HDL simulation is different from C execution

- Need to express concurrency (things happening simultaneously)
- Need to model time
- Need to model non-standard wordlengths
- Need to model non-standard values (X,Z)
- Need organization in modules rather than functions

HDL simulation is different from C execution

- Need to express concurrency (things happening simultaneously)
- Need to model time

  Simulation Time + Concurrency Model (events, cycles, ..)

- Need to model non-standard wordlengths
- Need to model non-standard values (X,Z)
- Need organization in modules rather than functions

  New data types, custom syntax

Focus of this lecture

- Need to express concurrency (things happening simultaneously)
- Need to model time

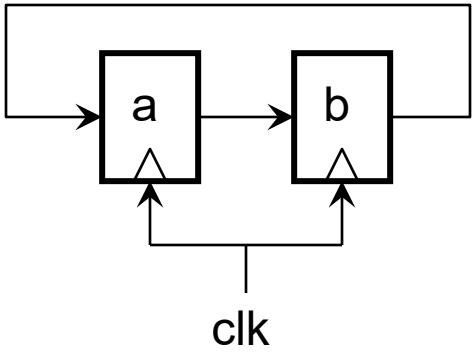
Simulation Time + Concurrency Model (events, cycles, ..)

Objectives:

- Understand the concept of event driven simulation
- Understand how gate-level models are simulated
- Understand how behavioral models are simulated

The need for concurrent hardware models

Let's write a simulation for this circuit in C

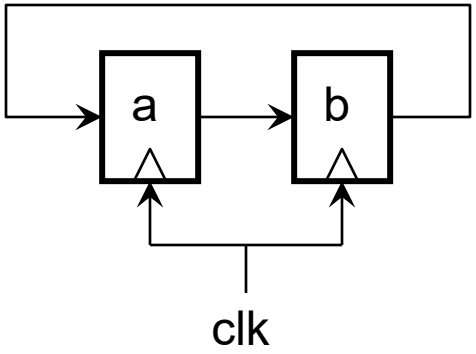


We create a function clock_tick which will be called for each clock cycle. Is the following a correct implementation of this function?

```
int a, b;
void clock_cycle() {
  a = b;
  b = a;
}
```

The need for concurrent hardware models

Let's write a simulation for this circuit in C



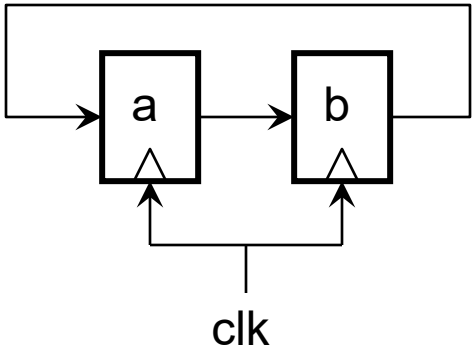
We create a function clock_tick which will be called for each clock cycle. Is the following a correct implementation of this function?

```
int a, b;
void clock_cycle() {
  a = b;
  b = a;
}
```

No! a and b will have the same value after one call

A better solution

Let's write a simulation for this circuit in C

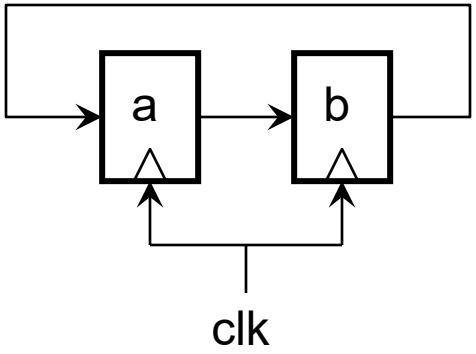


Is this better?

```
int a, b;
int new_a, new_b;
void clock_cycle() {
  new_a = b;
  new_b = a;
  b = new_b;
  a = new_a;
}
```

A better solution

Let's write a simulation for this circuit in C



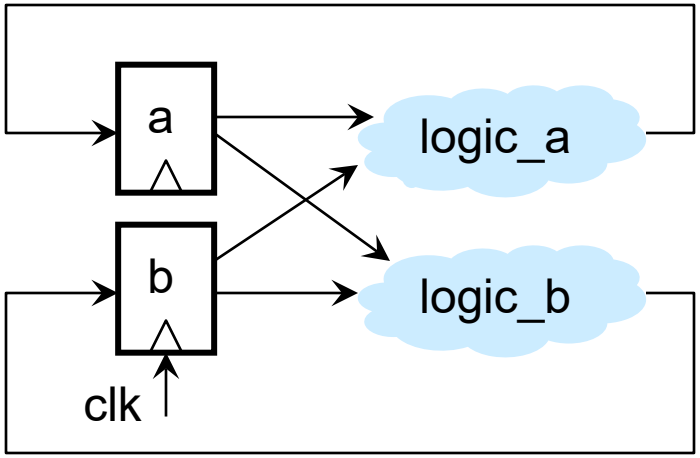

```
int a, b;
int new_a, new_b;
void clock_cycle() {

   // evaluate new inputs
   new_a = b;
   new_b = a;

   // update outputs
   b = new_b;
   a = new_a;
}
```

This is better.

new_a, new_b variables account for the fact that registers have two values: the previous value and the current value (i.e. the input and the output of a register can be different values)

Cycle-based simulation

Thus, we simulate **concurrency** by making the simulation progress in two phases, which alternate at the pace of the clk signal



logic_a and logic_b appear to execute concurrently

1. Evaluate

Find new values at the inputs of a and b

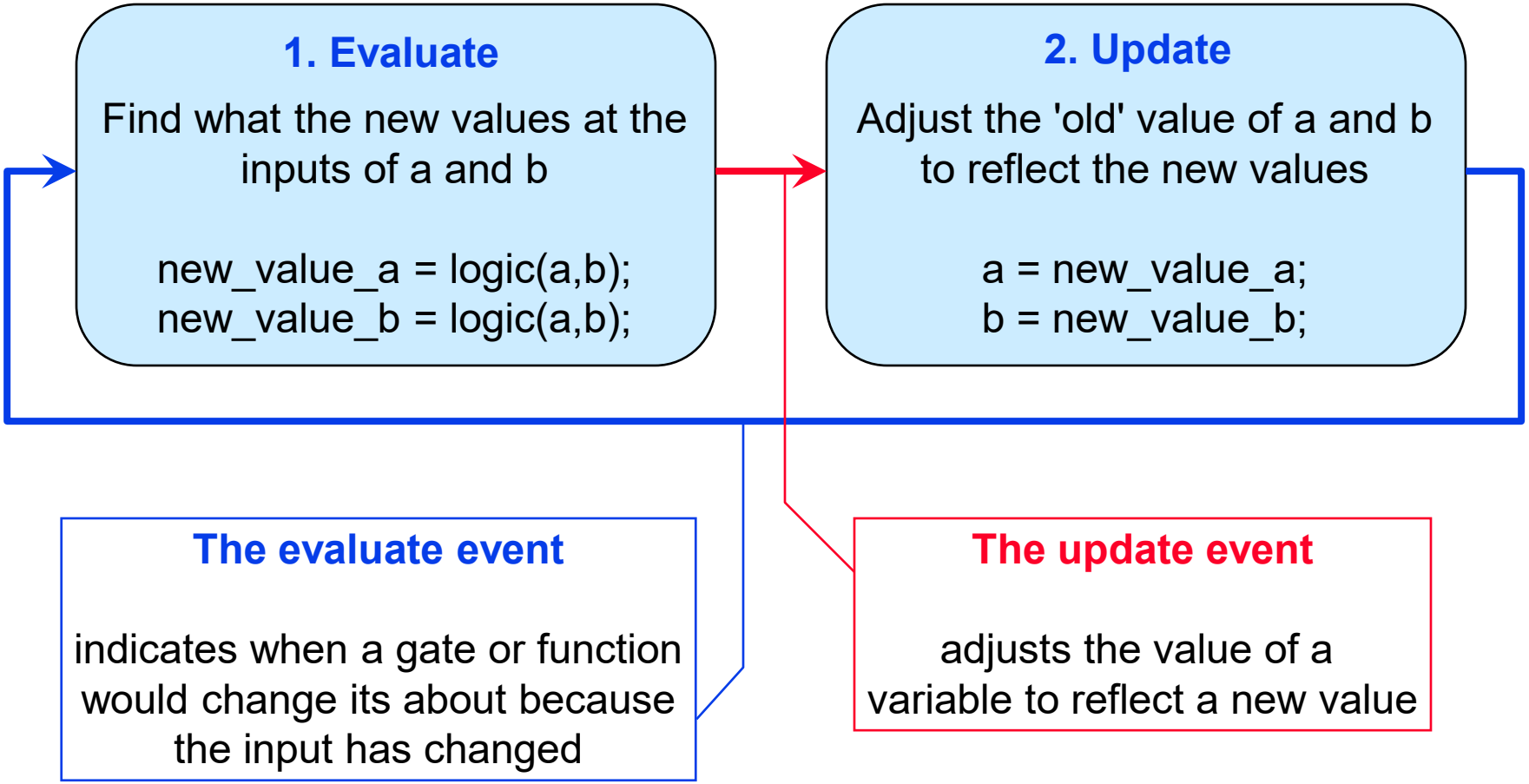
```
new_value_a = logic(a,b);
new_value_b = logic(a,b);
```

