RISC – V INTERNSHIP

MAVEN SILICON

BATCH: B2B DI 09 – RISC-V

RTL Implementation and Verification of a 32-bit RISC-V Processor Core (MSRV32)

Project submitted by:

Ethan Samuel Devadatta

Faculty Name: Megha

RTL Implementation and Functional Verification of a 32-bit RISC-V Processor Core

Abstract

This report documents the RTL implementation and functional verification of the MSRV32 processor core — a 3-stage pipelined 32-bit RISC-V (RV32I) design built as part of the Maven Silicon RISC-V training programme. The central aim was to develop a thorough understanding of pipelined processor architecture by coding all sixteen hardware sub-modules in Verilog HDL, running testbench-driven simulations for each, and examining the resulting waveforms to confirm correct behaviour.

All sixteen modules — spanning the Fetch, Decode, and Execute/Write-back pipeline stages — were individually coded and simulated in ModelSim. Purpose-built Verilog testbenches were created for every module to exercise it across a range of input scenarios, and the waveform outputs were carefully reviewed to validate that each module behaved in line with the MSRV32 Core Design Specification. Once module-level simulation was complete, the full design was compiled and run through the Maven Silicon UVM verification environment via MobaXterm, successfully executing R-type, B-type, and J-type instruction sequences.

The work delivered genuine hands-on exposure to RTL coding, pipelined micro-architecture, simulation-based verification, and hardware-level debugging — all carried out using the same industry-standard toolchain (ModelSim and Questa/UVM) that practising engineers rely on.

1. Introduction

Before any processor reaches silicon or an FPGA, it is first captured in a Hardware Description Language (HDL). Working at the Register Transfer Level (RTL), designers describe exactly how data moves between registers and what operations are applied at each step — giving a precise, synthesisable picture of the hardware.

Of the various processor ISAs available today, RISC-V has grown into one of the most prominent open-source options. Its deliberately simple, modular structure makes it well suited to both university coursework and commercial chip development. Crucially, because the specification is freely available, designers can build fully custom processors without running into the licensing constraints that come with proprietary ISAs.

The design at the heart of this project is the MSRV32 core, a three-stage pipelined RV32I processor that Maven Silicon developed specifically for its VLSI and RISC-V training programme. The goal was to build real RTL experience by implementing every one of the sixteen modules that together form the MSRV32 micro-architecture. Each module went through individual testbench verification before being considered ready for integration, with ModelSim handling the simulation work and the Maven Silicon UVM server running the instruction-level test sequences.

2. Objectives

The work was guided by the following objectives:

- To build a solid understanding of how the MSRV32 3-stage pipelined 32-bit RV32I processor is structured and how it operates.
- To write Verilog HDL code for all sixteen functional modules that make up the processor.
- To verify the correct behaviour of each module by developing and running dedicated testbenches.
- To inspect and interpret the ModelSim waveform outputs in order to validate module behaviour.

- To run instruction-level test sequences through the Maven Silicon UVM environment and confirm correct execution.
- To gain experience with the debugging methods commonly applied during RTL development.
- To develop hands-on familiarity with pipelined digital hardware design and the verification workflow.

3. Overview of the MSRV32 RISC-V Processor

RISC-V is a Reduced Instruction Set Computer (RISC) architecture built around the principles of simplicity, modularity, and extensibility. Like all RISC designs, it follows a load-store model: arithmetic and logical operations work exclusively on register values, and data is only moved to or from memory through dedicated load and store instructions.

The MSRV32 core uses a three-stage pipeline, with each stage handling a distinct phase of instruction processing:

3.1 Fetch Stage (Stage 1)

Stage 1 is responsible for working out the address of the next instruction and retrieving it from instruction memory. The PC MUX (msrv32_pc_mux) selects combinationally between the boot address, the exception PC, the trap address, and the standard next-sequential or branch/jump target. Reg Block 1 (msrv32_reg_block_1) then latches the chosen PC value across the rising clock edge so it is available to the following stage.

3.2 Decode Stage (Stage 2)

Stage 2 breaks down the fetched instruction into its component fields and puts it to work. Register operands are read, sign-extended immediates are formed, memory addresses are computed, branch conditions are evaluated in combinational logic, and store data is sent directly to data memory. The modules active here include the Decoder, Instruction MUX, Immediate Generator, Immediate Adder, Branch Unit, Integer Register File, Write Enable Generator, and Store Unit.

3.3 Execute / Write-back Stage (Stage 3)

Reg Block 2 first captures the decoded control and data signals coming out of Stage 2. From there, the ALU carries out arithmetic and logical computation, the Load Unit extracts and sign- or zero-extends load results arriving from memory, the CSR File manages privileged control and status registers, and the WB MUX Selection Unit picks the final value to commit to the general-purpose register file. The Machine Control unit runs across all three stages, coordinating trap and exception handling.

4. Project Workflow

The project was carried out as a disciplined RTL design-and-verify cycle. Every one of the sixteen hardware modules was built from the ground up using the MSRV32 Core Design Specification as the reference, put through simulation-based verification, and only then treated as a candidate for integration into the full processor.

The steps followed were:

- Studying the MSRV32 Core Design Specification carefully to understand what each module is expected to do.
- Writing Verilog HDL code for all sixteen modules.
- Compiling each module in ModelSim to check for syntax and structural errors.
- Writing a targeted testbench for each module to drive verification at the block level.

- Executing simulations and studying the generated waveforms.
- Tracking down and fixing syntax mistakes, logical errors, and integration issues.
- Running the verified instruction sequences inside the Maven Silicon UVM environment.
- Confirming correct execution by reviewing the terminal output in MobaXterm.

Working module by module made debugging significantly more manageable and gave confidence that every component was individually correct before being brought together at the processor level.

5. Software and Tools Used

Tool	Purpose
Verilog HDL	RTL Design
ModelSim and QuartusPrime	Compilation and Simulation
MobaXterm	Remote Linux / SSH access to the verification server

6. Verification Methodology

Verification was structured around two complementary levels. At the module level, every one of the sixteen RTL blocks was driven with a range of input combinations through its own dedicated Verilog testbench, and ModelSim was used to assess whether the outputs matched expectations across different operating scenarios. At the instruction level, the assembled core was put through its paces in the Maven Silicon UVM environment, executing R-type, B-type, and J-type instruction sequences on the verification server.

The verification workflow involved:

- Writing a bespoke testbench for every RTL module.
- Compiling the module and its testbench together in ModelSim and QuartusPrime.
- Running the simulation.
- Reviewing the waveform output.
- Checking that the simulation outputs matched the results described in the design specification.
- Fixing any syntax or functional errors uncovered during simulation.
- Re-running the simulation until the module produced the correct, expected behaviour.
- Running UVM instruction sequences (R-type, B-type, and J-type) on the Maven Silicon verification server through MobaXterm.

Together, the simulation waveforms and the passing UVM test reports gave strong confidence that the implemented RTL was functionally correct.

7. Modules Implemented

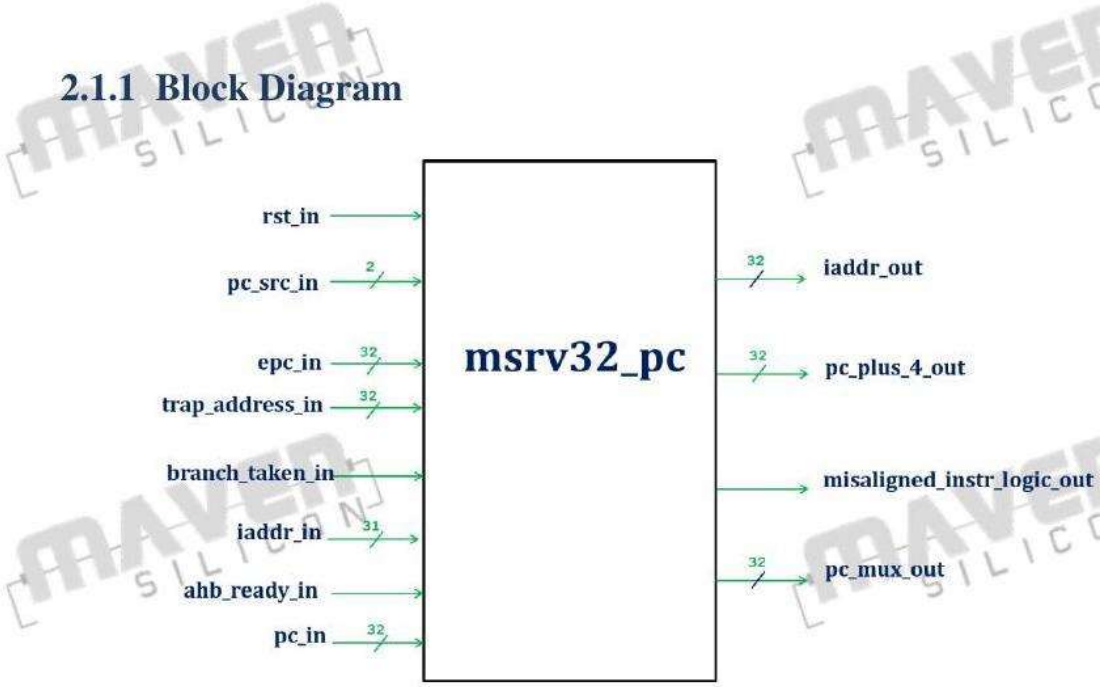
Building the MSRV32 processor meant implementing and verifying each of the sixteen RTL modules defined in the Core Design Specification, with every module serving a specific role within the three-stage pipeline. All modules were written in Verilog HDL and put through dedicated testbench verification on their own before being brought into the integrated design. Functional verification was performed in ModelSim by applying a variety of test cases and carefully examining the resulting waveforms.

The sixteen modules implemented over the course of this project are described below.

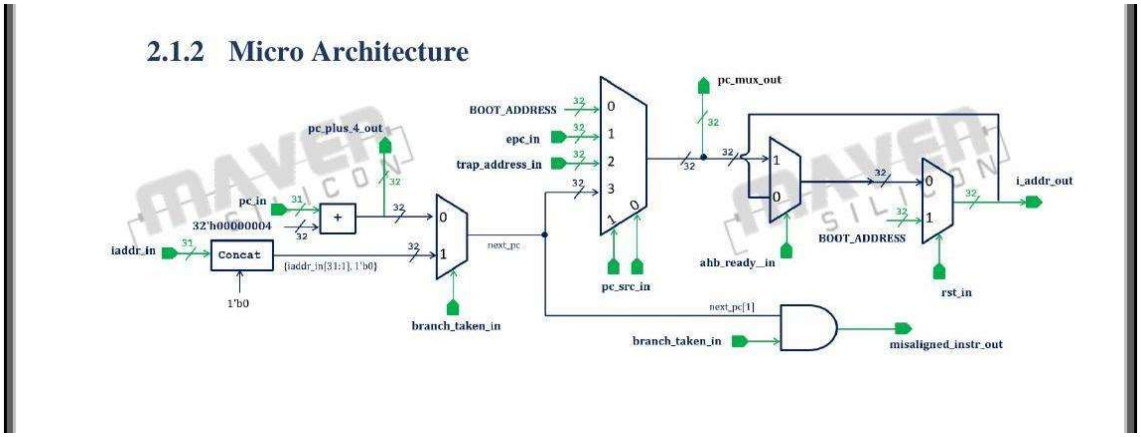
7.1 msrv32_pc_mux (Program Counter Multiplexer)

The PC MUX is a purely combinational block that determines which address the processor should fetch its next instruction from. Depending on the current processor state, it picks between the BOOT_ADDRESS (used during reset), the exception program counter epc (when returning from a trap via mret), the trap_address (when a trap is being taken), or the next_pc — which is either pc+4 for a sequential instruction or the computed branch/jump destination. When a branch or jump target falls on a non-4-byte-aligned boundary, the module also asserts the misaligned_instr_out flag.

