Simulators

Testing & Verification

Dept. of Computer Science & Engg, IIT Kharagpur

Pallab Dasgupta

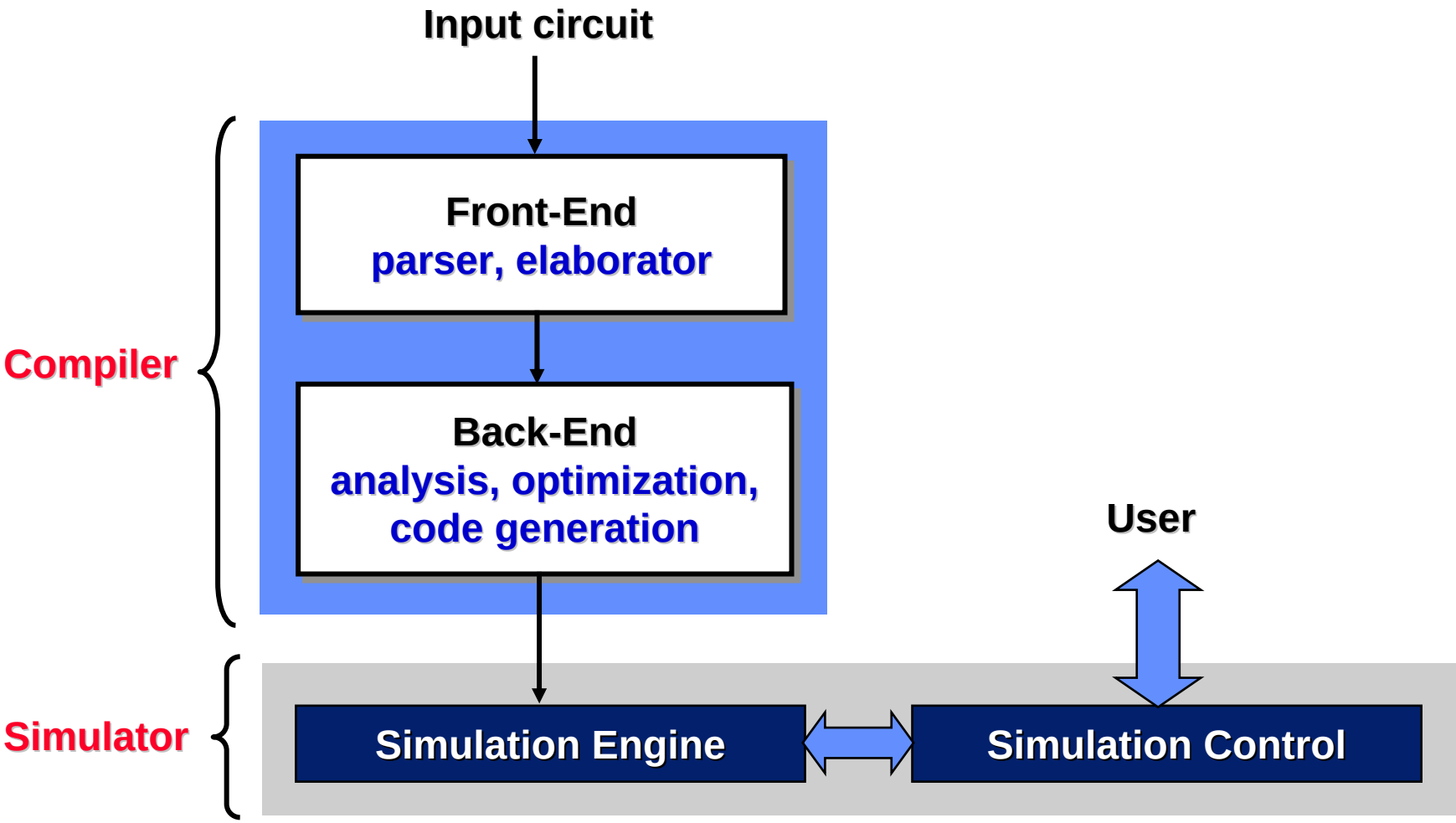
Professor, Dept. of Computer Science & Engg.,
Professor-in-charge, AVLSI Design Lab,
Indian Institute of Technology Kharagpur

Agenda

- The Compilers
- The Simulators
- Simulator Taxonomy
- Simulator Operations

Reference: ***Hardware Design Verification,*** **William K Lam**
Prentice Hall Modern Semiconductor Design Series

