

UMBC

AN HONORS UNIVERSITY IN MARYLAND

Department of Computer Science and Electrical Engineering

CMPE 415

Verilog Events Timing and Testbenches

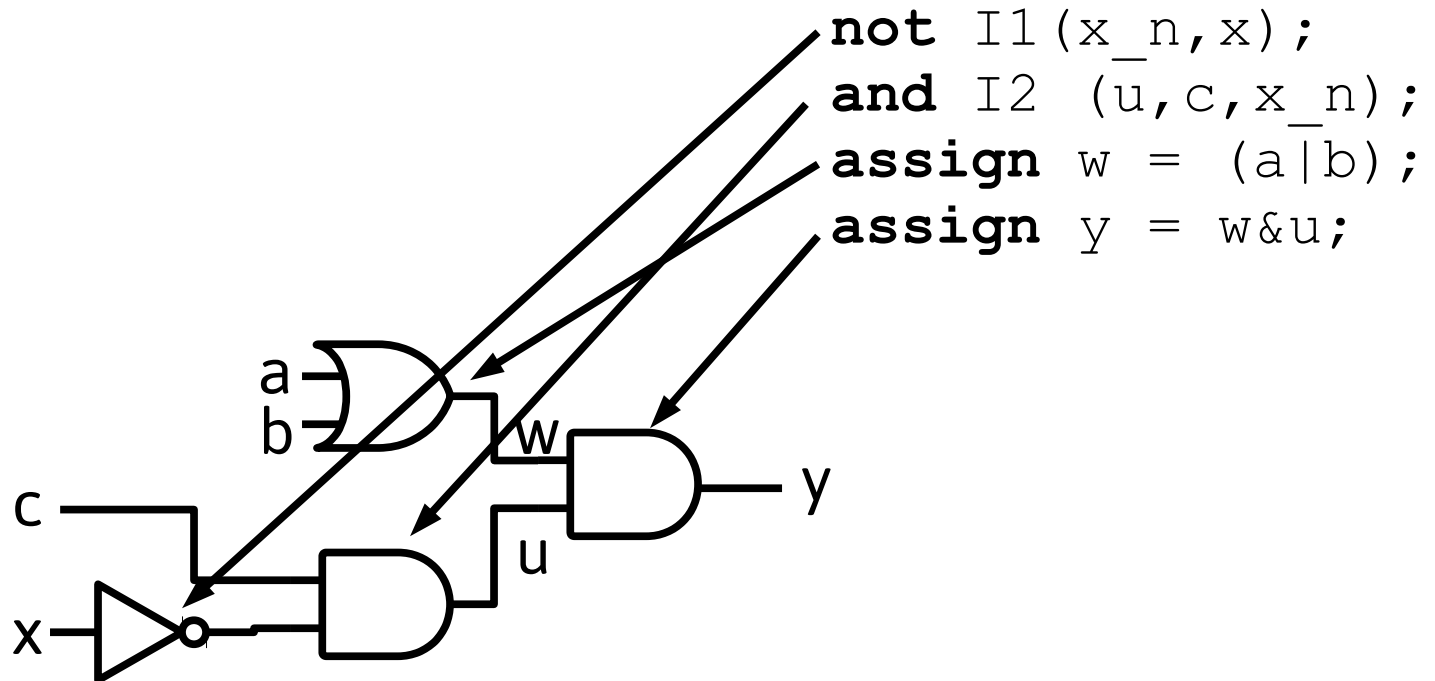
Prof. Ryan Robucci

An Event Driven Language also used for Synthesis

- We emphasize use of Verilog as a hardware description language for synthesis, but it is a general event-driven simulation language
- Verilog is event driven, events are triggered to cause **evaluation** events to be **queued** which cause **updates** to be queued which may in turn serve as triggers for other events to be queued.
- Events are entered into a priority queue for processing. Event elements in the priority queue are removed and processed according to two rules
 - earliest time first
 - then first come first serve stack behavior
- Nondeterminism of concurrent statements: By default, individual assignment statements and blocks in a module are considered to be concurrent. This means that their evaluation may be queued in any order if triggered.
- Certain coding guidelines constrain the use of the language to ensure deterministic behavior in simulation and are known to map to hardware with the same behavior
 - Most of these issues relate to concurrency

Verilog Language

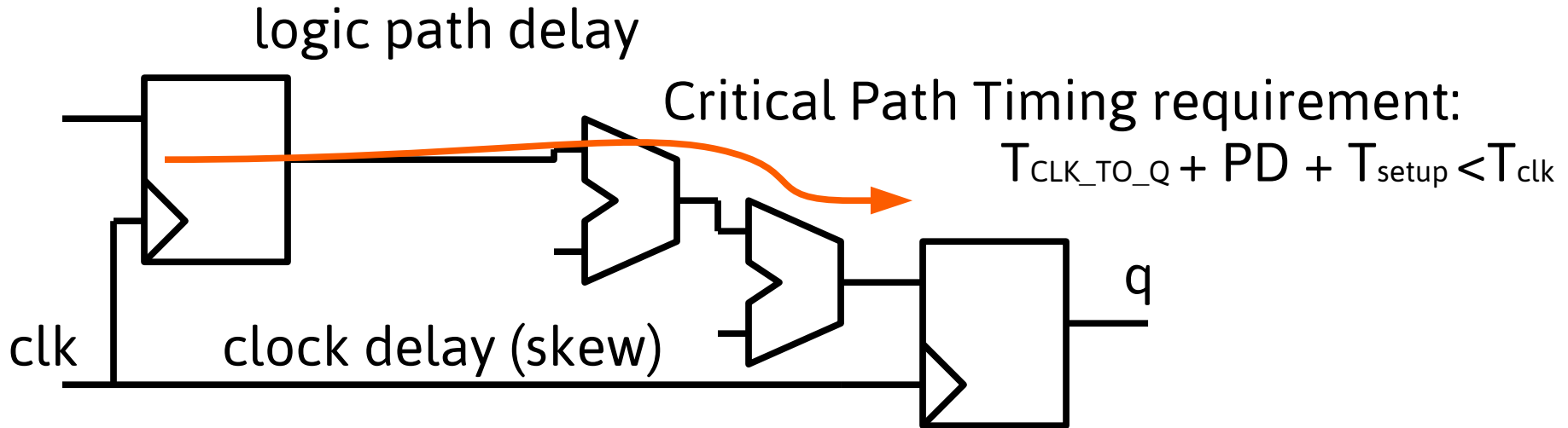
- Being a hardware description and modeling language, Verilog supports modeling of concurrency as well as time.



