Süleyman Ahmet SÖNMEZ - 150117010
Ahmet TURGUT - 150117046
Fatih BAŞ - 150117048

CSE3015 - Project Report

About Project

In this project, we are supposed to do design a processor in Logisim. This processor will have 16-bits address width and 16-bits data width. It will support AND, ANDI, ADD, ADDI, LD, ST, CMP, JUMP, JE, JA, JB, JBE, JBA instructions. And also our controller must be in finite state machine manner not as micro-programmed manner.

AND instruction's form is "AND DEST,SRC1,SRC2". DEST, SRC1 and SRC2 are the registers. DEST = SRC1 AND SRC2.

ANDI instruction's form is "ANDI DEST,SRC1,IMM". DEST and SRC1 are the registers. IMM is 7 bits 2's complement value (between -64, 63). DEST = SRC1 AND IMM.

ADD instruction's form is "ADD DEST,SRC1,SRC2". DEST, SRC1 and SRC2 are the registers. DEST = SRC1 + SRC2.

ADDI instruction's form is "ADDI DEST,SRC1,IMM". DEST and SRC1 are the registers. IMM is 7 bits 2's complement value (between -64, 63). DEST = SRC1 + IMM.

LD instruction's form is "LD DEST,ADDR". DEST is the register. ADDR is the 10 bits address of the data which will be fetched from data memory. It fetches data in the corresponding data and writes it into DEST register.

ST instruction's form is "ST SRC,ADDR". DEST is the register. ADDR is the 10 bits address of the memory which will be store data from register. It fetches data in the DEST and writes it into corresponding memory address.

CMP instruction's form is "CMP OP1,OP2". OP1 and OP2 are registers. This instruction will compare two register's data and corresponding to the result it sets CF and ZF.
If OP1 > OP2, ZF and CF will be 0. If OP1 = OP2, ZF will be 1 and CF will be 0.
If OP1 < OP2, ZF will be 0, CF will be 1.

JUMP, JE, JA, JB, JBE, JBA instructions' form is "X ADDR". X is type of jump instruction. ADDR is 9-bits PC-relative signed address. If X is JUMP, ADDR will be added to current PC value directly. If X is other types of jump instruction ADDR will be added to current PC value depending current flag values. These flag conditions are:
If ZF = 1 and CF = 0, JE instruction will add ADDR to current PC value.
If ZF = 0 and CF = 0, JA instruction will add ADDR to current PC value.
If ZF = 0 and CF = 1, JB instruction will add ADDR to current PC value.
If ZF = 1 and CF = 1, JBE instruction will add ADDR to current PC value.
If CF = 0, JAE instruction will add ADDR to current PC value.

Step 1 - ISA And Assembler Design

In this step, we needed to design our ISA (Instruction Architecture Set) which includes all instructions given above.

In our ISA we used 3-bits for opcode[15:13]. For all jump instructions we used “100" opcode and for distinction between different jump instructions we used last 4-bits[3:0].

After designing ISA, we had to design an assembler for converting assemble language to 16-bits hex values. We did that in Java programming language. It takes an input.txt in assemble language and it writes corresponding hex values into output.txt.

Step 2 - Logisim Components Design

In this step we needed to design our processor’s components except control unit in Logisim. So we had to design our register file, ALU, PC register and comparator.

1- Register File
For designing register file we started with designing our 16-bits registers. For this purpose we designed d-latch, and flip-flop.

Flip - Flop:
In flip-flop we used two d-latches. For clearing flip-flop, we used MUX. In that MUX, we have D at 0th index and constant 0 in 1st index. If clear signal comes, in clock rising edge our flip-flop is set to 0. Else, our flip-flop is set D’s value.
16-Bits Register:

In our 16-bits register we used 16 flip-flop sequentially which all of them stores 1 bit. If enable bit is zero in clock rising edge data in the register stays same. Else, 16-bits input writes into register. For this purpose we used MUX for selecting between previous data and new coming data. Also we have clear input, if it is on in clock rising edge our register clears itself.

Register File:
In our register file we have 8 registers (from R0 to R7). In here we have 7 inputs which are data (comes from ALU or data memory), write_select_reg (determines which register to write incoming data), write_enable (determines incoming data will write or not into register which is given in write_select_reg), read reg1 and read reg2 (determine which registers will read). And we have 2 outputs which are used for giving read register values to outside of register file.

2- ALU

For designing ALU we needed to design our adder first. For this purpose we designed half-adder, full-adder.

After designing these two component we composed 1 half-adder and 15 full-adder for designing 16-bits adder.

16 - Bits Adder:

In here we have two 16-bits inputs which are A and B (signed integers). Our adder adds two values gives result to output.

ALU:

In ALU we have 4 inputs which are A, B (16-bits value), select_bit (determines which operation's result will give into output) and aluSignal (determines result will be give into output or not).

