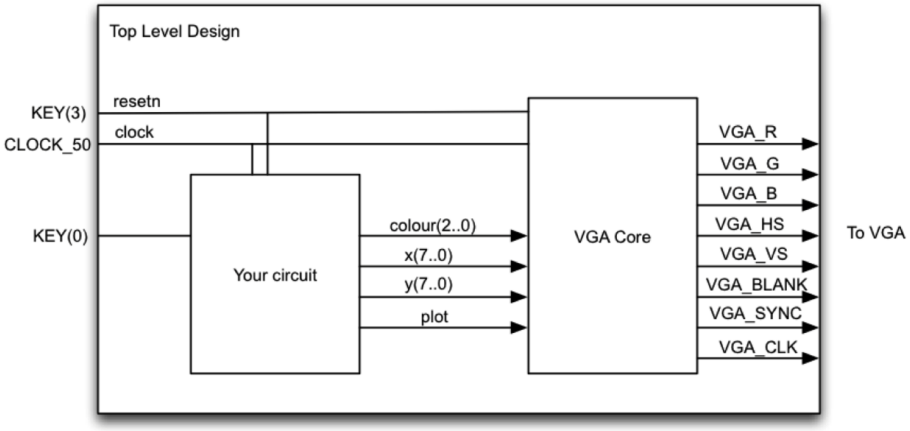
CPEN 311 Lab 4

1 Introduction

You will get more experience creating datapaths and state machines in this lab. You will also learn how to use an embedded VGA adapter core that we will give you to draw images on the screen. The top-level diagram of your lab is shown below. The VGA Core is the part given to you. Your design will be in the block labelled "your circuit."



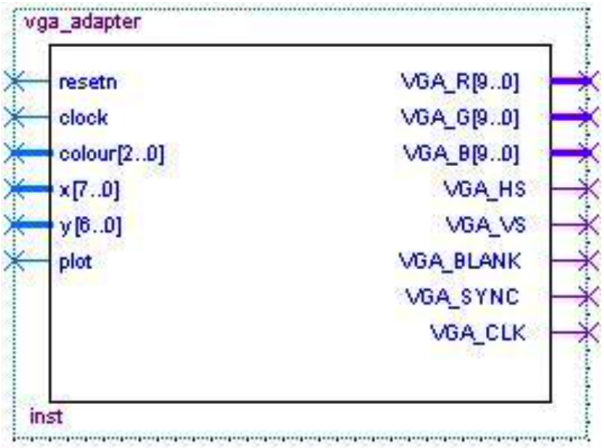
2 External IP

Most designs in the computing industry incorporate modules from another team or company. In this lab, we provide you with a VGA adapter core that converts a frame buffer view of the screen to the actual signals expected on the physical VGA port.

2.1 Task 1: Understanding the VGA adapter core

The VGA adapter was created at the University of Toronto for a course similar to CPEN 311. The description below is enough for you to use the core, but you can find the complete specification at the original site. Some of the following figures have been taken with permission from that website (with permission).

