Design and Verification of a Parametric N-Bit Universal Synchronous Counter

A Comprehensive Project Report

Supriyo Roy

September 14, 2025

Abstract

This report details the complete design and verification process of a parametric N-bit universal synchronous counter. The primary objective was to create a scalable and reusable digital logic module capable of performing four distinct operations: holding its state, counting up, counting down, and loading a parallel value. The project employed a modern, software-driven verification methodology, utilizing Verilog for the hardware description (RTL) and Python for creating a high-level "golden model" to automate test vector generation. A self-checking Verilog testbench was used to simulate the design, comparing its output against the Python model's predictions. The design was successfully implemented, simulated, and verified, proving its logical correctness and scalability from an 8-bit to a 16-bit configuration.

Contents

1 Introduction

The counter is a fundamental building block in digital electronics, serving as the basis for everything from simple timers to program counters in complex microprocessors. A universal counter, which can be controlled to perform multiple operations, is a particularly versatile component.

1.1 Project Objectives

The core goal of this project was to design a single, scalable N-bit counter module with the following features:

- **Parametric Width (N-bit):** The design should be easily configurable to any bit width without changing the core logic.
- **Synchronous Operation:** All state changes must occur on the rising edge of a clock signal.
- **Asynchronous Reset:** An active-low reset signal should immediately and asynchronously set the counter's output to zero.
- **Controlled Operations:** A 2-bit control input should select one of four operations:
  - `2'b00`: Hold the current state.
  - `2'b01`: Increment the count by 1.
  - `2'b10`: Decrement the count by 1.
  - `2'b11`: Load a parallel N-bit value.

1.2 Verification Methodology

Rather than relying on manual, directed testbenches which are often tedious and incomplete, this project adopted a modern, robust verification strategy. This software-driven approach combines the strengths of a hardware description language with a high-level programming language to achieve comprehensive and automated testing.

2 Toolchain and Workflow

The project was executed using a toolchain of open-source software, each with a specific role:

1. **Python:** Used to write a "golden model" of the counter. This high-level model acts as the specification, defining the mathematically correct output for any given input. The script then generates a rich set of directed and randomized test vectors.
2. **Verilog (RTL):** Used to describe the actual hardware implementation of the counter (`counter.v`).
3. **Verilog (Testbench):** A self-checking testbench (`tb_counter.v`) was created to instantiate the Verilog RTL, apply the stimuli from the Python-generated file, and automatically check the hardware's output against the expected value on every clock cycle.
4. **Icarus Verilog & VVP:** A command-line Verilog compiler and simulator used to run the verification.
5. **GTKWave:** A waveform viewer used to visually inspect and debug the signals from the simulation.
6. **Visual Studio Code:** The central IDE for writing all Verilog and Python code.

3 Design Implementation (Verilog RTL)

The hardware was described in a single Verilog file, `counter.v`. The design is fully synchronous and synthesizable.

3.1 Module Definition and Ports

The module uses a `parameter` named `N` to define its bit width, with a default value of 8. This makes the design inherently scalable.

The ports are:

- `clk`: Clock input.
- `rst_n`: Active-low asynchronous reset input.
- `control[1:0]`: 2-bit input to select the operation.
- `parallel_in[N-1:0]`: N-bit parallel data input for the load operation.
- `count_out[N-1:0]`: N-bit registered output of the counter.

3.2 Core Logic

The core logic resides within an `always @(posedge clk or negedge rst_n)` block. The sensitivity list ensures the block is triggered by a rising clock edge (for synchronous operations) or a falling edge of the reset signal (for asynchronous reset).

A `case` statement is used to cleanly implement the four operations based on the `control` input. This is the standard and most readable way to implement such multiplexed logic.

3.3 RTL Code

```verilog
`timescale 1ns / 1ps

module counter #(
    parameter N = 8 // Default counter width is 8 bits
) (
    input wire clk,
    input wire rst_n, // Active-low asynchronous reset
    input wire [1:0] control,
    input wire [N-1:0] parallel_in,
    output reg [N-1:0] count_out
);

    // Sequential logic block
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            count_out <= {N{1'b0}}; // Reset counter to 0
        end else begin
            case (control)
                // 2'b00: Hold current state
                2'b00: count_out <= count_out;

                // 2'b01: Count up
                2'b01: count_out <= count_out + 1;

                // 2'b10: Count down
                2'b10: count_out <= count_out - 1;

                // 2'b11: Parallel Load
                2'b11: count_out <= parallel_in;

                // Good practice to prevent latches
                default: count_out <= count_out;
            endcase
        end
    end

endmodule
```

Listing 1: Verilog RTL for the N-bit counter (`counter.v`)

4 Verification Strategy and Implementation

The verification was driven by the Python script, which defines the correct behavior and provides the stimulus.

