Chapter 8: Verilog (Part 2)

In Chap. 5 Verilog was presented as a way to describe the behavior of concurrent systems. The modeling techniques presented were appropriate for combinational logic because these types of circuits have outputs dependent only on the current values of their inputs. This means a model that continuously performs signal assignments provides an accurate model of this circuit behavior. In Chap. 7 sequential logic storage devices were presented that did not continuously update their outputs based on the instantaneous values of their inputs. Instead, sequential storage devices only update their outputs based upon an event, most often the edge of a clock signal. The modeling techniques presented in Chap. 5 are unable to accurately describe this type of behavior. In this chapter, we describe the Verilog constructs to model signal assignments that are triggered by an event in order to accurately model sequential logic. We can then use these techniques to describe more complex sequential logic circuits such as finite state machines and register transfer level systems. This chapter will also present how to create test benches and look at more advanced features that are commonly used in Verilog to model modern systems. The goal of this chapter is to give an understanding of the full capability of hardware description languages.

Learning Outcomes—After completing this chapter, you will be able to:

8.1 Describe the behavior of Verilog procedural assignment and how they are used to model sequential logic circuits.
8.2 Model combinational logic circuits using a Verilog procedural assignment and conditional programming constructs.
8.3 Describe the functionality of common Verilog system tasks.
8.4 Design a Verilog test bench to verify the functional operation of a system.

8.1 Procedural Assignment

Verilog uses *procedural assignment* to model signal assignments that are based on an event. An *event* is most commonly a transition of a signal. This provides the ability to model sequential logic circuits such as D-flip-flops and finite state machines by triggering assignments off of a clock edge. Procedural assignments can only drive variable data types (i.e., reg, integer, real, and time), thus they are ideal for modeling storage devices. Procedural signal assignments can be evaluated in the order they are listed, thus they are able to model sequential assignments.

A procedural assignment can also be used to model combinational logic circuits by making signal assignments when any of the inputs to the model change. Despite the left-hand-side of the assignment not being able to be of type wire in procedural assignment, modern synthesizers will recognize properly designed combinational logic models and produce the correct circuit implementation. Procedural assignment also supports standard programming constructs such as if-else decisions, case statements, and loops. This makes procedural assignment a powerful modeling approach in Verilog and is the most common technique for designing digital systems and creating test benches.

8.1.1 Procedural Blocks

All procedural signal assignments must be enclosed within a procedural *block*. Verilog has two types of procedural blocks, *initial* and *always*.

B.J. LaMeres, *Introduction to Logic Circuits & Logic Design with Verilog*,
DOI 10.1007/978-3-319-53883-9_8

8.1.1.1 Initial Blocks

An initial block will execute all of the statements embedded within it one time at the beginning of the simulation. An initial block is not used to model synthesizable behavior. It is instead used within test benches to either set the initial values of repetitive signals or to model the behavior of a signal that only has a single set of transitions. The following is the syntax for an initial block.

```
initial
  begin                  // an optional ": name" can be added after the begin keyword
    signal_assignment_1
    signal_assignment_2
            :
  end
```

Let's look at a simple model of how an initial block is used to model the reset line in a test bench. In the following example, the signal "Reset_TB" is being driven into a DUT. At the beginning of the simulation, the initial value of Reset_TB is set to a logic zero. The second assignment will take place after a delay of 15 time units. The second assignment statement sets Reset_TB to a logic one. The assignments in this example are evaluated in sequence in the order they are listed due to the delay operator. Since the initial block executes only once, Reset_TB will stay at the value of its last assignment for the remainder of the simulation.

Example:

```
initial
  begin
        Reset_TB = 1'b0;
    #15 Reset_TB = 1'b1;
  end
```

8.1.1.2 Always Blocks

An *always* block will execute forever, or for the duration of the simulation. An always block can be used to model synthesizable circuits in addition to non-synthesizable behavior in test benches. The following is the syntax for an always block.

```
always
  begin
   signal_assignment_1
   signal_assignment_2
         :
  end
```

Let's look at a simple model of how an always block can be used to model a clock line in a test bench. In the following example, the value of the signal Clock_TB will continuously change its logic value every 10 time units.

Example:

```
always
  begin
    #10 Clock_TB = ~Clock_TB;
  end
```

By itself, the above always block will not work because when the simulation begins, Clock_TB does not have an initial value so the simulator will not know what the value of Clock_TB is at time zero. It will also not know what the output of the negation operation (~) will be at time unit 10. The following example shows the correct way of modeling a clock signal using a combination of initial and always

blocks. Verilog allows assignments to the same variable from multiple procedural blocks, so the following example is valid. Note that when the simulation begins, Clock_TB is assigned a logic zero. This provides a known value for the signal at time zero and also allows the always block negation to have a deterministic value. The example below will create a clock signal that will toggle every 10 time units.

Example:

```
initial
  begin
    Clock_TB = 1'b0;
  end

always
  begin
    #10 Clock_TB = ~Clock_TB;
  end
```

8.1.1.3 Sensitivity Lists

A *sensitivity list* is used in conjunction with a procedural block to trigger when the assignments within the block are executed. The symbol @ is used to indicate a sensitivity list. Signals can then be listed within parenthesis after the @ symbol that will trigger the procedural block. The following is the base syntax for a sensitivity list.

```
always @ (signal1, signal2)
  begin
    signal_assignment_1
    signal_assignment_2
          :
  end
```

In this syntax, any transition on any of the signals listed within the parenthesis will cause the always block to trigger and all of its assignments to take place one time. After the always block ends, it will await the next signal transition in the sensitivity list to trigger again. The following example shows how to model a simple 3-input AND gate. In this example, any transition on inputs A, B, or C will cause the block to trigger and the assignment to F to occur.

Example:

```
always @ (A, B, C)
  begin
    F = A & B & C;
  end
```

Verilog also supports keywords to limit triggering of the block to only rising edge or falling edge transitions. The keywords are **posedge** and **negedge**. The following is the base syntax for an edge sensitive block. In this syntax, only rising edge transitions on signal1 or falling edge transitions on signal2 will cause the block to trigger.

```
always @ (posedge signal1, negedge signal2)
  begin
    signal_assignment_1
    signal_assignment_2
             :
  end
```

Sensitivity lists can also contain Boolean operators to more explicitly describe behavior. The following syntax is identical to the syntax above.

```
always @ (posedge signal1 or negedge signal2)
  begin
    signal_assignment_1
    signal_assignment_2
            :
  end
```

The ability to model edge sensitivity allows us to model sequential circuits. The following example shows how to model a simple D-flip-flop.

Example:

```
always @ (posedge Clock)
  begin
    Q = D;   // Note: This model does not include a reset.
  end
```

In Verilog-2001, the syntax to support sensitivity lists that will trigger based on any signal listed on the right-hand-side of any assignment within the block was added. This syntax is **@***. The following example how to use this modeling approach to model a 3-input AND gate.

Example:

```
always @*
  begin
    F = A & B & C;
  end
```

8.1.2 Procedural Statements

There are two kinds of signal assignments that can be used within a procedural block, **blocking** and **non-blocking**.

8.1.2.1 Blocking Assignments

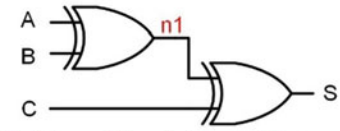
A *blocking assignment* is denoted with the **=** symbol and the evaluation and assignment of each statement takes place immediately. Each assignment within the block is executed in parallel. When this behavior is coupled with a sensitivity list that contains all of the inputs to the system, this approach can model synthesizable combinational logic circuits. This approach provides the same functionality as continuous assignments outside of a procedural block. The reason that designers use blocking assignments instead of continuous assignment is that more advanced programming constructs are supported within Verilog procedural blocks. These will be covered in the next section. Example 8.1 shows how to use blocking assignments within a procedural block to model a combinational logic circuit.

Example: Using Blocking Assignments to Model Combinational Logic

In this model, each of the inputs A, B, and C are listed in the sensitivity list so that the procedural block is triggered on any input transition. When using blocking assignments, the assignments inside of the block are evaluated and executed immediately. These two behaviors allow us to model combinational logic.

```
module BlockingEx1 (output reg  S,
                    input  wire A, B, C);

  reg  n1;

  always @ (A, B, C)
   begin
     n1 = A ^ B;     // statement 1
     S  = n1 ^ C;    // statement 2
   end

endmodule
```

Resulting Circuit



Both statement 1 and statement 2 are treated as separate circuits that execute concurrently when using blocking assignments.

Example 8.1
Using blocking assignments to model combinational logic

8.1.2.2 Non-blocking Assignments

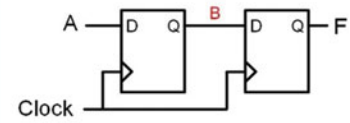
A *non-blocking assignment* is denoted with the <= symbol. When using non-blocking assignments, the assignment to the target signal is deferred until the end of the procedural block. This allows the assignments to be executed in the order they are listed in the block without cascading interim assignments through the list. When this behavior is coupled with triggering the block off of a clock signal, this approach can model synthesizable sequential logic circuits. Example 8.2 shows an example of using non-blocking assignments to model a sequential logic circuit.

