

CSE 598/CEN 598/CSE 494 Lab 1

Speed & Area Trade-offs in FPGA Design

“There are no solutions ... there are only trade-offs.”

— Thomas Sowell

Assigned on Thursday, January 18
Due on Thursday, February 8 @ 11:59 pm

Total points = 100

1 Objectives

This lab assignment is intended to help you:

1	Learn how to use several FPGA CAD tools.
---	--

- Intel Quartus II Prime for synthesis and analysis
- QuestaSim 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 QuestaSim 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. 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 report in PDF format** that 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)? Include the graph in the report. 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). 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.
3. **Verilog source files** of the two circuits that you implement (pipelined and shared hardware).

Create a zip file containing these five things - two Quartus project archives, one PDF report, and two Verilog design files. Use the naming convention for this file `lab1_<firstname1>_<lastname1>_<firstname2>_<lastname2>`. Upload this file to Canvas.

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 $i = 1, \dots, N$ and $\mathbf{z} = [z_1, \dots, z_N]$ 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 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%).

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

	Baseline Circuit	Pipelined Circuit	Shared HW Circuit
Resources for one circuit	36 ALMs + 6 DSPs		
Operating frequency	47.7 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	21.41 mW		
Max. throughput/Watt for a full device			

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

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.

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \dots \quad (2)$$

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 Questa 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.

Download the `lab1.zip` archive from Canvas, and transfer the archive to your home directory on the RC Sol Cluster. There are a couple of different options for this. The simplest option is to go to [My Interactive Sessions](#). Ensure you are connected to the ASU VPN if you are not connected to the ASU network directly before visiting the link. Go to **Files > Home Directory** to add files from your device to the home directory of your Research Computing account. Go back to **My Interactive Sessions** and start a **Sol Desktop** session. Always choose a general partition and a public QOS when starting a session. When choosing resources, remember that you are sharing the nodes with hundreds of other researchers, so be conservative with the number of cores, amount of memory, and session wall-time you choose. You will not need any GPU resources or additional sbatch options. Finally, click **Launch** and wait for your session to become active. After you launch your session, you can navigate to the directory where you placed the `lab1.zip` archive file and unzip it with the following command:

```
unzip lab1.zip -d <destination directory>
```

¹ The archive contains the following files:

- `lab1.v`: Verilog code for the baseline implementation for the circuit in Fig. 2
- `lab1_tb.v`: Verilog testbench
- `lab1.qpf`: Quartus project file
- `lab1.qsf`: Quartus settings file
- `SDC1.sdc`: Timing constraint 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`.

¹More information on working with RC Sol Computing Nodes can be found in Appendix A

- **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.

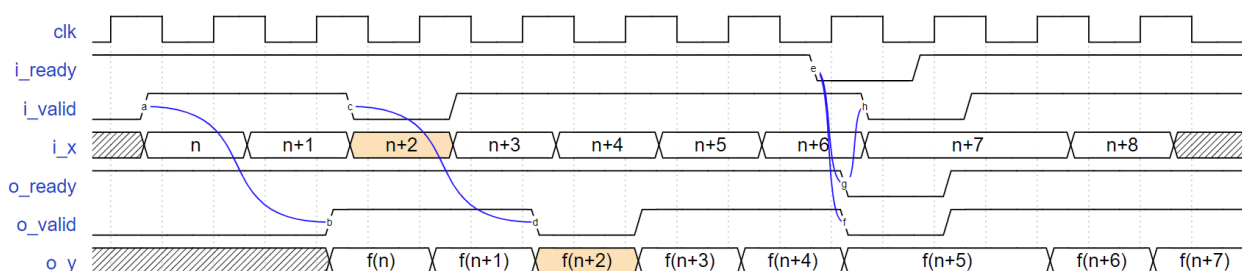


Figure 1: Waveform of the interface between your circuit and the testbench. Note that signals are slightly delayed from the clock rising edge to show an exaggerated effect of signal propagation delays.

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.v file.

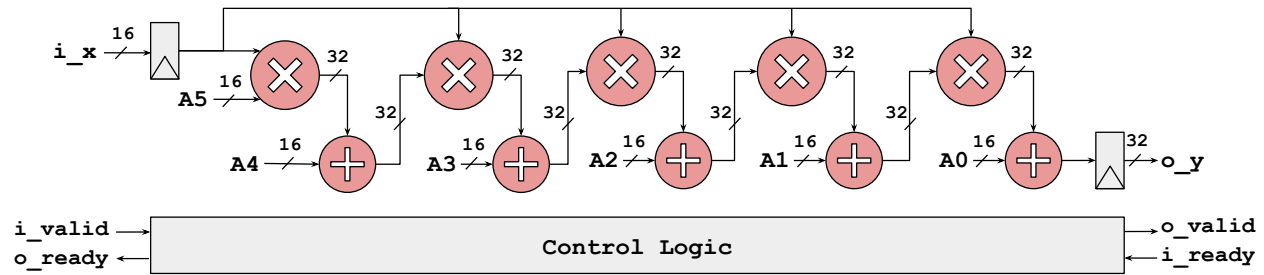


Figure 2: The baseline circuit for computing the 5th Taylor polynomial approximating e^x . For this implementation, eq. (2) is factored as $y = (((((a_5x + a_4)x + a_3)x + a_2)x + a_1)x + a_0)$, where $a_0 = 1, a_1 = 1, a_2 = \frac{1}{2}, a_3 = \frac{1}{6}, a_4 = \frac{1}{24}$, and $a_5 = \frac{1}{120}$.

