ImpLiFi FPGA Receiver Documentation

Jethro Reimann

March 13^{th}, 2024

1 Transmission Encoding Scheme

1.1 Packet Level Format

Each packet sent over ImpLiFi consists of 228 data bytes, 16 Reed-Solomon Error Correction bytes, and 11 "locator" bytes, for a total packet length of 255 bytes. This gives a Reed-Solomon notation $RS(255,239)$. Locator bytes are static predetermined bytes situated throughout the packet that allows the receiver to know when a packet has been fully received. The locator bytes are positioned at the following indices with the following values:

Locator Byte Index	Locator Byte Value (Hexadecimal)
2	0xF0
9	0x70
12	0x0C
25	0x3D
42	0xE8
66	0x90
74	0xF0
78	0xC7
83	0xB8
95	0xB2
119	0xF7

Table X: Locator Byte Indices and Values

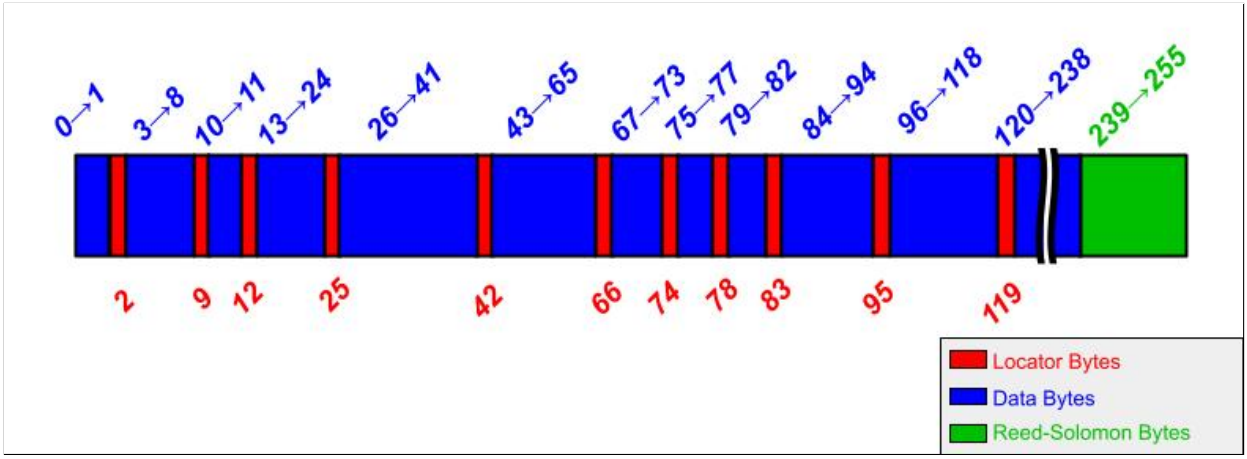

Figure X: Packet Level Diagram

1.2 Bit Level Format

The data signal is Manchester Encoded and utilizes the dedicated UART hardware of an STM32F103 microcontroller; therefore, our encoding scheme must adhere to the constraints of UART protocol. Our transmission scheme features a baud period for the start bit, 8 baud periods for the data, and a baud period for the stop bit, giving a frame a total of 10 baud periods.

Because the data is Manchester Encoded, each data bit sent takes two baud periods to be transmitted. Therefore, in order to send a single byte of data, two UART bytes are required. The diagram below shows the structure of a single data byte.

