Writing a Testbench in Verilog & Using Modelsim to Test

1. Synopsis:

In this lab we are going through various techniques of writing testbenches. Writing efficient testbenches to help verify the functionality of the circuit is non-trivial, and it is very helpful later on with more complicated designs. The purpose of this lab is to get you familiarized with testbench writing techniques, which ultimately help you verify your final project design efficiently and effectively. You will also learn scripting DO files to control simulation in modelsim and to facilitate quick repeated simulations during debugging.

2. Importance of Testing:

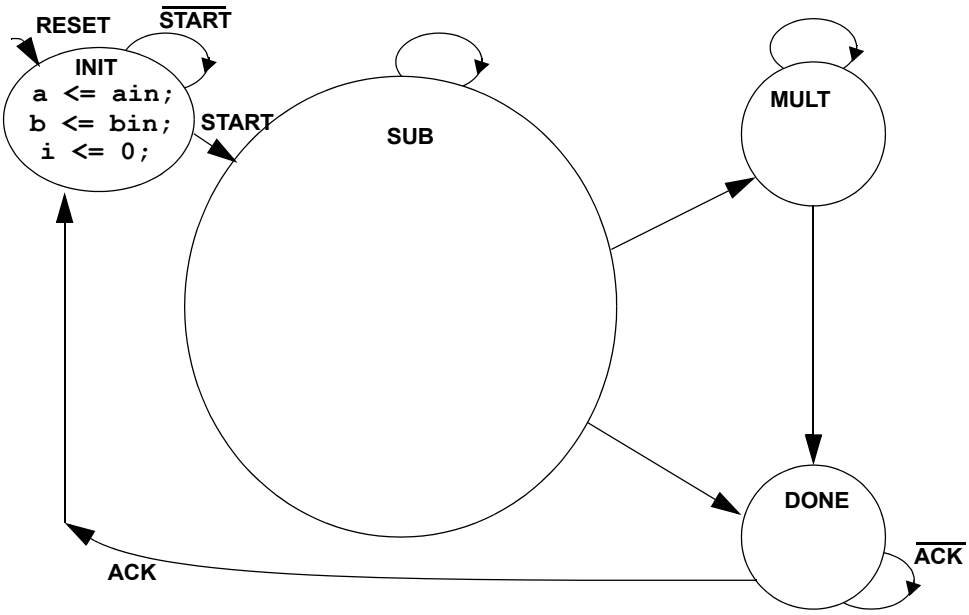
Most of the exercises that you have done in this course so far are to design the core system. Whether it is a state machine or some combinational logic, this is the most challenging part. In real life, however, testing a system is often equally as challenging and important as building the core design. This is because in hardware design, the cost of making a mistake can be extremely high. Since (non-FPGA-based) hardware cannot be "updated" like software can be, often times, the only way to fix an error in the hardware is to replace the part, and replacement can be a very expensive proposition.

Consider, for example, the case of the infamous *FDIV bug* that was found in Intel Pentium processors in the mid-90s. In late 1994, a mathematics professor at Lynchburg College in Virginia found that the Intel Pentium processor in his computer was consistently producing wrong result when certain two numbers were divided. The bug was so hard to find that, if you randomly tried to divide two numbers, there was only one in a 9 billion chance that you will encounter the case for which the division operation produced a wrong result. Yet, the story about this bug became a marketing nightmare for Intel, and despite all efforts, Intel had to recall all the defective processors. It has been reported that the final cost of the recall was $475 million!! Don't you think Intel wished it could just release a "security update" that Tuesday and get it over with?

Testing a design comprehensively and efficiently is a major challenge in a complex design. In this lab you are learning the Verilog syntaxes and coding techniques that can assist in writing efficient testbenches. Furthermore, in this lab Modelsim simulator will be used in standalone mode. That is, instead of creating our project in Xilinx ISE and launching Modelsim from the Project Navigator, you will use Modelsim's graphical user interface (GUI) to simulate the code. Since the focus of this lab is on writing testbenches, we will re-use the GCD design from previous lab and write a new, advanced testbench for it. Notice that this is a simulation-only exercise.

3. GCD Review:

The GCD state machine takes two 8-bit unsigned numbers as inputs, **`Ain`** and **`Bin`**. The values of **`Ain`** and **`Bin`** are accepted by the state machine when **`start`** goes high. Once **`start`** is received by the state machine, using the simple four-state mealy state machine, as shown below, the GCD of these numbers is computed. Upon the completion of the computation, the state machine enters the **`DONE`** state and stays there until the acknowledgement signal (**`ack`**) is asserted. When **`ack`** is received, the state machine enters the **`INITIAL`** state again and waits for the **`start`** to go active.



4. Using Modelsim Only (without Xilinx ISE) for simulation and verification

Unlike Xilinx ISE, Modelsim cannot synthesize/implement the design into real hardware, but it can compile and simulate HDL-based design, and display graphical and text information to facilitate debugging. The main advantages of using Modelsim standalone are convenience and speed. That is, instead of editing your code in the Project Navigator editor and re-invoking Modelsim every time a small change is made (like we have been doing so far), by using Modelsim standalone you can edit the code, re-compile it and re-simulate it -- all without closing the applications. The procedure to simulate a design in Modelsim is simple:

1. Create a new Modelsim project.

2. Add existing source files to the project or create new Verilog source files.

3. Compile all source files.

4. Start simulation.

5. Run the simulation for the desired length of time.

If you find some errors, whether they are compilation errors (syntax errors that are reported during compilation), or functional errors (you notice after simulating and observing waveforms), you can edit the code and repeat steps 3-5 to re-simulate the design. More details on each of the five steps are give in the Procedure section. Steps 3 to 5 can be automated using a DO file (with .do extension).

5. Writing Testbench

The function of a testbench is to apply stimulus (inputs) to the design under test (DUT), sometimes called the unit under test (UUT), and report the outputs in a readable and user-friendly format. In this section, we discuss how an efficient testbench can be written. Procedure steps requiring you to write the testbench for the GCD design directly refer to elements of the testbench discussed in this section. Note: Many of the coding techniques used in testbenches (such as file I/O, the ***initial*** block, etc) are not suitable for synthesis.

