

In this project, we were asked to design a CPU with 10 bit address and 20 bit data. While designing the CPU, we had an instruction set consisting of 16 registers and 17 instructions. These were our 17 instructions: SUB, ADD, AND, OR, XOR, ADDI, SUBI, ANDI, ORI, XORI, LD, ST, JUMP, PUSH, POP, BE, BNE

- *ADD* instruction uses 3 registers: dest, src1 and src2. It collects the data in src1 and the data in src2 and writes it to the destination register.  $\text{Dest} = \text{Src1} + \text{Src2}$
- *SUB* instruction uses 3 registers. It extracts the data in src2 from the data in the src1 register and writes it to the destination register.  $\text{Dest} = \text{Src1} - \text{Src2}$
- *AND* the instruction uses 3 registers. It ands the value in Src2 with the data in Src1 and writes it to the destination register.  $\text{Dest} = \text{Src1} \& \text{Src2}$
- *OR* instruction uses 3 registers. Ors the data in Src1 and the data in Src2 writes it to the destination.  $\text{Dest} = \text{Src1} \mid \mid \text{Src2}$
- *XOR* instruction uses 3 registers. It puts the data in Src1 and the data in Src2 into the Xor process and writes the result to the destination register.  $\text{Dest} = \text{Src1} \wedge \text{Src2}$
- The *ADDI* instruction collects 7-bit immediate data with the src register data and writes it to the destination register.  $\text{Dest} = \text{Src} + \text{IMM}$
- The *SUBI* instruction extracts 7 bits of immediate data from the src register and writes it to the destination register.  $\text{Dest} = \text{Src} - \text{IMM}$
- The *ANDI* instruction creates a 7-bit immediate data with the src register data and writes it to the destination register.  $\text{Dest} = \text{Src} \& \text{IMM}$
- The *ORI* instruction makes a 7-bit immediate data with the src register data and writes it to the destination register.  $\text{Dest} = \text{Src} \mid \mid \text{IMM}$ .
- The *XORI* instruction XORs a 7-bit immediate data with the src register data and writes it to the destination register.  $\text{Dest} = \text{Src} \wedge \text{IMM}$ .
- In the *LD* instruction DEST is the register number, ADDR is the address of the data which will be fetched from data memory to the DEST register.

- In the *ST* instruction 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.
- *JUMP* instruction will set the Program Counter (PC will hold current instruction's address) to the given value in the instruction. JUMP is an instruction where ADDR will be in PC relative mode. ADDR will be offset and it can be negative.
- *PUSH* instruction will store the value in register SRC1 to the stack, then it will decrement the stack pointer register.
- *POP* instruction will retrieve the pointed value from the stack and will write into register SRC1, then it will increment the stack pointer register.
- *BE* and *BNE* instruction will set PC value to given address according to values of two registers. BE will set PC to given address in instruction if two register holds the same value. BNE will set PC to given address in instruction if two register holds different value.
- 

## Part 1 - Assembly Language

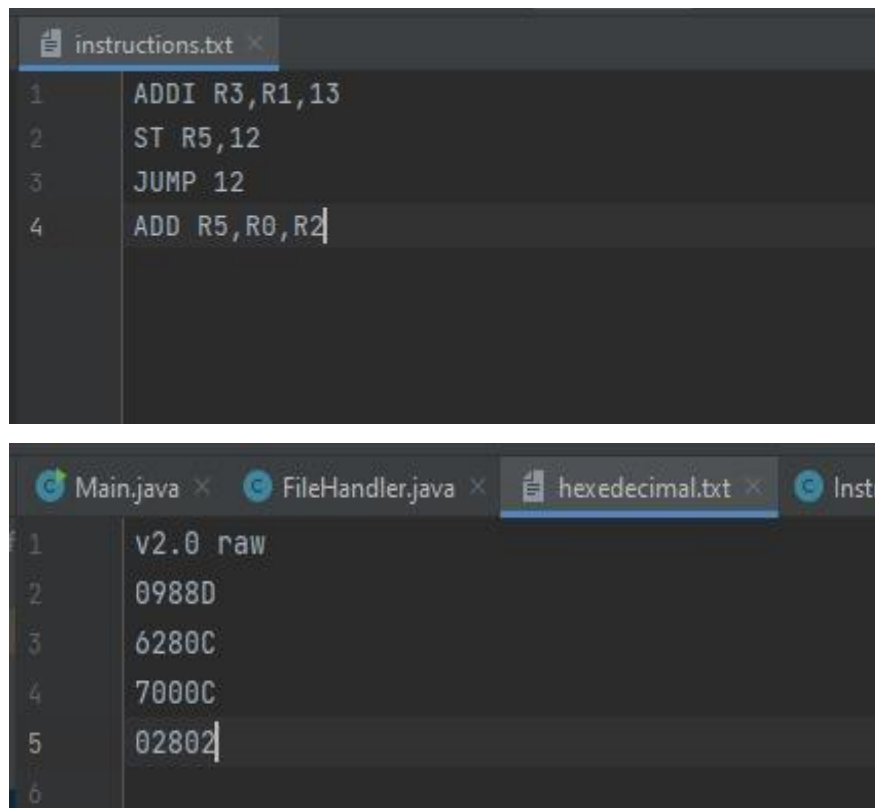
After analyzing all the instructions, we need to adjust their set architecture to suit our demands. We use the first five bits of a total of 20 data bits to indicate which instruction it is. We determine the next bits according to the structure of each instruction.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
ADD	0	0	0	0	0	0	Dest Register			SRC1				0	0	0	SRC2			
SUB	0	0	0	1	0	Dest Register			SRC1				0	0	0	SRC2				
AND	0	0	1	0	0	Dest Register			SRC1				0	0	0	SRC2				
OR	0	0	1	1	0	Dest Register			SRC1				0	0	0	SRC2				
XOR	0	1	0	0	0	Dest Register			SRC1				0	0	0	SRC2				
ADDI	0	0	0	0	1	Dest Register			SRC				IMM7							
SUBI	0	0	0	1	1	Dest Register			SRC				IMM7							
ANDI	0	0	1	0	1	Dest Register			SRC				IMM7							
ORI	0	0	1	1	1	Dest Register			SRC				IMM7							
XORI	0	1	0	0	1	Dest Register			SRC				IMM7							
LD	0	1	0	1	0	Dest Register			PCOffset11											
ST	0	1	1	0	0	SRC			PCOffset11											
JUMP	0	1	1	1	0	Address														
PUSH	1	0	0	0	0	SRC			0	0	0	0	0	0	0	0	0	0	0	1
POP	1	0	0	1	0	SRC			0	0	0	0	0	0	0	0	0	0	0	0
BE	1	0	1	0	0	SRC1			SRC2				Address							
BNE	1	0	1	0	0	SRC1			SRC2				Address							

