CSE3215

DIGITAL LOGIC DESIGN TERM PROJECT

Ferhat Sirkeci 150120067

Doğukan Onmaz 150120071

Muhammet Akyüz 150120028

Aim of the project:

General purpose of this project is implement a processor works with specific instruction set: (ADD, AND, NAND, NOR, ADDI, ANDI, LD, ST, CMP, JUMP, JE, JA, JB, JAE, JBE)

There are many components such as; Register File, Instruction Memory, Data Memory, Control Unit, Arithmetic Logic Unit (ALU)…

Processor writes & reads **18-bit data** from registers. There are **16 registers** and **10 bits wide address.**

Instruction Set Architecture / Assembler:

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD	0	0	0	0	DEST				SRC1				0	0	SRC2			
AND	0	0	0	1	DEST				SRC1				0	0	SRC2			
NAND	0	0	1	0	DEST				SRC1				0	0	SRC2			
NOR	0	0	1	1	DEST				SRC1				0	0	SRC2			
ADDI	0	1	0	0	DEST				SRC1				IMM					
ANDI	0	1	0	1	DEST				SRC1				IMM					
LD	0	1	1	0	DEST				ADDRESS									
ST	0	1	1	1	SRC1				ADDRESS									
CMP	1	0	0	0	0	0	0	0	0	0	OP1				OP2			
JUMP	1	0	0	1	0	0	0	0	ADDRESS									
JE	1	0	1	0	0	0	0	0	ADDRESS									
JA	1	0	1	1	0	0	0	0	ADDRESS									
JB	1	1	0	0	0	0	0	0	ADDRESS									
JAE	1	1	0	1	0	0	0	0	ADDRESS									
JBE	1	1	1	0	0	0	0	0	ADDRESS									

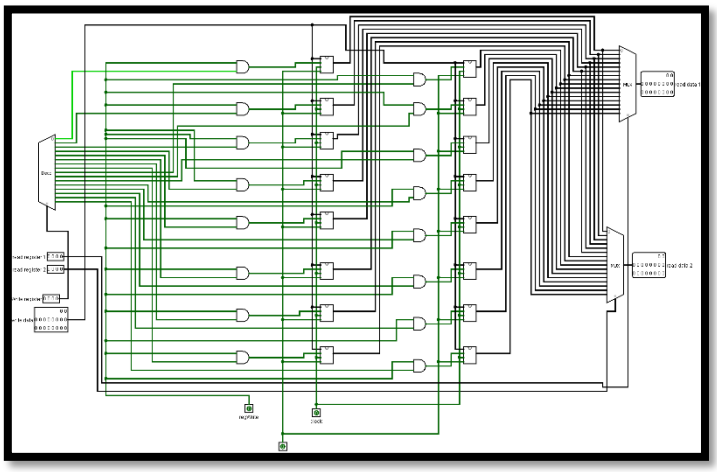
ISA: All the instructions are shown in this table.

We designed an **assembler** in java environment. Assembler takes the instruction from input.txt file and convert it into hexadecimal code, then writes that hexadecimal code in output.txt file. Assemble process starts with reading file, then instructionSeperator() function parses the instruction according the opcode. There are also several functions converting the parts of the instructions to binary. After converting all of the instructions to the binary, binaries are converted to hexadecimal and concatenated lastly.

Ex: Instruction: **ADD R5 R0 R2** converted into hexadecimal code: **01402**

Register File

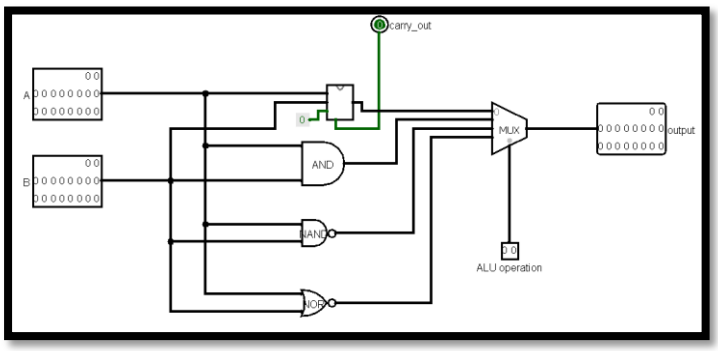
Register File reads two registers and fetches the **18-bit instructions** from inside of them. It also reads a **18-bit write data** and register to write. Decoder is used for finding the correct register to write and Multiplexers are used for finding the correct register to read.



