



Department of Electronic Engineering

Assessments 2019/20

ELE00067M

Digital Design

This assessment (**Lab Reports 1 & 2**) contributes **10%** of the assessment for this module.

Clearly indicate your **Exam Number** on every separate piece of work submitted.

Unless the assessment specifies a group submission, you should assume all submissions are individual and therefore should be your own work.

All assessment submissions are subject to the Department's policy on plagiarism and, wherever possible, will be checked by the Department using Turnitin software.

Submission is via VLE and is due by **12:00** on **24 October 2019 (Autumn Term, Week 4, Tuesday)**. Please try and submit early as any late submissions will be penalised.

Please remember that if this is your first year of study, you need to complete the mandatory Academic Integrity Tutorial <http://www.york.ac.uk/integrity/>

ELE00067M Digital Design

Laboratories: Session 1

Implementation of basic VHDL designs in a Xilinx FPGA

Report formatting:

There is no formal report structure – you will be marked on the items listed within each lab script. Most reports will include code printout and simulation screenshots – see Appendices for guidelines.

The reports for labs 1 and 2 must be handed in, together in a single pdf file, via the VLE by the deadline indicated on the front page of the script.

Each task should start on a new page, containing all the material required in the order specified. The exam number should be printed on each page.

The PDF file should be named **Yxxxxxxx_DDMSc_Labs1-2_.pdf**, where the Yxxxxxxx is the exam number. Exactly one file should be submitted, containing the entire submission.

In all cases, read carefully the instructions on the VLE submission page. Failure to follow the instructions could lead to your assignment not being marked and in any case to a mark penalty.

Submission weight on module mark: 10%

Mark breakdown [80 marks]:

- General issues (e.g. documentation, layout and comments, structural issues): 30 marks
- Task 1A (e.g. VHDL code, testbench, simulation): 5 marks
- Task 1B (e.g. VHDL code, testbench, simulation): 10 marks
- Task 2A (e.g. VHDL code, testbench, simulation): 15 marks
- Task 2B (e.g. VHDL code, testbench, simulation): 20 marks

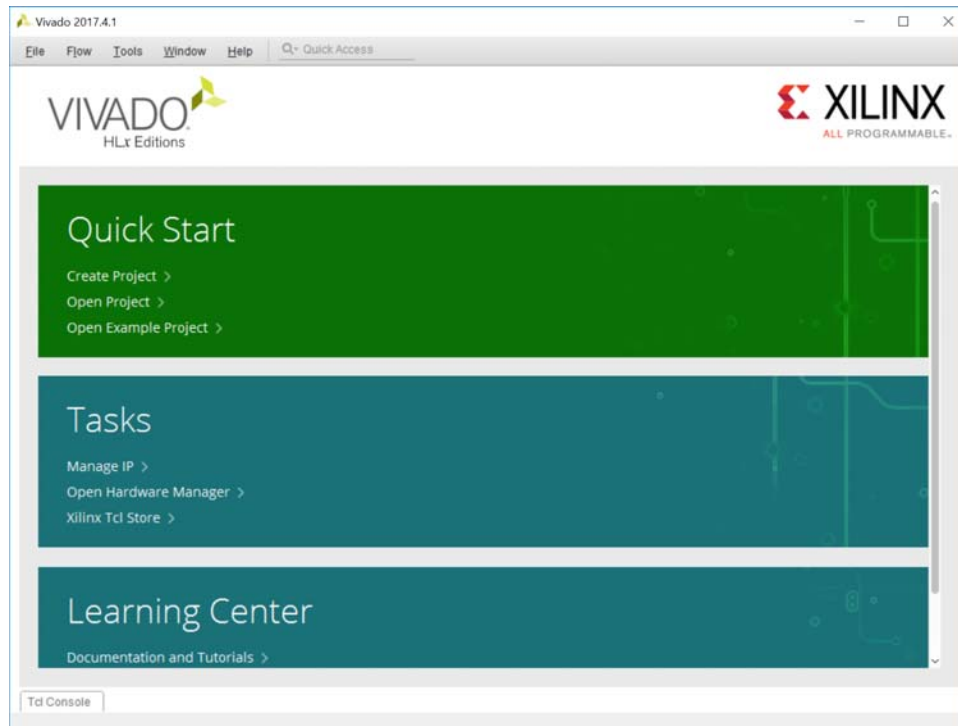
Task A: Implementation of a simple design

This task will guide you through the creation of a very simple circuit that can be implemented just using LEDs and sliders: a combinational full adder that sums two bits and a carry in and outputs a sum and a carry out.

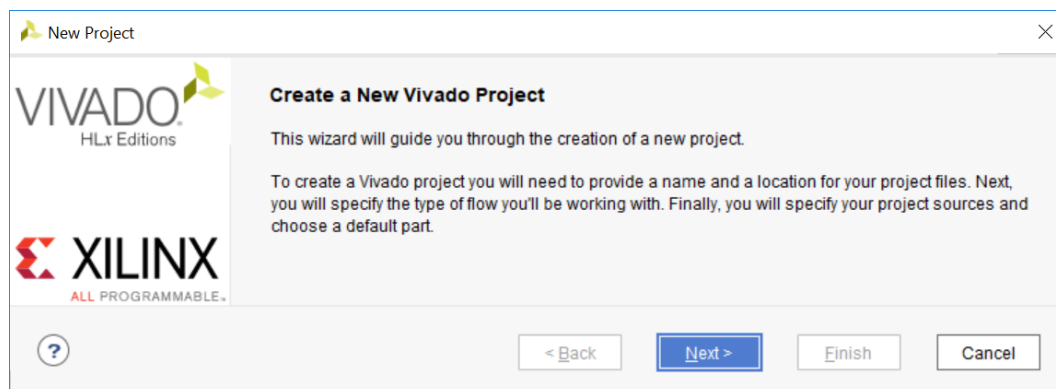
Step 1. Project creation

Start by opening your M drive and creating a folder (in the root directory) called “VivadoProjects”.

Launch “Vivado 2017.4”, located in the “Electronics Xilinx Design Tools” folder in the App list (you really want to create a desktop shortcut!).



In the “Quick Start” pane, click on “Create Project. This will launch the Project Creation “wizard”.



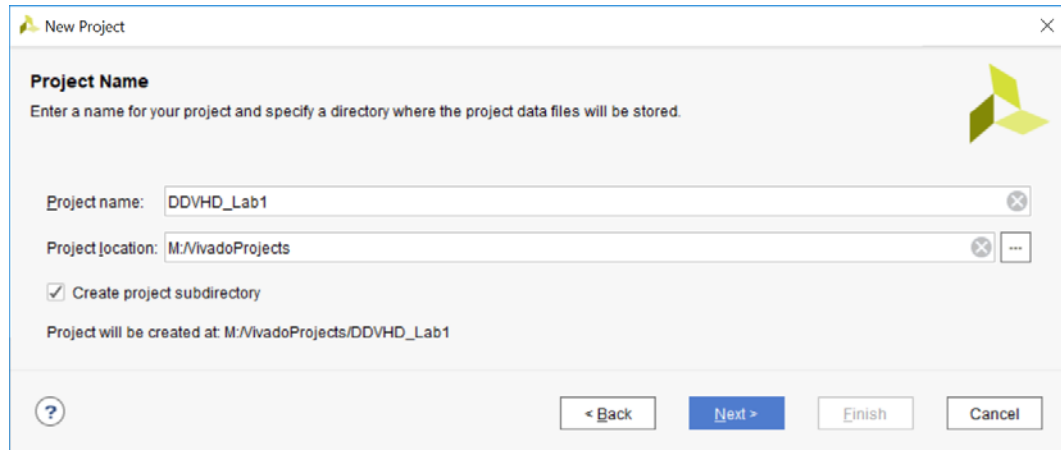
Click “Next”. Specify a Project name.

IMPORTANT: Xilinx software is known to be “tricky” where paths and filenames are concerned. For all your filenames/paths, it is **strongly** suggested that all your files and directories:

- **Do NOT use any spaces**
- **Start with a letter (A-Z, a-z) and use only alphanumeric characters (A-Z, a-z, 0-9) and underscores (_)**

As a project location, select the “VivadoProjects” folder you previously created. Make sure that the “Create project subdirectory” option is selected.

Make sure to use a meaningful project name. In the example below, I created a project called “DDVHD_Lab1”.



New Project

Project Name
Enter a name for your project and specify a directory where the project data files will be stored.

Project name: DDVHD_Lab1


Project location: M:/VivadoProjects

☒ Create project subdirectory

Project will be created at: M:/VivadoProjects/DDVHD_Lab1

< Back Next > Finish Cancel

Press “Next”. On the following page, select “RTL Project” and tick “Do not specify sources at this time”.



New Project

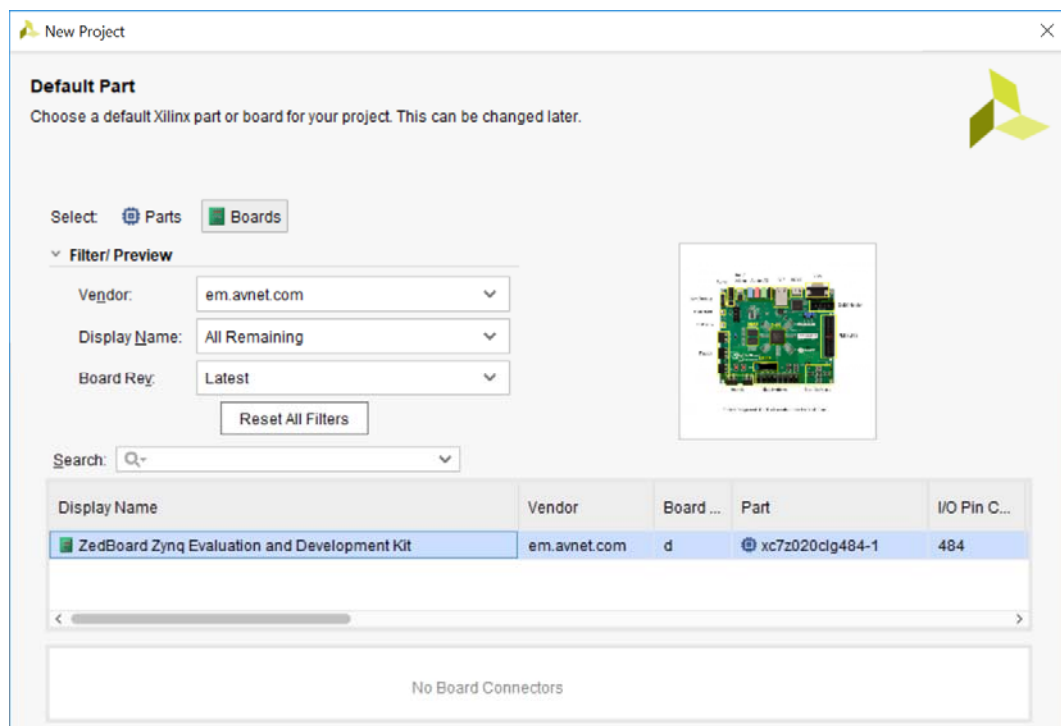
Project Type
Specify the type of project to create.

☒ **RTL Project**
You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.

☒ Do not specify sources at this time

☐ **Post-synthesis Project**: You will be able to add sources, view device resources, run design analysis, planning and implementation.

Press “Next”. On the following page, click on “Boards” and select “em.avnet.com” as Vendor. This should provide a single choice as board (“ZedBoard Zynq Evaluation and Development Kit”. Select it.



New Project

Default Part
Choose a default Xilinx part or board for your project. This can be changed later.

