CPU Verification Environment Product Summary (ZhenCPU)

Contents

1. Overview and Target Audience

1.1 Overview

The CPU Verification Environment developed in this project is not a user-facing software product, but rather an engineering verification framework designed to validate the correctness of a simplified RISC-style CPU. The environment demonstrates a complete simulation-based verification workflow and ensures the accurate behavior of the CPU's major architectural components—including the ALU, FSM controller, registers, multiplexers, counters, and memory subsystem.

To confirm functional correctness, the verification environment integrates directed simulation, SystemVerilog assertions, and functional coverage. These techniques collectively ensure that all instruction types, state transitions, and datapath interactions behave according to the specification. Waveform analysis and coverage reporting further provide quantitative and qualitative evidence of correct design functionality.

1.2 Target Audience

The intended audience is not general software users, but readers with foundational knowledge in digital design and hardware verification, including:

- Academic evaluators reviewing CPU correctness
- Entry-level verification engineers
- Students or researchers studying instruction flow and CPU microarchitecture

The audience is expected to have basic familiarity with SystemVerilog and digital logic concepts, and to understand waveform diagrams, although advanced ASIC or SoC experience is not required.

2. Features and Functionality

2.1 Verification Environment Features

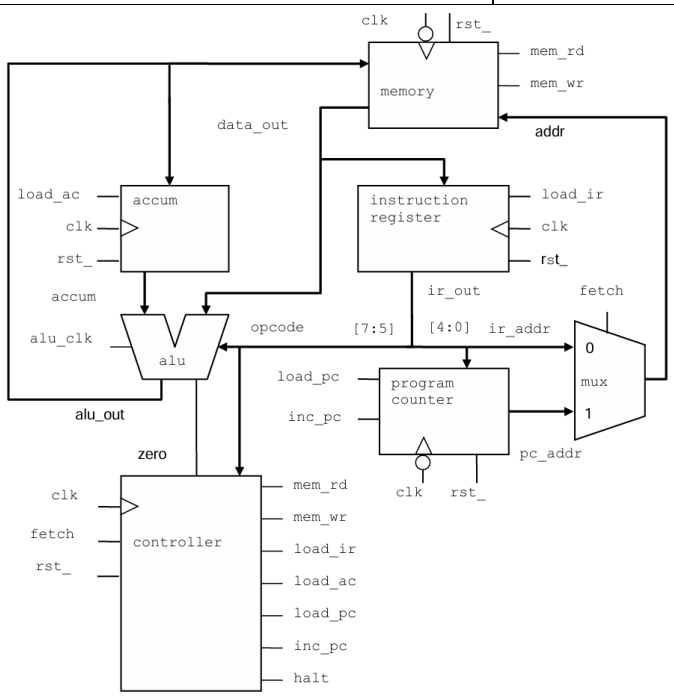
The verification environment provides the following key capabilities:

- **Directed Test Execution**
  Each CPU instruction, control signal, and datapath operation is exercised through deterministic, reproducible tests.
- **Assertion-Based Verification (ABV)**
  SystemVerilog Assertions check timing, one-hot FSM behavior, ALU correctness, memory sequencing, and illegal transitions.
- **Functional Coverage Collection**
  Coverage points track CPU instruction execution, FSM state occupancy, opcode usage, and key control events.
- **Waveform Inspection**
  All simulations generate waveforms viewable in Cadence SimVision, enabling cycle-by-cycle verification of internal CPU behavior.
- **Modular Testbench Architecture**
  Separate testbenches exist for module-level verification (FSM, ALU, Register, MUX, Counter) and top-level CPU integration.

- **Regression Capability**
  Multiple tests can be run in sequence to ensure stability and catch regressions after code updates.

2.2 Verified CPU Components

Module	Primary Function	Verification Objective
Register(accum & IR)	Stores intermediate computation results	Validate data integrity and synchronous updates
Counter	Generates clock-driven counts and timing references	Ensure accurate increment/decrement timing behavior
MUX	Selects input paths for ALU and memory operations	Verify correct input selection based on control signals
ALU	Performs arithmetic and logic operations (ADD, AND, XOR)	Confirm correctness across all opcode operations
FSM Controller	Manages FETCH–DECODE–EXECUTE instruction flow	Validate one-hot state encoding and transitions
Memory	Handles synchronous read/write data storage	Verify data integrity across multiple cycles



2.3 Deliverables Produced

The verification environment produces several artifacts that collectively demonstrate CPU correctness and verification completeness. Most outputs are generated through GUI tools such as SimVision and JasperGold, while formal coverage results are saved as standalone reports. The primary deliverables include:

• Waveform Visualizations (SimVision)

SimVision provides cycle-accurate waveform views of the CPU's internal behavior. These waveforms (stored internally as) reveal:.shm

- FSM transitions through FETCH → DECODE → EXECUTE
- ALU computation results for arithmetic and logical operations
- Register updates (ACC, IR)
- Memory read/write timing
- Control-to-datapath sequencing
- Instruction execution traces, which illustrate how each instruction propagates through the CPU pipeline

Waveform inspection serves as the primary method for validating instruction flow and signal correctness.

