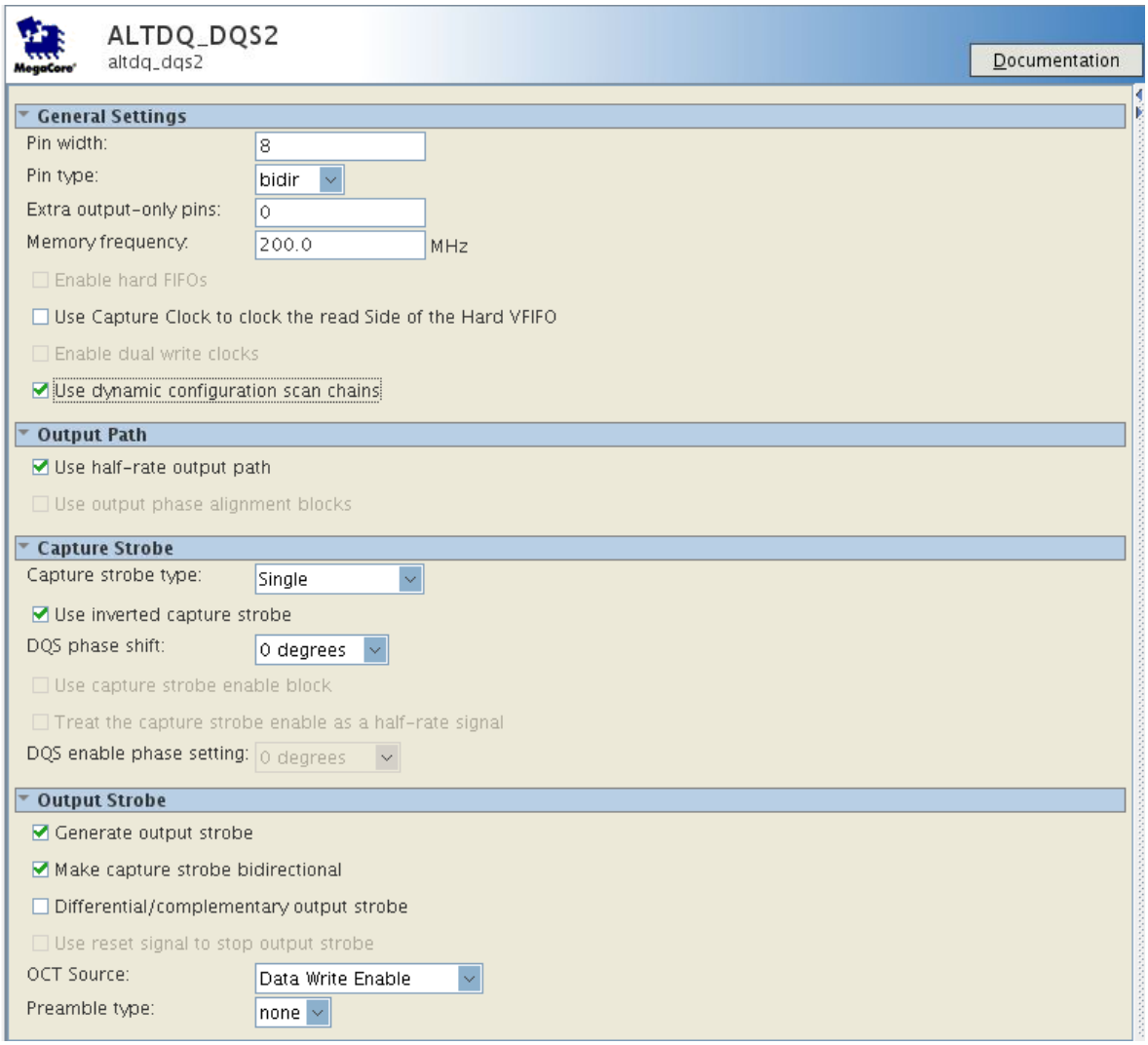
ALTDQ_DQS2 Example Design User Guide (Arria V)

To create a design related to ALTDQ_DQS2, follow instructions below. Alternatively, you may utilize the attached example design. (The example design is based on QP 15.1. Tested working with Modelsimse 10.3d, and vcsmx/2014.12-SP1)

1. File>New Project Wizard
2. In New Project Wizard, click Next.
3. Then, provide the working directory as well as the name of the project. For example, use “top” as the project name. Click Next.
4. Then click Next again.
5. Select ArriaV as the Device Family. Select targeted device part number if applicable. Click Next.
6. Enter the appropriate EDA Tool Settings. For example, in the Simulation section, select Modelsim as tool name. Click Next.
7. Click Finish.

Instantiating ALTDQ_DQS2

1. Go to Tools>IP Catalog.
2. In the IP Catalog Menu, select Basic Functions> I/O> ALTDQ_DQS2 v15.1. Enter the IP variation file name and select the IP variation file type. Example of the output file is “bidir_hardFIFO_dqdqs2”. Then click OK.
3. Follow settings below. Then click Finish. Then click Exit when done.

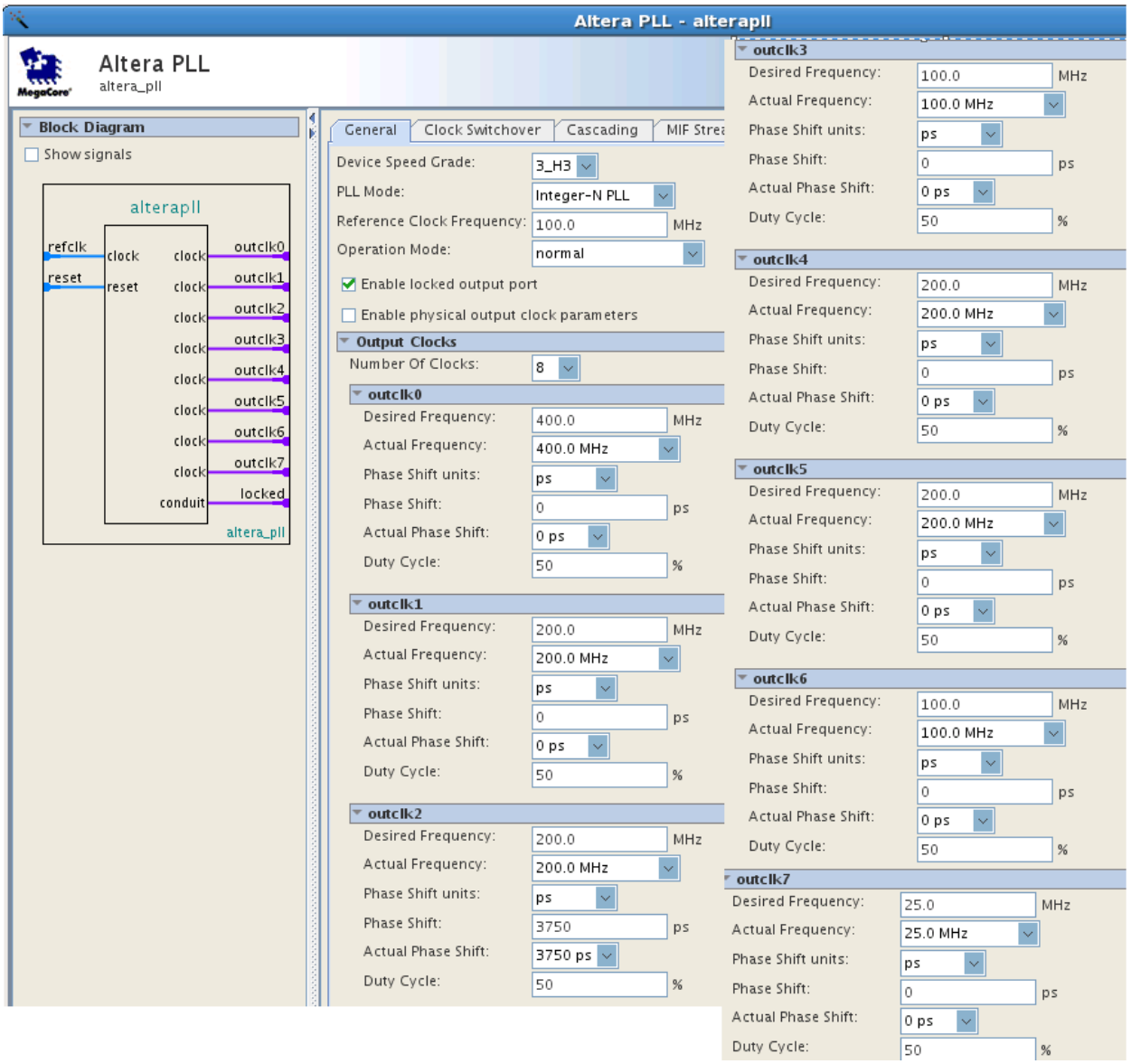


Note 1: If user requires bidirectional strobe, DQS phase shift must be set to 0 degrees to bypass the DQS delay chain. Please refer to the KDB link for details.

Note 2: To edit the ALTDQ_DQS2 Megafunction settings, user needs to execute as follows: File>Open>Select "bidir_hardFIFO_dqdqs2.v"

Instantiating Altera PLL

1. Go to Tools>IP Catalog
2. In the IP Catalog Menu, select Basic Functions>Clocks;PLLs and Resets>PLL>Altera PLL. Enter the IP variation file name and select the IP variation file type. Example of the output file is "alterapll". Then click OKb.
3. Follow settings below. Then click Finish. Then click Exit when done.