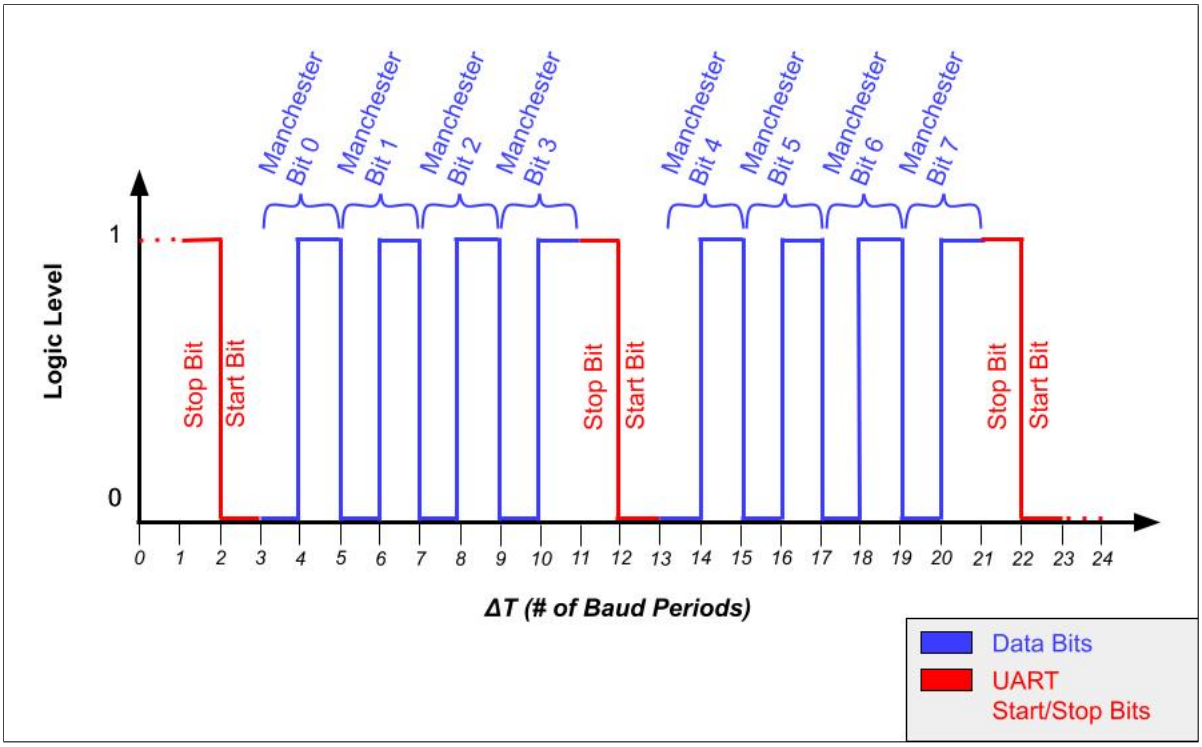

Figure X: Bit Level

1.3 Transmission Bit Rate Calculation

The bit rate of the ImpLiFi system, in this case meaning the amount of data bits transmitted as a function of baud periods elapsed, can be calculated. For this calculation, the Reed-Solomon CRC bytes, locator bytes, and UART start/stop bits are not considered data bits, and are instead considered part of the packet overhead.

Each transmitted byte taking 20 baud periods (see fig. X above), giving a bit rate of $\frac{8}{20} = 0.4$. Of our 255 total bytes in the package, 11 are locator bytes and 16 are Reed-Solomon Error Correction bytes. This gives $255 - (11 + 16) = 228$ transmitted data bytes per packet. In total, this gives a bit rate of $0.4(\frac{228}{255}) = 0.358$. This means that if our baud rate is 57.6 kbps, our total data rate would be $57600 \cdot 0.358 = 20600$ data bits per second.

2 System Overview

The FPGA receiver is based on the Digilent CMOD A7-35T that utilizes the XC7A35T-1CPG236C SoC FPGA. T

The receiver FPGA consists of the following modules:

1. XADC (Xilinx ADC IP)
2. AGC/CDR State Machine
3. UART Manchester Decoder
4. Reed-Solomon State Machine
5. MicroBlaze Soft Processor

3 XADC (Xilinx ADC IP)

The XADC is the Xilinx Vivado IP that allows sampling from the SoC's integrated ADC. In order to sample at the maximum sampling rate (1 MSPS), the XADC block must be fed a clock frequency that is a multiple of 26 MHz and between 104 and 250 MHz. For this reason, I have elected to run at 156 MHz. To bypass timing constraint errors associated with clock-domain crossing (CDC), all custom programmable logic code is also clocked at 156 MHz.

The XADC IP Block is configured to sample a single unipolar (0V to 3V3) channel in continuous mode at 1 MSPS. The analog signal is sampled through ADC channel 12. Some peripheral functionality is enabled, such as DC offset and gain calibration.

