IITB-CPU Report

Harsh Shah - 210100063 Soham Inamdar - 210100149
Mayank Gupta - 210101002 Ayush Joshi - 210100036

November 30,2022

Contents

1 Overview of the experiment

1.1 Purpose

The objective of the project is to design a computing system, IITB-CPU which is a 16-bit computer developed for teaching purposes. The IITB-CPU is an 8-register, 16-bit computer system, i.e., it can process 16 bits at a time. It uses point-to-point communication infrastructure. Developing a CPU has many benefits, it can be widely used for processing large amounts of data. The CPU developed can be used to load data for processing and also store data for future usage.

1.2 Procedure of the experiment

CPU requires a lot of separate components for functioning efficiently. Analysing the components required, we started by designing different components like memory, ALU, SE-7, SE-10, registers and effectively a register file. Using the computations of all the components we made the state diagram for the finite state machine, defining all states to perform different sets of instructions. Our initial design consisted of 52 states, and we optimized it to reduce it to 16 states. Then we started to code all the components using behavourial modelling and linked them using data paths using structural modelling. This helped us to create the FSM on VHDL.

1.3 Organization of the report

The report contains the various entities, architectures of different components and FSM used. It contains the output to different set of instruction stored in the memory and screenshots of the waveform in RTL simulation. It contains the component diagram as seen in the RTL view.

2 Approach to the assignment

We started out by designing the dataflow model of our CPU for various operations. The dataflow model basically describes the transfer of data from one storage entity to another for all the required operations. The document containing the same can be found here.

Further, we proceeded by designing the FSM for our CPU. The state transition diagram of our FSM is as shown below.

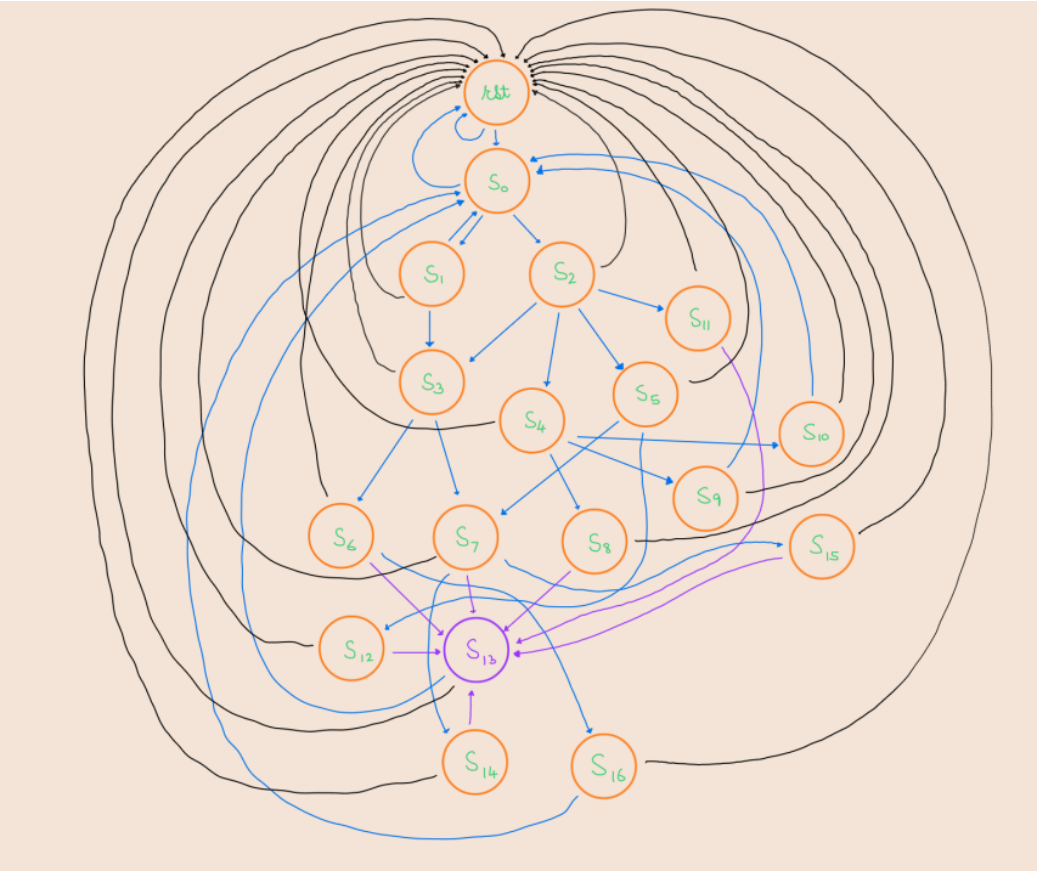

Figure 1: Finite State Machine

After designing the FSM for our CPU, we started with writing the behavioral description for the various components required for the CPU.

