2025

Low-Level HW Digital Systems II

LAB MANUAL & COURSEWORK

VASILIS F. PAVLIDIS – ASSOC. PROFESSOR, ECE DEPT.
EVANGELOS TZOUVARAS - PHD STUDENT

Contents

1 Introduction

This document describes briefly the prerequisites required to set up the tools for compiling, simulating, and verifying the circuits you will design as part of the coursework for this course. The document is divided into two sections. The following section includes some guidelines regarding the simulators you can use and some concise instructions on how to install and use a simulator offered for free for Intel's FPGAs. Section 3 describes the simulator setup, the coursework you need to complete, and the deliverables.

2 Simulation Tools

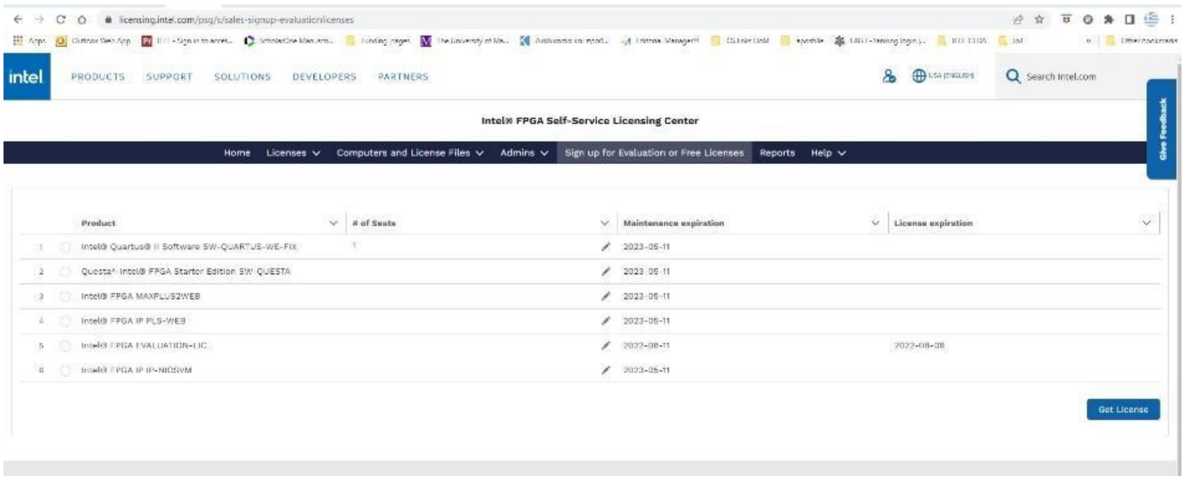
A well-established tool for RTL simulation has been the Modelsim simulator [1] of Mentor-Graphics (now part of Siemens). Unfortunately, the student edition of this great tool is not available any longer. Thus, you need to seek another simulator that effectively has integrated Modelsim in their environment. There are a few available options, which you can use for free for student projects. These options include, for example, the Quartus tool from Intel [2] and Libero® SoC Design Suite from Microchip [3].

We recommend the use of Quartus provided on [2], which supports both a Windows and a Linux version. Unfortunately, there are no simulation tools that we know of that presently support Mac OS. Thus, Mac users need to find a workaround. In the following, we succinctly describe how to install and use the Intel simulator. Note that we cannot offer you any technical support on this topic (I managed to get the tool up and running, hence, I am confident you will have no problems either).

2.1 Questa – Intel FPGA Edition

Follow the steps below to install the Quartus simulator for RTL simulation. Alternatively, you can use any other tools you like (even the one you used in the last semester but make sure that all work fine).

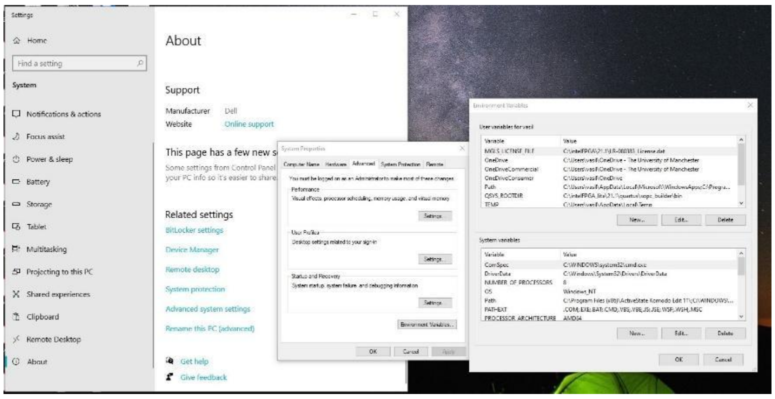
- Go to [2], click on Individual files in the Download Section, and select to download the Questa*-Intel® FPGA Edition. Please check you have sufficient disk space to proceed with the installation. Although the tool requires about 1 GB during the installation process you may need much more (possibly up to 30 GB [4]). Also, check the memory requirements of the simulator (as with any other tool).
- During the installation process, you SHOULD select Intel FPGA **Starter** Edition as only this edition offers a free license for 1 year.
- Once the tool installation is complete, a process that takes quite long (tens of minutes depending on your computer), you need to acquire a license file. Please visit the Intel® FPGA Self Service Licensing Center (SSLC) and sign up using your **institutional email** (please do not use your personal email, e.g., Gmail).
- After registration, you will receive a confirmation email from Intel. Using this email, log on SSLC and click on the "Sign up for Evaluation or Free Licences" tab and select Questa*-Intel® FPGA Starter Edition, as depicted in Fig. 1, where you edit the number of seats to one. Click on "Get license".


Intel® FPGA Self-Service Licensing Center

	Product	# of Seats	Maintenance expiration	License expiration
1	Intel® Quartus® II Software SW-QUARTUS-WE-FIX	1	2023-05-11	
2	Questa*-Intel® FPGA Starter Edition SW-QUESTA		2023-05-11	
3	Intel® FPGA MAXPLUS2WEB		2023-05-11	
4	Intel® FPGA IP PLS-WEB		2023-05-11	
5	Intel® FPGA EVALUATION-LIC		2022-08-11	2022-08-38
6	Intel® FPGA IP IP-NIOSVM		2023-05-11	

Fig. 1. SSCL Screen

- You will need to provide the NIC of your computer (physical or MAC address of your Ethernet or wireless card, which consists of 12 hex digits) that is used to identify your computer (execute `ipconfig -all` on the Windows command prompt to obtain this hex number). Please note that you can support up to 3 devices with the same license file. In this case, you need to install the software on all three machines. In addition, note that you cannot run the software remotely but only locally with this type of free license.
- Typically, it takes some time (nominally up to 12 hrs) to register your account on SSCL, although you have received the activation email. So, allow some time for this to happen before you attempt to check out a license.
- By selecting the proper type of license, you will receive an email where the license file with the extension .dat will be attached. Save this file to a desired location, typically, some folder within the top-level directory `intelFPGA`. This file is used by the license daemon of the tools, located in the path C:\intelFPGA\21.1\questa_fse\win64.
- An environment variable should also be added to the system. For Windows, right-click on Start, select System, Advanced System Settings, Environment Variables, and create a New variable called LM_LICENSE_FILE and provide the path where you stored the license file you received from Intel (*.dat). Click OK. An example of this setup is shown in Fig. 2.
- You can now start using the software. Also, use the Intel® FPGA Software Installation and Licensing manual available on the elearning course webpage. Note that you are interested in the Intel FPGA **Starter** Edition.


Environment Variables

User variables for vasil

Variable	Value
MGLS_LICENSE_FILE	C:\intelFPGA\21.1\LR-080333_License.dat
OneDrive	C:\Users\vasil\OneDrive - The University of Manchester
OneDriveCommercial	C:\Users\vasil\OneDrive - The University of Manchester
OneDriveConsumer	C:\Users\vasil\OneDrive
Path	C:\Users\vasil\AppData\Local\Microsoft\WindowsApps;C:\Progra...
QSYS_ROOTDIR	C:\intelFPGA_lite\21.1\quartus\sopc_builder\bin
TEMP	C:\Users\vasil\AppData\Local\Temp

System variables

Variable	Value
ComSpec	C:\WINDOWS\system32\cmd.exe
DriverData	C:\Windows\System32\Drivers\DriverData
NUMBER_OF_PROCESSORS	8
OS	Windows_NT
Path	C:\Program Files (x86)\ActiveState Komodo Edit 11\;C:\WINDOWS\...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
PROCESSOR_ARCHITECTURE	AMD64

Fig. 2. Setting up the environment library for Questa – Intel FPGA simulator.

3 Coursework

This section describes the tasks that you need to complete as part of the laboratory exercises. An example of a circuit described in SytemVerilog (SV) and tested with typical testbenches and SV Assertions (SVA) is discussed in subsection 3.1. The circuits that you need to design and verify with SV and SVA are presented in subsection 3.2. Each student should individually submit a report (**strictly in pdf format**) with all the deliverables as required in Section 4.

3.1 Setting up the Simulation Environment

The basics of setting up a circuit for functional simulation are first described. Specific steps that you need to follow in the Questa-Intel simulator to examine SVA are also provided. To simulate a design, you need a testbench (similar to the methodology you used in Low-Level HW Digital Systems I in the 7th semester). In addition to this file, another file that contains all the SVA, you have created for the design (DUT) or circuit (CUT) under test, must be included for simulation. This is done through the `bind` command described in the course. This binding process is schematically shown in Fig. 3.

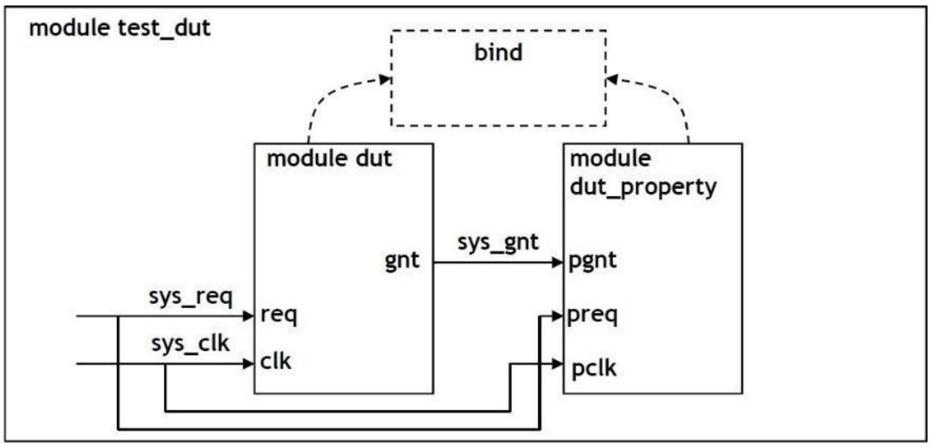


Fig. 3. In the module test_dut, the dut and dut_property modules are bound. The test_dut is the testbench that contains all required information to test the module dut. SVA are contained in the dut_property module.

