ECE1756 Assignment 1
Speed & Area Trade-offs in FPGA Design

"There are no solutions ... there are only trade-offs."

— Thomas Sowell

Assigned on Tuesday, September 19   **Total marks =   13/100**
Due on Monday, October 9 @ 11:59 pm

1 Objectives

This assignment is intended to help you:

1	Learn how to use several FPGA CAD tools.

- Intel Quartus II Prime for synthesis and analysis
- ModelSim for simulating digital circuits

2	Refresh your hardware design skills.

- Design hardware circuits in RTL (e.g. Verilog, VHDL)
- Use a given Verilog testbench to verify that your design is functionally correct and meets the requested interface specifications.

3	Explore area, speed, and power trade-offs for different implementation styles of a simple circuit on FPGAs.

- Understand the pros and cons of different FPGA implementations (baseline, pipelined, and shared-hardware) of an application circuit.
- Optimize both pipelined and shared-hardware designs.

2 Handout Overview

This handout provides a step-by-step tutorial on how to use the Quartus and ModelSim tools with a sample circuit that implements a commonly used mathematical function in many applications, the exponential function e^x. The input to this circuit x is a stream of 16-bit fixed-point numbers in Q2.14 format (i.e. 2 integer bits and 14 fractional bits), and the output y is a stream of 32-bit fixed-point numbers in Q7.25 format. The tutorial guides you through the process of compiling, simulating and analyzing the design for speed, resource usage and power. Then, you are then asked to implement two new versions of the hardware design (one pipelined for the highest speed, and one using resource sharing for the smallest area), and evaluate their speed, area, power and efficiency.

3   Deliverables

For this assignment, you are asked to hand in the following:

1. **Two Quartus project archives** for the pipelined and shared hardware designs that you implement. Use `Project > Archive Project` in Quartus to archive a project. The default options are fine since they save the source files but not the output files. Upload these files to Quercus using the naming convention `lab1_<pipelined/sharedhw>_<firstname>_<lastname>`. Any HDL files you create in this lab should be well commented and structured, and should use informative signal/variable names. Coding style will be evaluated.

2. **A LaTeX-typed report**[1] in PDF format using this template on Overleaf. You can also use this service to write your report if you do not want to install Latex on your machine. Your report should include the following:

   - A block diagram (which could come from the Quartus RTL Viewer or could be hand-drawn, but in either case should make your high-level design understandable) of the two circuits you implemented and a description of how they work.
   - Readable simulation waveforms and testbench output for each of the two designs you completed to show each design functions correctly.
   - A table, similar to Table 1, for the results. Briefly show how you arrived at these answers.
   - A discussion section covering the following questions:

   (a) What are the different sources of error (i.e. difference between $exp(x)$ and Hardware Output in the graph you plotted for the testbench output)? What changes could you make to the circuit to reduce this error?

   (b) Which of the 3 hardware circuits (baseline, pipelined, and shared) achieves the highest throughput/device? Explain the reasons for the efficiency differences between them.

   (c) Look at the average toggle rates (how often the average signal changes) for the 3 circuits (this information is in the messages section of the PowerAnalyzer report). Explain why some circuit styles lead to higher toggle rates than others. Comment on the relative efficiency of the 3 circuits in terms of computations/J, and explain why each style is more or less efficient in computations/J than the others.

   - An appendix that includes the HDL source codes of the two circuits that you implement (pipelined and shared hardware).

4   Background

In this assignment, you will be implementing a digital circuit that computes the exponential function e^x for 16-bit fixed-point input values. The exponential function is a simple but very common mathematical operation in many applications. For example, it is heavily used in deep neural networks in the *softmax* function used to calculate the final prediction values based on the model's outputs. The softmax function is shown in Eq. (1), where $i = 1, ..., N$ and $\mathbf{z} = [z_1, ..., z_N]$ is an N-element vector. For example, if the input vector to the softmax function (which contains the output scores for different classes for example) is $\mathbf{z} = [-4, 3, -2, -1, 2]$, the softmax function is computed by

---

[1]As a graduate student, you will heavily use LaTeX to write reports, papers, and your thesis. In case you never used it before, this is a good chance to start learning about it! You can find a nice introductory tutorial for Latex here.

Table 1: Implementation results for the 3 different implementations of the studied circuit.

	Baseline Circuit	Pipelined Circuit	Shared HW Circuit
Resources for one circuit	21 ALMs + 9 DSPs		
Operating frequency	44.8 MHz		
Critical path	DSP mult-add + 2x DSP mult + LE-based adder + 6x DSP mult + LE-based adder		
Cycles per valid output	1		
Max. # of copies/device			
Max. Throughput for a full device (computations/s)			
Dynamic power of one circuit @ 42 MHz	1.96 mW		
Max. throughput/Watt for a full device			

calculating the elementwise exponential function of all the vector elements and then dividing each of them by the sum of all. The resultant vector for this example is $\sigma = [0.001, 0.717, 0.005, 0.013, 0.264]$ indicating that the model's top prediction is the second class with 71.7% confidence. Also notice that all the values in the result vector have to sum up to 1 (i.e. predictions sum up to 100%).

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}} \tag{1}$$

There are many different approaches for implementing the exponential function (e^x) in hardware. We will implement a relatively simple and approximate version of it based on Taylor's expansion. The Taylor expansion approximates a function with an n-degree polynomial, such that the approximation is more accurate as the value of n increases. The Taylor polynomial for the exponential function is shown in Eq. (2). In this assignment, you are only required to implement the 5th Taylor polynomial (i.e. the first 6 terms of the approximation polynomial), but feel free to experiment with implementing higher degrees of the Taylor polynomial and show the effect of that on the approximation accuracy using the provided plotting script.

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + ... \tag{2}$$

5   Getting Started with Quartus: The Baseline Implementation

Quartus is the FPGA computer-aided design (CAD) flow used to synthesize, place, route and configure a user design on Intel FPGA devices. Each FPGA vendor typically has its own FPGA CAD tools such as Vivado from Xilinx and Libero from Microsemi. The main stages of all FPGA

CAD tools are somewhat similar. They all start by synthesizing the user design from a hardware description language (HDL) such as Verilog or VHDL into FPGA components (e.g. logic elements, memory blocks, etc.). Then, the blocks of the synthesized design are placed at specific locations on the FPGA and the programmable routing is configured to form the connections between these blocks. Finally, the CAD tool produces a bitstream (analogous to a program executable) that can be used to configure the FPGA device. Most CAD tools offer additional functionality such as timing and power analysis, simulation, system integration tools, high-level design tools, and others. Therefore, the experience that you gain by learning how to use the Quartus design tools in this course is transferable, with some effort, to other FPGA CAD tools from different vendors.

You can find basic tutorials on how to use Quartus here. The "Introduction to Intel Quartus Prime Software", "Timing Analyzer in Intel Quartus Software", and "Use ModelSim with Testbenches" are the most relevant tutorials for this course. Although we will give a short tutorial on how to use these tools in the context of this assignment, you might want to invest some time skimming through the contents of these tutorials for more details.

> **~90mins** For more detailed tutorials on the design tools we will use throughout the course, check out the tutorials in this link. The "Introduction to Intel Quartus Prime Software", "Timing Analyzer in Intel Quartus Software", and "Use ModelSim with Testbenches" are the most relevant ones.

Appendix A of this handout shows different ways you can access the tools for this assignment. We recommend that you use the already set up tools on the UG machines but, the appendix also explains how to install the tools on your own machine.

> **~60mins** Follow the steps in Appendices A and B to access the tools you will need to complete this assignment.

Download the `lab1.zip` archive from Quercus, and unzip it in your working directory. The archive contains the following files:

- `lab1.sv`: Verilog code for the baseline implementation for the circuit in Fig. 2
- `lab1_tb.v`: Verilog testbench
- `lab1.sdc`: Timing constraint file
- `lab1.qpf`: Quartus project file
- `lab1.qsf`: Quartus settings file
- `graph.gnu`: A GNU plotting script to draw the exact vs. hardware output results for your hardware implementation of the exponential function

A verilog testbench simulates the presence of hardware that passes inputs to your circuit and hardware that operates on outputs from your circuit (e.g. compare it to golden reference outputs). To interface with the testbench that we provide, you will need to use/generate the following signals:

- `i_x`: 16-bit input data for your circuit to process in the **Q2.14 fixed-point format** (2 integer bits and 14 fractional bits).
- `o_y`: 32-bit output data computed by your circuit in the **Q7.25 fixed-point format** (7 integer bits and 25 fractional bits).

- `i_valid`: an input to your circuit; it is 1 when `i_x` contains a valid number.
- `o_valid`: an output from your circuit; it is 1 when your circuit outputs a valid `o_y` number.
- `i_ready`: an input to your circuit; it is 1 when the downstream circuit is ready to accept a new value of `o_y`.
- `o_ready`: an output from your circuit; it is 1 when your circuit is ready for a new `i_x` number.

The following rules apply to all the valid and ready signals:

1. If a valid signal is low on the rising edge of the clock, the receiver circuit will not use the corresponding input data to compute a valid output.
2. If a ready signal is low on the clock's rising edge, the sender won't send valid data that cycle.
3. The sender circuit will not mark the same piece of data as valid for more than one rising edge.

