TSW14DL3200 ADC REFERENCE DESIGN

USER GUIDE

Contents

1. Overview

This document explains how to capture data from ADC12DL3200 in TSW14DL3200.

1.1 Platform description

TSW14DL3200 board is a hardware platform to evaluate data converters performance at data rate upto 1.6Gbps using LVDS. It is built with Ultrascale FPGA XCKU060-FFVA1517-2e device with FMC connectors and JTAG port.

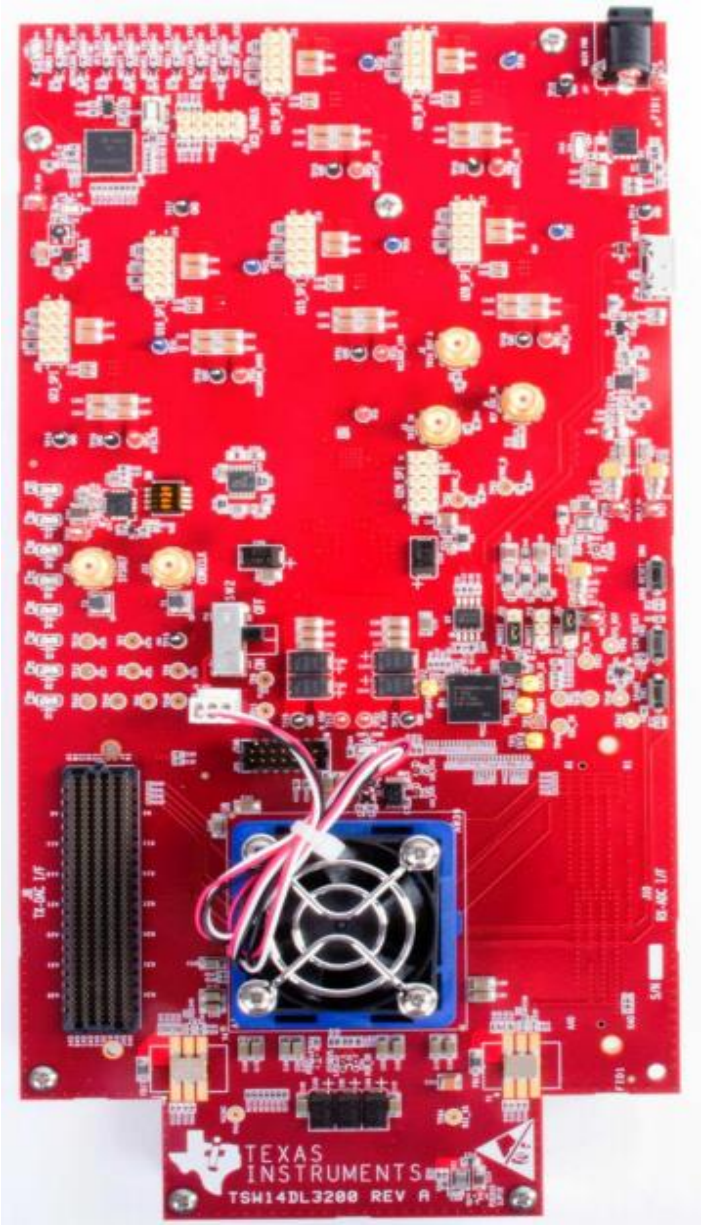


Fig 1.1(1)

1.2 Prerequisites for using the reference design

I. Vivado verison : 2017.1 – (Kintex Ultrascale package has to be installed)

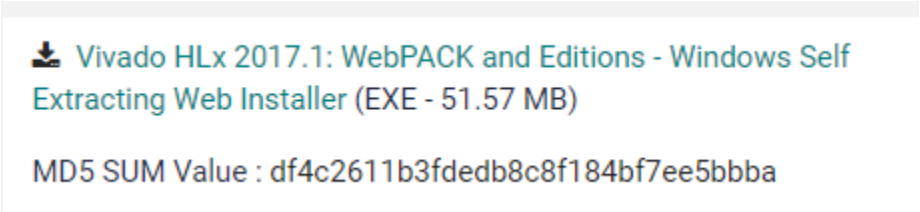


Fig 1.2(1)

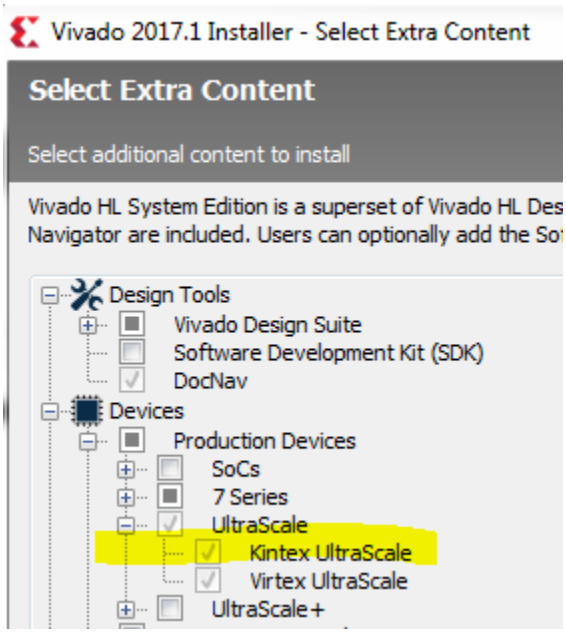


Fig 1.2(2)

II. Xilinx Platform cable
III. ADC12DLxx00 software GUI

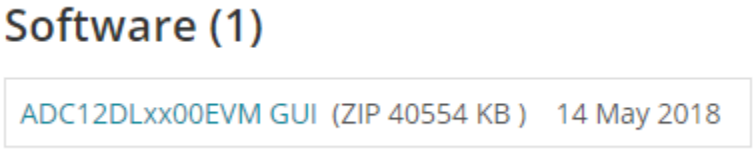


Fig 1.2(3)

1.3 Scope of the reference design

I. This design supports only single channel 4 bus mode (12 data lanes, 1 strobe lane and 1 forwarded clock per bus) of ADC12DL3200 (LDEMUX = 1, DES_EN =1)
II. The shared FPGA source firmware is built for 1.6Gbps lane rate.

Section 2 explains on how to execute TCL script and the compilation process involved. Section 3 discusses about how to connect the hardware setup. Section 4 discusses on how to get the ADC data in chipscope and validate the same. Section 5 explains the procedure to edit the reference design for different lane rates. Section 6 explains the details of custom LRX module and implementation of lane sync work around.

2. Details of the Reference Design

2.1 Folder structure

The tsw14dl3200_adc_ref_design.zip file contains the following folders:

I. **Ip_repos**: This folder contains all the necessary source file for the reference design packed as a different set of IP's.
II. **Tcl_scripts**: Scripts used to generate ADC12DL3200 reference design demo for the TSW14DL3200 platform
III. **Xdc**: Top constraint file needed for the reference design has placed in this folder.
IV. **Bitstream**: This folder contains pre-built bitstream and LTX. The bitstream and LTX was built using the files in xdc and ip_repos folder.

Note: Change in folder structure/folder name can cause error in block diagram development and tsw14dl3200_adc_ref_design.zip should be unzipped in the path with ASCII characters excluding space.

2.2 Execution of TCL script

Open vivado 2017.1 and run the tcl script located in "tcl_scripts" folder through tcl console of the vivado

Syntax: **source {*<tcl filepath>.tcl*}**

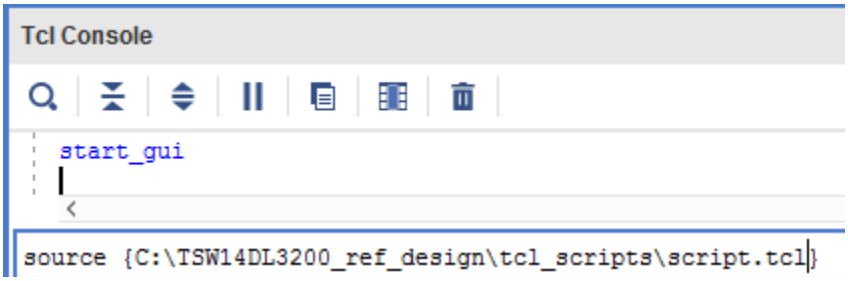


Fig 2.2(1)

On executing tcl script, block diagram will be developed and vivado generated files will be located at ../project/ . (New folder "project" will be created while running tcl script to save all the generated files)

2.3 IO ports

Following are the list of input and output ports of the reference design

Port	In/Out	Clock domain	Description
System signals connected through IO pins			

Clk_100m_p	input	NA	On-board differential 100Mhz clock P connected to PLL to derive RIU clk
Clk_100m_n	input	NA	On-board differential 100Mhz clock N connected to PLL to derive RIU clk
Hw_rst_n	input	NA	Active low hardware reset
System signals connected from FPGA fabric			
Sw_rst	input	NA	Active high software reset: For more details refer to sec 3
ADC control signals connected to/from IO pins			
Rx_clk_p[3:0]	input	NA	Forwarded differential clock input P connected to HSSIO IP PLL
Rx_clk_n[3:0]	input	NA	Forwarded differential clock input N connected to HSSIO IP PLL
Rx_data_p[47:0]	input	0-11 : rx_clk_p[0] 12-23 : rx_clk_p[1] 24-35 : rx_clk_p[2] 36-47 : rx_clk_p[3]	Forwarded differential data input P connected to bitslice
Rx_data_n[47:0]	input	0-11 : rx_clk_n[0] 12-23 : rx_clk_n[1] 24-35 : rx_clk_n[2] 36-47 : rx_clk_n[3]	Forwarded differential data input N connected to bitslice
Rx_strobe_p[3:0]	input	0 : rx_clk_p[0] 1 : rx_clk_p[1] 3 : rx_clk_p[2] 4 : rx_clk_p[3]	Forwarded differential strobe input P connected to bitslice
Rx_strobe_n[3:0]	input	0 : rx_clk_n[0] 1 : rx_clk_n[1] 3 : rx_clk_n[2] 4 : rx_clk_n[3]	Forwarded differential strobe input P connected to bitslice
Rx_sync	Output	app_clk	Indicates multi bank lane synchronization – 1: lane are synchronized, 0: lanes are not synchronized

ADC_busA interface signals connected through FPGA fabric			
busA_sample0[11:0]	Output	app_clk	S0 (0th sample) - Refer to timing diagram of single channel,4bus mode in ADC12DL3200 datasheet
busA_sample1[11:0]			S4
busA_sample2[11:0]			S8
busA_sample3[11:0]			S12
busA_sample4[11:0]			S16
busA_sample5[11:0]			S20
busA_sample6[11:0]			S24
busA_sample7[11:0]			S28
busA_strb[7:0]			Deserialized strobe
busA_valid			1: indicates the data from bus A is valid; 0: indicates the invalid data
ADC_busB interface signals connected through FPGA fabric			
busB_sample0[11:0]	Output	app_clk	S1(1st sample) - Refer to timing diagram of single channel,4bus mode in ADC12DL3200 datasheet
busB_sample1[11:0]			S5
busB_sample2[11:0]			S9
busB_sample3[11:0]			S13
busB_sample4[11:0]			S17
busB_sample5[11:0]			S21
busB_sample6[11:0]			S25
busB_sample7[11:0]			S29
busB_strb[7:0]			Deserialized strobe
busB_valid			1: indicates the data from bus B is valid; 0: indicates the invalid data
ADC_busC interface signals connected through FPGA fabric			

busC_sample0[11:0]	Output	app_clk	S2(2nd sample) - Refer to timing diagram of single channel,4bus mode in ADC12DL3200 datasheet
busC_sample1[11:0]			S6
busC_sample2[11:0]			S10
busC_sample3[11:0]			S14
busC_sample4[11:0]			S18
busC_sample5[11:0]			S22
busC_sample6[11:0]			S26
busC_sample7[11:0]			S30
busC_strb[7:0]			Deserialized strobe
busC_valid			1: indicates the data from bus C is valid; 0: indicates the invalid data
ADC_busD interface signals connected through FPGA fabric			
busD_sample0[11:0]	Output	app_clk	S3(3rd sample) - Refer to timing diagram of single channel,4bus mode in ADC12DL3200 datasheet
BusD_sample1[11:0]			S7
busD_sample2[11:0]			S11
busD_sample3[11:0]			S15
busD_sample4[11:0]			S19
busD_sample5[11:0]			S23
busD_sample6[11:0]			S27
busD_sample7[11:0]			S31
busD_strb[7:0]			Deserialized strobe
busD_valid			1: indicates the data from bus D is valid; 0: indicates the invalid data
Status signals connected to LED's			

Rx_rst_done_led	Output	riu_clk	Indicates all the 4 HSSIO Ip's are out of reset – LED ON: IP's are out of reset , LED OFF : IP's are in reset state

Table 2.3(1)

ADC_busA, ADC_busB, ADC_busC and ADC_busD interfaces in the block diagram has been connected to dummy_adc_data_capture IP to avoid logic optimization during vivado implementation. This IP and "and_of_all_busdata "port can be removed when connecting other user modules to ADC_busA, ADC_busB, ADC_busC and ADC_busD interfaces.

