Erman HAVUÇ
150114023

İbrahim AYCAN
150114075

MAIN GOAL OF THIS PROJECT:

Our main goal in this Project is design a processor for (ADD, AND, LD, ST, ADDI, ANDI, CMP, JUMP, JE, JA, JBE, JAE) instruction sets. Processor will have 12-bit address and 16 bits data width. Processor will have 5 parts. Register file holds register values and signal to write into any register. There will be 16 registers in processor. Instruction Memory will be a read-only memory and instructions will be stored in this component. If the current instruction is not one of the JUMP instructions (JUMP, JE, JA etc.); the next instruction will be fetched and executed consecutively from this memory. Data Memory will be read-write memory which will store data. Program will be able to read data from data memory, and store data to this memory. Data Memory will have 12 bits address width, and 16 bits data width. Control Unit will produce proper signals to all data path components. For example, if the instruction is ST, control unit should produce memWrite signal which will allow Data Memory component to write data value on its data input to the address on its address input. Arithmetic Logic Unit (ALU) will compute arithmetic operations ADD, AND, ADDI, ANDI. Operands will be fetched from register + register or register + immediate value. Result will be stored to the Register File. Control unit should produce proper signals to ALU according to instruction opcode.

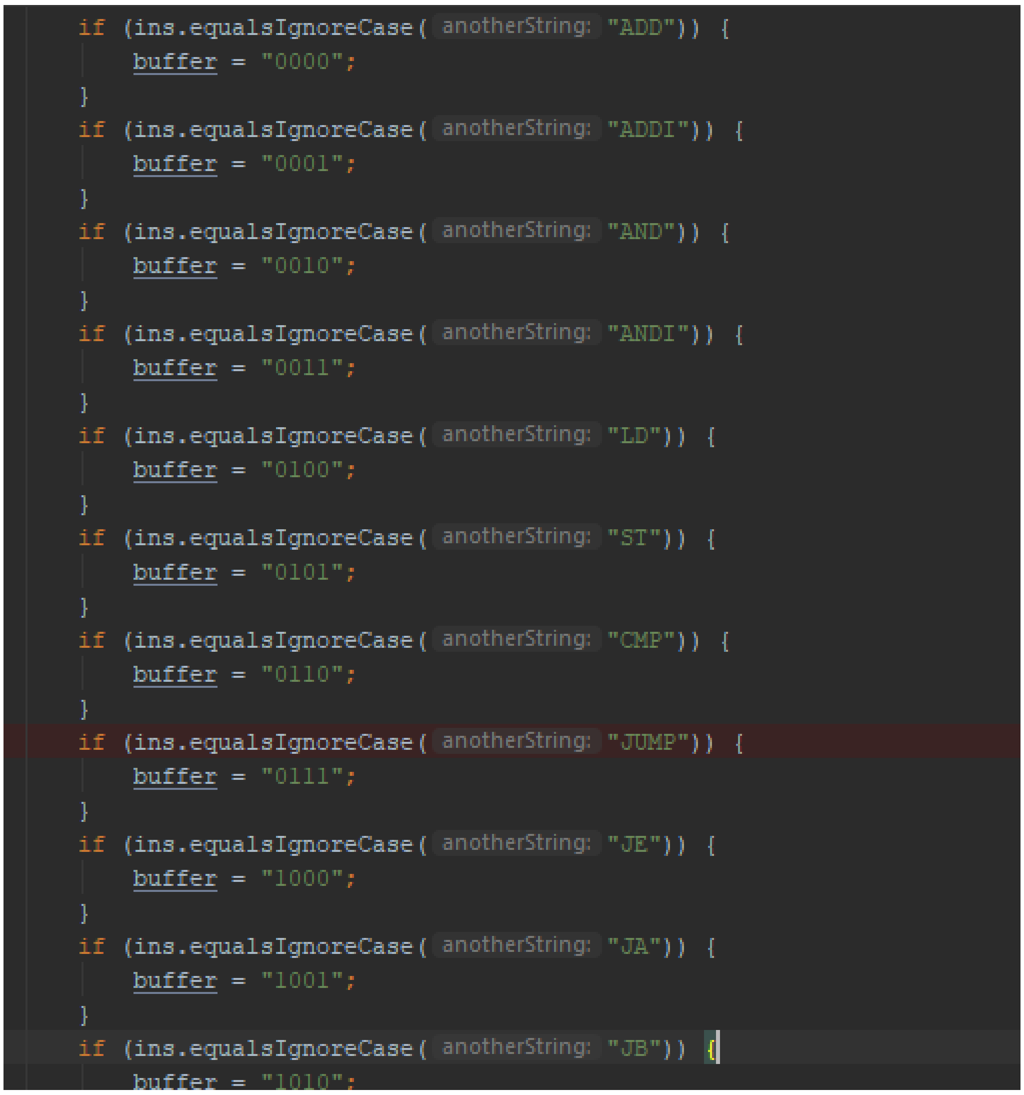
ASSEMBLER PART:

Here is our instruction sets:

	OPCODE															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD	*0*	*0*	*0*	*0*	*DEST*				*SRC1*				*SRC2*			
ADDI	*0*	*0*	*0*	*1*	*DEST*				*SRC1*				*IMM*			
AND	*0*	*0*	*1*	*0*	*DEST*				*SRC1*				*SRC2*			
ANDI	*0*	*0*	*1*	*1*	*DEST*				*SRC1*				*IMM*			
LD	*0*	*1*	*0*	*0*	*DEST*				*ADDR*							
ST	*0*	*1*	*0*	*1*	*SRC*				*ADDR*							
CMP	*0*	*1*	*1*	*0*	0	0	0	0	*OP1*				*OP2*			
JUMP	*0*	*1*	*1*	*1*	*OFFSET*											
JE	*1*	*0*	*0*	*0*	*ADDR*											
JA	*1*	*0*	*0*	*1*	*ADDR*											
JB	*1*	*0*	*1*	*0*	*ADDR*											
JBE	*1*	*0*	*1*	*1*	*ADDR*											
JAE	*1*	*1*	*0*	*0*	*ADDR*											

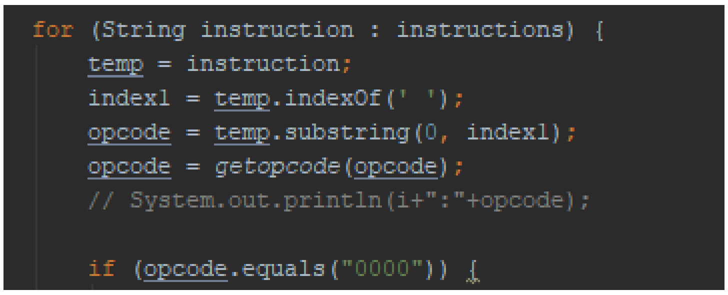
In Java Code:

In our code first we defined the opcodes in getopcode() method.



```
    if (ins.equalsIgnoreCase( anotherString: "ADD")) {
        buffer = "0000";
    }
    if (ins.equalsIgnoreCase( anotherString: "ADDI")) {
        buffer = "0001";
    }
    if (ins.equalsIgnoreCase( anotherString: "AND")) {
        buffer = "0010";
    }
    if (ins.equalsIgnoreCase( anotherString: "ANDI")) {
        buffer = "0011";
    }
    if (ins.equalsIgnoreCase( anotherString: "LD")) {
        buffer = "0100";
    }
    if (ins.equalsIgnoreCase( anotherString: "ST")) {
        buffer = "0101";
    }
    if (ins.equalsIgnoreCase( anotherString: "CMP")) {
        buffer = "0110";
    }
    if (ins.equalsIgnoreCase( anotherString: "JUMP")) {
        buffer = "0111";
    }
    if (ins.equalsIgnoreCase( anotherString: "JE")) {
        buffer = "1000";
    }
    if (ins.equalsIgnoreCase( anotherString: "JA")) {
        buffer = "1001";
    }
    if (ins.equalsIgnoreCase( anotherString: "JB")) {
        buffer = "1010";
```

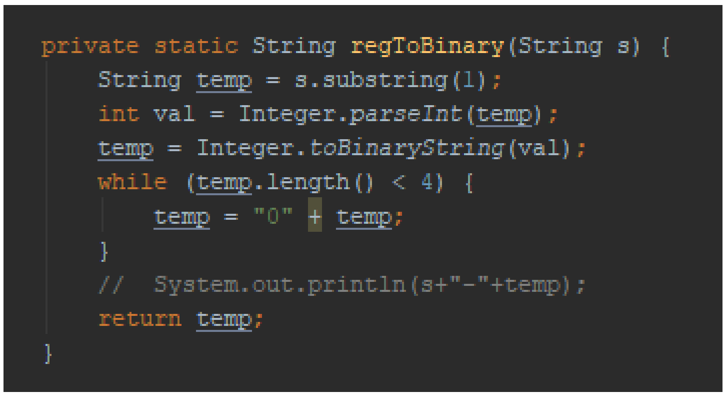
Then we read the input file and get the opcodes of the instructions.



```
for (String instruction : instructions) {
    temp = instruction;
    index1 = temp.indexOf(' ');
    opcode = temp.substring(0, index1);
    opcode = getopcode(opcode);
    // System.out.println(i+":"+opcode);

    if (opcode.equals("0000")) {
```

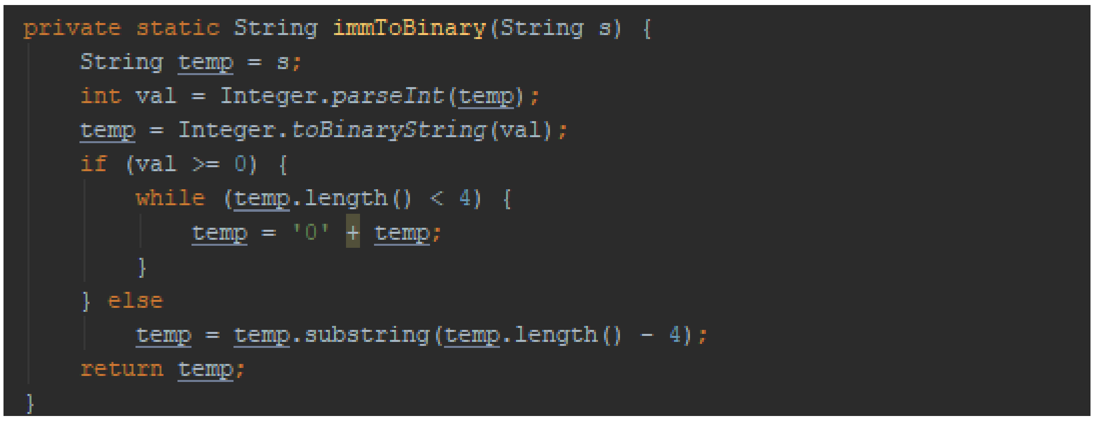
Depend on our input file:

-If there is a register in the inputs then we used registerToBinary() method and turned these register into binary values.



