Behavioral Verilog modeling at register-transfer level (RTL)

Christian Krieg

October 24, 2023

Welcome to this year's *Digital Integrated Circuits* course! Throughout this course, we will have to solve some assignments that will teach us fundamental concepts of integrated circuit design.

We will demonstrate digital integrated circuit design on a step-by-step simple example of a digital counter. As the name of this course suggests, our ultimate goal is to implement this counter as a digital integrated circuit. In the lab, we map this counter to a field-programmable gate array (FPGA) architecture, and run it on the board shown in Figure 1. The counter's functionality is executed on the FPGA. We will be able to control execution by pushing the buttons, and to visualize the counter value using the light-emitting diodes (LEDs). Because operating a hardware counter with buttons and LEDs may be inconvenient, we also will interface the hardware counter over a universal asynchronous receiver/transmitter (UART), so that we may be able to further process the counter value on a host computer. But this will all happen during the lab week at the end of the exercises.

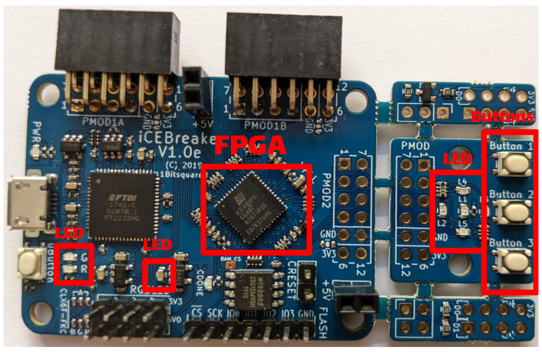

Figure 1: The *icebreaker* board, which will run our counter

Before we can bring our counter on hardware, it is important that we learn how to design digital circuits using state-of-the-art (and beyond) methods and tools, and to get a feeling for the results of our design decisions. We will draw attention on the results of the synthesis steps, which we will verify with several tools along the digital design flow.

Generally speaking, we perform the following steps:

1. Specify the counter's behavior
2. **Model (implement) the counter in a hardware description language (HDL)**
3. **Verify the counter's implementation**
4. Synthesize and optimize the counter's HDL model + verify
5. Map the counter to the target (FPGA) technology + verify
6. Generate an FPGA bitstream + verify
7. Configure the FPGA with the bitstream + verify
8. Run the counter on hardware

Our overall task is to build a synchronous counter that runs in two modes, based on a *mode* switch. The two modes are called *up* and *down*. In *up* mode, the counter increments the counter value by **4** at the rising edge of the clock signal, and in *down* mode, it decrements the counter value by **10** at the rising edge of the clock signal. The counter should implement synchronous reset behavior, such that it is set to an initial value of **17** upon reset. The counter should never exceed a value of **269**, and should never go below a value of **-263** (the counter value sticks near the minimum/maximum value, until the counting direction changes). The counter value should never be **-47** (the counter should jump over this value by one increment/decrement).

Wrapping up formal verification

In the previous tasks, we learned the basics of simulation (including testbenches), and formal verification. We learned that simulating a design under test (DUT) using a testbench may be more intuitive at the first glimpse, while at the second glimpse, formal verification comes with the huge benefit that we do not have to care about input signals – a model checker cares about them for us. In case we did not come up with a working solution for the previous task, we provide one in Listing 1.

Listing 1: A complete set of assertions that verify the properties of our counter (*tb_counter.sv*)

```
module tb_counter (
   input clk,
   input rst,
   input mode,
   output signed[9:0] cnt
);
   // Instantiate DUT
   counter dut(
      .clk(clk),
      .rst(rst),
      .mode(mode),
      .cnt(cnt)
   );
   // Connect DUT with assertions
   bind dut assertions tb (.*);
endmodule


module assertions (
   input clk,
   input rst,
   input mode,
   input signed[9:0] cnt
);
   // Generate reset sequence
   reg init = 1;
   always @(posedge clk) begin
      if (init) assume(rst);
      else assume(!rst);
      init <= 0;
   end
   // Check properties
   always @(posedge clk) begin
      if (rst) begin
         assert (cnt == 17);
      end
      if (!rst) begin
         //
         // Check if counter value is never lower than MIN, larger than MAX, or
         // equal to INV
         //
         assert (cnt <= 269);
         assert (cnt >= -263);
         assert (cnt != -47);
         //
         // Check if the counter value is correctly incremented and decremented
         //
         // Counting up
         if ($past(mode)) begin
            if (!$past(rst)) begin
               if ($past(cnt) == (-47-4)) // INV - INC, counter jumped over INV
                  assert ((cnt - $past(cnt)) == (4+4));
               else if ($past(cnt) <= 269 && $past(cnt) > (269-4)) // cnt was near MAX
                  assert ((cnt - $past(cnt)) == 0);
               else
                  assert ((cnt - $past(cnt)) == 4);
            end
         end
         // Counting down
         if (!$past(mode)) begin
            if (!$past(rst)) begin
               if ($past(cnt) == (-47+10)) // INV + DEC, counter jumped over INV
                  assert (($past(cnt) - cnt) == (10+10));
               else if ($past(cnt) >= -263 && $past(cnt) < (-263+10)) // cnt was near MIN
                  assert (($past(cnt) - cnt) == 0);
               else
                  assert (($past(cnt) - cnt) == 10);
            end
         end
      end
   end
endmodule
```

