Contents

Session 1

Introduction

1.1 Digital Design with HDLs

Hardware Description Languages (HDLs) like Verilog and VHDL are fundamental tools for designing digital circuits and complex systems, ranging from simple logic gates to processors and system on chips (SoC). In this hands on guide, Verilog/SystemVerilog HDLs is used to design

- Combinational Logic Circuits
- Sequential Logic circuits
- Finite State Machines
- Application Specific Integrated Circuits (ASICs)
- General Purpose Systems (ALU, Memory, ..)

In addition, special attention is given to the implementation of digital circuits in FPGAs.

1.2 Learning Resources

1.2.1 Online Resources

- Good Reference to Verilog Language: `www.chipverify.com/verilog/verilog-hello-world`
- `www.asic-world.com/verilog/index.html`
- Simulator for digital circuits `https://sourceforge.net/projects/circuit/`
- x86 Assembler compiler `https://sourceforge.net/projects/guitasm8086/`
- Learn Verilog by Examples: `www.referencedesigner.com/tutorials/verilog/verilog_01.php`
- ..

1.2.2 Text Books

- Ciletti, M. Advanced Digital Design with the Verilog HDL. 2nd ed. Boston: Pearson, 2020.
- Perry, D. L. and Thornton, H. VHDL and SystemVerilog: Digital Design and FPGA Implementation. Cham: Springer 2020.
- Lee, C. (2021). Digital Logic Design Using Verilog: Coding and RTL Synthesis. Cham: Springer, 2021.
- Roth, C. H., John, L. K., and Ugave, B. Digital Systems Design Using Verilog. 1st ed. Boston: Cengage Learning, 2017.
- Bhasker, J. A SystemVerilog Primer. 3rd ed. New York: Star Galaxy Publishing, 2018.
- ..

1.3 Setting up the Environment for Digital Design with Verilog

1.3.1 Install Verilog Compiler and GtkWave for Windows

- Download Free Verilog Compiler (Icarus) and Wave form Viewer (GtkWave)
  `iverilog-v11-20210204-x64_setup.exe [44.1MB]`
  from `https://bleyer.org/icarus/`.
- Install the compiler in the folder `C:\iverilog\`
- Add new folder locations `C:\iverilog\bin` and `C:\iverilog\gtkwave\bin` to the Windows Environmental variable PATH.

Now the Verilog compiler is ready.

1.3.2 Hello World in Verilog

- Create a file called `myfirstprog.v` with the following Verilog code.

```
module myfirstprog();
    initial begin
        $display("Hello, World");
        $finish
    end
endmodule
```

- Open the `Console` and enter the following command
  `> iverilog -o myfirstprog.v.out myfirstprog.v`
  to create the output file `myfirstprog.v.out`
- Run the output file in the Verilog Virtual Processor (VVP) with the command
  `> vvp myfirstprog.v.out`
- If successful, the text `Hello, world` should appear in the console.

1.3.3 Use of VSCode as IDE

After the installation of Verilog compiler, it is possible to configure VSCode as IDE for Verilog program development.

- Install VSCode
- Install required extensions
  - Verilog HDL (Go to `Manage` of this extension and make the extension "Apply to All Profiles. Go to `Manage>Settings` and make the extension `Run in Terminal` and `Show Run Icon in Editor Title Menu`)
  - Verilog-HDL/SystemVerilog/Bluespec SystemVerilog Support
  - Verilog Highlight
- Now it is possible edit and run Verilog source files (with extension v) by clicking the icon (Verilog:Run Verilog HDL Code) that appear in the title menu (in the right-top most corner of VSCode editor).
- Source file is compiled and executed in the integrated Terminal of VSCode.

1.3.4 Use of EDA Playground (Online Platform) as IDE

EDA Playground is an online platform designed for hardware description language (HDL) simulation and verification, primarily used by digital designers and engineers working with FPGAs, ASICs, and other electronic systems.

- `https://www.edaplayground.com/`
- Follow the instructions in EDA Playground

1.3.5 Use of ... as IDE

TBC

Session 2

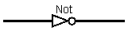
Inverter Design with Verilog - Getting Started

An inverter (also called a NOT gate) is the simplest combinational logic circuit that inverts its input signal. It has one input and one output, where the output is the logical complement of the input.

2.1 Truth Table

Input	Output
0	1
1	0

2.2 Gate Level Circuit Diagram



By the following, inverter is modeled by using Verilog in three modeling abstraction levels.