5.1 Defining the timescale

Before the module definition of the testbench module begins, Modelsim requires a compiler directive that defines the time unit and the precision level at which the simulation runs. Defining the time unit is necessary so that the simulator knows whether, say, `#10;` means wait for 10ns or 10ps or 10us. The syntax for this directive and a typical/recommended example is given below:

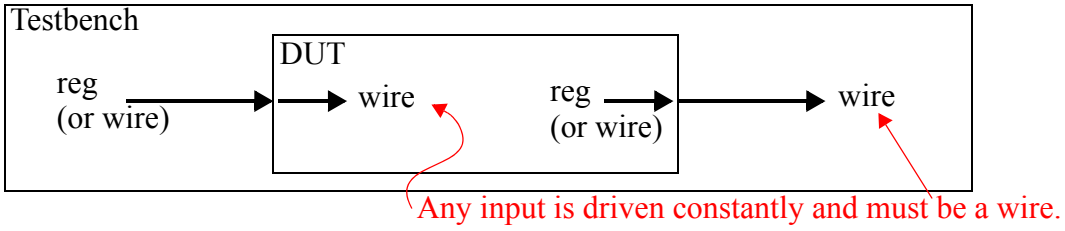
Syntax	Example
`` `timescale <unit>/<precision> ``	`` `timescale 1ns/1ns ``

All compiler directives begin with the `` ` `` (accent grave character). Recall the **`` `define ``** directive that we used in our state machine designs to create a macro for substituting the text with the macro name.

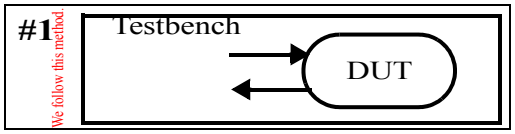
5.2 Instantiating the Design Under Test (DUT)

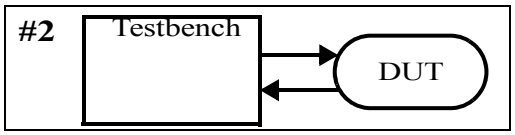
Every testbench must instantiate the design that it is expected to test. This design is usually referred to as the "design under test" (DUT) or "unit under test" (UUT). Before the DUT is instantiated, each of its inputs and outputs must be declared in the testbench. Inputs to the DUT are declared as `reg` and outputs are declared as `wire`. Note that the outputs of the DUT are inputs to the testbench. Remember that any input must be a wire. The inputs to DUT are the stimuli generated in testbench. Stimuli are usually generated in an `initial` or an `always` procedural block in the testbench. As you know, any object assigned inside an `initial` or an `always` procedural block shall be a *register* type object, `reg` or `integer` or `real` or `time`. (Note: We know that the *register* type object does not mean a *hardware register* unless it is assigned in a clocked **`always`** statement sensitive to **`posedge clk`** or **`negedge clk`**. The Verilog 2001 replaces the term "*register*" with the more intuitive term "*variable*" to avoid this common confusion among learners. Refer http://www.sutherland-hdl.com/papers/2000-HDLCon-paper_Verilog-2000.pdf). We are not using *register* types `real` or `time` in this course. We use `reg` in synthesizable designs (though we can use integer or real also in synthesizable designs) and use mostly `reg` and occasionally `integer` in testbenches.

Example: `integer clk_cnt, start_Clk_cnt, clocks_taken;`



Notice that there are usually no ports for the testbench itself. Ex: `module ee201_GCD_tb_Part1;` There are usually two methods for the testbench to interact with the DUT as shown below. We use the first method where the testbench *contains* the DUT and does not require any ports. In the second method, a higher module needs to instantiate the testbench and the DUT to tie them together, and the testbench module has ports.





Instantiation of the DUT can be made using 'positional' association or 'named' association. In positional (or implicit) association the order in which signals are listed in the instantiation determines how they are connected to the ports of the DUT. In named (or explicit) association, each DUT port signal is explicitly associated to a testbench signal. Named association is the preferred method of instantiating a DUT. The syntax for the two types of associations is shown below.

Named Instantiation

```
module_name instance_name (
.port_name_1 (tb_signal_name_1),
.port_name_2 (tb_signal_name_2)
);
```

Positional Instantiation

```
module_name instance_name (
tb_signal_name_1,
tb_signal_name_2
);
```

Given below is the example code for instantiating the GCD design using both named and positional associations. For readability, it is preferred that the signal names in the testbench are simply the port names with the suffix "_tb". For example, the port "**`Clk`**" has its corresponding signal in the testbench with the name "**`Clk_tb`**".

Named Instantiation

```
module ee201_GCD_tb_Part1;

// Declaring inputs
reg        Clk_tb;
reg        reset_tb;
reg        start_tb;
reg        ack_tb;
reg [7:0]  Ain_tb;
reg [7:0]  Bin_tb;

// Declaring outputs
wire [7:0] AB_GCD_tb;
wire       q_I_tb;
wire       q_Sub_tb;
wire       q_Mult_tb;
wire       q_Done_tb;

// Instantiating the DUT

ee201_GCD dut (
.Clk(Clk_tb),
.reset(reset_tb),
.start(start_tb),
.ack(ack_tb),
.Ain(Ain_tb),
.Bin(Bin_tb),
.AB_GCD(AB_GCD_tb),
.q_I(q_I_tb),
.q_Sub(q_Sub_tb),
.q_Mult(q_Mult_tb),
.q_Done(q_Done_tb)
);
```

Positional Instantiation

```
module ee201_GCD_tb_Part1;
```

I/O declarations are the same

```
// Instantiating the DUT

ee201_GCD dut (
 Clk_tb,
 reset_tb,
 start_tb,
 ack_tb,
 Ain_tb,
 Bin_tb,
 AB_GCD_tb,
 q_I_tb,
 q_Sub_tb,
 q_Mult_tb,
 q_Done_tb
);
```

5.3 Generating Clock

All sequential DUTs require a clock signal. To generate a clock signal, many different Verilog constructs can be used. Given below are two example constructs. Method 1 is preferred because the entire clock generation code is neatly encapsulated in one initial block.

Clock Generation Method # 1

```
initial
   begin: CLOCK_GENERATOR
    Clk_tb=0;
    forever
       begin
          #5 Clk_tb = ~ Clk_tb;
       end
   end
```

Clock Generation Method # 2

```
initial
   begin: CLOCK_INITIALIZATION
       Clk_tb = 0;
   end
always
   begin: CLOCK_GENERATOR
       #5 Clk_tb = ~ Clk_tb;
   end
```

5.4 Applying Stimulus and Timing Control

In addition to the clock, the testbench has to produce various other signals that are used as inputs to the DUT. These signals, also called “stimulus”, have to be applied with specific values at specific times. For instance, `reset` has to be asserted at the start of the simulation. Only after `reset` has been de-asserted, should other inputs be applied.

