

High Level Overview:

In this lab we will explore the concepts needed to deal with multiple clocks in a design, timing analysis, and methods for dealing with timing problems.

We will make use of IP (Intellectual Property) components. IP are functions/modules that someone else has already designed. Much like in software design, in large-scale digital designs we do not write everything we need. In software, this may take the form of libraries, such as the C++ standard template library, or perhaps things like ethernet stacks. Operating systems are another example of software based IP.

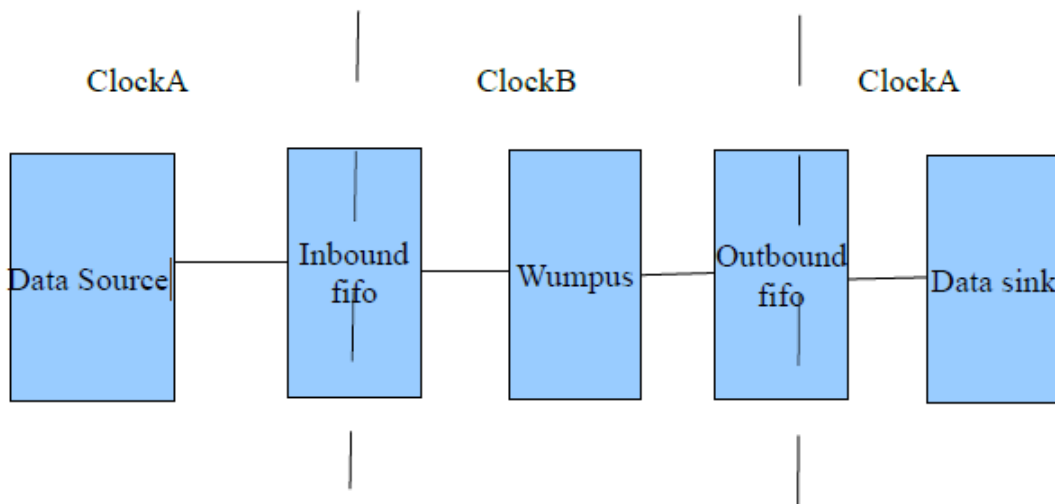
The Quartus tool has a lot of IP that it can generate. IP can also be purchased from 3rd parties, or found in open-source sites such as opencores.org. In Quartus, the built-in IP is accessed by means of the IP Catalog.

In this design, you will go through the steps needed to create a design with multiple clocks, set up the timing analyzer, build the design and check timing, show methods for improving the timing. This lab has no “real” purpose other than that but it does closely resemble some of the real world designs dealing with data streaming and encryption.

Detailed Overview:

What we are going to build is represented below. Data Source, Wumpus, Data sink and the top level file are all created. You will go through the steps of creating the PLL's and FIFO's. You will have your chance to modify the design later.

The ClockA and ClockB are going to be created by a Phase Locked Loop (PLL) for each clock (we will utilize 2 PLLs) which is an analog component and part of the FPGA. Using the Megawizard (described later) will can instantiate the PLL into our design with parameters that are specified. A good explanation of PLL's can be found here: <http://en.wikipedia.org/wiki/PLL>



There is a data source (sourcing of the data), in this case a counter that is operating in the ClockA domain. The data produced is going to be sent to a FIFO (First-In-First-Out) memory. A FIFO is a means of buffering data (holding it for later). If ClockA is faster than ClockB, the counter will produce data faster than it can be sent through the Wumpus block. The Wumpus block will process the data as fast as it can, given that it is on a different clock domain and then send the data on to another FIFO which will in turn send the data on to a data sink (consumer of data).

In real life, the data source might be something like a PCIe bus or a camera input that is operating at a high speed, sending in bursts of data. The Wumpus block might be some signal processing or encryption scheme that needs to run at a slower clock. The Data Sink might be something like an Ethernet port that can accept data in bursts.

The FIFO contains a memory block, and other logic to implement what is known as a ring buffer. There is an address pointer (pointer) to the first empty memory location, this is known as the head pointer. Data coming into the FIFO is stored at the head pointer, and then the pointer is incremented. When the pointer gets to the end of the memory, the address/pointer wraps back to the first address in the memory. There is also an address pointer to the last filled memory

location, this will either be the same as the head if the ring buffer is empty, or lagging the head pointer by the amount of data that has been stored in the ring buffer. When data is read from the FIFO, it is taken from the tail pointer location and then the pointer/address is incremented. So the tail pointer chases the head pointer, and the region in between contains the buffered data.

We need the FIFO for several reasons. First, it gives the Data Source a place to dump data in a high-speed burst. We keep track of how full the FIFO is, and when there is room for a burst we let the data source know. It gives the Data Sink a place to buffer up the size of data it wants to burst. The FIFO also allows us to cross clock domains safely.

A clock domain is nothing more than logic that operates with a particular clock, i.e. all the logic is running at 200Mhz. If we have modules/portions of the design operating at different clocks (144Mhz), and we need to send data from one module to the other, we could have a problem. Since the clocks are unrelated (completely different clock frequencies with no phase relationship), we can't under any circumstances say that we can meet the timing requirements when transferring data from one clock domain to another. Since the inbound data could change at any time relative to the other clock – we can't know if we are meeting setup and hold times.

The FIFO can be built so that it has one clock for filling the FIFO, and a different clock for reading from the FIFO, this safely lets us move data from one clock domain to another. The FIFO takes care of crossing information across the clock domain.

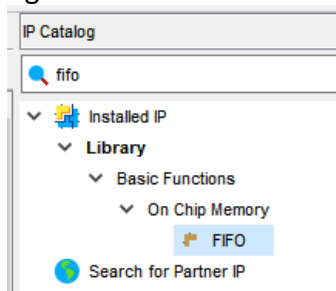
Here is some more information on FIFOs:

[https://en.wikipedia.org/wiki/FIFO_\(computing_and_electronics\)](https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics))

Lab Directions:

Restore the provided Quartus archive (Project->Restore Archived Project) for this lab *.qar. The design is a de1_soc_top with a few VHDL files. We will start making the FIFOs first followed by the PLL's (Phase Locked Loops) to generate the clocks we need.

We will generate the PLLs and FIFOs from the IP Catalog which will invoke the Megawizard. From Quartus go to Tools->IP Catalog (you will get a new window that shows up called IP

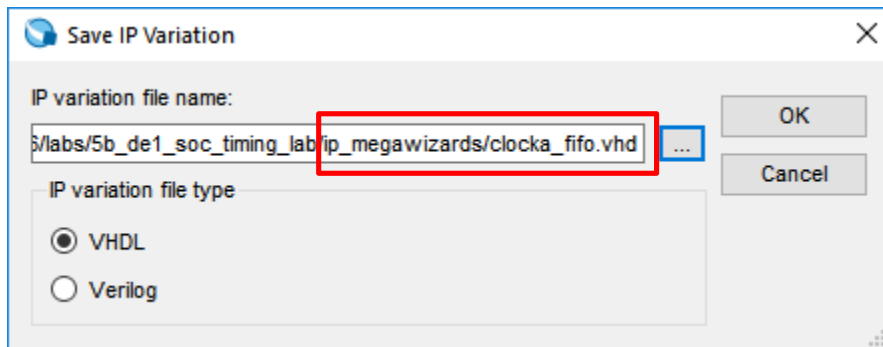


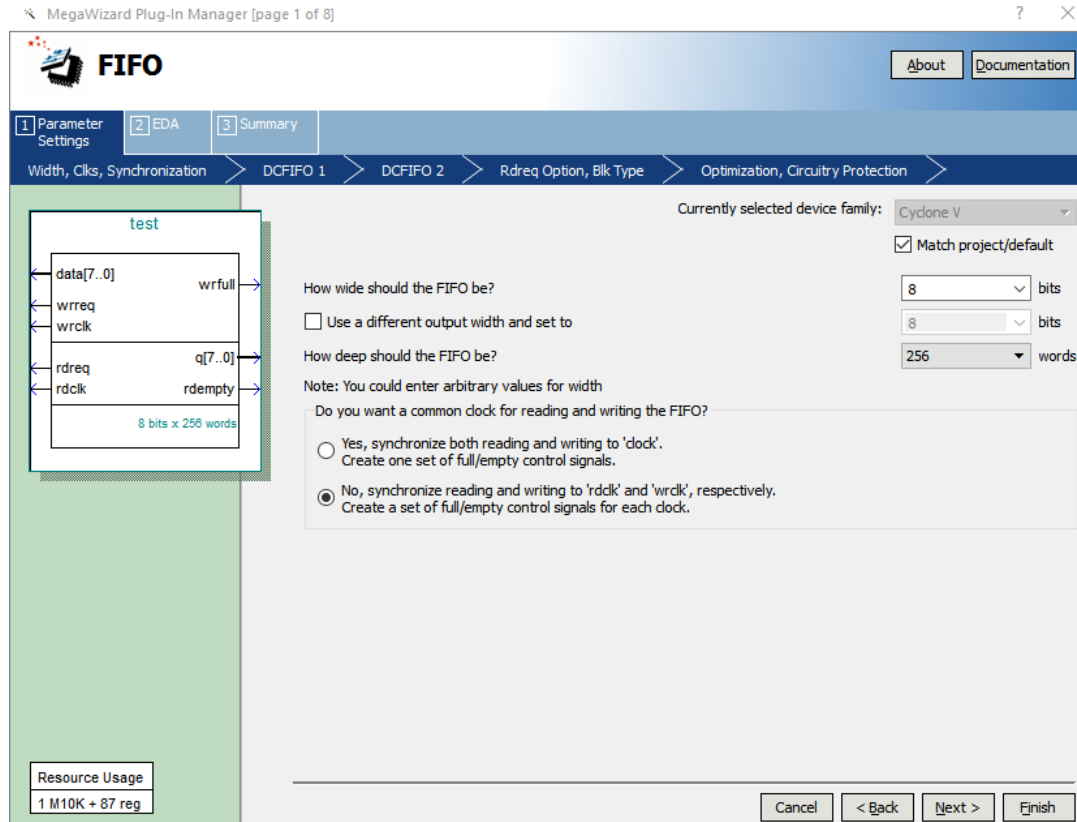
Catalog). In the Search window type in “fifo” and then double click on “FIFO” you will get a dialog box asking for the “IP Variation file name”. You must provide a