3 Components

3.1 Memory

The entity used for Memory is:

```
entity memory is
port (Mem_Addr, Mem_Data: in std_logic_vector(15 downto 0);
clk, rst, r_enable, w_enable: in std_logic;
Mem_Out: out std_logic_vector(15 downto 0));
end entity memory;
```

The architecture used for Memory is:

```
architecture behav of memory is
type locations is array(0 to 31) of std_logic_vector(15 downto 0);
-- Add instructions below

signal Mem : locations := ("0100001000000000","0100000000000000",
"0000000001000000", "0101000100000010", others => (others => '1'));

begin
process (clk, rst, r_enable, Mem_Addr, Mem_Data)
begin
if (rst = '1') then
Mem(to_integer(unsigned(Mem_Addr(4 downto 0)))) <= "0000000000000000";
elsif (clk'event and clk = '1') then
if (w_enable = '1') then
Mem(to_integer(unsigned(Mem_Addr(4 downto 0)))) <= Mem_Data;
end if;
end if ;

if (r_enable = '1') then
Mem_Out <= Mem(to_integer(unsigned(Mem_Addr(4 downto 0))));
end if;
end process;
end behav;
```

3.2 Register

The entity used for Register is:

```
entity reg is
port (D: in std_logic_vector(15 downto 0);
clk, rst, we: in std_logic;
Q: out std_logic_vector(15 downto 0));
end entity reg;
```

The architecture used for Register is:

```
architecture behav of reg is
begin
process (clk, rst)
begin
if (rst = '1') then
Q <= "0000000000000000";
elsif (clk'event and clk = '1') then
if (we = '1') then
Q <= D;
end if;
end if ;
end process;
end behav;
```

3.3 Demux 8:1

The entity used for Demux 8:1 is:

```
entity demux18 is
port (inp: in std_logic_vector(15 downto 0);
A: in std_logic_vector(2 downto 0);
outp1: out std_logic_vector(15 downto 0);
outp2: out std_logic_vector(15 downto 0);
outp3: out std_logic_vector(15 downto 0);
outp4: out std_logic_vector(15 downto 0);
outp5: out std_logic_vector(15 downto 0);
outp6: out std_logic_vector(15 downto 0);
outp7: out std_logic_vector(15 downto 0);
outp8: out std_logic_vector(15 downto 0));
end entity demux18;
```

The architecture used for Demux 8:1 is:

```
architecture behav of demux18 is
begin
process(inp, A)
begin
case A is
when "000" =>
outp1 <= inp;
when "001" =>
outp2 <= inp;
when "010" =>
outp3 <= inp;
when "011" =>
outp4 <= inp;
when "100" =>
outp5 <= inp;
when "101" =>
outp6 <= inp;
when "110" =>
outp7 <= inp;
```

```
when "111" =>
outp8 <= inp;
when others=>
end case;
end process;
end behav;
```

3.4 Mux 8:1

The entity used for Mux 8:1 is:

```
entity mux81 is
port (inp1: in std_logic_vector(15 downto 0);
inp2: in std_logic_vector(15 downto 0);
inp3: in std_logic_vector(15 downto 0);
inp4: in std_logic_vector(15 downto 0);
inp5: in std_logic_vector(15 downto 0);
inp6: in std_logic_vector(15 downto 0);
inp7: in std_logic_vector(15 downto 0);
inp8: in std_logic_vector(15 downto 0);
A: in std_logic_vector(2 downto 0);
outp: out std_logic_vector(15 downto 0));
end entity mux81;
```