Before discussing the various ways to control the timing of a signal in the testbench, two rules for stimulus timing are worth noting:

- ***Rule # 1****:* All inputs should be applied with a small delay consequent to the clock edge. For example, the start signal should be asserted a little (say, 1 time unit) after the clock edge.
Note that a little after the current clock edge means long before the next edge. The “little after” is to satisfy the hold-time requirements of the flip-flops and the “long before” is for satisfying the setup time of the flip-flops and any next state logic.

- ***Rule # 2****:* All outputs should be read somewhere in the middle of the clock (or in the later half of the clock or towards the end of the clock) at which time they are expected to be valid and stable.

Verilog provides three ways to control timing of signals in the testbench.

Delay Control (#): The simplest way to control the timing of a signal is to use the delay control operator “#” followed by the time for which the value is held. This simple method was used in all of our testbenches up to this point. The disadvantage of using delay control, however, is that the timing cannot be defined relative to other activity in the system. Therefore, the exact timing of each stimulus signal has to be determined *a priori*. In a design like the GCD, the length of time spent on computing the results is dependent upon the input data values. In this case, tedious manual calculations of timing of the acknowledgment signal are required for each set of inputs. Therefore, in this (and subsequent) testbenches, it is highly recommended that delay control be used only for trivial timing control or where it is the only option (such as in the clock and reset signal generation).

Level Sensitive Control (wait): If a signal has to be asserted consequent to a DUT output going active, level sensitive timing control "wait" can be used. Consider the example of the `ack` signal in the GCD design. `ack` goes active consequent to the state machine entering the DONE state which is indicated by `q_done` going high.

Code	Explanation
`wait (q_done);`	wait for q_done to be high
`# 1;`	after a small delay
`ack = 1;`	assert ack

Since `ack` has to be asserted only after `q_done` has gone high, instead of computing the time when `q_done` goes high, the above code implements a simple wait for the signal `q_done` to go high and assert **`ack`** consequent to it with a small delay (in accordance with Rule # 1 mentioned above).

Event Control (@): Level sensitive control can be used to wait for signals to go high or low. However, certain inputs have to be applied consequent to a specific edge (the two may be interchangeable in some instances). To do so, event control operator "@" has to be used.

Consider the example of the start signal in the GCD design. Suppose, it was required that start be asserted two clock cycles after the state machine enters the INITIAL state. Following code can be used to accomplish this. Note: The level-sensitive wait statements `wait(Clk_tb)` in the wrong code below do not help here, as right after the `q_I` is a "1", `Clk_tb` is already high and the two `wait` statements pass without any waiting.

Code	wrong code	Explanation
`wait (q_I);`	`wait (q_I);`	wait for `q_I` to be high
`@ (posedge Clk_tb);`	`wait(Clk_tb);`	wait for one clock (after q_I goes high)
`@ (posedge Clk_tb);`	`wait(Clk_tb);`	wait for another clock
`# 1;`	`#1;`	after a small delay
`start = 1;`	`start = 1;`	assert start

5.5 String variable display in the waveform

The state vector displayed in binary (in the modelsim waveforms) perhaps does not let us quickly understand the sequence of states our design went through. We end up looking up the state coding and state assignments. It is easy to display the state as an ascii string variable in the modelsim waveforms.

```
reg [5*8:1] state_string; // 5-character string for symbolic display of state
always @(*)
      case ({q_I, q_Sub, q_Mult, q_Done}) // Note the concatenation operator {}
            4'b1000: state_string = "q_I   ";
            4'b0100: state_string = "q_Sub ";
            4'b0010: state_string = "q_Mult";
            4'b0001: state_string = "q_Done";
      endcase
```

Extract from the GCD testbench

5.6 Displaying Outputs as Text

Reading the value of signals using the waveform is often times extremely useful in debugging the design. The waveforms allow designers to observe not only the values of the signals but also their relative timing (e.g., values "before" the clock edge and "after" the clock edge). However, once the designer has reasonable confidence in the correct timing of the design, it may be desirable to observe the value of a few signals in text form. For example, in a rigorous test of the GCD design where several sets of inputs are applied, it may be easier to read the final GCD values for each input set in a text form than to trace the final output values in a very long waveform.

Verilog provides several system tasks to display values of a signal. The simplest of them is the **$display** task, which can be used to print the values of signals in various formats. In Modelsim, these values are printed in the Transcript window (they appear as blue-color text). The syntax and an example of $display usage is given below.

Syntax	Example
`$display ( "formatted_string", signal_list );`	`$display ( "Ain=%d, Bin=%d", Ain, Bin );`

5.7 Useful system tasks

• **$display, $strobe, $monitor**: share the same syntax, and display values as text on the screen during simulation. However, **$display** and **$strobe** display once every time they are executed, whereas **$monitor** displays every time one of its parameters changes. The difference between **$display** and **$strobe** is that **$strobe** displays the parameters at the very end of the current simulation time unit (after all the delta_Ts are over for the current simulation time). The format string is like that in C/C++, and may contain format characters. Format characters include **%d** (decimal), **%h** (hexadecimal), **%b** (binary), **%c** (character), **%s** (string) and **%t** (time). Append **b**, **h**, **o** to the task name to change default format to binary, octal or hexadecimal.

Example:
```
$displayh (var); // displayed in hexadecimal
$monitor ("At time=%t, d=%h", $time, var);
```

• **$write**: the difference between `$write` (, `$writeb`, `$writeo`, `$writeh`) and the `$display` (, `$displayb`, `$displayo`, `$displayh`) is the absence (for `$write`) or presence (for `$display`) of the carriage-return and new-line characters at the end of the current line. If you want to "compose" a long line, whose segments depend on variables in the testbench, then you can use `$write` for all segments except for the las segment for which you would use `$display`.

• **$time, $stime, $realtime**: these system functions return the current simulation time as a 64-bit integer, a 32-bit integer, and a real number, respectively. Note: these are called *functions* and not *tasks* as they return values.

• **$reset, $stop, $finish**: **$reset** resets the simulation back to time 0; **$stop** halts the simulator and puts it in the interactive mode where the user can enter commands; **$finish** exits the simulator back to the operating system.

• **$deposit**: sets a net to a particular value.

Syntax: `$deposit (net_name, value);`

• **$random**: this function generates (and returns) a random integer every time it is called. If the random sequence is to be *repeatabl*e, the first time one invokes random give it a numerical argument (a seed). Otherwise the seed is derived from the computer clock.

