Procedural Modeling

Abstract

This lab assignment will walk you through a pair of procedural modeling tasks. The first is a combinational design that asserts an alarm signal under certain conditions. The second is simple accumulator. The third part of the lab is simply instructional on how to use the ModelSim to view waveforms of completed simulations.

Procedural Modeling

To date we've implemented designs using structural and dataflow constructs. These work very well and provide a clear path between the source itself and the logic that is created. Remember, we're building hardware so you should always be able to draw (or visualize) the blocks and gates that would be created from your source. These implementation styles, however, suffer at least two drawbacks.

1. Building complex decision trees is difficult because there is no if statement that can be used.
2. There is no mechanism to implement clocked devices (think flip flops), and

Procedural modeling fixes both of the above. First, if and case constructs are available for use inside of procedural blocks. Second, procedural modeling allows us to create blocks of source that are evaluated when some event occurs. Since it's an event driven construct, the rising edge of a clock can be used to indicate that the block should be evaluated and its LHS variables updated.

The always @

Procedural blocks generally start with the always @ construct. I say generally because there are procedural blocks that don't, e.g. initial. However, procedural blocks that start with anything other than always @ are generally not synthesizeable and shouldn't be used to build logic; they are suitable for test benches.

```
always @ ( <sensitivity list> ) begin
  ...
end
```

The sensitivity list is a list of events (or signals) that trigger the evaluation of the procedural block. When an event is in the list, e.g. posedge clock, the evaluation is trigger by that specific event, the occurrence of the rising edge of a clock in this case. When the sensitivity list contains a variable name by itself then any event changing the value of that variable will trigger the evaluation. E.g. if the list contained the variable A then it would be evaluated when A changed from 1 to 0, 0 to 1, x to 1, 1 to z, etc.

Note, the event triggering the evaluation of the block implies that the event has occurred in the past so that the signal value used in the procedural block for evaluation is post-change. E.g. if A is in the list and changes from 0 to 1 then the variable A will have the value 1 during the evaluation.

The if Clause

The if clause works very much, logically, like the if clause that you see in other languages. However, it's important to remember that we are building hardware so an if clause to use turns into a mux in real hardware. So, the following procedural block would become a mux using **sel** as the select line such that when **sel** is high the output, **out**, takes on the value of **A**. When **sel** is low the output, **out**, takes on the value of **B**.

```
always @( sel, A, B ) begin
  if( sel ) begin
    Out <= A;
  end
  else begin
    Out <= B;
  end
end
```

A couple of things to note in the above.

1. I've used begin/end to clearly define blocks of source. If there is more than one expression to be evaluated then begin/end blocks are required to collect those expressions. The begin/end block delimiters are not required if only a single expression is present.
2. The sensitivity list includes everything that is on the RHS in any expression in the procedural block. This includes the if clause itself. Since you're evaluating the logical value of **sel** then it may be thought of as being on the RHS of an expression.

You may remove the specific signals in the sensitivity list as an option by simply including an asterisk to indicate that if any signal on the RHS in this procedural block changes then consider it in the sensitivity list and evaluate the block. With the above simplifications, this same procedural block may be written as follows.

```
always @*
  if( sel )
    out <= A;
  else
    out <= B;
```

The entire if clause is considered a single expression. Each of it's clauses updating the value of **out** are also single expressions. So, not begin/end is needed.

Assignment Part 1: Alarm System

This first part is a combinational module that will assert a home alarm signal if certain conditions are met. Design and implement a system that meets the following requirements.

1. The system as 4 inputs and a 2 outputs
   a. alarm_set, input, 1-bit: Indicates that the alarm has been enabled
   b. alarm_stay, input, 1-bit: Indicates that the doors should be secured
   c. doors, input, 2-bits: Represents the state of 2 doors
   d. windows, input 3-bits: Represents the state of 3 windows
   e. secure, output, 1-bit: Represents that the home is secure
   f. alarm, output, 1-bit: Indicates that the alarm is blaring
2. Each input is represented by a numeric 1 if asserted and a numeric 0 if not. This matches Verilog's logical representation of numeric values.

