# IITB-CPU: Design Documentation

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

**Instructor:** Prof. Virendra Singh

Arin Weling (22B1230)
Chiransh Somani (22B1202)
Sathvik Reddy (22B3946)
Tanish Raghute (22B3974)

Team ID: 44

## Contents

# 1 Introduction

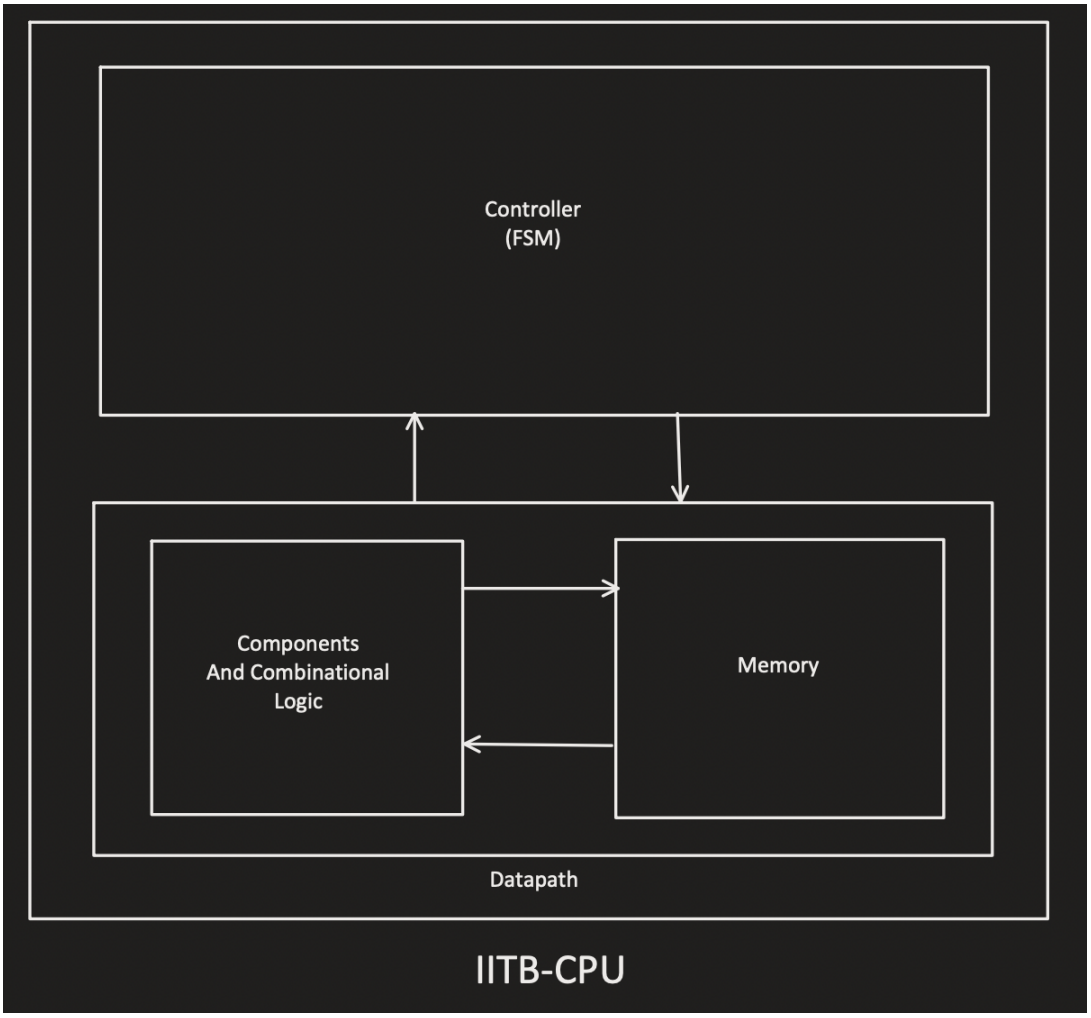
The **IITB-CPU** is an **8-register, 16-bit computer system**, i.e., it can process 16 bits at a time. It implements **14 different instructions**, as well as programs made using them. This design involves using **structural components, memory** and a **controller** (Moore FSM) with **9 total states**, all of which are highlighted in the following sections.

All of our code and our reports can be found on Github (https://github.com/Cove1/IITB-CPU/tree/main).

# 2 Components

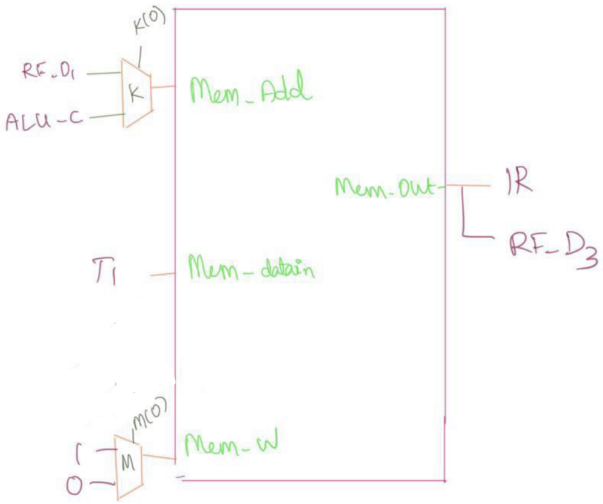
The IITB-CPU is an 8-register, 16-bit computer system. It has 8 general-purpose registers (R0 to R7). Register R7 is always stores Program Counter.

The entities which are part of the CPU are organised broadly in the following manner:



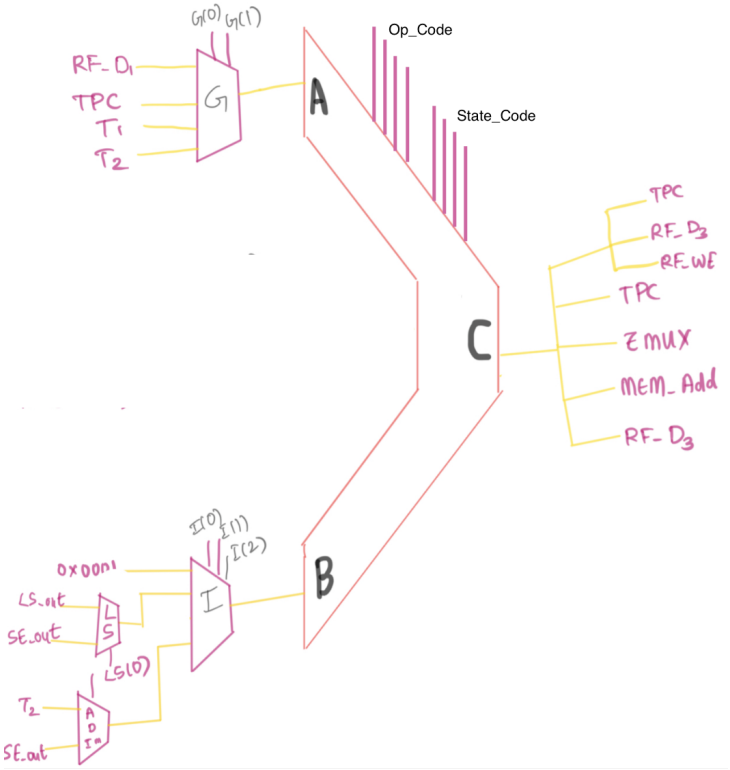

A brief overview of the individual components of the Datapath and their function is as follows:

- **Memory**




  - Stores the instructions to be executed, 2 bytes per instruction.
  - When Mem_Add is fed to the Memory, it gives out the specified 16 bits through Mem_Out.
  - If Memory Write Enable (Mem_W) is on, Data fed to the Memory at Mem_dataIn is updated at the specified Memory Address synchronously.
  - No separate Read Enable, the data at Mem_Add is always available at Mem_Out.

- **Arithmetic and Logic Unit (ALU)**

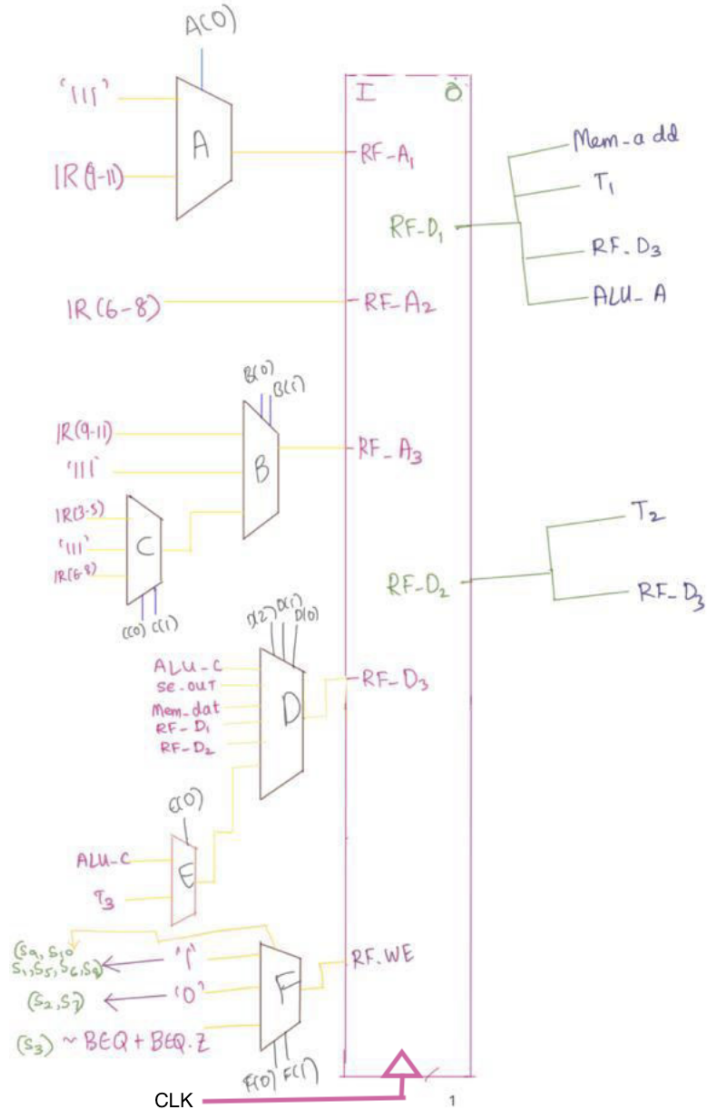



  - 2 16-bit operand inputs, A and B.

– 2 4-bit inputs, Operation code, and State of the Controller, specifying the operation to be carried out by the ALU.

– C, a 16 bit output (independant of Clock).

- **Register File (RF)**




– A collection of 8, 16-bit PIPO registers.

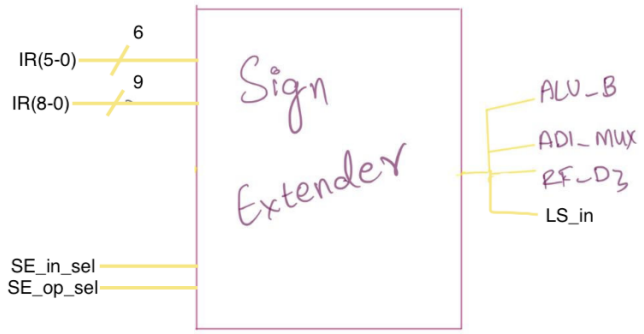
– Write enable bits of each register dependant on the address fed to the RF and RF write enable bit.