4. Information of the clock settings are as below (users may merge the similar frequency counters in their design, or the fitter will handle this automatically):

PLL mode : Normal

outclk_0 = 400 MHz (used as 2x frequency if necessary)

outclk_1 = 200 MHz (used as strobe/dqs clock)

outclk_2 = 200MHz, 270degree phase shifted (used as data/dq clock)

outclk_3 = 100MHz (used as half rate clock)

outclk_4 = 200MHz (used to drive the ALTDLL. Note that 300MHz is ALTDLL's minimum frequency for StratixV devices)1

outclk_5 = 200MHz (used to drive the Full Rate core clock)

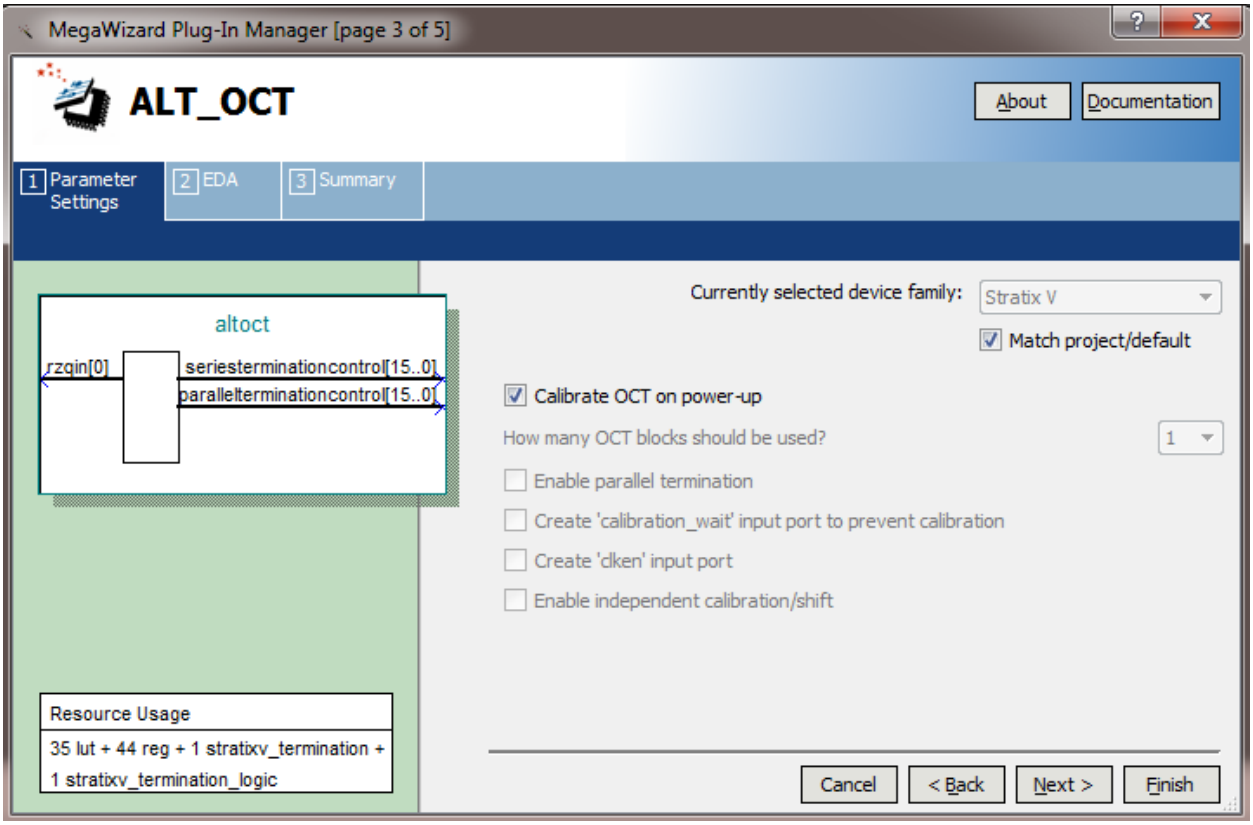
outclk_6 = 100 MHz (used to drive the Half Rate core clock)

outclk_7 = 25 MHz (used as config_clk)

Note: If the memory frequency is less than the ALTDLL minimum frequency, the ALTDLL needs to be driven at 2x or 4x of the memory frequency. Relatively, the DQS phase settings shrinks as well. For more information, refer to the EMI Handbook Chapter and Device Datasheet.

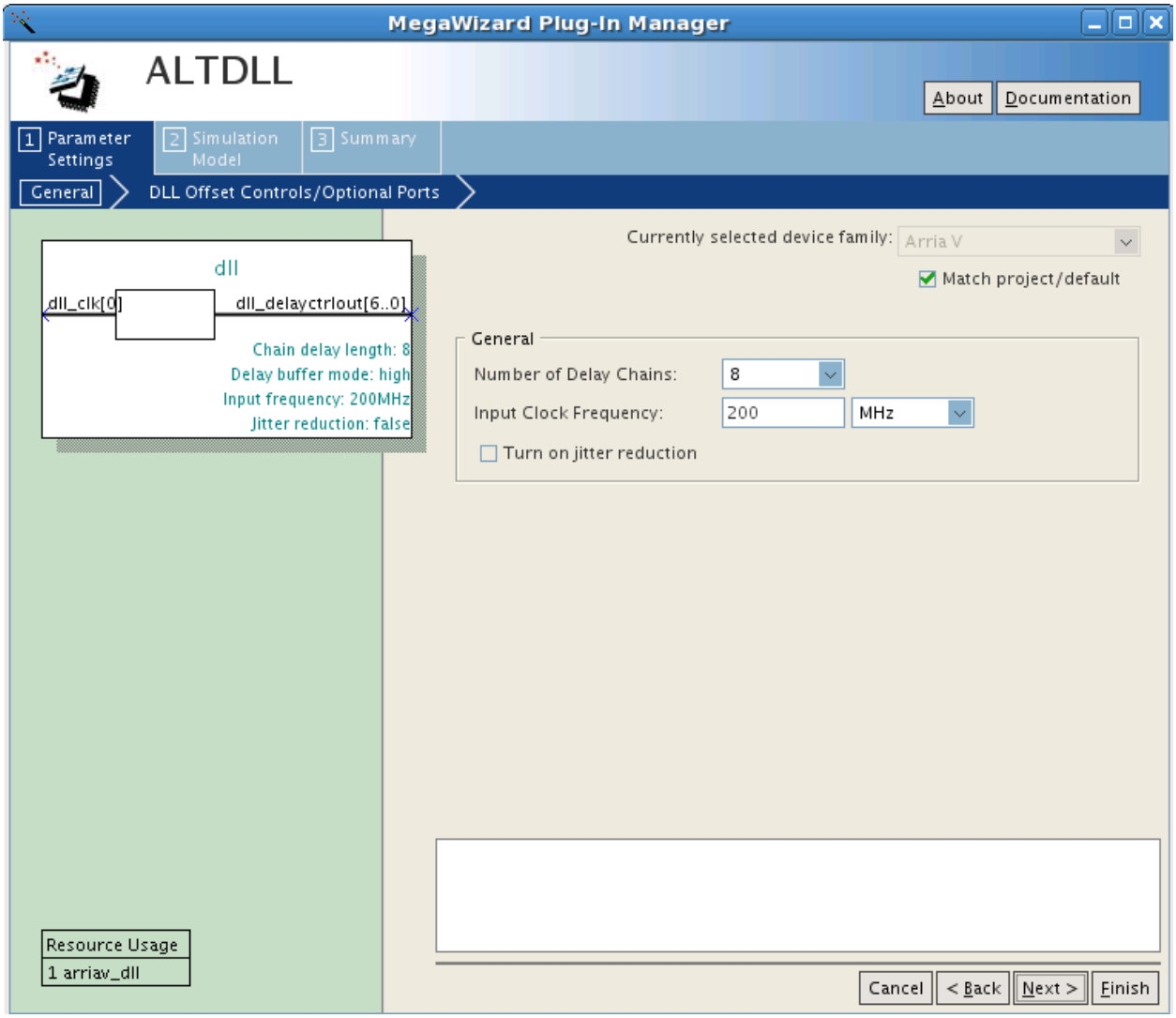
Instantiating ALTOCT

1. Go to Tools>IP Catalog
2. In the IP Catalog Menu, select Basic Functions>I/O>ALTOCT. Enter the IP variation file name and select the IP variation file type. Example of the output file is “altoct”. Then click Next.
3. Follow settings below. Then click Finish. Then click Finish when done.



Instantiating ALTDLL

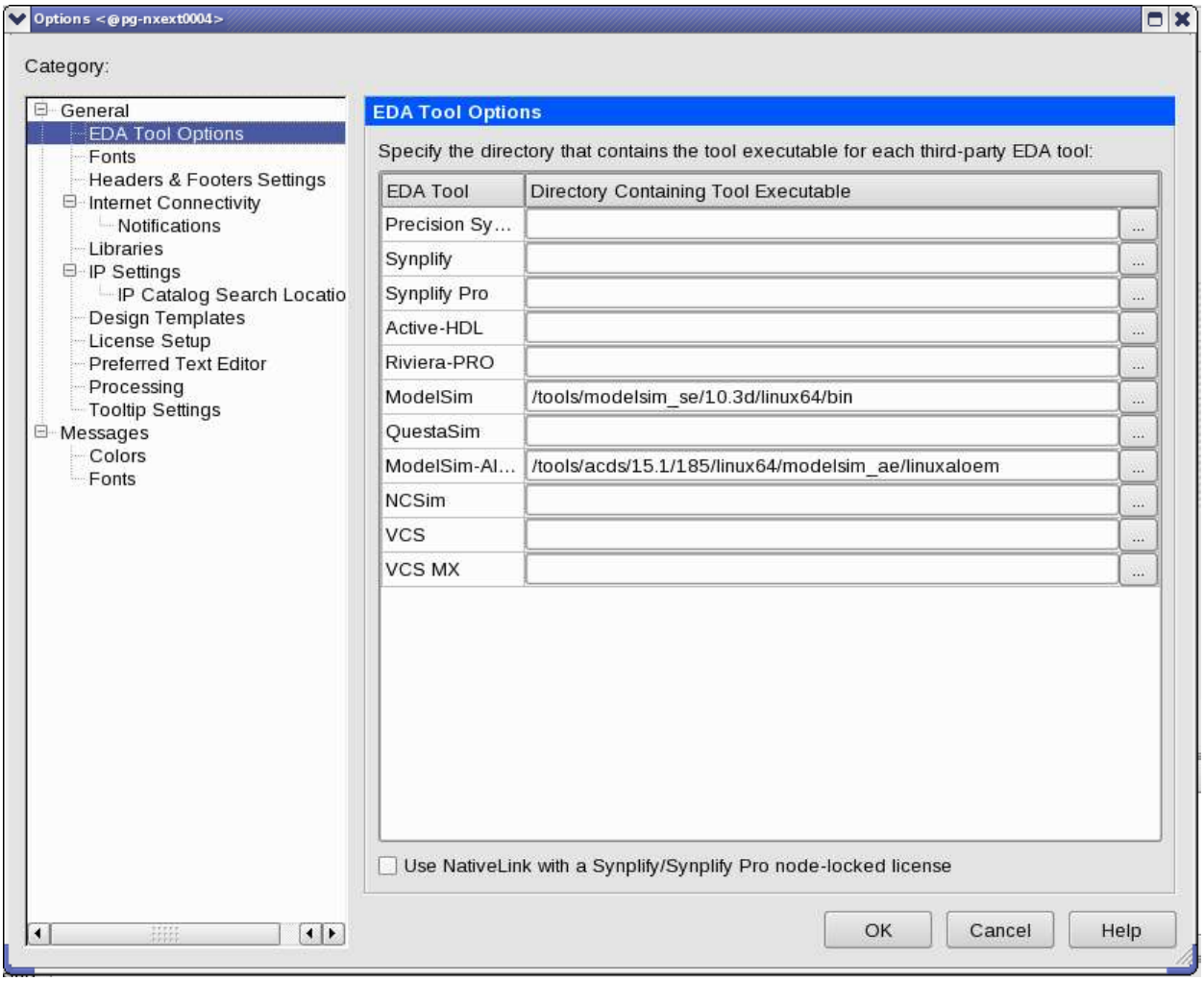
1. Go to Tools>IP Catalog
2. In the IP Catalog Menu, select Basic Functions>I/O>ALTDLL. Enter the IP variation file name and select the IP variation file type. Example of the output file is “altdll”. Then click Next.
3. Follow settings below. Then click Finish. Then click Finish when done.