ISA – Instruction Set Architecture

When designing ISA, we used the first 5 bits to specify instructions. We checked whether the 5th bit is immediate based on the 0 or 1 state. The next 4 bits are used for the destination register [6:9], the other 4 bits for the src1 register [10:13], and the remaining bits are used for src2 or immediate value, depending on the instruction.

After determining which bits of which instructions correspond to what, we convert these instructions to hexadecimal numbers using the assembler. We used the java language for this goal.



The image consists of two screenshots of a code editor. The top screenshot shows a file named 'instructions.txt' with four lines of assembly code: '1 ADDI R3,R1,13', '2 ST R5,12', '3 JUMP 12', and '4 ADD R5,R0,R2'. The bottom screenshot shows a file named 'hexadecimal.txt' with five lines of hexadecimal output: '1 v2.0 raw', '2 0988D', '3 6280C', '4 7000C', and '5 02802'. The editor tabs at the top of the bottom screenshot include 'Main.java', 'FileHandler.java', 'hexadecimal.txt', and 'Instr'.

```
1 ADDI R3,R1,13
2 ST R5,12
3 JUMP 12
4 ADD R5,R0,R2
```

```
1 v2.0 raw
2 0988D
3 6280C
4 7000C
5 02802
```

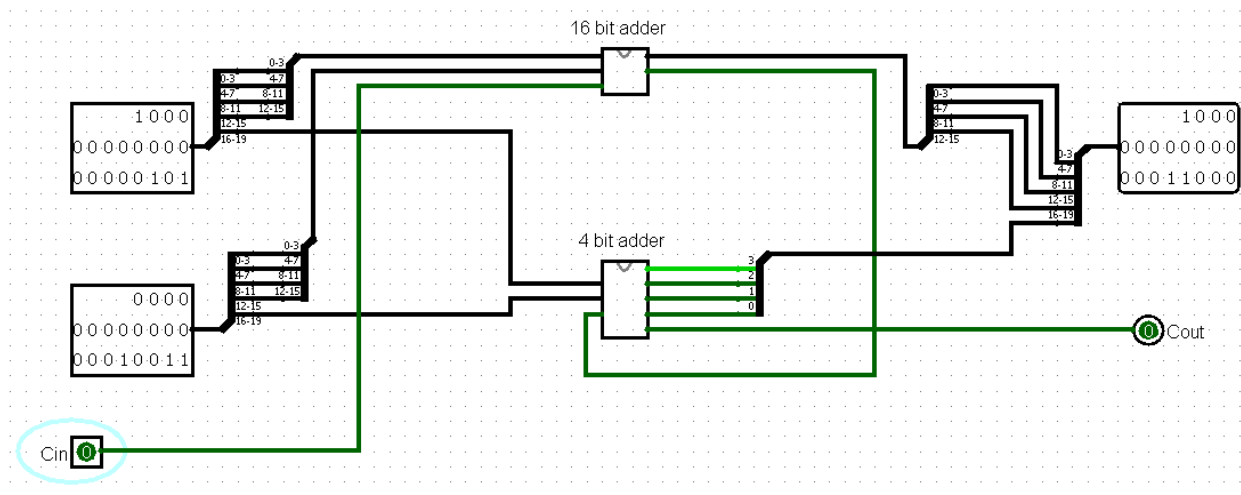
Here we see an example of an assembly. The inputs we enter as Instruction are output as hexadecimal numbers.

