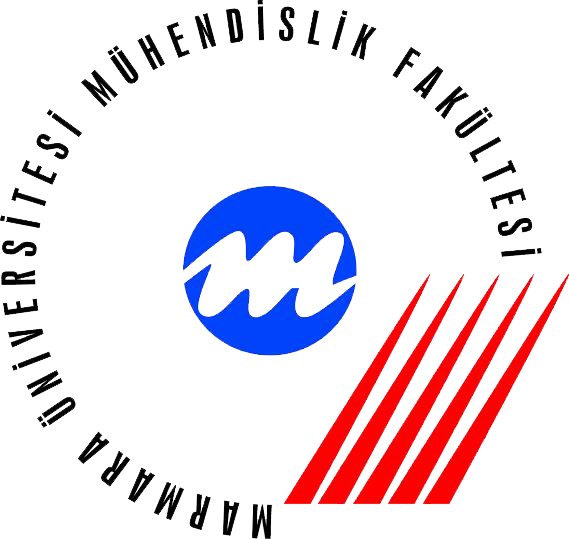
A blue and white logo

Description automatically generated

**MARMARA UNIVERSITY**

**FACULTY OF ENGINEERING**

CSE3215 – Digital Logic Design – Project 1

CPU Design

150120077 – Beyza Nur Kaya

150120047 – Sena Ektiricioğlu

150120061 – Selin Aydın

150120509 – Eren Duyuk

1. Introduction

In this project, we were responsible for creating a CPU design from scratch. To do that, we first designed our instruction set architecture. Then, we figured out the components we needed to do its functional operations. After this, we needed to connect them together so that it would be an end-to-end CPU with its states and signals. Then, we moved our project to Verilog in ModelSim. After all these steps, we simulated our results as well.

1. ISA Design

In this step, we were given instructions as 18 bit and we needed to design the opcode bits and register bits to improve the immediate value range.

For this step, we had a total of 18 bits for storing the instructions given to us. We had 4 bits for representing the registers. For instructions, we selected 4 bits to represent them. For that reason, we had 6 bits for representing the immediate values and as given to us 10 bits for representing the address.

We tried to make it more efficient for immediate values but since we had again 6 bits for immediate values. Not to make things complex, we stick to our basic plan that we mentioned above.

Below you can see our table for ISA.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| INSTRUCTION | OPCODE | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| ADD | 0 | 0 | 0 | 0 | **DST** | | | | **SRC1** | | | | **SRC2** | | | | **X** | **X** |
| ADDI | 0 | 0 | 0 | 1 | **DST** | | | | **SRC1** | | | | **IMM** | | | | | |
| NOR | 0 | 0 | 1 | 0 | **DST** | | | | **SRC1** | | | | **SRC2** | | | | **X** | **X** |
| NAND | 0 | 1 | 1 | 0 | **DST** | | | | **SRC1** | | | | **SRC2** | | | | **X** | **X** |
| AND | 0 | 1 | 0 | 0 | **DST** | | | | **SRC1** | | | | **SRC2** | | | | **X** | **X** |
| ANDI | 0 | 1 | 0 | 1 | **DST** | | | | **SRC1** | | | | **IMM** | | | | | |
| LD | 1 | 0 | 0 | 0 | **DST** | | | | **ADDR** | | | | | | | | | |
| ST | 1 | 0 | 0 | 1 | **SRC** | | | | **ADDR** | | | | | | | | | |
| CMP | 0 | 0 | 1 | 1 | **OP1** | | | | **OP2** | | | | **X** | **X** | **X** | **X** | **X** | **X** |
| JUMP | 1 | 1 | 1 | 1 | **ADDR** | | | | | | | | | | **X** | **X** | **X** | **X** |
| JE | 1 | 0 | 1 | 0 | **ADDR** | | | | | | | | | | **X** | **X** | **X** | **X** |
| JA | 1 | 0 | 1 | 1 | **ADDR** | | | | | | | | | | **X** | **X** | **X** | **X** |
| JB | 1 | 1 | 0 | 0 | **ADDR** | | | | | | | | | | **X** | **X** | **X** | **X** |
| JAE | 1 | 1 | 0 | 1 | **ADDR** | | | | | | | | | | **X** | **X** | **X** | **X** |
| JBE | 1 | 1 | 1 | 0 | **ADDR** | | | | | | | | | | **X** | **X** | **X** | **X** |
| NAN | 0 | 1 | 1 | 1 | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** |

For some instructions with immediate values, we used the last bits as 0 and 1 to determine it so that in ALU we can determine if we should use Source 2 or an immediate value.