In the following, the code for each of these three modules is provided such that you understand the methodology of binding modules, using SVAs, and producing an appropriate testbench. Note that you need to read and understand how these files work altogether, using also the examples and material you have been provided during lectures.

3.1.1 Design under test

A simple behavioral Verilog model that acts as the DUT driving a simple req/gnt protocol follows. The code in the text box below includes different assertions that may be desired to check the functionality of the DUT. Read this code carefully combined with the taught material in the lectures. Note the presence of macros (defined with the `ifdef elsif endif` structure), which can help you choose the property that is checked each time. You may opt to completely ignore such macros.

Also, note that any command starting with (`) is a compilation directive rather than part of the RTL code and, hence, of the DUT.

```
module dut_property(pclk,preq,pgnt);
input pclk,preq,pgnt;

`ifdef no_implication
property pr1;
  @(posedge pclk) preq ##2 pgnt;
endproperty
preqGnt: assert property (pr1) $display($stime,,,"\t\t %m PASS");
          else $display($stime,,,"\t\t %m FAIL");

`elsif implication
property pr1;
  @(posedge pclk) preq |-> ##2 pgnt;
endproperty
preqGnt: assert property (pr1) $display($stime,,,"\t\t %m PASS");
          else $display($stime,,,"\t\t %m FAIL");

`elsif implication_novac
property pr1;
  @(posedge pclk) preq |-> ##2 pgnt;
endproperty
preqGnt: assert property (pr1) else $display($stime,,,"\t\t %m FAIL");

property pr2;
  @(posedge pclk) preq ##2 pgnt;
endproperty
cpreqGnt: cover property (pr2) $display($stime,,,"\t\t %m PASS");
`endif

endmodule
```

```
module dut(clk, req, gnt);
input logic clk,req;
output logic gnt;

initial
begin
  gnt=1'b0;
end
initial
begin
@(posedge req);
  @(negedge clk); gnt=1'b0;
  @(negedge clk); gnt=1'b1;
@(posedge req);
  @(negedge clk); gnt=1'b0;
 @(negedge clk); gnt=1'b0;
end
endmodule
```

The binding process and the stimuli used in this example are presented in the following textbox. Invest as much time as required to understand what this code does, how inputs are stimulated and how real-time report messages are produced during the simulation. You can remove the $finish command, as this command pops up a window that asks to quit the tool, not just the simulation.

```
module test_dut;
bit sys_clk,sys_req;
wire sys_gnt;

/* Instantiate 'dut' */
dut dut1 (.clk(sys_clk),.req(sys_req),.gnt(sys_gnt));

bind dut dut_property dutbound (clk,req,gnt);
//
// You need to know the names of the ports in the design and the property module
// to be able to bind them. So, here they are:
// Design module (dut.v)
// ---------------------
// module dut(clk, req, gnt);
//            input logic clk,req; //            output logic gnt;
// Property module (dut_property.sv)
// --------------------------------
//module dut_property(pclk,preq,pgnt); //input pclk,preq,pgnt;

always @(posedge sys_clk)
  $display($stime,,,"clk=%b req=%b gnt=%b",sys_clk,sys_req,sys_gnt);
always #10 sys_clk = !sys_clk;

initial
begin
        sys_req = 1'b0;
  @(posedge sys_clk) sys_req = 1'b1; //30
  @(posedge sys_clk) sys_req = 1'b0; //50
  @(posedge sys_clk) sys_req = 1'b0; //70
  @(posedge sys_clk) sys_req = 1'b1; //90
  @(posedge sys_clk) sys_req = 1'b0; //110
  @(posedge sys_clk) sys_req = 1'b0; //130

@(posedge sys_clk);
@(posedge sys_clk); $finish(2);
end
```

3.1.2 Simulation tool

You can always compile and simulate these files using the Questa Intel HDL simulation tool as you did in the coursework of Low-Level HW Digital Systems I and, therefore, no detailed description will be provided. In general, you need to

- Create a new project (give a name of your preference) and fill in the fields on the displayed window.
- Add existing files or create a new file through the tool editor. When you add existing files, think about whether you want to copy the files to the working directory (easier for command line reference to the files) or keep them in the reference directory (Fig. 6 and Fig. 7 in Appendix).
- Once you have successfully compiled all related files (Fig. 8 and Fig. 9 in Appendix), you are ready to start the simulation. Click on Simulate -> Start Simulation. On the window that pops up, expand the "work" library and pick the files you want to simulate as shown in the related figure below (Fig. 10 in Appendix).
- Click on the optimization options and select full (or custom) visibility (Fig. 11 in Appendix) that allows retrieving all signals in the design hierarchy to view these signals on the waveform window. Select the component dut and add its signals to the waveform (Fig. 12 in Appendix).
- Click on the Run icon and you can simulate your circuit for 100 ns or, whichever, other option you prefer (Fig. 13 in Appendix).
- To activate the properties since these are encapsulated within macros, you need to explicitly define these arguments (one or more) during the compilation of the dut_property.sv file. The compile command (vlog) is of the form:

```
vlog -sv dut_property.sv +define+no_implication
```

- The argument "-sv" means compile with SystemVerilog options and "+define" defines a macro, in this case, the "no_implication" property. More macros can be simultaneously defined by adding more macros to the command. The tool returns a warning in this case that multiple macros are defined but you can safely ignore this.
- Restart the simulation (not the tool!) and you will see that some assertions now appear. Information on assertions can be found by clicking on the Assertions tab right at the bottom of the waveform subwindow (Fig. 14 in Appendix).
- **Tips**: type help <command> in the command line of the simulator to get help with any command. The simulator also provides auto-complete on any command or path. Use the Up arrow in the Transcript window to see the history of executed commands as you proceed with different tasks through GUI or typed commands.
- Click Help -> Questa sim – Intel FPGA Edition – Bookcase to find a rich set of manuals. Use these with caution and at your own discretion as they contain hundreds of pages!

3.2 Coursework Exercises

You are required to design with SV and verify with a testbench and SVA the circuit presented in the following subsections. You must:

- Produce the SV code
- Test the code correctness through a testbench
- Verify correct functionality of the circuit with SVA

3.2.1 Floating Point Representation

All floating-point numbers consist of three parts:

1. A *sign* - Indicates whether the number is positive (sign = 0) or negative (sign = 1)
2. An *exponent* - This is the power of 2 (binary) that the mantissa is multiplied with to get the actual value.
3. A *mantissa/fraction* - The "fractional" portion of the number. There is an implied leading 1 (hidden bit).

The sizes of the exponent and mantissa may differ based on the precision used. A few different precisions can be seen in Figure 1. For the module under design, you will be using only the **single precision**, which has an 8-bit long Exponent and a 23-bit long Mantissa.

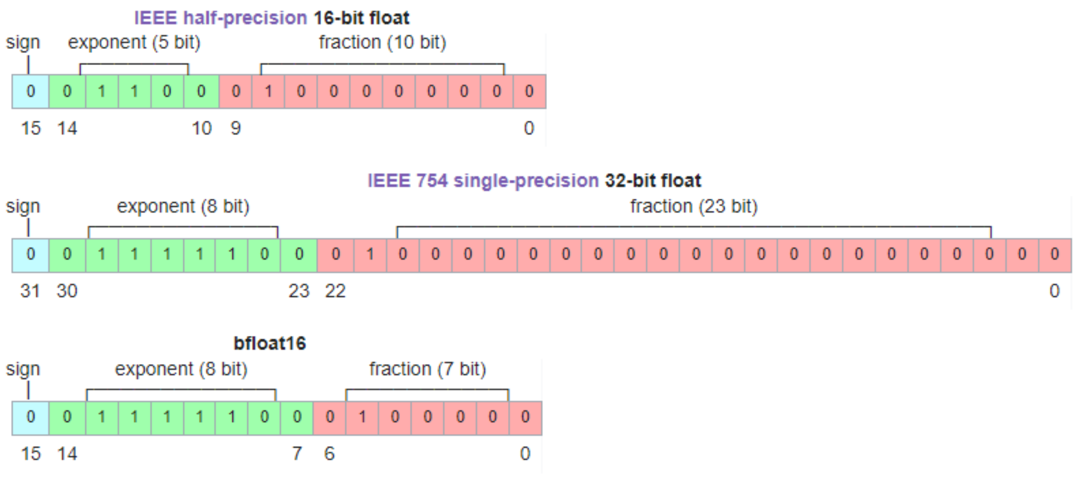


Figure 1: Three different types of floating point precisions. From top to bottom the 16-bit half, the 32-bit single, and the 16-bit bfloat16.

The actual value of a floating point number can be acquired through the representation above as can be seen in Figure 2 specifically for single precision. The exponent is always subtracted by a bias, in our case this number is equal to 127.

