# 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 any FPGA development board with a DAC, either on the board or on a daughter card. The DAC is assumed to be compatible with the Linear Technologies LTC2624.

The wave generator implements a look-up table of 1024 samples of 16 bits each in a 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. The table below shows the commands, their associated actions, and their responses. All input values are in hexadecimal.

| 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 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 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 |

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 the following figure.

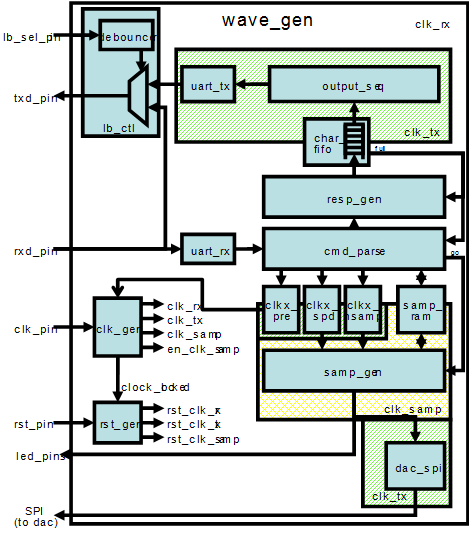


Figure 1: Top-Level Architecture

## Clock Generator (clk\_gen)

There are three clock domains within this design: clk\_rx, clk\_tx, and clk\_samp. This module instantiates all the clocking resources required for generating these three clocks. All three clocks are derived from a single differential clock input, coming in on clk\_pin\_p and clk\_pin\_n. The Kintex-Ultrascale FPGA board has 300MHz differential clock oscillator input.

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 (DAC). 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.

Because 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 Kintex-Ultrascale FPGA designs, clk\_rx and clk\_tx will be generated from a MMCM, 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 last domain, clk\_samp, will be generated using a BUFHCE driven from the BUFG, which generates clk\_tx.

Regardless of the architecture used, clk\_samp is a decimated version of clk\_tx, enabled for only one out of every pre\_clk\_tx clock cycles. Due to the checking done by the parser, pre\_clk\_tx can never be set to a value of less than 32. This means that all paths between synchronous elements running on clk\_samp are 32 cycle multi-cycle paths.

Because this design uses a synchronous reset methodology, it is essential that the design be held in reset until the MMCM properly locks. The clock\_locked signal is therefore fed to the reset generator.

The architectures of the clock generator in the Kintex-Ultrascale FPGA is shown below. The portion inside the dotted line is contained in a submodule called clk\_core, which can be created via the Clocking Wizard.

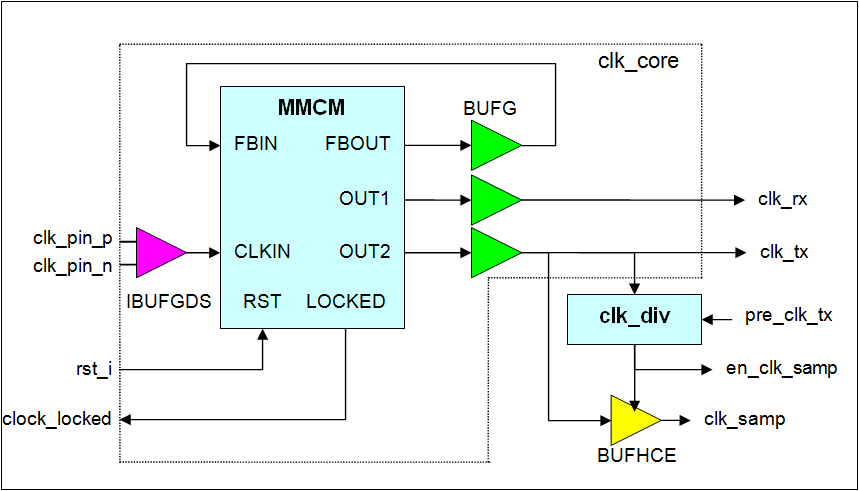


Figure 2: Architecture of the Clock Generator for the Kintex-Ultrascale FPGA

| Signal | Dir | W | Src/Dst | Description |
| --- | --- | --- | --- | --- |
| clk\_pin\_p | in | 1 | Pin | Primary clock positive input pin |
| clk\_pin\_n | in | 1 | Pin | Primary clock negative input pin |
| 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 |
| clock\_locked | out | 1 | rst\_gen | Locked signal from the DCM |

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 or BUFHCE 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. 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 |

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 clock\_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 |
| clock\_locked | in | 1 | clk\_gen | Locked signal from the clock generator |
| 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 |

Inputs and Outputs of rst\_gen

### 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. A constraint is required between these two flip-flops to ensure sufficient time for metastability resolution.

| 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 |

Inputs and Outputs of rst\_bridge

The connections to the three instances of reset\_bridge are shown below.

| 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 | | |
| rst\_dst | rst\_clk\_rx | rst\_clk\_tx | rst\_clk\_samp |

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 submodules, a metastability hardening circuit, the Baud rate generator, and the UART controller. The architecture is shown in the figure below.

