24/12/2018

CSE 315 Project Report

Student 1: Mert Kelkit – 150115013

Student 2: Furkan Nakıp – 150115032

About the Project

In this project, we are supposed to design a basic CPU which has the 12 bit address size, 16 bit data size, 16 registers and supports instructions ADD, AND, ADDI, ANDI, LD, ST, CMP, JUMP, JE, JA, JB, JAE, JBE.

- ADD instruction will have the form "ADD DEST,SRC1,SRC2. Here DEST, SRC1, SRC2 are the registers. DEST <- SRC1 + SRC2.
- AND instruction will have the form "AND DEST,SRC1,SRC2. Here DEST, SRC1, SRC2 are the registers. DEST <- SRC1 & SRC2.
- ADDI instruction will have the form "ADD DEST,SRC1,IMM. Here DEST, SRC1 are the registers, IMM is a signed decimal number. DEST <- SRC1 + IMM.
- ANDI instruction will have the form "ADD DEST,SRC1,IMM. Here DEST, SRC1 are the registers, IMM is a signed decimal number. DEST <- SRC1 & IMM.
- LD instruction will have the form "LD DEST,ADDR". DEST is the register number, ADDR is the address of the data which will be fetched from data memory to the DEST register.
- ST instruction will have the form "ST SRC,ADDR" SRC is the register number, ADDR is the address. This instruction will store the data in the SRC register to the ADDR of the data memory.
- CMP instruction will have the form "CMP OP1,OP2". OP1 and OP2 are register numbers. This instruction will compare values inside OP1 and OP2. If OP1 > OP2, ZF and CF will be 0. If OP1 = OP2, ZF will be 1, CF will be 0. If OP1 < OP2, ZF will be zero, CF will be 1. These flag values will be used with jump instructions.
- JUMP, JE, JA, JB, JAE, JBE instructions will have the form "x ADDR". x is a type of jump instruction, ADDR is the PC-relative signed address. If instruction is JUMP, ADDR will be added to the PC. If Instruction is not JUMP, this addition operation will be done if flag values are satisfied according to the instruction type.

Part 1 - Assembler

After understanding instructions above, we know that we have to design an instruction set architecture for all the instructions in order to make computer (our CPU) understand our instructions. First we designed an ISA, then we wrote an assembler according to ISA for converting these instructions to the machine code that our CPU will understand.

We used the Python programming language for its easy usage and fast development. Our delivery for assembler part contains two files: "assembler.py", "use_test.py".

24/12/2018

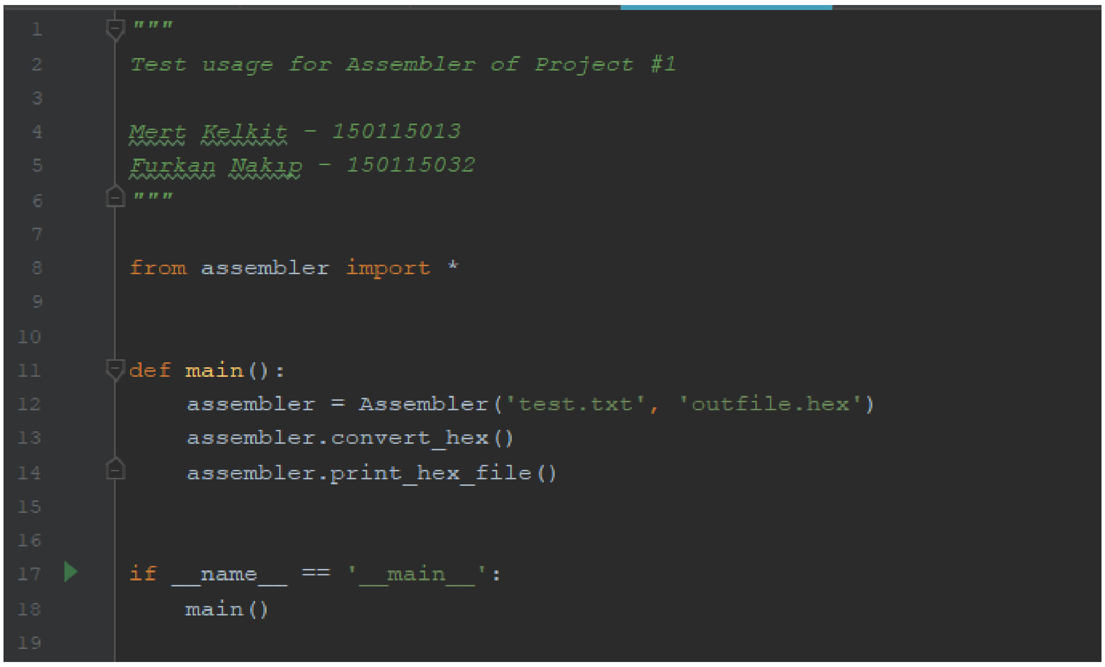
Here is our ISA:

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

Since we have 13 instructions, 4 bit allocation is enough to distinguish each instruction. We have 16 registers, so we allocated 4 bits to distinguish registers.

As we can see, we have some limitations. IMM value will be represented as two's complement and we allocated 4 bits for it. So, IMM value will be in the range [-8, 7]. Because of our data being 16 bits, we need to sign extend IMM bits. Also for LD and ST instructions, 8 bits are allocated for ADDR. Since we have 12 bit addresses – that means data memory will have 2^{12} = 4096 spaces –, ADDR will be in the range [0, 255]. We also need to extend ADDR value to 12 bit.

In our assebler code, we have classes to convert instructions to the hex codes. Here is a test usage:



```python
"""
Test usage for Assembler of Project #1

Mert Kelkit - 150115013
Furkan Nakıp - 150115032
"""

from assembler import *


def main():
    assembler = Assembler('test.txt', 'outfile.hex')
    assembler.convert_hex()
    assembler.print_hex_file()


if __name__ == '__main__':
    main()
```

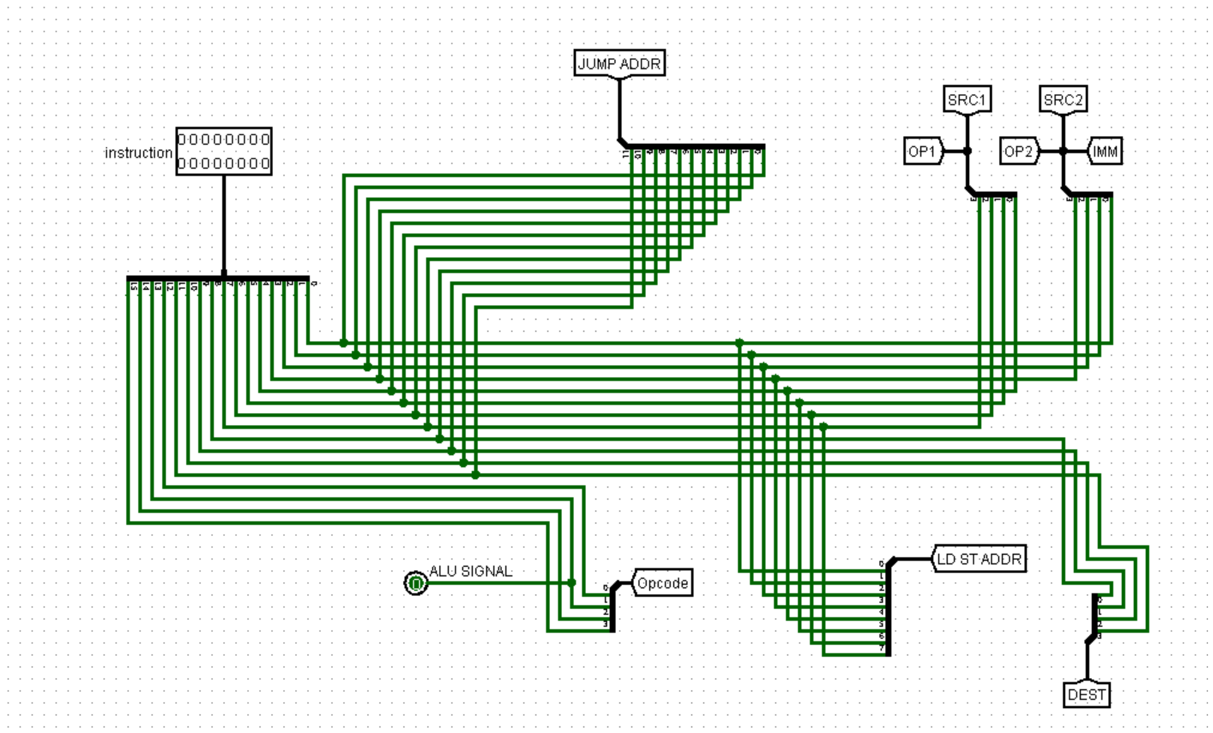
Assembler class needs two parameters, name of input file which containts instructions; name of output file which will have corresponding hex codes of instructions.

