Digital Circuits and Systems - Verilog

ECS 326/676

Course Instructor: Dr. Santanu Talukder

Teaching Assistant: Ishanya
ishanya21@iiserb.ac.in

Department: Electrical Engineering and Computer Science

Indian Institute of Science Education and Research, Bhopal

Jan 2025 – Apr 2025

Introduction

What is Digital Design?

- Design of circuits that process digital signals (0s and 1s).
- Used in computing, communication, and embedded systems.
- Everything in Digital Circuits and Systems-ECS 326/676, so pay attention in class xD

Why Hardware Description Languages (HDLs)?

- Manual circuit design is complex and error-prone.
- HDLs describe hardware behavior at an abstract level.
- Used for simulation, synthesis, and FPGA/ASIC design.

Verilog vs. Programming Languages

- **C:** Executes sequentially on a CPU.
- **Verilog:** Describes hardware that works in parallel.

Introduction to Verilog

What is Verilog?

- A **Hardware Description Language (HDL)** used to model digital circuits.
- Describes circuit behavior, structure, and timing.
- Used for designing and simulating hardware before fabrication.

Where is Verilog Used?

- **FPGA (Field-Programmable Gate Arrays):** Reconfigurable hardware.
- **ASIC (Application-Specific Integrated Circuits):** Custom-designed chips.
- **VLSI (Very Large-Scale Integration):** High-density integrated circuits.

What should you know?

- Combinational vs. Sequential Circuits
- Logic Gates: AND, OR, NOT, XOR
- Truth tables
- FSMs
  - Model systems with defined states and transitions.
  - Types: **Moore** (output depends only on state) and **Mealy** (output depends on state + input).
  - Used in control logic, counters, and protocol design.

Verilog Design Flow with Icarus Verilog

1. **Write Verilog Code:** Define circuit behavior.
2. **Compile with Icarus Verilog:** Use `iverilog` to generate an executable.
3. **Simulate with VVP:** Run the compiled output using `vvp`.
4. **View Waveforms in GTKWave:** Generate waveform output (`.vcd`) for debugging. Analyze results visually.

Testbench

- A **testbench** is a separate module used to verify circuit behavior.
- It applies **stimulus** (inputs) and observes the outputs.
- Common commands:
  - `$monitor` – Displays signal values during simulation.
  - `$display` – Prints custom messages in the simulation output.
  - `#delay` – Introduces time delays in testbenches.

Basic Syntax in Verilog

- Module Definition: Every Verilog design starts with a ‘module‘ declaration.
- End Statement: Use ‘endmodule‘ to indicate the end of a module.
- Do not start names with numbers.
- White spaces are ignored.
- Comments:
  - // for single-line comments.
  - /* ... */ for multi-line comments.
- Case Sensitivity: Verilog is case-sensitive (module is not the same as MODULE).
- Semicolon (;): Required at the end of each statement.

Verilog Data Types

- **wire** - Represents combinational logic.
- **reg** - Stores values, used in sequential logic.
- **integer** - Stores whole numbers.
- **parameter** - Defines constants.

```
wire A, B;
reg C;
integer count;
parameter WIDTH = 8;
```

Operators in Verilog (1/3): Arithmetic, Logical, Relational

- **Arithmetic Operators:**
  - +, -, *, /, %
- **Logical Operators:**
  - &&, ||, !
- **Relational Operators:**
  - >, <, >=, <=, ==, ==

```
assign sum  = a + b;
assign flag = (a > b) && (c != d);
assign eq   = (a == b);
```

Operators in Verilog (2/3): Bitwise and Shift

- **Bitwise Operators:**
  - &, |, ^, ~
- **Shift Operators:**
  - << (left shift) and >> (right shift)

```
assign bit_and = a & b;
assign shifted = a << 2;
```

Operators in Verilog (3/3): Concatenation & Replication

- **Concatenation:**
  - Uses braces `{...}`
  - Combines multiple bits or buses.
- **Replication:**
  - Uses braces with a repetition factor `{N{expr}}`

```
assign combined   = {a[2], a[1], a[0]};
assign replicated = {4{a[0]}};
```