2. Update

Adjust the 'output' value of a and b to reflect the new values

```
a = new_value_a;
b = new_value_b;
```

Transition between phases driven by *events*



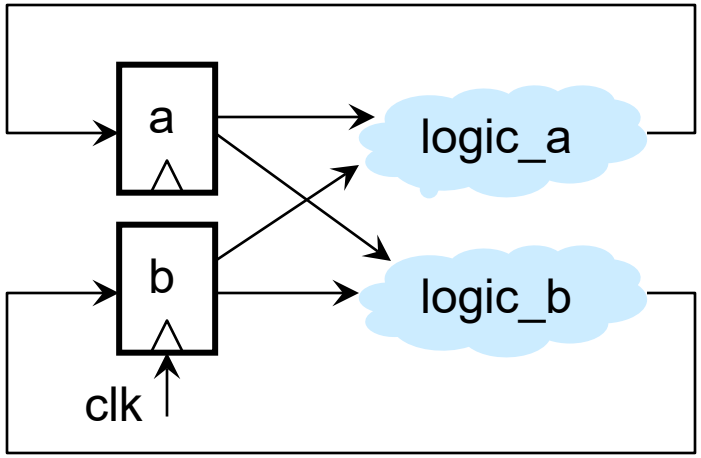
Cycle-based simulation has two types of *events*

The evaluate event

indicates when a gate or function would change its output because the input has changed

The update event

adjusts the value of a variable to reflect a new value



- **What is the cause of the evaluate event ?**

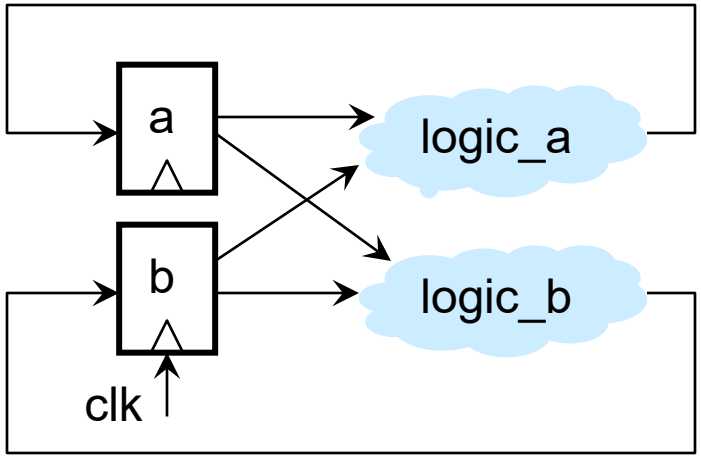
Cycle-based simulation has two types of *events*

The evaluate event

indicates when a gate or function would change its output because the input has changed

The update event

adjusts the value of a variable to reflect a new value



- **What is the cause of the evaluate event ?**

  The output of the registers (a, b) changing their value

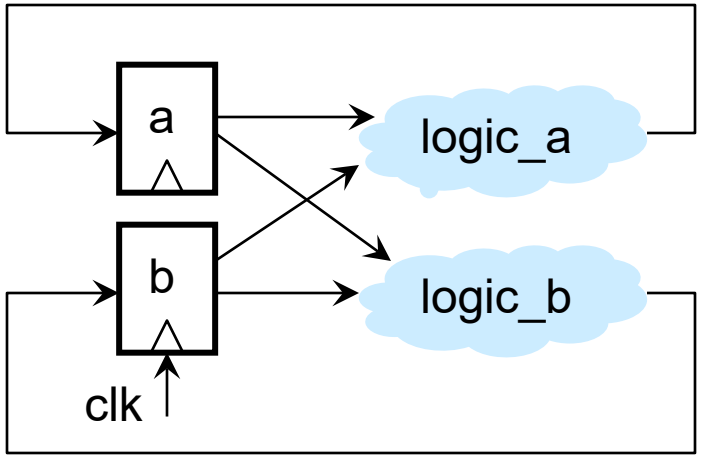
Transition between phases driven by *events*

The evaluate event

indicates when a gate or function would change its output because the input has changed

The update event

adjusts the value of a variable to reflect a new value



- **What is the cause of the update event ?**

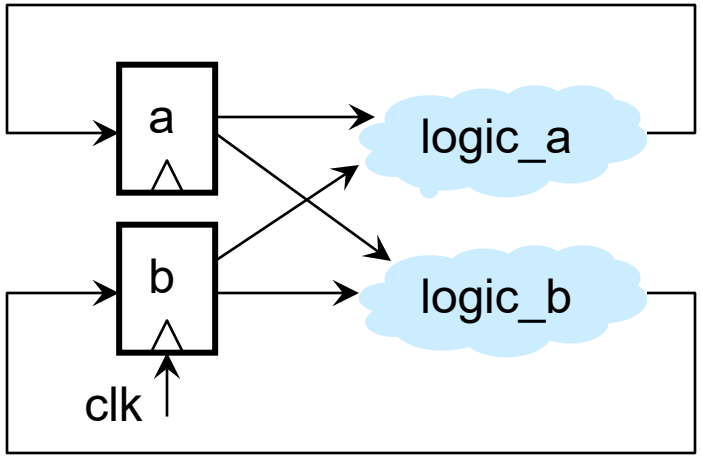
Cycle-based simulation has two types of *events*

The evaluate event

indicates when a gate or function would change its output because the input has changed

The update event

adjusts the value of a variable to reflect a new value

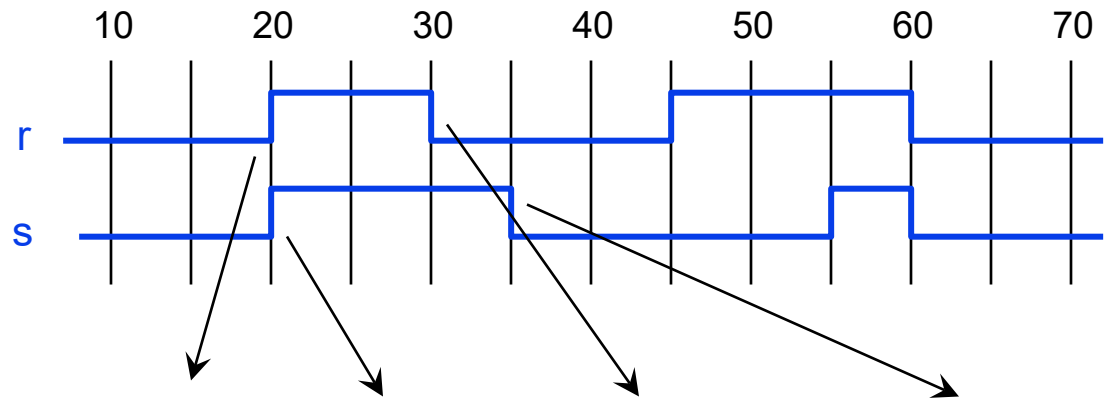


- **What is the cause of the update event ?**

The cause of the update event is the upgoing clock edge

But what exactly is an *event*?

