1 VCS Simulation Basics

Learning Objectives

After completing this lab, you should be able to:

- Generate a VCS simulation executable by compiling an existing Verilog design using VCS
- Simulate the operation of the Verilog design by executing the simulation binary executable generated by VCS
- Determine whether or not the Verilog design passes verification by reading the console messages generated by the Verilog system task calls in the Verilog source code

15 minutes

UNIT Unit 1

Getting Started

You will be using the following carry-select 8-bit adder for this lab:

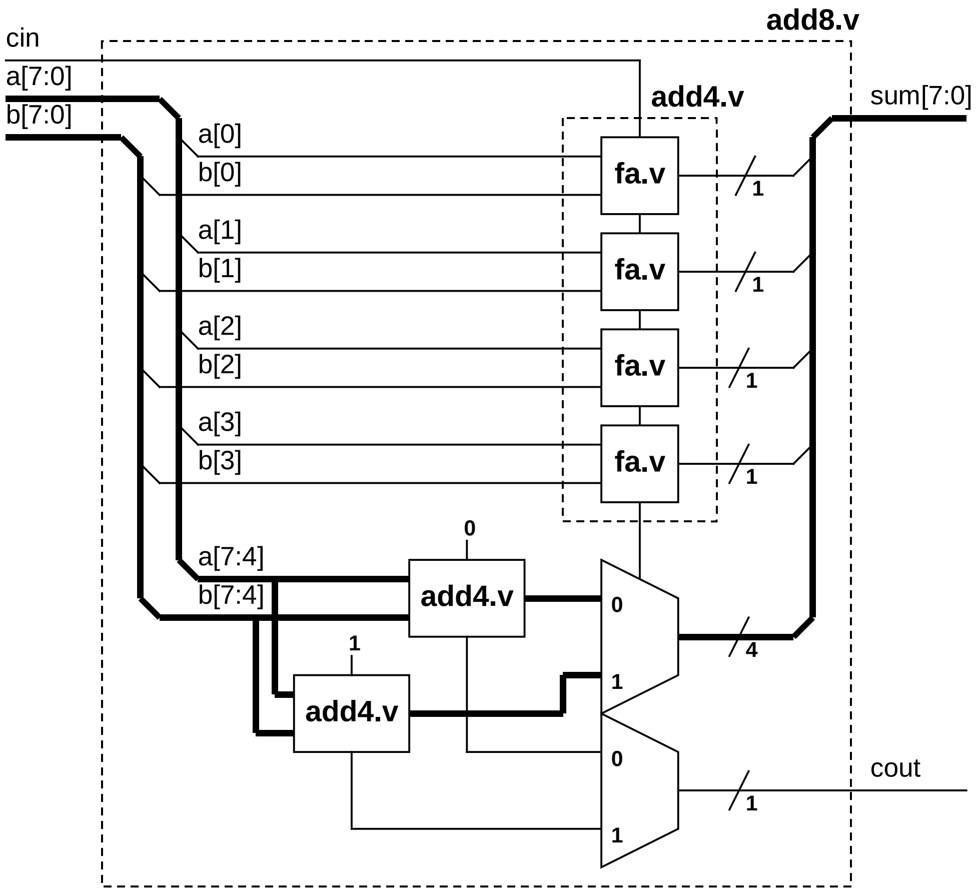
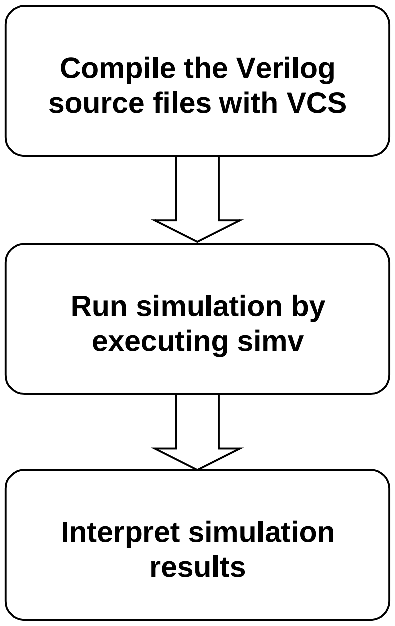


Figure 1-1: 8-bit Carry Select Adder Block Diagram

Our goal is to use this simple design to take you through the fundamentals of the two-step VCS simulation process. This lab is divided into three parts. Each part has its own associated tasks. Here's a preview of what you will be doing:

- Compile the adder Verilog source files to generate a simulation executable.
- Simulate the 8-bit adder by executing the simulation executable.
- Interpret the simulation results displayed on console to determine whether or not the 8-bit adder is working correctly.
- In Part A, all the Verilog source files for the 8-bit adder reside in the working directory.
- In Part B, some of the Verilog source files for the 8-bit adder are in the working directory, and the rest are in a library directory. You will compile them and then use a compile-time file to simplify the VCS compile command line typing.

Figure 1-2: Flow Diagram of Lab Exercise


Compile the Verilog source files with VCS
Run simulation by executing simv
Interpret simulation results

Part A: The two-step Simulation Process

Task 1 Compile to Generate Simulation Executable

In Part A of the lab1, all the Verilog source files for the 8-bit carry select adder reside in your lab working directory.

After logging on to the workstation, go into the lab1 Part A directory.

1. `shell>` **`cd vcs/lab1/parta`**

   You should see four files: `fa.v`, `add4.v`, `add8.v`, and `addertb.v`.

2. `shell>` **`ls`**

   `fa.v`, `add4.v` and `add8.v` are the Verilog source files for the blocks shown in Figure 1-1. `addertb.v` is the testbench used to check the funtionality of the adder.

   Compile the Verilog files and generate the `simv` simulation binary executatble.

3. `shell>` **`vcs addertb.v fa.v add4.v add8.v`**

   When the compilation is done, you should see the message

```
Simv generation successfully completed
```

Task 2 Run Simulation

Run the testbench and simulate the design by executing `simv`.

1. `shell>` **`simv`**

   When the simulation is done, you should see the message

```
$finish at simulation time          13107200
      V C S   S i m u l a t i o n   R e p o r t
Time: 13107200
CPU Time:  0.490 seconds; Data structure size: 0.0 Mb
Mon May 22 09:58:26 2000
```

(Note: your date & time will be different)

Indicating that the simulation has completed. The CPU time used and the memory used during the simulation are also reported.

Task 3 Check Simulation Results

You should also see the following printout generated by Verilog system task calls embedded in the testbench.

```
…
*** Testbench Successfully completed! ***
…
```

This verification run was successful! In Lab 2 we will see how to generate messages to help us debug code errors.

Task 4 Create Simulation Executable with Different Name

The VCS default simulation executable file name is `simv`. You can direct VCS to generate a different executable name by using the `–o` switch.

Recompile the adder design, this time, generate a simulation executable called `addertest`.

Compile the Verilog files and generate the `simv` simulation binary executatble. (please note that the switch is the letter "o" not the number "0")

1. `shell>` **`vcs addertb.v fa.v add4.v add8.v –o addertest`**

   Check the content of the **parta** directory.

2. `shell>` **`ls`**

   You should see 6 files including the simulation binary executable `addertest`.

   Execute this simulation binary to make sure that the simulation results remain the same.

3. `shell>` **`addertest`**

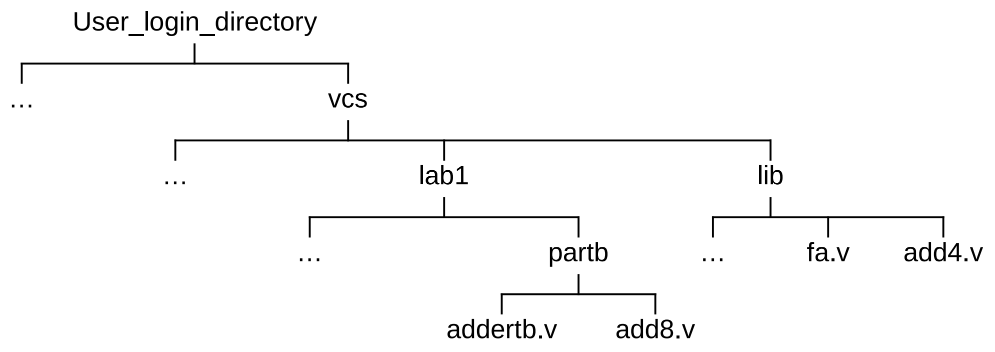
   You should once again, see the following print out generated by the testbench.

```
…
*** Testbench Successfully completed! ***
$finish at simulation time           13107200
      V C S   S i m u l a t i o n   R e p o r t
Time: 13107200
CPU Time:  0.490 seconds; Data structure size: 0.0 Mb
Mon May 22 10:08:21 2000
…
```

Part B: Working with Library Directories

Task 1 Compile & Simulate using Design Library Directory

In Part B of the lab1, we have moved `fa.v` and `add4.v` into a library directory. The new file directory structure now looks like the following:



The `fa.v` and `add4.v` modules are now modules within the library directory `lib`.

Go to lab1 Part B working directory.

1. `shell>` **`cd ../partb`**

2. `shell>` **`ls`**

   You now should only see two files: `add8.v`, and `addertb.v`.

   Compile the design again. Only, this time, we need to reference the library directory file.

   You will also use the `–R` switch as a shorthand to execute simulation immediately after compilation.

3. `shell>` **`vcs addertb.v add8.v –y ../../lib +libext+.v -R`**

   Notice that because our library files have the `.v` extension, the `+libext` switch is required to get vcs to search the `.v` extension files.

   The source file contents have not changed, only the physical placement of the file has changed, you should see identical simulation results as in part A.

Task 2 Compiling with –f File Switch

Simplify the command line entry by using the `–f` compile-time switch. First create a file which contains the names of all the source files or libraries for the design. When compiling the design, reference this file with the `–f` switch.

1. Use any editor you are comfortable with and create the file "`adder.f`" containing the following:

```
addertb.v
add8.v
-y ../../lib +libext+.v
```

Compile and simulate the design by using the `–f` switch as follows:

2. shell> **vcs –f adder.f -R**

The source file contents have not changed, only the physical placement of the file has changed, you should see identical simulation results as in part A and part B.

You are done! Compilation and simluation using VCS is very simple.

Try out some of what you've learned by answering the following questions.

Can you embed the `–R` switch in the adder.f file? ....

Can you use the `–v` switch instead of the `–y` switch? How? ....

2 VCS Debugging Basics

Learning Objectives

After completing this lab, you should be able to:

- Debug an existing Verilog design using Verilog system task calls.
- Debug an existing Verilog design using VCS UCLI features

	45 minutes
UNIT	Unit 2

Getting Started

You will once again be using the following carry-select 8-bit adder:

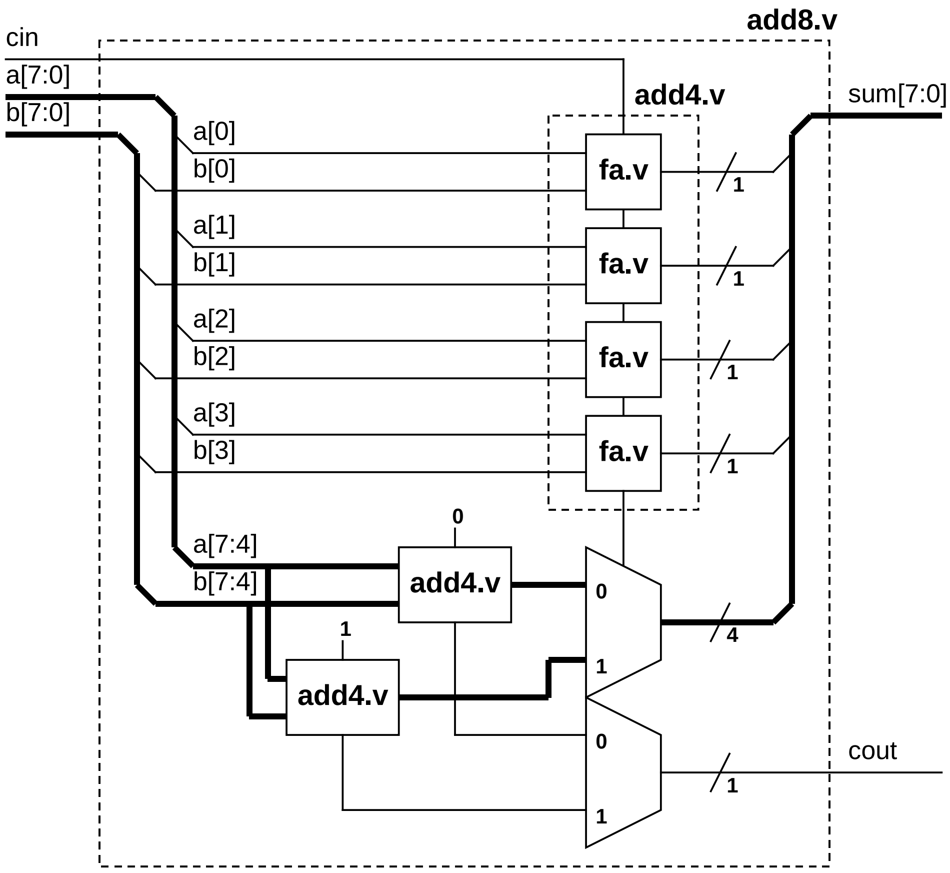


Figure 2-1: 8-bit Carry Select Adder Block Diagram

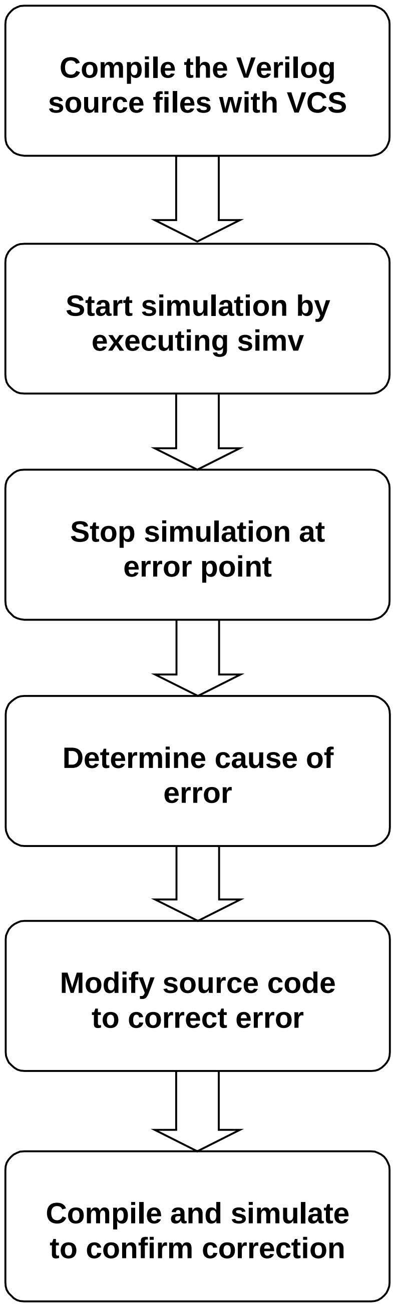
We have embedded errors in the lab files. The goal is to use the debugging techniques presented during lecture to locate and fix the errors. The block diagram that you will see is what the Verilog code intended to implement.

This lab is divided into two parts. Each part has its own associated tasks.

Here's a preview:

- Compile & simulate the adder to note what the errors are.
- **In Part A,** you will insert Verilog system task calls into the design files then compile, simulate and interpret the Verilog system task call outputs to try to pin point the error. You will repeat this process until the error is located and corrected.
- **In Part B,** you will compile with UCLI debugger enabled. You will use the UCLI interactive simulation control to step through the execution of the code and locate the source of the error. You will correct the error and simulate to confirm that the correction works.