Select: ☒ Parts ☒ Boards

Filter/Preview



Vendor: em.avnet.com

Display Name: All Remaining

Board Key: Latest

Reset All Filters

Search: Q

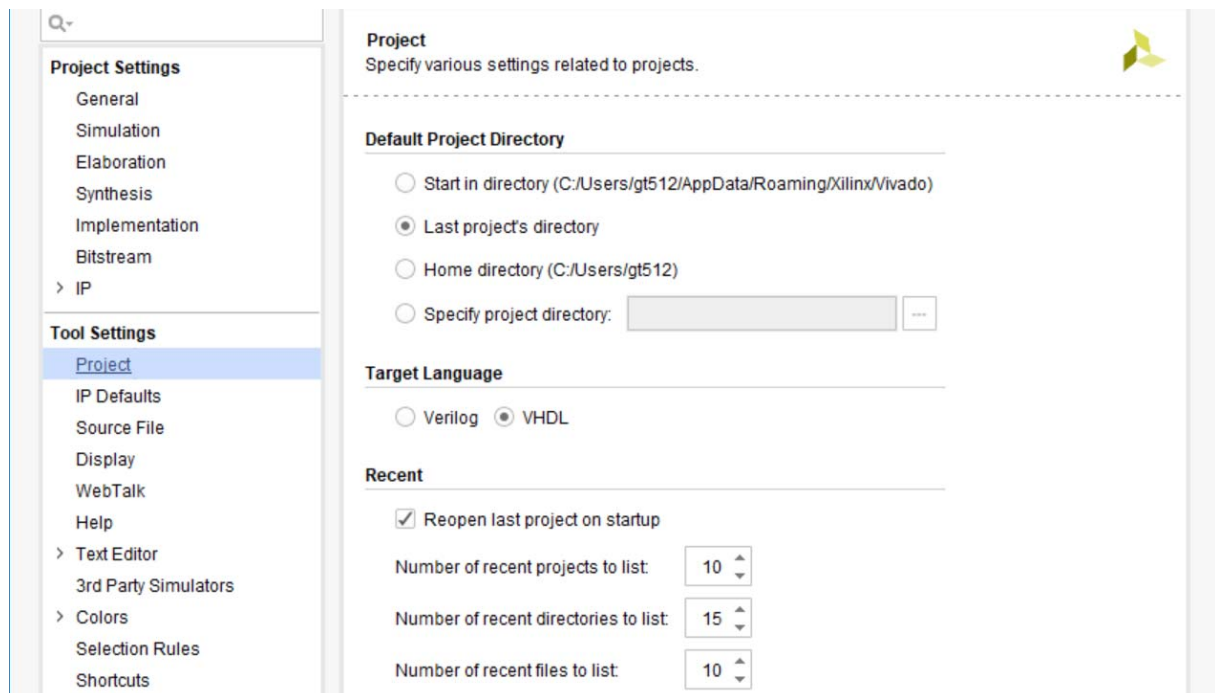
Display Name	Vendor	Board ...	Part	I/O Pin C...
 ZedBoard Zynq Evaluation and Development Kit	em.avnet.com	d	 xc7z020clg484-1	484

No Board Connectors

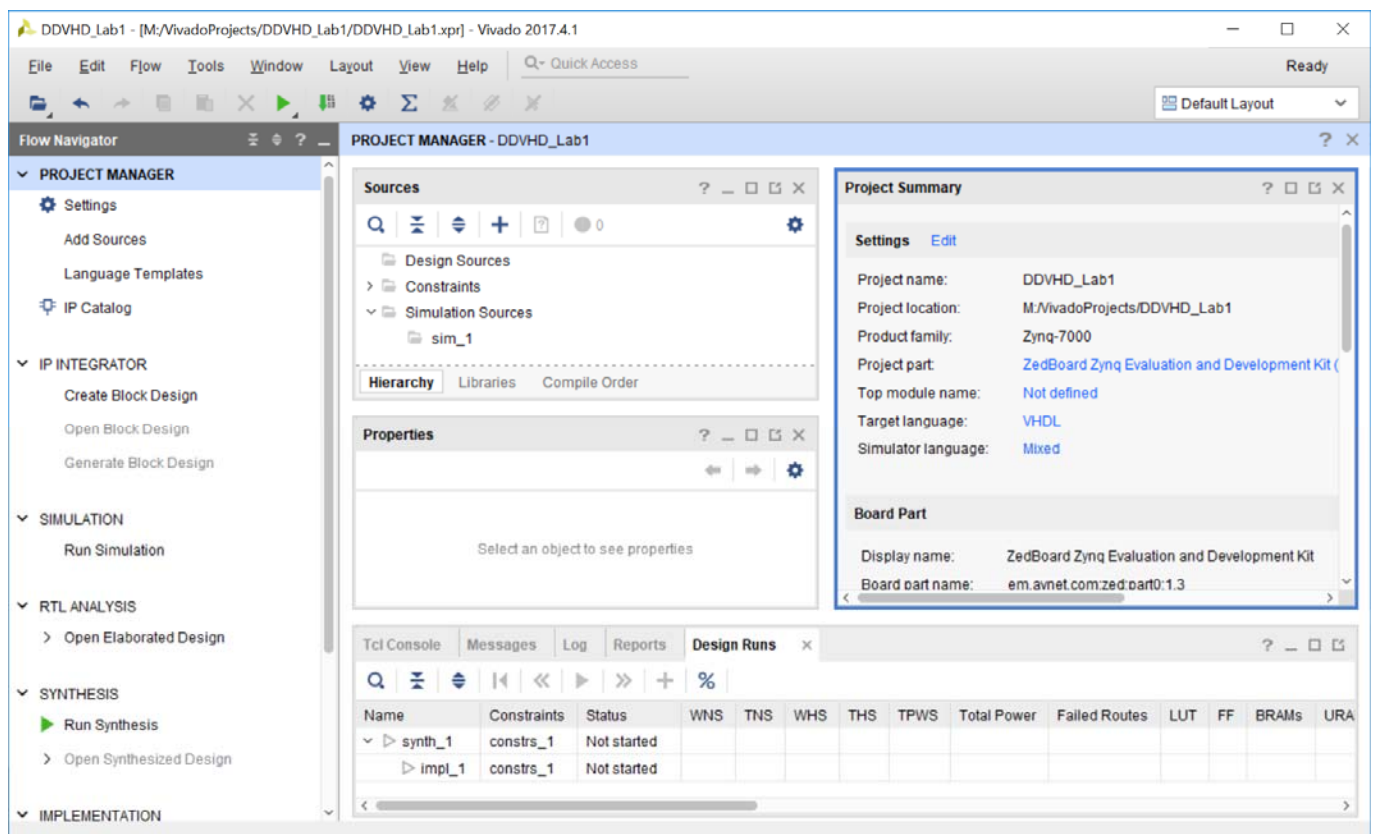
Press “Next”, then “Finish”. After a few seconds, you should see the main Vivado design flow interface. In the “Project Summary” pane, you will probably see that the “target language” is set to “Verilog” (if it is already set to “VHDL”, you can skip to step 2).

Click on the word “Verilog” (in blue). This will open a new window that allows you set the “Target language” to “VHDL”.

Do so, then click on “Project” under “Tool Settings” and change the Target Language to VHDL there as well. Also, set the “Reopen last project at startup” flag. Hopefully, this will set VHDL as the default language and all projects you create in the future will default to VHDL rather than Verilog (if they do not, repeat the above procedure). When you have changed all three parameters choices, click OK

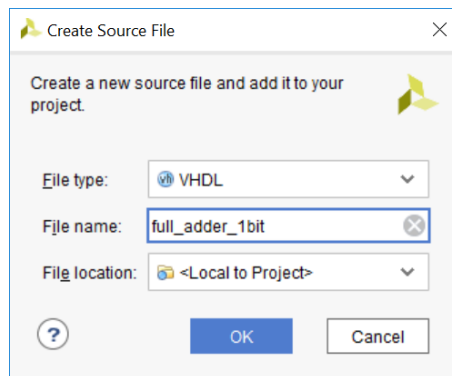


The Vivado interface should now look something like this:

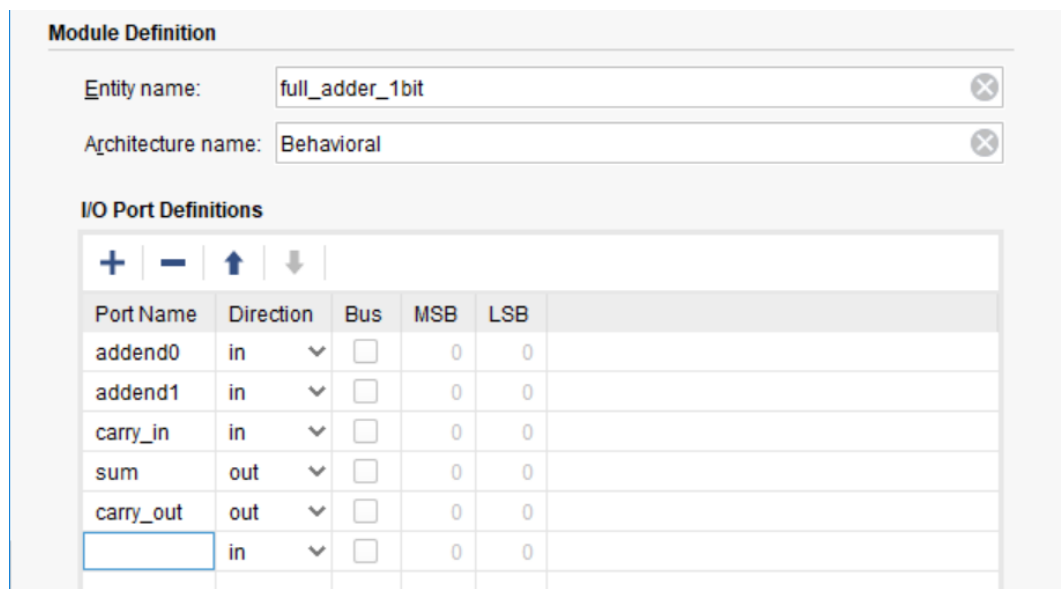


Step 2. Adding new VHDL modules

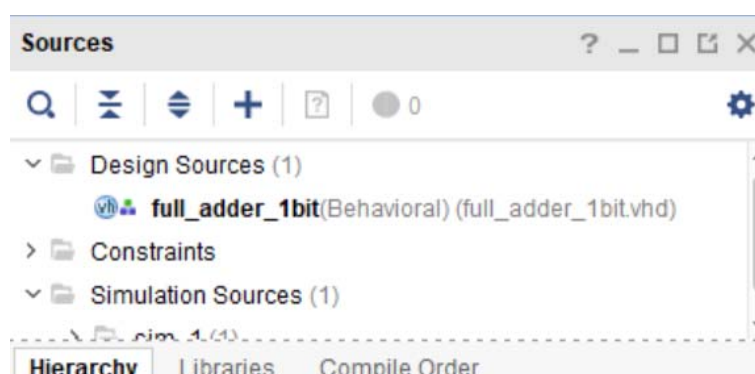
In the “Flow Navigator” pane, click on “Add Sources” (under “Project Manager”). Make sure that “Add or create design sources” is selected, then click “Next”. In the next window, select “Create File” and give a name to the file (remember the naming conventions outlined above!). In this lab, we will create a 1-bit full adder, so give it a meaningful name (naming of components, signals, ports, etc. should *always* be meaningful, and this will affect marking!). For example, call the file “full_adder_1bit”.



Click “Finish”. In the next window (“Module Definition”), you can specify the I/O ports for your components, including their direction (“in” or “out” – you should ignore “inout” for this module). Note that, if the I/O signal represents a bus (which is not the case in this example), you can click on the “Bus” box, and assign the MSB and LSB, defining the size of the bus. **Remember that any port definition you set at this stage can be changed later when you edit the VHDL file.** Set the ports as below.

