CSE 3015
DIGITAL LOGIC DESIGN

TERM PROJECT

Abstract

18-bit processor implementation using Logisim

Beyzanur Çabuk 150119632
Bihter Nilüfer Akdem 150119810
Melike Zeynep Avşar 150119818
Nidanur Tekin 150119794

1. DESIGN

In this project, an 18bit architecture processor will be designed and implemented in Logisim which will support instruction set; AND, OR, ADD, LD, ST, ANDI, ORI, ADDI, XOR, XORI, JUMP, BEQ, BGT, BLT, BGE, BLE. The data bus and address sizes will be 18-bits.

Firstly, the Instruction Set Architecture (ISA) was designed. The details and design details of ISA is represented in below table.

OPERATION	[17-14]	[13-10]	[9-6]	[5-2]	[1-0]
ADD	0000	DEST	SRC1	SRC2	-
AND	0001	DEST	SRC1	SRC2	-
OR	0010	DEST	SRC1	SRC2	-
XOR	0011	DEST	SRC1	SRC2	-
ADDI	0100	DEST	SRC1	IMMEDIATE	
ANDI	0101	DEST	SRC1	IMMEDIATE	
ORI	0110	DEST	SRC1	IMMEDIATE	
XORI	0111	DEST	SRC1	IMMEDIATE	
LD	1000	DEST	LD/ST ADDRESS		
ST	1001	SRC	LD/ ST ADDRESS		
JUMP	1010	JUMP ADDRESS			
BEQ	1011	OP1	OP2	BRANCH ADDRESS	
BLT	1100	OP1	OP2	BRANCH ADDRESS	
BGT	1101	OP1	OP2	BRANCH ADDRESS	
BLE	1110	OP1	OP2	BRANCH ADDRESS	
BGE	1111	OP1	OP2	BRANCH ADDRESS	

There will be 16 operations designed and implemented in this architecture. Therefore, we decided to allocate 4 bits for opcode. To have a more readable design and simplify the implementation, we decided to use last 4 bits for operation. Rest of the opcode is divided for source and destination registers, immediate value, and addresses. Again, we will allocate 4 bits for register addressing as we will implement 16 registers. First bits of code are reserved for immediate value, jump, branch, and LD/ST addresses. For ADD, AND, OR, and XOR operations, after using necessary bits for registers, the first 2 bits left unused.

We need to differentiate branch operations during the design. Although in the project document it's asked to use special bits for achieving this, we noticed that we could use different operation codes for each branch. Thus, we won't need to use those bits.

The address of registers is shown below.

REGISTER	CODE
R0	0000
R1	0001
R2	0010
R3	0011
R4	0100
R5	0101
R6	0110
R7	0111

REGISTER	CODE
R8	1000
R9	1001
R10	1010
R11	1011
R12	1100
R13	1101
R14	1110
R15	1111

2. ASSEMBLER

We chose Python version 3.9.5 as our programming language choice to develop our assembler. Python is known with its easy coding and achieving tasks with less code. Visual Studio Code was used as an integrated development environment (IDE) with its features like being free and light, easy to adopt, clean interface and fast responses.

Firstly, operations and registers were added to 2 dictionaries with their names and binary codes.

```
# create dictionaries
operations = {"ADD": "0000",
              "AND": "0001",
              "OR": "0010",
              "XOR": "0011",
              "ADDI": "0100",
              "ANDI": "0101",
              "ORI": "0110",
              "XORI": "0111",
              "LD": "1000",
              "ST": "1001",
              "JUMP": "1010",
              "BEQ": "1011",
              "BLT": "1100",
              "BGT": "1101",
              "BLE": "1110",
              "BGE": "1111"}
```

```
registers = {"R0": "0000",
             "R1": "0001",
             "R2": "0010",
             "R3": "0011",
             "R4": "0100",
             "R5": "0101",
             "R6": "0110",
             "R7": "0111",
             "R8": "1000",
             "R9": "1001",
             "R10": "1010",
             "R11": "1011",
             "R12": "1100",
             "R13": "1101",
             "R14": "1110",
             "R15": "1111"}
```

