**RISC V DESIGN USING VERILOG HDL**

AN INDUSTRIAL INTERNSHIP TRAINING REPORT

*Submitted by*

**Karthick Srivatsa R**

**20BEC1221**

**ECE1902 – INDUSTRIAL INTERNSHIP**

*in partial fulfillment for the award of the degree of*

**BACHELOR OF TECHNOLOGY**

in

**ELECTRONICS AND COMMUNICATION ENGINEERING**

****

November 2023

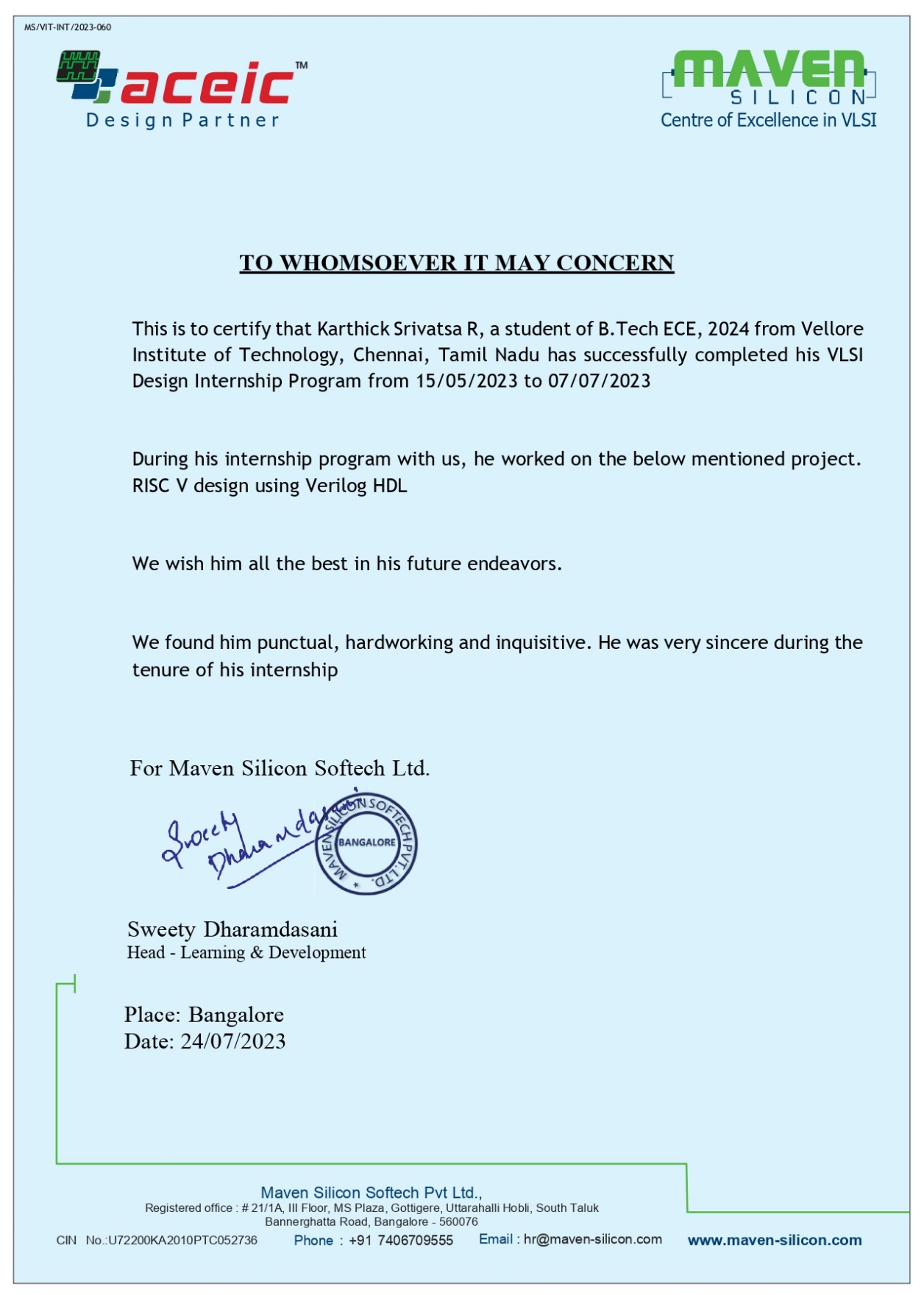
**School of Electronics Engineering**

**DECLARATION BY THE CANDIDATE**

I hereby declare that the Industrial Internship Report entitled “**RISC V DESIGN USING VERILOG HDL”** submitted by me to VIT University, Chennai in partial fulfillment of the requirement for the award of the degree of **Bachelor of Technology** in **Electronics and Communication Engineering** is a record of bonafide industrial training undertaken by me under the supervision of **Sweety Dharamdasani, Head – Learning & Development Maven Silicon Softech Ltd., Bangalore.** I further declare that the work reported in this report has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

Chennai: Signature of the Candidate

Date: 01-11-2023

****

****

**School of Electronics Engineering**

**BONAFIDE CERTIFICATE**

This is to certify that the Industrial Internship Report entitled “**RISC V DESIGN USING VERILOG HDL”** submitted by **Karthick Srivatsa R(20BEC1221)** to VIT, Chennai in partial fulfillment of the requirement for the award of the degree of **Bachelor of Technology** in **Electronics and Communication Engineering** is a record of bonafide industrial internship undertaken by him/her fulfills the requirements as per the regulations of this institute and in my opinion meets the necessary standards for submission. The contents of this report have not been submitted and will not be submitted either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

**Signature of the Examiner Signature of the Examiner**

Date: Date:

**Head of the Department (B.Tech ECE)**

**ACKNOWLEDGEMENT**

The student is free to acknowledge all those he/she feels to acknowledge on the basis of the guidance and help provided during internship.

Karthick Srivatsa R

**(20BEC1221)**

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **Chapter** | **Title** | **Page No** |
|  |  |  |
|  | Declaration |  |
|  | Certificate  Bonafide certificate |  |
|  | Acknowledgement |  |
|  | Table of contents |  |
|  | List of Tables |  |
|  | List of Figures |  |
|  | List of Abbreviation |  |
|  | Abstract |  |
| **1** | **INTRODUCTION** | **11** |
|  | 1.1 About the Company | 11 |
|  | 1.2 About the Project | 12 |
| **2** | **SUB-BLOCKS** | **13** |
|  | 2.1 PC MUX | 13 |
|  | 2.2 Reg Block 1 | 15 |
|  | 2.3 Immediate Generator | 16 |
|  | 2.4 Immediate Adder | 20 |
|  | 2.5 Integer File | 22 |
|  | 2.6 Write Enable Generator | 25 |
|  | 2.7 Instruction MUX | 27 |
|  | 2.8 Branch Unit | 29 |
|  | 2.9 Decoder | 31 |
|  | 2.10 Store Unit | 34 |
|  | 2.11 Load Unit | 36 |
|  | 2.12 ALU | 39 |
|  | 2.13 WB Selection Unit | 41 |
|  | 2.14 Reg Block 2 | 42 |
| **3** | **TOP MODULE** | **45** |
|  | 3.1 Top Module | 45 |
| **4** | **CONCLUSION** | **46** |
|  | **REFERENCES** | **47** |

**LIST OF FIGURES**

|  |  |  |
| --- | --- | --- |
| **Figure No.** | **Figure Description** | **Page No.** |
| 2.1 | PC Mux | 13 |
| 2.2 | Verification output of PC Mux | 15 |
| 2.3 | Reg Block 1 | 15 |
| 2.4 | Verification output of Reg block 1 | 16 |
| 2.5 | Immediate Generator | 16 |
| 2.6 | Verification output of Immediate Generator | 19 |
| 2.7 | Immediate Adder | 20 |
| 2.8 | Verification output of Immediate Adder | 22 |
| 2.9 | Integer File | 22 |
| 2.10 | Verification output of Integer File | 24 |
| 2.11 | Write Enable Generator | 25 |
| 2.12 | Verification output of Write Enable Generator | 26 |
| 2.13 | Instruction Mux | 27 |
| 2.14 | Verification output of Instruction Mux | 28 |
| 2.15 | Branch Unit | 29 |
| 2.16 | Verification output of Branch Unit | 30 |
| 2.17 | Decoder | 31 |
| 2.18 | Verification output of Decoder | 34 |
| 2.19 | Store Unit | 35 |
| 2.20 | Verification output of Store Unit | 36 |
| 2.21 | Load Unit | 36 |
| 2.22 | Verification output of Load Unit | 38 |
| 2.23 | ALU | 39 |
| 2.24 | Verification output of ALU | 40 |
| 2.25 | WB Mux Selection Unit | 41 |
| 2.26 | Verification output of WB Mux Selection Unit | 42 |
| 2.27 | Reg Block 2 | 43 |
| 2.28 | Verification output of Reg Block 2 | 44 |
| 3.1 | Top Module | 45 |
| 3.2 | Verification output of Top Module | 45 |

**LIST OF SYMBOLS AND ABBREVIATIONS**

RISC-V: Reduced Instruction Set Computer - V

ISA: Instruction Set Architecture

UVM: Universal Verification Methodology

**ABSTRACT**

Modern electronics are built on the foundation of Very Large Scale Integration (VLSI) technology, which makes it possible to create complex semiconductor circuits and microchips. The complex process of creating integrated circuits, which are the basic components of modern electronic devices, is known as VLSI design. Register-Transfer Level (RTL) design, a methodology that specifies the data flow between registers in a digital system, is one of the fundamental components of VLSI design. In order to create effective and useful integrated circuits, RTL design plays a crucial role in defining how data is handled and processed inside digital systems.

The results of our online certification-based internship program in semiconductor design and Very Large Scale Integration (VLSI) technology are summarized in this report. Maven Silicon Softech Ltd., a respectable business recognized for its proficiency in semiconductor design and VLSI training, oversaw our virtual internship. The main goal of our project was to create a three-stage pipeline processor using the RV32I instruction set architecture for RISC-V. Because of its flexibility and adaptability, RISC-V is a great platform for computer architecture research and instruction. Using Verilog HDL and the Universal Verification Methodology (UVM), we designed, verified, and integrated a number of sub-modules during this online internship, including the PC Mux, Register Blocks, Immediate Generator, ALU, and more. This virtual experience not only deepened our understanding of semiconductor design but also cultivated crucial skills for future careers in the semiconductor industry. Despite the remote nature of this internship, it successfully bridged the gap between theoretical knowledge and practical application, equipping us with invaluable expertise in semiconductor design.

**CHAPTER 1**

**INTRODUCTION**

**1.1 ABOUT THE COMPANY**

Maven Silicon Softech Ltd. is a company that specializes in providing training and education in the field of semiconductor design and VLSI (Very Large Scale Integration) technology. VLSI technology involves the design and development of integrated circuits and microchips, which are fundamental components of modern electronics.

Maven Silicon Softech offers various courses and programs related to VLSI design and semiconductor engineering. These courses typically cover a range of topics, including digital design, analog design, physical design, verification, and more. They often include both theoretical and practical components, enabling students to gain hands-on experience in designing and testing integrated circuits.

The company is known for its training programs, workshops, and online courses that cater to individuals looking to enter the semiconductor industry, as well as professionals seeking to enhance their skills in VLSI design. Their courses are designed to prepare students for careers in semiconductor companies and help them stay updated with the latest advancements in the field.

**1.2 ABOUT THE PROJECT**

RISC-V is a new instruction-set architecture designed for research and education in computer architecture. It is ideal for direct native hardware implementation and enables efficient performance in any microarchitecture style or technology. It has a small base integer ISA, optional standard extensions, extended ISA extensions, and customized variations. It also supports extensive ISA extensions and specialized variants, both 32-bit and 64-bit address space variants for applications, operating system kernels, hardware implementations, and highly-parallel multicore or many core implementations, including heterogeneous multiprocessors. Additionally, it has optional variable-length instructions to expand available instruction encoding space and support an optional dense instruction encoding for improved performance, static code size, and energy efficiency. The RISC-V processor provides a variety of instruction sets, including integer computational instructions, integer loads, integer stores, control-flow instructions, memory ordering instructions, unconditional leaps, conditional branches, and HINT instructions. Using these instructions enables tasks to be executed in less time, making the system more efficient.