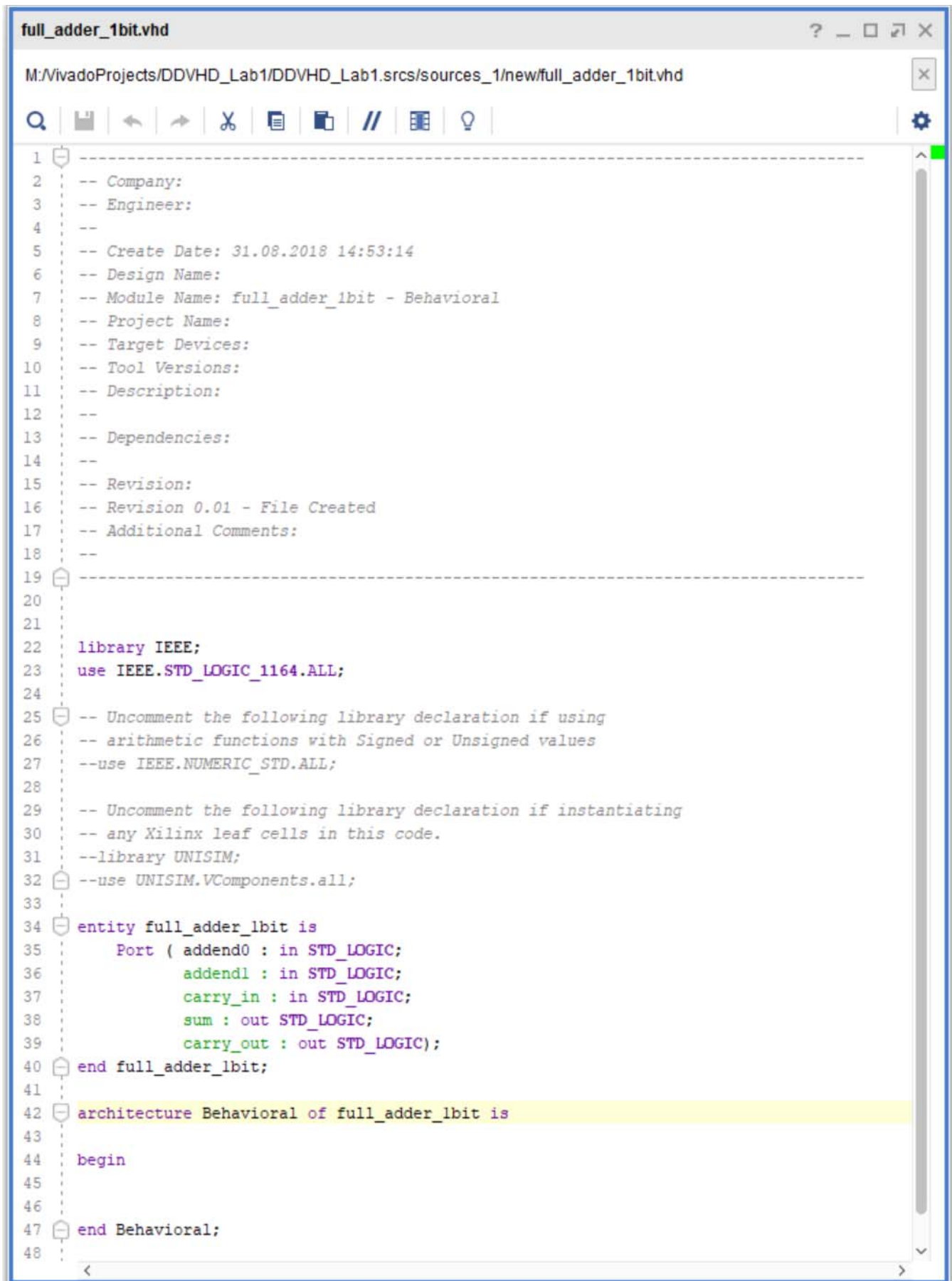


Click “OK”. After a few seconds, you should see the new component in the “Sources” pane



Double-click the component. A new pane, displaying an editable text file, should open next to the “Project Summary”. It might be a good idea to click on the “Float” icon in the top right corner of the pane to spawn a new window that will allow you to edit the text more comfortably.

The text window should display a template for the source code, including the I/O signals you have defined. You can now use this as the basis for your VHDL component by adding functionality.



```
1  -----
2  -- Company:
3  -- Engineer:
4  --
5  -- Create Date: 31.08.2018 14:53:14
6  -- Design Name:
7  -- Module Name: full_adder_1bit - Behavioral
8  -- Project Name:
9  -- Target Devices:
10 -- Tool Versions:
11 -- Description:
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19 -----
20
21
22 library IEEE;
23 use IEEE.STD_LOGIC_1164.ALL;
24
25 -- Uncomment the following library declaration if using
26 -- arithmetic functions with Signed or Unsigned values
27 --use IEEE.NUMERIC_STD.ALL;
28
29 -- Uncomment the following library declaration if instantiating
30 -- any Xilinx leaf cells in this code.
31 --library UNISIM;
32 --use UNISIM.VComponents.all;
33
34 entity full_adder_1bit is
35     Port ( addend0 : in STD_LOGIC;
36           addendl : in STD_LOGIC;
37           carry_in : in STD_LOGIC;
38           sum : out STD_LOGIC;
39           carry_out : out STD_LOGIC);
40 end full_adder_1bit;
41
42 architecture Behavioral of full_adder_1bit is
43
44 begin
45
46
47 end Behavioral;
48
```


Start by removing all the comments (greyed-out text starting with "--") – more on this later. Then, leaving the entity declaration as it is, in the architecture section of the code (assuming your component name is as defined above) replace:

```
architecture Behavioral of full_adder_1bit is

begin

end Behavioral;
```

with:

```
architecture Behavioral of full_adder_1bit is

signal P,G,Cprop: STD_LOGIC;

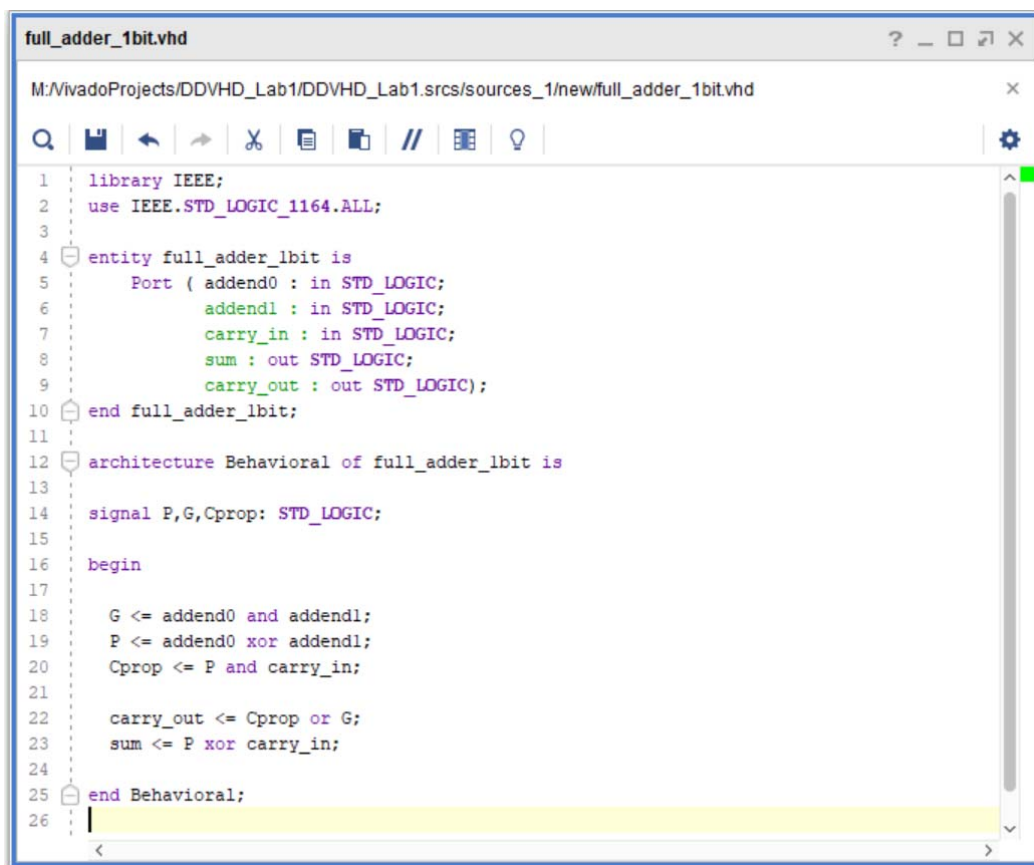
begin

    G <= addend0 and addendl;
    P <= addend0 xor addendl;
    Cprop <= P and carry_in;

    carry_out <= Cprop or G;
    sum <= P xor carry_in;

end Behavioral;
```

The editor window should now display the following (note the code highlights):



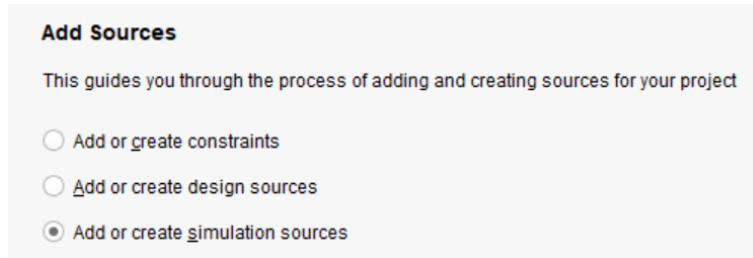
Remember to click the “Save” icon (or use Ctrl+S) or your changes will not be taken into account!

You can now close the text editor if you wish (or click on the “Dock” icon at the top left to move it back to the main interface).

Step 3. Simulation

It is now time to simulate your code to verify the operation of your component. For this, it is necessary to create a *VHDL testbench*.

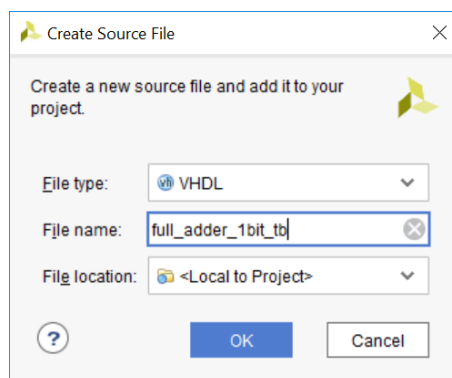
Click again on “Add Sources” in the Flow Navigator, but this time in the following window select “Add or create simulation sources”.



Click “Next” and in the following window select “Create File”. Assign a name to the testbench.

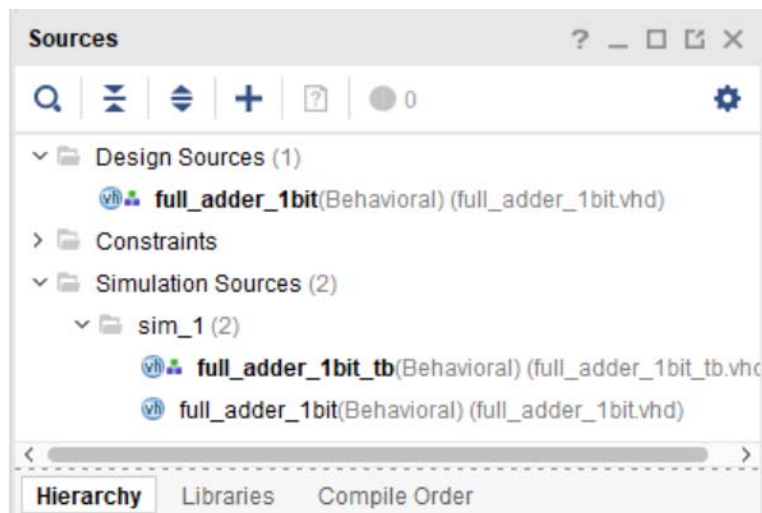
IMPORTANT: By convention (and to simplify your life later on), all testbench files should end with the string “_tb”. This will affect marking.

Since the testbench applies to the “full_adder_1bit” component, an appropriate name would be “full_adder_1bit_tb”.

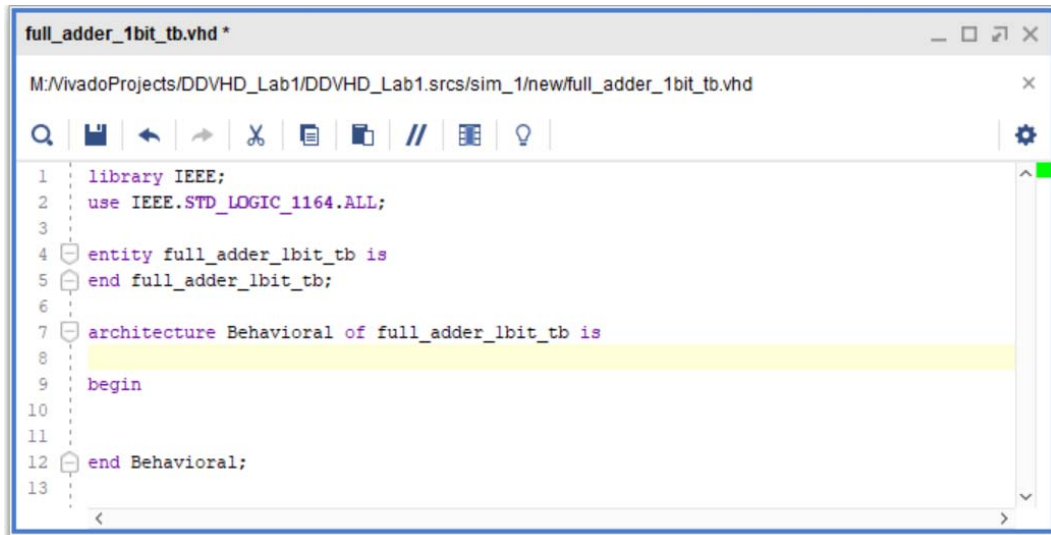


Click “Finish”, then “OK” in the next window, without defining any I/O ports (since this is a testbench and not a component). Note that you might need to confirm this through a pop-up window.

