CS 226 Project

Aniket Agrawal (190050017)
Parab Ridayesh Ramesh (190020079)
Shivam Raj (190050113)
Aman Singh (190020010)

May 23, 2021

Report

This project was done in a group of 4. Team members: Ridayesh Parab, Aman Singh, Aniket Agrawal, Shivam Raj.

Components

The overall module can be broken down into smaller parts or components as follows:

Data Path

1. Arithmetic and Logical Unit (ALU):

   This component is responsible for doing all the arithmetic and logical operations; such as addition, performing NAND operation between two operands, etc. It has two inputs (two temporary registers) and three outputs (answer, carry flag, zero flag).

2. Memory:
   As per the problem statement we have used a 2 byte addressable memory. It stores data and instructions.

3. Program Counter (PC): Its a 16 bit register which points to the address of the current instruction which is being executed.

4. Instruction Register (IR): Its a 16 bit register which stores the current instruction to be executed.

5. File Register (FR): Its a collection of 8 general purpose registers (R0 to R7); each of 16 bits. It has 5 inputs (read address, write address, write enable, clock, data).

6. Two Flag Registers (Carry and Zero flags): Store the values of carry and zero flags. The zero flag is set to 1 whenever an operation gives 0 as output. The carry flag is set to 1 if an operation results in a carry.

Control Path

Consists of a main controller which determines how components should interact with each other. Responsible for understanding instructions and asking the data path to perform operations accordingly.

Implementing Components

In our project, we have made separate modules for ALU, memory and controller components. The rest of the components (file register, PC and IR) were implemented and maintained by the main controller itself (in the form of variables). All components are synchronized by a clock.

1. **Memory:**

   This is the entity declaration of the memory.

```
entity Memory is
        port(
                clock         : in std_logic;
                data          : in std_logic_vector (15 downto 0);
                write_address : in std_logic_vector(15 downto 0);
                read_address  : in std_logic_vector(15 downto 0);
                we            : in std_logic;
                q             : out std_logic_vector (15 downto 0)
        );
end Memory;
```

   clock: Used for synchronization.
   data: This data will be used for writing into some memory address.
   write_address: Memory address where data has to be written. Note that data will be written only when 'we' is 1.
   read_address: Memory address from where data will be read.
   we (Write enable): Memory can be written to only when 'we' is 1.
   q: Outputs the value contained in read_address.

   Note: We have used a 256 sized memory; that is the memory can be indexed from 0 to 255. This was done because 2^{16} was too large to compile.
   Internally the memory has been implemented using an array (of size 256) of std_logic_vector.

2. **ALU:** This is the entity declaration for the ALU.

```
entity ALU is
   port(
                operation  : in  std_logic;
                inp1, inp2 : in  std_logic_vector(15 downto 0);
                outp       : out std_logic_vector(15 downto 0);
                c, z       : out std_logic
        );
end entity;
```

operation: When this is 1, the alu will add inp1 and inp2. When operation is 0, the alu will perform a nand operation in inp1 and inp2.
inp1, inp2: The two input operands.
outp: The output of the ALU.
c, z: Flags, which denote whether the last operation generated a carry or resulted in an output of 0.

3. **Program Counter (PC):** Implemented as a std_logic_vector(15 downto 0) variable in the main controller.
4. **Instruction Register (IR):** Implemented as a std_logic_vector(15 downto 0) variable in the main controller.
5. **File Register:** Implemented as an array of std_logic_vector(15 downto 0) of size 8. It is a variable in the main controller unit.
6. **Two Flag Registers (C and Z):** Implemented as two std_logic_vector(15 downto 0) variables in the main controller. flags(0) stores the carry flag while flags(1) stores the zero flag.
7. **Control path:** The control path has been implemented in the file IITB_Proc.vhd . Its entity declaration is:

```
entity IITB_Proc is
        port
        (
                clk, RST, start, writeF        : in std_logic;
                data, display_addr             : in std_logic_vector(15 downto 0);
                output_display                 : out std_logic_vector(15 downto 0);
                c, z                           : out std_logic;
                R0, R1, R2, R3, R4, R5, R6, R7 : out std_logic_vector(15 downto 0);
                stop_flag                      : out std_logic
        );
end entity;
```

clk: clock used to synchronize IITB Proc
RST: Used to restart the currently running program. When RST = 1, IITB Proc resets its flags, program counter to 0 and starts running instructions again.

