COGNICHIP HACKATHON 2026

RISC-V CPU Design using Generative AI

An ultra-lightweight 5-stage pipeline tailored for FPGA deployment.

Designed by **Jerry Zhou** & **Minjae Kim**

jz6277@nyu.edu | mk8400@nyu.edu

The Problem Domain

The Maker's Dilemma

Hobbyists and makers working on small personal projects often struggle to find computing solutions that meet strict power and size requirements.

Traditional x86: Too physically large, expensive, and power-hungry.

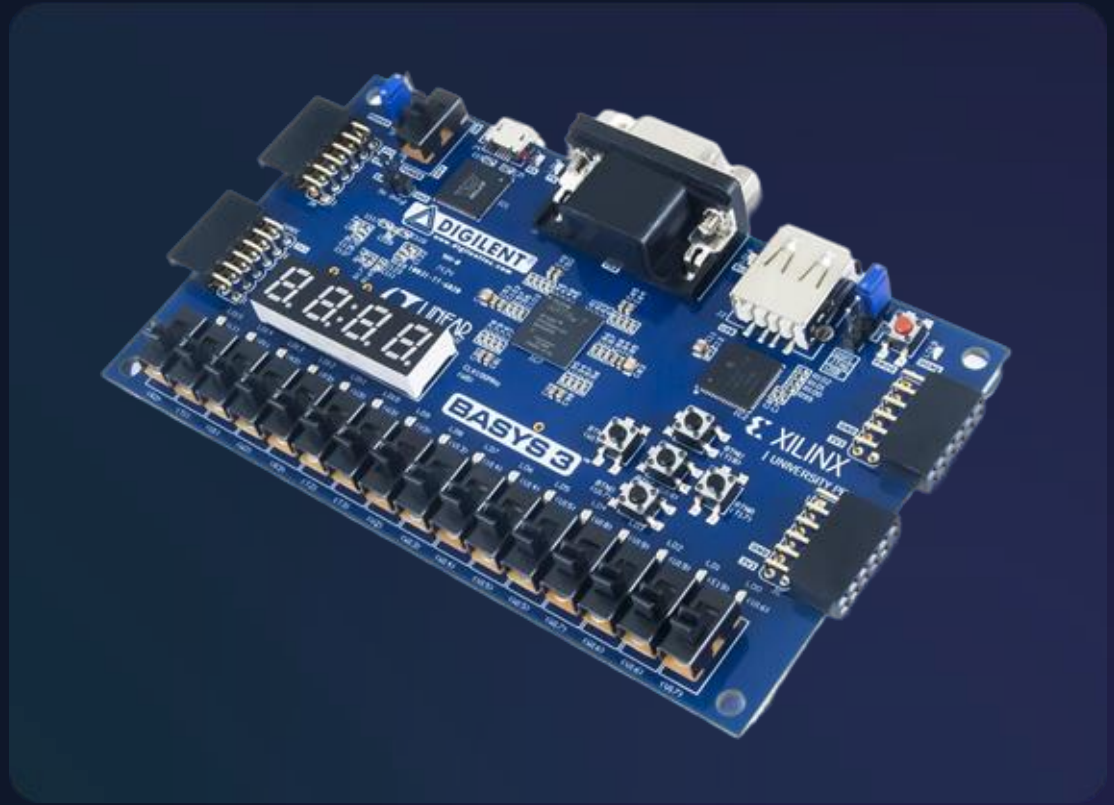
Off-the-shelf Microcontrollers: Closed systems limiting architectural customization.

Proposed Innovation

Accessible Bespoke Computing

We designed a custom computing solution perfectly sized for specific tasks without desktop overhead.

- **Target Hardware:** Specifically architected for low-cost hardware like the Basys 3 FPGA.
- **Streamlined ISA:** Focuses strictly on the RV32I base instruction set to optimize logic for speed and low resource usage.
- **Democratized Silicon:** Proves that custom, reliable processors can be built rapidly for niche applications.



Design Methodology: AI-Assisted Workflow

A key differentiator in our methodology is the integration of Generative AI (Cognichip) as a primary development tool.

RTL Generation

Instead of writing every line of Verilog by hand, we leveraged Large Language Models to accelerate the generation of Register Transfer Level (RTL) code for modular components like the ALU, register file, and control unit.

