SPI Communication and Synchronous Interface Constraining In FPGAs

Introduction

Serial Peripheral Interface (SPI) standard is widely used in the industry due to its simplicity and robustness. The standard requires only 4 wires, Master-Out-Slave-In (MOSI), Master-In-Slave-Out (MISO), SPI Clock (SCLK) and Chip-Select (CS) to transfer data, and provides different modes of operation based on forwarded clock polarity and phase. The task is to establish a SPI link between two Zedboards (Zynq 7020 chipset), with one acting as the master which sends encoded data over to the slave over SPI.

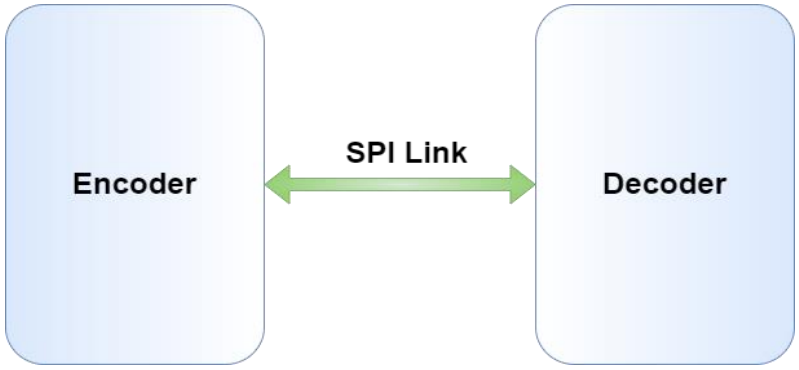

Figure 1. Encoder - Decoder communication

A simple SPI mode (mode 0), with data shift-out/sampling on negative/positive edge of SPI clock respectively as shown in *figure 2* is chosen to realize the link between the two Zedboards. The goal is to study the impact various resource elements within the FPGA have on establishing a dependable link. A secondary goal is to design the two units with flexibility and compactness in mind, that is, they should be able to operate on a wide range of transmission speeds and scenarios. Before we begin designing the modules, we must adhere to some design standards to avoid potential pitfalls concerned with FPGAs. We will explore these guidelines in the following section.

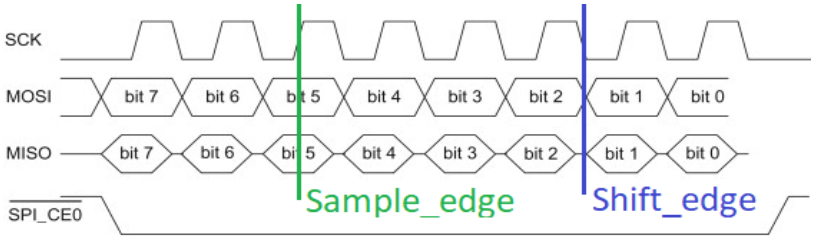

Figure 2. SPI transaction waveform (mode 0)

Design Guidelines

Clock and I/O resources:

Before starting any design, it is important to obtain a good grasp of the resources available within the FPGA at our disposal. Hence, it is recommended to read the '*Clocking Resources*' and '*Select I/O Resources*' user guides for the '*7 Series*' (since we are using a zynq 7020 SoC) FPGAs. Check out the following links:

- https://docs.xilinx.com/v/u/en-US/ug472_7Series_Clocking
- https://docs.xilinx.com/v/u/en-US/ug471_7Series_SelectIO

Asynchronous Resets:

Asynchronous resets are widely discouraged since their presence may lead to erratic behavior of flops if the asynchronous signal fails to adhere to the recovery and removal times of the flop. It is recommended to use synchronous resets wherever possible and use asynchronous resets only if the flop is not clocked during the de-assertion of async-reset. If an asynchronous reset is necessary in the presence of a clock signal, it is recommended to use a "Synchronous-Asynchronous Reset". To know more about the pitfalls of asynchronous resets and how to implement a sync-async reset, please refer to the following links:

- https://www.01signal.com/verilog-design/reset/
- https://www.intel.com/content/www/us/en/docs/programmable/683082/21-3/use-synchronized-asynchronous-reset.html

Flip-Flops Controlled by Asynchronous Signals:

Vivado discourages asynchronous signals driving control pins of synchronous elements. For low frequency designs this may not pose an issue since the time period is large enough to let the asynchronous signals stabilize, but in the case of high frequency circuits, this could lead to issues with timing, consequently affecting register place and route. Driving control signals directly from a LUT can cause metastability due to spurious behavior of signals around the clock edge. So make sure all asynchronous reset signals are driven by registers and not by LUTs, and in-case there exists an combinational logic (LUT) along the path of the control signal, the path must be timed to make sure the signal stabilizes before the next clock edge https://support.xilinx.com/s/feed/0D52E00006hpikKSAQ?language=en_US .

Inferring Flip-Flops in Vivado:

Vivado synthesis only has 4 kinds of flip-flops that will be inferred from HDL code, they are as follows:

- FDCE – D flip-flop with clock enable and Asynchronous Clear.
- FDPE – D flip-flop with clock enable and Asynchronous Preset.
- FDSE – D flip-flop with clock enable and Synchronous Set.
- FDRE – D flip-flop with clock enable and Synchronous Reset.

Anything else will not be inferred as a flip-flop and may end up being inferred as a Latch (LDC). For example, in Synopsis, the following HDL snippet will be realized as a flip-flip with asynchronous clear and preset.

```
always @(posedge clk, posedge async_set, negedge async_reset) begin
        if (!reset) …
        else if (set) …
        else …
end
```

But Vivado will synthesize the above code into a latch with clock going to the enable/gate pin of the latch. So, take care to not use coding style shown above and make sure the intention of the HDL code is to realize one of the available flip-flops (FDCE, FDPE, FDSE, FDRE) in Vivado.

Input/Output Block (IOB) Register Packaging:

There should be no combinational logic between the register and output port in-order for the register to be placed successfully in IOB. If an input port is connected to two or more registers, only one of them will

be placed into the IOB. A simple rule is for registers to be placed in IOB, make sure the register of interest is the last element in the path for output ports and the first one in the case of input ports. For more information on the rules for packaging a register successfully into an IOB, please refer to:

- https://support.xilinx.com/s/article/66668?language=en_US
- https://gist.github.com/brabect1/7695ead3d79be47576890bbcd61fe426

FPGA Clock Forwarding:

Any clock signal generated within the FPGA must go through the ODDR (Output Double Data Rate) register in the IOB if it must leave the FPGA board. This is to ensure the clock is routed through the clock dedicated resources instead of generic signal interconnect. This ensures there is minimal skew and keeps the forwarded clock clean. Vivado will not throw an error/warning if the clock is directly forwarded without and ODDR, however this will end up routing the clock through the interconnect. For low frequency clocks this is not an issue as the tolerances on the receiver are higher, but ODDR is a must for higher frequencies.

SPI – A Pseudo Source-Synchronous Interface

In a *'Source-Synchronous Interface'* the transmitter's data is sent in tandem with a clock signal that will be used by the receiver to sample the data as shown in *Figure 3* below. Source-Synchronous Interfaces can achieve higher operation speeds compared to a *'System-Synchronous Interface'*. The SPI protocol is source synchronous with respect to transmitter/master (encoder) since the master provides both data and a synchronous SPI clock signal to sample said data at the receiver/slave (decoder). When seen with respect to decoder, SPI may or may not be source-synchronous depending on the frequency of operation.

