Lab 1: Verilog review and Introduction to System Verilog

Contents

1 Introduction

In this lab, you will review basic Verilog concepts which you were taught in ECE241. You will also learn new concepts which you will use in ECE342

1. **System Verilog** : An advanced version of Verilog which you learnt in ECE241, with some key improvements.
2. **Creating generic hardware** : How to create multiple copies of the same hardware quickly and efficiently.
3. **Testbenches** : A more systematic and less error-prone way of testing your circuits, compared to the .do files you used in ECE241.

Before you start, please note that this lab document is **quite long** as it explains things in detail for your understanding. Please go through this document **patiently** and make sure you understand each step. You are required to know all the material covered in this lab for the rest of the labs in this course so spending the time on this lab will save you more time later on, by avoiding mistakes and issues. Handouts for future labs will be much shorter as they will simply tell you what you need to do, with the expectation that you already know how to use the tools from this lab. Be prepared to spend some time doing this lab as you can be sure that spending time now will likely save you 5× to 10× as much time in future labs, if you are not familiar with Verilog and/or how to use the tools effectively.

Also, as this is a long document, a Table of contents is provided at the top to make it easier for you to access the different sections quickly. You will want to refer to the Appendices throughout this course as they contain a lot of useful information. **Appendix C in particular lists many common issues you will encounter. Make sure to check there before posting a question on Piazza.**

Lastly, you should also **be prepared to answer questions from your TA about the concepts covered in this lab handout.** So it will be well worth your time to read it carefully to make sure you understand it fully.

2 Part I: 8-bit Ripple Carry Adder.

In this part, you must design an 8-bit ripple carry adder (RCA). You will also learn about some of the changes in System Verilog that can help you in ECE342.

2.1 New signal type: Logic

The first major change in System verilog is the introduction of a new signal type, namely `logic`. Recall that in ECE241, that you used two different types of signals, namely `wire` and `reg`. Signals that were set inside always blocks had to be declared as `reg`, while all other signals were declared as `wire`. `Wire` is also the default signal type in Verilog; any signal that you do not explicitly indicate a type of is automatically inferred as a wire. One of the major dis-advantages of using `wire` and `reg` types is that you must always remember to declare any signals set inside `always` blocks as `reg`. This is error prone and does not help in designing better hardware.

In contrast, in ECE342, as we are using System Verilog, you can use the new type called `logic`, which eliminates the need to worry about what signals should be declared as `reg`. Here, you can simply declare all your signals as `logic` and the compiler will take care of the rest.

There is an exception where using `wire` might make sense over using `logic`; `wire` variables can be initialized upon declaration, saving the need for an `assign` statement, but `logic` cannot, as shown below:

```
// Good
wire x;
assign x = a \& b;

// Good - shorthand
wire x = a \& b;
};
```

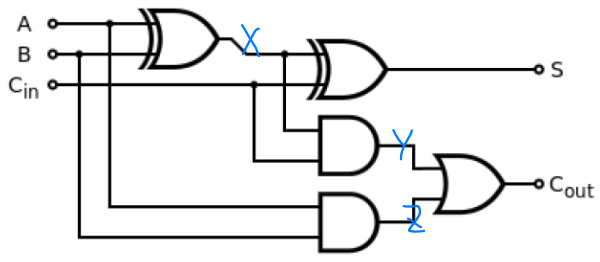
```
// Good
logic x;
assign x = a & b;

// Bad - compiles but doesn't
// actually perform the assignment
logic x = a & b;
```

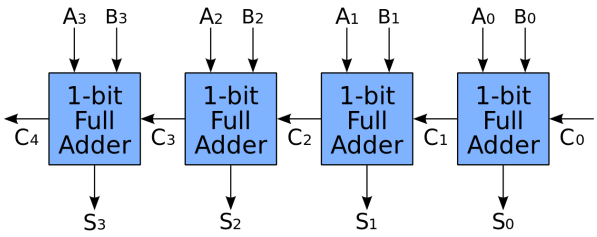
So please keep this in mind when trying to set a value to a `logic` signal and make sure to use a separate `assign` statement to do this. In effect, however, it is simpler to use `logic` everywhere and always use a separate `assign` statement to set signal values.

2.2 1-bit Full Adder Module

You will begin by designing a 1-bit full adder module (Figure 1a) first and then connecting it up to other full-adder blocks to make the final RCA (Figure 1b).



(a) 1-bit full adder circuit.



(b) 4-bit ripple carry adder (RCA).

Start by designing a module that instantiates a single 1-bit full-adder. You may use the module definition provided below. As mentioned above, since no signal type is indicated, all inputs and outputs are automatically declared as `wire`. So to keep things simple (and consistent), it is best to declare all your signals (including inputs and outputs) as `logic`.

```
module FA1bit(a, b, cin, s, cout);
input logic a, b, cin;
output logic s, cout;
// your code goes here
endmodule
```

2.3 8-bit RCA

Next, you must create an 8-bit RCA using your `FA1bit` modules. Complete the module declaration given below [1]:

```
module RCAmanual
(
  input logic [7:0] x,
  input logic [7:0] y,
  output logic [8:0] sum
);

logic [8:0] cin; // Internal signal for carry between FA modules

FA1bit fa0(.a(x[0]), .b(y[0]), .cin(1'b0), .s(sum[0]), .cout(cin[1]));
// Instantiate the rest of the full adder modules here.
endmodule
```

For simplicity, we assume there is no carry in to the 8-bit adder. Note that the way the input and output signals are declared is different from the `FA1bit` module above. This shows you another way that you can declare inputs/outputs. Feel free to use whichever style you prefer.

2.4 Generic hardware

To create 8 copies of the full-adder block, you must type out 8 separate lines of instantiation. This process is highly error prone, so (System)Verilog offers a better way to do this, using the `generate` block. These contain `if` statements and/or `for` loops that can conditionally instantiate hardware, or automatically create and name many instances of the same module.

The code for module `RCAgen` shows this. There are a few things to note about the syntax.

