# EE 361 Fall 2003        University of Hawaii
# Verilog Introduction

G. Sasaki
10/17/03

## 1. Introduction

This is an introduction to *verilog*, which is a hardware description language (HDL) for digital circuits. Section 2 describes how to design circuits in verilog. Section 3 has a brief overview of simulators and test benching. An example of designing and debugging a circuit using *veriwell* is presented in Chapter 4.

## 2 Verilog

Verilog can be used to describe combinational and sequential circuits. Verilog code

- Can be used to simulate circuits using a simulator such as *veriwell* or *Modelsim*, or
- Can be converted into an actual hardware design using a *synthesizer* such as the one in Xilinx's Webpack.

In verilog, circuits can be defined as *module*s. A template of a module is shown in Figure 2.1. Stick to this template to minimize mistakes. Later, when you gain more experience, you can use other styles and conventions of writing verilog code.

The circuit modules are composed of combinational subcircuits, sequential subcircuits, or a mixture. In Subsection 2.1 we describe combinational subcircuits, and in Subsection 2.2 we describe sequential subcircuits.

### 2.1 Combinational circuits

A combinational circuit has a set of inputs (x1, x2, ..., xk) and a set of outputs (y1, y2,..., ym). Each output essentially behaves like the output from a mathematical function, e.g., $y = f(x1, x2, ..., xk)$. This can be modeled within a verilog module by using a *continuous assign* or *procedural always*.

Continuous assign has the following form:

**assign**  y = expression with terms for x1, x2, ..., xk;

The variable `y` must be a *wire* variable, and "expression" is a mathematical expression, e.g., a boolean or arithmetic expression. The expression *cannot* have begin-end blocks, case statements, if-else statements or anything else. The variables `x1, x2, ..., xk` can be wire or reg variables.

The operations are on values with the same number of bits. For example, suppose y is 3 bits (e.g., you declared `wire [2:0] y;`) and variable `x1` is a single bit. Then

```
assign y = x;
```

does not make sense because you are assigning a 1-bit value into a 3-bit variable.

```verilog
// Module and port list.  Note that by convention the output ports
// come first, then clock inputs and resets, then input ports.
module name_of_module(output_port1, clk1, input_port1, input_port2,...)

// List of ports.  Note that the ports act as wires, but you can make
// an output port be a reg variable by declaring it so.
input in_port1;
input in_port2;
input clk1; // clock input
output out_port1;

// Declare wire and reg variables
wire [3:0] x;
reg y;

// Note that the declaration of port type and variables are
// done at the top of the module.  The rest of the module defines
// how the circuit behaves.  This is done by the "assign", "always",
// and module instantiations.  These can be listed in any order
//-----------------------------------------------------------------
// Continuous assign:  left side of "=" is a wire var.  Right
//  side is a boolean or arithmetic expression.
assign x = x + ({y,y,y,y} & 3); // Note that the processing is on 4-bits

// Procedural always for combinational circuits.  All assignments
// should be blocking ("=")
always @(variable1 or variable2 or variable3)
   begin
   case statements
   blocking assignments
   if-else
   end

// Procedural always for sequential circuits.  All assignments
// should be nonblocking ("<=")
always @(posedge clk1)
   begin
   case statements
   nonblocking statements
   if-else
   end

// Module instantiations.  These are for modules "Circuit1"
// and "Circuit2".  Note that wire variables are used to connect
// the instantiations with other things.
Circuit1 name_of_instance(wire_var1, wire_var2,...);
Circuit2 name_of_instance2(wire_varx, wire_vary,...);

endmodule
```

**Figure 2.1.  Verilog module template.**

Continuous assignment statements are limited to modeling simple circuits. Often they are just used to connect one variable to another, e.g.,

```
assign y = x;
```

Procedural-always statements can model any combinational circuit, so they are used more often. A procedural-always has the following form:

```
always @(x1 or x2 or ... or xk)
```

A description of how the output variable y is updated whenever the inputs change. In other words, it describes y = f(x1,x2,....,xk)

The basic form is to have all the inputs listed in the *sensitivity list*, e.g., "x1 **or** x2 **or** ... **or** xk". Whenever one or more of the input values change, the output values are updated. In this case, the output is y. The output variables are always reg variables.

The following are two examples.

```
always @(x1 or x2)   y = x1 + x2;
```

```
always @(x1 or x2 or select)
   case(select)
      0: y = x1 + x2;
      1: y = x1 - x2;
   endcase
```

In many cases, more than one line will be needed to describe how the output y is updated. To group multiple lines into a single entity, you can bind them into a begin-end block. This is similar to curly brackets "{ }" used in the C programming language.

```
always @(x1 or x2 or x3)
    begin
    if (x1 >= 0) choice = x1 + 3;
    else choice = x2;
    case (choice)
      0: y = x2 + x3;
      1: y = x2 - x3;
    endcase
    end
```

The begin-end block explains how the new value of output y is determined. Note that we have a variable "choice" that is not an input or output. We will discuss this variable later.