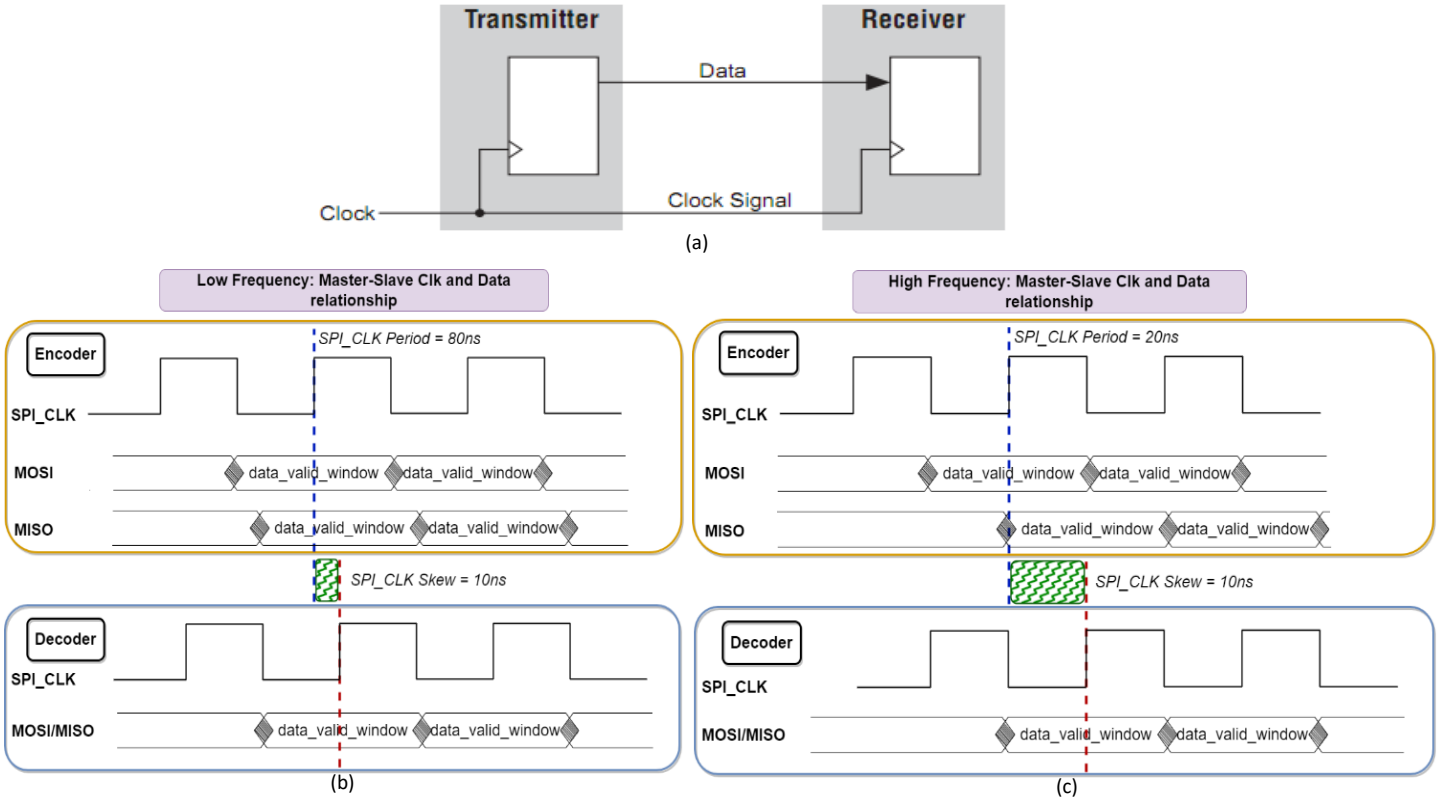

Figure 3. (a) Simple Source-Synchronous Interface representation. (b) Low frequency master-slave SPI clock and data relationship, blue and red lines define the encoder and decoder sampling edges. (c) High frequency master-slave SPI clock and data relationship.

For lower frequencies where the clock path, port, buffer delays contribute to the marginal clock skew at the decoder side, the *'MISO'* data sent back by the decoder is still synchronous with the source (encoder) SPI clock as shown in *Figure 3 (b).* But, for higher frequencies the skew is considerable and the *'MISO'* data is no longer synchronous with the source SPI clock as shown in *Figure 3 (c)*, also it should be noted that this skew results in the master sampling at the *data invalid window*, resulting in wrong data. Hence, at higher frequencies SPI can only be considered as source synchronous with respect to master (encoder).

Encoder Block Design

Overview:

The first part will be designing the encoder, which will be used to validate the decoder design down the line. So, it is important to make the encoder flexible and user interactive to trigger different SPI link test cases. The simplest way to achieve user interactivity is to use the Zynq Processing System (PS) to run an application that will generate the necessary test signals. An AXI-SPI interface is required to act upon the signals received from the PS. The AXI-SPI module will be placed in PL part of the FPGA and will drive the SPI signals outside the board. This can be easily achieved using the IP packager to generate a AXI-lite IP

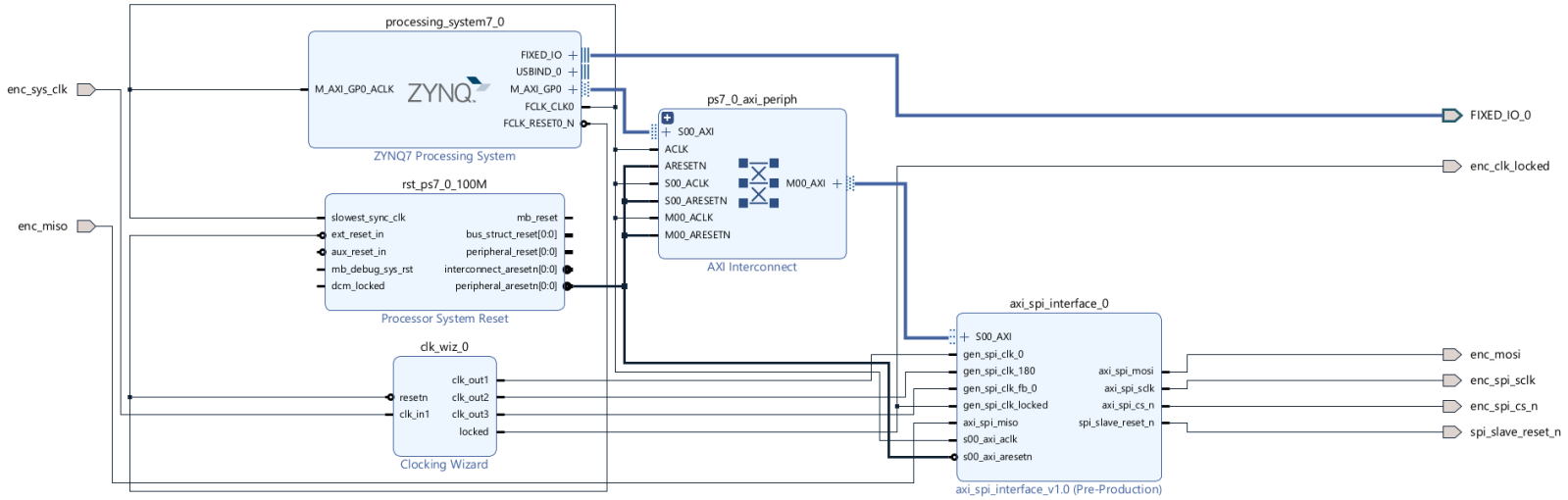

Figure 4. Encoder block design.

(shown in the encoder system block diagram below as "axi_spi_interface_0") and integrate the SPI master driver within the generated IP.

AXI – SPI Interface:

The encoding functionality is delegated to the Zynq processing system with the focus of the AXI-SPI Interface being SPI-Communication. The encoded data is transferred to the interface over the AXI-Lite protocol and the corresponding control bits are set in the control register (slave register 0). The SPI transaction is initiated by toggling the 'start' bit high and low in the control register. The start of an SPI transaction will pull the 'busy' control bit high and will go low at the end of the transaction. The busy bit is constantly polled by the PS to know when a transaction completes. *Table 2* provides the register map and their descriptions. The SPI clock that will be forwarded to the slave is generated by a MMCM block (Clocking Wizard in Figure. 3) to provide a clean and stable clock. The SPI clock is forwarded using an ODDR as specified in the design guidelines.

Table 1: AXI-SPI Interface Pin Description.

Pin Name	In / Out	Description
S00_AXI	In	Interface AXI-Lite slave connection.
S00_axi_clk	In	AXI clock for slave interface.
S00_axi_aresetn	In	AXI interface asynchronous reset.
gen_spi_clk_0	In	MMCM generated clock that will be forwarded out the FPGA. The '0' in the name indicates it is in phase with the forwarded clock.
gen_spi_clk_180	In	MMCM generated clock that is used to generate/shift SPI data. The '180' indicates it is 180 degrees out of phase with gen_spi_clk_0.
gen_spi_clk_fb_0	In	MMCM generated clock used to sample the MISO line.
gen_spi_clk_locked	In	MMCM clock locked signal to indicate AXI-SPI interface all generated clocks are stable.
axi_spi_miso	In	SPI MISO line from slave (decoder).
axi_spi_mosi	Out	SPI MOSI line to slave.
axi_spi_clk	Out	Forwarded SPI clock to slave (ODDR output).
axi_spi_cs_n	Out	SPI Chip-Select signal to enable slave SPI.
spi_slave_resetn	Out	Asynchronous reset signal to reset slave SPI interface. Used to force a slave spi interface reset.

The SPI interface is source synchronous with respect to encoder, but is not with respect to decoder since there is no reference clock with its data. At low frequencies the encoder can simply sample the *'MISO'* line with respect to its *'gen_spi_clk_0'* but, for higher frequencies the trace delay plus the internal buffer delays on the decoder's SPI clock input will cause the decoder's *data valid window* to shift by a considerable margin with respect to the SPI clock's time period, hence the same clock used by the encoder to send data cannot be used to sample data from decoder since it does not compensate for the delay in the clock-to-output of decoder. There are two ways to compensate for this, first one is to send the SPI clock signal back from the decoder but then this would no longer be an SPI interface since an extra signal is being added to the protocol. The second is to compensate for the delay on the encoder side by emulating the feedback clock with a suitable phase shift with respect to *'gen_spi_clk_0'* in hardware which will position it in the middle of decoder's *data valid window* and this is done by providing a separate *'gen_spi_clk_fb_0'*.

Table 2: AXI-SPI Interface Register map

Register Name	R/W	Size (bits)	Bit(s)	Description
Slv_reg0	R/W	32	0	AXI-SPI interface and Decoder asynchronous reset signal. Set to 1 to assert reset signal.
			1	SPI Transaction start bit. Set this to 1 and back to 0 to start a SPI transaction.
			2	Set to 1 to enable Free running SPI clock. When 0, the SPI clock is forwarded only during a SPI transaction, when 1, the

				clock is free running. Free running clock is only necessary if PLL is used on the decoder to retrieve SPI clock.
			[3 : SPI_CLK_FB_DELAY_WIDTH-1]	Sets the number of MISO bits to ignore. Refer to the AXI-SPI interface definition and Table 3 (parameters) for context.
			[SPI_CLK_FB_DELAY_WIDTH : 31]	Free for later use.
Slv_reg1	R/W	32	[0 : SPI_WR_ADDR_WIDTH-1]	Address where MOSI data will be written to on the decoder.
			[SPI_WR_ADDR_WIDTH : SPI_RD_ADDR_WIDTH-1]	Address of data from decoder register bank.
			[SPI_RD_ADDR_WIDTH : 31]	Free to use.
Slv_reg2	R/W	32	[0 : 31]	SPI Write data. Data to be written to decoder.
Slv_reg3	R	32	[0 : 31]	SPI Read data. Data read from decoder.