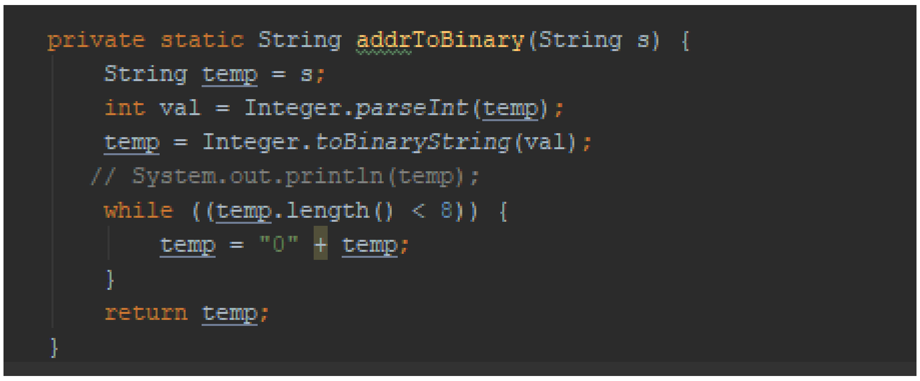
```
private static String regToBinary(String s) {
    String temp = s.substring(1);
    int val = Integer.parseInt(temp);
    temp = Integer.toBinaryString(val);
    while (temp.length() < 4) {
        temp = "0" + temp;
    }
    //  System.out.println(s+"-"+temp);
    return temp;
}
```

If there is an immediate value in our inputs, then we used immtoBinary() method to turn immediate value into binary form.



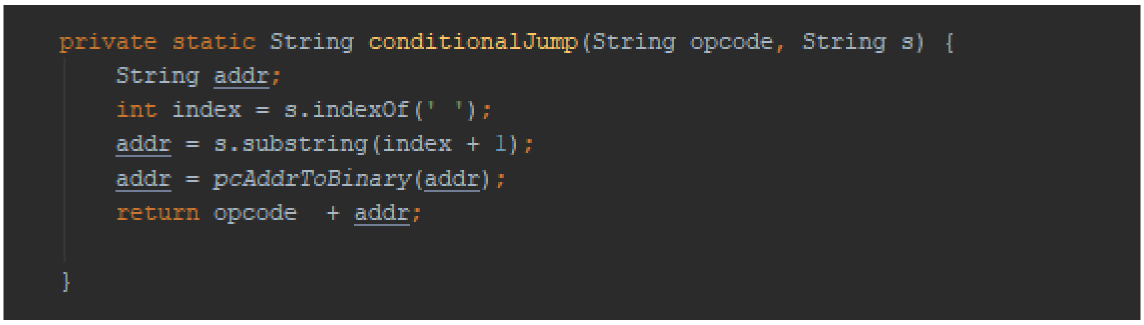
```
private static String immToBinary(String s) {
    String temp = s;
    int val = Integer.parseInt(temp);
    temp = Integer.toBinaryString(val);
    if (val >= 0) {
        while (temp.length() < 4) {
            temp = '0' + temp;
        }
    } else
        temp = temp.substring(temp.length() - 4);
    return temp;
}
```

If there is an address value then we turned this into binary form by using addrtobinary() function.



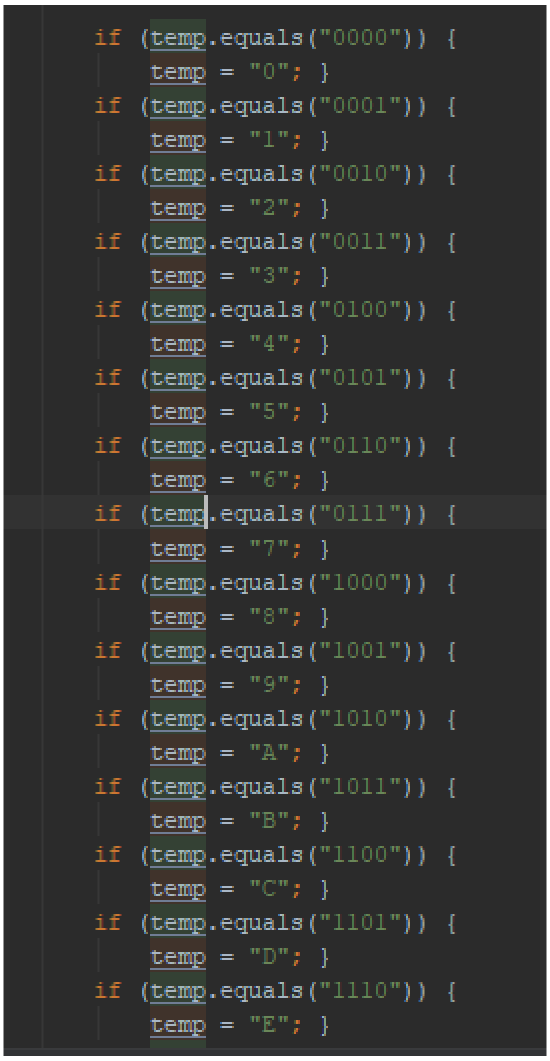
```
private static String addrToBinary(String s) {
    String temp = s;
    int val = Integer.parseInt(temp);
    temp = Integer.toBinaryString(val);
   // System.out.println(temp);
    while ((temp.length() < 8)) {
        temp = "0" + temp;
    }
    return temp;
}
```

If there is a jump operation which has PC offset value then we used conditionaljump() method to turn offset value into binary form.



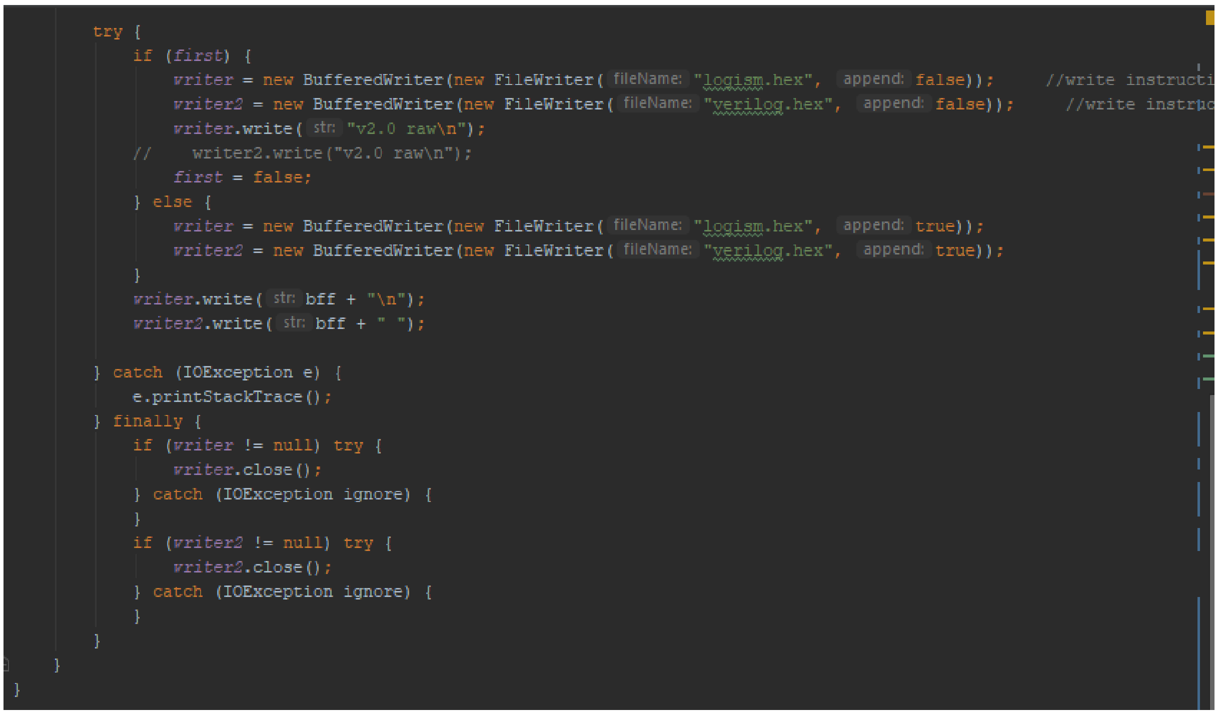
```java
private static String conditionalJump(String opcode, String s) {
    String addr;
    int index = s.indexOf(' ');
    addr = s.substring(index + 1);
    addr = pcAddrToBinary(addr);
    return opcode  + addr;

}
```

After all these operations our input file becomes in binary form. So we turned them into hexadecimal form by using binaryToHex() function.



```java
    if (temp.equals("0000")) {
        temp = "0"; }
    if (temp.equals("0001")) {
        temp = "1"; }
    if (temp.equals("0010")) {
        temp = "2"; }
    if (temp.equals("0011")) {
        temp = "3"; }
    if (temp.equals("0100")) {
        temp = "4"; }
    if (temp.equals("0101")) {
        temp = "5"; }
    if (temp.equals("0110")) {
        temp = "6"; }
    if (temp.equals("0111")) {
        temp = "7"; }
    if (temp.equals("1000")) {
        temp = "8"; }
    if (temp.equals("1001")) {
        temp = "9"; }
    if (temp.equals("1010")) {
        temp = "A"; }
    if (temp.equals("1011")) {
        temp = "B"; }
    if (temp.equals("1100")) {
        temp = "C"; }
    if (temp.equals("1101")) {
        temp = "D"; }
    if (temp.equals("1110")) {
        temp = "E"; }
```

After all these operations we write the output to the file. To use in Logisim we add ‘v2.0 raw’ to the top of the file.



```
        try {
            if (first) {
                writer = new BufferedWriter(new FileWriter( fileName: "logism.hex",  append: false));     //write instructi
                writer2 = new BufferedWriter(new FileWriter( fileName: "verilog.hex",  append: false));     //write instruc
                writer.write( str: "v2.0 raw\n");
            //    writer2.write("v2.0 raw\n");
                first = false;
            } else {
                writer = new BufferedWriter(new FileWriter( fileName: "logism.hex",  append: true));
                writer2 = new BufferedWriter(new FileWriter( fileName: "verilog.hex",  append: true));
            }
            writer.write( str: bff + "\n");
            writer2.write( str: bff + " ");

        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (writer != null) try {
                writer.close();
            } catch (IOException ignore) {
            }
            if (writer2 != null) try {
                writer2.close();
            } catch (IOException ignore) {
            }
        }
    }
}
```

ADD: ADD instruction adds the value of two register which are SRC1 and SRC2, and store result into another register DST.

AND: AND instruction reads two registers and makes bitwise operation ‘AND’ and writes the output to another register.

ADDI: ADDI instruction reads SRC1 register and adds with immediate value and writes into DST register.

ANDI: ANDI instruction reads SRC1 register and makes bitwise operation ‘AND’ with an immediate value and writes output into DST register.

LD: LD instruction load a value from Memory and write into DST register.

ST:ST instruction reads value from register and writes stores in memory

CMP:CMP instruction compares two operands

JUMP: JUMP instruction set Program Counter to given value.

JE:JE instruction compares two operand and jumps if they are equal

JA: JA instruction compares two operand and jumps if first operand is greater than second.

JB: JB instruction compares two operand and jumps if first operand is less than second.

JBE: JBE instruction compares two operand and jumps if first operand is less than or equal to second.

JAE: JAE instruction compares two operand and jumps if first operand is greater than or equal to second.