name for the IP you are generating and the path as well as VHDL or Verilog, observe that I have named this one clocka_fifo.vhd and also have placed it into the ip_megawizards folder. The IP creates a fair number of files and often makes sense to partition the directory structure to keep like files together.

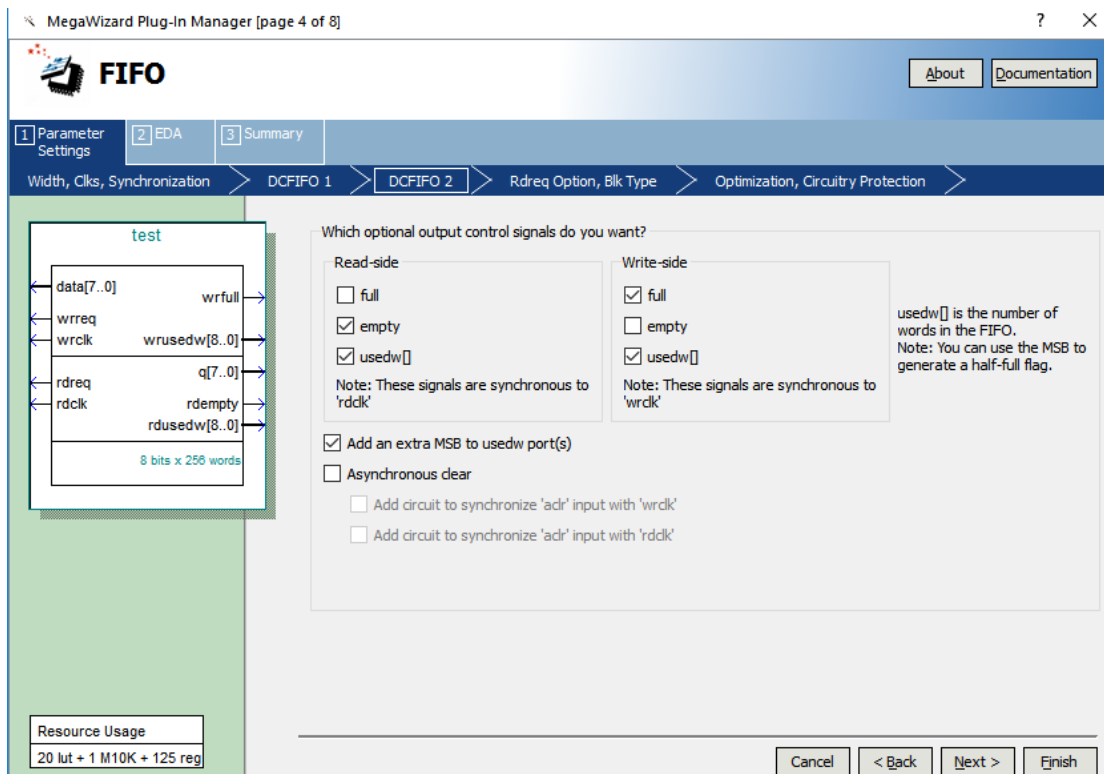
MAKE SURE YOU PLACE THE FILES IN THE IP_MEGAWIZARDS DIRECTORY.

Hit OK:



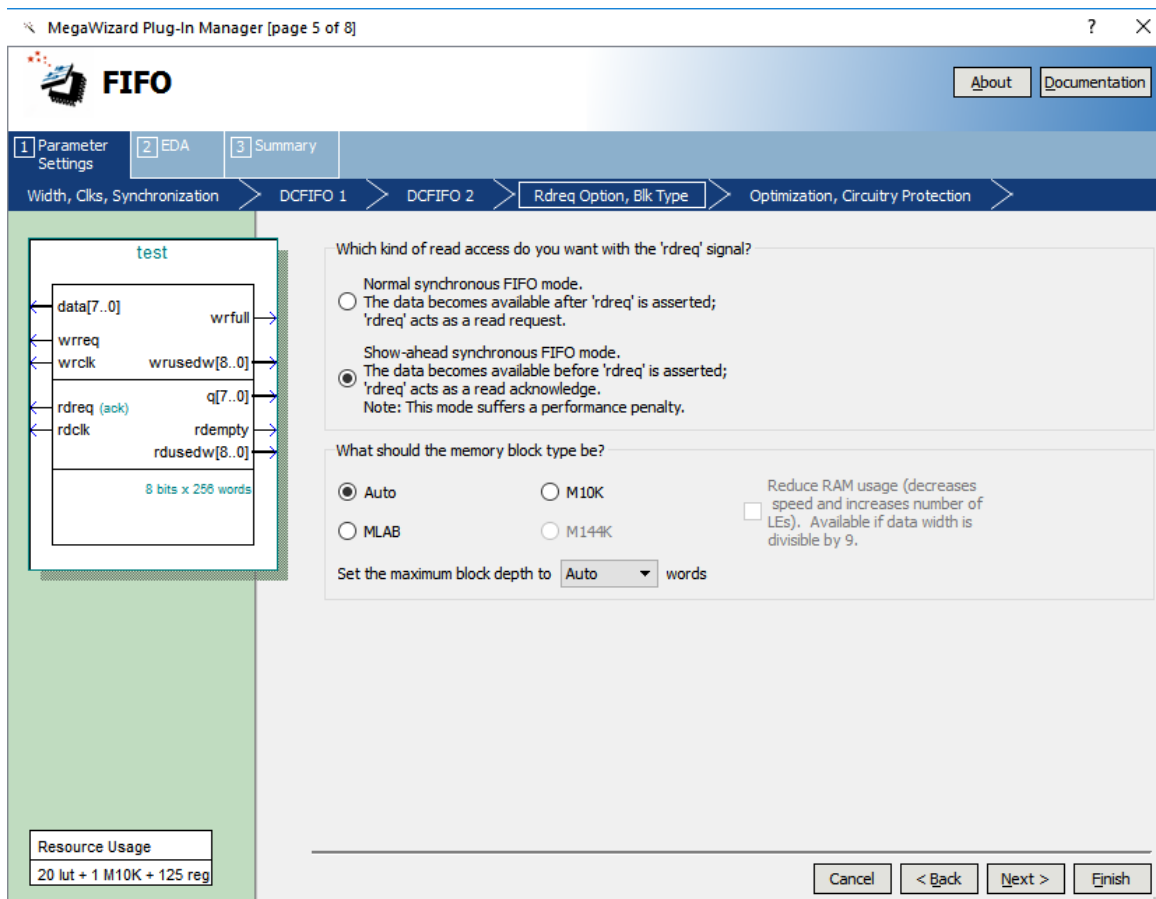


Select the options as shown. Observe that the symbol shown has a clock for writing, wrclk, and one for reading, rdclk. Also observe that you can change the width of the data, and specify the maximum amount of data the FIFO can hold. Hit Next, the default selection is fine, hit Next again. You should get a box like below, observe the check boxes selected, make sure you have selected the same boxes.



This will give us some signals that we can use to determine if we can read or write data from or to the FIFO. On the write side, we have a full signal – we can't put any more data in if it is full, and also a vector wrusedw (used words) that will tell us exactly how much data we have in the FIFO. On the read side, we have a signal that tells us when the FIFO is empty, rdempty. We can't read from the fifo if it is empty. We also have a signal rdusedw which tells us how much data is available to be read from the FIFO.

