BLOCK STATEMENT

Block Statement

- A block statement provides a mechanism to group two or more statements to act syntactically like a single statement.
- A block can be labeled optionally. If so labeled, registers can be declared locally within the block.
- Blocks can also be referenced; for example, a block can be disabled using a disable statement.
- A block label, in addition, provides a way to uniquely identify registers.
- However, there is one word of caution. All local registers are static, that is, their values remain valid throughout the entire simulation run.

There are two kinds of blocks in Verilog HDL

1. Sequential block: Statements are executed sequentially in the given order.

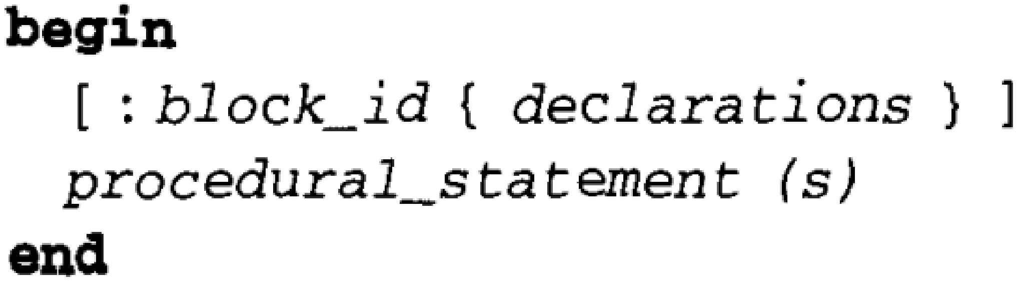
```
begin
     ..................
end
```

2. Parallel block: Statements in this block execute concurrently

```
fork
     ......................
join
```

1. Sequential Block

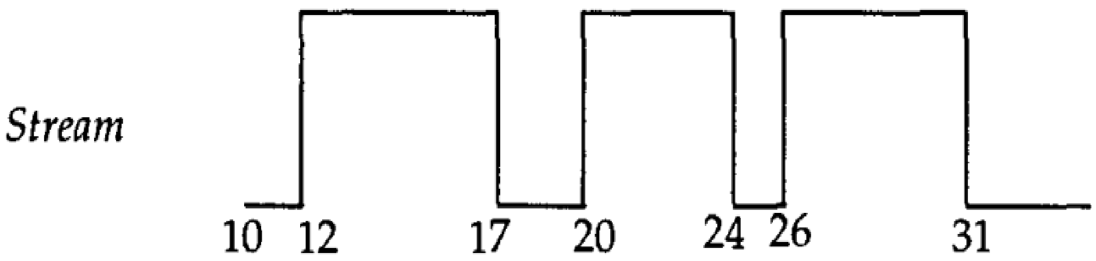
- Statements in a sequential block execute in sequence.
- **A delay value in each statement is relative to the simulation time of the execution of the previous statement**.
- Once a sequential block completes execution, execution continues with the next statement following the sequential block.
- Here is the syntax of a sequential block.



```
begin
  [ : block_id { declarations } ]
  procedural_statement (s)
end
```

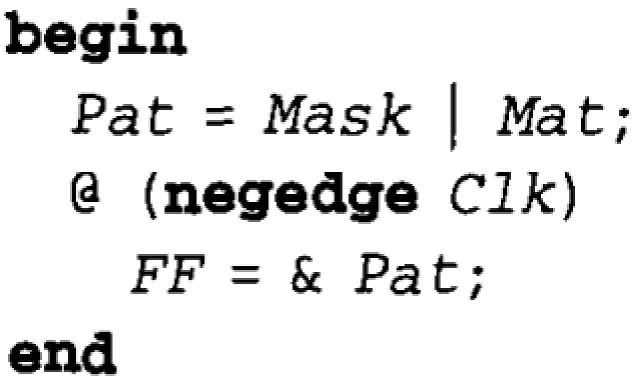
Here is an example of a sequential block.

```
// Waveform generation:
begin
  #2 Stream = 1;
  #5 Stream = 0;
  #3 Stream = 1;
  #4 Stream = 0;
  #2 Stream = 1;
  #5 Stream = 0;
end
```



- □ *Assume that the sequential block gets executed at 10 time units.*
- □ *The first statement executes after 2 time units, that is at 12 time units.*
- □ *After this execution has completed, the next statement is executed at 17 time units (because of the delay).*
- □ *Then the next statement is executed at 20 time units and so on.*