RTL modeling in Verilog

In this task, we finally **implement** our counter using Verilog.

Using Verilog comes with various benefits: First, Verilog is natively supported by Yosys, which means that we may use the open-source version, rather than the (partly proprietary) Tabby CAD suite which comes with VHDL and SystemVerilog support. In case we use the open-source version of Yosys, we must include the assertions into the same Verilog file in which we implement our counter, because the open-source version of does not understand the SystemVerilog *bind* operator. This is what we do in Listing 2. While it is perfectly fine to keep the assertions in a separate file *tb_counter.sv*, we show in Listing 2 how to make use of *SymbiYosys* when using the open-source version of *Yosys*.

Listing 2 shows a skeleton Verilog file where we first add our (existing) properties. Now, our task is to create a behavioral model of our counter in Verilog. Extend the code that is given in Listing 2. Remember, in behavioral modeling, we specify the algorithmic behavior of our system, rather than designing the boolean network and connecting its inputs and outputs. In other words, we use arithmetic operators (such as '+', '-', etc.) to implement the counter.

We do not use multipliers in our design. Multipliers are expensive in terms of area, so we carefully check whether we may model the counter's behavior without them.

Listing 2: Counter in Verilog (*counter.v*)

```
module counter (clk, rst, mode, cnt);

   // TODO: Put your behavioral code here



   // Let's specify the counter's intended behavior here
   `ifdef FORMAL
      reg init = 1;
      always @(posedge clk) begin
         if (init) assume(rst);
         else assume(!rst);
         init <= 0;
      end

      always @(posedge clk) begin

         if (rst) begin
            assert (cnt == 10'sd17);
         end

         if (!rst) begin

            //
            // Check if counter value is never lower than MIN, larger than MAX, or
            // equal to INV
            //
            assert (cnt <= 10'sd269);
            assert (cnt >= -10'sd263);
            assert (cnt != -10'sd47);

            //
            // Check if the counter value is correctly incremented and decremented
            //

            // Counting up
            if ($past(mode)) begin
               if (!$past(rst)) begin
                  if ($past(cnt) == -10'sd51) // INV - INC, counter jumped over INV
                     assert ((cnt - $past(cnt)) == 10'sd8);
                  else if ($past(cnt) <= 10'sd269 && $past(cnt) > 10'sd265) // cnt was near MAX
                     assert ((cnt - $past(cnt)) == 10'sd0);
                  else
                     assert ((cnt - $past(cnt)) == 10'sd4);
               end
            end

            // Counting down
            if (!$past(mode)) begin
               if (!$past(rst)) begin
                  if ($past(cnt) == -10'sd37) // INV + DEC, counter jumped over INV
                     assert (($past(cnt) - cnt) == 10'sd20);
                  else if ($past(cnt) >= -10'sd263 && $past(cnt) < -10'sd253) // cnt was near MIN
                     assert (($past(cnt) - cnt) == 10'sd0);
                  else
                     assert (($past(cnt) - cnt) == 10'sd10);
               end
            end

         end
      end
   `endif
endmodule
```

We use the properties of the previous task to verify that we implement the counter's behavior as specified. Verilog is not strictly typed, which may lead to undesired side effects. By default, Verilog assumes a bit width of 32 bits for signals and constants. To be sure everything works as expected, we explicitly specify the number format for every constant in Listing 2, which apparently is 10-bit signed decimal.

