Introduction to Verilog

Dov Kruger

Department of Electrical and Computer Engineering
Rutgers University

October 14, 2025

What is Verilog?

- Hardware Description Language (HDL)
- Used to model digital systems
- Part of the C family of languages
- VHDL is a competitor, based on Ada
- Widely used for FPGA and ASIC design
- Supports behavioral, structural, and gate-level modeling

Versions of Verilog

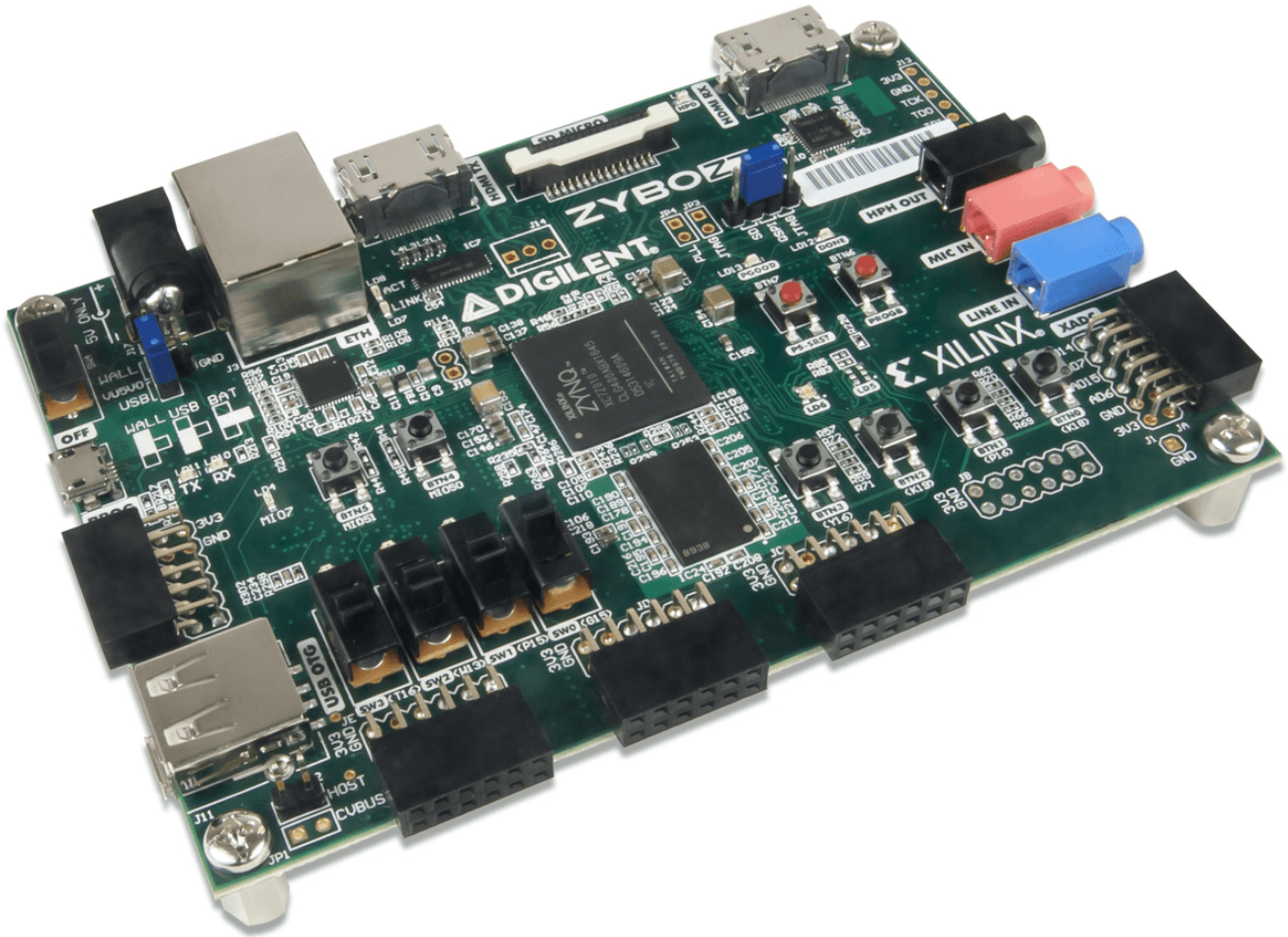
- 1984: Verilog introduced by Gateway Design Automation
- 1995: IEEE standardized Verilog as IEEE 1364-1995
- 2001: Verilog-2001, major enhancements including generate blocks
- 2005: Verilog-2005, minor updates and clarifications
- 2009: SystemVerilog, extended to include object-oriented features
- 2012: IEEE 1800-2012, latest version with enhanced features for verification and design

