

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	2
2	Components	2
3	Operations	7
4	States	8
4.1	S1	8
4.2	S2	8
4.3	S3	9
4.4	S4	9
4.5	S5	10
4.6	S6	10
4.7	S7	11
4.8	S8	11
4.9	S9	11
4.10	Final FSM	12
5	Control bits	12
6	Datapath	13
7	Transitional Logic	16
8	Testing	17
8.1	CPU Definition	17
8.2	Memory Initialisation	18
8.3	Automating the Memory Initialisation	18
8.4	Results	19
9	Work Distribution	25

1 Introduction

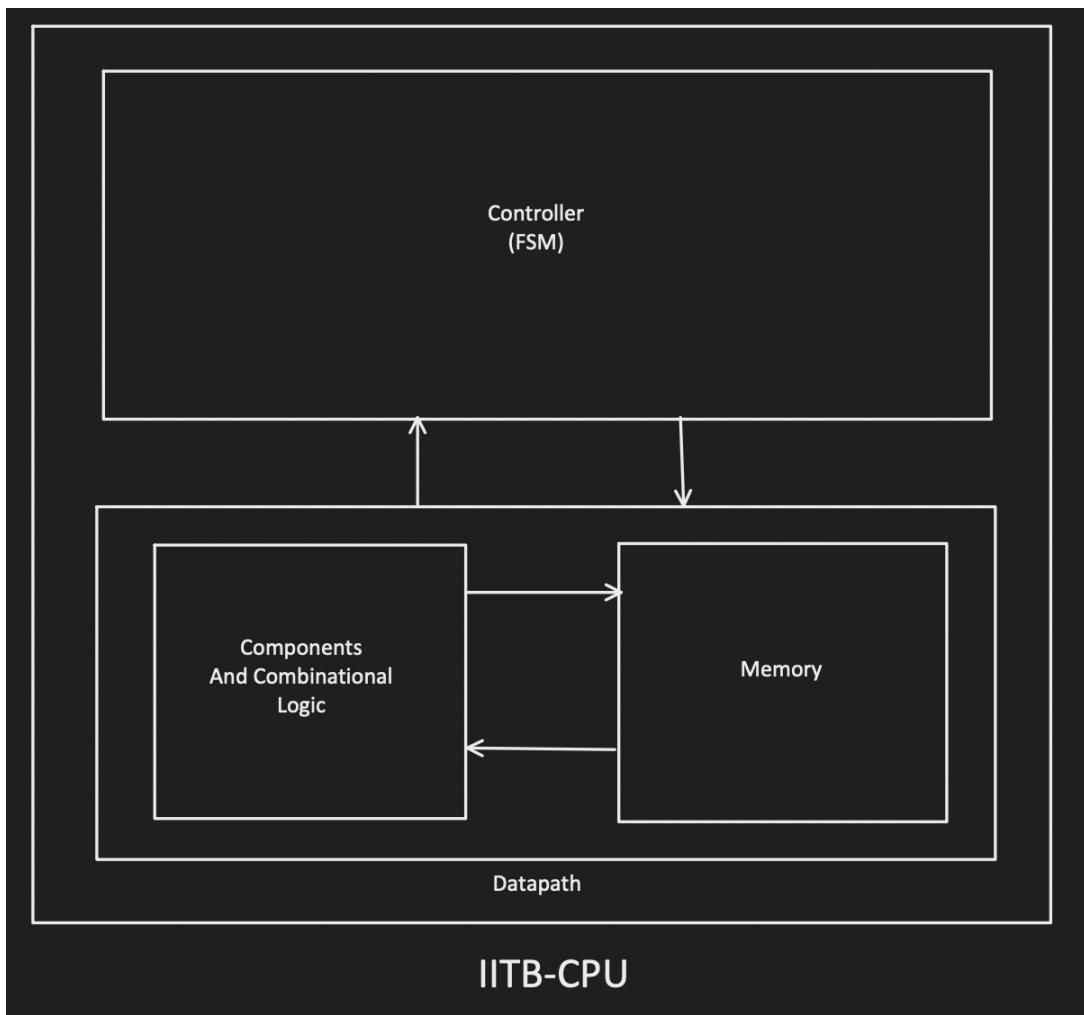
The IITB-CPU is an **8-register, 16-bit computer system**, i.e., it can process 16 bits at a time. It implements **14 different instructions**, as well as programs made using them. This design involves using **structural components, memory** and a **controller** (Moore FSM) with **9 total states**, all 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) (<https://github.com/Cove1/IITB-CPU/tree/main>).

2 Components

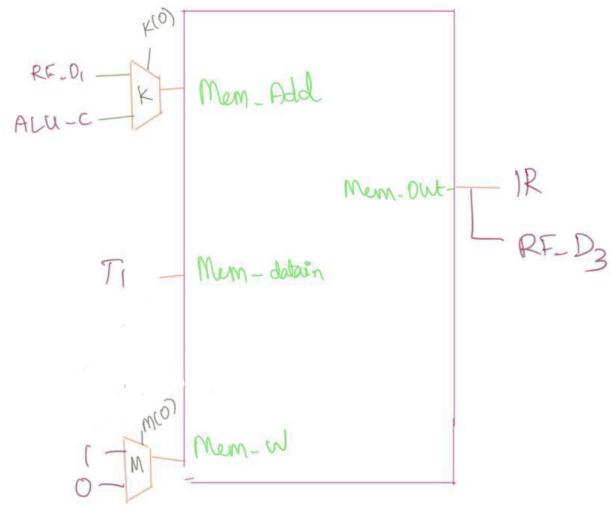
The IITB-CPU is an 8-register, 16-bit computer system. It has 8 general-purpose registers (R0 to R7). Register R7 is always stores Program Counter.

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



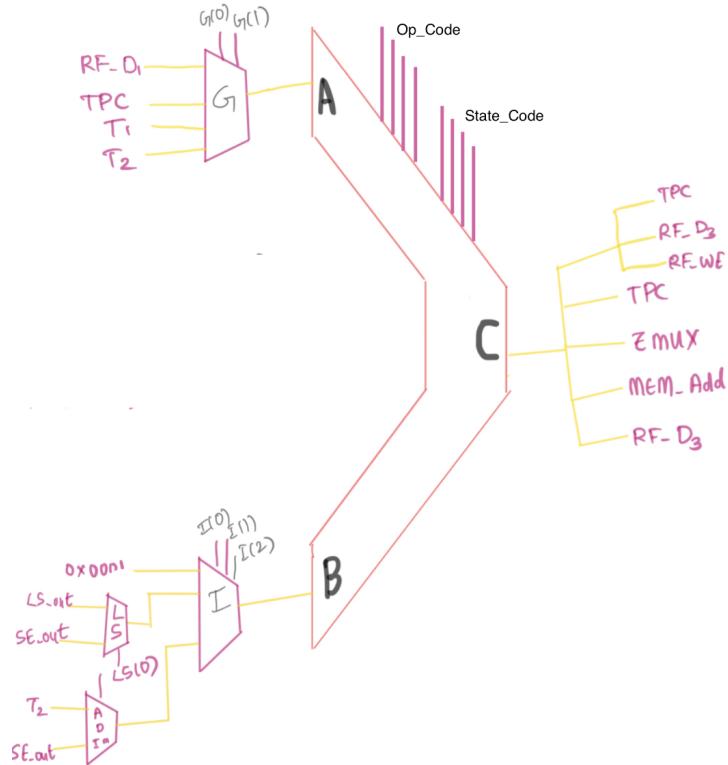
A brief overview of the individual components of the Datapath and their function is as follows:

- 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.