Syntax: `var = $random[(integer_seed)];` // integer_seed is an optional argument

5.8 Using Tasks

Very often in testbenches, the same piece of code is repeated several times. This is often the case when different sets of inputs are applied. In the GCD design, for example, for every pair of `Ain` and `Bin` values, the same piece of code is repeated for applying `start` and `ack`. To simplify the coding of the testbench, such re-usable code can be put in a user-defined "task" which can be called with an appropriate set of input arguments. Task can contain any type of timing control statement (#, wait or @) and system tasks (e.g. `$display`). In effect, virtually any piece of code that can be written in-line (meaning without containing it in a task) can be put in a task for re-usability. A task can have zero or more input and/or output (and/or inout) arguments but it does not *return* values like a function. The body of the task must be enclosed in `begin-end`. Tasks are defined outside procedural blocks (i.e., outside `initial` and `always` blocks) but can be called from within a procedural block. Finally, while in this exercise tasks are proposed only for testbenches, they can be used in synthesizable code provided they do not contain non-synthesizable constructs (such as timing controls) in them. An example task that asserts acknowledge signal (**`ack`**) in the GCD design at an appropriate time is shown below (Note that this task does not specify any arguments). Note that variables such as `Ack_tb`, which are visible at the line of the task call, are also visible to the task.

```
task SEND_ACKNOWLEDGE;          // semicolon needed
   begin                        // begin-end necessary
      // Send an acknowledge
      Ack_tb = 1;
      @(posedge Clk_tb);
      # 1;
      Ack_tb = 0;
      @(posedge Clk_tb);
      # 1;
   end                          // begin-end necessary
endtask                         // no semicolon
```

5.9 FOR loop

When using a FOR loop in Verilog, the loop index must be declared as an integer. **FOR** loops can be used inside an always or initial block. An example of a for loop (in a testbench) is:

```
integer i;
initial
   for (i=0; i<10; i = i+1)
      begin
      @(posedge Clk)
      #1;
      A_reg = i;
      end
```

5.10 File I/O in Verilog

Thorough testbenches need to process significant amounts of relevant data to ensure the correctness of a design. Often that data may be complex to produce and/or to analyze. In such cases, the use of domain-specific or otherwise appropriate tools assists the designer in producing useful test data and analyzing the output of a system. In order to incorporate other tools into a testing strategy, we need a way of transferring the necessary data. File I/O operations allow us to use standard files to do this. Other software tools used within specific domains can be useful for creating test stimulus (i.e. Matlab). Verilog-2001 provides extensive file I/O capability directly in the Verilog language. The following operations are supported:

- C or C++ type file operations (example c = $fgetc (fd) returns a -1 in c if end of file is encountered).
- Reading characters from file from a fixed location.
- Reading a formatted lines in file.
- Writing a formatted lines into file.

To open a file:

```
file_descriptor_ID = $fopen("filename","r"); // r = For reading
file_descriptor_ID = $fopen("filename","w"); // w = For writing
```

Note: `file_descriptor_ID` is the logical name (example: `my_gcd_file`) you use in your code in the system tasks such as `$fdisplay` and `$fwrite` whereas the `filename` is the physical name of the file on your hard-disc (example: `ee201_gcd_Part3_output.txt`).

To close a file

```
$fclose(file_descriptor_ID);
```

To read variables from a file

```
$fscanf(file_descriptor_ID, "formatted_string", [arguments]);
```

To write to a file

```
$fdisplay(file_descriptor_ID, "formatted_string"[, arguments]);
$fwrite(file_descriptor_ID, "formatted_string"[, arguments]);
```

`$fdisplay` includes a carriage-return and a new-line character at the end of the current line where as `$fwrite` does not.

Verilog 2001 has several tasks to operate on strings: `$sscanf`, `$swrite`, .. (similar to `$fscanf`, `$fwrite`, ..).

Please check the manuals or online help for more details on file I/O operations in Verilog.
Also please read the testbenches with file I/O provided to you for other labs/exercises.

An extract from the `copy_array_to_array_tb.v`

```
reg signed [3:0] signed_Ns_of_J; // To display elements of N as signed numbers
```

```
initial
   begin
    output_file = $fopen ("output.txt", "w");
   end
```

```
task display_M_and_N_arrays;
  begin
    $display("\n The M and the N arrays \n");
    $fdisplay(output_file, "\n The M and the N arrays \n");
    for (II_JJ=0; II_JJ<=II_JJ_max; II_JJ=II_JJ+1) // for all the rows in the matrices
        begin // for each row (there are p items in a row)
            signed_Ns_of_J = Ns_of_J_array[II_JJ];
            string = "\n";
            $sformat(string, "%s\t%d", string, II_JJ);  // Print index II_JJ in the string
            $sformat(string, "%s\t%d", string, Ms_of_I_array[II_JJ]);   // M[I] in decimal
            $sformat(string, "%s\t%b", string, Ms_of_I_array[II_JJ]);   // M[I] in binary
            $sformat(string, "%s\t%b", string, Ns_of_J_array[II_JJ]);   // N[J] in binary
            $sformat(string, "%s\t%d", string, signed_Ns_of_J);         // N[J] in decimal
            $display("%s", string);
            $fwrite(output_file, "%s\n", string);
        end
    $display ("\n Clocks_taken = %d . \n", Clocks_taken);
    $fdisplay (output_file, "\n Clocks_taken = %d . \n", Clocks_taken);
    $fclose(output_file);
  end
endtask
```

The above task produced display on the side:
"Growing" a string as shown in the line marked with an arrow above:

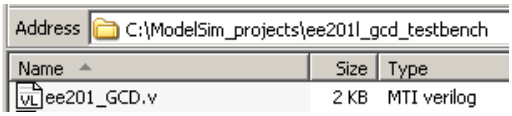
```
// the previous value of the "string" is
// concatenated with the the array item
// (represented in decimal format)
// to form the new string
```

```
# The M and the N arrays
#
#
#       0       2       0010    1001    -7
#
#       1       5       0101    1010    -6
#
#       9       15      1111    0111    7
#
# Clocks_taken =      13 .
#

/SIM 18>
```

6. Procedure

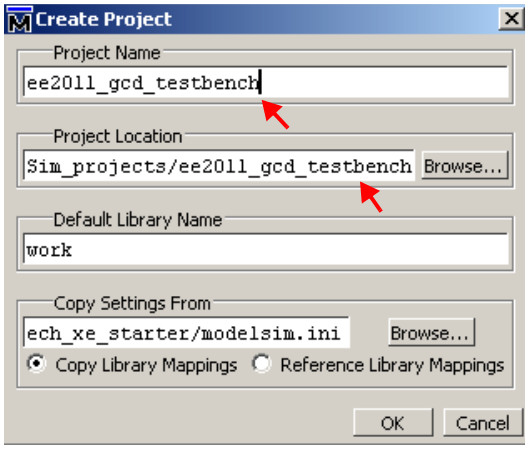
Before you can start writing the testbench, you must create a new Modelsim project and include the GCD design and the testbench file to it.

