eSim Summer Fellowship 2026

Report On

Digital IPs, Feature Additions & Microwatt Co-Simulation in eSim

Submitted by

Akshat Sharma

B.Tech ECE

VIT Bhopal University

Under the Guidance of

Prof. Prabhu Ramachandran

Principal Investigator,

Department of Aerospace Engineering,

Indian Institute of Technology Bombay

July 31, 2026

Acknowledgment

I would like to express my deepest gratitude to **Prof. Prabhu Ramachandran** for providing me the opportunity to be a part of the FOSSEE internship program and for supporting open-source engineering tool development at IIT Bombay. His guidance and vision have encouraged students and researchers to actively contribute to the open-source ecosystem.

I would also like to acknowledge **Prof. Kannan M. Moudgalya** for his foundational role in establishing and nurturing the FOSSEE initiative. His contributions toward open-source education and the creation of the FOSSEE fellowship framework have directly shaped the platform through which this internship was undertaken.

I would also like to acknowledge and thank my Professors at **VIT Bhopal University**, for their constant support, encouragement, and guidance throughout this internship. Their valuable advice and motivation have been instrumental in helping me manage academic and project responsibilities effectively and in successfully completing this internship.

My sincere appreciation extends to my mentor, **Sumanto Kar**, for his continual support, technical guidance, and encouragement throughout the duration of this project. His insights and feedback played a key role in refining ideas, overcoming challenges, and ensuring timely completion of the tasks assigned to me.

I would also like to thank my internal mentors, **Mr. Varad Patil** and **Ms. Shanthi Priya K**, for their valuable guidance, coordination, and technical inputs during the internship. Their mentorship contributed significantly to the clarity, progress, and successful execution of the work.

This internship has been an enriching learning experience, allowing me to work closely with open-source EDA tools, develop IC subcircuits in eSim, and gain exposure to real-world circuit modelling and simulation workflows. The knowledge acquired during this period will undoubtedly support my future academic and professional pursuits.

I would also like to thank the entire **FOSSEE team** for their coordination, assistance, and timely interactions at various stages of this work. Their collective efforts ensured smooth workflow, resource accessibility, and effective project execution.

Contents

1 The eSim Mixed-Signal Ecosystem

Before detailing the specific Digital Intellectual Property (IP) blocks designed during this fellowship, it is crucial to understand the underlying architecture of the eSim electronic design automation (EDA) environment.

eSim is an open-source EDA tool for circuit design, simulation, analysis, and PCB creation. Developed by FOSSEE at IIT Bombay, it serves as a powerful alternative to proprietary software by seamlessly integrating several robust open-source engines into a single unified workflow.

1.1 KiCad (Schematic Capture)

The front-end user interface of eSim relies on Eeschema, the schematic capture tool from the KiCad EDA suite. KiCad provides the graphical canvas where engineers place components, draw wires, and define global or local nets. When a schematic is completed in eSim, the software parses the KiCad graphical data and generates a standard SPICE netlist, which maps every component and its nodal connections.

1.2 Ngspice (Analog Simulation Engine)

The core mathematical solver driving eSim is Ngspice, a mixed-level, mixed-signal circuit simulator based on the industry-standard SPICE3 engine. Ngspice evaluates the generated netlist by constructing complex matrices to solve for nodal voltages and branch currents over time (Transient Analysis) or frequency (AC Analysis). Because Ngspice is fundamentally an analog solver, evaluating purely digital logic using discrete analog transistors is computationally expensive and prone to matrix convergence failures.

1.3 NgVeri & Makerchip (Digital-to-Analog Bridge)

To solve the computational limitations of simulating complex digital logic in an analog SPICE matrix, eSim utilizes the NgVeri module (often integrated with Makerchip/Verilator).

When a custom digital IP is written in Verilog, Verilator compiles the HDL code into a highly optimized C++ model. During simulation, Ngspice and the C++ model run concurrently. To bridge the gap between the continuous analog voltages of Ngspice and the discrete binary states of the C++ model, eSim employs specific interface components:

- **ADC Bridge (Analog-to-Digital):** Translates continuous SPICE node voltages into discrete `1`s and `0`s based on user-defined threshold parameters (e.g., voltages above 3V become Logic 1).
- **DAC Bridge (Digital-to-Analog):** Translates the binary outputs of the Verilator C++ model back into analog voltages to be plotted by Ngspice.

This mixed-signal architecture allows engineers to simulate advanced digital accelerators—such as Neural Network Activations and Mathematical Co-processors—alongside real-world analog sensors and power electronics with zero-delay precision.

2 Digital IP Design and Verification Methodology

Before the individual IP datasheets, this section sets out the design rules and the verification flow that every digital block in this report follows. Collecting them here avoids repeating the same reasoning in each chapter, and it documents a number of practical constraints imposed by the NgVeri tool-chain that are not obvious from eSim's user documentation.

2.1 Coding Style for NgVeri Compatibility

NgVeri drives two independent tool-chains from the same Verilog file: a Python front-end that parses the port list to build the KiCad symbol and the `modelParamXML` entry, and Verilator, which compiles the design into a C++ model. The two do not accept exactly the same subset of the language, and constructs that are perfectly standard Verilog can satisfy one while confusing the other. Declaring a port as `output reg` and assigning it inside an `always` block proved to be one such construct.

The defensive pattern adopted throughout is therefore: ports are declared as plain **wires**, all sequential and combinational logic writes to an **internal `_reg` variable**, and a single **continuous `assign`** drives the port from that register.

Listing 1: The NgVeri-safe port pattern

```
module example_ip (
    input        clk,
    input  [7:0] data_in,
    output [7:0] data_out      // plain wire, never "output reg"
);
    reg [7:0] data_out_reg;    // all logic writes here

    always @(posedge clk) begin
        data_out_reg <= data_in;
    end

    assign data_out = data_out_reg;   // single continuous driver
endmodule
```

This compiles cleanly under Verilator, is parsed correctly by the NgVeri front-end, and costs nothing in synthesised hardware — the register and the wire collapse into the same flip-flop.

2.2 Datapath Width Policy

Verilog `parameter` declarations are not handled reliably by the NgVeri parser, so every block in this report has its datapath width **hard-coded** rather than parameterised. Producing a different width is consequently a mechanical edit of the port declarations rather than a change of one constant.

Two approaches follow from that constraint. Where demonstrating scalability was itself a goal, a block is released as a family of explicitly instantiated variants — the Advanced Math ALU in 8, 16 and 32 bits, and the activation functions in 8, 16, 32 and 64 bits. Where scalability adds no new information, a **single 8-bit width** is fixed instead. That choice is deliberate rather than incidental: NgVeri generates a separate model, symbol and XML entry per module, so each additional variant enlarges the library; every extra bit costs two Ngspice bridge channels in every test schematic; and each variant multiplies the verification and documentation effort. Some widths are not a datapath

choice at all — the one-shot's 8-bit port is a *duration*, and the seven-segment driver's widths are fixed by the display hardware.

2.3 Port Naming

NgVeri flattens every vector port into individual pins, and long pin names overflow the generated symbol's bounding box and collide with neighbouring text in KiCad. Names are therefore kept **short but unambiguous**, capped at roughly twelve characters, and boolean controls encode their active sense in the name itself — `up_dwn_not` is high for up-counting, `cnt_en` enables counting — so the direction of a control line can be read off the schematic without consulting a datasheet.

2.4 Two-Phase Verification Flow

Every block passes through the same two-stage gate:

1. **Phase 1 — Pure digital (Xilinx Vivado XSim).** A self-checking testbench drives directed and, where appropriate, randomised stimulus, compares every result against an independently computed expectation, and prints a formatted pass/fail table to the Tcl console. A block is not converted until that table reports zero failures.

2. **Phase 2 — Mixed-signal (eSim / Ngspice).** The verified block is compiled by NgVeri into an Ngspice code model, instantiated in a KiCad schematic between `adc_bridge` and `dac_bridge` components, driven by analog `pulse` and `pwl` sources, and re-verified in the continuous-time domain.

Unless a chapter states otherwise, the bridge thresholds are left at the Ngspice defaults, which already suit 5 V logic: `adc_bridge` with `in_low = 1.0` and `in_high = 2.0`, and `dac_bridge` with `out_low = 0.0` and `out_high = 5.0`.

2.5 Generation of the Mixed-Signal Test Schematics

Wiring a digital block into an analog matrix by hand is mechanical and error-prone: an n-bit output needs n DAC channels, every unused input must be tied to a defined level to avoid a singular SPICE matrix, and a single mis-placed wire produces a netlist that simulates but is silently wrong. The later test schematics were therefore **generated programmatically** from the symbol library and then machine-checked.

The generator reads each block's pin list directly from the installed `eSim_Ngveri` symbol library — so a schematic can never disagree with the symbol eSim actually instantiates — and emits a KiCad 6 schematic with a fixed left-to-right topology:

$$\texttt{sources} \rightarrow \texttt{adc\_bridge} \rightarrow \textbf{DUT} \rightarrow \texttt{dac\_bridge} \rightarrow \texttt{plot\_v1}$$

Four rules keep the result both correct and readable:

- **Connectivity by global label.** Every pin is wired to a short stub terminated by a named global label, so net names in the exported netlist are meaningful (`cnt_val7`, `flt_flags3`) rather than auto-generated (`Net-_U2-Pad7_`).

- **Shared logic rails.** Pins that hold a constant level are not given individual sources: every logic-1 pin joins a single `VDD` net and every logic-0 pin is tied to `GND`. Only signals that actually move during a test — clock, reset, trigger, a deliberately corrupted channel — receive their own source.
- **Exact-width bridge banks.** Bridges are instantiated at precisely the required width (`adc_bridge_3`, `dac_bridge_4`, and so on), so no bridge channel is ever left floating.
- **Grid discipline.** Every symbol, wire endpoint and label anchor is snapped to KiCad's 1.27 mm grid, so a component can be moved without silently detaching a wire.
- **Standard annotation.** Reference designators are assigned in spatial order (left to right, then top to bottom) exactly as KiCad's own *Annotate* tool does, numbered contiguously from 1 within each prefix, with the designator drawn above the component value. Because the numbering has no gaps or duplicates and the `symbol_instances` table agrees with every symbol, a user who later drops an extra component onto a sheet simply receives the next free designator instead of being forced to re-annotate the whole drawing.

Each generated schematic also carries an on-sheet **setup note** listing the exact source values, the bridge defaults and the transient settings, so the information needed to reproduce a run travels with the drawing rather than living in a separate document.

2.5.1 Machine Validation of the Generated Schematics

Because these schematics are generated rather than drawn, they can be validated mechanically. Five independent checks were applied to every sheet:

Check	What it proves	Result
S-expression parse	the file will load in KiCad (balanced parentheses, terminated strings)	all sheets parse
Connectivity	every pin reaches a net; no net has fewer than two members	0 floating nets
Grid alignment	every coordinate is a multiple of 1.27 mm	0 off-grid coordinates
Geometric collision	no symbol body or label text overlaps another	0 collisions
Netlist equivalence	the exported net set is unchanged after cosmetic re-layout	all equivalent

Table 1: Automated validation applied to every generated test schematic.

The netlist-equivalence check deserves emphasis. The schematics were re-laid-out several times for legibility (spacing, grid snapping, label justification, probe orientation). After each pass the exported net set was compared against the previous one and found **identical**, confirming that the cosmetic work could not have altered circuit behaviour. This was then confirmed empirically: re-exporting and re-converting every project from scratch reproduced the original waveforms exactly.

2.5.2 Portability

The delivered projects contain **no absolute paths**, and each schematic **embeds its symbol graphics**, so every sheet opens and renders on any eSim installation as-is. The only machine-local dependency is the compiled Ngspice model itself; each project folder

therefore ships the block's Verilog source, so a user on a fresh machine registers the model once through NgVeri and then converts and simulates normally.

3 Digital IP Datasheet & Verification Report: Advanced Math ALU

3.1 Introduction & Purpose

The **Advanced Math ALU** is a purely combinational mathematical co-processor for mixed-signal simulation. In an analog-only SPICE environment, non-linear operations such as extracting a square root or evaluating a vector magnitude $\sqrt{A^2 + B^2}$ require unstable arrays of op-amps and analog multipliers, which suffer from voltage clipping, thermal drift and matrix convergence failures. This IP moves that arithmetic into the digital domain: an analog design routes sensor voltages through `adc_bridge` components, the ALU evaluates the expression exactly and with zero delay, and `dac_bridge` components return the result to the analog matrix.

The block is provided in three explicitly instantiated widths — `advanced_math_alu_8bit`, `_16bit` and `_32bit` — because the NgVeri parser does not handle Verilog `parameter` declarations reliably.

3.2 Interface Design: One Pin per Operation

The defining feature of this IP is how the operation is selected. An earlier revision used a conventional 3-bit `opcode` bus, where `3'b101` meant square root and `3'b110` meant vector magnitude. That encoding is compact, but it is **opaque on a schematic**: a user placing the symbol sees three anonymous pins and must consult a datasheet to discover which binary pattern produces which operation. In an EDA tool aimed at analog engineers, that is a genuine usability defect — and an easy source of silent error, because a mis-set opcode bit yields a perfectly valid-looking wrong answer.

The interface was therefore redesigned around a principle of **self-documenting ports**: every operation has **its own named enable pin**. To compute a square root the user asserts `op_sqrt`; to divide, `op_div`. The schematic then reads as a description of the circuit's intent, and no lookup table is needed at any point.

Aspect	Encoded opcode (previous)	One pin per operation (current)
Selecting $\sqrt{A}$	drive `opcode` = `3'b101`	assert `op_sqrt`
Readability on a schematic	requires the datasheet	the pin name states the function
Effect of a wrong bit	silently performs another operation	selects a different, clearly named pin
No selection made	`opcode` = `000` performs an addition	`result_valid` goes low, outputs read zero
Several selections at once	impossible to express	detected; `result_valid` goes low

Table 2: Why the operation-select interface was redesigned.

3.3 Architecture & Pin Specifications

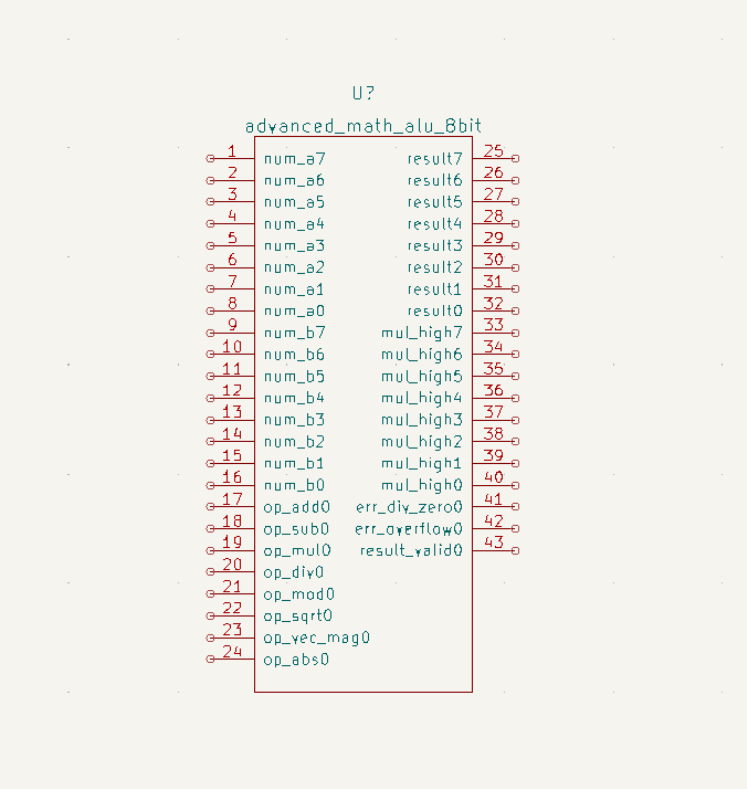


Figure 1: The NgVeri-generated KiCad symbol for `advanced_math_alu_8bit`. All 43 pins are named for their function: the operand buses `num_a7..0` and `num_b7..0`, the eight operation-select pins `op_add0` through `op_abs0`, and the outputs `result7..0`, `mul_high7..0`, `err_div_zero0`, `err_overflow0` and `result_valid0`. No opcode decoding is required to read this symbol.

Port	Width	Direction	Function
`num_a`	N	in	First operand
`num_b`	N	in	Second operand (unused by `op_sqrt` and `op_abs`)
`op_add` … `op_abs`	1 each	in	Eight operation-select pins; assert exactly one
`result`	N	out	Result of the selected operation (low word of a product)
`mul_high`	N	out	High word of the product; zero for every other operation
`err_div_zero`	1	out	Division or modulo attempted with `num_b` $= 0$
`err_overflow`	1	out	Result did not fit the output bus
`result_valid`	1	out	High only when exactly one operation pin is asserted

Table 3: Port specification. N is 8, 16 or 32 depending on the instantiated variant.

Variant	Input pins	Output pins	Total
`advanced_math_alu_8bit`	24	19	43
`advanced_math_alu_16bit`	40	35	75
`advanced_math_alu_32bit`	72	67	139

Table 4: Flattened pin counts after NgVeri expands the vector ports.

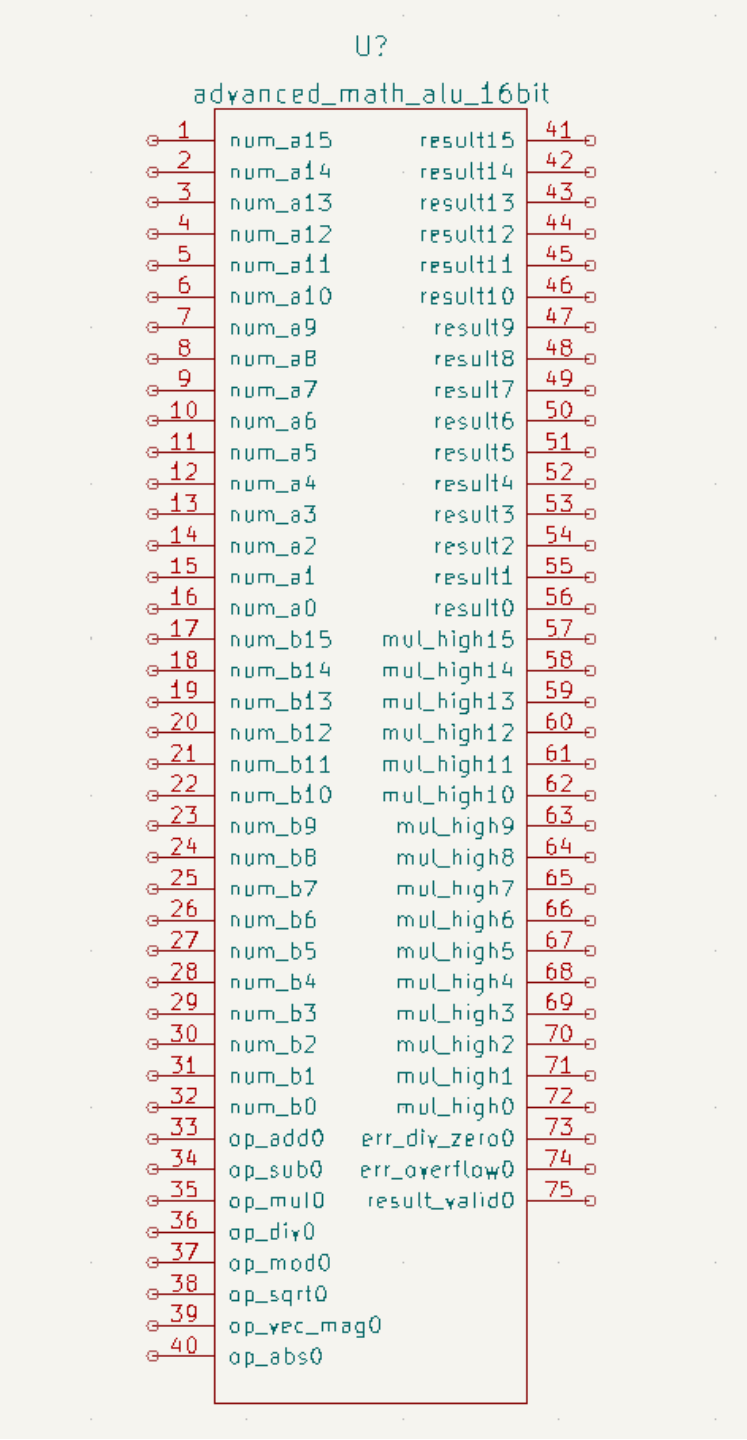
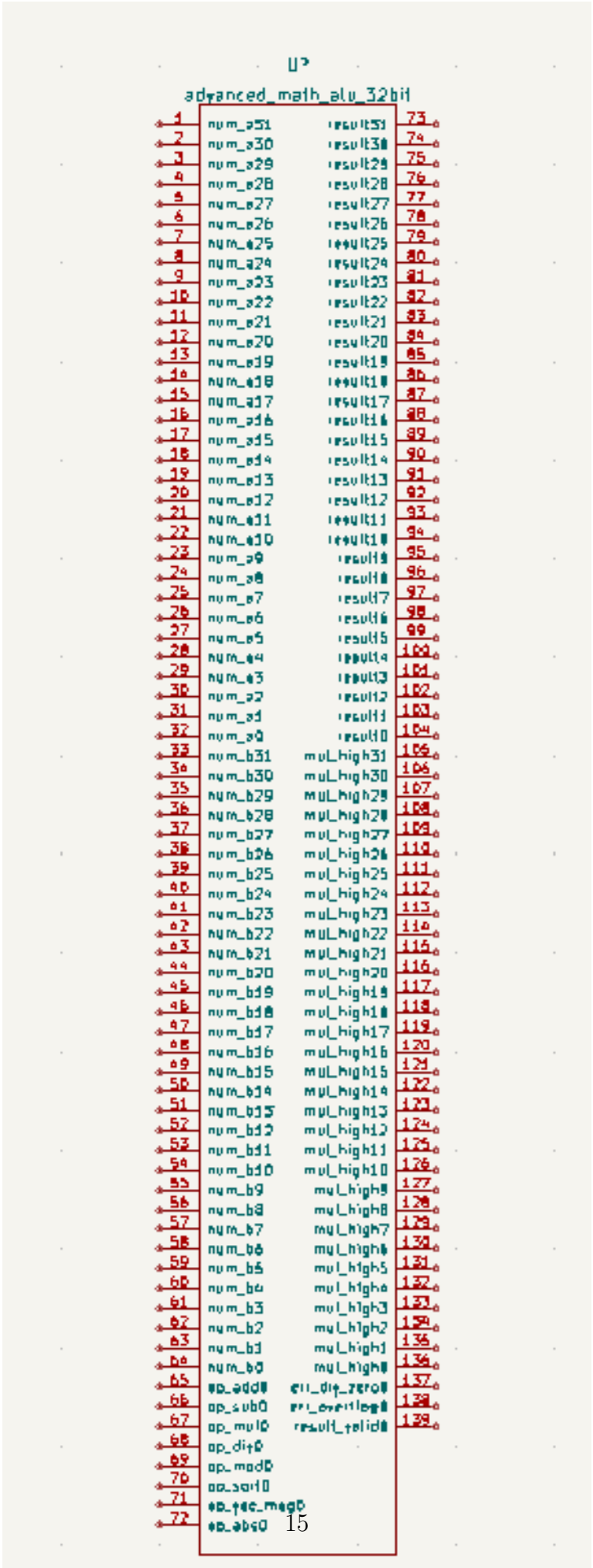


Figure 2: KiCad symbol for the 16-bit variant (75 pins). The port names are identical to the 8-bit block; only the bus widths change.


U?
advanced_math_alu_32bit
num_a31
num_a30
num_a29
num_a28
num_a27
num_a26
num_a25
num_a24
num_a23
num_a22
num_a21
num_a20
num_a19
num_a18
num_a17
num_a16
num_a15
num_a14
num_a13
num_a12
num_a11
num_a10
num_a9
num_a8
num_a7
num_a6
num_a5
num_a4
num_a3
num_a2
num_a1
num_a0
num_b31
num_b30
num_b29
num_b28
num_b27
num_b26
num_b25
num_b24
num_b23
num_b22
num_b21
num_b20
num_b19
num_b18
num_b17
num_b16
num_b15
num_b14
num_b13
num_b12
num_b11
num_b10
num_b9
num_b8
num_b7
num_b6
num_b5
num_b4
num_b3
num_b2
num_b1
num_b0
op_add
op_sub
op_mul
op_div
op_mod
op_sqrt
op_fac_msg
op_abs
result31
result30
result29
result28
result27
result26
result25
result24
result23
result22
result21
result20
result19
result18
result17
result16
result15
result14
result13
result12
result11
result10
result9
result8
result7
result6
result5
result4
result3
result2
result1
result0
mul_high31
mul_high30
mul_high29
mul_high28
mul_high27
mul_high26
mul_high25
mul_high24
mul_high23
mul_high22
mul_high21
mul_high20
mul_high19
mul_high18
mul_high17
mul_high16
mul_high15
mul_high14
mul_high13
mul_high12
mul_high11
mul_high10
mul_high9
mul_high8
mul_high7
mul_high6
mul_high5
mul_high4
mul_high3
mul_high2
mul_high1
mul_high0
div_by_zero
result_valid
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139

3.4 Operation Set

Pin asserted	Operation	Result	Notes
op_add	Addition	result = num_a + num_b	err_overflow on carry out
op_sub	Subtraction	result = num_a − num_b	err_overflow on borrow
op_mul	Multiplication	{mul_high, result} = num_a × num_b	full $2N$-bit product
op_div	Integer division	result = num_a / num_b	err_div_zero if num_b = 0
op_mod	Modulo	result = num_a % num_b	err_div_zero if num_b = 0
op_sqrt	Square root	result = $\lfloor\sqrt{\texttt{num\_a}}\rfloor$	num_b ignored
op_vec_mag	Vector magnitude	result = $\lfloor\sqrt{\texttt{num\_a}^2 + \texttt{num\_b}^2}\rfloor$	saturates with err_overflow
op_abs	Absolute value	result = $\lvert\texttt{num\_a}\rvert$	num_a read as two's complement

Table 5: The eight operations, each selected by its own pin. Arithmetic is unsigned except for op_abs.

3.5 Handling of an Invalid Selection

Because the operation is chosen by eight independent pins rather than an encoded field, two conditions exist that an opcode bus cannot express: **no pin asserted** and **more than one pin asserted**. Leaving either undefined would reintroduce exactly the silent-error problem the interface was designed to remove — an unconnected ADC bridge channel is an easy mistake in a large schematic.

The design therefore counts the asserted pins explicitly:

$$\texttt{result\_valid} = (\text{number of asserted } \texttt{op\_*} \text{ pins} = 1)$$

and evaluates an operation **only** when that condition holds. If the selection is invalid, result, mul_high and both error flags read zero and result_valid goes low. A user watching result_valid therefore sees immediately that the block is not being driven correctly, instead of reading a plausible but meaningless number. A pleasant side effect is that the priority order of the internal if/else if chain becomes irrelevant: it is only ever entered when exactly one condition can be true.

3.6 Arithmetic Implementation

3.6.1 Full-Width Multiplication

An $N \times N$ product needs $2N$ bits. Rather than truncate, the design computes the full product internally and presents it as two buses: result carries the low word and mul_high the high word. err_overflow is asserted whenever mul_high is non-zero, so a user who only wires result still receives an explicit warning that the product did not fit.

3.6.2 Square Root by Shift-and-Subtract

Verilog has no synthesisable square-root operator, so the root is computed by an **unrolled restoring shift-and-subtract** algorithm. The target is held in a $2N$-bit register and processed two bits per iteration over N iterations, which yields an N-bit root:

1. shift the running remainder left by two and append the next bit-pair of the target;
2. form a trial divisor from the root so far;

3. if the trial divisor fits, subtract it and append a `1` to the root, otherwise append a `0`.

The loop is fully unrolled inside a combinational `always @(*)` block, so the result appears with **zero latency** — no clock, no handshake and no sequential control for the eSim user to provide.

3.6.3 Vector Magnitude and Saturation

For `op_vec_mag` the square-root operand is $\texttt{num\_a}^2 + \texttt{num\_b}^2$. That sum can need $2N+1$ bits: for 8-bit operands the worst case is $255^2 + 255^2 = 130\,050$, which exceeds the $2^{16} - 1$ a $2N$-bit register can hold. The sum is therefore accumulated in a $2N+1$-bit register and tested for overflow. If it overflows, the square-root operand is **saturated** to all ones and `err_overflow` is asserted, so the block returns the largest representable magnitude together with an explicit warning rather than a silently wrapped value.

3.6.4 Error and Overflow Reporting

Condition	Reported by
Carry out of an addition	`err_overflow`
Borrow in a subtraction (`num_a` $<$ `num_b`)	`err_overflow`
Product wider than `result`	`err_overflow` (and `mul_high` $\neq 0$)
Vector magnitude saturated	`err_overflow`
Division or modulo with `num_b` $= 0$	`err_div_zero`
No operation, or several, selected	`result_valid` low

Table 6: Every abnormal condition is reported on a dedicated output.

3.7 RTL Source Code

The three variants are structurally identical and differ only in their declared widths and in the iteration count of the square-root loop.

Listing 2: Advanced Math ALU, 8-bit variant

```
`timescale 1ns / 1ps

// ----------------------------------------------------------------------
//  Advanced Math ALU - 8-bit
//
//  Operation select: assert EXACTLY ONE op_* input.  The selected
//  operation is named by its own pin, so no opcode table is required.
//
//    op_add      result = num_a + num_b
//    op_sub      result = num_a - num_b
//    op_mul      {mul_high, result} = num_a * num_b   (full product)
//    op_div      result = num_a / num_b               (integer quotient)
//    op_mod      result = num_a % num_b                (remainder)
//    op_sqrt     result = floor(sqrt(num_a))          (num_b unused)
//    op_vec_mag  result = floor(sqrt(num_a^2 + num_b^2))
//    op_abs      result = |num_a| , num_a read as two's complement
//
//  If no op_* pin or more than one op_* pin is asserted, result_valid
//  goes low and every data output reads zero.
//
//  Arithmetic is unsigned except op_abs, which reads num_a as signed.
```

```
// ---------------------------------------------------------------------

module advanced_math_alu_8bit (
    input  wire [7:0] num_a,
    input  wire [7:0] num_b,
    input  wire op_add,
    input  wire op_sub,
    input  wire op_mul,
    input  wire op_div,
    input  wire op_mod,
    input  wire op_sqrt,
    input  wire op_vec_mag,
    input  wire op_abs,
    output wire [7:0] result,
    output wire [7:0] mul_high,
    output wire err_div_zero,
    output wire err_overflow,
    output wire result_valid
);

    reg [7:0] result_reg;
    reg [7:0] mul_high_reg;
    reg       err_div_zero_reg;
    reg       err_overflow_reg;
    reg       result_valid_reg;

    // exactly-one-hot detector
    wire [3:0] op_count = {3'b000, op_add}     + {3'b000, op_sub}
                        + {3'b000, op_mul}     + {3'b000, op_div}
                        + {3'b000, op_mod}     + {3'b000, op_sqrt}
                        + {3'b000, op_vec_mag} + {3'b000, op_abs};

    reg [8:0] add_tmp;        // one extra bit for the carry out
    reg [15:0] mul_full;       // full double-width product
    reg [16:0] sq_sum;         // num_a^2 + num_b^2, one spare bit
    reg [15:0] sqrt_target;
    reg [15:0] sqrt_root;
    reg [15:0] sqrt_rem;
    reg [15:0] sqrt_test;
    integer i;

    always @(*) begin
        result_reg       = 8'd0;
        mul_high_reg     = 8'd0;
        err_div_zero_reg = 1'b0;
        err_overflow_reg = 1'b0;
        add_tmp          = 9'd0;
        mul_full         = 16'd0;
        sq_sum           = 17'd0;
        sqrt_target      = 16'd0;
        sqrt_root        = 16'd0;
        sqrt_rem         = 16'd0;
        sqrt_test        = 16'd0;

        result_valid_reg = (op_count == 4'd1);

        // ---- build the square-root operand ----
        if (op_sqrt) begin
            sqrt_target = {{8{1'b0}}, num_a};
        end else if (op_vec_mag) begin
            sq_sum = (num_a * num_a) + (num_b * num_b);
            if (sq_sum[16]) begin
                sqrt_target      = {16{1'b1}};   // saturate
                err_overflow_reg = 1'b1;
            end else begin
                sqrt_target = sq_sum[15:0];
            end
        end

        // ---- unrolled shift-and-subtract square root ----
        for (i = 7; i >= 0; i = i - 1) begin
            sqrt_rem  = (sqrt_rem << 2) | ((sqrt_target >> (i * 2)) & 16'd3);
            sqrt_test = (sqrt_root << 2) | 16'd1;
```

```
            if (sqrt_rem >= sqrt_test) begin
                sqrt_rem  = sqrt_rem - sqrt_test;
                sqrt_root = (sqrt_root << 1) | 16'd1;
            end else begin
                sqrt_root = (sqrt_root << 1);
            end
        end

        // ---- one operation, selected by its own pin ----
        if (result_valid_reg) begin
            if (op_add) begin
                add_tmp          = {1'b0, num_a} + {1'b0, num_b};
                result_reg       = add_tmp[7:0];
                err_overflow_reg = add_tmp[8];
            end else if (op_sub) begin
                result_reg       = num_a - num_b;
                err_overflow_reg = (num_a < num_b);   // unsigned borrow
            end else if (op_mul) begin
                mul_full         = num_a * num_b;
                result_reg       = mul_full[7:0];
                mul_high_reg     = mul_full[15:8];
                err_overflow_reg = (mul_full[15:8] != 8'd0);
            end else if (op_div) begin
                if (num_b == 8'd0) err_div_zero_reg = 1'b1;
                else               result_reg       = num_a / num_b;
            end else if (op_mod) begin
                if (num_b == 8'd0) err_div_zero_reg = 1'b1;
                else               result_reg       = num_a % num_b;
            end else if (op_sqrt) begin
                result_reg = sqrt_root[7:0];
            end else if (op_vec_mag) begin
                result_reg = sqrt_root[7:0];
            end else if (op_abs) begin
                result_reg = (num_a[7]) ? (~num_a + 8'd1) : num_a;
            end
        end
    end

    assign result       = result_reg;
    assign mul_high     = mul_high_reg;
    assign err_div_zero = err_div_zero_reg;
    assign err_overflow = err_overflow_reg;
    assign result_valid = result_valid_reg;

endmodule
```

Listing 3: Advanced Math ALU, 16-bit variant

```
`timescale 1ns / 1ps

// ------------------------------------------------------------------------
//  Advanced Math ALU - 16-bit
//
//  Operation select: assert EXACTLY ONE op_* input.  The selected
//  operation is named by its own pin, so no opcode table is required.
//
//    op_add      result = num_a + num_b
//    op_sub      result = num_a - num_b
//    op_mul      {mul_high, result} = num_a * num_b   (full product)
//    op_div      result = num_a / num_b               (integer quotient)
//    op_mod      result = num_a % num_b                (remainder)
//    op_sqrt     result = floor(sqrt(num_a))          (num_b unused)
//    op_vec_mag  result = floor(sqrt(num_a^2 + num_b^2))
//    op_abs      result = |num_a| , num_a read as two's complement
//
//  If no op_* pin or more than one op_* pin is asserted, result_valid
//  goes low and every data output reads zero.
//
//  Arithmetic is unsigned except op_abs, which reads num_a as signed.
// ------------------------------------------------------------------------

module advanced_math_alu_16bit (
    input  wire [15:0] num_a,
```

```
    input  wire [15:0] num_b,
    input  wire  op_add,
    input  wire  op_sub,
    input  wire  op_mul,
    input  wire  op_div,
    input  wire  op_mod,
    input  wire  op_sqrt,
    input  wire  op_vec_mag,
    input  wire  op_abs,
    output wire [15:0] result,
    output wire [15:0] mul_high,
    output wire  err_div_zero,
    output wire  err_overflow,
    output wire  result_valid
);

    reg [15:0] result_reg;
    reg [15:0] mul_high_reg;
    reg       err_div_zero_reg;
    reg       err_overflow_reg;
    reg       result_valid_reg;

    // exactly-one-hot detector
    wire [3:0] op_count = {3'b000, op_add}     + {3'b000, op_sub}
                        + {3'b000, op_mul}     + {3'b000, op_div}
                        + {3'b000, op_mod}     + {3'b000, op_sqrt}
                        + {3'b000, op_vec_mag} + {3'b000, op_abs};

    reg [16:0] add_tmp;        // one extra bit for the carry out
    reg [31:0] mul_full;       // full double-width product
    reg [32:0] sq_sum;         // num_a^2 + num_b^2, one spare bit
    reg [31:0] sqrt_target;
    reg [31:0] sqrt_root;
    reg [31:0] sqrt_rem;
    reg [31:0] sqrt_test;
    integer i;

    always @(*) begin
        result_reg       = 16'd0;
        mul_high_reg     = 16'd0;
        err_div_zero_reg = 1'b0;
        err_overflow_reg = 1'b0;
        add_tmp          = 17'd0;
        mul_full         = 32'd0;
        sq_sum           = 33'd0;
        sqrt_target      = 32'd0;
        sqrt_root        = 32'd0;
        sqrt_rem         = 32'd0;
        sqrt_test        = 32'd0;

        result_valid_reg = (op_count == 4'd1);

        // ---- build the square-root operand ----
        if (op_sqrt) begin
            sqrt_target = {{16{1'b0}}, num_a};
        end else if (op_vec_mag) begin
            sq_sum = (num_a * num_a) + (num_b * num_b);
            if (sq_sum[32]) begin
                sqrt_target      = {32{1'b1}};   // saturate
                err_overflow_reg = 1'b1;
            end else begin
                sqrt_target = sq_sum[31:0];
            end
        end

        // ---- unrolled shift-and-subtract square root ----
        for (i = 15; i >= 0; i = i - 1) begin
            sqrt_rem  = (sqrt_rem << 2) | ((sqrt_target >> (i * 2)) & 32'd3);
            sqrt_test = (sqrt_root << 2) | 32'd1;
            if (sqrt_rem >= sqrt_test) begin
                sqrt_rem  = sqrt_rem - sqrt_test;
                sqrt_root = (sqrt_root << 1) | 32'd1;
            end else begin
```

```
                sqrt_root = (sqrt_root << 1);
            end
        end

        // ---- one operation, selected by its own pin ----
        if (result_valid_reg) begin
            if (op_add) begin
                add_tmp          = {1'b0, num_a} + {1'b0, num_b};
                result_reg       = add_tmp[15:0];
                err_overflow_reg = add_tmp[16];
            end else if (op_sub) begin
                result_reg       = num_a - num_b;
                err_overflow_reg = (num_a < num_b);   // unsigned borrow
            end else if (op_mul) begin
                mul_full         = num_a * num_b;
                result_reg       = mul_full[15:0];
                mul_high_reg     = mul_full[31:16];
                err_overflow_reg = (mul_full[31:16] != 16'd0);
            end else if (op_div) begin
                if (num_b == 16'd0) err_div_zero_reg = 1'b1;
                else                result_reg       = num_a / num_b;
            end else if (op_mod) begin
                if (num_b == 16'd0) err_div_zero_reg = 1'b1;
                else                result_reg       = num_a % num_b;
            end else if (op_sqrt) begin
                result_reg = sqrt_root[15:0];
            end else if (op_vec_mag) begin
                result_reg = sqrt_root[15:0];
            end else if (op_abs) begin
                result_reg = (num_a[15]) ? (~num_a + 16'd1) : num_a;
            end
        end
    end

    assign result       = result_reg;
    assign mul_high     = mul_high_reg;
    assign err_div_zero = err_div_zero_reg;
    assign err_overflow = err_overflow_reg;
    assign result_valid = result_valid_reg;

endmodule
```

Listing 4: Advanced Math ALU, 32-bit variant

```
`timescale 1ns / 1ps

// ------------------------------------------------------------------
//  Advanced Math ALU - 32-bit
//
//  Operation select: assert EXACTLY ONE op_* input.  The selected
//  operation is named by its own pin, so no opcode table is required.
//
//    op_add      result = num_a + num_b
//    op_sub      result = num_a - num_b
//    op_mul      {mul_high, result} = num_a * num_b   (full product)
//    op_div      result = num_a / num_b               (integer quotient)
//    op_mod      result = num_a % num_b                (remainder)
//    op_sqrt     result = floor(sqrt(num_a))          (num_b unused)
//    op_vec_mag  result = floor(sqrt(num_a^2 + num_b^2))
//    op_abs      result = |num_a| , num_a read as two's complement
//
//  If no op_* pin or more than one op_* pin is asserted, result_valid
//  goes low and every data output reads zero.
//
//  Arithmetic is unsigned except op_abs, which reads num_a as signed.
// ------------------------------------------------------------------

module advanced_math_alu_32bit (
    input  wire [31:0] num_a,
    input  wire [31:0] num_b,
    input  wire  op_add,
    input  wire  op_sub,
    input  wire  op_mul,
```

```
    input  wire  op_div,
    input  wire  op_mod,
    input  wire  op_sqrt,
    input  wire  op_vec_mag,
    input  wire  op_abs,
    output wire [31:0] result,
    output wire [31:0] mul_high,
    output wire  err_div_zero,
    output wire  err_overflow,
    output wire  result_valid
);

    reg [31:0] result_reg;
    reg [31:0] mul_high_reg;
    reg       err_div_zero_reg;
    reg       err_overflow_reg;
    reg       result_valid_reg;

    // exactly-one-hot detector
    wire [3:0] op_count = {3'b000, op_add}     + {3'b000, op_sub}
                        + {3'b000, op_mul}     + {3'b000, op_div}
                        + {3'b000, op_mod}     + {3'b000, op_sqrt}
                        + {3'b000, op_vec_mag} + {3'b000, op_abs};

    reg [32:0] add_tmp;        // one extra bit for the carry out
    reg [63:0] mul_full;       // full double-width product
    reg [64:0] sq_sum;         // num_a^2 + num_b^2, one spare bit
    reg [63:0] sqrt_target;
    reg [63:0] sqrt_root;
    reg [63:0] sqrt_rem;
    reg [63:0] sqrt_test;
    integer i;

    always @(*) begin
        result_reg       = 32'd0;
        mul_high_reg     = 32'd0;
        err_div_zero_reg = 1'b0;
        err_overflow_reg = 1'b0;
        add_tmp          = 33'd0;
        mul_full         = 64'd0;
        sq_sum           = 65'd0;
        sqrt_target      = 64'd0;
        sqrt_root        = 64'd0;
        sqrt_rem         = 64'd0;
        sqrt_test        = 64'd0;

        result_valid_reg = (op_count == 4'd1);

        // ---- build the square-root operand ----
        if (op_sqrt) begin
            sqrt_target = {{32{1'b0}}, num_a};
        end else if (op_vec_mag) begin
            sq_sum = (num_a * num_a) + (num_b * num_b);
            if (sq_sum[64]) begin
                sqrt_target      = {64{1'b1}};   // saturate
                err_overflow_reg = 1'b1;
            end else begin
                sqrt_target = sq_sum[63:0];
            end
        end

        // ---- unrolled shift-and-subtract square root ----
        for (i = 31; i >= 0; i = i - 1) begin
            sqrt_rem  = (sqrt_rem << 2) | ((sqrt_target >> (i * 2)) & 64'd3);
            sqrt_test = (sqrt_root << 2) | 64'd1;
            if (sqrt_rem >= sqrt_test) begin
                sqrt_rem  = sqrt_rem - sqrt_test;
                sqrt_root = (sqrt_root << 1) | 64'd1;
            end else begin
                sqrt_root = (sqrt_root << 1);
            end
        end

```

```
        // ---- one operation, selected by its own pin ----
        if (result_valid_reg) begin
            if (op_add) begin
                add_tmp          = {1'b0, num_a} + {1'b0, num_b};
                result_reg       = add_tmp[31:0];
                err_overflow_reg = add_tmp[32];
            end else if (op_sub) begin
                result_reg       = num_a - num_b;
                err_overflow_reg = (num_a < num_b);   // unsigned borrow
            end else if (op_mul) begin
                mul_full         = num_a * num_b;
                result_reg       = mul_full[31:0];
                mul_high_reg     = mul_full[63:32];
                err_overflow_reg = (mul_full[63:32] != 32'd0);
            end else if (op_div) begin
                if (num_b == 32'd0) err_div_zero_reg = 1'b1;
                else                result_reg       = num_a / num_b;
            end else if (op_mod) begin
                if (num_b == 32'd0) err_div_zero_reg = 1'b1;
                else                result_reg       = num_a % num_b;
            end else if (op_sqrt) begin
                result_reg = sqrt_root[31:0];
            end else if (op_vec_mag) begin
                result_reg = sqrt_root[31:0];
            end else if (op_abs) begin
                result_reg = (num_a[31]) ? (~num_a + 32'd1) : num_a;
            end
        end
    end

    assign result       = result_reg;
    assign mul_high     = mul_high_reg;
    assign err_div_zero = err_div_zero_reg;
    assign err_overflow = err_overflow_reg;
    assign result_valid = result_valid_reg;

endmodule
```

3.8 Master Testbench

A single testbench instantiates all three widths and drives them from one shared 32-bit stimulus bus, so every operation is checked at every width in one run. Each DUT takes only the low bits of the stimulus, which gives a useful property: an **all-ones operand automatically presents each variant with its own maximum value**, allowing the carry, borrow, full-width product and saturation cases to be expressed as a single vector.

Listing 5: Master testbench covering the 8, 16 and 32-bit variants

```
`timescale 1ns / 1ps

// ------------------------------------------------------------------------
//  Master testbench for the Advanced Math ALU.
//
//  All three widths (8 / 16 / 32-bit) are instantiated together and
//  driven from one shared stimulus bus, so every operation is checked
//  across every width in a single run.  Each DUT takes only the low
//  bits of the stimulus, which means an all-ones operand automatically
//  presents each width with its own maximum value.
//
//  Results are printed to the console as one combined table.
// ------------------------------------------------------------------------

module tb_advanced_math_alu_master;

    reg [31:0] a_bus;
    reg [31:0] b_bus;
    reg [7:0]  op_bus;   // {abs, vec_mag, sqrt, mod, div, mul, sub, add}

```

```
    wire [7:0] r8;
    wire [7:0] h8;
    wire       d8, o8, v8;
    wire [15:0] r16;
    wire [15:0] h16;
    wire       d16, o16, v16;
    wire [31:0] r32;
    wire [31:0] h32;
    wire       d32, o32, v32;

    integer pass_cnt = 0;
    integer fail_cnt = 0;

    advanced_math_alu_8bit u8 (
        .num_a(a_bus[7:0]), .num_b(b_bus[7:0]),
        .op_add(op_bus[0]), .op_sub(op_bus[1]),
        .op_mul(op_bus[2]), .op_div(op_bus[3]),
        .op_mod(op_bus[4]), .op_sqrt(op_bus[5]),
        .op_vec_mag(op_bus[6]), .op_abs(op_bus[7]),
        .result(r8), .mul_high(h8),
        .err_div_zero(d8), .err_overflow(o8), .result_valid(v8)
    );

    advanced_math_alu_16bit u16 (
        .num_a(a_bus[15:0]), .num_b(b_bus[15:0]),
        .op_add(op_bus[0]), .op_sub(op_bus[1]),
        .op_mul(op_bus[2]), .op_div(op_bus[3]),
        .op_mod(op_bus[4]), .op_sqrt(op_bus[5]),
        .op_vec_mag(op_bus[6]), .op_abs(op_bus[7]),
        .result(r16), .mul_high(h16),
        .err_div_zero(d16), .err_overflow(o16), .result_valid(v16)
    );

    advanced_math_alu_32bit u32 (
        .num_a(a_bus[31:0]), .num_b(b_bus[31:0]),
        .op_add(op_bus[0]), .op_sub(op_bus[1]),
        .op_mul(op_bus[2]), .op_div(op_bus[3]),
        .op_mod(op_bus[4]), .op_sqrt(op_bus[5]),
        .op_vec_mag(op_bus[6]), .op_abs(op_bus[7]),
        .result(r32), .mul_high(h32),
        .err_div_zero(d32), .err_overflow(o32), .result_valid(v32)
    );

    task run_all;
        input [8*22:1] test_name;
        input [7:0]  ops;
        input [31:0] a;
        input [31:0] b;
        input [31:0] e_r8;
        input [31:0] e_r16;
        input [31:0] e_r32;
        input [31:0] e_h8;
        input [31:0] e_h16;
        input [31:0] e_h32;
        input        e_d0;
        input        e_ov;
        input        e_vl;
        reg ok8, ok16, ok32, all_ok;
        begin
            a_bus  = a;
            b_bus  = b;
            op_bus = ops;
            #10;
            ok8  = (r8  === e_r8[7:0])   && (h8  === e_h8[7:0])
                && (d8  === e_d0) && (o8  === e_ov) && (v8  === e_vl);
            ok16 = (r16 === e_r16[15:0]) && (h16 === e_h16[15:0])
                && (d16 === e_d0) && (o16 === e_ov) && (v16 === e_vl);
            ok32 = (r32 === e_r32)       && (h32 === e_h32)
                && (d32 === e_d0) && (o32 === e_ov) && (v32 === e_vl);
            all_ok = ok8 & ok16 & ok32;
            $display("| %-22s |  %02h  |  %04h  |  %08h  |  %b%b%b   | %-3s | %-3s | %-3s | %-4s |",
                     test_name, r8, r16, r32, e_d0, e_ov, e_vl,
                     ok8  ? "ok" : "ERR", ok16 ? "ok" : "ERR",
```

```
                    ok32 ? "ok" : "ERR", all_ok ? "PASS" : "FAIL");
            if (all_ok) pass_cnt = pass_cnt + 1;
            else        fail_cnt = fail_cnt + 1;
        end
    endtask

    initial begin
        $dumpfile("sim_out.vcd");
        $dumpvars(0, tb_advanced_math_alu_master);

        a_bus = 32'd0; b_bus = 32'd0; op_bus = 8'b0;

        $display("\n================================================================================
    ======");
        $display(" ADVANCED MATH ALU MASTER VERIFICATION SUITE - 8 / 16 / 32-BIT");
        $display(" One operation pin per function. Flags: d0 = div-by-zero, ov = overflow, vl = valid."
    );
        $display("================================================================================
    ====");
        $display("| Test Name              | 8-bit | 16-bit |   32-bit   | d0/ov/vl | 8   | 16  | 32  |
    Stat |");
        $display("|------------------------|-------|--------|------------|----------|-----|-----|-----|
    ------|");

        run_all("TC1: add 100+50", 8'b0000_0001, 32'h00000064, 32'h00000032,
                32'h00000096, 32'h00000096, 32'h00000096,
                32'h00000000, 32'h00000000, 32'h00000000, 0, 0, 1);
        run_all("TC2: add carry out", 8'b0000_0001, 32'hFFFFFFFF, 32'h00000001,
                32'h00000000, 32'h00000000, 32'h00000000,
                32'h00000000, 32'h00000000, 32'h00000000, 0, 1, 1);
        run_all("TC3: sub 100-50", 8'b0000_0010, 32'h00000064, 32'h00000032,
                32'h00000032, 32'h00000032, 32'h00000032,
                32'h00000000, 32'h00000000, 32'h00000000, 0, 0, 1);
        run_all("TC4: sub borrow", 8'b0000_0010, 32'h00000032, 32'h00000064,
                32'h000000CE, 32'h0000FFCE, 32'hFFFFFFCE,
                32'h00000000, 32'h00000000, 32'h00000000, 0, 1, 1);
        run_all("TC5: mul 12*12", 8'b0000_0100, 32'h0000000C, 32'h0000000C,
                32'h00000090, 32'h00000090, 32'h00000090,
                32'h00000000, 32'h00000000, 32'h00000000, 0, 0, 1);
        run_all("TC6: mul full width", 8'b0000_0100, 32'hFFFFFFFF, 32'hFFFFFFFF,
                32'h00000001, 32'h00000001, 32'h00000001,
                32'h000000FE, 32'h0000FFFE, 32'hFFFFFFFE, 0, 1, 1);
        run_all("TC7: div 100/7", 8'b0000_1000, 32'h00000064, 32'h00000007,
                32'h0000000E, 32'h0000000E, 32'h0000000E,
                32'h00000000, 32'h00000000, 32'h00000000, 0, 0, 1);
        run_all("TC8: div by zero", 8'b0000_1000, 32'h00000064, 32'h00000000,
                32'h00000000, 32'h00000000, 32'h00000000,
                32'h00000000, 32'h00000000, 32'h00000000, 1, 0, 1);
        run_all("TC9: mod 100%7", 8'b0001_0000, 32'h00000064, 32'h00000007,
                32'h00000002, 32'h00000002, 32'h00000002,
                32'h00000000, 32'h00000000, 32'h00000000, 0, 0, 1);
        run_all("TC10: mod by zero", 8'b0001_0000, 32'h00000064, 32'h00000000,
                32'h00000000, 32'h00000000, 32'h00000000,
                32'h00000000, 32'h00000000, 32'h00000000, 1, 0, 1);
        run_all("TC11: sqrt 144", 8'b0010_0000, 32'h00000090, 32'h00000000,
                32'h0000000C, 32'h0000000C, 32'h0000000C,
                32'h00000000, 32'h00000000, 32'h00000000, 0, 0, 1);
        run_all("TC12: sqrt 200 (floor)", 8'b0010_0000, 32'h000000C8, 32'h00000000,
                32'h0000000E, 32'h0000000E, 32'h0000000E,
                32'h00000000, 32'h00000000, 32'h00000000, 0, 0, 1);
        run_all("TC13: vec_mag 3,4", 8'b0100_0000, 32'h00000003, 32'h00000004,
                32'h00000005, 32'h00000005, 32'h00000005,
                32'h00000000, 32'h00000000, 32'h00000000, 0, 0, 1);
        run_all("TC14: vec_mag 5,12", 8'b0100_0000, 32'h00000005, 32'h0000000C,
                32'h0000000D, 32'h0000000D, 32'h0000000D,
                32'h00000000, 32'h00000000, 32'h00000000, 0, 0, 1);
        run_all("TC15: vec_mag saturate", 8'b0100_0000, 32'hFFFFFFFF, 32'hFFFFFFFF,
                32'h000000FF, 32'h0000FFFF, 32'hFFFFFFFF,
                32'h00000000, 32'h00000000, 32'h00000000, 0, 1, 1);
        run_all("TC16: abs positive", 8'b1000_0000, 32'h00000064, 32'h00000000,
                32'h00000064, 32'h00000064, 32'h00000064,
                32'h00000000, 32'h00000000, 32'h00000000, 0, 0, 1);
        run_all("TC17: abs of -1", 8'b1000_0000, 32'hFFFFFFFF, 32'h00000000,
```

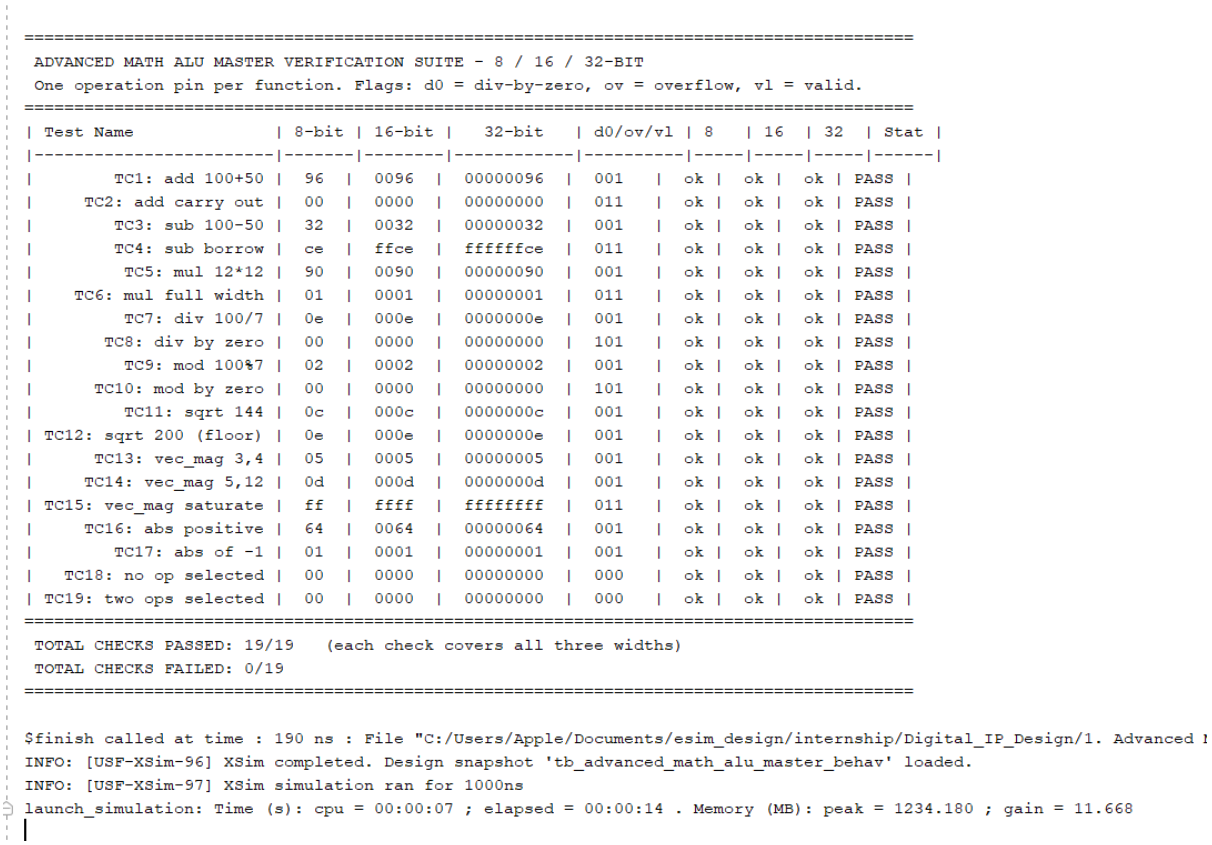
```
                32'h00000001, 32'h00000001, 32'h00000001,
                32'h00000000, 32'h00000000, 32'h00000000, 0, 0, 1);
        run_all("TC18: no op selected", 8'b0000_0000, 32'h00000064, 32'h00000032,
                32'h00000000, 32'h00000000, 32'h00000000,
                32'h00000000, 32'h00000000, 32'h00000000, 0, 0, 0);
        run_all("TC19: two ops selected", 8'b0000_0011, 32'h00000064, 32'h00000032,
                32'h00000000, 32'h00000000, 32'h00000000,
                32'h00000000, 32'h00000000, 32'h00000000, 0, 0, 0);

        $display("================================================================================
    ↪ ====");
        $display(" TOTAL CHECKS PASSED: %0d/%0d   (each check covers all three widths)",
                 pass_cnt, pass_cnt + fail_cnt);
        $display(" TOTAL CHECKS FAILED: %0d/%0d", fail_cnt, pass_cnt + fail_cnt);
        $display("================================================================================
    ↪ ====\n");
        $finish;
     end

endmodule
```

3.9 Verification Phase 1: Pure Digital (Vivado XSim)

Nineteen directed checks were applied. A check is recorded as `PASS` only if **all three widths** agree with the expectation on the result, the high word and all three status flags.

Checks	Property verified	Expectation
TC1, TC3, TC5, TC7, TC9	basic arithmetic	add, subtract, multiply, divide, modulo of small operands
TC2	addition carry out	`MAX` $+1$ wraps to 0 with `err_overflow`
TC4	subtraction borrow	$50 - 100$ wraps with `err_overflow`
TC6	full-width product	`MAX` $\times$ `MAX` splits across `mul_high` and `result`
TC8, TC10	divide/modulo by zero	`err_div_zero` asserted, `result` forced to 0
TC11, TC12	square root	exact ($\sqrt{144} = 12$) and floored ($\lfloor\sqrt{200}\rfloor = 14$)
TC13, TC14	vector magnitude	Pythagorean triples $3, 4 \to 5$ and $5, 12 \to 13$
TC15	vector-magnitude saturation	`MAX`,`MAX` saturates with `err_overflow`
TC16, TC17	absolute value	positive passes through; all-ones (-1) returns 1
TC18	no operation selected	`result_valid` low, outputs zero
TC19	two operations selected	`result_valid` low, outputs zero

Table 7: Coverage of the nineteen master-testbench checks.



```
================================================================================
 ADVANCED MATH ALU MASTER VERIFICATION SUITE - 8 / 16 / 32-BIT
 One operation pin per function. Flags: d0 = div-by-zero, ov = overflow, vl = valid.
================================================================================
| Test Name              | 8-bit | 16-bit |   32-bit   | d0/ov/vl | 8   | 16  | 32  | Stat |
|------------------------|-------|--------|------------|----------|-----|-----|-----|------|
|        TC1: add 100+50 |  96   |  0096  |  00000096  |   001    |  ok |  ok |  ok | PASS |
|     TC2: add carry out |  00   |  0000  |  00000000  |   011    |  ok |  ok |  ok | PASS |
|        TC3: sub 100-50 |  32   |  0032  |  00000032  |   001    |  ok |  ok |  ok | PASS |
|        TC4: sub borrow |  ce   |  ffce  |  fffffffe  |   011    |  ok |  ok |  ok | PASS |
|         TC5: mul 12*12 |  90   |  0090  |  00000090  |   001    |  ok |  ok |  ok | PASS |
|    TC6: mul full width |  01   |  0001  |  00000001  |   011    |  ok |  ok |  ok | PASS |
|         TC7: div 100/7 |  0e   |  000e  |  0000000e  |   001    |  ok |  ok |  ok | PASS |
|       TC8: div by zero |  00   |  0000  |  00000000  |   101    |  ok |  ok |  ok | PASS |
|         TC9: mod 100%7 |  02   |  0002  |  00000002  |   001    |  ok |  ok |  ok | PASS |
|      TC10: mod by zero |  00   |  0000  |  00000000  |   101    |  ok |  ok |  ok | PASS |
|         TC11: sqrt 144 |  0c   |  000c  |  0000000c  |   001    |  ok |  ok |  ok | PASS |
| TC12: sqrt 200 (floor) |  0e   |  000e  |  0000000e  |   001    |  ok |  ok |  ok | PASS |
|      TC13: vec_mag 3,4 |  05   |  0005  |  00000005  |   001    |  ok |  ok |  ok | PASS |
|     TC14: vec_mag 5,12 |  0d   |  000d  |  0000000d  |   001    |  ok |  ok |  ok | PASS |
| TC15: vec_mag saturate |  ff   |  ffff  |  ffffffff  |   011    |  ok |  ok |  ok | PASS |
|     TC16: abs positive |  64   |  0064  |  00000064  |   001    |  ok |  ok |  ok | PASS |
|        TC17: abs of -1 |  01   |  0001  |  00000001  |   001    |  ok |  ok |  ok | PASS |
|   TC18: no op selected |  00   |  0000  |  00000000  |   000    |  ok |  ok |  ok | PASS |
| TC19: two ops selected |  00   |  0000  |  00000000  |   000    |  ok |  ok |  ok | PASS |
================================================================================
 TOTAL CHECKS PASSED: 19/19   (each check covers all three widths)
 TOTAL CHECKS FAILED: 0/19
================================================================================

$finish called at time : 190 ns : File "C:/Users/Apple/Documents/esim_design/internship/Digital_IP_Design/1. Advanced I
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_advanced_math_alu_master_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:07 ; elapsed = 00:00:14 . Memory (MB): peak = 1234.180 ; gain = 11.668
```

Figure 4: Vivado Tcl console output for the master testbench. All nineteen checks report `PASS` across the 8, 16 and 32-bit variants simultaneously (**19/19**, 0 failures). Note TC6, where `MAX`×`MAX` returns `01/0001/00000001` in `result` with the remainder in `mul_high`, and TC18/TC19, where an invalid operation selection drives every output to zero with `result_valid` low.