The channel 12 positive and negative auxiliary ports of the IP Block (`vauxp12` and `vauxn12`, respectively) are pinned out to the top-level of the block diagram, and assigned to the corresponding pin in the physical constraints file of the project. Input and output data to the IP Block are fed via a Dynamic Reconfigure Port (DRP), a standard common to many Vivado IP along with AXI4-Lite. Through the DRP, the 7-bit address for ADC channel 12 is specified (address `= x"1C"`). The end-of-conversion output pin, `eoc_out`, is fed back to the channel enable pin `den_en`. This restarts the ADC conversion process as soon as it ends. Furthermore, the `eoc_out` port is connected to the `input_valid_i` port of the AGC/CDR state machine, meaning the state machine process begins each time a new sample is taken.

4 AGC/CDR State Machine

The automatic gain control (AGC) and clock/data recovery (CDR) modules process each new sample through a sequential state machine. The state machine waits for the `end_of_conversion (EOC)` output pin of the XADC block to go `HIGH`, signifying a new ADC sample, before beginning. Once a new sample is received, the state machine enables the AGC module and waits for the AGC process to complete. The scaled and zero-centered waveform from the AGC is then passed to the CDR module, which is enabled to process the

most recent sample. Once completed, the recovered clock and the most recent bit is output from the state machine.

State	**Description**	**Δt (Clock Periods)**
IDLE	Waits for `input_valid` signal to go high	Variable
WRITE_AGC	Writes most recent sample to the AGC	1
WAIT_AGC	Waits for valid output from AGC	37
INIT_CDR_VALS	Single clock period delay allowing input buffer registers or CDR state machine to be written to	1
WRITE_CDR	Initializes and starts CDR state machine	1
WAIT_CDR	Waits for CDR state machine to produce valid output	-
Total:	-	**-**

4.1 AGC State Machine

The AGC module centers the sampled waveform on zero, then scales the data by a factor determined by a reference peak-to-peak value. The zero-centering is accomplished by filtering the waveform with a moving average filter to get the digital equivalent of the DC Average, then subtracting the DC average from each sample. Once the data is zero-centered, the peak of the waveform is detected with a digital approximation of a peak detector circuit. A peak reference value is then divided by the sampled peak value to get a scaling gain factor, which is multiplied by the zero-centered data.

The AGC process described above is accomplished with an sequential propagation state machine, which means that a state machine initializes each step, waits for valid output, and then moves onto the next process. Each state, its description, and the time to perform its corresponding action are outlined in the table below.

State	**Description**	**Δt (Clock Periods)**
IDLE	Waits for `input_valid` signal to go high	Variable
AGC_WAIT_ MOVING_AVERAGE	Waits for valid output from the moving average filter	3
AGC_ZERO_ CENTER_SUBTRACT	Subtracts the moving average from incoming waveform. Also, increments `peak_detector_count_s`, which is used in the peak detector's decay.	1
AGC_ZERO_ CENTER_SUBTRACT_D1	Delay the subtract operation for timing closure purposes, and for `unsigned` to `signed` type conversion.	1
AGC_ZERO_ CENTER_SUBTRACT_D2	Delay the subtract operation for timing closure purposes, and for `unsigned` to `signed` type conversion.	1
AGC_PEAK_ DETECTOR_CALC_ CONDITIONALS	Determines whether `peak_s` register needs to be updated, due to a new peak value being detected or enough time elapsing to elicit decay.	1
AGC_PEAK_ DETECTOR_WRITE_ TEMP_PEAK	Writes new peak value to `peak_temp_s`	1

AGC_PEAK_ DETECTOR_WRITE_PEAK	Writes `peak_temp_s` to `peak_s`	1
AGC_START_ DIV_GEN	Start the Divider Generator Vivado IP	1
AGC_WAIT_ DIV_GEN	Waits for Divider to finish its operation	26
AGC_MULTIPLY_ GAIN	Multiplies the register holding the zero-centered data sample by the AGC gain value. Also, toggles the `output_valid_o` output port.	1
Total:	-	**37**

Some real-time signal waveforms pertaining to the AGC system were captured with an Integrated Logic Analyser are shown below.

