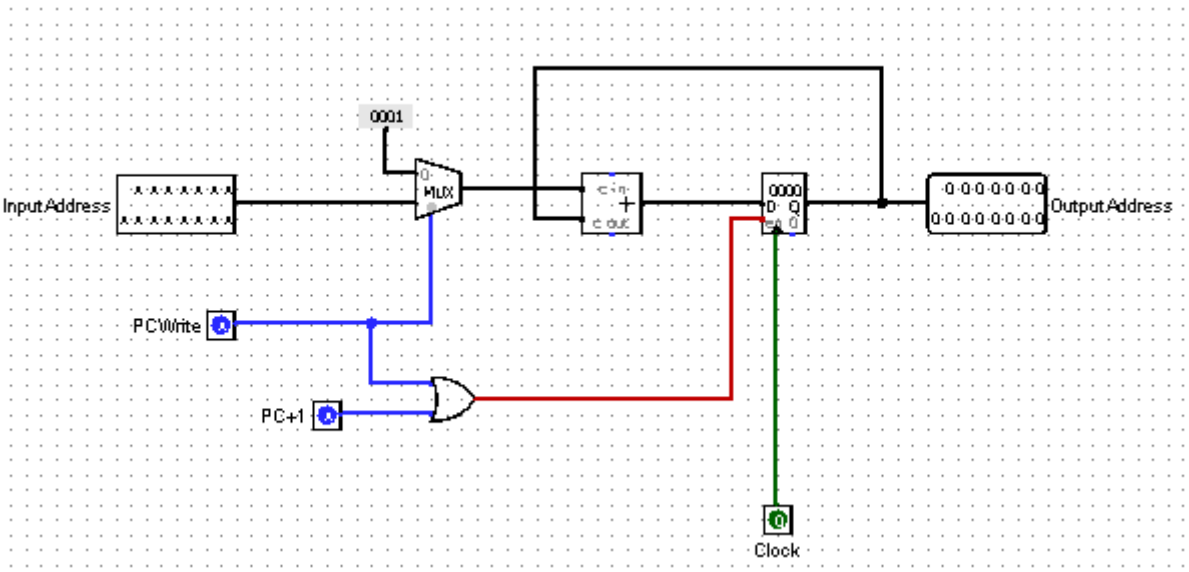
1 bit comparator (which we implemented), 1 bit AND gate, 1 bit OR gate and 1 bit XNOR gate were used in this component.

