EE224 Project: IITB-CPU
Documentation and Report

GROUP 13
Advay Bapat
Jigar Mehta
Saniya Khinvasara
Suketu Patni

December 4, 2024

Code Overview

Not including the testbench, our code consists of the following files. We have given a brief explanation for all files and a more comprehensive explanation below the table for some of the bigger files.

File Name	Explanation
`2to1mux`	Implements a one-bit 2-to-1 multiplexer, selecting one of two inputs based on a control signal.
`4to1mux`	Implements a one-bit 4-to-1 multiplexer, selecting one of four inputs based on the 2 control signals.
`mux8`	Implements a one-bit 8-to-1 multiplexer, selecting one of eight inputs based on the 3 control signals.
`16bit8to1mux`	Implements a 16-bit 8-to-1 mux, selecting one of the eight inputs based on the 3 control signals.
`16bit1to8demux`	connects a 16-bit input to one of eight possible outputs using a demultiplexer
`alu`	perform arithmetic and logic operations such as addition, subtraction, and bitwise operations (combines many low-level operations into one entity)
`bitwise_AND`	Executes bitwise AND operation on binary inputs (16-bit) and outputs corresponding 16-bit result
`bitwise_IMP`	Executes bitwise implication on binary inputs(16-bit) and outputs corresponding 16-bit result
`bitwise_OR`	Executes bitwise OR operation on binary inputs (16-bit) and outputs corresponding 16-bit result.
`demux2`	A 1-to-2 demultiplexer that directs an input to one of two outputs.
`demux4`	A 1-to-4 demultiplexer that directs an input to one of four outputs.
`demux8`	A 1-to-8 demultiplexer that directs an input to one of eight outputs.
`Flipflop`	A basic D flip-flop used for single-bit storage.
`fsm`	The main controlling file which is the FSM of the CPU
`full_adder`	Adds three binary bits and produces a sum and carry output.
`sixteen_bit_full_adder`	A ripple carry adder that inputs two 16-bit numbers and outputs their addition or subtraction depending on the 'm' input.
`multiplier`	Multiplies two 4 bit inputs(taken as 16 bit with zero padding) and produces the product (16 bit)
`memory`	Module for memory storage and retrieval, supporting read/write operations.
`register16bit`	A 16-bit data storage register.
`RegisterFile`	Manages multiple registers for temporary data storage in a CPU , has features similar to the ones taught in class
`shift_one`	Performs a single-bit shift operation to the left

`sign_ext_nine`	Extends a 9-bit binary input to a 16 bit output with sign extension, keeping in mind that the input could be a signed number.
`sign_ext_six`	Extends a 6-bit binary input to a 16 bit output with sign extension, keeping in mind that the input could be a signed number.
`XOR_NAND`	A simple XOR gate which was implemented using NAND gates as part of EE214 exercise
`zero_flag`	Checks if the 16 bit input is all zeros and gives the output of 1-bit as 1 if that is thr case, the output is zero otherwise
`zero_padding`	Adds leading zeros to make the input into a 16-bit output

Multiplier

Overview

The **4-bit Multiplier** implements a combinational circuit to compute the product of two 4-bit input vectors, `a` and `b`. The result is a 4-bit product represented in `c`. The architecture is built by combining bitwise logical operations, shift operations, zero padding, and multi-bit addition to achieve the multiplication.

Entity Description

Port	Description
`a` (`std_logic_vector(15 downto 0)`)	The first 16-bit operand for multiplication, with only the least significant 4 bits being used.
`b` (`std_logic_vector(15 downto 0)`)	The second 16-bit operand for multiplication, with only the least significant 4 bits being used.
`c` (`std_logic_vector(15 downto 0)`)	The computed 16-bit product.

Logical Flow and Components

1. Bitwise AND Operations

- For each bit pair (`a[i]`, `b[j]`), a bitwise AND operation is performed to generate partial products.
- Partial products are grouped into four signals: `s0`, `s1`, `s2`, and `s3`, representing the contributions of the four bits in `b` to the product.

2. Zero Padding

- Partial products (`s0`, `s1`, `s2`, `s3`) are extended to 16-bit vectors using the `zero_padding` component to align them correctly for further operations.