6.1 Create a project directory "`ee201l_gcd_testbench`" under `C:\ModelSim_projects`.

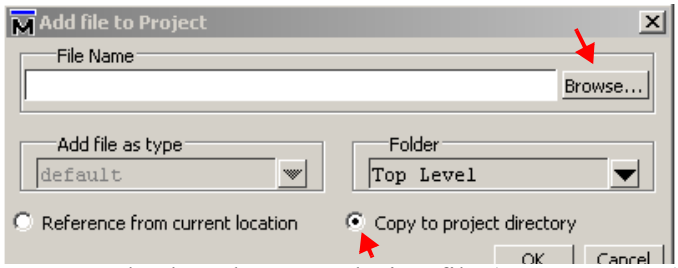


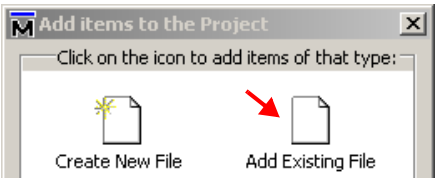
6.2 Copy the GCD design file (`ee201_gcd.v`) from previous lab to this directory. Your TA may provide you with a complete working GCD design if you could not finish your previous lab.

6.3 Open Modelsim and create a new project by clicking: `File -> New -> Project`. In the `Create Project` dialogue box, type the `Project Name: ee201l_gcd_testbench` and choose the project directory that you created above (`C:\ModelSim_projects\ee201l_gcd_testbench`) as the `Project Location` . Leave the other options unchanged. Click `OK` to continue.

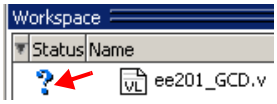


6.4 Click on Add Existing File .

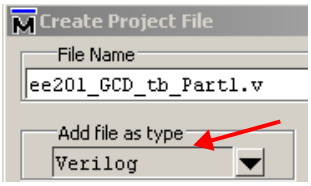




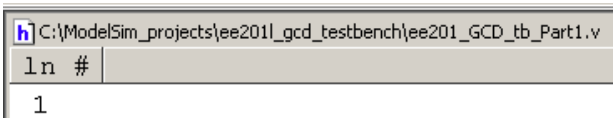
Browse and select the GCD design file (`ee201_gcd.v`). This adds our GCD design file to the project with a ? sign in front of it (indicating that is has not been compiled yet). Do not close the `Add Items to the Project` dialogue box yet.



6.5 Next, select Create New File in the Add items to the Project dialogue box. When prompted, give it the name `ee201_gcd_tb_part1.v` and choose the type as `Verilog`. This new (blank) testbench file gets added to the project. ? ee201_GCD_tb_Part1.v



6.6 Double click on the `ee201_gcd_tb_part1.v` file in the Workspace to open it. This will open the file in a text editor (the file is blank, of course). Now, its time to



write the testbench code!

If you prefer you may use Notepad++ 

for editing Verilog files.

Part 1: Writing a simple testbench

In this first part of the lab, you will write a simple testbench that applies two sets of inputs to the GCD core design. Use the example code snippets provided in this handout for help.

6.7 In the blank testbench file (`ee201_gcd_tb_part1.v`) add the code for: (1) defining the time units and precision, (2) module definition of the testbench module (along with the corresponding `endmodule`), and (3) instantiation of the GCD design. Use the sample code segments provided in Section 5 of this handout.

6.8 Using the recommended methods for coding the clock generator, write the code that generates `Clk_tb`.

6.9 Using **#**, **wait** and **@** to control the timing of signals, write the code that applies the stimulus to the GCD design. Use two sets of values for `Ain` and `Bin`: (24 & 36) and (5 & 15). The code you write must generate `reset_tb`, `Ain_tb`, `Bin_tb`, `start_tb` and `ack_tb` with the timing shown in the waveform (Part I, attached). Make sure that the timing of each signal is defined relative to the activity in the system. Use delay control only to add a small delay after a signal goes high or after the clock edge.

6.10 Add code to print the values of `Ain`, `Bin` and GCD when the computation of GCD for each set of inputs is completed.

```
# Ain: 36 Bin: 24 GCD: 12
# It took        8 clock(s) to compute the GCD
```

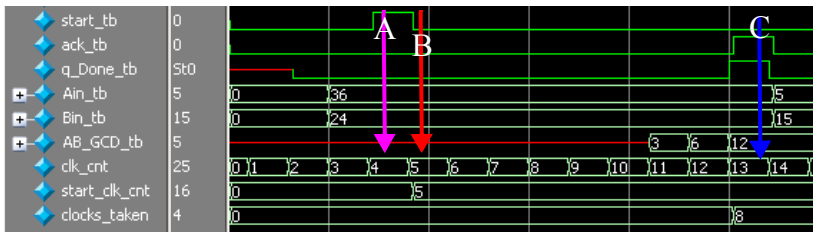
6.11 Finally, write code for a free-running clock counter. This counter helps in computing the number of clocks it takes the state machine to compute the GCD of a given input set. Look at the following waveform for more ideas about when to record `start_clk_count` and how to compute `clocks_taken`. You may want to declare three data objects of integer data type as shown below:

```
integer  clk_cnt, start_Clk_cnt, clocks_taken;
```

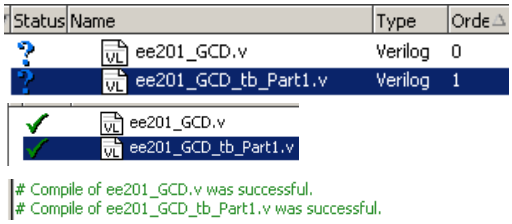
Clocks taken: C - B = 13 -5 = 8

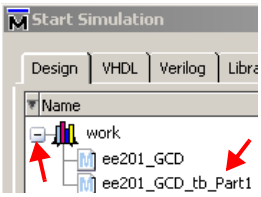
start_clk_cnt = clk_cnt;

clocks_taken = clk_cnt - start_clk_cnt;

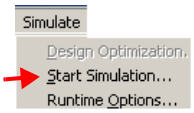


6.12 Once the code is complete, compile the code by selecting " Compile -> Compile All . The compilation order can be changed by Compile -> Compile Order... . Note: make sure that the order of compilation is from innermost to outermost. If no compilation errors are reported, proceed to start the simulation.

