Introduction to TCL Scripting for EEL4712

Bo Purtell

Alternate Title: Being able to interact with Modelsim (and MIFs) in a non-painful way

Objective:

Modelsim can be started outside of Quartus and used with a bunch of hand-configuration. But this requires clicking several steps per use:

1. Compilation (or recompile): right click -> compileall
2. Simulation (or restart): click simulation -> start simulation
3. Adding waves: click object window -> click + shift click first and last signals and drag into the Waveviewer
4. Setting radixes: click + shift click relevant range -> right click -> radix -> appropriate radix
5. Running the simulation: click simulate -> run -> run all
6. Wave window fitting: right click -> zoom full | alternatively just press 'F' while the Waveviewer is selected
7. Repeat

By utilizing Modelsim's built-in scripting language you can effortlessly turn all these steps into (1) single command run after a little configuration.

GOTO page 10 if you already have a basic understanding but want to use Modelsim scripting while in a Quartus-demonstrated instance of Modelsim.

Logistics:

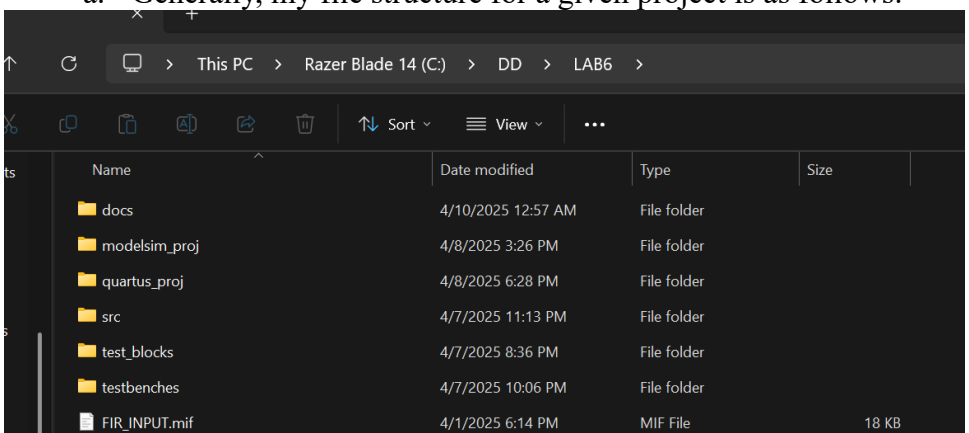
Requires a small bit of setup. Differs between (1) pure Modelsim projects and (2) Modelsim instances opened **via** Quartus.

Pure Modelsim Projects (Much more streamlined)

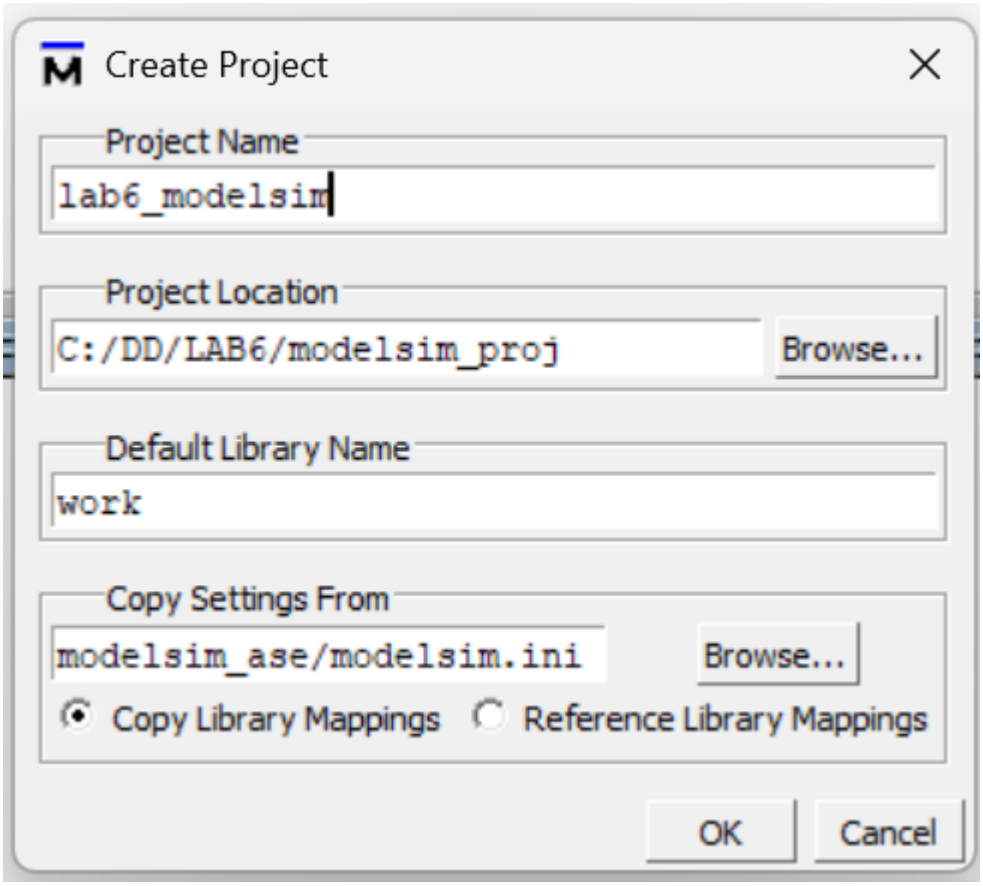
Step 1 – Make, Compile, Simulate, and Configure a Modelsim Instance

(Compile and set up a simulation exactly as you have done for every previous instance)

1. Make a new Modelsim Project
    a. Generally, my file structure for a given project is as follows:

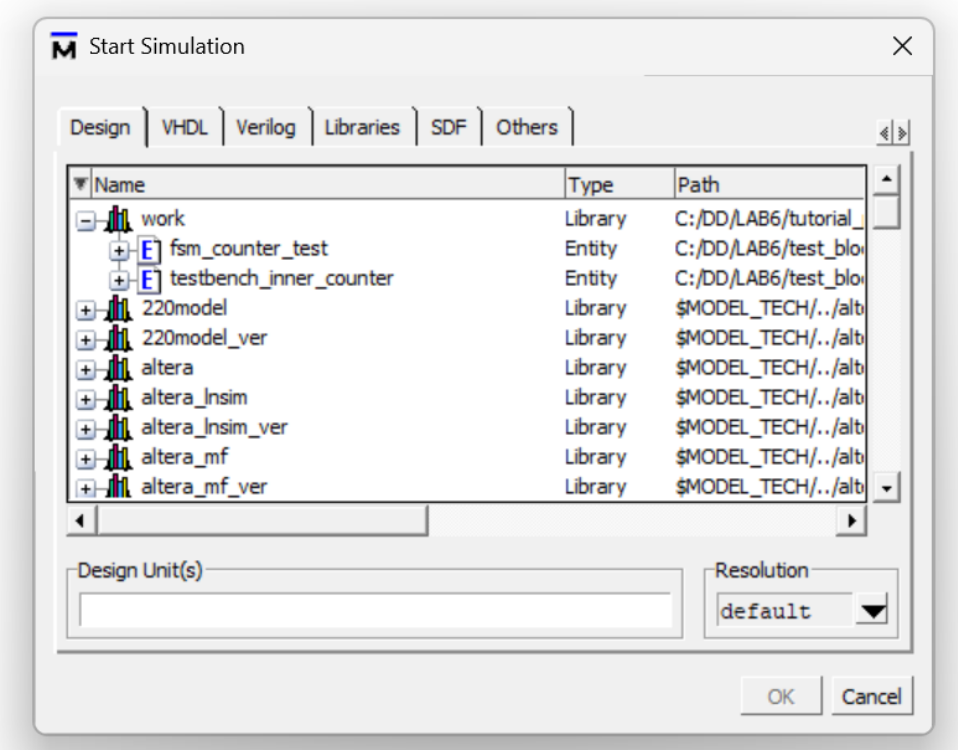


b. “modelsim_proj” and “quartus_proj” store the project files for the relevant tools specific to each lab.
c. In Modelsim, in the top-left corner select “File” -> “New” -> “Project”. This will open a Window that looks something like this. Note: If you have a previous project open it may ask to close that project. This is fine.
d. Now select “Browse…” and select the folder that you want as your project.
e. Give the project a name and leave everything else as default.
f. Hit “OK”.

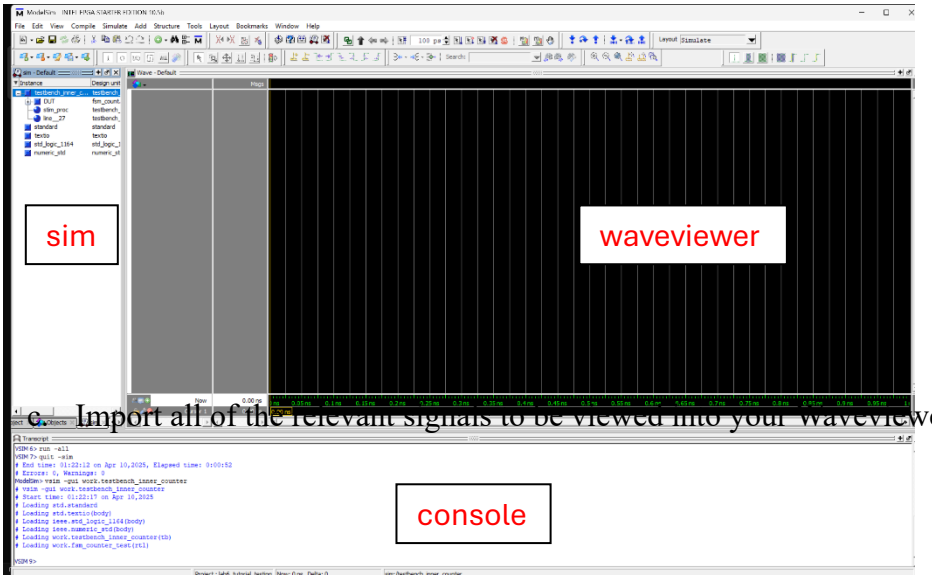


g. Modelsim will now ask to “Add Items to the Project”, hit “Add Existing File” and add browse to your VHDL code, select it, and hit “OK”. We trust you’ve figured out how to add files to a project by now.
    i. NOTE: adding files can be done later via right-clicking the project window and selecting “Add to Project”
h. The project should now contain all of the relevant VHDL design files and the testbenches (in VHDL or SystemVerilog).

2. Compile the project
    a. Right-click the project window and select “Compile” and “Compile All”.
        i. NOTE: You may have to resolve the compile order if errors show up.
        ii. To do this right-click the project window and select “Compile” and “Compile Order”.
        iii. Select “Auto Generate”.
3. Run a Simulation
    a. Select “Simulate”. A window should pop up.

