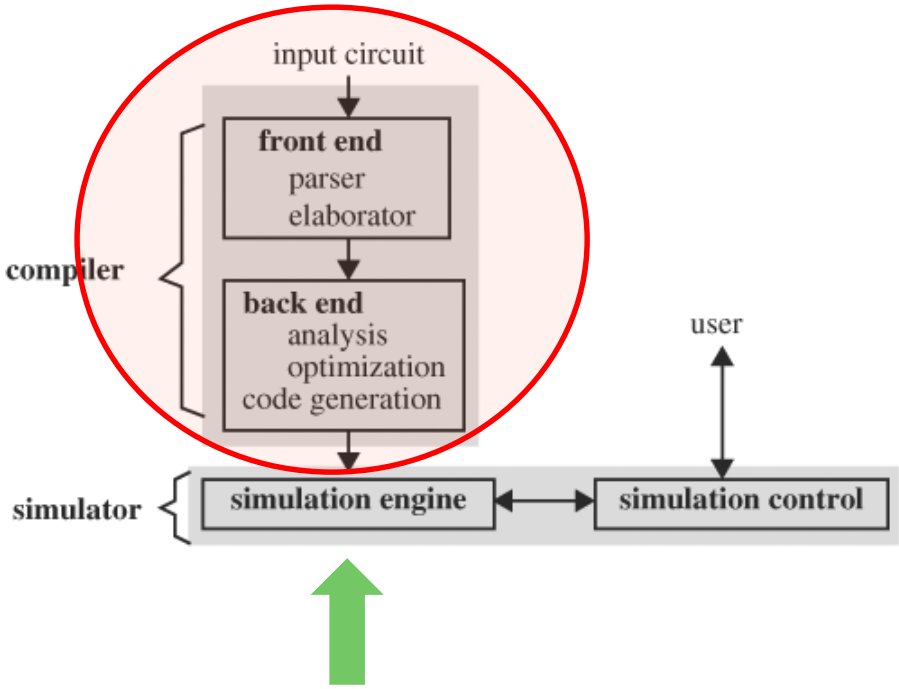
SystemVerilog Simulation

Simulator

- A simulator consists of 3 major components:
  - a front end, a back end, and a simulation engine/control.
- The front end is standard for most simulators and is a function only of the input language.
- The back end performs analysis, optimization, and generation of code to simulate the input circuit.
  - The main contributor to a simulator's speed.
- The front end and the back end form the compiler portion of a simulation system.



- The simulation engine takes in the generated code and computes the behavior accordingly.

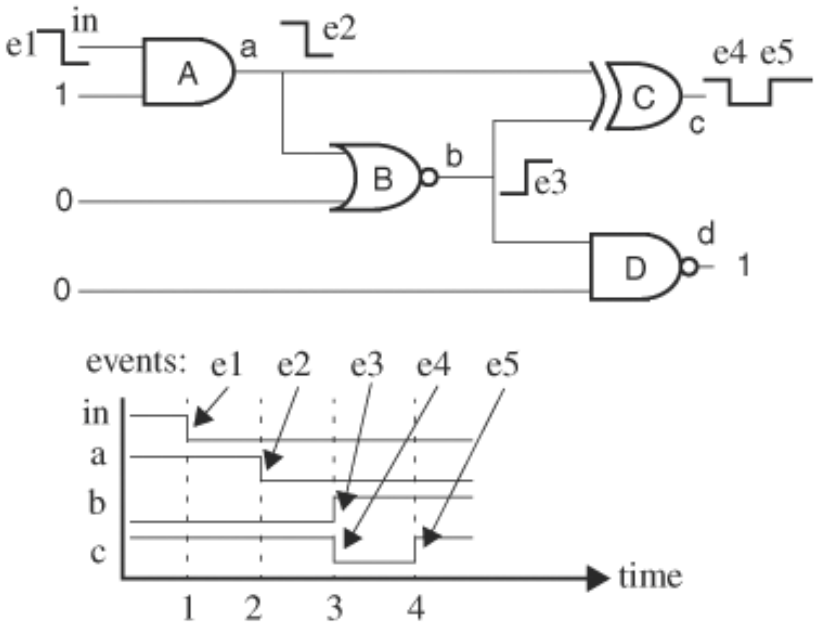
SystemVerilog: Concurrent Language

- SystemVerilog is a language for describing
  - digital systems (a parallel system).
- To verify that a model is correct, a simulator is used to understand the model.
- RTL synthesis attempts to generate low level hardware that behaves in the same way as the original code.
- The interpretation of SV structures for synthesis is based on the simulation model.

SystemVerilog: Concurrent Language

- SV is not C
  - Statements are not necessarily executed in a sequential manner
  - SV is Event driven
- SV is a concurrent language
  - Models hardware
  - Provides for the specification of concurrent activities
  - Allows for reasonably realistic timing specifications
  - Models multiple concurrent events (an output or state change)
  - An event in one element can cause activity in another
  - Execution of an element can be delayed

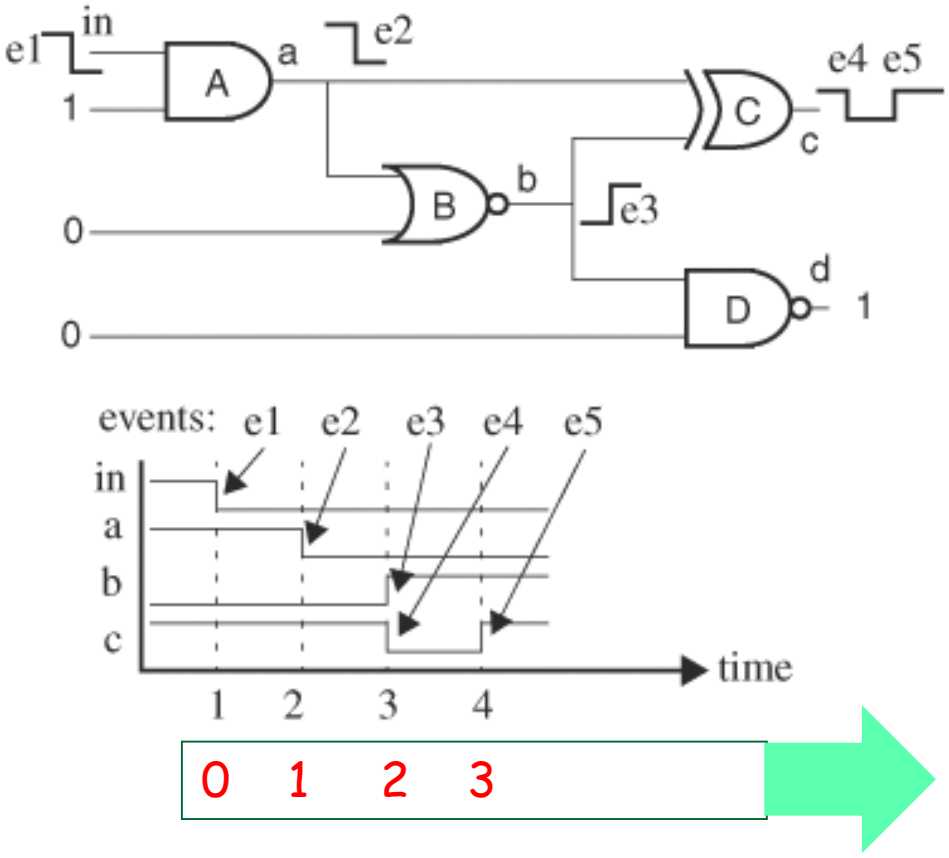
Event-Driven Simulation



- Each gate has a delay of one unit.
- 5 events: e1, e2, e3, e4, e5.

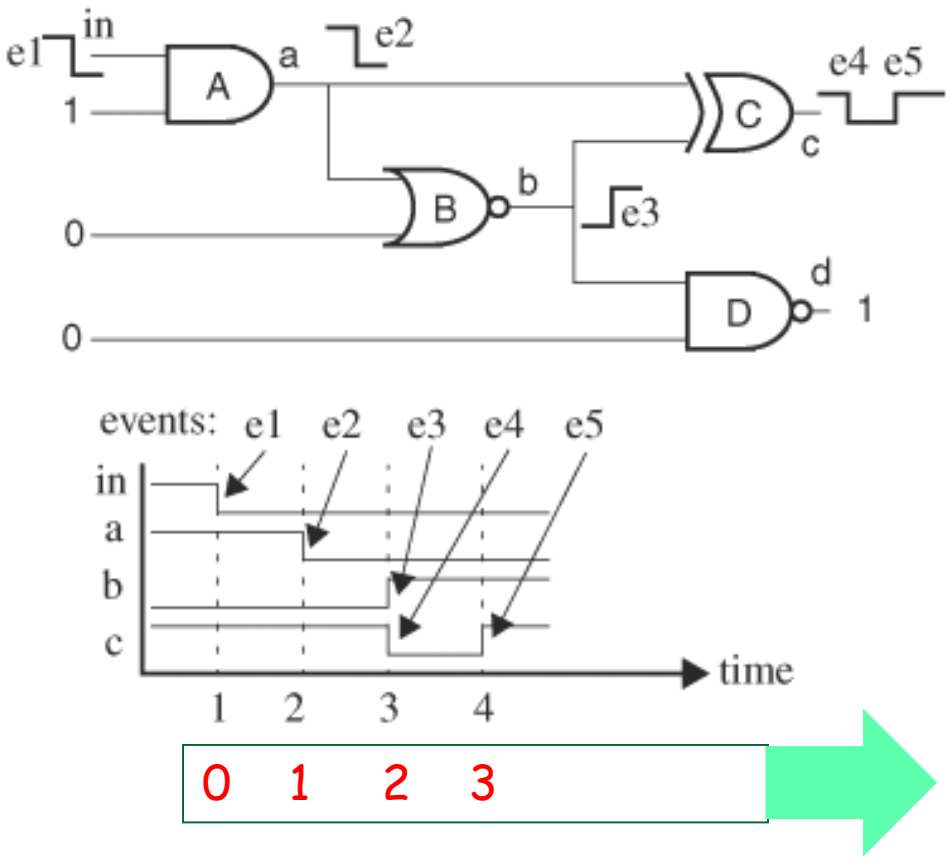
- An event is a value-change that occurs at a given time.
- The state of the circuit is only evaluated when a change occurs in the circuit.
- The output of an element might change as we know that such a change can only occur after an input changes.
- If we only monitor the inputs to elements, an output might change;
  - the logic function of the element determines whether or not a change actually occurs.