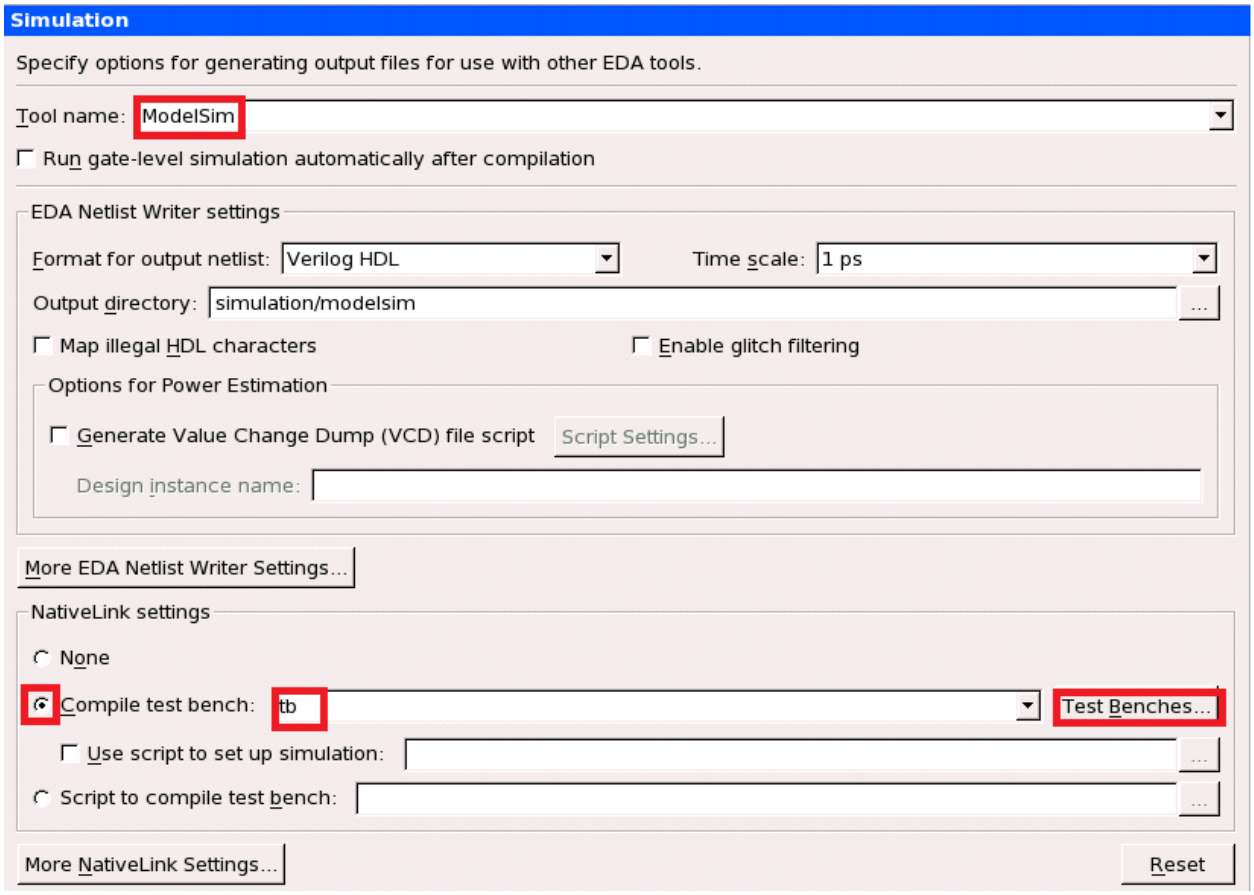
Connect all megawizards together. In this example design, this is done in top.v. An example of dynamic configuration controller logic (config_controller_acv.sv) is instantiated in top.v as well.

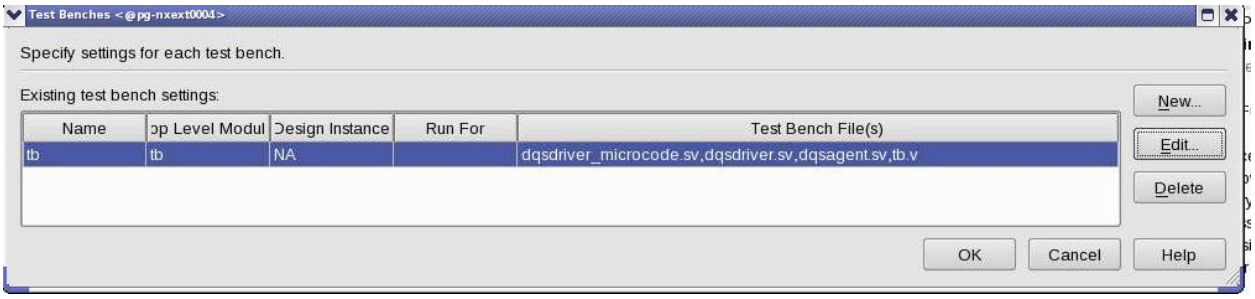
To setup the Nativelink and Simulation settings, follow the instructions below:

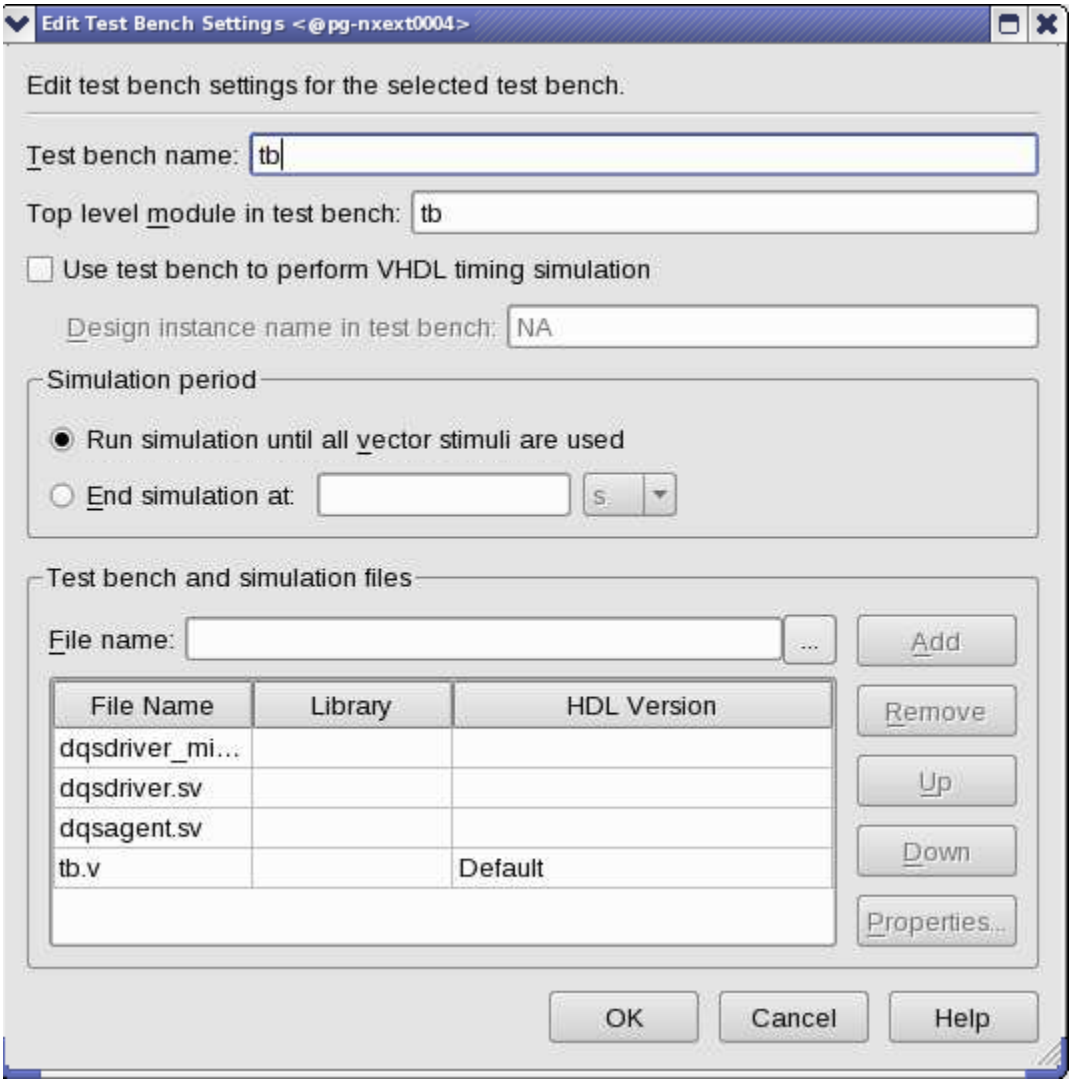
1. In Quartus Prime Main Window, Go to Tools>Options. An example of settings are shown as below.