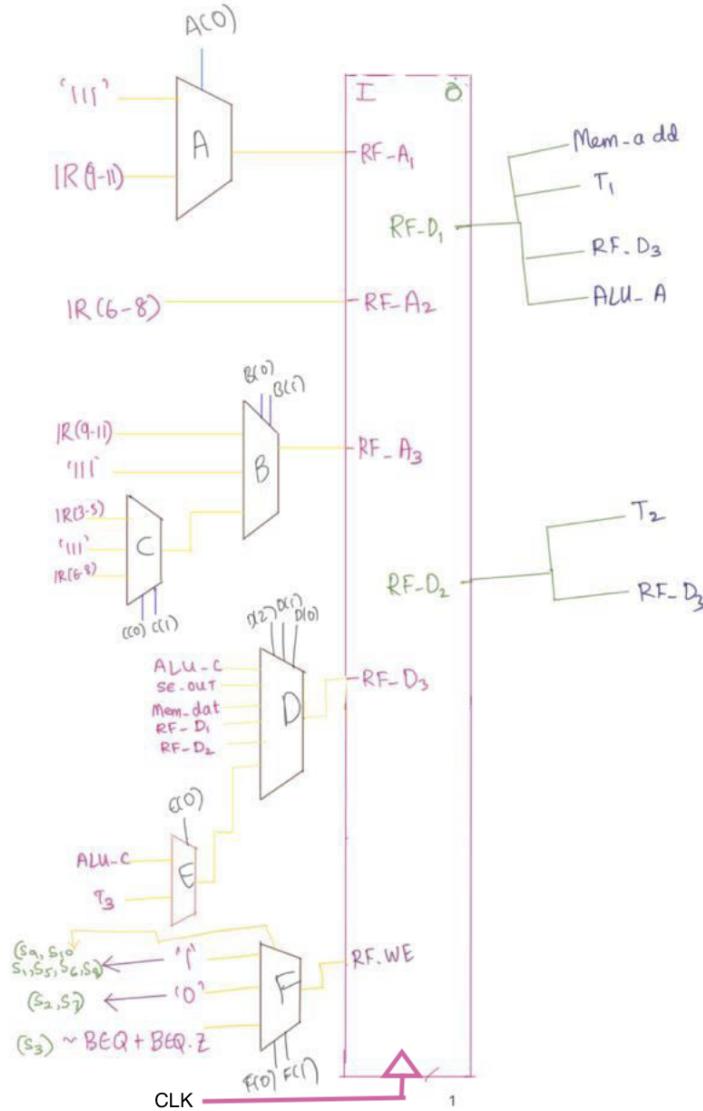
- Arithmetic and Logic Unit (ALU)



- 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)



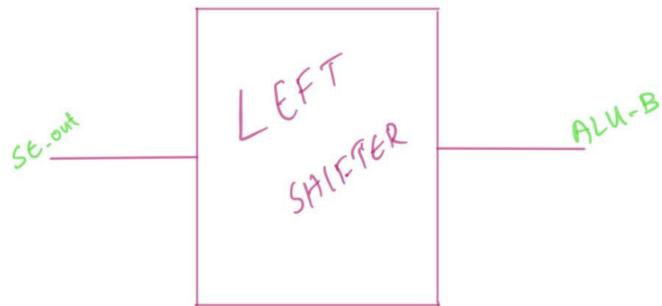
- A collection of 8, 16-bit PIPO registers.
- Write enable bits of each register dependant on the address fed to the RF and RF write enable bit.
- R7 Stores the Program Counter (PC).
- RF_A1, RF_A2 are address inputs for data reading, RF_A3, RF_D3 are inputs for data writing.
- RF_D1, RF_D2 are outputs to read data.

- Sign Extender (SE)

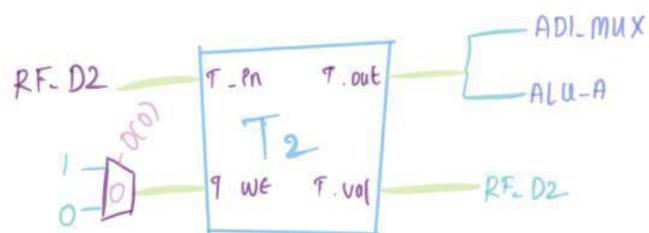
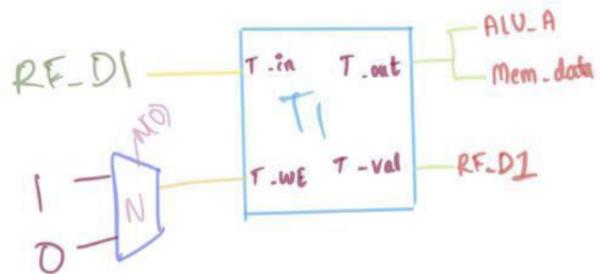


- Takes a 6-bit input and a 9-bit input, with a select line to choose between them, and outputs the 16-bit extended value.
- Has a select line for right and left 0-padding.

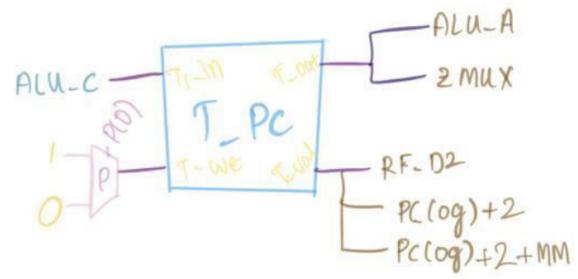
- Left Shifter (LS) : to double 16-bit values.



- T1, T2: Temporary Registers, operand related.



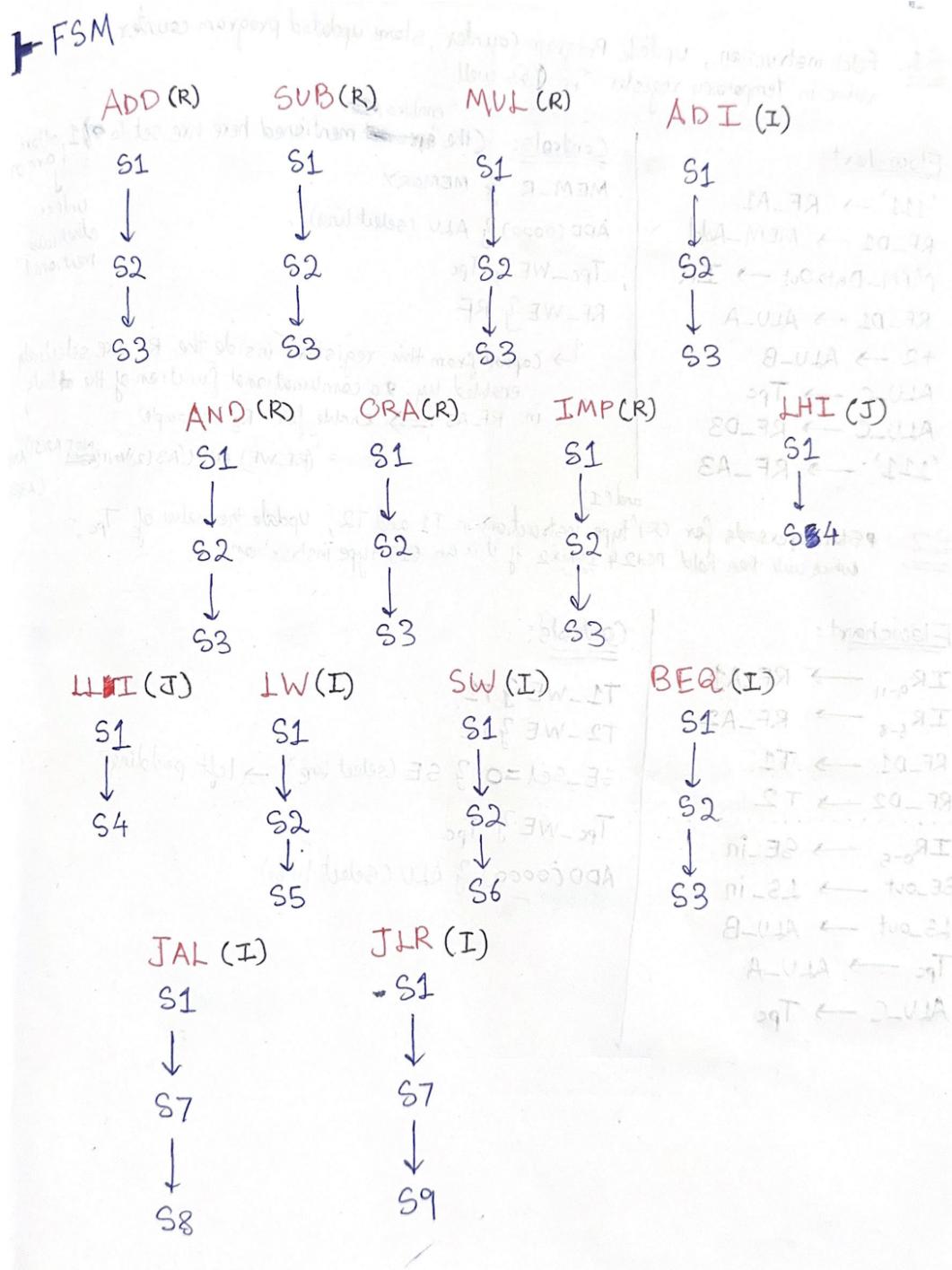
- **Tpc:** Temporary register, related to Program Counter operations.



