CSE 3055 Digital Logic Project Report

Prepared By: Musa Özkan, Barış Giray Akman, Hakkı Kokur
Instructor: Betül Demiröz Boz

February 12, 2024

1 Students

Musa ÖZKAN - 150121058
Barış Giray AKMAN - 150121822
Hakkı KOKUR - 150120033

2 Introduction

The project involves the design of a processor with a specific instruction set architecture (ISA). The processor has a defined width for addresses (10 bit) and registers (4 bits), and it is composed of several parts including a Register File, Instruction Memory, Data Memory, Control Unit, and an Arithmetic Logic Unit (ALU).
The processor supports a variety of operations, such as arithmetic operations, logical operations, memory access, and conditional jumps based on flags.

3 Instruction Set Architecture

The ISA was created by defining the opcode, operand registers, immediate values, and other necessary fields for each instruction. The ISA below has the details of format for each instruction type. For example, the ADD operation uses opcode '0000', specifying two source registers (SR1, SR2) and one destination register (DR).
The ISA design includes binary representations for each part of the instruction, which allows for direct translation from mnemonic to machine code.

	[17:14](opcode)	[13:10]	[9:6]	[5:2]	[1:0]
ADD	0000	DR	SR1	SR2	00
ADDI	0001	DR	SR1	IMM6	
AND	0010	DR	SR1	SR2	00
ANDI	0011	DR	SR1	IMM6	
NAND	0100	DR	SR1	SR2	00
NOR	0101	DR	SR1	SR2	00
JUMP	0110	ADDR10			0000
LD	0111	DR	ADDR10		
ST	1000	SR	ADDR10		
CMP	1001	OP1	OP2	000000	
JE	1010	ADDR10			0000
JA	1011	ADDR10			0000
JB	1100	ADDR10			0000
JAE	1101	ADDR10			0000
JBE	1110	ADDR10			0000

Figure 1: Instruction Set Architecture

4 Assembler

```python
def convert_ADD_ADDI_AND_ANDI_NAND_NOR(instruction: str) -> str:
    # get the operation
    operation = get_operation(instruction)
    result = ""
    # set the operation opcodes
    if operation == "ADD":
        result += "0000"
    elif operation == "ADDI":
        result += "0001"
    elif operation == "AND":
        result += "0010"
    elif operation == "ANDI":
        result += "0011"
    elif
        r (variable) operation: str
    elif operation == "NOR":
        result += "0101"

    if "I" in operation:
        registers = get_registers(instruction)
        result += convert_register(registers[0])
        result += convert_register(registers[1])
        result += convert_imm(registers[2], 6)
    else:
        registers = get_registers(instruction)
        result += convert_register(registers[0])
        result += convert_register(registers[1])
        result += convert_register(registers[2])
        result += "00"
    # check the size
    return result      You, last month • add, addi done
```

Figure 2: Converter Script

The script reads assembly-like instructions from a file, parses them to categorize each instruction, and applies specific conversion functions based on their types. By utilizing these functions, the script generates binary representations for every parsed instruction. Subsequently, it transforms these binary representations into their corresponding hexadecimal format before writing the output into a designated file. This comprehensive process allows for the translation of diverse assembly-like instructions into a machine-readable hexadecimal format, essential for testing and validating the functionality of a processor according to its specific design requirements.

4.1 Input & Output Formats Explained

The guidelines ensure clarity and coherence in the assembly-like instructions. Operations are delineated by spaces, while registers are separated by commas, providing a clear distinction between operands. Immediate values, when required, follow registers and are also separated by commas, enhancing readability and precise interpretation of each instruction by the processor according to the ISA guidelines.

