EECS 270 W25
Discussion 2

January 16th, 2025
By: Mick Gordinier

1. Complete the **NEW** Getting Started section to Follow Along (Optional)

2. Please fill out the Google form! —> (Link Here As Well)

Getting Started (Optional)

CAEN VNC - Connecting Virtually to UofM tools!

Link to VMware Horizon Web Login

Sometimes doesn't allow you to login. I prefer to use the application below

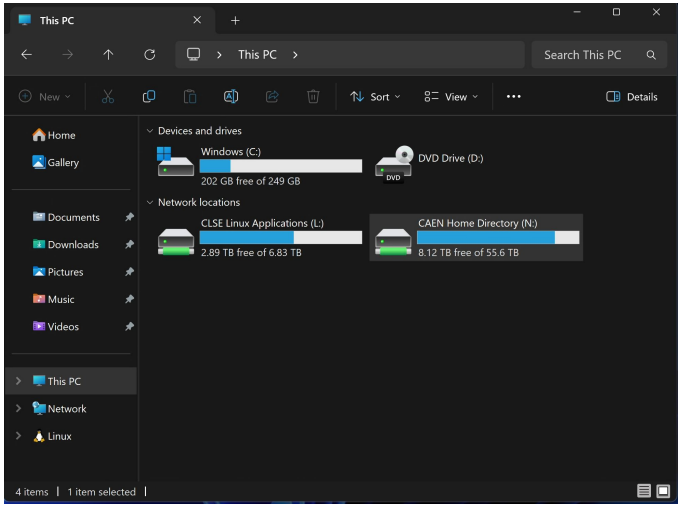
Dropbox Link to Download VMware Horizons (Mac)
Dropbox Link to Download VMware Horizons (Linux x64)
Dropbox Link to Download VMware Horizons (Windows x64)

- Can connect to CAEN Software from your local computer
- Can now access UofM tools anywhere at any time!!

Advice For Working with UofM CAEN Computers

- All work should be done in your CAEN Home Directory **(N:\)**
  - Rare occurrences of certain C:\ drive folders being wiped for no reason
  - **Additionally, all work done on N:\ drive will be saved on the cloud for ALL CAEN computers!**

- Finding the N:\ Drive
  1. Go to File Explorer
  2. Click on ‘This PC’ on the bottom left
  3. Click on ‘CAEN Home Directory (N:)’



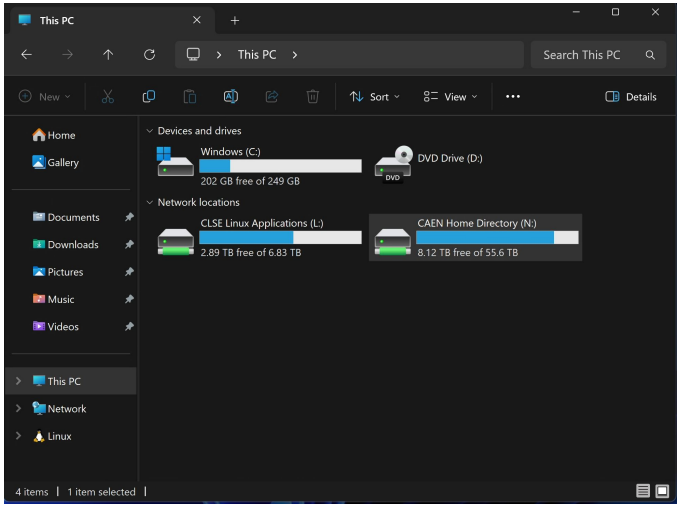
Folder Structure to Follow for ModelSim Demos

N:/

→ 270_Projects/

→Proejct05/

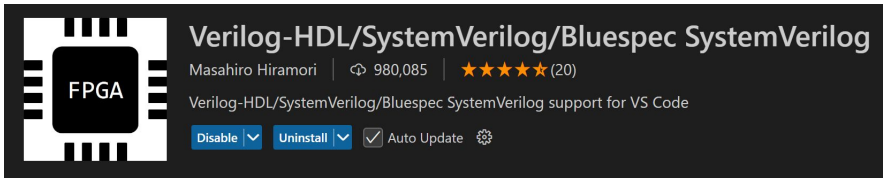
→ All Verilog files for Project05



Text Editor Recommendation for Verilog Coding

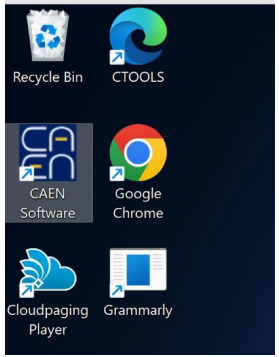
- Can use whatever you want to write Verilog
  - I've seen notepad, notepad++, Quartus Prime, … used as text editors

- My Recommendation - Use **VS Code**
  - Use Intellisense Extension below

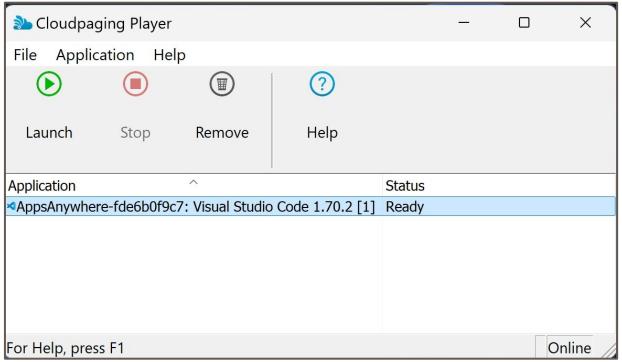


- Please please please don't use ModelSim as a text editor
  - It has led to many issues

Accessing VS Code From CAEN



1. Click on ‘CAEN Software’ desktop icon
   a. It takes you to https://appsanywhere.engin.umich.edu/
2. Search for application ‘Visual Studio Code’
   a. And launch VS Code
3. Application ‘Cloudpaging Player’ Should Have Opened
   a. If not, click on the desktop icon to open
4. Launch the VS Code Application
   a. Either through choosing VS Code and then ‘Launch’
   b. Or double clicking the VS Code application in Couldpaging



Adding Verilog Files to Our Project

1. Have VS Code Open on CAEN
2. Click on Files → Open Folder
3. Open the N:/270_Projects/Project05/ folder

4. Click on the new file icon in VS Code ‘Explorer’ tab
5. Add these specific files to the folder
    a. Checker.v
    b. CHECK1.v
    c. Project05.v
    d. CheckerTestbench.v
    e. Check1Testbench.v

Textual Solution to Project 0.5 (For Copy-Paste)

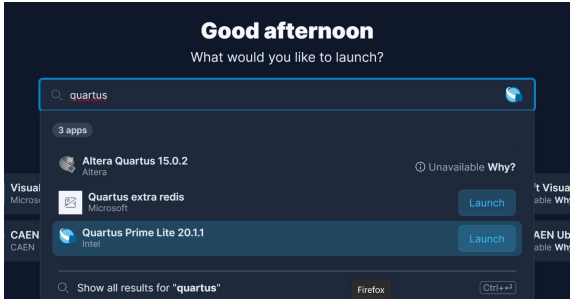
```
module Project05(
  input [16:0] SW,    // Using {SW[16], ..., SW[1], SW[0]} as inputs
  output [6:0] LEDR,  // Using {LEDR[6], ..., LEDR[0]} as outputs
  output [6:0] HEX0   // Using {HEX0[6], ..., HEX[0]} as outputs
);

  // Step 1. Declare your internal wires
  wire [6:0] A, B, F;

  // Step 2. Assign internal wires A and B to the inputs
  // Instantiating 2 buffer arrays, each with 7 buffers
  // Each array will give the value of Switches directly to A and B
  buf b0[6:0](A, SW[16:10]);
  buf b1[6:0](B, SW[6:0]);

  // Step 3. Instantiating Checker module
  // Passing A, B as inputs and F as output
  Checker c(A, B, F);

  // Step 4. Pass F to Outputs
  buf b2[6:0](LEDR[6:0], F);
  not n1[6:0](HEX0[6:0], F);

endmodule // Project 0.5
```

```
module Checker(
  input [6:0] A, B,
  output [6:0] F
);

  // Instantiate CHECK1 module for each bit pair
  CHECK1 C[6:0](A, B, F);

endmodule // Checker
```

```
module CHECK1(
  input a, b,
  output f
);

  // Step 1. Declare Your Wires
  // Declaring 1-bit wires w, x, y, z
  wire w, x, y, z;

  // Step 2. Instantiate the gates
  and a1(w, a, b);
  not n1(x, a);
  not n2(y, b);
  and a2(z, x, y);
  or o1(f, w, z);

endmodule // CHECK1
```

Getting ModelSim From CAEN Tools

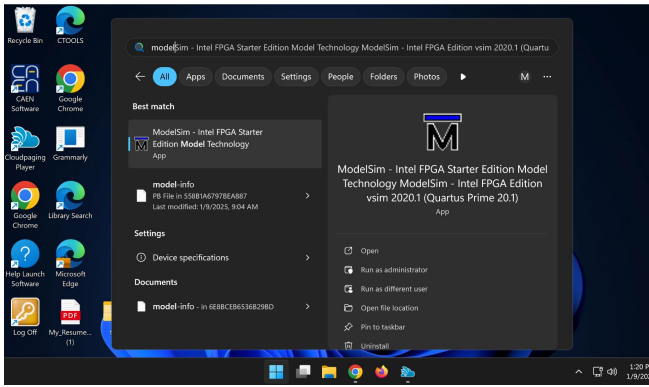
- Very strange (Not sure why it's like this)



1. Click on 'CAEN Software' desktop icon
   a. It takes you to https://appsanywhere.engin.umich.edu/

2. Search for application **'Quartus Prime Lite' (Not ModelSim)**
   a. Press launch

3. Once Quartus Installed, ModelSim is also installed
   a. Can be found through the regular application search



Textual Testbenches for Project 0.5

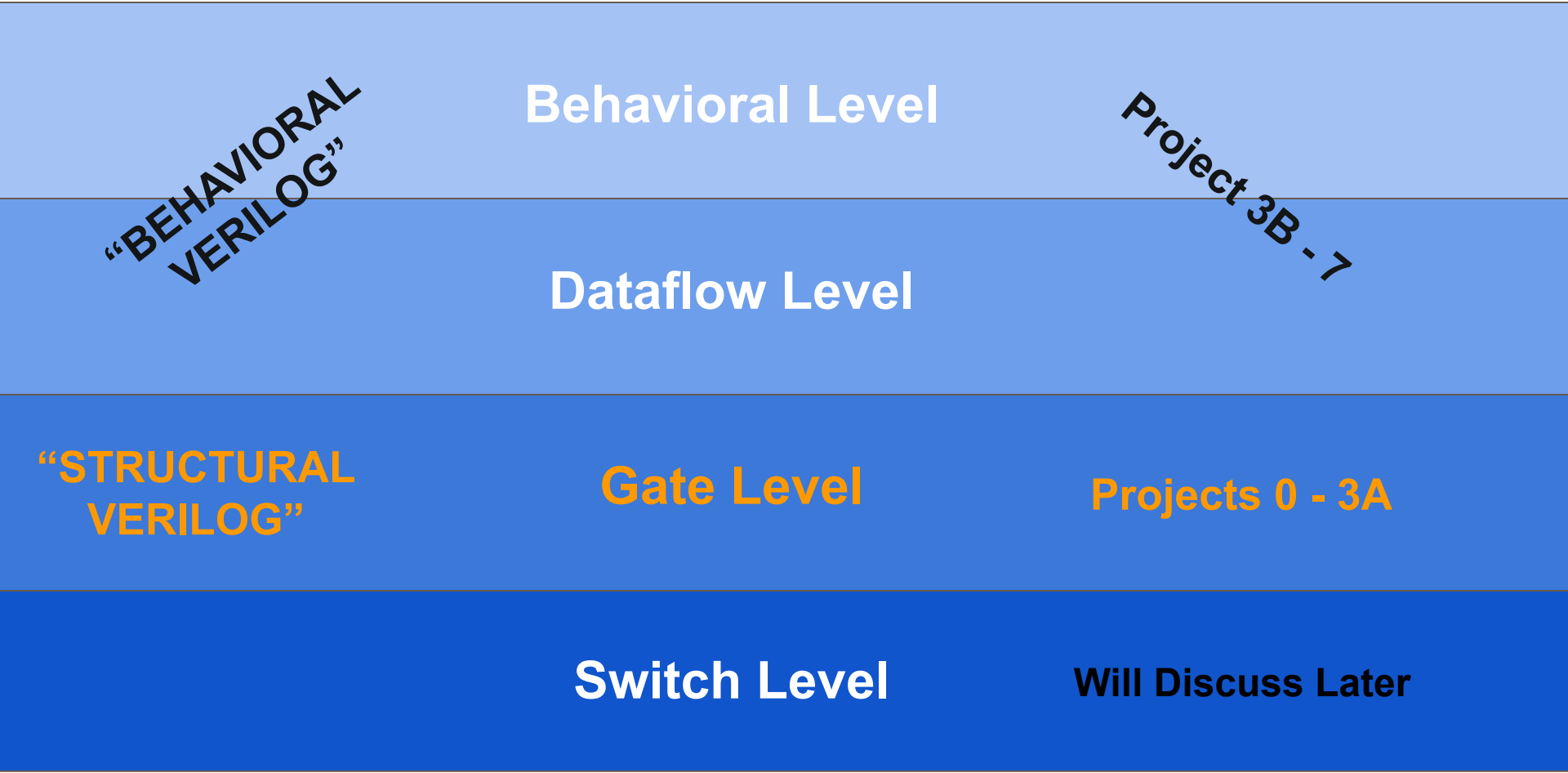
```
// Timescale on the top
// `timescale <time_unit>/<time_precision>
// My preference, use 1ns/1ns
`timescale 1ns/1ns

// Testbench module has no inputs/outputs
module CheckerTestbench;

    // All controlable inputs to DUT declared as reg
    reg [6:0] A, B;

    // All outputs of DUT declared as wire
    wire [6:0] F;

    // Instantiate module we want to test
    // DUT - Device Under Testing
    Checker dut(A, B, F);

    // Initial statement use to contorl inputs
    // Initial means right at t = 0
    initial begin

        // assign A and B to some value at t=0
        A = 7'b1111111;
        B = 7'b1111111;

        // Wait statement
        // Waiting 5 time units --> 5 ns
        #5;

        A[1] = 1'b0;

        #5;

        B[3] = 1'b0;

        #5;

        A[1] = 1'b1;
        A[3] = 1'b0;

    end  // initial

endmodule  // Testbench
```

```
// Timescale on the top
// `timescale <time_unit>/<time_precision>
// My preference, use 1ns/1ns
`timescale 1ns/1ns

// Testbench module has no inputs/outputs
module Check1Testbench;

    // All controlable inputs to DUT declared as reg
    reg a, b;

    // All outputs of DUT declared as wire
    wire f;

    // Instantiate module we want to test
    // DUT - Device Under Testing
    CHECK1 dut(a, b, f);

    // Initial statement use to contorl inputs
    // Initial means right at t = 0
    initial begin

        // assign a and b to logical low at t=0
        a = 1'b0;
        b = 1'b0;

        // Wait statement TO LET SIGNAL PASS THROUGH
        // Waiting 5 time units --> 5 ns
        #5;

        // updating b to logical high
        // We should see that eventually f becomes logical LOW
        b = 1'b1;

        // Wait statement TO LET SIGNAL PASS THROUGH
        #5;

        // updating b to logical high
        // We should see that eventually f becomes logical HIGH
        a = 1'b1;

        #5;

        // updating b back to logical low
        b = 1'b0;

    end  // initial

endmodule  // Testbench
```

1. Discussion 1 Recap
2. Deep Dive into ModelSim Simulation / Testbenches
3. Propagation Delay
4. Project 2 “Timing” Discussion
5. (Bonus) Structural Verilog Practice Problems
6. (Bonus) More Capture, Create, Implement Practice
7. (Bonus) Specified Delay Calculation Slides

Discussion 1 Recap

Verilog's Many Levels of Abstraction



USE STRUCTURAL VERILOG FOR Projects 0-3A!!!