Hit Next, you should see this:



Note that I have checked Show-ahead mode. This means that the next data available is always driven from the output (q[7:0]), when I want to read the data, the data is already there, when I assert rdreq it acknowledges that I have read the data and advances the read pointer 1 location internal to the FIFO. Hit Finish, the defaults are fine for the rest of the screens. The wizard generation should have put multiple files into your ip_megawizards project folder as well as place a ip_megawizards/clocka_fifo.qip in the project navigator files list. The QIP file is essentially a list of files for the IP.

clocka_fifo.vhd is the entity-architecture pair for the FIFO. clocka_fifo.cmp is the component declaration for the FIFO. The .cmp file is not part of the design files, it is there for reference and you can open the file in a text editor and copy and paste the component declaration into your design file that will instantiate the FIFO.

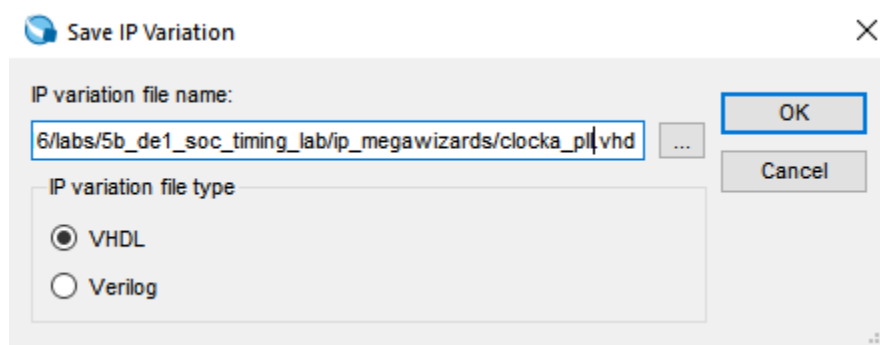
Here is the link to the FIFO users guide. We covered the surface of the FIFO, there are a lot more details on how each of the signals function:

SCFIFO:Single Clock FIFO,

DCFIFO: Dual Clock FIFO (we are using the DCFIFO)

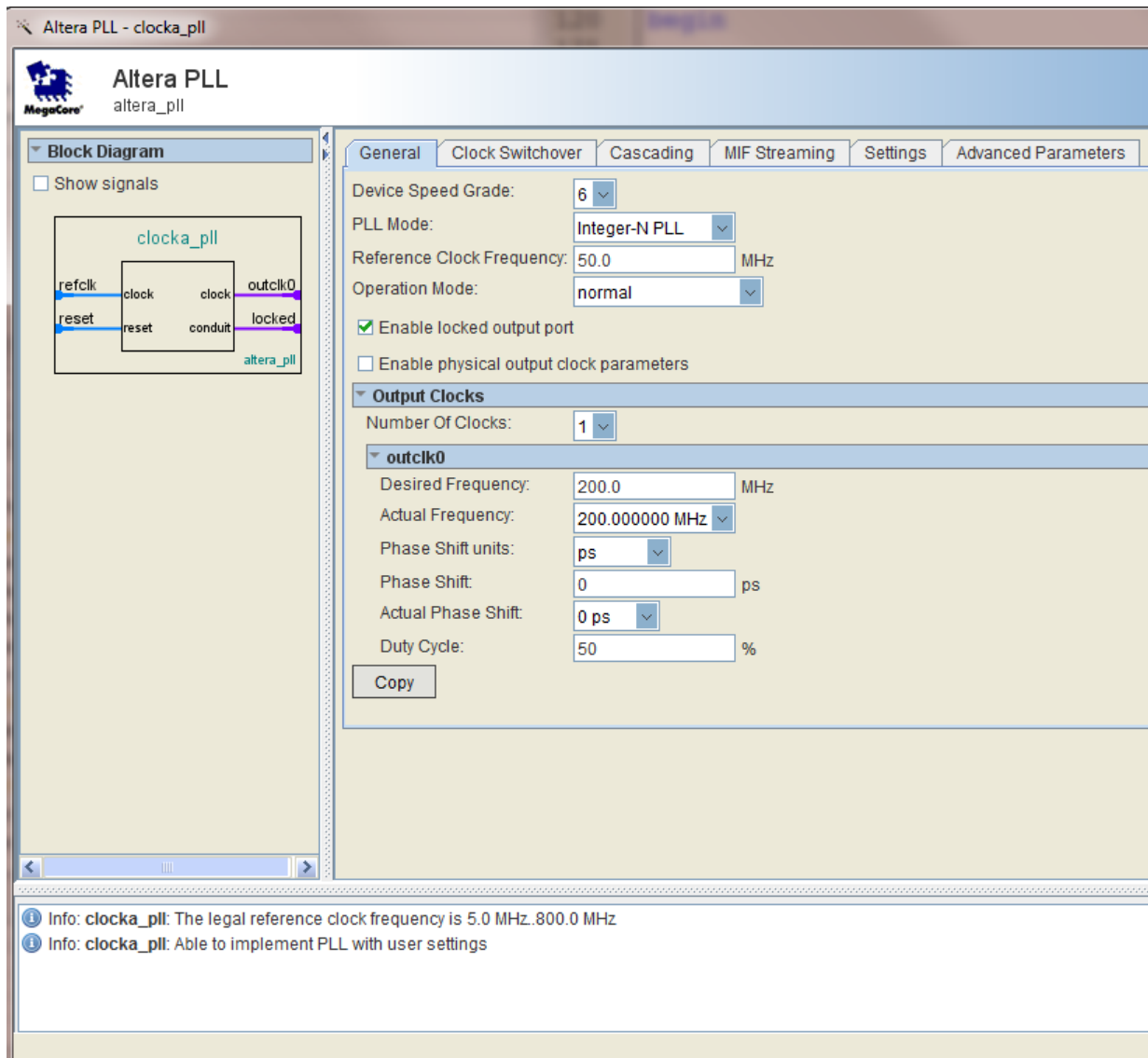
https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_fifo.pdf

Now we need the PLL's (Phase Locked Loops), back to the IP Catalog search, type "pll" and select the "Altera PLL" not the "Altera IOPLL". Give it the name clocka_pll (select VHDL) and place it into the ip_megawizards directory and click OK.