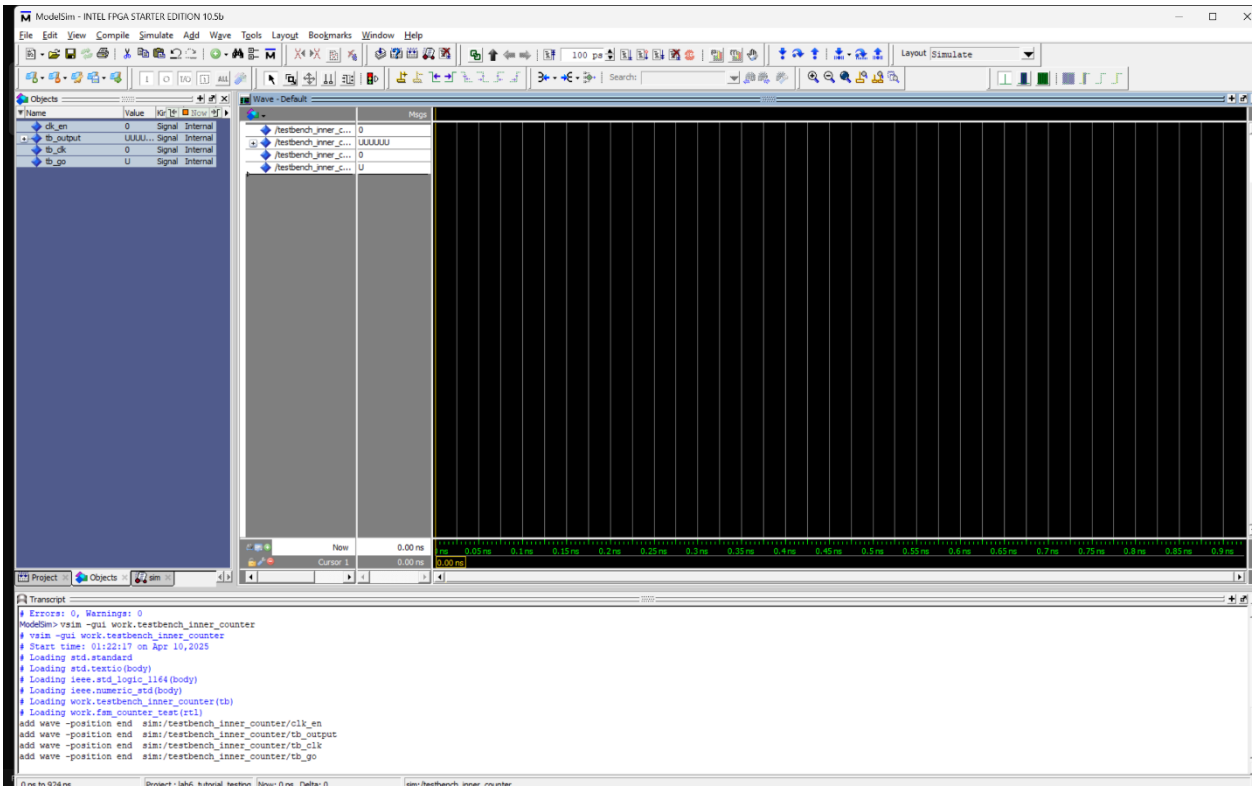


    b. Select “work” and the entity name of the your testbench. This should open the Waveviewer, Sim, and Console.



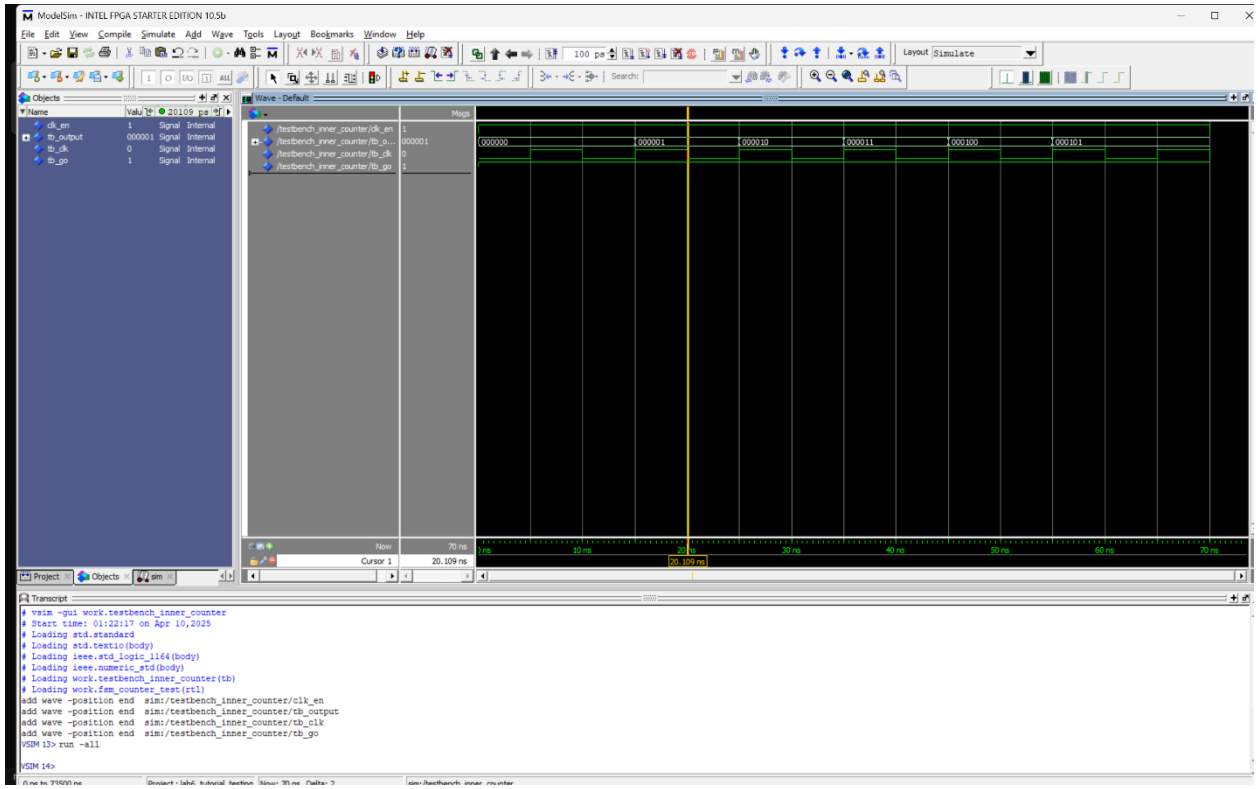
    c. Import all of the relevant signals to be viewed into your Waveviewer.

i. Switch to the “Objects” window in the lower left, select all of the signals you wish to view by left-clicking one, and then shift-left clicking another to select a range.
ii. Drag these signals into the Waveviewer.

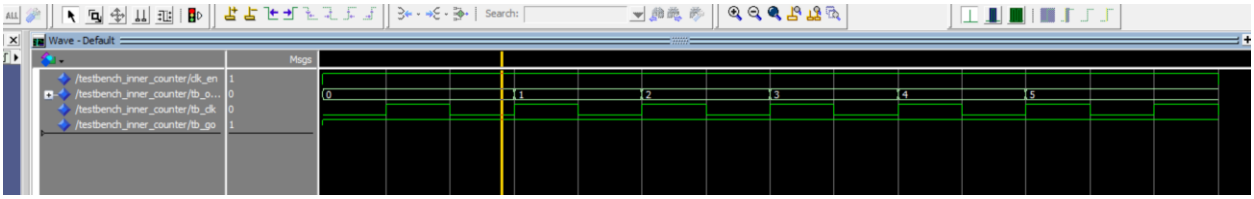


d. Your window should look something similar to this.
    i. NOTE: if you cannot see a certain window such as the console, select “View” and then select whichever window you wish to see.
e. Run the simulation by selecting “Simulate” from the top toolbar and then “Run” and “Run -all”.
f. IMPORTANT: Size your window by either selecting somewhere in the Waveviewer and pressing ‘i’ to zoom in or ‘o’ to zoom out.
    i. Alternatively press ‘f’ to zoom full.

g. Your screen should look similar.



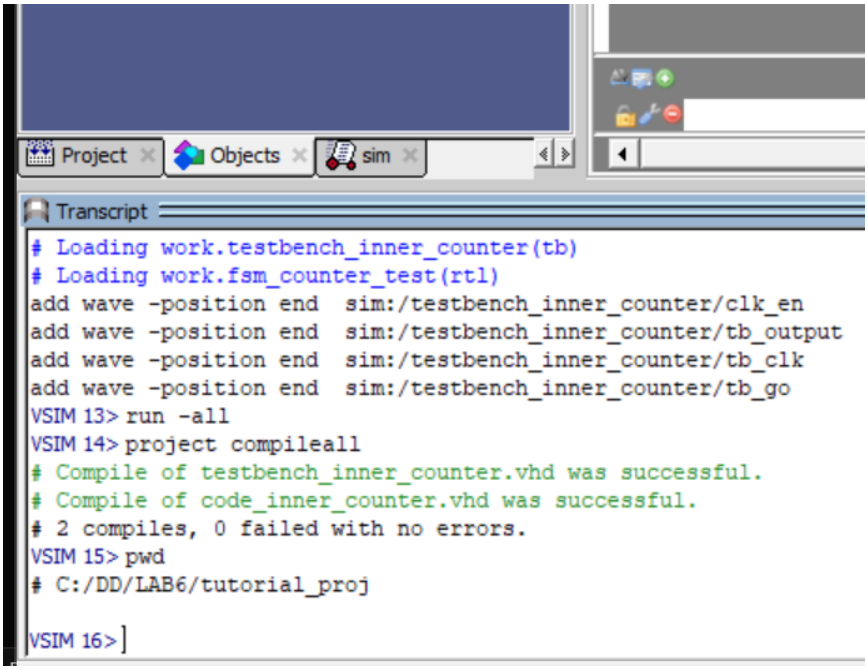
4. Change a radix to save for later by right-clicking one of the signals in the signal list of the simulation (the gray column). Select “Radix” and “Unsigned” for this example.



5. Your project has been created, you’ve run a simulation, and you’ve customized a piece of the simulation that you wish to see again that you don’t have to reconfigure. Keep going to see how to replicate this quickly using TCL scripting.
6. Do not close or change anything just yet and move to the next step.

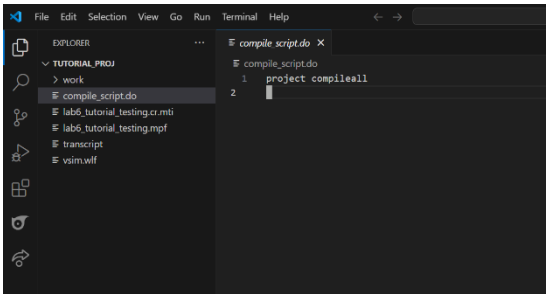
Step 2 : Setup a Basic Compile Script

1. Modelsim was originally a Comand-Line Interface tool (CLI) that had a GUI built on top of it. We can take advantage of this to streamline our experience with it.
2. While selected into the console window from previous figures, tryout a few commands
    a. Type “pwd” (print working directory) to see where our Modelsim console thinks we are.
    b. Type “ls” (list directory) to see what else is in the current directory.
    c. Type “project compileall” to compile the whole project again. We will make extensive use of this in a bit.
3. Figure out where our Modelsim console is. To do this, refer to the “pwd” command in the previous step. Save this location for later. (If you’ve setup a dedicated Modelsim project for a specific lab for example this will be the head project folder such as “modelsim_proj” in my example).

