Detailed Design Description for the *wave_gen* Lab Design

Introduction

The design used in these labs is a programmable waveform generator, also known as a signal generator.

Many analog electronic circuits are designed to process and modify analog input signals. These circuits are designed to perform desired transformations of the input signal to create an output signal. As part of designing and testing these circuits, test equipment is required that can generate input waveforms of various shapes at different frequencies. With a Digital-to-Analog Converter (DAC), it is possible to design an FPGA-based digital circuit that can generate these waveforms.

The waveform generator in this design is intended to be a “standalone” device that is controlled via a PC (or other terminal device) using RS-232 serial communication. The design described here implements the RS-232 communication channel, the waveform generator and connection to the external DAC, and a simple parser to implement a small number of “commands” to control the waveform generation. This design can be downloaded to the MicroBlaze™ Processor Development Kit, Spartan®-3E FPGA 1600E Edition development kit board.

The wave generator implements a look-up table of 1024 samples of 16 bits each in a RAM. Because the DAC used is only a 12-bit DAC, the four least significant bits of the RAM are ignored and all samples should use only the most significant 12 bits of the RAM.

The wave generator also implements three variables:

- nsamp: The number of samples to use for the output waveform. Must be between 1 and 1024.
- prescale: The prescaler for the sample clock. Must be 32 or greater.
- speed: The speed (or rate) for the output samples in units of the prescaled clock.

The wave generator can be instructed to send the appropriate number of samples once, cycling from 0 to nsamp-1 once, then stopping, or continuously, where it continuously loops the nsamp samples. When enabled, either once or continuously, the wave generator will send one sample to the DAC every (prescale x speed) clk_tx clocks.

The contents of the RAM, as well as the three variables, can be changed via commands sent over the RS-232 link, as can the mode of the wave generator. The wave generator will generate responses for all commands. **Table 1** shows the commands, their associated actions, and their responses. All input values are in hexadecimal. All responses include a "newline" at the end.

Cmd	***Input***	***Response***	***Description***
*W	*aaaavvvv*	-OK or -ERR	If *aaaa* is between 0 and 1023, the value *vvvv* is written to the RAM at location *aaaa* and "-OK" is output. Otherwise the data is discarded and "-ERR" is output
*R	aaaa	-hhhh ddddd or -ERR	If *aaaa* is between 0 and 1023, the data at RAM location *aaaa* is output both in hexadecimal and in decimal. Otherwise "-ERR" is output
*N	*vvvv*	-OK or -ERR	If *aaaa* is between 1 and 1024, the number of samples (nsamp) is set to the value *vvvv* and "-OK" is output. Otherwise "-ERR" is output
*P	vvvv	-OK or -ERR	If the *vvvv* is greater than or equal to the minimum allowable value for prescale (32), the prescaler (prescale) is set to vvvv and "-OK" is output. Otherwise the value is discarded and "-ERR" is output
*S	vvvv	-OK or -ERR	If the *vvvv* is greater than or equal to the minimum allowable value for speed (0), the speed (speed) is set to vvvv and "-OK" is output. Otherwise the value is discarded and "-ERR" is output
*n *p *s		-hhhh ddddd	The current value of nsamp, prescale, or speed is output in both hexadecimal and decimal
*G		-OK	Go: Trigger a single sweep through the samples
*C		-OK	Continuous: Turn on continuous looping through the samples
*H		-OK	Halt: Turn off continuous looping. The output generation will stop at the end of the current pass through the samples

Table 1. Wave Generator Commands

All other commands will result in "-ERR". In addition, when the parser is expecting a hexadecimal digit, and a non-hexadecimal character is entered, the parser will immediately abort the parsing of the current command and display "-ERR".

All response strings end with a "newline" (\n). The parser also echoes all input characters to the output.

The top level architecture of *wave_gen* is shown in **Figure 1**.