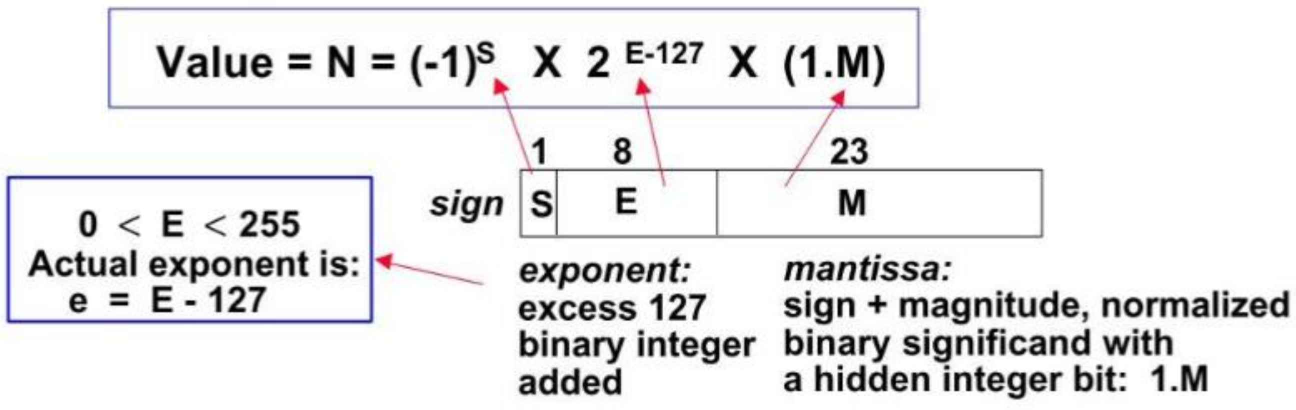


Figure 2: The actual single floating point value can be found using the sign, exponent, and mantissa bits of the representation.

Based on the values in each of the three parts, the numbers can be categorized as shown in Figure 3.

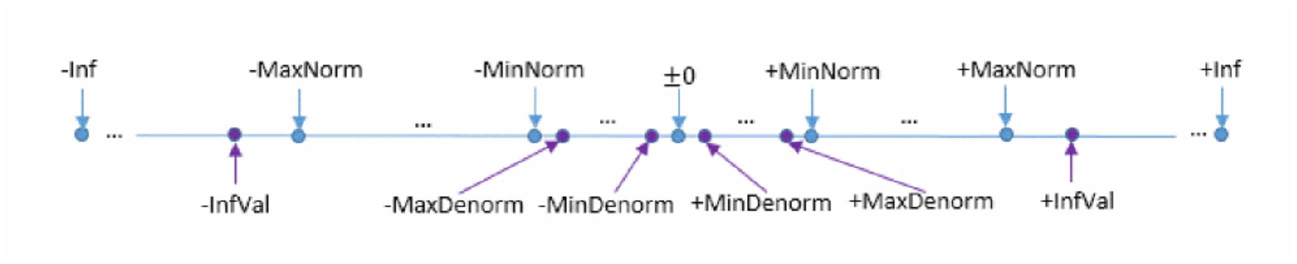


Figure 3: The location of all the special floating point numbers on a 1D line. The order is the following: -NaNs, -Inf, -Normals, -Denormals, -Zero, +Zero, +Denormals, +Normals, +Inf, +NaNs

Table 1 shows the pairing between each category and the representation in the respective floating point number regardless of the precision.

As can be seen in Table 1, a 0 Exponent and 0 Mantissa represent the value positive/negative zero depending on the sign value. Likewise, an Exponent with all bits equal to 1 and Mantissa equal to 0 represents the positive/negative infinities based on the sign.

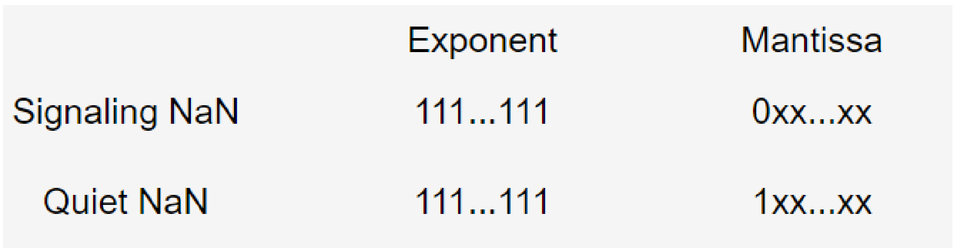
All the numbers that have the bits of the Exponent equal to 1 but the Mantissa has at least one bit equal to 1 are considered to be NaNs (as you might have guessed based on the explanation given, there are multiple representations of a NaN).

A denormal is a number that has all the bits of the Exponent equal to 0 and a Mantissa with at least one bit equal to 1. **Most of the time, as well as in our case, denormals are considered to be equal to zeroes**. All the other numbers are considered to be normals and represent a value based on Figure 2.

Sign (s)	Exponent (exp)	Significand (sig)	Value	Short Name
s	0	0	$(-1)^s 0$	±Zero
s	111...111	0	$(-1)^s$Infinity	±Inf
s	111...111	$0 < sig$	Not a Number	NaN
s	0	$0 < sig$	$(-1)^s\, 2^{1-bias}\, 0.sig$	Denormal
s	$0 < exp < 111...111$	sig	$(-1)^s\, 2^{exp-bias}\, 1.sig$	Normal

Table 1: Special values for floating point numbers. ‘S’ denotes the sign, ‘Sig’ denotes the Mantissa, and ‘exp’ denotes the Exponent. Bias is equal to 127 for single precision.

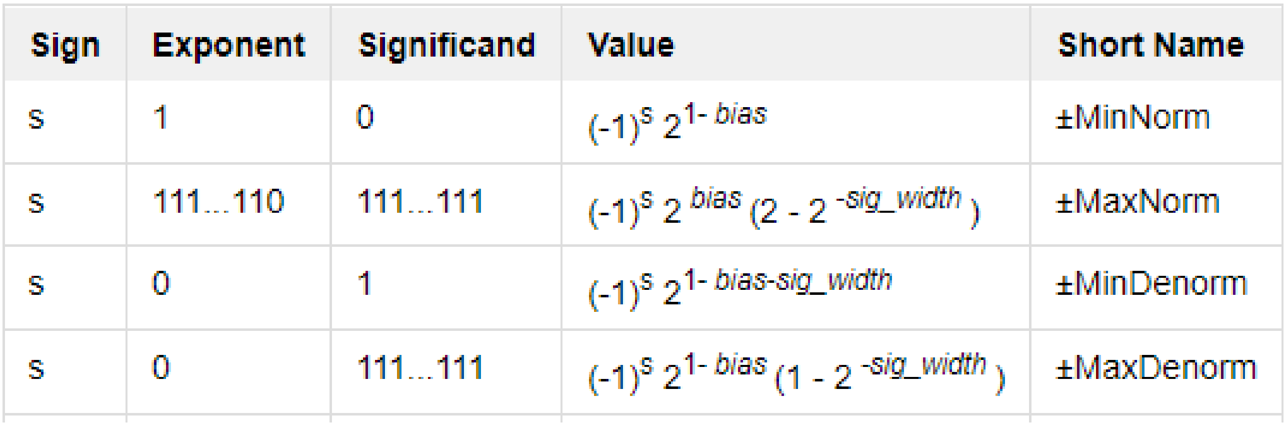
NaNs are the numbers which have all the bits of the Exponent equal to 1 and a Mantissa greater of equal to 1, as it is shown in the Table 1. According to the IEEE-754 standard the NaNs are separated into two sub-categories, the signaling and the quiet NaNs, based on the MSB of the Mantissa part. Thus, NaN numbers with the MSB of Mantissa equal to zero are categorized as Signaling NaNs and NaN numbers with the MSB of Mantissa equal to one are defined as Quiet NaNs (Fig. 4).



	Exponent	Mantissa
Signaling NaN	111...111	0xx...xx
Quiet NaN	111...111	1xx...xx

Figure 4: Sub-categories on NaN numbers.

Table 2 also shows some special floating point numbers, which can be seen in Figure 3 as well. The MinNorm and MaxNorm values represent the lower and upper bounds of the positive and negative normal numbers. Likewise, the MinDenorm and MaxDenorm values represent the lower and upper bounds for the denormals.



Sign	Exponent	Significand	Value	Short Name
s	1	0	$(-1)^s\, 2^{1-\mathit{bias}}$	±MinNorm
s	111...110	111...111	$(-1)^s\, 2^{\mathit{bias}}\, (2 - 2^{-\mathit{sig\_width}})$	±MaxNorm
s	0	1	$(-1)^s\, 2^{1-\mathit{bias}-\mathit{sig\_width}}$	±MinDenorm
s	0	111...111	$(-1)^s\, 2^{1-\mathit{bias}}\, (1 - 2^{-\mathit{sig\_width}})$	±MaxDenorm

Table 2: The upper and lower bounds of the normal and denormal numbers. ‘S’ denotes the sign, ‘sig_width’ denotes the Mantissa width, and bias is equal to 127 for single precision.

3.2.2 Describing the Floating Point Multiplier

The first task of the coursework is to write a floating point multiplier, with a pipeline stage, in system verilog. The multiplier will consist of four modules, the **main module**, the **normalization module**, the **rounding module**, and the **exception module**. A wrapper module (fp_mult_top) is given to be used as a top-level for the other four modules.

3.2.3 Main module

The block diagram representation of the main module can be seen in Figure 5. The description of the wrapper can be seen in Figure 6.

The main module of the multiplier will have three inputs and two outputs. The size of the two inputs (**a,b**) and the resulting output (**z**) of the multiplication are 32-bit long (single precision). The size of the **rnd** input is 3 bits long. The **status** output is 8-bit long where each bit has a different meaning based on the operations that happen inside the multiplier. The meaning of each bit for the status signal can be found in Table 3.