1. The `generate` block must have a matching `endgenerate`. Within it, any `if` or `for` blocks must have their own `begin` and `end` statements, and the blocks must be named (`'adders'` in this case).
2. Variables like `i` that are used in `generate for` blocks must be declared as a special `genvar` type, outside the `generate` block.
3. Although not shown, you are allowed to declare any other structural or behavioral code (like `assign` statements, creating `wire/logic` signals, entire `always` blocks) within `generate for` and `generate if` constructs.
4. Note that you cannot use `generate` statements inside an `always` block; they must be placed directly inside the module.

Carefully read through the code for this module to make sure you understand it fully. It is recommended that you draw out the circuit to make sure you understand how signals are connected. It is **VERY IMPORTANT** to understand that the for loop used here is not how you would use a for loop in C++. The for loop inside a generate statement is used to create multiple copies of hardware, not to run some instructions

[1] You will notice that all the code in this handout is provided as images rather than text. This is to make you actually write this code out instead of just copy-pasting it. Doing so will help you learn it better than just pasting it from the document.

```
module RCAgen
(
  input [7:0] x,
  input [7:0] y,
  output [8:0] sum
);
logic [8:0] cin;

assign cin[0] = 1'b0;
assign sum[8] = cin[8];

genvar i;
generate
  for (i = 0; i < 8; i++) begin : adders
    FA1bit fa_inst ( .x(x[i]), .y(y[i]), .cin(cin[i]), .s(sum[i]), .cout(cin[i+1]) );
  end
endgenerate
endmodule
```

multiple times (which is what for loops are used for in C++). This is a very common confusion for most students who see for loops in Verilog for the first time.

Once you understand how generate statements work, place both modules in the same file and save this file as `rca.sv` before proceeding. Note that System Verilog files are saved with the `.sv` extension, while Verilog files use the `.v` extension.

2.5 Writing Test benches

Before proceeding, it is important to make sure your module works as intended. In ECE241, you used a `.do` file using commands such as `force` to test your designs. In ECE342, you will be using `testbenches`, which offer much more flexibility compared to `.do` files.

2.5.1 Testbenches:

A testbench is a Verilog or SystemVerilog module that instantiates and exercises the actual hardware you wish to simulate. Although it is written in Verilog, it uses features of the language that make it unsuitable for synthesis to an FPGA in Quartus Prime. **It is only for use within a simulation environment such as ModelSim.**

Fundamentally, a testbench is a module that instantiates the hardware you wish to test, and stimulates its inputs in some way. It has no input/output signals of its own – any signals must be generated by the code within. Module `tb()` shows a simple testbench that just drives the adder with a single test case.

Compared to using a series of `force` commands in Modelsim, using testbenches has many benefits, including the ability to automate the creation of a large number of test cases and to have two-way interaction between the testing code and the hardware being simulated.

The testbench consists of signals to connect to your adder and then assigning them to fixed values for this simple test. There is no new Verilog syntax here yet. In fact, this looks just like any other module, except that there are no inputs or outputs.

The amount of time you run this simulation for is irrelevant, because the adder generates its output instantly. As you can see, the adder gives the correct result. If it did not, we'd dig deeper to find out why, and fix it.

```
module tb(); // no inputs or outputs
logic [7:0] dut_x, dut_y;
logic [8:0] dut_out;

// Instantiate the Design Under Test (the adder module we're simulating)
// Instantiate the RCA you want to test here: RCAmanual or RCAgen, from above.
RCA DUT (.x(dut_x), .y(dut_y), .sum(dut_out));

// Drive its inputs
assign dut_x = 8'd5;
assign dut_y = 8'd7;
endmodule
```

2.5.2 Testing multiple cases

We'll now replace the two `assign` statements with a more elaborate block of code that automatically drives *all* 65536 possible combinations. Each test case will be provided an (arbitrary) 5ns apart, because otherwise they would all happen at once. For this, we need new Verilog syntax that only works in a simulator (such as ModelSim) and not in real hardware.

```
`timescale 1ns/1ns  // Set default units for delay statements
module tb();
logic [7:0] dut_x, dut_y;
logic [8:0] dut_out;

RCA DUT ( .x(dut_x), .y(dut_y), .sum(dut_out) );

// initial block: execute this code only once, starting at the beginning of time
initial begin
  for (integer x = 0; x < 256; x++) begin
    for (integer y = 0; y < 256; y++) begin
      dut_x = x[7:0];
      dut_y = y[7:0];
      #5;  // Wait 5ns (ns because of the timescale directive)
    end
  end
end
endmodule
```

Rather than `assign` statements, the inputs are now driven from an `initial` block, which executes its contents much like a standard computer program (hence why this won't work in Quartus). Two `for` loops iterate over all possible inputs. We use separate `x` and `y` variables for the loop, rather than the `dut_x` and `dut_y` signals directly, because otherwise neither loop would actually reach the value 256 and would go on forever. The `integer` datatype is simply a shorthand for `logic [31:0]` and has enough bits for this.

Once again, note that you MUST NOT use an intial block inside any code that will run on an FPGA. Furthermore, note that we use a `for` loop in our testbench. This `for` loop is identical to the `for` loops you are familiar with in C++. But note that this `for` loop uses an `integer` as the iterator variable type, while the `for` loop in the synthesizable code, inside your full adder, uses a `genvar` type for the iterator. It is important to understand that, despite both types of `for` loops looking the same, they do very different things; one is used to create multiple copies of hardware while the other is used to provide multiple inputs to a module to test it. This is one of the most confusing aspects for many people when they are learning testbenches for the first time. So please take your time to understand this distinction so you can avoid potential problems in future labs.

There are still a few issues with this testbench that prevent it from being super useful. First, while all 65536

cases are indeed tested, you still need to manually go over each result and inspect it yourself in the ModelSim wave window, which is impractical. Second, it's not obvious how long to run the simulation for. Sure, you can multiply 65536 by 5ns, but we'd like to avoid that. With the above testbench code, the simulation will keep running forever after its last input, continuing to show the same values.