- Inclass: discuss flow of value propagation and events
 $\{a,b,c,x\}=\{1,1,1,1\} \rightarrow \{a,b,c,x\}=\{1,1,1,0\}$

Timing

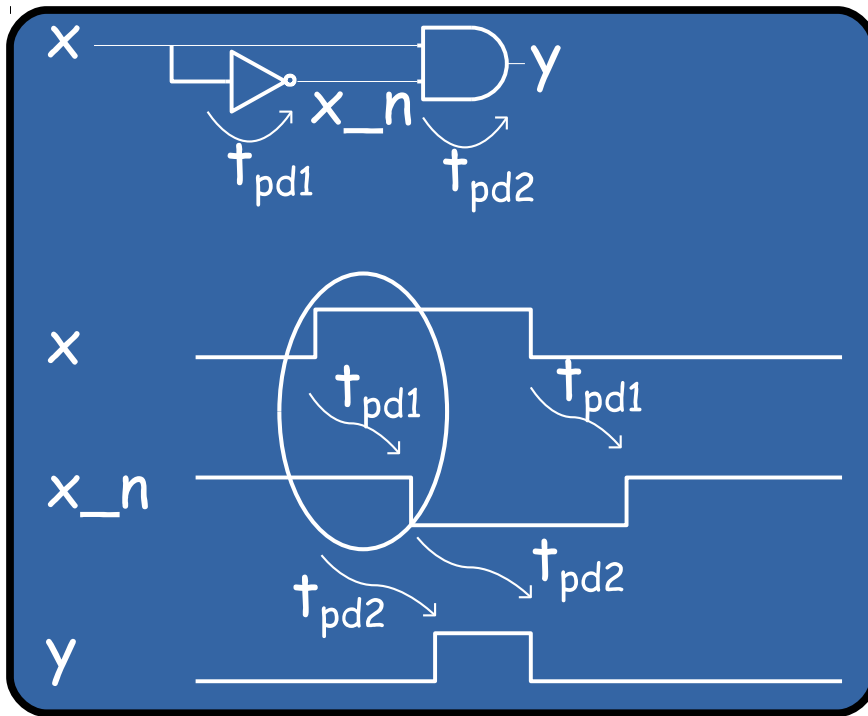
- Timing and delay are intrinsic aspects of physical circuits.
- Delay in a logic path might be a parasitic effect, and must be analyzed to ensure the circuit can operate fast enough.



- Delay in the logic path also helps prevent race conditions if the clk arrival at the downpath register is slightly delayed

Timing

- Other times, delay is fundamental to how a circuit works. In the circuit below delay is necessary for the circuit to generate a pulse.



Structural Example:

```
module pulser(y, x);  
  input    x;  
  output   y;  
  wire     x_n;  
  
  and      #5 (y, x, x_n)  
  not      #3 (x_n, x);  
endmodule
```

delays

Concurrent Operation

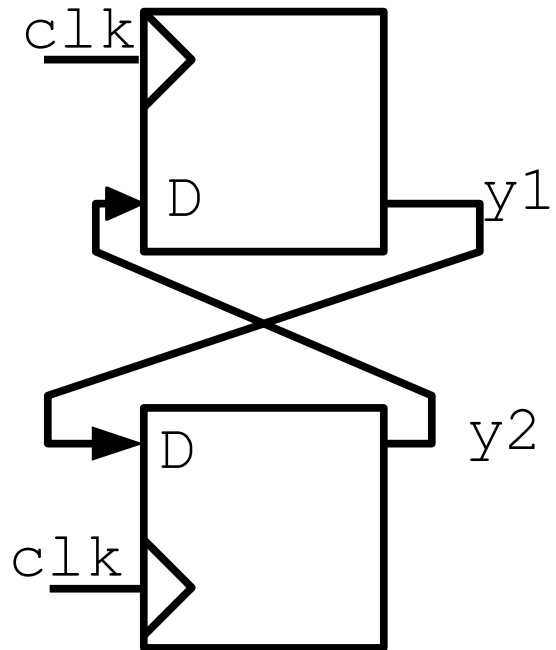
Concurrency, is another timing property of an HDL

This code attempts to model a swap of y1 and y2, as in

y1 = old y2

y2 = old y1

```
module fbosc2 (y1, y2, clk, rst);  
  output y1, y2;  
  input clk, rst;  
  reg y1, y2;  
  
  always @(posedge clk or posedge rst)  
    if (rst) y1 <= 0; // reset  
    else y1 <= y2;  
  
  always @(posedge clk or posedge rst)  
    if (rst) y2 <= 1; // preset  
    else y2 <= y1;  
  
endmodule
```



This is a case where handling of concurrency, is critical as two interdependent updates are at once