- **1 Bit Comparator:**



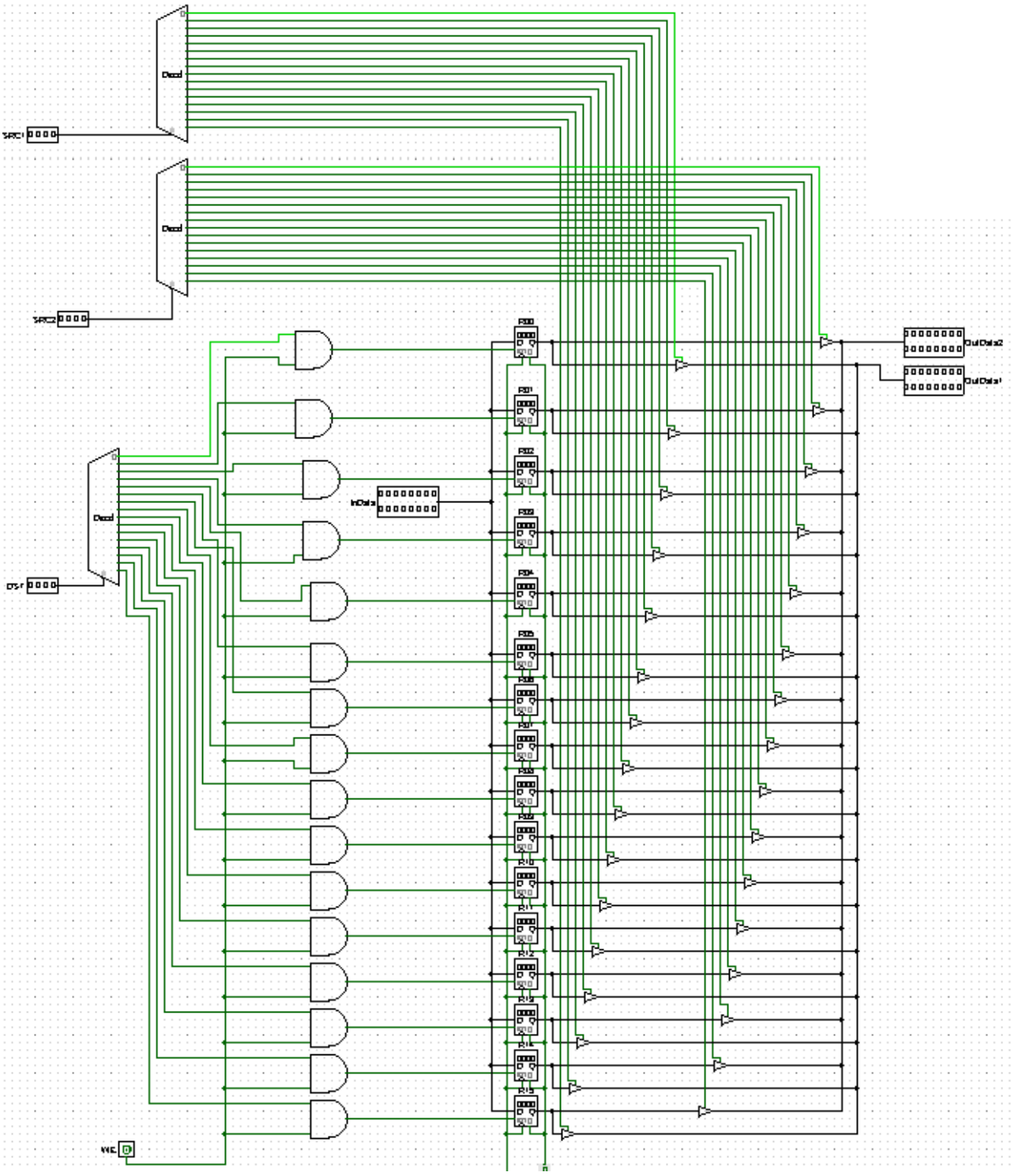
1 Bit Comparator takes two 1 bit input (a and b) and three 1 bit flags which represent comparisons of previous steps so far. Output is again three 1 bit flags to pass results to next step.

- **Program Counter (PC):**



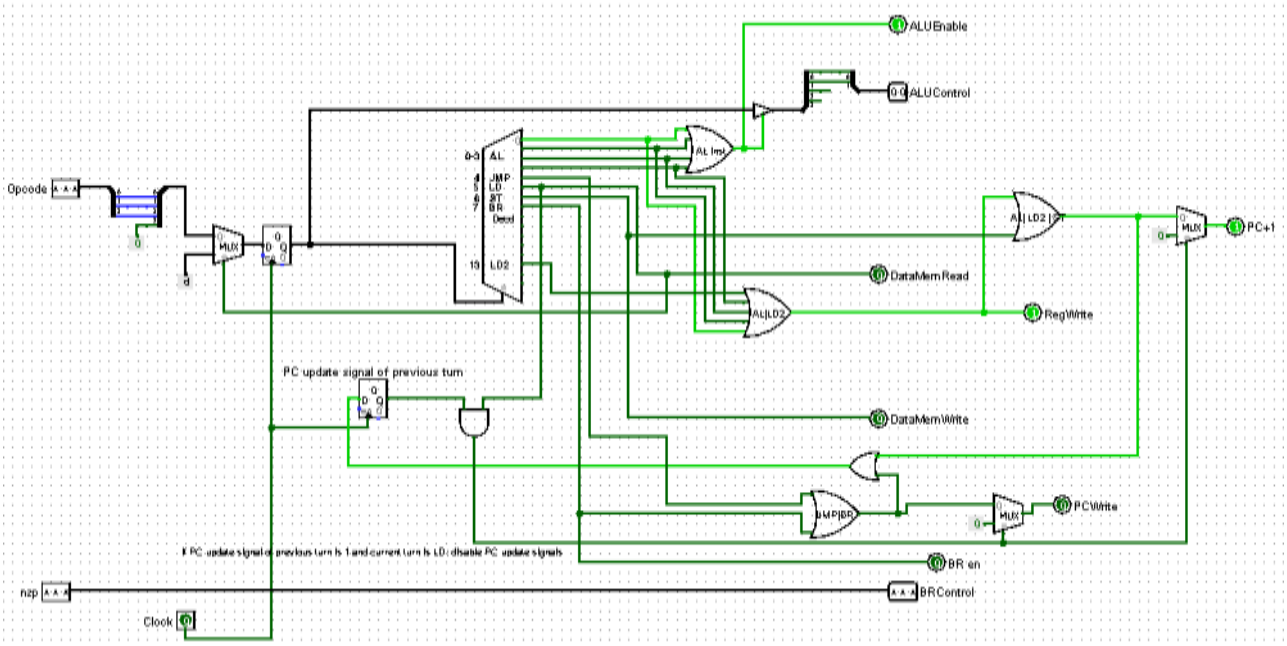
Program Counter takes one input as 15 bits address (InputAddress) and two 1 bit signals(PCWrite and PC+1). When PCWrite signal is true, multiplexer allows InputAddress to go in adder, otherwise PC+1 signal is true, multiplexer allows adder to increase +1. Other input of adder is current address so in general, PC sets the next instruction address. Output is 15 bits OutputAdress which goes to Instruction Memory unit.