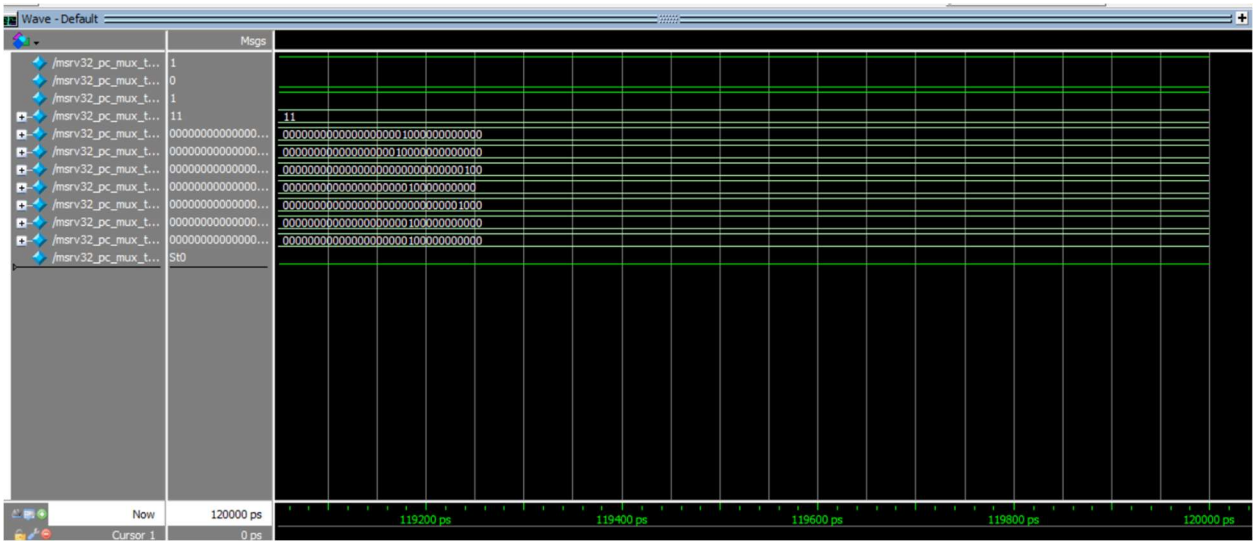
Block Diagram



Micro Architecture



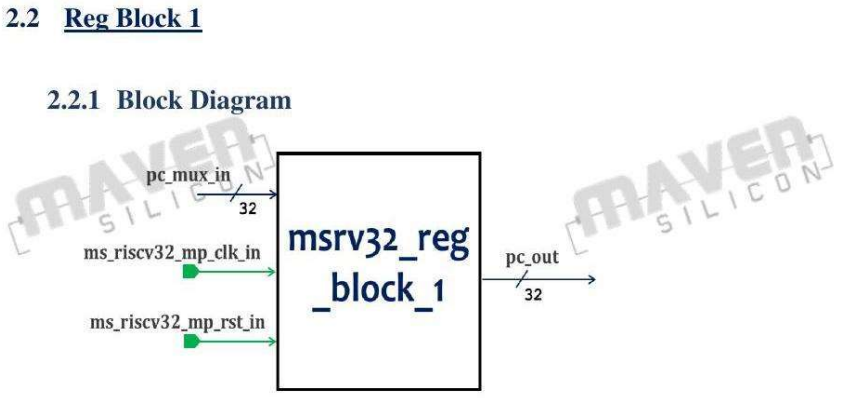
ModelSim Waveform



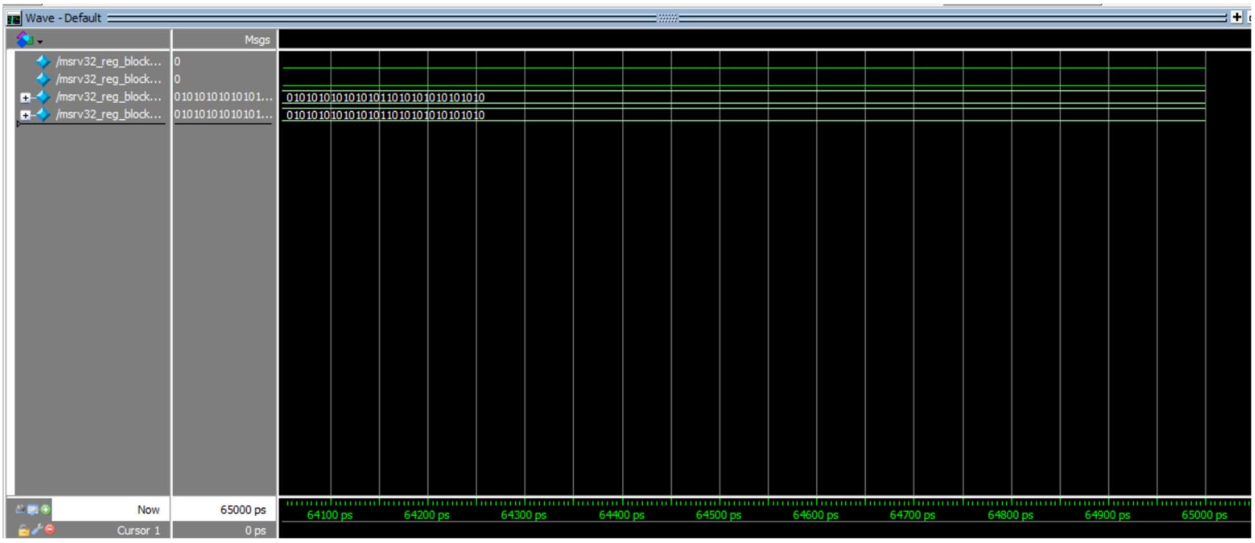
7.2 msrv32_reg_block_1 (Program Counter Register)

Reg Block 1 is the pipeline register that holds the current Program Counter value between stages. On every rising clock edge (as long as no reset is active), it captures the pc_mux_in value presented by the PC MUX and drives it as the stable PC output for the rest of the Fetch stage.

Block Diagram



ModelSim Waveform



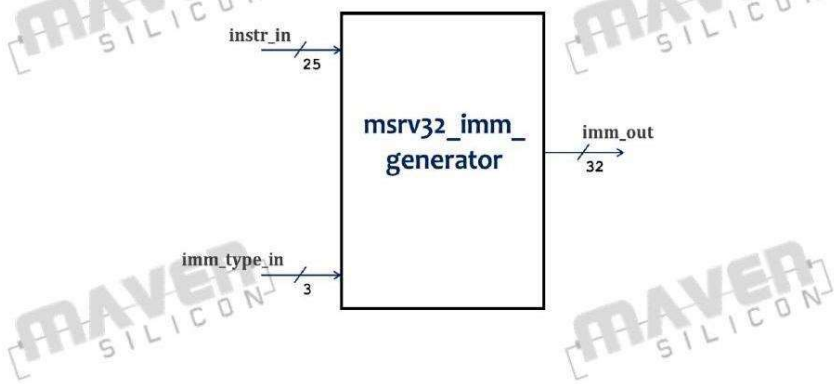
7.3 msrv32_imm_generator (Immediate Generator)

The Immediate Generator takes the raw bit fields embedded in the instruction word and reconstructs them into a proper 32-bit sign-extended immediate value. Because different instruction formats (I, S, B, U, J, and CSR) scatter their immediate bits across different bit positions, the module relies on the imm_type_in control signal from the Decoder to know exactly how to reassemble and sign-extend them.

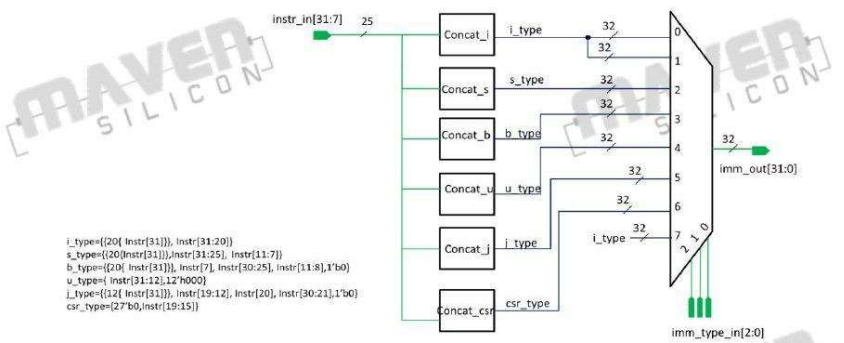
Block Diagram

2.3 Immediate generator

2.3.1 Block Diagram



Micro Architecture

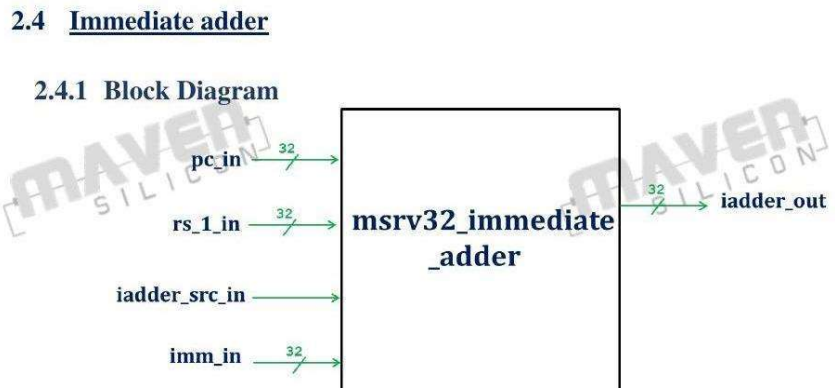


ModelSim Waveform

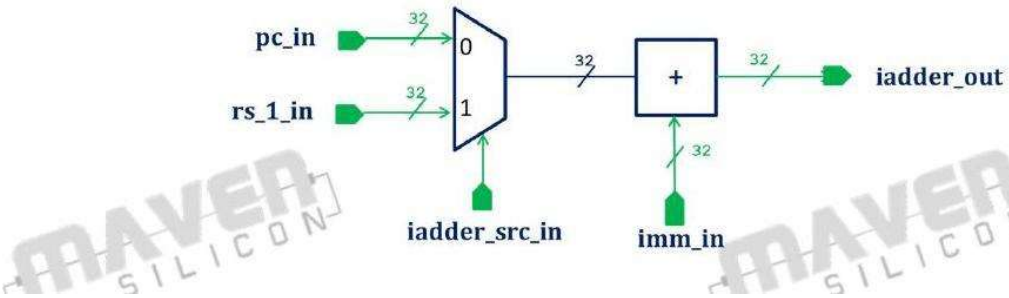
7.4 msrv32_immediate_adder

The Immediate Adder performs the address arithmetic needed by load, store, and jump instructions in a single combinational stage. Depending on the instruction type, it either adds the first source register value to the immediate (rs1 + imm) for load, store, and JALR operations, or adds the current program counter to the immediate (pc + imm) for branch and JAL instructions. The iadder_src_in control signal from the Decoder makes this selection.