The stratified event queue

- The IEEE standard identifies 5 regions for events to live:
 - Active events : in queue ready for execution in current simulation time. The Nondeterminism property states that the order that they are removed is not specified. One effect is that concurrent statements and blocks are not guaranteed execution in any order. Furthermore, blocks of behavioral statements may be returned to the active queue at any point even if only partially completed. This allows interleaving of process execution but also unpredictable behavior for concurrent processes with miscoded communication paths between them.
 - Inactive events: to be processed at the end of the current simulation time after any active events in the queue. An explicit zero delay (#0) is a way to force an event to be transferred to the inactive event queue.
 - Nonblocking assign update events: to be processed at the end of the current simulation time after active and inactive events in the queue. They are processed in the order they are added to the queue.
 - Monitor events: to be processed if no active, inactive, or nonblocking assign update events are in the queue. These events do not create any other events. \$monitor and \$strobe system tasks generate monitor events.
 - Future events: active, inactive and nonblocking assign update events scheduled with a future time

IEEE Specification of Event Handling

```
while (there are events) {
  if (no active events) {
    if (there are inactive events) {
      activate all inactive events;
    } else if (there are nonblocking assign update events) {
      activate all nonblocking assign update events;
    } else if (there are monitor events) {
      activate all monitor events;
    } else {
      advance T to the next event time;
      activate all inactive events for time T;
    }
  }
}

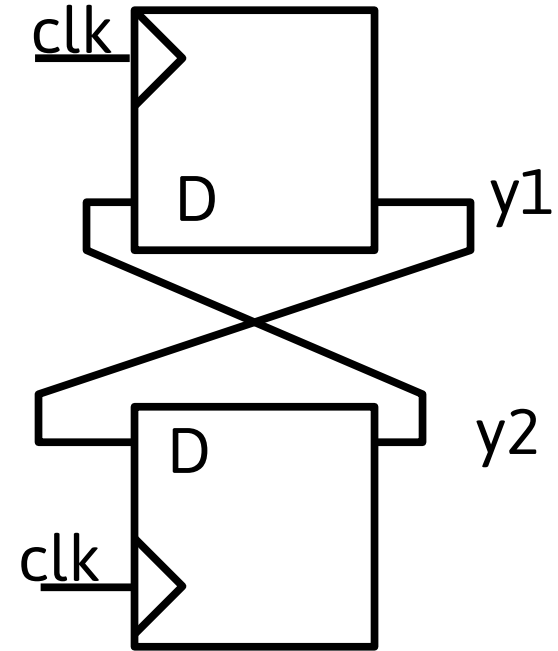
E = any active event;
if (E is an update event) {
  update the modified object;
  add evaluation events for sensitive processes to event queue;
} else { /* shall be an evaluation event */
  evaluate the process;
  add update events to the event queue;
}
}
```


Evaluation and Update Events

- Most events can be describe as evaluation or update events
 - **Evaluation events** involve processing or execution. These events can enable other events to enter the active queue, often it enters value assignment events into the queue.
 - **Update events** involve making a value assignment to a variable. These in turn may implicitly enable evaluation events that are sensitive to the changed variable

Bad Parallel Blocks

```
module fbosc1 (y1, y2, clk, rst);  
  output y1, y2;  
  input clk, rst;  
  reg y1, y2;  
  
  always @(posedge clk or posedge rst)  
    if (rst) y1 = 0; // reset  
    else y1 = y2;  
  
  always @(posedge clk or posedge rst)  
    if (rst) y2 = 1; // preset  
    else y2 = y1;  
endmodule
```



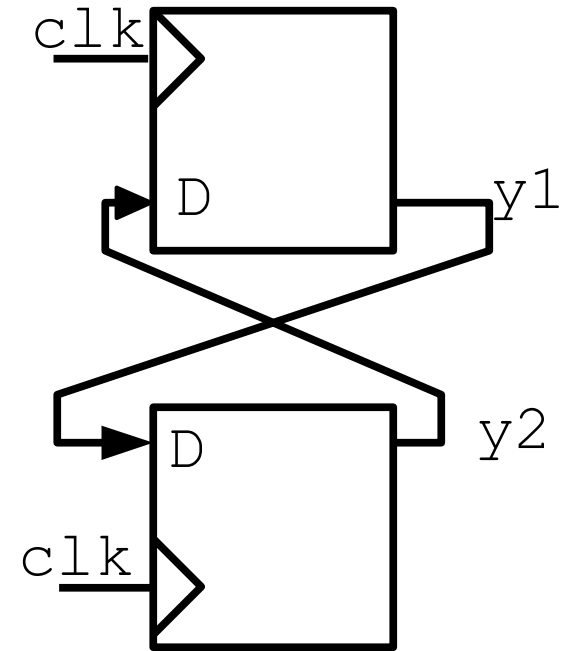
This code attempts to model a swap of y1 and y2

Timing of execution of parallel always blocks is not guaranteed in simulation – though synthesis will probably work since synthesis approaches each always blocks somewhat independently at first

Good Parallel Blocks

Will not only synthesize correctly, but also simulate correctly:

```
module fbosc2 (y1, y2, clk, rst);  
  output y1, y2;  
  input clk, rst;  
  reg y1, y2;  
  
  always @(posedge clk or posedge rst)  
    if (rst) y1 <= 0; // reset  
    else y1 <= y2;  
  
  always @(posedge clk or posedge rst)  
    if (rst) y2 <= 1; // preset  
    else y2 <= y1;  
endmodule
```



Continuous Assignment

- Continuous assignment statements “enable” (a.k.a. schedule or trigger) an evaluation event when any source element of the RHS expression changes. Upon any expression's resulting value change, an update event for the LHS is added to the active queue.

