



Digital System Design

Hajar Falahati

hfalahati@ipm.ir
hfalahati@ce.sharif.edu

Using modules

- There are some built-in primitive logic gates in Verilog that can be instantiated
 - Built-in primitives means there is no need to define a module for these gates
 - and, or, nor,

1. Gate instantiation → `and myand (out, ln1, ln2);`

2. assign → `assign out = ln1 & ln2;`

3. always block → `reg out;
always @(ln1, ln2)
out = (ln1 & ln2);`

```
module Myand(ln1, ln2, out)
  input ln1, ln2;
  output out;

  and myand (out, ln1, ln2);

  assign out = ln1 & ln2;







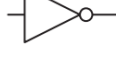
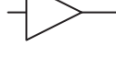
  reg out;
  always @(ln1, ln2)
    out = (ln1 & ln2);





endmodule
```



Primitive Gates

Logic Gates		
and	or	xor
nand	nor	xnor
Buffers		
buf	bufif0	bufif1
not	notif0	notif1
pulldown	pullup	

n-Input	n-Output , 3-state
and	buf
nand	not
or	bufif0
nor	bufif1
xor	notif0
xnor	notif1

and (w, i ₁ , i ₂ ...)	
nand (w, i ₁ , i ₂ ...)	
or (w, i ₁ , i ₂ ...)	
nor (w, i ₁ , i ₂ ...)	
xor (w, i ₁ , i ₂ ...)	
xnor (w, i ₁ , i ₂ ...)	
not (w, i)	
buf (w, i)	

bufif1 (w, i, c)	
bufif0 (w, i, c)	
notif1 (w, i, c)	
notif0 (w, i, c)	

pullup w	
pulldown w	

Operators

Operator category	Operators symbol
Arithmetic	* / + - % **
Logical	! &&
Relational	> < <= >=
Equality	== != === !==
Bitwise	~ & ^ ^~ ~^
Reduction	& ~& ~ ^ ~^ ^~
Shift	>> << >>> <<<
Concatenation	{ }
Replication	{ { } }
Conditional	? :

Outline

- Verification



Verification

How Do You Know That A Circuit Works?

- You have designed a circuit
 - Is its **functionally** correct?
 - Even if it is logically correct, does the hardware meet all **timing** constraints?
- How can you **test** for:
 - Functionality?
 - Timing?
- Answer: **simulation tools!**
 - Formal verification tools (e.g., SAT solvers)
 - HDL timing simulation (e.g., Vivado)
 - Circuit simulation (e.g., SPICE)

Testing Large Digital Designs

- Testing can be the **most time consuming** design stage
 - Functional correctness of **all logic paths**
 - Timing, power, etc. of **all circuit elements**
- **Low-level** (e.g., circuit) simulation is **much slower** than **high-level** (e.g., HDL, C) simulation

Testing Large Digital Designs: Solution

- We split responsibilities:
 - 1) Check **only functionality** at a **high level** (e.g., C, HDL)
 - (Relatively) **fast** simulation time allows **high code coverage**
 - **Easy** to write and run tests
 - 2) Check **only timing, power**, etc. at **low level** (e.g., circuit)
 - **No functional testing** of low-level model
 - Instead, test **functional equivalence** to high-level model
 - **Hard**, but **easier** than testing logical functionality at this level