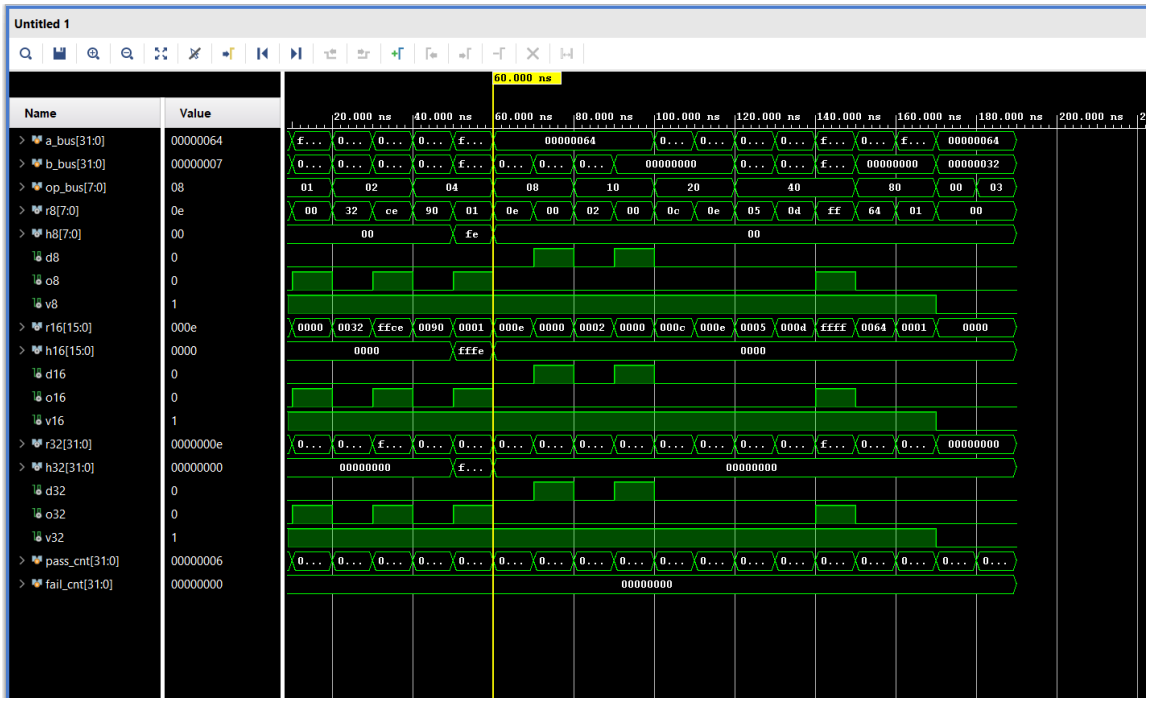


Figure 5: Vivado XSim waveform for the master testbench, showing the operation-select pins changing and the three result buses responding combinationally, with no clock present anywhere in the design.

3.10 Verification Phase 2: Mixed-Signal (eSim / Ngspice)

The mixed-signal test was designed to demonstrate the interface itself: rather than holding one operation and checking a number, it **walks through all eight operations in sequence**, so the plot shows each named pin causing its own operation.

3.10.1 Test Schematic

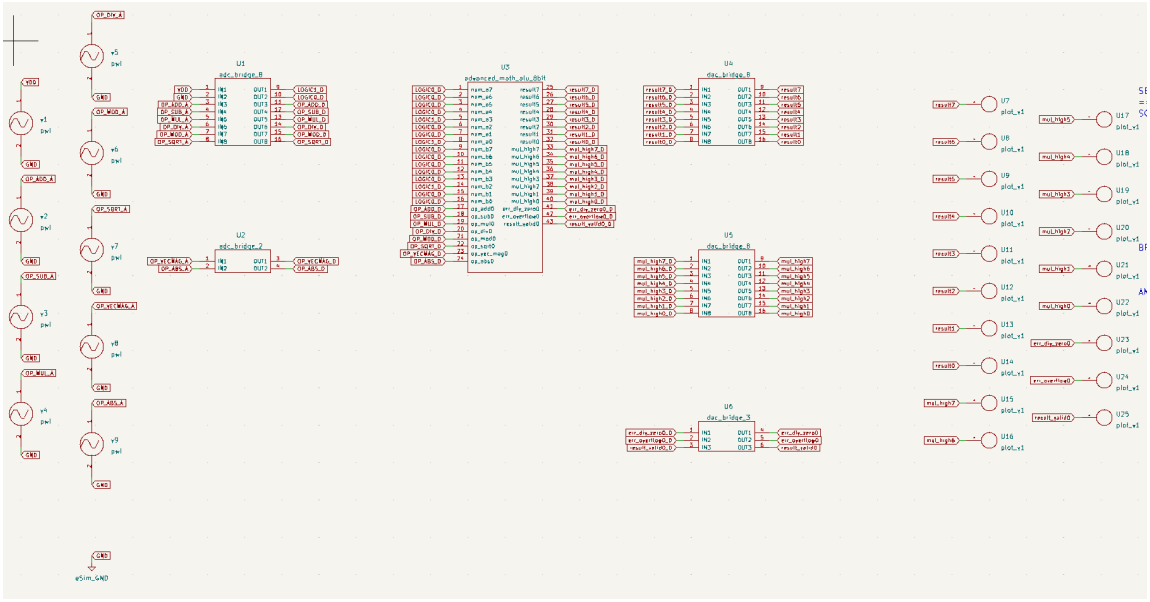


Figure 6: The eSim test schematic for the 8-bit ALU. The operand bits are tied to the shared `VDD`/`GND` rails to form the constants `num_a` = 9 and `num_b` = 4, while each of the eight operation pins is driven by its own `pwl` source through an `adc_bridge`. All nineteen outputs leave through `dac_bridge` banks to individual probes.

Because the operand bits ride the shared logic rails, the analog-to-digital side needs only **ten bridge channels at every width** — eight operation sources plus the two rails. Only the digital-to-analog side scales with the datapath, which is what keeps the 32-bit sheet tractable.

3.10.2 Stimulus and Expected Timeline

Time (ms)	Pin asserted	Operation	Expected `result`	Binary (8-bit)
0–5	`op_add`	$9 + 4$	**13**	00001101
5–10	`op_sub`	$9 - 4$	**5**	00000101
10–15	`op_mul`	9×4	**36**	00100100
15–20	`op_div`	$9/4$	**2**	00000010
20–25	`op_mod`	$9 \bmod 4$	**1**	00000001
25–30	`op_sqrt`	$\sqrt{9}$	**3**	00000011
30–35	`op_vec_mag`	$\lfloor\sqrt{81+16}\rfloor$	**9**	00001001
35–40	`op_abs`	$\lvert 9\rvert$	**9**	00001001
40–45	*(none)*	invalid selection	**0**, `result_valid` low	00000000

Table 8: The 45 ms transient walks every operation pin in turn. Operands are constant at `num_a` = 9, `num_b` = 4; the transient uses a 10 μs step.

The operands were chosen so that every operation returns a **different, easily read value**, which makes the plot self-evident: each change in the result bus is caused by exactly one named pin going high.

3.10.3 Reading the Stacked Plots: Why Many Traces Are Flat

A first look at the stacked output is striking: a large fraction of the traces are perfectly flat. This is **expected and mathematically necessary** for the chosen operands, not a symptom of a fault, and it is worth setting out explicitly because the effect becomes more pronounced at the wider variants.

The nine results produced during the run are $\{13, 5, 36, 2, 1, 3, 9, 9, 0\}$. The bitwise union of those values is `0b101111`, i.e. only bits 0, 1, 2, 3 and 5 are ever set. Every other output is therefore constant by construction:

Traces	State	Reason it cannot change
`result4`, `result6`, `result7`, and all higher bits	low	no result in $\{13, 5, 36, 2, 1, 3, 9, 9\}$ sets those bits
`mul_high` (all bits)	low	$9 \times 4 = 36$ fits the result bus, so the high word is zero
`err_div_zero`	low	`num_b` $= 4$, so division by zero never occurs
`err_overflow`	low	no operation overflows with these operands
`VDD`	high	constant logic-1 rail, by design

Table 9: Every flat trace in the mixed-signal runs, with the reason it is constant.

Counting the captured nodes makes the pattern precise. In all three runs exactly **fourteen** traces toggle — the eight operation stimuli, the five result bits that the arithmetic actually exercises (`result0`, `1`, `2`, `3`, `5`) and `result_valid`. Everything else is constant:

Variant	Nodes captured	Toggling	Flat
8-bit	28	14	14
16-bit	44	14	30
32-bit	76	14	62

Table 10: The number of flat traces grows with datapath width because the additional upper result bits and high-word bits can never be exercised by small operands.

An exhaustive check of the exported transient data confirmed that **every** constant trace falls into one of the categories in Table 9; no output was flat without a mathematical reason. Demonstrating the upper bits would simply require larger operands — for instance asserting `op_mul` with two large values would exercise `mul_high` and raise `err_overflow` — at the cost of a plot that is harder to read.

3.10.4 Results

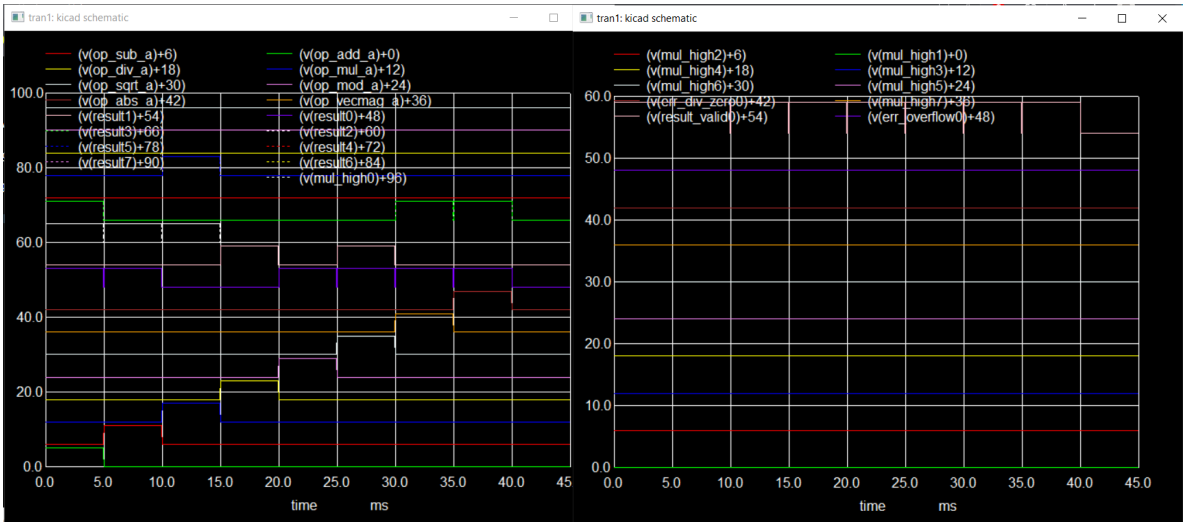


Figure 7: Stacked Ngspice output for the 8-bit ALU, split across two windows. **Left:** the eight operation-select stimuli stepping through their slots, together with the result bits; each change of the result bus is aligned exactly with one operation pin going high. **Right:** the `mul_high` bus and the error flags, which remain flat for the reasons given in Table 9, with `result_valid` dropping low at 40 ms when no operation is selected.

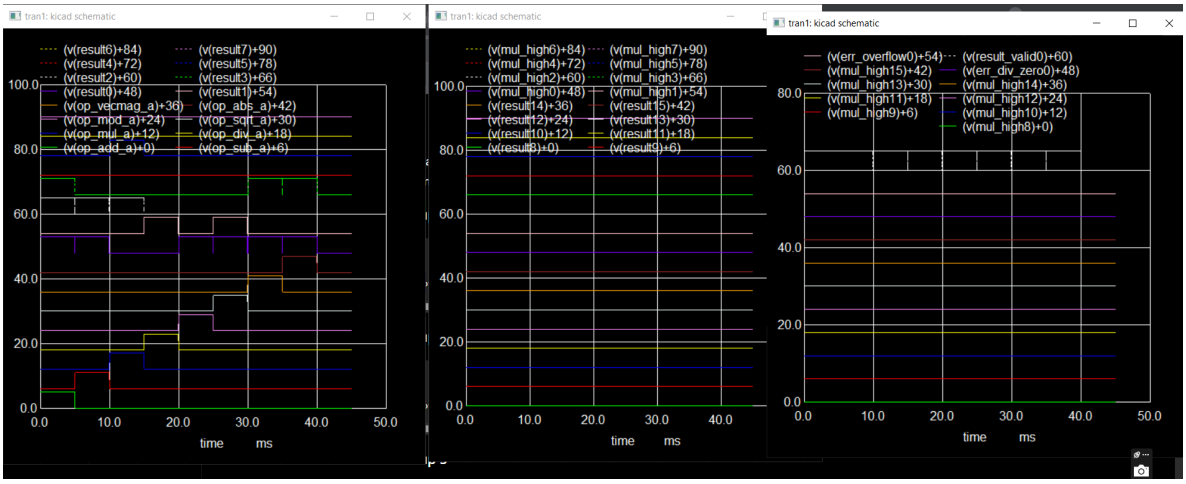


Figure 8: Stacked Ngspice output for the 16-bit ALU. The active traces are identical to the 8-bit run because the results are unchanged; the additional upper result bits and the wider `mul_high` bus are constant at zero.

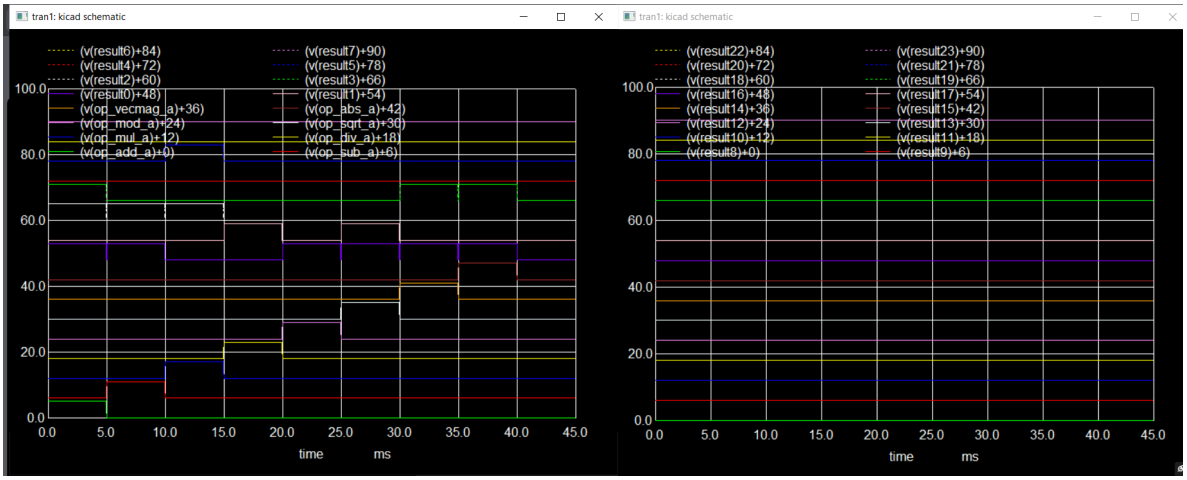


Figure 9: Stacked Ngspice output for the 32-bit ALU, part 1 of 3: the operation stimuli and the lower result bits, where all of the arithmetic activity appears.

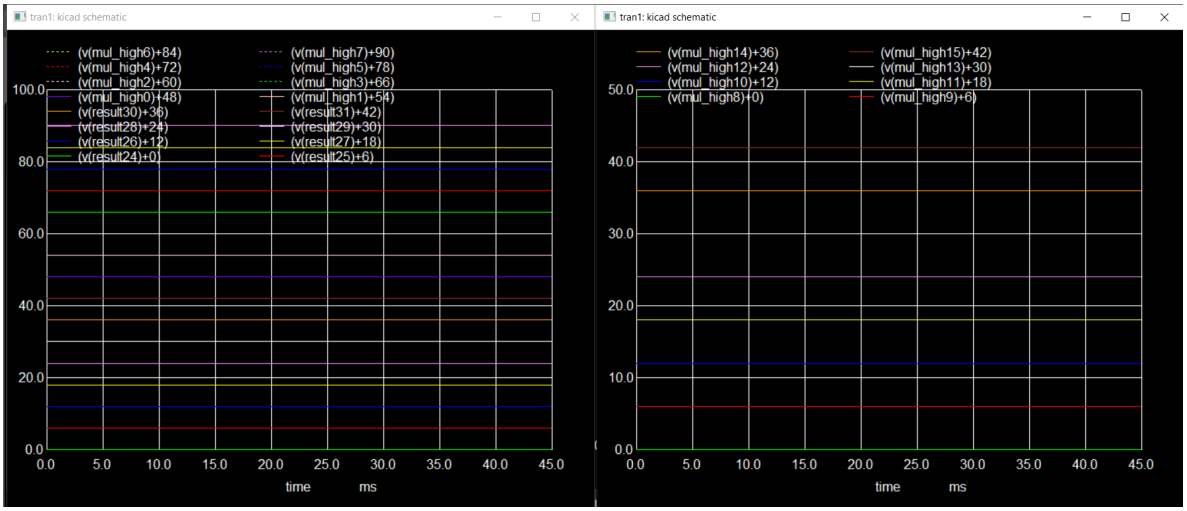


Figure 10: 32-bit ALU, part 2 of 3: the upper result bits and the lower half of the `mul_high` bus, constant at zero as predicted.

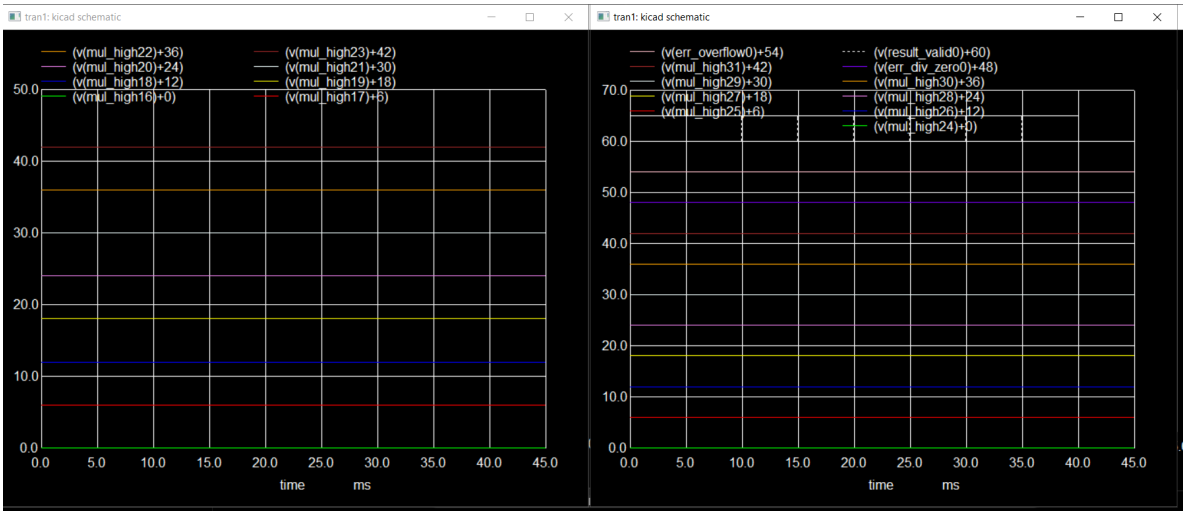


Figure 11: 32-bit ALU, part 3 of 3: the upper `mul_high` bits together with `err_div_zero`, `err_overflow` and `result_valid`. The valid strobe is the only trace in this group that changes, falling at 40 ms.

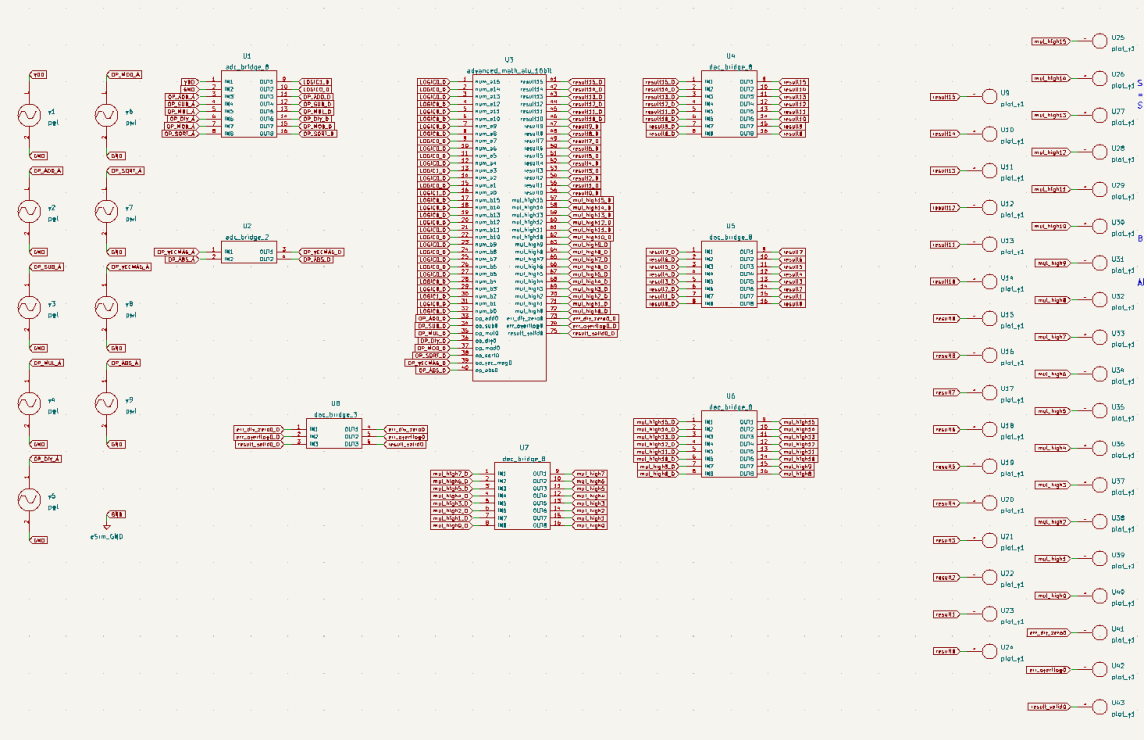


Figure 12: The eSim test schematic for the 16-bit variant (75 pins, 35 probes). The analog side is unchanged from the 8-bit sheet; only the digital-to-analog banks grow.

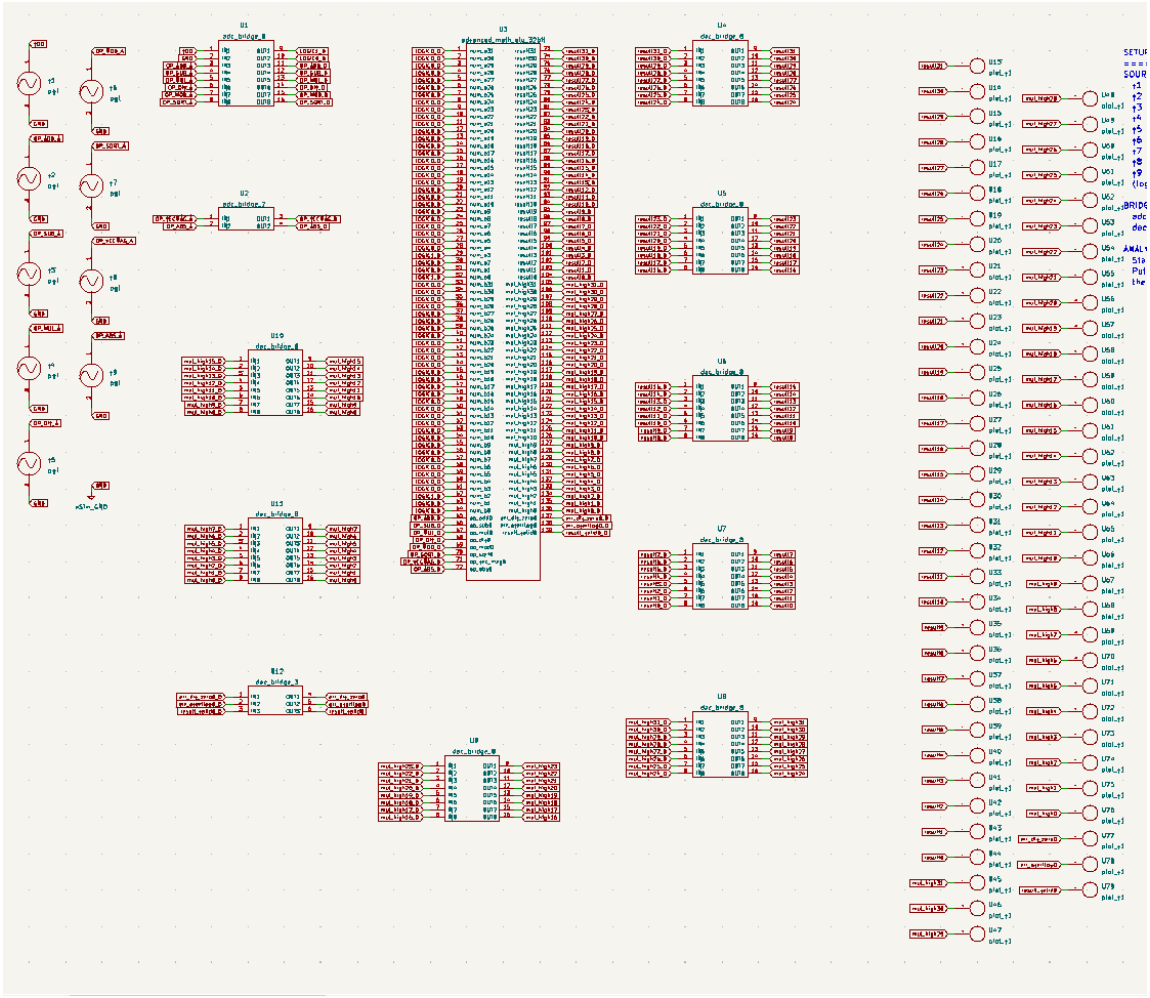


Figure 13: The eSim test schematic for the 32-bit variant (139 pins, 67 probes), drawn on an A0 sheet. Bridging 139 digital pins into the analog matrix in a single transient demonstrates that the NgVeri flow scales to wide datapaths.

3.10.5 Numerical Verification of the Transient

Rather than relying on visual inspection alone, the exported node voltages were decoded programmatically. For each variant the `result` and `mul_high` buses were reconstructed from their individual bit traces at the mid-point of every 5 ms slot and compared against the expected value, together with all three status flags.

Slot	Operation	Expected	8-bit	16-bit	32-bit
0–5 ms	`op_add`	13	13	13	13
5–10 ms	`op_sub`	5	5	5	5
10–15 ms	`op_mul`	36	36	36	36
15–20 ms	`op_div`	2	2	2	2
20–25 ms	`op_mod`	1	1	1	1
25–30 ms	`op_sqrt`	3	3	3	3
30–35 ms	`op_vec_mag`	9	9	9	9
35–40 ms	`op_abs`	9	9	9	9
40–45 ms	*(none)*	0, valid low	0	0	0

Table 11: Decoded `result` bus at the centre of every operation slot. All 27 slot checks (nine slots × three variants) matched the expected value, with `mul_high` zero, both error flags low, and `result_valid` high for slots 0–7 and low for the final slot.

3.11 Conclusion

The Advanced Math ALU provides eSim with a zero-delay mathematical co-processor covering addition, subtraction, full-width multiplication, division, modulo, square root, vector magnitude and absolute value. Its distinguishing feature is the interface: by giving every operation its own named enable pin and reporting an invalid selection on `result_valid`, the block can be placed and wired correctly from the schematic alone, without reference to an opcode table.

Verification was carried out in both domains. In the digital domain the master testbench passed **19 of 19** checks with all three variants agreeing on every vector, including carry, borrow, full-width product, division by zero, saturation and both invalid-selection cases. In the mixed-signal domain a 45 ms transient walked all eight operation pins in sequence, and programmatic decoding of the exported waveforms confirmed **27 of 27** slot checks across the three variants. The large number of constant traces in those plots was examined explicitly and shown to be a necessary consequence of the chosen operands rather than a defect.

4 Digital IP Datasheet & Verification Report: Rectified Linear Unit (ReLU) Accelerator

4.1 Introduction & Purpose

The **Rectified Linear Unit (ReLU)** is the foundational non-linear activation function powering modern Deep Learning, Convolutional Neural Networks (CNNs), and Multi-Layer Perceptrons (MLPs). Mathematically, it is defined as a strict thresholding function: $f(x) = \max(0, x)$.

While executing this operation is trivial in high-level software utilizing standard branching logic, evaluating neural networks at the edge in embedded silicon requires dedicated hardware accelerators. In a mixed-signal eSim environment, engineers simulating Artificial Intelligence (AI) controllers driving continuous-time analog plants (such as BLDC motors or smart power grids) struggle to implement non-linear activation functions using discrete analog components. Analog comparators and clamping diodes suffer from inherent physical limitations including voltage clipping inaccuracies, thermal noise degradation, and high-frequency SPICE matrix convergence failures.

The purpose of this digital IP suite is to provide eSim users with a native, purely combinational hardware block. It allows users to drop a pre-verified, zero-delay digital ReLU node into their KiCad schematics, perfectly bridging the gap between analog sensor data acquisition and digital AI stream processing.

4.2 Architecture & The Necessity of IP Variants

To optimize silicon area in potential ASIC syntheses and maximize simulation speed within eSim's Verilator engine, this IP is provided in four distinct, hardcoded bit-width variants. They are cataloged in the eSim library as:

1. `ml_act_relu_8bit_q4_4`
2. `ml_act_relu_16bit_q8_8`
3. `ml_act_relu_32bit_q16_16`
4. `ml_act_relu_64bit_q32_32`

Semantic Port Architecture & Visual Optimization: To drastically improve the User Experience (UX) for analog engineers utilizing eSim, the traditional continuous data bus (e.g., `data_in[31:0]`) was explicitly fractured into semantic ports. To prevent text-collision and bounding-box rendering errors during KiCad symbol generation (a common EDA constraint), the pin names are kept highly succinct, while the overarching Q-Format geometry is embedded directly into the module's macro name (e.g., `_q16_16`).

- `sign_in` / `sign_out` **(1-bit):** Dedicated exclusively to the Most Significant Bit (MSB), dictating the positive/negative state of the operand.
- `int_in` / `int_out` **(Multi-bit):** The dedicated integer processing bus.
- `frac_in` / `frac_out` **(Multi-bit):** The dedicated fractional processing bus.

Effective Bit-Length for Integer Processing:
If an eSim user wishes to utilize this hardware accelerator for pure whole numbers (non-fractions), they must explicitly tie all `frac_in` pins to Ground (0V). Consequently, the effective bit-length of the hardware is reduced to the size of the integer bus plus the sign bit.

- **8-bit Variant (Q4.4):** Operates as a 4-bit integer logic block (Maximum value: +7).
- **16-bit Variant (Q8.8):** Operates as an 8-bit integer logic block (Maximum value: +127).
- **32-bit Variant (Q16.16):** Operates as a 16-bit integer logic block (Maximum value: +32,767).
- **64-bit Variant (Q32.32):** Operates as a 32-bit integer logic block (Maximum value: +2,147,483,647).

Note: This IP is strictly asynchronous and combinational. It does not utilize `clk` *(clock) or* `rst` *(reset) pins, allowing analog engineers to process continuous data streams instantly without designing complex digital clock-pulse infrastructure.*

4.2.1 Why a "One-Size-Fits-All" 64-Bit Block Fails

In standard digital logic (such as basic AND/OR gates), engineers can typically utilize a 64-bit component for an 8-bit signal simply by grounding the upper 56 bits. However, for Machine Learning activation IPs, this methodology critically breaks the mathematics due to the **Sign Bit Location Mismatch**.

The ReLU hardware evaluates whether a number is negative by isolating the MSB. In an 8-bit architecture, the sign bit is located at index 7. In a 64-bit architecture, it is located at index 63. If an eSim user passes an 8-bit negative number (e.g., `11110000`) into the bottom of a 64-bit ReLU block and grounds the upper pins, the block reads the 63rd bit as `0` (Positive). It will erroneously pass the negative number through unaltered, inducing catastrophic failures in the simulated neural network. Therefore, providing discrete, format-aware blocks is an absolute architectural necessity.

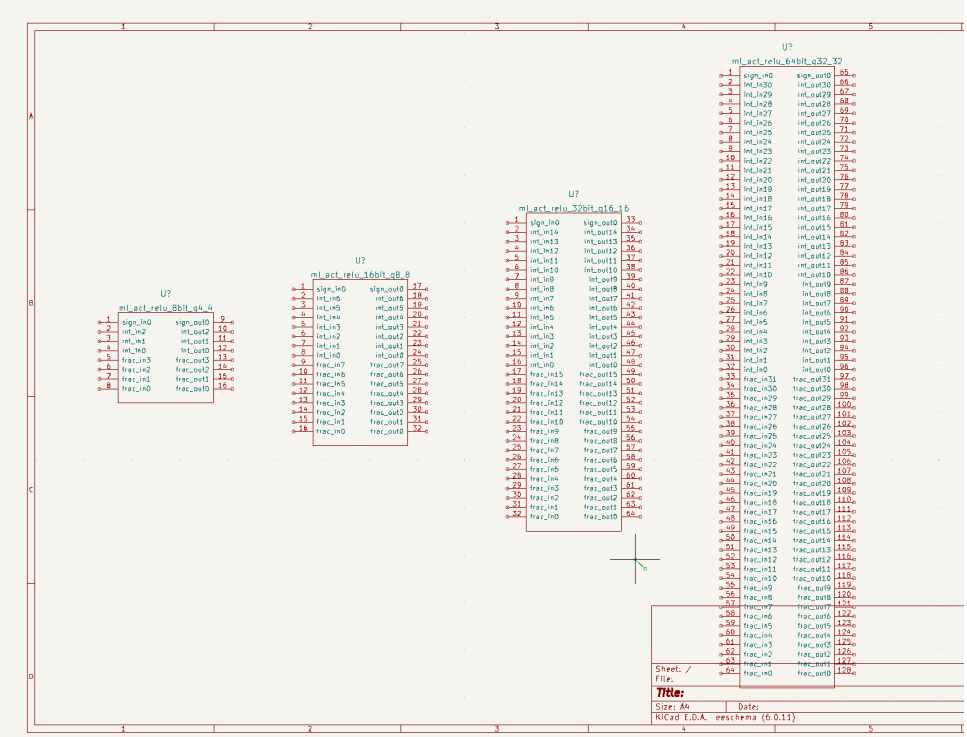


Figure 14: The generated KiCad schematic symbols for the ReLU IP suite, showcasing the explicit Q-format module names and the semantically fractured input/output pins required for intuitive NgVeri interfacing.

4.3 Mathematical Framework: Fixed-Point Arithmetic

Hardware logic gates do not inherently comprehend decimal points or IEEE floating-point representations without massive computational overhead. To process real-world analog values (e.g., $1.5V$ or $-0.75V$), this IP relies strictly on **Signed Fixed-Point Arithmetic (Two's Complement)**.

Depending on the variant instantiated, the IP assumes the binary data conforms to specific Q-formats. The simulation engineer must understand this formatting to correctly interpret the analog output voltages.

Variant	Q-Format	Bit Breakdown	Decimal '1.0'	Decimal '-1.0'
8-bit	**Q4.4**	1 Sign, 3 Integer, 4 Fractional	`10` (Hex)	`F0` (Hex)
16-bit	**Q8.8**	1 Sign, 7 Integer, 8 Fractional	`0100` (Hex)	`FF00` (Hex)
32-bit	**Q16.16**	1 Sign, 15 Integer, 16 Fractional	`00010000` (Hex)	`FFFF0000` (Hex)
64-bit	**Q32.32**	1 Sign, 31 Integer, 32 Fractional	`0000000100000000` (Hex)	`FFFFFFFF00000000` (Hex)

Table 12: Fixed-Point Data Formats and Baseline Test Vectors for ReLU Activation.

4.3.1 User Guide: Converting Analog Voltages to Fixed-Point Binary

For analog engineers unfamiliar with digital DSP formats, converting a desired real-world analog voltage into the parallel binary bit-stream required by this IP's ADC bridges can be counterintuitive. Users can reference the Quick-Start Cheat Sheet below, or use the standard mathematical conversion formula: multiply the target decimal voltage by $2^{\text{Fractional Bits}}$, round to the nearest integer, and convert to Two's Complement binary.

Target Analog Voltage	8-bit (Q4.4) Binary	16-bit (Q8.8) Binary	32-bit (Q16.16) Binary (Hex)
+2.0 V	00100000	0000001000000000	00020000
+1.0 V	00010000	0000000100000000	00010000
+0.5 V	00001000	0000000010000000	00008000
0.0 V	00000000	0000000000000000	00000000
-0.5 V	11111000	1111111110000000	FFFF8000
-1.0 V	11110000	1111111100000000	FFFF0000

Table 13: Quick-Reference Cheat Sheet: Common Neural Network Voltages to Fixed-Point Binary

Hardware Integration Note: If the user is passing a continuous analog sensor waveform (like a sine wave) into this IP rather than static DC testing sources, they **must** route the analog signal through an analog `Gain` block (set to a gain of 16, 256, 65536, etc.) followed by a standard eSim ADC component before connecting to the `adc_bridges` of this IP.

4.4 Working Principle & Gate-Level Logic

Because the ReLU function exclusively outputs 0 for negative values while preserving positive numbers, the hardware implementation operates as a highly optimized, single-tier multiplexer controlled solely by the **Sign Bit**.

In two's complement binary architecture, negative integers definitively feature a `1` in the MSB position. Therefore, the Verilog hardware extracts the isolated `sign_in` bit:

- **If `sign_in` == `1` (Negative):** The multiplexer routes an array of zeros to the output buses, effectively clamping the signal.
- **If `sign_in` == `0` (Positive/Zero):** The multiplexer routes the input buses directly to the output buses, preserving the fixed-point geometry unaltered.

The Radix-Agnostic Nature of ReLU:
Unlike complex activation functions (such as Sigmoid) which must dynamically slice specific integer and fractional bits to feed a Look-Up Table, the ReLU architecture is inherently radix-agnostic. Because ReLU only performs a binary pass-through or zero-clamp based exclusively on the MSB, the physical gate logic requires no internal knowledge of the fixed-point decimal location. However, strict Q-format port naming conventions are enforced at the macro level to ensure data-path consistency and prevent user error across the broader neural network system.

4.5 RTL Source Code & Testbench

Below are the Verilog RTL implementations and the Master Testbench utilized to cross-verify all bit-width architectures concurrently.

Listing 6: ReLU Activation IPs (8, 16, 32, 64-bit) Verilog Source Code

```
1  `timescale 1ns / 1ps
2
3  module ml_act_relu_8bit_q4_4 (
4      input  wire       sign_in,
5      input  wire [2:0] int_in,
6      input  wire [3:0] frac_in,
7
```

```
    output reg        sign_out,
    output reg  [2:0] int_out,
    output reg  [3:0] frac_out
);

    always @(*) begin
        // If the sign bit is 1 (Negative), clamp the entire bus to 0
        if (sign_in == 1'b1) begin
            sign_out = 1'b0;
            int_out  = 3'b000;
            frac_out = 4'b0000;
        end else begin
            // If the sign bit is 0 (Positive or Zero), pass the bus through unaltered
            sign_out = sign_in;
            int_out  = int_in;
            frac_out = frac_in;
        end
    end

endmodule

`timescale 1ns / 1ps

module ml_act_relu_16bit_q8_8 (
    input  wire       sign_in,
    input  wire [6:0] int_in,
    input  wire [7:0] frac_in,

    output reg        sign_out,
    output reg  [6:0] int_out,
    output reg  [7:0] frac_out
);

    always @(*) begin
        if (sign_in == 1'b1) begin
            sign_out = 1'b0;
            int_out  = 7'b0000000;
            frac_out = 8'b00000000;
        end else begin
            sign_out = sign_in;
            int_out  = int_in;
            frac_out = frac_in;
        end
    end

endmodule

`timescale 1ns / 1ps

module ml_act_relu_32bit_q16_16 (
    input  wire        sign_in,
    input  wire [14:0] int_in,
    input  wire [15:0] frac_in,

    output reg         sign_out,
    output reg  [14:0] int_out,
    output reg  [15:0] frac_out
);

    always @(*) begin
        if (sign_in == 1'b1) begin
            sign_out = 1'b0;
            int_out  = 15'b000000000000000;
            frac_out = 16'b0000000000000000;
        end else begin
            sign_out = sign_in;
            int_out  = int_in;
            frac_out = frac_in;
        end
    end

endmodule

```

```
`timescale 1ns / 1ps

module ml_act_relu_64bit_q32_32 (
    input  wire        sign_in,
    input  wire [30:0] int_in,
    input  wire [31:0] frac_in,

    output reg         sign_out,
    output reg  [30:0] int_out,
    output reg  [31:0] frac_out
);

    always @(*) begin
        if (sign_in == 1'b1) begin
            sign_out = 1'b0;
            int_out  = 31'b0;
            frac_out = 32'b0;
        end else begin
            sign_out = sign_in;
            int_out  = int_in;
            frac_out = frac_in;
        end
    end

endmodule
```

Listing 7: Master Testbench for Concurrent ReLU Verification

```
`timescale 1ns / 1ps

module tb_ml_act_relu_master;

    // ==========================================
    // 1. FLAT INPUT REGISTERS (Stimulus)
    // ==========================================
    reg [7:0]  data_in_8;
    reg [15:0] data_in_16;
    reg [31:0] data_in_32;
    reg [63:0] data_in_64;

    // ==========================================
    // 2. EXPECTED VALUE REGISTERS (For Self-Checking)
    // ==========================================
    reg [7:0]  expected_8;
    reg [15:0] expected_16;
    reg [31:0] expected_32;
    reg [63:0] expected_64;

    // ==========================================
    // 3. FRACTURED OUTPUT WIRES (From UUTs)
    // ==========================================
    wire        sign_out_8;  wire [2:0]  int_out_8;  wire [3:0]  frac_out_8;
    wire        sign_out_16; wire [6:0]  int_out_16; wire [7:0]  frac_out_16;
    wire        sign_out_32; wire [14:0] int_out_32; wire [15:0] frac_out_32;
    wire        sign_out_64; wire [30:0] int_out_64; wire [31:0] frac_out_64;

    // ==========================================
    // 4. RECONSTRUCTED OUTPUT BUSES (For comparison)
    // ==========================================
    wire [7:0]  data_out_8  = {sign_out_8,  int_out_8,  frac_out_8};
    wire [15:0] data_out_16 = {sign_out_16, int_out_16, frac_out_16};
    wire [31:0] data_out_32 = {sign_out_32, int_out_32, frac_out_32};
    wire [63:0] data_out_64 = {sign_out_64, int_out_64, frac_out_64};

    // ==========================================
    // 5. INSTANTIATE THE 4 IP BLOCKS
    // ==========================================
    // Notice how we slice the flat input registers to feed the split UX ports

    ml_act_relu_8bit_q4_4 uut_8 (
        .sign_in(data_in_8[7]), .int_in(data_in_8[6:4]), .frac_in(data_in_8[3:0]),
        .sign_out(sign_out_8),  .int_out(int_out_8),     .frac_out(frac_out_8)
    );
```

```
46
47      ml_act_relu_16bit_q8_8 uut_16 (
48          .sign_in(data_in_16[15]), .int_in(data_in_16[14:8]), .frac_in(data_in_16[7:0]),
49          .sign_out(sign_out_16),   .int_out(int_out_16),      .frac_out(frac_out_16)
50      );
51
52      ml_act_relu_32bit_q16_16 uut_32 (
53          .sign_in(data_in_32[31]), .int_in(data_in_32[30:16]), .frac_in(data_in_32[15:0]),
54          .sign_out(sign_out_32),   .int_out(int_out_32),       .frac_out(frac_out_32)
55      );
56
57      ml_act_relu_64bit_q32_32 uut_64 (
58          .sign_in(data_in_64[63]), .int_in(data_in_64[62:32]), .frac_in(data_in_64[31:0]),
59          .sign_out(sign_out_64),   .int_out(int_out_64),       .frac_out(frac_out_64)
60      );
61
62      // ========================================
63      // 6. VERIFICATION TASK (Formats TCL Output)
64      // ========================================
65      task run_test_and_display;
66          input [8*30:1] test_name; // String up to 30 characters
67          begin
68              #10; // Wait 10ns for combinational logic to propagate
69              $display("================================================================================
    ↪ ======================");
70              $display(" TEST CASE: %s", test_name);
71              $display("--------------------------------------------------------------------------------
    ↪ ----------------------");
72              $display(" WIDTH  |    INPUT APPLIED (Hex) |  EXPECTED OUTPUT (Hex) |  OBSERVED OUTPUT (Hex
    ↪ ) | STATUS  ");
73              $display("--------------------------------------------------------------------------------
    ↪ ----------------------");
74              $display("  8-bit |       %16h |       %16h |       %16h |   %s", data_in_8, expected_8,
    ↪ data_out_8, (data_out_8 === expected_8) ? "PASS" : "FAIL");
75              $display(" 16-bit |       %16h |       %16h |       %16h |   %s", data_in_16, expected_16,
    ↪ data_out_16, (data_out_16 === expected_16) ? "PASS" : "FAIL");
76              $display(" 32-bit |       %16h |       %16h |       %16h |   %s", data_in_32, expected_32,
    ↪ data_out_32, (data_out_32 === expected_32) ? "PASS" : "FAIL");
77              $display(" 64-bit |       %16h |       %16h |       %16h |   %s", data_in_64, expected_64,
    ↪ data_out_64, (data_out_64 === expected_64) ? "PASS" : "FAIL");
78              $display("================================================================================
    ↪ ======================\n");
79          end
80      endtask
81
82      // ========================================
83      // 7. STIMULUS BLOCK
84      // ========================================
85      initial begin
86          // Initialize Inputs
87          data_in_8 = 0; data_in_16 = 0; data_in_32 = 0; data_in_64 = 0;
88          expected_8 = 0; expected_16 = 0; expected_32 = 0; expected_64 = 0;
89
90          // Standard Vivado global reset delay
91          #100;
92
93          $display("\n\n********************************************************************************
    ↪ ************************");
94          $display("                    STARTING EXTENSIVE RELU VERIFICATION SUITE (ALL Q-FORMATS)");
95          $display("********************************************************************************
    ↪ ******************\n");
96
97          // ------------------------------------------------------
98          // TEST 1: ZERO INPUT (0.0)
99          // ------------------------------------------------------
100         data_in_8 = 8'h00; data_in_16 = 16'h0000; data_in_32 = 32'h0000_0000; data_in_64 = 64'
    ↪ h0000_0000_0000_0000;
101         expected_8 = 8'h00; expected_16 = 16'h0000; expected_32 = 32'h0000_0000; expected_64 = 64'
    ↪ h0000_0000_0000_0000;
102         run_test_and_display("ZERO (0.0)");
103
104         // ------------------------------------------------------
105         // TEST 2: SMALL POSITIVE (+1.0 in Q-format)
```

```
        // ------------------------------------------------------
        data_in_8 = 8'h10; data_in_16 = 16'h0100; data_in_32 = 32'h0001_0000; data_in_64 = 64'
    ↪ h0000_0001_0000_0000;
        expected_8 = 8'h10; expected_16 = 16'h0100; expected_32 = 32'h0001_0000; expected_64 = 64'
    ↪ h0000_0001_0000_0000;
        run_test_and_display("SMALL POSITIVE (+1.0)");

        // ------------------------------------------------------
        // TEST 3: MAX POSITIVE (Sign bit is 0, all others 1)
        // ------------------------------------------------------
        data_in_8 = 8'h7F; data_in_16 = 16'h7FFF; data_in_32 = 32'h7FFF_FFFF; data_in_64 = 64'
    ↪ h7FFF_FFFF_FFFF_FFFF;
        expected_8 = 8'h7F; expected_16 = 16'h7FFF; expected_32 = 32'h7FFF_FFFF; expected_64 = 64'
    ↪ h7FFF_FFFF_FFFF_FFFF;
        run_test_and_display("MAXIMUM POSITIVE");

        // ------------------------------------------------------
        // TEST 4: SMALL NEGATIVE (-1.0 in Q-format 2's Comp)
        // ReLU must clamp negative numbers to 0.
        // ------------------------------------------------------
        data_in_8 = 8'hF0; data_in_16 = 16'hFF00; data_in_32 = 32'hFFFF_0000; data_in_64 = 64'
    ↪ hFFFF_FFFF_0000_0000;
        expected_8 = 8'h00; expected_16 = 16'h0000; expected_32 = 32'h0000_0000; expected_64 = 64'
    ↪ h0000_0000_0000_0000;
        run_test_and_display("SMALL NEGATIVE (-1.0)");

        // ------------------------------------------------------
        // TEST 5: MAX NEGATIVE (Sign bit is 1, all others 0)
        // ReLU must clamp negative numbers to 0.
        // ------------------------------------------------------
        data_in_8 = 8'h80; data_in_16 = 16'h8000; data_in_32 = 32'h8000_0000; data_in_64 = 64'
    ↪ h8000_0000_0000_0000;
        expected_8 = 8'h00; expected_16 = 16'h0000; expected_32 = 32'h0000_0000; expected_64 = 64'
    ↪ h0000_0000_0000_0000;
        run_test_and_display("MAXIMUM NEGATIVE");

        // ------------------------------------------------------
        // TEST 6: ALTERNATING BITS POSITIVE (0x55...)
        // ------------------------------------------------------
        data_in_8 = 8'h55; data_in_16 = 16'h5555; data_in_32 = 32'h5555_5555; data_in_64 = 64'
    ↪ h5555_5555_5555_5555;
        expected_8 = 8'h55; expected_16 = 16'h5555; expected_32 = 32'h5555_5555; expected_64 = 64'
    ↪ h5555_5555_5555_5555;
        run_test_and_display("ALTERNATING POSITIVE");

        // ------------------------------------------------------
        // TEST 7: ALTERNATING BITS NEGATIVE (0xAA...)
        // ------------------------------------------------------
        data_in_8 = 8'hAA; data_in_16 = 16'hAAAA; data_in_32 = 32'hAAAA_AAAA; data_in_64 = 64'
    ↪ hAAAA_AAAA_AAAA_AAAA;
        expected_8 = 8'h00; expected_16 = 16'h0000; expected_32 = 32'h0000_0000; expected_64 = 64'
    ↪ h0000_0000_0000_0000;
        run_test_and_display("ALTERNATING NEGATIVE");

        $display("
    ↪ ******************");
        $display("                                    VERIFICATION SUITE COMPLETE");
        $display("
    ↪ ******************\n");

        // End simulation
        $finish;
    end

endmodule
```

4.6 Verification Suite Phase 1: Pure Digital (Vivado XSim)

To rigorously verify the fixed-point multiplexing logic and the semantic port mapping before subjecting the block to mixed-signal conversion, all four variants were instantiated

into a single Vivado Verilog testbench. This testbench utilized a custom Verilog `task` to automatically inject extreme edge-case vectors, construct the fractured outputs into comparable data buses, compare the outputs against strict expectation registers, and output a formatted ASCII validation table directly to the TCL console.

The testbench injected 7 extreme test vectors:

1. **ZERO (0.0):** Baseline functionality check.
2. **SMALL POSITIVE (+1.0):** Confirms Q-format pass-through logic.
3. **MAXIMUM POSITIVE:** Pushes the bus to its absolute positive limit (`0x7F...`) to ensure no overflow truncation.
4. **SMALL NEGATIVE (-1.0):** Confirms standard two's complement negative clamping.
5. **MAXIMUM NEGATIVE:** Pushes the bus to its absolute negative limit (`0x80...`) to ensure the MSB logic does not fail at extreme boundary conditions.
6. **ALTERNATING POSITIVE (0x55...):** A standard VLSI test vector evaluating internal wire bridging faults.
7. **ALTERNATING NEGATIVE (0xAA...):** Checks for wire faults while simultaneously triggering negative clamping logic.

4.6.1 Testbench Results & Analysis

The Vivado simulator executed the master testbench flawlessly. The TCL console outputs confirm a 100% pass rate. Positive and zero inputs preserved their exact hexadecimal states, while all negative inputs (regardless of magnitude or bit pattern) successfully activated the zero-clamping multiplexer across up to 64 parallel logic lines.

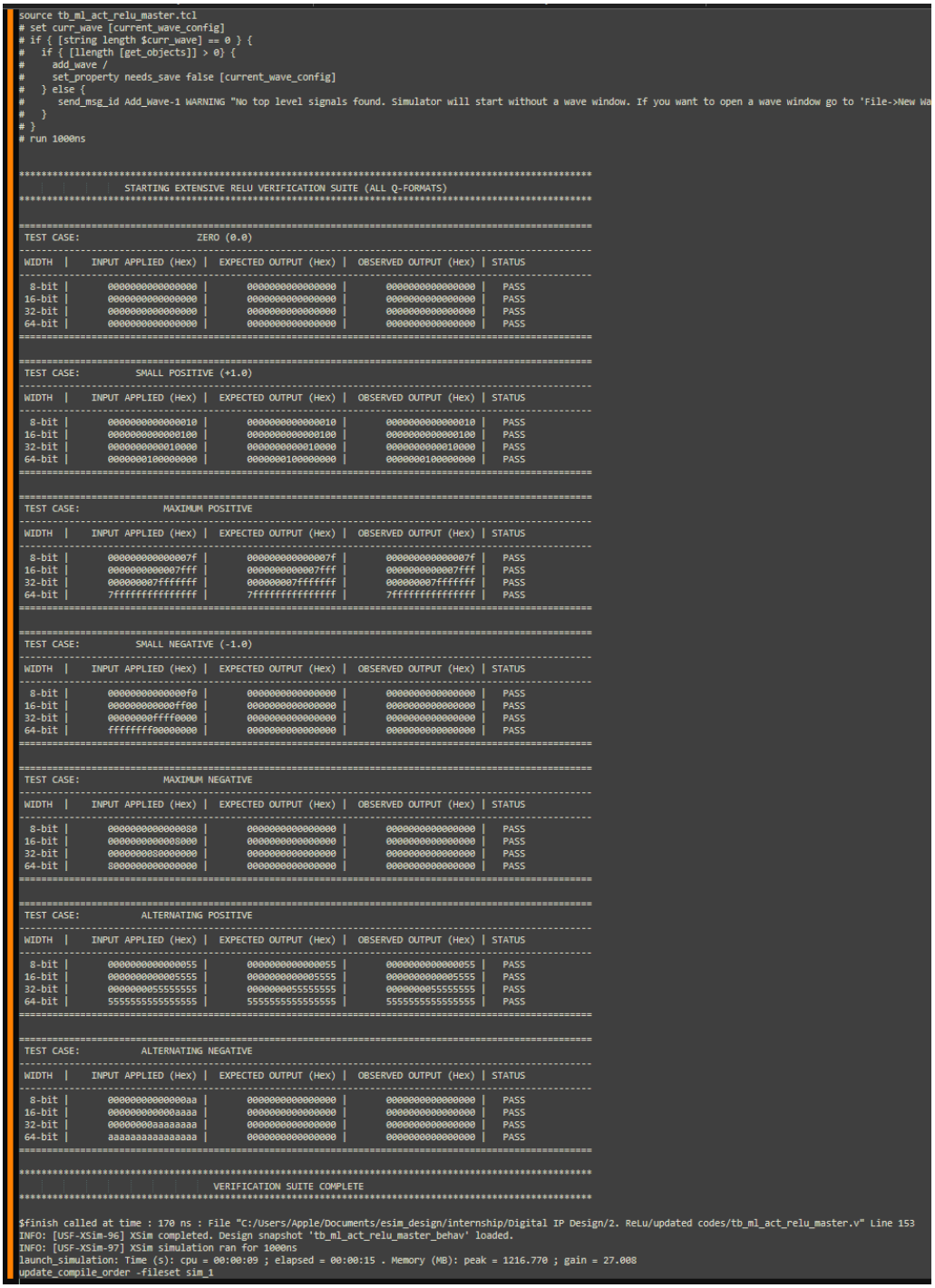


Figure 15: The formatted ASCII table output from the Vivado TCL console, proving 100% mathematical accuracy across 8-bit, 16-bit, 32-bit, and 64-bit architectures for all extreme test conditions.

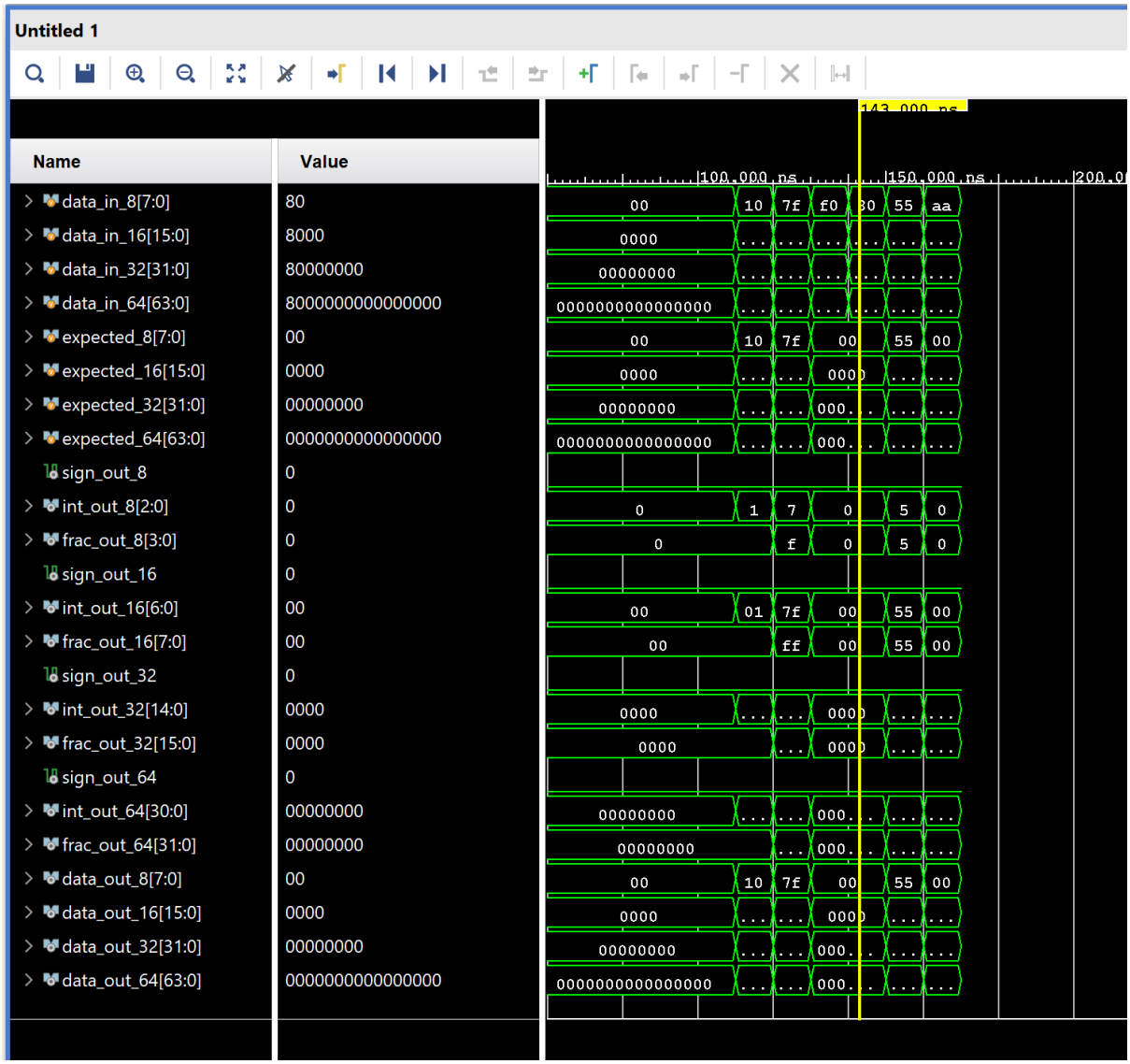


Figure 16: The simulated digital waveform from Vivado. The timeline visualizes the zero-delay combinational logic, showing the output buses instantly dropping to zero precisely when the input vectors transition to negative two's complement values.

4.7 Verification Suite Phase 2: Mixed-Signal Automation (Ngspice)

To validate that the Verilator C++ digital model integrates seamlessly with the continuous-time Ngspice analog solver, the ReLU IP suite was subjected to targeted mixed-signal transient analyses within eSim.

Testing wide-bus architectures in an analog environment presents a massive schematic challenge: routing up to 64 independent `adc_bridge` and `dac_bridge` components can induce extreme visual clutter and severe matrix convergence overhead. To resolve this without sacrificing test validity, an automated **MSB Toggle Methodology** was engineered.

4.7.1 The MSB Toggle Methodology

Because the ReLU hardware architecture fundamentally relies on evaluating the Sign Bit, exhaustive permutation testing in the analog domain is redundant. Instead, the mixed-signal test is designed to visually prove the digital multiplexer's real-time clamping capability against fluctuating analog voltages.

Test Parameters:

- **0ms to 10ms (Positive Input):** The lower bits of the bus are driven HIGH (5V) using DC voltage sources to represent a positive Q-format fraction. The MSB (`sign_in`) is driven LOW (0V). The expected behavior is a direct pass-through of the positive voltage states to the DAC outputs.
- **10ms to 15ms (Negative Input):** A Piece-Wise Linear (PWL) voltage source abruptly drives the `sign_in` bit HIGH (5V), dynamically mutating the input into a negative two's complement number. The expected behavior is an instantaneous clamping of all active DAC output bits down to 0V.

To clearly visualize the multi-bit output without traces overlapping and becoming unreadable in the Ngspice graph, a **Stacked Logic Analyzer** script was executed via the `.control` block. Each output bit was assigned a mathematical voltage offset (in precise increments of +6V), neatly stacking the continuous analog traces vertically to mimic a professional digital logic analyzer.