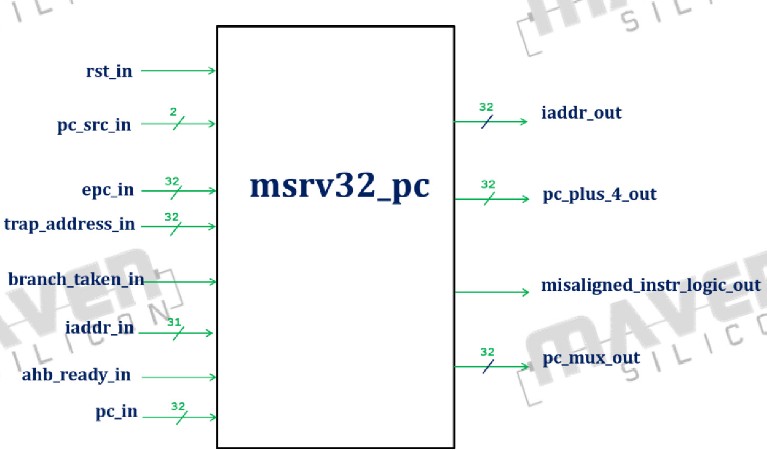
The project made used RISC-V’s RV32I instruction set architecture to build a three-stage pipeline processor. In this project several sub-blocks were made and verified individually using Verilog HDL, then the sub-blocks were integrated into a top module wrapper. This top module was then verified with Universal Verification Methodology (UVM). For this a System Verilog code was provided.

# PC MUX:

**CHAPTER 2**

**SUB - BLOCKS**

**Input ports:** rst\_in,pc\_src\_in,ahb\_ready\_in,epc\_in,trap\_address\_in branch\_taken\_in,iaddr\_in,pc\_in

**Output ports:** pc\_plus\_4\_out, pc\_mux\_out, misaligned\_instr\_logic\_out and iaddr\_out

The PC MUX (Programme Counter Multiplexer) block is a crucial part of the processor pipeline's instruction fetch stage in the RISC-V architecture. It is in charge of choosing the address of the subsequent instruction to be fetched and run. To choose the right address, the PC MUX considers a variety of control signals and circumstances.

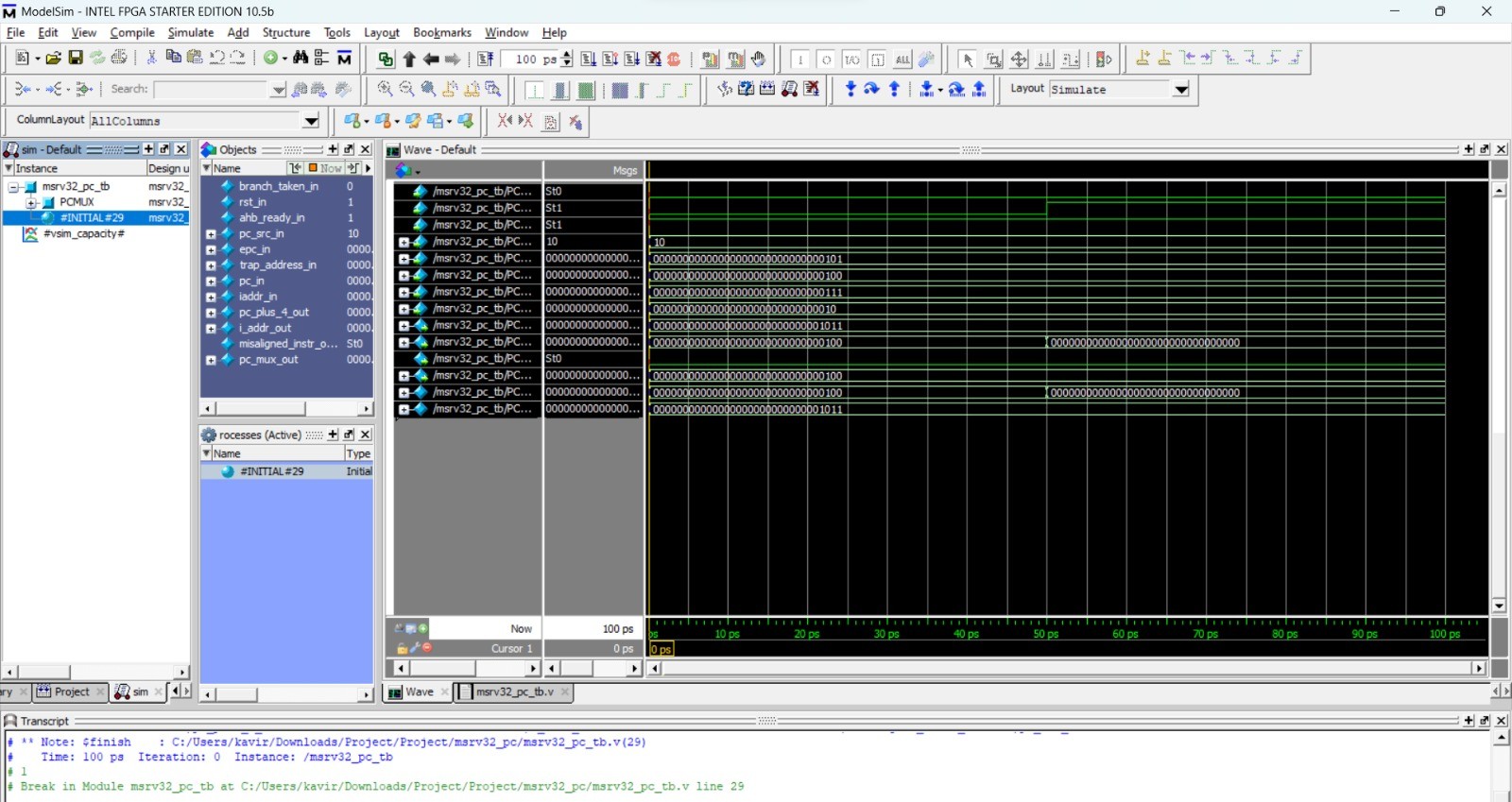
Branch conditions, jump conditions, exceptions, interrupts, and the PC MUX control signals themselves are among the control signals and conditions that direct the PC MUX's selection logic.

In order to maintain proper programme flow within the processor pipeline, the PC MUX block is essential. Programmes can proceed as intended because it makes sure the right instruction addresses are fetched and executed.

**Functionality:** This module will be used to hold the address of next instruction to be executed.

1. 'i\_addr\_out' is used to interface with instruction memory.
2. Based on the current state of the core, the program counter will be loaded.
   * When the interrupt happens then PC should go to trap\_address\_in.
   * After compilation of interrupt PC should go back to epc\_in.
   * For normal operation PC should go to new address ({iaddr\_in [31:1], 1'b0}) if there is a branch instruction to be executed or else 'pc\_in + 32'h00000004'.
   * For reset state PC should get `BOOT\_ADDRESS. (BOOT\_ADDRESS is a parameter of 32'h00000000 value).
3. 'pc\_plus\_4\_out' will be assigned with 'pc\_in + 32'h00000004'.
4. 'misaligned\_instr\_logic\_out' will be generated by doing 'AND' operation between 'branch\_taken\_in' and first bit of next\_pc.
5. If there is a reset then iaddr\_out should get `BOOT\_ADDRESS else if it checks if the 'ahb\_ready\_in' is high, then iaddr\_out gets pc\_mux\_out.

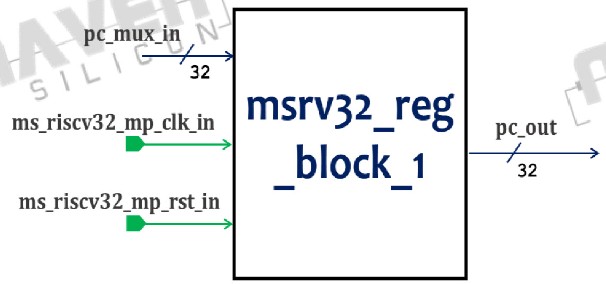
**Output:**



# REG BLOCK -1

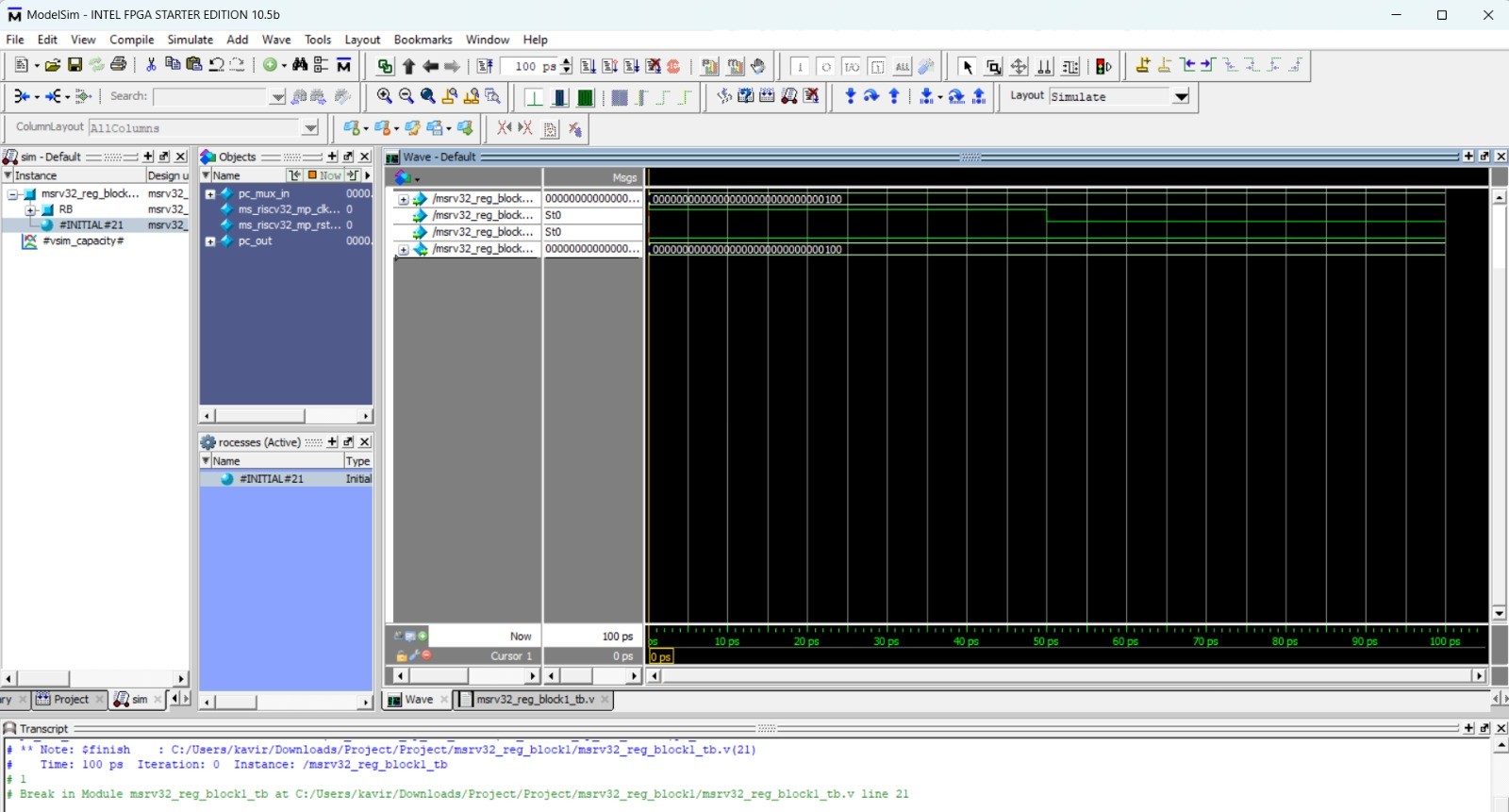
**Input ports:** pc\_mux\_in,ms\_riscv32\_mp\_clk\_in,ms\_riscv32\_mp\_rst\_in

**Output ports:**pc\_out



**Functionality:** It registers the pc\_mux\_in and produces the output at the posedge of clk if there is no reset.

**Outputs:**



# IMMEDIATE GENERATOR:

**Input ports:**instr\_in,imm\_type\_in **Output ports:** imm\_out

The Immediate Generator is a component that creates constants or immediate values for use as operands in instructions. Immediate arithmetic and logical operations, immediacy of addressing, and immediacy of branching conditions all make use of the immediate values.

For encoding and extending immediate values to the required bit width for instructions, RISC-V's Immediate Generator offers a mechanism. Different formats for encoding immediate values are defined by the RISC-V instruction set architecture depending on the type of instruction and operand size.

Immediate Formats: Numerous immediate formats are supported by RISC-V, including sign-extended, zero-extended, and compressed immediate. The range and encoding method for constant value representation are determined by the immediate formats.

Immediate Values: In RISC-V, immediate values can range in size from 12 to 20 bits. However, the specific instruction and its encoding format determine the size and range in detail. It may be necessary to use additional instructions or techniques, such as immediate splitting or multiple instruction sequences, for instructions with immediate values that are larger than the limit.