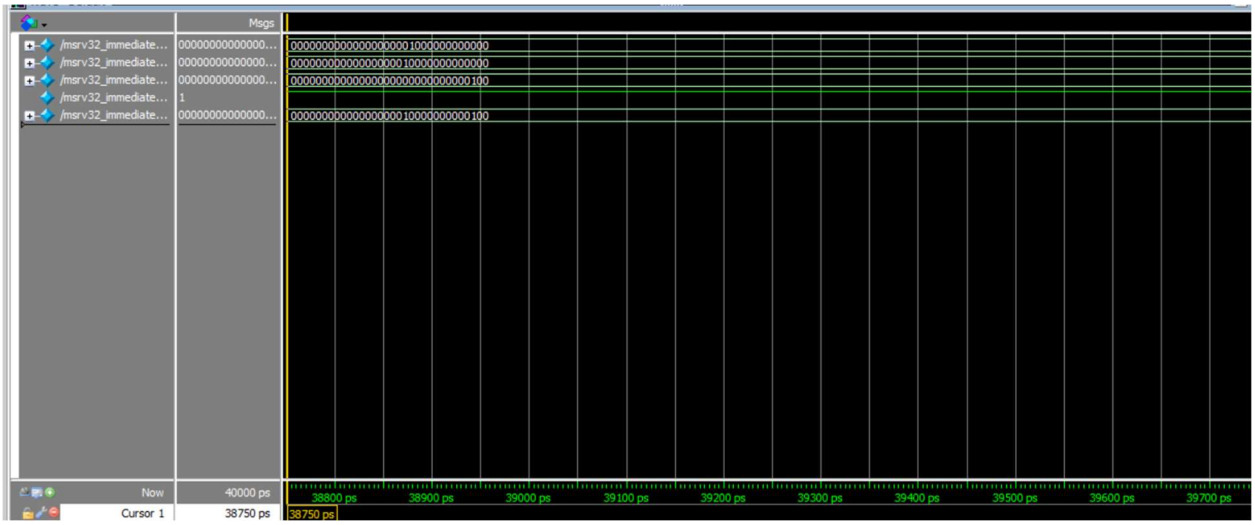
Block Diagram



Micro Architecture



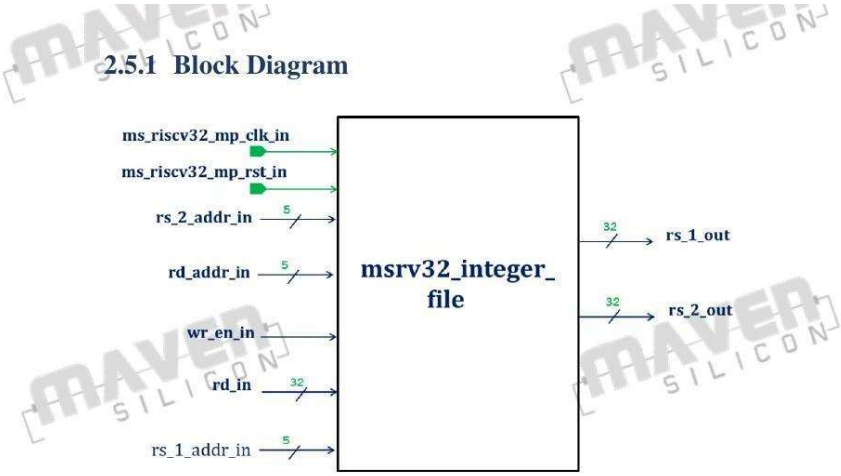
ModelSim Waveform



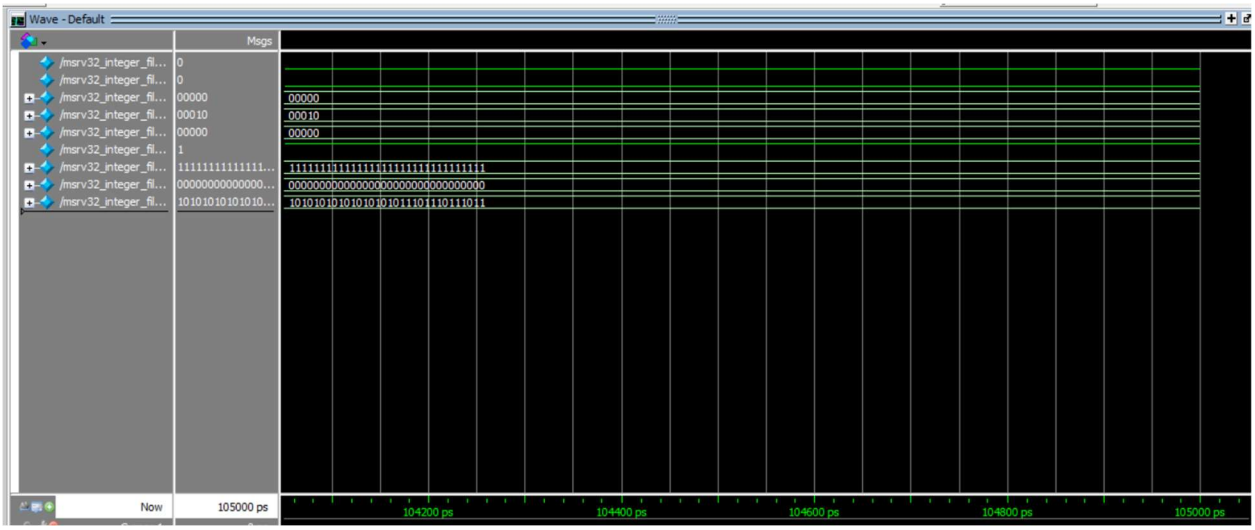
7.5 msrv32_integer_file (General-Purpose Register File)

The Integer Register File provides the processor's 32 general-purpose integer registers, with two simultaneous read ports and a single write port. To prevent data hazards without stalling the pipeline, the module includes internal forwarding logic: if a register being read in the Decode stage is the same one currently being written back in Stage 3, the incoming write-back value (rd_in) is forwarded directly to the rs_1_out or rs_2_out output. The forwarding logic is unconditionally disabled for register x0, ensuring it always reads as zero as the RISC-V specification requires.

Block Diagram



ModelSim Waveform



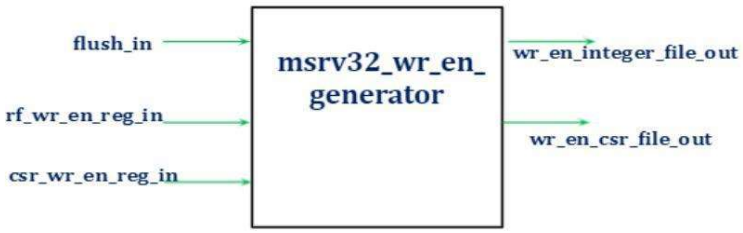
7.6 msrv32_wr_en_generator (Write Enable Generator)

The Write Enable Generator acts as a safety gate between the Decode stage and the register and CSR files. During a pipeline flush — when in-flight instructions are replaced with NOPs — this module drives the write-enable signals for both the Integer Register File and the CSR File to zero, ensuring that none of those cancelled instructions accidentally modify any architectural state.