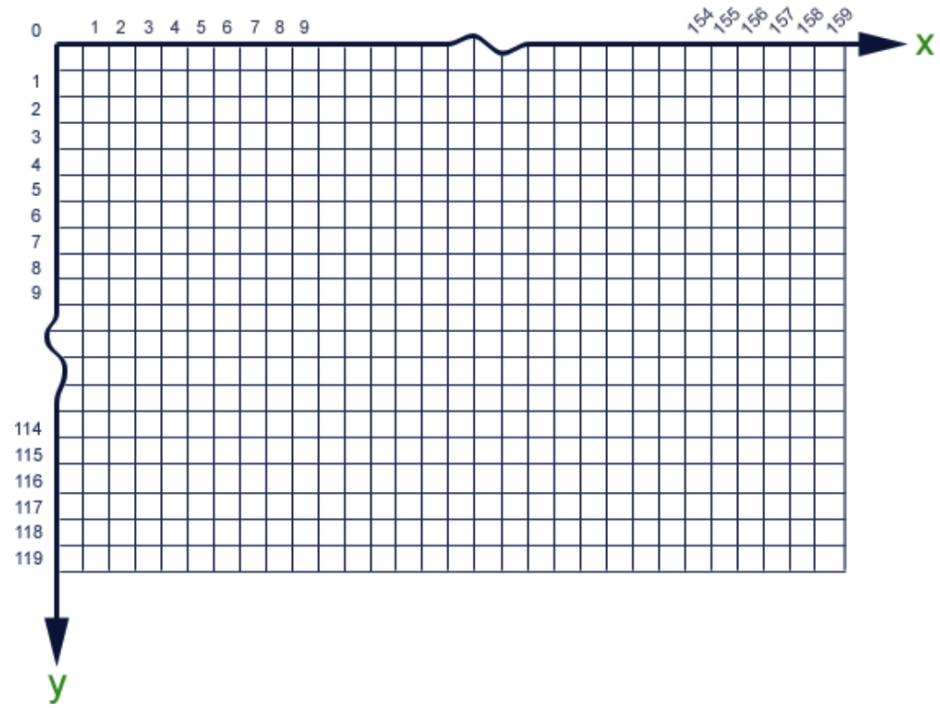
The VGA adapter has been set up to display a grid of 160×120 pixels, only *slightly* worse than the Atari 2600. The interface is as shown below:



Signal	Direction	Semantics
resetn	input	active-low reset
clock	input	50MHz clock
colour[2:0]	input	pixel colour (3 bits), each bit indicating the presence of one of the RGB components
x[7:0]	input	x coordinate of pixel to be drawn (0 ≤ x < 160)
y[6:0]	input	x coordinate of pixel to be drawn (0 ≤ x < 120)
plot	input	active-high plot enable signal
VGA_CLK	output	VGA clock (25 MHz)
VGA_R[9:0]	output	VGA red component
VGA_G[9:0]	output	VGA green component
VGA_B[9:0]	output	VGA blue component
VGA_HS	output	VGA horizontal sync
VGA_VS	output	VGA vertical sync
VGA_SYNC	output	VGA special synchronization mode
VGA_BLANK	output	VGA special blank mode

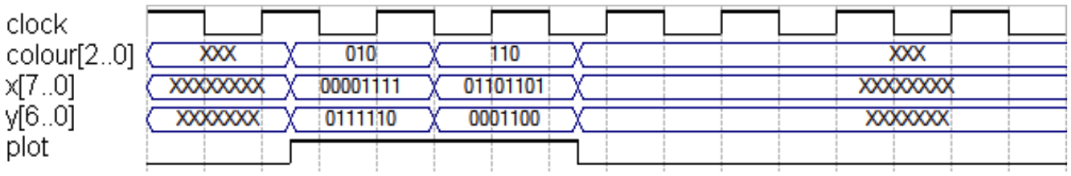
You will connect all outputs of the VGA core directly to the appropriate output pins of the FPGA. The VGA core will then continuously draw pixels on a VGA monitor connected to your board.

You can picture the VGA pixel grid as shown below. The (x,y) position (0,0) is located in the top-left corner, and (159,119) is at the other extreme end. The pixel colours are stored in each cell. This grid is stored in on-chip memory called a frame buffer.



Initially, all pixels have a value that depends on how the FPGA was powered up (black if you're lucky) because the framebuffer is just a memory on the FPGA. To change the colour of a pixel, you must write to the framebuffer memory. To do this, you drive the *x* input with the x position of the pixel, drive the *y* input with the y position of the pixel, and *colour* with the colour you wish to use. You then raise the *plot* signal. The new colour is written to the framebuffer at the next rising clock edge. At some point in the near future, during the next screen refresh cycle, the entire frame buffer is read out, and all of the then-current pixel colours will be drawn on the screen.

In the following timing diagram, two pixels are turned on: one at (15,62) and the other at (109,12). As you can see, the first pixel drawn is green and is placed at (15,62), while the second is a yellow pixel at (109,12). It is important to note that, at most, one pixel can be turned on each cycle — thus, if you want to turn on *m* pixels, you need *m* cycles.