## Part 2 - Logisim Component Design

In this iteration, we were asked to create the necessary components to design the CPU. The components needed to build the CPU are seen below.

### 2.1 20-bit adder

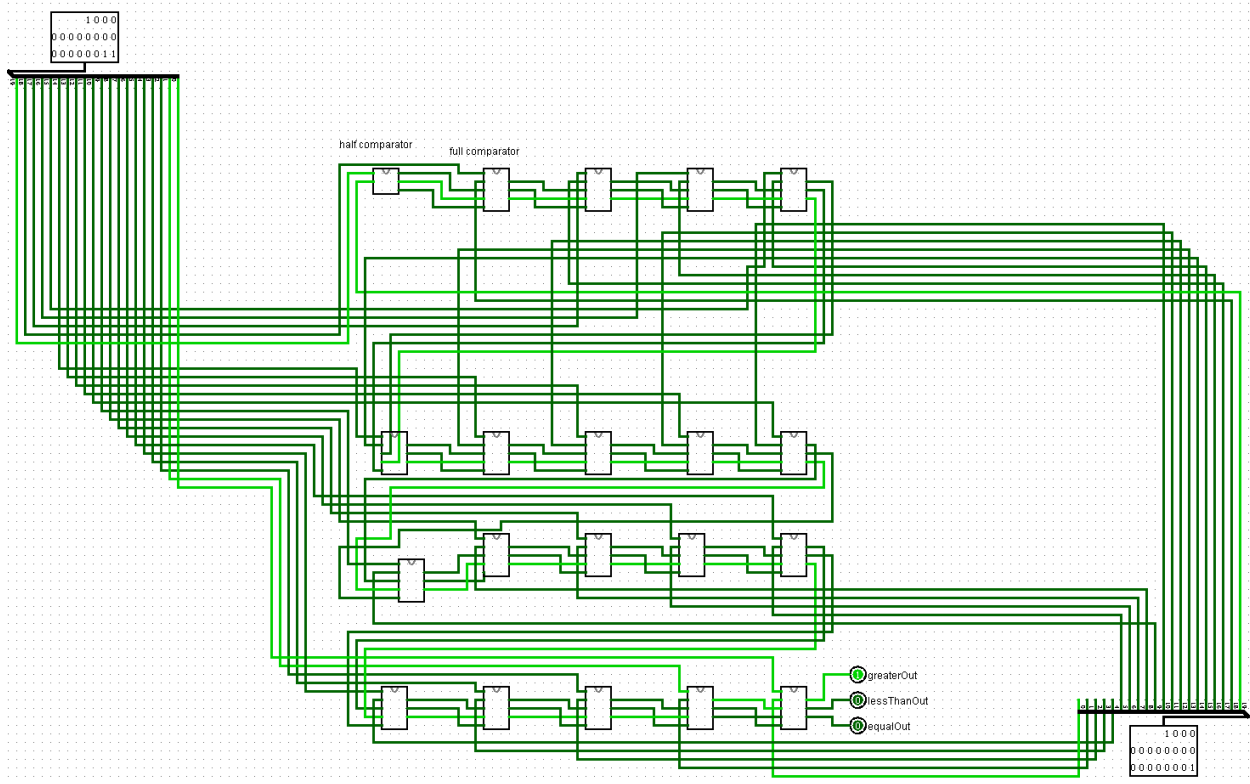
A 20-bit adder consists of a 16-bit adder and a 4-bit adder. The first 16 bits of each input add 16 bits of adder and the remaining 4 bits add up to 4 bits of adder. The 16-bit adder consists of 2 8-bit adders, and the 8-bit adder consists of 2 4-bit adders. A 4-bit adder consists of a combination of 4 full adders.



*20-bit adder*

### 2.2 20-bit comparator

The 20-bit comparator consists of a half comparator and full comparators. It compares each bit of the two inputs from largest to smallest and gives either greater equal or less than output depending on the comparison status. In the example below, we see a 20-bit comparator and an example.