- **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

S1: Fetch instruction, update Program Counter, store updated program counter value in temporary register Tpc as well

Flowchart:

$$\begin{aligned} '111' &\rightarrow RF_A1 \\ RF_D1 &\rightarrow MEM_Add \\ MEM_DataOut &\rightarrow IR \\ RF_D1 &\rightarrow ALU_A \\ +2 &\rightarrow ALU_B \\ ALU_C &\rightarrow Tpc \\ ALU_C &\rightarrow RF_D3 \\ '111' &\rightarrow RF_A3 \end{aligned}$$

Controls: (the controls mentioned here are set to 1, otherwise unless otherwise mentioned)

- MEM-R \exists MEMORY
- ADD(0000) \exists ALU (select lines)
- Tpc_WE \exists Tpc
- RF_WE \exists RF

L \hookrightarrow Control from this, registers inside the RF are selected by a combinational function of the bits in RF_A3 \Rightarrow Enable for R₅ (example)

$$= (RF_WE) \text{ AND } ((A3(2)) \text{ AND } \text{NOT } A3)$$

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

S2: Store operands for (R) type instructions in T1 and T2, update the value of Tpc, which will then hold PC+2 + IMM*2 if it is an (I) type instruction

Flowchart:

$$\begin{aligned} IR_{9-11} &\rightarrow RF_A1 \\ IR_{6-8} &\rightarrow RF_A2 \\ RF_D1 &\rightarrow T1 \\ RF_D2 &\rightarrow T2 \\ IR_{0-5} &\rightarrow SE_in \\ SE_out &\rightarrow LS_in \\ LS_out &\rightarrow ALU_B \\ Tpc &\rightarrow ALU_A \\ ALU_C &\rightarrow Tpc \end{aligned}$$

Controls:

$$\begin{aligned} T1_WE &\exists T1 \\ T2_WE &\exists T2 \\ SE_Sel = 0 &\exists SE \text{ (select line)} \rightarrow \text{left padding} \\ Tpc_WE &\exists Tpc \\ ADD(0000) &\exists ALU \text{ (select lines)} \end{aligned}$$

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

S3: In the execution phase, for all (R) type instructions, BEQ(I) and ADI(I), the updating of registers and operands in the ALU are controlled by ADI, BEQ, and Z bits, as well as select lines from MUXes or combinational functions fed to RF-WE (write enable).

Flowchart:

```

IR0-5 → SE-in
T1 → ALU-A
if (ADI == '1') then
    SE-out → ALU-B
else
    T2 → ALU-B
    if (Z == '1') then
        Tpc → RF-D3
    else
        ALU-C → RF-D3
    if (Z == '0' and ADI == '0') then
        IR3-5 → RF-A3
    elseif (Z == '0' and ADI == '1') then
        IR6-8 → RF-A3
    elseif (Z == '1' and ADI == '0') then
        '111' → RF-A3
    
```

Controls:

$$\begin{aligned} \text{SE-Sel} &= \overline{\text{BEQ}} + Z \cdot \overline{\text{BEQ}} \quad \text{RF} \\ \text{SE-Sel} &= '0' \quad \text{SE (Select lines)} \end{aligned}$$

$$\text{IR}_{12-15} \quad \text{(opcode)} \quad \text{ALU (select lines)}$$

$$\text{MEM_RF3} \leftarrow \text{taddr}$$

$$\text{EA_RF} \leftarrow \text{taddr}$$

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

S4: For LHI, LLI instructions, this state involves left extending or right extending immediate in (J) type instructions, and updating RegA with the value.

Flowchart:

```

IR0-8 → SE-in
SE-out → RF-D3
IR9-11 → RF-A3
    
```

Controls:

$$\text{SE-Sel} = \overline{\text{IR}(12)} \quad \text{SE}$$

$$\text{RF-WE} \quad \text{RF}$$

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

S5: This state involves computation of the Address in Load instruction and accessing the memory to update the value in the RF.

Flowchart:

```

IR0-5 → SE-in
SE-out → ALU-B
T2 → ALU-A
ALU-C → MEM-Add
MEM-DataOut → RF-D3
IR9-11 → RF-A3
  
```

Controls:

```

SE-Sel = '0' } SE
ADD ('cccc') } ALU (select lines)
MEM-R } MEMORY
RF-WE } RF
  
```

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

S6: This state, in the store instruction computes the address using RegB value stored in T2 and stores the RegA value in the Memory.

Flowchart:

```

IR0-5 → SE-in
SE-out → ALU-B
T2 → ALU-A
ALU-C → MEM-Add
T1 → MEM-DataIn
  
```

Controls:

```

SE-Sel = 'c' } SE
ADD ('cc0c') } ALU (select lines)
MEM-W } MEMORY
  
```

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

4.7 S7

S7: Used in JAL & JLR instructions, this state stores the program counter value in another register in RF.

Flowchart:

$$\begin{aligned} '111' &\rightarrow RF_A1 \\ RF_D1 &\rightarrow RF_D3 \\ IR_{9-11} &\rightarrow RF_A3 \end{aligned}$$

Controls:

$$RF_WE \uparrow RF$$

Stores the current PC value in JAL and JLR instructions.

4.8 S8

S8: For the JAL instruction, we use the value $PC + 2 + IMM_2$ stored (using T_{pc} for these instructions) and immediate. This value is stored in the RF.

Flowchart:

$$\begin{aligned} IR_{0-5} &\rightarrow SE\text{-in} \\ SE\text{-out} &\rightarrow LS\text{-in} \\ LS\text{-out} &\rightarrow ALU\text{-B} \\ T_{pc} &\rightarrow ALU\text{-A} \\ ALU\text{-C} &\rightarrow RF_D3 \\ '111' &\rightarrow RF_A3 \end{aligned}$$

Controls:

$$\begin{aligned} SE\text{-Sel} = '0' &\uparrow SE \\ ADD(0000) &\uparrow ALU \text{ (select lines)} \\ RF_WE &\uparrow RF \end{aligned}$$

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

4.9 S9

S9: For the JLR instruction, we directly update PC with the value stored in Reg B.