Example



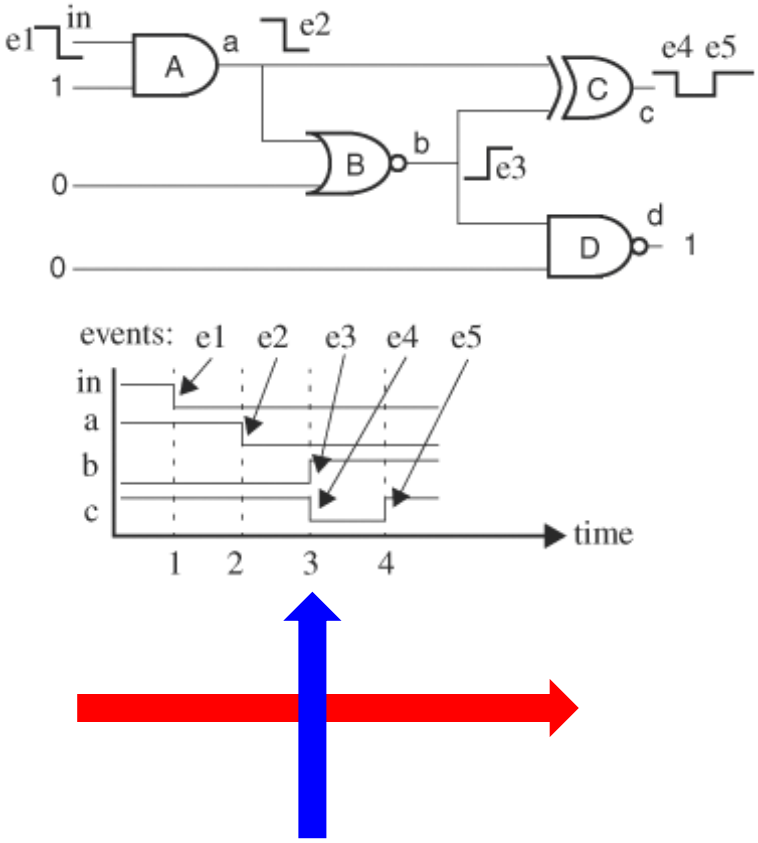
- The first event is the falling transition at input in occurring at time 0, and the affected gate is A.
- Event e1 is stored in an event queue at time slot 0, and it is the first event to be evaluated.
- An evaluation on gate A produces a falling transition on its output, e2, at time 1, due to the delay of the gate.
- Event e2 is placed into the event queue at time slot 1.
- When gate A is done evaluating, e1 is deleted from the event queue.

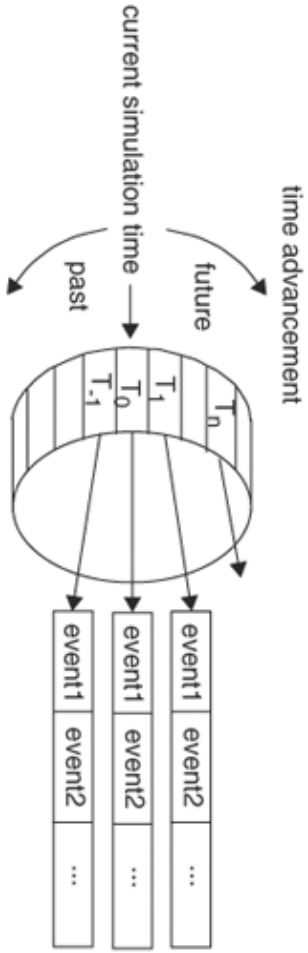
Example



- The next event e2 is taken off the queue.
- The affected gates are B and C.
- The evaluation of gates B and C resulting from e2 create output transitions e3 and e4, both at time 2, which are then placed at time slot 2.
- Event e2 is deleted.
- Event e4 is taken from the queue, and its evaluation generates no further events as there is no fanout.
- Thus, e4 is deleted.
- Next, e3 is evaluated and is found to produce e5 at time 3, which is queued at time slot 3.
- Then e3 is done and deleted.
- Finally, e5 affects no gates, so the chain of evaluation stops.

Event Scheduling & Evaluation



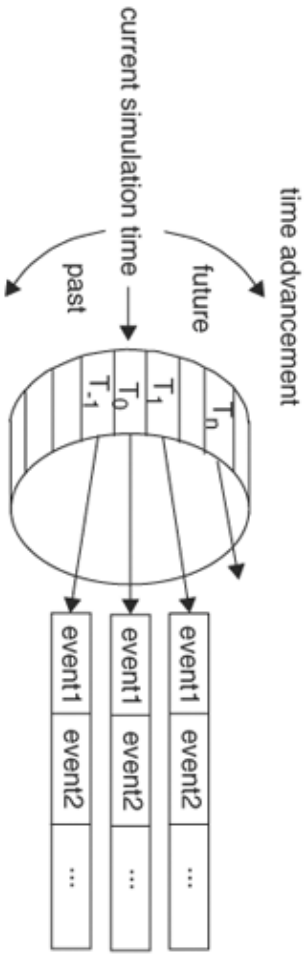


- The event manager uses a two-dimensional queue, called a *linked list.*
  - One dimension is a queue of time slots, with each entry pointing to
  - another queue that holds all the events occurring at that time.

Timing Wheel/Event Manager

- Events are stored in an event manager, which sorts them according to event occurrence time.
- Events occurring at the same time are assumed to have an arbitrary order of occurrence.
- Evaluations are then executed on the stored events starting from the earliest time.
- When the simulator is at time T, all events that occurred before time T must have been evaluated.

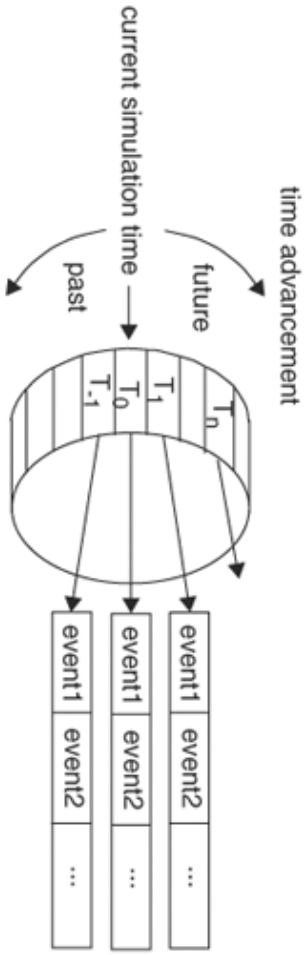
Timing Wheel



- A timing wheel has time slots, and each time slot points to a queue that stores all events occurring at that time.
- Simulation progresses along the time slots.
- At each time slot, the events of the queue are evaluated one at a time until it is empty.
- During an event evaluation, if events are generated, they are inserted into the queue in the time slots at which they will occur.
- When all the events in the queue are examined, simulation advances to the next time slot.

SV Simulation Stratified Regions

a) Preponed
b) Pre-Active
c) Active
d) Inactive
e) Pre-NBA
f) NBA
g) Post-NBA
h) Pre-Observed
i) Observed
j) Post-Observed
k) Reactive
l) Re-Inactive
m) Pre-Re-NBA
n) Re-NBA
o) Post-Re-NBA
p) Pre-Postponed
q) Postponed



- Events in the same time slot have to be prioritized according to IEEE Verilog standards.
- As in Verilog, we consider 5 abstracted layers of events in the following order of processing:
  - 1. Active
  - 2. Inactive
  - 3. Nonblocking assign update
  - 4. Monitor
  - 5. Future events.

Scheduling Semantics

1. Active
2. Inactive
3. Nonblocking assign update
4. Monitor
5. Future events.

- Active events at the same simulation time are processed in an arbitrary order.
- The sampling step is an active event and thus is executed at the moment the nonblocking statement is encountered.

Scheduling Semantics

1. Active
2. Inactive
3. Nonblocking assign update
4. Monitor
5. Future events.

- Inactive events are processed only after all active events have been processed.
  - An example of an inactive event is an explicit zero-delay assignment
  - (#0 x = y),
  - which occurs at the current simulation time
- but is processed after all active events at the current simulation time have been processed.

Scheduling Semantics

- A nonblocking assignment executes in two steps.
  - First it samples the values of the right-side variables.
  - Then it updates the values to the left-side variables.

1. Active
2. Inactive
3. Nonblocking assign update
4. Monitor
5. Future events.

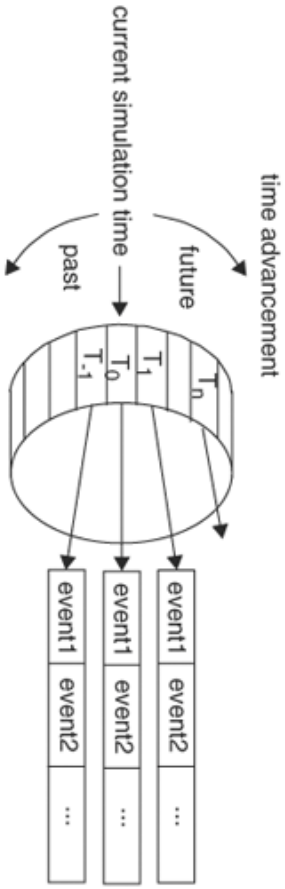
- The sampling step is an active event and
  - thus is executed at the moment the nonblocking statement is encountered.
- A nonblocking assign update event is the updating step of a nonblocking assignment and
  - it is executed only after both active and inactive events at the current simulation time have been processed.

Scheduling Semantics

1. Active
2. Inactive
3. Nonblocking assign update
4. Monitor
5. Future events.

- Monitor events are generated by system tasks $monitor and $strobe,
  - which are executed as the last events at the current simulation time to capture steady values of variables at the current simulation time.
  - These cannot create new events, so will always be executed last at a simulation time.
- Finally, events that are to occur in the future are future events.