Example: Using Non-Blocking Assignments to Model Sequential Logic

In this model, the always block will only trigger on the rising edge of a clock. When using non-blocking assignments, the assignments inside of the block are only executed at the end of the block. These two behaviors allow us to model sequential logic.

```
module NonBlockingEx1 (output reg  F,
                       input  wire A,
                       input  wire Clock);
  reg  B;

  always @ (posedge Clock)
    begin
      B <= A;    // statement 1
      F <= B;    // statement 2
    end

endmodule
```

Resulting Circuit

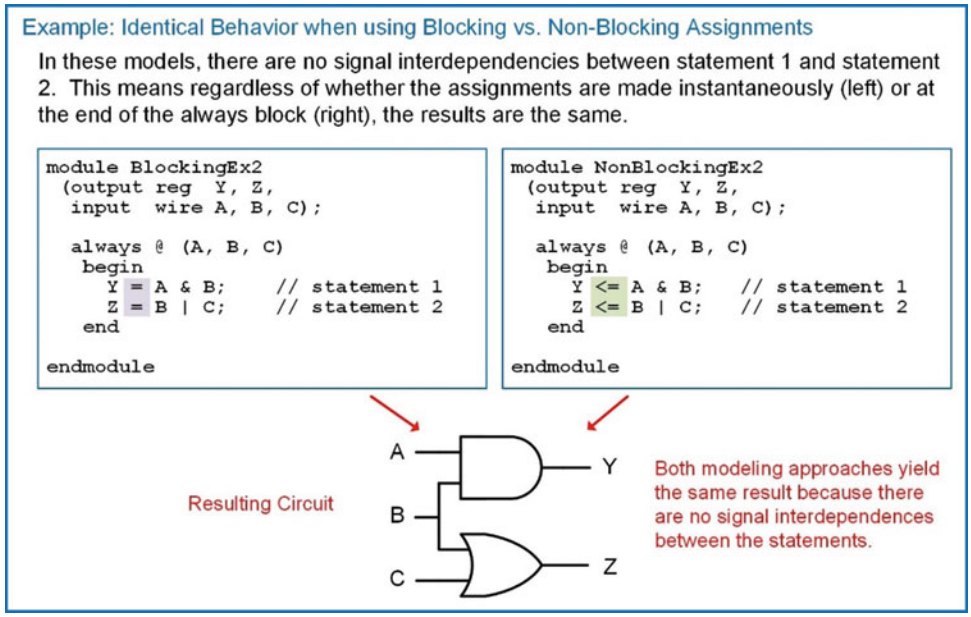


Notice that the value of B in statement 2 is not immediately updated with the assignment made in statement 1 due to the nature of non-blocking assignments.

Example 8.2
Using non-blocking assignments to model sequential logic

The difference between blocking and non-blocking assignments is subtle and is often one of the most difficult concepts to grasp when first learning Verilog. One source of confusion comes from the fact that blocking and non-blocking assignments *can* produce the same results when they either contains a

single assignment or a list of assignments that don't have any signal interdependencies. A *signal interdependency* refers to when a signal that is the target of an assignment (i.e., on the LHS of an assignment) is used as an argument (i.e., on the RHS of an assignment) in subsequent statements. Example 8.3 shows two models that produce the same results regardless of whether a blocking or non-blocking assignment is used.



Example 8.3
Identical behavior when using blocking vs. non-blocking assignments

When a list of statements within a procedural block *does* have signal interdependencies, blocking and non-blocking assignments will have different behavior. Example 8.4 shows how signal interdependencies will cause different behavior between blocking and non-blocking assignments. In this example, all inputs are listed in the sensitivity list with the intent of modeling combinational logic.

Example: Different Behavior when using Blocking vs. Non-Blocking Assignments (1)

In these examples, there is a signal interdependency and all of the inputs are listed in the sensitivity list. By listing all of the inputs in the sensitivity list, the intent is to model combinational logic such that the outputs update anytime there is a change on the inputs.

CORRECT

```
module BlockingEx3
 (output reg  S,
  input  wire A, B, C);

 reg  n1;

 always @ (A, B, C)
  begin
    n1 = A ^ B;   // statement 1
    S  = n1 ^ C;  // statement 2
  end
```

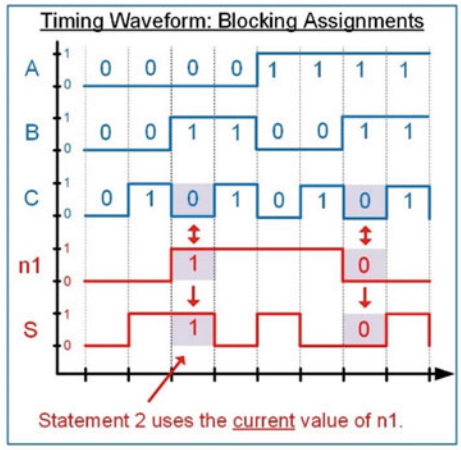
```
module NonBlockingEx3
 (output reg  S,
  input  wire A, B, C);

 reg  n1;

 always @ (A, B, C)
  begin
    n1 <= A ^ B;  // statement 1
    S  <= n1 ^ C; // statement 2
  end
```

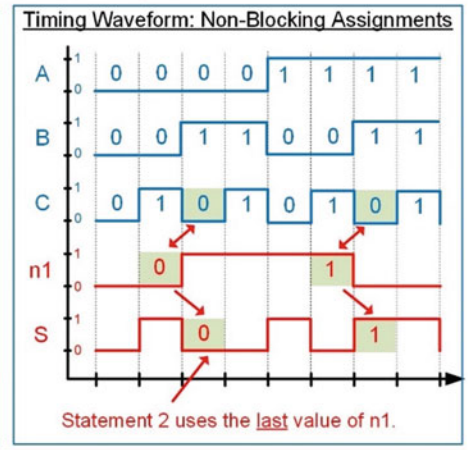
In both cases, statement 1 (the assignment to **n1**) will produce the same result. This is because the assignment only depends on the inputs A and B. Since A and B are listed in the sensitivity list, any change on them will trigger the block and their current value will be used in the assignment to n1.

However, statement 2 (the assignment to **S**) contains a signal interdependency and will NOT produce the same result in both cases.

Blocking Assignment Case (=): Blocking assignments take place immediately. This means that the assignment to n1 in statement 1 takes place immediately and the updated value of n1 is used in statement 2. When used in conjunction with listing all of the inputs in the sensitivity list, this approach successfully models **combinational logic**.

Non-Blocking Assignment Case (<=): Non-blocking assignments take place at the end of the procedural block. This means that the assignment to n1 in statement 1 does not take place before the assignment in statement 2. The value of n1 used in statement 2 will be the value of n1 when the block is triggered, not the new value of n1 assigned within the block. Said another way, the value of n1 used in statement 2 will be the value of n1 from the "prior" time the block triggered, or the "last" value of n1. This does **not model combinational logic**.





Example 8.4
Different behavior when using blocking vs. non-blocking assignments (1)

Example 8.5 shows another case where signal interdependencies will cause different behavior between blocking and non-blocking assignments. In this example, the procedural block is triggered by the rising edge of a clock signal with the intent of modeling two stages of sequential logic.

Example: Different Behavior when using Blocking vs. Non-Blocking Assignments (2)

In these examples, there is a signal interdependency and the sensitivity list is triggered by the edge of a clock. The intent of this model is to create a 2-stage sequential logic circuit such that the outputs only update when there is a rising edge of the clock.

```
module BlockingEx4
 (output reg  F,
  input  wire A,
  input  wire Clock);

  reg  B;

  always @ (posedge Clock)
    begin
      B = A;    // statement 1
      F = B;    // statement 2
    end

endmodule
```

CORRECT

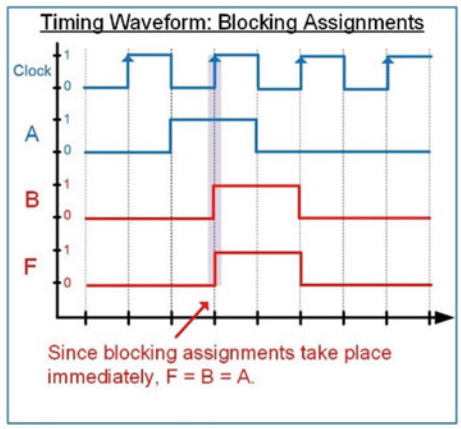
```
module NonBlockingEx4
 (output reg  F,
  input  wire A,
  input  wire Clock);

  reg  B;

  always @ (posedge Clock)
    begin
      B <= A;    // statement 1
      F <= B;    // statement 2
    end

endmodule
```

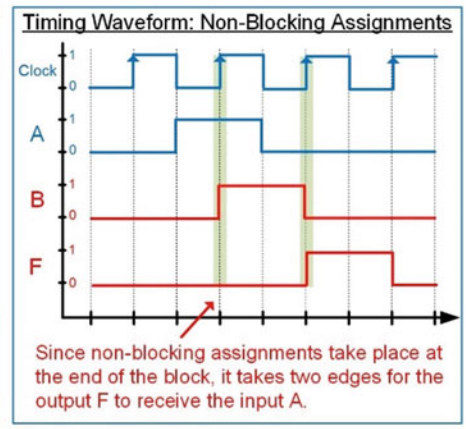
In both cases, the procedural block will trigger on the rising edge of a clock. Also in each case, the assignment in statement 1 will produce the same result.

However, statement 2 has a signal dependency and will NOT produce the same result in both cases.

Blocking Assignment Case (=): Blocking assignments take place immediately. This means that the assignment of A to B and then B to F will result in simply F = A. This will still result in sequential logic, but the output F will be updated with the input A on every rising edge of clock.

Non-Blocking Assignment Case (<=): Non-blocking assignments take place at the end of the procedural block. This means that an input on A will be stored to B on rising edge of clock, but not to F on the same edge. Instead, the subsequent clock edge will cause the result stored in B to be stored to F. This will result in sequential logic that has two edge triggered storage elements instead of just one as in the blocking example.





