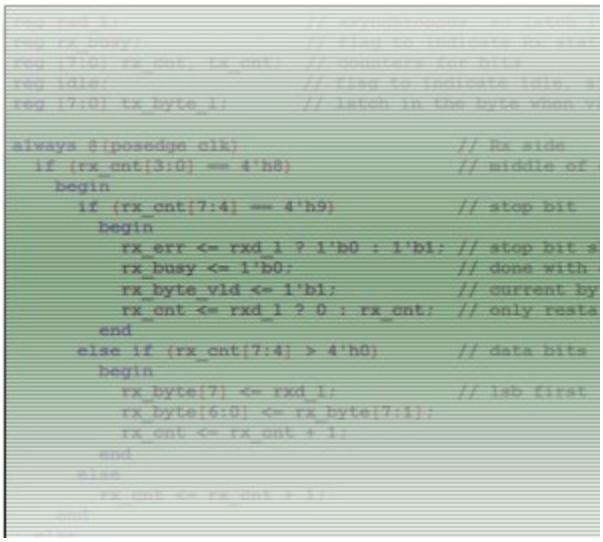
Sidekiq FPGA Development Manual

Version 3.15.1



EPIQ SOLUTIONS

Disclaimer

Epiq Solutions is disclosing this document (“Documentation”) as a general guideline for development. Epiq Solutions expressly disclaims any liability arising out of your use of the Documentation. Epiq Solutions reserves the right, at its sole discretion, to change the Documentation without notice at any time. Epiq Solutions assumes no obligation to correct any errors contained in the Documentation, or to advise you of any corrections or updates. Epiq Solutions expressly disclaims any liability in connection with technical support or assistance that may be provided to you in connection with the Information.

THE DOCUMENTATION IS DISCLOSED TO YOU “AS IS” WITH NO WARRANTY OF ANY KIND. EPIQ SOLUTIONS MAKES NO OTHER WARRANTIES, WHETHER EXPRESSED, IMPLIED, OR STATUTORY, REGARDING THE DOCUMENTATION, INCLUDING ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT WILL EPIQ SOLUTIONS BE LIABLE FOR ANY CONSEQUENTIAL, INDIRECT, EXEMPLARY, SPECIAL, OR INCIDENTAL DAMAGES, INCLUDING ANY LOSS OF DATA OR LOST PROFITS, ARISING FROM YOUR USE OF THE DOCUMENTATION.

Revision History

Date	Revision	Description
06/06/2014	1.0	Initial release
10/09/2014	1.3	Updated user_app and timestamp_block info
01/08/2015	1.5	PPS features and user metadata added
9/15/2015	2.0	Packed mode added, major user_app overhaul, processing path added to TX path, fixed glitches related to rapid retuning, numerous bug fixes. user_app top level changed – will break compatibility with previous 1.* versions.
5/9/2016	3.0	Bug fixes, register changes break compatibility – must use 3.0 SDK or newer with this PDK
8/22/2016	3.1	Dropkiq and M.2 support added
	3.2	Internal unreleased build
1/6/2017	3.3	USB initial release, configurable Tx FIFOS (M.2 only)
	3.4	Internal release
5/16/2017	3.5	M.2 improvements: Optimization improvements to reduce BRAM usage Rx/Tx build configurability XADC access M.2 USB transport layer support
4/13/2018	3.9	Minor updates and improvements to documentation, including correction of USB and M.2 build script names
01/04/2019	3.11.0	M.2 USB, fix lna enable and tx enable pins. All, add MANUAL_TRIGGER capabilities. M.2 USB, corrected USB clock period constraint. All USB, Corrected usb bus high impedance issue so that Tx works properly.
03/01/2019	3.11.2	Fix M.2 Dual Tx phase coherency. Fix M.2 and mPCIe reg_tx_last_ts bug.
04/25/2019	3.12.0	Fix PCIe Tx when packet size is larger than FIFO size. Add frequency hopping control logic. Add iq swap mode.
09/24/2019	3.12.1	Add support for sidekiq M.2 Stretch
01/13/2020	3.13.0	MPCIe: Fix dropkiq spi done bug, up the speed on the dropkiq i2c. M.2, M.2 USB, and M.2 Stretch: Add support for User App GPIO control and user app RFIC SPI interface, Add user_pdk_config.v parameter file, Add WBSTAR 100 ms delay when reconfiguring the FPGA via ICAPE. MPCIe USB and M.2 USB, Double Rx FIFO size. M.2 Stretch: Switch PPS default source to be from the GPS PPS. All: Fix packed_mode bug in which drops (fifo_full) were in the middle of packets M.2, M.2 USB: Upgrade to Vivado 2018.3.

06/05/2020	3.13.1	M.2 Stretch: Add temp valid bits to the gpsdo temperature read register. All: Change packet header format to v1.
08/02/2020	3.14.0	All: Fix dual Tx underrun reporting bug.
01/14/2021	3.14.1	M.2 and M.2 Stretch: Fix FIFO full write bug which resulted in the Rx header and sample data to be scrambled in certain circumstances. Fix for starting/stopping streaming on a 1PPS edge. M.2 Stretch: Add GPSDO functionality.
04/12/2021	3.15.1	M.2, M.2 Stretch: Fix Tx Timestamp dropped packets after late bug. M.2 Stretch: Add GPS_CONTROL_MASK read only register. M.2 Stretch: Add gps_pps mux control. M.2, M.2 USB, and M.2 Stretch: Fix missing timestamp reset related to clock crossing synchronization on the register interface. M.2, M.2 USB, and M.2 Stretch: Add register reset capability. M.2, M.2 USB, and M.2 Stretch: Add BASELINE_VCS_STATUS register. M.2, M.2 Stretch: Add frc_sel_for_tx to be used in Tx timestamp mode.

Table of Contents

Table of Figures

Table of Tables

1 About this Document

This document provides the necessary details for developing FPGA applications on the Sidekiq™ SDR developed by Epiq Solutions [1]. It is provided with the purchase of a Sidekiq Platform Development Kit.

2 Legal Considerations

The Sidekiq is distributed all over the world. Each country has its own laws governing the reception and/or transmission of radio frequencies. The user of the Sidekiq and associated software is solely responsible for insuring that it is used in a manner consistent with the laws of the jurisdiction in which it is used. Many countries, including the United States, prohibit the reception and/or transmission of certain frequency bands, or receiving certain transmissions without proper authorization. Again, the user is solely responsible for the user's own actions.

3 Proper Care and Handling

The Sidekiq unit is fully tested by Epiq Solutions before shipment, and is guaranteed functional at the time it is received by the customer, and ONLY AT THAT TIME. Improper use of the Sidekiq unit can cause it to become non-functional. In particular, a list of actions that may cause damage to the hardware include the following:

- Opening up the unit while it is powered up
- Handling the unit without proper static precautions (ESD protection) when the housing is removed or opened up
- Connecting a transmitter to the RX port without proper attenuation – A max input of -10 dBm is recommended.
- Executing custom software and/or an FPGA bitstream that was not developed according to guidelines

The above list is not comprehensive, and experience with the appropriate measures for handling electronic devices is required.

4 Introduction

The Sidekiq Platform Development Kit (PDK) provides the ability for users to create their own custom applications. This can be accomplished by customizing software or the RTL code that configures the FPGA. This manual gives an overview of the FPGA reference design, with the intention of empowering the user to build upon the design to create custom applications.

Detailed information about the software environment, including how to create custom software applications, can be found in a separate document, the Sidekiq Software Development Manual [2], which can be downloaded from the Epiq Solutions support website [3]:

http://www.epiqsolutions.com/support

In addition, the details of the hardware itself and system design of the unit is outside the scope of this document. For more details about the hardware, please download and review the Sidekiq Hardware User's Manual [4]. It is strongly recommended that the user read this document thoroughly before attempting to dive into FPGA development.

This manual is meant to concisely describe the FPGA reference design, but it is important for even an experienced developer to spend time evaluating the actual design (i.e. RTL source code), perhaps even while digesting the information presented here. The sections of the manual were intentionally created to align with the basic hierarchy of the design, and the source code itself is commented and will act as a supplement to the information provided here.

