Half Adder Module Test Plan and Results

Hongkuan Yu

Feb 2025

Contents

1 DUT

This module is simple, so I included the SystemVerilog code of `half_adder.sv` below.

```
    module half_adder(input clk,
                      input rst_n,
                      input [8:0] data_in0,
                      input [8:0] data_in1,
                      input in_valid,
                      output reg [9:0] data_out,
                      output reg out_valid
                      );

    always @(posedge clk) begin
        if(!rst_n) begin
            data_out    <= 9'h0;
            out_valid   <= 1'b0;
        end
        else if(in_valid) begin
            data_out    <= data_in0 + data_in1;
            out_valid   <= 1'b1;
        end
        else begin
            data_out    <= 9'h0;
            out_valid   <= 1'b0;
        end
    end

endmodule
```

This DUT is **a half adder**, meaning there is no carry-in for the Least Significant Bits. The I/O ports of `half_adder.sv` and their functions are shown in Table 1 below.

input (wire)	clk	1 bit	Global clock
input (wire)	rst_n	1 bit	Active-low reset
input (wire)	data_in0	9 bits	First data be added
input (wire)	data_in1	9 bits	Second data be added
input (wire)	in_valid	1 bit	Valid for input data
output (reg)	data_out	10 bits	Result of addition
output (reg)	out_valid	1 bit	Valid for output result

Table 1: I/O ports of `dut.sv` and their functions

The module's I/O graphic overview is shown in Figure 1 below.

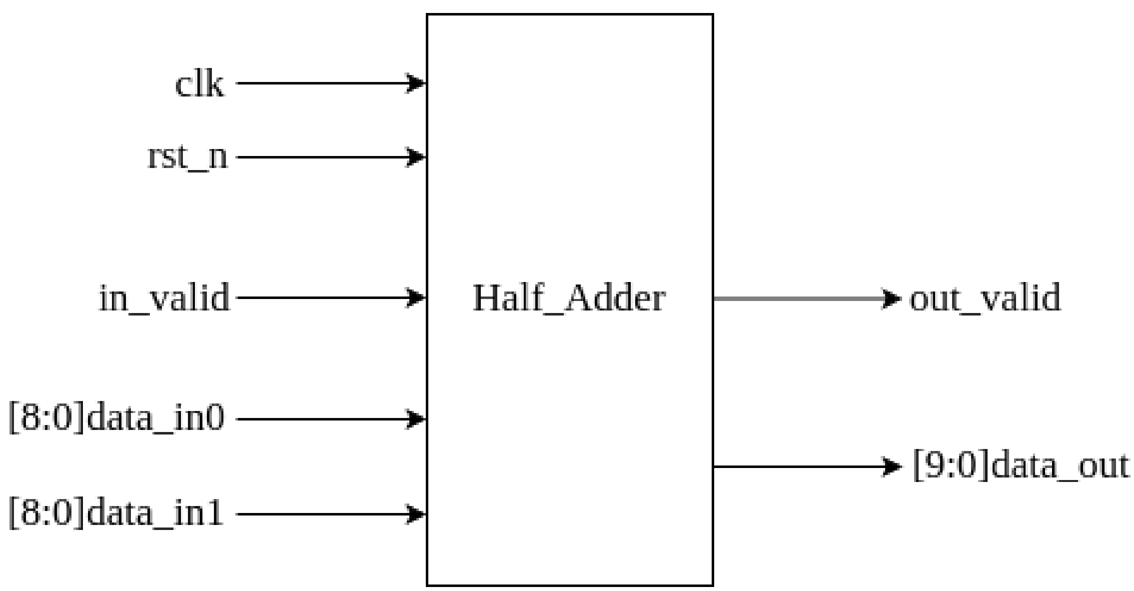

Figure 1: Half Adder Module's I/O Overview

2 Simulation Testbench (Dynamic)

Create `sim_tb` folder holding simulation testbench. Write a testbench `half_adder_tb.sv` to verify the `half_adder.sv` dynamically by Synopsys VCS.

Create `sim_tb_script` folder for necessary scripts. Create `filelist.f` and `Makefile` containing normal simulation commands.

Create `sim_tb_results` folder for simulation testbench results. The results of the simulation testbench meet expectations: the output signal `data_out` successfully gets the correct value of `data_in0` + `data_in1` after one clock period. `out_valid` also functions correctly after one clock period after `in_valid` being asserted. The result waveform is shown below in Figure 2 with decimal representation.