Major Components of a Simulator



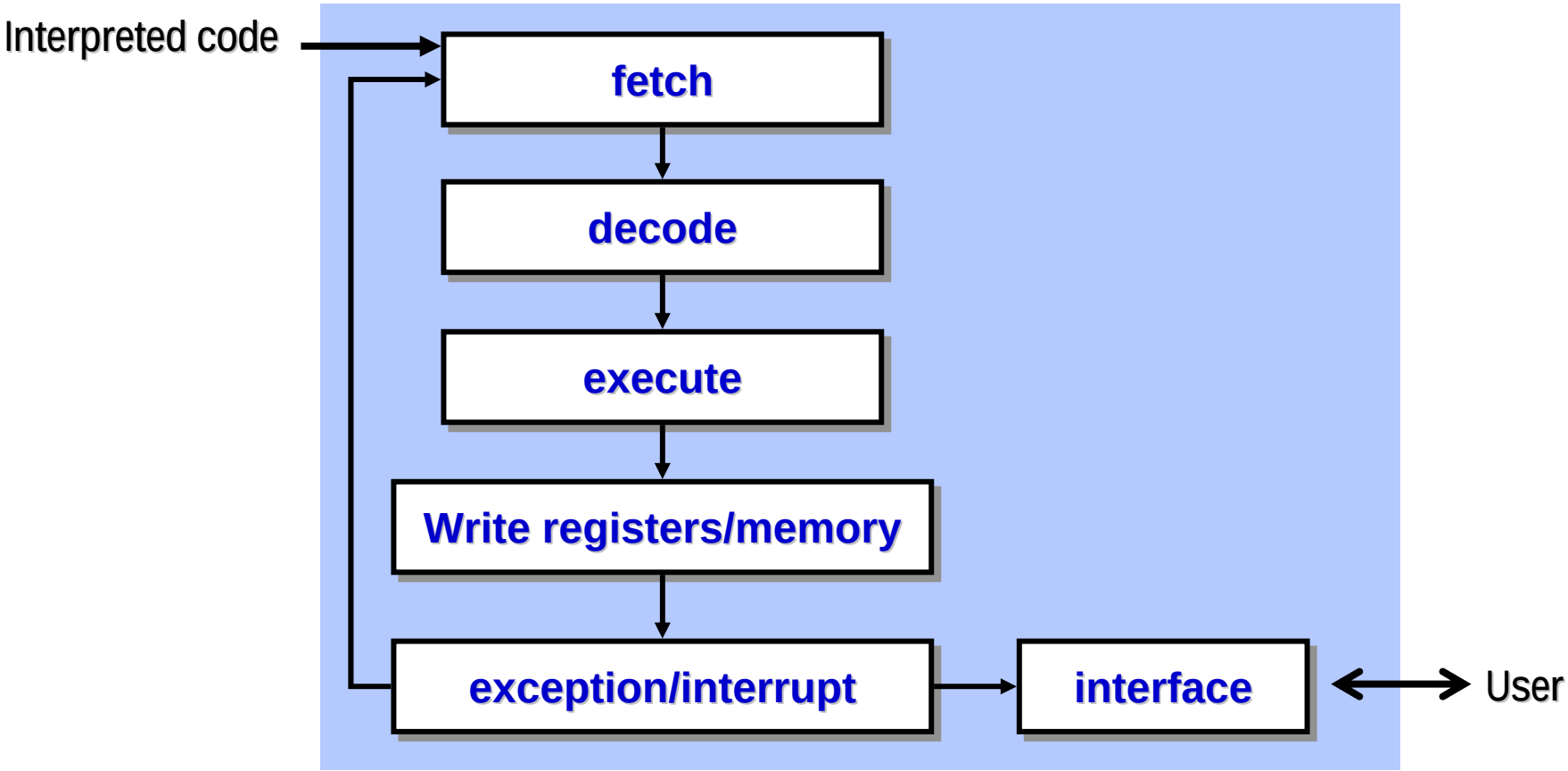
Parser and Elaborator

- The front-end portion of a compiler processes the input circuit and builds an internal representation of the circuit
  - Parser:
    - Interprets the input according to the language's grammar and creates corresponding internal components / data structures
  - Elaborator:
    - Substitutes module instantiations with their definitions and connects the internal objects
      - The end result is a complete description of the input circuit

Compiler Back-End

- **Functionality depends on the type of simulator:**
  - **Compiled code simulator**
    - High-level code
    - Native code
    - Emulation code
  - **Interpreted simulator**
    - The input circuit is compiled into an intermediate language
    - Can be regarded as a virtual machine

Interpreted Simulation



Interpreted Code

```
//circuit being simulated
initial
begin
  clk = 1'b0;
  #1 clk = ~clk;
  #1 clk = ~clk;
  #1 finish;
end

always @(clk)
begin
  a = b & c;
  if (a == 1'b0)
    p = q <<3;
end
```