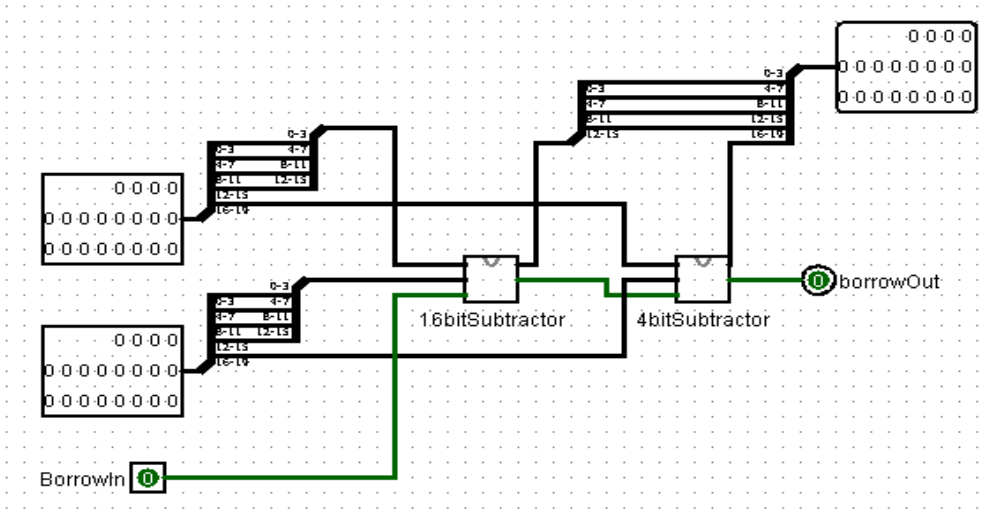


*20-bit comparator*

### 2.3 20-bit subtractor

The 20-bit subtractor consists of a 16-bit subtractor and a 4-bit subtractor. It subtracts the first 16 bits of both inputs and then the remaining 4 bits. The result is written to the 20-bit output.

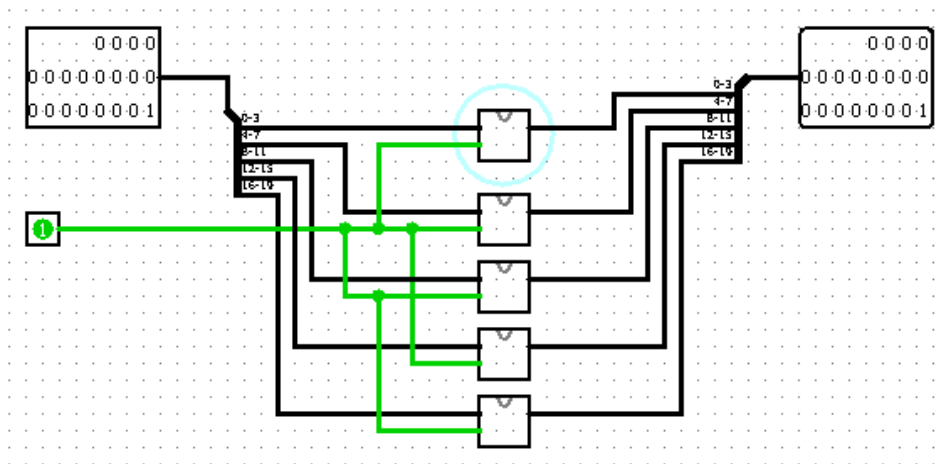
The 16-bit subtractor consists of two 8-bit subtractors, the 8-bit subtractor consists of two 4-bit subtractors and the 4-bit subtractor consists of 4 full subtractors.



20-bit subtractor

## 2.4 20-bit register

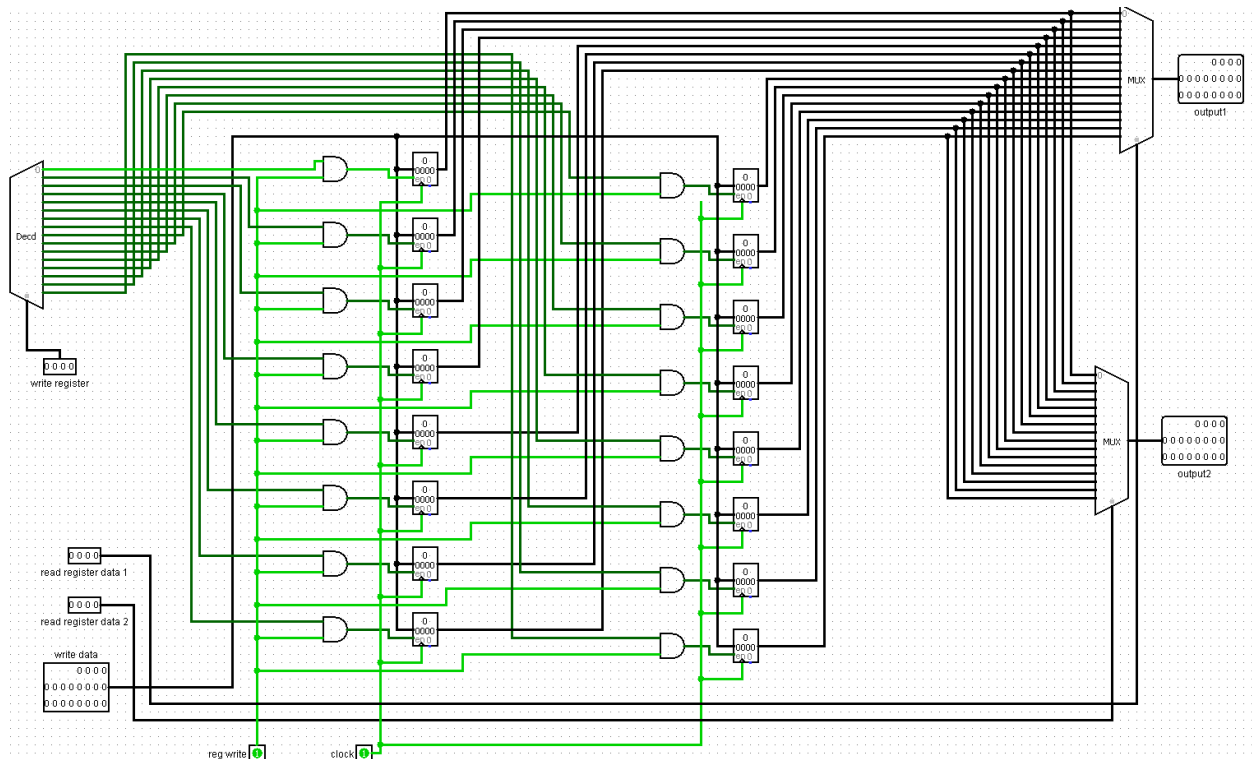
The 20-bit register consists of 5 4-bit registers. In the example below, we see 20bit register. These registers will then be used when creating the register file.