Flowchart:

$$\begin{aligned} IR_{6-8} &\rightarrow RF_A2 \\ RF_D2 &\rightarrow RF_D3 \\ '111' &\rightarrow RF_A3 \end{aligned}$$

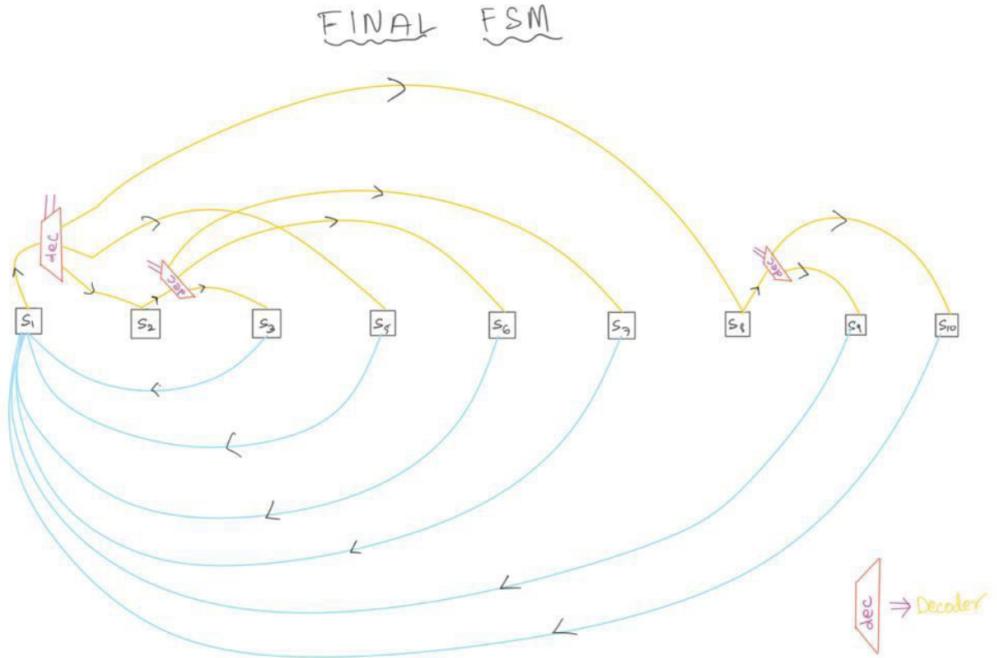
Controls:

$$RF_WE \uparrow RF$$

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		000000 000000 10	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
OUTPUT S										
C		TPC, RF_D3	TPC	Z MUX (0)	X	Mem_A dd	Mem_A dd	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 DECOD ER)	1	1	0	1	1	1
OUTPUTS										
RF_D1		Mem_dd, ALU_A	T1	X	X	X	X	RF_D3	ALU_A	X
RF_D2		X	T2	.	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_Dateln	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) (<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             end if;
43         when "1010" =>
44             if(current_state="0001") then
45                 next_state<="0010";
46             elsif (current_state="0010") then
47                 next_state<="0101";
48             elsif (current_state="0101") then
49                 next_state<="0001";
50             else
51                 next_state<="0001";
52             end if;
53         when "1011" =>
54             if(current_state="0001") then
55                 next_state<="0010";
56             elsif (current_state="0010") then
57                 next_state<="0110";
58             elsif (current_state="0110") then
59                 next_state<="0001";
60             else
61                 next_state<="0001";
62             end if;

```

```

63      end if;
64      when "1101" =>
65          if(current_state="0001") then
66              next_state<="0111";
67          elsif (current_state="0111") then
68              next_state<"1000";
69          elsif (current_state="1000") then
70              next_state<="0001";
71          else
72              next_state<="0001";
73          end if;
74      when "1111" =>
75          if(current_state="0001") then
76              next_state<="0111";
77          elsif (current_state="0111") then
78              next_state<"1001";
79          elsif (current_state="1001") then
80              next_state<="0001";
81          else
82              next_state<="0001";
83          end if;
84      when others =>
85          next_state<="0001";
86      end case;
87  end process;
88 state_transition:process(current_state,next_state,clock)
89 begin
90     if(clock='0' and clock'event) then
91         current_state<=next_state;
92     end if;
93  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;

9  architecture bhv of CPU is
10 component controller is
11     port (ir:in std_logic_vector(15 downto 0);
12             clock,zin:std_logic;
13             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;
14             alu_a_sel,alu_b_sel,rf_a3_sel:out std_logic_vector(1 downto 0);
15             rf_d3_sel:out std_logic_vector(2 downto 0);
16             out_state,out_op:out std_logic_vector(3 downto 0));
17 end component controller;
18
19
20 component DataPath is
21 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;
22             alu_a_sel,alu_b_sel,rf_a3_sel:in std_logic_vector(1 downto 0);
23             rf_d3_sel:in std_logic_vector(2 downto 0);
24             clock: in std_logic;
25             out_state,out_op:in std_logic_vector(3 downto 0);
26
27
28             ir_data: out std_logic_vector(15 downto 0);
29             zero_flag:out std_logic);
30
31 end component DataPath;
32
33
34 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;
35 signal alu_a_sela,alu_b_sela,rf_a3_sela: std_logic_vector(1 downto 0);
36 signal rf_d3_sela: std_logic_vector(2 downto 0);
37
38 signal out_statea,out_opa:std_logic_vector(3 downto 0);
39 signal ir_dataa: std_logic_vector(15 downto 0);
40 signal zero_flaga: std_logic;
41 begin
42
43 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_
44             alu_a_sela,alu_b_sela,rf_a3_sela,rf_d3_sela,out_statea,out_opa);
45
46 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_
47             alu_a_sela,alu_b_sela,rf_a3_sela,rf_d3_sela,clock,out_statea,out_opa,
48             ir_dataa,zero_flaga);
49
50 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.

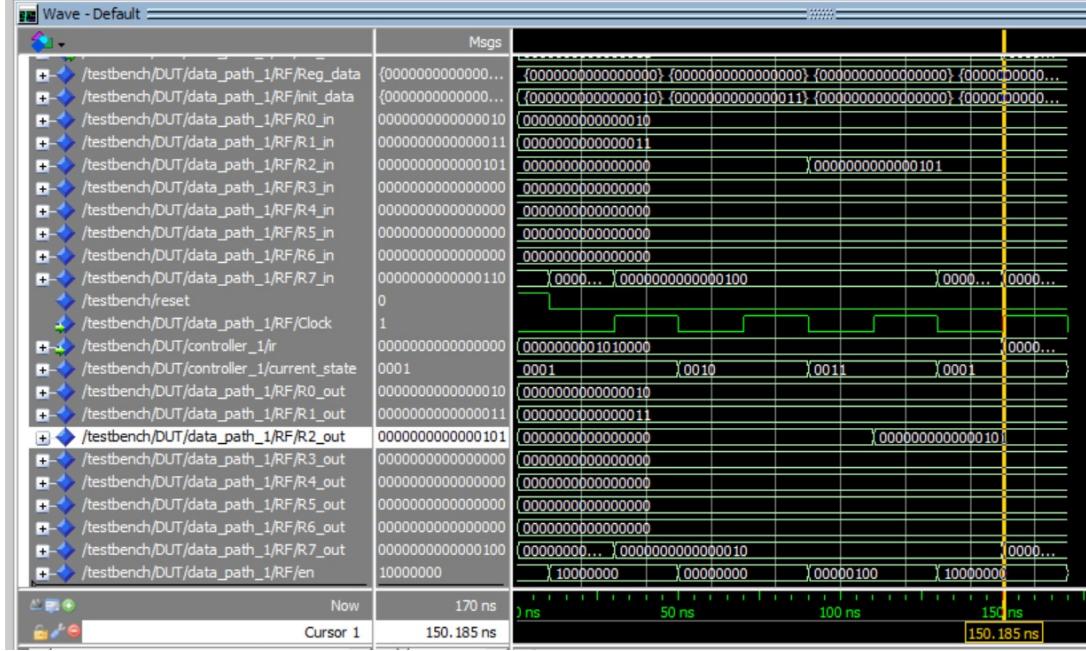
The code related to this, as well as the txt files can be found at the Github repository linked [here](#)

```
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
```