What is an FPGA?

- Field-Programmable Gate Array (FPGA)
- Configurable hardware components: logic blocks, I/O blocks, routing
- Used for prototyping, custom hardware, and flexible digital designs
- Can be reprogrammed after manufacturing
- Commonly used in embedded systems, signal processing, and more
- Much higher performance per watt than a computer

Example Board

Z7000 FPGA Board, $399 (2024)



FPGA Models and Specifications

Model	Year	Clock	Units	RAM	Price	Power
Xilinx Zynq 7000	2018	667 MHz	23k	1GB DDR3	$399	1.5W
Xilinx UltraScale+	2017	1.2 GHz	2.5M	5GB DDR4	$3962	62W

Verilog Usage in This Course

- We will focus on writing Verilog code
- Not programming actual FPGAs in the lab
- Verilog allows us to design complex circuits beyond basic lab setups
- FPGA access available upon request for additional hands-on practice

FPGA vs ASIC

- Application Specific Integrated Circuit (ASIC) are custom and must be built to order
- Requires volume, lead time, and high upfront cost
- FPGA are configurable after manufacturing
- FPGA are less power efficient than ASICs
- FPGA are larger than ASICs
- FPGA are more expensive than ASICs

Verilog Compilers/Simulators

- Several options are available, each with its own advantages and drawbacks.
- We will use Icarus Verilog, an open-source Verilog compiler and simulator.

Name	Type	Download Site	Notes
Icarus Verilog	Open Source	`http://iverilog.icarus.com`	Lightweight, easy to use
Xilinx Vivado	Commercial	`https://www.xilinx.com`	Industry standard, large install
ModelSim	Commercial	`https://www.mentor.com`	Widely used, powerful features

Additional Information on Verilog Tools

- **Icarus Verilog:** Best for small projects and learning environments. Limited support for advanced SystemVerilog features.
- **Xilinx Vivado:** Essential for FPGA development, especially with Xilinx hardware. Includes extensive support for IP cores and other hardware features.
- **ModelSim:** Excellent debugging and waveform analysis tools. Can handle large and complex designs.
- **Verilator:** Not a traditional simulator; it translates Verilog into C++/SystemC for fast simulation. Best for projects needing high-speed simulation and integration with other C++ systems.
- **Synopsys VCS:** Premium tool with advanced verification capabilities, including formal verification and coverage analysis. Used in professional ASIC/FPGA design.

Naming Conventions in Verilog

- `.v` — Verilog source files
- `.sv` — SystemVerilog source files
- `_tb.sv` — Testbench files (used for simulation)
- Names should be descriptive of the module's function
- Testbench files simulate how the circuit behaves (similar to a "main" in programming)

First Program

```
`timescale 1ns/1ns

module first_example;
  reg a, b;
  wire result;
  assign result = a & b; // AND gate

  initial begin
    $monitor("Time = %0t : a = %b, b = %b, result =
    a = 0; b = 0;
    #10 a = 0; b = 1;
    #10 a = 1; b = 0;
    #10 a = 1; b = 1;
    #10 $finish;
  end
endmodule
```