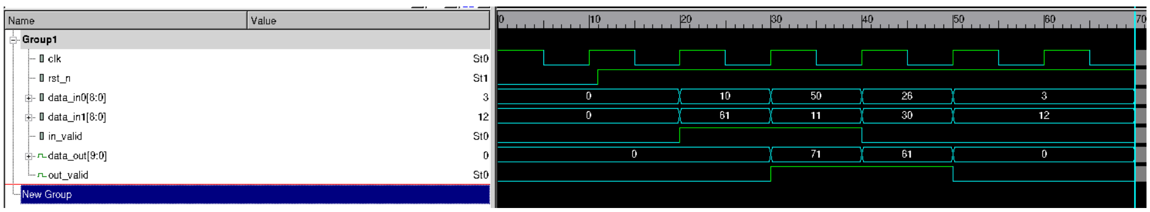

Figure 2: Simulation Testbench Result

3 Formal Property Verification (Static)

Create `sva` folder for formal property verification. Write a formal property verification `half_adder_sva.sv` to verify `half_adder.sv` statically.

Create `sva_script` folder for necessary scripts. Create `filelist.f` and `Makefile` containing operation commands of formal verification tool.

Create `sva_results` folder for formal property verification results.

All property passed. The result is shown below in Figure 3 and 4.