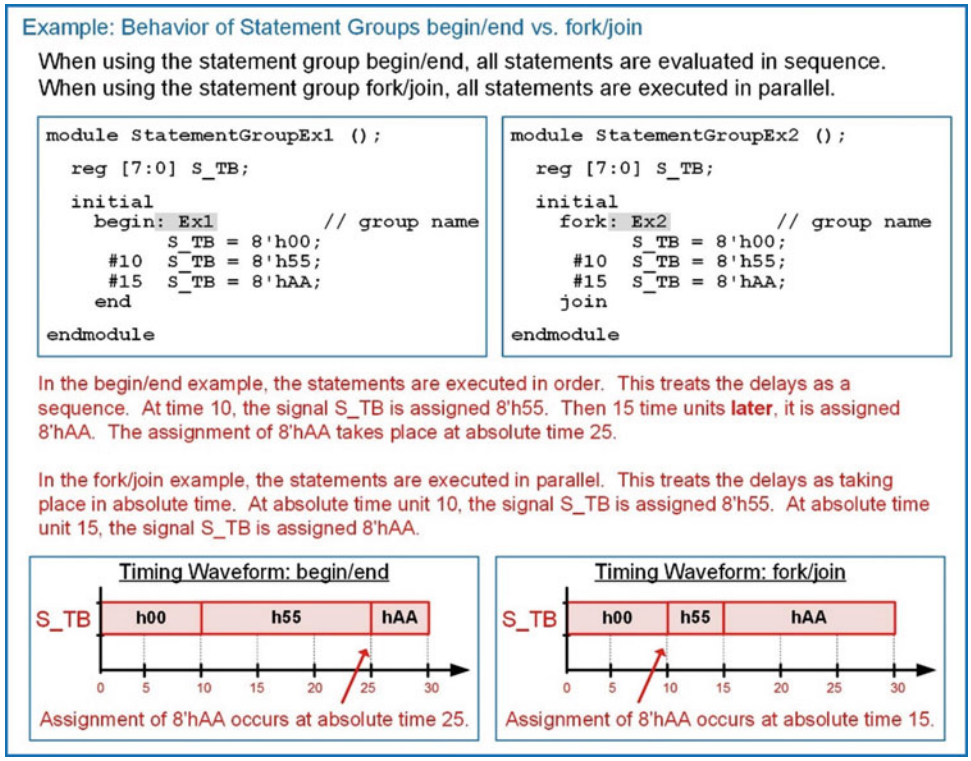
Example 8.5
Different behavior when using blocking vs. non-blocking assignments (2)

While the behavior of these procedural assignments can be confusing, there are two design guidelines that can make creating accurate, synthesizable models straightforward. They are:

1. When modeling **combinational logic**, use blocking assignments and list every input in the sensitivity list.
2. When modeling **sequential logic**, use *non-blocking assignments* and *only list the clock and reset lines* (if applicable) in the sensitivity list.

8.1.3 Statement Groups

A statement group refers to how the statements in a block are processed. Verilog supports two types of statement groups: **begin/end** and **fork/join**. When using begin/end, all statements enclosed within the group will be evaluated in the order they are listed. When using a fork/join, all statements enclosed within the group will be evaluated in parallel. When there is only one statement within procedural block, a statement group is not needed. For multiple statements in a procedural block, a statement group is required. Statement groups can contain an optional name that is appended after the first keyword preceded by a ":". Example 8.6 shows a graphical depiction of the difference between begin/end and fork/join groups. Note that this example also shows the syntax for naming the statement groups.



Example 8.6
Behavior of statement groups begin/end vs. fork/join

8.1.4 Local Variables

Local variables can be declared within a procedural block. The statement group must be named and the variables will not be visible outside of the block. Variables can only be of variable type.

Example:

```
initial
 begin: stim_block   // it is required to name the block when declaring
                        local variables
   integer i;        // local variables can only be of variable type
   i=2;
 end
```

CONCEPT CHECK

CC8.1 If a model of a combinational logic circuit excludes one of its inputs from the sensitivity list, what is the implied behavior?

(A) A storage element because the output will be held at its last value when the unlisted input transitions.

(B) An infinite loop.

(C) A don't care will be used to form the minimal logic expression.

(D) Not applicable because this syntax will not compile.

8.2 Conditional Programming Constructs

One of the more powerful features that procedural blocks provide in Verilog is the ability to use conditional programming constructs such as if-else decisions, case statements, and loops. These constructs are only available within a procedural block and can be used to model both combinational and sequential logic.

8.2.1 if-else Statements

An *if-else* statement provides a way to make conditional signal assignments based on Boolean conditions. The **if** portion of statement is followed by a Boolean condition that if evaluated TRUE will cause the signal assignment listed after it to be performed. If the Boolean condition is evaluated FALSE, the statements listed after the **else** portion are executed. If multiple statements are to be executed in either the if or else portion, then the statement group keywords begin/end need to be used. If only one statement is to be executed, then the statement group keywords are not needed. The else portion of the statement is not required and if omitted, no assignment will take place when the Boolean condition is evaluated FALSE. The syntax for an if-else statement is as follows:

```
if (<boolean_condition>)
   true_statement
else
   false_statement
```

The syntax for an if-else statement with multiple true/false statements is as follows:

```
if (<boolean_condition>)
  begin
    true_statement_1
    true_statement_2
  end
else
  begin
    false_statement_1
    false_statement_2
  end
```

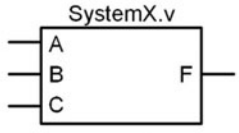
If more than one Boolean condition is required, additional if-else statements can be embedded within the *else* clause of the preceding if statement. The following shows an example of if-else statements implementing two Boolean conditions.

```
if (<boolean_condition_1>)
   true_statement_1
else if (<boolean_condition_2>)
   true_statement_2
else
   false_statement
```

Let's look at using an if-else statement to describe the behavior of a combinational logic circuit. Recall that a combinational logic circuit is one in which the output depends on the instantaneous values of the inputs. This behavior can be modeled by placing all of the inputs to the circuit in the sensitivity list of an always block and using blocking assignments. Using this approach, a change on any of the inputs in the sensitivity list will trigger the block and the assignments will take place immediately. Example 8.7 shows how to model a 3-input combinational logic circuit using if-else statements within a procedural always block.

Example: Using If-Else Statements to Model Combinational Logic

Implement the following truth table using an if-else statement within a procedural block.



A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

```
module SystemX
  (output reg F,
   input  wire A, B, C);

  always @ (A, B, C)
    begin
      if       (A==1'b0 && B==1'b0 && C==1'b0)
        F = 1'b1;
      else if (A==1'b0 && B==1'b1 && C==1'b0)
        F = 1'b1;
      else if (A==1'b1 && B==1'b1 && C==1'b0)
        F = 1'b1;
      else
        F = 1'b0;
    end

endmodule
```

When modeling combinational logic using a procedural assignment, all of the inputs to the circuit must be listed in the sensitivity list and blocking assignments are used.

In this model, three nested if-else statements are used to explicitly describe the input conditions corresponding to an output of a one. For all other input codes, the else clause is used to drive the output to a zero.

Example 8.7
Using if-else statements to model combinational logic

8.2.2 case Statements

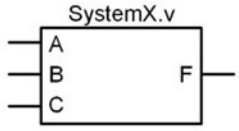
A *case* statement is another technique to model signal assignments based on Boolean conditions. As with the if-else statement, a case statement can only be used inside of a procedural block. The statement begins with the keyword **case** followed by the input signal name that assignments will be based off of enclosed within parenthesis. The case statement can be based on multiple input signal names by concatenating the signals within the parenthesis. Then a series of input codes followed by the corresponding assignment is listed. The keyword **default** can be used to provide the desired signal assignment for any input codes not explicitly listed. When multiple input conditions have the same assignment statement, they can be listed on the same line comma-delimited to save space. The keyword **endcase** is used to denote the end of the case statement. The following is the syntax for a case statement.

```
case (<input_name>)
   input_val_1 : statement_1
   input_val_2 : statement_2
                    :
   input_val_n : statement_n
   default     : default_statement
endcase
```

Example 8.8 shows how to model a 3-input combinational logic circuit using a case statement within a procedural block. Note in this example the inputs are scalars so they must be concatenated so that the input values can be listed as 3-bit vectors. In this example, there are three versions of the model provided. The first explicitly lists out all binary input codes. This approach is more readable because it mirrors a truth table form. The second approach only lists the input codes corresponding to an output of one and uses the default clause to handle all other input codes. The third approach shows how to list multiple input codes with the same assignment on the same line using a comma-delimited series.

Example: Using Case Statements to Model Combinational Logic

Implement the following truth table using a case statement within a procedural block.



A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

```
module SystemX
  (output reg F,
   input  wire A, B, C);

  always @ (A, B, C)
    begin
      case ( {A,B,C} )
          3'b000  : F = 1'b1;
          3'b001  : F = 1'b0;
          3'b010  : F = 1'b1;
          3'b011  : F = 1'b0;
          3'b100  : F = 1'b0;
          3'b101  : F = 1'b0;
          3'b110  : F = 1'b1;
          3'b111  : F = 1'b0;
          default : F = 1'bX;
      endcase
    end

endmodule
```

In this model, each binary input code is explicitly listed to create a model that mirrors the truth table format. Note that since the inputs are scalars, they must be concatenated so that 3-bit vectors can be listed as the input conditions. A default condition is needed to provide the output assignment for input codes containing X or Z.

Below are two alternative approaches of using a case statement that are more compact.

```
case ( {A,B,C} )
   3'b000  : F = 1'b1;
   3'b010  : F = 1'b1;
   3'b110  : F = 1'b1;
   default : F = 1'b0;
endcase
```

In this approach, only the input codes corresponding to an output of one are explicitly listed. The default clause is used to handle all other input codes corresponding to an output of zero.

```
case ( {A,B,C} )
    3'b000, 3'b010, 3'b111  : F = 1'b1;
    default                 : F = 1'b0;
endcase
```

In this model, the input codes corresponding to the output assignment of one are listed on the same line, comma-delimited.

Example 8.8
Using case statements to model combinational logic

If-else statements can be embedded within a case statement and, conversely, case statements can be embedded within an if-else statement.

8.2.3 casez and casex Statements

Verilog provides two additional case statements that support don't cares in the input conditions. The **casez** statement allows the symbols? and **Z** to represent a don't care. The **casex** statement extends the casez statement by also interpreting **X** as a don't care. Care should be taken when using the casez and casex statement as it is easy to create unintended logic when using don't cares in the input codes.

8.2.4 forever Loops

A *loop* within Verilog provides a mechanism to perform repetitive assignments infinitely. This is useful in test benches for creating stimulus such as clocks or other periodic signals. We have already covered a looping construct in the form of an always block. An always block provides a loop with a starting condition. Verilog provides additional looping constructs to model more sophisticated behavior. All looping constructs must reside with a procedural block.

