Chapter 6: Test Benches

One of the essential components of the modern digital design flow is verifying functionality through simulation. This functional verification is accomplished using a *test bench*. A test bench is a Verilog model that instantiates the system to be tested as a subsystem, generates the input patterns to drive into the subsystem, and observes the outputs. 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. This chapter provides the details of Verilog's built-in capabilities that allow test benches to be created and some examples of automated stimulus generation.

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

6.1 Design a Verilog test bench that manually creates each stimulus pattern using a series of signal assignments within a procedural block.
6.2 Design a Verilog test bench that uses for loops to automatically generate an exhaustive set of stimulus patterns.
6.3 Design a Verilog test bench that automatically checks the outputs of the system being tested using report and assert statements.
6.4 Design a Verilog test bench that uses external I/O as part of the testing procedures including reading stimulus patterns from, and writing the results to, external files.

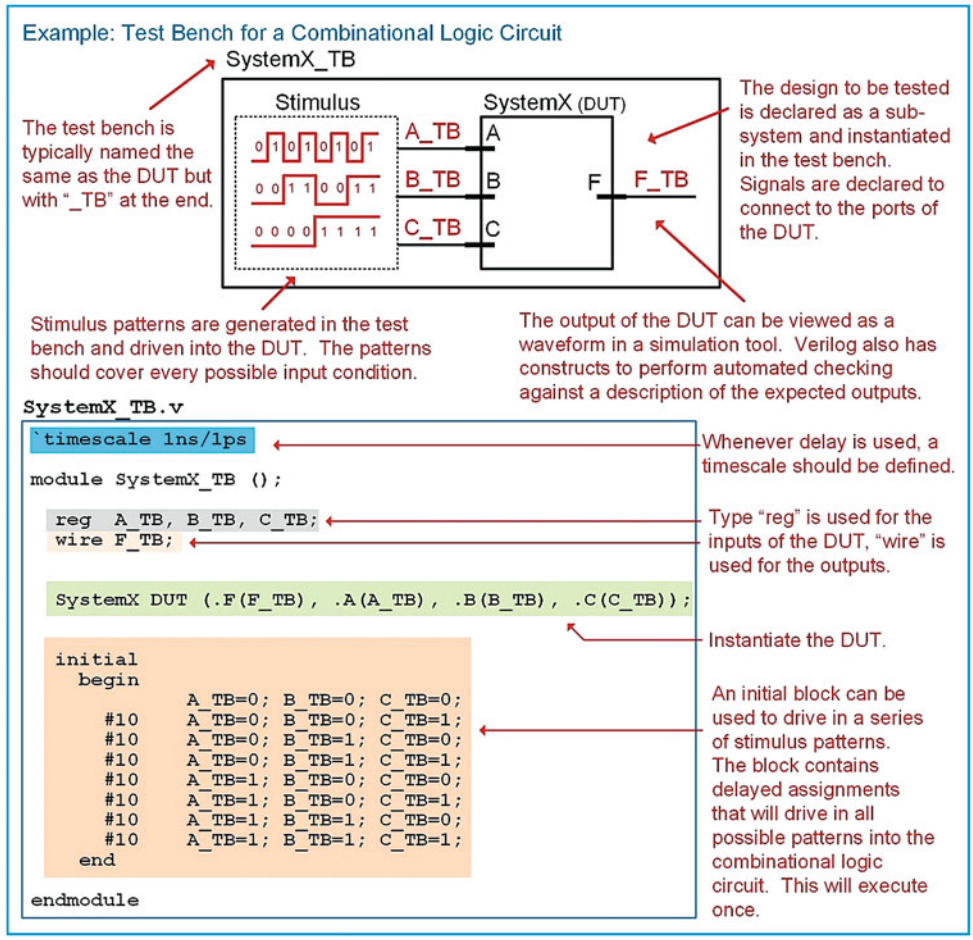
6.1 Test Bench Overview

A test bench is a file in Verilog that has no inputs or outputs. The test bench instantiates the system to be tested as a lower-level module. The system being tested is often called a *device under test (DUT)* or *unit under test (UUT).*