It is assumed that within the begin-end block, the lines are "executed" starting from the top and proceed downwards. Thus, it is interpreted as if it were part of a C program. For the example, to determine the new value of y, start with the if-else. Then continue by considering the case-statement.

Your procedural-always should lead to the truth table for the subcircuit.  Consider the next example (assume all the variables are 1-bit):

```
always @(x1 or x2 or s)
   begin
   if (s == 1) h = 0;
   else h = 1;
   case(h)
      0:  y = x1|x2; // AND the inputs
      1:  y = x1&x2; // OR the inputs
   endcase
   end
```

This procedural-always describes a combinational circuit that has a select input "s".  If s = 1 then the output y = x1 | x2.  Otherwise, y = x1 & x2.  This leads to the following truth table:

| Input | | | Output |
|---|---|---|---|
| s | x1 | x2 | y |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

If you cannot generate a truth table then obviously the procedural-always does not describe a combinational circuit.

Here are some rules in regards to writing a procedural-always:

*Rule 1*. Variables on the left side of an equality "=" must be reg variables.  For example, the variable "t" in the line "t = x1 + x2;" must be a reg variable.

*Rule 2*.  There are three kinds of variables.  There are input variables such as x1, x2, and s; and there are output variables such as y.  There can be other variables that are not inputs or outputs, e.g., in the examples above, "choice" and "h" are not inputs or outputs.  They are used to store intermediate values as we proceed to determine the output values.  You can think of them as "wires" that are internal to the subcircuit.  We will refer to them as "intermediate variables".  These variables should be reg variables.

As much as possible, in a procedural-always, "update" output and intermediate variables at most once.  For example, we want to *avoid* procedural-always such as the following:

```
always @(x1 or x2)
   begin
   y = x1 + x2;
   if (y > 4) y = x1;
   else y = x2;
   end
```

Here, the output variable y could get updated two times, at y = x1+x2 and at y = x1.  If these variables represent "wires" then it doesn't make sense to have them change values twice each time the inputs change.  A better implementation is as follows:

```
always @(x1 or x2)
   begin
   r = x1 + x2;
   if (r > 4) y = x1;
   else y = x2
   end
```

Then both `r` and `y` are updated once (y is updated as either y = x1 or y = x2, but is never updated twice).

*Rule 3.*  Within a begin-end block, you can have ordinary assignments (e.g., "y = x1+x2;), if-else statements, and case statements.

A common mistake is to interpret modules as "C functions".  Modules are really descriptions of circuits.  Thus, any instantiations of a module is just implying that a circuit should be placed there.  On the other hand, for a procedural-always, the begin-end block is an explanation of how an output value is computed.  Hence, it doesn't make sense to instantiate a circuit module within a procedural-always.

## 2.2  Sequential circuits

A sequential circuit has a clock input, data inputs, and outputs.  It also has a "state" which is stored in a "state register".  Note that the state register can be as simple as a few D flip flops or as complicated and large as a combination of RAM, register files, and counters.  When designing a verilog subcircuit, the state register is represented by a reg variable.  The following is an example of a procedural-always that implements a sequential subcircuit

```
always @(posedge clock) q <= d;
```

The variable "clock" is the clock input.  The procedural-always will update whenever "clock" has a positive edge.  The update is "q <= d".  Here, "q" is the state register that has its value updated by input "d".  This example is for a D flip flop.  Notice that instead of "=" we use "<=".  The assignment "=" is called a *blocking assignment*, and "<=" is called a *nonblocking assignment*.  These will be explained a little later.

Consider the next example for a T flip flop.

```
always @(posedge clock)
   begin
   if (t==1) q <= ~q;
   end
```

The example shows that we can use begin-end blocks and if-else.  You can also use case statements.

Basically, a procedural-always for a sequential circuit has the following form.

```
always @(posedge clock)
```

> Update state registers using nonblocking
> assignments "<=".

Note that when updating the state registers, we want to update *all* the registers at once, i.e., all their D flip flops get updated on the positive clock edge. This models the way real sequential circuits work. To get this effect, nonblocking assignments are used. Next is an explanation of the difference between blocking and nonblocking assignments.

In the case of nonblocking assignments ("<="), all the assignments are activated at once. It is "nonblocking" because the statements are executed simultaneously. Consider the following example with D flip flops labeled A, B, and C.
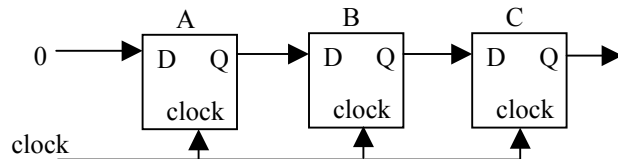


When the positive edge of the clock occurs, all D flops are updated. The following procedural-always models this

```
always @(posedge clock)
   begin
   A <= 0;
   B <= A;
   C <= B;
   end
```

Again, all of the updates are done simultaneously. For example, suppose initially A = 1, B = 0, and C = 1. Then after the positive clock edge, A = 0, B = 1, and C = 0, such as shown below.



