Frequently Asked Questions

MODELSIM

Q. In ModelSim, when I click the message saying *x* errors in the transcript window, the window that pops up does not show me any errors?

This is because your file name (complete path) has spaces in it. While using ModelSim, please make sure that the file name doesn't have any white spaces. In other words, do not have your programs saved on a path like "xyz\Documents and Settings\user1\lab 1\file.v". Please make a folder on the Z: drive of the computer you work on and keep your project/source files there.

Q. When I click on ModelSim, it gives me an error saying failed to checkout license.

In case invoking ModelSim shows a licensing error on the lab computers, please run the Licensing Wizard first (Start->Programs->ModelSim->Licensing Wizard), and then launch ModelSim.

Q. How do I create and run a do-file?

The ModelSim tutorial talks about creating a file of commands in the end (called a do-file), but does not explain how to do it clearly. Here is how you can do this: Basically, the commands like force, run, etc that you provide on the transcript window can be saved in a file and that file is called a "do-file". The benefit of having a do-file is to be able to re-run all the commands by just a single click, rather than typing them again and again. For example, if you have your do-file ready during the checkout, you can just execute it instead of typing the individual commands all over again.

There are two ways of creating a do-file.

1. You can manually write those commands in a file using a text editor and save it with a ".do" extension.
2. You can type the commands on the transcript window, and then have ModelSim create the file for you. For this, type the commands in the transcript window (keep the transcript window selected). Then go to "File->Save As", and then provide the name of the file with a .do extension.

To execute the commands in the do-file, make sure the transcript window is active. Then, go to "File -> Load" and then provide your do-file to the tool.

Q. Can I view variables on waveforms?

Viewing variables on the waves is just like viewing signals (of course, you should be simulating your design to view the waves). The ‘Objects’ window shows you the signals in a design. Similarly, the “Locals” window shows the variables in the selected module/always block. For seeing variables, go to the ‘View’ menu and click on ‘Locals’. A ‘Locals’ window will appear.

When you are simulating, you can see that a “Sim” pane appears near to your “Project” and “Library” panes. Click on the “Sim” pane and it will show you the design hierarchy. You can click on any module or a line number of an always statement whose variables you want to see. Now, in the “Locals” window, you can click on variables and then drag to the waveform window.

Q. Some signals in my design are not visible in the "Objects" window, and so I can't view their waveforms.
This is because ModelSim performs a series of optimizations on your design and can get rid of some signals. The 'optimized out' signals cannot be seen in the "Objects" window. You can disable optimization in two ways:

1. While starting simulation, instead of just double clicking on the module name in the "Library" window, right click and say "Simulate without Optimization".
2. On the transcript window, append " –novopt" to the "vsim" command

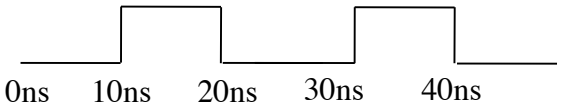
Q. Can I view waveforms of signals inside the design hierarchy (modules other than the top module)
When you are simulating a design, you can see that a "Sim" pane appears near to your "Project" and "Library" panes. Click on the "Sim" pane and it will show you the design hierarchy. You can click on any module in the design. When you click on a module, the "Objects" window shows the signals in that module. Now, in the "Objects" window, you can click on signals and then drag to the waveform window (or you can right click a signal and say Add->To Wave->Selected Signals).

Q. How can I change the way signals are shown on the waveforms (To change viewing 000101 to 5)
Right on the signal in the "Waves" window, got to "Radix" and select the one you want.

Q. How can I create a clock using the force statements in ModelSim during simulation?
To create/generate a clock, you can use the following command:

force clk 0 0 ns, 1 10 ns –repeat 20 ns

This command creates a clock of period 20 ns with 50% duty cycle as shown below:



You can change the period and duty cycle as you want by modifying the force statement appropriately.