- **Register File:**



Register File takes three 4 bits inputs (DST,SRC1,SRC2), one 16 bits data (InData) and one 1 bit signal (WE) to enable writing. Outputs are two 16 bits data (OutData1 and OutData2). When WE signal is true, Register File unit writes InData to given destination register (DST), otherwise reads source register two source registers (SRC1 and SRC2) and returns the values inside them as outputs.

- **Control Unit:**



Control Unit takes 4 bits Opcode as input and produces corresponding signals for each of operations. These signals are:

-AL operations: ALUEnable (1bit), ALUControl (2bits), RegWrite (1bit), PC+1 (1bit)

-JMP: PCWrite (1bit)

-LD: DataMemRead (1bit)

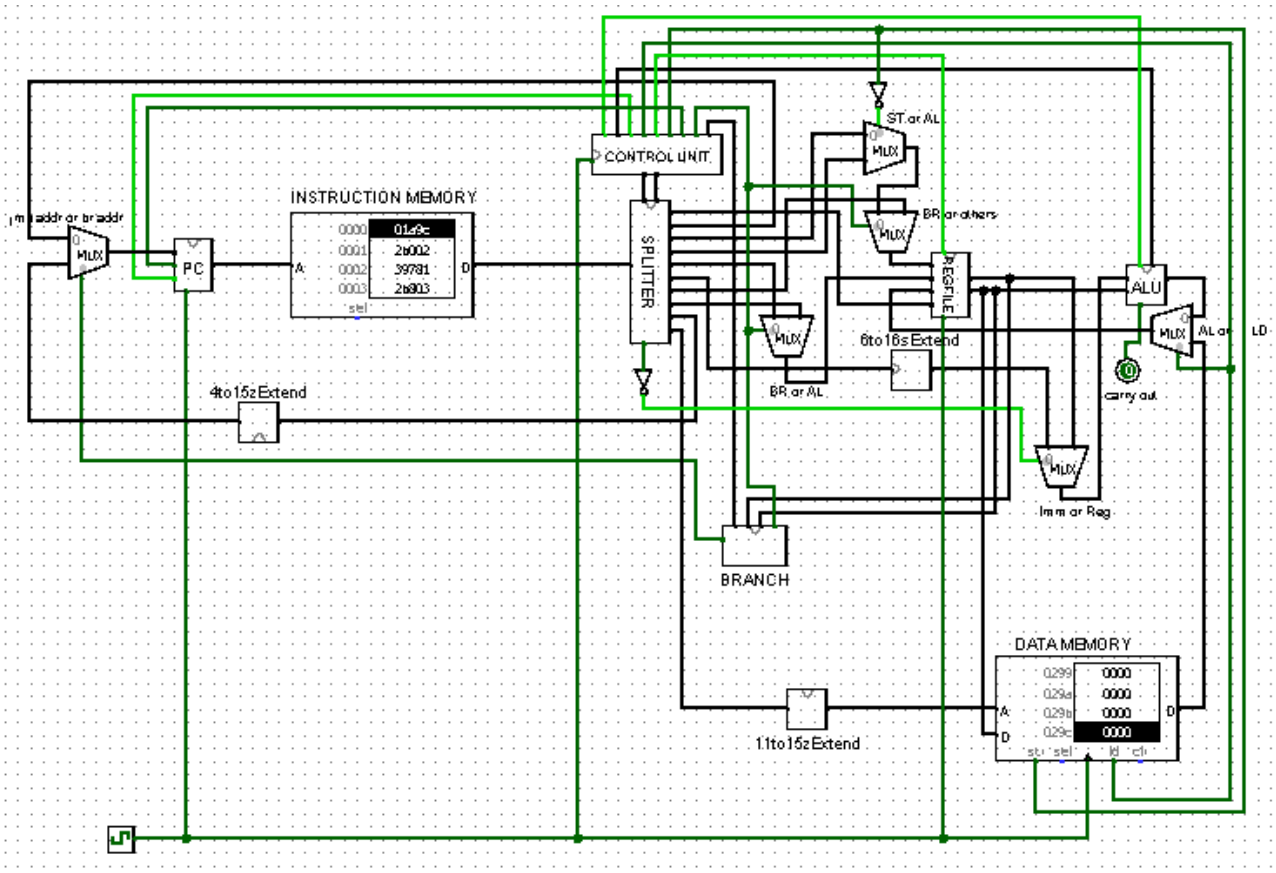
-LD2: RegWrite (1bit), PC+1 (1bit)