*Before clock edge*

*After clock edge*
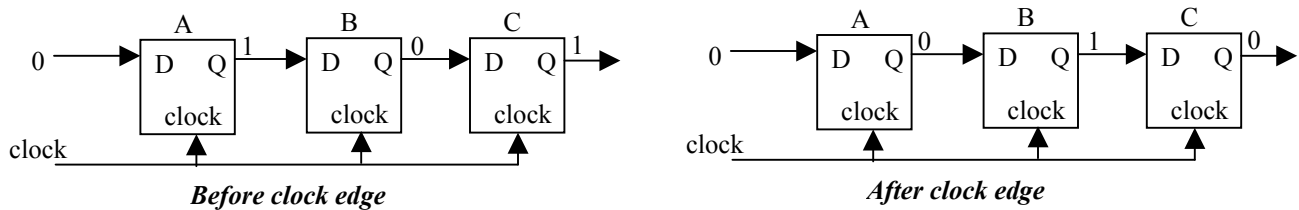
Blocking assignments ( "=") are different which is illustrated in the next example.

```
always @(posedge clock)
   begin
   A = 0;
   B = A;
   C = B;
   end
```

In this case, when there is a positive clock edge, each line is activated one at a time. In other words, "A = 0" is updated, then "B = A" is updated and finally "C = B" is updated. Note that the line "A = 0" essentially "blocks" the assignments below it. In other words, line "A = 0" must be updated before we

consider "B = A" and "C = B".  In turn, line "B = A" blocks "C = B".  This is *not* how a sequential circuit behaves.

Let us take a look at the two examples below of updating our D flip flops.  Assume initially that (A,B,C) = (1,1,1).  If the assignments are nonblocking then, after the positive clock edge, we have (A,B,C) = (0,1,1).  This is what we want since all the D flip flops should be updated together.  On the other hand if the assignments are blocking then, after the positive clock edge, we have (A,B,C) = (0,0,0).

```
begin
A <= 0;
B <= A;
C <= B;
end
```

```
begin
A = 0;
B = A;
C = B;
end
```

The following are some rules to design sequential subcircuits using procedural always.

*Rule 1*.  Always use nonblocking assignments "<=" and *never* use blocking assignments "=".

*Comment*:  Procedural always can be used to model combinational circuits or sequential circuits.  In the case of combinational circuits, always use blocking assignments "=".  In the case of sequential circuits, always use nonblocking assignments "<=".  Never mix the two in a procedural-always.

*Rule 2*.  State registers are reg variables.

*Rule 3*.  There are three types of variables: clock variable, input data variables, and state variables.  Your output should equal the state, so there is no need for output variables.  The clock variable should appear only once within "always @(posedge clock)".  For each nonblocking assignment, the left hand side should be a state variable.

*Rule 4*.  Each state variable should be updated at most once.

*Rule 5*.  You can use if-else and case statements.

For the rest of this section we will give an example of a sequential circuit.  It is a 2 bit counter with a two bit select "s".  If s = 0 then the counter resets to 0.  If s = 1 then the counter counts up, and if s = 2 then the counter counts down.  If s = 3 then the counter holds its value.

```
module counter2(q,clock,s,d)
out [1:0] q; // 2-bit output
in  clock;
s   [1:0] s; // Select input
in  [1:0] d; // Parallel load input

reg [1:0] q;  // This is our state variable

always @(posedge clock)
   begin
   case (s)
      0: q<=0;
      1: q<=q+1; // Counting up.  Note that the count wraps around
                 //    when it goes past the value 3
      2: q<=q-1; // Counting down.  Also has wrap around
      3: q<=q;   //
   endcase        // Actually, the begin-end is unnecessary
   end
endmodule
```

Suppose we want to modify the counter so that it counts in the Gray code (which is used in K maps). Recall that the Gray code is 00, 01, 11, 10, rather than 00, 01, 10, 11.

We can modify counter2 by having it output differently. When counting up, the new counter should count 00, 01, 11, 10, 00...., and when counting down, it should count 00, 10, 11, 01, 00....  Instead of having the output value be the same as the state of counter2, have it be different values but dependent on the state.  In particular, if the state = 2, the output should be 11, and if the state = 3, the output should be 10.  We build a separate combinational subcircuit using another procedural-always.

```
module counterGray(y,clock,d,s)
out [1:0] y; // 2-bit output
in  clock;
in  [1:0] d; // Parallel load input
s   [1:0] s; // Select input
reg [1:0] q;  // This is our state variable

always @(posedge clock) // This is the same as before
   begin
   case (s)
      0: q<=0;
      1: q<=q+1; // Counting up.  Note that the count wraps around
                 //    when it goes past the value 3
      2: q<=q-1; // Counting down.  Also has wrap around
      3: q<=q;   //
   endcase        // Actually, the begin-end is unnecessary
   end

// This is added to output the Gray code.   It's a combinational circuit,
// with the state being the input, and y being the output.
always @(q)
   case (q)
      0:  y = 0;
      1:  y = 1;
      2:  y = 3; // which is 11 in binary
      3:  y = 2; // which is 10 in binary
   endcase
endmodule
```