The waveform in Fig. 1 further clarifies how the interface between the testbench and your circuit works. A sample sequence of events is described for an example circuit that takes 1 cycle to compute valid output:

- In cycle 1 the testbench begins to produce valid inputs; `i_valid` is set to 1. Your circuit latches the valid value of `i_x` on the positive edge of cycle 2, performs computation, and outputs a valid value of `o_y` on the positive edge of cycle 3.
- On the rising edge of cycle 4 your circuit sees that there is no valid data (`i_valid` is low); therefore there are no computations to perform in cycle 4, and in cycle 5 there is no valid data to output. As long as `o_valid` is low, the value of `o_y` does not matter.
- Operation continues normally until cycle 8. In cycle 8, the testbench indicates that it is not ready to receive outputs from your circuit and sets `i_ready` to 0. This means that it will not be ready to latch data on the rising edge of cycle 9. The following sequence of events occurs:
  - Because the testbench is not ready to receive `o_y` values, your circuit must stall and preserve the value of `o_y`, represented by $f(n+5)$, until the rising edge of cycle 10 when the receiver is ready again. Your circuit also sets `o_valid` to 0 in accordance with rule 2 in the list above.
  - To avoid missing valid input data, your circuit also applies back-pressure by setting `o_ready` to 0. In keeping with protocol, the testbench sets the `i_valid` signal to 0 to indicate that on the rising edge of cycle 9 there will be no valid data.

> ⚠ Your circuit must abide to the specification of this interface and you are not allowed to change the testbench code to accommodate for a different interface.
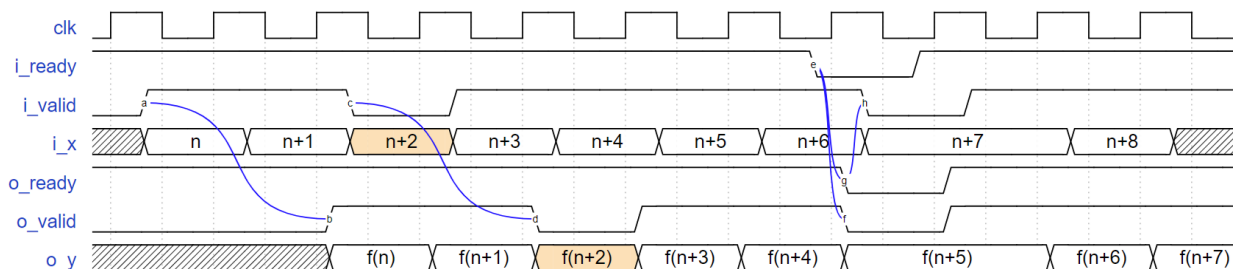

Figure 1: Waveform of the interface between your circuit and the testbench. Note that signals are slightly delayed from the clock rising edge to show an exaggerated effect of signal propagation delays.

Later in this assignment, you will explore different implementation styles to compute the 5th Taylor polynomial of the exponential function, but for this tutorial you will use the circuit shown in Fig. 2 which corresponds to the Verilog code we provide in the `lab1.sv` file.
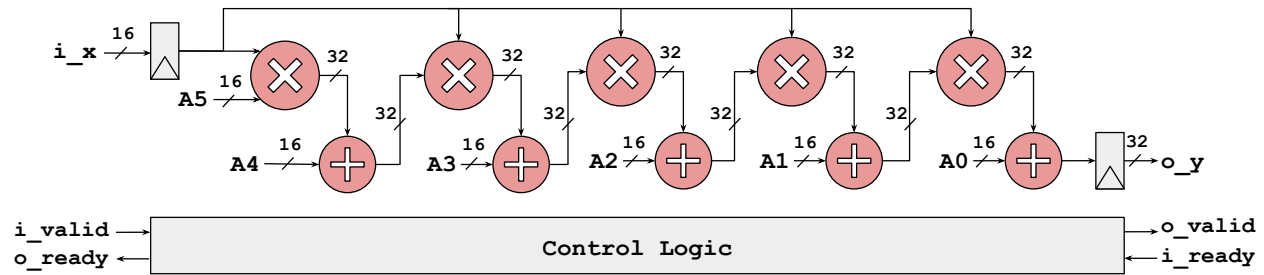
Figure 2: The baseline circuit for computing the 5th Taylor polynomial approximating e^x. For this implementations, eq. (2) is factored as $y = ((((a_5x + a_4)x + a_3)x + a_2)x + a_1)x + a_0$, where $a_0 = 1, a_1 = 1, a_2 = \frac{1}{2}, a_3 = \frac{1}{6}, a_4 = \frac{1}{24}$, and $a_5 = \frac{1}{120}$

5.1 Opening the Project

If you are using the UG machines, start Quartus by typing the following command in the terminal.

```
quartus.start &
```

Select `File > Open Project` and choose the Quartus project file, `lab1.qpf`. This project has the device set to the fastest (I1) speed grade of an Arria 10 GX device (10AX115N2F45I1SG), which is a large 20 nm FPGA.

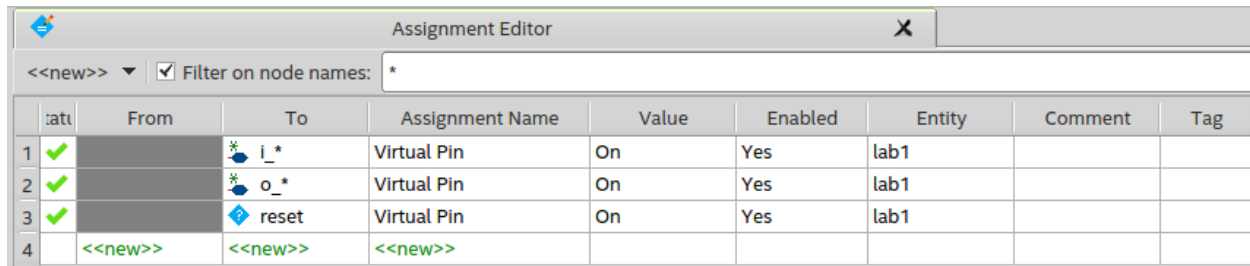## 5.2 Setting Up Virtual I/O Pins

Sometimes you will find yourself working on a sub-module of a larger digital design in isolation. For example, the circuit in this assignment may be interfaced with other hardware in the FPGA after it's designed and tested to implement a larger softmax core. However, if you were to simply compile the project, Quartus will connect all the inputs and outputs of your circuit to I/O pins in the FPGA. This may be undesirable for two reasons:

1. I/O pins have buffers which introduce a lot of delay to your circuit.
2. I/O pins are scarce resources which will constrain the placement of your circuit inside the FPGA, and this may be very sub-optimal and not representative of the circuit placement when your module is in a larger design and connected to other logic, rather than I/O pins.

Since we want to test the maximum performance of the circuit, we will tell Quartus not to connect the module inputs/outputs to FPGA I/O pins.

Open the Assignment Editor through `Assignments > Assignment Editor`. The Assignment Editor is used to assign properties to various FPGA resources. You should see that the 'Virtual Pin' property is set to 'On' for the module inputs, outputs and reset signal. Your Assignment Editor should now look similar to Fig. 4 below (note that i_* will match any signal starting with i_).

Note that we have not assigned the clock to be a virtual pin. This is because virtual pins hook into the general routing of the FPGA, but clocks need to remain on dedicated clock routing networks and these are most easily reached (driven) by I/Os and PLLs.

Figure 3: Assigning virtual pins using the Assignment Editor.

5.3   Compilation and Design Visualization

Choose `Processing > Start Compilation`. The design will be run through the default compiler flow, which is Analysis & Synthesis → Fitter (Place & Route) → Assembler (Generate programming files) → Timing Analysis → EDA Netlist Writer. The design should compile successfully, but fail to meet the timing requirements. Timing constraints are specified via SDC (Synopsys Design Constraint) files, in this case in `lab1.sdc`. Looking at this file, you should see:

```
1    create_clock -period 1 -name clk clk # in lab1.sdc
```

This SDC file has created a timing constraint for the clock net (called `clk` in `lab1.sv`) of 1 ns (i.e. we require the design to operate at 1 GHz). Quartus interprets very fast timing constraints of this type as "go as fast as possible"; they do not cause bad behaviour in the compiler even though 1 GHz is beyond what an Arria 10 FPGA can achieve.

Looking at the compiler messages, you will see:

```
1    Info: Analyzing Slow 900mV 100C Model
2    Critical Warning (332148): Timing requirements not met
3    Info (332146): Worst-case setup slack is -20.415
```

This indicates that the design did not meet its timing constraint (of a 1 ns clock period) by 20.415 ns. Consequently the fastest cycle time for the design is 1 + 20.415 ns = 21.415 ns, or 46.7 MHz.

In recent semiconductor processes (65 nm and below), sometimes the slowest speed occurs at low temperatures rather than high temperatures. Carrier mobility increases with reduced T (helping speed), but Vt also increases as T drops (reducing speed), so it is now necessary to check timing at both "temperature corners". Later in the compiler messages you should see:

```
1    Info: Analyzing Slow 900mV 0C Model
2    Info (332146): Worst-case setup slack is -21.311
```

In this case, the slack at 0 C (-21.311 ns) is actually more negative than that at 100 C (-20.415 ns). Therefore, the maximum operating frequency is 44.8 MHz based on the 0 C corner. You should check the timing report for any other frequency limitations: very high speed designs can sometimes hit limits (restricted Fmax) on the frequency of some FPGA building blocks, but our simple design is far from those limits.

To see how the design was implemented, select `Tools > Netlist Viewers > RTL Viewer`. This shows how the design is interpreted by the first stages of synthesis, but does not show the final optimized implementation.