Arithmetic Logic Unit (ALU)

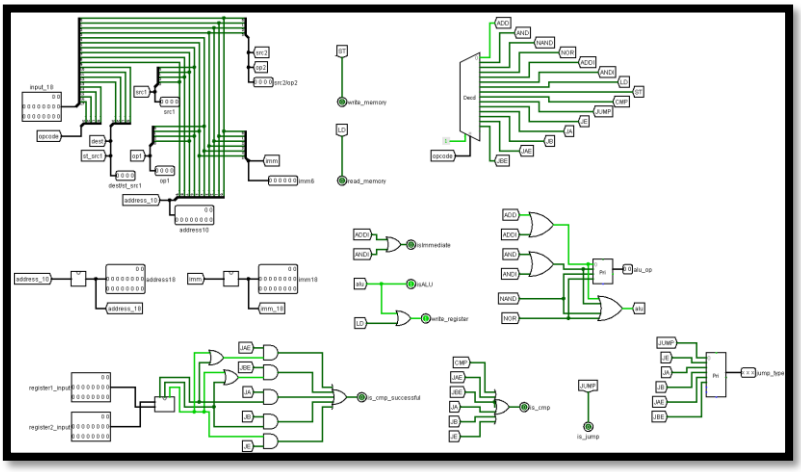
Arithmetic Logic Unit (**ALU**) is for arithmetic operations. This unit consists of two **18-bit inputs**, **18-bit Adder** and **2-bit selector (alu_operation)** for the multiplexer. 4 to 1 multiplexer with the 2 bit selector determines which arithmetic operation will be done.

(Operations: ADD, AND, NAND, NOR)



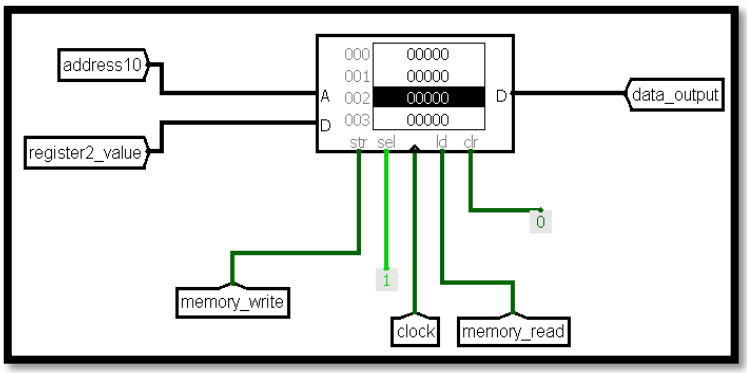
Control Unit

Control unit is responsible for **parsing** the 18-bit instruction and **grouping** its bits (opcode, src1, src2, op1, op2, dest, address, immediate) according to ISA. A **decoder** determines what operation is instruction holds and produce a signal for that operation. An **18-bit comparator** compares two 18-bit registers and determines whether jump condition is correct or not (JAE, JBE, JA, JB, JE). An **encoder** assigns current arithmetic operation to **alu_operation** output, and this output goes to **ALU** as selector of multiplexer.

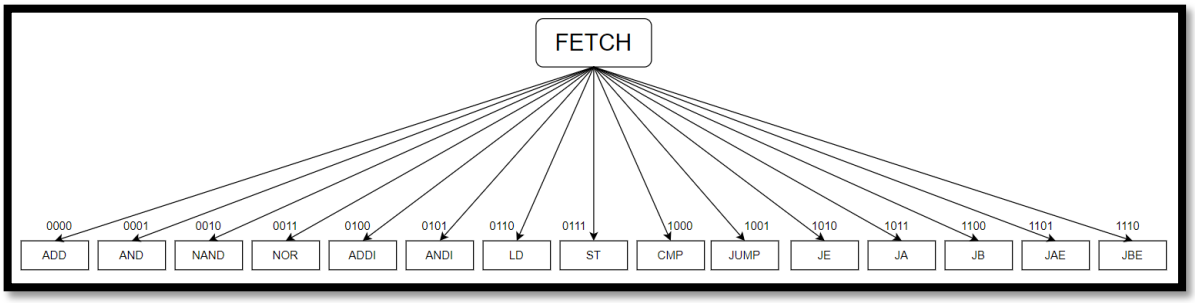


Data Memory

RAM is used as data memory. RAM takes **10-bit adress** and **register value** as inputs. 10-bit address determines which address to write or read and register value represents the value of the current register. It also takes two signals (**memory_write, memory_read**) as input. These signals determines whether register value will be written in RAM or current value in RAM will be read and store into register.