Sign Extension: Immediate values occasionally need to have their signs extended in order to match the size of the operands. The process of "sign extension" involves copying the immediate value's sign bit to fill the remaining bits. It guarantees that the immediate value is recognised as a signed value in the proper manner.

Encoding Scheme: Immediate values are typically encoded using a specific encoding scheme that adheres to the binary format of the instruction. The encoding

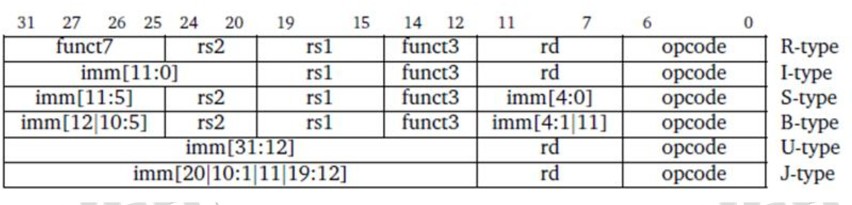
scheme determines how the immediate bits are placed within the instruction encoding, ensuring proper extraction and decoding during instruction execution.

The Immediate Generator allows instructions to work with immediate constants efficiently and avoids the need for separate memory accesses to retrieve immediate values.

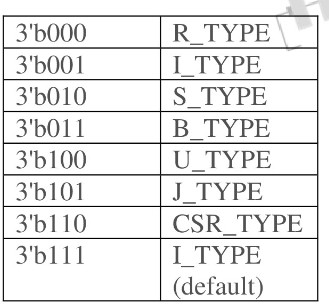
Efficient use of immediate values can help optimize code size, improve performance, and reduce memory accesses in RISC-V processors.

# Functionality:

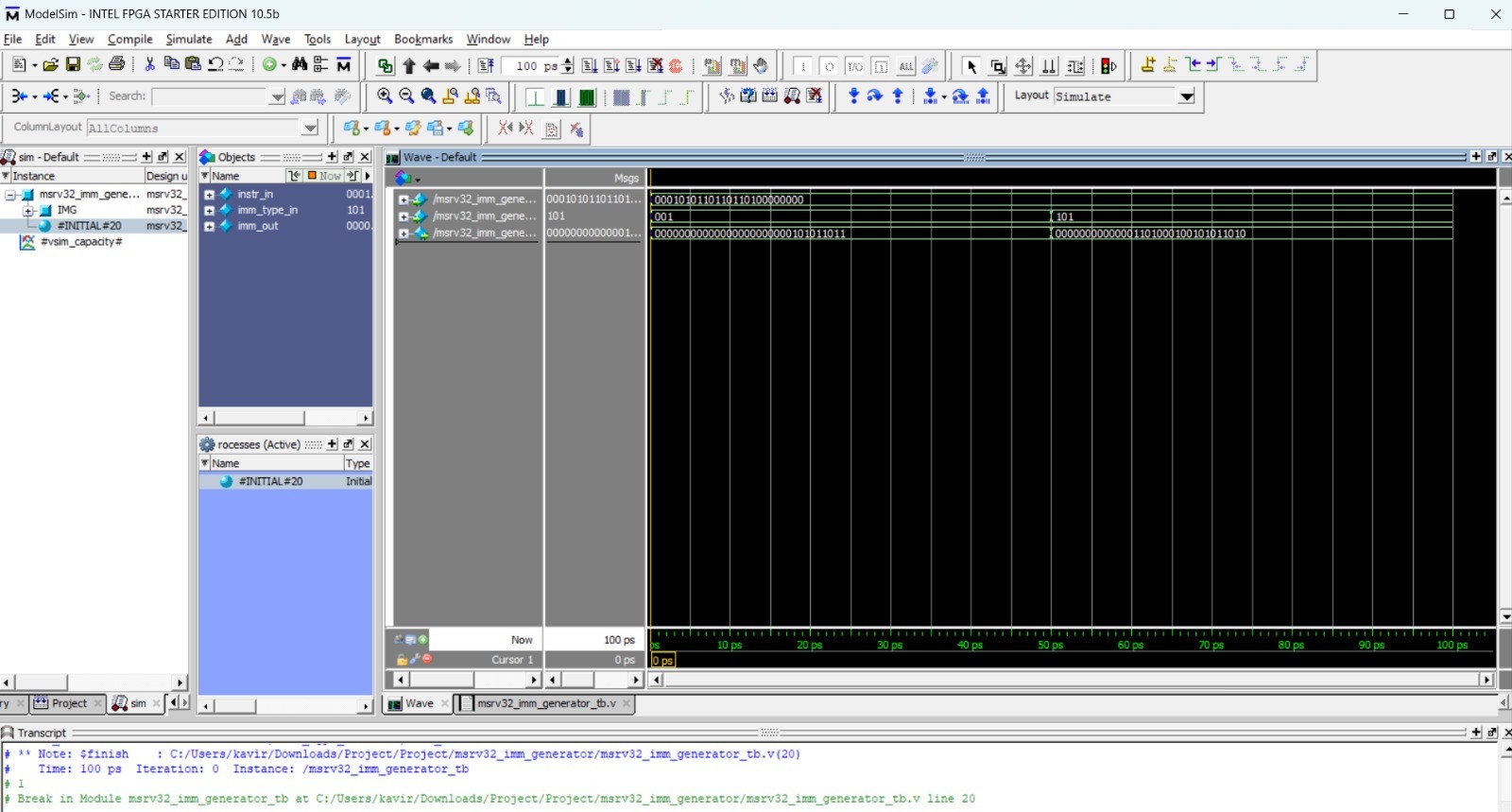
The Immediate Generator rearranges the immediate bits contained in the instruction and, if necessary sign-extends it to form a 32-bit value. The unit is controlled by the imm\_type\_out signal, generated by the Control Unit. Table shows the unit input and output signals. The imm\_type\_in is generated by msrv32\_decoder module.

The below table depicts the region of immediate data for different types of instructions

From the instruction set we can conclude that opcode[6:4] is used to distinguish between different types of instructions. imm\_out is assigned according to the below given table-



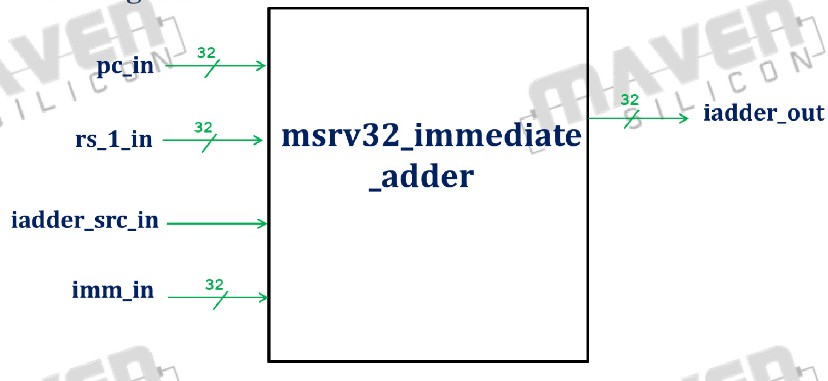
**Outputs:**



# IMMEDIATE ADDER:

**Input ports:** pc\_in,rs\_1\_in, iadder\_src\_in , imm\_in

**Output ports:** iadder\_out



When an immediate value needs to be added to a register value, the immediate value is first sign-extended or zero-extended based on the immediate format. Then, the ALU performs the addition operation, adding the immediate value to the contents of the specified register.

Sign Extension or Zero Extension: The immediate value is extended to match the operand size of the ALU operation. Sign extension is performed when the immediate value is treated as a signed value, while zero extension is used for unsigned values. The extension ensures that the immediate value is correctly aligned and ready for addition.

Immediate Encoding: The immediate value is encoded within the instruction itself. The immediate bits are placed in specific positions within the instruction's binary format.

Operand Selection: The immediate value is typically combined with the contents of a specific register. The register is selected based on the instruction's encoding and the desired operation.

Result Storage: The result of the immediate addition operation is typically stored back into a register. The destination register is specified in the instruction, allowing the processor to store the updated value after the addition operation.

**Functionality:** The output of immediate adder module will be 32 bit address depending on the instruction format (instruction to the core). This module will add the 'immediate' value with 'rs1' when core wants to perform load, store, jump and link reg (jalr) instructions. The core is performing 'branch' instruction and jump and link (jal) then 'immediate' value will be added in a 'PC'. So that core will jump on new address with the help of program counter. When core 'iadder\_src\_in' is high then 'iadder\_out' =rs\_1\_in + imm\_in else 'iadder\_out' = pc\_in + imm\_in.

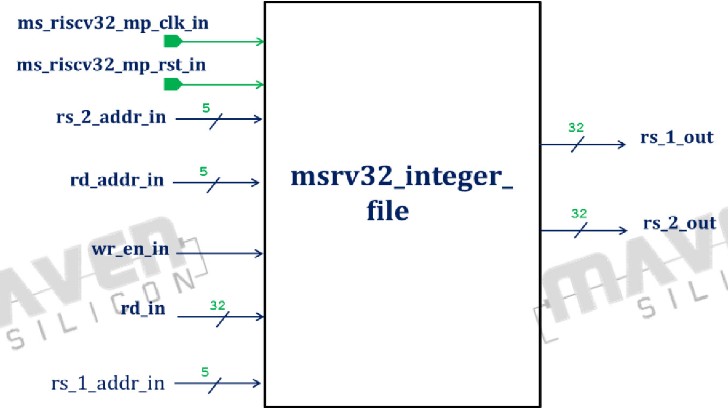
# Outputs:



**INTEGER FILE**

**Input ports:** rs\_1\_addr\_in , rs\_2\_addr\_in ,rd\_addr\_in ,rd\_in , wr\_en\_in

**Output ports:** rs\_1\_out ,rs\_2\_out



A bank of registers called the Integer Register File in the RISC-V architecture is used to store integer data. During the execution of instructions, it acts as the main storage for intermediate values, operands, and results. High-performance arithmetic and logical operations are made possible by the register file, which offers quick and effective access to data.

General-Purpose Registers: There are several general-purpose registers (GPRs) in the integer register file. There are 32 GPRs in RISC-V, denoted by the letters x0 to x31. These 32-bit registers can store addresses, integer values, and other data for arithmetic and logical operations.

Read and Write Ports:Multiple read and write ports are typically provided by the register file. This feature supports pipelining and allows for effective parallel instruction execution.

Register Numbering: A distinct register number is given to each register in the file. These register numbers are used in instructions to specify the source and destination registers.

Operand Access: By specifying the source register numbers, instructions can read data from the register file. The register file retrieves the needed information and gives it to the execution units, like the multiplier or divider or ALU, as operands. The destination register number specified in the instruction is used to write the results generated by the execution units back into the register file.

Particular Registers: The Integer Register File may also contain special-purpose registers in addition to general-purpose registers. These registers, including the programme counter (PC), the stack pointer (SP), the frame pointer (FP), and the control/status registers (CSR), each have a specific function.

