Lab 1: Hardware Design and Verification

CPE 333 - S23

In this lab, you will verify and debug designs for several hardware components. The primary objective of this lab is to introduce you to hardware verification methodologies using SystemVerilog. You will learn how to verify and design the following hardware components: An eight-bit multiplier, a synchronous FIFO, a content addressable memory (CAM), and a cacheline adapter.

What is Verification

When designing a digital circuit in a hardware description language (HDL), we are attempting to describe a hardware component whose behavior will comply with a high level description of an intended behavior (i.e. a specification). Hardware verification is a process which attempts to ensure that a design's behavior matches a specified behavior.

Verification is Hard. Digital hardware verification is a hard [1] problem. For example, consider the collection of Boolean functions $B_n = \{f | f : \{0,1\}^n \rightarrow \{0,1\}\}$. These are the functions with n binary inputs and a binary output. How would you go about writing a program which takes as input an element of B_n (i.e. the specification of the function), and a SystemVerilog description of a digital circuit (i.e. the design), and then determine whether or not the design matches the specification? Can you come up with something significantly better than iterating through all 2^n possible function inputs and ensuring that the output of the design matches the output of the specification? [2]

[1] coNP-Complete

[2] If you can, please give CPE333 a shout out as you claim your $1M prize

Verification is Necessary. We all have experienced buggy software, which ultimately required significant effort for debugging. This is also true for hardware development. However, hardware verification can be even more challenging and expensive.
There are numerous reasons, including the following [3]:

- Fabrication cost are much higher for hardware than for software
- Hardware bug fixes after delivery to customers are challenging/impossible.
- Quality expectations are usually higher for hardware than for software. (e.g. human safety is a risk if the hardware does not work properly).

[3] Thomas Kropf, editor. *Formal Hardware Verification - Methods and Systems in Comparison*, volume 1287 of *Lecture Notes in Computer Science*, 1997. Springer. ISBN 3-540-63475-4. doi: 10.1007/3-540-63475-4. URL https://doi.org/10.1007/3-540-63475-4

- Time to market severely affects potential revenue.

VERIFICATION IS NOT VALIDATION. Validation is a similar but different process. While verification is a process to ensure that a design matches **its** specification, validation is a process that ensures that a design matches **a** specification.

Consider the case where a truck is designed to meet a specification of being able to haul 20 tons of material. The truck designers at ACME Truck Co. must *verify* that their trucks can haul 20 tons. Likewise, ACE Hauling Co. requires a truck which can haul 25 tons. Thus the engineers at ACE Hauling Co. must *validate* that the ACME Truck Co.'s truck can haul 25 tons.

HOW TO DO VERIFICATION. There are three main tasks to verification [4]:

- Stimulate a design by providing sequences of stimuli.
- Check that the design outputs results in accordance with the specification.
- Measure how much of a design's execution state space [5]

In this lab, you will complete these tasks using dynamic simulation [6]. You will use specifications to generate (sometimes random) sequences of input stimuli, create checkers which confirm that the output of the design under test (DUT) conforms to the specification, and record DUT accuracy and coverage (scoreboard).

A SIMPLE VERIFICATION EXAMPLE. To demonstrate dynamic simulation, we will use the simple combinational circuit shown in Listing 1. We will verify that the module *comfunction* actually implements the truth-table shown in the comments of Listing 1. A truth-table is an example of a specification which descrbies the intended behvario of the circuit [7].

```
1
2  // Module implements the following truth-table:
3  /*
4      abc || x
5      000 || 0
6      001 || 0
7      010 || 1
8      011 || 1
9      100 || 0
10     101 || 1
11     110 || 0
12     111 || 1
13  */
14 module combfunction
15 (
```

[4] Erik Seligman, Tom Schubert, and M V Achutha Kiran Kumar. *Formal Verification: An Essential Toolkit for Modern VLSI Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2015. ISBN 9780128008157

[5] The full space of all RTL state and input values.

[6] In dynamic simulation, the design is simulated using cycle or gate level simulators, stimuli consist of sequences of input signals to the device under test, and outputs are verified against the specification using assertions. This is in contrast to *formal verification* techniques which use mathematical representations of the design, along with assumptions about possible inputs and states, to constrain the test space. By partitioning the execution state space into reachable space and unreachable space, formal verification techniques often drastically reduce the size of the space that needs to be tested and then use automated techniques to prove properties about the circuit.