Scheduling Semantics

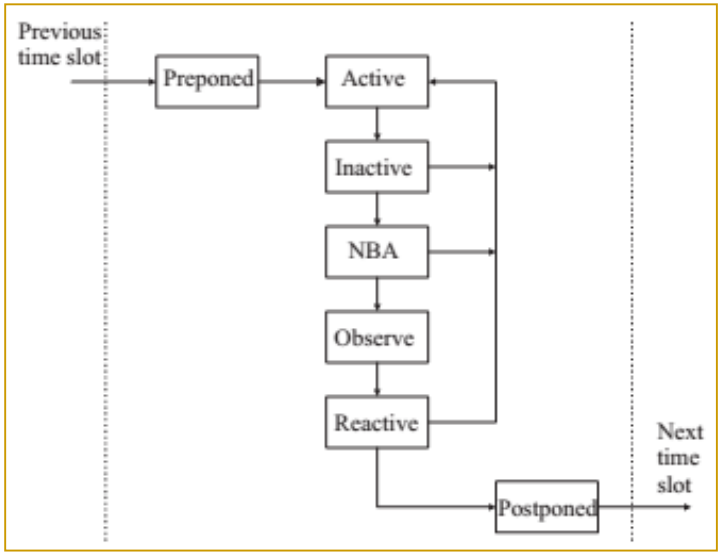


1. Active
2. Inactive
3. Nonblocking assign update
4. Monitor
5. Future events.

- Each time slot points to 4 subqueues corresponding to the 4 groups of events.
- Within each subqueue, the order of events is arbitrary.

The Simulation Cycle

- Each iteration of the loop is one cycle.
- T is the current simulation time.




```
while (there are events) {
  if (no active events) {
    if (there are events in the next list) {
      activate all events in the next list;
      else { /*if no more events at the current time*/
          advance T to the next event time; }
      }
      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;
          }
  }
```

Only Active Events Are Processed !

```
while (there are events) {
  if (no active events) {
    if (there are events in the next list) {
      activate all events in the next list;
      else { /*if no more events at the current time*/
          advance T to the next event time; }
      }
      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;
          }
  }
```

- **Only active events are processed**, but a new event may be assigned to one of regions listed above.
- The list of active events is one of the lists that has been created during some previous simulation cycle, together with any (active) events that are generated during the current cycle.

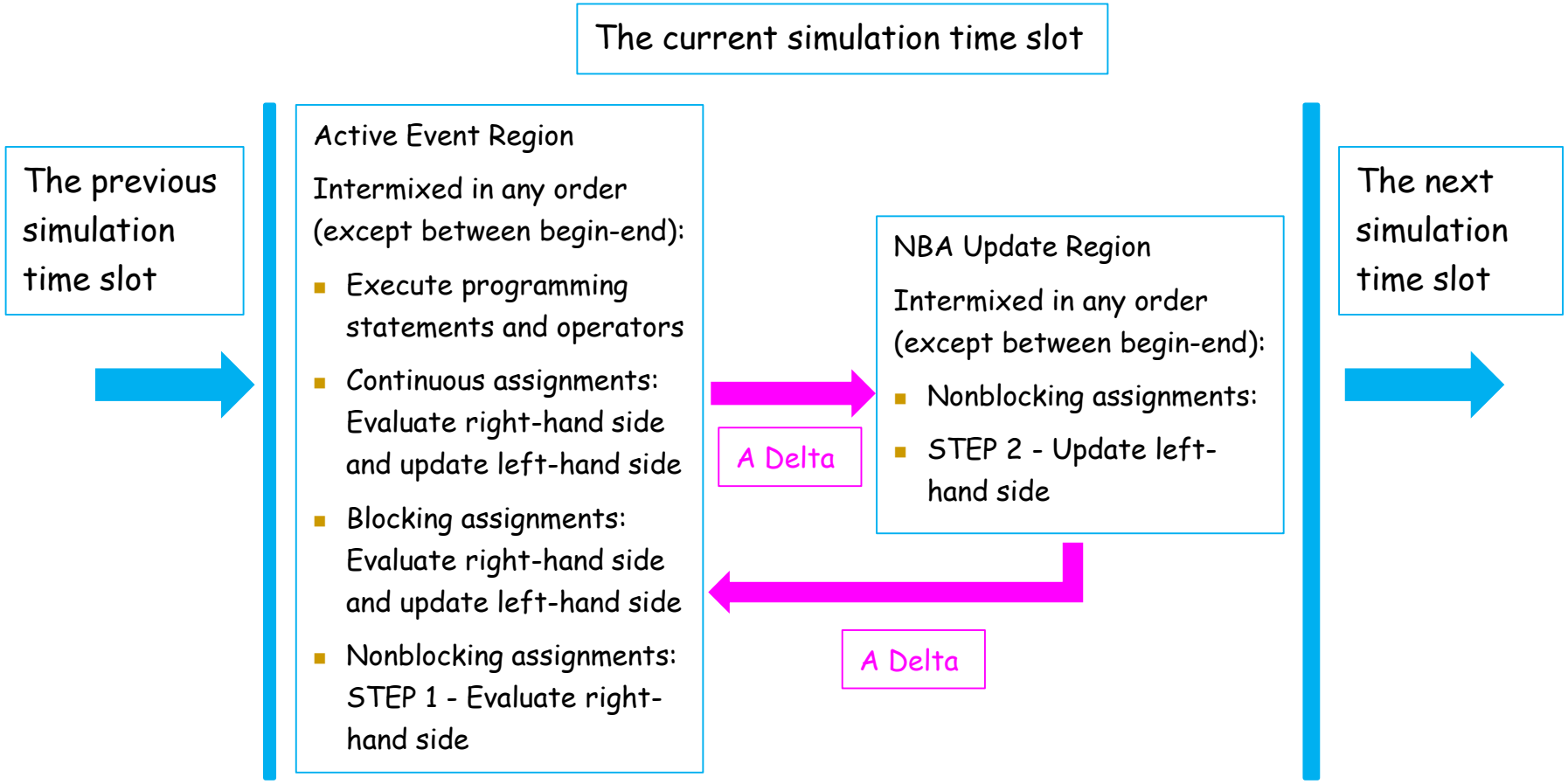
A Event-Driven Simulation Cycle

```
while (there are events) {
    if (no events for current time) advance simulation time.
    Foreach (event at the current time) {
        remove the event and process as follows:
        if (event is update)
            update the variables or nodes.
            schedule evaluation events for the affected processes.
        else // event is evaluation
            evaluate the processes
            schedule update events for outputs that change
    } // end of Foreach
} // end of while
```

Active and NBA Update events

- SystemVerilog divides a simulation time slot into several event regions, which are processed in a controlled order.
- This provides design and verification engineers some control over the order in which events are processed.
- Most of the event regions are for verification purposes, and are not discussed here.
- RTL and gate-level models primarily use two of these event regions:
  - the Active event region and the NBA Update event region.

A Simplified SV Event Scheduling Regions



The transition from one event region to the next is referred to as a delta.

Event Regions During Simulation

- Events in a begin-end statement group are scheduled into an event region in the order in which the statements are listed, and are executed in that order when the event region is processed.

- Events from concurrent processes, such as multiple always procedures, are scheduled into the event region in an arbitrary order chosen by the simulator.
  - The RTL designer has no control over the order in which concurrent events are scheduled, but the designer does have control regarding the region in which events are scheduled.

Event regions and event scheduling

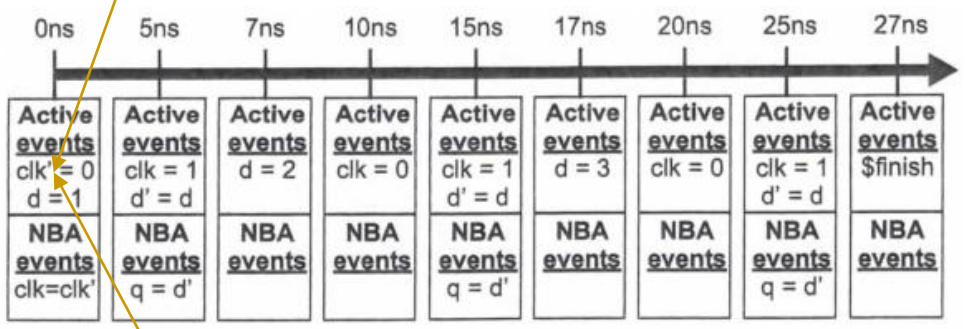
- Each simulation time slot is divided into several event regions.
- An event region is used to schedule an activity that must be processed by the simulator.
- For example, if a clock oscillator changes values every 5 nanoseconds, beginning at time 0, then the simulator would schedule a simulation event for the clock signal at 0ns, 5ns, 10ns, and so forth in the simulation time line.
- Likewise, if a programming statement will be executed at simulation time 7 nanoseconds, the simulator will schedule an event to evaluate that statement at the 7ns time slot in the simulation time line.

Event Regions During Simulation

- Simulators will execute all scheduled events in a region before transitioning to the next region.
  - As events are processed, they are permanently removed from the event list.
  - Each region will be empty before simulation proceeds to the next region.