- **You SHOULD NOT be using**
  - **Continuous ‘assign’ statements**
  - **Procedural statements ‘always’ or ‘initial’**

- These abstractural statements are used in “Behavioral” Verilog
  - Will be using in Projects 3B - 7

- **You SHOULD ONLY be using**
  - **Primitive gate instantiations as described in previous discussion**
  - **Module instantiations as described in previous discussion**

Capture, Create, Implement Process

1. **Capture** the behavior of the function
   a. Either through a truth table or equation (Whichever is easier)

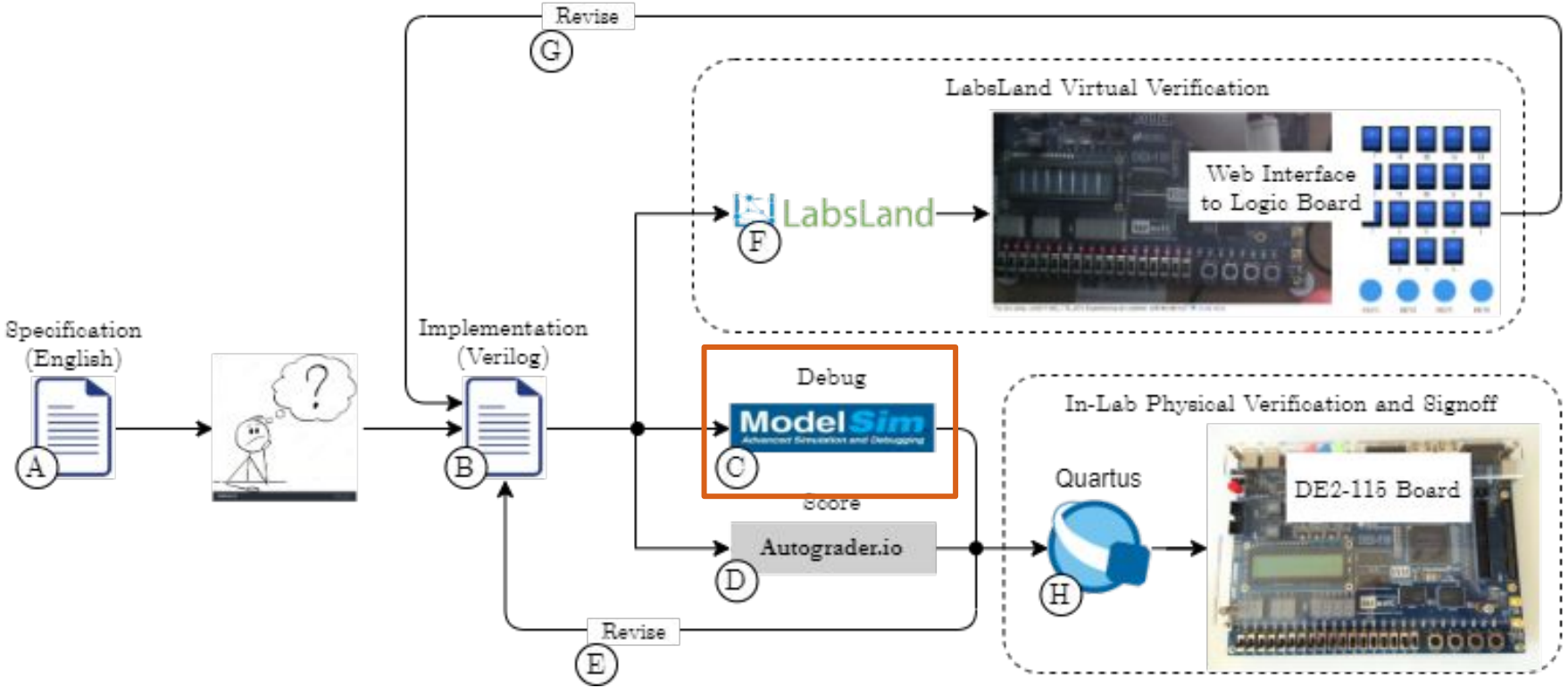
2. **Create** equations (If you haven't already)
   a. OR all of the minterms of the function
   (We will cover minterms more thoroughly in a future lecture)

3. **Implement** a gate-based circuit for <u>each</u> output
   a. Can then translate that gate directly into Structural Verilog

- **Another FULL project outline at the end of this slide deck!!**
  - **Added additional helpful structural verilog tips**

Deep Dive into ModelSim Simulation / Testbenches

Overview of the Tools



Why Do We Need to Simulate?

- Main Goal: Need to ensure our code behaves the correct way
  - Verifying the robustness of our design
  - **BEFORE FABRICATING THE PHYSICAL HARDWARE**

- If we don’t simulate before testing on the board
  - The possible issues could be endless

General Structure of a Testbench (Project 0.5)

```
// Timescale on the top
// `timescale <time_unit>/<time_precision>
// My preference, use 1ns/1ns
`timescale 1ns/1ns

// Testbench module has no inputs/outputs
module Testbench;

    // All controlable inputs to DUT declared as reg
    reg [6:0] A, B;

    // All outputs of DUT declared as wire
    wire [6:0] F;

    // Instantiate module we want to test
    // DUT - Device Under Testing
    Checker dut(A, B, F);

    // Initial statement use to contorl inputs
    // Initial means right at t = 0
    initial begin

        // Controling input here

    end  // initial

endmodule  // Testbench
```

```
// Initial statement use to contorl inputs
// Initial means right at t = 0
initial begin

    // assign A and B to some value at t=0
    A = 7'b1111111;
    B = 7'b1111111;

    // Wait statement
    // Waiting 5 time units --> 5 ns
    #5;

    A[1] = 1'b0;

    #5;

    B[3] = 1'b0;

    #5;

    A[1] = 1'b1;
    A[3] = 1'b0;

end  // initial
```

Timescale on a Testbench

```
`timescale <time_unit>/<time_precision>
```

- **FOR SIMULATION PURPOSES ONLY (NOT SYNTHESIZABLE)**

- Time Unit - Amount of delay/simulation time in a unit of time
- Time Precision - Rounding of values and increments on simulation software

- **NOTE: ` IS NOT AN APOSTROPHE, IT IS A BACK QUOTE!**

- **My Advice - Use time_unit and time_precision be ‘1ns’**

Verilog 2nd Main Data Type - Registers

- **Registers should NOT be used in your non-simulation code for projects 0-3A**
- **TLDR don’t worry about what registers are right now**

```
// Testbench module has no inputs/outputs
module Testbench;

    // All controlable inputs to DUT declared as reg
    reg [6:0] A, B;

    // All outputs of DUT declared as wire
    wire [6:0] F;

    // Instantiate module we want to test
    // DUT - Device Under Testing
    Checker dut(A, B, F);
```

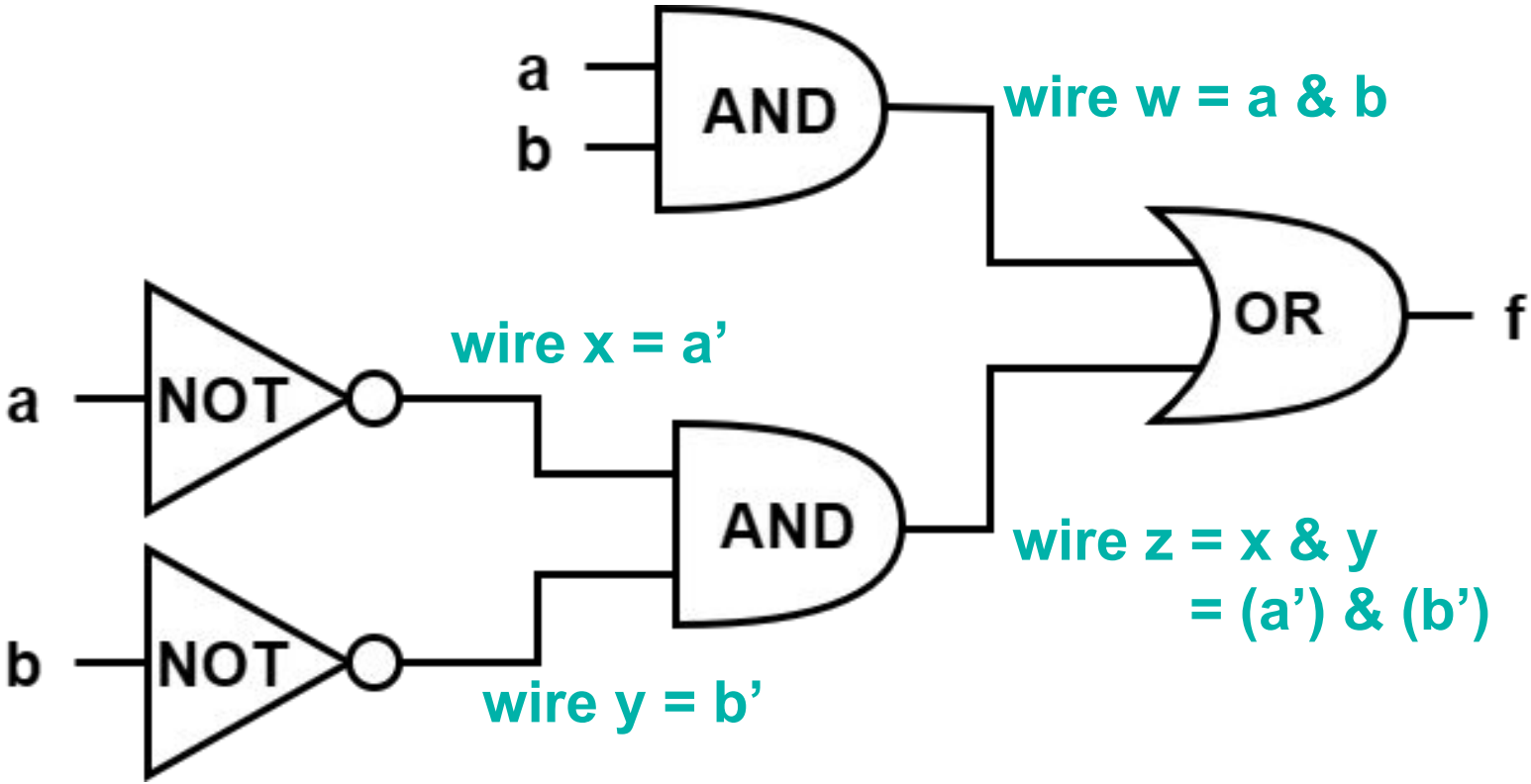
Number Specification in Verilog

- Specifying sized numbers: **{size}'{base format}{number}**
  - Size - Number of bits associated with the number
  - Base Format - Can be decimal (D or d), Binary (B or b), Hex (H or h), Octal (O or o)
  - Number - Number specified in the provided base format

Ex. 4'b1001 → **4-bit** number with **binary** value **1001** (9 in decimal)
Ex. 5'd13 → **5-bit** number with **decimal** value **13** (**01101** in binary)
Ex. 8'hAA → **8-bit** number with **hex** value **AA** (**1010_1010** in binary)

- Valid to use underscores anywhere for readability → 12'b0110_1001_0011

Demo of ModelSim
(Complete the Getting Started Section)


a
b
AND
wire w = a & b
OR
f
a
NOT
wire x = a'
AND
wire z = x & y
= (a') & (b')
b
NOT
wire y = b'