- Event = tuple (activity, simulation time)

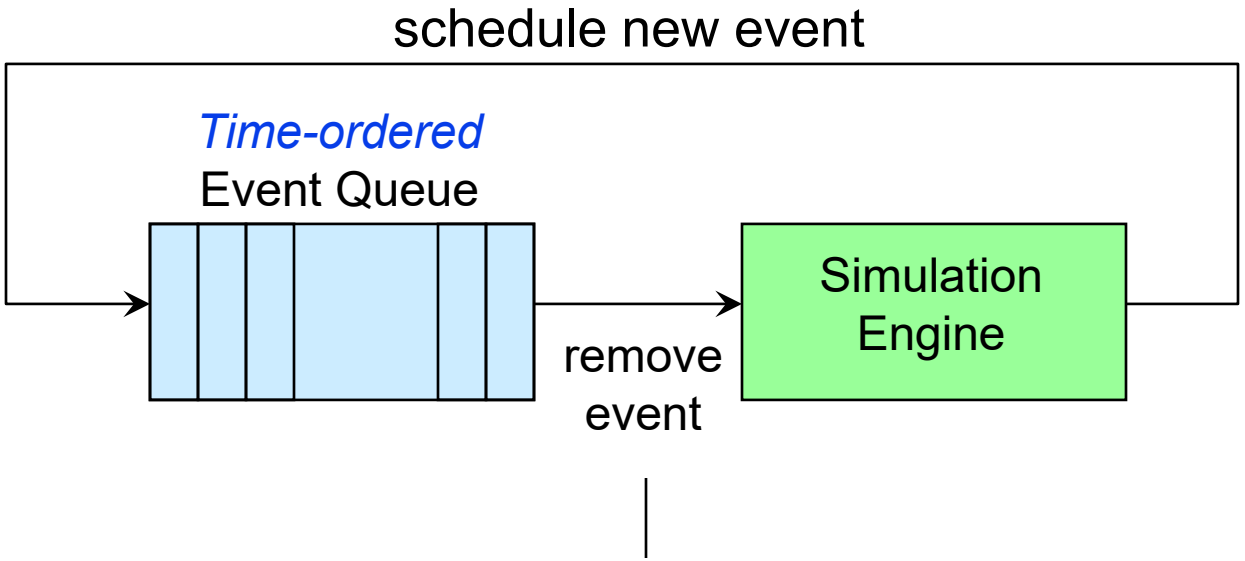


Events: (r_rises, 20), (s_rises, 20), (r_falls, 30), (s_falls, 35)

- A list of events expresses concurrent activities
  - E.g. (s_rises, 20) and (r_rises, 20) happen at the same time
  - **!!!** In fact, the simulator cannot tell which of (s_rises, 20) and (r_rises, 20) happens first.

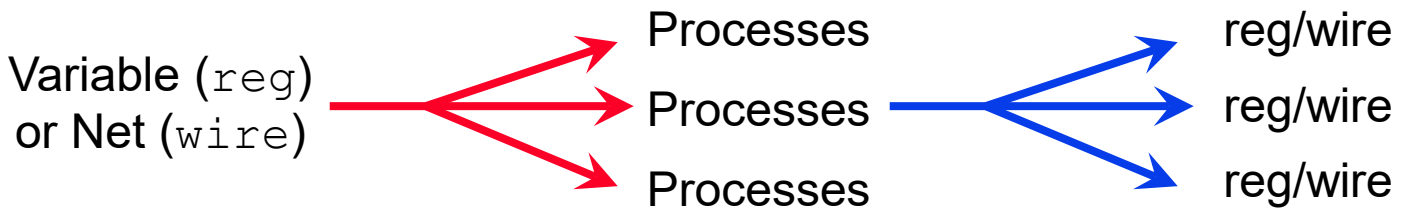
Verilog is an *event-driven* simulator

- The Verilog simulator maintains an ordered list of all future events of interest: the Event Queue
- The simulation engine reads events from the queue, executes them, and inserts new events as needed



Verilog is an *event-driven* simulator

- There are two types of events: evaluate events and update events



When a variable or a net changes its value, one or more processes can run (evaluate).

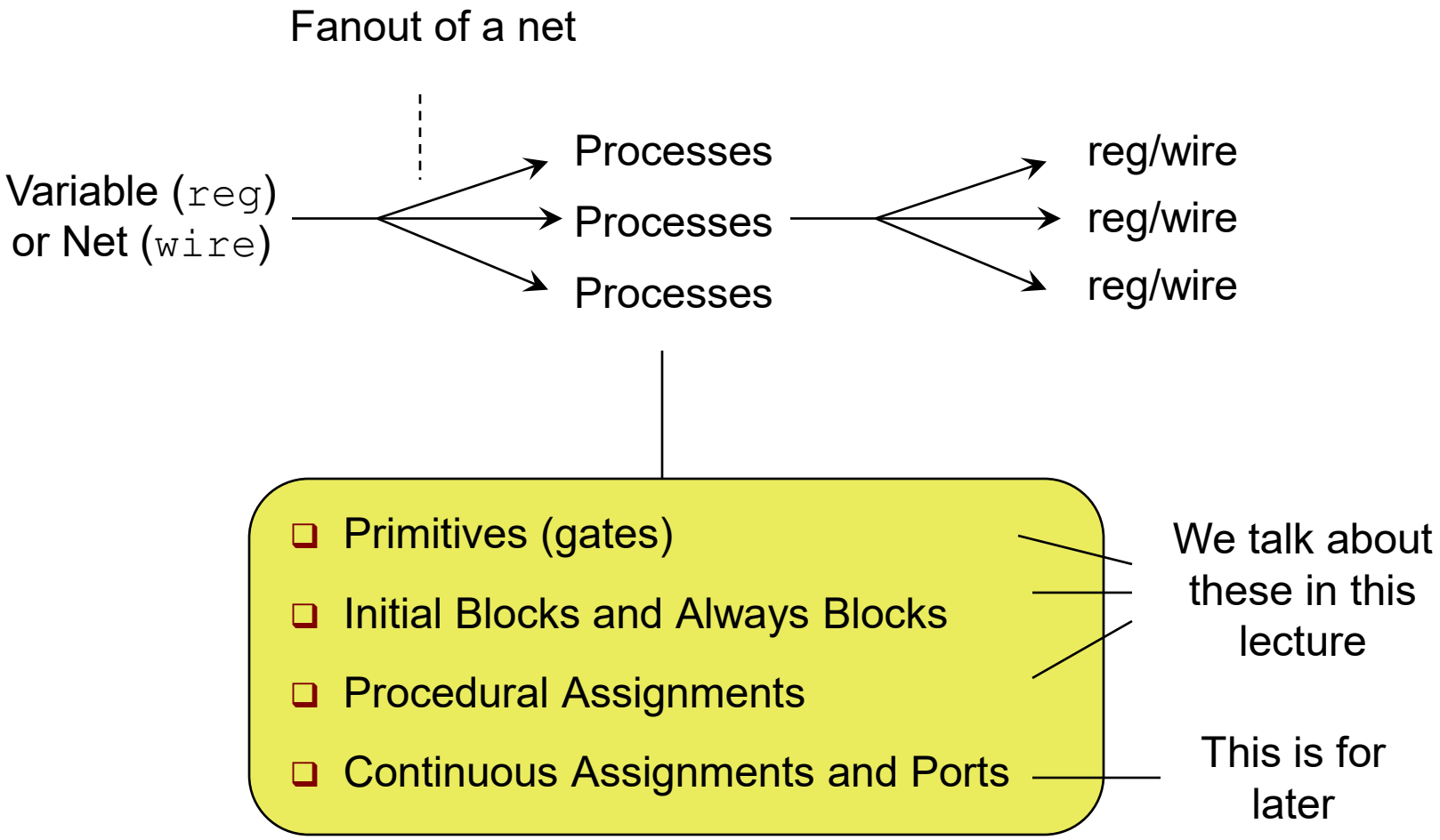
"**Evaluate**" is an event

When a process runs, it can update the value of one or more variables or nets.