## 2.3.  Bucknell Verilog Handbook

In the following table is a list of sections to review in the Bucknell Verilog Handbook.  (Note that the Bucknell Verilog Handbook is a bit confusing because it explains verilog in the context of both circuit

design and simulation and mixes them together. Actually, verilog used to design circuits is a little different than verilog used for simulation.)

| Section | Comments |
|---|---|
| 2.2. Lexical conventions | It explains basic syntax including how to specify constants in binary, hexadecimal, etc. |
| 2.3. Program structure | It explains the basic structure of modules. |
| 2.4.  Data types, but skip Subsection 2.4.2. | It explains variables, variables that are arrays of bits (e.g., reg [7:0] A;), the concatenation operation (e.g., "{x, y, z}"), and the repetition operation.  These help in writing concise modules. |
| 2.5  Operators. | The operators are similar to C language operators.  Note that the shifting operations are "<<" and ">>". |
| 2.6  Control Constructs but skip Subsection 2.6.2. | It explains if and case statements. |
| 2.7 Other statements but skip Subsections 2.7.1. | It explains continuous assignment, blocking, and nonblocking assignments. |

# 3 Simulation and Testbenches

After you build modules, you'd like to test them using a verilog simulator such as *veriwell*. The software tool *veriwell* is known as a *discrete time simulator*. It will simulate a circuit (specified in verilog) over a time period starting from time 0. The simulator will divide time into *time units* or *time steps*. You can think of each unit as the "tick" of some clock. Each time unit corresponds to some small duration such as 1 nanosecond or 0.1 nanoseconds.

To run a simulator, you first initialize it so that it starts from time 0. The simulator keeps track of the time by a variable, and in the case of veriwell the variable is $time. When the simulator runs a simulation, it computes what the wire and register values should be as time goes on. In particular, it determines what the variables should be at time 1 based upon what the variables were at time 0. Then it determines what the variables should be at time 2 based upon the variables are at time 1. It continues in this way until it reaches some stopping time. In the case of veriwell, $stop determines the stopping time.

What can you do with the simulator? You can test your verilog modules as if you were testing a real hardware circuit such as an IC chip. Figure 3.1 shows a set-up to test the IC chip. The chip is on a protoboard. Its outputs are connected to probes, which are LEDs. A clock generator is connected to the clock input. The other inputs are connected to wires that can be set to 5v and Gnd. To check whether the chip works or not, we can set the input wires to different voltages and observe the probes. If the outputs are what we expect then the chip works. This set-up is a *test bench*.
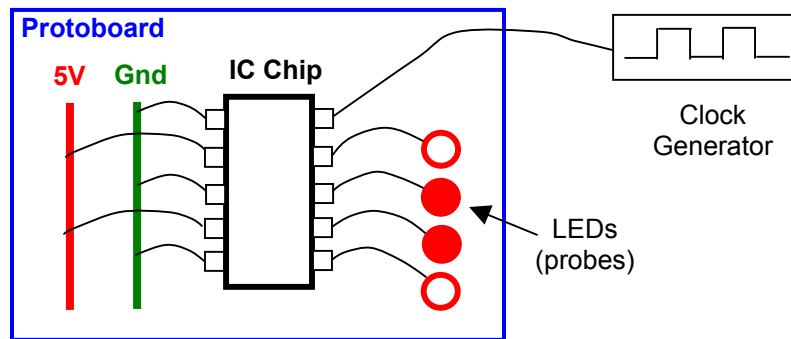


**Figure 3.1. A testbench**

We can also test bench our verilog modules. The following are verilog statements that are useful for test benching. Look these up in the Bucknell Verilog Handbook and read the descriptions.

| Verilog Statements | What are they used for |
|---|---|
| initial | This statement is used to initialize variables for simulation. However, with delays, this statement can be used to generate input signals that change over time. |
| #n | This inserts a delay of n time units. This is used in combination with initial to create input signals for your test bench. |
| $display | This is used to display variable values, like probes. This is similar to printf in the C language. |
| $monitor | This is also used to display variable values, like probes. This is similar to $display but it can occur only once in the verilog code. This can occur only once in your verilog code, while $display can occur many times. |