5 References

[1] **Epiq Solutions Website**
http://www.epiqsolutions.com

[2] **Sidekiq Software Development Manual**
http://www.epiqsolutions.com/support

[3] **Epiq Solutions Support Website**
http://www.epiqsolutions.com/support

[4] **Sidekiq Hardware User's Manual**
http://www.epiqsolutions.com/support

[5] **Open Core Protocol International Partnership Website**
http://www.ocpip.org

[6] **OpenCores® I2C Controller Website**
http://www.opencores.org/project,i2c

6 Terms and Definitions

Term	Definition
1PPS	1 Pulse Per Second
ADC	Analog to Digital (A/D) converter
DAC	Digital to Analog (D/A) converter
DSP	Digital Signal Processing
EOF	End-of-frame
EOP	End-of-packet
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FRC	Free-running counter
GPS	Global Positioning System
I/Q	In-Phase / Quadrature Phase
I2C	Inter-Integrated Circuit
IP	Intellectual Property
LSB	Least Significant Bits / Bytes
M.2	mSATA / mPCIe replacement card connector standard
MHz	Megahertz
MSB	Most Significant Bits / Bytes
PC	Personal Computer
PDK	Platform Development Kit
PR	Partial Reconfiguration
RF	Radio Frequency
RM	Reconfigureable Module
Rx	Receive
SDK	Software Development Kit
SDR	Software Defined Radio
SM	Static Module
SOF	Start-of-frame
SOP	Start-of-packet
SPI	Serial Peripheral Interface
Tx	Transmit
VHDL	VHSIC Hardware Description Language

Table 1: Terms and Definitions

7 FPGA Reference Design

7.1 Overview

The Sidekiq PDK provides a complete FPGA reference design that enables a user to quickly and efficiently create custom applications targeting Sidekiq MiniPCIe with a Xilinx Spartan®-6 LX45T, Sidekiq M.2 with a Xilinx Artix®-7 XC7A50T, or Sidkiq M.2 Stretch with a Xilinx Artix®-7 XC7A50T FPGA. On Sidekiq MiniPCIe and Sidekiq M.2, both PCIe and USB data transport modes are supported with separate FPGA reference designs.

The unmodified reference design provides a full FPGA implementation to flow raw Rx I/Q samples from one of up to two ADC channels (Sidekiq variant dependent) to the host system processor and to transmit from the host to the RF chip via the FPGA two channels (Sidekiq variant dependent) of Tx data.

The PDK structure is created with the ease of the end-user in mind.

The Rx path transfers baseband I/Q samples which are received from an Analog Devices AD9361 RF Transceiver. These samples pass through a DC offset correction block (which can be toggled on/off by software), then into a processing block which allows the user to process, timestamp, and transfer the samples via a FIFO interface to a PCIe subsystem. In the reference design, the user application processing block can function as a simple pass-through which timestamps and drives the PCIe FIFO with no changes to the samples. In this format, the 12-bit I and 12-bit Q components are sign-extended to 16 bits, for a 32 bit wide data bus. The user application can also pack the data to fully utilize the 32 bit bus, transferring four IQ pairs with every three 32-bit transfer. Note that this requires software to decode the data into the proper 12-bit I and 12-bit Q components.

On the Tx side, I/Q samples are transferred from the host system to FPGA over PCIe directly into the AD9361 DAC interface. Though the provided reference design does not include a Tx processing block, the PDK user has the ability to process Tx if so desired. Similar to the receive side, transmit data can be pushed down in 16-bit sign extended mode or packed mode. Non-IQ data of an arbitrary format can also be sent down and processed in the FPGA for the user to process and transmit as desired.

In Figure 1, orange design blocks are in NGC format (XCI format for M.2 and DCP format for M.2 Stretch) and cannot be changed (For M.2 Stretch, the only module that cannot be changed is the PCIe Block. The Rx FIFO and Tx FIFO modules are now part of the RTL, but still should not be modified). Red blocks are available in RTL, but should not be modified. Yellow blocks should not require modification, but certain applications may necessitate changes. Green blocks are the intended targets for user modification.

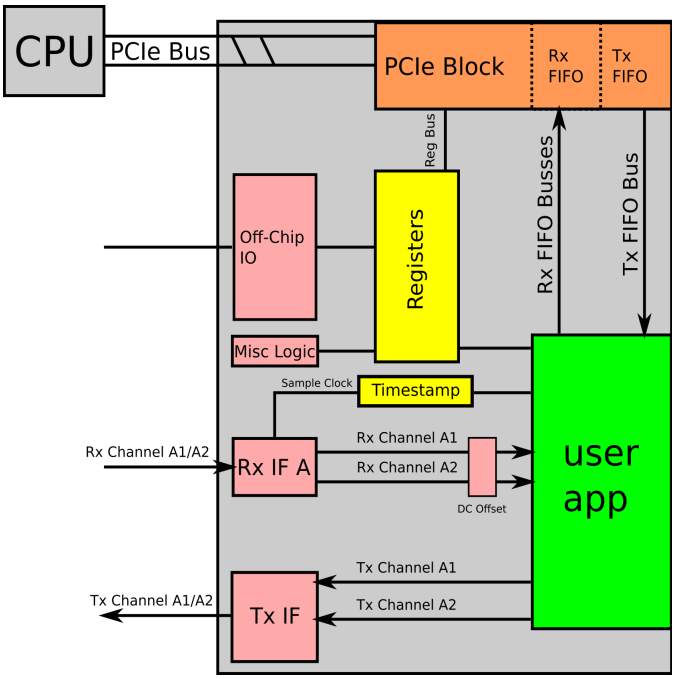


Figure 1: Sidekiq Simplified Block Diagram

7.2 Top Level

The Top Level block, sidekiq_top.v, sidekiq_usb_top.v, and sidekiq_m2_top.v are wrappers containing the top-level RTL and all submodules. This section will serve to describe each block's functions and use. Sections that have more significant impact on a PDK user will be discussed in greater detail.

The top level also contains the system GPIO pins, which can be mapped by a PDK user as desired.

7.3 user_app

The user app and register interface are Verilog files in which the majority of signal processing is expected to be done. The user_app interface is designed to allow reuse through multiple Epiq SDR platforms. user_reg_if is a submodule of user_app, which allows the user to maintain and customize their own register space. In most cases, only the user_app.v and user_reg_if.v will need to be modified to create custom FPGA images with advanced signal processing capabilities. See section 8.1.3 for more information on building custom user_apps for mPCIe and M.2 or section 10.2 for M.2 Stretch. The user_app structure is designed to allow for portable signal processing blocks between multiple Epiq SDR platforms, including all Sidekiq variants. This allows end users to share user_app modules between platforms allow upgrades to future platforms with minimal rework needed.

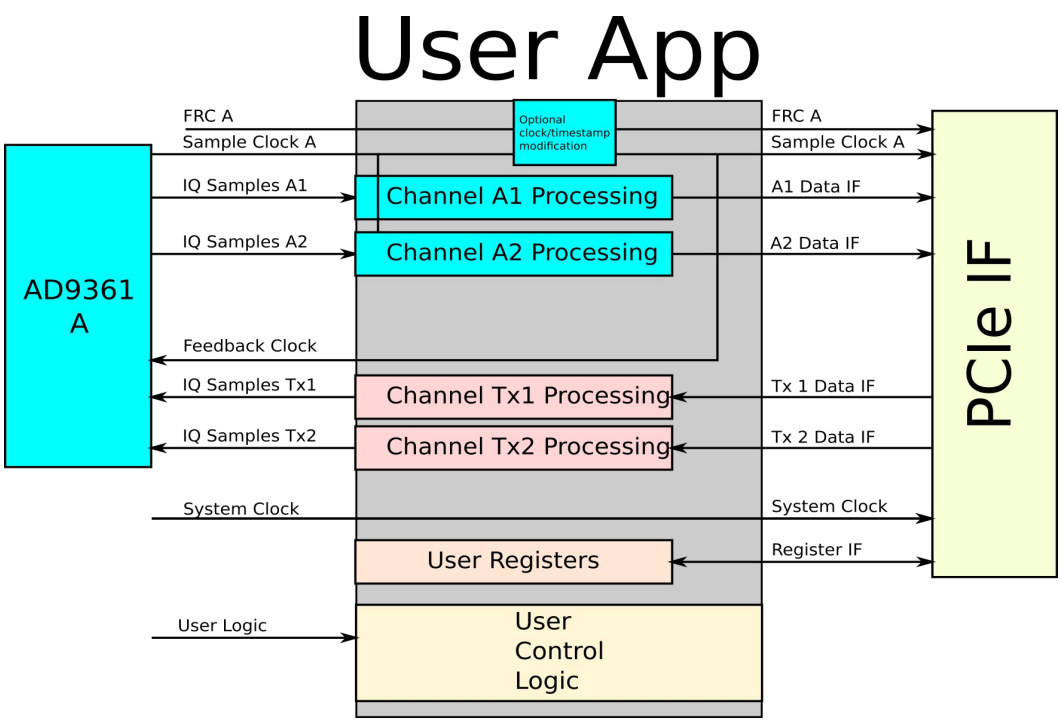


Figure 2: User App Block Diagram

7.3.1 user_app Signals

Several signals are provided to facilitate custom designs, some of which are not used in the stock user_app.