[7] In this case, the specification is a formal specification, written in a formal language. Often an initial specification will not be formalized so nicely.

```
16      input logic a_i,
17      input logic b_i,
18      input logic c_i,
19      output logic x_o
20  );
21
22  assign x_o = a_i ^ b_i ^ (a_i & c_i);
23
24  endmodule : combfuction
```

Listing 1: Simple Combinational Circuit

GENERATING THE STIMULUS. Combinational circuits have no initial or intermediate state, the size of the input and the output are fixed, and the runtime is constant. To verify the design, we can simply run through all possible inputs [8] and verify that the DUT generates the proper outputs, shown in Listing 2.

[8] in time exponential to the number of inputs

```
1   initial begin
2       for (int i = 0; i < 4'b1000; ++i) begin
3           {a_i, b_i, c_i} = i[2:0];
4           #1;
5       end
6   end
```

Listing 2: Generating the Stimulus

MODELING THE CORRECT BEHAVIOR. In addition to generating all possible inputs, we also must create a model for the circuits specified behavior. Listing 3 shows an example model. For a larger circuit with more inputs, we could also implement the model by loading a truth table into a memory indexed by the inputs. [9]

[9] Note that on line 11, we have a default case since logic encodes four-states. Thus if the input to the function is mistakenly x or z we can display an error showing our test bench is at fault, rather than the DUT.

```
1   function logic spec_output(logic a, logic b, logic c);
2       case ({a, b, c})
3           3'b000: return 0;
4           3'b001: return 0;
5           3'b011: return 1;
6           3'b010: return 1;
7           3'b110: return 0;
8           3'b100: return 1;
9           3'b101: return 0;
10          3'b111: return 1;
11          default: $error("Invalid input to spec_output function");
12      endcase
13  endfunction
```

Listing 3: Modeling the Correct Behavior

CHECKING THE OUTPUTS. Finally, we can use a for loop which generates the input stimuli to check that the output of the DUT matches the output of the model (Listing 4).

```
1   initial begin
2       for (int i = 0; i <= 4'b1000; ++i) begin
```

```
        {a_i, b_i, c_i} = i[2:0];
        #1;
        output_equiv: assert (x_o == spec_output(a_i, b_i, c_i))
                      else $error("{a,b,c}=%b, dut output: %b spec output: %b",
                                  {a_i,b_i,c_i},x_o,spec_output(a_i,b_i,c_i));
    end
    $finish;
end
```

Listing 4: Checking the Outputs

PUTTING THIS ALL TOGETHER, we can write our testbench to verify *combfunction* (Listing 5).

```
function logic spec_output(logic a, logic b, logic c);
    case ({a, b, c})
        3'b000: return 0;
        3'b001: return 0;
        3'b011: return 1;
        3'b010: return 1;
        3'b110: return 0;
        3'b100: return 1;
        3'b101: return 0;
        3'b111: return 1;
        default: $error("Invalid input to spec_output function");
    endcase
endfunction

module combfunction_tb;
    timeunit 1ns;
    timeprecision 1ns;

    logic a_i, b_i, c_i, x_o;

    combfunction dut(.*);

    initial begin
        reset = '1;
        // Generate sequence of inputs
        for (int i = 0; i <= 4'b1000; ++i) begin
            // Set input values to the dut, and let combinational logic settle
            {a_i, b_i, c_i} = i[2:0];
            #1;
            reset = '0;
            // Check dut output vs specification output
            output_equiv: assert (x_o == spec_output(a_i, b_i, c_i))
                          else $error("With {a, b, c}=%b, dut outputs: %b while spec outputs: %b",
                                      {a_i, b_i, c_i}, x_o, spec_output(a_i, b_i, c_i));
        end
        $finish;
    end
endmodule : combfunction_tb
```