Figure 3.2 is a test bench for a module called icChip which is a sequential circuit.  The test bench is used for simulations only.  You do not synthesize a test bench because it is not a circuit.

```verilog
// Note that the test bench module is similar to main( ) in C programs.
// It has no ports.  Also note the test bench can
// have any name, and in this case it's called "testbench"
module testbench;

// We need variables to connect to inputs and outputs of icChip.
// Note that signals to the inputs of icChip must be from reg variables.
// The outputs from icChip are connected to
// wire variables.
reg clock;
reg [1:0] A;
reg [2:0] B;
wire [2:0] C;

// The following is the instantiation of icChip.  Note
// that the wire and reg variables are connected to it.
icChip  c1(A,B,C,clock);

// The following is the clock generator.  It's initialized
// to 0 and then it toggles every time unit.  Thus, the
// clock period is 2 time units.
init clock = 0;
always #1 clock = ~clock;

// The following shows how to set inputs to different
// values over time.  Note that the assignments are blocking
// so that they are executed in sequence.
init
    begin:  Input values for testing
    A = 0;
    B = 0;
    #2  A = 1; // After 2 time units, A is changed to 1.
    #1
    B = 3;       // After another time unit, B changes to 3.
    #2 $stop;   // After 2 more time units, the simulation
                //   stops.
    end

// The following displays the outputs.
init
    begin:  Display
    // The following is displayed once at time 0 of the
    // simulation.
    $display("A   B   C   clock time");
    // The following will display whenever one of the
    // variables changes values.
    $monitor("%d %d %d %d %d",A,B,C,clock,$time);
    // Note that $monitor can occur at most once in verilog code
    // while $display can occur many times.
    end

endmodule
```

**Figure 3.2.  Example of a test bench.**

# 4 Example

*Creating and testing a 2:1 multiplexer.* Figure 4.1a shows the veriwell window at start up. We want to create a module for the 2:1 multiplexer and store it in a file called "mux2to1.V" Notice that verilog files should have the ".V" suffix. To begin, go to the file menu and select New. This creates a text file Noname1.v. Now, type in the module. A screen shot is shown in Figure 4.1b. Save it with the file name "mux2to1.V". Create a testbench file and call it testbench_mux.V. Type in a testbench such as in Figure 4.1.c.