2.4 Compilation procedure using Vivado IDE

To generate the bit stream with debug probes,

I. Generate output products for the block diagram with OOC per IP option enabled (For more details refer to UG896)

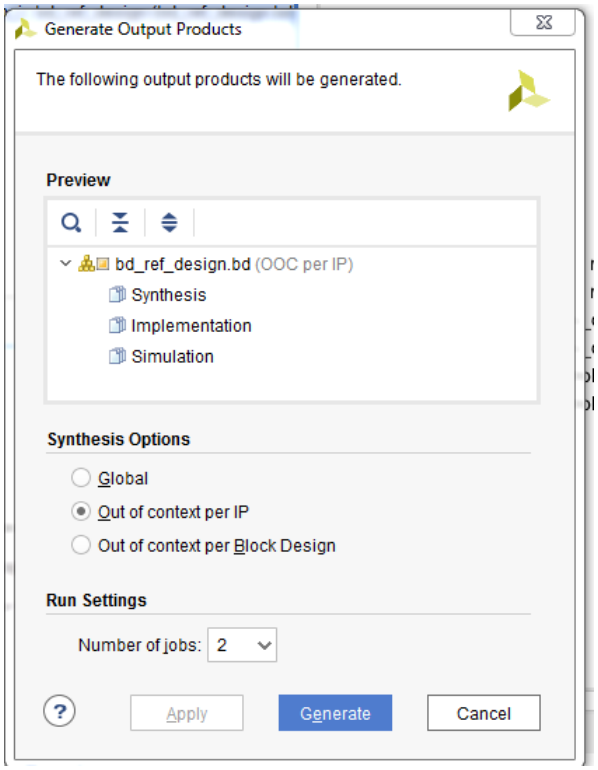


Fig 2.4(1)

II. After generating the output products, click "Run synthesis" option in flow navigator
III. After synthesis, open the synthesized design in flow navigator to add debug probes
IV. Click "Setup debug" option in flow navigator

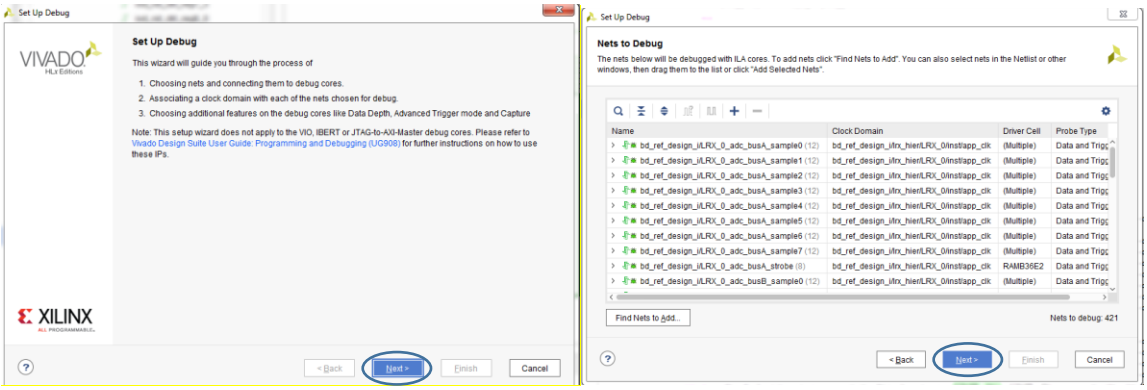


Fig 2.4(2)

V. Click “Next” on both above windows
VI. In “ILA core options” window, configure with below configurations
   a. Sample of data depth : less than 16384
   b. Input stages : 0 or more
   c. Capture control : Checked
   d. Advanced trigger : Unchecked

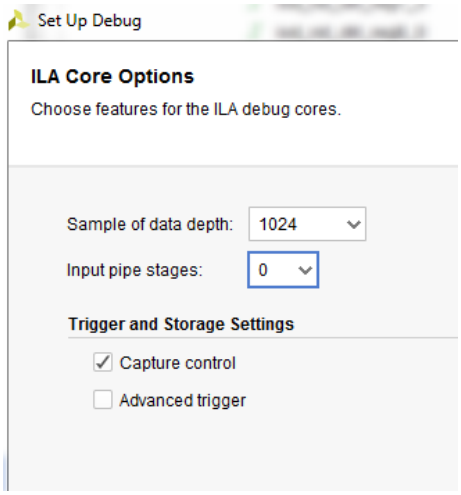


Fig 2.4(3)

VII. Click “Next” and “finish” to add the debug probes into the design
VIII. Save the project and click “Generate bitstream” option in flow navigator

Note: For details on chipscope, refer to UG936

3. Hardware and software setup

3.1 TSW14DL3200 Configuration

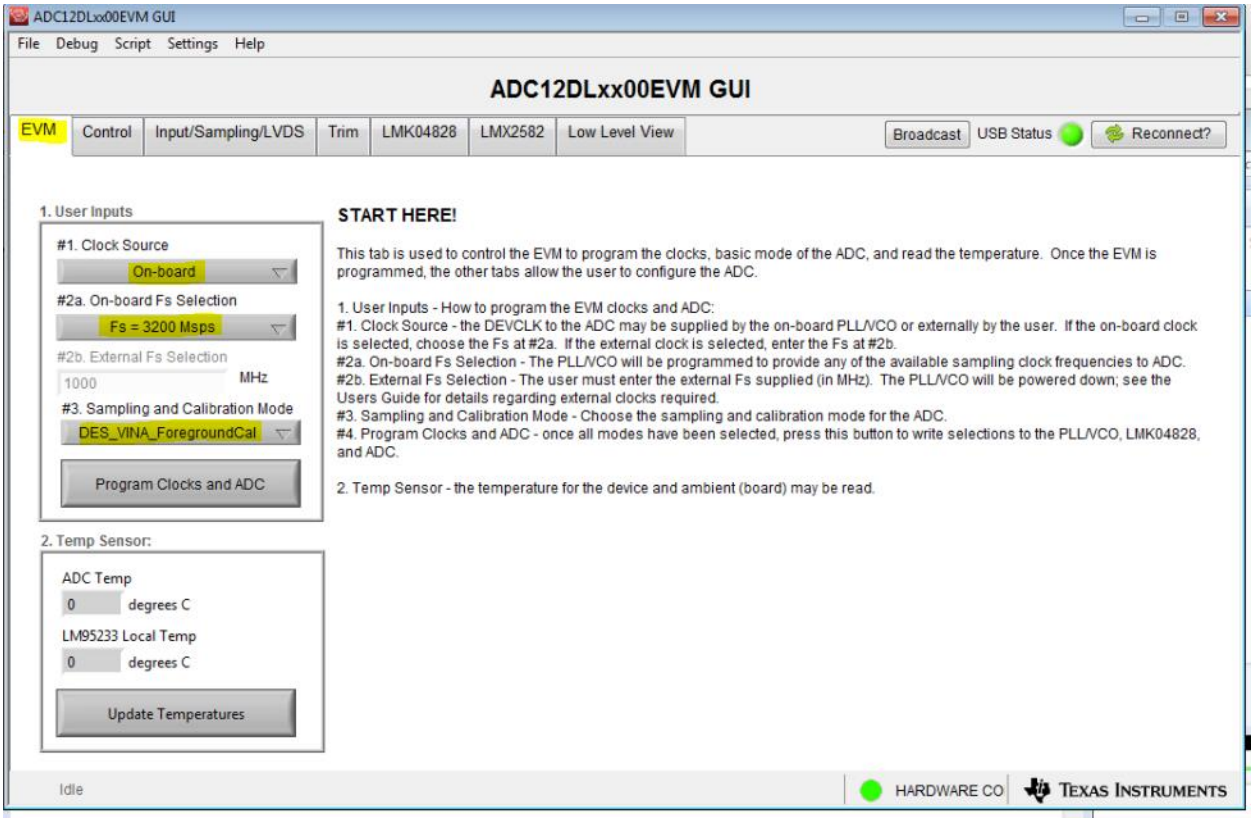
I. Populate R336 in TSW14DL3200.

3.2 ADC12DL3200 configuration

Refer to ADC12DL3200 user guide for the hardware setting and power requirement.

I. Open the ADC12DLxx00 EVM GUI
II. Navigate to “EVM” tab
III. Under “user inputs”, configure as below
    1. Clock source : On-board
    2. On board Fs selection : 3200Msps
    3. Sampling and calibration mode : DES_VINA_ForegroundCal (DES_VINA_ForegroundCal is for single channel,4 bus and staggered mode at 1.6Gbps data rate)
IV. Click “Program Clocks and ADC”

*LVDS Lane Rate = fs/2 for this mode of operation.



4. Hardware demonstration

4.1 Configuring Vivado Hardware Manager

Refer to ug908 xilinx document for details on how to configure the vivado to download the bit file and load debug probes.

4.2 Checking Results using Chipscope

Chipscope is the debugger option available in vivado. For more details on chipscope refer to UG936.

The internal probe points are defined in LTX file which will be located in ..\project\adc_ref_design.runs\impl_1\ bd_ref_design_wrapper.ltx

From the hardware manager, program the LTX file along with the bit file into the device.

