

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.

```
keywordstyle
1 module myfirstprog();
2   initial begin
3     $display("Hello, World");
4     $finish
5   end
6 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

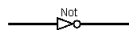
Inverter Design with Verilog - Getting Started

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.

```
keywordstyle
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.

```
keywordstyle
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.

```
keywordstyle
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

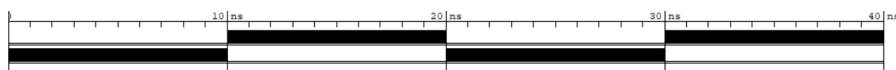
- **DUT (Design Under Test):** The module being tested (e.g., adder, FSM, processor)
- **Test Stimulus Generator:** Produces input signals (**reg** variables) for the DUT.
- **Clock & Reset Generation:** Simulates clock signals for sequential circuits.
- **Monitor & Checker:** Displays results (**\$display**,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

```
keywordstyle
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 `keywordstyleinitial` functions?

E 2.4 How procedural block `keywordstylealways` functions?

E 2.5 What is the default variable type in Verilog?

E 2.6 What are the differences between `keywordstylewire` and `keywordstylereg`?

<https://chatgpt.com/share/69541bf3-abcc-800c-9dab-d8>

Session 3

Comparator Design with Verilog

3.1 1-bit Comparator

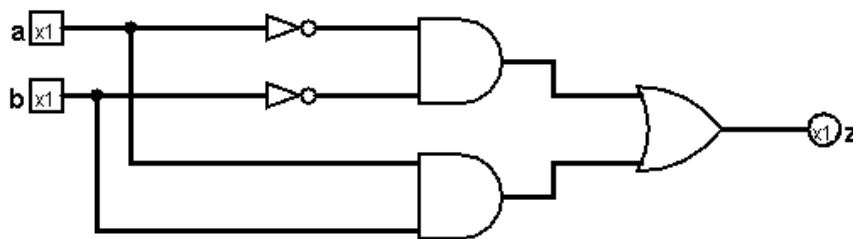
3.1.1 RTL Modeling

Truth Table

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

Gate Level Circuit Diagram

1-bit Comparator



Verilog Code

```
keywordstyle
1 module comp1b
2   (   input a,
3     input b,
4     output c
5   );
6
7   assign c = (~a & ~b) | (a & b);
8
9 endmodule
```

Test Bench

```
keywordstyle
1 `timescale 1ns/1ps
2 `include "comp1b.v"
3
4 module comp1b_tb;
5   reg a;
6   reg b;
7   wire c;
8
9   // Instantiate the unit under test
10  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

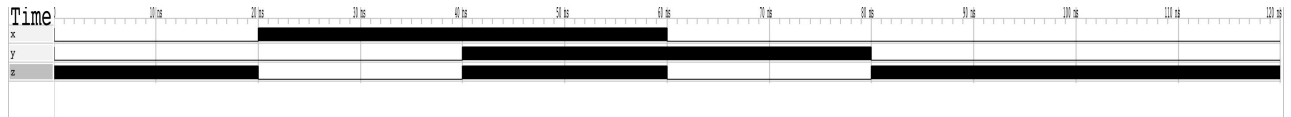
Terminal output

```

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

```

keywordstyle
1 module compib(
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

```

keywordstyle
1 module compib(
2     input wire a,
3     input wire b,
4     output reg c
5 );
6
7     always @(*) begin
8         if(a==b) c = 1;
9         else c = 0;
10    end
11
12 endmodule

```

3.1.4 Modeling with User Define Primitives

Verilog Code

```

keywordstyle
1 module compib(input a, input b, output z);
2     compare c0(c, a,b);
3 endmodule
4
5 primitive compare(output out, input in1, input in2);
6
7     table
8         //in1,in2:out
9         0  0  :  1;
10        0  1  :  0;
11        1  0  :  0;
12        1  1  :  1;
13    endtable
14
15 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

```
keywordstyle
1 // Full 1-bit Comparator
2 // 2 x Input (a, b), 3 x Outputs (a_eq_b, a_gt_b, a_ls_b)
3
4 module compfull1b(
5     input wire a,
6     input wire b,
7     output reg a_eq_b, // a = b
8     output reg a_gt_b, // a > b
9     output reg a_lt_b // a < b
10 );
11
12     always @(*) begin
13
14         if (a==b) begin
15             a_eq_b = 1'b1;
16             a_gt_b = 1'b0;
17             a_lt_b = 1'b0;
18         end
19         else if (a>b) begin
20             a_eq_b = 1'b0;
21             a_gt_b = 1'b1;
22             a_lt_b = 1'b0;
23         end
24         else begin
25             a_eq_b = 1'b0;
26             a_gt_b = 1'b0;
27             a_lt_b = 1'b1;
28         end
29     end
30 end
31
32 endmodule
```

Testbench

```
keywordstyle
1 `timescale 1ns/1ps
2 `include "compfull1b.v"
3
4 module tb();
5
6     reg a;
7     reg b;
8     wire a_eq_b;
9     wire a_gt_b;
10    wire a_ls_b;
11
12    compfull1b uut (
13        .a(a),
14        .b(b),
15        .a_eq_b(a_eq_b),
16        .a_gt_b(a_gt_b),
17        .a_lt_b(a_lt_b)
18    );
19
20    initial begin
21        // Dump waveform data (for GTKWave)
22        $dumpfile("dump.vcd"); // Create a VCD file
23        $dumpvars(0, tb); // Dump all variables
24
25        a = 1'b0; b = 1'b0;
26        #10; a = 1'b0; b = 1'b1;
27        #10; a = 1'b1; b = 1'b1;
28        #10; a = 1'b1; b = 1'b0;
29        #10; a = 1'b0; b = 1'b0;
30        #10;
31    end
32
33    initial begin
34        $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);
35    end
36 endmodule
```

```

35     end
36
37 endmodule

```

3.4 2 bit Comparator with 3 x Output

Verilog Code

```

keywordstyle
1 module comp2bfull(
2     input wire [1:0] a,
3     input wire [1:0] b,
4     output reg a_eq_b, // a = b
5     output reg a_gt_b, // a > b
6     output reg a_lt_b  // a < b
7 );
8
9 always @(*) begin
10     if (a==b) begin
11         a_eq_b = 1'b1; a_gt_b = 1'b0; a_lt_b = 1'b0;
12     end
13     else if (a>b) begin
14         a_eq_b = 1'b0; a_gt_b = 1'b1; a_lt_b = 1'b0;
15     end
16     else begin
17         a_eq_b = 1'b0; a_gt_b = 1'b0; a_lt_b = 1'b1;
18     end
19 end
20
21 endmodule

```

Test Bench

```

keywordstyle
1 `timescale 1ns/1ps
2 `include "comp2bfull.v"
3
4 module tb();
5
6     reg [1:0] a;
7     reg [1:0] b;
8     wire a_eq_b;
9     wire a_gt_b;
10    wire a_lt_b;
11
12    comp2bfull uut (
13        .a(a),
14        .b(b),
15        .a_eq_b(a_eq_b),
16        .a_gt_b(a_gt_b),
17        .a_lt_b(a_lt_b)
18    );
19
20    initial begin
21        // Dump waveform data (for GTKWave)
22        $dumpfile("dump.vcd"); // Create a VCD file
23        $dumpvars(0, tb);      // Dump all variables
24
25        a = 2'b00; b = 2'b00;
26        #10;      b = 2'b01;
27        #10;      b = 2'b10;
28        #10;      b = 2'b11;
29
30        a = 2'b01; b = 2'b00;
31        #10;      b = 2'b01;
32        #10;      b = 2'b10;
33        #10;      b = 2'b11;
34
35        a = 2'b10; b = 2'b00;
36        #10;      b = 2'b01;
37        #10;      b = 2'b10;
38        #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