Block Diagram

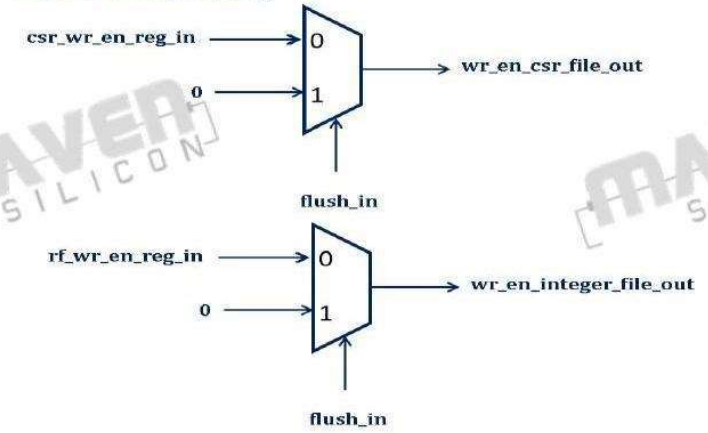
2.6 Write Enable Generator

2.6.1 Block Diagram

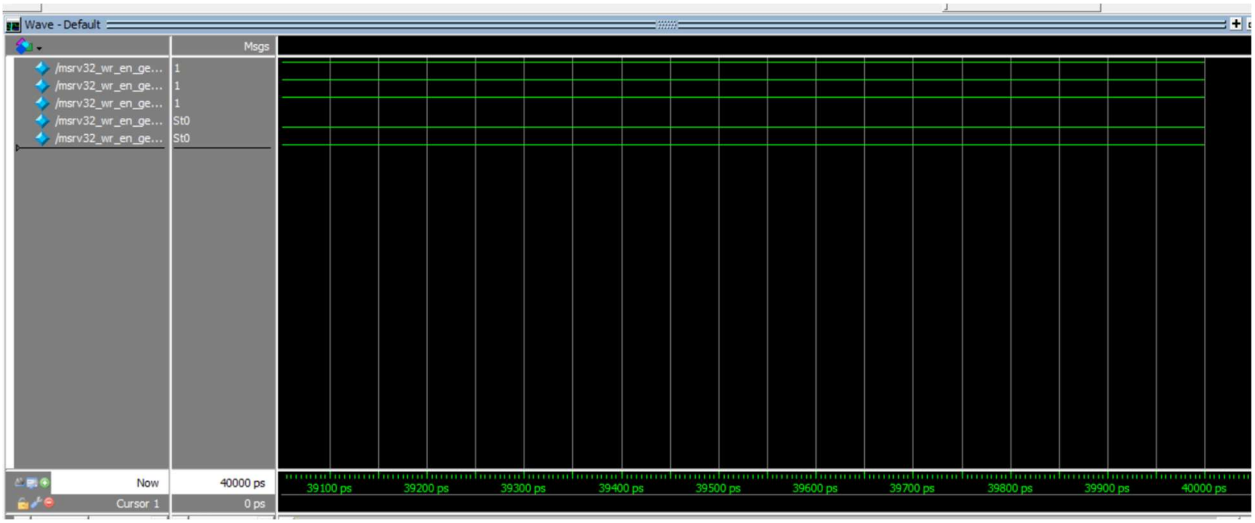


Functionality

2.6.3 Functionality



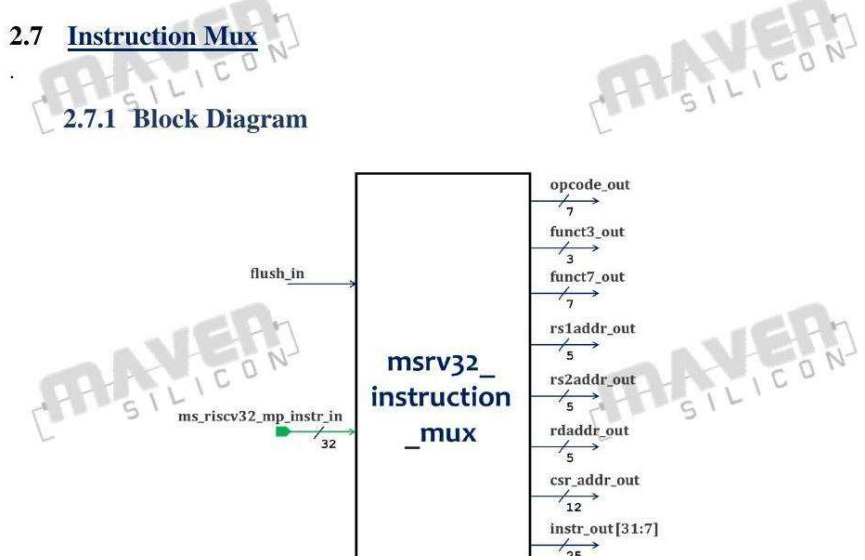
ModelSim Waveform



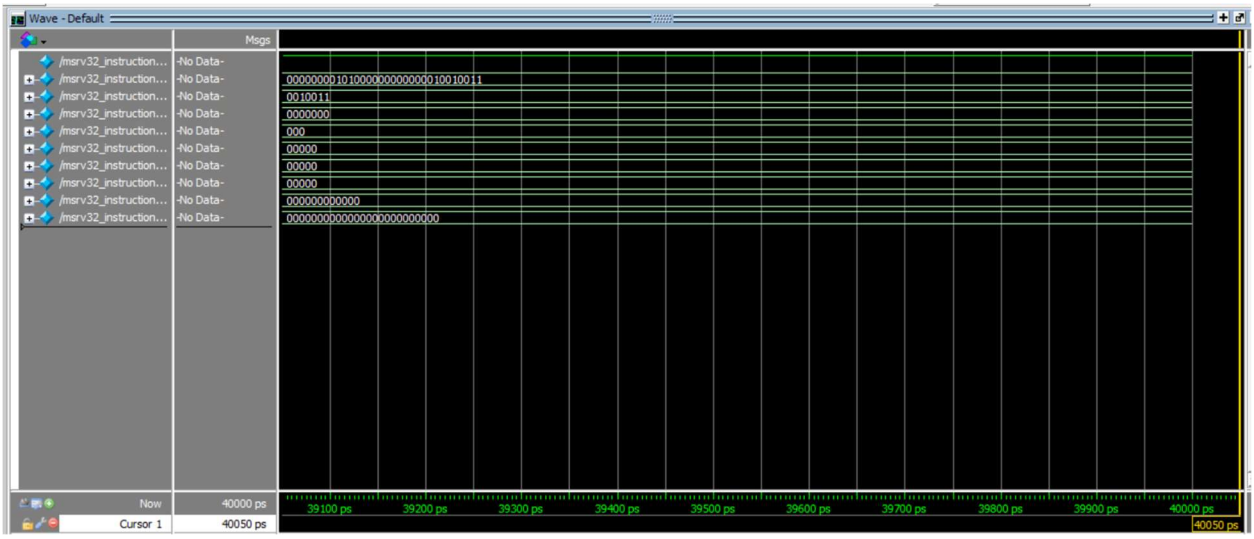
7.7 msrv32_instruction_mux (Instruction Parser & Flusher)

The Instruction MUX handles two jobs at once. First, it breaks apart the incoming 32-bit instruction word and routes each field — opcode, funct3, funct7, register addresses (rs1, rs2, rd), CSR address, and the immediate source bits — to the appropriate downstream modules. Second, whenever a pipeline flush is signalled, it replaces the instruction with the NOP encoding 32'h00000013, effectively emptying the Decode stage without requiring a stall.

Block Diagram



ModelSim Waveform

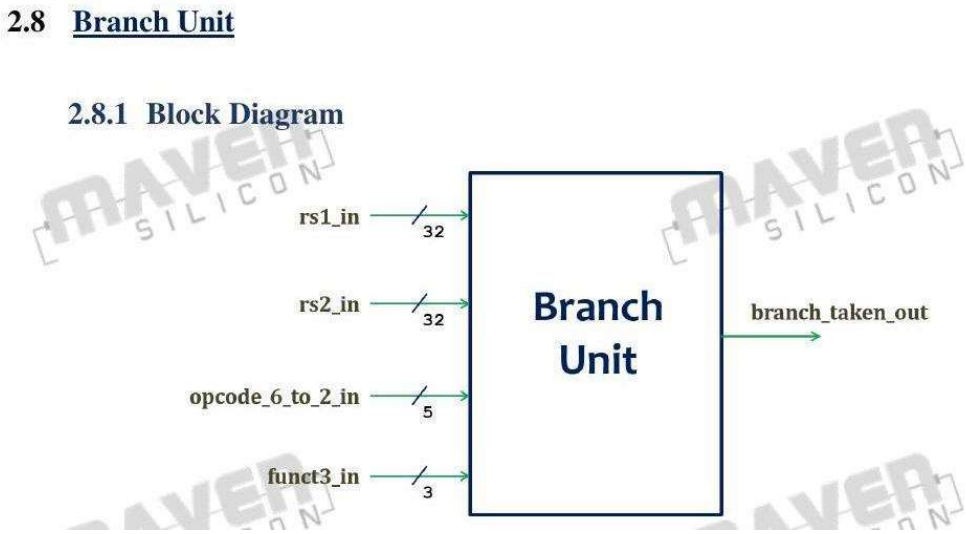


7.8 msrv32_branch_unit (Branch Comparator)

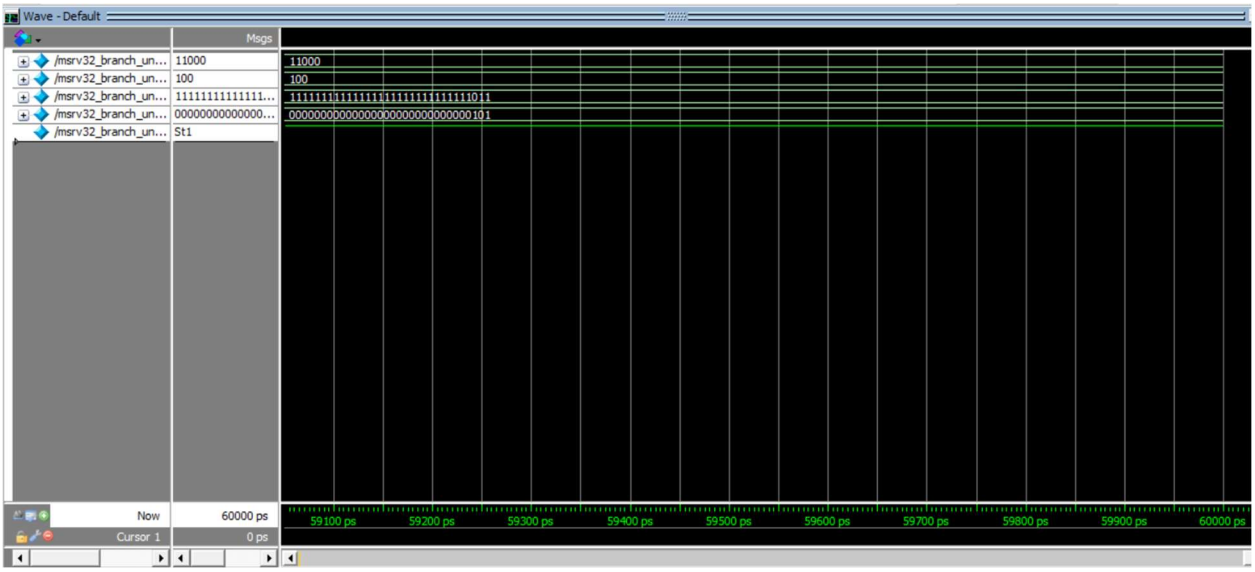
The Branch Unit is responsible for deciding whether a conditional branch should be taken. Working entirely in combinational logic, it takes the two source register values from the Integer Register File along with the opcode and funct3 fields and evaluates the branch condition for BEQ, BNE, BLT, BGE, BLTU, and BGEU instructions. Internally, only two fundamental comparisons are computed; the remaining four branch outcomes are derived

from those results. The unit also asserts branch_taken_out for unconditional jumps, so the PC MUX can redirect the program counter immediately.

Block Diagram



ModelSim Waveform



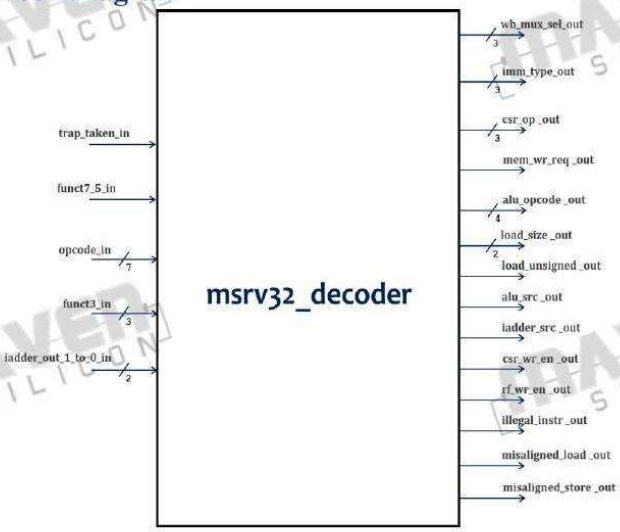
7.9 msrv32_decoder (Central Controller)

The Decoder sits at the centre of the pipeline and acts as the main control signal generator. It reads the opcode, funct3, and funct7 fields from the incoming instruction and produces the full set of control signals consumed by the ALU, the Load and Store Units, both register files, the Immediate Generator, and the Write-back Multiplexer. It also flags illegal instruction encodings and misaligned memory accesses, passing those error signals on to the Machine Control unit for appropriate exception handling.