Part 2 – Logisim Design

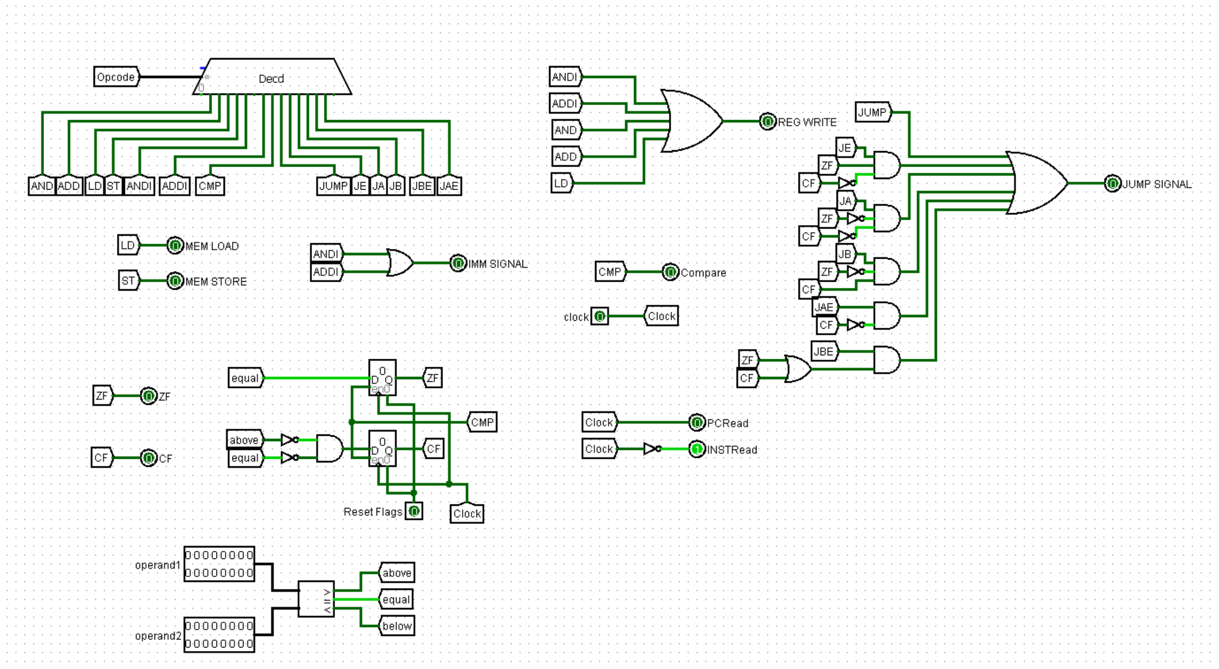
In this part, we are supposed to design our CPU in Logisim software. We have to design a register file containing 16 registers, a control unit parses instructions and send signals according to given instruction, an arithmetic logic unit in order to make calculations (in our case ADD and AND operations).

1- Control Unit:

Below, we parsed our instruction according to our ISA.



Here is our signal handling:



If operation is LD, MEM LOAD signal will be 1. If operation is ST, MEM STORE signal will be 1.

If there is an operation that contains immediate values (ANDI, ADDI), IMM SIGNAL will be 1.

If there is a compare operation, Compare will be one. This value will be used for enable pins of flag registers. This is for we want to change flags if and only if there is a compare operation.

If there is an operation that requires writing values to the registers (ANDI, ADDI, AND, ADD, LD), REG WRITE will be 1.

If there is a type of jump instruction and flag values are satisfied, JUMP SIGNAL will be 1.

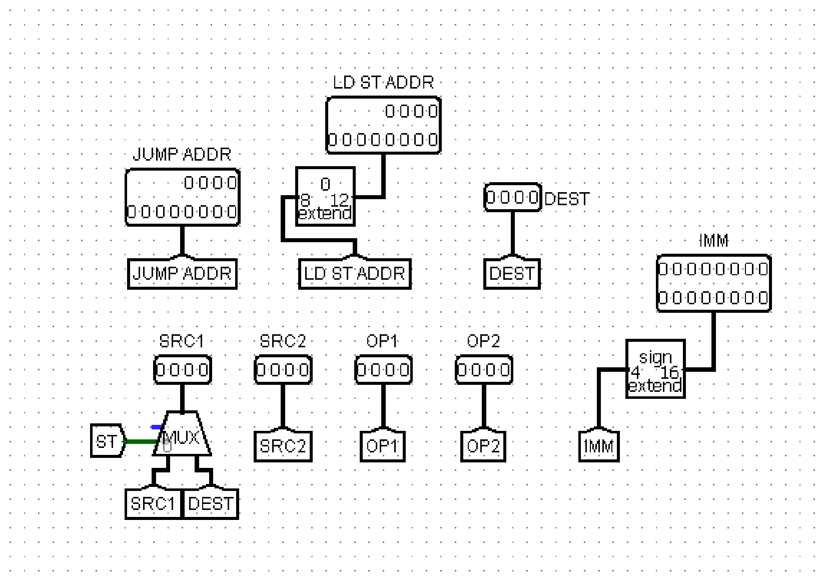
These flag conditions are:

- JE : ZF = 1 and CF = 0
- JA : ZF = 0 and CF = 0
- JB : ZF = 0 and CF = 1
- JAE: CF = 0
- JBE: ZF = 1 or CF = 1

When clock is high, PCRead signal will be 1, that means new PC value will be fetched.

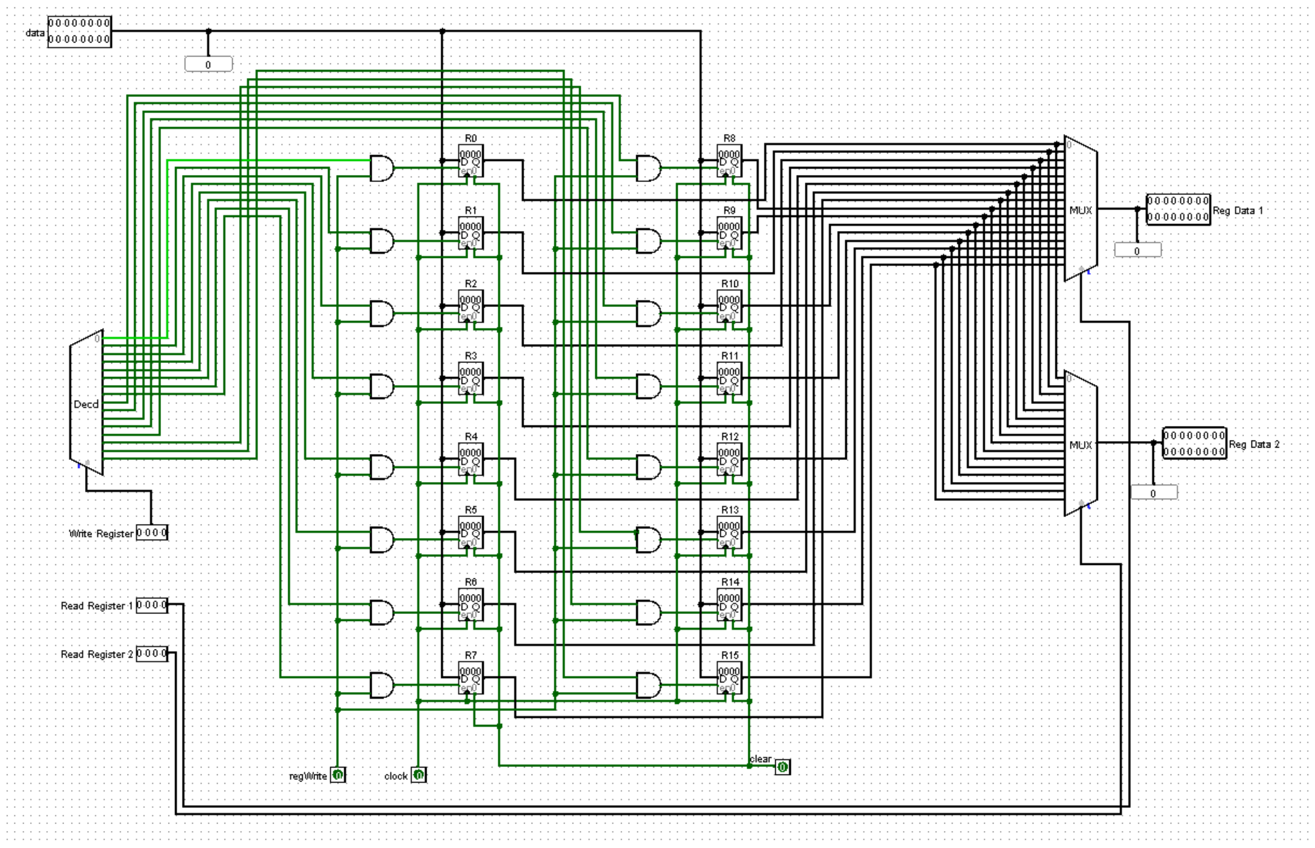
When clock is low, INSTRead signal will be 1, that means current instruction that PC address shows will fetched from instruction memory and will be executed. By this way, PCRead and INSTRead signals will never be 1 at the same time.

Here are the control unit outputs:



SRC1, SRC2, OP1, OP2, DEST values will be used with register file, IMM value will be used with ALU, JUMP ADDR will be used with PC Register, LD ST ADDR will be used with data memory. If operation is ST, register number called DEST will be used as SRC1 register, because of our ISA.

2- Register File:



Our register file has 7 inputs:

1- data: 16 bit data to store in a register.
2- Write Register: 4 bit input to select which register to store data.
3- Read Register 1: To select a register to output value inside of it.
4- Read Register 2: To select a register to output value inside of it.
5- regWrite: Enable signal for registers, if it's 1, CPU will be able to store a value inside a register.
6- clock: Shared clock value for all registers.
7- clear: Shared clear bit for all registers.