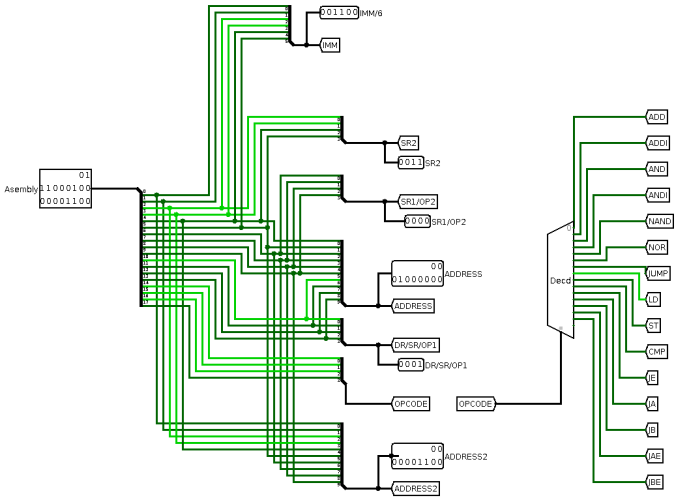
5 Logisim

List of Components

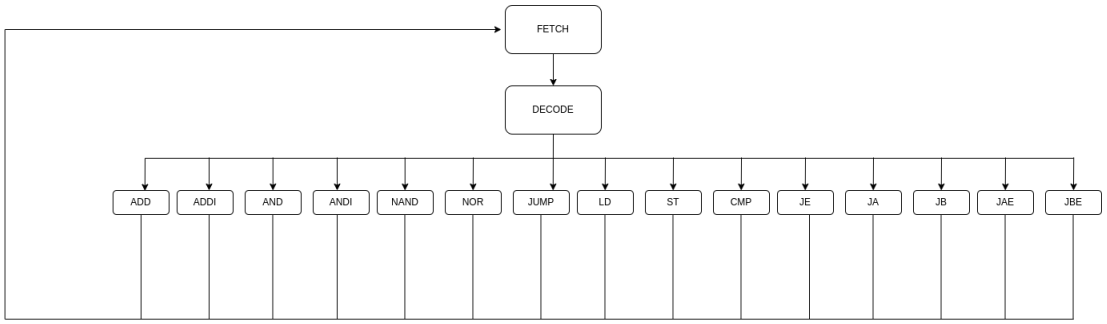
1. **Register File**
   - **Inputs:** *clk*, *reset*, *write_address* [3:0], *read_address1* [3:0], *read_address2* [3:0], *write_enable*, *input_data* [17:0]
   - **Outputs:** *data_out1* [17:0], *data_out2* [17:0]
   - **Components:** *registers* [0:15]: 16 registers, each 18 bits wide
   - **Behavior:**
     - Initializes registers to 0 at start.
     - Writes data to the selected register when *write_enable* is asserted.
     - Reads data from specified registers continuously.

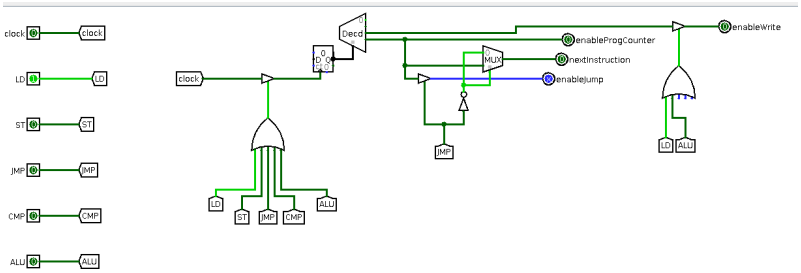
2. **CPU**

- Opcode Extraction: Use a combination of splitters to isolate the first 4 bits of the 18-bit input.
- Decoder: Create a 4-to-16 decoder using a combination of AND gates and NOT gates. Each output line (out1 to out16) will represent a decoded opcode.
- Control Signal Logic: Use additional logic gates to generate control signals based on the decoded opcode.