- `adc_data_s`: 12 bit unsigned. The sampled incoming data from the XADC. In this example, the waveform has a crest at approximately 850 and a trough at approximately 810.
- `moving_ave_s`: 12 bit unsigned. The moving average of `adc_data_s`.
- `centered_data_13_bit_s`:

Signal	Radix	Description
`adc_data_s`	12 bit unsigned	The sampled incoming data from the XADC. In this example, the waveform has a crest at approximately 850 and a trough at approximately 810.
`moving_ave_s`	12 bit unsigned	The moving average of `adc_data_s`.
`centered_data _13_bit_s`	13 bit signed	The waveform from `adc_data_s` centered around zero. Derived by subtracting `adc_data_s` - `moving_ave_s`. Bit 13 is just for unsigned to signed conversion.
`peak_s`	16 bit unsigned	The local maximum of `centered_data`, this signal decays over time.

agc_gain_s	24 bit unsigned fixed-point with 12 integer bits and 12 fractional bits	The gain value that is multiplied with centered_data_s. Derived by dividing target peak-to-peak reference (1000 in this case) by peak_s.
scaled_centered_data_s	12 bit signed	Output of AGC. Waveform has crest at approximately 1000 and trough at approximately -1000.

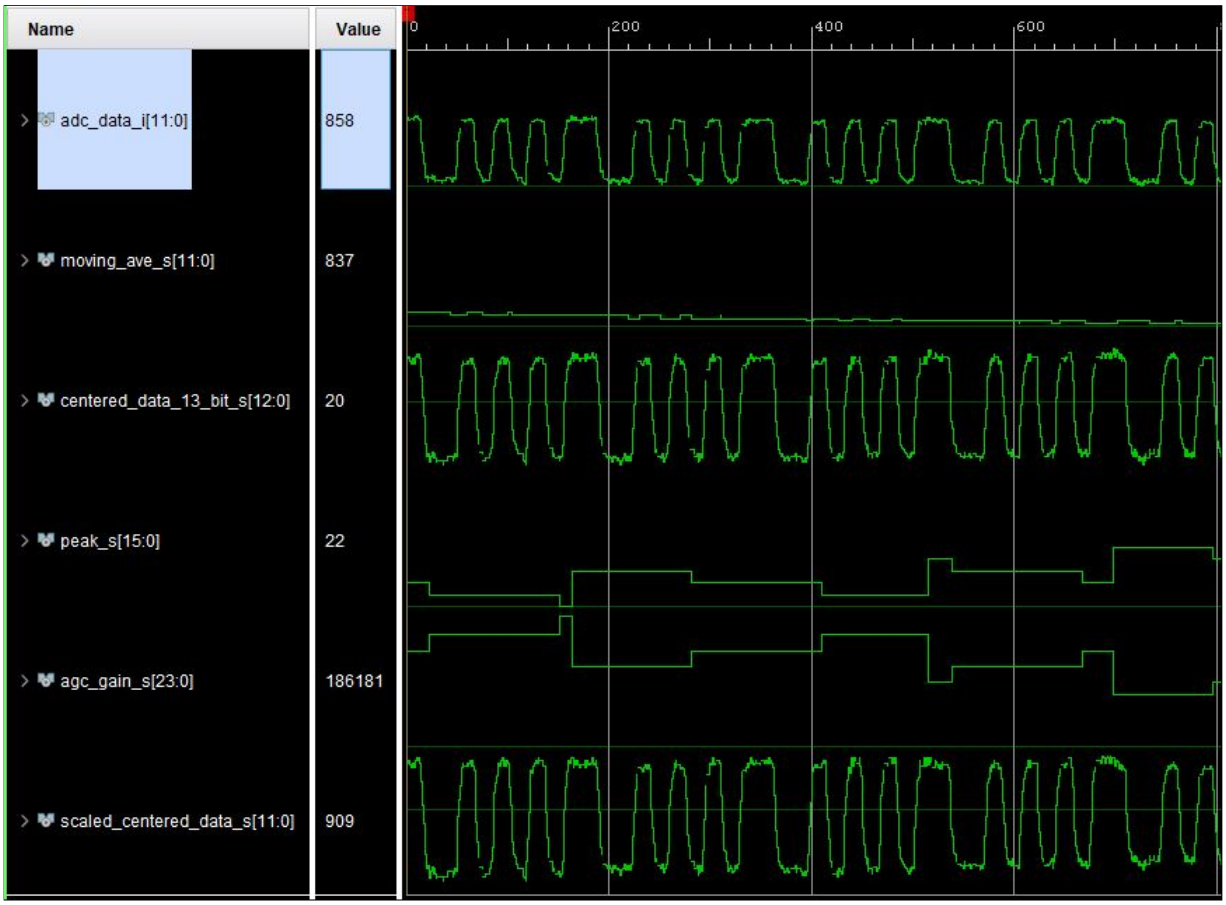

Figure X: AGC State Machine Diagram

4.2 CDR State Machine

The clock data recovery (CDR) state machine is based off a Costa's Loop topology for binary phase shift keyed (BPSK) signals. Once the ADC output waveform is centered and scaled by the AGC state machine, the waveform is split and multiplied by two sinusoidal waveforms 90° out of phase with each other to produce in-phase (I-Channel) and quadrature (Q-Channel) signals. The I-Channel and Q-Channel are