Block Diagram

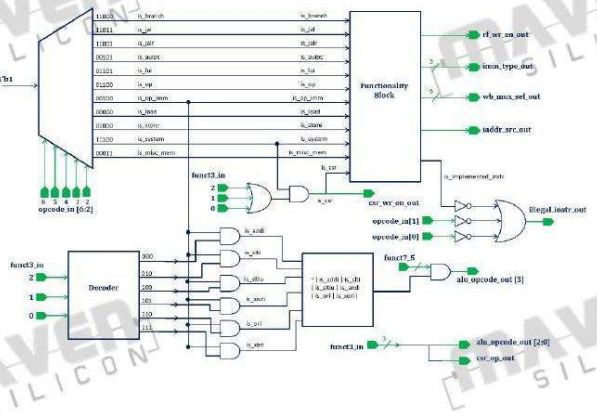
2.9 Decoder

2.9.1 Block Diagram

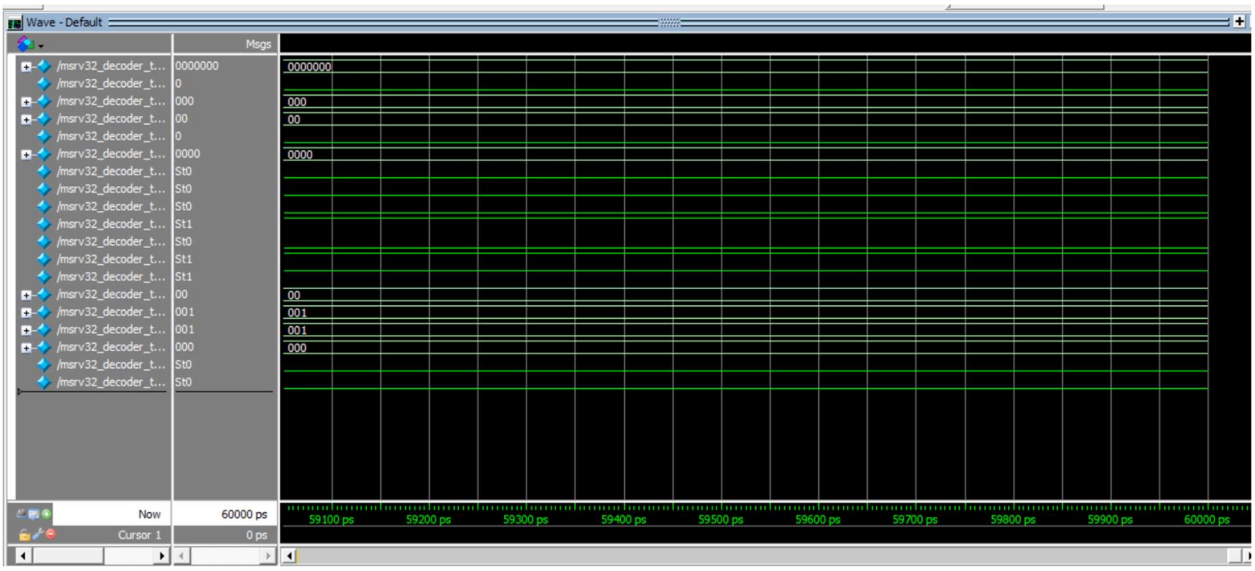


Micro Architecture

2.9.2 Micro Architecture



ModelSim Waveform



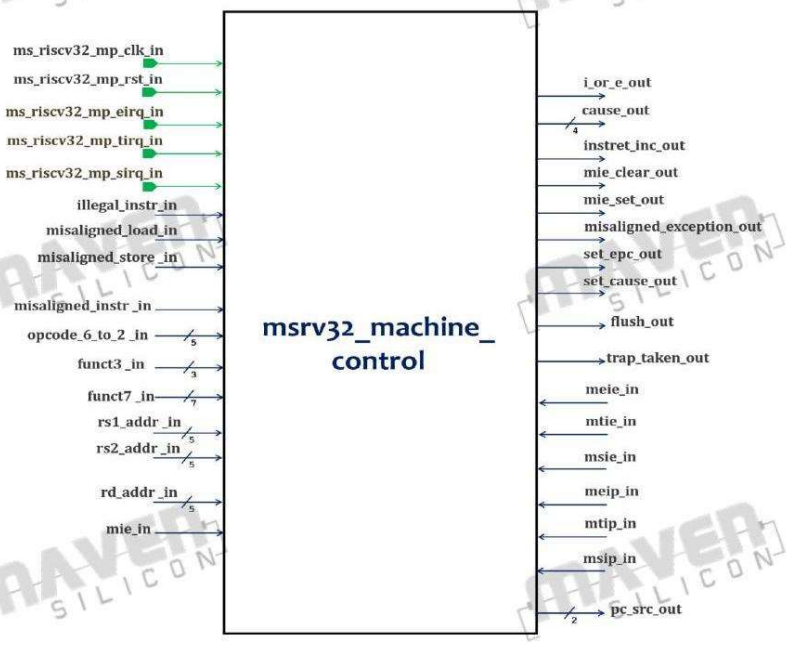
7.10 msrv32_machine_control (Trap & Exception Controller)

The Machine Control unit is the exception and interrupt handler of the processor, built around a four-state finite state machine with states RESET, OPERATING, TRAP_TAKEN, and TRAP_RETURN. When a trap occurs, the unit flushes any in-flight instructions from the pipeline, saves the return address into mepc, writes the appropriate exception or interrupt code into mcause, and steers the PC to the trap handler vector. When an mret instruction is executed, the machine transitions to TRAP_RETURN and resumes execution from the address stored in epc.

Block Diagram

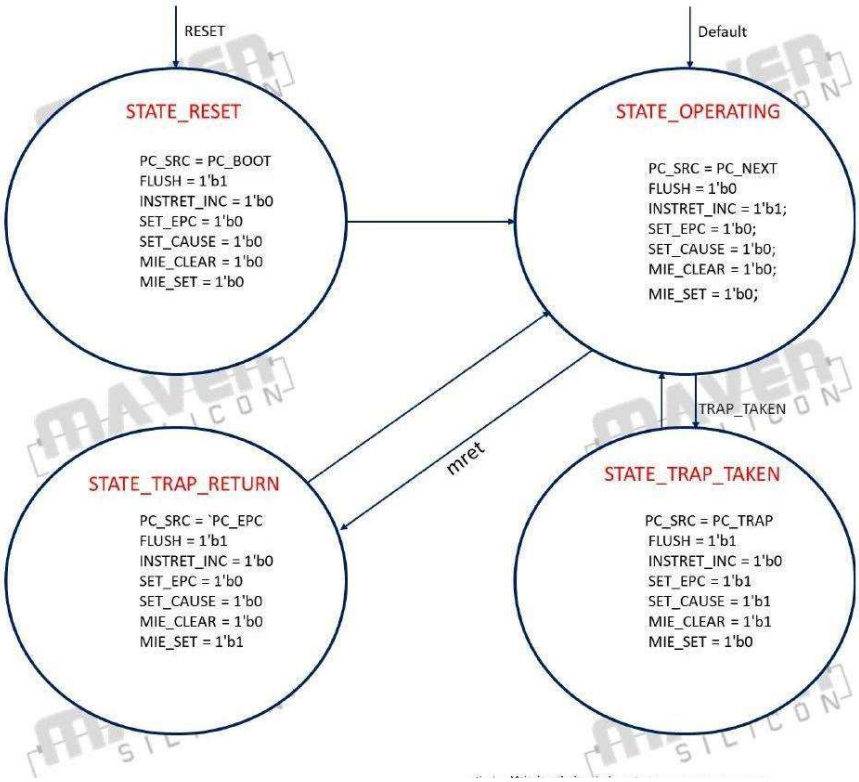
2.10 Machine Control

2.10.1 Block Diagram



State Diagram

2.10.2 State Diagram & Micro Architecture

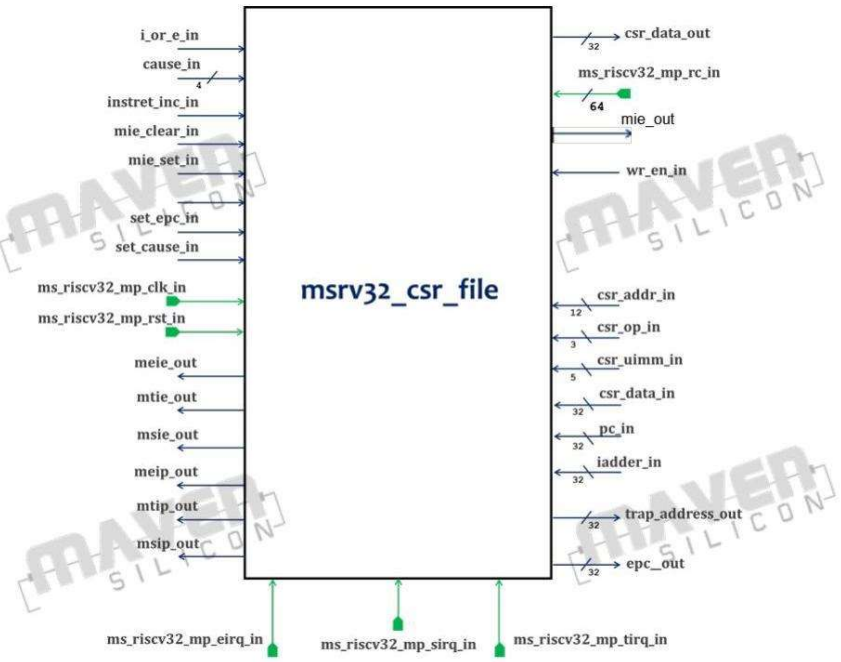


7.11 msrv32_csr_file (Control & Status Register File)

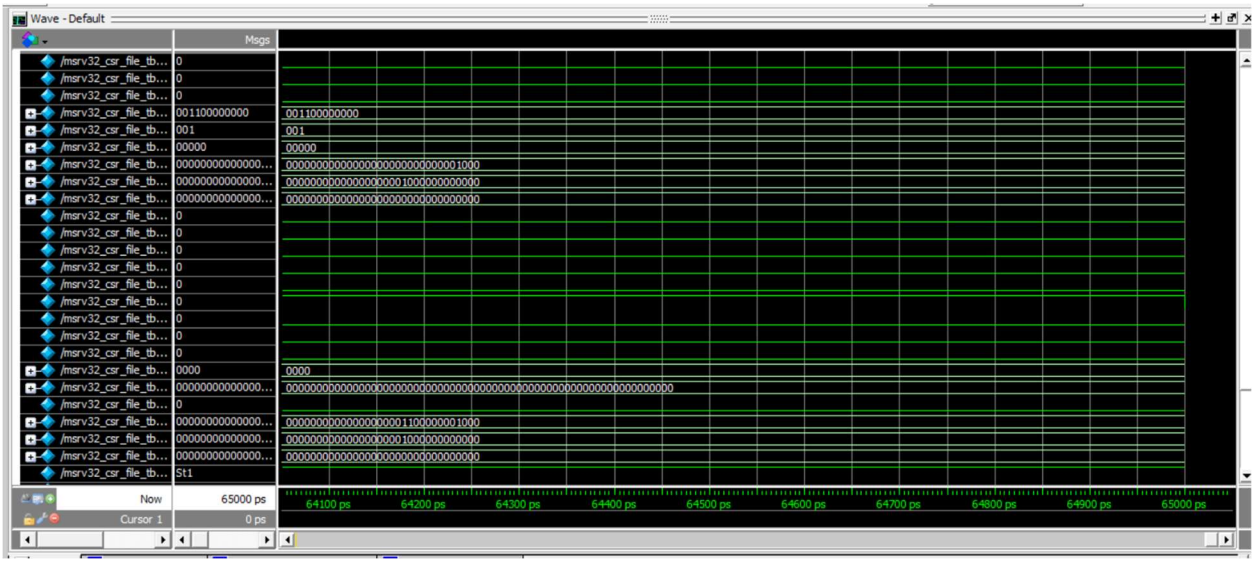
The CSR File is the repository for all privileged registers used in M-mode operation, including mstatus, mie, mip, mepc, mcause, and mtvec. It supports three types of register access: read/write, atomic bit-set, and atomic bit-clear. The file is also where the interrupt enable flags (MEIE, MTIE, MSIE) and interrupt pending bits (MEIP, MTIP, MSIP) live. On every cycle, it feeds the current trap vector address and exception PC back to the PC MUX so that the processor can redirect itself correctly when a trap occurs or when returning from one.

Block Diagram

2.11.1 Block Diagram



ModelSim Waveform



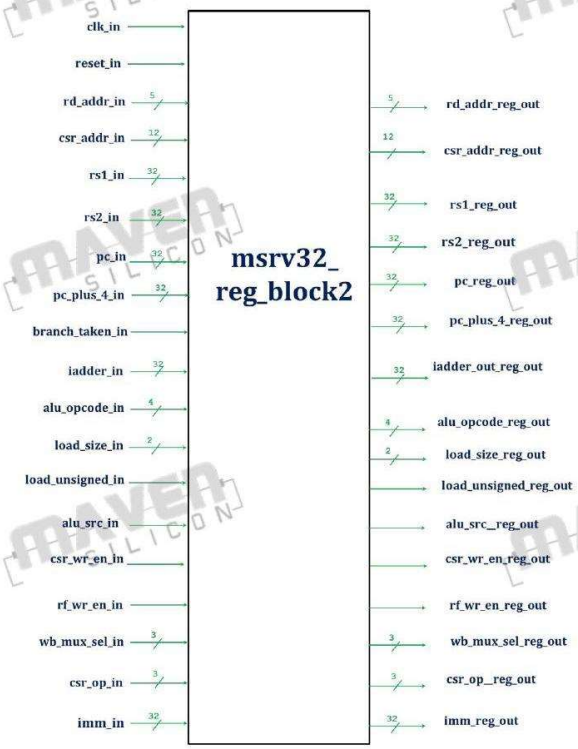
7.12 msrv32_reg_block_2 (Stage 2 → Stage 3 Pipeline Register)

Reg Block 2 is the pipeline register sitting on the boundary between Stage 2 and Stage 3. It captures every control and data signal produced by the Decode stage and holds them stable until the Execute/Write-back stage consumes them on the next clock edge (provided no reset is active). The block also contains a small 2:1 MUX that, when branch_taken_in is asserted, forces the least significant bit of the computed jump target to zero — a requirement imposed by the RISC-V JALR alignment rule.