- Register Transfer Level
- Gate Level
- Behavioral Level

2.3 Verilog Code

2.3.1 RTL Modeling

RTL (Register Transfer Level) modeling in Verilog is a design abstraction that describes the flow of digital signals between hardware registers and the logical operations performed on those signals. It's a fundamental level of hardware description used in digital design before synthesis to gate-level netlists.

```
1 module mynotgate(
2     input   wire a,
3     output  wire nota
4     );
5
6     assign nota = ~a;
7
8 endmodule
```

2.3.2 Gate Level Modeling

Gate Level of abstraction in Verilog HDL, describes electronic circuits in terms of logic gates and their interconnections. It closely resembles the actual physical implementation of a digital circuit.

```
1 module mynotgate(
2     input   wire a,
3     output  wire nota
4     );
```

```
5
6       not inv1(nota, a);
7
8   endmodule
```

2.3.3 Behavioral Modeling

Behavioral modeling in Verilog describes the functionality (behavior) of a digital circuit without explicitly defining its hardware structure (like gates or transistors). It focuses on what the circuit does rather than how it is built, making it a higher level of abstraction compared to gate-level or dataflow modeling.

```
1  module mynotgate(
2      input wire a,
3      output reg nota
4      );
5
6      always @(*) begin
7          if(a)   nota = 0;
8          else    nota = 1;
9      end
10
11 endmodule
```

2.4 Test Bench for Code Verification

A testbench in Verilog is a simulation environment used to verify the functionality and correctness of a digital design (called the DUT - Design Under Test). It generates input stimuli (test vectors), applies them to the DUT, and checks the outputs against expected results.

Key Components of a Verilog Testbench

- **DUT (Design Under Test)**: The module being tested (e.g., adder, FSM, processor)
- **Test Stimulus Generator**: Produces input signals (`reg` variables) for the DUT.
- **Clock & Reset Generation**: Simulates clock signals for sequential circuits.
- **Monitor & Checker**: Displays results (`$display`, `$monitor`) and checks correctness (`assert`).
- **Simulation Control**: Controls simulation time (`#delay`) and termination (`$finish`).

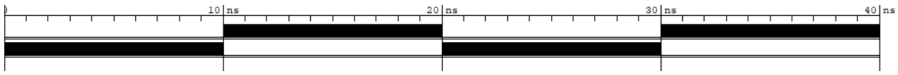
```
1  `timescale 1ns/1ps
2  `include "mynotgate.v"
3
4  module tb();
5
6      reg a;
7      wire nota;
8
9      mynotgate dut (.a(a), .nota(nota));
10
11     initial begin
12         a = 0;
13
14         $dumpfile("dump.vcd");
15         $dumpvars(0, tb);
16
17         #10; a = 1;
18         #10; a = 0;
19         #10; a = 1;
20         #10; a = 0;
21     end
22
23     initial begin
24         $monitor("a = %b, not a = %b", a, nota);
25     end
```

```
26
27     endmodule
```

2.4.1 Console Output

```
a = 0, not a = 1
a = 1, not a = 0
a = 0, not a = 1
a = 1, not a = 0
```

2.4.2 Wave Form Analyzer



2.5 Exercises

E 2.1 Write RTL code for basic logic gates; BUFFER, NOT, AND, NAND OR, XOR, NOR, XNOR.

E 2.2 Write Test Bench to verify each of the code.

E 2.3 How procedural block `initial` functions?

E 2.4 How procedural block `always` functions?

E 2.5 What is the default variable type in Verilog?

E 2.6 What are the differences between `wire` and `reg`?

Session 3

Comparator Design with Verilog

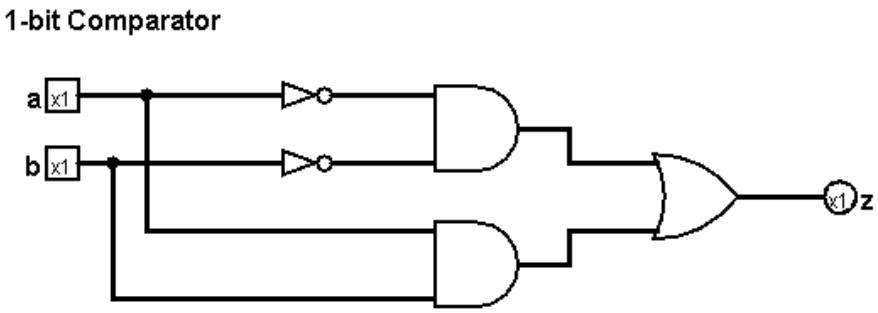
3.1 1-bit Comparator

3.1.1 RTL Modeling

Truth Table

Input a	Input b	Output c
0	0	1
0	1	0
1	0	0
1	1	1

Gate Level Circuit Diagram



Verilog Code

```
module comp1b
    (   input a,
        input b,
        output c
    );

    assign c = (~a & ~b) | (a & b);

endmodule
```

Test Bench

```
`timescale 1ns/1ps
`include "comp1b.v"

module comp1b_tb;
    reg a;
    reg b;
    wire c;

    // Instantiate the unit under test
    comp1b uut( .a(a), .b(b), .c(c));
```

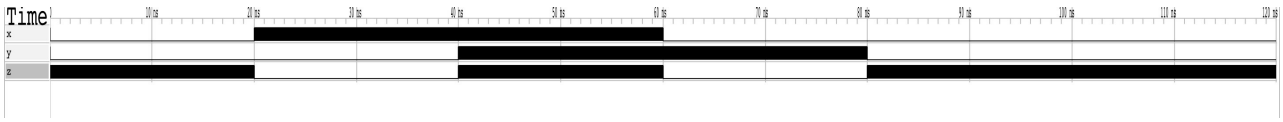
```
11
12      initial begin
13          $dumpfile("dump.vcd");
14          $dumpvars;
15
16                      a = 0; b = 0;
17          #10 x=1; a = 1; b = 0;
18          #10 y=1; a = 1; b = 1;
19          #10 x=0; a = 0; b = 1;
20          #10 y=0; a = 0; b = 0;
21          #10;
22      end
23
24  initial begin
25      $monitor("a=%1b b=%1b c=%1b",a,b,c);
26  end
27
28  endmodule
```