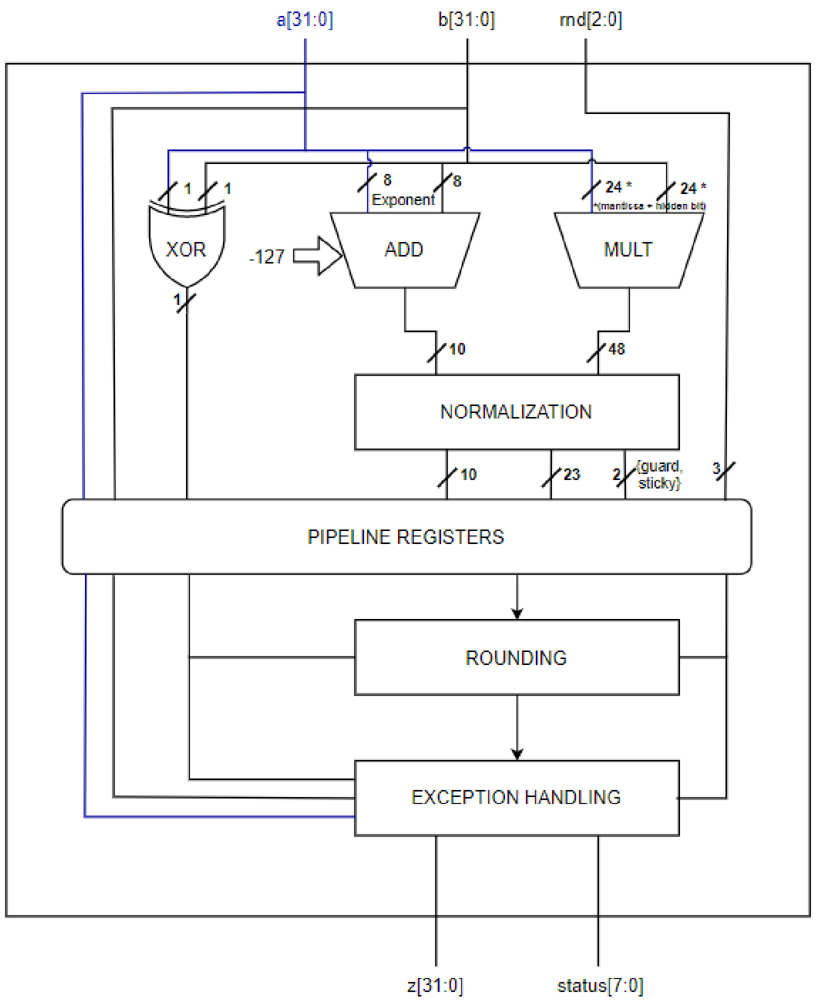


Figure 5: The block diagram of the main module of the multiplier.

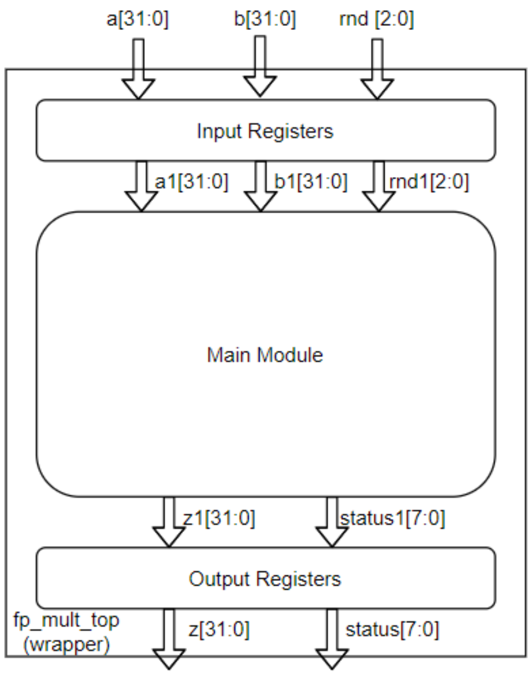
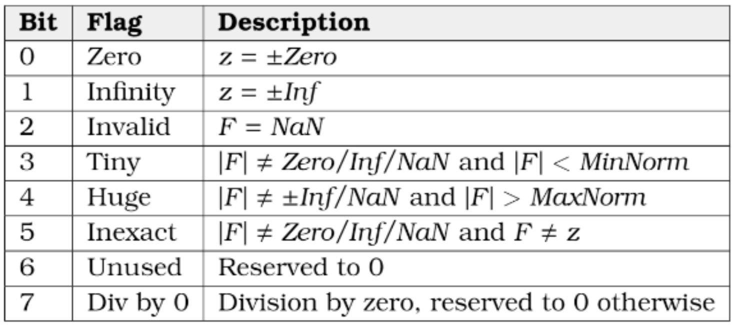


Figure 6: The relationship between the main module and the wrapper.

The main module (Figure 5) will consist of the following 8 stages in order:

1) Floating point number sign calculation
2) Exponent addition
3) Exponent subtraction of bias
4) Mantissa multiplication (**including leading ones**)
5) Truncation and normalization
6) Pipeline stage
7) Rounding and
8) Exception handling



Bit	Flag	Description
0	Zero	$z = \pm Zero$
1	Infinity	$z = \pm Inf$
2	Invalid	$F = NaN$
3	Tiny	$\|F\| \neq Zero/Inf/NaN$ and $\|F\| < MinNorm$
4	Huge	$\|F\| \neq \pm Inf/NaN$ and $\|F\| > MaxNorm$
5	Inexact	$\|F\| \neq Zero/Inf/NaN$ and $F \neq z$
6	Unused	Reserved to 0
7	Div by 0	Division by zero, reserved to 0 otherwise

Table 3: Meaning of each bit of the output status flag.

Warning: **z** is the component's output whereas **F** is the calculated result before it gets rounded to z.

After the normalization module, a pipeline stage must be implemented. Signals used in the next stages, after the normalization, must **pass through the pipeline registers**. Pipelining in digital design, store intermediate signals and allow a new operation to start while the previous one continues executing, improving the performance of the circuit. The pipeline registers will be triggered on the **positive edge of the clock** (@ posedge), and the **reset** signal will be **active-low** and will initialize the values to zero (like the *fp_mult_top* wrapper module).

3.2.4 Normalization module

The normalization module is responsible for normalizing the floating point number's mantissa and finding the guard and sticky bits. Normalization is the act of shifting the binary point in order to leave one non-zero binary digit before the binary point while changing the exponent accordingly.

The inputs of the module are the 48-bit long **multiplication result 'P'** and the 10-bit long **sum of exponents subtracted by the bias**. The outputs of the module are the **sticky** and **guard bits**, the 23-bit long **normalized mantissa**, and the 10-bit **normalized exponent**.

The block diagram of the normalization module can be seen in Figure 7. The increment of the exponent by one, as well as the normalized mantissa and the guard/sticky bits, depend on the value of the **MSB** of the mantissa multiplication result 'P'. The binary point is between the bits P[46] and P[45]. If the MSB is equal to 1, this means that the number before the binary point is equal to '10' or '11', the MSB is considered the leading one, therefore the binary point has to be shifted to the left by one, and the exponent changes. If the MSB is equal to 0, then the number before the binary point is equal to '01', so the binary point does not get shifted and the exponent stays the same. P[46] will be the leading one in this case.

The sticky bit is the **OR reduced** result of the remaining bits of the 48-bit long mantissa.

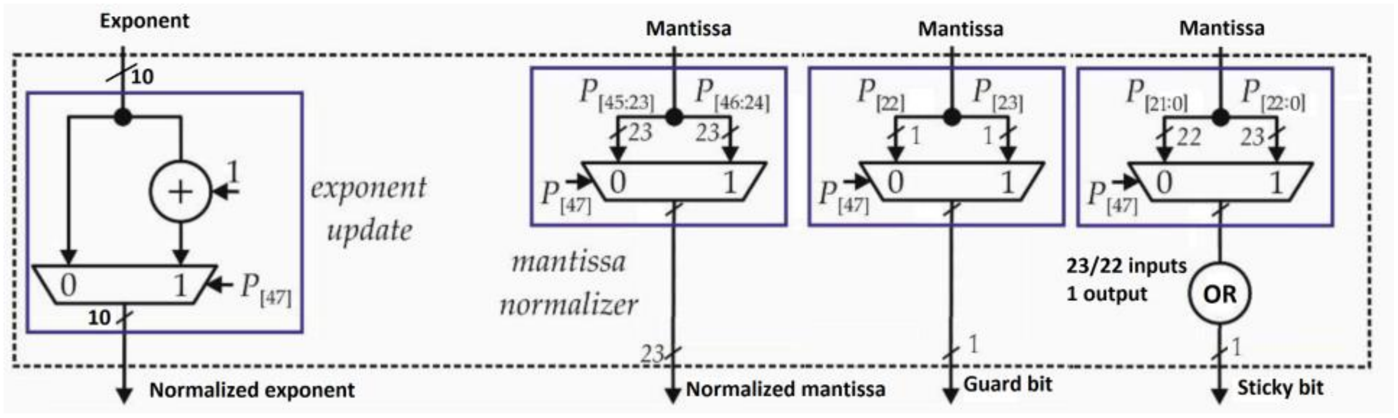


Figure 7: Block diagram of the normalization module. 'P' denotes the 48-bit long mantissa multiplication result. Normalized mantissa does not include the leading one.

3.2.5 Rounding module

The rounding module is responsible for rounding the post-normalization mantissa based on the '**round**' input. The possible values of this input can be seen in Table 4. The encoding of each input is up to you, however the creation of an enum type is strongly advised.

The first input of the module is the **24-bit long Mantissa** which consists of the leading one and the 23-bit long output of the normalization module. Other inputs of the rounding module, are the **guard** bit, and the **sticky** bits. The last input is the **calculated sign** after the first stage of the main module. The outputs of the rounding module are a **25-bit long result** (24 plus 1 more bit for possible overflow), and a 1-bit **inexact** signal, which shows if the result of the mantissa multiplication is exact or not.

To find if the multiplication of the two mantissas leads to an exact number, both the guard and sticky bits must be equal to 0.

The output '**result'** will be based on the '**round**' input. Considering the signals inexact and sign, Table 4, and the round input, create a case that gives the appropriate result value depending on what the 'round' input is equal to. **Warning: The result value can either be the same 24-bit long Mantissa signal as the input or the 24-bit Mantissa increased by one bit.**

If the 'round' value is invalid (default case), then the module must return the value of the IEEE_near rounding to the 'result' signal.

Name	Values	Description
round	IEEE_near	IEEE round to nearest, even. Round to the nearest representable value, if both are equally near, output the result with an even significand.
	IEEE_zero	IEEE round towards zero.
	IEEE_pinf	IEEE round to +Infinity.
	IEEE_ninf	IEEE round to -Infinity.
	near_up	Round to the nearest representable value, if both are equally near, output the result closer to +Infinity.
	away_zero	Round away from zero.

Table 4: Possible values for the 'round' input of the main module and their description.

After the rounding, if the MSB of the 25-bit long resulting Mantissa is equal to 1 then the Mantissa has to be shifted bitwise to the right by one and the post-normalized exponent to increase its value by one (post-rounding normalization), otherwise, both stay the same.

The final 24-bit long Mantissa is called the **post-rounding Mantissa** (*Warning: it includes the leading one*), and the 10-bit exponent is called the **post-rounding exponent**. These two signals can be combined to create an early version of the 'z' output of the multiplier called 'z_calc' that will be used later for the exception handling module. This 'z_calc' has similar form as 'z' meaning that it is a 32-bit long signal with the first bit being the sign bit (1-bit), the next 8 bits being the exponent and the rest being the mantissa (23 bits).