Now that we have both the module and testbench, we can create a project. Go to the Project menu and select New. Next, select Project menu again and "add "mux2to1.V" and "testbench_mux.V". After adding the files, the window should look Figure 4.1d. Now you can simulate by going to Project menu again and selecting Run. There was a compile error in "mux2to1.V" (colon following "begin" on line 15 should be deleted). After fixing the error, the project was run again. It compiled successfully with two "warnings". This is shown in Figure 4.1e. Then you can run the simulation by clicking the arrow button shown in Figure 4.1e. Figure 4.1f shows the result of the simulation.
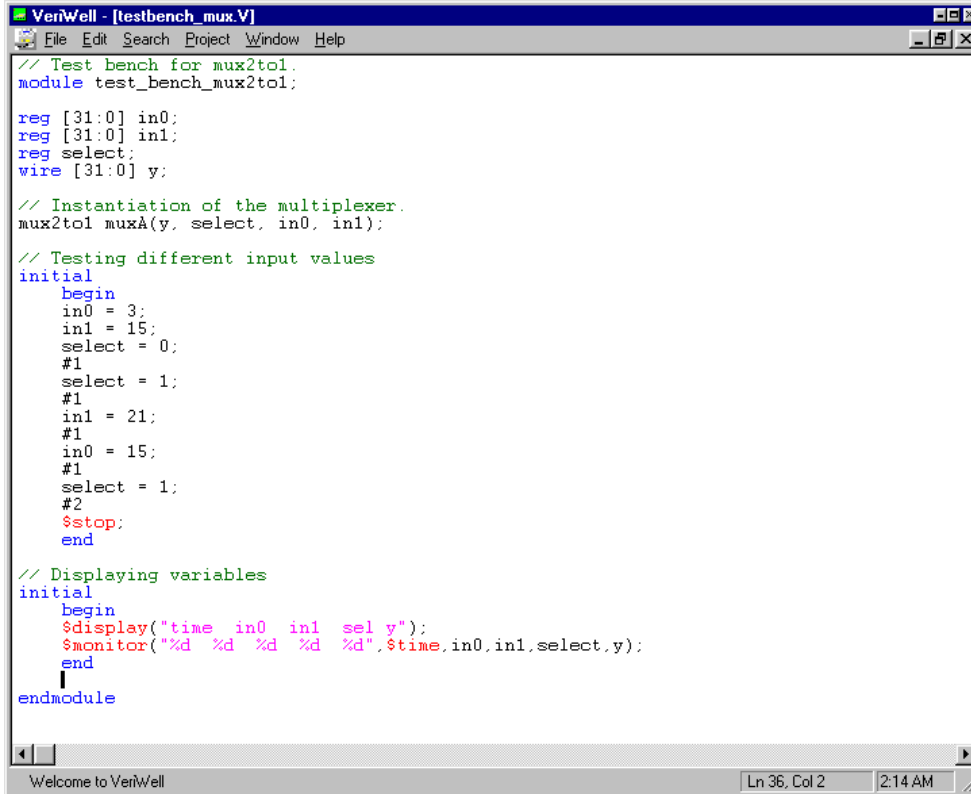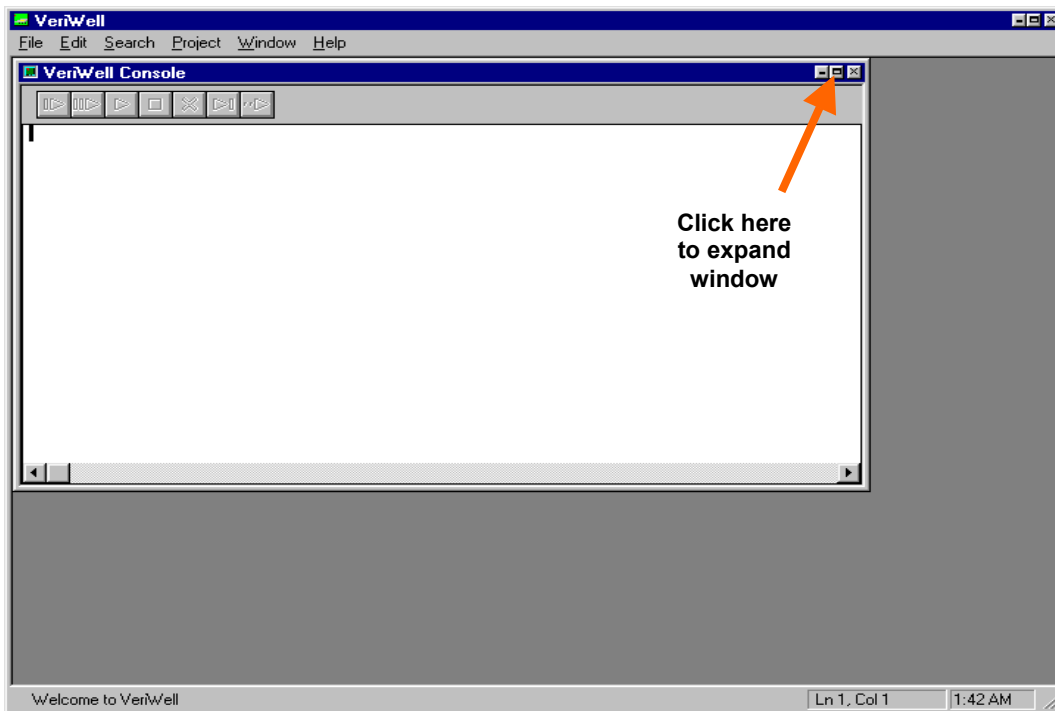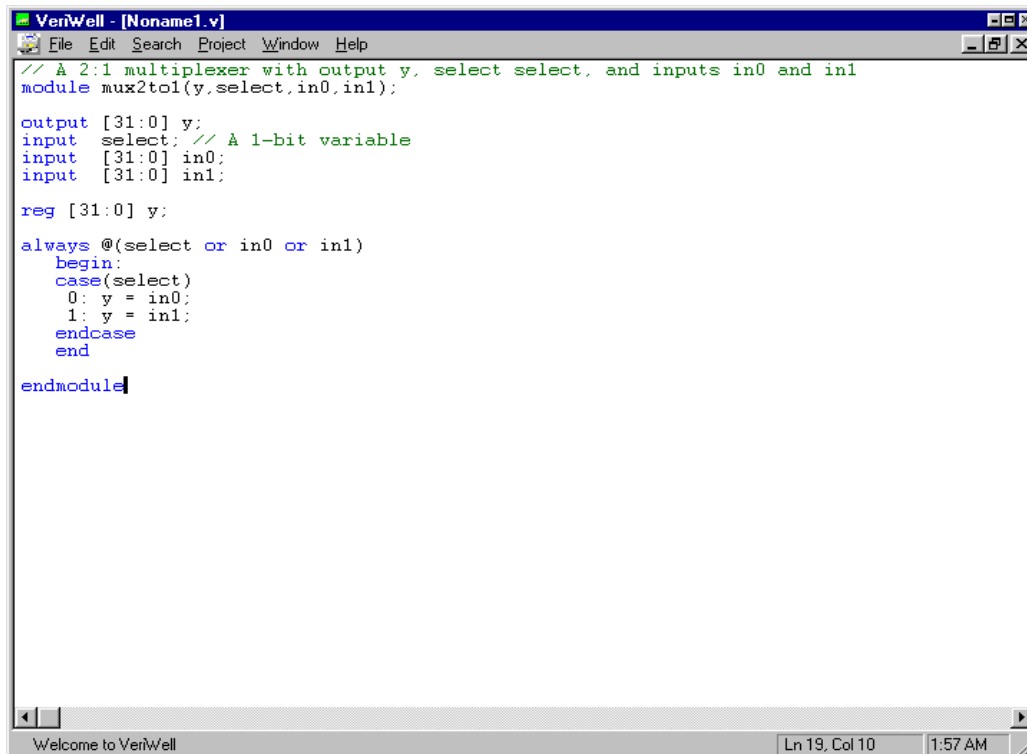
**VeriWell**