HDL Implementation: Structural (Gate-Level)

- Defines design by interconnecting basic gates and modules.
- Uses explicit instantiation of lower-level primitives.
- Offers detailed representation at the transistor or gate level.

```
module random_xor(input a, b, output y);
    xor g1 (a, b, y)
endmodule
```

HDL Implementation: Behavioral

- Describes functionality using high-level constructs.
- Uses procedural blocks such as `always` and `initial`.
- Abstracts the detailed gate-level connections.

```
module and_gate(input a, input b, output y);
    assign y = a & b;
endmodule
```

Initial vs. Always in Verilog

Initial: Process executes exactly once.

```
initial begin
    a = 1;
    #1;
    b = a;
end
```

Always: Process is rescheduled many times.

```
always @(a or b)
begin
    if (a)
        c = b;
    else
        d = ~b;
end
```

Procedural Assignment in Verilog

- **Blocking Assignment (=)**
  - Executes statements sequentially.
  - Used for combinational logic.
  - Statement order is strictly followed.
- **Non-Blocking Assignment (<=)**
  - Statements are scheduled concurrently.
  - Common in sequential (clocked) logic.
  - Allows switching values without temporary storage.

Blocking vs. Non-Blocking Example

Blocking Assignment Example:

```
initial begin
    a = 1;  // executes immediately
    b = a;  // b gets the updated value of a
end
```

Non-Blocking Assignment Example:

```
always @(posedge clk) begin
    a <= b;  // scheduled concurrently
    b <= a;  // old value of a is used
end
```

AND and OR Gate Modules

Is this structural or behavioral implementation ?

```
// 3-input AND gate module
module and3 (output Y, input A, B, C);
  and (Y, A, B, C);
endmodule

// 3-input OR gate module
module or3 (output Y, input A, B, C);
  or (Y, A, B, C);
endmodule
```

Testbench for AND and OR Gates (1/2)

```
// Testbench to test both AND and OR gates
module tb_gate;
  reg A, B, C;
  wire Y_and, Y_or;

  // Instantiate the 3-input AND gate
  and3 u1 (.Y(Y_and), .A(A), .B(B), .C(C));

  // Instantiate the 3-input OR gate
  or3 u2 (.Y(Y_or), .A(A), .B(B), .C(C));
```

Testbench for AND and OR Gates (2/2)

```
  initial begin
    // Monitor changes in signals
    $monitor("Time: %0d | A=%b B=%b C=%b |
       AND Output=%b OR Output=%b",
              $time, A, B, C, Y_and, Y_or);
    A = 0; B = 0; C = 0; #10;
    A = 0; B = 0; C = 1; #10;
    A = 0; B = 1; C = 0; #10;
    A = 0; B = 1; C = 1; #10;
    A = 1; B = 0; C = 0; #10;
    A = 1; B = 0; C = 1; #10;
    A = 1; B = 1; C = 0; #10;
    A = 1; B = 1; C = 1; #10;
    $finish;
  end
endmodule
```