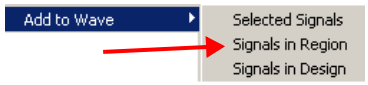




6.13 To begin simulating the design, select `Simulate -> Start Simulation`. Click the "+" next to "`work`" in the `Start Simulation` dialog. This will reveal your compiled designs. Select "`ee201_GCD_tb_Part1`" and click "`OK`".



6.14 Add all top-level signals to the waveform window. To do this, right click on any object, select Add to Wave | Signals in Region . If the waveform window is not available, select `View -> Wave` to make it viewable.



6.15 Type "`log -r *`" VSIM 3> log -r * to tell ModelSim to log all signals whether they are currently added to the wave display or not. To run simulations for a fixed length of (simulation) time (250ns in this example), type "`run 250ns`" at the `VSIM>` prompt in the Transcript window. Simulation data will appear as waveform and values of `Ain`, `Bin`, and GCD should appear as text in the transcript window. The transcript window should show the following:

```
VSIM4> run 250ns
# Ain: 36 Bin: 24 GCD: 12
# It took           8 clock(s) to compute the GCD
# Ain:  5 Bin: 15 GCD:  5
# It took           4 clock(s) to compute the GCD
```

6.16 In the waveform, you may want to change the radix of signals, `Ain_tb`, `Bin_tb`, and `AB_GCD_tb`, to unsigned. Practice using the zoom and cursor controls shown on the side.

Note: steps 6.12 through 6.16 should be repeated every time a change is made to the code. You need to end the current simulation by Simulate -> End Simulation. Or do step 6.12 as needed and type "**`restart -f`**" to restart the simulation. This will avoid the repetition of 6.13 to 6.16.

6.17 .do files in ModelSim: While "**`restart -f`**" helps multiple runs in the same "sitting", if we want to repeat the tests after a couple of days, we need a method to automate steps 6.12 to 6.15. The modelsim .do file is a "batch" file to execute a series of commands one after another. Basically, we want to compile the verilog codes, start simulation, add the desired signals to the waveform, change some signal display radices, simulate for certain length of time.

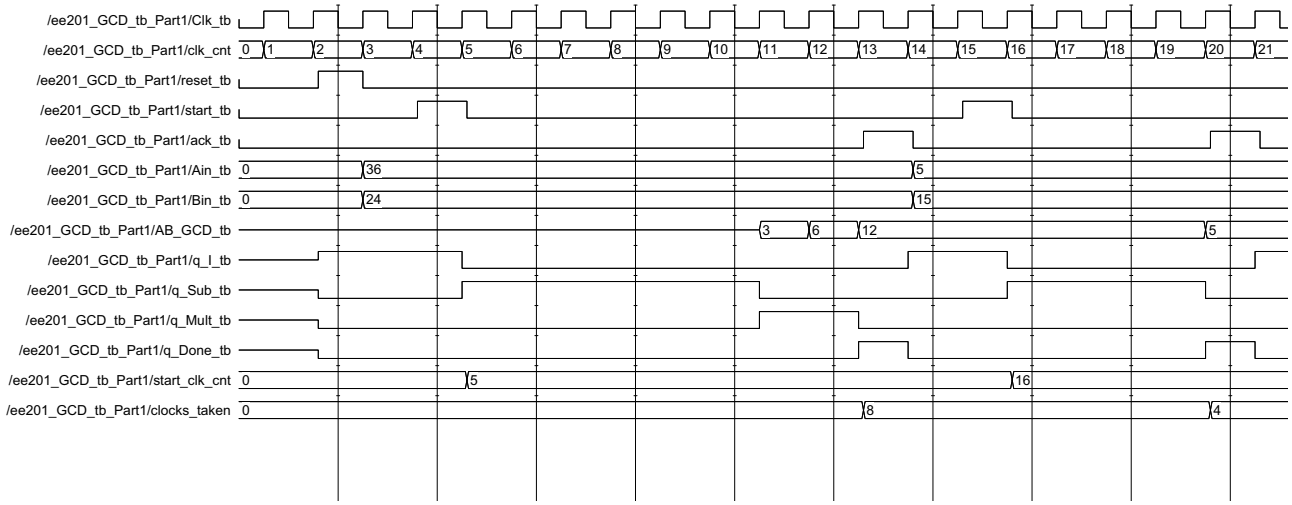
```
# ee201_gcd_tb_Part1.do
vlib work
vlog +acc  "ee201_gcd.v"
vlog +acc  "ee201_gcd_tb_Part1.v"
vsim -t 1ps -lib work ee201_gcd_tb_Part1
view objects
view wave
do {ee201_gcd_tb_Part1_wave.do}
log -r *
run 300ns
```

```
Some explanation of commands:
vlib : Specifies a project directory
vlog : verilog compiler invocation
(vcom for VHDL compilation)
vsim : simulator invocation
view : specifies a tool window for view
do : executes a .do file such as wave.do
log -r * : logs all signal activity
            whether you added or not
              some signals
run 300ns
```

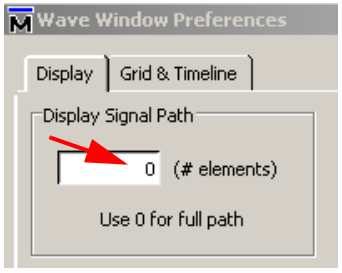
"quit -sim" lets you quit from simulation mode (and receive vsim> prompt), where as "quit" will quit from ModelSim altogether. So after you revise some of your design or testbench, you should "quit -sim" to ModelSim> prompt, and then run the DO file you just composed. The goal is to avoid bring up and down the ModelSim.