4.1 Python Golden Model and Test Vector Generation

The Python script `generate_vectors.py` contains a class, `CounterModel`, which perfectly mimics the intended behavior of the Verilog module. The script's main function generates a file, `test_vectors.txt`, containing hundreds of test cases. Each line in the file represents one test cycle and contains the control inputs, the parallel data input, and the golden "expected output" for that cycle.

```python
import random

# --- Configuration ---
N_BITS = 8
NUM_VECTORS = 100
FILENAME = "test_vectors.txt"

# --- Golden Model ---
class CounterModel:
    def __init__(self, n_bits):
        self.n_bits = n_bits
        self.max_val = (1 << n_bits) - 1
        self.current_count = 0

    def step(self, control, parallel_in):
        """Calculates the next state and returns that next state."""
        # Calculate the next state based on inputs
        if control == 0b00:
            next_count = self.current_count
        elif control == 0b01:
            next_count = (self.current_count + 1) & self.max_val
        elif control == 0b10:
            next_count = (self.current_count - 1) & self.max_val
        elif control == 0b11:
            next_count = parallel_in
        else:
            next_count = self.current_count

        # --- THE FIX IS HERE ---
        # The expected output is the state *after* the clock edge, which is next_count.
        expected_output = next_count

        # We still update the internal state for the next iteration.
        self.current_count = next_count

        return expected_output


def generate_test_vectors():
    """Generates a sequence of test vectors and writes them to a file."""
    model = CounterModel(n_bits=N_BITS)
    print(f"Generating {NUM_VECTORS} test vectors for an {N_BITS}-bit counter...")

    with open(FILENAME, 'w') as f:
        # We need to adjust the directed tests as well
        f.write(f"// Test vectors: control[1:0] parallel_in[N-1:0] expected_out[N-1:0]\n
    ")

        # After the reset, the initial state is 0.
        # The testbench checks AFTER the clock edge, so the expected value is the result
     of the operation.

        # 1. Test Load: Load 42. Expected output is 42.
        p_in = 42
        f.write(f"{0b11:02b} {p_in:0{N_BITS}b} {model.step(0b11, p_in):0{N_BITS}b}\n")
        # 2. Test Hold: State is 42. Hold. Expected output is 42.
        f.write(f"{0b00:02b} {0:0{N_BITS}b} {model.step(0b00, 0):0{N_BITS}b}\n")
        # 3. Test Count Up: State is 42. Count up. Expected output is 43.
        f.write(f"{0b01:02b} {0:0{N_BITS}b} {model.step(0b01, 0):0{N_BITS}b}\n")
        # 4. Test Count Down: State is 43. Count down. Expected output is 42.
        f.write(f"{0b10:02b} {0:0{N_BITS}b} {model.step(0b10, 0):0{N_BITS}b}\n")

        # Generate randomized test vectors
        for _ in range(NUM_VECTORS - 4):
            control = random.randint(0, 3)
            parallel_in = random.randint(0, (1 << N_BITS) - 1)
            expected_out = model.step(control, parallel_in)
            f.write(f"{control:02b} {parallel_in:0{N_BITS}b} {expected_out:0{N_BITS}b}\n
    ")

    print(f"Successfully created '{FILENAME}'")

if __name__ == "__main__":
    generate_test_vectors()
```

Listing 2: Python script for the golden model and test vector generation

4.2 Self-Checking Verilog Testbench

The testbench, `tb_counter.v`, is the core of the verification environment. Its main `initial` block performs the following sequence:

1. Opens and validates the `test_vectors.txt` file.
2. Initializes the waveform dump file (`.vcd`) for GTKWave.
3. Applies an initial reset to the Device Under Test (DUT).
4. Enters a loop that, on each clock cycle, reads a line from the text file using `$fscanf`, applies the inputs to the DUT, and compares the DUT's `count_out` with the `expected_out` read from the file.
5. If a mismatch occurs, it prints a detailed error message.
6. After all vectors are processed, it prints a final success or failure summary.

```verilog
`timescale 1ns / 1ps

module tb_counter;

    localparam N = 8;
    localparam CLK_PERIOD = 10;

    reg clk, rst_n;
    reg [1:0] control;
    reg [N-1:0] parallel_in;
    wire [N-1:0] count_out;

    integer file_handle, scan_status, error_count = 0, vector_count = 0;
    reg [N-1:0] expected_out;
    reg [1023:0] dummy_line;

    counter #(.N(N)) dut (.clk(clk), .rst_n(rst_n), .control(control), .parallel_in(
    parallel_in), .count_out(count_out));

    always #(CLK_PERIOD / 2) clk = ~clk;

    initial begin
        file_handle = $fopen("test_vectors.txt", "r");
        if (file_handle == 0) $finish;

        $dumpfile("counter_wave.vcd");
        $dumpvars(0, tb_counter);

        rst_n = 0; #(CLK_PERIOD * 2); rst_n = 1; #(CLK_PERIOD);

        scan_status = $fgets(dummy_line, file_handle); // Skip header

        while (!$feof(file_handle)) begin
            scan_status = $fscanf(file_handle, "%b %b %b\n", control, parallel_in,
    expected_out);
            @(posedge clk);
            #1; // Delay for checking after signal settles
            if (count_out !== expected_out) begin
                $display("ERROR @ time %0t: Vector %0d", $time, vector_count);
                error_count = error_count + 1;
            end
            vector_count = vector_count + 1;
        end

        if (error_count == 0) $display("------ Simulation SUCCESS ------");
        else $display("------ Simulation FAILED: %0d errors ------", error_count);

        $fclose(file_handle);
        $finish;
    end
endmodule
```