Q. I used the 'force' command to force a signal. Now I want the design to drive it. But it is just stuck to that value.
A force statement forces the specified value onto the specified signal at the specified time and then that value remains on that signal for the entire simulation. It can only be changed by another force statement.
Adding "-deposit" option to the "force" command puts the specified value on the specified signal at the specified time, but lets it change anytime after that (if another driver wants to modify/override it; for example an assignment statement in the design).
For example, let's assume that you have an output that you want to initialize to 0 at the beginning of the simulation. Assuming also that you have not initialized this output to 0 in your code, you may simply type: force Z 0 0 ns. You will note after running the simulation that Z never changes. To overcome this problem, change the above statement to: force –deposit Z 0 0 ns. The deposit will simply deposit the value of 0 to Z at 0ns instead of freezing it at 0.
The "-cancel" option cancels the force on a signal at a specified time.
You can look into more options of the force statement by going to "Help -> PDF Documentation -> Reference Manual" in ModelSim.

Vivado

Q. What is a XDC File?

The xdc file is the file which tells Vivado to map the inputs and outputs of your design to specific pins on the FPGA. The file also has other things like clock constraints etc, but we are not going to be concerned about them in this lab.

VERILOG

Q. Can I model combinational logic using always statements? How?

Ideally, concurrent statements are used to model combinational logic and always statements are used to model sequential logic (flip flops and latches). However, always statements are not restricted to that. You can model combinational logic using them. But it is important to note that when using an always statement to make combinational logic, the sensitivity list of the always statement should contain all the signals which are being 'read' in that always block. In other words, to synthesize combinational logic using an always block, all inputs must appear in the sensitivity list.

For example, if you were to model a mux, you would say:

```
always @(a, b, sel)
begin
    if (sel == 1) z <= a;
    else z <= b;
end
```

Using a always statement to model combinational logic is handy because statements like if, case, etc (which are very useful and intuitive) can only be written inside always statements.

Q. What care should I take when using the always statement to write sequential logic?

When using an always statement to model sequential logic, the only thing in the sensitivity list of the *always* statement should be the clock (and a reset signal, if it is an asynchronous reset). And there should be a' 'posedge' or 'negedge' in the sensitivity list before the clk (and asynchronous resets). Make sure to test the reset case FIRST with a *if* statement, and then do the normal operation stuff in the *else* clause. Order matters! Also, make sure that you have at most one asynchronous signal in the sensitivity list in a sequential *always* block.

Flip-flop without a reset

```
always @(posedge clk) //positive edge triggered
begin
      q <= d;
end
```

Flip-flop with an async reset

```
always @(posedge clk, negedge rst) //positive edge triggered with reset
begin
if(rst == 0)     //async active low reset
    begin
    q <= 0;      //must put initialization/reset code in the if clause!
    end
else
    begin
    q <= d;      //must put normal operation stuff in the else clause!
    end
end
```

Flip-flop with a sync reset

```
always @(posedge clk) //positive edge triggered
begin
if(rst == 0)     //sync active low reset
    begin
    q <= 0;
    end
else
```

```
    begin
    q <= d;
    end
end
```

On the other hand, a latch is a level triggered element. A resettable latch can be modeled as:

```
always @(en, rst, d)
begin
if(rst == 0)
    begin
    q <= 0;
    end
else if(en == 1)
    begin
    q <= d;
    end

end
```

Q. Why can I not instantiate a module inside an 'if' statement (or an always block, for that matter)?

It is important to realize that a module is not like 'calling' a function in C. It is an instantiation of that module. Therefore, it cannot be conditional. If you have to instantiate a block in your design, it will be always present there.

Let us take an example. Say you have an adder and a subtractor. You design's specifications say that when the input MODE is '1', the design should work as an adder, while when the MODE is '0', the design should work as an subtractor. Now, this does not mean that you can have something like this:

```
always(...)
begin
    if(MODE == 1)
        adder adder_inst(A,B,Sum);
    else
        subtractor subt_inst(A,B,Diff);
end
```

Since we are modeling hardware, we cannot say that if MODE is 1, Adder is 'called' and when MODE is 0, subtractor is 'called'. This is a wrong way of thinking.