2. In Quartus Prime Main Window, go to Assignments>Settings>EDA Tool Settings>Simulation. Fill in the appropriate NativeLink settings. An example of settings is shown as below. In this example design, a test bench (tb.v) is provided together with other supporting files.





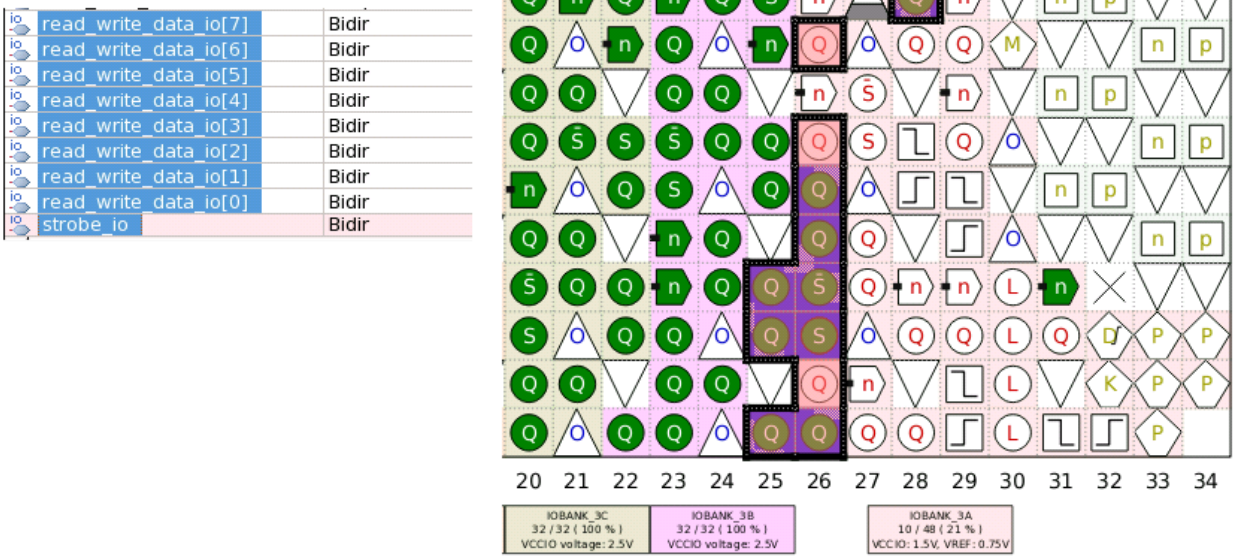


3. Run Analysis and Synthesis.
4. To see simulation results, go to Tools>Run Simulation Tool>RTL Simulation. See “Understanding Simulation Results” Section for more technical information.
    a. For successful simulation, user may need to manually change “alterapll.v” to “alterapll.vo” in the auto-generated “top_run_msim_rtl_verilog.do” file.

5. Prior to running the fitter, ensure that following settings are done in the Assignment Editor.
    a. I/O Standard
    b. Input Termination
    c. Output Termination
    d. DQ Group
    e. Location assignment for strobe pin – this helps the fitter to fit the related DQ pins in the appropriate I/O sub-banks. Users can then back-annotate the locations if desired.

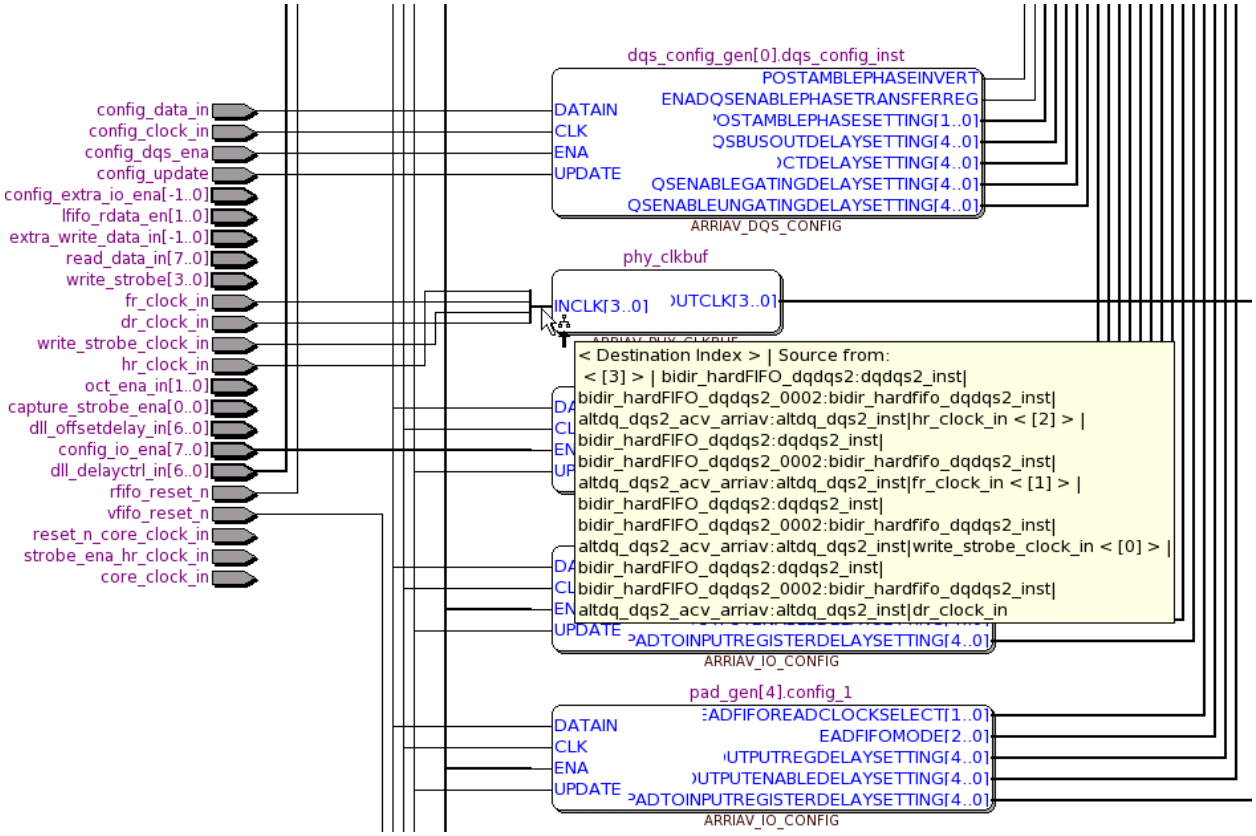
An example setting in Assignment Editor, as well as the resulted Pin Planner is shown as below:

	ati	From	To	Assignment Name	Value	Enabled	Entity
1	✓		*alterapll_0002*|...l:altera_pll_i*|*	PLL Compensation Mode	Normal	Yes	top
2	✓		*alterapll_0002*|...l:altera_pll_i*|*	PLL Automatic Self-Reset	Off	Yes	top
3	✓		*alterapll_0002*|...l:altera_pll_i*|*	PLL Bandwidth Preset	Auto	Yes	top
4	✓		read_write_data_io[*]	I/O Standard	SSTL-15 Class I	Yes	
5	✓		read_write_data_io[*]	Input Termination	Parallel 50 Ohm with Calibration	Yes	top
6	✓		read_write_data_io[*]	Output Termination	Series 50 Ohm with Calibration	Yes	top
7	✓		strobe_io	Location	PIN_AM26	Yes	
8	✓		strobe_io	I/O Standard	SSTL-15 Class I	Yes	
9	✓		strobe_io	Output Termination	Series 50 Ohm without Calibration	Yes	top
10	✓		strobe_io	Input Termination	Parallel 50 Ohm with Calibration	Yes	top
11	✓	strobe_io	read_write_data_io[*]	DQ Group	9	Yes	top
12		<<new>>	<<new>>	<<new>>			



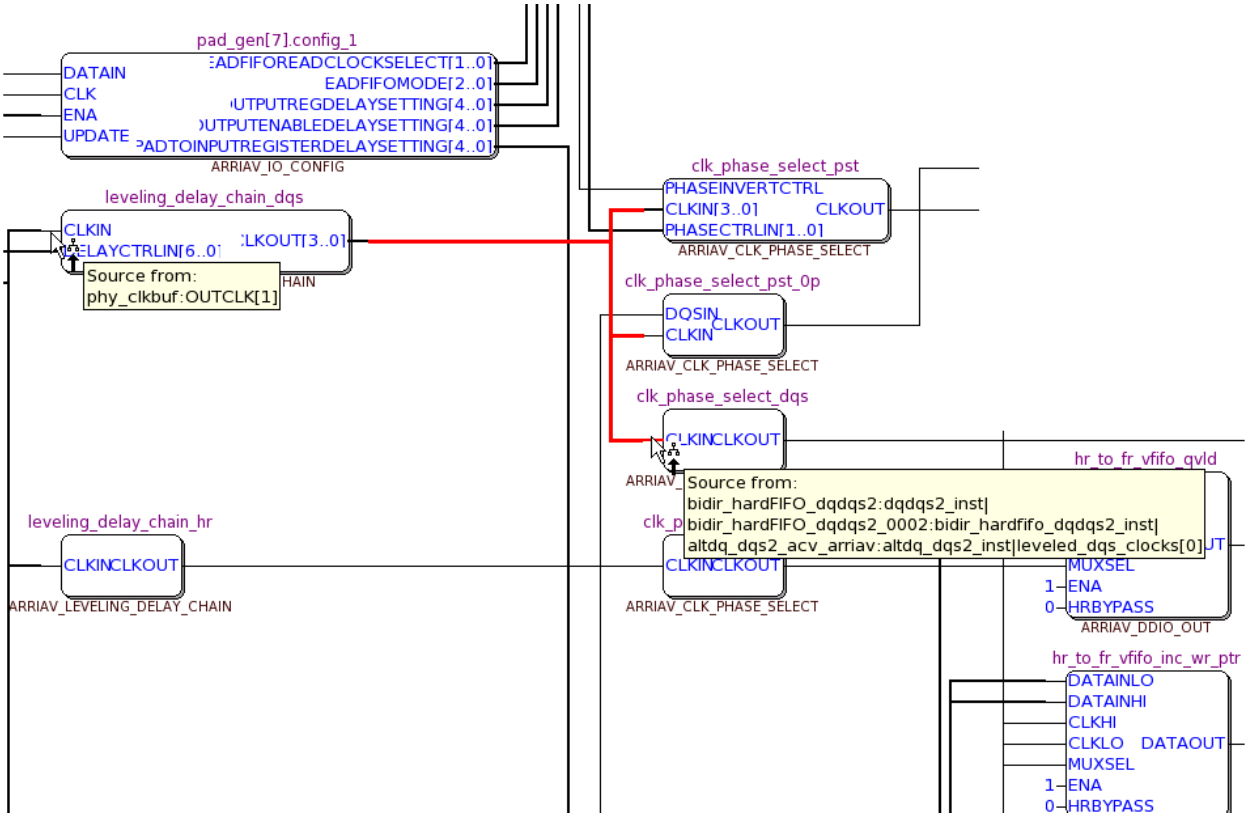
6. Run Fitter, Timing Analysis and Assembler. A SDC example (top.sdc) is included in the example design. See “SDC Brief Walkthrough” Section for more elaboration on the SDC Constraints example.