host_clock is the host system clock, which runs at 62.5MHz on PCIe based hosts and 40MHz on USB based hosts.

clk_tx_fb is a feedback clock used to drive the transmit path. It runs at twice the sample rate clock. It is discussed more in the user_app transmit section.

aux_clk is an always-on, low accuracy clock which runs at 40 MHz on Sidekiq MiniPCIe and 24 MHz on Sidekiq M.2. It provides a clock source even if the RF chip is not running or no external reference is present.

ref_clk is an accurate 40MHz clock shared with the RF chip. If no external reference is present or if the RF chip is turned off, this clock will not be present.

external_enable can be used as an external gating signal for transmit/receive. By default, it is wired at the top level to allow software to start/stop transmitting on a PPS edge. If this functionality is not needed but custom external gating is desired, this signal can be used.

timestamp_rst used to reset timestamps and free-running counters. Timestamps can also be reset by software.

reg_rx_# is a per-channel control register that allows the user to monitor the state of the channel as controlled by software. Values of user interest are:

Bit # (indexed from 0)	Value : 0	Value : 1
1	IQ Data flowing	Test mode: counter data flowing
2	Rx Reset (No data should flow)	Rx Enabled
8	Tx continuous mode	Tx timestamp mode
9	Tx FIFOs enabled	Tx FIFOs reset
10	Passthrough mode	Packed Mode
11	DC Offset disabled	DC Offset enabled

Table 2: Rx Control Register

Bit 1 indicates if IQ data is flowing or if software has set test mode, which results in incrementing counter data being placed on the I and Q data busses.

Bit 2 is especially important – when low, the user should not push data to PCIe and should reset any data counters, as the PCIe bus is inactive.

Bit 5 in reg_rx_a1 indicates if the RFIC is in single or dual channel mode. In single channel mode, clk_tx_fb is twice the sample rate of sample_clk. In dual channel mode, clk_tx_fb is four times the sample rate of sample_clk. The value in reg_rx_a2 will always be 0.

Bit 8 indicates if the transmit path is sending data as soon as it arrives, or if it is waiting to transmit data at a specific timestamp driven by software.

Bit 9 acts similar to bit 2, but on the transmit path. When asserted, the transmit FIFOs in the PCIe block are in reset.

Bit 10 indicates if the channel is operating in passthrough mode, which sign-extends the 12 bit I and Q data to 16 bits each. If high, packed mode is enabled, which keeps I and Q 12 bits, and packs in the next portion of I Q data in. This requires software decoding when received by the host but allows for a roughly 20% increase in throughput.

Bit 11 indicates if software has enabled or disabled the DC Offset block, which acts on the IQ data before entering the user_app.

7.3.2 Rx Path Inputs to user_app

I/Q samples, valid signals, and clocks are routed in to the user_app and can be modified within to perform user-desired DSP. The input ports related to I/Q samples used are:

[11:0] i_samples_in_#, q_samples_in_#

sample_clk_a

sample_valid_a_#

full_#

Sidekiq supports up to two I/Q channels. While sample_valid is high, each rising edge of the sample clock delivers a new sample consisting of twelve bits each of I and Q data. Each channel must be enabled by software before the sample_valid signal will go high.

The full flag is from the PCIe interface and is used to indicated that, while sample data flowing in may be valid, there is no longer room in the PCIe buffer to contain them. This may be due to incorrect PCIe signaling or may indicate the total data throughput on all active channels exceeds the PCIe data transfer rate.

Two counters are passed in, which are passed through to the PCIe interface to software.

[63:0] frc_a_in, frc_sys_in

frc_a begins running when either channel associated with it is enabled (channels 0 or 1). They continually increment as long as an associated clock is running. They can also be reset using a software-programmable PPS based reset or by user logic. On a retune or sample rate change, this clock will temporarily be lost.

frc_sys is a system counter that runs whenever the 40MHz reference clock is present. It is 64 bits wide, and is normally driven by the 40MHz reference clock to ensure accurate timekeeping. It can be reset using a software-programmable PPS based reset, or by user logic. This counter is continuous.

The following diagram illustrates the relationship between *sample_clk*, *samples_valid*, IQ data, and the free-running counter. Note that samples are valid only when samples_valid is high, but the counter increments as long as there is a valid clock.

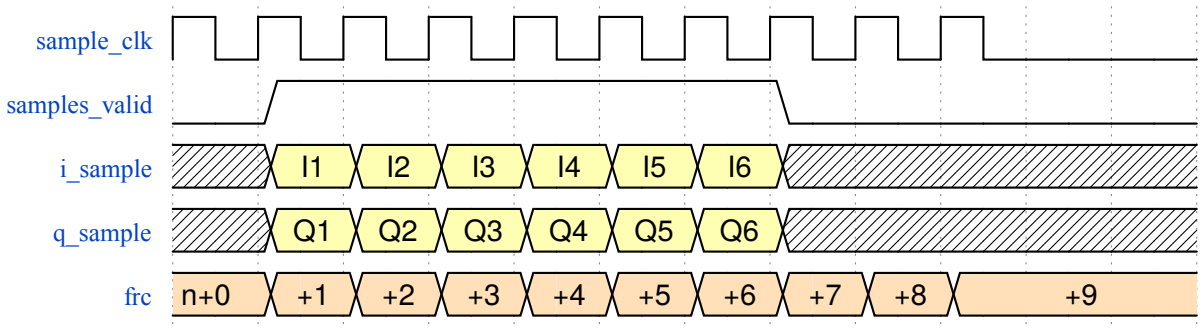


Figure 3: Sample Timing Diagram

7.3.3 Outputs from user_app

The user app drives data into a FIFO within the pcie_block NGC. Though the data can be thought of as a logical stream, each PCIe transfer consists of 1024 words of 32 bit length each. There are 1018 words of data and 6 words of metadata for 1024 total words. The first four words contain 64-bit timestamps based on *frc_a* and *frc_sys*. The fifth word contains channel and system control information. The sixth word is reserved for user definition.

Note that the PCIe bus works on 1018 data increments – until this size is reached in the FIFO, no data will be transmitted. If smaller data blocks are needed, the user must pad the remaining data

words until the 1018 data length is reached.

The following signals are used to drive the PCIe FIFO:

[31:0] fifo_din_#

fifo_wren_#

sample_clk_a_out,

[63:0] frc_a_out, frc_sys_out

[31:0] user_metadata_#

fifo_din_# is a 32 bit wide data bus containing the data to be transferred to software. In the stock app, this contains unprocessed IQ data. Custom apps may process and place data in whatever format desired on this bus. The *fifo_wren_#* signal must be driven high to push *fifo_din_#* data into the PCIe block buffers. The write enable does not need to be continuous – it can be asserted and deasserted as necessary to push the proper processed data through. If tied high, data will continuously be streamed into the PCIe block

Channel A1 and A2 FIFO clocks are driven by *sample_clk_#_out*. This is normally tied to *sample_clk_#*, but user applications may have other requirements.

If user blocks introduce a processing delay, the free-running counters can be modified to account for the delay. Alternatively, *frc_x_out* can be tied directly to *frc_x_in* and software can account for delays.

The *metadata_#* registers are optional. They contain up to 32 bits of user-supplied data which is embedded within each data block transferred over PCIe. This allows users to transfer extra data with each set of 1018 samples. The metadata is latched in on each the first of the 1018 data words.