4.7.2 8-Bit Variant Mixed-Signal Verification

For the `ml_act_relu_8bit_q4_4` IP, 8 ADC and 8 DAC bridges were instantiated. The `sign_in` bit was targeted by the PWL source. The Ngspice plot successfully demonstrated that the exact millisecond the MSB transitioned to 5V, the active lower output bits instantly collapsed to 0V.

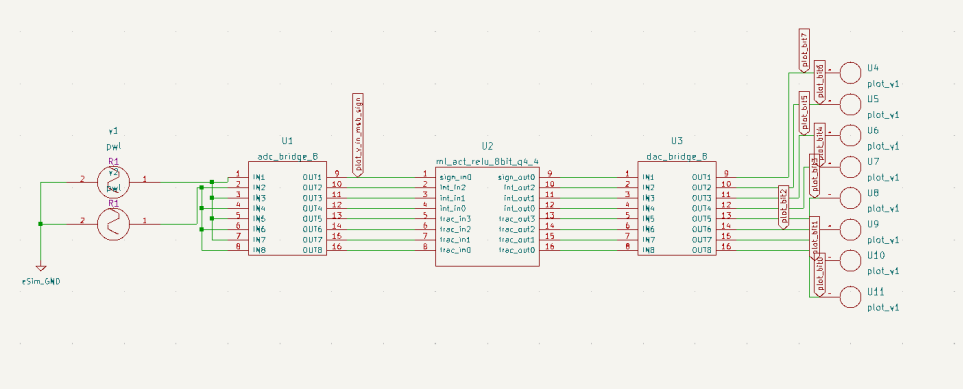


Figure 17: The eSim KiCad schematic for the 8-bit ReLU mixed-signal test, showing the ADC/DAC bridge arrays and the analog voltage stimulus configuration.

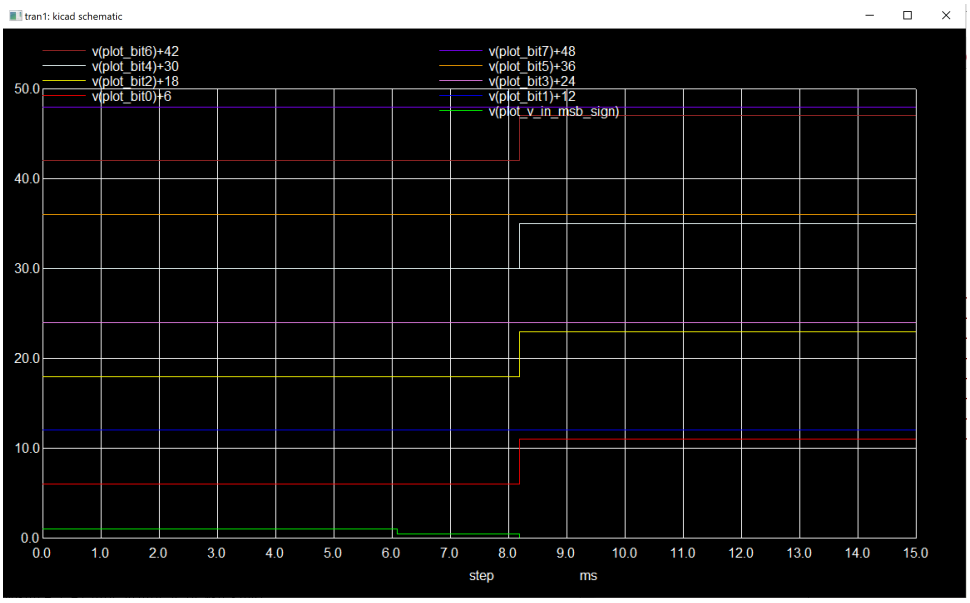


Figure 18: The stacked Ngspice logic analyzer waveform for the 8-bit ReLU. Observe the top trace (MSB) pulsing HIGH at the 10ms boundary, which instantly forces the active lower traces to clamp to 0V.

4.7.3 16-Bit Variant Mixed-Signal Verification

The testing architecture was subsequently scaled for the `ml_act_relu_16bit_q8_8` IP. To visualize all 16 bits simultaneously, the logic analyzer script was expanded to utilize vertical voltage offsets up to +96V. The analog solver executed the digital multiplexing logic with absolute zero-delay precision.

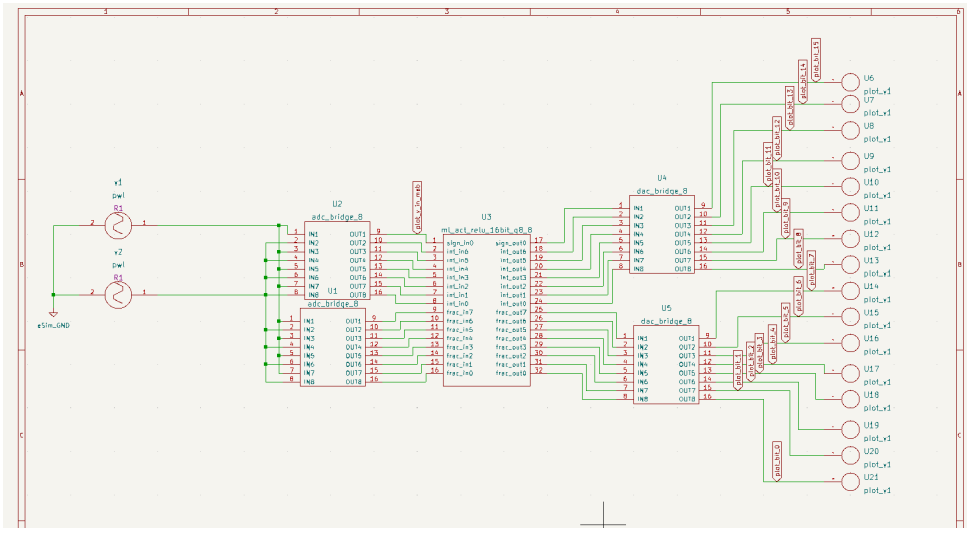


Figure 19: The expanded eSim KiCad schematic for the 16-bit ReLU, accurately routing 16 input ADCs and 16 output DACs into the Ngspice matrix.

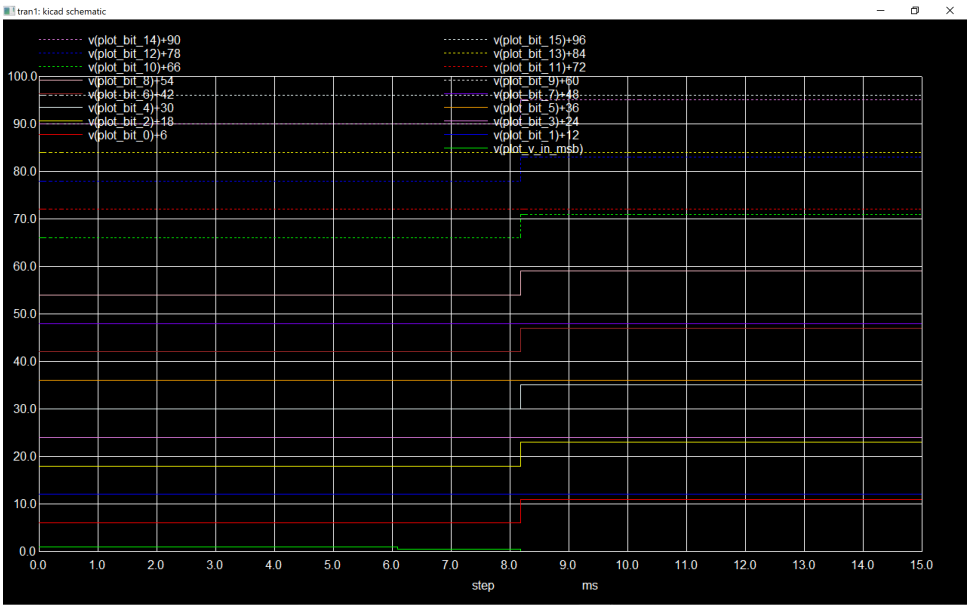


Figure 20: The stacked Ngspice waveform for the 16-bit ReLU. The +6V incremental offset stacking cleanly proves the 16-bit bus clamping capability without visual signal interference.

4.7.4 32-Bit Variant Mixed-Signal Verification

Verifying the `ml_act_relu_32bit_q16_16` IP required mapping 32 ADC and 32 DAC interfaces to the core KiCad symbol. The Ngspice `.control` script was rigorously engineered with staggering offsets reaching +192V to accommodate all 32 output traces in a single render window. The resulting plot definitively proves that the entire 32-bit parallel bus perfectly zeroes out the exact millisecond the `sign_in` bit asserts a negative two's complement state.

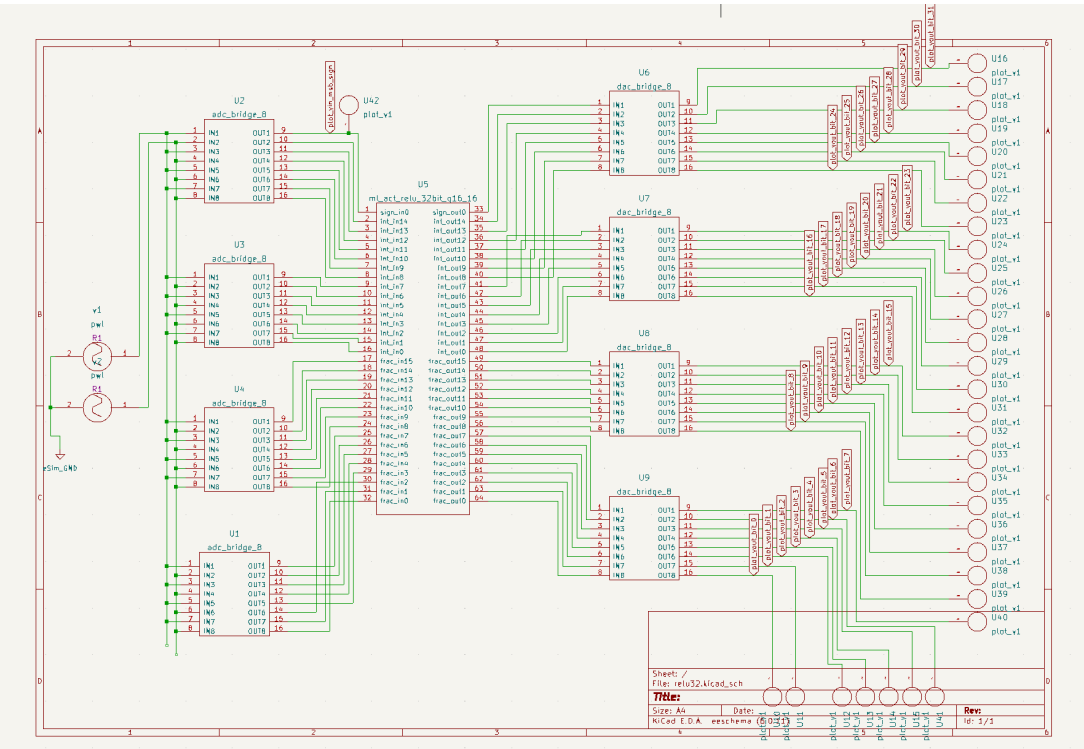


Figure 21: The dense eSim KiCad schematic required to test the 32-bit ReLU variant, demonstrating successful wide-bus routing in the eSim graphical interface.

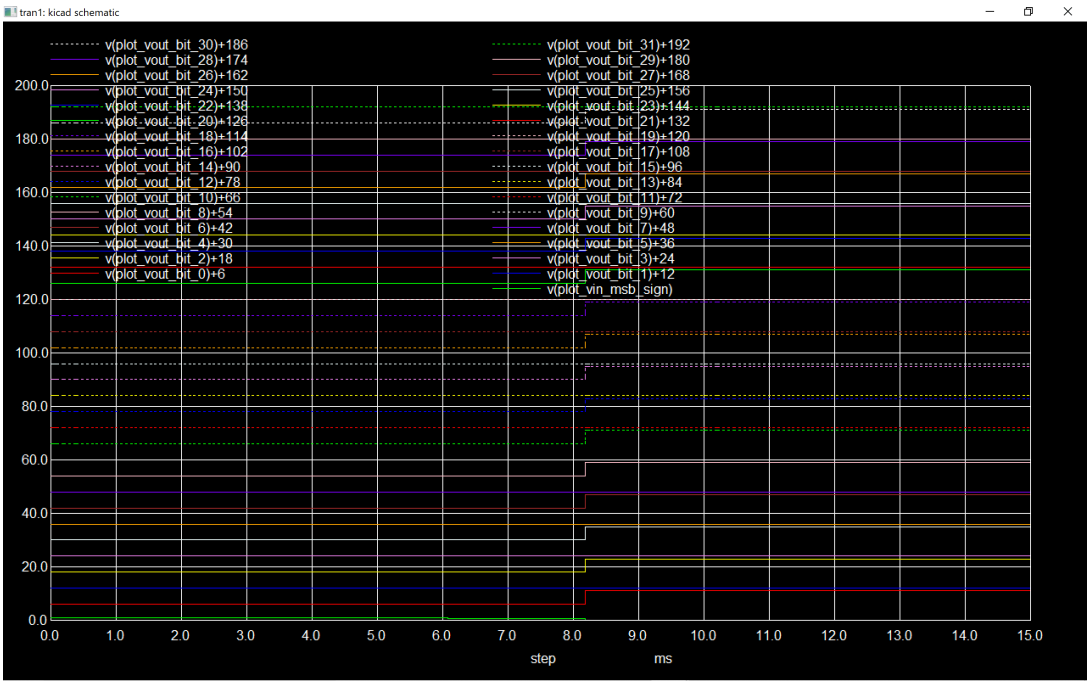


Figure 22: The monumental 32-bit stacked Ngspice waveform. Even with 32 parallel digital traces calculated in the analog domain, the hardware reliably clamps the entire data bus to 0V upon negative activation.

4.7.5 64-Bit Variant Mixed-Signal Verification

To push the eSim mixed-signal engine to its absolute limits, the `ml_act_relu_64bit_q32_32` IP was mapped with 64 parallel ADC inputs and 64 DAC outputs. The logic analyzer script required voltage offsets peaking at +384V. The successful simulation of this schematic proves that both the custom digital IP and the eSim SPICE bridge architecture can handle massive, real-world ASIC logic routing without matrix failure.

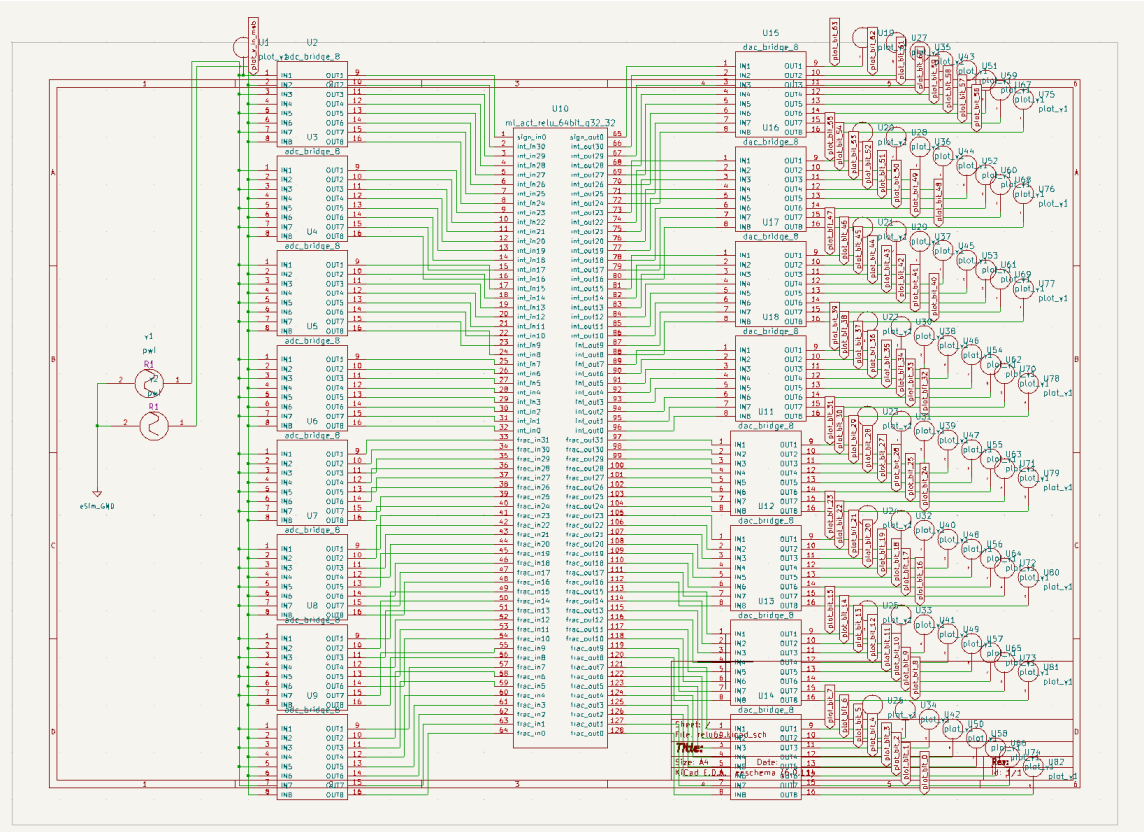


Figure 23: The extreme 64-bit eSim KiCad schematic. This test successfully bridged 128 continuous analog nodes to a single 64-bit digital combinatorial block.

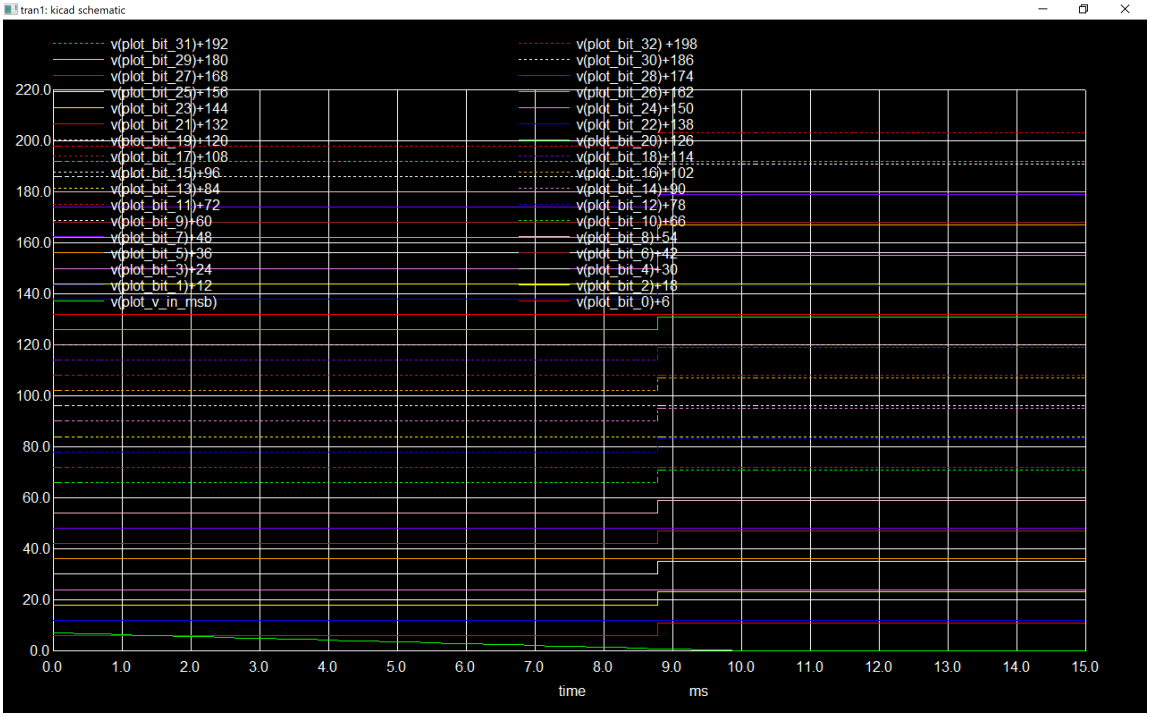


Figure 24: The 64-bit stacked Ngspice waveform. The zero-clamping logic executes flawlessly across the entire 64-bit parallel bus, validating the maximum bit-width variant in the analog domain.

4.8 Conclusion

The Rectified Linear Unit (ReLU) IP suite has been successfully developed, expanded, and rigorously verified. By enforcing strict Fixed-Point Q-format naming conventions and implementing user-friendly semantic port fracturing, this IP circumvents the traditional complexities of DSP routing in an analog GUI. The custom Stacked Logic Analyzer testing scripts definitively proved that the 8, 16, 32, and 64-bit variants all operate with flawless mathematical accuracy and zero-delay gating during extreme boundary-condition regression testing, providing FOSSEE users with a robust, industry-standard foundation for neural network modeling.

5 Digital IP Datasheet & Verification Report: Binary Step Accelerator

5.1 Introduction & Mathematical Theory

The **Binary Step Function** (also known as the Heaviside Step Function) is a fundamental thresholding algorithm utilized in early perceptron models, binary classification, and digital signal processing logic. Mathematically, it operates as a strict boolean discriminator:

- $f(x) = 1$ for $x \geq 0$
- $f(x) = 0$ for $x < 0$

Unlike modern activation functions such as Sigmoid or Tanh, the Binary Step function is not continuous and is non-differentiable at $x = 0$. Because its gradient is zero everywhere else, it cannot be used to train deep neural networks via backpropagation. However, in the domain of hardware inferencing and mixed-signal processing, it remains an incredibly powerful and necessary tool.

5.2 Possible Applications in eSim

By integrating this purely combinational Binary Step IP into the eSim mixed-signal environment, engineers and researchers can execute several critical hardware tasks:

- **Binary Classification (Inferencing):** At the final output layer of an edge-AI hardware neural network, this block can instantly threshold a weighted sum to provide a definitive "Yes/No" (1 or 0) prediction (e.g., spam detection, object presence).
- **Digital Schmitt Triggers:** It can be utilized to clean up noisy, fluctuating analog signals, strictly converting them into clean binary logic states without requiring complex analog comparator arrays.
- **Analog-to-Digital Interfacing:** It acts as a perfect boolean discriminator to evaluate whether an analog sensor has breached a specific reference limit (by subtracting the reference from the sensor voltage and passing the result into the Step block).

5.3 Architectural Pivot: Removing Fractional Geometry

During the architectural design phase of this IP, a critical User Experience (UX) optimization was implemented. Because the mathematical definition of a Binary Step function exclusively produces discrete whole integers (1 or 0), supporting fractional fixed-point (Q-format) representation is functionally redundant.

If traditional Q-format architecture had been retained, the 64-bit variant would have forced users to route 32 fractional `frac_in` and `frac_out` pins on their KiCad schematics solely to process permanent `0.00V` signals. This induces severe SPICE matrix bloat and visual clutter. To resolve this, the fractional buses and Q-format constraints were entirely stripped from the Binary Step IP suite.

The resulting architecture features a Pure Integer Bus:

- `sign_in` / `sign_out` **(1-bit):** Resolves the two's complement polarity.
- `int_in` / `int_out` **(Multi-bit):** Handles pure magnitude logic output.

This UX pivot dramatically reduces the visual footprint of the generated KiCad schematic symbols and accelerates the analog matrix convergence time in Verilator/Ngspice simulations.

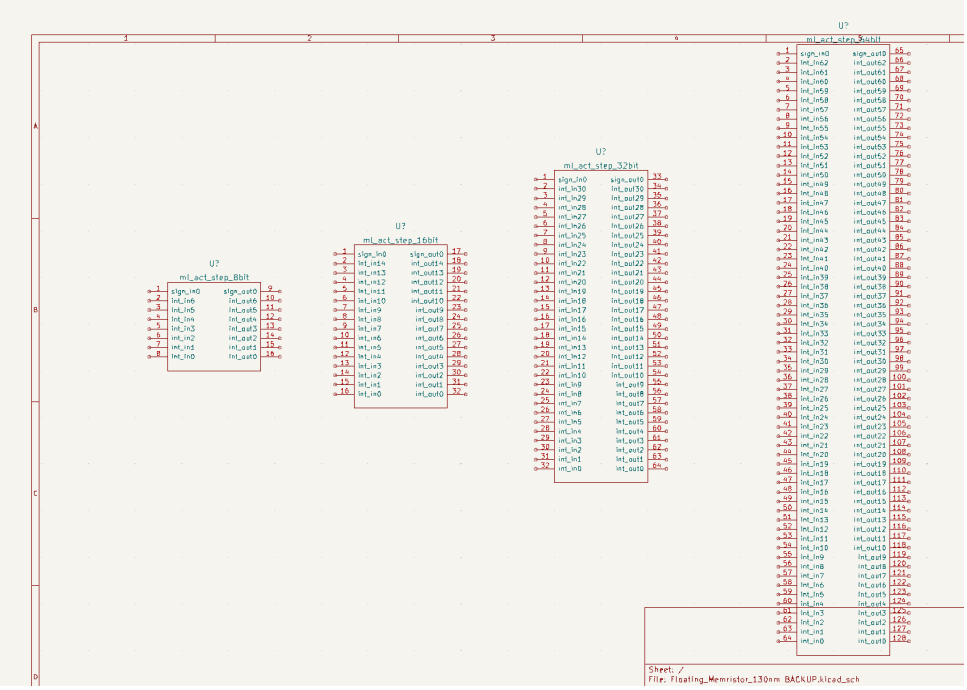


Figure 25: The generated KiCad schematic symbols for the Binary Step IP suite. Notice the absence of fractional pins, optimizing the block strictly for pure integer processing.

5.4 RTL Source Code

To bypass NgVeri parser limitations (which erroneously process inline Verilog comments as C++ variables), the RTL code was stripped of all inline documentation. Below are the clean, NgVeri-safe Verilog implementations for all four variants alongside the Master Testbench.

Listing 8: Binary Step IPs (8, 16, 32, 64-bit) Verilog Source Code

```
// ------------------------------------------------------
`timescale 1ns / 1ps

module ml_act_step_8bit (
    input  wire       sign_in,
    input  wire [6:0] int_in,
    output reg        sign_out,
    output reg  [6:0] int_out
);

    always @(*) begin
        sign_out = 1'b0;

        if (sign_in == 1'b0) begin
            int_out = 7'd1;
        end else begin
            int_out = 7'd0;
        end
```

```
    end

endmodule
// ------------------------------------------------------------

// ------------------------------------------------------------
`timescale 1ns / 1ps

module ml_act_step_16bit (
    input  wire        sign_in,
    input  wire [14:0] int_in,
    output reg         sign_out,
    output reg  [14:0] int_out
);

    always @(*) begin
        sign_out = 1'b0;

        if (sign_in == 1'b0) begin
            int_out = 15'd1;
        end else begin
            int_out = 15'd0;
        end
    end

endmodule// ------------------------------------------------------------

// ------------------------------------------------------------
`timescale 1ns / 1ps

module ml_act_step_32bit (
    input  wire        sign_in,
    input  wire [30:0] int_in,
    output reg         sign_out,
    output reg  [30:0] int_out
);

    always @(*) begin
        sign_out = 1'b0;

        if (sign_in == 1'b0) begin
            int_out = 31'd1;
        end else begin
            int_out = 31'd0;
        end
    end

endmodule// ------------------------------------------------------------

// ------------------------------------------------------------
`timescale 1ns / 1ps

module ml_act_step_64bit (
    input  wire        sign_in,
    input  wire [62:0] int_in,
    output reg         sign_out,
    output reg  [62:0] int_out
);

    always @(*) begin
        sign_out = 1'b0;

        if (sign_in == 1'b0) begin
            int_out = 63'd1;
        end else begin
            int_out = 63'd0;
        end
    end

endmodule// ------------------------------------------------------------
```

Listing 9: Master Testbench for Concurrent Binary Step Verification

```
// ------------------------------------------------------
`timescale 1ns / 1ps

module tb_ml_act_step_master;

    // ========================================
    // 1. INPUT REGISTERS (Stimulus)
    // ========================================
    reg        sign_in;
    reg [62:0] int_in; // Scaled to max width (63 bits) for stimulus injection

    // ========================================
    // 2. OUTPUT WIRES (Observation)
    // ========================================
    wire        s8, s16, s32, s64;
    wire [6:0]  i8;
    wire [14:0] i16;
    wire [30:0] i32;
    wire [62:0] i64;

    // ========================================
    // 3. INSTANTIATE UUTs (Unit Under Test)
    // ========================================
    ml_act_step_8bit uut_8 (
        .sign_in(sign_in), .int_in(int_in[6:0]),
        .sign_out(s8), .int_out(i8)
    );

    ml_act_step_16bit uut_16 (
        .sign_in(sign_in), .int_in(int_in[14:0]),
        .sign_out(s16), .int_out(i16)
    );

    ml_act_step_32bit uut_32 (
        .sign_in(sign_in), .int_in(int_in[30:0]),
        .sign_out(s32), .int_out(i32)
    );

    ml_act_step_64bit uut_64 (
        .sign_in(sign_in), .int_in(int_in),
        .sign_out(s64), .int_out(i64)
    );

    // ========================================
    // 4. VERIFICATION LOGIC & TABLE FORMATTING
    // ========================================
    integer pass_cnt = 0;
    integer fail_cnt = 0;

    task run_test;
        input [8*25:1] name;
        input exp_val; // 1 for Output=1, 0 for Output=0

        reg ok8, ok16, ok32, ok64, all_ok;
        begin
            #10; // Wait for combinational logic

            // Check outputs against expected (Check magnitude AND ensure sign_out is strictly 0)
            ok8  = (s8 == 0)  && (i8  == exp_val);
            ok16 = (s16 == 0) && (i16 == exp_val);
            ok32 = (s32 == 0) && (i32 == exp_val);
            ok64 = (s64 == 0) && (i64 == exp_val);

            all_ok = ok8 & ok16 & ok32 & ok64;

            // Formatted Data-Logging Row
            $display("| %-25s |   %b   | %16x |    %0d     |   %0d   |    %0d   |    %0d   |    %0d   |
    ↪ %-6s |",
                     name, sign_in, int_in, exp_val, i8, i16, i32, i64, all_ok ? "PASS" : "FAIL");

            if (all_ok) pass_cnt = pass_cnt + 1;
            else fail_cnt = fail_cnt + 1;
```

```
        end
    endtask

    // ========================================
    // 5. STIMULUS INJECTION
    // ========================================
    initial begin
        $display("\n================================================================================================
    ↪ ======================");
        $display(" BINARY STEP MASTER VERIFICATION SUITE (Pure Integer Logic)");
        $display("================================================================================================
    ↪ ====================");
        $display("| Test Name                 | Sign | Int Magnitude    | Expected | 8-bit | 16-bit |
    ↪ 32-bit | 64-bit | Status |");
        $display("|---------------------------|------|------------------|----------|-------|--------|--
    ↪ ------|--------|--------|");

        // --- POSITIVE & ZERO TESTS (Expect 1) ---
        sign_in = 0; int_in = 63'h0;
        run_test("TC1: Zero (0)", 1);

        sign_in = 0; int_in = 63'h1;
        run_test("TC2: Small Pos (+1)", 1);

        sign_in = 0; int_in = 63'h32; // Decimal 50
        run_test("TC3: Standard Pos (+50)", 1);

        sign_in = 0; int_in = 63'h7F; // Max 8-bit integer
        run_test("TC4: Max 8-bit Pos", 1);

        sign_in = 0; int_in = 63'h7FFF; // Max 16-bit integer
        run_test("TC5: Max 16-bit Pos", 1);

        sign_in = 0; int_in = 63'h5555_5555_5555_5555; // VLSI Alternating wire test
        run_test("TC6: Alt Bit Pattern Pos", 1);

        sign_in = 0; int_in = 63'h7FFF_FFFF_FFFF_FFFF; // Absolute Max 64-bit noise
        run_test("TC7: Massive Pos Noise", 1);

        // --- NEGATIVE TESTS (Expect 0) ---
        sign_in = 1; int_in = 63'h1;
        run_test("TC8: Small Neg (-1)", 0);

        sign_in = 1; int_in = 63'h32; // Decimal 50
        run_test("TC9: Standard Neg (-50)", 0);

        sign_in = 1; int_in = 63'h7F;
        run_test("TC10: Max 8-bit Neg", 0);

        sign_in = 1; int_in = 63'h7FFF;
        run_test("TC11: Max 16-bit Neg", 0);

        sign_in = 1; int_in = 63'h2AAA_AAAA_AAAA_AAAA; // VLSI Alternating wire test
        run_test("TC12: Alt Bit Pattern Neg", 0);

        sign_in = 1; int_in = 63'h7FFF_FFFF_FFFF_FFFF; // Absolute Max 64-bit noise
        run_test("TC13: Massive Neg Noise", 0);

        $display("================================================================================================
    ↪ ====================");
        $display(" TOTAL CHECKS PASSED: %0d/13", pass_cnt);
        $display(" TOTAL CHECKS FAILED: %0d/13", fail_cnt);
        $display("================================================================================================
    ↪ ====================\n");
        $finish;
    end
endmodule
// ------------------------------------------------------
```

5.5 Verification Suite Phase 1: Pure Digital (Vivado XSim)

To ensure the hardware operates as a flawless discriminator independent of input magnitude bloat, a Master Testbench was engineered to inject massive edge-case noise vectors concurrently into all four variants.

The testbench executed 13 exhaustive logic checks, including:

- **Baseline & Boundary Checks:** Sweeping zeroes, standard decimal values (e.g., 50), and maximum bus limits (`0x7F`, `0x7FFF`).
- **Wire Fault Testing:** Injecting alternating bit-patterns (`0x555...` and `0xAAA...`) to ensure no internal integer bits influence the boolean threshold.
- **Maximum 64-bit Noise:** Injecting `0x7FFFFFFFFFFFFFFF` to verify absolute immunity to arithmetic overflow.

```
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#     send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window. If you want to open a wave window go to 'Fi
#   }
# }
# run 1000ns

=============================================================================================
 BINARY STEP MASTER VERIFICATION SUITE (Pure Integer Logic)
=============================================================================================
| Test Name                 | Sign | Int Magnitude    | Expected | 8-bit | 16-bit | 32-bit | 64-bit | Status |
|---------------------------|------|------------------|----------|-------|--------|--------|--------|--------|
|            TC1: Zero (0)  |   0  | 0000000000000000 |    1     |   1   |    1   |    1   |    1   |   PASS |
|       TC2: Small Pos (+1) |   0  | 0000000000000001 |    1     |   1   |    1   |    1   |    1   |   PASS |
|   TC3: Standard Pos (+50) |   0  | 0000000000000032 |    1     |   1   |    1   |    1   |    1   |   PASS |
|       TC4: Max 8-bit Pos  |   0  | 000000000000007f |    1     |   1   |    1   |    1   |    1   |   PASS |
|       TC5: Max 16-bit Pos |   0  | 0000000000007fff |    1     |   1   |    1   |    1   |    1   |   PASS |
|  TC6: Alt Bit Pattern Pos |   0  | 5555555555555555 |    1     |   1   |    1   |    1   |    1   |   PASS |
|    TC7: Massive Pos Noise |   0  | 7fffffffffffffff |    1     |   1   |    1   |    1   |    1   |   PASS |
|       TC8: Small Neg (-1) |   1  | 0000000000000001 |    0     |   0   |    0   |    0   |    0   |   PASS |
|   TC9: Standard Neg (-50) |   1  | 0000000000000032 |    0     |   0   |    0   |    0   |    0   |   PASS |
|       TC10: Max 8-bit Neg |   1  | 000000000000007f |    0     |   0   |    0   |    0   |    0   |   PASS |
|      TC11: Max 16-bit Neg |   1  | 0000000000007fff |    0     |   0   |    0   |    0   |    0   |   PASS |
| TC12: Alt Bit Pattern Neg |   1  | 2aaaaaaaaaaaaaaa |    0     |   0   |    0   |    0   |    0   |   PASS |
|   TC13: Massive Neg Noise |   1  | 7fffffffffffffff |    0     |   0   |    0   |    0   |    0   |   PASS |
=============================================================================================
 TOTAL CHECKS PASSED: 13/13
 TOTAL CHECKS FAILED: 0/13
=============================================================================================

$finish called at time : 130 ns : File "C:/Users/Apple/Documents/esim_design/internship/Digital IP Design/3. Step/tb_ml_act_step_master.v" Line 129
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_ml_act_step_master_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:03 ; elapsed = 00:00:08 . Memory (MB): peak = 1268.730 ; gain = 0.000
```

Figure 26: The formatted ASCII data-logging table from the Vivado TCL console. The output proves the logic maps all positive/zero data states to exactly +1, and all negative data states to 0, achieving a 100% pass rate across 13 extreme stimulus conditions.

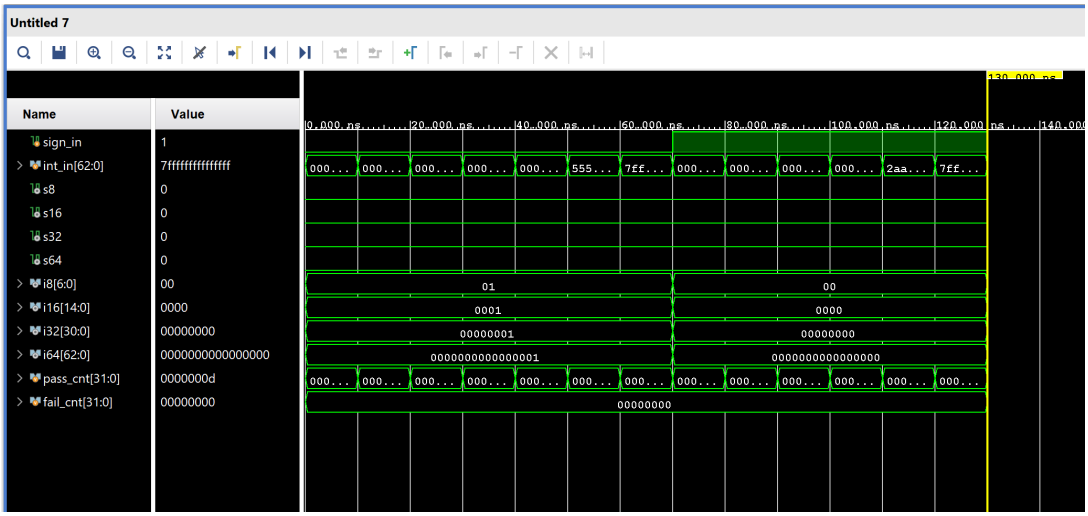


Figure 27: The simulated digital waveform from Vivado, verifying that the IP acts as a zero-delay binary discriminator evaluating solely on the MSB sign state.

5.6 Verification Suite Phase 2: Mixed-Signal Automation (Ngspice)

To validate the pure-integer Verilator C++ model against continuous analog dynamics, the IP suite was subjected to targeted mixed-signal transient analyses within eSim. The testing methodology utilized Piece-Wise Linear (PWL) analog voltage sources to dynamically mutate the MSB while the simulation ran.

5.6.1 Dynamic PWL Stimulus Injection

The test circuit was configured to push the ADC logic thresholds by utilizing $8V$ high-state logic pulses:

- **0ms to 10ms (Positive Pass):** The `sign_in` bit was held at $0V$ (Positive), while the integer input pins were stimulated with an $8V$ PWL pulse. The expected output is a discrete +1 (Logic HIGH on output bit 0).
- **10ms onwards (Negative Clamp):** The `sign_in` bit abruptly transitions to $8V$ (Negative), while the integer buses drop to $0V$. The expected output is an instant collapse to 0 on all output pins.

To clearly visualize the multi-bit output without traces overlapping and becoming unreadable in the Ngspice graph, a Stacked Logic Analyzer script was executed via the `.control` block. Each output bit was assigned a mathematical voltage offset (in increments of +6V), neatly stacking the continuous analog traces vertically.

5.6.2 8-Bit Variant Mixed-Signal Verification

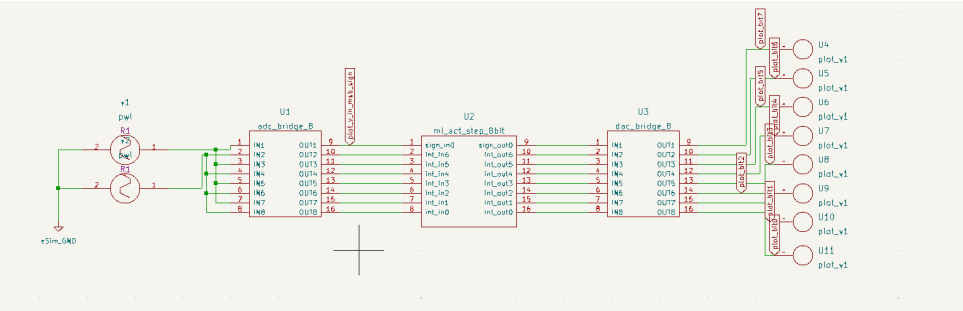


Figure 28: The eSim KiCad schematic for the 8-bit Binary Step mixed-signal test. ADC and DAC bridges facilitate the translation between the 8V PWL analog sources and the digital C++ thresholding core.

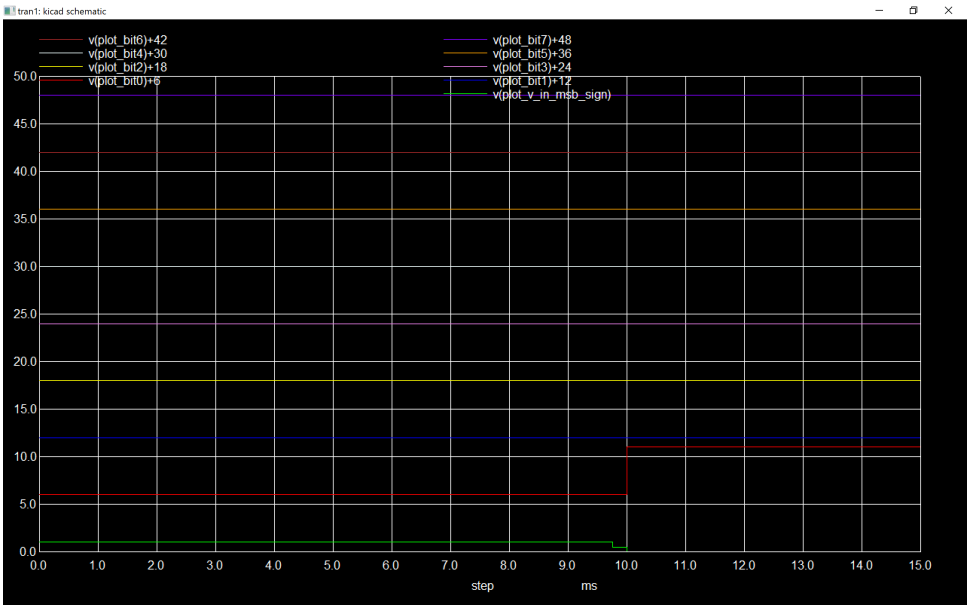


Figure 29: The stacked Ngspice logic analyzer waveform for the 8-bit variant. The output trace holds at a steady Logic 1 ($5V$) during the positive data phase, and instantly zeroes out at the 10ms boundary when the analog sign bit asserts a negative state.

5.6.3 16-Bit Variant Mixed-Signal Verification

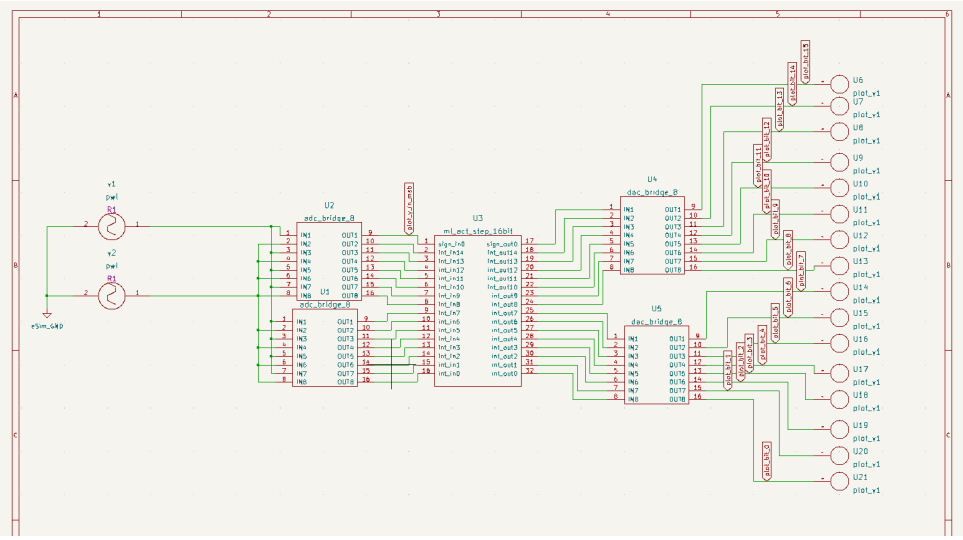


Figure 30: The expanded eSim KiCad schematic for the 16-bit Binary Step, accurately routing 16 input ADCs and 16 output DACs into the Ngspice matrix.

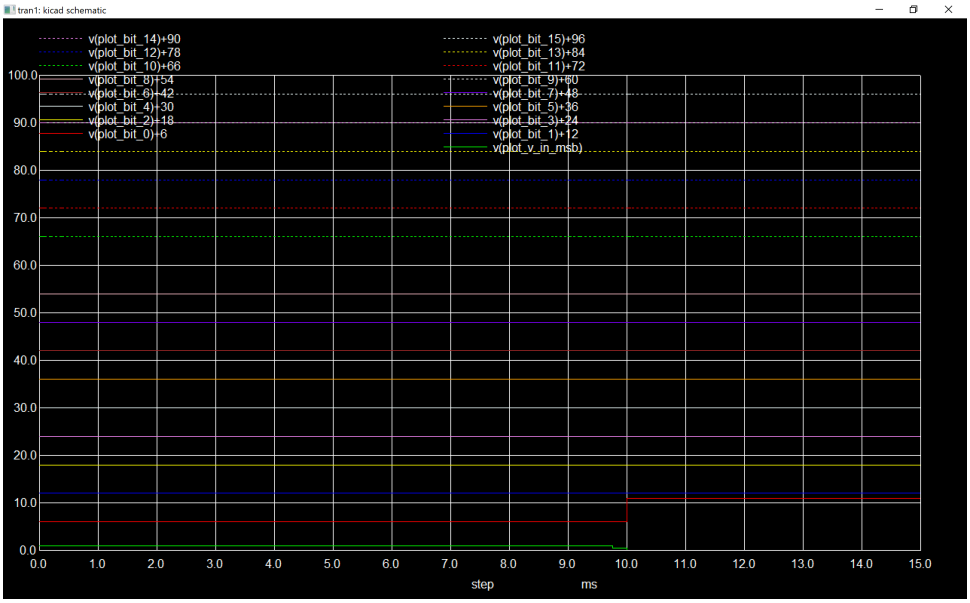


Figure 31: The stacked Ngspice waveform for the 16-bit variant. The +6V incremental offset stacking cleanly proves the 16-bit bus clamping capability without visual signal interference.

5.6.4 32-Bit Variant Mixed-Signal Verification

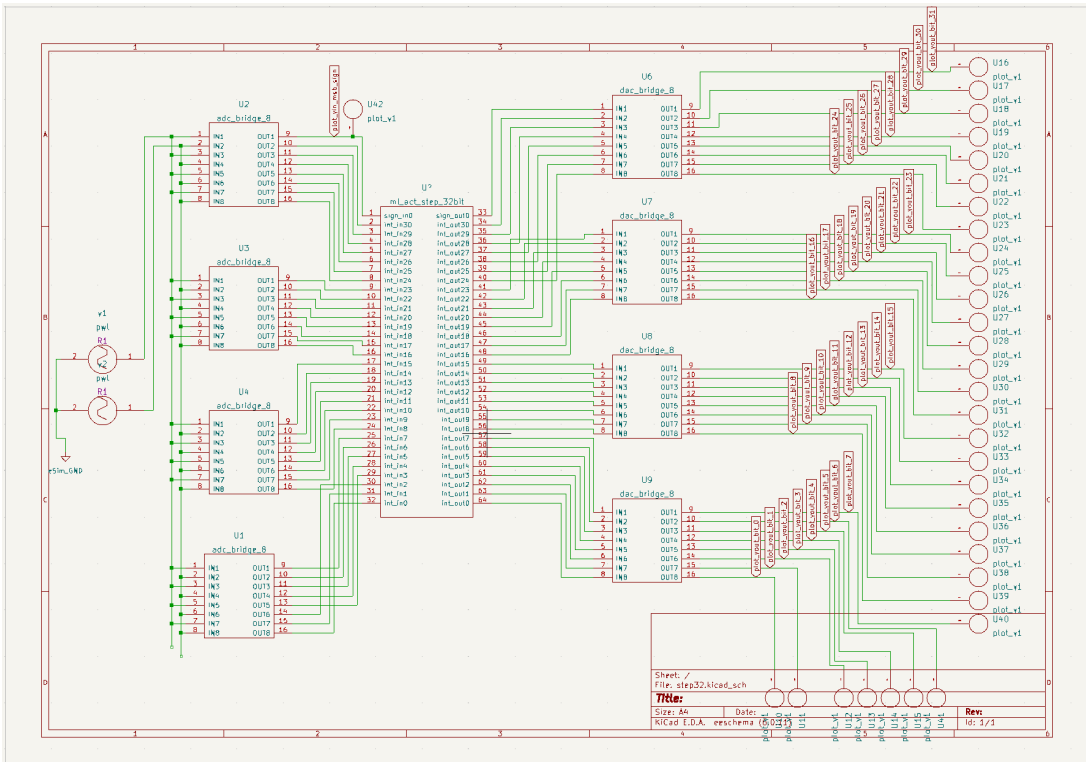


Figure 32: The dense eSim KiCad schematic required to test the 32-bit Binary Step variant, demonstrating successful wide-bus routing in the eSim graphical interface.

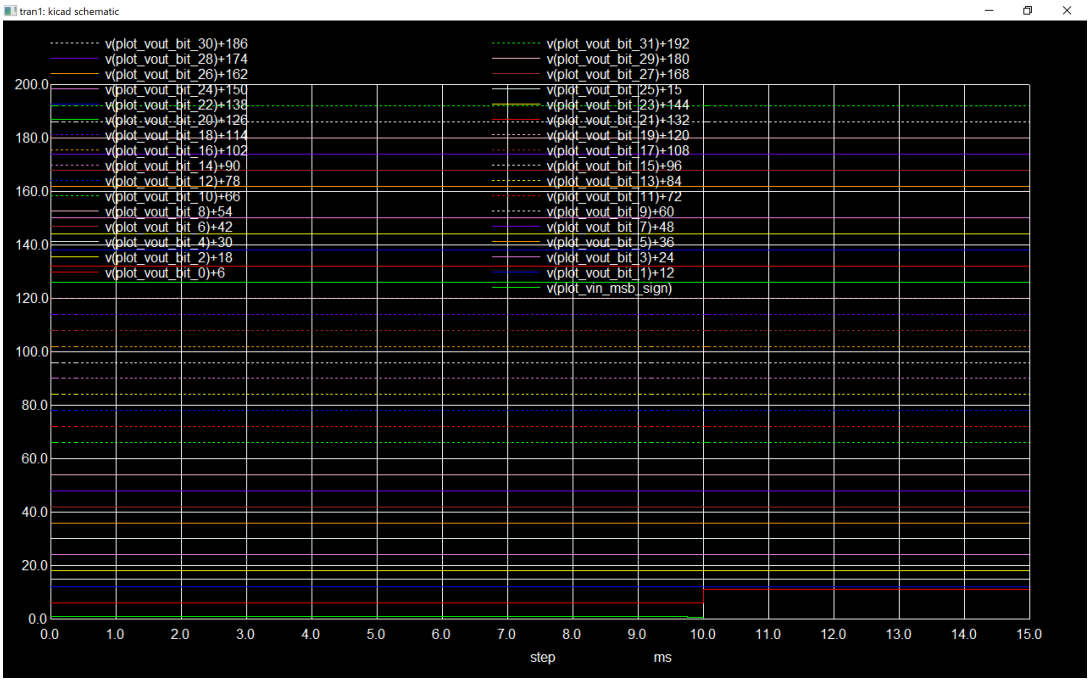


Figure 33: The monumental 32-bit stacked Ngspice waveform. Even with 32 parallel digital traces calculated in the analog domain, the hardware reliably clamps the entire data bus to 0V upon negative activation.

5.6.5 64-Bit Variant Mixed-Signal Verification

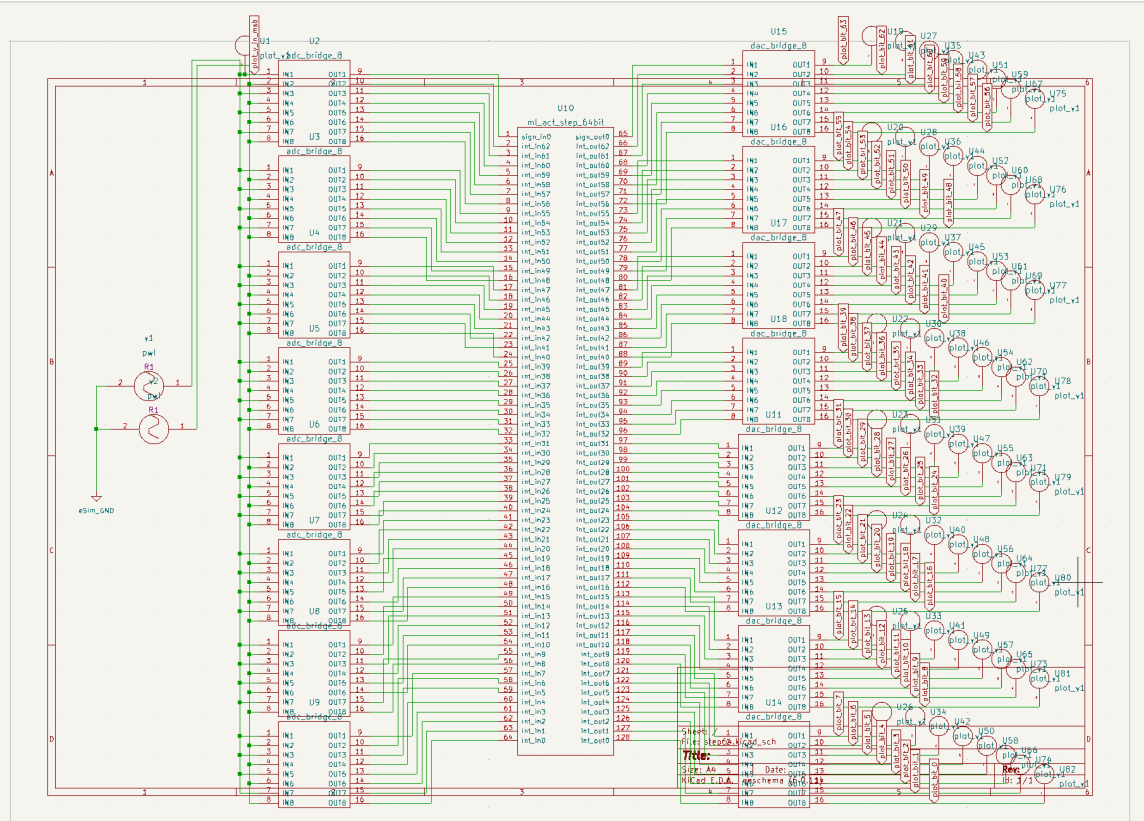


Figure 34: The extreme 64-bit eSim KiCad schematic. This test successfully bridged 128 continuous analog nodes to a single 64-bit digital combinatorial block utilizing the 8V PWL testing methodology.

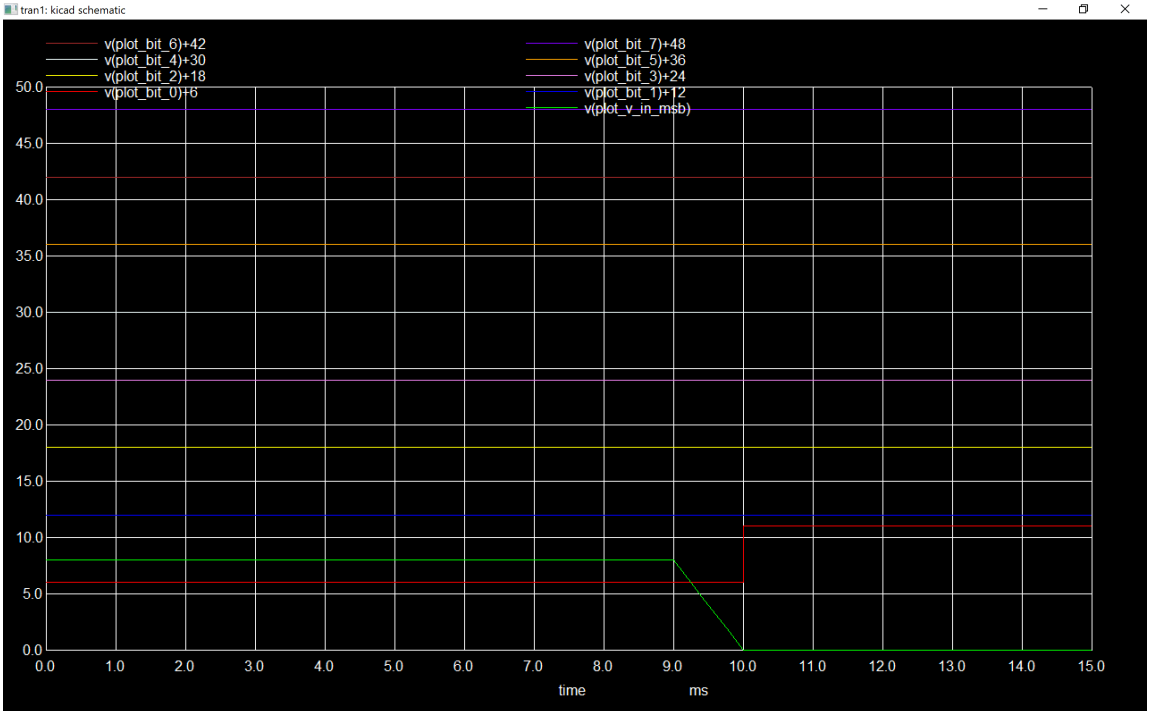


Figure 35: The 64-bit stacked Ngspice waveform. The thresholding logic executes flawlessly across the entire 64-bit parallel bus, validating the maximum bit-width variant in the analog domain.

5.7 Conclusion

The Binary Step IP suite was successfully engineered with a highly optimized, fractional-free architecture. By eliminating redundant Q-format mapping, the block achieves a minimal schematic footprint and incredibly fast SPICE convergence times. Rigorous Vivado regression testing and dynamic Ngspice mixed-signal verification confirmed that the 8, 16, 32, and 64-bit IPs operate as perfectly immune boolean discriminators, ready for integration into eSim neural networks and logic triggers.

6 Digital IP Datasheet & Verification Report: Universal Up/Down Counter

6.1 Introduction & Purpose

The counter is the single most frequently instantiated sequential element in digital design, yet the eSim library offered no general-purpose one. The bundled digital primitives provide flip-flops and a frequency divider (`d_fdiv`), but a divider produces only a square wave at a submultiple frequency — it cannot report *how many* events have occurred, cannot be preset to a starting value, and cannot count backwards.

The **Universal Up/Down Counter** fills that gap with the feature set of the classic 74 x 193 / 74 x 190 family in a single NgVeri-compatible core: bidirectional counting, synchronous parallel load, a count enable that freezes the state, selectable **binary or BCD** counting, and dedicated wrap outputs. In a mixed-signal context it serves as an event counter for comparator output pulses, a programmable timebase, or the digit source for the seven-segment driver documented later in this report.

6.2 Architecture & Pin Specifications

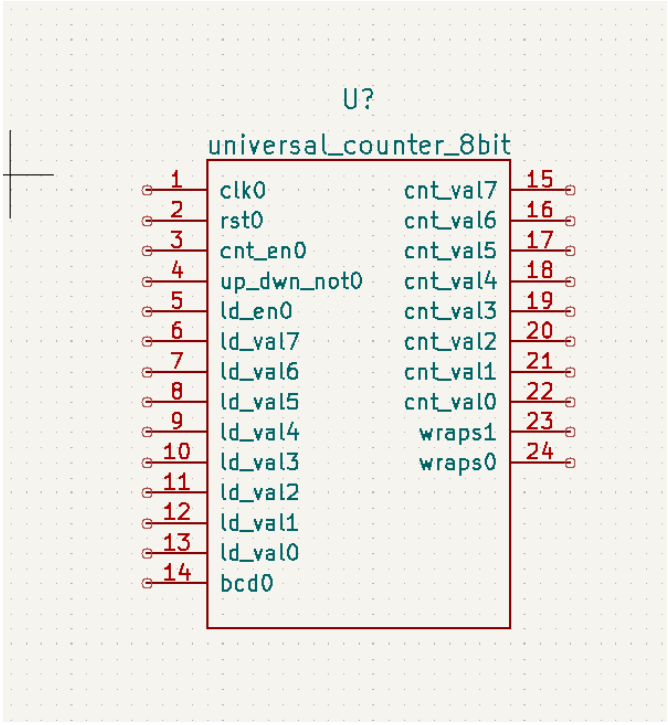


Figure 36: The KiCad schematic symbol generated by NgVeri for `universal_counter_8bit`. NgVeri flattens the vector ports into individual pins, giving 14 inputs (`clk`, `rst`, `cnt_en`, `up_dwn_not`, `ld_en`, `ld_val7..0`, `bcd`) and 10 outputs (`cnt_val7..0`, `wraps1..0`).

Port	Width	Direction	Function
clk	1	in	Rising-edge clock; all state changes are synchronous to it
rst	1	in	Synchronous reset; forces the count to 0x00
cnt_en	1	in	Count enable; when low the counter holds its value
up_dwn_not	1	in	Direction: 1 counts up, 0 counts down
ld_en	1	in	Parallel-load enable; loads ld_val on the next edge
ld_val	8	in	Preset value applied when ld_en is high
bcd	1	in	Mode: 0 = binary (00–FF), 1 = packed BCD (00–99)
cnt_val	8	out	Current count value
wraps	2	out	wraps[0] = overflow pulse, wraps[1] = underflow pulse

Table 14: Port specification of the Universal Up/Down Counter.

6.3 Working Principle

6.3.1 Control Priority

The three control inputs are resolved in a strict priority chain inside a single `always @(posedge clk)` block: `rst` overrides `ld_en`, which in turn overrides `cnt_en`. This ordering is the conventional one for loadable counters and guarantees that a reset can never be masked by a simultaneous load request. Because the reset is **synchronous**, the block contains exactly one clock-edge sensitivity and no asynchronous control path — a deliberate choice, since Verilator models zero-delay combinational feedback poorly and eSim users rarely have a clean asynchronous reset source available in an analog schematic.