Block Diagram

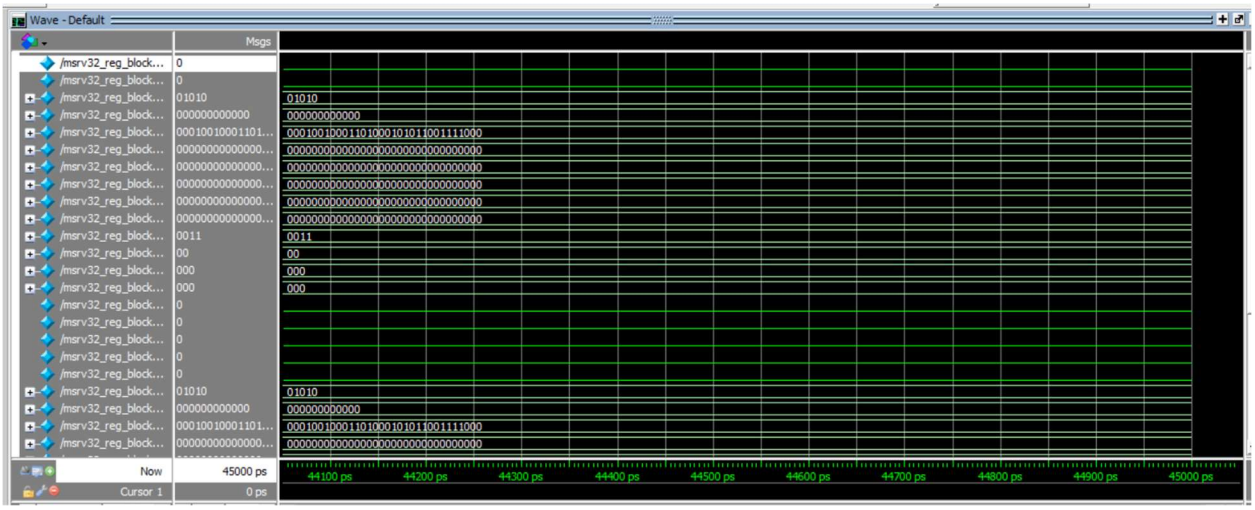
2.12 Reg Block 2

2.12.1 Block Diagram



2.12.2 Functionality

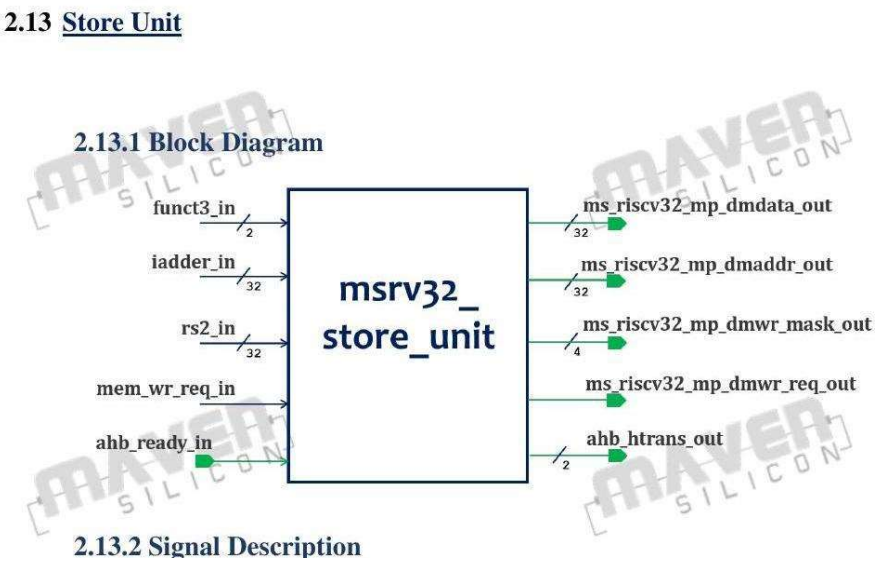
ModelSim Waveform



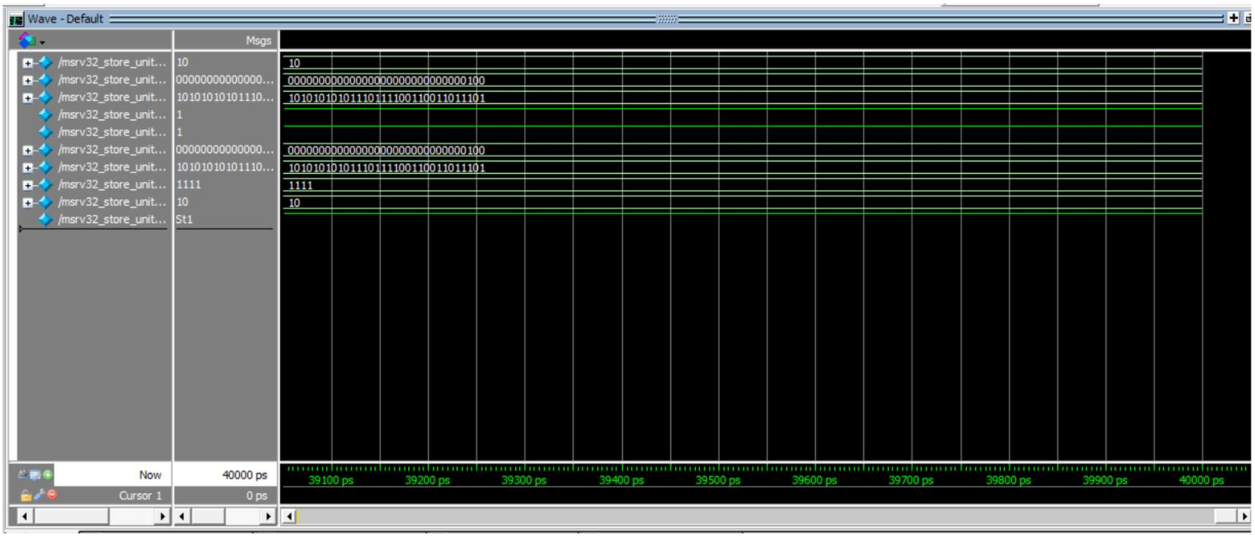
7.13 msrv32_store_unit (Store Aligner)

The Store Unit takes care of formatting outgoing write data before it reaches data memory. For byte stores (sb), half-word stores (sh), and full-word stores (sw), the unit shifts the write data into the correct position on the 32-bit data bus and produces the matching 4-bit byte-enable mask, which tells the memory which specific byte lanes should actually be written.

Block Diagram



ModelSim Waveform



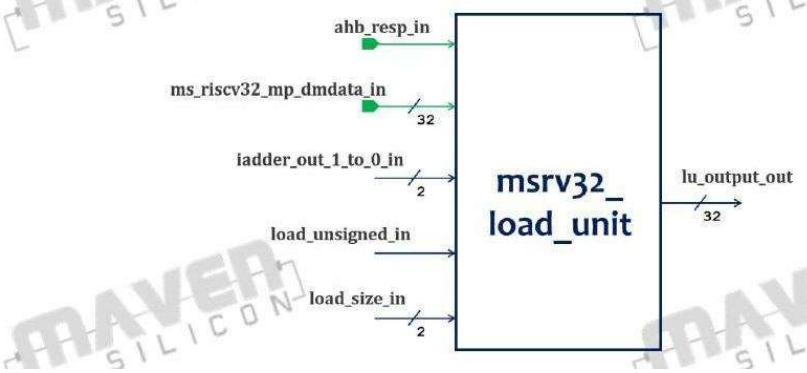
7.14 msrv32_load_unit (Load Aligner & Sign Extender)

The Load Unit does the inverse of the Store Unit, processing data coming back from memory. Given the full 32-bit word returned by the memory system, the unit uses the byte offset from the address to isolate the correct byte or half-word. The extracted value is then either sign-extended (for lb and lh) or zero-extended (for lbu and lhu) to produce a full 32-bit result, which is passed on to the Write-back Multiplexer.

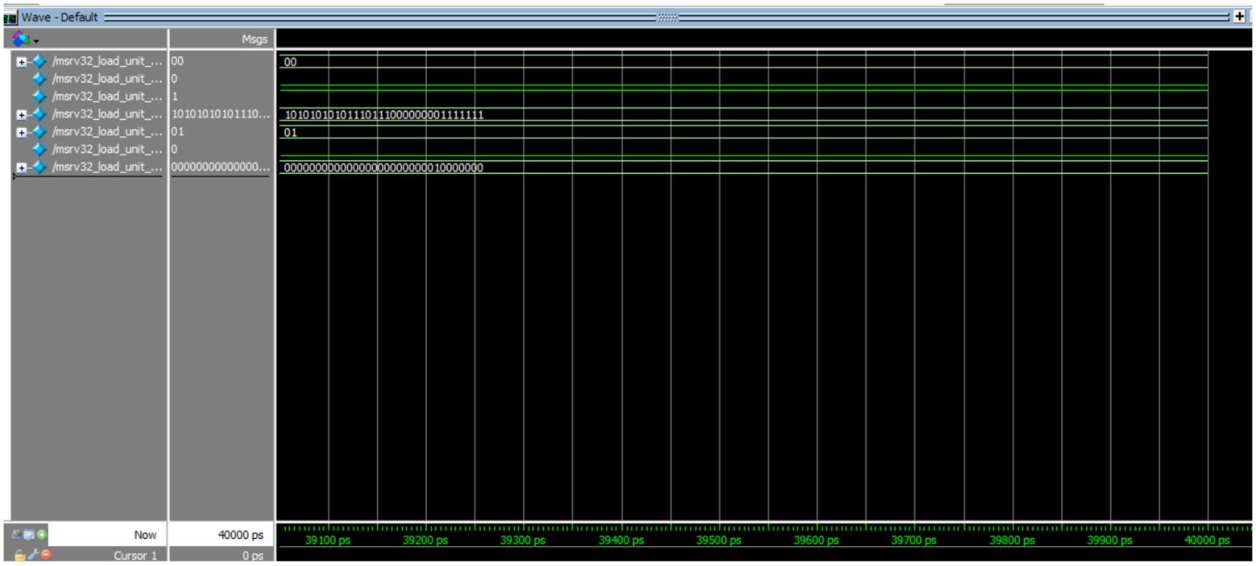
Block Diagram

2.14 Load Unit

2.14.1 Block Diagram



ModelSim Waveform



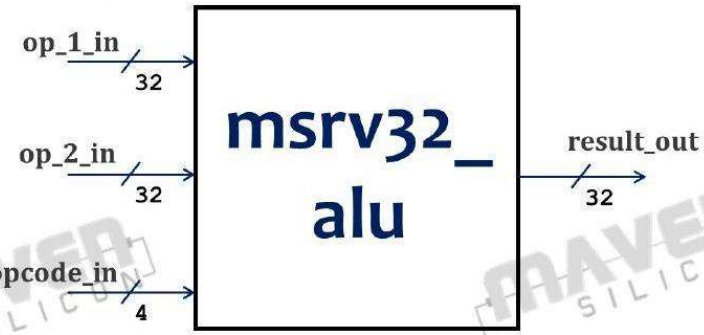
7.15 msrv32_alu (Arithmetic Logic Unit)

The ALU is the processor's main computation engine, responsible for every arithmetic and logical operation: ADD, SUB, AND, OR, XOR, left shift (SLL), logical right shift (SRL), arithmetic right shift (SRA), signed less-than comparison (SLT), and unsigned less-than comparison (SLTU). The specific operation to perform is encoded as a 4-bit control value derived from the instruction's funct7 and funct3 fields, making the ALU the functional heart of the Execute stage.