# Functionality:

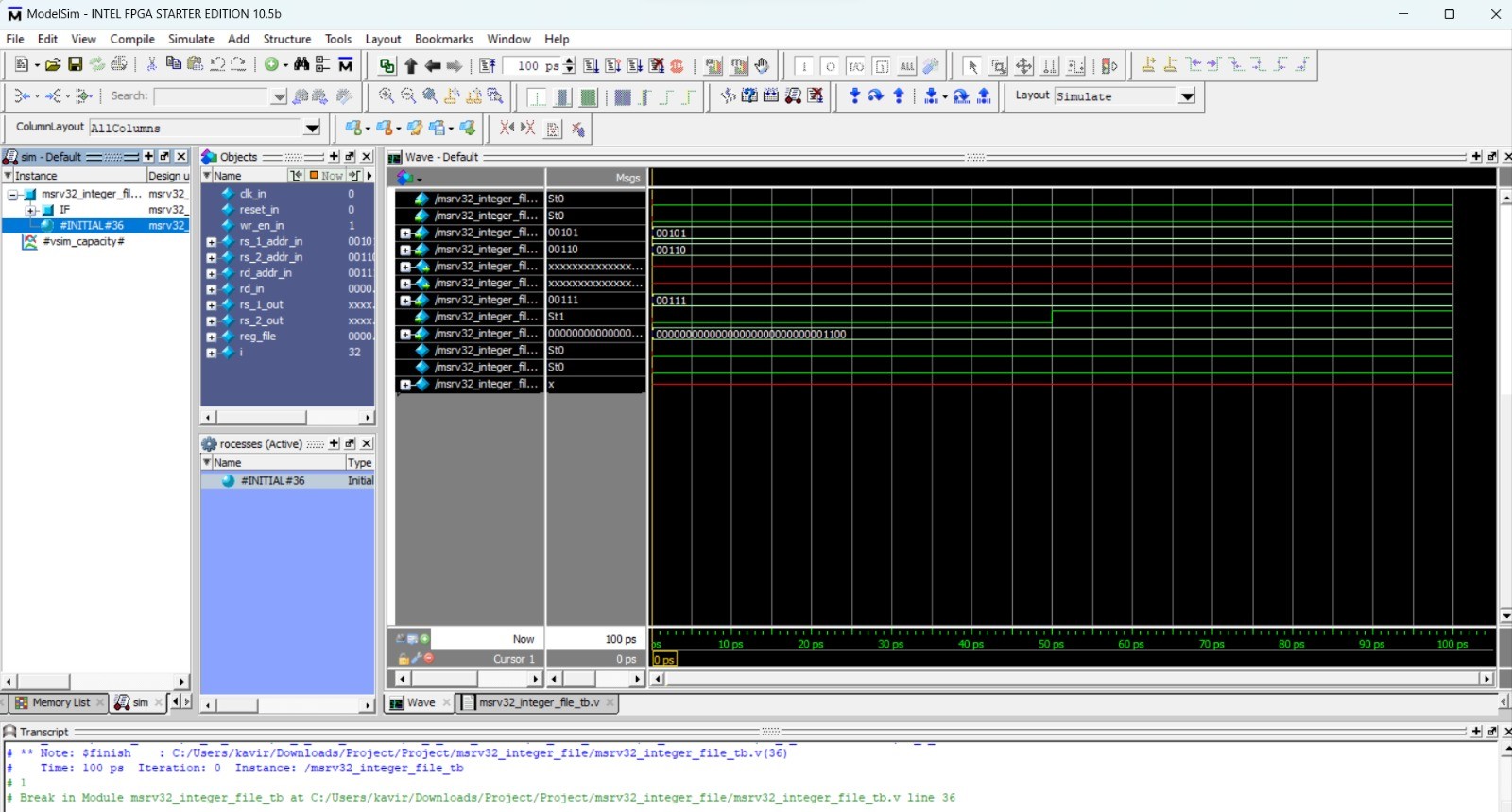
The msrv32\_integer\_file has 32 general-purpose registers and supports read and write operations. Reads are requested by pipeline stage 2 and provide data from one or two registers. Writes are requested by stage 3 and put the data coming from the Write back Multiplexer into the selected register.

Initially all registers can be initialized with 0. The first register is hardwired with 0. No write operations are permitted for reg\_file [0] location.

If stage 3 requests to write to a register being read by stage 2, the data to be written is immediately forwarded to stage 2 to avoid data hazard. This can be done by forwarding the rd \_in directly to rs\_1\_out and rs\_2\_out instead of reading from reg\_file.

If rs\_1\_addr\_in is equal to rd\_addr\_in and wr\_en\_in is asserted then drive rs\_1\_out with rd\_in else reg\_file [rs\_1\_addr\_in] .If rs\_2\_addr\_in is equal to rd\_addr\_in and wr\_en\_in is asserted then drive rs\_2\_out with rd\_in else reg\_file [rs\_2\_addr\_in]

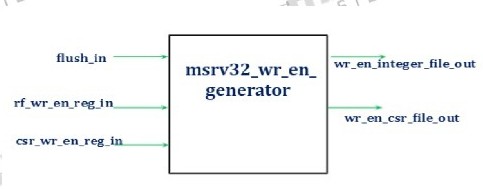
# Outputs:



**WRITE ENABLE GENERATOR:**

**Input Ports:** flush\_in, rf\_wr\_en\_reg\_in, csr\_wr\_en\_reg\_in

**Output Ports:** wr\_en\_integer\_file\_out, wr\_en\_csr\_file\_out



Depending on the instruction being executed, the Write Enable Generator (WEG) determines whether a register write operation should be carried out. It is a crucial part of the processor's control unit and is in charge of overseeing register updates. The Write Enable Generator determines whether or not the register's write-back should be enabled. The opcode and other pertinent fields of the instruction being executed are examined by the WEG in order to function. The majority of operations within the RISC-V ISA are carried out between registers thanks to its load-store architecture. The WEG decides whether or not to enable the register write when an instruction, such as an arithmetic operation or a memory load, calls for writing a value to a register. The WEG works by inspecting the opcode and other pertinent fields of the instruction being executed. It examines the instruction type and decides if the destination register indicated in the instruction should be altered. This choice is based on the semantics of the instruction and the desired behavior stated by the RISC-V ISA.

**Functionality:**

1. Instruction Analysis: As input, the WEG receives the opcode of the instruction being executed. It examines the opcode and other relevant fields to determine the instruction's type and semantics.
2. Register Write Decision: The WEG determines whether or not to perform a register write operation based on the instruction type and semantics. It determines whether the instruction's specified destination register should be updated.
3. Write Enable Signal Generation: As an output, the WEG generates the Write Enable (WE) signal. The WE signal indicates whether the current instruction's register writes should be enabled or disabled.
4. ISA Compliance: The WEG ensures that the register write behavior follows the RISC-V ISA semantics. It correctly interprets the instruction type and performs the register write operation per the architectural specifications.

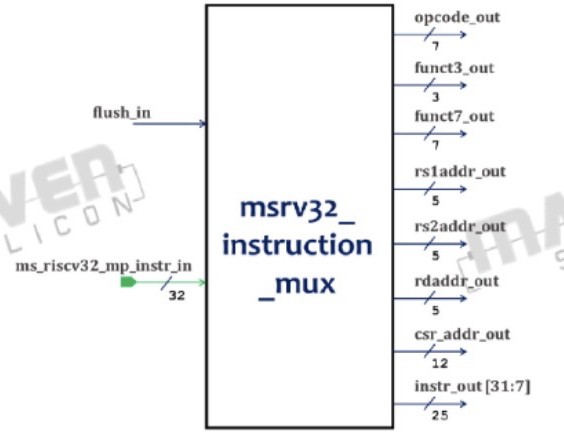
**Outputs:**



# INSTRUCTION MUX:

**Input Ports:** flush\_in, ms\_riscv32\_mp\_instr\_in.

**Output Ports:** opcode\_out, funct3 out, funct7\_out, rs1addr\_out, rs2addr\_out, rdaddr\_out, Csr\_addr\_out, instr\_out [31:7]



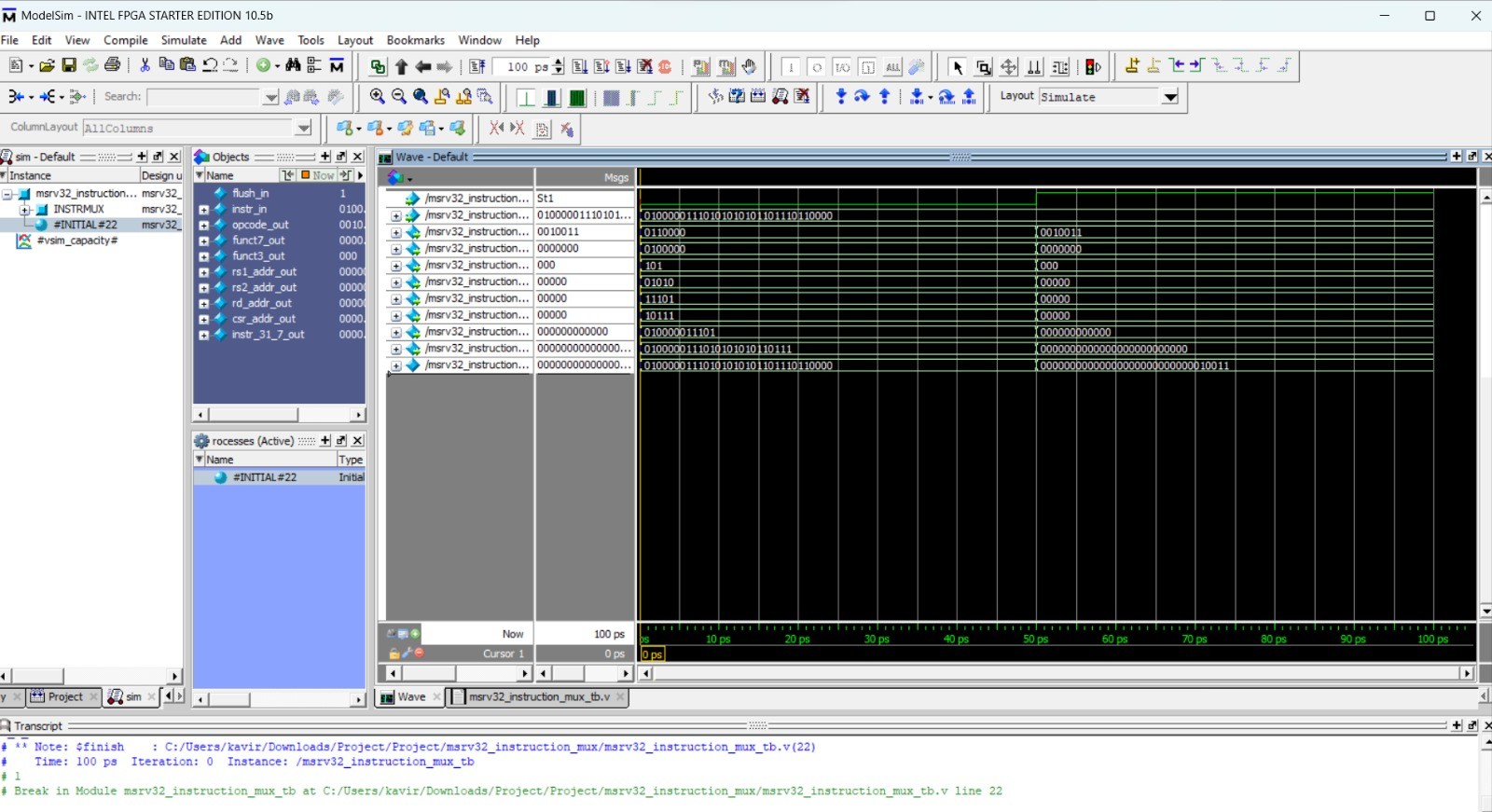
The microarchitecture of a RISC-V processor contains a submodule called the Instruction MUX that is in charge of choosing the best instruction to carry out. It is crucial for retrieving and decoding instructions from memory as well as making sure the right instruction is passed on to later execution stages. The current state of the processor and the control signals produced by the control unit determine the selection signal for the Instruction MUX. As a result, the processor can use the PC value to determine the next instruction to be retrieved from memory. The Instruction MUX block is a part of the RISC-V Processor's Fetch Stage, which is the first stage of Pipelining. The Instruction MUX plays a critical role in maintaining the flow of instructions within the processor and facilitating proper instruction fetch and decoding.

**Functionality:**

This module takes the instruction (input to the core) and provides the fields which are used by the other modules to perform their functionality.