3.2.6 Exception Handling Module

The exception handling module is responsible for handling extreme (corner) cases, like the multiplication of two big numbers that lead to overflow, or the multiplication of infinities.

The inputs of the module are the initial 32-bit long values (**a,b**), the 32-bit long output value calculated after the rounding (**z_calc**), and the **overflow**, **underflow**, and **inexact** bits. You also need the '**round'** input for this module.

The overflow and underflow 1-bit signals have to be calculated inside the main module after the rounding stage. Overflow occurs if the **post-rounding exponent** is bigger than the maximum possible exponent (without considering the two MSBs). Underflow occurs if the **normalized exponent** is smaller than the minimum possible exponent.

Outputs of the module are the 32-bit '**z**' that is connected to the output of the main module, and six 1-bit signals (**zero_f, inf_f, nan_f, tiny_f, huge_f, inexact_f**) that correspond to 6/8 of the status bits. These bits have to be combined after the exception handling in the main module in order to create the '**status**' output.

Inside the module you need to create an enumerated typedef called '**interp_t**' (interpreter), two helpful functions that will use this interpreter and a case block.

The enumerator '**interp_t**' will contain all the possible unsigned groups that the floating point number can be part of, as seen in Figure 3, as well as the minNormal and maxNormal cases. These are ZERO, INF, NORM, MIN_NORM, and MAX_NORM. The groups 'NaNs' and 'Denormals' do not need to be added, since they are considered infinities and zeroes respectively.

The first function is called '**num_interp**' (numeric interpretation) and will return an enum type from the enumerator **interp_t** based on the value of a 32-bit input logic signal. Be careful to consider denormals as zeroes, and NaNs as infinities at this point. The MIN_NORM and MAX_NORM do not need to be included in the function.

The second function called '**z_num**' will do the opposite, so it will take an enum value from the enumerator **interp_t** and based on that, will return the unsigned version (31 bits) of the value. For example, if the enumerator has the value 'ZERO' it will return 31 zero bits. The NORM value does not need to be included in this function.

Finally, inside an always_comb block, initialize the values for the status bit outputs of the module into zeroes. Then based on Table 4, define the value of '**z**' accordingly as well as activate the necessary **status bits**. The sign of ‘z_calc’ as well as the functions you defined above will help you in this case.

A sA.expA.sigA	B sB.expB.sigB	Infinitely Precision Result (F)
± Zero	± Zero	$(-1)^{sA+sB}$ Zero
± Zero	±Norm	$(-1)^{sA+sB}$ Zero
± Zero	± Inf	+ Inf
± Inf	± Inf	$(-1)^{sA+sB}$ Inf
± Inf	±Norm	$(-1)^{sA+sB}$ Inf
± Inf	± Zero	+ Inf
±Norm	± Zero	$(-1)^{sA+sB}$ Zero
±Norm	± Inf	$(-1)^{sA+sB}$ Inf

Table 4: Possible corner case combinations for a and b signals. Combination Norm x Norm is not included in the table.

For the Normal x Normal combination, you need to check in case there is *overflow* or *underflow*. If there is an *overflow*, then based on the ‘round’ input and the sign of the ‘z_calc’ signal, round the output either to maxNormal number or to infinity. Likewise, if there is an *underflow*, then based on the same things, round the output either to minNormal or to zero. If there is neither overflow nor underflow, then the ‘z’ should be equal to ‘z_calc’, and the ‘inexact_f’ should be equal to ‘inexact’.

3.2.7 Testbench

The second task of the coursework requires you to write a testbench in system verilog to test the correctness of the modules you have created in the previous exercise for all 6 possible roundings. The testbench will have two parts, in the first part you will be checking the multiplication result of random values for inputs, while in the second part, you will be checking the result of the corner cases. Choose the **clock period** equal to **10ns**.

For the first part, you may use the ‘*$urandom()*’ function to produce random values for the inputs ‘a’ and ‘b’. You must compare the result ‘z’ of the wrapper with the actual result of the multiplication and display an error in case there is a mismatch between the two. To produce the actual result you can use the function ‘*multiplication*’ that is given to you along with the wrapper. The amount of examined random variables is up to you.

Warnings:

1) *The ‘multiplication’ function gets the ‘round’ input as a string, not as an enum type, so you have to make few adjustments accordingly.*
2) *The ‘multiplication’ function does not have registers, so it calculates the result immediately, in contrast with the fp_mult which calculates the result after 3 cycles. An alignment between these outputs will be needed for correct comparison.*

For the second part, you have to prove the correctness of the multiplication of all the possible corner cases for the two inputs ‘a’ and ‘b’ as defined in Figure 3. It is advised to create a function in this case that matches each corner case with a respective enum type since it will be helpful (i.e. match the binary of positive infinity with a type called ‘pos_inf’). As corner cases for each of the ‘a’ and ‘b’ inputs choose the 12 written below:

- 1 random negative and 1 random positive signaling NaN
- 1 random negative and 1 random positive quiet NaN
- negative and positive infinity
- 1 random negative and 1 random positive normal
- 1 random negative and 1 random positive denormal
- negative and positive zero

You have to test all the possible combinations of the corner cases between the ‘a’ and ‘b’ signals (12x12 = 144 combinations) and check if the result of the wrapper is equal to the result of the *multiplication* function for each combination.

3.2.8 System Verilog Assertions

Based on Table 3, you may have realized that some of the status bits cannot assert at the same time. The first part of this task is to write all possible *Immediate Assertions,* testing each time that **two** specific status bits do not assert to 1 together. You need to identify which combinations of two status bits **should never** assert at the same time, based on the descriptions given above.

For the second part, you need to write five *Concurrent Assertions*. These assertions will be responsible for checking the following things (at each positive clock edge):

- If the ‘**zero**’ status bit asserts to 1 then at the same cycle all the bits of the exponent of ‘z’ must be equal to 0.
- If the ‘**inf**’ status bit asserts to 1 then at the same cycle all the bits of the exponent of ‘z’ must be equal to 1.
- If the ‘**nan**’ status bit asserts to 1 then 3 cycles before all the bits of the exponent of ‘a’ must be equal to 0 and the bits of the exponent of ‘b’ must be equal to 1 or the opposite.
- If the ‘**huge**’ status bit asserts to 1 then at the same cycle all the bits of the exponent of ‘z’ must be equal to 1, or all the bits of the exponent of ‘z’ except the LSB must be equal to 1, the LSB must be 0 and all the bits of the mantissa of ‘z’ to be equal to 1 (maxNormal case).
- If the ‘**tiny**’ status bit asserts to 1 then at the same cycle all the bits of the exponent of ‘z’ must be equal to 0, or all the bits of the exponent of ‘z’ except the LSB must be equal to 0, the LSB must be 1 and all the bits of the mantissa of ‘z’ to be equal to 0 (minNormal case).

Implement the two parts in two different modules called ‘**test_status_bits**’ and ‘**test_status_z_combinations**’ respectively, and then bind the two modules with the wrapper module.

4 Deliverables for Submission

For the multiplier discussed in Section 3.2, you SHOULD:

- Produce the SystemVerilog code for the modules fp_mult, normalize_mult, exception_mult, and round_mult (**4** .sv files)

- Produce a testbench that demonstrates the correct operation of the circuit as well as the correct operation of the corner cases (**1** .sv file)
- Using SVA, provide the necessary properties that produce proper messages on the simulator Transcript window (std out) to report on the PASS/FAIL of the additional checks required. (**1** .sv file)

You should prepare a report (in **pdf** format) that will describe in more detail your thought process, what you have designed and how you have verified the DUT, as well as contain screenshots from the respective simulations. The system verilog files should be included in separate folders based on the exercise (i.e. a folder named "exercise1" should include the 4 sv files from the first exercise, a folder named "exercise2" should include the testbench and a folder named "exercise3" should include the SVA module). If you add any more modules that are not mentioned in the coursework, you should describe these modules and their operation in your pdf report. Submit these folders along with the pdf report in a zip file named *<lastname_AEM>.zip* on elearning.

Warning: Please do not include greek characters in the names of your folders since elearning changes them into something incoherent whenever I download them. Your pdf report however can be in either english or greek, this is left up to you.

The coursework takes 3 marks out of the total mark (i.e., 30%) and is compulsory. If the coursework is not submitted on time, no marks are given but you can take the exam. In this case, the maximum mark that can be achieved is 7. The mark for the coursework is maintained for the next two exam periods ONLY (Sept. 2025, Feb. 2026).

The submission deadline will be until 30/5/2025.

Good Luck!!

5 References

[1]. ModelSim [Online]. Available: https://eda.sw.siemens.com/en-US/ic/modelsim/ (accessed on 12/5/22).

[2]. Questa*-Intel® FPGA Edition [Online]. Available: https://www.intel.com/content/www/us/en/software-kit/684216/intel-quartus-prime-lite-edition-design-software-version-21-1-for-windows.html (accessed on 12/5/2022).

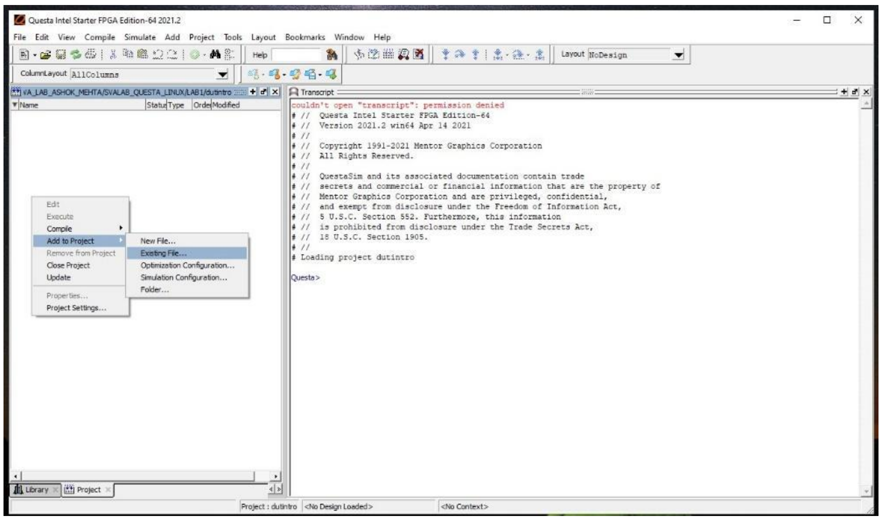
[3]. Libero [Online]. Available: https://www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/fpga/libero-software-later-versions (accessed on 12/5/2022).

