IITB-CPU: A 16 bit CPU

Project report for *Indian Institute of Technology, Bombay*
EE224: Digital Systems

Instructor: Prof. Virendra Singh

Abhiruchi Kotamkar	Arya Vishe	Shlesh Gholap	Shobhit Maheshwari
210070043	21d070018	210070080	210070081

November 30, 2022

Contents

1 Introduction

The IITB-CPU implements 15 instructions and utilises 8 programmer registers, of which register 7 ($R7$) is used as a program counter or an instruction register. Internally, 15 states are used to implement these instructions and are described in the following section.

2 States

2.1 Description

The states and a short description of each is given below:

State 1	Read from memory and increment PC
State 2	Read R type instruction
State 3	Perform ALU-specific operation
State 4	Update programmer registers
State 5	Read ADI type instruction
State 6	Read LW, SW type instruction
State 7	Read from memory
State 8	Store into memory
State 9	Read LHI type instruction
State 10	Branch to PC+Immediate
State 11	Store PC in Register A
State 12	Branch to Register B
State 13	Load from memory to register
State 14	Update input to priority encoder
State 15	Load from register to memory

2.2 ADD and NAND

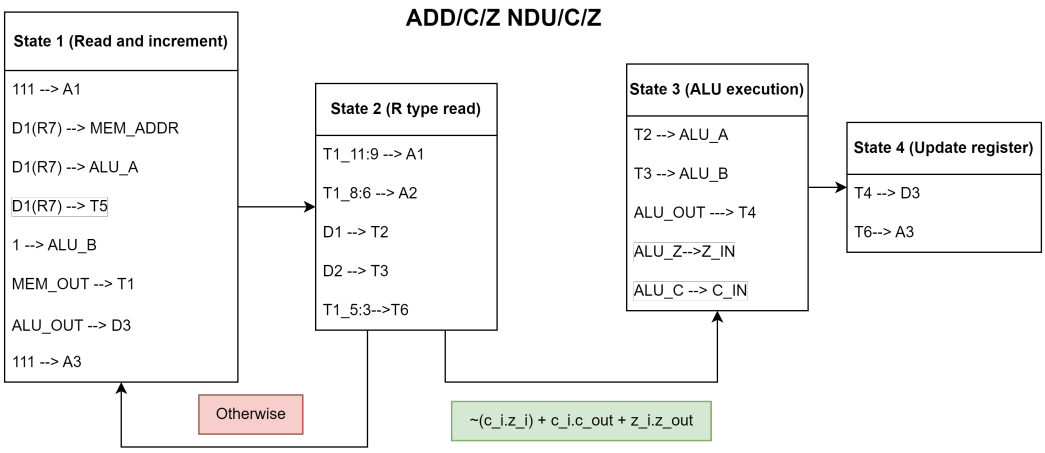


Figure 1: ADD/C/Z and NDU/C/Z instruction state-flow

2.3 ADI

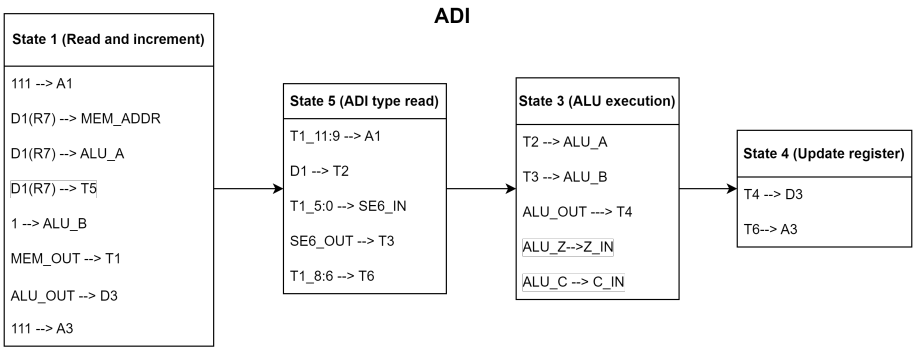


Figure 2: ADI instruction state-flow

2.4 BEQ

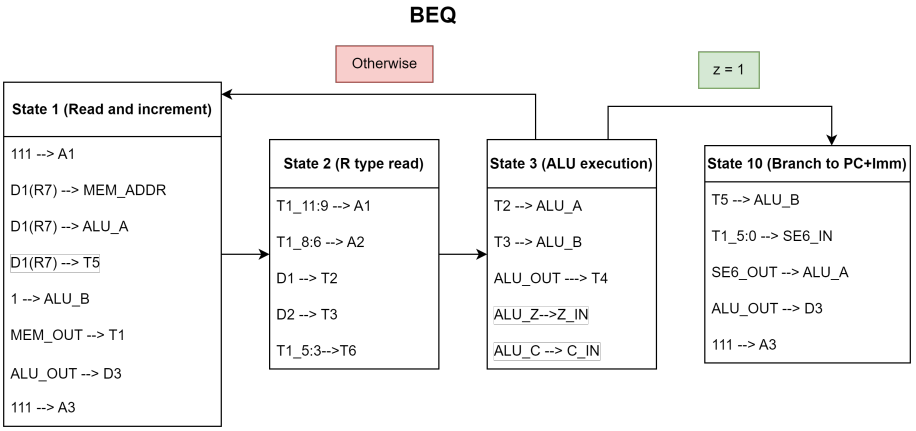
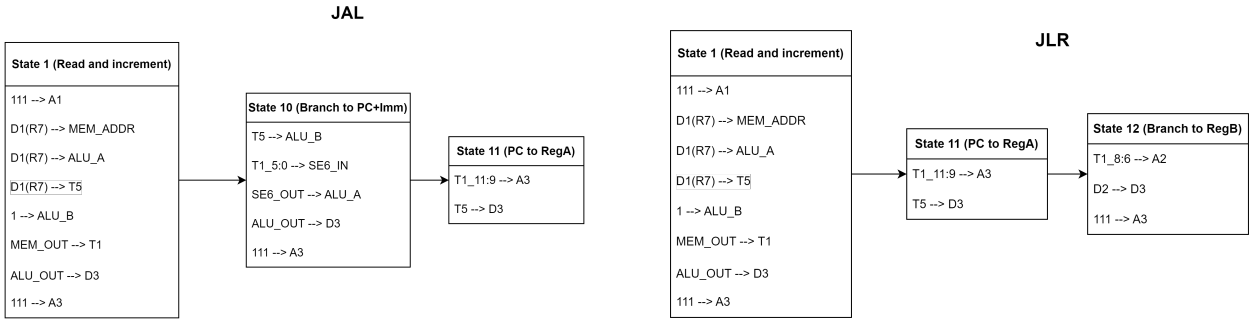