`assign {cout,b} = d+a;`

Left-hand-side (LHS)
expression

Right-hand-side (RHS)
expression

- An initial evaluation always occurs at time zero to propagate constant values.
In the the following, an initial evaluation and assignment will occur, but no other update to a will happen in the simulation.

`assign a=1;`

Procedural continuous assignment

- Consider this topic not covered until future notice
 - Keywords **assign**, **force**, **deassign**, **release** inside procedural code

Blocking and Nonblocking Procedural Assignment

- Blocking assignments cause an immediate evaluation of the right hand side. If delay is provide, then the assignment to the left-hand-expression is scheduled for a future time unless the delay is 0 and in which case the assignment event is entered into the inactive queue. If no delay is specified the statement performs the assignment and returns immediately. Once the assignment is performed, any change will enable any events dependent on the value.

$$a = \underbrace{\#D}_{\text{assignment delay}} b + c;$$

- Nonblocking assignments schedule an evaluation and assignment using the values of the inputs at the time of the scheduling. The event is placed in the nonblocking assignment update event queue for the current time or, if a delay is provided, a future time.

$a \leq \underbrace{\#D}_{\text{assignment delay}} b + c;$

Procedural Blocking and Procedural Assignment Delay

In the delay before evaluation form (**statement delay** or **evaluation delay**), the delay essentially “sleeps” the process until another time before the following code is evaluated.

```
#D a = b + c; //delay before evaluation
```

```
#D a <= b + c; //delay before evaluation
```

Code after these statements in a begin...end block will not execute until after the delay, nor will the block pass control

Assignment delay schedules an assignment for a future time, though the RHS expression is evaluated using the values at the time of the scheduling.

```
a = #D b + c; delay before assignment, execution in begin..end block  
does not progress until assignment event is handled, nor  
will the block will pass control
```

```
a <= #D b + c; delay before assignment: RHS evaluation is followed by  
scheduling a future nonblocking assignment event, execution  
within a begin...end block progresses and control may be  
passed out of the block
```


Code ex.

```
initial #0 $display($time,"  a  b  y  z");

initial begin
    $monitor($time,u8a,u8b,u8y,u8z) ;
    u8a=0; u8b=0;
    #5;
    #5 u8a = 1;
    #5 u8b = 1;
end

always@* begin
    $display($time,u8a,u8b,"          enter nonblocking");
    u8y <= #20 u8a+u8b;
    $display($time,u8a,u8b,"          leave nonblocking");
end

always@* begin
    $display($time,u8a,u8b,"          enter blocking");
    u8z = #20 u8a+u8b;
    $display($time,u8a,u8b,"          leave blocking");
end

initial #100 $finish;
```

Result

Simulator is doing circuit initialization process.

0	0	0		enter nonblocking
---	---	---	--	-------------------

0	0	0		leave nonblocking
---	---	---	--	-------------------

0	0	0		enter blocking
---	---	---	--	----------------

Finished circuit initialization process.

0	a	b	y	z
---	---	---	---	---

0	0	0	x	x
---	---	---	---	---

10	1	0		enter nonblocking
----	---	---	--	-------------------

10	1	0		leave nonblocking
----	---	---	--	-------------------

10	1	0	x	x
----	---	---	---	---

15	1	1		enter nonblocking
----	---	---	--	-------------------

15	1	1		leave nonblocking
----	---	---	--	-------------------

15	1	1	x	x
----	---	---	---	---

20	1	1		leave blocking
----	---	---	--	----------------

20	1	1	0	0
----	---	---	---	---

30	1	1	1	0
----	---	---	---	---

35	1	1	2	0
----	---	---	---	---

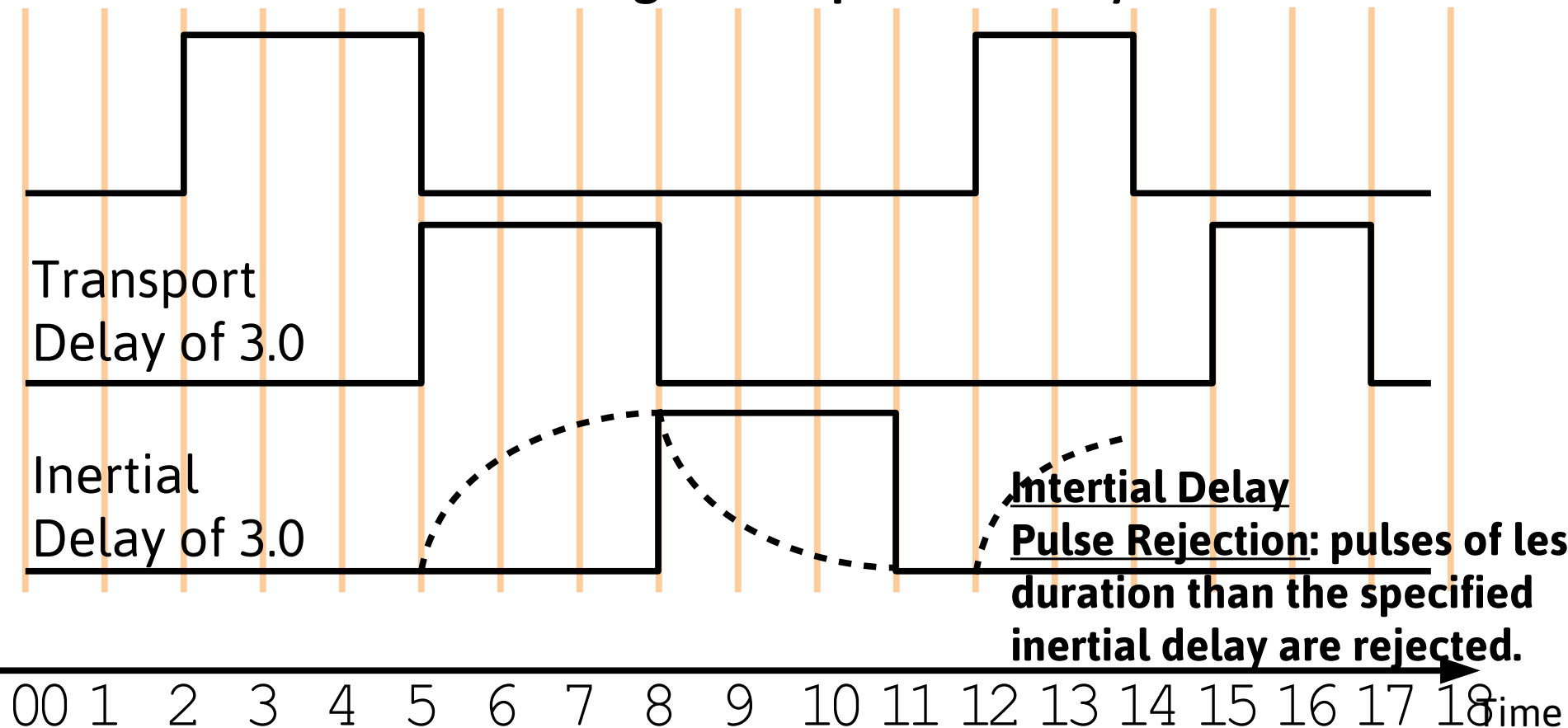
Stopped at time : 100 ns : File ".../delay_tests.v" Line 56