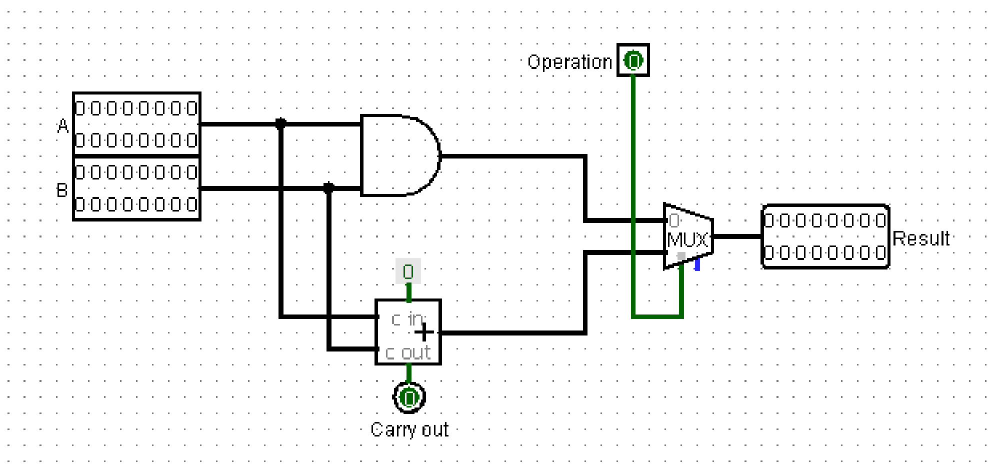
Our register file has 2 outputs:

1- Reg Data 1: Data stored inside register has the number "Read Register 1".
2- Reg Data 2: Data stored inside register has the number "Read Register 2".

We used a decoder to select register which the data will be stored.

We used multiplexers to select values stored inside registers, register numbers are used as select bits of these multiplexers.

3- Arithmetic Logic Unit:



Our ALU has 2 operations, ADD and AND, and 3 inputs:

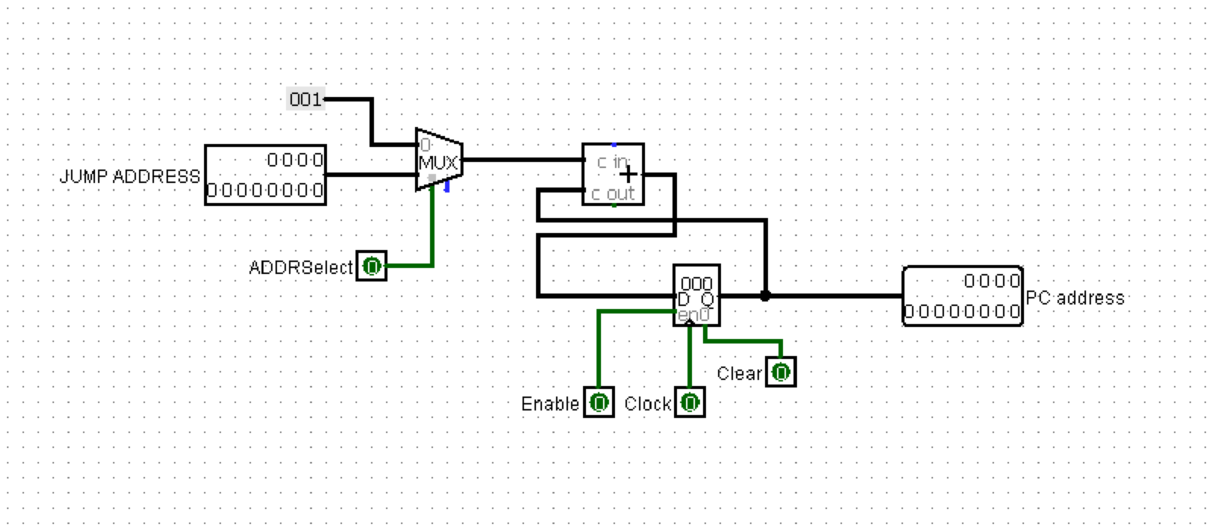
1- A: 16 bit data
2- B: 16 bit data
3- Operation: Select bit for operation multiplexer. If it's 1, ADD output will be selected. If it's 0, AND output will be selected as output.

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 and 16 bit adder were used in this component)

4- PC Register:



We designed an additional subcircuit to handle PC increment operations and storing current PC. If there is a jump instruction, JUMP ADDRESS will be added to PC. If there is not, 1 will be added to PC.

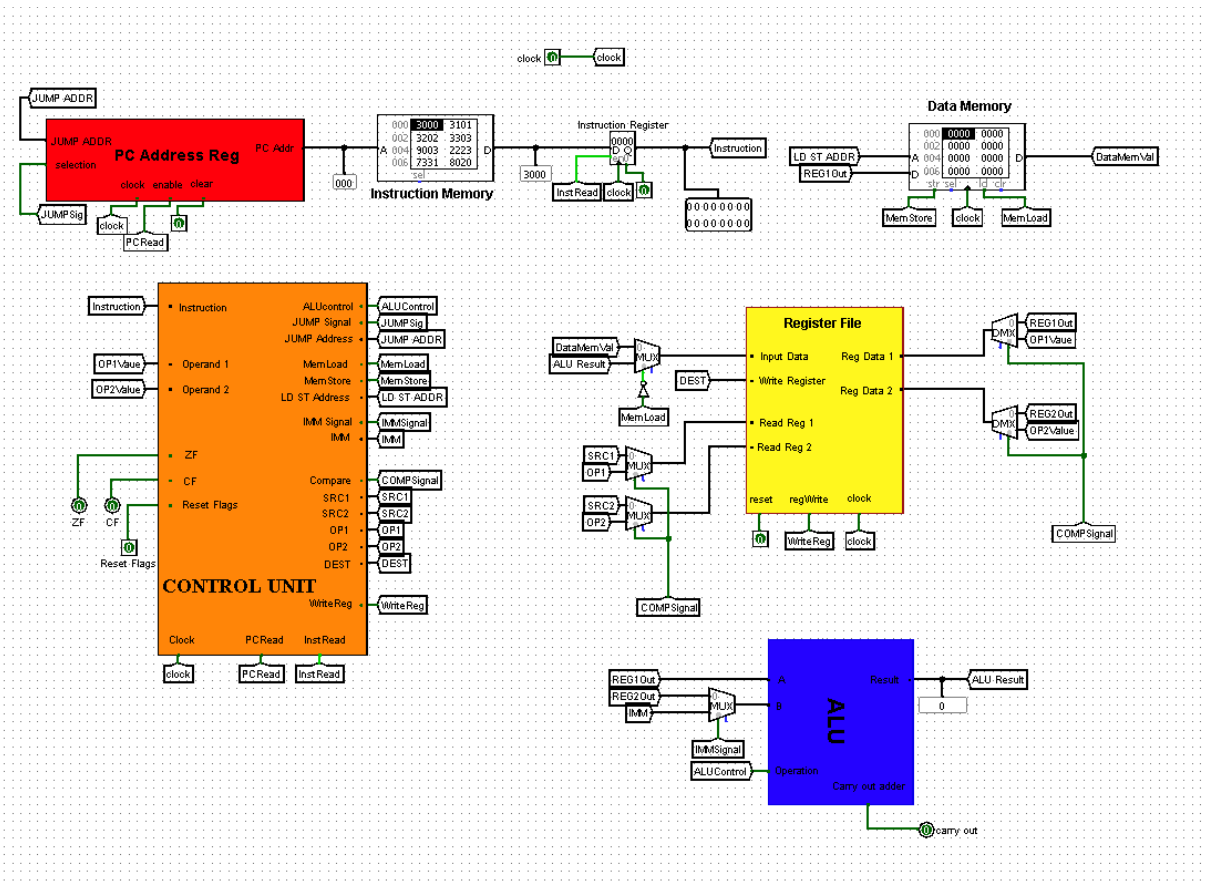
This component has 5 inputs:

1- JUMP ADDRESS: This input will come from cotrol unit, parsed JUMP ADDR value of given instruction.
2- ADDRSelect: This value will be 1 if JUMP SIGNAL is 1. It selects ADDR value of jump instruction if its value is 1.
3- Enable: This value will be 1 if PCRead signal is 1. This will update the value of register.
4- Clock: Clock for the register(This register triggered on falling edge due to synchronization problems).
5- Clear: Clear bit for the PC Register.

There is also one output:

1- PC address: Outputs updated PC value.