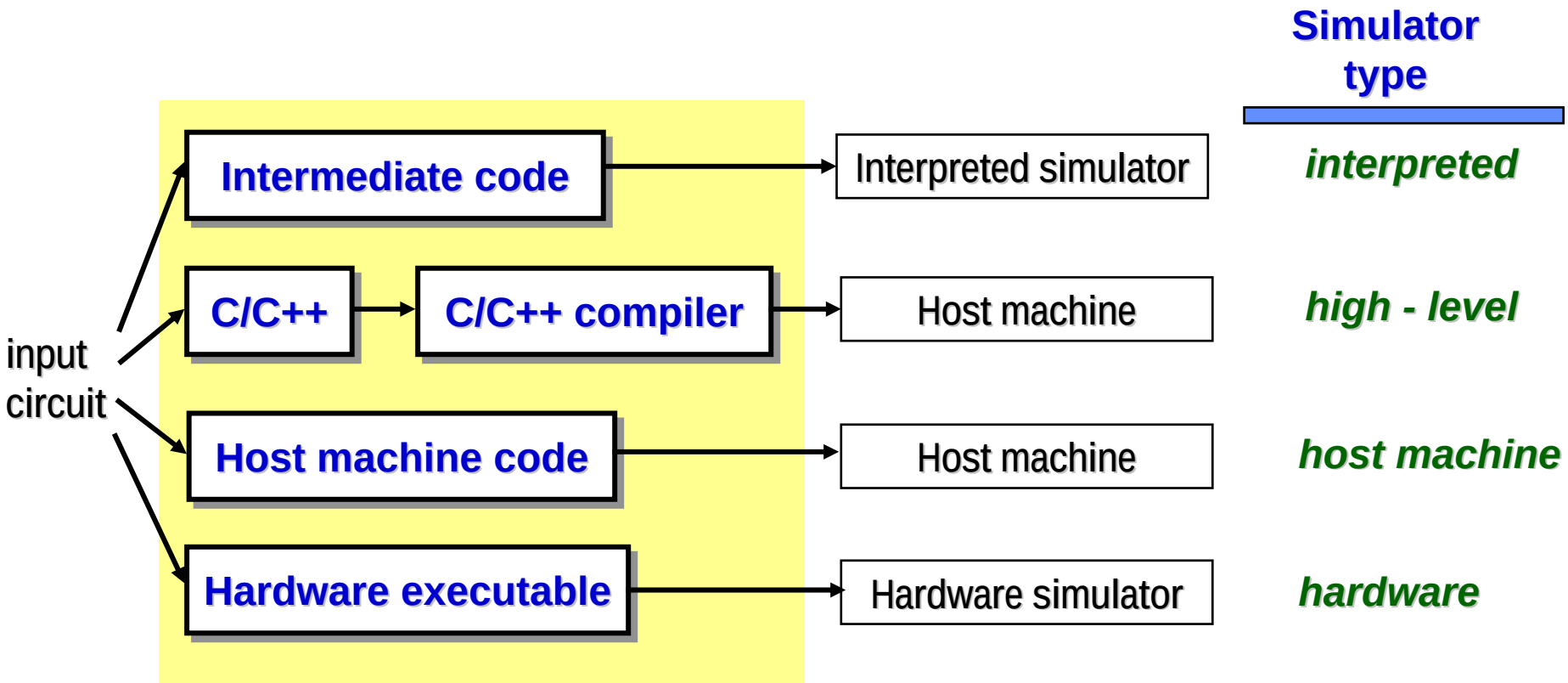
```
//generated interpreted code
 assign(clk, 0);
 invert(clk);
 evaluate(b1);
 invert(clk);
 evaluate(b1);
 exit( );

 b1:  //definition of routine b1
 {
   and(a, b, c);
   if (a, 0) left_shift(p, q, 3);
 }
```

