Indian Institute of Technology Bombay
Department of Electrical Engineering
EE-224: Digital Design

IITB-CPU Design

Team ID: 3

Saarthak Krishan (22B3959) Aniket Patel (22B3981)
Sanat Agrawal (22B3919) Utkarsh Prakash (22B3928)

From our learnings from Digital Systems course and the current state technological landscape, it is quite evident that digital circuits are an indispensable part of human life. Hence, we as a team, tried to venture in the domain of digital electronics, by making a full-fledged Central Processing Unit (CPU). This report offers comprehensive details of our project, wherein we executed the project of designing and implementing a 16-bit CPU utilizing VHDL:

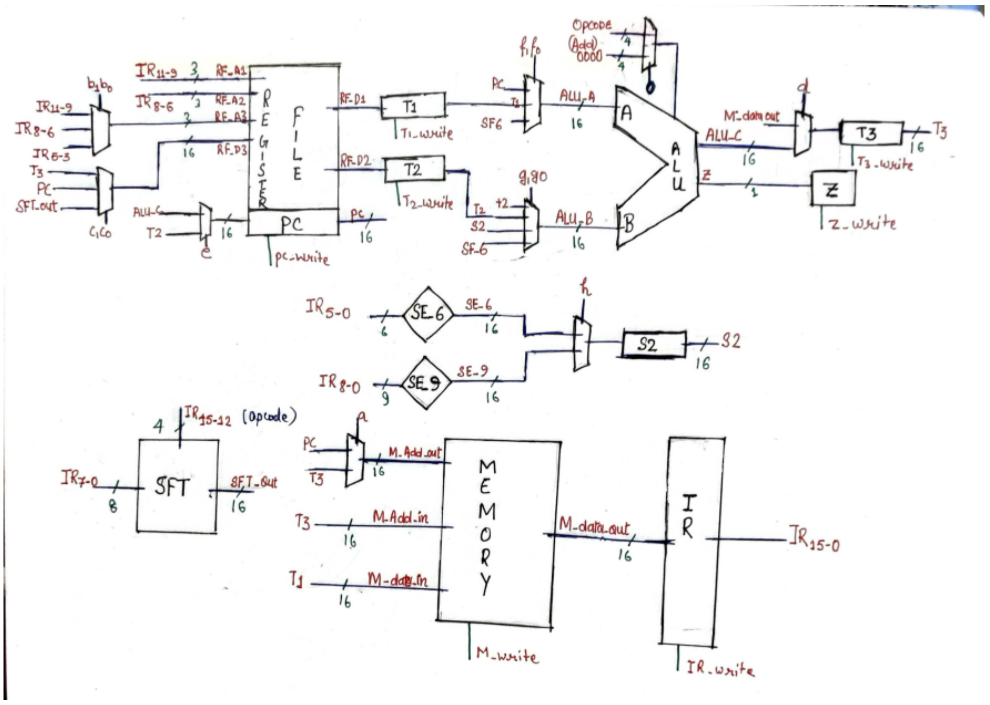
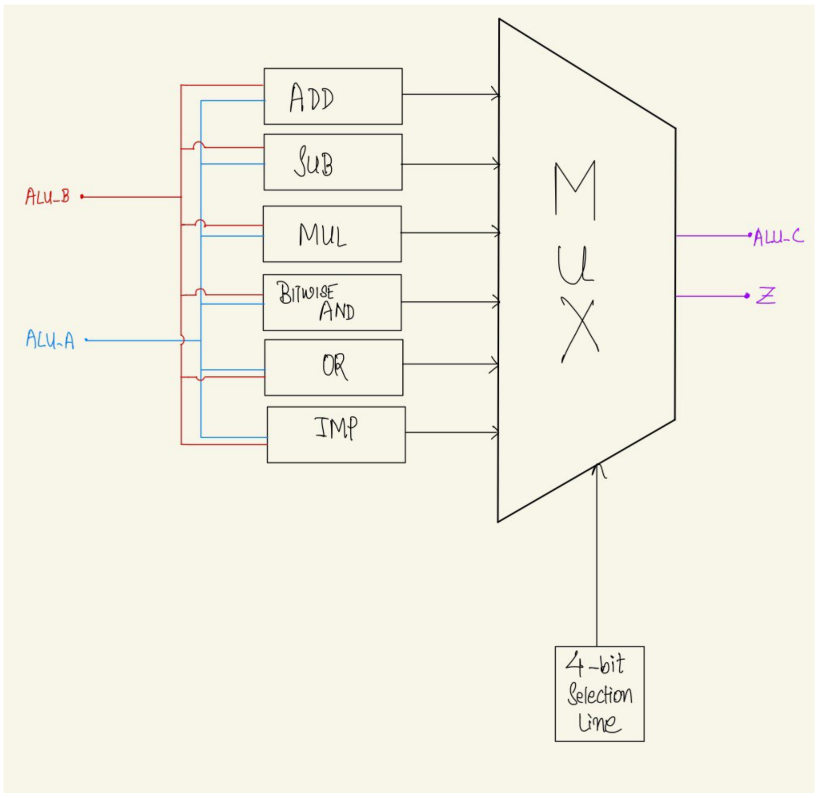


Fig: Overall Datapath of the CPU

"<u>16 bit</u>" means that it has 16 address lines. A 16 bit microprocessor is having 16 bit register set. It have 16 address and data lines to transfer address and data both.

COMPONENTS:

Arithmetic Logic Unit(ALU):



I think the main process block is the ALU. It takes a 4 bit control signal to pick various operators that we choose for the instruction.
"Operators" are pre-defined entities that are selected by the output of the control MUX.
ALU entity is a function within a function, where the operators are the subfunctons (not structural components).

OPERATORS:

- These are component blocks which are fed into the input of a custom made 6 x 1, 16 bit-wide MUX.
- Our implementation of these operators is in the form of functions, that are called upon by the ALU when needed.
- But this is true for only 6 out of 9 (basic) operators, that are, ADD, SUB, MUL, BITWISE AND, & LOGICAL IMPLICATION.
- Operators SE_6, SE_9, S2 & SFT exist **outside** of ALU, as per our instructions dataflow.
- **SE_n:** SE's are sign extenders which take a n-bit input and output the 16-bit equivalent.
- **S2:** Multiplies the 16-bit input by 2, i.e., shifts the input by a single bit.
- **SFT:** Shifts 8 bits left or right of the input digital signal. This component is used for implementing LLI and LHI instructions. It employs a 2x1 16-bit

MUX. The selection bit is the LSB of the OPCODE (as LLI/LHI differ at LSB 1001/1000).

The operands are fed into ALU (ALU_A and ALU_B) from the register file. ALU outputs two things: the operators out
put and flags.

FLAGS:

- Z: becomes HIGH when the loaded operands are equal.
- C:"Overflow"

Port Mapping:

- ALU_A: #1 input port of ALU
- ALU_B: #2 input port of ALU
- ALU_C: output port
- Z: Indicating zero flag

Register File:

Register file is like the cache of a processor, storing the information about the instructions, operator and operands (loaded from the memory and instruction register) & the result of ALU (for future instructions or to store it back to the memory). It consists of eight 16-bit registers, structurally bundled up.
The eight register is the PC (the instruction pointer).
<u>A n-bit register is a set of n parallel d flip-flops, storing a n-bit binary number.</u>

All the registers in RF are connected to two 8x1 16-bit MUXs and a 1x8 16-bit DEMUX.
A MUX selects the requested register of RF from where the data is to be loaded, whereas a DEMUX selects the requested register of RF in which the data is to be written.
Storing any data into the register file can be done by first injecting the address of a register (R0 to R7) in RF_A3 dataline then the data (which is to be stored) in RF_D3.
Accessing a data from a register of RF can be done by first injecting the address of the register (R0 to R7) in either RF_A1 or RF_A2 dataline, then the data is correspondingly accessed at RF_D1/RF_D2.
Instruction format:

Opcode	Register A (RA)	Register B (RB)	Register C (RC)	Unused	Condition (CZ)
(4 bit)	(3 bit)	(3-bit)	(3-bit)	(1 bit)	(2 bit)

This is a typical instruction format consists of addresses of Opcode, REG_A, REG_B, REG_C, Immediate, Condition bit. The address index starts from right to left, i.e., the condition(CZ) will have address 1-0, and so on. The instruction format is stored in a register in RF.

T1, **T2** and **T3** are exclusive registers that will be used as variables in data pipeline.

Port Mapping:

- RF_A1 - port 1 for the address of a register to be loaded from
- RF_A2 - port 2 for the address of a register to be loaded from
- RF_A3 - port for the address of the register to be updated
- RF_D1 - port for loading data from register in register file
- RF_D2 - port for loading data from register in register file
- RF_D3 - port for writing data in the register file
- RF_ write- port for write enable

TEMPORARY REGISTERS:

These registers are used as intermediate components to prevent simultaneous reading and writing of other components. Hence, we employ three temporary registers,i.e, T1, T2 & T3.

Port Mapping:

- Tn_In: Input port of nth temp register
- Tn_out: Output port of nth temp register
- Tn_write: Enable pin of nth temp register to perform write operation in it.

MEMORY & IR:

Memory is an array having 2^16 storage bits. It will store the instructions/instruction set and the ALU output.

Instruction Register: Another dedicated register, used to store the instruction that needs to be worked on, loaded from the memory.

Port Mapping(Memory):

- M_Add_in- Port for address of memory to be loaded in
- M_Data_in- Port for data to be loaded in memory
- M_Add_out- Port for address of memory to be loaded from
- M_Data_out- Port for data to be loaded from memory
- M_Write- Memory write enable

Synchronous and asynchronous operations:

ALU, SE_6, SE_9, SFT, MUXs and DEMUXs are **asynchronous** to the clock signal and operate as per the dataflow.