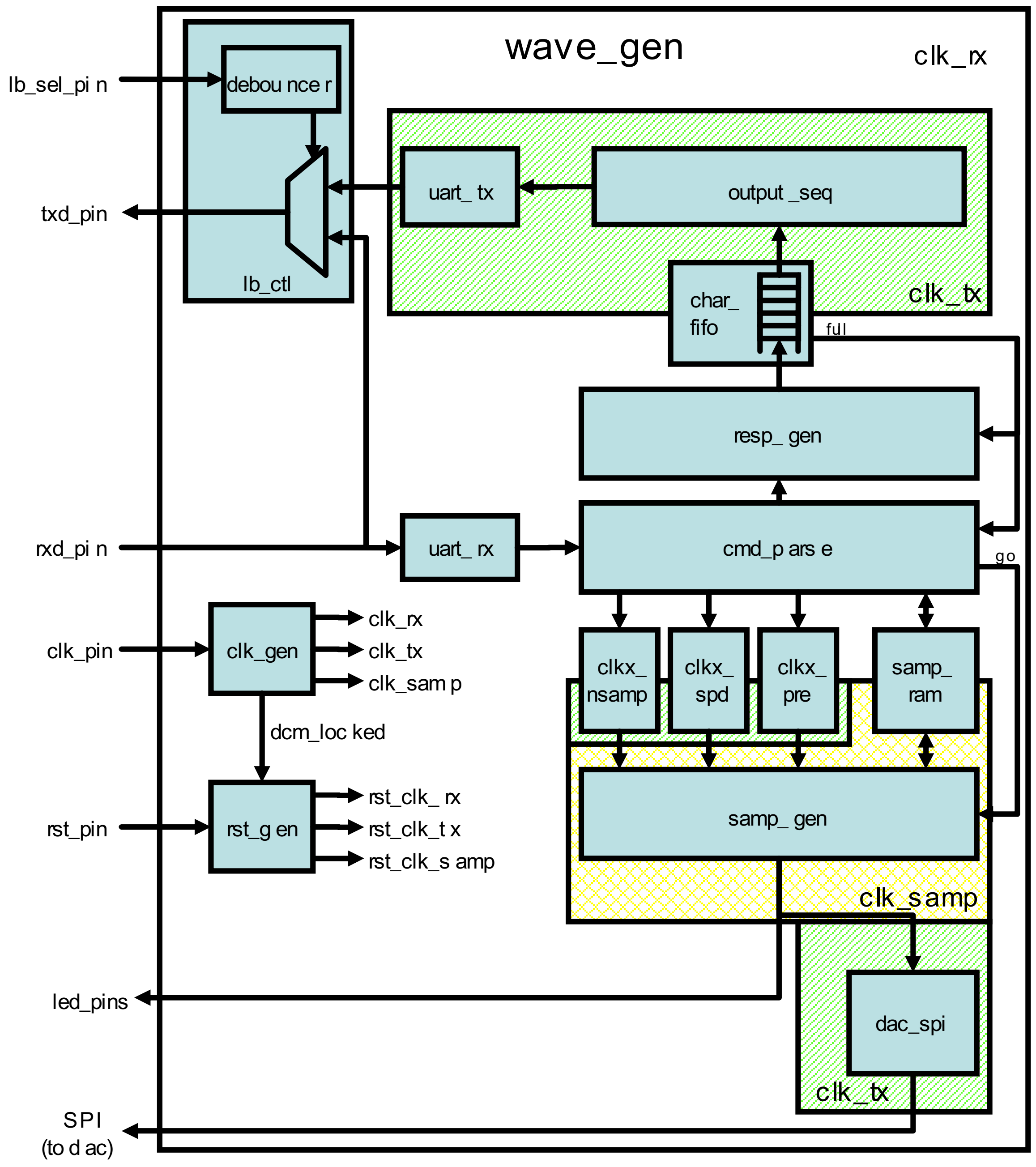

Figure 1. Top-Level Architecture

Clock Generator (*clk_gen*)

There are three clock domains within this design; ***clk_rx***, ***clk_tx***, and ***clk_samp***. All three clocks are generated inside the FPGA using a single 50-MHz clock input, coming in on ***clk_pin***. This module instantiates all the clocking resources required for generating these three clocks.

The first clock, ***clk_rx***, is the clock used for the receive portion of the Universal Asynchronous Receiver/Transmitter (UART) as well as other modules in the design (the command parser, response generator, etc…). This clock is used to clock internal FPGA resources; hence it is distributed using a BUFG.

The second clock, ***clk_tx***, is used to clock the transmit portion of the UART and the SPI generator for the digital to analog converter. In normal UARTs, there is no requirement to have a different clock for the receiver and transmitter; this is done to show examples of how to handle designs with multiple clocks. Even though ***clk_rx*** and ***clk_tx*** may come from the same on chip resource (and hence will be related clock domains), they will be treated as asynchronous clocks throughout the design.

The final clock, ***clk_samp***, is a decimated version of ***clk_tx*** that is used for clocking the sample generator and associated logic. This clock will be a divided version of ***clk_tx*** where the divider is determined by the prescaler value, which can be set by the user using the serial data interface. This clock is expected to be in phase with ***clk_tx***.

Since most evaluation boards have only one user clock frequency, all three clocks will be generated from the same source. To allow for ***clk_tx*** to run at a different rate than ***clk_rx***, on-chip frequency synthesis will be used.

For Spartan-6 designs, ***clk_rx*** and ***clk_tx*** will be generated from a DCM, with the CLK0 output used for ***clk_rx***, and the CLKFX output used for ***clk_tx***. In some implementations, these two clocks will be kept at the same frequency using M=2 and D=2, but for others, ***clk_tx*** will be at a lower frequency (31/32 times the frequency of ***clk_rx***), using M=31, D=32. Each of these clocks will use a BUFG to clock internal resources. The final clock ***clk_samp*** will be generated using a BUFGCE, also driven by the CLKFX output of the DCM.

For Virtex-6 designs, ***clk_rx*** and ***clk_tx*** will be generated using two outputs of a PLL. Again, depending on the implementation, both outputs will be at the same frequency, or at a ratio of 31/32. Both outputs will be driven directly to a BUFG. The last domain, ***clk_samp*** will be generated using a BUFHCE driven from the BUFG which generates ***clk_tx***. This will result in the same functionality as in the Spartan-6 design, but using one fewer global clock resources.

Because this design uses a synchronous reset methodology, it is essential to hold the design in reset until the DCM or PLL properly locks. The ***clock_locked*** signal is therefore fed to the reset generator.

The architecture of the clock generator in Spartan-6 is shown in Figure 2**,** and in Virtex-6 is shown in **Figure** 3. The portion inside the dotted line is contained in a sub-module called ***clk_core***, which can be created using the Clocking Wizard.