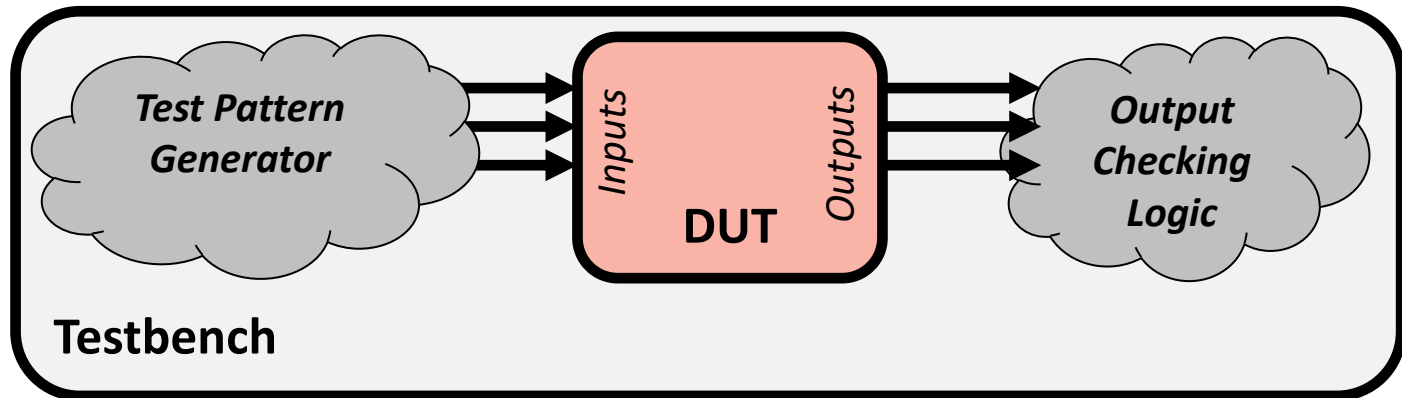
Functional Verification

Functional Verification

- Goal:
 - Check **logical correctness** of the design
- **Physical circuit timing** (e.g., $t_{\text{setup}}/t_{\text{hold}}$) is typically **ignored**
 - May implement simple checks to catch **obvious bugs**
- There are two primary approaches
 - **Logic simulation** (e.g., C/C++/Verilog test routines)
 - **Formal verification** techniques
- In this course, we will use Verilog for functional verification

Testbench-Based Functional Testing

- **Testbench:** a module created specifically to test a design
 - Tested design is called the “**device under test (DUT)**”



- **Testbench provides inputs (test patterns)** to the DUT
 - Hand-crafted values
 - Automatically generated (e.g., sequential or random values)
- **Testbench checks outputs** of the DUT against:
 - Hand-crafted values
 - A “golden design” that is known to be bug-free

More on Testbench-Based Functional Testing

- A testbench can be:
 - **HDL code** written to test other HDL modules
 - **Circuit schematic** used to test other circuit designs
- Testbench is **not** designed for hardware synthesis!
 - Runs in **simulation** only
 - HDL simulator (e.g., Vivado simulator)
 - SPICE circuit simulation
 - Testbench uses **simulation-only** constructs
 - E.g., “wait 10ns”
 - E.g., ideal voltage/current source
 - Not suitable to be physically built!

Common Verilog Testbench Types

Testbench	Input/Output Generation	Error Checking
Simple	Manual	Manual
Self-	Manual	Automatic
Automatic	Automatic	Automatic

Let's Start Verification

- We will walk through different types of testbenches to test a module that implements the logic function:

$$y = (\bar{b} \cdot \bar{c}) + (a \cdot \bar{b})$$

Example DUT

$$y = (\bar{b} \cdot \bar{c}) + (a \cdot \bar{b})$$

```
// performs  $y = \sim b \ \& \ \sim c \mid a \ \& \ \sim b$ 
module sillyfunction(input  a, b, c, output y);
    wire b_n, c_n;
    wire m1, m2;

    not not_b(b_n, b);
    not not_c(c_n, c);

    and minterm1(m1, b_n, c_n);
    and minterm2(m2, a, b_n);
    or  out_func(y, m1, m2);
endmodule
```


Useful Verilog Syntax for Testbenching

```
module example_syntax();  
    reg a;  
  
    // like "always" block, but runs only once at sim start  
    initial  
    begin  
        a = 0; // set value of reg: use blocking assignments  
        #10;   // wait (do nothing) for 10 ns  
        a = 1;  
        $display("printf() style message!"); // print message  
    end  
endmodule
```