Go to the waveform window display (displaying the needed signals in the desired order and radices), and save file: File => Save. Accept the default name `wave.do` (or choose to name it something like `ee201l_gcd_tb_Part1_wave.do`). It creates `C:/ModelSim_projects/ee201l_gcd_testbench/ee201l_gcd_tb_Part1_wave.do` Then create a text file, with name, say, `ee201l_gcd_tb_Part1.do` using word pad or note pad to include the commands shown in the inset. Make sure that the extension is not changed to .txt. If needed, you can rename the file. Next time, you want to simulate, you double-click on ee201l_gcd_testbench.mpf and at the **`VSIM>`** prompt, you type `do {ee201l_gcd_tb_Part1.do}` or just `do ee201l_gcd_tb_Part1.do`

Waveform for Part 1



In the above waveform, you see signal names shown (in the left-most pane) with complete hierarchical path names. This is the default behavior. If you want simple signal name with no hierarchical path, then go to **`Tools`** menu on the waveform window and open **`Window preferences`** .
Change the default 0 to 1 to get simple names for signals.



Part 2: User-defined Tasks in Testbenches

In this second part of the lab, you will use a user-defined task to re-use the code that applies the stimulus. Furthermore, you will write the testbench code necessary to test the GCD state machine exhaustively for all values of `Ain` and `Bin` between 2 and 63. Like in Part 1, your testbench should report the two inputs and the number of cycles it took the state machine to compute the GCD for each input pair.

6.18 Make a copy of the testbench file `ee201_GCD_tb_Part1.v` and name it as `ee201_GCD_tb_Part2.v` . Change the module name to `ee201_GCD_tb_Part2` .

6.19 Start coding a task **`APPLY_STIMULUS.`**

```
task APPLY_STIMULUS;
   input [7:0] Ain_value;
   input [7:0] Bin_value;
   begin
```

Move the replicated code for applying the stimulus to this task. The two inputs of the task will be the values of the formal parameters **`Ain_value`** and **`Bin_value`** and the task will apply these stimulus along with `start` and `ack` signals with necessary timing control.

6.20 First test your task by invoking the task for fixed pairs of values.

```
APPLY_STIMULUS (36,24);
APPLY_STIMULUS (5,15);
```

6.21 After making sure that the task works properly for the above pair, write the code to apply the inputs **`Ain`** and **`Bin`** over the range of 2...63. A nested for-loop (a for-loop within another for-loop) is one of the simplest ways to accomplish this. An example of a nested for loop is given below:

```
integer  hr, min;

for (hr = 0; hr <= 23; hr = hr+1)
     for (min = 0; min <= 59; min = min+1)
       // do whatever needs to be done with hr & min
```

6.22 The output (in the transcript window) of your testbench should look like this:

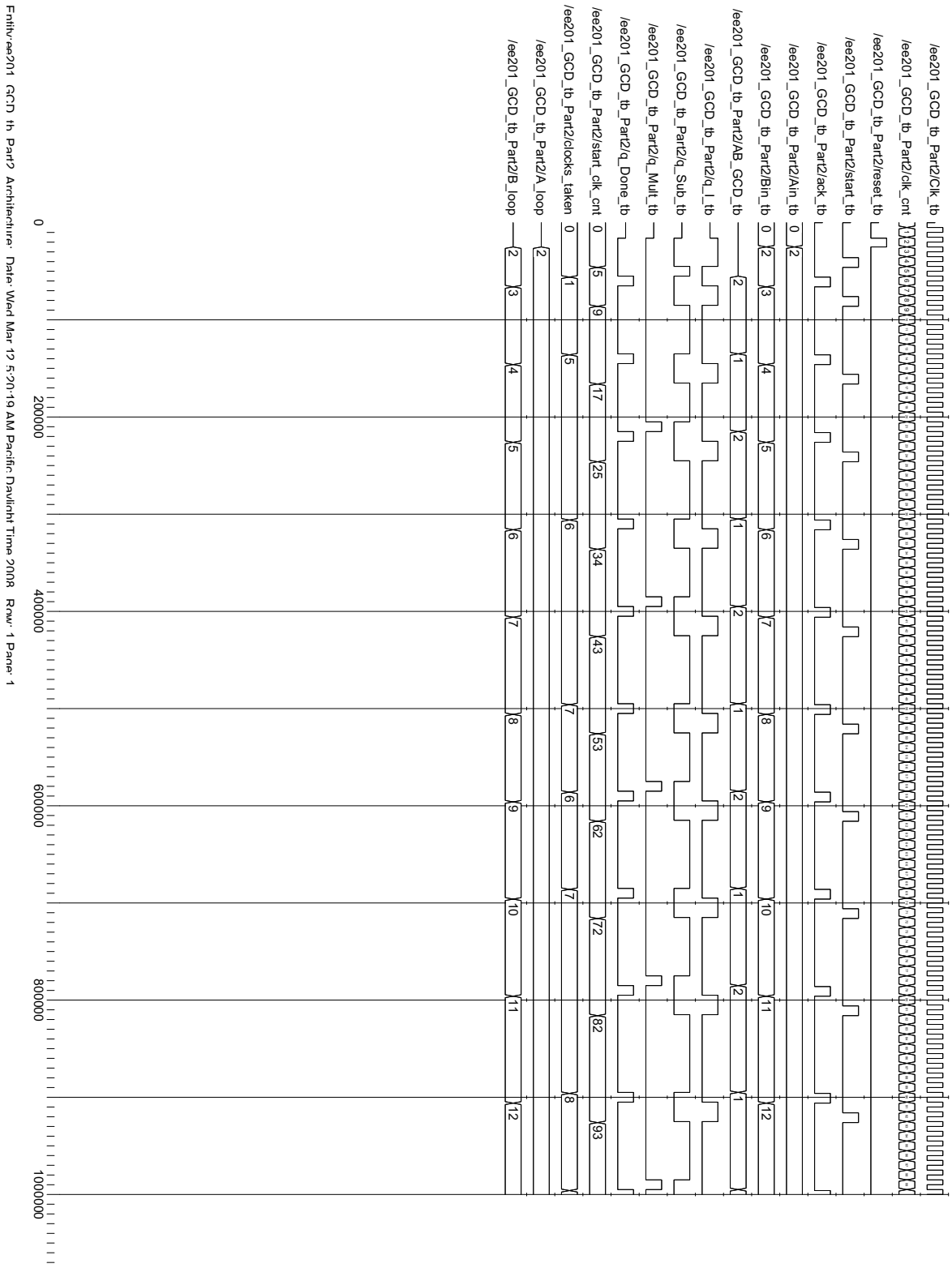
```
# Ain:  2 Bin:  2 GCD:  2
# It took           1 clock(s) to compute the GCD
# Ain:  2 Bin:  3 GCD:  1
# It took           5 clock(s) to compute the GCD
# Ain:  2 Bin:  4 GCD:  2
# It took           5 clock(s) to compute the GCD
# Ain:  2 Bin:  5 GCD:  1
# It took           6 clock(s) to compute the GCD
```

Attached is the waveform view of the first 1us of simulation.