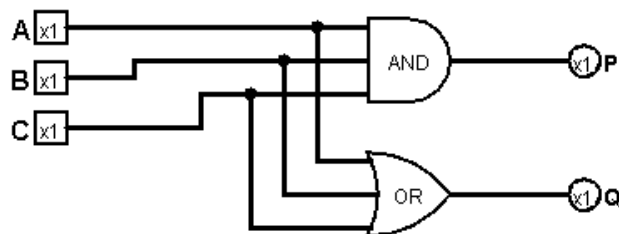
Simple IO and Buses

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

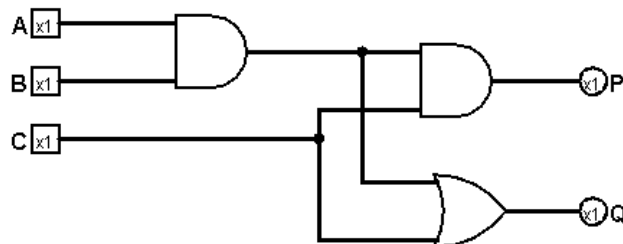
Simple In and Out



```
keywordstyle
1 module simpleio(
2   input wire a,
3   input wire b,
4   input wire c,
5   output wire p,
6   output wire q
7 );
8
9   assign p = a & b & c;
10  assign q = a | b | c;
11
12 endmodule
```

4.1.2 Intermediate Wire

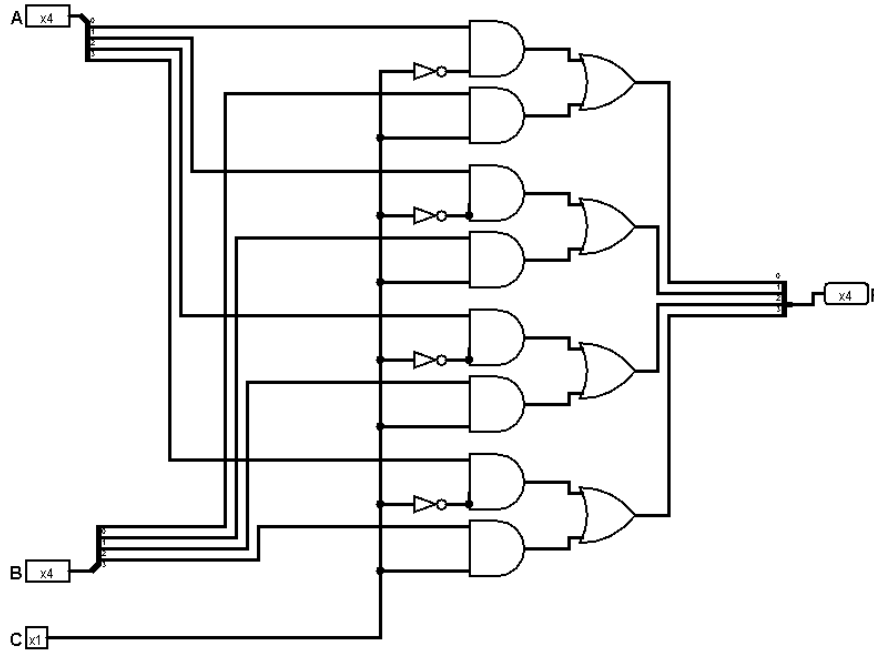
Intermediate Wire



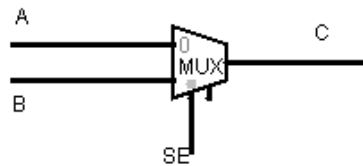
```
keywordstyle
1 module intmwire(
2   input wire a,
3   input wire b,
4   input wire c,
5   output wire p,
6   output wire q
7 );
8
9   wire intmwire;
10
11  assign intmwire = a & b;
12  assign p = intmwire & c;
13  assign q = intmwire | c;
14
15 endmodule
```

4.1.3 Bus Signals

Bus Signals



```
keywordstyle
1 module bussignals(
2   input wire [3:0] a,
3   input wire [3:0] b,
4   input wire c,
5   output wire p
6 );
7
8   wire [3:0] cbus;
9
10  assign cbus = {4{c}};
11
12  assign p = ( a & (~cbus) | ( b & cbus ) ;
13
14 endmodule
```

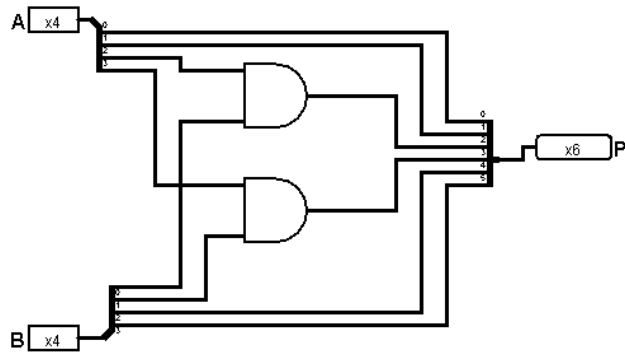


4.1.4 Simple Multiplexer

```
keywordstyle
1 module simplemux(
2   input wire [3:0] a, // Input 1
3   input wire [3:0] b, // Input 2
4   input wire se,      // Select
5   output wire [3:0] c // Output
6 );
7
8   assign c = se ? a : b;
9
10 endmodule
```

4.1.5 Bus Breakout

Bus Breakout



```
keywordstyle
1 module busbreakout(
2   input wire [3:0] a,
3   input wire [3:0] b,
4   output wire [5:0] p
5 );
6
7   assign p = { b[3:2], (a[3] & b[1]), (a[2] & b[0]), a[1:0]};
8
9 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, Subtractors, Comparitors, PLDs)
- Data Transmission (Multiplexers, Demultiplexers, Encoders, Decoders)
- Code Converters (Binary to BCD), 7-Segment)

5.1 Inverter

```
keywordstyle
1 module inv (
2     input [3:0] a,
3     output reg [3:0] y
4 );
5
6     always @ (*)
7         y = ~a;
8
9 endmodule
```

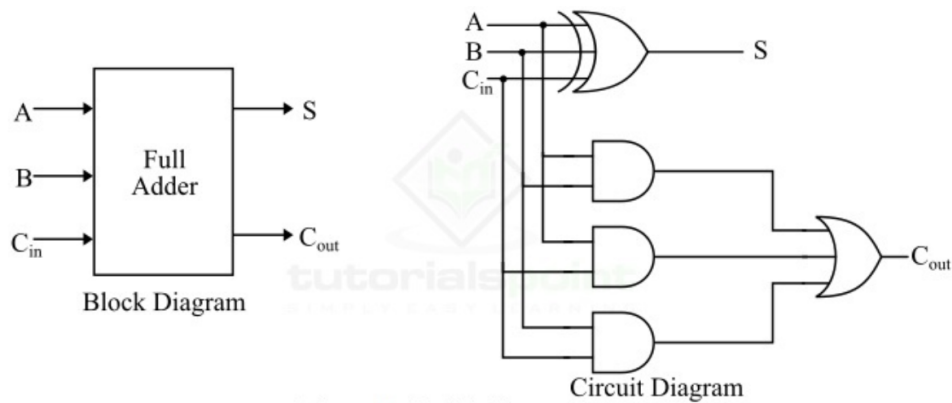
5.2 Full Adder

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

```
keywordstyle
1 module fulladder (
2     input a, b, cin,
3
4     output reg s, cout
5 );
6
7     reg p, g;
8
9     always @ (*) begin
10         p = a ^ b; // blocking
11         g = a & b; // blocking
12         s = p ^ cin; // blocking
13         cout = g | (p & cin); // blocking
14     end
15
16 endmodule
```

Verilog Code 2

```
keywordstyle
1 module fulladder (
2     input a,
3     input b,
4     input cin,
5     output s,
6     output cout
7 );
8
9     assign {cout, s} = cin + a + b;
10
11 endmodule
```

5.2.2 8-bit Full Adder

Module

```
keywordstyle
1 module fulladder8b(
2     input wire [7:0] a,
3     input wire [7:0] b,
4     input wire cin,
5     output wire [7:0] s,
6     output wire cout
7 );
8
9     assign {cout, s} = a + b + cin;
10
11 endmodule
```

Testbench

Self Study

5.2.3 N-Bit Full Adder (Parameterized)

Module