Simple Testbench

Simple Testbench

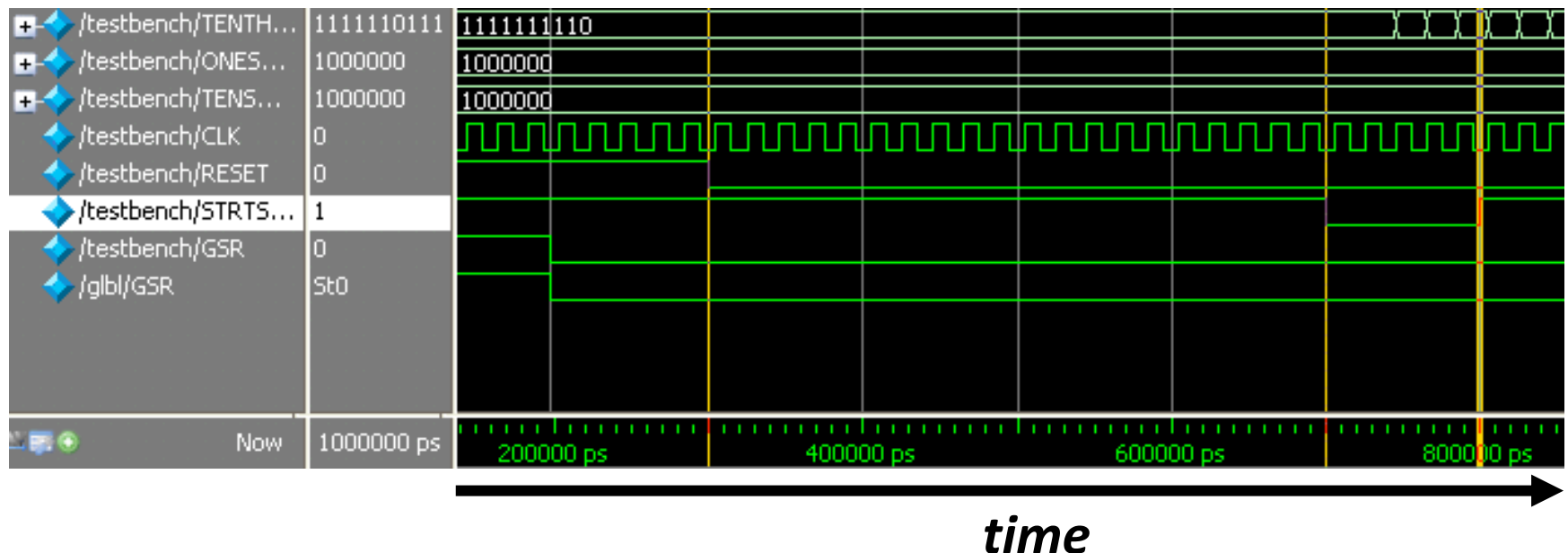
```
module testbench1(); // No inputs, outputs
    reg a, b, c;      // Manually assigned
    wire y;           // Manually checked

    // instantiate device under test
    sillyfunction dut (.a(a), .b(b), .c(c), .y(y) );

    // apply hardcoded inputs one at a time
    initial begin
        a = 0; b = 0; c = 0; #10; // apply inputs, wait 10ns
        c = 1; #10;                // apply inputs, wait 10ns
        b = 1; c = 0; #10;         // etc .. etc..
        c = 1; #10;
        a = 1; b = 0; c = 0; #10;
    end
endmodule
```

Simple Testbench: Output Checking

- Most common method is to look at **waveform diagrams**
 - **Thousands** of signals over **millions** of clock cycles
 - Too many to just printf()!
- **Manually check** that output is correct **at all times**



Simple Testbench

Pros:

- Easy to design
- Can easily test a few, specific inputs (e.g., corner cases)

Cons:

- **Not scalable** to many test cases
- Outputs must be checked **manually** outside of the simulation
 - E.g., inspecting dumped waveform signals
 - E.g., printf() style debugging