The simplest looping construct is the **forever** loop. As with other conditional programming constructs, if multiple statements are associated with the forever loop they must be enclosed within a statement group. If only one statement is used the statement group is not needed. A forever loop within an initial block provides identical behavior as an always loop without a sensitivity loop. It is important to provide a time step event or delay within a forever loop or it will cause a simulation to hang. The following is the syntax for a forever loop in Verilog.

```
forever
  begin
   statement_1
   statement_2
         :
   statement_n
  end
```

Consider the following example of a forever loop that generates a clock signal (CLK) with a period of 10 time units. In this example, the forever loop is embedded within an initial block. This allows the initial value of CLK to be set to zero upon the beginning of the simulation. Once the forever loop is entered, it will execute indefinitely. Notice that since there is only one statement after the forever keyword, a statement group (i.e., begin/end) is not needed.

Example:

```
initial
  begin
    CLK = 0;

    forever
      #10 CLK = ~CLK;

  end
```

8.2.5 while Loops

A **while** loop provides a looping structure with a Boolean condition that controls its execution. The loop will only execute as long as the Boolean condition is evaluated true. The following is the syntax for a Verilog while loop.

```
while (<boolean_condition>)
  begin
   statement_1
   statement_2
         :
   statement_n
  end
```

Let's implement the previous example of a loop that generates a clock signal (CLK) with a period of 10 time units as long as EN = 1. The TRUE Boolean condition for the while loop is EN = 1. When EN = 0, the while loop will be skipped. When the loop becomes inactive, CLK will hold its last assigned value.

Example:

```
initial
  begin
    CLK = 0;

    while (EN == 1)
       #10 CLK = ~CLK;

  end
```

8.2.6 repeat Loops

A **repeat** loop provides a looping structure that will execute a fixed number of times. The following is the syntax for a Verilog repeat loop.

```
repeat (<number_of_loops>)
  begin
   statement_1
   statement_2
         :
   statement_n
  end
```

Let's implement the previous example of a loop that generates a clock signal (CLK) with a period of 10 time units, except this time we'll use a repeat loop to only produce 10 clock transitions, or 5 full periods of CLK.

Example:

```
initial
  begin
    CLK = 0;
    repeat (10)
       #10 CLK = ~CLK;
  end
```

8.2.7 for Loops

A **for** loop provides the ability to create a loop that can automatically update an internal variable. A *loop variable* within a for loop is altered each time through the loop according to a *step assignment*. The starting value of the loop variable is provided using an *initial assignment*. The loop will execute as long as a Boolean condition associated with the loop variable is TRUE. The following is the syntax for a Verilog for loop:

```
for (<initial_assignment>; <Boolean_condition>; <step_assignment>)
  begin
   statement_1
   statement_2
         :
   statement_n
  end
```

The following is an example of creating a simple counter using the loop variable. The loop variable *i* was declared as an integer prior to this block. The signal Count is also of type integer. The loop variable will start at 0 and increment by 1 each time through the loop. The loop will execute as long as i $<$ 15, or 16 times total. For loops allow the loop variable to be used in signal assignments within the block.

Example:

```
initial
  begin
    for (i=0; i<15; i=i+1)
      #10 Count = i;
  end
```

8.2.8 disable

Verilog provides the ability to stop a loop using the keyword **disable**. The disable function only works on named statement groups. The disable function is typically used after a certain fixed amount of time or within a conditional construct such as an if-else or case statement that is triggered by a control signal. Consider the following forever loop example that will generate a clock signal (CLK), but only when an enable (EN) is asserted. When EN = 0, the loop will disable and the simulation will end.

Example:

```
initial
  begin
    CLK = 0;
    forever
       begin: loop_ex
         if (EN == 1)
           #10 CLK = ~CLK;
         else
           disable loop_ex; // The group name to be disabled comes after the
           keyword
       end
  end
```

CONCEPT CHECK

CC8.2 When using an if-else statement to model a combinational logic circuit, is using the *else* clause the same as using *don't cares* when minimizing a logic expression with a K-map?

(A) Yes. The else clause allows the synthesizer to assign whatever output values are necessary in order to create the most minimal circuit.

(B) No. The else clause explicitly states the output values for all input codes not listed in the if portion of the statement. This is the same as filling in the truth table with specific values for all input codes covered by the else clause and the synthesizer will create the logic expression accordingly.

8.3 System Tasks

A system task in Verilog is one that is used to insert additional functionality into a model that is not associated with real circuitry. There are three main groups of system tasks in Verilog: (1) text output; (2) file input/output; and (3) simulation control. All system tasks begin with a **$** and are only used during simulation. These tasks are ignored by synthesizers so they can be included in real circuit models. All system tasks must reside within procedural blocks.

8.3.1 Text Output

Text output system tasks are used to print strings and variable values to the console or transcript of a simulation tool. The syntax follows ANSI C where double quotes (**“”**) are used denote the text string to be printed. Standard text can be entered within the string in addition to variables. Variable can be printed in two ways. The first is to simply list the variable in the system task function outside of the double quotes. In this usage, the default format to be printed will be decimal unless a task is used with a different default format. The second way to print a variable is within a text string. In this usage, a unique code is inserted into the string indicating the format of how to print the value. After the string, a comma separated list of the variable name(s) is listed that corresponds positionally to the codes within the string. The following are the most commonly used text output system tasks.

Task	Description
$display()	Print text string when statement is encountered *and* append a newline.
$displayb()	Same as $display, but default format of any arguments is binary.
$displayo()	Same as $display, but default format of any arguments is octal.
$displayh()	Same as $display, but default format of any arguments is hexadecimal.
$write()	Same as $display, but the string is printed *without* a newline.
$writeb()	Same as $write, but default format of any arguments is binary.
$writeo()	Same as $write, but default format of any arguments is octal.
$writeh()	Same as $write, but default format of any arguments is hexadecimal.
$strobe()	Same as $display, but printing occurs *after* all simulation events are executed.
$strobeb()	Same as $strobe, but default format of any arguments is binary.
$strobeo()	Same as $strobe, but default format of any arguments is octal.
$strobeh()	Same as $strobe, but default format of any arguments is hexadecimal.
$monitor()	Same as $display, but printing occurs when the value of an argument *changes*.
$monitorb()	Same as $monitor, but default format of any arguments is binary.
$monitoro()	Same as $monitor, but default format of any arguments is octal.
$monitoron	Begin tracking argument changes in subsequent $monitor tasks.
$monitoroff	Stop tracking argument changes in subsequent $monitor tasks.

The following is a list of the most common text formatting codes for printing variables within a string.

Code	Format
%b	Binary values
%o	Octal values
%d	Decimal values
%h	Hexadecimal values
%f	Real values using decimal form
%e	Real values using exponential form
%t	Time values
%s	Character strings
%m	Hierarchical name of scope (no argument required when printing)
%l	Configuration library binding (no argument required when printing)

The format letters in these codes are not case sensitive (i.e., %d and %D are equivalent). Each of these formatting codes can also contain information about truncation of leading and trailing digits. Rounding will take place when numbers are truncated. The formatting syntax is as follows:

```
% < number_of_leading_digits > . < number_of_trailing_digits > <format_code_
letter>
```

There are also a set of string formatting and character escapes that are supported for use with the text output system tasks.

Code	Description
\n	Print a new line.
\t	Print a tab.
\”	Print a quote (“).
\\	Print a backslash (\).
%%	Print a percent sign (%).

The following is a set of examples using common text output system tasks. For these examples, assume two variables have been declared and initialized as follow: A = 3 (integer) and B = 45.6789 (real). Recall that Verilog uses 32-bit codes to represent type integer and real.

Example:

```
$display("Hello World");  // Will print: Hello World
$display("A = %b", A);    // This will print: A = 00000000000000000000000000000011
$display("A = %o", A);    // This will print: A = 00000000003
$display("A = %d", A);    // This will print: A = 3
$display("A = %h", A);    // This will print: A = 00000003
$display("A = %4.0b", A); // This will print: A = 0011

$display("B = %f", B);    // This will print: B = 45.678900
$display("B = %2.0f", B); // This will print: B = 46
$display("B = %2.1f", B); // This will print: B = 45.7
$display("B = %2.2f", B); // This will print: B = 45.68

$display("B = %e", B);    // This will print: B = 4.567890e+001
$display("B = %1.0e", B); // This will print: B = 5e+001
$display("B = %1.1e", B); // This will print: B = 4.6e+001
$display("B = %2.2e", B); // This will print: B = 4.57e+001

$write("A is ", A, "\n");  // This will print: A is 3
$writeb("A is ", A, "\n"); // This will print: A is 00000000000000000000000000000011
$writeo("A is ", A, "\n"); // Will print: A is 00000000003
$writeh("A is ", A, "\n"); // Will print: A is 00000003
```

8.3.2 File Input/Output

File I/O system tasks allow a Verilog module to create and/or access data files is the same way files are handled in ANSI C. This is useful when the results of a simulation are a large and need to be stored in a file as opposed to viewing in a waveform or transcript window. This is also useful when complex stimulus vectors are to be read from an external file and driven into a device under test. Verilog supports the following file I/O system task functions:

Task	Description
$fopen()	Opens a file and returns a unique file descriptor.
$fclose()	Closes the file associated with the descriptor.
$fdisplay()	Same as $display but statements are directed to the file descriptor.
$fwrite()	Same as $write but statements are directed to the file descriptor.
$fstrobe()	Same as $strobe but statements are directed to the file descriptor.
$fmonitor()	Same as $monitor but statements are directed to the file descriptor.
$readmemb()	Read binary data from file and insert into previously defined memory array.
$readmemh()	Read hexadecimal data from file and insert into previously defined memory array.

The **$fopen()** function will either create and open, or open an existing file. Each file that is opened is given a unique integer called a *file descriptor* that is used to identify the file in other I/O functions. The integer must be declared prior to the first use of $fopen. A file name argument is required and provided within double quotes. By default, the file is opened for writing. If the file name doesn't exist, it will be created. If the file name does exist, it will be overwritten. An optional *file_type* can be provided that gives specific action for the file opening including opening an existing file and appending to a file. The following are the supported codes for $fopen().