20-bit register

## 2.5 Register File

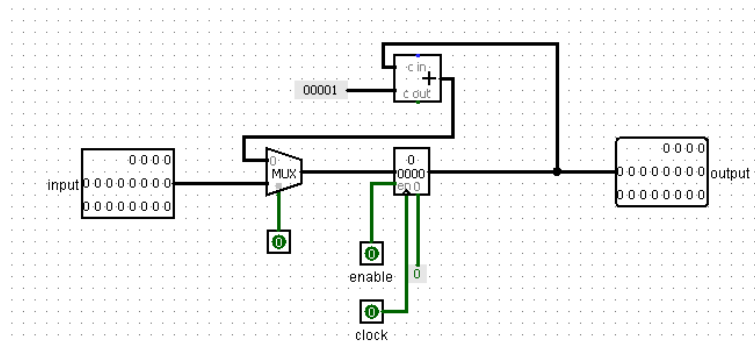
Register file is a structure that shows which register to write the input received from the user and reads the registers with data. The register file records whichever register the input is written to. Then we can access the data in that register by entering the register number into the read register.



*Register File*

## 2.6 Program Counter

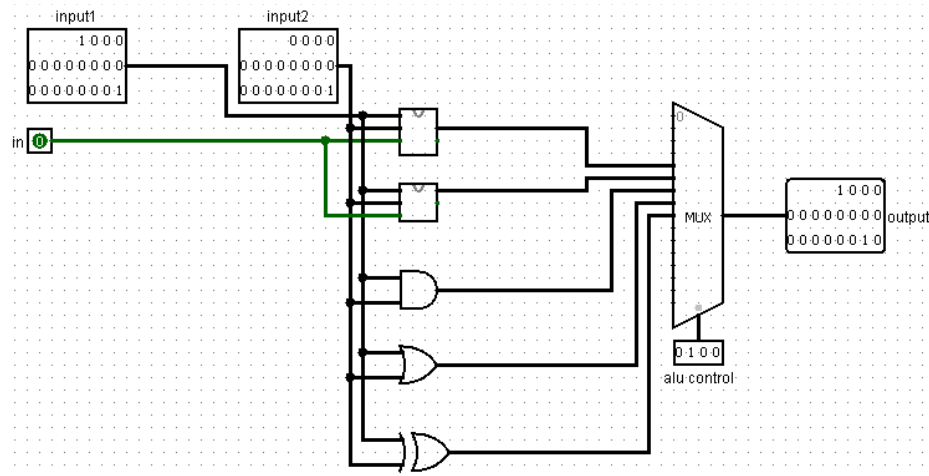
PC will hold current instruction's address. The program counter displays the data of the next instruction to run in the CPU. It makes these changes for bne, be and jump and these operations provide branching within the program.



*program counter*

## 2.7 Alu

ALU is able to perform a range of arithmetic and logical operations including ADD, SUB, AND, OR, XOR, ADDI, SUBI, ANDI, ORI, and XORI. The operands for these operations can be fetched from the register file or immediate values, and the result of the operation is typically stored back in the register file. The control unit produces the necessary control signals to coordinate the operation of the ALU and ensure that the correct operation is performed based on the current instruction.