The AXI-SPI interface IP parameters can be configured to obtain desired SPI behavior. The different parameters are described in *Table 3* and the following are some of their use-cases:

- For slower SPI clocks, the *spi_cs_n* signal needs to go low half a *unit-interval* (the time period between data changes) prior to the start of SPI clock. But for higher frequencies, *spi_cs_n* would need to go low earlier to accommodate for the asynchronous signal sampling rate of the decoder. This interval of when the *spi_cs_n* signal goes low and high with respect to the start/stop of SPI clock signal is determined by the *'SPI_Init/Exit_interval'* parameters.
- Another useful parameter is the *'SPI_Clk_Fb_delay_width',* coupled with the *'aspi_set_fb_delay'* function (described in the Vitis section) lets the user dynamically set the number of *'MISO'* bits to ignore in the beginning in an SPI transaction. For example, consider the case in which the feedback clock is situated within the *data valid window* of the decoder by phase shifting *'gen_spi_clk_fb_0'* in hardware, but the first valid bit on the *'MISO'* line does not appear for a couple cycles due to the delay in SPI clock seen by the decoder. These invalid bits can be ignored through dynamically setting the *'aspi_set_fb_delay'* function from Vitis application. The *'SPI_Clk_Fb_delay_width'* determines the number of bits used to represent this delay in hardware, hence describes the upper-bound on the total number of bits that can be ignored from application layer.

Table 3: AXI-SPI Interface Parameter Definitions.

Parameter	Description
SPI Read Address Width	The width of read address sent to decoder.
SPI Write Address Width	The width of write address sent to decoder.
SPI Init Interval	The interval between *spi_cs_n* going low and first SPI clock rising edge appearing on output, in terms of number of SPI clock neg. edges passed before SPI clock appears on output.
SPI Exit Interval	The interval between last SPI clock falling edge appearing on output and *spi_cs_n* going high, in terms of number of SPI clock neg. edges passed after SPI clock disappears from output.

SPI Dummy-bits Beg	Number of dummy bits sent out at the beginning of an SPI transaction.
SPI Dummy-bits End	Number of dummy bits sent out at the end of an SPI transaction.
SPI Shift-In Com	Sets hardware *'MISO'* bit read compensation.
SPI Clk_Fb_delay_width	Sets upper bound for software *'MISO'* bit read compensation.

Master SPI Driver:

The SPI driver handles transmission and receive separately to avoid erroneous communication due to difference in clock-paths between '*MOSI*' and '*MISO*' lines (refer to '*SPI – A Pseudo Source-Synchronous Interface*' section for details on clock skew and resulting data valid window mismatch) and keep them independent of each other. There are no explicit Clock Domain Crossing registers between the clock

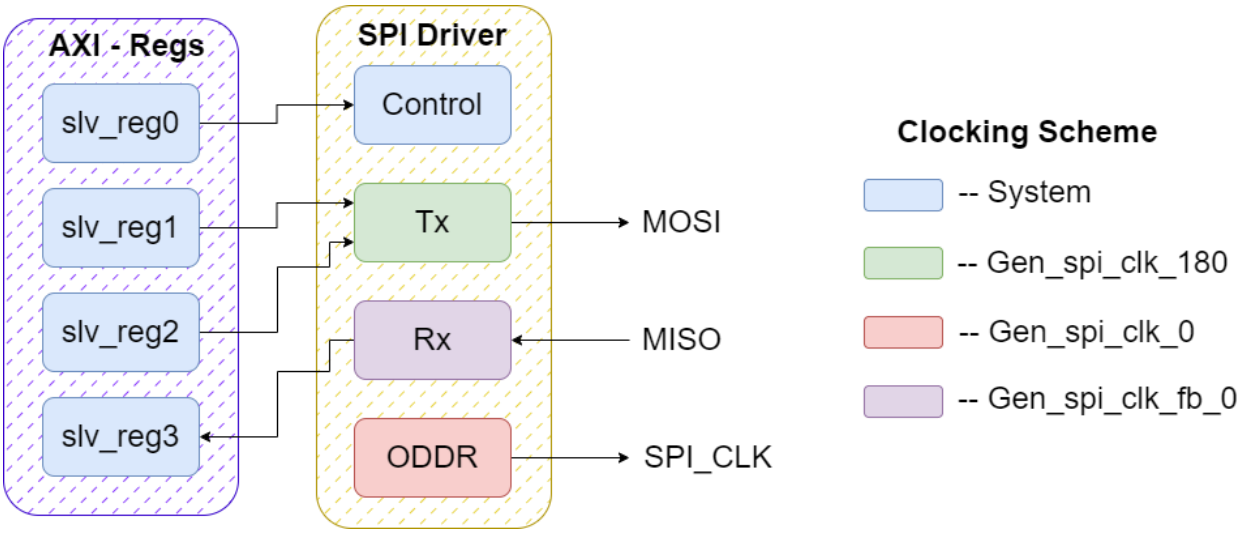

Figure 5. SPI Master Driver interaction and clock scheme.

domains since they are all related clocks (generated by the MMCM). The driver operation is straight forward, the output buffer combines the read and write address along with the write data. The output buffer shifts out its contents onto the '*MOSI*' at every positive edge of '*gen_spi_clk_180*', similarly the input buffer shifts in data from the '*MISO*' line at every positive edge of '*gen_spi_clk_fb_0*'. The SPI transaction begins once the '*start*' bit from '*slv_reg0*' toggles from 1 to 0, priming the respective buffers with initial values and enables the '*ODDR*' to forward the data-centered SPI clock.

Encoder Artifacts:

Communication was non-deterministic without asserting the async-reset signal, leading to the belief that there was wrong bit (from previous transaction) propagation among the input buffers on the decoder (described later in "*Decoder Block Design*"). So, the decoder spi_driver's behavior was changed to synchronously reset at the end of each SPI transaction to bring the registers to a clean state. But, the problem persisted, which directs us to the source of origin, the encoder. In the encoder, every buffer/register upon asynchronous reset goes to a known state, but that is also true during the start of every SPI transaction when the 'start' signal is asserted. All registers are primed when the 'start' signal is asserted, which is similar in behavior to when they are reset asynchronously. The only register that was

reset purely based on the async-reset signal was the ODDR used to drive the clock of-board. So, the reset signal was replaced with the negated clock enable signal, which resolved the issue.

Decoder Block Design

Overview:

The decoder is designed with compactness in mind. The slave driver should be lightweight so that it may fit onto any module to add SPI capabilities to it. Hence a reactive approach is taken towards designing the unit by avoiding the explicit use of FSM and try to make the FSM implicit to the design itself, but encourages the use of pipelining data/control signals to forward information to downstream units. The design tries to minimize Clock Domain Crossing wherever possible and in the case of CDC between asynchronous clocks, ensures that each signal (control and data) is sampled at least twice, once on rising edge and once on falling edge in the destination clock with discrepancy detection logic placed along data signals to ensure deterministic behavior.

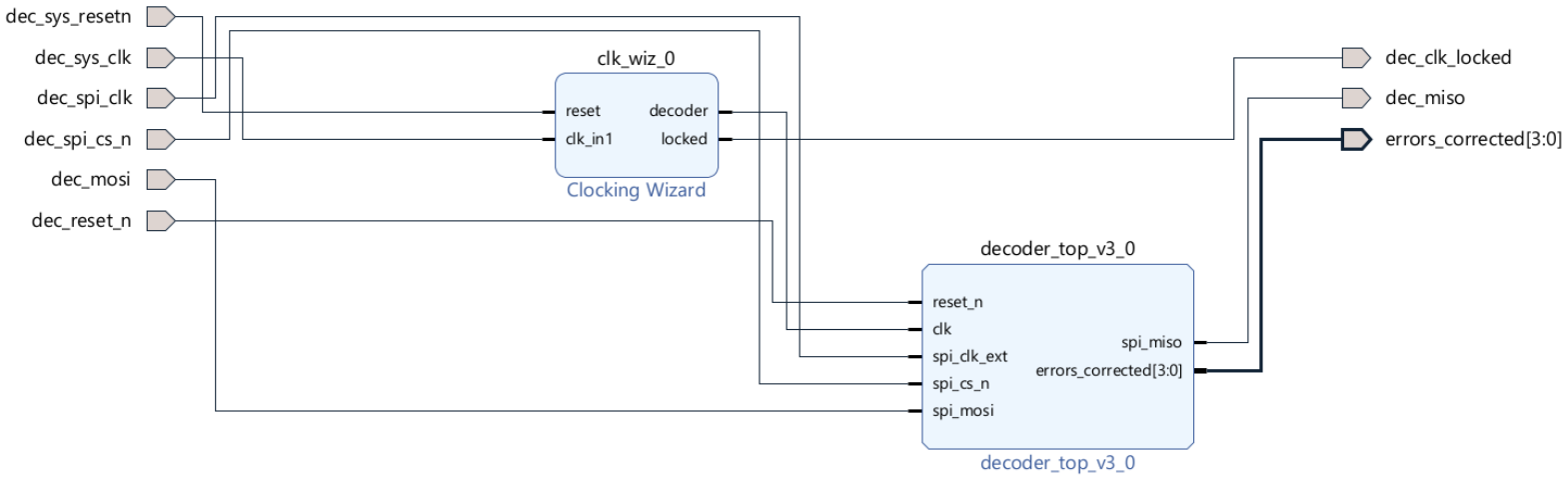

Figure 6. Decoder block design.