– R7 Stores the Program Counter (PC).

– RF_A1, RF_A2 are address inputs for data reading, RF_A3, RF_D3 are inputs for data writing.

– RF_D1, RF_D2 are outputs to read data.

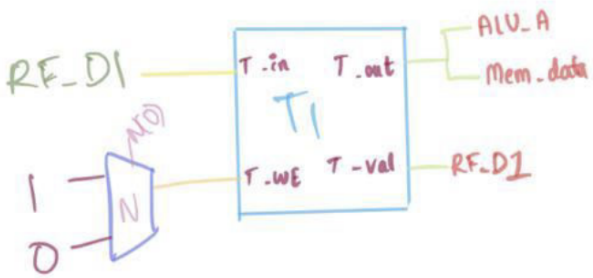
- **Sign Extender (SE)**

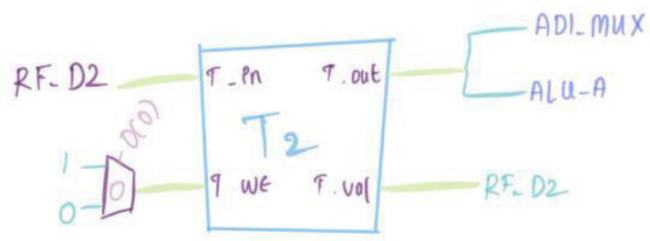



  - Takes a 6-bit input and a 9-bit input, with a select line to choose between them, and outputs the 16-bit extended value.
  - Has a select line for right and left 0-padding.

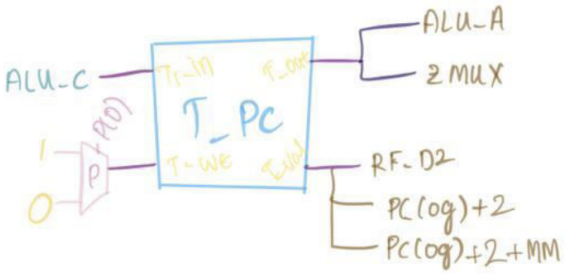
- **Left Shifter (LS)** : to double 16-bit values.


- **T1, T2**: Temporary Registers, operand related.



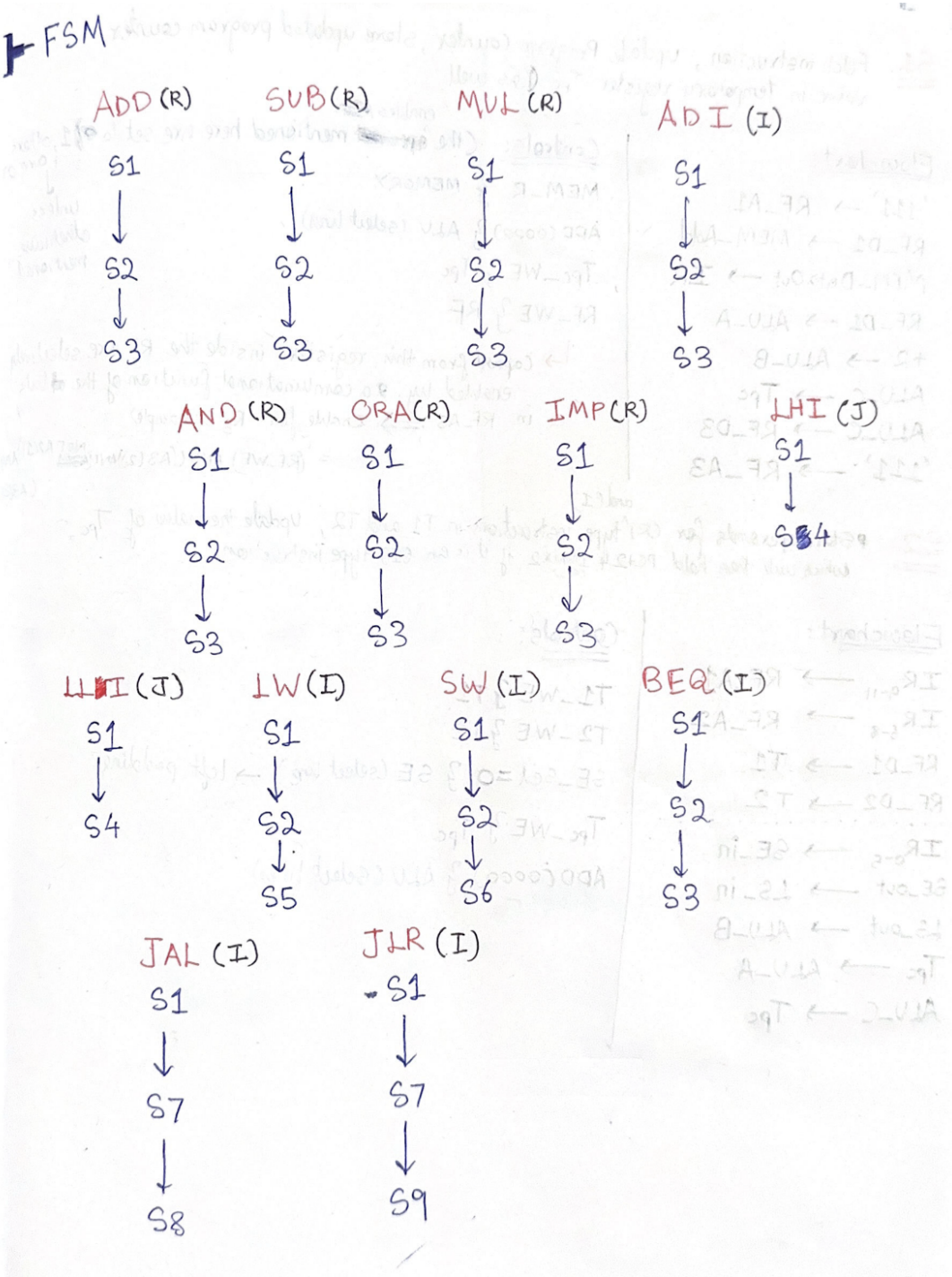
- **Tpc**: Temporary register, related to Program Counter operations.




- **Instruction Register (IR)**: Stores the 16-bit Instruction read from memory, to be used in various places.

# 3 Operations

Before getting into the individual states, here were the state transitions for each operation, according to which states were defined:



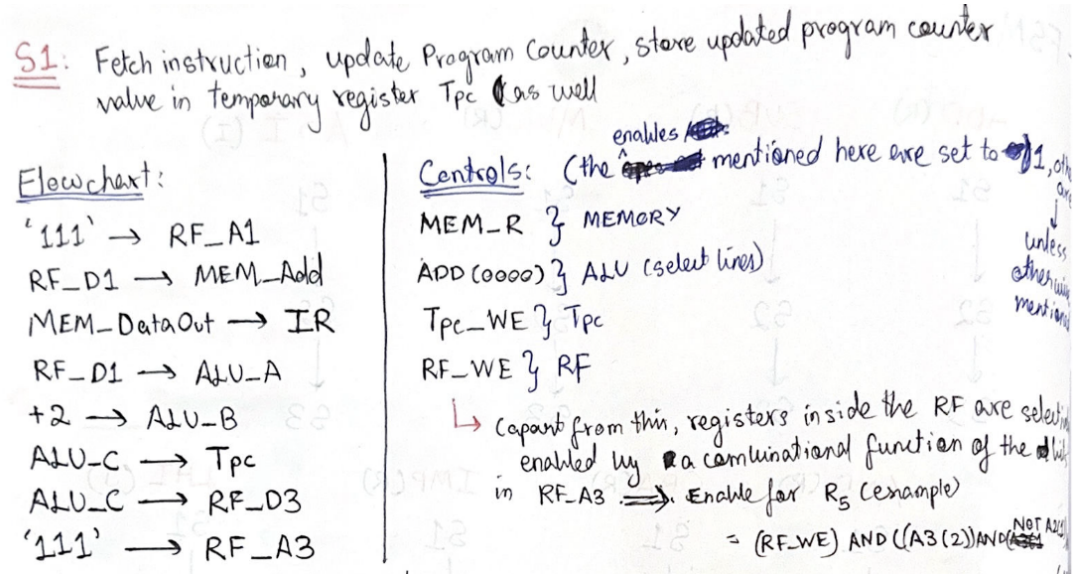

These transitions resulted from numerous iterations of state optimization and redefinition. They were optimized to 9 total states.

For all operations, after execution of the shown states, the controller goes back to S1.

# 4 States

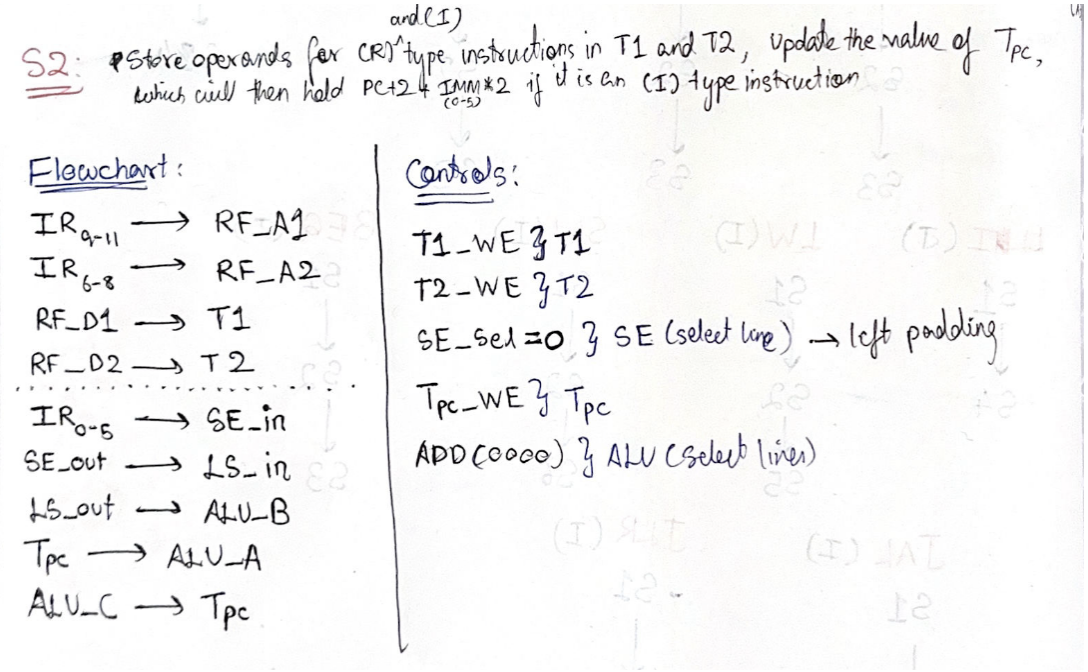
The following images entail the Data-flow for each state (9 in total):