6.1.1 Generating Manual Stimulus

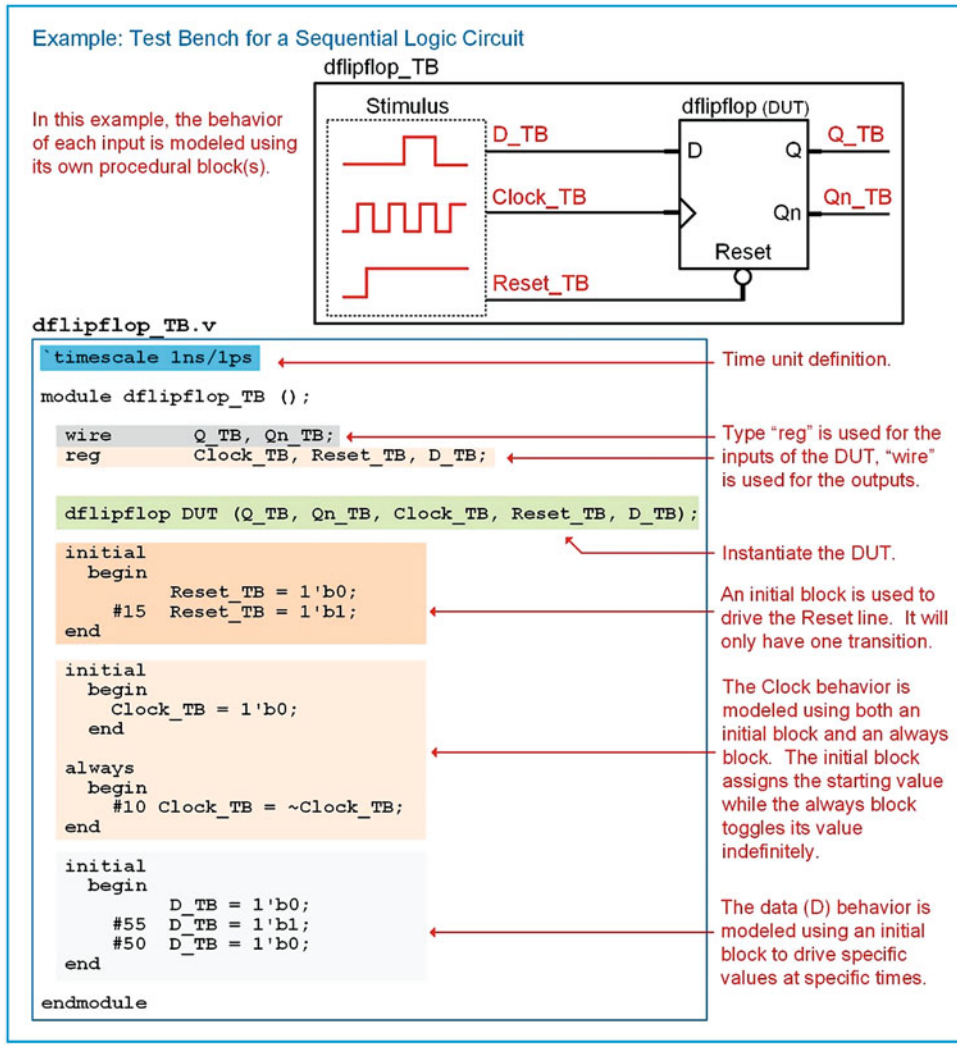
When creating stimulus for combinational logic circuits, it is common to use a procedural block to generate all possible input patterns to drive the DUT and especially any transitions that may cause timing errors. Example 6.1 shows how to create a simple test bench to verify the operation of a DUT called SystemX. The test bench does not have any inputs or outputs; thus, there are no ports declared in the module. SystemX is then instantiated (DUT) in the test bench. Internal signals of type *reg* are declared to connect to the DUT inputs (A_TB, B_TB, C_TB) and an internal signal of type *wire* is declared to connect to the DUT output (F_TB). A procedural block is then used to generate the inputs of SystemX. Within the procedural block, delayed assignments are used to control the timing of the input patterns. In this example, each possible input code is generated within an initial block. The output (F_TB) is observed using a simulation tool in either the form of a waveform or a table listing.