Instead you should think of this as: Adder and Subtractor are always present. The output of the design can be driven by either the Adder or the Subtractor depending on MODE. So you should have something like this:

```
adder adder_inst(A,B,Sum);
subtractor subt_inst(A,B,Diff);

always(...)
begin
    if(MODE == 1)
        output_ALU <= Sum;
    else
        output_ALU <= Diff;
end
```

GENERAL

Q. What tests should the 'do' file that I submit on canvas contain?
It is always better to submit a do-file which has sufficient number of input combinations (not just the ones given in the lab description).

Q. I am getting a multiple drivers error. What should I do?
A multiple driver error occurs because there is more than one thing driving a signal. This can happen if you are driving a signal from two sources, e.g. one always block and one concurrent statement, or two always blocks. Realistically, it is not possible to do so (without having contention, which we are staying away from). There is nothing you can do to get rid of this, other than changing your design.

Q. My design compiles successfully in ModelSim. When I simulate, I get weird errors (error loading design, etc) and I can't simulate.
The compilation process looks at individual modules in your design and checks for syntactical and semantic correctness. Simulation lets you apply inputs and observe outputs. Between compilation and simulation, is a step called elaboration (which is usually hidden from you, and happens when you start simulation in ModelSim). During this step the design hierarchy is generated. Connections between various modules, and search for entities referenced as components in a design, etc are done at this stage. If there is a problem at this stage (for example, there is a component declaration in your top module but the module for that component is missing), they are reported just before simulation. So, now you know where to look for when you get errors just when you start simulation.

Q. Will setup and hold time be met in my simulation? Or If I add some logic between two stages in my design, will the delay affect the output? Or should I force my input sometime before the clock edge to satisfy setup and hold time constraints?
Remember that the simulations that you are doing in the lab are all RTL simulations. They are zero-delay simulations (assuming you are not modeling delays using '#' statements). Therefore, there is no concept of delays of gates or setup-hold time of flip-flops. If we were doing post-synthesis simulations, then we would be concerned with timing issues.

GOOD DESIGN PRACTISES

Q. Are there any general 'good' design practices that I should follow?

1. Writing Verilog feels like writing software. But it is a good idea to think 'hardware' while writing code!
2. Do not use '#' (delay) statements in your designs in the lab. Testbenches may use these. For example, to generate a clock signal in a testbench you can say "#10 clk = ~clk;"
3. A state machine can be designed using either a single always block (like Figure 2.54 in the text) or using two always blocks (like Figure 2.52 in the text). Both ways are correct. However, it is easier to design it using a single always block. Generally, the single always block partakes less debugging effort.
4. Stay away from 'variables' unless you are absolutely sure.
5. Concurrent statements are continuous drivers. Do not use them for initializations.
6. It is a good idea to have a reset signal in your design (even if not mentioned in the lab description). Use this signal to reset all the things you want to.
7. While simulating your design, it is always a good idea to stagger your inputs with respect to the active clock edge. For example, if your active clock edge is occurring at 10ns, apply your inputs sometime before 10ns, say at 8ns. This ensures that when your design was clocked, the input was successfully read. If your active edge occurs at 10ns and your input also changes at 10ns, then it becomes hard to see whether the input was successfully captured by the clock edge or not. Debugging becomes harder if you have your inputs like that.
8. Don't limit your testing to the input sequences mentioned with the problem statement. During the checkouts, the TAs will apply several input combinations to test your design. So, make sure to do a thorough testing of your design using sufficient number of inputs.
9. Generally, we tend to ignore warnings from the tools. But make sure you look at all the warnings after the synthesis process is completed. Sometimes there are problems in your design like missing connections, latches, etc. Such issues make the tool infer your design differently from what you want or expect it to be. These warnings might contain the reason why your design does not work on the board.

Q. My design works in simulation. But it does not work on the board. What should I do?

There is no one sentence answer to this question. You can try the following things to help you debug your problem:

1. Follow the good design principles discussed above.
2. Look for any warnings in the synthesis report.
3. Make sure there are no latches in the synthesized output.
4. Follow the synthesis-friendly code guidelines discussed in the next question

Q. My design works in simulation. But Vivado throws an error during synthesis, saying "Bad Synchronous Description". What am I doing wrong?