Your “Sources” pane should now include your new file under “Simulation Sources / sim_1”.



The new file should also be open in a text editor pane. As for the design source, float the editor, then remove the comments. You should be left with:



Three steps are required to implement any testbench (do not worry if you don't understand some of this, as we will cover these concepts in lecture next week):

- 1) **Internal signals (wires) must be created to connect to the I/O ports of the entity you want to test.** The signals should always have the exact same names as the I/O ports.

In the space between “architecture” and “begin”, insert the following lines:

```
signal addend0, addend1: STD_LOGIC;
signal carry_in: STD_LOGIC;
signal sum : STD_LOGIC;
signal carry_out : STD_LOGIC;
```

- 2) **The entity you want to test (the UUT, for *Unit Under Test*) must be *instantiated* as a component in your testbench**, connecting its I/O ports to the signals you created in step 1.

Immediately below “begin”, insert the following code (note that the first line might indicate an error... it is actually a bug of the text editor, so ignore it):

```
UUT: entity work.full_adder_1bit
  PORT MAP (
    addend0 => addend0,
    addend1 => addend1,
    carry_in => carry_in,
    sum => sum,
    carry_out => carry_out);
```

- 3) **Finally, a *process* should be created to define the sequence of input stimuli (aka *test patterns*) that you will use to verify the behaviour of the UUT, always preceded by a 100ns-long wait (more on this next week) and usually followed by an infinite wait (unless you want the sequence to repeat over and over).**

The choice of test patterns will be a significant topic for discussion in this module (and is indeed a crucial part of logic design), but in this case the simplicity of the UUT allows for *exhaustive* testing of *all possible input combinations*.

The following code (which should be pasted below the UUT instantiation) contains an *example* of the first two test patterns for this circuit, sent to the UUT at 10ns intervals. **Up to you to complete the testbench with all other possible combinations!**

```

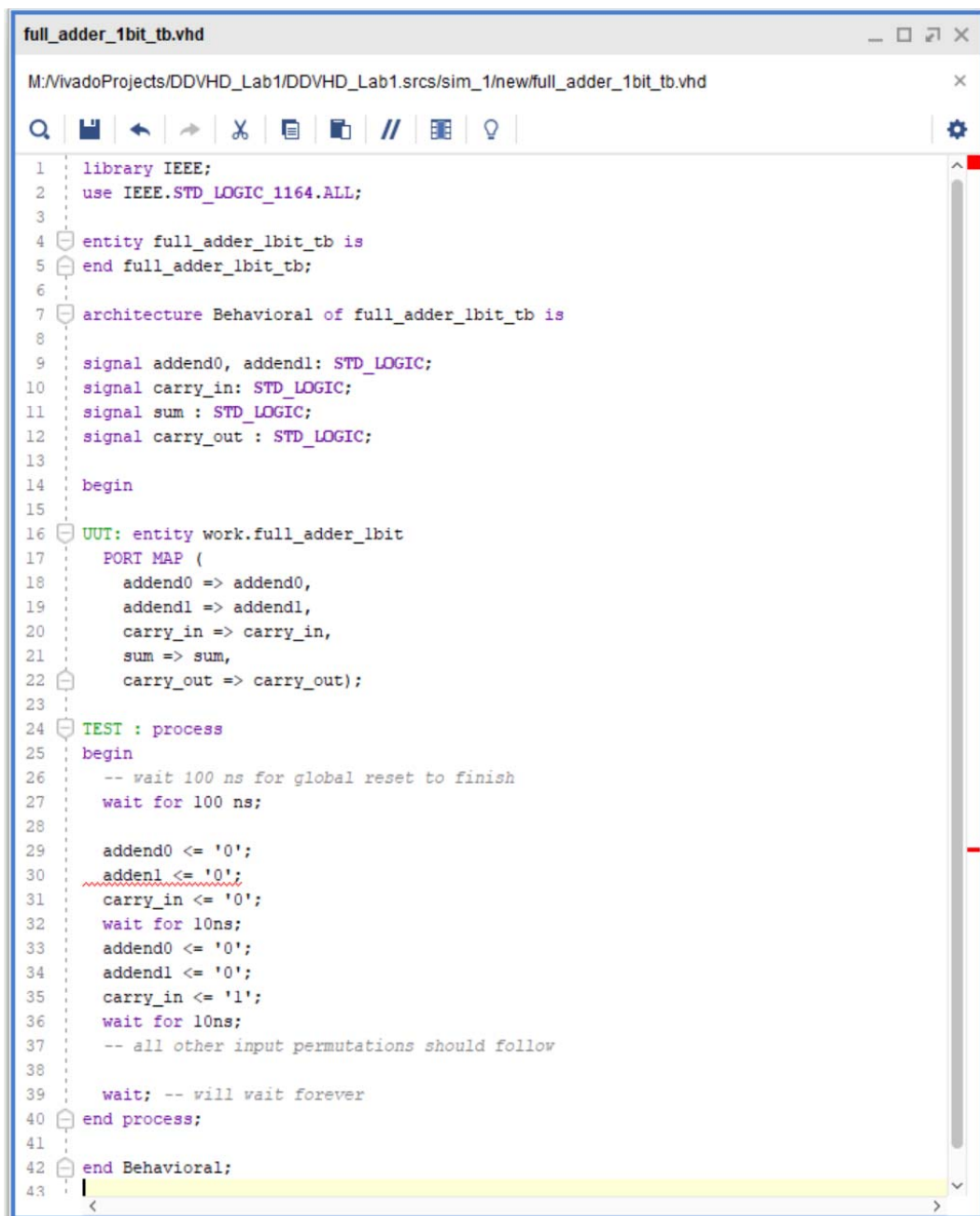
TEST : process
begin
    -- wait 100 ns for global reset to finish
    wait for 100 ns;

    addend0 <= '0';
    addend1 <= '0';
    carry_in <= '0';
    wait for 10ns;
    addend0 <= '0';
    addend1 <= '0';
    carry_in <= '1';
    wait for 10ns;
    -- all other input permutations should follow

    wait; -- will wait forever
end process;

```

Your final code should look as below (until you add the other permutations of the inputs).



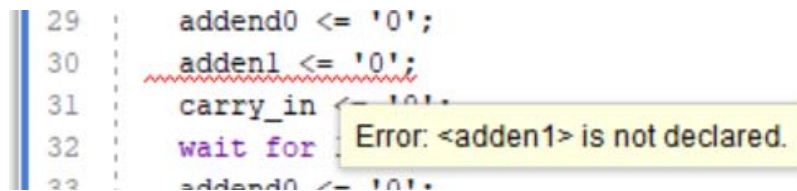
```

full_adder_1bit_tb.vhd
M:/VivadoProjects/DDVHD_Lab1/DDVHD_Lab1.srcs/sim_1/new/full_adder_1bit_tb.vhd

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity full_adder_1bit_tb is
5  end full_adder_1bit_tb;
6
7  architecture Behavioral of full_adder_1bit_tb is
8
9      signal addend0, addend1: STD_LOGIC;
10     signal carry_in: STD_LOGIC;
11     signal sum : STD_LOGIC;
12     signal carry_out : STD_LOGIC;
13
14     begin
15
16     UUT: entity work.full_adder_1bit
17         PORT MAP (
18             addend0 => addend0,
19             addend1 => addend1,
20             carry_in => carry_in,
21             sum => sum,
22             carry_out => carry_out);
23
24     TEST : process
25     begin
26         -- wait 100 ns for global reset to finish
27         wait for 100 ns;
28
29         addend0 <= '0';
30         addend1 <= '0';
31         carry_in <= '0';
32         wait for 10ns;
33         addend0 <= '0';
34         addend1 <= '0';
35         carry_in <= '1';
36         wait for 10ns;
37         -- all other input permutations should follow
38
39         wait; -- will wait forever
40     end process;
41
42 end Behavioral;
43

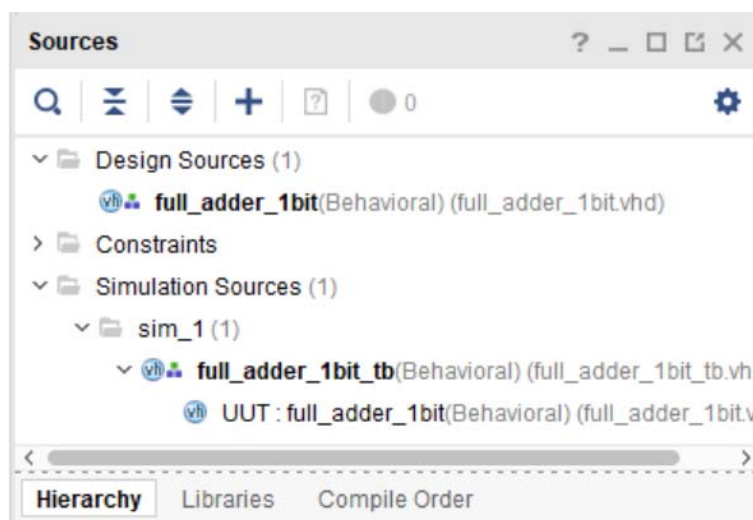
```

You will probably have noticed the red line under one of the VHDL assignments. As an example of the use of the syntax debugging functions of Vivado, I introduced an error in the text you pasted. I (intentionally, I swear!) mis-typed “addend1” as “adden1” in the second line of the test patterns. This was picked up by the debugger, as there is no “adden1” signal defined in the code. Hovering over the line with the mouse highlights an error to the effect that “adden1 is not declared” (i.e. does not exist).

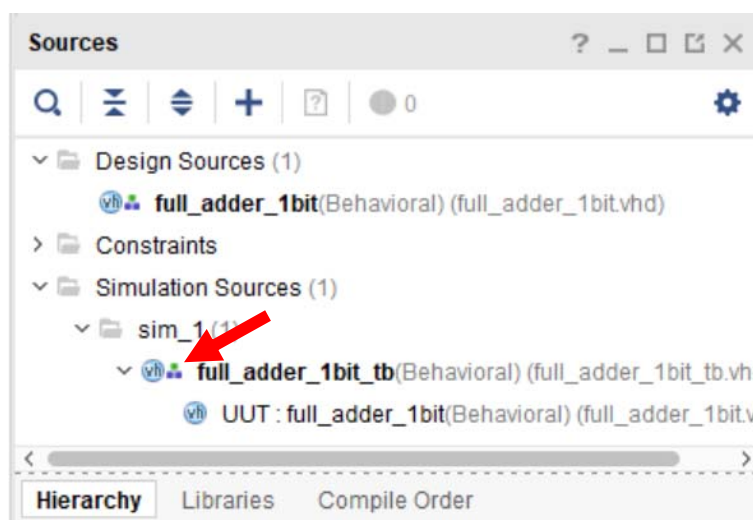


Do NOT fix the typo! We will use in the next step to see more “advanced” debug features.

Save the testbench. You will notice a change in the “Sources” pane: your original design (“full_adder_1bit”) should now appear as a sub-entry of your testbench (“full_adder_1bit_tb”). This testifies that the connection between the testbench and the UUT has been successful and should always be double-checked.



Select the testbench you have created and make sure that your testbench is the top simulation module (i.e. it has the icon highlighted in the screenshot below). **This is important, as the simulation will always run on the top module – keep that in mind for later labs where you might have multiple testbenches!**



In the Flow Navigator, select “Run Simulation”, then “Behavioral Simulation” (which should be the only possible option at this stage).