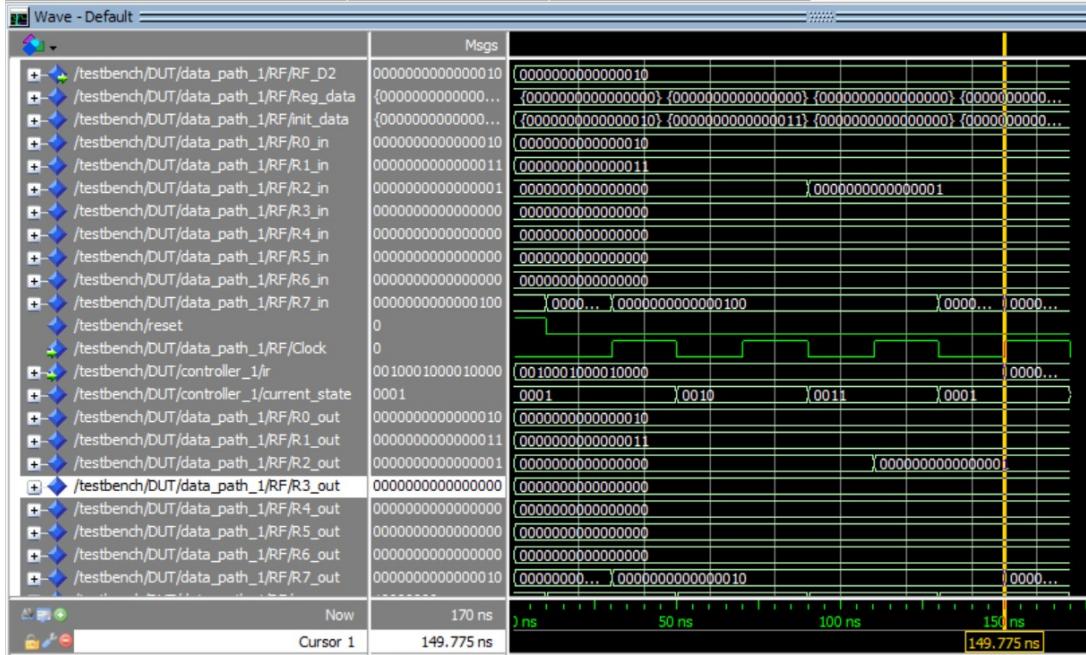
Figure 1: User Interface for the initialisation process

8.4 Results

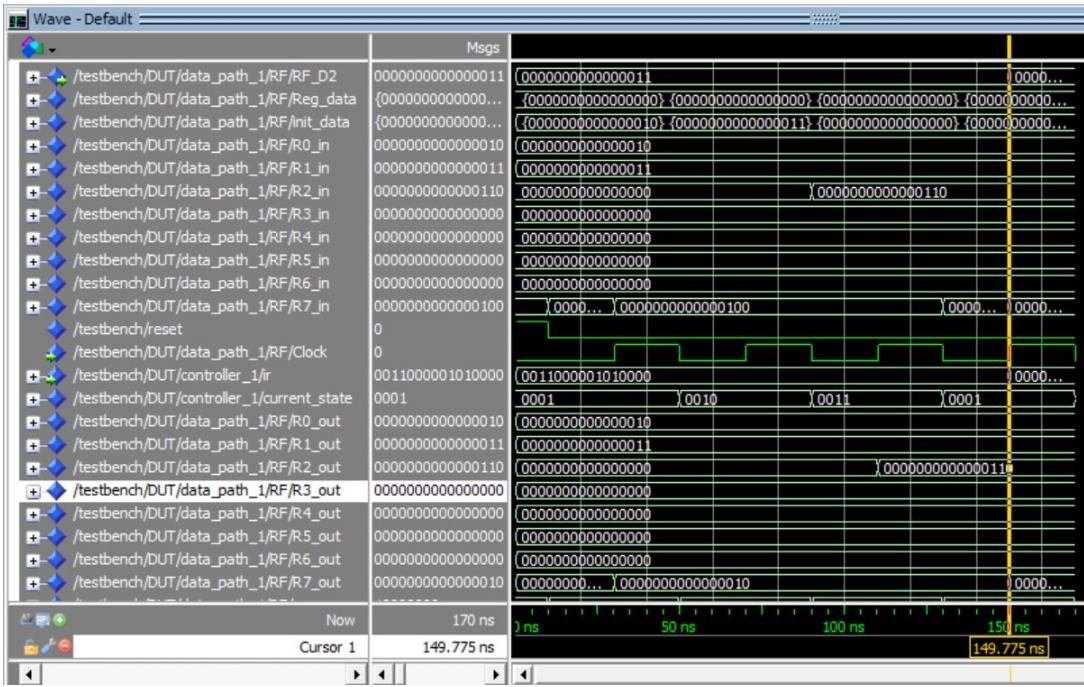
The CPU successfully carried out all instructions individually, which can be seen in the following simulation results:



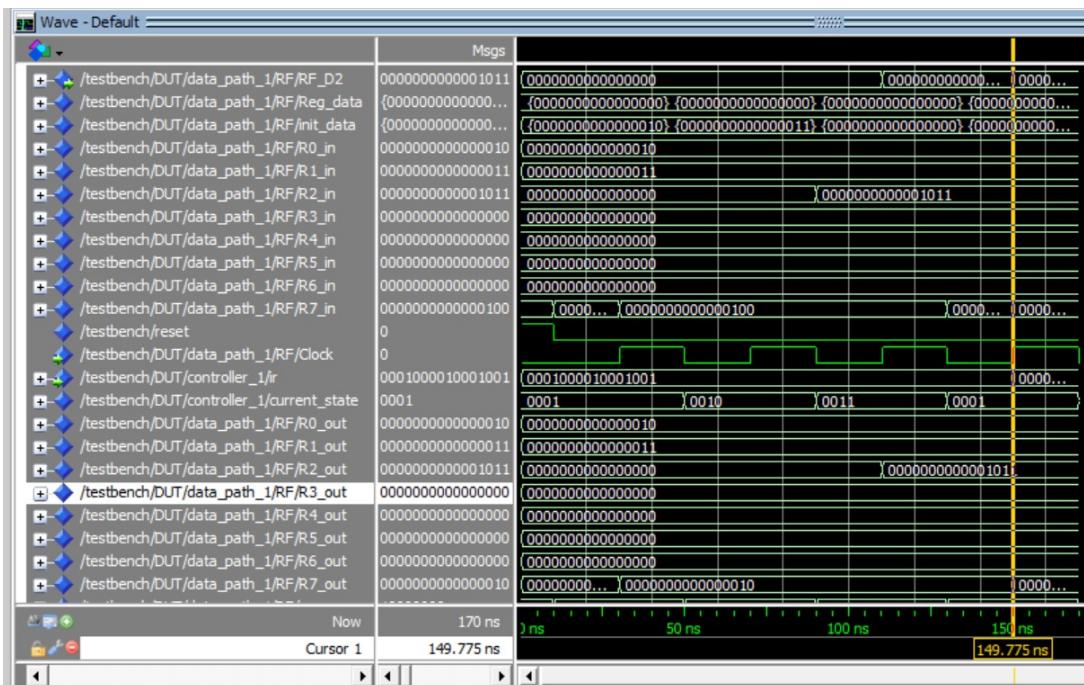
ADD R2,R0,R1



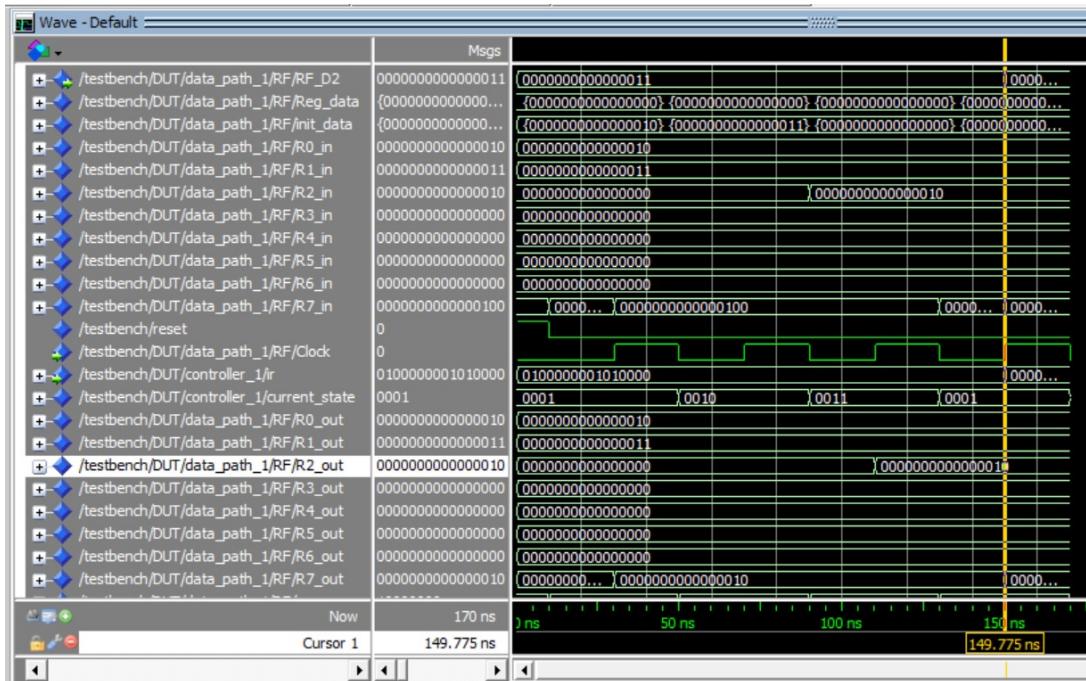
SUB R2,R0,R1



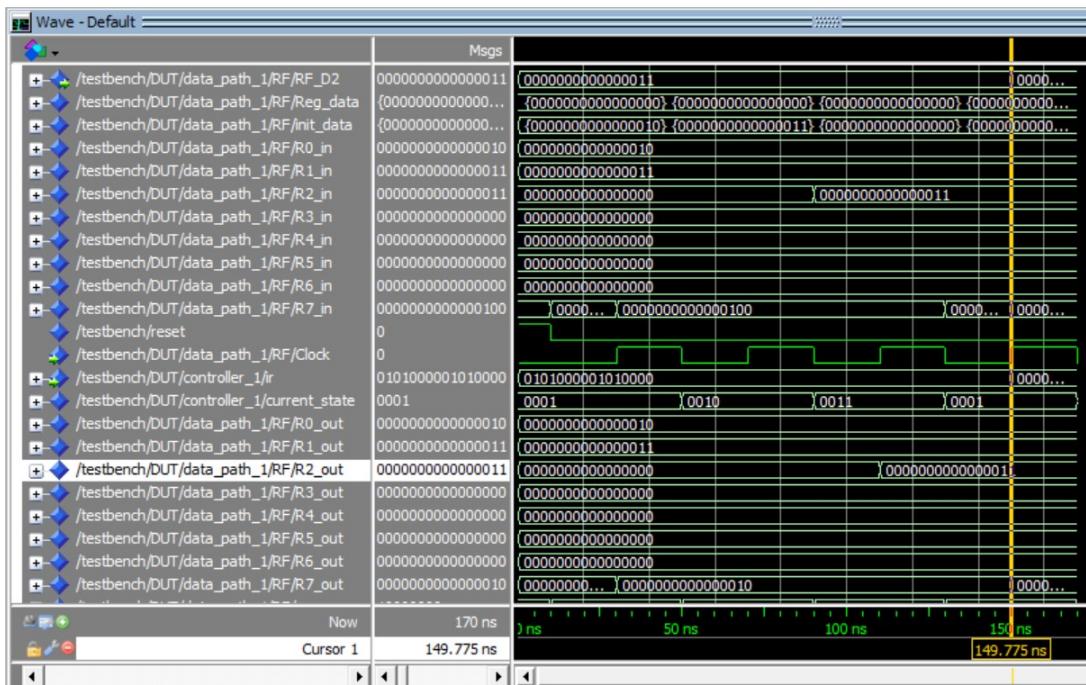
MUL R2,R0,R1



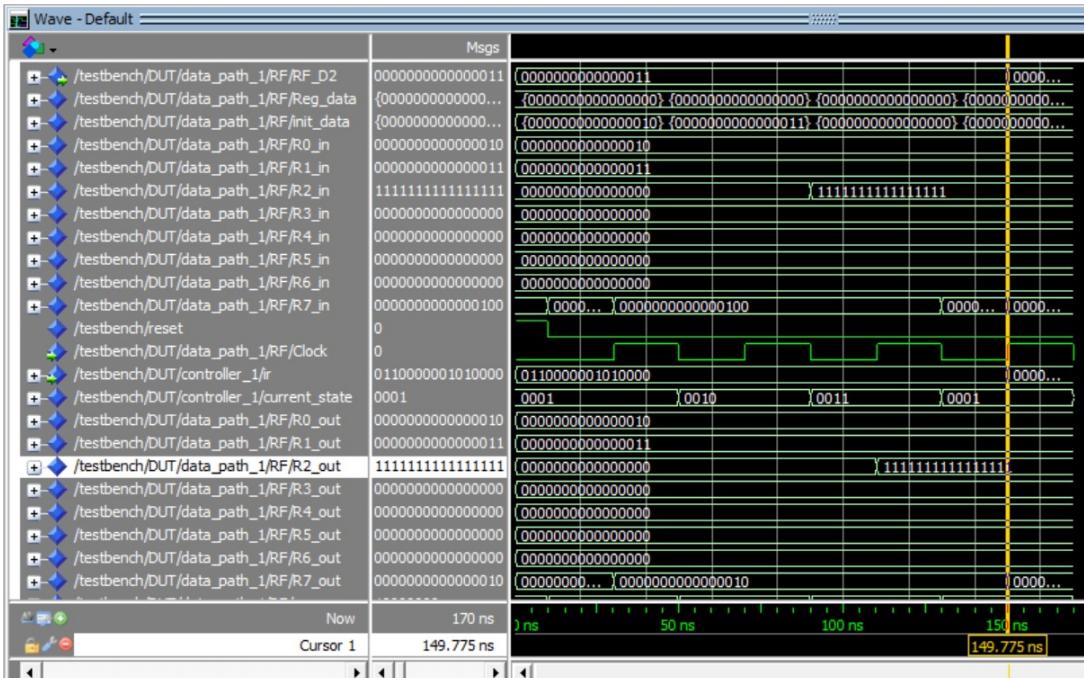
ADI R2,R0,001001