The design looks as expected, with 5 multipliers and 5 adders between the x and y register banks.
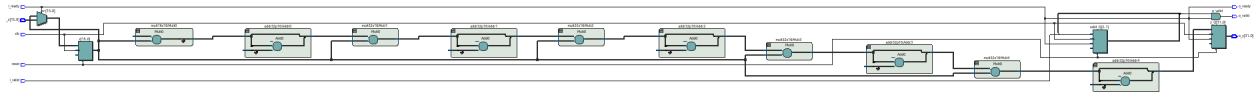
Figure 4: RTL view of the netlist.

You can click on the '+' sign on each box to view its internals; notice that one input of each adder is a constant. The `o_valid` signal is a shift-registered version of the `i_valid` signal ANDed with `i_ready`, which also makes sense and matches the `lab1.sv` HDL.

Finally, the timing constraint we are using in this lab analyzes the delay only between register to register paths in your design. It does not check the speed of any transfers from (virtual) I/Os to your design registers, or from your design to I/Os. You should therefore register your inputs and your outputs for two reasons: it is a good design practice, and it ensures the timing constraints are analyzing all the important paths in your design. If you ever find that you have an unreasonably fast design (e.g. > 800 MHz), check that you have not left input or output registers out of your design, as you should not be able to run a design on an Arria 10 device that fast if it actually has paths between registers.

5.4 Estimating Resource Usage

To see how much of the device is used to implement the design, select `Processing > Compilation Report`. Click on `Fitter > Resource Section > Resource Usage by Entity`. Each line in this report shows the resources used by a module of the design, including any sub-modules that it includes. The top line for example, summarizes the total resources used by the design. On the other hand, the line named `addr32p16:Addr1` summarizes the resources used only within one of the adders for example. Note that the values in brackets give the resources used in sub-modules that are instantiated within a certain module (hierarchy) in the design, while the values outside the brackets give the total resources in this level of hierarchy of the design and all modules contained within it.

Looking at the top line (blue circles) in the Resource Utilization by Entity report shown in Fig. 5, you can see that 43 look-up tables (called ALUTs by Intel), 34 registers and 9 DSP blocks are required to implement the design. Intuitively, we would expect to utilize 50 registers (16 for `i_x` + 32 for `o_y` + 1 for `i_valid` + 1 for `o_valid`). However, the report shows that only 34 registers are needed because Quartus is smart enough to pack the the 16 registers for `i_x` in the registers inside the DSP block. This can be seen under the `Fitter > Netlist Optimizations` section of the compilation report shown in Fig. 6. Notice that the register `x[0]`, for example, is duplicated and packed into DSP internal registers (as indicated by the destination node) to optimize timing (as indicated by the reason).

Each DSP block in Arria 10 can implement either two 18×18 multipliers or one 27×27 multiplier (in addition to other modes of operation as multiply-accumulate, pre-addition, etc.). Looking at each of the 5 entries for the 5 multipliers in Fig. 5, you can see that the first 16×16 multiplier (`Mult0`) uses a single DSP block (it technically uses half the DSP block). On the other hand, all the other 16×32 multipliers are implemented using 2 DSP blocks each as highlighted by the red circle. For the adders, `Addr0` is also mapped to the adder/accumulator inside the DSP block used to implement `Mult0` (i.e. the first DSP block is used in the 18×18 plus 36 mode, implementing $a_5x + a_4$ without the need for any soft logic resources). Notice also that the number of ALUTs used to implement the 4 adders decreases from 20 ALUTs for `Addr1` down to 7 ALUTs for `Addr4`. This is

due to the gradual increase in the number of trailing zeros in the coefficient values (a_1 to a_4) which renders a wider portion of the adders into just wires.

Fitter Resource Utilization by Entity

🔍 <<Filter>>

	Compilation Hierarchy Node ▼	ALMs needed [=A-B+C]	Combinational ALUTs	Dedicated Logic Registers	I/O Registers	Block Memory Bits	M20Ks	DSP Blocks
1	⊟ |lab1	48.0 (28.7)	43 (1)	34 (34)	0 (0)	0	0	9
1	|addr32p16:Addr1|	10.0 (10.0)	20 (20)	0 (0)	0 (0)	0	0	0
2	|addr32p16:Addr2|	4.0 (4.0)	8 (8)	0 (0)	0 (0)	0	0	0
3	|addr32p16:Addr3|	1.8 (1.8)	7 (7)	0 (0)	0 (0)	0	0	0
4	|addr32p16:Addr4|	1.2 (1.2)	7 (7)	0 (0)	0 (0)	0	0	0
5	|mult16x16:Mult0|	0.0 (0.0)	0 (0)	0 (0)	0 (0)	0	0	1
6	|mult32x16:Mult1|	0.0 (0.0)	0 (0)	0 (0)	0 (0)	0	0	2
7	|mult32x16:Mult2|	0.0 (0.0)	0 (0)	0 (0)	0 (0)	0	0	2
8	|mult32x16:Mult3|	0.0 (0.0)	0 (0)	0 (0)	0 (0)	0	0	2
9	|mult32x16:Mult4|	0.0 (0.0)	0 (0)	0 (0)	0 (0)	0	0	2

Figure 5: Resource utilization by entity.

Fitter Netlist Optimizations

🔍 <<Filter>>

	Node	Action	Operation	Reason	Node Port	Node Port Name	Destination Node	Destination Port
1	x[0]	Packed Register	Register Packing	Timing optimization	Q		mult16x16:Mult0|Mult0~mac	AY
2	x[0]	Duplicated	Register Packing	Timing optimization	Q		x[0]~_Duplicate_1	Q
3	x[0]~_Duplicate_1	Packed Register	Register Packing	Timing optimization	Q		mult32x16:Mult1|Mult0~12	AX
4	x[0]~_Duplicate_1	Duplicated	Register Packing	Timing optimization	Q		x[0]~_Duplicate_2	Q
5	x[0]~_Duplicate_2	Packed Register	Register Packing	Timing optimization	Q		mult32x16:Mult1|Mult0~mult_hlmac	AY
6	x[0]~_Duplicate_2	Duplicated	Register Packing	Timing optimization	Q		x[0]~_Duplicate_3	Q
7	x[0]~_Duplicate_3	Packed Register	Register Packing	Timing optimization	Q		mult32x16:Mult2|Mult0~12	AX
8	x[0]~_Duplicate_3	Duplicated	Register Packing	Timing optimization	Q		x[0]~_Duplicate_4	Q
9	x[0]~_Duplicate_4	Packed Register	Register Packing	Timing optimization	Q		mult32x16:Mult2|Mult0~mult_hlmac	AY
10	x[0]~_Duplicate_4	Duplicated	Register Packing	Timing optimization	Q		x[0]~_Duplicate_5	Q
11	x[0]~_Duplicate_5	Packed Register	Register Packing	Timing optimization	Q		mult32x16:Mult3|Mult0~12	AX
12	x[0]~_Duplicate_5	Duplicated	Register Packing	Timing optimization	Q		x[0]~_Duplicate_6	Q
13	x[0]~_Duplicate_6	Packed Register	Register Packing	Timing optimization	Q		mult32x16:Mult3|Mult0~mult_hlmac	AY
14	x[0]~_Duplicate_6	Duplicated	Register Packing	Timing optimization	Q		x[0]~_Duplicate_7	Q
15	x[0]~_Duplicate_7	Packed Register	Register Packing	Timing optimization	Q		mult32x16:Mult4|Mult0~12	AX
16	x[0]~_Duplicate_7	Duplicated	Register Packing	Timing optimization	Q		x[0]~_Duplicate_8	Q
17	x[0]~_Duplicate_8	Packed Register	Register Packing	Timing optimization	Q		mult32x16:Mult4|Mult0~mult_hlmac	AY

Figure 6: Netlist optimizations performed by Quartus. The registers for `i_x` are duplicated and mapped to the internal DSP block input registers (AY and AX).

Next, take a look at the Resource Usage Summary section of the compilation report shown in Fig. 7. This section gives a breakdown of how the various FPGA resources are used by your design. The logic utilization line gives the best estimate of how full the FPGA is in terms of LUTs and registers (48 ALMs). ALMs stands for Adaptive Logic Modules which is the Intel naming for the logic element which in Arria 10 contains a 6-input fracturable LUT (ALUT), two bits of arithmetic and four registers. You can also see that 27 ALMs are used to implement virtual pins. When you report the logic utilization of your designs you should subtract the number of ALMs used by virtual pins from the logic utilization value given in this resource usage summary. In this case, that adjusted logic utilization is $48 - 27 = 21$ ALMs.

5.5   Analyzing Design Speed

To see what is limiting the speed of your circuit, select `Tools > Timing Analyzer`. Double-click on the `"Report Timing"` item in the tasks window. The Report Timing window will come up. Select the `"from clock"` and `"to clock"` to both be `clk` (which is the only clock signal in `lab1.sv`). The Tcl command corresponding to what you have selected in the menu is shown at the bottom of the window as shown in Fig. 8, in case you would prefer to directly type the command in the future.