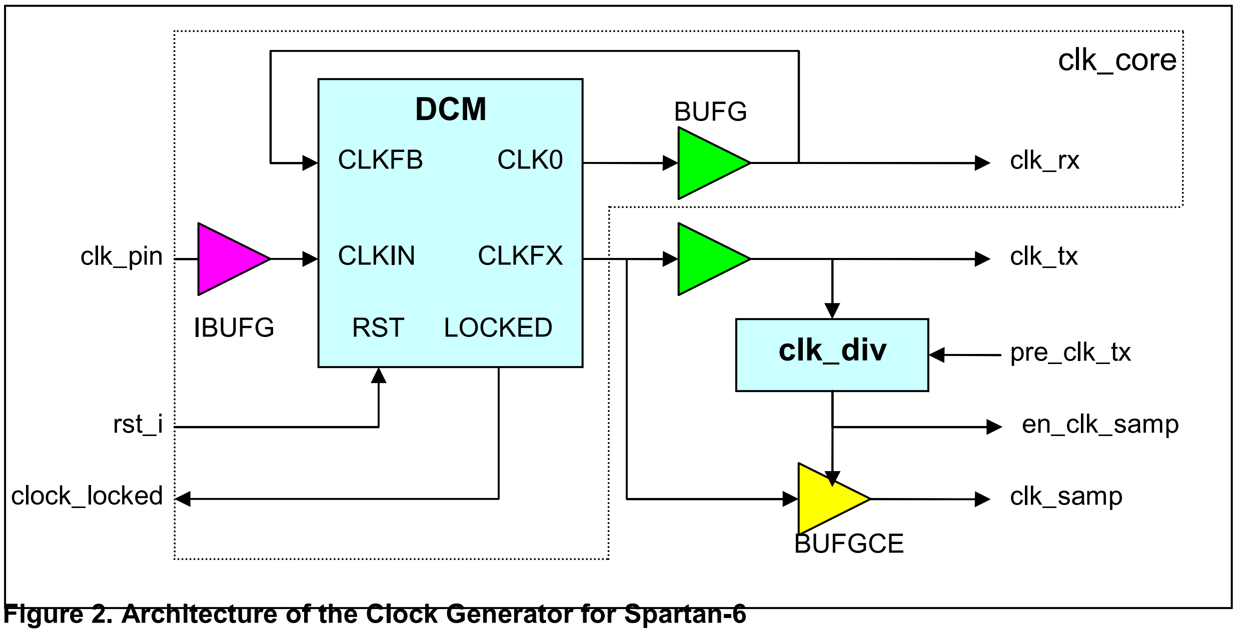

Figure 2. Architecture of the Clock Generator for Spartan-6

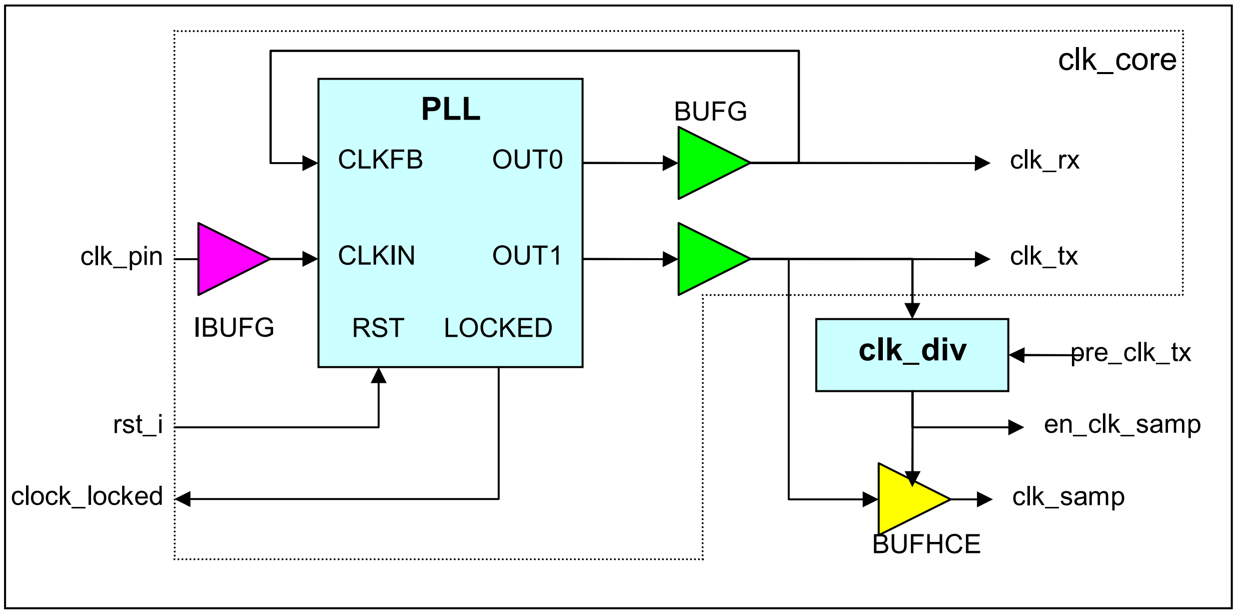

Figure 3. Architecture of the Clock Generator for Virtex-6

Signal	Dir	W	Src/Dst	Description
clk_pin	in	1	Pin	Primary clock input
rst_i	in	1	IBUFG	Asynchronous reset input for DCM
rst_clk_tx	in	1	rst_gen	Reset signal synchronized to clk_tx
pre_clk_tx	in	16	clkx_pre	Current prescaler value. Synchronized to clk_tx
clk_rx	out	1	Various	Clock for UART receiver and parser portion of design
clk_tx	out	1	Various	Clock for UART transmitter and output portion of design
clk_samp	out	1	Various	Clock for the sample portion of the design
en_clk_samp	out	1	Various	Indication that the next rising edge of clk_tx will coincide with the rising edge of clk_samp
dcm_locked	out	1	rst_gen	Locked signal from the DCM

Table 2. Inputs and Outputs of *clk_gen*

Clock Divider (*clk_div*)

This module is a simple divider. The division is set via the ***pre_clk_tx*** input to the module, which is the current value of the ***prescaler*** variable on the ***clk_tx*** domain. This module continuously counts from ***pre_clk_tx***–1 to 0, issuing the ***en_clk_samp*** output when the counter reaches 0. This signal enables BUFGCE and causes the next clock pulse on ***clk_tx*** to appear on ***clk_samp***. The counter automatically reloads to the prescale value whenever the counter reaches 0, or the prescale value changes. The ***en_clk_samp*** signal is also used to enable logic running on ***clk_tx*** that needs to run once per ***clk_samp*** period.

To ensure a proper synchronous reset for all flip-flops running on ***clk_samp***, ***en_clk_samp*** will be asserted during reset.

Signal	Dir	W	Src/Dst	Description
clk_tx	in	1	BUFG	Transmitter clock
rst_clk_tx	in	1	rst_gen	Reset signal synchronized to clk_tx
pre_clk_tx	in	16	clkx_pre	Current prescaler value. Synchronized to clk_tx
en_clk_samp	out	1	BUFGCE and Various	Enable for BUFGCE and also used by modules for managing crossing from the clk_samp to clk_tx domain.

Table 3. Inputs and Outputs of *clk_div*

Reset Generator (*rst_gen*)

This module generates the synchronized resets required for the three clock domains. An internal asynchronous reset is generated which is asserted when either the rst_i signal is asserted, or when the dcm_locked signal from the clock generator is not yet asserted. This internal reset is then synchronized using the ***reset_bridge*** module to generate versions synchronized to the three clock domains.

Signal	*Dir*	*W*	*Src/Dst*	*Description*
clk_tx	in	1	clk_gen	Transmitter clock
clk_rx	in	1	clk_gen	Receiver clock
clk_samp	in	1	clk_gen	Sample clock
rst_i	in	1	IBUF	Asynchronous reset input
dcm_locked	in	1	clk_gen	Locked signal from the DCM
rst_clk_tx	out	1	various	rst synchronized to clk_tx
rst_clk_rx	out	1	various	rst synchronized to clk_rx
rst_clk_samp	out	1	various	rst synchronized to clk_samp

Table 4. Inputs and Outputs of *clk_div*

Reset Bridge (*reset_bridge*)

This module is a specialized synchronizing circuit for resets. The input is an asynchronous, active-high reset, and the output is a synchronized version that asserts asynchronously (hence combinatorially from the asserting edge of the input), but deasserts synchronously.

Two flip-flops are used, both of which are asynchronously **preset** with the incoming signal. The D input of the first flip flop is tied low, and the second flip-flop is connected to the output of the first. On the deasserting edge of the incoming signal, the first flip-flop may become metastable (due to the potential violation of the recovery requirement of the flip-flop); the second flip-flop will re-clock that signal; thus reducing the chance that the final output also goes metastable.