Decoder Interface:

The decoder interface is highly flexible and allows user modification. Custom feedback signals may be brought out to the surface for testing or control purposes. The following tables 4, 5 provide a brief overview of each top-level signal and parameters used to configure the decoder block and its SPI driver.

Table 4. Decoder Interface pin description.

Pin Name	In / Out	Description
reset_n	In	Asynchronous reset for Decoder interface. The MMCM is reset by asserting on '*dec_sys_resetn*' pin in figure 5.
clk	In	Decoder system clock.
spi_clk_ext	In	External SPI clock signal from encoder (master) driving the spi_driver (slave) side logic.
spi_cs_n	In	SPI chip-select / decoder SPI slave enable pin.
spi_mosi	In	SPI Master Out Slave In pin.

spi_miso	Out	SPI Master In Slave Out pin.
errors_corrected	Out	Feedback signal for testing purpose. Drives feedback LEDs on board to display values written into LEDs register of Decoder.

The decoder interface can be further classified internally into its core components:

- SPI slave driver
- Register Bank
- Control Unit
- Compute Unit

The '*SPI Slave drive*' will be discussed in the next section due to its complexity. Whereas the rest of the units are very simple and straightforward in their operation and design structure. They have been included within the Decoder to simply test the basic functionality of the module as a whole and to provide a base template for future upgrades. The '*Compute Unit*' is a simple [8, 4] combinational hamming decoder which exists to test the communication (SPI) link and decoding capabilities of the module. The data to be decoded is written into the register bank by the encoder through the SPI link into the designated input register and similarly the decoded output of the hamming unit is written to a designated output register, which may later be read by the encoder, once again through the SPI link. The register bank holds two important registers (apart from the other general-purpose registers), control and status registers shown in *figure 7*.

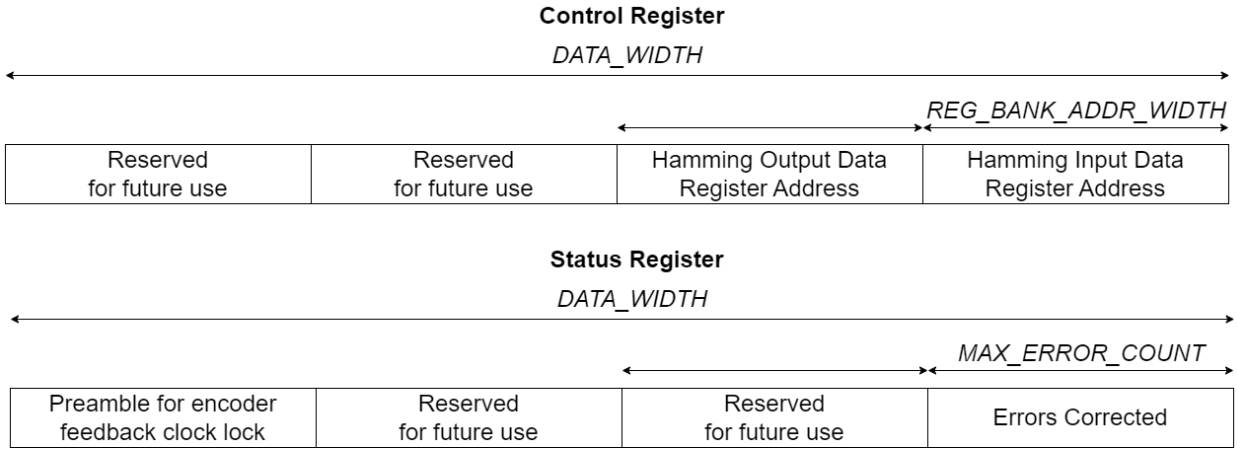

Figure 7. Register Map of Control and Status registers.

All registers in the register bank are completely configurable through the parameters presented in *table 5.* The '*Data Width*' parameter controls how wide the registers are and the maximum SPI data payload. Similarly, '*Max Error Count*' and '*Reg Bank Addr Width*' (a derivative of '*Num Regs Per Bank*') may be set according to application requirements.

Table 5. Decoder parameter description.

Parameter	Description
Ctrl Reg Num	Set which register is to function as the control register in the decoder's register bank.
Data Width	The bit width of SPI transaction.

Max Error Count	Controls the maximum errors decoder can count, i.e, controls the bit width of '*errors_corrected*'.
Num Regs Per Bank	Maximum number of registers available per register bank.
Spi Rd Addr Width	Set SPI read address width.
Spi Wr Addr Width	Set SPI write address width.
Stat Reg Num	Set which register is to function as the status register in the decoder's register bank.
Leds Reg Num	Set which register functions as LEDs register from register bank. Any value written here will light up the corresponding LEDs on-board through the 'errors_corrected' pins.
Comms Debug	Enable Communication line debugging. When enabled, will allow the SPI driver to write to the status register and disable hamming unit writes to status register. When disabled, no writes to the status register may be performed by the SPI driver and hamming unit may do so.
Shift In Comp	Sets the number of bits to ignore at the beginning of an SPI transaction before reading valid bits.

Slave SPI Driver:

The SPI driver uses two separate clock domains, first is the SPI clock domain run by the external '*SPI_Clk*' supplied by the encoder through the SPI link and the second being System clock domain. The decision to

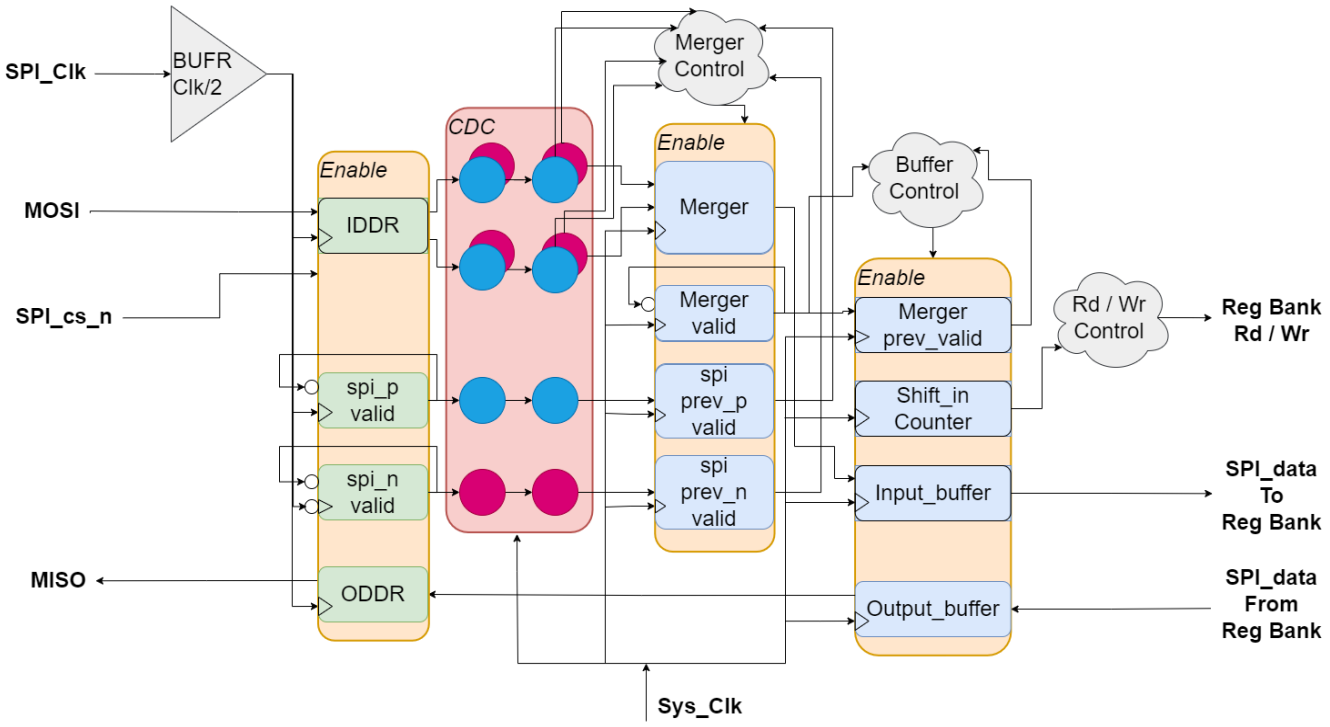

Figure 8. SPI slave driver block design. Green blocks are clocked at SPI_Clk/2, Blue blocks are clocked at rising edge of Sys_Clk and Red blocks at falling edge of Sys_Clk.

not sample the SPI signals directly using the system clock was to avoid constraining the system clock to a very high frequency (at-least twice the SPI clock frequency). This allows the two domains to be somewhat independent and their clocks can cater to separate needs. Through multiple iterations, the design was made simpler by utilizing the '*BUFR*' buffer in division mode to divide the SPI clock twice and feed the resultant clock to the '*IDDR*' and '*ODDR*' (Input/Output Double-Data-Rate Registers). In essence, a double-data-rate scheme is emulated on a Single-Data-Rate (SDR) SPI link, resulting in a slower SPI clock seen by the System Clock domain, hence requiring a slower system clock to sample the SPI clock domain signals.