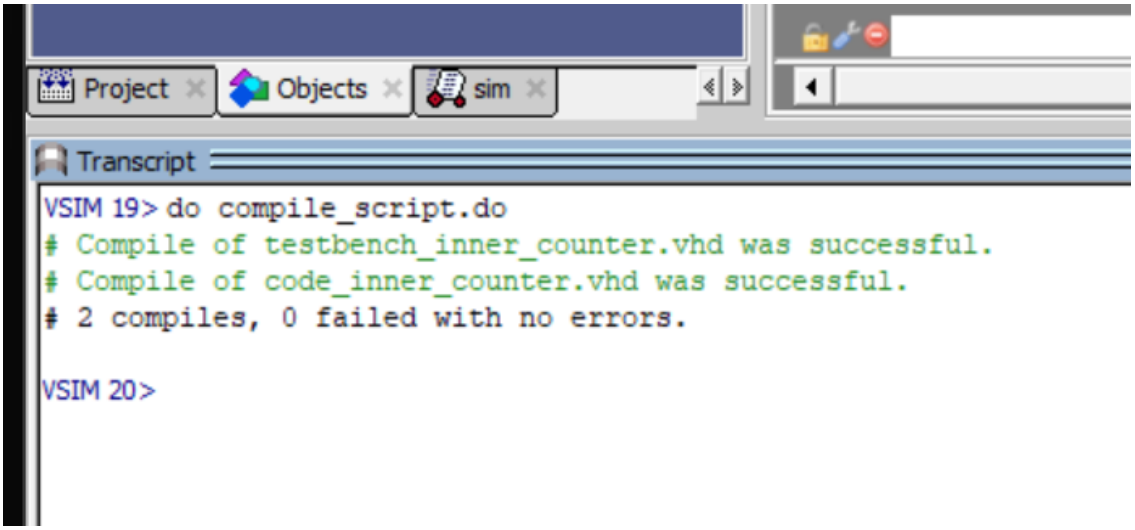


4. We can see here that our working directory is “C:/DD/LAB6/tutorial_proj”.
5. Write a compile script.
    a. In your favorite editor, make a new script and call it something like “compile_script.do”. (You should save it with this extension .do to ensure the console can interpret it correctly).
    b. Inside of this script put a single line “project compileall”.
    c. Save the script to the directory we found above (……./LAB6/tutorial_proj)

6. Here is an example of the script opened in a VSCode window.

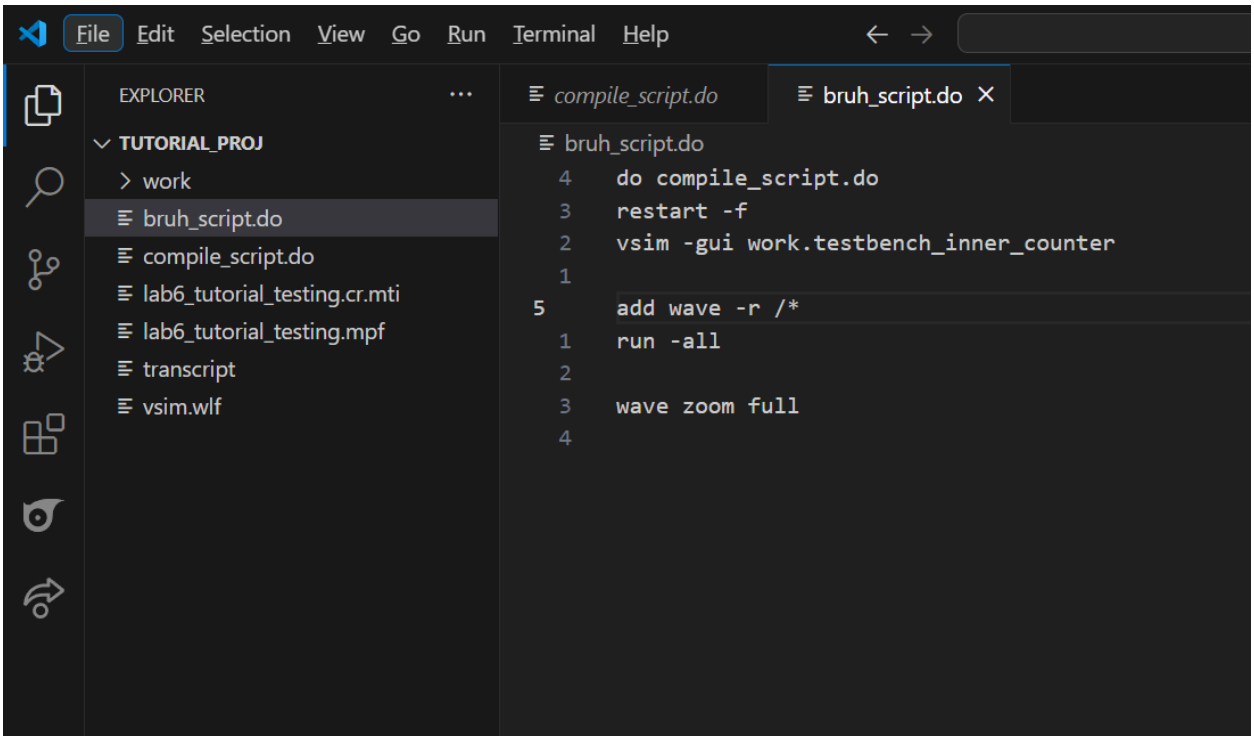


7. Run the script by typing “do compile_script.do” while in the console. “Do” tells Modelsim to do something, and the filename that follows is the specific thing we want it to do.



8. Now hang in there we’re almost to the good part.
    a. NOTE: You can use autocomplete to find the script
    b. Type “do c” (don’t press enter yet) and then press tab and it should complete to the script name.

9. Make a new script (let's call it bruh_script.do) and put the following code into it.

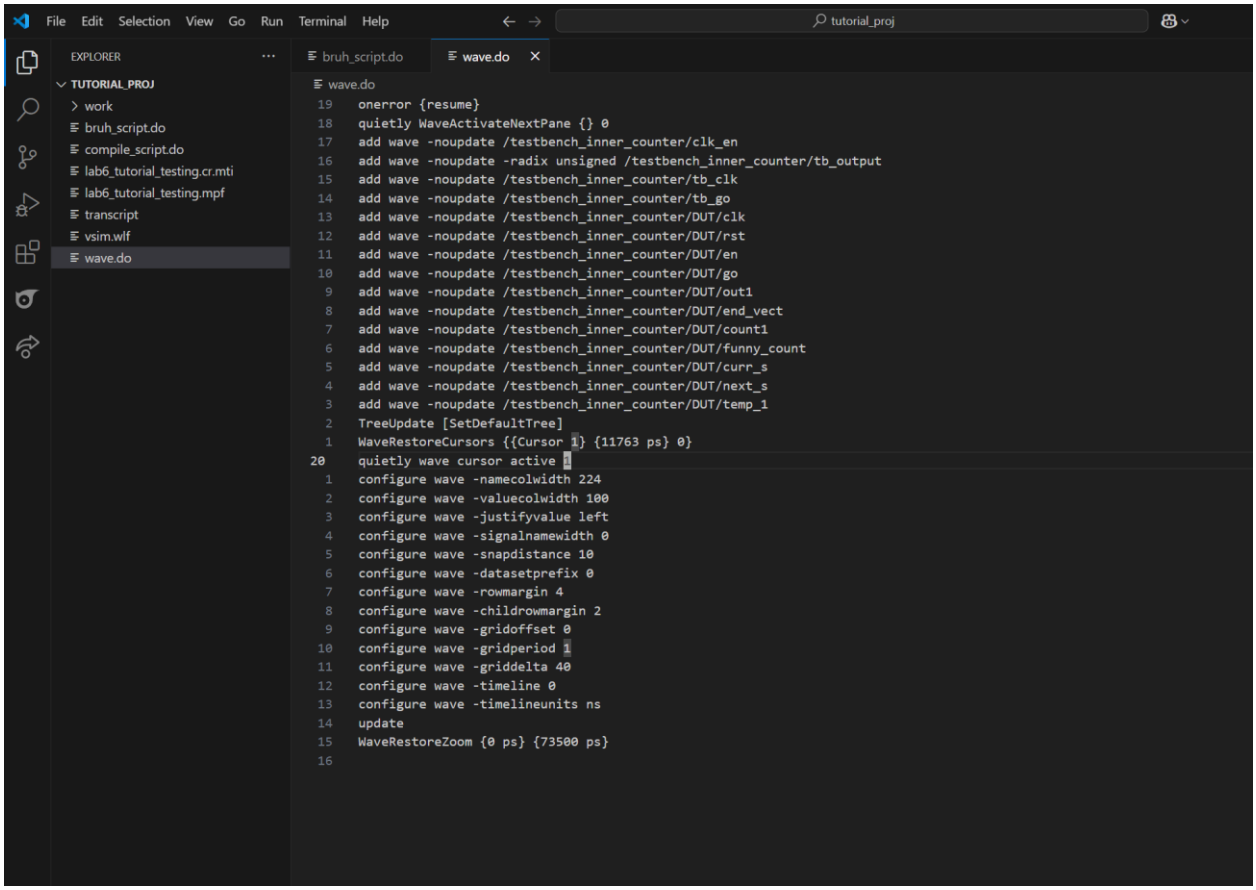


10. Again save this to the same directory we reference early (…../LAB6/tutorial_proj).
    a. "do compile_script.do" this will recompile our design. It also serves as an example of a script being able to call another script. This will be useful in a bit.
    b. "restart -f", assuming you didn't close out the simulation (you shouldn't have), this will restart the entire simulation module
    c. "vsim -gui work.testbench_inner_counter"
        i. IMPORTANT: rename "testbench_inner_counter" to whatever the entity name of your testbench is.
        ii. For example if your testbench (entity) is called "alu_tb" then you should have "vsim -gui work.alu_tb" as the corresponding line.
    d. "add wave -r /*" this does a recursive wave add of all signals in the testbench and the DUTs. May be useful for looking at the internal signals of an entity when the testbench doesn't have proper diagnostic signals.
    e. "run -all" reruns the simulation to it's entirety.
    f. "wave zoom full" zooms the Waveviewer to match the entire window size.
11. Run the script via the same method we just used.
    a. Type "do bruh_script.do" in the console window. Use autocomplete to shorten the typing.
12. You should see your Modelsim totally whig out for a second. This is completely normal.