```
keywordstyle
1 module fulladderNb #(parameter N = 4) // Default width = 4 bits
2 (
3     input [N-1:0] a, b, // N-bit inputs
4     input cin, // Carry-in
5     output [N-1:0] sum, // N-bit sum
6     output cout // Carry-out
7 );
8
9     assign {cout, sum} = a + b + cin; // Simple behavioral addition
10
11 endmodule
```

Testbench

```
keywordstyle
1 'timescale 1ns/1ps
2 'include "fulladderNb.v"
3
4 module tb ();
5
6     parameter N = 16; // Testbench also parameterized
7
8     reg [N-1:0] a, b;
9     reg      cin;
10    wire [N-1:0] sum;
11    wire      cout;
12
13    integer i,j;
14    // Instantiate the N-bit adder
15    fulladderNb #(N(N)) uut (
16        .a(a),
17        .b(b),
18        .cin(cin),
19        .sum(sum),
20        .cout(cout)
21    );
22
23    // Stimulus generation
24    //initial begin
25    always @ (*) begin
26
27        // Test all combinations for small N (e.g., N=4)
28        for ( i = 0; i < (1 << N); i = i + 1) begin
29            for (j = 0; j < (1 << N); j = j + 1) begin
30                a = i;
31                b = j;
32                cin = 0; // Test cin=0
33                #10;
34                $display("a=%b, b=%b, cin=%b, sum=%b, cout=%b", a, b, cin, sum, cout);
35
36                cin = 1; // Test cin=1
37                #10;
38                $display("a=%b, b=%b, cin=%b, sum=%b, cout=%b", a, b, cin, sum, cout);
39            end
40        end
41
42        // Edge cases
43        a = {N{1'b1}}; // All 1s (max value)
44        b = {N{1'b1}};
45        cin = 1;
46        #10;
47        $display("Edge Case: a=%b, b=%b, cin=%b, sum=%b, cout=%b", a, b, cin, sum, cout);
48
49        $finish;
50    end
51 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

```
keywordstyle
1 module mux_2to1(
2     input wire a, // Input A
3     input wire b, // Input B
4     input wire se, // Select
5     input wire en, // 1/0=Enables/disables chip
6     output wire y // Output
7 );
8
9 assign y = en ? (se ? a : b) : 1'bz;
10
11 endmodule
```

Testbench

```
keywordstyle
1 'timescale 1ns/1ps
2 'include "mux_2to1.v"
3
4 module tb();
5
6     reg a;
7     reg b;
8     reg se;
9     reg en;
10    wire y;
11
12    mux_2to1 uut ( .a(a), .b(b), .se(se), .en(en), .y(y) );
13
14    initial begin
15        // Dump waveform data (for GTKWave)
16        $dumpfile("dump.vcd"); // Create a VCD file
17        $dumpvars(0, tb); // Dump all variables
18        en=1;
19        a = 1'b1; b = 1'b0; se = 1'b1;
```

```

20 #10; a = 1'b0; b = 1'b1; se = 1'b1;
21 #10; a = 1'b1; b = 1'b0; se = 1'b1;
22 #10; a = 1'b1; b = 1'b0; se = 1'b0;
23 #10; a = 1'b0; b = 1'b1; se = 1'b0;
24
25 en=0;
26 a = 1'b1; b = 1'b0; se = 1'b1;
27 #10; a = 1'b0; b = 1'b1; se = 1'b1;
28 #10; a = 1'b1; b = 1'b0; se = 1'b1;
29 #10; a = 1'b1; b = 1'b0; se = 1'b0;
30 #10; a = 1'b0; b = 1'b1; se = 1'b0;
31
32 end
33
34 initial begin
35     $monitor("en:%1b, se:%1b, ab:%1b%1b, y:%1b", en, se, a, b, y);
36 end
37
38 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

```

keywordstyle
1 module mux_4to1(
2     input wire [3:0] d, // Input D3-0
3     input wire [1:0] se, // 0/1/2/3 Select A/B/C/D
4     input wire en, // 1 - Chip Enable
5     output reg y // Output
6 );
7
8     always @(*) begin
9         if (en) begin
10             case (se)
11                 2'b00: y = d[0];
12                 2'b01: y = d[1];
13                 2'b10: y = d[2];
14                 2'b11: y = d[3];
15                 default: y = 1'bz;
16             endcase
17         end
18         else
19             y=1'bz;
20         end
21     end
22 endmodule

```

Testbench

```

keywordstyle
1 `timescale 1ns/1ps
2 `include "mux-4to1.v"
3
4 module tb();
5
6     reg [3:0] d;
7     reg [1:0] se;
8     reg en;
9     wire y;
10
11     mux_4to1 uut (
12         .d(d),
13         .se(se),
14         .en(en),
15         .y(y)
16     );
17
18     initial begin
19         // Dump waveform data (for GTKWave)
20         $dumpfile("dump.vcd"); // Create a VCD file
21         $dumpvars(0, tb); // Dump all variables
22
23         d = 4'b0001;
24
25         se = 2'b00; en = 1'b1;
26         #10; se = 2'b01;
27         #10; se = 2'b10;
28         #10; se = 2'b11;
29
30         #10; d = 4'b0010;
31
32         #10; se = 2'b00;
33         #10; se = 2'b01;
34         #10; se = 2'b10;
35         #10; se = 2'b11;
36
37         #10; d = 4'b0100; se = 2'b10;
38
39         #10; se = 2'b00;
40         #10; se = 2'b01;
41         #10; se = 2'b10;
42         #10; se = 2'b11;
43
44         #10; d = 4'b1000; se = 2'b11;
45
46         #10; se = 2'b00;
47         #10; se = 2'b01;
48         #10; se = 2'b10;

```

```

49     #10; se = 2'b11;
50 end
51
52 initial begin
53     $monitor("en(%1b), se(%2b), abcd(%4b), y(%1b)", en, se, d, y);
54 end
55 endmodule

```

5.4 Demultiplexers

Self Study

5.5 Decoders

5.5.1 3:8 Decoder

Module

```

keywordstyle
1 module decoder3to8(
2     input wire [2:0] a,
3     input wire en,
4     output reg [7:0] y
5 );
6
7 always @ (a) begin
8     if(en)
9         case (a)
10            3'b000: y = 8'b0000_0001;
11            3'b001: y = 8'b0000_0010;
12            3'b010: y = 8'b0000_0100;
13            3'b011: y = 8'b0000_1000;
14            3'b100: y = 8'b0001_0000;
15            3'b101: y = 8'b0010_0000;
16            3'b110: y = 8'b0100_0000;
17            3'b111: y = 8'b1000_0000;
18            default : y = 8'bzzzz_zzzz;
19        endcase
20    else
21        y = 8'bzzzz_zzzz;
22    end
23 end
24 endmodule

```

Testbench

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

Module

```

keywordstyle
1 module decoderN #( parameter N = 3)
2 (
3     input wire [N-1:0] a,
4     input wire en,
5     output reg [2*N-1:0] y // equivalent [1 << N-1:0]
6 );
7
8 always @* begin
9     if(en) begin
10         y = 0; // Default all outputs to 0
11         y[a] = 1'b1; // Set the corresponding output bit to 1
12     end
13     else
14         y='bz;
15 end
16
17 endmodule

```

Testbench

```

keywordstyle
1 `timescale 1ns/1ps
2 `include "decoderN.v"
3
4 module tb();
5
6     parameter N = 3; // Test 2:4 decoder
7     reg [N-1:0] a;
8     reg en;
9     wire [(2*N)-1:0] y;
10
11     integer i;
12     // Instantiate the decoder
13     decoderN #(N(N)) dut (.a(a), .en(en), .y(y));
14
15     initial begin
16         // Test all input combinations
17         en = 1;
18         for (i = 0; i < 1<<N; i = i + 1) begin
19             a = i;
20             #10;
21         end
22
23         en = 0;
24         for (i = 0; i < 1<<N; i = i + 1) begin
25             a = i;
26             #10;

```