6.3.2 Binary Mode

With `bcd` low the counter is a plain modulo-256 up/down counter. Counting up from `0xFF` loads `0x00` and simultaneously asserts `wraps[0]`; counting down from `0x00` loads `0xFF` and asserts `wraps[1]`.

6.3.3 BCD (Decade) Mode

With `bcd` high the eight bits are reinterpreted as **two packed binary-coded-decimal digits**, so the legal range becomes `0x00`–`0x99` and each nibble is constrained to 0–9. The increment logic is a two-level decade cascade:

- if the value is `0x99`, wrap to `0x00` and pulse `wraps[0]`;
- else if the low nibble is 9, clear it and increment the high nibble (decade carry);
- else increment the low nibble.

Decrementing mirrors this exactly: `0x00` wraps to `0x99` with `wraps[1]`; a low nibble of 0 becomes 9 with a decade borrow; otherwise the low nibble decrements. The illegal codes `0x0A`–`0x0F` (and the corresponding high-nibble cases) are unreachable from any legal starting value, and the mode is directly useful in eSim: BCD digits feed the seven-segment driver without any binary-to-decimal conversion stage.

6.3.4 Wrap-Pulse Semantics

Both wrap outputs are **registered single-cycle pulses**, not sticky flags. The register is cleared unconditionally at the top of every clock evaluation and set only in the cycle in which the boundary is crossed. This makes the outputs directly usable as a carry into a second counter stage, and it is what the Phase-1 testbench checks in TC11/TC12 and TC15/TC16: the pulse must appear in exactly one cycle and be gone in the next.

6.4 RTL Source Code

Listing 10: Universal Up/Down Counter RTL (`universal_counter_8bit.v`)

```
`timescale 1ns / 1ps

module universal_counter_8bit (
    input  clk,
    input  rst,
    input  cnt_en,
    input  up_dwn_not,
    input  ld_en,
    input  [7:0] ld_val,
    input  bcd,
    output [7:0]  cnt_val,
    output [1:0]  wraps
);

    reg [7:0]  cnt_val_reg;
    reg [1:0]  wraps_reg;


    always @(posedge clk) begin
        wraps_reg <= 2'b00;

        if (rst) begin
            cnt_val_reg <= 8'd0;
        end else if (ld_en) begin
            cnt_val_reg <= ld_val;
        end else if (cnt_en) begin
            if (bcd) begin
                if (up_dwn_not) begin
                    if (cnt_val_reg == 8'h99) begin
                        cnt_val_reg <= 8'h00;
                        wraps_reg[0] <= 1'b1;
                    end else if (cnt_val_reg[3:0] == 4'd9) begin
                        cnt_val_reg[3:0] <= 4'd0;
                        cnt_val_reg[7:4] <= cnt_val_reg[7:4] + 4'd1;
                    end else begin
                        cnt_val_reg[3:0] <= cnt_val_reg[3:0] + 4'd1;
                    end
                end else begin
                    if (cnt_val_reg == 8'h00) begin
                        cnt_val_reg <= 8'h99;
                        wraps_reg[1] <= 1'b1;
                    end else if (cnt_val_reg[3:0] == 4'd0) begin
                        cnt_val_reg[3:0] <= 4'd9;
                        cnt_val_reg[7:4] <= cnt_val_reg[7:4] - 4'd1;
                    end else begin
                        cnt_val_reg[3:0] <= cnt_val_reg[3:0] - 4'd1;
                    end
                end
            end else begin
                if (up_dwn_not) begin
                    if (cnt_val_reg == 8'hFF) begin
                        cnt_val_reg <= 8'h00;
                        wraps_reg[0] <= 1'b1;
                    end else begin
                        cnt_val_reg <= cnt_val_reg + 8'd1;
                    end
```

```
                end else begin
                    if (cnt_val_reg == 8'h00) begin
                        cnt_val_reg <= 8'hFF;
                        wraps_reg[1] <= 1'b1;
                    end else begin
                        cnt_val_reg <= cnt_val_reg - 8'd1;
                    end
                end
            end
        end
    end

    assign cnt_val = cnt_val_reg;
    assign wraps = wraps_reg;
endmodule
```

6.5 Self-Checking Testbench

The testbench walks the counter through a directed 24-step scenario. A single `step_check` task advances one clock, then compares the count value **and** both wrap pulses against the expected triple, printing one table row per check.

Listing 11: Universal Up/Down Counter self-checking testbench (`tb_universal_counter_8bit.v`)

```
`timescale 1ns / 1ps

module tb_universal_counter_8bit;

    reg        clk;
    reg        rst;
    reg        cnt_en;
    reg        up_dwn_not;
    reg        ld_en;
    reg  [7:0] ld_val;
    reg        bcd;
    wire [7:0] cnt_val;
    wire [1:0] wraps;


    integer pass_cnt = 0;
    integer fail_cnt = 0;

    universal_counter_8bit uut (
        .clk(clk),
        .rst(rst),
        .cnt_en(cnt_en),
        .up_dwn_not(up_dwn_not),
        .ld_en(ld_en),
        .ld_val(ld_val),
        .bcd(bcd),
        .cnt_val(cnt_val),
        .wraps(wraps)

    );

    always #5 clk = ~clk;

    // Advance one clk, then compare count value and both wrap pulses
    task step_check;
        input [8*30:1] test_name;
        input [7:0]    expected_count;
        input          expected_overflow;
        input          expected_underflow;
        reg ok;
        begin
            @(posedge clk);
            #1;
            ok = (cnt_val === expected_count) &&
```

```
                (wraps[0] === expected_overflow) &&
                (wraps[1] === expected_underflow);
            $display("| %-30s |  %02h  |    %02h    |  %b/%b  |  %b/%b  | %-6s |",
                     test_name, expected_count, cnt_val,
                     expected_overflow, wraps[0],
                     expected_underflow, wraps[1],
                     ok ? "PASS" : "FAIL");
            if (ok) pass_cnt = pass_cnt + 1;
            else    fail_cnt = fail_cnt + 1;
        end
    endtask

    initial begin
$dumpfile("sim_out.vcd");

$dumpvars(0, tb_universal_counter_8bit);

        clk = 0; rst = 1; cnt_en = 0; up_dwn_not = 1;
        ld_en = 0; ld_val = 8'd0; bcd = 0;

        $display("\n==========================================================================================
    ↪ ==");
        $display(" UNIVERSAL UP/DOWN COUNTER (8-BIT, BINARY + BCD) VERIFICATION SUITE");
        $display("==========================================================================================
    ↪ ");
        $display("| Test Name                      | Exp  | Actual   | OV e/a | UN e/a | Status |");
        $display("|--------------------------------|------|----------|--------|--------|--------|");

        step_check("TC1: rst holds zero", 8'h00, 0, 0);
        @(negedge clk); rst = 0;

        @(negedge clk); ld_en = 1; ld_val = 8'h37;
        step_check("TC2: Parallel load 0x37", 8'h37, 0, 0);

        @(negedge clk); ld_en = 0; cnt_en = 1; up_dwn_not = 1;
        step_check("TC3: Count up (1)", 8'h38, 0, 0);
        step_check("TC4: Count up (2)", 8'h39, 0, 0);
        step_check("TC5: Count up (3)", 8'h3A, 0, 0);

        @(negedge clk); up_dwn_not = 0;
        step_check("TC6: Count down (1)", 8'h39, 0, 0);
        step_check("TC7: Count down (2)", 8'h38, 0, 0);

        @(negedge clk); cnt_en = 0;
        step_check("TC8: Hold when disabled", 8'h38, 0, 0);

        @(negedge clk); ld_en = 1; ld_val = 8'hFE;
        step_check("TC9: Load 0xFE", 8'hFE, 0, 0);
        @(negedge clk); ld_en = 0; cnt_en = 1; up_dwn_not = 1;
        step_check("TC10: Up to 0xFF", 8'hFF, 0, 0);
        step_check("TC11: Binary overflow wrap", 8'h00, 1, 0);
        step_check("TC12: Overflow pulse clears", 8'h01, 0, 0);

        @(negedge clk); ld_en = 1; ld_val = 8'h01;
        step_check("TC13: Load 0x01", 8'h01, 0, 0);
        @(negedge clk); ld_en = 0; up_dwn_not = 0;
        step_check("TC14: Down to 0x00", 8'h00, 0, 0);
        step_check("TC15: Binary underflow wrap", 8'hFF, 0, 1);
        step_check("TC16: Underflow pulse clears", 8'hFE, 0, 0);

        @(negedge clk); bcd = 1; ld_en = 1; ld_val = 8'h29;
        step_check("TC17: BCD load 29", 8'h29, 0, 0);
        @(negedge clk); ld_en = 0; up_dwn_not = 1;
        step_check("TC18: BCD digit carry 29->30", 8'h30, 0, 0);

        @(negedge clk); ld_en = 1; ld_val = 8'h98;
        step_check("TC19: BCD load 98", 8'h98, 0, 0);
        @(negedge clk); ld_en = 0;
        step_check("TC20: BCD up to 99", 8'h99, 0, 0);
        step_check("TC21: BCD overflow 99->00", 8'h00, 1, 0);

        @(negedge clk); up_dwn_not = 0;
        step_check("TC22: BCD underflow 00->99", 8'h99, 0, 1);
```

```
116         step_check("TC23: BCD digit borrow 99->98", 8'h98, 0, 0);
117         step_check("TC24: BCD down 98->97", 8'h97, 0, 0);
118
119         $display("================================================================================
    ↪ ");
120         $display(" TOTAL CHECKS PASSED: %0d/%0d", pass_cnt, pass_cnt + fail_cnt);
121         $display(" TOTAL CHECKS FAILED: %0d/%0d", fail_cnt, pass_cnt + fail_cnt);
122         $display("================================================================================
    ↪ \n");
123         $finish;
124       end
125
126 endmodule
```

6.6 Verification Phase 1: Pure Digital (Vivado XSim)

Checks	Feature exercised	Key expectation
TC1	Synchronous reset	count held at `0x00`
TC2, TC9, TC13	Parallel load	`ld_val` appears on the next edge
TC3–TC5	Up-counting	`0x37` → `0x3A`
TC6–TC7	Down-counting	`0x3A` → `0x38`
TC8	Count enable	value frozen while `cnt_en` is low
TC10–TC12	Binary overflow	`0xFF` → `0x00`, `wraps[0]` pulses for one cycle only
TC14–TC16	Binary underflow	`0x00` → `0xFF`, `wraps[1]` pulses for one cycle only
TC17–TC18	BCD decade carry	`29` → `30`
TC19–TC21	BCD overflow	`99` → `00` with `wraps[0]`
TC22–TC24	BCD underflow and borrow	`00` → `99` → `98` → `97`

Table 15: Coverage of the 24 directed checks applied to the Universal Counter.

```
================================================================================
 UNIVERSAL UP/DOWN COUNTER (8-BIT, BINARY + BCD) VERIFICATION SUITE
================================================================================
| Test Name                      | Exp  | Actual   | OV e/a | UN e/a | Status |
|--------------------------------|------|----------|--------|--------|--------|
|            TC1: rst holds zero |  00  |    00    |  0/0   |  0/0   |   PASS |
|        TC2: Parallel load 0x37 |  37  |    37    |  0/0   |  0/0   |   PASS |
|              TC3: Count up (1) |  38  |    38    |  0/0   |  0/0   |   PASS |
|              TC4: Count up (2) |  39  |    39    |  0/0   |  0/0   |   PASS |
|              TC5: Count up (3) |  3a  |    3a    |  0/0   |  0/0   |   PASS |
|            TC6: Count down (1) |  39  |    39    |  0/0   |  0/0   |   PASS |
|            TC7: Count down (2) |  38  |    38    |  0/0   |  0/0   |   PASS |
|        TC8: Hold when disabled |  38  |    38    |  0/0   |  0/0   |   PASS |
|                TC9: Load 0xFE |  fe  |    fe    |  0/0   |  0/0   |   PASS |
|              TC10: Up to 0xFF |  ff  |    ff    |  0/0   |  0/0   |   PASS |
|     TC11: Binary overflow wrap |  00  |    00    |  1/1   |  0/0   |   PASS |
|    TC12: Overflow pulse clears |  01  |    01    |  0/0   |  0/0   |   PASS |
|               TC13: Load 0x01 |  01  |    01    |  0/0   |  0/0   |   PASS |
|             TC14: Down to 0x00 |  00  |    00    |  0/0   |  0/0   |   PASS |
|    TC15: Binary underflow wrap |  ff  |    ff    |  0/0   |  1/1   |   PASS |
|   TC16: Underflow pulse clears |  fe  |    fe    |  0/0   |  0/0   |   PASS |
|              TC17: BCD load 29 |  29  |    29    |  0/0   |  0/0   |   PASS |
|   TC18: BCD digit carry 29->30 |  30  |    30    |  0/0   |  0/0   |   PASS |
|              TC19: BCD load 98 |  98  |    98    |  0/0   |  0/0   |   PASS |
|             TC20: BCD up to 99 |  99  |    99    |  0/0   |  0/0   |   PASS |
|      TC21: BCD overflow 99->00 |  00  |    00    |  1/1   |  0/0   |   PASS |
|     TC22: BCD underflow 00->99 |  99  |    99    |  0/0   |  1/1   |   PASS |
|  TC23: BCD digit borrow 99->98 |  98  |    98    |  0/0   |  0/0   |   PASS |
|          TC24: BCD down 98->97 |  97  |    97    |  0/0   |  0/0   |   PASS |
================================================================================
 TOTAL CHECKS PASSED: 24/24
 TOTAL CHECKS FAILED: 0/24
================================================================================

$finish called at time : 246 ns : File "C:/Users/Apple/Documents/esim_design/internship/Digital_IP_Design/more_design/1_Universal_Counter/tb_universal_counter_8bit.v" Line
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_universal_counter_8bit_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:07 ; elapsed = 00:00:14 . Memory (MB): peak = 1279.988 ; gain = 0.000
update_compile_order -fileset sim_1
```

Figure 37: Vivado Tcl console output for the Universal Counter. All 24 directed checks report `PASS` (**24/24**), including the two single-cycle wrap-pulse assertions and the full BCD decade-carry and borrow sequence.

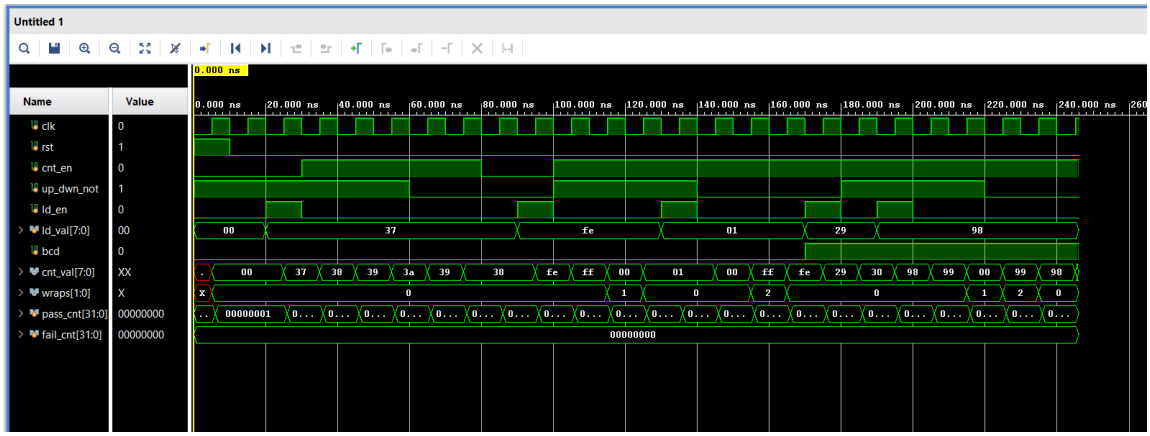


Figure 38: Vivado XSim waveform for the Universal Counter, showing the synchronous load of the preset value, the change of direction, and the wrap boundaries at which `wraps[0]` and `wraps[1]` pulse for exactly one clock cycle.

6.7 Verification Phase 2: Mixed-Signal (eSim / Ngspice)

The mixed-signal test was designed to exercise the most characteristic behaviour of the core — **preset, count, and wrap** — inside a single short transient.

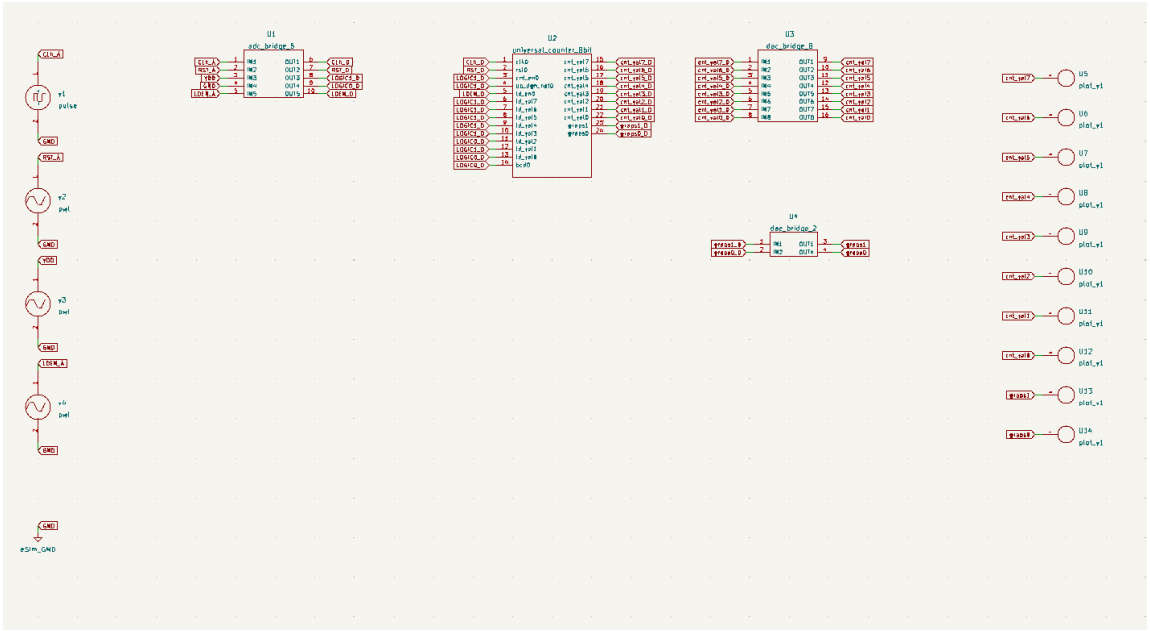


Figure 39: The generated eSim test schematic for the Universal Counter. Five analog nets (`CLK`, `RST`, `VDD`, `GND`, `LDEN`) enter through an `adc_bridge` bank; the ten digital outputs leave through `dac_bridge` banks to individual `plot_v1` probes.

6.7.1 Stimulus Definition

Net	Source	Value	Purpose
CLK_A	pulse	0 5 0 1u 1u 0.5m 1m	1 kHz clock (rising edges at 0, 1, 2, ... ms)
RST_A	pwl	0 5 1.9m 5 2m 0	reset asserted until 2 ms
VDD	pwl	0 5 100 5	logic-1 rail (cnt_en, up_dwn_not)
LDEN_A	pwl	0 0 2.4m 0 2.5m 5 3.5m 5 3.6m 0	load window spanning the 3 ms edge
GND	—	0 V	bcd = 0 (binary mode) and the zero bits of ld_val

Table 16: Analog stimulus for the Universal Counter transient. Transient settings: step 10 μs, stop 15 ms.

The preset value is wired as a constant ld_val = **250** (0xFA) by tying the appropriate ld_val pins to VDD and the remainder to GND. Loading 250 places the counter only five counts below its binary maximum, so the wrap boundary is reached quickly and the whole preset-count-wrap sequence fits in a 15 ms window.

6.7.2 Expected Timeline

Time	Event	Expected cnt_val
0–2 ms	reset asserted	0x00
3 ms	ld_en high at the clock edge	0xFA (250)
4–8 ms	counting up	251, 252, 253, 254, 255
9 ms	binary overflow	0x00, wraps[0] pulses
10–15 ms	counting continues	1, 2, 3, ...

Table 17: Predicted behaviour of the Universal Counter during the 15 ms transient.

6.7.3 Results

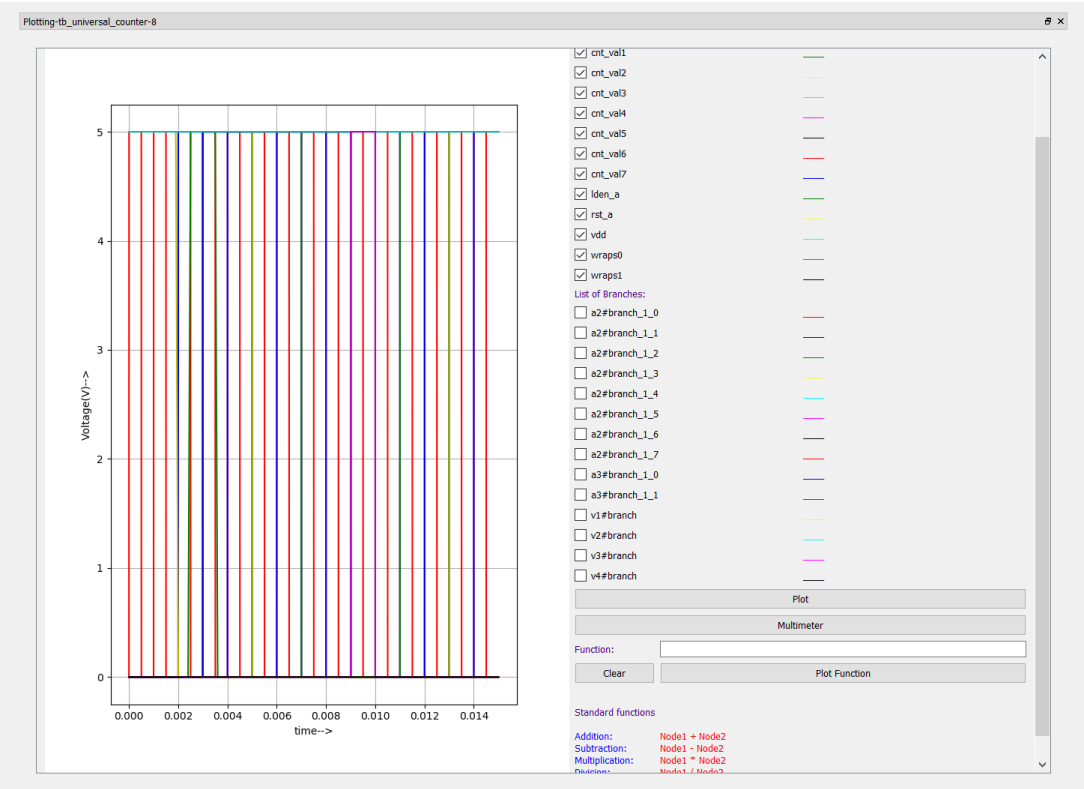


Figure 40: The eSim plotting window after the Universal Counter transient, listing every captured node. All ten digital outputs and the four analog stimulus nets are available for inspection; the raw overlaid traces switch cleanly between 0 V and 5 V.

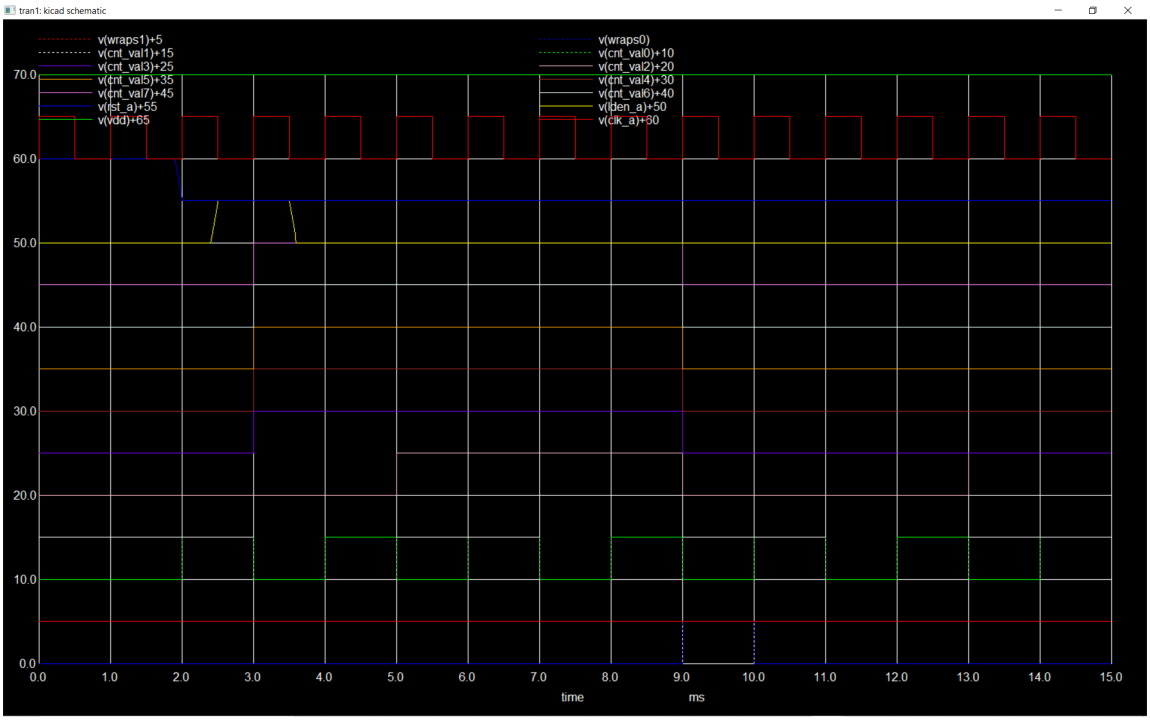


Figure 41: Stacked logic-analyser view of the Universal Counter transient. Voltage offsets separate the traces vertically. The predicted behaviour is reproduced exactly: the preset value appears at 3 ms, the count advances once per millisecond, and `v(wraps0)` produces a single overflow pulse at 9 ms while `v(wraps1)` never asserts.

Numerical inspection of the exported transient data confirms the plot. The most significant bit `cnt_val7` rises at 3 ms (the instant 250 is loaded) and falls at 9 ms (the `255` $\rightarrow$ `0` wrap); `cnt_val0` toggles once per count; `wraps[0]` produces exactly one pulse, at 9 ms; and `wraps[1]` remains low for the entire run, correctly indicating that no underflow occurred while counting upward.

6.8 Conclusion

The Universal Up/Down Counter supplies eSim with the general-purpose counting element the library previously lacked. Both counting modes, both directions, the load and enable controls, and the single-cycle semantics of both wrap outputs were verified exhaustively in the digital domain (24/24 checks) and then reproduced in the analog domain, where the preset-count-overflow sequence appears exactly as predicted.

7 Digital IP Datasheet & Verification Report: Digital One-Shot (Monostable) Timer

7.1 Introduction & Purpose

Of all the digital blocks in this report, the one-shot is the one an analog engineer recognises immediately: it is the digital equivalent of the **555 timer in monostable configuration**. eSim already ships `Monostable555` and `Astable555` as stock example circuits, which makes the comparison concrete — the analog 555 sets its pulse width through an external RC network, with the accuracy and drift of the components, whereas this core sets it by **counting clock cycles**, giving an exactly reproducible width with no passive components at all.

Functionally the block converts an **event** into a **bounded interval**: a rising edge on the trigger produces a single output pulse of programmable duration, after which the output returns low and stays there until the next trigger. That primitive underlies switch debouncing, watchdog strobes, fixed-duration actuator drives, delay-and-capture logic, and pulse-width conversion for sensors.

7.2 Architecture & Pin Specifications

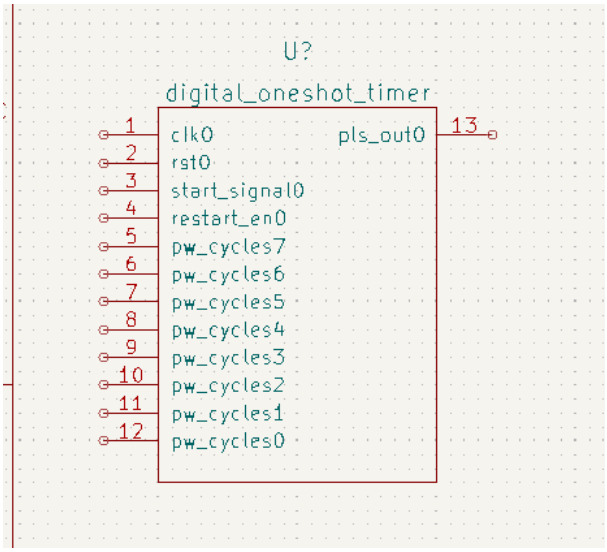


Figure 42: The NgVeri-generated KiCad symbol for `digital_oneshot_timer`: twelve input pins (`clk`, `rst`, `start_signal`, `restart_en` and the eight `pw_cycles` bits) and a single output pin `pls_out`.

Port	Width	Direction	Function
clk	1	in	Timebase; the pulse width is measured in cycles of this clock
rst	1	in	Synchronous reset; aborts any pulse in progress
start_signal	1	in	Trigger; a **rising edge** starts the pulse
restart_en	1	in	0 = non-retriggerable, 1 = retriggerable
pw_cycles	8	in	Pulse width in clock cycles (1–255; 0 disables)
pls_out	1	out	The monostable output pulse

Table 18: Port specification of the Digital One-Shot Timer.

7.3 Working Principle

7.3.1 Edge Detection

The core must respond to a trigger *edge*, not a level, otherwise a trigger held high would retrigger continuously. A one-bit history register `start_previous` is updated every clock, and the firing condition tests `start_signal && !start_previous` — true for exactly one cycle per rising edge. This is why the trigger source in the analog test only needs to be a slow, clean ramp: the edge detector reduces any sufficiently long high level to a single internal event.

7.3.2 Firing Condition and the Two Trigger Modes

The complete guard for starting a pulse is

$$\texttt{start\_signal} \wedge \overline{\texttt{start\_previous}} \wedge (\texttt{pw\_cycles} \neq 0) \wedge (\overline{\texttt{pls\_out}} \vee \texttt{restart\_en})$$

The final term selects the trigger mode. With `restart_en` low the guard fails while `pls_out` is already high, so mid-pulse triggers are **ignored** and the output width is always exactly `pw_cycles` — the behaviour required for a stable, predictable strobe. With `restart_en` high the guard succeeds regardless, so the countdown register is **reloaded** and the pulse is extended; a stream of triggers arriving faster than `pw_cycles` then holds the output continuously high, which is precisely the classic "activity detector" or watchdog application.

7.3.3 Countdown and Termination

When the guard passes, `cycles_remaining` is loaded with `pw_cycles` and `pls_out` is set. On each subsequent clock the counter decrements, and when it reaches 1 the output is cleared, producing a total high time of exactly `pw_cycles` clock periods. A programmed width of **zero** is explicitly rejected by the guard, so a floating or un-driven width bus can never generate a spurious pulse — a small but important safety property in a schematic where an unconnected ADC bridge channel is an easy mistake.

7.4 RTL Source Code

Listing 12: Digital One-Shot Timer RTL (`digital_oneshot_timer.v`)

```
1  `timescale 1ns / 1ps
2
3  module digital_oneshot_timer (
4      input  clk,
```

```
    input  rst,
    input  start_signal,
    input  restart_en,
    input  [7:0] pw_cycles,
    output pls_out
);

    reg pls_out_reg;


    reg [7:0] cycles_remaining;
    reg       start_previous;

    always @(posedge clk) begin
        if (rst) begin
            pls_out_reg        <= 1'b0;
            cycles_remaining <= 8'd0;
            start_previous <= 1'b0;
        end else begin
            start_previous <= start_signal;

            if (start_signal && !start_previous && (pw_cycles != 8'd0) && (!pls_out_reg || restart_en))
    ↪  begin
                pls_out_reg        <= 1'b1;
                cycles_remaining <= pw_cycles;
            end else if (pls_out_reg) begin
                if (cycles_remaining <= 8'd1) begin
                    pls_out_reg <= 1'b0;
                end
                cycles_remaining <= cycles_remaining - 8'd1;
            end
        end
    end

    assign pls_out = pls_out_reg;
endmodule
```

7.5 Self-Checking Testbench

Measuring a pulse *width* requires a different technique from checking a value: the testbench forks a `measure_high_cycles` task that timestamps the rise and fall of `pls_out` and converts the interval into clock cycles. The measuring task is started **before** the trigger is fired so that no edge can be missed. A global timeout guards against a hung wait.

Listing 13: Digital One-Shot Timer self-checking testbench (`tb_digital_oneshot_timer.v`)

```
`timescale 1ns / 1ps

module tb_digital_oneshot_timer;

    reg        clk;
    reg        rst;
    reg        start_signal;
    reg        restart_en;
    reg  [7:0] pw_cycles;
    wire       pls_out;

    integer pass_cnt = 0;
    integer fail_cnt = 0;
    integer high_cycles;
    integer sample_highs;
    integer k;

    digital_oneshot_timer uut (
        .clk(clk),
        .rst(rst),
```

```
        .start_signal(start_signal),
        .restart_en(restart_en),
        .pw_cycles(pw_cycles),
        .pls_out(pls_out)
    );

    always #5 clk = ~clk;

    // Safety watchdog so a hung wait can never stall the simulator forever
    initial begin
        #200000;
        $display("GLOBAL TIMEOUT REACHED - ABORTING SIMULATION");
        $finish;
    end

    task log_result;
        input [8*34:1] test_name;
        input integer  expected_val;
        input integer  actual_val;
        reg ok;
        begin
            ok = (expected_val === actual_val);
            $display("| %-34s | %8d | %8d | %-6s |",
                     test_name, expected_val, actual_val, ok ? "PASS" : "FAIL");
            if (ok) pass_cnt = pass_cnt + 1;
            else    fail_cnt = fail_cnt + 1;
        end
    endtask

    // Fire a single-clk trigger pulse aligned to the falling edge
    task fire_start;
        begin
            @(negedge clk); start_signal = 1;
            @(negedge clk); start_signal = 0;
        end
    endtask

    // Timestamp the rise and fall of pls_out, return width in clk cycles.
    // Must be started in parallel (fork) BEFORE the trigger is fired.
    task measure_high_cycles;
        output integer cycles;
        integer t_rise;
        begin
            wait (pls_out === 1'b1);
            t_rise = $time;
            wait (pls_out === 1'b0);
            cycles = ($time - t_rise) / 10;
        end
    endtask

    initial begin
        $dumpfile("digital_oneshot_timer_waveform.vcd");
        $dumpvars(0, tb_digital_oneshot_timer);

        clk = 0; rst = 1; start_signal = 0; restart_en = 0;
        pw_cycles = 8'd0;
        repeat (2) @(posedge clk);
        @(negedge clk); rst = 0;

        $display("\n================================================================================");
        $display(" DIGITAL ONE-SHOT (MONOSTABLE) TIMER VERIFICATION SUITE");
        $display("================================================================================");
        $display("| Test Name                          | Expected | Actual   | Status |");
        $display("|------------------------------------|----------|----------|--------|");

        log_result("TC1: Idle output low", 0, pls_out);

        // Basic width: 5 clk cycles
        pw_cycles = 8'd5;
        fork
            measure_high_cycles(high_cycles);
            fire_start;
        join
```

```
        log_result("TC2: Width-5 pulse duration", 5, high_cycles);
        log_result("TC3: Output returns low", 0, pls_out);

        // Non-retrigable: a second trigger mid-pulse must be ignored
        pw_cycles = 8'd6;
        restart_en = 0;
        fork
            measure_high_cycles(high_cycles);
            begin
                fire_start;
                repeat (2) @(negedge clk);
                fire_start;
            end
        join
        log_result("TC4: Non-retrig total width", 6, high_cycles);

        // retrigable: a second trigger mid-pulse restarts the timing.
        // First trigger at edge E1 (width 4), retrig lands at edge E5 and
        // reloads the counter, so the pulse falls at E9: total 8 cycles high.
        pw_cycles = 8'd4;
        restart_en = 1;
        fork
            measure_high_cycles(high_cycles);
            begin
                fire_start;
                repeat (2) @(negedge clk);
                fire_start;
            end
        join
        log_result("TC5: retrig extends width", 8, high_cycles);

        // Zero programmed width must produce no pulse at all
        restart_en = 0;
        pw_cycles = 8'd0;
        fire_start;
        sample_highs = 0;
        for (k = 0; k < 5; k = k + 1) begin
            @(posedge clk); #1;
            if (pls_out === 1'b1) sample_highs = sample_highs + 1;
        end
        log_result("TC6: Zero width gives no pulse", 0, sample_highs);

        // rst must kill an active pulse immediately
        pw_cycles = 8'd50;
        fire_start;
        @(posedge clk); #1;
        log_result("TC7: Long pulse starts", 1, pls_out);
        @(negedge clk); rst = 1;
        @(posedge clk); #1;
        log_result("TC8: rst kills active pulse", 0, pls_out);
        @(negedge clk); rst = 0;

        // Minimum width of a single clk cycle
        pw_cycles = 8'd1;
        fork
            measure_high_cycles(high_cycles);
            fire_start;
        join
        log_result("TC9: Minimum width-1 pulse", 1, high_cycles);

        $display("================================================================================");
        $display(" TOTAL CHECKS PASSED: %0d/%0d", pass_cnt, pass_cnt + fail_cnt);
        $display(" TOTAL CHECKS FAILED: %0d/%0d", fail_cnt, pass_cnt + fail_cnt);
        $display("================================================================================\n");
        $finish;
    end

endmodule
```

7.6 Verification Phase 1: Pure Digital (Vivado XSim)

Check	Feature exercised	Expected result
TC1	Idle state	output low before any trigger
TC2	Programmed width	measured width = 5 cycles
TC3	Self-termination	output returns low unaided
TC4	**Non-retriggerable** mode	mid-pulse trigger ignored; total width still 6
TC5	**Retriggerable** mode	mid-pulse trigger reloads; total width extends to 8
TC6	Zero-width safety	`pw_cycles` = 0 produces no pulse at all
TC7–TC8	Reset abort	reset clears an active pulse immediately
TC9	Minimum width	`pw_cycles` = 1 gives a clean 1-cycle pulse

Table 19: The nine directed checks applied to the Digital One-Shot Timer.

TC4 and TC5 are the discriminating pair: the same stimulus sequence is applied twice, differing only in `restart_en`, and the measured widths differ exactly as the mode definition requires (6 cycles versus 8).

```
source tb_digital_oneshot_timer.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window. If you want to open a wave window go to 'File->New Waveform
#   }
# }
# run 1000ns

================================================================================
 DIGITAL ONE-SHOT (MONOSTABLE) TIMER VERIFICATION SUITE
================================================================================
| Test Name                          | Expected | Actual   | Status |
|------------------------------------|----------|----------|--------|
|              TC1: Idle output low |        0 |        0 |   PASS |
|        TC2: Width-5 pulse duration |        5 |        5 |   PASS |
|            TC3: Output returns low |        0 |        0 |   PASS |
|        TC4: Non-retrig total width |        6 |        6 |   PASS |
|          TC5: retrig extends width |        8 |        8 |   PASS |
|     TC6: Zero width gives no pulse |        0 |        0 |   PASS |
|             TC7: Long pulse starts |        1 |        1 |   PASS |
|        TC8: rst kills active pulse |        0 |        0 |   PASS |
|         TC9: Minimum width-1 pulse |        1 |        1 |   PASS |
================================================================================
 TOTAL CHECKS PASSED: 9/9
 TOTAL CHECKS FAILED: 0/9
================================================================================

$finish called at time : 365 ns : File "C:/Users/Apple/Documents/esim_design/internship/Digital_IP_Design/more_design/2_Oneshot_Monostable_Timer/tb_digital_oneshot_timer.v
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_digital_oneshot_timer_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:03 ; elapsed = 00:00:08 . Memory (MB): peak = 2256.883 ; gain = 0.000
```

Figure 43: Vivado Tcl console output for the Digital One-Shot Timer: **9/9** checks pass, including both trigger modes, the zero-width safety interlock and the reset abort.

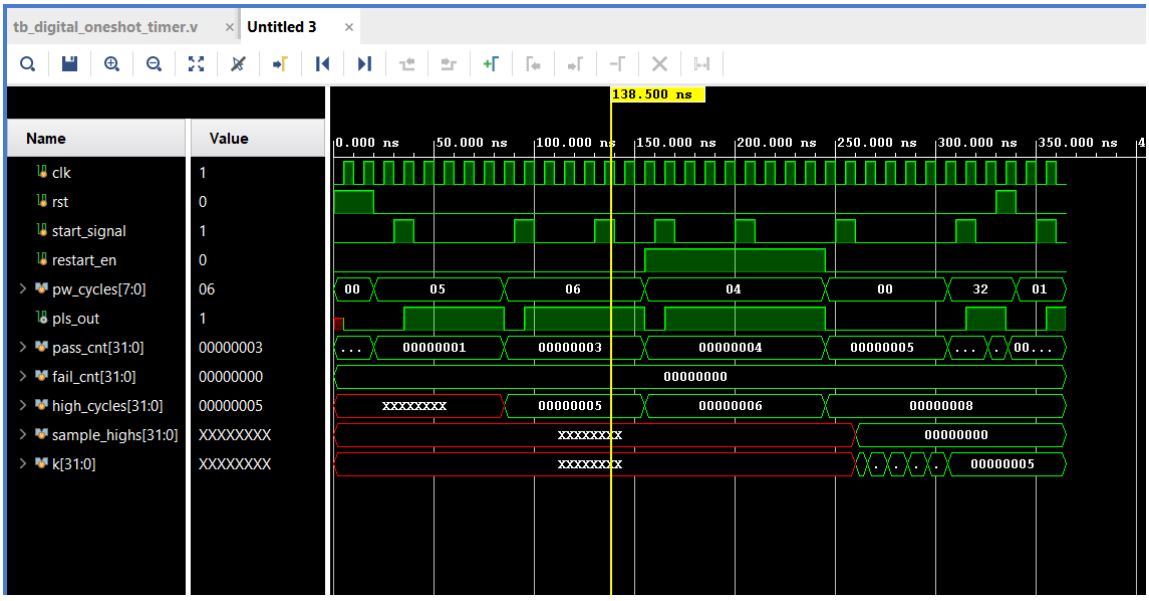


Figure 44: Vivado XSim waveform for the One-Shot Timer. Each trigger edge launches a single bounded pulse; the retriggerable case visibly extends the high time when a second trigger arrives before the countdown completes.

7.7 Verification Phase 2: Mixed-Signal (eSim / Ngspice)

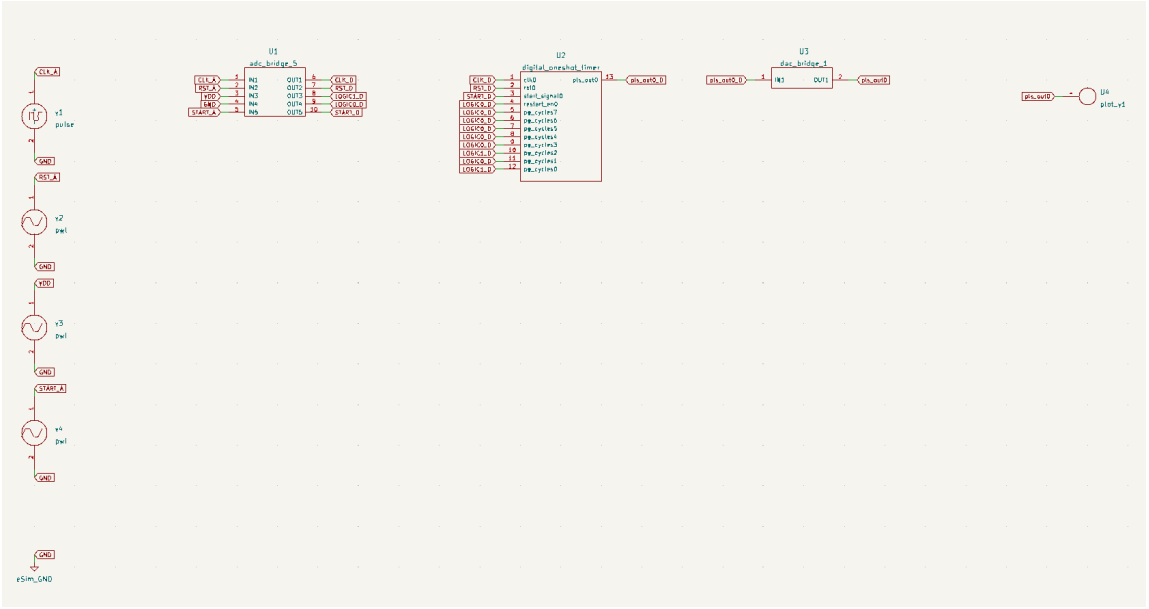


Figure 45: The generated eSim test schematic for the One-Shot Timer. The `pw_cycles` bus is hard-wired to the constant 5 through the `VDD`/`GND` rails, so only four analog nets need real sources.

7.7.1 Stimulus Definition

Net	Source	Value	Purpose
`CLK_A`	`pulse`	`0 5 0 1u 1u 0.5m 1m`	1 kHz timebase (1 ms per counted cycle)
`RST_A`	`pwl`	`0 5 1.9m 5 2m 0`	release the core at 2 ms
`VDD`	`pwl`	`0 5 100 5`	logic-1 rail for the `pw_cycles` bits
`START_A`	`pwl`	`0 0 4.9m 0 5m 5 6m 5 6.1m 0`	single trigger edge at 5 ms

Table 20: Analog stimulus for the One-Shot transient. `restart_en` is tied low (non-retriggerable). Transient settings: step 10 μs, stop 30 ms.

With a 1 kHz clock each counted cycle lasts 1 ms, so a programmed width of five cycles must produce a **5 ms** output pulse. Choosing the timebase this way makes the result directly readable off the time axis without any scaling — the pulse width in milliseconds *is* the programmed cycle count.

7.7.2 Results

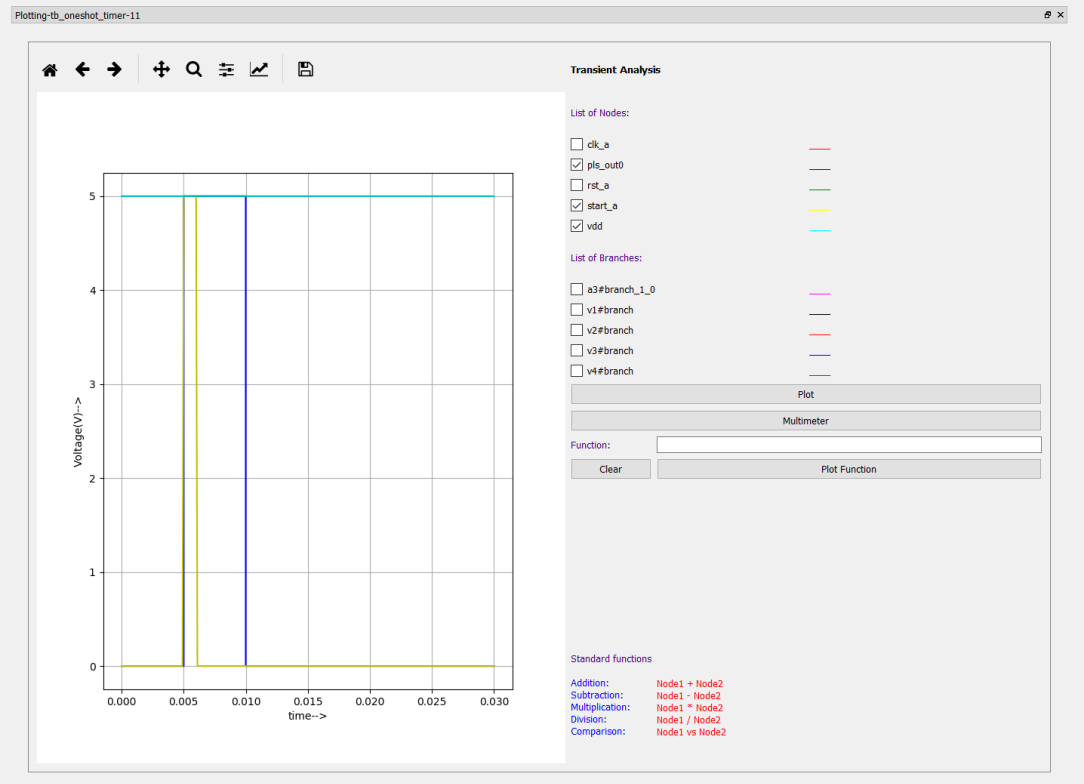


Figure 46: eSim plotting window for the One-Shot transient, listing the captured analog nets and the single digital output `pls_out`.

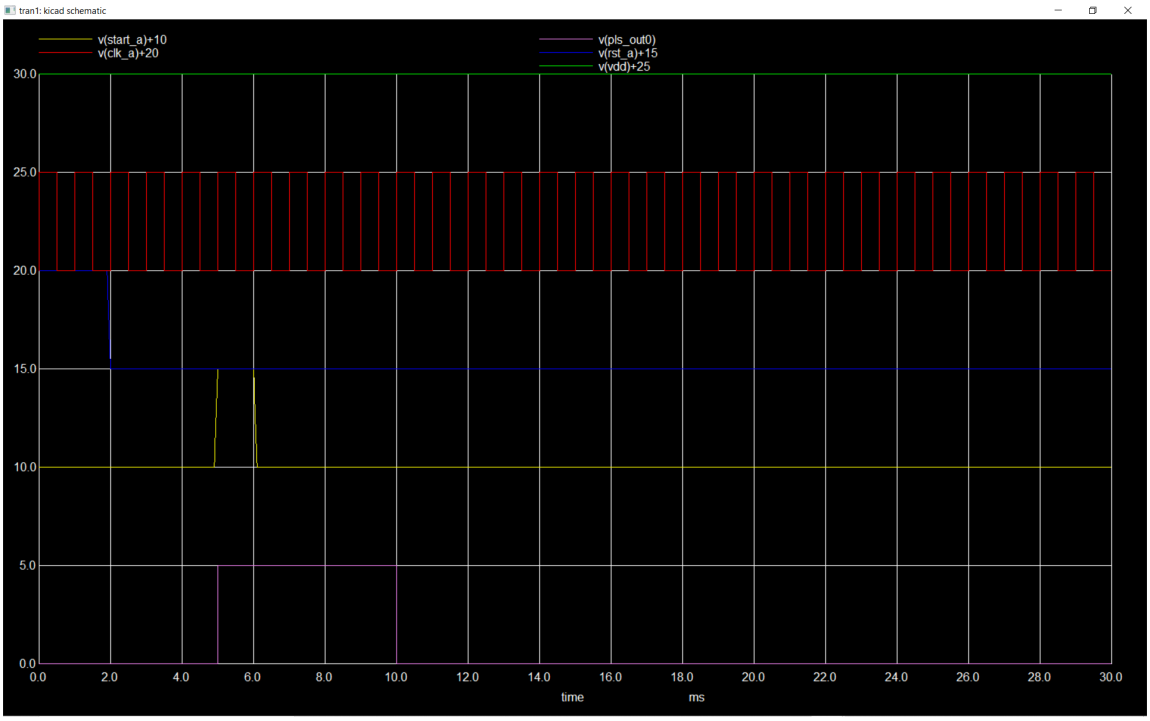


Figure 47: Stacked Ngspice view of the One-Shot transient. The trigger rises at 5 ms and `v(pls_out)` goes high for exactly 5 ms, falling at 10 ms without any further stimulus — the defining monostable behaviour, with the width set purely by the programmed cycle count.

Numerical inspection of the transient data confirms the measurement: `pls_out` rises at 5.000 ms and falls at 10.000 ms, a high time of exactly five 1 ms clock cycles, and it remains low for the remaining 20 ms of the run despite the trigger source having long since returned to 0 V.

7.8 Conclusion

The Digital One-Shot Timer provides eSim with a clock-accurate monostable whose width is set by an integer rather than by an RC product. Both trigger modes, the zero-width interlock and the reset abort passed all nine digital checks, and the mixed-signal run reproduced the programmed width exactly. Because eSim ships a 555 monostable as a stock example, the block also invites a direct pedagogical comparison between the analog and digital approaches to the same timing problem.

8 Digital IP Datasheet & Verification Report: LFSR Pseudo-Random Generator

8.1 Introduction & Purpose

Testing an analog signal chain requires a stimulus that is **broadband and uncorrelated** — a filter's response, a comparator's hysteresis or an ADC's differential non-linearity cannot be characterised with a single sine wave. Ngspice can produce noise in a `.noise` analysis, but that is a small-signal frequency-domain calculation; it does not give a *time-domain* pseudo-random waveform that a transient run can drive through a real circuit.

The **LFSR Pseudo-Random Generator** supplies exactly that. A linear-feedback shift register produces a deterministic bit sequence with white-noise-like statistics; routed through a `dac_bridge` the parallel output becomes a stepped analog waveform suitable as a noise source, and the serial output is a PRBS bit stream for jitter, coding and synchronisation experiments. Crucially, because the sequence is generated by a known recurrence it is **exactly repeatable** — the same seed always produces the same waveform, which is what makes a pseudo-random source useful for regression testing rather than merely random.

8.2 Architecture & Pin Specifications

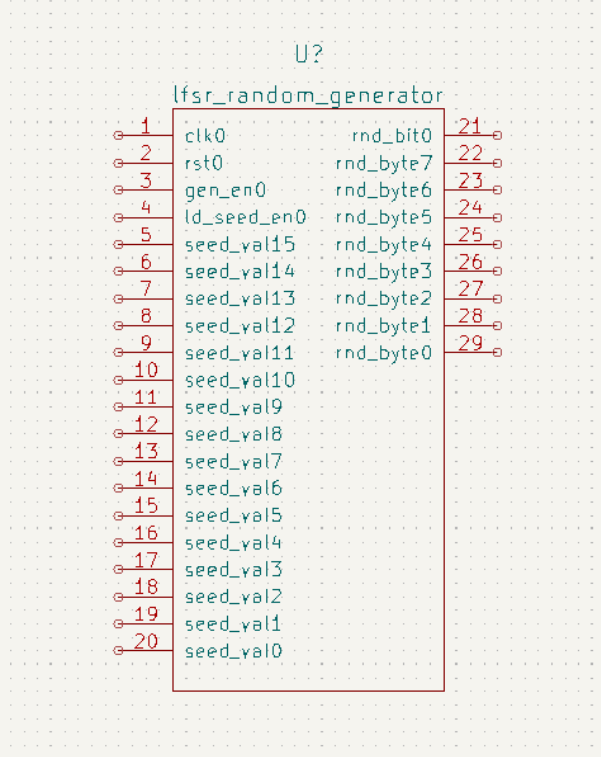


Figure 48: The NgVeri-generated KiCad symbol for `lfsr_random_generator`. The 16-bit `seed_val` bus dominates the pin count; in normal use those pins are tied to a constant seed (or to ground, which selects the built-in default).

Port	Width	Direction	Function
clk	1	in	Advance clock; one fresh output byte per rising edge
rst	1	in	Synchronous reset; restores the default seed
gen_en	1	in	Generate enable; when low the output is frozen
ld_seed_en	1	in	Seed load enable
seed_val	16	in	Seed value; 0x0000 is remapped to the safe default
rnd_bit	1	out	Serial pseudo-random bit (PRBS stream)
rnd_byte	8	out	Parallel pseudo-random byte

Table 21: Port specification of the LFSR Pseudo-Random Generator.

8.3 Working Principle

8.3.1 Galois Configuration and Tap Selection

The core implements a 16-bit **Galois** (right-shifting, internal-XOR) LFSR with the tap mask 0xB400, corresponding to the primitive polynomial

$$x^{16} + x^{14} + x^{13} + x^{11} + 1.$$

Because the polynomial is primitive, the register traverses all $2^{16} - 1 = 65\,535$ non-zero states before repeating — a **maximal-length** sequence. Each step shifts the state right by one bit and, if the bit shifted out was 1, XORs the tap mask into the remaining state. The Galois form is preferred over the Fibonacci form here because it needs only a single XOR stage regardless of the number of taps, which keeps the unrolled combinational logic shallow.

8.3.2 Eight Steps per Clock: a Fresh Byte Every Cycle

A naïve LFSR emits one bit per clock, so assembling a byte would take eight cycles. To give eSim users a new **byte** on every clock edge, the core unrolls **eight shift steps** inside a combinational `always @(*)` block, accumulating the eight bits shifted out into `byte_next` while carrying the state forward through `lfsr_next`. The sequential block then commits `lfsr_next` and `byte_next` in one step. This matters in a mixed-signal context: a stepped noise waveform is far more useful when its update rate equals the clock rate rather than one-eighth of it.

8.3.3 The All-Zero Lock-Up State and Its Prevention

An LFSR's one pathological state is **all zeros**: the XOR feedback of a zero state is zero, so the register locks up permanently and the output dies. Since the seed arrives from outside the block — and in a KiCad schematic the natural thing to do with a 16-bit bus a user does not care about is to **ground every pin** — an unguarded design would fail in exactly the most likely usage. The core therefore intercepts a seed of 0x0000 and substitutes the non-zero constant 0xACE1, which is also the reset value. The practical consequence is important for the schematic: grounding the entire `seed_val` bus is a **legitimate, fully supported configuration** that yields the default maximal-length sequence, and it removes sixteen sources from the test sheet.

8.4 RTL Source Code

Listing 14: LFSR Pseudo-Random Generator RTL (`lfsr_random_generator.v`)

```
`timescale 1ns / 1ps

module lfsr_random_generator (
    input  clk,
    input  rst,
    input  gen_en,
    input  ld_seed_en,
    input  [15:0] seed_val,
    output rnd_bit,
    output [7:0]   rnd_byte
);

    reg rnd_bit_reg;
    reg [7:0]   rnd_byte_reg;


    reg [15:0] lfsr_state;
    reg [15:0] lfsr_next;
    reg [7:0]  byte_next;
    integer    bit_index;

    always @(*) begin
        lfsr_next = lfsr_state;
        byte_next = 8'd0;
        for (bit_index = 0; bit_index < 8; bit_index = bit_index + 1) begin
            byte_next = {byte_next[6:0], lfsr_next[0]};
            if (lfsr_next[0]) begin
                lfsr_next = {1'b0, lfsr_next[15:1]} ^ 16'hB400;
            end else begin
                lfsr_next = {1'b0, lfsr_next[15:1]};
            end
        end
    end

    always @(posedge clk) begin
        if (rst) begin
            lfsr_state      <= 16'hACE1;
            rnd_bit_reg  <= 1'b0;
            rnd_byte_reg <= 8'd0;
        end else if (ld_seed_en) begin
            if (seed_val == 16'd0) begin
                lfsr_state <= 16'hACE1;
            end else begin
                lfsr_state <= seed_val;
            end
        end else if (gen_en) begin
            lfsr_state      <= lfsr_next;
            rnd_byte_reg <= byte_next;
            rnd_bit_reg  <= byte_next[0];
        end
    end

    assign rnd_bit = rnd_bit_reg;
    assign rnd_byte = rnd_byte_reg;
endmodule
```

8.5 Self-Checking Testbench

Verifying a pseudo-random generator requires more than a spot check: the output must match the mathematics exactly, *and* it must look statistically random. The testbench does both. It contains an independent **golden reference model** of the recurrence written as a Verilog `function`, and compares the DUT byte-for-byte against it; it then measures the statistical properties of the stream over thousands of samples using a range-

based check (`log_range`) rather than an equality check.