"**Update**" is an event

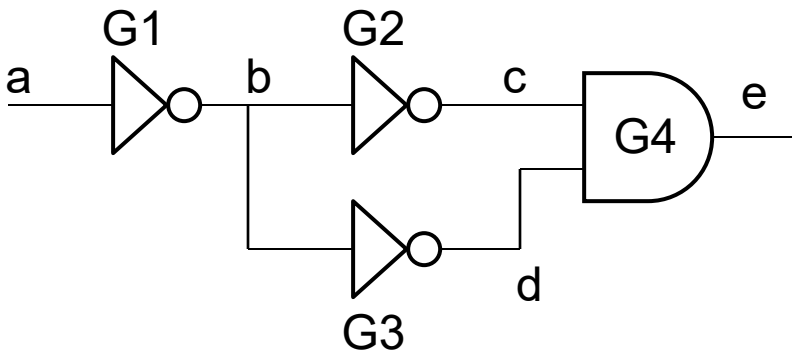
- We will just call them 'events'.

Some Verilog 'processes'



Let's see how gate-level simulation works

```
initial begin
  #20 a = 1;
end
```



All gates have propagation delay of 5 units

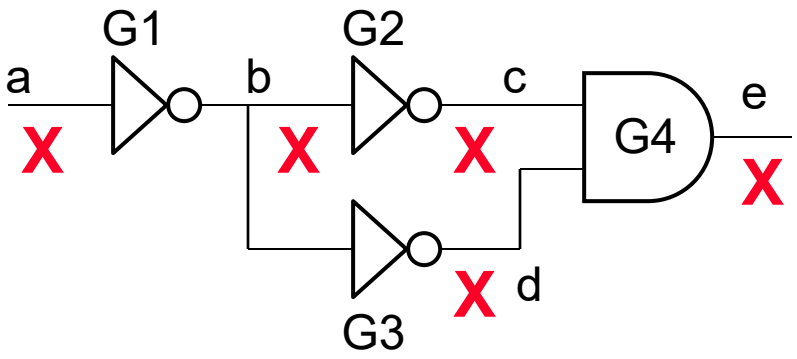
Event Queue

empty

T = 0
Initialize simulation
All nets are at X

Let's see how gate-level simulation works

T=0



All gates have propagation delay of 5 units

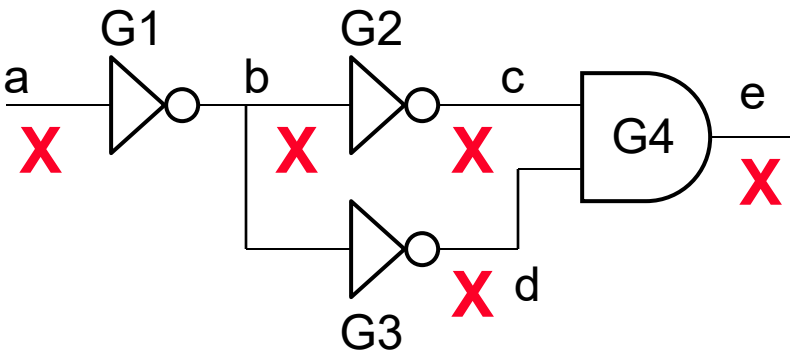
Event Queue

update
a = 1
@T = 20

T = 0
Initialize simulation
All nets are at X

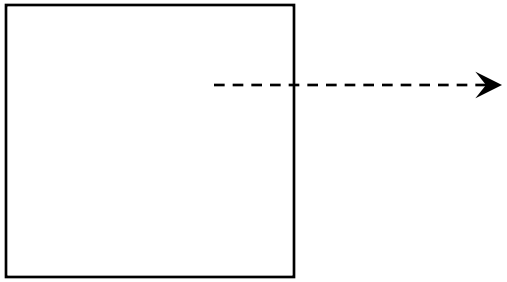
Let's see how gate-level simulation works

T=0



All gates have propagation delay of 5 units

Event Queue

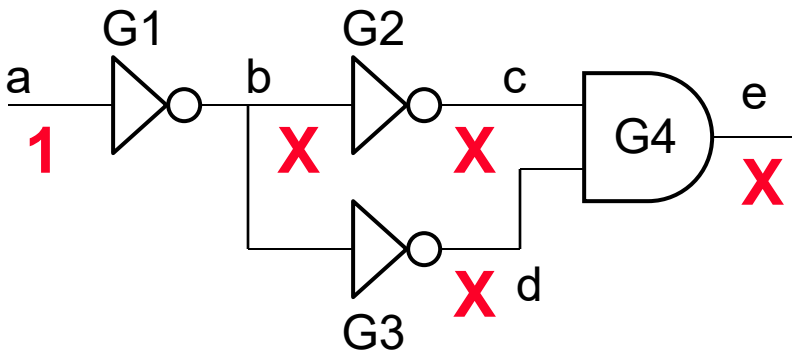


update
a = 1
@T = 20

Remove event from Q. Event T = 20
Adjust simulation time to T = 20
Update value of a
Fanout of a = {G1}
Execute G1. Output b should change to 0 at T = 20 + 5 = 25

Let's see how gate-level simulation works

T=20



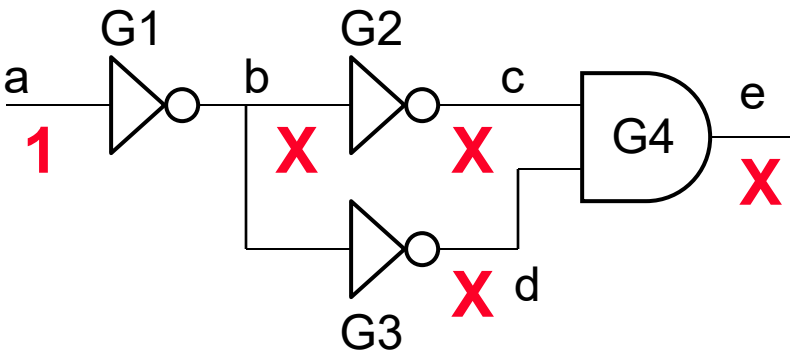
All gates have propagation delay of 5 units

Event Queue

update
b = 0
@T = 25

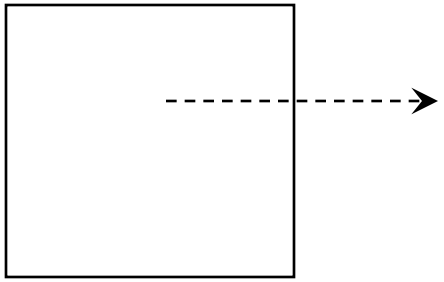
Let's see how gate-level simulation works

T=20



All gates have propagation delay of 5 units

Event Queue

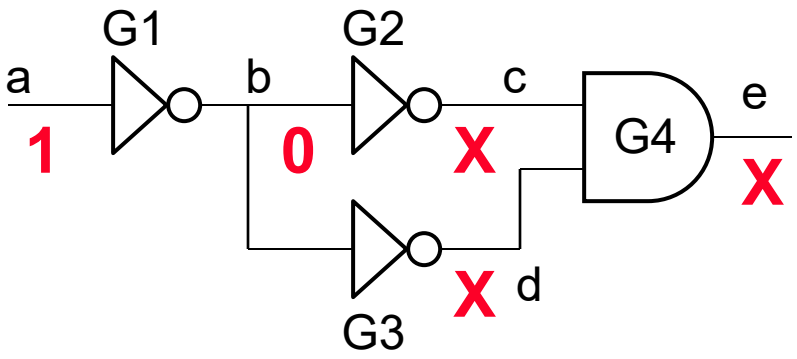


update
b = 0
@T = 25

Remove event from Q. Event T = 25
Adjust simulation time to T = 25
Update value of b
Fanout of b = {G2, G3}
Execute G2. c should change to 1 at T = 25 + 5 = 30
Execute G3. d should change to 1 at T = 25 + 5 = 30