$fopen types	Description
"**r**" or "**rb**"	Open file for reading.
"**w**" or "**wb**"	Create for writing.
"**a**" or "**ab**"	Open for writing and append to the end of file.
"**r+**" or "**r + b**" or "**rb+**"	Open for update, reading or writing file.
"**w+**" or "**w + b**" or "**wb+**"	Create for update.
"**a+**" or "**a + b**" or "**ab+**"	Open or create for update, append to the end of file.

Once a fie is open, data can be written to it using the **$fdisplay()**, **$fwrite()**, **$fstrobe()**, and **$fmonitor()** tasks. These functions require two arguments. The first argument is the file descriptor and the second is the information to be written. The information follows the same syntax as the I/O system tasks. The following example shows how to create a file and write data to it. This example will create a new file called "Data_out.txt" and write two lines of text to it with the values of variables A and B.

Example:

```
integer A = 3;
real    B = 45.6789;
integer FILE_1;

initial
  begin
    FILE_1 = $fopen("Data_out.txt", "w");
    $fdisplay(FILE_1, "A is %d", A);
    $fdisplay(FILE_1, "B is %f", B);
    $fclose(FILE_1);
  end
```

When reading data from a file, the functions **$readmemb()** and **$readmemh()** can be used. These tasks require that a storage array be declared that the contents of the file can be read into. These tasks have two arguments, the first being the name of the file and the second being the name of the storage array to store the file contents into. The following example shows how to read the contents of a file into a storage array called "memory". Assume the file contains eight lines, each containing a 3-bit vector. The vectors start at 000 and increment to 111 and each symbol will be interpreted as binary using the $readmemb() task. The storage array "memory" is declared to be an 8x3 array of type reg. The

$readmemb() task will insert each line of the file into each 3-bit vector location within “memory”. To illustrate how the data is stored, this example also contains a second procedural block that will print the contents of the storage element to the transcript.

Example:

```
reg[2:0] memory[7:0];

initial
  begin: Read_Block
    $readmemb("Data_in.txt", memory);
  end
initial
  begin: Print_Block
    $display("printing memory %b", memory[0]);  // This will print “000”
    $display("printing memory %b", memory[1]);  // This will print “001”
    $display("printing memory %b", memory[2]);  // This will print “010”
    $display("printing memory %b", memory[3]);  // This will print “011”
    $display("printing memory %b", memory[4]);  // This will print “100”
    $display("printing memory %b", memory[5]);  // This will print “101”
    $display("printing memory %b", memory[6]);  // This will print “110”
    $display("printing memory %b", memory[7]);  // This will print “111”
  end
```

8.3.3 Simulation Control and Monitoring

Verilog also provides a set of simulation control and monitoring tasks. The following are the most commonly used tasks in this group.

Task	Description	
$finish()	Finishes simulation and exits.	
$stop()	Halts the simulation and enters an interactive debug mode.	
$time()	Returns the current simulation time as a 64-bit vector.	
$stime()	Returns the current simulation time as a 32-bit integer.	
$realtime()	Returns the current simulation time as a 32-bit real number.	
$timeformat()	Controls the format used by the %t code in print statements.	
	The arguments are: (<unit>, <precision>, <suffix>, <min_field_width>)	
	where:	
	<unit>	0 = 1 sec
		-1 = 100 ms
		-2 = 10 ms
		-3 = 1 ms
		-4 = 100us
		-5 = 10us
		-6 = 1us
		-7 = 100 ns
		-8 = 10 ns
		-9 = 1 ns
		-10 = 100 ps
		-11 = 10 ps
		-12 = 1 ps
		-13 = 100 fs
		-14 = 10 fs
		15 = 1 fs

	<precision > = The number of decimal points to display.
	<suffix > = A string to be appended to time to indicate units.
	<min_field_width > = The minimum number of characters to display.

The following shows an example of how these tasks can be used.

Example:

```
initial
  begin

    $timeformat (-9, 2, "ns", 10);
    $display("Stimulus starting at time: %t", $time);

    #10  A_TB=0; B_TB=0; C_TB=0;
    #10  A_TB=0; B_TB=0; C_TB=1;
    #10  A_TB=0; B_TB=1; C_TB=0;
    #10  A_TB=0; B_TB=1; C_TB=1;
    #10  A_TB=1; B_TB=0; C_TB=0;
    #10  A_TB=1; B_TB=0; C_TB=1;
    #10  A_TB=1; B_TB=1; C_TB=0;
    #10  A_TB=1; B_TB=1; C_TB=1;

    $display("Simulation stopping at time: %t", $time);
  end
```

This example will result in the following statements printed to the simulator transcript:

```
Stimulus starting at time: 0.00 ns
Simulation stopping at time: 80.00 ns
```

CONCEPT CHECK

CC8.3 How can Verilog system tasks be included in synthesizable circuit models when they provide inherently unsynthesizable functionality?

(A) They can't. System tasks can only be used in test benches.

(B) The "$" symbol tells the CAD tool that the task can be ignored during synthesis.

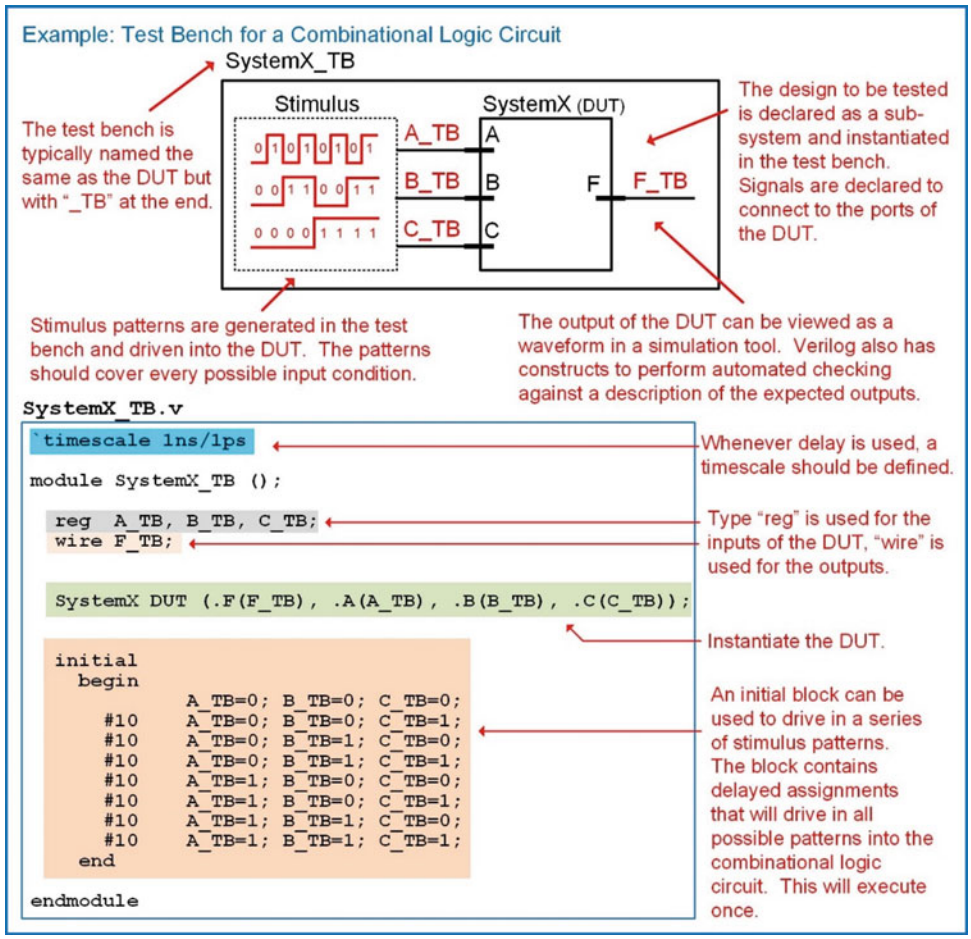
(C) The designer must only use system tasks that model sequential logic.

8.4 Test Benches

The functional verification of Verilog designs is accomplished through simulation using a *test bench*. A test bench is a Verilog model that instantiates the system to be tested as a sub-system, generates the input patterns to drive into the sub-system, and observes the outputs. The system being tested is often called a *device under test (DUT)* or *unit under test (UUT).* Test benches are only used for simulation so they can use abstract modeling techniques that are unsynthesizable to generate the stimulus patterns. Verilog conditional programming constructions and system tasks can also be used to report on the status of a test and also automatically check that the outputs are correct.

8.4.1 Common Stimulus Generation Techniques

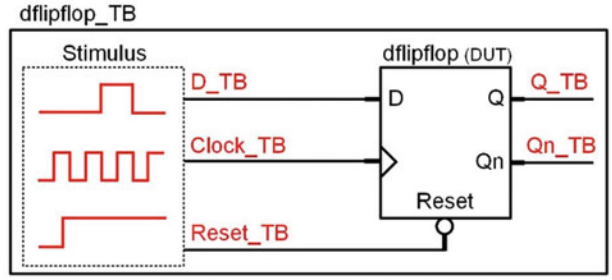
When creating stimulus for combinational logic circuits, it is common to use a procedural block to generate all possible input patterns to drive the DUT and especially any transitions that may cause timing errors. Example 8.9 shows a test bench for a combinational logic circuit where an initial block contains a series of delayed assignments to provide the stimulus to the DUT. This block creates every possible input pattern, delayed by a fixed amount. Note that the initial block will only execute once. If the patterns were desired to repeat indefinitely, an always block without a sensitivity list could be used instead.



Example 8.9
Test bench for a combinational logic circuit

Multiple procedural blocks can be used within a Verilog module to provide parallel functionality. Using both initial and always blocks allows the test bench to drive both repetitive and aperiodic signals. Initial and always blocks can also be used to drive the same signal in order to provide a starting value and a repetitive pattern. Example 8.10 shows a test bench for a rising edge triggered D-flip-flop with an asynchronous, active LOW reset in which multiple procedural blocks are used to generate the stimulus patterns for the DUT.