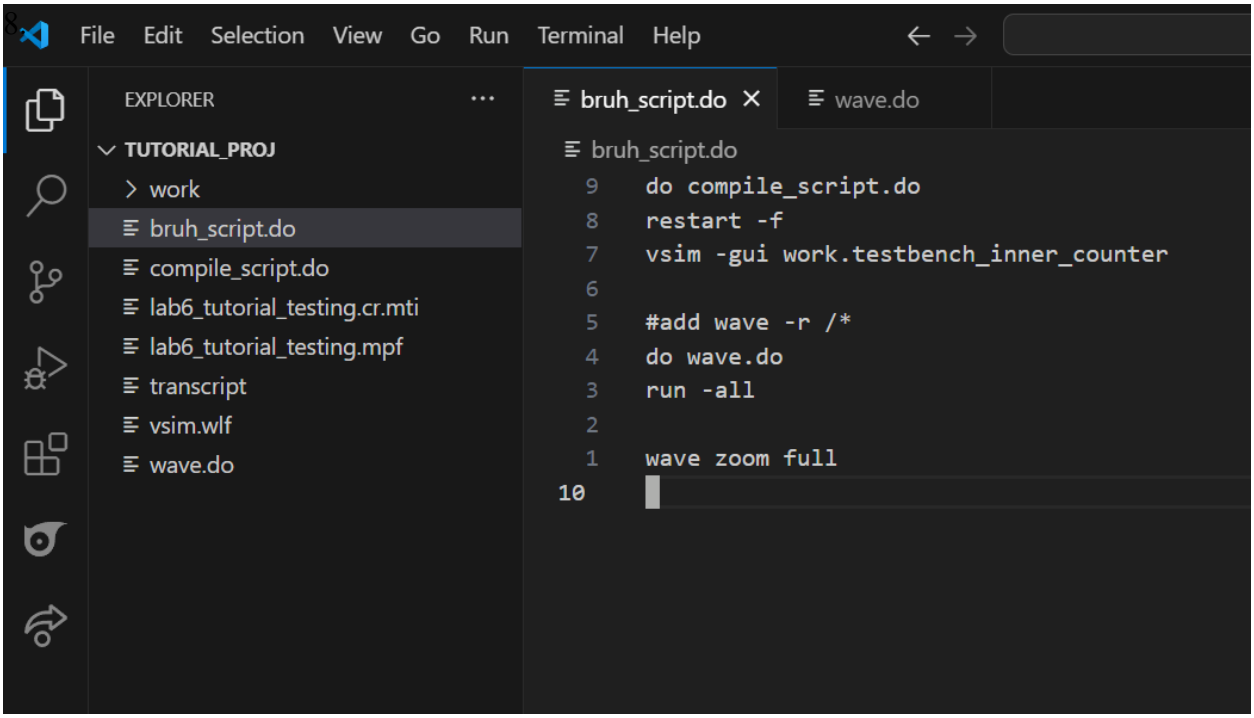
13. You've now successfully written a basic script that recompiles any changes to your project and reruns the simulation.
    a. If you want to make changes to the testbench or any of the VHDL code feel free to. Then simply rerun the "do bruh_script.do" command. This will recompile all of that code and rerun the entire simulation.
14. NOTE: this only works when you've run a simulation for the first time aka done "Simulation" and "Start Simulation" and selected the relevant testbench. This script simply restarts the entire simulation.
    a. You can write your own script just for the initial simulation step if you want but I find this annoying. Refer to the commands that Modelsim runs in it's console whenever you select a GUI dropdown item and use those in your own scripts.

Step 3: Save a Wave Configuration

1. Now you may have noticed that the radix presets **are not** preserved across reruns of the simulation. To do this, first set whatever radixes you want manually.
2. Select the Waveviewer and then "File" in the upper left corner.
3. Select "Save Format" and save this file wherever you want (best to leave it in the default directory as this is where the scripts can reference each other).
4. If you look inside the .do script that Modelsim just saved you should see something similar.



5. If you take note of line 4 “add wave -noupdate -radix ….” you can see the radix of the wave is preserved.
6. Now for the important part, where you add this preset of the wave format to the main script.
7. In your main script (bruh_script.do for us last time), make edits to a similar capacity.



8. You’ll notice that “wave.do” is present in the directory, and also that a ‘#’ has been added to the “add wave….” command run previously. This comments out the line (tells Modelsim not to consider anything on this line to ensure we don’t add a whole mess of extra waves more times than we want).
9. You’ll also notice “do wave.do”. As demonstrated before the bruh_script.do has the capacity to run other scripts it knows about. Because we (hopefully) saved wave.do (or whatever you named it) into the same directory, we can call that script in our main script.
10. Run the main script again by typing “do bruh_script.do” and observe that preset radixes are preserved across instances of the simulation.
11. IMPORTANT: If you add more clock cycles to your testbench then the wave zoom that “wave.do” calls may be inadequate or incorrect. The “wave zoom full” command should mitigate this but keep in mind you have to save a new “wave.do” format every now and then if you undergo significant changes in your benching.

Quartus-Opened Modelsim

As we progress through the labs and start making use of memory modules it becomes apparent that we must use the Quartus Intellectual Property (IP) for certain things such as the RAM or Adder units. Things like the Adder and Multiplier units can be referenced by looking into the Quartus project folder and adding the relevant VHDL files from there **AS WELL AS** their relevant Altera packages from the respective folders that store the IP information. Such simulations can be run separate from Quartus (outside of using Quartus to create the IP variation files) by digging around for the proper includes.

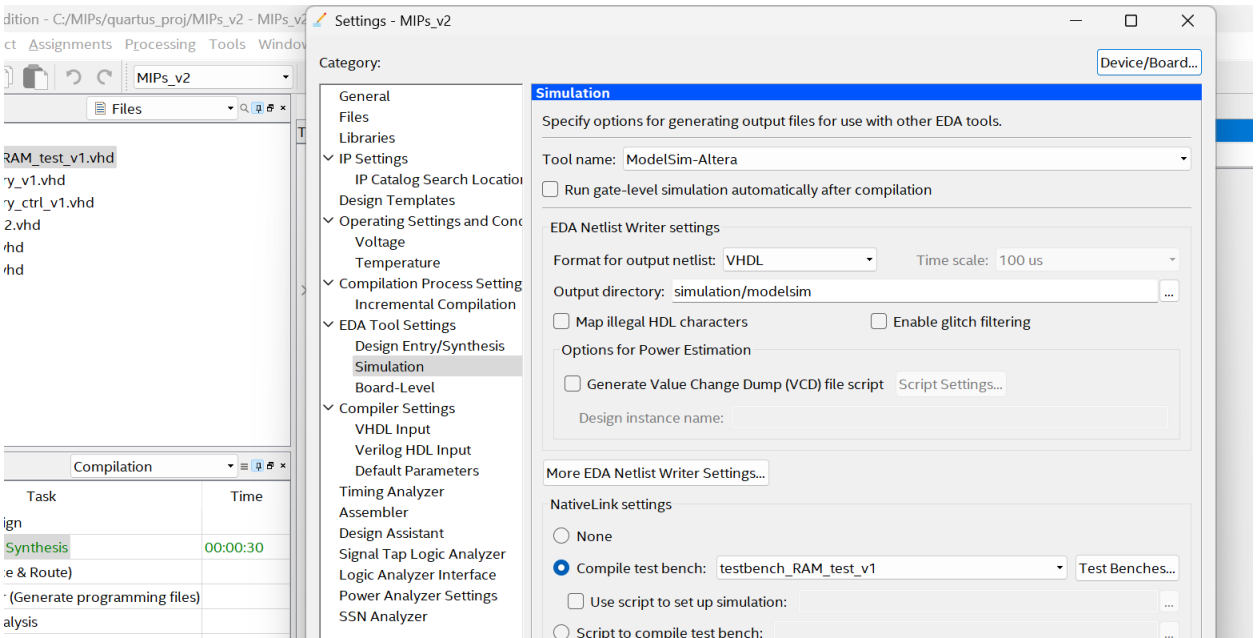
More importantly, however, is making use of the RAM entities and the corresponding **MIF** memory configuration files (redundant statement). On its own RAM can be included from the relevant libraries in a solely-Modelsim setup, however RAM it best utilized with Quartus-Opened instances of Modelsim.

Step 1: Setup the MIF into the RAM IP

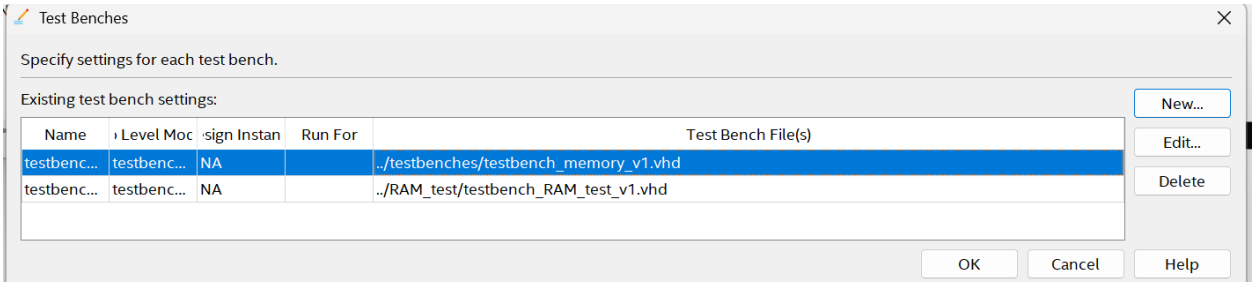
1. We will assume you have gotten to the point where you can declare a RAM IP Variation and load a MIF file into RAM and successfully run compile the Quartus Project with a RAM module instantiated somewhere.

Step 2: Setting up Testbenches in Quartus

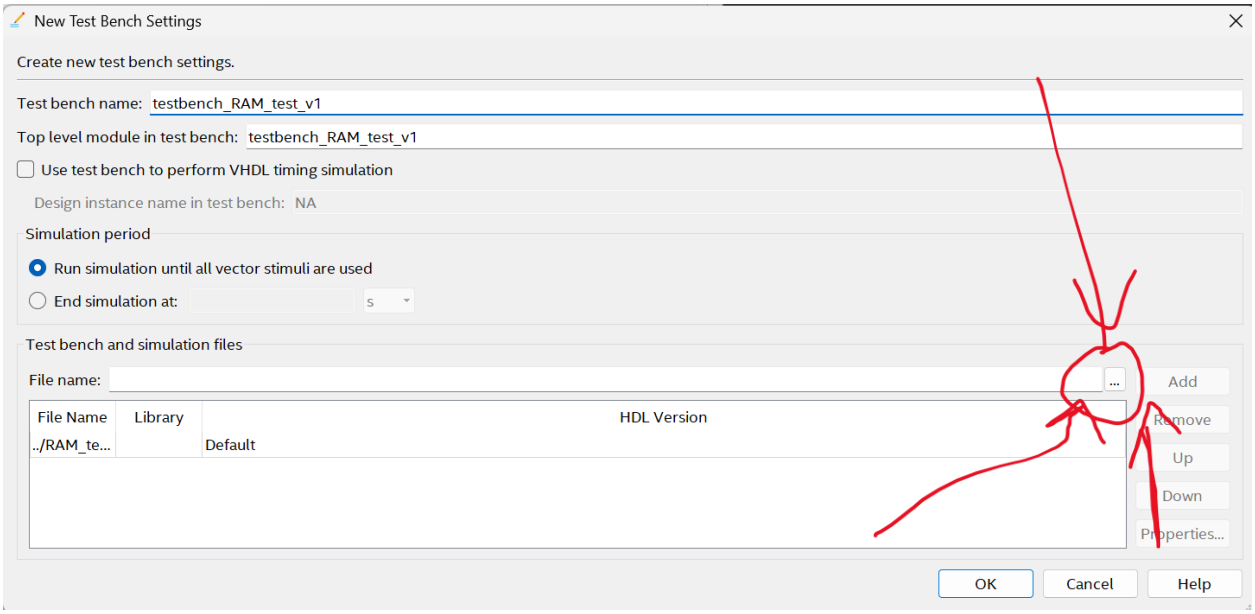
1. Testbenches in Modelsim are a simple matter of including them in the compile order. In Quartus however there is a small bit more configuration required.
2. Select “Assignments” and “Settings”, then select “Simulation” from the “EDA Tool Settings” section. Your screen should look similar outside of the “None” selection for NativeLink.