B. J. LaMeres, *Quick Start Guide to Verilog*, https://doi.org/10.1007/978-3-030-10552-5_6



Example 6.1
Test bench for a combinational logic circuit with manual stimulus generation

Multiple procedural blocks can be used within a Verilog test bench to provide parallel stimulus generation. 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 6.2 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.

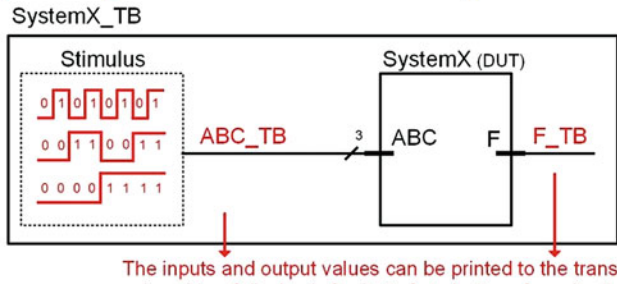


Example 6.2
Test bench for a sequential logic circuit

6.1.2 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 6.3 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

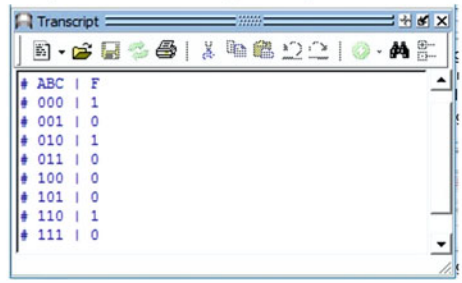


SystemX_TB.v

```
`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 6.3
Printing test bench results to the transcript

CONCEPT CHECK

CC6.1 How can the output of a DUT be verified when it is connected to a signal that does not go anywhere?

(A) It can't. The output must be routed to an output port on the test bench.

(B) The values of any dangling signal are automatically written to a text file.

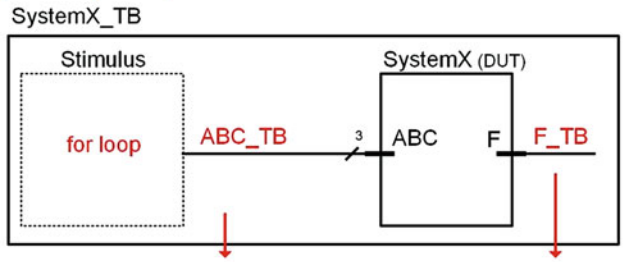
(C) It is viewed in the logic simulator as either a waveform or text listing.

(D) It can't. A signal that does not go anywhere will cause an error during simulation.

6.2 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 match 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 6.4 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

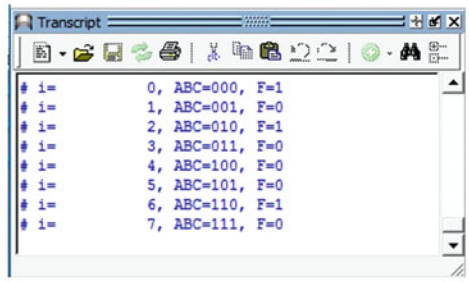
```
`timescale 1ns/1ps

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 6.4
Using a loop to generate stimulus in a test bench

Concept Check

CC6.2 If you used two nested for loops to generate an exhaustive set of patterns for the inputs of an 8-bit adder, how many patterns would be generated? There is no carry-in bit.