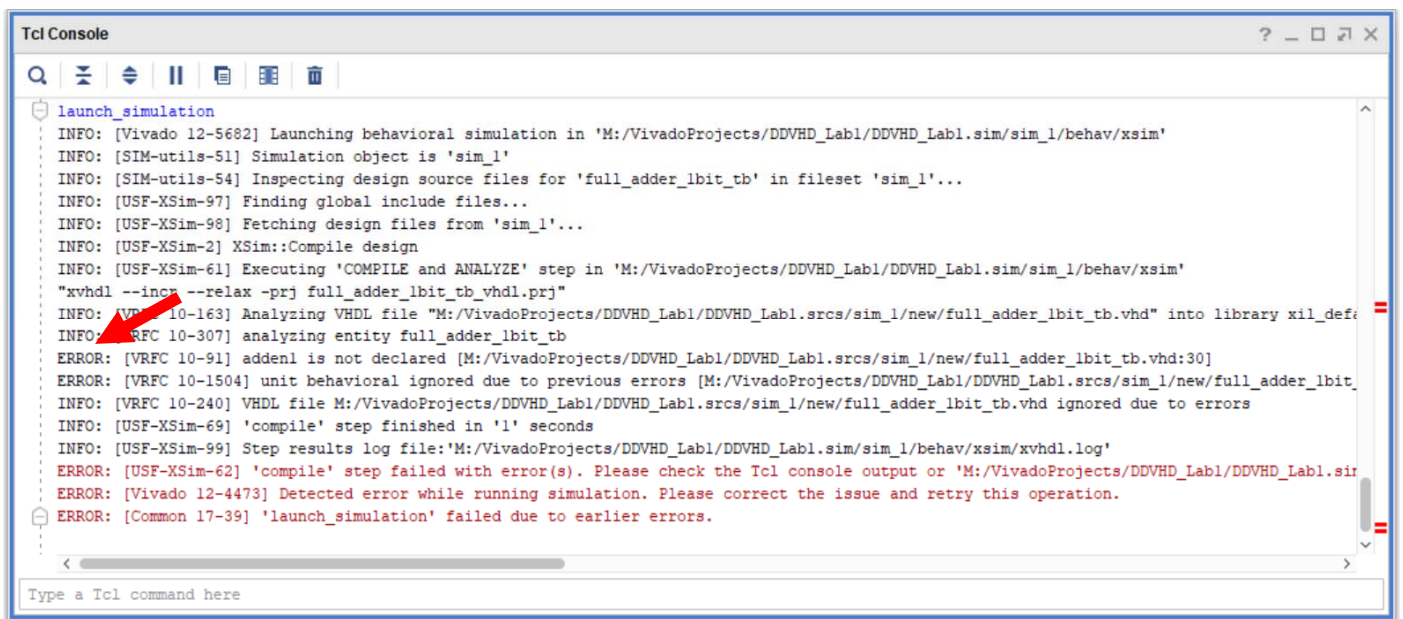
Because of the typo, you will now see an error window pop up. Click OK. Another popup will display the error messages. In typical Xilinx fashion, the messages are not very helpful... just about the only useful information is that the *compile* step failed. Click OK again to get rid of the window.

At the bottom of the main window, you should be able to see the all-important “Tcl Console” (if it is not already selected, click the tab on the top left of the bottom pane). You might need to scroll/expand the pane, but you should be able to visualise the console outputs from the moment you launched the simulation.

The console will contain several lines, most of which will be labelled “INFO” (which are OK), some that might be “WARNING” (not in this case, but it might be something to watch out for), and, if you are looking at this for debugging, some “ERROR”. You always want to spot the first error (as many of the following ones could well be a consequence of the first and disappear when it is fixed). In this case, the first error is:

```
adden1 is not declared
[M:/VivadoProjects/DDVHD_Lab1/DDVHD_Lab1.srcs/sim_1/new/full_adder_1bit_tb.vhd:30]
```

Which tells you that the error is caused by an undeclared signal in line 30 of the file specified (you knew that, of course, but that will not always be the case).

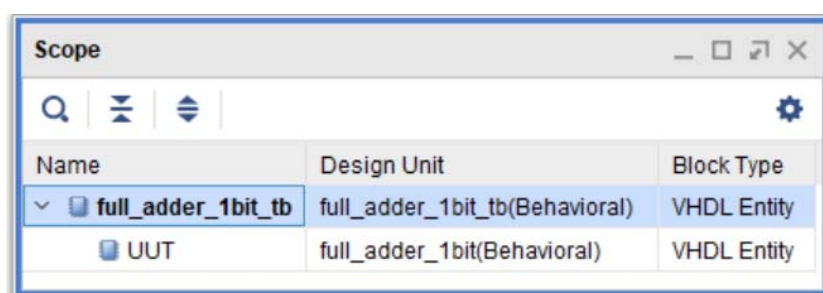


Now fix the typo in the testbench (remember to save!) and re-launch the simulation. Hopefully, this time the error message pop-up will not appear. Instead, the main window should now display the simulator interface (probably laid out in a horrible way that actually does not show anything useful, but we’ll fix that).

Incidentally, if you want to get back to the previous interface, you can click on “Project Manager” in the Flow Navigator (and on “Simulation” to switch back).

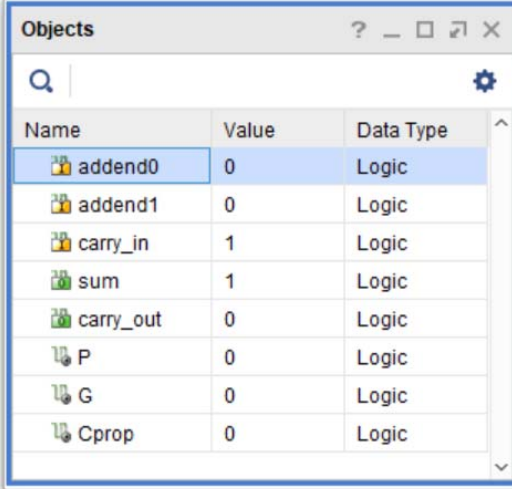
In addition to the Tcl console, the simulation interface consists essentially of three components (panes):

- 1) The *Scope* pane displays the hierarchy of components in your design, starting with testbench at the top. In this lab, there is a single entity in addition to the testbench (the “full_adder_1bit”) but this will change as the circuits become more complex and start using sub-components. Clicking on the “>” symbol next to the entity names allows you to expand/hide all the sub-components.



2) The *Objects* pane displays all the signals and ports of the entity selected in the *Scope* pane. For example, the screenshot below shows the objects that appear when the “full_adder_1bit” is selected. Input ports, output ports, and internal signals are all displayed, with a different icon for each type. It also displays the value of the signal at the time specified in the Simulation (which can be useful at times, but to be honest generally is not).

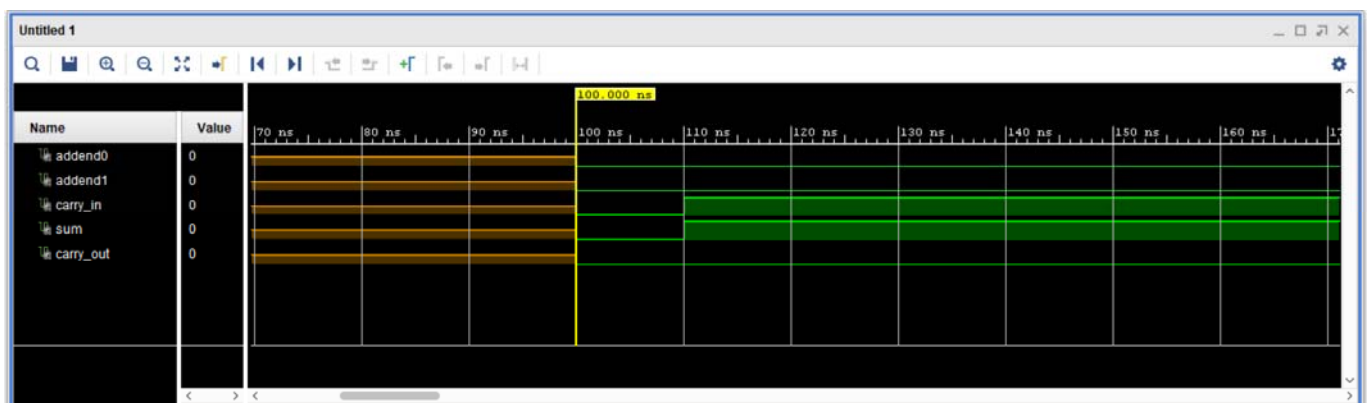
Note that this pane is extremely important for simulation as it allows (as we will see) to select any internal signal for display in the simulation. This is absolutely essential for debugging a circuit!



Name	Value	Data Type
addend0	0	Logic
addend1	0	Logic
carry_in	1	Logic
sum	1	Logic
carry_out	0	Logic
P	0	Logic
G	0	Logic
Cprop	0	Logic

3) The *Simulation* pane (currently labelled, probably, “Untitled1”) is the main debugging tool for your circuits, as it allows you to observe the behaviour of any signal within the circuit as a *waveform*. At the moment, it is probably unreadable and “squashed”. You will invariably want to float this window and make it as wide as possible (allowing you to observe a longer period of time in the simulation).

Float the window and expand it horizontally. What do you see? Nothing interesting? Well, have a look at the simulation time on top of the graphics display. Is it showing the simulation starting at time 0? Why not? Excellent question: ask Xilinx. Anyway, fiddle with the slider below the graphic display and with the zoom out function (the looking glass icon with a “-” symbol) and you should be able to move the display to the “interesting” area, say between 80ns and 160ns.

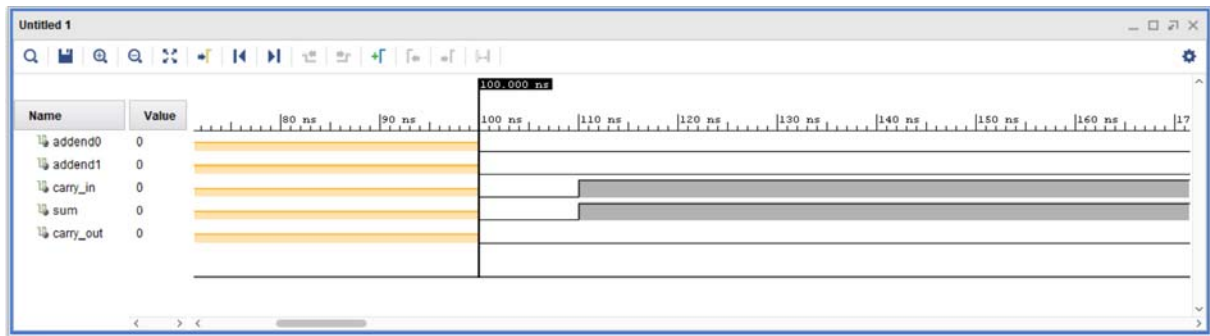


What do you see now? Well, signals are undefined (orange) for the first 100ns. Why? Remember that 100ns wait before your test patterns? **This is absolutely fine and indeed you should *always* see this kind of behaviour at the start of your simulations!** Do NOT try and “get rid” of it as you *will* lose marks if you do (the reason for this will be made clear in lectures).

After 100ns, your test patterns start. At the moment, you only have two (unless you’ve already completed the testbench, in which case you will see some more transitions). You can check that your circuit works by making sure that the outputs are correct given the inputs.

Next, we will go through some steps to improve the visual quality of the simulation. Note that documentation is a fundamental part of professional design, and its quality will constitute a significant part of your marks. First, a step that hopefully will only be required once: getting rid of the “green on black” look and replacing it with a more professional black on white.

Click on the “Settings” icon on the top right of the simulation window, then select the “Colors” tab. Turn all black colours to white and vice-versa (you can use the pull-down for simplicity). Then, set the following values to black: Cursor, Default Waveform, Floating Ruler. Up to you if you want to change any of the other colours: follow your own judgment (keep in mind that some of these you might never see in this module). After you close the setting, your window should look similar to this:



Next, we will **add internal signals to the simulation** (remember that this is fundamental for debugging!)

In the “Scope” pane, select the UUT. The “Objects” pane should now display all signals (I/O ports, internal signals) of the UUT. One by one, select P, G, and Cprop and drag them onto the Simulation window (or right-click on them and select “Add to Wave Window”). They should now appear on the right side of the window, but you will notice that there is no corresponding waveform.