Listing 5: Testbench for combfunction

VERIFYING A SEQUENTIAL CIRCUIT. When verifying a small circuit we can exhaust all possible inputs simply by iterating through each possible input combination. Consider a sequential circuit that takes an arbitrarily large input serially. Clearly, verifying the circuit by simply monitoring the input and outputs is insufficient, since the circuit can potentially process infinitely many different input "strings". Listing 6 shows an example serial circuit.

```
module div5 (
    input logic clk,
    input logic rst,
    input logic serial_in,
    input logic run,
    output logic decision
);

logic [2:0] state;
localparam logic [2:0] initial_state = '1;

always_ff @(posedge clk) begin
    if (rst) begin
        state <= initial_state;
    end
    else if (run) begin
        case (state)
            initial_state,
            3'b000: state <= serial_in ? 3'b001 : 3'b000;
            3'b001: state <= serial_in ? 3'b011 : 3'b010;
            3'b010: state <= serial_in ? 3'b000 : 3'b100;
            3'b011: state <= serial_in ? 3'b010 : 3'b001;
            3'b100: state <= serial_in ? 3'b100 : 3'b011;
        endcase
    end
    else begin
        state <= initial_state;
    end
end

assign decision = state == 3'b000;

endmodule : div5
```

Listing 6: A Sequential circuit with binary string input

Listing 6 is a SystemVerilog representation of a sequential circuit [10] which "decides" if the input string is divisible by five. On completion of the input processing, the output *decision* should go high. Similarly, if the input string is not divisible by five, then on finishing the processing of the input, the output *decision* should go low.

[10] specifically a Deterministic Finite Automaton (DFA)

Since there is no limit on how long input strings can be, if we test the functionality by looking only at inputs and outputs of the design module, then we can only give guarantees qualified by a certain input size (e.g. "all inputs of less than 16-bits produced the correct outputs"). Luckily, we can verify whether this design is functionally correct without qualifications. Rather than specifying that the design

produces a certain output signal based on the sequence of input signals, we instead specify that the design implements a specific DFA which we can prove decides the DIV5 problem (language). Therefore we only need to verify that the design implements the DFA.

The DFA that we implement has six states, five of which are labeled 0 through 4, which represent the value of the in-process inptu string mod 5. The sixth state is the initial state, labeled s. The next state transition function, δ, which takes the current state i and the input bit b as follows: $\delta(i, b) = (2i + b) mod\ 5$ if $i \in \{0, 1, 2, 3, 4\}$ else b

An input string is divisible by five iff the DFA moves to state 0 upon processing the last (lsb) bit in the string. We consider the DFA to consume its input string from left-to-right (i.e. msb first). [11]

Therefore to verify the design, we must move the design into every possible state it can enter, and then ensure the transitions from these states are correct. [12]

General Testbench Components. In the prior examples, the verification steps of input stimulus generation, driving the DUT and mode, and comparing the results were implemented "in-place" using basic modules. As we scale the complexity of the designs, you may use a universal verification model (UVM), where we seperate functionality of the verification process into multiple indepdent parts:

- A *Sequencer* generates input stimuli, independent of the bus or interface used by the DUT.
- A *Driver* generates the bus or interface control input stimuli and transfers the sequencer's data to the DUT.
- A *Monitor* acts as a complement of the driver. The monitor observes and collects the inptus and outputs of the DUT to identify when a transaction is complete and ready to be evaluated by the scoreboard.
- A *Scoreboard* consumes the output of the monitor and evaluates whether the DUT produced the appropriate value. The scoreboard may also measure testing coverage.

In this lab, you will see designs which you will be able to fully cover (similar to *combfunction* above) and desigsn whose execution state space is too large to fully cover.

[11] Given a binary w string, after every symbol is read, the DFA state is $k = w \bmod 5$, where w is interpreted as a binary number

Proof: Let w be an arbitrary binary string. Assume for every string x, s.t. $1 \leq |x| \leq |w|$, the DFA state is $k = x \bmod 5$.

- Suppose $w = 0$. Then the DFA state 0 = w mod 5.
- Suppose $w = 1$. Then the DFA state 1 = w mod 5.
- Suppose $w = \{x, 0\}$, for some binary string x, s.t. $|x||w|$. By the inductive assumption, the DFA is in state $k = x \bmod 5$. After processing $w = \{x, 0\}$ from left-to-right, the DFA's next state is:

$$\begin{aligned} \delta(k, 0) &= (2k + 0) \bmod 5 \\ &= (2(x \bmod 5)) \bmod 5 \\ &= 2x \bmod 5 \\ &= w \bmod 5 \end{aligned} \tag{1}$$

- Suppose $w = \{x, 1\}$, for some binary string x, s.t. $|x||w|$. By the inductive assumption, the DFA is in state $k = x \bmod 5$. After processing $w = \{x, 1\}$ from left-to-right, the DFA's next state is:

$$\begin{aligned} \delta(k, 1) &= (2k + 1) \bmod 5 \\ &= (2(x \bmod 5 + 1)) \bmod 5 \\ &= (2x + 1) \bmod 5 \\ &= w \bmod 5 \end{aligned} \tag{2}$$

Therefore the DFA state (k) is w mod 5 for any non-empty binary string w.