multiplied together to produce a phase error signal, which is then filtered by a low-pass FIR filter and fed into a PI-Controller. The PI-controller output is added with an initial phase-step "estimate value," which is summed with an arithmetic accumulator to produce a roll-over sawtooth signal. The slope of this sawtooth signal is what is modulated by the PI-Controller, and corresponds to the recovered clock frequency. This sawtooth is fed into a Vivado Direct Digital Synthesizer (DDS) IP Core, and the resulting sine and cosine waveforms generated are XOR-ed together to produce the recovered clock signal. The sine and cosine are also the two sinusoidal waveforms 90° out of phase with each other, which are mixed with the incoming data, closing the feedback loop.

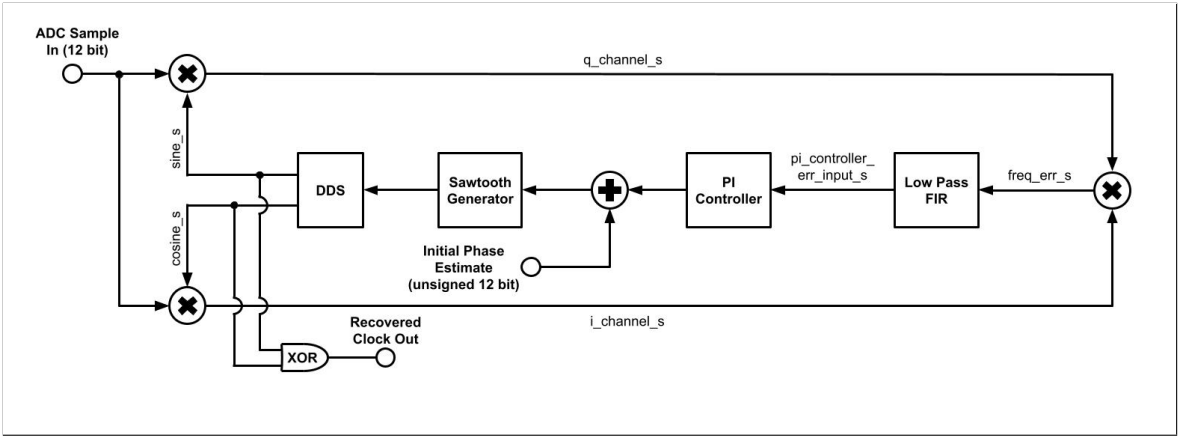

Figure X: Clock Data Recovery Modular Diagram

4.2.1 CDR Costa's Loop Arithmetic

The purpose of the CDR is to generate a sine and cosine wave with the same frequency (referred to as the "estimated frequency") which is close in frequency to the incoming data, then determine the frequency error between the estimated frequency and frequency of the incoming data. Although the sampled incoming waveform is a ideally a square wave, the clock can be recovered by regarding the waveform as a BPSK sinusoidal. Treating it as a sinusoidal, the incoming waveform will have the formula $sin(2\pi f_{in}t)$, where f_{in} is the frequency of the incoming data. The waveforms generated with the estimated frequency have the equations $sin(2\pi f_{est}t)$ and $cos(2\pi f_{est}t)$. The frequency error is derived by sections 3.2.2: Channel Mixing and 3.2.3: FIR Low Pass Filter.