Figure 2-2: Flow Diagram of Lab Exercise


Compile the Verilog source files with VCS
Start simulation by executing simv
Stop simulation at error point
Determine cause of error
Modify source code to correct error
Compile and simulate to confirm correction

Part A: Debugging with Verilog System Task Calls

Task 1 A useful VCS Switch

Go into the lab2 Part A directory.

1. `shell>` **`cd ../../lab2/parta`**

Starting this lab, you will use an assortment of compile and run-time switches. In case, you forgot what these switches are, here's a quick way to get a reminder:

2. `shell>` **`vcs -h`**

This command will give you a list of commonly used VCS compile-time and run-time switches, along with a brief explanation of their function.

Task 2 Compile and Run First-Pass Verification

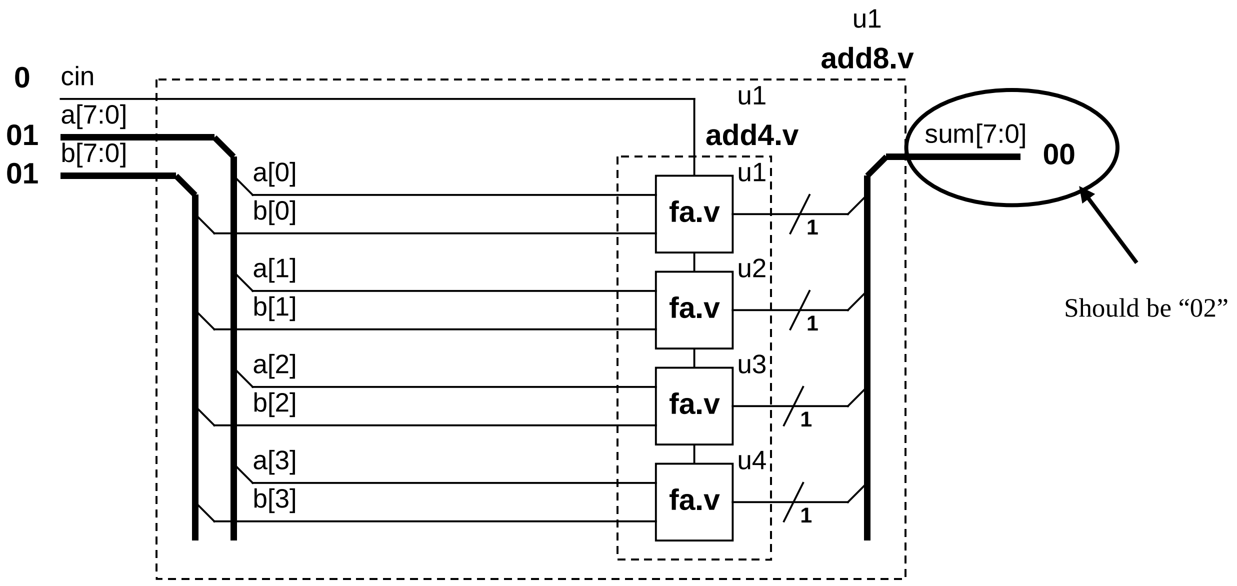
Compile and simulate the carry select adder.

1. `shell>` **`vcs –f adder.f -R`**

An error is reported:

```
***ERROR at time = 25750 ***
a = 01, b = 01, sum = 00;  cin = 0, cout =0
```

With inputs a = 1, b = 1 and cin = 0, the sum is 0 rather than 2. The error could be in any one of the rtl modules. We will examine the operation of each.



Task 3 Debugging the Error

In debugging, you must be able to trace the errors through the design. If you chose to debug using only the Verilog system task calls, there are generally two ways of tracing the errors. You can insert the Verilog system task calls in the rtl modules directly, or you can add the Verilog system task calls in the testbench.

It is better to add debugging statements in the testbench. There are two reasons for this: One, the testbench is typically where you are doing results checking. Using Verilog system task calls along with the result checking routines can effectively give you a breakpoint capability. Two, for performance reasons, you want to avoid re-compilation. By placing debugging statements in only the testbench, you can restrict the re-compilation to just the testbench file.

You will modify the testbench to follow the error through the design hierarchy. First, you will look at the testbench to see where you should place the Verilog system task call. Enter the following UNIX command:

1. `shell>` **`more addertb.v`**

Design Under Test (DUT) instantiation

Useful Verilog system task call for monitoring all activities of an operation

Input stimulus (We will see a better method of managing input stimulus in lab3)

Result checking. This is where insertion of $display system task calls will help the debugging process the most

The $finish is used to terminate the simulation after an error is detected. If you change this to $stop you created a breakpoint at the error state.

You should always have a message to tell you that the simulation has completed. If you don't see this message, your simulation may be caught in an infinite loop

```
module addertb;
reg [7:0] a_test, b_test;
wire [7:0] sum_test;
reg cin_test;
wire cout_test;
reg [17:0] test;

add8 u1(a_test, b_test, cin_test, sum_test, cout_test);

initial
if (!$test$plusargs(“monitoroff”))
  $monitor ($time, "  %h + %h = %h;  cin = %h, cout = %h",
            a_test, b_test, sum_test, cin_test, cout_test);

initial
begin
  for (test = 0; test <= 18'h1ffff; test = test +1) begin
    cin_test = test[16];
    a_test = test[15:8];
    b_test = test[7:0];
    #100;
    if ({cout_test, sum_test} !== (a_test + b_test + cin_test)) begin
      $display("***ERROR at time = %0d ***", $time);
      $display("a = %h, b = %h, sum = %h;  cin = %h, cout = %h",
               a_test, b_test, sum_test, cin_test, cout_test);
      $finish;
    end
  end
  $display("*** Testbench Successfully completed! ***");
  $finish;
end
endmodule
```

Usage of $time to display the simulation time can be very helpful in debugging

The best insertion point is right after the detection of the error. Insert more $display system task calls here to follow the error through the design hierarchy. First take a look at what is happening in the add8 module.

2. Add the following three lines into `addertb.v`

```
$display(“\nIn add8(u1)”);
$display(“a = %b, b = %b, sum = %b;  cin = %b, cout = %b”,
          u1.a, u1.b, u1.sum, u1.cin, u1.cout);
```

```
…
    if ({cout_test, sum_test} !== (a_test + b_test + cin_test)) begin
      $display("***ERROR at time = %0d ***", $time);
      $display("a = %b, b = %b, sum = %b;  cin = %b, cout = %b",
               a_test, b_test, sum_test, cin_test, cout_test);

      $finish;
    end
…
```

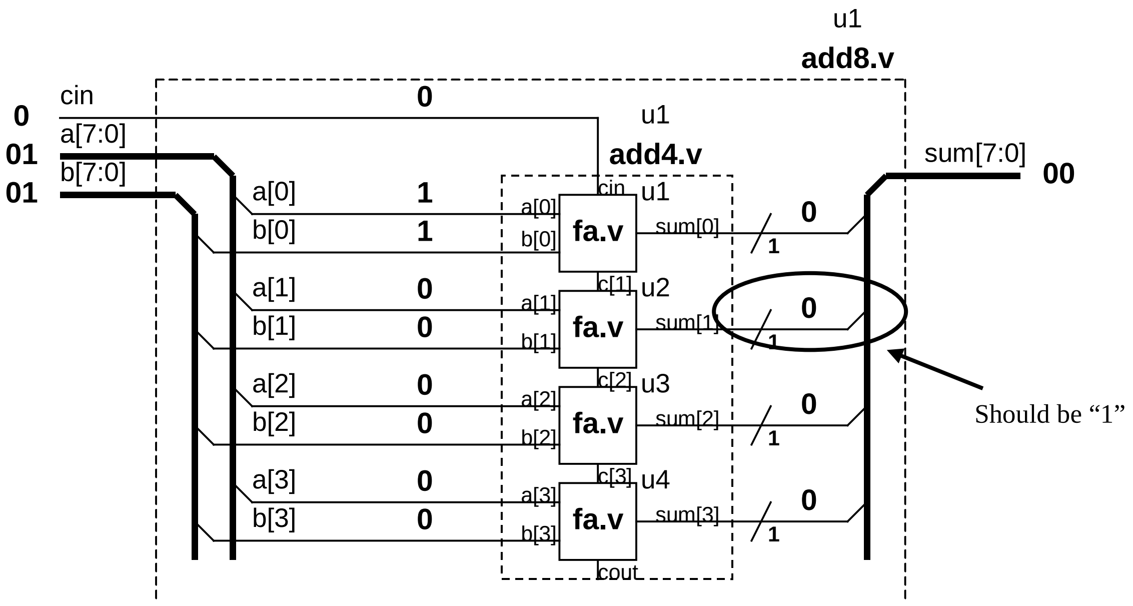
Compile and simulate the adder again to see the results in the add8 module level.

3. `shell>` **`vcs –f adder.f -R`**

The result from this insertion of $display is as follows:

```
In add8(u1)
a = 00000001, b = 00000001, sum = 00000000, cin = 0, cout = 0
```

Bit[1] of the output of add4(u1) should have been a 1. Instead, it is erroneously calculated as 0. Go into add4(u1) to see what’s happening.



4. Insert the following in `addertb.v`.

```
$display(“\nIn add4(u1)”);
$display(“a = %b, b = %b, sum = %b;  cin = %b, c = %b, cout = %b”,
      u1.u1.a, u1.u1.b, u1.u1.sum, u1.u1.cin, u1.u1.c, u1.u1.cout);
```

```
…
     $display("\nIn add8(u1)");
     $display("a = %b, b = %b, sum = %b;  cin = %b, cout = %b",
             u1.a, u1.b, u1.sum, u1.cin, u1.cout);

     $finish;
   end
…
```

Create a breakpoint here by changing the `$finish` system task call to `$stop`.

5. Change this `$finish` to `$stop`.

   With this modification, every time an error is detected, the results listed in the `$display` will be printed and the testbench will be halted at this time step.

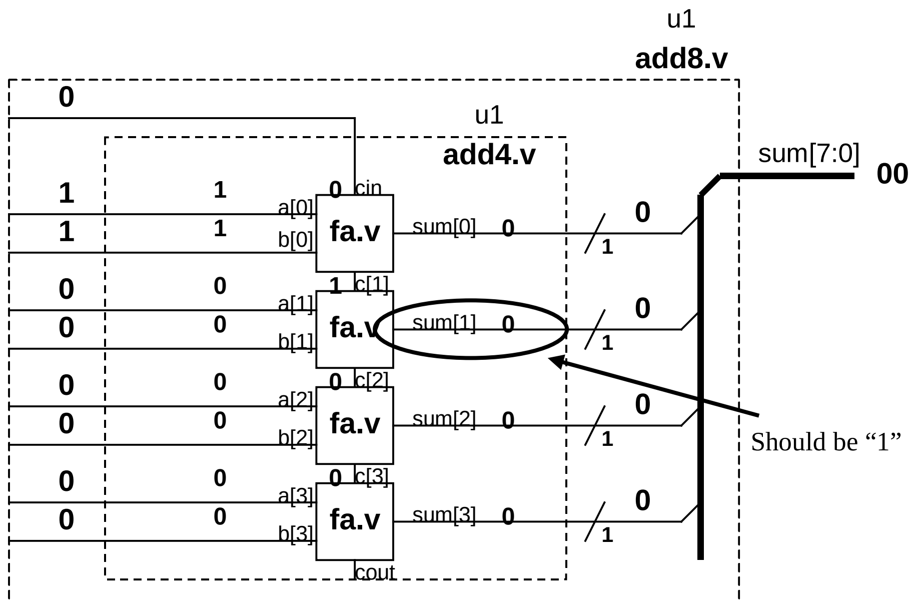
   Compile and run the testbench again.

6. shell> **vcs –f adder.f -R**

   When VCS encounters the `$stop` system task call we embedded, it puts you into the CLI debugger. You will see the CLI prompt `C1>`.

   Take a look at the $display message first:

```
In add4(u1)
a = 0001, b = 0001, sum = 0000, cin = 0, c = 001, cout =0
```



The lsb operation looks okay, however, the second lsb has a problem. It looks like the carry-in is not being added.

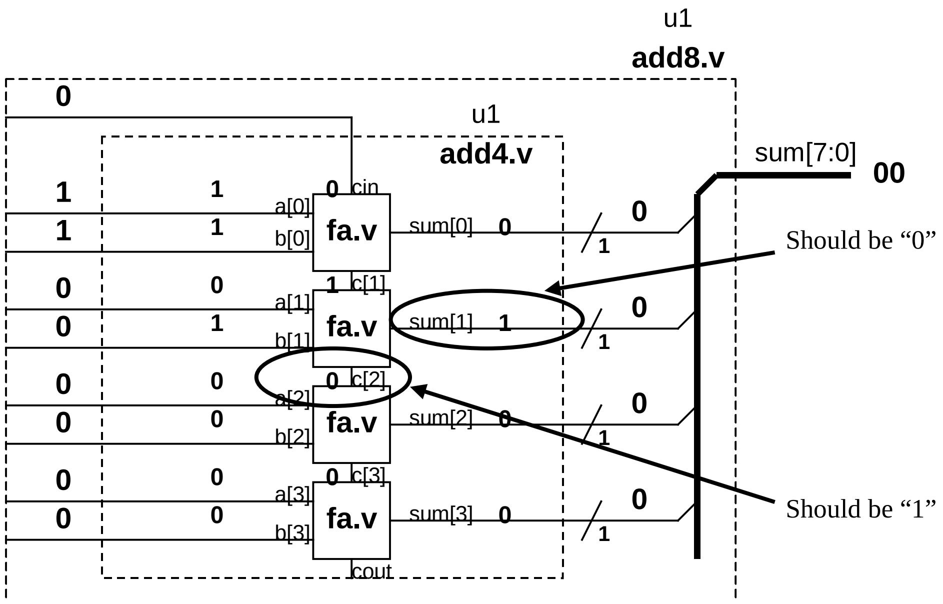
Continue the simulation to the next error point and see if you can deduce this really is a problem. Please continue the simulation with ".". Before continuing, try some other CLI commands to see if they work.

Type the following at the CLI prompt:

7. C1> **.**

The simulation should continue to the next error point and stop. You should see:

```
In add4(u1)
a = 0001, b = 0011, sum = 0010, cin = 0, c = 001, cout =0
```



Try the "." MINI CLI command two more times and record the results. You will begin to notice that the problem appears to be in how the fa module handles the carry-in.

Exit the MINI CLI command mode.

8. C1> **quit**

Display the content of the fa.v file on the console.

9. shell> **more fa.v**

```
module fa(a, b, cin, cout);
input a, b, cin;
output sum, cout;
```

```
assign {cout, sum} = a + b;
endmodule
```

Do you see the problem? Edit `fa.v` to fix the problem then re-compile and simulate to verify that your correction works.

Did the simulation take a while to complete? Why?

Re-simulate with the “monitoroff” plus argument. This will tell VCS to skip the $monitor lines in the testbench.

```
…
add8 u1(a_test, b_test, cin_test, sum_test, cout_test);

initial
if (!$test$plusargs(“monitoroff”))
  $monitor ($time, “ %h + %h = %h;  cin = %h, cout = %h”,
           a_test, b_test, sum_test, cin_test, cout_test);

initial
…
```

10. `shell>` **`simv +monitoroff`**

Is there a difference in the simulation speed?

Excessive use of message passing will slow down simulation. When you need faster simulation speed, the first thing to do is to remove the unnecessary message passing statements.

