Exercises: Digital Design with Verilog

CS-173 Fundamentals of Digital Systems

28th March 2024

Mirjana Stojilović
Parallel Systems Architecture Lab (PARSA)

Part I: Digital Design with Verilog

[Exercise 1] Logic Circuits and Verilog: Gate-Level Modeling

Consider the logic circuit shown in Figure 1. The circuit has three inputs a, b, c, one output f, and one intermediate wire p (shown here with a Logisim probe).

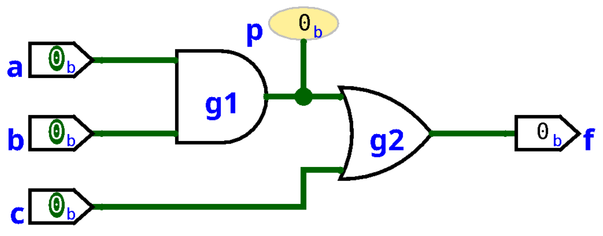

Figure 1: A simple combinational circuit with three inputs and one output.

Listing 1.1 shows one way of modeling the same circuit in Verilog structurally, at the gate level. Listing 1.2 shows an example Verilog testbench for simulating the operation of the circuit.

Listing 1.1: A structural Verilog description of the circuit in Figure 1.

```verilog
module structural_example (
  // Three input signals.
  input  a,
  input  b,
  input  c,
  // One output signal.
  output f
);

  // One intermediate wire.
  wire p;

  // Two gates specifying name, output, and inputs.
  and g1 (p, a, b);
  or  g2 (f, p, c);

endmodule
```

Listing 1.2: Verilog testbench for the circuit in Figure 1.

```verilog
module test_structural_example;

  // Define three inputs that can procedurally
  // be assigned values.
  reg  a, b, c;
  // And one output that responds to them.
  wire f;

  // Connect them to an instance of the module being tested.
  structural_example ex (.a(a), .b(b), .c(c), .f(f));

  // Loop variable.
  integer i;

  initial begin
    // Write this test's data to a .vcd file
    // that GTKWave can read.
    $dumpfile ("structural_example.vcd");
    $dumpvars;

    // Print values whenever they change.
    $monitor ("Time %2t, a=%b, b=%b, c=%b, f=%b",
              $time, a, b, c, f);

    // Exhaust all input combinations,
    // each time followed by a delay of 1 time unit.
    for (i = 0; i < 8; i += 1) begin
      // Each variable gets one bit from i.
      // First 0, 0, 0
      // then  0, 0, 1, etc.
      {a, b, c} = i;
      #1;
    end

    // Done.
    $finish;
  end

endmodule
```

a) ⌛ Start by creating two files:

- Verilog source file, called `structural_example.v`, containing the gate-level model given in Listing 1.1; and
- Verilog testbench file, called `structural_example_tb.v`, containing the code for generating various input combinations (see Listing 1.2), so that you can test the Verilog model of your circuit.

Note: The two files above are also available for download from Moodle.

Run circuit simulation and generate the waveforms. To do so, feel free to follow the sequence of commands given in Listing 1.3 below. Inspect the timing waveforms to verify that the Verilog description matches the expected behavior of the logic circuit in Figure 1.

Listing 1.3: Inspecting the waveform simulation in GTKWave.

```
$ iverilog -o example structural_example.v \
                      structural_example_tb.v
$ ./example
$ gtkwave structural_example.vcd
```

b) ⌛ Consider a slightly different logic circuit shown in Figure 2, which has three inputs a, b, c, and one output f. Write a structural (i.e., gate-level) Verilog description of the circuit. Use only Verilog gate primitives (e.g., **not**, **and**, **nor**).

Simulate your circuit (i.e., generate waveforms to verify its correct operation).
Hint: You can use the same testbench as in the previous question.

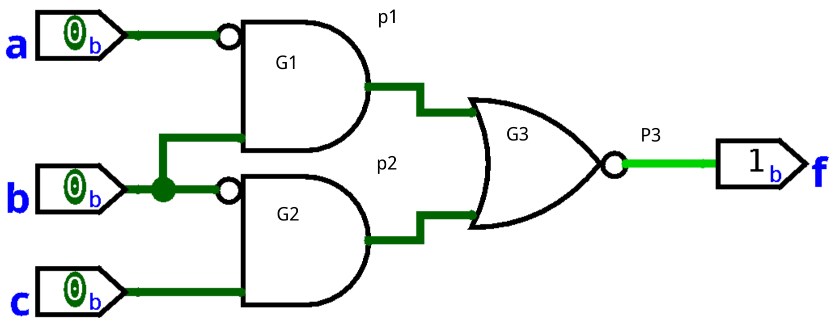

Figure 2: A more involved combinational circuit with three inputs and one output.

c) ⌛ Recall that a Full-Adder is a one-bit adder that can be realized with two Half-Adders (see Figure 3). A Full-Adder has three binary inputs and two binary outputs:

- one-bit inputs x, y;
- one-bit input carry-in c_{in};
- one-bit output s, which is the one-bit binary sum of $x + y$; and
- one-bit output carry-out c_{out}.