[4]. https://www.intel.com/content/www/us/en/docs/programmable/683472/22-1/minimum-hardware-requirements.html (accessed on 13/5/2022).

[5]. Intel® FPGA Self Service Licensing Center (SSLC) [Online]. Available: https://tinyurl.com/yc8f8es4 (accessed 13/5/2022).

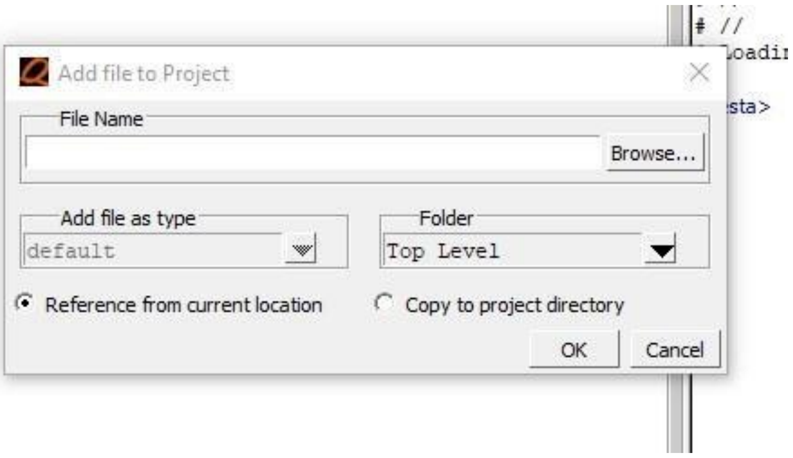
6 Appendix

The following figures help you understand the simulation tool and should be read with subsection 3.1.2.


Questa Intel Starter FPGA Edition-64 2021.2
File Edit View Compile Simulate Add Project Tools Layout Bookmarks Window Help
Layout NoDesign
ColumnLayout AllColumns
VA_LAB_ASHOK_MEHTA/SVALAB_QUESTA_LINUX/LAB1/dutintro
Name Status Type Order Modified
Edit
Execute
Compile
Add to Project
Remove from Project
Close Project
Update
Properties...
Project Settings...
New File...
Existing File...
Optimization Configuration...
Simulation Configuration...
Folder...
Library
Project
Project : dutintro
<No Design Loaded>
<No Context>

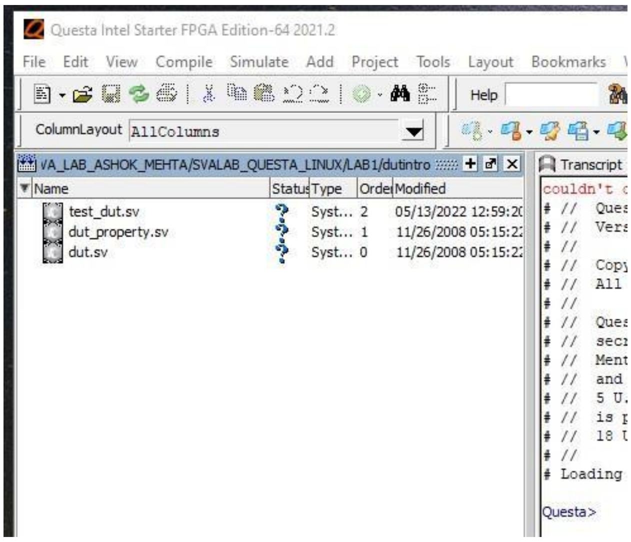
```
couldn't open "transcript": permission denied
# //  Questa Intel Starter FPGA Edition-64
# //  Version 2021.2 win64 Apr 14 2021
# //
# //  Copyright 1991-2021 Mentor Graphics Corporation
# //  All Rights Reserved.
# //
# //  QuestaSim and its associated documentation contain trade
# //  secrets and commercial or financial information that are the property of
# //  Mentor Graphics Corporation and are privileged, confidential,
# //  and exempt from disclosure under the Freedom of Information Act,
# //  5 U.S.C. Section 552. Furthermore, this information
# //  is prohibited from disclosure under the Trade Secrets Act,
# //  18 U.S.C. Section 1905.
# //
# Loading project dutintro

Questa>
```

Fig. 6. Started a new project and ready to add files (right click in the project area for these menus to appear).



Add file as type: default
Folder: Top Level
[✓ Reference from current location] Copy to project directory

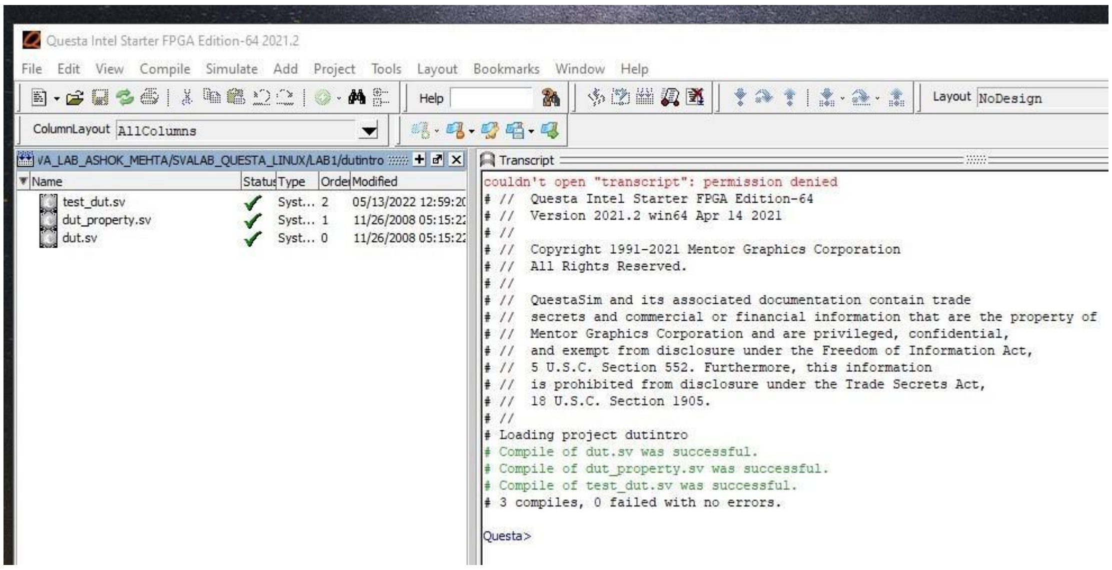
Fig. 7. Browse and pick the files you need (copy or not the files to the project directory).


Questa Intel Starter FPGA Edition-64 2021.2
File Edit View Compile Simulate Add Project Tools Layout Bookmarks
Help
ColumnLayout AllColumns
VA_LAB_ASHOK_MEHTA/SVALAB_QUESTA_LINUX/LAB1/dutintro
Transcript

Name	Status	Type	Order	Modified
test_dut.sv	?	Syst...	2	05/13/2022 12:59:2(
dut_property.sv	?	Syst...	1	11/26/2008 05:15:2:
dut.sv	?	Syst...	0	11/26/2008 05:15:2:

```
couldn't c
# //  Ques
# //  Vers
# //
# //  Copy
# //  All
# //
# //  Ques
# //  secr
# //  Ment
# //  and
# //  5 U.
# //  is p
# //  18 U
# //
# Loading

Questa>
```

Fig. 8. Files were added to the project and are now ready to be compiled.

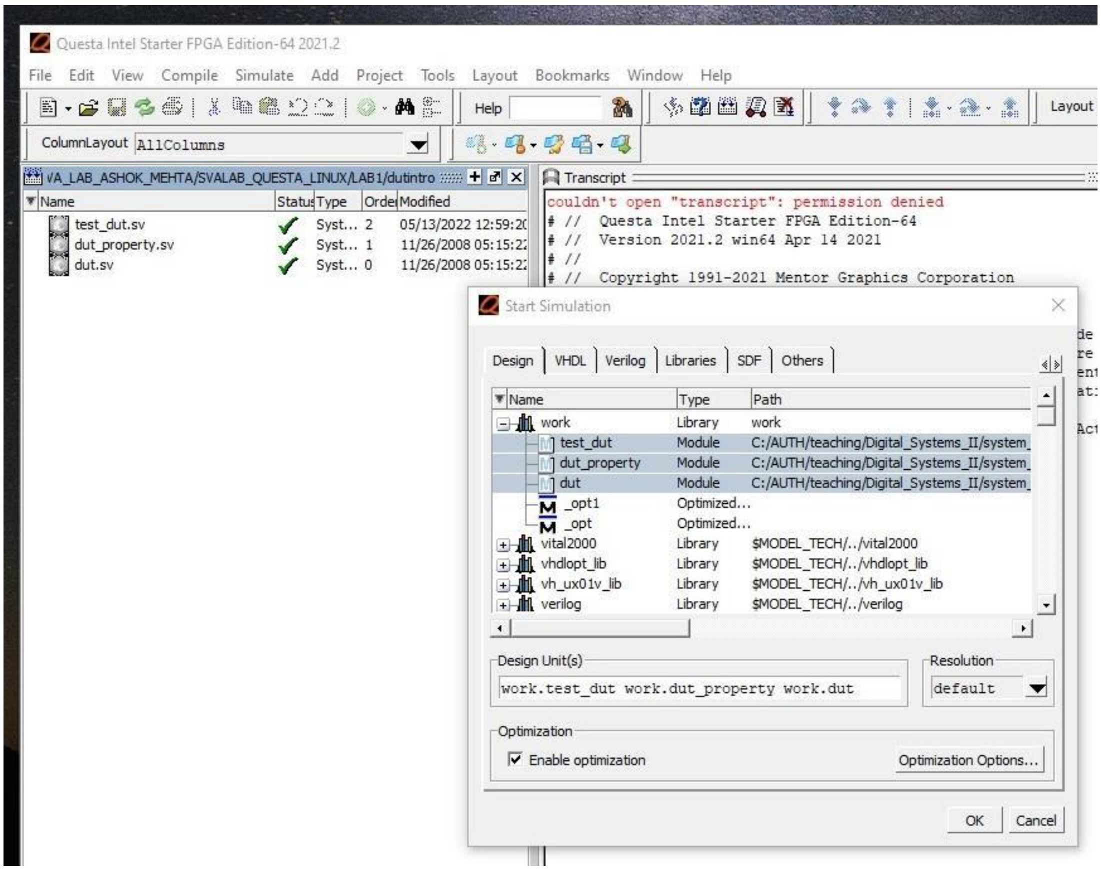

Questa Intel Starter FPGA Edition-64 2021.2
File Edit View Compile Simulate Add Project Tools Layout Bookmarks Window Help
Help
Layout NoDesign
ColumnLayout AllColumns
Transcript