Part 3: File Output

In this third part of the lab, you are required to output the results (for **`Ain`** and **`Bin`** over the range of 2...63) to a text file named "`ee201_gcd_Part3_output.txt`" instead of the Transcript window. The format of the text is similar to Part 2. Make a copy of the testbench file `ee201_GCD_tb_Part2.v` and name it as `ee201_GCD_tb_Part3.v` . Change the module name to `ee201_GCD_tb_Part3` .

Waveform Part 2



7. Lab Report:

Name:_____	Date: _____
Lab Session: _____	TA's Signature: _____

For TAs: 3-part Implementation (15+20+15): _____ Report (out of 50): ___
Comments:

Q 7. 1: Submit online the following 4 Verilog testbench files and one text output file.
`ee201_gcd_tb_Part1.v, ee201_gcd_tb_Part2.v, ee201_gcd_tb_Part3.v, ee201_gcd_Part3_output.txt, names.txt`
Submit hardcopy (paper) for waveform from part 1.

Q 7. 2: (5 pts) What should the signals that connect to the inputs and outputs of the DUT be declared as in the testbench (check one or more):

a. Inputs to DUT as wire, outputs as reg b. Inputs as reg, output as wires

c. Both as wires d. Both as reg

Q 7. 3: (2 pts) Which of the two methods of instantiating the DUT is preferred?

positional association named association

Q 7. 4: (5 pts) What is the frequency of the clock defined in the clock generator example codes in sec. 5.3) in this handout? Use the time scale definition from Sec. 5.1

Q 7. 5: (10 pts) Suppose that you are asked to generate a clock of period 30ns but the duty cycle is 33.33% (one-third). Write the corresponding Verilog code.

Q 7. 6: (3 pts) Do you instantiate the DUT (or UUT) in the `initial` procedural block or in the `always` procedural block? Miss Bruin says that it shall be in the `initial` procedural block as it should be instantiated only once.

(a) `initial` procedural block (b) `always` procedural block (c) neither

Q 7. 7: (4 + 6 = 10 pts) Compare the following codes. The intent is to keep the `start` signal active for 1 clock, wait for two clocks, activate the **`ack`** for 1 clock. Which of the following work? (4 pts) Select *one* or *more* as appropriate.

a. Code A b. Code B c. Code C

Hint: One of the following is level-sensitive and the other is edge-sensitive.

```
@(posedge clk);     wait (clk);
```

If you're not sure, just simulate the codes and see what happens!

Code A

```
@(posedge clk);
#1;
start <= 1;

@(posedge clk);
#1;
start <= 0;

@(posedge clk);
@(posedge clk);
#1;
ack <= 1;

@(posedge clk);
#1;
ack <= 0;
```

Code B

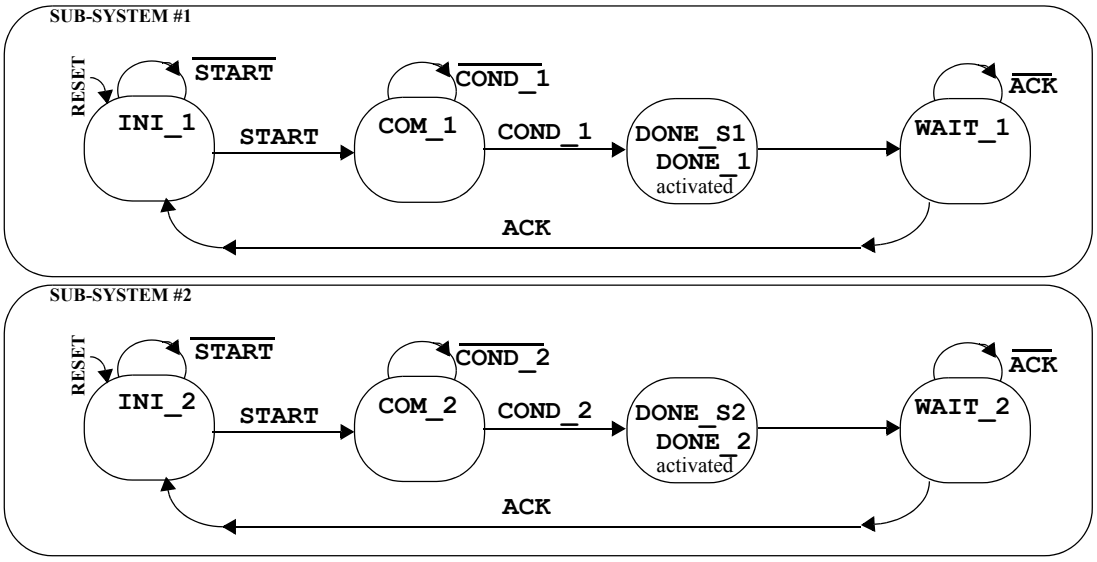
```
wait (clk);
#1;
start <= 1;

wait (clk);
#1;
start <= 0;

wait (clk);
wait (clk);
#1;
ack <= 1;

wait (clk);
#1;
ack <= 0;
```

Code C

```
wait (~clk);
wait (clk);
#1;
start <= 1;

wait (~clk);
wait (clk);
#1;
start <= 0;

wait (~clk);
wait (clk);
wait (~clk);
wait (clk);
#1;
ack <= 1;

wait (~clk);
wait (clk);
#1;
ack <= 0;
```

Explain your reasoning/analysis of the above 3 code segments. (6 pts)

Q 7. 8: (15 pts) Suppose the core design (some other core design other than the GCD) has two sub-systems each having their own state machines. However, they share the same **START** and **ACK** signals. They both start when **START** is given (by the testbench or the top design). Each can take different number of clocks and the number of clocks is data dependent like in GCD. So sometimes SS1 (sub-system #1) can finish first and sometimes SS2. Or sometimes both may finish in the same clock! When a sub-system finishes, it produces a done signal just for 1 clock. SS1 produces a one-clock wide **DONE_1** and SS2 produces one-clock wide **DONE_2**.



Your job in the testbench, so far as sending **ACK** is concerned, is to

-- wait for both **DONE_1** and **DONE_2** to arrive
-- wait for one clock and then
-- issue a one-clock wide acknowledge **ACK** and
-- of course get ready for the next test iteration.

Write a code segment to do the above to send **ACK** correctly Include declarations of any new variables (objects). Pay attention to *blocking* and *non blocking* assignments.