Self-Checking Testbench

Self-Checking Testbench

```
module testbench2();  
    reg  a, b, c;  
    wire y;  
    sillyfunction dut (.a(a), .b(b), .c(c), .y(y) );  
    initial begin  
        a = 0; b = 0; c = 0; #10; // apply input, wait 10ns  
        if (y !== 1) $display("000 failed."); // check result  
        c = 1; #10;  
        if (y !== 0) $display("001 failed.");  
        b = 1; c = 0; #10;  
        if (y !== 0) $display("010 failed.");  
    end  
endmodule
```

Self-Checking Testbench

Pros:

- Still easy to design
- Still easy to test a few, specific inputs (e.g., corner cases)
- **Simulator will print** whenever an error occurs

Cons:

- **Still not scalable** to millions of test cases
- Easy to make an **error** in **hardcoded** values
 - You make just as many **errors** writing a testbench as actual code
 - **Hard to debug** whether an issue is in the testbench or in the DUT

Self-Checking Testbench using Test vectors

- Write *testvector file*
 - List of inputs and expected outputs
 - Can create vectors **manually** or **automatically** using an already verified, simpler “**golden model**” (more on this later)

Example file:

```
$ cat testvectors.tv
```

```
000_1
```

```
001_0
```

```
010_0
```

```
011_0
```

```
100_1
```

```
101_1
```

```
110_0
```

```
111_0
```

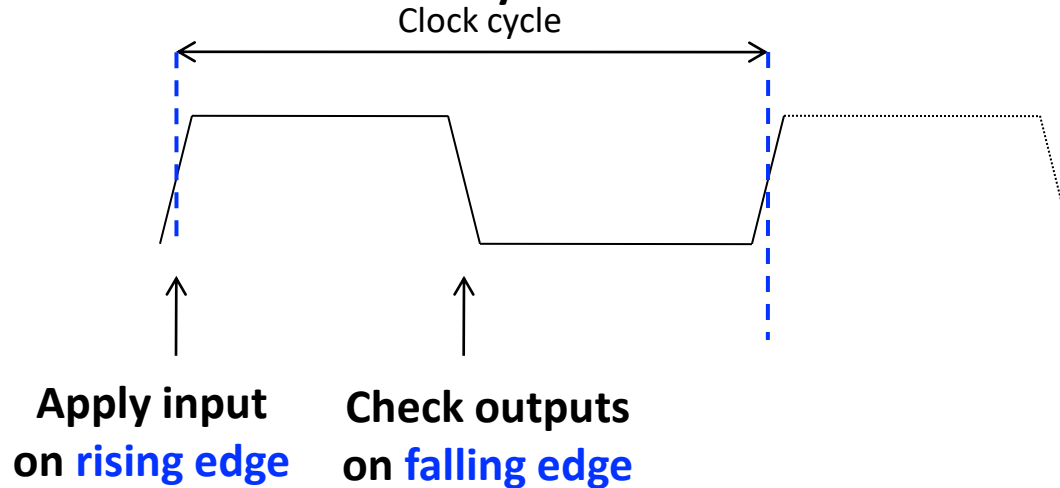
```
...
```

Format: **input_output**



Testbench with Testvectors Design

- Use a “**clock signal**” for assigning inputs, reading outputs
 - Test one **testvector** each “**clock cycle**”



- Note: “clock signal” simply separates **inputs** from **outputs**
 - Allows us to *observe* the inputs/outputs in waveform diagrams
 - Not used for checking physical circuit timing (e.g., t_{setup} / t_{hold})

Testbench Example (1/5): Signal Declarations

- Declare signals to hold internal state

```
module testbench3();  
  
    reg          clk, reset;           // clock and reset are internal  
    reg          a, b, c, yexpected;  // values from testvectors  
    wire         y;                   // output of circuit  
    reg [31:0]    vectornum, errors;   // bookkeeping variables  
    reg [3:0]     testvectors[10000:0]; // array of testvectors  
  
    // instantiate device under test  
    sillyfunction dut(.a(a), .b(b), .c(c), .y(y) );
```

Testbench Example (2/5): Clock Generation

```
// generate clock
always      // no sensitivity list, so it always executes
begin
    clk = 1; #5; clk = 0; #5;      // 10ns period
end
```