1. Component Design

For our component design, we first listed some of the components we needed for this CPU and ISA to work. We started with ALU and its operations. Since we cannot use the default adder, we first created a half adder and an adder. Then, we created for some size of bit to use later. For example, we created 10-bit adder for address adding operations and 18-bit adders for ALU.

Also, we designed a comparator which is using the adder and a negator that we created to subtract operand 1 from operand 2. According to the result of the subtraction, it sets the zero flag and carry flag. In negator, we first take the complement of the bits and then add 1 to it which is 2’s complement.

In the ALU, we have 2 operands and immediate value as inputs and according to is\_immediate value, we chose immediate or operand 2. Then, according to the instruction bits, we do ADD/I, NOR, AND/I, and NAND operation. Then we give the result as output.

Also, we created the registers from scratch as well. We first created the D-Latch and then the D Flip Flop and then the one-bit register. After that, we needed to have registers as 10 bits and 18 bits, so we made them as well. After that, we also created a register file with 16 registers. It writes and reads from these 16 registers using their addresses.

Also, we created a sign extender for 6-bit to 18-bit and 6-bit to 10-bit as well. We created a zero extender for 6-bit to 18-bit too.

Then, we created a program counter that keeps the value of itself and increment and do jump operation accordingly. We also used an incrementer for this operation that just does the +1 operation.

1. Control Unit Design and Finite State Machine

After creating the components, we had to create a control unit and finite state machine for instructions to be fetched, decoded and done. For this, we had to create an opcode decoder and control unit. Also, we needed to create a finite state machine to give the right signals to be activated.

To start with, for our instructions we created a finite state machine to see which states we will be needing and what signals we should have. Even if it is not a best design, we will put our design below and give its details.

A diagram of a diagram

Description automatically generated with medium confidence

We have designed our FSM with starting with INIT state which resets using the reset function. Then, we move on to the FETCH state that gets the instruction from instruction memory. Then, we decode the instruction in the DECODE state. Then according to the opcode, we divide to 5 branches ALU, CMP, LD, ST, JUMP. After doing it successfully, we moved on to the write part of it to write it to memory or register.

A screenshot of a spreadsheet

Description automatically generated

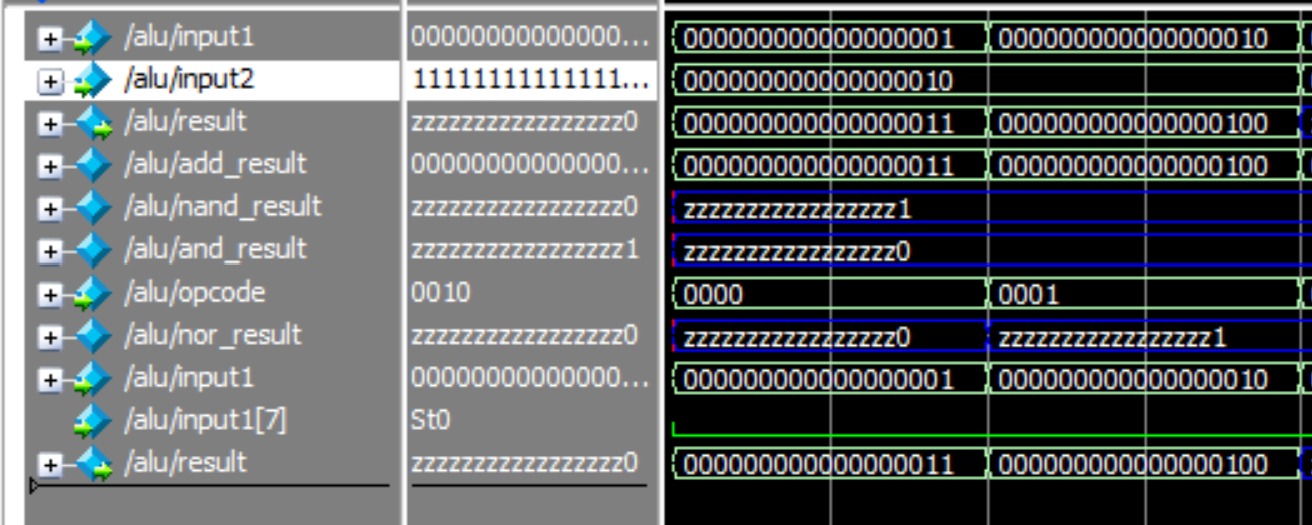
We constructed a truth table for this and accordingly, we created a component for it as well.