## 2.8 Extenders

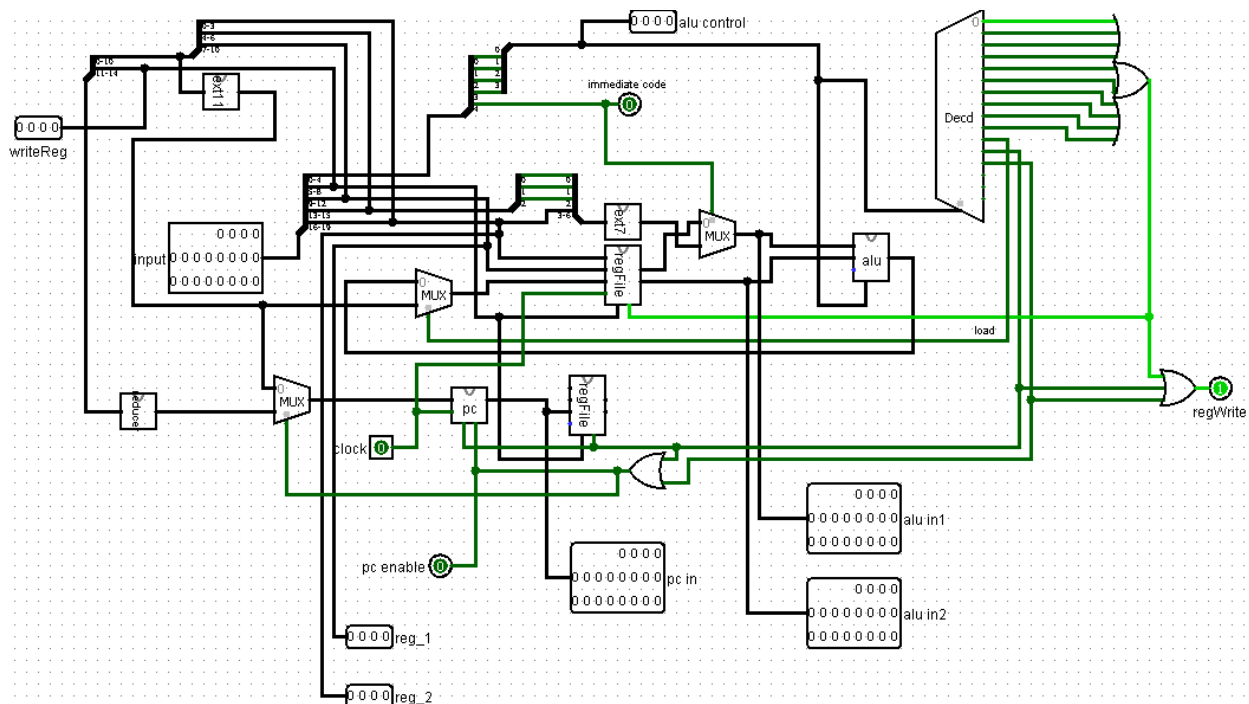
If the incoming bits are not in the size we want, we use bit extenders to bring them to the state we want. We have 3 different extenders for this process. 7to20, 11to20 and 15to20. We use the 7-to-20 extender to extend the input immediate data to 20 bits. We use the 11 to 20 extender to pull the incoming PCOffset bits to 20 bits. We use the 15 to 20 extender in the jump operation.



### Part 3 - Logisim Design with Control Unit

### 3.1 Control Unit

After designing all the components, it is necessary to design the control unit that decides where to send the signals of the incoming instruction code. We combine the components we have made so far in the control unit. The values sent to the arithmetic logic unit and register file come from here. For example, if there are instructions with immediate value such as ANDI or ORI, the immediate output is 1, otherwise 0. You see the control unit design we made below.