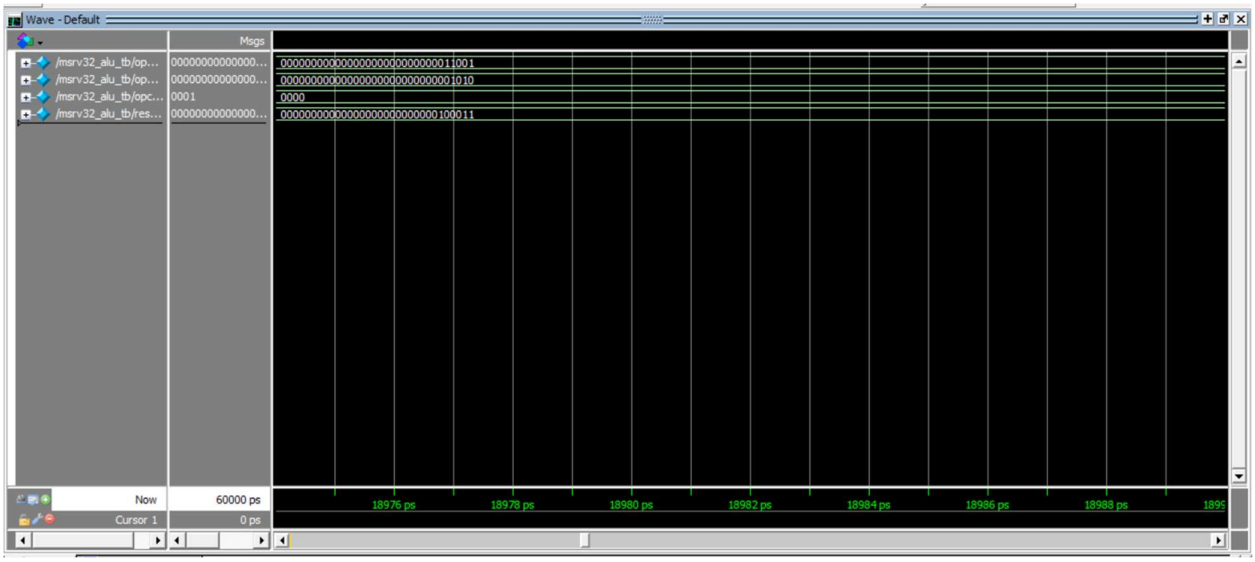
Block Diagram

2.15 ALU

2.15.1 Block Diagram



ModelSim Waveform



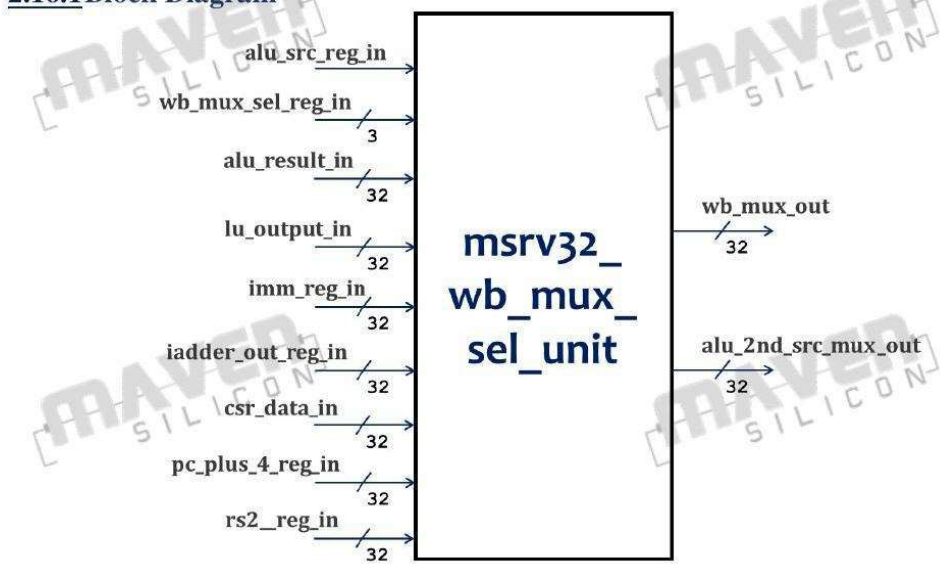
7.16 msrv32_wb_mux_sel_unit (Write-back Selector)

The WB MUX Selection Unit makes the final routing decision at the end of the pipeline: it picks which value will be committed to the destination register in the Integer Register File. The choices are the ALU result, the Load Unit's formatted output, the sign-extended immediate, the address computed by the Immediate Adder (either PC+imm or rs1+imm), the data read from the CSR File, or PC+4 (for link-register-style jumps). The selection is controlled by the wb_mux_sel_reg_in signal. The unit also contains a secondary MUX that steers the ALU's second input between the rs2 register value and the immediate, producing alu_2nd_src_mux_out.

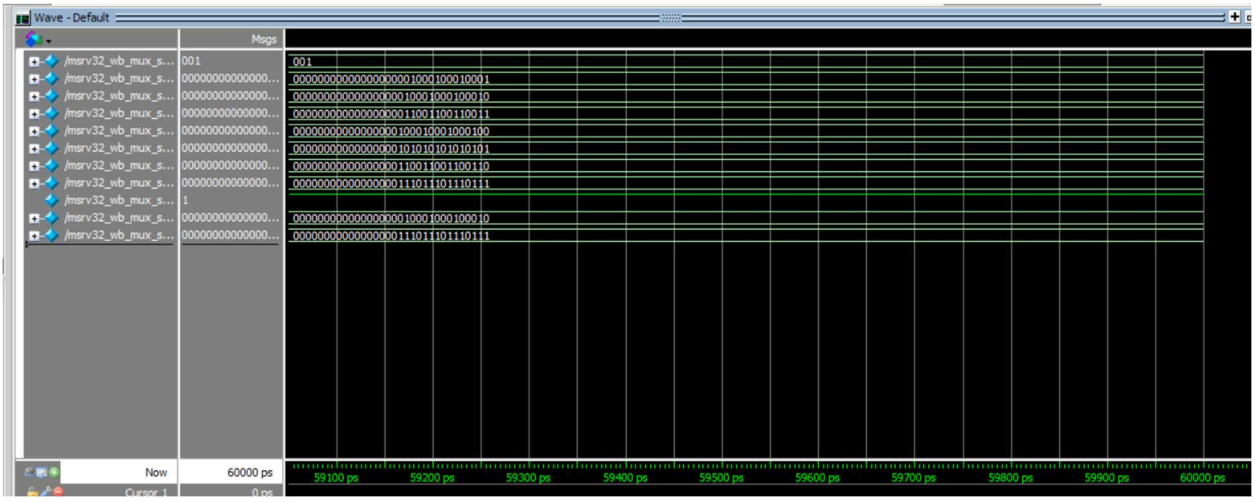
Block Diagram

2.16 WB Mux Selection Unit

2.16.1 Block Diagram



ModelSim Waveform



8. Verification Results

Once each RTL module was coded, it was put through functional verification using a testbench written specifically for that block. Each testbench applied a variety of input combinations to probe the module under different operating conditions, with all simulations running in ModelSim. The resulting waveforms were then examined in detail to confirm that the module outputs matched the expected behaviour set out in the MSRV32 Core Design Specification.

The ModelSim waveforms tell a clear story for every module: each one shows precisely how the outputs respond to the applied inputs, providing visible confirmation that the RTL is doing what the specification requires. Beyond validation, the simulation runs also served as a debugging tool during development, helping to surface and resolve syntax mistakes, logic errors, and interface mismatches.

With module-level simulation complete, the next step was instruction-level verification of the integrated core. This was performed using the Maven Silicon UVM environment accessed through MobaXterm. The terminal output for each run shows that the R-type, B-type, and J-type instruction sequences compiled cleanly and ran to completion without any issues, with the UVM report summary recording zero UVM_ERROR and zero UVM_FATAL counts in every case.

The sections that follow present the verification output for each module. For each one, the report includes:

- The RTL block diagram — and where relevant, the micro-architecture diagram — showing the internal structure of the module.
- The ModelSim waveform captured after simulation.

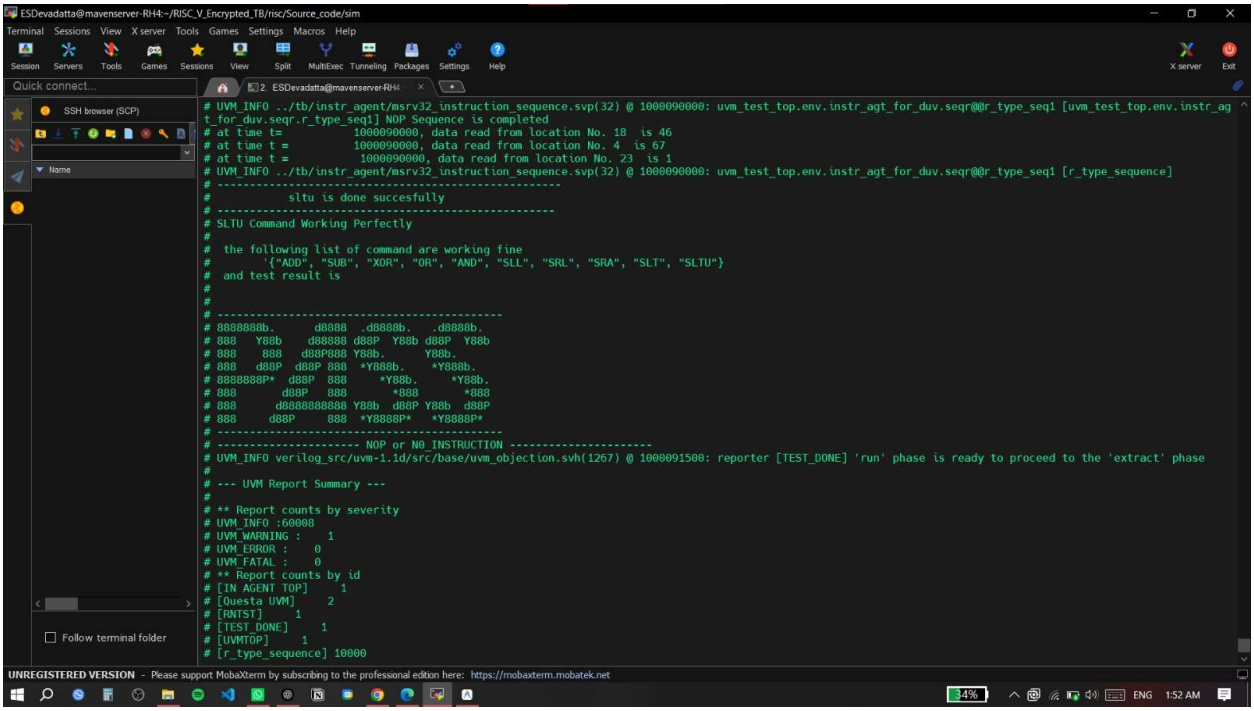
Following the per-module results, the instruction-level verification logs from the Maven Silicon UVM environment are presented, demonstrating that the R-type, B-type, and J-type instruction sequences all ran successfully on the verification server.