Part B: Debugging with VCS UCLI Debugger

Task 1 Compile and Run First-Pass Verification

One thing you might have noticed in part A, is that debugging by using Verilog system task call insertions is a lot of work. You are constantly executing the complete modify-compile-simulate-verify loop. Not only is the data entry time consuming, every other step in this debugging loop is also time consuming. In part B, we will use the UCLI command set to simplify the debugging process.

In part A, you needed to use pen & paper to help you to diagnose the problem. UCLI simplifies, but does not eliminate the need for pen & paper.

Go to the lab2 Part B directory.

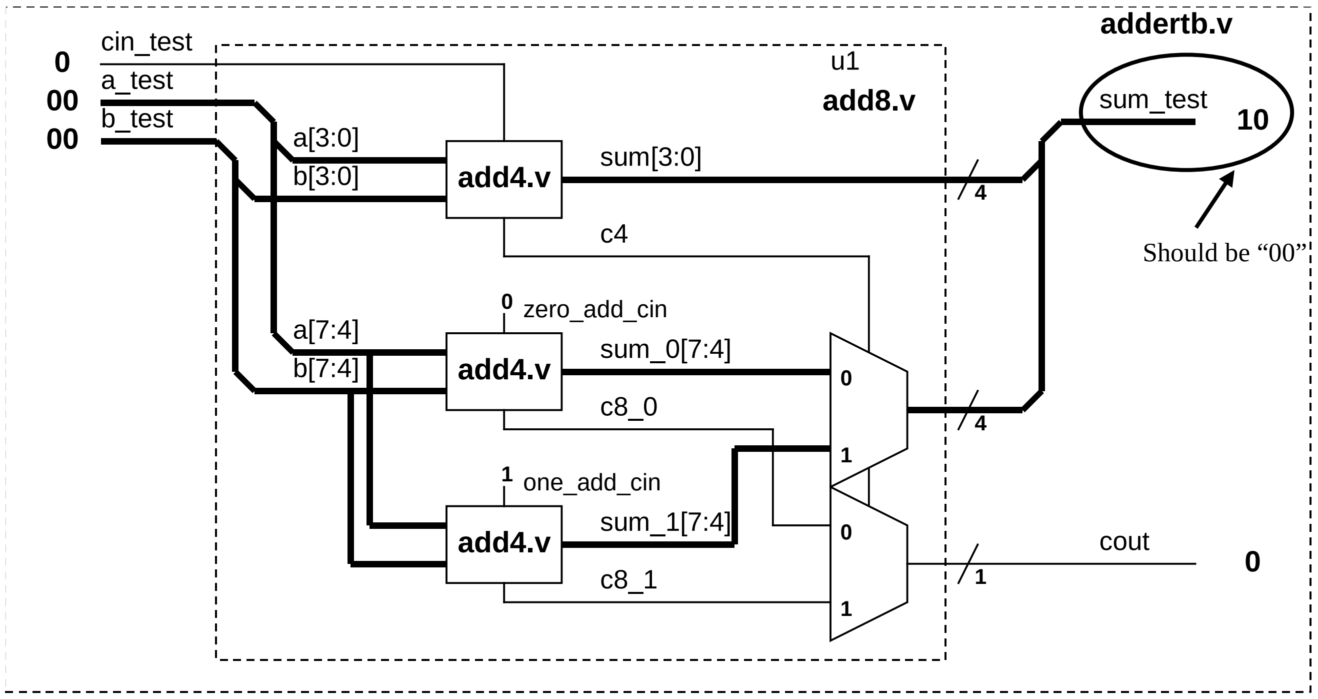
1. shell> **cd ../partb**

Once again, compile and simulate the adder to see if it's working properly.

2. shell> **vcs –f adder.f –R**

Notice there is an error at time 50.

```
***ERROR at time = 50 ***
a = 00, b = 00, sum = 10;  cin = 0, cout =0
$finish at simulation time                    50
```



Task 2 Compile with UCLI Debugger Switch Enabled

Re-compile, this time, enable the UCLI debugger.

1. First, add a hard breakpoint in `addertb.v` by modifying the file as follows:

Change to `$stop`

```
…
    if ({cout_test, sum_test} != {a_test + b_test + cin_test)) begin
      $display(“***ERROR at time = %0d ***”, $time);
      $display("a = %h, b = %h, sum = %h;  cin = %b, cout = %b",
                 a_test, b_test, sum_test, cin_test, cout_test);
      $finish;
…
```

Compile & simulate with UCLI enabled.

2. `shell>` **`vcs –f adder.f –R –debug_all -ucli`**

When VCS encounters the `$stop` system task call we embedded, it puts you into the UCLI debugger. You will see the UCLI prompt `ucli%`.

Task 3 Debugging with UCLI Debugger

When the UCLI debugger is invoked, you are placed at simulation time 0.

```
$stop at time 0
ucli%
```

If you forget what UCLI commands are available, try the following to get a quick summary of the UCLI commands:

1. `ucli%` **`help`**

To find out where we are in the design hierarchy enter:

2. `ucli%` **`scope`**

The scope is testbench module `addertb`.

To get to the problem area enter:

3. `ucli%` **`run`**

```
***ERROR at time = 50 ***
a = 00, b = 00, sum = 10;  cin = 0, cout =0
$stop at time 50 Scope: addertb File: addertb.v Line: 25
```

Take a look at what signals are defined in `addertb`.

4. `ucli%` **`show`**

```
sum_test
cout_test
a_test
b_test
cin_test
test
u1
```

List all the signals/ports/scopes, and so on. To see value of signal that are of interest, use the `get` command.

5. `ucli%` **`get sum_test -radix hex`**

```
'h10
```

The sum_test bus is incorrect. Trace the sum_test signal to locate the problem. Move into the `u1(add8)` module using the `scope` command.

6. `ucli%` **`scope u1`**

```
addertb.u1
```

7. `ucli%` **`show -ports -value -radix hex`**

```
a 'h00
b 'h00
cin 'h00
sum 'h10
cout 'h00
```

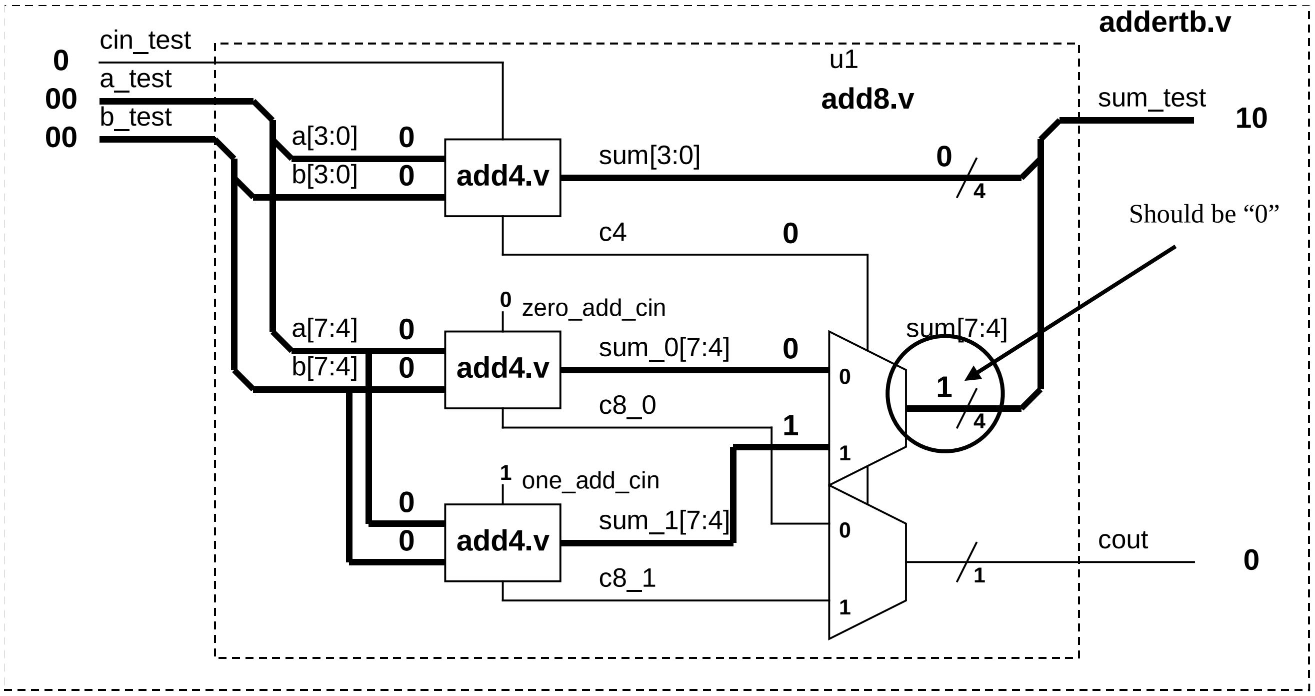
In debugging, you often need to print out the values of the same set of variables multiple times. Creating an alias for this commonly executed task will make things much easier. Once we have created the alias, executing it is as simple as just typing in the alias name.

8. `ucli%` **`alias vars show -value -radix hex a b sum sum_0 sum_1 cin c4`**

9. `ucli%` **`vars`**

```
a 'h00
b 'h00
sum 'h10
sum_0 'h0
sum_1 'h1
cin 'h00
c4 'h0
```

You can see the details of the error here. The 2:1 mux should select the sum_0 value of “0” but it selected the sum_1 value of “1”. This may be the problem.

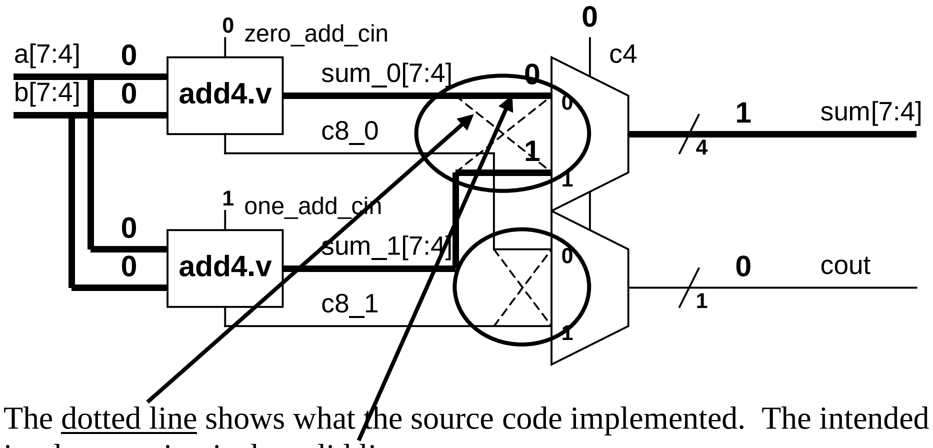


Leave the UCLI window as is (**DO NOT exit UCLI**). Open a new UNIX window and take a look at the add8.v file.

10. `shell>` **`more add8.v`**

Take a look at the mux code. The inputs to the mux are reversed. In fact, both the sum and the carry mux have the same error.

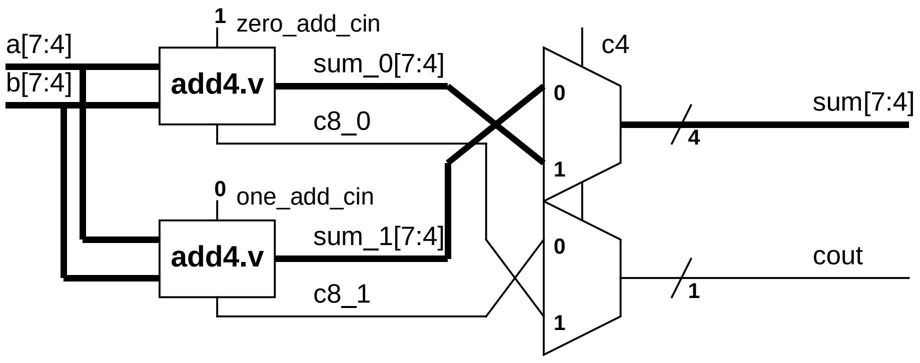
```
Assign sum[7:4] = c4?sum_0:sum_1;
Assign cout = c4?c8_0:c8_1;
```



The dotted line shows what the source code implemented. The intended implementation is the solid line.

DO NOT fix the error yet. Go back into the UCLI debugger window.

Fortunately, you can emulate a correction without leaving the UCLI environment. If you change carry-in of the sum_0 adder to “1” and change carry-in of the sum_1 adder to “0”, you will have effectively corrected the problem.



Notice what the current value of the zero_add_cin and one_add_cin signals are.

11. `ucli%` **`show zero_add_cin one_add_cin -value`**

```
zero_add_cin 0
one_add_cin 1
```

Force the zero_add_cin signal to a “1” and the one_add_cin signal to a “0”.

12. `ucli%` **`force zero_add_cin 1`**

13. `ucli%` **`force one_add_cin 0`**

To take a look at all the values, use the “`prvars`” alias that you created. You have not, however, included `zero_add_cin` and `one_add_cin` when you created the alias. Because they are needed to view all values, you will need to add them to the command line.

14. `ucli%` **`vars zero_add_cin one_add_cin`**

```
a 'h00
b 'h00
sum 'h10
sum_0 'h0
sum_1 'h1
cin 'h00
c4 'h0
zero_add_cin 'h1
one_add_cin 'h0
```

The carry-in values have changed, `sum[7:4]` remains the same. This is because we have not advanced the simulation time to allow for the execution of the new values. You are going to advance the simulation time by 10 time increments.

15. `ucli%` **`run 10s`**

```
60 s
```

Take a look at the values now.

16. ucli% **vars**

    Now, all values should be correct. Return to the testbench level and verify that the adder is working properly.

17. ucli% **scope -up**

18. ucli% **show a_test b_test cin_test cout_test sum_test –value**

```
a_test 0
b_test 0
cin_test 0
cout_test 0
sum_test 0
```

These are the correct results. Set a breakpoint where c4 in module add8 changes from ‘0’ to ‘1’ and see if this works.

19. ucli% **stop -posedge u1.c4**

```
1
```

20. ucli% **run**

```
Stop point #1 @ 26400 s;
```

You have advanced the simulation from 60 to 26400 without encountering an error. Take a look at the values.

21. ucli% **show a_test b_test cin_test cout_test sum_test –value**

```
a_test 1
b_test 8
cin_test 0
cout_test 0
sum_test 8
```

Is this right?

VCS encountered the transition of c4 from ‘0’ to ‘1’ and stopped the simulation at that point. There may, however, still be other events in addition to these that have not yet been encountered. In this specific case, the 4-bit MSB (`sum_1` in `addertb.u1`) addition takes place after the transition of the u1.c4 signal. The current value for sum_test, therefore, has not been updated yet. What you are seeing is the previous value of sum_test.

You need to advance the simulation to at least the next time step to see the full effect of all events in the current time step.

22. `ucli%` **`run 1`**

```
26401 s
```

23. `ucli%` **`show a_test b_test cin_test cout_test sum_test –value`**

```
a_test 1
b_test 8
cin_test 0
cout_test 0
sum_test 9
```

Now this looks right.

Remove the breakpoint and continue the simulation.

24. `ucli%` **`stop`**

```
1: -posedge addertb.u1.c4
```

Breakpoint number

We can remove a breakpoint by deleting its breakpoint number.

25. `ucli%` **`stop -delete 1`**

```
1
```

Resume the simulation.

26. `ucli%` **`run`**

```
*** Testbench successfully completed! ***

addertb.v, 30 :   $stop;
```

You have successfully detected the error and worked around it in the simulation.