5.1 Opening the Project

Start Quartus by typing the following command in the terminal.

```
1 module load bittware/quartus-pro-23.4
2 quartus &
```

Notice that the command to load the module for QuestaSim may be different. To see all available modules for RC Computing, run the following command in the terminal:

```
1 module avail
```

Look for the bittware package corresponding to a version of Questa and load it with the command above. You can also check your \$PATH variable after loading the module to ensure that all of the /bin directories are included correctly. More information on working with the RC Sol Cluster can be found in Appendix A. When you start Quartus for the first time, it will ask you to select a license for the version we are using. We have acquired a license from Intel for use in this class, you just need to set it up. For Quartus, choose the option that allows you to specify a license file. If you ever need to reconfigure the license in Quartus, you can always go to **Tools > License Setup...** In the license setup options, enter `27002@en42282831.cidse.dhcp.asu.edu` as the License File. Wait for the application to read the license file and for the core functions to load, then your license is ready for use. QuestaSim requires the same license, however, instead of configuring the license in the GUI, run the following command in the terminal:

```
1 export LM_LICENSE_FILE="27002@en42282831.cidse.dhcp.asu.edu"
```

This is the environment variable that QuestaSim will access when checking for a license. For those who may not be familiar with Linux, each instance of a terminal has a unique environment, so you will need to load the module for Quartus and QuestaSim and also export the license file for QuestaSim each time you open a new terminal.

Now you are ready to start working on the lab! Select **File > Open Project** in Quartus 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. If it tells you this device is not available, you can change the device under **Assignments > Devices**. Your compilation results will be very different that what is reported in this document, but the behavior will be the same so it is not a huge issue.

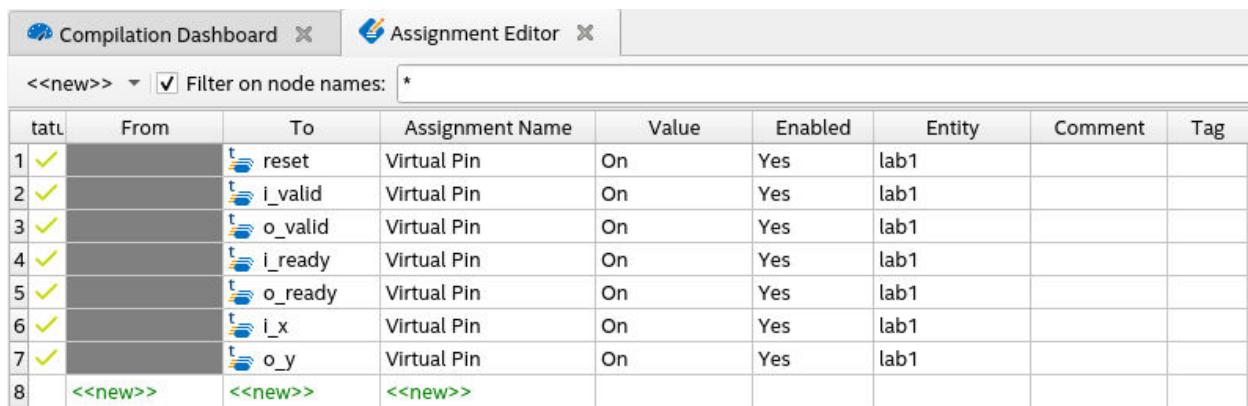
5.2 Setting Up Virtual I/O Pins

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.



tatl	From	To	Assignment Name	Value	Enabled	Entity	Comment	Tag
1 ✓		reset	Virtual Pin	On	Yes	lab1		
2 ✓		i_valid	Virtual Pin	On	Yes	lab1		
3 ✓		o_valid	Virtual Pin	On	Yes	lab1		
4 ✓		i_ready	Virtual Pin	On	Yes	lab1		
5 ✓		o_ready	Virtual Pin	On	Yes	lab1		
6 ✓		i_x	Virtual Pin	On	Yes	lab1		
7 ✓		o_y	Virtual Pin	On	Yes	lab1		
8	<<new>>	<<new>>	<<new>>					

Figure 3: Assigning virtual pins using the Assignment Editor.

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.

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 `SDC1.sdc`. Looking at this file, you should see:

```
1 create_clock -period 1 -name clk clk # in SDC1.sdc
```

This SDC file has created a timing constraint for the clock net (called `clk` in `lab1.v`) 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.

A window called **Timing Analyzer** will open up at the end of compilation. If the timing analyzer does not open, you can open it by clicking **Tools > Timing Analyzer**. You will see that it says "Timing requirements not met", and a slack of -19.956 is shown in the Setup Summary panel. Also shown is the word-case operating condition - **Slow 900mV -40C Model**.