Example: Test Bench for a Sequential Logic Circuit



In this example, the behavior of each input is modeled using its own procedural block(s).

dflipflop_TB.v

```
`timescale 1ns/1ps

module dflipflop_TB ();

  wire        Q_TB, Qn_TB;
  reg         Clock_TB, Reset_TB, D_TB;

  dflipflop DUT (Q_TB, Qn_TB, Clock_TB, Reset_TB, D_TB);

  initial
    begin
          Reset_TB = 1'b0;
      #15 Reset_TB = 1'b1;
  end

  initial
    begin
      Clock_TB = 1'b0;
    end

  always
    begin
      #10 Clock_TB = ~Clock_TB;
  end

  initial
    begin
          D_TB = 1'b0;
      #55 D_TB = 1'b1;
      #50 D_TB = 1'b0;
  end

endmodule
```

Time unit definition.

Type "reg" is used for the inputs of the DUT, "wire" is used for the outputs.

Instantiate the DUT.

An initial block is used to drive the Reset line. It will only have one transition.

The Clock behavior is modeled using both an initial block and an always block. The initial block assigns the starting value while the always block toggles its value indefinitely.

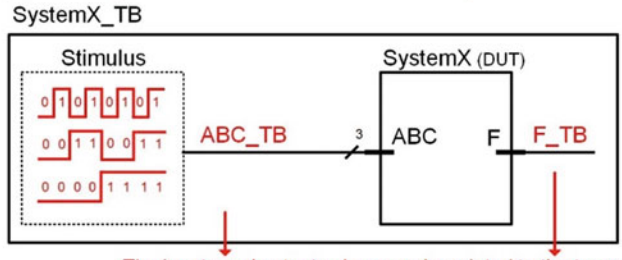
The data (D) behavior is modeled using an initial block to drive specific values at specific times.

Example 8.10
Test bench for a sequential logic circuit

8.4.2 Printing Results to the Simulator Transcript

In the past test bench examples, the input and output values are observed using either the waveform or listing tool within the simulator tool. It is also useful to print the values of the simulation to a transcript window to track the simulation as each statement is processed. Messages can be printed that show the status of the simulation in addition to the inputs and outputs of the DUT using the text output system tasks. Example 8.11 shows a test bench that prints the inputs and output to the transcript of the simulation tool. Note that the test bench must wait some amount of delay before evaluating the output, even if the DUT does not contain any delay.

Example: Printing Test Bench Results to the Transcript

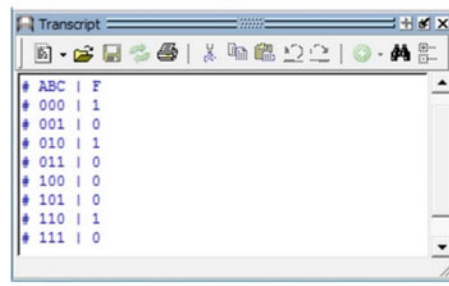


SystemX_TB.v

```
`timescale 1ns/1ps

module SystemX_TB ();

  reg  [2:0] ABC_TB;
  wire       F_TB;


 SystemX DUT (.F(F_TB), .ABC(ABC_TB));

  initial
    begin
                          $display("ABC | F");
        ABC_TB=3'b000;  #1 $display("%b  | %b", ABC_TB, F_TB);
     #9 ABC_TB=3'b001;  #1 $display("%b  | %b", ABC_TB, F_TB);
     #9 ABC_TB=3'b010;  #1 $display("%b  | %b", ABC_TB, F_TB);
     #9 ABC_TB=3'b011;  #1 $display("%b  | %b", ABC_TB, F_TB);
     #9 ABC_TB=3'b100;  #1 $display("%b  | %b", ABC_TB, F_TB);
     #9 ABC_TB=3'b101;  #1 $display("%b  | %b", ABC_TB, F_TB);
     #9 ABC_TB=3'b110;  #1 $display("%b  | %b", ABC_TB, F_TB);
     #9 ABC_TB=3'b111;  #1 $display("%b  | %b", ABC_TB, F_TB);
    end

endmodule
```

Even if the DUT model does not contain delay, the test bench needs to delay before evaluating the output.

The above test bench will print out the following message to the transcript of the simulator tool.



Example 8.11
Printing test bench results to the transcript

8.4.3 Automatic Result Checking

Test benches can also perform automated checking of the results using the conditional programming constructs described earlier in this chapter. Example 8.12 shows an example of a test bench that uses if-else statements to check the output of the DUT and print a PASS/FAIL message to the transcript.

Example: Test Bench with Automatic Output Checking

SystemX_TB.v

```
`timescale 1ns/1ps

module SystemX_TB ();

  reg  [2:0] ABC_TB;
  wire       F_TB;

 SystemX DUT (.F(F_TB), .ABC(ABC_TB));

  initial
    begin
                              $display("ABC | F");

        ABC_TB=3'b000;  #1  $write("%b | %b", ABC_TB, F_TB);
        if(F_TB == 1'b1) $display(" PASS"); else $display(" FAIL");

      #9 ABC_TB=3'b001;  #1  $write("%b | %b", ABC_TB, F_TB);
         if(F_TB == 1'b0) $display(" PASS"); else $display(" FAIL");

      #9 ABC_TB=3'b010;  #1  $write("%b | %b", ABC_TB, F_TB);
         if(F_TB == 1'b1) $display(" PASS"); else $display(" FAIL");

      #9 ABC_TB=3'b011;  #1  $write("%b | %b", ABC_TB, F_TB);
         if(F_TB == 1'b0) $display(" PASS"); else $display(" FAIL");

      #9 ABC_TB=3'b100;  #1  $write("%b | %b", ABC_TB, F_TB);
         if(F_TB == 1'b0) $display(" PASS"); else $display(" FAIL");

      #9 ABC_TB=3'b101;  #1  $write("%b | %b", ABC_TB, F_TB);
         if(F_TB == 1'b0) $display(" PASS"); else $display(" FAIL");

      #9 ABC_TB=3'b110;  #1  $write("%b | %b", ABC_TB, F_TB);
         if(F_TB == 1'b1) $display(" PASS"); else $display(" FAIL");

      #9 ABC_TB=3'b111;  #1  $write("%b | %b", ABC_TB, F_TB);
         if(F_TB == 1'b0) $display(" PASS"); else $display(" FAIL");

    end

endmodule
```

if/else statements check the value of F_TB after each input.

Note that a $write() task is used so that the PASS/FAIL messages are printed on the same line as the I/O values.

This message will be printed to the transcript when the DUT has the correct outputs.

```
# ABC | F
# 000 | 1 PASS
# 001 | 0 PASS
# 010 | 1 PASS
# 011 | 0 PASS
# 100 | 0 PASS
# 101 | 0 PASS
# 110 | 1 PASS
# 111 | 0 PASS
```

The DUT was altered to have the wrong output for input codes 010 and 011.

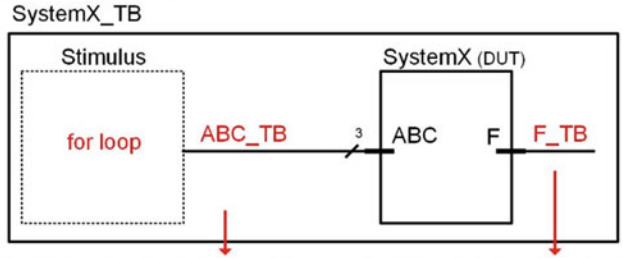
```
# ABC | F
# 000 | 1 PASS
# 001 | 0 PASS
# 010 | 0 FAIL
# 011 | 1 FAIL
# 100 | 0 PASS
# 101 | 0 PASS
# 110 | 1 PASS
# 111 | 0 PASS
```

Example 8.12
Test bench with automatic output checking

8.4.4 Using Loops to Generate Stimulus

When creating stimulus that follow regular patterns such as counting, loops can be an effective way to produce the input vectors. A for loop is especially useful for generating exhaustive stimulus patterns for combinational logic circuits. An integer loop variable can increment within the for loop and then be assigned to the DUT inputs as type reg. Recall that in Verilog, when an integer is assigned to a variable of type reg, it is truncated to matched the size of the reg. This allows a binary count to be created for an input stimulus pattern by using an integer loop variable that increments within a for loop. Example 8.13 shows how the stimulus for a combinational logic circuit can be produced with a for loop.

Example: Using a Loop to Generate Stimulus in a Test Bench



The inputs and output values will be printed to the transcript using $display().

SystemX_TB.v

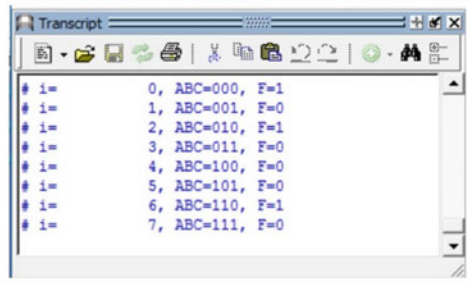
```
`timescale 1ns/1ps

module SystemX_TB ();

  reg  [2:0] ABC_TB;
  wire       F_TB;
  integer    i;

  SystemX DUT (.F(F_TB), .ABC(ABC_TB));

  initial
    begin

      for (i=0; i<8; i=i+1)
        begin
          ABC_TB = i;
          #10 $display("i=%d, ABC=%b, F=%b", i, ABC_TB, F_TB);
        end

    end

endmodulee
```

When using a for loop, the loop variable must be declared.

Each time through the loop, *i* will increment by one. It will count from 0 to 7.

The bottom 3-bits of the integer *i* are used in the assignment to ABC_TB.

The above test bench will print out the following message to the transcript of the simulator tool.

