A logo for a laboratory

AI-generated content may be incorrect.A close-up of a logo

AI-generated content may be incorrect.

Customization of RISCV Code and Backend Design of a RISCV Processor Project report

**Students:**

Muhammad Biadsy

Omar Sharafy

**Supervised By:**

Amnon Stanislavsky

Contents

[**Abstract** 2](#_Toc199273892)

[**1.** **Introduction** 3](#_Toc199273893)

[1.1 Project Goals and Requirements 3](#_Toc199273894)

[1.2 Alternative Solutions 3](#_Toc199273895)

[1.3 Selected Solution 3](#_Toc199273896)

[**2. Verilog Implementation’s** 4](#_Toc199273897)

[2.1 SRAM & memory loading method Integration into RISCV 4](#_Toc199273898)

[2.1.1 Instruction memory changes 5](#_Toc199273899)

[2.1.2 Data memory changes 5](#_Toc199273900)

[2.1.3 Memory Loading Mechanism 6](#_Toc199273901)

[2.2 Debug unit Integration into RISCV 7](#_Toc199273902)

# **Abstract**

Our primary objective in this project is to modify and prepare the pipelined RISC-V processor, as taught in the “Digital Systems & Computer Structure” course, for a VLSI backend flow and potential chip fabrication while improving some of its functionally and adding debug capabilities.

The processor design provided in the course was intended for simulation and educational purposes only and is not synthesizable in its original form. To prepare it for fabrication, we made several structural and functional changes.

These include replacing the non-synthesizable memory initialization method with an external memory interface, switching to area-efficient compiled memories, and later improving the memory loading method and functionally of the module we worked on. and adding debugging capabilities that allow access to internal registers after fabrication.

As part of this effort, we also improved the Verilog implementation to support higher clock frequencies and better performance in the physical design flow.

On the path to achieving our goal, we modified and verified the RTL, followed by a backend flow including synthesis, scan insertion, logical equivalence checking, floor planning, power grid design, placement, clock tree synthesis, routing, and timing analysis.

# **Introduction**

## Project Goals and Requirements

Transit our current pipelined RISC-V module for implementation using Tower's 18nm RAM technology, reaching layout stage, with the following design goals:

* Maximumzing area & power efficiency.
* Optimizing chip memory.
* Create memory loading method.
* Debug capabilities for accessing internal registers post-fabrication.
* Optimizing chip Frequency.

## Alternative Solutions

The RISC-V processor we worked with can utilize FPGA-based memories and using the control unit as a debug tool to monitor internal signals. It also leverages the initial lines of code to load specific content into the memory. While this approach enables functional testing and debugging during development, it presents several limitations.

FPGA memories are not efficient in terms of area and power consumption, and the Verilog code used to integrate them is not optimal.

Additionally, the original design imposes restrictions on accessible memory addresses, which limits flexibility and scalability in more advanced use cases.

## Selected Solution

Using SRAM memories (Designed by Tower), which are both area & power-efficient, allowing simultaneous read and write operations to multiple addresses, and making the chip synthesizable.

A memory loading method will be added, to allow memory initialization and testing post-fabrication.

Additionally, creating a dedicated debugging unit, separate from the control unit, to provide additional debugging capabilities & accessing internal registers after fabrication, one register at a time, while halting the processor.

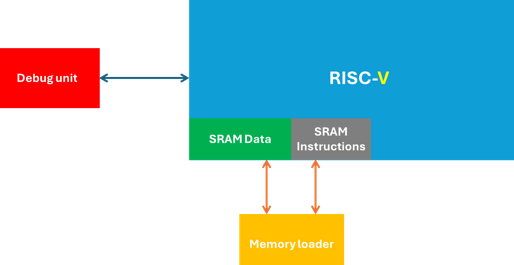


Figure 1: top-level description of the solution

* Add a second photo of the RISC-V with the debug unit wires and mux

# **2. Verilog Implementation’s**

## 2.1 SRAM & memory loading method Integration into RISCV

Both the instruction & data memories in the original RISC-V design had to be replaced.

However, the selected SRAMs do not support direct memory initialization during simulation. This limitation prevented the loading of test programs and data, effectively blocking the ability to verify and test the RISC-V core in a simulated environment.

To overcome this, a custom memory loading mechanism was implemented, Allowing the loading of both instruction and data memories at runtime via an external interface or testbench control, by enabling pre-execution memory initialization.

this solution ensures full support for simulation-based testing, preserving the functional validation and debugging capabilities of the RISC-V processor.

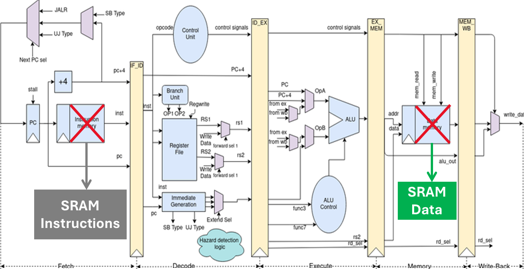


Figure 2: SRAM changes

### 2.1.1 Instruction memory changes

In the used RISC-V, an instruction is a vector of 32 bits.