Here is our sample inputs and outputs:

ADD R3,R7,R4	0000001101110100	0374
ADDI R1,R0,7	0001000100000111	1107
And R2,R0,-3	0010001000000011	2203
ANDI R5,R0,-4	0011010100001100	350C
ST R2,3	0101001000000011	5203
LD R4,4	0100010000000100	4404
ST R3,11	0101001100001011	530B
LD R7,3	0100011100000011	4703
JUMP 5	0111000000000101	7005
JUMP -6	0111000011111010	70FA
ADD R0,R0,R0	0000000000000000	0000
CMP R5,R7	0110000001010111	6057
JA -41	1001111111010111	9FD7
JE 65	1000000001000001	8041
JB -3	1010111111111101	AFFD
JBE 7	1011000000000111	B007
JAE -7	1100111111111001	CFF9

Erman HAVUÇ
150114023

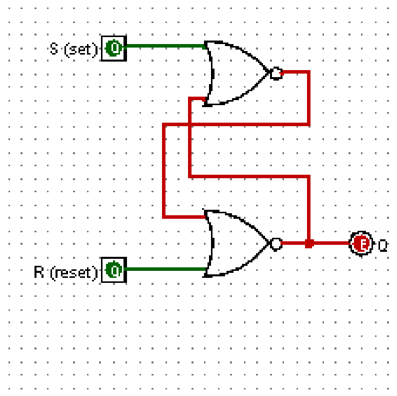
İbrahim AYCAN
150114075

LOGISM PART:

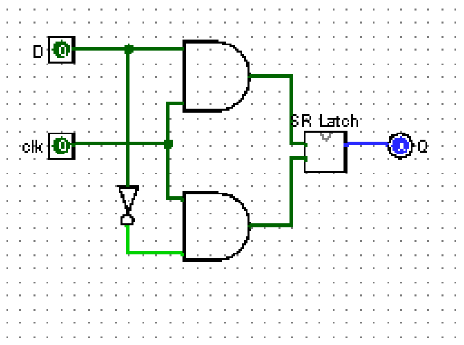
S-R LATCH:

The simplest bistable device is SR to create an S-R latch, we wired two NOR Gates in such a way that the output of one feed back to the input of another and vice versa.

To create an S-R latch, we wired two NOR gates in such a way that the output of one feed back to the input of another, and vice versa, The Q and not-Q outputs are supposed to be in opposite states.



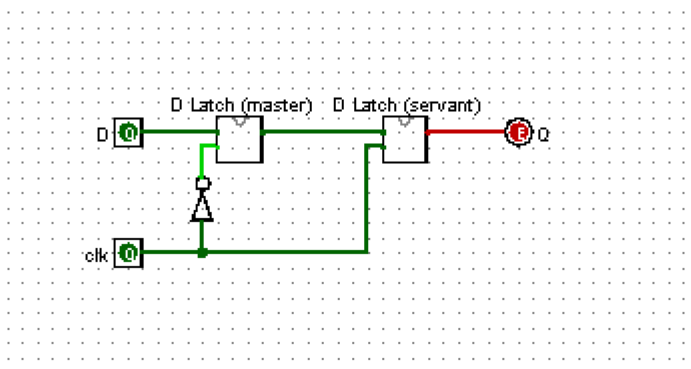
D-LATCH: D-latch is a bitsable device that can be used to store one bit of information. It is an S-R latch with 1 input. There is an inverter added to make R the complement (inverse) of S.



D value is an input that will be stored in D-Latch.

Clk signal will fit the output with D value.

D-FLIP-FLOP:



D-flip flop stores the value on the data. It is a basic memory cell. When clock input goes one, it stores the value on D. When clock goes 0, it will not change the state and store the data in the input and output will not change.

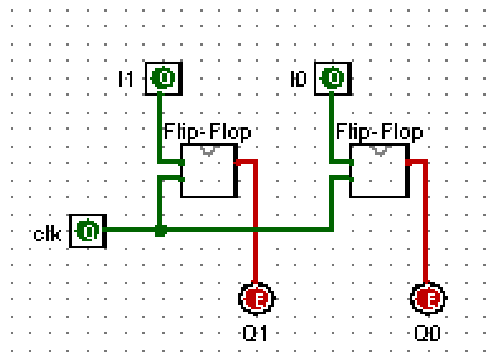
Erman HAVUÇ
150114023

İbrahim AYCAN
150114075

REGISTER:

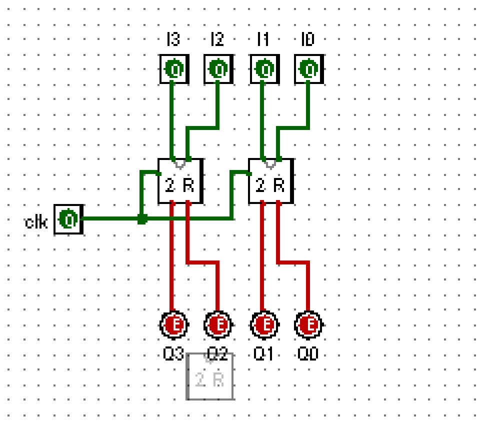
Registers are small set of data holding place. It is a part of our processor. It may holds the address or data in it.

2-bit Register:



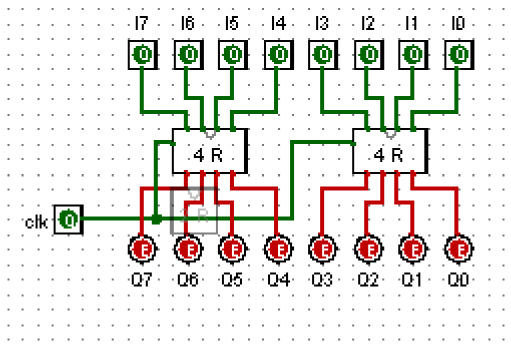
2-bit register stores 2 bits data. We used this register to make 4-bit register. We made this by using two D-flipflop. When clock set from zero to 1 , Register stores the input values.

4-bit Register:



4-bit register stores 4 bits data. We used this register to make 8-bit register and 12-bit register. We made this by using two 4-bit register. When clock set from zero to 1, Register stores the input values.

8-bit Register:

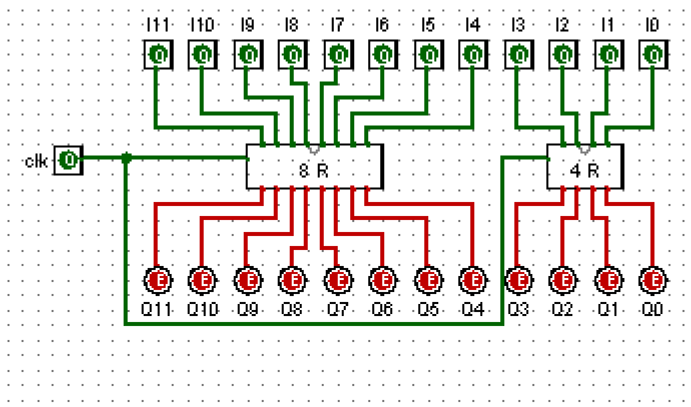


8-bit register stores 8 bits data. We used this register to make 16-bit register. We made this by using two 4-bit register. When clock set from zero to 1 , Register stores the input values.

Erman HAVUÇ
150114023

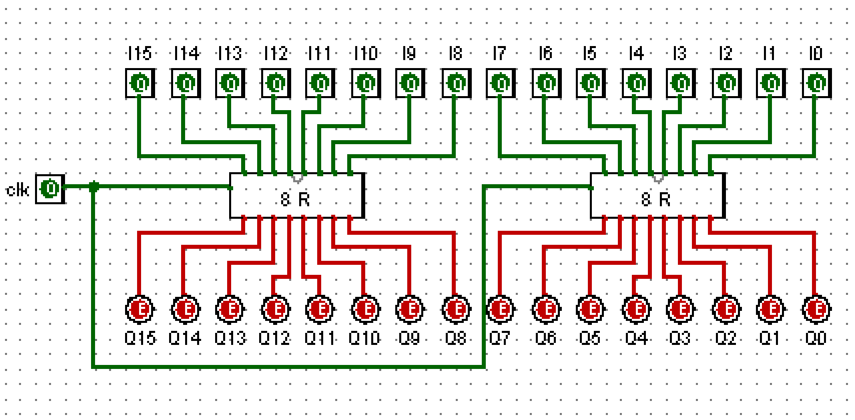
İbrahim AYCAN
150114075

12-bit Register:



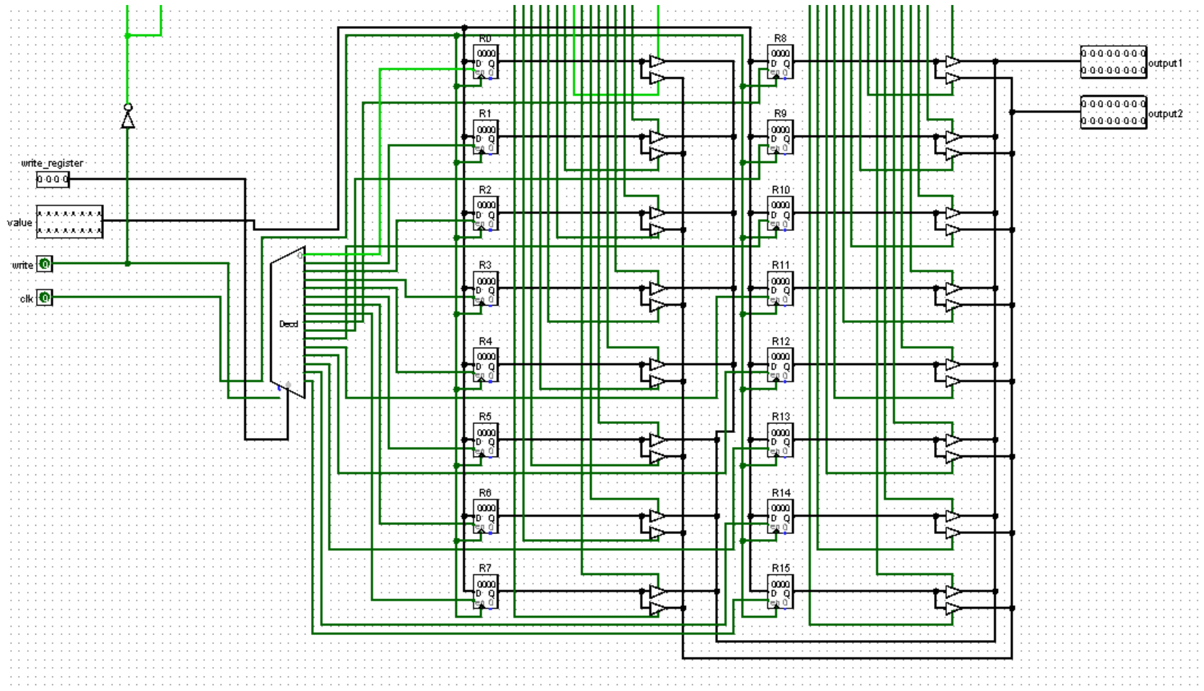
12-bit register stores 12 bits data. We made this by using one 4-bit register and one 8-bit register. When clock set from zero to 1, Register stores the input values.

16-bit Register:



16-bit register stores 16 bits data. We made this by using two 8-bit register. When clock set from zero to 1, Register stores the input values.