This indicates that the design did not meet its timing constraint (of a 1 ns clock period) by 19.95 ns. Consequently the fastest cycle time for the design is $1 + 19.95 \text{ ns} = 20.95 \text{ ns}$, or 47.7 MHz.

You will see the the timing analysis was run on 4 corners - Slow 900mV 100C Model, Slow 900mV -40C Model, Fast 900mV 100C Model, Fast 900mV -40C Model. 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 V_t also increases as T drops (reducing speed), so it is now necessary to check timing at both “temperature corners”. In the Timing Analyzer, you can select each operating condition/corner in the **Set Operating Conditions** panel, and click **Reports > Timing Slack > Report setup summary**, to see the slack for each 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. You can see the timing report in the compilation report as well. Open it by clicking **Processing > Compilation Report** from the main Quartus window.

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

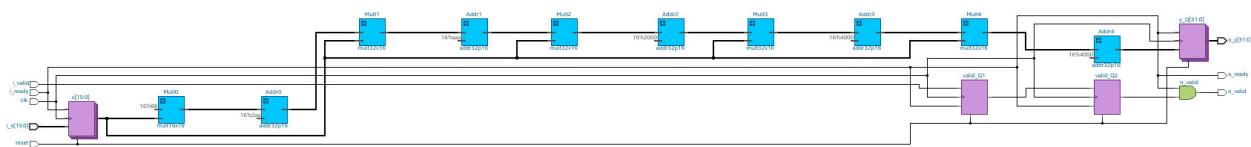


Figure 4: RTL view of the netlist

The design looks as expected, with 5 multipliers and 5 adders between the x and y register banks. 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.v` 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. $> 800 \text{ MHz}$), 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 > Place Stage > Resource Utilization 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 **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 in the Resource Utilization by Entity report shown in Fig. 5, you can see that 70 look-up tables (called ALUTs by Intel), 52 registers and 8 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 slightly more than that.

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**) does not use a DSP block. That is because **Mult0** involves multiplication by a constant. Quartus understands this and maps it to LUTs/ALMs because any bits multiplied by zeros in the partial products get optimized away. On the other hand, all the other 16×32 multipliers are implemented using 2 DSP blocks each. Notice also that the number of ALUTs used to implement the 4 adders changes based on exact bitwidths required.

Fitter Resource Utilization by Entity								
Q <<Filter>> (use <string> to invert filter)								
	Compilation Hierarchy Node	[A] ALMs used in final placement	Combinational ALUTs	Dedicated Logic Registers	M20Ks	[A] DSP Blocks used in final placement	Pins	Virtual Pins
1	▼	45.0 (16.2)	70 (1)	52 (52)	0	8	2	53
1	Addr0	5.0 (5.0)	10 (10)	0 (0)	0	0	0	0
2	Addr1	10.0 (10.0)	20 (20)	0 (0)	0	0	0	0
3	Addr2	4.0 (4.0)	8 (8)	0 (0)	0	0	0	0
4	Addr3	2.0 (2.0)	7 (7)	0 (0)	0	0	0	0
5	Addr4	1.8 (1.8)	7 (7)	0 (0)	0	0	0	0
6	Mult0	6.0 (6.0)	17 (17)	0 (0)	0	0	0	0
7	Mult1	0.0 (0.0)	0 (0)	0 (0)	0	2	0	0
8	Mult2	0.0 (0.0)	0 (0)	0 (0)	0	2	0	0
9	Mult3	0.0 (0.0)	0 (0)	0 (0)	0	2	0	0
10	Mult4	0.0 (0.0)	0 (0)	0 (0)	0	2	0	0

Figure 5: Resource utilization by entity.

Next, take a look at the Resource Usage Summary section of the compilation report shown in Fig. 6. 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 (63 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 $63 - 27 = 36$ ALMs.

Scrolling down in the resource usage summary, you see that although 8 DSP blocks were needed to map four 32×32 multiplications, Quartus is able to pack them intelligently (“dense merging”) and use only six DSP blocks overall. Therefore, in Table 1, we mention that 6 DSP blocks are required for one instance of our circuit.

Fitter Resource Usage Summary			
🔍 <<Filter>> (use !<string> to invert filter)			
	Resource	Usage	%
3	Logic utilization (ALMs needed / total ALMs on device)	63 / 427,200	< 1 %
4	▼ ALMs needed [=A-B+C]	63	
1	▼ [A] ALMs used in final placement [=a+b+c+d]	46 / 427,200	< 1 %
1	[a] ALMs used for LUT logic and registers	13	
2	[b] ALMs used for LUT logic	23	
3	[c] ALMs used for registers	10	
4	[d] ALMs used for memor... to half of total ALMs)	0	
2	[B] Estimate of ALMs recoverable by dense packing	10 / 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	
5			
6	Difficulty packing design	Low	
7			
8	▼ Total LABs: partially or completely used	7 / 42,720	< 1 %
1	-- Logic LABs	7	
2	-- Memory LABs (up to half of total LABs)	0	
9			
10	▼ Combinational ALUT usage for logic	70	
1	-- 7 input functions	0	
2	-- 6 input functions	0	
3	-- 5 input functions	0	
4	-- 4 input functions	0	
5	-- <=3 input functions	70	
11	Combinational ALUT usage for route-throughs	17	
12			
13	▼ Dedicated logic registers	52	
1	▼ -- By type:		
1	-- Primary logic registers	44 / 854,400	< 1 %
2	-- Secondary logic registers	8 / 854,400	< 1 %

Figure 6: Resource usage summary report.