If you get tired of re-typing the same commands over and over again, use the log file as a reference to generate a script file. This will simplify your debugging command entries.

We have already generated a script file for you. In this script file, the commonly used aliases are defined and the emulated carry-in fix commands are executed.

Note: Remember this is not a real fix! It is used only to allow simulation to continue without leaving the UCLI environment.

Restart the simulation.

27. `shell>` **`simv –ucli –i test.s`**

You can see that `zero_add_cin` and `one_add_cin` have been set to the emulated values. Try the alias tb.

28. `ucli%` **`tb`**

This alias works! Let's simulate.

29. `ucli%` **`run`**

Congratulations! Exit UCLI and take a look at the simulation log file. The log file should show you everything that you've done during the simulation session.

30. `shell>` **`more ucli.key`**

Does the content look familiar? Use the content of this file as a reference to generate your script files.

You are done with part B. Continue to part C.

Part C: Getting Help

Task 1 Submitting Files for Help

If you ever need to get help from the VCS support team, there are two switches that will make working with the support staff go a lot smoother. They are `–ID` and `–Xman=4`. Let's see what these switches generate.

You do not need to change the directory.

Enter the following at the UNIX prompt:

1. `shell>` **`vcs –ID > id.txt; more id.txt`**

   You should see a summary of the VCS version being used and workstation OS information. This will help the support staff to quickly focus on issues relevant to the specific VCS version and workstation OS.

   When phone support does not resolve your issue, your next plan of action should be to submit a testcase for the Synopsys engineers to play with, use the `–Xman=4` switch for this purpose.

   Enter the following at the UNIX prompt: (to simplify the command typing, the file script is already written for you)

2. `shell>` **`vcs –f adder.f –Xman=4; ls`**

   You should see that the file tokens.v has been generated. Take a look at it.

3. `shell>` **`more tokens.v`**

   Do you see all the modules of the design in this file? ....

Note: PLI files are not included in `tokens.v`. You will need to submit them separately.

3 Debugging with DVE

Learning Objectives

After completing this lab, you should be able to:

- Debug an existing Verilog design using DVE GUI.
-

	30 minutes
UNIT	Unit 3

Getting Started

- You are handed this design, told there's a bug in it, and to fix it. The usual.
- This is a FIFO written in Verilog with a Verilog testbench. The testbench detects bugs with checkers.
- It prints output and terminates simulation upon failures which we will debug with DVE.
- The FIFO uses count to determine how many entries are in the FIFO and to determine empty and full. There is a head and tail pointer that shows where to write and read from, respectively.
- If you ever get lost and need to back up to the beginning, the original .v and .f files are in the ./start_over directory. Copy them to ".", then go back to the COMPILE AND RUN step. (can use cleanup script)

FIFO REQUIREMENTS

The testbench has its own counter, count_checker, to determine how many entries are in the FIFO and to determine empty and full.
The checkers implemented in the testbench cover the following requirements:

1. FIFO count always equals TB count_checker
2. If count_checker is empty, empty flag must be asserted.
3. Never allow underflow
4. Never allow overflow

COMPILE AND RUN

The first thing you decide to do is compile and run this design in VCS.
Since you will debug in DVE, you compile with the "-debug" flag and have $vcdpluson; in your testbench code.

```
% vcs -debug -f run1.f        (can use run_debug script)
% simv
```

You notice the message:

```
FAILURE: Empty flag missed at time 250000000000.000 ps. Exiting test.
...
$finish at simulation time 261000000000.000 ps
```

This is what we will debug.

DEBUG THE EMPTY FLAG BUG

We'll start up DVE and open the vcdplus.vpd simulation result file.

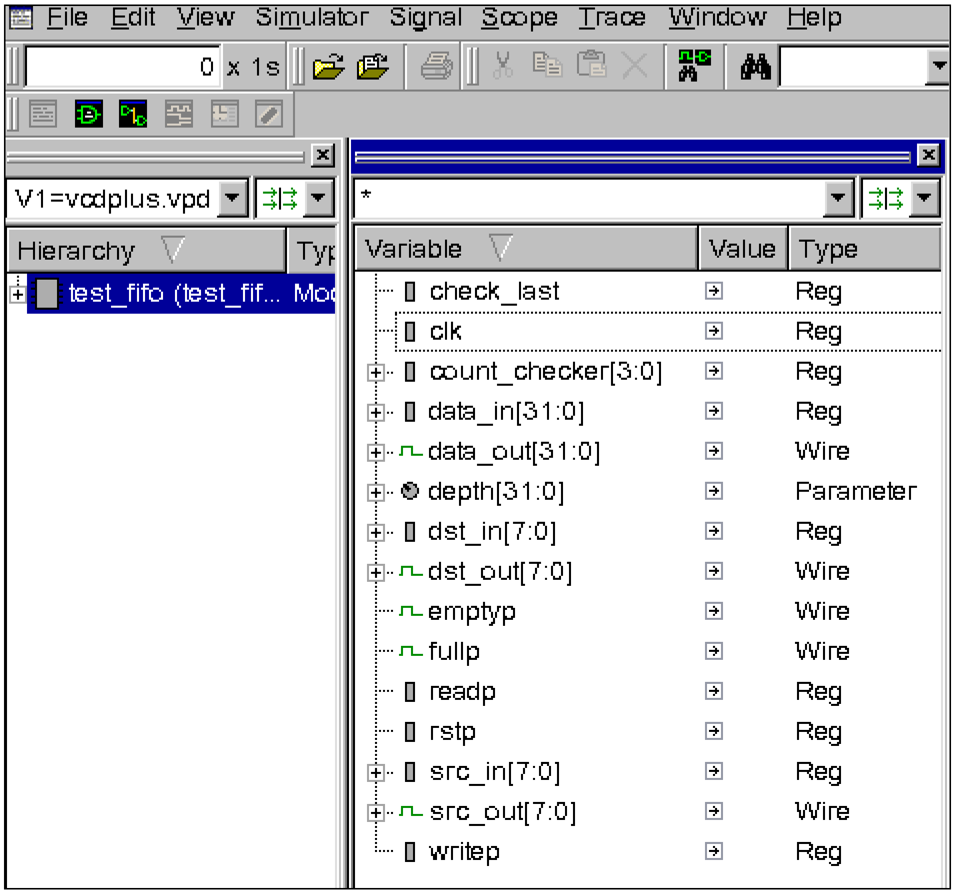
```
% dve &
```

File -> Open Database... In the dialog, select vcdplus.vpd. Click Open.
The hierarchy pane shows test_fifo.

The Design dropdown menu shows the name of the displayed file, V1=vcdplus.vpd
Expand test_fifo, and you can see components of this design.

Display the testbench signals in a Wave window.

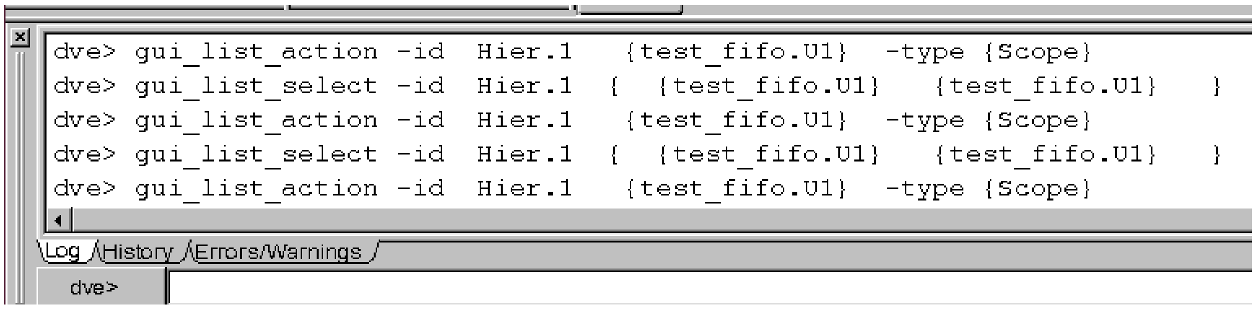
Select New -> Wave Window from the Window menu. A new wave window opens.
In Hierarchy pane, select test_fifo. Notice its signals are displayed in the Variable pane.



Variable	Value	Type
check_last		Reg
clk		Reg
count_checker[3:0]		Reg
data_in[31:0]		Reg
data_out[31:0]		Wire
depth[31:0]		Parameter
dst_in[7:0]		Reg
dst_out[7:0]		Wire
emptyp		Wire
fullp		Wire
readp		Reg
rstp		Reg
src_in[7:0]		Reg
src_out[7:0]		Wire
writep		Reg

Right click on test_fifo to view the Hierarchy context menu.
Select Add to Waves.
All signals in test_fifo are added to the Wave window.

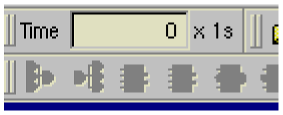
Notice at the bottom of the main DVE window is the Console pane. In the console pane, you see a Log, History and Errors/Warnings tab. Each action that you have done in DVE has been logged to this pane as a Tcl command. You can enter Tcl commands at the command prompt. The content of these tabs are also saved in history logs in the current directory.



```
dve> gui_list_action -id  Hier.1   {test_fifo.U1}  -type {Scope}
dve> gui_list_select -id  Hier.1  {  {test_fifo.U1}   {test_fifo.U1}   }
dve> gui_list_action -id  Hier.1   {test_fifo.U1}  -type {Scope}
dve> gui_list_select -id  Hier.1  {  {test_fifo.U1}   {test_fifo.U1}   }
dve> gui_list_action -id  Hier.1   {test_fifo.U1}  -type {Scope}
```

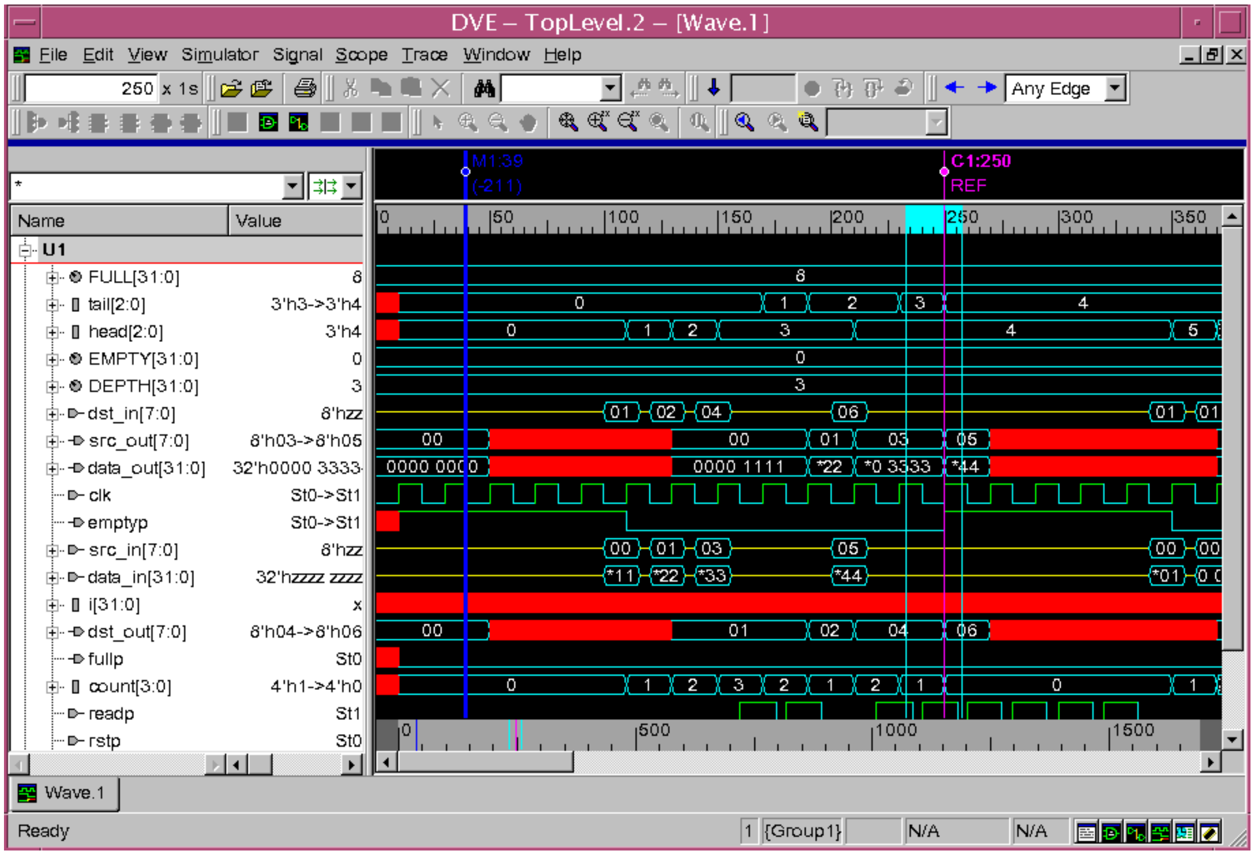
Back to figuring out that empty flag bug...

As you can see in the Time field units at the top of the windows, the timescale is currently 1s, and our simulation was in 100ps. This could be modified via View -> Set Time Scale... But, we'll just leave it.



Zoom around in Wave window.

In the Wave window, we see a timescale bar at the top of the wave window and one at the bottom.
The bottom one always shows all time available in the open database.
The top one shows you the portion that is currently visible in the wave window.



Since our bug was at time 250, let's zoom over there using the bottom timescale.
On the bottom timescale, click down at about time 240 and drag through 250 and let go.

The top timescale and waveform now shows about 240 to 260. This is much quicker than scrolling.

Put a marker at the time of failure.

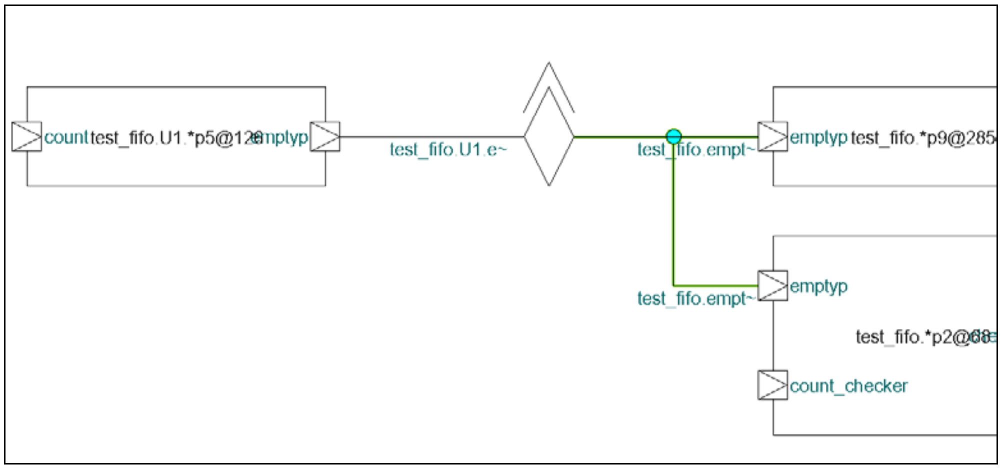
Let's put a marker at our problem time. In waveform area, right click and select "Create Marker" from the context menu.
You get a moving cursor that you can place at time 250 by left clicking a transition at time 250 on a waveform (ex: data_out[31:0]).
It will snap the marker to the nearest transition.
Notice the bottom timescale has a white vertical line through time 250. This denotes the marker we just created.
Notice also that there is a pink line at time 0 in the bottom timescale. Zoom into that pink line using the bottom timescale click and drag technique.
The top timescale now displays from time 0 and you see cursor C1=0 in the marker pane.