Listing 15: LFSR Pseudo-Random Generator self-checking testbench (`tb_lfsr_random_generator.v`)

```
`timescale 1ns / 1ps

module tb_lfsr_random_generator;

    reg         clk;
    reg         rst;
    reg         gen_en;
    reg         ld_seed_en;
    reg  [15:0] seed_val;
    wire        rnd_bit;
    wire [7:0]  rnd_byte;

    integer pass_cnt = 0;
    integer fail_cnt = 0;
    integer k;
    integer ones_total;
    integer change_total;
    integer mismatch;

    reg [15:0] golden_state;
    reg [7:0]  golden_byte;
    reg [23:0] golden_pack;
    reg [7:0]  capture_run1 [0:15];
    reg [7:0]  capture_run2 [0:15];
    reg [7:0]  previous_byte;

    lfsr_random_generator uut (
        .clk(clk),
        .rst(rst),
        .gen_en(gen_en),
        .ld_seed_en(ld_seed_en),
        .seed_val(seed_val),
        .rnd_bit(rnd_bit),
        .rnd_byte(rnd_byte)
    );

    always #5 clk = ~clk;

    // Golden reference: 16-bit Galois LFSR, taps 0xB400, advanced 8 bits
    function [23:0] golden_advance8;
        input [15:0] state_in;
        reg [15:0] s;
        reg [7:0]  b;
        integer    n;
        begin
            s = state_in;
            b = 8'd0;
            for (n = 0; n < 8; n = n + 1) begin
                b = {b[6:0], s[0]};
                if (s[0]) s = {1'b0, s[15:1]} ^ 16'hB400;
                else      s = {1'b0, s[15:1]};
            end
            golden_advance8 = {s, b};
        end
    endfunction

    task log_result;
        input [8*34:1] test_name;
        input integer  expected_val;
        input integer  actual_val;
        reg ok;
        begin
            ok = (expected_val === actual_val);
            $display("| %-34s | %8d | %8d | %-6s |",
                     test_name, expected_val, actual_val, ok ? "PASS" : "FAIL");
            if (ok) pass_cnt = pass_cnt + 1;
            else    fail_cnt = fail_cnt + 1;
        end
```

```
    endtask

    task log_range;
        input [8*34:1] test_name;
        input integer  low_bound;
        input integer  high_bound;
        input integer  actual_val;
        reg ok;
        begin
            ok = (actual_val >= low_bound) && (actual_val <= high_bound);
            $display("| %-34s | %4d-%-4d| %8d | %-6s |",
                     test_name, low_bound, high_bound, actual_val, ok ? "PASS" : "FAIL");
            if (ok) pass_cnt = pass_cnt + 1;
            else    fail_cnt = fail_cnt + 1;
        end
    endtask

    initial begin
        $dumpfile("lfsr_random_generator_waveform.vcd");
        $dumpvars(0, tb_lfsr_random_generator);

        clk = 0; rst = 1; gen_en = 0;
        ld_seed_en = 0; seed_val = 16'd0;
        repeat (2) @(posedge clk);
        @(negedge clk); rst = 0;

        $display("\n================================================================================");
        $display(" LFSR PSEUDO-RANDOM GENERATOR VERIFICATION SUITE");
        $display("================================================================================");
        $display("| Test Name                          | Expected | Actual   | Status |");
        $display("|------------------------------------|----------|----------|--------|");

        // TC1: 20 bytes must match the golden model from the default seed
        golden_state = 16'hACE1;
        mismatch = 0;
        @(negedge clk); gen_en = 1;
        for (k = 0; k < 20; k = k + 1) begin
            golden_pack  = golden_advance8(golden_state);
            golden_state = golden_pack[23:8];
            golden_byte  = golden_pack[7:0];
            @(posedge clk); #1;
            if (rnd_byte !== golden_byte) mismatch = mismatch + 1;
        end
        log_result("TC1: 20 bytes match golden model", 0, mismatch);

        // TC2: freeze when gen_en is low
        @(negedge clk); gen_en = 0;
        previous_byte = rnd_byte;
        mismatch = 0;
        for (k = 0; k < 5; k = k + 1) begin
            @(posedge clk); #1;
            if (rnd_byte !== previous_byte) mismatch = mismatch + 1;
        end
        log_result("TC2: Output frozen when disabled", 0, mismatch);

        // TC3: same seed twice produces the identical sequence
        @(negedge clk); ld_seed_en = 1; seed_val = 16'hBEEF;
        @(posedge clk); #1;
        @(negedge clk); ld_seed_en = 0; gen_en = 1;
        for (k = 0; k < 16; k = k + 1) begin
            @(posedge clk); #1;
            capture_run1[k] = rnd_byte;
        end
        @(negedge clk); gen_en = 0;
        @(negedge clk); ld_seed_en = 1; seed_val = 16'hBEEF;
        @(posedge clk); #1;
        @(negedge clk); ld_seed_en = 0; gen_en = 1;
        for (k = 0; k < 16; k = k + 1) begin
            @(posedge clk); #1;
            capture_run2[k] = rnd_byte;
        end
        mismatch = 0;
        for (k = 0; k < 16; k = k + 1)
```

```
142             if (capture_run1[k] !== capture_run2[k]) mismatch = mismatch + 1;
143         log_result("TC3: Seeded sequence repeatable", 0, mismatch);
144
145         // TC4: all-zero seed is remapped to the nonzero default
146         @(negedge clk); gen_en = 0;
147         @(negedge clk); ld_seed_en = 1; seed_val = 16'h0000;
148         @(posedge clk); #1;
149         @(negedge clk); ld_seed_en = 0; gen_en = 1;
150         golden_state = 16'hACE1;
151         mismatch = 0;
152         for (k = 0; k < 8; k = k + 1) begin
153             golden_pack  = golden_advance8(golden_state);
154             golden_state = golden_pack[23:8];
155             golden_byte  = golden_pack[7:0];
156             @(posedge clk); #1;
157             if (rnd_byte !== golden_byte) mismatch = mismatch + 1;
158         end
159         log_result("TC4: Zero seed uses safe default", 0, mismatch);
160
161         // TC5: serial bit balance over 2048 samples (expect close to 50%)
162         ones_total = 0;
163         for (k = 0; k < 2048; k = k + 1) begin
164             @(posedge clk); #1;
165             if (rnd_bit === 1'b1) ones_total = ones_total + 1;
166         end
167         log_range("TC5: Bit balance (of 2048)", 819, 1229, ones_total);
168
169         // TC6: byte stream keeps changing (no lock-up state)
170         change_total = 0;
171         previous_byte = rnd_byte;
172         for (k = 0; k < 300; k = k + 1) begin
173             @(posedge clk); #1;
174             if (rnd_byte !== previous_byte) change_total = change_total + 1;
175             previous_byte = rnd_byte;
176         end
177         log_range("TC6: Byte changes (of 300)", 280, 300, change_total);
178
179         $display("================================================================================");
180         $display(" TOTAL CHECKS PASSED: %0d/%0d", pass_cnt, pass_cnt + fail_cnt);
181         $display(" TOTAL CHECKS FAILED: %0d/%0d", fail_cnt, pass_cnt + fail_cnt);
182         $display("================================================================================\n");
183         $finish;
184     end
185
186 endmodule
```

8.6 Verification Phase 1: Pure Digital (Vivado XSim)

Check	Property verified	Criterion	Measured
TC1	Bit-exact recurrence	20 consecutive bytes match the golden model	0 mismatches
TC2	Enable gating	output frozen while `gen_en` is low	0 changes
TC3	Determinism	the same seed replays the identical 16-byte sequence	0 mismatches
TC4	Zero-seed protection	seed `0x0000` yields the default sequence	0 mismatches
TC5	Statistical balance	ones in 2048 serial bits within 819–1229	**1002** (48.9 %)
TC6	No lock-up	byte changes in 300 clocks within 280–300	**300** (100 %)

Table 22: The checks applied to the LFSR, combining exact-model comparison with statistical measurement.

TC5 and TC6 are the statistical arguments. A maximal-length LFSR should produce ones almost exactly half the time; the measured 1002 ones out of 2048 samples is **48.9 %**, comfortably inside the accepted band. TC6 confirms that the output byte changes on

every one of 300 consecutive clocks, which rules out both the all-zero lock-up state and any short cycle.

```
source tb_lfsr_random_generator.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window. If you want
#   }
# }
# run 1000ns

================================================================================
 LFSR PSEUDO-RANDOM GENERATOR VERIFICATION SUITE
================================================================================
| Test Name                          | Expected | Actual   | Status |
|------------------------------------|----------|----------|--------|
|   TC1: 20 bytes match golden model |        0 |        0 |   PASS |
|   TC2: Output frozen when disabled |        0 |        0 |   PASS |
|    TC3: Seeded sequence repeatable |        0 |        0 |   PASS |
|   TC4: Zero seed uses safe default |        0 |        0 |   PASS |
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_lfsr_random_generator_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:02 ; elapsed = 00:00:08 . Memory (MB): peak = 2256.883 ; gain = 0.000
run all
|         TC5: Bit balance (of 2048) |  819-1229|     1002 |   PASS |
|         TC6: Byte changes (of 300) |  280-300 |      300 |   PASS |
================================================================================
 TOTAL CHECKS PASSED: 6/6
 TOTAL CHECKS FAILED: 0/6
================================================================================

$finish called at time : 24206 ns : File "C:/Users/Apple/Documents/esim_design/internship/Digital_IP_Design/more_design/3_
```

Figure 49: Vivado Tcl console output for the LFSR: **6/6** checks pass. The golden-model comparison reports zero mismatches, the bit balance measures 1002 ones in 2048 samples, and all 300 sampled byte transitions are distinct.

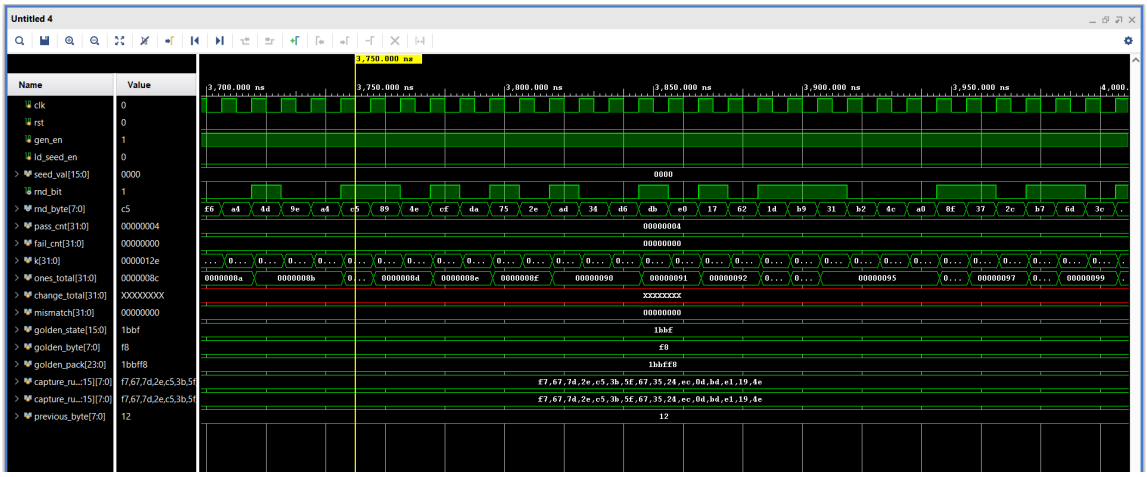


Figure 50: Vivado XSim waveform of the LFSR (first capture): the parallel `rnd_byte` output takes an apparently unrelated value on every clock edge, with no visible short-period pattern.

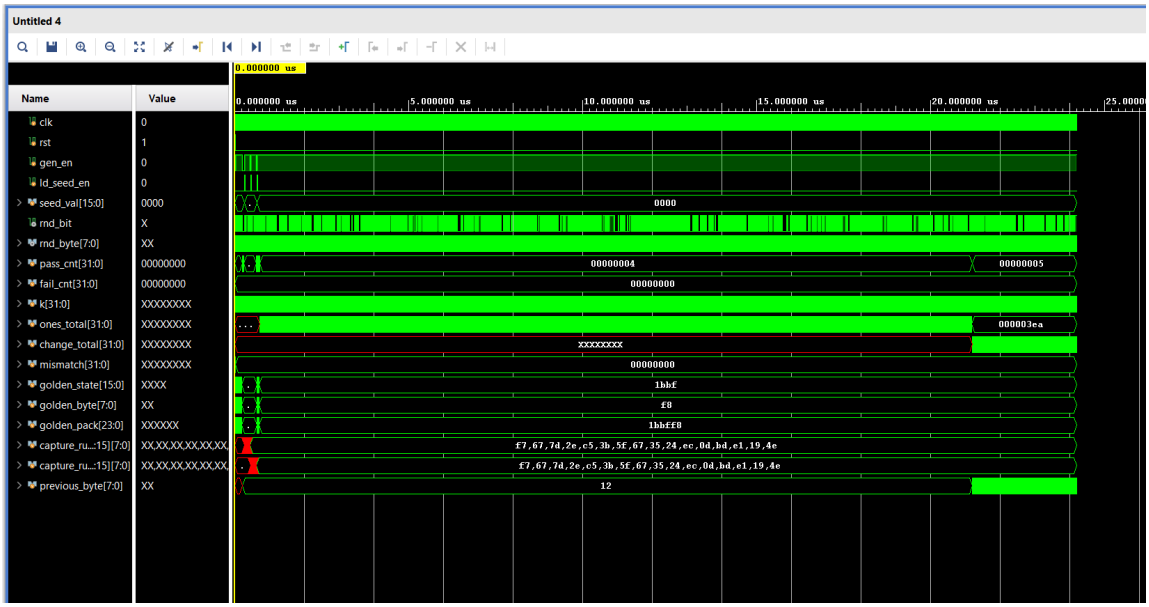


Figure 51: Vivado XSim waveform of the LFSR (second capture, different region of the sequence), together with the seed-load and enable-gating intervals used by TC2–TC4.

8.7 Verification Phase 2: Mixed-Signal (eSim / Ngspice)

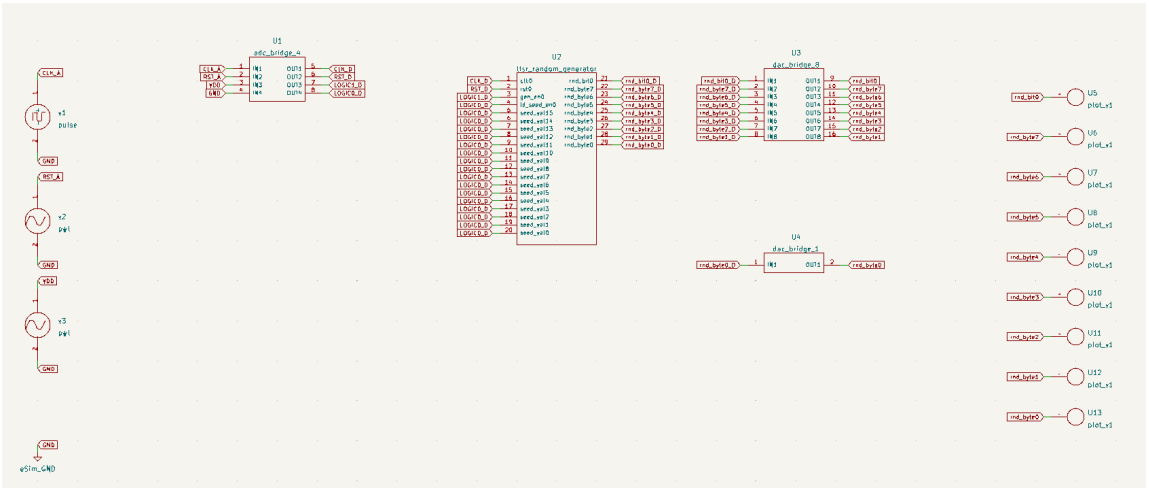


Figure 52: The generated eSim test schematic for the LFSR. The entire 16-bit `seed_val` bus and `ld_seed_en` are tied to `GND` — a supported configuration that selects the built-in default seed — so only three analog sources are required despite the core having twenty input pins.

8.7.1 Stimulus Definition

Net	Source	Value	Purpose
`CLK_A`	`pulse`	`0 5 0 1u 1u 0.5m 1m`	1 kHz advance clock
`RST_A`	`pwl`	`0 5 1.9m 5 2m 0`	load the default seed, release at 2 ms
`VDD`	`pwl`	`0 5 100 5`	`gen_en` held high (free-running)
`GND`	—	0 V	`ld_seed_en` and all 16 `seed_val` bits

Table 23: Analog stimulus for the LFSR transient. Transient settings: step 10 μs, stop 50 ms.

8.7.2 Results

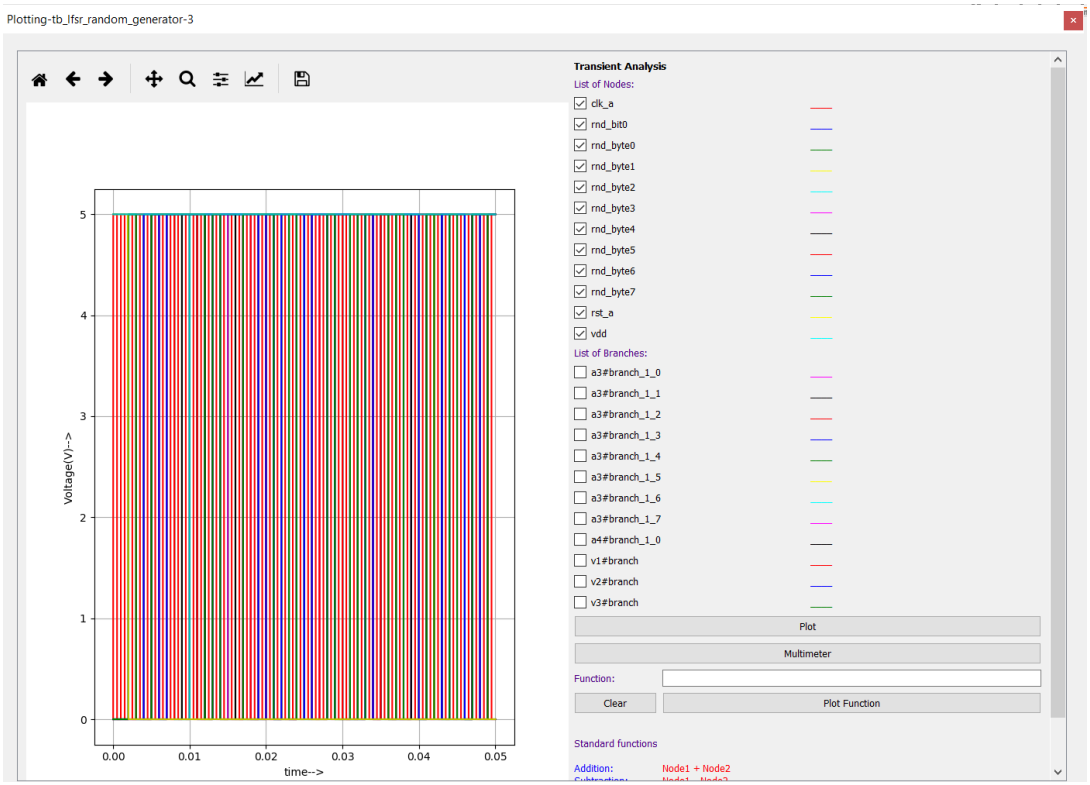


Figure 53: eSim plotting window after the LFSR transient, listing the serial `rnd_bit` output and all eight `rnd_byte` bits.

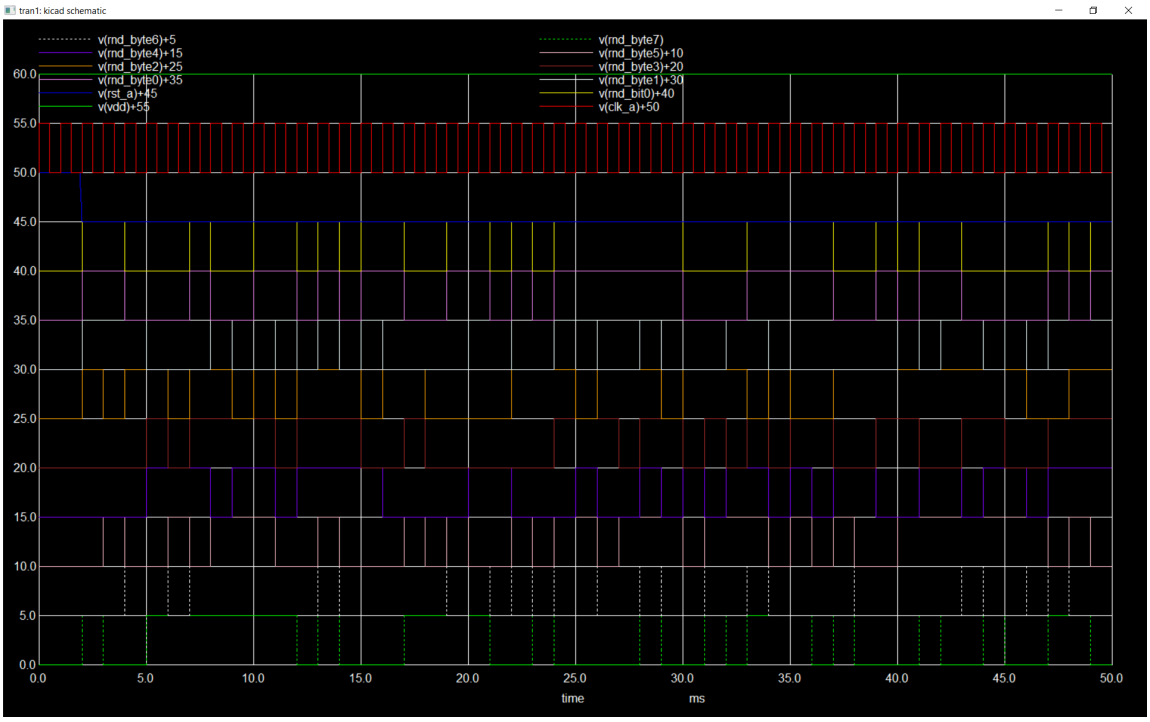


Figure 54: Stacked Ngspice view of the LFSR transient. Every output bit switches irregularly throughout the 50 ms window with no discernible repeating pattern and no bit stuck at a constant level — the time-domain signature of a maximal-length sequence surviving translation into the analog domain.

The exported transient data was analysed numerically to confirm the visual impression. Over the 50 ms run each observed output bit registers between **43 and 52 transitions** — close to the theoretical expectation of one transition every other clock for an uncorrelated stream — and each bit spends between **42 % and 54 %** of the time high. No bit is stuck, and no bit merely follows the clock.

8.8 Conclusion

The LFSR Pseudo-Random Generator gives eSim a repeatable broadband stimulus source in both serial and parallel form. Its output was proven bit-exact against an independent golden model of the recurrence, statistically balanced over thousands of samples, and immune to the all-zero lock-up state that the most natural schematic wiring would otherwise trigger. The mixed-signal run confirms that the byte-per-clock output survives translation through the Ngspice bridges with its statistical character intact.

9 Digital IP Datasheet & Verification Report: Signal Statistics Unit (Min / Max / True-RMS)

9.1 Introduction & Purpose

Every block presented so far manipulates signals. This one **measures** them. An eSim user simulating a power stage, an amplifier or a sensor front-end frequently needs a *number* rather than a picture: the true-RMS value of a node, or the extreme values a signal reached during a run. Obtaining those from a plot means exporting the raw data and post-processing it outside the tool.

The **Signal Statistics Unit** performs the measurement in hardware, in-circuit and in real time. Over each window of 256 samples it reports the **minimum**, the **maximum** and the **true root-mean-square** of a signed input stream, and raises a pulse when a fresh result is available. It is, in effect, the digital core of a multimeter, and it is the one measurement block that performs non-trivial arithmetic: computing an RMS value requires squaring, accumulating, dividing and then taking a **square root** — the last of which has no native operator in synthesisable Verilog.

An important continuity point: the square-root engine is the **same unrolled shift-and-subtract algorithm** developed for the Advanced Math ALU earlier in this report. That hardware has therefore now been validated twice, in two independent designs, which is a meaningful reuse result rather than a duplicated effort.

9.2 Architecture & Pin Specifications

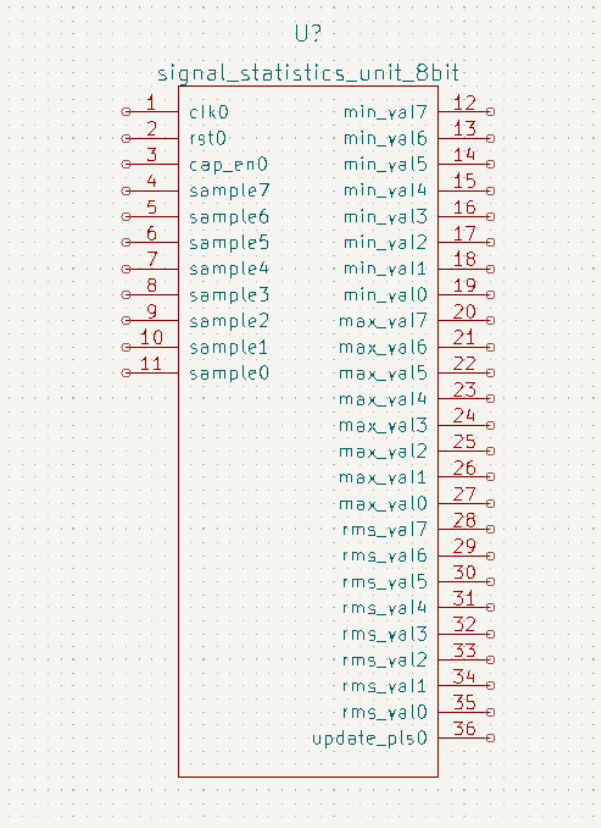


Figure 55: The NgVeri-generated KiCad symbol for `signal_statistics_unit_8bit`: eleven input pins and twenty-five output pins (three 8-bit result buses plus the `update_pls` strobe), making it the widest of the blocks documented here.

Port	Width	Direction	Function
`clk`	1	in	Sample clock; one sample is consumed per rising edge
`rst`	1	in	Synchronous reset; clears the window and all results
`cap_en`	1	in	Capture enable; clocks with `cap_en` low are ignored
`sample`	8	in	**Signed** (two's complement) input sample
`min_val`	8	out	Most negative sample of the completed window
`max_val`	8	out	Most positive sample of the completed window
`rms_val`	8	out	True-RMS magnitude of the completed window
`update_pls`	1	out	One-cycle pulse marking a freshly completed window

Table 24: Port specification of the Signal Statistics Unit.

9.3 Working Principle

9.3.1 Signed Comparison Without a Signed Comparator

The input is two's complement, so an unsigned comparison would mis-order it: `0xFF` (-1) would appear larger than `0x01` ($+1$). Rather than rely on Verilog's `signed` keyword

— deliberately avoided throughout this project because NgVeri's tool-chain handles it inconsistently — the core applies a **bias transformation**:

$$\texttt{sample} \oplus \texttt{0x80}$$

XOR-ing the sign bit converts the signed range $[-128, +127]$ into the unsigned range $[0, 255]$ while **preserving order**, so an ordinary unsigned `<` comparison becomes a correct signed comparison. This is the same offset-binary trick used in ADC encoding, and it costs one XOR gate.

The trackers are initialised to the *opposite* extremes — `min_val` to `0x7F` (+127) and `max_val` to `0x80` (−128) — so that the very first sample of a window always replaces both.

9.3.2 Magnitude, Squaring and Accumulation

RMS depends only on magnitude, so the core first computes |`sample`| by conditionally negating (two's complement: invert and add one) when the sign bit is set. The magnitude is squared and added to a **23-bit accumulator**. The width is chosen from the worst case:

$$128^2 \times 256 = 4\,194\,304 = 2^{22},$$

which fits in 23 bits with a bit to spare, so the accumulator **cannot overflow** even for a full-scale square wave.

9.3.3 Division by the Window Length as a Free Shift

The window length is fixed at **256 samples**, a power of two, so dividing the accumulated sum of squares by the sample count requires no divider at all — it is a right-shift by eight, realised simply by slicing bits `[22:8]` out of the accumulator. This is the reason the window length is 256 rather than a decimal round number: it turns the division into wiring.

9.3.4 Integer Square Root by Shift-and-Subtract

The mean square is converted to an RMS value by the unrolled restoring square-root algorithm inherited from the Advanced Math ALU. Eight iterations process the 16-bit target two bits at a time: the remainder is shifted left by two and the next bit-pair appended, a trial divisor is formed, and if it fits it is subtracted and a 1 is appended to the root. The loop is fully unrolled inside a combinational `always @(*)` block, so the result appears with **zero latency** after the mean square is latched — no clock cycles, no handshake, and no sequential control logic for the eSim user to drive.

9.3.5 Windowing and the Update Strobe

A sample counter advances only when `cap_en` is high. On the 256^{th} sample the three results are latched, `update_pls` is asserted for one cycle, and the trackers and accumulator are re-initialised for the next window. Because the gate is on the counter rather than on the clock, pausing `cap_en` mid-window is safe: the window resumes exactly where it left off, which is what test TC6 verifies.

9.4 RTL Source Code

Listing 16: Signal Statistics Unit RTL (`signal_statistics_unit_8bit.v`)

```
`timescale 1ns / 1ps

module signal_statistics_unit_8bit (
    input  clk,
    input  rst,
    input  cap_en,
    input  [7:0] sample,
    output [7:0]  min_val,
    output [7:0]  max_val,
    output [7:0]  rms_val,
    output update_pls
);

    reg [7:0]  min_val_reg;
    reg [7:0]  max_val_reg;
    reg [7:0]  rms_val_reg;
    reg update_pls_reg;


    reg [7:0]  window_minimum;
    reg [7:0]  window_maximum;
    reg [22:0] square_accumulator;
    reg [7:0]  sample_counter;
    reg [14:0] latched_mean_square;

    reg [7:0]  sample_magnitude;
    reg [15:0] sample_square;
    reg [22:0] accumulator_next;
    reg [7:0]  minimum_next;
    reg [7:0]  maximum_next;

    reg [15:0] sqrt_target;
    reg [15:0] sqrt_root;
    reg [15:0] sqrt_remainder;
    reg [15:0] sqrt_test;
    integer    i;

    always @(*) begin
        if (sample[7]) begin
            sample_magnitude = (~sample) + 8'd1;
        end else begin
            sample_magnitude = sample;
        end

        sample_square    = sample_magnitude * sample_magnitude;
        accumulator_next = square_accumulator + {7'd0, sample_square};

        if ((sample ^ 8'h80) < (window_minimum ^ 8'h80)) begin
            minimum_next = sample;
        end else begin
            minimum_next = window_minimum;
        end

        if ((sample ^ 8'h80) > (window_maximum ^ 8'h80)) begin
            maximum_next = sample;
        end else begin
            maximum_next = window_maximum;
        end
    end

    always @(posedge clk) begin
        update_pls_reg <= 1'b0;

        if (rst) begin
            window_minimum      <= 8'h7F;
            window_maximum      <= 8'h80;
            square_accumulator  <= 23'd0;
            sample_counter      <= 8'd0;
            latched_mean_square <= 15'd0;
```

```
            min_val_reg   <= 8'd0;
            max_val_reg   <= 8'd0;
        end else if (cap_en) begin
            if (sample_counter == 8'd255) begin
                min_val_reg   <= minimum_next;
                max_val_reg   <= maximum_next;
                latched_mean_square <= accumulator_next[22:8];
                update_pls_reg <= 1'b1;
                window_minimum      <= 8'h7F;
                window_maximum      <= 8'h80;
                square_accumulator  <= 23'd0;
                sample_counter      <= 8'd0;
            end else begin
                window_minimum      <= minimum_next;
                window_maximum      <= maximum_next;
                square_accumulator  <= accumulator_next;
                sample_counter      <= sample_counter + 8'd1;
            end
        end
    end

    always @(*) begin
        sqrt_target    = {1'b0, latched_mean_square};
        sqrt_root      = 16'd0;
        sqrt_remainder = 16'd0;
        sqrt_test      = 16'd0;

        for (i = 7; i >= 0; i = i - 1) begin
            sqrt_remainder = (sqrt_remainder << 2) | ((sqrt_target >> (i * 2)) & 16'd3);
            sqrt_test      = (sqrt_root << 2) | 16'd1;

            if (sqrt_remainder >= sqrt_test) begin
                sqrt_remainder = sqrt_remainder - sqrt_test;
                sqrt_root      = (sqrt_root << 1) | 16'd1;
            end else begin
                sqrt_root      = sqrt_root << 1;
            end
        end

        rms_val_reg = sqrt_root[7:0];
    end

    assign min_val = min_val_reg;
    assign max_val = max_val_reg;
    assign rms_val = rms_val_reg;
    assign update_pls = update_pls_reg;
endmodule
```

9.5 Self-Checking Testbench

Each check feeds a complete 256-sample window and then compares all three results plus the update strobe. The expected RMS values are computed by hand from the definition, which makes the test an independent check of the arithmetic rather than a restatement of the RTL.

Listing 17: Signal Statistics Unit self-checking testbench (`tb_signal_statistics_unit_8bit.v`)

```
`timescale 1ns / 1ps

module tb_signal_statistics_unit_8bit;

    reg        clk;
    reg        rst;
    reg        cap_en;
    reg  [7:0] sample;
    wire [7:0] min_val;
    wire [7:0] max_val;
    wire [7:0] rms_val;
```

```
    wire       update_pls;

    integer pass_cnt = 0;
    integer fail_cnt = 0;
    integer n;
    integer midwindow_pulse_errors;

    signal_statistics_unit_8bit uut (
        .clk(clk),
        .rst(rst),
        .cap_en(cap_en),
        .sample(sample),
        .min_val(min_val),
        .max_val(max_val),
        .rms_val(rms_val),
        .update_pls(update_pls)
    );

    always #5 clk = ~clk;

    // Feed one full 256-sample window, alternating between two sample values.
    // cap_en is raised together with the first driven sample so no
    // stray clk edge can slip an unintended sample into the window.
    task feed_window;
        input [7:0] value_for_even_samples;
        input [7:0] value_for_odd_samples;
        begin
            for (n = 0; n < 256; n = n + 1) begin
                @(negedge clk);
                cap_en = 1;
                if (n[0]) sample = value_for_odd_samples;
                else      sample = value_for_even_samples;
                @(posedge clk);
                #1;
            end
        end
    endtask

    task check_stats;
        input [8*28:1] test_name;
        input [7:0]    expected_min;
        input [7:0]    expected_max;
        input [7:0]    expected_rms;
        reg ok;
        begin
            ok = (min_val === expected_min) &&
                 (max_val === expected_max) &&
                 (rms_val     === expected_rms) &&
                 (update_pls === 1'b1);
            $display("| %-28s | %02h/%02h/%02h    | %02h/%02h/%02h    |   %b    | %-6s |",
                     test_name,
                     expected_min, expected_max, expected_rms,
                     min_val, max_val, rms_val,
                     update_pls,
                     ok ? "PASS" : "FAIL");
            if (ok) pass_cnt = pass_cnt + 1;
            else    fail_cnt = fail_cnt + 1;
        end
    endtask

    initial begin
        $dumpfile("signal_statistics_unit_8bit_waveform.vcd");
        $dumpvars(0, tb_signal_statistics_unit_8bit);

        clk = 0; rst = 1; cap_en = 0; sample = 8'd0;
        repeat (2) @(posedge clk);
        @(negedge clk); rst = 0;

        $display("\n================================================================================
    ↪ ======");
        $display(" SIGNAL STATISTICS UNIT (MIN / MAX / TRUE-RMS, 256-SAMPLE WINDOW) VERIFICATION SUITE"
    ↪ );
        $display("================================================================================
```

```
    ↪ ====");
        $display("| Test Name                   | Exp mn/mx/rms| Act mn/mx/rms| Pulse | Status |");
        $display("|-----------------------------|--------------|--------------|-------|--------|");

        // Window 1: constant +100. Also confirm no pulse fires mid-window.
        midwindow_pulse_errors = 0;
        for (n = 0; n < 200; n = n + 1) begin
            @(negedge clk); cap_en = 1; sample = 8'd100;
            @(posedge clk); #1;
            if (update_pls === 1'b1) midwindow_pulse_errors = midwindow_pulse_errors + 1;
        end
        for (n = 0; n < 56; n = n + 1) begin
            @(negedge clk); sample = 8'd100;
            @(posedge clk); #1;
        end
        check_stats("TC1: Constant +100", 8'h64, 8'h64, 8'd100);
        if (midwindow_pulse_errors == 0) begin
            pass_cnt = pass_cnt + 1;
            $display("| %-28s | %-12s | %-12s |   -    | PASS   |", "TC2: No mid-window pulse", "0
    ↪ errors", "0 errors");
        end else begin
            fail_cnt = fail_cnt + 1;
            $display("| %-28s | %-12s | %0d errors    |   -    | FAIL   |", "TC2: No mid-window pulse",
    ↪ "0 errors", midwindow_pulse_errors);
        end

        // Window 2: constant -100 (0x9C two's complement)
        feed_window(8'h9C, 8'h9C);
        check_stats("TC3: Constant -100", 8'h9C, 8'h9C, 8'd100);

        // Window 3: square wave +100 / -100
        feed_window(8'd100, 8'h9C);
        check_stats("TC4: Square wave +/-100", 8'h9C, 8'h64, 8'd100);

        // Window 4: alternating 0 / +100, RMS = sqrt(5000) = 70
        feed_window(8'd0, 8'd100);
        check_stats("TC5: Alternating 0/+100", 8'h00, 8'h64, 8'd70);

        // Window 5: cap_en gating must ignore paused samples.
        // 100 samples + pause (10 ignored clocks of 0x55) + 156 samples = 256.
        for (n = 0; n < 100; n = n + 1) begin
            @(negedge clk); cap_en = 1; sample = 8'd100;
            @(posedge clk); #1;
        end
        @(negedge clk); cap_en = 0; sample = 8'h55;
        for (n = 0; n < 10; n = n + 1) begin
            @(posedge clk); #1;
        end
        @(negedge clk); cap_en = 1; sample = 8'd100;
        @(posedge clk); #1;
        for (n = 0; n < 155; n = n + 1) begin
            @(negedge clk); sample = 8'd100;
            @(posedge clk); #1;
        end
        check_stats("TC6: Pause-proof window", 8'h64, 8'h64, 8'd100);

        // Window 6: all-zero input
        feed_window(8'd0, 8'd0);
        check_stats("TC7: All zero input", 8'h00, 8'h00, 8'd0);

        // Window 7: extreme full-scale square wave +127 / -128
        feed_window(8'h7F, 8'h80);
        check_stats("TC8: Full-scale +127/-128", 8'h80, 8'h7F, 8'd127);

        $display("================================================================================
    ↪ ====");
        $display(" TOTAL CHECKS PASSED: %0d/%0d", pass_cnt, pass_cnt + fail_cnt);
        $display(" TOTAL CHECKS FAILED: %0d/%0d", fail_cnt, pass_cnt + fail_cnt);
        $display("================================================================================
    ↪ ====\n");
        $finish;
    end

```

```
151  endmodule
```

9.6 Verification Phase 1: Pure Digital (Vivado XSim)

Check	Window content	Expected min / max / rms	Reasoning
TC1	constant +100	`64 / 64 / 64`	RMS of a constant is the constant
TC2	— (strobe timing)	no mid-window pulse	`update_pls` only at window end
TC3	constant −100	`9c / 9c / 64`	RMS is a magnitude, so it stays +100
TC4	square wave ±100	`9c / 64 / 64`	RMS of a symmetric square wave equals its amplitude
TC5	alternating 0/ + 100	`00 / 64 / 46`	mean square = 5000, $\lfloor\sqrt{5000}\rfloor = 70 =$ `0x46`
TC6	+100 with a mid-window pause	`64 / 64 / 64`	`cap_en` gating must not corrupt the window
TC7	all zeros	`00 / 00 / 00`	degenerate case
TC8	full scale +127/ − 128	`80 / 7f / 7f`	extreme range; accumulator must not overflow

Table 25: The eight window-level checks applied to the Signal Statistics Unit.

TC5 is the strongest arithmetic check applied to this core. A stream alternating between 0 and +100 has mean square $\frac{1}{2}(0^2 + 100^2) = 5000$, whose integer square root is 70. The core returns `0x46` = 70 — confirming the squaring, the accumulation, the power-of-two division *and* the shift-and-subtract root in a single measurement. TC8 confirms the accumulator sizing argument: with the input alternating between the extreme codes the accumulator reaches its theoretical maximum and the reported RMS is `0x7F` = 127, exactly $\lfloor\sqrt{16256}\rfloor$.

```
source tb_signal_statistics_unit_8bit.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window. If you want to open a wave window go to 'File->New Waveform
#   }
# }
# run 1000ns

================================================================================
 SIGNAL STATISTICS UNIT (MIN / MAX / TRUE-RMS, 256-SAMPLE WINDOW) VERIFICATION SUITE
================================================================================
| Test Name                    | Exp mn/mx/rms| Act mn/mx/rms| Pulse | Status |
|------------------------------|--------------|--------------|-------|--------|
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_signal_statistics_unit_8bit_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:02 ; elapsed = 00:00:08 . Memory (MB): peak = 2273.816 ; gain = 0.000
run full
ERROR: [Simtcl 6-18] Time value full is missing the numeric part.
run all
|          TC1: Constant +100 | 64/64/64    | 64/64/64    |   1   |   PASS |
|     TC2: No mid-window pulse |     0 errors |     0 errors |   -   | PASS   |
|          TC3: Constant -100 | 9c/9c/64    | 9c/9c/64    |   1   |   PASS |
|      TC4: Square wave +/-100 | 9c/64/64    | 9c/64/64    |   1   |   PASS |
|      TC5: Alternating 0/+100 | 00/64/46    | 00/64/46    |   1   |   PASS |
|      TC6: Pause-proof window | 64/64/64    | 64/64/64    |   1   |   PASS |
|          TC7: All zero input | 00/00/00    | 00/00/00    |   1   |   PASS |
|    TC8: Full-scale +127/-128 | 80/7f/7f    | 80/7f/7f    |   1   |   PASS |
================================================================================
 TOTAL CHECKS PASSED: 8/8
 TOTAL CHECKS FAILED: 0/8
================================================================================

$finish called at time : 18046 ns : File "C:/Users/Apple/Documents/esim_design/internship/Digital_IP_Design/more_design/4_Signal_Statistics_RMS/tb_signal_statistics_unit_
```

Figure 56: Vivado Tcl console output for the Signal Statistics Unit: **`8/8`** checks pass. Note TC5, where the alternating 0/ + 100 stream returns an RMS of `0x46` (70), and TC8, where the full-scale window returns `0x7f` (127) with no accumulator overflow.

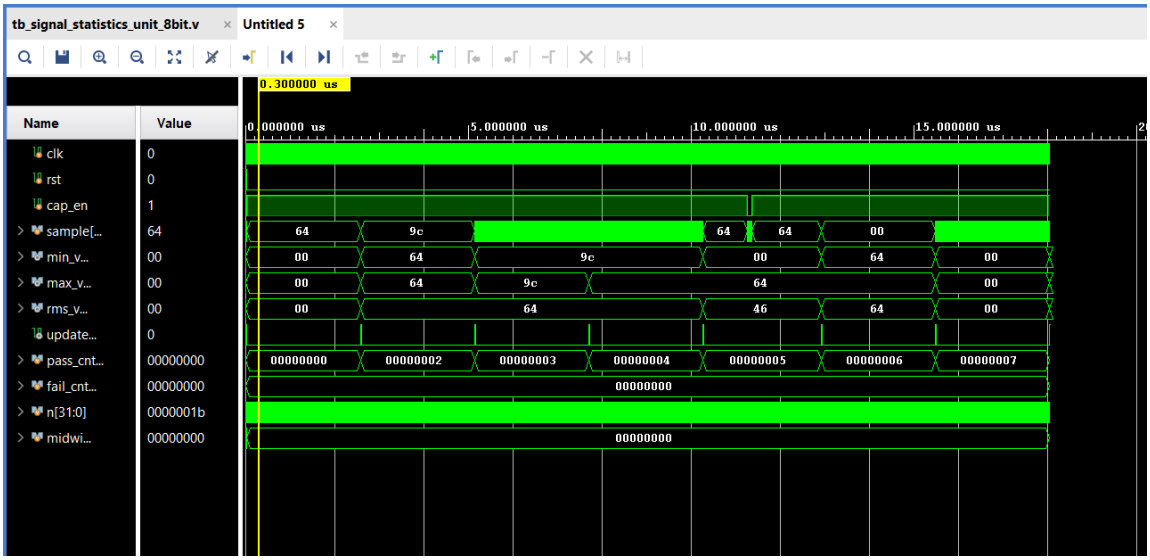


Figure 57: Vivado XSim waveform for the Signal Statistics Unit. The three result buses hold their previous values throughout each window and update simultaneously at the window boundary, coincident with the single-cycle `update_pls` strobe.

9.7 Verification Phase 2: Mixed-Signal (eSim / Ngspice)

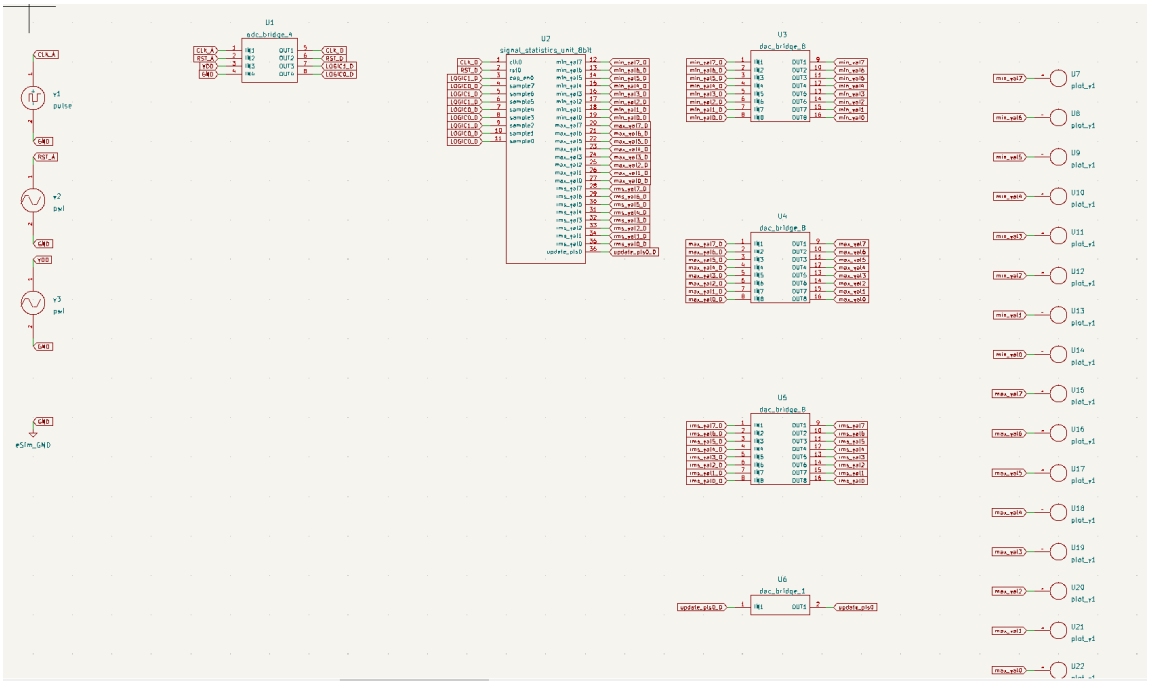


Figure 58: The generated eSim test schematic for the Signal Statistics Unit. The 25 digital outputs require four `dac_bridge` banks (8+8+8+1) and 25 `plot_v1` probes, making this the densest of the test sheets.

9.7.1 Stimulus Definition

Net	Source	Value	Purpose
`CLK_A`	`pulse`	`0 5 0 10n 10n 5u 10u`	100 kHz sample clock (10 μs period)
`RST_A`	`pwl`	`0 5 19u 5 20u 0`	release after two clocks
`VDD`	`pwl`	`0 5 100 5`	`cap_en` high, and the set bits of `sample`
`GND`	—	0 V	the clear bits of `sample`

Table 26: Analog stimulus for the Statistics transient. `sample` is hard-wired to +100 (`0x64`). Transient settings: step 0.1 μs, stop 8 ms.

A constant input of +100 was chosen deliberately: it makes the expected answer **analytically certain** — for a constant stream the minimum, the maximum and the RMS must all equal that constant — so any deviation on any of the three buses is unambiguously a fault. With a 10 μs sample period a 256-sample window completes every $256 \times 10\,\mu\text{s} = 2.56$ ms, so the 8 ms transient spans three complete windows and therefore three update strobes.

9.7.2 Results

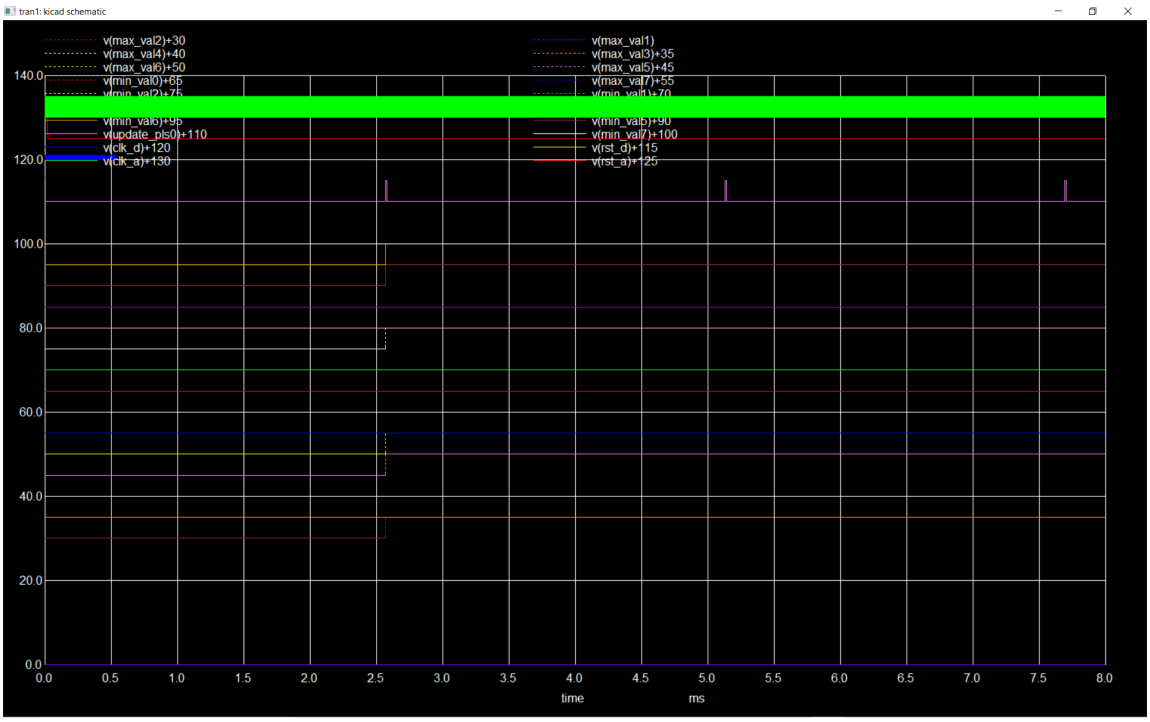


Figure 59: Stacked Ngspice view of the Signal Statistics transient. All three result buses settle to the same code at the first window boundary and hold it; `v(update_pls)` produces one narrow strobe per completed window.

Numerical analysis of the exported data confirms the result bit by bit. On all three output buses, bits 6, 5 and 2 are high and every other bit — including bit 7 and bit 0 — is low:

$$2^6 + 2^5 + 2^2 = 64 + 32 + 4 = 100 = \texttt{0x64}.$$

So `min_val` = `max_val` = `rms_val` = 100, exactly as required for a constant stream. The `update_pls` strobe fires at **2.57 ms, 5.13 ms and 7.69 ms** — three pulses separated by 2.56 ms, matching the 256-sample window length precisely.

This is a particularly significant result: it demonstrates the complete arithmetic chain — signed comparison, magnitude extraction, squaring, 23-bit accumulation, power-of-two division and integer square root — executing correctly **inside an analog SPICE simulation**, with the answer read back as a binary code on analog nodes.

9.8 Conclusion

The Signal Statistics Unit turns eSim from a tool that plots waveforms into one that can *measure* them in-circuit. It reuses the previously validated shift-and-subtract square-root hardware, guarantees overflow-free accumulation by construction, and reduces the window division to a wire by fixing the window at a power of two. All eight window-level checks passed in Vivado, including the demanding $\sqrt{5000} = 70$ and full-scale cases, and the mixed-signal run returned a bit-exact `0x64` on all three result buses with correctly spaced update strobes.

10 Digital IP Datasheet & Verification Report: Seven-Segment Display Driver

10.1 Introduction & Purpose

A measurement is only useful if it can be read. The eSim team already contributed a **Binary-to-BCD (double-dabble) converter**, which turns a binary value into decimal digits — but nothing existed to *display* those digits. This core completes that chain.

The **Seven-Segment Display Driver** accepts four independent hexadecimal digit inputs and drives a standard multiplexed four-digit display: it decodes each digit to segment patterns, scans the digits in sequence, and supports both **common-cathode** and **common-anode** hardware. Combined with the Universal Counter (as a digit source in BCD mode) and the Signal Statistics Unit (as a measurement source), it closes a complete loop: measure an analog node, convert it to decimal, and display it — entirely within eSim.

10.2 The Multiplexing Principle

A four-digit display with individual segment drivers would need $4 \times 8 = 32$ output pins. Real hardware avoids this by **time-division multiplexing**: all four digits share one 8-line segment bus, and four digit-select lines enable exactly one digit at a time. The driver cycles through the digits fast enough that persistence of vision makes all four appear lit simultaneously. Pin count falls from 32 to 12.

This core implements that scheme with a **4-bit prescaler**: the active digit advances when the prescaler wraps, i.e. once every **16 clock cycles**, so a complete four-digit scan takes **64 clock cycles**. The user sets the refresh rate purely by choosing the input clock frequency.

10.3 Architecture & Pin Specifications

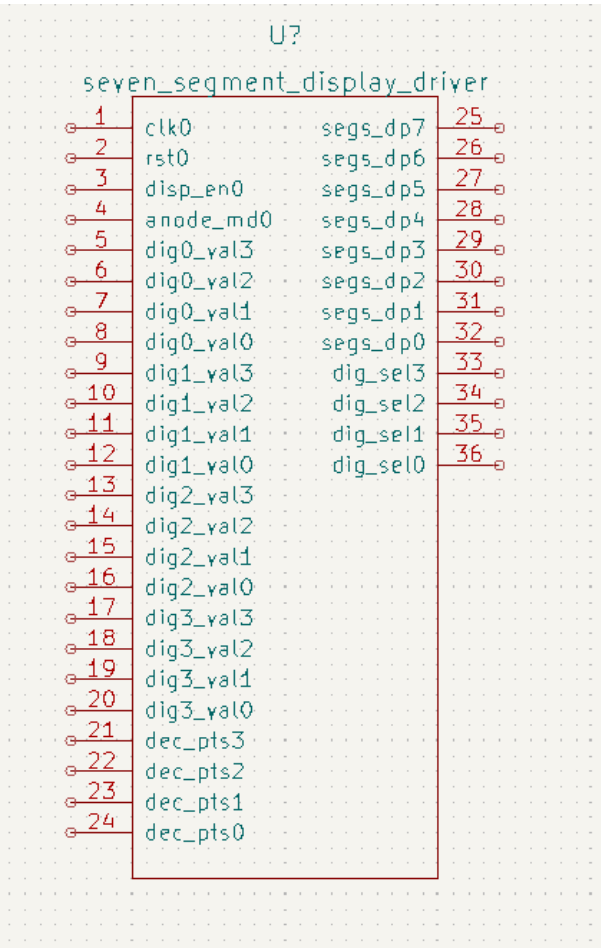


Figure 60: The NgVeri-generated KiCad symbol for `seven_segment_display_driver`: 24 input pins (four 4-bit digit values, four decimal-point flags and four controls) and 12 output pins (the 8-bit segment bus and the 4-bit digit select).

Port	Width	Direction	Function
`clk`	1	in	Scan clock; the digit advances every 16 cycles
`rst`	1	in	Synchronous reset of the scan position
`disp_en`	1	in	Display enable; low blanks the whole display
`anode_md`	1	in	`0` = common cathode, `1` = common anode (inverts all drive)
`dig0_val` … `dig3_val`	4 each	in	Hexadecimal value (`0`–`F`) for each digit
`dec_pts`	4	in	Per-digit decimal-point enable
`segs_dp`	8	out	`[6:0]` = segments a–g, `[7]` = decimal point
`dig_sel`	4	out	One-hot digit select (`0001` → `1000`)

Table 27: Port specification of the Seven-Segment Display Driver.

10.3.1 Segment Bit Ordering

The seven-bit pattern is ordered with segment a in the **most** significant position:

$$\texttt{segs\_dp[6:0]} = \{a, b, c, d, e, f, g\}, \qquad \texttt{segs\_dp[7]} = \text{dp}$$

So `segs_dp[0]` is the **centre bar** g and `segs_dp[6]` is the **top bar** a. This ordering matters when reading the waveforms: the digit `1` lights only segments b and c (pattern `0110000`, with g **off**), whereas `2`, `3` and `4` all require the centre bar (so g is **on**). That single bit therefore distinguishes "1" from the following digits at a glance — a fact used directly in the analysis of the mixed-signal result below.

10.3.2 Decoder, Blanking and Polarity

Three combinational stages act in sequence on every clock:

1. **Digit multiplexer** — selects the active digit's value, its decimal-point flag and its one-hot select pattern from the scan position.
2. **Hexadecimal decoder** — a 16-entry lookup mapping `0`–`F` to segment patterns. Supporting the full hexadecimal range rather than only `0`–`9` costs nothing and makes the block useful for displaying raw register contents.
3. **Blanking, then polarity** — if `disp_en` is low the pattern, the decimal point and the digit select are all forced to zero; **afterwards**, if `anode_md` is high, all three are inverted.

The order of the last two steps is deliberate and is verified explicitly. In common-cathode mode a blanked display drives all lines **low**; in common-anode mode the same blanked state must drive them all **high**, because the polarity inversion applies to the blanked pattern as well. Tests TC7 and TC8 check exactly this pair.

10.4 RTL Source Code

Listing 18: Seven-Segment Display Driver RTL (`seven_segment_display_driver.v`)

```
1  `timescale 1ns / 1ps
2
3  module seven_segment_display_driver (
4      input  clk,
5      input  rst,
6      input  disp_en,
7      input  anode_md,
8      input  [3:0] dig0_val,
9      input  [3:0] dig1_val,
10     input  [3:0] dig2_val,
11     input  [3:0] dig3_val,
12     input  [3:0] dec_pts,
13     output [7:0]  segs_dp,
14     output [3:0]  dig_sel
15 );
16
17     reg [7:0]  segs_dp_reg;
18     reg [3:0]  dig_sel_reg;
19
20
21     reg [3:0] scan_prescaler;
22     reg [1:0] active_digit_index;
23     reg [3:0] active_hex_value;
24     reg [6:0] segment_pattern;
25     reg       active_decimal_point;
26     reg [3:0] digit_select_pattern;
27
28     always @(posedge clk) begin
29         if (rst) begin
30             scan_prescaler     <= 4'd0;
31             active_digit_index <= 2'd0;
```

```
        end else begin
            scan_prescaler <= scan_prescaler + 4'd1;
            if (scan_prescaler == 4'd15) begin
                active_digit_index <= active_digit_index + 2'd1;
            end
        end
    end

    always @(*) begin
        case (active_digit_index)
            2'd0: begin
                active_hex_value     = dig0_val;
                active_decimal_point = dec_pts[0];
                digit_select_pattern = 4'b0001;
            end
            2'd1: begin
                active_hex_value     = dig1_val;
                active_decimal_point = dec_pts[1];
                digit_select_pattern = 4'b0010;
            end
            2'd2: begin
                active_hex_value     = dig2_val;
                active_decimal_point = dec_pts[2];
                digit_select_pattern = 4'b0100;
            end
            default: begin
                active_hex_value     = dig3_val;
                active_decimal_point = dec_pts[3];
                digit_select_pattern = 4'b1000;
            end
        endcase

        case (active_hex_value)
            4'h0: segment_pattern = 7'b1111110;
            4'h1: segment_pattern = 7'b0110000;
            4'h2: segment_pattern = 7'b1101101;
            4'h3: segment_pattern = 7'b1111001;
            4'h4: segment_pattern = 7'b0110011;
            4'h5: segment_pattern = 7'b1011011;
            4'h6: segment_pattern = 7'b1011111;
            4'h7: segment_pattern = 7'b1110000;
            4'h8: segment_pattern = 7'b1111111;
            4'h9: segment_pattern = 7'b1111011;
            4'hA: segment_pattern = 7'b1110111;
            4'hB: segment_pattern = 7'b0011111;
            4'hC: segment_pattern = 7'b1001110;
            4'hD: segment_pattern = 7'b0111101;
            4'hE: segment_pattern = 7'b1001111;
            default: segment_pattern = 7'b1000111;
        endcase

        if (!disp_en) begin
            segment_pattern      = 7'b0000000;
            active_decimal_point = 1'b0;
            digit_select_pattern = 4'b0000;
        end

        if (anode_md) begin
            segment_pattern      = ~segment_pattern;
            active_decimal_point = ~active_decimal_point;
            digit_select_pattern = ~digit_select_pattern;
        end

        segs_dp_reg = {active_decimal_point, segment_pattern};
        dig_sel_reg = digit_select_pattern;
    end

    assign segs_dp = segs_dp_reg;
    assign dig_sel = dig_sel_reg;
endmodule
```

10.5 Self-Checking Testbench

The testbench carries its **own independent copy** of the segment lookup table as a Verilog `function`, so the check compares the DUT against a separately written reference rather than against itself. A `wait_for_select` task synchronises to the scan by advancing until the requested one-hot digit becomes active, which makes the checks independent of the prescaler ratio.

Listing 19: Seven-Segment Display Driver self-checking testbench (`tb_seven_segment_display_driver.v`)

```
`timescale 1ns / 1ps

module tb_seven_segment_display_driver;

    reg        clk;
    reg        rst;
    reg        disp_en;
    reg        anode_md;
    reg  [3:0] dig0_val;
    reg  [3:0] dig1_val;
    reg  [3:0] dig2_val;
    reg  [3:0] dig3_val;
    reg  [3:0] dec_pts;
    wire [7:0] segs_dp;
    wire [3:0] dig_sel;

    integer pass_cnt = 0;
    integer fail_cnt = 0;
    integer wait_count;

    wire [6:0] segments_actual;
    wire       decimal_point_out;
    reg  [3:0] select_before_invert;

    assign segments_actual = segs_dp[6:0];
    assign decimal_point_out = segs_dp[7];

    seven_segment_display_driver uut (
        .clk(clk),
        .rst(rst),
        .disp_en(disp_en),
        .anode_md(anode_md),
        .dig0_val(dig0_val),
        .dig1_val(dig1_val),
        .dig2_val(dig2_val),
        .dig3_val(dig3_val),
        .dec_pts(dec_pts),
        .segs_dp(segs_dp),
        .dig_sel(dig_sel)
    );

    always #5 clk = ~clk;

    // Reference segment table, order {a,b,c,d,e,f,g}, common-cathode polarity
    function [6:0] expected_pattern;
        input [3:0] hex_value;
        begin
            case (hex_value)
                4'h0: expected_pattern = 7'b1111110;
                4'h1: expected_pattern = 7'b0110000;
                4'h2: expected_pattern = 7'b1101101;
                4'h3: expected_pattern = 7'b1111001;
                4'h4: expected_pattern = 7'b0110011;
                4'h5: expected_pattern = 7'b1011011;
                4'h6: expected_pattern = 7'b1011111;
                4'h7: expected_pattern = 7'b1110000;
                4'h8: expected_pattern = 7'b1111111;
                4'h9: expected_pattern = 7'b1111011;
                4'hA: expected_pattern = 7'b1110111;
```

```
                4'hB: expected_pattern = 7'b0011111;
                4'hC: expected_pattern = 7'b1001110;
                4'hD: expected_pattern = 7'b0111101;
                4'hE: expected_pattern = 7'b1001111;
                default: expected_pattern = 7'b1000111;
            endcase
        end
    endfunction

    task log_check;
        input [8*32:1] test_name;
        input [6:0]    expected_segments;
        input          expected_dp;
        input          condition_ok;
        reg ok;
        begin
            ok = condition_ok;
            $display("| %-32s | %07b |  %07b | %b/%b | %-6s |",
                     test_name, expected_segments, segments_actual,
                     expected_dp, decimal_point_out,
                     ok ? "PASS" : "FAIL");
            if (ok) pass_cnt = pass_cnt + 1;
            else    fail_cnt = fail_cnt + 1;
        end
    endtask

    // Scan until the requested digit select line goes active
    task wait_for_select;
        input [3:0] wanted_select;
        begin
            wait_count = 0;
            @(posedge clk); #1;
            while ((dig_sel !== wanted_select) && (wait_count < 500)) begin
                @(posedge clk); #1;
                wait_count = wait_count + 1;
            end
        end
    endtask

    initial begin
        $dumpfile("seven_segment_display_driver_waveform.vcd");
        $dumpvars(0, tb_seven_segment_display_driver);

        clk = 0; rst = 1; disp_en = 1; anode_md = 0;
        dig0_val = 4'h0;
        dig1_val = 4'h1;
        dig2_val = 4'hA;
        dig3_val = 4'hF;
        dec_pts = 4'b0101;
        repeat (2) @(posedge clk);
        @(negedge clk); rst = 0;

        $display("\n================================================================================
    ↪ ==");
        $display(" SEVEN SEGMENT DISPLAY DRIVER (4-DIGIT SCAN) VERIFICATION SUITE");
        $display("================================================================================
    ↪ ");
        $display("| Test Name                       | Exp seg | Act seg  |DP e/a| Status |");
        $display("|---------------------------------|---------|----------|------|--------|");

        wait_for_select(4'b0001);
        log_check("TC1: Digit0 shows 0, dp on", expected_pattern(4'h0), 1'b1,
                  (segments_actual === expected_pattern(4'h0)) && (decimal_point_out === 1'b1));

        wait_for_select(4'b0010);
        log_check("TC2: Digit1 shows 1, dp off", expected_pattern(4'h1), 1'b0,
                  (segments_actual === expected_pattern(4'h1)) && (decimal_point_out === 1'b0));

        wait_for_select(4'b0100);
        log_check("TC3: Digit2 shows A, dp on", expected_pattern(4'hA), 1'b1,
                  (segments_actual === expected_pattern(4'hA)) && (decimal_point_out === 1'b1));

        wait_for_select(4'b1000);
```

```
131         log_check("TC4: Digit3 shows F, dp off", expected_pattern(4'hF), 1'b0,
132                   (segments_actual === expected_pattern(4'hF)) && (decimal_point_out === 1'b0));
133
134         // Live hex value change on the active digit
135         dig3_val = 4'h7;
136         #2;
137         log_check("TC5: Live update digit3 to 7", expected_pattern(4'h7), 1'b0,
138                   (segments_actual === expected_pattern(4'h7)) && (decimal_point_out === 1'b0));
139
140         // Common-anode polarity inversion (checked combinationally, no clock edge)
141         wait_for_select(4'b0001);
142         select_before_invert = dig_sel;
143         anode_md = 1;
144         #2;
145         log_check("TC6: Common-anode inversion", ~expected_pattern(4'h0), 1'b0,
146                   (segments_actual === ~expected_pattern(4'h0)) &&
147                   (decimal_point_out === 1'b0) &&
148                   (dig_sel === ~select_before_invert));
149         anode_md = 0;
150
151         // Blanking in common-cathode mode: everything low
152         disp_en = 0;
153         #2;
154         log_check("TC7: Blanked (cathode mode)", 7'b0000000, 1'b0,
155                   (segments_actual === 7'b0000000) &&
156                   (decimal_point_out === 1'b0) &&
157                   (dig_sel === 4'b0000));
158
159         // Blanking in common-anode mode: everything high
160         anode_md = 1;
161         #2;
162         log_check("TC8: Blanked (anode mode)", 7'b1111111, 1'b1,
163                   (segments_actual === 7'b1111111) &&
164                   (decimal_point_out === 1'b1) &&
165                   (dig_sel === 4'b1111));
166
167         $display("================================================================================
    ↪ ");
168         $display(" TOTAL CHECKS PASSED: %0d/%0d", pass_cnt, pass_cnt + fail_cnt);
169         $display(" TOTAL CHECKS FAILED: %0d/%0d", fail_cnt, pass_cnt + fail_cnt);
170         $display("================================================================================
    ↪ \n");
171         $finish;
172      end
173
174  endmodule
```

10.6 Verification Phase 1: Pure Digital (Vivado XSim)

Check	Feature exercised	Expected
TC1–TC4	Scan sequence and decoding	digits `0`, `1`, `A`, `F` appear on selects `0001`–`1000` with the correct decimal points
TC5	Live update	changing `dig3_val` to `7` takes effect immediately (combinational path)
TC6	Common-anode inversion	segments, decimal point **and** digit select all invert
TC7	Blanking, cathode mode	every line driven low
TC8	Blanking, anode mode	every line driven high

Table 28: The eight checks applied to the Seven-Segment Display Driver.