Port	*Dir*	*W*	*Description*
clk_dst	in	1	Destination clock
rst_in	in	1	Asynchronous reset input
rst_dst	out	1	Reset signal, synchronized to clk_dst

Table 5. Inputs and Outputs of *reset_bridge*

The connections to the three instances of reset_bridge are shown in **Table 6**.

Port	*reset_bridge _clk_rx_i0*	*reset_bridge _clk_tx_i0*	*reset_bridge _clk_samp_i0*
clk_dst	clk_rx	clk_tx	clk_samp
rst_in	rst_i		
clk_dst	rst_clk_rx	rst_clk_tx	rst_clk_samp

Table 6. Port Connections for the Instances of *reset_bridge*

UART Receiver (*uart_rx*)

This module is the top level of the UART receiver. It consists of three sub-modules, a metastability hardening circuit, the Baud rate generator, and the UART controller. The architecture is shown in **Figure 4**.

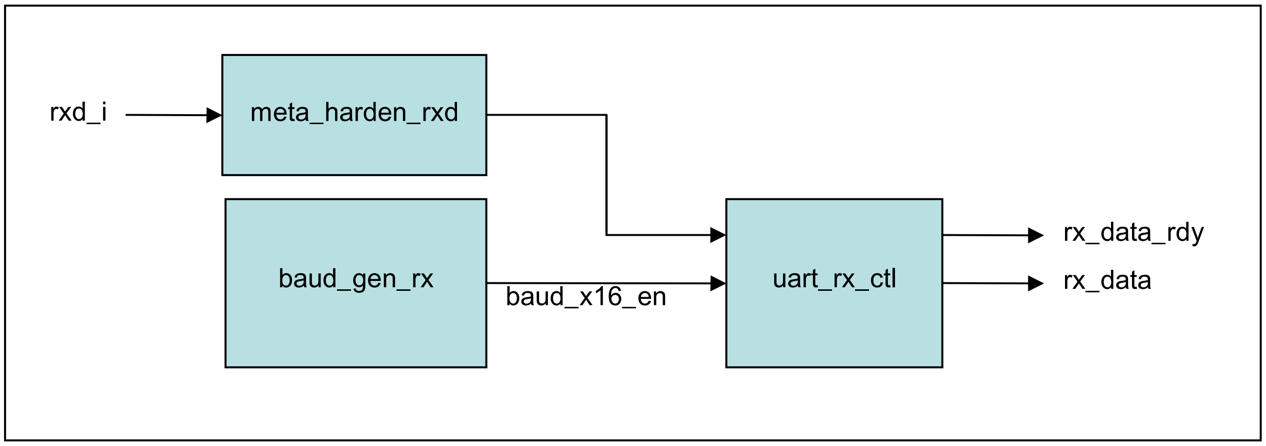

Figure 4. Architecture of the UART Receiver

RXD Metastability Hardener (*meta_harden_rxd*)

This module is an instantiation of the module ***meta_harden***, which is used more than once in the design. The ***meta_harden*** module implements a simple metastability hardening circuit. The asynchronous input signal is sampled into a first flip-flop, which may become meta-stable and is then re-sampled in a second flip-flop, which has a much lower probability of becoming metastable.

This instance brings the asynchronous input ***rxd_i*** into the ***clk_rx*** clock domain.

Port	***Signal***	***Dir***	***W***	***Src/Dst***	***Description***
clk_dst	clk_rx	in	1	clk_gen	Destination clock
rst_dst	rst_clk_rx	in	1	rst_gen	Reset signal, synchronized to clk_dst
signal_src	rxd_i	in	1	IBUF	Input signal, asynchronous to clk_dst
signal_dst	rxd_clk_rx	out	1	uart_rx_ctl	Output signal, synchronized to clk_dst

Table 7. Inputs and Outputs of *meta_harden_rxd*

Receive Baud Rate Generator (*baud_gen*)

This module is an instantiation of the module ***baud_gen***, which is used more than once in the design. The ***baud_gen*** module generates a 16x oversampled Baud enable, ***baud_x16_en***. This signal is asserted for one clock period with a frequency determined by the clock frequency and desired Baud rate, both of which are provided as parameters to this module. All flip-flops within the rest of the UART (that is, in ***uart_rx_ctl***) update only on clocks where ***baud_x16_en*** is asserted.

The number of clocks between assertions of this signal is determined by the formula (CLOCK_RATE)/BAUD_RATE*16, with appropriate rounding.

This instance of ***baud_gen*** generates the Baud rate for the RS-232 receiver.

Port	Signal	Dir	W	Src/Dst	Description
CLOCK_RATE		param			Frequency of the input clock
BAUD_RATE		param			Desired Baud rate
clk	clk_rx	in	1	clk_gen	Receiver clock
rst	rst_clk_rx	in	1	rst_gen	Reset signal, synchronized to clk_rx
baud_x16_en	baud_x16_en	out	1	uart_rx_ctl	Enable for flip-flops in uart_rx_ctl. Asserted for one clk_rx period, 16 time per bit period

Table 8. Inputs and Outputs of *baud_gen* for the Receiver

UART Receiver Controller (*uart_rx_ctl*)

This module implements an RS-232 receiver.

The synchronized receive signal is sampled at 16 times the Baud rate. When a high-to-low change is detected, this module waits for 8 ***baud_x16_en*** periods, and re-samples the incoming signal to confirm the START bit.

Assuming the START bit is confirmed, this module waits sixteen ***baud_x16_en*** periods and starts sampling the incoming data bits. Data bits are sampled once every sixteen ***baud_x16_en*** periods, starting with the least significant bit. After the last bit has been received, the module places the received byte on the ***rx_data*** output, and asserts ***rx_data_rdy***. Then the module waits sixteen additional ***baud_x16_en*** periods, de-asserts ***rx_data_rdy***, and samples the incoming data signal to check for the STOP bit. If the STOP bit is not correctly received, the ***frm_err*** signal is asserted.