Delayed Blocking assignment prevented multiple calls.

Inertial and Transport Delay

Transport Delay maintains all transitions and delays them by a specified amount.

Inertial Delay also delays transitions, but it only maintains transitions to states that are held for as long as the specified delay.



Inertial Delay on Wires

The following are all examples of inertial delay on a wire.

delay in the continuous assignment

```
wire x_n;  
assign #5 x_n = ~n;
```

delay in the implicit assignment

```
wire #5 x_n = ~n;
```

delay in the wire declaration (added to any assignment delay)

```
wire #5 x;  
assign x = ~x;
```

In this example, the inertial delay to **a** is **4** and for **b** is **3**.

```
wire #3 a;  
wire #2 b;  
assign #1 {a,b} = {x,x};
```

Inertial Delay using Primitives

Inertial delay may be provide to primitives using the simplist form of the delay operator.

```
wire y;  
and #3 I1 (y, a, b) ;
```

In this next example, the inertial delay to **y** is **4** since delay provide in the declaration is added.

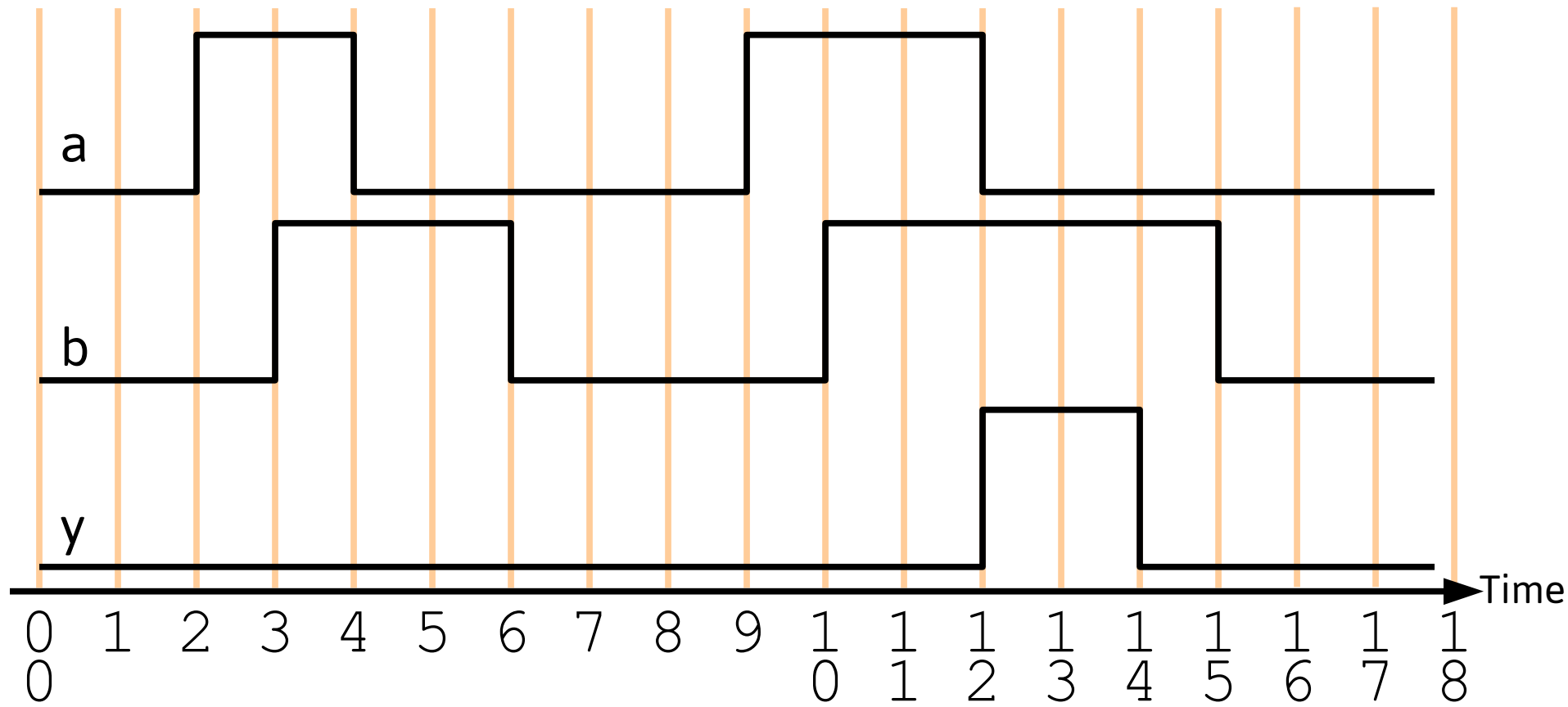
```
wire #1 y;  
and #3 I1 (y, a, b) ;
```