Rigorous Verification

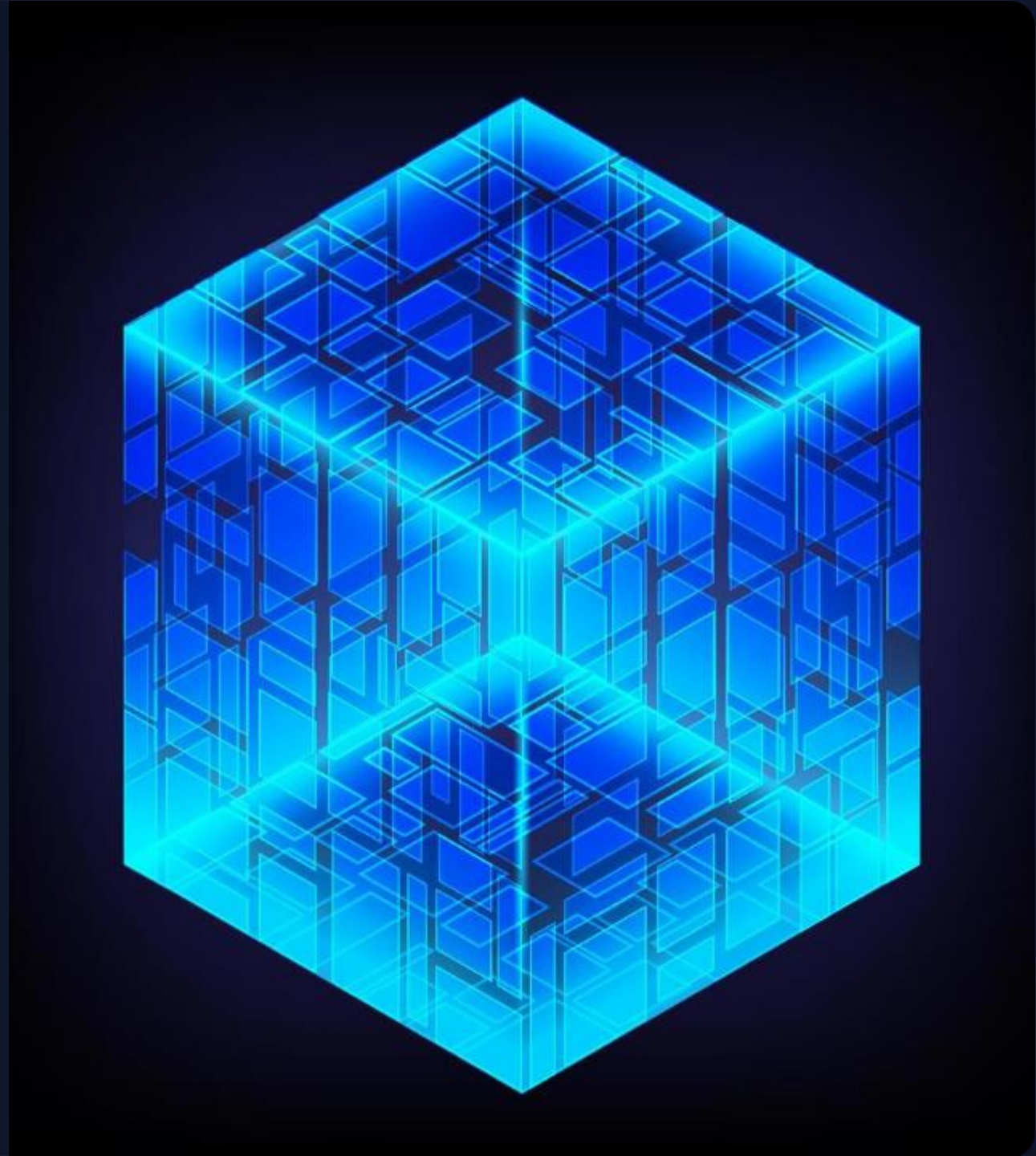
AI was utilized to generate comprehensive testbenches. This allowed us to rigorously validate our design using verilator for simulation and vaporview for signal verification, drastically reducing development time.