2.5.3 Automatic Correctness Verification

We'll address those shortcomings in the final version of the testbench for this adder. What we're going to do is make the testbench calculate the *expected* correct answer of the adder by itself (using the + operator), and then compare it to what the adder module outputs. If there's a mismatch, we can actually print a message into the simulator console giving details about what failed, and stop the simulation. This is accomplished using *system functions*, another simulation-only Verilog feature. These are Verilog functions that begin with the $ sign. Here's the new `initial` block of the testbench (everything else remains the same).

```
initial begin
  for (integer x = 0; x < 256; x++) begin
    for (integer y = 0; y < 256; y++) begin
      logic [8:0] realsum;
      realsum = x + y;

      dut_x = x[7:0];
      dut_y = y[7:0];
      #5;

      if (dut_out !== realsum) begin   // Not a typo
        $display("Mismatch! %d + %d should be %d, got %d instead", x, y, realsum, dut_out);
        $stop();
      end
    end // for y
  end // for x

$display("Test passed!");
$stop();

end // initial
```

The `$display` system function behaves just like `printf` does in C, with placeholders that are replaced by actual parameters following the format string. You can use `%0d` instead of `%d` to fix the strange way ModelSim indents the values when printing. The `$stop` function ends the simulation, meaning that you no longer have to pick a specific amount of time to simulate for with the `run` command. Instead, you can use `run -all`.

Note that you can also view the `realsum` signal in the wave window. Since it's a local variable, it will be found in the `sim` window inside several nested entries that have names like `#anonblk#` and `#ublk#`. The signal itself will then be found in the Objects tab.

Lastly, the `!==` operator lets you compare against `'x'` and `'z'` values. Recall that `'x'` is an 'unknown' value, which happens when you set a signal at multiple points in your circuit. `'z'` is 'high impedence' which means that that signal is not being driven anywhere and is 'floating' (i.e, it is wire connected to nothing).

Now that we have a testbench, we can use this in ModelSim to test the RCA to make sure it's working as intended. Save this file as `rca_tb.sv` before proceeding.

2.6 Running ModelSim

Just like in ECE241, you will be using ModelSim to test your designs in ECE342. **Please note that ModelSim and Quartus are two completely separate softwares. You do not need to compile (or even open!) Quartus to check if your code is working.** You should do the majority of your development work using ModelSim to make sure you design works fully before moving to Quartus to test on

the FPGA. **You only need to compile your code in Quartus if you are going to run it on the FPGA.**

To get started, you must first compile your source files (Verilog (.v) and SystemVerilog(.sv)) using the `vlog` command in ModelSim. By default, files will be searched for in the current directory, which can be changed or viewed with the `cd` and `pwd` commands, respectively.

Compiled versions of (System)Verilog modules are placed into a *library*, with the default one being named "`work`" by convention, and will be created in a subfolder of the same name. You must also create this library before you can compile:

```
cd ../path/to/source/files
vlib work          # creates 'work' library
vlog <filename>    # compiles source files into 'work' library
```

You can compile individual files (e.g., `vlog file1.sv`), multiple files (e.g., `vlog file1.sv file2.sv`) or even all the `.sv` files in that folder (e.g., `vlog *.sv`) Compile the two files you saved earlier as shown above and then start simulation. To do this, run the following command:

```
# Starts simulation of the previously-compiled module called 'tb'.
# Additional errors in your code may be caught here, rather than during vlog
vsim -novopt tb
```

The `-novopt` flag will disable optimization, allowing you to peek at internal signals in your hardware, rather than just the input/output pins. This is similar to how when writing C/C++ code, you pass the `-g` flag to `gcc` or `g++` to disable optimization and allow debugging. This is of course optional but recommended for you to add when doing simulation. Once you run this command, you should see that your testbench runs and you should see the message `Test passed!`, displayed at the terminal window, at the bottom of the screen.

By default, you will not see a waveform (as you saw in previous courses). However, to aid with debug, you can show the waveform by running:

```
log *
add wave *
```

The `log` commands tells ModelSim to save values for those signals. You can then add them to the waveform window with the `add wave *` command. The last thing to do is to `run` the simulation, you do that by running this command:

```
run -all
```

Lastly, since this is a lot of commands so to avoid writing them out many times, you can put all the commands into a separate script file, and run it from ModelSim with '`do` ***`scriptfile`***'. Now that we have covered how to write and run a testbench, take a look at **Appendix B** which provides some recommendations on how to effectively use the ModelSim GUI window. This can help you out during the term as you will be using ModelSim extensively. These are provided in a separate Appendix to make it easier for you to refer to it in future labs. Once your code is working and you pass all the test cases, you can move on to Part II.

3 Part II: N-bit upcounter

In this part of the lab, you must design an N-bit upcounter. You will learn about the two different types of `always` blocks in System Verilog, as well as how to build parameterized hardware. Lastly, you will learn about how to test sequential modules such as counters using testbenches.

3.1 New `always` blocks

SystemVerilog introduces two new kinds of `always` blocks that help you keep your code error-free.

3.1.1 `always_comb`

A common problem with writing Verilog for FPGAs is the accidental creation of latches. This happens when the behavior specified in an `always @*` block requires one or more `reg` variables to maintain its old value. Without a clock, the only possible hardware to realize this is a latch, which is unreliable when compiling for the FPGA and behaves differently in simulation.