The *IDDR* produces two data outputs *d1* and *d2*, for positive and negative edge of divided SPI clock respectively. Both data lines are sampled by the system clock during Clock-Domain-Crossing (CDC) using CDC registers as shown in *figure 8*. Every time a valid data is sampled by the *IDDR*, the corresponding '*spi_p/n_valid*' register is toggled for rising and falling edge respectively to indicate the presence of new data to the '*merger*' register. The *CDC* registers are shown as blue/red circles operating on rising/falling edge of system clock. The *merger* merges the separate data lines (*d1* and *d2*) into a single stream read by the '*input_buffer*' register, which is enabled by the '*Buffer Control*' logic cloud. At the end of an SPI transaction, the data to be written (in *input_buffer*) is written to the register bank at the corresponding address. Similarly, the '*output_buffer*' is loaded for next transaction with SPI write-back data to encoder.

Decoder Artifacts:

In *figure 8*, notice that data lines *d1 (spi_p_data)* and *d2 (spi_n_data)* are sampled on both positive and negative edge of system clock whereas its counterparts *spi_p/n_valid* are only sampled on a single edge. This is due to metastability introduced when *d1* and *d2* are not double sampled. There are four scenarios seen at the end of the *CDC* regs when entering system clock domain as shown in the following *table 6*.

Table 6. Metastability visualisation: "settle-old" and "settle-new" describes the value at the end of metastability.

Case	Spi_p_valid previous (prev)	Spi_p_valid current (curr)	Spi_p_data (Init_data = 0) (next_data = 1)	Data Sampled (if --> p_valid_prev ^ p_valid_curr == 1)	System data Corrupt
(baseline)	0	0	0	No	No
(1)	0	0 (settle – old)	0 (settle – old)	No	No
(2)	0	0 (settle – old)	1 (settle – new)	No	No
(3)	0	1 (settle – new)	0 (settle – old)	Yes	Yes
(4)	0	1 (settle – new)	1 (settle – new)	Yes	No

It is clear from case (3) that metastability on the data line causes error propagation into the *input_buffer*. This is clearly observed using an '*ILA*' core to capture the waveforms of the SPI link as shown in *figure 9*.

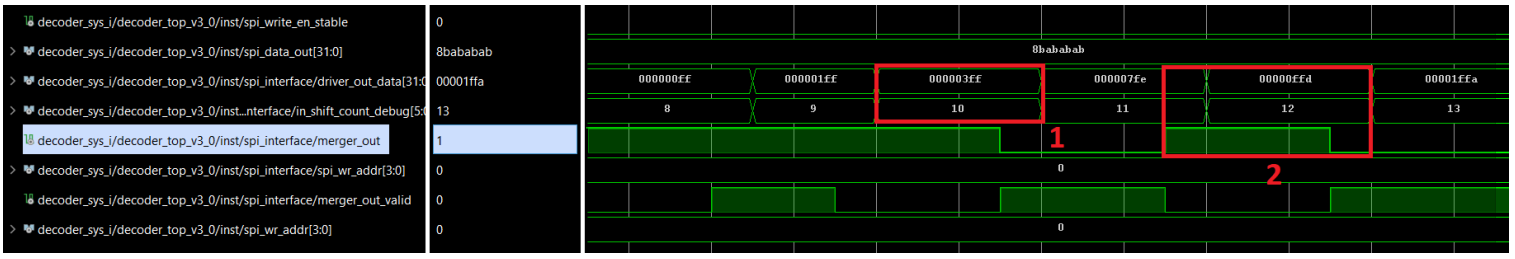

Figure 9. Error propagation due to metastability. Red box 1 is a bit that is ignored due to 'Shift In Comp' parameter being set to 1. Red box 2 shows the error propagation due to metastability, this bit must be '0' but it is sampled as a '1' due to metastability.

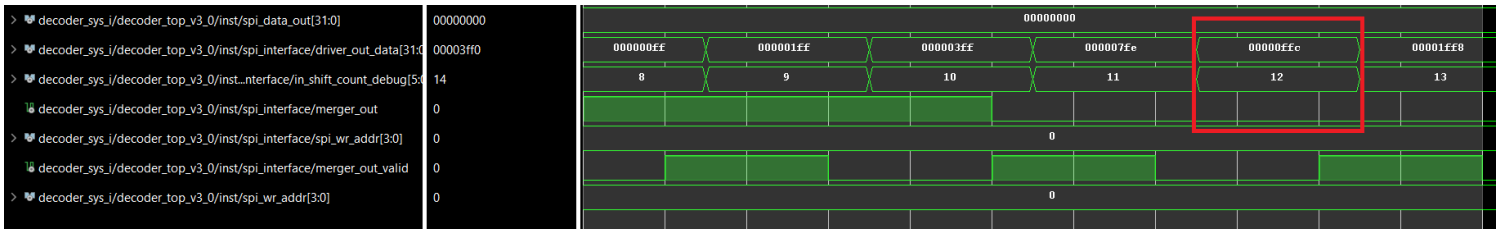

Figure 10. Error propagation due to metastability is resolved by double sampling. Red box shows the sampled bit is '0', as it should be.

To avoid error propagation due to metastability, we sample the data lines at both rising and falling edges of the system clock and the *merger* only samples when both rising and falling edge *CDC* registers exhibit the same value. This ensures there is no erroneous sampling as shown in *figure 10*.

Testbench

The testbench is created purely for simulation purposes and is part of the '*encoder_board*' design folder.

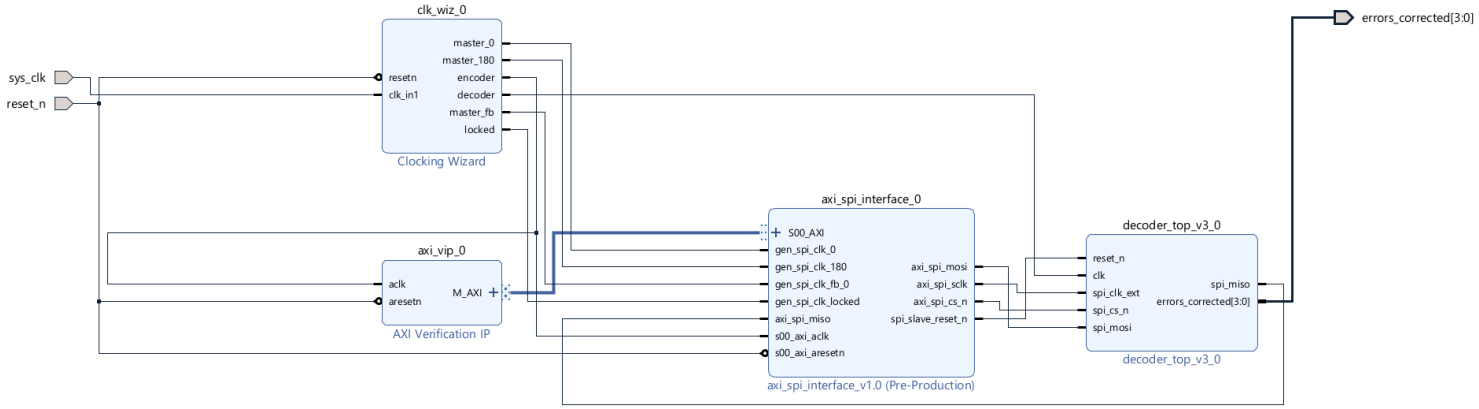

Figure 11. Testbench block design using Vivado's 'AXI-Verification IP'.

The simulation files are written using '*System Verilog*' so that the '*AXI Verification IP*' can be integrated into the testbench. It is recommended to go through the following links to understand the '*AXI-VIP*' and configure it according to the needs of the testing environment:

- https://support.xilinx.com/s/topic/0TO2E000000YNxCWAW/axi-basics-series?language=en_US&tabset-50c42=2&tabset-b221e=2
- https://docs.xilinx.com/r/en-US/pg267-axi-vip/Finding-the-AXI-VIP-Hierarchy-Path-in-IP-Integrator

In this testbench, the *AXI-VIP* is configured to act as a master using the AXI4-LITE protocol, since the '*axi_spi_interface*' will be acting as the slave device. The *axi_spi_interface* (encoder unit) will then communicate with the decoder unit through the designed SPI link. An MMCM is instantiated within the block design to provide full control over the clocking mechanisms of the individual devices under test, namely encoder and decoder units. For example, the system clocks of the encoder and decoder have been set to have a phase difference to emulate independent, asynchronous operation of the two boards to debug issues that may arise in the design during sampling of SPI signals on the decoder unit.

Device Constraining

Constraining is closely tied with implementation as it is an iterative process and will take time to find the right constraints for a specific implementation of the design. Once again, go through the '*Design Guidelines*' especially '*Clock and I/O resources*' section, since it is important to understand the available board resources before any design can be adequately constrained.