Pulse Rejection is Based on the Result

The pulse rejection behavior is **based on the result, not the inputs**.
The result of an expression to be assigned must maintain a new value for the specified time.

```
wire y;  
assign #2 y = a&b;
```

```
wire y;  
and #2 I1(y, a, b);
```

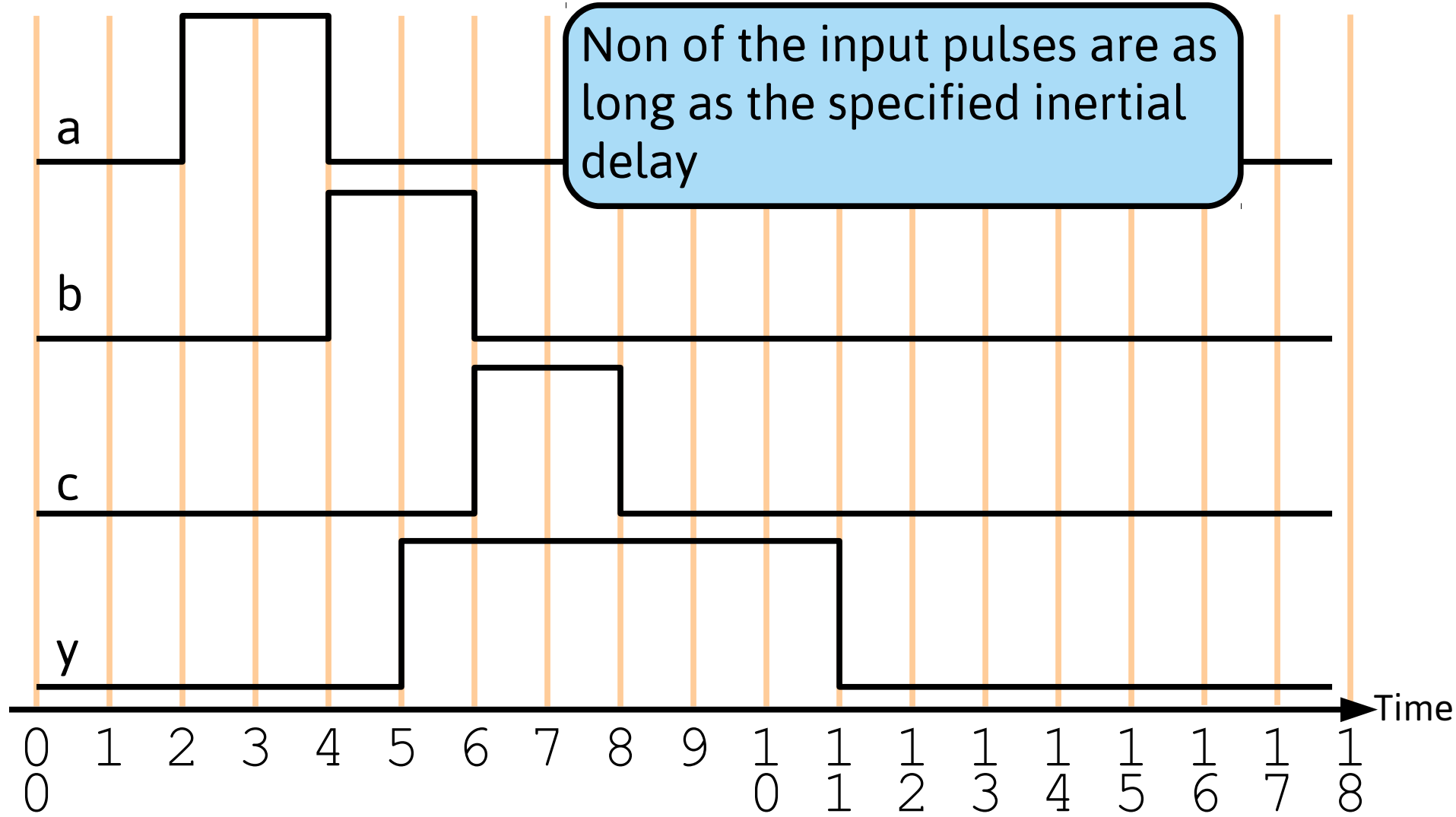


```
wire y;
```

```
assign #3 y = |{a,b,c};
```

```
wire y;
```

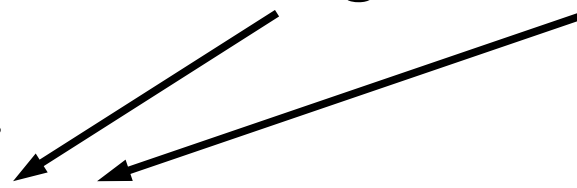
```
or #3 I1(y,a,b,c);
```



Rising and Falling Delay

Separate delays may be provided for **rising** and **falling** output edges

```
wire y;  
and # (3, 2) I1 (y, a, b) ;
```



Rising Edges:

0 → 1, X→1 Z→1

Falling Edges:

1 → 0, X→0, Z→0

Other output edges (e.g. X→Z , 1→Z) assume the minimum value between the of the rising and falling delays (2 in the example above)

Inertial Delay for reg

For a **reg** signal, to model inertial delay it is recommend to create a delayed wire copy. Direct modeling of inertial delay is difficult otherwise.

```
wire c;  
reg c_nodelay  
assign #5 c = c_nodelay;  
always @ (b,d)  
    c_nodelay = ~b|d;  
end
```

Transport Delay

Transport Delay can be modeled using a procedural block with **assignment delay**.