5- CPU:

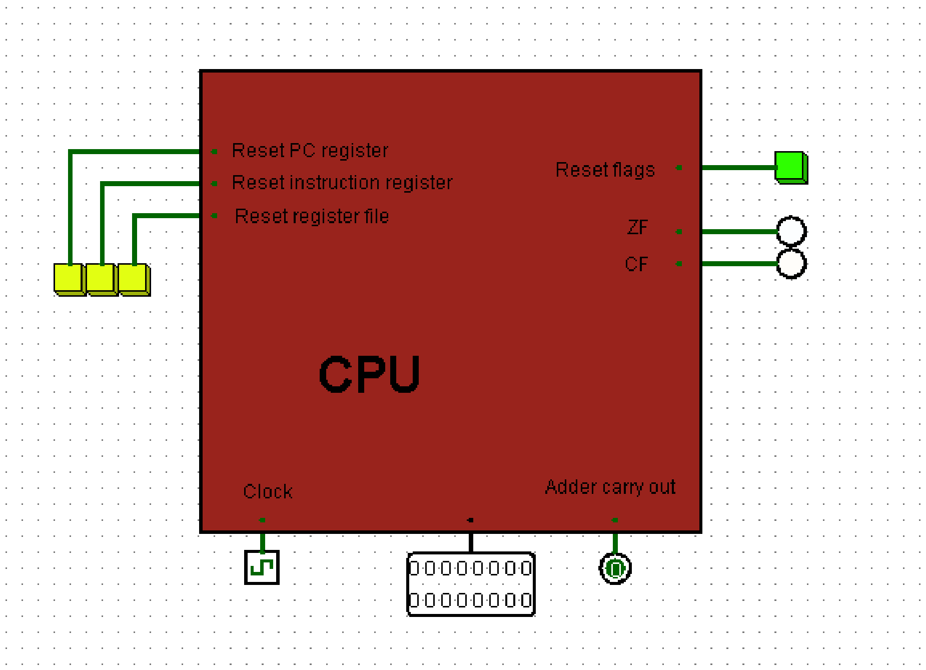


This is the whole CPU, all components are connected.

When InstRead signal is 1, instruction that PC shows will be fetched from memory and executed. When PCRead signal is 1, next instruction address will be evaluated.

ALU can have two operands at the same times, first one (A) came from SRC1 register, second one (B) either come from SRC2 register or IMM value according to IMMSignal.

Register file input data can come either from ALU or data memory according to MemLoad signal. This value will be written to DEST register if regWrite signal is 1. Inputs for reading register values will be OPi or SRCi (i=1,2) according to CompareSignal. Their output will be OPiValue or SRCiValue (i=1,2) also according to CompareSignal.



Above, CPU as a black box. We have reset buttons to reset flags and registers, a clock as input, flags, current instruction and adder carry out as outputs.

Part 3 – Verilog Design

In this part, we need to implement our design with Verilog language, based on our Logisim design. We used ModelSim simulation software while testing.

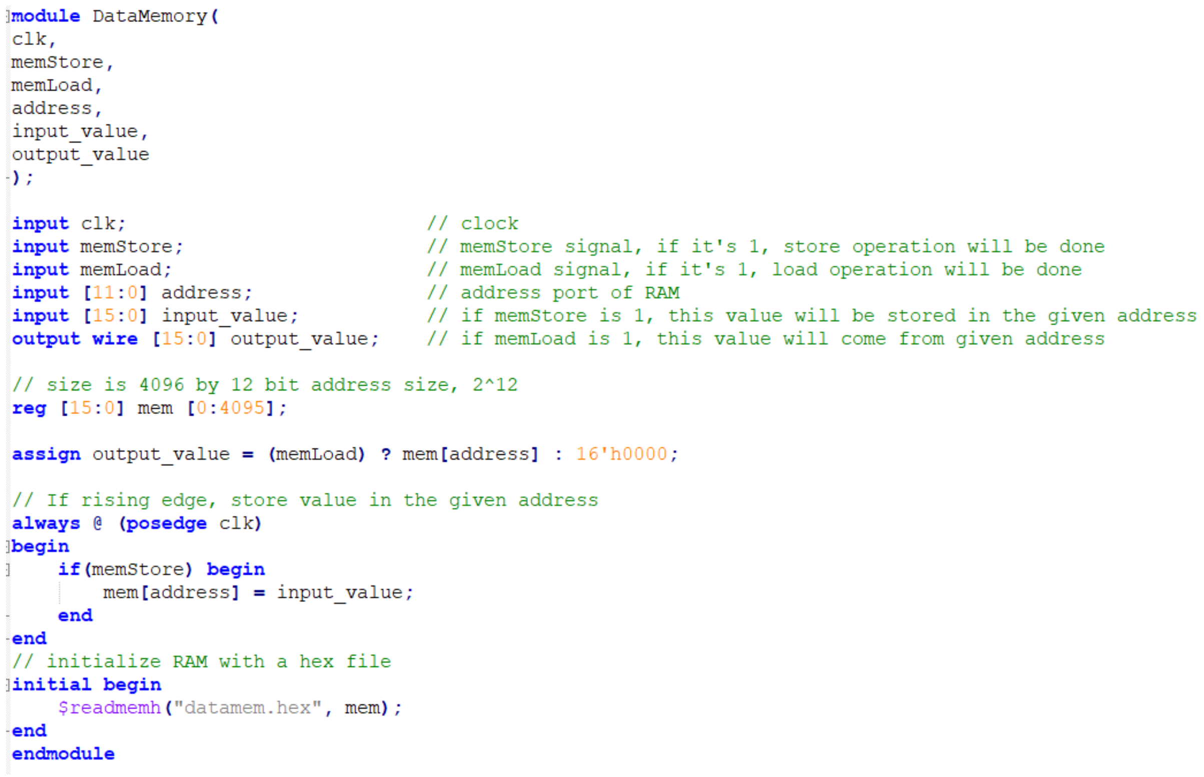
Our implementation contains 7 modules:

1- Arithmetic Logic Unit (ALU.v):

```
module ALU (
A,
B,
operation,
carry_out,
result
);
// A and B are operands
input [15:0] A;
input [15:0] B;
// operation signal (ALUSignal, comes from control unit). If it's 1, addition operation will be done, else and operation will be done
input operation;
// carry out coming from adder
output wire carry_out;
// result of operation
output wire [15:0] result;
// carry in is a constant, 0
parameter carry_in = 1'b0;
// assign result with vector concatenation
assign {cout, result} = (operation) ? carry_in + B + A : {1'b0, A & B};
endmodule
```

There are 2 inputs which have 16 bits lengths. We are adding them or anding them according to operation selection signal.

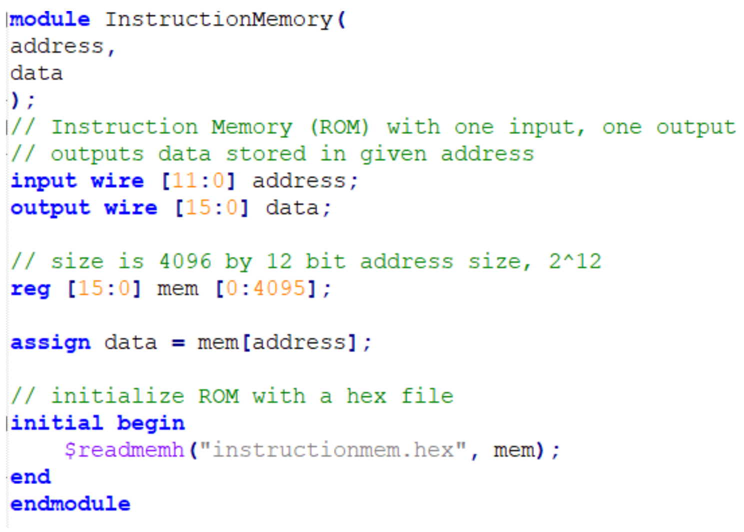
2- Data Memory (RAM) (DataMemory.v)



```verilog
module DataMemory(
clk,
memStore,
memLoad,
address,
input_value,
output_value
);

input clk;                          // clock
input memStore;                     // memStore signal, if it's 1, store operation will be done
input memLoad;                      // memLoad signal, if it's 1, load operation will be done
input [11:0] address;               // address port of RAM
input [15:0] input_value;           // if memStore is 1, this value will be stored in the given address
output wire [15:0] output_value;    // if memLoad is 1, this value will come from given address

// size is 4096 by 12 bit address size, 2^12
reg [15:0] mem [0:4095];

assign output_value = (memLoad) ? mem[address] : 16'h0000;

// If rising edge, store value in the given address
always @ (posedge clk)
begin
    if(memStore) begin
        mem[address] = input_value;
    end
end
// initialize RAM with a hex file
initial begin
    $readmemh("datamem.hex", mem);
end
endmodule
```

There will be 4096 16-bit sized spaces in the RAM, because we have 12 bits address size. There is only one address port for both load and store operations. This address will be used according to memLoad and memStore signals. If memStore is 1, input value will be stored in the given address. If memLoad is 1, data inside given address will be output_value.