3. Multi-bit inputs are not encoded. Each bit represents a different entity. E.g. doors[0] represents the state of one door while doors[1] represents the state of another.
4. If the alarm is not set (alarm_set == 0) then the system is not secure and the alarm should not be blaring.
5. If the alarm is set then the state of secure and alarm is dependent on whether stay is set and the state of the doors and windows.
    a. While the alarm is set and nothing is causing the alarm to sound then the value of secure should be 1; this indicates that the alarm is set and nothing is amiss. If, however, something is causing the alarm to sound (see the next two requirements), then the value of secure should be 0.
    b. If stay is enabled that means the owner would like the alarm to sound if any of the windows or doors are opened (represented by a value of 1).
    c. However, if stay is not enabled then the owner would like to be able to move in and out of the house through the doors but would like the alarm to sound if any of the windows are opened.

We could certainly build this system by working through the truth table.

alarm_set	alarm_stay	doors[1]	doors[0]	windows[2]	windows[1]	windows[0]	secure	alarm
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0
0	0	0	0	0	1	0	0	0
1	1	1	1	1	1	1	0	1

However, it's pretty clear that this could get out of hand quickly. There are 7 bits of input meaning that this table would be 2^7, or 128, lines long. A better solution is to create a procedural block that leverages a nested set of if clauses. Yes, Verilog supports nested if clauses.

Turn In

The module has been started for you as lab_006_alarm.v. Complete the module and check your implementation with the provided lab_006_alarm_tb.v testbench. You may wish to add cases to the testbench to ensure that you've covered the cases of interest.

Commit your final lab_006_alarm.v file to Git and push the contents to GitHub for grading.

Sequential Logic

In the first part of the assignment we saw that we can leverage if clauses inside of procedural blocks. Procedural blocks also allow us to build logic that responds to clock edge events. These events use the posedge and negedge keywords to identify the type of event in the sensitivity list. The most natural use of these constructs is to create a clocked device that responds to the rising edge of a clock. For example.

```
always @( posedge clock )
  Q <= D;
```

This is the simplest form of D flip flop (DFF). It almost reads as a sentence. “Always at” the positive edge of the signal “clock” the value of “Q” should get (<=) the value of “D”. There are no resets or other controls in this logic so the value in simulation of the output Q would be represented as unknown until its first capture of D. We can fix that by including an if clause with a reset.

```
always @( posedge clock )
  if( !rst_n )
    Q <= 0;
  else
    Q <= D;
```

This now has an active low synchronous reset. Notice, however, that the signal, rst_n, is not in the sensitivity list. That’s what we want. The block shouldn’t evaluate just because rst_n changes value; we only want it to evaluate as a result of a rising clock edge. That’s what makes this a synchronous reset. The active low part is because we’re checking against the logical not (!) of the signal in the if clause.

Assignment Part 2: Accumulator

The second part of the lab is to design and implement a simple accumulator. The system has the following requirements.

1. The system as 2 global inputs, 1 data input, and 1 data output
    a. rst_n, input, active low synchronous reset
    b. clock, input, free-running clock
    c. d_in, input, 4-bits, data to accumulate
    d. d_out, output, 8-bits, accumulated value
2. Upon an active low synchronous reset, **rst_n**, the 8-bit output, **d_out**, shall be set to zero.
3. When not in reset and at the rising edge of the signal **clock**, if the value of **d_in** is greater than 6 then the accumulator output, **d_out**, shall be incremented by the value of **d_in**. Otherwise the value of **d_out** shall remain the same.
4. There is nothing that will stop the accumulation. If the accumulation reaches the maximal value that may be represented in 8-bits then the value of **d_out** just wraps around.

A couple of notes.

1. Verilog supports a plus operator, +, that will add two values together. While we don’t know the specific type of adder that would be implemented, it’s sufficient to use that operator for this assignment. E.g. d_out <= d_out + d_in.
2. Verilog supports numeric comparison operators. These produce logical results indicating whether the comparison result is true or false. Operators include greater-than (>), less-than (<), equal (==), etc. You’re free to use these operators for this assignment.

Turn In

The module has been started for you as lab_006_acc.v. Complete the module and check your implementation with the provided lab_006_acc_tb.v testbench and its associated lab_006_acc.dat file. You may wish to add cases to the testbench to ensure that you’ve covered the cases of interest.

Commit your final lab_006_acc.v file to Git and push the contents to GitHub for grading.

Viewing Waveforms with ModelSim

I will leave playing around with the tool up to you. This includes zooming, reorganizing signal layout, changing signal radix, etc. The following should get you up and running with signals in the waveform window.