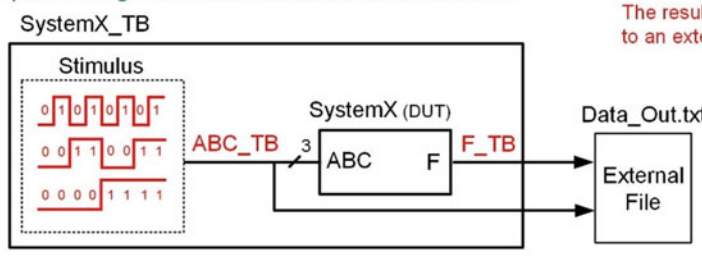


Example 8.13
Using a loop to generate stimulus in a test bench

8.4.5 Using External Files in Test Benches

There are often cases where the results of a test bench need to be written to an external file, either because they are too verbose or because there needs to be a stored record. Verilog allows writing to external files via the file I/O system tasks (i.e., $fdisplay(), $fwrite(), $fstrong(), and $fmonitor()). Example 8.14 shows a test bench in which the input vectors and the output of the DUT are written to an external file using the $fdisplay() system task.

Example: Printing Test Bench Results to an External File



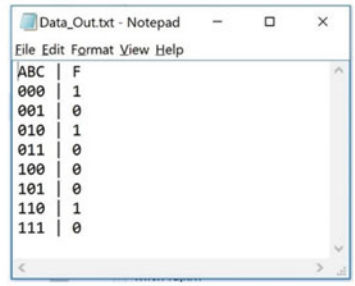
SystemX_TB.v

```
`timescale 1ns/1ps

module SystemX_TB ();

  reg  [2:0] ABC_TB;
  wire       F_TB;
  integer    FILE;

  SystemX DUT (.F(F_TB), .ABC(ABC_TB));

  initial
    begin
      FILE = $fopen("Data_Out.txt");
                          $fdisplay(FILE, "ABC | F");
         ABC_TB=3'b000;  #1  $fdisplay(FILE, "%b | %b", ABC_TB, F_TB);
      #9 ABC_TB=3'b001;  #1  $fdisplay(FILE, "%b | %b", ABC_TB, F_TB);
      #9 ABC_TB=3'b010;  #1  $fdisplay(FILE, "%b | %b", ABC_TB, F_TB);
      #9 ABC_TB=3'b011;  #1  $fdisplay(FILE, "%b | %b", ABC_TB, F_TB);
      #9 ABC_TB=3'b100;  #1  $fdisplay(FILE, "%b | %b", ABC_TB, F_TB);
      #9 ABC_TB=3'b101;  #1  $fdisplay(FILE, "%b | %b", ABC_TB, F_TB);
      #9 ABC_TB=3'b110;  #1  $fdisplay(FILE, "%b | %b", ABC_TB, F_TB);
      #9 ABC_TB=3'b111;  #1  $fdisplay(FILE, "%b | %b", ABC_TB, F_TB);
      $fclose(FILE);
    end

endmodule
```

A unique file descriptor is generated when $fopen() is called. The descriptor is an integer. An integer variable must be setup before calling $fopen().

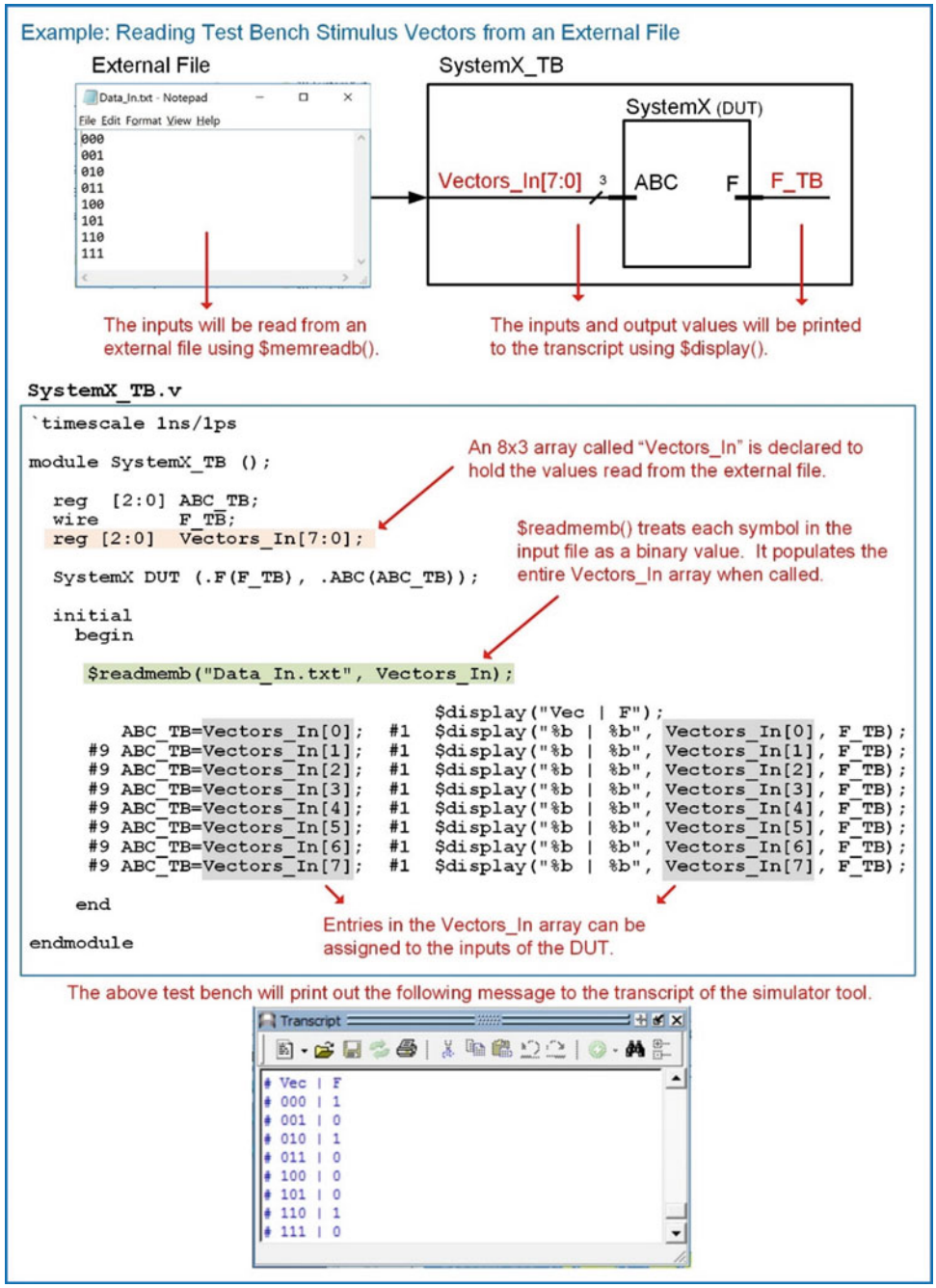
$fdisplay() directs the test strings to a file.

The above test bench will create the file "Data_Out.txt" and print the following text to it.



Example 8.14
Printing test bench results to an external file

It is often the case that the input vectors are either too large to enter manually or were created by a separate program. In either case, a useful technique in test benches is to read input vectors from an external file. Example 8.15 shows an example where the input stimulus vectors for a DUT are read from an external file using the $readmemb() system task.



Example 8.15
Reading test bench stimulus vectors from an external file

Concept Check

CC8.4 Could a test bench ever use always blocks and sensitivity lists exclusively to create its stimulus? Why or why not?

(A) Yes. The signal assignments will simply be made when the block ends.

(B) No. Since a sensitivity list triggers when there is a change on one or more of the signals listed, the blocks in the test bench would never trigger because there is no method to make the initial signal transition.

Summary

- To model sequential logic, an HDL needs to be able to trigger signal assignments based on an event. This is accomplished in Verilog using *procedural assignment*.
- There are two types of procedural blocks in Verilog, *initial* and *always*. An initial block executes one time. An always block runs continually.
- A *sensitivity* list is a way to control when a Verilog procedural block is triggered. A sensitivity list contains a list of signals. If any of the signals in the sensitivity list transitions it will cause the block to trigger. If a sensitivity list is omitted, the block will trigger immediately. Sensitivity lists are most commonly used with always blocks.
- Sensitivity lists and always blocks are used to model synthesizable logic. Initial blocks are typically only used in test benches. Always blocks are also used in test benches.
- There are two types of signal assignments that can be used within a procedural block, blocking and non-blocking.
- A blocking assignment is denoted with the $=$ symbol. All blocking assignments are made immediately within the procedural block. Blocking assignments are used to model combinational logic. Combinational logic models list all input to the circuit in the sensitivity list.
- A non-blocking assignment is denoted with the $<=$ symbol. All non-blocking assignments are made when the procedural block ends and are evaluated in the order they appeared in the block. Blocking assignments are used to model sequential logic. Sequential logic models list only the clock and reset in the sensitivity list.
- Variables can be defined within a procedural block as long as the block is named.
- Procedural blocks allow more advanced modeling constructs in Verilog. These include if-else statements, case statements, and loops.
- Verilog provides numerous looping constructs including forever, while, repeat, and for. Loops can be terminated using the disable keyword.
- System Tasks provide additional functionality to Verilog models. Tasks begin with the $ symbol and are omitted from synthesis. System tasks can be included in synthesizable logic models.
- There are three groups of system tasks: text output, file input/output, and simulation control and monitoring.
- System tasks that perform printing functions can output strings in addition to variable values. Verilog provides a mechanism to print the variable values in a variety of format.
- A test bench is a way to simulate a device under test (DUT) by instantiating it as a sub-system, driving in stimulus, and observing the outputs. Test benches do not have inputs or outputs and are unsynthesizable.
- Test benches for combinational logic typically exercise the DUT under an exhaustive set of stimulus vectors. These include all possible logic inputs in addition to critical transitions that could cause timing errors.
- Text I/O system tasks provide a way to print the results of a test bench to the simulation tool transcript.
- File I/O system tasks provide a way to print the results of a test bench to an external file and also to read in stimulus vectors from an external file.