Eg. 2

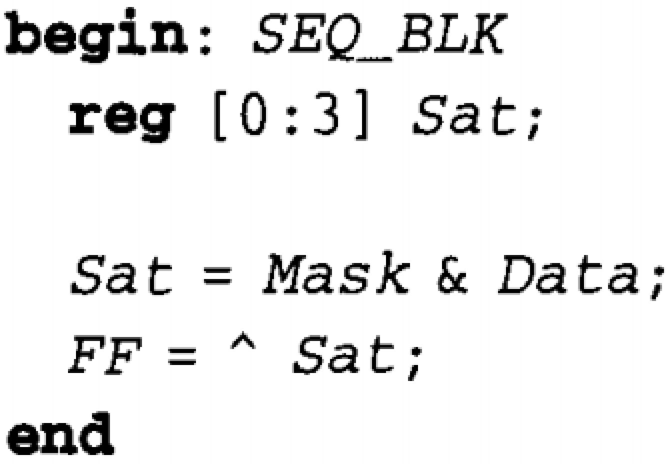


```
begin
   Pat = Mask | Mat;
   @ (negedge Clk)
      FF = & Pat;
end
```

□ In this example, the first statement executes first and then the second statement executes.

□ Of course, the assignment in the second statement occurs only when a negative edge appears on Clk.

Eg.3



```
begin: SEQ_BLK
   reg [0:3] Sat;

   Sat = Mask & Data;
   FF = ^ Sat;
end
```

- In this example, the sequential block has a label SEQ_BLK and it has a local register declared.
- Upon execution, the first statement is executed, then the second statement is executed

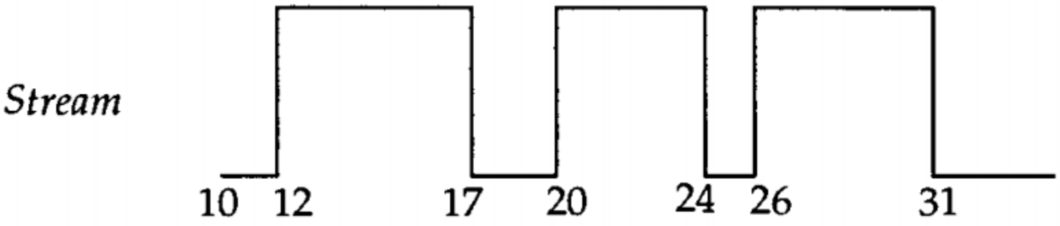
2. Parallel Block

- A parallel block has the delimiters fork and join (a sequential block has the delimiters begin and end).
- Statements in a parallel block execute concurrently.
- Delay values specified in each statement within a parallel block are relative to the time the block starts its execution.
- When the last activity in the parallel block has completed execution (this need not be the last statement), execution continues after the block statement.
- Stated another way, all statements within the parallel block must complete execution before control passes out of the block.

Syntax:

```
fork
   [ : block_id { declarations } ]
   procedural_statement (s)
join
```

Here is an example.

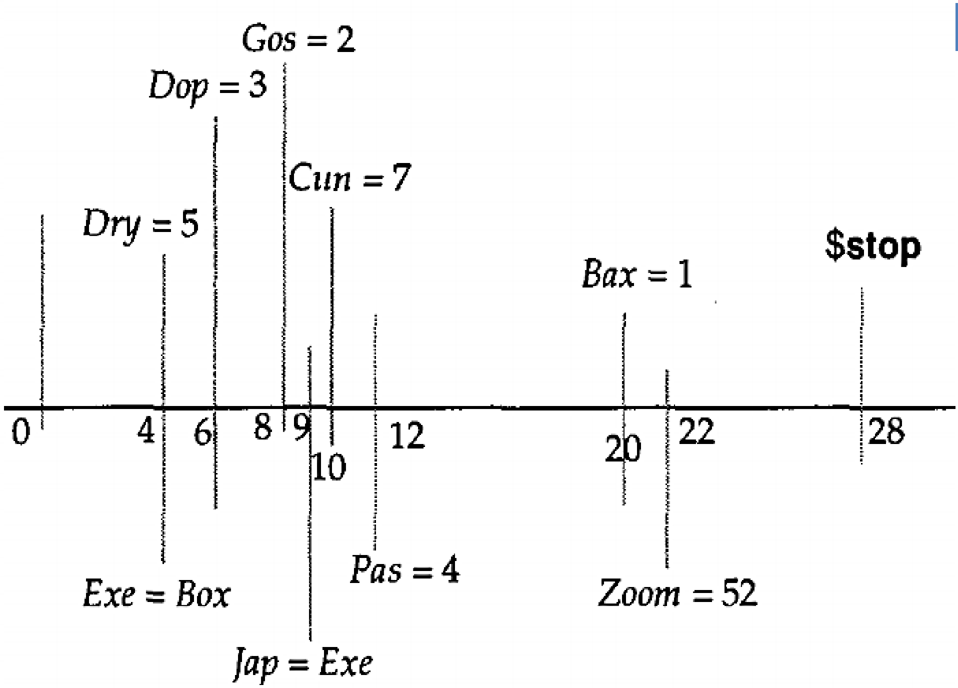


```
// Waveform generation:
fork
   #2 Stream = 1;
   #7 Stream = 0;
   #10 Stream = 1;
   #14 Stream = 0:
   #16 Stream = 1;
   #21 Stream = 0;
join
```

