IITB CPU

November 30, 2022

Abhishek Savaliya	21D070065
Simran Singh	21D070073
Jangid Nikita	21D070032
Lohitaksh Mahajan	21D070042

Contents

1 Overview of the Design

We started our design by first writing the states for every instruction without considering state minimization. After then we started to minimize the states. After effective minimization we had a total of 33 states with state 1 and 2 common to all. To minimize states we used different muxes(if else conditions). After state minimization we constructed individual components and then made pen paper design of our data path. Below is the diagram of our data path:

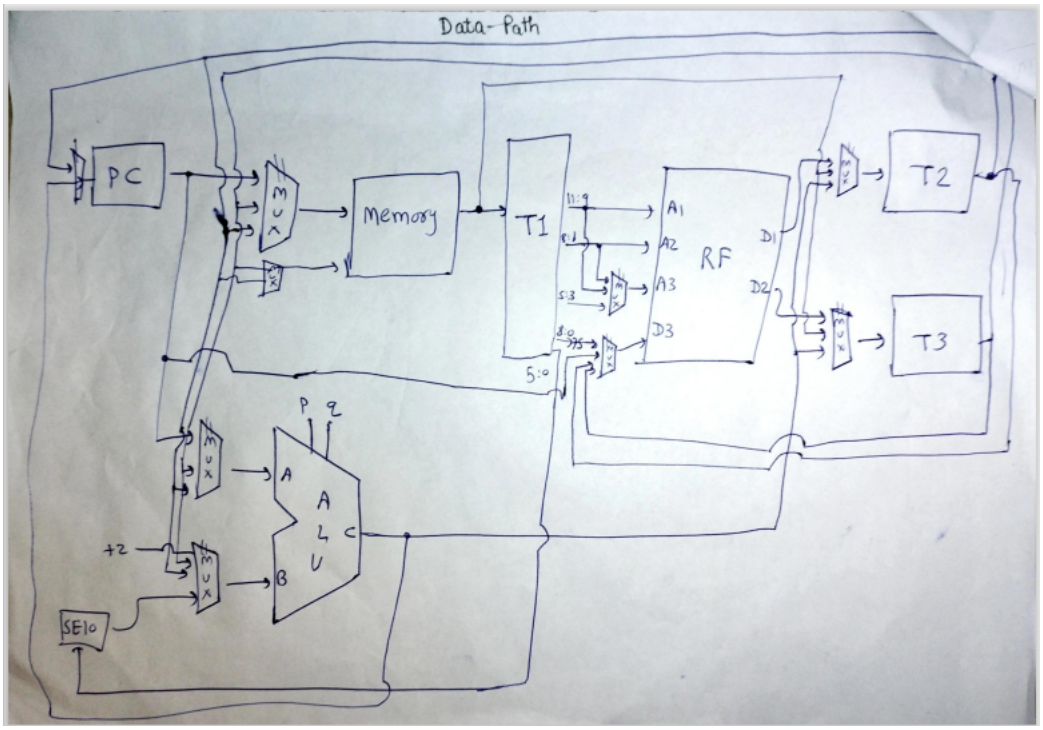

Figure 1: Pen Paper Design of Data Path

2 Working of Design

Our CPU works by moving the program counter 1 step ahead after every instruction. To stop the instructions we added an extra state *111111* which is an infinite state i.e. if it comes, it remains forever. Memory contains all the instructions and all the data to be fed into registers. Registers are

16 bit storage devices. We constructed 8 registers from R0 to R7. A FSM was made to guide the CPU through particular states. In each particular state; our ALU, PC, Memory and Temporary Registers had particular tasks to perform.

3 Component Memory

IITB-CPU is a 16-bit microprocessor having 64 bytes of data memory and 64 bytes of instruction memory. Instructions can be given to the processor using the memory.vhdl file in which the instructions can be directly fed into the instruction memory array.

```
18
19  architecture working of mem is
20      type mem_array is array (0 to 31 ) of std_logic_vector (15 downto 0);
21      signal mem_data: mem_array :=(
22      x"0000",x"0000", x"0000", x"0000",
23      x"0000",x"0000", x"0000", x"0000",
24      x"0000",x"0000", x"0000", x"0000",
25      x"0000",x"0000", x"0000", x"0000",
26      x"0000",x"0000", x"0000", x"0000",
27      x"0000",x"0000", x"0000", x"0000",
28      x"0000",x"0000", x"0000", x"0000",
29      x"0000",x"0000", x"0000", x"0000"
30      );
31
32
33      signal mem_ins: mem_array := (
34      b"0000000001010000",x"FFFF", x"FFFF", x"FFFF",
35      x"FFFF",x"FFFF", x"FFFF", x"0000",
36      x"0000",x"0000", x"0000", x"0000",
37      x"0000",x"0000", x"0000", x"0000",
38      x"0000",x"0000", x"0000", x"0000",
39      x"0000",x"0000", x"0000", x"0000",
40      x"0000",x"0000", x"0000", x"0000",
41      x"0000",x"0000", x"0000", x"0000"
42      );
43      begin
44      mem_action: process(clk)
45      begin
46      if (rising_edge(clk)) then
47      if(state="001011" and op_code= "0101") then ---s11 for sw
48       mem_data(to_integer(unsigned(t3_addr))) <= data_t2;
49      elsif(state="001011" and op_code= "0111") then ---s11 for sm
50          mem_data(to_integer(unsigned(t2_addr))) <= data_t3;
51      end if;
52      end if;
53      end process;
```