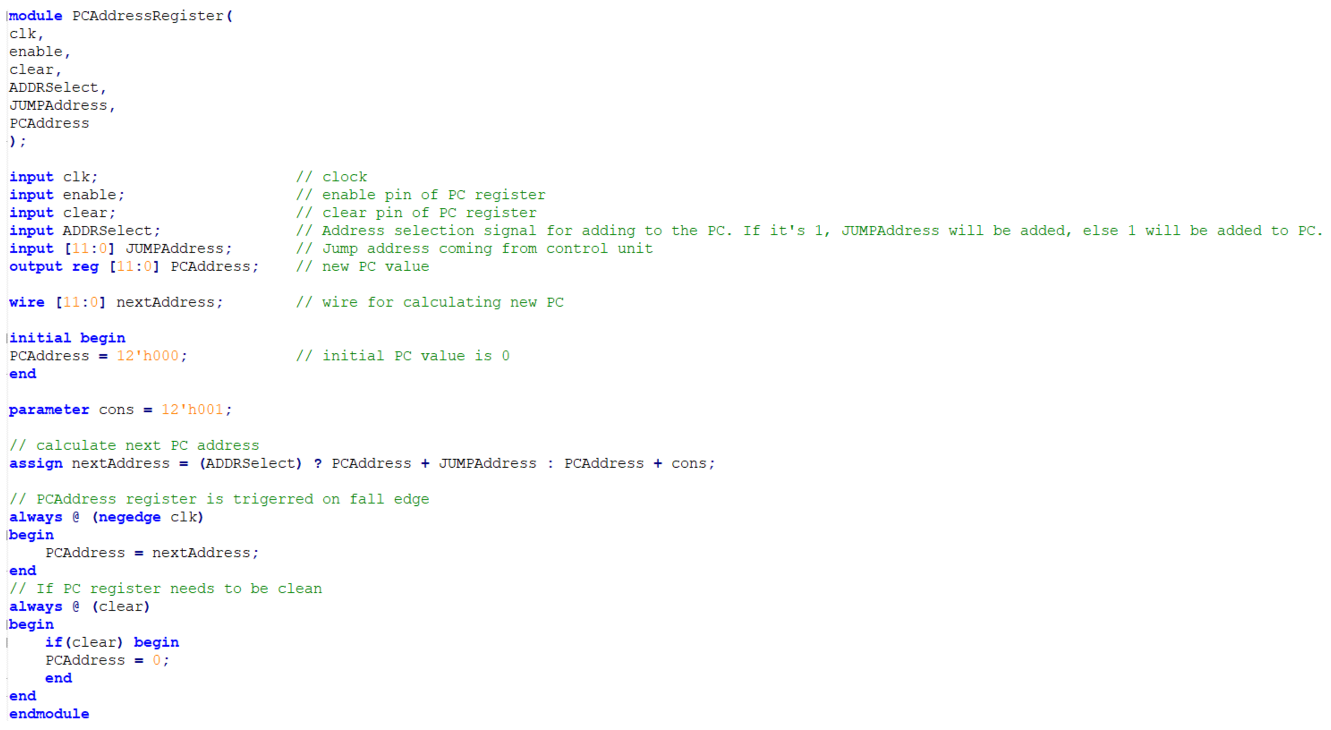
3- Instruction Memory (ROM) (InstructionMemory.v)



```verilog
module InstructionMemory(
address,
data
);
// Instruction Memory (ROM) with one input, one output
// outputs data stored in given address
input wire [11:0] address;
output wire [15:0] data;

// size is 4096 by 12 bit address size, 2^12
reg [15:0] mem [0:4095];

assign data = mem[address];

// initialize ROM with a hex file
initial begin
    $readmemh("instructionmem.hex", mem);
end
endmodule
```

There is only one input port and one output port. This module will output instruction stored in given address.

4- PC Address Register (PCAddressRegister.v)

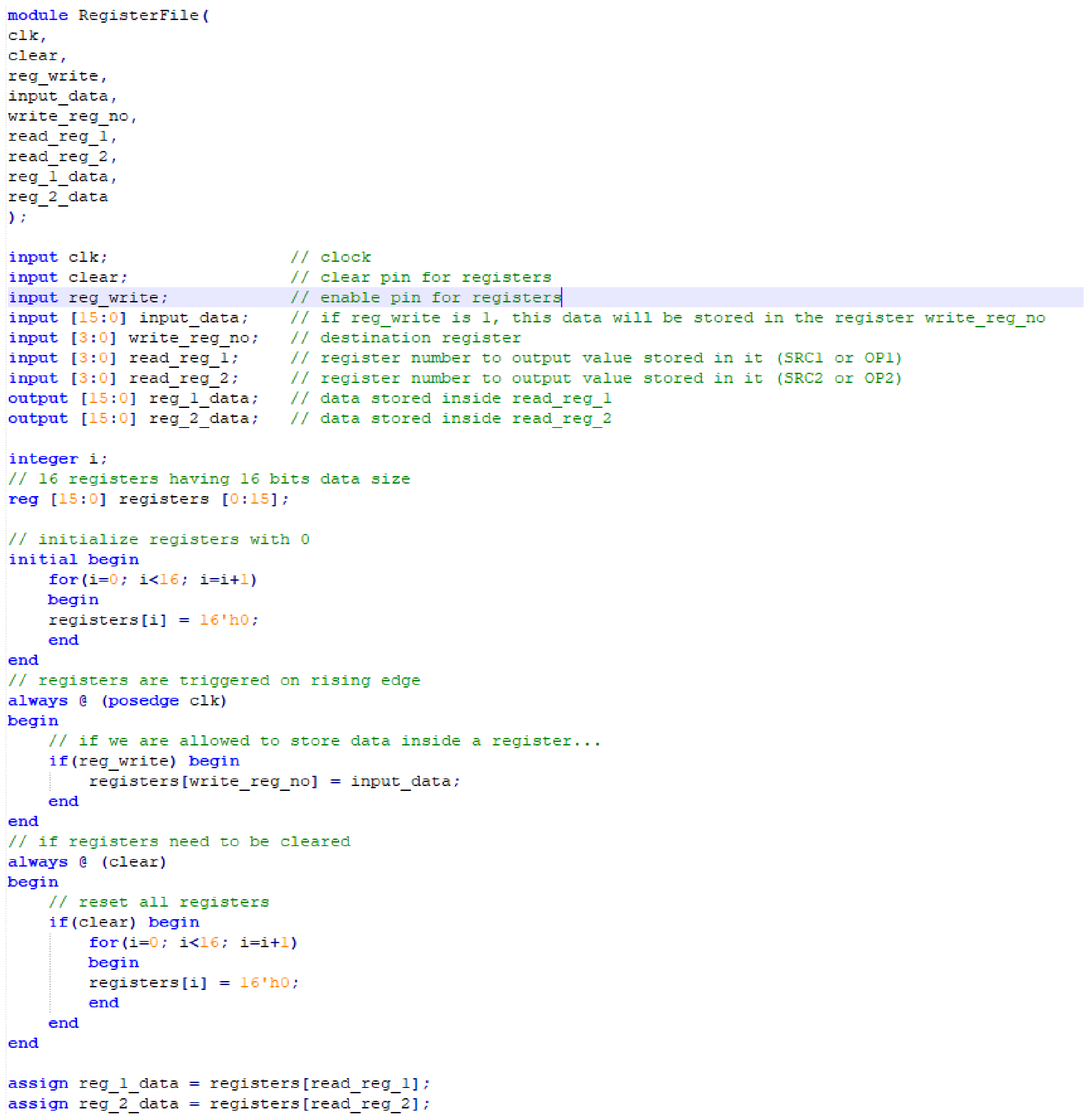


```
module PCAddressRegister(
clk,
enable,
clear,
ADDRSelect,
JUMPAddress,
PCAddress
);

input clk;                       // clock
input enable;                    // enable pin of PC register
input clear;                     // clear pin of PC register
input ADDRSelect;                // Address selection signal for adding to the PC. If it's 1, JUMPAddress will be added, else 1 will be added to PC.
input [11:0] JUMPAddress;        // Jump address coming from control unit
output reg [11:0] PCAddress;     // new PC value

wire [11:0] nextAddress;         // wire for calculating new PC

initial begin
PCAddress = 12'h000;             // initial PC value is 0
end

parameter cons = 12'h001;

// calculate next PC address
assign nextAddress = (ADDRSelect) ? PCAddress + JUMPAddress : PCAddress + cons;

// PCAddress register is trigerred on fall edge
always @ (negedge clk)
begin
    PCAddress = nextAddress;
end
// If PC register needs to be clean
always @ (clear)
begin
    if(clear) begin
    PCAddress = 0;
    end
end
endmodule
```

This module calculates the next PC value according to JUMPSignal and PC relative jump address, updates and outputs the new PC value when clock is on falling edge.

If ADDRSelect is 1, that means we are able to do jump operation, JUMPAddr will be added to the current PC. Else, 1 will be added to the current PC value.