The architecture used for Mux 8:1 is:

```
architecture behav of mux81 is
begin
process(inp1, inp2, inp3, inp4, inp5, inp6, inp7, inp8, A)
begin
case A is
when "000" =>
outp <= inp1;
when "001" =>
outp <= inp2;
when "010" =>
outp <= inp3;
when "011" =>
outp <= inp4;
when "100" =>
outp <= inp5;
when "101" =>
outp <= inp6;
when "110" =>
```

```
outp <= inp7;
when "111" =>
outp <= inp8;
when others=>
end case;
end process;
end behav;
```

3.5 Register File

The entity used for Register File is:

```
entity reg_file is
port (RF_A1, RF_A2, RF_A3: in std_logic_vector(2 downto 0);
RF_D3: in std_logic_vector(15 downto 0);
clk, rst, we: in std_logic;
RF_D1, RF_D2: out std_logic_vector(15 downto 0));
end entity reg_file;
```

The architecture used for Register is:

```
component reg is
port (D: in std_logic_vector(15 downto 0);
clk, rst, we: in std_logic;
Q: out std_logic_vector(15 downto 0));
end component reg;

component demux18 is
port (inp: in std_logic_vector(15 downto 0);
A: in std_logic_vector(2 downto 0);
outp1: out std_logic_vector(15 downto 0);
outp2: out std_logic_vector(15 downto 0);
outp3: out std_logic_vector(15 downto 0);
outp4: out std_logic_vector(15 downto 0);
outp5: out std_logic_vector(15 downto 0);
outp6: out std_logic_vector(15 downto 0);
outp7: out std_logic_vector(15 downto 0);
outp8: out std_logic_vector(15 downto 0));
end component demux18;

component mux81 is
port (inp1: in std_logic_vector(15 downto 0);
inp2: in std_logic_vector(15 downto 0);
inp3: in std_logic_vector(15 downto 0);
inp4: in std_logic_vector(15 downto 0);
inp5: in std_logic_vector(15 downto 0);
```

```
inp6: in std_logic_vector(15 downto 0);
inp7: in std_logic_vector(15 downto 0);
inp8: in std_logic_vector(15 downto 0);
A: in std_logic_vector(2 downto 0);
outp: out std_logic_vector(15 downto 0));
end component mux81;

type registers is array(0 to 7)
of std_logic_vector(15 downto 0);
signal ri, ro : registers;

begin
DMX: demux18 port map (inp => RF_D3, A => RF_A3,
     outp1 => ri(0), outp2 => ri(1),
     outp3 => ri(2), outp4 => ri(3),
     outp5 => ri(4), outp6 => ri(5),
     outp7 => ri(6), outp8 => ri(7));

L1: for i in 0 to 7 generate
R: reg port map (D => ri(i), clk => clk, rst => rst, we => we,
    Q => ro(i));
end generate;

MX1: mux81 port map (inp1 => ro(0),
    inp2 => ro(1), inp3 => ro(2),
    inp4 => ro(3), inp5 => ro(4),
    inp6 => ro(5), inp7 => ro(6),
    inp8 => ro(7), A => RF_A1,
    outp => RF_D1);

MX2: mux81 port map (inp1 => ro(0),
    inp2 => ro(1), inp3 => ro(2),
    inp4 => ro(3), inp5 => ro(4),
    inp6 => ro(5), inp7 => ro(6),
    inp8 => ro(7), A => RF_A2,
    outp => RF_D2);
end arch;
```

3.6 Sign Extension 7

The entity used for Sign Extension 7 is:

```
entity Sign_Extension_7 is
port(ip: in std_logic_vector(8 downto 0);
    op: out std_logic_vector(15 downto 0));
end entity;
```

The architecture used for Sign Extension 7 is:

```
architecture Struct of Sign_Extension_7 is
begin
op(15 downto 7) <= ip(8 downto 0);
op(6 downto 0) <= "0000000";

end Struct;
```

3.7 Sign Extension 10

The entity used for Sign Extension 10 is:

```
entity Sign_Extension_10 is
port(ip: in std_logic_vector(5 downto 0);
    op: out std_logic_vector(15 downto 0));
end entity;
```

The architecture used for Sign Extension 10 is:

```
architecture Struct of Sign_Extension_10 is
begin
op(5 downto 0) <= ip(5 downto 0);
op(15 downto 6) <= "0000000000";
end Struct;
```