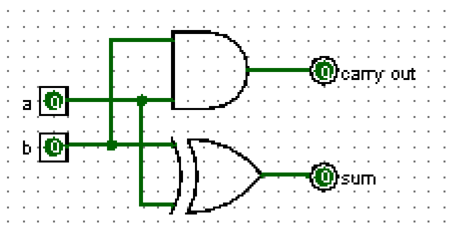
Register File: Register file is a combination of registers. It is an array of processor registers in CPU. Register file contains bits of data and mapping these data into locations.



Erman HAVUÇ
150114023

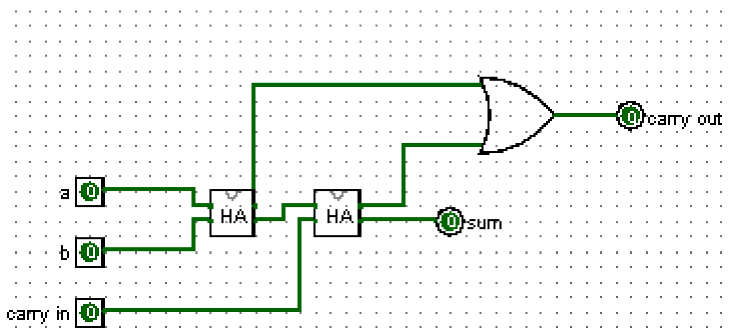
İbrahim AYCAN
150114075

Half Adder:

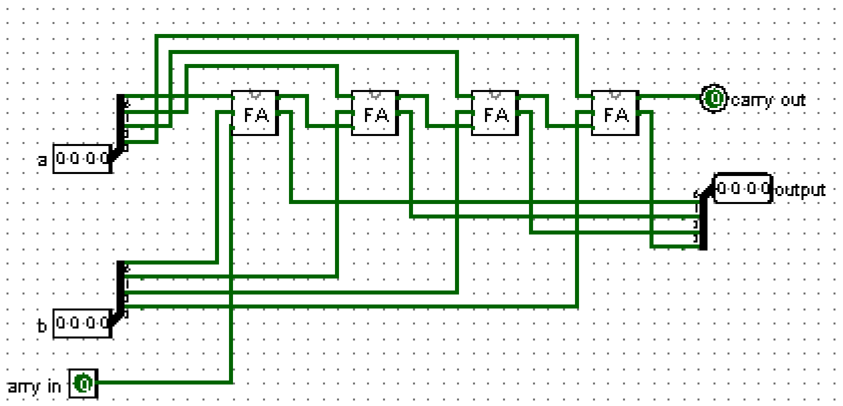


Half adder adds two 1-bit values.

Full Adder:

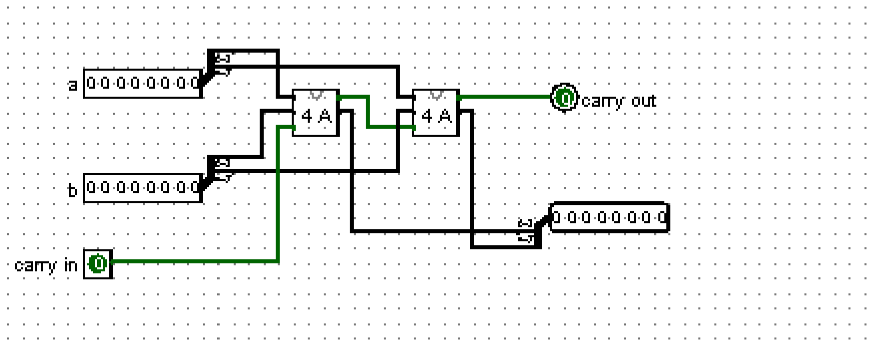


4-bit Adder:



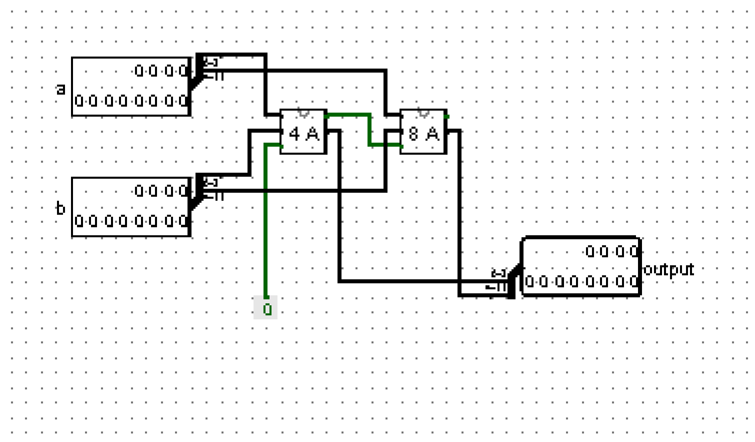
4-bit adder adds 4 bit two numbers. If there is a carry out bit then carry out goes to 1.

8-bit Adder:



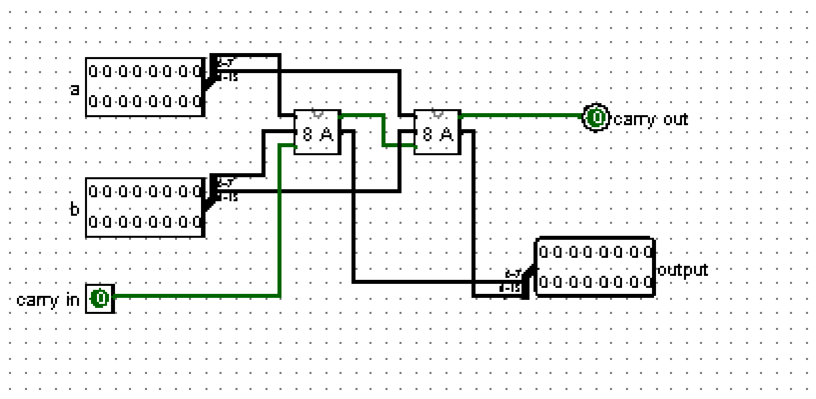
8-bit adder adds 8 bit two numbers. If there is a carry out bit, then carry out goes to 1.

12-bit Adder:



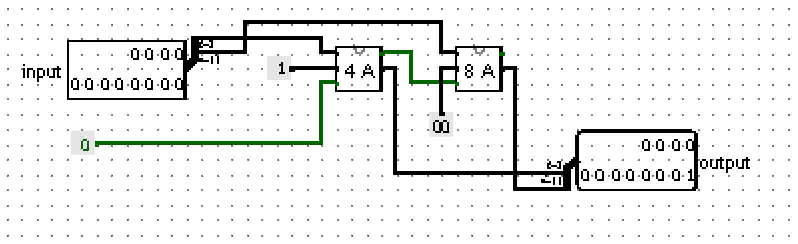
12-bit adder adds 12 bit two numbers. If there is a carry out bit then carry out goes to 1.We used 12-bit adder to increment program counter.

16-bit Adder



16-bit adder adds 16 bit two numbers. If there is a carry out bit, then carry out goes to 1. We used this in register 'ADD' and 'ADDI' instructions.

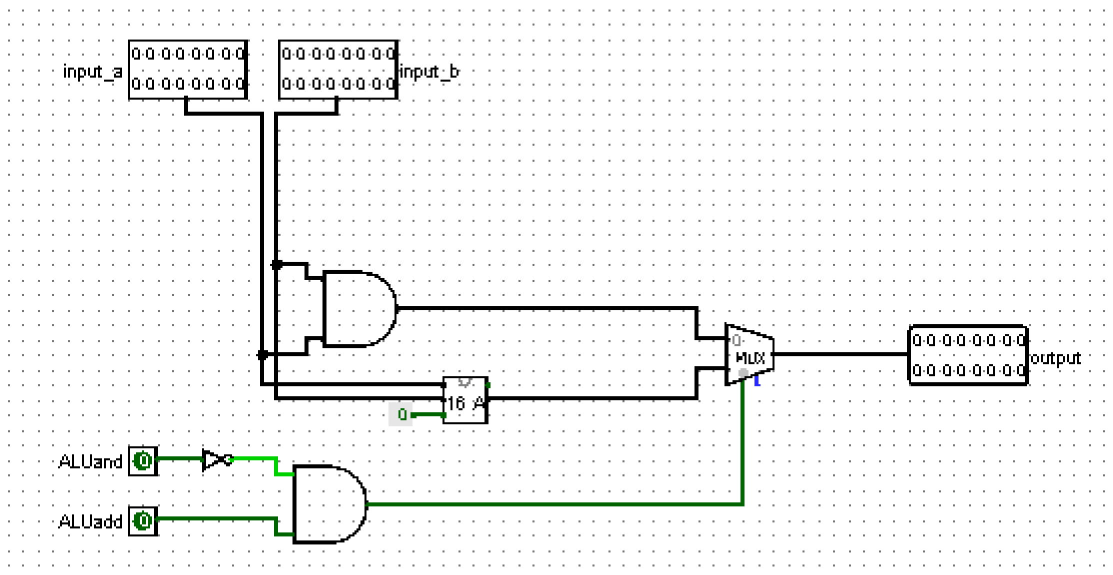
Incrementer:



Incrementer increase the value 1. We used this to increase Program counter each clock cycle.

ALU:

In ALU, we make the arithmetic and logic operations. Which are 'ADD', 'AND', 'ADDI' and 'ANDI'. ALU takes two 16-bit input. If it is an AND operation, it makes bitwise 'AND' operation. If it is an ADD operation, it adds 2 input and goes output value to the output.



Control Unit:

In control unit, Control unit separates input as an opcode.

- If it is an 'ADD' operation, regRead, Regwrite and ALUadd output becomes 1.

- If it is 'ADDI' operation, regRead, Regwrite, immediate and ALUadd output becomes 1.

- If it is 'AND' operation, regRead, Regwrite and ALUand output becomes 1.

- If it is 'ANDI' operation, regRead, Regwrite, immediate and ALUand output becomes 1.

- If it is 'LD' operation, dataMemRead and regWrite output becomes 1.

- If it is 'ST' operation, dataMemWrite and regRead output becomes 1.

- If it is 'CMP' operation, PCselect comparator output becomes 1.

- If it is 'JUMP' operation, PCselect output becomes 1.