The four digits chosen for TC1–TC4 (`0`, `1`, `A`, `F`) are not arbitrary: `0` exercises six segments at once, `1` the sparsest pattern, `A` a letter beyond the decimal range, and `F` the last table entry — so a lookup-table indexing error at either end of the range would be caught. The alternating decimal-point pattern (`0101`) simultaneously confirms that the decimal point tracks the *active* digit rather than being static.

```
source tb_seven_segment_display_driver.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window. If you want to open a wave window go to 'File->New Waveform
#   }
# }
# run 1000ns

================================================================================
 SEVEN SEGMENT DISPLAY DRIVER (4-DIGIT SCAN) VERIFICATION SUITE
================================================================================
| Test Name                      | Exp seg | Act seg  |DP e/a| Status |
|--------------------------------|---------|----------|------|--------|
|       TC1: Digit0 shows 0, dp on | 1111110 |  1111110 | 1/1 |   PASS |
|      TC2: Digit1 shows 1, dp off | 0110000 |  0110000 | 0/0 |   PASS |
|       TC3: Digit2 shows A, dp on | 1110111 |  1110111 | 1/1 |   PASS |
|      TC4: Digit3 shows F, dp off | 1000111 |  1000111 | 0/0 |   PASS |
|     TC5: Live update digit3 to 7 | 1110000 |  1110000 | 0/0 |   PASS |
|      TC6: Common-anode inversion | 0000001 |  0000001 | 0/0 |   PASS |
|      TC7: Blanked (cathode mode) | 0000000 |  0000000 | 0/0 |   PASS |
|        TC8: Blanked (anode mode) | 1111111 |  1111111 | 1/1 |   PASS |
================================================================================
 TOTAL CHECKS PASSED: 8/8
 TOTAL CHECKS FAILED: 0/8
================================================================================

$finish called at time : 662 ns : File "C:/Users/Apple/Documents/esim_design/internship/Digital_IP_Design/more_design/5_Seven_Segment_Driver/tb_seven_segment_display_drive
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_seven_segment_display_driver_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:04 ; elapsed = 00:00:07 . Memory (MB): peak = 1223.578 ; gain = 0.000
```

Figure 61: Vivado Tcl console output for the Seven-Segment Display Driver: **8/8** checks pass. The expected and actual seven-bit segment patterns match exactly for every digit, and both blanking polarities behave correctly.

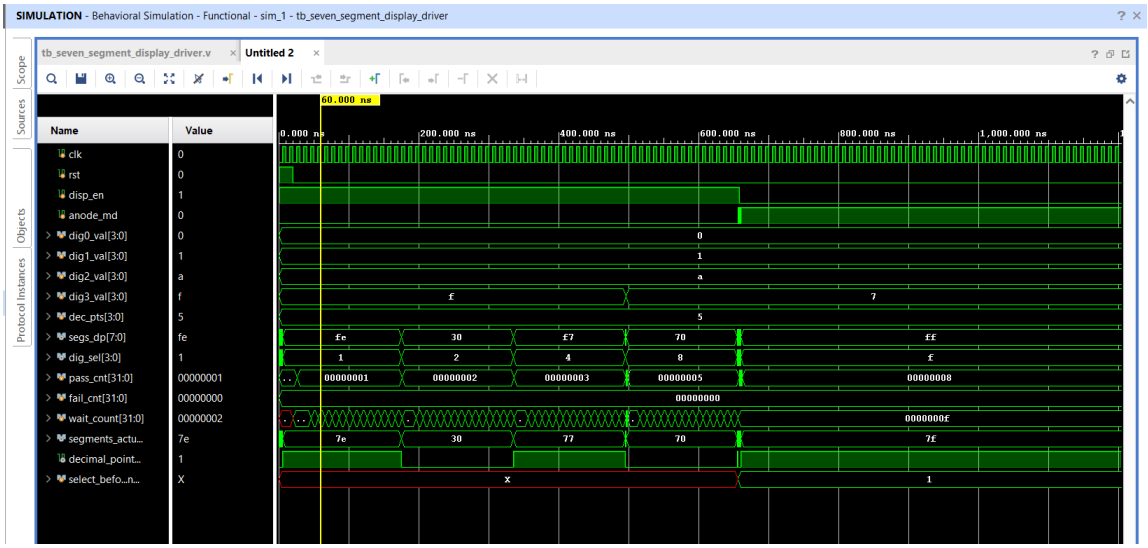


Figure 62: Vivado XSim waveform for the Seven-Segment Display Driver, showing the one-hot `dig_sel` rotating through the four digits and the `segs_dp` bus changing to the corresponding pattern in lock-step.

10.7 Verification Phase 2: Mixed-Signal (eSim / Ngspice)

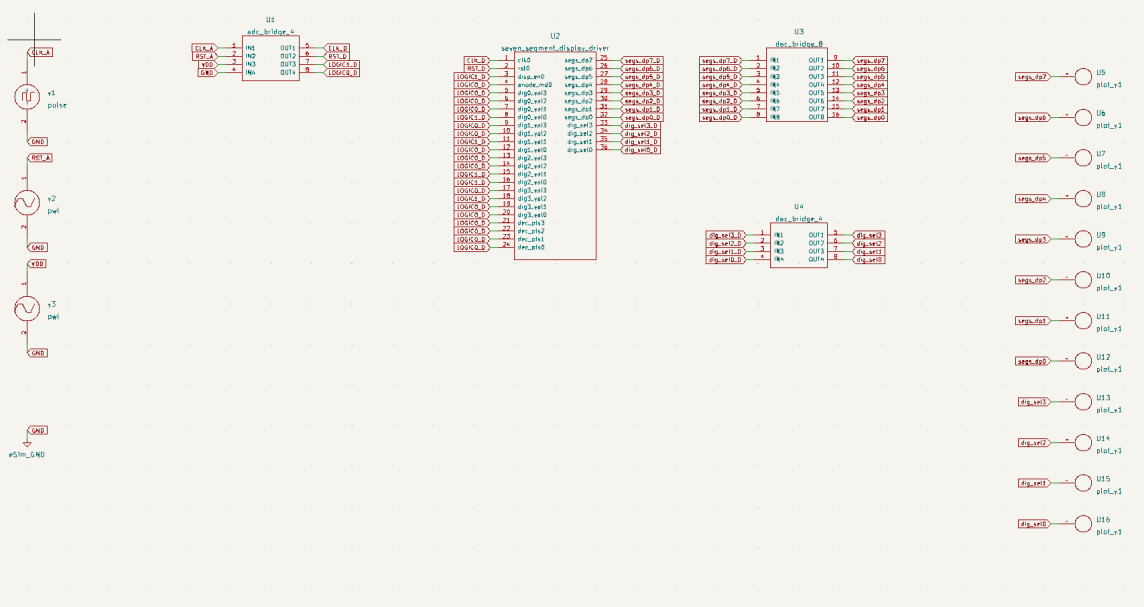


Figure 63: The generated eSim test schematic for the Seven-Segment Display Driver. The four digit values are hard-wired through the `VDD`/`GND` rails, so the sheet needs only three analog sources despite the core having 24 input pins.

10.7.1 Stimulus Definition

Net	Source	Value	Purpose
`CLK_A`	`pulse`	`0 5 0 10n 10n 5u 10u`	100 kHz scan clock (10 μs period)
`RST_A`	`pwl`	`0 5 19u 5 20u 0`	reset the scan position
`VDD`	`pwl`	`0 5 100 5`	`disp_en` high and the set digit bits
`GND`	—	0 V	`anode_md` low, `dec_pts` = 0, clear digit bits

Table 29: Analog stimulus for the Seven-Segment transient. The digits are wired to **1, 2, 3, 4**. Transient settings: step 0.1 μs, stop 3 ms.

The digits are set to the ascending sequence **1, 2, 3, 4** so that each scan position produces a *different* segment pattern; a scan bug that repeated one digit, or a decoder bug that ignored the digit index, would be immediately visible. With a 10 μs clock the digit advances every $16 \times 10\,\mu\text{s} = 160\,\mu\text{s}$ and a complete scan takes 640 μs, so the 3 ms transient covers roughly **4.7 full scans**.

10.7.2 Results

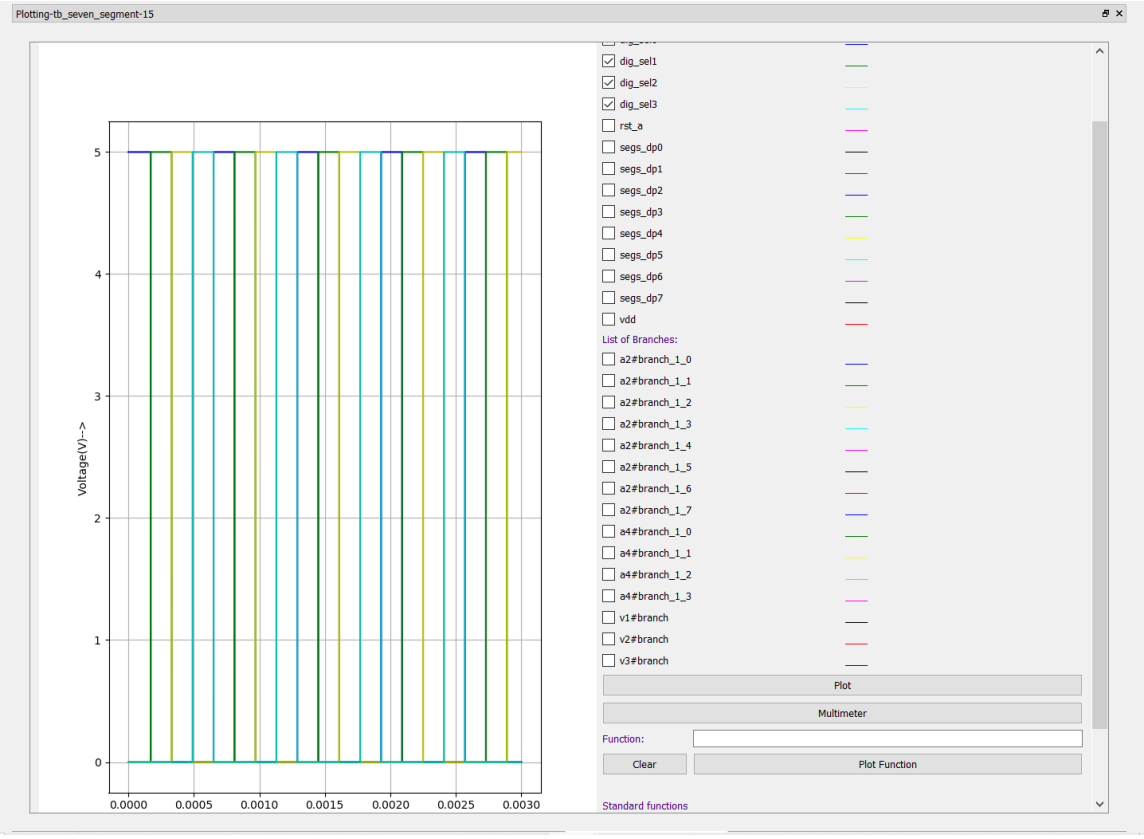


Figure 64: Ngspice output for the four `dig_sel` lines. The one-hot pattern advances `0001` $\rightarrow$ `0010` $\rightarrow$ `0100` $\rightarrow$ `1000` and repeats, with exactly one line active at any instant and each dwelling for 160 μs.

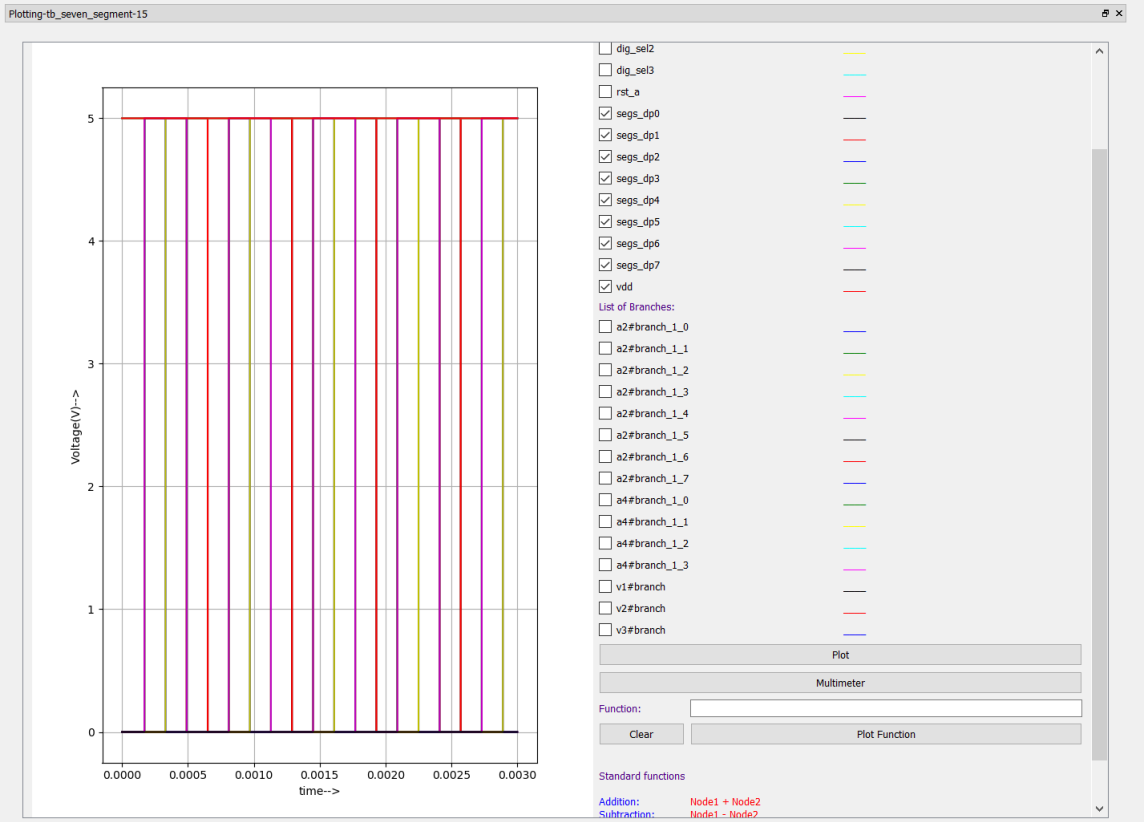


Figure 65: Ngspice output for the eight `segs_dp` lines. The pattern changes at every digit boundary, confirming that the decoder output tracks the active scan position rather than holding a fixed value.

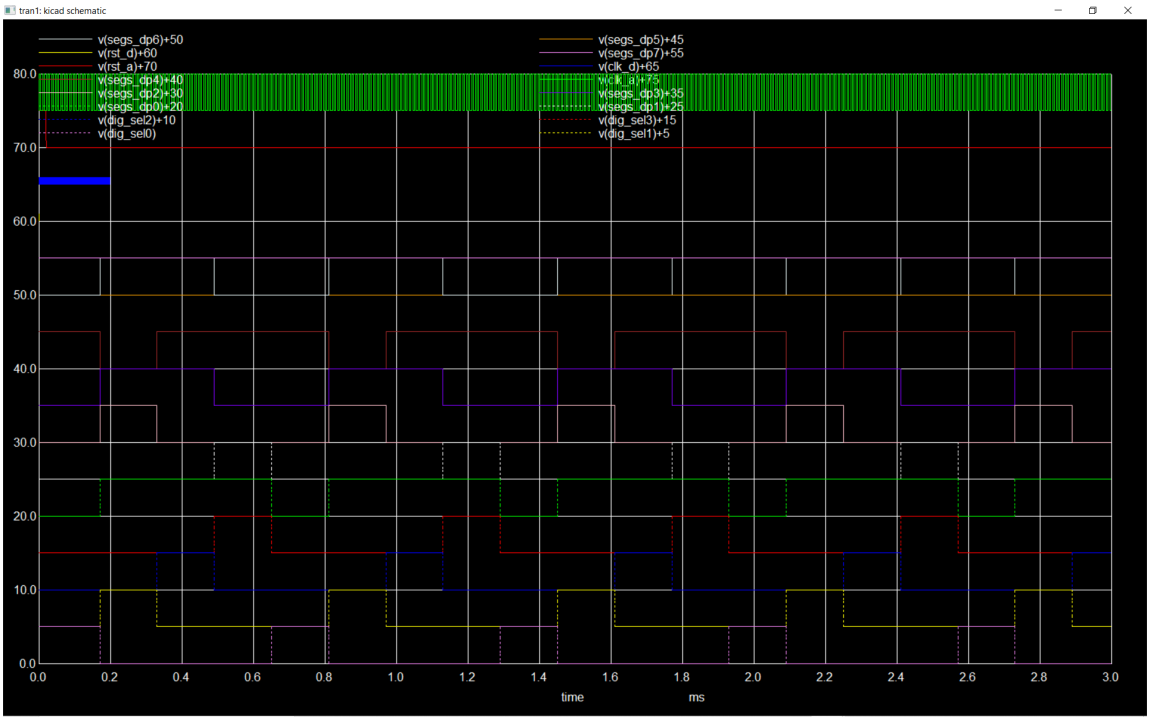


Figure 66: Combined stacked Ngspice view of the Seven-Segment transient, showing the digit-select scan and the segment bus together so the correspondence between scan position and segment pattern can be read directly.

Numerical analysis of the transient confirms both the scanning and the decoding. The four `dig_sel` lines activate strictly in sequence with a **160 μs** dwell each, exactly the predicted 16 clock periods, and never overlap.

More importantly, the segment data proves the **decoder** is working and not merely the scanner. Trace `segs_dp0` — the centre bar g — is **low** during the digit showing `1` and **high** during the digits showing `2`, `3` and `4`. That is precisely the expected behaviour, since `1` is the only one of those four digits whose glyph does not use the centre bar. A driver that scanned correctly but decoded incorrectly could not produce that pattern.

10.8 Conclusion

The Seven-Segment Display Driver provides eSim with the output stage its digit-generation blocks previously lacked. Scan sequencing, full hexadecimal decoding, per-digit decimal points, display blanking and both display polarities were verified across eight digital checks, and the mixed-signal run reproduced both the 160 μs scan cadence and the digit-dependent segment patterns. Together with the Universal Counter and the Signal Statistics Unit it enables a complete measure–convert–display chain inside a single eSim schematic.

11 Digital IP Datasheet & Verification Report: TMR Majority Voter

11.1 Introduction & Purpose

The final core addresses **dependability**. In avionics, spacecraft, nuclear instrumentation and safety-rated industrial control, a single sensor or logic channel is not trusted on its own. **Triple Modular Redundancy** (TMR) replicates the channel three times and votes on the results: as long as no more than one channel is faulty, the majority is correct, so the fault is **masked** rather than merely detected.

The eSim library contained fault-detection blocks — a clock-failure detector and a lockstep comparator — but both *detect and halt*. A lockstep pair can tell that two channels disagree, but it cannot tell *which* one is wrong, so its only safe response is to stop. A three-channel voter can identify the deviating channel and **keep operating**. That difference between fail-stop and fail-operational is the reason TMR exists, and it is what this core adds.

The intended eSim application is redundant analog acquisition: three sensors measuring the same physical quantity, each digitised, with the voter producing a trustworthy result plus per-channel diagnostics.

11.2 Architecture & Pin Specifications

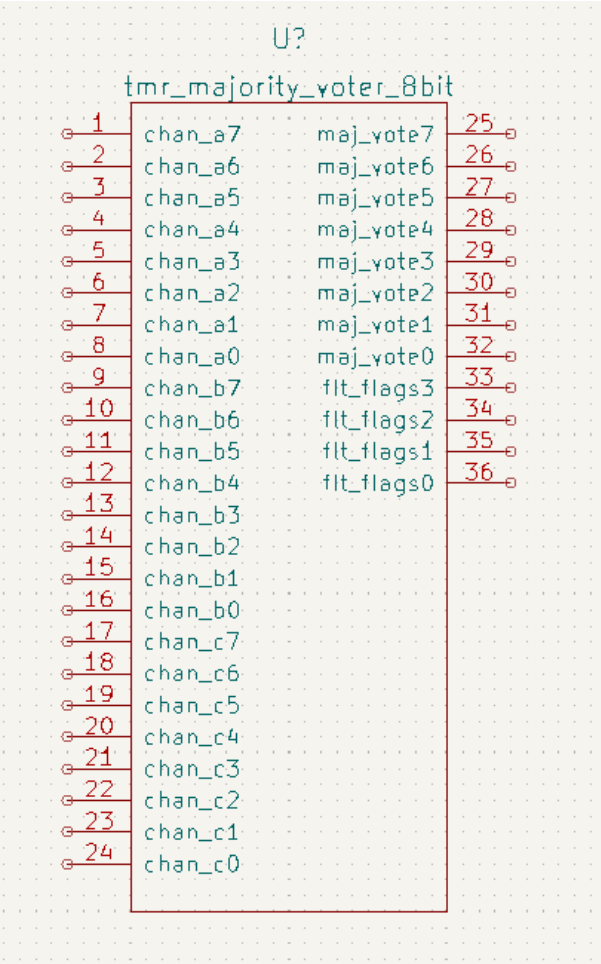


Figure 67: The NgVeri-generated KiCad symbol for `tmr_majority_voter_8bit`: three 8-bit input channels (24 pins) producing the 8-bit voted output and four fault flags. The core is purely combinational and has no clock or reset.

Port	Width	Direction	Function
`chan_a`	8	in	Redundant channel A
`chan_b`	8	in	Redundant channel B
`chan_c`	8	in	Redundant channel C
`maj_vote`	8	out	Bitwise 2-of-3 majority result
`flt_flags`	4	out	`[0]` A faulty, `[1]` B faulty, `[2]` C faulty, `[3]` any fault

Table 30: Port specification of the TMR Majority Voter. Note the fault-flag bit assignment, which is used when interpreting the waveforms.

11.3 Working Principle

11.3.1 Bitwise Majority

The voter is purely combinational and evaluates, for all eight bit positions in parallel:

$$\texttt{maj\_vote} = (\texttt{chan\_a} \wedge \texttt{chan\_b}) \vee (\texttt{chan\_b} \wedge \texttt{chan\_c}) \vee (\texttt{chan\_a} \wedge \texttt{chan\_c})$$

Each product term detects one agreeing pair, so a bit of the result is 1 exactly when at least two of the three inputs have that bit set. The voting is deliberately **per-bit** rather than per-word: if channel A is corrupted in bit 3 and channel C in bit 6, a word-level voter would find no majority at all, whereas the bitwise voter still recovers the correct value in both positions. This makes the design robust against the independent single-bit upsets that radiation-induced faults actually produce.

11.3.2 Diagnostic Flags

Masking a fault silently is dangerous — a system could exhaust its redundancy without anyone noticing. Each channel is therefore compared against the voted result, and any mismatch raises that channel's flag; `flt_flags[3]` is the OR of the three and serves as a single "attention" line. Because the flags are derived from the *voted* value rather than from pairwise comparisons, the deviating channel is named unambiguously whenever a majority exists.

11.3.3 Behaviour When All Three Disagree

If all three channels differ, no bit-position majority need exist, and the voter reports all three channels as faulty (`flt_flags[2:0]` = `111`). The output is then the bitwise majority of the individual bits, which is still well defined but should not be trusted — and the flags say so. Test TC5 covers exactly this case rather than leaving it undefined.

11.4 RTL Source Code

Listing 20: TMR Majority Voter RTL (`tmr_majority_voter_8bit.v`)

```
1  `timescale 1ns / 1ps
2
3  module tmr_majority_voter_8bit (
4      input  [7:0] chan_a,
5      input  [7:0] chan_b,
6      input  [7:0] chan_c,
7      output [7:0]  maj_vote,
8      output [3:0]  flt_flags
9  );
10
11     reg [7:0]  maj_vote_reg;
12     reg [3:0]  flt_flags_reg;
13
14
15     always @(*) begin
16         maj_vote_reg = (chan_a & chan_b) |
17                        (chan_b & chan_c) |
18                        (chan_a & chan_c);
19
20         flt_flags_reg[0] = (chan_a != maj_vote_reg);
21         flt_flags_reg[1] = (chan_b != maj_vote_reg);
22         flt_flags_reg[2] = (chan_c != maj_vote_reg);
23
24         flt_flags_reg[3] = flt_flags_reg[0] | flt_flags_reg[1] | flt_flags_reg[2];
25     end
26
27     assign maj_vote = maj_vote_reg;
28     assign flt_flags = flt_flags_reg;
29 endmodule
```

11.5 Self-Checking Testbench

Because the voter is combinational and narrow, it can be verified far more aggressively than the sequential cores. The testbench combines an **exhaustive agreement sweep** across all 256 byte values, eight directed fault-injection cases, and a **100-vector randomised regression** in which the expected result and all four flags are recomputed independently for every vector.

Listing 21: TMR Majority Voter self-checking testbench (`tb_tmr_majority_voter_8bit.v`)

```
`timescale 1ns / 1ps

module tb_tmr_majority_voter_8bit;

    reg  [7:0] chan_a;
    reg  [7:0] chan_b;
    reg  [7:0] chan_c;
    wire [7:0] maj_vote;
    wire [3:0] flt_flags;




    integer pass_cnt = 0;
    integer fail_cnt = 0;
    integer k;
    integer mismatch;

    reg [7:0] expected_vote;
    reg       expected_fault_a;
    reg       expected_fault_b;
    reg       expected_fault_c;

    tmr_majority_voter_8bit uut (
        .chan_a(chan_a),
        .chan_b(chan_b),
        .chan_c(chan_c),
        .maj_vote(maj_vote),
        .flt_flags(flt_flags)



    );

    task apply_and_check;
        input [8*30:1] test_name;
        input [7:0] value_a;
        input [7:0] value_b;
        input [7:0] value_c;
        reg ok;
        begin
            chan_a = value_a;
            chan_b = value_b;
            chan_c = value_c;
            #10;
            expected_vote    = (value_a & value_b) | (value_b & value_c) | (value_a & value_c);
            expected_fault_a = (value_a != expected_vote);
            expected_fault_b = (value_b != expected_vote);
            expected_fault_c = (value_c != expected_vote);
            ok = (maj_vote === expected_vote) &&
                 (flt_flags[0] === expected_fault_a) &&
                 (flt_flags[1] === expected_fault_b) &&
                 (flt_flags[2] === expected_fault_c) &&
                 (flt_flags[3] === (expected_fault_a | expected_fault_b | expected_fault_c));
            $display("| %-30s | %02h %02h %02h |   %02h    |   %02h   | %b%b%b/%b%b%b | %-6s |",
                     test_name, value_a, value_b, value_c,
                     expected_vote, maj_vote,
                     expected_fault_a, expected_fault_b, expected_fault_c,
                     flt_flags[0], flt_flags[1], flt_flags[2],
```

```
                    ok ? "PASS" : "FAIL");
            if (ok) pass_cnt = pass_cnt + 1;
            else    fail_cnt = fail_cnt + 1;
        end
    endtask

    initial begin
        $dumpfile("tmr_majority_voter_8bit_waveform.vcd");
        $dumpvars(0, tb_tmr_majority_voter_8bit);

        chan_a = 8'd0;
        chan_b = 8'd0;
        chan_c = 8'd0;

        $display("\n================================================================================================
    ↪ ======");
        $display(" TMR MAJORITY VOTER (2-OF-3, BITWISE) VERIFICATION SUITE");
        $display("================================================================================================
    ↪ ====");
        $display("| Test Name                      | A  B  C  | Exp Vote | Act Vote| Flt e/a | Status |
    ↪ ");
        $display("|--------------------------------|----------|----------|---------|---------|--------|
    ↪ ");

        // Exhaustive agreement sweep: all channels equal for every byte value
        mismatch = 0;
        for (k = 0; k < 256; k = k + 1) begin
            chan_a = k[7:0];
            chan_b = k[7:0];
            chan_c = k[7:0];
            #2;
            if (maj_vote !== k[7:0]) mismatch = mismatch + 1;
            if (flt_flags[3] !== 1'b0)      mismatch = mismatch + 1;
        end
        if (mismatch == 0) begin
            pass_cnt = pass_cnt + 1;
            $display("| %-30s | sweep    | 0 errors | 0 errors|   -     | PASS   |", "TC1: 256-value
    ↪ agreement");
        end else begin
            fail_cnt = fail_cnt + 1;
            $display("| %-30s | sweep    | 0 errors | %0d errors|  -     | FAIL   |", "TC1: 256-value
    ↪ agreement", mismatch);
        end

        apply_and_check("TC2: Channel C corrupted", 8'h55, 8'h55, 8'hAA);
        apply_and_check("TC3: Channel A corrupted", 8'h00, 8'hF0, 8'hF0);
        apply_and_check("TC4: Channel B single bit", 8'hF0, 8'hF1, 8'hF0);
        apply_and_check("TC5: All channels differ", 8'h0F, 8'hF0, 8'hFF);
        apply_and_check("TC6: All zero", 8'h00, 8'h00, 8'h00);
        apply_and_check("TC7: All ones", 8'hFF, 8'hFF, 8'hFF);
        apply_and_check("TC8: Two low one high", 8'h00, 8'h00, 8'hFF);

        // Randomized regression sweep
        mismatch = 0;
        for (k = 0; k < 100; k = k + 1) begin
            chan_a = $random;
            chan_b = $random;
            chan_c = ($random % 4 == 0) ? chan_a : $random;
            #2;
            expected_vote = (chan_a & chan_b) |
                            (chan_b & chan_c) |
                            (chan_a & chan_c);
            if (maj_vote !== expected_vote) mismatch = mismatch + 1;
            if (flt_flags[0] !== (chan_a !== expected_vote)) mismatch = mismatch + 1;
            if (flt_flags[1] !== (chan_b !== expected_vote)) mismatch = mismatch + 1;
            if (flt_flags[2] !== (chan_c !== expected_vote)) mismatch = mismatch + 1;
        end
        if (mismatch == 0) begin
            pass_cnt = pass_cnt + 1;
            $display("| %-30s | random   | 0 errors | 0 errors|   -     | PASS   |", "TC9: 100 random
    ↪ vectors");
        end else begin
            fail_cnt = fail_cnt + 1;
```

```
126             $display("| %-30s | random   | 0 errors | %0d errors|  -     | FAIL   |", "TC9: 100 random
    ↪ vectors", mismatch);
127         end
128
129         $display("================================================================================
    ↪ ====");
130         $display(" TOTAL CHECKS PASSED: %0d/%0d", pass_cnt, pass_cnt + fail_cnt);
131         $display(" TOTAL CHECKS FAILED: %0d/%0d", fail_cnt, pass_cnt + fail_cnt);
132         $display("================================================================================
    ↪ ====\n");
133         $finish;
134     end
135
136 endmodule
```

11.6 Verification Phase 1: Pure Digital (Vivado XSim)

Check	Scenario	Expected vote	Expected flags (A,B,C)
TC1	all three agree, 256 values swept	the common value	none
TC2	channel C corrupted (55/55/AA)	55	C only
TC3	channel A corrupted (00/F0/F0)	F0	A only
TC4	channel B off by one bit (F0/F1/F0)	F0	B only
TC5	all three differ (0F/F0/FF)	bitwise majority	all three
TC6	all zero	00	none
TC7	all ones	FF	none
TC8	two low, one high	00	C only
TC9	100 random vectors	recomputed per vector	recomputed per vector

Table 31: The nine checks applied to the TMR Majority Voter, spanning an exhaustive sweep, directed fault injection and randomised regression.

TC4 is the most demanding directed case: channel B differs from the other two in a **single bit**. The voter must still return the correct byte and must flag B alone — which it does. Taken together the sweep and the random regression amount to **several hundred independent vector evaluations**, all correct.

```
source tb_tmr_majority_voter_8bit.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window. If you want
#   }
# }
# run 1000ns

================================================================================
 TMR MAJORITY VOTER (2-OF-3, BITWISE) VERIFICATION SUITE
================================================================================
| Test Name                      | A  B  C  | Exp Vote | Act Vote| Flt e/a | Status |
|--------------------------------|----------|----------|---------|---------|--------|
|       TC1: 256-value agreement | sweep    | 0 errors | 0 errors|    -    | PASS   |
|       TC2: Channel C corrupted | 55 55 aa |   55     |   55    | 001/001 |   PASS |
|       TC3: Channel A corrupted | 00 f0 f0 |   f0     |   f0    | 100/100 |   PASS |
|       TC4: Channel B single bit | f0 f1 f0 |   f0     |   f0    | 010/010 |   PASS |
|       TC5: All channels differ | 0f f0 ff |   ff     |   ff    | 110/110 |   PASS |
|                  TC6: All zero | 00 00 00 |   00     |   00    | 000/000 |   PASS |
|                  TC7: All ones | ff ff ff |   ff     |   ff    | 000/000 |   PASS |
|          TC8: Two low one high | 00 00 ff |   00     |   00    | 001/001 |   PASS |
|        TC9: 100 random vectors | random   | 0 errors | 0 errors|    -    | PASS   |
================================================================================
 TOTAL CHECKS PASSED: 9/9
 TOTAL CHECKS FAILED: 0/9
================================================================================

$finish called at time : 782 ns : File "C:/Users/Apple/Documents/esim_design/internship/Digital_IP_Design/more_design/6_TMF
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_tmr_majority_voter_8bit_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:04 ; elapsed = 00:00:10 . Memory (MB): peak = 1219.000 ; gain = 1.484
```

Figure 68: Vivado Tcl console output for the TMR Majority Voter: **9/9** checks pass, including the 256-value agreement sweep (0 errors) and the 100-vector randomised regression (0 errors). The flag columns confirm the faulty channel is identified correctly in every directed case.

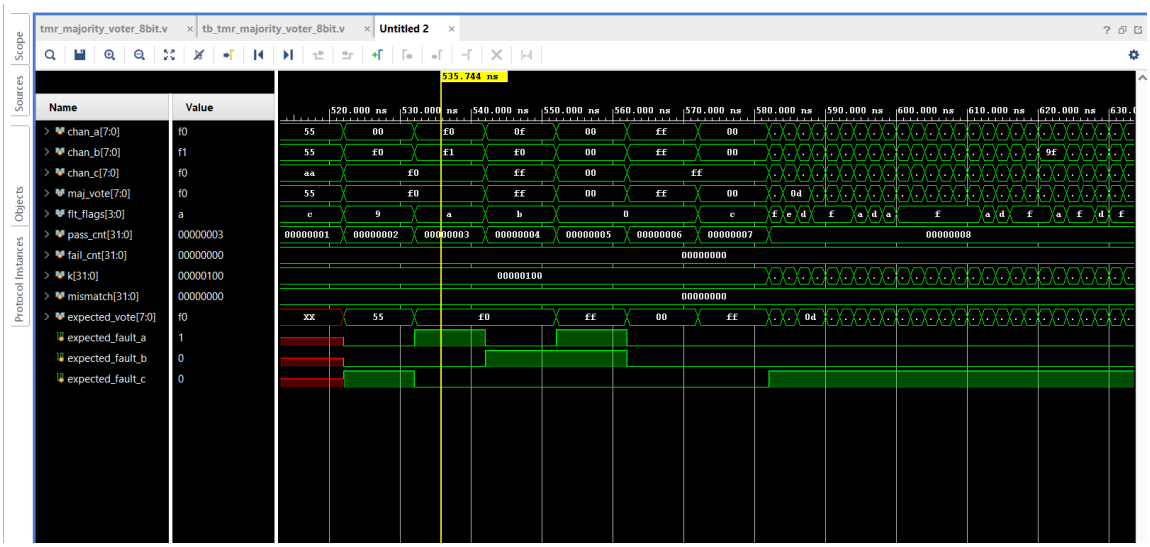


Figure 69: Vivado XSim waveform of the TMR Majority Voter (first capture): during the exhaustive agreement sweep the voted output tracks the common input value exactly and no fault flag is raised.

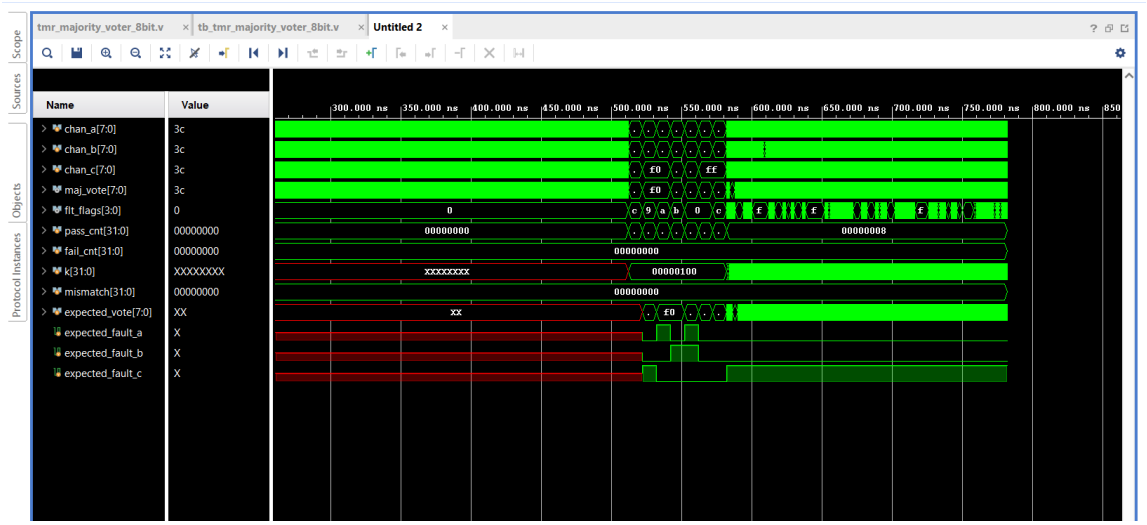


Figure 70: Vivado XSim waveform of the TMR Majority Voter (second capture): during fault injection the voted output holds the correct majority value while the flag of the deviating channel asserts.

11.7 Verification Phase 2: Mixed-Signal (eSim / Ngspice)

The mixed-signal test was designed to demonstrate the **defining property** of TMR: that a channel can fail *during operation* without disturbing the output.

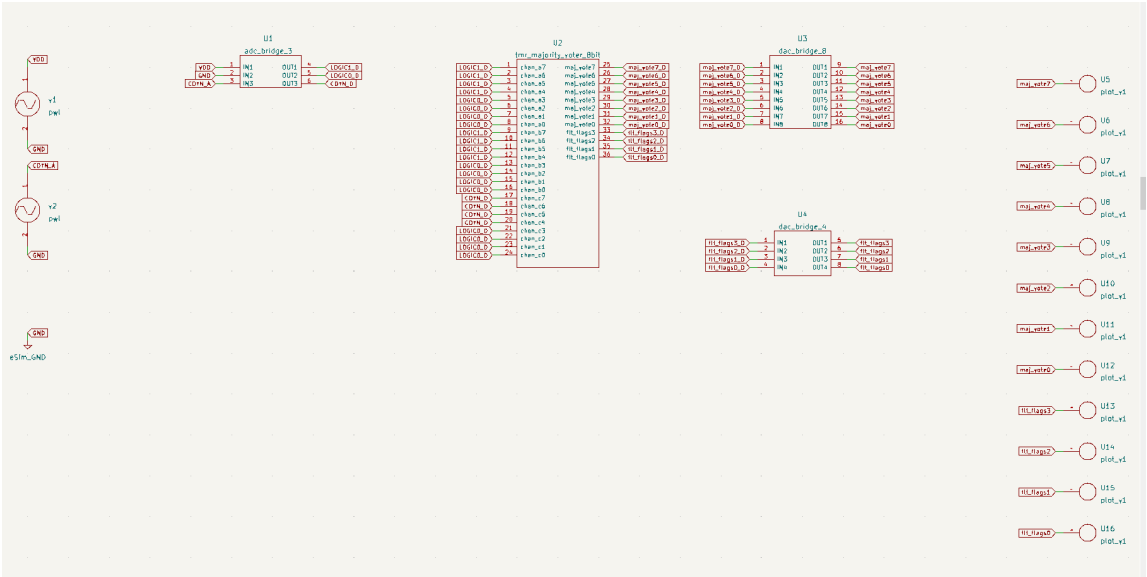


Figure 71: The generated eSim test schematic for the TMR Majority Voter. Channels A and B are tied to a constant pattern through the `VDD`/`GND` rails, while the upper nibble of channel C is driven by a single `pwl` source that fails part-way through the run.

11.7.1 Stimulus Definition

Net	Source	Value	Purpose
`VDD`	`pwl`	`0 5 100 5`	holds `chan_a` = `chan_b` = `0xF0`
`CDYN_A`	`pwl`	`0 5 24.9m 5 25m 0`	drives the upper nibble of `chan_c`; **fails at 25 ms**
`GND`	—	0 V	lower nibble of all three channels

Table 32: Analog stimulus for the TMR transient. Transient settings: step 0.1 ms, stop 50 ms.

Channels A and B are held permanently at `0xF0`. Channel C is initially identical, so for the first 25 ms all three channels agree. At **25 ms** the `CDYN` source collapses to 0 V, driving `chan_c` to `0x00` — a catastrophic, total failure of one channel, in the middle of the run. Because the voter is purely combinational, no clock is required and the schematic contains no clock source at all.

11.7.2 Expected Behaviour

- `maj_vote` must remain `0xF0` **throughout**, before and after the failure, because A and B still form a majority.
- `flt_flags[2]` (channel C) must assert at 25 ms.
- `flt_flags[3]` (any fault) must assert at 25 ms.
- `flt_flags[0]` and `flt_flags[1]` (channels A and B) must **never** assert.

11.7.3 Results

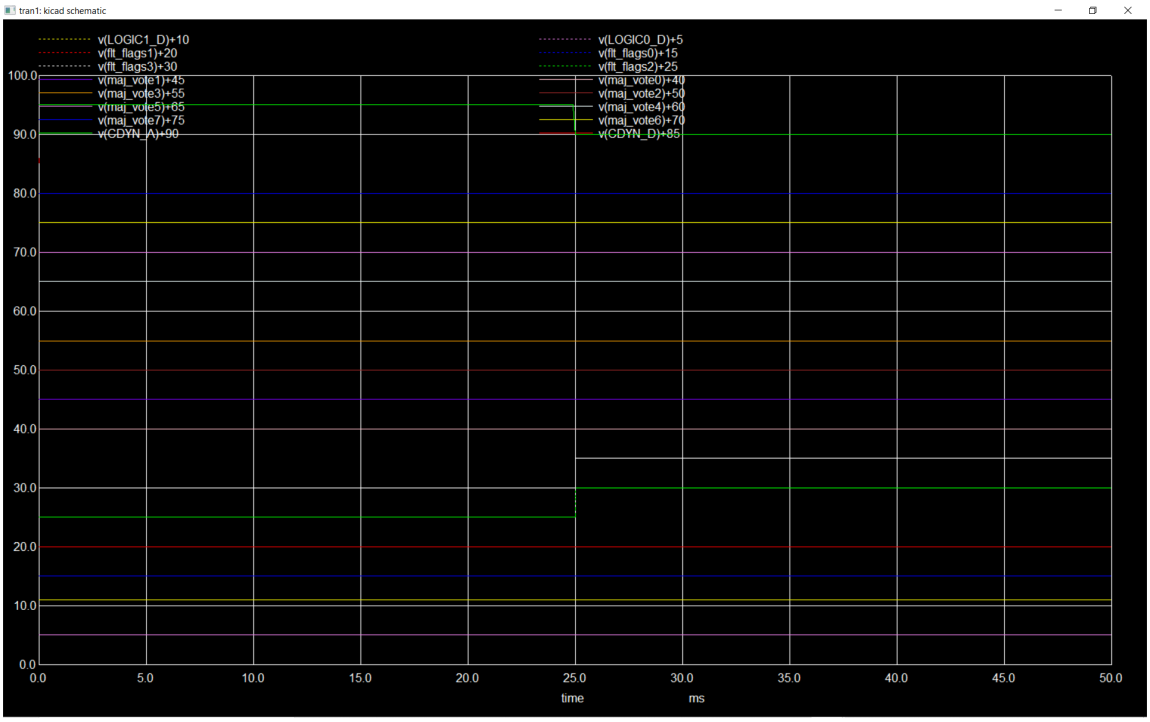


Figure 72: Stacked Ngspice view of the TMR transient. The voted output traces are perfectly flat across the entire 50 ms window — the channel failure at 25 ms is completely masked — while the channel-C and any-fault flags step high at exactly that instant.

Numerical analysis of the exported data gives an unusually clean confirmation. The four upper bits `maj_vote7`–`maj_vote4` sit at 5 V and the four lower bits `maj_vote3`–`maj_vote0` at 0 V, i.e. `0xF0`, and every one of those eight traces registers **exactly zero transitions** over the whole run. The total failure of channel C at 25 ms produced *no change whatsoever* on the output.

Simultaneously, `flt_flags[2]` and `flt_flags[3]` both rise at **25.000 ms**, and `flt_flags[0]` and `flt_flags[1]` remain at 0 V for the entire simulation. The voter therefore masked the fault **and** correctly named the channel responsible, which is the complete TMR contract.

11.8 Conclusion

The TMR Majority Voter adds fail-operational redundancy to the eSim library, complementing the existing fail-stop detection blocks. Bitwise voting was verified exhaustively over all 256 agreement values, across eight directed fault-injection cases and a 100-vector random regression, and the mixed-signal run produced the clearest possible demonstration of the concept: a total channel failure mid-simulation, zero transitions on the voted output, and an immediate, correctly attributed diagnostic flag.

12 Mixed-Signal Co-Simulation of the OpenPOWER Microwatt Soft-Processor

12.1 Introduction & Objective

Beyond the design of individual combinational IP blocks, a capstone objective of this fellowship was to demonstrate **true digital–analog co-simulation**: bridging a complete, general-purpose soft-processor written in VHDL with the continuous-time Ngspice analog solver inside eSim. The processor selected was the **OpenPOWER Microwatt**, an open-source 64-bit POWER ISA core.

The goal was to prove *cycle-accurate determinism* end-to-end: author firmware in C, cross-compile it, pre-load the machine code into the processor's simulated Block RAM (BRAM), boot the CPU using an **analog** clock generated by Ngspice, and observe the processor actively driving a real analog transmission line through its hardware UART. Success is defined not by a print statement, but by decoding the raw analog voltage waveform at the bit level and recovering the exact ASCII byte the firmware intended to transmit.

12.2 System Architecture

The co-simulation is a federation of three independent engines exchanging state over a TCP socket. The Microwatt core is elaborated by GHDL into a native executable that doubles as a socket **server**; Ngspice, driven by the eSim schematic, connects to it as a **client** through an XSPICE code model. At every synchronization point the analog node voltages and the digital `std_logic` values are translated across the boundary.

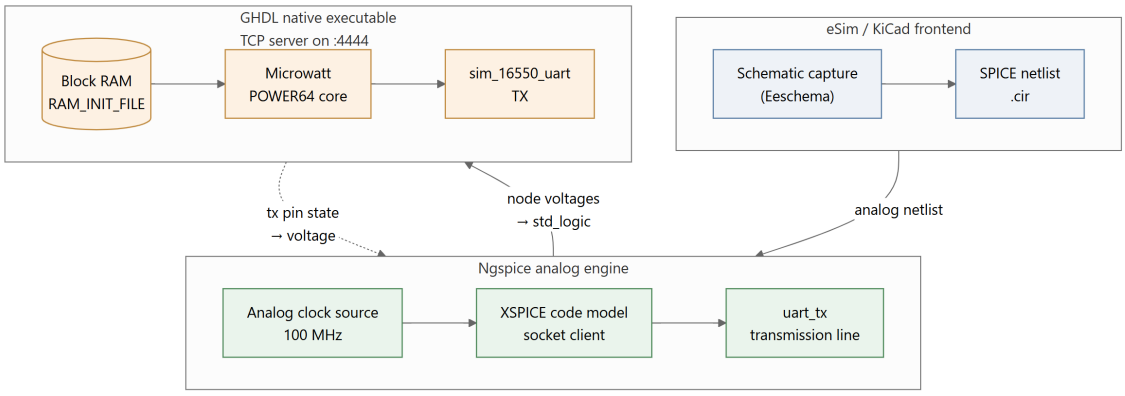


Figure 73: Top-level co-simulation architecture. The eSim/KiCad schematic emits a SPICE netlist into Ngspice, whose XSPICE code model bridges analog node voltages to the GHDL native executable over TCP port 4444. Inside the digital domain, the Microwatt POWER core executes firmware pre-loaded in BRAM and drives the 16550 UART transmit pin, whose logic state is returned to Ngspice as the analog `uart_tx` node.

12.2.1 The GHDL Socket Bridge

GHDL elaborates the VHDL testbench `microwatt_cosim_tb` together with a small C runtime (`ghdlserver.c`, `ghdlserver.o`) into a single binary. This binary opens a listening TCP socket on port `4444`. Each transaction receives the analog input voltages sampled

by Ngspice, drives them into the VHDL top level as `std_logic` stimuli, advances the Microwatt simulation, and returns the resulting output logic states to be reconverted into analog voltages.

12.2.2 The XSPICE Code Model in Ngspice

On the analog side, a custom XSPICE code model instantiated in the netlist acts as the socket client. It performs the analog→digital and digital→analog mapping identical in spirit to eSim's standard `adc_bridge`/`dac_bridge` components, but routed across the socket to the external GHDL process rather than to an in-process Verilator model. This is the mechanism that lets an entire POWER CPU behave as a single mixed-signal component on the schematic.

12.2.3 Firmware Residency in Block RAM

The Microwatt core boots from an on-chip BRAM. The VHDL generic `RAM_INIT_FILE` in `microwatt_cosim.vhdl` points at the compiled firmware binary; at elaboration time GHDL reads this file and pre-loads it into the BRAM contents so the CPU has valid instructions at $t = 0\,\text{ns}$.

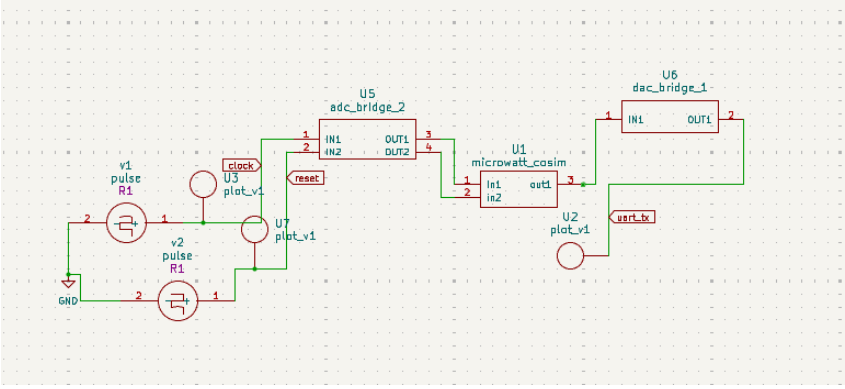


Figure 74: The eSim/KiCad schematic symbol representing the Microwatt co-simulation block, exposing the analog `clock`, `reset`, and `uart_tx` pins used to interface the digital soft-processor to the Ngspice matrix.

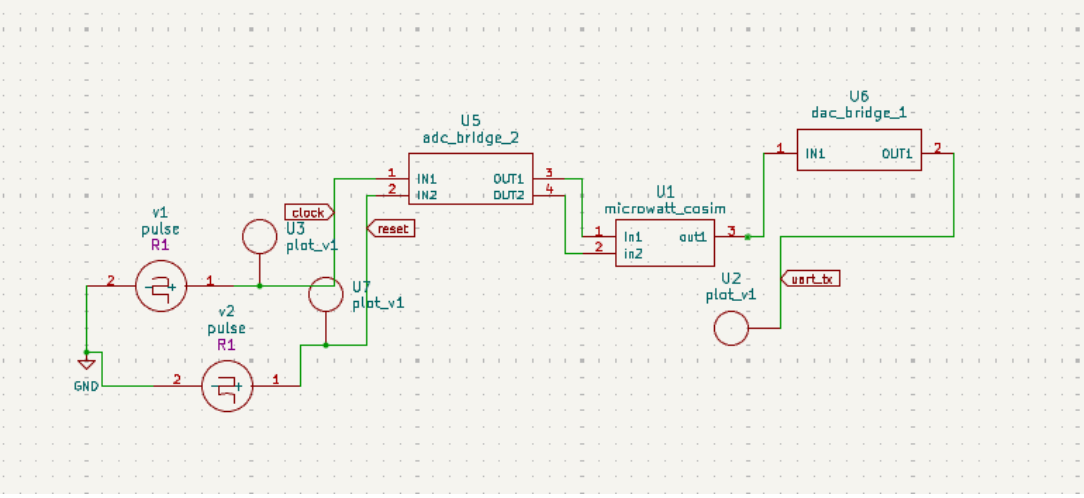


Figure 75: The eSim test schematic driving the Microwatt co-simulation. An analog clock source and reset stimulus feed the processor, while the `uart_tx` node is routed to a transmission line and probed for transient capture.

12.3 Co-Simulation Data-Flow

The two solvers advance in lockstep. Ngspice owns simulation time; at each synchronization step it hands the current analog state to the GHDL server, which evaluates the digital logic and returns the updated pin states. This closed loop is what enforces cycle-accurate determinism between the software running on the CPU and the analog physics on the wire.

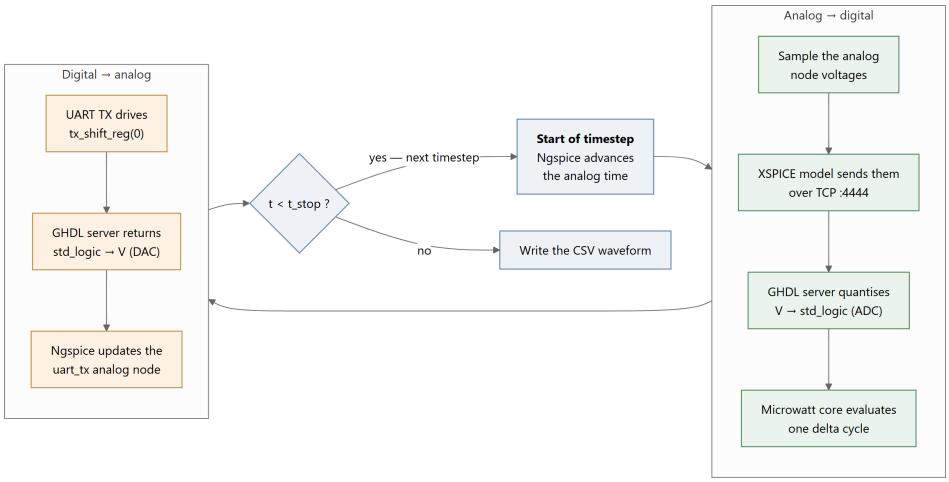


Figure 76: Per-timestep synchronization loop between the Ngspice analog solver and the GHDL digital server. Voltages cross the socket, are quantized to logic, drive one evaluation of the Microwatt core, and the resulting UART pin state is mapped back to an analog voltage before Ngspice advances to the next timestep.

12.4 Firmware Compilation & BRAM Initialisation Pipeline

Switching the behaviour of the processor means changing the machine code baked into the BRAM. Because `RAM_INIT_FILE` is resolved at elaboration, a firmware change is not a

runtime swap—it requires re-elaborating the VHDL so the new binary is embedded into a freshly built server executable.

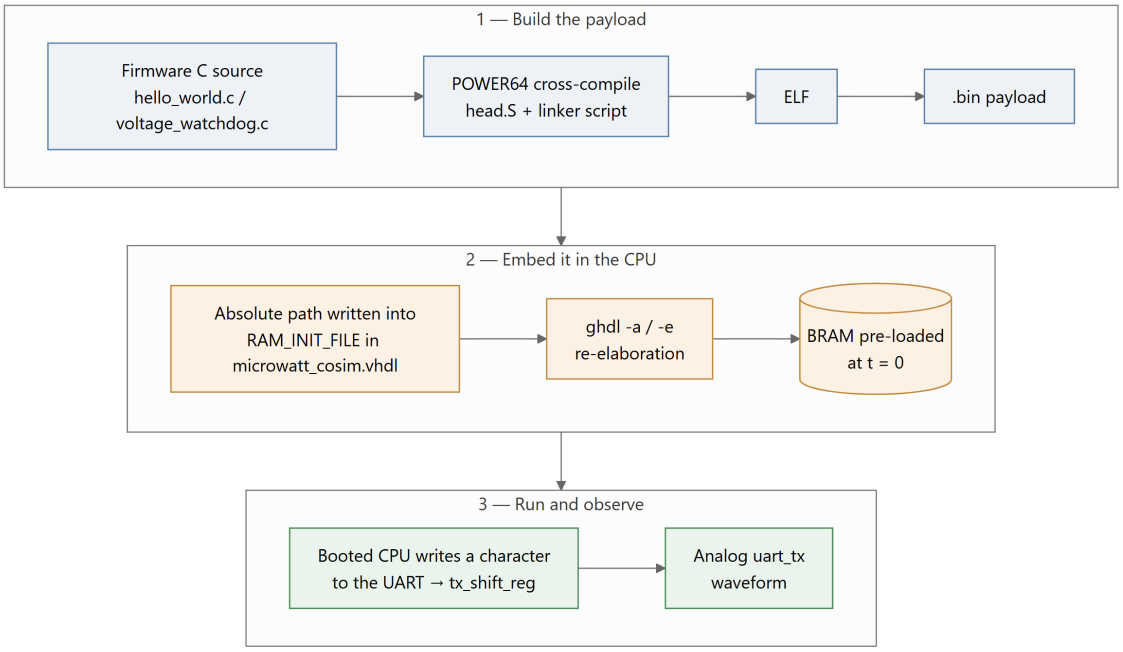


Figure 77: Firmware-to-waveform pipeline. C source is cross-compiled for the POWER ISA into a `.bin` payload, whose absolute path is written into `RAM_INIT_FILE`. GHDL re-elaboration embeds the payload into BRAM; the booted CPU then writes characters to the UART, producing the analog `uart_tx` waveform.

12.5 Engineering Challenges & Resolutions

Bringing a full soft-processor into an analog SPICE matrix surfaced a series of integration failures at the digital–analog seam. Each is documented below with its root cause and the verified resolution. Table 33 summarizes the campaign.

Failure Mode	Root Cause	Resolution
Instant server crash at $t = 0$	UART pin in uninitialized `'U'` state; no analog map	Force `tx_shift_reg` idle-high at declaration
Impractical simulation runtime	Full boot exceeds the transient window	Emit target character in the first instructions of `main()`
Firmware swap has no effect	`RAM_INIT_FILE` is a hardcoded elaboration-time path	Path-rewrite scripts + full GHDL re-elaboration
Linker failure on rebuild	GHDL defaulted to the `mcode` backend	Enforce `export GHDL_BACKEND=gcc`
Subsequent runs fail to connect	Zombie server holding TCP port 4444	`pkill` sanitization before each run

Table 33: Summary of co-simulation integration failures and their resolutions.

12.5.1 A. The Uninitialized `'U'`-State Crash

When Ngspice began the transient at $t = 0\,\text{ns}$, the VHDL UART transmit register sat in the standard IEEE-1164 uninitialized state `'U'`. The XSPICE bridge has no analog voltage to which `'U'` maps, so the server aborted before the first clock edge. The fix forces the transmit shift register to a defined idle-high pattern at signal declaration, guaranteeing a stable logic-1 (interpreted as 5 V idle) at time zero. This is directly observable in the captured waveforms, which sit cleanly at 5 V before the first start bit.