Signal	Dir	W	Src/Dst	Description
clk_rx	in	1	clk_gen	Receiver clock
rst_clk_rx	in	1	rst_gen	Reset signal synchronized to clk_rx
baud_x16_en	in	1	baud_gen	Enable for flip-flops in uart_rx_ctl. Asserted for one clk_rx period, 16 time per bit period
rxd_clk_rx	in	1	meta_harden	RS-232 receive signal, synchronized to clk_rx
rx_data	out	8	cmd_parse	Received character. Only valid when rx_data_rdy is asserted
rx_data_rdy	out	1	cmd_parse	Indicates that a new character is available on rx_data
frm_err	out	1	Not Used	Indicates that a framing error has been detected. Not used in this design

Table 9. Inputs and Outputs of *baud_gen* for the Receiver

Command Parser (*cmd_parse*)

This module parses the incoming character stream, looking for valid commands, and either acts on them internally, or signals other modules to perform the appropriate action.

Each character arriving from the ***uart_rx*** is processed by this module and potentially causes a state change in the main state machine. Arriving characters are always sent to the response generator (***resp_gen***), as long as there is room in the character FIFO for the arriving character. If there is no room in the FIFO, the character is ignored (hence it will not cause a state change in this module). The state machine implements the commands described in **Table 1**, looking for commands that start with the asterisk, and then capturing the remaining characters of the command, performing error checking throughout the command. Possible errors after the asterisk are an illegal command (not W, R, N, P, S, n, p, s, G, H, or C), or an illegal hexadecimal digit when a digit is expected. As soon as an error is detected, the command is aborted, and the parser returns to idle.

This module has two independent communication channels to the response generator (***resp_gen***). Each accepted character (which is any character that arrives when the character FIFO is not full) is immediately sent to the response generator. The second communication channel initiates response strings; the possible strings are an error string, an acknowledgement string, or the outputting of a single numerical value. Because the FIFO is known not to be full during the echoing of the command, and pushing a single character into the FIFO can be accomplished in one clock, this module implicitly assumes the character is accepted. However, the response strings may be longer (thus requiring several clocks to push into the FIFO), and the response generator may need to wait for room in the FIFO to complete the response. Therefore, when requesting a response generation, the state machine in this module stalls until it receives an acknowledgement from the response generator.

When a legal command is fully processed, this module either acts on it internally, or signals another module to take action.

This module holds the current values of ***nsamp***, ***prescale***, and ***speed*** internally. These values will be updated when the appropriate command to update them is received.

Additionally, this module communicates with one port of ***samp_ram***. When the *W command is correctly received, this module will initiate a write to the RAM.

When a *R command is received, this module will read from ***samp_ram***, and send the value to the response generator. Similarly when the *n, *p, or *s command is received, this module will send the current value of ***nsamp***, ***prescale***, or ***speed*** to the response generator.

Finally, when the *G, *C, or *H commands are received, this module will signal the sample generator (***samp_gen***) appropriately.

Signal	***Dir***	***W***	***Src/Dst***	***Description***
PW	param			Number of clocks to assert pulses going to the ***clk_tx*** domain. Must be enough to guarantee that the resulting signal is asserted for at least 2 full ***clk_tx*** periods (for valid clock crossing). Should be set to 3 if ***clk_rx*** and ***clk_tx*** are similar frequencies
clk_rx	in	1	clk_gen	Receiver clock
rst_clk_rx	in	1	rst_gen	Reset signal synchronized to clk_rx
rx_data	in	8	uart_rx	Received character. Only valid when rx_data_rdy is asserted
rx_data_rdy	in	1	uart_rx	Indicates that a new character is available on rx_data
char_fifo_full	in	1	char_fifo	Indicates that the character FIFO is currently full. No characters can be accepted

Signal	***Dir***	***W***	***Src/Dst***	***Description***
send_char_val	out	1	resp_gen	Pulsed for one clock when a new character is ready to send to the user. The character is on ***send_char***
send_char	out	8	resp_gen	Character to send back to the user. Only valid when ***send_char_val*** is asserted
send_resp_val	out	1	resp_gen	Asserted when a new response is required. The response type is indicated on ***send_resp_type***. Remains asserted until ***send_resp_done*** is asserted
send_resp_type	out	2	resp_gen	Indicates the type of response required. Only valid when ***send_resp_type*** is asserted 00: Acknowledge 01: Error 10: Data – the data to send is on ***send_resp_data*** 11: Invalid type (not used)
send_resp_data	out	16	resp_gen	Data to send. Only valid when ***send_resp_val*** is asserted and ***send_resp_type*** is 10
send_resp_done	in	1	resp_gen	Pulsed for one clock when the requested response is complete. ***send_resp_val*** must be deasserted on the next clock
nsamp_clk_rx	out	11	clkx_nsamp	Current value of ***nsamp***
nsamp_new_clk_rx	out	1	clkx_nsamp	Pulsed for one clock when ***nsamp*** is changed
pre_clk_rx	out	10	clkx_pre	Current value of ***prescale***
pre_new_clk_rx	out	1	clkx_pre	Pulsed for one clock when ***prescale*** is changed
spd_clk_rx	out	10	clkx_spd	Current value of ***speed***
spd_new_clk_rx	out	1	clkx_spd	Pulsed for one clock when ***speed*** is changed
samp_gen_go_clk_rx	out	1	samp_gen	Asserted continuously between the receipt of a *C and *H command, or pulsed for a *PW* clocks on the receipt of a *G command
cmd_samp_ram_din	out	16	samp_ram	Write data to Sample RAM
cmd_samp_ram_addr	out	10	samp_ram	Address to Sample RAM
cmd_samp_ram_we	out	1	samp_ram	Write enable to Sample RAM
cmd_samp_ram_dout	in	16	samp_ram	Read data from Sample RAM

Table 10. Inputs and Outputs of *cmd_parse*

Response Generator (*resp_gen*)

This module generates textual responses to commands. The command parser (***cmd_parse***) has two interfaces for requesting responses: one for the echoing of incoming commands and another for the generation of response strings. When requested by the command parser, the appropriate character or string is pushed into the character FIFO (***char_fifo)*** one character at a time.

The command parser knows the state of the character FIFO and will not initiate the request for an echoed character while the FIFO is full (characters arriving when the FIFO is full will be dropped by the command parser). However, response strings may be requested at any time, including when the FIFO is full as well as at the same time that a character echo is requested. When both a character and response string are requested at the same time, the character echo takes precedence (as it will correspond to the last character of a command).

Pushing echoed characters into the FIFO is done unconditionally and immediately upon the request (because the FIFO is known not to be full at the time of the request).

Response strings will take several operations over several clocks to accomplish. There is an internal state machine and associated counters that will sequence the correct set of pushes into the character FIFO to complete the response string. If the FIFO becomes full at any point, the state machine will stall, waiting for more room to become available in the FIFO (the FIFO always empties at a constant rate through the UART transmitter).