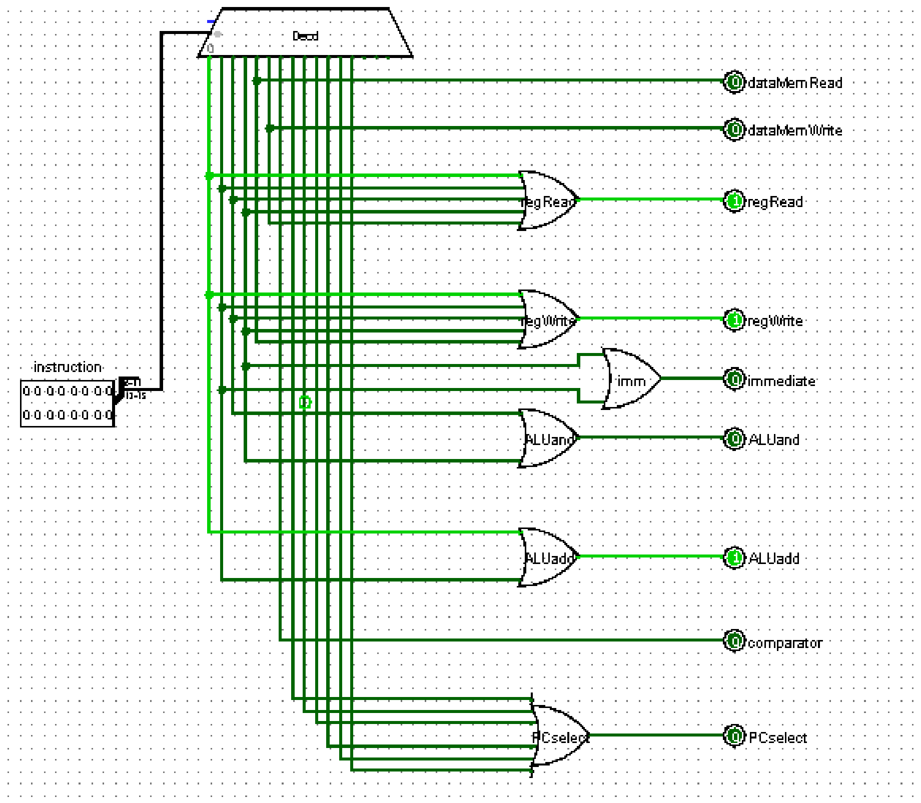
- If it is 'JE' operation, PCselect output becomes 1.

- If it is 'JA' operation, PCselect output becomes 1.

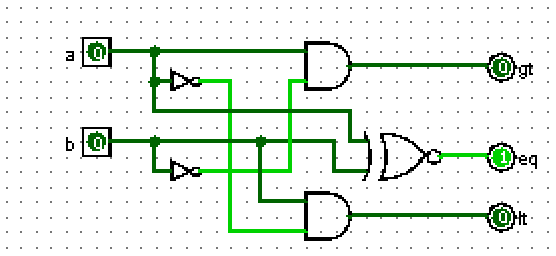
- If it is 'JB' operation, PCselect output becomes 1.

- If it is 'JBE' operation, PCselect output becomes 1.

- If it is 'JAE' operation, PCselect output becomes 1.

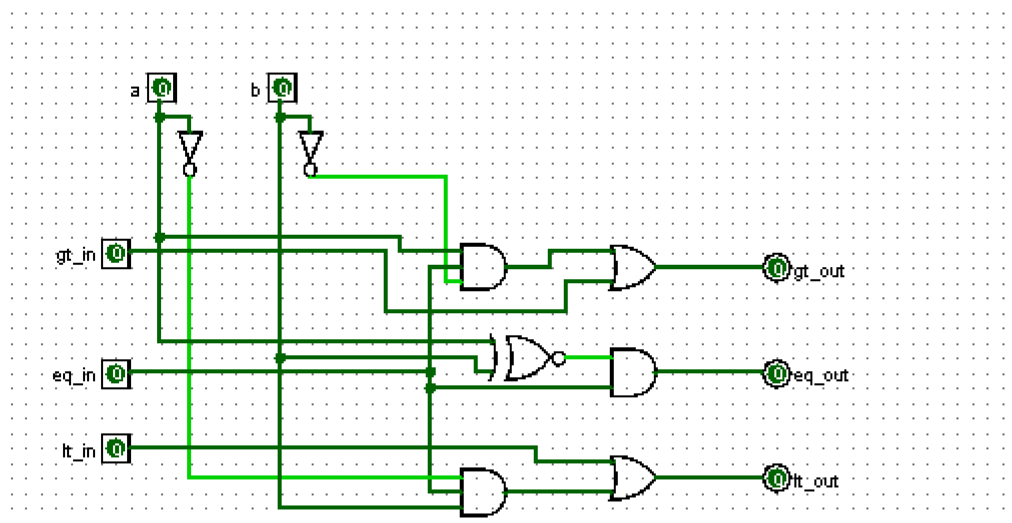


Half Comparator:

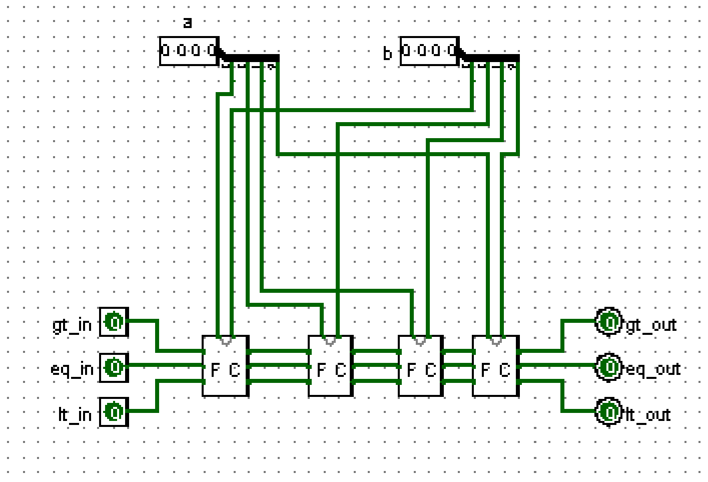


Full Comparator:

Full comparator compares two 1-bit input. If first one is greater than second, then gt_out becomes 1. If they are equal, then eq_out becomes 1 and if second one is greater than first one, then lt_out becomes 1.

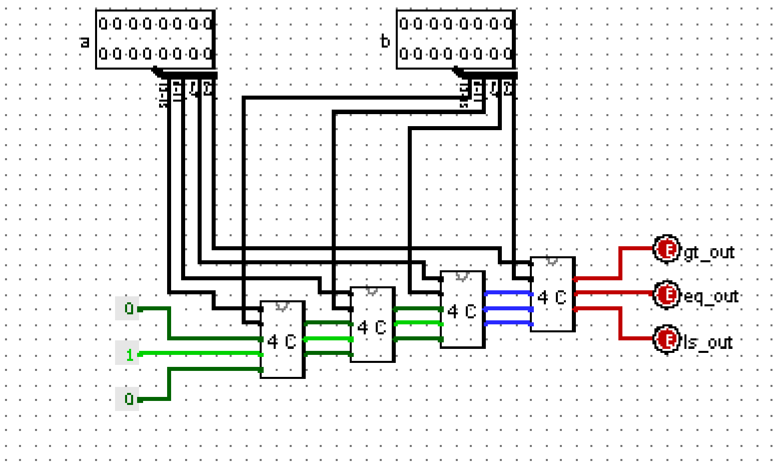


4-bit Comparator: 4-bit comparator compares two 4 bit input. If first one is greater than second, then gt_out becomes 1. If they are equal, then eq_out becomes 1 and if second one is greater than first one, then lt_out becomes 1.



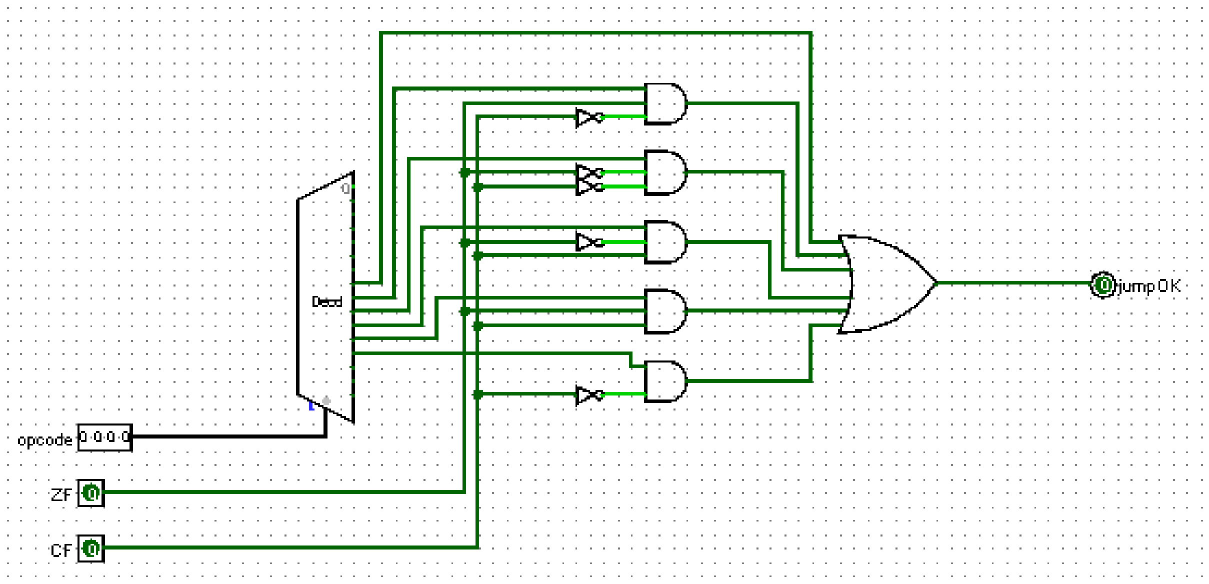
16-bit comparator:

16-bit comparator compares two 1-bit input. If first one is greater than second, then gt_out becomes 1. If they are equal, then eq_out becomes 1 and if second one is greater than first one, then lt_out becomes 1.



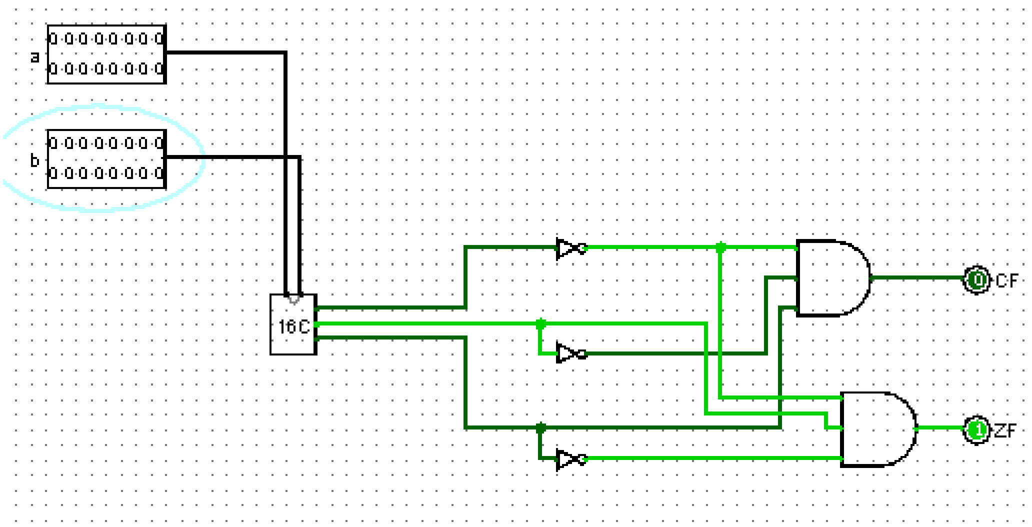
Jump Unit:

Jump Unit controls if there is a jump equation by their opcode. If there is which are ’JUMP’, ’JE’, ’JA’, ’JB’, ’JBE’, ’JAE’, then jumpOK becomes 1.