Note that in AV and CV, PHYCLK is used by default, instead of going through global clock routing in SV case.



To know which outclk of the PHYCLK is driving the DQS (strobe_io), trace it in the Netlist Viewer. See example below. This results in SDC command below:

create_generated_clock -name dqs_out -source [get_pins { dqdqs2_inst|bidir_hardfifo_dqdqs2_inst|altdq_dqs2_inst|phy_clkbuf|outclk[1] }] -phase 0 [get_ports {strobe_io}] -add

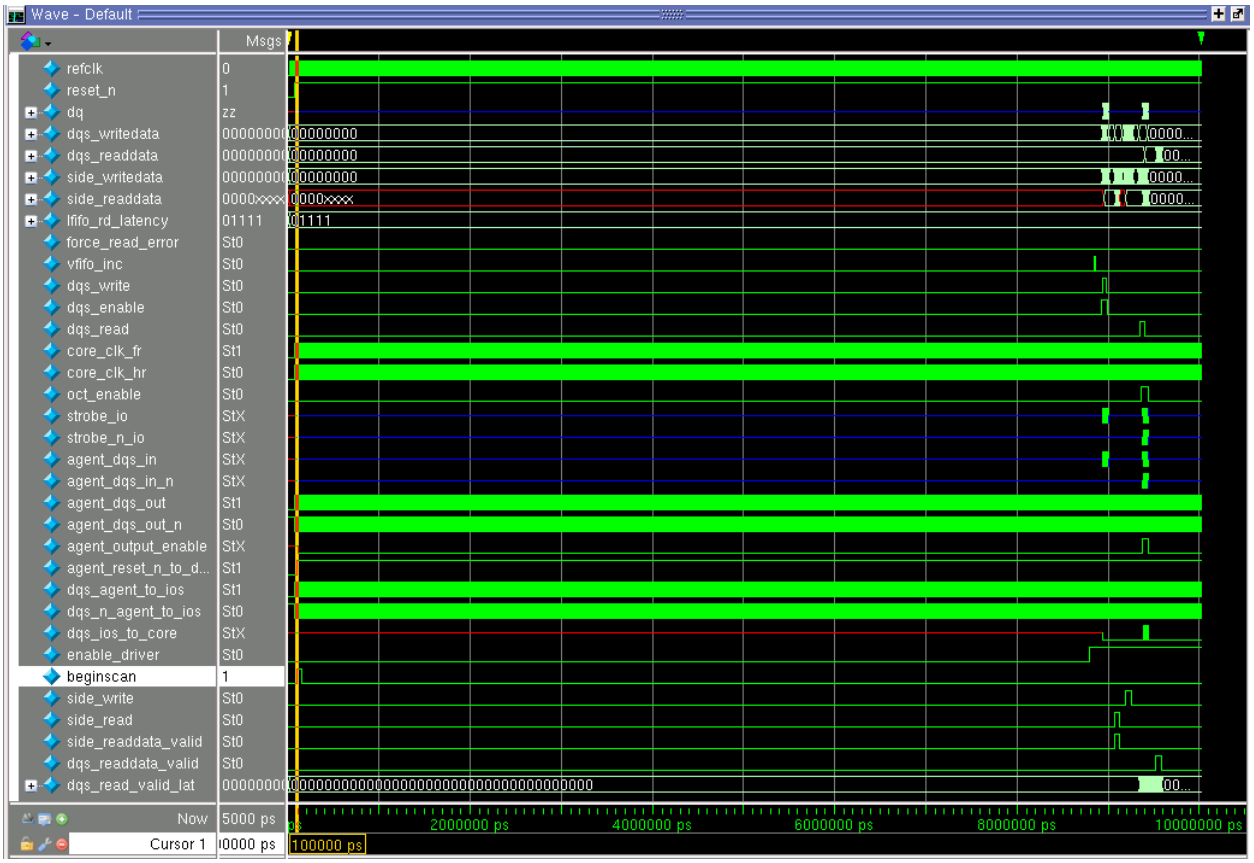


“Understanding Simulation Results - AV Example Design”

In this AV example design, a generic test bench is used to test the write and read operations in the ALTDQ_DQS2. In this test bench, DQSDriver which acts like the host controller, sends read/write commands to ALTDQ_DQS2. DQSAgent acts like an external memory device. DQSDriver also compares data read back from DQSAgent to what it should be. DQSDriver/DQSAgent has a side channel communicating directly with each other, bypassing ALTDQ_DQS2. These are called “Side Reads/Writes”. The data in the Side Reads/Writes will be used to compare with the data sent/received from the ALTDQ_DQS2.

Note: Random data is generated and used in the test bench. Thus user may see other data values if different operating system, seeds and so on is used.

Note: Descriptions below are all referring to the waveform generated after executing “top_run_msim_rtl_verilog.do”as shown below, unless specified.

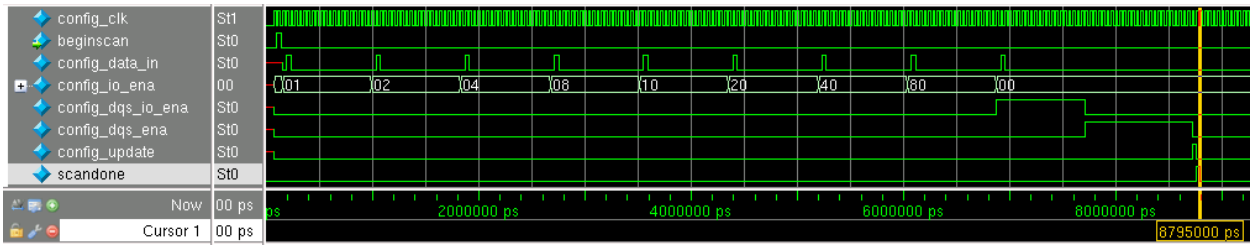


All ports are in reset mode until ***reset_n*** is asserted at 70ns time mark. Shortly after that, the major clocks (***core_clk_fr*** and ***core_clk_hr***) started toggling. ***Agent_reset_n_to_dqs*** is asserted at 95ns time mark to reset the ALTDQ_DQS2 block which resides in top_inst.

Dynamic Configuration

At 100ns time mark, there is a high pulse on ***beginscan.*** During the period when ***agent_output_enable*** is pulled low, ***strobe_io*** and ***agent_dqs_in*** goes to Hi-Z at between 110ns and 8.94ms time mark, some internal calibration is being carried out. Dynamic configuration is the main feature used. At 8.795ms time mark, ***enable_driver*** is asserted. This marks that the internal calibration is completed, and the control is now passed on to the host controller (in this case is the DQSdriver) to perform the normal read/write operation. Refer to "config_controller_acv.sv" for the RTL related to dynamic reconfiguration operations executed in this example design.

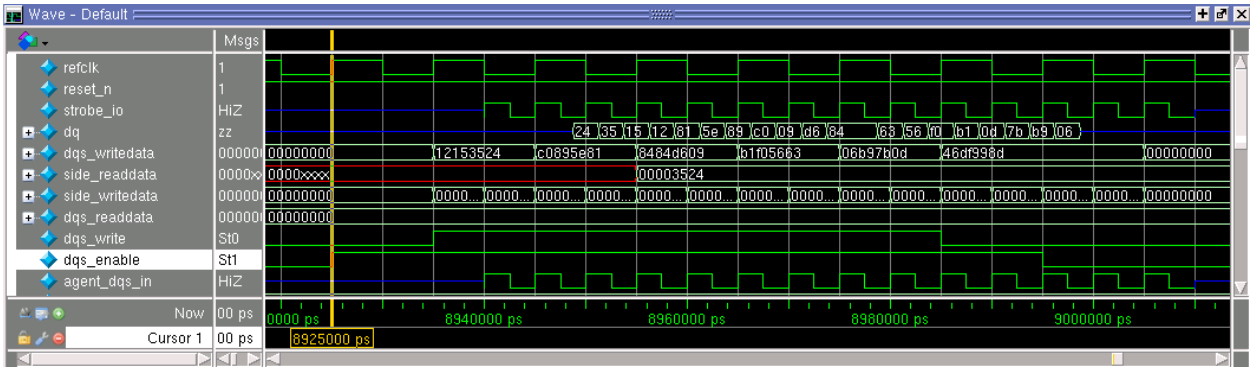
Through ALTDQ_DQS2's dynamic configuration feature, you can override the static values at runtime with a scan chain. This helps counter challenging static timing closure. Each I/O and the DQS logic contain its own scan chain block (shift registers). This mini section shows how to serially scan configuration bits into each scan chain block, happening between 100ns and 8.795ms time mark. Detailed dynamic configuration simulation can be viewed in the waveform generated after executing "topwave.do", in between the high pulses of ***beginscan*** and ***scandone*** as shown in the figure below. For more descriptions, refer to "Dynamic Reconfiguration for ALTDQ_DQS2 Megafunction" section in the User Guide.