3.8 CPU

The above components were used to make the CPU. The entity used for the CPU is:

```
entity cpu is
generic(operand_width : integer := 16);
port(clock, reset: in std_logic;
  output: out std_logic);
end entity cpu;
```

The architecture used for the CPU is:

```
architecture bhv of cpu is

--Memory
component memory is
port (Mem_Addr, Mem_Data: in std_logic_vector(operand_width-1 downto 0);
clk, rst, r_enable, w_enable: in std_logic;
Mem_Out: out std_logic_vector(operand_width-1 downto 0));
end component memory;

--Register file
component reg_file is
port (RF_A1, RF_A2, RF_A3: in std_logic_vector(2 downto 0);
RF_D3: in std_logic_vector(operand_width-1 downto 0);
clk, rst, we: in std_logic;
RF_D1, RF_D2: out std_logic_vector(operand_width-1 downto 0));
end component reg_file;

--Register
component reg is
port (D: in std_logic_vector(operand_width-1 downto 0);
clk, rst, we: in std_logic;
Q: out std_logic_vector(operand_width-1 downto 0));
end component reg;

--DeMux
```

```
component demux18 is
port (inp: in std_logic_vector(operand_width-1 downto 0);
A: in std_logic_vector(2 downto 0);
outp1: out std_logic_vector(operand_width-1 downto 0);
outp2: out std_logic_vector(operand_width-1 downto 0);
outp3: out std_logic_vector(operand_width-1 downto 0);
outp4: out std_logic_vector(operand_width-1 downto 0);
outp5: out std_logic_vector(operand_width-1 downto 0);
outp6: out std_logic_vector(operand_width-1 downto 0);
outp7: out std_logic_vector(operand_width-1 downto 0);
outp8: out std_logic_vector(operand_width-1 downto 0));
end component demux18;

--ALU
--00:ADD
--01:SUBTRACT
--10:NAND
--11:SHIFTER_LEFT_7
component ALU is
generic(operand_width : integer:=16);
port(ALU_op: in std_logic_vector(1 downto 0);
  ALU_A: in std_logic_vector(operand_width-1 downto 0);
  ALU_B: in std_logic_vector(operand_width-1 downto 0);
  ALU_C: out std_logic_vector(operand_width-1 downto 0);
  ALU_zero: out std_logic;
  ALU_carry: out std_logic);
end component ALU;

component Sign_Extension_10 is
port(ip: in std_logic_vector(operand_width-11 downto 0);
 op: out std_logic_vector(operand_width-1 downto 0));
 end component;

component Sign_Extension_7 is
port(ip: in std_logic_vector(operand_width-8 downto 0);
 op: out std_logic_vector(operand_width-1 downto 0));
end component;
```

```
---------------Define state type here-----------------------------
type state is (rst, s0, s1, s2, s3, s4, s5, s6, s7, s8,s9, s10,
 s11, s12, s13, s14, s15, s16); -- Fill other states here
---------------Define signals of state type-----------------------
signal y_present1, y_next1 : state := rst;

--state transition control signal
signal stcon: std_logic_vector(3 downto 0);
signal read_1, read_2, write_1: std_logic_vector(2 downto 0):="000";

--sign extender
signal SE10_inp: std_logic_vector(operand_width-11 downto 0);
signal SE7_inp: std_logic_vector(operand_width-8 downto 0);

signal mem: std_logic_vector(4 downto 0);
signal running : std_logic := '1';
signal reg_reset, reg_we, mem_re, mem_reset, mem_we, t1_reset,
 t1_we,t2_reset, t2_we,t3_reset, t3_we, alu_zflag, alu_cflag,