## 4.1 S1



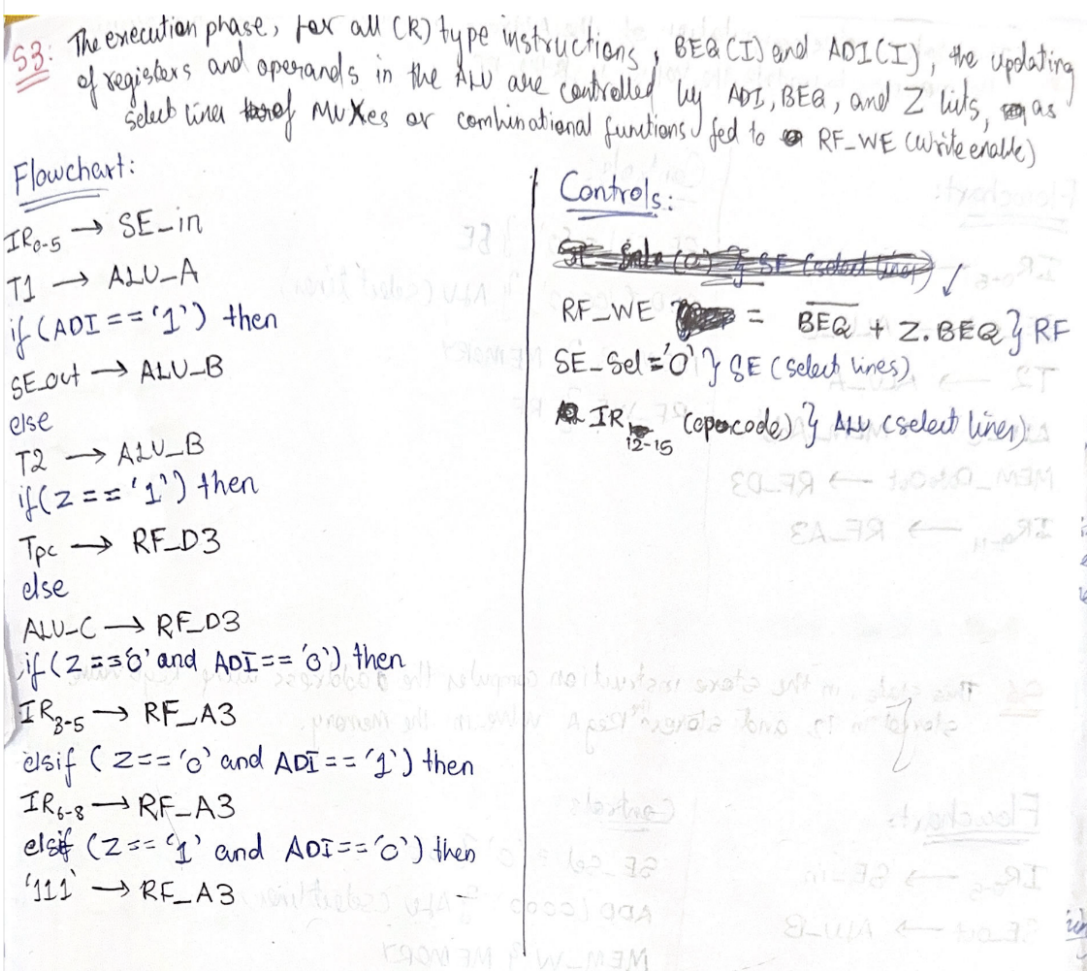

This state mainly consists of fetching the instruction from the memory and updating the program counter in R7, Tpc, a temporary register tracks R7 for a future operation related to BEQ.

## 4.2 S2



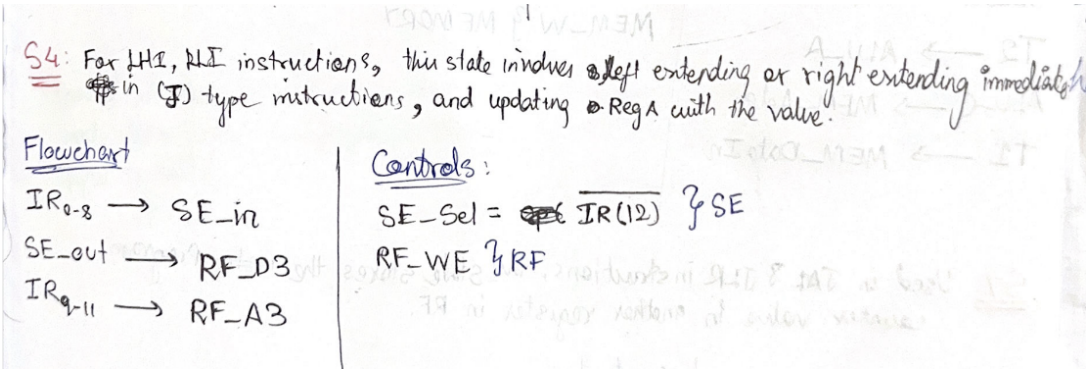

This state involves storing of the appropriate operands in Temporary registers T1 and T2.
In case of BEQ instructions, Tpc also gets updated with the possible branched Instruction counter.

### 4.3 S3



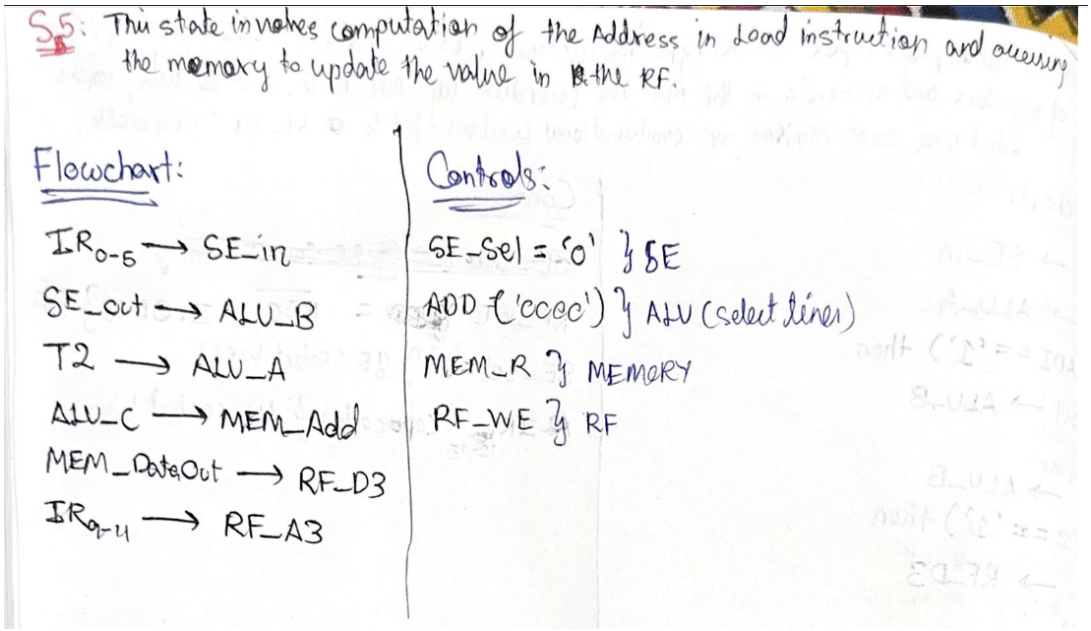

This state is concerned with execution of the operation and subsequent updation of RF values. ADI Operation dependant bits determine the input to ALU_B (in case of ADI, it comes through the sign extender).
The Z-bit of the ALU also controls the updation of the RF in case of BEQ instructions (The RF write enable is also (BEQ,Z) dependant).

### 4.4 S4



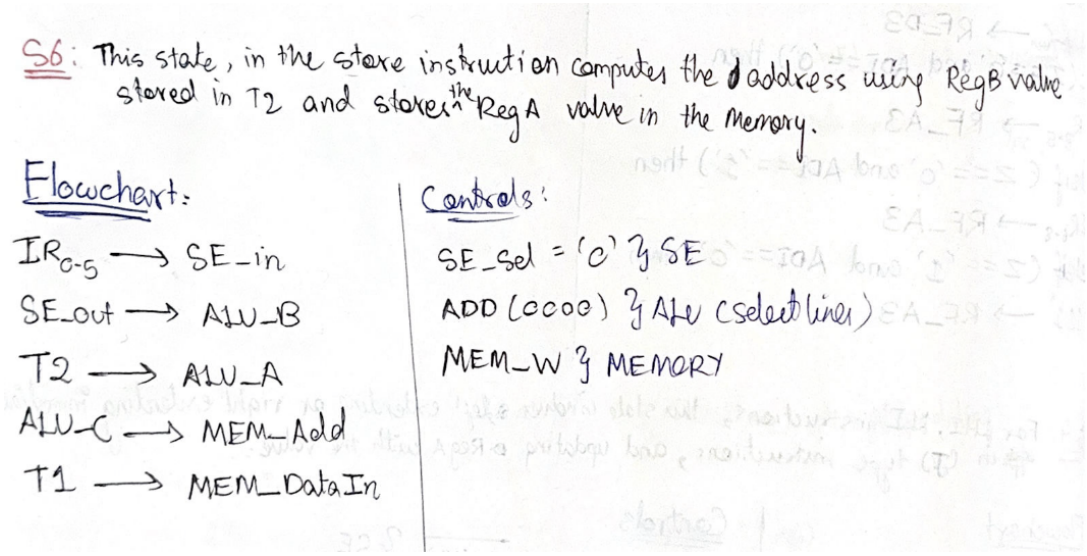

This state, concerning LHI and LLI operations, involves sign extending the 6 bit immediate based on the instruction Op_code, and updating the RF.

### 4.5 S5



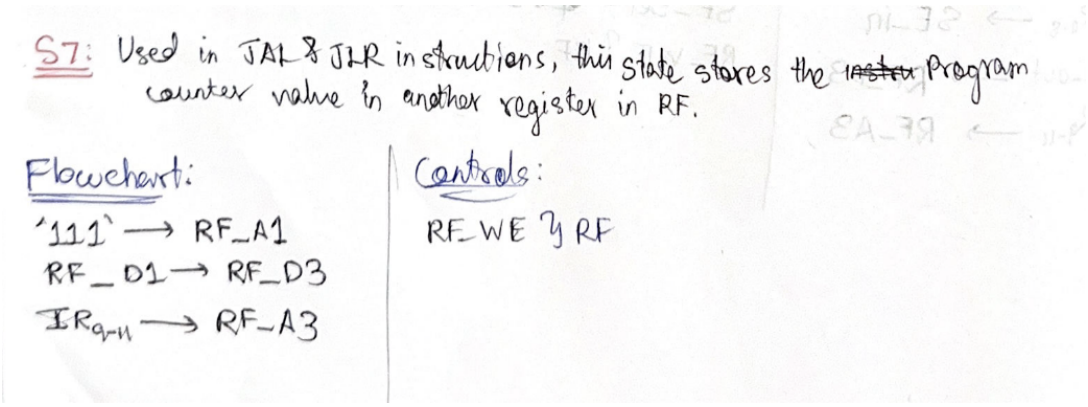

Exclusive to LW instructions, S5 is the memory address computation state (separated from S3 as D3 and A3 are occupied in S3). The memory is also accessed, and the RF updated in this state.