Figure 3: BEQ instruction state-flow

2.5 JAL and JLR



(a) JAL instruction state-flow

(b) JLR instruction state-flow

Figure 4: Combined state-flows of JAL and JLR

2.6 LHI

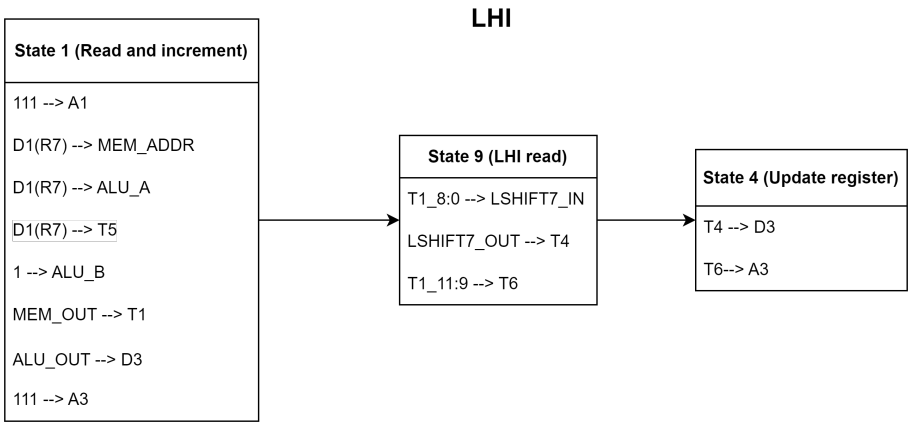
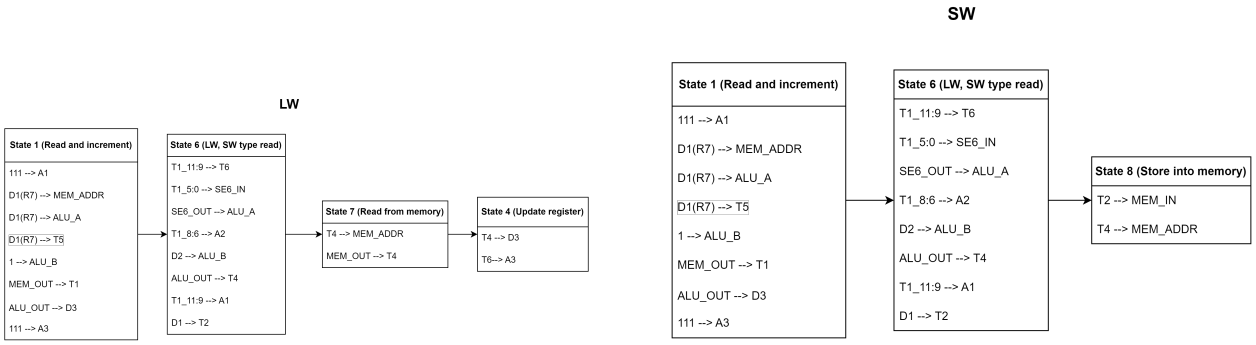


Figure 5: LHI instruction state-flow

2.7 LW and SW

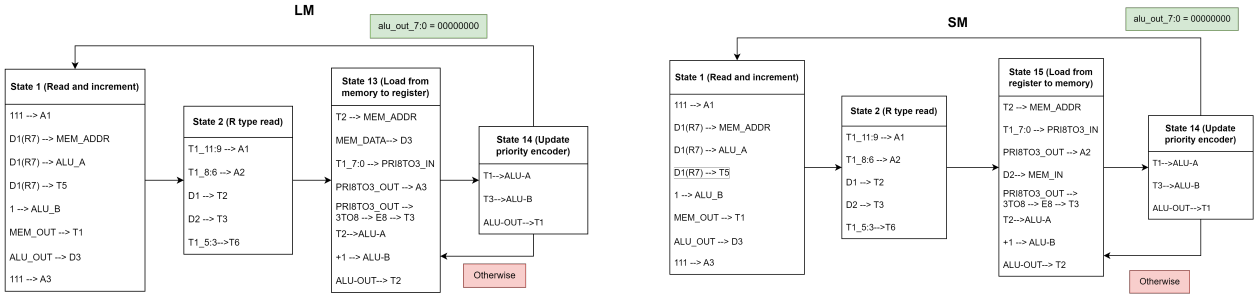


(a) LW instruction state-flow

(b) SW instruction state-flow

Figure 6: Combined state-flows of LW and SW

2.8 LM and SM



(a) LM instruction state-flow

(b) SM instruction state-flow

Figure 7: Combined state-flows of LM and SM

3 Transition Logic and Control Signals

Using behavioural modelling, state transition can be facilitated based on the values of:

1. Op. Code (4 most significant bits of instruction)
2. C and Z priority bits (2 least significant bits of instruction)
3. C and Z flag (contents of flag register)

The following control signals had to be defined:

- Memory read enable (M_RD)
- Memory write enable (M_WR)
- Programmer register write enable (PR_WR)
- Temporary register 1 write enable (T1_WR)
- Temporary register 2 write enable (T2_WR)
- Temporary register 3 write enable (T3_WR)
- Temporary register 4 write enable (T4_WR)
- Temporary register 5 write enable (T5_WR)
- Temporary register 6 write enable (T6_WR)
- Z flag write enable (Z_EN)
- C flag write enable (C_EN)
- ALU select signal (ALU_SELECT_0 and ALU_SELECT_1)

The MUX select signals have been defined by accounting for all possible choices in each state.

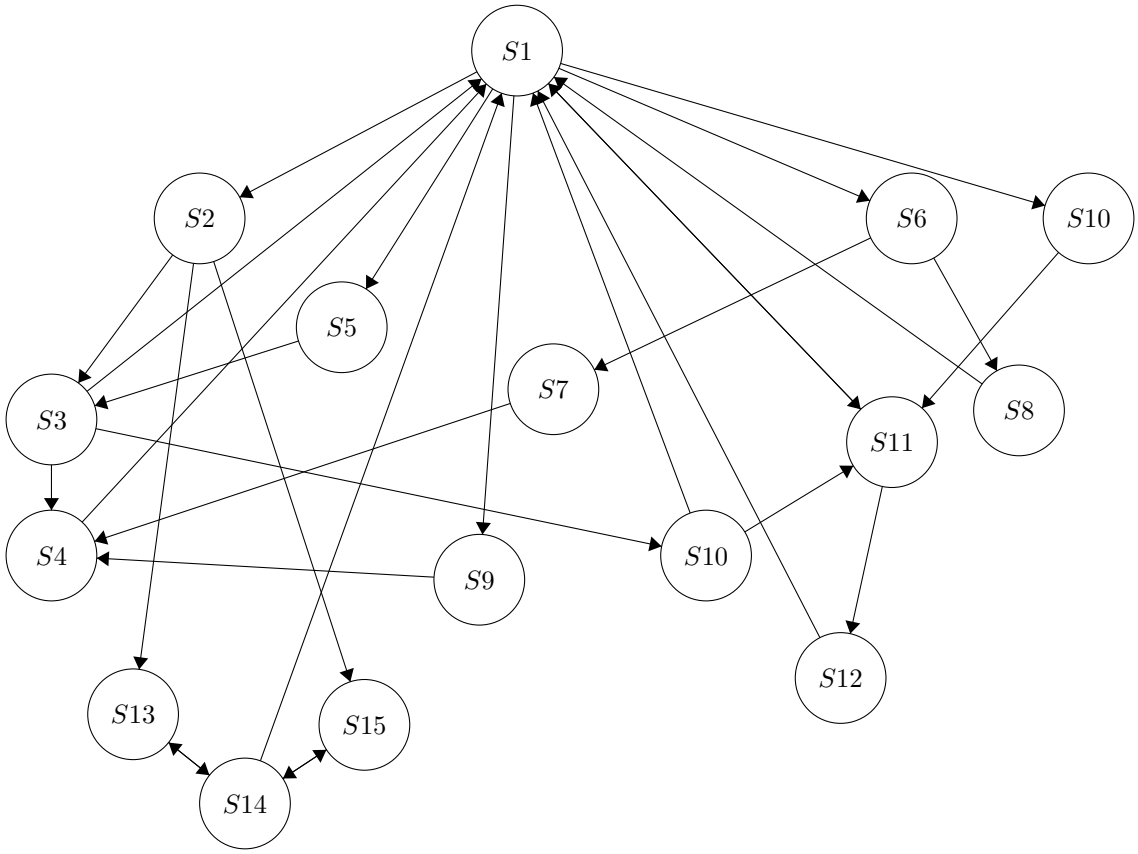
State	PR_WR	M_WR	M_RD	T1_WR	T2_WR	T3_WR	T4_WR	T5_WR	T6_WR	Z_EN	C_EN	ALU1	ALU0
1	1	0	1	1	0	0	0	1	0	0	0	0	0
2	0	0	0	0	1	1	0	0	1	0	0	0	0
3	0	0	0	0	0	0	1	0	0	(!op[3])*(!op[2])*(!(op[1]*op[0]))	(!op[3])*(!op[2])*(!op[1])	!op[3]*!op[2]*op[1]*!op[0]	op[3]*op[2]
4	1	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	1	1	0	0	1	0	0	0	0
6	0	0	0	0	1	0	1	0	1	!op[3]*op[2]*!op[1]*!op[0]	0	0	0
7	0	0	1	0	0	0	1	0	0	0	0	0	0
8	0	1	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	1	0	1	0	0	0	0
10	1	0	0	0	0	0	0	0	0	0	0	0	0
11	1	0	0	0	0	0	0	0	0	0	0	0	0
12	1	0	0	0	0	0	0	0	0	0	0	0	0
13	1	0	1	0	1	1	0	0	0	0	0	0	0
14	0	0	0	1	0	0	0	0	0	0	0	0	1
15	0	1	0	0	1	1	0	0	0	0	0	0	0