 z_flag, c_flag: std_logic := '0';
signal PC, mem_inp, mem_read, mem_store, outp_1, outp_2, inp_1,
 t1_inp, t1_outp,t2_inp, t2_outp,t3_inp, t3_outp, alu_inp1,
 alu_inp2, alu_outp1, SE10_outp, SE7_outp:
 std_logic_vector(operand_width-1 downto 0):="0000000000000000";
signal alu_op: std_logic_vector(1 downto 0);

begin

RF1: reg_file port map (RF_A1=>read_1, RF_A2=>read_2,
RF_A3=>write_1, RF_D3=>inp_1, clk=>clock, rst=>reg_reset,
we=>reg_we, RF_D1=>outp_1, RF_D2=>outp_2);

Mem1: memory port map (Mem_Addr=>mem_inp, Mem_Data=>mem_store,
clk=> clock, rst=>mem_reset, r_enable=>mem_re, w_enable=>mem_we,
 Mem_Out=> mem_read);

T1: reg port map (D=>t1_inp,
clk=> clock, rst=>t1_reset, we=>t1_we,
```

```
Q=>t1_outp);
T2: reg port map (D=>t2_inp,
clk=> clock, rst=>t2_reset, we=>t2_we,
Q=>t2_outp);
T3: reg port map (D=>t3_inp,
clk=> clock, rst=>t3_reset, we=>t3_we,
Q=>t3_outp);
SE10: Sign_Extension_10 port map (ip => SE10_inp, op => SE10_outp);
SE7: Sign_Extension_7 port map (ip => SE7_inp, op => SE7_outp);

alu1: ALU port map (ALU_op=>alu_op,
  ALU_A=>alu_inp1,
  ALU_B=>alu_inp2,
  ALU_C=>alu_outp1,
  ALU_zero=>alu_zflag,
  ALU_carry=>alu_cflag);

output <= running;

clock_proc:process(clock,reset)
begin
if(clock = '1' and clock' event) then
if(reset = '1') then
y_present1 <= rst;
else
y_present1 <= y_next1;
end if;
end if;
end process clock_proc;

--State transition process
state_trans_proc:process(y_present1)
begin
case y_present1 is
when rst =>
--Initialising the register file and obtaing the value of mem_inp from R7
reg_reset <= '1';
```

```
if (reset = '1') then
y_next1 <= rst;
else
y_next1 <= s0;
end if;

when s0 =>
-- Reading PC from Register File
read_1 <= "111";
PC <= outp_1;

-- Reading data from memory at PC
mem_re <= '1';
mem_inp <= PC;
t1_we <= '1';
t1_inp <= mem_read;

stcon <= mem_read(operand_width-1 downto operand_width-4);

if (reset = '1') then
y_next1 <= rst;

elsif(stcon(3 downto 2) = "00" and stcon(0) = '0') then
--ADD, ADC, ADZ, NDU, NDC, NDZ
y_next1 <= s1;

else
--ADI, LHI, LW, LM, SM, SW, BEQ, JAL, JLR
y_next1 <= s2;

end if;

when s1 =>
--PC update
alu_inp1 <= PC;
alu_inp2 <= "0000000000000001";
alu_op <= "00";
PC <= alu_outp1;
```

```
--For storing value of PC in Reg file 7
reg_we <= '1';
write_1 <= "111";
inp_1 <= alu_outp1;

if (reset = '1') then
y_next1 <= rst;
end if;

if (((z_flag = '0') and (stcon(0) = '1')) or ((c_flag = '0')
and (stcon(1) = '1'))) then
--If flags not 1, go to initial state s0
y_next1 <= s0;

else
--ADD, ADC, ADZ, NDU, NDC, NDZ
y_next1 <= s3;

end if;

when s2 =>
--PC update
alu_inp1 <= PC;
alu_inp2 <= "0000000000000001";
alu_op <= "00";
PC <= alu_outp1;

--For storing value of PC in Reg file 7
reg_we <= '1';
write_1 <= "111";
inp_1 <= alu_outp1;

--if conditions
if (reset = '1') then
y_next1 <= rst;

elsif (stcon = "0101" or stcon = "1100") then
```

```
--SW, BEQ
y_next1 <= s3;

elsif (stcon = "0001" or stcon = "0011" or stcon = "0111"
or stcon = "0110" ) then
--ADI, LHI, SM, LM
y_next1 <= s4;

elsif (stcon = "0100" or stcon = "1001") then
--JLR, LW
y_next1 <= s5;