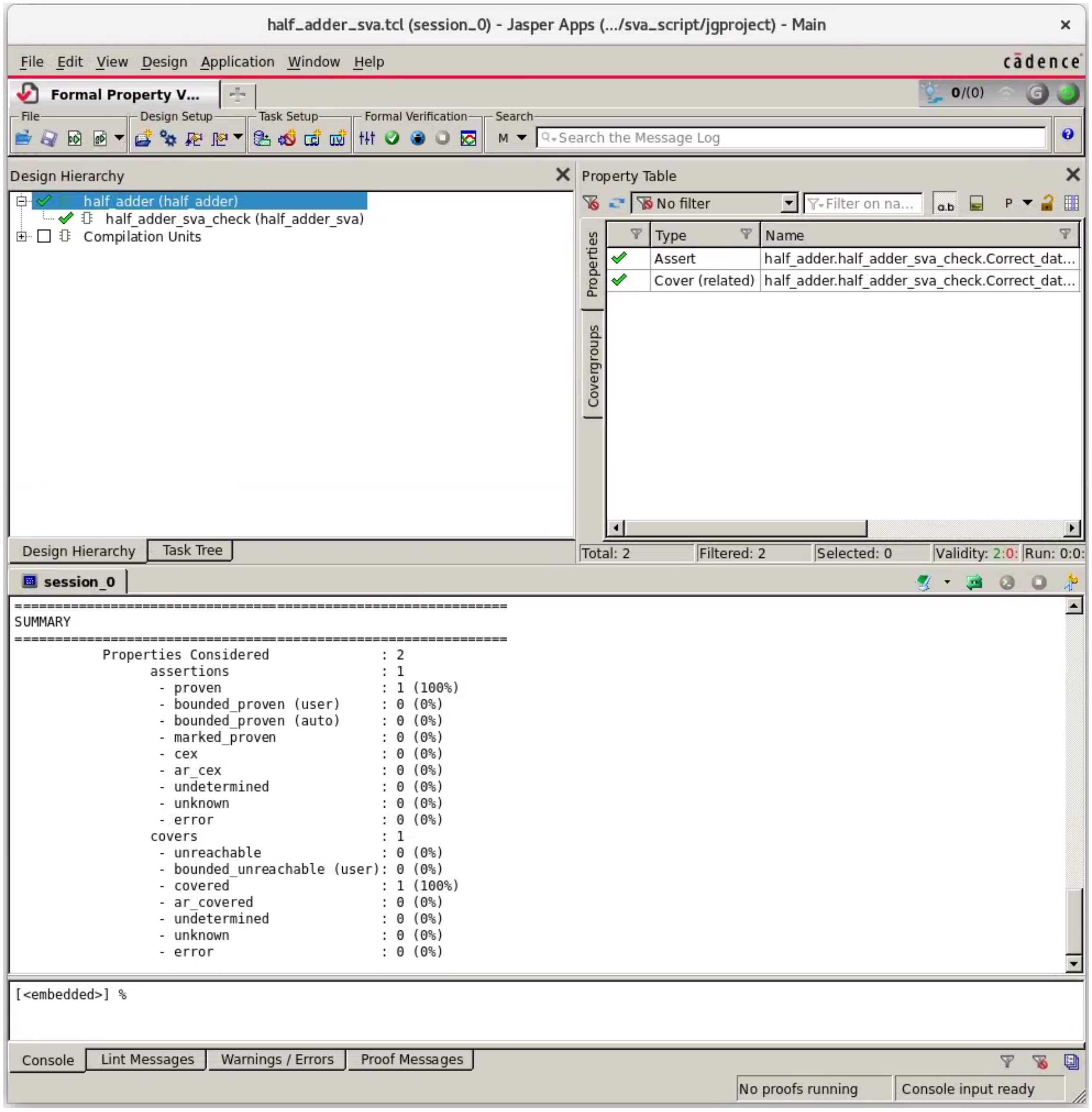

Figure 3: JasperGold FPV Result

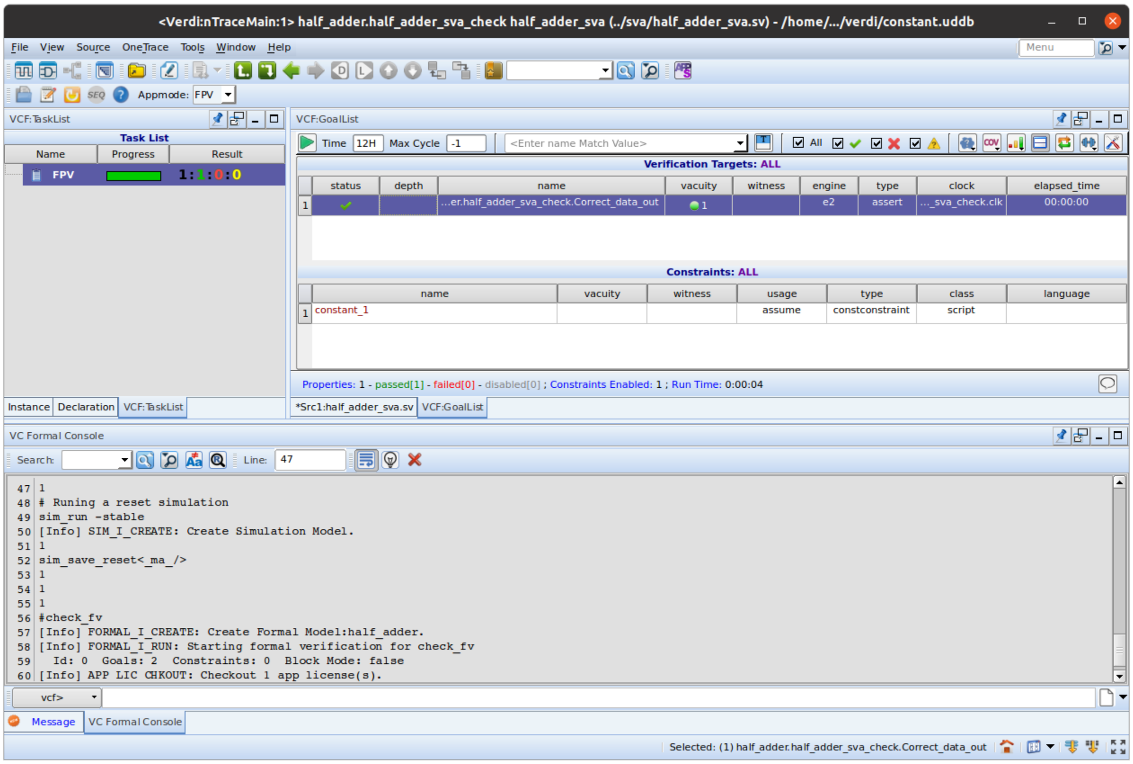

Figure 4: VC Formal FPV Result

4 UVM (1.1d)

Create `uvm` folder holding different UVM parts. `uvm/components` folder contains all UVM components; `uvm/interfaces` folder contains interfaces used in UVM; `uvm/test_cases` folder contains all test cases used in UVM.

The UVM components include (from top to bottom): a base test, environment, model, scoreboard, agent_in, agent_out, sequencer, transaction_in, transaction_out, driver, monitor_in, and monitor_out. Use a virtual sequencer to control the order of testing sequences. Six different test cases serve as six different sequences. Two interfaces are included for portable design. `top_tb.sv` connects them all.

Create `uvm_script` folder for necessary scripts. Create `filelist.f` and `Makefile` containing UVM simulation commands.

Create `uvm_results` folder for UVM results.

The structure of this UVM platform is shown in Figure 5 below. This platform does not contain the UVM register model.