(Memory and RF)’s read and write operations happen at clock pulses. Registers of RF operate at rising edge of clock while the read/write operations of Memory happen at falling edge.

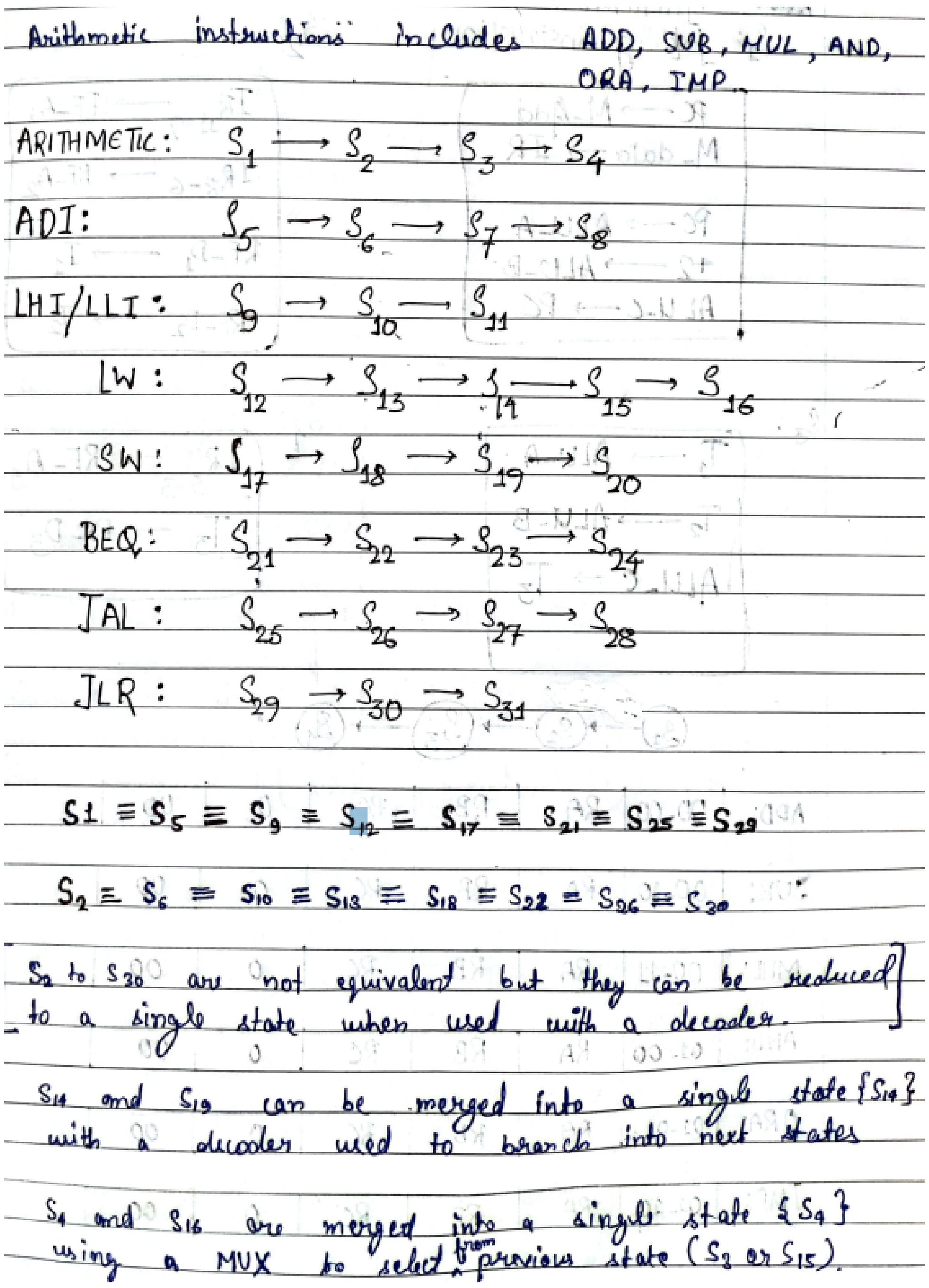
CPU:

This is the top-level entity of our project file, that serves as the structural definition of a 16-bit CPU implemented in VHDL. It encompasses the functionality of a basic processor architecture, coordinating the operation of key components to execute instructions and perform data processing tasks. It serves as the highway for the input and output signals of the associated component.

PORT MAPPING:

- PG_ADD:Used to specify the address of the memory, where the specific instruction from the instruction.txt is to be stored.
- PG_DATA: Used to feed the data (instruction) which is to be stored in the memory.
- PG_Write: Works as an enable, indicating when to write the entire instruction set on the memory.
- Clk: used to provide clock signal to the clock-synchronous components of the CPU.
- Reset: Resets every component of the CPU
- State: Used by the user to read the current state of the FSM that the CPU is in.
- IR: It outputs the current instruction loaded in Instruction Register.
- ALU_out: port for showing the output of the ALU

STATE WISE EXPLANATION:



Arithmetic instructions includes ADD, SUB, MUL, AND, ORA, IMP.

ARITHMETIC: $S_1 \longrightarrow S_2 \longrightarrow S_3 \longrightarrow S_4$

ADI: $S_5 \longrightarrow S_6 \longrightarrow S_7 \longrightarrow S_8$

LHI/LLI: $S_9 \longrightarrow S_{10} \longrightarrow S_{11}$

LW: $S_{12} \longrightarrow S_{13} \longrightarrow S_{14} \longrightarrow S_{15} \longrightarrow S_{16}$

SW: $S_{17} \longrightarrow S_{18} \longrightarrow S_{19} \longrightarrow S_{20}$

BEQ: $S_{21} \longrightarrow S_{22} \longrightarrow S_{23} \longrightarrow S_{24}$

JAL: $S_{25} \longrightarrow S_{26} \longrightarrow S_{27} \longrightarrow S_{28}$

JLR: $S_{29} \longrightarrow S_{30} \longrightarrow S_{31}$

$S_1 \equiv S_5 \equiv S_9 \equiv S_{12} \equiv S_{17} \equiv S_{21} \equiv S_{25} \equiv S_{29}$

$S_2 \equiv S_6 \equiv S_{10} \equiv S_{13} \equiv S_{18} \equiv S_{22} \equiv S_{26} \equiv S_{30}$

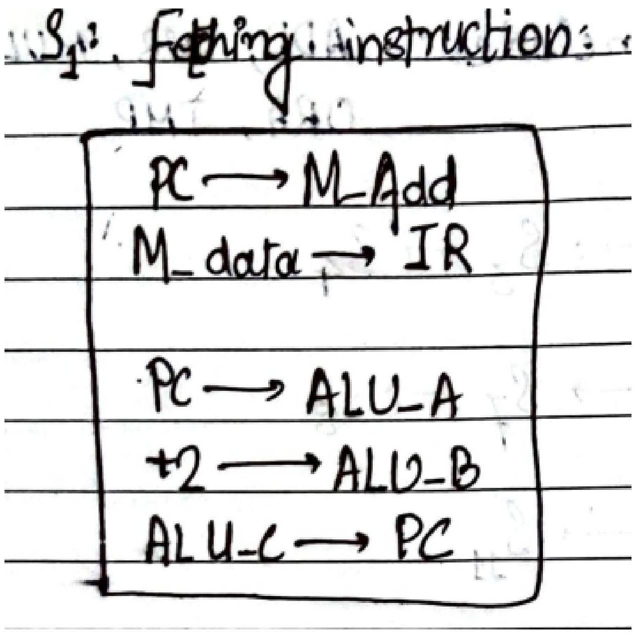
S_2 to S_{30} are not equivalent but they can be reduced to a single state when used with a decoder.

S_{14} and S_{19} can be merged into a single state $\{S_{14}\}$ with a decoder used to branch into next states

S_4 and S_{16} are merged into a single state $\{S_4\}$ using a MUX to select from previous state (S_3 or S_{15}).

States are numbered as per the sequence of instructions stored in the MEMORY. The states are then reduced as said.

S1:


S_1: Fetching instruction:

$PC \rightarrow M\_Add$

$M\_data \rightarrow IR$

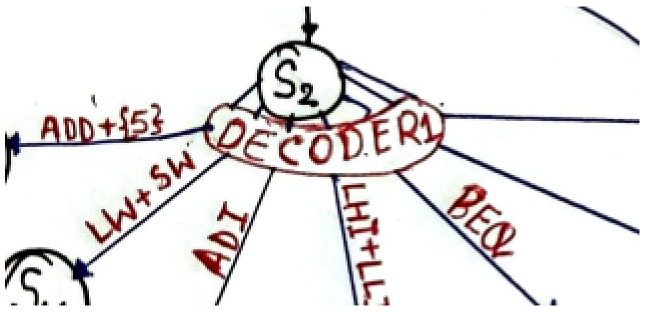
$PC \rightarrow ALU\_A$

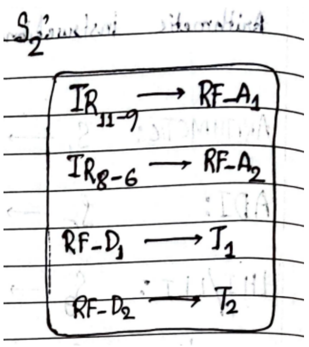
$+2 \rightarrow ALU\_B$

$ALU\_C \rightarrow PC$

State S1 and S2 are common for all the operations. S1 operation is all about fetching pieces of the 16-bit instruction. The address value stored in the PC register is fed into the memory, which then outputs the value stored in it corresponding to the address value fed.
This state also updates the address value stored in PC, making it to point to the next instruction.