Clock Domain Crossing:

Clock domain crossing should be carefully analyzed and constrained at all paths. Although it might be tempting to simply ignore the CDC by setting the two clocks as asynchronous, it is advised against to prevent the creation of large register islands (due to the paths between clocks being ignored during Static-Timing-Analysis), which may not necessarily reflect the design intent. It is important to constrain each CDC path from the source clock domain to the synchronizer registers of the destination clock domain with the appropriate path/timing exceptions and. A more elaborate reasoning behind performing explicit CDC constraining can be found here:
https://support.xilinx.com/s/question/0D52E00006hpa28SAA/cdc-constrains-setclockgroups-precedes-setmaxdelay?language=en_US

Clock Forwarding:

For a source synchronous design, it is important to ensure the forwarded clock has very little skew with respect to data. One way of ensuring this is to drive both the output data and clock from '*IOB*' registers, which makes sure they leave the board at the same time. This is achieved by routing the forwarded clock through an *ODDR* (this ensures the clock stays on the global clock routing paths) and packaging the final data output register into an *IOB*. To inform Vivado tool of said routing through *ODDR*, the following constraint is used:

- create_generated_clock -name <*forwarded_clock_name*> -source <*ODDR_instance*> -divide_by 1 <*output port*>

Synchronous I/O:

In the previous section of '*Clock Forwarding*' we explored the importance of driving both clock and data signal of-board with minimum skew. It is equally important to make sure the clock path exhibits minimum skew with respect to data path within the board so that when the decoder's *IDDR* samples the *MOSI* line, it will sample the correct values. This is achieved by applying appropriate input constraints to the pin that is sampling the data and output constraints to the pin that is generating the data (clocked by *gen_spi_clk_180*) with respect to the forwarded clock (*gen_spi_clk_0*). At lower speeds, I/O constraining is simple and can be achieved by using '*set_max_delay*' on the input and output pins (not recommended). But the proper way to constrain I/O ports is through the '*set_input/output_delay*' constraints.

Encoder Input

According to the SPI protocol, the encoder samples data on the *MISO* line with respect to the forwarded clock *gen_spi_clk_0,* but from section '*SPI – A Pseudo Source-Synchronous Interface*' we understand simply constraining the input port with respect to *gen_spi_clk_0* will lead to issues with sampling of data due to clock-data skew. To circumvent this issue, we constrain the encoder input port with respect to an internally generated feedback clock *gen_spi_clk_fb_0*. We want our sampling clock to be at the center of the *data_valid_window* of our *MISO* data. Using the information from the following two articles as a base:

- https://cdrdv2-public.intel.com/653688/an433.pdf
- https://docs.xilinx.com/r/en-US/ug949-vivado-design-methodology/Using-XDC-Templates-Source-Synchronous-Interfaces

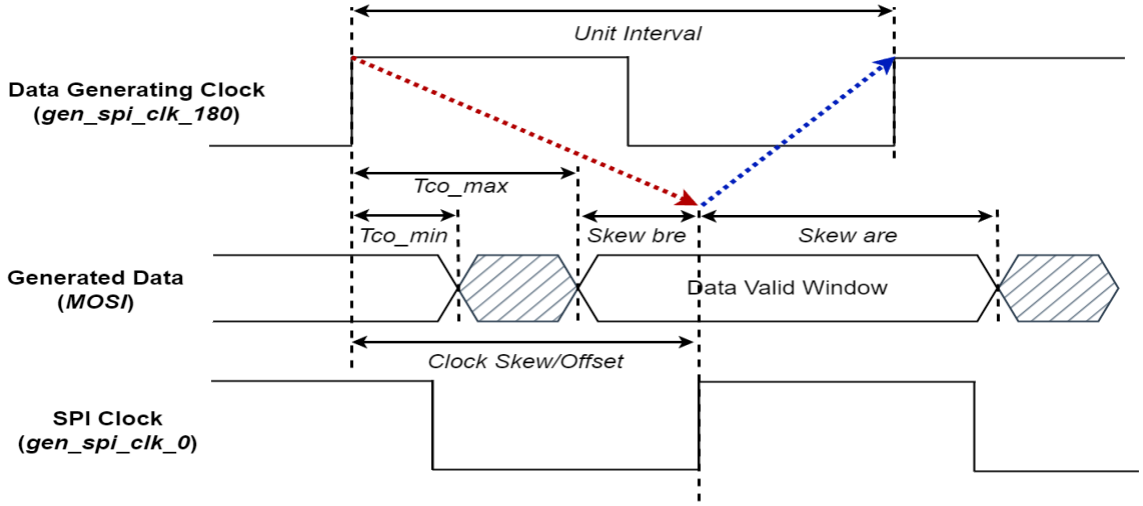

Figure 12. Representation of 'Tco_max', 'Tco_min' with respect to Data Generating Clock. 'Skew Before Rising Edge' (skew bre) and 'Skew After Rising Edge' (skew are) with respect SPI clock. The red and blue arrow show setup and hold relations between clocks.

We can simply constrain the encoder setup requirement as '*skew_before_rising_edge*' (*dv_bre* in Vivado XDC template) and encoder hold requirement as '*skew_after_rising_edge*' (*dv_are*) as shown in *figure 12*. In order to find *dv_bre* and *dv_are*, first the *data_valid_window* of the decoder must be obtained, this can be identified using the following equations:

$$data\ valid\ window_{decoder} = unit\ inteval - t_{co_{max}decoder} + t_{co_{min}decoder}$$

$$skew\ before\ rising\ edge_{encoder} = skew\ after\ rising\ edge_{encoder} = \frac{data\ valid\ window_{decoder}}{2}$$

The 't_{co_max}' and 't_{co_min}' values can be obtained from the port timing path reports that are generated after Vivado implementation of a design. Simply follow the following steps to obtain these values for each port:

1. Run implementation with '0' setup and hold time for output ports of decoder using the constraints: '*set_output_delay -clock [get_clocks <virtual_enc_spiClk>] 0.0 [get_ports dec_miso]* '.
2. Run the tcl command '*report_timing -to [get_ports dec_miso]*'.
3. Note down the value of '*data path delay*', which corresponds to 't_{co_max}'.
4. Similarly run command '*report_timing -hold -to [get_ports dec_miso]*' to get the value of 't_{co_min}'.

```
Timing Report

Slack (MET) :             0.412ns  (required time - arrival time)
  Source:                 decoder_sys_i/decoder_top_v3_0/inst/spi_interface/spi_out_oddr/C
                            (rising edge-triggered cell ODDR clocked by spi_clk_inv_buffered  {rise@20.000ns fall@60.000ns period=80.000ns})
  Destination:            dec_miso
  Path Group:             **default**
  Path Type:              Max at Slow Process Corner
  Requirement:            4.500ns  (MaxDelay Path 4.500ns)
  Data Path Delay:        4.088ns  (logic 4.087ns (99.976%)  route 0.001ns (0.024%))
  Logic Levels:           1  (OBUFT=1)
  Output Delay:           0.000ns
```

Figure 13. Inference of tco_max from 'report_timing' tcl command.

Now compute the *data_valid_window* of decoder and set input delay constraints with appropriate values as done in the encoder '*timing*' constraints file.

Encoder Output

From *figure 12* we can derive the output delay values in terms of '*Data Valid Window*', 't_{co_max}', 't_{co_min}' and '*Clock Skew*'. The following equations represent setup (max) and hold (min) values for output delay:

$$Output\ Delay\ Max = t_{latch_{decoder}} - t_{launch_{encoder}} - t_{setup_{decoder}} - t_{co_{max}encoder}$$

$$= Clock\ Skew - t_{setup_{decoder}} - t_{co_{max}encoder}$$

$$Output\ Delay\ Min = t_{latch_{decoder}} - t_{launch_{encoder}} + t_{hold_{decoder}} - t_{co_{min}encoder}$$

$$= Clock\ Skew - Unit\ Interval + t_{hold_{decoder}} - t_{co_{min}encoder}$$

Extra values '*enc_clk_skew_shift*' and '*enc_hold_comp*' are added in the constraint file to provide compensation when Vivado's *Static Timing Analysis* fails to realize the design due to setup or hold violations and helps the design pass *STA* rule checks to a certain extent.

Decoder Input

The decoder unit samples the input data synchronous with respect to the forwarded SPI clock from the encoder. Hence, follow the same steps as *Encoder Input* but instead of the feedback clock *gen_spi_clk_fb_0* use the forwarded SPI clock from the encoder as the reference.

Decoder Output

The decoder unit does not forward a clock along with its data (*MISO*). Hence, the natural approach would be to use a virtual clock and constrain the output similar to encoder's case. But, experimentation with this approach led to inconsistent data reads on the encoder unit at high frequencies. So, another approach was chosen where the outputs are constrained using '*set_max_delay*' and '*set_min_delay*' and the *gen_spi_clk_fb_0* clock on the encoder is carefully placed within the decoder output's *data valid window*.

Implementation and Vitis Environment

Both encoder and decoder units have been implemented and tested to work with a 100 Mhz system clock and therefore in theory can support SPI communication speeds up to 100 Mhz. For internal debugging, an '*Integrated Logic Analyzer*' IP from Vivado is instantiated after synthesis phase. Follow the following links to get started on how to use the Vivado *ILA* IP with any design project:

- https://docs.xilinx.com/r/en-US/ug936-vivado-tutorial-programming-debugging/Using-the-Netlist-Insertion-Method-to-Debug-a-Design
- https://vhdlwhiz.com/using-ila-and-vio/

The preset *ILA* core can be loaded during implementation of decoder board by making '*constr_1_ila*' active (found under constraints) before synthesis and implementation of the design. Once the implementation and bitstream generation is complete, it is time to move on to the *Vitis* platform to create an application to run on the ARM Processing System, which can be used to control the encoder from software through the *AXI-Lite* interface. Create and setup a Vitis platform project from the exported '*.xsa'* file from Vivado using the following link as a walkthrough (follow instructions only until the platform setup, do not build):

- https://xilinx.github.io/Embedded-Design-Tutorials/docs/2022.2/build/html/docs/Introduction/Zynq7000-EDT/2-using-zynq.html

Once the platform project has been setup, if Vitis version 2022.1 is being used, an extra step explained by this link ' https://support.xilinx.com/s/article/000034569?language=en_US ' must be performed (shown in *figure 14*) to be able to properly build the platform project. For further details on this and other commonly faced issues, refer to the '*know_issues_and_fixes*' document under the *Vitis* folder in the git.