(A) 16

(B) 256

(C) 512

(D) 65,536

6.3 Automatic Result Checking

Test benches can also perform automated checking of the results using the conditional programming constructs described earlier in this book. Example 6.5 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 6.5
Test bench with automatic output checking

Concept Check

CC6.3 Will the test bench approach of checking the results and then printing PASS/FAIL to the transcript window stop the simulation?

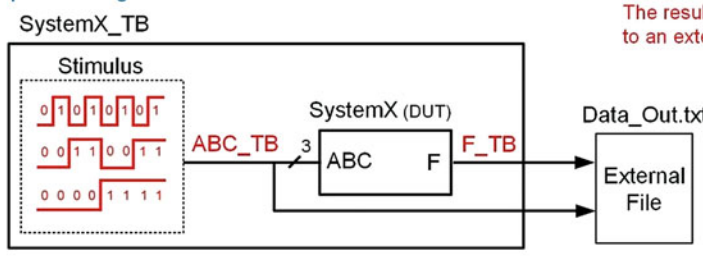
(A) Yes. As soon as “FAIL” is printed, the simulation will halt.

(B) No. The printing of PASS/FAIL is just simple text and doesn’t influence the simulation.

6.4 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 for visual inspection or because there needs to be a stored record of the system’s validation. Verilog allows writing to external files via the file I/O system tasks (i.e., $fdisplay(), $fwrite(), $fstrong(), and $fmonitor()). Example 6.6 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



Test bench values can be printed to a file using either $fdisplay(), $fwrite(), $fstrobe(), or $fmonitor().

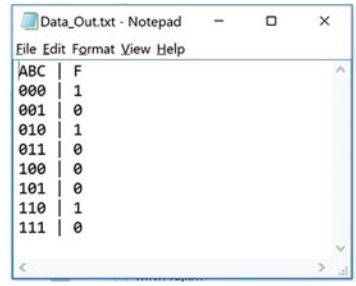
SystemX_TB.v

```
`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.

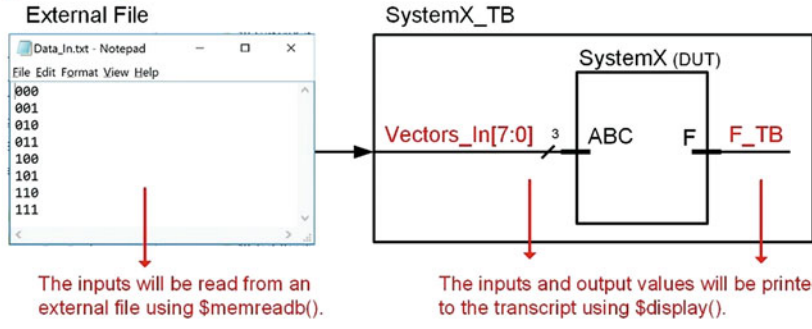
The above test bench will create the file "Data_Out.txt" and print the following text to it.



Example 6.6
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 6.7 shows an example where the input stimulus vectors for a DUT are read from an external file using the $readmemb() system task.

Example: Reading Test Bench Stimulus Vectors from an External File



SystemX_TB.v

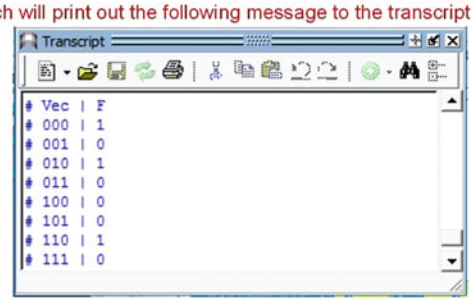
```
`timescale 1ns/1ps

module SystemX_TB ();

  reg  [2:0] ABC_TB;
  wire       F_TB;
  reg [2:0]  Vectors_In[7:0];

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

  initial
    begin

    $readmemb("Data_In.txt", Vectors_In);