C1 is the application time, it is displayed in the Time field at the top of the windows.
You can move C1 by entering a time in the Time field, likewise, you can change the time in the time field by moving C1.
Why is this interesting? This C1 application time is the time attached to the values when you annotate values in source, or schematics.
We'll use these features soon.

Let's move the application time, C1, to 250 where our marker is.
Zoom back over to the white M1 marker using the bottom timescale. In the Waveform pane, left click on a transition at time 250.
Left clicks in the waveform pane snap C1 to the nearest transition.
Now the Time field shows 250. The time our empty flag bug showed up.
We can see that signal emptyp in the wave window is indeed 0 at time 250 and the checker says it should be 1.

Show connections of emptyp in path schematic window.

In main window Variable pane, select emptyp. Click the Show Path Schematic Window button in the toolbar buttons, or select New -> Show Path Schematic from the Window menu. A path schematic window opens with the connections of emptyp in the testbench.



What do we have here?

- The cyan line shows emptyp connections with solder points in the testbench.
- The green components labelled "*p9@285" and "*p2@68" show process 9 at line 285 and process 2 at line 68, respectively. These represent blocks of source code.
- The orange component shows connections that cross a level of hierarchy.

Since emptyp comes out the top, we can tell it represents connection to a child component.
Hold your mouse over the hierarchy transition component to see the tooltip.
It shows that it connects with test_fifo.U1.emtpyp.
We need to see where emptyp comes from, so double click on the orange hierarchy transition.
Now we are looking at 2 levels of hierarchy, the processes in the testbench AND a process in test_fifo_U1.
Zoom in a little using the Zoom In 2x toolbar button.
Click the Pan Tool button and grab and drag the schematic with the pan tool so you can see the process in test_fifo.U1, called *p5@126.
(Or use the window scroll bars)
Change back to the Selection Tool.

Annotate pin values in schematic and expand path.

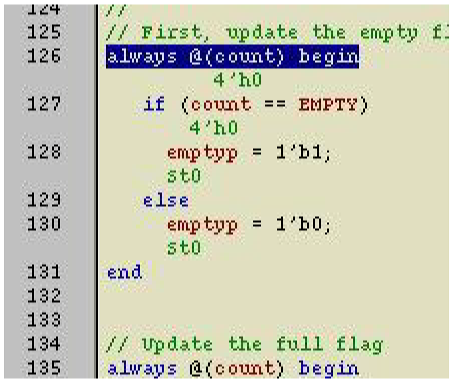
Select Scope -> Annotate Values, now we see the St0 value for emptyp.

To get the value for count, we will need to expand the path schematic. Click to select the count port on the process.
Right click and select Expand Path from the menu.   Zoom in.
Now we see the source code component connected to count, in addition to its value, 4'h0. The FIFO is empty.

Show source code from schematic.

In the path schematic window, select the source code block with count as input and empty as output, that is, the one called *p5@126.
It is now highlighted in white. Now right click and select Show Source from the menu.

DVE source window pops forward highlighting text on line 126.
The source code is an always block. Let's annotate values here (if they are not already present).



```
124  //
125  // First, update the empty fl
126  always @(count) begin
              4'h0
127     if (count == EMPTY)
             4'h0
128        emptyp = 1'b1;
           st0
129     else
130        emptyp = 1'b0;
           st0
131  end
132
133
134  // Update the full flag
135  always @(count) begin
```

Select Source -> Annotate Values. Values are annotated in green, below the signals.
We see count is empty and based on this code, it should have gone through the if, not the else.
The "if" condition, count == EMPTY, must have failed. Let's verify that.
Searching text in source code. Double click the word EMPTY so that it is highlighted.
Now click the Find... toolbar button. (Looks like binoculars)
It opens the Find tool. Select Match case, so that we don't stop on emptyp and click Find Next.
We have a match on line 24. We see that EMPTY = (1<<3), the same as the FULL parameter.

Here's our bug.

It should be EMPTY = 0. Close the Find dialog.

Edit source to fix the bug.

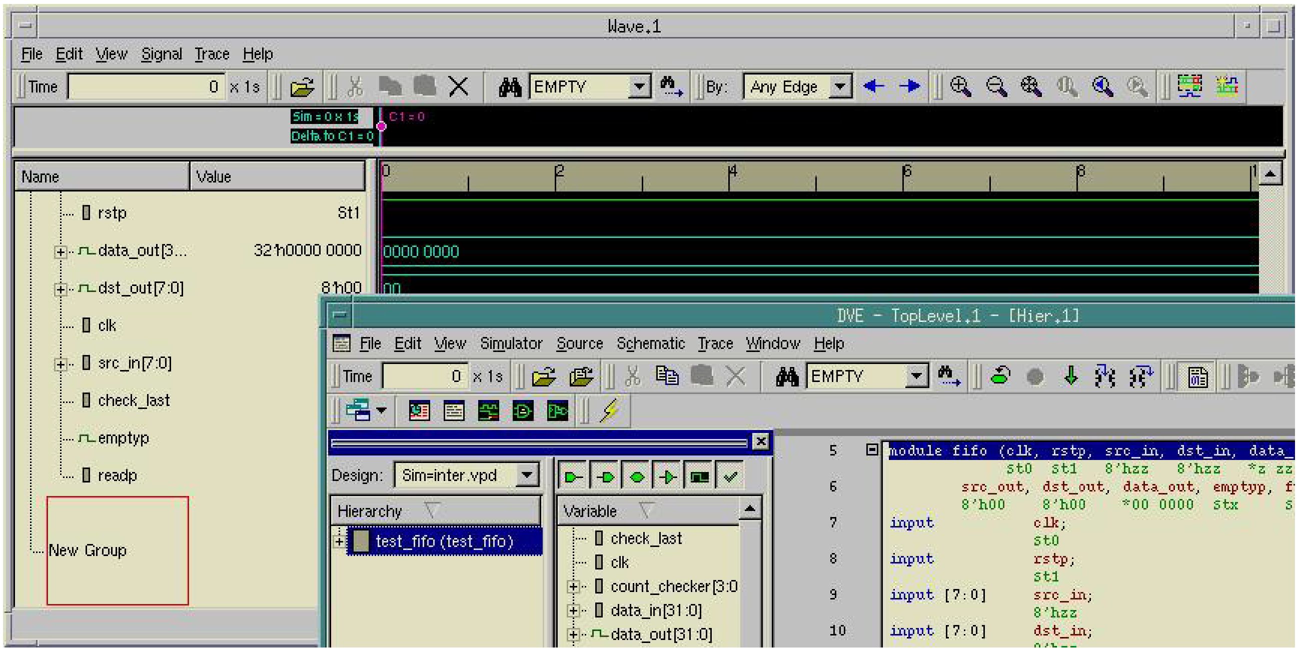
Select Scope -> Edit Source.

DVE chose the editor based on your EDITOR environment variable.
Change to EMPTY = 0; and save it.

Recompile and simulate in DVE to verify bug fix.

Recompile the design outside of the GUI - the same as you did in Compile and Run. (run_debug)
Back in DVE, Select Simulator - Setup. A new window opens, select OK which will run simv in interactive mode.

The current design is Sim=inter.vpd, and V1=vcdplus.vpd is still there in the menu.
Let's drag test_fifo from Hierachy Window to the Wave Window over "New Group".
Click on test_fifo with the middle mouse button and drag it to the wave window.



```
module fifo (clk, rstp, src_in, dst_in, data_
             st0  st1   8'hzz   8'hzz   *z zz
        src_out, dst_out, data_out, emptyp, f
        8'h00    8'h00    *00 0000  Stx     S
input           clk;
                st0
input           rstp;
                st1
input [7:0]     src_in;
                8'hzz
input [7:0]     dst_in;
```

(Now we have one group with the vcdplus.vpd signals in it and another with the interactive signals in it.
Notice that the signals in Group2 are all gray since we have not yet run the simulation)
Our C1 cursor is now hooked up to Sim time and is back to 0.

Let's run it to 260 to see if it fixed that empty flag bug.
Enter 260 in the Time field and hit Enter. Sim runs to time 260, C1 and app time are 260.
Scroll the waveform display to find our interacitve "emptyp" signal in the Group2 signal group.
Verify its value is 1 at the time of our marker, time 250. The bug is fixed.

Close the buggy simulation results.

Let's close the post-processing, vcdplus.vpd database.
File -> Close Database. Select V1 .../vcdplus.vpd. Click OK.
Leave the interactive one open, inter.vpd.

RUN SIMULATION IN DVE

Let's look for more bugs.
To run the simulation forward, click the Continue button in the main window toolbar.
Another FAILURE!
You see the VCS simulation output in the Log tab of the Console pane when you are running simulations in DVE.
This time, is prints and exits simulation:

```
FAILURE: Counter mismatch at time 16800000000000.000 ps! Exiting test.
...
$finish at simulation time 16900000000000.000 ps
```

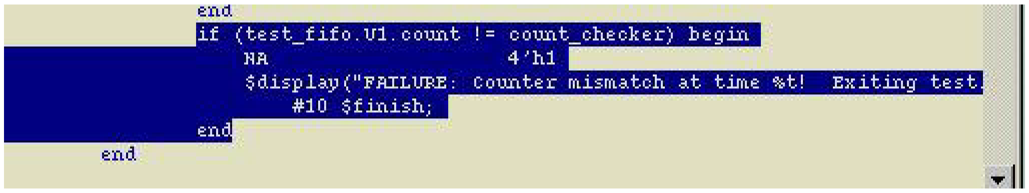
DEBUG THE COUNTER MISMATCH BUG

Find the checker that printed the error message.

In Hierarchy pane, double click the test_fifo testbench to populate the source window.
Click the Find toolbar button at the top of the window.
Enter "Counter mismatch" and click Find Next to take us to the checker.

We see that the comparison that failed was the tb counter not equal the FIFO counter and it was detected at the negedge of clk.

In Wave window, zoom full with Zoom Full toolbar button.. Locate the count_checker signal in the wave window.
Middle mouse click on that signal to place the red insertion bar below it.
Any signals added to the Wave window, will be added at this point.
In Source window, select the text of the checker from the if to the end.



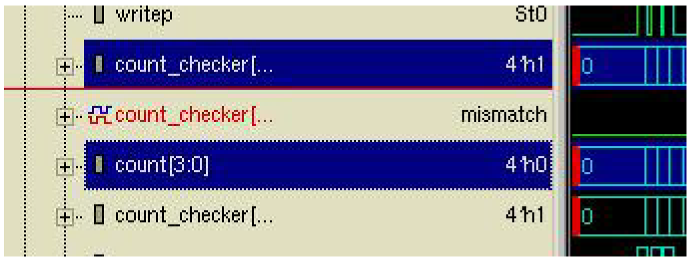
```
                    end
                    if (test_fifo.U1.count != count_checker) begin
                        NA                          4'h1
                        $display("FAILURE: Counter mismatch at time %t!  Exiting test.
                            #10 $finish;
                    end
            end
```

Right click and select Add to Waves from the context menu.

Now we have count and count_checker (our mismatched counters in the Wave Window.)

Compare count signals to find the mismatch.

Select the count and count_checker in the Wave window, so that both are highlighted (use Ctrl or Shift for multiple selection).
Select Signal -> Compare Signals... The compare tool is automatically populated with the signals you selected.
The top shows the database and signals to compare (you can also compare instances, not just signals). The middle shows the options for comparison, and the bottom is a summary.
Just click OK.



writep	St0	
count_checker[...	4'h1	0
count_checker[...	mismatch	
count[3:0]	4'h0	0
count_checker[...	4'h1	0

The comparison result is added in red to the Wave window at the insertion point.

Search forward to get to mismatch.

Select the compare result signal in the wave window.
Move your C1 cursor to time 0 and then click the Search Forward toolbar button in the Wave window. (blue, right arrow)

The Search is controlled by the "By:" dropdown menu, which is Any Edge, by default.
The C1 cursor moves to time 1670, the time of the first mismatch. Zoom in around that C1 cursor using the top timescale.
We see that the FIFO count is 0 and the testbench counter is 1 at this time.

Get drivers of the count signal.

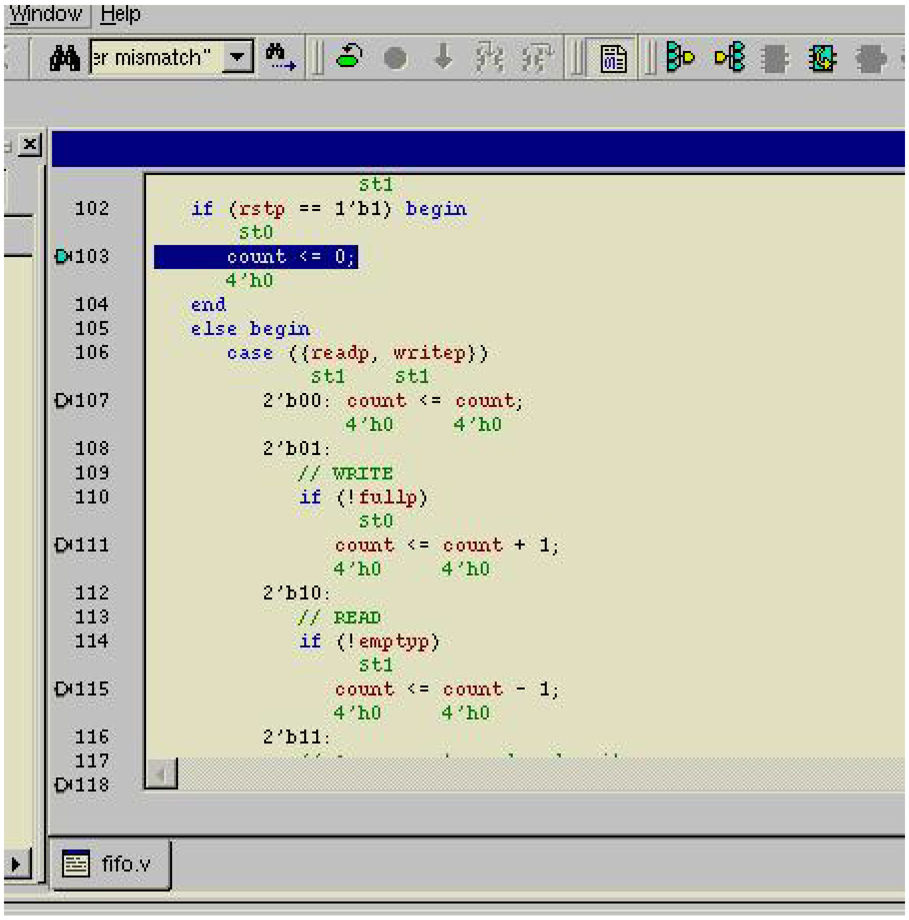
Double click the waveform transition of the count signal at time 1670, when it transitioned to 0. This opens the Drivers and Loads window. It shows that there are four drivers in one source file.

The Source window pops forward with driver icons on the lines that drive count.

Annotate values and step through drivers to find the active one.

In the main window, click the Annotate Values toolbar button, or select Source -> Annotate Values. (may already be shown)
These values are tied to C1 cursor, now 1670 s, since we searched to the first mismatch.