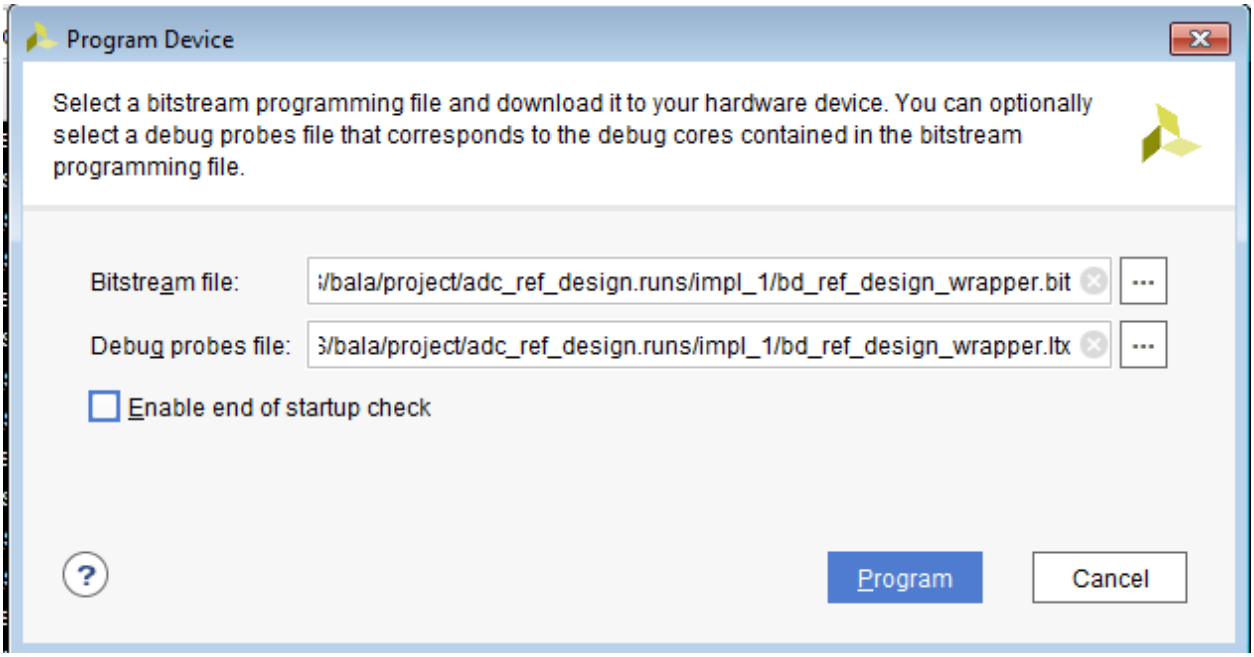


Fig 4.2(1)

Note: ADC has to be configured before downloading the firmware as chipscope expects continuous clock else vivado may fail to detect the probes.

In "**hw_ila_1**", signals that are added to the probe windows are displayed. The list of signals probed are

I. In Table 2.3(1), all the signals under adc_busA interface, adc_busB interface, adc_busC interface, adc_busD interface are probed
II. Rx_sync : Indicates bitslip and lane synchronization has been completed

The hw_ila_1 window is shown in Fig 4.2(2), (3). Click immediate capture button (circled in Fig 4.2(2)) to capture the live data.

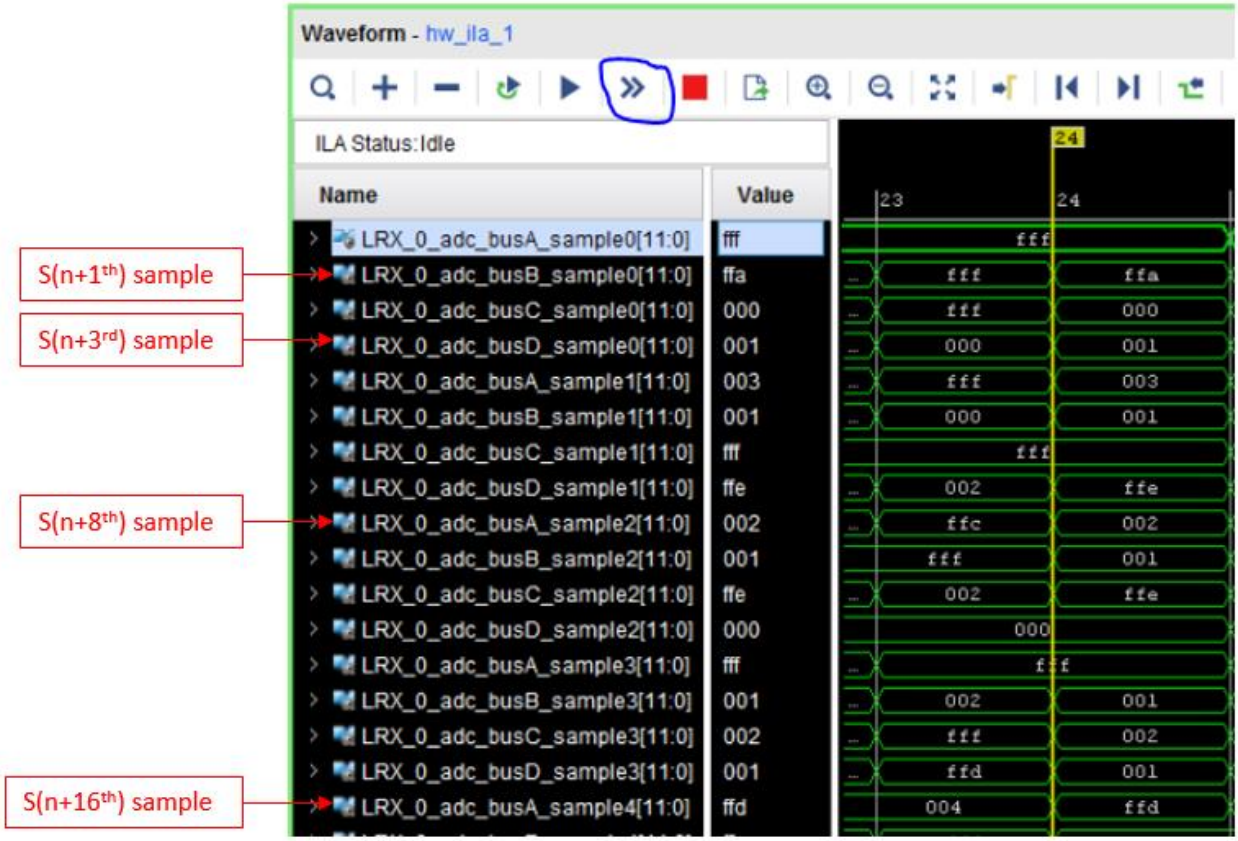


Fig 4.2(2)

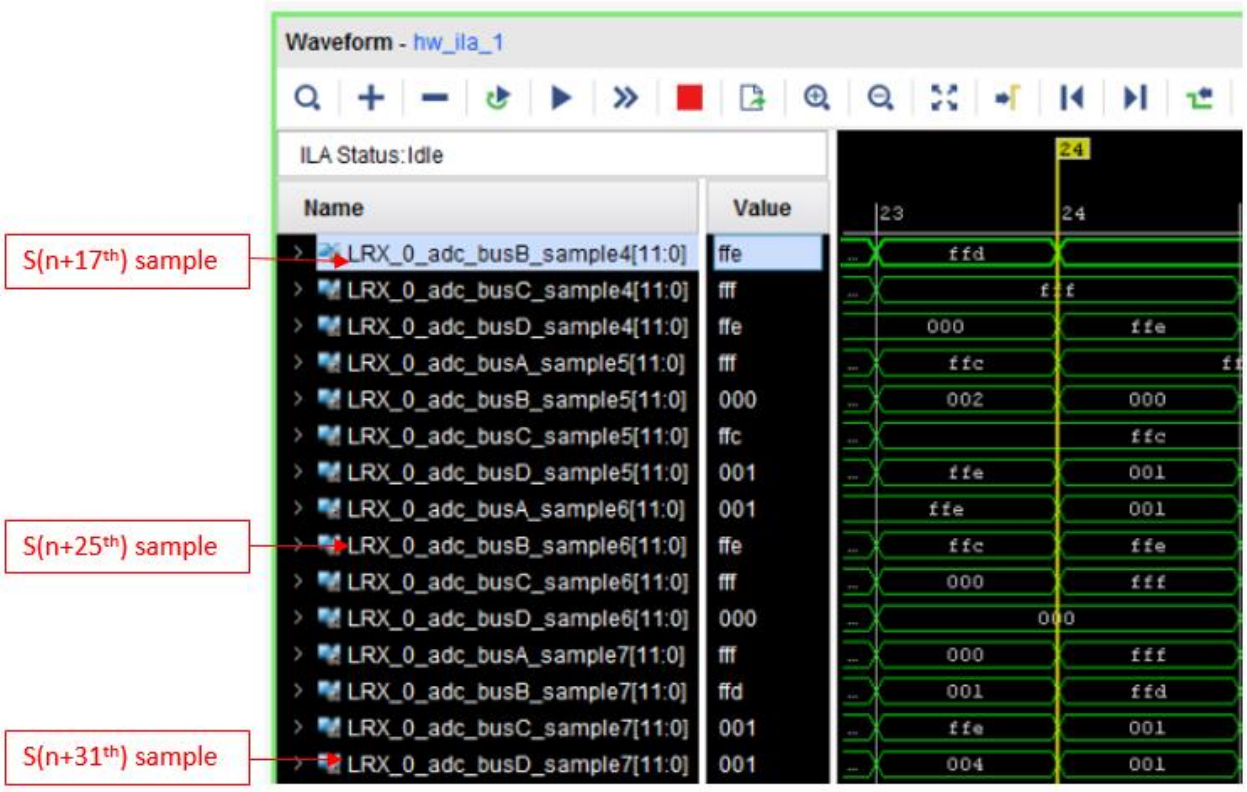


Fig 4.2(3)

Fig 4.2(4)

The signals are grouped in the chipscope window as samples per lane rate clock. Refer to timing diagram of single channel, 4bus mode in ADC12DL3200 datasheet.

Since the deserialization factor is 8, a single app clock will generate 8 samples per bus. Totally, 32 samples per deserialized clock (lane rate/ deserialized factor – ex: For 1.6Gbps data rate, the capture (or app) clock will be 200Mhz).

Valid signal from all the four buses will be at same instant due to multi-bank synchronization. Any one of the valid signals can be used for the future blocks to determine the valid sample from each bus interface.

5. Procedure for changing the lane rate

To re-use the reference design for different set of lane rates (0.3Gbps – 1.6Gpbs), below changes should be updated in the firmware.