Compiled Code

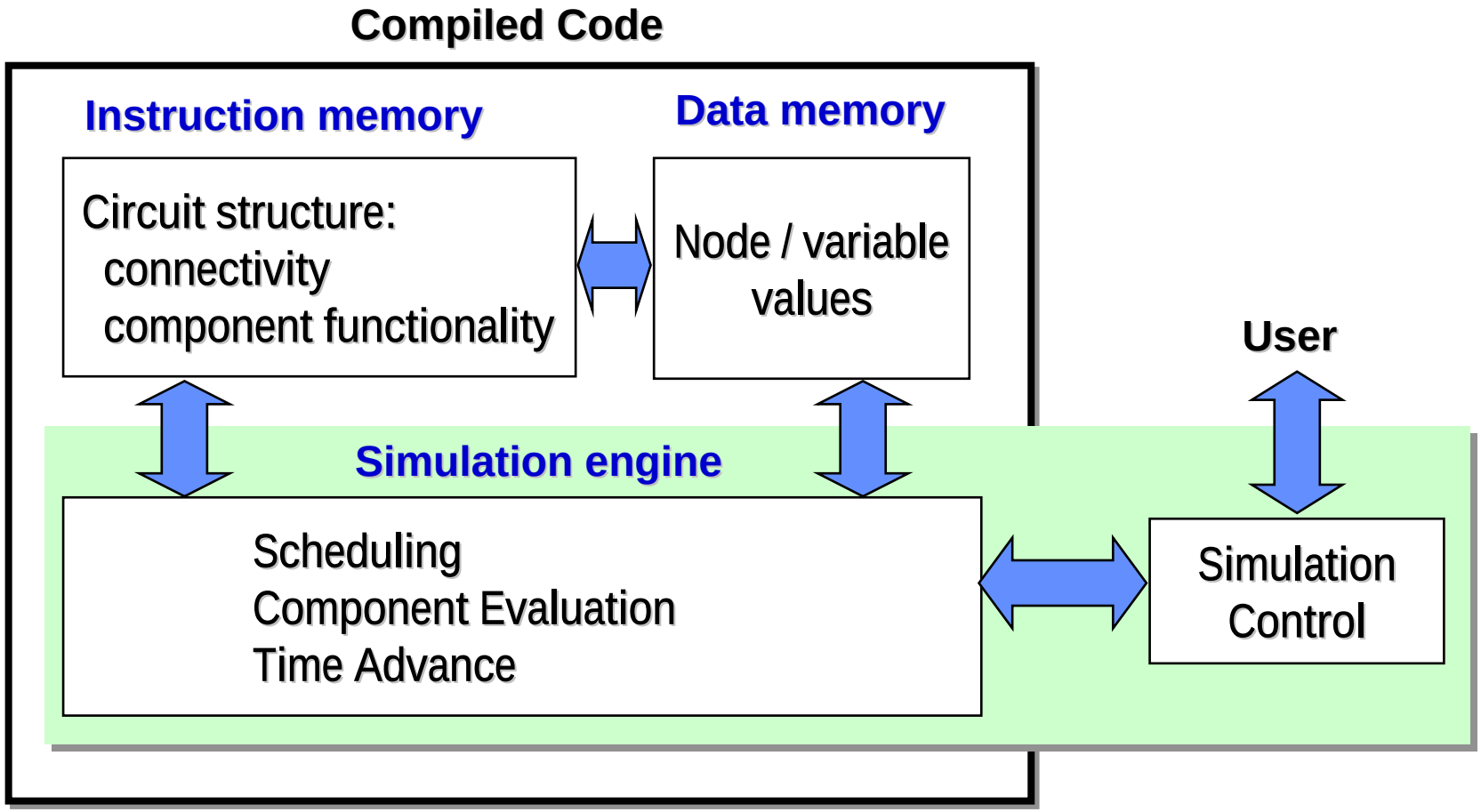
```
//circuit being simulated
initial
begin
  clk = 1'b0;
  #1 clk = ~clk;
  #1 clk = ~clk;
  #1 finish;
end

always @(clk)
begin
  a = b & c;
  if (a == 1'b0)
    p = q <<3;
end
```

```
main( )
{
    int clk;
    int i;
    int a, b, c, p, q;
    clk=0 ;
    for (i=0; i<2; i++) {
       clk = (clk==0) ? 1:0 ;
       a = b&c;
       if (a==0)
          p = q<<3;
    }
}
```

Simulator Types



Compiled Simulation System Structure



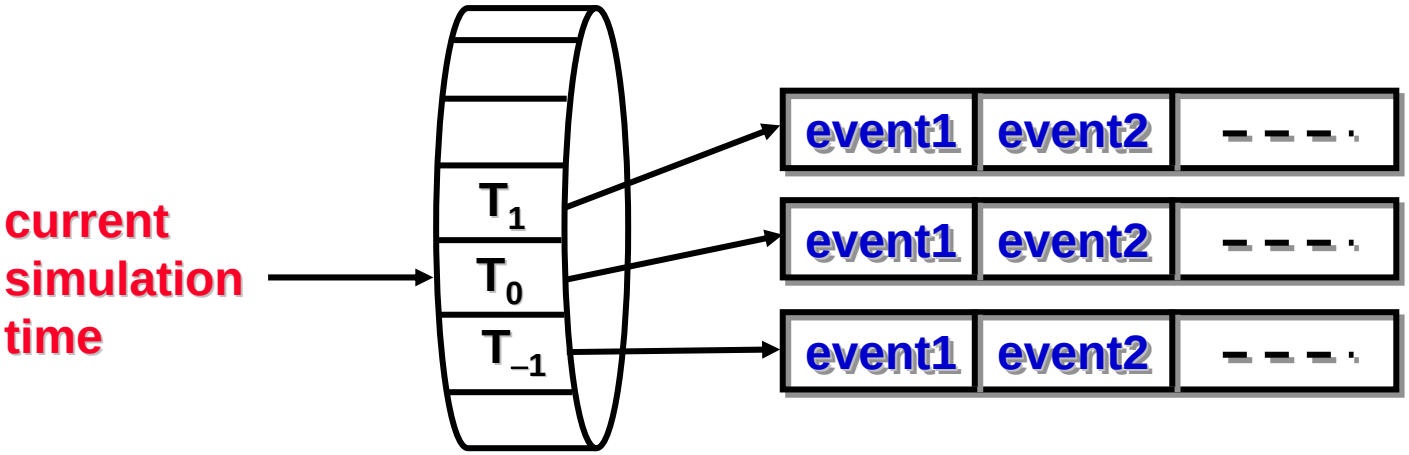
Simulator Architectures

- Event driven simulation
  - Evaluates a component only when there is an event at an input
    - This event ripples throughout the circuit until it causes no more events, at which time evaluation stops
- Cycle based simulation
  - Simulation performed only on cycle boundaries
    - The circuit must have clearly defined clocks
    - Asynchronous circuits and circuits with combinational loops cannot be simulated

Event Driven Simulation

- Timing wheel / Event Manager
- Scheduling semantics
- Update and Evaluation Events
- Event propagation
- Time advancement and oscillation detection
- Event-driven scheduling algorithm