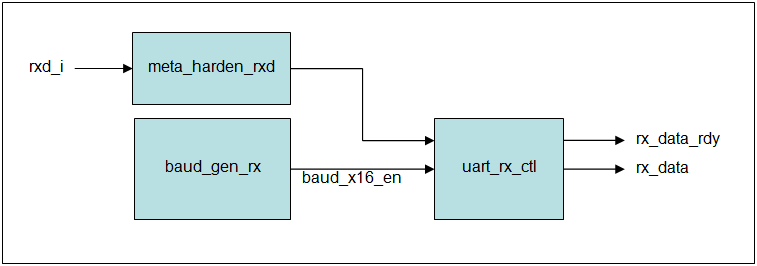


Figure 3: Architecture of the UART Receiver

### 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. A constraint is required between the two flip-flops to ensure sufficient time for metastability resolution.

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 |

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. This results in all paths between synchronous elements in uart\_rx\_ctl being n-cycle multi-cycle paths where n is determined by the number of clocks between assertions of baud\_x16\_en.

This is determined by the following formula: n=(CLOCK\_RATE)/BAUD\_RATE\*16, rounded to the nearest integer (with 0.5 rounded up). For most applications, the BAUD\_RATE is to 115,200, and the CLOCK\_RATE is the rate of clk\_rx, which is the same as the frequency of the primary input clock (on clk\_pin).

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 |

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 eight 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.

Because all flip-flops in this module are enabled by baud\_x16\_en, all paths between flip-flops in this module are multi-cycle paths based on the rate of baud\_x16\_en, which is described in the Receive Baud Rate Generator module.

| 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 |

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 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 the Wave Generator Commands table (see "[Introduction](#O_34904)" on page 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 acknowledgment 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 acknowledgment from the response generator.

In the case where a the command requires a numeric response (either a \*R command, which reads a sample value from the RAM, or a \*n, \*s, or \*p command which prints out the current value of the corresponding register), the 16-bit value is placed on the output bus send\_resp\_data (which comes directly from flip-flops), and is held until the next command (which will be at least several thousand clocks later, which is the time required to receive at least two ASCII characters).

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 |
| 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 |

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).

Three types of responses exist, as determined by send\_resp\_type. Acknowledge responses (-OK), error responses (-ERR), and data responses. Data responses consist of a dask, followed by the four digit hexadecimal number followed by a space, followed by the five digit decimal representation. The hexadecimal to decimal conversion is done by the submodule to\_bcd.

Because there are at least six clocks between the assertion of send\_resp\_data, and the time the decimal version of the data is required (one clock to push each of the six characters preceding the first decimal digit), the decimal conversion is constructed as a two-cycle, multi-cycle path; the output flip-flops (bcd\_out) are only updated on the second clock after the assertion of send\_resp\_val.

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 (send\_resp\_done), 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 |
| 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 |

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.

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

The depth of the FIFO is at least 2048 bytes; it may be slightly more due to the extra entries provided by the first-word-fall-through capabilities of the CORE Generator tool FIFOs.

## UART Transmitter (uart\_tx)

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

### Transmit Baud Rate Generator (baud\_gen)

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 the table below.

This module generates a second baud\_x16\_en signal (local to the UART transmitter). This signal is asserted one out of every n clk\_tx cycles, where n is determined by the following formula: n=(CLOCK\_RATE)/BAUD\_RATE\*16, rounded to the nearest integer (with 0.5 rounded up). For most applications, the BAUD\_RATE is to 115,200, and the CLOCK\_RATE is the rate of clk\_tx, which is either the same as clk\_rx, or the frequency of clk\_rx multiplied by 31/32.

| Port | Signal | Dir | W | Src/Dst | Description |
| --- | --- | --- | --- | --- | --- |
| CLOCK\_RATe |  | param |  |  | Frequency of the input clock |
| BAUD\_RATE |  | param |  |  | Desired Baud rate |
| clk | clk\_txt | in | 1 | clk\_gen | Transmitter clock |
| rst | rst\_clk\_tx | in | 1 | rst\_gen | Reset signal, synchronized to clk\_tx |
| baud\_x16\_en | baud\_16\_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 |

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.

Like the UART Receive Controller, all flip-flops in this module are enabled only when baud\_x16\_en (generated by baud\_gen for the transmitter), and hence all paths in this module are n-cycle, multi-cycle paths, based on the rate of baud\_x16\_en, described in the Transmit Baud Rate Generator.

| 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. Combinatorially 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 |

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 |
| txt\_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 |

Inputs and Outputs of lb\_ctl

## 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. The clock crossing of the stretched pulse is done using two flip-flops implemented in an instance of the meta\_harden; a constraint is required to ensure that sufficient time is allowed for metastability resolution.

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 |
| 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 |

Inputs and Outputs of clkx\_bus

The connections to the three instances of clkx\_bus are shown in the table below.

| Port | clk\_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 |

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.

## 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. Like other instances of meta\_harden, a constraint is required to ensure sufficient time for metastability resolution. 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\_den | 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 |

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, 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. Due to the DDR nature of this spi\_clk output signal, a specific constraint is required for this pin.

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.

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 |

Inputs and Outputs of dac\_spi