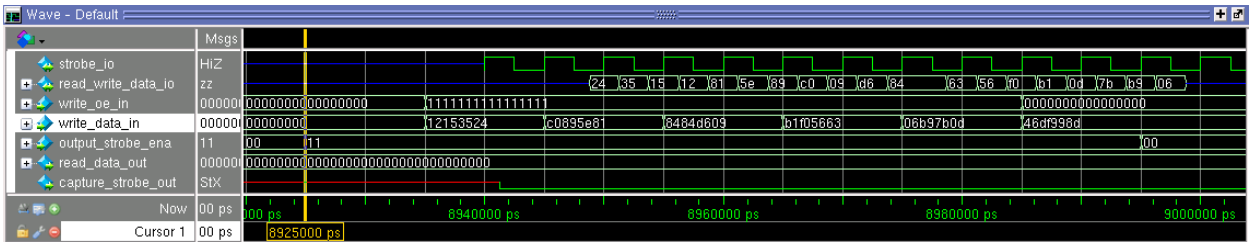
Since ***enable_driver*** is asserted at 8.795ms time mark, following read and write operations will be executed by DQSDriver per stated in dqsdriver.sv. (The driver_clk is running at the same rate as the core.)

DQS Write Operation

See waveform captured below. At 8.925ms time mark, Write Operation is started by the driver asserting ***dqs_enable***. This will configure the ALTDQ_DQS2's ***output_strobe_ena*** to high, getting ready to send out strobe to the DQSAgent. ***Dqs_write*** is asserted at 8.935ms. This set ALTDQ_DQS2's ***write_oe_in*** to high, getting ready to send out data to the DQSAgent. Note that as the data are first written to DQSDriver's ***dqs_writedata***, and then were reflected on the ALTDQ_DQS2's ***dq***. The data written out from the DQSDriver is also stored in ***check_fifo***. Notice also that during the DQS Write Operation, the ***strobe_io*** (and ***agent_dqs_in***) is toggling. Before and after this operation, it is in Hi-Z mode. ***Dqs_write*** deasserts at 8.985ms and ***dqs_enable*** deasserts at 8.995ms. Outgoing data from the ALTDQ_DQS2 (***dq***) is centre aligned to the strobe (***strobe_io***) as per expected.

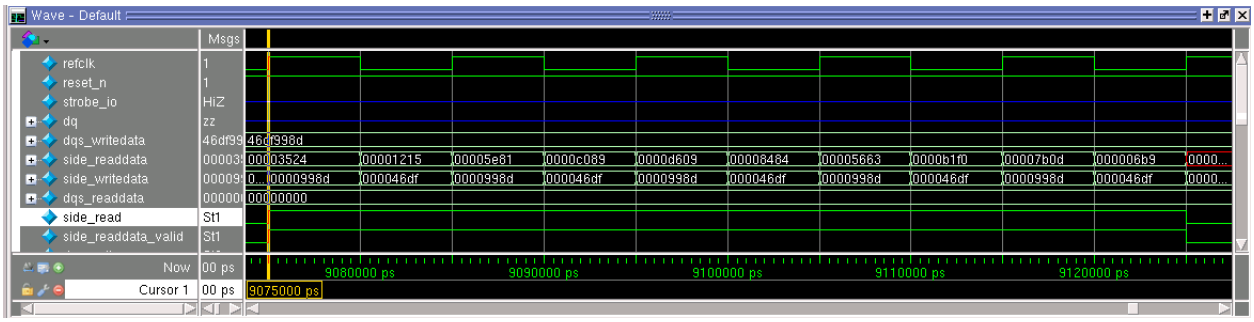


After executing "topwave.do", if look into more details the DQS write operation as shown in wave captured below, notice that ***output_strobe_ena*** is held high to from 8.925ms to 8.995ms while the ***strobe_io*** starts toggling only between 8.94ms and 9.01ms. On the other hand, ***write_oe_in*** is held high throughout the 5 sets of valid ***write_data_in***, which is between 8.935ms and 8.955ms. Centre aligned output data appears on ***read_write_data_io*** between 8.949ms and 8.999ms.



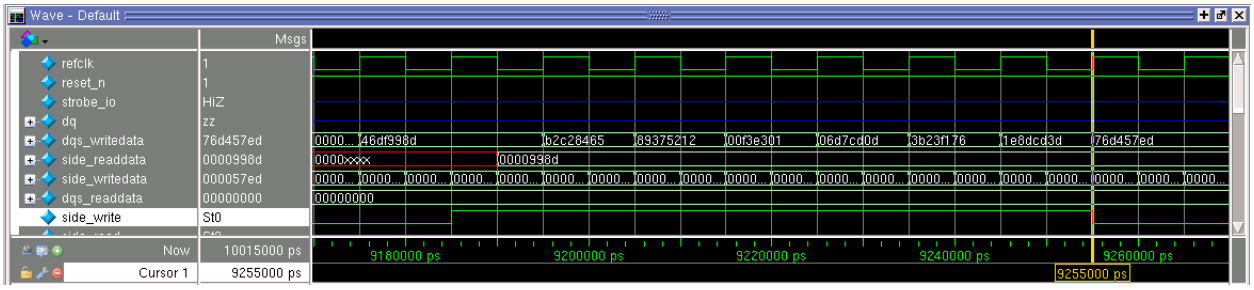
Side Read Operation

Consequently, Side Read operation was performed between 9.075ms and 9.125ms, where the data is output from DQSAgent to DQSDriver via ***side_readdata***. Data validation is carried out in parallel, by comparing the ***side_readdata*** against the content of the ***check_fifo***(data which was written out during DQS Write). Error message will be displayed if there is any mismatch. Waveform below shows the Side Read process.



Side Write Operation

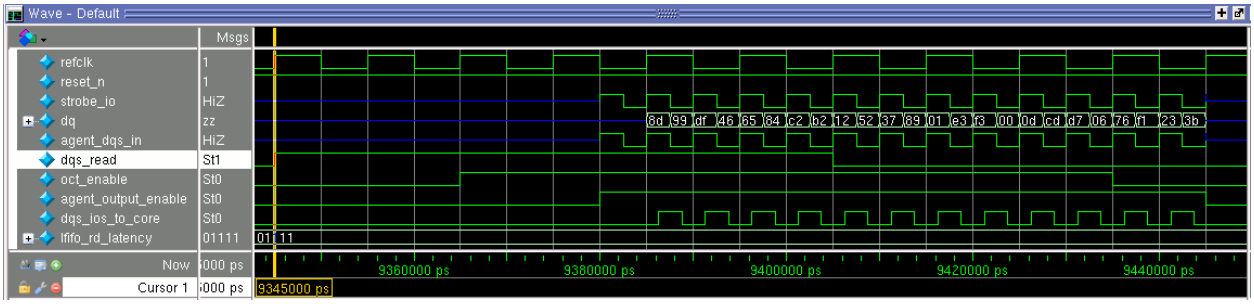
Then, Side Write operation takes place between 9.185ms and 9.255ms. The data written out from the DQSDriver is also stored in ***check_fifo***. Waveform below shows the Side Write process.



Side Write is followed by DQS Read. By comparing the data received by the ALTDQ_DQS2 core (***read_data_out***) against the ***check_fifo***'s content, we are able to validate if the DQS Read operation was successful. Take note that the incoming data at ***dq*** is edge-aligned to the ***strobe_io*** as per expected.

DQS Read Operation

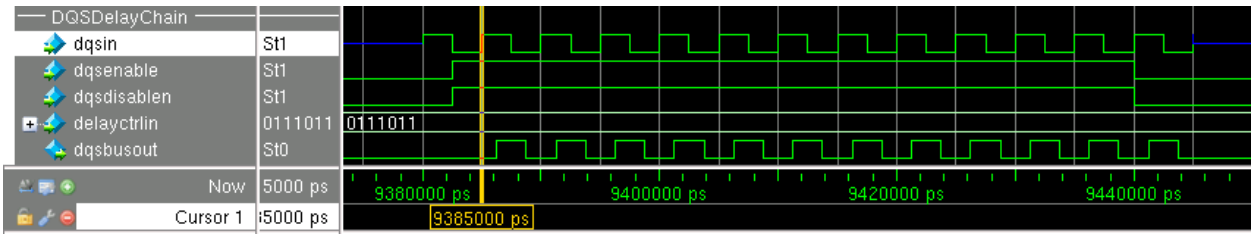
As DQS Read is usually the most challenging part for an external memory interface design, detailed elaboration will be covered in this section. DQS Read is initiated by the DQSDriver when ***dqs_read*** is asserted at 9.345ms. At this point, the ***lfifo_rdata_en_full*** (***dqs_read***) of the ALTDQ_DQS2 is also asserted. It is asserted for the entire length of the desired read burst, which in this case is 12 full-rate cycles. As the DQSAgent receive the read command, it is then ready to sends out data as per requested. This is shown with ***agent_output_enable*** being asserted from 9.380ms to 9.445ms. During this period, the DQSAgent drives the external memory interface's strobe and data lines. In this test bench, s***trobe_io*** is driven by ***dqs_agent_to_ios***. When output enable is deasserted, ***strobe_io*** will be set to Hi-Z.



As the incoming data arrives at the ALTDQ_DQS2, noticed the edge aligned data on ***read_write_data_io***, and strobe on ***strobe_io*** ports. Following will discuss how the data is captured in the FPGA before it is made available in the core.

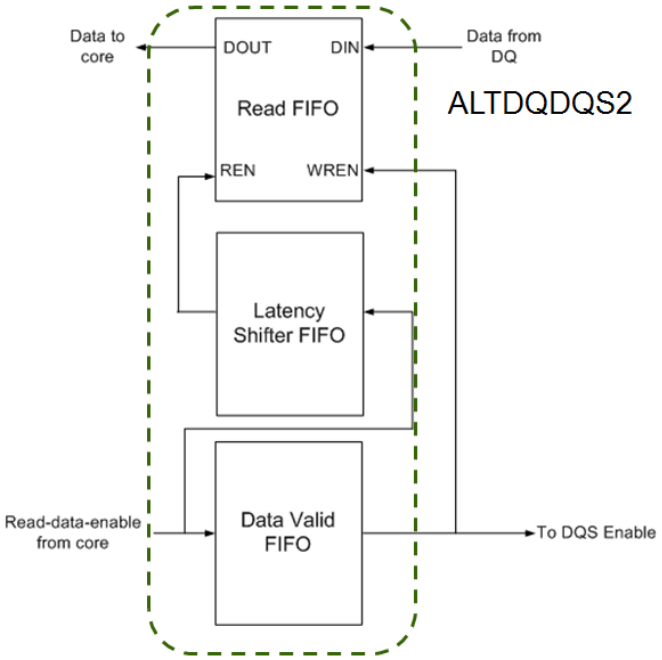
DQS Delay Chain

Dqsenable grounds the DQS input strobe after the strobe goes to Hi-Z**.** This is especially important for bidirectional strobes, where glitches can be filtered effectively through the DQS enable. And **dqsbusout** is the delayed **dqsin** signal that drives onto the dedicated DQS clock network to clock the DQ capture registers, so that data is captured at the centre of the eye. Notice that there is a 90 degree phase shift between dqsin and dqsbusout as shown in figure below. This is consistent as per settings in the ALTDQ_DQS2 GUI in this example design.