Listing 3: Verilog self-checking testbench (`tb_counter.v`)

5 Results and Analysis

The simulation was executed using the following commands after generating the test vectors:

```
iverilog -o counter_sim.vvp counter.v tb_counter.v
vvp counter_sim.vvp
```

The terminal output confirmed that all 100 test cases passed successfully.

```
VCD info: dumpfile counter_wave.vcd opened for output.
------ Starting Simulation ------
------ Simulation SUCCESS: All 100 tests passed! ------
tb_counter.v:93: $finish called at 1036000 (1ps)
```

Visual inspection of the waveform in GTKWave provided further confirmation. As shown in Figure 1, the `count_out` signal is a perfect overlay of the `expected_out` signal, proving the hardware's logical correctness.

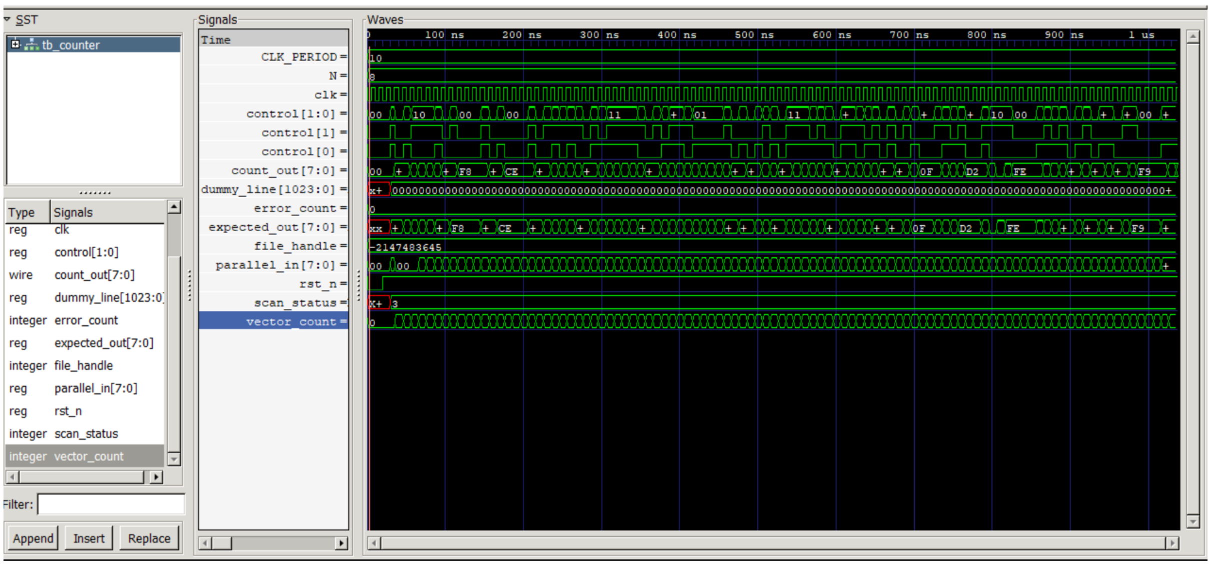

Figure 1: GTKWave output for the 8-bit counter simulation. Note the identical values of `count_out` and `expected_out`.

6 Scalability Demonstration (16-Bit Test)

To prove the scalability of the design, the counter width was changed from 8 to 16 bits. This required only two one-line changes:

1. In `generate_vectors.py`, `N_BITS` was changed to 16.
2. In `tb_counter.v`, `localparam N` was changed to 16.

The core RTL in `counter.v` remained untouched. The entire flow was re-run, and the simulation passed successfully for the 16-bit configuration, confirming the design's parametric nature.

7 Conclusion

This project successfully achieved its goal of designing and verifying a parametric N-bit universal synchronous counter. The adoption of a Python-driven, automated verification workflow proved to be highly effective, allowing for comprehensive testing with minimal manual effort. The final design is robust, scalable, and logically correct, serving as a reliable and reusable component for future, more complex digital systems.