Testbench Output

Terminal output

```
x=0 y=0 z=1
a=1 b=0 z=0
a=1 b=1 z=1
a=0 b=1 z=0
a=0 b=0 z=1
```

Resulting signal wave forms using `..> gtkwave.exe .\build\dump.vcd`



3.1.2 Gate Level Modeling

Verilog Code

```
1  module comp1b(
2      input   wire a,
3      input   wire b,
4      output  wire c
5      );
6
7      wire na, nb, na_or_nb;
8      wire a_or_b;
9
10     not NG1(na, a);
11     not NG2(nb, b);
12     and AND1(na_or_nb, na, nb);
13     and AND2(a_or_b, a, b);
14
15     or OR1(c, na_or_nb, a_or_b);
16
17 endmodule
```

3.1.3 Behavioral Modeling

Verilog Code

```
1  module comp1b(
2      input wire a,
3      input wire b,
```

```
    output reg c
    );

    always @(*) begin
        if(a==b)  c = 1;
        else      c = 0;
    end

endmodule
```

3.1.4 Modeling with User Define Primitives

Verilog Code

```
module comp1b(input a, input b, output z);
    compare c0(c, a,b);
endmodule

primitive compare(output out, input in1, input in2);

    table
        //in1,in2:out
        0   0   :   1;
        0   1   :   0;
        1   0   :   0;
        1   1   :   1;
    endtable

endprimitive
```

3.2 Exercises

E 3.7 Write code that compares two input values x and y and gives 1 if x is greater than or equal to y. Write stimulus to verify it.

E 3.8 Implement and verify the Verilog code for a circuit that has three inputs and one one output. The three inputs represent a binary number (from 0 to 7) and output is 1 if the value is greater than 5 else it is 0.

E 3.9 Write Test Bench to verify each of the code.

3.3 1-bit Comparator with 3 x Outputs

Write Verilog code for 1-bit comparator that compares bits at input ports a and b. The comparator has 3 output ports: a_eq_b, a_gt_b and a_lt_b, which become 1 when $a = b$, $a > b$ and $a < b$, respectively. Corresponding truth table is given below.

a	b	aeb	agb	alb
0	0	1	0	0
0	1	0	0	1
1	0	0	1	0
1	1	1	0	0

Verilog Module Code

```
// Full 1-bit  Comparator
// 2 x Input (a, b), 3 x Outputs (a_eq_b, a_gt_b, a_ls_b)

module compfull1b(
    input   wire a,
    input   wire b,
    output  reg a_eq_b, // a = b
    output  reg a_gt_b, // a > b
    output  reg a_lt_b  // a< b
```

```
    );

    always @(*) begin

        if (a==b) begin
            a_eq_b = 1'b1;
            a_gt_b = 1'b0;
            a_lt_b = 1'b0;
        end
        else if(a>b) begin
            a_eq_b = 1'b0;
            a_gt_b = 1'b1;
            a_lt_b = 1'b0;
        end
        else begin
            a_eq_b = 1'b0;
            a_gt_b = 1'b0;
            a_lt_b = 1'b1;
        end

    end

endmodule
```

Testbench

```
`timescale 1ns/1ps
`include "compfull1b.v"

module tb();

    reg a;
    reg b;
    wire a_eq_b;
    wire a_gt_b;
    wire a_ls_b;

    compfull1b uut (
        .a(a),
        .b(b),
        .a_eq_b(a_eq_b),
        .a_gt_b(a_gt_b),
        .a_lt_b(a_lt_b)
        );

    initial begin
        // Dump waveform data (for GTKWave)
        $dumpfile("dump.vcd");  // Create a VCD file
        $dumpvars(0, tb);       // Dump all variables

                a = 1'b0; b = 1'b0;
        #10;    a = 1'b0; b = 1'b1;
        #10;    a = 1'b1; b = 1'b1;
        #10;    a = 1'b1; b = 1'b0;
        #10;    a = 1'b0; b = 1'b0;
        #10;
    end

    initial begin
        $monitor("a(%1b), b(%1b), a==b(%1b), a>b(%1b), a<b(%1b)", a, b, a_eq_b,
            a_gt_b, a_lt_b);
    end

endmodule
```

3.4 2 bit Comparator with 3 x Output

Verilog Code

```
module comp2bfull(
        input   wire [1:0] a,
        input   wire [1:0] b,
        output  reg a_eq_b, // a = b
        output  reg a_gt_b, // a > b
        output  reg a_lt_b  // a< b
);

always @(*) begin
    if (a==b) begin
        a_eq_b = 1'b1; a_gt_b = 1'b0; a_lt_b = 1'b0;
    end
    else if(a>b) begin
        a_eq_b = 1'b0; a_gt_b = 1'b1; a_lt_b = 1'b0;
    end
    else begin
        a_eq_b = 1'b0; a_gt_b = 1'b0; a_lt_b = 1'b1;
    end
end

endmodule
```