-ST: DataMemWrite (1bit), PC+1 (1bit)

-BR: Bren (1bit), PCWrite (1bit), BRControl (3bits)

Note: In our implementation there's a register to hold previous PC update signal (PCWrite or PC+1). If this register is 1 and current instruction is LD, we disable PC update signals because there's a LD2 instruction for next step.

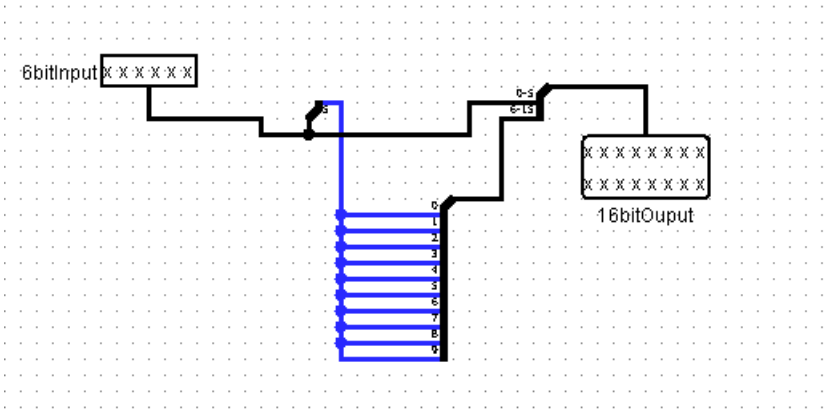
- **Main:**



In main, we did all proper connections and added 6 multiplexers for overlapped wires. For example:
-We have to choose AL or LD as RegFile input since they both have to write results in to DST register.
-We have to choose Imm or Reg value for input of ALU since second arithmetic logic input can be a immediate value or a register value.
- We have to choose BR or AL for the input of RegFile since they both uses SRC2.
- We have to choose BR, ST or AL for the input of RegFile since they both uses SRC1. There are 3 choices here, so we had to use 2 sequential multiplexers.
- We have to choose JMP address or BR address for the input of PC since they both uses pc relative addresses.

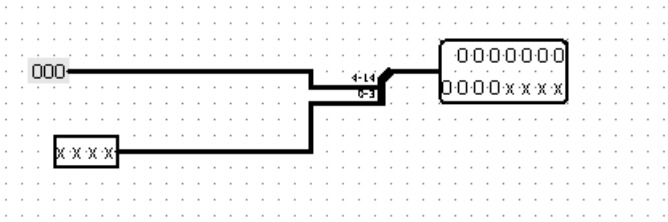
We also used 3 extenders:
- 11to15zExtend for LD/ST address which goes to DataMemory unit
- 4to15zExtend for BR address which goes to PC
- 6to16sExtend for ImmData which goes to ALU (in2).

- **6 To 16 Sign Extend:**



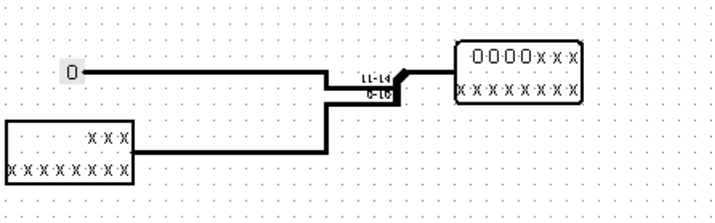
It has 6 bit input data and 16 bit output data.Used for extension of IMM values of ADDI, ANDI, ORI and XORI operations according to our ISA.