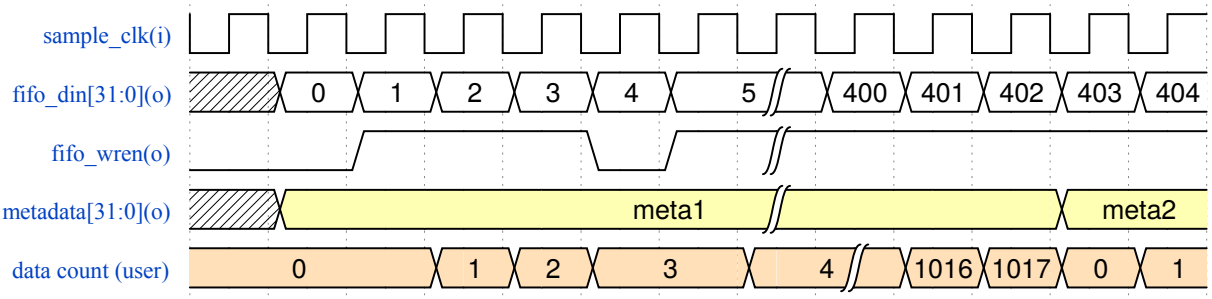


Figure 4: Sample User App to PCIe Diagram

7.3.4 user_app Tx Interface

The Tx user interface allows software to pass down either IQ or other encoded data for processing to the FPGA to then be transmitted via the RF DAC interface. The DAC interface will be described first.

The interface to the RF chip is a 6 bit wide DDR bus, which means the clock driving it will run at twice the sample clock rate in a mode with 1 receive and 1 transmit channel. On the receive side, the data_clk is also twice the sample clock rate, but this is transparent to user_app as it receives a sample_clock so each rising edge brings in a new IQ pair. On the transmit side, the faster clock is used throughout the design.

The DAC interface can not be modified by the user. When the user_app signals *tx_dac_en_out* high, the dac_if block will begin driving *tx_rd_en_in* as shown below, with a 50% duty cycle (due to the 2x clk_tx_fb clock). I and Q data must be provided after each rising edge in order to provide uninterrupted transmission of data.

In passthrough mode, where each 32 bit data word *tx_din_#* coming from PCIe contains a single I and Q pair and no additional data to be transmitted, *tx_dac_en_in/out* and *tx_rd_en_in/out* can be wired together to pass data straight from the PCIe FIFO to the dac_if block.

The PCIe interface can be modified if a user desires additional processing on transmit data. This interface pulls data from the PCIe bus. To signal data is available, *tx_dac_en_in* goes high. On each *tx_rd_en_out* strobe, data is pulled from the PCIe FIFO.

Note that if timestamp mode is used, the PCIe interface will not drive *tx_dac_en_in* high until the proper timestamp is reached. In continuous mode, data is provided as soon as it is available in the FIFO. Continuous/timestamp mode is set by software.

An example timing diagram is below for an app which would, in each 32 bit word over PCIe, contain two IQ pairs of 8 bits each. Since each word contains two pairs, only one word is pulled from the PCIe FIFO for each 2 IQ pairs sent out to the DAC_if.

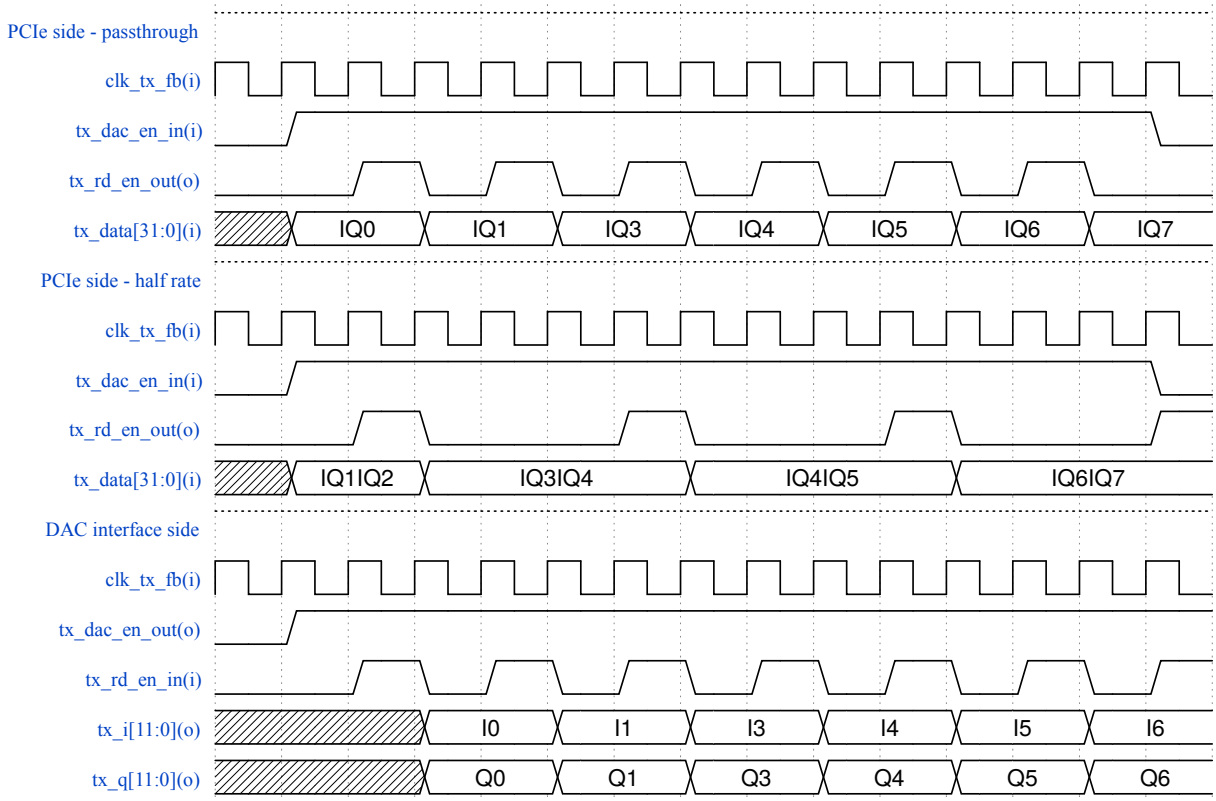


Figure 5: Sample Tx Timing

Several other signals are available to assist in debug.

tx_ts_# indicates the timestamp (on the frc_# domain) that the transmit path is waiting on.

tx_err_# indicates that a timestamp error has occurred, which is typically when data is not be able to be pushed down over the PCIe bus fast enough to transmit a packet at the correct time.

tx_empty_# indicates the PCIe FIFO is empty.

7.3.5 user_reg_if

Within the user_app, user_reg_if provides an address space to drive or read status of user logic. As the functionality of user_reg_if is the nearly identical to reg_if, please see Section 7.4 for information. The code comments within user_reg_if serve to provdie a template for adding user registers.

7.4 reg_if / user_reg_if

reg_if and user_reg_if are Verilog file that provides a register map for all FPGA functions, starting at address 0x8000 as referenced by software. Addresses below 0x8000 will be visible on the bus, but should be ignored, as they deal with PCIe transactions beyond the scope of this document.

The R/W data bus is 32 bits wide, and 14 address lines are provided. The FPGA considers each 32 register a single address, while software addresses registers byte-wise. Logically, this results in the software address's two least significant bits being truncated off. The first software address, b1000 0000 0000 0000 (x8000), is seen as b10 0000 0000 0000 by the 14 address lines on the FPGA. The second address, b1000 0000 0000 0100 (x8004) is seen as b10 0000 0000 0001 by the FPGA, and so on.

Within the module, bits [9:6] on the address bus are used to denote the logical bank of the register, and bits [5:0] denote the address. This mapping results in the third nibble of the address from software being the bank(i.e. x8000 is bank zero, x8100 is bank two, x8300 is bank three, and so on).

Banks 0-6 are used by the system, and it is recommended they not be modified, though PDK/SDK customers do have the ability to do so. These banks are located in reg_if.v, which is instantiated in the top level.

Bank 0 is a general purpose bank, and does not contain any clock domain crossing logic.

Banks 1-4 provide system control over each Rx/Tx channel, and are synced to the sample clock on the channel domain.

Banks 5-6 are currently unused, but reserved.

Banks 7-9 are user register space, and are located with user_reg_if.v, which instantiated within user_app.

Banks 8-15 are user register space, and are currently unused within user_reg_if.v. However, they are available for customer use within the user register space along with Banks 7-9 mentioned above.