### 4.6 S6



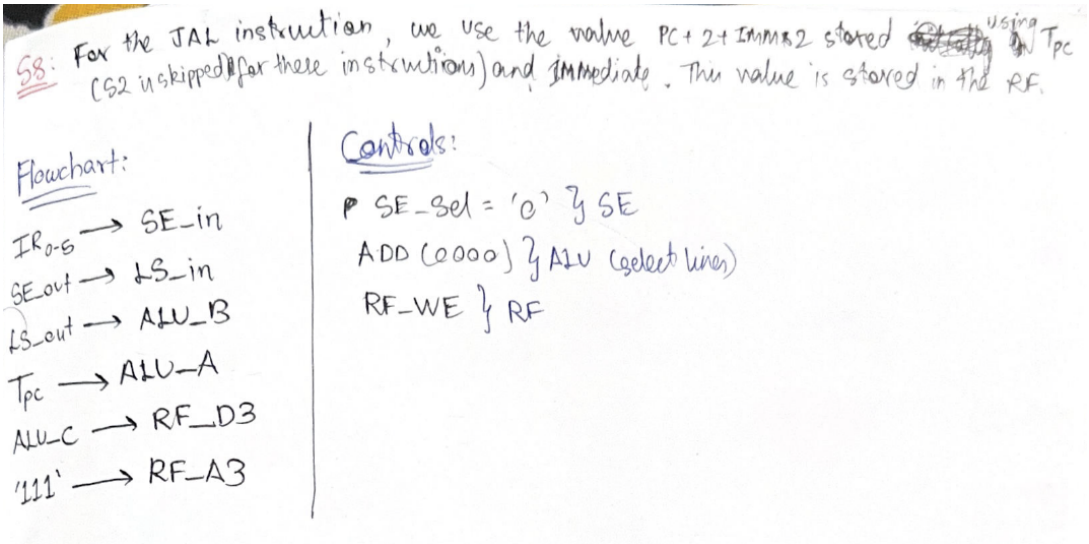

Exclusive to SW instructions, S6 is related to Memory address computation and data storing.

### 4.7 S7



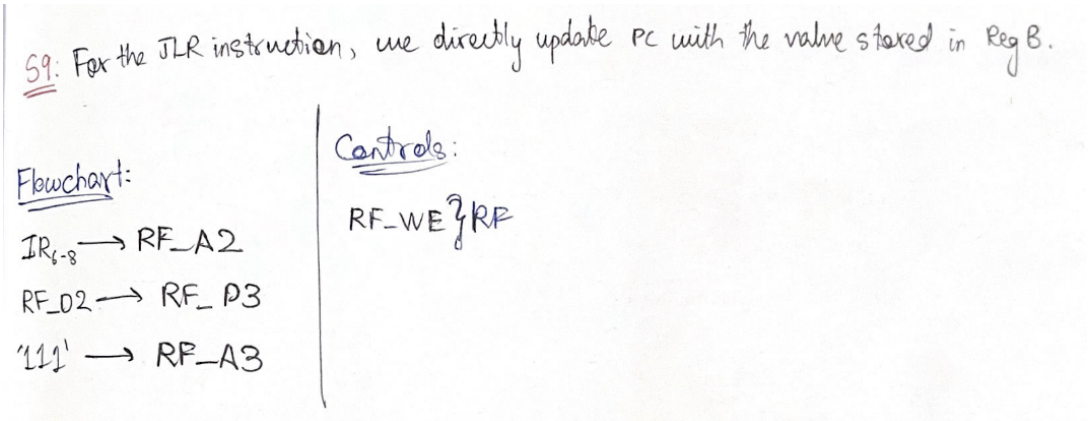

Stores the current PC value in JAL and JLR instructions.

### 4.8 S8




Exclusive to JAL, Computes the PC value to jump to and update the PC (R7)

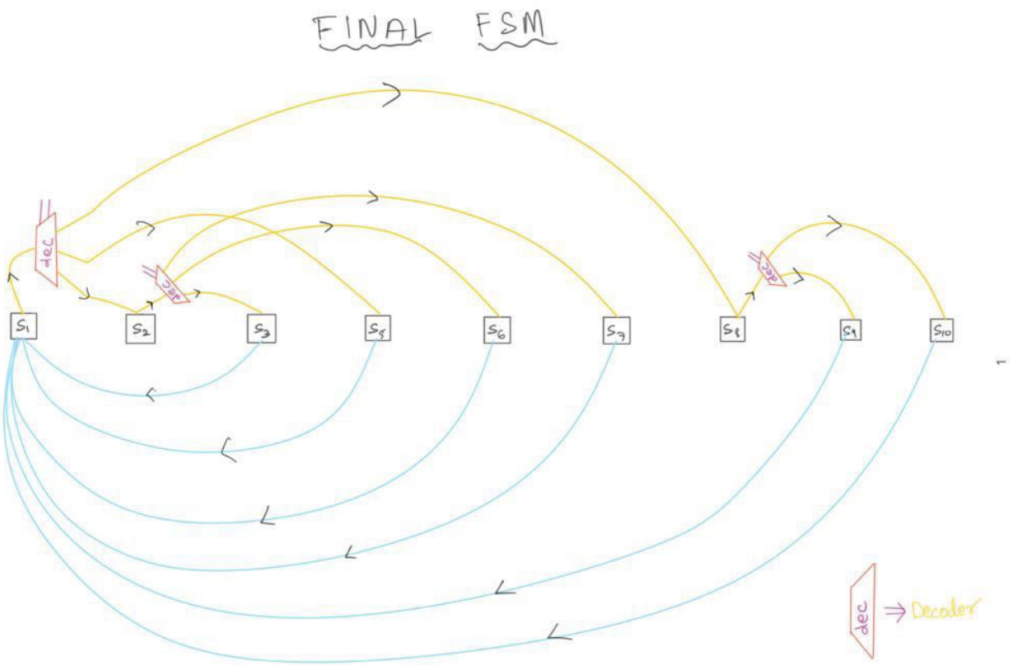
### 4.9 S9



Exclusive to JLR, Directly updates PC with value of Reg B.

These states were combined into our Mealy FSM Controller with the following state diagram:

### 4.10 Final FSM




## 5 Control bits

The following control bits/signals defined the operation of the Datapath and Memory in a given state, hence these bits are State-Dependant, and came as outputs from the Controller to the Datapath and Memory.

- Enable bits [1 bit]
  - IR Write Enable (ir_write) (only 1 in S1)
  - T1 Write Enable (t1_write)
  - T2 Write Enable (t2_write)
  - Tpc Write Enable (t_pc_write)
  - RF Write Enable (rf_write)
  - Memory Write Enable (mem_write)
- Input Selection bits (for MUXs at the inputs)
  - RF_A1 Select (rf_a1_sel) [1 bit]
  - Memory Address Select (mem_addr_sel) [1 bit]
  - Sign Extender Input Select (se_in_sel) [1 bit]
  - Sign Extender Operation Select (se_op_sel) [1 bit]
  - RF_A3 Select (rf_a3_sel) [2 bits]

- ALU A Select (alu_a_sel) [2 bits]
- ALU B Select (alu_b_sel) [2 bits]
- State of the Controller (out_state) [3 bits]

Along with this, the state S3 had multiple purposes, controlled by Operation Dependant bits:

- Z [1 bit]
- Operation code [3 bits]

# 6 Datapath

Combining all components, along with the control/enable bits they required, our final datapath took shape.

The input and output mapping various components is highlighted in the following images state-wise (We felt this was a less confusing way to represent our Dataflow than designing the actual circuit).

| ALU | STATES | S1 | S2 | S3 | S4 | S5 | S6 | S7 | S8 | S19 |
|---|---|---|---|---|---|---|---|---|---|---|
| INPUT | | | | | | | | | | |
| A | | RF_D1 | TPC | T1 | X | T2 | T2 | X | RF_D1 | X |
| B | | 00000000000010 | LS MUX | ADI MUX | X | LS MUX | LS MUX | X | LS MUX | X |
| SEL | | ADD (0000) | ADD (0000) | IR(15 down to 12) | X | ADD (0000) | ADD (0000) | X | ADD (0000) | X |
| OUTPUTS | | | | | | | | | | |
| C | | TPC, RF_D3 | TPC | Z MUX (0) | X | Mem_Add | Mem_Add | X | RF_D3 | X |

| RF | STATES | S1 | S2 | S3 | S4 | S5 | S6 | S7 | S8 | S9 |
|---|---|---|---|---|---|---|---|---|---|---|
| INPUT | | | | | | | | | | |
| RF_A1 | | 111 | IR(9-11) | X | X | X | X | 111 | 111 | X |
| RF_A2 | | X | IR(6-8) | X | X | X | X | X | X | IR(6-8) |
| RF_A3 | | 111 | X | Z&ADI MUX \|\| IR3-5 (Z=0, ADI=0)/ '111'(Z=1,ADI=0)/ IR(6-8) (Z=0,ADI=1) | IR(9-11) | IR(9-11) | X | IR9-11 | 111 | 111 |
| RF_D3 | | ALU_C | X | Z MUX \|\| ALU_C (Z=0) / T3 | SE_out | Mem_data | X | RF_D1 | ALU_C | RF_D2 |
| RF_WE | | 1 | 0 | ~BEQ+ BEQ.Z (USE DECODER) | 1 | 1 | 0 | 1 | 1 | 1 |
| OUTPUTS | | | | | | | | | | |
| RF_D1 | | Mem_add, ALU_A | T1 | X | X | X | X | RF_D3 | ALU_A | X |
| RF_D2 | | X | T2 | X | X | X | X | X | X | RF_D3 |

| Memory | STATES | S1 | S2 | S3 | S4 | S5 | S6 | S7 | S8 | S9 |
|---|---|---|---|---|---|---|---|---|---|---|
| INPUT | | | | | | | | | | |
| MEM_Add | | RF_D1 | X | X | X | ALU_C | ALU_C | X | X | X |
| MEM_DataIn | | X | X | X | X | X | T1 | X | X | X |
| MEM_W | | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| OUTPUTS | | | | | | | | | | |
| MEM_DataOut | | IR | X | X | X | RF_D3 | X | X | X | X |

| SE | STATES | S1 | S2 | S3 | S4 | S5 | S6 | S7 | S8 | S9 |
|---|---|---|---|---|---|---|---|---|---|---|
| INPUT | | | | | | | | | | |
| SE_in_6bit | | X | IR(0-5) | IR(0-5) | X | IR(0-5) | IR(0-5) | X | IR(0-5) | X |
| SE_in_9bit | | X | X | X | IR(0-8) | X | X | X | X | X |
| SE_op_sel | | X | 0 | 0 | NOT IR(12) | 0 | 0 | X | 0 | X |
| SE_in_sel | | X | 0 | 0 | 1 | 0 | 0 | X | 0 | X |
| OUTPUTS | | | | | | | | | | |
| SE_out | | X | ALU_B | ADI MUX (ADI = 1) | RF_D3 | ALU_B | ALU_B | X | ALU_B | X |