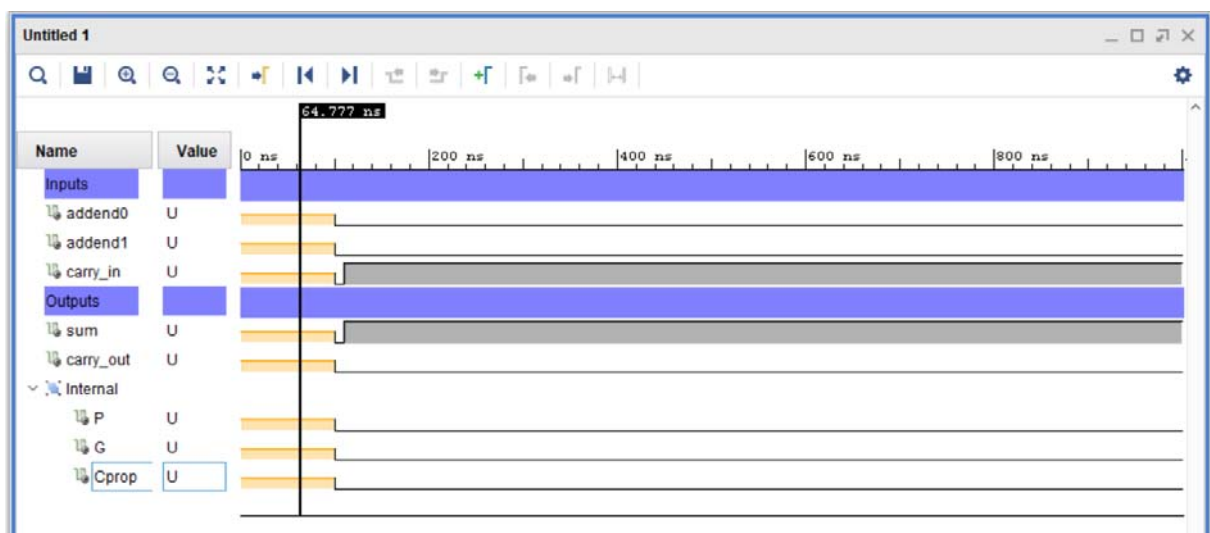
In the main window, click on the “Restart” icon, then set the simulation time to 1us (instead of the default 10us, since this simulation does not need that long) and click on the “Run for 1us” icon. Zoom again. The new signals should now have a waveform.



When a lot of signals are present (as will be the case in later labs), it can be very useful to **divide and/or group** them in the simulations.

Right click in the Name area of the simulation window and select “New Divider”. Click on the label and type “Inputs”. Drag the divider above the first signal (“addend0”). Repeat the operation for a second divider called “Outputs” and drag it above the first output (“sum”). Next, select the names of the three internal signals (shift-click), right-click and select “New Group”. Label it “Internal” and expand it.

The use of dividers and/or groups is down to choice, but in general it will become very useful later on for complex simulations. At the end of the above, your simulation window should look as follows.



None of this work (dividers, groups, choice of signals, etc.) is permanent. In other words, if you now close Vivado and re-launch it at a later time/date, all this will be lost, **unless you save the simulation setup.** In the simulation window, click on the “Save Waveform Configuration” icon. Use the default file name (this will automatically associate the configuration to this testbench) and save the .wcfg file. Select “Yes” in the popup to add it to your project. If you now close the simulation and/or the Vivado toolchain, the groups and dividers will be retrieved automatically (note that if you used a different filename for the configuration file – or if you use multiple configurations – you can load the correct .wcfg file using “Open Waveform Configuration” in the *File* menu of the main window).

Now all that is left is to get everything ready for submission.

Go back to the “Project Manager” in the main window. Re-open (assuming you have closed it) the testbench file. **Add the missing test patterns** (if you have not already done it). You might want to **add the comments here and in the adder entity as well** (see report requirements and Appendix C), or you can do it later.

In the main window, select the Simulation window again. Unless you have made major modifications to your circuit and/or testbench, you don’t need to close and re-open the simulation, but you can simply re-launch it. Do so by clicking on the “Re-launch” icon on the main window toolbar. After zooming, you should be able to see the entire simulation with all test patterns (assuming you did not make any mistakes!)



For the submission, you want to zoom the simulation to highlight the “interesting” area (or areas, in some of the later labs). In general, you should follow a couple of basic rules (in addition to the layout examples of Appendix B):

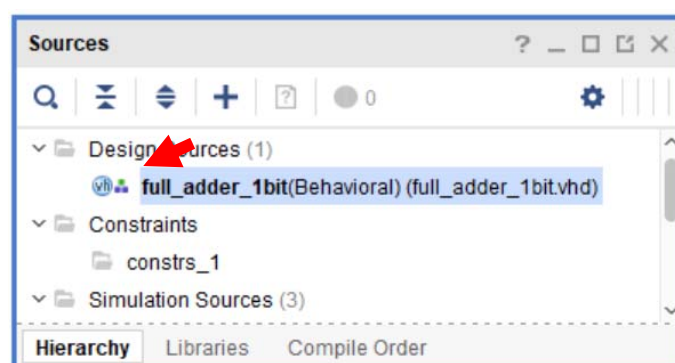
- 1) Always include the start of the simulation (i.e. the exit from the 100ns wait period)
- 2) Always zoom to show clearly *all* the significant events of the simulation, leaving a bit of margin before the first and after the last event (i.e. show what happens to the circuit after the last valid test pattern in the sequence has been fully elaborated)
- 3) Always make sure that all values are clearly readable (here, you only have binary values but later on you will have decimal/hex values)

In the case of this lab, all of this should be very straightforward and one screenshot should be more than sufficient (for point 2, remember that “111” is one of the test patterns!)

Step 4. Synthesis

Once you are satisfied that your design corresponds to the specifications, you can proceed to the next step: synthesis. In this step, your VHDL is transformed into a *netlist*, ready to be implemented.

In the “Project Manager”, make sure that the entity you want to synthesize (“full_adder_1bit”) is the top level entity in your design. In this case, there is only one entity, so that will always be the case, but you should double-check whenever you have multiple entities (which will happen soon).



In the “Flow Navigator”, click on “Run Synthesis”, then click “OK”. If all goes well (and there is no reason why that should not be the case), after a few seconds you will see a pop-up window with a few options. Select “View Reports”, then click OK.

At this stage, the tools have *inferred* a hardware implementation from your design. It is at this stage that any VHDL design errors (as opposed to syntax errors) can be detected. The synthesis reports, automatically produced at this stage, are extremely useful debug tools and should always be looked at to verify that your VHDL has been interpreted correctly.

The bottom pane should display two reports. Double-clicking the second one (“Vivado Synthesis Report”) will open a text document. Float the window for readability if you want, and scroll down to the section labelled “RTL Component Statistics”.

This section (and the following one, where the design is broken down into components, which is irrelevant for this task but can be useful later on) contains a list of the main hardware resources (not all, since it will not list basic logic gates except XOR) generated for the implementation of your circuit. Obviously, in this particular case it is not very interesting (only the XOR gate has been picked up as non-standard) but in more complex designs it will highlight components such as adders, registers, counters, even FSMs.

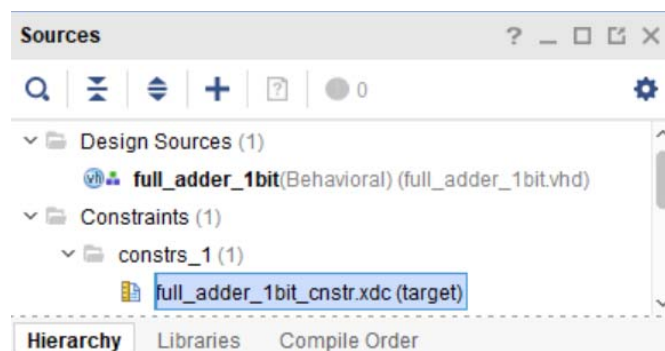
It is fundamental to check that the components found match what you wanted to design in the first place! For example, a very common mistake in VHDL are processes that are not correctly defined. We will look at processes in the next lectures, so don’t worry about it now, but keep in mind that in many cases this will cause synthesis to produce *latches* instead of registers. No latches should ever be produced in any of the labs for this module. This is a very useful debugging tool (particularly since the presence of latches will lower your mark in the labs)! **This section of the Synthesis Report will have to be copy-pasted into the reports for all of the labs in this module** (and yes, this means that you should read it as well”)

Step 5. Implementation

You are now ready for the next step, the implementation of your design in the Xilinx device. However, one additional step is necessary: you have to assign the inputs and outputs of your design to specific pins on the Xilinx device. In this case, you need to assign the three inputs to sliders and the two outputs to LEDs.

In the Flow Navigator, expand “Open Synthesized Design” under “Synthesis”. Click on “Constraints Wizard”, then on “Define Target”, then on “Create File”. Give a name to the constraints file (e.g. “full_adder_1bit_cnstr”) then click “OK”. Select the new file as target (tick the box on the left of the filename) then click “OK”. You might now need to close the synthesis interface (“Close design”) as the constraints have changed

In the Project Manager, you should now have a new source in the constraints tab.



Note that this is a text file and if you double-click it, a text editor will open. While there is a way to do this through a visual interface, by far the easiest way to add pin constraints in a simple design is to manually edit the file. For each I/O port of your design, you must add two lines in the format:

```
set_property PACKAGE_PIN XXX [get_ports {port_name}]
set_property IOSTANDARD LVCMOS18 [get_ports {port_name}]
```

Where XXX is the pin number (see below) and *port_name* is the I/O port signal you want to connect to the pin (for future reference, in the case of vectors the syntax is *vector[index]*, e.g. *bus[3]*).

The list of pins that will be used in this module is the following (you will need to refer to this table for later labs, so keep it in mind):

Pushbuttons		Slide Switches		LEDs		100Mhz clock	
<i>Label</i>	<i>Pin Site</i>	<i>Label</i>	<i>Pin Site</i>	<i>Label</i>	<i>Pin Site</i>	<i>Label</i>	<i>Pin Site</i>
BTNU	T18	SW0	F22	LD0	T22	GCLK	Y9
BTNC	P16	SW1	G22	LD1	T21		
BTNR	R18	SW2	H22	LD2	U22		
BTNL	N15	SW3	F21	LD3	U21		
BTND	R16	SW4	H19	LD4	V22		
		SW5	H18	LD5	W22		
		SW6	H17	LD6	U19		
		SW7	M15	LD7	U14		

For example, to connect the “carry_in” input port to the rightmost slider (SW0), you will need to write:

```
set_property PACKAGE_PIN F22 [get_ports {carry_in}]
set_property IOSTANDARD LVCMOS18 [get_ports {carry_in}]
```

Now connect the other two inputs to two additional sliders and the two outputs to two LEDs. Save the constraints file. You are now ready to implement the circuit, which will transform your VHDL into a circuit that can be realised in the FPGA board.

In order to do this, click on “Run Implementation” in the Flow Navigator. You might get a message about synthesis being out of date, in which case just click “Yes”, then “OK”. After a few seconds (well, ok, maybe more than a few...), a pop-up will ask you what to do next. For the moment, select “Open Implemented Design”. There is a lot of information here, but nothing that directly concerns you (in this module!)

Step 6. Bitstream generation and downloading

The final step is to download the circuit into the FPGA and verify that it works. **Power up the board** (normally, the plug behind the monitors has to be switched on), then in the Flow Navigator, click on “Generate Bitstream” (under “Program and Debug”). Again wait a bit, select “Open Hardware Manager”, then click “OK”.

A message will notify you that “No hardware target is open”. Click on “Open Target”, either in the Flow Generator (under “Open Hardware Manager”) or using the link at the top of the window. Select “Auto Connect”.

Click on “Program Device”. The bitstream file you just created (“full_adder_1bit.bit”) should be automatically selected. Click on “Program”. If all goes well, you should not get any error messages.

Now turn to the board, and verify that your circuit is working by playing around with the switches you have selected and looking at the LEDs.

Remember to power off the Xilinx board!!

Report:

The report for this task should include, in this order:

- The commented VHDL code for the adder (see Appendices A and C)
- The commented VHDL code for the testbench (comments should be applied to the stimulus process only)
- Screenshot(s) of the simulation, zoomed in around the “interesting” area, including all internal signals (see Appendix B).
- The “RTL Component Statistics” section of the Synthesis Report – include the entire section and only this. Keep in mind that this will not be very “interesting” for this lab, with only a couple of lines of report.
- The .XDC file (pin assignments)

Task B: Implementation of components

Create a new project following the instruction in the first laboratory script (remember to use a meaningful name, e.g. “DDVHD_Lab1B”).