start: Used by the user to denote that he is done setting up instructions in the memory and IITB Proc should start executing them.
writeF: Used by the user when he/she wants to write into the memory.
data: Used by the user to feed in instructions to the memory.
display_addr: Used by the user to navigate the memory. This is the memory address currently under operation by the user (writing into the memory or reading from the memory). output_display: We have tried to mimic an actual 'microprocessor'. This is the like the digital LED display of a microprocessor. It outputs the content in the memory address display_addr.
c: The carry flag.
z: The zero flag.
R0 to R7: Value of registers.
stop_flag: When the processor is executing instructions, the stop_flag is 0. When the processor is done executing, the stop_flag becomes 1.

Block Diagram and FSM

The block diagram for IITB Proc is as follows:

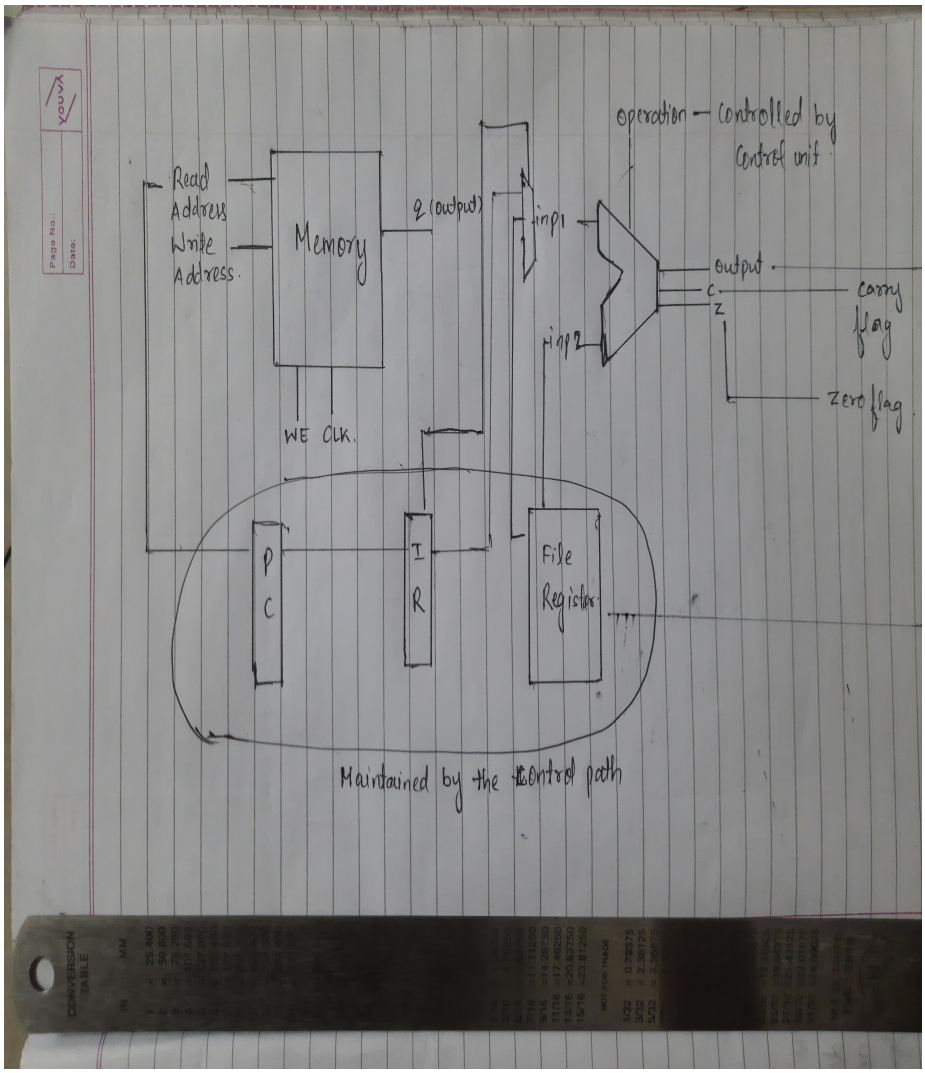
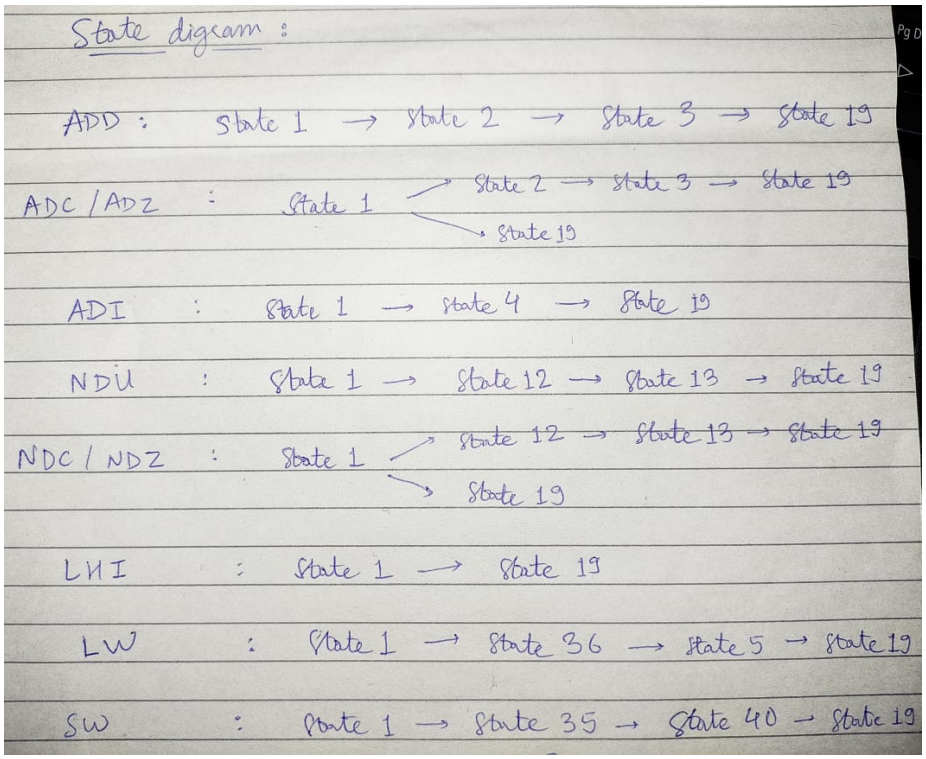
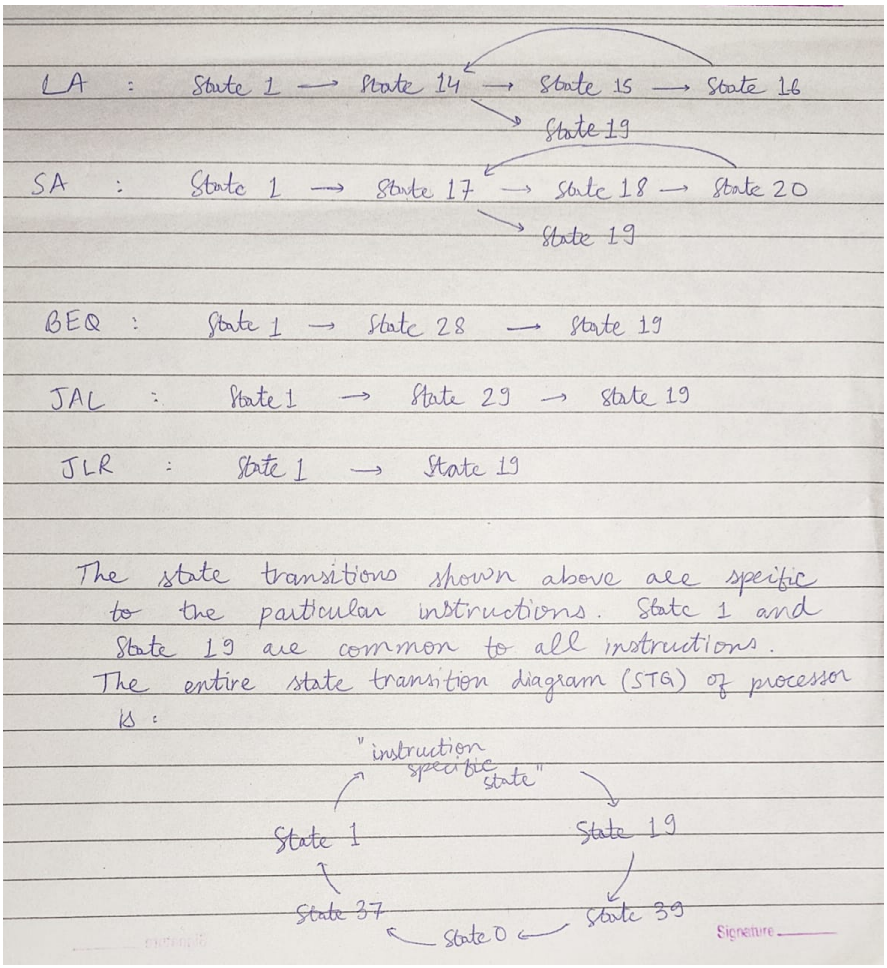


