7.5 ALWAYS CONSTRUCT

The **`always`** process signifies activities to be executed on an "always basis." Its essential characteristics are:

- Any behavioral level design description is done using an always block.
- The process has to be flagged off by an event or a change in a net or a reg. Otherwise it ends in a stalemate.

- The process can have one assignment statement or multiple assignment statements. In the latter case all the assignments are grouped together within a "**begin – end**" construct.
- Normally the statements are executed sequentially in the order they appear.

7.5.1 Event Control

The **always** block is executed repeatedly and endlessly. It is necessary to specify a condition or a set of conditions, which will steer the system to the execution of the block. Alternately such a flagging-off can be done by specifying an event preceded by the symbol "@". The event can be a change in the variable specified in either direction or a change in a specified direction. For example,

- @(**negedge** clk) :
  executes the following block at the negative edge of the **reg** (variable) clk.
- @(**posedge** clk) :
  executes the following block at the positive edge of the **reg** (variable) clk.
- @clk :
  executes the following block at both the edges of clk.

The event can be a combination as well.

- @(prt or clr) :
  With the above event the block is executed whenever either of the variables prt or clr undergoes a change.
- @(**posedge** clk1 or **negedge** clk2) :
  With the above event the block is executed in two cases – whenever the clock clk1 changes from 0 to 1 state or the clock clk2 changes from 1 to 0. One can specify more elaborate events by OR'ing individual ones. The following are to be noted:
- The events can be changes in **reg**, **integer**, **real** or a signal on a net. These should be declared beforehand.
- No algebra or logic operation is permitted as an event. The OR'ing signifies "execute the block if any one of the events takes place."
- The edge transition on each event is to be specified separately
- Note the difference between the following:
  - (**posedge** clk1 or clk2): means "execute the block following if clk1 goes to 1 state or clk2 changes state (whether 0 to 1 or 1 to 0)."
  - (**posedge** clk1 or **posedge** clk2): means "execute the block following if clk1 goes to 1 state or clk2 goes to 1 state."

- The positive transition for a reg type single bit variable is a change from 0 to1. For a logic variable it is a transition from false to true.
- The "**posedge**" transition for a signal on a net can be of three different types:
    - 0 to1
    - 0 to **x** or **z**
    - **x** or **z** to 1
- The "**negedge**" transition for a signal on a net can be of three different types:-
    - 1 to 0
    - 1 to **x** or **z**
    - **x** or **z** to 0
- If the event specified is in terms of a multibit **reg**, only its least significant bit is considered for the transition. Changes in the other bits are ignored.
- The event-based flagging-off of a block is applicable only to the **always** block.
- According to the recent version of the LRM, the comma operator (,) plays the same role as the keyword **or**. The two can be used interchangeably or in a mixed form. Thus the following are identical:

```
@ (a or b or c)
@ (a or b, c)
@ (a, b, c)
@ (a, b or c)
```

7.4 INITIAL CONSTRUCT

A set of procedural assignments within an **initial** construct are executed only once – and, that too, at the times specified for the respective assignments. Consider the **initial** process shown in Figure 7.3. It is characterized by the following:

- In any assignment statement the left-hand side has to be a storage type of element (and not a net). It can be a **reg**, **integer**, or **real** type of variable. The right-hand side can be a storage type of variable (**reg**, **integer**, or **real** type of variable) or a net.
- As already mentioned in Section 7.2, all the operations described in Tables 6.1 to 6.9 for continuous assignment can be used for procedural assignments as well. The context decides whether the assignment is of a continuous type or procedural type. In the latter case it is present within an **always** or an **initial** construct.
- All the procedural assignments appear within a **begin–end** block explained earlier.
- All the procedural assignments are executed sequentially – in the same order as they appear in the design description. The waveforms of a and b conforming to the assignments in the block are shown in Figure 7.4.
- Initially (at time t = 0 ns), a and b are set equal to zero.

```
reg a,b;
initial
      begin
            a = 1'b0;
            b = 1'b0;
      #2    a = 1'b1;
      #3    b = 1'b1;
      #1    a = 1'b0;
      #100$stop;
      end
```

Figure 7.3 A typical initial block.

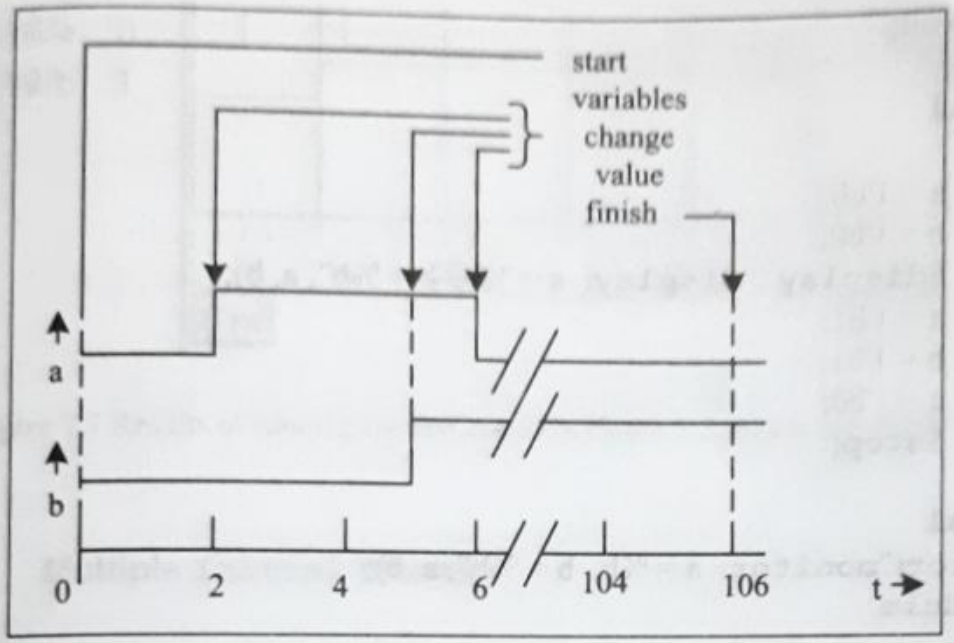

Figure 7.4 Nature of variation of a and b with time in the module of Figure 7.3.