5- Register File (RegisterFile.v)



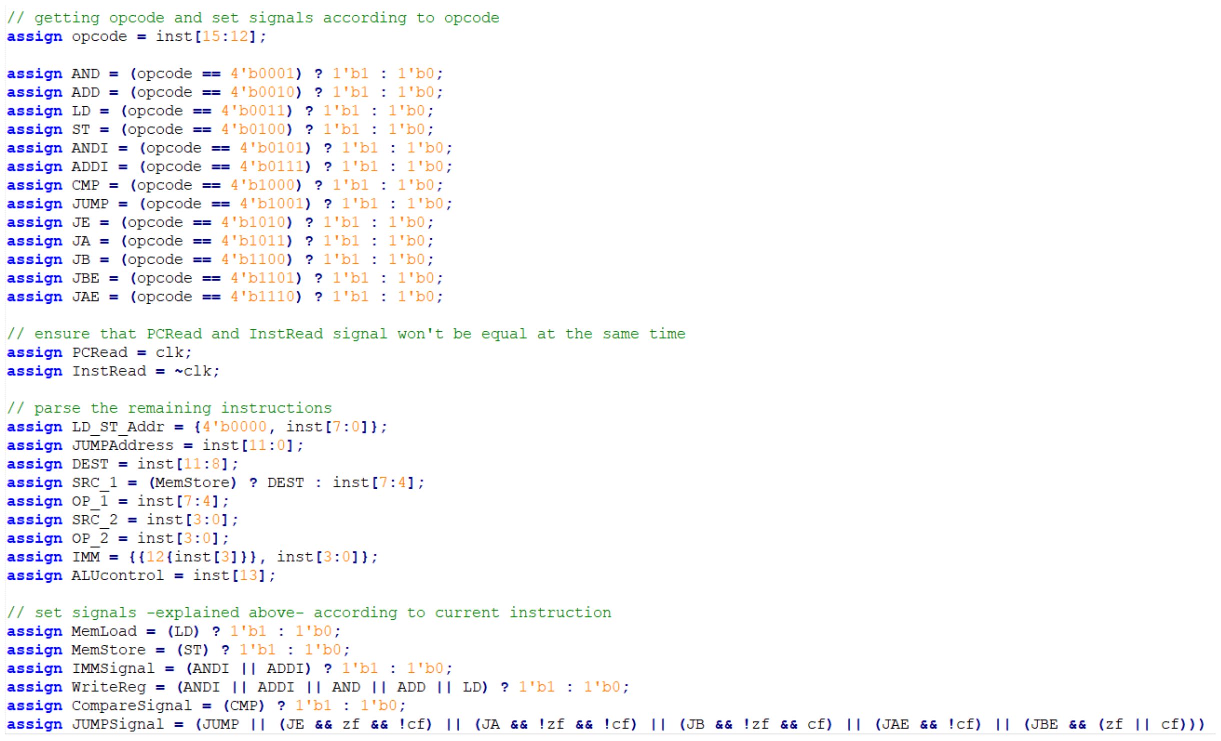
```verilog
module RegisterFile(
clk,
clear,
reg_write,
input_data,
write_reg_no,
read_reg_1,
read_reg_2,
reg_1_data,
reg_2_data
);

input clk;                    // clock
input clear;                  // clear pin for registers
input reg_write;              // enable pin for registers
input [15:0] input_data;      // if reg_write is 1, this data will be stored in the register write_reg_no
input [3:0] write_reg_no;     // destination register
input [3:0] read_reg_1;       // register number to output value stored in it (SRC1 or OP1)
input [3:0] read_reg_2;       // register number to output value stored in it (SRC2 or OP2)
output [15:0] reg_1_data;     // data stored inside read_reg_1
output [15:0] reg_2_data;     // data stored inside read_reg_2

integer i;
// 16 registers having 16 bits data size
reg [15:0] registers [0:15];

// initialize registers with 0
initial begin
    for(i=0; i<16; i=i+1)
    begin
    registers[i] = 16'h0;
    end
end
// registers are triggered on rising edge
always @ (posedge clk)
begin
    // if we are allowed to store data inside a register...
    if(reg_write) begin
        registers[write_reg_no] = input_data;
    end
end
// if registers need to be cleared
always @ (clear)
begin
    // reset all registers
    if(clear) begin
        for(i=0; i<16; i=i+1)
        begin
        registers[i] = 16'h0;
        end
    end
end

assign reg_1_data = registers[read_reg_1];
assign reg_2_data = registers[read_reg_2];
```

In this module, we have 16 registers which have 16 bits data sizes. This module has input ports for reading values inside two registers; destination register for selecting which register to store input data; input data to store in a register; enable pin; clear pin and clock pin. Register file outputs 2 values stored inside selected 2 registers.

6- Control Unit (ControlUnit.v)

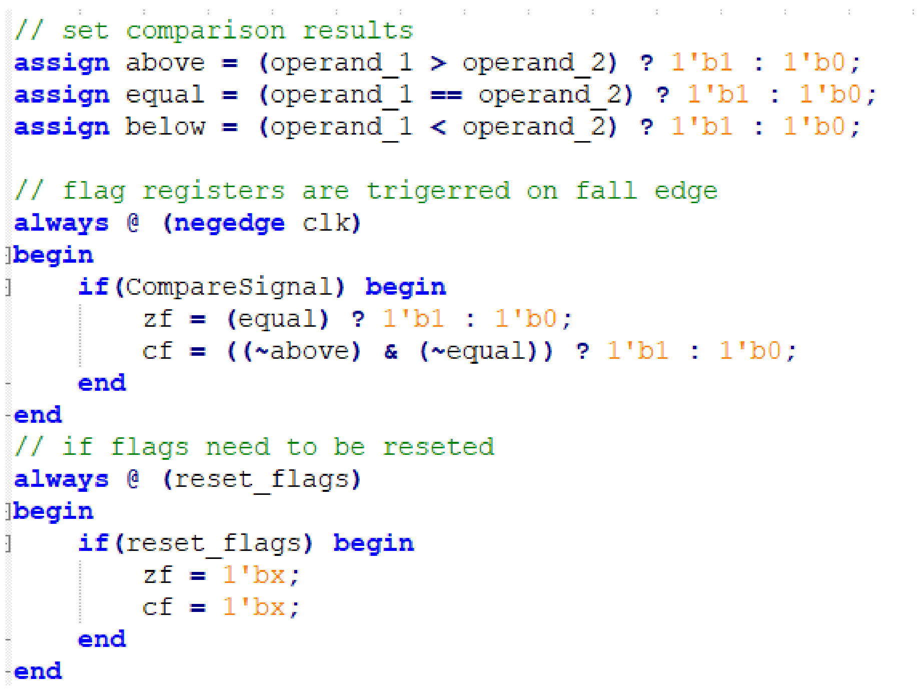
In this component, what we are basically doing is parsing incoming instruction and setting signals according to this instruction.

Here is parsing and setting signals part:



```
// getting opcode and set signals according to opcode
assign opcode = inst[15:12];

assign AND = (opcode == 4'b0001) ? 1'b1 : 1'b0;
assign ADD = (opcode == 4'b0010) ? 1'b1 : 1'b0;
assign LD = (opcode == 4'b0011) ? 1'b1 : 1'b0;
assign ST = (opcode == 4'b0100) ? 1'b1 : 1'b0;
assign ANDI = (opcode == 4'b0101) ? 1'b1 : 1'b0;
assign ADDI = (opcode == 4'b0111) ? 1'b1 : 1'b0;
assign CMP = (opcode == 4'b1000) ? 1'b1 : 1'b0;
assign JUMP = (opcode == 4'b1001) ? 1'b1 : 1'b0;
assign JE = (opcode == 4'b1010) ? 1'b1 : 1'b0;
assign JA = (opcode == 4'b1011) ? 1'b1 : 1'b0;
assign JB = (opcode == 4'b1100) ? 1'b1 : 1'b0;
assign JBE = (opcode == 4'b1101) ? 1'b1 : 1'b0;
assign JAE = (opcode == 4'b1110) ? 1'b1 : 1'b0;

// ensure that PCRead and InstRead signal won't be equal at the same time
assign PCRead = clk;
assign InstRead = ~clk;

// parse the remaining instructions
assign LD_ST_Addr = {4'b0000, inst[7:0]};
assign JUMPAddress = inst[11:0];
assign DEST = inst[11:8];
assign SRC_1 = (MemStore) ? DEST : inst[7:4];
assign OP_1 = inst[7:4];
assign SRC_2 = inst[3:0];
assign OP_2 = inst[3:0];
assign IMM = {{12{inst[3]}}, inst[3:0]};
assign ALUcontrol = inst[13];

// set signals -explained above- according to current instruction
assign MemLoad = (LD) ? 1'b1 : 1'b0;
assign MemStore = (ST) ? 1'b1 : 1'b0;
assign IMMSignal = (ANDI || ADDI) ? 1'b1 : 1'b0;
assign WriteReg = (ANDI || ADDI || AND || ADD || LD) ? 1'b1 : 1'b0;
assign CompareSignal = (CMP) ? 1'b1 : 1'b0;
assign JUMPSignal = (JUMP || (JE && zf && !cf) || (JA && !zf && !cf) || (JB && !zf && cf) || (JAE && !cf) || (JBE && (zf || cf)))
```

Control unit also controls the flags. Flags will be set after compare operation:

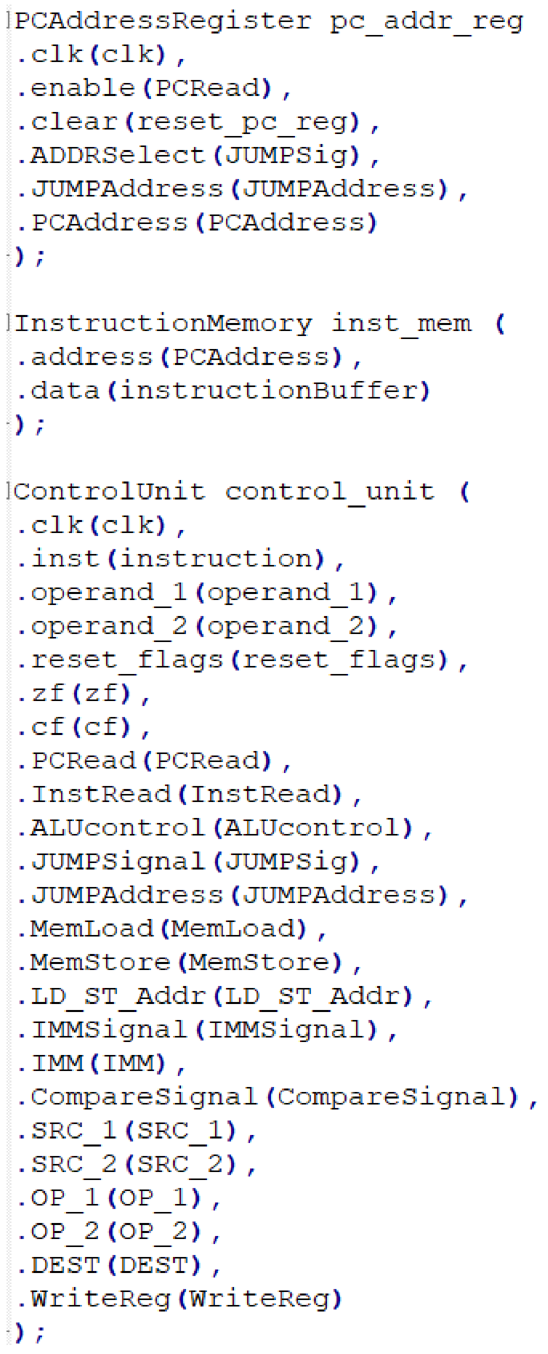


```
// set comparison results
assign above = (operand_1 > operand_2) ? 1'b1 : 1'b0;
assign equal = (operand_1 == operand_2) ? 1'b1 : 1'b0;
assign below = (operand_1 < operand_2) ? 1'b1 : 1'b0;

// flag registers are trigerred on fall edge
always @ (negedge clk)
begin
    if(CompareSignal) begin
        zf = (equal) ? 1'b1 : 1'b0;
        cf = ((~above) & (~equal)) ? 1'b1 : 1'b0;
    end
end
// if flags need to be reseted
always @ (reset_flags)
begin
    if(reset_flags) begin
        zf = 1'bx;
        cf = 1'bx;
    end
end
```

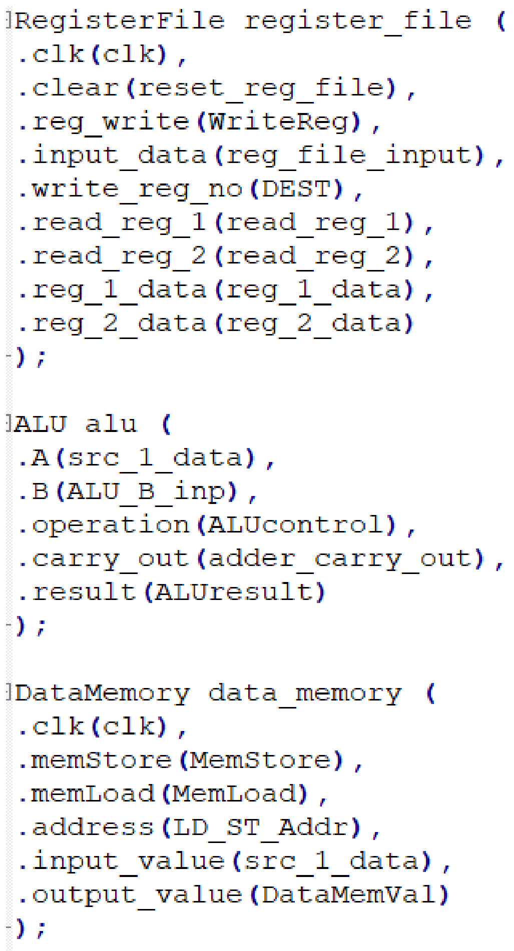
Flags are stored inside two registers and these registers are triggered on falling edge.

7- CPU (CPU.v)

In this module, we gathered all modules that we have designed and connected them to each other correctly, like designed in Logisim.



```
PCAddressRegister pc_addr_reg (
.clk(clk),
.enable(PCRead),
.clear(reset_pc_reg),
.ADDRSelect(JUMPSig),
.JUMPAddress(JUMPAddress),
.PCAddress(PCAddress)
);

InstructionMemory inst_mem (
.address(PCAddress),
.data(instructionBuffer)
);

ControlUnit control_unit (
.clk(clk),
.inst(instruction),
.operand_1(operand_1),
.operand_2(operand_2),
.reset_flags(reset_flags),
.zf(zf),
.cf(cf),
.PCRead(PCRead),
.InstRead(InstRead),
.ALUcontrol(ALUcontrol),
.JUMPSignal(JUMPSig),
.JUMPAddress(JUMPAddress),
.MemLoad(MemLoad),
.MemStore(MemStore),
.LD_ST_Addr(LD_ST_Addr),
.IMMSignal(IMMSignal),
.IMM(IMM),
.CompareSignal(CompareSignal),
.SRC_1(SRC_1),
.SRC_2(SRC_2),
.OP_1(OP_1),
.OP_2(OP_2),
.DEST(DEST),
.WriteReg(WriteReg)
);
```



```
RegisterFile register_file (
.clk(clk),
.clear(reset_reg_file),
.reg_write(WriteReg),
.input_data(reg_file_input),
.write_reg_no(DEST),
.read_reg_1(read_reg_1),
.read_reg_2(read_reg_2),
.reg_1_data(reg_1_data),
.reg_2_data(reg_2_data)
);

ALU alu (
.A(src_1_data),
.B(ALU_B_inp),
.operation(ALUcontrol),
.carry_out(adder_carry_out),
.result(ALUresult)
);

DataMemory data_memory (
.clk(clk),
.memStore(MemStore),
.memLoad(MemLoad),
.address(LD_ST_Addr),
.input_value(src_1_data),
.output_value(DataMemVal)
);
```

Clock (clk) is shared for all components and it's an input for CPU module.

For example for instruction "ADDI R2,R1,-1" , first hex code of this instruction is fetched from instruction memory, passed to the control unit. Control unit parsed the hexadecimal instruction code, says that: "This is an operation that requires ALU usage and addition operation will be done in ALU. First input for ALU will be data inside R1, second input for ALU won't be a data inside a register, it will be -1 immediate value as 16 bit sign extended. After calculating the result, store it in the R2 register." by setting signals WriteReg=1, ALUcontrol=1, IMMSignal=1.

Another example is "CMP OP1,OP2" after that "JBE -2". After converting these instructions to hex code, control unit will parse instruction and say that:

"Compare values inside OP1 register and OP2 register. If first operand is larger than second operand, set ZF=0 and CF=0. If first operand is equal to second operand, set ZF=1, CF=0. If first operand is less than second operand, set ZF=0, CF=1." by setting signal CompareSignal=1.

"Check flags. For JBE, ZF or CF need to be 1. If flag conditions are satisfied, PC needs to be decreased by 3." by setting JUMPSignal=1 and JumpAddress=-3.

Test Program

```
LD R0,0          // Load value inside 0th address of RAM to R0
LD R1,1          // Load value inside 1st address of RAM to R1
LD R2,2          // Load value inside 2nd address of RAM to R2
LD R3,3          // Load value inside 3rd address of RAM to R3
JUMP 3           // PC <- PC + 3 - go to compare
ADD R2,R2,R3     // R2 <- R2 + R3
ADDI R3,R3,1     // R3 <- R3 + 1
CMP R2,R0        // Compare values inside R2 and R0
JB -3            // If R2 is less than R0, PC <- PC - 3
ST R2,4          // Store data inside R2 to the 4th address of RAM
ADDI R2,R2,-3    // R2 <- R2 - 3
CMP R2,R1        // Compare values inside R2 and R1
JAE -2           // If R2 is above or equal to R1, PC <- PC - 2
ST R2,5          // Store data inside R2 to the 5th address of RAM
ANDI R4,R2,1     // R4 <- R2 & 1
ST R4,6          // Store data inside R4 to the 6th address of RAM
```

In this program, we are basically incrementing a number until reaching an upper limit. We also increment the increment amount of the number. After that we decrement that number by an immediate value until reaching a lower limit. At last, we get the number's last bit and store it inside RAM.
Address 0: Upper limit, stored in R0
Address 1: Lower limit, stored in R1
Address 2: Initial value of the number
Address 3: Increment amount
Data memory instance at the beginning:

Address	Data		
0	32	10	1
3	5	x	x
6	x	x	x
9	x	x	x

* These values are represented as hexadecimal.

In this case, our number is 1, our initial increment amount is 5, upper limit is 32 (50_{10}) and lower limit is 10 (16_{10}).

24/12/2018

While incrementing, corresponding registers should have these values for each iteration:

Registers	Values							
R2	1	6	12	19	27	36	46	57
R3	5	6	7	8	9	10	11	12

* Decimal representation

After increment operation, we are storing the last value of R2(57) in the 4th address of the RAM.
After that, we decremented this number by 3 until it's being less than 10 (16_{10}).

Registers	Values							
R2	57	54	51	48	45	…	18	15

* Decimal representation

After decrement operation, we store last value of R2(15) in the 5th address of the RAM. Then, we made R2 & 1 in order to get last bit of the data inside R2 and store it inside R4 and 6th address of RAM. In this case, last bit is 1.