3- PC Register

In PC register we have 5 inputs which are offset (comes from jump instruction's ADDR), addOffsetSignal, increment_PC, clock and clear.
In 16-bits register (which we designed earlier) we are storing current PC value. When increment_PC signal is on in clock rising edge it gives its output to adder (which we designed earlier). Other input of adder comes from MUX. In MUX 0th index we have 16-bits constant 1 and 1st index we have 16-bits signed integer offset which is extended with 9x16 extender. When addOffsetSignal is on in clock rising edge MUX selects offset value for adder's input. Else it selects 16-bits 1 for adder's input.
After this selection if increment_PC signal is on in clock rising edge add operation are performed and result writes into PC register.

4- Comparator

For comparator design, we designed first half-comparator and full-comparator.

After designing these 2 component, we composed 1 half-comparator and 15 full-comparator for designing 16-bits comparator.

16-Bits Comparator:
In 16-bits comparator we have 2 inputs which are 16-bits signed integer OP1 and OP2. And according to compare process we have 3 outputs which are gt, lt, eq.

STEP 3 - Logisim Control Unit Design

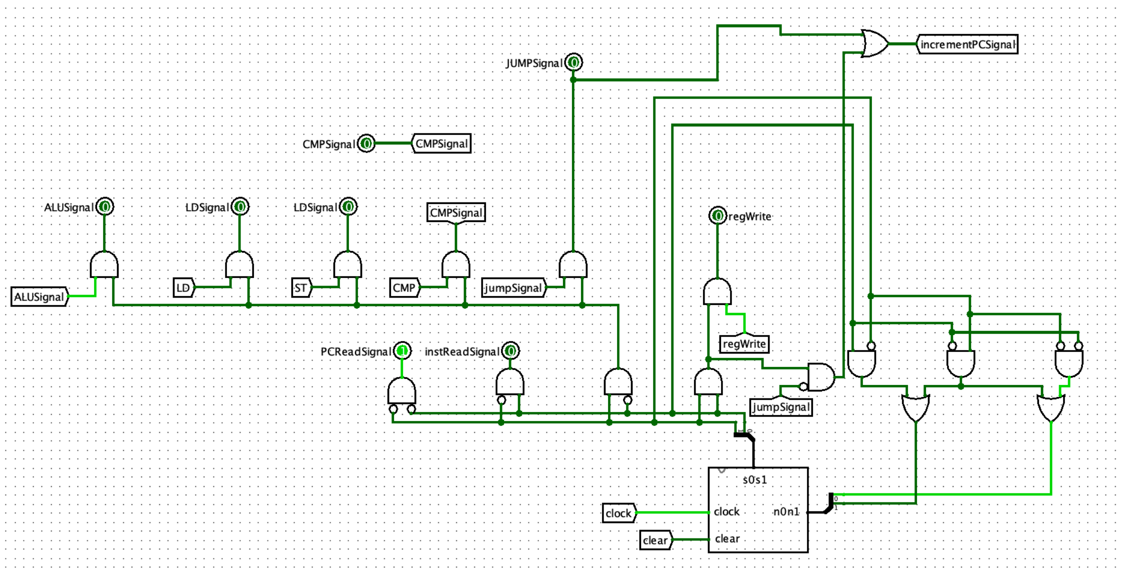
In this step we had to design a control unit which controls our components (which are designed in previous step) and complete our processor.

1- Control Unit

First of all we designed an instruction parser which parses coming instruction from instruction memory according to our ISA (which are designed in 1st step).
After parsing instruction, we need do determine to corresponding operation for that parsed instruction.

For that purpose we have used 2 decoder. In first decoder, we determine operation if opcode not belongs to a jump instruction. If it is jump instruction's opcode then in second decode we determine which jump instruction it is with help of jump_selector (Last 4 bits of instruction).

After determining operation, we implemented a four state Finite State Machine.



First two state is same for all instructions (Reading PC value from PC register and reading instruction from instruction register). In third step according to operations we have some output signals. In four state we incrementing current PC and writing data to register file.

PCReadSignal fetches current PC value.

InstReadSignal fetches corresponding instruction from instruction memory to instruction register and send this instruction to control unit.

Signal Handling Mechanisms:

If operation is ADD or ADDI or AND or ANDI we are sending ALUSignal to ALU.

If ADDI or ANDI operation is performing, we are sending imm_signal to ALU for selecting coming imm value instead of register value.

If operation is ADD or ADDI operation we are sending addSignal to ALU for performing add process. Otherwise, our ALU will performs an and (AND, ANDI) process.

For reading registers' value we are using read_reg1 and read_reg2 inputs of register file. If ST signal on MUX selects st_src. If cmpSignal on MUX selects OP1 and OP2 instead of SRC1 and SRC2.

For writing purposes in fourth state, we are sending regWrite signal to register file. After performing ALU, we are writing result of ALU to register file or if LD operation is performing we are writing fetched data from the data memory.

For incrementing PC we are sending incrementPCSignal to PC register.

If operation is CMP we are using comparator for compare two 16-bits values and according the result we are setting CF and ZF flags. In here we also used two 1-bit register to store CF and ZF flags.

If operation is a jump instruction firstly we determine that which jump instructions can perform according to CF and ZF flags. Then we are sending jumpSignal.

If operation is ST, we are reading data from register and writing that value into data memory with stSignal.