The source code for the VGA adapter is in the `vga-core` folder in this repository.

To help you understand the interface, we have created a `vga_demo.sv` file, which connects the core I/O to switches on the DE1-SoC so you can experiment. We suggest you understand this module and try it out to ensure you understand how the inputs of the core work. (Note that you will not use this file for the other tasks in this Lab.)

2.2 VGA bus widths on the DE1-SoC

The VGA module we have provided has 10 bits of precision for the VGA red, green, and blue component signals. The DE1-SoC FPGA board we use has a DAC with only eight bits of precision for the same signals (you can see this in the pin assignment file). Thus, you must take the 8 most significant bits output by the VGA adapter core to drive the board outputs.

2.3 Simulation with the VGA core

To simulate the VGA core in ModelSim, you must load the `altera_mf_ver` library to provide the block RAMs and PLL modules needed by the VGA core. For post-synthesis simulation, you will additionally require `cyclonev_ver`, `altera_ver`, and `altera_lnsim_ver` to model the FPGA logic cells, registers, and internal memories – similar to lab 3.

We have also hacked up a simulation GUI. It intercepts the signals passed to the VGA adapter and draws the pixels in an on-screen window. You need to load the `de1_vga_gui.tcl` script; the SystemVerilog component is included in the VGA adaptor Verilog (at the end of the file). We have also explained how to use the Virtual DE1-SoC at the end of this file.

2.4 Verification with the VGA core

So, how do you verify that things are working?

Well, if you were really building a chip, you would verify the entire thing end-to-end by checking that the analog VGA outputs are exactly what they should be. But to save you some time (and learn the finer points of VGA signalling), we will not require you to test the VGA core itself — i.e., for this lab, it will suffice to test the inputs that go into the VGA core.

This is easy to do with the RTL since you can reach in and look at what is being fed to the VGA core. But what about the post-synthesis netlist? All those signals will have gone away, so you can't reach in and examine them using hierarchical naming.

To help you with this, we have provided external ports for the VGA core inputs in the toplevel files. These ports will survive synthesis (and be assigned pin locations on the FPGA), so you can access them to validate your post-synthesis netlist.

3 Design

3.1 Task 2: Fill the Screen

You will first create a simple circuit that fills the screen. Each column will be a different colour (repeating every 8 columns). Remember that you can only turn on one pixel at a time, so you would need an algorithm similar to this pseudocode:

```
for x = 0 to 159:
    for y = 0 to 119:
        turn on pixel (x, y) with colour (x mod 8)
```

You are to create a circuit that implements the above algorithm. A skeleton file `fillscreen.sv`, and a top-level skeleton `task2.sv`, are in the task2 folder. The interface to the `fillscreen` module is as follows:

Signal	Direction	Semantics
clk	input	clock
rst_n	input	active-low *asynchronous* reset
colour	input	fill colour (ignored for Task 2)
start	input	assert to start filling the screen
done	output	goes high once the entire screen is filled
vga_*	output	outputs to the VGA adapter core

The user of this module will assert `start` and hold it high until your module asserts `done`. You must ignore the `colour` input for Task 2, but it will be useful for the later tasks.

The `task2` module is the toplevel module you will load into the FPGA. It should instantiate the VGA adapter and your `fillscreen` module and fill the entire screen with the colour stripes on the reset (KEY3, active-low, asynchronous). This must be finished in 19,210 cycles of CLOCK_50 from the time the reset is *deasserted*, i.e., one cycle per screen pixel plus 10 extra cycles; you will need to finish drawing and then assert `done` at some point within that time budget.

Note that you are using CLOCK_50, the 50MHz clock, to clock your circuit. This differs from Lab 2, where you used a pushbutton switch for your clock.

Exhaustively test your design by writing a testbench and simulating everything in ModelSim (hint: look at the counters first). You will need to demonstrate and submit comprehensive testbenches both for the fillscreen module and the toplevel module, with separate RTL and netlist testbenches. Make sure your pixel colours are correct for the demo.