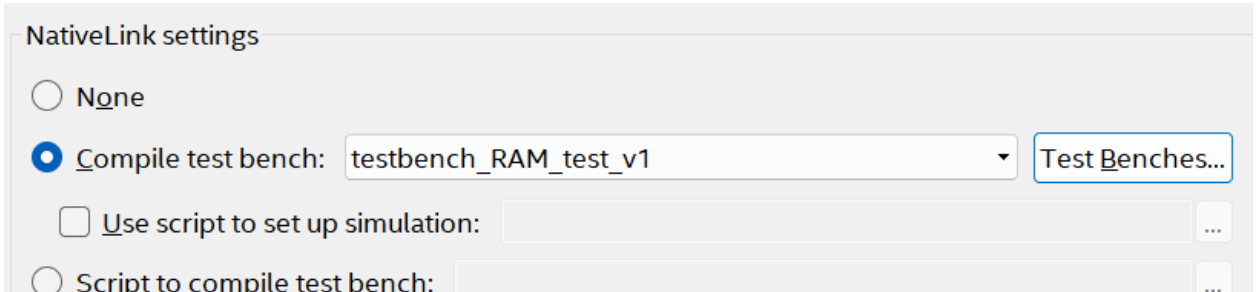
3. Select “Compile test bench” on the lower half of the menu and select “Test Benches…”.
4. This should bring up a window that looks like this (it should be blank for the testbenches, we’ll define ours momentarily).



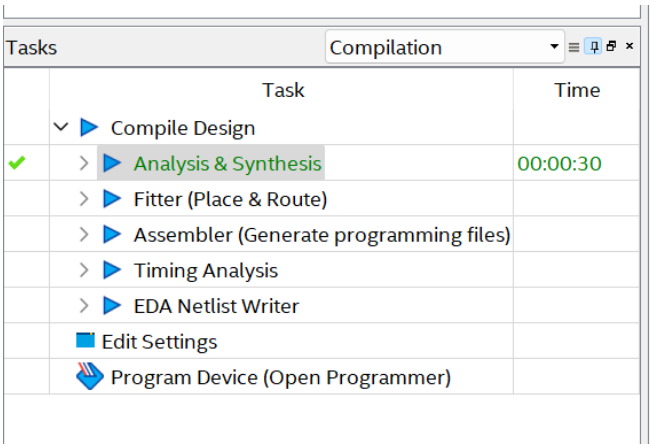
5. Select “New…”, then a window like this should pop up.



6. Click the “…” in the above image and browse to your testbench and select it, then more importantly hit “Add” and you should see the testbench listed as mine is.
7. Fill out the “Test bench name” field with the name of the **entity** in your testbench.
8. Hit “OK” at the bottom of the screen, and “OK” again on the “Test Benches” window.
9. Make sure the relevant testbench is listed next to “Compile test bench” field as displayed a few images above.

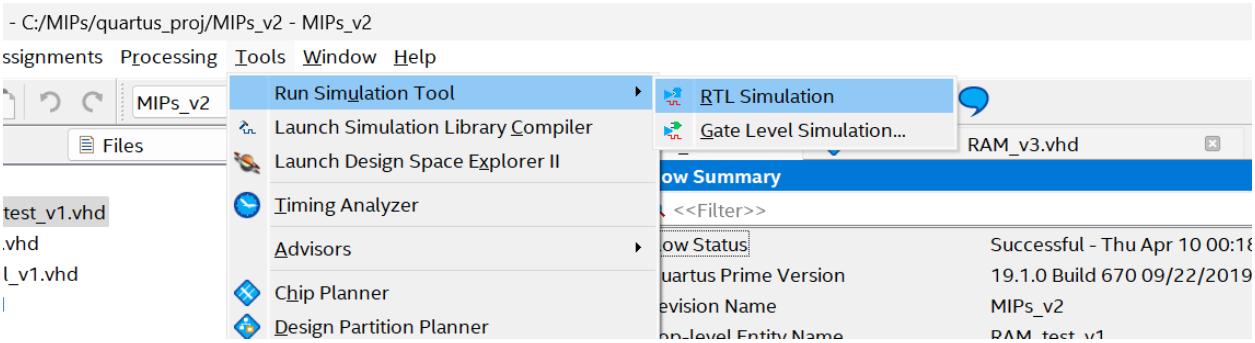


10. Hit “Apply” and “OK”.
11. Compile your design one more time for good measure with the “Analysis & Synthesis” button



12. For checking of syntax and running functional simulations (one’s not on the board), you should only ever select “Analysis & Synthesis” as anything else may output a .vho file that can cause conflicts with Modelsim. If you do a full compile (aka routing the circuit across the FPGA), simply close and reopen Quartus after making a change to your code somewhere and delete the .vho file if the problem persists. This is also magnitudes faster as functionally compiling your design makes up the large majority of compilation time and it is insanely wasteful to fully compile a design that is not going to be immediately flashed onto the DE-10 Lite board.
13. It is a spare thing to note that you do not have to include testbenches in the project files themselves to be compiled by Quartus. If you need syntax checking for the testbench however (assuming you do not have a linter elsewhere via TerosHDL or an LSP) then do include the testbench and Quartus will discern testbench statements like “wait” apart from synthesizable VHDL code and provide relevant syntax checking.

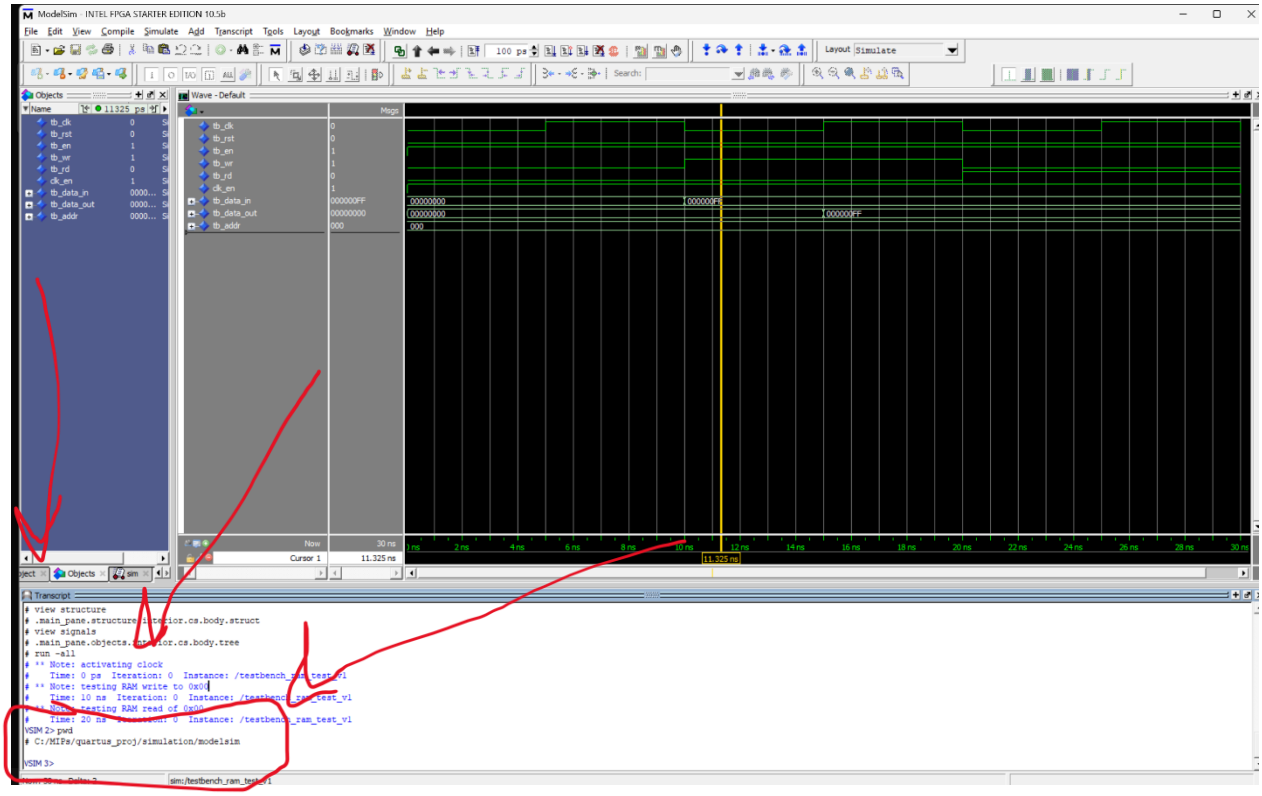
Step 3: Opening Modelsim from Quartus



1. Select “Tools”, then “Run Simulation Tool”, and then “RTL Simulation”. This should open Modelsim and after a small time run the testbench you’ve declared.

Step 4: Opening Modelsim from Quartus

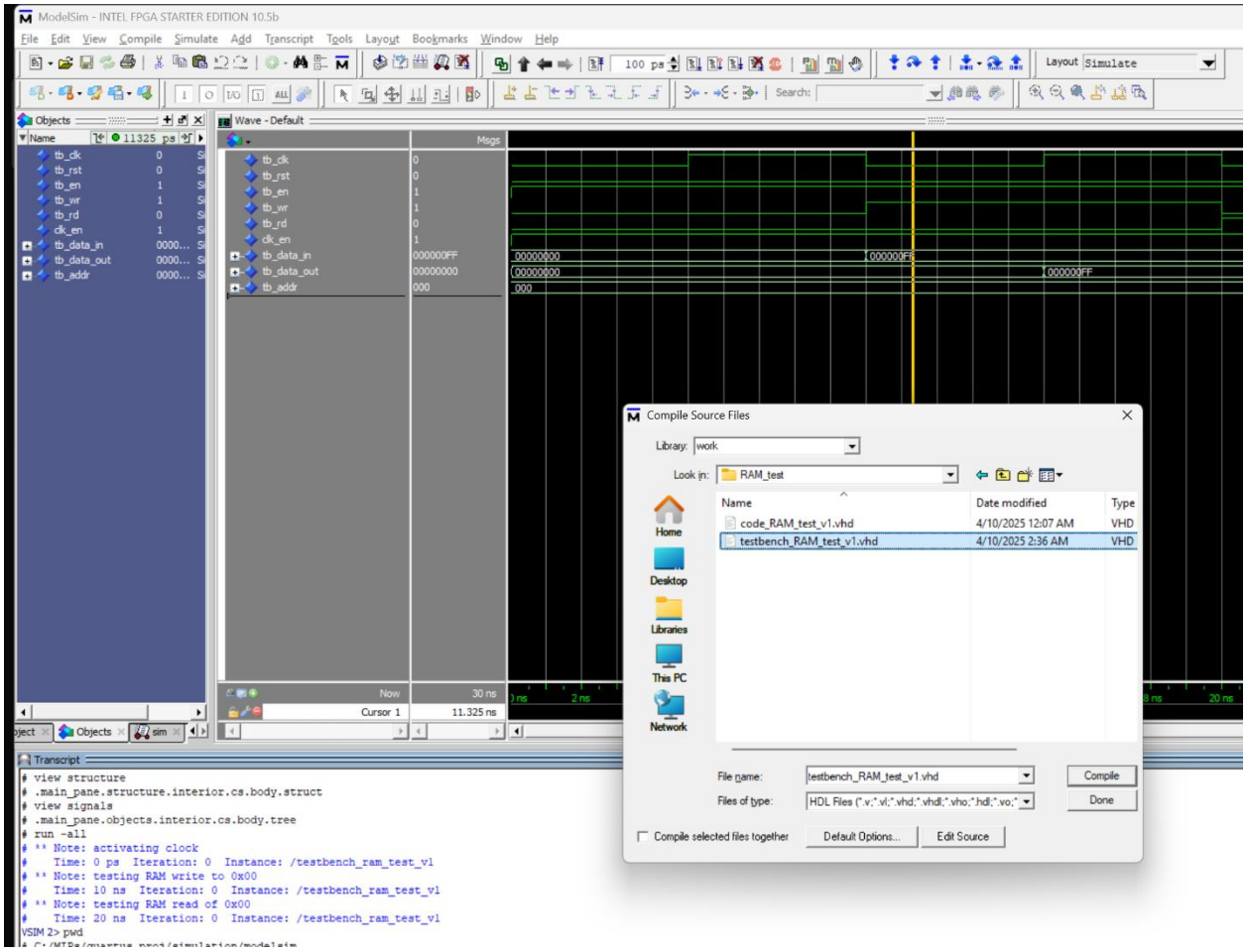
1. Once again set all of the relevant prefixes you deem necessary. This may include radixes, zooming, or internal object signals.
    a. I have no idea how to grab internal object signals outside of the recursive wave add all call so lmk if u find something lmao.
2. Save the wave format to wherever you wish (default directory is best).
3. Get the current directory that Modelsim is using (type “pwd” in the console).