S2:(& S6, S10, S13, S18, S22, S26, S30)



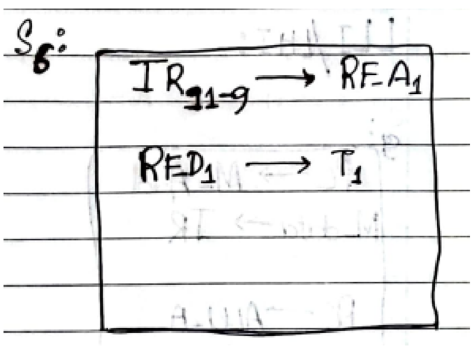

S_2:

$IR_{11-9} \rightarrow RF\text{-}A_1$

$IR_{8-6} \rightarrow RF\text{-}A_2$

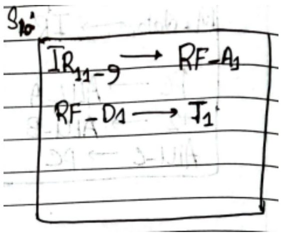
$RF\text{-}D_1 \rightarrow T_1$

$RF\text{-}D_2 \rightarrow T_2$


S_6:

$IR_{11-9} \rightarrow RF\text{-}A_1$

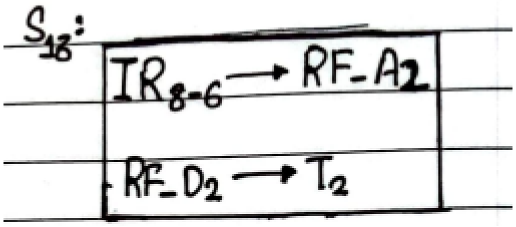
$RF\text{-}D_1 \rightarrow T_1$



S_{10}:

$IR_{11-9} \rightarrow RF\_A_1$

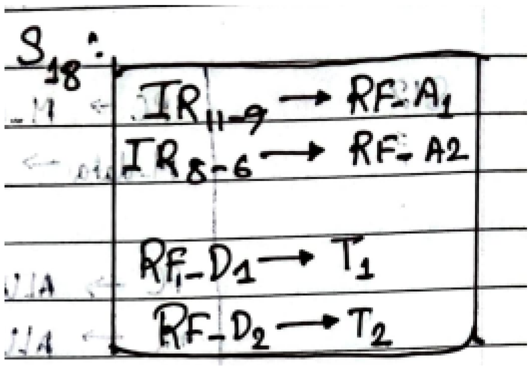
$RF\_D_1 \rightarrow T_1$



S_{18}:

$IR_{8-6} \rightarrow RF\_A_2$

$RF\_D_2 \rightarrow T_2$



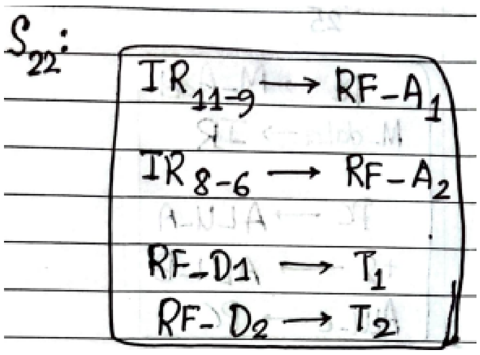
S_{18}:

$IR_{11-9} \rightarrow RF\_A_1$

$IR_{8-6} \rightarrow RF\_A2$

$RF\_D_1 \rightarrow T_1$

$RF\_D_2 \rightarrow T_2$



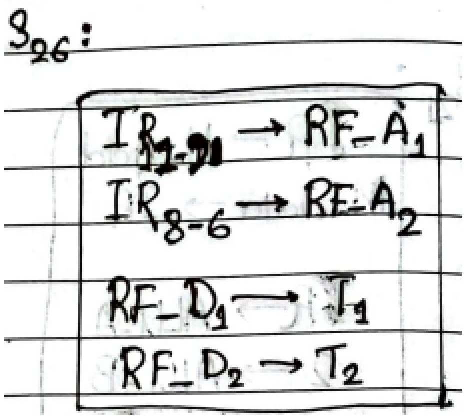
S_{22}:

$IR_{11-9} \rightarrow RF\_A_1$

$IR_{8-6} \rightarrow RF\_A_2$

$RF\_D1 \rightarrow T_1$

$RF\_D_2 \rightarrow T_2$



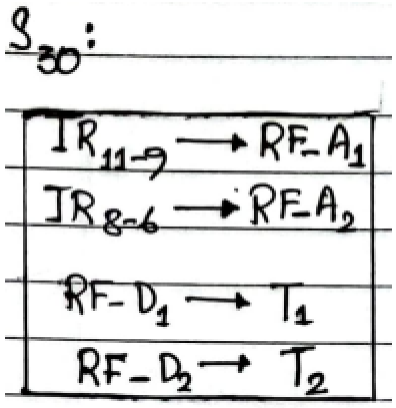
S_{26}:

$IR_{11-9} \rightarrow RF\_A_1$

$IR_{8-6} \rightarrow RF\_A_2$

$RF\_D_1 \rightarrow T_1$

$RF\_D_2 \rightarrow T_2$



S_{30}:

$IR_{11-9} \rightarrow RF\_A_1$

$IR_{8-6} \rightarrow RF\_A_2$

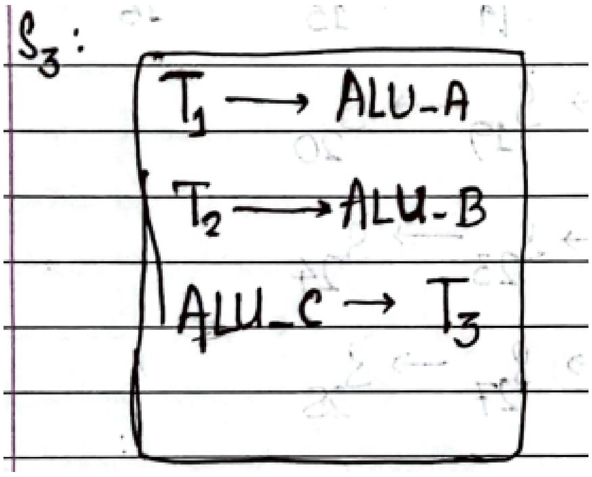
$RF\_D_1 \rightarrow T_1$

$RF\_D_2 \rightarrow T_2$

The address of the operands are fed to the address lines of the RF, which will spew out the operands for the ALU. The operands are stored in the temporary variables (16-bit registers) T1 and T2.
Operands are loaded from the memory to the RF using the LOAD operations before executing any state of the next instruction.
FLOW:
LOAD1 → LOAD2 → Execute the instruction by resuming the FSM.

From the previously given state minimization chart, S2 is not just a state of ARITHMETIC instruction bundle, but also branches off to every other instruction set. Henceforth, we have used a decoder (in theory), that uses OPCODE as its control lines to direct S2.

S3:



S_3:

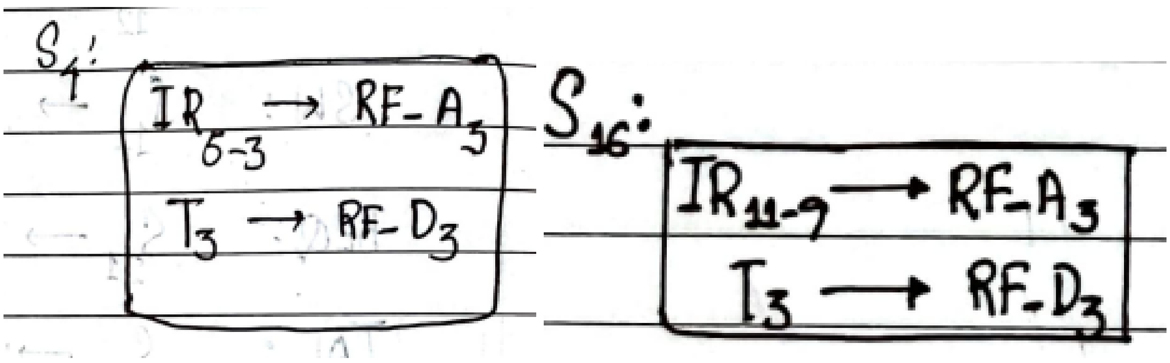
$T_1 \longrightarrow$ ALU-A

$T_2 \longrightarrow$ ALU-B

ALU_C $\rightarrow T_3$

The operands are given to the input pins ALU_A and ALU_B, and the output is stored in the third variable register T3.