Import your adder design from Task A into the new project. To do so, click on “Add Sources”, then select “Add or create design sources”. In the next window, click on “Add Files” and navigate to the VHDL file you created in task A (assuming you have used the default names suggested in the script, the path should be “M:/VivadoProjects/DDVHD_Lab1/DDVHD_Lab1.srcs/sources_1/new/full_adder_1bit.vhd”). Double-click on the file to add it the list, then **tick the box next to “Copy sources into project”**. Click “Finish”. The adder should now be in your Sources.

You will now create a four-bit adder by *instantiating* four one-bit adder components and connecting them together. Start by creating a new source with a meaningful name (e.g. “full_adder_4bit”). As I/O ports, you will need two 4-bit busses for the two addends, one 4-bit bus for the sum, and two 1-bit signals for the carry in and out.

Note that **signal names are local to an entity**. This means that you *can* use the same names for the signals in the 4-bit adder as in the 1-bit adder, but this will not actually mean that they are in any way related. Also, signal names are *not* case sensitive. To experiment, I suggest you use a mix of same and different names. For example:

Module Definition

Entity name:

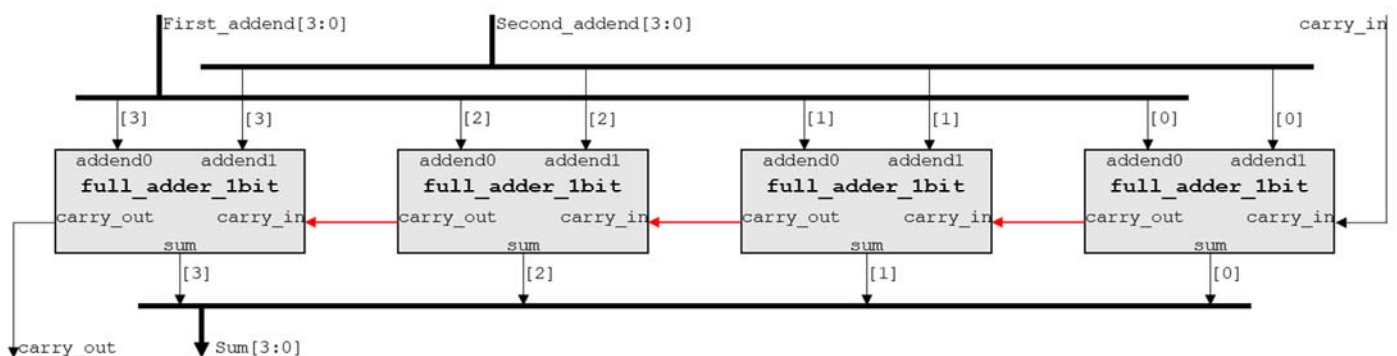
Architecture name:

I/O Port Definitions

Port Name	Direction	Bus	MSB	LSB
First_Addend	in	<input checked="" type="checkbox"/>	3	0
Second_Addend	in	<input checked="" type="checkbox"/>	3	0
carry_in	in	<input type="checkbox"/>	0	0
Sum	out	<input checked="" type="checkbox"/>	3	0
carry_out	out	<input type="checkbox"/>	0	0

Note that, by default, the I/O busses will be created as **STD_LOGIC_VECTOR**. In this lab, that will be acceptable, but in other cases you will have to change them to **SIGNED** or **UNSIGNED** by hand.

Next, you want to define the following circuit:



In order to implement the circuit, first you need to **create all internal signals**. Looking at the circuit, the I/O signals (black lines) are already available (you defined them as I/O ports). The three carry lines (red lines) that connect the adders together, however, are not I/O ports and must be defined.

Define the three lines, either as three STD_LOGIC signals or as a 3-bit STD_LOGIC_VECTOR. Always use names that match the function.

```
architecture Behavioral of full_adder_4bit is

-- Internal signals are defined here

begin
```

Next, *instantiate* four one-bit adders using direct instantiation (the method introduced lecture 2) in the code area. Remember that the generic instantiation template for direct instantiation is:

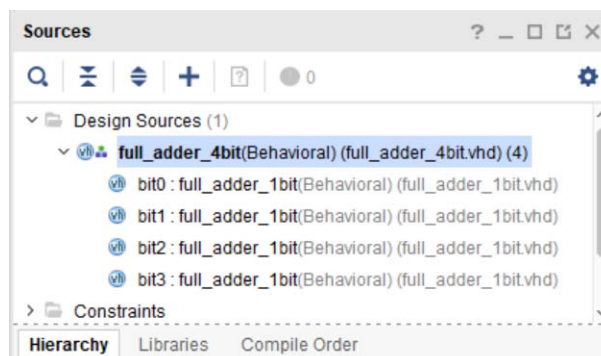
```
<instance_name> : entity work.<component_name>
port map (
    <port_name> => <signal_name>,
    <other ports>..
);
```

In this particular case, the template for the one-bit adder would then be:

```
<instance_name> : entity work.full_adder_1bit
port map (
    addend0    => <signal_name>,
    addend1    => <signal_name>,
    carry_in   => <signal_name>,
    sum        => <signal_name>,
    carry_out  => <signal_name>
);
```

Where <instance_name> is a unique identifier for the component (use meaningful names!) and <signal_name> is the name of the signal that will be connected to the I/O port of the component (remember that the notation for individual bits in a vector is vector(n), where n is the index of the bit – e.g. First_Addend(1)).

Once you have instantiated to components, you should see that the “Sources” pane has updated and should look something like this:



Get rid of the “standard” comments and add your own (you can do this now or later, but don’t forget to do it before you submit). Note that, when describing the component instantiations, it is perfectly acceptable to re-use some or all the comments used in the components themselves (saves a lot of time!)

The next step is to define a testbench to verify the operation of the adder. You can follow the same procedure outlined in the Lab 1 script. However, things here are a bit more “interesting”...

Obviously, not all input combinations can be verified (you would need 2^9 combinations!) **Select a few patterns that you believe will thoroughly test the implementation:** consider that each single-bit adder component has already been tested and therefore can be trusted to work correctly, so you need to test how they work together and in particular any new signal that has been introduced in the design. **The comments to the testbench should always illustrate why you have selected a particular set of test patterns** (and your choice should not be random...)

Simulate the design and verify its operation. In your testbench, you probably want to use binary when assigning test patterns to the input vectors (e.g. `First_Addend <= "0101";`) but this might not be the best notation (radix) to display the values in the simulation. What notation (radix) should you use to represent `First_Addend`, `Second_Addend`, and `Sum`? Binary gives the most detail, hexadecimal is the most compact and is bit-compatible, decimal allows you to verify the results more easily. **The choice of notation depends on the data being represented** and it is not always immediately obvious. In general, you should always **use the most compact and readable notation that provides enough detail to verify the design**. In this case, hexadecimal notation is probably the most appropriate (considering that you are using 4-bit data), but keep in mind that **changing notation to display more detail can be useful for debugging**.

Report:

The report for this task should include, in this order:

- The commented VHDL code for the 4-bit adder.
- The commented VHDL code for the testbench (comments should be applied to the stimulus process). In particular, add comments to justify why you believe that the input patterns you have selected are sufficient to fully verify the operation of the 4-bit adder (you should not use exhaustive testing but rather select a small set of test vectors and explain your choice).
- Screenshot(s) of the simulation, including all internal signals in the four-bit adder (i.e. no need to include the internal signals of the single adders).
- The “RTL Component Statistics” and “RTL Hierarchical Component Statistics” sections of the Synthesis Report.

APPENDIX A: Report Guidelines – VHDL code

Example of good code printout:

- relevant comments (see appendix C)
- proper indenting (always align properly the code and use monospaced fonts)
- try to avoid breaking VHDL structures (e.g. processes, multiplexers) across page boundaries and wrapping lines
- syntax highlighting greatly improves readability

```
library ieee;
use ieee.std_logic_1164.all;

-- Positive-edge parameterizable left-shift register with serial load
-- and asynchronous reset
entity shift_reg is
    generic (
        WIDTH      : positive := 8           -- register size
    );
    port (
        rst        : in  std_logic;          -- active-high reset
        clk        : in  std_logic;          -- active-high clock
        load       : in  std_logic;          -- active-high load signal
        lsb        : in  std_logic;          -- serial data in
        output     : out std_logic_vector(WIDTH-1 downto 0) -- parallel data out
    );
end shift_reg;

architecture BHV of shift_reg is
    signal reg : std_logic_vector(WIDTH-1 downto 0); -- register output
begin

    -- Process for shift register - asynchronous reset
    process(clk, rst)
    begin
        if (rst = '1') then                -- asynchronous reset to zero
            reg <= (others => '0');
        elsif (rising_edge(clk)) then
            if (load = '1') then
                reg <= reg(WIDTH-2 downto 0) & lsb; -- shift in the new LSB by concatenating
                                                    -- the lsb input on the right-hand side
            end if;
        end if;
    end process;

    output <= reg;

end BHV;
```

How to achieve this using Notepad++ and Microsoft Word:

- Open the VHDL file in Notepad++ (“Utilities” folder in the Start Menu)
- Use menu command “Plugins” > “NppExport” > “Copy all format to clipboard”
- Paste in Microsoft Word using the “Keep source formatting” option
- **Note that Notepad++ and Word should also be used to spell-check your comments and to adjust indenting!**

Example of acceptable (barely) code printout (minimal comments, indenting correct but much too deep, proportional font, no syntax highlighting, comment lines wrapping):

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity shift_reg is
  generic (WIDTH : positive := 8);    -- size of register
  port (rst      : in std_logic;
        clk      : in std_logic;
        load     : in std_logic;
        lsb      : in std_logic;
        output   : out std_logic_vector(WIDTH-1 downto 0) );
end shift_reg;

architecture BHV of shift_reg is
  signal reg : std_logic_vector(WIDTH-1 downto 0);
begin

  -- shift-left register with asynchronous reset and load
  sh_reg: process(clk, rst)
    begin
      if (rst = '1') then
        reg <= (others => '0');
      elsif (rising_edge(clk)) then
        if (load = '1') then
          reg <= reg(WIDTH-2 downto 0) & lsb; -- shift in the new LSB by
concatenating the lsb input on the right-hand side
        end if;
      end if;
    end process sh_reg;

  output <= reg;

end BHV;
```

Example of NON-ACCEPTABLE code printout (barely any comments, no indenting, proportional font, no syntax highlighting, wrapping lines, random layout formatting, double-space lines, useless header text):

```
-----  
  
-- Company:  
  
-- Engineer:  
  
--  
  
-- Create Date:  13:25:42 01/09/2014  
  
-- Design Name:  
  
-- Module Name:  tmp - Behavioral  
  
-- Project Name:  
  
-- Target Devices:  
  
-- Tool versions:  
  
-- Description:  
  
--  
  
-- Dependencies:  
  
--  
  
-- Revision:  
  
-- Revision 0.01 - File Created  
  
-- Additional Comments:  
  
--  
  
-----  
  
library IEEE;  
  
use IEEE.STD_LOGIC_1164.ALL;  
  
  
-- Uncomment the following library declaration if using  
-- arithmetic functions with Signed or Unsigned values  
  
--use IEEE.NUMERIC_STD.ALL;
```

```
-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.