                                $display("Vec | F");
       ABC_TB=Vectors_In[0];  #1  $display("%b | %b", Vectors_In[0], F_TB);
    #9 ABC_TB=Vectors_In[1];  #1  $display("%b | %b", Vectors_In[1], F_TB);
    #9 ABC_TB=Vectors_In[2];  #1  $display("%b | %b", Vectors_In[2], F_TB);
    #9 ABC_TB=Vectors_In[3];  #1  $display("%b | %b", Vectors_In[3], F_TB);
    #9 ABC_TB=Vectors_In[4];  #1  $display("%b | %b", Vectors_In[4], F_TB);
    #9 ABC_TB=Vectors_In[5];  #1  $display("%b | %b", Vectors_In[5], F_TB);
    #9 ABC_TB=Vectors_In[6];  #1  $display("%b | %b", Vectors_In[6], F_TB);
    #9 ABC_TB=Vectors_In[7];  #1  $display("%b | %b", Vectors_In[7], F_TB);

    end

endmodule
```

An 8x3 array called "Vectors_In" is declared to hold the values read from the external file.

$readmemb() treats each symbol in the input file as a binary value. It populates the entire Vectors_In array when called.

Entries in the Vectors_In array can be assigned to the inputs of the DUT.

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



Example 6.7
Reading test bench stimulus vectors from an external file

Concept Check

CC6.4 What is an advantage of using external files as the input/output in test benches compared to the built-in stimulus generation and reporting functionality within a Verilog module?

(A) External stimulus files allow more complex input stimulus vectors to be used.

(B) External output files allow more sophisticated post-processing of the results.

(C) External files allow much larger datasets to be used and analyzed.

(D) All of the above.

Summary

- A *test bench* is a way to simulate a device under test (DUT) by instantiating it as a subsystem, 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 6.1: Test Bench Overview

6.1.1 What is the purpose of a test bench?

6.1.2 Does a test bench have input and output ports?

6.1.3 Can a test bench be simulated?

6.1.4 Can a test bench be synthesized?

6.1.5 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.1. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., "0000," "0001," "0010," . . .). Your test bench should use a procedural block and individual signal assignments for each pattern. Your test bench should change the input pattern every 10 ns.

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

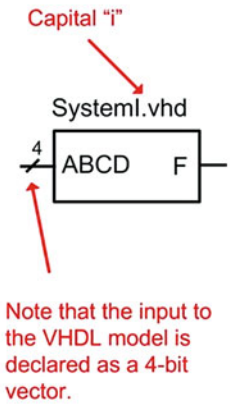


Fig. 6.1
System I Functionality

6.1.6 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.2. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., "0000," "0001," "0010," . . .). Your test bench should use a procedural block and individual signal assignments for each pattern. Your test bench should change the input pattern every 10 ns.

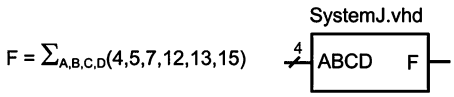


Fig. 6.2
System J Functionality

6.1.7 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.3. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., "0000," "0001," "0010," . . .). Your test bench should use a procedural block and individual signal assignments for each pattern. Your test bench should change the input pattern every 10 ns.

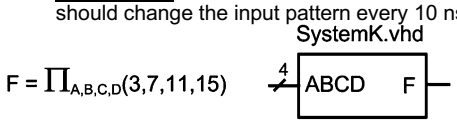


Fig. 6.3
System K Functionality

6.1.8 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.4. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., "0000," "0001," "0010," . . .). Your test bench should use a procedural block and individual signal assignments for each pattern. Your test bench should change the input pattern every 10 ns.

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

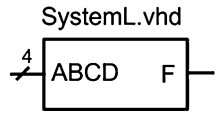


Fig. 6.4
System L Functionality

Section 6.2: Generating Stimulus Vectors Using for Loops

6.2.1 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.1. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., "0000," "0001," "0010," . . .). Your test bench should use a single for loop within a procedural block to generate all of the stimulus patterns automatically. Your test bench should change the input pattern every 10 ns.

6.2.2 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.2. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., "0000," "0001," "0010," . . .). Your test bench should use a single for loop within a procedural block to generate all of the stimulus patterns automatically. Your test bench should change the input pattern every 10 ns.

6.2.3 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.3. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., "0000," "0001," "0010," . . .). Your test bench should use a single for loop within a procedural block to generate all of the stimulus patterns automatically. Your test bench should change the input pattern every 10 ns.