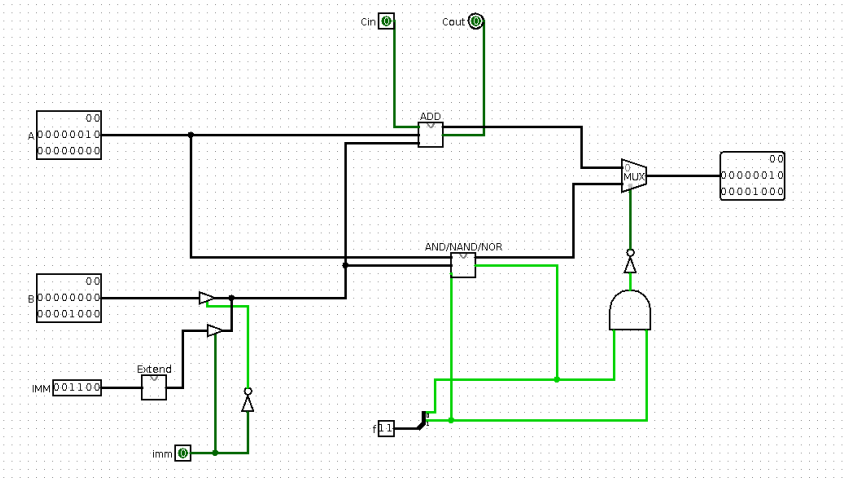
3. **Finite State Machine (FSM)**
The Finite State Machine (FSM) restart every four clock cycles. During the first clock cycle, the CPU decodes the binary input. In the subsequent clock cycle, it loads necessary information from RAM, the register file, and other sources, and performs calculations if required. Moving on to the third clock cycle, the CPU executes any write instructions, updating the register file or RAM as needed. Finally, in the fourth clock cycle, the program counter is enabled, allowing the CPU to proceed to the next instruction in the sequence.



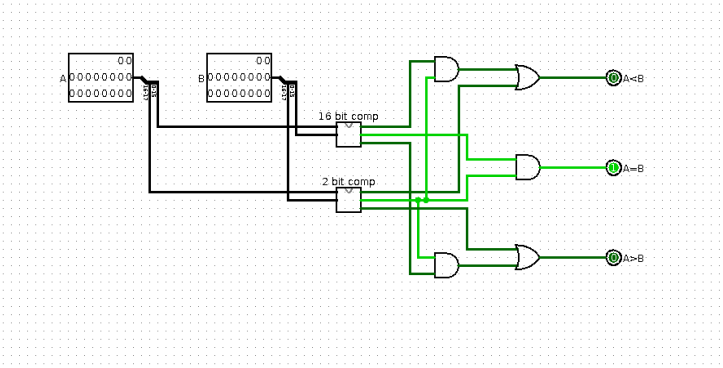


4. **ALU**

- Arithmetic Unit: Implement the 18-bit adder as described above.
- Logic Unit: Use AND, NAND, NOR gates for logic operations.
- MUX: Use multiplexers to select between arithmetic and logic unit outputs.
- Sign Extension: Use splitters and extenders to extend the immediate value.



5. **CMP (Comparator)**
   It reads the values at the specified addresses, performs a comparison based on A and B values over 18 bits, and determines the numerical relationship between A and B. Subsequently, it sets the CF and ZF values based on this relationship

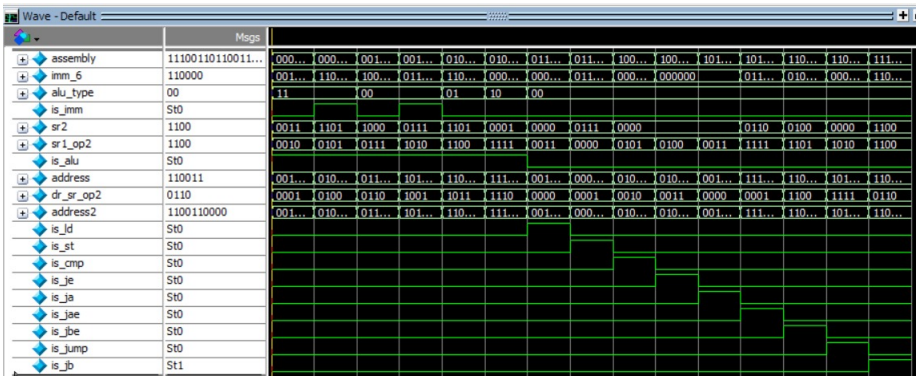


6. **Counter**
   According to the input value coming from previous image that we introduced , current address value is incremented by 1 or re-calculated by using input . (output += input) or (output++) . When pc enable signal which comes from FSM is 1 , this operation is processed.

7. **Data Memory**
   We utilize a RAM component as data memory with a 10-bit address input for read/write operations. Reg1-VALUE holds the data read from the register, and data out contains the value from the register specified by the 10-bit address. When IS-ST is enabled (1), it writes the register value to the RAM. Conversely, when IS-ST is activated (1), it reads from the RAM and stores the value into the designated register.