[12] These combinations of internal design state and input signals are called "coverage points" (or "coverages" of "covers').

Verifying a Multiplier

In this part of the lab, you will verify a multiplier (i.e. an unsigned add-shift multiplier).

SPECIFICATION. The multiplier has the following port listing:

- **clk_i** is the clock which drives the sequential logic in the multiplier
- reset_n_i is an active low, synchronous reset signal. If this signal is asserted at @(posedge clk_i), the multiplier should halt any ongoing multiplication and reset its state to allow for the start of a new multiplication.
- **multiplicand_i** and multiplier_i are the input operands for the multiplication. When a multiplication begins, these signals are registered in the multiplier and thus are not required to be continuously asserted throughout the multiplication.
- **start_i** begins a new multiplication if it is asserted at @(posedge clk_i) and the multiplier is in a 'ready' state. If the multiplier is not in a 'ready' state, assertion of this signal has no effect.
- **ready_o** asserts that the multiplier is in a 'ready' state and can begin a new multiplication. product_o contains the 2 * width_p bit output of the multiplication when the multiplier is in a 'done' state.
- **done_o** is asserted when the multiplier is in a 'done' state. This occurs when multiplication is complete, meaning (product_o contains the product of the registered input operands OR a synchronous reset has occurred), AND a new multiplication has not been started.

Figure 1 shows the timing diagram for this protocol. The number of cycles that the multiplier takes to complete the multiplication is not specified.

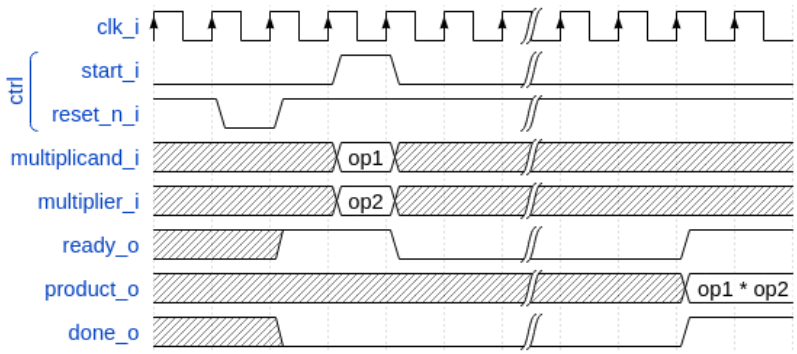

Figure 1: Multiplier Timing Diagram

COVERAGES. Your testbench must cover at least the following:

- From a 'ready' state (see types.sv), assert start_i with every possible combination of multiplicand and multiplier, and without any resets until the multiplier enters a 'done' state (resets while the device is in a 'done' state are acceptable).
- For each 'run' state s, assert the start_i signal while the multiplier is in state s.
- For each 'run' state s, assert the active-low reset_n_i signal while the multiplier is in state s.

ERROR REPORTING. Your testbench must detect the following errors (defined in types.sv):

- Upon entering the 'DONE' state, if the output signal product_o holds an incorrect product, report a BAD_PRODUCT error.
- If the ready_o signal is not asserted after a reset, report a NOT_READY error.
- If the ready_o signal is not asserted upon completion of a multiplication, report a NOT_READY error.
- To report an error, pass the appropriate error type to report_error task defined in testbench.sv. An example is given below.

```
assert (/* your assertion here */)
  else begin
    $error ("%0d: %0t: BAD_PRODUCT error detected", `__LINE__, $time);
    report_error (BAD_PRODUCT);
  end
```

Listing 7: example for reporting error

CLOCKING BLOCKS. In SystemVerilog, clocking blocks are an abstraction used to capture precise timing information and allow the verification engineer to write verification code at the 'cycle' level. The clocking blocks allow you to specify input and output skews, but in this lab you only use them to specify clocks. When using a default clocking construct, signals should be assigned using non-blocking assignments. Further, you can insert a delay of N cycles using the syntax (N). To delay until some condition holds, use the 'if and only if' keyword: @(<clk> iff <conditon>);.