```

27         end
28     end
29
30 initial begin
31     $monitor("Time = %5t: en=%1b, in=%b, out=%b, i=%2d", $time, en, a, y,i);
32 end
33 endmodule

```

5.5.3 7-Segment Display Decoder

Module

```

keywordstyle
1 module sevenseg (
2     input [3:0] data,
3     output reg [6:0] segments);
4
5     always @ (*)
6
7         case (data)
8             // abc_defg
9             0: segments = 7'b111_1110;
10            1: segments = 7'b011_0000;
11            2: segments = 7'b110_1101;
12            3: segments = 7'b111_1001;
13            4: segments = 7'b011_0011;
14            5: segments = 7'b101_1011;
15            6: segments = 7'b101_1111;
16            7: segments = 7'b111_0000;
17            8: segments = 7'b111_1111;
18            9: segments = 7'b111_1011;
19            default: segments = 7'b000_0000;
20            endcase
21
22 endmodule

```

Testbench

Self Study

5.6 Encoders

Self Study

Session 6

Basic 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

```
keywordstyle
1  /* D-Latch
2  - D Data input, EN- Input Enable, Q-Output
3  - Output Q follows input D, When EN is HIGH (Level Sensitive)
4  - Output holds its last value, when EN is LOW
5  */
6  module dlatch1b(
7      input wire d,          // D data input
8      input wire en,         // EN enable
9      output reg q           // Q output
10 );
11
12 always @ (en or d) begin
13     if (en) begin
14         q <= d;
15     end
16 end
17 endmodule
```

Test Bench

```
keywordstyle
1  `timescale 1ns/1ps
2  `include "dlatch1b.v"
3
4  module tb();
5
6      reg d;
7      reg en;
8      wire q;
9
10     dlatch1b uut (
11         .d(d),
12         .en(en),
13         .q(q)
14     );
15
16     initial begin
17         // Dump waveform data (for GTKWave)
18         $dumpfile("dump.vcd"); // Create a VCD file
19         $dumpvars(0, tb);      // Dump all variables
20
21         en = 1'b1; d = 1'b0;
22         #10; d = 1'b1;
23         #10; d = 1'b0;
```

```

24         #5 ;           d = 1'b1;
25         #5 ;           d = 1'b0;
26         #10; en = 1'b0; d = 1'b1;
27         #10;           d = 1'b0;
28         #10;           d = 1'b1;
29
30     end
31
32     initial begin
33         $monitor("en=%1b, d=%1b, q=%1b", en, d, q);
34     end
35 endmodule

```

6.2 D Flip Flop

Module

```

keywordstyle
1  /* D-Flip-Flop
2  - D Data input, CLK- Clock Input, RESET reent input, Q-Output
3  - Output Q is set to input D, at the rising edge of CLK
4  - Output holds its last value, for all other changes in CLK
5  -
6
7  */
8  module dflipflop(
9      input wire d,          // D data input
10     input wire clk,        // CLK clock input
11     input wire en,         // Enable (the IC)
12     input wire reset,      // Asynchronous/Synchronous RESET (active-high)
13
14     output reg q           // Q output
15 );
16
17 //always @ (posedge clk) begin // Synchronous (with P0sitive edge of CLK) RESET (active-high)
18 always @ (posedge clk or posedge reset) begin // Asynchronous RESET (active-high)
19     if (reset) q <= 0;
20     else if(en) begin
21         q <= d;
22     end
23 end
24
25 endmodule

```

Test Bench

```

keywordstyle
1  'timescale 1ns/1ps
2  'include "dflipflop.v"
3
4  module tb();
5
6      reg d;
7      reg clk;
8      reg en;
9      reg reset;
10     wire q;
11
12     dflipflop uut (
13         .d(d),
14         .clk(clk),
15         .en(en),
16         .reset(reset),
17         .q(q)
18     );
19
20     initial begin
21         // Dump waveform data (for GTKWave)
22         $dumpfile("dump.vcd"); // Create a VCD file
23         $dumpvars(0, tb);      // Dump all variables
24     end
25
26     initial begin
27         clk <= 0;
28         forever begin
29             #5; clk <= ~clk;
30         end
31     end
32
33     initial begin
34         en=0; reset = 0; d = 0;
35         #7 d=1;
36         #6 d=0;
37         #10 d=1;
38         #10 d=0;
39         #10 en=1;
40         #10 d=1;
41         #10 d=0;
42         #10 d=1;
43         #10 reset=1;
44         #10 d=0;
45         #10 en=0;
46         #10 d=1;;
47         #10 en=1;
48         #10 reset =0;
49         #10 d=0;
50         #10;
51         $finish;
52     end
53
54     initial begin
55         $monitor("t=%3d, reset=%1b, en=%1b, clk=%1b, d=%1b, q=%1b", $time, reset, en, clk, d, q);
56     end
57 endmodule

```

6.3 D Flip Flop - Parameterized

Module

```
keywordstyle
1  /* Parallel Input Parallel Output Register
2  * Width (Num Of Bits) is Parameterized
3  */
4  module pipo_reg #(parameter WIDTH = 8)
5  (
6      input wire clk,           // Clock input
7      input wire reset,        // Active-high reset
8      input wire load,         // Load enable signal
9      input wire [WIDTH-1:0] data_in, // Parallel data input
10     output reg [WIDTH-1:0] data_out // Parallel data output
11 );
12
13 always @(posedge clk or posedge reset) begin
14     if (reset) begin
15         // Reset all bits to 0
16         data_out <= {WIDTH{1'b0}};
17     end
18     else if (load) begin
19         // Load new data on rising clock edge when load is high
20         data_out <= data_in;
21     end
22     // If load is low, data_out retains its value
23 end
24
25 endmodule
```

Testbench

```
keywordstyle
1  `timescale 1ns/1ps
2  `include "nbitreg_pipo.v"
3
4  module tb;
5
6      // Parameters
7      localparam WIDTH = 8;
8      localparam CLK_PERIOD = 10;
9
10     // Signals
11     reg clk;
12     reg reset;
13     reg load;
14     reg [WIDTH-1:0] data_in;
15     wire [WIDTH-1:0] data_out;
16
17     // Instantiate the PIP0 register
18     pipo_reg #(.WIDTH(WIDTH)) uut (
19         .clk(clk),
20         .reset(reset),
21         .load(load),
22         .data_in(data_in),
23         .data_out(data_out)
24     );
25
26     // Clock generation
27     always #(CLK_PERIOD/2) clk = ~clk;
28
29     // Test procedure
30     initial begin
31         // Initialize inputs
32         clk = 0;
33         reset = 1;
34         load = 0;
35         data_in = 0;
36
37         // Apply reset
38         #20 reset = 0;
39
40         // Test load operation
41         #10 data_in = 8'hAA; load = 1;
42         #10 load = 0;
43
44         // Test another load
45         #20 data_in = 8'h55; load = 1;
46         #10 load = 0;
47
48         // Test without load
49         #20 data_in = 8'hFF;
50
51         // End simulation
52         #50 $finish;
53     end
54
55     // Monitor changes
56     initial begin
57         $monitor("Time = %0t, clk = %b, reset = %b, load = %b, data_in = %h, data_out = %h",
58             $time, clk, reset, load, data_in, data_out);
59     end
60
61 endmodule
```

6.4 Registers

6.4.1 Parallel In Parallel Out

Module

```
keywordstyle
1  /*
2  * Parallel Input Parallel Output Register
3  * Width (Num Of Bits) is Parameterized
```

```

4  */
5  module pipo_reg #(parameter WIDTH = 8)
6  (
7      input wire clk,           // Clock input
8      input wire reset,        // Active-high reset
9      input wire load,          // Load enable signal
10     input wire [WIDTH-1:0] data_in, // Parallel data input
11     output reg [WIDTH-1:0] data_out // Parallel data output
12 );
13
14     always @(posedge clk or posedge reset) begin
15         if (reset) begin
16             data_out <= {WIDTH{1'b0}}; // Reset all bits to 0
17         end
18         else if (load) begin
19             data_out <= data_in; // Load new data on rising clock edge when load is high
20         end
21         // If load is low, data_out retains its value
22     end
23
24 endmodule

```

Testbench