The `always_comb` construct is a drop-in replacement for `always @*`. It forces the creation of purely combinational logic. If a latch is accidentally inferred, it generates a *compile-time error* rather than an easily-overlooked warning. Figure 2 shows the difference between the two when compiling in Quartus Prime.

```
always @* begin
    if (condition) signal = 1'b1;
end
```

```
12125 Using design file DE1_SOC_golden_top.sv, which is not specified as a design file for the current project, but co
12127 Elaborating entity "DE1_SOC_golden_top" for the top level hierarchy
10240 Verilog HDL Always Construct warning at DE1_SOC_golden_top.sv(260): inferring latch(es) for variable "signal", w
10034 Output port "DRAM_ADDR" at DE1_SOC_golden_top.sv(100) has no driver
10034 Output port "DRAM_BA" at DE1_SOC_golden_top.sv(101) has no driver
```

```
always_comb begin
    if (condition) signal = 1'b1;
end
```

```
12125 Using design file DE1_SOC_golden_top.sv, which is not specified as a design file for the current project, but contains definiti
12127 Elaborating entity "DE1_SOC_golden_top" for the top level hierarchy
10240 Verilog HDL Always Construct warning at DE1_SOC_golden_top.sv(260): inferring latch(es) for variable "signal", which holds its
10166 SystemVerilog RTL Coding error at DE1_SOC_golden_top.sv(260): always_comb construct does not infer purely combinational logic.
```

```
always_comb begin
    if (condition) signal = 1'b1;
    else signal = 1'b0;
end
```

OK

Figure 2: `always @*` versus `always_comb`

3.1.2 `always_ff`

The `always_ff` block is another way to make clocked `always` blocks for creating sequential logic (registers). It works exactly the same as using the `always` block in Verilog with `posedge` or `negedge` specifiers, except that it will throw a compiler error if `posedge/negedge` are missing. This is shown in the code segments shown below.

Together, `always_comb` and `always_ff` blocks can help you write error-free code in System Verilog. You will use these new blocks to design your upcounter next.

```
// makes registers - works in Verilog
// and SystemVerilog
always @ (posedge clock) begin
...
end

// also makes registers - SV only
always_ff @ (posedge clock) begin
...
end
```

```
// Forgot 'posedge' - compiles but
// makes combinational logic or latches
always @(clock) begin
...
end

// Compiler error - can't infer registers
// without pos/negedge
always_ff @(clock) begin
...
end
```

3.2 Designing a 4-bit upcounter

To get started, complete the module below to design a 4-bit upcounter. You must fill in the missing parts indicated with .... The comments next to each line tell you what should be added to complete the code.

```
// Module to count upto 15.
module upcount15
(
input clk,
input sreset,       // Stands for 'synchronous reset'
output [3:0] o_val, // Debug output to see the count value.
input i_enable,     // input enable. Counter should only count when this is 1'b1
output o_last       // Output to indicate the counter has reached 15.
);
logic [3:0] count;

always_ff @ (posedge clk) begin
  if (sreset) count <= ...;      // On reset, set the value to 0
  else begin
    if (i_enable) begin
      if (o_last) count <= ...;  // What should you do if the counter reaches max value?
      else count <= ...;         // Increment the counter
    end
  end
end

assign o_val = count;
assign o_last = (count == ...); // set this output high when count reaches the final value of 15

endmodule
```

Similar to Part I, we will now write a testbench and simulate the code in ModelSim to test it. However, writing testbenches for sequential circuits, such as the upcounter, requires learning about several new testbench features.

3.3 Writing sequential testbenches

In contrast to combinational testbenches (such as the one you wrote for Part I), sequential testbenches must change their inputs and check your design's outputs, according to a clock signal. We begin by seeing how you can generate a clock in your testbenches.

3.3.1 Generating Clocks

A clock is essential for every sequential circuit. To generate a clock inside a testbench, you must do:

```
logic clk;
initial clk = 1'b0;
always #10 clk = ~clk;
```

An `initial` block sets the clock to a known value at the beginning of simulation. Note that multiple `initial` blocks are allowed in your testbench, as long as they don't try to drive the same signals. They all start running at time=0 and run in parallel. An unconditional `always` block loops its statements forever. The statement being looped is `#10 clk = ~clk`, which waits 10ns and then flips the clock, yielding a 50 MHz signal.

3.3.2 Waiting and Synchronizing

There are statements that let you wait for the next positive (or negative) clock edge, and wait for arbitrary conditions to be satisfied before continuing. The serial adder circuit takes some number of clock cycles to finish before raising its 'done' signal, and we can simply wait for this condition to be true.

```
@(posedge clk);   // Waits until the next positive clock edge
@(negedge clk);   // ... or negative clock edge
wait(dut_done);   // Waits for dut_done == 1'b1
```

This can be very useful when testing code such as a counter where you are waiting for a specific output (`o_last`, in the case of the counter). This is shown in the `upcount15_tb()` module.

```
'timescale 1ns/1ns
module upcount15_tb();
// Generates a 50MHz clock.
logic clk;
initial clk = 1'b0;     // Clock starts at 0
always #10 clk = ~clk;  // Wait 10ns, flip the clock, repeat forever

logic sreset;
logic dut_enable;
logic dut_last;
logic [3:0] dut_val;
upcount15 DUT(.clk(clk), .sreset(sreset), .o_val(dut_val),
              .i_enable(dut_enable), .o_last(dut_last));

initial begin
  dut_enable = 1'b0;  // Start with the enable signal off
  reset = 1'b1;       // Start with reset on

  @(posedge clk);
  reset = 1'b0;       // Leave it on for a clock cycle and then turn it off

  @(posedge clk);
     dut_enable = 1'b1;  // Set the enable signal on

  wait(dut_last);     // Wait for counter to finish.
  if (dut_val !== 4'd15) begin
    $display("Error! Counter asserted o_last, but o_val was %d, instead of 15.", dut_val);
    $stop();
  end

  @(posedge clk);  // Wait 1 cycle before doing something else
  $stop();
end
endmodule
```

Note that all your cases go in an `initial` block, except for the single `always` statement to generate the clock. Run this testbench through ModelSim to ensure your `upcount15` module is working as intended before moving onto the next part. As before, you can add your commands to a `.do` file to make re-running simulations easier.

Be careful when using the `wait()` command. It will wait as long as necessary for the signal to go to `1'b1`. If there is a bug in your code and the signal never becomes `1'b1`, your testbench will run forever. If you see it running for more than a few seconds, you should probably stop it, debug your code (using waveforms) to make sure your signal is correctly toggling when you expect it to.

3.4 Creating parameterized hardware.

The code you designed only works to count to 16 cycles. However, it is often beneficial to write verilog once to generate counters (or other hardware) that can be parameterized. In this section, we will see how to modify your upcounter to count up to any given value.