To check our counter's correctness, we run SymbiYosys as given in Listing 3, along with the configuration file as given in Listing 4

Listing 3: Command line to verify our counter

```
sby -f counter.sby
```

Listing 4: Configuration file for SymbiYosys (*counter.sby*)

```
[options]
mode prove
expect pass

[engines]
aiger suprove

[script]
read_verilog -sv -formal counter.v
prep -top counter

[files]
counter.v
```

RTL synthesis

Once our counter's implementation conforms to its specification, it is time to synthesize our register transfer level (RTL) description into a netlist. We use *Yosys* to create the high-level netlist (in particular, the RTL netlist).

Listing 5 shows a basic synthesis script to perform behavioral synthesis with *Yosys*. In Line 1, the Verilog description of our counter is parsed into *Yosys*'s intermediate representation. In Line 2, the hierarchy of the counter's design is elaborated. The *hierarchy* command looks which modules are instantiated by which other modules, and creates a hierarchy tree. The *-auto-top* option identifies the top-level entity, which in our case is the only module in the design: the *counter* module. The *proc* command given in In Line 3 synthesizes the design's processes (defined in *always* blocks). The *clean* command given in Line 4 removes unused cells and wires.

We may interactively issue all the commands from Listing 5 into the *Yosys* shell step-by-step, and quickly have a look at the resulting netlist by using the *show* command. Let's try it out! We start *Yosys* by typing `yosys` (all lower-case) into the command line.

Listing 5: *Yosys* script to synthesize our counter (*synth.ys*)

```
1 read_verilog counter.v
2 hierarchy -check -auto-top
3 proc
4 clean
```

Besides providing the interactive shell, *Yosys* also supports a batch mode. We simply enter the command given in Listing 6 to execute the *Yosys* script given in Listing 5.

Listing 6: Command to invoke (*synth.ys*) in batch mode

```
yosys -s synth.ys
```

Resource analysis

Now, let us have a look at the resources which are consumed by your counter, by adding the *stat* command to the script given in Listing 5. *stat*'s output may look like the output given in Listing 7. It basically shows the types and numbers of instantiated cells in the synthesized netlist, as well as wires to connect these cells. It may be interesting that the output in Listing 7 shows two instantiations of an adder cell in Line 13, as well as two instantiations of an subtractor cell in Line 18. Let's take some time to think about this; why may there be two adders, why two subtractors?

Listing 7: Example output of the *stat* command

```
1 7. Printing statistics.
2
3 === counter ===
4
5   Number of wires: 24
```

```
Number of wire bits: 143
Number of public wires: 4
Number of public wire bits: 13
Number of memories: 0
Number of memory bits: 0
Number of processes: 0
Number of cells: 19
  $add 2
  $dff 1
  $le 4
  $mux 8
  $ne 2
  $sub 2
```

Coding style matters – Your task

To answer the above question, we primarily may focus on getting a correctly working implementation, without having any resource optimization in mind. In the general case, we may have two additions, and two subtractions because we most-likely implement one process in which the counter register is updated based on it's value in the previous clock cycle. This is perfectly okay for this simple use case; however, resource consumption is directly correlated with the coding style we use for our implementation.

To demonstrate this issue, our task is to implement the counter in Verilog with **just one adder**, and **zero subtractors**. Hint: have a look into what happens when we separate combinational logic from sequential logic.

We synthesize our design with *Yosys* as given in Listing 5 and use *stat* and *show* to investigate our design. *stat* should output something similar to Listing 8. As we can see, there is only one adder used in this design, and no subtractor.

Listing 8: Resources consumed by our Verilog counter

```
Number of wires: 14
Number of wire bits: 95
Number of public wires: 5
Number of public wire bits: 23
Number of memories: 0
Number of memory bits: 0
Number of processes: 0
Number of cells: 11
  $add 1
  $ge 2
  $le 2
  $mux 3
  $ne 2
  $sdff 1
```

This is a remarkable result: By optimizing our coding style only, we may control how many resources are instantiated in the RTL netlist. Even in this simple example, we reduced the number of adders/subtractors from (potentially) four to one. While this might not seem critical for this simple example, it however can have huge impact in real-world designs. Please remember: Always create clean and simple code in order to enable code optimization.

Submission details

Attach your extended *counter.v* to your submission e-mail.