Timing Wheel / Event Manager



- Events are stored in an event manager, which sorts them on time
  - Events occurring at the same time have an arbitrary order of occurrence
- The event queue may be implemented as a circular queue or a timing wheel

Example

Scheduling Semantics

- In Verilog, events at a simulation time are stratified into five layers of events in the following order of processing:
  - **Active**
    - The processing of all active events is called a *simulation cycle*
  - **Inactive**
    - Example: (#0 x=y) – *occurs at the current simulation time, but after all active event times*
  - **Non-blocking assign update**
    - First it samples the values of the right-side variables. Then it updates the left-side variables
  - **Monitor**
    - Executed as the last events at the current simulation time to capture steady state values of variables
  - **Future events**
    - Future events

For each time slot, in reality four sub-queues are maintained for the four groups of events.

Example

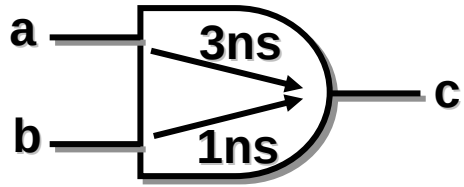
```
always @( posedge clock )
 begin
   x = a;
 end

 always @( posedge clock )
 begin
    x = b;
    y <= x;
    y = c;
 end
```

- x=a and x=b are active events, and their order of execution is arbitrary
- The value of y is either a or b, but never c, since y <= x is executed after y=c

Update and Evaluation Events

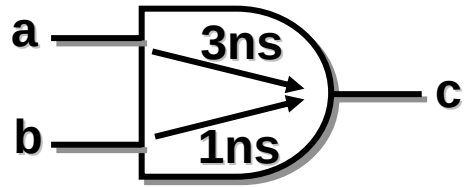
- **When an event is placed in a queue, it only means that the event may happen. Whether it actually will happen has to be evaluated**
  - **An *update event* occurs when a variable or a node changes its value**
  - **When an update event has occurred, all processes sensitive to the variable or node are triggered and must be evaluated.**
    - **This evaluation process is called an *evaluation event***
    - **If an evaluation event changes the values of some variables, then update events are generated for the affected variables**
  - **Update events simply replace the existing value of a variable or node with the new value**

Example



- At time 1, input b=1 and input a has a rising transition (update event)
  - This triggers an evaluation event for the AND gate
  - Since delay from a to c is 3, a future event, E1, for output c at time 4 is scheduled – *it appears a rising transition may occur at c at time 4.*
- At time 2, a fall transition occurs at input b
  - Since delay from b to c is 1, a future event, E2, for output c at time 3 is scheduled
  - When time advances to 3, we evaluate E2 and conclude that output c is 0. At time 4, *evaluating* E1 shows that c remains at 0. Hence event E1 is suppressed / cancelled.

Event Validation Algorithm



How did we validate and cancel E1 in the previous example?

Algorithm

1. Let the present time be T, g have n inputs, and the functionality of the gate g be f ( ).
2. For each input x_i of g, let the value of x_i at time $T - d_i$ be y_i, where d_i is the delay from x_i to the output of g.
3. The output value of g is $f(y_1, \ldots, y_n)$.

Event Propagation

- When an event has been confirmed to happen, all fanout gates or blocks sensitive to the event must be examined for event propagation
  - It is dangerous to delete a future event at the current simulation time when it appears not to happen in the future
  - It is interesting to note that fanouts can change during simulation
    - Example:

```
gate A(.out(x), …) ;
gate B(.out(y), …) ;
always
begin
   @x   a=b;   // Here this block is in the fanout of A
   @y   b=c;   // Here it is not in the fanout of A
 end
```

Time Advancement

- When no event for the current simulation time remains, the simulation time is advanced
  - If a maximum number of iterations is exceeded without time advancement, the simulator declares that an oscillation has occurred

Event-driven Scheduling Algorithm

A simulation cycle for event-driven simulator

```
while (there are events) {
    if (no events for current time) advance simulation time.
    for each (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
    }
}
```