Then, moving on to our Control Unit and Opcode Decoder in Logism, we created signals for every write, read, jump enable signal there. To keep it simple, we can say that for each instruction there is an enable bit, to read there is an enable bit, to write there is an enable bit. Also, we check the jump conditions there as well.

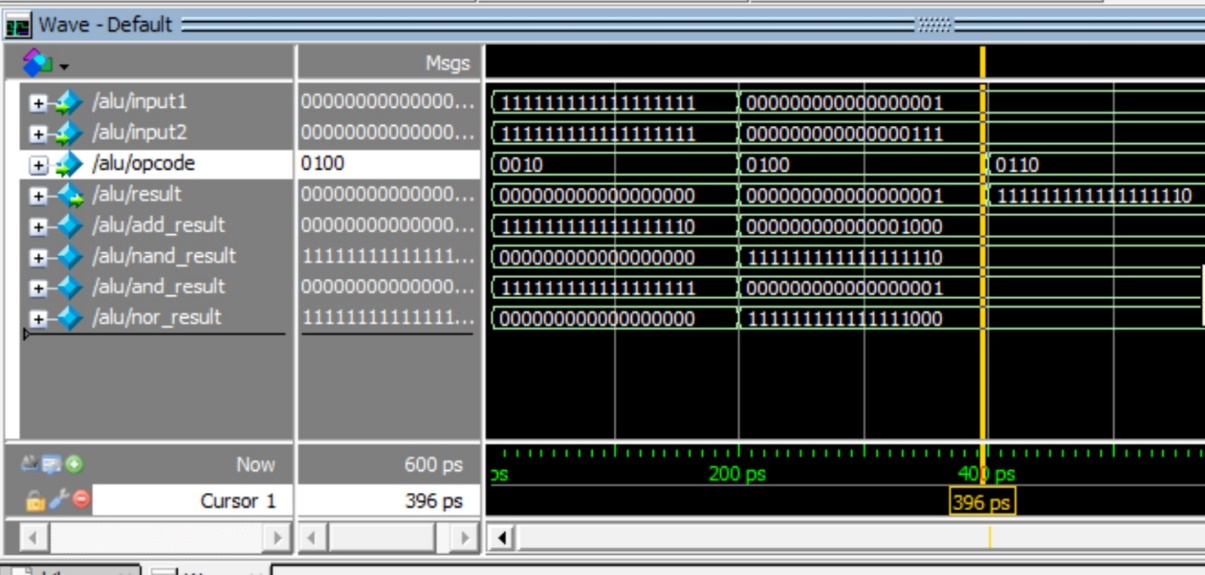
In main, we used Control Unit, ALU, Register File and other non-dependent components and tied their wires together to get the values, we have a clock and a start button there too.