The `parameter` keyword in Verilog allows a module to be configured during instantiation, with the configuration value visible as a named constant for use within the module. Here is the code to instantiate an 8-bit adder using parameterized instantiation:

```
module upcount #
(
parameter N = 20,
parameter Nbits = $clog2(N)
)
(
input clk,
input sreset,
output [Nbits-1:0] o_val,
input i_enable,
output o_last
);
logic [Nbits-1:0] count;

always_ff @ (posedge clk) begin
  if (sreset)
    count <= 'd0;
  else if (i_enable) begin
    if (o_last) count <= 'd0;
      else count <= count + 'd1;
    end
end

assign o_val = count;
assign o_last = count == N-1;
endmodule
```

At the start of the counter module is a parameters section that begins with a #, coming before the usual input/output signals section. In here you can put one or more `parameter` declarations. The parameter N is used later within the module to set a maximum value to count to.

Parameters can also depend on other parameters. The `$clog2` built-in system function (which is only available in SystemVerilog) evaluates (at compile time!) the base 2 logarithm of the value and rounds it up to the next integer. In effect, it counts how many bits are required to represent a given value. But it can only work with a constant value; so don't try to pass in input to `$clog2`. It will result in a compile-time error.

Also, note that the values loaded to the `count` register do not have a bit-width specified. System Verilog allows for values to be assigned without a specific width (e.g., 'd0 instead of 4'd0). This can be handy, particularly when using parameterized modules.

Lastly, parameters can also have default values assigned to them, so that the instantiating module doesn't have to explicitly set them. This was done above for `Nbits` for example. However, the `N` parameter does not have a default value and therefore must be set when this module is instantiated. Let's see how that is done now.

3.5 Instatiating parameterized hardware.

Now, let's see how such parameterized modules are instantiated. Make the following changes to your upcount testbench from before.

```
logic [3:0] count17_val;  // goes to 16
logic [6:0] count112_val; // goes to 111

// explicit parameter and signal mappings
upcount # (.N(17)) count17
(
.clk(clk),
.o_val(count17_val),
// ... rest of connections
);

// ordered parameters, explicit signals
upcount #(112) count112
(
.o_val(count112_val),
// ...
);
```

Inside the testbench, there are two instances of the counter: one that counts from 0 to 16 and one that counts to 111. You can keep the `clk, sreset` and `dut_enable` signals. But you will need to add separate signals for the two counters since they each count up to different values. Note that the bit-widths of the two signals are different so that they match the bit-widths in the `upcount` modules.

When instantiating a parameterized module, there is a section that begins again with a # **but comes before the instance name**. This section assigns concrete values to the parameters, and looks a lot like the usual signal-connection section (with parameter names being set instead of signals being connected). It also comes in the same two flavours: one which uses explicit by-name assignments to the parameters (using .), and a more compact one that does it by order. Both are shown.

There is an older style of syntax that you may still see. It's still valid in the latest versions of Verilog and SystemVerilog, so for completeness, here's the same example using it:

```
// upcount_old.v
module upcount
(
input clk,
input sreset,
output [7:0] o_val,
input i_enable,
output o_last
);
reg [7:0] cnt;
parameter N;

// rest of code is the same
endmodule
```

```
// top_old.v
upcount count64
(
// just port connections
);

defparam count64.N = 64;
```

Next, write a testbench to test the parameterized upcounter for the two values of N (i.e., 12 and 117) shown above. You will need to test them one after the other.

4 In-lab demo

For each lab, you must demonstrate by showing that your code passes all the required test cases in ModelSim.

1. **Part I**: (4 marks) Show that your 8-bit Ripple carry adder passes all 65,536 test cases, using a testbench.
2. **Part II**: (6 marks) Show that your N-bit upcounter works for at least 3 different values of `N`, using a testbench.

4.1 (OPTIONAL) Running on an FPGA

While it is not required for marks, you may wish to run your code on an FPGA (if you had purchased one for ECE241 or ECE243). Tables 1 and 2 show a possible mapping you can use to connect the inputs and outputs of your modules to the FPGA. Refer to Appendix A for instructions on how to download and run your code on the FPGA in the lab.

Signal name	x	y	sum
FPGA port	SW[3:0]	SW[7:4]	LEDR[5:0] & HEX1,HEX0

Table 1: Signal mapping for Part I

Signal name	clk	sreset	i_enable	o_last	o_val
FPGA port	CLOCK_50	KEY[3]	KEY[0]	LEDR[9]	LEDR[3:0] & HEX0

Table 2: Signal mapping for Part II

It is often beneficial to connect a single output to multiple ports on the FPGA, to make debugging easier. For example, for Part I, you can connect the `sum` output to `LEDR[5:0]` as well as to `HEX1, HEX0`. This means that that `sum[4]` must be connected to `HEX1` and `sum[3:0]` must be connected to `HEX0`. Similarly, you can think about what you should do with the remaining inputs to `HEX1` to make sure the number displayed on the `HEX` is correct.

5 Appendix A: Running your code on the FPGA

If you own an DE1-SoC, you may wish to try your code on the FPGA. To do this, download the `fpga_kit` that has been provided for you. This kit contains the `de1soc_top.sv` file as well as some other files you will need to get accurate timing results for the FPGA. You do not need to worry about modifying these files; just copy them along with the `de1soc_top.sv` to a new folder for each part that you want to download to the board.