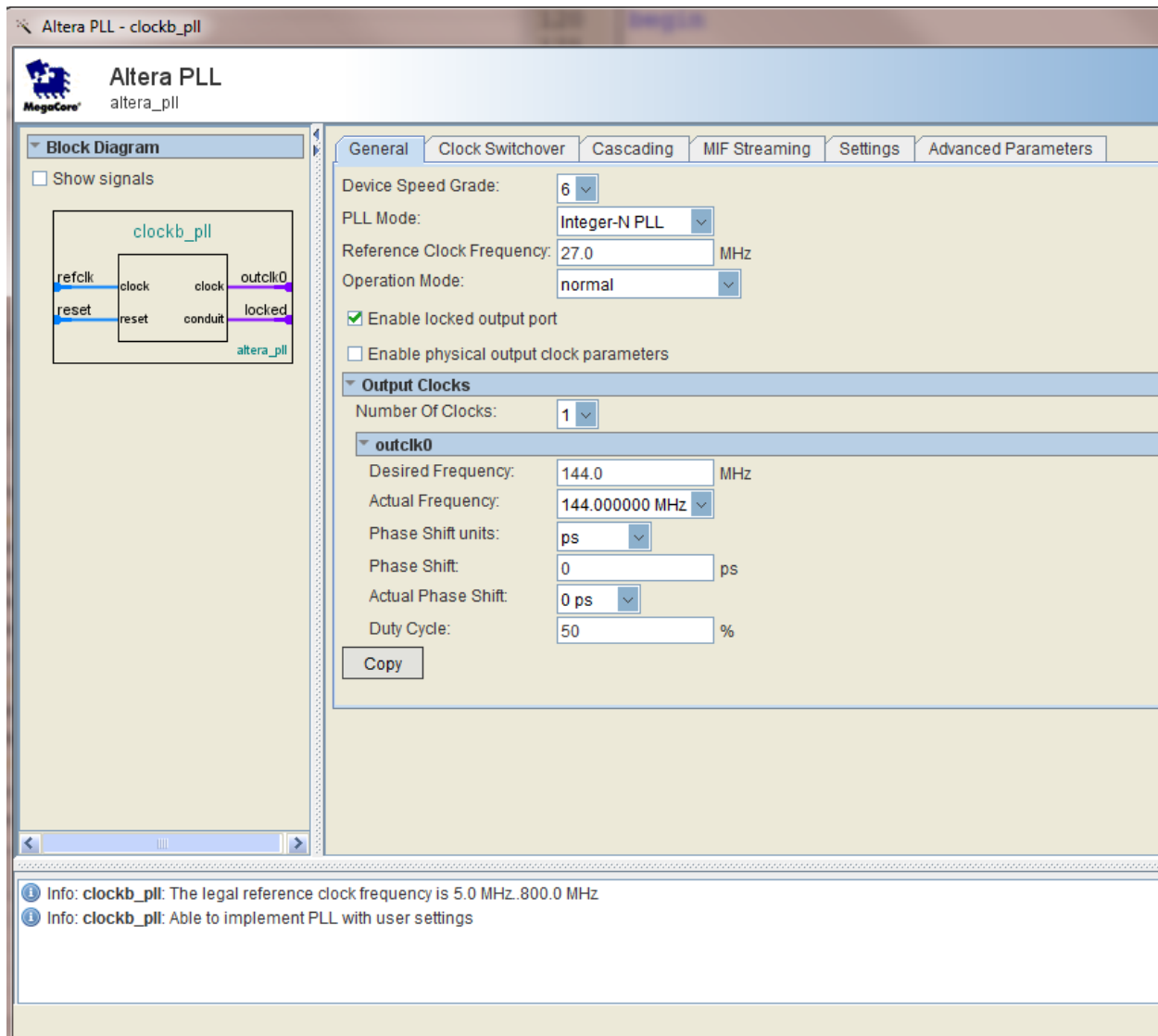


You should get the screen below. You need to change the input clock frequency to 50 Mhz since we will be driving this one from CLOCK_50. The defaults are fine for the rest. The operation mode, normal, is telling us that the output clock "outclk0" will be phase-aligned with the input clock.

Set the PLL up as shown below as we are boosting/increasing the 50Mhz input clock to 200Mhz clock with no phase shift to the output clock. Then hit finish on the lower right side (not shown below), and when generation is successful, click the exit button.



Now, repeat the megawizard flow to create a clockb_pll.vhd file. The input clock for this PLL will be 27 Mhz since it will be driven by TD_CLK27 pin which is 27Mhz on the board. Also place this megawizard design into the ip_megawizards directory.



27 Mhz is a clock commonly used for digital video and we are actually using the 27Mhz from the video interface on the board. Make a 144 Mhz clock for the output by typing in 144 in the Desired Frequency box. Click finish and then exit when generation is complete.

OK, that is it for the MegaWizard generation. For the rest of the modules, namely the Data Source, Data Sink, and the Wumpus are files already created and already part of the project. The basic architcture is that the Data Source will feed the Inbound FIFO until the FIFO has 64 words of data in it, representing a burst from a fast bus. Once the Inbound FIFO has 64 words, the Data Source module can start to pull data from the Inbound FIFO through the Wumpus module and put it in the Outbound FIFO. Once the Outbound FIFO has the whole burst (64 words), the Data Sink module can start to pull data from the Outbound FIFO until it is empty. So, we get a burst in, buffer it, process it, buffer the processed data, and then get rid of it.

A note about the Wumpus module; this module contains logic similar to what you might see in a

Block Cipher. This is a cryptography method for encrypting/decrypting data. Basically, the input data is put through a series of rotations, additions, or other logic operations such as XOR to scramble the data. Decryption is done by reversing the steps. If the block cipher is reasonably complex, guessing the needed operations by just looking at the encrypted data is difficult, even if you know what the input data is. In Wumpus, the blocks instantiated are Frazzle modules, each performs the same scrambling.

Now, with all these clock domains, the timing analysis can be difficult if you were to do this by hand. Timing analysis inside the FPGA or CPLD luckily is not done by hand. The Quartus tool provides the TimeQuest timing analyzer to do the static timing analysis for us. It will make sure that all the internal setup and hold times are met. In order for the TimeQuest analyzer to work, it needs the compiled design so it has a netlist and some information about the clocks. It gets the information about the clocks from a text file that contains a Synopsys Design Constraints (SDC) file which are tcl commands, or commonly called timing constraints. In this case the file is `de1_soc_top.sdc` located in the `timing_files` directory. Let's look at how we tell the analyzer about the clocks.

First we need to identify and specify the clocks going into the FPGA. This is done with these lines:

```
create_clock -period 20 -name {clock_50} [get_ports clock_50]
create_clock -period "27 MHz" -name {clock_27} [get_ports td_clk27]
```

This gives logical names (`clock_50` and `clock_27`) to the clocks (`clock_50` and `td_clk27`) and specifies the clock period in either nano-seconds, i.e. 20ns period for the 50Mhz input clock or as a frequency "27Mhz", and associates the clock with the right ports on the top level design. Ports are pins on the FPGA.

Since we are using PLL's in this design, which create clocks from the input clocks, we need to tell the analyzer to create the clock constraints for the Phase Locked Loop (PLL) generated clocks. This is easy we just need a line like this:

```
derive_pll_clocks
```


The last thing we need to do is to tell the analyzer if the clocks are related. Related clocks mean that they have a relationship to each other. For example, a 50 Mhz clock and a 100 Mhz clock have a constant phase relationship (if they come from the same oscillator, will not get into PPM (Part Per Million) differences on clocks in this class). A 50 Mhz clock and a 27 Mhz clock can have no constant phase relationship as they are not integer multiples of each other. If clocks are related, the analyzer can check the timing of transfers of data from one clock to the other, if they are not related, it can't. Well, it does but you will see setup times that don't make sense, but the timing analyzer will provide timing analysis based on the constraints provide. IP that is designed to cross clock domains like FIFOs will often cut timing paths internally as it was

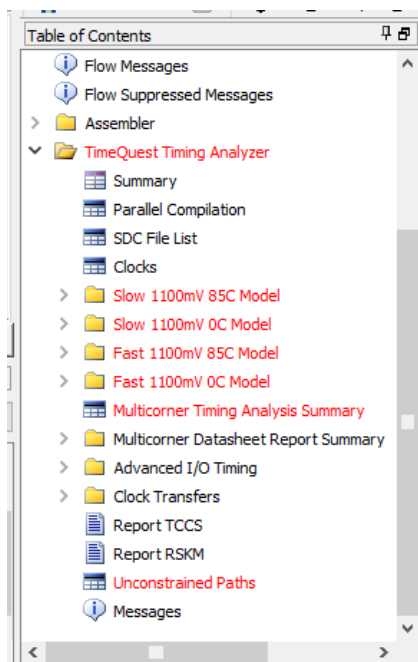
designed knowing the clocks are unrelated. That is not always the case and the Engineer may have to cut path internal to the FIFO. Cut paths means you are telling the timing analyzer not to worry about performing a timing analysis on that path, i.e. it can ignore the path.

The following lines are telling Quartus during synthesis and the Timquest tool during timing analysis that the clocks are asynchronous and unrelated and in our design we are managing the clock crossings. (Clock crossing is when you want to transfer a signal from 1 clock domain to another)

```
set_clock_groups -asynchronous -group [get_clocks {CLOCK_27}]
set_clock_groups -asynchronous -group [get_clocks {CLOCK_50}]
set_clock_groups -asynchronous \
-group [get_clocks {p1la|clocka_pll_inst|altera_pll_i|general[0].gp1l~PLL_OUTPUT_COUNTER|divclk}} \
-group [get_clocks {p1lb|clockb_pll_inst|altera_pll_i|general[0].gp1l~PLL_OUTPUT_COUNTER|divclk}}
```

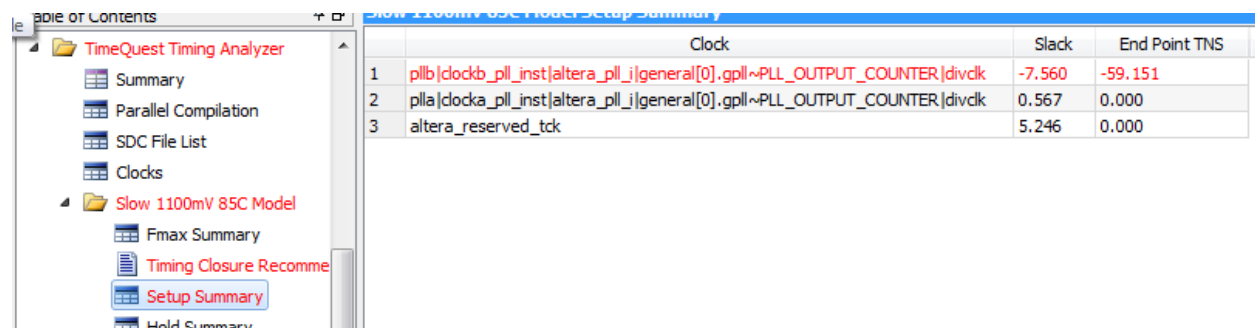
We rely on the FIFOs to provide safe transfer of data from these two exclusive clock groups.

The TimeQuest analyzer will run automatically during compilation. To get the results, expand the TimeQuest folder in the Compilation Report:  icon in the toolbar ribbon of Quartus.