3. Shifting

- To account for the positional significance of bits in binary multiplication:
  - `s1` is shifted left by 1 position (multiplication by 2) to form `s1_final`.
  - `s2` is shifted left by 2 positions(multiplication by 4), resulting in `s2_final`.
  - `s3` is shifted left by 3 positions (multiplication by 8), forming `s3_final`.

4. Addition of Partial Product

The extended and shifted partial products are summed using three stages of 16-bit addition (since each stage can only compute on 2 operands).

ALU

Overview

The Arithmetic Logic Unit (ALU) is a combinational circuit implemented in VHDL that integrates multiple arithmetic and logic operations listed above into a single entity. It allows for the selection of operations via a control signal (`sel`, 3 bit) and outputs the result along with additional flags for zero (`zf`) and carry (`cf`). The two operands and the main output are assumed to be 16-bit.

Entity Definition

The `alu` entity declares the following ports:

Inputs

- `A, B`: Two 16-bit input operands for arithmetic and logic operations.
- `sel`: A 3-bit control signal to select the operation.

Outputs

- `C`: A 16-bit result of the selected operation.
- `zf`: A zero flag indicating whether the result is zero.
- `cf`: A carry flag indicating whether a carry occurred in arithmetic operations.

Supported Operations

The ALU supports the following operations based on the `sel` signal:

Operation	`sel`	**Description**
Addition	`000`	Adds inputs `A` and `B`
Subtraction	`010`	Subtracts `B` from `A`.

Multiplication	`011`	Multiplies `A` and `B`.
Bitwise OR	`101`	Performs bitwise OR on `A` and `B`.
Bitwise AND	`100`	Performs bitwise AND on `A` and `B`.
Bitwise Implication	`110`	Computes `A → B` (if `A`, then `B`).

Execution Flow

1. Inputs `A` and `B` are fed into arithmetic and logic components.
2. All the operations are conducted and the results of all are stored in signals.
3. The `mux8bit16` component selects the output result (`Y`) out of the signals based on `sel`.
4. The `zero_flag` component checks if the result is zero and sets `zf`.
5. The carry logic determines and outputs the carry flag (`cf`).

Register and Register File

16-bit Register Description

Overview: The 16-bit register is constructed using D flip-flops and additional components for asynchronous reset and enable control.

Key Features

- **16-bit Data Storage:** Stores 16-bit wide data, with input and output represented as `std_logic_vector(15 downto 0)`.
- **Asynchronous Reset:** Clears all stored data when the `Reset` signal is asserted, setting the output to `0000_0000_0000_0000`.
- **Enable Control:** Captures new data only when the `En` signal is active; otherwise, retains the previous state.
- **Flip-Flop Construction:** Utilizes D flip-flops (`dff_reset`) to store individual bits
- **Input Selection:** A 2-to-1 multiplexer (`mux2bit16`) enables selection between new input data and the retained output.

Working

1. **Data Flow:** The input data (`data_in`) passes through a multiplexer controlled by the `En` signal. If `En` is high, new data is forwarded to the flip-flops; otherwise, the current output is fed back.
2. **Flip-Flop:** Each flip-flop stores one bit of data. The `Clk` signal triggers data capture, ensuring synchronous operation.
3. **Reset:** Asserting the `Reset` signal clears all flip-flops, resetting the output to zero.
4. **Output:** The stored data is continuously reflected on `data_out`.

Entity Description

Port	Description
`Clk` (`std_logic`)	Clock signal that triggers data capture on the rising edge.
`Reset` (`std_logic`)	Asynchronous reset signal to clear the stored data.
`En` (`std_logic`)	Enable signal to control whether new data is captured.
`data_in` (`std_logic_vector(15 downto 0)`)	16-bit input data to be stored in the register.
`data_out` (`std_logic_vector(15 downto 0)`)	16-bit output reflecting the current stored data.

Register File Description

Overview: This register file consists of 8 16-bit registers with functionality for two simultaneous reads and one write. Addressing and data routing are handled using multiplexers (MUX) and demultiplexers (DEMUX). The design supports enable control and asynchronous reset.