Let's see how gate-level simulation works

T=25



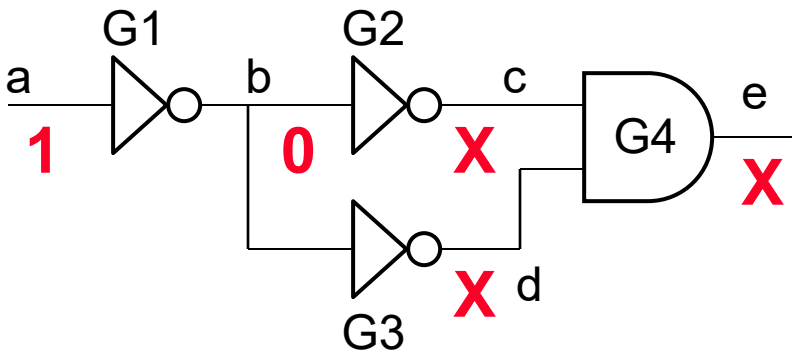
All gates have propagation delay of 5 units

Event Queue

update c = 1 @T = 30	update d = 1 @T = 30

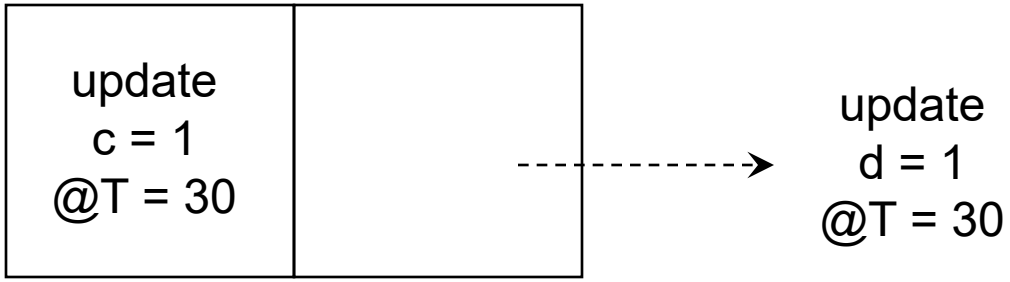
Let's see how gate-level simulation works

T=25



All gates have propagation delay of 5 units

Event Queue

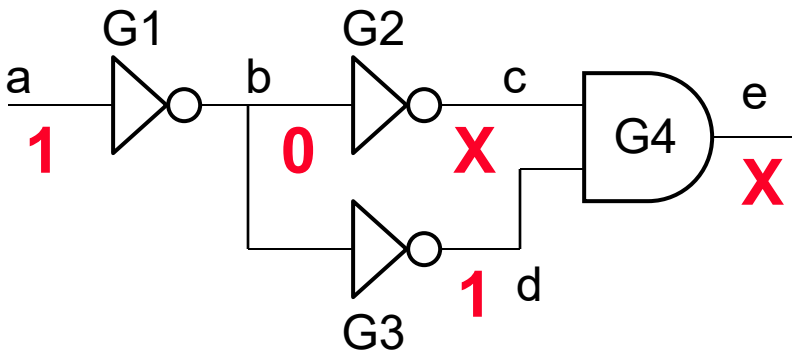


Note: It is equally valid to choose update-c event. Event processing order for equal timestamps is not guaranteed.

Remove event from Q. Event T = 30.
Adjust simulation time to T = 30
Update value of d
Fanout of d = {G4}
Execute G4. e will remain at X. No new event.

Let's see how gate-level simulation works

T=30



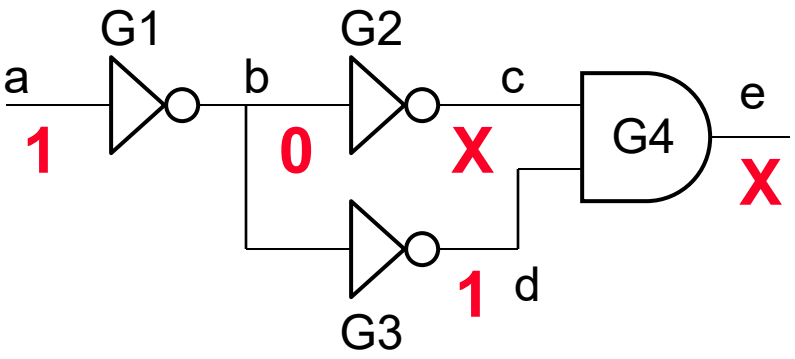
All gates have propagation delay of 5 units

Event Queue

update
c = 1
@T = 30

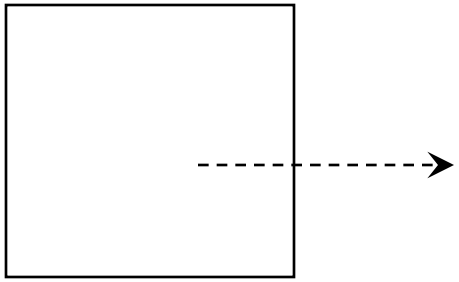
Let's see how gate-level simulation works

T=30



All gates have propagation delay of 5 units

Event Queue

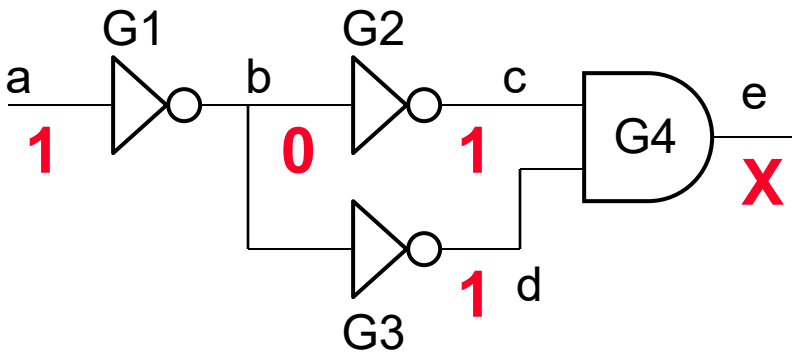


update
c = 1
@T = 30

Remove event from Q. Event T = 30.
Simulation time remains at 30.
Update value of c
Fanout of c = {G4}
Execute G4. e should change to 1 at T = 30 + 5 = 35.

Let's see how gate-level simulation works

T=30



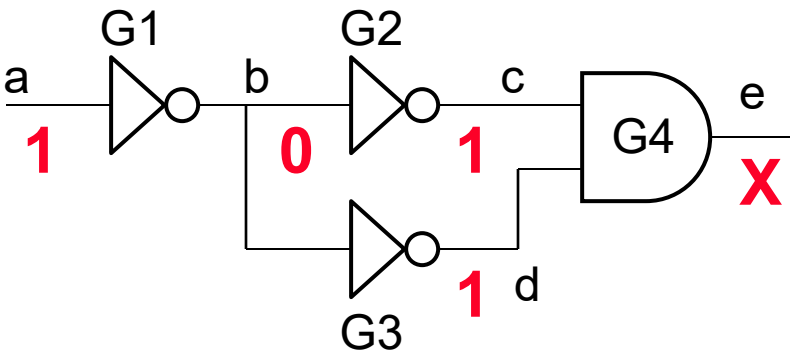
All gates have propagation delay of 5 units

Event Queue

update
e = 1
@T = 35

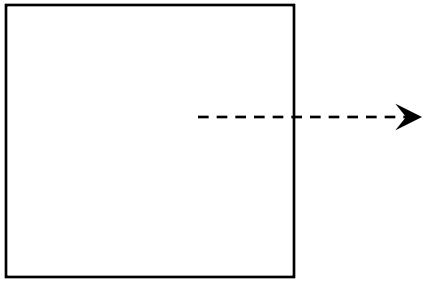
Let's see how gate-level simulation works

T=30



All gates have propagation delay of 5 units

Event Queue

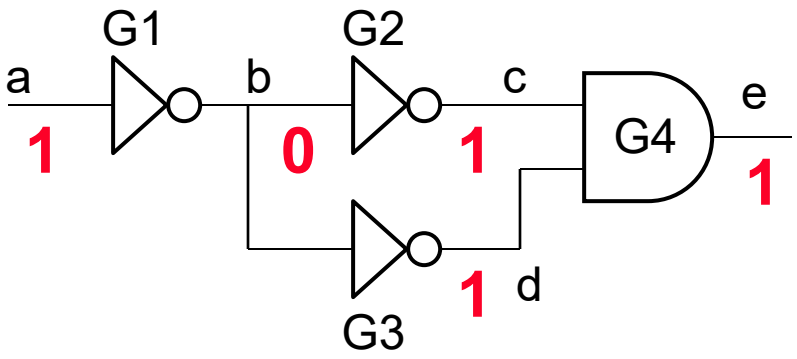


update
e = 1
@T = 35

Remove event from Q. Event T = 35.
Simulation time adjusts to 35.
Update value of e
Fanout of e = { }