else
--JAL
y_next1 <= s11;

end if;

when s3 =>
--Read RF_A1, RF_A2 and write to T2, T3 respectively
read_1 <= t1_outp(11 downto 9);
read_2 <= t1_outp(8 downto 6);
t2_we <= '1';
t3_we <= '1';
t2_inp <= outp_1;
t3_inp <= outp_2;

if (reset = '1') then
y_next1 <= rst;

elsif (stcon = "0000" or stcon = "0010" or stcon = "1100") then
--ADD, ADC, ADZ, NDU, NDC, NDZ, BEQ
y_next1 <= s6;

elsif (stcon = "0101") then
--SW
y_next1 <= s7;
```

```
end if;

when s4 =>
--Read RF_A1 and write to T2
read_1 <= t1_outp(11 downto 9);
t2_we <= '1';
t2_inp <= outp_1;

if (reset = '1') then
y_next1<=rst;

elsif (stcon = "0001") then
--ADI
y_next1 <= s7;

elsif (stcon = "0011") then
--LHI
y_next1 <= s8;

elsif (stcon = "0110") then
--LM
y_next1 <= s9;

elsif (stcon = "0111") then
--SM
y_next1 <= s10;

end if;

when s5 =>
--Read RF_A1 and write to T2
read_1 <= t1_outp(8 downto 6);
t2_we <= '1';
t2_inp <= outp_1;

if (reset = '1') then
y_next1 <= rst;
```

```
elsif (stcon = "0100") then
--LW
y_next1 <= s7;

elsif (stcon = "1001") then
--JLR
y_next1 <= s12;

end if;


when s6 =>
--Execution
--Feed T2,T3 to ALU
alu_inp1 <= t2_outp;
alu_inp2 <= t3_outp;

--00 for ADD, 01 for SUBTRACT, 10 for NAND
alu_op(1) <= stcon(1);
alu_op(0) <= stcon(3);
t2_we <= '1';
t2_inp <= alu_outp1;

if t1_outp(operand_width-1) = '0' then
--Set zero flag
z_flag <= alu_zflag;
end if;

if stcon = "0000" then
--Set carry flag
c_flag <= alu_cflag;
end if;

if (reset = '1') then
y_next1 <= rst;

elsif (stcon = "1100") then
--BEQ
```

```
y_next1 <= s16;

else
--ADD, ADC, ADZ, NDU, NDC, NDZ
y_next1 <= s13;

end if;

when s7 =>
--Execution
alu_inp1 <= t2_outp;
SE10_inp <= t1_outp(5 downto 0);
alu_inp2 <= SE10_outp;

--ALU_ADD
alu_op <= "00";
t2_we <= '1';
t2_inp <= alu_outp1;

if(t1_outp(14) = '1' xor t1_outp(12) = '1') then
z_flag <= alu_zflag;
end if;

if(t1_outp(14) = '0' and t1_outp(12) = '1') then
c_flag <= alu_cflag;
end if;

if (reset = '1') then
y_next1 <= rst;

elsif (stcon = "0001") then
--ADI
y_next1 <= s13;

elsif (stcon = "0100") then
--LW
y_next1 <= s14;
```

```
elsif (stcon = "0101") then
--SW
y_next1 <= s15;

end if;

when s8 =>
--Execution
SE7_inp <= t1_outp(8 downto 0);
t2_inp <= SE7_outp;
alu_inp1 <= SE7_outp;

--ALU_LSHIFT_7
alu_op <= "11";
t2_we <= '1';
t2_inp <= alu_outp1;

if (reset = '1') then
y_next1 <= rst;

elsif (stcon = "0011") then
--LHI
y_next1 <= s13;

end if;

when s9 =>
--Used for loop to check which bit is 1. If the bit is 1,
--we need to load the memory of address stored in T2.
--If for loop isn't possible to fabricate on the FPGA board, then
32 states will have to be defined
L1: for i in 0 to 7 loop
if (t1_outp(i) = '1') then
-- Reading the data stored in memory at t2_outp
mem_inp <= t2_outp;
t3_we <= '1';
t3_inp <= mem_read;
mem_we <= '0';
```

```
mem_re <= '1';

-- Writing the data extracted from memory to 'i'th register
write_1 <= std_logic_vector(to_unsigned(i, write_1'length));
reg_we <= '1';
inp_1 <= t3_outp;

-- Updating T2 to point to next address in memory
alu_inp1 <= t2_outp;
alu_inp2 <= "0000000000000001";
alu_op <= "00";
t2_we <= '1';