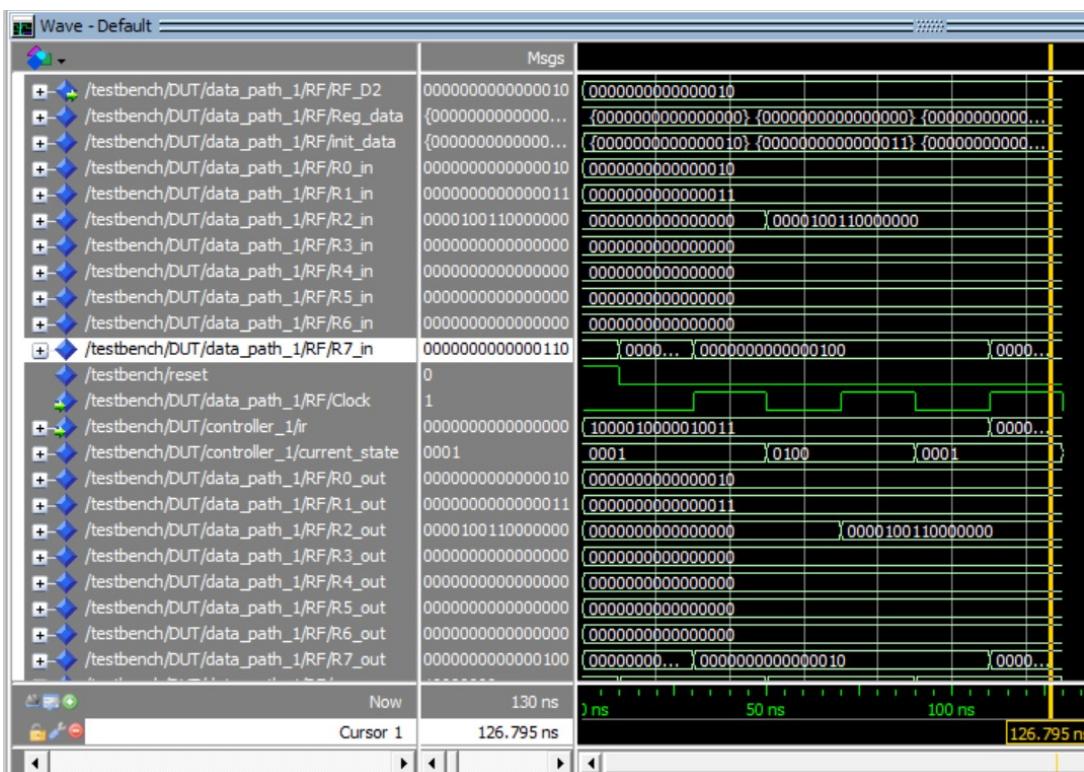
AND R2,R0,R1



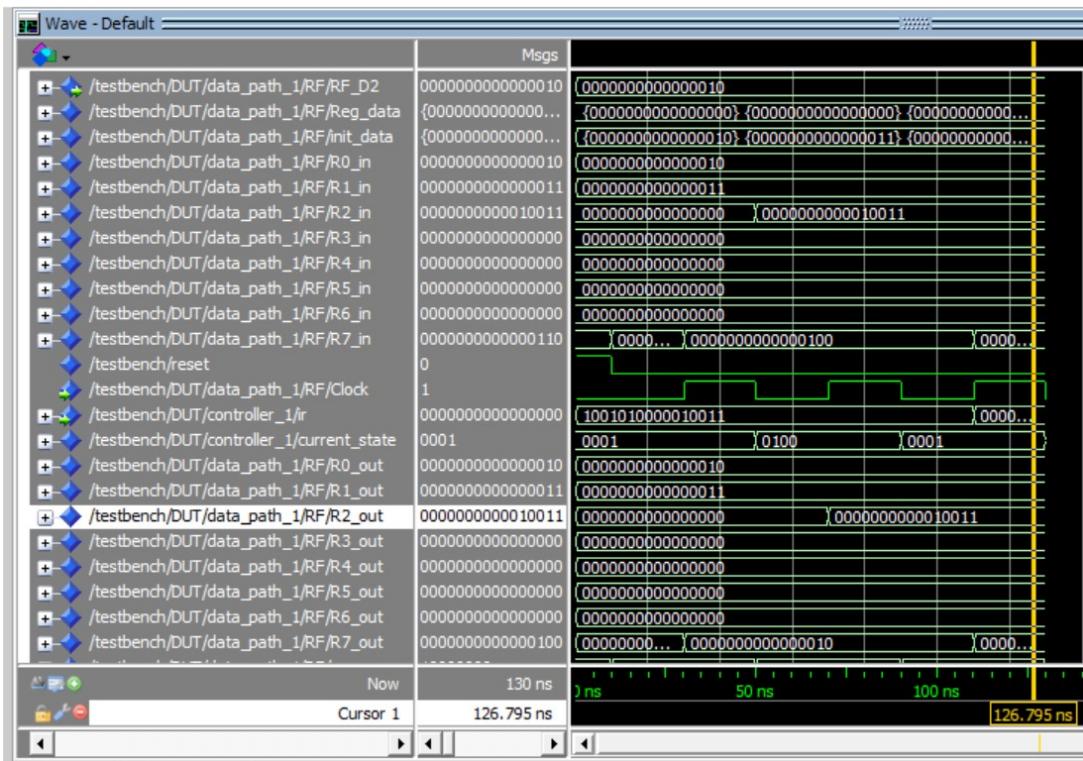
ORA R2,R0,R1



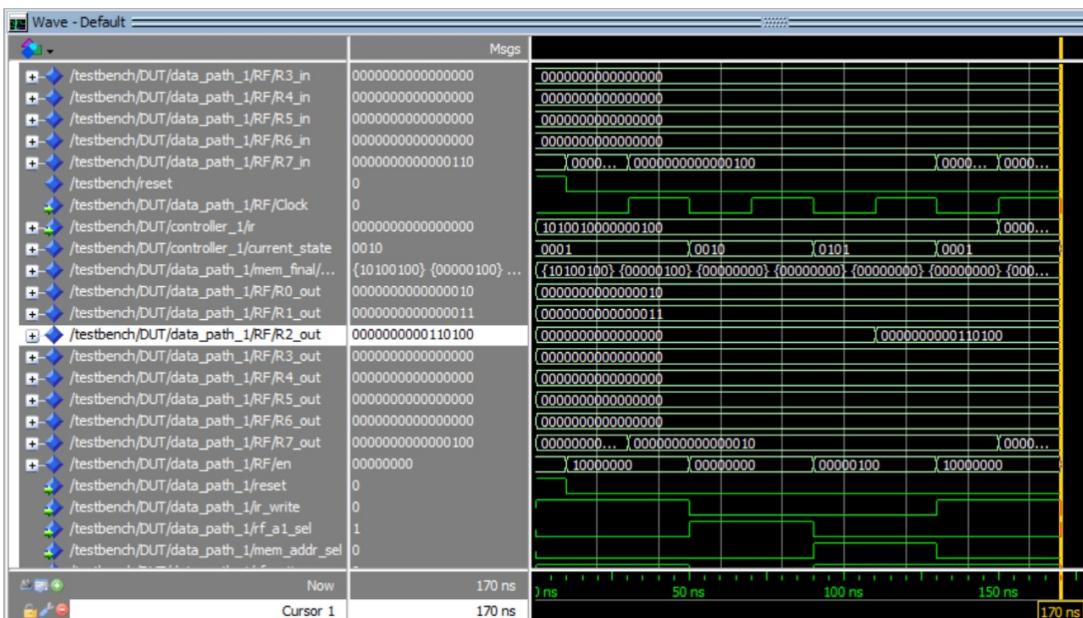
IMP R2,R0,R1



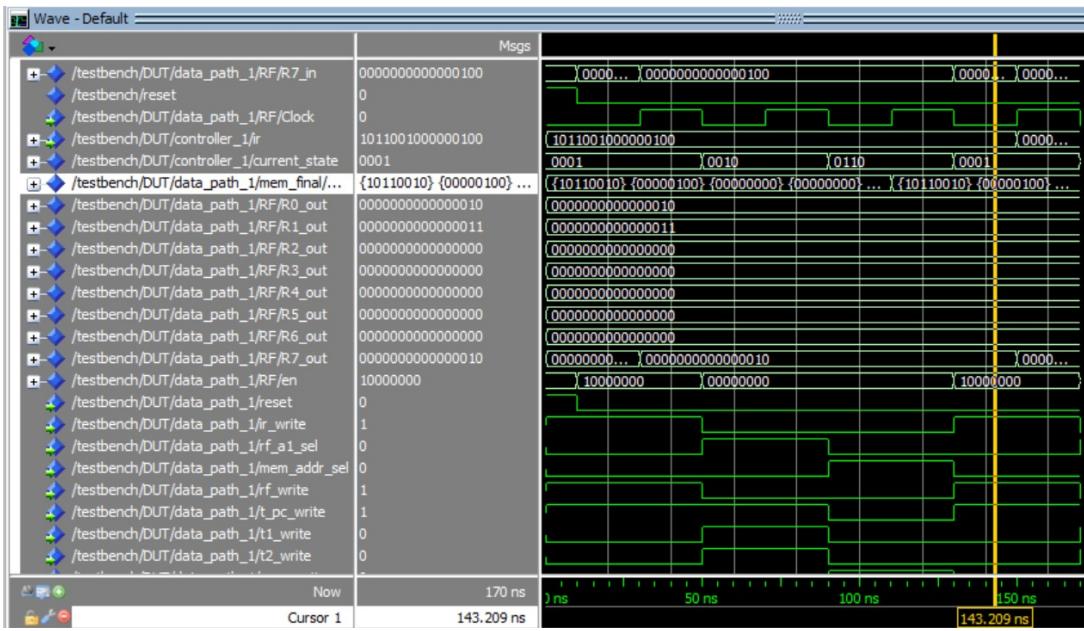
LHI R2,000010011