I. Open LRX_0 IP with “Edit IP packager” option (Right click LRX_0 IP in lrx_hier).

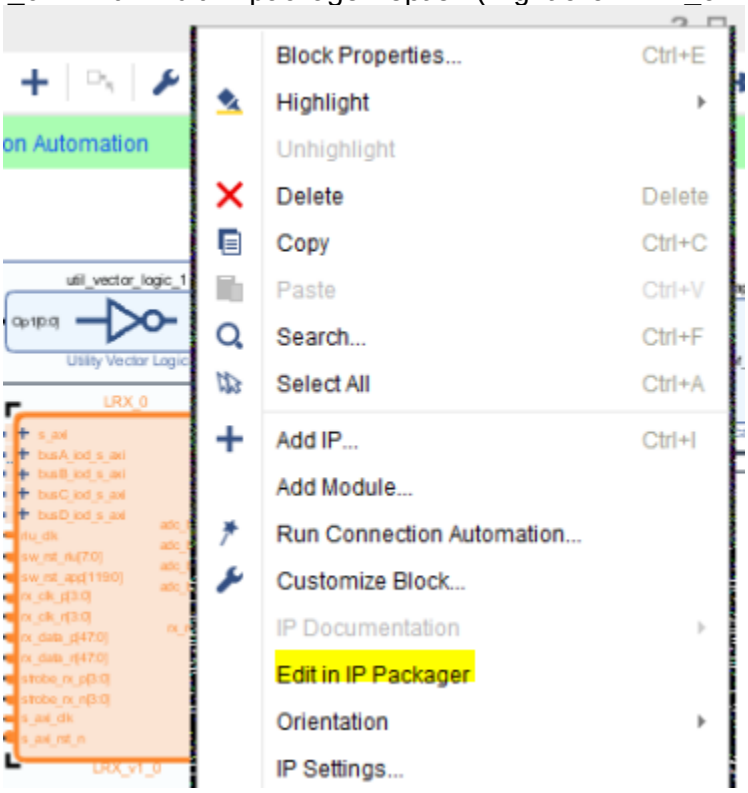


Fig 5(1)

II. Under sources tab, expand ch0_i module.

III. Double click on rx0_i module to edit the interface speed

Fig 5(2)

IV. Interface speed should be updated in High Speed SelectIO wizard and click “OK”

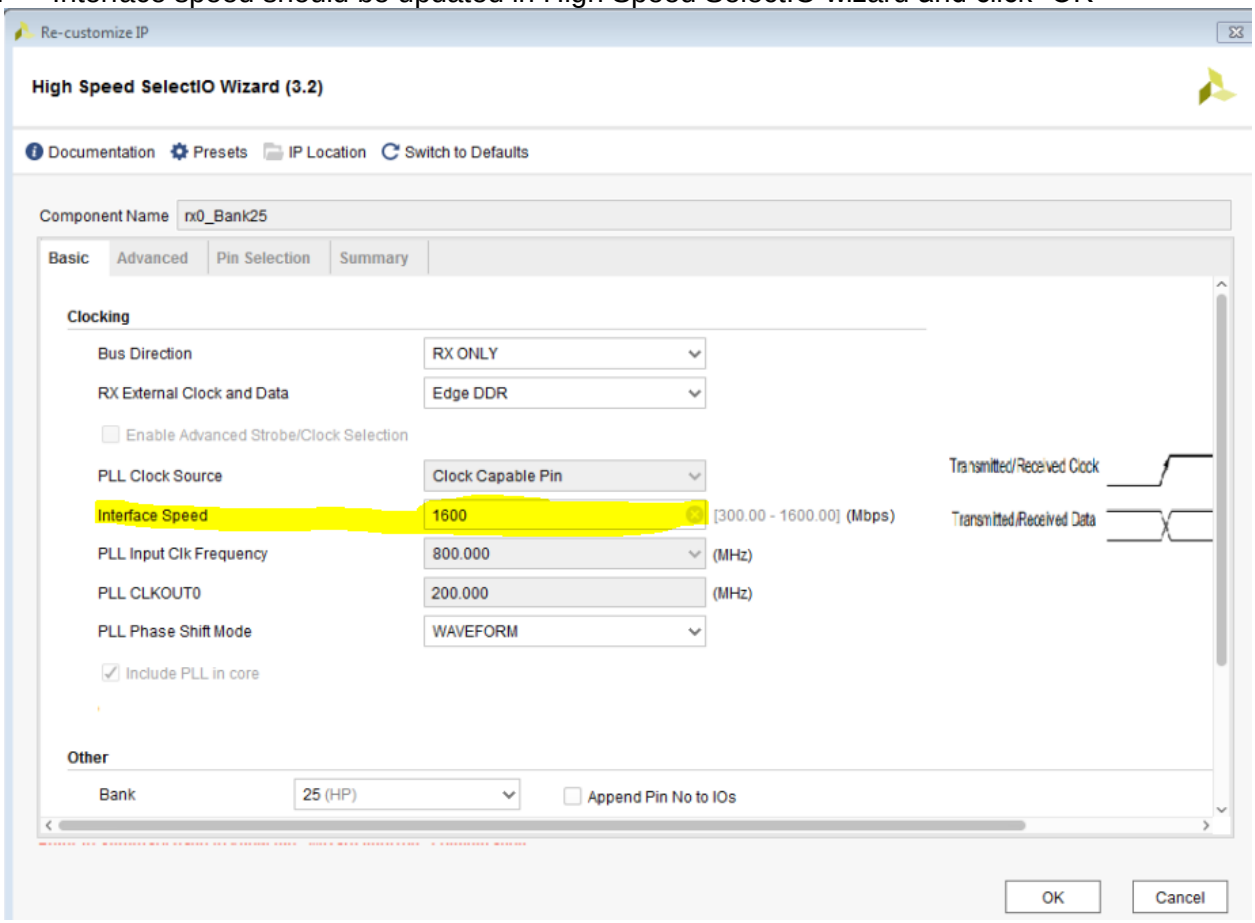


Fig 5(3)

V. Similarly edit the interface speed in
   a. rx1_i under ch1_i
   b. rx2_i under ch2_i
   c. rx3_i under ch3_i.

VI. Generate output products for all the HSSIO IP with OOC option enabled

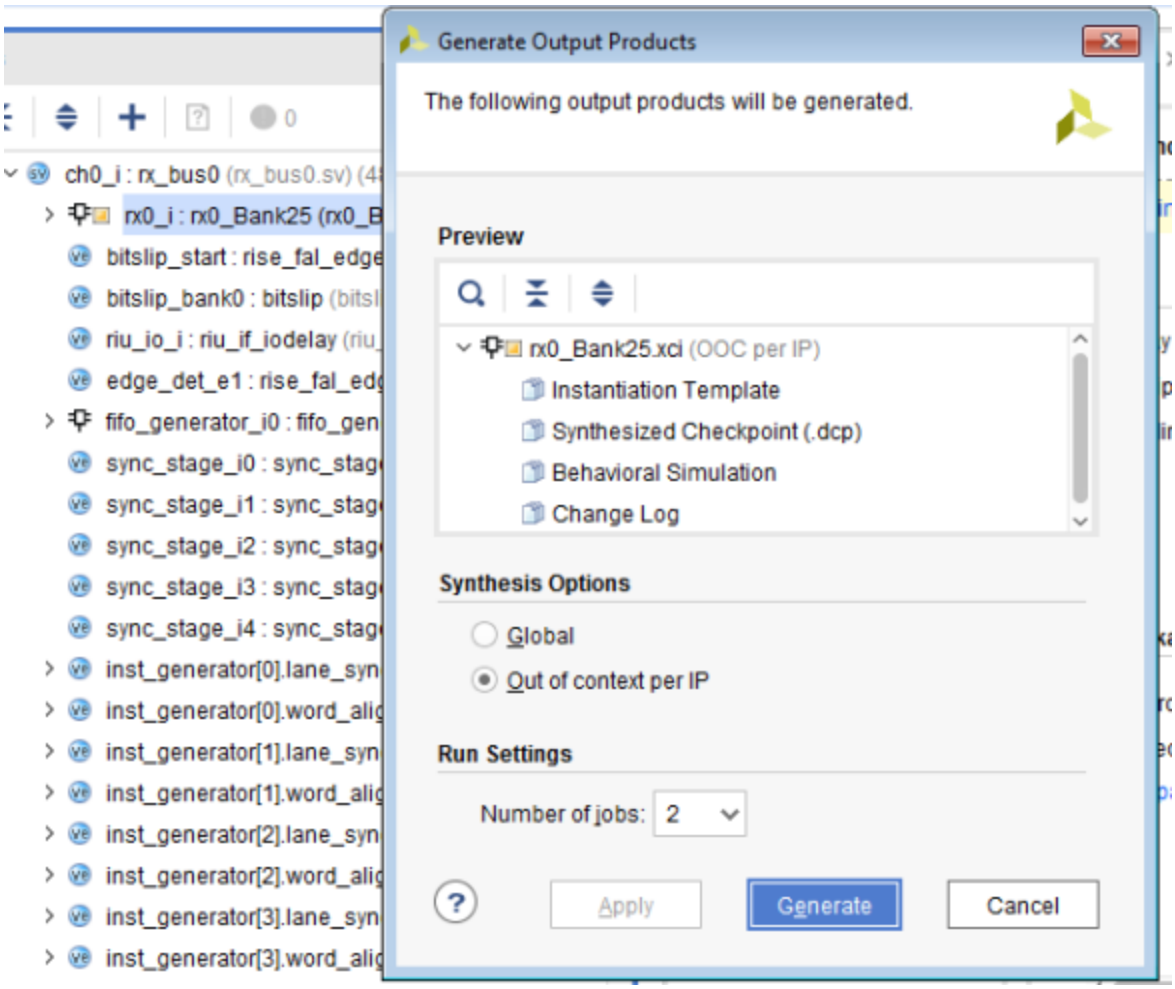


Fig 5(4)

VII. Update the clock in period of the rx_clk_p[*],rx_clk_n[*] as 2/lane rate (ns) in LRX module XDC as in fig 5(5)

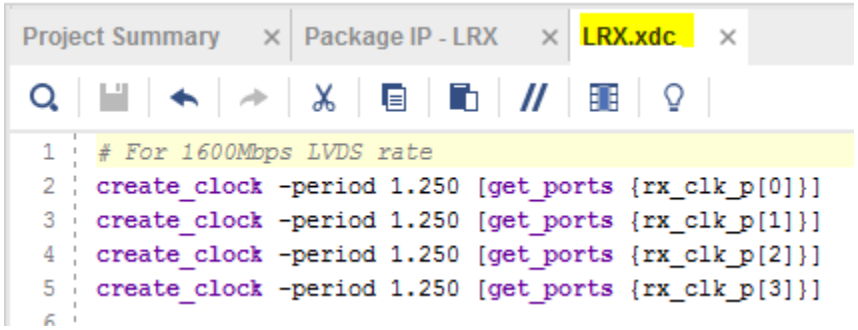
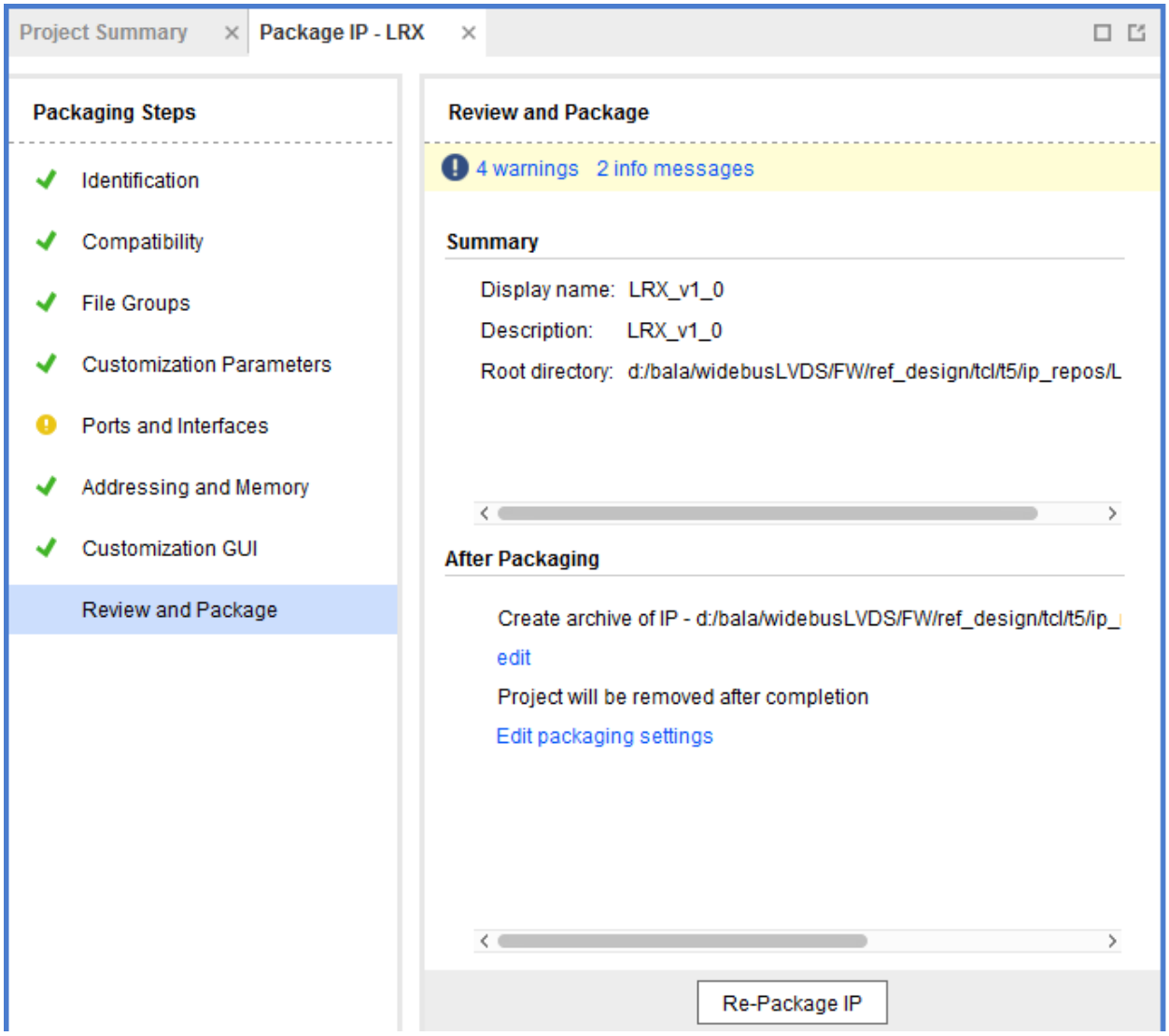