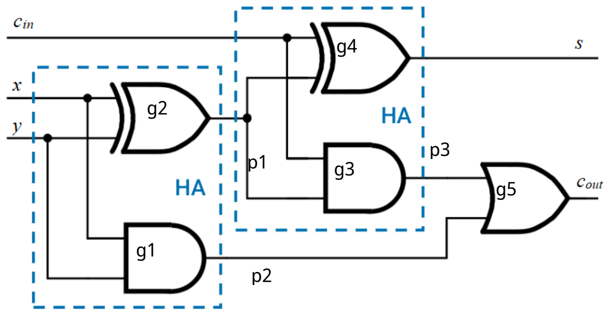

Figure 3: A logic circuit performing the function of a Full-Adder.

Write a structural (gate-level) Verilog description of a Full-Adder.

i) Variant 1: For a moment, ignore the fact that some gates can be grouped into a Half-Adder. In other words, make your Full-Adder be entirely described with basic Verilog gates. Name your **`module`** `full_adder1` and give it three inputs `x`, `y`, `cin`, and two outputs `s`, `cout`. As before, check that your Verilog model is correct by inspecting the waveforms in GTKWave. This time, use the testbench file `full_adder1_tb.v` (available for download from Moodle).

ii) Variant 2: Create a Verilog module `half_adder` that describes the functionality of a Half-Adder using the gate-level modeling approach. Then, use `half_adder` as a submodule to simplify the description of your Full-Adder. Name your Full-Adder **`module`** `full_adder2` and give it three inputs `x`, `y`, `cin`, and two outputs `s`, `cout`. As before, check that your Verilog model is correct by inspecting the waveforms in GTK-Wave. This time, use the testbench file `full_adder2_tb.v` (available for download from Moodle).

[Exercise 2] Logic Circuits and Verilog: Behavioral Modeling

a) ⌛ Recall that an n-to-1 Multiplexer (MUX) is a circuit which takes $n+1$ inputs:

- n data inputs $x_1, \dots, x_n$; and
- an m-bit selection signal s whose value determines which of the n data inputs to select as the circuit's output.

An example of a 2-to-1 MUX is shown in Figure 4.

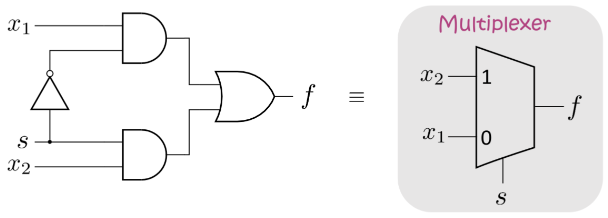

Figure 4: A 2-to-1 MUX.

In this question, you are asked to write behavioral Verilog descriptions of multiplexers using procedural statements (e.g., **`if-else`**) instead of structural descriptions using gates (e.g., **`and`**). Recall that procedural code should be wrapped in an **`always`** block. In particular, combinational circuits such as multiplexers can react to all module inputs by starting the block with the sensitivity list `always @*`, and they can set variables using the = blocking assignment operator.

Listing 1.4 shows a basic example of this syntax with a NOT gate.

Listing 1.4: A behavioral Verilog description of a NOT gate.

```verilog
module behavioral_not (
  input      a,
  // Output declared as a reg variable so that it can
  // procedurally be assigned to within always.
  output reg f
);
  always @* begin
    f = !a; // Executed every time the input changes.
  end
endmodule
```

i) Write a Verilog description of a 2-to-1 MUX using an **if-else** statement rather than gate primitives. Name your **module** `mux_2to1`, give it two data inputs `x1`, `x2`, a select input `s`, and one output `f`, and save it in a file named `mux_2to1.v`.

Check that your description is correct by inspecting its waveform in GTKWave. Listing 1.5 shows how you can achieve this with the testbench file `mux_2to1_tb.v` (available for download on Moodle).

Listing 1.5: Inspecting the 2-to-1 MUX waveform simulation in GTKWave.

```
$ iverilog -o mux_2to1 mux_2to1.v mux_2to1_tb.v
$ ./mux_2to1
$ gtkwave mux_2to1.vcd
```

ii) Write a Verilog description of a 3-bit 8-to-1 MUX using a **case** statement rather than gate primitives. Name your **module** `mux_8to1`. Give it eight 3-bit data inputs `x1,...,x8`, a select input `s`, and one 3-bit output `f`.

Hint: Recall that the number of select bits in `s` changes with the number of inputs n.

As before, check that your description is correct by inspecting its waveform in GTKWave. This time, use the testbench file `mux_8to1_tb.v` (available for download on Moodle).

Acknowledgments

List of contributors and authors: Basil Contovounesios.
Some questions are inspired by examples in the following books:

[1] M. M. Mano and M. D. Ciletti. Digital Design. Pearson, sixth global edition, 2019.

[2] J. F. Wakerly. Digital Design: Principles and Practices. Pearson, fifth edition, 2018.