| LS | STATES | |
|---|---|---|
| INPUT | | |
| LS_in | | In all states, LS_in comes from SE_out (when used, only one possible mapping) |
| OUTPUTS | | |
| LS_out | | In all states, LS_out goes to the input MUX of ALU_B |

| T1 | STATES | S1 | S2 | S3 | S4 | S5 | S6 | S7 | S8 | S9 |
|---|---|---|---|---|---|---|---|---|---|---|
| INPUT | | | | | | | | | | |
| T1_In | | X | RF_D1 | X | X | X | X | X | X | X |
| T1_WE | | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| OUTPUTS | | | | | | | | | | |
| T1_Out | | X | X | ALU_A | X | X | MEM_DataIn | X | X | X |
| VALUE | | | | | | | | | | |
| T1_Value | | X | X | RF_D1 | X | RF_D1 | RF_D1 | X | X | X |

| T2 | STATES | S1 | S2 | S3 | S4 | S5 | S6 | S7 | S8 | S9 |
|---|---|---|---|---|---|---|---|---|---|---|
| INPUT | | | | | | | | | | |
| T2_In | | X | RF_D2 | X | X | X | X | X | X | X |
| T2_WE | | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| OUTPUTS | | | | | | | | | | |
| T2_Out | | X | X | ADI MUX (ADI=0) | X | ALU_A | ALU_A | X | X | X |
| VALUE | | | | | | | | | | |
| T2_Value | | X | X | RF_D2 | X | RF_D2 | RF_D2 | X | X | X |

| Tpc | STATES | S1 | S2 | S3 | S4 | S5 | S6 | S7 | S8 | S9 |
|---|---|---|---|---|---|---|---|---|---|---|
| INPUT | | | | | | | | | | |
| Tpc_In | | ALU_C | ALU_C | X | X | X | X | X | X | X |
| Tpc_WE | | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| OUTPUTS | | | | | | | | | | |
| Tpc_Out | | X | ALU_A | Z MUX (Z=1) | X | ALU_A | ALU_A | X | X | X |
| VALUE | | | | | | | | | | |
| Tpc_Value | | X | PC(og) +1 | PC(og)+ 1+IMM | X | RF_D2 | RF_D2 | X | X | X |

The VHDL code for the Components and the Datapath Implementation can be found on Github (https://github.com/Cove1/IITB-CPU/tree/main).

# 7 Transitional Logic

To realise this controller, this FSM, state transitions can be carried out depending on the values of:

- State of the Controller [3 bits]
- Operation code [3 bits]

The following process code represents the transition logic of our CPU:

```
22  state_machine:process(op_state,current_state,next_state)
23  begin
24      case op_state is
25          when "0000"|"0010"|"0011"|"0001"|"0100"|"0101"|"0110"|"1100" =>
26              if(current_state="0001") then
27                  next_state<="0010";
28              elsif (current_state="0010") then
29                  next_state<="0011";
30              elsif (current_state="0011") then
31                  next_state<="0001";
32              else
33                  next_state<="0001";
34              end if;
35          when "1000"|"1001" =>
36              if(current_state="0001") then
37                  next_state<="0100";
38              elsif (current_state="0100") then
39                  next_state<="0001";
40              else
41                  next_state<="0001";
42
43              end if;
44          when "1010" =>
45              if(current_state="0001") then
46                  next_state<="0010";
47              elsif (current_state="0010") then
48                  next_state<="0101";
49              elsif (current_state="0101") then
50                  next_state<="0001";
51              else
52                  next_state<="0001";
53              end if;
54          when "1011" =>
55              if(current_state="0001") then
56                  next_state<="0010";
57              elsif (current_state="0010") then
58                  next_state<="0110";
59              elsif (current_state="0110") then
60                  next_state<="0001";
61              else
62                  next_state<="0001";
```

```
            end if;
        when "1101" =>
            if(current_state="0001") then
                next_state<="0111";
            elsif (current_state="0111") then
                next_state<="1000";
            elsif (current_state="1000") then
                next_state<="0001";
            else
                next_state<="0001";
            end if;
        when "1111" =>
            if(current_state="0001") then
                next_state<="0111";
            elsif (current_state="0111") then
                next_state<="1001";
            elsif (current_state="1001") then
                next_state<="0001";
            else
                next_state<="0001";
            end if;
        when others =>
            next_state<="0001";
    end case;
end process;
state_transition:process(current_state,next_state,clock)
begin
    if(clock='0' and clock'event) then
        current_state<=next_state;
    end if;
end process;
```

# 8 Testing

## 8.1 CPU Definition

The **CPU** combined the **Datapath**, as well as the **Controller**, with the only input being **clock**:

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

entity CPU is
    port(clock,reset:in std_logic);
end entity CPU;
```

```
architecture bhv of CPU is
component controller is
    port (ir:in std_logic_vector(15 downto 0);
            clock,z:in std_logic;
            ir_write,rf_a1_sel,mem_addr_sel,rf_write,t_pc_write,t1_write,t2_write,mem_write,se_in_sel,se_op_sel:out std_logic;
            alu_a_sel,alu_b_sel,rf_a3_sel:out std_logic_vector(1 downto 0);
            rf_d3_sel:out std_logic_vector(2 downto 0);
            out_state,out_op:out std_logic_vector(3 downto 0));
end component controller;


component DataPath is
port (  reset,ir_write,rf_a1_sel,mem_addr_sel,rf_write,t_pc_write,t1_write,t2_write,mem_write,se_in_sel,se_op_sel:in std_logic;
            alu_a_sel,alu_b_sel,rf_a3_sel:in std_logic_vector(1 downto 0);
            rf_d3_sel:in std_logic_vector(2 downto 0);
            clock: in std_logic;
            out_state,out_op:in std_logic_vector(3 downto 0);


            ir_data: out std_logic_vector(15 downto 0);
            zero_flag:out std_logic);

end component DataPath;


signal ir_writea,rf_a1_sela,mem_addr_sela,rf_writea,t_pc_writea,t1_writea,t2_writea,mem_writea,se_in_sela,se_op_sela: std_logic;
signal alu_a_sela,alu_b_sela,rf_a3_sela: std_logic_vector(1 downto 0);
signal rf_d3_sela: std_logic_vector(2 downto 0);

signal  out_statea,out_opa:std_logic_vector(3 downto 0);
signal  ir_dataa:  std_logic_vector(15 downto 0);
signal  zero_flaga: std_logic;
begin

controller_1: controller port map(ir_dataa,clock,zero_flaga,ir_writea,rf_a1_sela,mem_addr_sela,rf_writea,t_pc_writea,t1_writea,t2_
                                  alu_a_sela,alu_b_sela,rf_a3_sela,rf_d3_sela,out_statea,out_opa);

data_path_1:DataPath port map(reset,ir_writea,rf_a1_sela,mem_addr_sela,rf_writea,t_pc_writea,t1_writea,t2_writea,mem_writea,se_in_
                              alu_a_sela,alu_b_sela,rf_a3_sela,rf_d3_sela,clock,out_statea,out_opa,
                              ir_dataa,zero_flaga);

end architecture bhv;
```

The clock frequency used was **25MHz**.

## 8.2 Memory Initialisation

Initially, we hard coded the instructions into the memory component, and ran simulations for different Instructions.

## 8.3 Automating the Memory Initialisation

To better the testability of our CPU, we changed our memory initialisation method from hard coding in VHDL to creating a txt file through python code.

This txt file would then be read by our memory component into an array of bytes as before, and the Datapath would access this memory as before.

A similar type of initialisation was implemented for the Register File values as well.

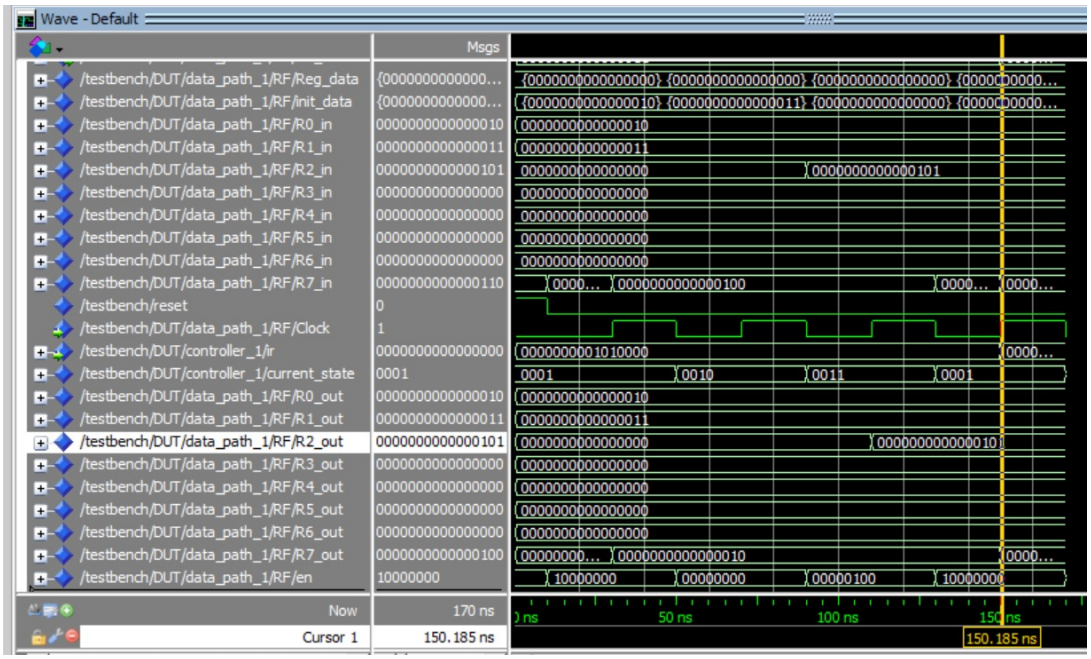
The code related to this, as well as the txt files can be found at the Github repository linked here