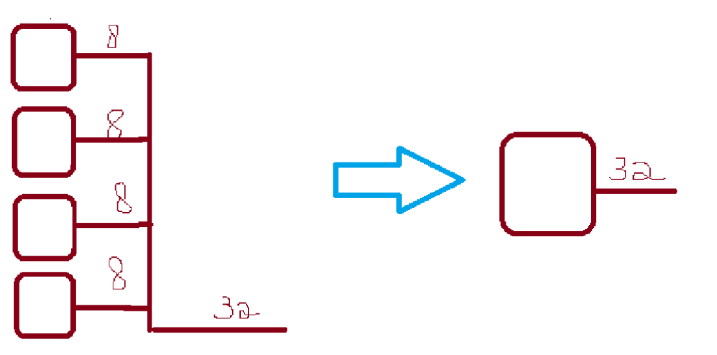
4 Fpga RAMS on chip (65536\*8) were replaced by one tower RAM (2048\*32), thus reducing wires and simplifying logic (less wires & eliminating the need of slicing of current instruction address & joining the sliced instruction parts from all Fpgas) in the processor.

Figure 3: Simplified Diagram of Instruction Memory Modifications

### 2.1.2 Data memory changes

In the implemented RISC-V design, load and store operations for bytes, half-words, and full words are fully supported.

However, it is important to note that correct and expected processor behaviour is guaranteed only when memory addresses used for these operations are word-aligned—that is, the address must be divisible by 4 (address % 4 == 0).

Misaligned accesses may lead to undefined behaviour or require additional hardware handling, which was not implemented in this design.

The four original FPGA memory blocks (each 65,536 × 8) were replaced with four Tower 18nm SRAM blocks (each 4,096 × 8), ensuring continued support for the previously mentioned memory operations.

To obtain full functionality Verilog logic was implemented, this logic extends the processor's capabilities to correctly handle load and store operations for all addressable memory locations—including those that are not word-aligned (i.e., addresses not divisible by 4). As a result, the processor now fully supports byte-, half-word-, and word-level memory access across the entire memory space.

The added logic **Reorders memory block addresses and data** accordingly, dynamically mapping each byte to the correct SRAM address and block, the least significant two bits of the address determine the alignment.

A diagram of a block diagram

AI-generated content may be incorrect.

Figure 4: Simplified Diagram of data Memory Modifications

### 2.1.3 Memory Loading Mechanism

As explained previously, Tower SRAMs do not support direct memory initialization during simulation, additionally, in the real world, post-fabrication, it is desirable to be able to load memory onto the chip to run benchmarks and tests.

A memory loading mechanism was implemented, utilizing three 32-bit vectors to specify the data and instructions to be loaded—two vectors for data and one for instructions. Additionally, three vectors were used to designate the addresses where the data and instructions would be loaded, along with two control wires to manage the loading process.

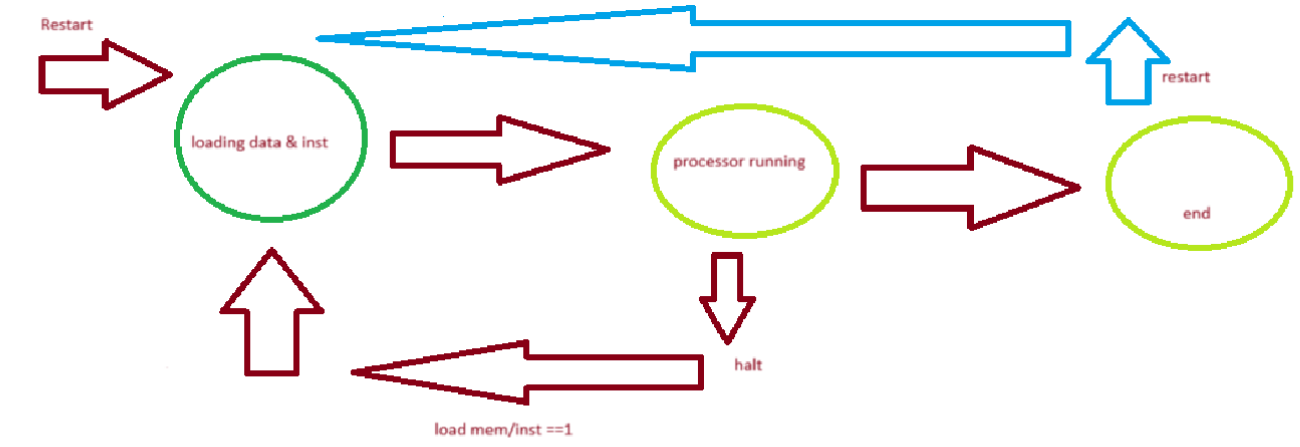
When one of the control wires is set, the processor halts and enters a “loading” state, where it loads the provided data to the specified addresses every cycle.

Figure 5: Simplified Diagram of processor states

## 2.2 Debug unit Integration into RISCV

In real-world chip design, post-fabrication testing and debugging are critical phases that require dedicated hardware support.