6 Verilog

6.0.1 CPU



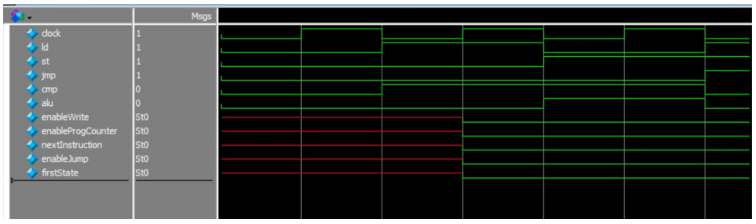
Inputs: *assembly* (18 bits)
Outputs: Control signals (imm_6, alu_type, is_imm, $sr2$, $sr1\_op2$, is_alu, etc.)
Key Components:

- *Opcode Extraction*: Extracts the 4-bit opcode from *assembly*.
- *Decoder*: Uses a 4-to-16 decoder (*decoder_4to16*) translating the opcode to 16 output lines (*out*1 to *out*16).

Logic Flow:

- *always @(*) Block*: Decodes the *assembly* instruction using the opcode and assigns its parts to various control signals.
- Control signals determination is based on the opcode and logical operations between the decoded output lines.

6.0.2 Finite State Machine

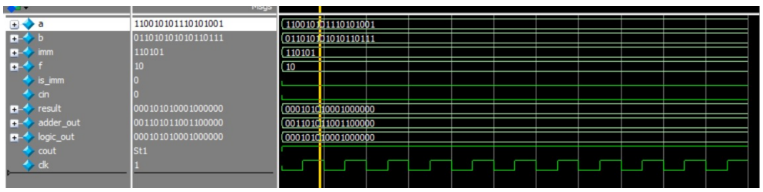


Inputs: *assembly* (18 bits)
Outputs: Control signals (imm_6, alu_type, is_imm, $sr2$, $sr1\_op2$, is_alu, etc.)
Key Components:

- *Opcode Extraction*: Extracts the 4-bit opcode from *assembly*.
- *Decoder*: Utilizes a 4-to-16 decoder (*decoder_4to16*) to translate the opcode into 16 output lines (*out*1 to *out*16).
  Logic Flow:
  - *always @(*) Block*: Decodes the *assembly* instruction using the opcode and assigns its parts to various control signals.
  - Control signals are determined based on the opcode and logical operations between the decoded output lines.

6.0.3 Arithmetic Logic Unit (ALU)

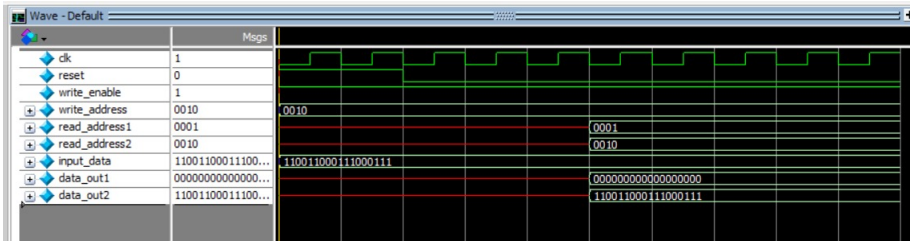
Inputs: a (18 bits), b (18 bits), imm (6 bits), f (2 bits), is_imm, cin
Outputs: $result$ (18 bits), $cout$, $adder\_out$ (18 bits), $logic\_out$ (18 bits)



6.0.4 Registers and Register File

Register File

- **Inputs:** *clk*, *reset*, *write_address* [3:0], *read_address1* [3:0], *read_address2* [3:0], *write_enable*, *input_data* [17:0]
- **Outputs:** *data_out1* [17:0], *data_out2* [17:0]
- **Components:** *registers* [0:15]: 16 registers, each 18 bits wide
- **Behavior:**
  - Initializes registers to 0 at start.
  - Writes data to the selected register when *write_enable* is asserted.
  - Reads data from specified registers continuously.



6.0.5 Comparator (CMP)

CMP (Comparator)

- **Inputs:** A [17:0], B [17:0]
- **Outputs:** $A\_greater\_than\_B$, A_equals_B, $A\_less\_than\_B$
- **Behavior:**
  - Determines relations between A and B (greater, equal, or less) using conditional assignments.