Eight read and four write registers are provided in the example design in the 8700 register space, and are not synced to any specific clock domain. They are driven by the PCIe clock. The user may rename, add, or delete as desired to interface with user logic. If further register customization is desired, follow the template found in the source code. The provided read/write (software driven) registers and read-only (FPGA driven) registers are:

SW Address (hex)	FPGA Name	R / R/W
0x8700	reg_7_0	R/W
0x8704	reg_7_1	R/W
0x8708	reg_7_2	R/W
0x870C	reg_7_3	R/W
0x8710	reg_7_4	R
0x8714	reg_7_5	R
0x8718	reg_7_6	R
0x871C	reg_7_7	R

Table 3: User Registers

Please note that bit zero of x8708 is currently used to illustrate how a timestamp reset can be driven up to the top level via custom logic with a user_app register bit in the user_app. The following line of RTL is used for this in user_app.v:

```
assign timestamp_rst = user_reg_2_w[0];
```

If you do not want this functionality and would like to be able to write to 0x8708 without it affecting the timestamps, replace the line of code in user_app.v with the following:

```
assign timestamp_rst = 1'b0;
```

Banks 8 is synced to sample_clk_a via a FIFO. The write side of the FIFO is driven by the PCIe interface, and the read out from the FIFO is driven by the respective sample_clk. This gives a block of registers which can be used in sync-sensitive designs. It is recommended to only add registers and not modify the FIFO sync portion of the code. A template for adding registers is provided within the comments of user_reg_if.v.

If custom clock syncs are desired, it is suggested to use the 8700 register space, and sync the desired registers individually.

7.5 pcie_block

In mPCIe versions of Sidekiq, the pcie_block is contained in a static netlist. Due to changes in the Xilinx toolset and improvements made to the code, the M.2 and M.2 Stretch project allows the end user to customize the size and number of the transmit FIFOs in order to conserve FPGA resources, primarily BRAM. There are 150 BRAM(18k each) available. In a fully populated design, over 1/3 the BRAM resources are dedicated to transmit. If sample rates are low, or multiple transmit channels are not needed, configure the IP to best suit your design. If transmit underflows are reported in software, the size of the FIFO will need to be increased.

For M.2, M.2 USB, and M.2 Stretch, simply modify the verilog parameter settings that affect the

pcie_block instance in sidekiq_m2_top.v. These parameters are defined as top level parameters in sidekiq_m2_top.v and should be modifed at that level. As such, the build tcl scripts could also be modified to change the top level parameter settings for your particular build instead of modifying these parameters at the top of sidekiq_m2_top.v

The number of receive channels can be reconfigured to a single channel, saving 5 BRAM (18k) in the process. Valid number of receive channels is “10” or “01”. A value of “00” or no receive path is not currently supported. It is suggested to only remove the second Rx channel if the additional BRAM savings is necessary to support a user design.

Use the following table to configure the transmit FIFOs – other parameters should not be changed to avoid build issues.

#Channels / FIFO Size	Num Tx	Tx FIFO Size	Tx Capable	BRAM (18k) used
2 x 16K	“10”	“10”	“1”	56
2 x 8K	“10”	“01”	“1”	28
1 x 16K	“01”	“10”	“1”	28
1 x 8K	“01”	“01”	“1”	14
No Tx	“00”	“00”	“0”	0

Table 4: Tx FIFO Customization

7.6 timestamp_block

The timestamp_block.v is a Verilog file which handles driving and resetting the free-running counters (FRC) which serve to timestamp samples. It also controls signaling to start and stop receipt/transmission of data based on a PPS signal.

Time syncing multiple units is handled primarily by software; see the SDK documentation [2] for more information.

As it relates to the FPGA, the user_app can drive a reset, which will clear all free-running counters for as long as it is held high.

There are three counters present: frc_a, frc_b, and frc_sys. frc_a is driven by sample_clk_a, frc_b is dirven by sample_clk_b, and frc_sys is driven by the 40MHz reference clock. As long as reset is not held, these counters will increment as long as there is a clock present.

7.7 gpio_tristate

The gpio_tristate_bist.v block is a block used to implement the functionality of various GPIO lines. It is recommended to not modify this block.

8 Building Sidekiq mPCIe and Sidekiq M.2

There are two steps to follow when building custom designs for the Sidekiq platform.

1. Create the user_app
2. Build the project and generate the bitstream

Each of these will be addressed in this section.

8.1 Building a user_app for mPCIe and M.2

The reference design can be built out of the box with a standard user_app named user_app.v because raw I/Q samples pass through unmodified before being sent over PCIe. The user_app within the reference design can be used as a model for building custom user_apps. This section provides insight into building the included design, as well as how to generate a custom design to work with the included build scripts. The PDK user may customize the build flow to better match their existing processes if desired.

8.1.1 Reference Design for mPCIe

The **user_app** folder can be found in the root directory of the Sidekiq project file. This contains several files.

user_app.v contains the top level file. In this design, I/Q samples, clocks, and timestamp counters are passed straight through to the PCIe FIFOs.

user_reg_if.v contains read and write user registers, as discussed in Table 3.

user_app_rx.v and user_app_tx.v contain the receive and transmit processing blocks. These can be instantiated per-channel, if different apps are desired on different channels.

user_reg_bank_# contains a generated bank of registers, which can be modifed to interact and control/read status of a custom user_app.

user_app_sidekiq.ucf (user_app_sidekiq.xdc for M.2) is a constraint file which contains timing constraints for the clocks coming in from the AD9361 chips on mPCIE. On M.2, timing closure is easier to meet and the constraints are fixed as the max AD9361 rate.

8.1.2 Reference Design for M.2

The **user_app** RTL can be found in the hdl directory of the reference design. This contains several files.

hdl/user_app.v contains the top level file. In this design, I/Q samples, clocks, and timestamp counters are passed straight through to the PCIe FIFOs.

hdl/auto_gen/user_reg_if.v contains read and write user registers, as discussed in Table 3.

hdl/user_app_rx.v and hdl/user_app_tx.v contain the receive and transmit processing blocks. These can be instantiated per-channel, if different apps are desired on different channels.

hdl/auto_gen/user_reg_bank_# contains a generated bank of registers, which can be modifed to interact and control/read status of a custom user_app.

constraints/user_app_sidekiq.xdc is a constraint file which contains timing constraints for the clocks coming in from the AD9361 chips on mPCIE. On M.2, timing closure is easier to meet and the constraints are fixed as the max AD9361 rate.

8.1.3 Custom user_apps for mPCIe

To create a custom app, first create a copy of the **user_app** directory. While the new directory should have a new name, the user_app files should keep their names.

Create your design in user_app.v. The user has full use of Xilinx components, such as BRAM, DSP blocks, and PLLs. Most designs will not need to modify the user_app top level ports. If the top level ports are modified, /ref_design/hdl/sidekiq.top or sidekiq_m2_top will need to be modified so the instantiation of user_app.v will match the user's design.

Modify the build script, located in /ref_design/par/sidekiq/sidekiq_build_pdk.tcl for mPCIE, /ref_design/par/sidekiq/sidekiq_usb_build_pdk.tcl for mPCIe USB, or vivado_build.tcl in the top level directory for M.2, to include any extra submodules or IP that your design includes. If you do not include all necessary modules, the build will fail with an error indicating what needs to be added to the file.

If your design includes custom clocking that creates a faster clock to process samples, it is likely you will need to modify the user_app.ucf (or user_app.xdc for M.2) file to reflect your changes. The default sample clock constraint is 62 MHz, which reflects the max throughput which can be transferred via PCIe. If the user design does not require these faster sample rates, it is recommended to reduce this constraint to ease timing closure.

8.1.4 Custom user_apps for M.2

To create a custom app, first modify the user_app related files as necessary.

Create your design in hdl/user_app.v. The user has full use of Xilinx components, such as BRAM, DSP blocks, and PLLs. Most designs will not need to modify the user_app top level ports. If the top level ports are modified, hdl/sidekiq_m2_top will need to be modified so the instantiation of user_app.v will match the user's design.

Modify the build script vivado_build.tcl in the top level directory for to include any extra submodules or IP that your design includes. If you do not include all necessary modules, the build will fail with an error indicating what needs to be added to the file.