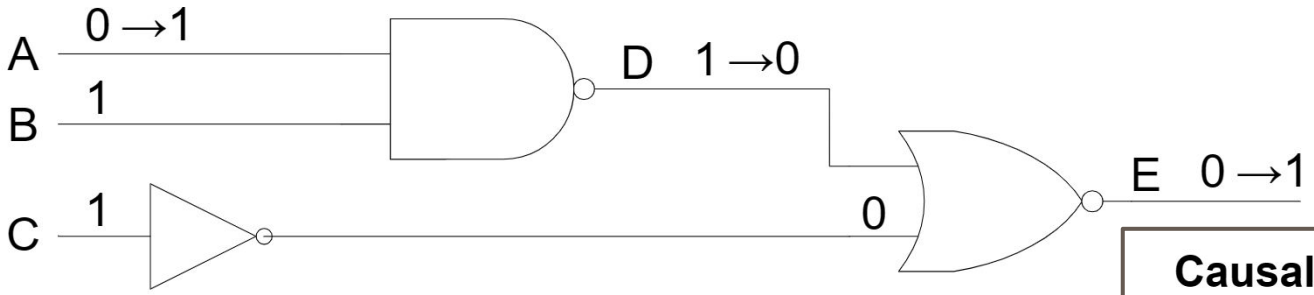
Propagation Delay

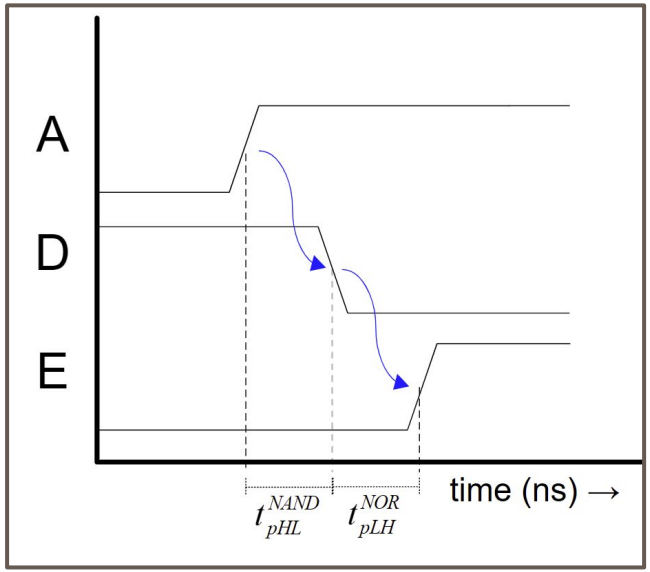
Output Changes are NOT Immediate

- Logical circuitry is built using physical wires and transistors
  - Each have some internal delay
  - In real implementations, need to account for delay

- **Propagation Delay** - Time it takes for the input change to be reflected onto the output

Causality Graphs and Terminology





Causality Arrow

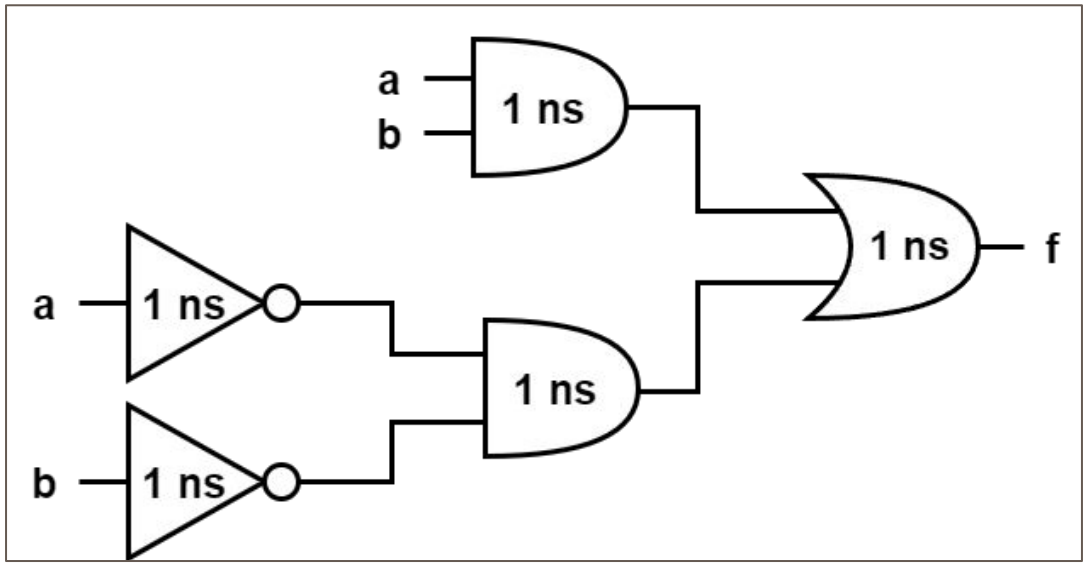
Gate Propagation Delay

t_{pHL}^{NAND}

gives NAND gate propagation delay from input to output when output is changing from H to L

Calculating Propagation Delay - Unit Delay

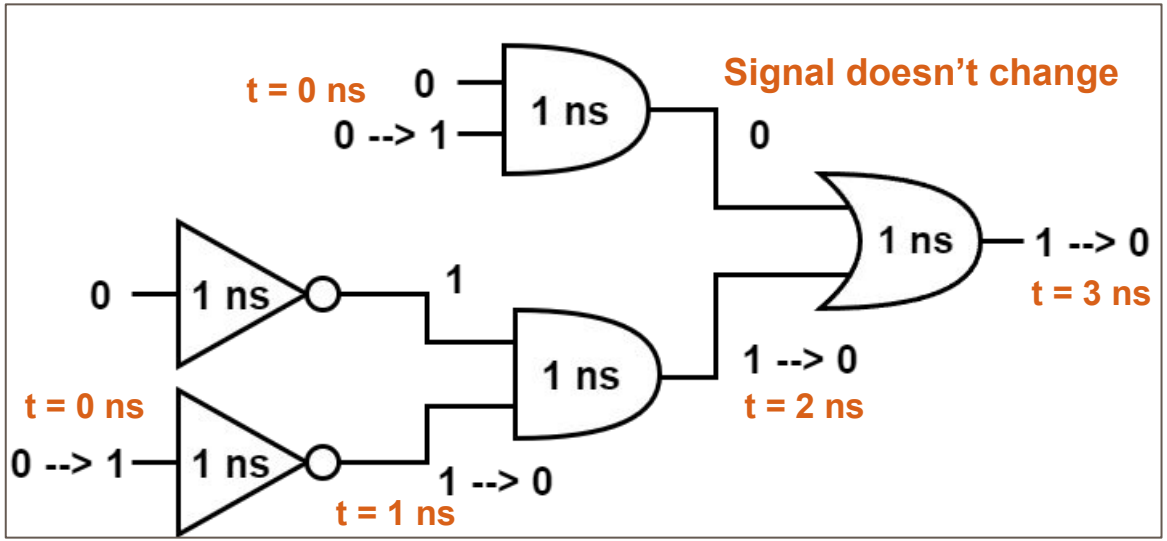
- **Unit Delay** - Takes **1 unit of delay** to propagate through the gate
  - We will be assuming 1 time unit = 1ns for these example problems



CHECK1 Circuit with Unit Delay Gates

Calculating Propagation Delay - Unit Delay

1. Assume gates are **SETTLED** at some initial combination of values
2. Update **one** of the controllable inputs (Ex. Updating 'b' from 0 → 1)
3. Let the signal propagate through the circuit until the output has **SETTLED**



CHECK1 Circuit with Unit Delay Gates

Starting with combination (a, b) = (0, 0)

→ $t^{b\rightarrow f}_{pHL}$ = 3 ns

Remember: pHL is the OUTPUT going from H→L

This is not true for all initial combinations of (a, b)

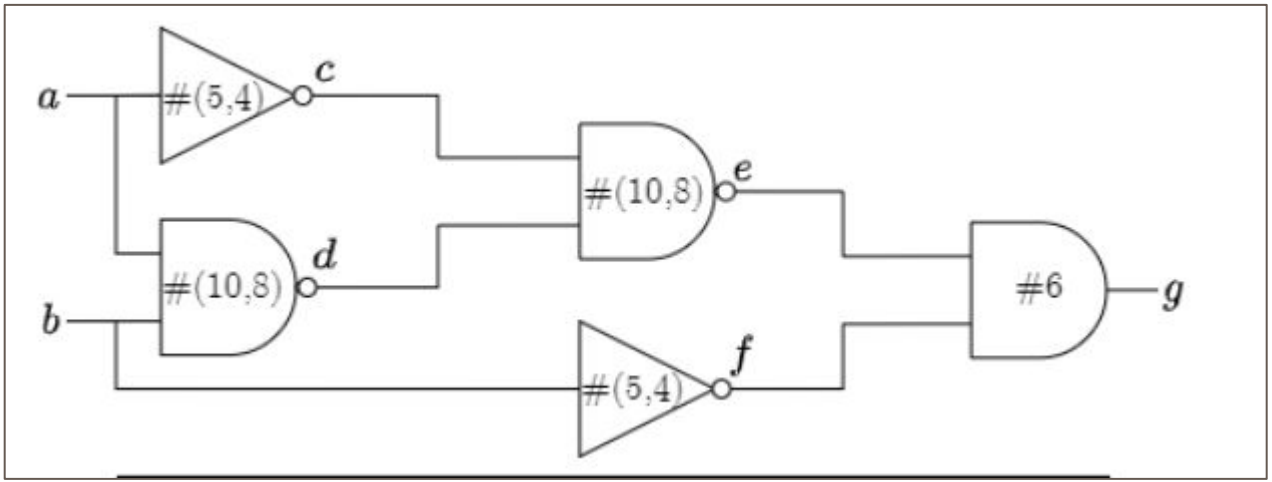
ModelSim Demo 2
(Viewing CHECK1 Unit-Delay Propagation and Cursors)

Updating CHECK1 for Unit Delay

```
// NEW: timescale is now needed to indicate time unit to simulator
`timescale 1ns/1ns

module CHECK1(
  input a, b,
  output f
);

  // Step 1. Declare Your Wires
  // Declaring 1-bit wires w, x, y, z
  wire w, x, y, z;

  // Step 2. Instantiate the gates
  // NEW: ADDING SIMULATED DELAYS TO OUR GATES
  // NEW: Specifying Unit Delays (From both L→H and H→L)
  and #1 a1(w, a, b);
  not #1 n1(x, a);
  not #1 n2(y, b);
  and #1 a2(z, x, y);
  or  #1 o1(f, w, z);

endmodule // CHECK1
```

Calculating Propagation Delay - Specified Delay

- **Specified Delay** - Rise and Fall delays can be different for each gate
  - Given in #(Rise Delay Time Units, Fall Delay Time Units)



Lecture Circuit with Specified Delay Gates

Project 2 “Timing” Discussion

Ring Oscillator

- Odd number of inverting gates
- Will oscillate infinitely (Not Stable)
- Generate periodic waveforms