S4:(& S16)



S_4:

$IR_{5-3} \rightarrow$ RF-A_3

$T_3 \rightarrow$ RF-D_3

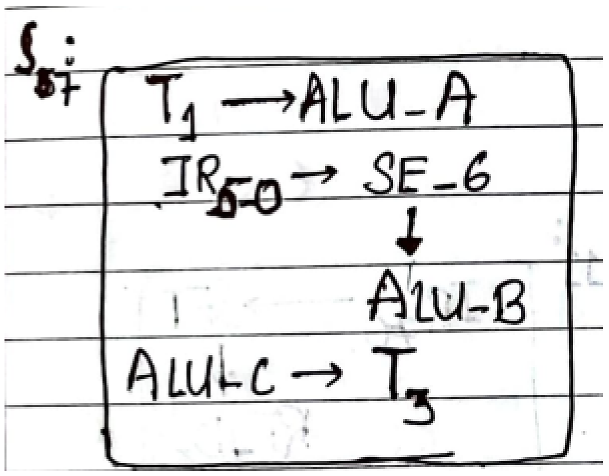
S_{16}:

$IR_{11-9} \longrightarrow$ RF-A_3

$T_3 \longrightarrow$ RF-D_3

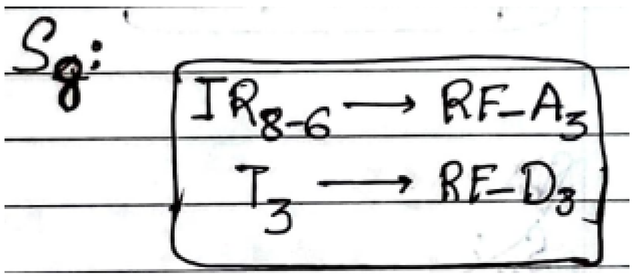
The stored result is sent to be stored in the location of REG_C (arithmetic)/ REG_A(Load word). A MUX controls the address that goes in RF_A3.

S7:



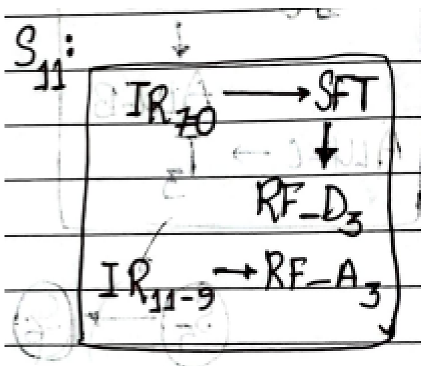
Adds the content of regA(in T1) with 6-bit Immediate, after the signed extension of the value stored in it.(Performs ADI=Adding Immediate)

S8:



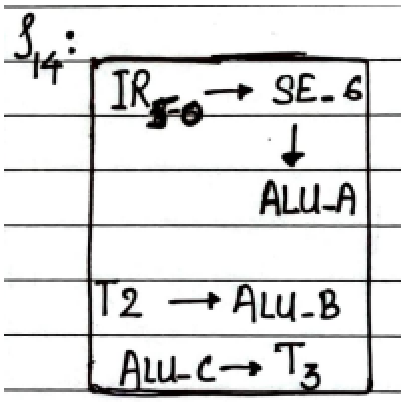
The sum is then stored in REG_B.

S11:



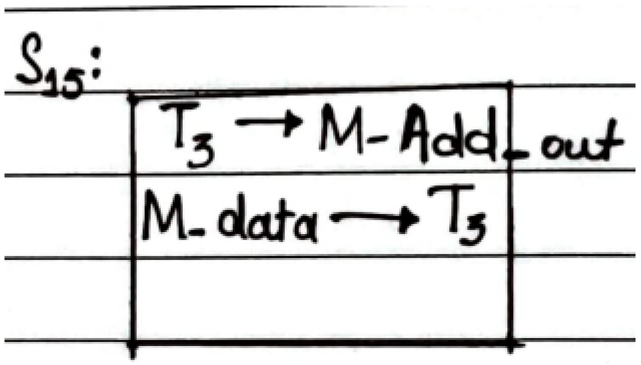
Stores the shifted contents of IMM according to instruction executed, LLI or LHI, into REG_A. Whether the content is to be placed at the least or most significant bits is decided by SFT component, that is independent of FSM.

S14(& S19):



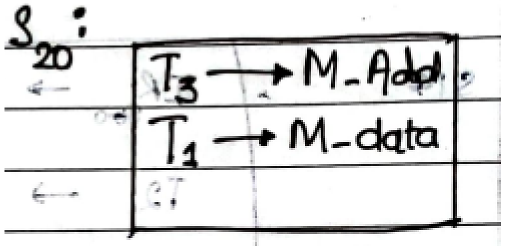
Adding the signed extension of the 6-bit Imm with the content of REG_B. The sum is the address where the value is to be stored in memory.

S15:



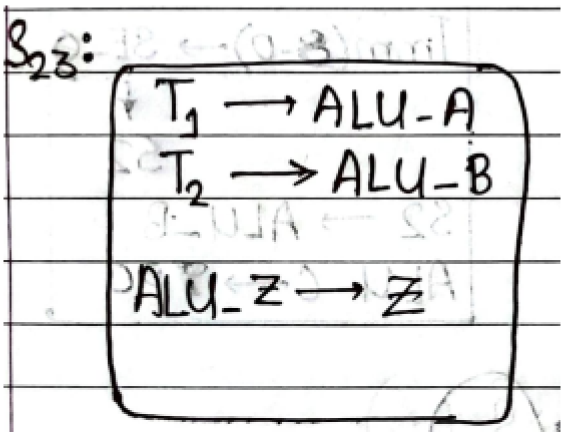
The address is injected into the Memory at Mem_Add pin from T3 (T3_out).And simultaneously, the extracted data is stored in T3(T3_in).

S20:



The address computed in S19/S14 (stored in T3) is used to store the value present in REG_A (loaded in T1), in the memory, performing SW(Store Word) instruction.

S23:



S_{23}:

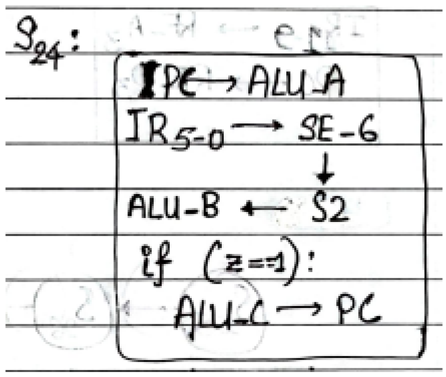
$T_1 \longrightarrow$ ALU-A

$T_2 \longrightarrow$ ALU-B

ALU_Z $\longrightarrow$ Z

The values stored in REG_A and REG_B are passed to ALU.

S24:



S_{24}:

PC $\rightarrow$ ALU-A

$IR_{5-0} \longrightarrow$ SE-6

$\downarrow$

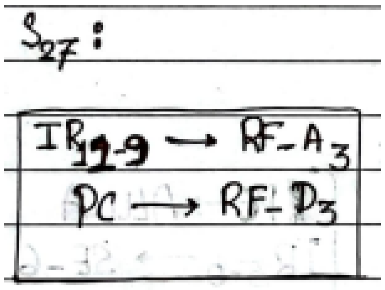
ALU-B $\leftarrow$ S2

if (z==1):

ALU-C $\rightarrow$ PC

In S24 we calculate the branching address by adding address of BEQ instruction with twice the value of the content stored in Imm (content in Imm is first sign-extended). If the Zero flag is 1, then the PC points to the branching address.

S27:



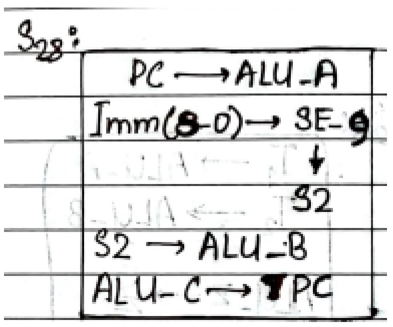
S_{27}:

$IR_{11-9} \rightarrow$ RF-A_3

PC $\longrightarrow$ RF-D_3

Stores the address in memory where the PC is pointing to, in REG_A.

S28:

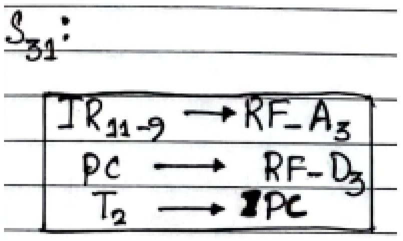


S_{28}:

PC $\longrightarrow$ ALU-A
Imm(8-0) $\longrightarrow$ SE-9
$\downarrow$
S2
S2 $\longrightarrow$ ALU-B
ALU-C $\longrightarrow$ PC

The PC is branched off to the address calculated by summing the current address stored in PC (of JAL instruction) with the content stored in the 9-bit Immediate after being sign-extended.

S31:

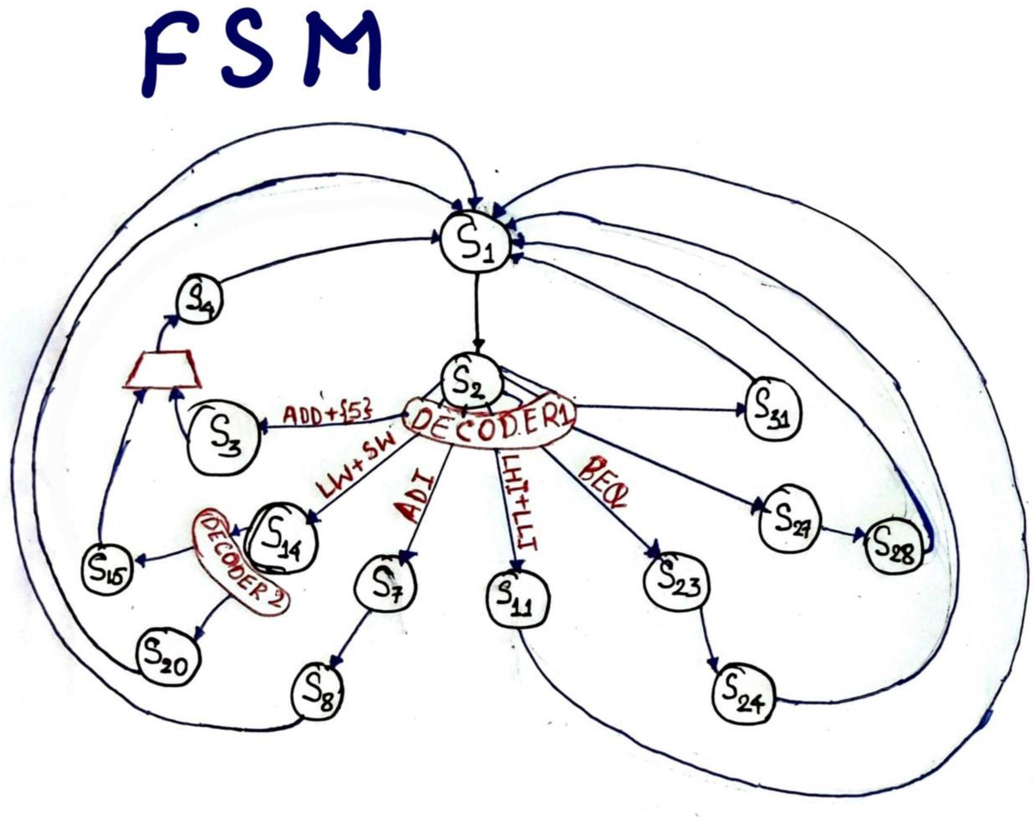


S_{31}:

$IR_{11-9} \longrightarrow RF\_A_3$
PC $\longrightarrow$ $RF\text{-}D_3$
$T_2 \longrightarrow$ PC

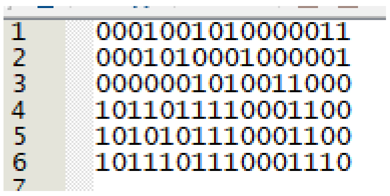
Stores PC into REG_A and branches PC to address stored in REG_B (loaded in T2).

REDUCED FSM DIAGRAM:



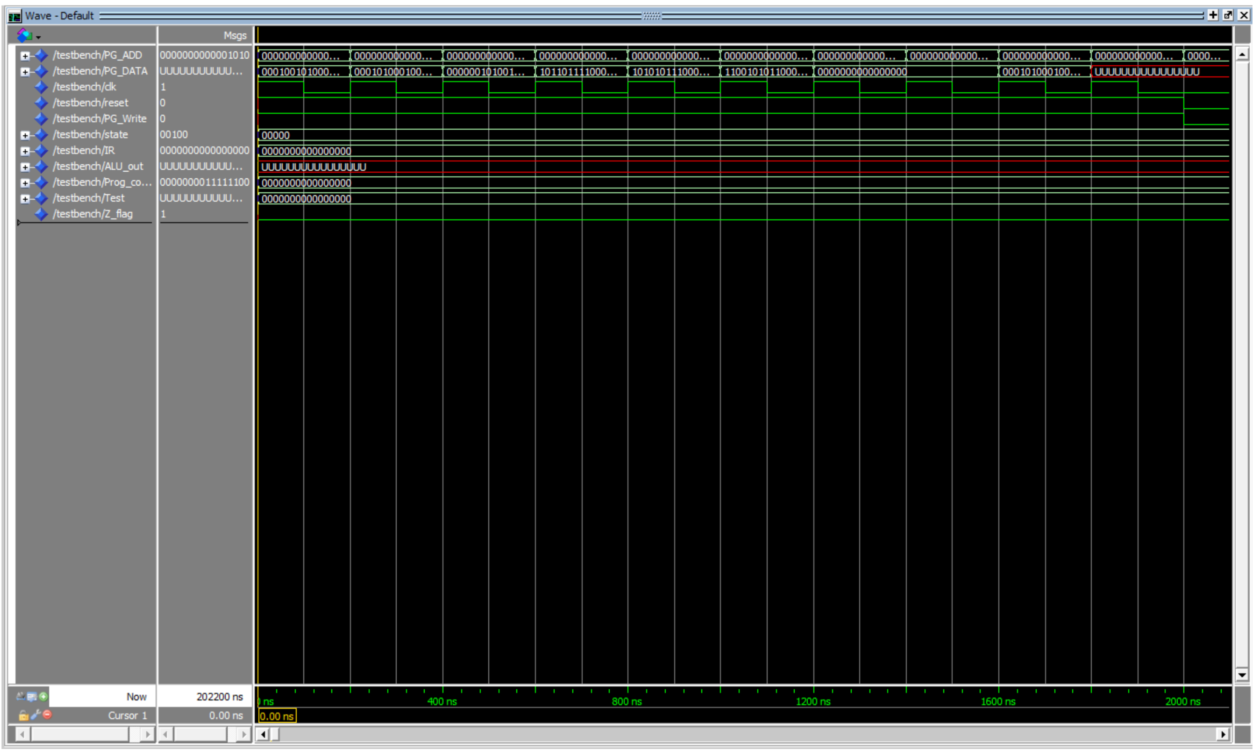
RTL Simulation View:

Instruction.txt:



```
0001001010000011
0001010001000001
0000001010011000
1011011110001100
1010101110001100
1011101110001110
```

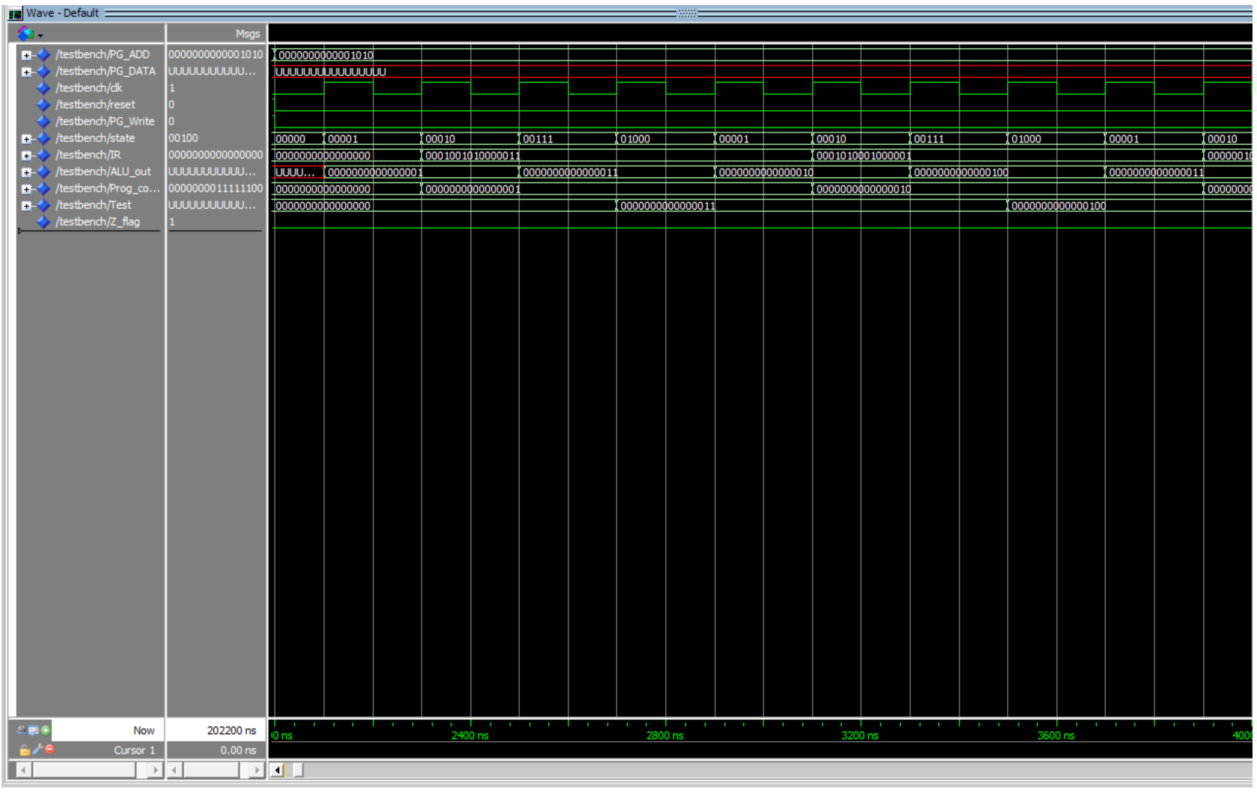
PG_Write Operation:



```
Wave - Default
                           Msgs
/testbench/PG_ADD          0000000000001010
/testbench/PG_DATA         UUUUUUUUUUUU...
/testbench/clk             1
/testbench/reset           0
/testbench/PG_Write        0
/testbench/state           00100
/testbench/IR              0000000000000000
/testbench/ALU_out         UUUUUUUUUUUU...
/testbench/Prog_co...      0000000011111100
/testbench/Test            UUUUUUUUUUUU...
/testbench/Z_flag          1

Now        202200 ns
Cursor 1   0.00 ns
```