- As events are processed in a later region, they can possibly schedule new events in a previous region.
  - After the later region has been processed, simulation will cycle back through the event regions to process any newly scheduled events.
  - The iterations through all event regions will continue until all regions are empty (i.e.: no new events for that moment of simulation time are being scheduled).

```
module top;
  timeunit 1ns/1ns;
  logic       clock;
  logic [7:0] d;
  logic [7:0] q;
  test    i1 (.*);   // connect top module to test module
  d_reg i2 (.*);  // connect top module to d_reg module
  initial begin    // clock oscillator
    clock <= 0;      // initialize clock at time 0
    forever #5 clock = ~clock;   // toggle clock every 5ns
  end
endmodule: top
```

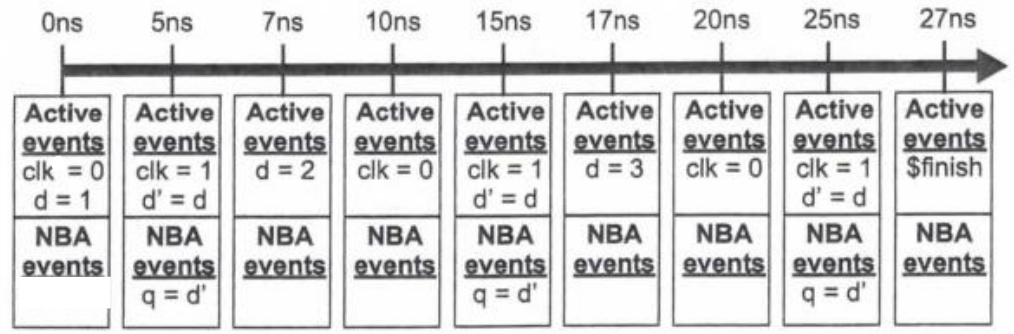


Variable clk' samples the value 0.

```
module d_reg (input  logic  clock,
              input  logic [7:0] d,
              output logic [7:0] q );
  timeunit 1ns/1ns;
  always @(posedge clock)
    q <= d;
endmodule: d_reg
```

```
module test (input  logic clock,
             output logic [7:0] d,
             input  logic [7:0] q  );
  timeunit 1ns/1ns;
  initial begin
    d = 1;
    #7  d = 2;
    #10 d = 3;
    #10 $finish;
  end
endmodule: test
```

```
module top;
  timeunit 1ns/1ns;
  logic       clock;
  logic [7:0] d;
  logic [7:0] q;
  test    i1 (.*);   // connect top module to test module
  d_reg i2 (.*);  // connect top module to d_reg module
  initial begin    // clock oscillator
    clock = 0;     // initialize clock at time 0
    forever #5 clock = ~clock;   // toggle clock every 5ns
  end
endmodule: top
```




```
module d_reg (input  logic  clock,
              input  logic [7:0] d,
              output logic [7:0] q );
  timeunit 1ns/1ns;
  always @(posedge clock)
    q <= d;
endmodule: d_reg
```

```
module test (input  logic clock,
             output logic [7:0] d,
             input  logic [7:0] q  );
  timeunit 1ns/1ns;
  initial begin
    d = 1;
    #7  d = 2;
    #10 d = 3;
    #10 $finish;
  end
endmodule: test
```

Five Ways to Model Delays

- (a) Left-hand side (LHS) of blocking assignments:
  - #5 a = b;
- (b) Right-hand side (RHS) of blocking assignments:
  - a = #5 b;
- (c) LHS of nonblocking assignments:
  - #5 a <= b;
- (d) RHS if nonblocking assignments:
  - a <= #5 b;
- (e) LHS of continuous assignments:
  - **assign** #5 a = b;

Five Ways to Model Delays

- (a) #5 a = b;
- (b) a = #5 b;
- (c) #5 a <= b;
- (d) a <= #5 b;
- (e) **assign** #5 a = b;

- None of these constructs is needed for RTL modeling.
- (a) and possibly (c) and (d) are useful for testbench writing.
- (d) is a transport (pure) delay that can be used to model delays in sequential logic (such as clock to output delay in a flip-flop).
- (e) is an inertial delay that can be used to model delays in combinational logic.

(a) #5 a = b

```
#5 a = b;
```

- It is the Simulation event scheduling time!
  - In form (a), it means: #5 [event a=b]
  - the simulator waits for, e.g. 5 time units and then executes the assignment.
  - Any changes to inputs during this wait period are ignored, and the final input values will be used at the execution moment.
- It does not model real hardware delay,
  - but is useful for describing a waveform in a testbench.

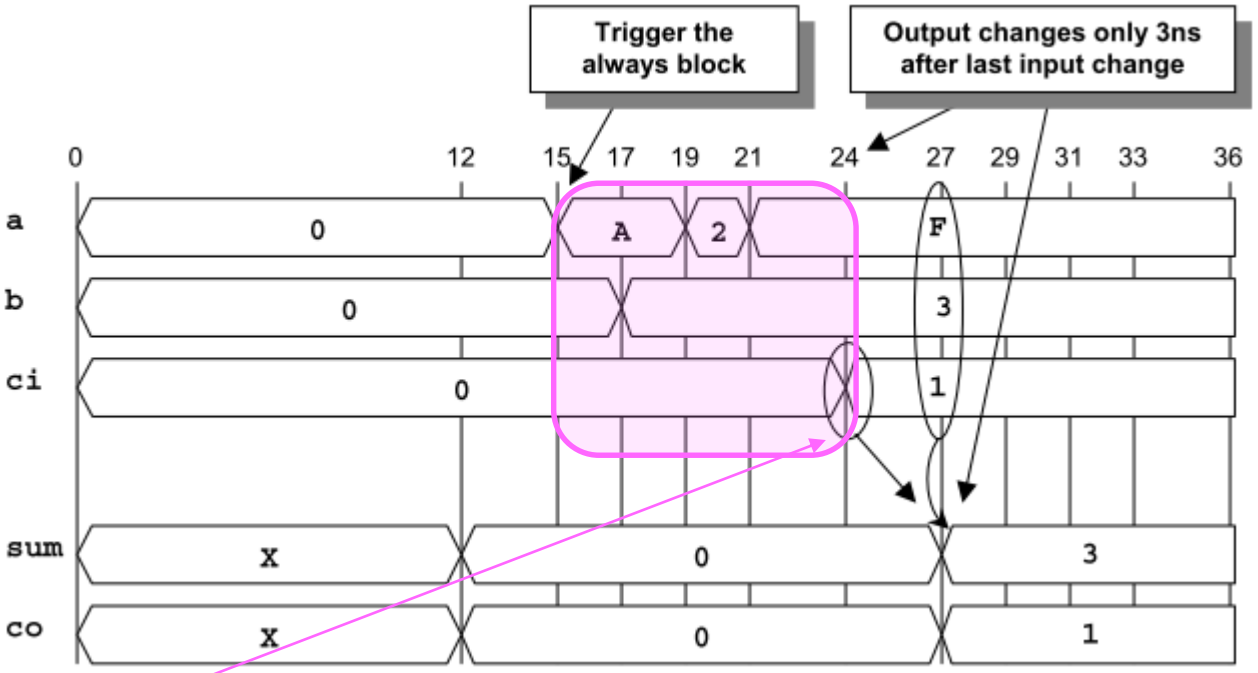
#12 {co, sum} = a + b + ci

```
module adder_t1 (co, sum, a, b, ci);
output      co;
output      [3:0] sum;
input       [3:0] a, b;
input       ci;
reg         co;
reg         [3:0] sum;
always @(a or b or ci)
#12 {co, sum} = a + b + ci;
endmodule
```

- The outputs should be updated 12ns after input changes.
- If a changes at time 15, then if the a, b and ci inputs all change during the next 9ns, the outputs will be updated with the latest values of a, b and ci.

#12 {co, sum} = a + b + ci

```
module adder_t1 (co, sum,
a, b, ci);
output      co;
output      [3:0] sum;
input       [3:0] a, b;
input       ci;
reg         co;
reg         [3:0] sum;
always @(a or b or ci)
#12 {co, sum} = a + b + ci;
endmodule
```



- The outputs should be updated 12ns after input changes. (F_{16}=15)
- If a changes at time 15, then if the a, b and ci inputs all change during the next 9ns, the outputs will be updated with the latest values of a, b and ci.

(b) a = #5 b (Transport Delay)

```
a = #5 b;
```

- It causes the present value of the RHS to be scheduled for assignment at time 5 in the future.
- This is a transport or pure delay
  - every change, no matter how rapid, is transmitted to the output.
- This could be used to model a transmission line.
- This is not particularly useful for testbench design.

(c) #5 a <= b

```
#5 a <= b;
```

- It is the Simulation event scheduling time!
  - In form (c), it means: #5 [a<=b]
  - the simulator waits for, e.g. 5 time units and then executes the assignment.
  - Any changes to inputs during this wait period are ignored, and the final input values will be used at the execution moment.
- It does not model real hardware delay, but is useful for describing a waveform in a testbench.
- It has no advantage over form (a).

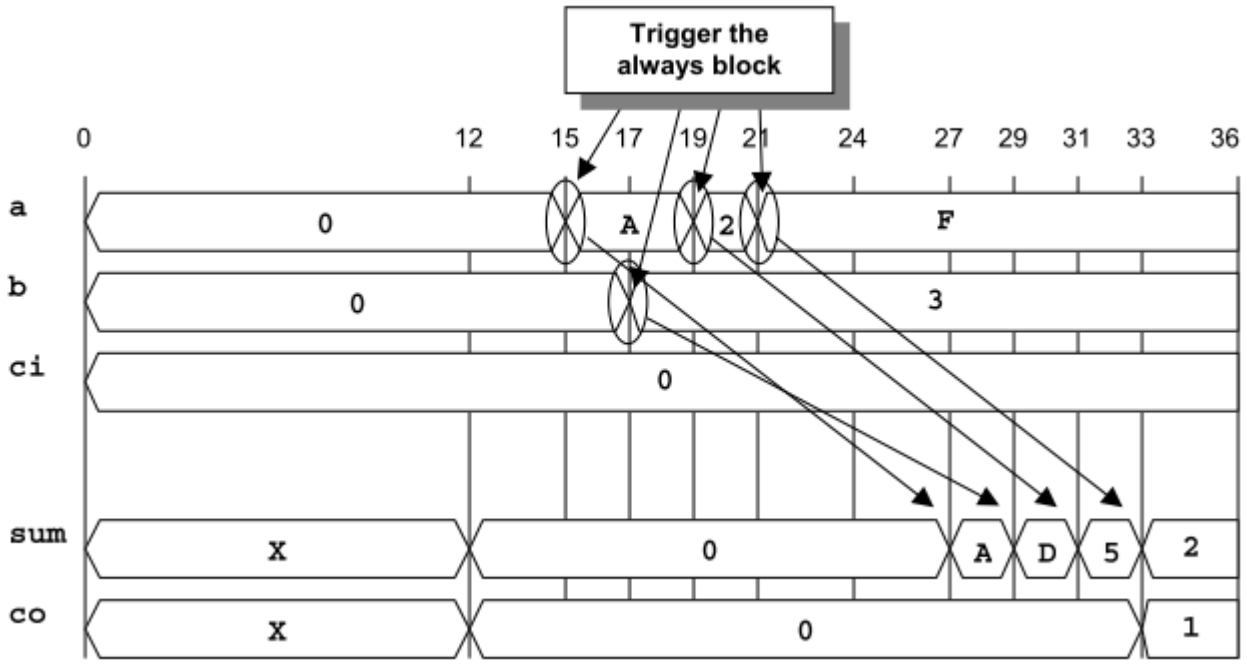
(d) a <= #5 b (Transport Delay)

```
a <= #5 b;
```

- It can be used to model delays in combinational logic.
- It is a transport delay.