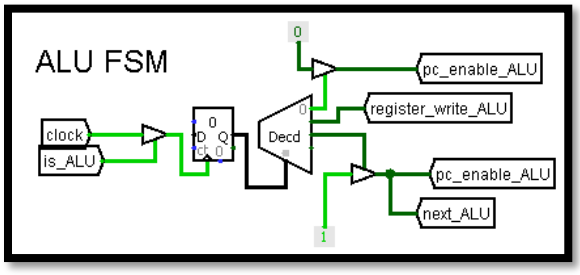


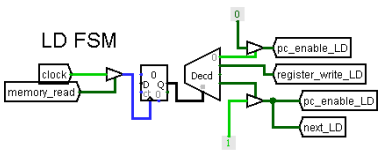
Finite State Machine (FSM)

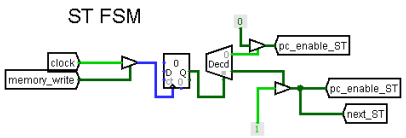


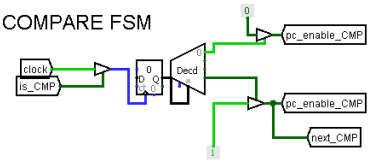
We have implemented 5 different finite state machines. They are for ALU, LD, ST, CMP and JUMP operations.

For example, **ALU FSM** works with 3 clock time. In the first state, pc_enable signal is disabled so that the program counter does not move to the next instruction. In the next state, **register_write** signal is on and value will be written in register. In last state, pc_enable and next signals are enabled and program counter will move to the next instruction.









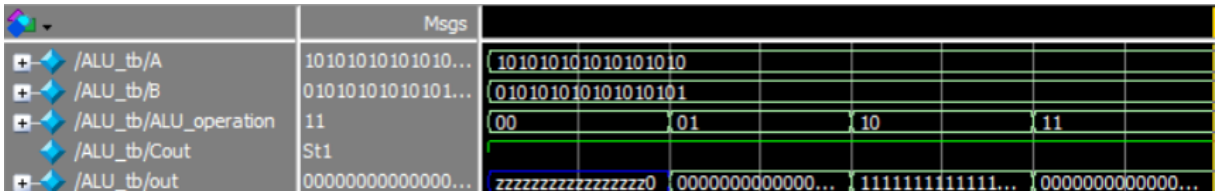
Verilog

We have implemented 30 verilog file. Verilog code of our components and some necessary units(mux, encoder etc.) are implemented. We have simulated some of the components and recorded their wave graph.

adder2bit.v	Verilog	14	01/05/2024 04:17:51
adder4bit.v	Verilog	15	01/05/2024 04:31:09
adder8bit.v	Verilog	16	01/05/2024 04:18:52
adder18bit.v	Verilog	17	01/05/2024 06:46:29
adder18bitTestBan...	Verilog	26	01/05/2024 10:22:39
ALU.v	Verilog	13	01/05/2024 04:14:00
ALU_TestBanch.v	Verilog	28	01/05/2024 10:32:52
comparator4bit.v	Verilog	7	01/05/2024 02:32:04
comparator18bit.v	Verilog	8	01/05/2024 02:30:48
comparator18bit_T...	Verilog	27	01/05/2024 10:31:41
Compare.v	Verilog	5	01/05/2024 02:55:56
Compare_TestBan...	Verilog	29	01/05/2024 10:34:06
controlUnit.v	Verilog	21	01/05/2024 09:59:20
controlUnit_TestBa...	Verilog	30	01/05/2024 10:35:48
decoder4to16.v	Verilog	22	01/05/2024 08:46:30

dFlipFlop.v	Verilog	3	01/05/2024 02:34:03
dLatch.v	Verilog	2	01/05/2024 01:56:28
encoder4to2.v	Verilog	23	01/05/2024 09:13:18
encoder8to3.v	Verilog	24	01/05/2024 09:40:35
extend6to18.v	Verilog	0	01/05/2024 04:20:06
extend10to18.v	Verilog	1	01/04/2024 05:46:29
FSMfiles.v	Verilog	25	01/05/2024 10:18:17
fullAdder.v	Verilog	19	01/05/2024 04:30:46
fullComparator.v	Verilog	9	01/05/2024 02:30:49
halfAdder.v	Verilog	18	01/05/2024 04:29:33
halfComparator.v	Verilog	10	01/05/2024 02:30:49
mux.v	Verilog	6	01/05/2024 03:12:24
ProgramCounter.v	Verilog	20	01/05/2024 07:19:07
Register.v	Verilog	11	01/05/2024 04:24:12
registerFile.v	Verilog	12	01/05/2024 04:23:15
TwosNegativeToPo...	Verilog	4	01/04/2024 07:02:44

The graph below is the simulated wave graph of ALU. When **ALU_operation** input is 00 it performs ADD operation, AND for 01, NAND for 10, NOR for 11.



ALU wave graph