And last data memory instance should be like this:

Address	Data		
0	32	10	1
3	5	39	f
6	1	x	x
9	x	x	x

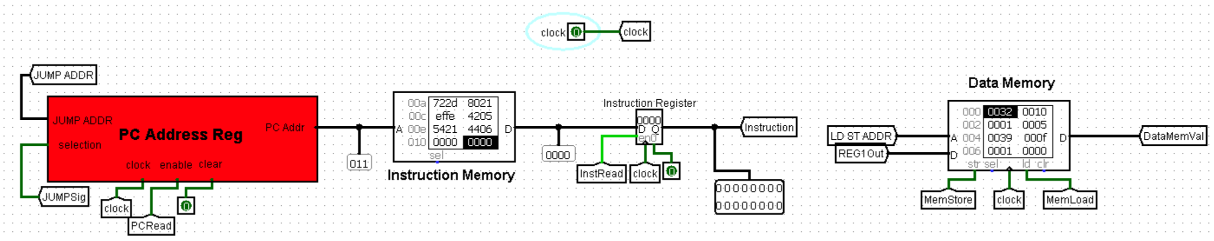
* Hexadecimal representation.

And register values should be like this:

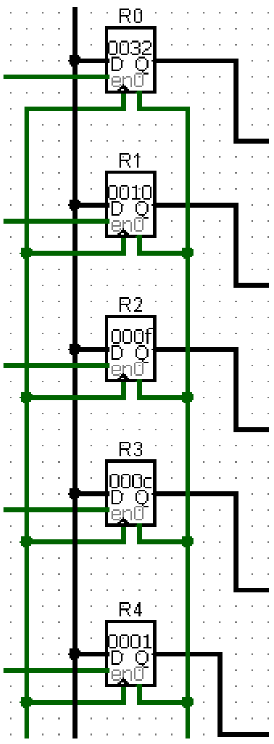
Registers	Values
R0	32
R1	10
R2	f
R3	c
R4	1

* Hexadecimal representation.

After testing this program with Logisim, we get the same results:

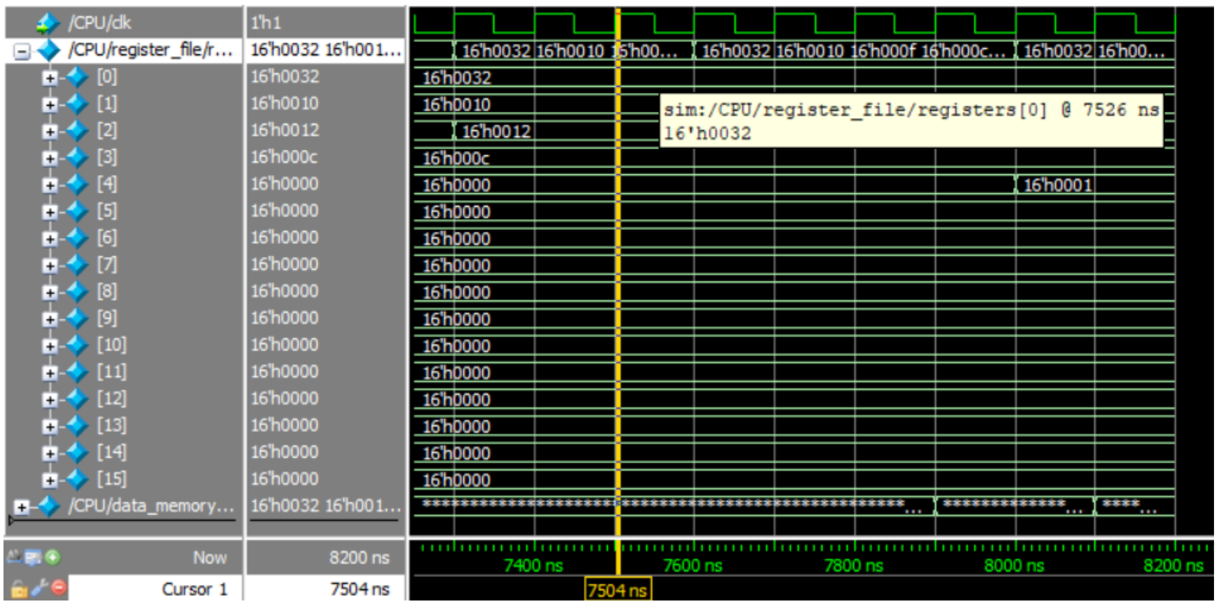


* Instance of Data Memory



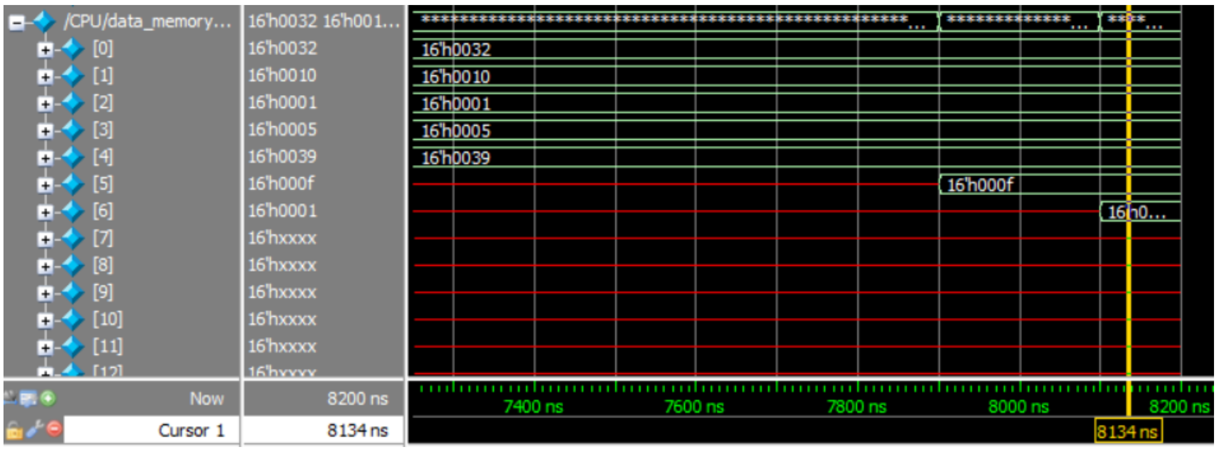
* Inside of register file

Then we tested our Verilog design. Here are the data memory and register file snapshots at the end of our test program:



```
/CPU/clk                1'h1
/CPU/register_file/r... 16'h0032 16'h001...   16'h0032 16'h0010 16'h00... 16'h0032 16'h0010 16'h000f 16'h000c... 16'h0032 16'h00...
  [0]                   16'h0032              16'h0032
  [1]                   16'h0010              16'h0010
  [2]                   16'h0012              16'h0012
  [3]                   16'h000c              16'h000c
  [4]                   16'h0000              16'h0000                                      16'h0001
  [5]                   16'h0000              16'h0000
  [6]                   16'h0000              16'h0000
  [7]                   16'h0000              16'h0000
  [8]                   16'h0000              16'h0000
  [9]                   16'h0000              16'h0000
  [10]                  16'h0000              16'h0000
  [11]                  16'h0000              16'h0000
  [12]                  16'h0000              16'h0000
  [13]                  16'h0000              16'h0000
  [14]                  16'h0000              16'h0000
  [15]                  16'h0000              16'h0000
/CPU/data_memory...     16'h0032 16'h001...   ********************************************...  **************...  ****...

sim:/CPU/register_file/registers[0] @ 7526 ns
16'h0032

Now       8200 ns      7400 ns   7600 ns   7800 ns   8000 ns   8200 ns
Cursor 1  7504 ns      7504 ns
```

* Register file



```
/CPU/data_memory...   16'h0032 16'h001...
  [0]                 16'h0032
  [1]                 16'h0010
  [2]                 16'h0001
  [3]                 16'h0005
  [4]                 16'h0039
  [5]                 16'h000f
  [6]                 16'h0001
  [7]                 16'hxxxx
  [8]                 16'hxxxx
  [9]                 16'hxxxx
  [10]                16'hxxxx
  [11]                16'hxxxx
  [12]                16'hxxxx
Now                   8200 ns
Cursor 1              8134 ns
```

* Data memory

Problems That We've Faced

- ### Synchronization Problem

  While designing Logisim and Verilog parts, we had problems about synchronizing reading PC, updating PC, reading instruction that PC shows (PCRead and InstRead signals). As we know, this can only caused by using clock and enable pins of registers. First we ensure that PCRead and InstRead will never be 1 at the same time. By this way, sequential execution is done. Sequence is "Read PC – Execute Instruction – Update PC – Read PC – Execute Instruction – Update PC ...".

  After handling this, when instruction register and PC register have the same trigger edges, we had some problems like jumping one instruction per 2 instructions i.e. execute sequence mem[PC+1] -> mem[PC+3] -> mem[PC+5] -> .... We made PC register triggered on falling edge; instruction register triggered on rising edge.

  At last, we had problems about regWrite signal in these kind of cases:
  ```
  ADD R1,R1,R2
  JUMP -1
  ```
  In that case, our design does "ADD R1,R1,R2" part two times after JUMP instruction. This problem has one more reason, that is when DEST and SRC registers are the same. We solved that problem with reading PC and executing instruction in only one clock period.

- ### Storing Flag Values

  At first, we were updating flags without checking incoming instruction is CMP or not in Logisim. Because of that, we were having problems on jump instruction with conditions (JE, JA, JB, JAE, JBE). In order to solve this problem, we stored flags inside registers. These registers can only be updated if the instruction is CMP by connecting CompareSignal to the enable pins of registers.