Fig 5(5)

VIII. Re-Package the IP as mentioned in UG1119

Note: busX_iod_s_axi register map clock should be associated with s_axi_clk before packing the IP



IX. Report IP status (tools -> Report -> report IP status)

X. Click “Upgrade Selected” in console window and generate output products with OOC enabled

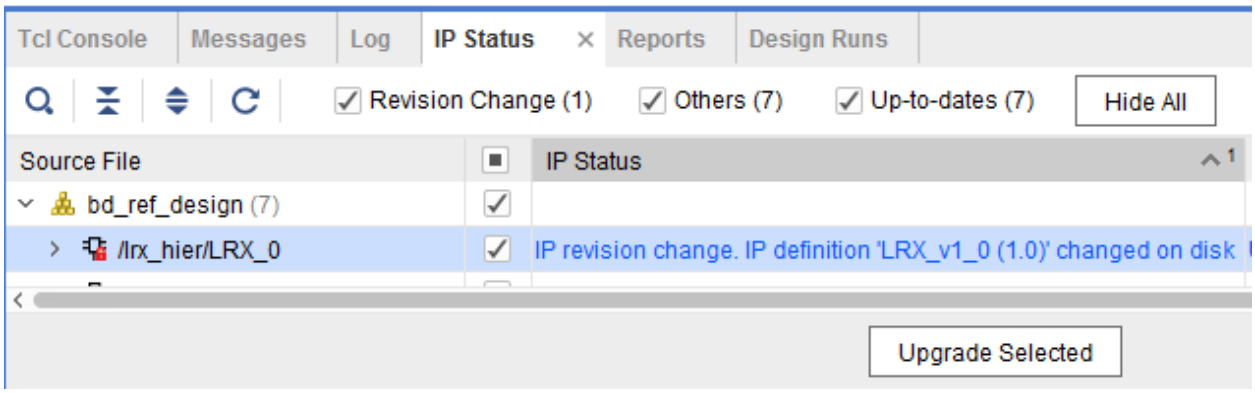


Fig 5(6)

XI. Update the clock in period of the rx_clk_p[*],rx_clk_n[*] as 2/lane rate (ns) in block diagram XDC as in fig 5(6).

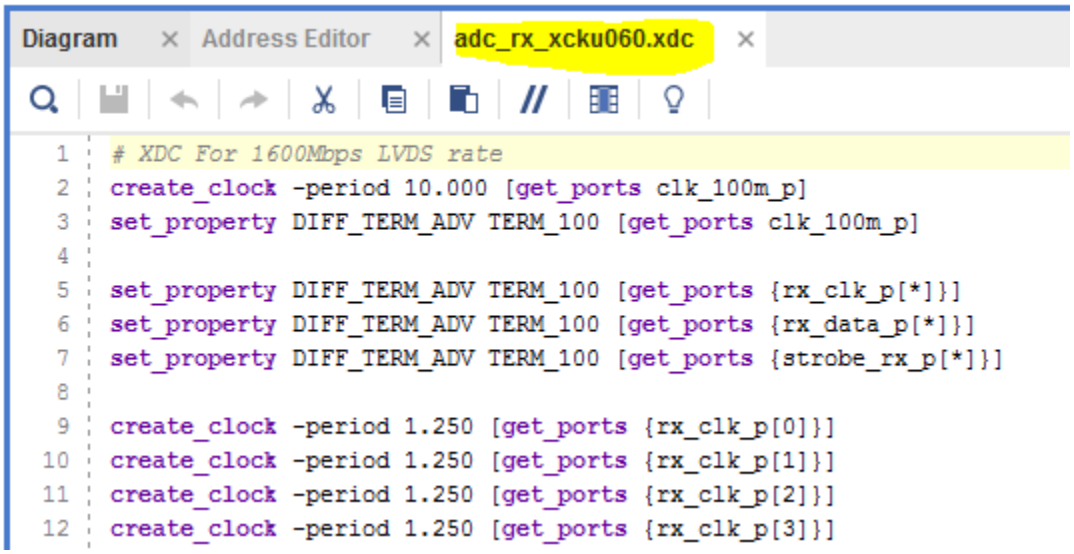


Fig 5(7)

XII. Refer section 2.4 to generate bitstream

6. Design and implementation of custom LRX IP

6.1 Module Overview

The lrx_hier is grouped with JTAG to AXI Master, Clock manager, AXI interconnect and LRX module. JTAG to AXI master IP core can generate AXI transactions and drive the slave AXI signals internal to FPGA with TCL commands. AXI interconnect is used to control the memory-mapped IO delay of each bus and clock manager is used to generate the necessary clocks and clock synchronous resets (riu_clk is used for IO delay operation).

The clocking architecture of the reference design in show in Fig 6.1(1)

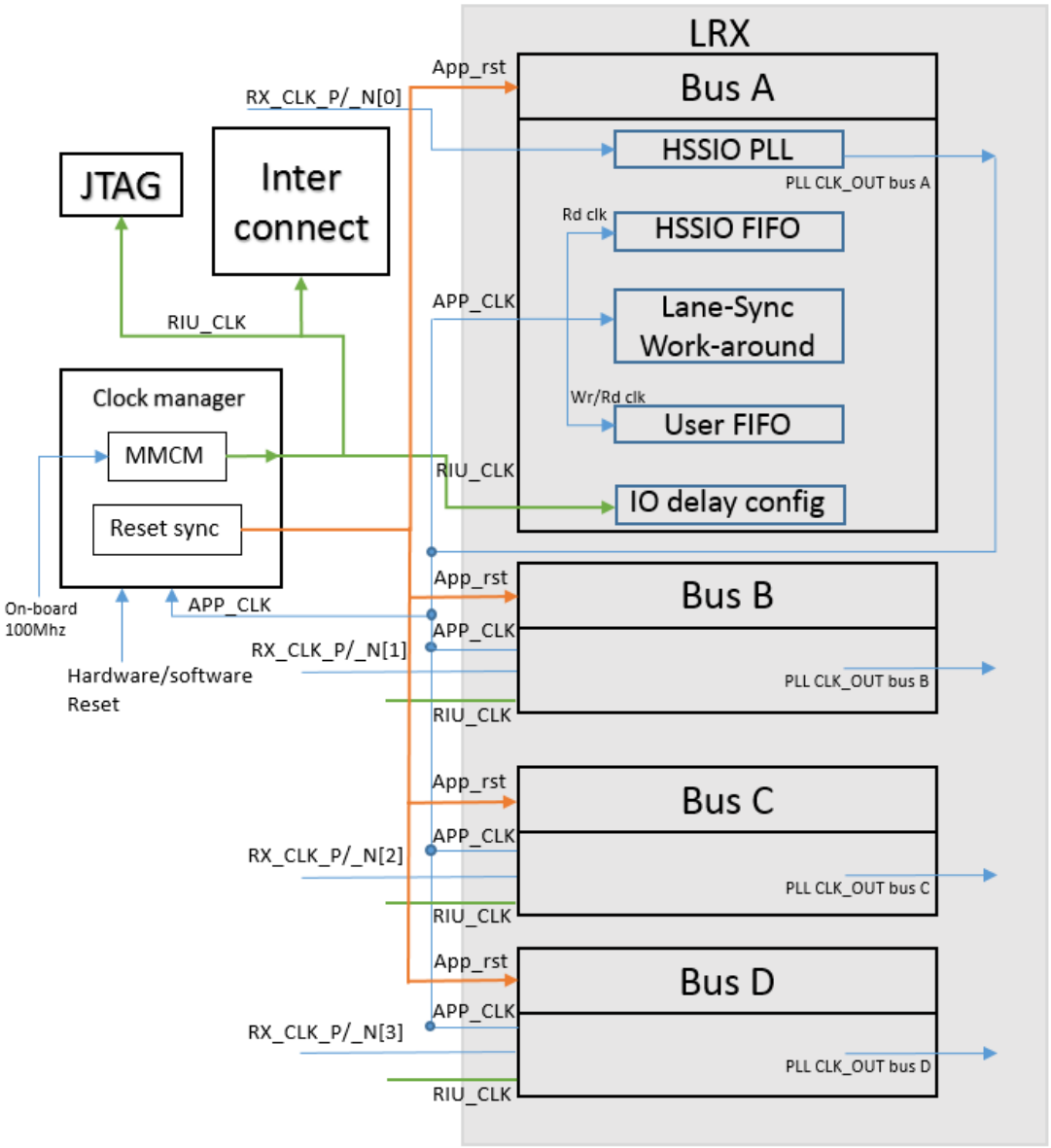


Fig 6.1(1)

The sw_rst port has been connected to hw_rst_n in this design.

Note: sw_rst signal can be utilized to reset the design via software by disconnecting it from hw_rst_n port and connecting it to user added control signal.

6.2 High Speed Select IO IP

HSSIO IP simplifies the integration of selectIO technology into high speed system designs for ultrascale devices. For more details refer to pg188 and UG571 documents of Xilinx.