Key Features

- **8 16-bit Registers:** Stores 8 independent 16-bit data values, accessible for both read and write operations.
- **Two Simultaneous Reads:** Two separate multiplexers enable simultaneous reads from different registers, routed to outputs `D1` and `D2`.
- **Write Operation:** A demultiplexer ensures that input data (`D3`) is routed to the target register based on the write address (`A3`).
- **Enable Control:** Write operations are gated by an enable signal (`En`), allowing selective writes.
- **Asynchronous Reset:** Clears the content of all registers when the `Reset` signal is asserted.

Working

1. **Write Operation:** The global enable signal (`En`) is demultiplexed to activate a specific register based on the write address (`A3`). The input data (`D3`) is routed to the enabled register using a separate demultiplexer.
2. **Read Operations:** Two multiplexers select data from the registers based on the read addresses (`A1`, `A2`). The selected data is routed to the respective outputs (`D1`, `D2`).
3. **Register Storage:** Each register retains its data until updated by a write operation or cleared by the asynchronous reset (`Reset`).

Entity Description

Port	Description
`A1, A2` (`std_logic_vector(2 downto 0)`)	Read addresses for the two read operations.
`A3` (`std_logic_vector(2 downto 0)`)	Write address for the register to be written to.
`Clk` (`std_logic`)	Clock signal to synchronize data capture.
`En` (`std_logic`)	Enable signal to control register writes.
`Reset` (`std_logic`)	Asynchronous reset signal to clear all registers.
`D1, D2` (`std_logic_vector(15 downto 0)`)	Outputs for the two read operations.
`D3` (`std_logic_vector(15 downto 0)`)	Input data for the write operation.

Memory

Overview: This module implements a 2 KB memory system for a CPU, organized as an array of 256 bytes. It supports byte-addressable operations and facilitates 16-bit data reads and writes. The memory is initialized with specific values for testing purposes and can be updated during runtime.

Key Features

- **Memory Size:** The memory consists of 256 bytes, organized as an array (`arr`) of 8-bit values. This should ideally be $65535(2^{16} - 1)$ bytes but due to limited time for compilation we have limited the size to 256 bytes.
- **Addressing:** A 16-bit address (`Mem_Add`) is used to access specific memory locations, with each address corresponding to a byte.
- **Data Access:**
  - **Read:** Two consecutive bytes are read and concatenated to form a 16-bit word (`Mem_Out`).
  - **Write:** The 16-bit input (`Mem_In`) is split into high and low bytes and written to consecutive memory locations.
- **Initialization:** The memory array is loaded with predefined values for specific locations based on the kind of program or operation being conducted, while the rest are initialized to zero.

Working Mechanism

1. **Address Calculation:** The 16-bit address (`Mem_Add`) is converted to an integer (`add_int`), which is used as an index for the memory array.
2. **Read Operation:**
   - The high byte is read from the memory location indexed by `add_int`.
   - The low byte is read from the subsequent location (`add_int + 1`).
   - Both bytes are concatenated to produce the 16-bit output (`Mem_Out`).

3. **Write Operation:**
    - If `Mem_Write` is asserted, the high byte of `Mem_In` is written to the memory location indexed by `add_int`.
    - The low byte is written to the subsequent location (`add_int + 1`).

Entity Description

Port	Description
`Mem_Add` (`std_logic_vector(15 downto 0)`)	16-bit address for accessing memory.
`Mem_In` (`std_logic_vector(15 downto 0)`)	16-bit input data for write operations.
`Mem_Out` (`std_logic_vector(15 downto 0)`)	16-bit output data for read operations.
`Mem_Write` (`std_logic`)	Write enable signal. When asserted, data from `Mem_In` is written to memory.