Testbench Example (3/5): Read Testvectors into Array

```
// at start of test, load vectors and pulse reset

initial    // Only executes once
begin

    $readmemb("example.tv", testvectors); // Read vectors
    vectornum = 0; errors = 0;              // Initialize
    reset = 1; #27; reset = 0;             // Apply reset wait

end

// Note: $readmemb reads testvector files written in
// hexadecimal
```

Testbench Example (4/5): Assign Inputs/Outputs

```
// apply test vectors on rising edge of clk
always @(posedge clk)
begin
    {a, b, c, yexpected} = testvectors[vectornum];
end
```

- Apply {**a**, **b**, **c**} inputs on the *rising edge* of the clock
- Get **yexpected** for checking the output on the *falling edge*
- Rising/falling edges are chosen only by convention
 - You can use any part of the clock signal

Testbench Example (5/5): Check Outputs

```
always @(negedge clk)
begin
    if (~reset) // don't test during reset
    begin
        if (y !== yexpected)
        begin
            $display("Error: inputs = %b", {a, b, c});
            $display("  outputs = %b (%b exp)", y, yexpected);
            errors = errors + 1;
        end

        // increment array index and read next testvector
        vectornum = vectornum + 1;

        if (testvectors[vectornum] === 4'bx)
        begin
            $display("%d tests completed with %d errors",
                vectornum, errors);
            $finish;                // End simulation
        end
    end
end
end
```

Self-Checking Testbench with Testvectors

Pros:

- Still easy to design
- Still easy to test a few, specific inputs (e.g., corner cases)
- Simulator will print whenever an error occurs
- **No need** to change hardcoded values for **different tests**

Cons:

- May be **error-prone** depending on source of testvectors
- More scalable, but still **limited** by reading a file
 - Might have many more combinational paths to test than will fit in memory

Automatic Testbench

Golden Models

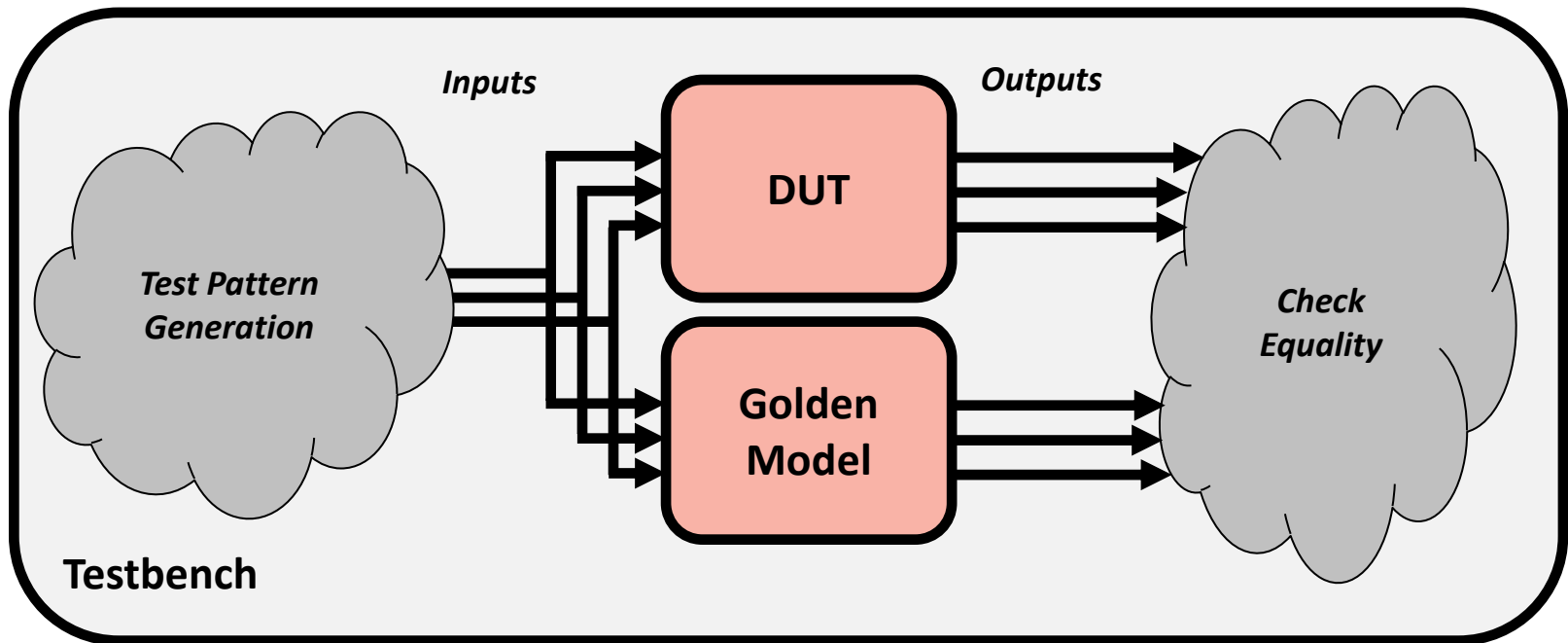
- A **golden model** represents the ideal circuit behavior
 - Must be developed, and might be **difficult** to write
 - Can be done in C, Perl, Python, Matlab or even in Verilog
- For our example circuit:

```
module golden_model(input  a, b, c,  
                    output y);  
    assign y = ~b & ~c | a & ~b; // high-level  
    abstraction  
endmodule
```