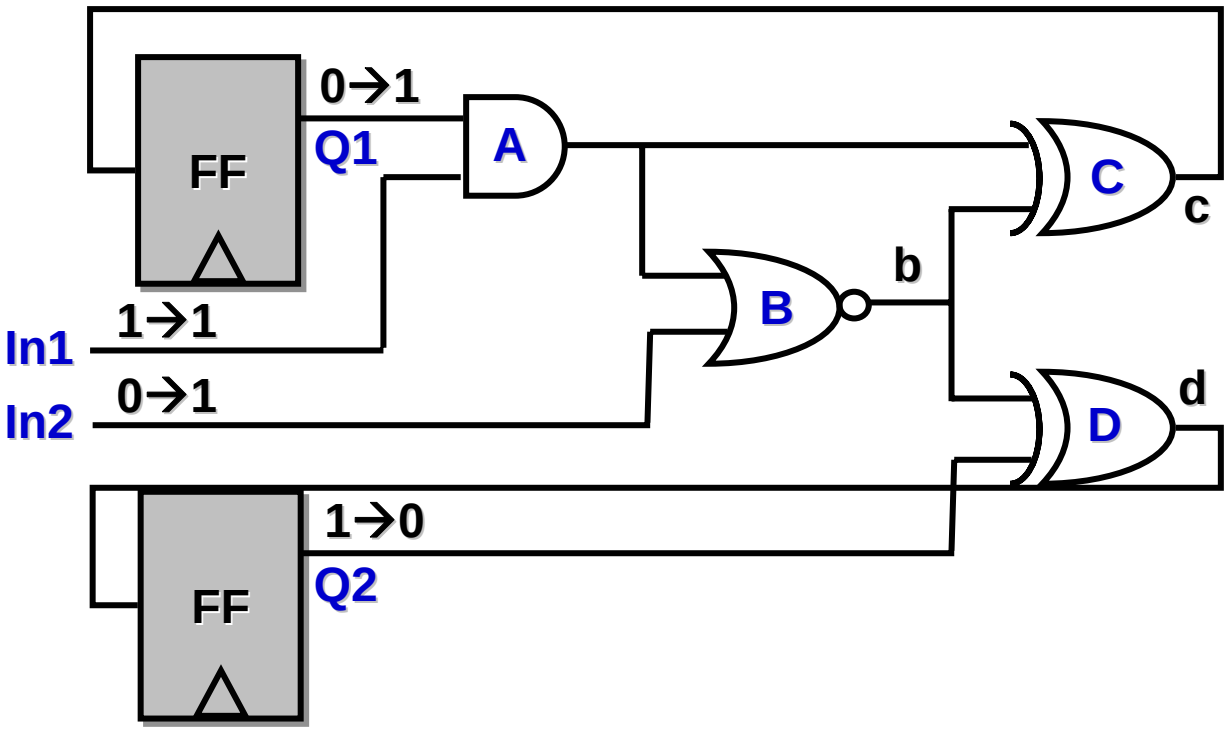
Cycle-Based Simulators

- In a sequential circuit, every time the FFs change:
  - many events are generated in the combinational logic,
  - but only the steady state is latched at the next clock edge
  - *Evaluations of all intermediate events are wasted*

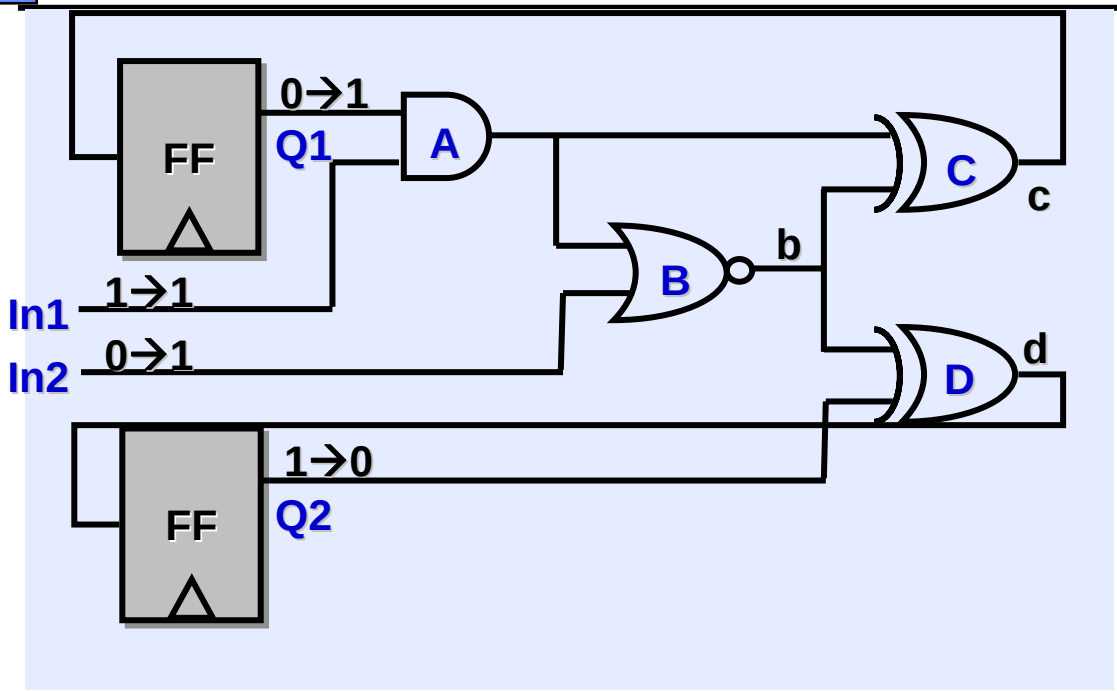
- Cycle-based simulators evaluate the combinational logic at each clock boundary
  - each gate is evaluated once in each cycle