3.2 Task 3: Bresenham Circle Algorithm

The Bresenham Circle algorithm is an integer-only circle-drawing algorithm. The basic algorithm is described in the following pseudocode (modified from Wikipedia):

```
    draw_circle(centre_x, centre_y, radius):
    offset_y = 0
    offset_x = radius
    crit = 1 - radius
    while offset_y is less than or equal to offset_x:
        setPixel(centre_x + offset_x, centre_y + offset_y)   -- octant 1
        setPixel(centre_x + offset_y, centre_y + offset_x)   -- octant 2
        setPixel(centre_x - offset_x, centre_y + offset_y)   -- octant 4
```

```
setPixel(centre_x - offset_y, centre_y + offset_x)   -- octant 3
setPixel(centre_x - offset_x, centre_y - offset_y)   -- octant 5
setPixel(centre_x - offset_y, centre_y - offset_x)   -- octant 6
setPixel(centre_x + offset_x, centre_y - offset_y)   -- octant 8
setPixel(centre_x + offset_y, centre_y - offset_x)   -- octant 7
offset_y = offset_y + 1
if crit is less than or equal to 0:
    crit = crit + 2 * offset_y + 1
else:
    offset_x = offset_x - 1
    crit = crit + 2 * (offset_y - offset_x) + 1
```

In this task, you are to implement a circuit that draws a circle with its centre at specified coordinates, a specified radius, and a specified colour. The interface to this `circle` module is as follows:

Signal	Direction	Semantics
clk	input	clock
rst_n	input	active-low *asynchronous* reset
colour	input	drawing colour
centre_x	input	the x-coordinate of the centre
centre_y	input	the y-coordinate of the centre
radius	input	the radius in pixels
start	input	assert to start filling the screen
done	output	goes high once the entire circle is drawn
vga_*	output	outputs to the VGA adapter core

As with the `fillscreen` module, we will assert `start` and hold it high until your module asserts `done`.

Your `circle` module must work for *any* centre and radius inputs, *including* centres and radii where parts of the circle would be off the screen (in which case you must draw only the on-screen parts). Any pixels whose coordinates fall outside of the screen area must not be drawn.

The clock cycle budget you have for `circle` is the number of pixels that are actually drawn for the circle with the specified radius, plus 10 extra cycles. This budget includes pixels that *would have been* drawn if the circle fit on the screen. To receive full credit, you must assert `done` within the cycle budget.

The top-level module `task3` will, on reset (KEY3, active-low, asynchronous), fill the screen with black pixels and draw a pure-green circle with the centre at (80,60) and a radius of 40; the clock cycle budget for this is 19,200 plus your `circle` budget plus 10. Note that you must clear the screen to black even if on your board it is initially black; the initialization state of circuits cannot, in general, be relied upon.

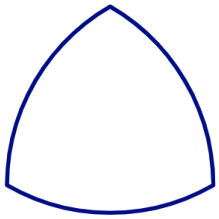
In designing the datapath for this algorithm, remember that you can only update one pixel per clock cycle. This means that each iteration through the loop will take at least 8 clock cycles since you might be drawing a pixel in each octant.

As before, you will need to demonstrate and submit a comprehensive testbenches both for the circle module and the toplevel module, separately for RTL and the netlist.

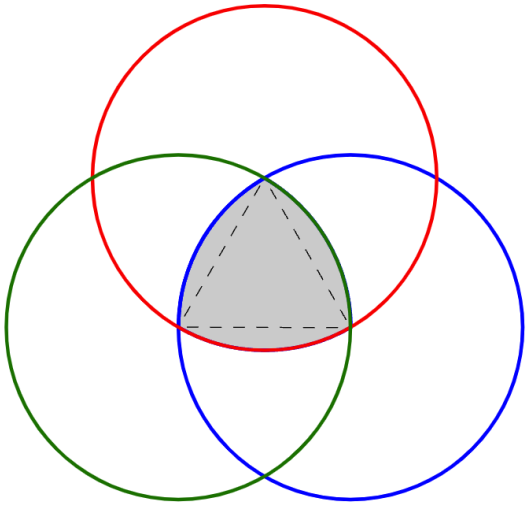
Be especially careful that your pixel positions are **exactly correct**. The demo for this lab will check for looking for overlapping, extra, and missing pixels. If you draw a circle in a different place than requested or with an incorrect radius, it is not likely that many pixels will overlap.