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.v`). The Tcl command corresponding to what you have selected in the menu is shown at the bottom of the window as shown in Fig. 7, in case you would prefer to directly type the command in the future. Click the **Ok** 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**.

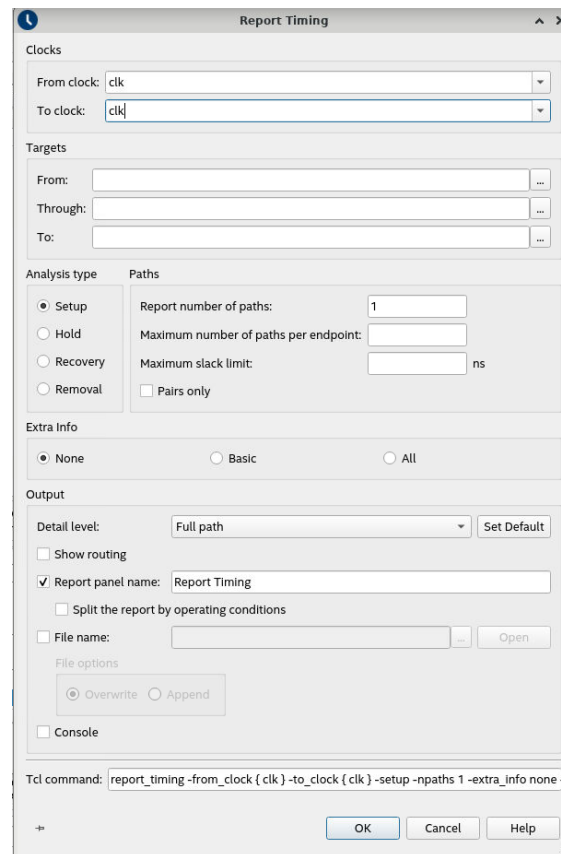


Figure 7: Report timing window.

The graphical waveform display shown in Fig. 8 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 (yellow highlight) is $3.789 - 3.518 = 0.271$ ns, which represents the effect of the clock skew on the speed of the circuit. The data path delay is fairly long, 21.356 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 green box in Fig. 8 list the more interesting “data path” that shows the computation that is 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 ALMs which implement the first multiplier and adder (**Mult0** and **Addr0**),
- followed by another multiplier (**Mult1**) split accross two DSP blocks (indicated by locations **MPDSP_X100_Y93** and **MPDSP_X100_Y94**),
- followed by an adder (**Addr1**) created from ALMs i.e. 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[31]** 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