FLAG UNIT:

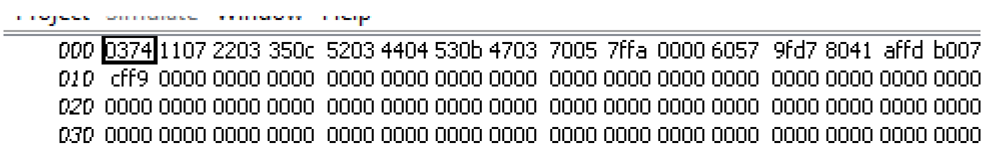
Flag unit compares two 16-bit input.If they are equal then ZF output becomes 1 and CF becomes 0. If First one is greater than second one then both ZF and CF becomes 0. If first input is less than second input than CF becomes 1 and ZF becomes 0.



CPU:

ROM takes 16-bit hexadecimal values as an input which we created in our assembler.

This is our ROM UNIT which is the output of our assembler.



```
000 0374 1107 2203 350c  5203 4404 530b 4703  7005 7ffa 0000 6057  9fd7 8041 affd b007
010  cff9 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
020 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
030 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
```

First, we read input from memory. Control Unit gets input and creates signals. If there is a register read or write signal, then control unit send those signals into register file.

Register file: Instruction is 'ADD' or 'AND' then register file reads two source register and writes the output. If 'ADDI' or 'ANDI' then register file reads source register and makes operation with immediate value and write it into register. After reading register if control unit sends ALUadd or ALUand signal then values into source registers goes into ALU and makes operation. After that output of ALU goes into register and write into DST register

We also have RAM in CPU which stores the data in a specific address. When LOAD signal comes from Control Unit then reads the data in the address and writes it into Register. And when STORE SIGNAL comes from Control Unit then read value from register writes to the given address in the memory.

If Control unit sends comparator signal, then flag unit compares two value in source registers if they are equal then ZF becomes 1 and CF becomes 0. If first registers value is greater than second ,than both ZF and CF values become 0.And if first one is less than second one than CF becomes 1 and ZF becomes 0.

If Control Unit sends PCselect signal, then if there is not a conditional jump, then the value will be set to the PC value. If there is a condition,

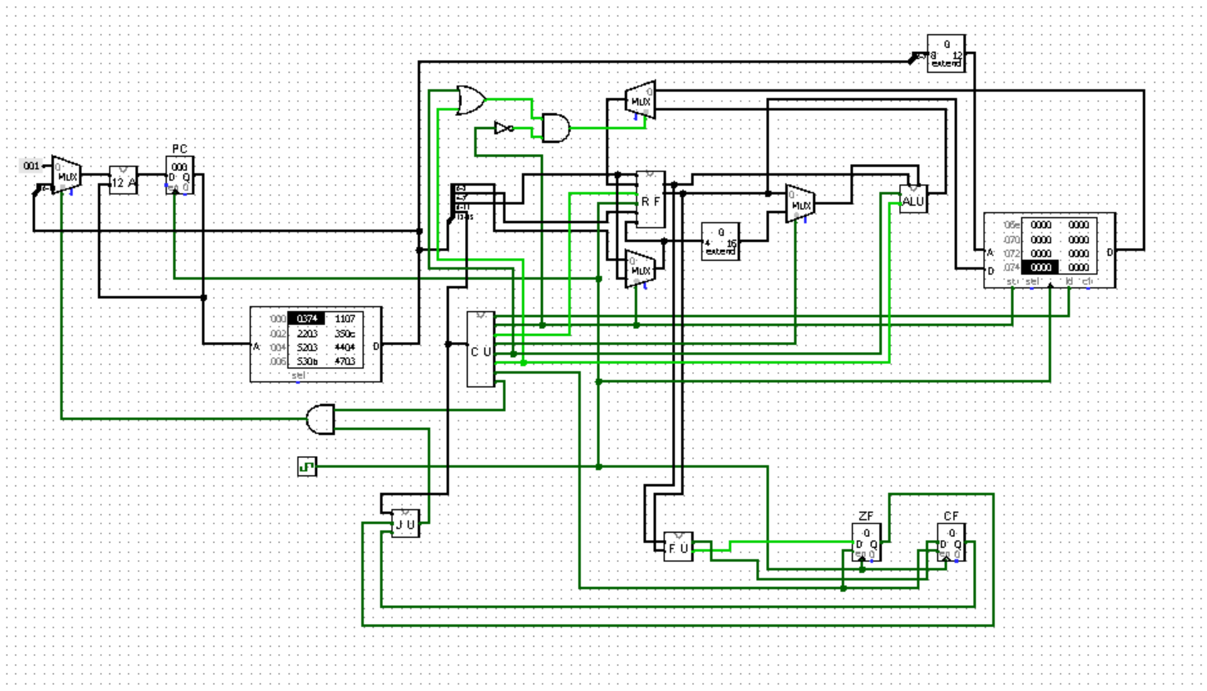
-If there is JE instruction and flag values are ZF=1 and CF=0, then the value will be set to the PC value.

-If there is JA instruction and flag values are ZF=0 and CF=0, then the value will be set to the PC value.

-If there is JB instruction and flag values are ZF=0 and CF=1, then the value will be set to the PC value.

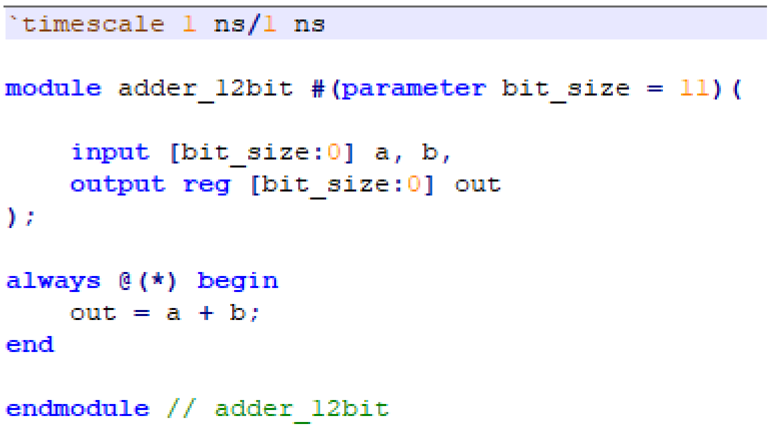
-If there is JAE instruction and flag value of CF=0, then the value will be set to the PC value.

-If there is JBE instruction and flag values are ZF=1 or CF=1, then the value will be set to the PC value.



VERILOG PART:

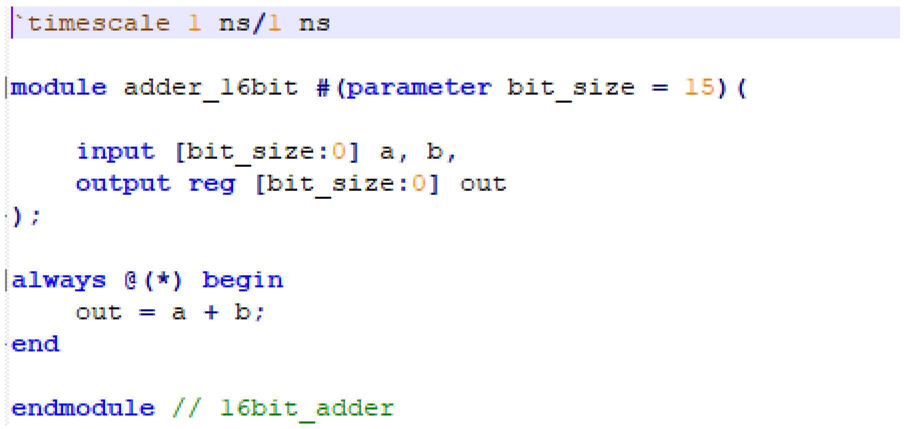
12-bit adder:



```
`timescale 1 ns/1 ns

module adder_12bit #(parameter bit_size = 11)(

    input [bit_size:0] a, b,
    output reg [bit_size:0] out
);

always @(*) begin
    out = a + b;
end

endmodule // adder_12bit
```

12-bit adder adds 12 bit two numbers. If there is a carry out bit then carry out goes to 1.

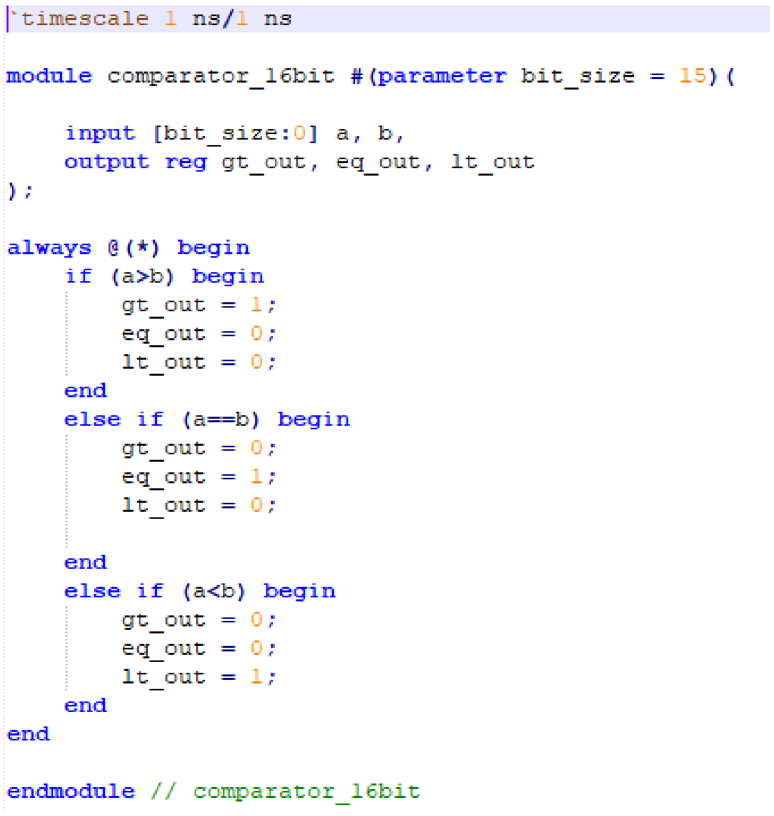
16-bit adder:



```
`timescale 1 ns/1 ns

module adder_16bit #(parameter bit_size = 15)(

    input [bit_size:0] a, b,
    output reg [bit_size:0] out
);

always @(*) begin
    out = a + b;
end

endmodule // 16bit_adder
```

16-bit adder adds 16 bit two numbers. If there is a carry out bit then carry out goes to 1. We used this in register 'ADD' and 'ADDI' instructions.