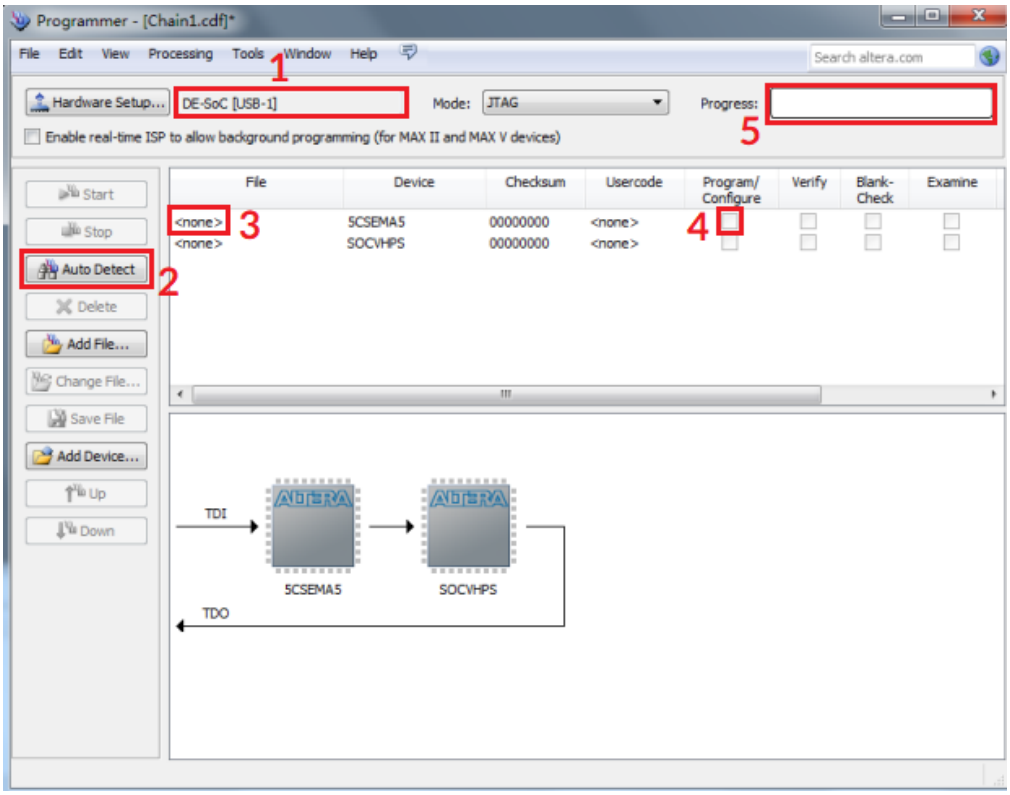


Figure 3: Quartus programmer window

Now, you can create an FPGA project to run your code on the board.

1. Once your folders are created, copy the contents of the `fpga_kit` into the folder you want to use. You should do this each time for each part of each project of each lab. For convenience, it may be useful to save a copy of the `fpga_kit` directly in the `ECE342` folder so you have easy access to it. Just copy these files into a new folder each time and continue.

2. Next, copy the verilog files you want to run on the board into this folder. Note that you only need to copy the files for your own modules; you do not need the testbench files or any `.do` files you may have used for ModelSim earlier. (However, it may be easier for you to just keep all the files for each part of each lab in a single place. In which case, feel free to keep the test bench files here as well. Just be sure not to include them in your Quartus project as they cannot be run on the board.)

3. Edit the `de1soc_top.sv` file to instantiate your module and make the necessary connections. The connections you should make are provided in the 'In-lab Demo' section of each lab.

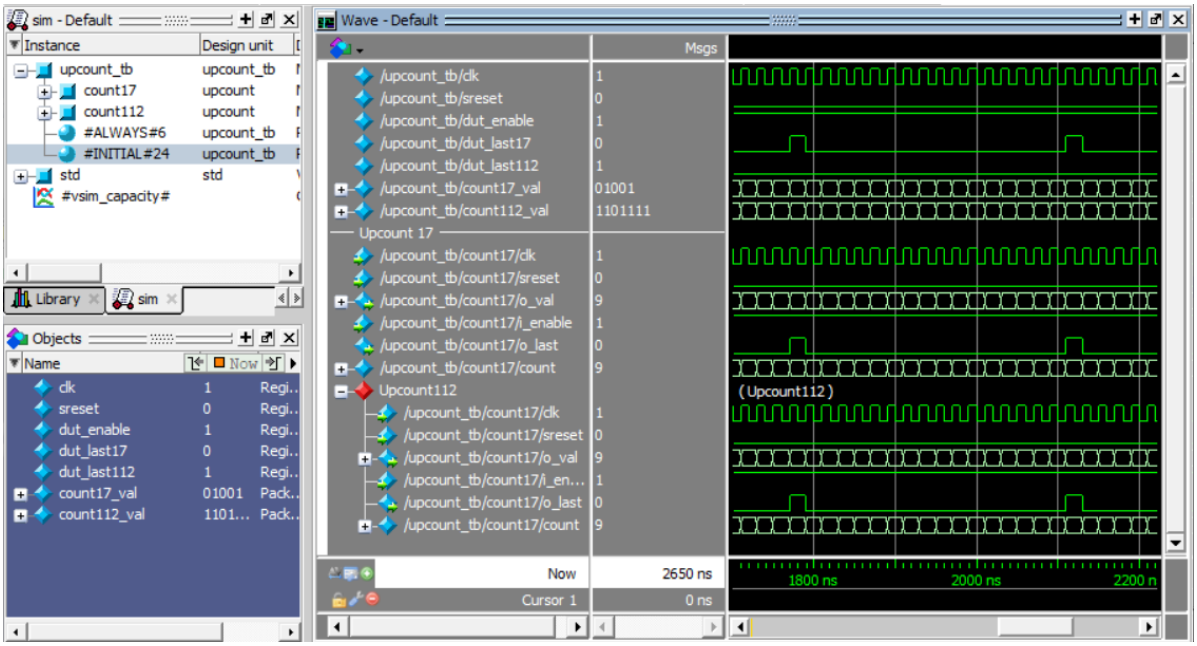
4. Once finished, compile your program. **Make sure** to read through any warning messages you get. Quartus often prints errors as warning messages. This can often lead to you spending a long time debugging an issue only to find out Quartus simply optimized a part of your hardware away. So its good to get into the habit of always looking at warning messages when you compile your program using Quartus. Common Quartus issues (and solutions) are provided in Appendix 7.