Table 2: State-wise control signals

Present State	Next State	Condition
State 1	State 2	!(op[3]) * !(op[2]) * !(op[0]) + (op[2] * op[1]) + (op[3] * op[2])
	State 5	!(op[3]) * !(op[2]) * !(op[1]) * op[0]
	State 6	!(op[3]) * op[2] * !(op[1])
	State 9	!(op[3]) * !(op[2]) * op[1] * op[0]
	State 10	op[3] * !(op[2]) * !(op[1]) * !(op[0])
	State 11	op[3] * !(op[2]) * !(op[1]) * op[0]
State 2	State 1	!(op[3]) * !(op[2]) * !(op[0])) * !(!(c_i *z_i) + c_i * c_f + z_i * z_f))
	State 3	(!(op[3]) * !(op[2]) * !(op[0])) * (!(c_i *z_i) + c_i * c_f + z_i * z_f)) + (op[3] * op[2])
	State 13	!(op[3]) * op[2] * op[1] * !(op[0])
	State 15	!(op[3]) * op[2] * op[1] * op[0]
State 3	State 1	op[3] * op[2] * !(z_f)
	State 4	!(op[3]) * !(op[2]) * !(op[1] * op[0])
	State 10	op[3] * op[2] * z_f
State 4	State 1	Unconditional
State 5	State 3	Unconditional
State 6	State 7	!(op[3]) * op[2] * !(op[1]) * !(op[0])
	State 8	!(op[3]) * op[2] * !(op[1]) * op[0]
State 7	State 4	Unconditional
State 8	State 1	Unconditional
State 9	State 4	Unconditional
State 10	State 1	op[3] * op[2]
	State 11	op[3] * !(op[2]) * !(op[1]) * !(op[0])
State 11	State 1	op[3] * !(op[2]) * !(op[1]) * !(op[0])
	State 12	op[3] * !(op[2]) * !(op[1]) * op[0]
State 12	State 1	Unconditional
State 13	State 14	Unconditional
State 14	State 1	T1_7:0 == 00000000
	State 13	(T1_7:0 != 00000000) * (!(op[3]) * op[2] * op[1] * !(op[0]))
	State 15	(T1_7:0 != 00000000) * (!(op[3]) * op[2] * op[1] * op[0])
State 15	State 14	Unconditional

Table 1: Transition logic table

4 Finite State Machine



5 Datapath

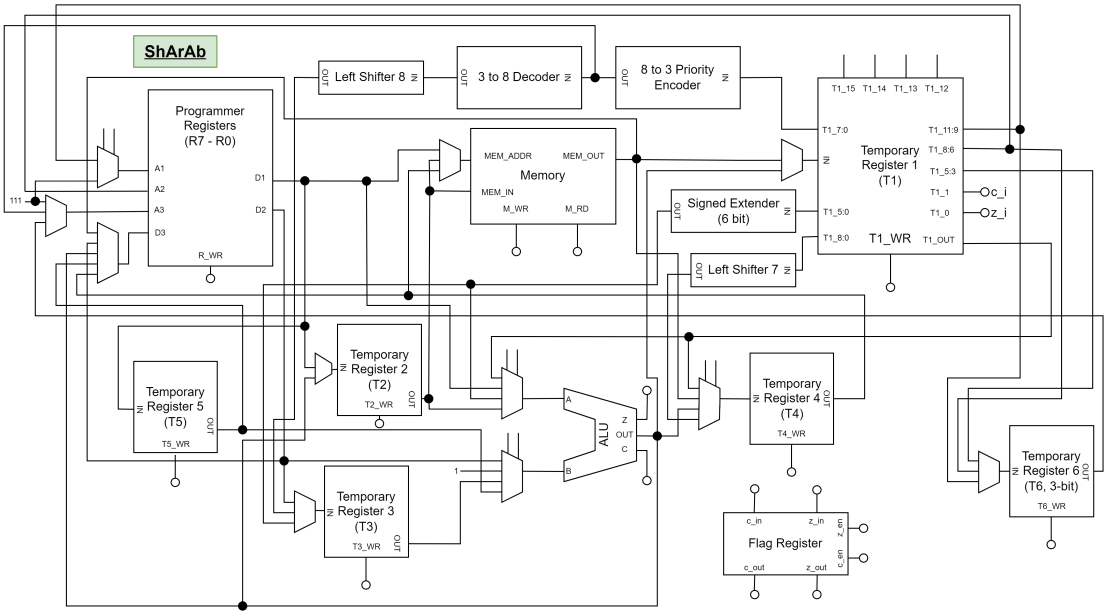


Figure 8: Data Path

5.1 Components used in Data Path

The components used in Data Path are all contained in a package `Components`. The code for the same is given in a drive file. (https://drive.google.com/file/d/1mGKq8yeuYL9Km-GQuibvaU7kGaJ25YWm/view?usp=sharing)

5.2 Code implementation of Data Path

For VHDL implementation of the Data Path, the code can be found on Github. (https://github.com/shobman17/IITB-CPU)