I-Channel Mixing Arithmetic

As the incoming data is multiplied by a cosine wave with the estimated frequency of f_{est}, the following equation is derived and simplified.

$$sin(2\pi f_{in}t) \cdot cos(2\pi f_{est}t)$$

Substituting $e = 2\pi f_{est}t$ and $i = 2\pi f_{in}t$:

$$sin(i) \cdot cos(e)$$
$$= \frac{1}{2}[sin(i+e) + sin(i-e)]$$

Q-Channel Mixing Arithmetic

As the incoming data is multiplied by a sine wave with the estimated frequency of f_{est}, the following equation is derived and simplified.

$$sin(2\pi f_{in}t) \cdot sin(2\pi f_{est}t)$$

Substituting $e = 2\pi f_{est}t$ and $i = 2\pi f_{in}t$:

$$sin(i) \cdot sin(e)$$
$$= \frac{1}{2}[cos(i-e) - cos(i-e)]$$

I-Channel and Q-Channel Mixing Arithmetic

When the I-Channel and Q-Channel are multiplied together, the following equation is derived and simplified:

$$\frac{1}{4}[(sin(i+e) + sin(i-e)) \cdot (cos(i-e) - cos(i-e))]$$

Substituting $i + e = \alpha$ and $i - e = \beta$ and expanding:

$$\frac{1}{4}[sin(\alpha)cos(\beta) - sin(\alpha)cos(\alpha) + sin(\beta)cos(\beta) - sin(\beta)cos(\alpha)]$$

It is important to note here that the main purpose is to isolate the frequency difference, aka $\beta = i - e$. For example, if our estimated frequency is $f_{est} = 100$ kHz but our incoming data signal is arriving at $f_{in} = 101$ kHz, then $\beta = f_{in} - f_{est} = 101$ kHz -100 kHz $= 1$ kHz.

Using Angle-Sum Identity $sin(\alpha - \beta) = sin(\alpha)cos(\beta) - cos(\alpha)sin(\beta)$:

$$= \frac{1}{4}[sin(\alpha - \beta) - sin(\alpha)cos(\alpha) + sin(\beta)cos(\beta)]$$

Finally, with the Product Identity $sin(\alpha)cos(\alpha) = \frac{1}{2}[sin(\alpha+\beta)+sin(\alpha-\beta)]$, our equation simplifies to:

$$= \frac{1}{4}[sin(\alpha - \beta) - \frac{1}{2}sin(2\alpha) + \frac{1}{2}sin(2\beta)]$$

Our reduced equation can be separated into three terms:

1. $sin(\alpha - \beta)$
2. $\frac{1}{2}sin(2\alpha)$
3. $\frac{1}{2}sin(2\beta)$

In the previous example described where $f_{est} = 100$ kHz and $f_{in} = 101$ kHz, $\alpha = 201$ kHz and $\beta = 1kHz$. The first term is a sinusoidal with frequency of 100kHz, the second term is a sinusoidal with frequency of 402 kHz, and the third term is a sinusoidal with frequency of 2 kHz. As we have isolated β in the third term, an FIR Low-Pass Filter can be used to attenuate the first two terms. The isolated third term is a sinusoidal that represents the phase difference between the estimated frequency and the frequency of the actual data. This is fed into a PI-Controller, which controls the estimates' frequency and drives the frequency error to zero.

4.2.2 Channel Mixing

Mixing the incoming data with two sinusoidal waveforms 90° out of phase with each other is achieved with three Vivado Multiplier IP blocks cascaded as shown in the diagram below. The resulting signal `freq_err_s` is a signed 48-bit integer. Note that the delay blocks in the diagram below are for FPGA meta-stability purposes to close failing timing endpoints. The delays are handled by the wrapper state machine.