Listing 22: Idle-high initialization of the UART transmit register (sim_16550_uart.vhdl)

```
    signal tx_shift_reg : std_ulogic_vector(9 downto 0) := (others => '1');
    ...
    stx_pad_o <= tx_shift_reg(0);   -- driven pin idles at logic 1 -> 5 V

```

12.5.2 B. Emitting the Payload Within the Transient Window

Co-simulating a 100 MHz digital clock alongside a transmission-line transient is expensive; simulating even a couple hundred microseconds of real time consumes minutes of wall-clock time. Waiting for a full firmware boot banner to scroll out was therefore infeasible. The firmware was customized to write the target character directly to the UART hardware in the very first instructions of `main()`, so a decodable byte appears inside the transient window rather than after a lengthy console initialization. That the firmware was genuinely customized is confirmed by the compiled binary itself: the voltage watchdog payload embeds the string `"SYSTEM VOLTAGE FAIL!"` in place of the stock Microwatt boot banner, and the captured waveform delivers its first framed byte at approximately 4.7 μs.

12.5.3 C. Hardcoded BRAM Path & Demo Switching

Because `RAM_INIT_FILE` is a fixed absolute path evaluated at elaboration, simply selecting a different `.bin` in the eSim GUI does nothing. Switching demonstrations required rewriting the path and completely rebuilding the CPU architecture with the new firmware baked in.

Listing 23: Hardcoded firmware path in microwatt_cosim.vhdl

```
    RAM_INIT_FILE => "/home/work/Desktop/Microwatt_Video_Files/Microwatt_Source_Code/
     ↪ esim_voltage_watchdog/voltage_watchdog.bin",

```

Listing 24: Demonstration-switching script (path rewrite via sed)

```
    sed -i 's|DEMO_DIR=.*|DEMO_DIR="/home/work/Desktop/Microwatt_Video_Files/Microwatt_Source_Code/esim"|
     ↪ ' \
       /home/work/nghdl-simulator/src/xspice/icm/ghdl/microwatt_cosim/DUTghdl/start_server.sh

```

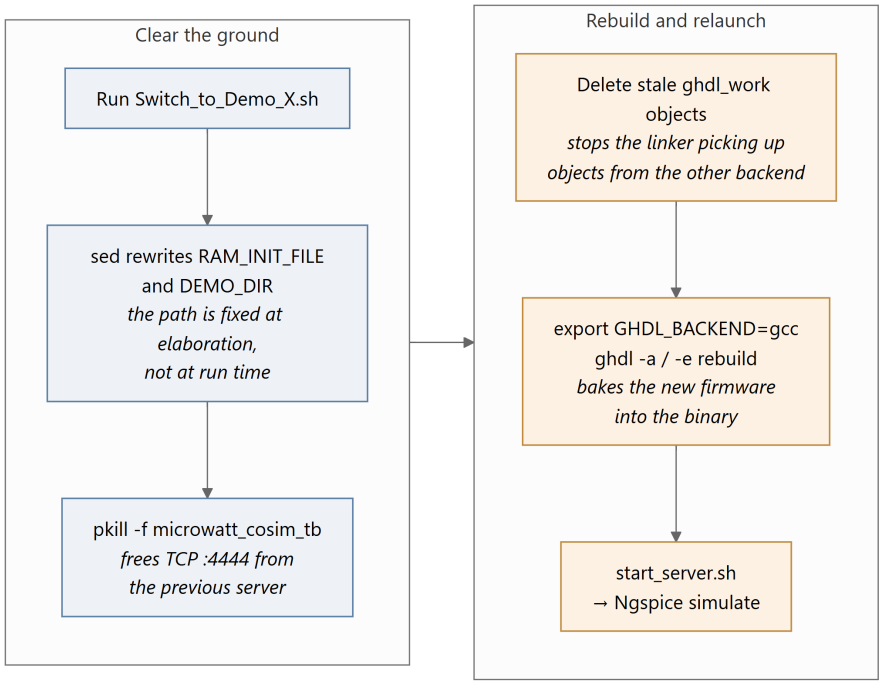


Figure 78: Demonstration-switching sequence: rewrite the firmware path, kill any stale server holding the socket, purge old GHDL work objects, and re-elaborate under the GCC backend before re-launching the simulation.

12.5.4 D. GHDL Backend Mismatch (Linker Failure)

Rebuilds intermittently failed with an obscure `-Wl` linker error. The cause was GHDL defaulting to its `mcode` backend, which cannot link the external C wrappers (`ghdlserver.o` and the console/VHPI helper objects) that the socket bridge requires. Explicitly forcing the GCC backend produced a clean link against the server object.

Listing 25: Enforcing the GCC backend during elaboration (start_server.sh)

```
export GHDL_BACKEND=gcc
ghdl -e --std=08 --workdir="$DEMO_DIR/ghdl_work" \
     -Wl,ghdlserver.o \
     -Wl,sim_console_c.o -Wl,sim_vhpi_c.o -Wl,sim_bram_helpers_c.o \
     microwatt_cosim_tb

```

12.5.5 E. Zombie Processes Blocking the Socket

An aborted or failed run could leave the background `microwatt_cosim_tb` server alive, holding TCP port 4444 open. Every subsequent run then failed to bind. The operational remedy was to forcibly clear the port before initializing a new run.

Listing 26: Socket sanitization before a new run

```
pkill -f microwatt_cosim_tb

```

12.6 UART Protocol & Baud-Rate Physics

The processor transmits over a standard asynchronous serial link at 115200 baud, 8 data bits, no parity, one stop bit (8N1). The nominal duration of a single bit is therefore

$$T_{\text{bit}} = \frac{1}{115200} = 8.6806\,\mu\text{s}.$$

Each transmitted byte is framed as a ten-symbol sequence: one start bit (logic 0), eight data bits sent **least-significant-bit first**, and one stop bit (logic 1). The line idles at logic 1 (5 V) between frames.

Symbol #	Role	Idle/Level
0	Start bit	Logic 0 (0 V)
1–8	Data bits D0–D7 (LSB first)	Data-dependent
9	Stop bit	Logic 1 (5 V)

Table 34: 8N1 UART frame structure at 115200 baud.

12.7 Verification: Bit-Level Waveform Decode

To prove the co-simulation was mathematically exact rather than merely plausible, the raw analog `uart_tx` column was exported from the Ngspice transient to CSV ($\tilde{}4 \times 10^5$ samples over a 200 μs window) and decoded independently. The decode procedure: threshold each sample at 2.5 V, locate the first idle-high→low transition as the start bit, then sample the line at the centre of each of the following ten bit periods ($t_{\text{start}} + (k + 0.5)\,T_{\text{bit}}$).

The observed inter-sample spacing was 8.681 μs, matching the theoretical 8.6806 μs bit period to within the sampling resolution, and both captures confirmed a clean idle level of 5.00 V with the start bit at 0.00 V.

12.7.1 Full Boot & Transmission Overview

Before isolating individual bytes, the complete 200 μs transient window is inspected as a whole to establish context. Figure 79 stacks the three interface signals of the co-simulation: the analog `clock` generated by Ngspice (running at 100 MHz), the `reset` stimulus that releases the core, and the resulting `uart_tx` output. Unlike the single-byte decode figures that follow, this overview shows the **entire** serial burst—the full train of start, data, and stop transitions the processor emits across the window. It is from this composite capture that the individual bytes are windowed out and decoded bit by bit. The plot also makes the cost noted in Section 5.5.B concrete: every one of these microseconds is co-simulated against a 100 MHz digital clock, which is precisely why the firmware was tuned to emit its payload early in the window rather than after a full boot banner.

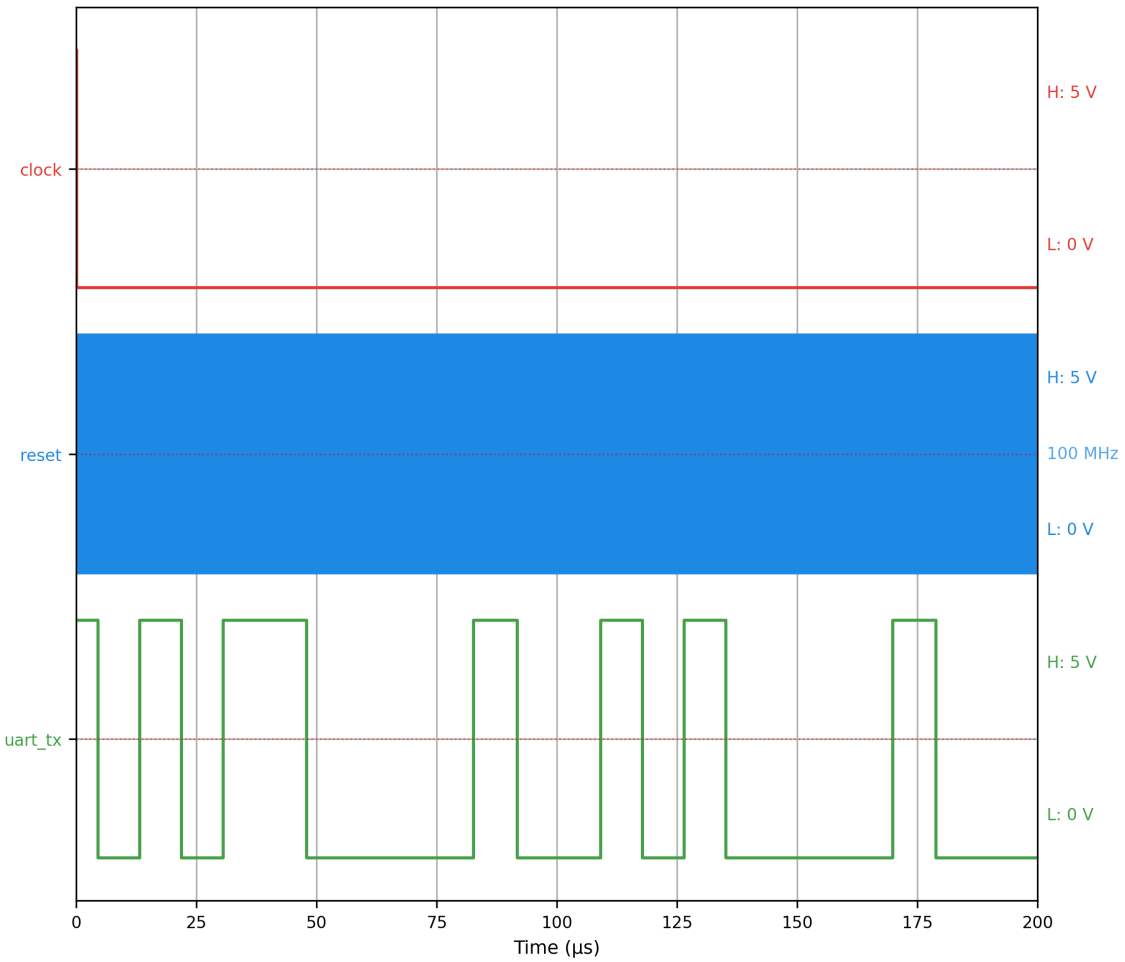


Figure 79: Full 200 μs co-simulation overview. Top: the 100 MHz analog `clock` driving the Microwatt core. Middle: the `reset` stimulus. Bottom: the complete `uart_tx` serial burst, with logic levels at 5 V (high) and 0 V (low). The individual 'H' and 'V' bytes decoded below are windowed from this composite waveform.

12.7.2 Demonstration 1 — The 'H' Payload

The first falling edge occurred at 4.673 μs. Sampling the ten bit centres yields the decode in Table 35.

Bit	Role	Sample t (μs)	Voltage (V)	Logic
0	START	9.013	0.00	0
1	D0	17.694	0.00	0
2	D1	26.374	0.00	0
3	D2	35.055	0.00	0
4	D3	43.736	5.00	1
5	D4	52.416	0.00	0
6	D5	61.097	0.00	0
7	D6	69.777	5.00	1
8	D7	78.458	0.00	0
9	STOP	87.138	5.00	1

Table 35: Bit-level decode of Demonstration 1. Data (LSB-first) = `00010010`.

Reading the data field D0–D7 as transmitted (LSB first) gives `00010010`. Reversing to conventional MSB-first order yields `01001000`, which is exactly `0x48` — the ASCII character **'H'**.

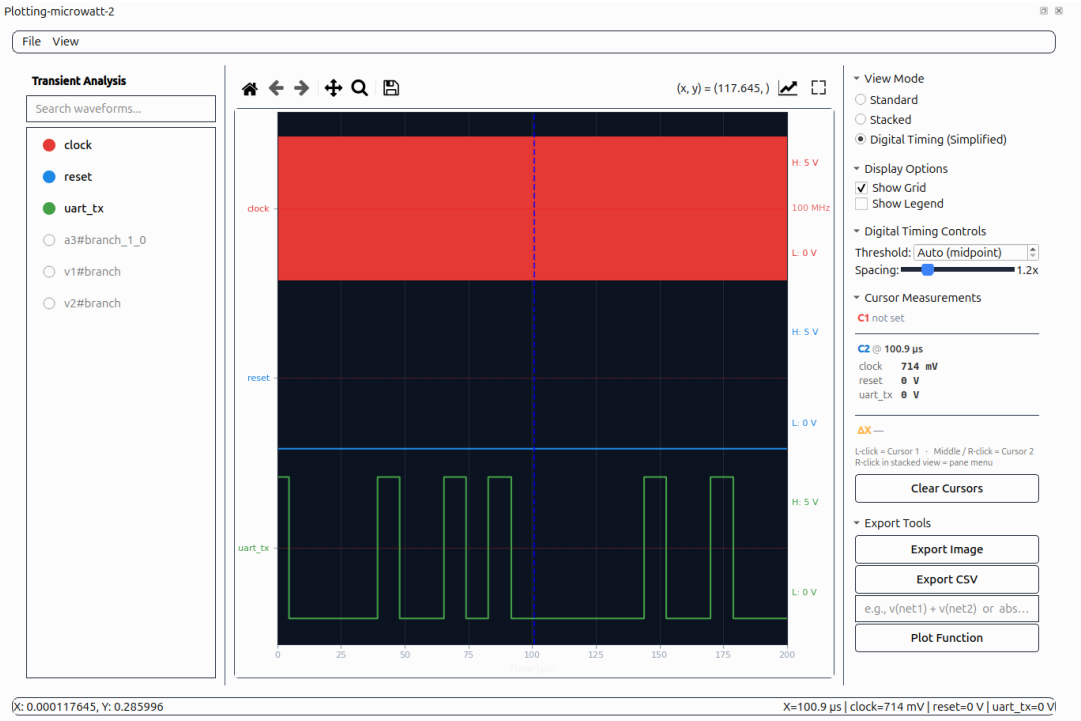


Figure 80: Ngspice transient capture of the 'H' payload on the analog `uart_tx` node, showing the 5 V idle, the start bit, and the eight LSB-first data bits that decode to `0x48`.

12.7.3 Demonstration 2 — The 'V' Payload (Voltage Watchdog)

Applying the identical procedure to the voltage-watchdog capture produces the decode in Table 36.

Bit	Role	Voltage (V)	Logic
0	START	0.00	0
1	D0	0.00	0
2	D1	5.00	1
3	D2	5.00	1
4	D3	0.00	0
5	D4	5.00	1
6	D5	0.00	0
7	D6	5.00	1
8	D7	0.00	0
9	STOP	5.00	1

Table 36: Bit-level decode of Demonstration 2. Data (LSB-first) = `01101010`.

The LSB-first data field `01101010` reverses to `01010110`, which is exactly `0x56` — the ASCII character **'V'**, the intended output of the voltage watchdog firmware.

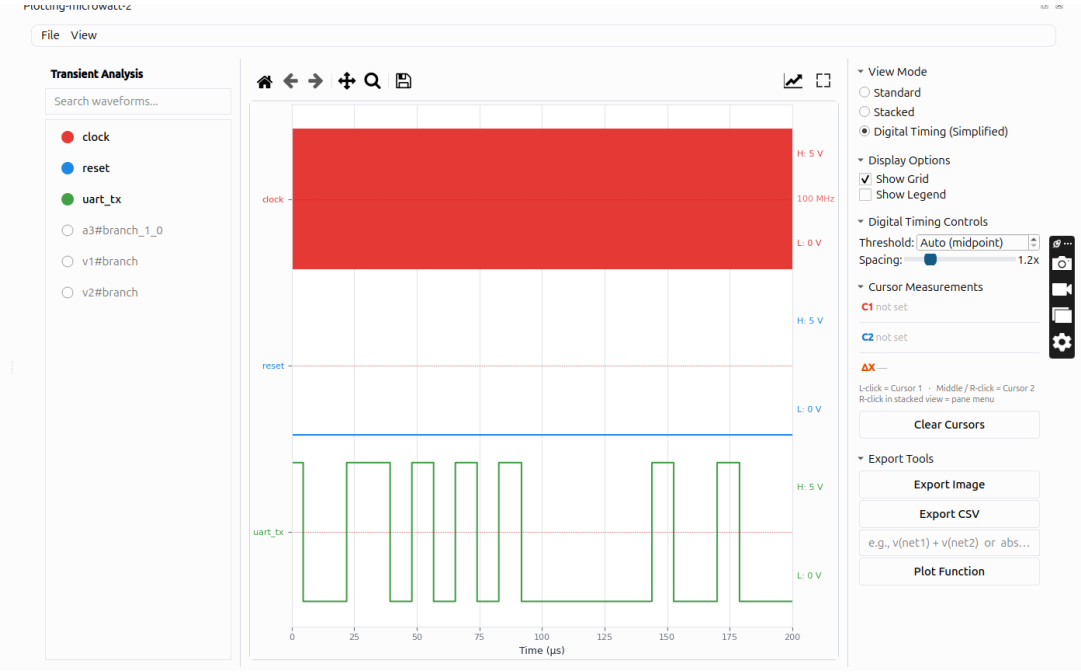


Figure 81: Ngspice transient capture of the 'V' payload. The framed byte decodes to `0x56`, confirming the customized voltage-watchdog firmware drove the analog line exactly as designed.

12.8 Improvements & Future Prospects

While the co-simulation successfully proves cycle-accurate determinism, it currently operates as a hand-assembled proof of concept. Two directions dominate the path from demonstration to a production-grade eSim feature.

12.8.1 Constraint: Simulation Performance

The single largest limitation today is throughput. The architecture couples a 100 MHz digital clock to a continuous-time analog transient, exchanging state across a TCP socket at every synchronization point. Each timestep therefore pays for a socket round-trip, a full delta-cycle evaluation of the entire Microwatt core, and an analog matrix solve—and the fast clock forces a very fine analog timestep to resolve it. The practical consequence is that only **very short simulation windows** (on the order of a couple hundred microseconds) are feasible, and even those consume minutes of wall-clock time. Booting a full firmware image to completion before observing output is impractical, which is exactly why the payloads were tuned to emit early.

Several avenues can lift this ceiling:

- **Reduced synchronization frequency:** generate the high-speed clock inside the VHDL domain and exchange only the slow-changing analog nodes across the socket, relaxing the analog timestep dramatically.
- **Event-driven co-simulation:** synchronize on signal events rather than a fixed timestep, so idle intervals cost nothing.
- **Faster IPC:** replace the per-step TCP socket with shared-memory or batched transactions to amortize the transport overhead.
- **Boot fast-forwarding:** checkpoint a pre-booted BRAM/register state so the simulation can begin at the point of interest instead of re-executing boot each run.
- **Configurable clock rate:** expose the clock frequency so users can trade physical fidelity for logical progress during development.

12.8.2 Future Prospect: A Fully Automated GUI Environment

At present the firmware-to-simulation pipeline is entirely manual: the BRAM path is hardcoded in the VHDL, demonstrations are switched by shell scripts that rewrite paths with `sed`, the design must be re-elaborated with GHDL by hand under the correct backend, zombie servers on port 4444 must be cleared manually, and the socket server must be launched before simulating. This workflow demands deep tool-specific knowledge and is error-prone.

The principal future goal is to collapse this five-step manual pipeline into a **single-click eSim GUI workflow** that programs the OpenPOWER core automatically. The envisioned environment would:

1. accept firmware directly from the user—either C source or a pre-built `.elf`/`.bin` payload;
2. invoke the POWER cross-compiler transparently to produce the binary;
3. regenerate the co-simulation VHDL, injecting the new `RAM_INIT_FILE` path automatically;
4. re-elaborate through GHDL with the GCC backend enforced, with no user interaction;

5. manage the socket-server lifecycle—sanitize port 4444, launch, and tear down automatically around each run;

6. run the transient and **auto-decode** the `uart_tx` waveform back to ASCII, presenting the recovered characters directly in the GUI.

Delivered as a reusable eSim library component with parametric firmware selection, this would let any user drop a programmable OpenPOWER processor onto a schematic, load their own firmware, and observe real analog behaviour without touching a single script—transforming a specialist research demonstration into an accessible mixed-signal design tool.

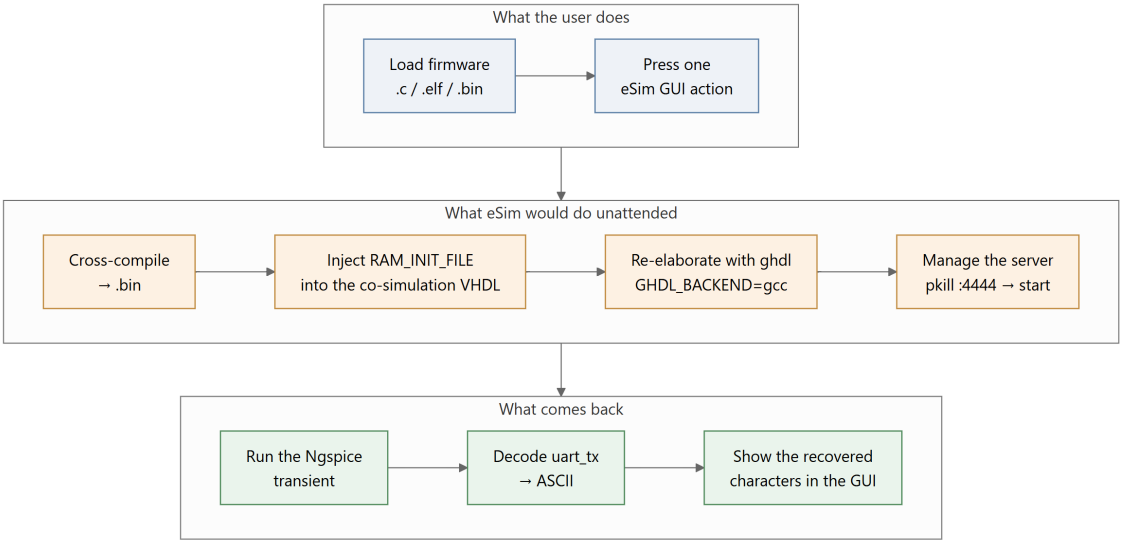


Figure 82: Proposed one-click GUI automation flow. The manual firmware-swap, path-rewrite, re-elaboration, server-management, and decode steps are absorbed behind a single eSim action that programs the Microwatt core and returns the decoded ASCII output.

12.9 Conclusion

This co-simulation closes the loop from C source code to analog physics. Firmware compiled for the OpenPOWER ISA was pre-loaded into the Microwatt core's BRAM, the processor was booted by an analog Ngspice clock, and its hardware UART drove a transmission line whose voltage was captured and decoded byte-for-byte. Both demonstrations recovered the exact ASCII characters their firmware was written to emit — `0x48` ('H') and `0x56` ('V') — with the measured bit period matching the 115200-baud theoretical value. This constitutes an independent, bit-level proof of cycle-accurate determinism between a software change in the digital domain and the resulting analog waveform, validating eSim's socket-bridged XSPICE mechanism as a viable path for co-simulating full soft-processors alongside real analog circuitry.

13 The eSim User-Interface Framework

13.1 Introduction & Need

eSim's capability had grown considerably faster than its presentation. The application already drove KiCad, Ngspice, GHDL, Verilator and Makerchip, but it did so through an interface assembled feature by feature, with each panel styling itself, colours written literally wherever they were needed, and no shared vocabulary for spacing, depth or emphasis. The practical consequences were visible: a light theme that was not consistently legible, dialogs that did not resemble the windows that opened them, and no way for a user to make the application match the rest of their desktop.

The work documented here replaces that with a single, named design system — *Aurora* — and rebuilds the main window, the dialogs, the project explorer and the editing surfaces on top of it. The aim was not decoration. An EDA tool is used for hours at a stretch, often at night, and legibility, visual hierarchy and a predictable place for every control are functional requirements rather than cosmetic ones.

Collaboration and Attribution

The user-interface framework, the in-application Verilog Verifier, and the Icarus-based co-simulation backend (Chapters 13–15) were co-architected and prototyped in close collaboration with my colleague **Mr. Varadharam R. S.** Following initial design and joint prototyping, Mr. Varadharam led the production hardening, bug fixes, and edge-case validation to bring these modules to release readiness. The technical accounts in the following chapters reflect this shared engineering effort and detail the joint architectural decisions that shaped the platform.

13.2 The Aurora Design System

The system has one rule that governs everything else: **no brand colour is written anywhere except the token table**. A single module defines the dark and light palettes — background, raised and surface levels, strokes, primary and muted text, accent and status colours — and every other consumer reads from it.

Those tokens reach the screen along two paths. Most of the interface is styled by two stylesheets, one per theme, into which the token values are substituted when the theme is applied. The remainder is painted directly in Qt, because a stylesheet cannot express it: gradient-filled text, the diagonal accent surface used on hero panels, rounded tooltips, drop shadows and the inline SVG icon set.

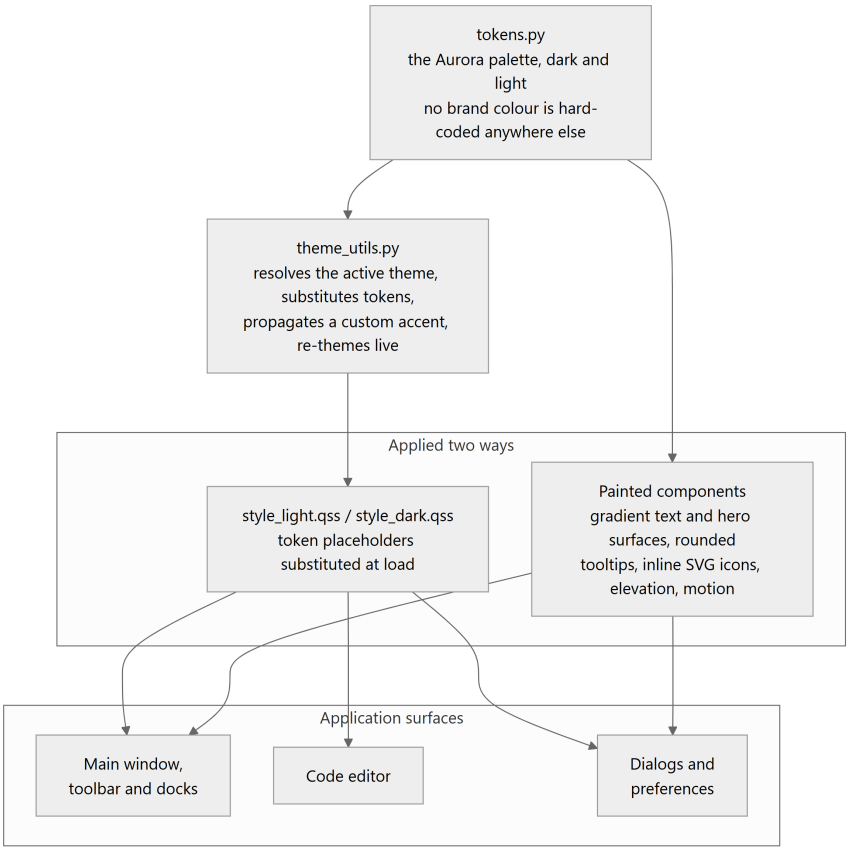


Figure 83: The Aurora design system. One token table feeds both the generated stylesheets and the natively painted components, so a colour change is made in exactly one place.

13.3 Anatomy of the Main Window

The main window is a dock host. Every panel — the project explorer, the central work area, the console and the snapshots panel — can be docked, floated, tabbed or hidden, and the arrangement survives a restart. The central area is where the existing flows appear: the schematic editor, the KiCad-to-Ngspice conversion tabs, the NgVeri and Flow Navigator surfaces, and the plotting window.

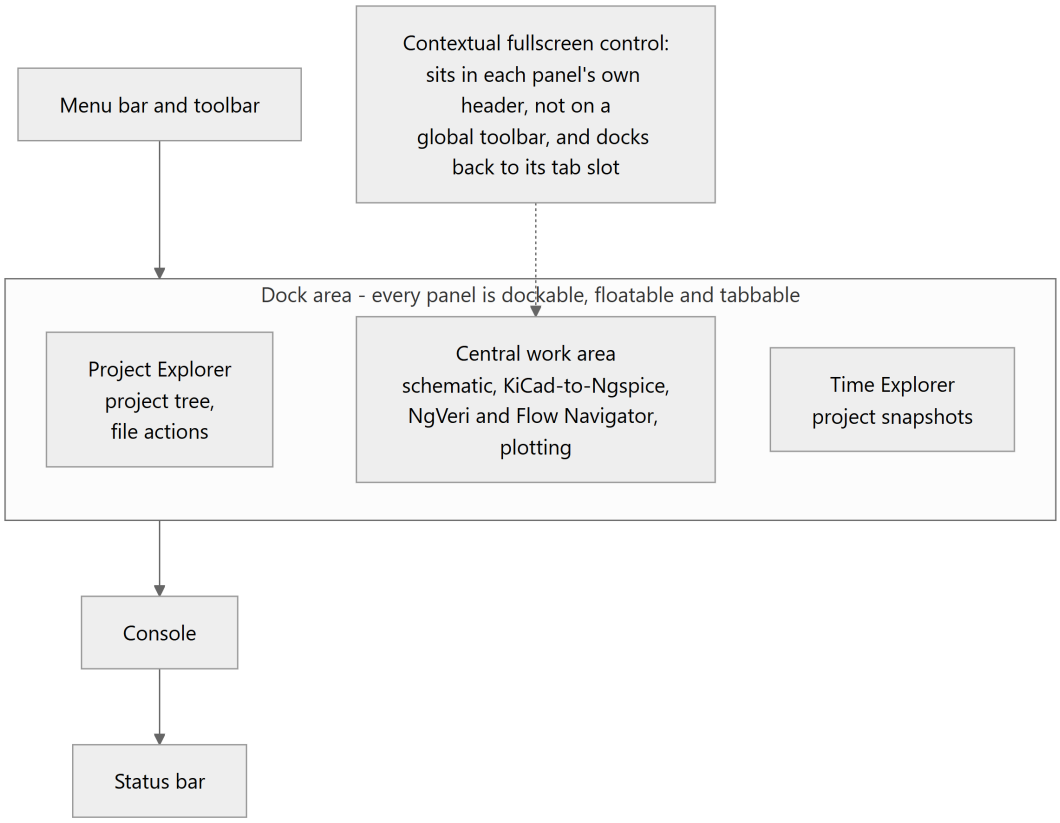


Figure 84: Main-window anatomy. The dock area hosts every panel; the fullscreen affordance deliberately lives in each panel's own header rather than on a global toolbar.

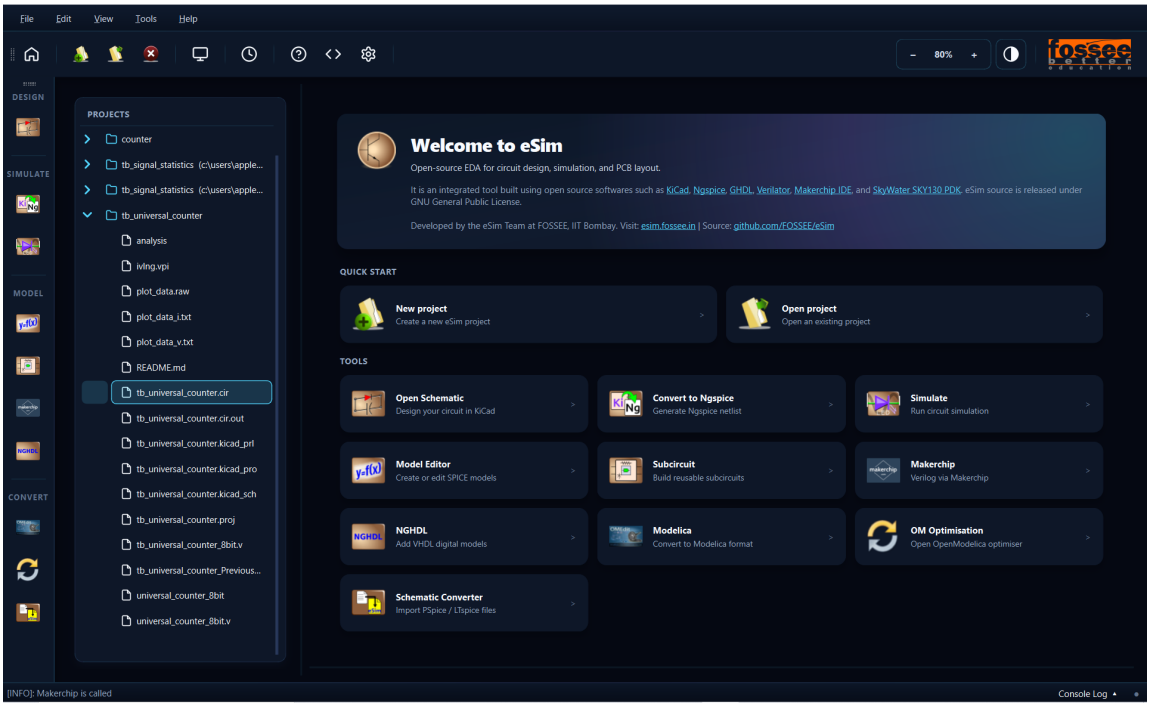


Figure 85: The rebuilt main window in the dark theme, with a project open.

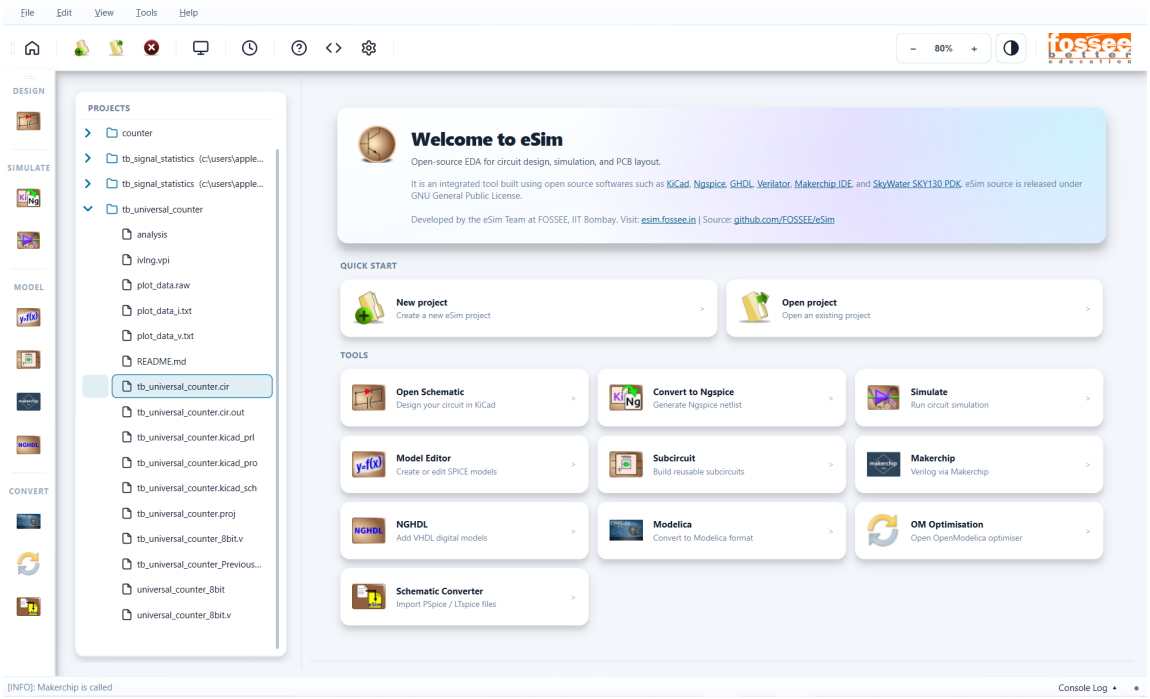


Figure 86: The same window in the light theme. Both themes are first-class: the light palette is a designed counterpart rather than an inverted dark one, and shadows in it are tinted so that they read as soft ambient occlusion on white.

13.4 Theming

The user chooses **Auto**, **Light** or **Dark**. Auto follows the operating system. Whatever is chosen is applied to the running application — every open window, dialog and painted component re-themes without a restart — and on Windows the native title bar is switched to match, so a dark session does not end in a white frame.

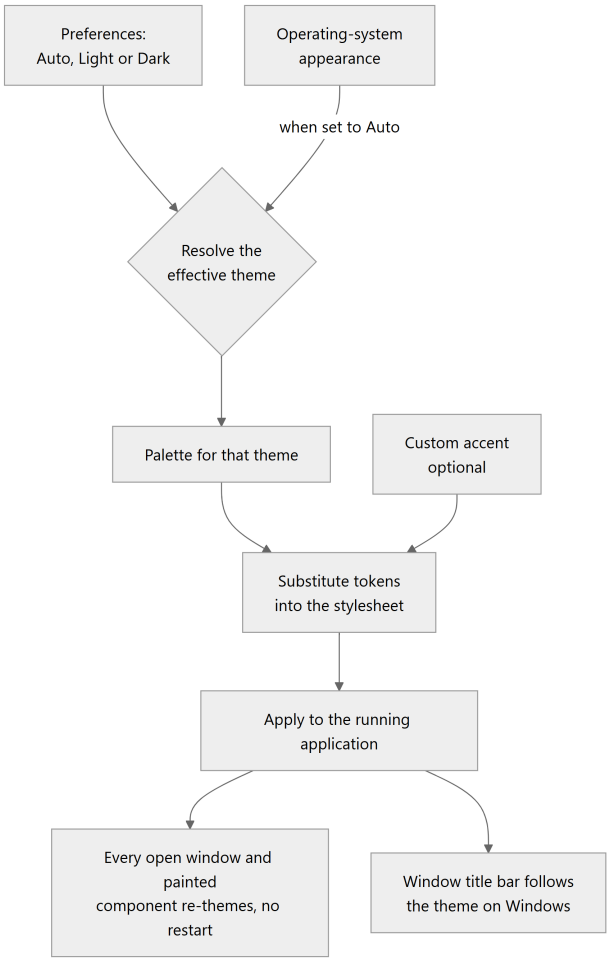


Figure 87: Theme resolution and application. The chosen mode selects a palette, the tokens are substituted into the stylesheet, and the result is pushed to the live application.

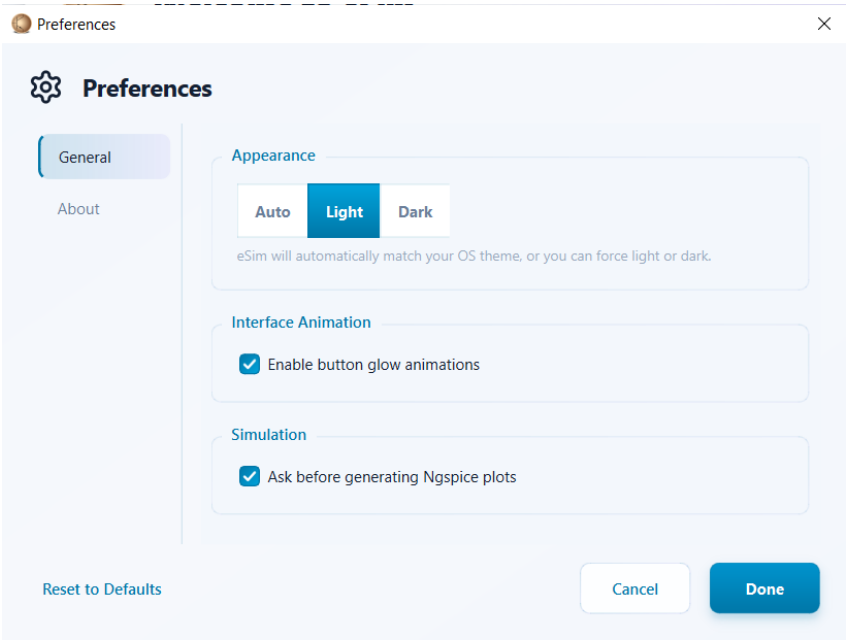


Figure 88: The Preferences dialog. It is deliberately small — display mode, interface animation, and the simulation-plot prompt — a focused settings surface rather than a customisation playground.

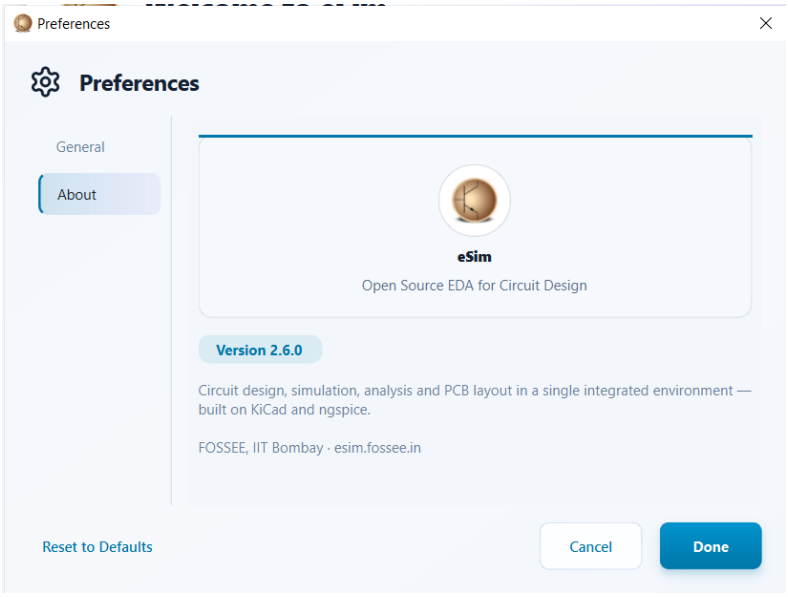


Figure 89: The About page of the same dialog, carrying the product identity, the running version and the project's attribution. It is the second and last page — the dialog has no third.

13.5 Depth, Motion and Tooltips

Three pieces of the system exist because Qt's defaults could not express what was wanted.

- **A named elevation scale** replaces per-widget shadow constants. It carries two alpha tracks, one per theme, and offers two ways to spend a shadow: on the widget

itself, or on an empty sibling placed behind it. The second exists for a specific reason — a widget carrying a graphics effect loses subpixel antialiasing on its text, so anything that permanently displays text gets the shadow on a backdrop instead and keeps its text sharp.

- **Motion is created on demand.** Hover glows are built when the pointer enters a control and torn down when it leaves, so at most one live blur exists at a time rather than one per button; the resting appearance comes from the stylesheet, which the style engine paints for free.
- **Tooltips are painted, not native.** A stylesheet border radius cannot clip the corners of Qt's tooltip window, so tooltips are replaced application-wide with a frameless, translucent, rounded card that follows the active theme.

13.6 The Integrated Code Editor

Project files previously had to be opened outside eSim. The interface now carries a multi-tab editor window that sits beside the schematic and simulation rather than behind them, with a menu bar, an inline find bar and a status bar.

It is built on QScintilla, with SPICE, Verilog and VHDL highlighting. SPICE receives a purpose-built lexer, because the stock one cannot tell a component instance from a node and miscolours stray single letters. If QScintilla is not installed the window falls back to a plain editor, so an optional dependency can never leave the project explorer unable to open a file.

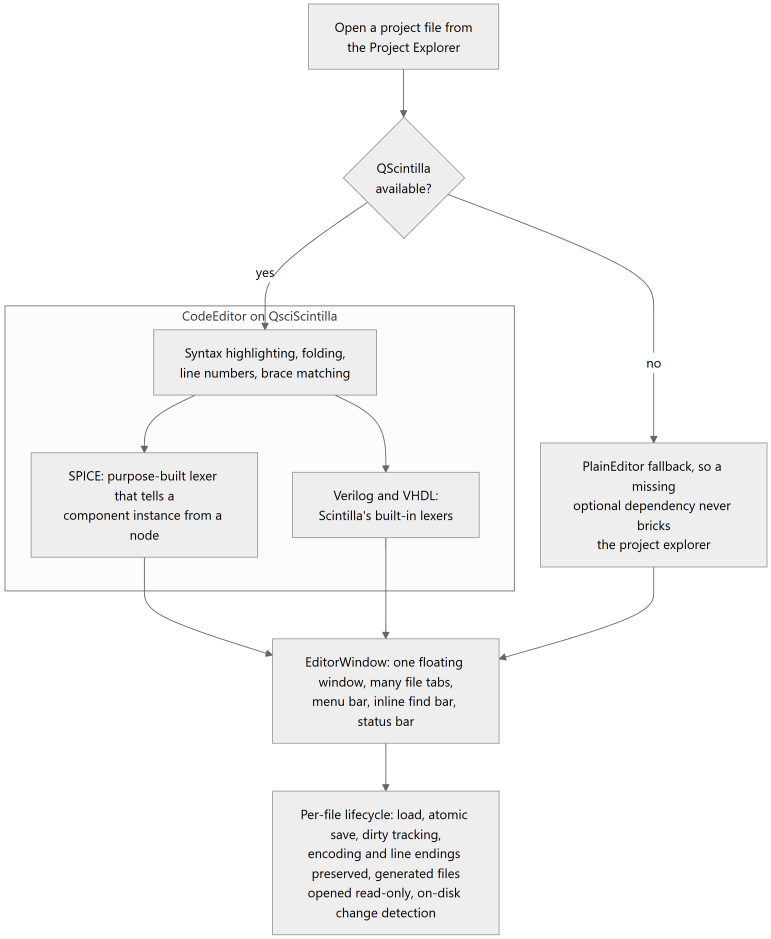


Figure 90: The editor stack. One floating window hosts many file tabs; each buffer owns its whole file lifecycle, and a missing optional dependency degrades to a plain editor rather than to an error.

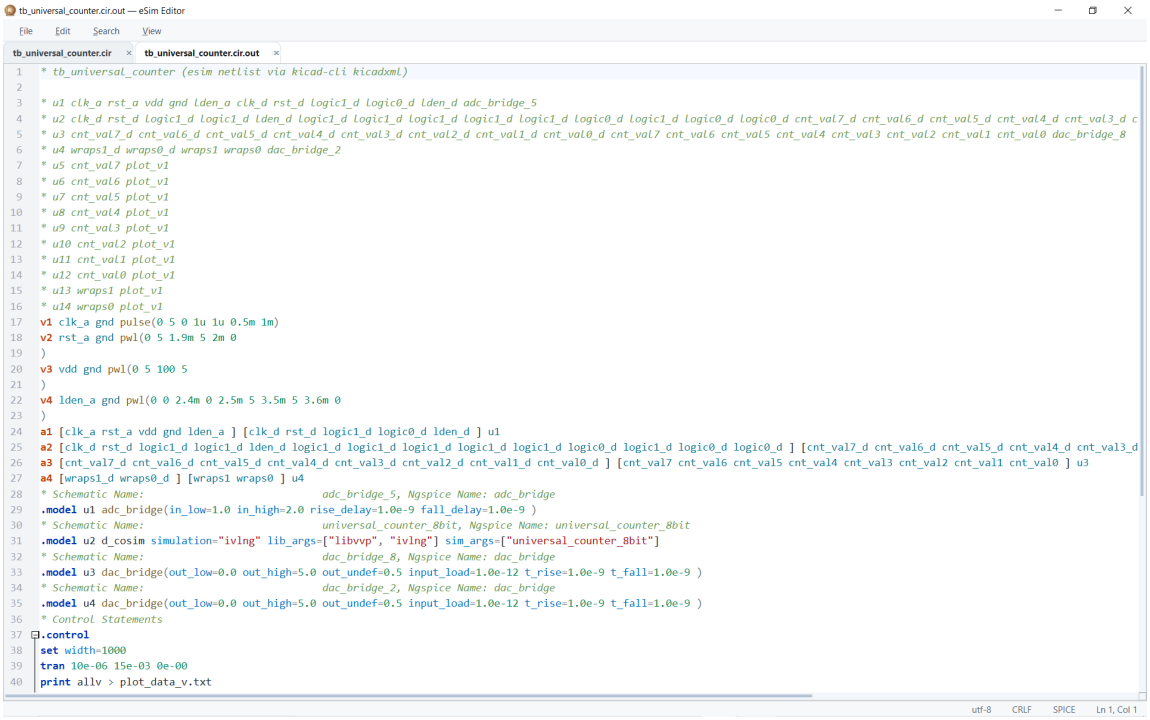


Figure 91: The editor window with project files open in tabs.

13.7 Project Snapshots

A snapshot is a point-in-time copy of a project, taken so that a user can experiment and roll back if a change breaks something. The storage model is deliberately plain: one folder per snapshot, holding a manifest and the captured bytes, named so that the folders sort chronologically. The model carries no Qt at all and is testable on its own; the Time Explorer panel is the view on top of it.

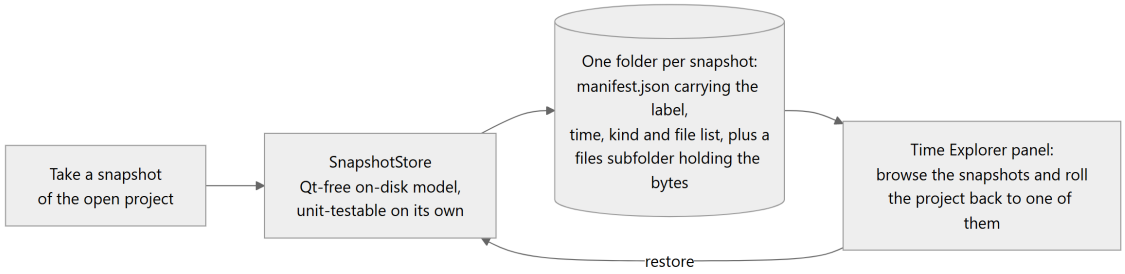


Figure 92: The snapshot model. Storage is a plain on-disk format with no database and no loose files, which keeps a snapshot readable and recoverable outside eSim.

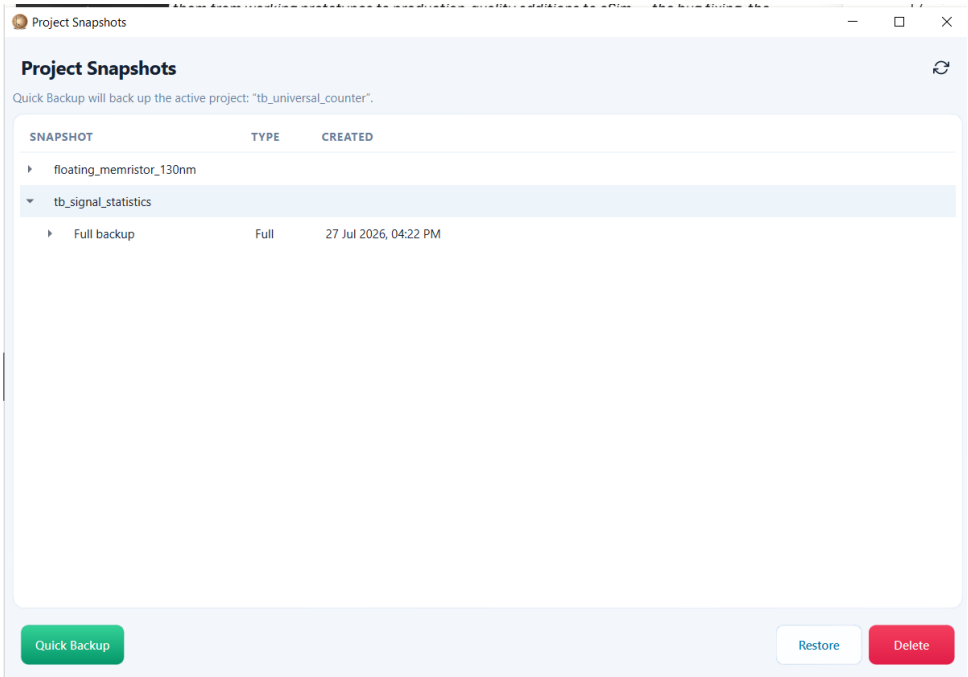


Figure 93: The Time Explorer panel listing a project's snapshots.

13.8 Contextual Fullscreen

Any docked panel can be taken fullscreen from a control in its own header. On activation the panel's content is reparented into a frameless top-level window covering the whole screen — not merely the application's work area — and the same control, `Esc` or `F11` brings it back to the tab slot it came from. Placing the affordance in the panel header rather than on a global toolbar makes it read as "fullscreen *this*, then dock it back", which is what a user wants when a waveform or a schematic needs the whole display.

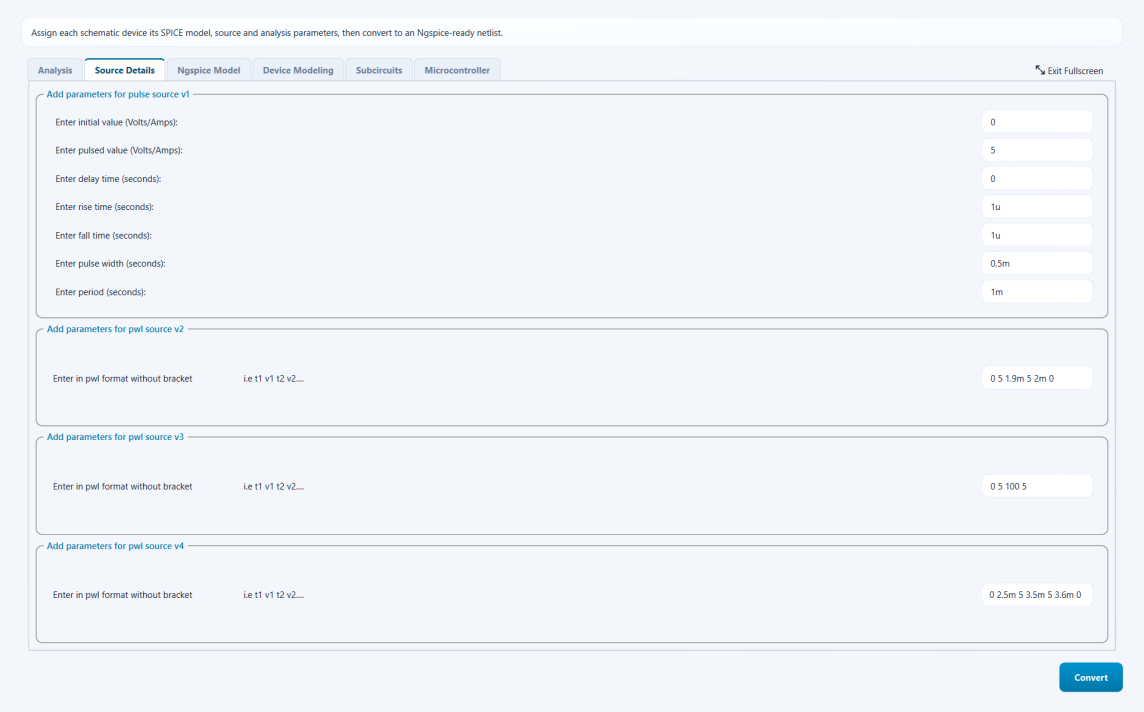


Figure 94: A docked panel taken fullscreen from its own header. The KiCad-to-Ngspice *Source Details* tab is shown filling the display, with the *Exit Fullscreen* affordance sitting in the tab-strip corner — the same place it was invoked from, rather than on a global toolbar.

13.9 The Project Explorer

The project explorer presents the open project as a tree, with the file actions available in place and the integrated editor a double-click away, so moving between a schematic, its netlist and its Verilog sources no longer means leaving the application.

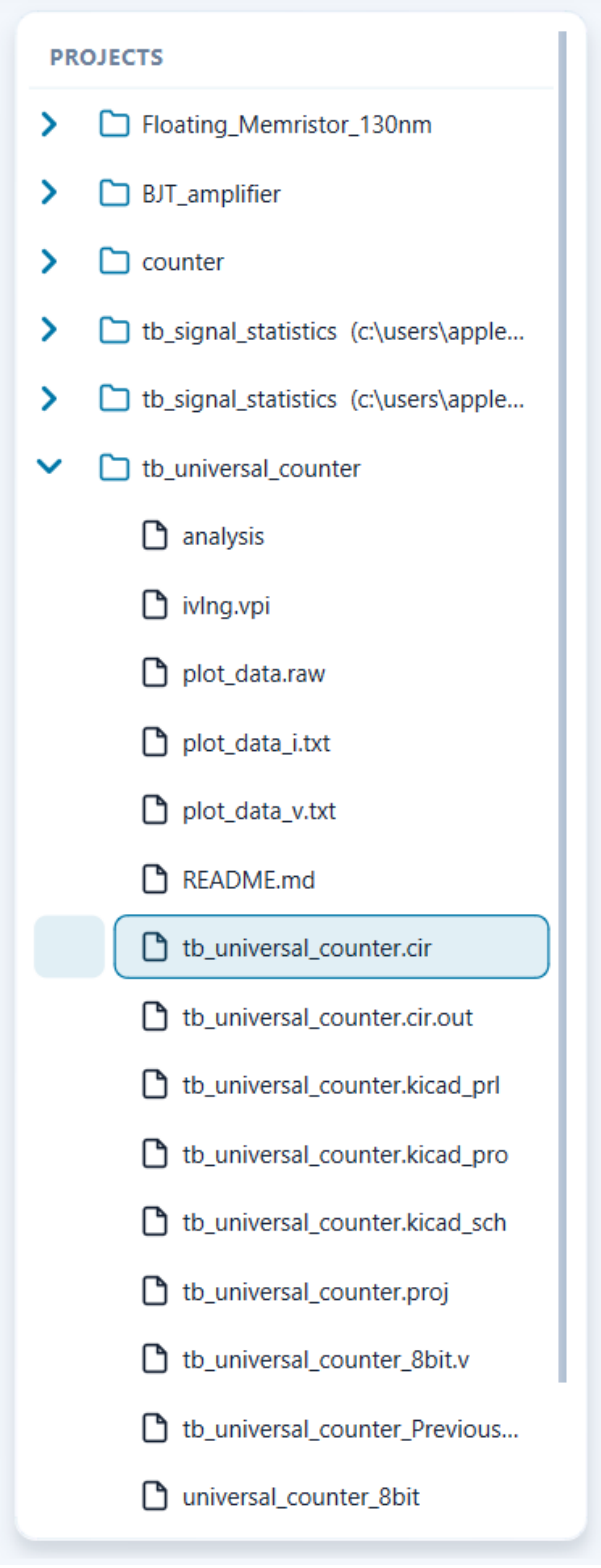


Figure 95: The project explorer with a project open.

13.10 Implications

The change is not only visual. Because colour, depth and spacing now come from one place, a new panel inherits the application's appearance instead of re-inventing it — which is precisely how the Flow Navigator described in the next chapter was able to look

native on the day it was written. A light theme that is genuinely legible widens the range of environments eSim is comfortable in, and an integrated editor and a snapshot facility remove two reasons a user previously had to leave the application mid-task.

13.11 Conclusion

The interface was rebuilt on a single design system, with both themes treated as first-class, a dockable main window whose layout is the user's to arrange, an integrated multi-language editor, project snapshots, and a contextual fullscreen affordance. The result is an application that presents its considerable existing capability coherently, and a foundation that later features can build on rather than work around.

13.12 Future Prospects & Improvements

- **Accessibility passes:** verify contrast ratios across both themes against WCAG guidance, and complete keyboard navigation for every panel.
- **Layout presets:** named dock arrangements for distinct tasks — schematic capture, mixed-signal debugging, HDL authoring — switchable in one action.
- **Editor depth:** project-wide search, and diffing a file against a snapshot directly in the editor.
- **Snapshot scope:** automatic snapshots at significant events, such as immediately before a netlist conversion overwrites a generated file.

14 The In-Application Verilog Verifier

14.1 Introduction & Need

Every digital IP documented in the preceding chapters passed through a two-stage verification methodology: a pure-digital "Phase 1" in an external simulator, followed by a mixed-signal "Phase 2" inside eSim. That workflow exposed a structural gap in the tool: eSim historically offered **no way to compile, simulate, or debug a Verilog design inside the application itself**. A designer had to leave eSim, run a third-party simulator, confirm the logic behaved, and only then return to perform the KiCad-to-Ngspice conversion. Design authoring and functional verification lived in two disconnected tools, and every logic fix demanded a full context switch.

The **Verilog Verifier** closes this gap. It is a complete, in-application Verilog Simulator IDE that compiles and simulates HDL with the Icarus Verilog engine and renders the resulting digital waveform natively — allowing a design to be proven *before* it is ever converted into a mixed-signal block. In effect, it collapses the previously external Phase 1 step into the same environment that ultimately performs the mixed-signal conversion.

Collaboration and Attribution

The user-interface framework, the in-application Verilog Verifier, and the Icarus-based co-simulation backend (Chapters 13–15) were co-architected and prototyped in close collaboration with my colleague **Mr. Varadharam R. S.** Following initial design and joint prototyping, Mr. Varadharam led the production hardening, bug fixes, and edge-case validation to bring these modules to release readiness. The technical accounts in the following chapters reflect this shared engineering effort and detail the joint architectural decisions that shaped the platform.

14.2 The Unified Verilog Flow Navigator (Host Environment)

The Verifier is surfaced inside a redesigned, workflow-shaped panel that also hosts the co-simulation feature described in the next chapter. A segmented language toggle selects between an independent Verilog path and a VHDL path, so the two toolchains never collide. The Verilog path is expressed as three freely navigable stages:

Author → Verify → Convert

Author is where modules are written or loaded, *Verify* is the Verilog Verifier, and *Convert* is the model-generation panel. The stages are a guide, not a wizard: any stage is reachable in any order, with no gating.

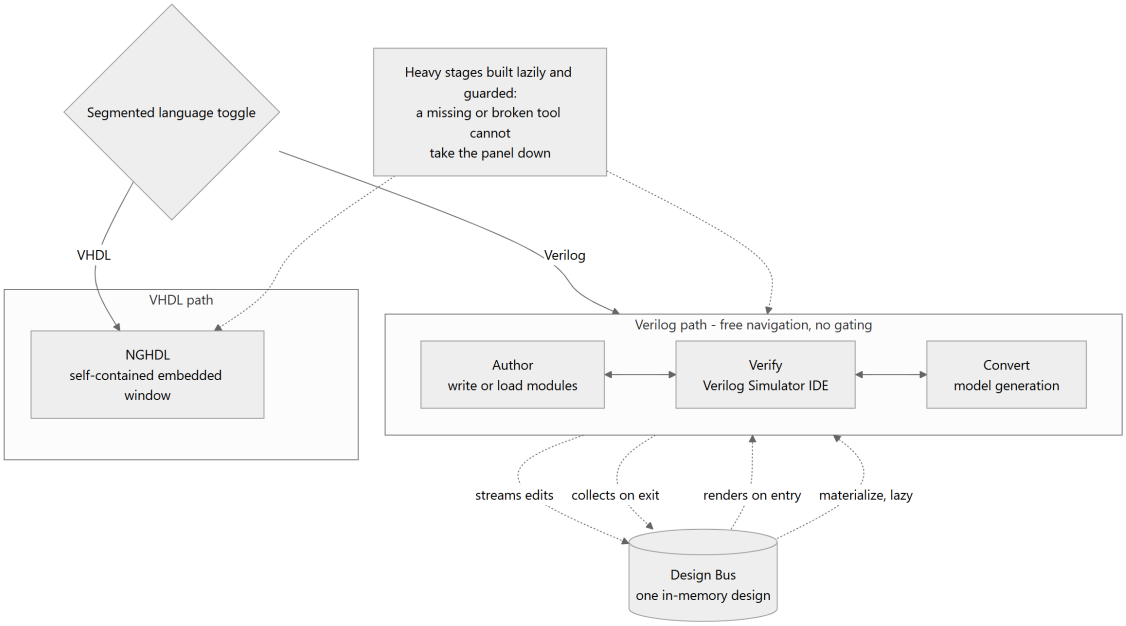


Figure 96: Architecture of the Verilog Flow Navigator. A language toggle separates the Verilog path (Author → Verify → Convert) from the VHDL path. All three Verilog stages operate on one shared in-memory design held by the Design Bus; heavy stages are built lazily and guarded so that a single failure can never take down the panel.