Then the *input.txt* file is opened for reading. A binary result variable is declared. Each line read and split by first space and then comma. Firstly, operation code found from operations dictionary and added to binary result. Register binary values read from dictionary according to register number and added to the result. The address and immediate values are converted to fixed length and signed binary values to be merged to the binary result. Lastly 2 zero bits added to ADD, AND, OR, and XOR operations to make a 20 bits result. Final binary result is converted to 5 digits hex string and added to the hex_result variable.

```
with open("input.txt", "r") as inputFile:
    lines = inputFile.readlines()
    # loop through each instruction
    for line in lines:
        # create variable for binary result, add 2 bits to make total 20 bits
        binary_result = "00"
        # split line with space to define operation
        parts = line.replace('\n', "").split(" ")
        operation = parts[0]
        binary_result += operations[operation]
        # split rest line with comma
        regs = parts[1].split(',')
        # add registers
        for reg in regs:
            if reg.startswith("R"):
                binary_result += registers[reg]
        # check operation for various results
        if operation == "ADD" or operation == "AND" or operation == "OR" or operation == "XOR":
            # first 2 bits
            binary_result += "00"
        elif operation == "LD" or operation == "ST":
            # LD/ST address value
            binary_result += '{:010b}'.format(int(reg) & 0x3ff)
        elif operation == "JUMP":
            # jump address value
            binary_result += '{:014b}'.format(int(regs[0]) & 0x3fff)
        else:
            # immediate or branch address value
            binary_result += '{:06b}'.format(int(reg) & 0x3f)

        # produce result string as hex
        hex_result += '{0:05x}'.format(int(binary_result, 2)) + "\n"
```

After looping all lines in input file and converting them to hex values as described above, we need to create an output file and write these hex values in that file. Here, we chose *output.hex* as file name. If the file does not exist it's created, else it's cleared, and new values are added.

```
# open output file for writing
with open("output.hex", 'w+') as outputFile:
    outputFile.writelines("v2.0 raw\n")
    outputFile.writelines(hex_result)
```

To run the application, the following statement can be executed in target folder where the input file should also be in.

python.exe assambler.py

3. IMPLEMENTATION

Logisim v2.7.1 used during the implementation of this project. The explanation of the circuits generated are detailed below.

3.1 Register File:

We have 16 registers in register file with 18-bits data size each. 2 inputs are used to choose which registers to get data from where 1 input addresses the register to be written if writing control input is enabled. In writing case, the data input is written to the target register. During the implementation of this circuit, Decoder, Multiplexers, AND gates, clock and registers are used.

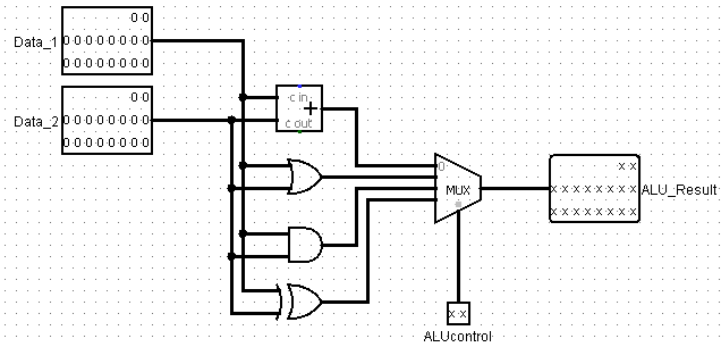
Beside ready to use circuits in Logisim, bit-extender, added and comparator circuits are designed for operations.

3.2 ALU:

ALU is responsible for the arithmetic operations. There are 2 18-bits input to work on and 1 18-bits output as result. The decision of the operation which will be chosen as result is done by the 2bits of ALU control input that is generated in Control Unit. By the help of a multiplexer and control inputs, we can decide the operation to be executed.

ALUControl bits and related operations are below.

OPERATION	ALUControl
ADD/ADDI	00
ORR/ORRI	01
AND/ANDI	10
XOR/XORI	11



3.3 Program Counter (PC):

Program counter controls the reading of instructions from the read-only memory. We have used We have enableJump, branch, clock, reset, next, JumpAddress and BranchAddress inputs. FSM generates when to enable PC, execute next statement or jump to another address. With the help of a multiplexer, if the jump function is enabled, jump or branch address is added to current counter value and loaded into counter. Meaning next instruction that will run would be the relative address to the counter value.

There's only one operation which is 18-bit adder for both positive and negative addresses. That's because negative values are kept signed.

The clock is given from outside to PC, but it's also controlled by enablePC signal from FSM circuit. Reset function is used to control the simulation and for debugging purposes.

3.4 Control Unit:

Control unit is the circuit where we decide and differentiate the opcode to its partitions. We split opcode into its bits and take last 4 bits for operation choice. We have the inputs of instruction from ROM and 2 data inputs from register file. The decision of the operation is decided and sent as output. If this will be a branch operation, the result is executed it and sent again as a signal to be used in program counter. The 2^{nd} register output is chosen from either source itself or immediate/branch address data. Immediate and branch is also chosen from internal signals.

3.5 Finite State Machine (FSM)

In FSM it's decided which instruction will be executed next and controlling of other circuits according to the state of the current instruction. The decision of program counter enabling, register writing, reading next instruction or jumping to a relative address is controlled by the circuits in FSM.