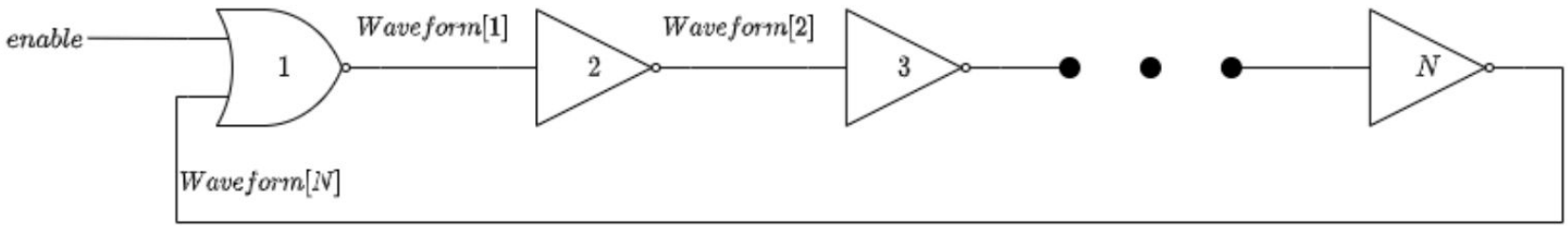


Figure 2: *N*-stage ring oscillator circuit

NEW GATE: NOR (Not OR) Logical Gate

NOR

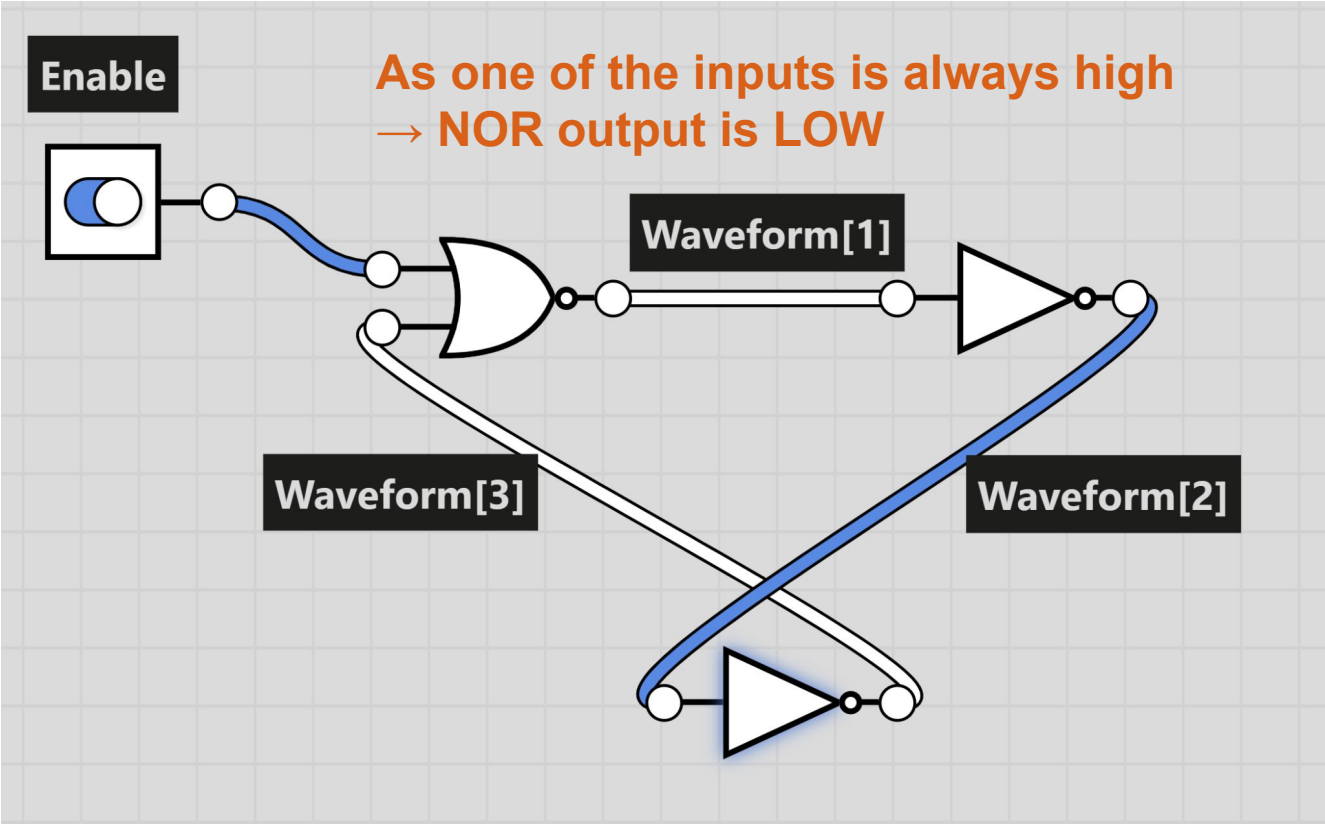
A	B	OUT
0	0	1
0	1	0
1	0	0
1	1	0

Inverted OR

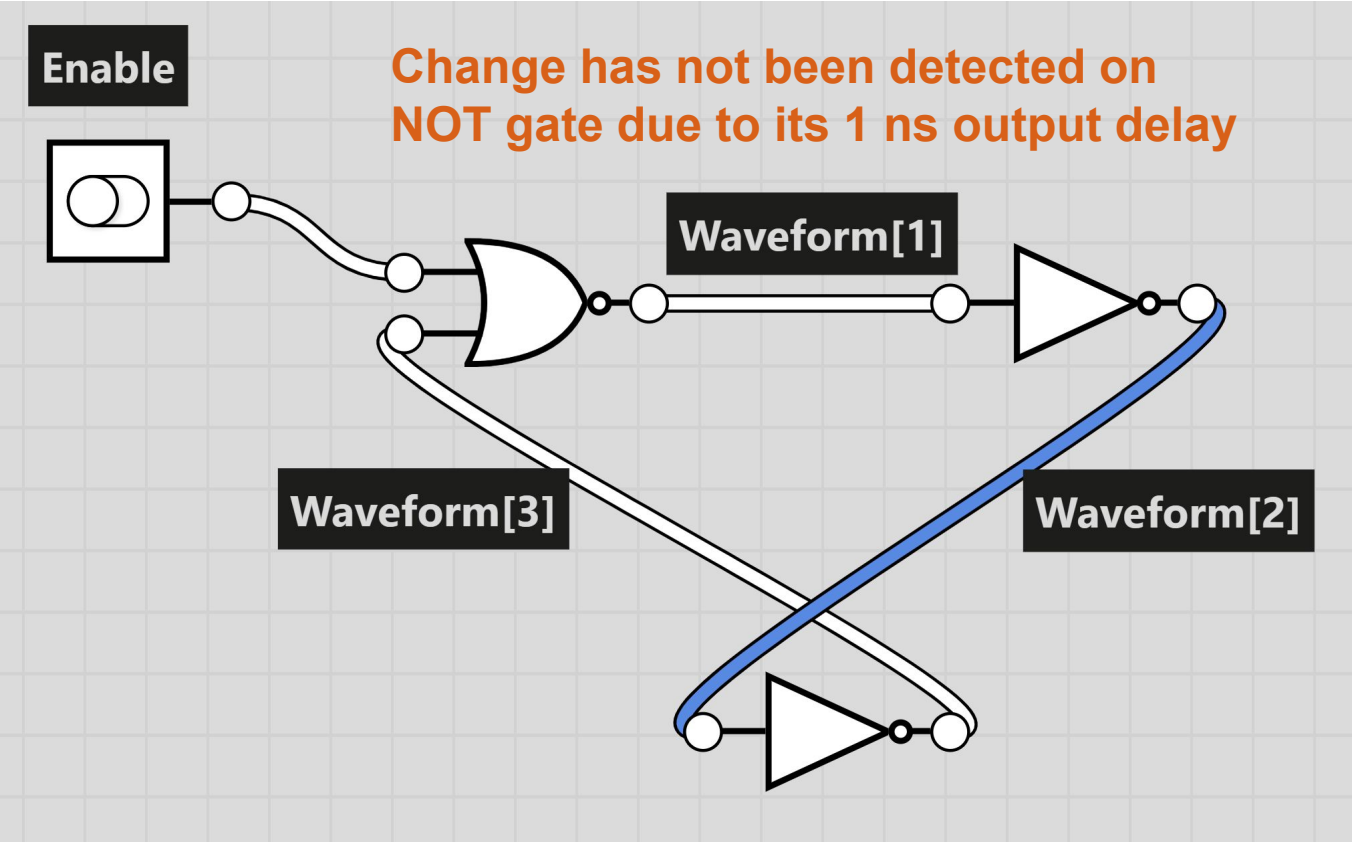
Walkthrough of the 3-Stage Ring Oscillator Code

```
`timescale 1ns/1ns
module RingOsc3(enable);
        input enable;   // When 1, stop; when 0, start
        wire [3:1] Waveform;

        nor #1 norGate(Waveform[1], Waveform[3], enable);
        not #1 notGate1(Waveform[2], Waveform[1]);
        not #1 notGate2(Waveform[3], Waveform[2]);
endmodule
```