t2_inp <= alu_outp1;
end if;
end loop L1;

if (reset='1') then
y_next1 <= rst;
else
--Back to S0
y_next1 <= s0;
end if;


when s10 =>
--Used for loop to check which bit is 1. If the bit is 1,
--we need to store the value of R(i) in the memory of address stored in T2.
--If for loop isn't possible to fabricate on the FPGA board, then 32 states will h
L2: for i in 0 to 7 loop
if (t1_outp(i) = '1') then
-- Reading the data stored in 'i'th register
read_1 <= std_logic_vector(to_unsigned(i, write_1'length));
t3_we <= '1';
t3_inp <= outp_1;

-- Storing the data in memory
mem_we <= '1';
mem_re <= '0';
```

```
mem_inp <= t2_outp;
mem_store <= t3_outp;

-- Updating T2 to point to next address in memory
alu_inp1 <= t2_outp;
alu_inp2 <= "0000000000000001";
alu_op <= "00";
t2_we <= '1';
t2_inp <= alu_outp1;
end if;
end loop L2;

if (reset='1') then
y_next1 <= rst;
else
--Back to S0
y_next1 <= s0;
end if;

when s11 =>
--Extracting values of PC
read_1 <= "111";
PC <= outp_1;
   alu_inp1 <= PC;
SE7_inp <= t1_outp(8 downto 0);
alu_inp2 <= SE7_outp;
PC <= alu_outp1;

--For storing value of PC in Reg file 7
reg_we <= '1';
write_1 <= "111";
inp_1 <= PC;

if (reset='1') then
y_next1 <= rst;
else
--JAL
y_next1 <= s13;
```

```
end if;

when s12 =>
--Extracting values of PC
read_1 <= "111";
PC <= outp_1;
   alu_inp1 <= PC;
alu_inp2 <= t2_outp;
PC <= alu_outp1;

--For storing value of PC in Reg file 7
reg_we <= '1';
write_1 <= "111";
inp_1 <= PC;

if (reset = '1') then
y_next1 <= rst;
else
--JLR
y_next1 <= s13;
end if;

when s13 =>
--Store in RF_A3
reg_we <= '1';

if (t1_outp(15 downto 14) = "00" and t1_outp(12) = '1') then
-- ADD, ADC, ADZ, NDU, NDC, NDZ
write_1 <= t1_outp(5 downto 3);

elsif (t1_outp(15 downto 12) = "0001") then
-- ADI
write_1 <= t1_outp(8 downto 6);

else
-- LHI, LW, SW, JAL, JLR
write_1 <= t1_outp(11 downto 9);
```

```
end if;

if (t1_outp(15 downto 13) = "100") then
-- JAL, JLR
inp_1 <= PC;

else
-- ADD, ADC, ADZ, NDU, NDC, NDZ, ADI, LHI, LW, SW
inp_1 <= t2_outp;

end if;

if (reset = '1') then
y_next1 <= rst;
else
--Back to S0
y_next1 <= s0;
end if;

when s14 =>
mem_re <= '1';
mem_inp <= t2_outp;
t2_we <= '1';
t2_inp <= mem_read;

if (reset='1') then
y_next1 <= rst;
else
y_next1 <= s13;
end if;

when s15 =>
mem_we <= '1';
mem_inp <= t2_outp;
mem_store <= t3_outp;

if (reset = '1') then
y_next1 <= rst;
```

```
else
y_next1 <= s13;
end if;

when s16 =>
if(z_flag = '1') then
-- Extracting values of PC
read_1 <= "111";
PC <= outp_1;

-- Decreasing PC by 1 to get back the address of BEQ
alu_inp1 <= PC;
alu_inp2 <= "0000000000000001";
alu_op(1) <= stcon(1);
alu_op(0) <= stcon(3);

-- Sign Extending the Immediate bits by 10 bits and adding to PC
SE10_inp <= t1_outp(5 downto 0);
alu_inp1 <= SE10_outp;
alu_inp2 <= alu_outp1;
alu_op <= "00";
PC <= alu_outp1;

-- Storing value of PC in Reg file 7
reg_we <= '1';
write_1 <= "111";
inp_1 <= PC;
end if;

-- State Transition
if (reset = '1') then
y_next1 <= rst;
else
--Back to S0
y_next1 <= s0;
end if;

when others =>
```

```
end case;

end process state_trans_proc;
end bhv;
```