VFIFO, LFIFO and Read FIFO Relationship

To understand more on the DQS Read operation, let's recap on the hard FIFOs (VFIFO, LFIFO and Read FIFO) as well as each key relevant port. In AV and CV, Valid FIFO (VFIFO) generates the input signal to the DQS_ENABLE_CTRL block, and is also connected to the Read FIFO's Write Enable port. Latency FIFO (LFIFO) on the other hand, is connected to the Read Enable port of the Read FIFO. LFIFO and VFIFO will implement each configurable latencies according to determine the time to read enable and write enable for the Read FIFO respectively. ***lfifo_rd_latency*** determines the latency setting in the LFIFO, while ***vfifo_inc_wr_ptr*** determines the latency setting for the VFIFO. The Read FIFO is on every input-data path and user can set to have it handle conversion between FR-FR or FR-HR in AV and CV.

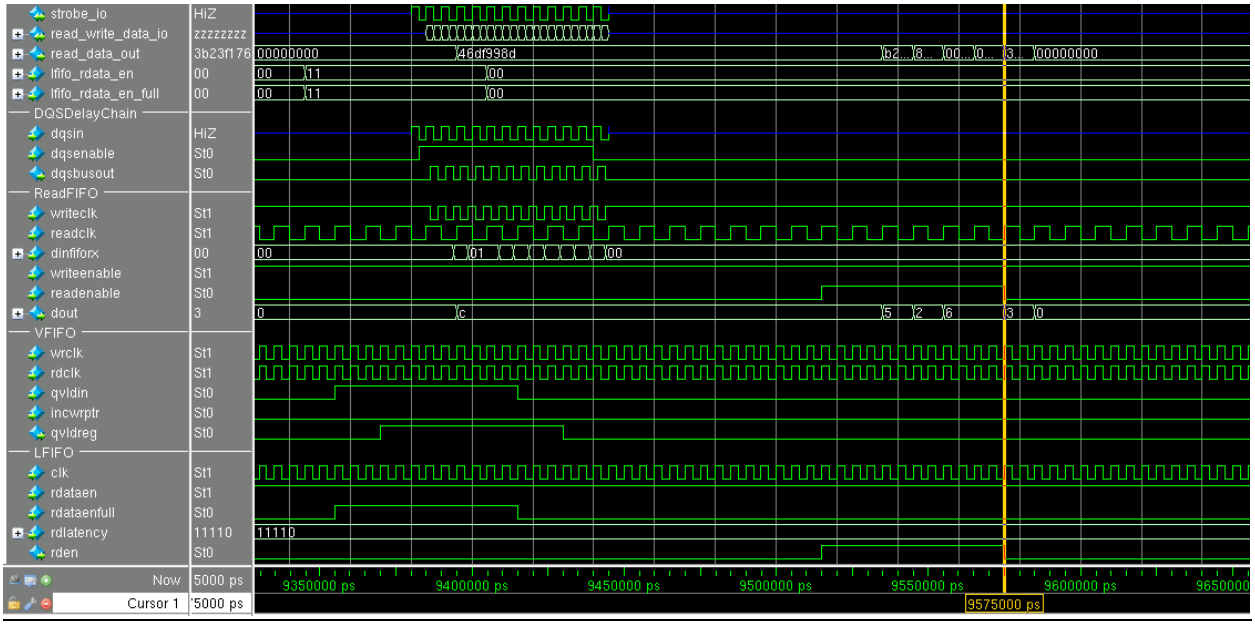


When the host sends out a read command, a token is also sent to the LFIFO and the VFIFO. This token should be asserted for the length of the desired read burst. This token is referred to ***lfifo_rdata_en_full*** (for LFIFO) and ***vfifo_qvld*** (for VFIFO). In this testbench, the ***lfifo_rd_latency*** is set to 15d. Note that this is consistent with the latency noticed between LFIFO's ***rdataenfull*** and Read FIFO's ***readenable,*** which is 16 half- rate clock cycles.

lfifo_rdata_en_full and ***vfifo_qvld*** signals passes through each HR-FR DDIO before it reaches the LFIFO's ***rdataenfull*** and VFIFO's ***qvldin*** at 9.355ms. The delayed VFIFO's ***qvldreg*** (1.5 half-rate cycles) will feed the ***DQSENABLEIN*** of the dqs_enable_ctrl block. Meanwhile the delayed (by 16 half-rate cycles) LFIFO's ***RDEN*** will feed the ***READENABLE*** of Read FIFO.

The effective DQS Enable (***dqsenable***) is asserted at 9.3825ms. Edge aligned input data starts flowing in at 9.385ms and ends at 9.445ms. DQS Enable is deasserted at 9.44ms. (In actual application, users are required to have some form of DQS Enable run time calibration to obtain the correct DQS gating-ungating window.) Note that ***strobe_io*** goes to Hi-Z when the DQS Read operation completes.

As the ***writeenable*** of the ReadFIFO is always asserted, data is written into the Read FIFO whenever available, which is as early as 9.395ms. However, it is only being read out when the ***readenable*** is asserted between 9.515ms and 9.575ms. These data are available on ***read_data_out***. Users may further optimize the timing to read the Read FIFO (by adjusting the LFIFO Latency delay) to create sufficient space between read and write pointers in the Read FIFO to maximize the throughput.



*Note 1: The Hard Read FIFO's **we** and **re** signals is different from the DCFIFO's **wrreq** and **rdreq**. In DCFIFO, the data will only be made available at the FIFO output ports when the **rdreq** is asserted. For the*
*Hard Read FIFO, the **write enable (we)** signal control when to advance write address counter, while the*
read enable (re)** signal control when to advance read address counter. When read/write address pointers are the same, write data shows up at the read port as soon as a write is completed. This explains why we see the first written data available almost immediately at the FIFO output port. When **re
is asserted, it will advance to the next read address and then only the second written data is available at
the FIFO output port.

*Note 2: **lfifo_rdata_en** and **lfifo_rdata_valid** will be removed from QII 13.1 onwards.*

DQS Enable Control

The goal of DQS enable calibration is to find settings that satisfy the following conditions:

- The DQS enable signal rises before the first rising edge of DQS.
- The DQS enable signal is at one after the second-last falling edge of DQS.
- The DQS enable signal falls before the last falling edge of DQS.

The ideal position for the falling edge of the DQS enable signal is centered between the second-last and last falling edges of DQS.

For more details, please refer to http://www.altera.com/literature/hb/external-memory/emi_fd_uniphy.pdf, UniPHY Calibration Stages chapter.

Simulation Results

Below is the printed result (showing successful simulation) which you may find in the message panel. If the simulation was not successful, it is due to the data sent/received at the ALTDQ_DQS2 is not the same as the expected ones.

```
#     [8945000] DQS WRITE: 12153524
#     [8955000] DQS WRITE: c0895e81
#     [8965000] DQS WRITE: 8484d609
#     [8975000] DQS WRITE: b1f05663
#     [8985000] DQS WRITE: 06b97b0d
#     [9095000] SIDE READ: 12153524
#     [9105000] SIDE READ: c0895e81
#     [9115000] SIDE READ: 8484d609
#     [9125000] SIDE READ: b1f05663
#     [9135000] SIDE READ: 06b97b0d
#     [9195000] SIDE WRITE: 46df998d
#     [9205000] SIDE WRITE: b2c28465
#     [9215000] SIDE WRITE: 89375212
#     [9225000] SIDE WRITE: 00f3e301
#     [9235000] SIDE WRITE: 06d7cd0d
#     [9245000] SIDE WRITE: 3b23f176
#     [9255000] SIDE WRITE: 1e8dcd3d
#     [9535000] DQS READ: 46df998d
#     [9545000] DQS READ: b2c28465
#     [9555000] DQS READ: 89375212
#     [9565000] DQS READ: 00f3e301
#     [9575000] DQS READ: 06d7cd0d
#     [9585000] DQS READ: 3b23f176
# Simulation SUCCESS
```

SDC Brief Walkthrough

When starting a new SDC file, the first thing to do is constrain the clocks coming into the FPGA with create_clock. Following command create the base clock for the input clock port driving the PLL.
create_clock -name refclk -period 10.000 [get_ports {refclk}]

Next, create the generated clocks for the PLL.
derive_pll_clocks

Following commands constraint the virtual input clock (for incoming DQS strobe) and the strobe_io port. In this example design, it is based on a 200MHz input clock, with a 50% duty cycle, where the first rising edge occurs at 0ns.
create_clock -name virtual_dqs_in -period 5.000
create_clock -name dqs_in -period 5.000 [get_ports {strobe_io}]

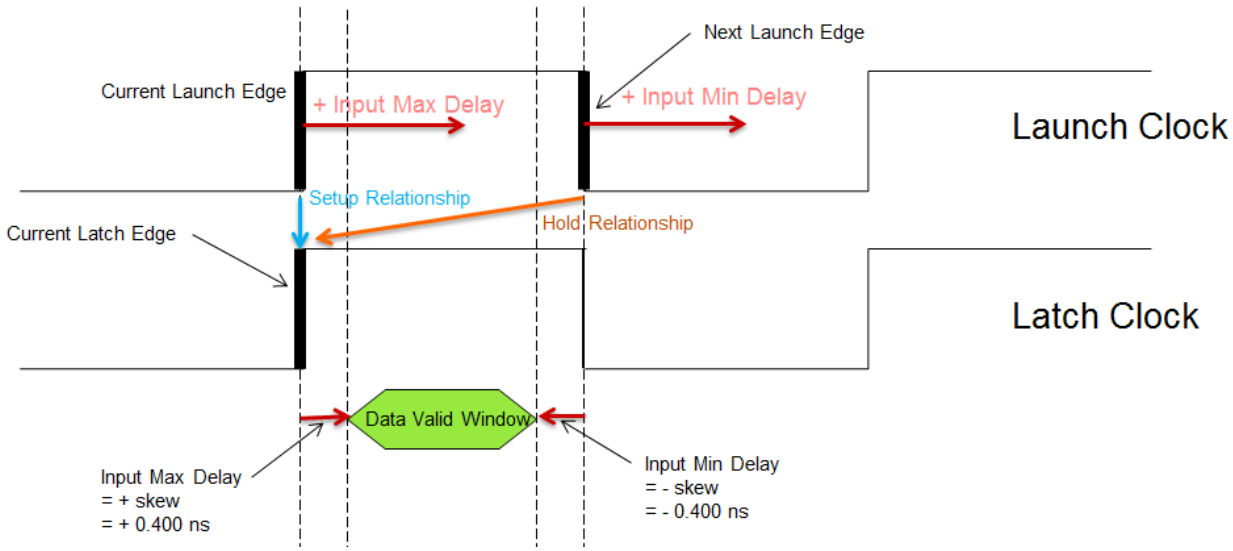
Incoming data is edge aligned to the DQS strobe, and min/max input delay is assumed to be +/- 0.4ns in this example design. Users must modify constraints to reflect data/clock relationship in their system. Use the -add option to add the user-defined delay constraint instead of overriding previous constraints.