Figure 1: Block Diagram

State diagram explanation:

1. State 1: Read instruction from the memory address pointed to by the program counter and store the instruction in the Instruction register.
2. State 2: Decode and understand the instruction.
3. State 3: Follow the instruction FSM as demonstrated below:


State digram :
ADD : State 1 → State 2 → State 3 → State 19
ADC / ADZ : State 1 → State 2 → State 3 → State 19
State 19
ADI : State 1 → State 4 → State 19
NDU : State 1 → State 12 → State 13 → State 19
NDC / NDZ : State 1 → State 12 → State 13 → State 19
State 19
LHI : State 1 → State 19
LW : State 1 → State 36 → State 5 → State 19
SW : State 1 → State 35 → State 40 → State 19



4. State 4: Once the PC has been incremented, go back to state 1.

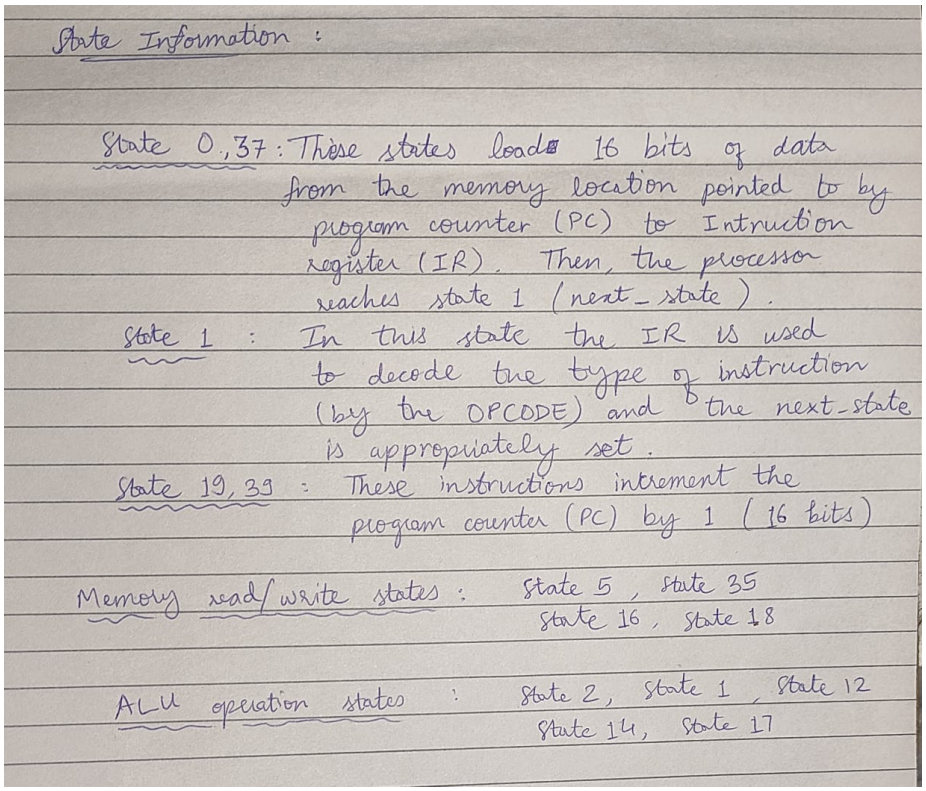
Note: We have used 1111000000000000 to denote the HLT instruction. Used to halt the execution.