1. Verilog Design and Simulation Results

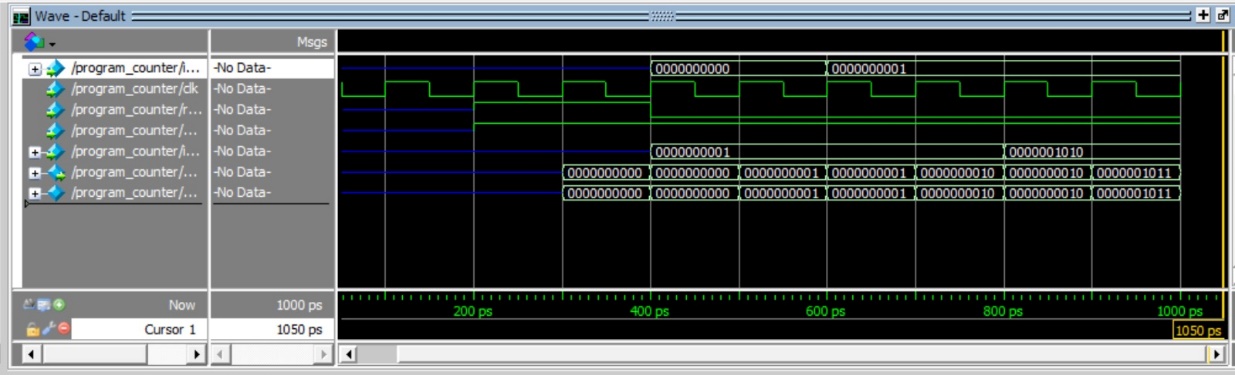
First of all, we wrote our modules for components such as ALU, Program Counter, RAM, Comparator, Control Unit, Opcode Decoder in the CPU design in the logisim using the Verilog language. We simulated these modules after compiling them in the Modelsim application.



In this image, the addition process is first performed by giving the ADD opcode and the numbers to be added in the register as input to the ALU. After running these inputs 000000000000000001 and 000000000000000010, we can see that alu\_result gives the correct result 00000000000000011. Then, when ADDI is given as opcode, 000000000000000010 as input, and 000000000000000010 as immediate value, the result is 000000000000000100.



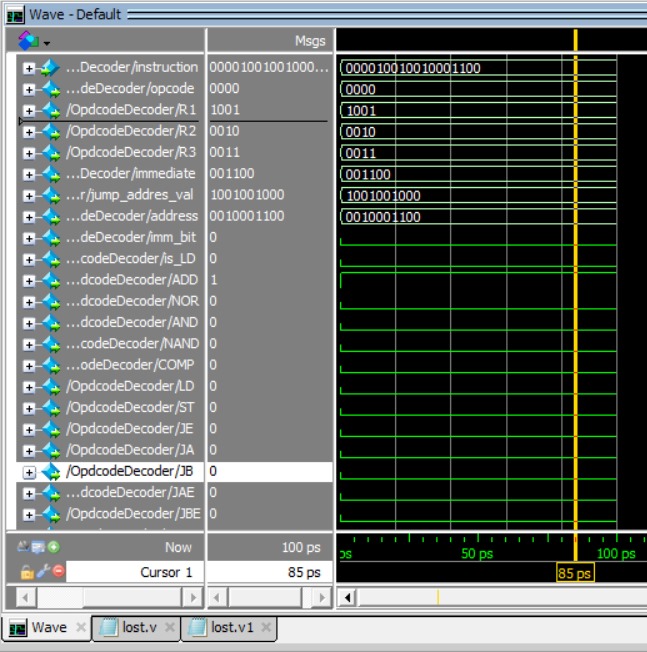
We see that NOR, AND, NAND operations are simulated on the ALU, respectively. We give 0010 opcode for NOR and two inputs are given as 111111111111111111, 11111111111111111 and we get the result as 000000000000000000. For AND, we give the inputs 000000000000000001 and 000000000000000111 and get the result 000000000000000001. Finally, we give the inputs 0000000000000000001 and 000000000000000111 for NAND and get the result 111111111111111110.



In this image, we see that the program counter increases according to the given increment value every time we run it.



It seems that FSM goes into different states every time it runs. Respectively, 0000 init state, 0001 fetch state, 0010 decode state, 0011 alu state and finally 1000 write state.



We see that the instruction 000010010010001100 is given to the Opcode Decoder as input. When we simulate it, we see that the AND instruction is 0000, R1 1001, R2 1001, R3 0010 as opcode.