Procedural block demonstrating **clk-to-q delay**:

```
parameter TC2Q;  
reg q;  
always @ (posedge clk)  
    q <= #TC2Q ~b;
```

Upon the clock edge, b is immediately sampled and the result of ~b is scheduled for future nonblocking assignment to q

The following is not the same. It would evaluate b after the delay. This samples b later than the clk with assignment immediately thereafter opposed to “sample and evaluate now and assign later”.

```
parameter TC2Q;  
reg q;  
always @ (posedge clk)  
    #TC2Q q <= ~b;
```

models some delay between the the clock edge and evaluation

Sequential Logic

```
parameter TC2Q;  
reg q;  
always @(posedge clk)  
    q <= #TC2Q ~b;
```

- Evaluate now, assign later
- Models **Clk-to-Q delay**

```
parameter TC2Q;  
reg q;  
always @(posedge clk)  
    q = #TC2Q ~b;
```

- Evaluate now, assign later
- Bad: Nondeterminism of assignment at later time

```
parameter TC2Q;  
reg q;  
always @(posedge clk)  
    #TC2Q q <= ~b;
```

- Evaluate after delay then assign
- Models a delay in the clock signal. Not recommended, better to create and use a delayed clock using other delay methods

```
parameter TC2Q;  
reg q;  
always @(posedge clk)  
    #TC2Q q = ~b;
```

- Evaluate after delay then assign
- Bad: Nondeterminism of assignment at later time

Combinatorial Logic

```
parameter D;  
reg y;  
always @ (a,b)  
    y <= #D a&b;
```

```
always @ (y)  
    y_delayed <= #D y;
```

- Evaluate now, assign later
- Can be use to model transport delay for combinatorial logic or a delayed signal

```
parameter D;  
reg y;  
always @ (a,b)  
    #D y <= a&b;
```

- Evaluate later and assign
- Bad: Blocks reevaluation

```
parameter D;  
reg y;  
always @ (a,b)  
    y = #D a&b;
```

- Evaluate now, assign later
- Bad: Blocks reevaluation

```
parameter D;  
reg y;  
always @ (a,b)  
    #D y = a&b;
```

- Evaluate later and assign
- Bad: Blocks reevaluation

Simulation Use of RHS non-blocking assignment delay

- RHS non-blocking delays may be conveniently used in testbenches to schedule several delayed assignments (not something I use)

Ex 1:

```
initial begin
  a <= 0;
  a <= #10 255;
  a <= #20 22;
  a <= #30 10;
  b <= 'bx;
  b <= #1 1193;
  b <= #10 122;
  c <= #10 92;
  c <= #15 93;
end
```

Ex 2:

```
initial begin
  a = 0;
  b = 0;
  c = 0;
  wait (rst==0) ; delays relative
  a <= #10 255; to rst signal
  a <= #20 22; change
  a <= #30 10;
  b <= #1 1193;
  b <= #10 122;
  c <= #10 92;
  c <= #15 93;
end
```

Procedural Timing Controls

- Delay control: delay between encountering the expression and when it executes.
 - Introduced by simple #
- Event control: delay until event
 - Explicit Events are named events that allow triggering from other procedures
 - A named event may be declared using the keyword **event**

```
event triggerName;
```
 - The event control operator @ can be used to hold procedural code execution

```
@ (triggerName) ;
```
 - operator -> triggers the event

```
-> triggerName;
```
 - Implicit events are responses to changes in variables
 - The event control operator @ can be provided a sensitivity list with variables and an optional event selectivity using keywords **posedge** and **negedge**

Event Control Operator

The event control operator may be provided with variable multiple variables using comma separation. (older syntax is to us **or**)

@(a,b,c)

@(a or b or c)

Negedge and posedge restrict sensitivity to the following transitions

@(**posedge** clk or **negedge** edge en)

Negedge: $1 \rightarrow 0$

Posedge: $0 \rightarrow 1$

$1 \rightarrow x \text{ or } z$

$x \text{ or } z \rightarrow 1$

$x \text{ or } z \rightarrow 0$

$0 \rightarrow x \text{ or } z$

When posedge and negedge modify a multi-bit operand, only the lsb is used to detect the edge

Level-sensitive event control using wait

- The wait statement suspends execution until a condition is true.
- It can be considered as a **level-sensitive** control since it doesn't wait for a transition edge.

@ (**posedge** clk) if clk is already true, wait for next rising edge

wait (clk); if clk is already true, produce without delay

Example to change data on the falling clock edge that follows a variable exceeding the value 10

```
wait (a>10) ;
```

```
@ (negedge clk) ;
```

```
data=data+1;
```


Repeat

Any timing control may be modified so as to be repeated/multiplied, by using the keyword **repeat**

```
repeat (count) @ (event expression)
```

If count is positive, repeat the timing control that number of time. If count is equal or less than 0, skip

Wait for 10 clk rising edges before proceeding execution:

```
repeat (10) @ (posedge clk);
```

Delay an assignment by 5 clk edges

```
a <= repeat (5) @ (posedge clk) data;
```

initial and always

The **initial** construct is used to denote code to be executed once at the beginning of the simulation.