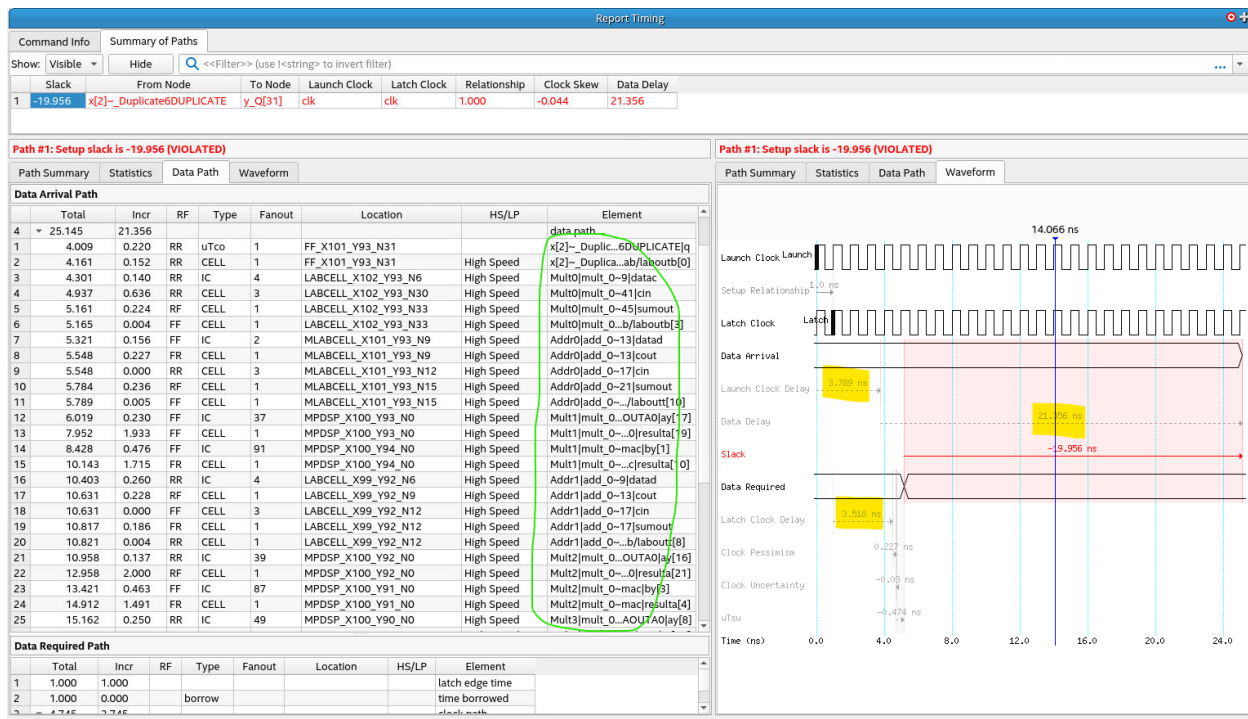


Figure 8: Analyzing the critical path delay of the circuit.

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. Purple elements indicate blocks on the chip, while grey 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. 9 shows the entire critical path of the design, showing:

- the launching register (in pink)
- the first multiplier **Mult0** being implemented in logic cells (purple boxes)
- the first adder **Addr0** being implemented in logic cells
- feeding **Mult1** (second grey block with two purple DSP blocks inside it), then
- a sequence of logic elements 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 capturing 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. 10, it is again clear that the critical path has been localized well, as it spans a very small part of the chip.

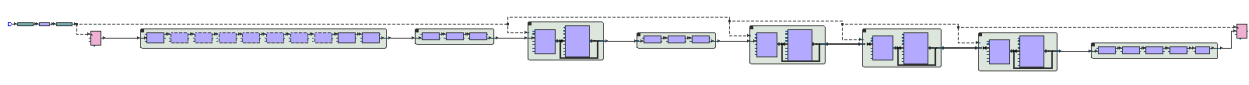


Figure 9: Viewing the critical path in the Technology Map Viewer.

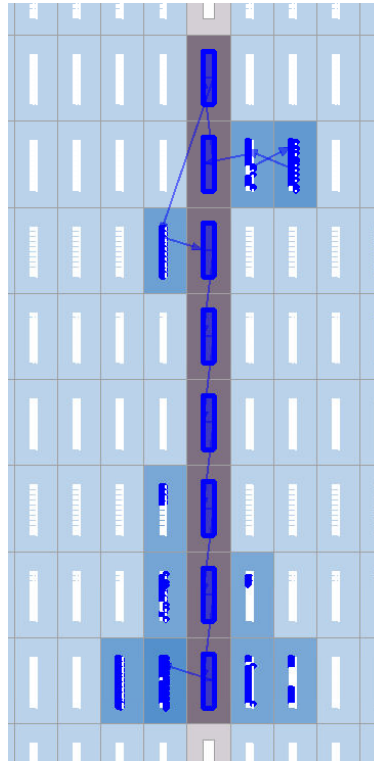


Figure 10: Viewing the critical path in the Chip Planner. DSP blocks are in grey and logic blocks are in blue.

Now let's do some experiments related to timing closure. Change the clock constraint in the SDC file (SDC1.sdc) to correspond to a delay of 50 MHz. Recompile the project and after the Timing Analyzer finishes, it will still report a negative slack. This slack will be much smaller than the slack we obtained earlier. Using the fastest cycle time you calculated from the worst-case slack time from the Timing Analyzer, update the clock period specified in the `SDC1.sdc` file. Recompile the project and after the Timing Analyzer finishes, it should report a positive slack time. Your design now meets the timing specifications. If you still see timing violations, go back to determining the fastest cycle time that the design can achieve and try recompiling the design. Timing closure is a crucial part of the design flow and understanding how to identify critical paths and how to adjust your design for timing constraints can be complicated.

5.6 Simulating for Correctness

We are now ready to simulate the design to see if it is behaving correctly. Before you execute QuestaSim, we need to set environment variable as follows:

```
1 export LM_LICENSE_FILE="27002@en42282831.cidse.dhcp.asu.edu"
```

Start QuestaSim by typing the following command in the terminal:

```
1 /packages/apps/fpga/Quartus_Pro/23.4/questa_fe/bin/vsim &
```

A tutorial on how to use the QuestaSim GUI to compile and simulate a design is downloadable from [this link](#) as the “Use Questa with Testbenches” tutorial.