6 Testing

6.1 CPU definition

Entity `CPU` combines both the entities `FSM` and `DataPath` such that the only input it takes is `clk`.

```
library ieee;
use ieee.std_logic_1164.all;

library Work;
use work.Components.all;

entity CPU is
  port(clk: in std_logic);
end entity;

architecture completete of CPU is
  component FSM is
    port( opcode:in std_logic_vector(3 downto 0);
        t1_70:in std_logic_vector(7 downto 0);
        c_i, z_i, z_in, c_out, z_out:in std_logic;
        clk:in std_logic;
        output_state: out std_logic_vector(3 downto 0)
       );
  end component;

  component DataPath is
    port(clk: in std_logic; state: in std_logic_vector(3 downto 0);
      out_c_i, out_z_i,out_z_in, out_c_out, out_z_out: out std_logic;
        opcode: out std_logic_vector(3 downto 0);
        t17downto0: out std_logic_vector(7 downto 0));
  end component;

  for all: Datapath
    use entity work.Datapath(trivial);

  for all: FSM
    use entity work.FSM(shatranj);

  signal t17downto0: std_logic_vector(7 downto 0);
  signal state, opcode: std_logic_vector(3 downto 0);
  signal c_i, z_i, z_in, c_out, z_out: std_logic;

  begin
    Main_Data: Datapath
      port map(clk, state, c_i, z_i, z_in, c_out, z_out, opcode, t17downto0);

    Main_FSM: FSM
      port map(opcode, t17downto0, c_i, z_i, z_in, c_out, z_out, clk, state);
end architecture;
```

`CPU_tb` is the testbench used for `CPU`. It gives a `clk` input with a clock period of 20ns. For testing using this testbench, we hardcoded specific instructions into the Memory storage and Register Files.

6.2 Testing Code

The memory and programmer registers were hard-coded with instructions to test them. The instructions are saved in the submitted code files. Following are the simulation results:

1. ADD R1 R4 R2

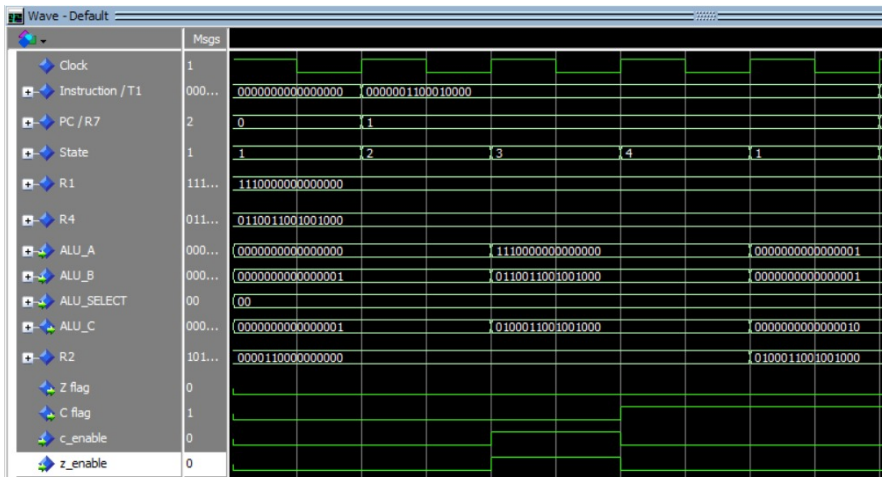


Figure 9: ADD R1 R4 R2

2. ADD R1 R0 R2

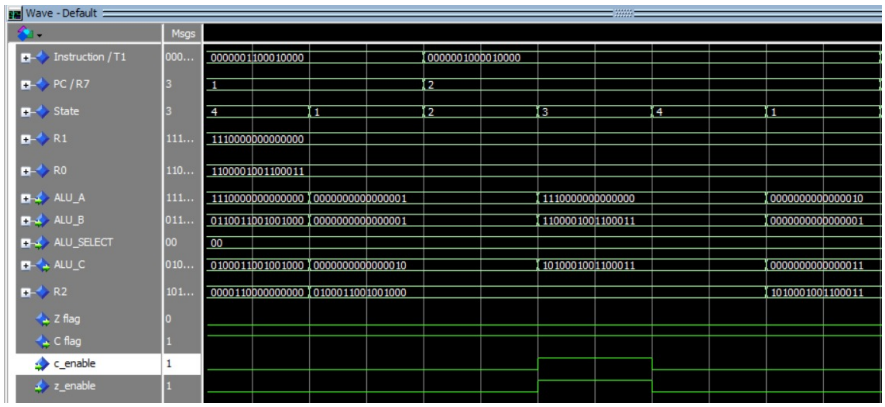


Figure 10: ADD R1 R0 R2

3. ADC R1 R4 R2

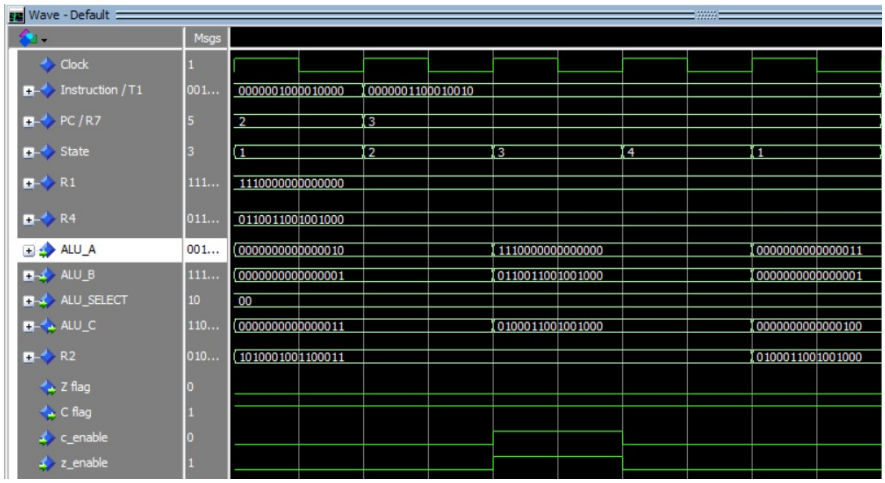


Figure 11: ADC R1 R4 R2

4. ADZ R3 R1 R2

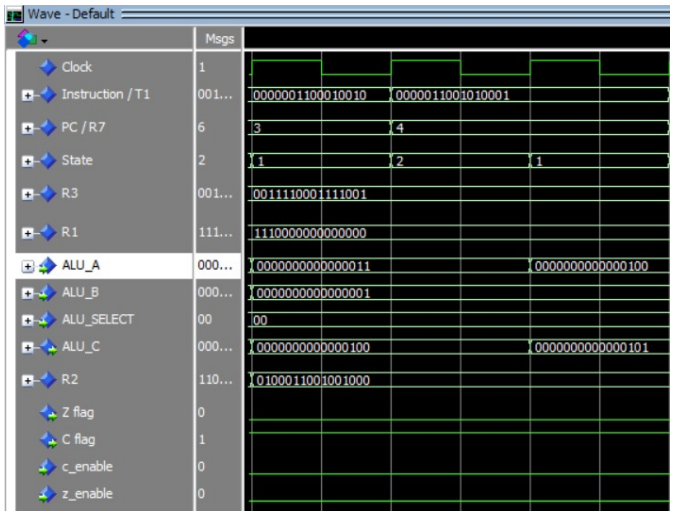


Figure 12: ADZ R3 R1 R2

5. NDU R3 R1 R2

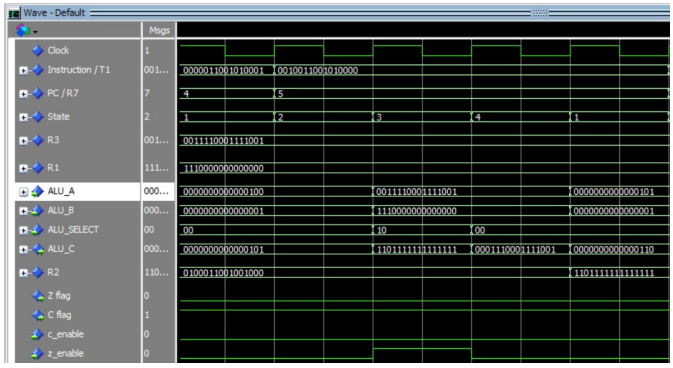


Figure 13: NDU R3 R1 R2

6. NDZ R7 R6 R2

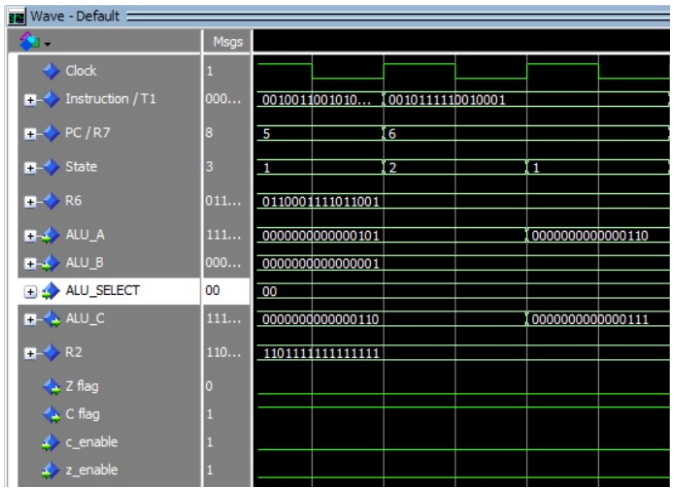


Figure 14: NDZ R7 R6 R2

7. NDC R5 R1 R2

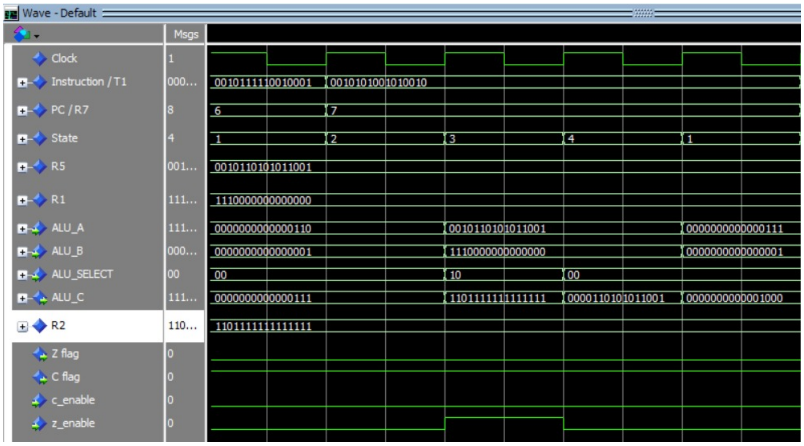


Figure 15: NDC R5 R1 R2

8. ADI R1 R2 010000

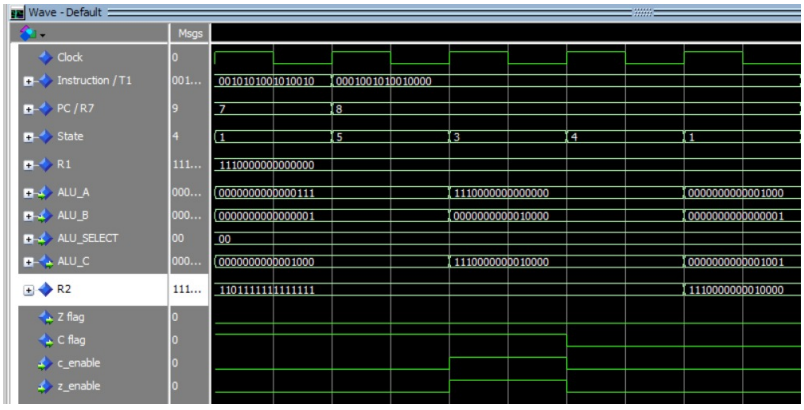


Figure 16: ADI R1 R2 010000

9. LHI R2 100000000

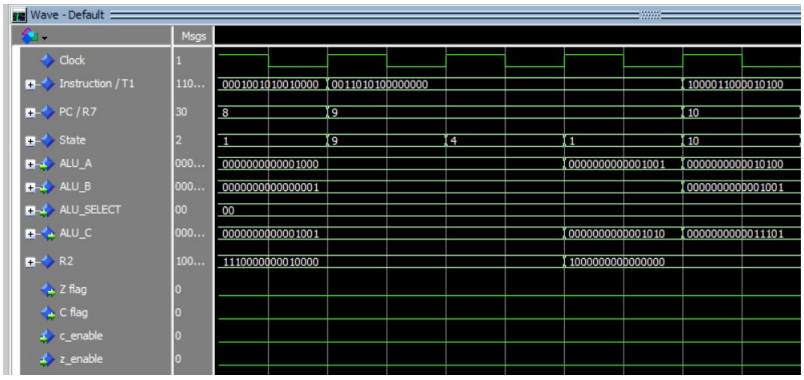


Figure 17: LHI R2 100000000

10. JAL R3 000010100

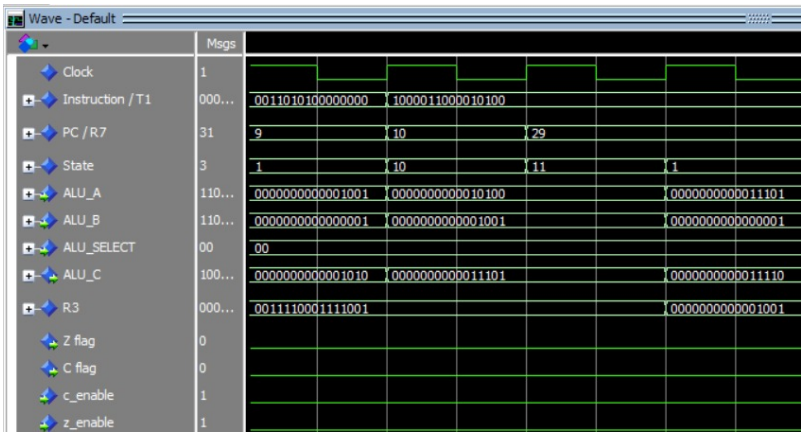


Figure 18: JAL R3 000010100

11. BEQ R3 R3 00000101101

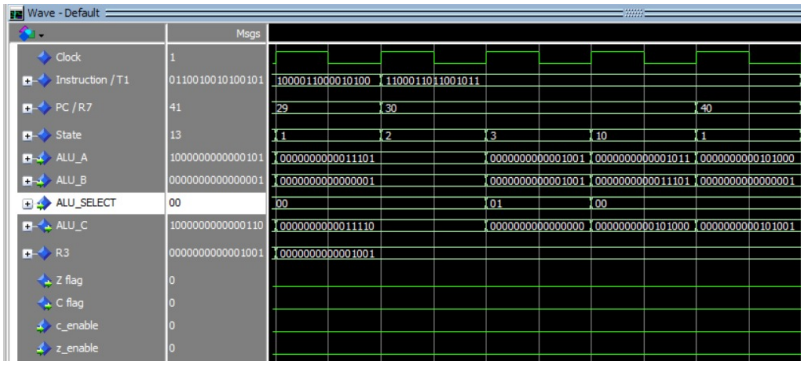


Figure 19: BEQ R3 R3 00000101101

12. LM R2 010101100

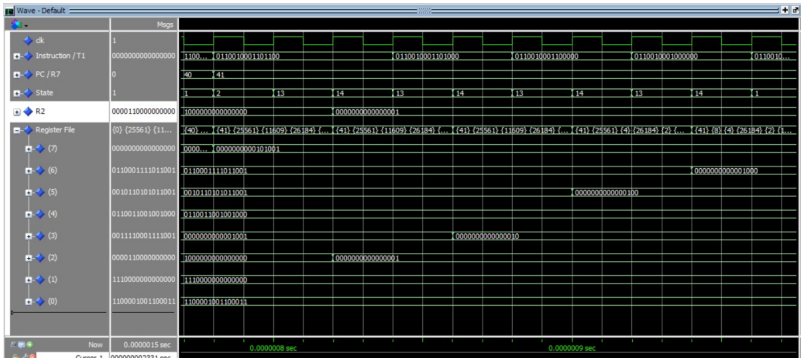