{co, sum} <= #12 a+b+ci;

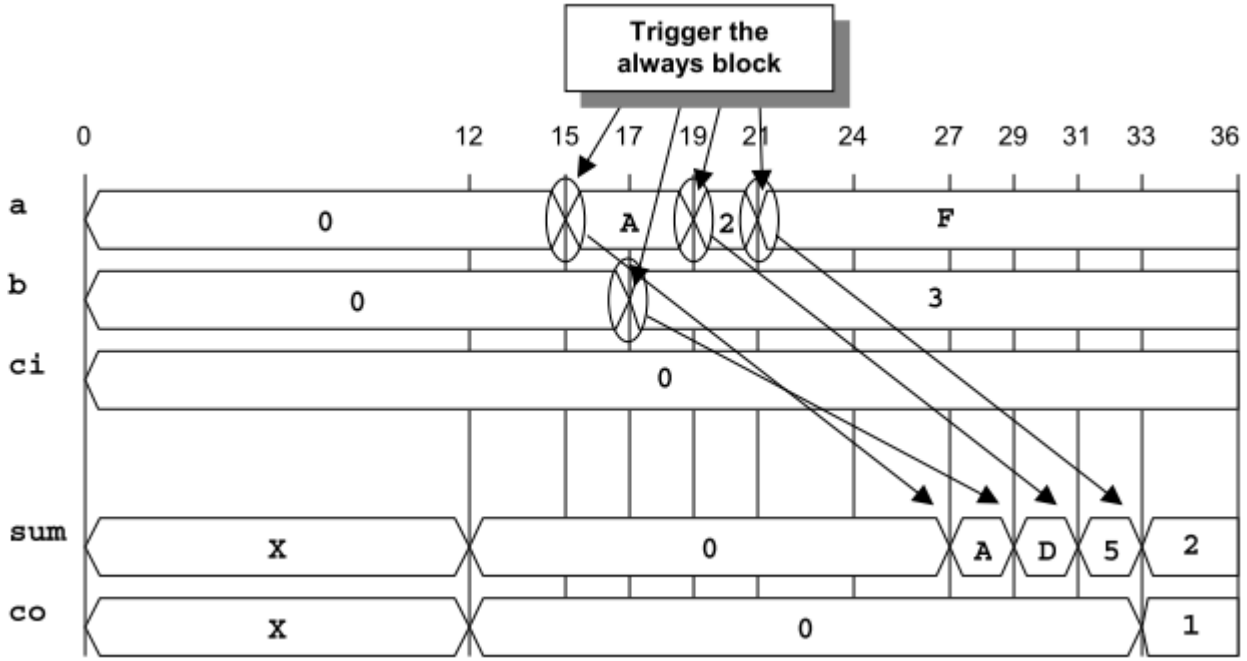
```
module adder_t3 (co, sum,
a, b, ci);
output      co;
output [3:0] sum;
input  [3:0] a, b;
input       ci;
reg         co;
reg         [3:0] sum;
always @(a or b or ci)
{co, sum} <= #12 a+b+ci;
endmodule
```



- If a changes at time 15, then all inputs will be evaluated and new output values will be queued for assignment 12ns later.

{co, sum} <= #12 a+b+ci;

```
always @(a or b or ci)
{co, sum} <= #12 a+b+ci;
endmodule
```



- Immediately after the outputs have been queued (scheduled for future assignment) but not yet assigned, the always block will again be setup to trigger on the next input event.
- All input events will queue new values to be on the outputs after a 12ns delay.
- This coding style models combinational logic with transport delays.

(e) assign #5 a = b (Inertial Delay)

```
assign #5 a = b;
```

- It models an inertial delay.
- This form of delay best models combinational logic.
- Continuous assignments do not "queue up" output assignments, they only keep track of the next output value and when it will occur.

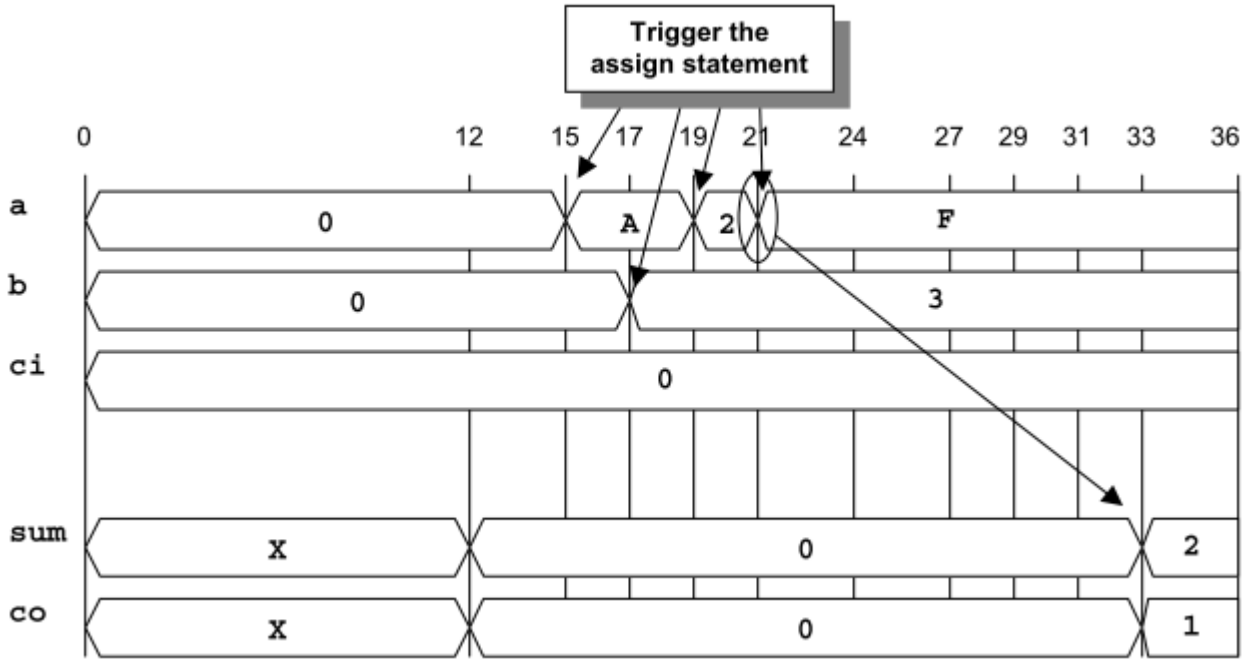
assign #12 {co, sum} = a+b+ci;

```
module adder_t4 (co, sum, a, b, ci);
output      co;
output      [3:0] sum;
input       [3:0] a, b;
input       ci;
assign #12 {co, sum} = a+b+ci;
endmodule
```

- The outputs do not change until 12ns after the last input change (12ns after all inputs have been stable).
- Any sequence of input changes that occur less than 12ns apart will cause any future scheduled output-event (output value with corresponding assignment time) to be replaced with a new output-event.

assign #12 {co, sum} = a+b+ci;

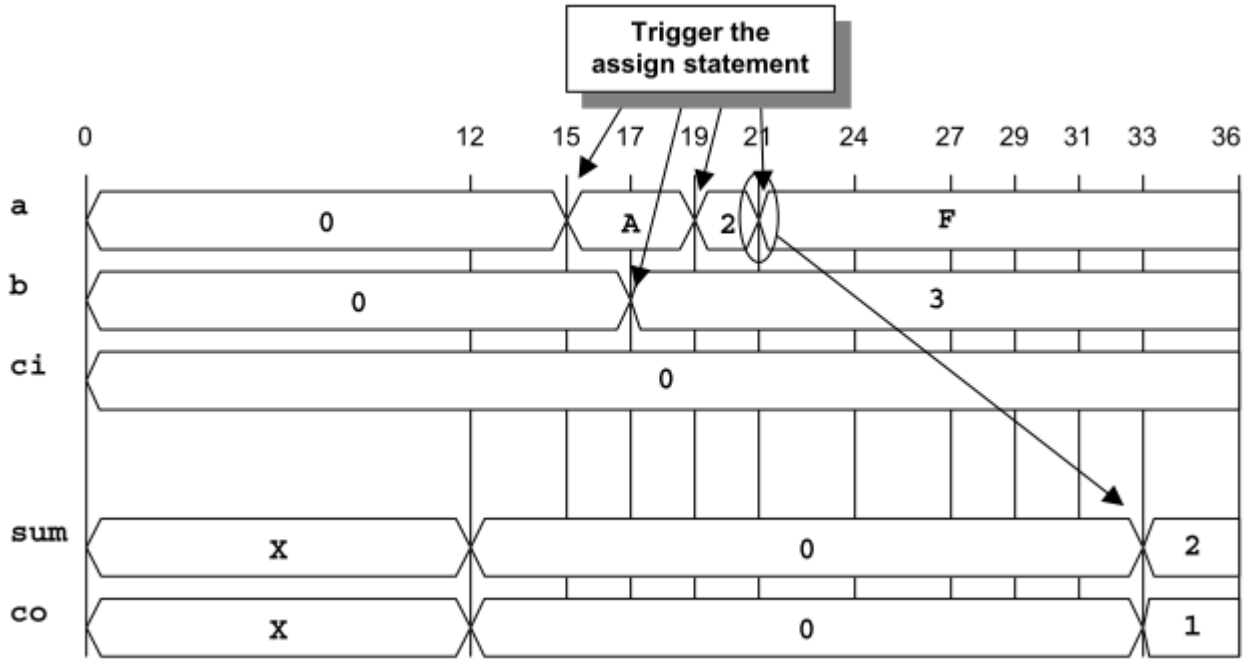
```
module adder_t4 (co, sum, a,
b, ci);
output      co;
output      [3:0] sum;
input       [3:0] a, b;
input       ci;
assign #12 {co, sum} = a+b+ci;
endmodule
```



- The outputs do not change until 12ns after the last input change (12ns after all inputs have been stable).
- Any sequence of input changes that occur less than 12ns apart will cause any future scheduled output-event (output value with corresponding assignment time) to be replaced with a new output-event.

assign #12 {co, sum} = a+b+ci;

```
module adder_t4 (co, sum, a,
b, ci);
output      co;
output      [3:0] sum;
input       [3:0] a, b;
input       ci;
assign #12 {co, sum} = a+b+ci;
endmodule
```



- The first a-input change occurs at time 15, which causes an output event to be scheduled for time 27,
  - but a change on the b-input and two more changes on the a-input at times 17, 19 and 21, respectively, cause three new output events to be scheduled.
- Only the last output event completes and the outputs are assigned at time 33.