- If the parallel block gets executed at 10 time units, all statements execute concurrently and all delay values are relative to 10.
- For e.g, the 3rd assignment executes at 20 time units, the 5th assignment executes at 26 time units, and so on

A mix of sequential and parallel blocks to emphasize their differences

```
always
  begin: SEQ_A
    #4 Dry = 5;                // S1

    fork: PAR_A                // S2
      #6 Cun = 7;              // P1

      begin: SEQ_B             // P2
        Exe = Box;             // S6
        #5 Jap = Exe;          // S7
      end

      #2 Dop = 3;              // P3
      #4 Gos = 2;              // P4
      #8 Pas = 4;              // P5
    join

    #8 Bax = 1;                // S3
    #2 Zoom = 52;              // S4
    #6 $stop;                  // S5
  end
```

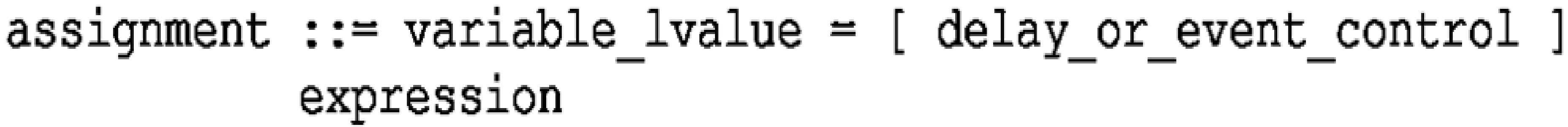


- The always statement contains a sequential block SEQ_A and all statements within the sequential block (SI, S2, S3, S4, S5) execute sequentially.
- Since the always statement executes at time 0, Dry gets assigned 5 at 4 time units, and the parallel block PAR_A begins its execution at 4 time units.
- All statements within the parallel block (PI,P2,P3,P4,P5) execute concurrently, that is, at 4 time units. Thus Cun gets assigned a value at 10 time units, Dop gets assigned at 6, Gos gets assigned at 8, and Pas gets assigned at 12.
- The sequential block SEQ_B starts execution at 4 time units, causing statements S6 and then S7 to execute. Jap gets its new value at 9.
- Since all statements within the parallel block PAR_A complete execution at time 12, statement S3 is executed at 12 time units, assignment to Bax occurs at 20, then statement S4 executes, assignment to Zoom occurs at 22, then the next statement executes.
- Finally the system task $stop executes at time 28.

PROCEDURAL ASSIGNMENTS

Procedural Assignments

- A procedural assignment is an assignment within an initial statement or an always statement.
- It is used to assign to only a register data type
- The right-hand side of the assignment can be any expression.
- The value placed on a variable will remain unchanged until another procedural assignment updates the variable with a different value.
- These are unlike continuous assignments (Dataflow Modeling), where one assignment statement can cause the value of the right-hand-side expression to be continuously placed onto the left-hand-side net.
- The syntax for the simplest form of procedural assignment



```
assignment ::= variable_lvalue = [ delay_or_event_control ]
         expression
```

- There are two types of procedural assignment statements:

1. BLOCKING **2. NON-BLOCKING**

- Let us first discuss the notion of intra-statement delays.

INTRA-STATEMENT DELAY

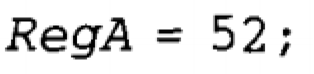
- A delay appearing on the left of an expression in an assignment statement is an intra-statement delay. It is the delay by which the right-hand side value is delayed before it is applied to the left-hand side target.
- The important thing to note about this delay is that the right-hand side expression is evaluated before the delay, then the wait occurs and then the value is assigned to the left-hand side target.

Eg:

```
Done = #5 'b1;
Done = #5 'b1;          // Intra-statement delay control.

// is the same as:
begin
  Temp = 'b1;
  #5 Done = Temp;       // Inter-statement delay control.
end


Q = @(posedge Clk) D;   // Intra-statement event control.

// is the same as:
begin
  Temp = D;
  @(posedge Clk)        // Inter-statement event control.
    Q = Temp;
end
```