```

keywordstyle
1  'timescale 1ns/1ps
2  'include "pipo_reg.v"
3
4  module tb;
5
6      // Parameters
7      localparam WIDTH = 8;
8      localparam CLK_PERIOD = 10;
9
10     // Signals
11     reg clk;
12     reg reset;
13     reg load;
14     reg [WIDTH-1:0] data_in;
15     wire [WIDTH-1:0] data_out;
16
17     // Instantiate the PIP0 register
18     pipo_reg #(.WIDTH(WIDTH)) uut
19     (
20         .clk(clk),
21         .reset(reset),
22         .load(load),
23         .data_in(data_in),
24         .data_out(data_out)
25     );
26
27     // Clock generation
28     always #(.CLK_PERIOD/2) clk = ~clk;
29
30     initial begin
31         // Dump waveform data (for GTKWave)
32         $dumpfile("dump.vcd"); // Create a VCD file
33         $dumpvars(0, tb); // Dump all variables
34         end
35
36     // Test procedure
37     initial begin
38         // Initialize inputs
39         clk = 0;
40         reset = 1;
41         load = 0;
42         data_in = 0;
43
44         // Apply reset
45         #20 reset = 0;
46
47         // Test load operation
48         #10 data_in = 8'hAA; load = 1;
49         #10 load = 0;
50
51         // Test another load
52         #20 data_in = 8'h55; load = 1;
53         #10 load = 0;
54
55         // Test without load
56         #20 data_in = 8'hFF;
57
58         // End simulation
59         #50 $finish;
60     end
61
62     // Monitor changes
63     initial begin
64         $monitor("Time = %0t, clk = %b, reset = %b, load = %b, data_in = %h, data_out = %h",
65             $time, clk, reset, load, data_in, data_out);
66     end
67 endmodule

```

6.4.2 Serial In Serial Out

Module

keywordstyle

Testbench

keywordstyle

6.4.3 Parallel In Serial Out

Module

keywordstyle

Testbench

keywordstyle

6.4.4 Serial In Parallel Out

Module

keywordstyle

Testbench

keywordstyle

6.4.5 Universal Shift Register (Bi-Directional with Parallel Code)

Module

keywordstyle

Testbench

keywordstyle

Session 7

Finite State Machines

7.1 Introduction

A **Finite State Machine (FSM)** is a mathematical model of computation used to design sequential logic circuits and algorithms. It consists of:

- A finite set of **states**
- A set of **inputs**
- A set of **outputs**
- A **transition function**
- An **output function** (varies by type)

FSMs are widely used in digital systems, compilers, network protocols, and control systems.

7.2 Moore and Mealy Machines

7.2.1 Moore Machine

In a **Moore machine**, outputs depend only on the current state.

Example: A traffic light controller where:

- States: Red, Yellow, Green
- Outputs (lights) depend only on current state

```
keywordstyle
1 module traffic_light_controller
2 (
3     input clk,           // Clock signal
4     input reset,         // Active-high reset
5     output reg red,       // Red light output
6     output reg yellow,    // Yellow light output
7     output reg green      // Green light output
8 );
9
10 // Define states (Moore FSM)
11 typedef enum logic [1:0] {
12     RED_STATE,
13     YELLOW_STATE,
14     GREEN_STATE
15 } state_t;
16
17 reg [1:0] current_state, next_state;
18 reg [31:0] timer;        // Timer for state transitions
19
20 parameter RED_TIME = 10; // Time for RED light (in clock cycles)
21 parameter YELLOW_TIME = 2; // Time for YELLOW light
22 parameter GREEN_TIME = 10; // Time for GREEN light
23
24 // State transition logic
25 always @(posedge clk or posedge reset) begin
26     if (reset) begin
27         current_state <= RED_STATE;
28         timer <= 0;
29     end else begin
30         timer <= timer + 1;
31         case (current_state)
32             RED_STATE: begin
33                 red <= 1;
34                 yellow <= 0;
35                 green <= 0;
36                 if (timer >= RED_TIME) begin
37                     current_state <= GREEN_STATE;
38                     timer <= 0;
39                 end
40             end
41             YELLOW_STATE: begin
42                 red <= 0;
43                 yellow <= 1;
44                 green <= 0;
```

```

45         green <= 0;
46         if (timer >= YELLOW_TIME) begin
47             current_state <= RED_STATE;
48             timer <= 0;
49         end
50     end
51
52     GREEN_STATE: begin
53         red <= 0;
54         yellow <= 0;
55         green <= 1;
56         if (timer >= GREEN_TIME) begin
57             current_state <= YELLOW_STATE;
58             timer <= 0;
59         end
60     end
61
62     default: current_state <= RED_STATE;
63 endcase
64 end
65 end
66
67 endmodule

```

7.2.2 Mealy Machine

In a **Mealy machine**, outputs depend on both current state *and* inputs.

Example: Binary sequence detectors (e.g., 101 detector)

- Output (product delivery) depends on both current state (amount deposited) and input (product selection)