The CPU reads and stores the instructions one-by-one in the Memory

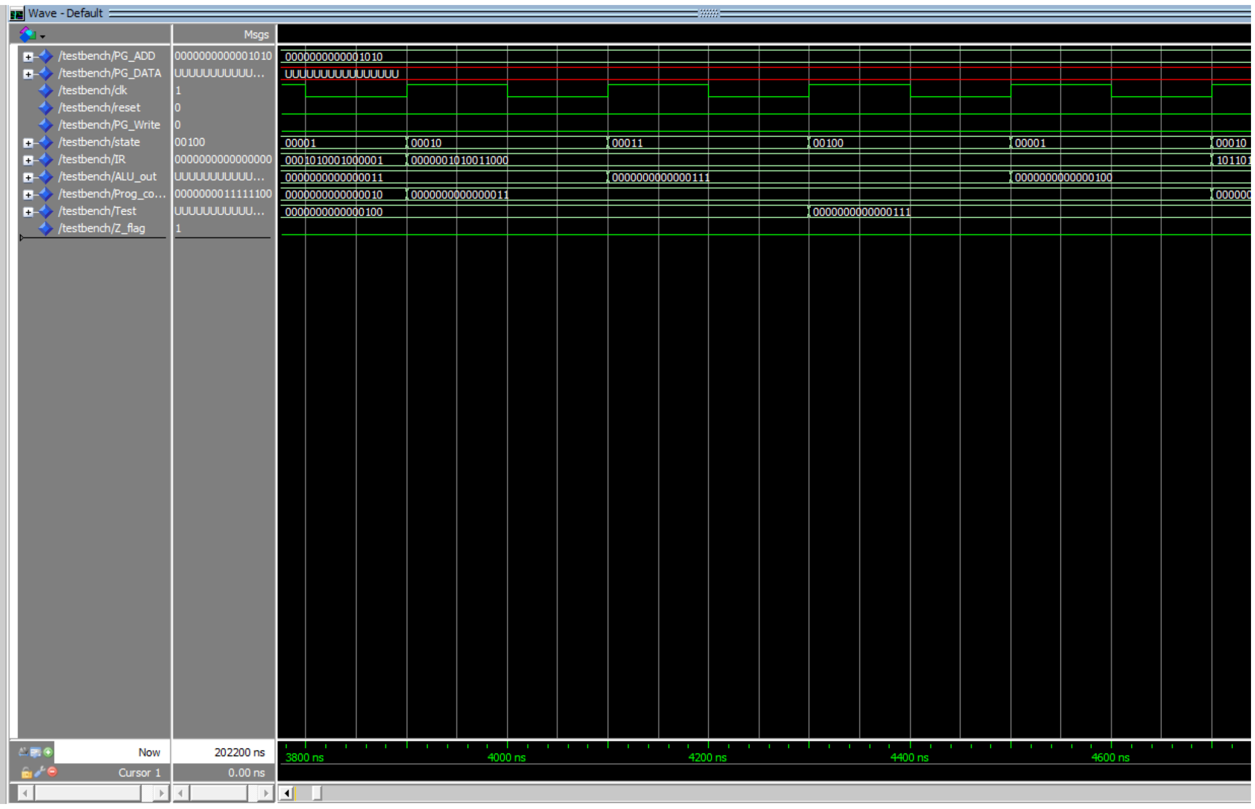
ADI Execution:



```
Wave - Default
                      Msgs
/testbench/PG_ADD     0000000000001010
/testbench/PG_DATA    UUUUUUUUUUU...
/testbench/clk        1
/testbench/reset      0
/testbench/PG_Write   0
/testbench/state      00100
/testbench/IR         0000000000000000
/testbench/ALU_out    UUUUUUUUUUU...
/testbench/Prog_co... 0000000011111100
/testbench/Test       UUUUUUUUUUU...
/testbench/Z_flag     1

Now       202200 ns
Cursor 1  0.00 ns
```

```
PG_ADD    0000000000001010
PG_DATA   UUUUUUUUUUUUUUUU
state     00000 | 00001 | 00010 | 00111 | 01000 | 00001 | 00010 | 00111 | 01000 | 00001 | 00010
IR        0000000000000000 | 0001001010000011 | 0001010001000001 | 00000010
ALU_out   UUUU... | 0000000000000001 | 0000000000000011 | 0000000000000010 | 0000000000000100 | 0000000000000011
Prog_co   0000000000000000 | 0000000000000001 | 0000000000000010 | 00000000
Test      0000000000000000 | 0000000000000011 | 0000000000000100
```

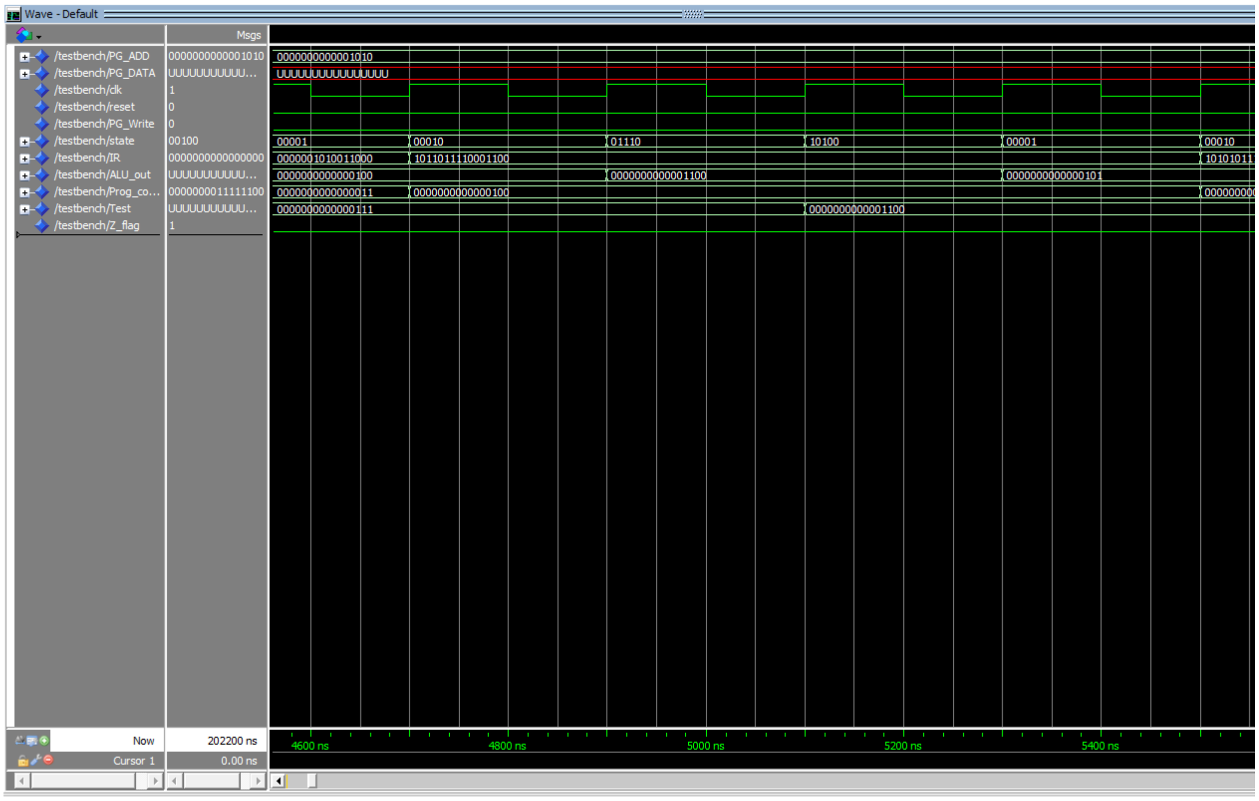
ADD Execution:



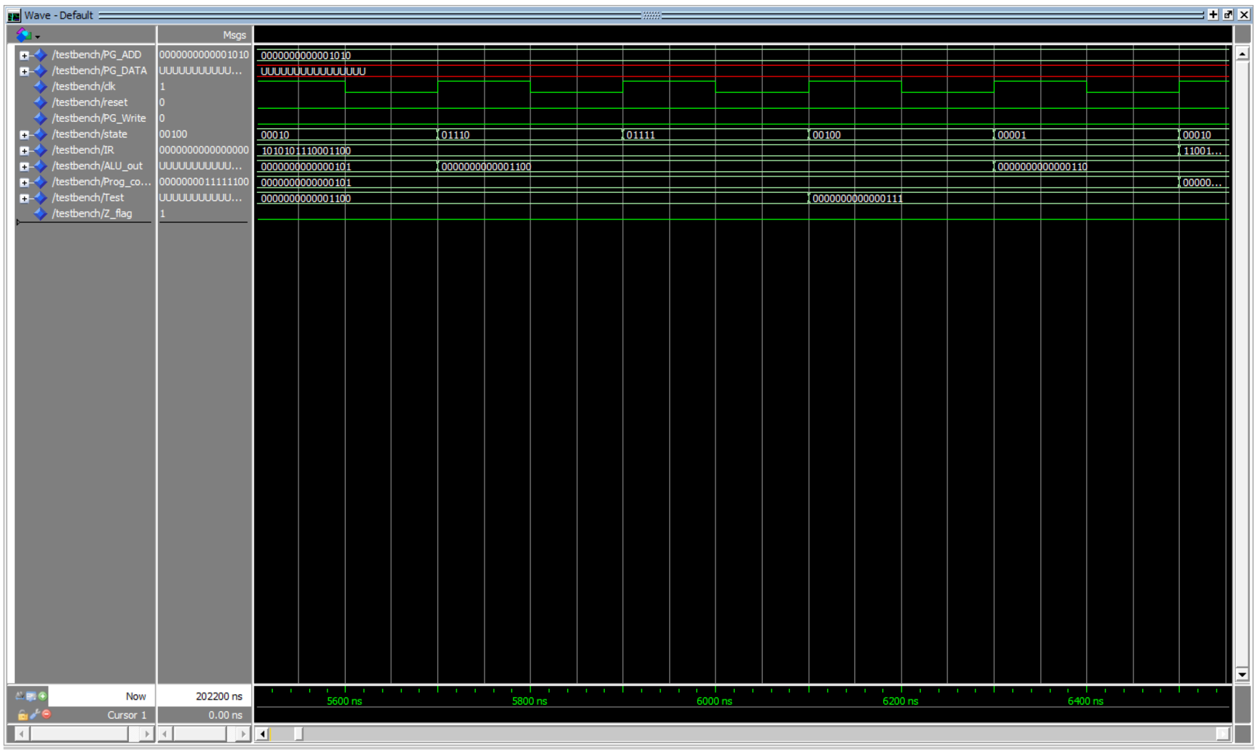
```
Wave - Default
                      Msgs
/testbench/PG_ADD     0000000000001010
/testbench/PG_DATA    UUUUUUUUUUU...
/testbench/clk        1
/testbench/reset      0
/testbench/PG_Write   0
/testbench/state      00100
/testbench/IR         0000000000000000
/testbench/ALU_out    UUUUUUUUUUU...
/testbench/Prog_co... 0000000011111100
/testbench/Test       UUUUUUUUUUU...
/testbench/Z_flag     1

Now       202200 ns
Cursor 1  0.00 ns
```

```
PG_ADD    0000000000001010
PG_DATA   UUUUUUUUUUUUUUUU
state     00001 | 00010 | 00011 | 00100 | 00001 | 00010
IR        0001010001000001 | 0000001010011000 | 101101
ALU_out   0000000000000011 | 0000000000000111 | 0000000000000100
Prog_co   0000000000000010 | 0000000000000011 | 000000
Test      0000000000000100 | 0000000000000111
```