First, ensure that you are using some graphical environment. I highly recommend running Fast X3 and connecting to grendel.ece.ncsu.edu for your work. Details may be found in previous labs and videos on how to set that up.

Note, until viewing your results in a waveform using the ModelSim GUI, the commands are all run from the command line of a terminal.

Set Up Your Environment

Review previous labs for more detail. Add the ModelSim module, copy a modelsim.ini file, and set your MODELSIM environment variable. Then, create a work library and map it.

```
% add modelsim10.7c
% cp path/to/modelsim.ini .
% export MODELSIM=modelsim.ini
% vlib work
% vmap work work
```

Compile Your Source

There is nothing special about compiling your source. Review previous labs for more detail.

```
% vlog *.v
```

Simulate Your Design

For this example, I'll use part 1 of the lab assignment. The testbench that is provided is called lab_006_alarm_tb and is found in the file of the same name. For the simulation, you reference the testbench name, not the file name.

You'll load the simulation with the vsim command and some options. Then, once loaded, you'll tell the simulator to log all of the signals in an output file and then run the simulation and quit. When complete, you'll see a new file called vsim.wlf.

```
% vsim –c –voptargs=”+acc” lab_006_alarm_tb

VSIM> log –r *
VSIM> run -all
VSIM> quit
```

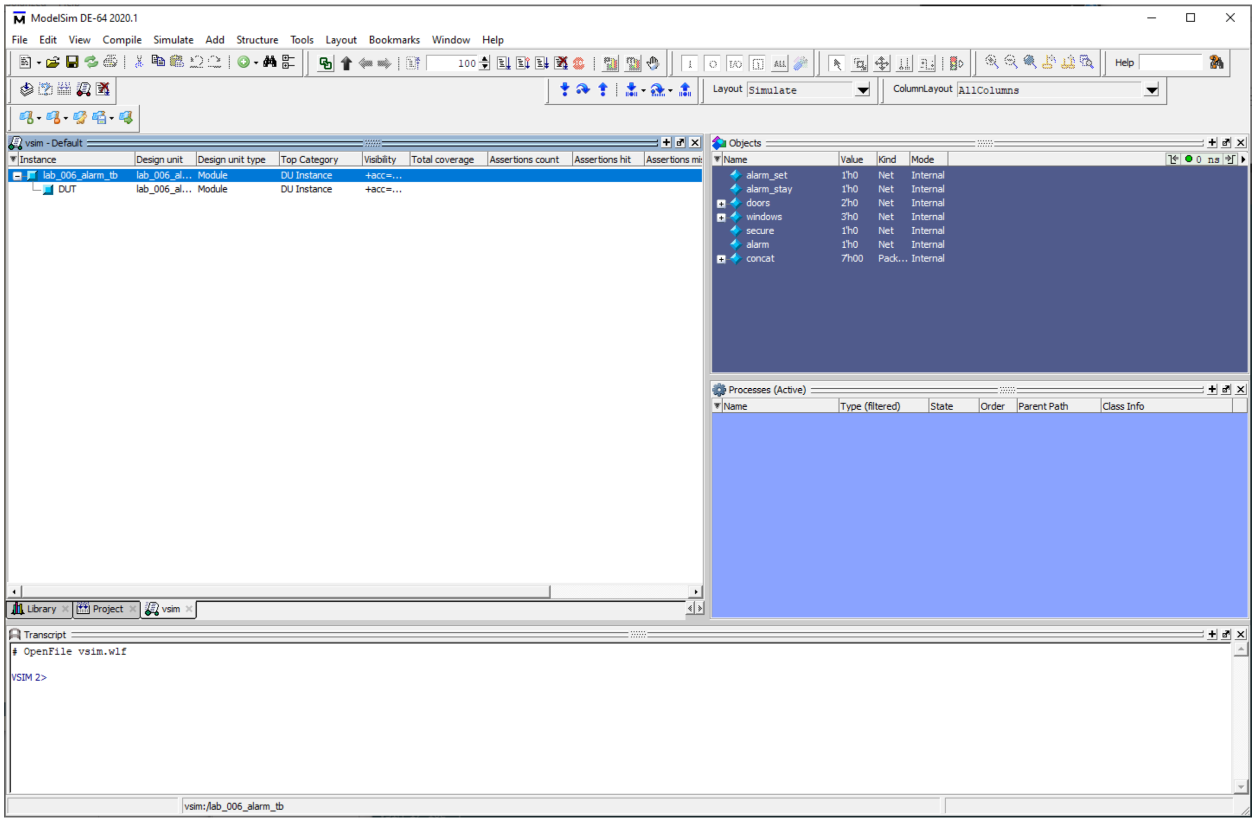
NOTE: You may not be able to just copy and paste from this manual. Hyphens and quotes are generally abused in Word so, while the character may look correct on the command line, the tools won't interpret them correctly. Please type these commands in to your terminal.

