Event-Driven Simulation

Simulation is used to verify the design.

CAD tools incorporate logic-level simulation to reduce simulation complexity and time.

In Verilog, signals are represented as 0 and 1.

However, in reality, signals can be any value inbetween, so Verilog adds *x* and *z* to handle cases where the signal value is ambiguous.

Signal *contention* and *open circuits* can introduce ambiguity.

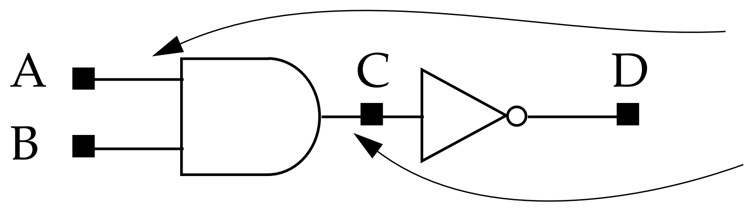
Event driven simulation exploits the fact that most signals are quiescent at any given point in time.

In event driven simulation, no computational effort is expended on quiescent signals, i.e., their values are not recomputed at each time step.

Rather, the simulator waits for an *event* to occur, i.e., for a signal to undergo a change in value, and ONLY the values of those signals are recomputed.

Event-Driven Simulation

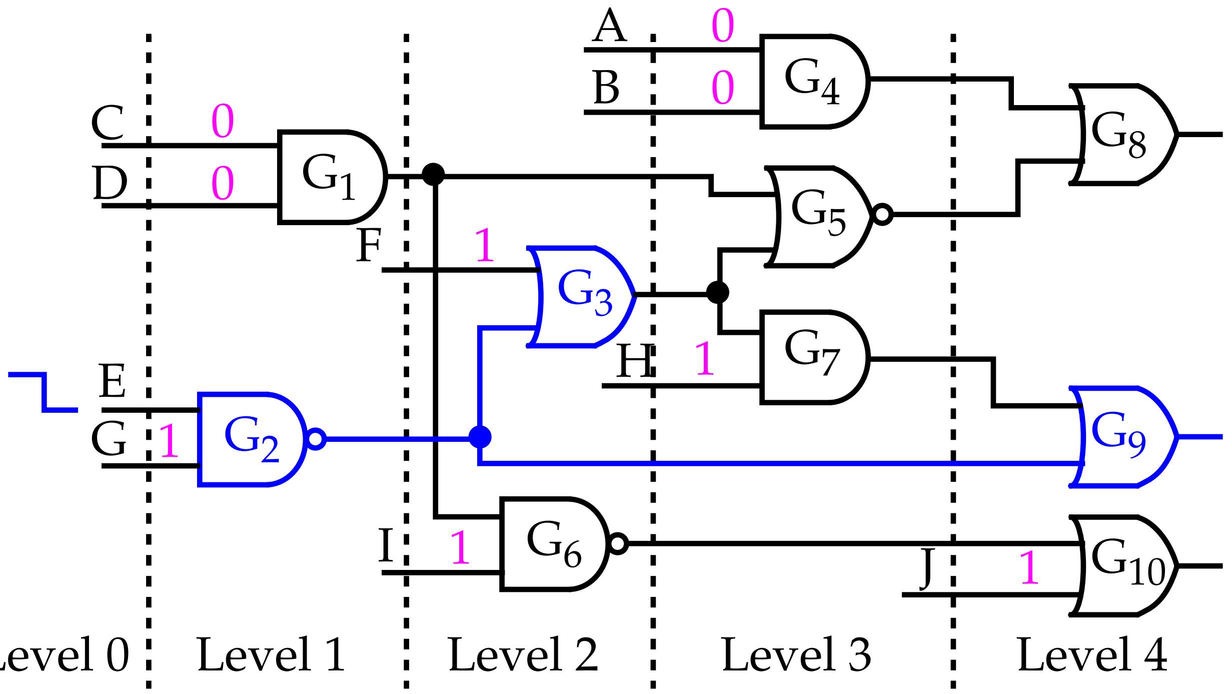
For example, the simulator monitors input *A* and *B* of the AND gate.



If a change to *A* occurs, the AND gate is scheduled for execution.

If its output changes, then an *event* is scheduled for *C*, and so on

The falling transition on *E* causes the highlighted gates to be evaluated.



Event-Driven Simulation

The simulator creates and manages an ordered list of *event-times*, times at which events have been scheduled to occur.

An *event queue* is associated with each *event time*, that contains the names and new values of the signals that are about to change.

The *event time* that is assigned depends on the timing model.

A functional model excludes timing, i.e., no delay is modeled and therefore all updates for a given input signal change are immediately reflected at other nodes.