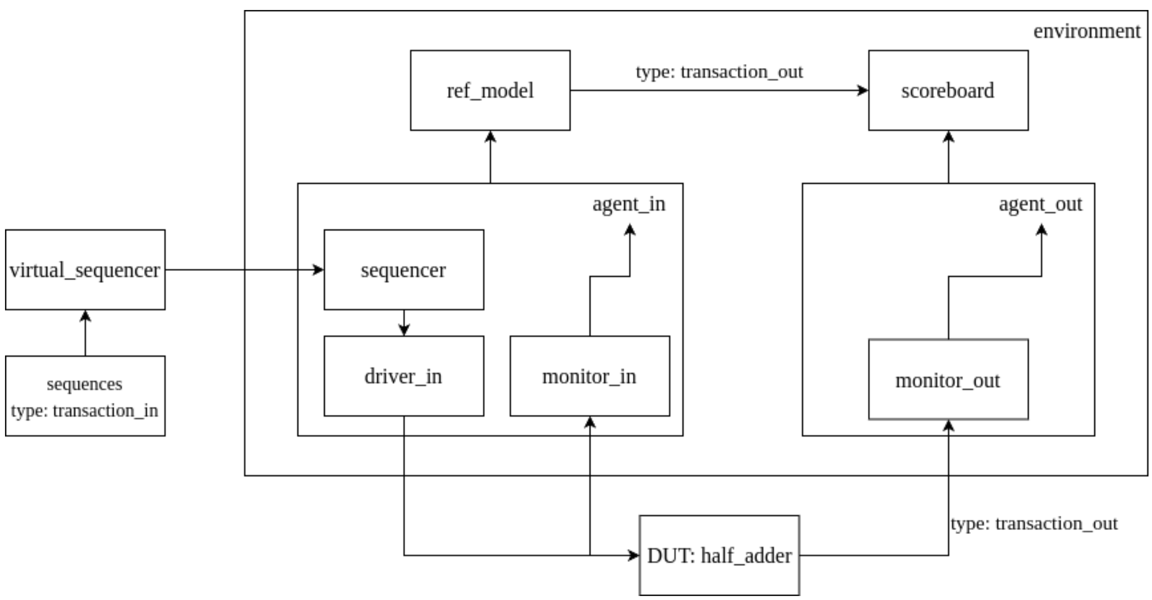

Figure 5: Structure of UVM Platform

This platform's UVM tree is shown in Figure 6 below.

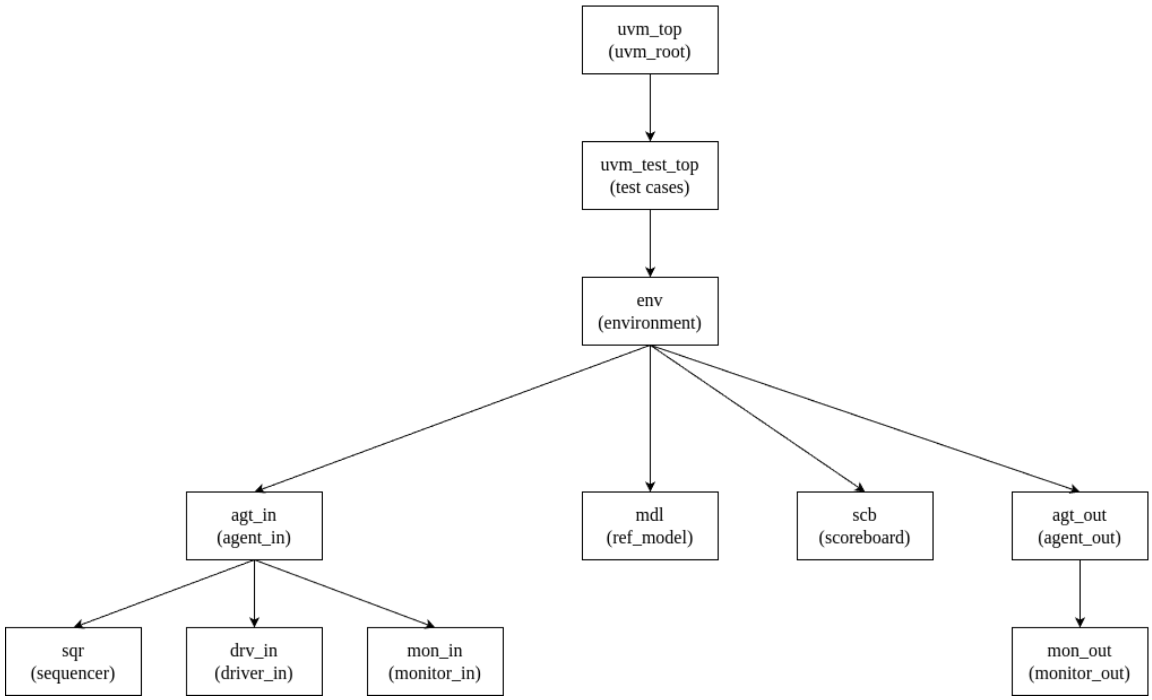

Figure 6: UVM Tree

The `transaction_in` contains unsigned `data0` and `data1` that can be randomized or set manually to feed into the input `data_in0` and `data_in1`. It also contains `ndelay` that can be randomized or set manually to activate `in_valid` with different delay clock cycles.

The `transaction_out` serves for the channel of `monitor_out` to the `data_out` data type.

Test cases contain random and ordered `transaction_in` sequences with different delay clock cycles, including burst (no delay), one cycle delay, and random cycle delay. All test cases passed. Please refer to the `.log` file respectively in the `uvm_results` folder.

References