16-bit Comparator:

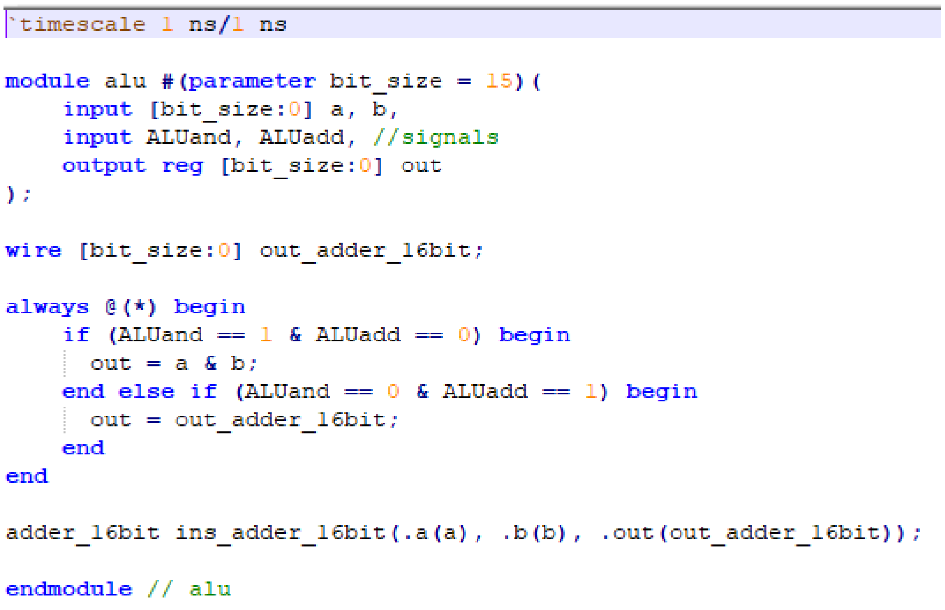


```
`timescale 1 ns/1 ns

module comparator_16bit #(parameter bit_size = 15)(

    input [bit_size:0] a, b,
    output reg gt_out, eq_out, lt_out
);

always @(*) begin
    if (a>b) begin
        gt_out = 1;
        eq_out = 0;
        lt_out = 0;
    end
    else if (a==b) begin
        gt_out = 0;
        eq_out = 1;
        lt_out = 0;

    end
    else if (a<b) begin
        gt_out = 0;
        eq_out = 0;
        lt_out = 1;
    end
end

endmodule // comparator_16bit
```

16-bit comparator compares two 1 bit input. If first one is greater than second, then gt_out becomes 1. If they are equal, then eq_out becomes 1 and if second one is greater than first one, then lt_out becomes 1.

ALU:

In ALU, we make the arithmetic and logic operations. Which are 'ADD', 'AND', 'ADDI' and 'ANDI'. ALU takes two 16-bit input. If it is an AND operation, it makes bitwise 'AND' operation. If it is an ADD operation, it adds 2 input and goes output value to the output.



```
`timescale 1 ns/1 ns

module alu #(parameter bit_size = 15)(
    input [bit_size:0] a, b,
    input ALUand, ALUadd, //signals
    output reg [bit_size:0] out
);

wire [bit_size:0] out_adder_16bit;

always @(*) begin
    if (ALUand == 1 & ALUadd == 0) begin
      out = a & b;
    end else if (ALUand == 0 & ALUadd == 1) begin
      out = out_adder_16bit;
    end
end

adder_16bit ins_adder_16bit(.a(a), .b(b), .out(out_adder_16bit));

endmodule // alu
```

CONTROL UNIT:

Control Unit:

In control unit, Control unit separates input as an opcode.

-If it is an ‘ADD’ operation, regRead, Regwrite and ALUadd output becomes 1.

-If it is ‘ADDI’ operation, regRead, Regwrite, immediate and ALUadd output becomes 1.

- If it is ‘AND’ operation, regRead, Regwrite and ALUand output becomes 1.

- If it is ‘ANDI’ operation, regRead, Regwrite, immediate and ALUand output becomes 1.

- If it is ‘LD’ operation, dataMemRead and regWrite output becomes 1.

- If it is ‘ST’ operation, dataMemWrite and regRead output becomes 1.

- If it is ‘CMP’ operation, PCselect comparator output becomes 1.

- If it is ‘JUMP’ operation, PCselect output becomes 1.

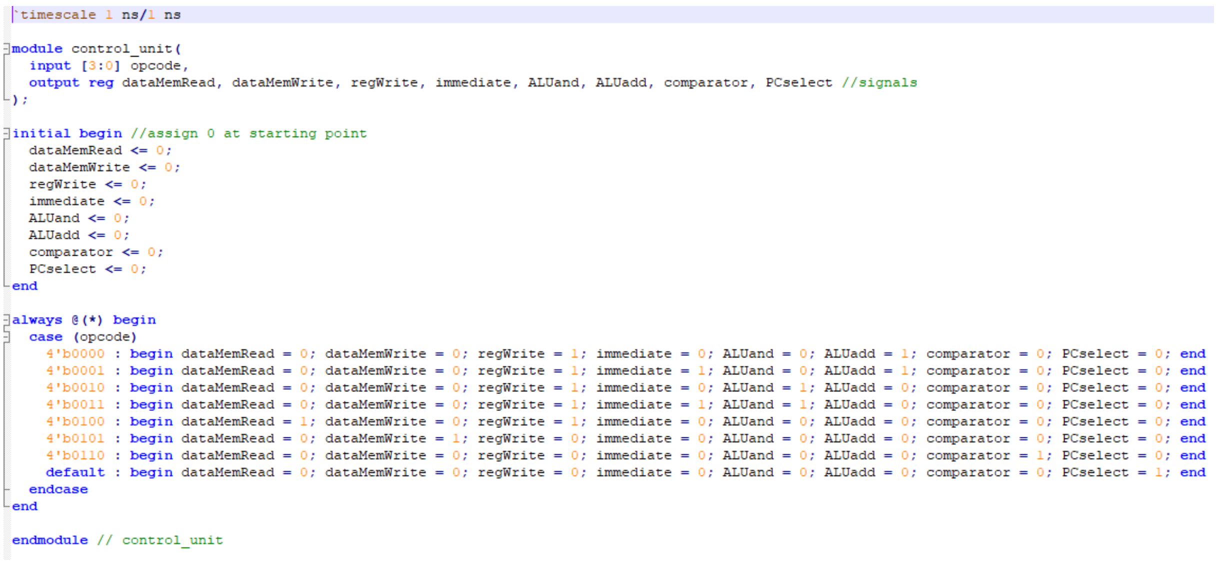
- If it is ‘JE’ operation, PCselect output becomes 1.

- If it is ‘JA’ operation, PCselect output becomes 1.

- If it is ‘JB’ operation, PCselect output becomes 1.

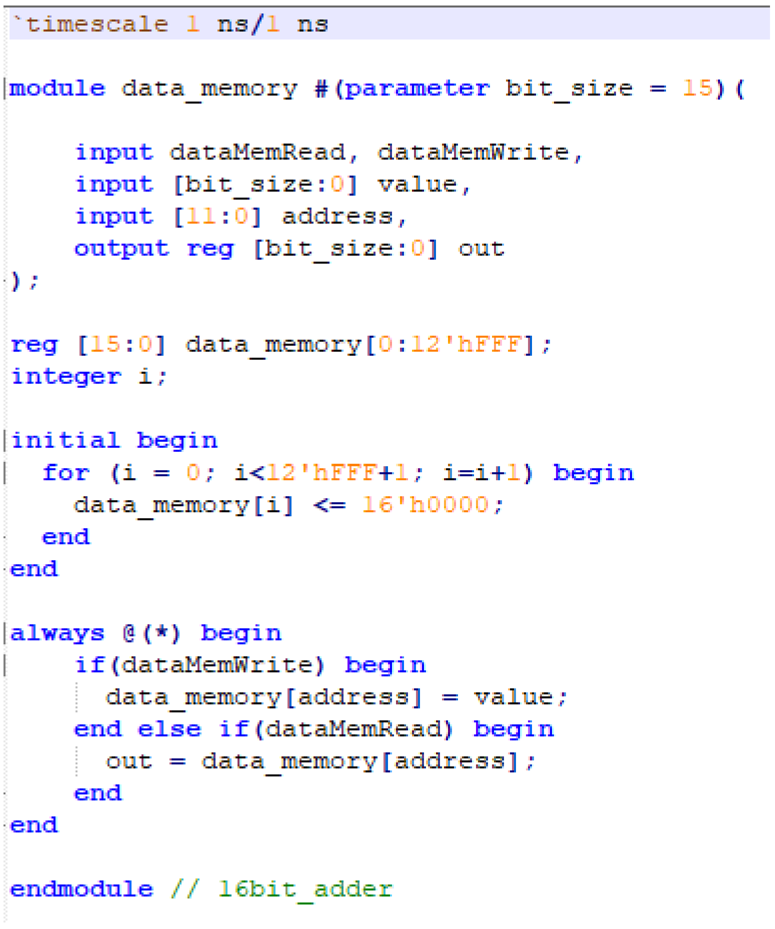
- If it is ‘JBE’ operation, PCselect output becomes 1.

- If it is ‘JAE’ operation, PCselect output becomes 1.



```
`timescale 1 ns/1 ns

module control_unit(
  input [3:0] opcode,
  output reg dataMemRead, dataMemWrite, regWrite, immediate, ALUand, ALUadd, comparator, PCselect //signals
);

initial begin //assign 0 at starting point
  dataMemRead <= 0;
  dataMemWrite <= 0;
  regWrite <= 0;
  immediate <= 0;
  ALUand <= 0;
  ALUadd <= 0;
  comparator <= 0;
  PCselect <= 0;
end

always @(*) begin
  case (opcode)
    4'b0000 : begin dataMemRead = 0; dataMemWrite = 0; regWrite = 1; immediate = 0; ALUand = 0; ALUadd = 1; comparator = 0; PCselect = 0; end
    4'b0001 : begin dataMemRead = 0; dataMemWrite = 0; regWrite = 1; immediate = 1; ALUand = 0; ALUadd = 1; comparator = 0; PCselect = 0; end
    4'b0010 : begin dataMemRead = 0; dataMemWrite = 0; regWrite = 1; immediate = 0; ALUand = 1; ALUadd = 0; comparator = 0; PCselect = 0; end
    4'b0011 : begin dataMemRead = 0; dataMemWrite = 0; regWrite = 1; immediate = 1; ALUand = 1; ALUadd = 0; comparator = 0; PCselect = 0; end
    4'b0100 : begin dataMemRead = 1; dataMemWrite = 0; regWrite = 1; immediate = 0; ALUand = 0; ALUadd = 0; comparator = 0; PCselect = 0; end
    4'b0101 : begin dataMemRead = 0; dataMemWrite = 1; regWrite = 0; immediate = 0; ALUand = 0; ALUadd = 0; comparator = 0; PCselect = 0; end
    4'b0110 : begin dataMemRead = 0; dataMemWrite = 0; regWrite = 0; immediate = 0; ALUand = 0; ALUadd = 0; comparator = 1; PCselect = 0; end
    default : begin dataMemRead = 0; dataMemWrite = 0; regWrite = 0; immediate = 0; ALUand = 0; ALUadd = 0; comparator = 0; PCselect = 1; end
  endcase
end

endmodule // control_unit
```

DATA MEMORY:

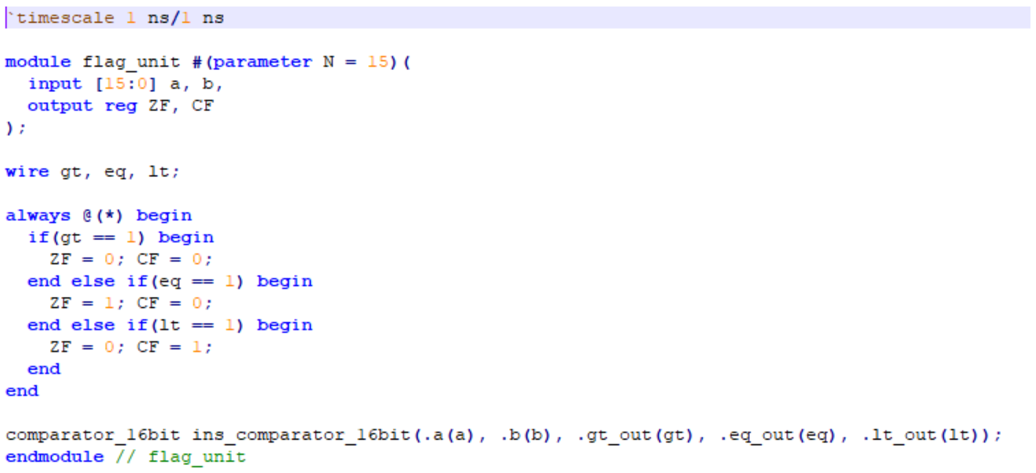


```
`timescale 1 ns/1 ns

module data_memory #(parameter bit_size = 15)(

    input dataMemRead, dataMemWrite,
    input [bit_size:0] value,
    input [11:0] address,
    output reg [bit_size:0] out
);

reg [15:0] data_memory[0:12'hFFF];
integer i;

initial begin
  for (i = 0; i<12'hFFF+1; i=i+1) begin
    data_memory[i] <= 16'h0000;
  end
end

always @(*) begin
    if(dataMemWrite) begin
      data_memory[address] = value;
    end else if(dataMemRead) begin
      out = data_memory[address];
    end
end

endmodule // 16bit_adder
```

In data memory, when dataMemWrite signal comes, then value will be written into memory. If dataMemRead signal comes, then value will read from memory.