Load the Simulation

Now that the vsim.wlf file has been created you may load it directly into the GUI.

```
% vsim vsim.wlf &
```

Once the GUI settles (there will be a lot of window reorganization while the tool loads) you’ll see something like the following.



Instance	Design unit	Design unit type	Top Category	Visibility	Total coverage	Assertions count	Assertions hit	Assertions mi
lab_006_alarm_tb	lab_006_al…	Module	DU Instance	+acc=…				
DUT	lab_006_al…	Module	DU Instance	+acc=…				

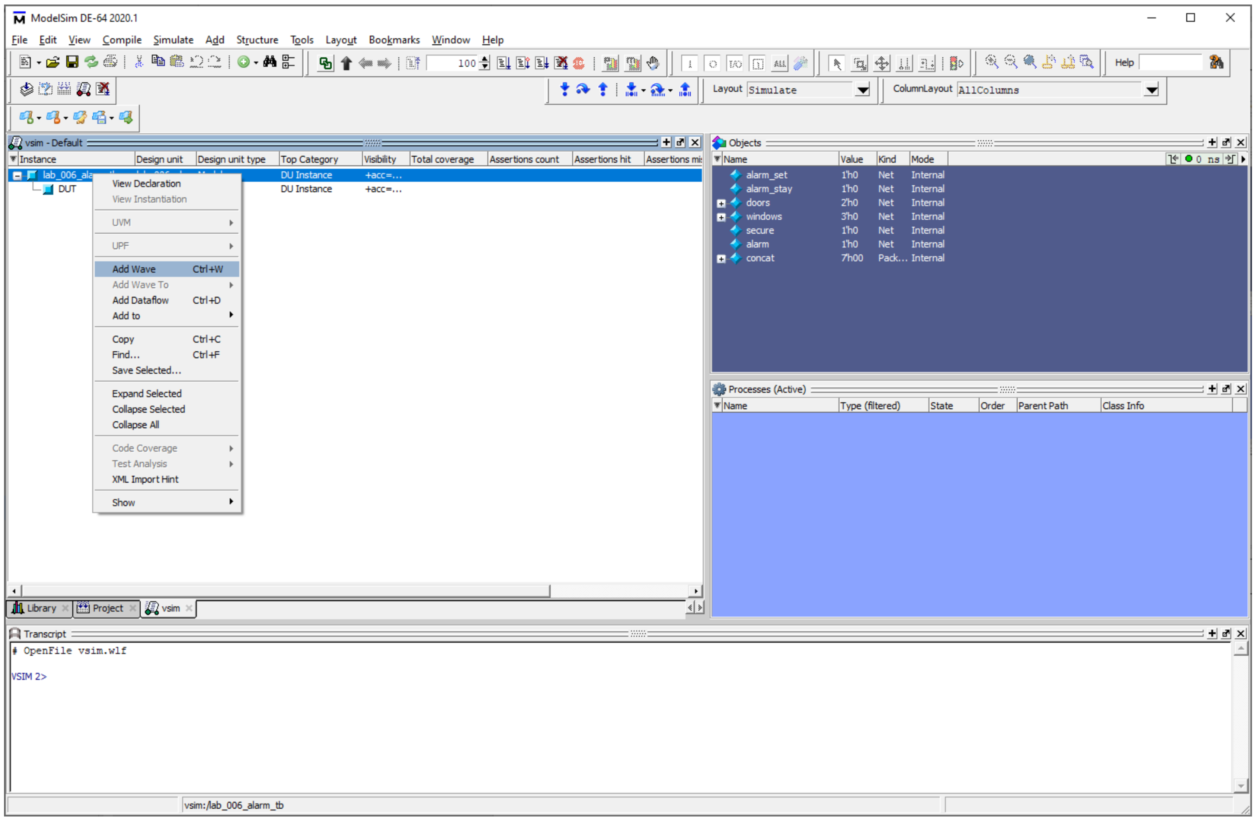
Name	Value	Kind	Mode
alarm_set	1'h0	Net	Internal
alarm_stay	1'h0	Net	Internal
doors	2'h0	Net	Internal
windows	3'h0	Net	Internal
secure	1'h0	Net	Internal
alarm	1'h0	Net	Internal
concat	7'h00	Pack…	Internal