1. When 'flush' is high use 32'h00000013 to provide the fields.
2. When 'flush' is low use core instruction (ms\_riscv32\_mp\_instr\_in) to provide the fields.
3. opcode\_out = instr\_mux [6:0] funct3\_out = instr\_mux [14:12] funct7\_out = instr\_mux [31:25] csr\_addr\_out = instr\_mux [31:20] rs laddr\_out = instr\_mux [19:15] rs2addr\_out = instr\_mux [24:20] rdaddr\_out = instr\_mux [11:7] instr\_out instr\_mux [31:7]

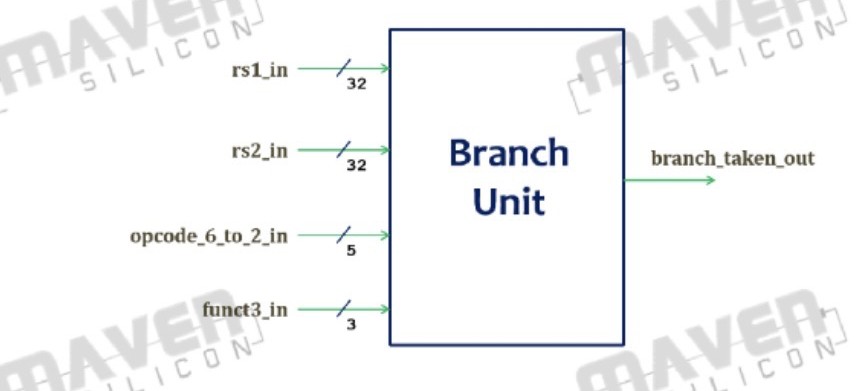
**Outputs:**



**BRANCH UNIT:**

**Input Ports:** rs1\_in, rs2\_in, opcode\_6\_to\_2\_in, funct3\_in

**Output Ports:** branch\_taken\_out



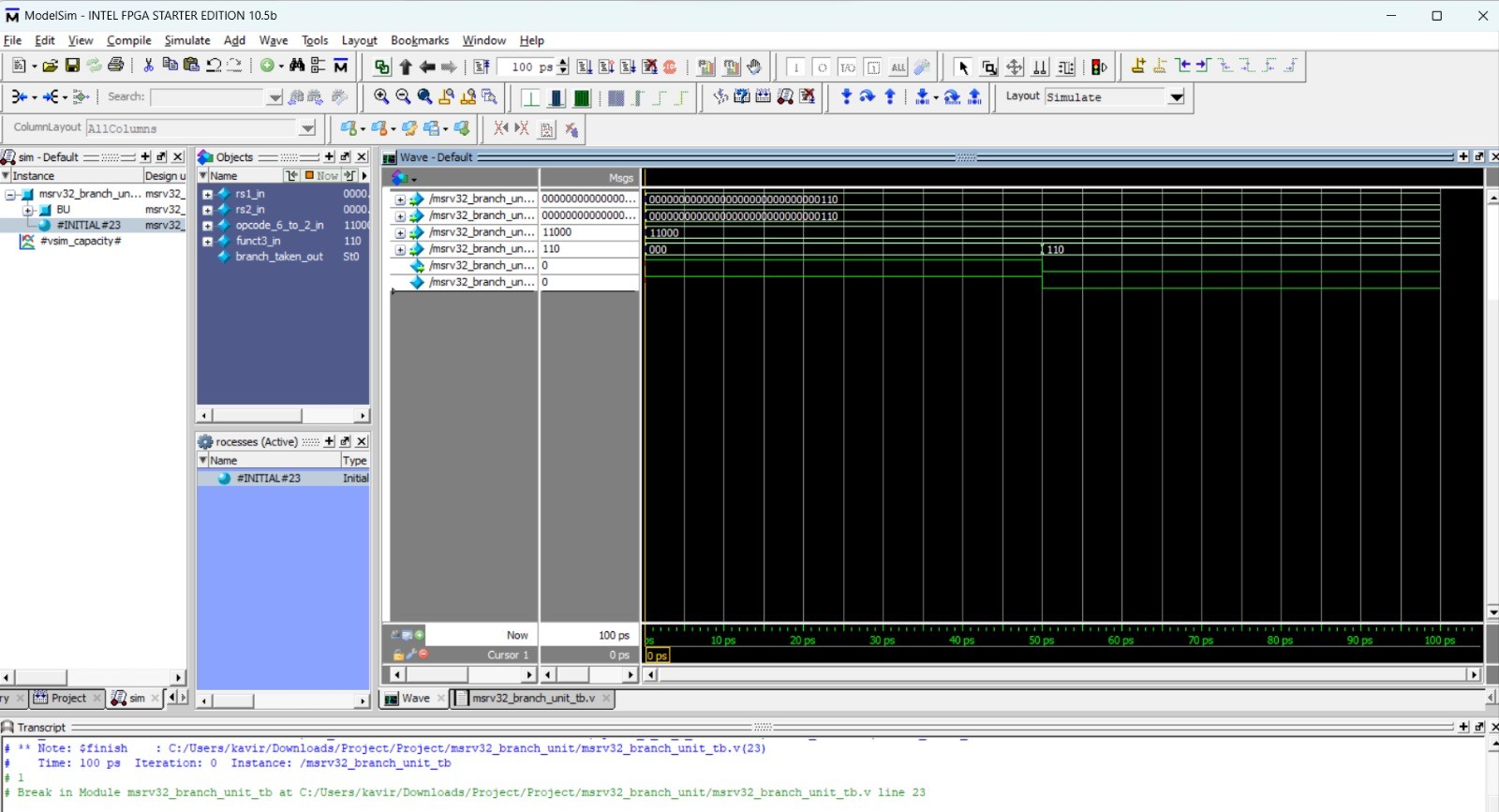
A branch instruction in the RISC-V instruction set is handled by the Branch Unit, a submodule of the microarchitecture of a RISC-V processor. It is crucial for predicting how branch conditions will turn out and for managing the program's flow. This unit performs a number of functions, including Control Flow Management, Branch Prediction, and Pipeline Synchronisation. In order to make sure the right instructions are fetched and executed in the right sequence, it coordinates the fetch of instructions, branch condition evaluation, and programme counter updates. The microarchitecture of a RISC-V processor includes a Branch Unit that manages branch instructions, assesses branch conditions, determines branch target addresses, controls programme control flow, and makes sure pipeline synchronisation is correct. It is an essential part of the RISC-V instruction set for managing programme execution and enabling conditional branching. It is significant because it assists in Control Flow, Increases the Program efficiency, Loop Optimization, and most importantly the Pipeline Optimization.

The Branch Unit block belongs to the Execute Stage i.e the second stage of Pipelining in the RISC - V Processor.

**Functionality:**

The Branch Unit decides if a branch instruction must be taken or not. It receives two operands from the Integer Register File and, based on the value of opcode and funct3 instruction fields, it decides the branch. Branch conditions are jumps. Jum conditional jumps. Jump instructions are interpreted as unconditional jumps that must always be taken. Internally, the unit realizes just two comparisons, deriving other four from them.

**Outputs:**



# DECODER:

**Input Ports:** trap\_taken\_in, funct7.5\_in, opcode\_in, funct3\_in, ladder\_out\_1\_to\_0\_ in.

**Output Ports:** wb\_mux\_sel\_out, imm\_type\_out, csr\_op\_out, mem\_wr\_req\_out, alu\_opcode\_out, load\_size\_out, load\_unsigned\_out, alu\_src\_out, ladder\_src\_out, csr\_wr\_en\_out, rf\_wr\_en\_out, illegal\_instr\_out, misaligned\_load\_out, misaligned\_store\_out.

The Decoder block is a part of the RISC-V processor's Execute Stage, which is the second stage of pipelining. The main building block of the entire RISC-V 32ISA Processor is the decoder block. Decoding the fetched instruction depends on the decoder block, which is a component of the Instruction Fetch step. To determine the type of instruction being processed, it assesses the opcode and other relevant

data. By decoding the instruction, the decoder block determines the precise operation to be carried out and retrieves any necessary operand values. It generates control signals that the pipeline stages after it will use to precisely carry out the command. The decoded instruction is subsequently sent to the Execute stage, where the relevant actions are done based on the instruction type and operand values.

**Functionality:**

The Decoder decodes the instruction and generates the signals that control the memory, the Load Unit, the Store Unit, the ALU, the two register files (Integer and CSR), the Immediate Generator and the Write back Multiplexer.

Note: The output imm\_type\_out should be generated as per the imm\_generator module & wb\_mux\_sel\_out per wb\_mux\_sel\_unit module.

The output logic for decode is given below.

1. alu\_opcode\_out[2:0] is assigned to funct3\_in
2. alu\_opcode\_out[3] = funct7\_5\_in& -(is\_addi | is\_slti | is\_sltiu | is\_andi | is\_ori | is\_xori) c) load\_size\_out is asserted for funct3\_in[1:0] and load\_unsigned\_out is asserted for

funct3\_in[2] d) alu\_src\_out is asserted for 5th bit of opcode\_in

1. iadder\_src\_out is enabled if either is load, is\_store or is\_jalr occurs.
2. csr\_wr\_en\_out is enabled if is csr occurs.
3. rf\_wr\_en\_out is\_lui | is\_auipe | is\_jalr | is\_jal | is\_op | is\_load | is\_csr | is\_op\_imm ON
4. wb\_mux\_sel\_out

wb\_mux\_sel\_out [0]= is load | is\_auipe | is\_jal | is\_jalr wb\_mux\_sel out [1]= is\_lui | is\_auipc

wb\_mux\_sel\_out [2]= is csr | is\_jal | is\_jalr

1. imm\_type\_out

imm\_type\_out [0] = is\_op\_imm | is\_load | is\_jalr | is\_branch | is\_jal

imm\_type\_out [1]= is\_store | is\_branch | is\_csr imm\_type\_out [2]= is\_lui | is\_auipe

| is\_jal | is\_csr

1. is implemented\_instr is enabled if any of the valid opcode occurs.
2. csr\_op\_out is asserted by funct3\_in.
   1. misaligned\_load\_out is asserted if mal\_word or mal\_half occurs along with is load. m) misaligned\_store\_out is asserted if mal\_word or mal\_half occurs along with is\_store. n) mem\_wr\_req\_out is enabled if is\_store occurs and trap\_taken in, mal\_word and

mal\_half are low.

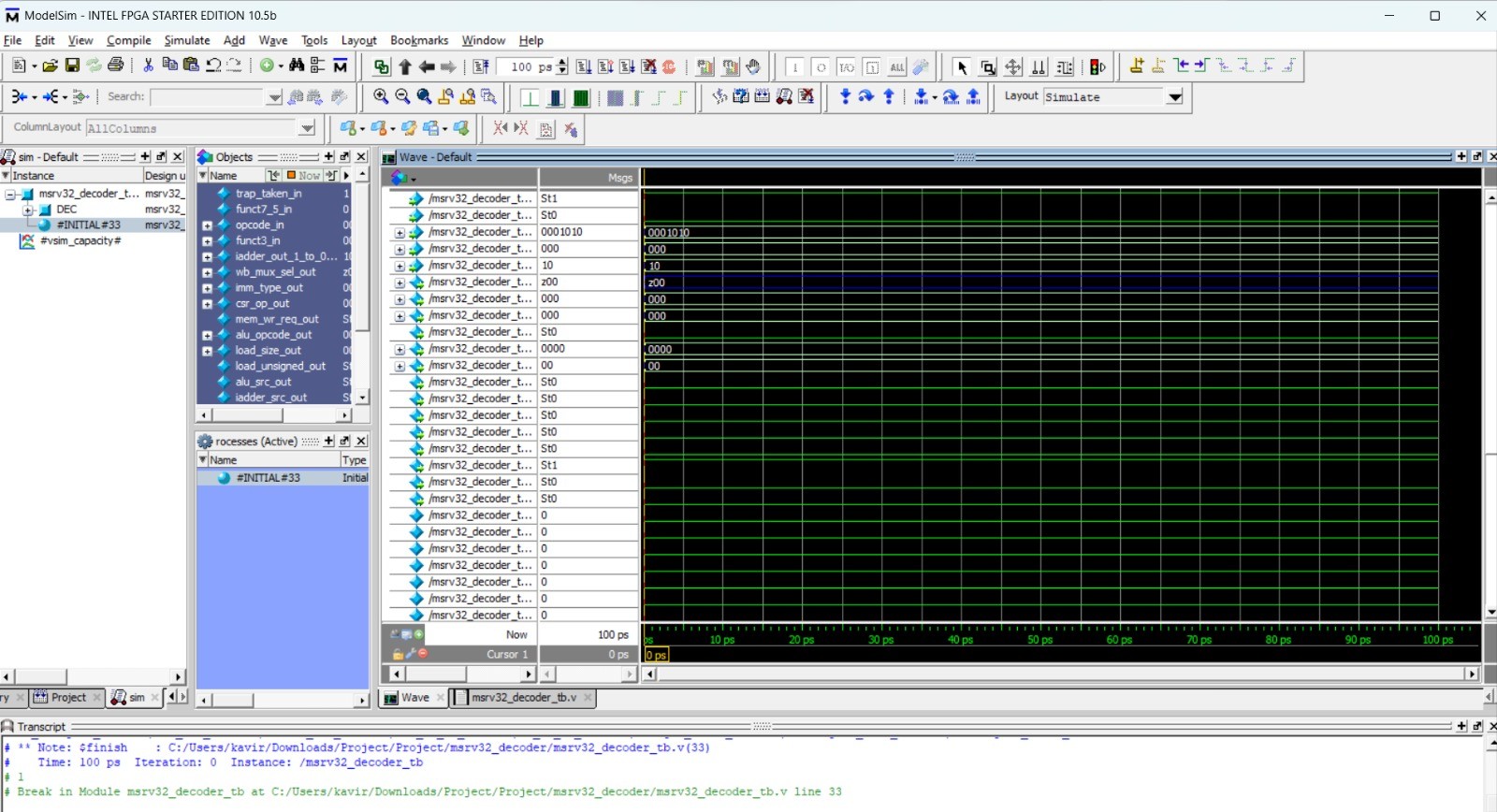
The logic for internal wires are given below.

Internal wire mal word is asserted if funct3\_in refers to word and if iadder\_1\_to\_0\_in is not zero.