```

keywordstyle
1 module sequence_detector_101
2 (
3     input clk,           // Clock signal
4     input reset,         // Active-high reset
5     input data_in,       // Input bit stream
6     output reg detected  // Output (1 when sequence "101" is detected)
7 );
8
9 // Define states
10 typedef enum logic [1:0] {
11     S0, // Initial state (no match)
12     S1, // Detected "1"
13     S2  // Detected "10"
14 } state_t;
15
16 reg [1:0] current_state, next_state;
17
18 // State transition and output logic (Mealy Machine)
19 always @(posedge clk or posedge reset) begin
20     if (reset) begin
21         current_state <= S0;
22         detected <= 0;
23     end else begin
24         case (current_state)
25             S0: begin
26                 detected <= 0;
27                 if (data_in == 1)
28                     next_state <= S1; // Move to S1 if "1" is detected
29                 else
30                     next_state <= S0; // Stay in S0
31             end
32
33             S1: begin
34                 detected <= 0;
35                 if (data_in == 0)
36                     next_state <= S2; // Move to S2 if "0" follows "1"
37                 else
38                     next_state <= S1; // Stay in S1 (wait for "0")
39             end
40
41             S2: begin
42                 if (data_in == 1) begin
43                     detected <= 1; // Output 1 when "101" is complete
44                     next_state <= S1; // Overlap allowed: "1" of "101" can start next sequence
45                 end else begin
46                     detected <= 0;
47                     next_state <= S0; // Reset if input breaks sequence
48                 end
49             end
50
51             default: next_state <= S0;
52         endcase
53         current_state <= next_state;
54     end
55 end
56
57 endmodule

```

7.3 FSM Examples

7.3.1 Basic FSMs

1. Binary sequence detectors (e.g., 101 detector)
2. Traffic light controllers
3. Simple vending machines
4. Parity checkers
5. Edge detectors

7.3.2 Intermediate FSMs

1. UART transmitters/receivers
2. Cache coherence protocols
3. Elevator controllers
4. Digital combination locks
5. Packet header parsers

7.3.3 Advanced FSMs

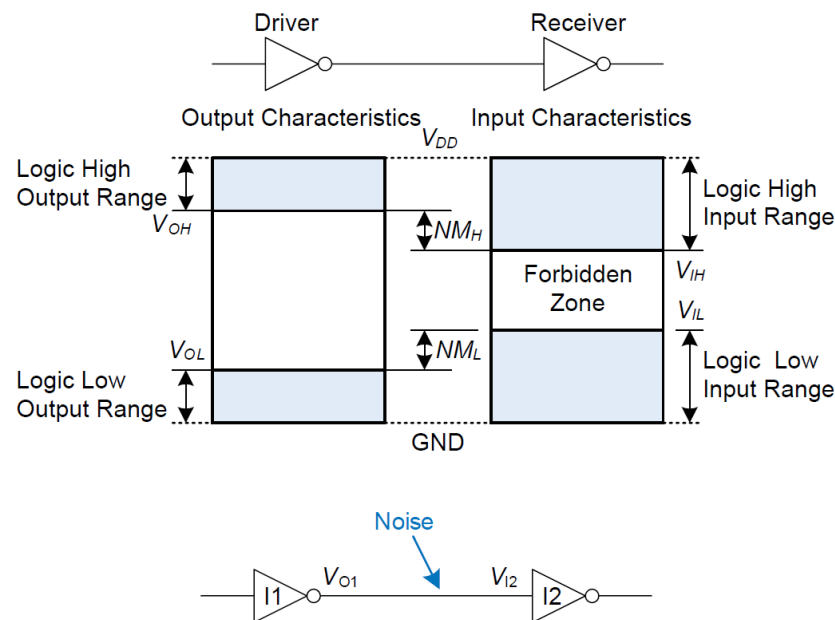
1. TCP state machines
2. Lexical analyzers (for compilers)
3. Protocol state machines (e.g., SPI, I2C)
4. Hardware security modules
5. AI behavior trees (hierarchical FSMs)

Session 8

Beneath the Digital Abstraction

8.1 Representation of Logic Levels in Digital Electronics

- $0V$ indicates logic level 0 (LOW) and $5V$ indicates logic level 1 (HIGH).
- Lowest voltage in the system is $0V$ (also called ground or GND).
- The highest voltage in the system comes from the power supply and is usually called V_{DD} . (typical values; $5V$, $3.3V$, $2.5V$, $1.8V$, $1.5V$, $1.2V$, or even lower)
- The mapping Voltage levels to logic levels is performed as shown in the figure below.



- The first gate is called the *driver* and the second gate is called the *receiver*.
- The output of the driver is connected to the input of the receiver. The driver produces a LOW (0) output in the range of 0 to V_{OL} or a HIGH (1) output in the range of V_{OH} to V_{DD} .
- If the receiver gets an input in the range of 0 to V_{IL} , it will consider the input to be LOW.
- If the receiver gets an input in the range of V_{IH} to V_{DD} , it will consider the input to be HIGH.
- If, for some reason such as noise or faulty components, the receiver's input falls in the forbidden zone between V_{IL} and V_{IH} , the behavior of the gate is unpredictable.
- V_{OH} , V_{OL} , V_{IH} , and V_{IL} are called the output and input high and low logic levels.

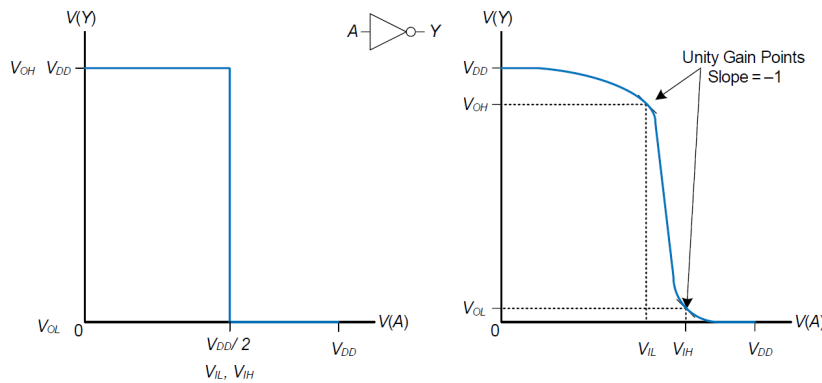
8.2 Noise Margins

- If the output of driver is to be correctly interpreted at the input of receiver, we must choose $V_{OL} < V_{IL}$ and $V_{OH} > V_{IH}$.

- Thus, even if the output of driver is contaminated by some noise, the input of receiver will still detect the correct logic level.
- The noise margin is the amount of noise that could be added to a worst-case output such that the signal can still be interpreted as a valid input.
- As can be seen in figure, the low and high noise margins are, respectively
 $N_{ML} = V_{IL} - V_{OL}$
 $N_{MH} = V_{OH} - V_{IH}$

8.3 DC Transfer Characteristics

- To understand the limits of the digital abstraction, we must delve into the analog behavior of a gate.
- The DC transfer characteristics of a gate describe the output voltage as a function of the input voltage when the input is changed slowly enough that the output can keep up.
- They are called transfer characteristics because they describe the relationship between input and output voltages.
- An *ideal* inverter would have an abrupt switching threshold at $V_{DD}/2$, as shown in figure (left).
 - For $V(A) < V_{DD}/2$, $V(Y) = V_{DD}$.
 - For $V(A) > V_{DD}/2$, $V(Y) = 0$.
 - In such a case, $V_{IH} = V_{IL} = V_{DD}/2$, $V_{OH} = V_{DD}$ and $V_{OL} = 0$.



- A *real* inverter changes more gradually between the extremes, as shown in figure (right).