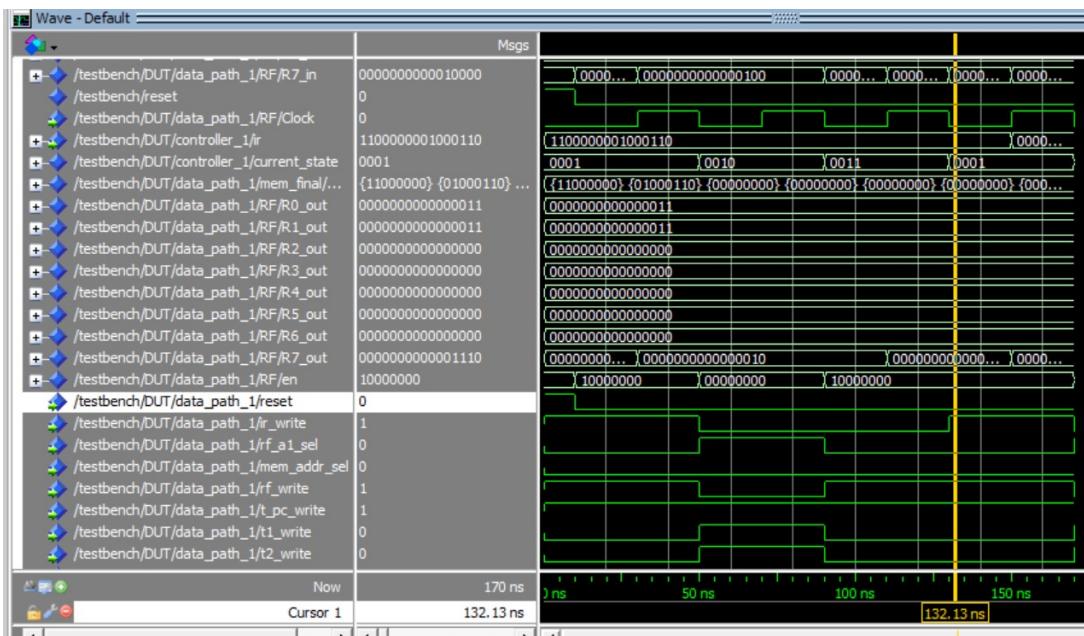
LLI R2,000010011



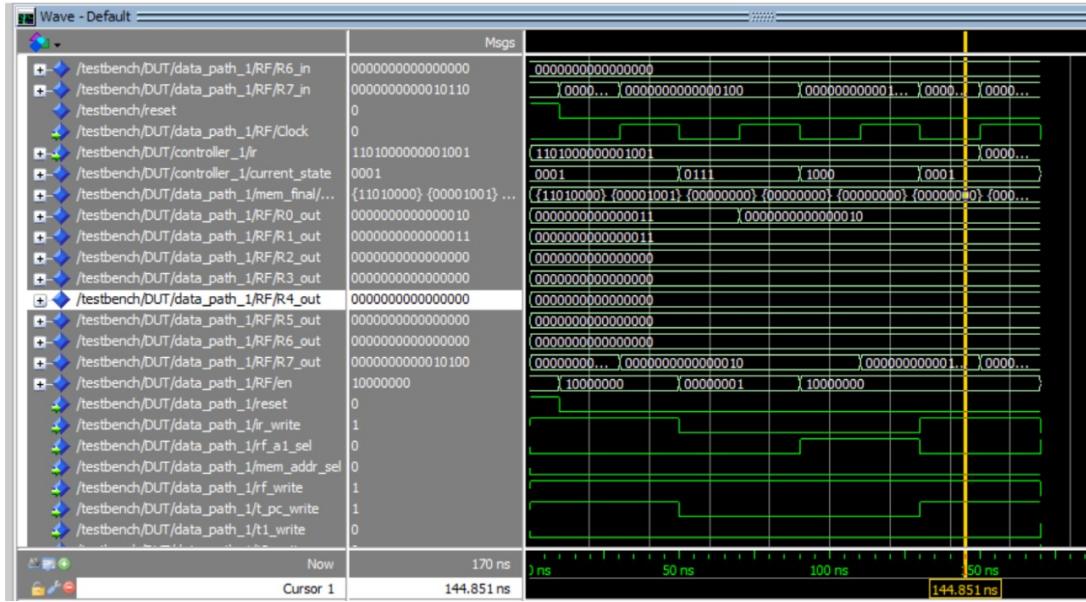
LW R2,R0,000100



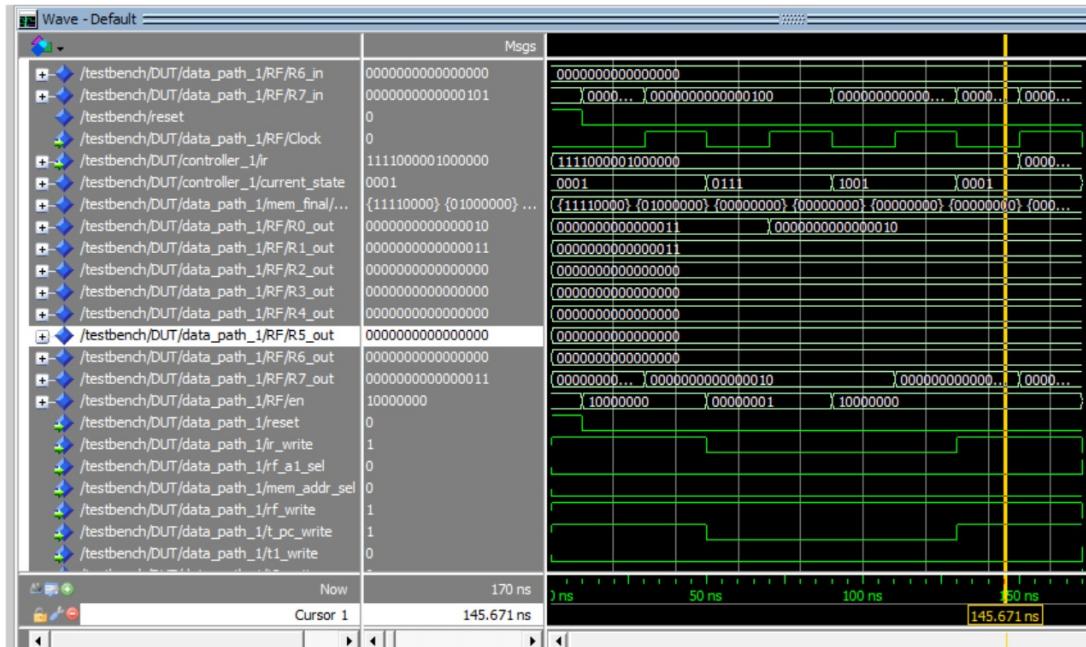
SW R2,R0,0000100



BEQ R0,R1,0000110



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.