Internal wire mal\_half is asserted if funct3\_in refers to half-word and if iadder\_1\_to\_0\_in [0] is not zero.

is\_csr= is\_system& (funct3\_in [2] | funct3\_in [1] | funct3\_in [0])

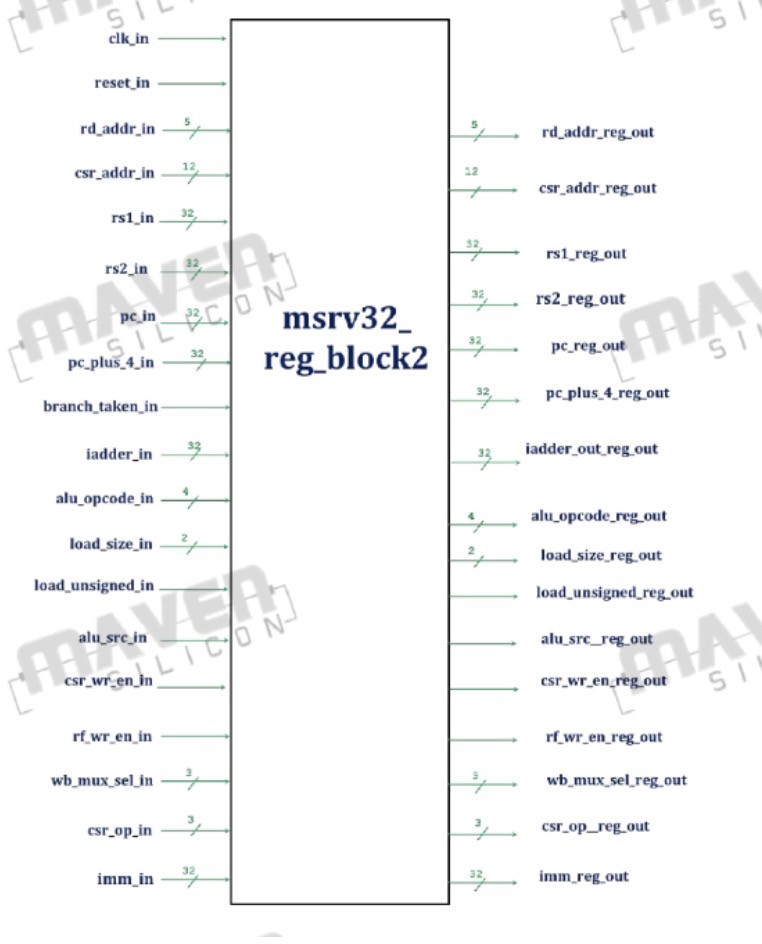
**Outputs:**



# REG BLOCK 2:

**Input Ports:** cik\_in, reset\_in, rd\_addr\_in, csr\_addr\_in, rs1\_in, rs2\_in, pc\_in, pc\_plus\_4\_in, branch\_taken\_in, ladder\_in, alu\_opcode\_in, load\_size\_in, load\_unsigned\_in, alu\_src\_in, csr\_wr\_en\_in, rf\_wr\_en\_in, wb\_mux\_sel\_in, csr\_op\_in, imm\_in.

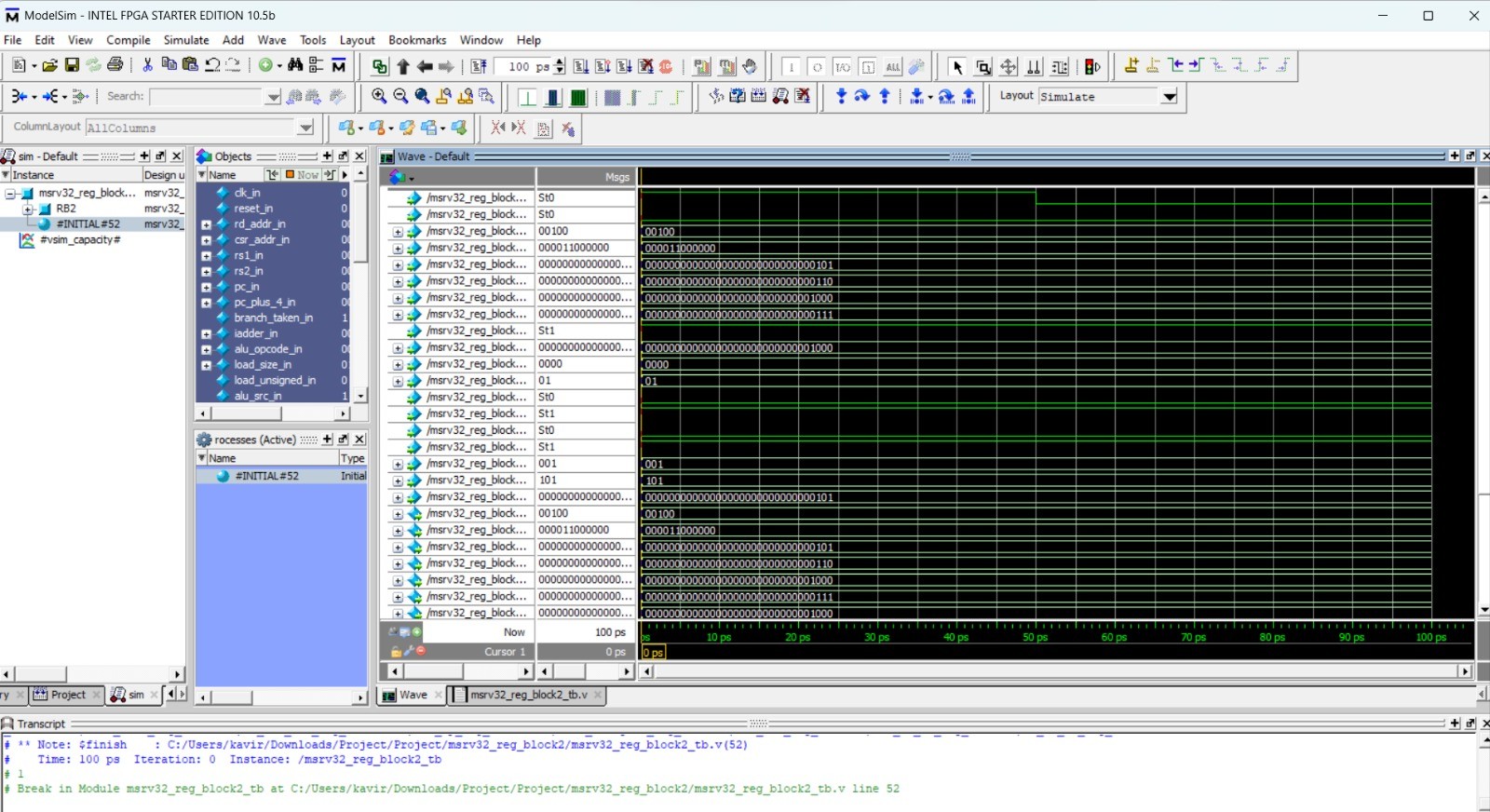
**Output Ports:** rd\_addr\_reg\_out, csr\_addr\_reg\_out, rs1\_reg\_out, rs2\_reg\_out, pc\_reg\_out, pc\_plus\_4\_reg\_out, iadder\_out\_reg\_out, alu\_opcode\_reg\_out, load\_size\_reg\_out, load\_unsigned\_reg\_out, alu\_src\_reg\_out, csr\_wr\_en\_reg\_out, rf\_wr\_en\_reg\_out, wb\_mux\_sel\_reg\_out, csr\_op\_reg\_out, imm\_reg\_out.



**Functionality:**

It registers all the inputs and produces the outputs at the posedge of elk if there is no reset. The block also integrates a 2:1 MUX with select-line as branch\_taken\_in. The LSB of ladder\_out\_reg\_out is assigned with O if branch\_taken\_in is 1, else iadder\_out\_reg\_out [O] is assigned with registered value of laddr\_in [0].

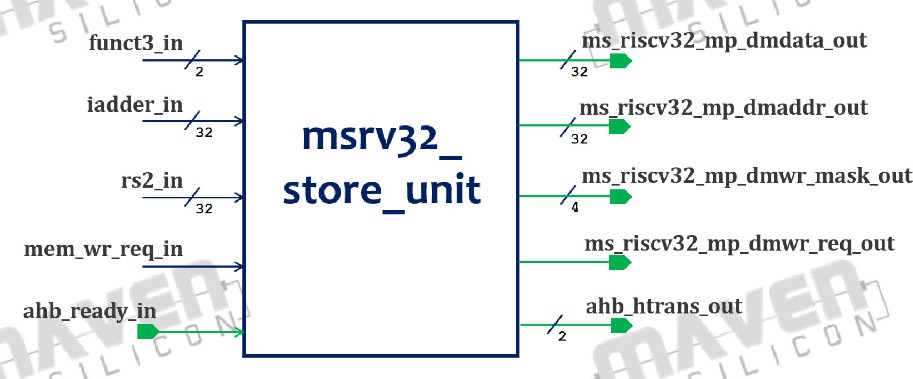
**Outputs:**



# STORE UNIT:

**Input Ports:**funct3\_in iadder\_in,ahb\_ready\_in,rs2\_in mem\_wr\_req\_in

**Output Ports:**ms\_riscv32\_mp\_dmdata\_out, ms\_riscv32\_mp\_dmaddr\_out, ms\_ riscv32\_mp\_dmwr\_mask\_out, ms\_riscv32\_mp\_dmwr\_req\_out, ahb\_htrans\_out



The organisation and addressing of memory is done by the RISC-V ISA (Instruction Set Architecture), which is referred to as the storage unit. The RISC-V architecture supports the manipulation of single bytes in memory by utilising a byte-addressable memory system. The RISC-V ISA is capable of supporting a wide range of data types and storage devices. The three main types of storage for data are Byte, Word, and Half-Word.These storage units are used to organise and manage data within the memory subsystem of the RISC-V architecture. The specific instructions and addressing modes provided by the ISA permit accessing, loading, and storing data in these storage units depending on the requirements of the programme being executed.

**Functionality:**

The signals that interface with the external data memory are driven by the store unit.It correctly positions the data to be written in data\_out (which can be a byte, half-word, or word), and it sets the value of wr\_mask\_out in a suitable manner.

1. Depending on the funct3\_in signal & ahb\_ready\_in, data\_out signal will get values.If data\_hready\_in is high, then if funct3\_in=2'b00 a byte of data will be stored in data\_out depending on the iaddr\_in [1:0].

For ex. If iaddr\_in=2'b00 then data\_out = {8'b0, 8'b0, 8'b0, rs2\_in[7:0]}, iaddr in =2'b01 then data\_out = {8'b0, 8'b0, rs2\_in[15:8], 8'60},

For funct3\_in=2'b01, a half word of data will be stored in data\_out (depending on the iaddr\_in [1]. For ex. iaddr\_in [1]=1'b1 then data\_out= (rs2\_in [31:16], 16'b0} and for other combinations rs2\_in will be stored in data\_out.