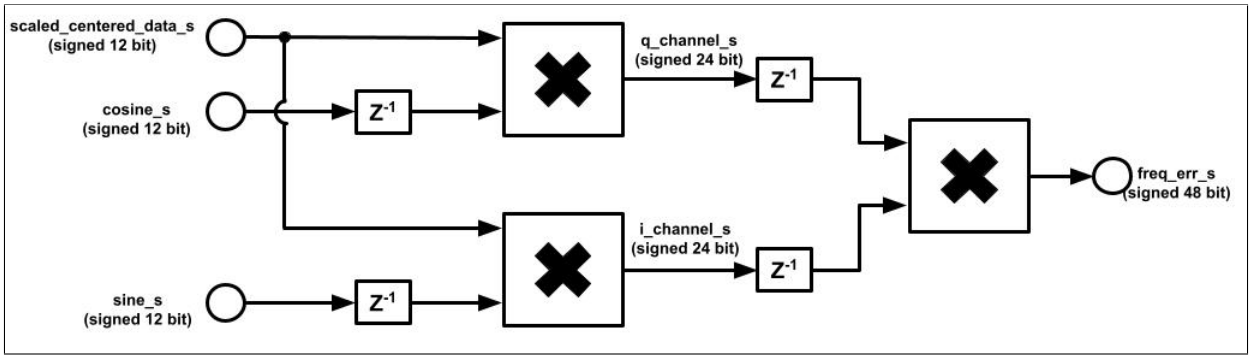

Figure X: Channel Mixing Diagram

4.2.3 FIR Low Pass Filter

In order to extract the phase difference from the `freq_err_s` signal in *Figure X*, a finite impulse response (FIR) low-pass filter (LPF). Vivado provides an FIR Compiler IP, which allows a user to specify the filter coefficients, and the IP will properly pipeline the filter and implement DSP blocks where necessary. This filter is designed as a 65-order filter with a -3dB cut-off frequency of $f_c = 9.7$ kHz, and an output that is an 40 bit signed integer. It has a sample latency of 33 cycles, after which it toggles the `m_axis_data_tvalid` output. This filter is used by writing a single sample, enabling the filter, waiting for `m_axis_data_tvalid` to go high, then disabling the filter until the next sample is ready to be processed.

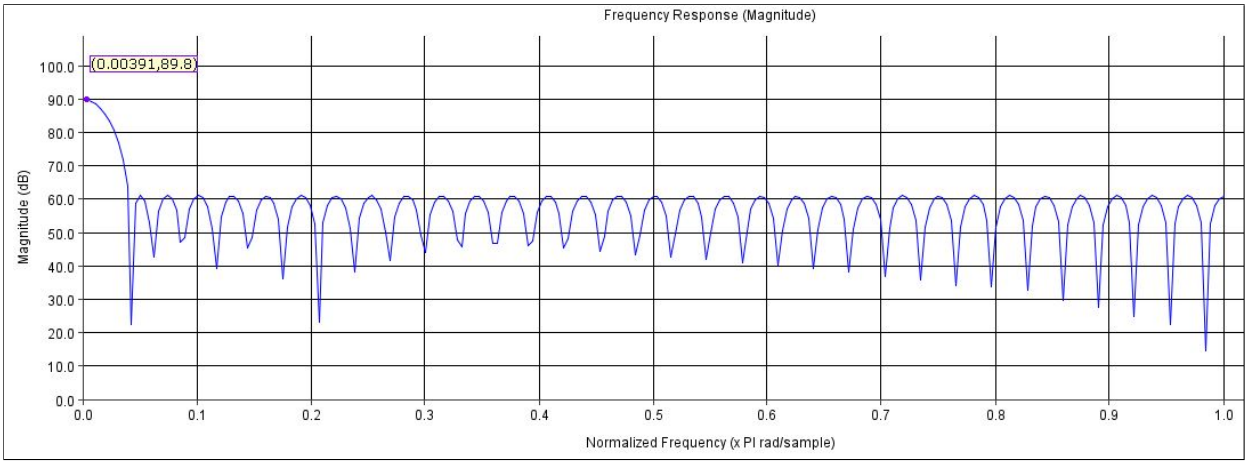

Figure X: FIR LPF Magnitude Response

4.2.4 PI Controller

The filtered phase difference signal `pi_controller_err_input_s` is then fed into a PI-Controller. The PI-Controller is accomplished in 3 clock cycles with 3 states in the state machine. The first state calculates the proportional branch by multiplying the error input by K_p, and adds the error input to an accumulator (aka integrator) for the integral branch. The second state multiplies the accumulator by K_i by the accumulator value to calculate the value of the integral branch. The final branch calculates the PI-Controller output by adding the values of the proportional and integral branches. The output is sliced into a 12 bit signed integer, which is necessary to eventually be fed into the DDS.