• Coverage Report (cov.rpt)

A text-based coverage summary is generated to document:

- Instruction coverage
- FSM state coverage
- Selected functional coverage events

This report provides quantitative evidence that the CPU was exercised across a comprehensive set of behaviors.

• Assertion and Formal Proof Results (JasperGold)

JasperGold performs formal property checking on the CPU's SystemVerilog Assertions. Instead of verifying individual components in isolation, JasperGold evaluates CPU-level behavior and safety conditions to ensure that the processor operates correctly under all legal input scenarios. Its GUI provides:

- Assertion pass/fail results across the CPU subsystem
- Formal verification of one-hot FSM encoding and legal state transitions
- Proof of correct instruction flow sequencing (FETCH → DECODE → EXECUTE)
- Analysis of control-signal safety (no illegal enable combinations, no conflicting writes)
- Validation of datapath integrity across registers, ALU outputs, and memory access timing
- Counterexample traces if a property fails
- Proof convergence summaries demonstrating complete property evaluation

These formal results complement simulation by confirming CPU correctness across all possible logical states, not only those reached through directed tests.

3. How the Environment Is Used to Verify the CPU

The verification workflow follows a structured, repeatable process that mirrors professional ASIC/SoC verification practices:

Step 1 — Navigate to the module directory

For example, for ALU verification:

```
cd ALU
```

Step 2 — Run the simulation script

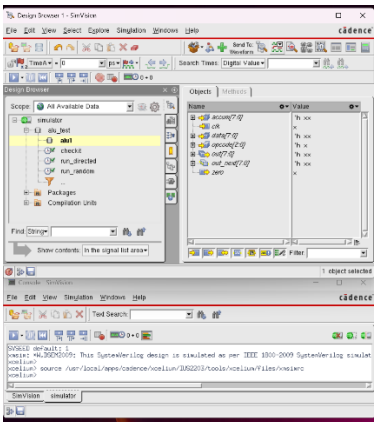
Each module includes a runme.sh script that automatically invokes Xcelium:

```
./runme.sh
```

What the script does

When Questa/Xcelium is installed and accessible in the PATH, executing runme.sh performs the following actions:

- Calls xrun to compile and simulate all SystemVerilog source files.
- Generates waveform output (shm database).
- Automatically opens SimVision to allow waveform inspection.

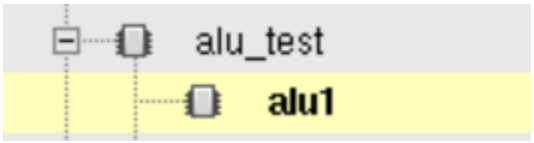


This enables the user to enter the folder and start the entire verification environment with a single command.

Step 3 — Select the ALU instance (alu1)

In the Design Browser, expand:

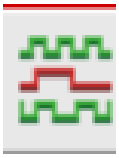
```
alu_test → alu1
```



This tells SimVision which module's internal signals you want to observe.

Step 4 — Click the waveform icon

Click the waveform icon (green/red square-wave symbol) to send the selected signals to the waveform window.

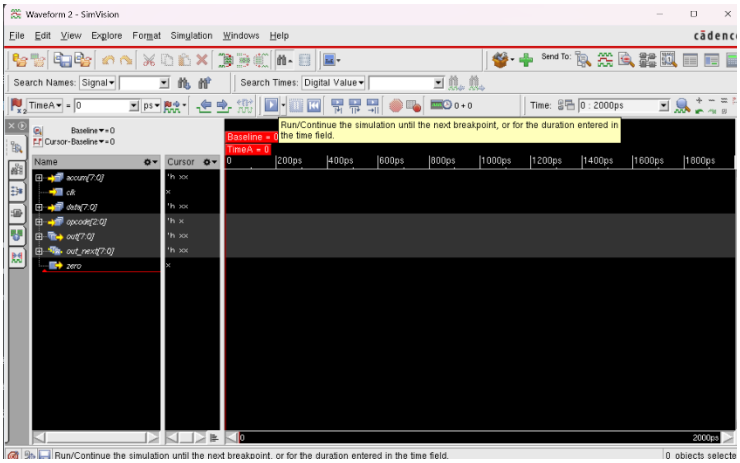


Step 5 — Waveform window appears

SimVision opens a new waveform viewer containing signals such as:

- accum
- clk
- data

- opcode
- out
- zero

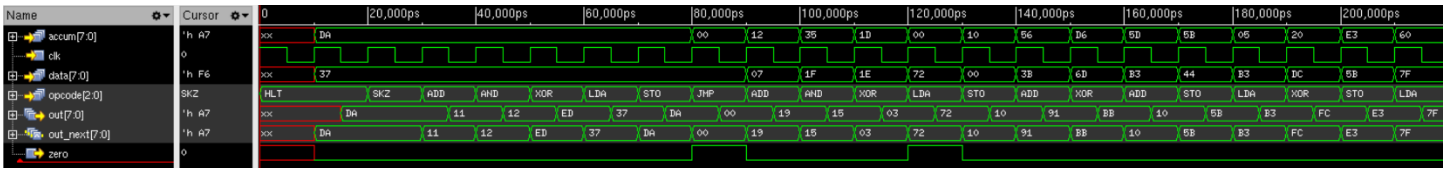


Step 6 — Run the simulation

Click the Run button (blue triangle) to populate the waveform with signal activity.

Step 7 — Observe instruction execution

After pressing Run, the waveform shows complete ALU/CPU behavior across cycles.



Through this waveform, the user can verify:

- Correct opcode sequencing
- ALU computation for ADD / AND / XOR
- Accumulator updates
- Memory read/write timing

Step 8 — Generate Coverage Report (cov.rpt)

After inspecting the waveform, the user should close the SimVision window before running coverage collection. Xcelium cannot generate a coverage report while the GUI is open.

Once SimVision is closed, the user can check functional coverage to confirm that all instructions and ALU behaviors were exercised during simulation. In the same module directory, run:

```
./runcover.sh
```

This script performs the following:

- Re-runs the simulation in coverage collection mode
- Extracts covergroups, coverpoints, and bins
- Produces a final report named cov.rpt

The generated report includes metrics such as:

- Opcode coverage
- ALU output bin coverage (low/mid/high ranges)
- Zero-flag coverage
- Cross-coverage where applicable

```
Covergroup Coverage:
    Covergroups                  3        na        na   100.00%
        Coverpoints/Crosses      3        na        na        na
            Covergroup Bins     13        13         0   100.00%
--------------------------------------------------------------------------------------------
Covergroup                                          Metric       Goal       Bins    Status

--------------------------------------------------------------------------------------------
 TYPE /alu_test/cg_opcode                         100.00%        100          -    Covered
    covered/total bins:                                 8          8          -
    missing/total bins:                                 0          8          -
    % Hit:                                        100.00%        100          -
    Coverpoint cp_opcode                          100.00%        100          -    Covered
        covered/total bins:                             8          8          -
        missing/total bins:                             0          8          -
        % Hit:                                    100.00%        100          -
 Covergroup instance \/alu_test/cov_opcode        100.00%        100          -    Covered
    covered/total bins:                                 8          8          -
    missing/total bins:                                 0          8          -
    % Hit:                                        100.00%        100          -
    Coverpoint cp_opcode                          100.00%        100          -    Covered
        covered/total bins:                             8          8          -
        missing/total bins:                             0          8          -
        % Hit:                                    100.00%        100          -
        bin HLT                                        38          1          -    Covered
        bin SKZ                                        38          1          -    Covered
        bin ADD                                        31          1          -    Covered
        bin AND                                        39          1          -    Covered
        bin XOR                                        32          1          -    Covered
        bin LDA                                        45          1          -    Covered
        bin STO                                        39          1          -    Covered
        bin JMP                                        52          1          -    Covered
 TYPE /alu_test/cg_zero                           100.00%        100          -    Covered
    covered/total bins:                                 2          2          -
    missing/total bins:                                 0          2          -
    % Hit:                                        100.00%        100          -
    Coverpoint cp_zero                            100.00%        100          -    Covered
        covered/total bins:                             2          2          -
        missing/total bins:                             0          2          -
        % Hit:                                    100.00%        100          -
 Covergroup instance \/alu_test/cov_zero          100.00%        100          -    Covered
    covered/total bins:                                 2          2          -
    missing/total bins:                                 0          2          -
    % Hit:                                        100.00%        100          -
    Coverpoint cp_zero                            100.00%        100          -    Covered
        covered/total bins:                             2          2          -
        missing/total bins:                             0          2          -
        % Hit:                                    100.00%        100          -
        bin z0                                        309          1          -    Covered
        bin z1                                          4          1          -    Covered
 TYPE /alu_test/cg_out                            100.00%        100          -    Covered
    covered/total bins:                                 3          3          -
    missing/total bins:                                 0          3          -
    % Hit:                                        100.00%        100          -
    Coverpoint cp_out                             100.00%        100          -    Covered
        covered/total bins:                             3          3          -
        missing/total bins:                             0          3          -
        % Hit:                                    100.00%        100          -
 Covergroup instance \/alu_test/cov_out           100.00%        100          -    Covered
    covered/total bins:                                 3          3          -
    missing/total bins:                                 0          3          -
    % Hit:                                        100.00%        100          -
    Coverpoint cp_out                             100.00%        100          -    Covered
        covered/total bins:                             3          3          -
        missing/total bins:                             0          3          -
        % Hit:                                    100.00%        100          -
        bin low                                        43          1          -    Covered
        bin mid                                       134          1          -    Covered
```

This coverage report gives the user a measurable way to confirm that the verification environment reached all required CPU behaviors.