 - When the input voltage $V(A)$ is 0 , the output voltage $V(Y) = V_{DD}$. When $V(A) = V_{DD}$, then $V(Y) = 0$.
 - However, the transition between these endpoints is smooth and may not be centered at exactly $V_{DD}/2$.
- This raises the question of how to define the logic levels.
 - A reasonable place to choose the logic levels is where the slope of the transfer characteristic $dV(Y)/dV(A)$ is -1 .
 - These two points are called the *unity gain points*.
 - Choosing logic levels at the unity gain points usually maximizes the noise margins.
 - If V_{IL} were reduced, V_{OH} would only increase by a small amount.
 - But if V_{IL} were increased, V_{OH} would drop precipitously.

8.4 Static Discipline

- To avoid inputs falling into the forbidden zone, digital logic gates are designed to conform to the *static discipline*, which requires that, given logically valid inputs, every circuit element will produce logically valid outputs.

- The choice of V_{DD} and logic levels is arbitrary, but all gates that communicate must have compatible logic levels.
- Therefore, gates are grouped into logic *families* such that all gates in a logic family obey the static discipline when used with other gates in the family.
- Four major logic families
 - Transistor-Transistor Logic (TTL)
 - Complementary Metal-Oxide-Semiconductor Logic (CMOS, pronounced sea-moss)
 - Low Voltage TTL Logic (LVTTTL)
 - Low Voltage CMOS Logic (LVCMOS).

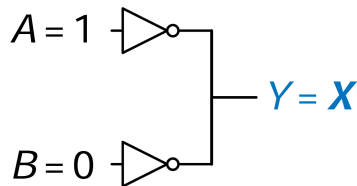
Logic Family	V_{DD}	V_{IL}	V_{IH}	V_{OL}	V_{OH}
TTL	5 (4.75-5.25)	0.8	2.0	0.4	2.4
CMOS	5 (4.5-6)	1.35	3.15	0.33	3.84
LVTTTL	3.3 (3-3.6)	0.8	2.0	0.4	2.4
LVCMOS	3.3 (3-3.6)	0.9	1.8	0.36	2.7

8.5 X and Z

Boolean algebra is limited to 0-s and 1-s. However, real circuits can also have illegal and floating values, represented symbolically by X and Z .

8.5.1 X the Illegal Value

- The symbol X indicates that the circuit node has an unknown or illegal value.
- This commonly happens if it is being driven to both 0 and 1 at the same time (in the figure, node Y is driven both HIGH and LOW).

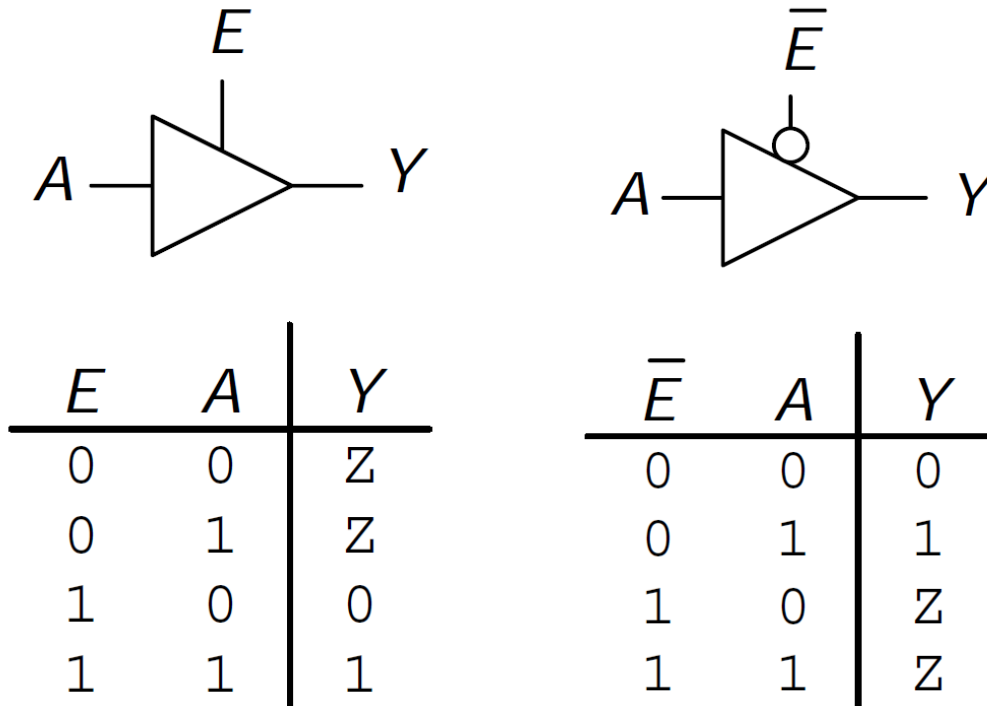


- This situation, called *contention*, is considered to be an error and must be avoided.
- The actual voltage on a node with contention may be somewhere between 0 and V_{DD} , depending on the relative strengths of the gates driving HIGH and LOW. It is often, but not always, in the forbidden zone.
- Contention also can cause large amounts of power to flow between the fighting gates, resulting in the circuit getting hot and possibly damaged.
- X values are also sometimes used by circuit simulators to indicate an uninitialized value. For example, if you forget to specify the value of an input, the simulator may assume it is an X to warn you of the problem.
- Digital designers also use the symbol X to indicate “don’t care” values in truth tables. Be sure not to mix up the two meanings.
- When X appears in a truth table, it indicates that the value of the variable in the truth table is unimportant.
- When X appears in a circuit, it means that the circuit node has an unknown or illegal value.

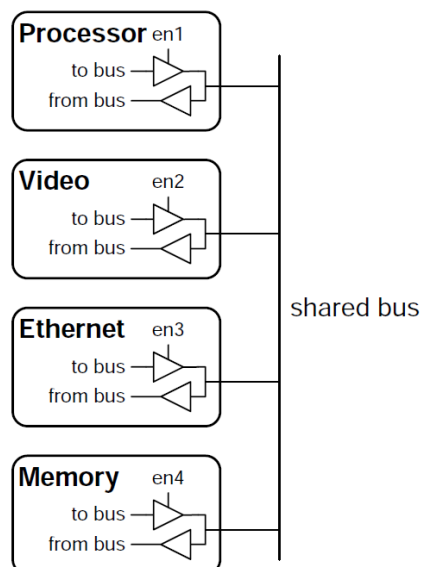
8.5.2 Z the Floating Value

- The symbol Z indicates that a node is being driven neither HIGH nor LOW. The node is said to be *floating*, *high impedance*, or *high Z*.
- A typical misconception is that a floating or undriven node is the same as a logic 0. In reality, a floating node might be 0, might be 1, or might be at some voltage in between, depending on the history of the system.

- A floating node does not always mean there is an error in the circuit, so long as some other circuit element does drive the node to a valid logic level when the value of the node is relevant to circuit operation.
- One common way to produce a floating node is to forget to connect a voltage to a circuit input, or to assume that an unconnected input is the same as an input with the value of 0. This mistake may cause the circuit to behave erratically as the floating input randomly changes from 0 to 1. Indeed, touching the circuit may be enough to trigger the change by means of static electricity from the body. We have seen circuits that operate correctly only as long as the student keeps a finger pressed on a chip.
- The *tristate* buffer, shown in figure (left), has three possible output states: HIGH (1), LOW (0), and floating (Z). The tristate buffer has an input, A , an output, Y , and an enable, E . When the enable is TRUE, the tristate buffer acts as a simple buffer, transferring the input value to the output. When the enable is FALSE, the output is allowed to float (Z).



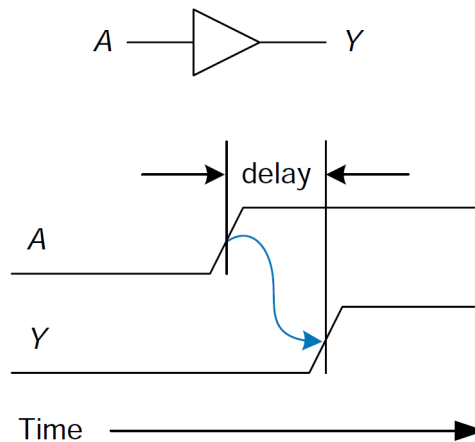
- The tristate buffer in figure (left) has an active high enable. That is, when the enable is HIGH (1), the buffer is enabled. Figure (right) shows a tristate buffer with an active low enable. When the enable is LOW (0), the buffer is enabled.
- Tristate buffers are commonly used on *busses* that connect multiple chips. For example, a microprocessor, a video controller, and an Ethernet controller might all need to communicate with the memory system in a personal computer. Each chip can connect to a shared memory bus using tristate buffers, as shown in figure.