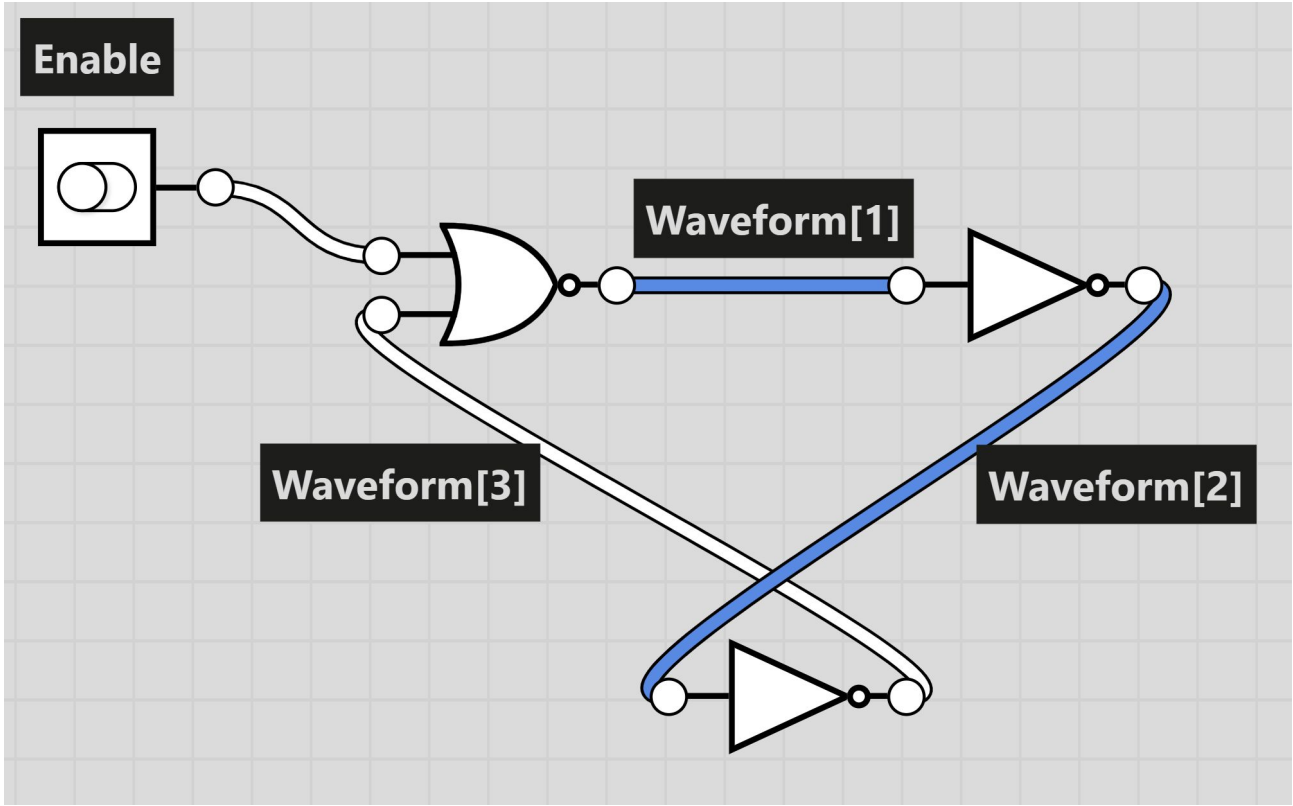
Enable HIGH → Stable Circuit



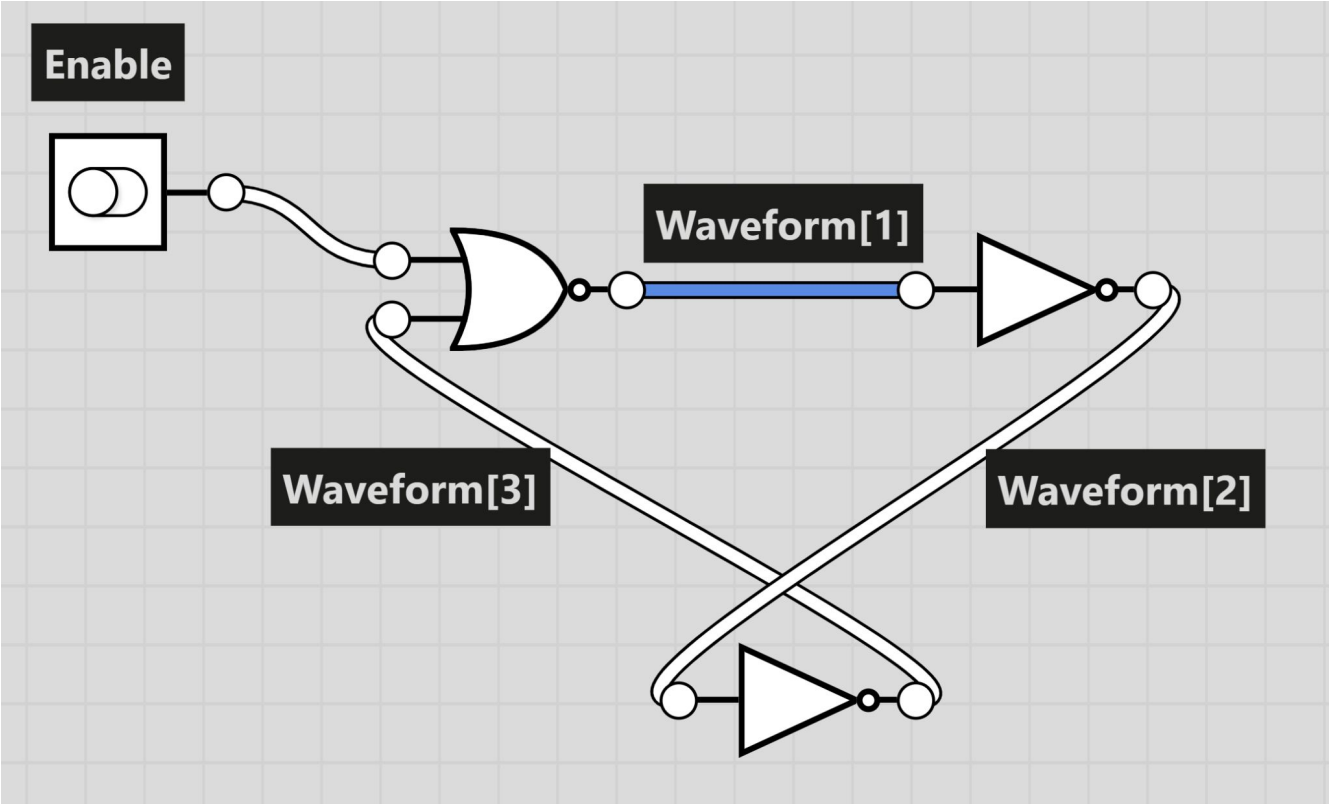
Enable goes LOW at **t = 0 ns**



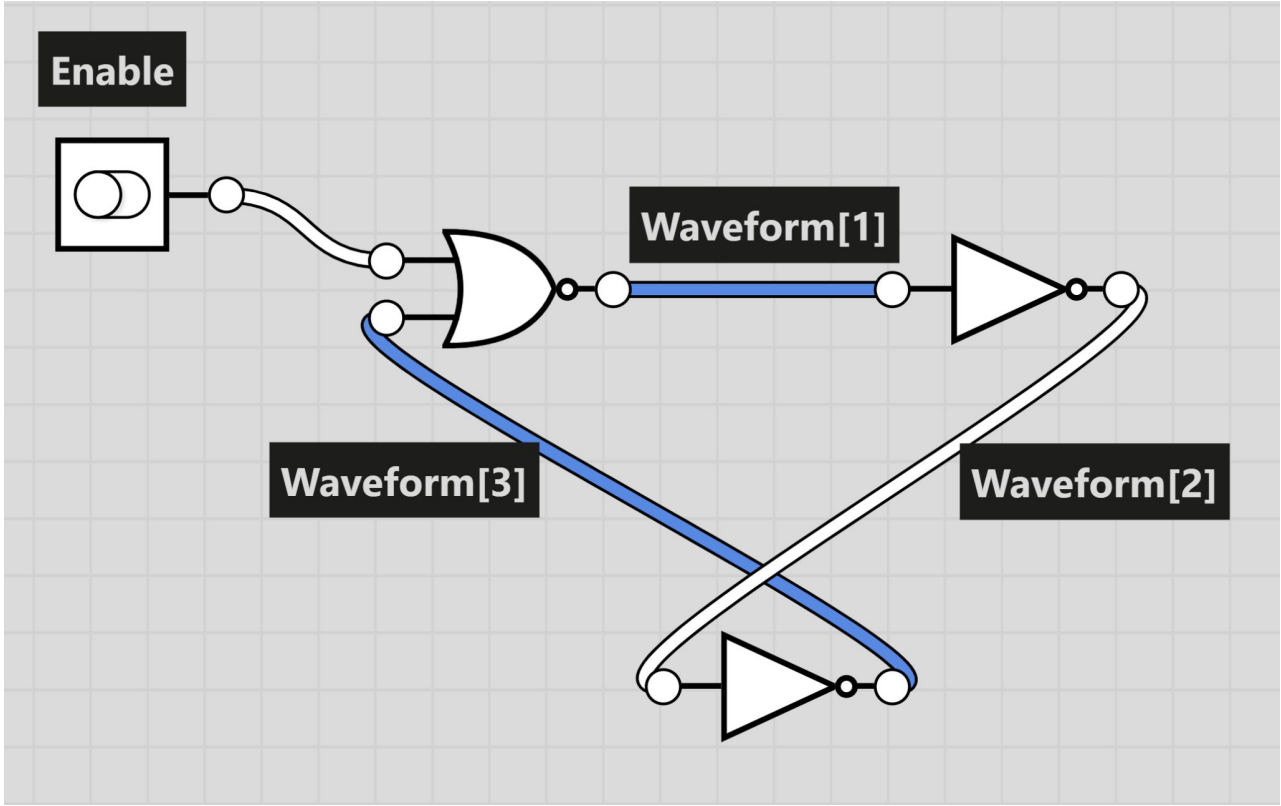
Waveform[1] goes HIGH at **t = 1 ns**



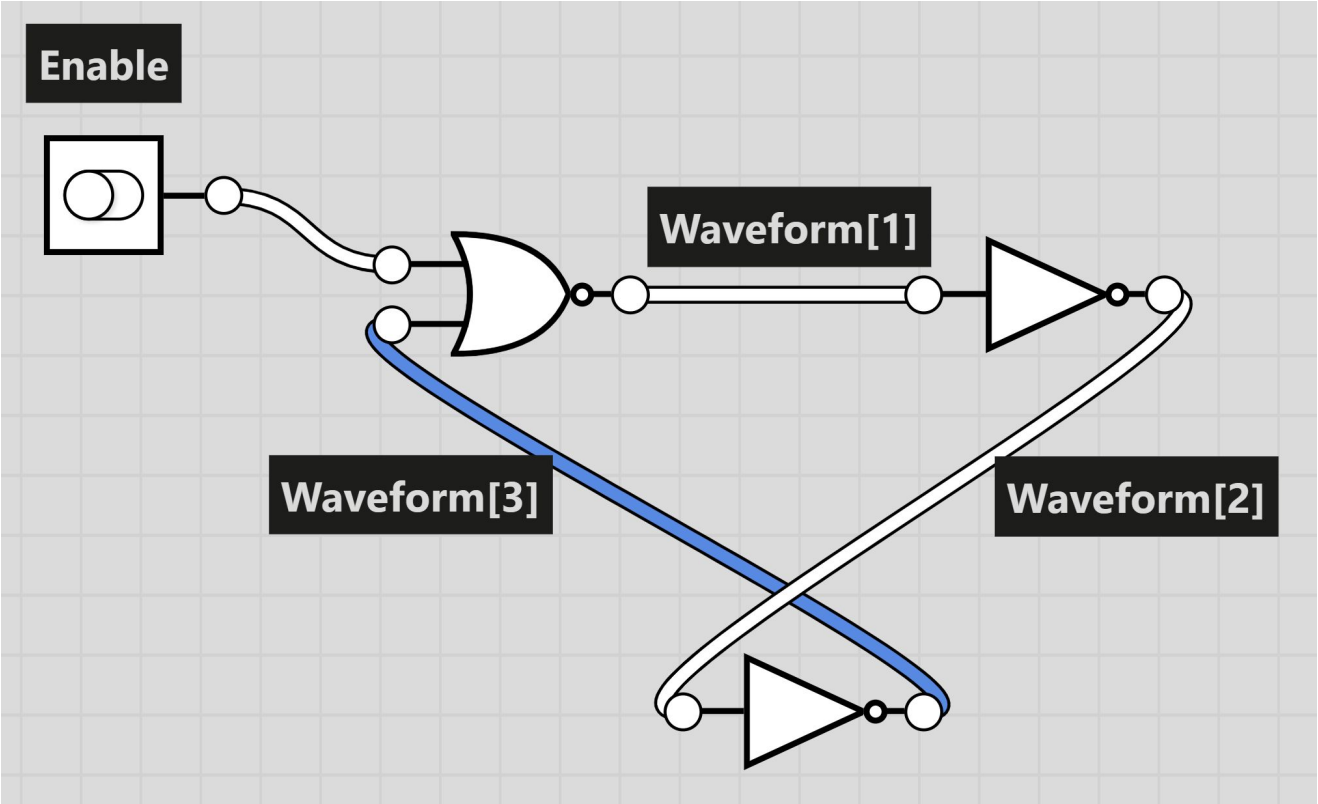
Waveform[2] goes LOW at **t = 2 ns**



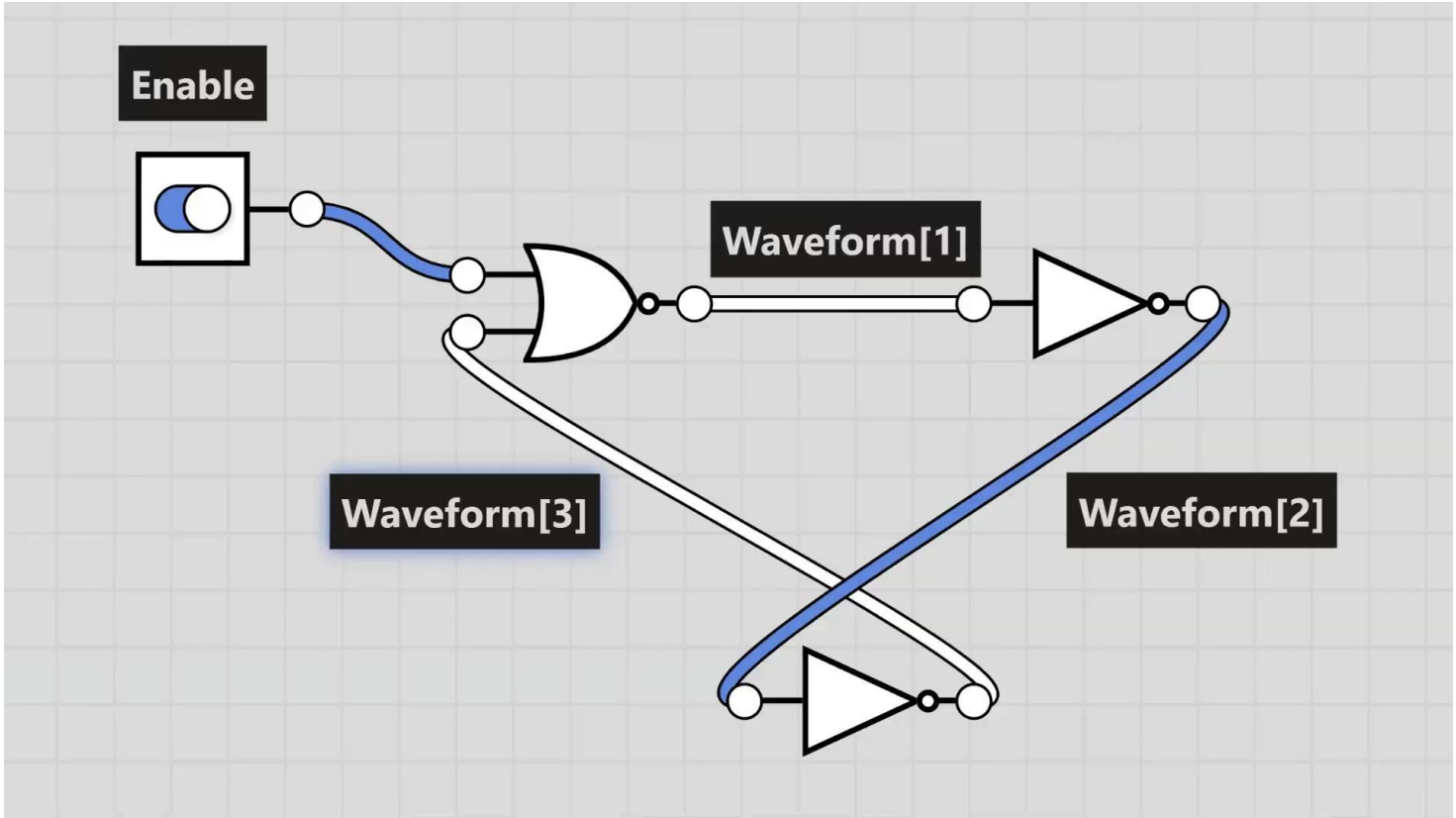
Waveform[3] goes HIGH at **t = 3 ns**



Waveform[1] goes BACK to LOW at **t = 4 ns**

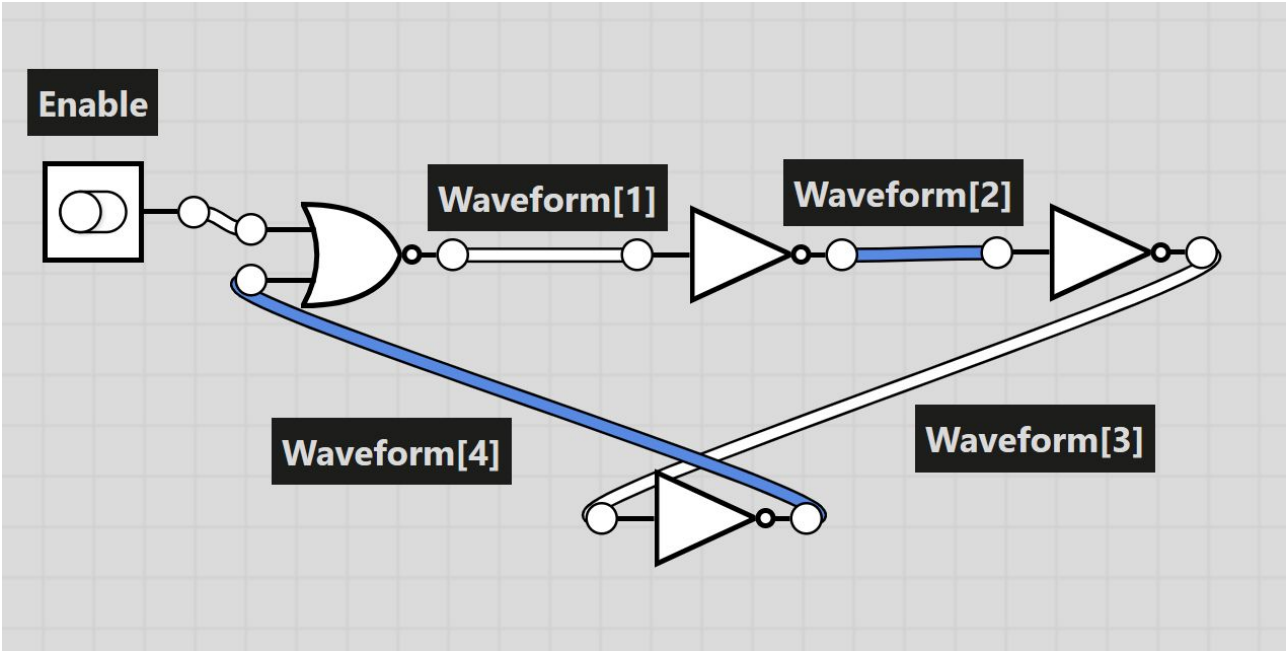


The Ring Oscillator Infinitely Repeats



Bi-Stable Circuit

- Even number of inverting gates → Results in stable circuit



ModelSim Demo 3
(3-Stage Ring Oscillator &
4-Stage Bi-Stable Circuit)

Starter Code for Demo 3

```
// What goes on top?

// Project 2 spec code
module RingOsc3(enable);
  input enable;  // When 1, stop; when 0, start
  wire [3:1] Waveform;

  nor #1 norGate(Waveform[1], Waveform[3], enable);
  not #1 notGate1(Waveform[2], Waveform[1]);
  not #1 notGate2(Waveform[3], Waveform[2]);
endmodule

module Bistable4(enable);

  // Fill out Bistable4 with unit delay

endmodule

module TestbenchRingOsc;

  // Fill out this testbench!

endmodule

module TestbenchBistable;

  // Fill out this testbench!

endmodule
```

Going Over Parameterized Oscillator Code

```
1  // Parameterized Ring Oscillator
2  `timescale 1ns/1ns
3
4  module RingOsc();
5          reg enable;                     // When 1, stop; when 0, start
6          parameter N = 9;                // This is the number of stages and must be odd
7          parameter notDELAY = 10;
8          parameter norDELAY = 15;
9          wire [N:1] Waveform;
10
11 // Generate the oscillator
12         genvar i;
13         generate
14                 for (i = 1; i < N; i = i + 1)
15                         begin : notGates
16                                 not #notDELAY notGate(Waveform[i+1], Waveform[i]);
17                         end
18                 nor #norDELAY norGate(Waveform[1], Waveform[N], enable);
19         endgenerate
20
21         initial begin
22                 enable <= 1; #initDELAYExpression;    // Run long enough to initialize all waveforms
23                 enable <= 0;                          // Start the oscillation
24         end
25 endmodule
```

Setting simulation time unit = 1ns and time precision = 1ns

Defining reusable constant parameters

‘generate’ allows us to instantiate parameter number of NOT gates

generating N NOT gates with Rise/Fall delays of ‘notDELAY’

‘initial’ statement - Executes at t=0 on simulation time

You need to create an expression for initDELAYExpression

THANK YOU

Please fill out form if you haven't already done so!
(Link Here As Well)

(Bonus) Verilog Wires Practice!

Go through these practice problems!