Test Bench

```
`timescale 1ns/1ps
`include "comp2bfull.v"

module tb();

    reg [1:0] a;
    reg [1:0] b;
    wire a_eq_b;
    wire a_gt_b;
    wire a_ls_b;

    comp2bfull uut (
        .a(a),
        .b(b),
        .a_eq_b(a_eq_b),
        .a_gt_b(a_gt_b),
        .a_lt_b(a_lt_b)
    );

    initial begin
        // Dump waveform data (for GTKWave)
        $dumpfile("dump.vcd");  // Create a VCD file
        $dumpvars(0, tb);       // Dump all variables

                a = 2'b00;  b = 2'b00;
        #10;                b = 2'b01;
        #10;                b = 2'b10;
        #10;                b = 2'b11;

                a = 2'b01;  b = 2'b00;
        #10;                b = 2'b01;
        #10;                b = 2'b10;
        #10;                b = 2'b11;

                a = 2'b10;  b = 2'b00;
        #10;                b = 2'b01;
        #10;                b = 2'b10;
        #10;                b = 2'b11;
```

```
39                  a = 2'b11;  b = 2'b00;
40          #10;                b = 2'b01;
41          #10;                b = 2'b10;
42          #10;                b = 2'b11;
43
44          #10;    a = 2'b00;  b = 2'b00;
45      end
46
47      initial begin
48          $monitor("a(%2b), b(%2b), a==b(%1b), a>b(%1b), a<b(%1b)", a, b, a_eq_b,
                a_gt_b, a_lt_b);
49      end
50
51  endmodule
```

3.5 IC 74LS85

The 74LS85 is a 4-bit magnitude comparator that supports cascading for multi-bit comparisons (e.g., 8-bit, 16-bit). Refer to the 74LS85 datasheet for detailed specifications. Write a Verilog module to emulate the functionality of the IC74LS85 4-bit magnitude comparator. Then, write a testbench to compare two 8-bit values by cascading two IC74LS85 modules.

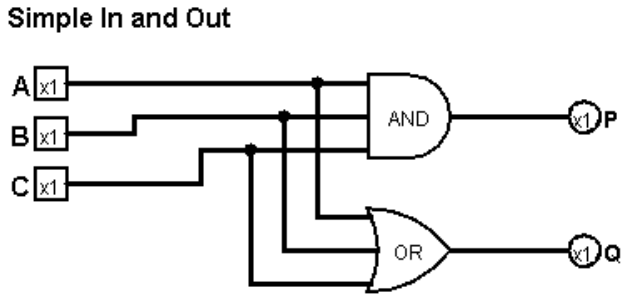
Session 4

Simple IO and Buses

4.1 Examples on IO and Buses

Write Verilog modules to describe the following circuits named as; Simple in and out, Intermediate Wire, Bus Signals, and Bus Breakout. Also write suitable test benches to test the modules.

4.1.1 Simple In and Out

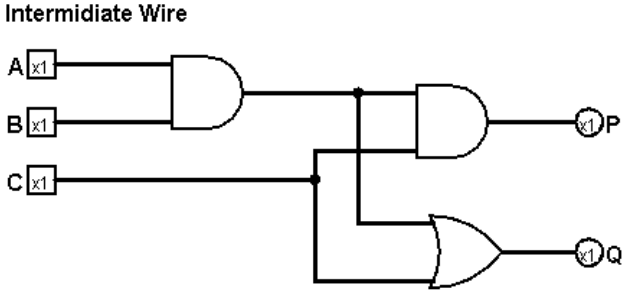



```
module simpleio(
    input wire a,
    input wire b,
    input wire c,
    output wire p,
    output wire q
);

    assign p = a & b & c;
    assign q = a | b | c;

endmodule
```