Fitter Resource Usage Summary

🔍 <<Filter>>

		Resource	Usage	%
1		Logic utilization (ALMs needed / total ALMs on device)	48 / 427,200	< 1 %
2		⊟ ALMs needed [=A-B+C]	48	
1		⊟ [A] ALMs used in final placement [=a+b+c+d]	28 / 427,200	< 1 %
1		[a] ALMs used for LUT logic and registers	8	
2		[b] ALMs used for LUT logic	14	
3		[c] ALMs used for registers	6	
4		[d] ALMs used for memory (up to half of total ALMs)	0	
2		[B] Estimate of ALMs recoverable by dense packing	7 / 427,200	< 1 %
3		⊟ [C] Estimate of ALMs unavailable [=a+b+c+d]	27 / 427,200	< 1 %
1		[a] Due to location constrained logic	0	
2		[b] Due to LAB-wide signal conflicts	0	
3		[c] Due to LAB input limits	0	
4		[d] Due to virtual I/Os	27	
3		...		
4		Difficulty packing design	Low	
5		...		
6		⊟ Total LABs: partially or completely used	5 / 42,720	< 1 %
1		-- Logic LABs	5	
2		-- Memory LABs (up to half of total LABs)	0	
7		...		
8		⊟ Combinational ALUT usage for logic	43	
1		-- 7 input functions	0	
2		-- 6 input functions	0	
3		-- 5 input functions	0	
4		-- 4 input functions	0	
5		-- <=3 input functions	43	
9		...		
10		⊟ Dedicated logic registers	34	
1		⊟ -- By type:		
1		-- Primary logic registers	27 / 854,400	< 1 %
2		-- Secondary logic registers	7 / 854,400	< 1 %

Figure 7: Resource usage summary report.

Click the `"Report Timing"` button and the clock path to the source register, the data path from the source register to the destination register, and the clock path to the destination register, will all be listed.

> ⚠ Since we know from the compiler messages that the circuit speed is restricted by the worst-case slack in the 0 C corner, make sure that you are analyzing the correct temperature corner by choosing the correct model in the top-left `Set Operating Conditions` panel. If the report is not yet generated, right-click `Slow 900mv 0C Model` in the `Report` panel on the left-hand side of the window and then click `Regenerate`.

The graphical waveform display shown in Fig. 9 gives an overview of the timing path that is limiting the speed of the circuit. In this case, the difference between the clock delays to the source and destination register is 4.586-4.249 = 0.34 ns, which represents the effect of the clock skew on the speed of the circuit. The data path delay is fairly long, 22.725 ns.

Looking at the Data Arrival Path listing you can see that the first few lines give the "clock path" to the register that begins the critical (speed-limiting) path of the circuit. The lines in the
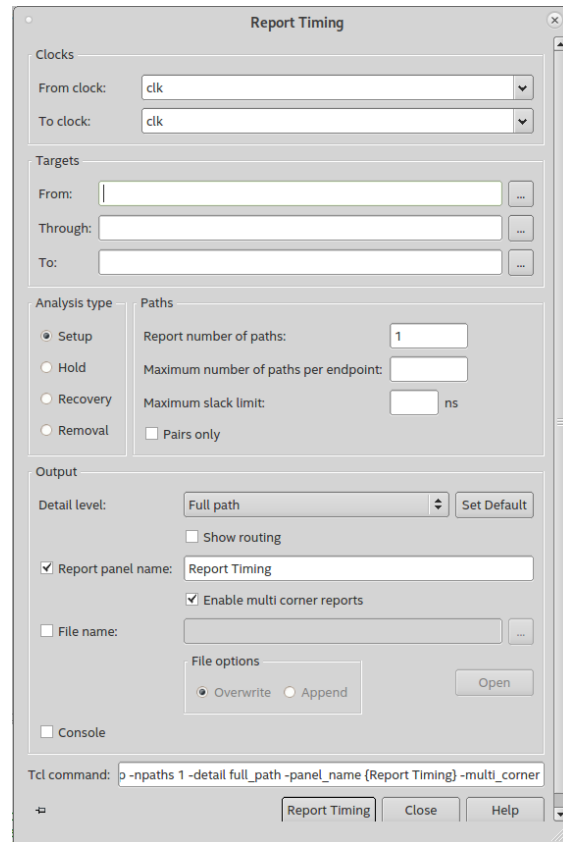
Report Timing

Clocks

From clock: clk

To clock: clk

Targets

From:

Through:

To:

Analysis type

● Setup
○ Hold
○ Recovery
○ Removal

Paths

Report number of paths: 1

Maximum number of paths per endpoint:

Maximum slack limit: ns

☐ Pairs only

Output

Detail level: Full path　　Set Default

☐ Show routing

☑ Report panel name: Report Timing

☑ Enable multi corner reports

☐ File name:

File options
● Overwrite　○ Append　　Open

☐ Console

Tcl command: o -npaths 1 -detail full_path -panel_name {Report Timing} -multi_corner

Report Timing　　Close　　Help

Figure 8: Report timing window.

Slow 900mV 0C Model

Command Info | Summary of Paths

	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	-21.311	mult16x16:Mult0|Mult0~mac~reg0	y_Q[30]	clk	clk	1.000	-0.028	22.725

Path #1: Setup slack is -21.311 (VIOLATED)

Path Summary | Statistics | Data Path | Waveform | Extra Fitter Information

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Location	HS/LP	Element
8	4.586	3.284	RR	IC	17	MPDSP_X128_Y122_N0	High Speed	Mult0|Mult0~mac|clk[0]
9	4.586	0.000	RR	CELL	96	MPDSP_X128_Y122_N0	High Speed	mult16x16:Mult0|Mult0~mac~reg0
3	⊟ 27.311	22.725						data path
1	7.434	2.848	RR	uTco	1	MPDSP_X128_Y122_N0		Mult0|Mult0~mac|resulta[11]
2	7.737	0.303	RR	IC	49	MPDSP_X128_Y121_N0	High Speed	Mult1|Mult0~12|ay[8]
3	9.976	2.239	RF	CELL	1	MPDSP_X128_Y121_N0	High Speed	Mult1|Mult0~12|resulta[21]
4	10.488	0.512	FF	IC	87	MPDSP_X128_Y120_N0	High Speed	Mult1|Mult0~mult_hlmac|by[3]
5	12.086	1.598	FF	CELL	1	MPDSP_X128_Y120_N0	High Speed	Mult1|Mult0~mult_hlmac|resulta[1
6	12.507	0.421	FF	IC	6	MLABCELL_X129_Y119_N6	High Speed	Addr1|Add0~9|dataf
7	12.805	0.298	FR	CELL	1	MLABCELL_X129_Y119_N9	High Speed	Addr1|Add0~13|cout
8	12.805	0.000	RR	CELL	3	MLABCELL_X129_Y119_N12	High Speed	Addr1|Add0~17|cin
9	12.985	0.180	RF	CELL	1	MLABCELL_X129_Y119_N12	High Speed	Addr1|Add0~17|sumout
10	12.990	0.005	FF	IC	1	MLABCELL_X129_Y119_N12	High Speed	addr32p16:Addr1|Add0~17~la_ml
11	13.259	0.269	FF	IC	39	MPDSP_X128_Y119_N0	High Speed	Mult2|Mult0~12|ay[16]
12	15.351	2.092	FF	CELL	1	MPDSP_X128_Y119_N0	High Speed	Mult2|Mult0~12|resulta[26]
13	15.880	0.529	FF	IC	77	MPDSP_X128_Y118_N0	High Speed	Mult2|Mult0~mult_hlmac|by[8]
14	17.352	1.472	FR	CELL	1	MPDSP_X128_Y118_N0	High Speed	Mult2|Mult0~mult_hlmac|resulta[1
15	17.662	0.310	RR	IC	39	MPDSP_X128_Y117_N0	High Speed	Mult3|Mult0~12|ay[16]
16	19.806	2.144	FF	CELL	1	MPDSP_X128_Y117_N0	High Speed	Mult3|Mult0~12|resulta[28]
17	20.366	0.560	FF	IC	74	MPDSP_X128_Y116_N0	High Speed	Mult3|Mult0~mult_hlmac|by[10]

Data Required Path

	Total	Incr	RF	Type	Fanout	Location	HS/LP	Element
1	1.000	1.000						latch edge time
2	⊟ 5.558	4.558						clock path
1	1.000	0.000						source latency
2	1.000	0.000			1	PIN_G37		clk

Path #1: Setup slack is -21.311 (VIOLATED)