Data Types Overview

The following code shows integer data types and how to print to the screen

```
module datatypes_overview_tb;
  reg [7:0] a;
  reg [15:0] b;
  logic signed [15:0] c;
  logic signed [31:0] d;
  logic signed [63:0] e;

initial begin
  a = 8'hAA;
  b = 16'h5555;
  c = 16'hFFFF;
  d = 123;
  e = 64'h1234567812345678;

  $display("byte (8-bit): %b", a);
```

Bitwise Operations

The following example shows bitwise operations

```
module bitwise_ops_tb;
    reg [3:0] a = 4'b1010;
    reg [3:0] b = 4'b1100;

    initial begin
        $display("a & b = %b", a & b);
        $display("a | b = %b", a | b);
        $display("a ^ b = %b", a ^ b);
        $display("~a    = %b", ~a);
        $finish;
    end
endmodule
```

NAND Gate

```
module nand_gate(
    input wire a,
    input wire b,
    output wire y
);
    assign y = ~(a & b);
endmodule
```

NAND Gate TestBench

```
module nand_tb;
  reg a, b;
  wire y;
  nand_gate uut (  // instantiate (create) the NAND
    .a(a),
    .b(b),
    .y(y)
  );

  initial begin
    a = 0; b = 0; #1 $display("NAND(0,0) = %b", y);
    a = 0; b = 1; #1 $display("NAND(0,1) = %b", y);
    a = 1; b = 0; #1 $display("NAND(1,0) = %b", y);
    a = 1; b = 1; #1 $display("NAND(1,1) = %b", y);
    $finish;
  end
endmodule
```

NAND Gate, v2

assignment without keyword "assign" must be inside an "always" block

```
module nand_gate(
    input wire a,
    input wire b,
    output wire y
);
  always @(*) begin
    y = ~(a & b);
  end
endmodule
```

Example: Majority Vote

- Write a module that has 3 inputs A,B,C
- returns 1 if 2 of the 3 inputs are 1, 0 otherwise
- How will you test this module?

Multiplexer Example

The following code shows a multiplexer based on a two-bit input

```
module mux2_8bit(
    input wire [7:0] a,
    input wire [7:0] b,
    input wire sel,
    output logic[7:0] y
);

always @(*) begin
    case (sel)
        2'b00: y = a;
        2'b01: y = b;
        default: y = 8'b0;
    endcase
end

endmodule
```

If Statement

The following example demonstrates a simple if statement:

```
module if_example();
    initial begin
        int x = 5;
        if (x > 3) begin
            $display("x is greater than 3");
        end
    end
endmodule
```

If-Else Statement

The following example shows if-else control flow:

```
module if_else_example();
    initial begin
        int score = 85;
        if (score >= 90) begin
            $display("Grade: A");
        end else begin
            $display("Grade: B");
        end
    end
endmodule
```

Comparison Operators

The following operators are used for comparison in SystemVerilog:

```
module comparison_operators();
    initial begin
        int a = 5, b = 3;

        // Equal to
        if (a == b) $display("a equals b");

        // Not equal to
        if (a != b) $display("a is not equal to b")

        // Greater than
        if (a > b)  $display("a is greater than b")

        // Less than
        if (a < b)  $display("a is less than b");
```

Logical Operators

The following operators are used for logical operations in SystemVerilog:

```
module logical_operators();
    initial begin
        logic a = 1, b = 0;

        // Logical AND
        if (a && b) $display("Both a AND b are true

        // Logical OR
        if (a || b) $display("Either a OR b is true

        // Logical NOT
        if (!b) $display("b is false");

        // Can combine operators
        if ((a || b) && !b) $display("a is true AND
```

For Loop

The following example demonstrates a for loop:

- There are 3 parts to a for loop:
- Initialization (happens once at the beginning)
- Condition (checked at the beginning of each iteration)
- Update (happens after each iteration)

```
module for_loop_example();
  initial begin
    for (int i = 0; i < 3; i++) begin
      $display("Iteration %0d", i);
    end
  end
endmodule
```