The one line answer to this question is that you are not writing synthesizable code. Here are a few tips:

1. posedge/negedge should only be used on clocks

2. An always statement used to model sequential logic should only have clock (and reset, if you need one) in the sensitivity list, and an always statement to model combinational logic should not have clock in the sensitivity list. (this is illustrated in detail below)

3. A signal cannot change on both negative and positive edges of clock (This is specific to the design you do in the lab because the FPGA hardware does not have dual edge triggered flops. This is true for most

industrial designs also. However, there may be some very high end designs which use dual edge triggered flops, in which case this constraint on your code gets removed.)
4. Make sure the tool is able to decipher the value of each signal under each condition.

Synthesis friendly 'always' statements

A. If you use always statement for a combinational logic, make sure the sensitivity list contains all inputs. And the clock should not be amongst those inputs! If you feel like you need the clock, it means you want to write sequential logic. Think again!

B. All always blocks other than the ones used for combinational logic will have a structure similar to this:

```
always @(posedge clk, negedge rst)
begin
if(rst == 0)     //async active low reset
    begin
    //initializations
    end
else
    begin   //positive edge triggered sequential logic
    //actual stuff
    end
end
```

C. So, any always block in your design should fall into either of the following categories:

```
always @(posedge clk, negedge rst) //model posedge triggered sequential logic with
reset
begin
if(rst == 0)     //async active low reset
    begin
    //initializations - this must go inside the if!
    end
else
    begin
    //actual stuff - this must go inside the else!
    end
end

always @(posedge clk) //model positive edge triggered sequential logic
begin
    //stuff
end

always @(a,b,c) //model combinational logic
begin
    //stuff
end
```

Q. VIVADO reports there are latches in my design. Where am I going wrong?
Latches are caused when you forget an 'else' block in an 'if' or 'case' statement in an always block intended to make combinational logic. Look at your design and find such cases.

Example:
The following always statement was intended to make do some selection. It was expected that a mux will be generated for both f and g.

```
always @(sel, a, b, c)
begin
    case (sel)
        3'b000 : f <= a; g <= c;
        3'b001 : f <= b; g <= d;
        3'b010 : f <= a; g <= c;
        3'b011 : f <= b; g <= d;
        3'b101 : f <= b; g <= d;
        3'b110 : f <= a; g <= b;
    endcase
end
```

But notice that the assignment to 'g' was missed in one case. And one case (100) was not mentioned. Therefore, latches were inferred for both 'g' and 'f'. Here is the correct way to write this:

```
always @(sel, a, b, c)
begin
    case (sel)
        3'b000 : f <= a; g <= c;
        3'b001 : f <= b; g <= d;
        3'b010 : f <= a; g <= c;
        3'b011 : f <= b; g <= d;
        3'b101 : f <= b; g <= d;
        3'b110 : f <= a; g <= b;
       default : f <= a; g <= b;
    endcase
end
```

Also, in an always block used to model combinational logic, if you forget to assign all signals under all conditions, you will end up with latches. So, to synthesize combinational logic using an always block, all signals must be assigned under all conditions.

Example:

```
always @(state, a, b, c, d, e)
begin
 case (state)
    0: if (a == 0) next_state <= 1; //IDLE STATE
    1:  //INITIAL STATE
    begin
        if (a == 1) next_state <= 2;
        else next_state <= 3;
    end
    2:  ...
```

Since 'next_state' is not assigned when a is '1', a latch is inferred. To avoid unwanted latches, a good way is to make sure you assign all signal under all possible conditions.

```
always @(state, a, b, c, d, e)
begin
 case (state)
    0: //IDLE STATE
    begin
        if (a == 0) next_state <= 1;
        else next_state <= 0;
    end
    1: //INITIAL STATE
    begin
        if (a == 1) next_state <= 2;
        else next_state <= 3;
    end
    2:  ...
```

But an easier (sometimes; depends on functionality) way can be to create a default assignment for all the variables in the always block.

```
always @(state, a, b, c, d, e)
begin
 next_state <= 0; //default assignment
 case state
    0: if (a == 0) next_state <= 1; //IDLE STATE
    1: //INITIAL STATE
    begin
        if (a == 1) next_state <= 2;
        else next_state <= 3;
    end
    2:  ...
```