Following are the IP configuration details used in this reference design:

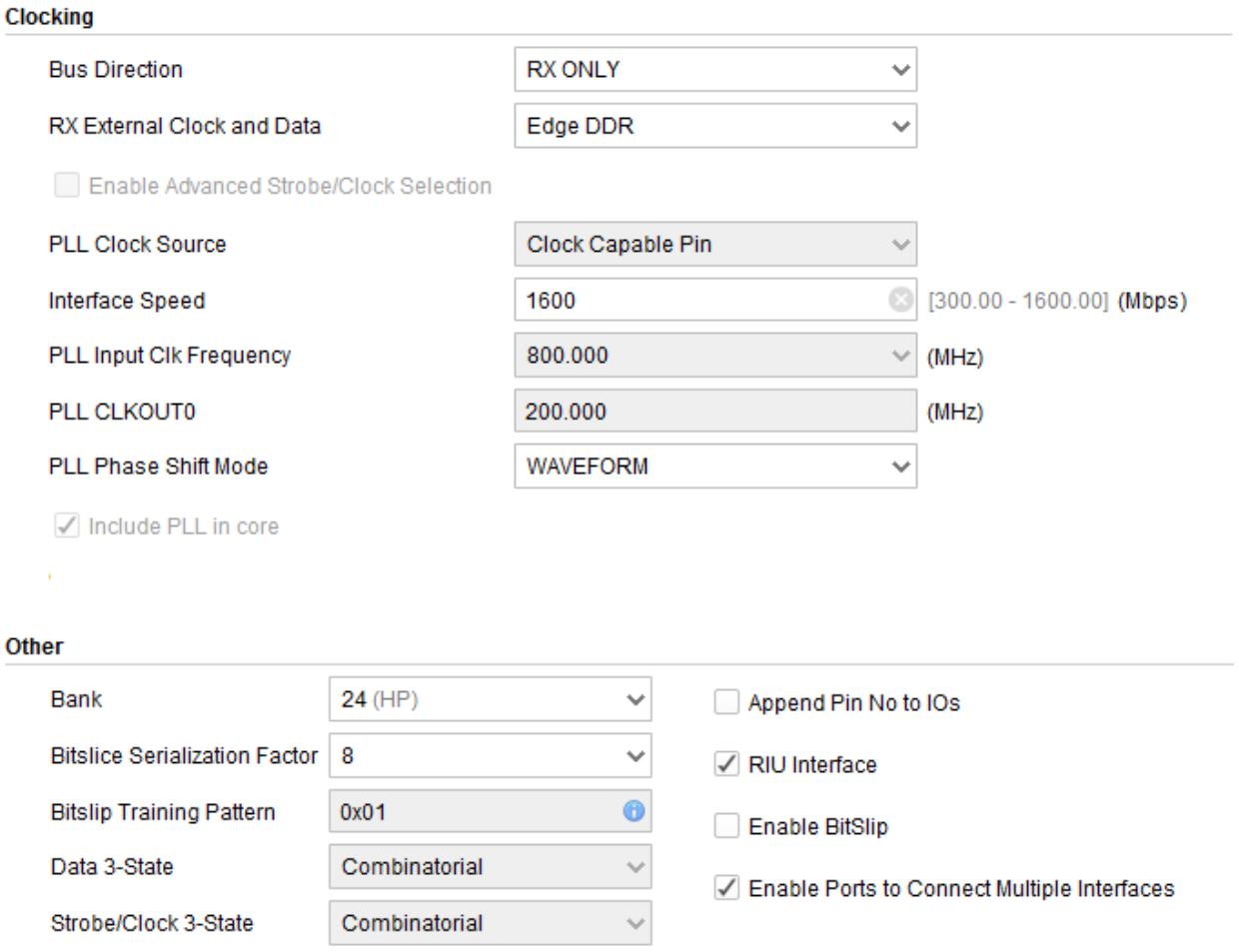


Fig 6.2(1)

Under ”Other” section each bank number is mentioned – In this design IO banks 24 ,25 ,44 and 45 are used to handle the data from 4 buses of ADC. In order to achieve multi bank synchronization, tick the “Enable Ports to Connect Multiple Interfaces” checkbox is in the IP. For more details on multi bank synchronization refer to AR68620

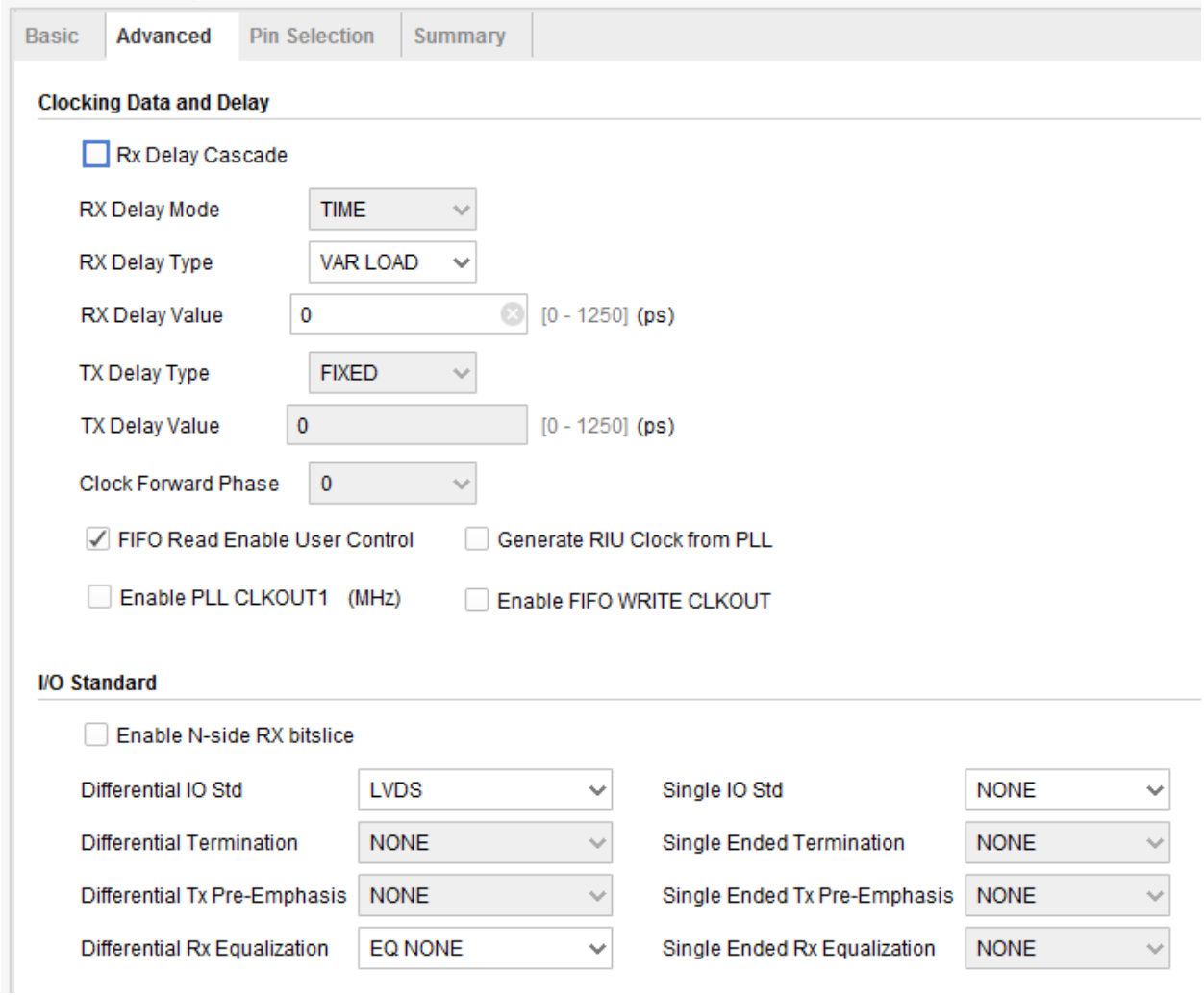


Fig 6.1(2)

For list of pins selected in each bank, refer to XDC file.

6.3 Lane sync workaround

The output of HSSIO IP has a lane synchronization problem which means deserialized output of different lanes from single bus are not aligned.

For example, say A*,B* and C* represents samples per lane rate clock in a bus

Fig 6.3(1) shows the typical scenario how data is expected from the HSSIO IP.

	sample 0	sample 1	sample 2
lane 11	A11	B11	C11
lane 10	A10	B10	C10
lane 9	A9	B9	C9
lane 8	A8	B8	C8
lane 7	A7	B7	C7
lane 6	A6	B6	C6
lane 5	A5	B5	C5
lane 4	A4	B4	C4
lane 3	A3	B3	C3
lane 2	A2	B2	C2
lane 1	A1	B1	C1
lane 0	A0	B0	C0

Fig 6.3(1)

But alignment will miss between the lanes randomly across the trails and causes synchronization issue like in fig 6.3(2)

	sample 0	sample 1	sample 2		
lane 11	A11	B11	C11		
lane 10	A10	B10	C10		
lane 9	A9	B9	C9		
lane 8	xx	A8	B8	C8	
lane 7	A7	B7	C7		
lane 6	A6	B6	C6		
lane 5	A5	B5	C5		
lane 4	A4	B4	C4		
lane 3	A3	B3	C3		
lane 2	A2	B2	C2		
lane 1	xx	xx	A1	B1	C1
lane 0	A0	B0	C0		

Fig 6.3(2)

Hence custom lane sync workaround has been implemented to address the issue. A constant 16bit test pattern has to be played across all LaneX to resolve the issue. (LaneX includes 12 data lanes + 1 strobe lane)

Lane sync workaround has been implemented in each bus with sub–modules as per in block diagram Fig 6.3(1).

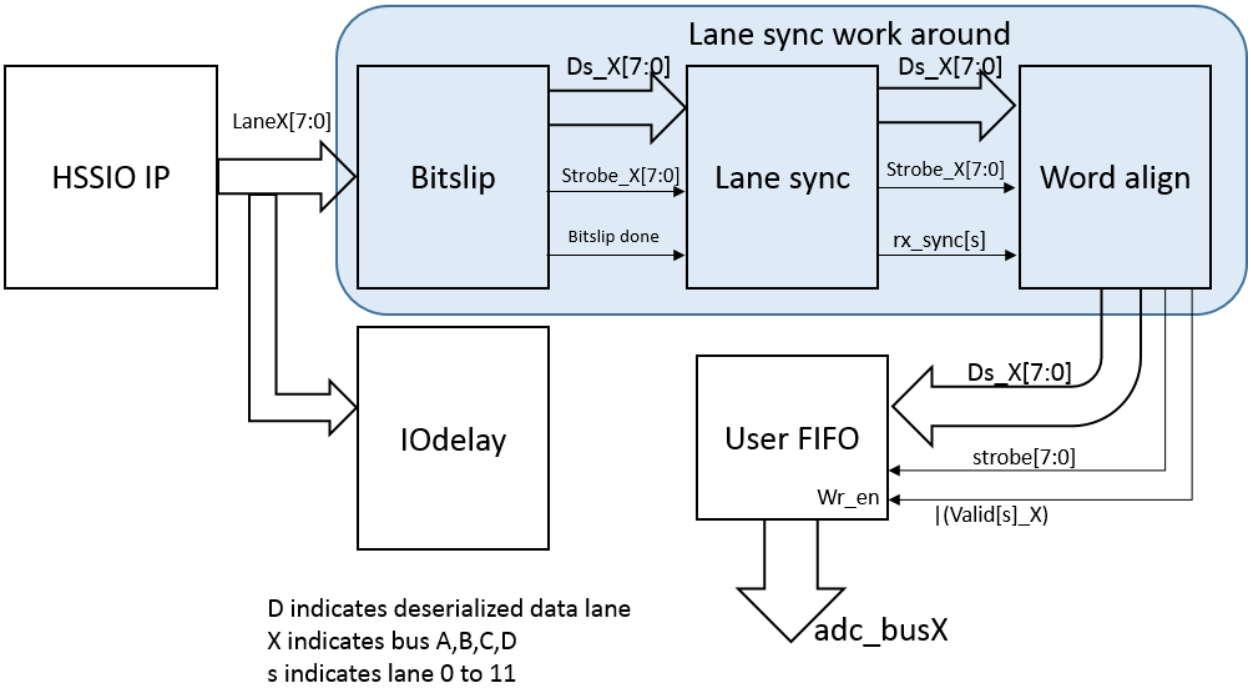


Fig 6.3(1)

Note: In order to align all the lanes, 16-bit test pattern is required to play across all the data lanes and strobe lanes.