3.3 Task 4: The Reuleaux triangle

This task requires you to draw a Reuleaux triangle. This is a figure similar to a chubby equilateral triangle, where each triangle side is replaced by a section of a circle centred at the opposite corner:



You can think of this figure as the boundary of the intersection of three circles like this:



For the Reuleaux triangle, the *centre* is defined as the point equidistant from all three corners, while the *diameter* is defined as usual as any segment that passes through the centre and whose endpoints lie on the Reuleaux triangle. Observe that this diameter is the same as the length of one side of the inscribed triangle, and the same as the *radius* of the three circles used to construct it.

Your job is to write a module which draws this figure with the centre coordinates and the diameter as inputs (pointy end up, as in the figures above). You will probably want to use your circle design and carefully control when the pixels are drawn.

Any fractional coordinates that you may need to compute should be truncated down to the nearest integer.

The interface for module `reuleaux` is the same as the one for `circle` from Task 3, except we specify the *diameter* of the Reuleaux triangle instead of the radius of the circle.

The clock cycle budget you have for `reuleaux` is the number of cycles that would be required to fully draw all three circles used to construct the Reuleaux triangle, plus 15 cycles; this includes pixels that fall outside of the screen coordinates. The `done` signal that indicates you are finished must be asserted within this budget. (Note that this is *a lot* more than is needed to draw the actual triangle; you might want to think about how you would do it without wasting cycles on pixels you never draw.)

At the toplevel stage you will need to clear the screen to black and draw a green Reuleaux triangle with the centre at (80,60) and a diameter of 80, so that is what your top-level module `task4` must do on reset (KEY3, active-low, asynchronous). The cycle budget is 19,200 plus your `reuleaux` budget, plus 10 cycles.

As before, you will need to demonstrate and submit comprehensive testbenches both for the reuleaux module and the toplevel module, both for the RTL and the netlist. Be especially careful that your pixel positions are **exactly correct**.

4 Deliverables and Evaluation

4.1 Using Canvas

Ensure you don't include any extra files that will cause your compiler to fail.

Any template files we give you should be directly modified and submitted using **the same filename**.

Do not share your code or look at other codes. We will run academic integrity checks in the month of December for all the labs. If you are deemed to have cheated, we will follow the academic integrity procedures mandated by the department and UBC to the fullest extent.

4.2 Submission Rules for all students

Please follow these rules strictly.

1. You must not **rename any files** we have provided.
2. You must not **add** any files that contain unused Verilog code; this may cause compilation to fail.
3. Your testbench files must begin with `tb_` and **correspond to design file names** (e.g., `tb_rtl_foo.sv` and `tb_syn_foo.sv` for design `foo.sv`).
4. You must not have **multiple copies of the same module** in separate committed source files in the same task folder. This will cause the compiler to fail because of duplicate module definitions.
5. Your modules must not **rely on files from another folder** except for the VGA modules.
6. You must not copy the VGA files into the task folders.
7. You must not **alter the module declarations, port lists, etc.**, in the provided skeleton files.
8. You must not **rename any modules, ports, or signals** in the provided skeleton files.
9. You must not **alter the width or polarity of any signal** in the skeleton files (e.g., everything depending on the clock is posedge-triggered, and `rst_n` must remain active-low).
10. Your sequential elements must be triggered **only on the positive edge of the clock** (and the negative edge of reset if you have an asynchronous active-low reset). No non-clock (or possibly reset) signal edges, no negative-edge clock signals, or other shenanigans.
11. You must not add logic to the clock and reset signals (e.g., invert them). When building digital hardware, the clock and reset must arrive at exactly the same time to all your FFs; otherwise, your circuit will be slow and, at worst, not working.

If your code does not compile, synthesize, and simulate under these conditions (e.g., because of syntax errors, misconnected ports, or missing files), you will receive **0 marks**. You must submit separate testbench files for the RTL and the post-synthesis netlist; see the naming convention in the deliverables section.