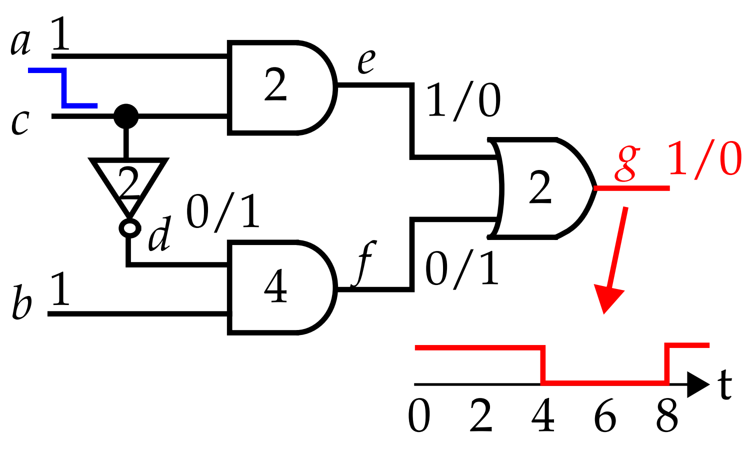
Unit delay simulators give information on the *evolution* of events, but are not able to portray the actual timing behavior of the design.

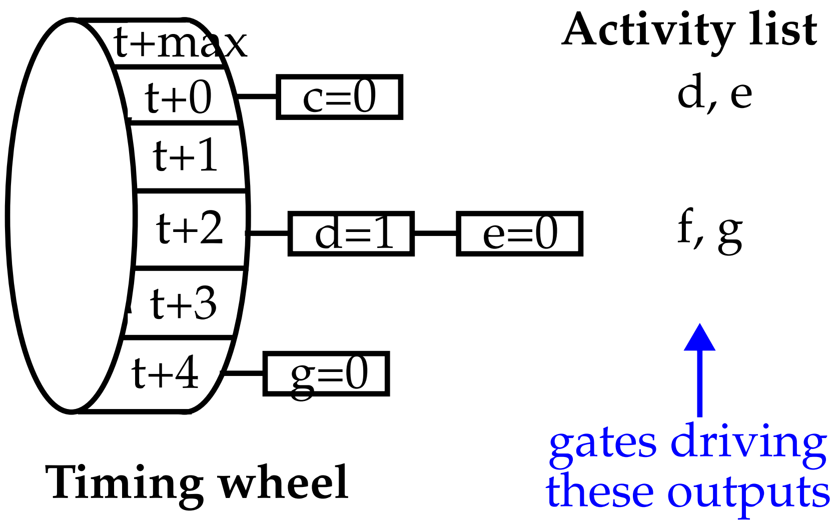
Verilog enables a more accurate *timing* model by way of delay statements.

For example, delay can be associated with Verilog *primitives* and *continuous assignment* stmts, which the simulator uses to schedule events.

Event-Driven Simulation

Gates whose inputs change go on the *activity list*.





Simulation involves evaluating a gate on the activity list.

If output changes, then gates at fanout added to *activity list*.

Gate delay models the time it takes to charge or discharge the output node, often referred to as the ***inertial*** delay of the gate.

If an input changes value and then changes back again in time less than the inertial delay, the output of the gate does NOT change.

Delay Control Operator

The delay control operator suspends the activity flow within a behavior by postponing the execution of a procedural statement.

If the '# *delay_value*' proceeds an assignment statement, the actual assignment is delayed until **after** the specified time elapses.

```
initial                              //At t_sim = 0, IN3, IN4 and IN5 are 'x'
  begin
    #0 IN1 = 0; IN2 = 1;             // Executes at t_sim = 0
    #100 IN3 = 1;                    // Executes at t_sim = 100
    #100 IN4 = 1, IN5 = 1;           // Executes at t_sim = 200
    #400 IN5 = 0;                    // Executes at t_sim = 600
  end
```

This construct is referred to as a *blocking* delay.

All stmts following a *blocked* stmt are also suspended.

This works fine for creating waveforms in test benches (to be discussed) but care must be taken when using blocking delays to model propagation delay.

Delay Control Operator

The #5 appearing before the assignment delays BOTH the sampling of *a* and *b* and the assignment to *y*:

```
module bit_or8_gate4(y, a, b)
   input [7:0] a, b;
   output [7:0] y;
   reg [7:0] y;

   always @(a or b) begin
      #5 y = a | b;
   end
endmodule
```

Therefore, *y* gets old data, i.e., the values of *a* and *b* 5 time units after the activating event.

Second problem: the event control expression cannot respond to events on *a* and *b* because the simulator remains at #5 *y* = *a* | *b*.

intra-assignment delay can be used to deal with the first problem.

Here, the timing control is placed on the righthand side (in RHS) in an assignment stmt.