4.1.2 Intermediate Wire




```
module intmwire(
    input wire a,
    input wire b,
    input wire c,
```

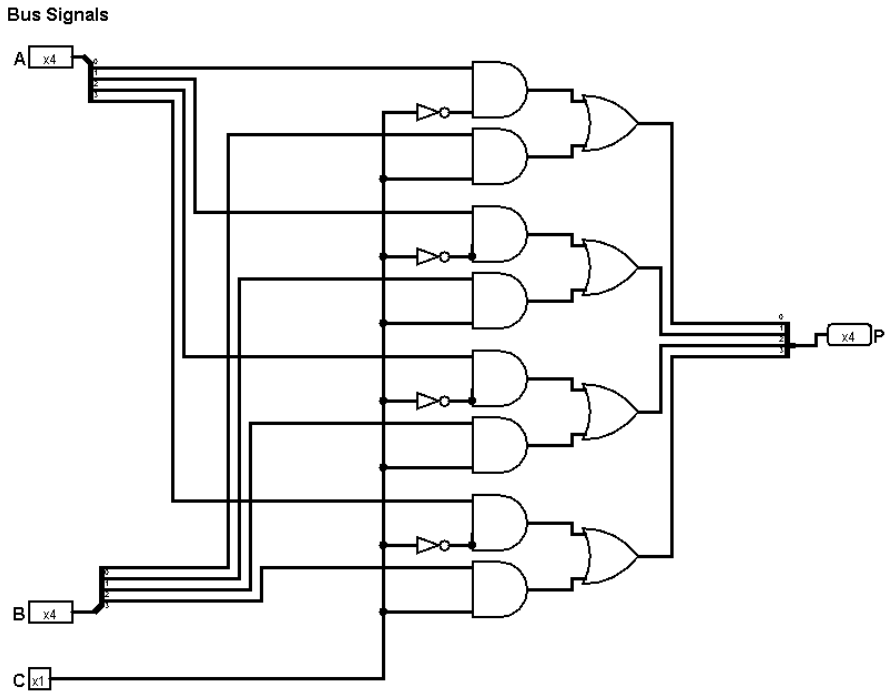
```
    output wire p,
    output wire q
);

    wire intmwire;

    assign intmwire = a & b;
    assign p = intmwire & c;
    assign q = intmwire | c;

endmodule
```

4.1.3 Bus Signals



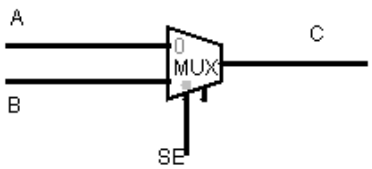

```
module bussignals(
    input wire [3:0] a,
    input wire [3:0] b,
    input wire c,
    output wire p
);

    wire [3:0] cbus;

    assign cbus = {4{c}};

    assign p = ( a & (~cbus)  |  ( b & cbus ) ;

endmodule
```



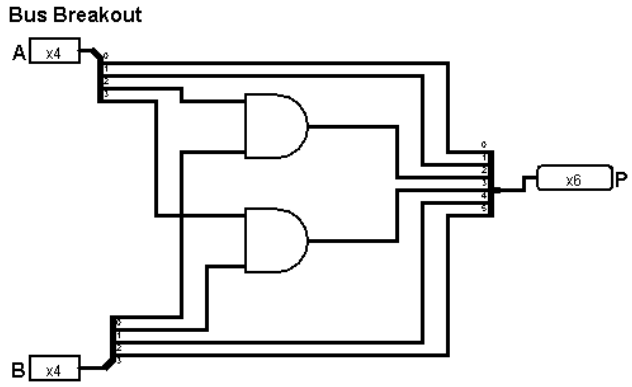
4.1.4 Simple Multiplexer

```
module simplemux(
    input wire [3:0] a, // Input 1
    input wire [3:0] b, // Input 2
    input wire se,      //Select
    output wire [3:0] c // Output
);

    assign c = se ? a : b;

endmodule
```

4.1.5 Bus Breakout




```
module busbreakout(
    input wire [3:0] a,
    input wire [3:0] b,
    output wire [5:0] p
);

    assign p = { b[3:2], (a[3] & b[1]), (a[2] & b[0]), a[1:0]};

endmodule
```

Session 5

Combinational Building Blocks

Combinational logic circuits are digital circuits where the output depends only on the current input values and not on previous inputs or states (i.e., they have no memory). These circuits are built using logic gates (AND, OR, NOT, NAND, NOR, XOR, XNOR) and perform specific Boolean functions.
Combinational logic is often grouped into larger building blocks to build more complex systems Other basic Combinational Logic Circuits are

- Arithmetic and Logic Functions (Adders, Substractors, Comparitors, PLDs)
- Data Transmission (Multiplexers, Demultiplexers, Encoders, Decorders)
- Code Converters (Binary to BCD), 7-Segment)

5.1 Inverter

```
module inv (
    input [3:0] a,
    output reg [3:0] y
    );

    always @ (*)
        y = ~a;

endmodule
```

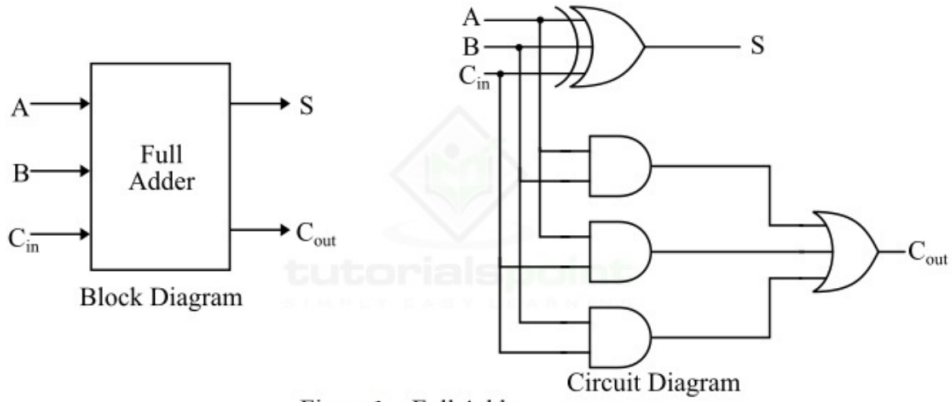
5.2 Full Adder

5.2.1 1-bit Full Adder

Truth Table

Carry In	Input A	Input B	Carry Out	Output S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Gate Level Circuit Diagram