More For Loop Examples

Here are different ways to use for loops with various patterns:

```
module for_loop_patterns();
  initial begin
    for (int i = 10; i > 0; i--) begin
      $display("Counting down: %0d", i);
    end

    for (int n = 1; n <= 64; n = n * 2) begin
      $display("Powers of 2: %0d", n);
    end

    for (int n = 2; n < 100; n = (n * 3) + 1) begin
      $display("n= %d", n);
    end
  end
endmodule
```

Edge Effects in For Loops

The following examples show how different comparison operators affect loop behavior:

```
module edge_effects();
  initial begin
    $display("Using < 4:");
    for (int i = 1; i < 4; i++) begin
      $display("Count: %0d", i);
    end
    $display("\nUsing <= 5:");
    for (int i = 1; i <= 5; i++) begin
      $display("Count: %0d", i);
    end
  end
endmodule
```

While Loop

The following example demonstrates a while loop: This while is equivalent to the for loop shown

```
module while_loop_example();
   initial begin
      int i = 0;
      while (i < 5) begin
         $display("Count: %0d", i);
         i++;
      end
      for (int i = 0; i < 5; i++) begin
         $display("Count: %0d", i);
      end
   end
endmodule
```

More Edge Effects in For Loops

```
module edge_effects_2();
  initial begin
    $display("Using < 4:");
    for (int i = 1; i < 4; i++) begin
      $display("Count: %0d", i); // Prints 1,2,3
    end
    $display("\nUsing <= 3:");
    for (int i = 1; i <= 3; i++) begin
      $display("Count: %0d", i); // Also prints 1,2
    end
  end
endmodule
```

Common Loop Pitfalls

```
module loop_pitfalls();
    logic [7:0] count;
    initial begin
        for (int i = 0; i > -5; i++) begin
            $display("%0d", i);
        end

        for (count = 1; count < 1000; count++) begi
            $display("%0d", count);
        end

//verilog requires all 3 parts to be there
//          for (int i = 0; i < 5; ) begin
//              $display("%0d", i);
//          end

        for (int i = 0; i >= 0; i++) begin
```

Floating Point Pitfalls

```
module floating_point_pitfalls();
  real w,x,y,z;
  initial begin
    for (x = 0.0; x != 1.0; x += 0.1) begin
      $display("%f", x);
    end
    for (real y = 1e8; y <= 1e8 + 100; y++) begin
      $display("%f", y);
    end
    z = 0.1 + 0.2;
    if (z == 0.3) begin
      $display("Equal!");
    end
    w = 0.0;
    for (int i = 0; i < 10; i++) begin
      w += 0.1;
    end
```

Arrays in Verilog

- An array is a collection of variables of the same type
- Arrays are fixed size and must be declared with size at compile time
- No bounds checking is performed at runtime
- Out of bounds access leads to undefined behavior
- Arrays cannot be resized dynamically
- No built-in array operations (like sort, find, etc.)
- Memory arrays require special syntax for synthesis

Array Example

```
module arrays();
  int a[4];                    // Array of 4 integers
  reg [7:0] b[4];
  reg [7:0] c[4];
  reg [7:0] d[4];
  initial begin
    a[0] = 10; // initializing one by one
    a[1] = 20;
    a[2] = 30;
    a[3] = 40;
    // array block assignment does not work in iver
//    b = {8'h0A, 8'h0B, 8'h0C, 8'h0D};  // concate
//    c= {4{8'hFF}};                     // set all
//    d= '{default: 8'h00};
  end
endmodule
```

Array TestBench

```
module testbench;
  // Parameters
  parameter int NUM_TESTS = 4;

  // Test vectors
  reg [7:0] test_inputs[NUM_TESTS];
  reg [7:0] expected_outputs[NUM_TESTS];

  // DUT (Device Under Test) signals
  reg [7:0] input_signal;
  wire [7:0] output_signal;

  // Instantiate the DUT
  my_module dut (
    .input_signal(input_signal),
    .output_signal(output_signal)
);
```