Always Blocks Using Event Control Statements

An alternative form for an always block uses wait or event control statements instead of a sensitivity list. If a sensitivity list is omitted at the always keyword, delays or time- controlled events must be specified inside the always block.

```
always
begin
   sequential-statements
   wait-statement
   sequential-statements
   wait-statement
   . . .
end
```

Such an always block could look like

```
always
begin
   rst = 1; // sequential statements
   @(posedge CLK); //wait until posedge CLK
   // more sequential statements
end
```

This always block will execute the `sequential-statements` until a wait (event control) statement is encountered. Then it will wait until the specified condition is satisfied. It will then execute the next set of `sequential-statements` until another wait is encountered. It will continue in this manner until the end of the always block is reached. Then it will start over again at the beginning of the block.

Example

For a half adder, sum and carry can be found using the equations sum = x XOR y ; carry = x AND y. What is wrong with the following code for a half adder that must add if add signal equals 1?

```
always @(*)
begin
if (add == 1)
    sum = x ^ y;
    carry = x & y;
end
```

(a) It will compile but not simulate correctly
(b) It will compile and simulate correctly but not synthesize correctly
(c) It will work correctly in simulation and synthesis
(d) It will not even compile

Answer: (a). This code will compile but will not simulate correctly. The if statement is missing begin and end. Currently only the sum is part of the if statement. The carry statement will get executed regardless of the add signal. This can be corrected by adding begin and end for the if statement. That will result in correct simulation. It can still lead to latches in synthesis. Latches can be avoided by adding else clause or by initializing sum and carry to 0 at the beginning of the always statement.

Delays

- Basically, delays in Verilog can be categorized into two models: inertial delay and transport delay. The inertial delay for combinational blocks can be expressed in the following three ways:

```
// explicit continuous assignment
wire D;
assign #5 D = A && B;

// implicit continuous assignment
wire #5 D = A && B;

// net declaration
wire #5 D;
assign D = A && B;
```

You

Delays in Verilog

```
always @ (X)
begin
  Z1 <= #10 (X);      // transport delay
end
assign #10 Z2 = X;    // inertial delay
```

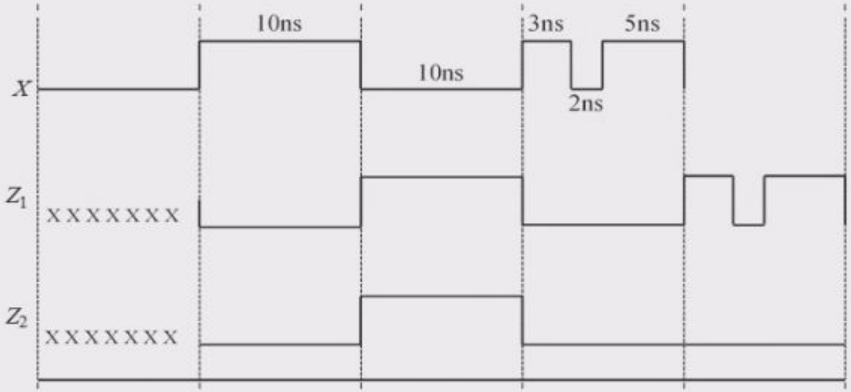
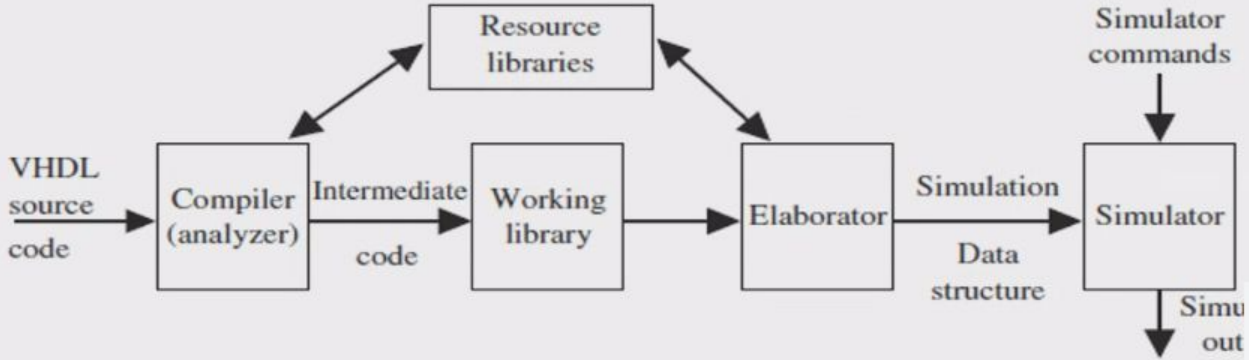


FIGURE 2-23: Inertial and Transport Delays

- **Inertial delay** is intended to model gates and other devices that do not propagate short pulses from the input to the output. If a gate has an ideal inertial delay *T*, in addition to delaying the input signals by time *T*, any pulse with a width less than *T* is rejected. For example, if a gate has an inertial delay of 5 ns, a pulse of width 5 ns would pass through, but a pulse of width 4.999 ns would be rejected.
- **Transport delay** is intended to model the delay introduced by wiring
- it simply delays an input signal by the specified delay time. In order to model this delay, a delay value must be specified on the right-hand side of the statement.