6.2.4 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.4. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., "0000," "0001," "0010," . . .). Your test bench should use a single for loop within a procedural block to generate all of the stimulus patterns automatically. Your test bench should change the input pattern every 10 ns.

6.2.5 Design a Verilog model for an 8-bit Ripple Carry Adder (RCA) using a structural design approach. This involves creating a half adder (half_adder.v), full adder (full_adder.v), and then finally a top-level adder (rca.v) by instantiating eight full adder subsystems. Model the ripple delay by inserting 1 ns of gate delay for the XOR, AND, and OR operators using a delayed signal assignment. The general topology and module definition for the design are shown in Example 4.8. Design a Verilog test bench to exhaustively verify this design under all input conditions. Your test bench should use two nested for loops within a procedural block to generate all of the stimulus patterns automatically. Your test bench should change the input pattern every 30 ns in order to give sufficient time for the signals to ripple through the adder.

Section 6.3: Automated Result Checking

6.3.1 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.1. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., “0000,” “0001,” “0010,” . . .). Your test bench should change the input pattern every 10 ns. Your test bench should include automatic result checking for each input pattern and then print either “PASS” or “FAIL” depending on the output of the DUT.

6.3.2 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.2. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., “0000,” “0001,” “0010,” . . .). Your test bench should change the input pattern every 10 ns. Your test bench should include automatic result checking for each input pattern and then print either “PASS” or “FAIL” depending on the output of the DUT.

6.3.3 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.3. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., “0000,” “0001,” “0010,” . . .). Your test bench should change the input pattern every 10 ns. Your test bench should include automatic result checking for each input pattern and then print either “PASS” or “FAIL” depending on the output of the DUT.

6.3.4 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.4. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., “0000,” “0001,” “0010,” . . .). Your test bench should change the input pattern every 10 ns. Your test bench should include automatic result checking for each input pattern and then print either “PASS” or “FAIL” depending on the output of the DUT.

Section 6.4: Using External Files in Test Benches

6.4.1 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.1. Your test bench read in the input patterns from an external file called “input.txt.” This file should contain an exhaustive list of input patterns for the vector ABCD in the order they would appear in a truth table (i.e., “0000,” “0001,” “0010,” . . .). Your test bench should read in a new input pattern every 10 ns. Your test bench should write the input pattern and the corresponding output of the DUT to an external file called “output.txt.”

6.4.2 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.2. Your test bench read in the input patterns from an external file called “input.txt.” This file should contain an exhaustive list of input patterns for the vector ABCD in the order they would appear in a truth table (i.e., “0000,” “0001,” “0010,” . . .). Your test bench should read in a new input pattern every 10 ns. Your test bench should write the input pattern and the corresponding output of the DUT to an external file called “output.txt.”

6.4.3 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.3. Your test bench read in the input patterns from an external file called “input.txt.” This file should contain an exhaustive list of input patterns for the vector ABCD in the order they would appear in a truth table (i.e., “0000,” “0001,” “0010,” . . .). Your test bench should read in a new input pattern every 10 ns. Your test bench should write the input pattern and the corresponding output of the DUT to an external file called “output.txt.”

6.4.4 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.4. Your test bench read in the input patterns from an external file called “input.txt.” This file should contain an exhaustive list of input patterns for the vector ABCD in the order they would appear in a truth table (i.e., “0000,” “0001,” “0010,” . . .). Your test bench should read in a new input pattern every 10 ns. Your test bench should write the input pattern and the corresponding output of the DUT to an external file called “output.txt.”