- **Simpler** than our earlier gate-level description
 - Golden model is usually **easier to design and understand**
 - Golden model is much **easier to verify**

Automatic Testbench

- The DUT **output** is compared against the **golden model**



- **Challenge:** need to **generate inputs** to the designs
 - Sequential values to cover the entire input space?
 - Random values?

Automatic Testbench: Code

```
module testbench1();  
    ... // variable declarations, clock, etc.  
    // instantiate device under test  
    sillyfunction dut (a, b, c, y_dut);  
    golden_model gold (a, b, c, y_gold);  
  
    // instantiate test pattern generator  
    test_pattern_generator tgen (a, b, c, clk);  
  
    // check if y_dut is ever not equal to y_gold  
    always @(negedge clk)  
    begin  
        if(y_dut !== y_gold)  
            $display(...)  
    end  
endmodule
```

Automatic Testbench

Pros:

- Output checking is **fully automated**
- Could even compare **timing** using a **golden timing model**
- **Highly scalable** to as much simulation time as is feasible
 - Leads to **high coverage** of the input space
- Better **separation of roles**
 - Separate designers can work on the DUT and the golden model
 - DUT testing engineer can focus on **important test cases** instead of output checking

Cons:

- Creating a correct golden model may be (very) **difficult**
- Coming up with **good testing inputs** may be **difficult**

However, Even with Automatic Testing...

- How long would it take to test a **32-bit adder**?
 - In such an adder there are **64** inputs = 2^{64} possible inputs
 - If you test **one input in 1ns**, you can test 10^9 inputs per second
 - or 8.64×10^{14} inputs per day
 - or 3.15×10^{17} inputs per year
 - we would still need **58.5 years** to test all possibilities
- Brute force testing is **not feasible** for most circuits!
 - Need to prune the overall testing space
 - E.g., formal verification methods, choosing 'important cases'
- **Verification is a hard problem**

Part 5: Timing Verification

Timing Verification Approaches

- High-level simulation (e.g., C, Verilog)
- Circuit-level timing verification

High-level Timing Verification

- High-level simulation (e.g., C, Verilog)
 - Can **model timing** using “#x” statements in the DUT
 - Useful for hierarchical modeling
 - Insert delays in FF's, basic gates, memories, etc.
 - High level design will have some notion of timing
 - Usually **not as accurate** as real circuit timing