6.3.1 Bit Slip

The module is used to align the deserialized strobeX with frame boundary. The delayed (n-1) and current(n) de-serialized strobeX are concatenated and compared with pre-defined 16bit value(test pattern) to find the number of bits shifted.

The list of comparison cases implemented in bitslip module is tabulated in fig 3.3.1(1) under “comparison statement”. In comparison, the number of bits shifted will be known and all the lanes are bit slipped by that number.

No. of bits shifted	Deserialized strobe possible way that can be received for Test pattern= h0080																Hex value	Comparison statement concat_strobe[*:*]==test pattern	output bit position
7 bit shift	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	h0001	{[8:0],[15:9]}==test pattern	14-:8
6 bit shift	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	h0002	{[9:0],[15:10]}==test pattern	13-:8
5 bit shift	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	h0004	{[10:0][15:11]}==test pattern	12-:8
4 bit shift	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	h0008	{[11:0],[15:12]}==test pattern	11-:8
3 bit shift	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	h0010	{[12:0],[15:13]}==test pattern	10-:8
2 bit shift	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	h0020	{[13:0],[15:14]}==test pattern	9-:8
1 bit shift	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	h0040	{[14:0],[15]}==test pattern	8-:8
No shift	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	h0080	[15:0]==test pattern	7-:8
	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7			
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**			
	n-1								**n**										

n = current clock cycle

concat_strobe = {strobe(n),strobe(n-1)}

Fig 6.3.1(1)

For example, consider one bit shift case and a test pattern 16'h0080 is played across all the LaneX.

In deserialized strobeX, the pattern received will be 16'h0040. This value is compared with all the cases in comparison statement and valid shift case will be detected (1 bit shift in this case).

On detection of valid shift case,"bitslip done" signal will be asserted and all the lanes will be shifted by 1 bit right (say output strobe = concat_strobe[8-:8] in this particular case).

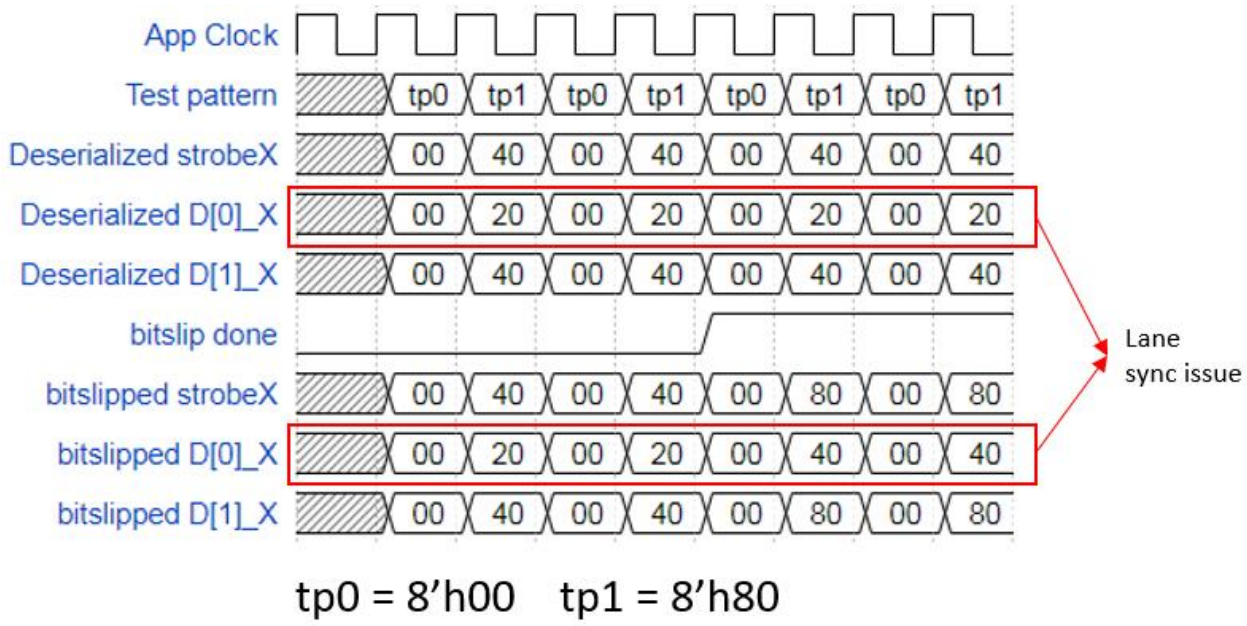


Fig 6.3.1(2)

6.3.2 Lane Sync

In lane sync module, each bitslipped Ds_X are aligned with bitslipped strobeX (refer to the fig 6.3(1) to understand the D, S, X).

After assertion of “bitslip done”, whenever strobeX[n-1] = tp0 and strobeX[n] = tp1 (n represents current clock cycle), Ds_X lanes are compared with test pattern. The comparison is done between concat_data [*:*] and test pattern (refer fig 6.3.2(2) for list of comparison cases), where concat_data = {Ds_X[n], Ds_X[n-1], Ds_X[n-2], Ds_X[n-3]}.

On comparison, whether data lane is delayed or ahead will be detected and then output bits are adjusted (as per in fig 6.3.2(2)) to align each data lane with strobe lane. After synchronizing data lane with strobe lane, rx_sync[s] signal will be asserted.

Fig 6.3.2(1) represents possible ways that data lane can be received for test pattern 16h’0080.

bitslipped Ds_X possible way that can be recevied for test patten 16'h0080

Ds_X(n-3)								Ds_X(n-2)								Ds_X(n-1)								Ds_X(n)								data in hex	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	00	01
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	00	02
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	00	04
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	00	08
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	00	10
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	00	20
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	00	40
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	00	80
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	01	00
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	02	00
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	04	00
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	08	00
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	10	00
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	20	00
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	40	00
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	15:8	23:16
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**		

Fig 6.3.2(1)

For example:

Consider fig 6.3.1(2), the D[0]_X is asynchronous to strobeX where D[1]_X is synchronous to strobe_X. On assertion of “bitslip done”, both the lanes will be compared separately with test pattern whenever StrobeX[n] = tp1 StrobeX[n-1] = tp0.

data in hex		Comparison statement concat_data[*:*]==testpattern	Synchronized lane Output bit postion
00	01	[16-:16]== test pattern	8-:8
00	02	[17-:16]== test pattern	9-:8
00	04	[18-:16]== test pattern	10-:8
00	08	[19-:16]== test pattern	11-:8
00	10	[20-:16]== test pattern	12-:8
00	20	[21-:16]== test pattern	13-:8
00	40	[22-:16]== test pattern	14-:8
00	80	[23-:16]== test pattern	15-:8
01	00	[24-:16]== test pattern	16-:8
02	00	[25-:16]== test pattern	16-:8
04	00	[26-:16]== test pattern	17-:8
08	00	[27-:16]== test pattern	18-:8
10	00	[28-:16]== test pattern	19-:8
20	00	[29-:16]== test pattern	20-:8
40	00	[30-:16]== test pattern	21-:8

Fig 6.3.2(2)

Four bitslipped D[0]_X (n-3,n-2,n-1,n)samples will be concatenated and compared with 16 bit test pattern as per in comparison statement of fig 6.3.2(2). (Fig 6.3.2(2) is continuation of fig 6.3.2(1))

On successful detection of comparison case (in this scenario h0040), rx_sync[0] will set to high and output of synchronized lane will be 14-:8 in 32bit concat_data of D[0]_X.

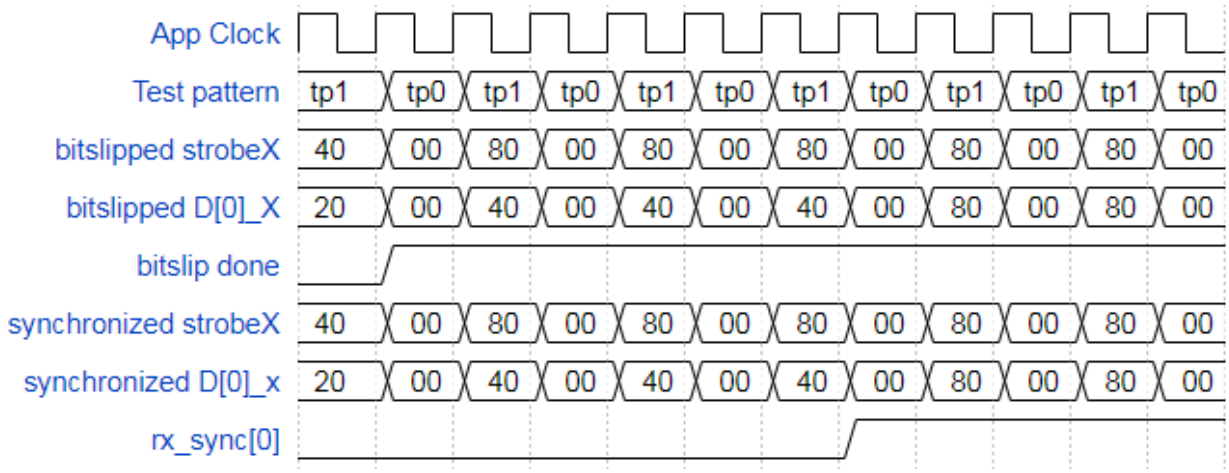


Fig 6.3.2(3) – D[0]_X synchronization with strobeX

Similarly for D[1]_X, h0080 case will be detected in comparison statement(refer fig 6.3.2(2)), the output of synchronized lane will be 15-:8 in 32bit concat_data of D[1]_X. On synchronization in D[1]_x, rx_sync[1] will set to high.

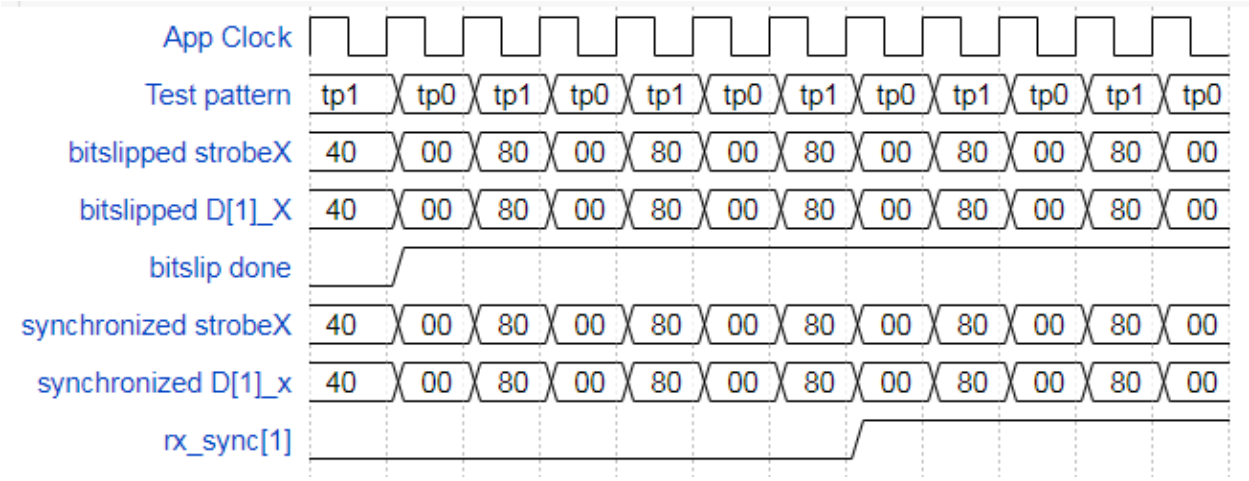


Fig 6.3.2(4) – D[1]_X synchronization with strobeX

Note: On assertion of rx_sync, ADC will stream output data in all the buses instead of frame strobe(test pattern) on all the lanes.