1. Blocking Procedural Assignment

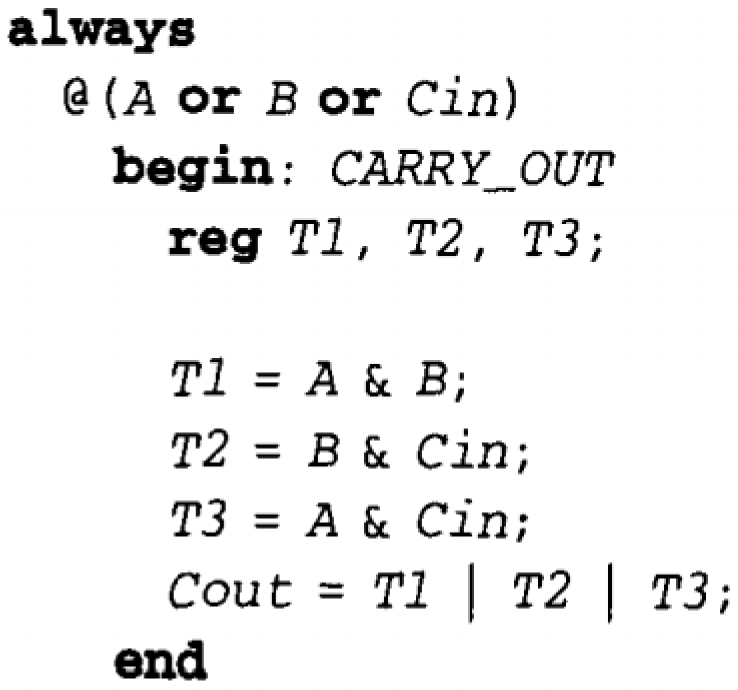
- □ A procedural assignment in which the assignment operator is an "=" is a blocking procedural assignment.
- □ For example



```
RegA = 52;
```

- □ Blocking assignment statements are executed in the order they are specified in a sequential block.
- □ A blocking assignment will block execution of statements that follow in a parallel block.

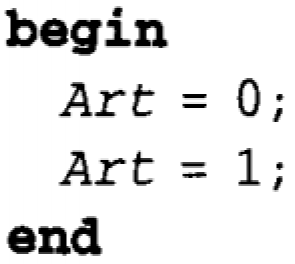
Eg. 1:



```
always
  @(A or B or Cin)
    begin: CARRY_OUT
      reg T1, T2, T3;

      T1 = A & B;
      T2 = B & Cin;
      T3 = A & Cin;
      Cout = T1 | T2 | T3;
    end
```

The T1 assignment occurs first, T1 is computed, then the second statement executes, T2 is assigned and then the third statement is executed and T3 is assigned, and so on.

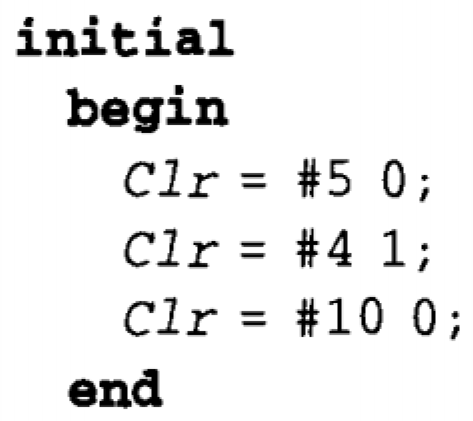
Eg. 2:



```
begin
  Art = 0;
  Art = 1;
end
```

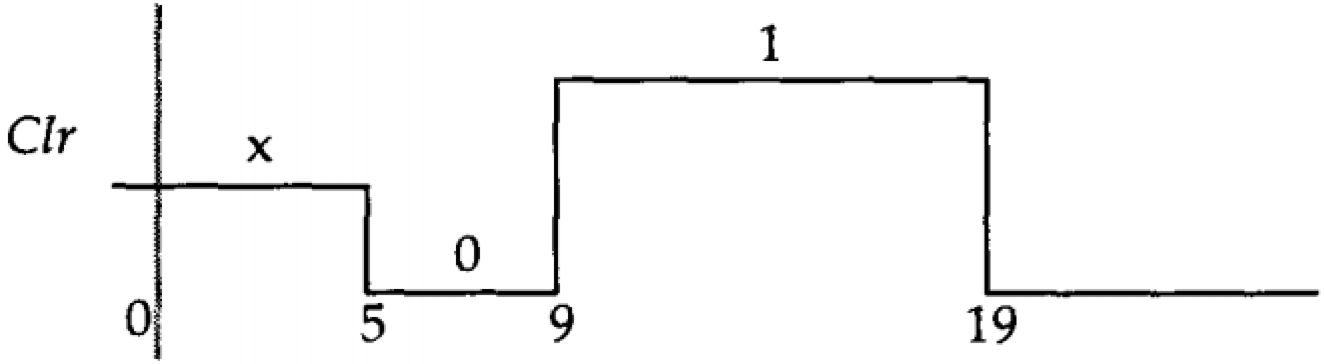
In this case, Art gets assigned the value 1. This is because, first Art gets assigned 0, then the next statement executes that causes Art to get 1 after zero delay. Thereforethe assignment of 0 to Art is lost.

Eg. 3:



```
initial
  begin
    Clr = #5 0;
    Clr = #4 1;
    Clr = #10 0;
  end
```

The 1st statement executes at time 0 and Clr gets assigned 0 after 5 time units, then the 2nd statement executes causing Clr to get assigned a 1 after 4 time units (9 time units from time 0), and then the third statement executes causing Clr to get a 0 after 10 time units (19 time units from time 0).