--library UNISIM;

--use UNISIM.VComponents.all;


ENTITY SHIFT_REG IS

GENERIC (WIDTH : POSITIVE := 8);

PORT (RST : IN STD_LOGIC; CLK : IN STD_LOGIC; LOAD : IN STD_LOGIC; LSB : IN STD_LOGIC;
OUTPUT : OUT STD_LOGIC_VECTOR(WIDTH-1 DOWNT0 0));

END SHIFT_REG;


ARCHITECTURE BHV OF SHIFT_REG IS

SIGNAL REG : STD_LOGIC_VECTOR(WIDTH-1 DOWNT0 0);

BEGIN


PROCESS(CLK, RST)

BEGIN

IF (RST = '1') THEN

REG <= (OTHERS => '0');

ELSIF (RISING_EDGE(CLK)) THEN

IF (LOAD = '1') THEN

REG <= REG(WIDTH-2 DOWNT0 0) & LSB; -- SHIFT LEFT

END IF;

END IF;

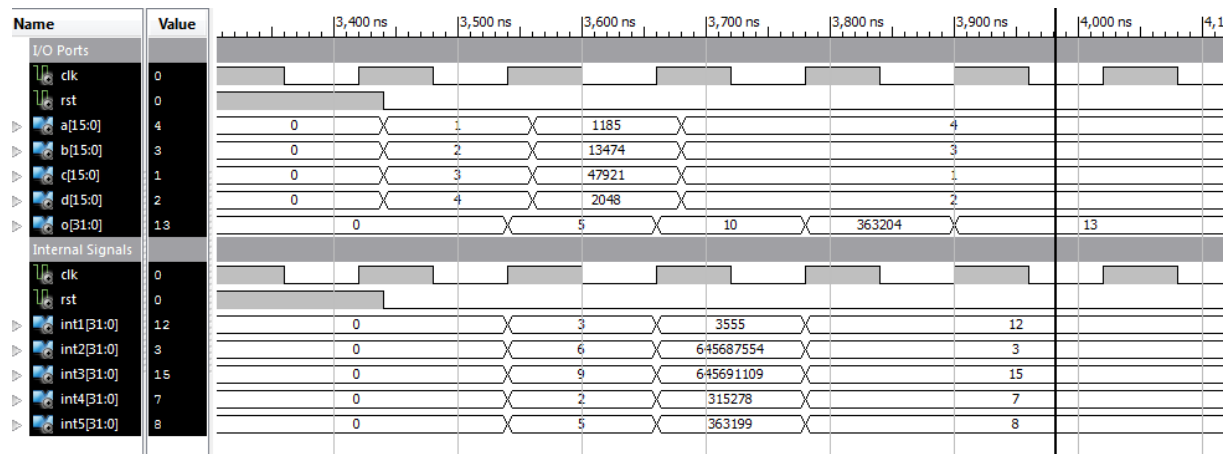
END PROCESS;

OUTPUT <= REG;

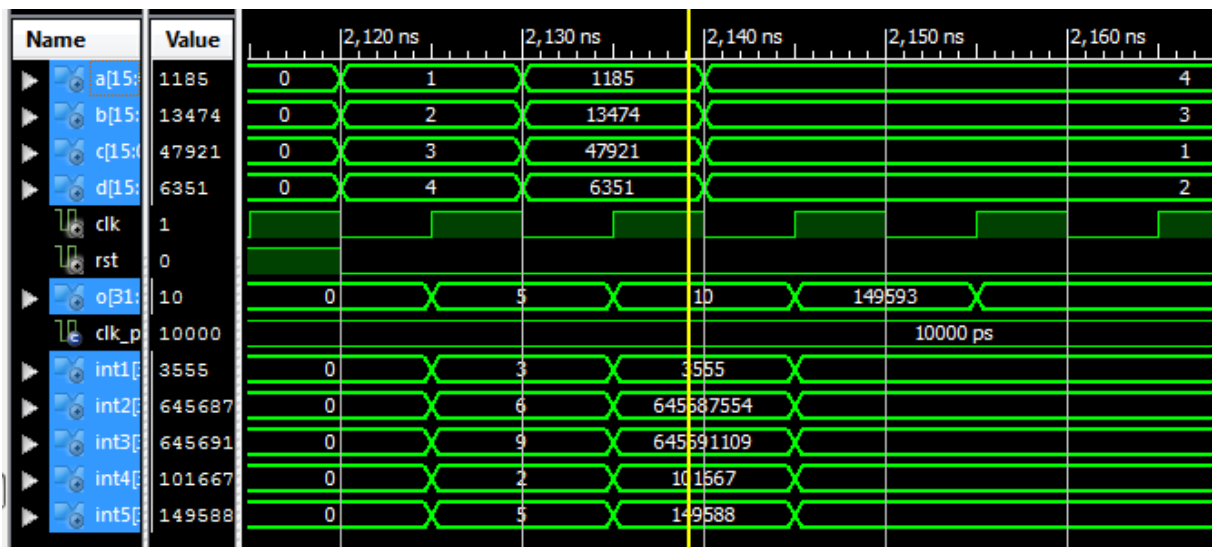
END BHV;
```

APPENDIX B: Report Guidelines – Simulation screenshots

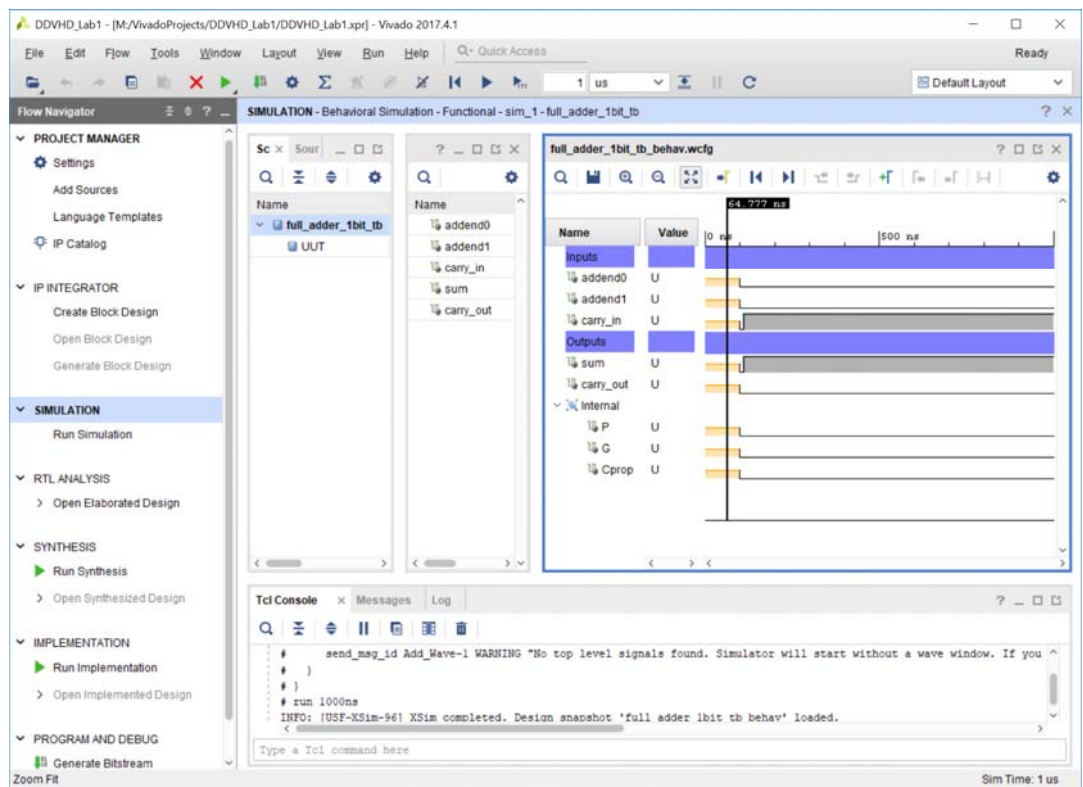
Example of good simulation screenshot:



Example of barely acceptable simulation screenshot (minor penalty for colour scheme):



Example of **NON-ACCEPTABLE** simulation screenshot:



APPENDIX C: Report Guidelines – Commenting

Examples of good code comments:

- Avoid “obvious” comments (e.g.: “input port”, “counter is incremented”, “A gets value ‘0’”, “go back to 0”). Comments should add information that is not immediately obvious from the code.
- Short descriptions of the function of entity, I/O ports, any components, and any internal signals should always be included. Comments should be about the design of the circuit, not about VHDL syntax.
- There is often redundancy in these comments (especially when I/O ports of the entity are directly connected to the I/O ports of the components), but it is usually easier to cut and paste than to selectively remove comments.

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
use IEEE.NUMERIC_STD.ALL;
```

ALWAYS Include a description of the circuit implemented by the entity and of the I/O signals

```
-- Control logic for a garage door opener  
-- The entity contains a FSM component to control the door and a counter to  
-- generate a one-second pause required by the operation of the FSM
```

Usually, no need to comment on clock and reset (unless there is something non-standard)

```
entity control_logic is  
  port (clock : in std_logic;  
        reset : in std_logic;  
        button : in std_logic;           -- remote control button  
        bottom_sensor : in std_logic;    -- door sensor: high when door closed  
        top_sensor : in std_logic;       -- door sensor: high when door open  
        -- door motor control: 00 - stop; 01 - close; 10 - open; 11 - unused  
        Motor : out std_logic_vector (1 downto 0));  
end control_logic;
```

Avoid long comments that wrap on a new line – use multiple comment lines, preferably aligned, or separate lines

```
architecture Behavioral of control_logic is
```

```
signal count_en : STD_LOGIC; -- enable for one-second pause counter  
signal one_second : STD_LOGIC; -- high for 1 clock cycle when one second is reached  
constant SECOND : natural := 100000000; -- number of clock cycles corresponding  
-- to one second in real time (100MHz clock)  
signal Counter : unsigned (26 downto 0); -- vector for one-second counter  
-- (100M = 27 bits)
```

Use meaningful names for your components and signals

```
begin
```

```
-- Counter to one second (see SECOND constant)  
-- Rising edge, synchronous reset to 0, and enable  
count_one_sec : process (clock)
```

Describe what kind of sequential element are implemented for each process – no need to comment every line within the process (unless there is something not obvious)

```
begin  
  if (rising_edge(clock)) then  
    if (reset = '1') then  
      Counter <= (others => '0');  
    else  
      if (count_en = '1') then  
        if (Counter = SECOND) then  
          Counter <= (others => '0');  
        else  
          Counter <= (Counter + 1);  
        end if;  
      end if;  
    end if;  
  end if;  
end process count_one_sec;
```

When printing, try to avoid breaking structures (components, processes) across pages. Insert blank lines if necessary (within reason, obviously)

```

-- Finite State Machine for a garage door opener
door: entity work.garage_door_FSM PORT MAP(
  reset => reset,
  clock => clock,
  button => button,
  bottom_sensor => bottom_sensor,
  top_sensor => top_sensor,
  one_second => one_second,
  -- door motor control: 00 - stop; 01 - close; 10 - open; 11 - unused
  Motor => Motor,
  count => count_en -- enable signal for one-second pause counter
);

-- This signal will be '1' for a single clock cycle when the counter has reached 100M
one_second <= '1' when (Counter = SECOND and count_en = '1') else '0';

end Behavioral;

```

One line per port

Add a description of each component you use and its I/Os (usually a straight copy/paste from the entity description of the component)

You don't need to comment every assignment – do it only if there is useful, non-obvious information to add (in this case, that the signal will be '1' for a single cycle!)

Some counter-examples (i.e. “bad” comments):

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

-- Entity declaration
entity control_logic is
  port (clock : in std_logic; -- I/O ports
        reset : in std_logic;
        button : in std_logic;
        bottom_sensor : in std_logic;
        top_sensor : in std_logic;
        Motor : out std_logic_vector (1 downto 0));
end control_logic;

architecture Behavioral of control_logic is
  -- Internal signals for the control logic
  signal sig1, sig2 : STD_LOGIC;
  signal Counter : unsigned (26 downto 0);

begin
  -- Counter
  proc_inst : process (clock)
  begin
    if (rising_edge(clock)) then
      if (reset = '1') then -- when reset is high
        Counter <= (others => '0'); -- counter goes to 0
      else
        if (sig1 = '1') then --when the counter is enabled
          if (Counter = 100000000) then -- if the counter has reached 100000000
            Counter <= (others => '0'); -- then it goes back to 0
          else
            Counter <= (Counter + 1); -- otherwise it increments by one
          end if; end if; end if; end if;
        end process proc_inst;
      end if; end if; end if; end if;
    end process proc_inst;

  end if; end if; end if; end if;
end process proc_inst;

Inst: entity work.garage_door_FSM PORT MAP(
  reset => reset, clock => clock, button => button, bottom_sensor => bottom_sensor,
  top_sensor => top_sensor, one_second => sig2, Motor => Motor, count => count_en);

-- sig2 is equal to '1' when the counter is equal to 100000000 and is enabled,
otherwise '0'
sig2 <= '1' when (Counter = 100000000 and sig1 = '1') else '0';

end Behavioral;

```

No description of circuit, I/Os, and internal signals. Comments on syntax instead of design.

Meaningless, “generic” internal signal and component names

Obvious comments that add nothing to the VHDL code

Explicit value instead of constant (needs multiple replace if changed)

No component description

All ports on a single line (no comments possible)

Wrapping comment