- **4 To 15 Zero Extend:**



It has 4 bit input data, 11 bit zero value as input and 15 bit output data. Used for extension of ADDR values of Branch operations (BEQ, BLT, BGT, BLE, BGE) according to our ISA.

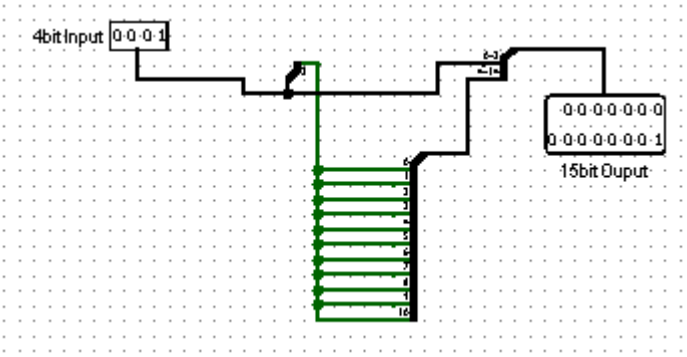
- **11 To 15 Zero Extend:**



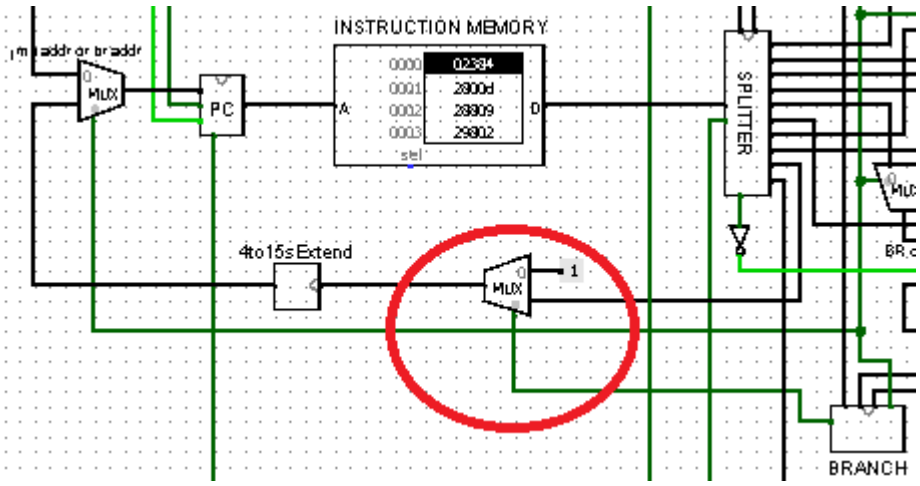
It has 11 bit input data, 4 bit zero value as input and 15 bit output data.Used for extension of ADDR values of LD and ST operations according to our ISA.

- **Problems and Improvements**
  - Non-negative PC relative address: We had zero extend for branch unit at first by mistake so to handle this problem we changed it to a sign extender.

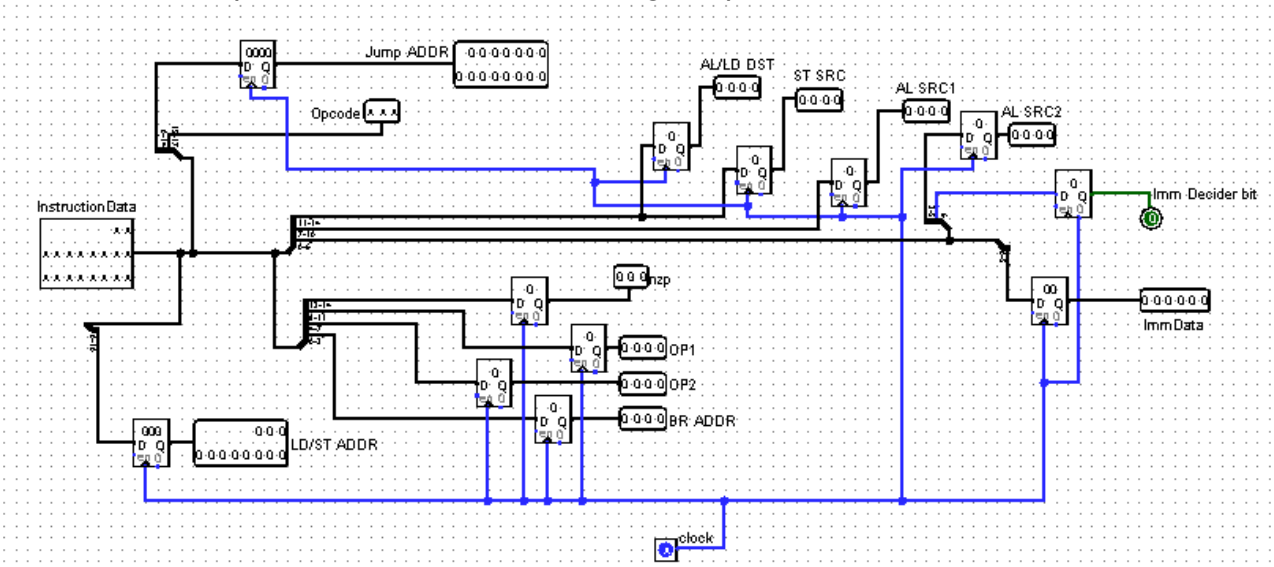
(4to15zExtend -> 4to15sExtend)