6.3.3 Word align

Mutli-bank synchronization is achieved with word align module and user FIFO (each bus has dedicated user FIFO) by writing 1st valid synchronized Ds_X into user fifo.

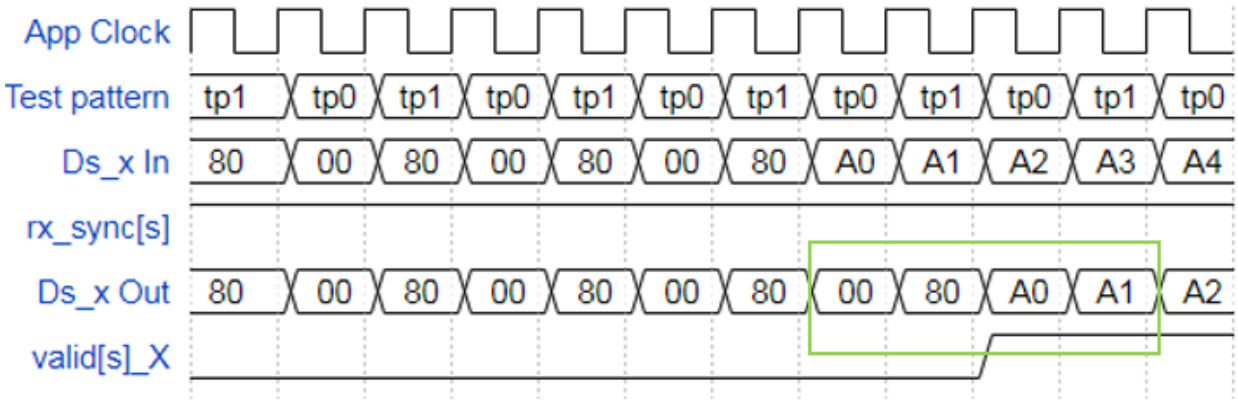


Fig 6.3.3(1)

The change in test pattern in data lanes per bus is considered as the 1st valid sample and is written to user fifo of that bus.

The change is test pattern is detected using below procedure

I. Wait till rx_sync[s] goes high, on high to go step ll.
II. For every 2 clock cycles, check concat_lane_sync_data == test pattern.
III. If comparison failed, goto step lV else goto ll
IV. Assert valid[s]_X to high

Where Concat_lane_sync_data = {Ds_X[n-1],Ds_X[n]}

User Fifo wr_en = |(Valid[s]_X)

Note: This design is currently build for test pattern 16'h0080. To change the test pattern, parameter STROBE_PAT1 and STROBE_PAT2 has to be modified in LRX.sv.

Appendix

A. IO delay

HSSIO IP in native mode support IDELAY3 primitive on each of its lanes. This section describes how the FW makes use of the VAR LOAD Mode of the IDELAY3 to implement the delay feature.

A.1 Implementation of IDELAY

The procedure to modify IO Delay is available in “VAR_LOAD Mode” section in UG571 document. It has been implemented as a state machine in the FW in iodelay_n_metastab_chk.v module.

In implementation, we have incremented the delay in steps of 8 taps at once. If incremented in step size greater than 8 taps, delay can become unstable and cause the delay to shift in to an unknown region. The increment by 8 method as briefed below has been recommended in *Xilinx Answer Record 67246*

I. Check the *difference* between existing delay and new delay required.
II. If difference >8, increment delay by 8 taps and continue to step 1, else step 3
III. Increment by *difference* taps and finish the update delay.

A.2 Metastability checking

Whenever a new value of delay is forced on the lanes, it is necessary to check if the data is stable and good at that value of delay. It is done in the Firmware using the check_metastability.v submodule in the iodelay_n_metastab_chk.v.

The check_metastability.v submodule does the following:

I. Wait for rising edge on EN_VTC
II. Capture 2 samples (16 bits of data) from the data lanes. This is the comparison_pattern. (Comparison pattern can be anything except all 0’s)
III. Check the incoming data vs the comparison_pattern. If mismatch occurs & if comparison_pattern is not all 0’s, a sticky error bit is set high.
IV. Clear error bit and go to step1 when EN_VTC falls.
V. This error bit can be read from the status register to check if a delay is good or not.

A.3 IO Delay Registers

In the FW, we have used dedicated AXI Register Map IPs for IO Delay of each bus. Through the register map, we specify the IO Delay commands and the command is executed accordingly in the FW.

IOD register address:

1. Register map address used in the FW for different banks is as below:

Bus	Interface used in HSSIO IP	Lane	Command (Control Register) Address	Read back (Status Register) Address
BUSX_iod_s_axi	IDELAY3	Lane 0	0xXXXX_0001 [LSB 16 bits]	0xXXXX_1001 [LSB 16 bits]
		Lane 1	0xXXXX_0001 [MSB 16 bits]	0xXXXX_1001 [MSB 16 bits]
		Lane 2	0xXXXX_0002 [LSB 16 bits]	0xXXXX_1002 [LSB 16 bits]
		Lane 3	0xXXXX_0002 [MSB 16 bits]	0xXXXX_1002 [MSB 16 bits]
		Lane 4	0xXXXX_0003 [LSB 16 bits]	0xXXXX_1003 [LSB 16 bits]
		Lane 5	0xXXXX_0003 [MSB 16 bits]	0xXXXX_1003 [MSB 16 bits]
		Lane 6	0xXXXX_0004 [LSB 16 bits]	0xXXXX_1004 [LSB 16 bits]
		Lane 7	0xXXXX_0004 [MSB 16 bits]	0xXXXX_1004 [MSB 16 bits]
		Lane 8	0xXXXX_0005 [LSB 16 bits]	0xXXXX_1005 [LSB 16 bits]
		Lane 9	0xXXXX_0005 [MSB 16 bits]	0xXXXX_1005 [MSB 16 bits]
		Lane 10	0xXXXX_0006 [LSB 16 bits]	0xXXXX_1006 [LSB 16 bits]
		Lane 11	0xXXXX_0006 [MSB 16 bits]	0xXXXX_1006 [MSB 16 bits]
		Strobe	0xXXXX_0008 [LSB 16 bits]	0xXXXX_1008 [LSB 16 bits]
	RIU	All lanes	0xXXXX_0007 [MSB 16 bits]	0xXXXX_1007 [MSB 16 bits]

Table 3.4.3(1)

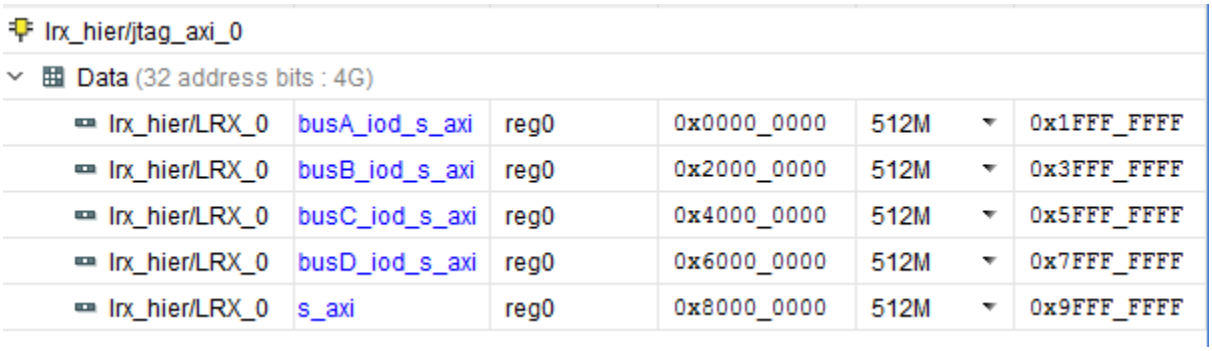


Fig 3.4.3(1)

The base address in Fig 3.4.3(1) can be edited in vivado GUI under address editor tab in block diagram

2. Command format (control register) for using the IO Delay is as below:

Lane x+1					Lane x														
31	30	..	..	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Same format as in [15:0]					Trig	Command ID			Reserved			User Delay								

Bit	Field	Type	Reset	Description
15	Trig	R/W	0h	Trigger for IO Delay – 0 by default, write as 1 to do any IO Delay related operation. This will be made 0 by FW after every IO Delay operation.
14:12	Command ID	R/W	0h	This is to denote the operation to be performed. It can be one of the four below. 000 = Read the current IO Delay value on the lane [Default command] 001 = Increment the delay by 1 tap 010 = Decrement the delay by 1 tap 011 = Update the delay to “User Delay”
11:9	Reserved	NA	0h	Reserved
8:0	User Delay	R/W	0h	Delay value to be written to the lane. [0 – 511 taps]

3. Read back format (status register) for IO Delay is as below:

Lane x+1					Lane x														
31	30	..	..	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Same format as in [15:0]					Done	Error	Delay modified	Reserved				Read Delay								

Bit	Field	Type	Reset	Description
15	Done	R/W	0h	Denotes that the issued IO Delay command has been completed
14	Error	R/W	0h	Indicates that there is metastability error at the specified delay value
13	Delay modified	R/W	0h	Denotes that the IO Delay has been modified from the default value after FW download.
12:9	Reserved	NA	0h	Reserved
8:0	Read Delay	R/W	0h	The current IO Delay value will be updated here when a “Read” Command is issued

Example to update or read the IO delay of busD – lane0 is explained below.

I. To read existing delay in a lane(READ_DELAY)

Step 1: Write 0x00000000 to register 0x6000_0001

Step 2: Write 0x00008000 to register 0x6000_0001

Step 3: Read the register 0x6000_1001 for current delay value in lane 0

(0x6000_1001[8:0] indicates lane 0 IO delay value)

II. To Increment the delay in a lane(INCREMENT_DELAY)

Step 1: Write 0x00001000 to register 0x6000_0001

Step 2: Write 0x00009000 to register 0x6000_0001

Step 3: Read the register 0x6000_1001 for current delay value in lane 0

(0x6000_1001[8:0] indicates lane 0 IO delay value)

III. To decrement the delay in a lane(DECREMENT_DELAY)

Step 1: Write 0x00002000 to register 0x6000_0001

Step 2: Write 0x0000A000 to register 0x6000_0001

Step 3: Read the register 0x6000_1001 for current delay value in lane 0

(0x6000_1001[8:0] indicates lane 0 IO delay value)

IV. To force update the IO delay value (UPDATE_DELAY)

Step 1: Write {16{1'b0}, 16'b{0011000,forced_delay_value[8:0]}} to register 0x6000_0001

Step 2: Write {16{1'b0}, 16'b{1011000,forced_delay_value[8:0]}} to register 0x6000_0001

Step 3: Read the register 0x6000_1001 for current delay value in lane 0

(0x6000_1001[8:0] indicates lane 0 IO delay value)

To update these values using JTAG interface, TCL commands are issued in vivado hardware manager using below format

create_hw_axi_txn trans_name **[get_hw_axis** hw axi handle number**] -type** write/read **–address** register_address **-len 1 -data {**register_value**} -force**

For more details, on using TCL commands through JTAG interface refer PG174.

For example

I. Write to a register

create_hw_axi_txn iodelay_lane0_wr [get_hw_axis hw_axi_1] -type write -address 6000_0001 -len 1 -data {00008000} -force

II. Read from a register

create_hw_axi_txn iodelay_lane0_rd [get_hw_axis hw_axi_1] -type read -address 6000_1001 -len 1 -force