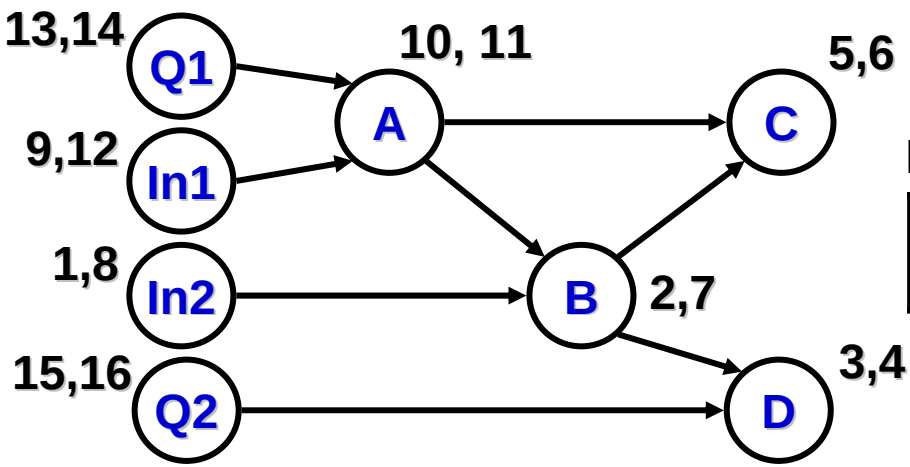
- Requirement: *The circuit must have clearly defined clocks and their associated boundaries*

Levelization for gate evaluation



Topological Sort for Levelization





Front List

Q2	Q1	In1	A	In2	B	C	D

Topological Sort Algorithm and DFS

Input: G(V, E)

Output: A queue of ordered nodes called List

Initialization: N=1

```
TopologicalSort(G) {
    while (node v in V is not marked visited) Visit(v);
}

Visit(v) {
    mark v visited;
    v.entry = N; N = N+1;          // Record node entry time
    for_each (u = fanout of v)
        if (u is not marked visited) Visit(u);
    v.exit = N; N = N+1;
    insert u in front of List;     // This line is only for topological sort
}
```

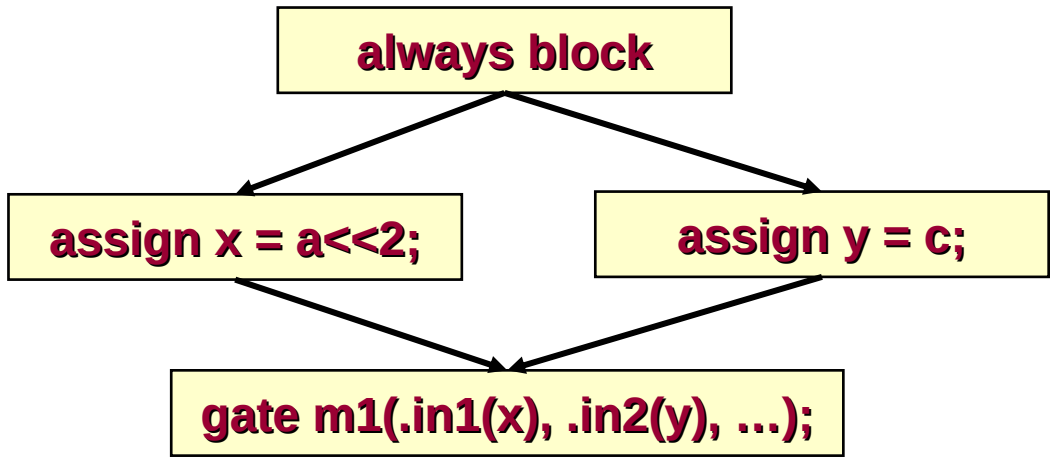
RTL Code Scheduling for Cycle-based Sim

RTL Code:

```
always @(posedge clk)
begin
    a = b;
    c <= a;
    $myPLI(a, b, d); //d is an output
    $strobe("a=%d, b=%d, c=%d", a, b, c);
    e = d;
end
assign x = a << 2;
assign y = c;
gate gate1(.in1(x), .in2(y), ...);
```

Order of execution:

1. a = b;
2. $myPLI(a, b, d);
3. e = d;
4. assign x = a<<2; assign y = c;
5. gate gate1(.in1(x), .in2(y), …);
6. c <= a;
7. $strobe("a=%d, b=%d, c=%d", a, b, c);



Clock Domain Analysis

- When a circuit has multiple clocks
  - Not all logic has to be evaluated at every clock transition
  - We have to determine the part of the circuit that requires evaluation at each clock's transition
    - This task is called *clock domain analysis*