Path Summary | Statistics | Data Path | Waveform | Extra Fitter Information

Launch Clock
Setup Relationship 1.0 ns
Latch Clock
Data Arrival
Clock Delay　4.586 ns
Data Delay　22.725 ns
Slack　-21.311 ns
Data Required
Clock Delay　4.249 ns
Clock Pessimism　0.309 ns
Clock Uncertainty　-0.00 ns
uTsu　-0.472 ns
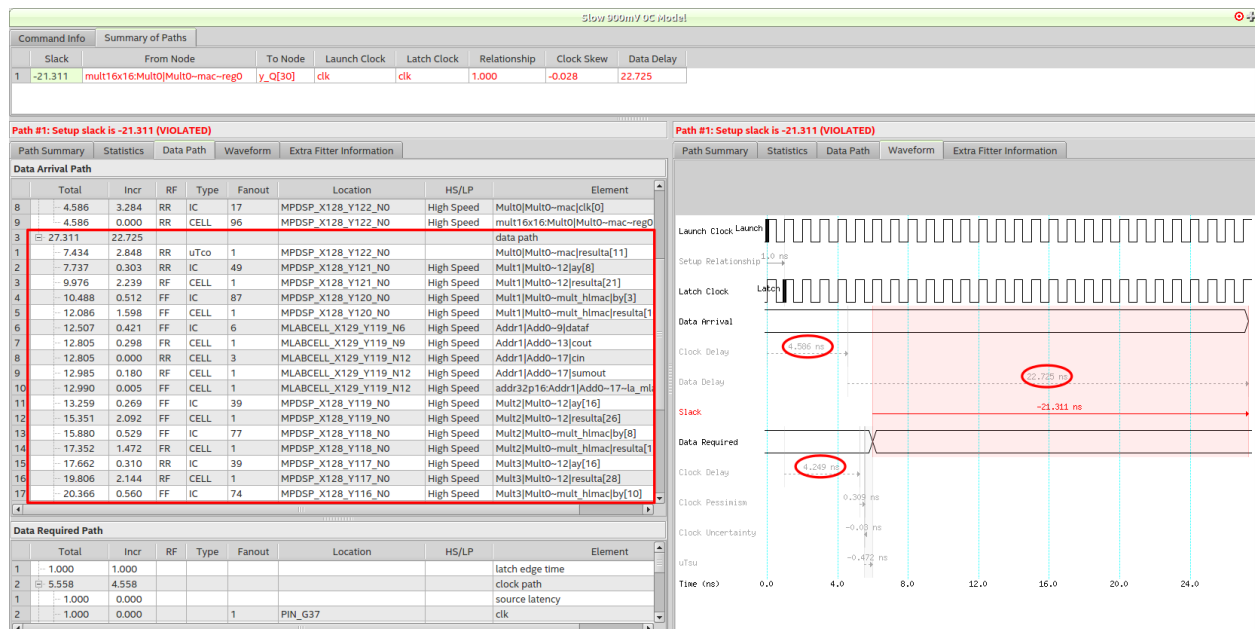Time (ns)　0.0　4.0　8.0　12.0　16.0　20.0　24.0

Figure 9: Analyzing the critical path delay of the circuit.

red rectangle in Fig. 9 list the more interesting "data path" that shows the computation that is
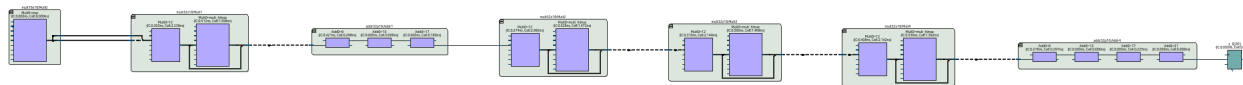
Figure 10: Viewing the critical path in the Technology Map Viewer.

limiting the circuit speed. By carefully examining the types of the blocks and their names, one can see that the critical path:

- passes through the first DSP block which implements the first multiplier and adder (`Mult0~mac`),
- followed by another multiplier (`Mult1`) split accross two DSP blocks (indicated by locations `MPDSP_X128_Y121` and `MPDSP_X128_Y120`),
- followed by an adder (`Addr1`) created from logic elements (LUTs and carry chains),
- followed by another multiplier (`Mult2`) split accross two DSP blocks,
- followed by another multiplier (`Mult3`) split accross two DSP blocks,
- followed by another multiplier (`Mult4`) split accross two DSP blocks,
- followed by another adder (`Addr4`) created from logic elements,
- and ending in the `y_Q[30]` register.

Notice that the two adders, `Addr2` and `Addr3`, are not on the critical path, as they feed only the second DSP block in `Mult2` and `Mult3` since the least significant bits of the corresponding coefficient values are all zeros (i.e. the outputs of the adder are of higher significance and are only used as inputs to the second DSP block). Lines with a type of IC represent interconnect (programmable routing) delays, while lines with a type of Cell represent delays within blocks (LUTs, multipliers etc.). Examining the X and Y coordinates in the location column, one sees that generally the critical path is well localized, which will lead to small routing delays. Right click on the Data Arrival Path window, and select `Locate Path > Locate in Technology Map Viewer`. You can now visually see the critical path elements, and navigate them. Blue elements indicate blocks on the chip, while green boxes are the hierarchy boundaries from the HDL description of your design. Hovering your mouse over a block will bring up a tooltip that shows the block's name, and double clicking on a block or signal will zoom into that block, or move the view to show you how that signal was generated. Fig. 10 shows the entire critical path of the design, showing:

- the first DSP block implementing `Mult0` and `Addr0`
- feeding `Mult1` (second green block with two blue DSP blocks inside it), then
- a sequence of logic elements (blue boxes) that implement `Addr1`, followed by
- the four DSP blocks that implement `Mult2` and `Mult3`, followed by
- another sequence of logic elements implementing `Addr4`, and then ending on
- the `y_Q[30]` register.

You can also locate critical paths in the chip planner, if you wish to see exactly where the circuit elements are placed and how the routing between them is implemented. Right click on the Data Arrival Path, and select `Locate Path > Locate in Chip Planner`. As shown in Fig. 11, it is again clear that the critical path has been localized well, as it spans a very small part of the chip.

5.6 Simulating for Correctness

We are now ready to simulate the design to see if it is behaving correctly. If you are using the UG machines, start ModelSim by typing the following command in the terminal:
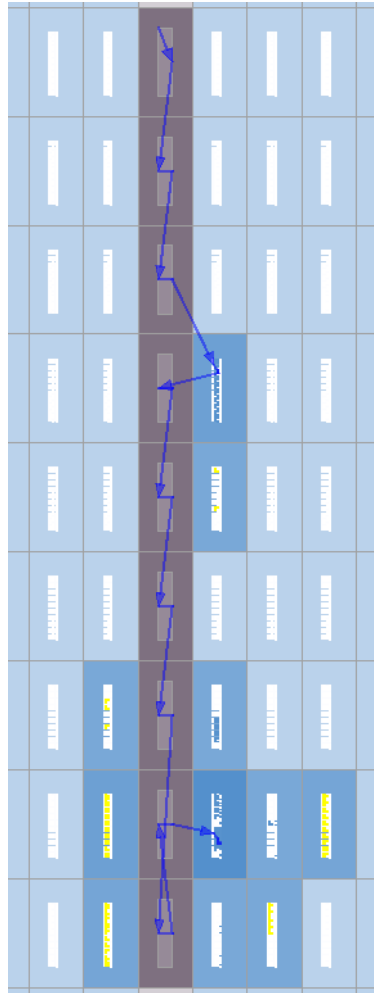
```
modelsim18.start &
```

Figure 11: Viewing the critical path in the Chip Planner. DSP blocks are in grey and logic blocks are in blue.

A tutorial on how to use the ModelSim GUI to compile and simulate a design is downloadable from this link as the "Use ModelSim with Testbenches" tutorial.

Alternatively, you can simulate your design by typing the commands at the ModelSim command prompt at the bottom of the ModelSim window as follows. The comments in green beside each command explains what it does (you shouldn't type that in the command prompt).

```
1  cd <path to lab1.sv> # Navigate to the directory containing your design's Verilog files
2  vlib work # Create a library called work in which your results will be placed
3  vlog lab1.sv # Compile lab1.sv
4  vlog lab1_tb.v # Compile the testbench
5  vsim work.lab1_tb # Starts the simulator, with a top-level module of the lab1 testbench
6  view wave # Open the waveform window
```

We want to monitor all the signals in `lab1.sv`. To add these signals to the waveform viewer, double-click the "dut" line in the sim window to open the HDL code of your design. Highlight the module inputs and outputs in `lab1.sv` and right click on them, then select `Add > To Wave`. You could also choose to highlight any intermediate signals from `lab1.sv` and add them to the waveform window as well, if you want to see more detail. After that, type the following command in the ModelSim command prompt:
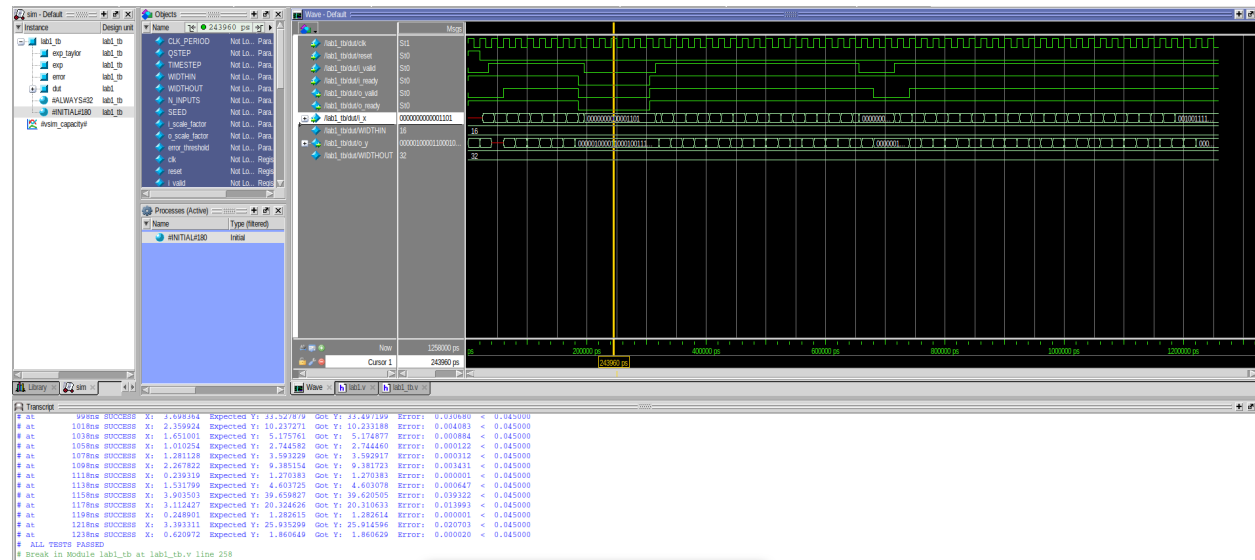
Figure 12: The ModelSim simulation window.

```
1  run -all # Run simulation to completion which is OK because the testbench applies a fixed number of
       inputs and then issues a $stop command
```

Select `View > Wave` if the waveform window isn't visible. Examine the waveforms in the waveform window; you can right click and select Zoom Full to zoom the view to show the entire time of the waveforms. You can right click on any signal and change its radix (display mode) if needed. Remember that the `i_x` and `o_y` are represented in unsigned fixed-point Q2.14 and Q7.25, respectively. Notice that the `o_valid` signal goes high at the same time the first valid `o_y` output comes out. The `lab1_tb.v` file also automatically checks the circuit outputs against the expected outputs (exact e^x), and prints messages indicating which inputs pass and fail as shown in Fig. 12. To account for the Taylor polynomial approximation error, an error tolerance of 0.045 is allowed, within which the result is accepted as correct. Note that the testbench provided for this assignment uses a 42 MHz clock (a period of 24ns). The simulation we did above is called a *functional* or *RTL* simulation, and it does not model the delay of your FPGA circuitry, so you should use it to check that your design is correct, but never to try to use it to see how fast your design can run. The testbench always changes data input to your design well away from the rising edge of the clock to avoid any ambiguity about whether the data is changing just before or just after the rising edge of a clock.