Verilog Code 1

```verilog
module fulladder (
    input a, b, cin,

    output reg s, cout
    );

    reg p, g;

    always @ (*) begin
        p = a ^ b; // blocking
        g = a & b; // blocking
        s = p ^ cin; // blocking
        cout = g | (p & cin); // blocking
    end

    endmodule
```

Verilog Code 2

```verilog
module fulladder (
    input a,
    input b,
    input cin,
    output s,
    output cout
    );

    assign {cout, s}= cin + a + b;

endmodule
```

5.2.2 8-bit Full Adder

Module

```verilog
module fulladder8b(
        input  wire [7:0] a,
        input  wire [7:0] b,
        input  wire cin,
        output wire [7:0] s,
        output wirw cout
    );

assign {cout, s} = a + b + cin;

endmodule
```

Testbench

Self Study

5.2.3 N-Bit Full Adder (Parameterized)

Module

```
module fulladderNb #(parameter N = 4)  // Default width = 4 bits
        (
            input  [N-1:0] a, b,  // N-bit inputs
            input          cin,   // Carry-in
            output [N-1:0] sum,   // N-bit sum
            output         cout   // Carry-out
        );

    assign {cout, sum} = a + b + cin;  // Simple behavioral addition

endmodule
```

Testbench

```
`timescale 1ns/1ps
`include "fulladderNb.v"

module tb ();

    parameter N = 16;  // Testbench also parameterized

    reg  [N-1:0] a, b;
    reg          cin;
    wire [N-1:0] sum;
    wire         cout;

    integer i,j;
    // Instantiate the N-bit adder
    fulladderNb #(.N(N)) uut (
                                .a(a),
                                .b(b),
                                .cin(cin),
                                .sum(sum),
                                .cout(cout)
                            );

    // Stimulus generation
    //initial begin
    always @ (*) begin

        // Test all combinations for small N (e.g., N=4)
        for ( i = 0; i < (1 << N); i = i + 1) begin
            for (j = 0; j < (1 << N); j = j + 1) begin
                a = i;
                b = j;
                cin = 0;  // Test cin=0
                #10;
                $display("a=%b, b=%b, cin=%b, sum=%b, cout=%b", a, b, cin, sum, cout
                    );

                cin = 1;  // Test cin=1
                #10;
                $display("a=%b, b=%b, cin=%b, sum=%b, cout=%b", a, b, cin, sum, cout
                    );
            end
        end

        // Edge cases
        a = {N{1'b1}};  // All 1s (max value)
```

```
        b = {N{1'b1}};
        cin = 1;
        #10;
        $display("Edge Case: a=%b, b=%b, cin=%b, sum=%b, cout=%b", a, b, cin, sum,
            cout);

        $finish;
    end
endmodule
```

5.3 Multiplexers

5.3.1 2:1 Multiplexer

- 2 x inputs a and b (of 1 bit each)
- 1-bit Select input se
- 1-bit Chip Enable input en

Module

```
module mux_2to1(
input  wire a, // Input A
input  wire b, // input B
input  wire se, // Select
input  wire en, // 1/0=Enables/disables chip
output wire y  // Output
);

assign y = en ? (se ? a : b) : 1'bz;

endmodule
```

Testbench

```
`timescale 1ns/1ps
`include "mux_2to1.v"

module tb();

    reg a;
    reg b;
    reg se;
    reg en;
    wire y;

    mux_2to1 uut ( .a(a), .b(b), .se(se), .en(en), .y(y) );

initial begin
    // Dump waveform data (for GTKWave)
    $dumpfile("dump.vcd");  // Create a VCD file
    $dumpvars(0, tb);       // Dump all variables
    en=1;
    a = 1'b1; b = 1'b0; se = 1'b1;
    #10; a = 1'b0; b = 1'b1; se = 1'b1;
    #10; a = 1'b1; b = 1'b0; se = 1'b1;
    #10; a = 1'b1; b = 1'b0; se = 1'b0;
    #10; a = 1'b0; b = 1'b1; se = 1'b0;

    en=0;
    a = 1'b1; b = 1'b0; se = 1'b1;
    #10; a = 1'b0; b = 1'b1; se = 1'b1;
    #10; a = 1'b1; b = 1'b0; se = 1'b1;
```

```
    #10; a = 1'b1; b = 1'b0; se = 1'b0;
    #10; a = 1'b0; b = 1'b1; se = 1'b0;

    end

initial begin
    $monitor("en:%1b, se:%1b, ab:%1b%1b, y:%1b", en, se, a, b, y);
end

endmodule
```

5.3.2 4:1 Multiplexer

- D0-3: 4 x Inputs (each 1-bit)
- SE: 1 x SE Input (2-bits)
- EN: 1 x EN Input (Chip Enable bit)
- Y: 1 x Output

