SystemVerilog Simulation

- Semantics (meaning) of SystemVerilog is defined by the simulation model
- Want to write SystemVerilog that synthesises correctly and that behaves in the same way before and after synthesis
- Different simulators may produce different behaviour with poorly written models

Event-Driven Simulation

- The basis of SystemVerilog simulation is event processing. SystemVerilog simulators are event-driven simulators.
- There are three essential concepts to event-driven simulation:
  - simulation time
  - simulation regions
  - event processing
- The VHDL model is different – we will look at that later

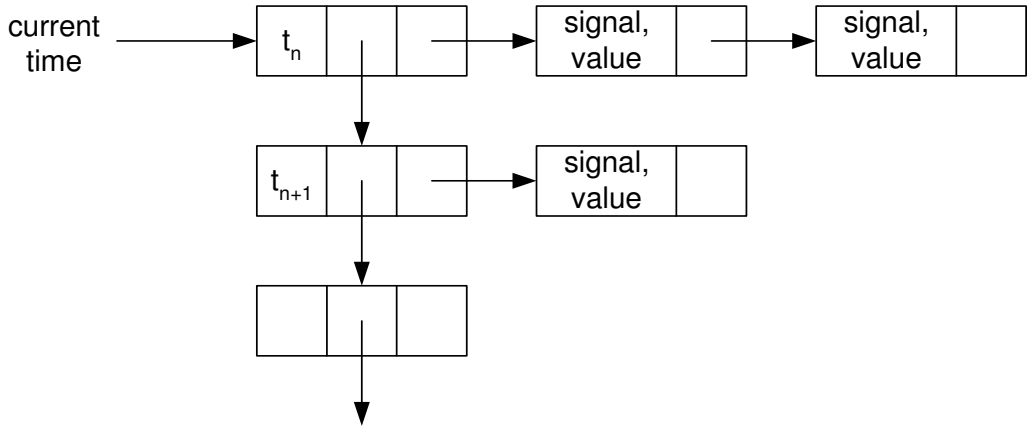
Simulation Time

- The simulator models time – this is called *simulation time*
- Simulation time is an integral multiple of the *resolution limit*.
- The simulator cannot measure time delays less than the resolution limit.
- For gate-level simulations the resolution limit may be quite fine, possibly down to 1ps. For RTL simulations, there is no need to specify a fine resolution since we are only interested in clock-cycle by clock-cycle behaviour and the transfer functions are described with zero delay or unit delay. In this case, a resolution limit of 1ns is usually used.

Simulation Cycle

- Process Execution:
  - each process is recalculated *only* if its sensitivity list has an *event*.
  - this creates a new value for each target signal and a time at which the signal gets the value. This value/time pair is a new event and is scheduled for the current or a future time.