Run in Terminal

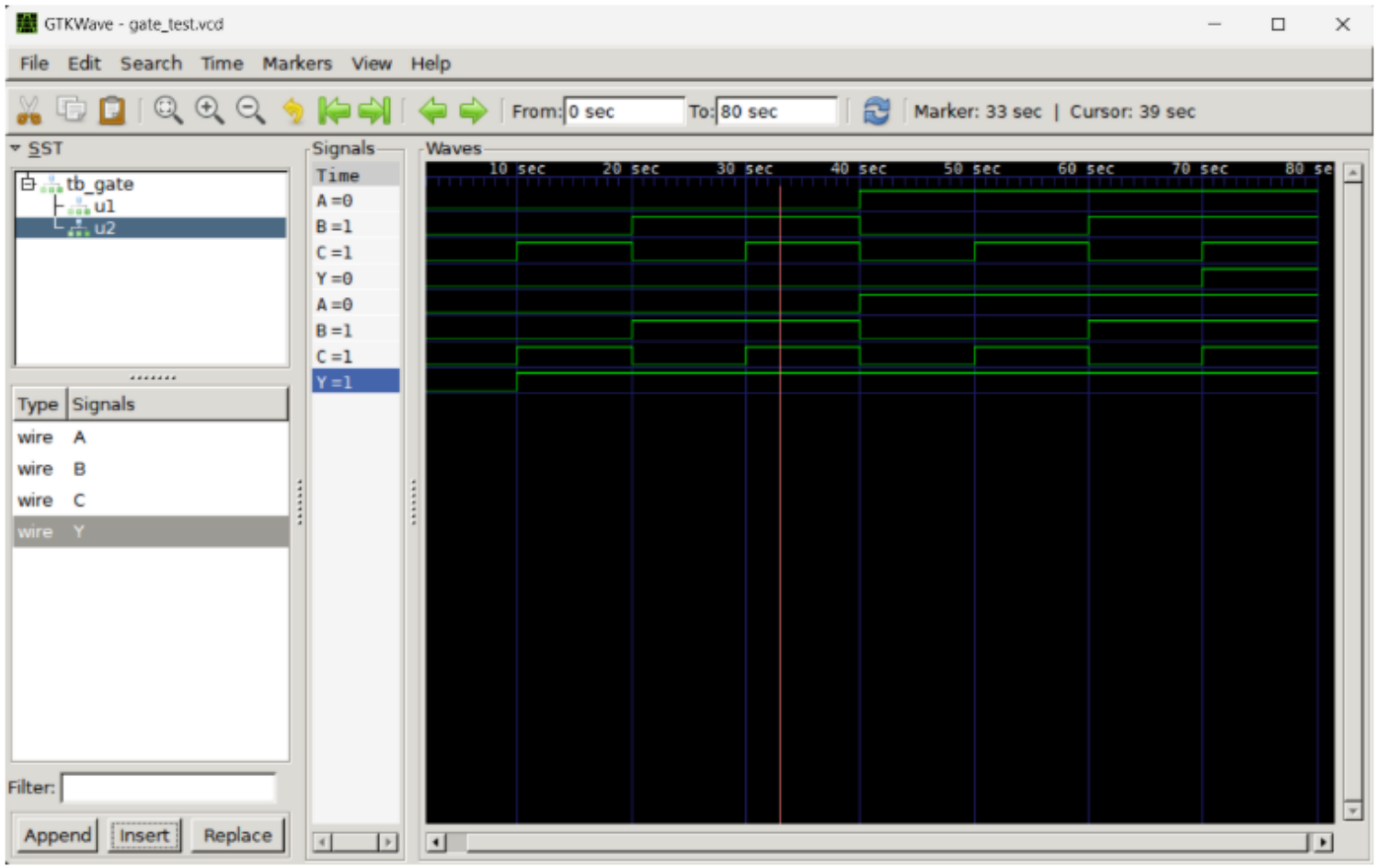
```
C:\iverilog\bin>iverilog -o gate_test gate_test.v

C:\iverilog\bin>vvp gate_test
Time:  0 | A=0 B=0 C=0 | AND Output=0 OR Output=0
Time: 10 | A=0 B=0 C=1 | AND Output=0 OR Output=1
Time: 20 | A=0 B=1 C=0 | AND Output=0 OR Output=1
Time: 30 | A=0 B=1 C=1 | AND Output=0 OR Output=1
Time: 40 | A=1 B=0 C=0 | AND Output=0 OR Output=1
Time: 50 | A=1 B=0 C=1 | AND Output=0 OR Output=1
Time: 60 | A=1 B=1 C=0 | AND Output=0 OR Output=1
Time: 70 | A=1 B=1 C=1 | AND Output=1 OR Output=1
```

GTKWave

Terminal Command

```
gtkwave gate_test.vcd
```



Half Adder Module (Structural)

```
module HA(input x, y, output b, d);
    // Generate carry: AND gate
    and g2(b, x, y);    // b = x AND y

    // Generate sum (difference): XOR gate
    xor g3(d, x, y);    // d = x XOR y
endmodule
```