When you run the ModelSim simulation, two files (`testcase.txt` and `exp.txt`) will be generated in the project directory. You can use the provided `graph.gnu` script to plot your hardware output compared to the exact exponential function by running the following commands in the Unix terminal.

```
1  gnuplot graph.gnu
2  gimp graph.png
```

GNU plot is a graphing utility that is already installed on the UG machines. In case you are using your own machine, you will have to download and install GNU plot to perform this step. After running the `graph.gnu` script, a `graph.png` plot (similar to that shown in Fig. 13) will be generated in the project directory and can be viewed with image viewers like gimp. Feel free to implement more terms of the exponential function Taylor expansion in `lab1.sv` and observe the effect on the
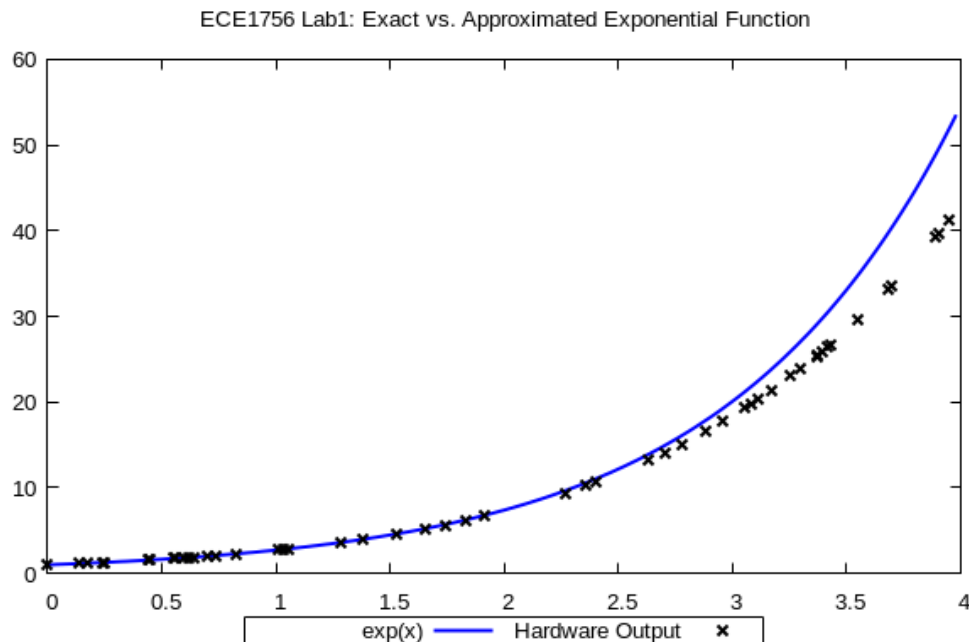
Figure 13: Plot comparing the exact exponential function vs. hardware output of the example circuit. This can be generated using the `graph.gnu` script provided in this assignment.

difference between the exact and hardware output traces in this plot.

5.7   Estimating Power Dissipation

Next we will estimate the power of this design. Power estimation for FPGAs is tricky, as the power dissipated strongly depends on the design, both in terms of the device resources used and how often the signals in the design toggle (change state). We will use a method that is quite accurate, but involves a moderate amount of work.

The most accurate form of power analysis would simulate a design in an analog circuit simulator, such as SPICE, and measure the current drawn from the power supply. Power would then be calculated as $I_{avg} * V_{supply}$. This method would be impossibly slow however for large circuits. Consequently, the best practical method to estimate power using a digital simulator (e.g. ModelSim) to determine how many times each signal in a design toggles (changes state) per second. This data is then combined with models built into Quartus that give the dynamic power consumed for each toggle of each type of block (DSP multiplier, LUT, etc.) and for each type of routing wire used to interconnect them . Quartus also calculates the static (leakage) power for the device, which is strongly temperature dependent, and grows exponentially as the junction temperature increases.

To estimate power, we will first obtain the toggle rate for each signal during the design operation from ModelSim. We are going to use ModelSim to do a more detailed simulation than in the previous section: instead of just simulating the behaviour of the RTL, we will simulate all the LUTs, FF, and multipliers making up our design implementation, along with the delays between them. This *gate-level* or *timing* simulation is slower than behavioural or RTL simulation, but is the only way to get accurate information on how often each signal in our design (including all the internal signals between LUTs etc.) toggles.

Open the `Assignments > Settings > EDA Tool Settings > Simulation` dialog. Note that if you use the Quartus project file that is included with the lab1 files, most of the settings below will already be set, and you need only verify that they are properly set.

1. Select ModelSim-Altera as the tool, Verilog HDL as the output language, and check the "Generate Verilog Change Dump (VCD) file script" option.
2. Enter "dut" as the design instance name, as this is the top-level instance name used in the testbench.
3. The "enable glitch filtering" option should be checked to increase power estimation accuracy, as described later in this section.
4. Select the "Compile test bench" option, and click on the "Test Benches" button. Select `lab1_tb.v` and click the "Edit" button. Ensure the various fields match the dialog in Fig. 14.

Select `Processing > Start > Start EDA Netlist Writer`. This will write out a gate-level (really device primitive level like LUTs and FFs) netlist, with annotated timing information, of your design. It will also write out appropriate scripts to run ModelSim in a mode that will produce the most accurate toggle rates. ModelSim will be run with "glitch filtering" enabled. Glitches are transitions of combinational logic that occur before the output of the logic settles down to its final value for a clock cycle. For example, consider a 2-input XOR gate that initially has both inputs at 0, and then both go to 1 during a certain clock cycle. The xor output will start and end the cycle with an output of 0. However, if the two inputs transition at different times, the xor output will temporarily transition to 1 in the middle of the cycle. This is called a glitch, and it dissipates power. ModelSim can over-predict glitching because they model gates as having infinitely fast rise and fall times, while physical gates transition more slowly. Glitch filtering is a technique that modifies the delay model in ModelSim to more closely approximate the real transition time of gates.
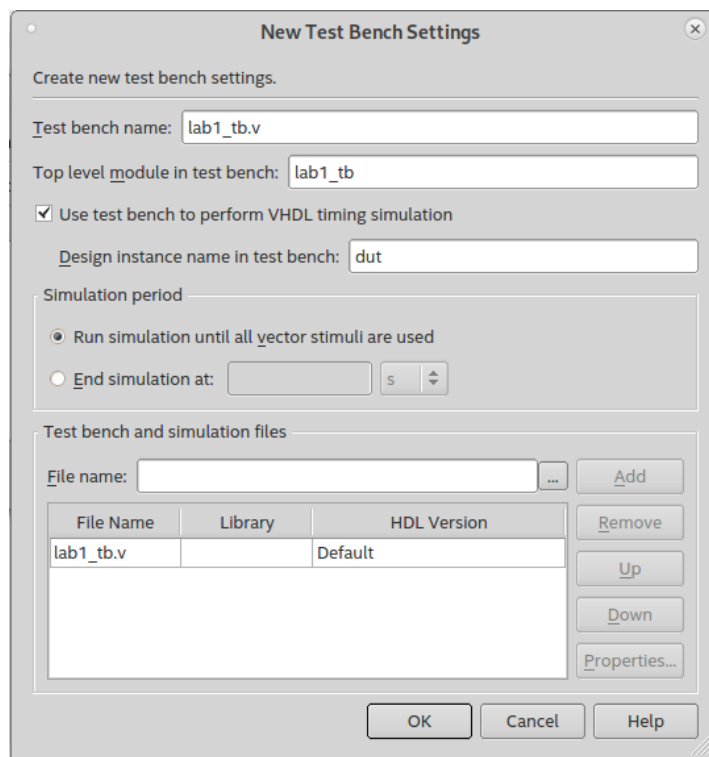
Figure 14: Testbench settings dialog.

Select `Tools > Run Simulation Tool > Gate Level Simulation`. ModelSim will automatically start, compile the gate-level netlist of the design and testbench sent to it by Quartus, and output a `lab1.vcd` file that lists every signal transition during the simulation [2].