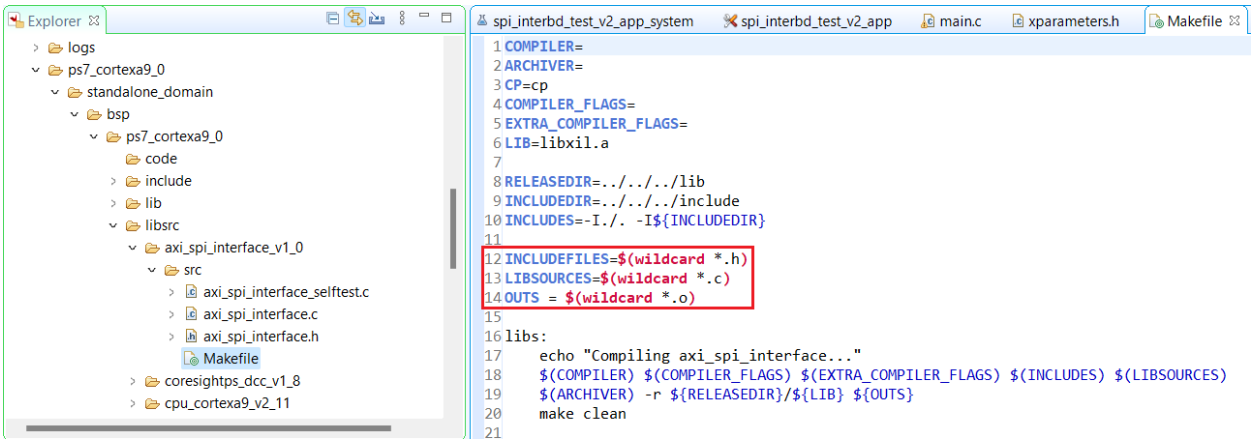

Figure 14. Makefile modification to build platform on Vitis 2022.1.

After the successful creation and building of a platform project, start by creating a new application project, following this link ' https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Create-an-Application-Project ' and copy the '*main.c*' file into the *source* (src) folder in Vitis application. Since we are running the application as a '*Bare-metal*' application, the target encoder board must be setup to operate in '*JTAG*' mode, else there might occur some problems due to the ARM *Memory Manager Unit* trying to perform OS related operations at invalid memory locations (due to the absence of Linux OS in bare-metal mode). Before configuring the board with the bitstream and running the application, ensure the right bitstream file is being uploaded to the right board by setting up '*Run-Configurations*' from the Vitis

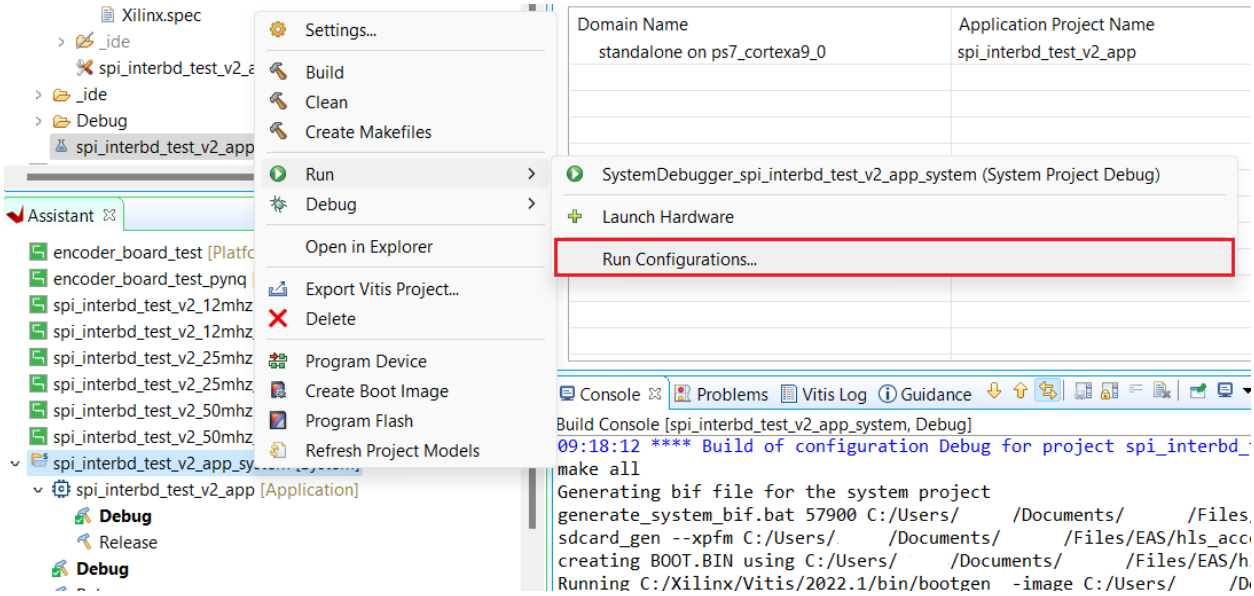

Figure 15. Accessing Run Configuration setup from Assistant window.

assistant window as shown in *figure 15 and 16*. Finally, connect to the encoder board using Vitis terminal and start debugging by running the application and monitoring the outputs on the terminal window.

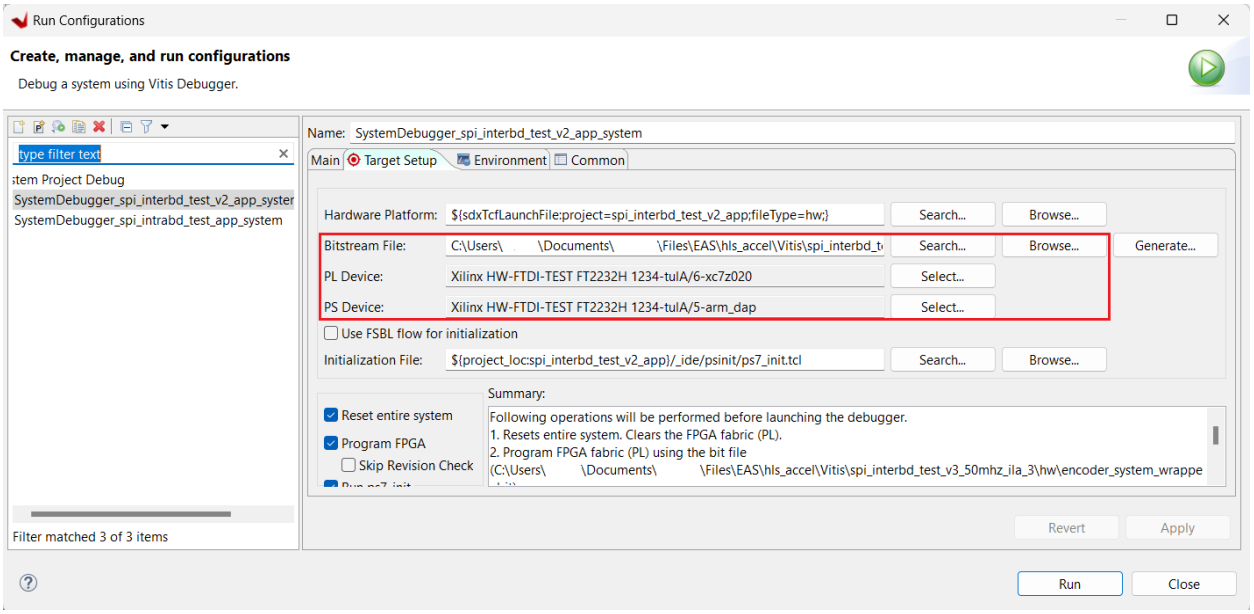

Figure 16. Setting-up Run Configurations.

Conclusion and Future Upgrades

The SPI clock transmission seems to be the principle driving force behind erratic behavior during SPI transaction. Testing between the two Zedboards (in MMCM mode) for speeds lower than 50 Mhz, the MMCM attains solid lock, but for 50Mhz and 100Mhz, there is no good lock attained (this can be physically visualized through the brightness level of the '*SPI_clk_locked*' LED in MMCM mode). This impacts the behavior of the SPI slave driver where some bits are not read or are missed due to bad clock signal. But to test the functionality of the HDL at 50Mhz and above, intra-board testing was done using a Pynq-Z2 board running the same Zynq 7020 chipset. At 50 Mhz, the clock is driven outside the board by the encoder through PMOD-B and received by decoder through PMOD-A, but it is suspected that this SPI clock signal might be working in this setup due to cross-coupling of the signals within the board traces and thus enhancing the quality of the received clock. At 100Mhz, unfortunately this configuration too fails as there is no clock detected by the decoder. To test the RTL design at 100Mhz, we use the PynqZ2 experimental block design (part of the *encoder_decoder_test_v3* board files), where the clock is routed to the decoder within the board and not through the PMODS, and as expected, works fine. More on the experimentation setup and results can be obtained from the *Appendix* section.

Hence, it is safe to conclude that a source synchronous interface's quality is only as good as the physical connection between the two boards. In this case, the bottle neck is the transfer quality of the SPI clock signal, as this is evident at frequencies higher than 25Mhz, especially at 50Mhz and above where the clock signal completely fails to show up on the decoder board.

For future upgrades, the MMCM can be dynamically configured allowing the encoder to attain lock on the feedback clock through dynamic phase shift, moving the process of setting the feedback clock phase from implementation level to application level, speeding up the test process significantly. In addition, the dynamic behavior may be extended to frequency setting to unlock dynamic transmission speed settings.