Let's see how gate-level simulation works

T=35

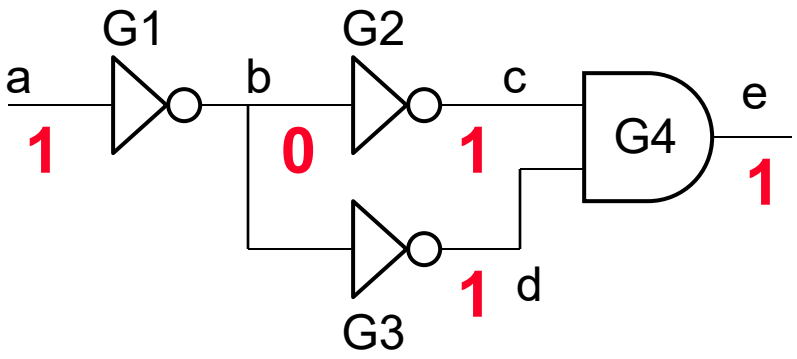


All gates have propagation delay of 5 units

Event Queue

No new events. Done.

Gate-level simulation



- Gate-level simulation works as follows
  - If a wire is updated, then all gates attached to the fan-out of that wire are executed.
  - *Gate Execution* really means: generate update event for the wire at the gate output

Event driven simulation in Behavioral Models

- We next discuss event modeling in always and initial blocks, and for procedural blocking assignments. E.g.:

```
module tata;
    reg a, b;
    initial begin
    b = 0;
    #100 a = 1;
    end
    always begin
        @(upedge a);
        b = a + 1;
        a = #20 b;
    end
endmodule
```

- A few constructs (procedural non-blocking assignments, continuous assignments, module ports) will be covered later.

Modeling time and events in behavioral code

- **#(number):**
  - Delay a specified units of time
- **@(expression):**
  - Wait for an event of that expression
- **wait(expression):**
  - Test the expression. If true, continue, else wait for update of that expression and test again.

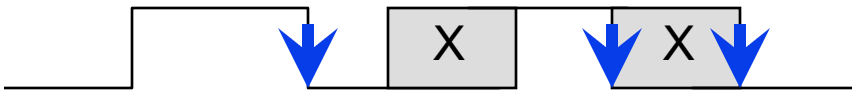
@(expression)

- Means: stop execution and wait for an update event for that expression

```
wire a;

always @(a)
begin
   ...
end
```


```
always @(posedge a)
begin
   ...
end
```


```
always @(negedge a)
begin
   ...
end
```



wait(expression)

- Wait for the *value* of expression to become TRUE.
- **wait** tests levels, @ tests edges.

```
reg a;

always
begin
   ...
   wait(a == 5);
   ...
end
```

```
test the value of a
if a !=5 then
   stop execution
   wait for update a
   repeat the test
else
   continue execution
```

while and wait

- Wait is *not* a busy loop - it really pauses the execution of behavioral code

```
reg[4:0] a, b;

initial a = 0;
initial b = 0;
always #10 a = a + 1;

always
begin
  wait(a == 5);
  #10 b = 1;
end
```

This will pause the bottom always and continue with the top always when a reaches 5, bottom always continues and sets b to 1

```
reg[4:0] a, b;

initial a = 0;
initial b = 0;
always #10 a = a + 1;

always
begin
  while (a != 5) b = 0;
  #10 b = 1;
end
```

This will never set b to 1; a while loop does not interrupt procedural flow. Once the bottom always block starts, it cannot exit

Simulating Behavioral Models with Events

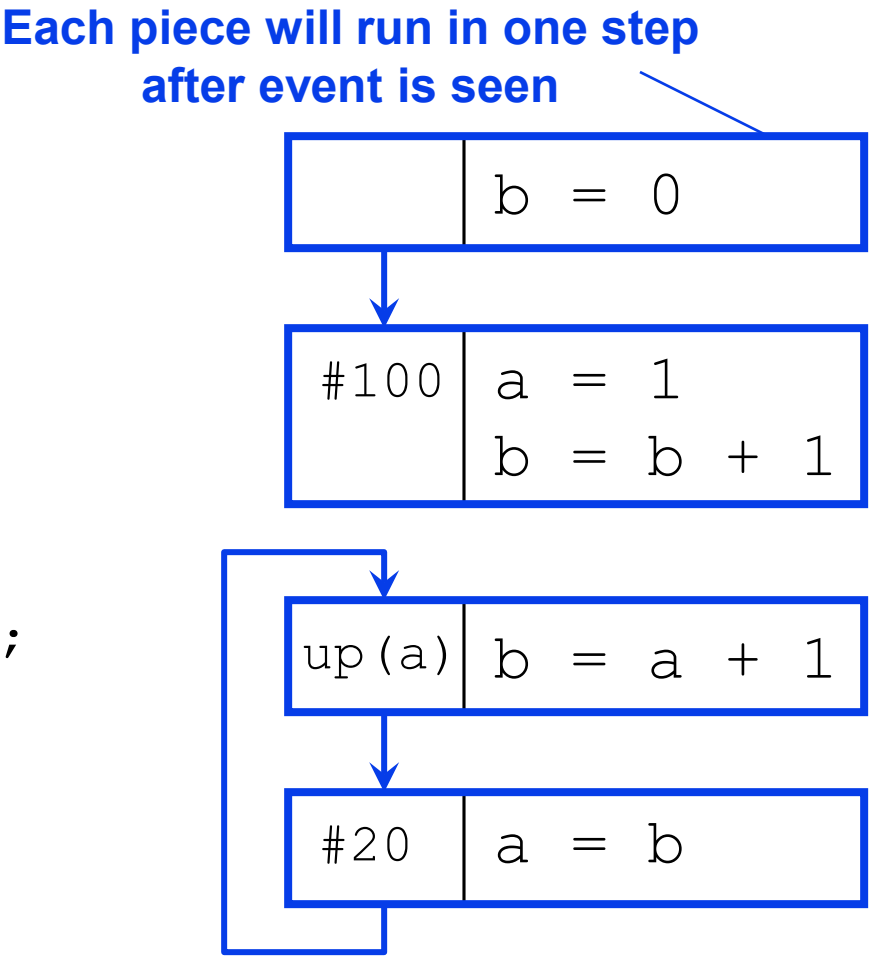
- In a block statement, we can control execution by using multiple #, @, wait

```
module tata;
    reg a, b;
    initial begin
    b = 0;
    #100 a = 1;
    b = b + 1;
    end
    always begin
        @(upedge a);
        b = a + 1;
        #20 a = b;
    end
endmodule
```

Simulating Behavioral Models with Events

- You can think of a single block statement as several pieces of code.

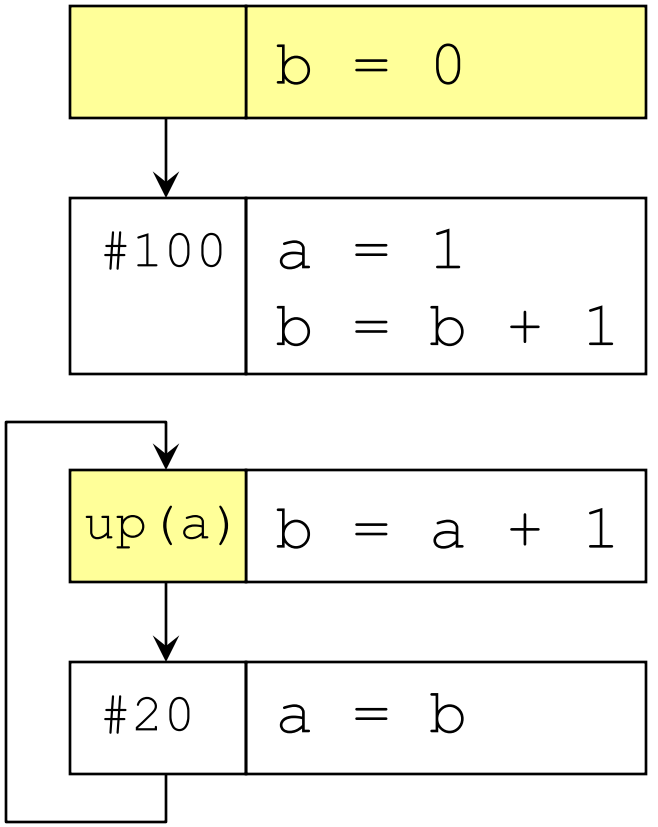
```
module tata;
    reg a, b;
    initial begin
    b = 0;
    #100 a = 1;
    b = b + 1;
    end
    always begin
        @(upedge a);
        b = a + 1;
        #20 a = b;
    end
endmodule
```



Simulating Behavioral Models with Events

- **T = 0**. initial block and always block start. b set to 0. always block waits for upedge of a.