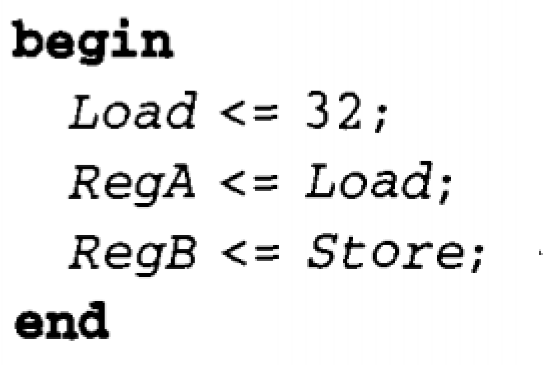


2. Non-blocking Procedural Assignment

- Non blocking assignments allow scheduling of assignments without blocking execution of the statements that follow in a sequential block.
- A <= operator is used to specify non-blocking assignments.
- Note that this operator has the same symbol as a relational operator, less_than_equal_to. The operator <= is interpreted as a relational operator in an expression and as an assignment operator in the context of a non-blocking assignment.
- In a non-blocking procedural assignment, the assignment to the target is not blocked (due to delays) but are scheduled to occur in the future (based on the delays; if zero delay, then at the end of current time step).

- When a non-blocking procedural assignment is executed, the right-hand side expression is evaluated and its value is scheduled to be assigned to the left-hand side target, and execution continues with the next statement.
- The earliest an output would be scheduled is at the end of the current time step; this case would occur if there were no delay in the assignment statement.
- At the end of the current time step or whenever the outputs are scheduled, the assignment to the left-hand side target occurs.

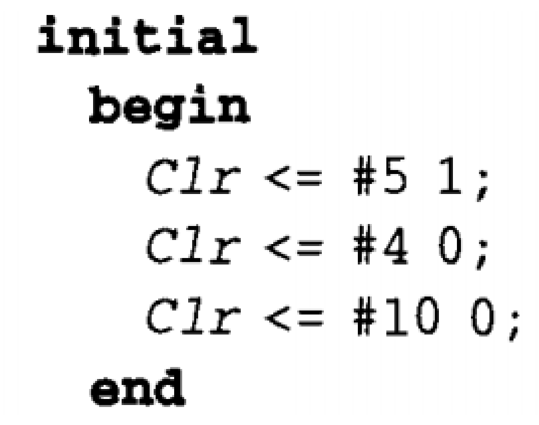
Example 1:



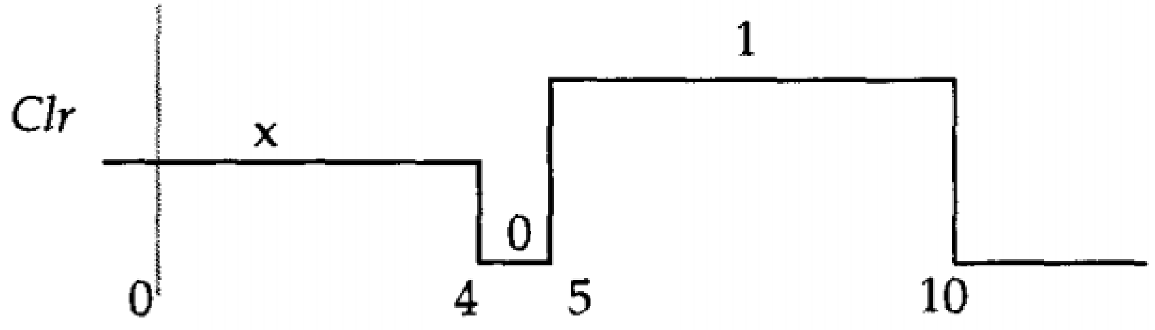
```
begin
   Load <= 32;
   RegA <= Load;
   RegB <= Store;
end
```

- In the above example, let us assume that the sequential block executes at time 10. The first statement causes the value 32 to be assigned to Load at the end of time 10
- Then the 2nd statement executes, the old value of Load is used (note that time has not advanced and Load in the first assignment has not yet been assigned a new value); the assignment to RegA is scheduled at the end of time step 10
- The next statement executes and RegB is scheduled to be assigned a value at the end of time 10.
- After all events at time 10 have occurred, all scheduled assignments to the left-hand side target are made.