```
PS C:\Users\arin weling> python -u "c:\Users\arin weling\Desktop\Apple_M69\user.py"
Reset Memory? [1 for Yes, 0 for No]: 1
Reset Register File?[1 for Yes, 0 for No]: 0
Enter option: [0 for Data_input_memory, 1 for Data_input_register, 2 for Instruction input, -1 to stop]
2
Enter operation type[ADD,SUB,MUL,ADI,AND,ORA,IMP,LHI,LLI,LW,SW,BEQ,JAL,JLR]: IMP
Enter address:0
Enter address of register_a:0
Enter address of register_b:1
Enter address of register_c:2
Enter option: [0 for Data_input_memory, 1 for Data_input_register, 2 for Instruction input, -1 to stop]
-1
End
PS C:\Users\arin weling> python -u "c:\Users\arin weling\Desktop\Apple_M69\user.py"
Reset Memory? [1 for Yes, 0 for No]: 1
Reset Register File?[1 for Yes, 0 for No]: 0
Enter option: [0 for Data_input_memory, 1 for Data_input_register, 2 for Instruction input, -1 to stop]
2
Enter operation type[ADD,SUB,MUL,ADI,AND,ORA,IMP,LHI,LLI,LW,SW,BEQ,JAL,JLR]: LHI
Enter address:0
Enter address of register_a:2
Enter immediate:19
Enter option: [0 for Data_input_memory, 1 for Data_input_register, 2 for Instruction input, -1 to stop]