CLOCKING BLOCKS. In order to facilitate checking of your testbench (i.e. autograding), your testbench should set signal values only at time 0 (the beginning of an initial block) or using the tb_clk clock as described in the Clocking Blocks section above. Additionally, at time 0, your testbench must assert the reset_n_i signal.

SAMPLING SIGNALS. All time delaying constructs should be associated with this default clock. That is, they should either be of the form ##(n), which waits for n cycles with respect to the clocking block, or @(tb_clk [iff <predicate>]) which delays for a single cycle, or delays until 'predicate' is evaluated true with samples taken with respect to the clocking block.

For example, the following are appropriate procedural blocks for your testbench:

```
initial reset_n = 0;   // initialize reset signal
initial begin
    ##(5);               // Ensure DUT is reset
    reset_n <= 1;
    multiplicand_i <= 16;
    multiplier_i <= 32;  // NBA: signals still have their initial values
    @(tb_clk);           // Wait for clock signal (could use '##(1)')
                         // Now, when the values get assigned
end

always @() begin
    $display("SystemVerilog Functions cannot block");
end
```

Listing 8: Example of appropriate procedural blocks for your testbench

```
initial begin
    reset_n_i = 1'b1;    // reset not initialized to active low 0
    @(posedge clk);      // Using clk rather than tb_clk
    multiplier_i = 32;   // signal value set at rising edge of clock
end

always @(negedge clk) begin
    reset_n_i = 1'b0;
    multiplicand_i = 16;
    multiplier_i = 32;  // Only use NON Blocking Assignments
                        // with a clocking block
    @(tb_clk);
end
```

Listing 9: Example of inappropriate procedural blocks

Verifying a FIFO

In this part of the lab, you will write a testbench to verify a synchronous FIFO with a single enqueuer and a single dequeuer. A FIFO is called 'synchronous' when the enqueue clock and the dequeue clock are the same. [13] The FIFO is described in fifo.sv. In this exercise, you will design a test bench to verify this design.

[13] If the clocks are distinct, then it is an asynchrnous FIFO, and much more complicated

SPECIFICATION The FIFO implements a valid-ready enqueue protocol, and a valid-yumi dequeue protocol, and has the following port listing:

- clk_i is the clock which drives the sequential logic in the fifo.
- reset_n_i is an active low, synchronous reset signal. If this signal is asserted at @(posedge clk_i), the FIFO sets itself to 'empty'.
- The valid-ready protocol is:
  - data_i contains the enqueued data word.
  - valid_i is asserted by the enqueuer to enqueue data_i into the FIFO.
  - ready_o asserts that the FIFO is not full and has capacity to enqueue a word. The behavior when item valid_i is asserted while the FIFO is full is undefined and should be avoided.
- The valid-yumi protocol is:
  - valid_o asserts that the FIFO is not empty and that the value on data_o is the oldest word stored in the FIFO.
  - yumi_i is asserted by the dequeuer to signal to the FIFO that the word in data_o must be removed from the FIFO.

  Figure 2 shows a timing diagram for the fifo.

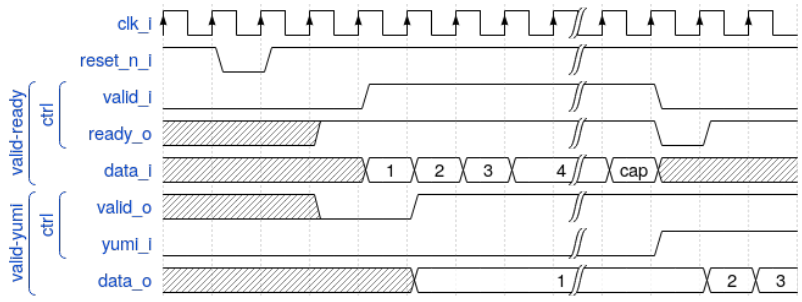

Figure 2: FIFO Timing Diagram

COVERAGES. Your testbench must cover at least the following for the FIFO with capacity cap_p:

- You must enqueue words while the FIFO has size in [0, cap_p-1].
- You must dequeue words while the FIFO has size in [1, cap_p].
- You must simultaneously enqueue and dequeue while the FIFO has size in [1, cap_p-1].