File  Edit  Search  Project  Window  Help

**VeriWell Console**

**Click here
to expand
window**

Welcome to VeriWell   Ln 1, Col 1   1:42 AM

**Figure 4.1a.  Veriwell console.**

**VeriWell - [Noname1.v]**

File  Edit  Search  Project  Window  Help

```
// A 2:1 multiplexer with output y, select select, and inputs in0 and in1
module mux2to1(y,select,in0,in1);

output [31:0] y;
input  select; // A 1-bit variable
input  [31:0] in0;
input  [31:0] in1;

reg [31:0] y;

always @(select or in0 or in1)
   begin
   case(select)
    0: y = in0;
    1: y = in1;
   endcase
   end

endmodule
```

Welcome to VeriWell   Ln 19, Col 10   1:57 AM

**Figure 4.1b.  2:1 multiplexer module, "mux2to1.V"**
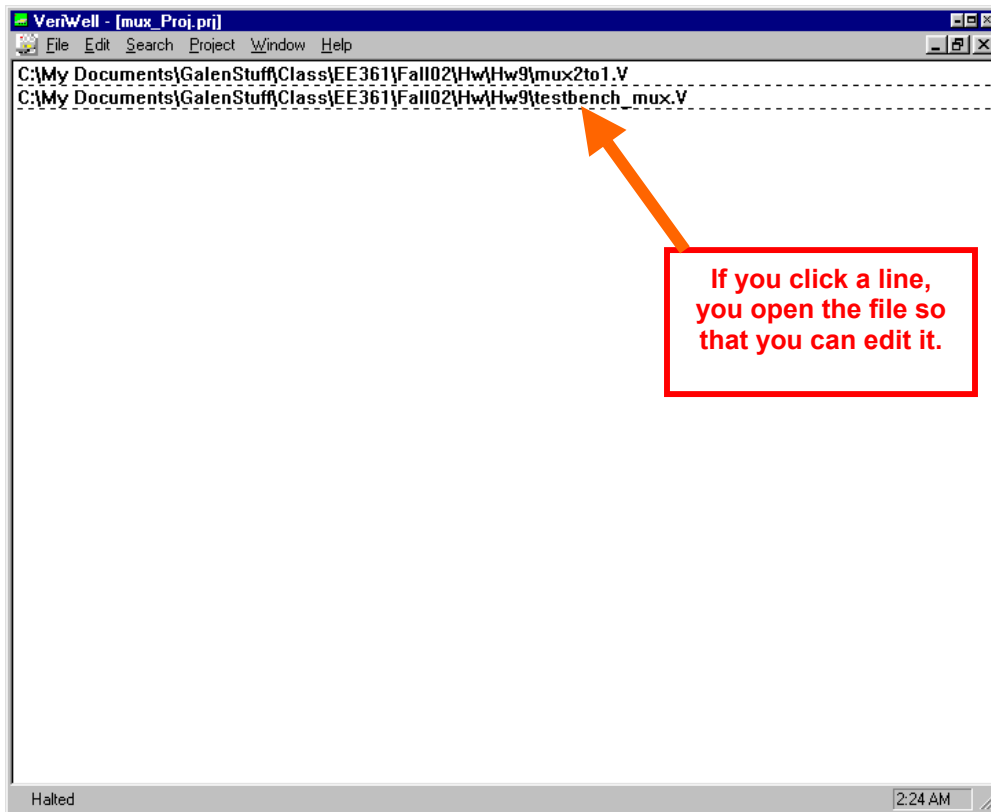