- Attempt these problems **using ONLY Structural Verilog**
  - **DO NOT USE ASSIGN STATEMENTS YET**
  - Hints for the problems are on the following slides

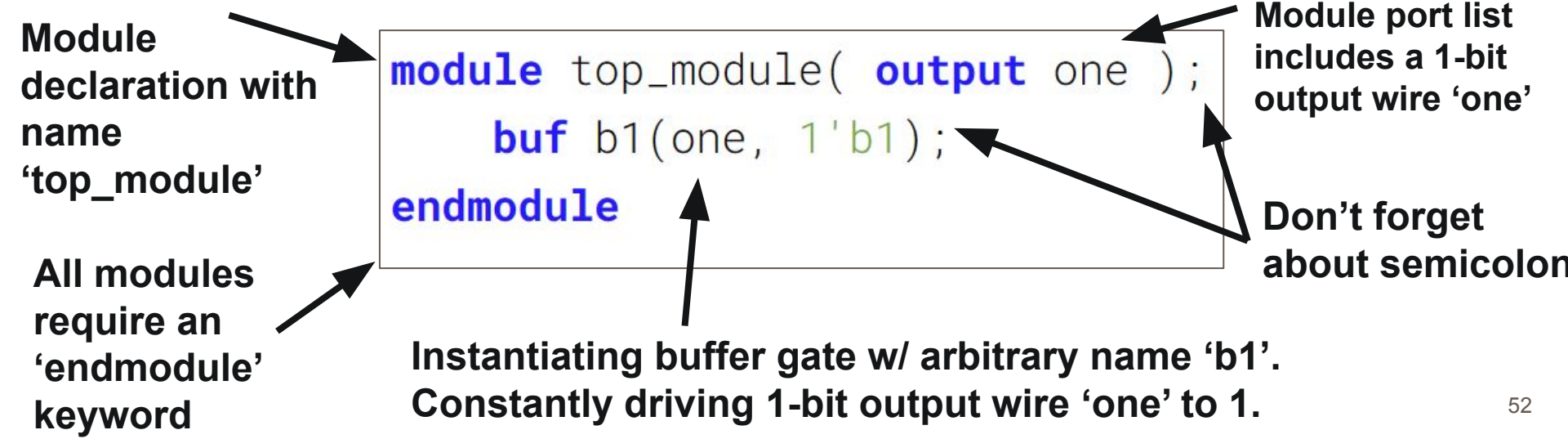
- HDLBits Sections (https://hdlbits.01xz.net/wiki/Problem_sets)
  - Getting Started
  - Verilog Language - Basics
  - Verilog Language - Vectors

- ASK QUESTIONS IF YOU HAVE ANY!

Going over “Getting Started” Problem Together

- Read through everything on your own, it’s more good introduction to Verilog

Build a circuit with no inputs and one output. That output should always drive 1 (or logic high).



(Bonus)
More Capture, Create, Implement
(Combinational Logic Design Process)

Capture, Create, Implement Process

1. **Capture** the behavior of the function
   a. Either through a truth table or equation (Whichever is easier)

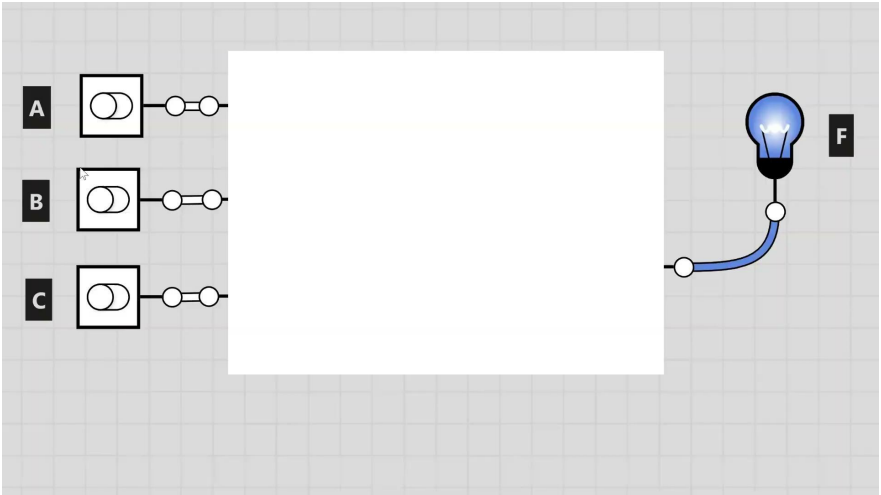
2. **Create** equations (If you haven't already)
   a. OR all of the minterms of the function
      (We will cover minterms more thoroughly in a future lecture)

3. **Implement** a gate-based circuit for <u>each</u> output
   a. Can then translate that gate directly into Structural Verilog

- **Exact process we followed in Project 0.5 (And should follow for Project 1)**

Minority Function Implementation

- **Minority Function**
  - Inputs: 3 1-bit inputs (A, B, C)
  - Outputs: 1-bit output (F)
  - **F will go high if there are one or fewer logical high across the 3 inputs**



Capture the Function Behavior (Truth Table)

Attempt this on your own for a few minutes.

Upcoming Questions you should be able to answer:

1. **Why are we using a truth table?**
2. **What equation do we use do determine the number of rows needed?**
3. **How many rows / combinations do we need for this truth table?**
4. **What are all of the input combinations?**
5. **What are the associated outputs with each of those inputs?**

If you have additional time, begin creating your equation

Capture the Function Behavior (Truth Table)

- **Minority Function**
  - Inputs: 3 1-bit inputs (A, B, C)
  - Outputs: 1-bit output (F)
  - **F will go high if there are one or fewer logical high across the 3 inputs**

A	B	C	F
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Create Your Equation(s)

Attempt this on your own for a few minutes.

Upcoming Questions you should be able to answer:

1. **How many equations do we need to make? Why?**
2. **What is the 3-step process taken to go from truth table → equations?**
3. **What is the final equation after the 3 step process?**

YOU DO NOT NEED TO SIMPLIFY THIS EQUATION

If you have additional time, begin implementing your gate-based circuit

Create Your Equation(s)

1. Determine which row resulted in ‘F’ being True (1)

2. Generate an minterm for Each Combination
   a. Only that combination should result in output being true
   b. Here’s a short video about minterms

For combination (0, 0, 0) → A’ & B’ & C’

The only time this expression is True is when (0, 0, 0) is passed

Fot combination (0, 0, 1) → A’ & B’ & C

The only time this expression is True is when (0, 0, 1) is passed

3. OR each of the minterms

A	B	C	F
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

F = (A’ & B’ & C’) | (A’ & B’ & C) | (A’ & B & C’) | (A & B’ & C’)

Implement Your Gate-Based Circuit

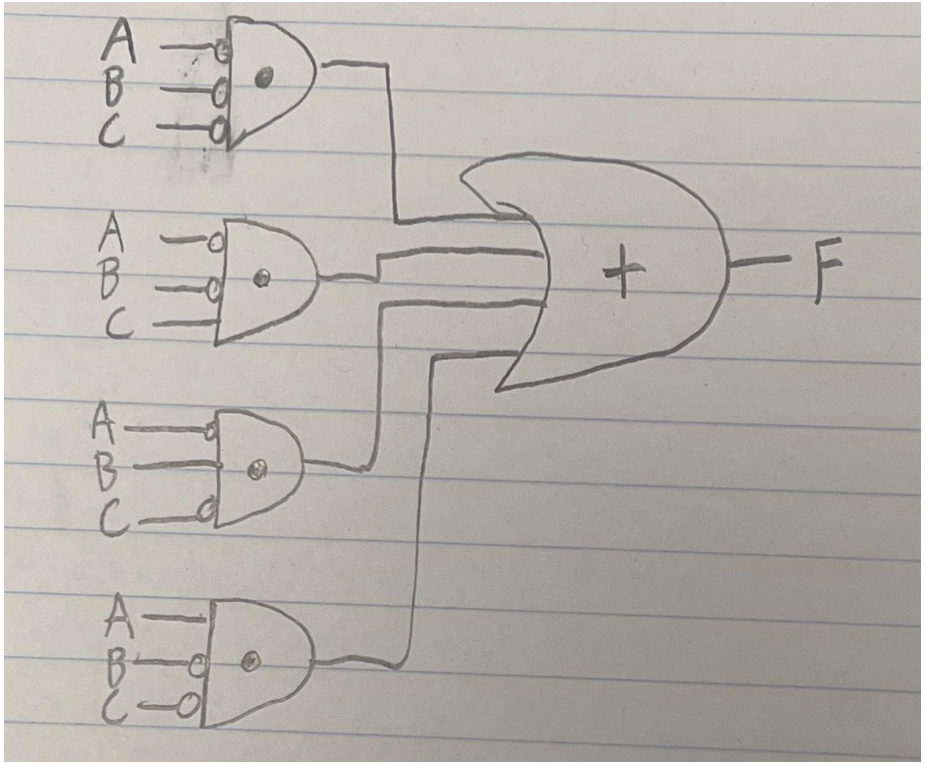
F =

(A’ & B’ & C’) |

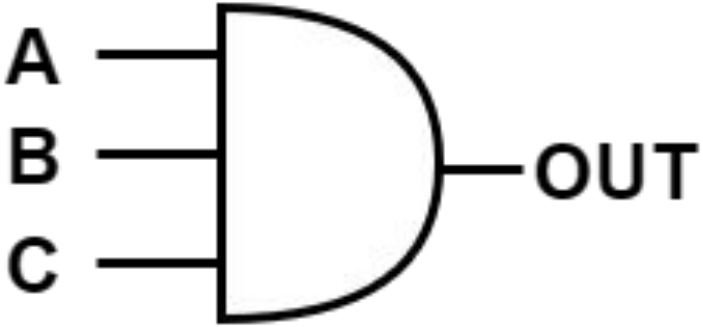
(A’ & B’ & C) |

(A’ & B & C’) |

(A & B’ & C’)

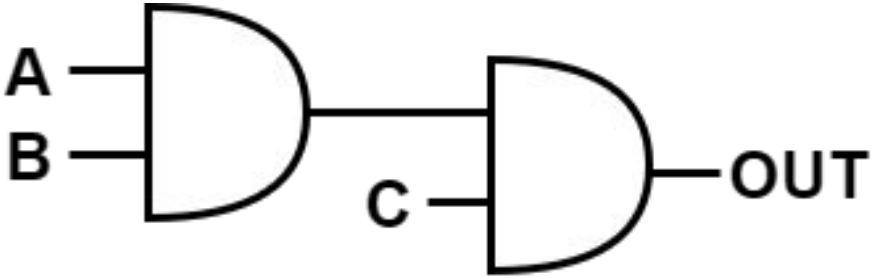


AND Gates Implementations in Structural Verilog



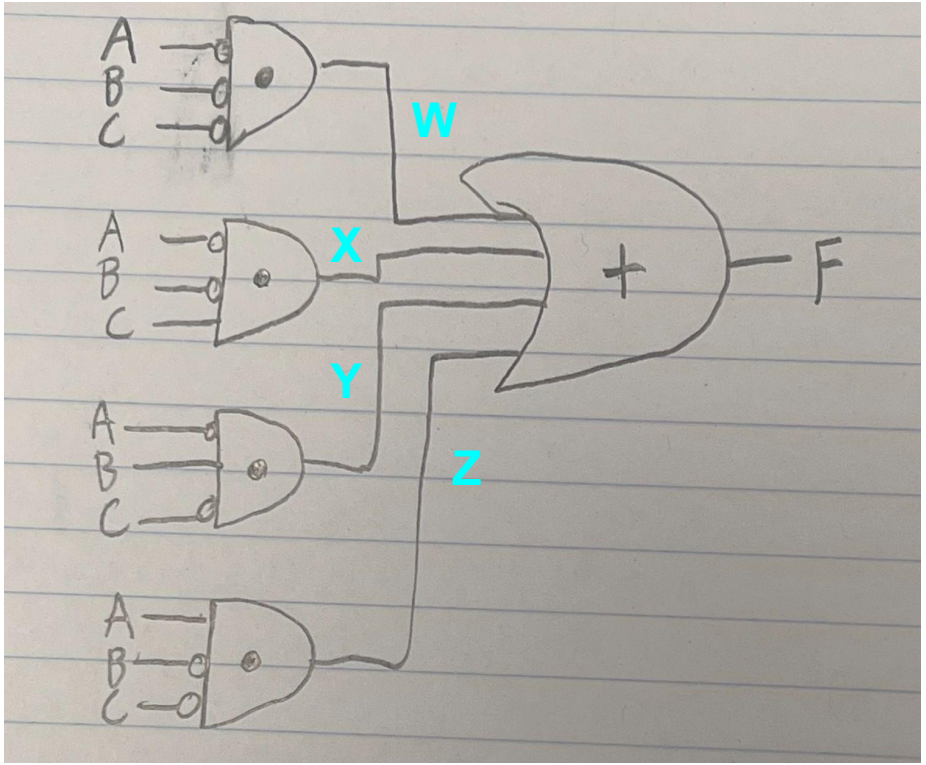

```
and a1(out, a, b, c);
```

Can define more than 2 inputs for input list!!




```
wire ab;
and a1(ab, a, b);
and a2(out, ab, c);
```

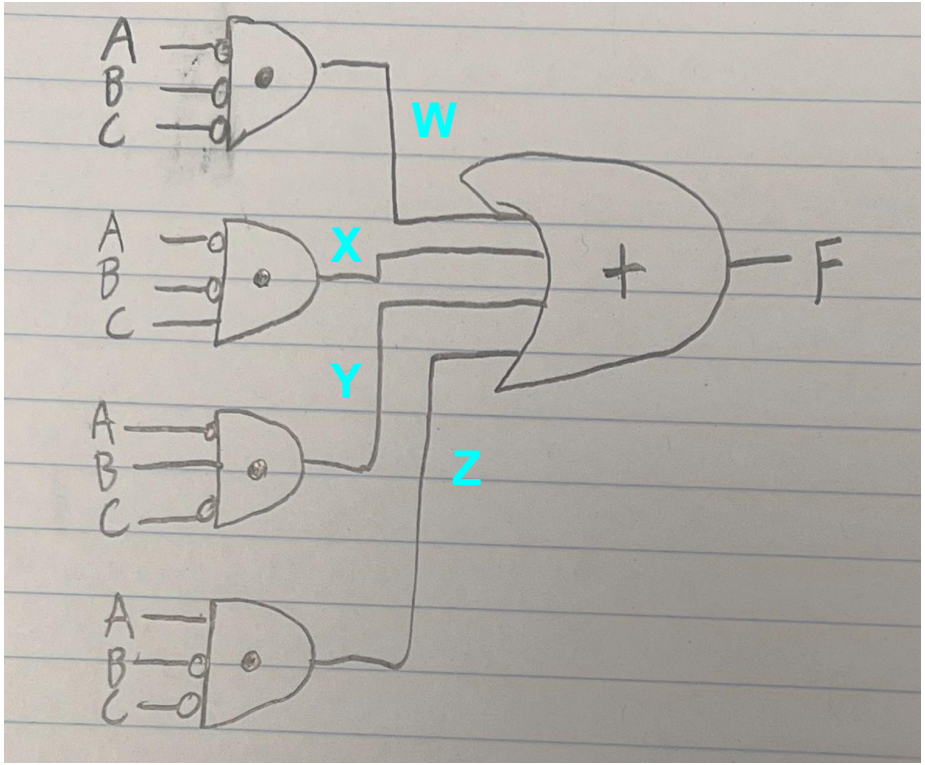
Implementation in Verilog (Declaring Wires)