Compilation, Simulation, and Synthesis of Verilog Code

- There are three phases in the simulation of Verilog code: **analysis (compilation), elaboration, and simulation**



- Before the Verilog model of a digital system can be simulated, the Verilog code must first be compiled. The Verilog compiler, also called an **analyzer**, first checks the Verilog source code to see that it conforms to the syntax and semantic rules of Verilog.
- The compiler also checks to see that references to libraries are correct. If the Verilog code conforms to all of the rules, the compiler generates intermediate code, which can be used by a simulator or by a synthesizer.

- A design consists of connected threads of execution or processes. Processes are objects that can be evaluated, that may have state, and that can respond to changes on their inputs to produce outputs. Processes include modules, initial and always procedural blocks, continuous assignments, procedural assignment statements, sys-tem tasks
- Events can occur at different times. In order to keep track of the events and to make sure they are processed in the correct order, the events are kept on an **event queue**, ordered by simulation time. Putting an event on the queue is called **scheduling an event**.
- The Verilog event queue is logically segmented into five different regions:
- **Active event region:** Events that occur at the current simulation time are in this region. Events can be added to any of the five regions but can be removed only

- **Inactive event region:** Events that occur at the current simulation time but that shall be processed after all the active events are processed are in this region. Blocking assignments with zero delays are in this region until they get moved later to the active region.
- **Non-blocking assign update region:** Events that have been evaluated during some previous simulation time but that shall be assigned at this simulation time after all the active and inactive events are processed are in this region.
- **Monitor event region:** Events that shall be processed after all the active, inactive, and non-blocking assign update events are processed are in this region. These are the *monitor* events.
- **Future event region:** Events that occur at some future simulation time are in this region. These are the *future* events. Future events are divided into *future inactive events* and *future non-blocking assignment update events*.

- When each Verilog statement is processed, events are added to the various queue regions according to the following convention for each type of statement:
- **Continuous assignment**—evaluate RHS and add to active region as an active update event.
- **Procedural continuous assign**—evaluate RHS and add to active region as an update event.
- **Blocking assignment with delay**—compute RHS and put into future event region for time after delay.
- **Blocking assignment with no delay**—compute RHS and put into inactive region for current time.
- **Non-blocking assignment with no delay**—compute RHS and schedule as non-blocking assign update event for current time if zero delay.
- **Non-blocking assignment with delay**—compute RHS and schedule as non- blocking assign update event for future time if zero delay.
- **$monitor** and **$strobe** system tasks—create monitor events for these system tasks. (These events are continuously reenabled in every successive time step.)

- The processing of all the active events is called a **simulation cycle**.
- For each simulation time, the following actions are performed in order:

1. Process all active update events. (Whenever there is an active update event, the corresponding object is modified and new events are added to the various event queue regions for other processes sensitive to this update.)
2. Then activate all inactive events for that time (and process them because now they are active).
3. Then activate all non-blocking assign update events and process them.
4. Then activate all monitor events and process them.
5. Advance time to the next event time and repeat from step i.

- All of these five steps happen at the same time, but the events occur in the order active, inactive, non-blocking update, and monitor events.

Simulation with Multiple Processes (Initial or Always Blocks)

```
module twoprocess
  reg A,B;

initial
begin
  A = 0;
  B = 0;
end

// process P1 always
@(B) begin
  A <= 1;
  A <= #5 0;
end

// process P2 always
@(A) begin
   if (A)
       B <= #10 ~B;
end
```

Verilog Data Types

Data Types

- Verilog has two main groups of data types: the variable data types and the net data types. These two groups differ in the way that they are assigned and hold values. They also represent different hardware structures.
- The *net* data types can represent physical connections between structural enti- ties, such as gates. Generally, it does not store values. Instead, its value is deter- mined by the values of its drivers, such as a continuous assignment or a gate. A very popular *net* data type is the **wire**. There are also several other predefined data types

- that are part of nets. Examples are **tri** (for tristate), **wand** (for wired and), **wor** (for wired or).
- The *variable* data type is an abstraction of a data storage element. A variable shall store a value from one assignment to the next. An assignment statement in a procedure acts as a trigger that changes the value in the data storage element. A very popular *variable* data type is the **reg**. There are also several other predefined data types that are part of *variables*. Examples are **reg**, **time**, **integer, real,** and **real-time.**

- Unlike VHDL, all data types are predefined by the Verilog language and not by the user. Some of the popular predefined types are
- nets connections between hardware elements (declared with keywords such as **wir**e)
- variables data storage elements that can retain values (declared with the keywords such as **reg**)
- integer an integer is a variable data type (declared with the keyword **integer**)
- real real number constants and real variable data types for floating-point number (declared with the keyword **real**)
- time a special variable data type to store time information (declared with the keyword **time**)
- vectors wire or reg data types can be declared as vectors (multiple bits) (vectors can be declared with [range1 : range2])
- In previous versions of the Verilog standard, the term **register** was used to encompass the **reg**, **integer**, **time**, **real**, and **realtime** types, but starting with the 2005 IEEE 1364Standard, that term is no longer used as a Verilog data type. A net or reg declaration without a range specification shall be considered 1 bit wide and is known as a **scalar**. Multiple bit net and reg data types shall be declared by specifying a range, which is known as a **vector**.
- While VHDL is a strongly typed language where signals and variables of differ- ent types generally cannot be mixed in the same assignment statement, Verilog uses weak typing, which means some mixing of related data types is allowed.