Our first driver "count <= 0;" We see that the "if" condition is "rstp == 1'b1", and the annotated value shows rstp is 0 now. So that's not it.



```
                  St1
102      if (rstp == 1'b1) begin
            St0
103         count <= 0;
            4'h0
104      end
105      else begin
106         case ({readp, writep})
                   St1    St1
107            2'b00: count <= count;
                      4'h0     4'h0
108            2'b01:
109               // WRITE
110               if (!fullp)
                       St0
111                  count <= count + 1;
                     4'h0     4'h0
112            2'b10:
113               // READ
114               if (!emptyp)
                       St1
115                  count <= count - 1;
                     4'h0     4'h0
116            2'b11:
117
118
```

Lets' try the next. Select Trace -> Drivers/Loads -> Next In This Instance, or select the Next In This Instance toolbar button.
Now we're in the "case" statement. Looking at the case condition ({readp, writep}), we see from the annotated values (St1 and St1) that the "2'b11" case applies.
Click the Next In This Instance toolbar button until the cyan current driver icon is on the source line that is inside the 2'b11 case.
That is, count <= count -1;, the concurrent read and write. (line 117)
When writing and reading, we should not be changing the counter.

This is our bug.

It should be count <= count;.

Edit source to fix the bug.

Select Scope -> Edit Source.
Fix the source code as described above, and save it.

Recompile and simulate in DVE to verify bug fix.

Again, you will need to re-compile the design outside of the GUI. This time however, you will need to use the following:

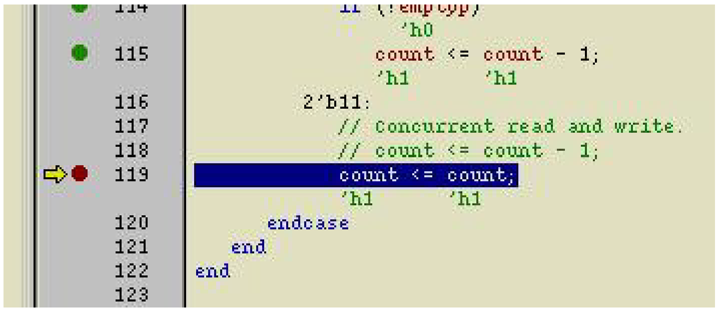
vcs -debug_all -f run1.f   (can use run_debug_all script)

Back in DVE, Select Simulator -> Setup. A new window opens, select OK which will run simv in interactive mode.

The signals (including the signal compare) are still availalbe in the Wave window so we can verifiy the fix.

Set a breakpoint on the line we just fixed to verify it.

Click on the + in front of test_fifo in the Hierarchy to expand it.
Double click the test_fifo.U1 in Hierarchy to populate the source window.
Scroll down to the line we edited (or use Find to jump to it.) (line 117)
Green dots to the left of the line number denote breakable lines during an interactive simulation.
Set a breakpoint by clicking the green breakable line on the source line we just edited (count <= count;)
It turns red to show that we have a breakpoint set here.



```
 ●  114                 if (!emptyp)
                           'h0
 ●  115                   count <= count - 1;
                          'h1      'h1
    116             2'b11:
    117                 // Concurrent read and write.
    118                 // count <= count - 1;
⇨●  119                 count <= count;
                        'h1      'h1
    120           endcase
    121       end
    122   end
    123
```

Now click the Continue toolbar button to run the simulation.
The simulation stops at the breakpoint, we see from annotated values that count is now 1. Fixed.

Disable the breakpoint and continue running.

Click on the breakpoint (red dot) so that it becomes a hollow red dot, denoting that it has been disabled.
Now click the Continue toolbar button to run the simulation.

It runs to completion without anymore checker errors. You're done.

File -> Exit to exit DVE.

DEBUG THE COUNTER MISMATCH BUG using SVA

Setup design, compile and setup simulation

We want to use SVA to debug the simulation for the counter mismatch, so we need to remove the fix and recompile.
This has been automated in the a script, so simply run run_debug_sva.
After the compile completes, start DVE with: dve &

Back in DVE, Select Simulator - Setup. A new window opens, select OK which will run simv in interactive mode.
You will notice that another window, Assertions, also opens. We will be using this window in this session.

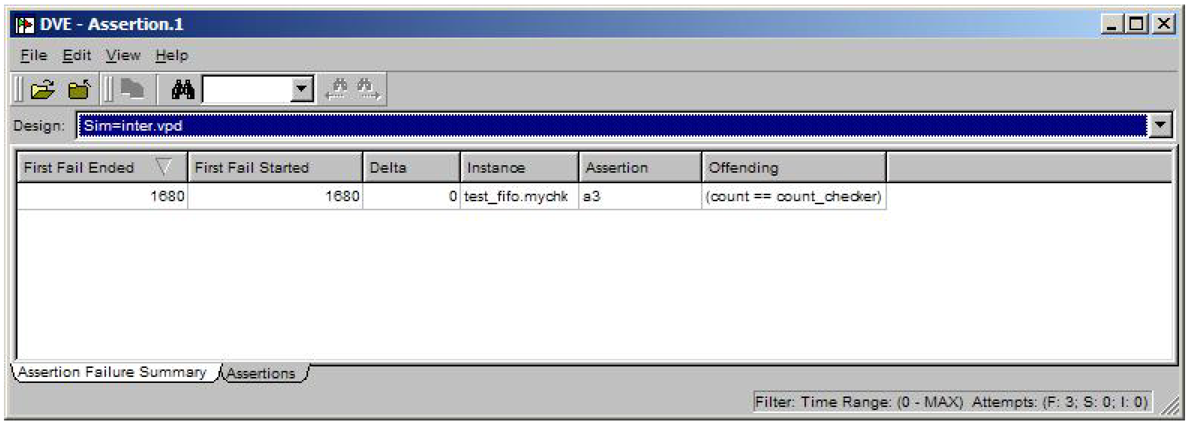
Simply click the continue (blue down arrow) in the banner to run the simulation to find the error. In the log window you will see the same Counter mismatch error as well as the SVA assertion error at time 1680.

```
emptyp = 1
FAILURE: Counter mismatch at time 1680000000000.000 ps!  Exiting test.
time = 1680000000000.000 ps
"fifo.sva", 18: test_fifo.mychk.a3: started at 1680s failed at 1680s
   Offending '(count == count_checker)'
$finish at simulation time 1690000000000.000 ps
```

Follow the checker that printed the error message.

Fi the Assertion window is not visible, click Window -> Panes -> Assertion to see the assertion debug information.

In Assertion window, double click the a3 assertion to populate the wave window.


DVE - Assertion.1
File Edit View Help
Design: Sim=inter.vpd
Assertion Failure Summary
Assertions
Filter: Time Range: (0 - MAX) Attempts: (F: 3; S: 0; I: 0)

First Fail Ended	First Fail Started	Delta	Instance	Assertion	Offending
1680	1680	0	test_fifo.mychk	a3	(count == count_checker)

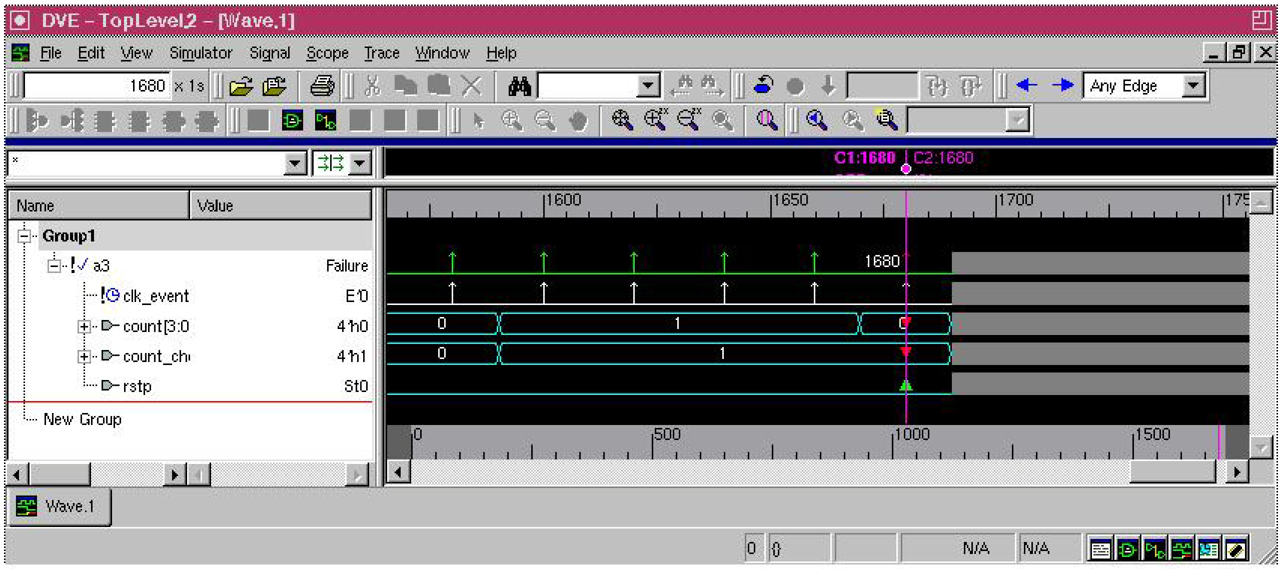
In the Wave window you will see the a3 assertion with the green up arrows.

You should see the first assertion failure (red arrow) at time 1680.

To see why this assertion is failing, we'll look at the parts of the assertion.

We have count and count_checker (our mismatched counters) in the Wave Window.

Since we are checking this assertion on the negative edge of clk, the previous values (time 1670) are the ones that show the failure.



Double click on the transition of count.
This will open the Drviers and Loads window, but since all of the drivers are in the same source file we'll look at that file. Select Close.

Now in the sourcecode window we see the first driver for count.

This is exactly where we were in the previous debug session, so we could continue in the exact same manner.

Using SVA reaches the same problem with with fewer/simpler steps, even for this simple example.

SUMMARY

We detected, found, fixed, and verified 2 bugs in this Verilog FIFO using the following features of DVE.

- Hierarchy, Wave, Source, Path Schematic, Console windows
- Post and interactive debugging.
- Quick Timescale zooming in Wave window
- Cursors and Markers and application time.
- Connectivity across hierarchy in path schematic window
- Expand paths in path schematic
- Annotated values and zooming in schematics
- Cross linking of schematic to source code
- Cross linking of wave to source code

- Cross linking of hierarchy to source code
- Searching text in source
- Edit source and recompile sim
- Multiple databases
- Waveform comparison
- Edge searching in waveforms
- Get drivers in waveforms
- Annotated values in source code
- Step through drivers in source code
- Interactive simulations with breakpoints and disabled breakpoints.
- SVA assertion reports
- Cross linking of assertions to wave and source code

5

Debugging simulation mismatches

Learning Objectives

After completing this lab, you should be able to:

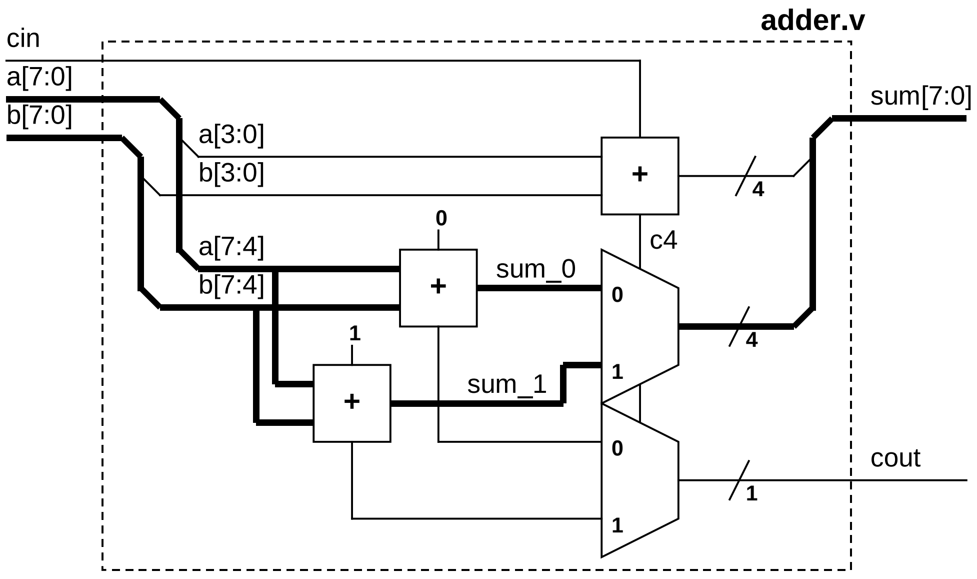
- Locate race condition in an existing Verilog design using the race checker utility in VCS
- Locate a simulation mismatch in an existing Verilog design

30 minutes

UNIT Unit 5

Getting Started

You will continue to use the Select Adder for this lab.



In the lab files, there are errors caused by race conditions. The goal is to use VCS to identify and locate the cause of the race conditions. Once, corrected, you will re-compile and re-simulate to verify the functionality of our corrections.

Here's a preview of what you'll do:

- In part A, you will compile and simulate to see if the adder is operating correctly. Then, we will use the +race compile-time switch and re-compile and re-simulate the adder. By examining the race.out file and locating the race condition in the source code, you will be able to make the proper correction.
- In part B, you will compile with $vcdpluson to generate a VPD dump file. Examining the differences in DVE, you will locate the race condition. The source code will be modified. Finally you will re-compile and re-simulate to verify the correctness of the adder operation.

Figure 5-1: Flow Diagram of Lab Exercise

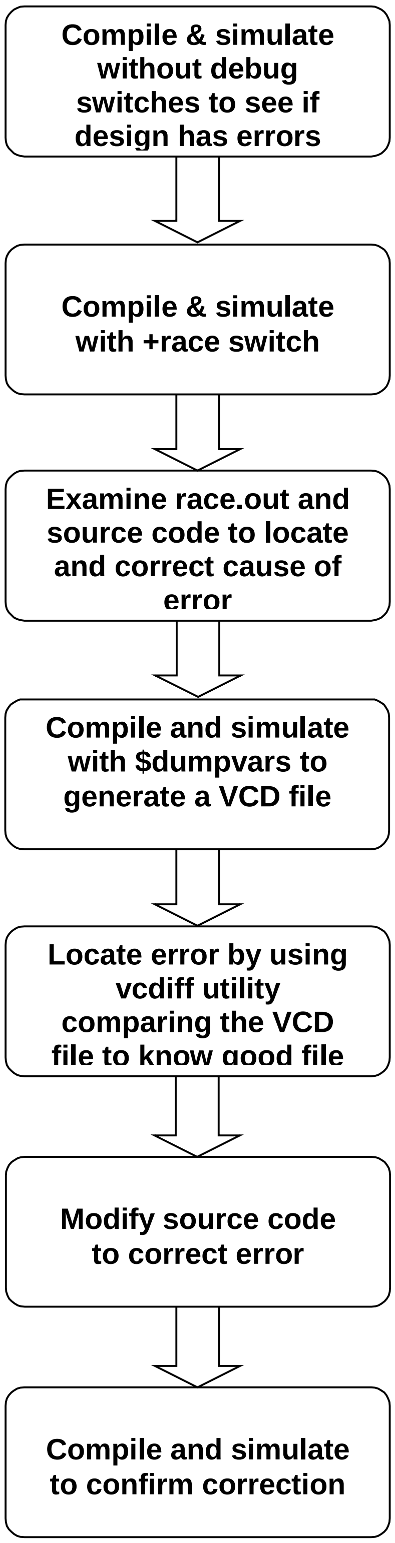

Compile & simulate without debug switches to see if design has errors
Compile & simulate with +race switch
Examine race.out and source code to locate and correct cause of error
Compile and simulate with $dumpvars to generate a VCD file
Locate error by using vcdiff utility comparing the VCD file to know good file
Modify source code to correct error
Compile and simulate to confirm correction

Figure 5-1: Flow Diagram of Lab Exercise

Part A: Debugging with +race Switch

Task 1 Verify Adder Functionality

The first step of any debugging process is to compile and simulate the design without using the debugging switches to see if there are any errors.