-1
End
PS C:\Users\arin weling> python -u "c:\Users\arin weling\Desktop\Apple_M69\user.py"
Reset Memory? [1 for Yes, 0 for No]: 1
Reset Register File?[1 for Yes, 0 for No]: 0
Enter option: [0 for Data_input_memory, 1 for Data_input_register, 2 for Instruction input, -1 to stop]
2
Enter operation type[ADD,SUB,MUL,ADI,AND,ORA,IMP,LHI,LLI,LW,SW,BEQ,JAL,JLR]: LLI
Enter address:0
Enter address of register_a:2
Enter immediate:19
```

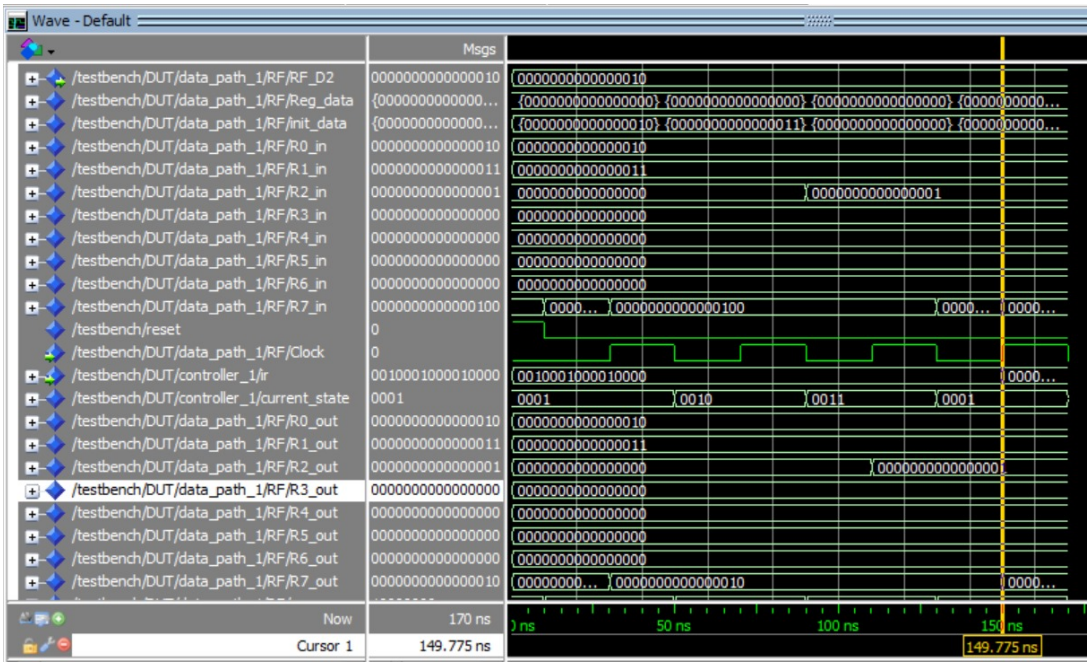
Figure 1: User Interface for the initialisation process

## 8.4 Results

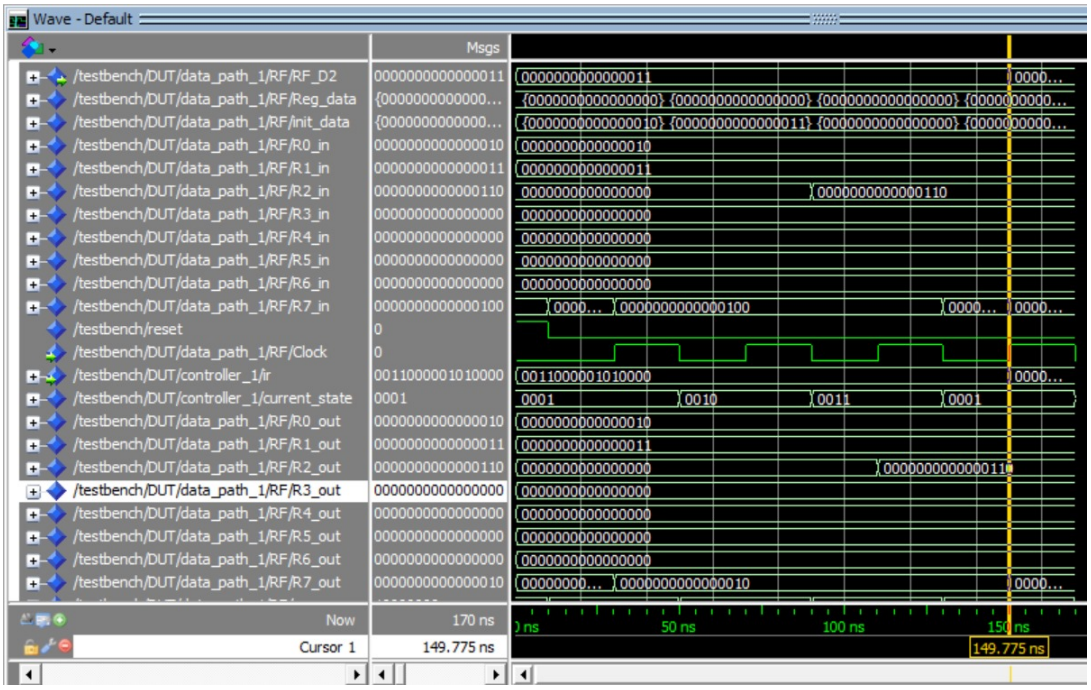
The CPU successfully carried out all instructions individually, which can be seen in the following simulation results:



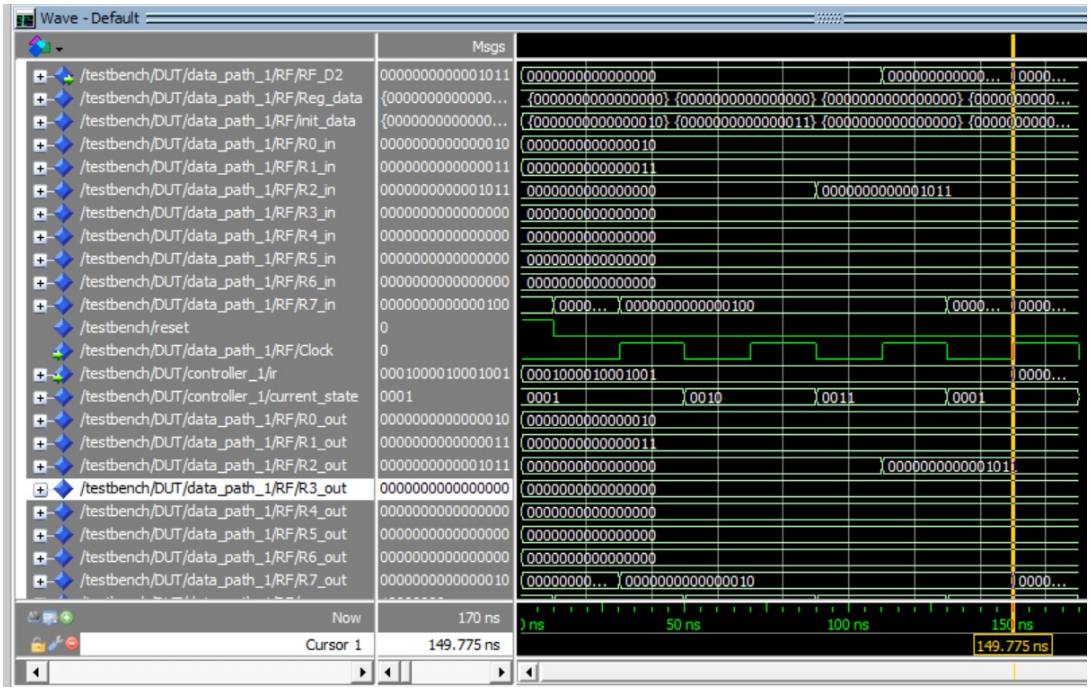

ADD R2,R0,R1



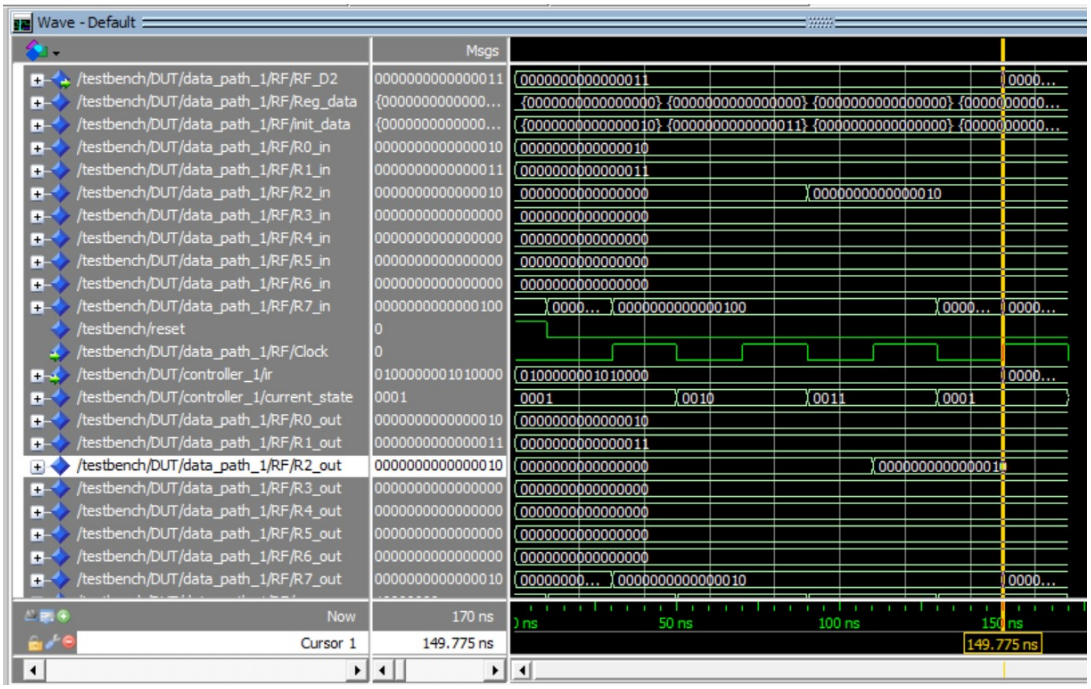

SUB R2,R0,R1



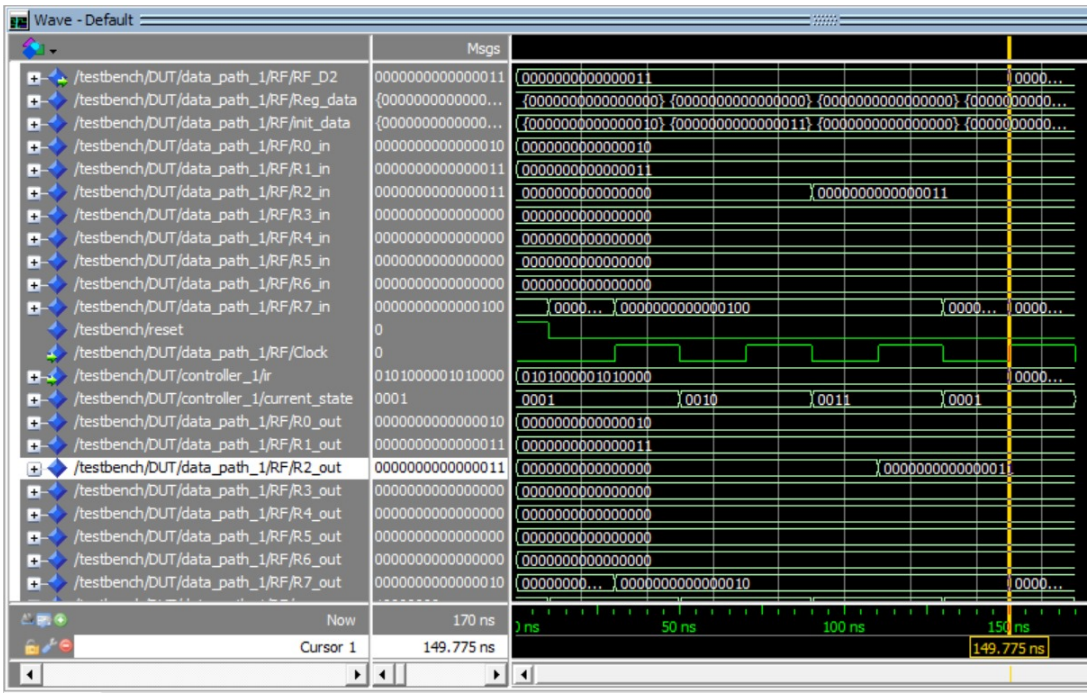

MUL R2,R0,R1



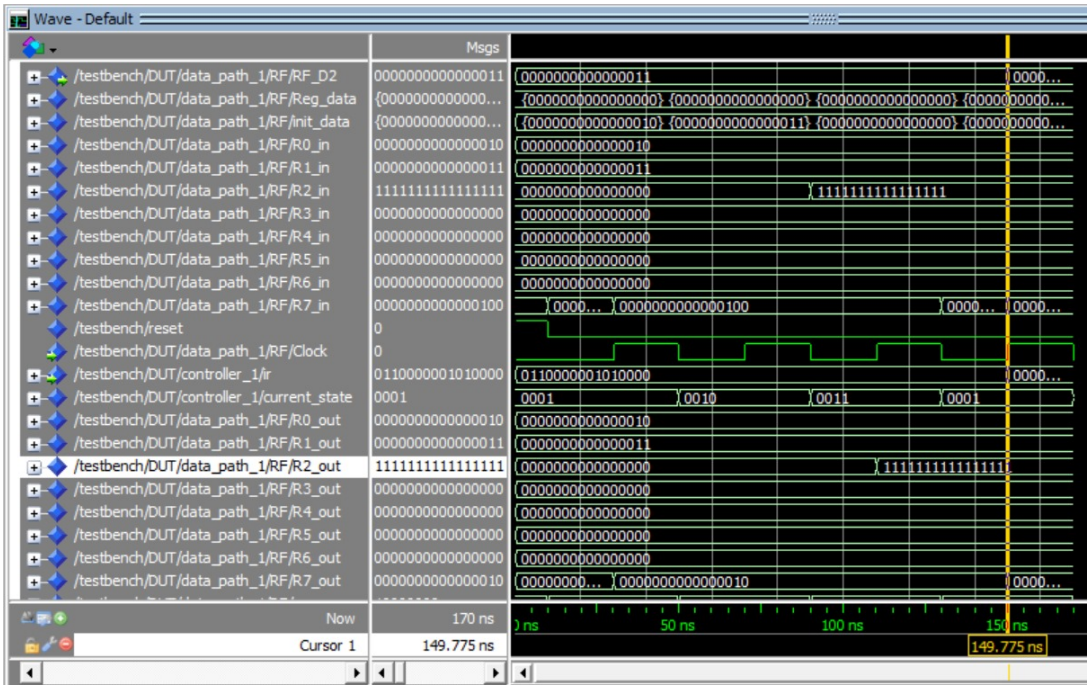

ADI R2,R0,001001



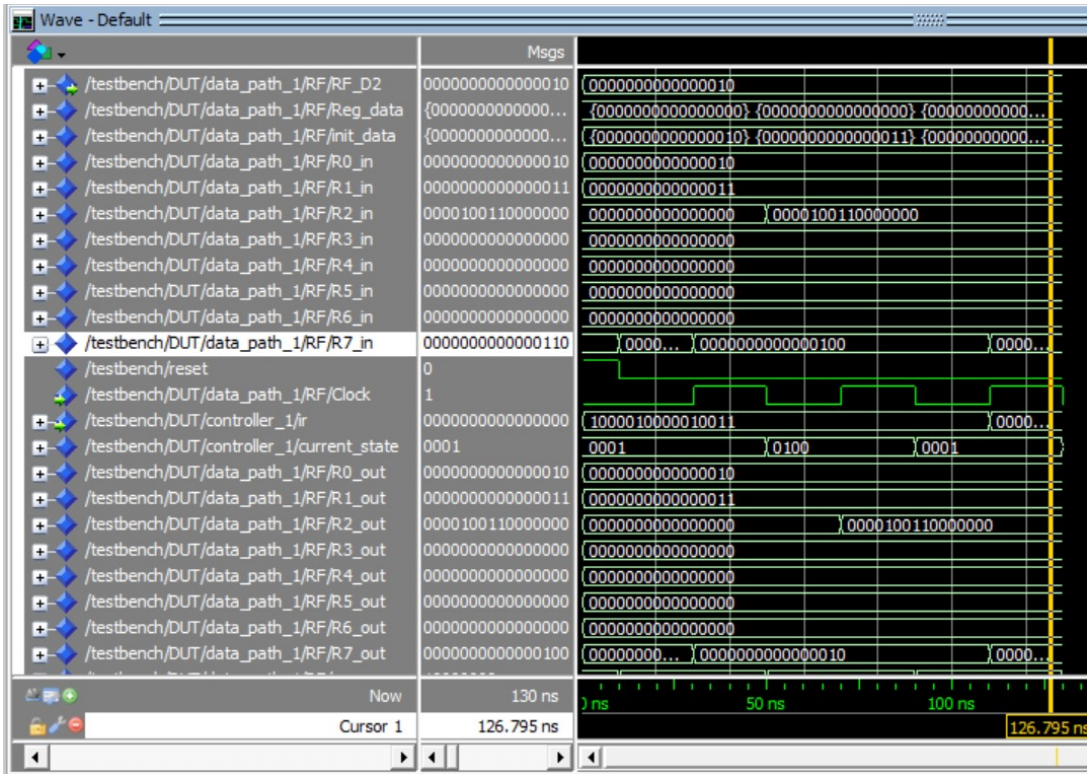

AND R2,R0,R1



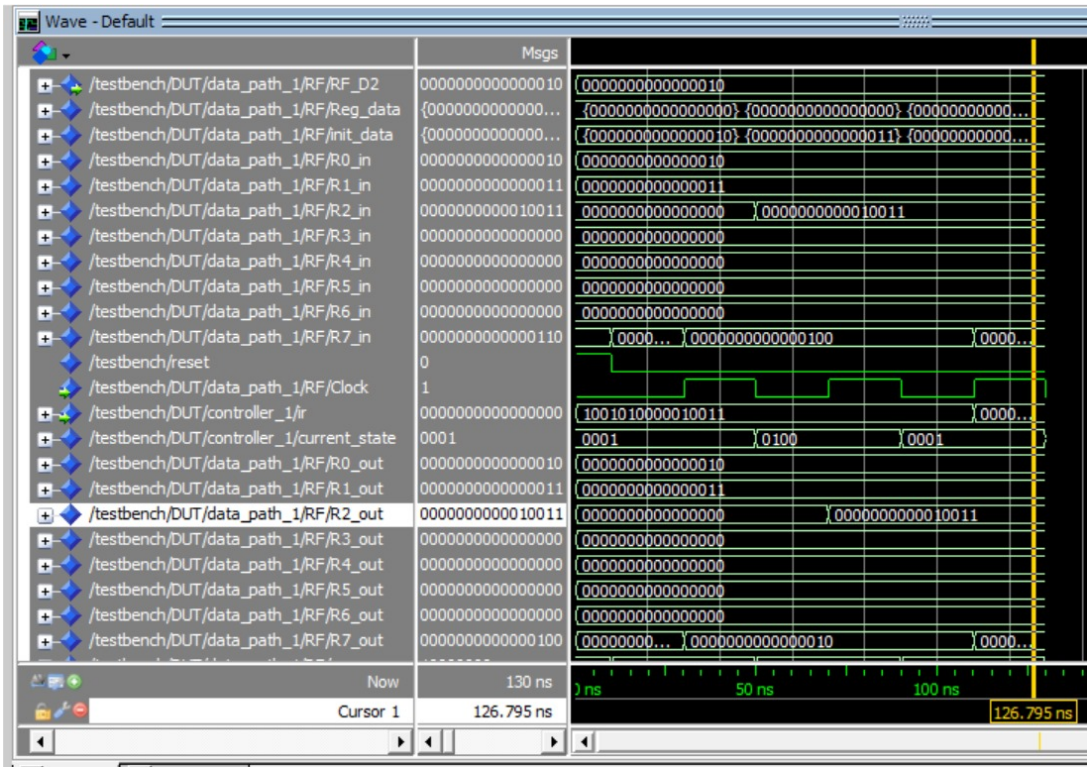