Now try to write this in Verilog!
Here is starter code

```
module MockProjectTopLevel(
  input [2:0] SW,
  output [0:0] LEDR
);

  Minority m(SW[2], SW[1], SW[0], LEDR);

endmodule

module Minority(
  input a, b, c,
  output out
);

  // Step 1. Declare your wires

  // Step 2. Drive some not a, b, c signals

  // Step 3. Drive w, x, y, z

  // Step 4. Drive our output

endmodule
```

Implementation in Verilog

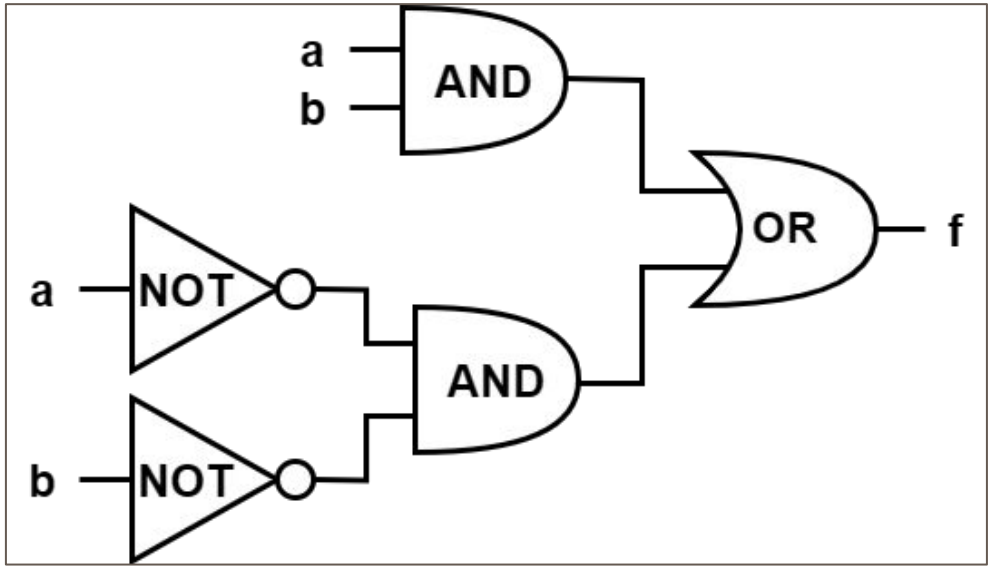



```
module MickProjectTopLevel(
  input [2:0] SW,
  output [0:0] LEDR
);

  Minority m(SW[2], SW[1], SW[0], LEDR);

endmodule

module Minority(
  input a, b, c,
  output out
);

  // Step 1. Declare your wires
  wire na, nb, nc;
  wire w, x, y, z;

  // Step 2. Drive some not a, b, c signals
  not n1(na, a);  // a'
  not n2(nb, b);  // b'
  not n3(nc, c);  // c'

  // Step 3. Drive w, x, y, z
  and a1(w, na, nb, nc);  // a' & b' & c'
  and a2(x, na, nb, c);   // a' & b' & c
  and a3(y, na, b, nc);   // a' & b & c'
  and a4(z, a, nb, nc);   // a & b' & c'

  // Step 4. Drive our output
  or o1(out, w, x, y, z);

endmodule
```

(Bonus) Specified Delay Propagation Calculation Slides

Systematic Calculation of LONGEST Delay

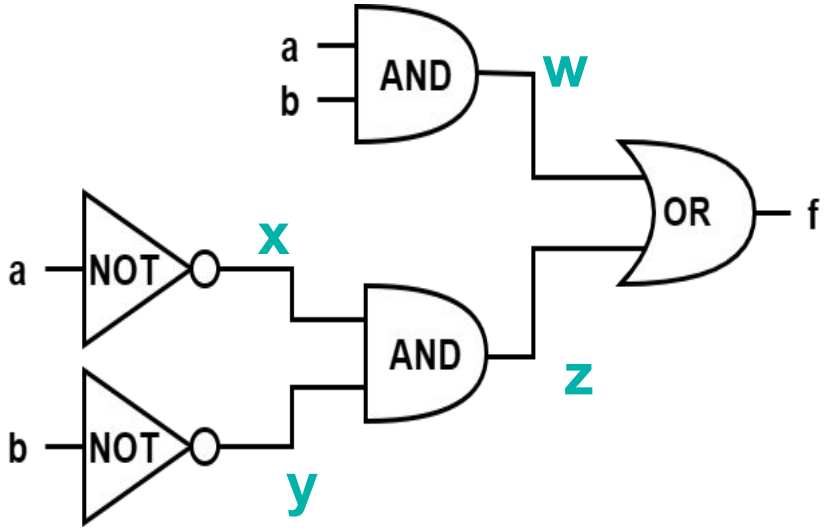
- Focussing on toggling one input at a time
- Goal: Want to determine longest propagation delay from toggling input → output



Finding Longest Delay with Toggling ‘a’

1. Determine all of the combinations of inputs
   a. Not including the input we are toggling

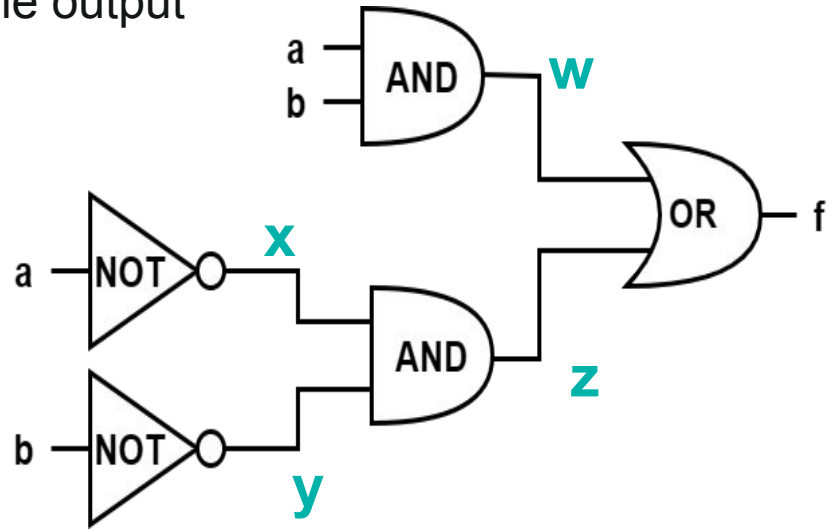
b	w	x	y	z	f
0					
1					



Finding Longest Delay with Toggling ‘a’

2. Determine if the toggling input will even affect the output
   a. Write each internal variable in terms of the toggling variable
   b. See how the toggling input affects the output

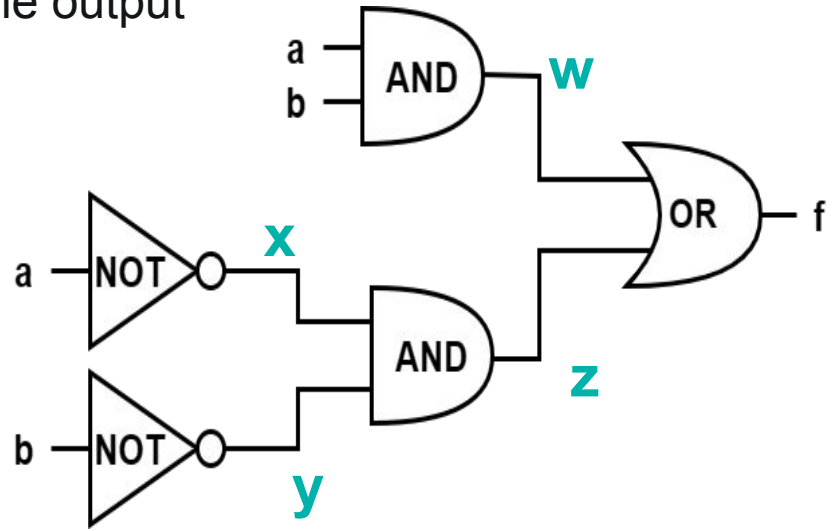
b	w	x	y	z	f
0					
1					



Finding Longest Delay with Toggling ‘a’

2. Determine if the toggling input will even affect the output
   a. Write each internal variable in terms of the toggling variable
   b. See how the toggling input affects the output

b	w	x	y	z	f
0	0				
1	a				

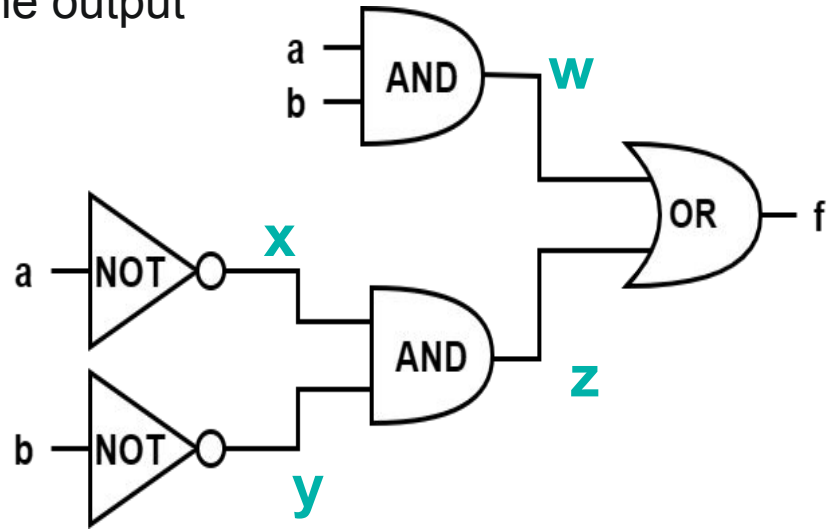


If b = 0, then w will always be 0

Finding Longest Delay with Toggling ‘a’

2. Determine if the toggling input will even affect the output
   a. Write each internal variable in terms of the toggling variable
   b. See how the toggling input affects the output

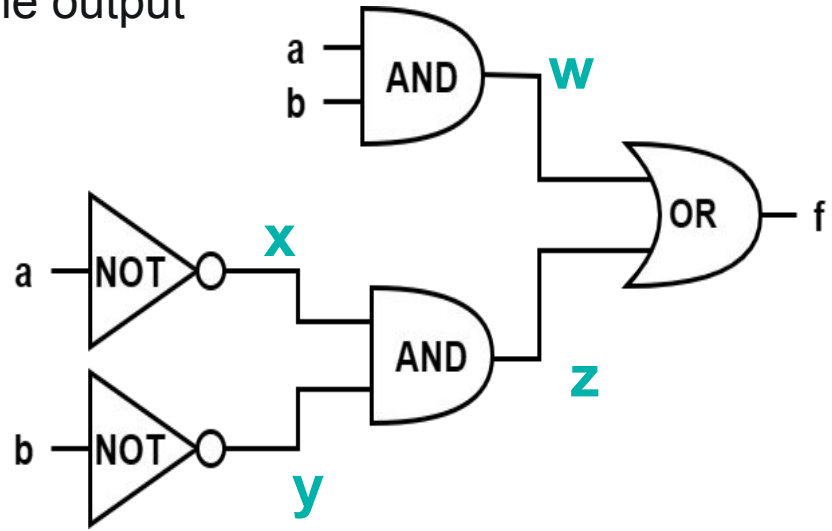
b	w	x	y	z	f
0	0	a’			
1	a	a’			



Finding Longest Delay with Toggling ‘a’

2. Determine if the toggling input will even affect the output
   a. Write each internal variable in terms of the toggling variable
   b. See how the toggling input affects the output

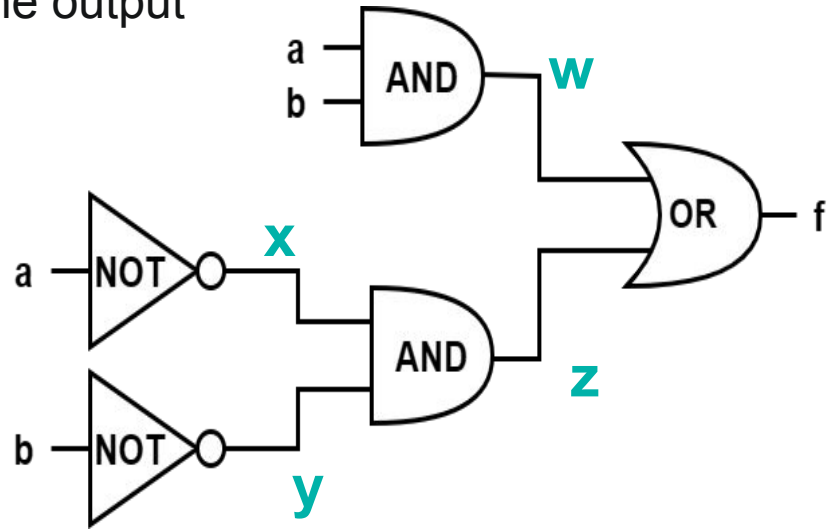
b	w	x	y	z	f
0	0	a’	1		
1	a	a’	0		



Finding Longest Delay with Toggling ‘a’

2. Determine if the toggling input will even affect the output
   a. Write each internal variable in terms of the toggling variable
   b. See how the toggling input affects the output

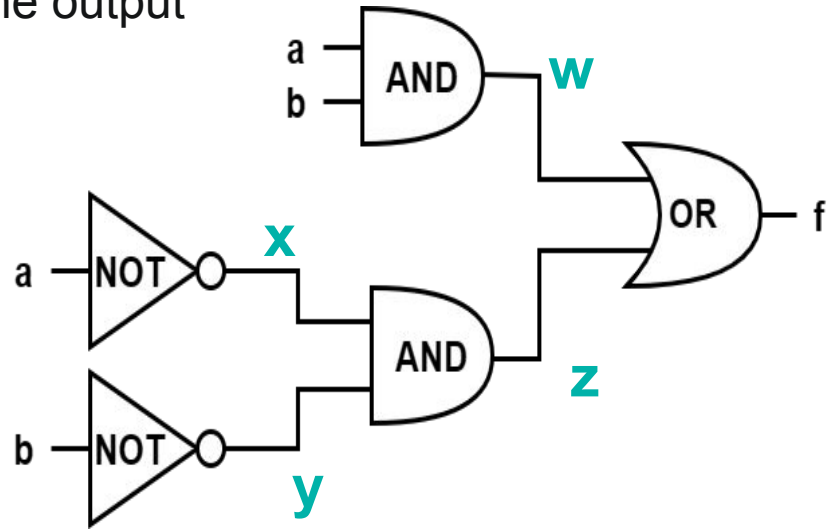
b	w	x	y	z	f
0	0	a’	1	a’	
1	a	a’	0	0	



Finding Longest Delay with Toggling ‘a’