Delay Control Operator

In the following example, the RHS is evaluated **immediately** but the assignment doesn't take place until the designated time has ellapsed.

```
module bit_or8_gate4(y, a, b)
  input [7:0] a, b;
  output [7:0] y;
  reg [7:0] y;

  always @(a or b) begin
    y = #5 a | b;
  end
endmodule
```

In particular, *a* and *b* are sampled but *y* is not assigned a new value for 5 more time units.

This separates referencing and evaluation from the actual assignment.

However, the second problem remains.

i.e., the assignment is blocking, preventing further events occurring on *a* and *b* to be missed.

Delay Control Operator

The solution is to use a *non-blocking* assignment with *intra-assignment* delay.

```
module bit_or8_gate4(y, a, b)
   input [7:0] a, b;
   output [7:0] y;
   reg [7:0] y;

   always @(a or b) begin
      y <= #5 a | b;
   end
endmodule
```

The '<=' indicates a non-blocking assignment.

If *a* or *b* change, they are sampled immediately but the actual assignment to *y* is delayed for 5 time units (as before).

Control then passes back to the **always** statement in the same simulation time step to allow *a* and *b* to be monitored again for change.

Further changes, even if they occur before the 5 time units have passed, cause the assignment to re-execute, scheduling future assignments to *y*.

This is the correct way to model the behavior of the actual hardware.

Delay Control Operator

Non-blocking assignments enable concurrency, like that present in the actual hardware -- as the following code fragments illustrate.

```
module nb1;
reg a, b, c, d, e, f;
// blocking assignments
initial
   begin
      a = #10 1;
      b = #2 0;
      c = #3 1;
   end

// non-blocking assignments
initial
   begin
      d <= #10 1;
      e <= #2 0;
      f <= #3 1;
   end
endmodule
```

t	a	b	c	d	e	f
0	x	x	x	x	x	x
2	x	x	x	x	0	x
3	x	x	x	x	0	1
10	1	x	x	1	0	1
12	1	0	x	1	0	1
15	1	0	1	1	0	1

On the right, all assignments are evaluated concurrently and scheduled.

Test Benches

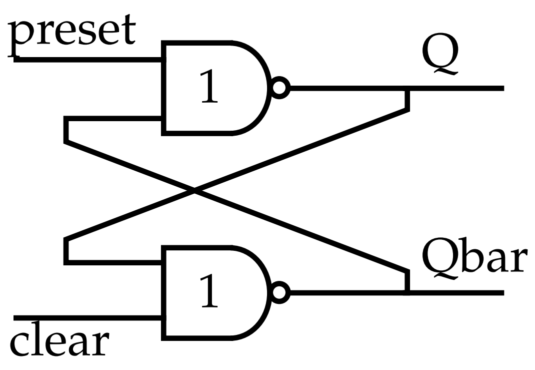
A test bench separates the design of the module from its testing module.

The test bench contains an instantiation of the *unit-under-test* (UUT) and Verilog behaviors that:

- Generate the input waveforms that are applied to the UUT (*stimulus generator*).
- Monitor the response of the UUT.
- Compare responses with those that are expected and issue messages.

The *stimulus generator* consists of a set of procedural statements that assign value to register variables to create wfms on the input ports of the UUT.

```
module nand_latch(Q, Qbar, preset, clear)
    output Q, Qbar;
    input preset, clear;

    nand 1 G1(Q, preset, Qbar),
           G2(Qbar, clear, Q);
endmodule
```



Test Benches

Variables *ps* and *clr* are declared as **reg** since they will be assigned value.

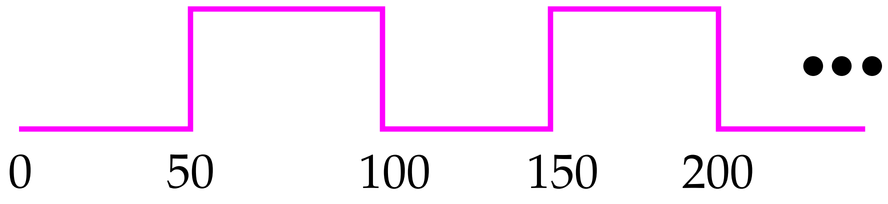
```
module test_nand_latch;                  // TEST BENCH for nand_latch
   reg ps, clr
   wire q, qbar;

   nand_latch M1 (q, qbar, ps, clr);   // Instantiate a copy of nand_latch

   initial begin
      $monitor ($time, "ps = %b clr = %b q = %b qbar = %b", ps, clr, q, qbar);
   end                              // Only one $monitor task is in effect at a time --
   initial begin                    //  subsequent calls overwrite the format string.
      #10 ps = 0; clr = 1;
      #10 ps = 1; $stop;            // Stop and allow user to interact with simulator.
      #10 clr = 0;                  // Type "." to proceed.
      #10 clr = 1;
      #10 ps = 0;
   end

   initial
      #60 $finish;     // Return control to OS
endmodule
```

Verilog provides special variables, e.g., **$monitor**, **$time**, **$stop** to support simulation.