Appendix

The values calculated in section *Device Constraining* should only be considered as ball park values for high frequencies. The current recommended method of debugging and figuring out the right constrain values or clock phase shift value for MMCM is with decoder *ILA* implementation for monitoring and inference of decoder signals. The figures provide some insight into the issue of clock forwarding at 25 Mhz and 50 Mhz.

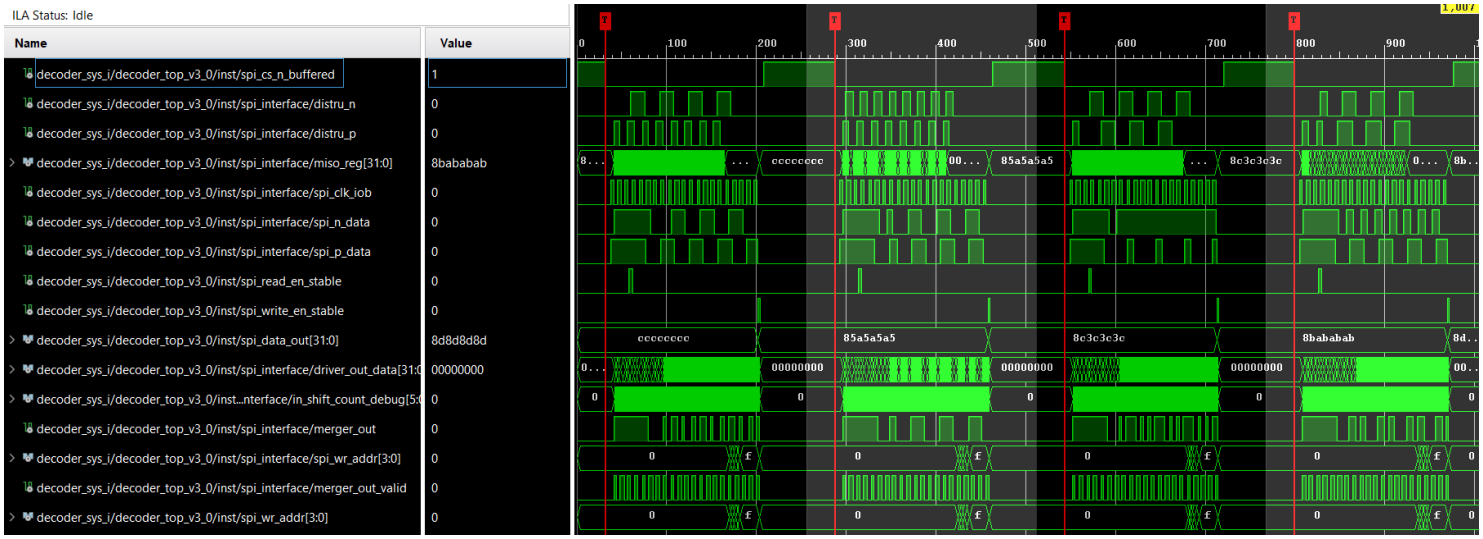

Figure 17. Clean SPI clock received at 25 Mhz.

The experiment was conducted at 50 Mhz using both inter-board (between two physically separate boards) and intra-board (encoder and decoder unit implemented on the same board) setup. In *figure 18*, there is no SPI clock is detected by the decoder unit and hence there is no reaction from the decoder.

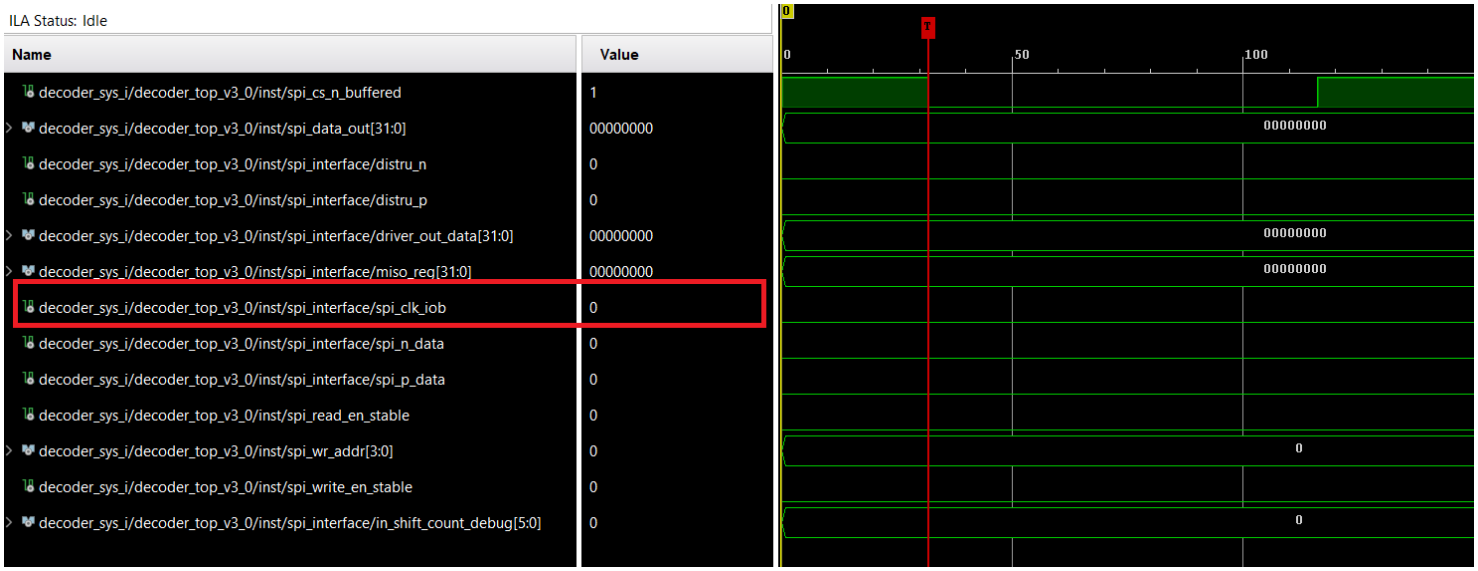

Figure 18. No SPI clock detected at 50 Mhz during inter-board test. Transmission of SPI signals using 20cm jumper wires.

To further investigate the effect of transmission lines on SPI communication and determine the weakest link, a second experiment was performed by replacing the 20cm jumper wires with breakout header pins, connecting the PMODs of encoder and decoder board together (inter-board setup). This produced a significant improvement but still does not provide the decoder unit with a clean and stable SPI clock as shown in *figure 19*.

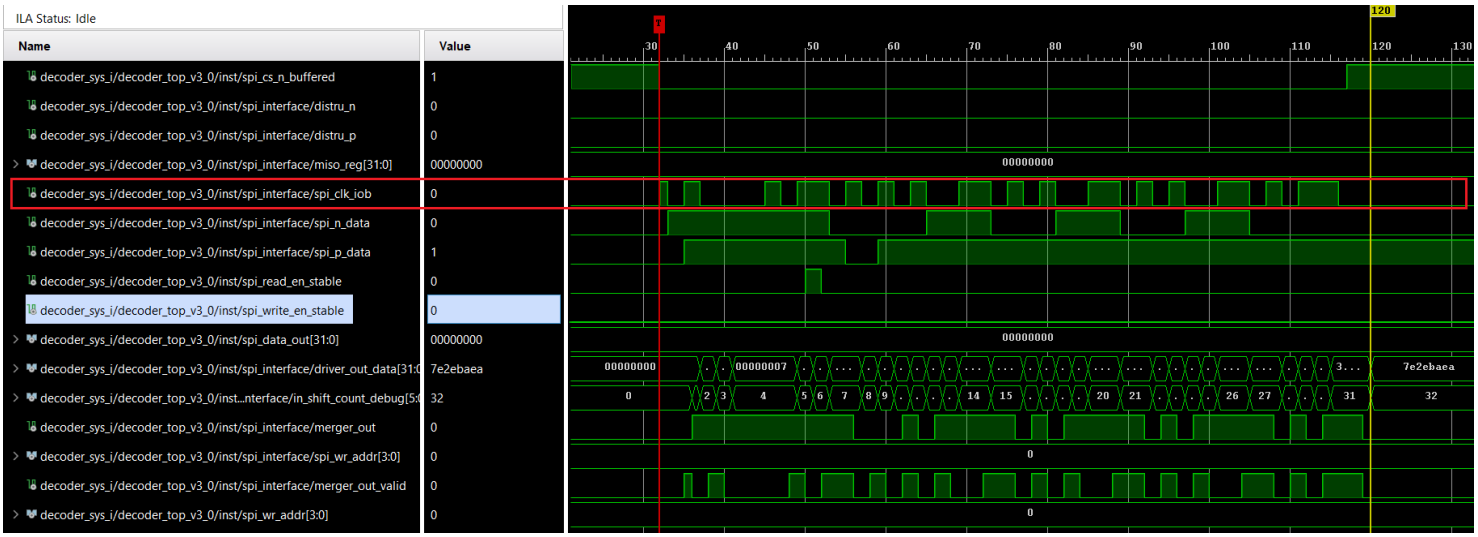

Figure 19. Inter-board test conducted with breakout header pins as transmission lines. SPI clock is transmitted but is still not clean. Hence, decoder misses sampling some data bits when the clock edge is not present.

Due to the unstable SPI clock (missing clock edges) received by the decoder unit, it fails to sample some data bits, resulting in a corrupt SPI transaction.