Figure 2: Component Memory

4 Component Register

We constructed 8 registers R0 to R7 each being 16 bit storage device. Registers can be accessed through register.vhdl file. Below image shows value 1

stored in register 0 and 1. R7 contains our PC.

```
 7  entity registers is
 8     port (reg_a1: in std_logic_vector(2 downto 0);
 9           reg_a2: in std_logic_vector(2 downto 0);
10           reg_a3: in std_logic_vector(2 downto 0);
11           t2: out std_logic_vector(15 downto 0);
12           t2_in: in std_logic_vector(15 downto 0);
13           t3: out std_logic_vector(15 downto 0);
14           t3_in: in std_logic_vector(15 downto 0);
15           shift7: in std_logic_vector(15 downto 0);
16           clk: in std_logic;
17           state: in std_logic_vector(5 downto 0);
18           pc: in std_logic_vector(15 downto 0);
19           pc_update: in std_logic_vector(15 downto 0)
20     );
21     end entity;
22
23  architecture working of registers is
24  type mem_array is array (0 to 7 ) of std_logic_vector (15 downto 0);
25  signal regs: mem_array :=(
26     x"0001",x"0001", x"0000", x"FFFF",
27     x"FFFF",x"FFFF", x"FFFF", x"FFFF"
28     );
29  begin
30  regs_read: process(reg_a1, reg_a2, state)
31  begin
32     if (state = "000010") then
33        t2 <= regs(to_integer(unsigned(reg_a1)));
34        t3 <= regs(to_integer(unsigned(reg_a2)));
35     elsif (state="011010") then
36        t3 <= regs(0);
37     elsif (state="011011") then
38        t3 <= regs(1);
39     elsif (state="011100") then
40        t3 <= regs(2);
41     elsif (state="011101") then
42        t3 <= regs(3);
43     elsif (state="011110") then
44        t3 <= regs(4);
```

Figure 3: Component Register

5 Final Design

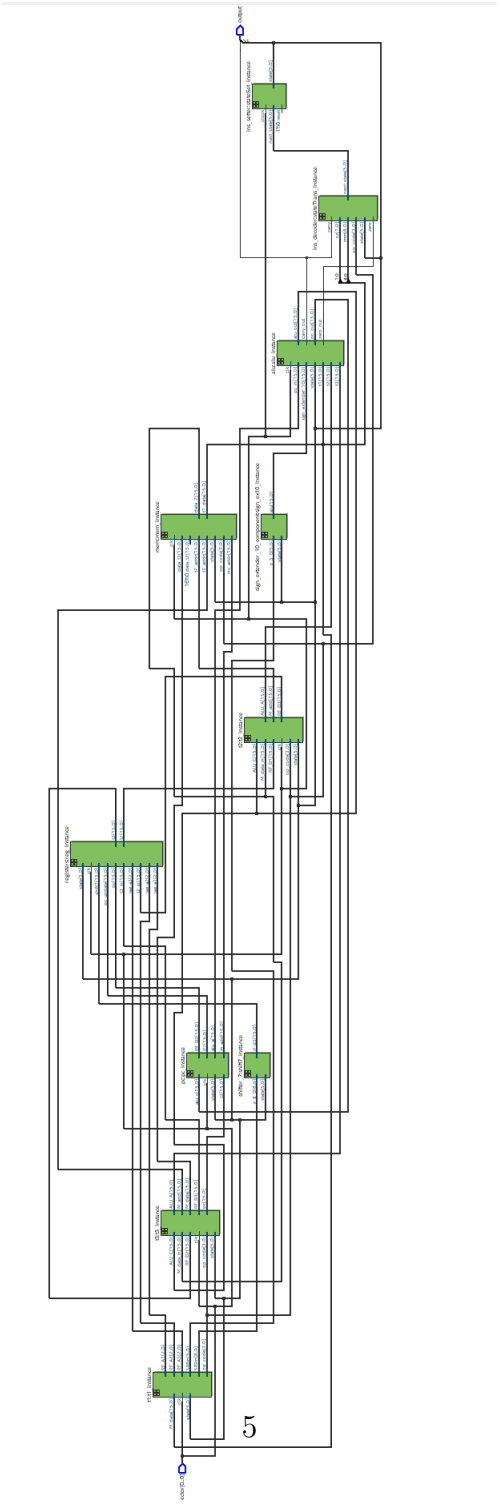

Figure 4: Netlist View

6 Running Instructions

Below images shows the add instruction in our CPU. State transition flows from S1 to S4 and the input of Register 0 and Register 1 being stored in Register 2. PC updates by 1 step.

Figure 5: RTL Wave

Figure 6: Register Data

7 Generating Fibonacci Sequence

Below is just a fun demo code run by our CPU. We generated a fibonacci sequence in our registers by giving following instructions as input.

```
signal mem_ins: mem_array := (
b"0000000001010000",b"0000001010011000", b"0000010011100000", b"0000011100101000",
b"0000100101110000",b"0000101110111000", x"FFFF", x"0000",
x"0000",x"0000", x"0000", x"0000",
x"0000",x"0000", x"0000", x"0000",
x"0000",x"0000", x"0000", x"0000",
x"0000",x"0000", x"0000", x"0000",
x"0000",x"0000", x"0000", x"0000",
x"0000",x"0000", x"0000", x"0000"
);
```

Figure 7: Fibonacci Input

```
00000000 0000000000000001 0000000000000001 0000000000000010 0000000000000011 0000000000000101 0000000000001000 0000000000001101 0000000000000111
00000009
```

Figure 8: Fibonacci Sequence