The interface to the FIFO is ***not*** coming directly from flip-flops. The state machine in this module needs to know the status of the “full” flag of the FIFO immediately after each “push”—therefore the push (and associated data) will come from combinatorial logic. Because these signals go directly to the FIFO (through no other intervening combinatorial logic), this is not a performance issue.

After the last character has been successfully pushed into the character FIFO, this module will signal the command parser that the operation is complete, and that it can continue parsing new commands.

Signal	***Dir***	***W***	***Src/Dst***	***Description***
clk_rx	in	1	clk_gen	Receiver clock
rst_clk_rx	in	1	rst_gen	Reset signal synchronized to clk_rx
send_char_val	in	1	cmd_parse	Pulsed for one clock when a new character is ready to send to the user. The character is on ***send_char***
send_char	in	8	cmd_parse	Character to send back to the user. Only valid when ***send_char_val*** is asserted
send_resp_val	in	1	cmd_parse	Asserted when a new response is required. The response type is indicated on ***send_resp_type***. Remains asserted until ***send_resp_done*** is asserted
send_resp_type	in	2	cmd_parse	Indicates the type of response required. Only valid when ***send_resp_type*** is asserted 00: Acknowledge 01: Error 10: Data – the data to send is on ***send_resp_data*** 11: Invalid type (not used)
send_resp_data	in	16	cmd_parse	Data to send. Only valid when ***send_resp_val*** is asserted and ***send_resp_type*** is 10
send_resp_done	out	1	cmd_parse	Pulsed for one clock when the requested response is complete. ***send_resp_val*** must be deasserted on the next clock
char_fifo_dout	out	8	char_fifo	Character to push to FIFO. Not directly from a flip-flop

Signal	***Dir***	***W***	***Src/Dst***	***Description***
char_fifo_wr_en	out	1	char_fifo	Write enable (push) to FIFO. Not directly from a flip-flop
char_fifo_full	in	1	char_fifo	Character FIFO full indication

Table 11. Inputs and Outputs of *resp_gen*

Character FIFO (*char_fifo*)

The character FIFO is an 8-bit wide clock crossing FIFO. Characters are pushed into it by the response generator and are popped out by the UART transmitter. It is a first-word-fall-through FIFO, which allows simplified control in the UART transmitter.

The depth of the FIFO can be determined by what is efficient in the chosen architecture. For the Spartan-6 FPGA, the block RAM is 18 kb in size, so a 2048x8 FIFO is the logical implementation.

This module is generated via the CORE Generator™ tool (coregen).

UART Transmitter (*uart_tx*)

This module is the top level of the UART transmitter. It consists of two sub-modules: the Baud rate generator and the UART controller.

The baud rate generator is a second instantiation of ***baud_gen***, the baud rate generator used in the UART receiver. The ports are connected as shown in **Table 12**.

Port	***Signal***	***Dir***	***W***	***Src/Dst***	***Description***
CLOCK_RATE		param			Frequency of the input clock
BAUD_RATE		param			Desired Baud rate
clk	clk_tx	in	1	clk_gen	Transmitter clock
rst	rst_clk_tx	in	1	rst_gen	Reset signal, synchronized to clk_tx
baud_x16_en	baud_x16_en	out	1	uart_tx_ctl	Enable for flip-flops in uart_tx_ctl. Asserted for one clk_rx period, 16 time per bit period

Table 12. Inputs and Outputs of *baud_gen* for the Transmitter

UART Transmitter Controller (*uart_tx_ctl*)

This module implements an RS-232 transmitter.

Whenever the character FIFO (***char_fifo***) signals that it is not empty, this module will transmit the character at the head of the FIFO. Because the FIFO is first-word-fall-through, it is known that the character to be transmitted is already present at the output of the FIFO whenever the empty signal is not asserted.

The transmission of a character is relatively simple. A simple state machine passes through three states, corresponding to the START bit, the DATA bits, and the STOP bit. It remains in each state for 16 ***baud_x16_en*** periods, except for the DATA state, in which it remains for eight complete sets of 16 ***baud_x16_en*** periods. In the START state, the serial output ***txd_tx*** is pulled low; in the STOP state, it is high; and during the 8 data states, it sends bit 0 to 7 of the data, each for 16 ***baud_x16_en*** periods.

On the last ***baud_x16_en*** clock of the STOP period, the FIFO is popped, indicating that the character transmission is complete. Because the flops in this module are all enabled by ***baud_x16_en***, this would result in the pop signal to the FIFO being asserted for multiple clocks. In order to ensure that it is only asserted to the FIFO for one clock period, the ***uart_rd_en*** signal generated by this module is combinatorially derived using signals from the state machine ANDed with ***baud_x16_en***. This signal, even though it is the output of a module, does ***not*** come directly from a flip-flop.

Signal	***Dir***	***W***	***Src/Dst***	***Description***
clk_tx	in	1	clk_gen	Transmitter clock
rst_clk_tx	in	1	rst_gen	Reset signal synchronized to clk_tx
baud_x16_en	in	1	baud_gen	Enable for flip-flops in uart_tx_ctl. Asserted for one clk_tx period, 16 time per bit period
char_fifo_empty	in	1	char_fifo	Empty signal from the FIFO. Since the FIFO is first-word-fall-through, valid data is available whenever this signal is not asserted
char_fifo_rd_en	out	1	char_fifo	Read enable (pop) signal to the character FIFO. Combinationally derived.
char_fifo_dout	in	8	char_fifo	Character to send from the character FIFO. Valid whenever char_fifo_empty is not asserted
txd_tx	out	1	lb_ctl	RS-232 transmit signal

Table 13. Inputs and Outputs of *uart_rx_ctl*

Loopback Control (*lb_ctl*)

In addition to the normal RS-232 receive/transmit path, this design implements loopback functionality from the serial receive pin (***rxd_pin***) to the transmit pin (***txd_pin***). The loopback is purely combinatorial—the signal from the input buffer is routed directly to the output buffer through the loopback multiplexer.

The multiplexer is controlled by a slide switch on the board. When the switch is off (connected to ground), the loopback is disabled, and the normal transmit path is selected. When the switch is on (connected to VCC), the loopback is enabled.

Mechanical switches have significant "bounce"—they do not make clean transitions when the switch is moved. As such, the signal from the switch needs "debouncing," which is done using the ***debouncer*** module.