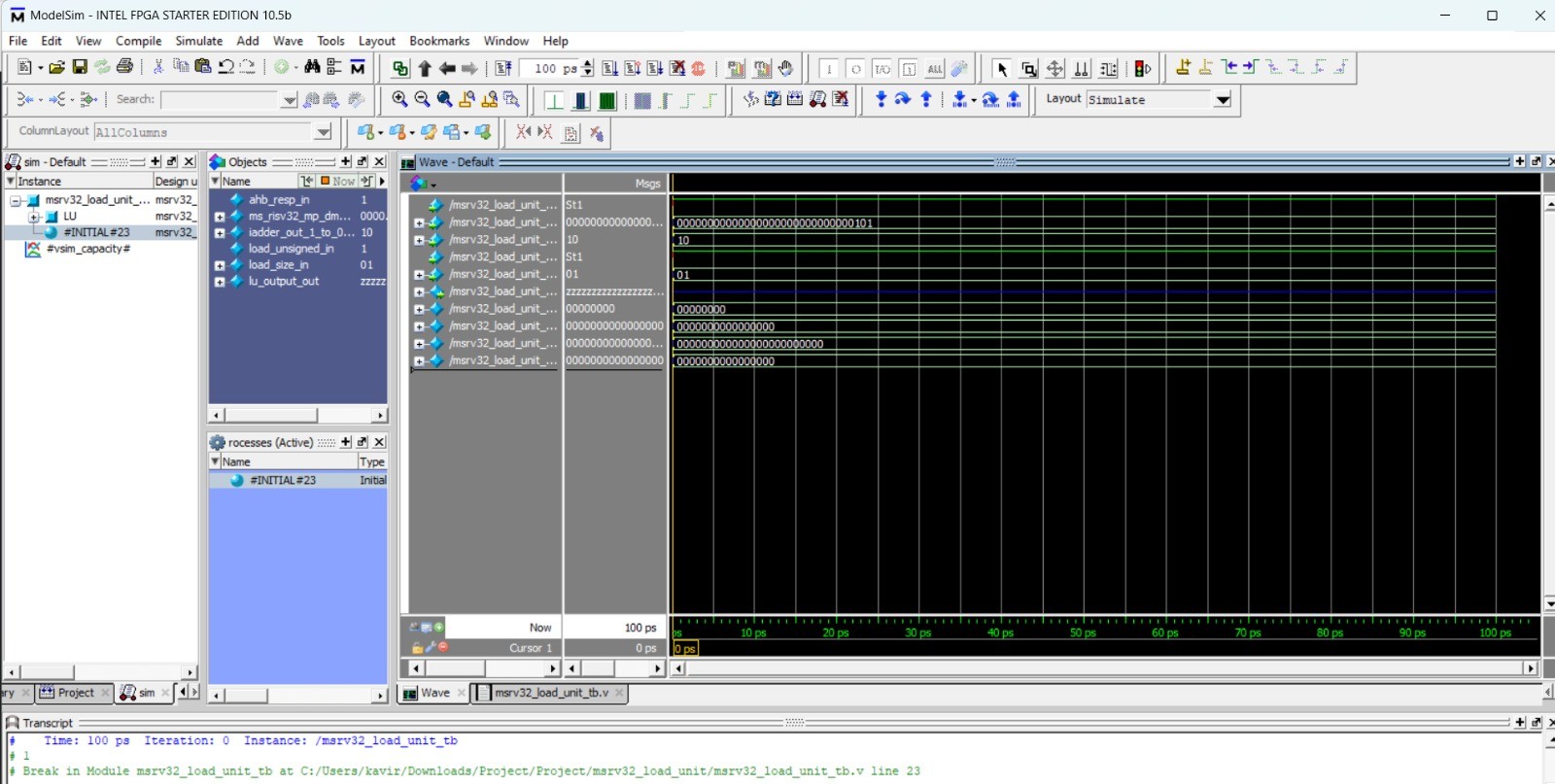
1. Depending on the funct3\_in signal wr\_mask\_out signal will get values.If funct3

\_in =2'b00, depending on the iaddr\_in [1:0], i.e. if iaddr\_in=2'b01 then wr\_mask\_ out= {2'b0, mem\_wr\_req\_in, 1'b0} If funct3\_in=2'b01, depending on the iaddr\_in

[1] i.e. if iaddr\_in [1] =1'b1 then wr\_mask\_out= {{2{mem\_wr\_req\_in}}, 2'b0} and for other combinations word of data ({4{mem\_wr\_req\_in}}) will be stored.

1. The dm\_adder\_out should be aligned with respect to {iaddr\_in [31:2], 2'b00}.
2. The wr\_req\_out should be aligned with respect to mem\_wr\_req\_in.
3. The ahb\_htrans\_out is 2'b10 during a valid store instruction and is 2'b00 when the transfer is completed.

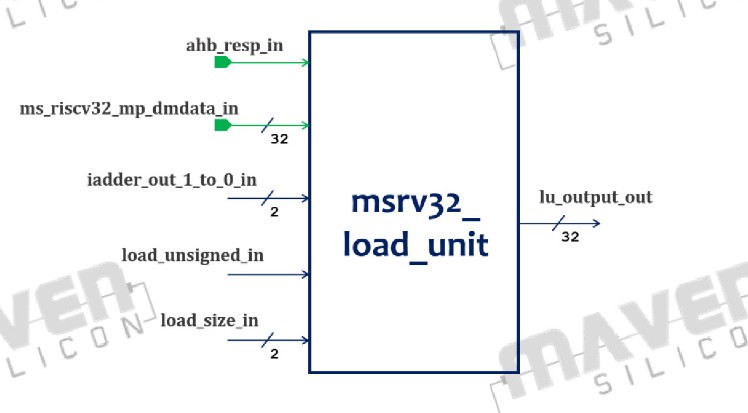
**Output:**



# LOAD UNIT:

**Input Ports:** load\_size\_in , load\_unsigned\_in, ahb\_resp\_in ms\_riscv32\_mp\_dm data\_in, iadder\_out\_1\_to\_0\_in

**Output Ports:** lu\_output\_out

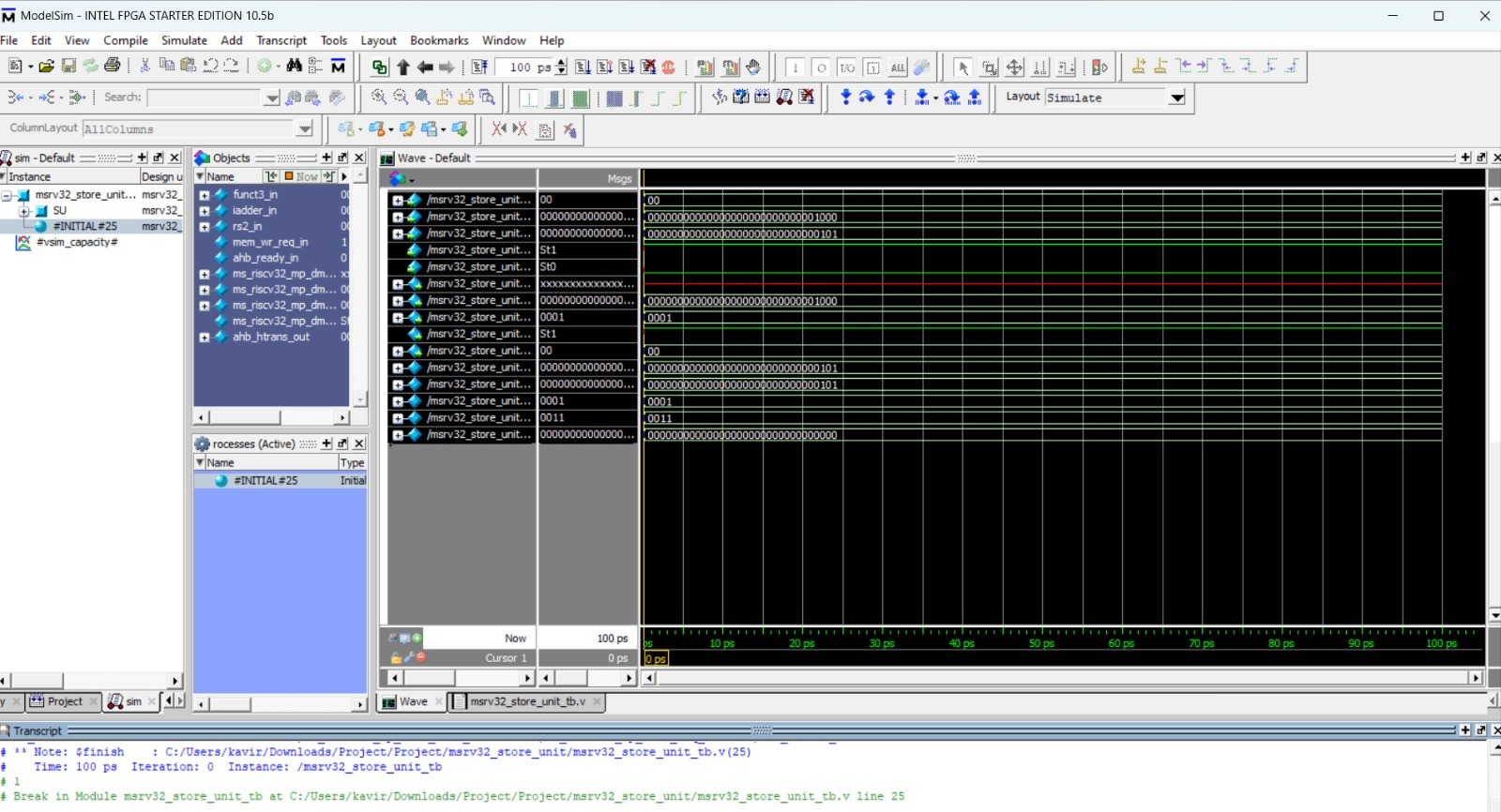


The portion of the processor responsible for transferring information from memory into registers is known as the load unit in the RISC-V ISA (Instruction Set Architecture). The load unit is in charge of carrying out load instructions, which retrieve data from the memory locations listed in the instruction.When a load instruction is executed in RISC-V, the load unit performs Memory Address Calculation, Memory Access, Data Transfer, and Handling Data Alignment.The load unit is necessary when transferring data from memory to the processor for processing or when storing it in registers. It enables RISC-V programmes to quickly access and manipulate data that is stored in memory.

**Functionality:**

The Load Unit reads the data\_in input signal and forms a 32-bit value based on the load instruction type (encoded in the funct3 field). The formed value (placed on output) can then be written in the Integer Register File. The module input and output signals are shown in the above table. The value of output is formed as shown in the above table.

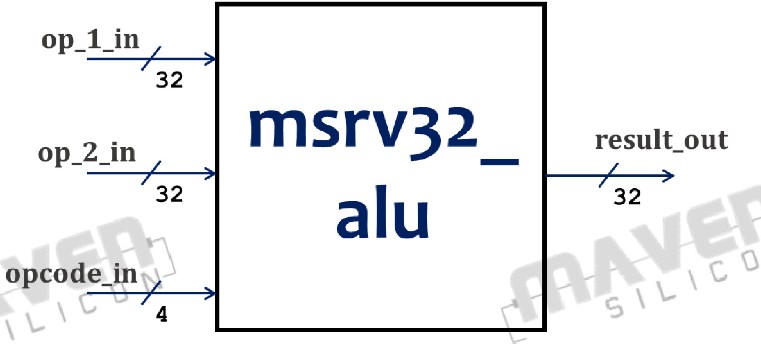
**Output**:



**ALU**:

**Input Ports:** op\_1\_in, op\_2\_in, opcode\_in

**Output Ports:**result\_out



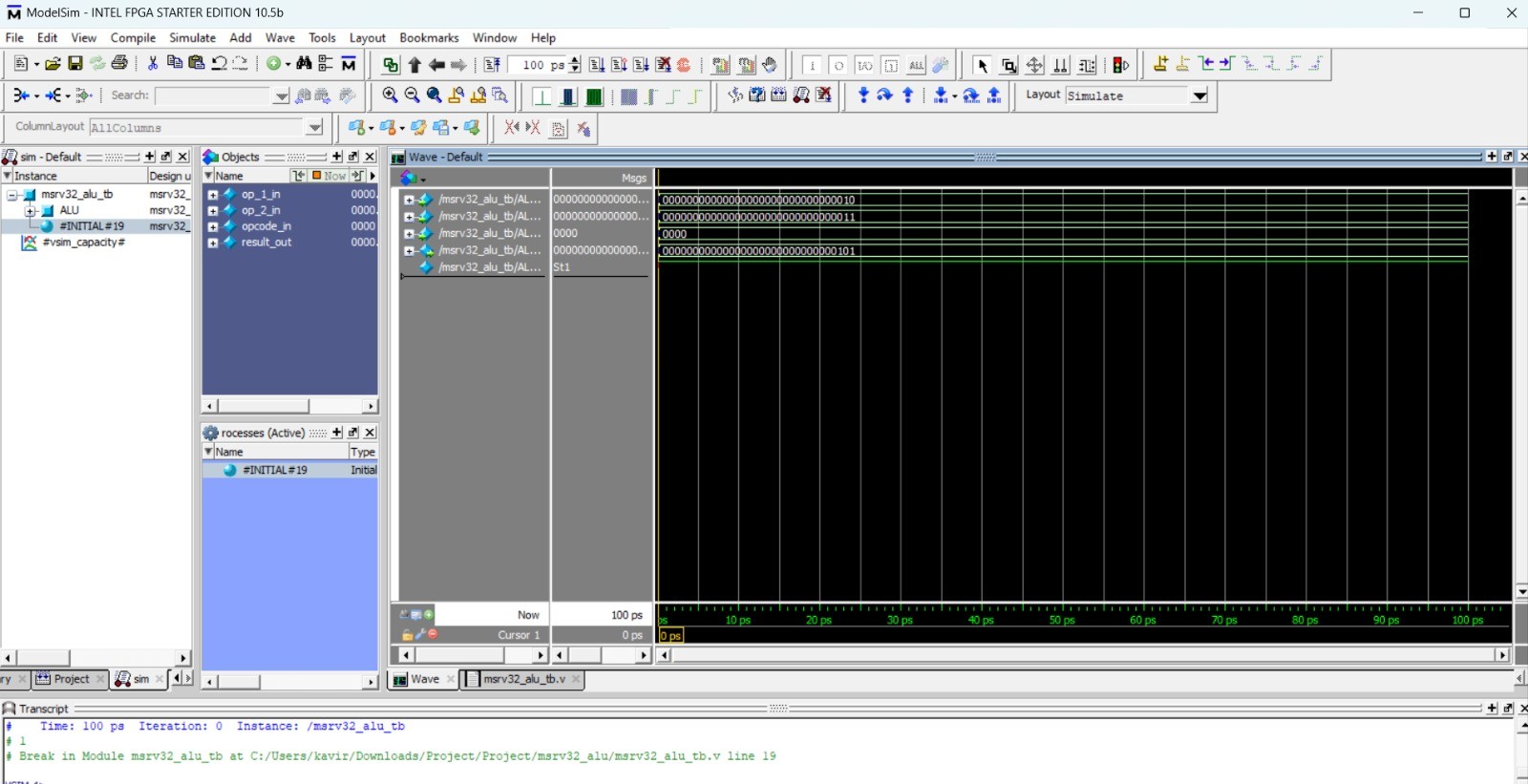
The ALU (Arithmetic Logic Unit) in the RISC-V ISA (Instruction Set Architecture) performs mathematical and logical operations on data. The processor's core function is to execute RISC-V programme instructions.Bit manipulation, shift operations, logical operations, and integer arithmetic are some of the operations supported by the RISC-V ALU.ALU capabilities and particular implementations can vary between different RISC-V implementations, including different extensions or customizations. All RISC-V processors share the same core functionality of the ALU, which provides the necessary computational capabilities for processing RISC-V instructions.