To address this need, a **debug unit logic** was integrated into the processor architecture, providing visibility into the internal operation of the system.

In this design, **27 crucial observation points** were selected across different pipeline stages and control/data paths, these points were connected to a **multiplexer-based system (32-way MUX)** controlled externally, allowing a user to dynamically select and monitor specific signals, enabling the inspection of key Datapath & memory elements such as the ALU result, fetched opcodes, register values, memory addresses, and data being read or written.

These points include the program counter (PC\_debug), control and data signals from all five pipeline stages (Fetch, Decode, Execute, Memory, and Write-back), memory interfaces, register file accesses, and the result from the write-back stage. The unit also tracks instruction decoding fields such as opcode, Funct3, and Funct7, propagating them through the pipeline via registered outputs.

The unit incorporates a **five-stage pipeline register tracing mechanism** for the opcode and function fields, allowing real-time observation of how an instruction moves through the pipeline. The outputs are exposed externally to provide full visibility into the instruction flow and control signal evolution, which is invaluable for debugging complex behaviours like hazards, control flow errors, or incorrect memory operations.

The enable debug signal allows an external user to activate debug mode and halt the processor, freezing its state for inspection. This is especially useful for post-silicon benchmarking and fault diagnosis. Combined with the memory loading mechanism, the debug unit provides a reliable post-fabrication interface for effective testing and validation.

The full logic implemntaion can be found in Design/DebugUnit.sv file.

# **Simulation results**

# **4.** **Synthesis and Layout**

## 4.1 Used Technology

To pressed with simplified backend flow for the design, the following tools and technologies were used:

**Synthesis:**

* **Tool:** Synopsys Design Vision
* **Technology:** TSMC TSL 108 Tower Design Kit  
  This setup was used to convert the RTL design into a gate-level netlist that matches the target manufacturing process.

**Layout:**

* **Tool:** Cadence Innovus.
* **Technology:** TSMC TSL 108 Tower Design Kit  
  This was used to create the physical layout of the chip, including placement, routing, and checking for design rule violations.

Using these tools ensured that the design could be correctly prepared for fabrication using the TSMC TSL 108 process.

## 4.1 Synthesis Flow

The **synthesis process** was performed using **Synopsys Design Vision** along with the **TSMC TSL 108 Tower Design Kit**.

* Firstly, the design files were setup, by both reading the relevant system Verilog files, analysing & elaborating parametrized designs, and creating the Hierarchy.
* Secondly, we define constraints: by creating a primary clock, maximum delays between signals were specified using commands like set\_max\_delay and set\_input\_delay.
* The design was synthesized using the compile function in Design Vision. This step mapped the RTL code to gate-level components supported by the technology library (TSL 108), optimizing for area, speed, and power.
* After synthesis, the design was checked for warnings and errors using Design → Check Design. Reports on area, timing, and power were generated to verify synthesis quality.

## 4.2 Layout Flow

# **5. Summary and Conclusions**

## 5.1 summary

In this project, we implemented code changes and Tower SRAM memory integration to a RISC-V-based processor with added support for post-fabrication testing, debugging, and memory loading.

We integrated Tower SRAMs, adding support for all memory space in memory operations, and using area & power effecting memories

We developed a custom debug unit, allowing external users to monitor internal signals and halt execution when needed. A memory loading mechanism was also integrated to simplify testing and benchmarking.

To test it, we modified a script to allow creating complex test benchmark for simulations and performed mutable tests.

For backend flow, we performed synthesis using Synopsys Design Vision and the TSL 108 Tower Design Kit, and used Cadence Innovus for physical layout.

## 5.2 Problems, Challenges & Solution

There were several challenges we faced, most notable is the size of our design.

Firstly, the added **Debug unit** was meant to expose to the user all the 27 critical points we determined as important, this led to a large number of inputs and outputs, this increasing the size of our design exponentially.

This meant our design was huge and contained multiple large wires that we passed thro the debug unit from the RISCV logic, creating a high density & complex design, which was also power consuming.

The large size and complex big wires also led to a high clock skew, resulting in low work frequency for the RISC-V.

To counter part this problem, we used a technique that is largely known in chip DFT industry, the Debug unit outputs were all connected to a 32 way mux, thus reducing the 27 outputs with ~600 bits to a 32bit output wire, thus reducing design size ( by about ~%60) and complexity ( skew decreased by ~%52, from 2.7 ns to 1.2)

# **6. Future Work**

During the semester (winter 2024/25), the same design was used in another project, which included creating a branching unit.

We would recommend merging the projects in the future, thus creating a powerful RISC-V with a branching unit, Full memory space support in load/store operations, an efficient SRAMS and a debug unit.

We would also like to integrate formal verification tools, specially jasper (which was added to the lab in 2024/25) to the project, since simulations are not powerful enough, especially in projects with small code & logic changes that have large impact.

Lastly, a full-backend layout prosses would increase even future our design and the maximum working frequency of the design.

# **7. References**