Full Subtractor

```
module fs(input a, b, c, output bo, d);
wire na;
wire x, y, z;

not g0(na, a);
xor g1(d, a, b, c);
or g2(x, c, b); //borrow
and g3(y, na, x);
and g4(z, c, b);
or g5(bo, y, z);
endmodule

module tb_fs;
reg a, b, c;
wire bo, d;
fs uut (.a(a), .b(b), .c(c), .d(d), .bo(bo));

initial begin
$dumpfile("fs.vcd");
$dumpvars(0, tb_fs);
$monitor("Time %t,a = %b, b = %b, c = %b, bo = %b, d = %b", $time, a,b,c, bo,d);

#1; a = 0; b = 0; c = 0;
#10; a = 0; b = 0; c = 1;
#1; a = 0; b = 1; c = 0;
#1; a = 0; b = 1; c = 1;
#1; a = 1; b = 0; c = 0;
$finish;
end
endmodule
```

Random Function

```
module fun(input a, b, c, output y);

assign y = a & ~b | ~b & ~c | ~a & b & c;

endmodule

module tb_fun;
reg a, b, c;
wire y;
fun uut (.a(a), .b(b), .c(c), .y(y));

initial begin
$dumpfile("func.vcd");
$dumpvars(0, tb_fun);

$monitor("Time %t, a = %b, b = %b, c = %b, y = %b", $time, a, b, c, y);

#10; a = 0; b = 0; c = 0;
#10; a = 0; b = 0; c = 1;
#10; a = 0; b = 1; c = 0;
#10; a = 1; b = 0; c = 1;
#10; a = 1; b = 0; c = 1;
#10; a = 1; b = 1; c = 0;
#10; a = 1; b = 1; c = 1;
#10; a = 1; b = 1; c = 1;

$finish;
end

endmodule
```

3 to 8 Decoder

```
module decoder3to8(input a2, a1, a0, output y0, y1, y2, y3, y4, y5, y6, y7);
    wire na2, na1, na0;

    // Invert the inputs to create not versions of a2, a1, a0
    not n0(na2, a2);  // na2 = NOT a2
    not n1(na1, a1);  // na1 = NOT a1
    not n2(na0, a0);  // na0 = NOT a0

    // Create AND gates for each output
    and g0(y0, na2, na1, na0);  // 000
    and g1(y1, na2, na1, a0);   // 001
    and g2(y2, na2, a1, na0);   // 010
    and g3(y3, na2, a1, a0);    // 011
    and g4(y4, a2, na1, na0);   // 100
    and g5(y5, a2, na1, a0);    // 101
    and g6(y6, a2, a1, na0);    // 110
    and g7(y7, a2, a1, a0);     // 111
endmodule
```

3 to 8 Decoder Testbench

```
module tb_decoder3to8;
    reg a2, a1, a0;
    wire y0, y1, y2, y3, y4, y5, y6, y7;

    // Instantiate the 3-to-8 decoder
    decoder3to8 uut (.a2(a2), .a1(a1), .a0(a0),
                     .y0(y0), .y1(y1), .y2(y2), .y3(y3),
                     .y4(y4), .y5(y5), .y6(y6), .y7(y7));

    initial begin
        // Create the VCD file for waveform analysis
        $dumpfile("decoder3to8.vcd");
        $dumpvars(0, tb_decoder3to8);

        // Display the results in the console
        $monitor("Time %t, a2 = %b, a1 = %b, a0 = %b, y0 = %b, y1 = %b, y2 = %b,
              y3 = %b, y4 = %b, y5 = %b, y6 = %b, y7 = %b",
                 $time, a2, a1, a0, y0, y1, y2, y3, y4, y5, y6, y7);

        // Apply test cases for the decoder inputs
        #10 a2 = 0; a1 = 0; a0 = 0; // Select y0
        #10 a2 = 0; a1 = 0; a0 = 1; // Select y1
        #10 a2 = 0; a1 = 1; a0 = 0; // Select y2
        #10 a2 = 0; a1 = 1; a0 = 1; // Select y3
        #10 a2 = 1; a1 = 0; a0 = 0; // Select y4
        #10 a2 = 1; a1 = 0; a0 = 1; // Select y5
        #10 a2 = 1; a1 = 1; a0 = 0; // Select y6
        #10 a2 = 1; a1 = 1; a0 = 1; // Select y7

        // Finish the simulation
        $finish;
    end
endmodule
```