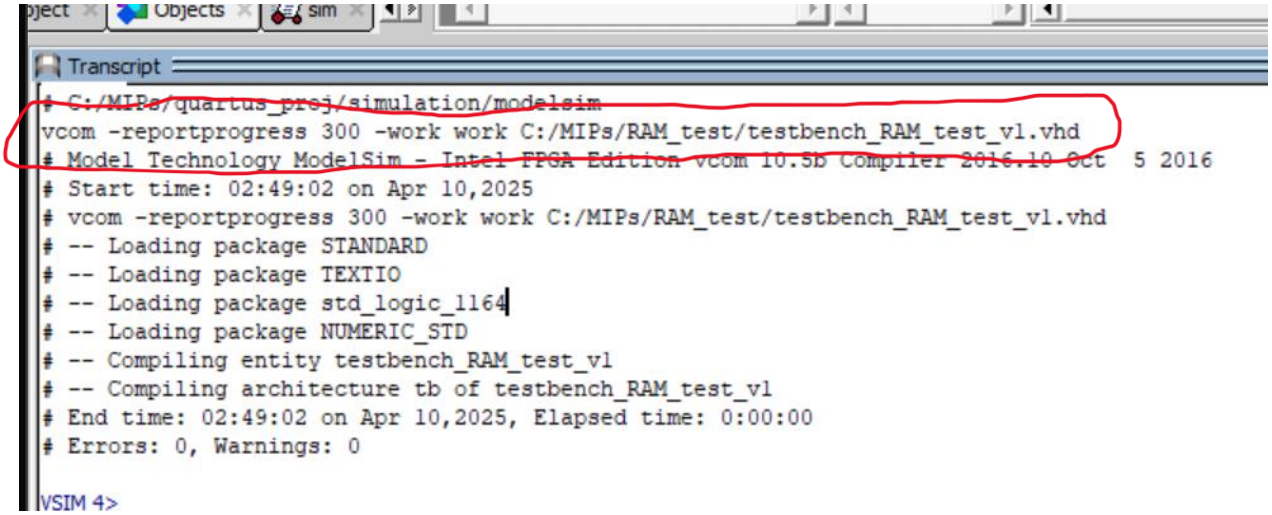
4. Keep this directory location handy. You may notice this is within the Quartus Project and further directories inside it. (C:/MIPs/quartus_proj/simulation/modelsim). This is the directory we will be writing our scripts into.
5. The main difference here is that Modelsim has **not** been opened in a project scope. Nativelink, the software that connects the Quartus and Modelsim instance instead runs a mass of compile and simulation commands manually from its console.

Step 5: Scripting in Quartus'd Modelsim

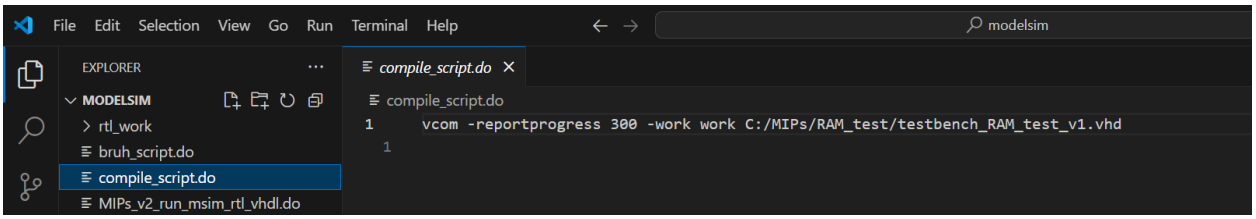
1. Instead of being able to run "project compileall" as a command (since this **isn't**) a project, we must do a targeted compile call.
2. Select "Compile" and then "Compile" (again) and browse to your testbench. Select it.

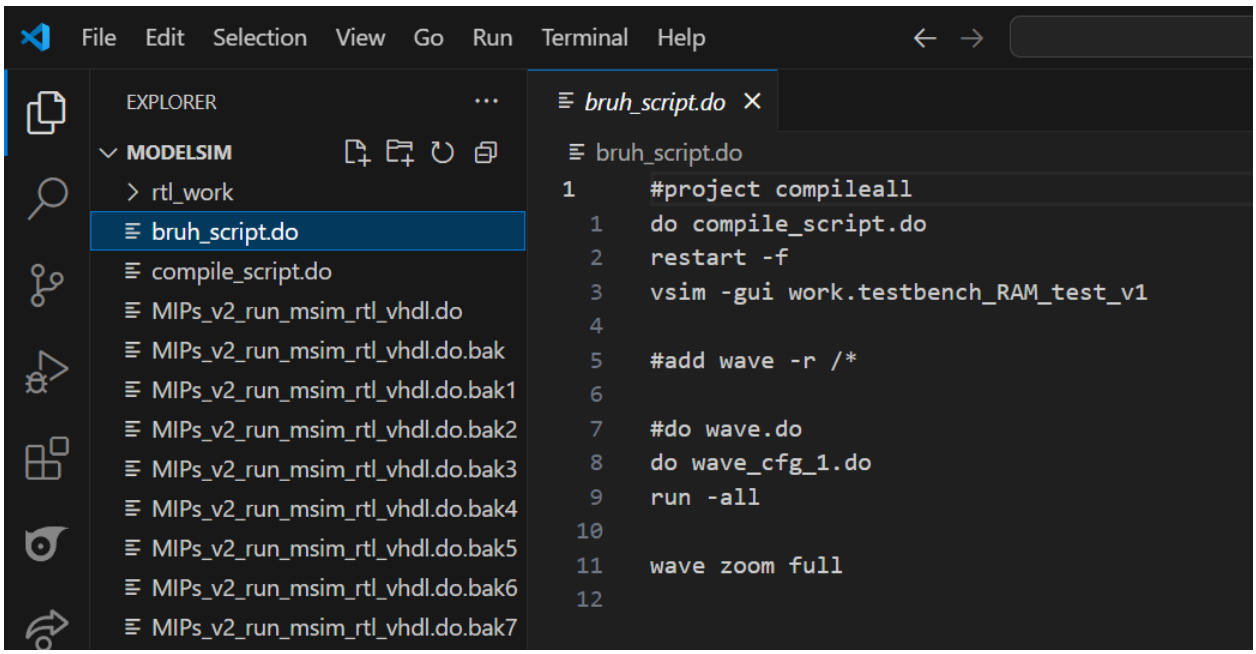


3. Take note of the command (in the console) that Modelsim runs to compile the testbench once you compile.



4. This is the command we will put into our special script. The main point of annoyance is that we have to do compilation in a targeted fashion meaning additional files being added requires more lines to our main script (in contrast to our testbenching which remains somewhat more stagnant).
5. Make a new script that runs this command, restarts the simulation, and runs the wave config that we want.





6. NOTE: This targeted compilation is targeting the testbench. Normally the “project compileall” call would cover this but we must do this ourselves. You can do in a separate script as I have done (for example) or do it in the main script.
7. Notice that a ‘#’ has been added to the first line which comments it out. I also saved my wave configuration as “wave_cfg_1.do” instead of the default.
8. These scripts have been saved into the directory we figured out a few steps ago (which differs from a standard Modelsim project).
   “C:/DD/MIPs/quartus_proj/simulation/modelsim”.
9. Run the script with “do bruh_script.do” in the console and watch Modelsim rerun the simulation with the same presets as before.
10. Again, we are able to make changes to the testbench as we see fit and simply rerun the main script to recompile it instead of closing Modelsim, recompiling in Quartus, reopening Modelsim and repeating. If you however want to make changes to the design code (for example code_RAM_test_v1.vhd) we must have Modelsim recompile this.
11. To do this you will have to add another specific-target compile to your compile script. Alternatively, you can close out of Modelsim and compile in Quartus and then rerun Modelsim from there.
12. The main use of this separate section relates to the inclusion of **MIF** files for initialization of memory. Modelsim on its own is not capable of loading these into the RAM files, which we must use Quartus to do. However, once that is done, we can use our modified verification-flow to streamline the process of making changes and testing them in Modelsim.

Goodluck on your MIPs everyone!