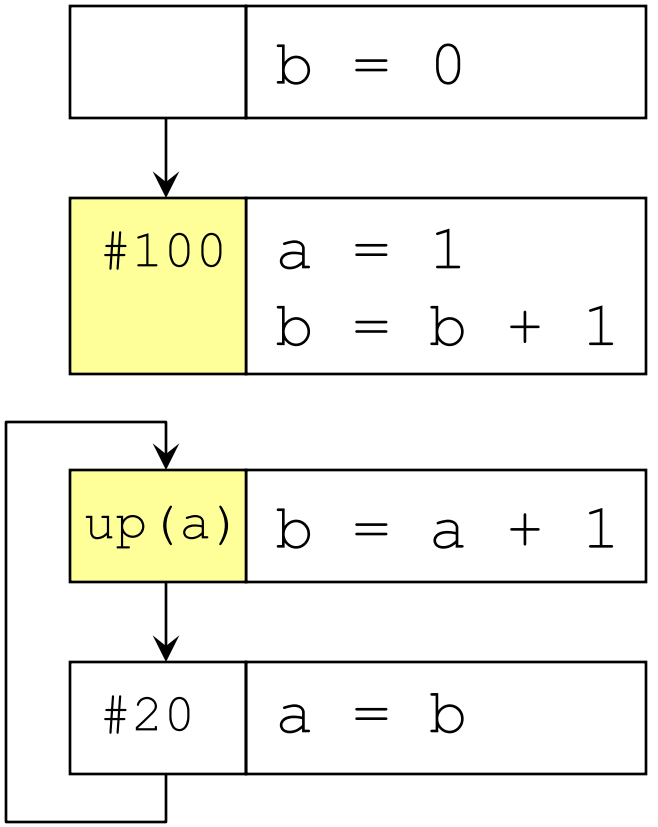
```
module tata;
    reg a, b;
    initial begin
    b = 0;
    #100 a = 1;
    b = b + 1;
    end
    always begin
        @(upedge a);
        b = a + 1;
        #20 a = b;
    end
endmodule
```



Simulating Behavioral Models with Events

- **T = 0**. #100 schedules evaluate of initial block at 100. always block still waits for upedge of a.

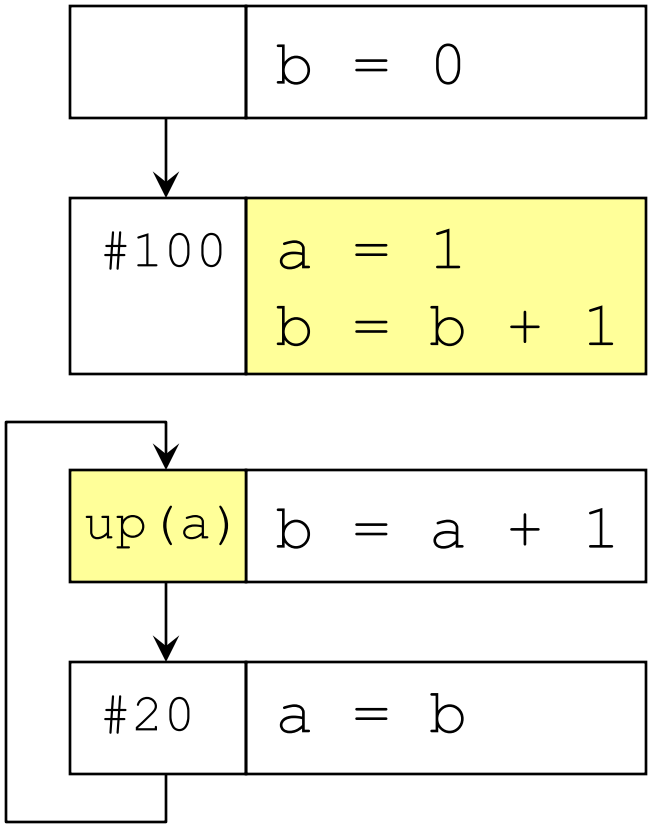
```
module tata;
    reg a, b;
    initial begin
    b = 0;
    #100 a = 1;
    b = b + 1;
    end
    always begin
        @(upedge a);
        b = a + 1;
        #20 a = b;
    end
endmodule
```



Simulating Behavioral Models with Events

- **T = 100**. Initial block updates a, b to 1 (update event of a,b at 100). Initial block terminates. Always block still waits for upedge of a.

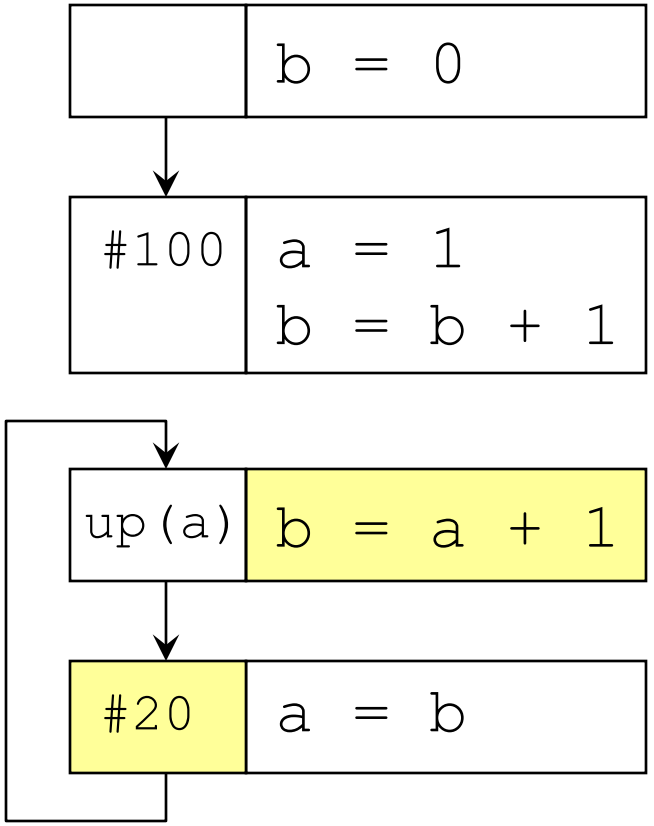
```
module tata;
     reg a, b;
     initial begin
     b = 0;
     #100 a = 1;
     b = b + 1;
     end
     always begin
          @(upedge a);
          b = a + 1;
          #20 a = b;
     end
endmodule
```



Simulating Behavioral Models with Events

- **T = 100**. Update event a triggers @(upedge a). b updated to 2. #20 schedules an evaluate of the always block at 120.