Guidelines on Delays

- Any delay added to statements inside of an always block does not accurately model the behavior of real hardware and should not be done.
- The one exception is to carefully add delays to the right hand side of nonblocking assignments,
  - which will accurately model transport delays, generally at the cost of simulator performance.
- Adding delays to any sequence of continuous assignments, or modeling complex logic with no delays inside of an always block and driving the always block outputs through continuous assignments with delays,
  - both accurately model inertial delays and are recommended coding styles for modeling combinational logic.

Assignment Statements

```
module evaluates (out);
    output out;
    logic a, b, c;
    initial begin
        a = 0;
        b = 1;
        c = 0;
    end
    always c = #5 ~c;
    always @(posedge c) begin
        a <= b; // evaluates, schedules,
        b <= a; // and executes in two steps
    end
endmodule
```

a = 0

b = 1

a = ?

b = ?

Assignment Statements

```
module evaluates (out);
    output out;
    logic a, b, c;
    initial begin
        a = 0;
        b = 1;
        c = 0;
    end
    always c = #5 ~c;
    always @(posedge c) begin
        a <= b; // evaluates, schedules,
        b <= a; // and executes in two steps
    end
endmodule
```

a = 0

b = 1

a = 1

b = 0

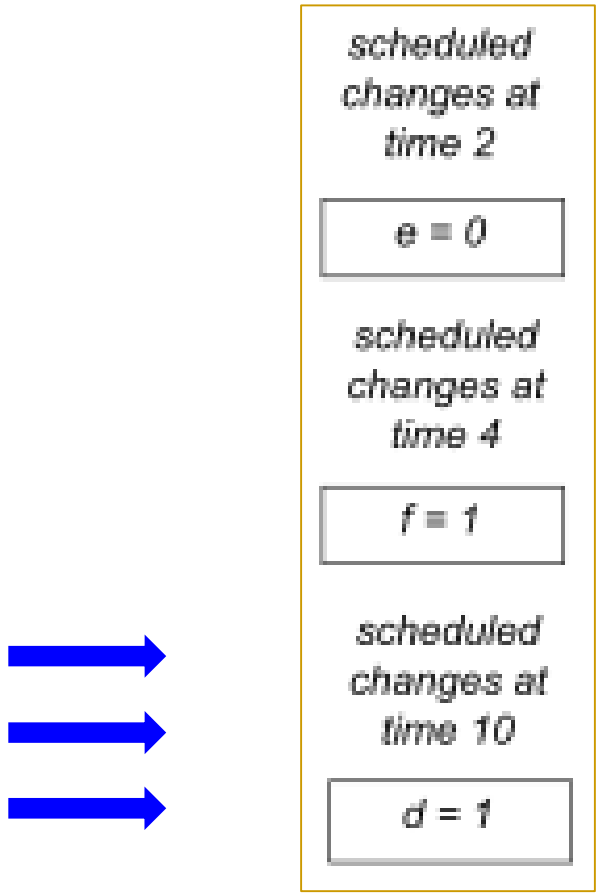
When to Get the Values?

```
module nonblock1;
logic a, b, c, d, e, f;
initial begin
      a = #10 1;
      b = #2 0;
      c = #4 1;
end
initial begin
      d <= #10 1;
      e <= #2 0;
      f <= #4 1;
end
endmodule
```

- The nonblocking assignment evaluates and schedules the assignment, but it does not block the execution of subsequent statements in a begin-end block.

Assignment Statements

```
module nonblock1;
logic a, b, c, d, e, f;
// blocking assignments
initial begin
    a = #10 1; // a will be assigned 1 at time 10
    b = #2 0;  // b will be assigned 0 at time 12
    c = #4 1;  // c will be assigned 1 at time 16
end
// nonblocking assignments
initial begin
    d <= #10 1; // d will be assigned 1 at time 10
    e <= #2 0;  // e will be assigned 0 at time 2
    f <= #4 1;  // f will be assigned 1 at time 4
end
endmodule
```



Assignment Statements

```
module nonblock2;
logic a, b;
initial begin
a  = 0;
b  = 1;
a <= b;
b <= a;
end
initial begin
$monitor ($time, ,"a = %b b = %b", a, b);
#100 $finish;
end
endmodule
```

- a = ?
- b = ?

Assignment Statements

```
module nonblock2;
logic a, b;
initial begin
a  = 0;
b  = 1;
a <= b; // evaluates, schedules,
b <= a; // and executes in two steps
end
initial begin
$monitor ($time, ,"a = %b b = %b", a, b);
#100 $finish;
end
endmodule
```

a = 0

b = 1

a = 1

b = 0

What is the Final Value?

```
module multiple;
logic a;
initial a = 1;
initial begin
    a <= #4 0;
    a <= #4 1;
end
endmodule
```

- The nonblocking assignment evaluates and schedules the assignment, but it does not block the execution of subsequent statements in a begin-end block.

- a = ?

Nonblocking Inside Procedural Blocks

```
module multiple;
logic a;
initial a = 1;
// The assigned value of the variable is determinate

initial begin
    a <= #4 0; // schedules a = 0 at time 4
    a <= #4 1; // schedules a = 1 at time 4
end // At time 4, a = 1
endmodule
```

- Multiple nonblocking assignments to a common variable.
- If they have the same scheduled execution time, then the last one in the code will be the final value.

Nonblocking Inside Procedural Blocks

```
module multiple;
logic a;
initial a = 1;
initial begin
    a <= #5 0;
    a <= #4 1;
    end
endmodule
```

- a = ?

Nonblocking Inside Procedural Blocks

```
module multiple;
logic a;
initial a = 1;
// The assigned value of the variable is determinate

initial begin
    a <= #5 0; // schedules a = 0 at time 5
    a <= #4 1; // schedules a = 1 at time 4
end // At time 5, a = 0
endmodule
```

- Multiple nonblocking assignments to a common variable.
- If they have the different scheduled execution times, then the last one event in the scheduled queue will give the final value.

Concurrent Procedural Blocks

```
module multiple2;
logic a;
initial a  = 1;
initial a <= #4 0;
initial a <= #4 1;
endmodule
```

- a = ?

Concurrent Procedural Blocks

```
module multiple2;
logic a;
initial a  = 1;
    initial a <= #4 0; // schedules 0 at time 4
    initial a <= #4 1; // schedules 1 at time 4
// At time 4, a = ??
// The assigned value of the variable is indeterminate
endmodule
```

- a = 1 (simulator's decision)

- If the simulator executes two procedural blocks concurrently and if these procedural blocks contain nonblocking assignment operators to the same variable, the final value of that variable is indeterminate.

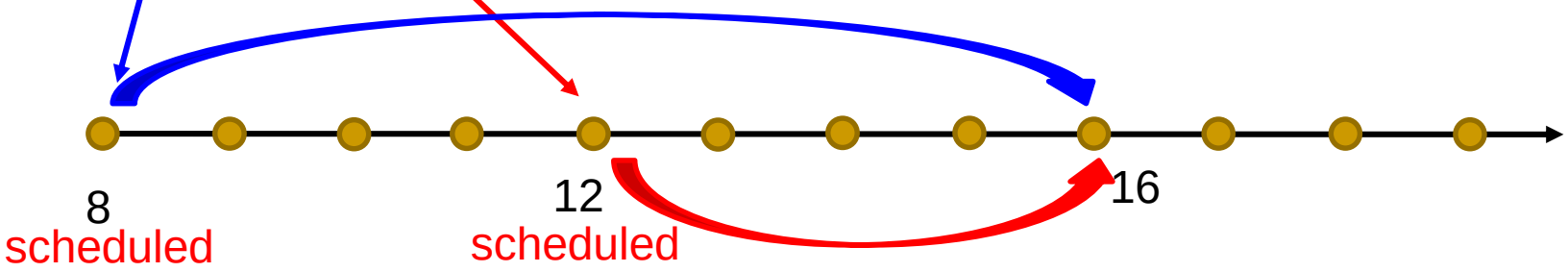
Assignment Statements

```
module multiple3;
logic a;
initial #8  a <= #8 1;
initial #12 a <= #4 0;
endmodule
```

- a = ?

Assignment Statements

```
module multiple3;
logic a;
initial #8 a  <= #8 1;  // executed at time 8; schedules an update of 1 at time 16
initial #12 a <= #4 0; // executed at time 12; schedules an update of 0 at time 16
endmodule
```



- Since the update of a to the value 1 is scheduled before the update of a to the value 0, then it is determinate that a will have the value **0** at the end of time slot 16.

The Order in Active Region

```
module tester();
logic [1:0] a, b;
initial
begin
a = 2'b00;
b = 2'b11;
#10;
a <= b;
b <= a;
a = 2'b10;
b = 2'b01;
#20 $finish();
end
endmodule

A=?
B=?
```

```
module tester();
logic [1:0] a, b;
initial
begin
a = 2'b00;
b = 2'b11;
#10;
a = 2'b10;
b = 2'b01;
a <= b;
b <= a;
#20 $finish();
end
endmodule
A=?
B=?
```

```
module tester();
logic [1:0] a, b;
initial
begin
a = 2'b00;
b = 2'b11;
#10;
a <= b;
a = 2'b10;
b <= a;
b = 2'b01;
#20 $finish();
end
endmodule

A=?
B=?
```

The Order in Active Region

```
module tester();
logic [1:0] a, b;
initial
begin
a = 2'b00;
b = 2'b11;
#10;
a <= b;
b <= a;
a = 2'b10;
b = 2'b01;
#20 $finish();
end
endmodule

A=11
B=00
```

```
module tester();
logic [1:0] a, b;
initial
begin
a = 2'b00;
b = 2'b11;
#10;
a = 2'b10;
b = 2'b01;
a <= b;
b <= a;
#20 $finish();
end
endmodule
A=01
B=10
```

```
module tester();
logic [1:0] a, b;
initial
begin
a = 2'b00;
b = 2'b11;
#10;
a <= b;
a = 2'b10;
b <= a;
b = 2'b01;
#20 $finish();
end
endmodule

A=11
B=10
```

The Order in Active Region

```
always @(posedge clk)
begin
x = 1;
y <= x;
y = 2;
end

always @(posedge clk)
begin
x = 0;
end

result:
0        x: x        y: x
5        x: 0        y: 1
```

```
always @(posedge clk)
begin
x = 0;
end

always @(posedge clk)
begin
x = 1;
y <= x;
y = 2;
end

result:
0        x: x        y: x
5        x: 1        y: 1
```

Nondeterministic

- Events may be processed from the active event list in any order
  - events can be added to the event lists in any order.
- This is the fundamental cause of non-determinism in SystemVerilog.
- Two things:
  - Statements between begin and end will be executed in the order stated.
  - Nonblocking assignments will be performed after blocking assignments.
- Everything else is indeterminate.
- The execution of a procedural block can be interrupted to allow another procedural block to be executed.