4.3 Submitting with a partner

You do not need a partner, but you may have one if you like.

- If you have a partner, **both you and your partner need to submit the exact same zip file** using your respective Canvas logins. If only one of you submits, you both will get 0 marks.
- **Both partners** must **write the name of their respective partner as a comment** on their submission. **Failing to do so will result in 0 marks**.
- Your partner must be **from the same lab section** and **must be present during the grading process**.

- **You and your partner must arrive at the lab right at the beginning of the grading lab** and write your names on a sheet that the TA will provide. **If you or your partner arrive late, both you and your partner will get 0 marks.**
- The TA has finite hours, and we cannot wait for team members to arrive at arbitrary times, regardless of whether it is your fault or not. **So please choose your partner (if you want one) very carefully**.

5 Marking: Total 12 Marks

The marks are assigned in the following manner. *Nearly* 70% of your grade will be from the in-person demo for each task. The remaining (*nearly*) 30% of your grade will be from answering the questions about your code by your TA.

5.1 Task 2 [2 marks]

Deliverables in folder `task2`:

- Modified `task2.sv` (the toplevel)
- Modified `fillscreen.sv`
- Modified `tb_rtl_task2.sv`
- Modified `tb_rtl_fillscreen.sv`
- Modified `tb_syn_task2.sv`
- Modified `tb_syn_fillscreen.sv`
- Any other modified/added source/testbench files for your design

5.2 Task 3 [5 marks]

Deliverables in folder `task3`:

- Modified `task3.sv` (the toplevel)
- Modified `circle.sv`
- Modified `tb_rtl_task3.sv`
- Modified `tb_rtl_circle.sv`
- Modified `tb_syn_task3.sv`
- Modified `tb_syn_circle.sv`
- Any other modified/added source/testbench files for your design

5.3 Task 4 [5 marks]

Deliverables in folder `task4`:

- Modified `task4.sv` (the toplevel)
- Modified `reuleaux.sv`
- Modified `tb_rtl_task4.sv`
- Modified `tb_rtl_reuleaux.sv`
- Modified `tb_syn_task4.sv`
- Modified `tb_syn_reuleaux.sv`
- Any other modified/added source/testbench files for your design

6 Example: TCL-Based Simulation for a Virtual DE1-SoC Board

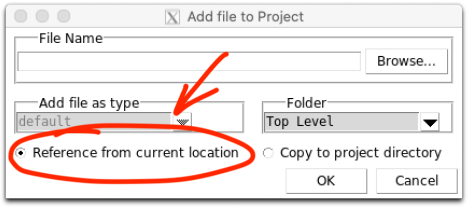
We have developed a "virtual" DE1-SoC that emulates a portion of the functionality of the real board as an example. This is present in the `de1-gui folder`. You can use this example to learn to use the `de1_vga_gui.tcl` script in the `vga-core folder`.

The virtual DE1-SoC consists of two parts. First, a Tcl file (de1_gui.tcl) that you need to load in ModelSim before simulation to implement the GUI. Second, s SystemVerilog file (de1_gui.sv) that you need to instantiate in your testbench to connect to the GUI from your design. The directory also contains a simple synthesizable design you can use to test that you are using the virtual board correctly (button_pusher.sv) and a testbench that shows you how to connect that design to the GUI.

Important: If you are debugging your circuit using the GUI, do not copy these files to your task folder, but reference them directly from the de1-gui folder ("Reference from current location" when you add the file). The de1_gui.sv file does not work without the de1_gui.tcl interface, and it is not synthesizable.