Architecture Overview

RV32I Core

The core of our innovation is the implementation of a classic five-stage in-order pipeline architecture. Written entirely in SystemVerilog.

- **Instruction Set:** RISC-V 32-bit Integer (RV32I).
- **Data Path:** Fully pipelined to maximize throughput and efficiency on limited hardware.
- **Hazard Handling:** Dedicated stall logic and data forwarding paths ensure reliable operation under complex instruction sequences.



RTL Description: Key Modules

ALU riscv_alu.sv

The largest module (~12.7 KB). Implements all RV32I arithmetic (ADD, SUB), logic (AND, OR, XOR), shifts, and comparisons.

Control Unit control_unit.sv

Decodes instruction opcodes, funct3, and funct7 fields to orchestrate control signals across all downstream pipeline stages.

L/S Unit load_store_unit.sv

Handles byte, halfword, and word-level memory accesses with proper sign extension for loads (e.g., LB, LH, LW).

The 5-Stage Pipeline Mechanics

Four inter-stage registers (pipeline_*.sv) hold signals to propagate state.

1. IF

Instruction Fetch

Fetches next instruction from ROM using PC.

2. ID

Instruction Decode

Decodes opcode, reads registers, sign-extends imm.

3. EX

Execute

ALU performs arithmetic/logic, resolves branches.

4. MEM

Memory Access

Handles Load/Store memory operations.

5. WB

Write Back

Writes final results back to the register file.

Simulation & Verification Strategy

A comprehensive testbench suite with one dedicated file per module ensures robustness.

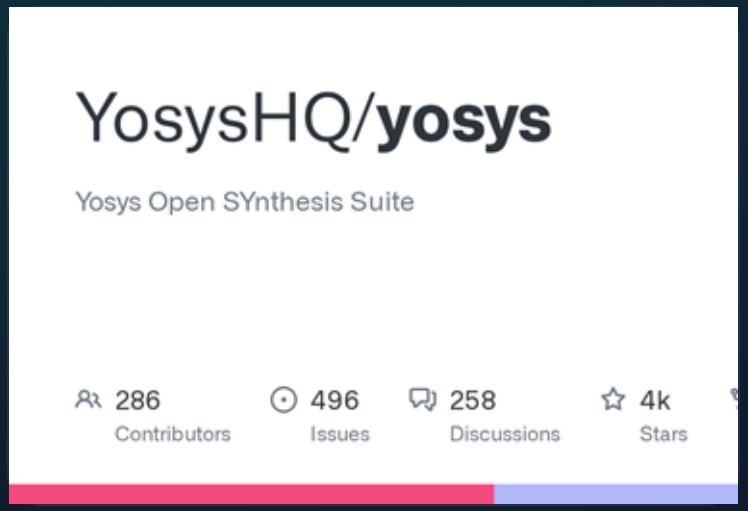
Testbench File	Target Component	Coverage Focus	Size
`tb_control_unit.sv`	Control Decoder	Exhaustive instruction decode coverage	~17.8 KB
`tb_riscv_alu.sv`	ALU	All arithmetic, logic, and shift operations	~12.6 KB
`tb_register_file.sv`	Register File	Read/write behavior, zero register enforcement	~12.0 KB
`tb_load_store_unit.sv`	Memory Interface	All load/store widths and sign extension rules	~11.3 KB
`tb_riscv_cpu_top.sv`	Top-Level System	Full pipeline integration and execution flow	~1.7 KB