Name	Type (filtered)	State	Order	Parent Path	Class Info

```
# OpenFile vsim.wlf

VSIM 2>
```

vsim:/lab_006_alarm_tb

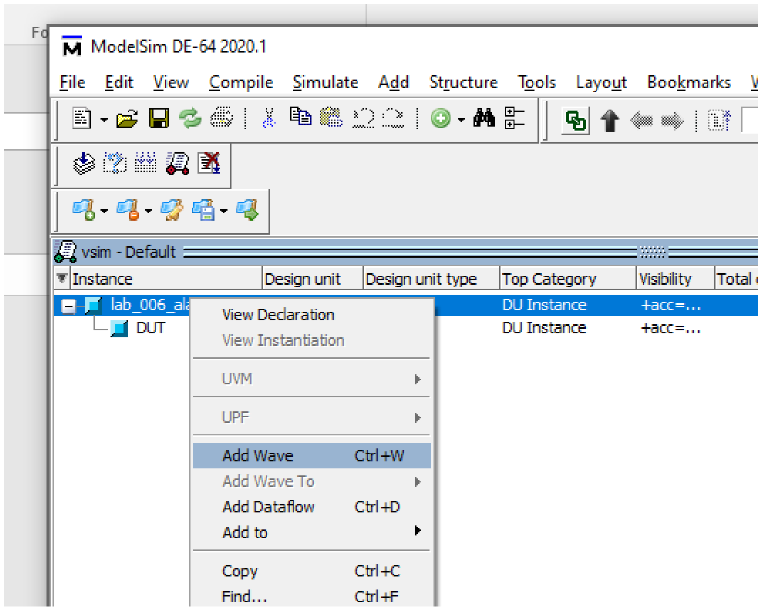
To add all of the signals at the top of the simulation you right-click on the lab_006_alarm_tb line (that’s the module that you’re selecting) and select Add Wave.



Name	Value	Kind	Mode
alarm_set	1'h0	Net	Internal
alarm_stay	1'h0	Net	Internal
doors	2'h0	Net	Internal
windows	3'h0	Net	Internal
secure	1'h0	Net	Internal
alarm	1'h0	Net	Internal
concat	7'h00	Pack...	Internal

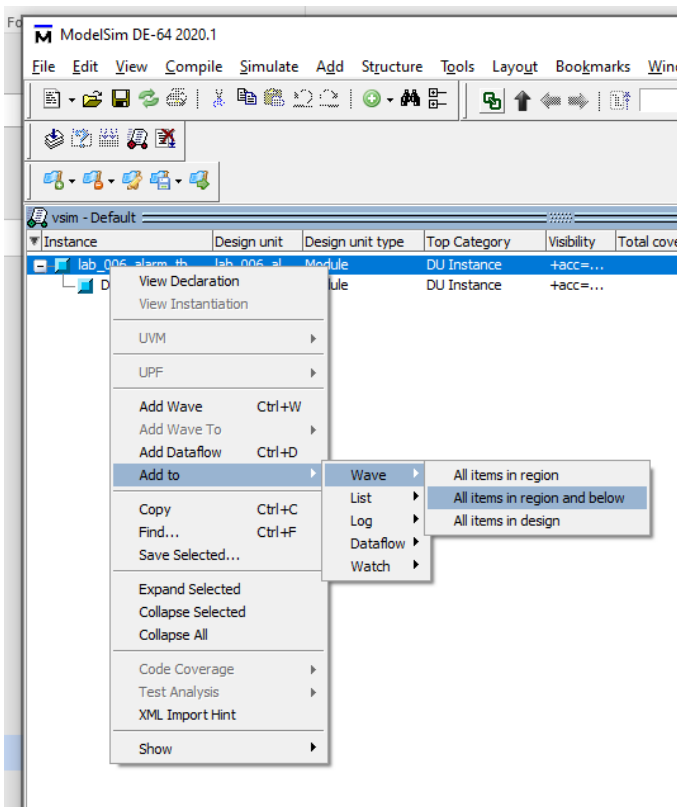
```
# OpenFile vsim.wlf

VSIM 2>
```

A little more zoomed in.