/A_LAB_ASHOK_MEHTA/SVALAB_QUESTA_LINUX/LAB1/dutintro

Name	Status	Type	Order	Modified
test_dut.sv	✓	Syst...	2	05/13/2022 12:59:2
dut_property.sv	✓	Syst...	1	11/26/2008 05:15:2
dut.sv	✓	Syst...	0	11/26/2008 05:15:2

```
couldn't open "transcript": permission denied
# //  Questa Intel Starter FPGA Edition-64
# //  Version 2021.2 win64 Apr 14 2021
# //
# //  Copyright 1991-2021 Mentor Graphics Corporation
# //  All Rights Reserved.
# //
# //  QuestaSim and its associated documentation contain trade
# //  secrets and commercial or financial information that are the property of
# //  Mentor Graphics Corporation and are privileged, confidential,
# //  and exempt from disclosure under the Freedom of Information Act,
# //  5 U.S.C. Section 552. Furthermore, this information
# //  is prohibited from disclosure under the Trade Secrets Act,
# //  18 U.S.C. Section 1905.
# //
# Loading project dutintro
# Compile of dut.sv was successful.
# Compile of dut_property.sv was successful.
# Compile of test_dut.sv was successful.
# 3 compiles, 0 failed with no errors.

Questa>
```

Fig. 9. All files have now been complied (question marks turns to ticks).


Questa Intel Starter FPGA Edition-64 2021.2
File Edit View Compile Simulate Add Project Tools Layout Bookmarks Window Help
Help
Layout
ColumnLayout AllColumns
Transcript
Start Simulation
Design VHDL Verilog Libraries SDF Others
OK
Cancel
Optimization Options...

/A_LAB_ASHOK_MEHTA/SVALAB_QUESTA_LINUX/LAB1/dutintro

Name	Status	Type	Order	Modified
test_dut.sv	✓	Syst...	2	05/13/2022 12:59:2
dut_property.sv	✓	Syst...	1	11/26/2008 05:15:2
dut.sv	✓	Syst...	0	11/26/2008 05:15:2

```
couldn't open "transcript": permission denied
# //  Questa Intel Starter FPGA Edition-64
# //  Version 2021.2 win64 Apr 14 2021
# //
# //  Copyright 1991-2021 Mentor Graphics Corporation
```

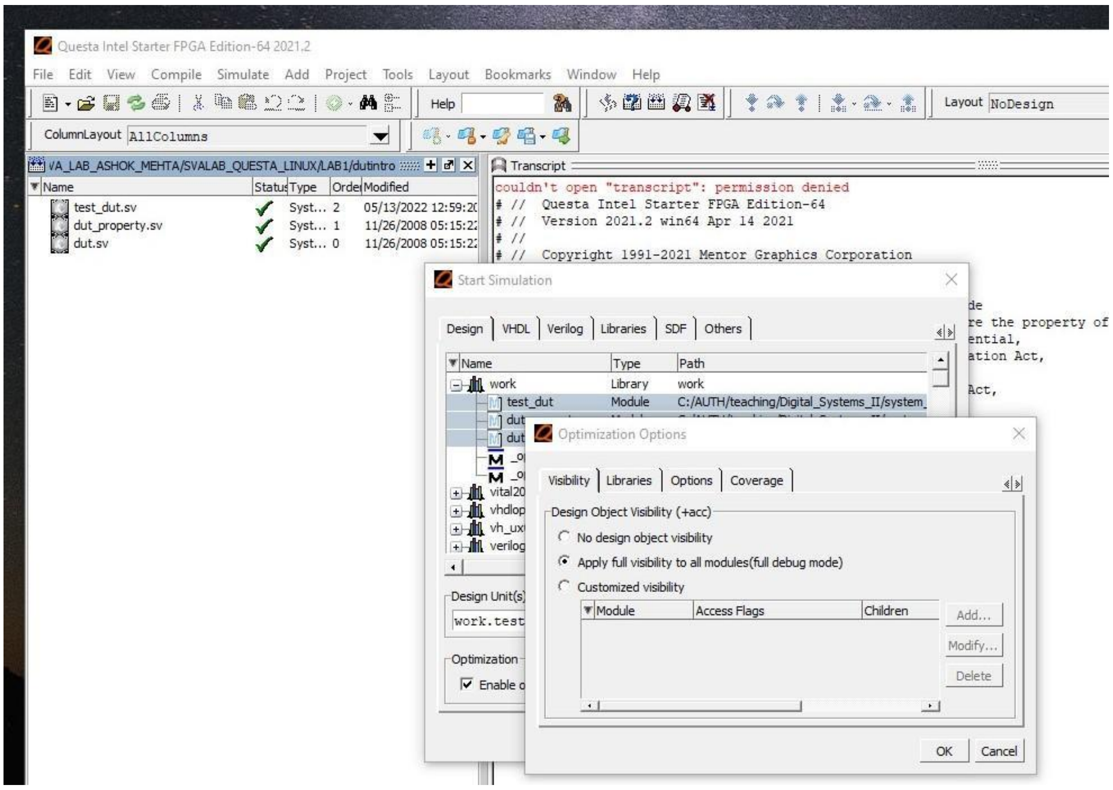
Name	Type	Path
work	Library	work
test_dut	Module	C:/AUTH/teaching/Digital_Systems_II/system
dut_property	Module	C:/AUTH/teaching/Digital_Systems_II/system
dut	Module	C:/AUTH/teaching/Digital_Systems_II/system
_opt1	Optimized...	
_opt	Optimized...	
vital2000	Library	$MODEL_TECH/../vital2000
vhdlopt_lib	Library	$MODEL_TECH/../vhdlopt_lib
vh_ux01v_lib	Library	$MODEL_TECH/../vh_ux01v_lib
verilog	Library	$MODEL_TECH/../verilog

Design Unit(s): work.test_dut work.dut_property work.dut

Resolution: default

Optimization: [✓] Enable optimization

Fig. 10. Click on Simulate -> Start Simulation and select the compiled files to be simulated. Click on optimization options (see next Figure).


Questa Intel Starter FPGA Edition-64 2021.2
File Edit View Compile Simulate Add Project Tools Layout Bookmarks Window Help
Layout NoDesign
ColumnLayout AllColumns
Transcript
Start Simulation
Design VHDL Verilog Libraries SDF Others
Design Unit(s)
Optimization
Enable o
Optimization Options
Visibility Libraries Options Coverage
No design object visibility
Customized visibility
Module Access Flags Children
Add... Modify... Delete
OK Cancel

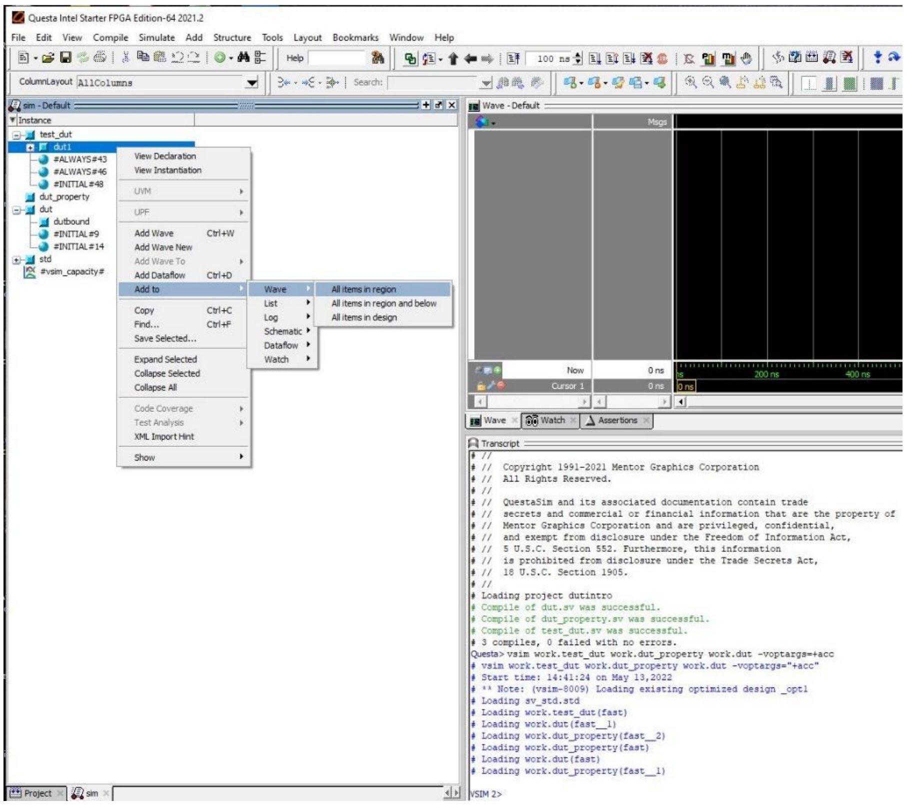
Name	Status	Type	Order	Modified
test_dut.sv	✓	Syst...	2	05/13/2022 12:59:2
dut_property.sv	✓	Syst...	1	11/26/2008 05:15:2
dut.sv	✓	Syst...	0	11/26/2008 05:15:2

```
couldn't open "transcript": permission denied
# //  Questa Intel Starter FPGA Edition-64
# //  Version 2021.2 win64 Apr 14 2021
# //
# //  Copyright 1991-2021 Mentor Graphics Corporation
```

Name	Type	Path
work	Library	work
test_dut	Module	C:/AUTH/teaching/Digital_Systems_II/system_