Look at the ModelSim messages to confirm the design passed the testbench simulation. Close ModelSim, to ensure the VCD file (which lists every signal transition during the simulation) is properly written out and closed. From Quartus, select `Processing > Power Analyzer Tool`. Click on "Use input files to initialize toggle rates and static probabilities for during power analysis" followed by "Add Power Input Files". Click the Add button and add the file `simulation/modelsim/lab1.vcd` as shown in Fig. 15. The Power Analyzer settings should now be similar to Fig. 16.
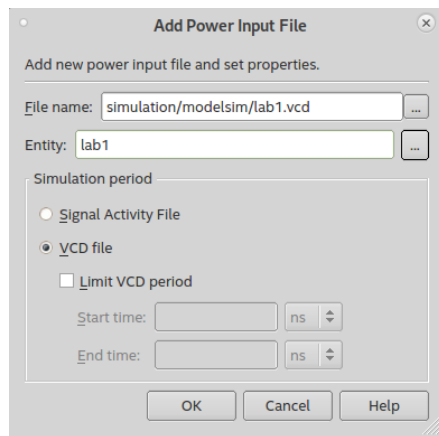
Figure 15: Add power input file (`lab1.vcd`) dialog.
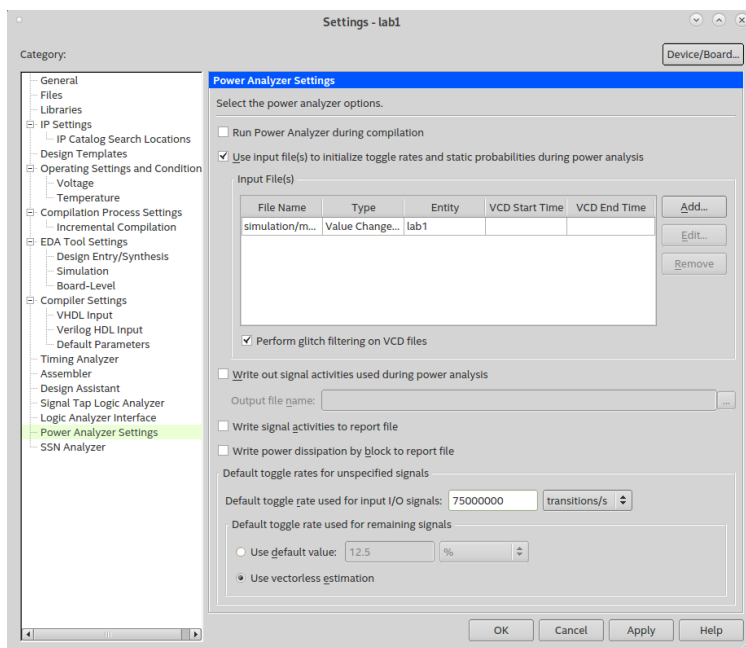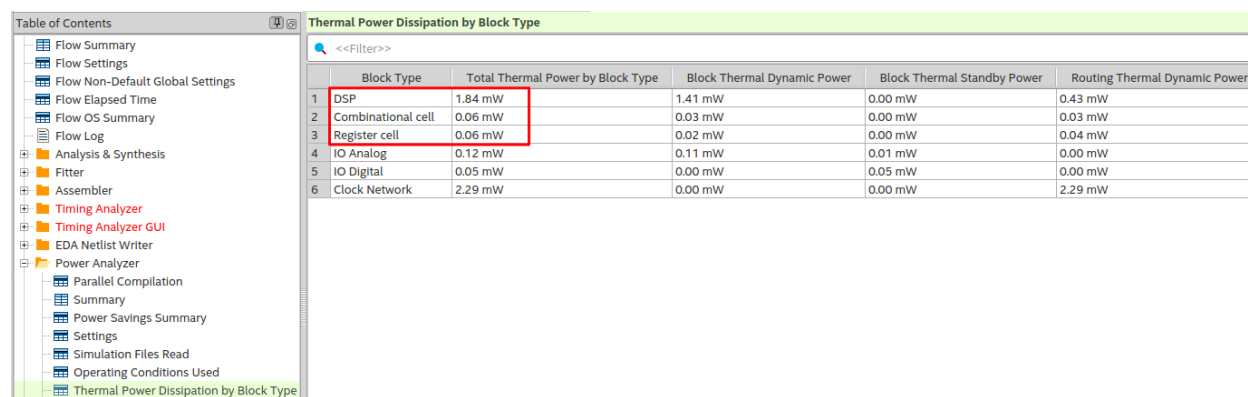
Figure 16: Power Analyzer Settings.

Click `OK`, and then `Start` in the Power Analyzer Tool. Note that to run the Power Analyzer, you

---

[2] If Quartus is unable to launch ModelSim via NativeLink, check that you have the correct path to the modelsim executable specified under `Tools > Options > EDA Tool Options > ModelSim-Altera`.

must have compiled the design, including running the assembler to create a programming file.[3] Click on the Power Analyzer Report Summary tab. Since our design is very small and the testbench is running it at only 42 MHz[4] most of the power is leakage (static power) and I/O power. We wish to estimate how much power the design would take if we scaled it up to fill the device and ran it at the maximum operating frequency. To compute that, we need to look at the "Thermal Power Dissipation by Block Type" tab. This lists how much dynamic power each type of block in the design is dissipating. The "Clock Network" block gives the clock power; assume this stays constant as you increase the design size. The DSP block, Combinational cell, and Register cell power will all increase linearly as the clock frequency increases, and as the design size increases. The total power for these block types, including the routing between them is $1.84 + 0.06 + 0.06 = 1.96$ mW @ 42 MHz as shown in Fig. 17, for one instance of your computation engine. From the summary report, one can also see that the device static power is 1704.84 mW, and the I/O power is 0.17 mW as shown in Fig. 18.

Figure 17: Power consumption by block type.

Given the resource utilization estimates you have obtained from Quartus, estimate the maximum number of exponential function throughput (i.e. computations per second) that could be sustained by the Arria 10 device you are using, if you were to replicate your circuit as much as possible until you ran out of some device resource. Assume the maximum clock frequency of your circuit does not change as you fill the device. Estimate the throughput per Watt you could attain if you filled the whole device. You can assume that static, I/O and clock power stay the same as you fill the device (this is an approximation, but doesn't add much error), but that the power for the DSP and logic cell blocks increases linearly with the number of circuit replicas. You can also assume linear scaling of dynamic power with clock frequency.

---

[3]If you are working on your home computer, running the assembler requires compiling while being connected to the license server (not just using the 30 day evaluation license).

[4]You might be tempted to change the testbench to run the design at the maximum frequency at which Quartus reported it could run, instead of 42 MHz. This is trickier than it seems; the testbench we are using is changing data away from rising clock edges, and hence to properly check if we can communicate with the testbench at a higher frequency with the design we implemented in the FPGA, we would have to write more timing constraints in the `lab1.sdc` file to more tightly constrain the delay to and from the virtual I/Os (which represent the connection to the testbench). This is really not worth the effort, so instead we will simulate the design at 42 MHz and scale the dynamic power results to represent what would happen at a higher frequency.

Power Analyzer Summary

<<Filter>>

Power Analyzer Status	Successful - Thu Sep 17 00:12:04 2020
Quartus Prime Version	20.1.0 Build 711 06/05/2020 SJ Standard Edition
Revision Name	lab1
Top-level Entity Name	lab1
Family	Arria 10
Device	10AX115N2F45I1SG
Power Models	Final
Total Thermal Power Dissipation	1709.27 mW
Transceiver Standby Thermal Power Dissipation	0.00 mW
Transceiver Dynamic Thermal Power Dissipation	0.00 mW
I/O Standby Thermal Power Dissipation	0.06 mW
I/O Dynamic Thermal Power Dissipation	0.11 mW
Core Dynamic Thermal Power Dissipation	4.25 mW
Device Static Thermal Power Dissipation	1704.84 mW
Power Estimation Confidence	Medium: user provided moderately complete toggle rate data

Figure 18: Power consumption summary.

> Determine the number of copies of the baseline circuit that fit on the Arria 10 device you are using. Then calculate the throughput, power consumption of the full device at the maximum operating frequency, and the throughput/Watt of the full FPGA. Use these results to fill the first column of Table 1 in your report.

6 Develop a Pipelined Implementation

Copy your Quartus project and HDL codes to a new directory (to maintain the same Quartus project settings). Modify the baseline implementation you used in Section 5 so that it is pipelined for maximum throughput, and confirm that your design works by simulating it with ModelSim and the provided testbench, which will work for a pipelined design so long as you assert the `o_ready` and `o_valid` signals at the proper time.

> Implement a pipelined version of the same example circuit and verify its functionality using the same provided testbench. Optimize the deisgn to maximize clock frequency and throughput. Collect all the results as you learned from the tutorial in Section 5 and fill the second column of Table 1 in your report.

7 Develop a Shared Operator Implementation

In another Quartus project, implement a shared operator version of the same example circuit in which you have only one multiplier and one adder that are re-used over multiple clock cycles to compute the exponential function. You will need some extra multiplexers to select the inputs to the multiplier and adder, some temporary storage, and a finite-state machine to control the computation. You should try to keep your design efficient (small) by minimizing the amount of temporary storage and multiplexing you need, but you will certainly need some of each.

> Implement a shared operator version of the same example circuit and verify its functionality using the same provided testbench. Optimize the throughput of the design, which will depend on your design's frequency, how many cycles it takes to compute a complete output, amd how many copies you can fit in the FPGA. Collect all the results as you learned from the tutorial in Section 5 and fill the third column of Table 1 in your report.

8 How You Will Be Graded

Your grade out of 13 will mainly depend on:

1. **The correctness and performance of your solution (~50%):** You should try to optimize your hardware designs for throughput and, as a secondary objective, for area. A portion of your grade will depend on the quality of results and the effort you spend on optimization.