The **always** construct causes code to run repeatedly in an infinite loop. It is only useful with a delay or control construct, otherwise will create a zero delay infinite loop that can block time progression in simulation

always x=a&b; This would run infinitely

This prints endlessly in the beginning of the simulation:

```
always begin  
    $display("hello %0t");  
end
```

```
0 : hello  
0 : hello  
0 : hello  
0 : hello  
0 : hello ...
```

Sequential and Parallel Blocks

- Sequential procedural blocks are wrapped with begin...end
- Parallel procedural blocks are wrapped with fork...join
 - Statement Delay control is relative to entering the block
 - Control passes out of the block when the last statement, which may be delayed in time, is completed

Here is a mix, waiting for two events to perform assignment:

```

                begin
sequential  ——— fork
                @event1; } parallel
                @event2; }
                join
                areg = breg;
                end
```

Don't confuse sequential procedural block to mean coding a procedural block for sequential hardware

Sim Using Test Bench: DUT

```
module Nand_Latch_1 (q,qbar,preset,clear);  
    output q, qbar;  
    input preset,clear;  
    Nand1 #1 G1 (q,preset,bar);  
    G2 (qbar,clear,q)  
endmodule
```

Sim Using Test Bench:Test Bench

```
module test_Nand_Latch_1; // Design Unit Testbench
    reg        preset, clear;
    wire        q, qbar;
    Nand_Latch_1 M1 (q, qbar, preset, clear); // Instantiate DUT
    initial // Create DUTB response monitor
        begin
            $monitor ($time,
                "preset = %b clear = %b q = %b 1 qbar = %b",
                preset, clear, q, qbar);
        end
    initial
    begin // Create DUTB stimulus generator
        #10 preset = 0; clear = 1;
        #10 preset = 1; $stop; // Enter, to proceed
        #10 clear = 0;
        #10 clear = 1;
        #10 preset = 0;
    end
    initial
        #60 $finish ; // Stop watch
    end
endmodule
```

Testbench Results

0	preset	=	x	clear	=	x	q	=	x	qbar	=	x
10	preset	=	0	clear	=	1	q	=	x	qbar	=	x
11	preset	=	0	clear	=	1	q	=	1	qbar	=	x
12	preset	=	0	clear	=	1	q	=	1	qbar	=	0
20	preset	=	1	clear	=	1	q	=	1	qbar	=	0
30	preset	=	1	clear	=	0	q	=	1	qbar	=	0
31	preset	=	1	clear	=	0	q	=	1	qbar	=	1
32	preset	=	1	clear	=	0	q	=	0	qbar	=	1
40	preset	=	1	clear	=	1	q	=	0	qbar	=	1

Timescale Directive

This directive sets the time for delays

reference_time_unit: what the delays should be interpreted as

precision: and what the precision should be for. The precision affects rounding.

```
`timescale  
<reference_time_unit>/<time_precision>
```

```
example : `timescale 1ns/1ps
```

```
    then
```

```
        #17.0402 is 17.040ns
```

Generating a clock in a testbench



```
initial begin
    clk = 0;
end
always begin
    #5 clk = 0;
    #5 clk = 1;
end
```

```
initial begin
    clk = 0;
end
always begin
    #5 clk =
    ~clk;
end
50% duty
cycle
```

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

The procedural construct **forever** can be used to create an infinite loop.

Generating an irregular clock in a testbench



```
initial begin
  clk = 0;
  forever begin
    repeat (16) begin
      #5 clk = ~clk;
    end
    #20;
  end
end
```

The procedural construct **repeat** can be used to create a limited loop.

Testing all combinatorial inputs

```
reg a,b,c,d;  
initial begin  
    a = 0; b = 0;  
    c = 0; d = 0;  
end
```

```
always begin  
    #5 a = ~a;  
end
```

```
always begin  
    #10 b = ~b;  
end
```

```
always begin  
    #20 c = ~c;  
end
```

```
always begin  
    #40 d = ~d;  
end
```

```
wire a,b,c,d;  
reg [3:0] count;  
initial begin  
    count = 0;  
end  
  
assign  
{a,b,c,d}=count;  
always begin  
    #5 count = count+1;  
end
```

Testing a memory

```
wire data_out;
reg write;
reg [3:0] addr;
reg [7:0] memory_buffer [0:15]; // 16 entries 8 bit #'s
reg [7:0] data_in;
integer i; integer file;
my_mem I0 (data_out,data_int,addr,write);
initial begin
    $readmemh("memory_hex_in.txt", memory_buff); //init memory_buff from
file
    file = $fopen("memory_hex_out.txt"); //open a file for writing
results
    #5 write = 0; addr =0;
    for (i=0; i<16; i++) begin //write to mem
        #5 write = 0; data_in= memory_buff[i]; addr= i;
        #5 write = 1;
    end
    #10 write = 0;

    //reading and writing to file
    for (i=0; i<16; i++) begin
        #5 addr =i;
        $fstrobe(file,"%2H",data_out);
    end
    $fclose(file);
end
end
```

The procedural construct **for** can be used to create simulation loops.

Simulation vs Synthesis Loops

- In the previous slides you were taught **for**, **forever** and **repeat** in the context of simulation code such as for a testbench.
 - At this point you are expected to know these
 - **However you should NOT start using these for synthesis.**
Rules for using loops in synthesis will be taught at a later time.