A lot of information here, let's explain a few things. Note the Slow Model and Fast Model folders. These are the results of the analysis using the worst-case delays (Slow Model) and best-case delays (Fast Model). Observe that the Slow Model is in red, this means that there are timing violations in the slow model (as well as the fast model). If you have timing violations, your design may malfunction in wonderful and hard-to-find ways under certain conditions. Observe that under Slow Model, the Setup Summary is in red. If I click on it, I see the below

screen shot (you may have slightly different numbers):

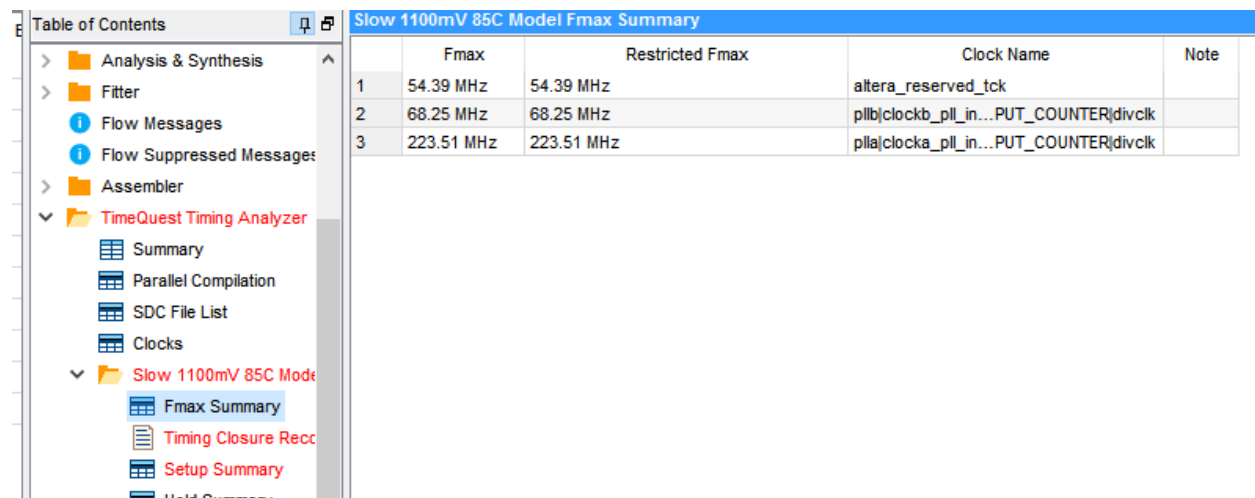


The screenshot shows the 'Table of Contents' on the left with 'Slow 1100mV 85C Model' expanded and 'Setup Summary' selected. The main window displays the 'Slow 1100mV 85C Model Setup Summary' table.

	Clock	Slack	End Point TNS
1	pllbclockb_pll_inst[altera_pll_i]general[0].gpll~PLL_OUTPUT_COUNTER divclk	-7.560	-59.151
2	plla clocka_pll_inst[altera_pll_i]general[0].gpll~PLL_OUTPUT_COUNTER divclk	0.567	0.000
3	altera_reserved_tck	5.246	0.000

This tells me that I have timing problems involving the clockb from the PLL, but not clocka. The positive slack is the amount of time that the data is MEETING the setup times. A negative slack means that it does not meet the setup times.

To solve timing violations, the easiest thing to do is to slow down the offending clock, but to what? It will tell us if we click on the Fmax Summary folder, observe:



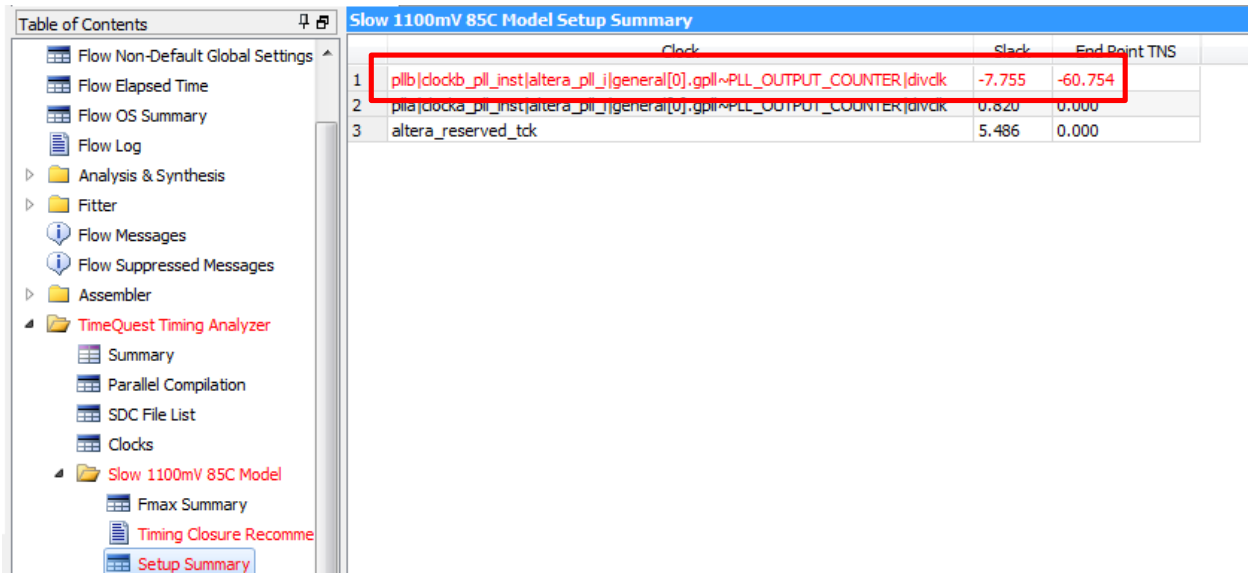
The screenshot shows the 'Table of Contents' on the left with 'Slow 1100mV 85C Model' expanded and 'Fmax Summary' selected. The main window displays the 'Slow 1100mV 85C Model Fmax Summary' table.

	Fmax	Restricted Fmax	Clock Name	Note
1	54.39 MHz	54.39 MHz	altera_reserved_tck	
2	68.25 MHz	68.25 MHz	pllbclockb_pll_in...PUT_COUNTER divclk	
3	223.51 MHz	223.51 MHz	plla clocka_pll_in...PUT_COUNTER divclk	

This tells us that the maximum frequency that clockb can run at is 68.25 Mhz. So if we go into the clockb_pll and change the clock to be less than that, we should be OK.

Often slowing the clock down is not an option – after all, it slows down the entire design. Another approach is to try to meet the timing by means of **pipelining**. To understand this, let's determine where the timing problem is. To accomplish this highlight and right click on the path boxed in red below and select “report timing .. (In Timequest UI) this will open up the

Timequest GUI, which is the static timing analyzer.



	Clock	Slack	End Point TNS
1	pll0[clckb_pll_inst[altera_pll_i]general[0].gppll~PLL_OUTPUT_COUNTER]divclk	-7.755	-60.754
2	pll0[clcka_pll_inst[altera_pll_i]general[0].gppll~PLL_OUTPUT_COUNTER]divclk	0.820	0.000
3	altera_reserved_tclk	5.486	0.000

The below window should show up and click on the “report timing box” with the defaults

Report Timing

×

Clocks

From dock:

To clock:

Targets

From: ...

Through: ...

To: ...

Analysis type

☒ Setup

☐ Hold

☐ Recovery

☐ Removal

Report number of paths:

10

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

☐ File name: ...

File options

☒ Overwrite

☐ Append

Open

☐ Console

Tcl command:

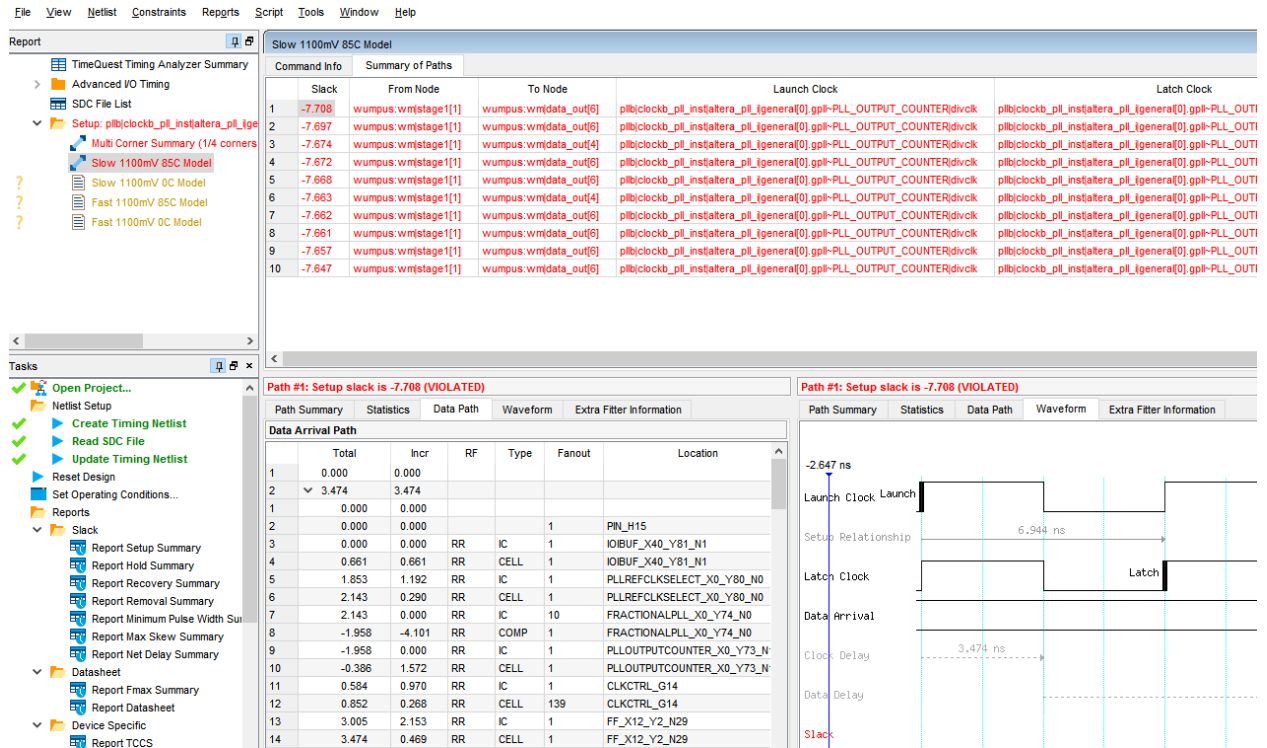
report_timing -setup -npaths 10 -detail full_path -panel_name (Report Timing)

Report Timing

Close

Help

Below is the Timequest GUI that launches.



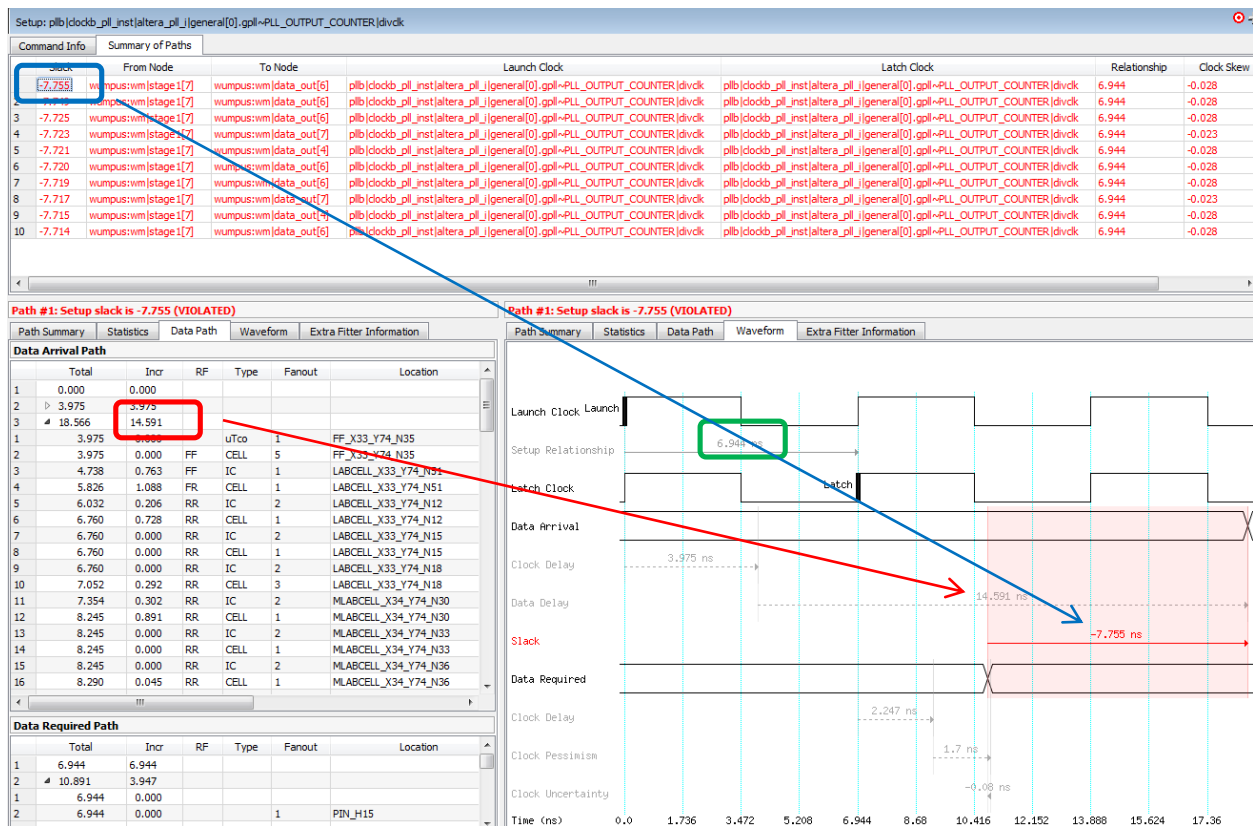
Zooming in below, you can see the “from node” and the “to node” and it indicates it is in the wumpus design. (you may have slightly different bits being the worst case path, but it should be the wumpus)

Setup: pll clockb_pll_inst altera_pll_i general[0].gpll~PLL_OUTPUT_COUNTER divclk				
Command Info		Summary of Paths		
	Slack	From Node	To Node	Launch Clock
1	-7.755	wumpus:wm stage1[7]	wumpus:wm data_out[6]	pll clockb_pll_inst altera_pll_i general[0].gpll~PLL_OUTPUT_COUNTER divclk
2	-7.749	wumpus:wm stage1[7]	wumpus:wm data_out[6]	pll clockb_pll_inst altera_pll_i general[0].gpll~PLL_OUTPUT_COUNTER divclk
3	-7.725	wumpus:wm stage1[7]	wumpus:wm data_out[6]	pll clockb_pll_inst altera_pll_i general[0].gpll~PLL_OUTPUT_COUNTER divclk
4	-7.723	wumpus:wm stage1[7]	wumpus:wm data_out[7]	pll clockb_pll_inst altera_pll_i general[0].gpll~PLL_OUTPUT_COUNTER divclk
5	-7.721	wumpus:wm stage1[7]	wumpus:wm data_out[4]	pll clockb_pll_inst altera_pll_i general[0].gpll~PLL_OUTPUT_COUNTER divclk
6	-7.720	wumpus:wm stage1[7]	wumpus:wm data_out[6]	pll clockb_pll_inst altera_pll_i general[0].gpll~PLL_OUTPUT_COUNTER divclk
7	-7.719	wumpus:wm stage1[7]	wumpus:wm data_out[6]	pll clockb_pll_inst altera_pll_i general[0].gpll~PLL_OUTPUT_COUNTER divclk
8	-7.717	wumpus:wm stage1[7]	wumpus:wm data_out[7]	pll clockb_pll_inst altera_pll_i general[0].gpll~PLL_OUTPUT_COUNTER divclk
9	-7.715	wumpus:wm stage1[7]	wumpus:wm data_out[4]	pll clockb_pll_inst altera_pll_i general[0].gpll~PLL_OUTPUT_COUNTER divclk
10	-7.714	wumpus:wm stage1[7]	wumpus:wm data_out[6]	pll clockb_pll_inst altera_pll_i general[0].gpll~PLL_OUTPUT_COUNTER divclk

Path #1: Setup slack is -7.755 (VIOLATED)

