Before getting started with actual examples, here are a few notes on conventions. First, command lines and sequences take the same arguments on all supported operating environments, including Linux, Windows and the various Unix systems. When an example command is shown in a figure, the generic prompt character "% " takes the place of whatever prompt string is appropriate for your system. Under Windows, the commands are invoked in a command window.

Second, when creating a file to hold Verilog code, it is common to use the ".v" or the ".vl" suffix. This is not a requirement imposed by Icarus Verilog, but a useful convention. Some people also use the suffixes ".ver" or even ".vlg". Examples in this book will use the ".v" suffix.

So let us start. Given that you are going to use Icarus Verilog as part of your design process, the first thing to do as a designer is learn how to compile and execute even the most trivial design. For the purposes of simulation, we use as our example *the* most trivial simulation, a simple *Hello, World* program.

module main;

initial

begin

$display("Hello, World");

$finish ;

end

endmodule

Use a text editor to place the program in a text file, *hello.v*, then compile this program with the command:

% iverilog -o hello hello.v

The results of this compile are placed into the file "hello", because the "-o" flag tells the compiler where to place the compiled result. Next, execute the compiled program like so:

% vvp hello

Hello, World

And there it is, the program has been executed. So what happened? The first step, the "iverilog" command, read and interpreted the source file, then generated a compiled result. The compiled form may be selected by command line switches, but the default is the "vvp" format, which is actually run later, as needed. The "vvp" command of the second step interpreted the "hello" file from the first step, causing the program to execute.

The "iverilog" and "vvp" commands are the most important commands available to users of Icarus Verilog. The "iverilog" command is the compiler, and the "vvp" command is the simulation runtime engine. What sort of output the compiler actually creates is controlled by command line switches, but normally it produces output in the default vvp format, which is in turn executed by the vvp program.

As designs get larger and more complex, they gain hierarchy in the form of modules that are instantiated within other modules. And as designs start including lots of modules, it becomes convenient to organize them into multiple files. A common convention is to write one moderate sized module per file (or group related tiny modules into a single file) then combine the files of the design together during compilation. For example, the counter model in counter.v:

module counter(out, clk, reset);

parameter WIDTH = 8;

output [WIDTH-1 : 0] out;

input clk, reset;

reg [WIDTH-1 : 0] out;

wire clk, reset;

always @(posedge clk)

out <= out + 1;

always @reset

if (reset)

assign out = 0;

else

deassign out;

endmodule // counter

and the test bench in counter\_tb.v:

module test;

/\* Make a reset that pulses once. \*/

reg reset = 0;

initial begin

# 17 reset = 1;

# 11 reset = 0;

# 29 reset = 1;

# 11 reset = 0;

# 100 $stop;

end

/\* Make a regular pulsing clock. \*/

reg clk = 0;

always #5 clk = !clk;

wire [7:0] value;

counter c1 (value, clk, reset);

initial

$monitor("At time %t, value = %h (%0d)",

$time, value, value);

endmodule // test

are written into different files.

The "iverilog" command supports multi-file designs by two methods. The simplest is to list the files on the command line:

% iverilog -o my\_design counter\_tb.v counter.v

% vvp my\_design

This command compiles the design, which is spread across two input files, and generates the compiled result into the "my\_design" file. This works for small to medium sized designs, but gets cumbersome when there are lots of files.

Another technique is to use a commandfile, which lists the input files in a text file. For example, create a text file called "file\_list.txt" with the files listed one per line:

counter.v

counter\_tb.v

Then compile and execute the design with a command like so:

% iverilog -o my\_design -c file\_list.txt

% vvp my\_design

The command file technique clearly supports much larger designs simply by saving you the trouble of listing all the source files on the command line. Name the files that are part of the design in the command file and use the "-c" flag to tell iverilog to read the command file as a list of Verilog input files.

As designs get more complicated, they almost certainly contain many Verilog modules that represent the hierarchy of your design. Typically, there is one module that instantiates other modules but is not instantiated by any other modules. This is called a *root* module. Icarus Verilog chooses as roots (There can be more than one root) all the modules that are not instantiated by other modules. If there are no such modules, the compiler will not be able to choose any root, and the designer must use the "-s*root*" switch to identify the root module, like this:

% iverilog -s main -o hello hello.v

If there are multiple candidate roots, all of them will be elaborated. The compiler will do this even if there are many root modules that you do not intend to simulate, or that have no effect on the simulation. This can happen, for example, if you include a source file that has multiple modules, but are only really interested in some of them. The "-s" flag identifies a specific root module and also turns off the automatic search for other root modules. You can use this feature to prevent instantiation of unwanted roots.

As designs get even larger, they become spread across many dozens or even hundreds of files. When designs are that complex, more advanced source code management techniques become necessary. These are described in later chapters, along with other advanced design management techniques supported by Icarus Verilog.