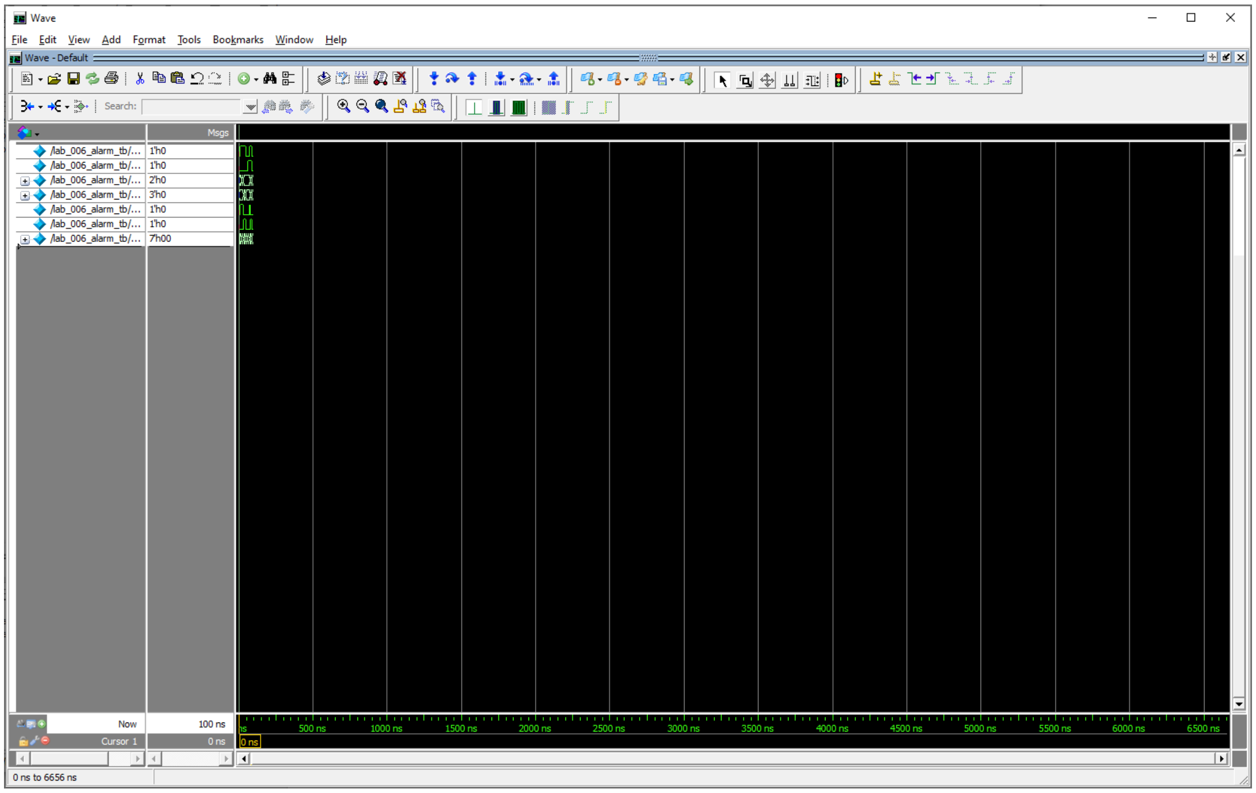


This will add all of the signals at the top level to the waveform. You may, of course, select blocks further down the hierarchy to add. E.g. right-click on DUT to add the signals from the DUT to the waveform; this would include signals internal to the DUT that aren't available at the top of the hierarchy.

You may also add all of the signals throughout the entire hierarchy by selecting Add to -> Wave -> All items in region and below.