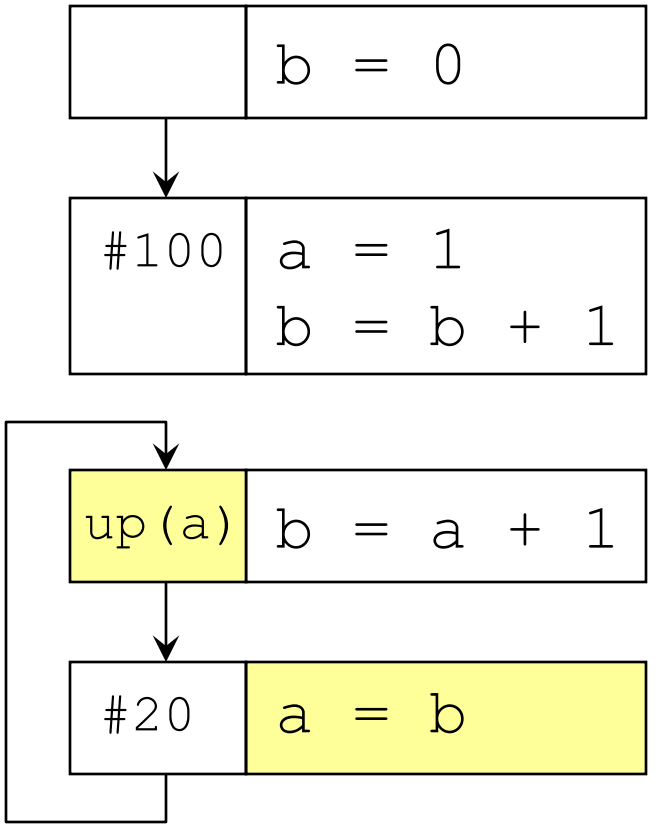
```
module tata;
    reg a, b;
    initial begin
    b = 0;
    #100 a = 1;
    b = b + 1;
    end
    always begin
        @(upedge a);
        b = a + 1;
        #20 a = b;
    end
endmodule
```



Simulating Behavioral Models with Events

- **T = 120**. Update a to 2. Wait for upedge of a (which will never come).

```
module tata;
    reg a, b;
    initial begin
    b = 0;
    #100 a = 1;
    b = b + 1;
    end
    always begin
        @(upedge a);
        b = a + 1;
        #20 a = b;
    end
endmodule
```



LHS and RHS Delay

```
module tata;
    reg a, b;
    initial begin
    b = 0;
    #100 a = 1;
    end
    always begin
        @(upedge a);
        b = a + 1;
        #20 a = b;
    end
endmodule
```

```
module tata;
    reg a, b;
    initial begin
    b = 0;
    #100 a = 1;
    end
    always begin
        @(upedge a);
        b = a + 1;
        a = #20 b;
    end
endmodule
```

Intra-assignment events

- Intra-assignments means: while an assignment is executing

```
always
    #10 a = a + 1;
```

at time = 10, read a, add 1, assign to a

```
always
    a = #10 a + 1;
```

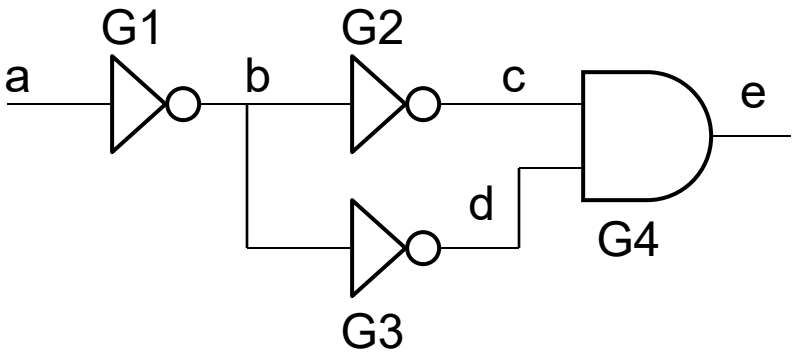
Used for transport delay

at time = 0, read a and add 1.
at time = 10 assign result to a.
Equivalent to:

```
always
    tmp = a + 1;
    #10 a = tmp;
```

Sensitivity List

- **Sensitivity List** of a process is the list of nets that will affect a given process
- Gate-level models have a fixed sensitivity list:



Sensitivity(G3) = {c, d}

Sensitivity List

- Behavioral models have a variable sensitivity list, changing during execution

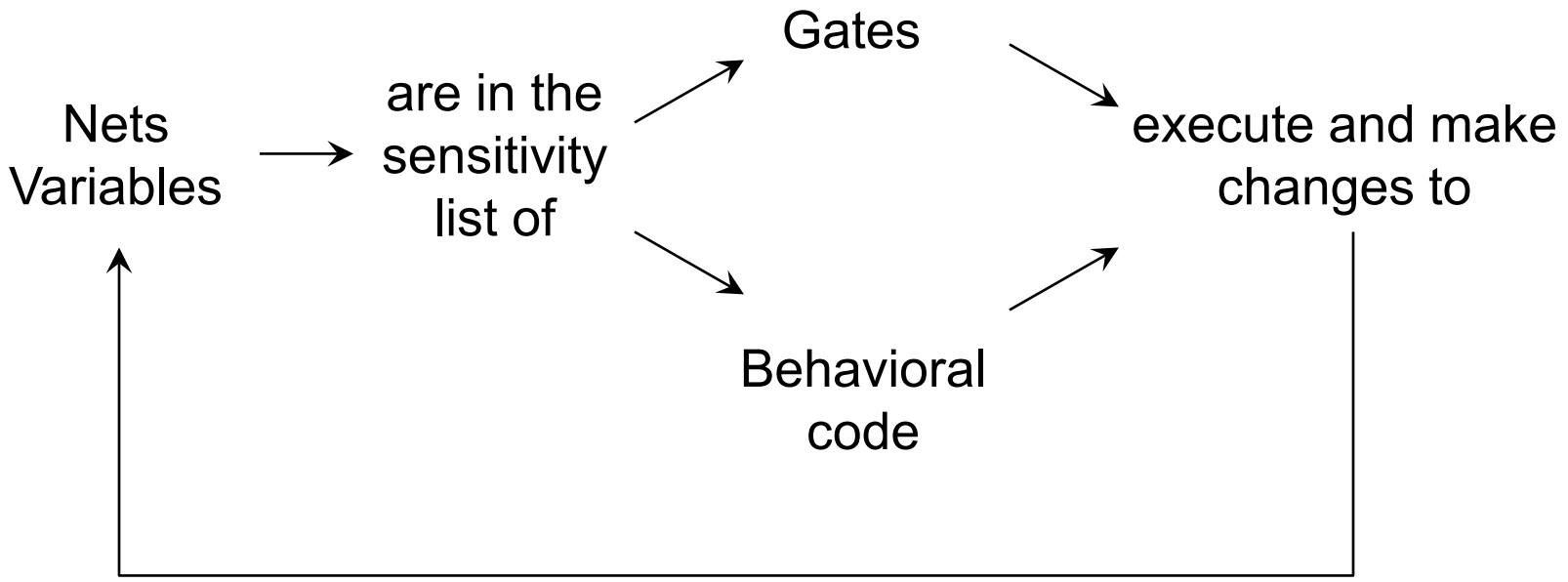
```
always begin
  @(upedge a);
  b = a + 1;
  wait (c == 3);
  #20 b = a + 1;
end
endmodule
```

Here it waits for an upedge of a

Here it waits for c to be 3

Here it waits for time to advance 20 units

So in a nutshell



- Events glue everything together and allow to mix gate-level code and behavioral code

Concurrent Programming - Pay attention!

- We already discussed this one ..

```
module syntst;
reg b;

initial
begin
  b = 0;
end

initial
begin
  b = 1;
end

endmodule;
```

Concurrent Programming - Pay attention!

- But sometimes it gets very subtle

```
module doh(muxout, select, a, b)
  output muxout;
  reg    muxout;
  input  a, b, select;
  wire notselect;

  always
    @select muxOut = (a & select) | (b & notselect);

  not (notselect, select);

endmodule;
```

What is the issue here ?

Concurrent Programming - Pay attention!

- But sometimes it gets very subtle

```
module doh(muxout, select, a, b)
  output muxout;
  reg    muxout;
  input  a, b, select;
  wire notselect;

  always
    @(select or notselect or a or b)
        muxOut = (a & select) | (b & notselect);

  not (notselect, select);

endmodule;
```

This may be better

Summary

- Event driven simulation:
  - Update events are tuples of (time + signal_value)
  - Evaluate events are tuples of (time + process_execution)
  - Events are sorted over time in an event queue and processed
- Fan-out = Set of process inputs driven by a signal
- Sensitivity List = Set of signals attached to process execution
- Gates have fixed fanout and sensivity list
- Behavioral models have variable fanout and sensitivity
  - They make use of event control (@, #, wait)