2. **The correctness, completeness and clarity of your report (~40%):** Your report should include enough details to explain your design, implementation, and functional verification process. Use block diagrams, screenshots, and/or graphs whenever needed to explain your solutions and optimizations. Avoid using your report as a screenshot dump that might not be very readable or missing comments/discussions in your text.

3. **Coding style (~10%):** You should write readable and clear HDL code. Include sufficient comments, use meaningful signal/variable names and indent appropriately. Use named values for states (e.g. use `enum`) instead of testing binary constants throughout the code. Avoid repetitive, duplicated code by creating and instantiating reusable modules, or by using `generate` or `for` loops as appropriate.

Appendix A: Using the CAD Tools

There are two ways you can use the CAD tools (Quartus & ModelSim) that are needed to complete this, as well as the next, assignment. You can install the tools on your machine or use the pre-installed tools on the UG machines. We highly recommend that you use the tools on the UG machines since we already tested this setup and can guarantee it works smoothly (without any licensing or configuration issues). If it is more convenient for you to install them on your own machine, we will try our best to help you with any setup/licensing problems. However, it is your responsibility to get all parts of the assignment done.

Appendix A.1: Remote Access to the UG Machines

The UG machines are part of the EECG system (not ECF). You can access them in person in the GB 251, GB 243, SF 2102 and SF 4102 rooms. Alternatively, you can access them remotely using ssh (for command line access) and vnc (for graphical access). Your username will be your utorid, and your initial password (if you have not logged in and changed it yet) will be your student number. If you have trouble logging in, email Tim Trant (tim@eecg.utoronto.ca).

If you want to use the UG machines remotely, please see the ECE 297 VNC Quick Start guide on quercus at `https://q.utoronto.ca/courses/309928/pages/remote-access-to-the-ug-computers-using-vn module_item_id=4911667`. It guides you through setting up a graphical link to the UG machines so you can use them from home just as if you were sitting in front of the UG computer.

Once you are logged into the UG computers, use `/local/bin/quartus.start` and `/local/bin/modelsim18.sta` (at a command prompt) to start each of these tools with the appropriate license server connection.

8.1 Installing the Tools on Your Own Machine

Visit this link, and download **Quartus Prime 20.1**[5] **Standard Edition**[6]. You need both Quartus Prime and ModelSim-Intel FPGA Edition (in the latest releases of Quartus ModelSim has been renamed Questa, but it is still the same simulator). They are available for both Linux and Windows; pick whatever OS you prefer. You can click on Individual Files to download only Quartus, ModelSim, and only the device families you need if you want to save download time and disk space. You only require the Arria 10 device families for the assignments in this course, so you can download only those ones if you wish as shown in Fig. 19. You will need to create an Intel account to be able to download any of these components.

No license option   When you start Quartus, it will prompt you for a license file. You can either select the 30-day Quartus trial license (and you can download again to get another 30 days if necessary), or you can connect to the University of Toronto Quartus licenses using a VPN or

---

[5]You can use any version number of Quartus you like that supports Arria 10 devices; it doesn't have to be Quartus 20.1 (which we use in this assignment). All the screen shots and timing results in this handout are obtained using Quartus 20.1, and hence may differ slightly from the numbers you get using a different Quartus version. You will also obtain slightly different timing results on the Linux vs. Windows versions of Quartus. This is normal and fine.

[6]The Lite Edition of Quartus does not require a license, but has some features disabled that you will need to complete this assignment, so you can not use it.
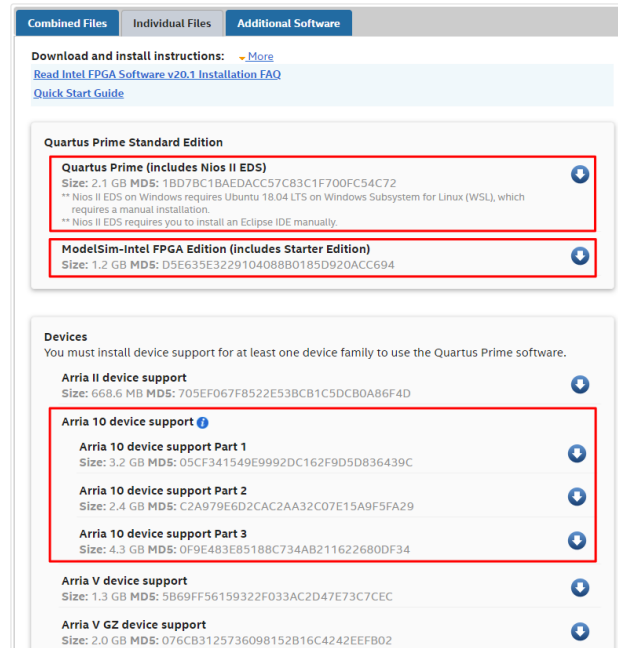
Figure 19: Download Quartus Prime 20.1 Standard Edition, ModelSim, and the Arria 10 device support.

port forwarding, as described below. You can also use the `ModelSim-Intel Starter Edition` of ModelSim/Questa, which does not require a license.

License over EECG VPN   If you are connected to the EECG research network via VPN[7] choose "If you have a valid license file, specify the location of your license file", then enter `27007@mlm.ece.utoronto.ca`[8] as the license file as shown in Fig. 20.[9]

Accessing the UG license with port forwarding   You can use ssh port forwarding from your home computer to the EECG network as follows:

- localhost: personal computer

- remotehost: any ug machine (e.g. ug250.eecg.utoronto.ca, ug251.eecg.utoronto.ca, ...)

- servicehost: gordini.eecg .utoronto.ca

- serviceports: 1802 (flexlm) 27002 (alterad) 7327 (mgcld) (for Modelsim)

If you use port forwarding, the license server is gordini.eecg .utoronto.ca. Forwarding port 1802 and 27002 through the ECE firewall will enable Quartus, while forwarding port 7327 enables Modelsim. Once you have ssh forwarding set up, the License Setup you specify in Quartus should set the license to `<port forwarded to 1802>@localhost`.

---

[7]Contact ecehelp@ece.utoronto.ca for help related to any VPN-related issues, or see these resources: Windows, Ubuntu, or Mac OS. ecehelp is generally reluctant to set up new vpn connections solely for course work so you should try the ssh forwarding solution if you don't have a vpn.

[8]The EECG research network consists of machines like bastet.eecg and anubis.eecg. It is not the UG machines (e.g. ug71.eecg) which are targeted at teaching. The UG machines already have Quartus installed, and the license set up.

[9]You shouldn't need a license for ModelSim as the starter edition doesn't require a license, but the EECG ModelSim license is hosted on 27016@mlm.ece.utoronto.ca.

There is extensive documentation about how to set up port forwarding on the web; a good document for Linux, Windows and MacOS is at https://www.uvic.ca/engineering/ece/current/graduate/computing/port-forwarding/index.php.
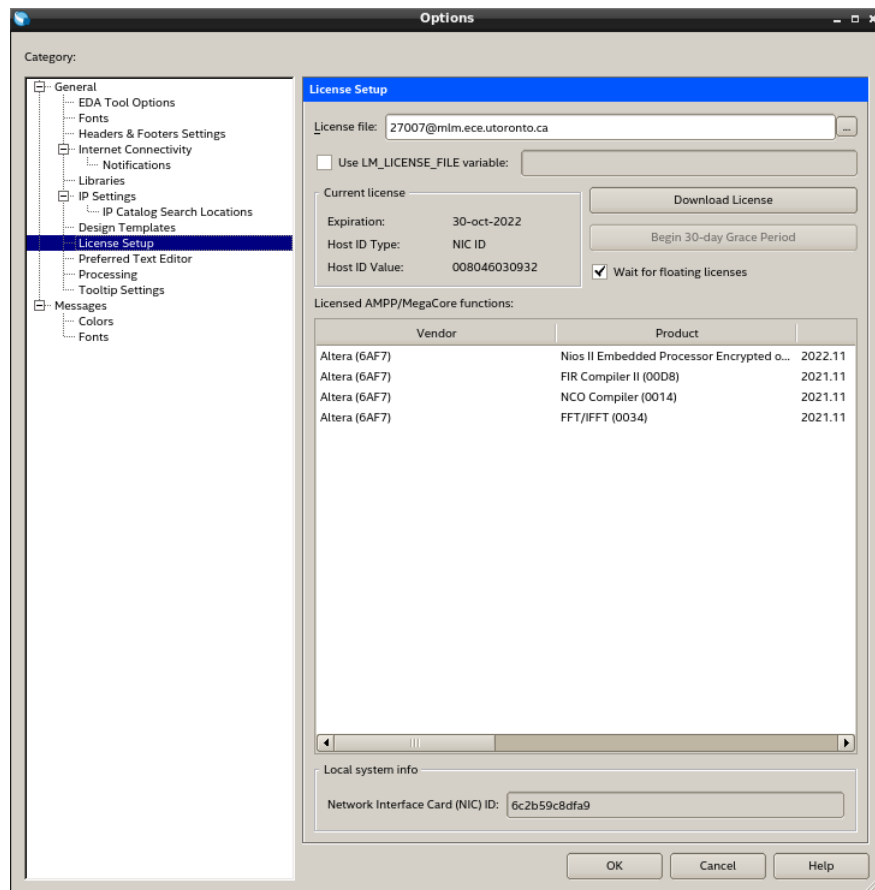

Figure 20: Quartus license setup.