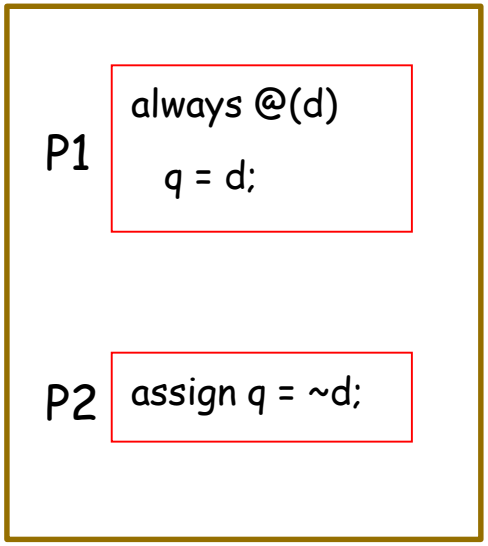
Race (Nondeterminism)

- When multiple processes execute simultaneously, a natural question to ask is how these processes are scheduled.
- When multiple processes are triggered at the same time,
  - the order in which the processes are executed **is not specified** by (IEEE) standards.
- It is arbitrary and it varies from simulator to simulator.
- This phenomenon is called nondeterminism.

Race (Nondeterminism)

- If the code is badly written, a race condition is likely to result a situation where
  - the procedure writing a value and the procedure reading that value are racing each other.
- Depending which completes first, either the original or the updated value may be read.

Nondeterministic Type I



- The first case arises when multiple statements execute in zero time, and there are several legitimate orders in which these statements can execute.
- The order in which they execute affects the results.
- A different order of execution gives different but correct results.
- Execution in zero time means that the statements are evaluated without advancing simulation time.
- Scheduled in the same class!

Nondeterministic Type I

```
always @(d)
   q = d;
```

```
assign q = ~d;
```

d: 0->1

a1) q = 1;

a2) q =0;

Final q=0

b1) q=0

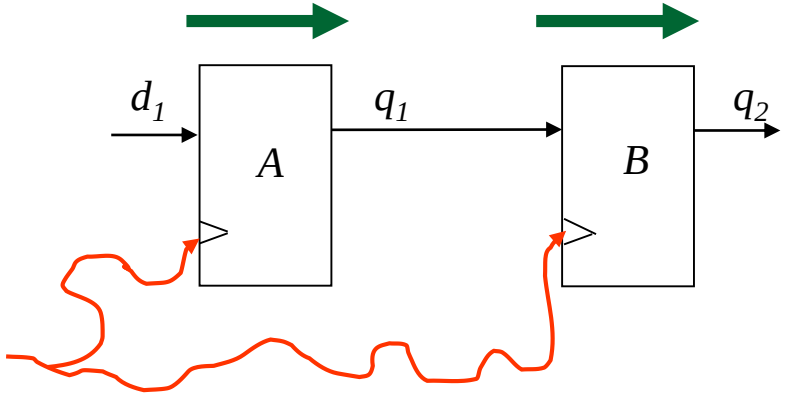
b2) q =1

Final q = 1

- Because the order in which they execute is not specified by the IEEE standard, two possible orders are possible.
- The two orders produce opposite results.
- This is a manifestation of nondeterminism.

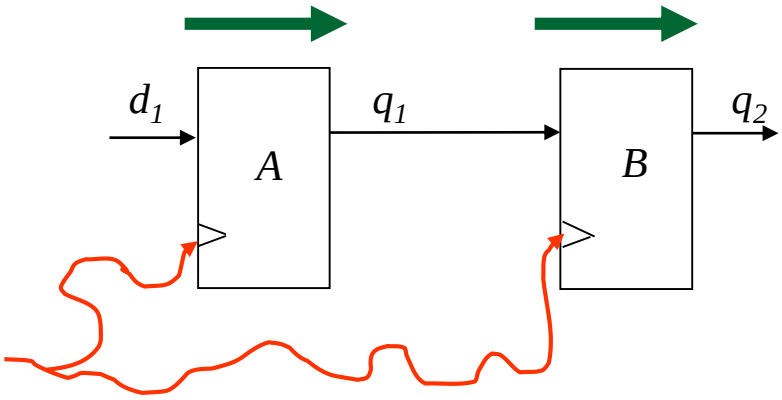
Nondeterministic Type I

- Another well-known example of nondeterminism of the first type occurs often in RTL code.
- The following D-flip-flop (DFF) module uses a blocking assignment and causes nondeterminism when the DFFs are instantiated in a chain.




```
module DFF (clk, q, d);
        input clk,  d;
        output q;
        reg q;
always @(posedge clk)
          q = d; // source of the problem
endmodule
module DFF_chain;
        DFF dff1(clk, q1, d1); // DFF1
        DFF dff2(clk, q2, q1); // DFF2
endmodule
```

```
module DFF (clk, q, d);
        input clk,  d;
        output q;
        reg q;
always @(posedge clk)
        q = d; // source of the problem
endmodule
module DFF_chain;
        DFF dff1(clk, q1, d1); // DFF1
        DFF dff2(clk, q2, q1); // DFF2
endmodule
```



- When the positive edge of clk arrives, either DFF instance can be evaluated first.
- If dff1 executes first, q1 is updated with the value of d1.
  - Then dff2 executes and makes q2 the value of q1.
  - In this order, q2 gets the latest value of d1.
- If dff2 is evaluated first, q2 gets the value of q1 before q1 is updated with the value of d1.
- Either order of execution is correct, and thus both values of q2 are correct.
- Therefore, **q2 may differ from simulator to simulator.**

Solution to Nondeterminism I: Nonblocking Assignments (Delta Model)

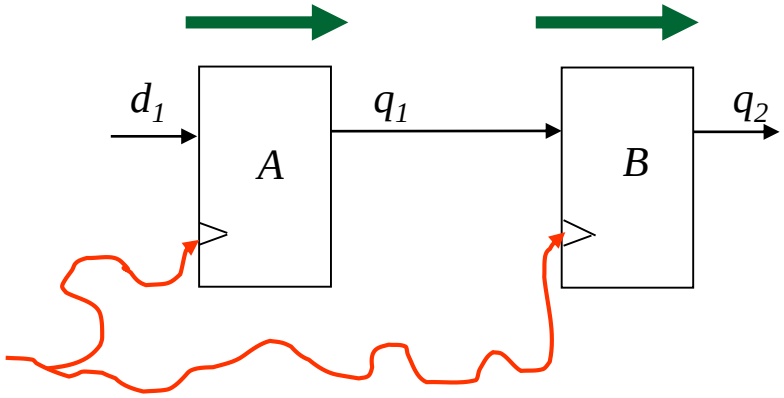
- A remedy to this race problem is to change the blocking assignment to a **nonblocking** assignment.
- Use different scheduled steps to implement the 2-step delta model.

- Scheduling of nonblocking assignments
  - samples the values of the variables on the right-hand side at the moment the nonblocking assignment is encountered, and
  - assigns the result to the left-side variable at the end of the current simulation time (for example, after all blocking assignments are evaluated).

Solution to Type I: Nonblocking

```
module DFF (clk, q, d);
        input clk,  d;
        output q;
        reg q;
always @(posedge clk)
        q <= d;
endmodule
module DFF_chain;
        DFF dff1(clk, q1, d1); // DFF1
        DFF dff2(clk, q2, q1); // DFF2
endmodule
```

- If we change the blocking assignment to a nonblocking assignment,
  - the output of dff1 always gets the updated value of d1,
  - whereas that of dff2 always gets the preupdated value of q1, regardless of the order in which they are evaluated.



Nondeterminism with Nonblocking Assignments

- The two orders produce different values of x and both values of x are correct:
- always @(posedge clk)
- begin
- x <= 1'b0;
- x <= 1'b1;
- end

- Nonblocking assignment does not eliminate **all** nondeterminism or race problems.
- The code contains a race problem even though both assignments are nonblocking.
- Both assignments to variable x occur at the end of the current simulation time and the order that x gets assigned is unspecified in IEEE standards.
- Scheduled in the Same class.

Nondeterministic Type II

```
always @(posedge clk)
begin
   x = 1'b0;
   y = x;
end

always @(posedge clk)
begin
   x = 1'b1;
end
```

- The second case of nondeterminism arises from interleaving procedural statements in blocks executed at the same time.
- When two procedural blocks are scheduled at the same time, there is no guarantee that all statements in a block finish before the statements in the other block begin.
- The statements from the two blocks can execute in any interleaving order.

Nondeterministic Type II

```
always @(posedge clk)
begin
  x = 1'b0;
  y = x;
end

always @(posedge clk)
begin
  x = 1'b1;
end
```

- One interleaving order is
  - x = 1'b0;
  - y = x;
  - x = 1'b1;
- In this case, y = 0.

- Another interleaving order is
- x = 1'b0;
- x = 1'b1;
- y = x;
- In this case, y = 1.

- The always block does not define atomicity.
- There is no simple fix for this nondeterminism.

Rules to Avoid Races

- Do not assign to the same variable from two or more procedures.
- Some experts argue that blocking and nonblocking assignments should not be mixed in the same procedure.
- It is not a good idea to use both types of assignment to the same variable as some synthesis tools will object.