If your design includes custom clocking that creates a faster clock to process samples, it is likely you will need to modify constraints/user_app.xdc to reflect your changes. The default sample clock constraint is 62 MHz, which reflects the max throughput which can be transferred via PCIe. If the user design does not require these faster sample rates, it is recommended to reduce this constraint to ease timing closure.

8.2 Building the project and bitstream for mPCIe and M.2

The Sidekiq PDK reference design should be used as the starting point for building custom FPGA bitstreams targeting the Sidekiq. It is strongly recommended to first build the design without modification to ensure that the build environment is suitable for generating valid bitstreams. Furthermore, a suitable version control system should be used to facilitate development. A Git flow is already built into the included build script; if Git is not used, the git hash embedded in the bitstream will

be all zeros.

The Xilinx tools must be installed and properly configured in order generate a bitstream.

The Xilinx tools can be run in batch mode on the command line, or in GUI mode. There are several parameters that are passed in with batch mode that are not automatically configured in GUI mode which help identify bitstreams.

The first is the the Git hash; if Git version control is used, the top 7 digits of the current git hash, along with a clean/dirty flag in the most significant nibble, will be placed into address 0x8000 of the register map.

The second is the build date. The current date will be placed into address 0x8004.

There is a third register that contains the PDK version number at address 0x8008.

The user can modify the build script to pass in additional parameters to configure their own designs.

Once the build script is run, subsequent GUI builds will retain the parameters used in the previous batch mode build. Be aware that GUI builds may have inaccurate version numbers and build dates.

8.2.1 Linux

8.2.1.1 Sidekiq mPCIe (and USB)

A build script written in Tcl is provided to allow building of bitstreams through the command line. The reference design build script and project file are located in:

/ref_design/par/sidekiq/

To run in batch mode on the command line, run the following command in the directory with the build script to build for a PCIe-based host:

$ xtclsh sidekiq_build_pdk.tcl rebuild_project

or to build on USB, run:

$ xtclsh sidekiq_usb_build_pdk.tcl rebuild_project

This will create and set up a project, rebuild all of the required Xilinx cores, and perform the stages of synthesis, implementation, and bitstream generation using Xilinx ISE. The output bitstream file named sidekiq_top.bin will be generated.

For users who prefer to use a GUI, the provided sidekiq_top.xise will allow the project to be imported into the ISE application (but take note of the version number caution already mentioned above). This sidekiq_top.xise project file is for the standard mPCIe build. When first opening, ISE will look for the IP core .xise project files and will not find them because they need to be regenerated. This can easily be done by following the instructions provided by Xilinx here:

https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/pp_p_process_regenerate_all_cores.htm

(In short, click on the Spartan-6 device entry in the Hierarchy pane, then in the Processes pane, expand “Design Utilities”. Right-click “Regenerate All Cores” process, select “Process Properties”, set the “Regenerate Core” property, and double-click “Regenerate All Cores”).

ISE version 14.7 is used for building all releases.

8.2.1.2 Sidekiq M.2 (and USB)

A build script written in Tcl is provided to allow building of bitstreams through the command line.

To run in batch mode on the command line, run the following command in the top level directory with the build script:

$ vivado -mode batch -source vivado_build.tcl

This will create and set up a project, rebuild all of the required Xilinx cores, and perform the stages of synthesis, implementation, and bitstream generation using Vivado. A bitstream will be output in the top level directory, which will be named sidekiq_m2_pdk.bit (sidekiq_m2_usb_pdk.bit for the usb reference design).

For release 3.1 through 3.3, Vivado version 2015.2 is required. For 3.3 through 3.12.1, Vivado version 2016.4 is required. For all releases including and after 3.13.0, Vivado version 2018.3 is required.

8.2.2 Windows

8.2.2.1 Sidekiq mPCIe (and USB)

The xtclsh command on Windows is easiest to run through the ISE GUI. The tools can be run via command line in Windows, but ISE configuration is outside the scope of this document.

In the View -> Panels menu of ISE, Tcl console must be checked. In the Tcl Console window, run

> xtclsh sidekiq_build_pdk.tcl rebuild_project

or to build on USB, run:

$ xtclsh sidekiq_usb_build_pdk.tcl rebuild_project

For users who prefer to use a GUI, the provided sidekiq_top.xise will import the project in the ISE application (but take note of the version number caution already mentioned above). This sidekiq_top.xise project file is for the standard mPCIe build. When first opening, ISE will look for the IP core .xise project files and will not find them because they need to be regenerated. This can easily be done by following the instructions provided by Xilinx here:

https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/pp_p_process_regenerate_all_cores.htm

(In short, click on the Spartan-6 device entry in the Hierarchy pane, then in the Processes pane, expand “Design Utilities”. Right-click “Regenerate All Cores” process, select “Process Properties”, set the “Regenerate Core” property, and double-click “Regenerate All Cores”).

8.2.2.2 Sidekiq M.2 (and USB)

In the Vivado GUI, in the Tcl shell (Window → Tcl Console to open the shell), in the top level directory, run:

> source vivado_build.tcl

9 Programming the Sidekiq mPCie and Sidekiq M.2 Flash

Note that this section pertains only to Sidekiq mPCIe and Sidekiq M.2. For more detailed information about programming the PR capable M.2 Stretch, see the sections below.

The generated binary files can be loaded onto the hardware using software utilities run from the host system.

In the /test_apps/ directory, *prog_fpga* can be used to push the sidekiq_top.bin (for mPCIe) or sidekiq_m2_pdk.bit (for M.2) file down to the unit. No reboot is necessary after programming. Run *prog_fpga* with the *–help* argument to see usage. When programmed this way, a hard reboot will remove the image and the image stored on flash will be used.

The on-board flash memory can be programmed to set the Sidekiq to a specific image at boot, and reprogram the FPGA with that image automatically when power is restored. To program the flash memory, run *store_user_fpga* on the sidekiq_top.bin file for mPCIE based Sidekiq, or sidekiq_m2_pdk.bit for M.2 based Sidekiqs.

10 Building Sidekiq M.2 Stretch

There are three steps to follow in building custom designs for the Sidekiq M.2 Stretch platform.

1. Determine which build flow to use, SM or/and RM
    a. Static Module (SM) refers to sidekiq_m2s_pdk_static_vx_x_x.tar.gz
    b. Reconfigureable Module (RM) refers to sidekiq_m2s_pr_pdk_vx_x_x.tar.gz
2. Create your modified user_app, and replace existing user_app files in the RM or/and SM designs
3. Build the project and generate the bitstream

10.1 Terminology Used for M.2 Stretch

The following terminology applies when discussing the Sidekiq M.2 Stretch build and programming procedures over the new few sections.

1. PR = Partial Reconfiguration
2. SM = Static Module
3. RM = Reconfigureable Module (in this case, this is user_app)
4. The SM build flow relates to the following provided reference design (where vx_x_x is version):

    sidekiq_m2s_pdk_static_vx_x_x.tar.gz

5. The RM build flow relates to the following provided reference design (where vx_x_x is version):

    sidekiq_m2s_pr_pdk_vx_x_x.tar.gz

6. When the bitstreams are referenced with a version number (i.e. vx_x_x), these are referring to the stock bitstreams that have been provided as part of the reference design. For example:

    sidekiq_m2s_pr_pdk_${date}_${git_hash}_vx_x_x.bin

7. When the bitstreams are NOT referenced with a version number (i.e. vx_x_x), these are referring to bitstreams that have been generated by the user. For example:

    sidekiq_m2s_pr_pdk_${date}_${git_hash}.bin

10.2 Building a user_app for M.2 Stretch

The reference design can be built out of the box with a standard user_app named user_app.v because raw I/Q samples pass through unmodified before being sent over PCIe. The user_app within the reference design can be used as a model for building custom user_apps. This section provides insight into building the included design, as well as how to generate a custom design to work with the included build scripts. The PDK user may customize the build flow to better match their existing processes if desired.