ORA R2,R0,R1



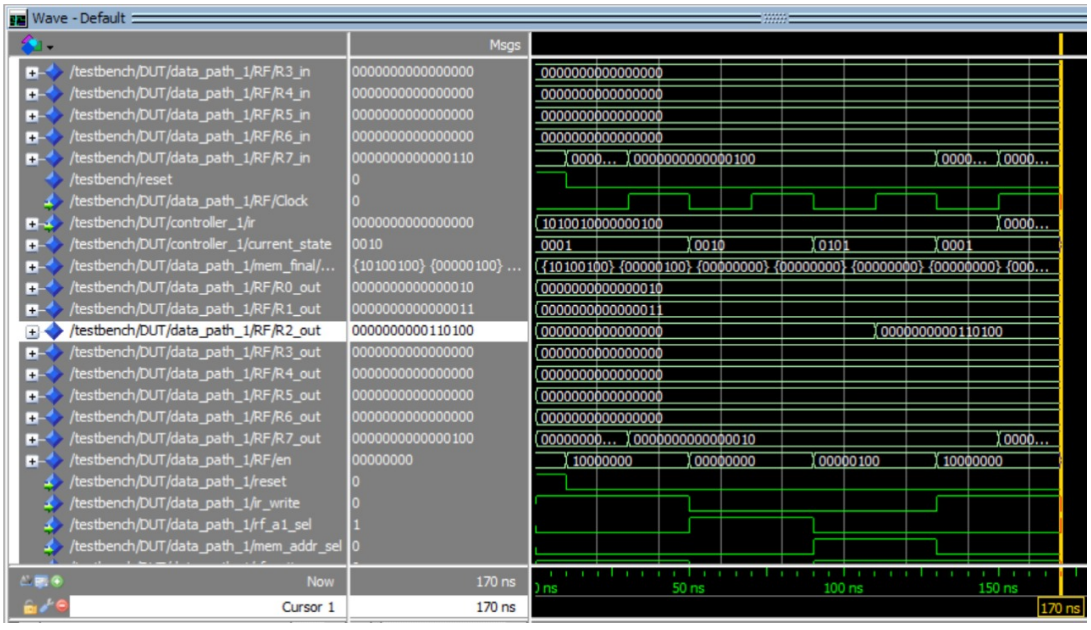

IMP R2,R0,R1



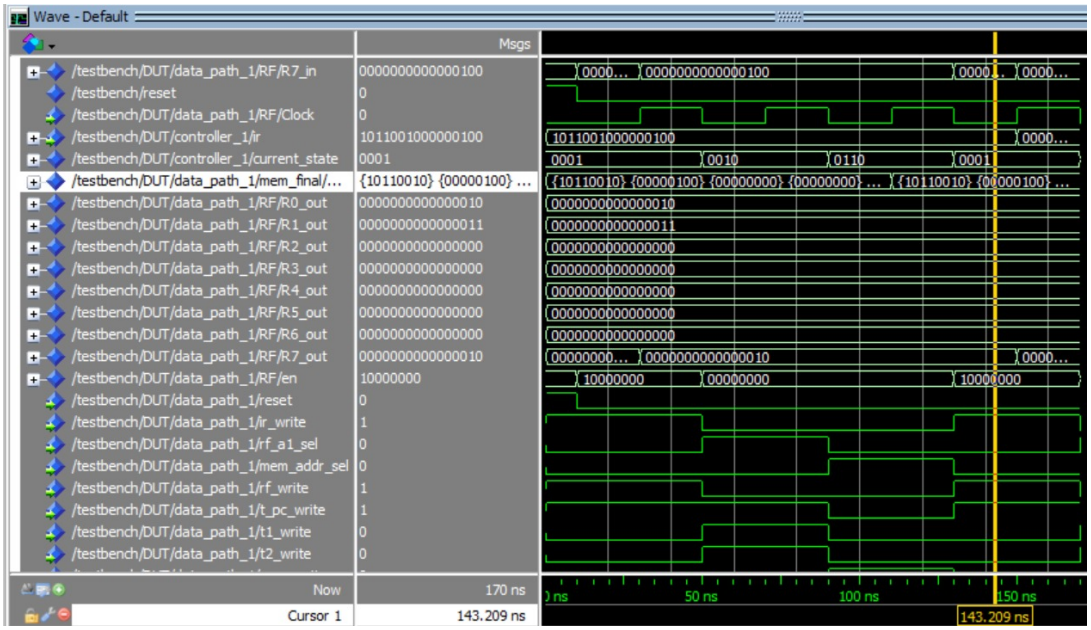

LHI R2,000010011



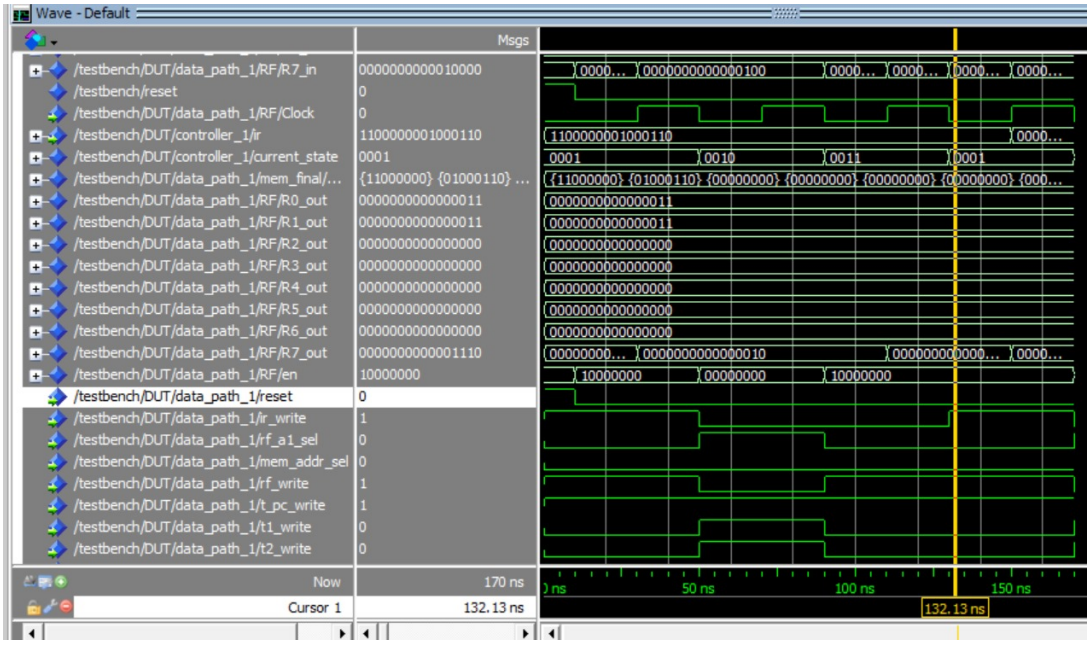

LLI R2,000010011



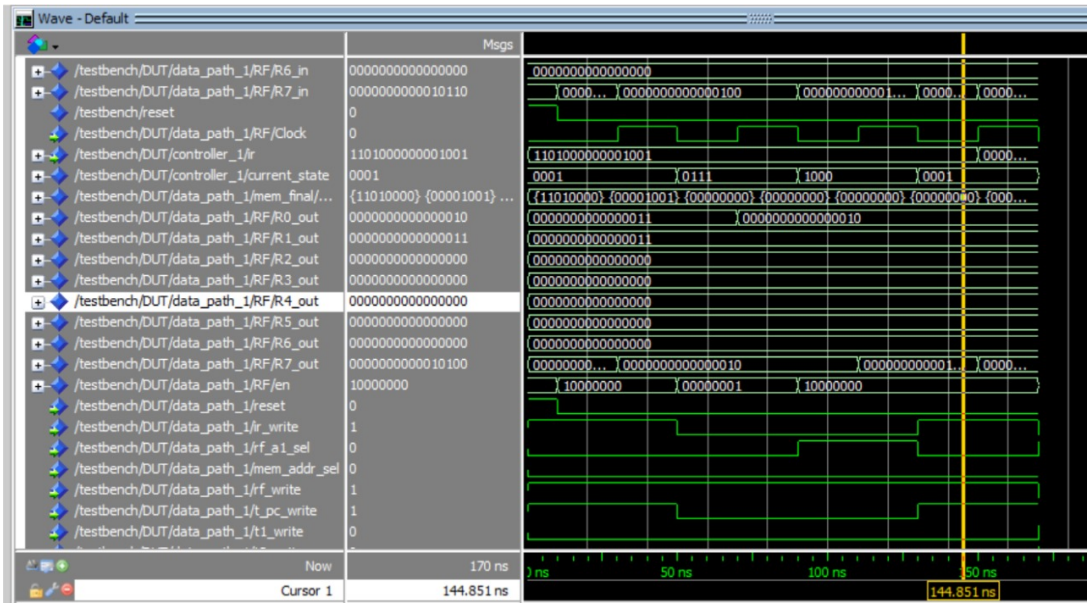

LW R2,R0,000100



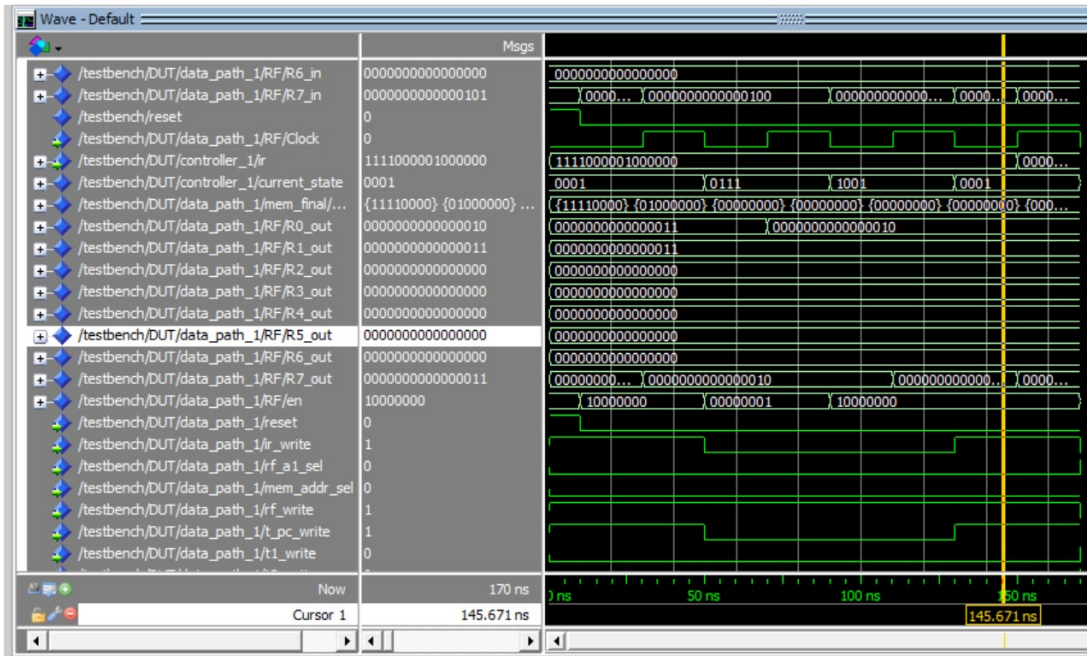

SW R2,R0,000100




BEQ R0,R1,000110




JAL R0,000001001




JLR R0,R1

# 9 Work Distribution

- **Arin Weling (22B1230):** Initial FSM design, FSM State Optimization, VHDL coding related to (ALU, Controller, Datapath), Automation of Memory Initialisation using Python, Debugging, Simulations.
- **Chiransh Somani (22B1202):** Optimizing FSM, VHDL coding related to (Left Shifter, Sign Extender,Temporary Registers, Multiplier in ALU).
- **Sathvik Reddy (22B3946):** FSM State Optimization, VHDL coding related to (Register File, Data Path, Memory, Registers), txt file implementation of Memory, Debugging, Simulations.
- **Tanish Raghute (22B3974):** FSM State Optimization, VHDL coding related to (Register File, Controller, Registers) Documentation, Report.