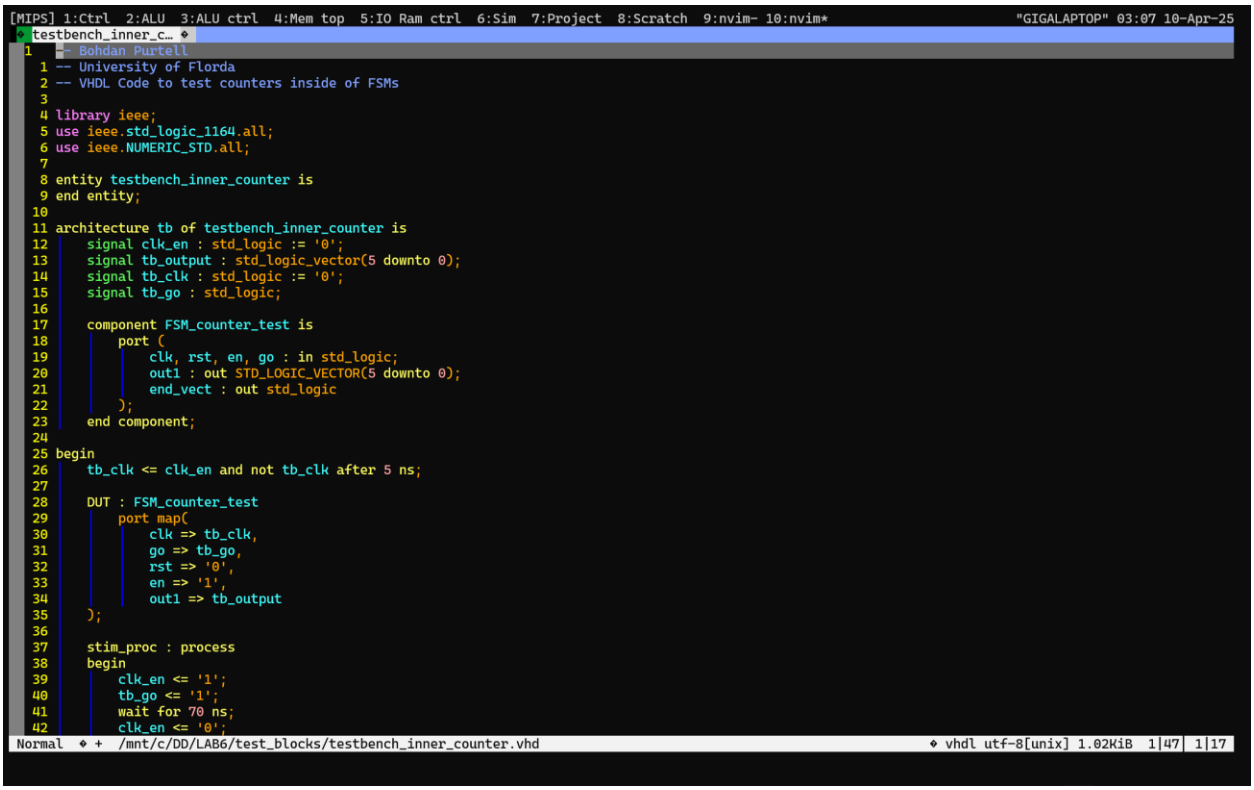
Appendix

code_inner_counter

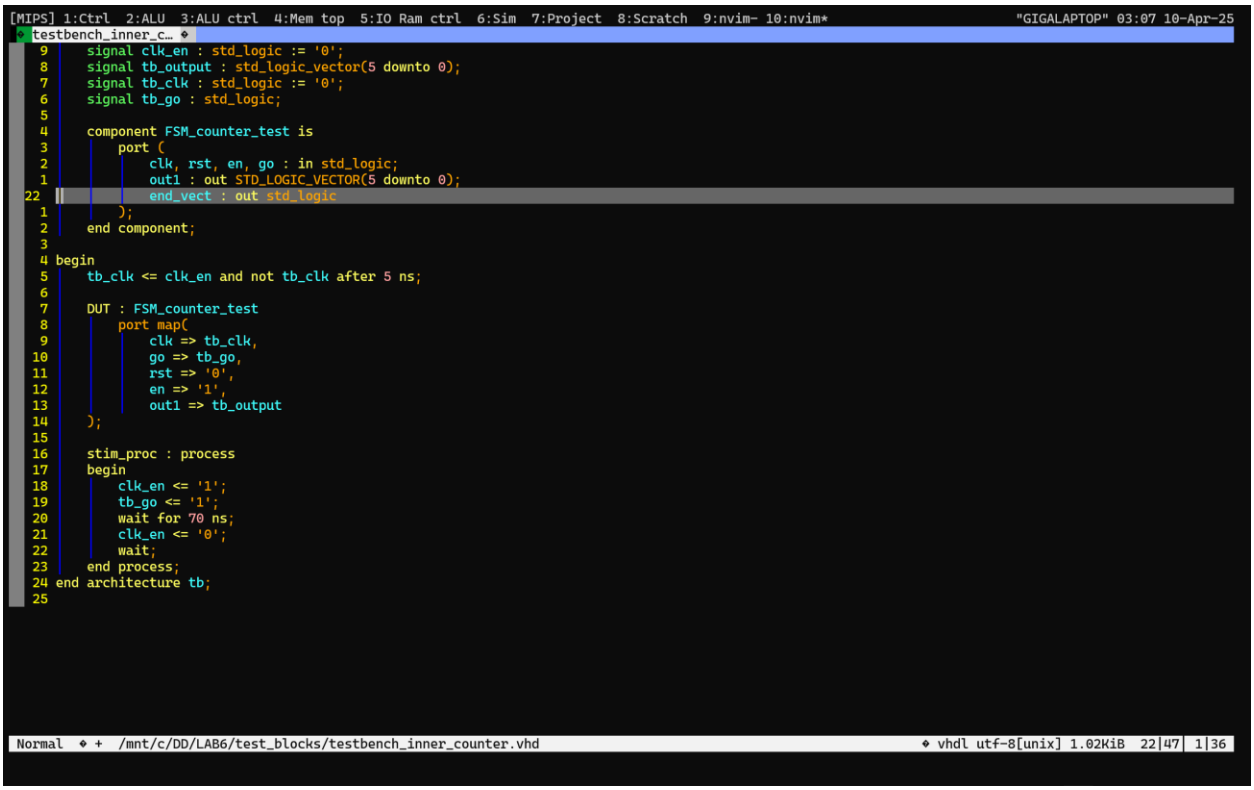
```
[MIPS] 1:Ctrl  2:ALU  3:ALU ctrl  4:Mem top  5:IO Ram ctrl  6:Sim  7:Project  8:Scratch  9:nvim- 10:nvim*                    "GIGALAPTOP" 03:06 10-Apr-25
 code_inner_counte… 
 23 -- Bohdan Purtell
 22 -- University of Florda
 21 -- VHDL Code to test counters inside of FSMs
 20 -- i have deduced the need for the next_count register to account for sequential assignment of the increment register
 19 -- truly a moment of all time
 18 -- this only applies to this internal register usage in FSMs since FSMs are inherently sequntial while the addr generator itself is a merely a counter,
     problem being the control of the external thingy
 17
 16 library ieee;
 15 use ieee.std_logic_1164.all;
 14 use ieee.NUMERIC_STD.all;
 13
 12 entity FSM_counter_test is
 11     port (
 10         clk, rst, en, go : in std_logic;
  9         out1 : out STD_LOGIC_VECTOR(5 downto 0);
  8         end_vect : out std_logic
  7     );
  6 end entity;
  5
  4 architecture rtl of FSM_counter_test is
  3     signal count1 : unsigned(5 downto 0) := (others => '0');
  2     signal funny_count : unsigned(5 downto 0) := (others => '0');
  1     type state_type is (s1, s2, s3, compute, empty, done);
 24
  1     signal curr_s, next_s : state_type := s1;
  2     signal temp_1 : std_logic;
  3
  4 begin
  5     -- combinatonal assignments
  6     out1 <= STD_LOGIC_VECTOR(count1);
  7
  8     -- clk_proc
  9     process(clk, rst)
 10     begin
 11         if rst = '1' then
 12             count1 <= (others => '0');
 13         elsif rising_edge(clk) then
 14             curr_s <= next_s;
 15             if temp_1 = '1' then
 16                 count1 <= funny_count;
 17             end if;
 18         end if;
 Normal  ◆ +  /mnt/c/DD/LAB6/test_blocks/code_inner_counter.vhd                ◆ vhdl utf-8[unix] 2.33KiB  24|86| 1|0
86 lines yanked
```

```
[MIPS] 1:Ctrl  2:ALU  3:ALU ctrl  4:Mem top  5:IO Ram ctrl  6:Sim  7:Project  8:Scratch  9:nvim- 10:nvim*                    "GIGALAPTOP" 03:06 10-Apr-25
 code_inner_counte… 
 43     end process;
  1
  2     -- netx state proc
  3     process(curr_s, go, count1)
  4     begin
  5         next_s <= curr_s;
  6         funny_count <= count1;
  7
  8         case curr_s is
  9             when s1 =>
 10                 if go = '1' then
 11                     next_s <= s2;
 12                 else
 13                     next_s <= s1;
 14                 end if;
 15
 16             when s2 => -- already asserted go
 17                 if count1 = 5 then
 18                     next_s <= compute;
 19                 else
 20                     funny_count <= count1 + 1;
 21                     next_s <= s2;
 22                     temp_1 <= '1';
 23                 end if;
 24
 25             when compute =>
 26                 if count1 = 10 then
 27                     next_s <= done;
 28                 else
 29                     funny_count <= count1 + 1;
 30                     next_s <= compute;
 31                 end if;
 32
 33             when done =>
 34                 next_s <= done;
 35
 36             when others =>
 37                 next_s <= empty;
 38
 39         end case;
 40     end process;
 41
 42 end architecture rtl;
 Normal  ◆ +  /mnt/c/DD/LAB6/test_blocks/code_inner_counter.vhd                ◆ vhdl utf-8[unix] 2.33KiB  43|86|16|16
86 lines yanked
```

testbench_inner_counter.vhd



```
[MIPS] 1:Ctrl  2:ALU  3:ALU ctrl  4:Mem top  5:IO Ram ctrl  6:Sim  7:Project  8:Scratch  9:nvim- 10:nvim*                "GIGALAPTOP" 03:07 10-Apr-25
 testbench_inner_c… 
1   -- Bohdan Purtell
 1 -- University of Florda
 2 -- VHDL Code to test counters inside of FSMs
 3
 4 library ieee;
 5 use ieee.std_logic_1164.all;
 6 use ieee.NUMERIC_STD.all;
 7
 8 entity testbench_inner_counter is
 9 end entity;
10
11 architecture tb of testbench_inner_counter is
12     signal clk_en : std_logic := '0';
13     signal tb_output : std_logic_vector(5 downto 0);
14     signal tb_clk : std_logic := '0';
15     signal tb_go : std_logic;
16
17     component FSM_counter_test is
18         port (
19             clk, rst, en, go : in std_logic;
20             out1 : out STD_LOGIC_VECTOR(5 downto 0);
21             end_vect : out std_logic
22         );
23     end component;
24
25 begin
26     tb_clk <= clk_en and not tb_clk after 5 ns;
27
28     DUT : FSM_counter_test
29         port map(
30             clk => tb_clk,
31             go => tb_go,
32             rst => '0',
33             en => '1',
34             out1 => tb_output
35     );
36
37     stim_proc : process
38     begin
39         clk_en <= '1';
40         tb_go <= '1';
41         wait for 70 ns;
42         clk_en <= '0';
Normal  ◆ +  /mnt/c/DD/LAB6/test_blocks/testbench_inner_counter.vhd                ◆ vhdl utf-8[unix] 1.02KiB  1|47|  1|17
```