10.2.1 Reference Design for M.2 Stretch

The user_app files can be found in the hdl directory in either the RM or SM reference design after uncompressing the respective tar file. The user_app files contained in each are the same. Once you have your custom user_app changes ready to go, you will need to replace the user_app files in the RM reference design or the SM reference design, or possibly in both locations, depending on which build flow you will be using. See the “Selecting the RM or SM Build Flow for M.2 Stretch” section below for determining which build flow you will be using.

user_app.v (located in the hdl folder) contains the top level file. In this design, I/Q samples, clocks, and timestamp counters are passed straight through to the PCIe FIFOs.

user_reg_if.v (located in the hdl/auto_gen folder) contains read and write user registers, as discussed in Table 3.

user_app_rx.v and user_app_tx.v (located in the hdl folder) contains the receive and transmit processing blocks. These can be instantiated per-channel, if different apps are desired on different channels.

user_reg_bank_#.v (located in the hdl/auto_gen folder) contains a generated bank of registers, which can be modifed to interact and control/read status of a custom user_app.

user_app_sidekiq.xdc (located in the constraints folder) is a constraint file which contains timing constraints for the clocks coming in from the AD9361. On M.2 Stretch, timing closure is easier to meet and the constraints are fixed at the max AD9361 rate.

10.2.2 Custom user_apps for M.2 Stretch

To create a custom app, simply modify any of the user_app related files mentioned above.

Create your design in user_app.v. The user has full use of Xilinx components, such as BRAM, DSP blocks, and PLLs. Most designs will not need to modify the user_app top level ports. If the top level ports are modified, sidekiq_m2_top.v will need to be modified in the SM so the instantiation of user_app.v will match the user's modified user_app design. Also, once the sidekiq_m2_top.v is modified, this will require the rebuild of the SM (**see below section for more information on this**).

Modify the build script vivado_build.tcl for the RM or/and vivado_build_route_place_synth.tcl for the SM based on your chosen flow to include any extra sub-modules or IP that your design might need to include. If you do not include all necessary modules, the build will fail with an error indicating what needs to be added to the file.

If your design includes custom clocking that creates a faster clock to process samples, it is likely you will need to modify the user_app.xdc file to reflect your changes. The default sample clock constraint is 62 MHz, which reflects the max throughput which can be transferred via PCIe. If the user design does not require these faster sample rates, it is recommended to reduce this constraint to ease timing closure.

10.3 Building the project and bitstream for M.2 Stretch

The Sidekiq M.2 Stretch PDK reference design should be used as the starting point for building custom FPGA bitstreams targeting the Sidekiq M.2 Stretch. It is strongly recommended to first build the design without modification to ensure that the build environment is suitable for generating valid

bitstreams. Furthermore, a suitable version control system should be used to facilitate development. A Git flow is already built into the included build script; if Git is not used, the git hash embedded in the bitstream will be all zeros.

The Xilinx tools must be installed and properly configured in order generate a bitstream.

The Xilinx tools can be run in batch mode on the command line, or in GUI mode. There are several parameters that are passed in with batch mode that help identify bitstreams.

The first is the the Git hash; if Git version control is used, the top 7 digits of the current git hash, along with a clean/dirty flag in the most significant nibble, will be placed into address 0x8000 of the register map.

The second is the build date. The current date will be placed into address 0x8004.

There is a third register that contains the PDK version number at address 0x8008.

The user can modify the build script to pass in additional parameters to configure their own designs.

10.4 Selecting the SM or RM (Or Both) Build Flow for M.2 Stretch

The fact that M.2 Stretch supports Partial Reconfiguration (PR) leads to several options that the user has in how to build the bitstreams. For M.2 Stretch, the user_app is a re-programmable module (RM). The rest of the design is considered to be the static module (SM). This is the reason that the Sidekiq M.2 Stretch PDK contains two different build reference designs. The SM build flow uses the sidekiq_m2s_pdk_static_vx_x_x.tar.gz reference design. The SM includes the full FPGA. However, at the end of the SM build process, user_app is replaced with an empty black box. The SM build flow then creates a sidekiq_m2s_pdk_static.dcp Vivado file that is used during the RM build process. The RM build flow uses the sidekiq_m2s_pr_pdk_vx_x_x.tar.gz reference design.

10.4.1 When to use the SM only build for M.2 Stretch

The following must all be true to use the SM build (sidekiq_m2s_pdk_static_vx_x_x.tar.gz) only:

1. Do not care about PR
2. Do not care if custom user_app is stored on flash
3. Want to have as many resources as possible available for your custom user_app

If the above conditions are true, then you can use the SM only build flow. You can completely ignore the RM reference design (sidekiq_m2s_pr_pdk_vx_x_x.tar.gz). However, to use this flow, you will need to make a few minor changes to the vivado_build_route_place_synth.tcl script located in the SM reference design. This script is set up assuming that it will be used with a combined SM/RM build flow. So, the fact that the SM build will only be used, will require a few modifications that will be described in the next section.

10.4.1.1 Building M.2 Stretch with the SM only build flow

Vivado version 2018.3 is currently required to build Sidekiq M.2 Stretch.

A build script written in Tcl is provided to allow building of bitstreams through the command line. It is included in the SM reference design (sidekiq_m2s_pdk_static_vx_x_x.tar.gz), and is called vivado_build_route_place_synth.tcl. To build with the SM only build flow, make the following changes to this script:

1. Remove “-verilog_define BUILD_WITH_PR=1” from the synth_design command
2. Comment out the line: read_xdc [file normalize "./constraints/pr_settings.xdc"]
3. Comment out the line: update_design -cell user_app -black_box
4. Comment out the line: lock_design -level routing
5. Replace PrYes, with PrNo in the set argv commad
6. Replace KeepBitStreamsNo, with KeepBitStreamsYes in the set argv commad

To run in batch mode on the command line (after you have copied your user_app changes to the SM reference design, and updated the .tcl with any new RTL, modules, or constraints), run the following command in the directory with the build script after the above changes have been made:

$ vivado -mode batch -source vivado_build_route_place_synth.tcl

This will run the build in non project mode, rebuild all of the required Xilinx cores, and perform the stages of synthesis, implementation, and bitstream generation using Vivado. One bitstream will be output in the current working directory, sidekiq_m2s_pdk_static_${date}_${git_hash}.bin. Use this bitstream with the store_fpga app to store this image to flash. This is a full image that contains the entire design. The user_app in this case is not a RM, so this is a traditional type of bitstream.

10.4.1.2 Programming M.2 Stretch with the SM only build flow

There is only one option when using the SM only build flow:

1. No PR (i.e. traditional style of bitstream):
    a. use store_user_fpga (this will be a complete fpga including a non-RM user_app):
        sidekiq_m2s_pdk_static_${date}_${git_hash}.bin

10.4.2 When to use the RM build only for M.2 Stretch

The following must all be true to use the RM build (sidekiq_m2s_pr_pdk_vx_x_x.tar.gz) only:

1. Want PR and/or Do not want custom user_app stored in flash
1. All changes will be contained inside user_app.v AND no port changes on user_app.v
2. Having less resources available for your custom user_app is o.k.

The RM build flow (sidekiq_m2s_pr_pdk_vx_x_x.tar.gz) should be used if all of your changes will be

contained inside user_app.v and you do not have any port level modifications that need to be made to user_app.v for your design. This method will use the pre-built SM, so there is no need to recreate it by building it with the SM flow. In this case, the SM design flow (sidekiq_m2s_pdk_static_vx_x_x.tar.gz) is not needed at all and can be completely ignored.

Another thing to consider when using the RM only flow, is that you may have less resources available for your user_app design with this flow. The RM has to be contained to specific layout resources indicated with the user_app pblock in vivado. When not using the PR flow, the user_app resources are free to be placed anywhere on the FPGA which typically allows it to have more resources available for use.

With this flow, if you so choose, you can create your own multiple custom versions of user_app. These can then be used to partial reconfigure the user_app for each implementation (one at a time) without reprogramming the entire FPGA (and without losing the PCIe bus during the partial reconfiguration of the RM module).

Also with this flow, if you do not want your custom user_app to be written to flash, simply write the stock .bin file provided with this release (sidekiq_m2s_pr_pdk_${date}_${git_hash}_vx_x_x.bin) using store_fpga. Then, use prog_fpga with your new RM partial .bit file (sidekiq_m2s_pr_pdk_${date}_${git_hash}_pblock_user_app_partial.bit). This way, your custom user_app will never be stored to flash and will be gone from the FPGA when power is lost.

10.4.2.1 Building M.2 Stretch with the RM build flow

Vivado version 2018.3 is currently required to build Sidekiq M.2 Stretch.

A build script written in Tcl is provided to allow building of bitstreams through the command line. It is included in the RM reference design (sidekiq_m2s_pr_pdk_v3_12_1.tar.gz), and is called vivado_build.tcl.