2. Determine if the toggling input will even affect the output
   a. Write each internal variable in terms of the toggling variable
   b. See how the toggling input affects the output

b	w	x	y	z	f
0	0	a’	1	a’	**a’**
1	a	a’	0	0	**a**



We need to check all cases that ‘a’ affects ‘f’ to determine which causes the longest delay

Finding Longest Delay with Toggling ‘a’

3. Use ModelSim to determine propagation delay
    a. Or calculate by hand (Try to get good with ModelSim though)

Goal: Which causes the longest $t^{a \rightarrow f}_{pLH}$ delay (The longest delay of ‘f’ going from L→H because of toggling ‘a’)

Case 1: Have b = 0 and toggle ‘a’ from H → L
Case 2: Have b = 1 and toggle ‘a’ from L → H

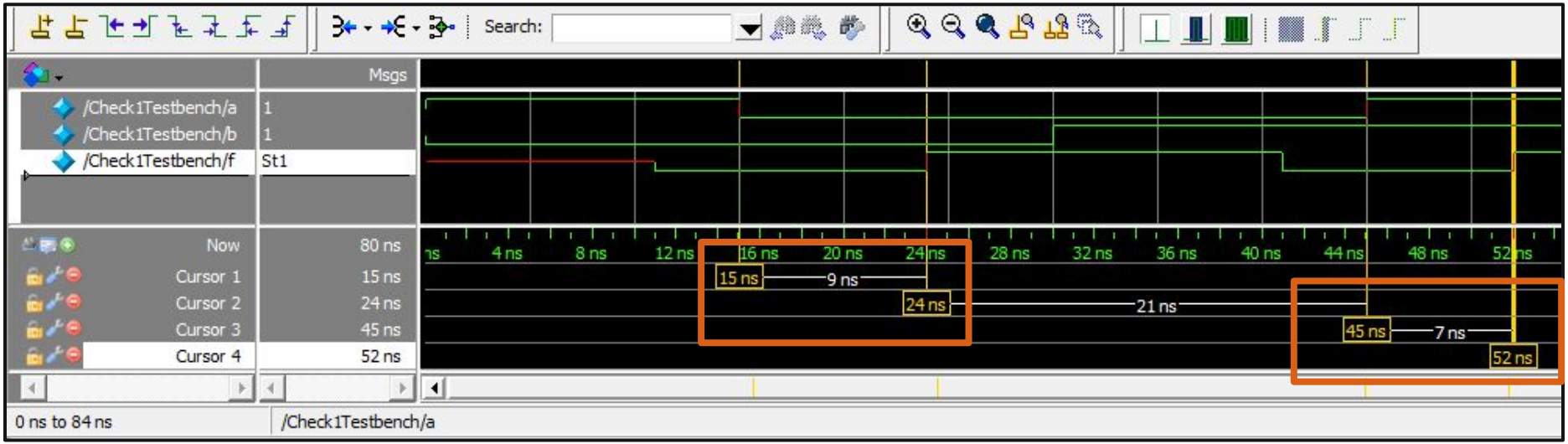
b	**w**	**x**	**y**	**z**	**f**
0	0	a’	1	a’	**a’**
1	a	a’	0	0	**a**

Follow Along! Update Your CHECK1 Module

```
// timescale is now needed to indicate time unit to simulator
`timescale 1ns/1ns

module CHECK1(
  input a, b,
  output f
);
  // Parameters - Reusable constants in Verilog
  parameter notDelay = 2;  // Rise/Fall Delay of NOT gates is 2 ns
  parameter andDelay = 4;  // Rise/Fall Delay of AND gates is 4 ns

  // Step 1. Declare Your Wires
  // Declaring 1-bit wires w, x, y, z
  wire w, x, y, z;

  // Step 2. Instantiate the gates
  // NEW: Specifing parameterized non-unit delays
  and #andDelay a1(w, a, b);
  not #notDelay n1(x, a);
  not #notDelay n2(y, b);
  and #andDelay a2(z, x, y);

  // Indicating different Rise/Fall delays
  // tpLH = 3 ns (Z goes from L-->H at 3 ns after input change)
  // tpHL = 5 ns (Z goes from H-->L at 5 ns after input change)
  or  #(3, 5) o1(f, w, z);

endmodule // CHECK1
```

```
// Timescale on the top
// `timescale <time_unit>/<time_precision>
// My preference, use 1ns/1ns
`timescale 1ns/1ns

// Testbench module has no inputs/outputs
module Check1Testbench;

    // All controlable inputs to DUT declared as reg
    reg a, b;

    // All outputs of DUT declared as wire
    wire f;

    // Instantiate module we want to test
    // DUT - Device Under Testing
    CHECK1 dut(a, b, f);

    // Initial statement use to contorl inputs
    // Initial means right at t = 0
    initial begin

        // Case #1: Have b = 0 and toggle 'a' from H → L
        a = 1'b1;
        b = 1'b0;

        // Wait statement TO LET SIGNAL PASS THROUGH
        // Waiting 15 time units --> 15 ns
        #15;

        // Now toggling 'a' to Low to observe 'f' going High
        // Now calculate propagation from 'a' → 'f'
        a = 1'b0;

        // Wait statement TO LET SIGNAL PASS THROUGH
        #15;

        // Case #2: Have b = 1 and toggle 'a' from L → H
        a = 1'b0;
        b = 1'b1;

        #15;

        // Now toggling 'a' to High to observe 'f' going High
        // Now calculate propagation from 'a' → 'f'
        a = 1'b1;

    end  // initial

endmodule  // Testbench
```

Determining Longest $t^{a \rightarrow f}_{pLH}$ Delay



Case #1: b = 0 and a = 1 → 0 **Propagation Delay = 9 ns**

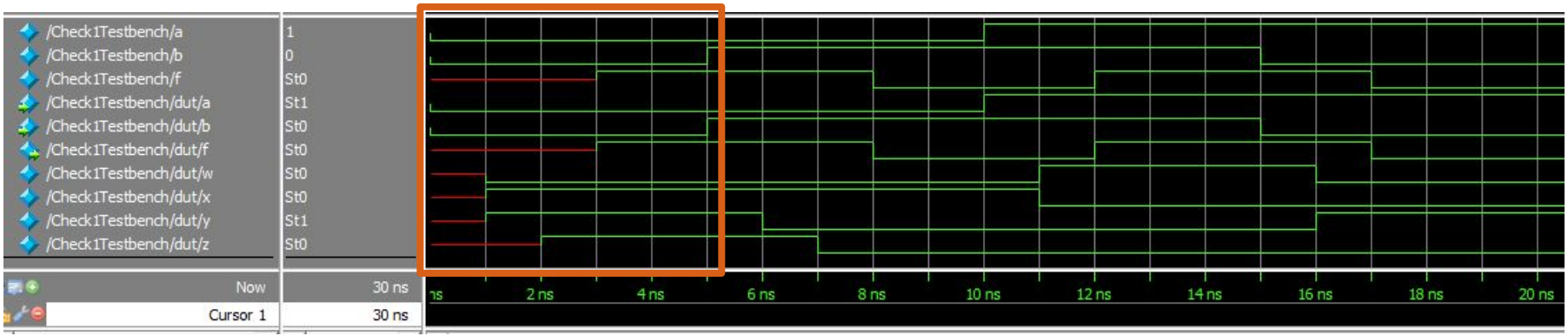
Case #2: b = 1 and a = 0 → 1 Propagation Delay = 7 ns

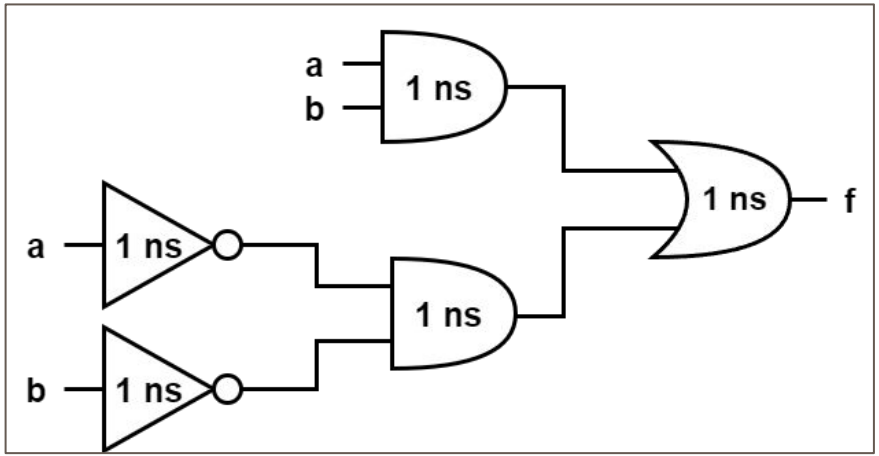
Additional ModelSim Slides

Updating CHECK1 for Unit Delay

```
// NEW: timescale is now needed to indicate time unit to simulator
`timescale 1ns/1ns

module CHECK1(
  input a, b,
  output f
);

  // Step 1. Declare Your Wires
  // Declaring 1-bit wires w, x, y, z
  wire w, x, y, z;

  // Step 2. Instantiate the gates
  // NEW: ADDING SIMULATED DELAYS TO OUR GATES
  // NEW: Specifying Unit Delays (From both L→H and H→L)
  and #1 a1(w, a, b);
  not #1 n1(x, a);
  not #1 n2(y, b);
  and #1 a2(z, x, y);
  or  #1 o1(f, w, z);

endmodule // CHECK1
```

Let's Look at the ModelSim Output

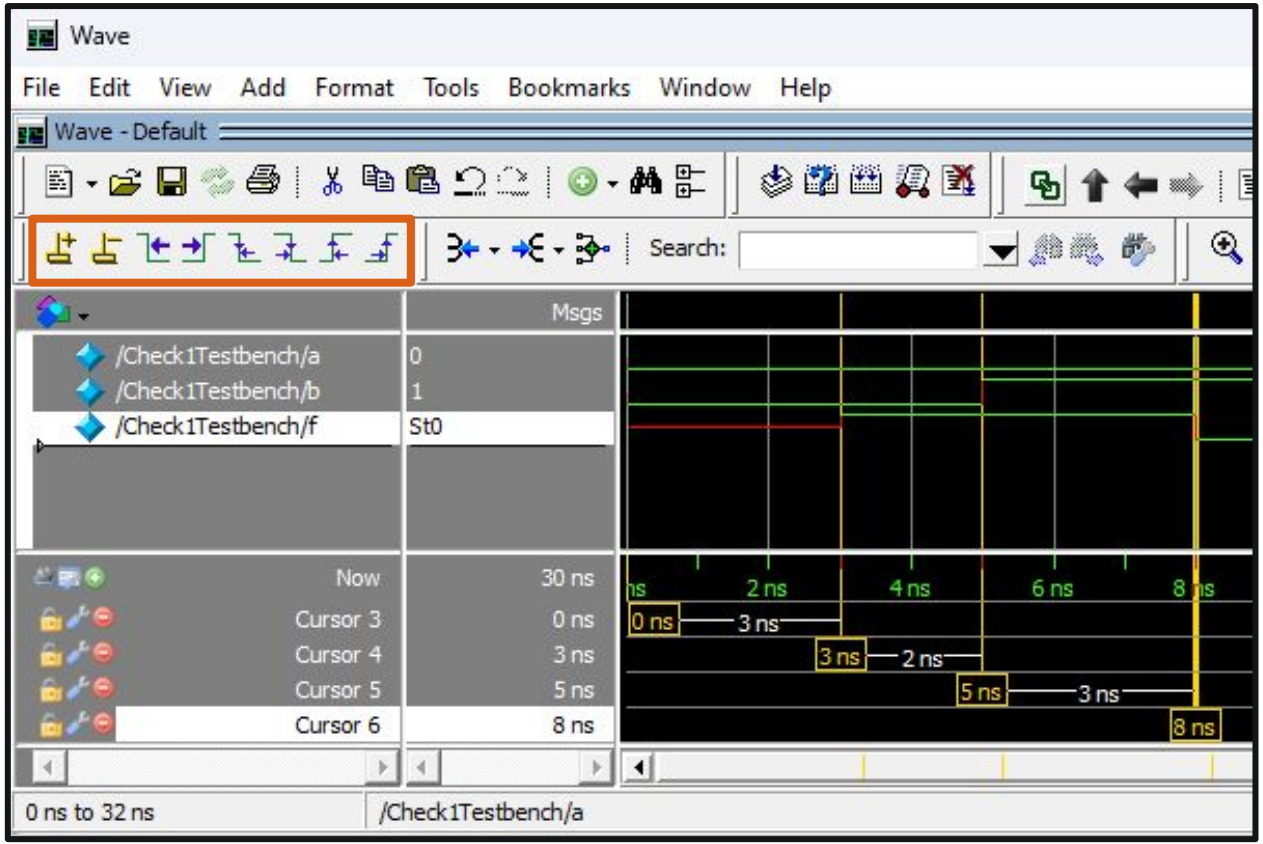


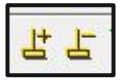


Need to give time to circuit to fully stabilize at the beginning of your simulation!!

```
`timescale 1ns/1ns
module Check1Testbench;
   …
   initial begin
      // assign a and b to logical low at t=0
      a = 1'b0; b = 1'b0;

      // Wait statement TO LET SIGNAL PASS THROUGH
      #5;
      …
   end  // initial
endmodule  // Testbench
```

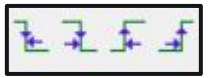
Cursors in ModelSim (Propagation Calculations)





Adding/Removing Cursors

Previous/Next Signal Transition



Previous/Next Rising/Fall Edge Signal Transition