- No PC update on false comparisons of BR unit: To handle this problem we added an extra multiplexer to main.

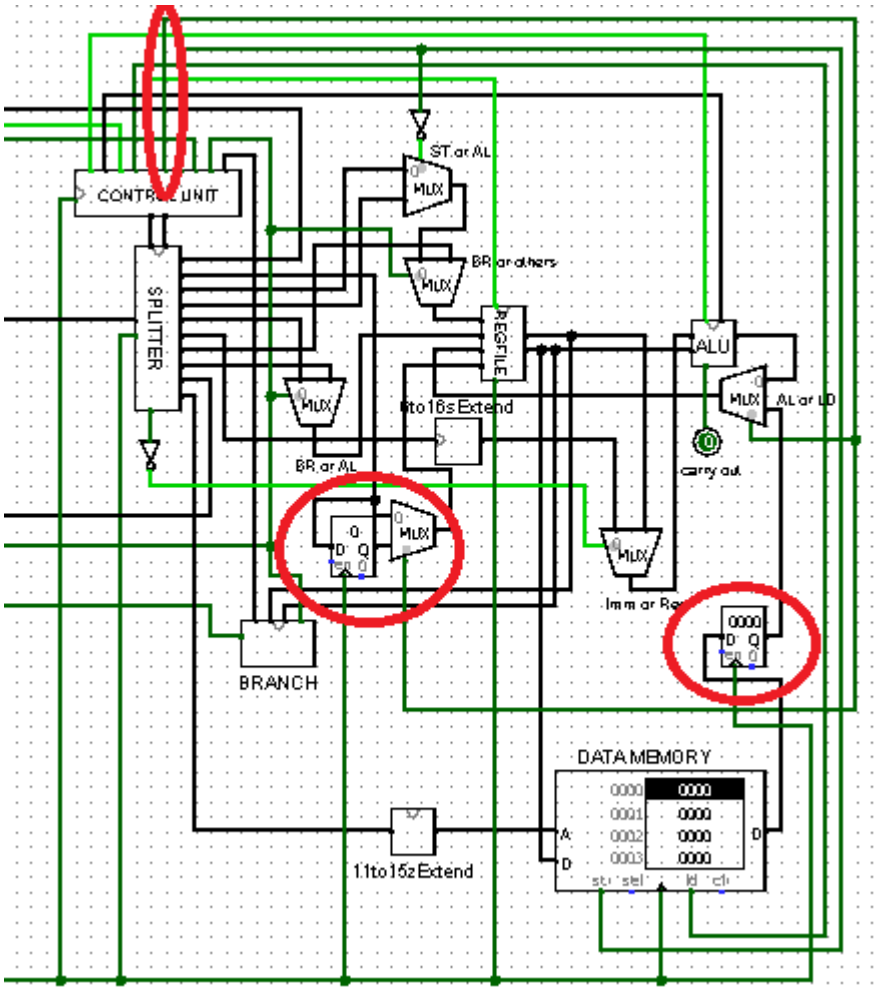


- Synchronization problem in Splitter: Our splitter was using the values of next step while Control Unit was using the previous values since we had an extra



register in Control Unit. So we added registers to all of outputs of Splitter like below:

- LD2 synchronization problem in main: We added 2 extra registers to output of DataMemory unit and AL/LD DST output of splitter to use values of previous step since LD2 and its previous step LD uses same values. We also added ChooseLD2 signal to handle multiplexer that next to the register newly added.



And in control unit,