4 Observations

It was observed that the CPU efficiently uses all the components to process the given instructions. The states were made using the state diagram. The states were executed according to the state diagram.
Waveform simulation in RTL for CPU:

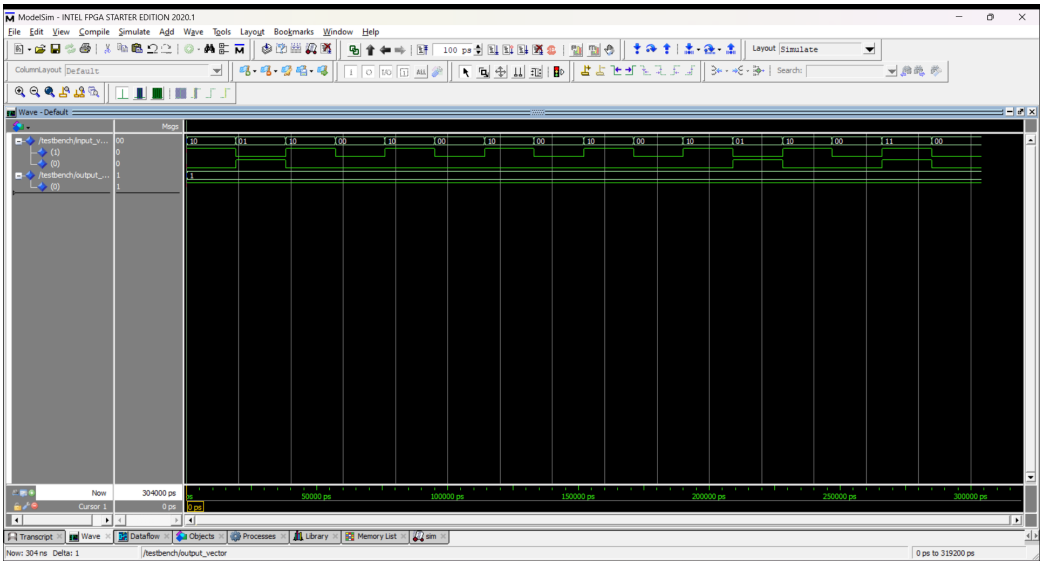

Figure 2: CPU RTL Waveform

RTL viewer for CPU:

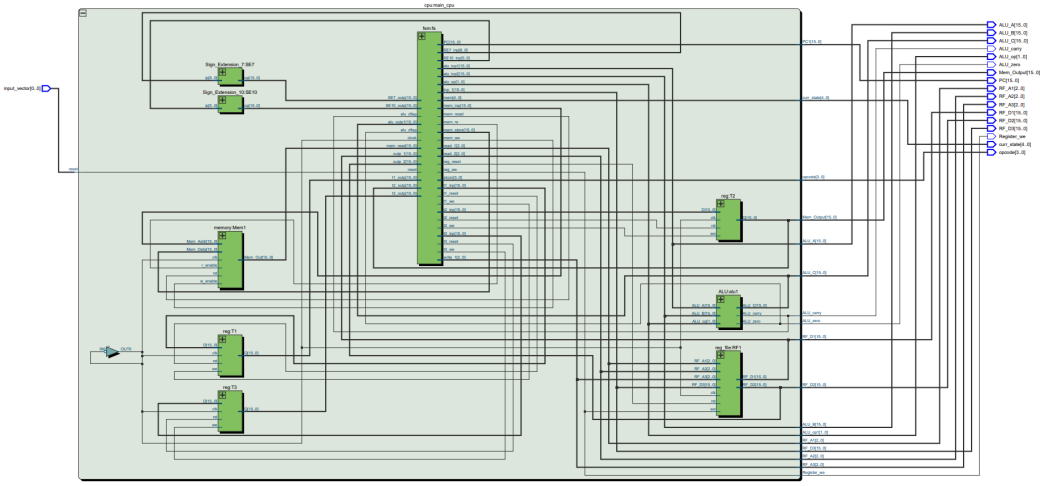

Figure 3: CPU RTL viewer