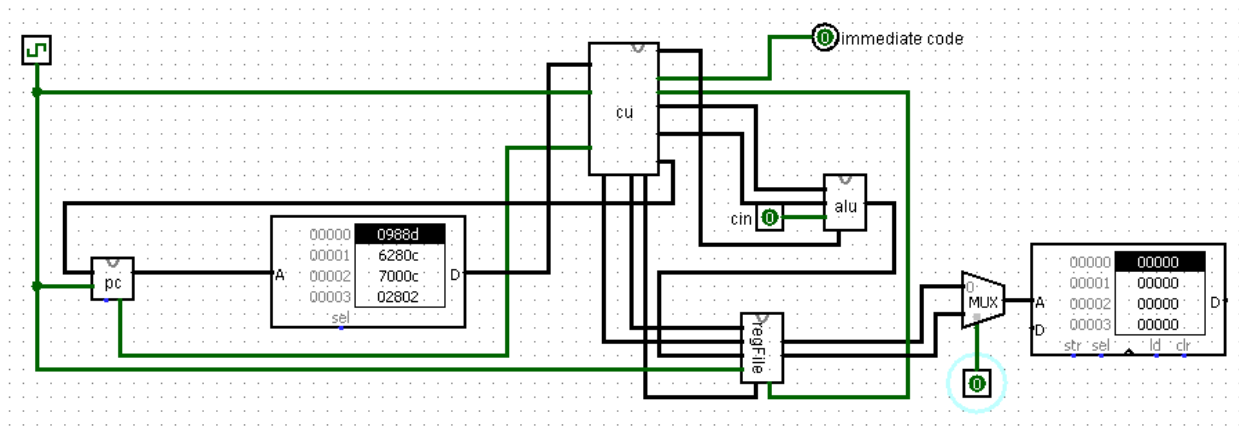


Control Unit

### 3.2 CPU

This CPU has a data bus capable of 20-bit wide data operations and has a 10-bit address bus. This means the ability to address data that can be placed up to 1 kilobyte. The CPU contains 16 registers and has 17 instruction sets. Instruction sets involve manipulating data on the bus and moving data between registers. For example, the "ADD" instruction set collects the two data on the bus and stores the result in a register, and the "JUMP" instruction set jumps the program's working directory to a specific address.

The control unit (CU) enables and manages the arithmetic-logic unit (ALU) and register file in a CPU. For example, when the CU executes an "ADD" instruction set, it sends signals to the ALU to collect the two data on the bus and to store the result in a particular register in the register file. Similarly, when the CU executes an "LD" instruction set, it sends the necessary signals to load the data on the bus into a register in the register file. In this way, the CU enables and manages the arithmetic-logic unit and register file as needed during program execution.



CPU

## Part 4 - Verilog Design

### 4.1 16-bit register

This code defines a Verilog module for a 16-bit register. In this module, the contents of the register are updated with the d\_in and wr entries. With the clk input, the time to update the contents of the register is determined. With the reset input, the contents of the register are reset. D\_out output shows the contents of the register.

```

1  module register (
2      input clk,
3      input reset,
4      input [15:0] din,
5      input wr,
6      output [15:0] dout
7  );
8      reg [15:0] dout;
9
10     always @(posedge clk) begin
11         if (reset)
12             dout <= 16'b0;
13         else if (wr)
14             dout <= din;
15     end
16 endmodule

```

## 4.2 20-bit adder

```
1 module adder (  
2     input [19:0] a,  
3     input [19:0] b,  
4     output [19:0] sum  
5 );  
6     assign sum = a + b;  
7 endmodule
```

This code defines a Verilog module that adds two 20-bit numbers and gives the sum to the "sum" output. With the a and b inputs, the numbers to be added are specified and the sum output shows the total.

## 4.3 20-bit subtractor

This code defines a Verilog module that calculates the difference between two 20-bit numbers and outputs the difference to the "diff" output. The a and b entries specify the numbers to subtract, and the diff output shows the difference.

```
1 module subtractor (  
2     input [19:0] a,  
3     input [19:0] b,  
4     output [19:0] diff  
5 );  
6     assign diff = a - b;  
7 endmodule
```

## 4.4 20\_bit program counter

This code defines a Verilog module for a 20-bit program counter. In this module, the program counter is updated with pc\_in and pc\_en entries. With the clk input, the update time of the program counter is determined. With the reset input, the program counter is reset. The pc\_out output shows the contents of the program counter.

```
1 module program_counter (  
2     input clk,  
3     input reset,  
4     input [19:0] pc_in,  
5     input pc_en,  
6     output [19:0] pc_out  
7 );  
8     reg [19:0] pc;  
9  
10    always @(posedge clk) begin  
11        if (reset)  
12            pc <= 20'b0;  
13        else if (pc_en)  
14            pc <= pc_in;  
15    end  
16  
17    assign pc_out = pc;  
18 endmodule
```

## 4.5 Control unit

The control unit uses a case statement to decode the opcode input and set the appropriate output signals. For example, when the opcode is 00000, the control unit sets the alu\_op1 signal to 1 and the reg\_write signal to 1 to indicate that the instruction is the ADD instruction and that the result should be written to the register file. When the opcode is 01010, the control unit sets the ld signal to 1, the mem\_read signal to 1, and the reg\_write signal to 1 to indicate that the instruction is the LD instruction and that the data memory should be read and the result should be written to the register file.

```
1  module control_unit(  
2      input [4:0] opcode,  
3      output reg mem_read,  
4      output reg mem_write,  
5      output reg reg_write,  
6      output reg alu_op1,  
7      output reg alu_op2,  
8      output reg alu_op3,  
9      output reg alu_op4,  
10     output reg alu_op5,  
11     output reg alu_op6,  
12     output reg alu_op7,  
13     output reg alu_op8,  
14     output reg alu_op9,  
15     output reg alu_op10,  
16     output reg jump,  
17     output reg ld,  
18     output reg st,  
19     output reg be,  
20     output reg bne,  
21     output reg push,  
22     output reg pop  
23 );  
  
24     always @(*) begin  
25         case(opcode)  
26             5'b00000: alu_op1 = 1'b1; reg_write = 1'b1; // ADD  
27             5'b00010: alu_op2 = 1'b1; reg_write = 1'b1; // SUB  
28             5'b00100: alu_op3 = 1'b1; reg_write = 1'b1; // AND  
29             5'b00110: alu_op4 = 1'b1; reg_write = 1'b1; // OR  
30             5'b01000: alu_op5 = 1'b1; reg_write = 1'b1; // XOR  
31             5'b00001: alu_op6 = 1'b1; reg_write = 1'b1; // ADDI  
32             5'b00011: alu_op7 = 1'b1; reg_write = 1'b1; // SUBI  
33             5'b00101: alu_op8 = 1'b1; reg_write = 1'b1; // ANDI  
34             5'b00111: alu_op9 = 1'b1; reg_write = 1'b1; // ORI  
35             5'b01001: alu_op10 = 1'b1; reg_write = 1'b1; // XORI  
36             5'b01110: jump = 1'b1; // JUMP  
37             5'b01010: ld = 1'b1; mem_read = 1'b1; reg_write = 1'b1; // LD  
38             5'b01100: st = 1'b1; mem_write = 1'b1; // ST  
39             5'b10100: be = 1'b1; // BE  
40             5'b10101: bne = 1'b1; // BNE  
41             5'b10000: push = 1'b1; // PUSH  
42             5'b10010: pop = 1'b1; // POP  
43         endcase  
44     end  
45 endmodule
```