5. If you do not have any errors/warnings, click `Tools` and select `Programmer`.
6. In the programmer window (shown in Figure 3), you should see the text `DE1-SoC [USB-1]` next to `Hardware Setup`. This is shown in Figure 3 as item 1. If you do not see this, make sure you are connected to the FPGA and that the FPGA is powered on. Close the programmer window and reopen it to make sure the programmer can see the FPGA.
7. Now, you may see one device at the bottom. (Figure 3 shows two devices for example). Select the one device and press `Delete` to remove it. When both the top and bottom windows on the right side are empty, click on `Auto Detect` (2 in Figure 3).
8. In the window that appears, select the 2nd option (i.e., `5CEMA5`) and click OK.
9. You should now see the Programmer window again and it should look identical to Figure 3.
10. Double-click on the text `<none>`, next to the device `5CEMA5`. This will now prompt you to select a file to program.
11. You should be in your current projects root folder. Open the `output files` folder and select the `.sof` file there.
12. Next, check the `Program/Configure` box (shown as 4 in Figure 3). Then click `Start`.
13. The FPGA should now be programmed. You can confirm this by checking the `Progress` box (shown as 5 in Figure 3). This should say `100%`. If it instead says `Failed`, that means something went wrong and you should debug your design.
14. When you close the Programmer window, it will ask if you wish to save the `.cdf` file. You can click yes to this, as Quartus will now use the same settings each time you program that project and save you from having to do these steps each time for the same project.

You can confirm your project was downloaded to the board by seeing that the `LEDs` and `HEXes` are no longer changing but should remain fixed. You can now try your project on the FPGA to make sure it works as expected.

6 Appendix B: ModelSim GUI layout

Since you will be spending a lot of time in ModelSim to debug your designs, it will be useful to know more about the ModelSim GUI and how you can make the best use of it. The figure below shows the window after you have run a `vsim` command for the `upcount` module shown in Part II. All the windows you see will not be visible at first until you start simulation.



On the right is the 'waveform' window which you should be familiar with. On the top left is the 'sim' window which lists all the modules in the current simulation, following the hiearchy you made when writing your code. You can click on the '+' sign next to each module to see sub-modules (if any). The sim window also lists 'processes' which correspond to the various blocks in your design. But for your simulation purposes you will only need to work with the 'modules' listed in the sim window.

Below the Sim window is the 'objects' window which lists all the signals in each module. Note that this only shows the top level signals in each module and not signals in any sub-modules. So between the sim and objects windows, you can navigate through all the modules and signals in your design.

6.1 Adding Signals

By default when you use `add wave *`, only the signals in the top-level module you simulate are added to the waveform window. But you will almost always want to add signals further down the module hiearchy. There are two ways to add signals to the Wave window: with commands, or by dragging-and-dropping (an entire module instance from the 'sim' tab, or individual signals from the Objects tab).

At first, you can drag-and-drop signals to see the command that is run in the terminal at the bottom. You can then add that command to your `.do` file to add those signals each time you run a simulation.

Another useful feature is changing the base of signals displayed in the Wave window. Select one or more signal names, right click, and select the *Radix* menu. *Unsigned* interprets the signal as unsigned decimal, and *Decimal* as signed decimal. You can also set these in your `do` file by passing in the `-radix` option.

There are also other commands to better organize the waveform window, such as:

1. **Dividers** These add a horizontal line in the waveform to help visually separate different groups of signals.

2. **Groups** This adds a set of signals as a single group so you can easily hide a set of signals.

3. **Leaf names** By default, ModelSim will show the full name of each signal, starting from the top-level module. This allows you to only show the signal name to make it easier to find the signals you want.

Let's see how to use each of these above using an example `do` file. This creates the output seen in the figure above.

```
vlib work
vlog -novopt upcount.sv upcount_tb.sv
vsim upcount_tb
log *
# Add the signals from the top-level module first.
add wave *
# Add a divider called Upcount 17
add wave - divider "Upcount 17"
# Add all  the signals for upcount17
add wave sim:/upcount_tb/count17/*
# Create a group for all the signals in upcount 112
add wave -group FSM /tb/DUT/state /tb/DUT/nextstate
```

Lastly, you will notice that ModelSim always open your testbench file in its editor when it sees a `$stop` command. If you would like to turn this off, you can do so by going to: `Tools > Edit preference > By name > source` and setting `Open on Break` to 0.

6.1.1 Running the Simulation

After adding the signals you wish to monitor, use the `run` command to begin, or continue, the simulation:

```
run 5us   # run for a specific amount of time
run -all  # run until the testbench code explicitly stops the simulation
```

The most common thing you will be doing is fixing/modifying your source code and then re-simulating. This is possible to do **without** having to exit and re-enter simulation mode:

```
<modify and save some Verilog files>
vlog *.sv      # recompile one or more files...
do compile.do  # ... or run your compile script again
restart -f     # restarts simulator at time=0, keeping the same Wave signals
```

Finally, to exit simulator mode:

```
quit -sim
```

6.1.2 GUI Hotkeys

The table below also lists some useful hotkeys you can use inside the ModelSim GUI.

F	Zooms to fit entire simulation into the Wave window
C	Centers and zooms Wave window on active yellow cursor
+/-	Zoom in and out on position pointed at by mouse cursor
Tab	Advances yellow cursor to next transition of selected signal
Shift+Tab	... previous transition ...

7 Appendix C: Common Software issues

This appendix lists several common issues you may encounter when using Quartus. If you encounter a problem with Quartus, be sure to check here first. This appendix, however, **DOES NOT** tell you how to solve Verilog issues. For those, you must debug them yourself and figure it out. Almost always, Verilog issues are due to syntax. So be sure to check the exact line number of the error to spot the issue.

7.1 Quartus issues

This section covers several common errors you may see with Quartus. Keep in mind that following the steps listed in this handout will help you avoid many issues you may face (e.g., using `always_comb` and `always_ff` blocks, using the provided kit when you wish to program the FPGA). Below you will find some other issues and how to resolve them.

Please note that Quartus reports some major issues warnings rather than errors. So it is a good idea to always go through the warnings after you compile your program. Warnings that you should fix are also listed below.

1. **Cannot find database file** or **Error: Symbolic name "WIDTH" is used but not defined.**

   You have files saved somewhere other than W: drive. Make sure you files are stored in W: and not in any other location such as Desktop or My Documents or C: . To confirm you have opened the project correctly, check the path shown at the top of the quartus window. The path is shown right after the text 'Quartus Prime Starter Edition'. If it starts with '//', then your path is wrong. If it starts with 'W:/' then your path is correct.