Taken together, the RTL designs, simulation waveforms, and passing verification logs confirm that every module was properly verified at the block level before being considered for incorporation into the complete processor.

8.1 Instruction-Level Verification on the Maven Silicon Server

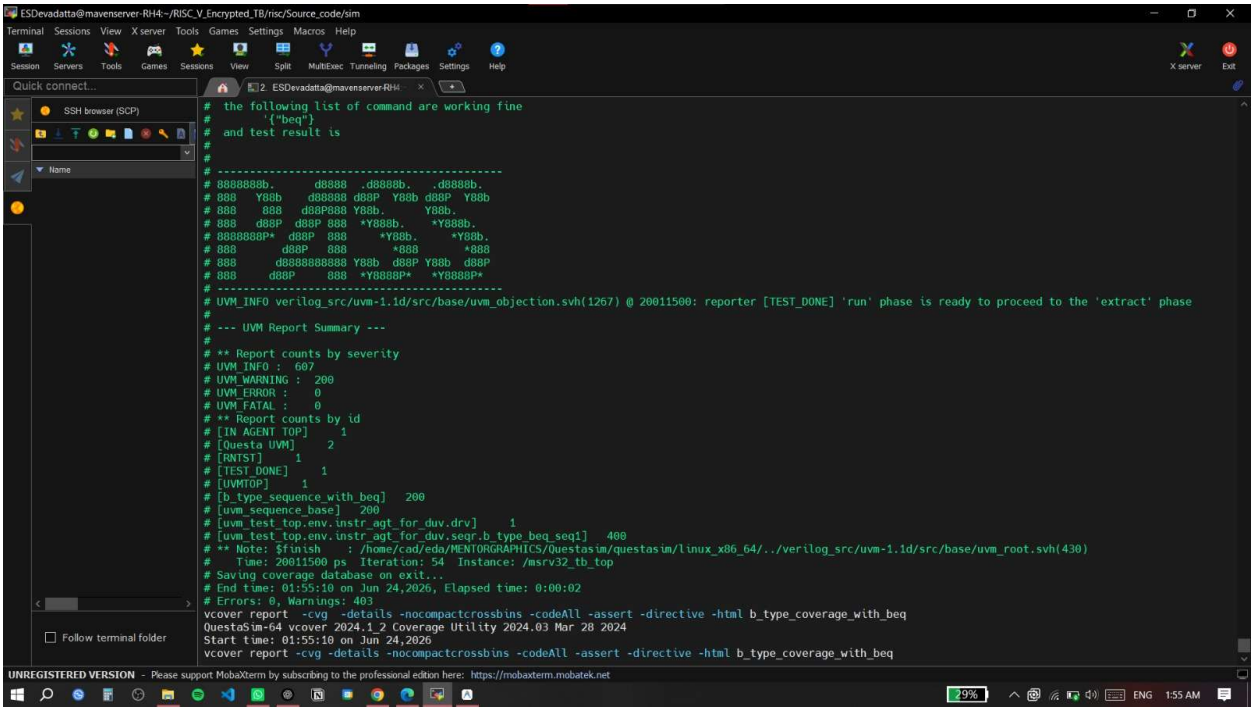
The MobaXterm terminal screenshots below capture the UVM simulation reports generated on the Maven Silicon verification server (mavenserver-RH4) for each instruction sequence run against the MSRV32 core. In all three cases — R-type, B-type, and J-type — the UVM Report Summary at the end of the log shows zero UVM_ERROR and zero UVM_FATAL counts, confirming that every instruction sequence executed against the design without any reported failures.

R type:

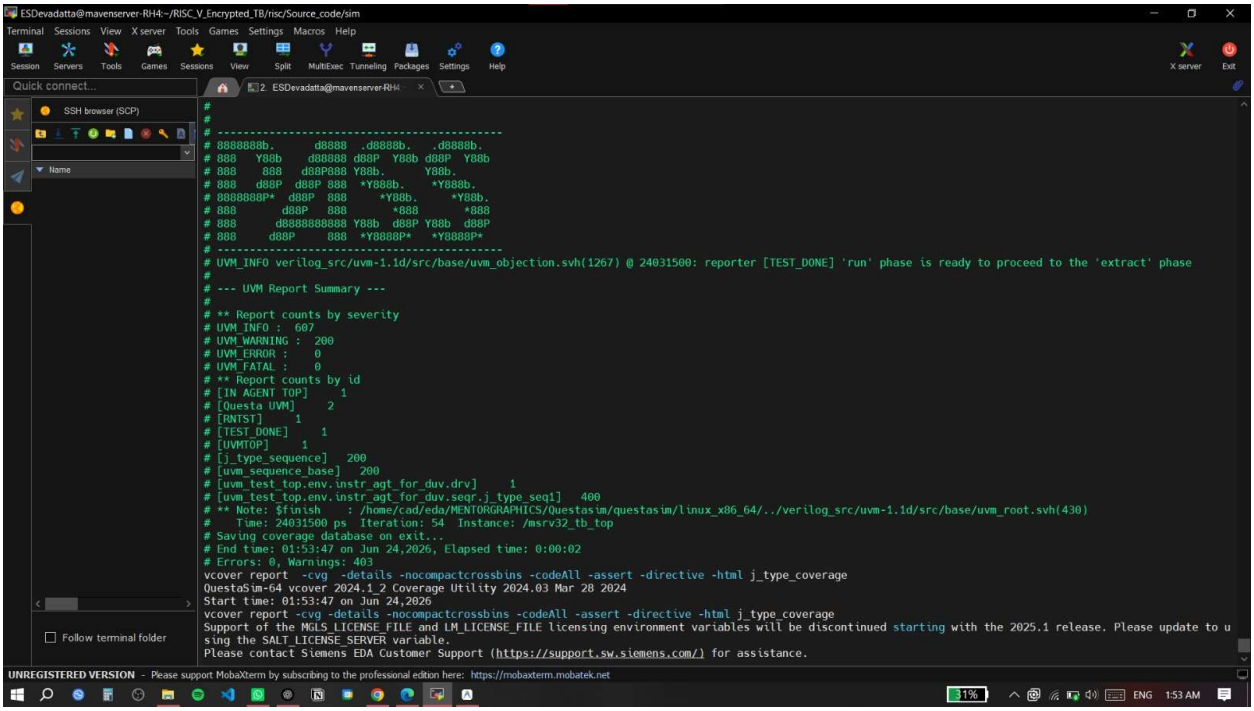


.

B type:


ESDevadatta@mavenserver-RH4:~/RISC_V_Encrypted_TB/risc/Source_code/sim
Terminal Sessions View X server Tools Games Settings Macros Help
Session Servers Tools Games Sessions View Split MultiExec Tunneling Packages Settings Help
X server Exit
Quick connect...
SSH browser (SCP)
Name
Follow terminal folder
2. ESDevadatta@mavenserver-RH4:
the following list of command are working fine
'{"beq"}
and test result is
8888888b. d8888 .d8888b. .d8888b.
888 Y88b d88888 d88P Y88b d88P Y88b
888 888 d88P888 Y88b. Y88b.
888 d88P d88P 888 *Y888b. *Y888b.
8888888P* d88P 888 *Y88b. *Y88b.
888 d88P 888 *888 *888
888 d8888888888 Y88b d88P Y88b d88P
888 d88P 888 *Y8888P* *Y8888P*
UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 20011500: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
--- UVM Report Summary ---
** Report counts by severity
UVM_INFO : 607
UVM_WARNING : 200
UVM_ERROR : 0
UVM_FATAL : 0
** Report counts by id
[IN_AGENT TOP] 1
[Questa UVM] 2
[RNTST] 1
[TEST_DONE] 1
[UVMTOP] 1
[b_type_sequence_with_beq] 200
[uvm_sequence_base] 200
[uvm_test_top.env.instr_agt_for_duv.drv] 1
[uvm_test_top.env.instr_agt_for_duv.seqr.b_type_beq_seq1] 400
** Note: $finish : /home/cad/eda/MENTORGRAPHICS/Questasim/questasim/linux_x86_64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
Time: 20011500 ps Iteration: 54 Instance: /msrv32_tb_top
Saving coverage database on exit...
End time: 01:55:10 on Jun 24,2026, Elapsed time: 0:00:02
Errors: 0, Warnings: 403
vcover report -cvg -details -nocompactcrossbins -codeAll -assert -directive -html b_type_coverage_with_beq
Questa Sim-64 vcover 2024.1_2 Coverage Utility 2024.03 Mar 28 2024
Start time: 01:55:10 on Jun 24,2026
vcover report -cvg -details -nocompactcrossbins -codeAll -assert -directive -html b_type_coverage_with_beq
UNREGISTERED VERSION - Please support MobaXterm by subscribing to the professional edition here: https://mobaxterm.mobatek.net
29%
ENG
1:55 AM

J type:


ESDevadatta@mavenserver-RH4:~/RISC_V_Encrypted_TB/risc/Source_code/sim
Terminal Sessions View X server Tools Games Settings Macros Help
Session Servers Tools Games Sessions View Split MultiExec Tunneling Packages Settings Help
X server Exit
Quick connect...
SSH browser (SCP)
Name
Follow terminal folder
UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 24031500: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
--- UVM Report Summary ---
** Report counts by severity
UVM_INFO : 607
UVM_WARNING : 200
UVM_ERROR : 0
UVM_FATAL : 0
** Report counts by id
[IN AGENT TOP] 1
[Questa UVM] 2
[RNTST] 1
[TEST_DONE] 1
[UVMTOP] 1
[j_type_sequence] 200
[uvm_sequence_base] 200
[uvm_test_top.env.instr_agt_for_duv.drv] 1
[uvm_test_top.env.instr_agt_for_duv.seqr.j_type_seq1] 400
** Note: $finish : /home/cad/eda/MENTORGRAPHICS/Questasim/questasim/linux_x86_64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
Time: 24031500 ps Iteration: 54 Instance: /msrv32_tb_top
Saving coverage database on exit...
End time: 01:53:47 on Jun 24,2026, Elapsed time: 0:00:02
Errors: 0, Warnings: 403
vcover report -cvg -details -nocompactcrossbins -codeAll -assert -directive -html j_type_coverage
QuestaSim-64 vcover 2024.1_2 Coverage Utility 2024.03 Mar 28 2024
Start time: 01:53:47 on Jun 24,2026
vcover report -cvg -details -nocompactcrossbins -codeAll -assert -directive -html j_type_coverage
Support of the MGLS_LICENSE_FILE and LM_LICENSE_FILE licensing environment variables will be discontinued starting with the 2025.1 release. Please update to using the SALT_LICENSE_SERVER variable.
Please contact Siemens EDA Customer Support (https://support.sw.siemens.com/) for assistance.
UNREGISTERED VERSION - Please support MobaXterm by subscribing to the professional edition here: https://mobaxterm.mobatek.net
31%
ENG 1:53 AM

9. Conclusion

All sixteen RTL modules of the MSRV32 3-stage pipelined 32-bit RV32I processor core were successfully implemented and verified over the course of this project. Every module was written independently in Verilog HDL, exercised against a dedicated testbench in ModelSim, and checked against the expected behaviour defined in the MSRV32 Core Design Specification. The assembled core then passed instruction-level verification through the Maven Silicon UVM environment, with R-type, B-type, and J-type test sequences all running to completion on the verification server with zero errors reported in any run.

Beyond the technical deliverables, the project was a meaningful learning experience. Working through the full design-and-verify cycle for a real pipelined processor built genuine proficiency in micro-architecture, RTL coding practice, testbench-driven simulation, and UVM-based instruction-level validation — grounding the theoretical knowledge of digital hardware design in practical, tool-driven work.