## 4.6 Register file for 16 registers

This code defines a register file module containing 16 20-bit registers. This module takes inputs of clock (clk), enable, write\_data and write\_register indicating which register to write to. It also takes inputs select\_read\_1 and select\_read\_2, indicating from which registers the outputs of read\_data\_1 and read\_data\_2 will be read. The registers array represents 16 20-bit registers. The outputs of read\_data\_1 and read\_data\_2 are always read from the selected registers. On the rising edge of the clk input, if the enable input is 1, the write\_data value is written to the selected register write\_register.

```
1 module register_file (  
2     input clk,  
3     input enable,  
4     input [19:0] write_data,  
5     input [3:0] write_register,  
6     input [3:0] select_read_1,  
7     input [3:0] select_read_2,  
8     output [19:0] read_data_1,  
9     output [19:0] read_data_2  
10 );  
11 // Declare registers  
12 reg [19:0] registers [15:0];  
13 // Read data from selected registers  
14 always @(*) begin  
15     read_data_1 = registers[select_read_1];  
16     read_data_2 = registers[select_read_2];  
17 end  
18 // Write to register on rising edge of clock  
19 always @(posedge clk) begin  
20     if (enable) begin  
21         registers[write_register] <= write_data;  
22     end  
23 end  
24 endmodule
```

## 4.7 Alu

The ALU is a module that performs arithmetic and logical operations on register and immediate values and updates the register file with the results of these operations.

It has input signals that control its operation and output signals that hold the results of its operations. It has several always blocks that implement different tasks, such as the add and subtract operations and the selection of source and destination registers.

```
1 module alu(  
2     input clk,  
3     input enable,  
4     input [4:0] opcode,  
5     input [3:0] src1_sel,  
6     input [3:0] src2_sel,  
7     input [3:0] dst_sel,  
8     output reg [19:0] dst,  
9     input [19:0] src1,  
10    input [19:0] src2,  
11    input [19:0] src1_imm,  
12    input [19:0] src2_imm,  
13    output reg [19:0] adder_out,  
14    output reg [19:0] subtractor_out,  
15    output reg [19:0] pc_out  
16 );  
17  
18 // Register file to hold register values  
19 //and signal to write into any register  
20 reg [19:0] register_file[15:0];  
21  
22 // Adder to perform add operation  
23 always@(posedge clk) begin  
24     if (enable) begin  
25         adder_out <= src1 + src2;  
26     end  
27 end  
28  
29 // Subtractor to perform sub operation  
30 always@(posedge clk) begin  
31     if (enable) begin  
32         subtractor_out <= src1 - src2;  
33     end  
34 end  
35  
36 // PC output  
37 always@(posedge clk) begin  
38     if (enable) begin  
39         pc_out <= pc_out + src2;  
40     end  
41 end  
42  
43 // Select read registers and assign src1 and src2 values  
44 always@(posedge clk) begin  
45     if (enable) begin  
46         src1 <= register_file[src1_sel];  
47         src2 <= register_file[src2_sel];  
48     end  
49 end  
50  
51 // Select destination register and assign dst value  
52 always@(posedge clk) begin  
53     if (enable) begin  
54         case (opcode)  
55             5'b00000: register_file[dst_sel] <= adder_out;  
56             5'b00010: register_file[dst_sel] <= subtractor_out;  
57             // Other operations  
58             default: register_file[dst_sel] <= 0;  
59         endcase  
60     end  
61 end  
62  
63 // Assign immediate values to src1 and src2  
64 always@(posedge clk) begin  
65     if (enable) begin  
66         if (opcode[4] == 1) begin  
67             src1 <= src1_imm;  
68         end  
69         if (opcode[3] == 1) begin  
70             src2 <= src2_imm;  
71         end  
72     end  
73 end  
74  
75 endmodule
```