Alternatively, you can simulate your design by typing the commands at the QuestaSim command prompt at the bottom of the QuestaSim 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 <absolute path to lab1.v> # 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.v # Compile lab1.v
4 vlog lab1_tb.v # Compile the testbench
5 vsim work.lab1_tb -voptargs=+acc
```

You can also put these commands in a file (let’s call it ‘runfile’). And then on the QuestaSim prompt, you can run ‘do runfile’. This is easier than typing individual commands.

We want to monitor all the signals in `lab1.v`. Select **View > Wave** if the waveform window isn’t visible. To add these signals to the waveform viewer, right-click the “dut” line in the “sim” panel and click ‘Add Wave’. You could also choose to show any other signals from `lab1` or `lab1_tb` and add them to the waveform window as well. You can right-click on specific signals in the “Objects” panel and then click ‘Add Wave’.

After that, type the following command in the QuestaSim command prompt:

```
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
```

² 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. 11. 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 QuestaSim 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.

²You 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 `SDC1.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.

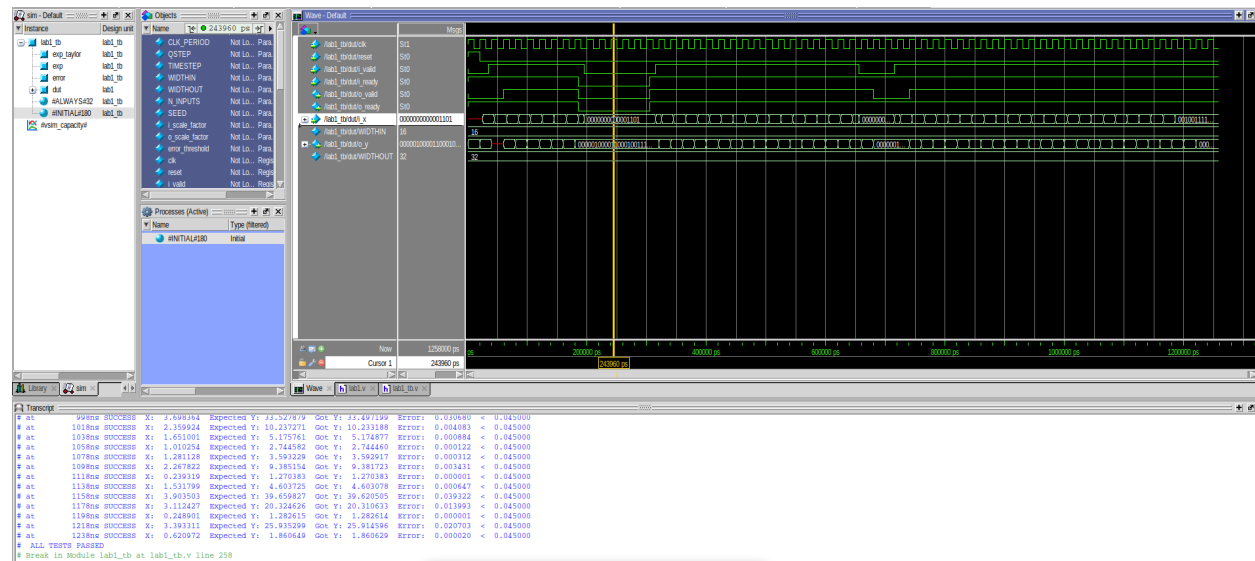


Figure 11: The QuestaSim simulation window.

```
gnuplot graph.gnu
```

GNU plot is a graphing utility that is already installed on the RC Sol 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. 12) will be generated in the project directory and can be viewed with image viewers. On the Research Computing machine, you can run the command below on the terminal to open the plot:

```
display graph.png
```

Or you can download it to your local device and view it there. Feel free to implement more terms of the exponential function Taylor expansion in `lab1.v` and observe the effect on the 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).

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. QuestaSim) 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. We will use a method

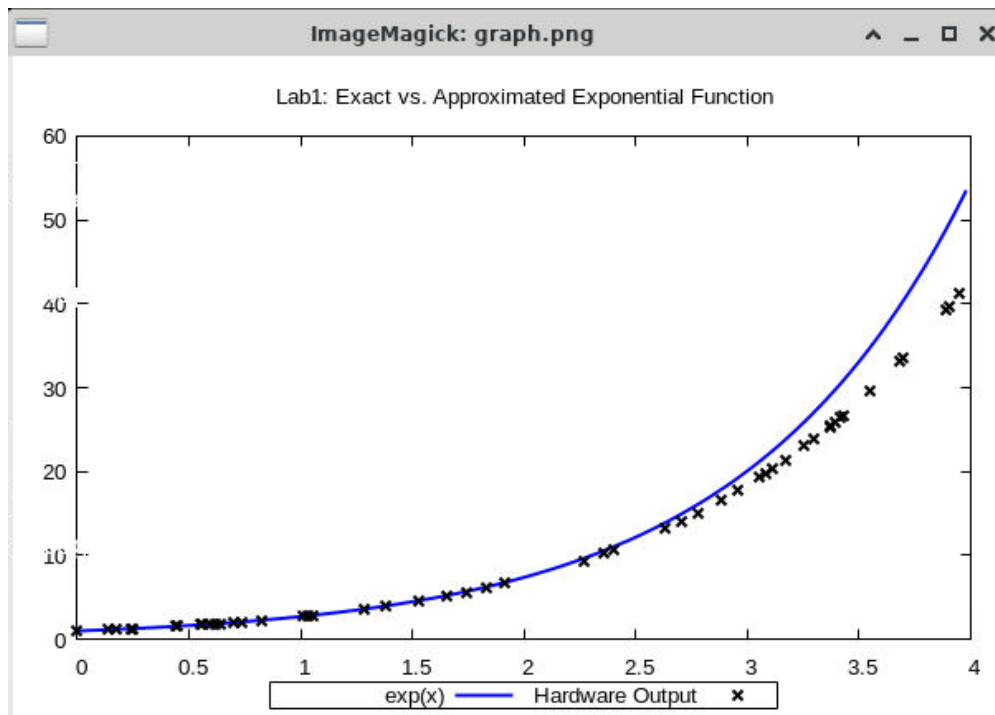


Figure 12: Plot comparing the exact exponential function vs. hardware output of the example circuit. This can be generated using the `graph.gnu` script provided in this assignment.

that is approximate and provides an estimate of power consumption without performing even any digital simulation.

From Quartus, select **Processing > Settings > Power Analyzer Settings**. Uncheck 'Use input files to initialize toggle rates and static probabilities for during power analysis'. Then click on 'Use default value' under the 'Default toggle rates for unspecified signals'. Keep the value as 12.5%. This means that we are assuming that, on average, during the application execution, the signals in the design toggle 12.5% of the time. This is a commonly accepted value of signal activity and is used to estimate power consumption. The Power Analyzer settings should now be similar to Fig. 13.

Click OK, and then **Processing > Start > Start Power Analyzer**. Note that to run the Power Analyzer, you must have compiled the design, including running the assembler to create a programming file. In the 'Compilation Report', click on the Power Analyzer and then on Summary. The power summary will look similar to Fig. 14. The 'Total On-Chip Power Dissipation' is about 3000mW or 3W. Observe the 'Core Dynamic On-Chip Power Dissipation' and 'Device Static On-Chip Power Dissipation'. Since our design is very small, 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. To compute that, we need to look at the "On-Chip 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 $17.15 + 3.59 + 0.67 = 21.41$ mW @ 47.7 MHz as shown in Fig. 15, for one instance of your computation engine.

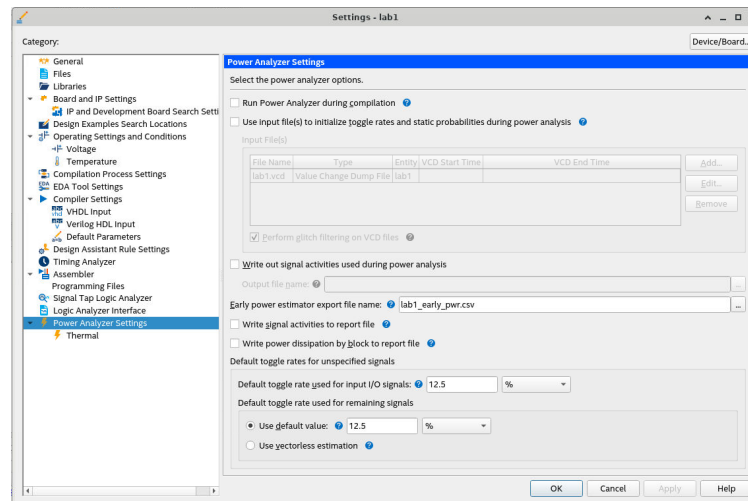


Figure 13: Power Analyzer Settings.

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.



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 QuestaSim 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.

Start by creating a version that has two pipeline stages. Observe the critical path in the timing reports generated during Timing Analysis. This will point you to the frequency of operation of your design. Now, pipeline the design further to achieve a higher clock frequency.

Table of Contents	
<ul style="list-style-type: none"> Flow Summary Flow Settings Flow Non-Default Global Settings Flow Elapsed Time Flow OS Summary Flow Log Synthesis Fitter Timing Analyzer Assembler Timing Analyzer GUI EDA Netlist Writer <ul style="list-style-type: none"> Summary Messages Simulation Power Analyzer <ul style="list-style-type: none"> Summary Power Savings Summary Settings Messages Generated Files Operating Conditions Used On-Chip Power Dissipation by Block Type <ul style="list-style-type: none"> Current Drawn per Supply Confidence Metric Details Signal Activities Flow Messages Flow Suppressed Messages 	
Power Analyzer Summary	
<input type="text" value="Filter (use <string> to invert filter)"/>	
Power Analyzer Status	Successful - Sat Jan 20 18:19:57 2024
Quartus Prime Version	23.4.0 Build 79 11/22/2023 SC Pro Edition
Revision Name	lab1
Top-level Entity Name	lab1
Family	Arria 10
Device	10AX115N2F4511SG
Timing Models	Final
Power Models	Final
Device Status	Final
Total On-Chip Power Dissipation	3078.64 mW
Transceiver Standby On-Chip Power Dissipation	0.00 mW