Signal	***Dir***	***W***	***Src/Dst***	***Description***
clk_tx	in	1	clk_gen	Transmitter clock
rst_clk_tx	in	1	rst_gen	Reset signal synchronized to clk_tx
lb_sel_i	in	1	IBUF	Switch input
txd_tx	in	1	uart_tx	RS-232 transmit line from the UART transmitter
rxd_i	in	1	IBUF	RS-232 receive line from the IBUF
txd_o	out	1	OBUF	Multiplexed RS-232 transmit signal

Table 14. Inputs and Outputs of *lb_ctl*

Switch Debouncer (*debouncer*)

This module implements a simple switch debouncer. This module generates an output signal based on the history of the switch input.

The input from the switch is first metastability hardened using the ***meta_harden*** module used elsewhere in the design. The synchronized signal, though, may still "bounce" back and forth between states for a significant number of clock periods when the switch position is changed. As a result, this module generates a “filtered” output that only changes state when the sampled value from the switch is a constant for a paramaterized number of clocks.

During each clock, the synchronized input is sampled. If it is the same as the current output state of the debouncer, a counter is pre-loaded with the value of the debouncing delay parameter. If the input is different than the current output state, the counter is allowed to decrement. If the counter reaches 0, this indicates that the input has been at the new state for the proper number of clocks, and the output is switched to the new state.

Port	***Signal***	***Dir***	***W***	***Src/Dst***	***Description***
	FILTER	param			Number of clocks required to switch state
clk	clk_tx	in	1	clk_gen	Transmitter clock
rst	rst_clk_tx	in	1	rst_gen	Reset signal synchronized to clk_tx
signal_in	switch_i	in	1	IBUF	The raw input from the switch
signal_out	lb_sel	out	1	loopback MUX	The debounced output of the switch

Table 15. Inputs and Outputs of *debouncer*

Bus Clock Crossers (*clkx_nsamp*, *clkx_pre*, *clkx_speed*)

These three blocks are instances of the module ***clkx_bus***.

These three sets of busses (***nsamp***, ***speed***, and ***prescale***) are generated by the command parser, which is running on **clk_rx**, but are also needed in blocks that are running on ***clk_tx*** and/or ***clk_samp*** (which are synchronous to each other). As a result, a clock crossing circuit is required.

Because these buses can only be updated by the command parser and only in response to a command from the RS-232 serial input, it is known that they can only change once in a very large number of clocks. Each of these signals is accompanied by a “***_new***” signal, which indicates when the bus has changed—at all other times, the bus in question is stable.

This module detects the pulse of the ****_new*** signal, pulse stretches it to the required length to ensure that it can be crossed into the destination clock domain, where it can be edge detected. When the rising edge of the ****_new*** signal is detected in the destination domain, it is known that you are “sufficiently” past the change in the bus (so the bus is stable), and (because the bus cannot change more than once every N clocks, where N is a large number), it is “sufficiently” far from the next change. As a result, the bus is stable in the source clock domain and can be sampled in the destination clock domain. This module samples the bus and generates a pulse on the destination domain to indicate that the signal has changed.

In order to ensure that the reset values of these signals in the source domain propagate to the destination domain, this module will update the output bus based on the input bus during reset. This is legal because the reset signal in all domains is based on the same input signal—therefore, all domains will be in reset at the same time. Because the domain is in reset, all signals are static (not changing). Hence, it is legal to cross data buses from one domain to another.

Port	***Dir***	***W***	***Src/Dst***	***Description***
PW	param			Number of clocks to assert pulses going to the ***clk_dst*** domain. Must be enough to guarantee that the resulting signal is asserted for at least 2 full ***clk_dst*** periods (for valid clock crossing). Should be set to 3 if ***clk_dst*** and ***clk_src*** are similar frequencies
WIDTH	param			Width of the bus to be crossed

Port	***Dir***	***W***	***Src/Dst***	***Description***
clk_src	in	1	clk_gen	Source clock
rst_clk_src	in	1	rst_gen	Reset signal synchronized to clk_src
clk_dst	in	1	clk_gen	Source clock
rst_clk_dst	in	1	rst_gen	Reset signal synchronized to clk_src
bus_src	in	WIDTH	various	Bus to be crossed, synchronous to clk_src
bus_new_src	in	1	various	Pulsed for one clk_src period when bus_src changes
bus_dst	out	WIDTH	various	Bus crossed to clk_dst domain
bus_new_dst	out	1	various	Pulsed for one clk_dst period when bus_dst changes

Table 16. Inputs and Outputs of *clkx_bus*

The connections to the three instances of clkx_bus are shown in Table 17.

Port	***clkx_nsamp***	***clkx_pre***	***clkx_spd***
PW	3		
WIDTH	11	16	16
clk_src	clk_rx	clk_rx	clk_rx
rst_clk_src	rst_clk_rx	rst_clk_rx	rst_clk_rx
clk_dst	clk_tx	clk_tx	clk_tx
rst_clk_dst	rst_clk_tx	rst_clk_tx	rst_clk_tx
bus_src	nsamp_clk_rx	pre_clk_rx	spd_clk_rx
bus_new_src	nsamp_new_clk_rx	pre_new_clk_rx	spd_new_clk_rx
bus_dst	nsamp_clk_tx	pre_clk_tx	spd_clk_tx
bus_new_dst	nsamp_new_clk_tx	pre_new_clk_tx	spd_new_clk_tx

Table 17. Port Connections for the Instances of *clkx_bus*

Sample RAM (*samp_ram*)

The sample RAM is a 1024x16 true dual port SRAM. One port is read and read by the command parser, and the other port is read by the sample generator. Because these two modules are running on different clocks, the two ports of the RAM are run asynchronously.

This module is generated via the CORE Generator tool (coregen).

Sample Generator (*samp_gen*)

This module sequences out the samples from the sample RAM. The majority of this module runs on ***clk_samp***; the decimated clock that is enabled for one ***clk_tx*** period out of every ***prescale*** clocks. When the sample generator is enabled, it issues one sample in ***speed clk_samp*** periods.

The module is enabled using the ***samp_gen_go_clk_rx*** signal from the command parser, which is either pulsed for three clocks when a *G command is issued or is continuously asserted between a *C and *H command. This signal must first be synchronized to the ***clk_tx*** domain using the ***meta_harden*** module. Once synchronized, its state must be captured in the ***clk_tx*** domain to ensure that it is held until the next rising edge of ***clk_samp***. This is done using an internal signal running on ***clk_tx*** that is set whenever the synchronized version of ***samp_gen_go*** is asserted and cleared whenever ***samp_gen_go*** is zero on the rising edge of ***clk_samp***, which is indicated using the ***clk_samp_en*** signal from the clock generator.