Figure 20: LM R2 010101100

13. SM R3 010101100

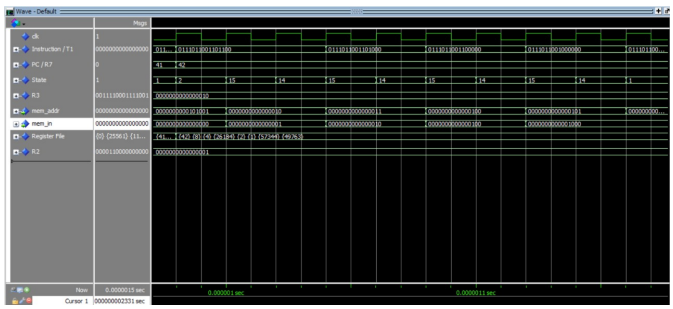


Figure 21: SM R3 010101100

14. SW R1 R2 000001

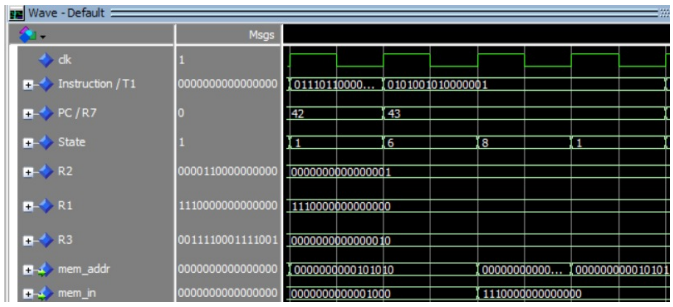


Figure 22: SW R1 R2 000001

15. LW R1 R2 000001

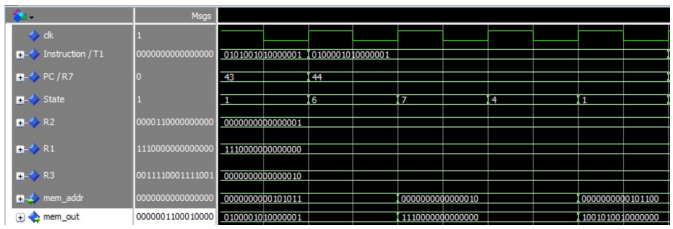


Figure 23: LW R1 R2 000001

16. JLR R2 R2 000000

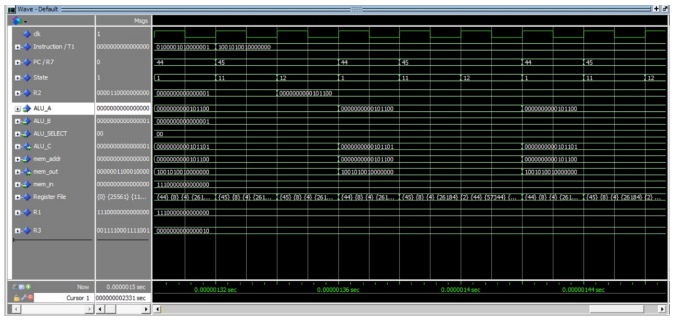


Figure 24: JLR R2 R2 000000

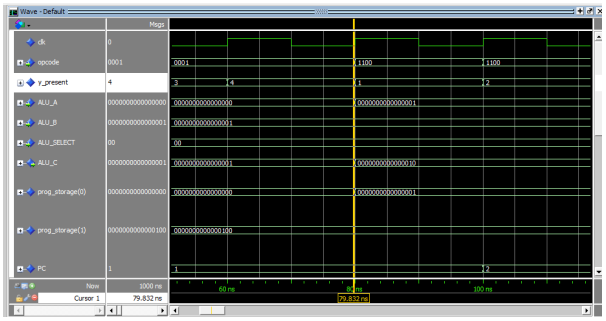
7 Making a for loop

We can implement a for loop using the JLR, BEQ, ADD, and the load and store instructions. $R0$ is initialised with the counter value and $R1$ is initialised with the stop value.

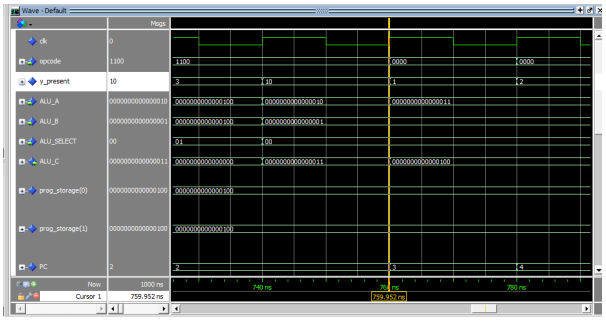
```
--R0 is initialised to "0000000000000000"
--R1 is initialised to "0000000000000100"
--R2 is initialised to "0000000000000000"
--0 has ADI R0 R0 000001
--1 has BEQ R0 R1 000010
--2 has JLR R3 R2 000000
signal prog_storage : reg_array := (1 => "0000000000000100", others => "0000000000000000");
signal memorykagyaan : mem_vec := (0 => "0001000000000001", 1 => "1100000001000010", 2 => "
    1001011010000000", others => "0000000000000000");
```

Listing 1: Memory and Programmer Register initialisation

Following are the RTL simulation results of this for loop implementation:



(a) Counter variable being incremented



(b) Termination of loop

Figure 25: Simulation results of for loop

8 Conclusion

1. Thus using an FSM of 15 states, we implemented 15 instructions each of which can be distinguished using the given op-codes and the C and Z priority bits. The transition between the states is determined by the op-codes, contents of the C and Z flag registers and the C and Z priority bits.

2. The DataPath was implemented by taking in the state and clock as input and produced as output the connections between the various components and the values of the control signals (memory read enable, memory write enable, Program Register write enable, etc.) corresponding to that state.

3. For Testing the entity CPU was made which included both the FSM and the DataPath and took only the clock as input.