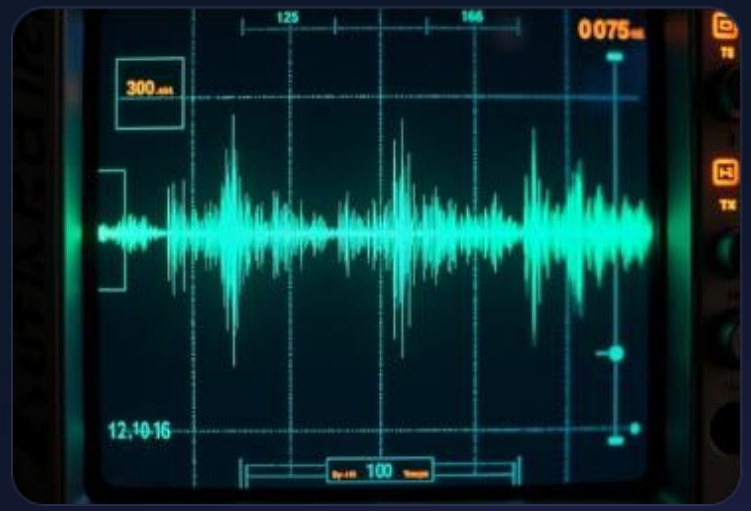
Simulation & Verification Strategy

```
========================================
RISC-V CPU Comprehensive ISA Test
========================================
Reset released, executing program...

Comprehensive RV32I Test Program Loaded:
  R-type: ADD, SUB, AND, OR, XOR
  I-type: ADDI, LW
  S-type: SW
  B-type: BEQ
  U-type: LUI
  J-type: JAL

========================================
Checking Results:
========================================
[PASS] x1 (I-type ADDI) = 10 (expected 10)
[PASS] x2 (I-type ADDI) = 5 (expected 5)
[PASS] x3 (R-type ADD) = 15 (expected 15)
[PASS] x4 (R-type SUB) = 5 (expected 5)
[PASS] x5 (R-type AND) = 0 (expected 0)
[PASS] x6 (R-type OR) = 15 (expected 15)
[PASS] x7 (R-type XOR) = 15 (expected 15)
```

```
[PASS] x8 (U-type LUI) = 0x12345000 (expected 0x12345000)
[PASS] x9 (J-type JAL) = 0x00000024 (expected 0x24)
[PASS] x10 (skipped by JAL) = 0 (expected 0)
[PASS] x11 (after JAL) = 20 (expected 20)
[PASS] x12 (I-type LW after S-type SW) = 10 (expected 10)
[PASS] x13 (skipped by BEQ) = 0 (expected 0)
[PASS] x14 (after BEQ) = 30 (expected 30)

====================================
Register File Summary:
====================================
x0  = 0x00000000 (0)
x1  = 0x0000000a (10)
x2  = 0x00000005 (5)
x3  = 0x0000000f (15)
x4  = 0x00000005 (5)
x5  = 0x00000000 (0)
x6  = 0x0000000f (15)
x7  = 0x0000000f (15)
x8  = 0x12345000 (305418240)
x9  = 0x00000024 (36)
x10 = 0x00000000 (0)
x11 = 0x00000014 (20)
x12 = 0x0000000a (10)
x13 = 0x00000000 (0)
x14 = 0x0000001e (30)

====================================
Test Results: 14 PASS, 0 FAIL
====================================
```

Synthesis & Simulation Toolchain





1. Yosys Synthesis

Two scripts provided (synth_alu.ys, synth_alu_simple.ys) for mapping RTL to logic gates targeted for FPGA architectures.

2. Verilator

Verilator used to compile the SystemVerilog source files and execute the AI-generated testbenches for functional verification.

3. Vaporview

Analyzed .vcd dump files to visually inspect signal propagation, pipeline hazards, and register states cycle-by-cycle.

Performance & Architecture Stats

5

Pipeline Stages

Breaking execution into discrete steps maximizes throughput and clock frequency.

32

Bit Architecture

RV32I base integer instruction set provides a balance of capability and extreme efficiency.

By relying heavily on dedicated stall logic and data forwarding paths, the design avoids pipeline flushing, maintaining high Instructions Per Cycle (IPC) even with data dependencies.

Challenges & Lessons Learned

Handling Pipeline Hazards Data dependencies and memory conflicts required precise timing. Designing the forwarding unit to intercept stale register data before execution was complex but critical for performance.

Managing AI "Hallucinations" in RTL While LLMs accelerated coding, they occasionally generated syntactically correct but functionally flawed logic (e.g., incorrect bit-widths in immediate generation).

The Value of Modular Verification Lesson learned: Generating the testbench before integrating the module into the top-level pipeline saved hours of debugging complex state issues.

Next Steps

The final deliverable will be a fully synthesized processor capable of executing assembly programs.

Focus: Implementing a UART interface over USB to communicate results back to a host PC.

Questions?

GitHub Repository: jz6277/5-Stage-Pipeline-RISC-V