13

**Figure 4.1c. "testbench_mux.V"**



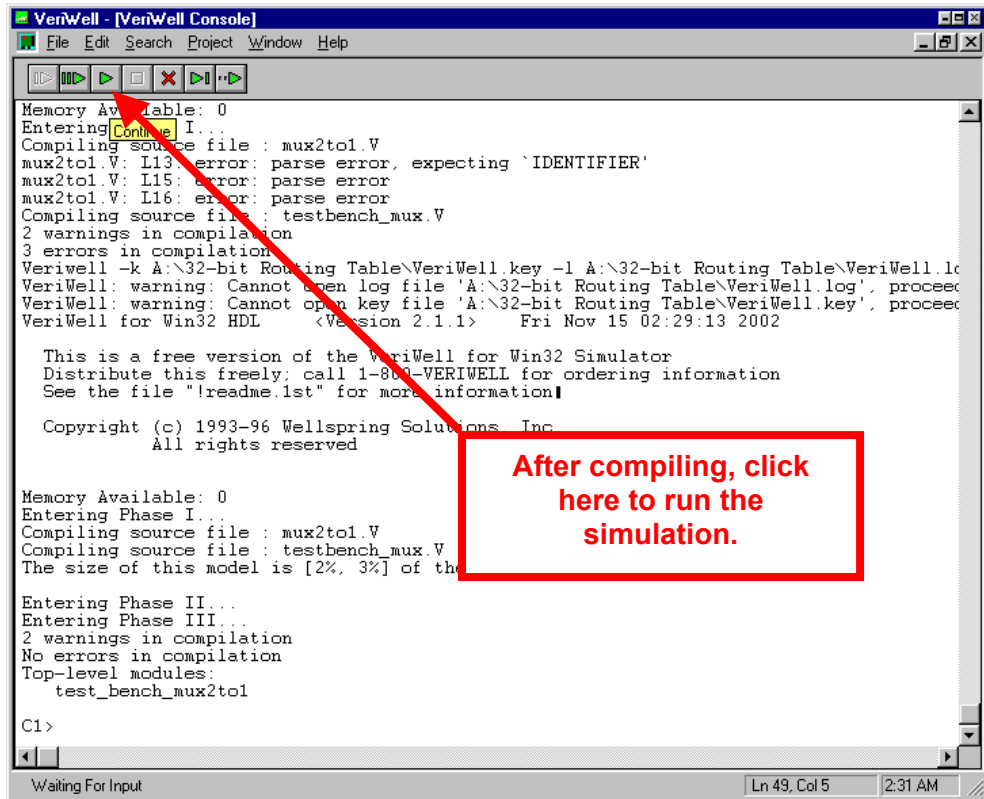**Figure 4.1d. Adding "mux2to1.V" and "testbench_mux.V" to a project.**

**Figure 4.1e. Running the project, which leads to "compiling" the project.**
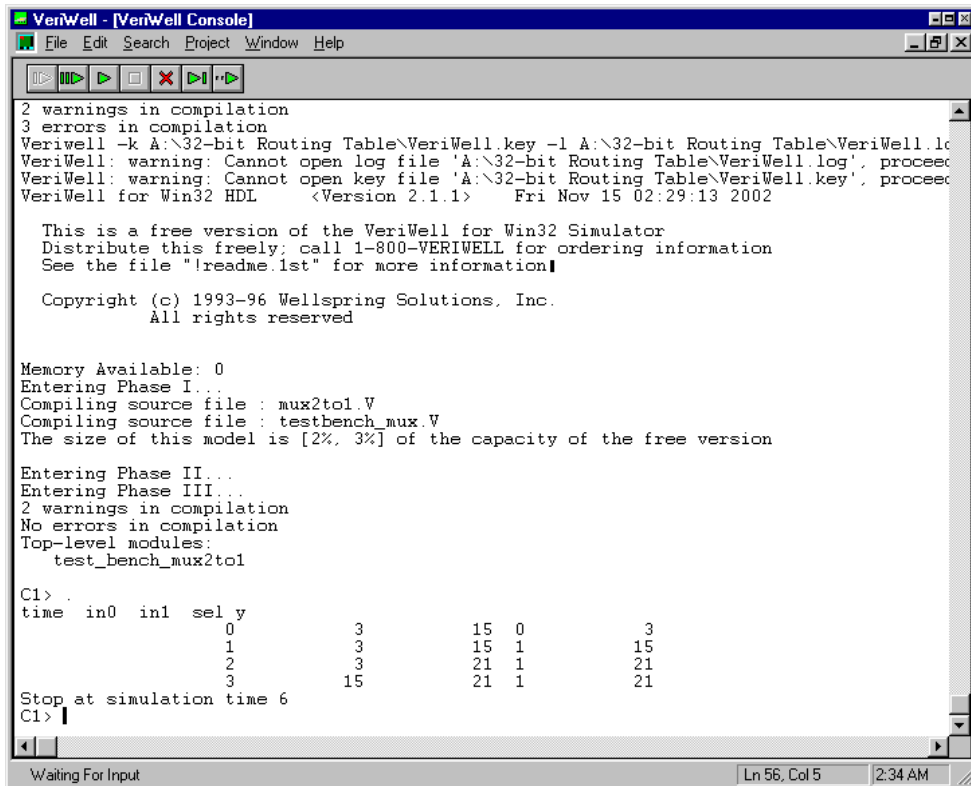


**Figure 4.1f. Running the simulation.**

Notice that in Figure 4.1f, the output was not spaced nicely.  The output should have looked like this

| time | in0 | in1 | sel | y |
|------|-----|-----|-----|-----|
| 0 | 3 | 15 | 0 | 3 |
| 1 | 3 | 15 | 1 | 15 |
| 2 | 3 | 21 | 1 | 21 |
| 3 | 15 | 21 | 1 | 21 |

You can check that the output y is equal to the input (in0 or in1) depending on the value of sel.  Thus, the multiplexer works for this test bench.