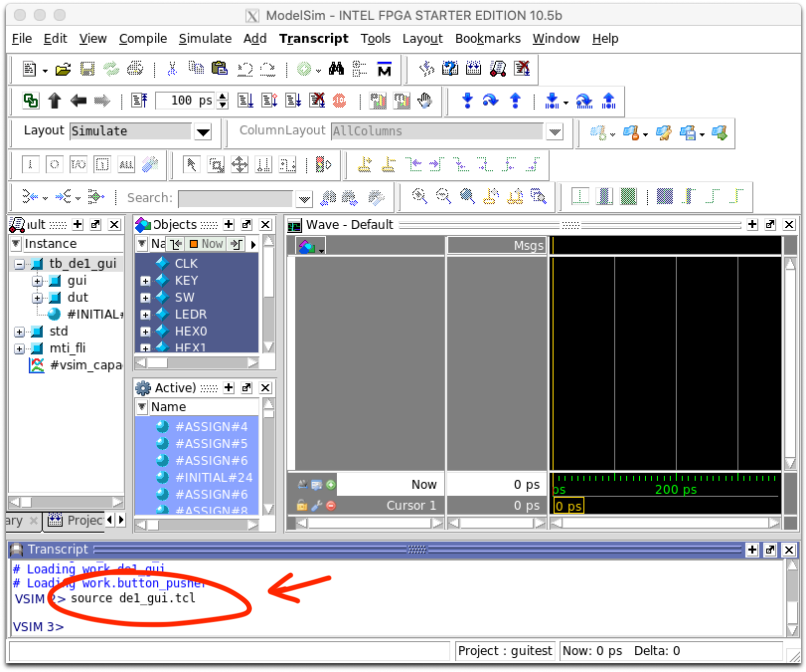
6.1 Setting up the simulation

Launch ModelSim and create your project as covered in the Tutorial. Add button_pusher.sv, de1_gui.sv, and tb_de1_gui.sv to the project. To prevent accidentally submitting these files for marking, add files by referencing them in the existing directory rather than copying them to your working directory:



Compile the entire design as usual, and load the compiled design by double-clicking on the tb_de1_gui in the Library tab.

Next, load the de1_gui.tcl file by issuing the command source de1_gui.tcl in the Transcript frame:



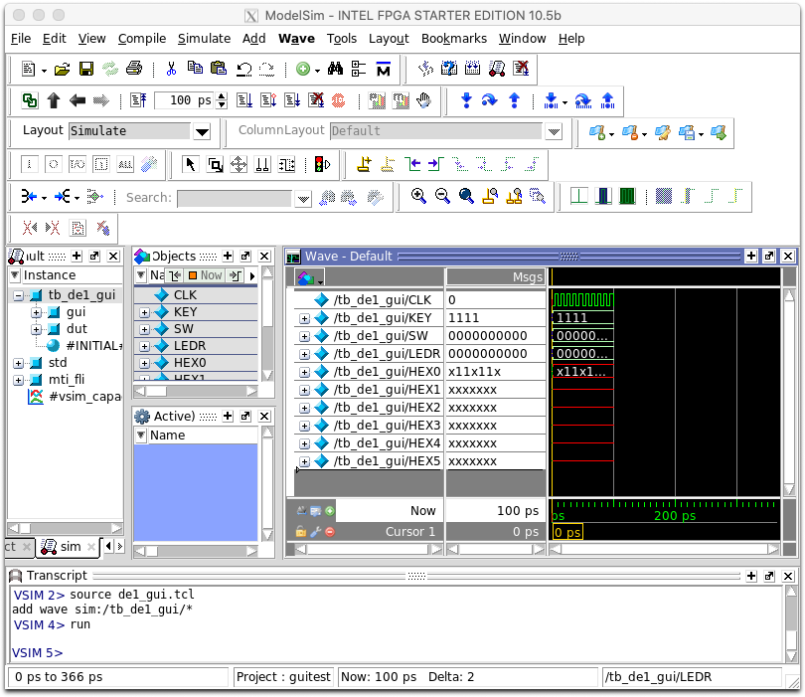
You should see a new window that shows the switches, buttons, LEDs, and 7-segment displays of the DE1-SoC board:

All switches and buttons are in the "off" position (i.e., the SW signals are all 0, and the KEY signals are all 1 as they are active-low). The LEDs simulate the dimly lit state they appear in when they are not driven on the real board. Move the switches