Example 2:

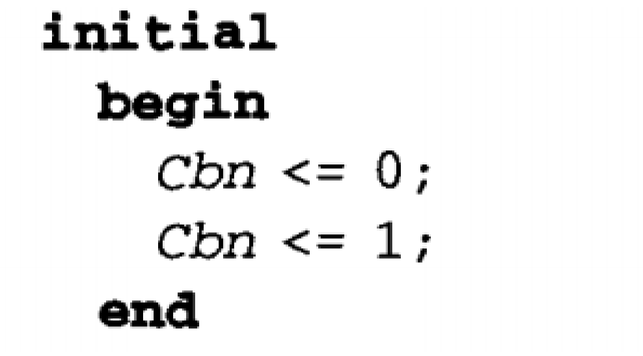


```
initial
  begin
    Clr <= #5 1;
    Clr <= #4 0;
    Clr <= #10 0;
  end
```



- The execution of the first statement causes a 1 to be scheduled to appear on Clr at 5 time units, the execution of the second statement causes Clr to get a value 0 at 4 time units (4 time units from time 0), and finally the execution of the third statement causes a 0 to be scheduled on Clr at 10 time units (10 time units from 0).
- Note that all the three statements execute at time 0.
- In addition, notice that the order of execution of non-blocking assignments become irrelevant in this case.

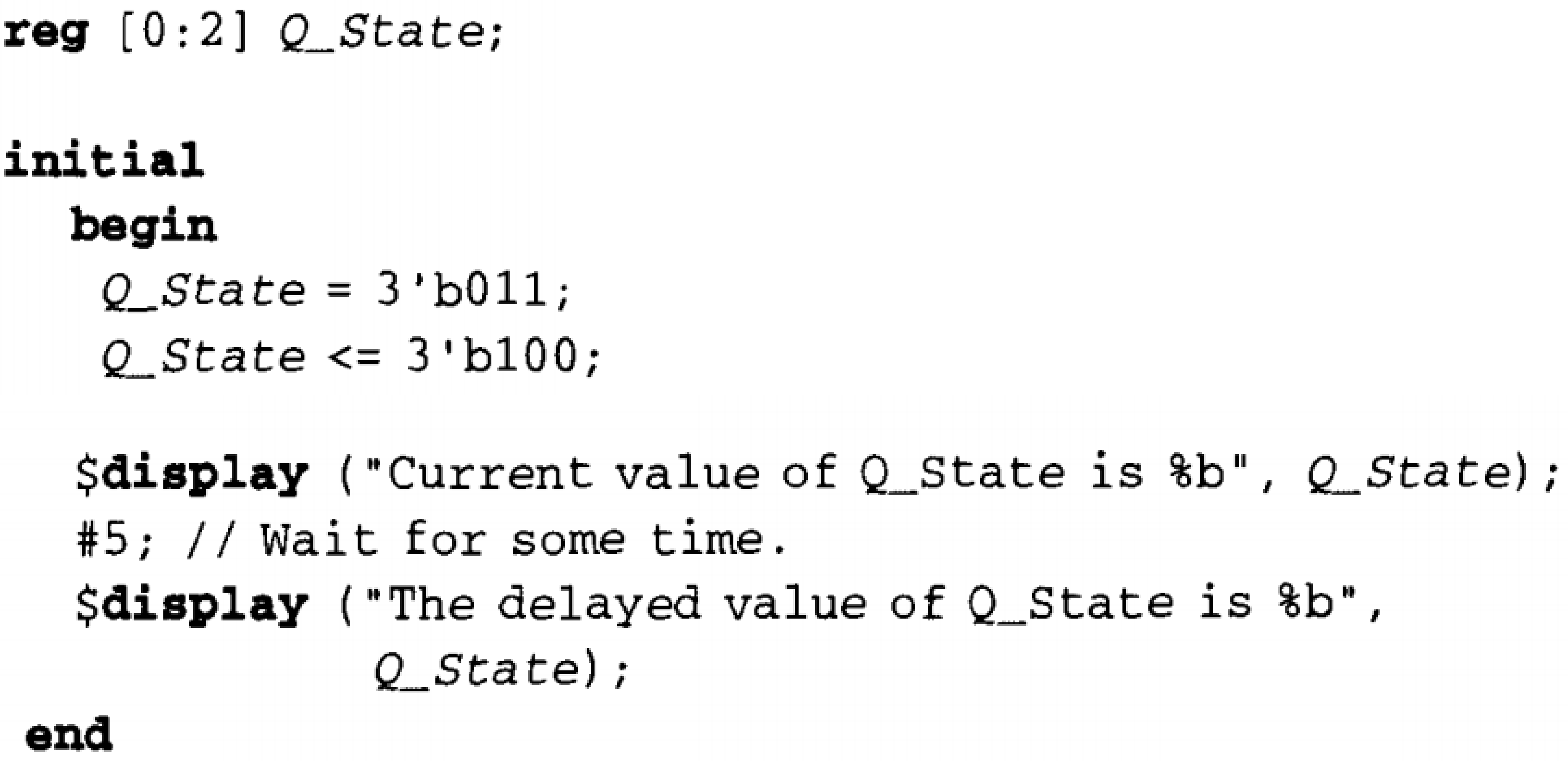
Example 3: zero delay



```
initial
  begin
    Cbn <= 0;
    Cbn <= 1;
  end
```