FLAG UNIT:

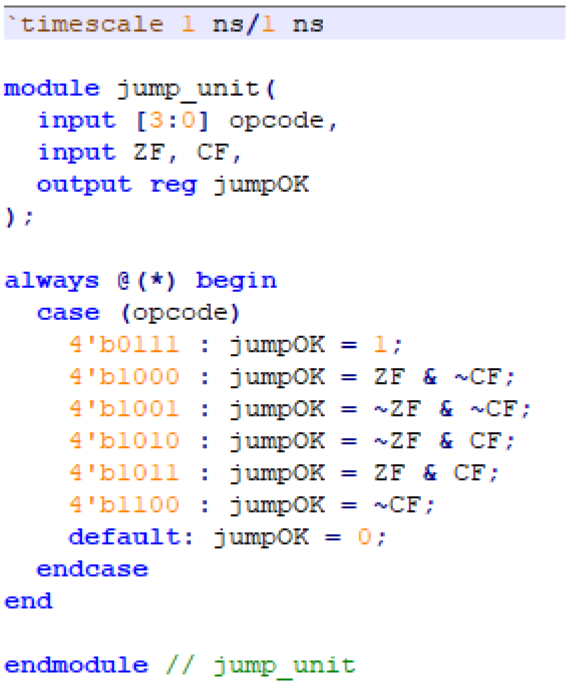
Flag unit compares two 16-bit input.If they are equal then ZF output becomes 1 and CF becomes 0. If First one is greater than second one then both ZF and CF becomes 0. If first input is less than second input than CF becomes 1 and ZF becomes 0.



```
`timescale 1 ns/1 ns

module flag_unit #(parameter N = 15)(
  input [15:0] a, b,
  output reg ZF, CF
);

wire gt, eq, lt;

always @(*) begin
  if(gt == 1) begin
    ZF = 0; CF = 0;
  end else if(eq == 1) begin
    ZF = 1; CF = 0;
  end else if(lt == 1) begin
    ZF = 0; CF = 1;
  end
end

comparator_16bit ins_comparator_16bit(.a(a), .b(b), .gt_out(gt), .eq_out(eq), .lt_out(lt));
endmodule // flag_unit
```

JUMP UNIT:

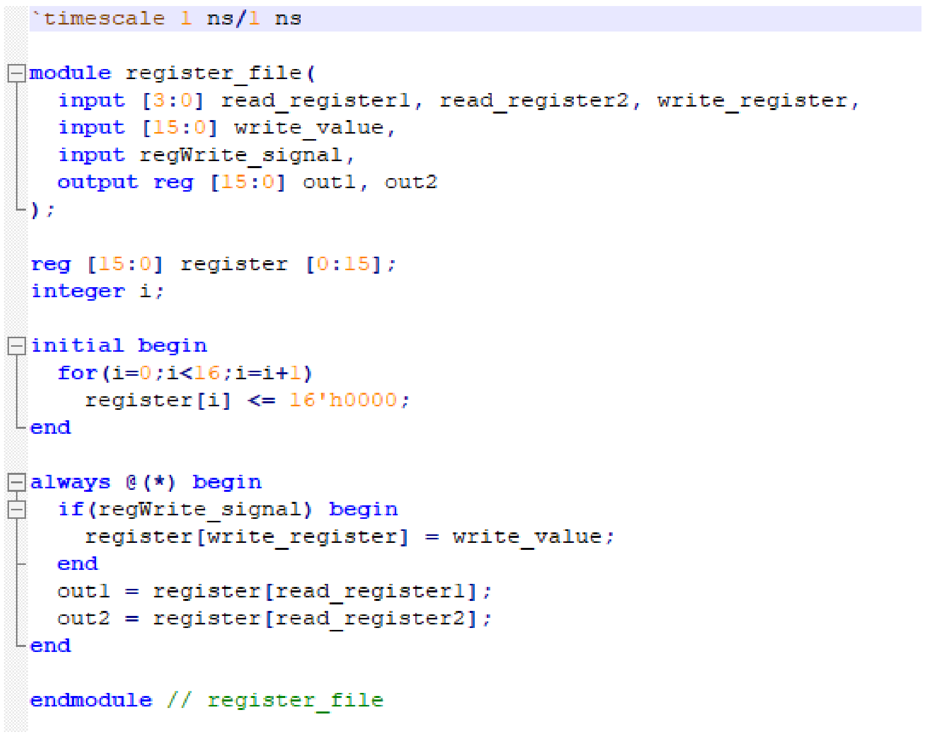
Jump Unit controls if there is a jump equation by their opcode. If there is which are 'JUMP', 'JE', 'JA', 'JB', 'JBE', 'JAE', then jumpOK becomes 1.



```
`timescale 1 ns/1 ns

module jump_unit(
  input [3:0] opcode,
  input ZF, CF,
  output reg jumpOK
);

always @(*) begin
  case (opcode)
    4'b0111 : jumpOK = 1;
    4'b1000 : jumpOK = ZF & ~CF;
    4'b1001 : jumpOK = ~ZF & ~CF;
    4'b1010 : jumpOK = ~ZF & CF;
    4'b1011 : jumpOK = ZF & CF;
    4'b1100 : jumpOK = ~CF;
    default: jumpOK = 0;
  endcase
end

endmodule // jump_unit
```

Register File:

Register file is a combination of registers. It is an array of processor registers in CPU. Register file contains bits of data and mapping these data into locations.



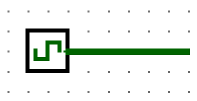
```
`timescale 1 ns/1 ns

module register_file(
  input [3:0] read_register1, read_register2, write_register,
  input [15:0] write_value,
  input regWrite_signal,
  output reg [15:0] out1, out2
);

reg [15:0] register [0:15];
integer i;

initial begin
  for(i=0;i<16;i=i+1)
    register[i] <= 16'h0000;
end

always @(*) begin
  if(regWrite_signal) begin
    register[write_register] = write_value;
  end
  out1 = register[read_register1];
  out2 = register[read_register2];
end

endmodule // register_file
```

Erman HAVUÇ
150114023

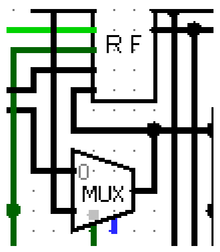
İbrahim AYCAN
150114075

CPU:

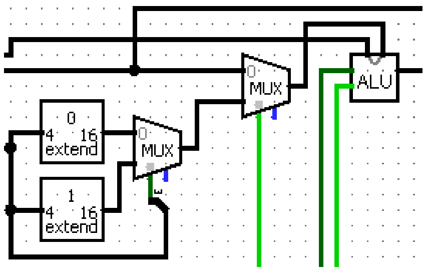
```
always begin  //generate clock cycle
  #50 clock = ~clock;
end
```



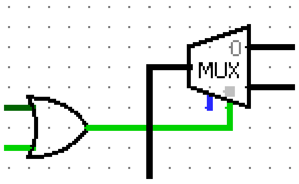
```
  read_register1 = instruction[7:4];
  write_register = instruction[11:8];
  dataMemAddress = {4'h0, instruction[7:0]};

  if(dataMemWrite) begin
    read_register2 = instruction[11:8];
  end else begin
    read_register2 = instruction[3:0];
  end
```



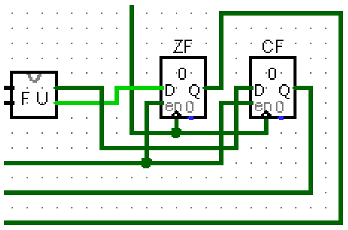
```
 if(immediate) begin
    if(read_register2[3]) begin
      alu_in = {12'hFFF, read_register2};
    end else begin
      alu_in = {12'h000, read_register2};
    end
  end else begin
    alu_in = reg_out2;
  end
```



```
  if(ALUadd | ALUand) begin
    write_value = alu_out;
  end else if(dataMemRead)begin
    write_value = data_memory_out;
  end
```

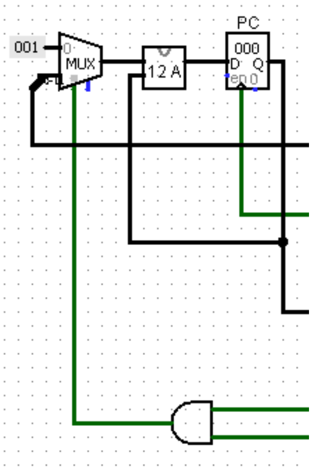


```
  if(comparator) begin
    ZF = zf;
    CF = cf;
  end
```



```
  if(PCselect & jumpOK) begin
    PC_flag = 1;
  end else begin
    PC_flag = 0;
  end
end
```

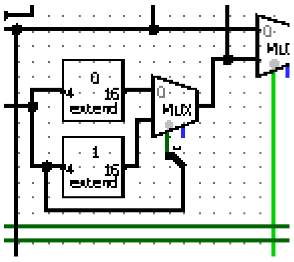
```
  if(PC_flag) begin
    PC = PC + instruction[11:0];
  end else begin
    PC = PC + 1;
  end
```



CORRECTION OF SOME MISTAKES:

-One of the mistakes of that we made is our registers. We made our registers but by using them we had some errors. This error is caused by Logisim itself. So, we used built-in register of Logisim.

-Another mistake is than when we are extending the immediate value to 16-bit, we accidently extend them by adding 0 in front of immediate value. But this caused a problem. When immediate value is negative and when we extend it with 0 then value became positive and cause mistake of calculation. We solved this problem by adding decoder and if value is positive, we extended it with 0's and if value is negative, then we extended it with 1's.



-In firstly we take data memory as an array list in the CPU. But when we simulated it, a synchronization error occurred. We created a module for data memory and issue fixed.