Step 9 — Running Formal Verification in JasperGold

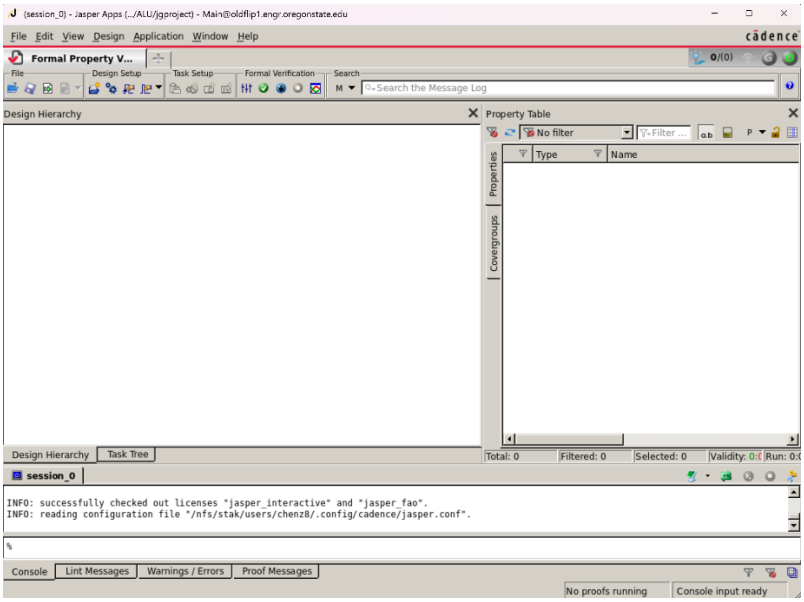
In addition to simulation and coverage, the user can run formal verification to mathematically prove ALU and CPU correctness. JasperGold evaluates SystemVerilog Assertions and checks all legal input combinations, not just the ones reached during simulation.

Launching JasperGold

To start JasperGold, invoke the tool from the command line:

```
jg
```

This opens the JasperGold GUI environment.



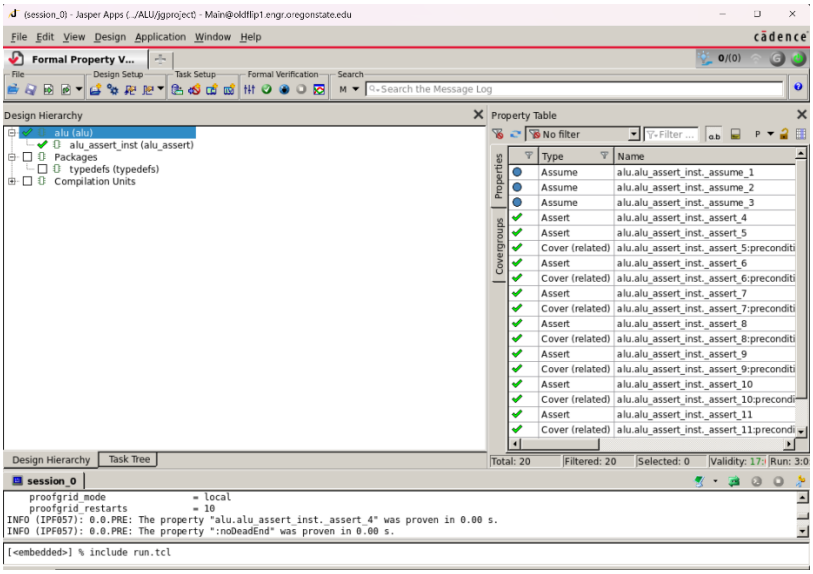
Running the formal verification script

Once JasperGold is open, the user can load and run the provided Tcl automation script:

```
include run.tcl
```

The Tcl script automatically:

- Loads the ALU RTL and alu_assert.sv assertion module
- Compiles and elaborates the design
- Runs all proof engines
- Reports results in the GUI (assumptions, assertions, coverage properties)



Step 10 — Review functional coverage

The simulator generates a cov.rpt file summarizing which behaviors were exercised.

Step 11 — Confirm correctness

By combining waveform inspection, formal proofs, and functional coverage, the user verifies that the CPU behaves correctly across the entire instruction set.