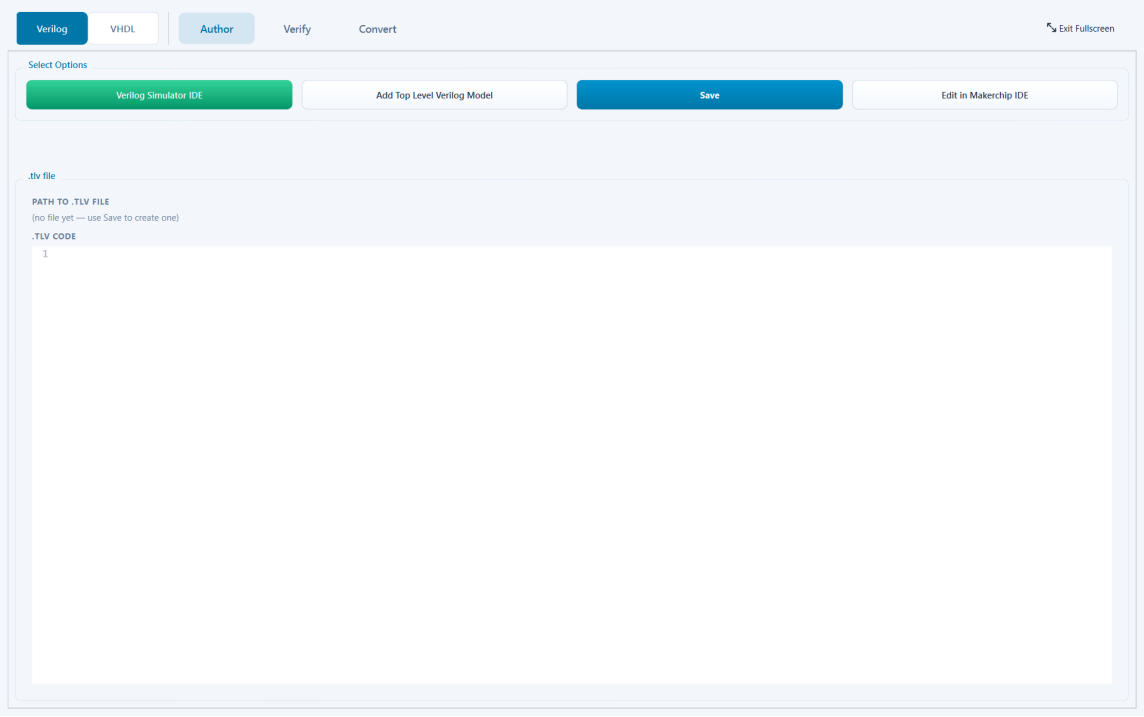


Figure 97: The Verilog Flow Navigator as rendered in eSim. The segmented Verilog/VHDL toggle sits at the far left and the *Author* → *Verify* → *Convert* stage tabs run across the top. The *Author* stage is shown here, hosting the authoring surface and the entry point into the Verilog Simulator IDE; any stage can be selected directly, with no gating between them.

14.2.1 The Design Bus: One Design, Three Views

A recurring source of friction in multi-stage tools is state synchronization: if each stage keeps its own copy of the design and hands it off through disk, the stages drift out of sync and a file watcher cannot distinguish the tool's own writes from genuine external edits. To eliminate this, the three Verilog stages are treated as **three views onto one design**, held in memory by a single owner — the *Design Bus*.

The Author stage streams its edits into the bus live; the Verify stage renders from it on entry and collects its edits back on exit; the Convert stage materializes it to a real file only when its toolchain needs a path. Disk writes therefore happen only on purpose, and each write records the SHA-256 of the exact bytes written, so the external-edit watch is echo-proof and fires only on a genuine outside modification.

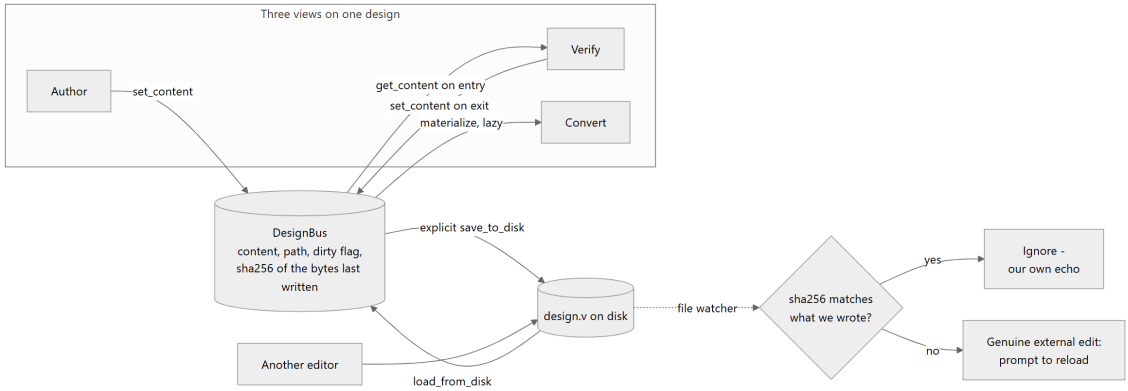


Figure 98: The Design Bus data model. A single in-memory design is shared by all three stages; navigation carries it without a disk round-trip, while an echo-proof hash guard distinguishes the tool's own writes from real external edits.

14.3 Layered Architecture

The Verifier is built as two cleanly separated layers. A **Qt-free core** performs all HDL work — process invocation, source parsing, and waveform decoding — as pure Python functions that touch only strings, files, and toolchains. On top of this sits the **presentation layer**: the editor tabs, hierarchy sidebar, console, and controls. This separation is deliberate. The core is fully unit-testable without a display, and, crucially, it is the *same* core reused by the co-simulation build described in the next chapter — so both features invoke the compiler through one identical code path, guaranteeing byte-for-byte consistent behaviour.

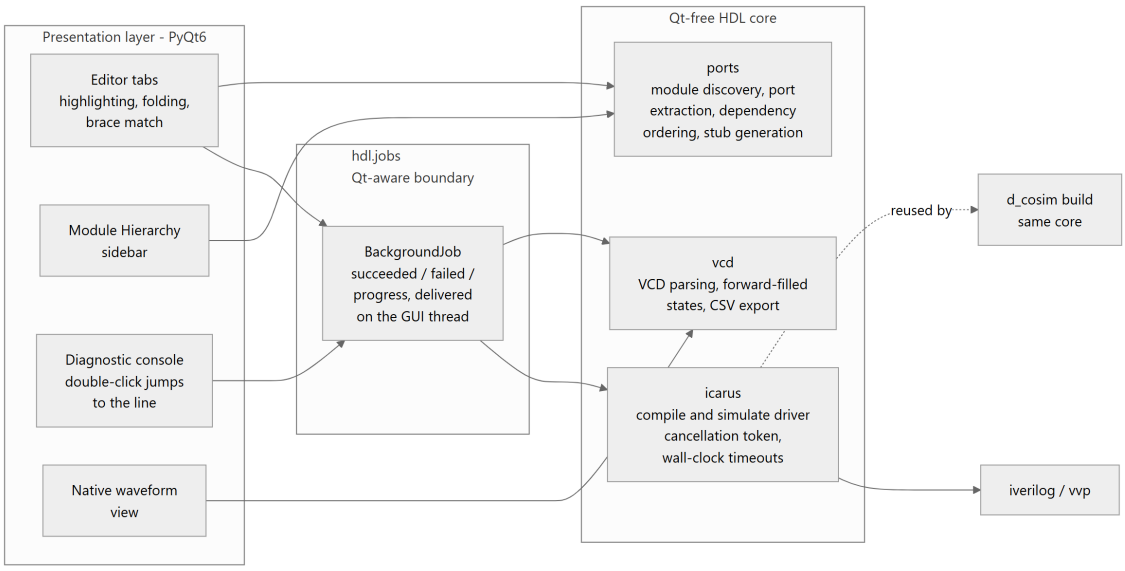


Figure 99: Layered architecture of the Verilog Verifier. The presentation layer drives a Qt-free HDL core (compile/simulate, structural parsing, VCD decoding) that is shared with the co-simulation backend. Long-running work is dispatched to a worker thread so the interface never freezes.

The core is organised into four focused modules, summarized in Table 37. Three of them are entirely Qt-free and therefore unit-testable without a display; `jobs` is the single Qt-aware member, and exists precisely to be the boundary at which results cross back onto the interface thread.

Module	Responsibility	Key Capabilities
`icarus`	Compile + simulate driver	`iverilog`/`vvp` invocation, cancellation token, wall-clock timeouts
`ports`	Structural analysis	Module discovery, port extraction, dependency ordering, testbench generation
`vcd`	Waveform decoding	VCD parsing, forward-filled signal states, CSV export
`jobs`	Concurrency (Qt boundary)	Worker-thread wrapper delivering results, failures and progress back to the interface thread

Table 37: The four modules of the HDL core. `icarus`, `ports` and `vcd` are Qt-free; `jobs` is the Qt-aware boundary between them and the interface.

A notable robustness property lives in the `ports` module. Its parser is comment- and string-proof (a module name mentioned inside a comment can never be mistaken for real code) and carries bus widths through faithfully, so a generated testbench declares matching-width registers rather than truncating a bus to a single bit. It also topologically orders a multi-module design so a parent module is presented before the modules it instantiates. For a design whose ports it can read, it emits a ready-to-simulate stub, wiring clock and reset stimulus and the `$dumpfile`/`$dumpvars` directives required for waveform capture:

Listing 27: Representative auto-generated testbench stub for a 4-bit counter

```
1   `timescale 1ns/1ps
2
3   module tb_counter;
4     // Inputs
5     reg clk;
6     reg rst;
7
8     // Outputs
```

```
    wire [3:0] count;

    // UUT Instance
    counter uut (
      .clk(clk), .rst(rst), .count(count)
    );

    always #5 clk = ~clk;

    initial begin
      clk = 0;
      rst = 1;
      #20 rst = 0;
      $dumpfile("sim_out.vcd");
      $dumpvars(0, tb_counter);
      #500;
      $finish;
    end
  endmodule

```

14.4 Functionality & User Workflow

The Verifier presents a full IDE surface:

- **Multi-tab source editors** with Verilog/VHDL syntax highlighting, line numbers, code folding, and brace matching, reusing eSim's shared editor theme so HDL looks identical to the rest of the application.
- **A Module Hierarchy sidebar** that auto-detects the structural ordering of a multi-module design and allows manual reordering.
- **A pinned Testbench tab** plus one-click *Auto-Generate Testbench*.
- **An interactive console** in which a compiler diagnostic of the form `design.v:15: error` is clickable: a double-click jumps directly to the offending line and marks it with an error squiggle in the correct editor tab.
- **A native waveform view**: on a successful run the decoded VCD is rendered through eSim's own plotting window, adapted for digital traces and opened as its own full-width tab. The same data can be exported to CSV.

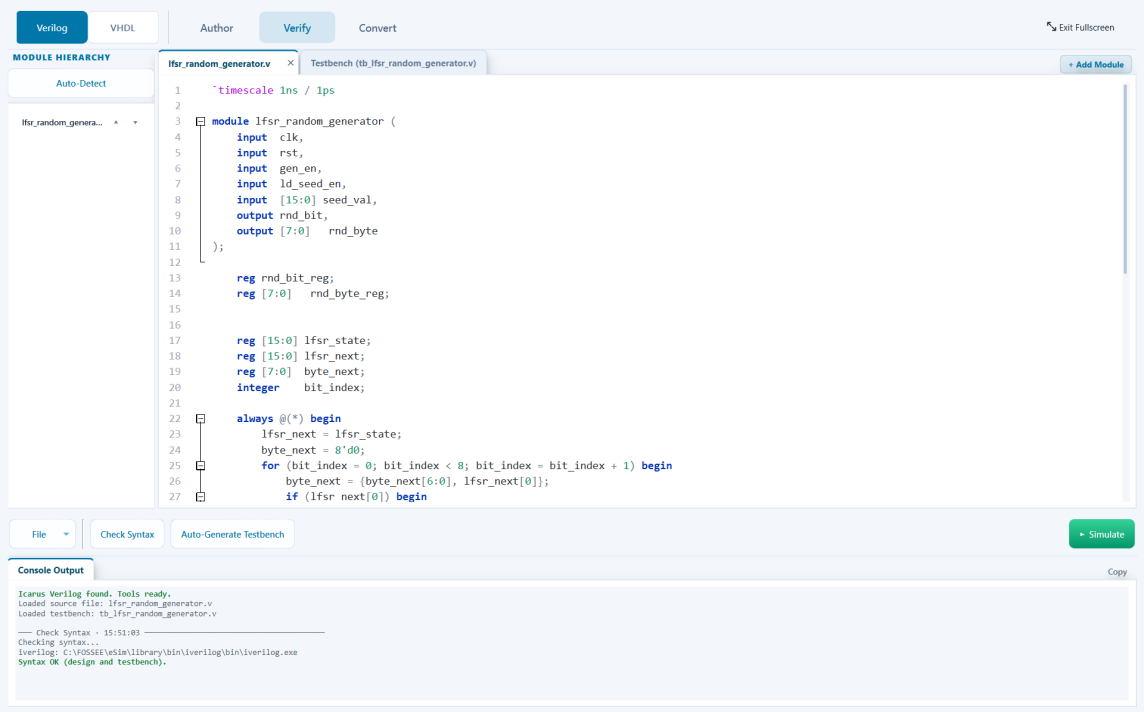


Figure 100: The Verilog Verifier interface during a real session on the LFSR pseudo-random generator. The design and its testbench occupy separate editor tabs, the Module Hierarchy sidebar has auto-detected `lfsr_random_generator` as the top module, and the console confirms that the bundled Icarus toolchain was located and that both files pass the syntax check.

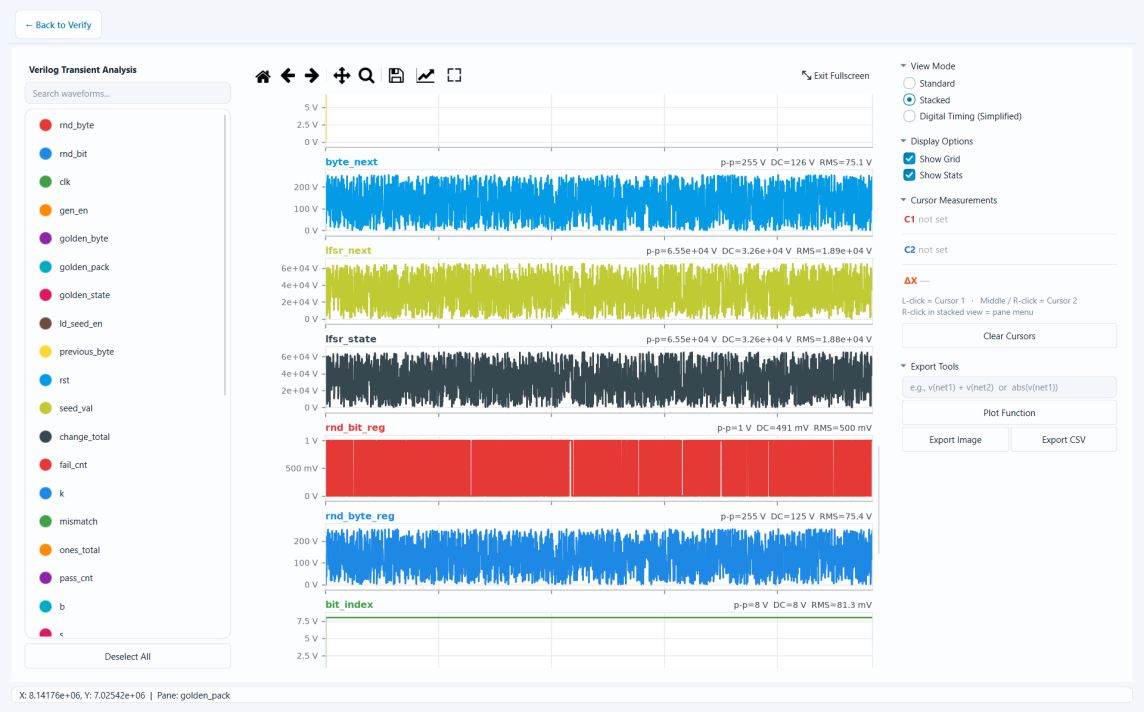


Figure 101: Native waveform rendering of the simulated LFSR. The VCD produced by the Icarus run is decoded and plotted in eSim's own waveform window in stacked mode. The trace list is not limited to the module's ports: internal registers such as `lfsr_state`, `lfsr_next` and `byte_next` are plotted alongside them, and per-trace peak-to-peak, DC and RMS figures are computed on the fly.

14.4.1 Execution Pipeline: From Source to Waveform

When the user runs a simulation, the Verifier writes each source tab to a temporary directory under a sanitized, per-tab filename (so a compiler diagnostic references the tab the user actually sees), compiles the design with `iverilog`, executes the resulting artifact under `vvp`, reads back the generated VCD, decodes it, and renders the waveform. The entire blocking sequence runs on a worker thread, leaving the interface responsive and cancellable throughout.

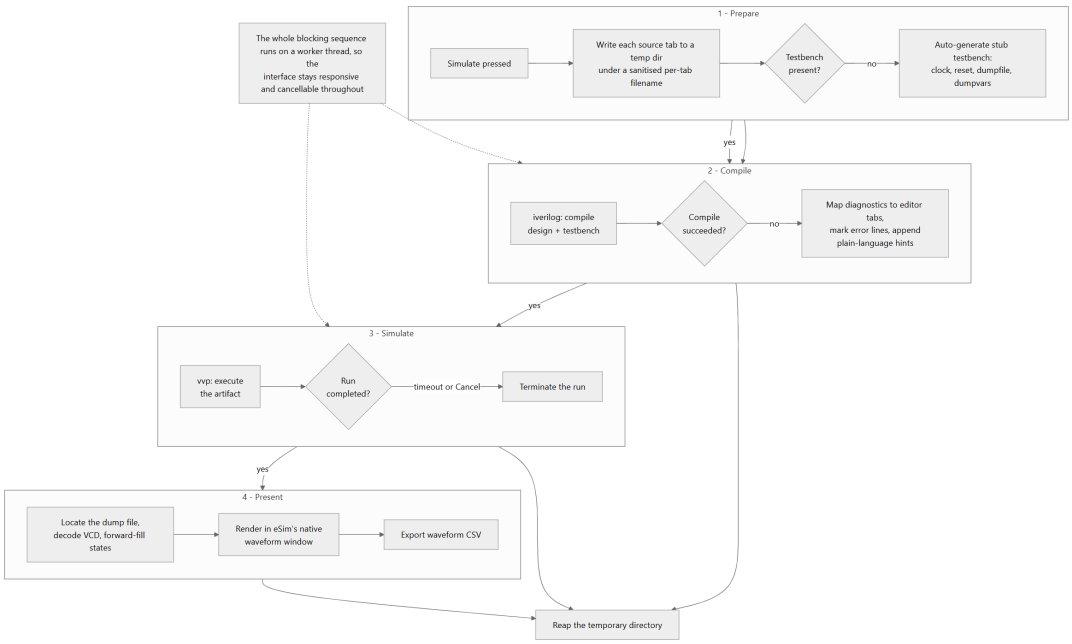


Figure 102: The Verifier execution pipeline. Source and testbench are compiled with Icarus; on success the artifact is simulated, its VCD decoded, and the waveform rendered natively. A compilation failure short-circuits the run and drives the error-mapping path instead.

14.4.2 Robustness Engineering

Several failure modes intrinsic to driving an external simulator are handled explicitly. A runaway testbench (for instance a free-running clock with no `$finish`) cannot hang the tool: compile and simulate operations carry wall-clock timeouts, and a user-visible *Cancel* control kills the underlying process immediately through a cancellation token, closing the race in which a process is spawned just as cancellation is requested. Temporary build directories are reaped even if the panel is torn down mid-run. A small diagnostic layer scans compiler output for common error signatures and appends targeted, plain-language hints (for example distinguishing an undeclared identifier from an illegal left-hand-side assignment), turning raw compiler noise into actionable feedback. Finally, if the Icarus toolchain is absent, the interface locks its build controls and offers a single "locate" action rather than failing obscurely at run time.

14.5 Worked Example: Verifying the LFSR Pseudo-Random Generator

A verification tool is only as good as its agreement with an independent reference. The LFSR pseudo-random generator makes a demanding subject for that comparison: it is sequential, it is checked against a golden model, and two of its six checks are *statistical* rather than directed, so an off-by-one in the delivered clock would move the measured numbers rather than simply flipping a pass to a fail. Its Phase 1 results in Table 22 were produced by an entirely separate simulator (Xilinx XSim) on an entirely separate toolchain, which makes them an ideal control.

Running the identical design and testbench through the in-application Verifier — Figure 100 shows the session, Figure 101 the resulting waveform — reproduces every

reported quantity exactly.

Quantity	Vivado XSim (Phase 1)	eSim Verifier (Icarus)	Agreement
Checks passed	6 of 6	`pass_cnt` = 6	exact
Checks failed	0	`fail_cnt` = 0	exact
Golden-model mismatches	0	`mismatch` = 0	exact
Ones in 2048 serial bits (TC5)	1002 (48.9 %)	`ones_total` = 1002	exact
Byte transitions in 300 clocks (TC6)	300 (100 %)	`change_total` = 300	exact

Table 38: Cross-simulator agreement on the LFSR. The two statistical measures are the strongest evidence here: a single mis-delivered clock edge would perturb the bit balance away from 1002, and any missed edge would drop the transition count below 300.

This is a stronger result than a pass/fail match. `ones_total` and `change_total` are accumulated sample-by-sample over 2048 and 300 clocks respectively, so they are sensitive to the exact number of edges the simulator delivers. Both landing on the reference value confirms that the Verifier is not merely running the design, but advancing it cycle-for-cycle in step with an independent implementation.

14.5.1 Full-Design Visibility

The decoded VCD covers the whole design hierarchy rather than just the module boundary. For this run the Verifier captured **38 traces** spanning the full 24.2 μs testbench sequence: the six ports, the internal state registers (`lfsr_state`, `lfsr_next`, `byte_next`, `rnd_byte_reg`, `bit_index`), the testbench's own bookkeeping variables (`golden_state`, `pass_cnt`, `fail_cnt`, `ones_total`), and even the local variables of its checking tasks, which appear under their own scopes as `log_range.*` and `log_result.*`. Over the run `rnd_byte` was observed to take **all 256** distinct 8-bit values.

That distinction matters for this project specifically. In the mixed-signal test schematic only the *ports* cross the ADC/DAC bridges, so a fault inside the feedback path can be observed only through its effect on the output byte. The Verifier exposes the state register itself, which is where such a fault would actually be visible — and it does so before any bridge has been placed.

Figure 103: The complete captured signal set for the LFSR run, exported from the Verifier's waveform window and laid out as three columns to fit the page. Ports, internal registers, testbench bookkeeping variables and the checking tasks' own locals are all present, each annotated with its peak-to-peak, mean and RMS value. The verification result can be read straight off those annotations: `pass_cnt` reaches 6 while `fail_cnt` and `mismatch` stay flat at zero — no check failed, and the golden model was never contradicted.

The same capture exports to CSV as one column per captured signal and one row per time point. That is how the right-hand column of Table 38 was read back out and compared against the Phase 1 numbers: verification evidence leaves the tool in a form that can be checked by a script rather than by eye.

14.6 Implications

The Verifier changes the shape of the digital design loop in eSim. Functional bugs are now caught in the same window in which the RTL is authored, so an IP is proven correct before a single ADC/DAC bridge is placed on a schematic. Because the waveform is rendered through eSim's own plot window, the digital and mixed-signal views share one visual language. And because a verified design is carried forward in memory to the Convert

stage, there is no export/re-import step between proving a design and converting it.

14.7 Conclusion

The Verilog Verifier brings native HDL compilation, simulation, and waveform inspection directly into eSim for the first time. Built on a tested, Qt-free core and surfaced through the workflow-shaped Flow Navigator, it removes the dependence on an external simulator for functional verification and provides a smooth path from an authored module to a proven one, ready for mixed-signal conversion.

14.8 Future Prospects & Improvements

- **Assertion- and coverage-driven checks:** surface pass/fail assertions and basic toggle/line coverage alongside the waveform, so correctness is reported, not just plotted.
- **Waveform measurements:** cursor-based time/value readouts and signal search within the native plot, approaching a dedicated waveform viewer.
- **Regression batches:** run a suite of testbenches headlessly and summarise results, enabling the kind of automated Phase 1 sweeps performed manually in this report.
- **Deeper language support:** broaden SystemVerilog construct coverage and optional linting to catch style and synthesizability issues before conversion.

15 Icarus-Based Digital Co-Simulation (`d_cosim`)

15.1 Introduction & Need

The IP blocks in this report reached the analog matrix through eSim's established model-generation flow, in which a Verilog module is transpiled by Verilator into a C++ model and compiled into a static Ngspice code model. That path is powerful, but heavy: it depends on a full C/C++ toolchain on the user's machine and rebuilds an Ngspice code-model library for *every* model created. For rapid iteration on small digital IPs, the round-trip cost is significant.

Icarus-based co-simulation introduces a second, lightweight backend for the same goal. Rather than transpiling to C++, the Verilog module is compiled *once* by `iverilog` into a compact runtime artifact, and that artifact is loaded **by Ngspice itself at simulation time** through Ngspice's upstream `d_cosim` code model and its Icarus adapter. The practical consequences are direct: no C/C++ compiler is required, Ngspice is never rebuilt, and model creation is a single fast compile rather than a full code-model reinstall.

In the Convert stage the two backends are offered with equal billing — the lightweight Icarus option (labelled *Dual Co-sim*) alongside the established transpilation option — so the user chooses the trade-off appropriate to their design. Table 39 contrasts them.

Aspect	**Legacy transpilation backend**	**Icarus `d_cosim` backend**
Translation	Verilog → C++ (Verilator)	Verilog → compiled Icarus artifact
Host toolchain	Requires a C/C++ compiler	No C/C++ compiler required
Ngspice impact	Rebuilds an Ngspice code-model library	No Ngspice rebuild; loaded at run time
Model creation cost	Full library reinstall per model	Single `iverilog` compile
Runtime mechanism	Static code model linked into Ngspice	`d_cosim` code model + Icarus adapter

Table 39: The two conversion backends, offered side by side in the Convert stage.

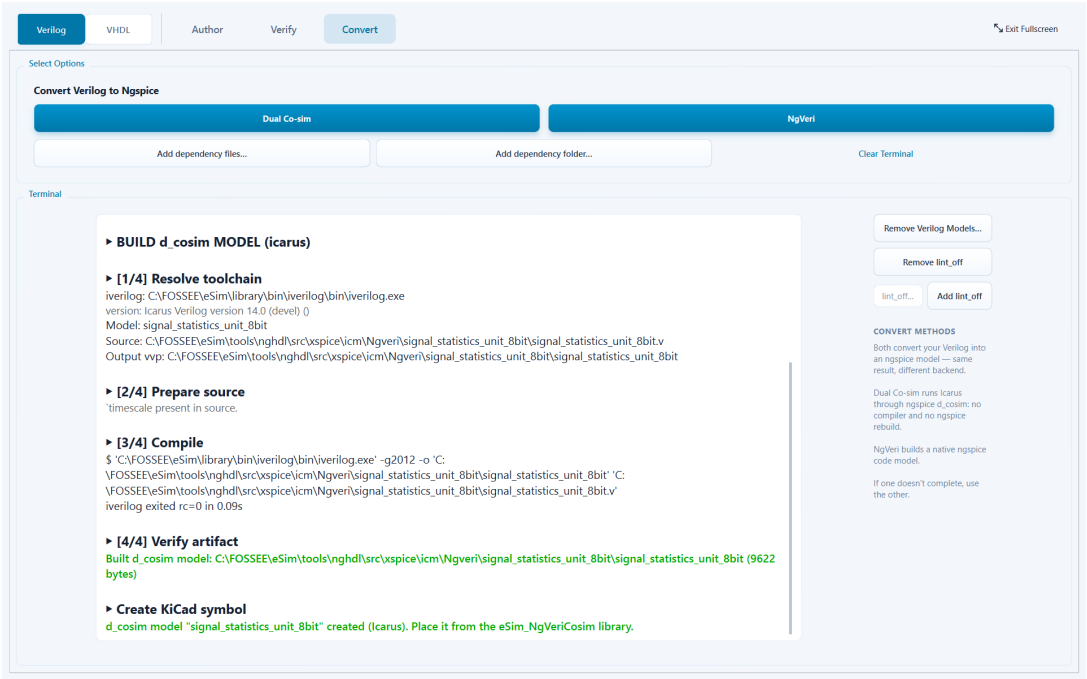


Figure 104: The Convert stage offering both backends side by side: the lightweight Icarus *Dual Co-sim* path and the established transpilation path. The terminal records the four-stage Icarus build for the Signal Statistics Unit — toolchain resolution, source preparation, compilation, and artifact verification — followed by KiCad symbol generation into the dedicated `eSim_NgVeriCosim` library.

Collaboration and Attribution

The user-interface framework, the in-application Verilog Verifier, and the Icarus-based co-simulation backend (Chapters 13–15) were co-architected and prototyped in close collaboration with my colleague **Mr. Varadharam R. S.** Following initial design and joint prototyping, Mr. Varadharam led the production hardening, bug fixes, and edge-case validation to bring these modules to release readiness. The technical accounts in the following chapters reflect this shared engineering effort and detail the joint architectural decisions that shaped the platform.

15.2 The Co-Simulation Stack

At simulation time, Ngspice's `d_cosim` code model presents the Verilog block to the analog matrix as a single mixed-signal component with a fixed two-port digital interface — one grouped digital input vector and one grouped digital output vector. Its Icarus adapter drives the compiled artifact to advance the digital logic in step with the analog solver, translating between the discrete logic states of the Verilog domain and the continuous node voltages of the SPICE domain, exactly as the standard `adc_bridge`/`dac_bridge` components do for other digital blocks.

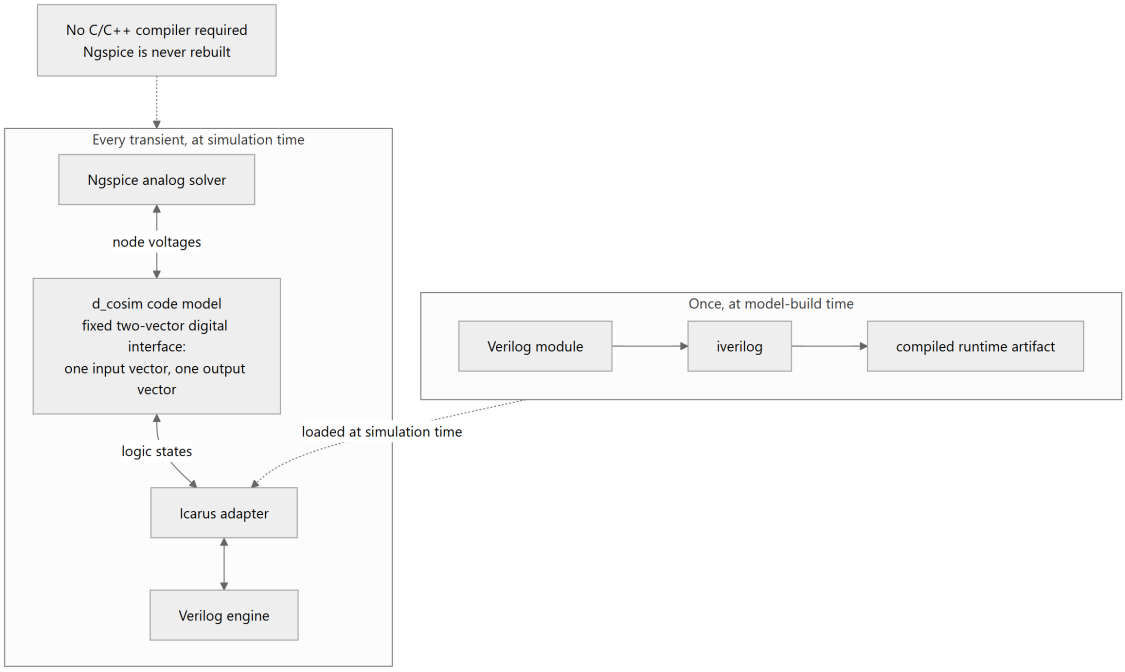


Figure 105: The `d_cosim` co-simulation stack. Icarus compiles the Verilog module once into a runtime artifact; during the transient, Ngspice's `d_cosim` code model and its Icarus adapter load and drive that artifact, exchanging logic states and node voltages with the analog solver at each step.

15.3 Architecture & Build Pipeline

The feature is composed of four cooperating parts, all of which resolve their paths at call time so nothing is hardcoded to an install location.

- **Toolchain resolver.** A dedicated resolver locates the Icarus toolchain using a single, shared search order: an explicit environment override, then recorded configuration, then a bundled location, then the system path, and finally the platform-standard install directory. This is the *same* resolver the Verilog Verifier uses, so both features always find the identical compiler. Capability gates report clearly when a prerequisite is missing rather than failing mid-run.

- **Model builder.** The build step compiles the chosen `.v` to a compiled artifact at one canonical location that the netlister independently re-derives, so the build and the simulation always agree on the artifact's path without storing it anywhere. If the source lacks a `` `timescale ``, one is injected transparently so the digital engine advances correctly. The compile itself runs on a worker thread so the interface stays responsive.

- **Symbol generator.** A KiCad symbol and its parameter record are emitted into a dedicated library so that `d_cosim` blocks and legacy blocks coexist in the same schematic. The digital ports are collapsed onto the fixed two-port `d_cosim` interface (one input vector, one output vector).

- **Netlister hook.** During KiCad-to-Ngspice conversion, the block is emitted as a `d_cosim` model line naming the adapter, with the compiled artifact passed to it as

an argument. Both runtime files the adapter needs — the compiled artifact and the adapter library itself — are staged next to the netlist, where they resolve relative to the Ngspice working directory.

A structured, dual-sink logger records the entire flow — build, netlist staging, and run — to both the terminal and a persistent, timestamped log file, and (when a graphical context is present) to the Convert panel as severity-coloured lines, so build, netlist, and run events are visible to both developers and end users.

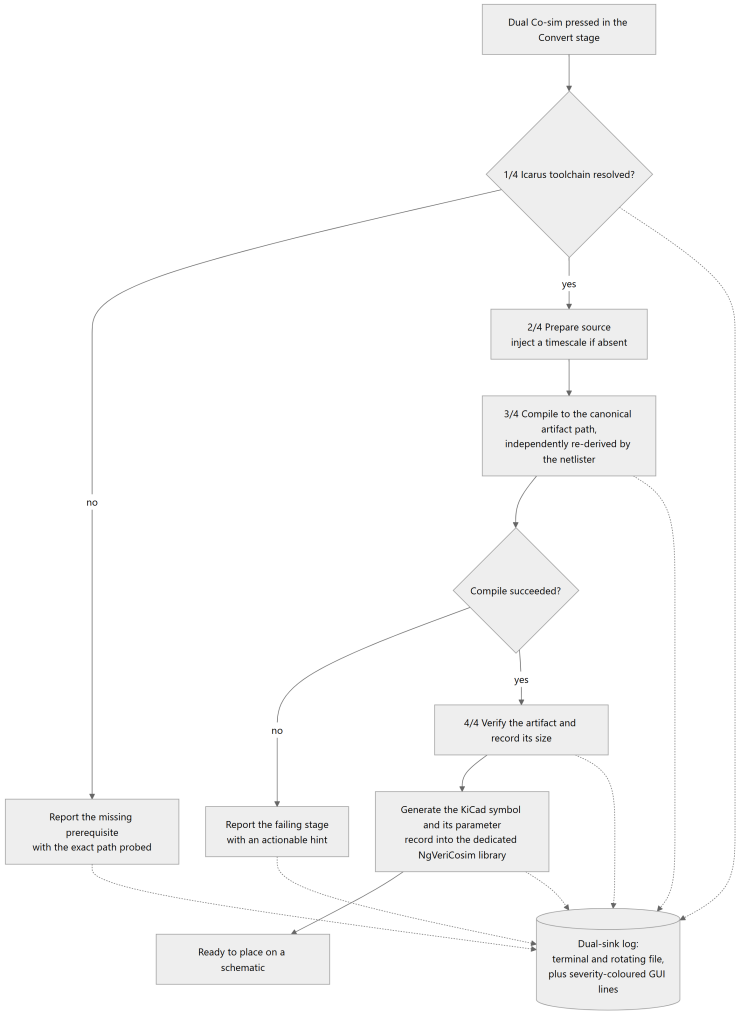


Figure 106: The `d_cosim` model-build pipeline. After a toolchain check, the source is compiled to a canonical artifact path, a KiCad symbol is generated in a dedicated library, and the block is ready to place on a schematic. Failures are reported with an actionable hint at the exact stage they occur.

15.4 Functionality & Simulation Data-Flow

From the user's standpoint the flow is: author or load a Verilog module in the Author stage, open the Convert stage, and press *Dual Co-sim*. The model is compiled and a KiCad symbol is produced and registered, ready to be placed on a schematic from its dedicated library. When the schematic is netlisted and simulated, Ngspice loads the

compiled artifact through the `d_cosim` adapter and advances the Verilog logic in lockstep with the analog transient, producing a mixed-signal waveform in the usual way.

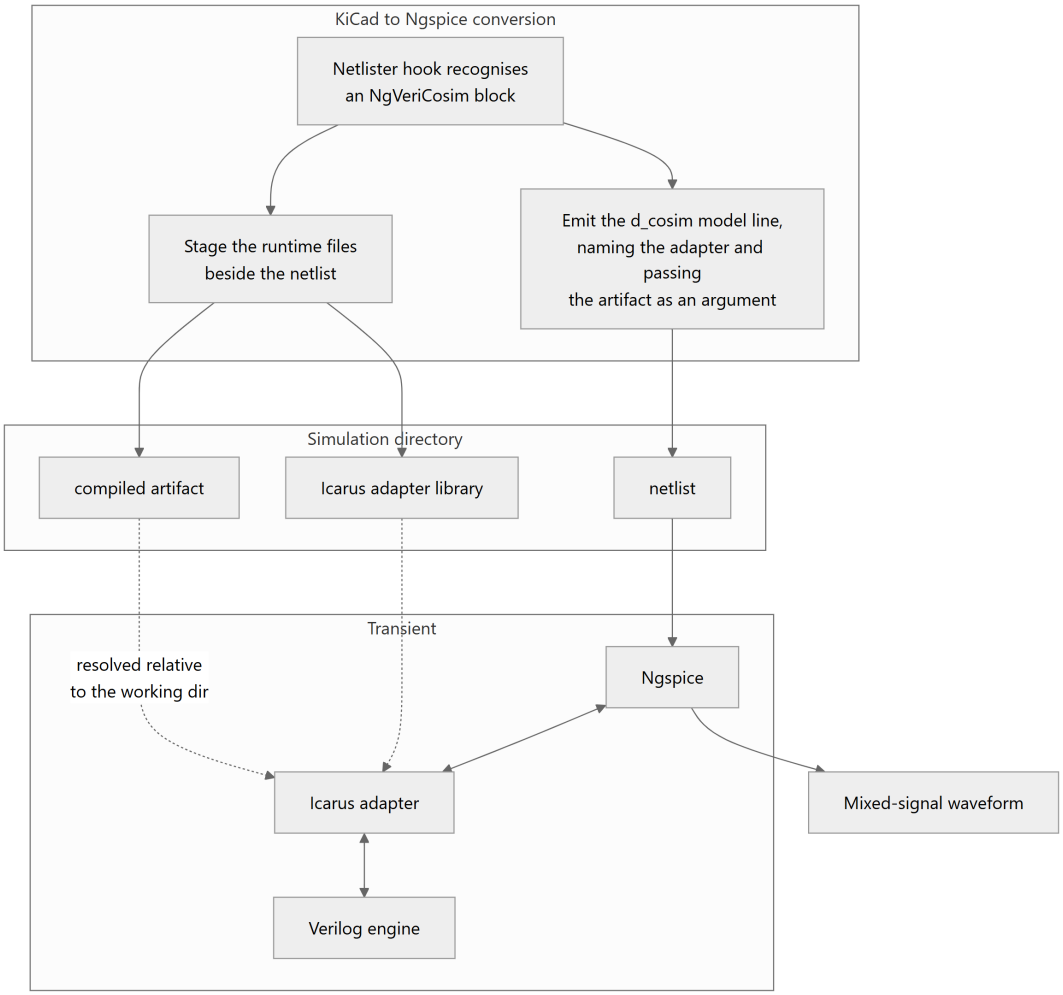


Figure 107: The `d_cosim` runtime data-flow. The netlister emits the model line and stages the compiled artifact beside the netlist; during the transient, Ngspice loads the artifact through the Icarus adapter and advances the digital logic together with the analog solver, yielding a mixed-signal waveform.

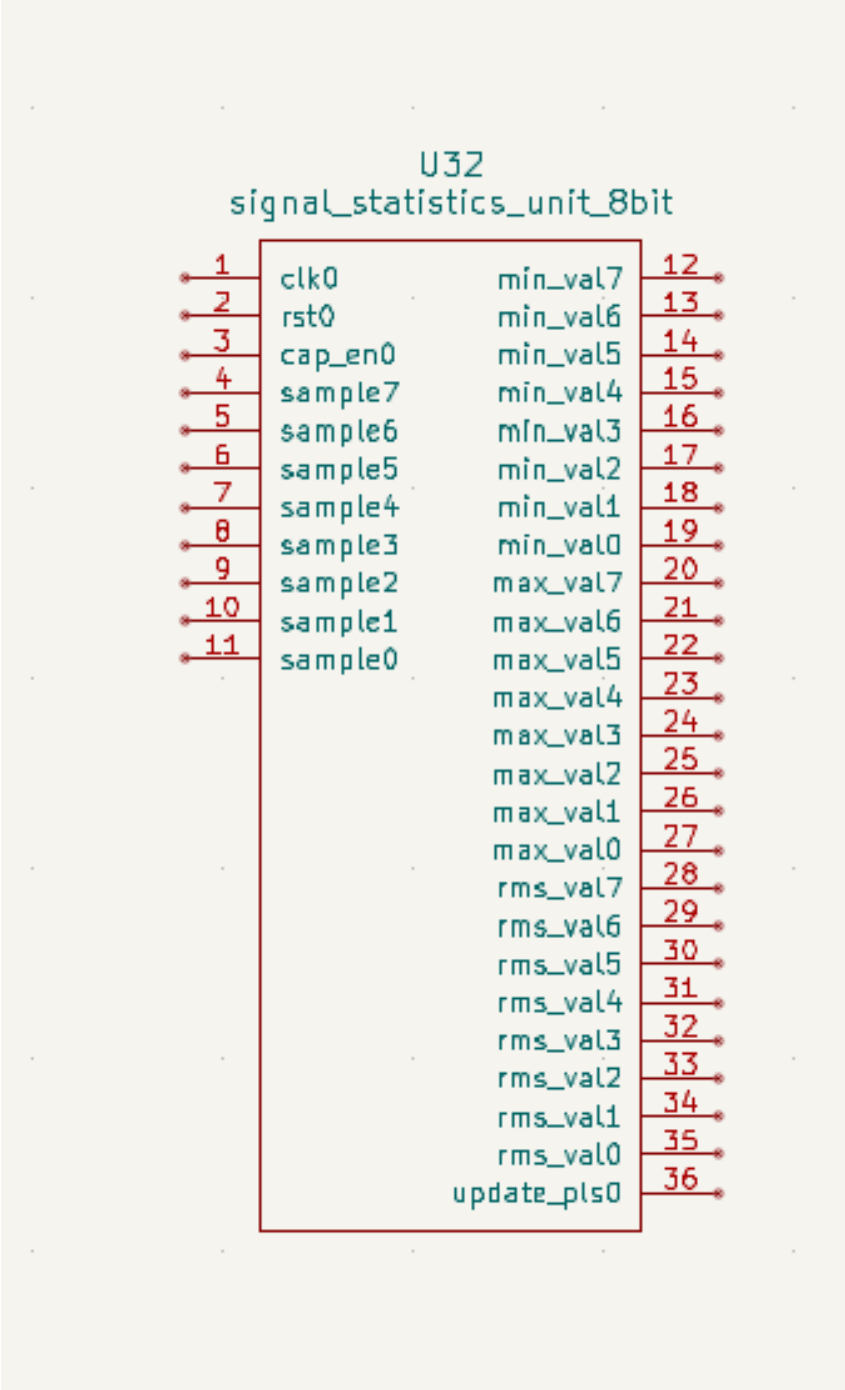


Figure 108: The `d_cosim` KiCad symbol generated for the Signal Statistics Unit, placed from its dedicated library onto a schematic. The *symbol* keeps the individually named pins a designer needs in order to wire a sheet — 11 inputs (`clk0`, `rst0`, `cap_en0`, `sample7..0`) and 25 outputs (`min_val7..0`, `max_val7..0`, `rms_val7..0`, `update_pls0`) — while the netlister collapses exactly those pins onto the fixed two-vector `d_cosim` interface. An existing test schematic therefore accepts the block without rewiring.

15.5 Worked Example: The Signal Statistics Unit through the Icarus Backend

The Signal Statistics Unit was taken end to end through the Icarus path, reusing the test schematic already built for it. Because the generated symbol carries the same pin names as the transpilation-backend symbol, no rewiring was needed — the existing sheet accepted the generated block and was converted as it stood.

15.5.1 Author and Verify

The design was first proven in the Verify stage, which reports a clean syntax check on both the module and its testbench against the bundled Icarus toolchain.

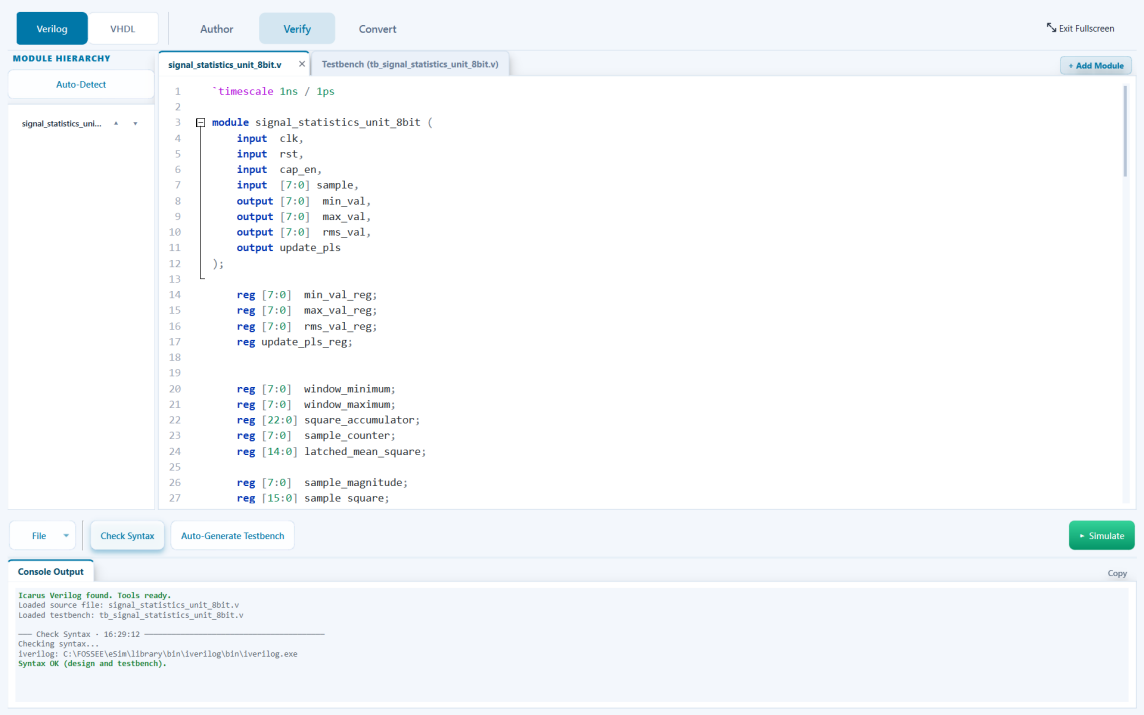


Figure 109: The Signal Statistics Unit in the Verify stage prior to conversion. The Module Hierarchy sidebar has resolved `signal_statistics_unit_8bit` as the top module and the console reports `Syntax OK (design and testbench)`.

Figure 110: The pure-digital run of the same design in the Verifier, shown in digital-timing mode. Alongside the ports, the internal arithmetic of the RMS path is visible — `square_accumulator`, `latched_mean_square`, `sqrt_root`, `sqrt_remainder` — together with the testbench's `expected_min`/`expected_max`/`expected_rms` references and its `pass_cnt`/`fail_cnt` counters.

15.5.2 Build and Symbol Generation

Pressing *Dual Co-sim* in the Convert stage runs the four-stage build of Figure 104: the toolchain is resolved, the `‘timescale` is confirmed present, the source is compiled, and the artifact is verified. `iverilog` returns in **0.09 s** and yields a **9 622-byte** runtime artifact. No Ngspice library is rebuilt and no C/C++ compiler is invoked at any point. The symbol produced by that build is the one shown in Figure 108.

15.5.3 What the Netlister Emits

Comparing the converted project against the same schematic converted through the transpilation backend shows exactly where the two paths diverge. The analog half of the netlist — sources, bridges, probes and the bridge model cards — is unchanged; three things differ, and all three are consequences of the `d_cosim` interface contract.

	Transpilation backend	**Icarus `d_cosim` backend**
Port grouping	eight bracketed groups, one per port	two bracketed vectors: 11 inputs, 25 outputs
Model card	`.model u2 signal_statistics_unit_8bit(...)`	`.model u2 d_cosim simulation="ivlng" ...`
Transient card	`.tran` at top level, `run` in `.control`	`tran` issued from within `.control`

Table 40: The three netlist-level differences between the two conversion backends for an otherwise identical schematic.

The emitted model card names the adapter rather than the design, and passes the compiled artifact to it as an argument:

Listing 28: The model card emitted for the Icarus backend

```
a2 [clk_d rst_d logic1_d logic0_d logic1_d logic1_d logic0_d logic0_d
    logic1_d logic0_d logic0_d]
   [min_val7_d ... min_val0_d max_val7_d ... max_val0_d
    rms_val7_d ... rms_val0_d update_pls0_d] u2

.model u2 d_cosim simulation="ivlng"
          lib_args=["libvvp", "ivlng"]
          sim_args=["signal_statistics_unit_8bit"]

```

Alongside the netlist the converter stages the two files the adapter needs at run time:

Artifact	Size	Role
`signal_statistics_unit_8bit`	9 622 bytes	the compiled Icarus runtime artifact for the design
`ivlng.vpi`	268 858 bytes	the Icarus adapter Ngspice loads to drive that artifact

Table 41: Files staged beside the netlist so the adapter resolves them relative to the Ngspice working directory.

15.5.4 Mixed-Signal Run

With the model built and the netlist emitted, the schematic was simulated in Ngspice exactly as any other mixed-signal design would be. The transient card requests a `0.1` μ`s` step over an `8 ms` window, and the run returns the full set of bridged node voltages together with the branch currents of the bridges themselves. Figure 111 places that result beside the same schematic converted through the transpilation backend.

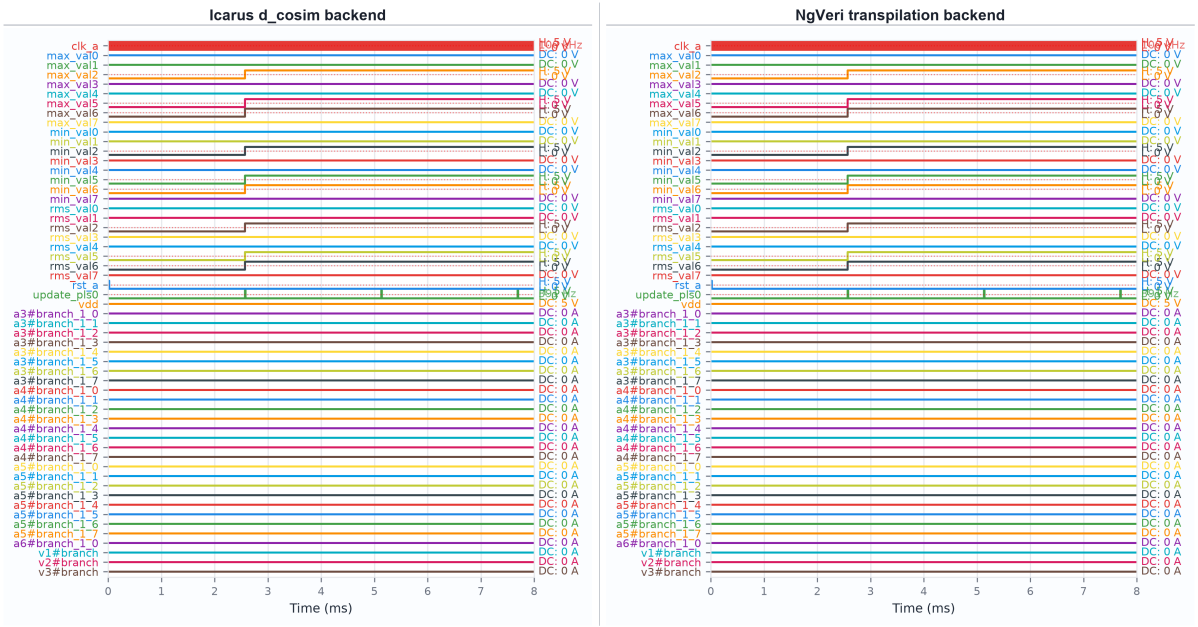


Figure 111: The Signal Statistics Unit simulated from the same schematic through each backend — the Icarus `d_cosim` path on the left, the transpilation path on the right. The `min_val`, `max_val` and `rms_val` buses settle at 2.57 ms and hold, as expected for a constant input, and `update_pls0` appears as three narrow one-clock strobes.

15.5.5 Model Lifecycle

Because a `d_cosim` model is a compiled artifact plus a registered symbol rather than a linked library, it can also be withdrawn cleanly. The removal action deletes the generated symbol, its parameter record and the build directory in one step, leaving no entry behind in the library — a practical difference from the transpilation path, where removing a model touches an Ngspice code-model library shared with every other block.

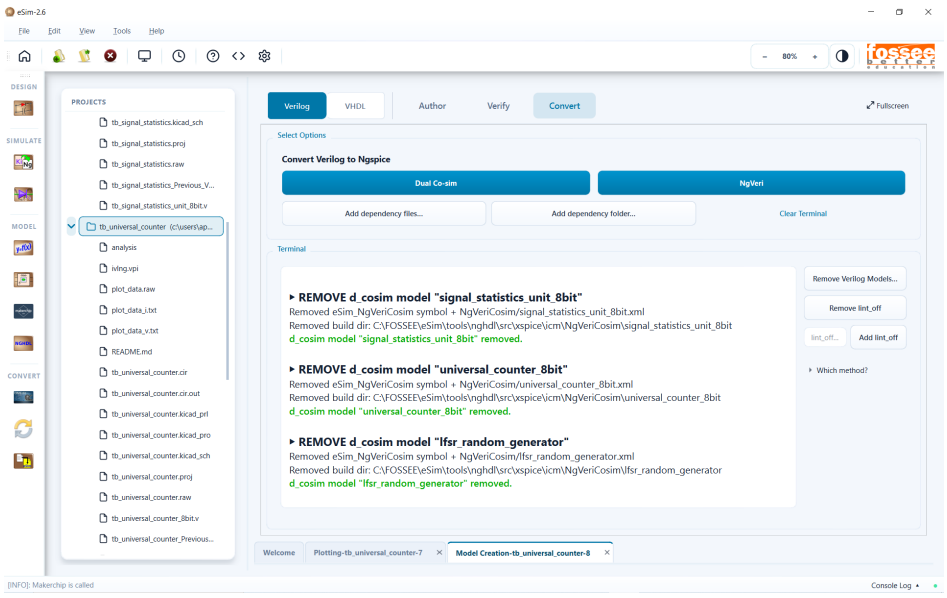


Figure 112: Removing `d_cosim` models. Three models are withdrawn in a single action, and for each one the generated symbol, the `NgVeriCosim` parameter record and the build directory are reported individually as removed — so the operation is auditable rather than silent, and the library is left with no orphaned entries.

15.6 Cross-Backend Equivalence

Because both backends are driven from the same schematic, the two paths can be compared directly: convert one project each way and look at the waveforms. Only the model card of Table 40 differs between the two netlists — the sources, bridges, probes and transient card are identical — so any difference in the outputs would have to come from the conversion itself.

Three designs were taken through both paths: the Universal Up/Down Counter, the LFSR Pseudo-Random Generator and the Signal Statistics Unit. In each case the two backends produced the same waveform. The counter loads and counts `0, 1, 250, 251, ..., 255, 0, 1, 2, ...` with a single one-clock `wraps0` strobe at 9 ms, and the LFSR emits the same byte stream either way — `135, 35, 70, 220, 176, 221, 238, ...`, 44 distinct values over its 50 clocks. The pseudo-random case is the more searching of the two, since the state is carried through a feedback shift register and any difference in the delivered clock would send the sequence down a different branch.

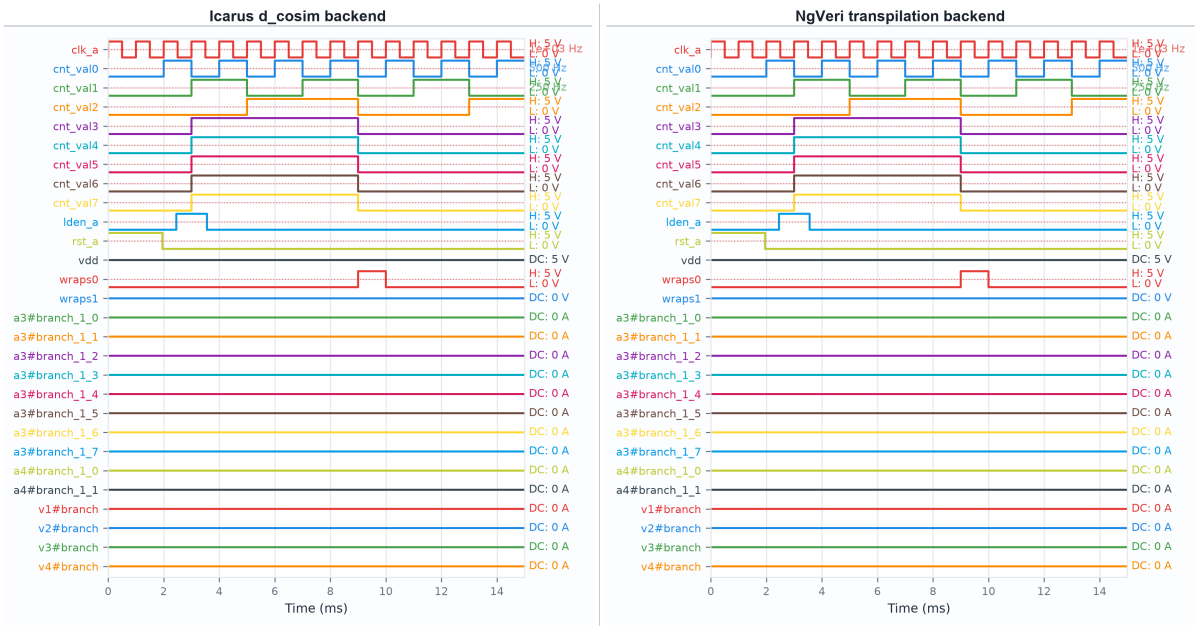


Figure 113: The Universal Up/Down Counter converted through each backend from the same schematic — the Icarus `d_cosim` path on the left, the transpilation path on the right.

Figure 114: The LFSR Pseudo-Random Generator through the Icarus backend over a 50 ms window, the export laid out as three columns to fit the page. The eight `rnd_byte` bits and the serial `rnd_bit0` output show the aperiodic switching expected of the maximal-length sequence; the bridge branch currents in the remaining columns sit at zero throughout, as they should for a purely digital block.

In practice this means the choice of backend is a choice of build mechanism rather than of behaviour, which is what makes the two interchangeable rather than merely both

available.

15.7 Implications

The `d_cosim` backend lowers the barrier to mixed-signal digital design in eSim. A user without a C/C++ toolchain can still bring a Verilog block into an analog simulation, and iteration is faster because each model is a single compile rather than a code-model rebuild. Because the two backends coexist, the choice is additive: the established path remains for cases that need it, while the lightweight path serves rapid iteration and simpler deployment. Paired with the Verilog Verifier, the two features form one continuous loop — author, prove, and convert — all within eSim and all backed by the same shared compiler core.

15.8 Conclusion

Icarus-based co-simulation adds a second, lightweight route from Verilog to a simulation-ready mixed-signal block in eSim. By compiling once and letting Ngspice load the result at run time, it removes the C/C++ toolchain dependency and the per-model Ngspice rebuild of the transpilation path, while producing a first-class KiCad symbol that lives comfortably alongside existing models. Offered with equal billing in the Convert stage and sharing its toolchain resolver and compiler core with the Verilog Verifier, it rounds out a coherent, in-application digital design and verification workflow.

15.9 Future Prospects & Improvements

- **Bidirectional (`inout`) ports:** extend the interface mapping to cover bidirectional signals, broadening the class of designs the backend accepts.
- **Automatic test harness generation:** turn a verified testbench from the Verifier stage directly into a mixed-signal test schematic, removing manual bridge placement.
- **Parameterized blocks:** carry Verilog parameters through to the generated symbol so a single source can produce a family of configured blocks.
- **Unified reporting:** surface the co-simulation log inline in the schematic view, so build and run diagnostics appear where the user is working.