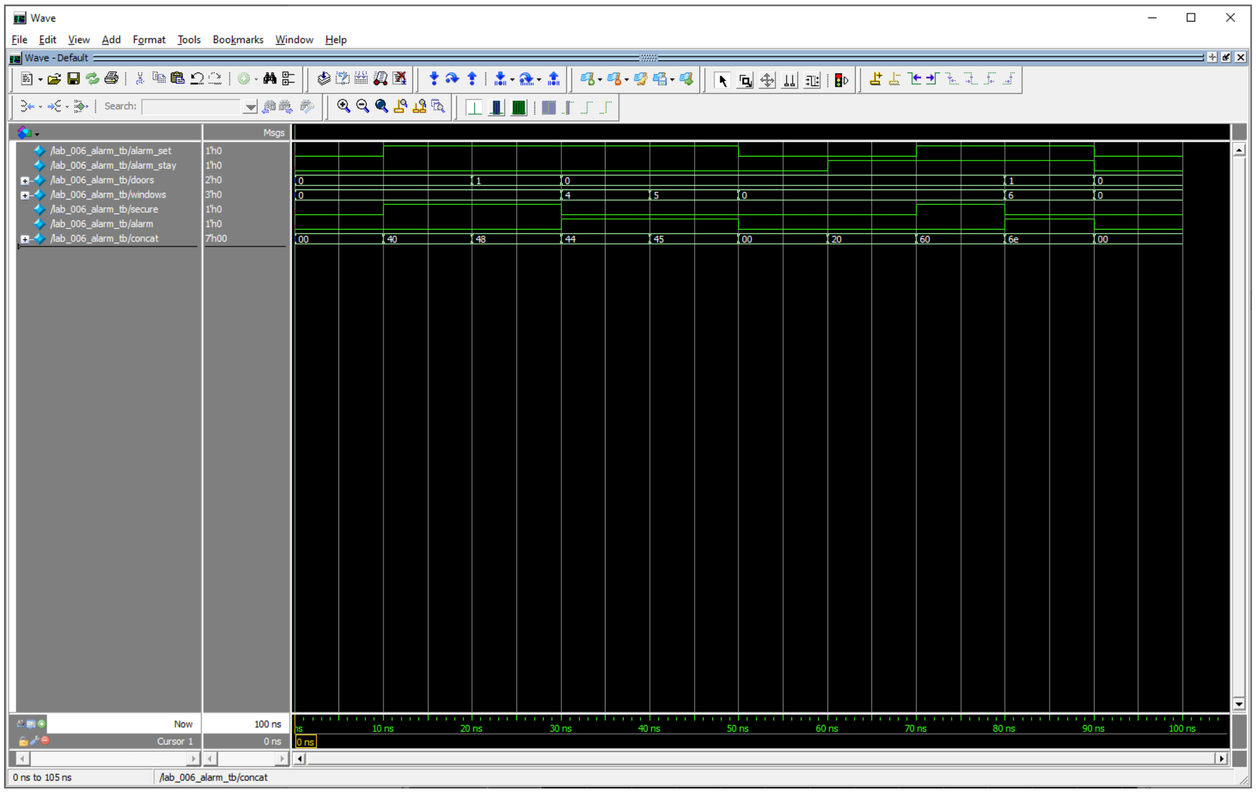


The wave form window is updated dynamically as you add signals. For the top of the simulation, the wave form will look like the following.


Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Search:
Msgs
Now
Cursor 1
ns 500 ns 1000 ns 1500 ns 2000 ns 2500 ns 3000 ns 3500 ns 4000 ns 4500 ns 5000 ns 5500 ns 6000 ns 6500 ns
0 ns

```
/lab_006_alarm_tb/...   1'h0
/lab_006_alarm_tb/...   1'h0
/lab_006_alarm_tb/...   2'h0
/lab_006_alarm_tb/...   3'h0
/lab_006_alarm_tb/...   1'h0
/lab_006_alarm_tb/...   1'h0
/lab_006_alarm_tb/...   7'h00
Now      100 ns
Cursor 1   0 ns
0 ns to 6656 ns
```

This is zoomed very far out. To fill in the window with the simulation clock on the magnifying glass with the plus sign (to zoom in) or the magnifying glass with that's filled in (zoom full which will fill the window with the entire simulation). The zoomed out simulation looks like the following


Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Search:
Msgs
0 1 0
0 4 5 0 6 0
00 40 48 44 45 00 20 60 6e 00
ns 10 ns 20 ns 30 ns 40 ns 50 ns 60 ns 70 ns 80 ns 90 ns 100 ns
0 ns

```
/lab_006_alarm_tb/alarm_set     1'h0
/lab_006_alarm_tb/alarm_stay    1'h0
/lab_006_alarm_tb/doors         2'h0
/lab_006_alarm_tb/windows       3'h0
/lab_006_alarm_tb/secure        1'h0
/lab_006_alarm_tb/alarm         1'h0
/lab_006_alarm_tb/concat        7'h00

Now        100 ns
Cursor 1     0 ns

0 ns to 105 ns    /lab_006_alarm_tb/concat
```