Go into lab5 part A directory.

1. shell> **cd ../lab5/parta**

2. shell> **vcs –f adder.f –R**

There are errors being reported.

```
***ERROR at time = 0 ***
a = 00, b = 00, sum = xx;  cin = 0, cout =x
…
…
```

Task 2 Re-Compile and Re-Simulate with +race Switch

With race conditions, it is generally difficult to see from the error message that the cause is Verilog race condition. When things don't look quite right, you might want to quickly check to see if the error is caused by race condition.

1. shell> **vcs –f adder.f –R +race**

This will generate a `race.out` file.

Take a look at the race.out file after the simulation has completed.

2. shell> **more race.out**

At time 0, three signals are reported being written and read at the same time.

```
0 "error_count": read  addertb (addertb.v: 32) && write addertb (addertb.v:21)
0 "sum_test": write  addertb.u1 (add8.v: 14) && read addertb (addertb.v:35)
0 "cout_test": write  addertb (add8.v: 15) && read addertb (addertb.v:35)
```

Where is the race condition in each of these cases?

3. Correct the race condition in the source code, repeat step 1 & 2 to make sure the race condition is resolved. (Hint: The race condition is in the testbench)

Part B: Debugging with $vcdplustraceon Task

Task 1 Verify Adder Functionality

Again, you want to verify the operation of the design first.

Go into lab5 part B directory.

1. shell> **cd ../partb**

2. shell> **vcs –f adder.f –R**

There are errors being reported.

```
***ERROR at time = 50 ***
a = 00, b = 00, sum = xx;  cin = 0, cout =0

***ERROR at time = 150 ***
a = 00, b = 01, sum = 00;  cin = 0, cout =0

…

…
```

Task 2 Re-Compile and Re-Simulate with +race Switch

Check the possibilities for race conditions by running a race report.

1. shell> **vcs –f adder.f –R +race**

Take a look at the race.out file after the simulation has completed.

2. shell> **more race.out**

At time 0, four signals are reported being written and read at the same time.

```
0 "sum_0": read  addertb.u1 (adder.v: 21) && write addertb.u1 (adder.v:15)
0 "sum_1": read  addertb.u1 (adder.v: 22) && write addertb.u1 (adder.v:16)
0 "sum_temp": write  addertb.u1 (adder.v: 14) && read addertb.u1 (adder.v:23)
0 "c4": write  addertb.u1 (adder.v: 13) && read addertb.u1 (adder.v:25)
```

Let's keep these in mind and dump a VCD+ file.

Task 3 Generate VPD dump file

In the testbench, sections of the code are already written for dumping VPD file. Take a look at it.

The relevant sections are as follows:

```
initial
begin
…
…
`ifdef vcdplusdump
  $display("\n*** VCD+ file dump is turned on ***\n");
  $vcdpluson;
  #1000;
  $vcdplusoff;
`endif
…
end
```

```
initial
begin
…
      if (error_count == 10) begin
…
        `ifdef vcdplusdump
          $vcdplusoff;
        `endif
…
end
```

The $vcdpluson task can be enabled with a +define+vcdplusdump compile-time switch.

The #1000 and $vcdplusoff are used in the first section of the code to control the size of the reference file. This was done so that the size of the file generated is manageable for download from the Synopsys web site.

In the second section, VPD dumpping is turned off after ten error counts are detected. Again, this is done to manage the VPD file size.

In your own debugging, you might want to consider using similar mechanisms for controlling VPD file size.

Compile and generate the VPD file.

1. `shell>` **vcs –f adder.f –debug_all –R +define+vcdplusdump**

You will got error message and `vcdplus.vpd` in current directory.

Task 4 Compare VCD File with Reference File

There is a reference VCD file already generated called "reference.dump". You will compare this reference file with the VCD file that you generated.

1. conver VPD to VCD

   ```
   shell> vpd2vcd vcdplus.vpd vcdplus.vcd
   ```

2. `shell>` **`vcdiff reference.dump vcdplus.vcd`**

```
vcdiff - Version X-2005.06-SP2
               Copyright (c) 1991-2003 by Synopsys Inc.
                         ALL RIGHTS RESERVED

vcdiff  reference.dump  vcdplus.vcd
< reference.dump: scopes:3 signals:18
> vcdplus.vcd: scopes:3 signals:18

--- addertb.sum_test --- 0 ---
<   0  00000000
---
>   0  xxxxxxxx

--- addertb.u1.sum --- 0 ---
<   0  00000000
---
>   0  xxxxxxxx


--- vcdiff summary ---
--- compares: 70
--- diffs: 2
```

At time 0, the adder.v generated a sum value of xxxxxxxx instead of 00000000 as was expected in the reference file.

Recall that the `race.out` file reported sum_0 and sum_1 had potential race conditions. The problem is likely to be in how we generated the sum.

Task 5 Examine File Differences in DVE

Look at the differences graphically.

Invoke DVE in post-processing mode.

1. `shell> dve&`

2. Select **File** -> **Open Database…**

   An Open Database Dialog window will open up.

3. In the Files pane, select "reference.dump" then click on **Open**

   As soon as you do this, DVE will translate VCD to VPD file. (the translated file name is "reference.dump.vpd")

   Let's also open the VCD file for the DUT.

4. In the Open Database Dialog window select "vcdplus.vpd" then click on **Open**

   Do you see that the new opened file is given the designator "V2=vcdplus.vpd" in the Open File pane near the top of the dialog window?

   Look at the signals in Waveform window.

5. Click + before "addertb" in Hierarchy pane.

   The modules and the signals now displayed in the Hierarchy pane and Variable pane are the signals captured in the last file that you opened – vcdplus.vpd. You can see this by looking at the title bar at the top of the Hierarchy pane.

6. Show all signals in u1 module into the Waveform window, left click select "u1" in Hierarchy pane and select "Add To Waves" in right click menu.

   Each signal in the Waveform window should have the V2 designator (leave mouse point on signals for one second), indicating that these are the vcdplus.vpd signals.

   Look at the DUT signals.

7. In Hierarchy pane, select "V1=reference.dump"

   Look at the same set of signals from u1 module.

8. Drag and drop the u1 module onto the "New Group" in the Waveform window

   This create another group u1_1 for new group signal, you can edit group name to ref_u1.

All the signals are identical with the exception of the sum signal. Use Zoom features to view the appropriate amount of data.

Task 6 Examine Source File to Locate Error

1. Double click `sum[7:0]` signal from the Waveform window(if you forget which is from V2, just put your pointer on it for one second), The adder.v module is opened in the source code pane.

   Recall what the race.out file reported. Where is the error?

   (Hint: this is a form of the Continuous Assignment Evaluation race condition)

2. Correct the error then re-compile and re-simulate to verify that the error is removed.

6 Fast RTL Verification

Learning Objectives

After completing this lab, you should be able to:

- Improve simulation performance by modifying Verilog code identified by VCS profiler utility as simulation bottlenecks
- Compile an existing Verilog design with the +rad compilation switch and execute the simulation binary to demonstrate an improvement in simulation performance
-

60 minutes

UNIT Unit 6

Getting Started

You will use the lossless codec for this lab:

You will improve the simulation performance at the RTL-level via two methods: changing the coding style and using VCS optimization switches.

This lab has two parts.

In Part A, you will use the +prof utility to locate simulation bottlenecks. You will then modify the Verilog code by changing the coding style to improve the simulation performance.

Here's a preview:

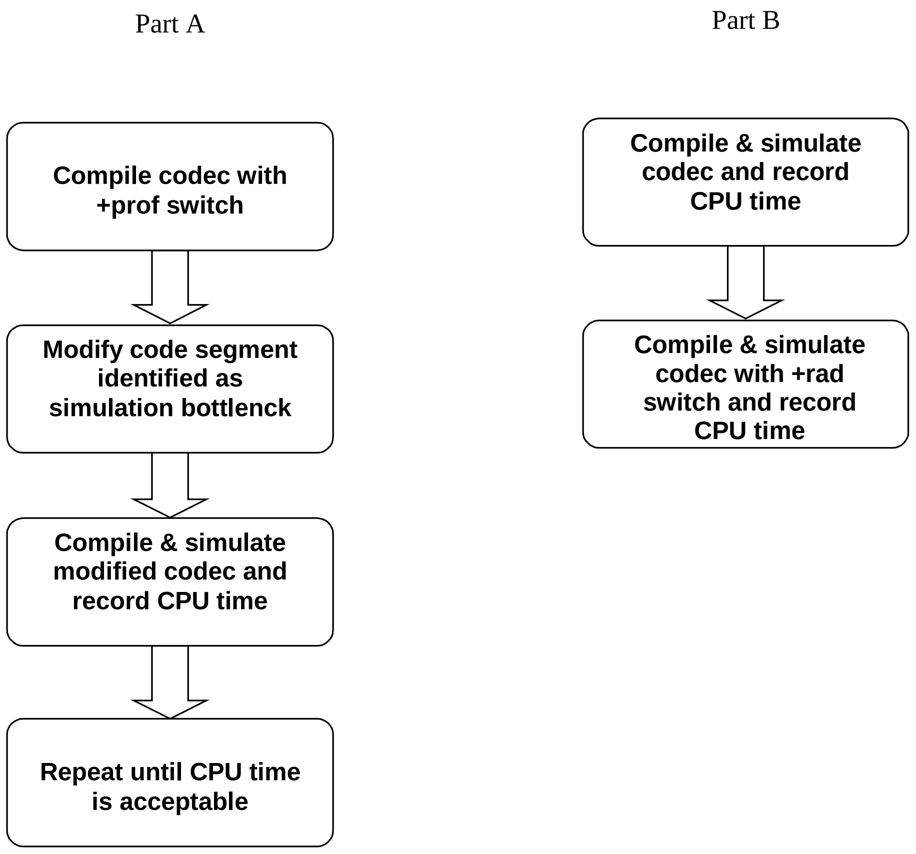
- Profile codec simulation with VCS profiler and record CPU run time
- Modify codec Verilog code segment identified as bottleneck
- Re-simulate and record CPU run time
- Repeat the first three steps until simulation speed is acceptable

In Part B, you will use the +rad VCS optimization switches to improve simulation performance.

Here's a preview:

- Simulate codec design and record CPU run time
- Compile & simulate codec with +rad switch and record CPU run time

Figure 6-1: Flow Diagram of Lab Exercise


Part A
Compile codec with +prof switch
Modify code segment identified as simulation bottlenck
Compile & simulate modified codec and record CPU time
Repeat until CPU time is acceptable
Part B
Compile & simulate codec and record CPU time
Compile & simulate codec with +rad switch and record CPU time

Part A: Using VCS profiler

Task 1 Record codec Simulation CPU Time

Go into the lab6 directory.

1. shell> **cd ./lab6/parta**

Compile and simulate the codec design without any optimization.

2. shell> **vcs –f codec.f CAM.v fileio.o –R**

3. Record CPU time & Data structure size

CPU time: ....

Data structure size: ....

For an input stimulus file of 323 bytes, this simulation time seems too long. Let's improve this simulation time.

Task 2 Profile Simulation CPU Time with +prof

First, run the profiling utility to see how simulation time is being spent.

1. shell> **vcs –f codec.f CAM.v fileio.o –R +prof**

Task 3 Evaluate codec Simulation Performance

Once the simulation run completes, examine the vcs.prof file.

1. Open **`vcs.prof`** with an editor

In Top Level View, the design uses greater than 90% of the CPU run time. This is good, you want simulation executing design operations rather than overhead.

```
================================================================================
                                TOP LEVEL VIEW
================================================================================
                          TYPE       %Totaltime
--------------------------------------------------------------------------------
                           DPI          0.00
                           PLI          0.00
                           VCD          0.00
                        KERNEL          0.00
                       MODULES        100.00
                      PROGRAMS          0.00
                    PROGRAM GC          0.00
--------------------------------------------------------------------------------
```

In Module View, the cam module requires most of the simulation time. This may or may not be a problem. Since this codec relies heavily on processing data in the cam module but it is an area to watch out for.

```
===========================================================================
                               MODULE VIEW
===========================================================================
Module(index)              %Totaltime    No of Instances   Definition
---------------------------------------------------------------------------
cam                    (1)     98.67          2            CAM.v:5.
Addr_gen                (2)     1.14           2           codec.v:347
```

In the Module to Construct Mapping, we start to see something a little more out of wack. The simulation time within the cam module is spent in the always block at lines 80-83 of the CAM.v source code. This could be a problem.

```
============================================================================
                       MODULE TO CONSTRUCT MAPPING
============================================================================


____________________________________________________________________________
                                  1. cam
----------------------------------------------------------------------------
Construct type    %Totaltime    %Moduletime     LineNo
---------------------------------------------------------------------
Always               40.80         41.35        CAM.v : 80-83.
Always               32.26         32.69        CAM.v : 36-61.
Always               12.33         12.50        CAM.v : 90-92.
Always               11.95         12.12        CAM.v : 86-88.
Always               0.76          0.77         CAM.v : 66-76.
Combinational        0.57          0.58         CAM.v : 5, 31, 33.
```

Please take a look at lines 80-83 of the CAM.v source code.

2. Open **`CAM.v`** with an editor and locate line 80

   Here's what this section of the code looks like:

```
always @(din or match_search or match3_dly or
         bit0 or bit1 or bit2 or bit3 or bit4 or bit5 or bit6 or bit7)
  for (l = 0; l <= 4095; l = l+1)
    match1[l] = ((match3_dly[l] | match_search) & (din == {bit7[l], bit6[l],
                  bit5[l], bit4[l], bit3[l], bit2[l], bit1[l], bit0[l]}));
```

   This looping structure is not efficient.

Task 4 Modify CAM.v module and Save as CAM1.v

1. Substitute the always block with the following:

```
assign match1 = (match3_dly | {4096 {match_search} }) &
                (({4096 {din[0]} } ~^ bit0) & ({4096 {din[1]} } ~^ bit1) &
                 ({4096 {din[2]} } ~^ bit2) & ({4096 {din[3]} } ~^ bit3) &
                 ({4096 {din[4]} } ~^ bit4) & ({4096 {din[5]} } ~^ bit5) &
                 ({4096 {din[6]} } ~^ bit6) & ({4096 {din[7]} } ~^ bit7));
```

   You will also need to change the match1 from reg type to wire.

2. Make the changes and save to "CAM1.v"

Task 5 Compile & Simulate to Record CPU Time

Compile & simulate the codec design with the modified cam module.

1. `shell>` **`vcs –f codec.f CAM1.v fileio.o –R`**

2. Record CPU time & Data structure size

   CPU time: ....

   Data structure size: ....

The simulation run time improved but not significantly. The bottleneck may have just move from one area to another. Please continue to look for the bottlenecks.

Task 6 Profile codec Simulation Performance

Run the profiling utility again to see how simulation time is being spent.

1. shell> **vcs –f codec.f CAM1.v fileio.o –R +prof**

Task 7 Evaluate codec Simulation Performance

Once the simulation run completes, examine the vcs.prof file.