Regular Delay Control

- Regular delays defer the execution of the entire assignment
- A non-zero delay is specified to the left of a procedural assignment

```
//define parameters
parameter latency = 20;
parameter delta = 2;
//define register variables
reg x, y, z, p, q;

initial
begin
    x = 0; // no delay control
    #10 y = 1; // delay control with a number. Delay execution of
               // y = 1 by 10 units

    #latency z = 0; // Delay control with identifier. Delay of 20
units
    #(latency + delta) p = 1; // Delay control with expression

    #y x = x + 1; // Delay control with identifier. Take value of
y.

    #(4:5:6) q = 0; // Minimum, typical and maximum delay values.
                   //Discussed in gate-level modeling chapter.
end
```

Delay-based Timing: Intra-assignment Delay Control (cont'd)

```
//define register variables
reg x, y, z;

//intra assignment delays
initial
begin
    x = 0; z = 0;
    y = #5 x + z; //Take value of x and z at the time=0, evaluate
                  //x + z and then wait 5 time units to assign value
                  //to y.

end

//Equivalent method with temporary variables and regular delay control
initial
begin
    x = 0; z = 0;
    temp_xz = x + z;
    #5 y = temp_xz; //Take value of x + z at the current time and
                  //store it in a temporary variable. Even though x and
z
                  //might change between 0 and 5,
                  //the value assigned to y at time 5 is unaffected.

end
```

Delay-based Timing: Zero Delay Control (cont'd)

- Zero delay control
 - Ensure that a statement is executed **last**, after all other statements in that simulation time are executed
 - Eliminate race conditions

```
initial
begin
    x = 0;
    y = 0;
end
```

```
initial
begin
    #0 x = 1; //zero delay control
    #0 y = 1;
end
```

Circuit-level Timing Verification Approaches

- Circuit-level timing verification
 - Need to first **synthesize** your design to actual circuits
 - No general approach (Very **design flow specific**)
 - Your FPGA/ASIC/etc. technology has **special tool(s)** for this
 - E.g., Xilinx Vivado (what you're using in lab)
 - E.g., Synopsys/Cadence Tools (for VLSI design)

The Good News

- Tools will try to meet timing for you!
 - Setup times, hold times
 - Clock skews
 - ...
- They usually provide a **'timing report'** or **'timing summary'**
 - **Worst-case** delay paths
 - Maximum operation **frequency**
 - Any timing **errors** that were found

The Bad News

- The **tool can fail** to find a solution
 - Desired clock frequency is too **aggressive**
 - Can result in **setup time violation** on a particularly long path
 - **Too much logic** on clock paths
 - Introduces excessive **clock skew**
 - Timing issues with asynchronous logic
- The tool will provide (hopefully) **helpful errors**
 - Reports will contain paths that failed to meet timing
 - Gives a place from where to start debugging
- **Q:** How can we **fix timing errors**?

Meeting Timing Constraints

- Unfortunately, this is often a **manual, iterative** process
 - Meeting strict timing constraints (e.g., high performance designs) can be **tedious**
- Can try **synthesis/place-and-route** with different options
 - Different **random seeds**
 - Manually provided **hints** for place-and-route
- Can **manually optimize** the reported **problem paths**
 - Simplify **complicated logic**
 - Split up **long combinational logic paths**
 - Recall: fix hold time violations by adding **more** logic!

Meeting Timing Constraints: Principles

- Let's go back to the fundamentals
 - Clock cycle time is determined by **the maximum logic delay** we can accommodate without violating timing constraints
- Good design principles
 - **Critical path design**: Minimize the maximum logic delay
 - Maximizes performance
 - **Balanced design**: Balance maximum logic delays across different parts of a system (i.e., between different pairs of flip flops)
 - No bottlenecks + minimizes wasted time
 - **Bread and butter design**: Optimize for the common case, but make sure non-common-cases do not overwhelm the design
 - Maximizes performance for realistic cases

Lecture Summary

- Timing in **combinational circuits**
 - Propagation delay and contamination delay
 - Glitches
- Timing in **sequential circuits**
 - Setup time and hold time
 - Determining how fast a circuit can operate
- **Circuit Verification**
 - How to make sure a circuit works correctly
 - Functional verification
 - Timing verification

Thank You