set_input_delay -clock { virtual_dqs_in } -max -add_delay 0.400 [get_ports {read_write_data_io[*]}]

set_input_delay -clock { virtual_dqs_in } -min -add_delay -0.400 [get_ports {read_write_data_io[*]}]

set_input_delay -clock { virtual_dqs_in } -clock_fall -max -add_delay 0.400 [get_ports {read_write_data_io[*]}]

set_input_delay -clock { virtual_dqs_in } -clock_fall -min -add_delay -0.400 [get_ports {read_write_data_io[*]}]



Following set_false_path commands ensure that we are analyzing only the same edge transfers, by removing the opposite edge transfers. Note that these assignments are optional (they do not hurt timing if done properly, but since they do not help timing either, they were left out of the example .sdc.) These SDC constraints cut timing on paths which are not within interest, which are much looser than the real input paths. So cutting these paths does not make it any easier to meet timing. However, mistakenly cutting the real paths with these assignments may result in a design that easily meets timing but does not work in hardware.

set_false_path -setup -rise_from [get_clocks {virtual_dqs_in}] -fall_to [get_clocks {dqs_in}]

set_false_path -setup -fall_from [get_clocks {virtual_dqs_in}] -rise_to [get_clocks {dqs_in}]

set_false_path -hold -rise_from [get_clocks {virtual_dqs_in}] -rise_to [get_clocks {dqs_in}]

set_false_path -hold -fall_from [get_clocks {virtual_dqs_in}] -fall_to [get_clocks {dqs_in}]

The default setup relationship is to latch data on the next edge. Following multicycles tell TimeQuest to analyze the paths as a same-edge transfer, whereby the same edge that launches data is going to latch it. The reason it is latched on the same edge is that latch edge will be delayed by the DQS circuitry (hardened 90 degree in this example design) into the middle of the data eye.

set_multicycle_path -rise_from [get_clocks {virtual_dqs_in}] -rise_to [get_clocks {dqs_in}] -setup -end 0

set_multicycle_path -fall_from [get_clocks {virtual_dqs_in}] -fall_to [get_clocks {dqs_in}] -setup -end 0

Following commands constraint the outgoing DQS strobe. Note that we need to have –add option for the create_generated_clock command for the strobe_io port since it is bidirectional. As the design sends the data out by a clock shifted 270 degrees, so that the non-shifted clock is center-aligned. These constraints say the external device adds +/-250ps of skew, which could also be described as a setup requirement of 250ps and hold requirement of 250ps. These numbers are an example, and user must modify constraints to reflect data/clock relationship in their system. Use the -add option to add the user-defined delay constraint instead of overriding previous constraints.

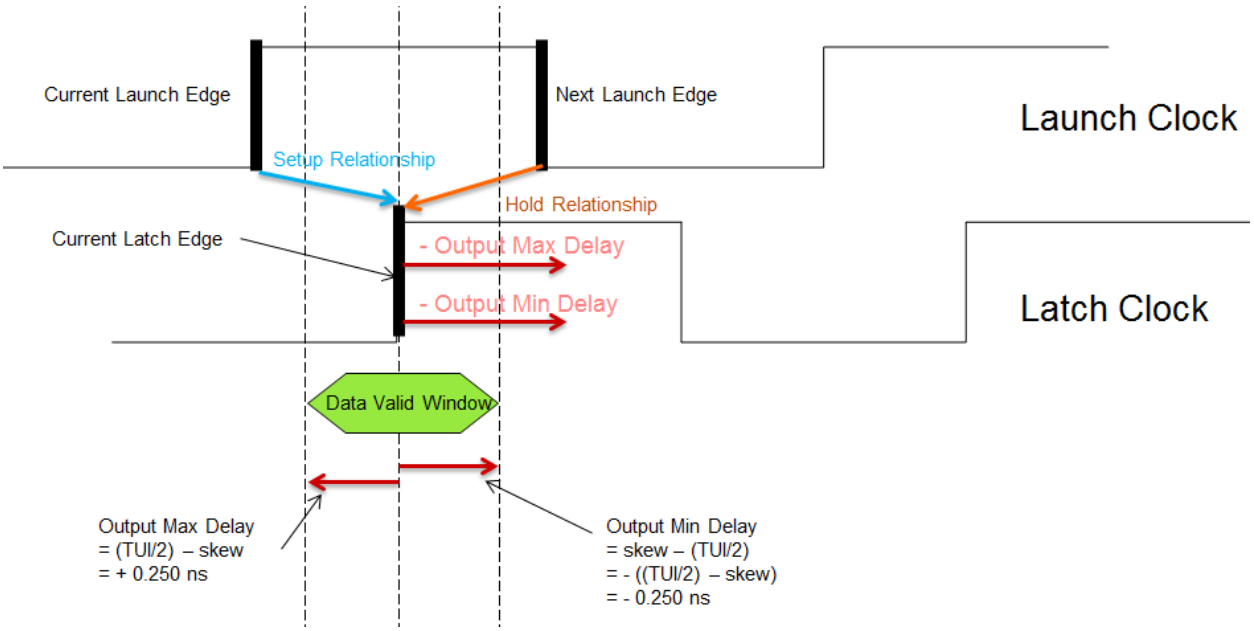
create_generated_clock -name dqs_out -source [get_pins { dqdqs2_inst|bidir_hardfifo_dqdqs2_inst|altdq_dqs2_inst|phy_clkbuf|outclk[1] }] -phase 0 [get_ports {strobe_io}] -add

set_output_delay -clock { dqs_out } -max 0.250 [get_ports {read_write_data_io[*]}] -add_delay

set_output_delay -clock { dqs_out } -max 0.250 -clock_fall [get_ports {read_write_data_io[*]}] -add_delay

set_output_delay -clock { dqs_out } -min -0.250 [get_ports {read_write_data_io[*]}] -add_delay

set_output_delay -clock { dqs_out } -min -0.250 -clock_fall [get_ports {read_write_data_io[*]}] -add_delay



Following set_false_path commands ensure that we are analyzing only the same edge transfers, by removing the opposite edge transfers. Note that these assignments are optional (they do not hurt timing if done properly, but since they do not help timing either, they were left out of the example .sdc.) These SDC constraints cut timing on paths which are not within interest, which are much looser than the real input paths. So cutting these paths does not make it any easier to meet timing. However, mistakenly cutting the real paths with these assignments may result in a design that easily meets timing but does not work in hardware.

set_false_path -setup -rise_from [get_clocks
{ pll_inst|alterapll_inst|altera_pll_i|general[3].gpll~PLL_OUTPUT_COUNTER|divclk }] -fall_to
[get_clocks {dqs_out}]

set_false_path -setup -fall_from [get_clocks
{ pll_inst|alterapll_inst|altera_pll_i|general[3].gpll~PLL_OUTPUT_COUNTER|divclk }] -rise_to
[get_clocks {dqs_out}]

set_false_path -hold -rise_from [get_clocks
{ pll_inst|alterapll_inst|altera_pll_i|general[3].gpll~PLL_OUTPUT_COUNTER|divclk }] -rise_to
[get_clocks {dqs_out}]

set_false_path -hold -fall_from [get_clocks
{ pll_inst|alterapll_inst|altera_pll_i|general[3].gpll~PLL_OUTPUT_COUNTER|divclk }] -fall_to
[get_clocks {dqs_out}]

The strobe port functions as input or output at a time. Non relevant transfers below should be set to false path and do not need to be analyzed.

set_false_path -from [get_clocks {virtual_dqs_in}] -to [get_clocks {dqs_out}]

Do note that there is a timing violation in the example design, as shown below. This path is related to DQS Enable Control and is valid. Some calibration algorithm is required to control the DQS Enable Block.

Hold: dqs_out

Command Info | Summary of Paths

	Slack	From Node	To Node
1	-2.063	bidir_hardFIFO_dqdqs2:dqdqs...nable_ctrl~DQSENABLEOUT_DFF	bidir_hardFIFO_dqdqs2:dqdqs2_...dqs_delay_chain~POSTAMBLE_DFF
2	-2.063	bidir_hardFIFO_dqdqs2:dqdqs...nable_ctrl~DQSENABLEOUT_DFF	bidir_hardFIFO_dqdqs2:dqdqs2_...dqs_delay_chain~POSTAMBLE_DFF

Without any calibration algorithm in place, this path cannot be set as false path in the static timing analysis.

#set_false_path -from [get_keepers {|dqs_enable_ctrl~DQSENABLEOUT_DFF}] -to [get_clocks {dqs_out}]*

For more information related to the PHY Clock and DQS Logic Blocks, please refer to http://www.altera.com/literature/hb/arria-v/av_52007.pdf.

For more information on some basic calibration, please refer to http://www.altera.com/literature/hb/external-memory/emi_fd_uniphy.pdf, "UniPHY Calibration Stages"

In this example design, TQ_analysis.tcl has been created for you. This is a script that can be used to analyze specific paths of the dqdqs I/O timing. Note that due to read_data_out and write_data_in are not registered outside of the core, there is nothing to analyze. Since the user should be changing their I/O constraints for their specific implementation, having this TCL script may help them to quickly run specific timing analysis.