Test Benches

initial and **always** are used to initialize variables and to define wfms.

```
module simple_clock_gen (clock)
   output clock;
   reg clock;

   parameter half_cycle = 50;
   parameter max_time = 1000;

   initial   // Assign at t_sim = 0
      clock = 0;

   always
      begin
         #half_cycle clock = ~clock;
      end

   initial
      #max_time $finish;

endmodule
```



The #*half_cycle* introduces 50 units of delay.

The simulation finishes after 10 clock cycles.

Test Benches: Waveform Generation

Using non-blocking assignments with intra-assignment delay to create a schedule of assignments to a target register variable.

```
module multiple_no_block_1;
   reg wave;
   reg [2:0] i;

   initial begin
      for (i = 0; i <= 5; i = i+1)
         wave <= #(i*10) i[0];
      end
endmodule
```

All RHS are sampled at the same time, but the value depends on *i* (the for loop executes in 0 time).

```
module multiple_no_block_2;
   reg wave1, wave2;

   initial begin
      #5 wave1 = 0;
         wave2 = 0;
         wave1 <= #5 1;
         wave2 <= #10 1;
         wave2 <= #20 0;
     #10 wave1 = 1;
         wave1 <= #5 0;
      end
endmodule
```

Note that the reference to *i[0]* refers to the low order bit of the counter variable, *i*, used in the **for** loop.

Draw the waveforms associated with these modules.

Test Benches: Waveform Generation

Here, a waveform pair is generated from a *reference* signal, *sig_c*.

```
module non_block(sig_a, sig_b, sig_c);
   reg sig_a, sig_b, sig_c;
   initial
      begin
         sig_a = 0;                          // Non-overlapping wfms generated
         sig_b = 1;                          // (sig_a and sig_b) from a clock signal
         sig_c = 0;                          // sig_c
      end
   always sig_c = #5 ~sig_c;
   always @ (posedge sig_c)
      begin
         sig_a <= sig_b;
         sig_b <= sig_a;
      end
   endmodule
```

You should be able to draw the wfms from these descriptions.

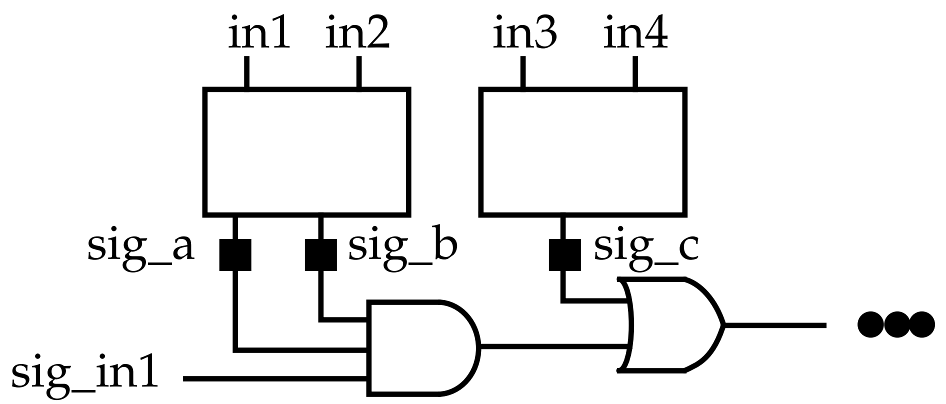
MISC: force ... release

force ... **release** is a special form of *procedural continuous assignment* that can be used to control registers or nets during simulation.

This construct is **not** accepted by synthesis engines.

For example:

```
force sig_a = 1;
force sig_b = 1;
force sig_c = 0;
sig_in1 = 0;
#5 sig_in1 = 1;
#5 sig_in1 = 0;
//  other code
release sig_a;
release sig_b;
release sig_c;
```



A test to sensitive a path

When **force** is applied to a net, it remains in effect until a **release** is executed.

The **force ... release** stmt overrides other assignments, e.g., from a primitive, a continuous assignment, etc.

MISC: wait

The **wait** statement models *level-sensitive* behavior by suspending activity flow until the expression becomes TRUE.

If true when evaluated, no suspension occurs.

If false, the simulator suspends the activity thread and sets up a monitor.

For example:

```
module wait_example(...)
   ...
   always
      begin
         ...
         wait (enable) register_a = register_b;
         #10 register_c = register_d;
      end
endmodule
```