- Conditional programming constructs can be used within a test bench to perform automatic checking of the outputs of a DUT within a test bench.
- Loops can be used in test benches to automatically generate stimulus patterns. A for loop is a convenient technique to produce a counting pattern.
- Assignment from an integer to a reg in a for loop is allowed. The binary value of the integer is truncated to fit the size of the reg vector.

Exercise Problems

Section 8.1: Procedural Assignment

8.1.1 When using a sensitivity list with a procedural block, what will cause the block to *trigger*?

8.1.2 When a sensitivity list is not used with a procedural block, when will the block trigger?

8.1.3 When are statements executed when using blocking assignments?

8.1.4 When are statements executed when using non-blocking assignments?

8.1.5 When is it possible to exclude statement groups from a procedural block?

8.1.6 What is the difference between a begin/end and fork/join group when each contain multiple statements?

8.1.7 What is the difference between a begin/end and fork/join group when each contain only a single statements?

8.1.8 What type of procedural assignment is used when modeling combinational logic?

8.1.9 What type of procedural assignment is used when modeling sequential logic?

8.1.10 What signals should be listed in the sensitivity list when modeling combinational logic?

8.1.11 What signals should be listed in the sensitivity list when modeling sequential logic?

Section 8.2: Conditional Programming Constructs

8.2.1 Design a Verilog model to implement the behavior described by the 4-input truth table in Fig. 8.1. Use procedural assignment and an if-else statement. Declare the module to match the block diagram provided. Use the type wire for the inputs and type reg for the output. Hint: Notice that there are far more input codes producing F = 0 than producing F = 1. Can you use this to your advantage to make your if-else statement simpler?

A	B	C	D	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

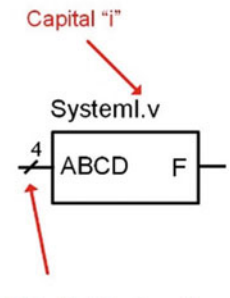


Fig. 8.1
System I functionality

8.2.2 Design a Verilog model to implement the behavior described by the 4-input truth table in Fig. 8.1. Use procedural assignment and a case statement. Declare the module to match the block diagram provided. Use the type wire for the inputs and type reg for the output.

8.2.3 Design a Verilog model to implement the behavior described by the 4-input minterm list in Fig. 8.2. Use procedural assignment and an if-else statement. Declare the module to match the block diagram provided. Use the type wire for the inputs and type reg for the output.

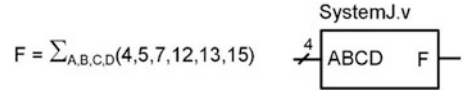


Fig. 8.2
System J functionality

8.2.4 Design a Verilog model to implement the behavior described by the 4-input minterm list in Fig. 8.2. Use procedural assignment and a case statement. Declare the module to match the block diagram provided. Use the type wire for the inputs and type reg for the output.

8.2.5 Design a Verilog model to implement the behavior described by the 4-input maxterm list in Fig. 8.3. Use procedural assignment and an if-then statement. Declare the module to match the block diagram provided. Use the type wire for the inputs and type reg for the output.

Fig. 8.3
System K functionality

8.2.6 Design a Verilog model to implement the behavior described by the 4-input maxterm list in Fig. 8.3. Use procedural assignment and a case statement. Declare the module to match the block diagram provided. Use the type wire for the inputs and type reg for the output.

8.2.7 Design a Verilog model to implement the behavior described by the 4-input truth table in Fig. 8.4. Use procedural assignment and an if-else statement. Declare the module to match the block diagram provided. Use the type wire for the inputs and type reg for the output. Hint: Notice that there are far more input codes producing F = 1 than producing F = 0. Can you use this to your advantage to make your if-else statement simpler?

A	B	C	D	F
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

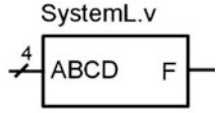


Fig. 8.4
System L functionality

8.2.8 Design a Verilog model to implement the behavior described by the 4-input truth table in Fig. 8.4. Use procedural assignment and a case statement. Declare the module to match the block diagram provided. Use the type wire for the inputs and type reg for the output.

8.2.9 Fig. 8.5 shows the topology of a 4-bit shift register when implemented structurally using D-Flip-Flops. Design a Verilog model to describe this functionality using a single procedural block and non-blocking assignments instead of instantiating D-Flip-Flops. The figure also provides the block diagram for the module port definition. Use the type wire for the inputs and type reg for the outputs.

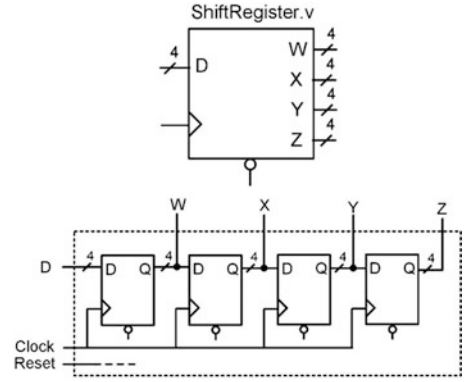


Fig. 8.5
4-bit shift register functionality

8.2.10 Design a Verilog model for a counter using a for loop with an output type of integer. Fig. 8.6 shows the block diagram for the module definition. The counter should increment from 0 to 31 and then start over. Use delay in your loop to update the counter value every 10 ns. Consider using the loop variable of the for loop to generate your counter value.

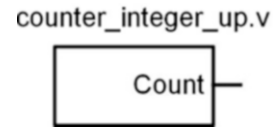


Fig. 8.6
Integer counter block diagram

8.2.11 Design a Verilog model for a counter using a for loop with an output type of reg[4:0]. Fig. 8.7 shows the block diagram for the module definition. The counter should increment from 000002 to 111,112 and then start over. Use delay in your loop to update the counter value every 10 ns. Consider using the loop variable of the for loop to generate an integer version of

your count value, and then assign it to the output variable of type reg[4:0].

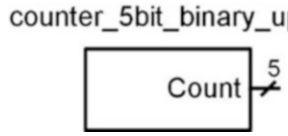


Fig. 8.7
5-bit binary counter block diagram

Section 8.3: System Tasks

8.3.1 Are system tasks synthesizable? Why or why not?

8.3.2 What is the difference between the tasks $display() and $write()?

8.3.3 What is the difference between the tasks $display() and $monitor()?

8.3.4 What is the data type returned by the task $fopen()?

Section 8.4: Test Benches

8.4.1 Design a Verilog test bench to verify the functional operation of the system in Fig. 8.1. Your test bench should drive in every possible input code for the vector ABCD (i.e., "0000", "0001", "0010", . . ., "1111"). Have your test bench change the input pattern every 10 ns using delay within your procedural block.

8.4.2 Design a Verilog test bench to verify the functional operation of the system in Fig. 8.1 with automatic checking. Your test bench should drive in every possible input code for the vector ABCD (i.e., "0000", "0001", "0010", . . ., "1111"). Have your test bench change the input pattern every 10 ns using delay within your procedural block. Use conditional statements to check whether the output of the DUT is correct. For each input vector, print a message using $display() that indicates the current input vector being tested, the resulting output of your DUT, and whether the DUT output is correct.

8.4.3 Design a Verilog test bench to verify the functional operation of the system in Fig. 8.2. Your test bench should drive in every possible input code for the vector ABCD (i.e., "0000", "0001", "0010", . . ., "1111"). Have your test bench change the input pattern every 10 ns using delay within your procedural block.

8.4.4 Design a Verilog test bench to verify the functional operation of the system in Fig. 8.2 with automatic checking. Your test bench should drive in every possible input code for the vector ABCD (i.e., "0000", "0001", "0010", . . ., "1111"). Have your test bench change the input pattern every 10 ns using delay within your procedural block. Use conditional statements to check whether the output of the DUT is correct. For each input vector, print a message using $display() that indicates the current input vector being tested, the resulting output of your DUT, and whether the DUT output is correct.

8.4.5 Design a Verilog test bench to verify the functional operation of the system in Fig. 8.3. Your test bench should drive in every possible input code for the vector ABCD (i.e., "0000", "0001", "0010", . . ., "1111"). Have your test bench change the input pattern every 10 ns using delay within your procedural block.

8.4.6 Design a Verilog test bench to verify the functional operation of the system in Fig. 8.3 with automatic checking. Your test bench should drive in every possible input code for the vector ABCD (i.e., "0000", "0001", "0010", . . ., "1111"). Have your test bench change the input pattern every 10 ns using delay within your procedural block. Use conditional statements to check whether the output of the DUT is correct. For each input vector, print a message using $display() that indicates the current input vector being tested, the resulting output of your DUT, and whether the DUT output is correct.

8.4.7 Design a Verilog test bench to verify the functional operation of the system in Fig. 8.4. Your test bench should drive in every possible input code for the vector ABCD (i.e., "0000", "0001", "0010", . . ., "1111"). Have your test bench change the input pattern every 10 ns using delay within your procedural block.

8.4.8 Design a Verilog test bench to verify the functional operation of the system in Fig. 8.4 with automatic checking. Your test bench should drive in every possible input code for the vector ABCD (i.e., "0000", "0001", "0010", . . ., "1111"). Have your test bench change the input pattern every 10 ns using delay within your procedural block. Use conditional statements to check whether the output of the DUT is correct. For each input vector, print a message using $display() that indicates the current input vector being tested, the resulting output of your DUT, and whether the DUT output is correct.

8.5.9 Design a Verilog test bench to verify the functional operation of the system in Fig. 8.4. Your test bench should drive in every possible input code for the vector ABCD (i.e., "0000", "0001", "0010", . . ., "1111"). Have your test bench change the input pattern every 10 ns using delay within your procedural block. Print the results to an external file named "output_vectors.txt" using $fdisplay().

8.5.10 Design a Verilog test bench that reads in test vectors from an external file to verify the functional operation of the system in Fig. 8.4. Create an input text file called "input_vectors.txt" that contains each input code for the vector ABCD (i.e., "0000", "0001", "0010", . . ., "1111"), each on a separate line in the file. Your test bench should read in the vectors using $readmemb(), drive each code into the DUT, and print the results to the transcript using $display().