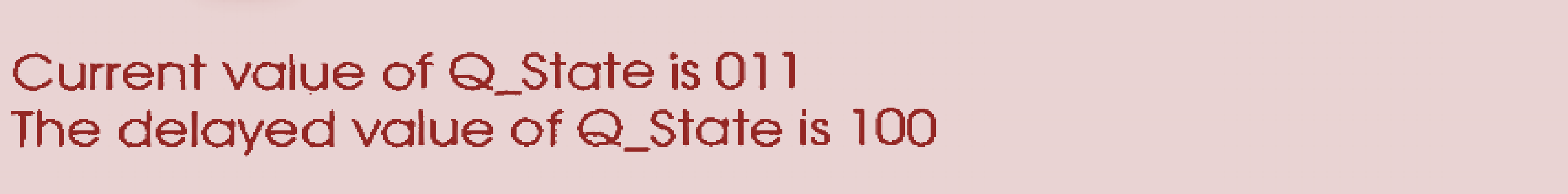
□ The value of Cbn after the initial statement executes is 1 since the Verilog HDL standard specifies that all non-blocking assignments to a variable shall occur in the order the assignment statements are executed. Thus, Cbn gets the value 0 first and then 1.

Q. For the following statements compute the results after the execution of the initial statement



```
reg [0:2] Q_State;

initial
  begin
   Q_State = 3'b011;
   Q_State <= 3'b100;

  $display ("Current value of Q_State is %b", Q_State);
  #5; // Wait for some time.
  $display ("The delayed value of Q_State is %b",
             Q_State);
 end
```

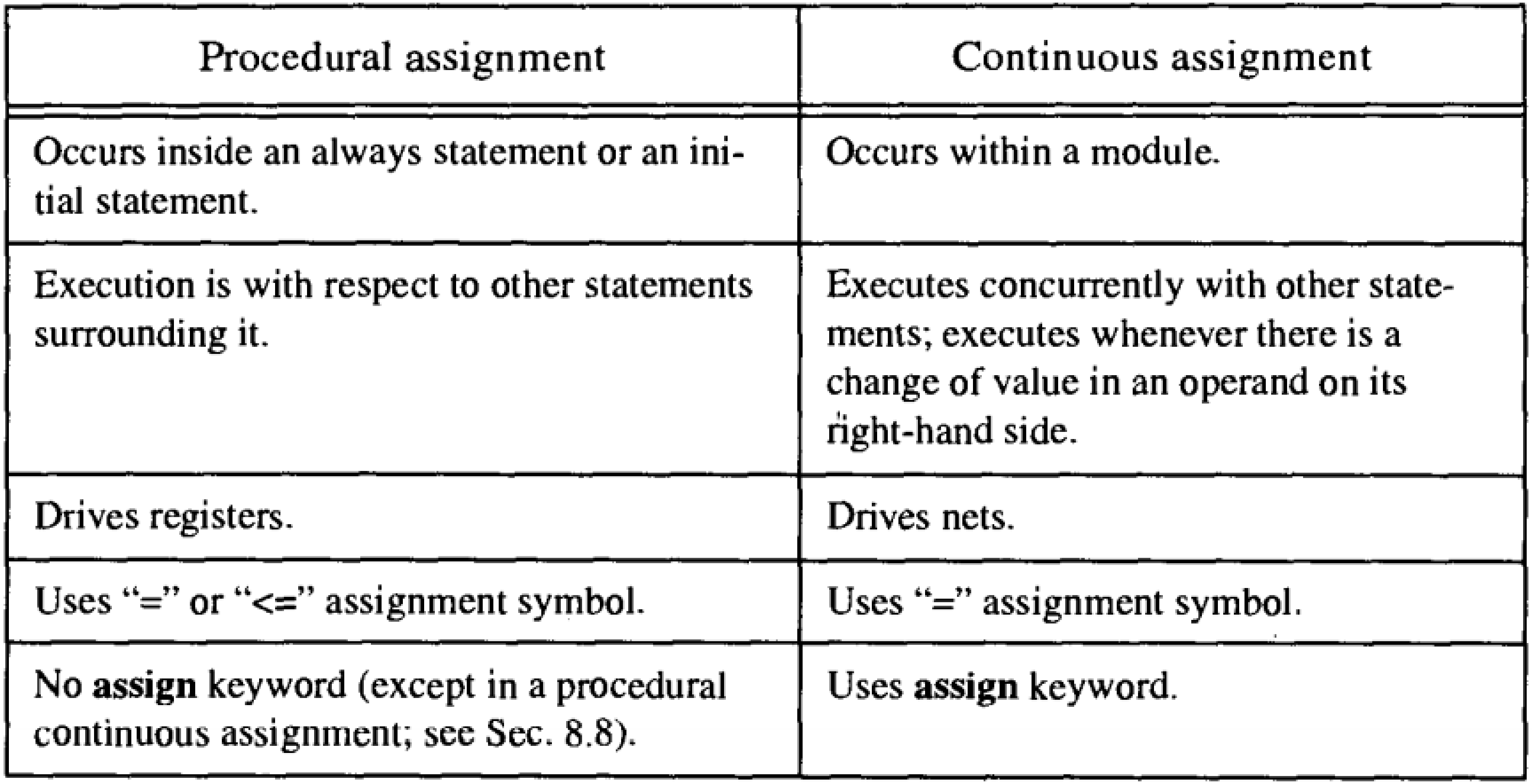
Solution: The execution of the initial statement produces the result:



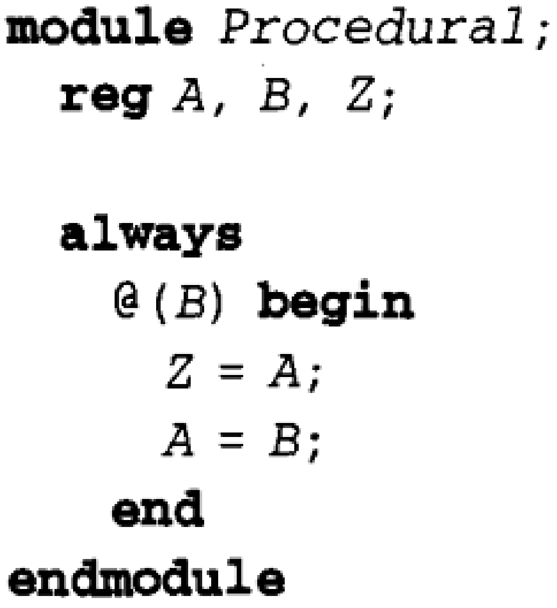
```
Current value of Q_State is 011
The delayed value of Q_State is 100
```

- □ *The first blocking assignment causes Q_State to get the value of 3'b011.*
- □ *The execution of the second assignment statement, which is a non-blocking one, causes the value 3'bl00 to be scheduled for Q_State at the end of the current time step (which is 0).*
- □ *Therefore when the first $display task is executed, Q_State still has the value from the first assignment, which is 3'b011.*
- □ *When the #5 delay is executed, this causes the scheduled assignment of Q_State to occur, Q_State gets updated with its new value, a delay of 5 time units occurs and then the next $display task is executed, this time displaying the updated value of Q_State*

Continuous Assignment vs Procedural Assignment

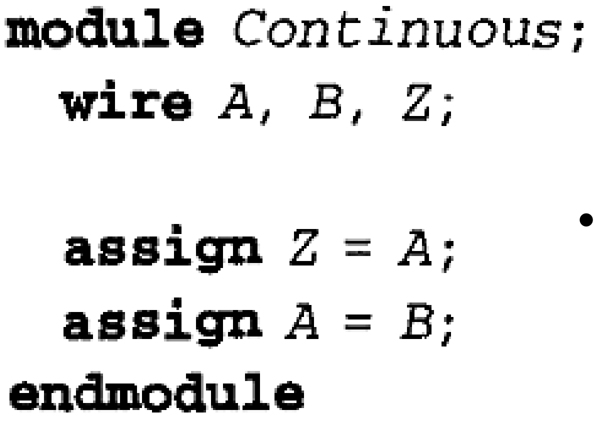


Procedural assignment	Continuous assignment
Occurs inside an always statement or an initial statement.	Occurs within a module.
Execution is with respect to other statements surrounding it.	Executes concurrently with other statements; executes whenever there is a change of value in an operand on its right-hand side.
Drives registers.	Drives nets.
Uses "=" or "<=" assignment symbol.	Uses "=" assignment symbol.
No **assign** keyword (except in a procedural continuous assignment; see Sec. 8.8).	Uses **assign** keyword.



```
module Procedural;
  reg A, B, Z;

  always
    @(B) begin
      Z = A;
      A = B;
    end
endmodule
```

- *Say that B has an event at time 10ns. In module Procedural, the two procedural statements are executed sequentially and A gets the new value of B at 10ns. Z does not get the value of B since the assignment to Z occurs before the assignment to A.*
- *However, if an event occurred on A, the always statement in module Procedural does not execute since A is not in the timing control event list for that always statement.*



```
module Continuous;
  wire A, B, Z;

  assign Z = A;
  assign A = B;
endmodule
```

- *In module Continuous, the second continuous assignment is triggered since there is an event on B. This in turn causes an event on A, which causes the first continuous assignment to be executed, which in turn causes Z to get the value of A which is really B.*
- *However the first continuous assignment in the module Continuous executes and Z gets the new value of A.*

THANK YOU