The main state machine in this module will have two states: IDLE and RUNNING. When IDLE, it will look for the synchronized ***samp_gen_go*** or the captured version and start a sweep through the ***nsamp*** samples. After the sweep is complete, it will re-examine the synchronized version of ***samp_gen_go*** to determine if it should perform another sweep; if clear, it will return to the IDLE state.

Once enabled, this module will sweep through the sample RAM starting at 0, and ending at ***nsamp***-1, with one sample generated every ***speed*** clocks. The sample will be issued along with a valid signal indicating that a sample is valid. Both the sample and the valid signal will be asserted for one full ***clk_samp*** period.

The most significant eight bits of the sample are also sent out on the signal ***led_out*** and sent to the four LEDs on the board, and the four LEDs on the optional daughter card..

Signal	***Dir***	***W***	***Src/Dst***	***Description***
clk_tx	in	1	clk_gen	Transmitter clock
rst_clk_tx	in	1	rst_gen	Reset signal synchronized to clk_tx
clk_samp	in	1	clk_gen	Sample clock
rst_clk_samp	in	1	rst_gen	Reset signal synchronized to clk_samp
nsamp_clk_tx	in	11	clkx_bus	Current value of nsamp on clk_tx
spd_clk_tx	in	16	clkx_bus	Current value of speed on clk_tx
en_clk_samp	in	1	clk_gen	Indication that the next rising edge of clk_tx will coincide with the rising edge of clk_samp
samp_gen_go_clk_rx	in	1	cmd_gen	Asserted continuously between the receipt of a *C and *H command, or pulsed for a *PW* clocks on the receipt of a *G command
spd_clk_tx	in	16	clkx_spd	Current value of speed, on clk_tx
nsamp_clk_tx	in	11	clkx_nsamp	Current value of nsamp, on clk_tx
samp_gen_samp_ram_addr	out	10	samp_ram	Address to Sample RAM
samp_gen_samp_ram_dout	in	16	samp_ram	Read data from Sample RAM
samp	out	16	dac_spi	The current sample being output. Only valid when samp_val is asserted
samp_val	out	1	dac_spi	A valid sample is being output. Asserted for one clk_samp period for each sample
led_out	out	8	MUX	When enabled by setting the appropriate parameter at the top level, these signals are sent to the four LEDs on the board and the four LEDs on the optional daughter card

Table 18. Inputs and Outputs of *samp_gen*

DAC SPI controller (*dac_spi*)

This module takes the samples generated by the sample generator and serializes them out to the Linear Technologies LTC2624 Digital to Analog Converter (DAC) using the SPI protocol. Because the data returned from the DAC is not needed, this module only implements the write portion of the SPI protocol.

For each new sample (signified by the assertion of ***samp_val*** during the rising edge of ***clk_tx*** while ***en_clk_samp*** is asserted), this module will begin an SPI write cycle. All SPI writes send samples to DAC channel A. Because the sample (on ***samp***) will remain valid throughout the full 32 ***clk_tx*** periods of the SPI cycle, it does not need to be captured internally.

This module generates the chip select to the DAC (***dac_cs_n_o***), as well as the SPI clock (***spi_clk_o***), and the SPI output data (***spi_mosi_o***). The SPI clock runs at the same frequency as ***clk_tx*** (either 50 MHz or 48.44 MHz), and is generated using an output DDR flip-flop to echo the transmit clock onto this pin. The clock sent will be 180 degrees out of phase with ***clk_tx*** by sending a ‘1’ after the falling edge of ***clk_tx***. This inversion allows for ½ clock of setup time and ½ clock of hold time of the SPI data with respect to the SPI clock, which is sufficient for this device. The clock will only run during a DAC transfer; thus for 32 consecutive clock pulses for each sample sent to the DAC.

The DAC chip select (***dac_cs_n_o***), which is active low, will be asserted during the entire transfer but deasserted outside the transfer. It will make all transitions on the rising edge of ***clk_tx*** to ensure that it meets the setup and hold requirement to ***spi_clk***. On the MicroBlaze1600E board, which has this DAC on board, the design must ensure that the chip selects to all other devices on the board are disabled; hence, it must tie the ***spi_ss_b_o***, ***amp_cs_o***, ***sf_ce0_o***, and ***fpga_init_b*** signals to 0 and the ***ad_conv_o*** signal to 1. For other boards, it is assumed that the DAC is on a daughter card, and that no other SPI devices exist on this bus, hence these other chip selects are not implemented.

The data output (***spi_mosi_o***) will also make all transitions on the rising edge of ***clk_tx*** to ensure setup and hold requirements are met. The complete command cycle will consist of serializing the 4-bit command (0011 – “Write and Update”), followed by the 4-bit address (0000 to access DAC channel A), followed by the 16 bits of data provided by the sample generator. Because the LTC2624 on this board is a 12-bit DAC only, the least significant 4 bits of the sample will be ignored by the DAC.

Signal	***Dir***	***W***	***Src/Dst***	***Description***
clk_tx	in	1	clk_gen	Transmitter clock
rst_clk_tx	in	1	rst_gen	Reset signal synchronized to clk_tx
en_clk_samp	out	1	clk_gen	Indication that the next rising edge of clk_tx will coincide with the rising edge of clk_samp
samp	in	16	samp_gen	The current sample being output. Only valid when samp_val is asserted
samp_val	in	1	samp_gen	A valid sample is being output. Asserted for one clk_samp period for each sample
spi_clk_o	out	1	OBUF	SPI clock
spi_mosi_o	out	1	OBUF	SPI master-out-slave-in data
dac_cs_n_o	out	1	OBUF	DAC SPI chip select – active low
dac_clr_n_o	out	1	OBUF	DAC clear – active low
spi_ss_b_o	out	1	OBUF	SPI serial flash chip select – tied to 1 (disabled)
amp_cs_o	out	1	OBUF	SPI programmable pre-amplifier chip select – tied to 1 (disabled)
sf_ce0_o	out	1	OBUF	StrataFlash chip select – tied to 1 (disabled)
fpga_init_b_o	out	1	OBUF	SPI platform flash chip select – tied to 1 (disabled)
ad_conv_o	out	1	OBUF	SPI ADC chip select – tied to 0 (disabled)

Table 19. Inputs and Outputs of *dac_spi*