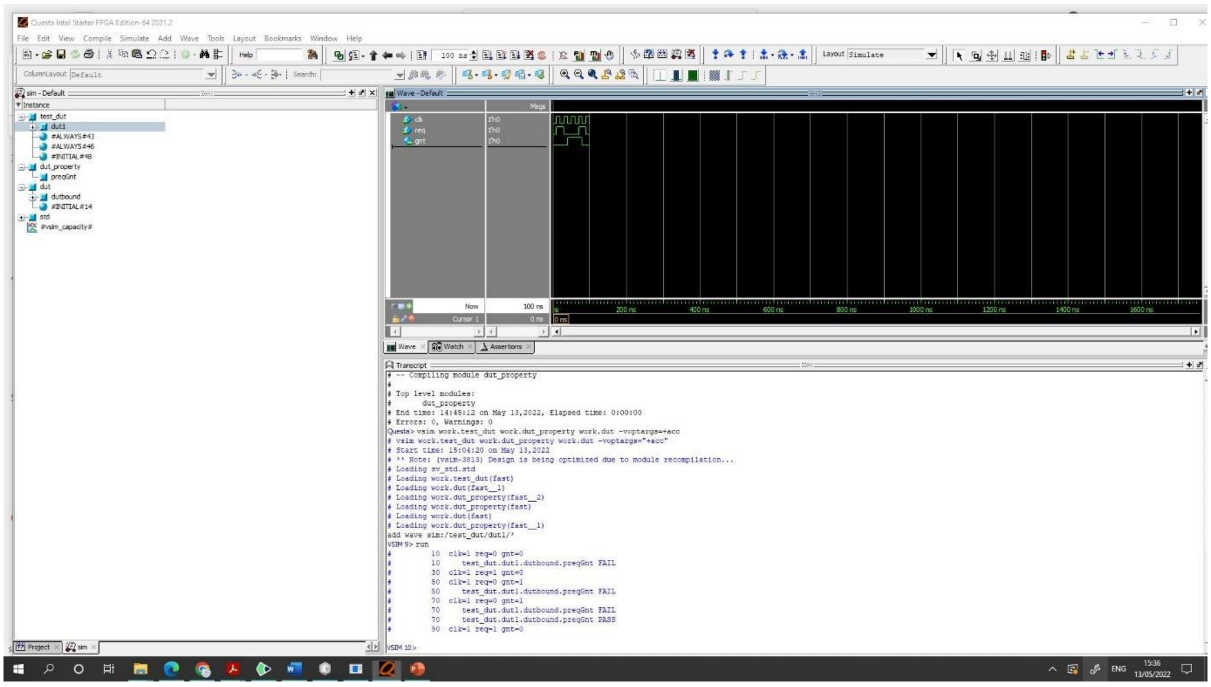
Design Object Visibility (+acc): Apply full visibility to all modules(full debug mode) [✓]

Fig. 11. Full visibility allows you to display all internals signals in the waveform window.



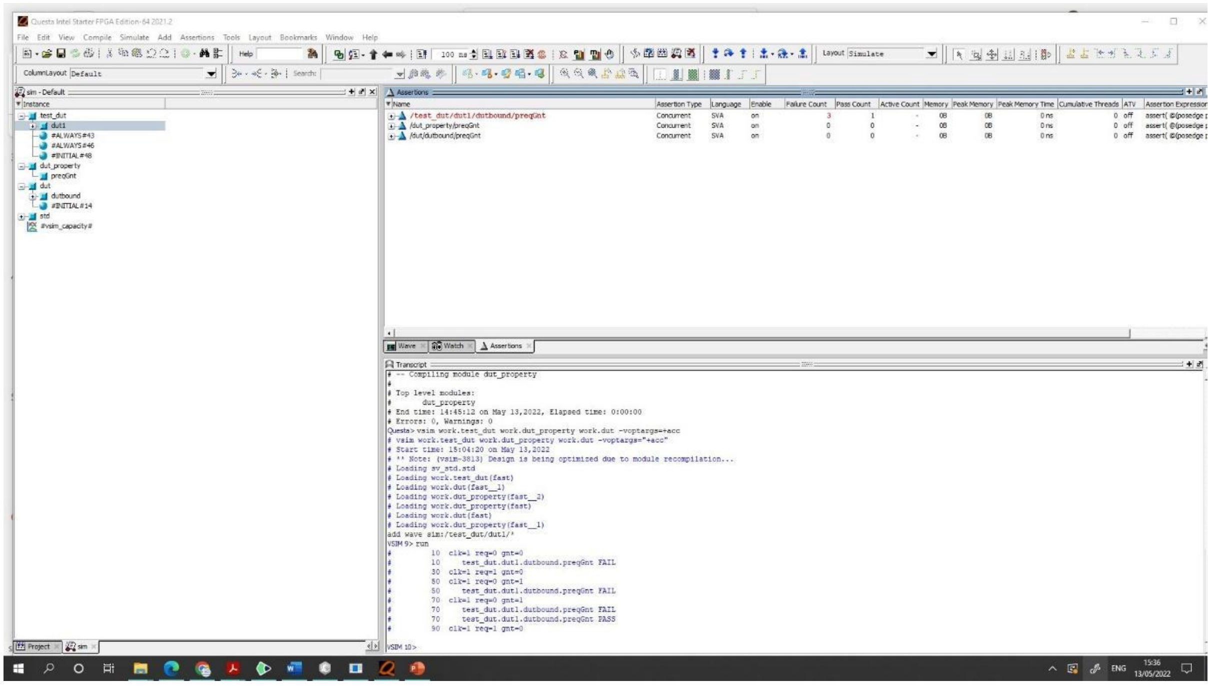
```
# //
# //  Copyright 1991-2021 Mentor Graphics Corporation
# //  All Rights Reserved.
# //
# //  QuestaSim and its associated documentation contain trade
# //  secrets and commercial or financial information that are the property of
# //  Mentor Graphics Corporation and are privileged, confidential,
# //  and exempt from disclosure under the Freedom of Information Act,
# //  5 U.S.C. Section 552. Furthermore, this information
# //  is prohibited from disclosure under the Trade Secrets Act,
# //  18 U.S.C. Section 1905.
# //
# Loading project dutintro
# Compile of dut.sv was successful.
# Compile of dut_property.sv was successful.
# Compile of test_dut.sv was successful.
# 3 compiles, 0 failed with no errors.
Questa> vsim work.test_dut work.dut_property work.dut -voptargs=+acc
# vsim work.test_dut work.dut_property work.dut -voptargs="+acc"
# Start time: 14:41:24 on May 13,2022
# ** Note: (vsim-8009) Loading existing optimized design _opt1
# Loading sv_std.std
# Loading work.test_dut(fast)
# Loading work.dut(fast__1)
# Loading work.dut_property(fast__2)
# Loading work.dut_property(fast)
# Loading work.dut(fast)
# Loading work.dut_property(fast__1)

VSIM 2>
```

Fig. 12. Once the files are loaded for simulation, you can select which signals to be displayed (right click).



```
# -- Compiling module dut_property
#
# Top level modules:
#       dut_property
# End time: 14:45:12 on May 13,2022, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
Questa> vsim work.test_dut work.dut_property work.dut -voptargs=+acc
# vsim work.test_dut work.dut_property work.dut -voptargs="+acc"
# Start time: 15:04:20 on May 13,2022
# ** Note: (vsim-3813) Design is being optimized due to module recompilation...
# Loading sv_std.std
# Loading work.test_dut(fast)
# Loading work.dut(fast__1)
# Loading work.dut_property(fast__2)
# Loading work.dut_property(fast)
# Loading work.dut(fast)
# Loading work.dut_property(fast__1)
add wave sim:/test_dut/dut1/*
VSIM 9> run
#          10  clk=1 req=0 gnt=0
#          10     test_dut.dut1.dutbound.preqGnt FAIL
#          30  clk=1 req=1 gnt=0
#          50  clk=1 req=0 gnt=1
#          50     test_dut.dut1.dutbound.preqGnt FAIL
#          70  clk=1 req=0 gnt=1
#          70     test_dut.dut1.dutbound.preqGnt FAIL
#          70     test_dut.dut1.dutbound.preqGnt PASS
#          90  clk=1 req=1 gnt=0

VSIM 10>
```

Fig. 13. Simulated DUT. See also the simulator transcript window at the bottom and the related reports.



Name	Assertion Type	Language	Enable	Failure Count	Pass Count	Active Count	Memory	Peak Memory	Peak Memory Time	Cumulative Threads	ATV	Assertion Expression
/test_dut/dut1/dutbound/preqGnt	Concurrent	SVA	on	3	1	-	0B	0B	0 ns	0	off	assert(@(posedge
/dut_property/preqGnt	Concurrent	SVA	on	0	0	-	0B	0B	0 ns	0	off	assert(@(posedge
/dut/dutbound/preqGnt	Concurrent	SVA	on	0	0	-	0B	0B	0 ns	0	off	assert(@(posedge

```
# -- Compiling module dut_property
#
# Top level modules:
#       dut_property
# End time: 14:45:12 on May 13,2022, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
Questa> vsim work.test_dut work.dut_property work.dut -voptargs=+acc
# vsim work.test_dut work.dut_property work.dut -voptargs="+acc"
# Start time: 15:04:20 on May 13,2022
# ** Note: (vsim-3813) Design is being optimized due to module recompilation...
# Loading sv_std.std
# Loading work.test_dut(fast)
# Loading work.dut(fast__1)
# Loading work.dut_property(fast__2)
# Loading work.dut_property(fast)
# Loading work.dut(fast)
# Loading work.dut_property(fast__1)
add wave sim:/test_dut/dut1/*
VSIM 9> run
#          10  clk=1 req=0 gnt=0
#          10     test_dut.dut1.dutbound.preqGnt FAIL
#          30  clk=1 req=1 gnt=0
#          50  clk=1 req=0 gnt=1
#          50     test_dut.dut1.dutbound.preqGnt FAIL
#          70  clk=1 req=0 gnt=1
#          70     test_dut.dut1.dutbound.preqGnt FAIL
#          70     test_dut.dut1.dutbound.preqGnt PASS
#          90  clk=1 req=1 gnt=0

VSIM 10>
```

Fig. 14. Simulated DUT. The Assertions tab displays all related info about the properties checked in file dut_property.sv