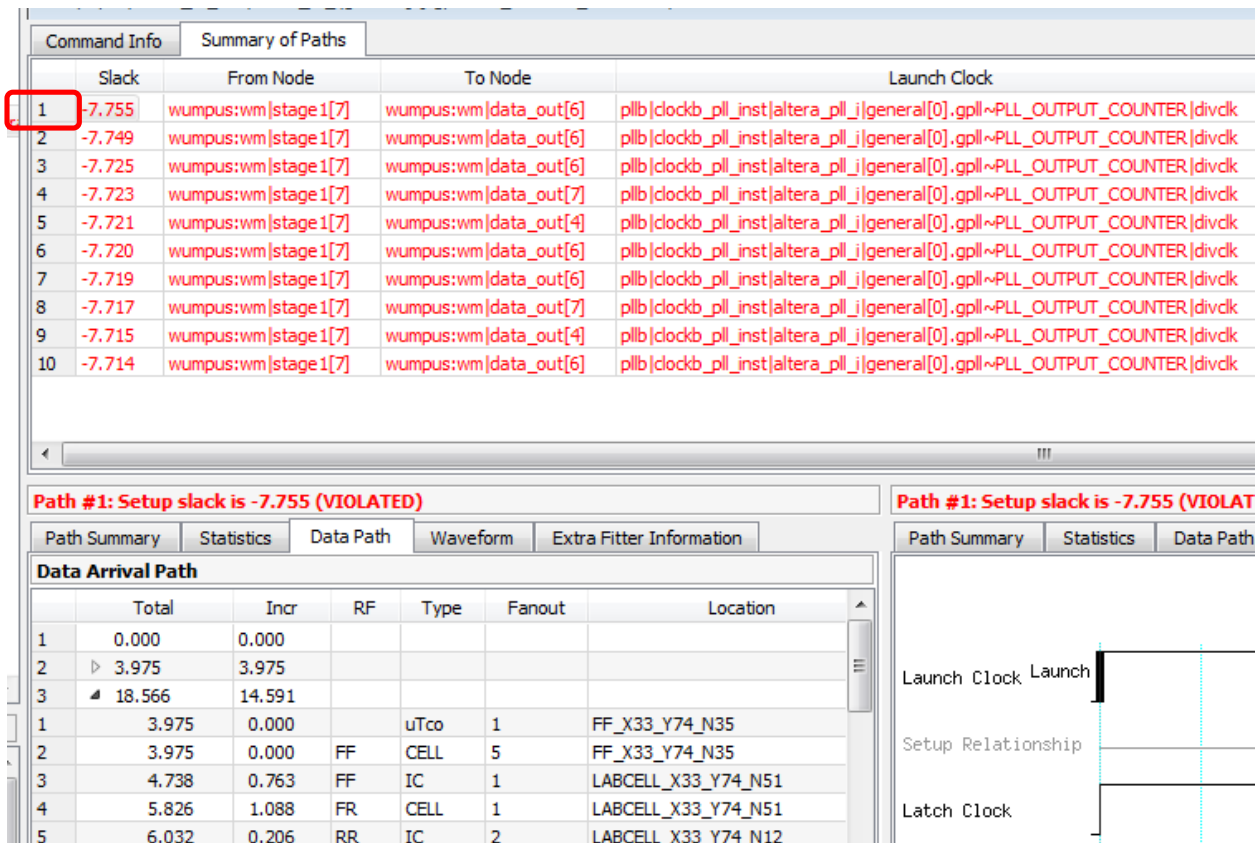
Path Summary		Statistics	Data Path	Waveform	Extra Fitter Information	
Data Arrival Path						
	Total	Incr	RF	Type	Fanout	Location
1	0.000	0.000				launch edge time
2	▷ 3.975	3.975				clock path
3	◀ 18.566	14.591				data path
1	3.975	0.000		uTco	1	FF_X33_Y74_N35
2	3.975	0.000	FF	CELL	5	FF_X33_Y74_N35
3	4.738	0.763	FF	IC	1	LABCELL_X33_Y74_N51
4	5.826	1.088	FR	CELL	1	LABCELL_X33_Y74_N51
5	6.032	0.206	RR	IC	2	LABCELL_X33_Y74_N12
6	6.760	0.728	RR	CELL	1	LABCELL_X33_Y74_N12
7	6.760	0.000	RR	IC	2	LABCELL_X33_Y74_N15
8	6.760	0.000	RR	CELL	1	LABCELL_X33_Y74_N15
9	6.760	0.000	RR	IC	2	LABCELL_X33_Y74_N18
10	7.052	0.292	RR	CELL	3	LABCELL_X33_Y74_N18
11	7.354	0.302	RR	IC	2	MLABCELL_X34_Y74_N30
12	8.245	0.891	RR	CELL	1	MLABCELL_X34_Y74_N30
13	8.245	0.000	RR	IC	2	MLABCELL_X34_Y74_N33
14	8.245	0.000	RR	CELL	1	MLABCELL_X34_Y74_N33
15	8.245	0.000	RR	IC	2	MLABCELL_X34_Y74_N36
16	8.290	0.045	RR	CELL	1	MLABCELL_X34_Y74_N36

And looking further (see below), our clock cycle/setup relationship is 6.994ns (Green box) our 144Mhz domain, and our data delay from register to register is 14.991ns (Red Box). We are violating the timing, we have a negative slack of -7.755 (Blue Box)



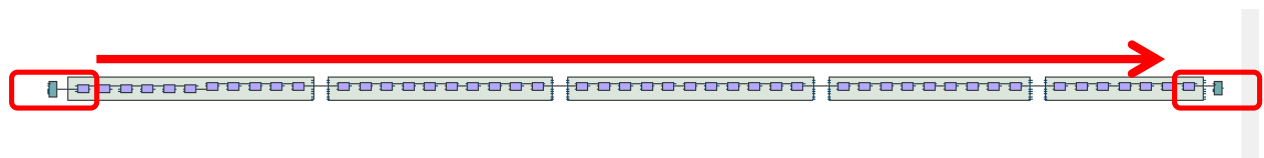
There are multiple netlist viewers in Quartus; RTL, Technology Map Viewer, etc. From Timequest you can locate the offending path in the Technology Map Viewer. The Technology Map Viewer shows the path after synthesis and place and route, it is representing the actual implementation of logic into the logic elements in the FPGA.

Highlight the “1” (shown below, the Red Circle) and right click and select “locate path” and



Select the "Locate in Technology Map Viewer".

Doing some zooming...



The left side is the source register (launch register) and the right side is the destination register (latch register). It may not be obvious, but this *is* many layers of logic between the registers. This is all combinatorial logic.

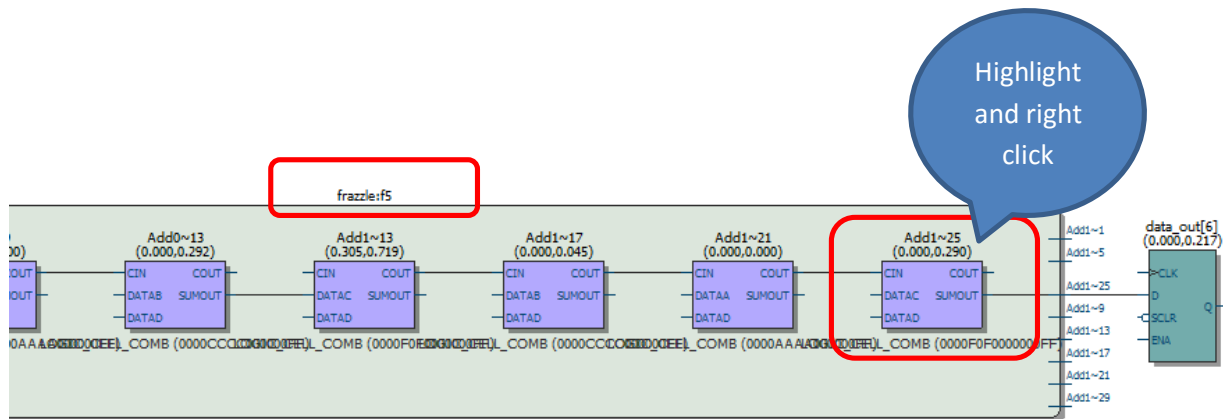
You can also see the levels of logic in Timequest as well. Back in Timquest, the screen shot below, you can also see that the long path is in Wumpus, and heading down are all the layers of logic and it looks like a string of "adds".

	Slack	From Node	To Node	Launch Clock	
1	-7.755	wumpus:wm stage1[7]	wumpus:wm data_out[6]	pll clockb_pll_inst altera_pll_i general[0].gp ~PLL_OUTPUT_COUNTER divclk	pll
2	-7.749	wumpus:wm stage1[7]	wumpus:wm data_out[6]	pll clockb_pll_inst altera_pll_i general[0].gp ~PLL_OUTPUT_COUNTER divclk	pll
3	-7.725	wumpus:wm stage1[7]	wumpus:wm data_out[6]	pll clockb_pll_inst altera_pll_i general[0].gp ~PLL_OUTPUT_COUNTER divclk	pll
4	-7.723	wumpus:wm stage1[7]	wumpus:wm data_out[7]	pll clockb_pll_inst altera_pll_i general[0].gp ~PLL_OUTPUT_COUNTER divclk	pll

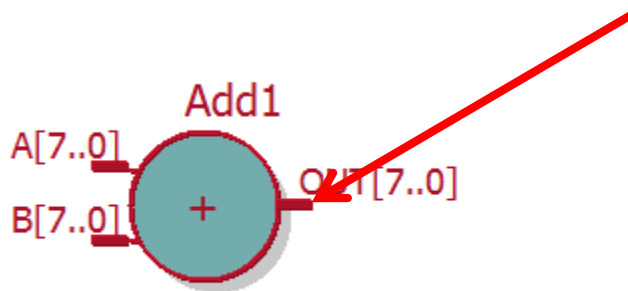
Path #1: Setup slack is -7.755 (VIOLATED)