Rules to Avoid Races

```
always @(posedge clock)
  b = c;
always @(posedge clock)
  a = b;
```

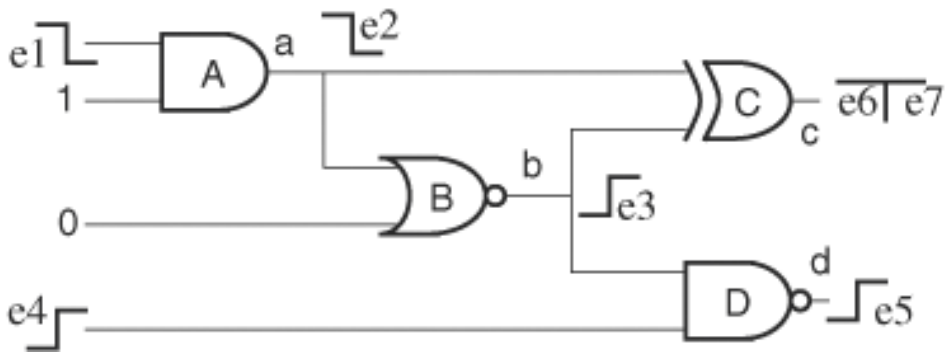
```
always @(posedge clock)
  b <= c;
always @(posedge clock)
  a <= b;
```

- Use nonblocking assignments for modeling sequential logic.
  - The example evaluates correctly if the two assignments are made nonblocking, irrespective of the order of evaluation.
  - This is because nonblocking assignments are evaluated last and cannot influence each other.

Rules to Avoid Races

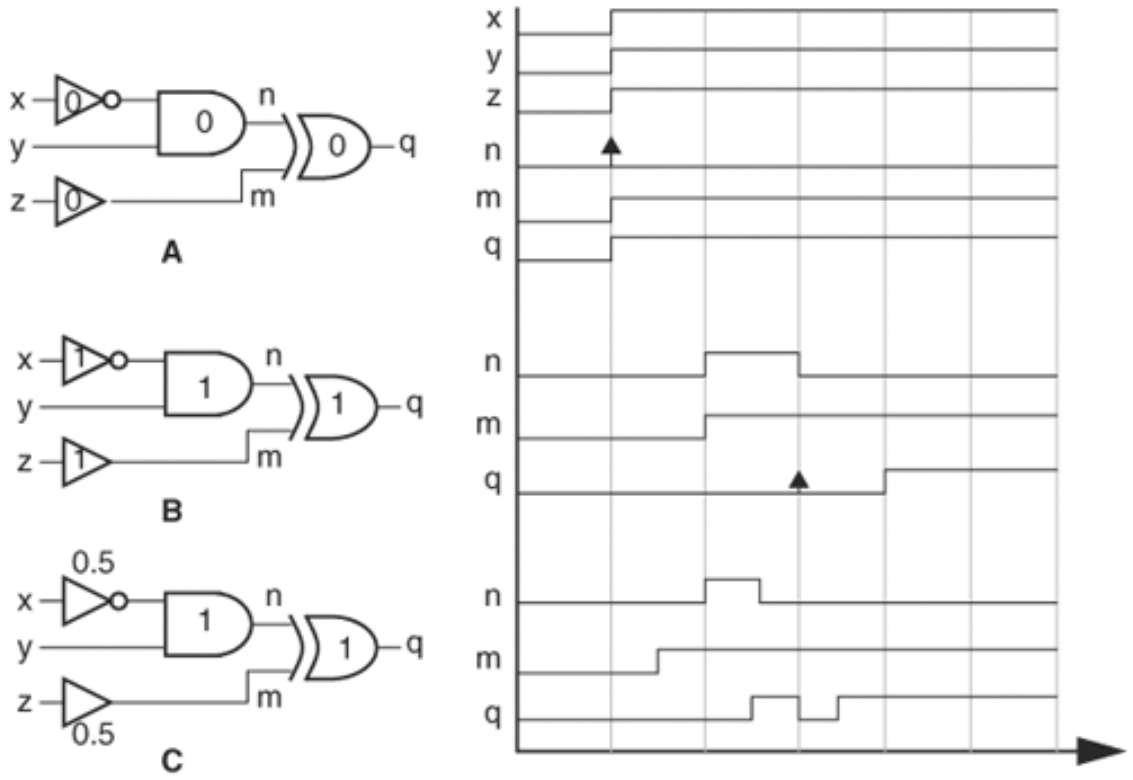
- Conversely, use blocking assignments to evaluate combinational logic.
  - Assignments made in combinational logic models are supposed to take immediate effect.
  - The use of nonblocking assignments would often be confusing (and sometimes wrong).

Event-Oriented Evaluation



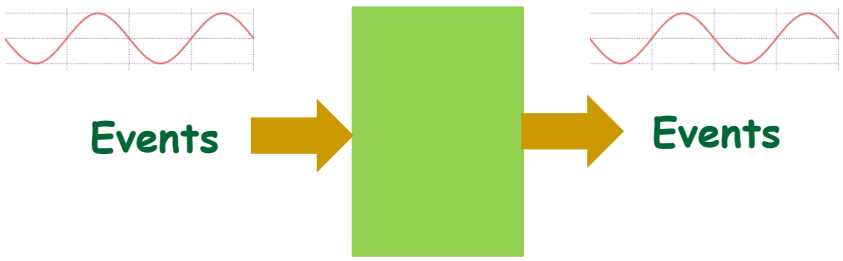
- Consider the circuit with zero delays.
- If the evaluation follows the events,
  - e1 → A → e2.
  - e2 → B → e3.
  - e2 → C → e6.
  - e3 → C → e7.
  - e3 → D → e5.
  - e4 → D → e5.
- There is a total of 6 evaluations.

Zero- versus Real-Delay Simulators



Dynamic Simulation for Timing

- Dynamic Timing Simulation
  - Thousands of test vectors are required to test all timing paths using logic simulation
  - Simulation complexity: 2^{2n}
  - inadequate for complex designs
- Advantages
  - Can be very accurate (spice-level)
- Disadvantages:
  - Analysis quality depends on stimulus vectors
  - Non-exhaustive, slow.
    - Examples: VCS, Spice, ACE



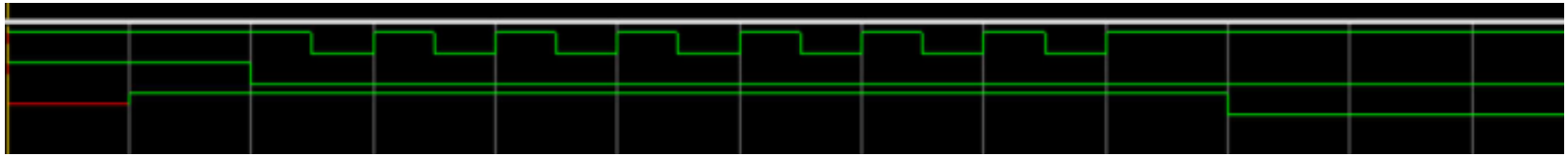
Events = ( ) → ( )

Inertial Delay

```
1.  `timescale 1ns / 1ns
2.  module inertial_delay ();
3.  logic a,b,y;
4.  int i;
5.  assign #10 y = a & b;  // 10ns gate delay
6.  initial begin
7.          a = 1;
8.          b = 1;  // Set inputs high and wait 20ns to set Y high
9.          #20;
10.         b = 0;  // Set B low, toggle A faster than AND gate delay
11.         for (i=0; i < 10; i++)
12.                 #5 a = ~a;
13.         #20;  // Allow A to settle, check Y after 10ns
14.         end
15. endmodule
```

Inertial Delay by Modelsim

```
1.  `timescale 1ns / 1ns
2.  module inertial_delay ();
3.  logic a,b,y;
4.  int i;
5.  assign #10 y = a & b;  // 10ns gate delay
6.  initial begin
7.          a = 1;
8.          b = 1;  // Set inputs high and wait 20ns to set Y high
9.          #20;
10.         b = 0;  // Set B low, toggle A faster than AND gate delay
11.         for (i=0; i < 10; i++)
12.                 #5 a = ~a;
13.         #20;  // Allow A to settle, check Y after 10ns
14.         end
15. endmodule
```



VCS Is Correct !

```
1.  `timescale 1ns / 1ns
2.  module inertial_delay ();
3.  logic a,b,y;
4.  int i;
5.  assign #10 y = a & b;  // 10ns gate delay
6.  initial begin
7.          a = 1;
8.          b = 1;  // Set inputs high and wait 20ns to set Y high
9.          #20;
10.         b = 0;  // Set B low, toggle A faster than AND gate delay
11.         for (i=0; i < 10; i++)
12.                 #5 a = ~a;
13.         #20;  // Allow A to settle, check Y after 10ns
14.         end
15. endmodule
```

VCS	Modelsim
t: 0, a=1, b=1, y=x	t: 0, a=1, b=1, y=x
t: 10, a=1, b=1, y=1	t: 10, a=1, b=1, y=1
t: 20, a=1, b=0, y=1	t: 20, a=1, b=0, y=1
t: 25, a=0, b=0, y=1	t: 25, a=0, b=0, y=1
t: 30, a=1, b=0, y=0	t: 30, a=1, b=0, y=1
t: 35, a=0, b=0, y=0	t: 35, a=0, b=0, y=1
t: 40, a=1, b=0, y=0	t: 40, a=1, b=0, y=1
t: 45, a=0, b=0, y=0	t: 45, a=0, b=0, y=1
t: 50, a=1, b=0, y=0	t: 50, a=1, b=0, y=1
t: 55, a=0, b=0, y=0	t: 55, a=0, b=0, y=1
t: 60, a=1, b=0, y=0	t: 60, a=1, b=0, y=1
t: 65, a=0, b=0, y=0	t: 65, a=0, b=0, y=1
t: 70, a=1, b=0, y=0	t: 70, a=1, b=0, y=1
	t: 80, a=1, b=0, y=0

References

- "Correct Methods For Adding Delays To Verilog Behavioral Models" Clifford E. Cummings, HDLCON 1999.
- Hardware Design Verification. William K. Lam, Prentice Hall; 2005. ISBN: 0137010923.
- Principles of VLSI RTL Design: A Practical Guide by Sanjay Churiwala & Sapan Garg, Springer, 2011
- Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill! By Cliff Cummings and Sunburst Design
- "Digital System Design with SystemVerilog" by Mark Zwolinski. Prentice Hall, 2009.