Module

```
module mux_4to1(
    input  wire [3:0] d,  // Input D3-0
    input  wire [1:0] se, // 0/1/2/3 Select A/B/C/D
    input  wire en, // 1 - Chip Enable
    output reg y  // Output
    );

    always @ (*) begin
        if (en) begin
            case (se)
            2'b00: y = d[0];
            2'b01: y = d[1];
            2'b10: y = d[2];
            2'b11: y = d[3];
            default: y = 1'bz;
            endcase
        end
        else
            y=1'bz;
    end

endmodule
```

Testbench

```
`timescale 1ns/1ps
`include "mux-4to1.v"

module tb();

    reg [3:0] d;
    reg [1:0] se;
    reg en;
    wire y;

    mux_4to1 uut (
            .d(d),
            .se(se),
            .en(en),
            .y(y)
        );

    initial begin
```

```
        // Dump waveform data (for GTKWave)
        $dumpfile("dump.vcd");  // Create a VCD file
        $dumpvars(0, tb);       // Dump all variables

        d = 4'b0001;

        se = 2'b00; en = 1'b1;
        #10; se = 2'b01;
        #10; se = 2'b10;
        #10; se = 2'b11;

        #10; d = 4'b0010;

        #10; se = 2'b00;
        #10; se = 2'b01;
        #10; se = 2'b10;
        #10; se = 2'b11;

        #10; d = 4'b0100; se = 2'b10;

        #10; se = 2'b00;
        #10; se = 2'b01;
        #10; se = 2'b10;
        #10; se = 2'b11;

        #10; d = 4'b1000; se = 2'b11;

        #10; se = 2'b00;
        #10; se = 2'b01;
        #10; se = 2'b10;
        #10; se = 2'b11;
    end

    initial begin
        $monitor("en(%1b), se(%2b), abcd(%4b), y(%1b)", en, se, d, y);
    end
endmodule
```

5.4 Demultiplexers

Self Study

5.5 Decoders

5.5.1 3:8 Decoder

Module

```
    module decoder3to8(
    input wire [2:0] a,
    input wire en,
    output reg [7:0] y
    );

    always @ (a) begin
    if(en)
    case (a)
    3'b000: y = 8'b0000_0001;
    3'b001: y = 8'b0000_0010;
    3'b010: y = 8'b0000_0100;
    3'b011: y = 8'b0000_1000;
    3'b100: y = 8'b0001_0000;
    3'b101: y = 8'b0010_0000;
    3'b110: y = 8'b0100_0000;
    3'b111: y = 8'b1000_0000;
```

```
    default : y = 8'bzzzz_zzzz;
    endcase
    else
    y = 8'bzzzz_zzzz;
    end

    endmodule
```

Testbench

5.5.2 $N : 2^N$ Decoder (Parameterized)

Module

```
module decoderN #( parameter N = 3)
    (
        input wire [N-1:0] a,
        input wire en,
        output reg [2**N-1:0] y   // equvivalent [1 << N-1:0]
    );

    always @* begin
        if(en) begin
            y = 0;                // Default all outputs to 0
            y[a] = 1'b1;         // Set the corresponding output bit to 1
        end
        else
            y='bz;
    end

endmodule
```

Testbench

```
`timescale 1ns/1ps
`include "decoderN.v"

module tb();

    parameter N = 3;  // Test 2:4 decoder
    reg [N-1:0] a;
    reg en;
    wire [(2**N)-1:0] y;

    integer i;
    // Instantiate the decoder
    decoderN #(.N(N)) dut (.a(a), .en(en), .y(y));

    initial begin
        // Test all input combinations
        en = 1;
        for (i = 0; i < 1<<N; i = i + 1) begin
            a = i;
            #10;
        end

        en = 0;
        for (i = 0; i < 1<<N; i = i + 1) begin
            a = i;
            #10;
        end
    end

initial begin
$monitor("Time = %5t: en=%1b, in=%b, out=%b, i=%2d", $time, en, a, y,i);
end
```

```
endmodule
```

5.5.3 7-Segment Display Decoder

Module

```
module sevenseg (
    input [3:0] data,
    output reg [6:0] segments);

    always @ (*)

        case (data)
        // abc_defg
        0: segments = 7'b111_1110;
        1: segments = 7'b011_0000;
        2: segments = 7'b110_1101;
        3: segments = 7'b111_1001;
        4: segments = 7'b011_0011;
        5: segments = 7'b101_1011;
        6: segments = 7'b101_1111;
        7: segments = 7'b111_0000;
        8: segments = 7'b111_1111;
        9: segments = 7'b111_1011;
        default: segments = 7'b000_0000;
        endcase

endmodule
```

Testbench

Self Study

5.6 Encoders

Self Study

Session 6

Sequential Logic Circuits

Sequential Logic Circuits are digital circuits whose outputs depend not only on the current inputs but also on the sequence of past inputs (i.e., they have memory). Unlike combinational logic circuits (where outputs depend only on current inputs), sequential circuits incorporate feedback loops to store previous states.