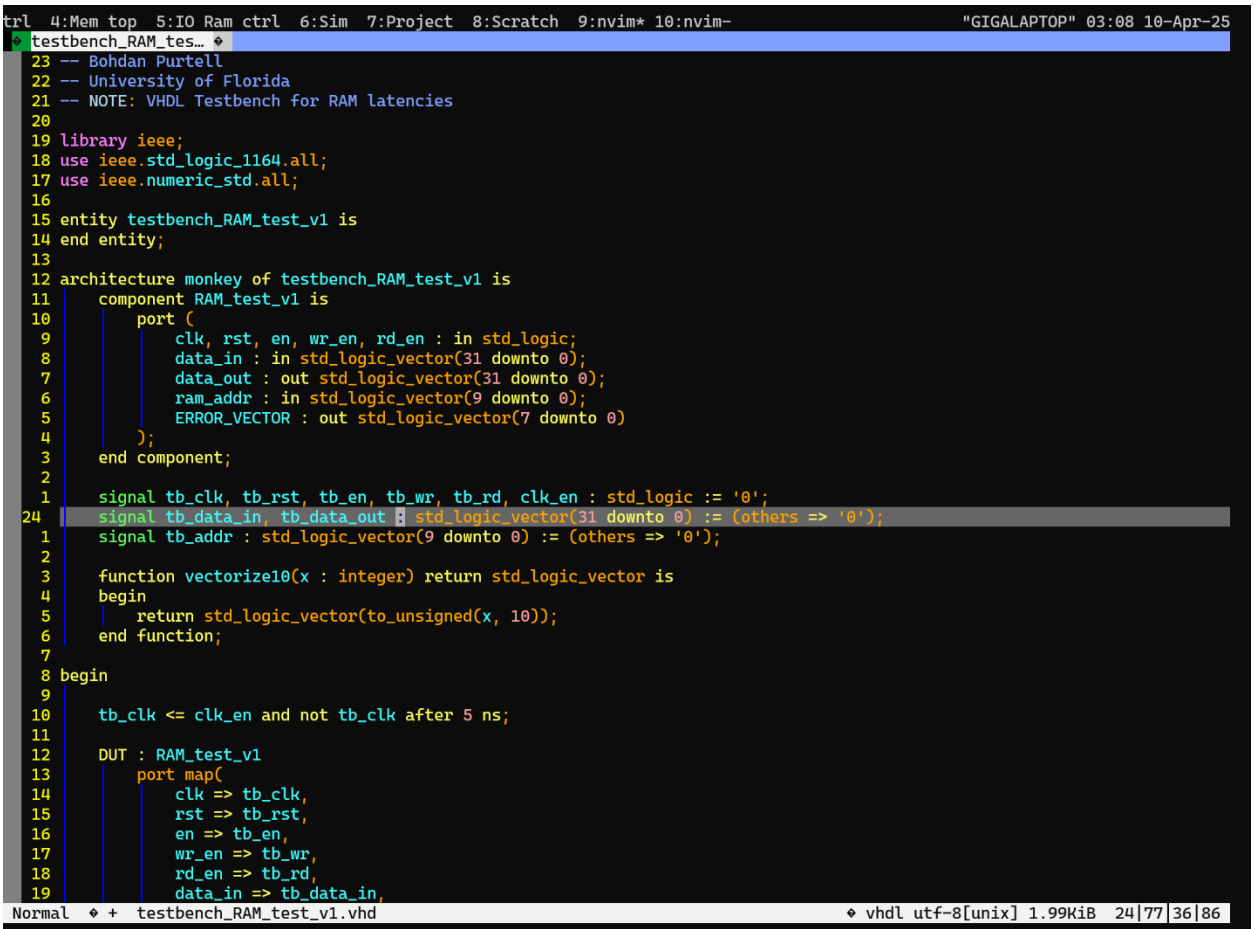


```
[MIPS] 1:Ctrl  2:ALU  3:ALU ctrl  4:Mem top  5:IO Ram ctrl  6:Sim  7:Project  8:Scratch  9:nvim- 10:nvim*                "GIGALAPTOP" 03:07 10-Apr-25
 testbench_inner_c… 
 9     signal clk_en : std_logic := '0';
 8     signal tb_output : std_logic_vector(5 downto 0);
 7     signal tb_clk : std_logic := '0';
 6     signal tb_go : std_logic;
 5
 4     component FSM_counter_test is
 3         port (
 2             clk, rst, en, go : in std_logic;
 1             out1 : out STD_LOGIC_VECTOR(5 downto 0);
22             end_vect : out std_logic
 1         );
 2     end component;
 3
 4 begin
 5     tb_clk <= clk_en and not tb_clk after 5 ns;
 6
 7     DUT : FSM_counter_test
 8         port map(
 9             clk => tb_clk,
10             go => tb_go,
11             rst => '0',
12             en => '1',
13             out1 => tb_output
14     );
15
16     stim_proc : process
17     begin
18         clk_en <= '1';
19         tb_go <= '1';
20         wait for 70 ns;
21         clk_en <= '0';
22         wait;
23     end process;
24 end architecture tb;
25
Normal  ◆ +  /mnt/c/DD/LAB6/test_blocks/testbench_inner_counter.vhd                ◆ vhdl utf-8[unix] 1.02KiB  22|47|  1|36
```

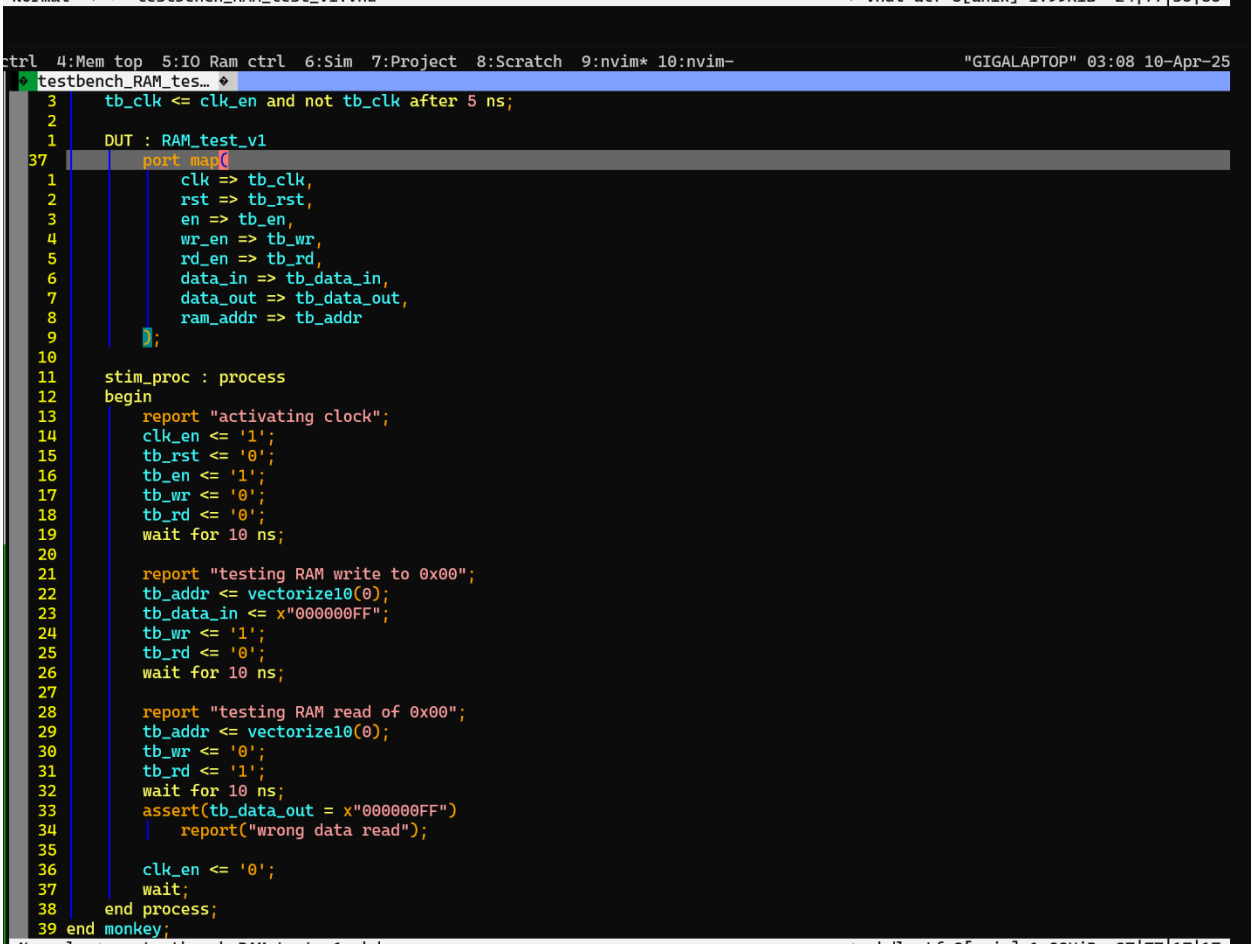
code_RAM_test_v1.vhd

```
[MIPS] 1:Ctrl  2:ALU  3:ALU ctrl  4:Mem top  5:IO Ram ctrl  6:Sim  7:Project
-- Bohdan Purtell
-- University of Florida
-- NOTE: this is a test thingy to test the RAM latency

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity RAM_test_v1 is
    port (
        clk, rst, en, wr_en, rd_en : in std_logic;
        data_in : in std_logic_vector(31 downto 0);
        data_out : out std_logic_vector(31 downto 0);
        ram_addr : in std_logic_vector(9 downto 0);
        ERROR_VECTOR : out std_logic_vector(7 downto 0)
    );
end entity;

architecture cheese of RAM_test_v1 is
    component RAM_v3 IS
        PORT (
            address     : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
            clock       : IN STD_LOGIC  := '1';
            data        : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
            wren        : IN STD_LOGIC ;
            q       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
        );
    end component;

    signal wire_RAM_data_in : std_logic_vector(data_in'range);
    signal wire_RAM_data_out : std_logic_vector(data_in'range);

begin
    bruh : RAM_v3
        port map(
            address => ram_addr,
            clock => clk,
            data => data_in,
            wren => wr_en,
            q => data_out
        );
end cheese;

 N  code_RAM_test_v1.vhd                                  vhdl  17 11
"code_RAM_test_v1.vhd" 43L, 1137B written
```

testbench_RAM_test_v1.vhd



```
-- Bohdan Purtell
-- University of Florida
-- NOTE: VHDL Testbench for RAM latencies

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity testbench_RAM_test_v1 is
end entity;

architecture monkey of testbench_RAM_test_v1 is
    component RAM_test_v1 is
        port (
            clk, rst, en, wr_en, rd_en : in std_logic;
            data_in : in std_logic_vector(31 downto 0);
            data_out : out std_logic_vector(31 downto 0);
            ram_addr : in std_logic_vector(9 downto 0);
            ERROR_VECTOR : out std_logic_vector(7 downto 0)
        );
    end component;

    signal tb_clk, tb_rst, tb_en, tb_wr, tb_rd, clk_en : std_logic := '0';
    signal tb_data_in, tb_data_out : std_logic_vector(31 downto 0) := (others => '0');
    signal tb_addr : std_logic_vector(9 downto 0) := (others => '0');

    function vectorize10(x : integer) return std_logic_vector is
    begin
        return std_logic_vector(to_unsigned(x, 10));
    end function;

begin

    tb_clk <= clk_en and not tb_clk after 5 ns;

    DUT : RAM_test_v1
        port map(
            clk => tb_clk,
            rst => tb_rst,
            en => tb_en,
            wr_en => tb_wr,
            rd_en => tb_rd,
            data_in => tb_data_in,
```



```
    tb_clk <= clk_en and not tb_clk after 5 ns;

    DUT : RAM_test_v1
        port map(
            clk => tb_clk,
            rst => tb_rst,
            en => tb_en,
            wr_en => tb_wr,
            rd_en => tb_rd,
            data_in => tb_data_in,
            data_out => tb_data_out,
            ram_addr => tb_addr
        );

    stim_proc : process
    begin
        report "activating clock";
        clk_en <= '1';
        tb_rst <= '0';
        tb_en <= '1';
        tb_wr <= '0';
        tb_rd <= '0';
        wait for 10 ns;

        report "testing RAM write to 0x00";
        tb_addr <= vectorize10(0);
        tb_data_in <= x"000000FF";
        tb_wr <= '1';
        tb_rd <= '0';
        wait for 10 ns;

        report "testing RAM read of 0x00";
        tb_addr <= vectorize10(0);
        tb_wr <= '0';
        tb_rd <= '1';
        wait for 10 ns;
        assert(tb_data_out = x"000000FF")
            report("wrong data read");

        clk_en <= '0';
        wait;
    end process;
end monkey;
```