- At time 2 ns `a` is made equal to 1. After 3 more nanoseconds – that is, at the 5th ns – `b` is made equal to 1.
- After one more ns – that is, at the 6th ns – `a` is made equal to 0.
- `$stop` is a system task. 100 ns later – that is, at the 106th ns – the simulation comes to an end (see Figure 7.4).

Integer values have been used here to decide time delay values. In a more general case the delay value can be a constant expression. It is evaluated and decided dynamically as the simulation proceeds.

The **`initial`** block above does three controlling activities during the simulation run.

- Initialize the selected set of **`reg`**'s at the start.
- Change values of **`reg`**'s at predetermined instances of time. These form the inputs to the module(s) under test and test it for a desired test sequence.
- Stop simulation at the specified time.

Figure 7.4 depicts the events for the above case; *t* is the time axis here.

Specific system tasks available in Verilog can be used to tabulate the values of selected variables. Providing such output display in a desired or preferred format is the activity of the simulation run. Two system tasks are useful here – **`$display`** & **`$monitor`** [see Section 3.15 and Chapter 11]. By way of illustration consider the simulation routine in Figure 7.5. It incorporates the block

```
module nil;
reg a, b;
initial
begin
        a = 1'b0;
        b = 1'b0;
        $display ("display: a = %b, b = %b", a, b);
  #2    a = 1'b1;
  #3    b = 1'b1;
  #1    a = 1'b0;
  #100  $stop;
end
initial
$monitor("monitor: a = %b, b = %b", a, b);
endmodule
```

Figure 7.5 A typical module with an **initial** block.

Figure 7.3 and two system tasks. The result of the simulation is shown in Figure 7.6. The **$display** task is a one-time activity. It is executed when encountered. At that instant in simulation the values of *a* and *b* are zero and the same are displayed. In contrast, **$monitor** is a repeated activity. It need be present only once in a simulation routine – all the specified variables will be monitored. If multiple **$monitor** tasks are present in the routine, only the last one will be active. All others will be ignored. In contrast, the **$display** task may appear any number of times in a module. It is executed every time it is encountered.

Simulators have the facility to observe the waveforms and changes in the magnitudes of different variables with simulation time. The necessary facility is provided with the help of user-friendly menus and icons. Waveforms of *a* and *b* obtained with the test bench of Figure 7.5 are shown in Figure 7.7; they can be seen to be consistent with their values shown in Figure 7.6.

```
output
        # display : a = 0 ,b = 0
        # monitor : a = 0 ,b = 0
        # monitor : a = 1 ,b = 0
        # monitor : a = 1 ,b = 1
        # monitor : a = 0 ,b = 1
```

Figure 7.6 Results of running the test bench in Figure 7.5.

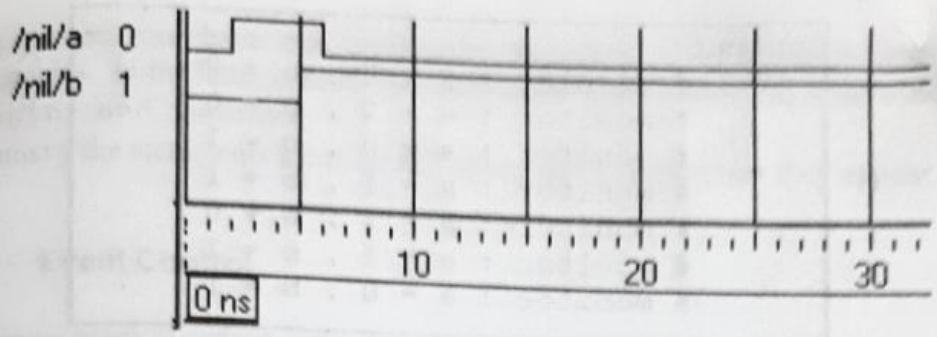

Figure 7.7 Results of running the test bench in Figure 7.5 shown as waveforms.

7.4.1 Multiple `Initial` Blocks

A module can have as many **`initial`** blocks as desired. All of them are activated at the start of simulation. The time delays specified in one **`initial`** block are exclusive of those in any other block. Consider the module in Figure 7.8 which is a modified version of that in Figure 7.5. It has four **`initial`** blocks. The **`$monitor`** task is declared separately (a healthy practice). The simulated results are shown in Figure 7.9. The following observations are in order here:

```
module nil1;
initial
reg a, b;
begin
        a = 1'b0;
        b = 1'b0;
        $display ($time,"display:  a = %b,  b = %b", a, b);
 #2     a = 1'b1;
 #3     b = 1'b1;
 #1     a = 1'b0;
end
initial #100$stop;
initial $monitor ($time, "monitor:  a = %b,  b = %b", a, b);
initial
        begin
 #2     b = 1'b1;
        end
endmodule
```

Figure 7.8 A typical module with multiple initial blocks.