- Only one chip at a time is allowed to assert its enable signal to drive a value onto the bus. The other chips must produce floating outputs so that they do not cause contention with the chip talking to the memory. Any chip can read the information from the shared bus at any time.

8.6 Timing

One of the most challenging issues in circuit design is timing: making a circuit run fast. An output takes time to change in response to an input change. Figure below shows the delay between an input change and the subsequent output change for a buffer.



The figure is called a timing diagram; it portrays the transient response of the buffer circuit when an input changes. The transition from LOW to HIGH is called the rising edge. Similarly, the transition from HIGH to LOW (not shown in the figure) is called the falling edge. The blue arrow indicates that the rising edge of Y is caused by the rising edge of A . We measure delay from the 50% point of the input signal, A , to the 50% point of the output signal, Y . The 50% point is the point at which the signal is half-way (50%) between its LOW and HIGH values as it transitions.

8.6.1 Propagation and Contamination Delay

Combinational logic is characterized by its propagation delay and contamination delay. The propagation delay, t_{pd} , is the maximum time from when an input changes until the output or outputs reach their final value. The contamination delay, t_{cd} , is the minimum time from when an input changes until any output starts to change its value.

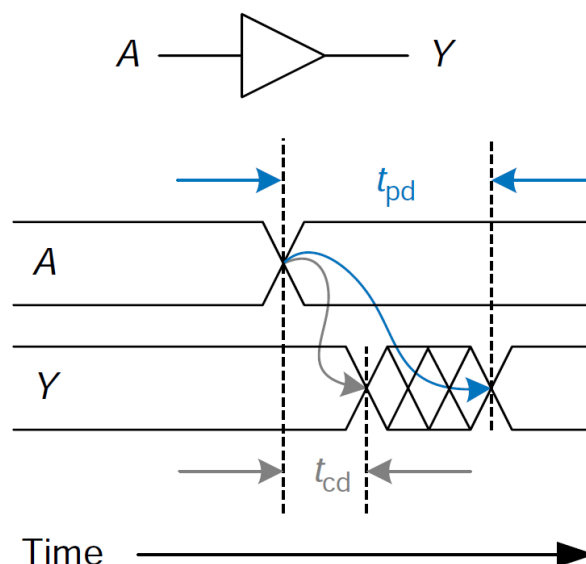


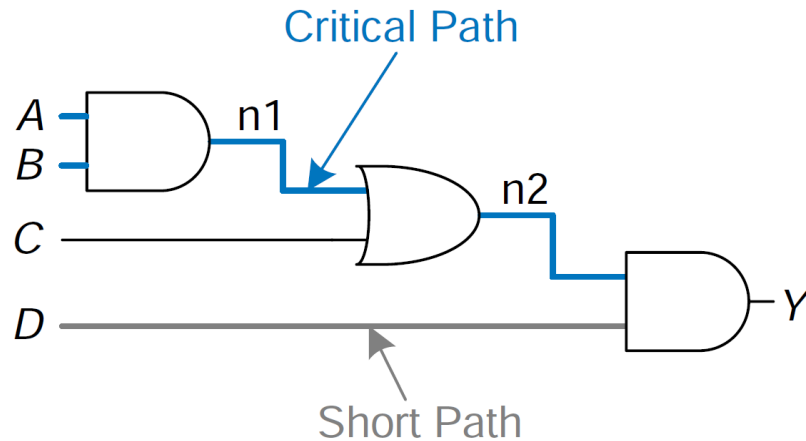
Figure above illustrates a buffer's propagation delay and contamination delay in blue and gray, respectively. The figure shows that A is initially either HIGH or LOW and changes to the other state at a particular time; we are interested only in the fact that it changes, not what value it has. In response, Y changes some time later. The arcs indicate that Y may start to change t_{cd} after A transitions and that Y definitely settles to

its new value within t_{pd} . The underlying causes of delay in circuits include the time required to charge the capacitance in a circuit and the speed of light. t_{pd} and t_{cd} may be different for many reasons, including

- different rising and falling delays
- multiple inputs and outputs, some of which are faster than others
- circuits slowing down when hot and speeding up when cold

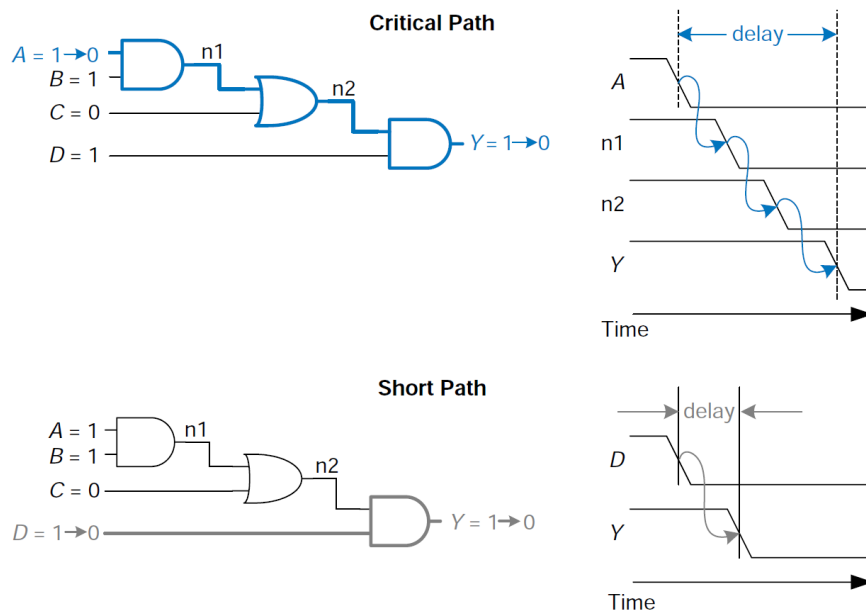
Manufacturers normally supply data sheets specifying these delays for each gate.

Along with the factors already listed, propagation and contamination delays are also determined by the path a signal takes from input to output. Figure below shows a four-input logic circuit.



The critical path, shown in blue, is the path from input A or B to output Y. It is the longest, and therefore the slowest, path, because the input travels through three gates to the output. This path is critical because it limits the speed at which the circuit operates. The short path through the circuit, shown in gray, is from input D to output Y. This is the shortest, and therefore the fastest, path through the circuit, because the input travels through only a single gate to the output.

The propagation delay of a combinational circuit is the sum of the propagation delays through each element on the critical path. The contamination delay is the sum of the contamination delays through each element on the short path. These delays are illustrated in Figure below and are described by the following equations:

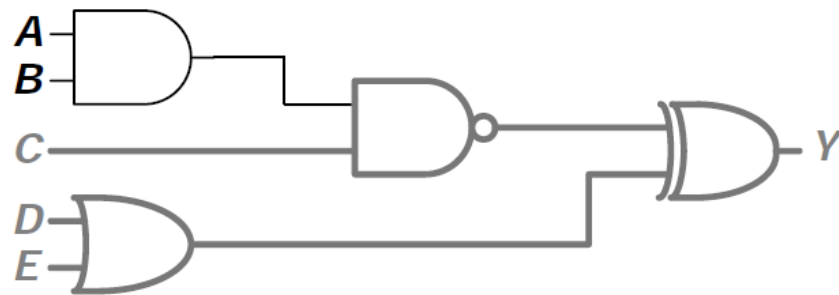
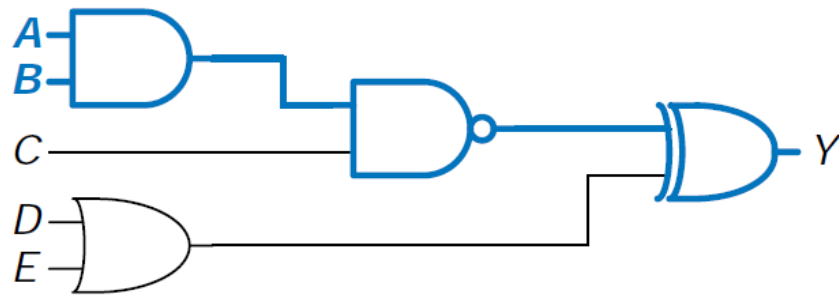
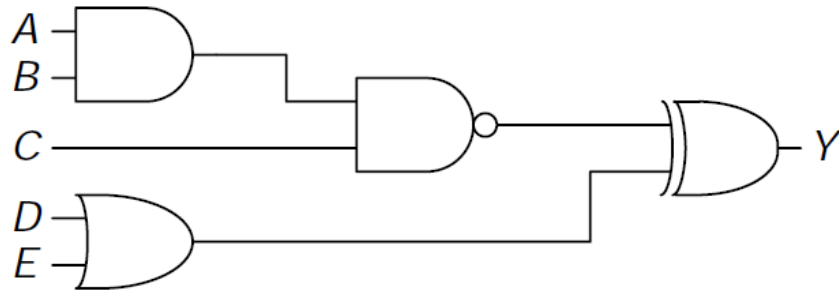


$$t_{pd} = 2t_{pd_{AND}} + t_{pd_{OR}}$$

$$t_{cd} = t_{cd_{AND}}$$

Example

Find the propagation delay and contamination delay of the circuit shown in Figure if each gate has a propagation delay of 100 ps and a contamination delay of 60 ps.



Ben begins by finding the critical path and the shortest path through the circuit. The critical path, highlighted in blue, is from input A or B through three gates to the output, Y. Hence, t_{pd} is three times the propagation delay of a single gate, or 300 ps.

The shortest path, shown in gray is from input C, D, or E through two gates to the output, Y. There are only two gates in the shortest path, so t_{cd} is 120 ps.

Session 9

Project Titles

1. Four way traffic light system with pedestrian crossings
2. Fully automatic washing machine controller
3. Elevator controller for 4 floors
4. Vending machine controller
5. DDR3/DDR4 memory controller
6. Cache memory simulator
7. UART (Universal Asynchronous Receiver-Transmitter)
8. 8-bit RISC processor (MIPS-like) that supports ADD, SUB, LOAD, STORE, and JUMP instructions.
9. Fully automated car parking system with slot counter

Project Requirements

- Group project (max 3 persons for a group)
 - Block-level design (Modules, IO selection and overall functionality)
 - Verilog implementation + simulation (testbenches)
 - Demo + Viva
-