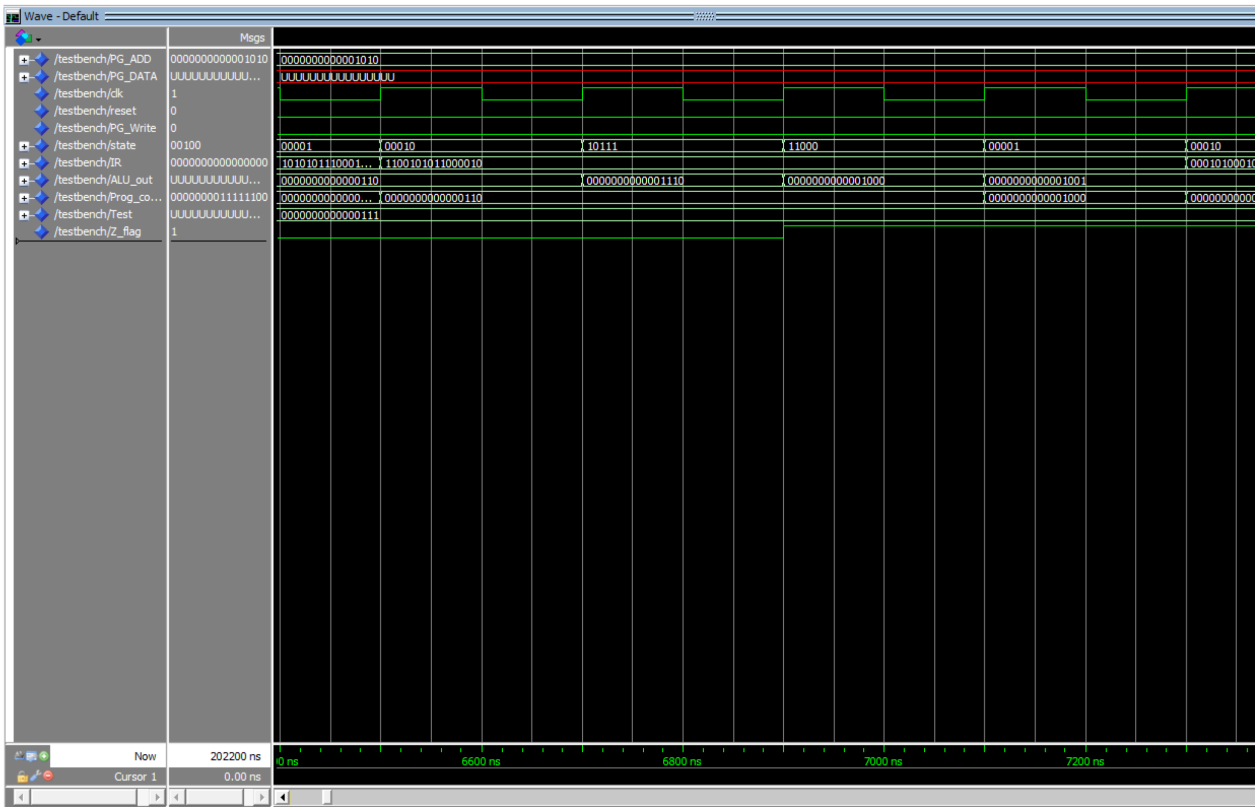
SW Execution:



LW Execution:

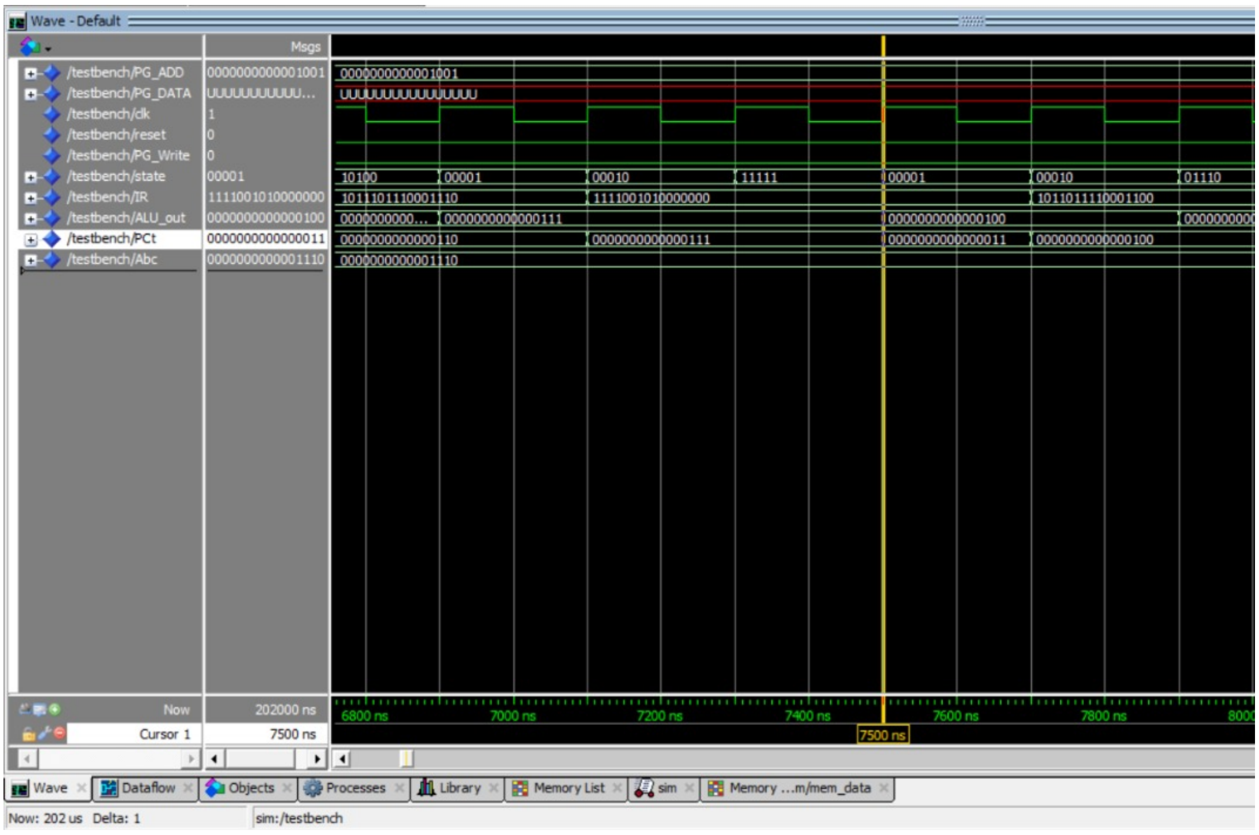


BEQ Execution:



```
Wave - Default
                          Msgs
/testbench/PG_ADD         0000000000001010
/testbench/PG_DATA        UUUUUUUUUUU...
/testbench/clk            1
/testbench/reset          0
/testbench/PG_Write       0
/testbench/state          00100
/testbench/IR             0000000000000000
/testbench/ALU_out        UUUUUUUUUUU...
/testbench/Prog_co...     0000000011111100
/testbench/Test           UUUUUUUUUUU...
/testbench/Z_flag         1
Now                       202200 ns
Cursor 1                  0.00 ns
```

JLR Execution:

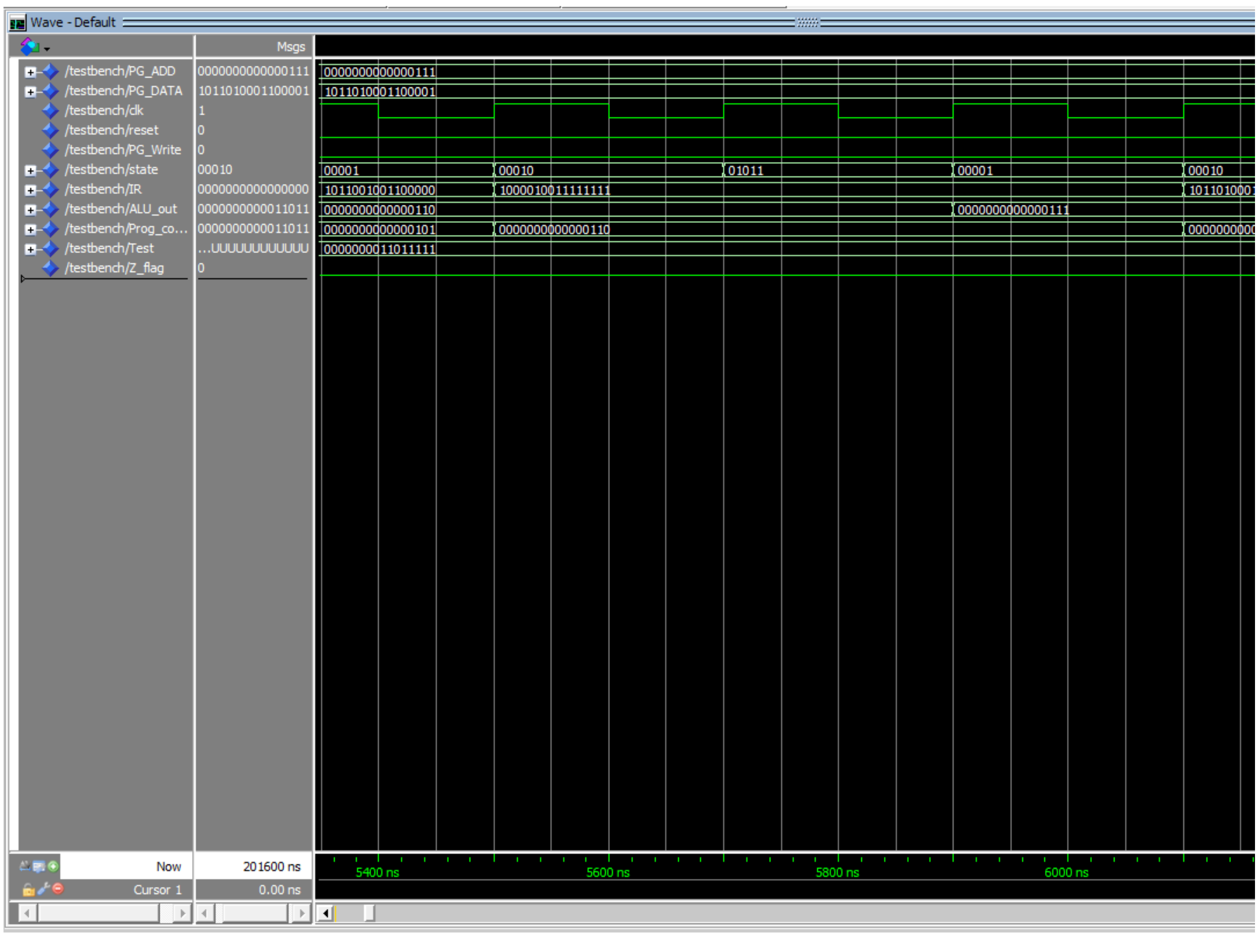


```
Wave - Default
                          Msgs
/testbench/PG_ADD         0000000000001001
/testbench/PG_DATA        UUUUUUUUUUU...
/testbench/clk            1
/testbench/reset          0
/testbench/PG_Write       0
/testbench/state          00001
/testbench/IR             1111001010000000
/testbench/ALU_out        0000000000000100
/testbench/PCt            0000000000000011
/testbench/Abc            0000000000001110
Now                       202000 ns
Cursor 1                  7500 ns
```

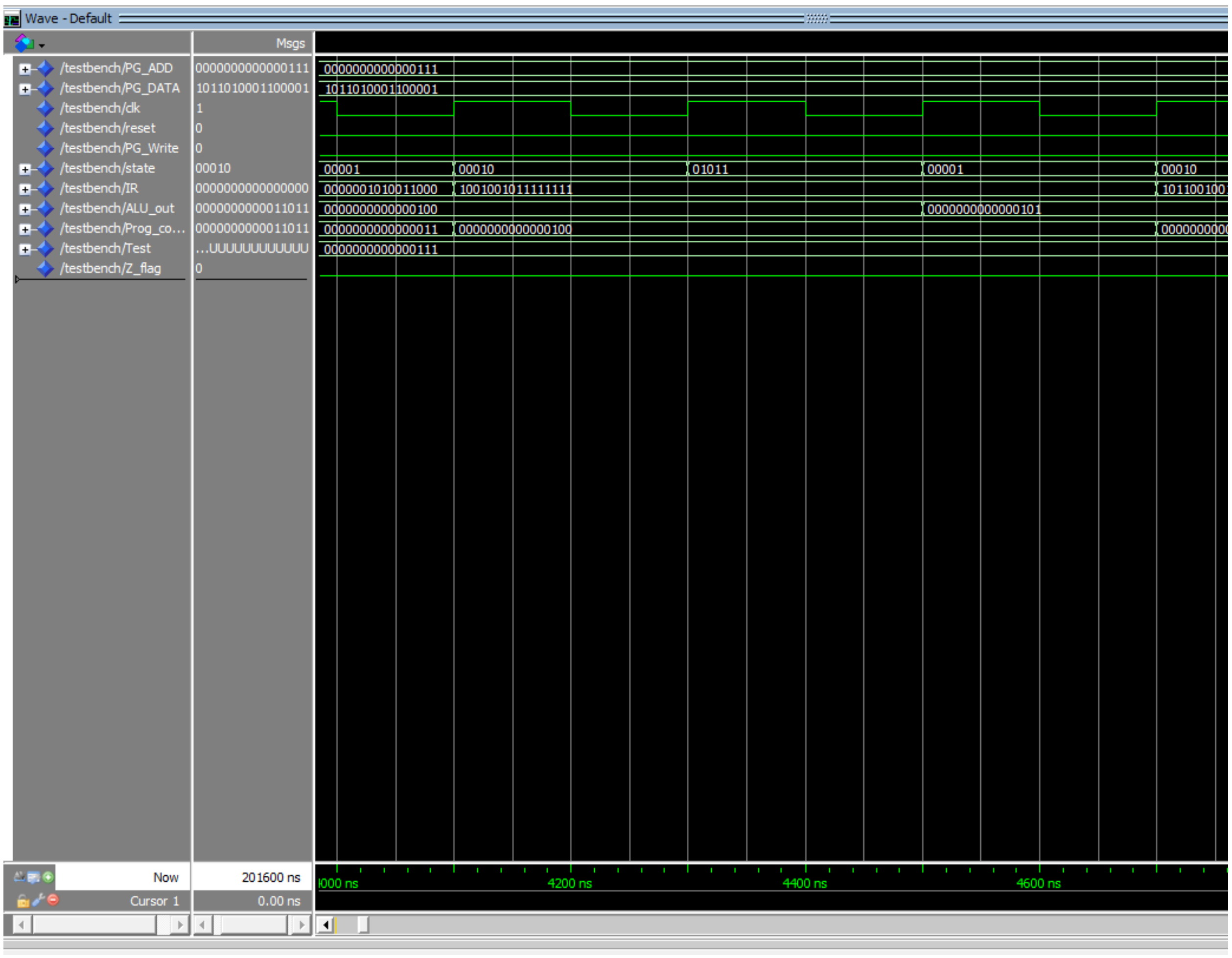
Instruction for LLI and LHI

0001001010000011
0001010001000001
0000001010011000
1001001011111111
1011001001100000
1000010011111111
1011010001100001

LHI Execution:


```
Wave - Default
                           Msgs
/testbench/PG_ADD          0000000000000111   0000000000000111
/testbench/PG_DATA         1011010001100001   1011010001100001
/testbench/clk             1
/testbench/reset           0
/testbench/PG_Write        0
/testbench/state           00010              00001              00010              01011              00001              00010
/testbench/IR              0000000000000000   1011001001100000   1000010011111111                                         1011010001
/testbench/ALU_out         0000000000011011   0000000000000110                                         0000000000000111
/testbench/Prog_co...      0000000000011011   0000000000000101   0000000000000110                                         0000000000
/testbench/Test            ...UUUUUUUUUUUU    0000000011011111
/testbench/Z_flag          0

Now       201600 ns        5400 ns     5600 ns     5800 ns     6000 ns
Cursor 1  0.00 ns
```

LLI Execution:



```
Wave - Default
                          Msgs
/testbench/PG_ADD         0000000000000111   0000000000000111
/testbench/PG_DATA        1011010001100001   1011010001100001
/testbench/clk            1
/testbench/reset          0
/testbench/PG_Write       0
/testbench/state          00010              00001   00010   01011   00001   00010
/testbench/IR             0000000000000000   0000001010011000   1001001011111111   101100100
/testbench/ALU_out        0000000000011011   0000000000000100   0000000000000101
/testbench/Prog_co...     0000000000011011   0000000000000011   0000000000000100   000000000
/testbench/Test           ...UUUUUUUUUUUU    0000000000000111
/testbench/Z_flag         0

Now                       201600 ns
Cursor 1                  0.00 ns
```

Work Distribution:

In this section, we outline the roles and responsibilities undertaken by each team member in the execution of the project. The allocation of tasks was done to expedite the progress of the project. The following provides a transparent overview of the work distribution:

Task allocation:

Along the names of the team members are the components/areas on which they have partially/ fully worked on.

- Saarthak Krishan: ALU, FSM(Finite State Machine), Adder-Subtractor, Multiplier, Final CPU Integration
- Aniket Patel: FSM, Register File (RF), SE_6, SE_9,Final CPU Integration
- Sanat Kumar Agrawal: FSM, SE_6, SE_9 , SFT, Memory,Final CPU Integration
- Utkarsh Prakash: 16-bit register, RF, overall documentation and report making,Final CPU Integration

Apart from making components individually, each team member has contributed in debugging and state minimization of the FSM and overall code, as well as pen-paper design of datapath and components.