FSM

The FSM code by and large represents the exact flowchart made in the pen-paper documentation. The code is intuitive in the sense it uses the same signal names as used in the penpaper FSM.
Other than this we have also concatenated all the enable signal into a single signal which makes the code a bit smaller. At the end of a state, branching if present has been implemented by if-else statements conditioned on the first 4 bits of the Instruction Register or the OP-code. An error state along with multiple test-signals have also been added for debugging.
The only modification present in the code from the pen-paper FSM is that a single state has on multiple occasions been broken down into smaller bits with the help of a counter. This avoids the conflict of writing and reading the same register on the same clock-edge. We understand that this could be have been implemented by making numerous smaller states but this would have hampered the readability of the code while making no functional improvements.
Finally, we have ourselves conducted tests of the FSM with a testbench for all 15 instructions. We have managed to get the FSM to work satisfactorily after some debugging. Following are the waveforms for all the tests conducted:

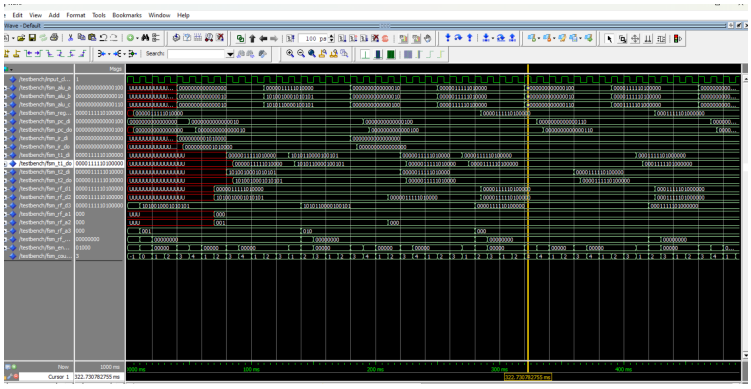

Figure 1: Add

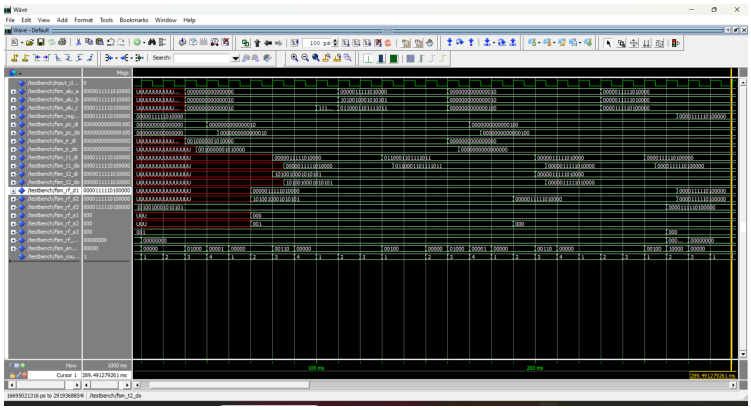

Figure 2: Sub

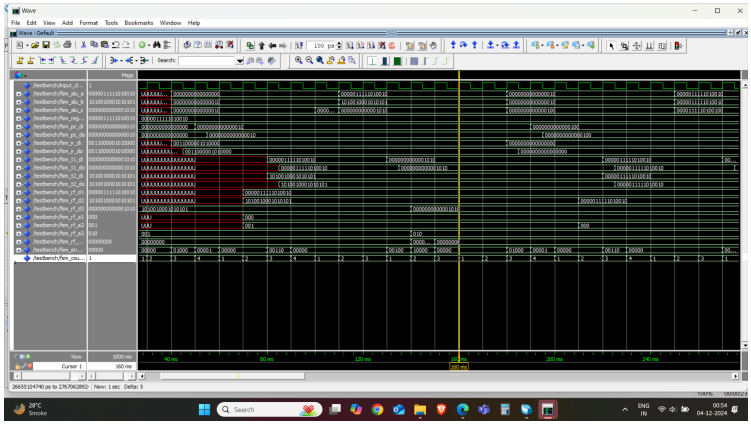

Figure 3: Mul

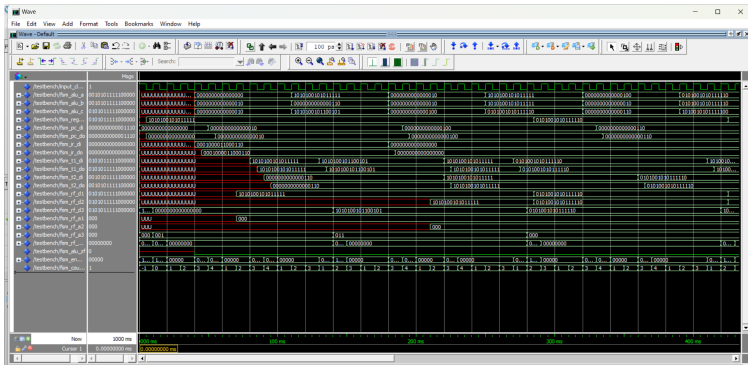

Figure 4: ADI

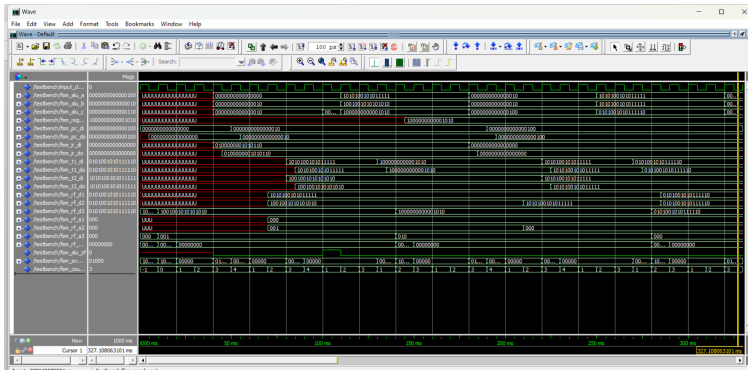

Figure 5: AND

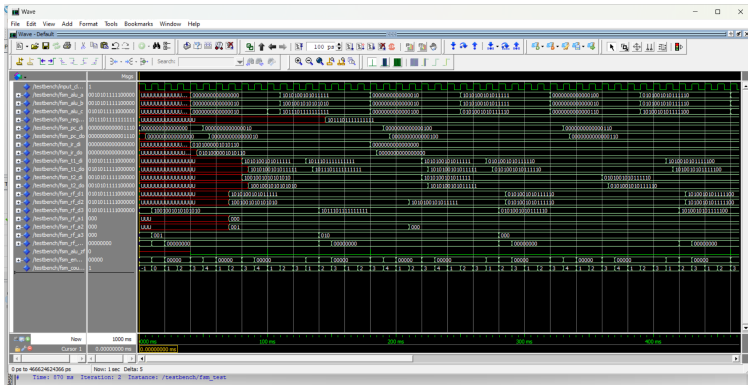

Figure 6: OR

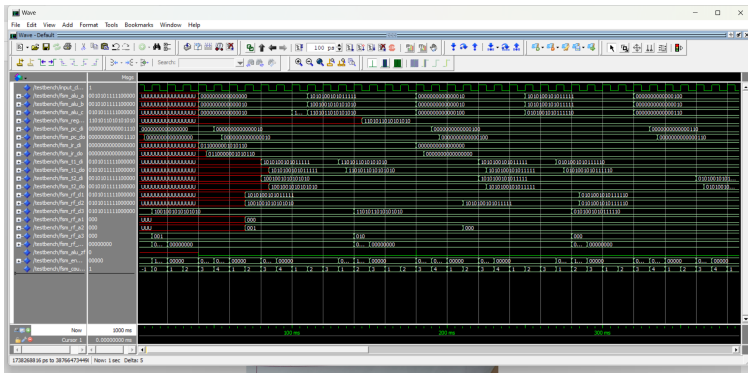

Figure 7: IMP

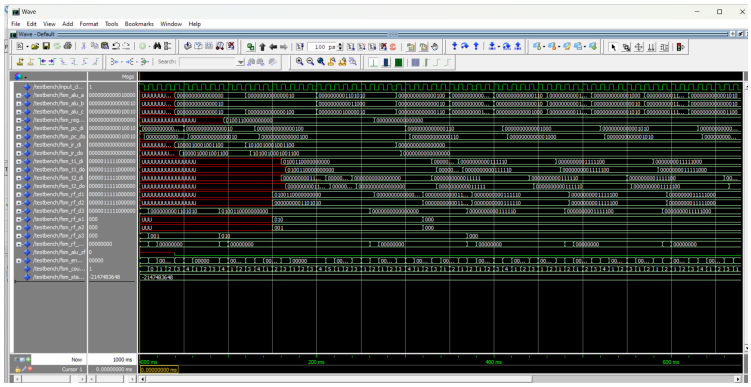