Interaction between components

The top-level component `IITB_Proc` declares two sub-components of type `ALU` and `Memory`.
When the processor wants to read the data stored at a memory location, it sets the `read_address` pin on the `Memory` component to that memory location. After one clock cycle the processor can read the corresponding data at `q` pin of `Memory` component. Similarly, to write data to a memory location, the processor sets the `write_address`, `we` and `data` pins on the `Memory` component. And, at the next falling edge of the clock, provided data is written to memory.

The `ALU` componrnt is built to be able to perform *addition* and *nand*. The input pins `inp1`, `inp2` and `operation` are set by the processor to the first argument, second argument and, 1 for *addition* and 0 for *nand* respectively. The processor, then, reads the output of the operation at `outp` pin and sets its flags to the `c,z` pins of the `ALU` component.

Description of special states



Overall working and use:

Initially, the user has to set the start flag to 0 and the reset flag to 0.
Writing instructions or data to memory: To write instructions into the memory, the user has to set the writeF flag to 1 and navigate the memory by changing the input address in display_addr, and then changing the 16 bit input 'data'. After this, a gap of '1' clock cycle is needed for the instruction or data to be written into memory.
Starting the processor: The user has to set the input flag start to 1. This will start the processor. To stop the processor in between, the RST input flag can be set to 1.
Reading values after processing: After processing, the stop_flag will be set to 1. The final values of the registers can be seen in the outputs R0 to R7. To read from a memory address, set the value of display_addr to that memory address. After one clock cycle, the value of output_display will be the value in the memory address.

Testbench Explanation

The testbench included in the submission simulates a counter on the processor. We explain parts of the file `Testbench.vhd` below.

```
display_addr <= "0000000000000000";
data         <= "0011011000000001"; -- LHI R3 1
-- some clock cycles
display_addr <= "0000000000000001";
data         <= "0100000011000001"; -- LW R0 R3 1
-- some clock cycles
display_addr <= "0000000000000010";
data         <= "0001011011000001"; -- ADI R3 R3 1
-- some clock cycles
display_addr <= "0000000000000011";
data         <= "0100001011000001"; -- LW R1 R3 1
-- some clock cycles
display_addr <= "0000000000000100";
data         <= "0001001001000001"; -- ADI R1 R1 1
-- some clock cycles
display_addr <= "0000000000000101";
data         <= "1100001000000010"; -- BEQ R1 R0 2
-- some clock cycles
display_addr <= "0000000000000110";
data         <= "1000101111111110"; -- JAL R5 -2
-- some clock cycles
display_addr <= "0000000000000111";
data         <= "0101001011000010"; -- SW R1 R3
-- some clock cycles
```

```
display_addr <= "0000000000001000";
data         <= "1111000000000000"; -- HALT
-- some clock cycles
display_addr <= "0000000010000001";
data         <= "0000000000000101"; -- memory[129] = 5
-- some clock cycles
display_addr <= "0000000010000010";
data         <= "0000000000000000"; -- memory[130] = 0
```

We demonstrate out module by running a simple counter code. The register R0 stores the value 5 (demonstration Load instruction). R1 initially is loaded with a value of 0 and is incremented (demonstrating ADI instruction) till it becomes equal to R0 (demonstrating BEQ instruction). Till they are not equal, the PC goes back 2 steps (JAL instruction) and increments R1.
The processor starts reading instructions from memory location 0 and halts when it reads a `HALT` instruction (`opcode` = "1111"). the following instructions are executed in our testbench. The explanation below explains how this set of instructions simulates a counter on the processor.

- In our testbench, the first instruction loads 128 (0000000010000000) into R3.
- The second instruction, then, loads into register R0 the data stored at memory[128+1] = memory[129] = 5.
- Then, register R3 is incremented by 1 ,i.e. it stores 129.
- memory[129+1] = memory[130] = 0 is loaded into register R1. R1 is incremented by one. This register will act as the counter.
- The `BEQ` instruction compares the value of R1 with R0 (i.e. 5) and if they are equal, jumps to the `SW` instruction effectively breaking out of the counter loop.
- The `JAL` instruction loops back to the instruction which increments the value of register R1.
- The next instruction (`SW`) stores the value of in R1 to memory[129].
- The last (`HALT`) instruction stops the program execution.