Basic Sequential Logic Circuits

- Flip-Flops
  - D Flip-Flop (DFF) – The most fundamental sequential element
  - JK Flip-Flop (Less common in modern designs but useful for learning)
  - T Flip-Flop (Toggle Flip-Flop)
- Latches
  - D Latch
  - SR Latch
- Registers
  - Basic N-bit Register (Parallel-in, Parallel-out)
  - Shift Registers:
  - Serial-In, Serial-Out (SISO)
  - Serial-In, Parallel-Out (SIPO)
  - Parallel-In, Serial-Out (PISO)
  - Parallel-In, Parallel-Out (PIPO)
  - Universal Shift Register (Bidirectional with parallel load)
- Counters
  - Binary Up Counter
  - Binary Down Counter
  - Up/Down Counter
  - Modulo-N Counter (e.g., 0 to $N-1$)
  - Ring Counter
  - Johnson Counter (Twisted Ring Counter)

Intermediate Sequential Circuits

- Finite State Machines (FSMs)
  - Moore Machine
  - Mealy Machine
  - State Encoding Techniques (Binary, One-Hot, Gray Code)
- Memory Elements
  - Single-Port RAM

  - Dual-Port RAM (Basic)
  - FIFO (First-In-First-Out Buffer)
  - Synchronous and Asynchronous
  - LIFO (Last-In-First-Out / Stack)
- Synchronizers and Metastability Handling
  - Two-Flop Synchronizer
  - Handshake-based synchronization

Advanced Sequential Circuits

- Sequence Detectors (Using FSMs)
  - Overlapping and Non-overlapping sequence detectors
- Clock Domain Crossing (CDC) Circuits
  - MUX-based CDC handling
  - FIFO-based CDC
- Pipelined Designs
  - Basic 2-stage/3-stage pipelines
  - Hazard handling (e.g., data forwarding, stalling)
- Arithmetic Sequential Circuits
  - Sequential Multiplier
  - Accumulator
- Programmable Timers and PWM Generators
- CPUs

6.1 D Latch

Module

```
/* D-Latch
- D Data input, EN- Input Enable, Q-Output
- Output Q follows input D, When EN is HIGH (Level Sensitive)
- Output holds its last value, when EN is LOW
*/
module dlatch1b(
        input wire d,       // D data input
        input wire en,      // EN enable
        output reg q        // Q output
    );

    always @ (en or d) begin
        if (en) begin
            q <= d;
        end
    end
endmodule
```

Test Bench

```
`timescale 1ns/1ps
`include "dlatch1b.v"

module tb();

    reg d;
    reg en;
    wire  q;

    dlatch1b uut (
        .d(d),
        .en(en),
        .q(q)
    );

    initial begin
        // Dump waveform data (for GTKWave)
        $dumpfile("dump.vcd");  // Create a VCD file
        $dumpvars(0, tb);       // Dump all variables

        en = 1'b1; d = 1'b0;
        #10;             d = 1'b1;
        #10;             d = 1'b0;
        #5 ;             d = 1'b1;
        #5 ;             d = 1'b0;
        #10; en = 1'b0; d = 1'b1;
        #10;             d = 1'b0;
        #10;             d = 1'b1;

    end

    initial begin
        $monitor("en=%1b, d=%1b, q=%1b", en, d, q);
    end
endmodule
```

6.2 D Flip Flop

Module

```
/* D-Flip-Flop
- D Data input, CLK- Clock Input, RESET reent input, Q-Output
- Output Q is set to input D, at the rising edge of CLK
- Output holds its last value, for all other changes in CLK
-

*/
module dflipflop(
    input wire d,       // D data input
    input wire clk,     // CLK clock input
    input wire en,      // Enable (the IC)
    input wire reset,   // Asynchronous/Synchronous RESET (active-high)

    output reg q        // Q output
    );

    //always @ (posedge clk) begin  // Synchronous (with POsitive edge of CLK) RESET (active-high)
    always @ (posedge clk or posedge reset) begin  // Asynchronous RESET (active-high)
        if (reset)   q <= 0;
        else if(en) begin
            q <= d;
        end
```

```
    end

endmodule
```

Test Bench

```
`timescale 1ns/1ps
`include "dflipflop.v"

module tb();

    reg d;
    reg clk;
    reg en;
    reg reset;
    wire q;

    dflipflop uut (
        .d(d),
        .clk(clk),
        .en(en),
        .reset(reset),
        .q(q)
    );

    initial begin
        // Dump waveform data (for GTKWave)
        $dumpfile("dump.vcd");  // Create a VCD file
        $dumpvars(0, tb);       // Dump all variables
    end

    initial begin
        clk <= 0;
        forever begin
            #5; clk <= ~clk;
        end
    end

    initial begin
        en=1; reset = 0; d = 0;
        #3;  d = 1;
        #9;  d = 0;
        #6;  d = 1;
        #10; d = 0;
        #20; d = 1;
        #23; reset = 1;
        #5;  reset = 0;
        #10;
        en = 0;
        #3;  d = 1;
        #9;  d = 0;
        #6;  d = 1;
        #23; reset = 1;
        #5;  reset = 0;
        #10;
        $finish;
    end

    initial begin
        $monitor("t=%3d, reset=%1b, en=%1b, clk=%1b, d=%1b, q=%1b", $time, reset, en
            , clk, d, q);
    end
    endmodule
```

—-