2. **Top level design entity "..." is not defined.**

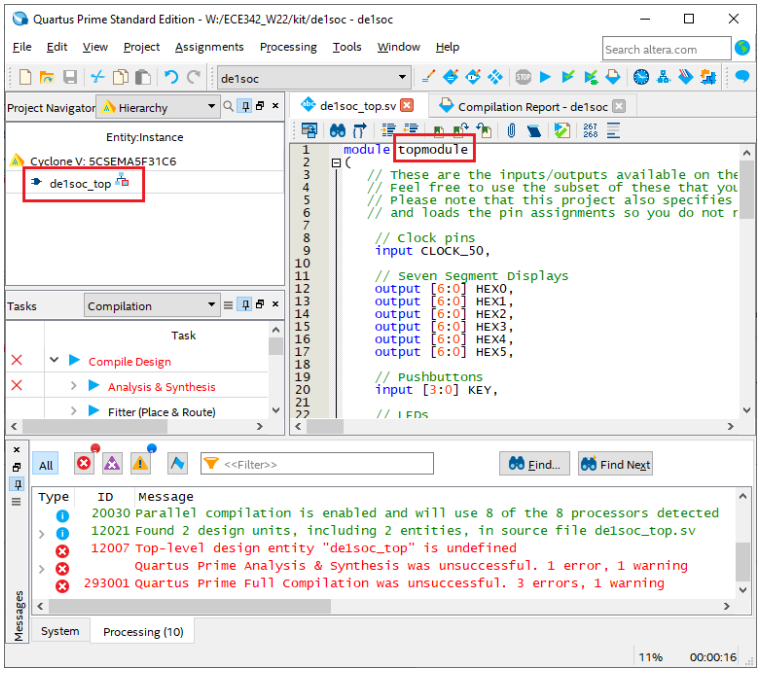


Figure 4: Top level module undefined error in Quartus

This error is shown in Figure 4. The box on the left shows the name of your project. The box on the right shows the name of the top level module. Quartus requires these to be the same. If they do not match, rename one of them to match the other. To rename your project, right click on the project name (the left box in the Figure), and select 'Settings'. In the dialog box that appears, you can type in the name of your top level module in the 'Top level entity' box or select from one of the modules in your design using the '...' option next to the box.

3. **Quartus crashes unexpectedly**

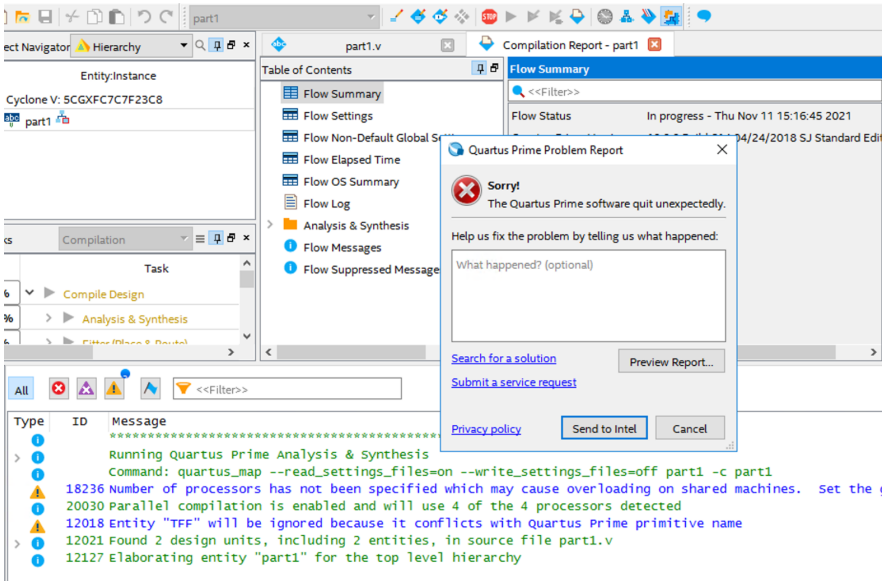


Figure 5: Quartus has unexpectedly crashed.

This is a tricky error to pinpoint exactly. But it is often caused by using a reserved Quartus keyword as shown in Figure 5. If you are not using any reserved keywords but are still seeing this crash, try to comment out sections of code to see which line is causing the crash. This can often help you solve the issue causing the crash.

4. **Verilog HDL assignment warning: truncated value with size 32 to match size of target (8)**

   This error is caused by assigning a 32-bit integer value to an N-bit logic signal. Numbers with no bit-width specified (e.g., '1' instead of '1'b1') are automatically inferred as 32-bit integer values. So if you have the code snippet below:

```
logic [7:0] count;
count <= count + 1;
```

   The 32-bit value (i.e., '1') is assigned to the 8-bit value 'count', causing this warning. To avoid this, be sure to always specify bit-widths for all numbers. You can easily do this by changing the increment to `count <= count + 'b1;`. This warning is often not an issue but in some cases, it it possible you are actually assigning signals incorrectly.

5. **Implicit net created**

7.2 ModelSim issues

In general ModelSim issues tend to be Verilog syntax related. So be sure to check all the error messages you see in the modelsim terminal and fix those first.

1. **Error: (vsim-3601) Iteration limit reached at time x ns.**

   This error is caused by the presence of a 'combinational loop' in your code. A combinational loop is caused by having a feedback path in your loop, without a flip-flop in the path. The code shown below on the left shows this.

   This asks the compiler to create an adder where one input is the output of the adder. This causes a problem for modelsim; how can it calculate the output if at each time, the input changes along with

```
// Bad - Combinational loop.
always_comb begin
  count <= count + 1;
end
```

```
// Good
always_ff @ (posedge clock) begin
  count <= count + 1;
end
```

the output. (This is also, of course, incorrect in code. In a real system, this will result in 'glitches' where the value of count would be indeterminate). You can fix this by making sure the code is placed inside an `always_ff` block, as shown in the code above, on the right.