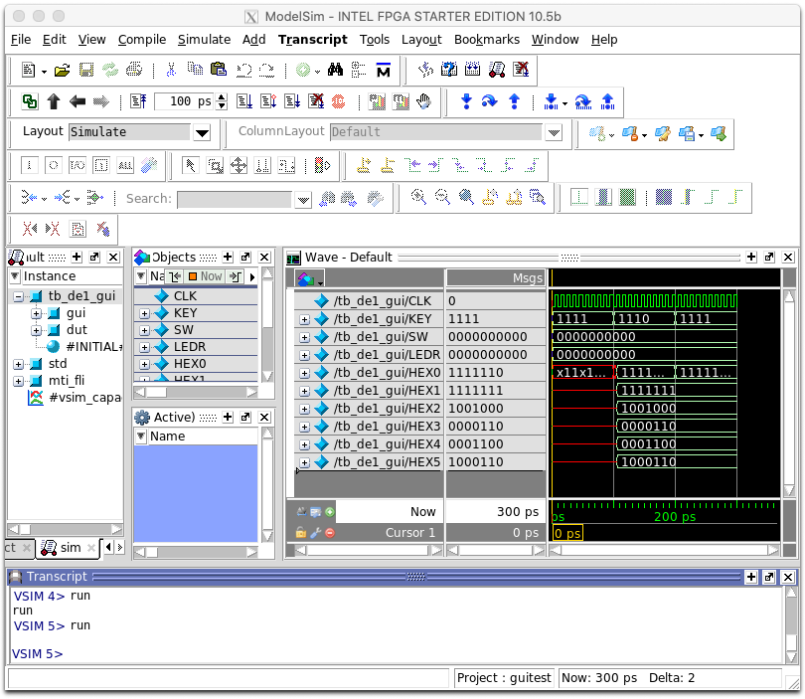
and push the buttons to see what you can do. Note that unlike on the physical board, buttons stay pushed if you click on them to let you gradually advance the simulation without holding the mouse button; clicking the buttons again will release them.

6.2 Simulating RTL

Now add the testbench signals to a waveform and simulate for 100ps. Next, reset your design. The button_pusher design uses KEY0 as reset, so click KEY0 to push it in, to advance for 100ps, click KEY0 again to release it, and advance for 100ps:



You should see that the virtual board display has changed:

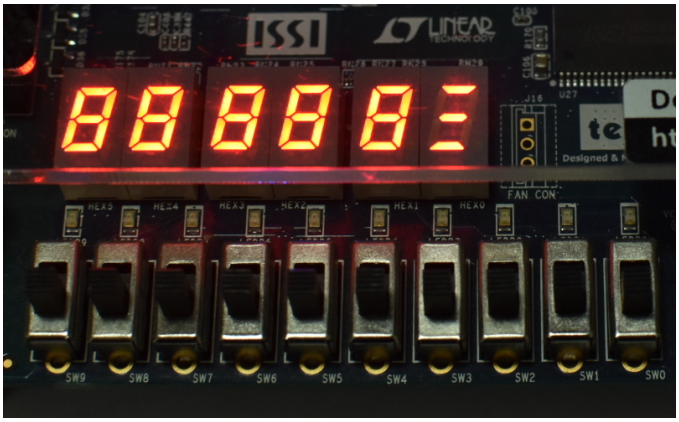


Try out the various switches and buttons and watch the design react. Remember that you have to advance the simulation in the

ModelSim window manually. Otherwise, the simulated hardware will not react to your GUI inputs.

6.3 Simulation versus The Real Thing

If you have a physical DE1-SoC, synthesize button_pusher and download it to your FPGA. You will see that the initial state of your simulation differs from the real board, which will, in all likelihood, have the seven-segment displays lit:



Once you reset the design using KEY0, the two should behave correspondingly.

Why is this happening? In your RTL simulation, several signals — in particular the HEX0...HEX5 registers that drive the 7-segment display drivers — are undefined ('x) and acquire logical 0 or 1 values only after reset. But in the real hardware, there is no such thing as undefined, so in the real hardware, those registers will have some logical value; most likely, this is 0, which causes the 7-segment displays to light up (as they are active-low). While you cannot generally rely on this initialization, it explains the discrepancy between the simulation and the hardware.