Path Summary	Statistics	Data Path	Waveform	Extra Fitter Information			
Data Arrival Path							
	Total	Incr	RF	Type	Fanout	Location	Element
1	0.000	0.000					launch edge time
2	3.975	3.975					clock path
3	18.566	14.591					data path
1	3.975	0.000		uTco	1	FF_X33_Y74_N35	wumpus:wm stage1[7]
2	3.975	0.000	FF	CELL	5	FF_X33_Y74_N35	wm stage1[7] q
3	4.738	0.763	FF	IC	1	LABCELL_X33_Y74_N51	wm f1 Add0~17 dataa
4	5.826	1.088	FR	CELL	1	LABCELL_X33_Y74_N51	wm f1 Add0~17 sumout
5	6.032	0.206	RR	IC	2	LABCELL_X33_Y74_N12	wm f1 Add1~17 datac
6	6.760	0.728	RR	CELL	1	LABCELL_X33_Y74_N12	wm f1 Add1~17 cout
7	6.760	0.000	RR	IC	2	LABCELL_X33_Y74_N15	wm f1 Add1~21 cin
8	6.760	0.000	RR	CELL	1	LABCELL_X33_Y74_N15	wm f1 Add1~21 cout
9	6.760	0.000	RR	IC	2	LABCELL_X33_Y74_N18	wm f1 Add1~25 cin
10	7.052	0.292	RR	CELL	3	LABCELL_X33_Y74_N18	wm f1 Add1~25 sumout
11	7.354	0.302	RR	IC	2	MLABCELL_X34_Y74_N30	wm f2 Add0~21 datab
12	8.245	0.891	RR	CELL	1	MLABCELL_X34_Y74_N30	wm f2 Add0~21 cout
13	8.245	0.000	RR	IC	2	MLABCELL_X34_Y74_N33	wm f2 Add0~25 cin
14	8.245	0.000	RR	CELL	1	MLABCELL_X34_Y74_N33	wm f2 Add0~25 cout
15	8.245	0.000	RR	IC	2	MLABCELL_X34_Y74_N36	wm f2 Add0~29 cin
16	8.290	0.045	RR	CELL	1	MLABCELL_X34_Y74_N36	wm f2 Add0~29 cout
17	8.290	0.000	RR	IC	2	MLABCELL_X34_Y74_N39	wm f2 Add0~1 cin
18	8.290	0.000	RR	CELL	1	MLABCELL_X34_Y74_N39	wm f2 Add0~1 cout
19	8.290	0.000	RR	IC	2	MLABCELL_X34_Y74_N42	wm f2 Add0~5 cin
20	8.589	0.299	RR	CELL	1	MLABCELL_X34_Y74_N42	wm f2 Add0~5 sumout
21	8.791	0.202	RR	IC	2	MLABCELL_X34_Y74_N3	wm f2 Add1~5 datac
22	9.554	0.763	RR	CELL	1	MLABCELL_X34_Y74_N3	wm f2 Add1~5 cout

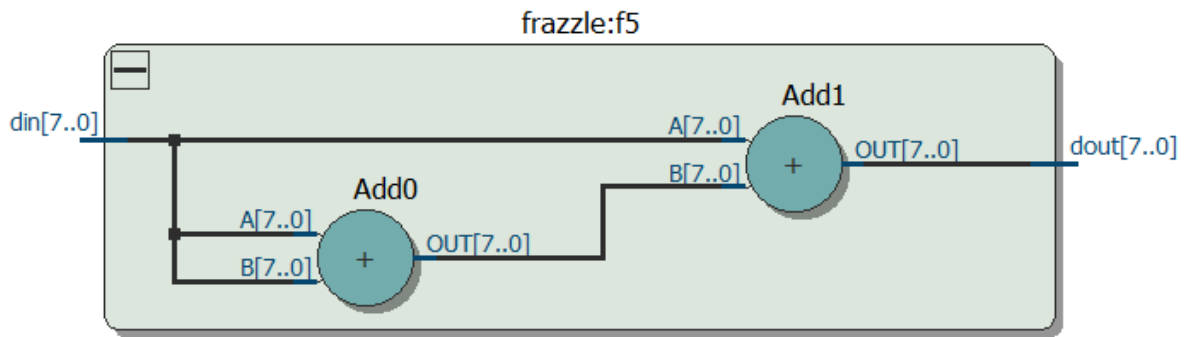
Going back to the Technology Map Viewer and zooming in, we can see that there is a frazzle block in the path. What is in Frazzle? It looks like a series of Adds. We can look at the RTL viewer (Register Transfer Level) netlist viewer. The RTL viewer is a schematic view of your VHDL design without optimizations. It represents the logic that the VHDL is creating. There are many ways to invoke the RTL viewer. This is one flow; select/highlight one of the add blocks, and then right click and select “locate node” and “locate in RTL viewer”



The RTL viewer should open up and show an Add, you can expand the view by double clicking on the connection points (the arrow below is pointing a connection point).

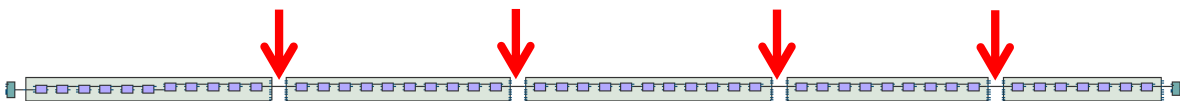


You should be able to build out the RTL viewer to something similar to below by clicking on the connection points. You can see, Frazzle is a series of “adds”. It is very, very important to understand when writing VHDL what type of logic you are actually creating. Did I mention this was important?



So how can we speed this up?

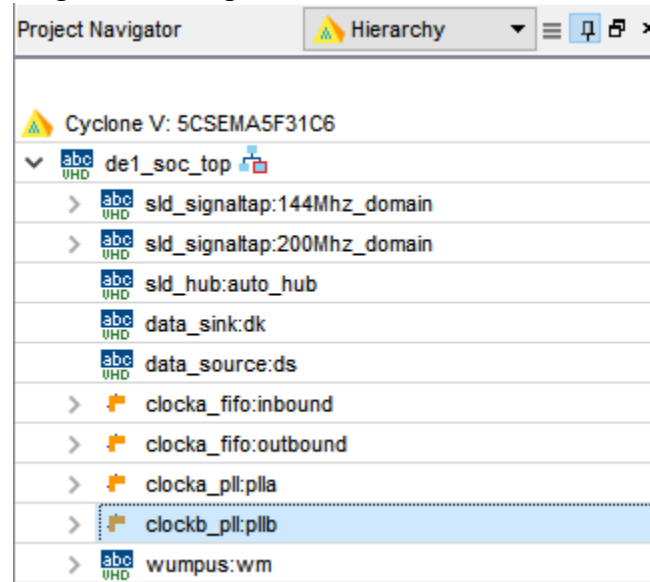
Going back to the Technology Map Viewer, we can put in registers (pipelining registers) between the frazzle blocks, a good location is shown below by the red arrows. This will add 4 register to the data path and will take 4 extra rising edges of the clock for the data to get through the Wumpus/Frazzle blocks. This is called Latency and after you have waited the 4 extra clock cycles, then every clock cycle after that you have new data coming out and at your higher data rate/frequency. This is a very common practice to meet timing. Now if you have a FSM (Finite State Machine) in the design, you may have to adjust the control to accommodate for the register pipelining.



Lab Report:

1. Go through the steps at the beginning of this tutorial to create the PLL's and FIFO's.
2. Draw up a block diagram of the design with the PLLs, logic blocks and FIFOs (you can do this by hand and scan in)
3. Compile the design in Quartus, run the simulation, run.do (located in the simulation directory), observe how the data flows.
4. Observe the timing analysis in Quartus/Timequest
 - a. Find the Fmax that will fix the ClockB violations (i.e what is the new clock frequency we need to have the design meet timing)
 - b. change the appropriate PLL to a frequency that will work (you need to regenerate the PLL) to re-open the PLL, go to the project navigator and select

the Hierarchy and then double click on the appropriate PLL, this will re-launch the megawizard/megacore



- c.
- d. Recompile in Quartus and provide a screen shot that shows the timing is OK with the new frequency.
5. Undo what you just did on the PLL (i.e. set the PLL back to 144Mhz)
 - a. Modify Wumpus to add pipeline registers between the Frazzle blocks to pipeline the design to meet timing. Compile in Quartus and check the timing.
 - b. Report if the timing is good, and what the new Fmax for clock B is.
6. What happens when you halt (key1)?
 - a. You will need to modify the testbench to see the behavior.
 - b. Show in simulation and describe in words.
7. What happens when you stall (key2)?
 - a. You will need to modify the testbench to see the behavior.
 - b. Show in simulation and describe in words.
8. Your verification should show me that you understand the data flow through the design, add notations to Modelsim waveform screen shots that show you understand the design / dataflow.