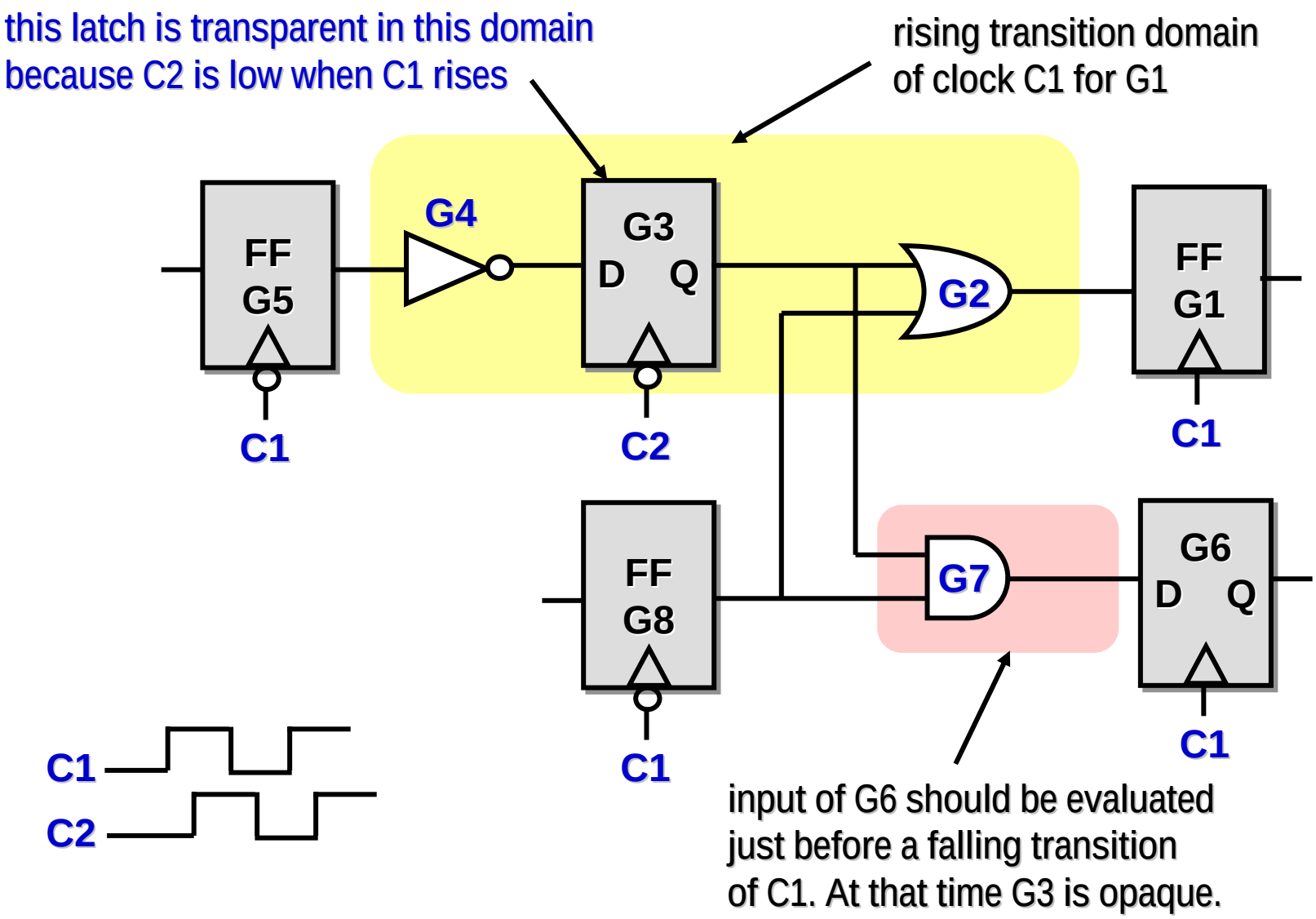
- A clock can potentially have two clock domains – one for a rising transition and one for a falling transition
  - This is because FFs and latches may be sensitive to rising/falling transitions of the clock

- A latch (level-sensitive) can be *opaque* or *transparent* depending on the current level of the clock (high/low)

Clock Domain Partitioning Algorithm

1. Identify all FFs and latches triggered by C
2. For an FF or latch, back trace from its input until it arrives at a primary input, an FF, or an opaque latch. The traversed logic is part of a clock domain
3. Repeat Step 2 for each of the FFs and latches
4. The union of all traversed logic of positive-triggered FFs and low transparent latches is the rising transition clock domain. The domain for the falling transition of the clock is similarly defined.

Clock Domain Partitioning – *Multiple clocks*



Hybrid Simulators

- **Compiled Event Driven Simulators**
  - Components of the circuit are compiled code
  - Triggering the evaluation of a component is dictated by the events among the components

- **Leveled Event Processing for Zero-Delay Simulation**
  - Resembles cycle-based simulation
  - Main difference:
    - Cycle-based simulation evaluates all circuit components
    - Levelized even-driven simulation evaluates only the ones with input events

Handling Combinational Loops

- A cycle-based simulator cannot handle combinational loops.
  Why?

- Solution: *Encapsulate combinational loops in macro models*
  - The macro models are simulated with an event driven simulator
  - The circuit with the macro models is simulated with a cycle based simulator

- Challenge: *To find and isolate all combinational loops*

Algorithm for Finding SCCs

Input: Graph G

Output: A collection of SCCs in G

1. Execute DFS on G and record the exit times for the nodes
2. Reverse the edges of G and apply DFS to this graph, selecting nodes in the order of decreasing exit number during the while loop step
3. The vertices of a DFS tree from Step2 form an SCC

Example

Simulator Taxonomy

- Two-State and Four-State Simulators
- Zero- versus Unit-Delay Simulators
- Event-Driven versus Cycle-Based Simulators
- Interpreted versus Compiled Simulators