4 to 2 Encoder

```
module en4to2(input [3:0] in, output reg [1:0] out);
    always @(*) begin
        case (in)
            4'b0001: out = 2'b00; // Input: 0001 -> Output: 00
            4'b0010: out = 2'b01; // Input: 0010 -> Output: 01
            4'b0100: out = 2'b10; // Input: 0100 -> Output: 10
            4'b1000: out = 2'b11; // Input: 1000 -> Output: 11
            default: out = 2'b00; // Default: Catch invalid inputs
        endcase
    end
endmodule
```

4 to 2 Encoder Testbench

```
module tb_en4to2;
    reg [3:0] in;          // Input to test the encoder
    wire [1:0] out;        // Output from the encoder

    // Instantiate the 4-to-2 Encoder
    en4to2 uut (
        .in(in),
        .out(out)
    );

    // Testbench logic
    initial begin
        // Initialize GTKWave dump
        $dumpfile("en4to2.vcd");
        $dumpvars(0, tb_en4to2);

        // Display signal values on console
        $monitor("Time: %0t | Input: %b | Output: %b", $time, in, out);

        // Test cases
        in = 4'b0000; #10; // Invalid input
        in = 4'b0001; #10; // Expected Output: 00
        in = 4'b0010; #10; // Expected Output: 01
        in = 4'b0100; #10; // Expected Output: 10
        in = 4'b1000; #10; // Expected Output: 11
        in = 4'b1010; #10; // Invalid input

        // End simulation
        $finish;
    end
endmodule
```

8 to 3 Priority Encoder

```
module pe(input [7:0] in, output reg [2:0] out);
always @(*)
begin
if (in[7]) out = 3'b111;
else if (in[6]) out = 3'b110;
else if (in[5]) out = 3'b101;
else if (in[4]) out = 3'b100;
else if (in[3]) out = 3'b011;
else if (in[2]) out = 3'b010;
else if (in[1]) out = 3'b001;
else if (in[0]) out = 3'b000;
else out = 3'b000;
end
endmodule

module tb_pe;  //Testbench
reg [7:0] in;
wire [2:0] out;
pe uut(.in(in), .out(out));
initial begin

$dumpfile("pe.vcd");
$dumpvars(0, tb_pe);
$monitor("Time %t, in = %b, out = %b", $time, in, out);
#10; in = 8'b0000_0010;
#10; in = 8'b0000_0001;
#10; in = 8'b0000_0100;
#10; in = 8'b0000_1000;
#10; in = 8'b0010_0000;
#10; in = 8'b0100_0000;
$finish;
end
endmodule
```

Practice Questions: Verilog Coding

Try both structural and behavioral implementation. You are free to combine both too.

1. Write a Verilog module for a 2-to-1 Multiplexer.
2. Write a Verilog module for a 1-to-4 Demultiplexer.
3. Write a Verilog module for a positive edge-triggered D Flip-Flop.
4. Write a Verilog module for a 4-bit Binary Counter.
5. Write a Verilog module for a Half Adder, Full Adder, Half subtractor, Full subtractor.

References

- **Course:** ECS 326/676 2023-24 by Dr. Santanu Talukder.
- **Course:** ECS409/609 Computer Orgaisation 2024-25
- **Textbook:** Digital design by Morris Mano (5th edition)
- **Textbook:** *Digital Design and Computer Architecture*, Chapter 4, by Sarah Harris and David Harris.
- **GitHub Repository:** https://github.com/ishanyaa/computer_organisation
- Youtube and Internet. (Our Lord and Saviour xD)

PS: Let me know, if there are any mistakes and I will update the slides.

Advice

A Word of Advice:

Should you encounter difficulties or find yourself stuck, remember that help is always within reach. Whether it be asking for guidance from your profs/peers or exploring the wealth of resources available on the internet, do not hesitate to seek assistance.
Embrace challenges as opportunities for growth, for as the adage goes, *"Practice makes you perfect; Every line of code brings you closer to mastery."*.
Push the boundaries of your knowledge!
Happy learning Verilog :)

Source: Internet