Figure 8: LHI

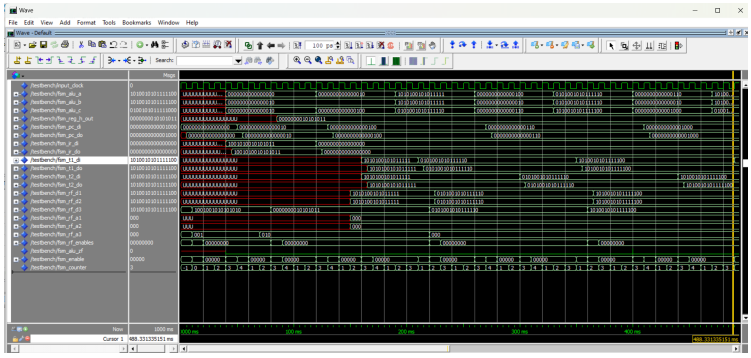

Figure 9: LLI

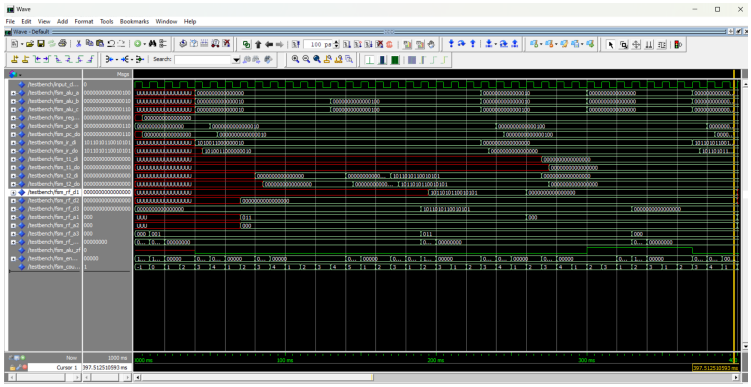

Figure 10: Load

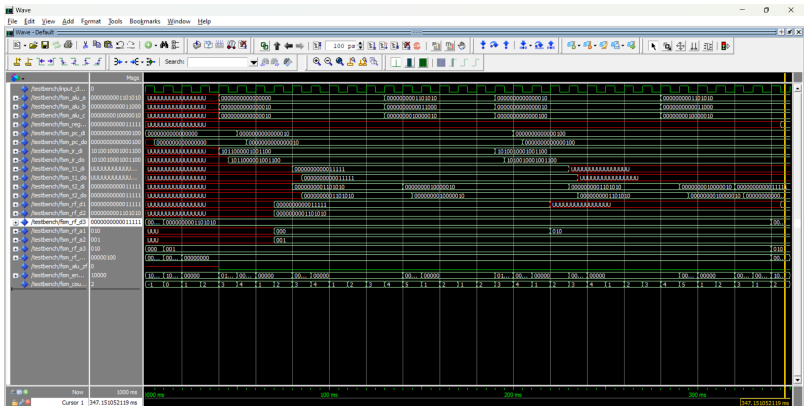

Figure 11: Store and Load combined

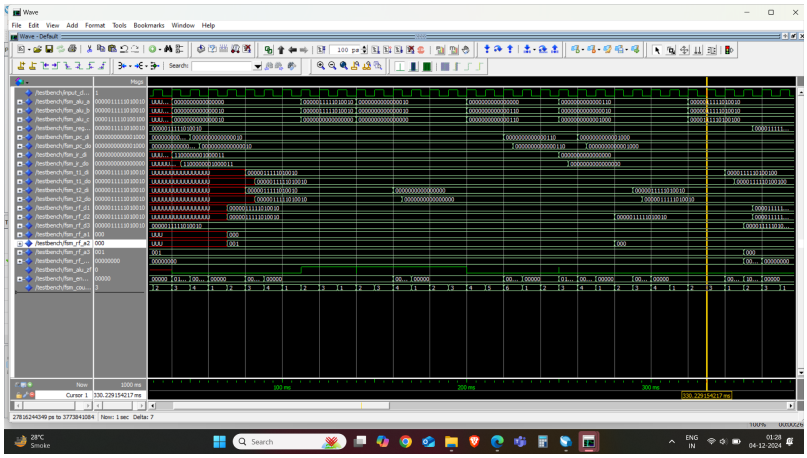

Figure 12: BEQ with 2 equal inputs

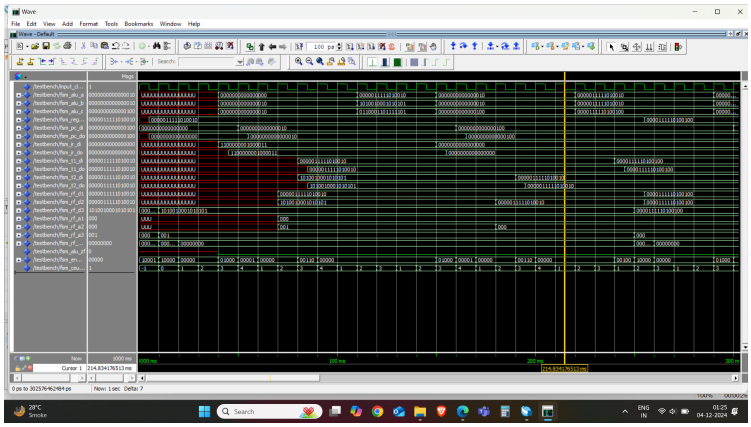

Figure 13: BEQ with 2 unequal inputs

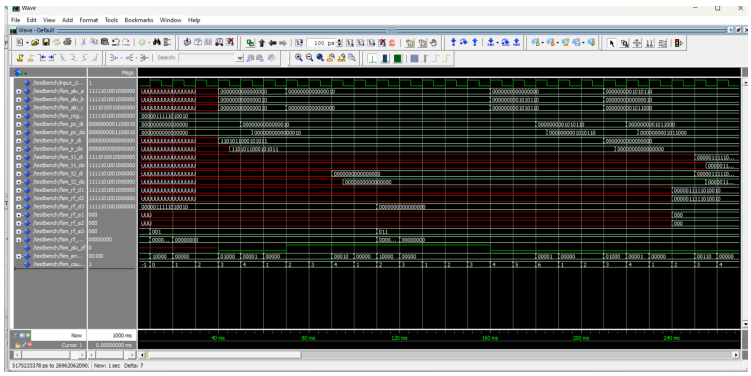

Figure 14: JAL

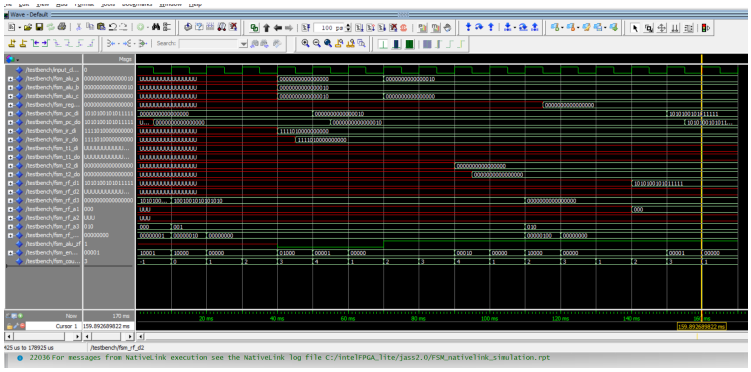

Figure 15: JLR

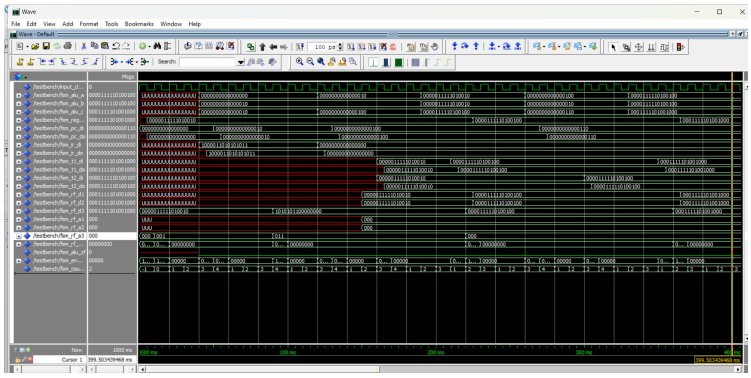

Figure 16: J