Transceiver Dynamic On-Chip Power Dissipation	0.00 mW
I/O Standby On-Chip Power Dissipation	0.06 mW
I/O Dynamic On-Chip Power Dissipation	2.69 mW
Core Dynamic On-Chip Power Dissipation	75.30 mW
HPS Standby On-Chip Power Dissipation	0.00 mW
HPS Dynamic On-Chip Power Dissipation	0.00 mW
Device Static On-Chip Power Dissipation	3000.58 mW
High Bandwidth Memory Standby On-Chip Power Dissipation	0.00 mW
High Bandwidth Memory Dynamic On-Chip Power Dissipation	0.00 mW
Analog/Digital Converter Standby On-Chip Power Dissipation	0.00 mW
Analog/Digital Converter Dynamic On-Chip Power Dissipation	0.00 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data

Figure 14: Power consumption summary.



Implement a pipelined version of the same example circuit and verify its functionality using the same provided testbench. Optimize the design 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 may also recognize this as a Serialized Implementation. 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, and 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.

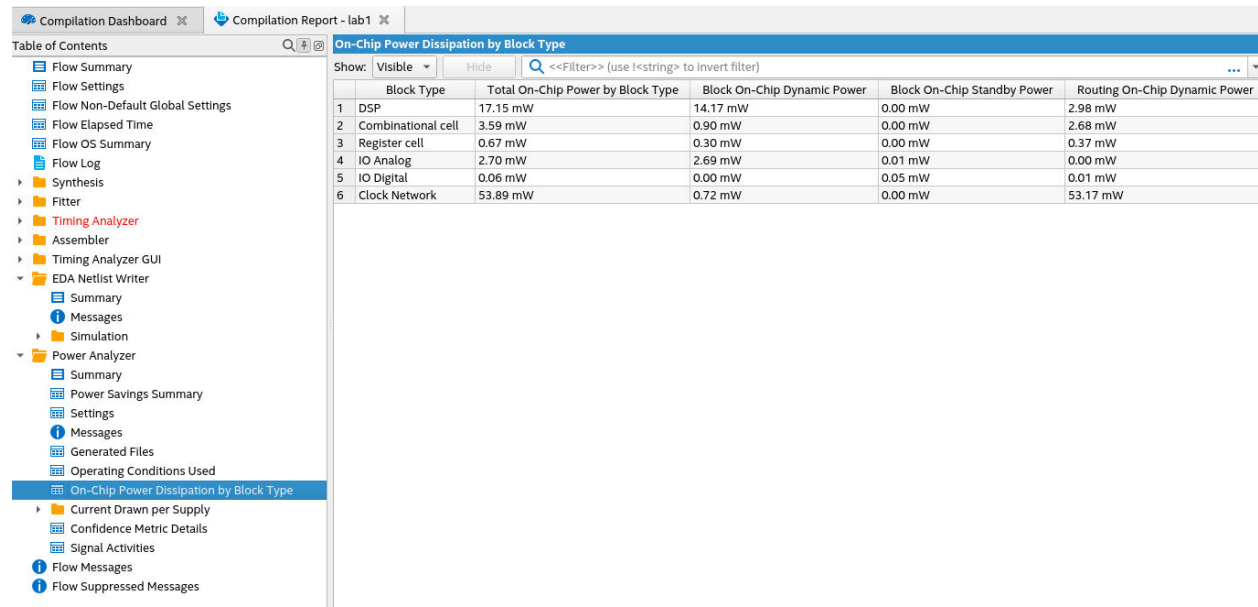


Figure 15: Power consumption by block type.

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.
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. Avoid repetitive, duplicated code by creating and instantiating reusable modules.

9 Appendix A: Using ASU Research Computing Sol Cluster

For the labs in this class, you will mainly be using the ASU Research Computing Sol Cluster. Research Computing is a core facility within the Knowledge Enterprise's Research Technology Office dedicated to enabling research, accelerating discovery, and spurring innovation at ASU through the application of advanced computational resources to grand research challenges. Learn more at researchcomputing.asu.edu. Your account on the Sol supercomputer has been created and is ready to use.

9.1 Getting Started

Our supercomputers are accessed using your ASURITE login and password. For uninterrupted access to the supercomputer, please connect to the Cisco AnyConnect Client VPN. If you do not have the Cisco AnyConnect Client VPN already installed, visit sslvpn.asu.edu and follow the installation instructions. For additional details and troubleshooting, please read this document.

Once connected to the VPN, you can access the supercomputer through the web portal:

The web portal for the Sol supercomputer can be found at <https://sol.asu.edu>.

Before submitting jobs to the supercomputers, please review the [ASU Research Computing Acceptable Use Policy](#).

9.2 Training and Documentation

ASU Research Computing offers frequent workshops throughout the year to help you use our systems. A comprehensive list of training can be found [here](#), and recordings and materials from past workshops can be found [here](#).

In addition, Research Computing offers a large collection of helpful how-to resources for our supercomputers that can be found on our [documentation page](#). Follow this link to read more about how to set up interactive sessions, upload files, etc.

9.3 Office Hours and Consultation

Research Computing offers weekly office hours over Zoom. Our technical experts can assist you with using the supercomputer, installing and troubleshooting software, and providing assistance with your computing needs. The schedule and Zoom information for office hours can be found on [documentation page](#).

9.4 Contact

The Research Computing Slack channel: [#rc-support](#) is available for quick questions and online support during normal business hours. For all other questions or for assistance after 5 PM or on weekends, submit your request to rtshelp@asu.edu and a ticket will be created on your behalf.