ERROR REPORTING. Your testbench must detect the following errors (defined in types.sv):

- Asserting reset_n_i at @(tb_clk) should result in ready_o being high at @(posedge clk_i). If it is not, report the appropriate error.
- When asserting yumi_i at @(tb_clk) when data is ready, the value on data_o is the CORRECT value. If not, report the appropriate error. Recall that asserting yumi_i when the FIFO is empty results in undefined behavior, so avoid doing this.

To report an error, pass the appropriate error type to report_error task defined in testbench.sv. An example is given below.

```
assert (/* your assertion here */)
  else begin
    $error ("%0d: %0t: %s error detected", `__LINE__, $time, err.name);
    report_error (err);
  end
```

Listing 10: Example error reporting

Verifying a CAM

In this part of the lab, you will write a testbench to verify a content addressable memory (CAM). A CAM is similar to an 'associative array' abstract data type in software, with the distinction that a CAM is of fixed size. A CAM, then, is a collection of key-value pairs, and supports read and write operations. When reading a CAM, a key is provided, and the CAM responds with the appropriate value, or a signal indicating that there is no value associated with the key in the CAM. On a write, both a key and a value are provided, and these get stored into the CAM.

Since a CAM has a fixed number of entries (eight - for this lab), some type of 'replacement policy' must be used when writing a new key to a full CAM. The replacement policy used by the CAM in this lab is the 'least recently used' (LRU) policy, which evicts (removes) the entry whose key was least recently used by a read or write. On writes, a CAM takes different actions depending on whether the key is already present in the CAM, and whether the CAM is full.

- If the key is present in the CAM, the value associated with the key is updated.
- If the key is not present and the CAM is not full, then a new entry is allocated and both the key and value are stored into this new entry.
- If the key is not present and the CAM is full, then an entry is evicted, meaning the new key and value are written in the location of the previous entry.

In all write cases, metadata associated with the replacement policy is updated.

Specification. The CAM has the following port listing:

- **clk_i** is the clock which drives the sequential logic in the CAM.
- **reset_n_i** is an active low, synchronous reset signal. If this signal is asserted at @(posedge clk_i), the CAM resets itself to 'empty'.
- **rw_n_i** decides whether the operation is a read (if set to 1'b1) or a write (if set to 1'b0). This value has no effect on the CAM unless valid_i is asserted.
- **valid_i** is asserted when a read or write operation is performed.
- **key_i** is the key input used by both read and write operations.
- **val_i** is the value input used by write operations.
- **val_o** is the output value on reads.

- **valid_o** is asserted by the CAM on reads to assert that the value in val_o is correct (that is, the CAM found a value associated with key_i).

Write and read operations are serviced at the rising edge of clk_i. That is, the CAM updates its internal state (both key-value pairs as well as LRU metadata) sequentially. Additionally, the CAM guarantees that val_o and valid_o show the correct value on a read at the rising edge of clk_i.

Coverages. Your testbench must cover at least the following:

- The CAM must evict a key-value pair from each of its eight indices.
- The CAM must record a 'read-hit from each of its eight indices.
- You must perform writes of different values to the same key on consecutive clock cycles.
- You must perform a write then a read to the same key on consecutive clock cycles.

Error Reporting Your testbench must detect the following errors:

Assert a read error when the value read from the CAM is incorrect.

To report an error, pass the appropriate error type to itf.tb_report_dut_error task defined in cam_itf.sv. An example is given below:

```
@(clk);
assert (itf.val_o == val) else  begin
    itf.tb_report_dut_error(READ_ERROR);
    $error("%0t TB: Read %0d, expected %0d", $time, itf.val_o, val);
end
```

Listing 11: Example error reporting

References

Thomas Kropf, editor. *Formal Hardware Verification - Methods and Systems in Comparison*, volume 1287 of *Lecture Notes in Computer Science*, 1997. Springer. ISBN 3-540-63475-4. DOI: 10.1007/3-540-63475-4. URL https://doi.org/10.1007/3-540-63475-4.

Erik Seligman, Tom Schubert, and M V Achutha Kiran Kumar. *Formal Verification: An Essential Toolkit for Modern VLSI Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2015. ISBN 9780128008157.