**Functionality:**

The ALU applies ten distinct logical and arithmetic operations in parallel to two 32-bit operands, outputting the result selected by opcode\_in. The ALU input/output signals and the opcodes are shown in tables below.The opcode values were

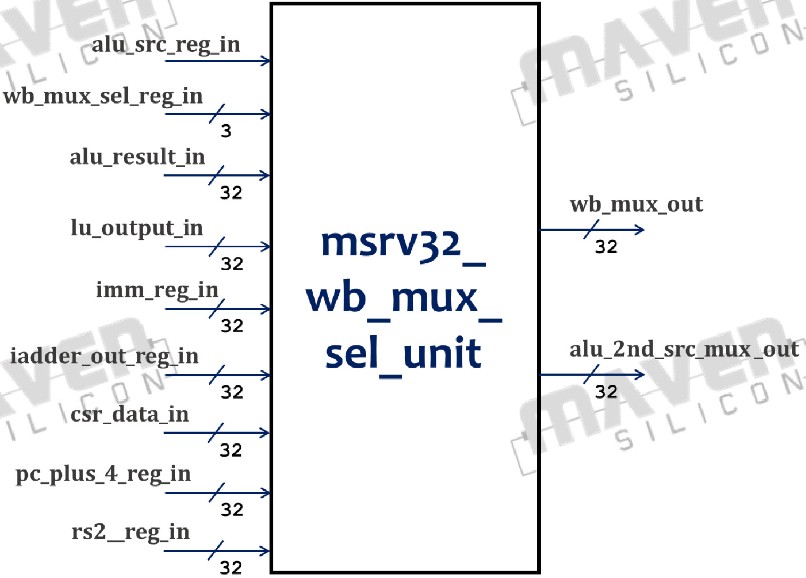
assigned to facilitate instruction decoding. The most significant bit of opcode\_in matches with the second most significant bit in the instruction funct7 field. The remaining three bits match with the instruction funct3 field.

**Output:**



# WB MUX SELECTION UNIT:

**Input Ports:**wb\_mux\_sel\_reg\_in, alu\_result\_in, lu\_output\_in, imm\_reg\_in, iadder\_out\_reg\_in, csr\_data\_in, pc\_plus\_4\_reg\_in , rs2\_reg\_in, alu\_src\_reg\_in **Output Ports:**wb\_mux\_out, alu\_2nd\_src\_mux\_out



The writeback mux selection unit in the RISC-V ISA (Instruction Set Architecture) determines the appropriate data to be written back to a register in the processor's register file. This unit typically functions as a pipeline component of the processor and takes part in the execution of instructions.During the execution of an instruction in RISC-V, the writeback mux selection unit completes the following tasks: Instruction Result Generation, Selecting the Writeback Source Register-to-Register Instructions, Memory Load Instructions, Immediate Instructions, and Register File Updating.The writeback mux selection unit ensures that the right data is written back to the register file based on the type and outcome of the instruction being executed. In order to guarantee accurate and consistent register values throughout the execution of a RISC-V programme, it is crucial.

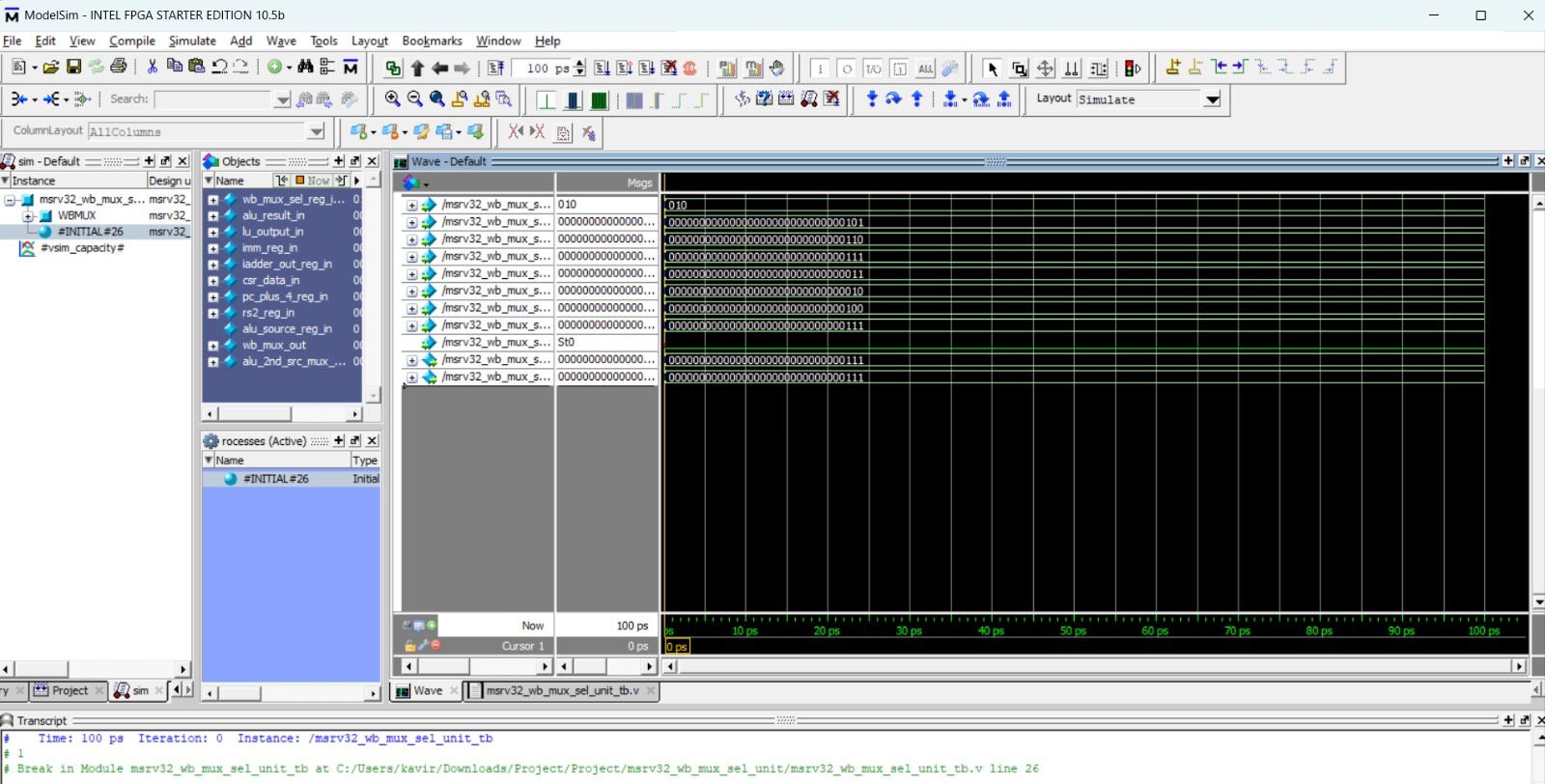
**Functionality:**

If 'alu\_src\_reg\_in' is high then 'alu\_2nd\_src\_mux\_out' = 'rs2\_reg\_in' Else 'alu\_2nd

\_src\_mux\_out' = 'imm\_reg\_in'.Based on the 'wb\_mux\_sel\_reg\_in' output signal ' wb\_mux\_out' will be assigned

* If 'wb\_mux\_sel\_reg\_in' = WB\_ALU then 'wb\_mux\_out' = 'alu\_result\_in'.
* If 'wb\_mux\_sel\_reg\_in' = WB\_LU then 'wb\_mux\_out' = 'lu\_output\_in'.
* If 'wb\_mux\_sel\_reg\_in' = WB\_IMM then 'wb\_mux\_out' = 'imm\_reg\_in'.
* If 'wb\_mux\_sel\_reg\_in'=WB\_IADDER\_OUT then 'wb\_mux\_out': = 'iadder\_out\_reg\_in'.
* If 'wb\_mux\_sel\_reg\_in' = WB\_CSR then 'wb\_mux\_out' = 'csr\_data\_in'.
* If 'wb\_mux\_sel\_reg\_in' = WB\_PC\_PLUS then 'wb\_mux\_out' = 'pc\_plus\_4\_reg\_in'. For remaining combination 'wb\_mux\_out' = 'alu\_result\_in'.
* wb\_mux\_sel\_reg\_in value, description and the instruction type is given below.

**Output:**

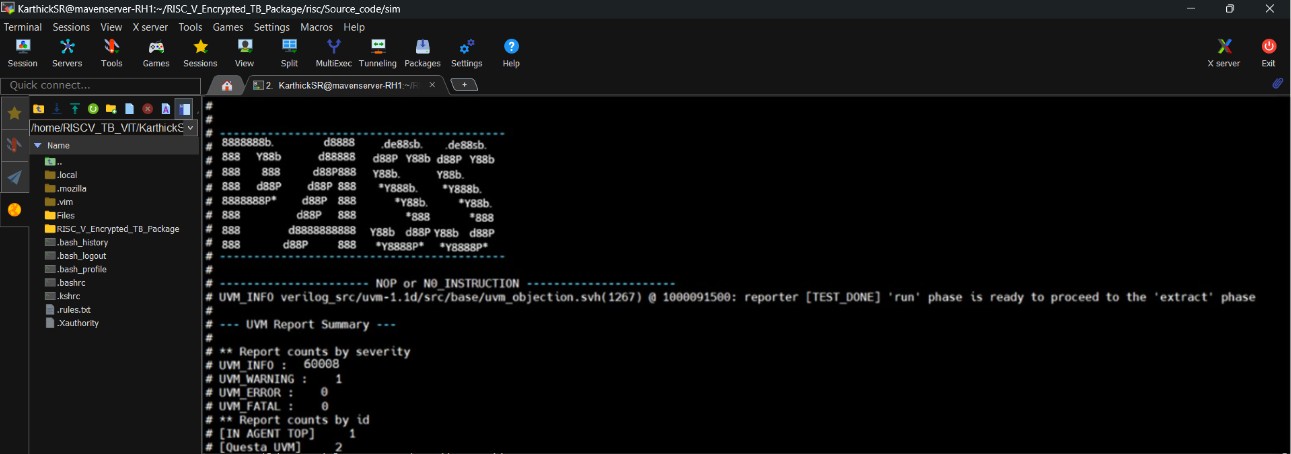


**TOP MODULE**

**TOP MODULE:**



**Outputs:**



**CHAPTER 4**

**CONCLUSION**

**4.1 CONCLUSION**

In conclusion, our internship project on RISC-V processor design has been a fascinating exploration of the field of VLSI technology and semiconductor design. We have successfully addressed the challenges of utilising the RV32I instruction set architecture to implement a three-stage pipeline processor through this project. Working on a variety of sub-modules has improved our knowledge of Verilog HDL and processor architecture, including the PC Mux, Register Blocks, Immediate Generator, and ALU, among many others. Furthermore, the significance of following architectural specifications and industry standards has highlighted the need for accuracy and precision in this field. We appreciate Maven Silicon Softech Ltd.'s guidance and assistance, as this internship has enhanced our technical knowledge and given us essential skills for our future employment in the semiconductor sector. This project has been a crucial step in our development as skilled semiconductor designers.

**REFERENCES**

* https://www.maven-silicon.com/