1. Open **`vcs.prof`** with an editor

   In the Module to Construct Mapping, the simulation time in the cam module is now primarily spent in the always block, line 37-62 of the CAM1.v source code.

```
===========================================================================
                       MODULE TO CONSTRUCT MAPPING
===========================================================================
___________________________________________________________________________
                                  1. cam
---------------------------------------------------------------------------
Construct type    %Totaltime    %Moduletime     LineNo
---------------------------------------------------------------------------
Always               52.37         54.79        CAM1.v : 38-63.
Always               19.56         20.46        CAM1.v : 100-102.
Always               17.67         18.48        CAM1.v : 96-98.
Combinational        5.36          5.61         CAM1.v : 5, 33, 35, 90.
Always               0.63          0.66         CAM1.v : 68-78.
```

2. Open **`CAM1.v`** with an editor and locate line 38

   Here's what this section of the code looks like:

```
always @(posedge clk or negedge rstN)
begin
  if (!rstN) begin
    bit0 <= 0; bit1 <= 0; bit2 <= 0; bit3 <= 0;
    bit4 <= 0; bit5 <= 0; bit6 <= 0; bit7 <= 0;
  end
  else if (!ldN) begin
    bit0[0] <= din[0];
    bit1[0] <= din[1];
    bit2[0] <= din[2];
    bit3[0] <= din[3];
    bit4[0] <= din[4];
    bit5[0] <= din[5];
    bit6[0] <= din[6];
    bit7[0] <= din[7];
    for (i = 1; i <= 4095; i = i+1) begin
      bit0[i] <= bit0[i-1];
      bit1[i] <= bit1[i-1];
      bit2[i] <= bit2[i-1];
      bit3[i] <= bit3[i-1];
      bit4[i] <= bit4[i-1];
      bit5[i] <= bit5[i-1];
      bit6[i] <= bit6[i-1];
      bit7[i] <= bit7[i-1];
    end
  end
end
```

Examine this code closely, you will notice that this is really just a shift operation.

Modify CAM1.v module and Save as CAM2.v

Please recode the cam module as follows:

```
always @(posedge clk or negedge rstN)
begin
  if (!rstN) begin
    bit0 <= 0; bit1 <= 0; bit2 <= 0; bit3 <= 0;
    bit4 <= 0; bit5 <= 0; bit6 <= 0; bit7 <= 0;
  end
  else if (!ldN) begin
    bit0 <= {bit0[4094:0], din[0]};
    bit1 <= {bit1[4094:0], din[1]};
    bit2 <= {bit2[4094:0], din[2]};
    bit3 <= {bit3[4094:0], din[3]};
    bit4 <= {bit4[4094:0], din[4]};
    bit5 <= {bit5[4094:0], din[5]};
    bit6 <= {bit6[4094:0], din[6]};
    bit7 <= {bit7[4094:0], din[7]};
  end
end
```

1. Make the changes and save to “CAM2.v”

Task 8 Compile & Simulate to Record CPU Time

Compile and simulate codec with CAM2.v.

1. `shell>` **`vcs –f codec.f CAM2.v fileio.o –R`**
2. Record CPU time & Data structure size

   CPU time: ....

   Data structure size: ....

   The simulation run time has decreased significantly.

Profile codec Simulation Performance

Please take one last look at the profile report.

1. `shell>` **`vcs –f codec.f CAM2.v fileio.o –R +prof`**
2. Open `vcs.prof` file with editor

   The simulation time is now dominated by two always blocks.

   We have increased the performance sufficiently for now. Move on and try the effects of the optimization switches.

Part B: Using VCS Optimization Switches

Task 1 Record codec Simulation CPU Time

Go into the lab5 partb directory.

1. shell> **cd ../partb**

In this lab, we have reproduced the CAM2.v module and included it in the codec.f file. We also made the input stimulus file test larger. It is now seven times the size that was used in part A. Theoretically, this should simulate seven times slower. As you shall see, with the aid of performance options in VCS, the exact opposite is true. We will be able to run simulation at a much higher speed.

Please record the simulation time without optimization switches.

2. shell> **vcs –f codec.f fileio.o –R**

3. Record CPU time & Data structure size

CPU time: ....

Data structure size: ....

This simulation took a significant amount of time to complete. In the following steps you will improve this situation time.

Task 2 Simulate with +rad Optimization

Try using the +rad optimization switch.

1. shell> **vcs –f codec.f fileio.o –R +rad**

2. Record CPU time & Data structure size

CPU time: ....

Data structure size: ....

This is quite a difference in CPU time. The improvement in simulation speed depends heavily on your coding style and design complexity. Your own designs will experience a different speed up result.

7 Fast Gate-Level Verification

Learning Objectives

After completing this lab, you should be able to:

- Verify the Verilog Gate-Level netlist matches the RTL-Level simulation by compiling and simulating the Gate-Level netlist with the RTL-level testbench without enabling any optimization switch and without timing using VCS
- Demonstrate the same Verilog Gate-Level netlist when compiled and simulated with the +rad optimization switch exhibit an improvement in simulation performance
- Verify an existing Verilog Gate-Level netlist with timing matches the RTL-Level simulation

45 minutes

UNIT Unit 7

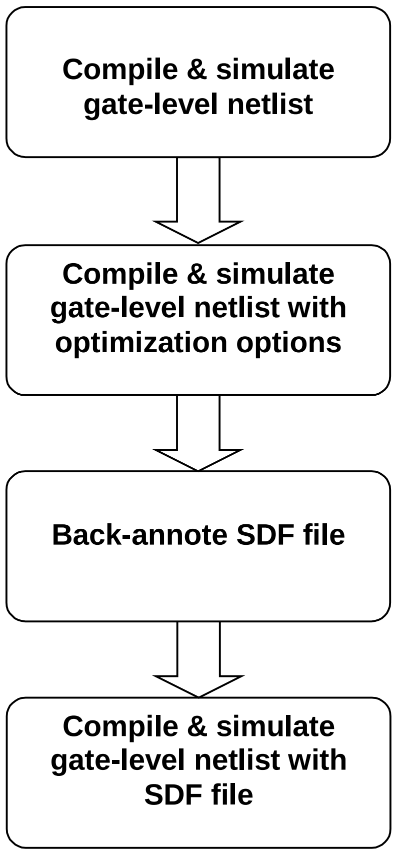
Getting Started

You will use a FIFO for this lab.

Here's a preview:

- Verify that the Verilog RTL code passes verification
- Using the same testbench, verify that a gate-level netlist generated from the same source code also simulates correctly.
- Compile and simulate the same gate-level netlist with optimization options and observe an increase in simulation performance.
- Add back-annotation system task to source code
- Compile and simulate SDF back-annotated gate-level netlist and observe that this also simulates correctly.

Figure 7-1: Flow Diagram of Lab Exercise


Compile & simulate gate-level netlist
Compile & simulate gate-level netlist with optimization options
Back-annote SDF file
Compile & simulate gate-level netlist with SDF file

Gate-Level Simulation

Task 1 Examine Gate-level Verilog Files

Go into the lab8 directory.

1. `shell>` **`cd lab7`**

Take a look at the gate-level Verilog files created by Design Compiler.

2. `shell>` **`more fifo32X8_gate.v`**

You should see the gate cells instantiated in all the _gate.v files. These were created through synthesis with Design Compiler with the vendor's technology libraries.

In order to simulate the gate-level Verilog files, you must also have the vendor's Verilog simulation files. In our case, this simulation file is located in vcs/lib/vendor_lib under the file name `core.v`.

Take a look at `core.v`.

3. `shell>` **`more ../lib/vendor_lib/core.v`**

As you scan through the core.v file, you will see both UDP (User Defined Primitives), functional modules, and gate-level cell modules.

Where are the timing information defined?

Task 2 Compile and Simulate the RTL Netlist

Let's verify that the RTL netlist works.

Here's the content of fifo.f:

```
fifo32X8tb.v
fifo_cntrl.v
fifo32X8.v
fifo_mem.v
ram16X8.v
-P ../lib/fileio2.0/fileio.tab
```

Compile and simulate the RTL code.

1. shell> **vcs –f fifo.f fileio.o –R**
2. Record CPU time & Data structure size

   CPU time: ....

   Data structure size: ....

Task 3 Compile and Simulate the Gate-level Netlist

Let's verify that this gate-level netlist works.

Here's the content of fifo_gate.f:

```
fifo32X8tb.v
fifo32X8_gate.v
-P ../lib/fileio2.0/fileio.tab
```

Notice that all the RTL codes are replaced by the single fifo32X8_gate.v gate-level netlist.

Compile and simulate the Gate-level netlist without SDF backannotation

1. shell> **vcs –f fifo_gate.f fileio.o –R**
2. Record CPU time & Data structure size

   CPU time: ....

   Data structure size: ....

This is a integrity test to verify that the gate-level netlist does perform the operation in the same way that the RTL code did. Notice that the same testbench is used for both the RTL and gate-level verification.

But, this is not the ideal way to perform the gate-level to RTL confidence test. A better way may be to use a formal verification tool like Formality,

If you really want to use VCS to check the functionality of the gate-level netlist, use the following compile-time options.

Task 4 Compile and Simulate with Optimization Options

1. `shell>` **`vcs –f fifo_gate.f fileio.o –R +rad +nospecify`**

2. Record CPU time & Data structure size

   CPU time: ....

   Data structure size: ....

   Notice how much faster the verification is?

Task 5 Back-Annotate SDF File

Let’s back-annotate the SDF file information for simulation with timing.

The only thing that you will need to do is to add the $sdf_annotate system task into the testbench. (Looks for the reserved space for this code)

1. Insert the following code in the fifo32X8tb.v testbench as follows:

```
initial
begin
  $sdf_annotate(“fifo32X8.sdf”, fifo);
end
```

Task 6 Simulate without Timing Check

If you are simulating your design with timing just to make sure that circuit delays does not introduce errors and you are using a separate timing verification tool, then, you should use the following compile-time options to get the best performance:

1. `shell>` **`vcs –f fifo_gate.f fileio.o –R +notimingcheck`**

2. Record CPU time & Data structure size

   CPU time: ....

   Data structure size: ....

Task 7 Simulate with Full Timing Check

If you are simulating your design with full timing check then use the following compile-time options.

1. `shell>` **`vcs –f fifo_gate.f fileio.o –R +neg_tchk`**

2. Record CPU time & Data structure size

   CPU time: ....

   Data structure size: ....

   Full timing verification will always take the longest time to run. Yet, it does not guarantee full timing verification. A better approach is to use timing verification tools for timing check and use VCS to verify circuit operations.

8 Coverage Metrics

Learning Objectives

Upon completion of this exercise you will be able to:

- Include coverage metrics in your VCS simulations
- Interpret the coverage metrics results and merge results from multiple simulations
- Use the auto-grading feature of VCS with Coverage Metrics to find a subset of test vectors that meet a user-defined coverage goal.

30 minutes

UNIT Unit 8

♦ *Part I- FSM Coverage*

1. Go to the subdirectory **lab8**

What other types of coverage are available with VCS Coverage Metrics? __________
____

What are the vcs & simv options to invoke each type of coverage? _______________
____

What option would you use to specify line and condition coverage only?__________

Let's take a closer look at FSM (Finite State Machine) coverage.

1. Compile so that FSM coverage is enabled.

   VCS COMMAND LINE:____

2. Rerun the simulation with FSM coverage.

   SIMV COMMAND LINE:____

3. Invoke cmView with FSM coverage.

   CMVIEW COMMAND LINE:____

4. In the main cmView window, left click on the **FSM Coverage** icon.
5. Left click on Module List to display all modules where a finite state machine was found. You should see that the CPU module has one state machine.
6. Double click left on the CPU module (under the **List of all modules**). This will bring up a window with coverage details for the FSM in the CPU module. Double click the **Transitions** tab you will see a matrix of transitions. A green check indicates the transition was covered during your simulation. A red X indicates that the transition

was not covered. An empty (gray) box indicates that VCS found that transition was not possible.

Which state transitions were not covered?________________________________

____

7. Click on the **States** tab. This displays each possible state in the FSM and whether it was covered. There are five states in this FSM and each one was covered at least once during the simulation run.
8. Click on the **Reachability** tab. This shows information about transitions between two states *through intervening states*. Lets look at an example…
9. Click on the yellow box that represents the transition from MEMORY to FETCH. The **1 / 2** displayed in this box indicates that there are two possible transitions from MEMORY to FETCH and that only one was covered. After clicking this transition you will see that VCS has determined two possible transitions:

   MEMORY -> WRITE_BACK -> FETCH (covered)

   MEMORY -> FETCH (not covered)
10. Exit the cmView GUI

♦ *Part II - Merging Coverage Results and Auto-Grading*

So far we have been writing coverage results into default filenames under the simv.cm directory. For example…

simv.cm/coverage/verilog/test.line

simv.cm/coverage/verilog/test.fsm

This is fine as long as there is only one test vector to run. But what if there are multiple test vectors? You don't want to overwrite the default filenames with each new test vector that is simulated. It is more useful save the coverage results for each simulation with a new vector and then merge the results . In this section we will examine how to do this.

1. First clean up the directory you are working in. Make sure you have exited the cmView GUI and then run the **clean** script.
2. The directory **CODE** contains five different tests. Copy the first test into the run directory

```
cp CODE/text1 text_segment
```

cp CODE/data1 data_segment

3. Run a vcs compile and enable all types of coverage (see “run_all” script)

   VCS COMMAND LINE____

4. Examine the contents of **simv.cm/coverage.** It should be empty at this point.
5. Run simv with all coverage types enabled. Be sure to include the option to rename the coverage metrics report files. Use **test1** as the filename root for the report files.

   SIMV COMMAND LINE____

6. One by one, copy the other vectors into the run directory and rerun the simulation with all coverage types enabled. For each simulation run, be sure to rename the coverage report files. (A script called **run_all** is provided.)

   What file contains condition coverage results for test5?_____________________

   What file contains line coverage results for test2?__________________________

7. We will now merge the data together. At the command line type:

   vcs –cm_pp –cm_dir test1 –cm_dir test2 –cm_dir test3 –cm_dir test4 –cm_dir test5 –cm_name merged

This will create a merging of all the 5 tests into a merged database and the merged coverage report will be stored in "simv.cm/reports" with the filename prefix "merged" (i.e. simv.cm/reports/merged.long_l will contain the merged coverage results from the 5 tests).

8. You can also observe these data and do test grading through GUI. Invoke the cmView GUI. Type:

   vcs –cm_pp gui –cm line+tgl+cond+fsm –cm_dir test1 –cm_name test1

Go to the “**File**” on the menu bar and drop down to “**Open Coverage Statement**”. This will open the hierarchy and coverage window.   On the coverage menu bar select “**add**”, this opens the window that will allow you to add the line coverage data for grading. In the filter window or thru selecting, set the path for ./test2/*.line select **test2.line** and add it. You can do the same steps to add other tests (test3 to test5). Close the window after adding the line coverage. You can now compare tests using the compare menu.

9. We will use cmView’s auto-grading feature to determine a subset of test vectors that meet a user-defined coverage goal. From the main cmView window select

Tools -> AutoGrading

This brings up the **cmAutoGrade** window.

10. Click any one of the three tabs for Statement, Condition, or Toggle coverage to view the data for each test listed in the Test Name column.

    The four kinds of coverage data are described as follows:

    Incremental data is the amount of additional coverage provided by a given test over the previous test.
    Difference data is the difference in coverage between a given test and the accumulated total up to the previous total.
    Test coverage data is the coverage for a given test. That is, the data reflects the total coverage for that test only.
    Accumulated data is the cumulative result of all tests added in the current session up to that point in the order of tests.

11. Enter **65** desired test coverage goal and select **Line Coverage**. Then **click Run AutoGrade** The Test Grading window appears.

    What subset of test vectors are required to obtain 65% line coverage? _____________

    ____

12. Exit the cmView GUI.

Congratulations. You’re done with the Coverage Metrics lab!