Event Queue



- When an event is created it is added to the queue for the scheduled time

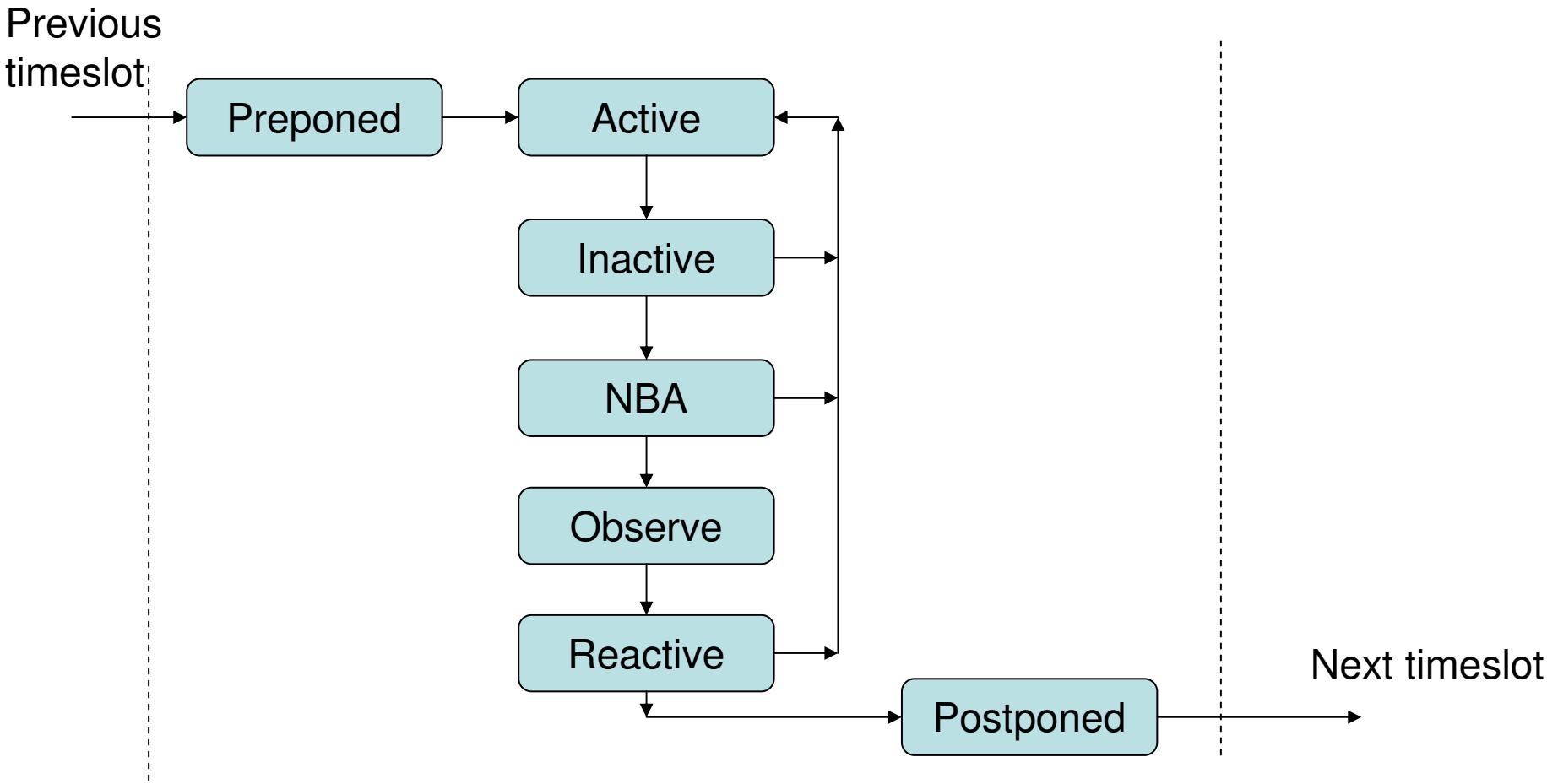
Verilog Simulation

- A new event may be one of five types. The LRM talks of a stratified event queue, with five regions as follows:

1. Active events. These occur at the present time and can be processed in any order.
2. Inactive events. These occur at the present time and are processed after all the active events have been processed.
3. Nonblocking assign update events.
4. Monitor events.
5. Future events.

SystemVerilog Event Regions

- Slightly different to Verilog (simplified here)



Event Regions

- Preponed – sample stable values for later checking
- Active – blocking assignments
- Inactive – zero delay assignments here – DO NOT USE!
- NBA – Nonblocking Assignments
- Observe – Evaluate assertions
- Reactive – Execute programs in testbenches
- Postponed - $strobe and $monitor print routines

Simulation Cycle

```
while (there are events) {
  if (no active events) {
    if (there are events in the other lists (in order) ) {
      activate all events in the next list;
    } else {
      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;
  }
}
```

- The list of active events is one of the lists, Inactive, Nonblocking, Reactive, that has been created during some previous simulation cycle, together with any (active) events that are generated during the current cycle.

- Nonblocking assign update events are created by nonblocking assignments (<=). The evaluation of the right hand side of all nonblocking assignments is *always* done before *any* nonblocking assign updates are done. This is important as it allows sequential systems to be modelled correctly.

- Inactive events are those events that are due to occur at the current time but that have been explicitly delayed. In practice, this can be done with a zero delay (#0). As a general guideline, do not use zero delays! A zero delay does not represent real hardware (nor a useful testbench construct). Therefore you are simply trying to fool the simulator. Unless you know exactly what you are doing, it will probably fool you!

- Reactive events are generated by assignments in program blocks
- A program block is like a module, but:
  - Can only be instantiated in a testbench
  - Can only contain initial procedural blocks
- Introduced in SystemVerilog to remove testbench problems

- Events may be processed from the active event list in any order (or to think of it another way, events can be added to the event lists in any order). This is the fundamental cause of non-determinism in Verilog.
- We can be sure of only two things:
  - Statements between begin and end will be executed in the order stated.
  - Nonblocking assignments will be performed after active events.

- *Everything* else is indeterminate. Moreover, the execution of a procedural block can be interupted to allow another procedural block to be executed. The skill in writing Verilog code is therefore to ensure that this indeterminism does not matter. If the code is badly written, a *race* condition is likely to result – that is 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.

Races - 1

```
assign p = q;
initial
  begin
  q = 1;
  #1 q = 0;
  $display(p);
  end
```

- Because the execution of the initial procedure can be interleaved with the assign statement, the value of p that is displayed could be 1 or 0. Either is “correct” and different simulators may give different results.

Races - 2

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

- This is supposed to model two flip-flops connected in series. If the procedures are evaluated in the order written, at a rising clock edge the value of c will be copied to b. That same value is copied to a. This is clearly not the intended behaviour. If the procedures are evaluated in the opposite order, the correct behaviour is modelled.

Avoiding Races

- Do not assign to the same variable from two or more procedures. Not only is contention liable, but multiple assignments can cause ambiguous behaviour. Part of the problem is the keyword “initial”. Procedures defined with the initial keyword are executed once; they are not initialisation blocks.
- Use nonblocking assignments for modelling sequential logic. The example, above, 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.

- Use blocking assignments to evaluate combinational logic. Assignments made in combinational logic models take immediate effect. The use of nonblocking assignments would often be confusing (and wrong).
- Do not mix blocking and nonblocking assignments in the same procedure.
- Don’t use zero delays (#0). They are not necessary and will cause confusion.

Different ways to model delays

- **Left-hand side (LHS) of blocking assignments:**

  #5 a = b;

  Wait then do assignment. Used for generating waveforms in a testbench

- **Right-hand side (RHS) of blocking assignments:**

  a = #5 b;

  Pure delay, e.g. a transmission line

- **LHS of nonblocking assignments:**

  #5 a <= b;

  No real difference to first form

- **RHS of nonblocking assignments:**

  a <= #5 b;

  Pure delay. Useful to model delays in flip-flops

Different ways to model delays

- **LHS of continuous assignments:**

  assign #5 a = b;

  Inertial delay in combinational logic
- **Inertial delay means that any transition is delayed by 5 units AND any pulse less than 5 units wide is suppressed**
- **Can get weird behaviour if you use the wrong form**
- **Can get differences between simulators**
- **Can specify zero delays – but don't!**