To run in batch mode on the command line (after you have copied your user_app changes to the RM reference design, and updated the .tcl with any new RTL, modules, or constraints), run the following command in the directory with the build script:

$ vivado -mode batch -source vivado_build.tcl

This will run the build in non project mode, rebuild all of the required Xilinx cores, and perform the stages of synthesis, implementation, and bitstream generation using Vivado. two bitstreams will be output in the current working directory, sidekiq_m2s_pr_pdk_${project_date}_${git_hash}.bin, and sidekiq_m2s_pr_pdk_${project_date}_${git_hash}_pblock_user_app_partial.bit.

Use sidekiq_m2s_pr_pdk_${project_date}_${git_hash}.bin with the store_fpga app to store this image to flash. This is a full image that contains the entire design including the SM and your version of the user_app RM module. Once the device is booted after the flash has been written, the fpga will be fully functional. At this point, you can use sidekiq_m2s_pr_pdk_${project_date}_${git_hash}_pblock_user_app_partial.bit with the prog_fpga app to do a partial reconfiguration on the user_app RM module. If you have several versions of user_app, they can all be loaded this way. Simply use prog_fpga to start using a different version of user_app verses that one that is currently programmed into the FPGA.

10.4.2.2 Programming M.2 Stretch with the RM build flow only

Choose the scenario that is best for your project:

1. Do not care about PR
    a. use store_user_fpga:
        sidekiq_m2s_pr_pdk_${date}_${git_hash}.bin
2. Want to take advantage of PR
    a. use store_user_fpga (must always do this first):
        sidekiq_m2s_pr_pdk_${date}_${git_hash}.bin
    b. use prog_fpga (this step not necessary if you only have one RM module):
        sidekiq_m2s_pr_pdk_${date}_${git_hash}_pblock_user_app_partial.bit
3. Want to take advantage of PR or/and Do not want custom user_app stored in flash
    a. use store_user_fpga (must always do this first, this is .bin provided with reference design):
        sidekiq_m2s_pr_pdk_${date}_${git_hash}_vx_x_x.bin
    b. use prog_fpga (this will be your custom user_app RM:
        sidekiq_m2s_pr_pdk_${date}_${git_hash}_pblock_user_app_partial.bit

10.4.3 When To Use A Combined SM and RM Build Flow For M.2 Stretch

The following must all be true to use the combined SM and RM build flow:

1. Want PR and/or Do not want custom user_app stored in flash
1. All changes will NOT be contained inside user_app.v and/or port changes on user_app.v
2. Having less resources available for your custom user_app is o.k.

The main motive behind using a combined SM/RM flow is that PR is needed, but there will be changes outside of user_app.v or on the user_app.v ports. In this case, you will need to copy your user_app changes to both the SM and RM reference designs. Then, the SM build will be run which will create a static .dcp. This .dcp will then need to be copied to the RM reference design before it is run. The details for this are contained in the next section.

10.4.3.1 Building M.2 Stretch with the combined SM and RM build flow

Vivado version 2018.3 is currently required to build Sidekiq M.2 Stretch.

A build script written in Tcl is provided to allow building of the bitstreams through the command line in

each of the SM (sidekiq_m2s_pdk_static_vx_x_x.tar.gz) and RM (sidekiq_m2s_pr_pdk_vx_x_x.tar.gz) reference designs.

Note that you will need to extract the two .tar.gz reference designs to two different working directories. Make sure that you have the changes needed for the custom user_app copied to both the SM and the RM reference designs.

After this is complete, the SM needs to be built. To run in batch mode on the command line, run the following command in the directory with the build script. Note that if you will be using programming option 3. described in the next section, you will need to change the KeepBitStreamsNo, with KeepBitStreamsYes in the set argv commad at the bottom of this tcl scipt.

```
$ vivado -mode batch -source  vivado_build_route_place_synth.tcl
```

This will run the build in non project mode, rebuild all of the required Xilinx cores, and perform the stages of synthesis, implementation, and bitstream generation using Vivado. two bitstreams will be output in the current working directory, sidekiq_m2s_pdk_static_${date}_${git_hash}.bin, and sidekiq_m2s_pdk_static_${date}_${git_hash}_pblock_user_app_partial.bit. For information on how to handle the bitstreams, see the next section.

This run will also produce a sidekiq_m2s_pdk_static.dcp vivado file (note that this file is written to the directory directly above your current SM run directory. If you would like to change the location where this file is written, modify the following command in the .tcl file, “write_checkpoint -force ../sidekiq_m2s_pdk_static.dcp”). This generated file needs to be copied to the RM reference design (the separate working directory where you uncompressed the RM reference design and added your custom user_app changes) and replace the existing file located in the RM reference design at:

netlists/sidekiq_m2s_pdk_static.dcp

Now, the RM is ready to be built. If you would like the new git_hash and date values from the SM build to be applied to the RM build output file naming conventions, update the contents of README_GIT_HASH_OF_STATIC_BUILD and README_BUILD_DATE_OF_STATIC_BUILD in the RM build with the values from README_GIT_HASH_OF_LAST_BUILD and README_BUILD_DATE_OF_LAST_BUILD from the SM build. Note, that the RM bitstream will always contain the git_hash and date from the SM build as this logic is implemented in the SM. So updating as described above, only affects the file name naming conventions of the RM build outputs.

Then, to run in batch mode on the command line, run the following command in the directory with the build script:

```
$ vivado -mode batch -source  vivado_build.tcl
```

This will run the build in non project mode, rebuild all of the required Xilinx cores, and perform the stages of synthesis, implementation, and bitstream generation using Vivado. two bitstreams will be output in the current working directory, sidekiq_m2s_pr_pdk_${date}_${git_hash}.bin, and sidekiq_m2s_pr_pdk_${date}_${git_hash}_pblock_user_app_partial.bit. For information on how to handle the bitstreams, see the next section. Please note the the following command at the bottom of the scripts/sidekiq_m2s_pr_post_build_hook.tcl tcl script: “pr_verify

./netlists/sidekiq_m2s_pdk_static.dcp ./sidekiq_m2s_pr_pdk.dcp". This checks that the bitstreams created are PR compatible. If successful, you should see the following message in the vivado.log file after this command has been run: "INFO: [Vivado 12-3253] PR_VERIFY: check points ./netlists/sidekiq_m2s_pdk_static.dcp and ./sidekiq_m2s_pr_pdk.dcp are compatible"

10.4.3.2 Programming M.2 Stretch with the combined SM and RM build flow

Choose the scenario that is best for your project:

1. Do not care about PR
    a. use store_user_fpga:
        sidekiq_m2s_pr_pdk_${date}_${git_hash}.bin
2. Want to take advantage of PR
    a. use store_user_fpga (must always do this first):
        sidekiq_m2s_pr_pdk_${date}_${git_hash}.bin
    b. use prog_fpga (this step not necessary if you only have one RM module):
        sidekiq_m2s_pr_pdk_${date}_${git_hash}_pblock_user_app_partial.bit
3. Want to take advantage of PR or/and Do not want custom user_app stored in flash
    a. use store_user_fpga (must always do this first, includes empty box for the user_app RM):
        sidekiq_m2s_pdk_static_${date}_${git_hash}.bin
    b. use prog_fpga (to load custom user_app RM module):
        sidekiq_m2s_pr_pdk_${date}_${git_hash}_pblock_user_app_partial.bit
    c. use prog_fpga (to load an empty user_app RM module):
        sidekiq_m2s_pdk_static_${date}_${git_hash}_pblock_user_app_partial.bit

11 Testing the Bitstream

The /test_apps/ directory contains several apps that can be scripted to run a regression test. As each user app will perform differently, the user may need to modify the source of each app to properly test their bitstream. See the SDK documentation [2] for descriptions of the provided test apps. Each app can be run with no parameters to view proper usage.

12 Using JTAG for Debug

A JTAG port is provided to facilitate debug for PDK customers. Xilinx's Chipscope application can be used to view internal FPGA signaling. Full Chipscope use is beyond the scope of this document. Both Xilinx and Digilent type programmers can be connected to the JTAG board. On these platforms, it is not intended to use JTAG to program a bitstream, as this will disrupt operation on the PCIe bus. Use the provide *prog_fpga* application to program new bitstreams.