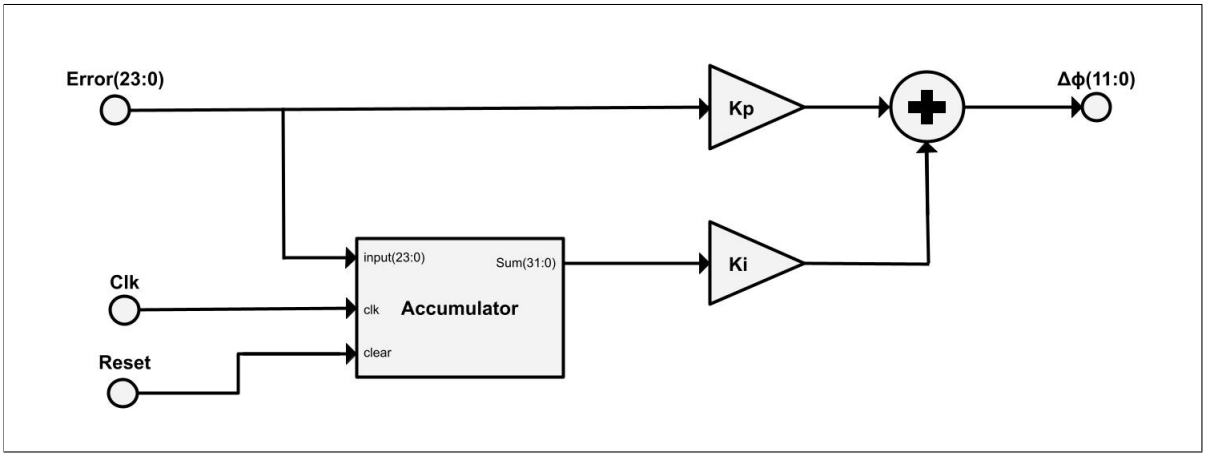

Figure X: FIR LPF Magnitude Response

4.2.5 Sawtooth Phase Generator

The purpose of the sawtooth phase generator is to control the frequency of the sine and cosine waveforms generated by the (Direct Digital Synthesizer) DDS. To achieve this, an initial frequency estimate is tuned to match the frequency of the incoming data waveform. For example, if the incoming data has a frequency of approximately 50 kHz and the sampling rate is 1 MSPS, each received bit will have $\frac{1{,}000{,}000\frac{samples}{sec}}{50{,}000\frac{bits}{sec}} = 20$ samples per bit. Because the DDS has 65536 positions around the unit circle, generating an initial frequency estimate of 50 kHz would require the `inital_phase_offset_s` signal to be $\frac{65536}{20} = 3277$. As the phase difference between the generated signal and the actual incoming data signal is fed into the PI-Controller, the phase difference is summed with the `inital_phase_offset_s` signal to get the phase of the corrected signal. This is then added to a 16 bit accumulator, designed to rollover past 65536 (meaning once per period of a received bit). The instantaneous value of the accumulator is the instantaneous phase of the recovered clock, and the rate of change of the accumulator (the number that is added to the accumulator each sample) is the frequency.

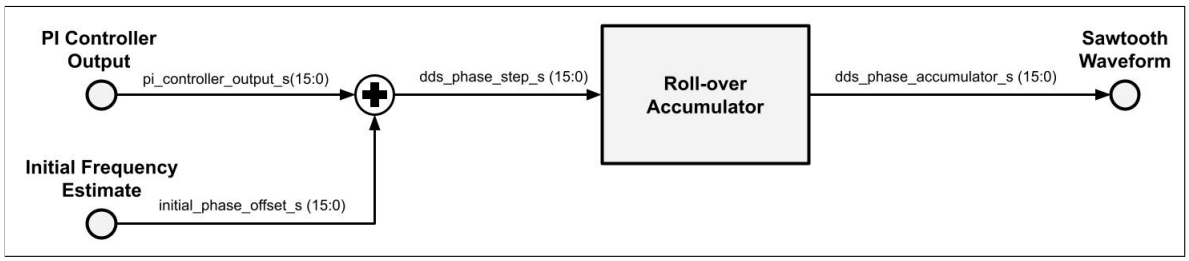

Figure X: Sawtooth Generator Modular Diagram

4.2.6 Direct Digital Synthesizer (DDS)

The DDS is a lookup table capable of simultaneously generating sine and cosine waveforms. Because the DDS accepts a 16 bit unsigned integer as a phase input, it is capable of generating $2^{16} = 65536$ positions around the unit circle. As the sawtooth waveform is fed into the DDS,

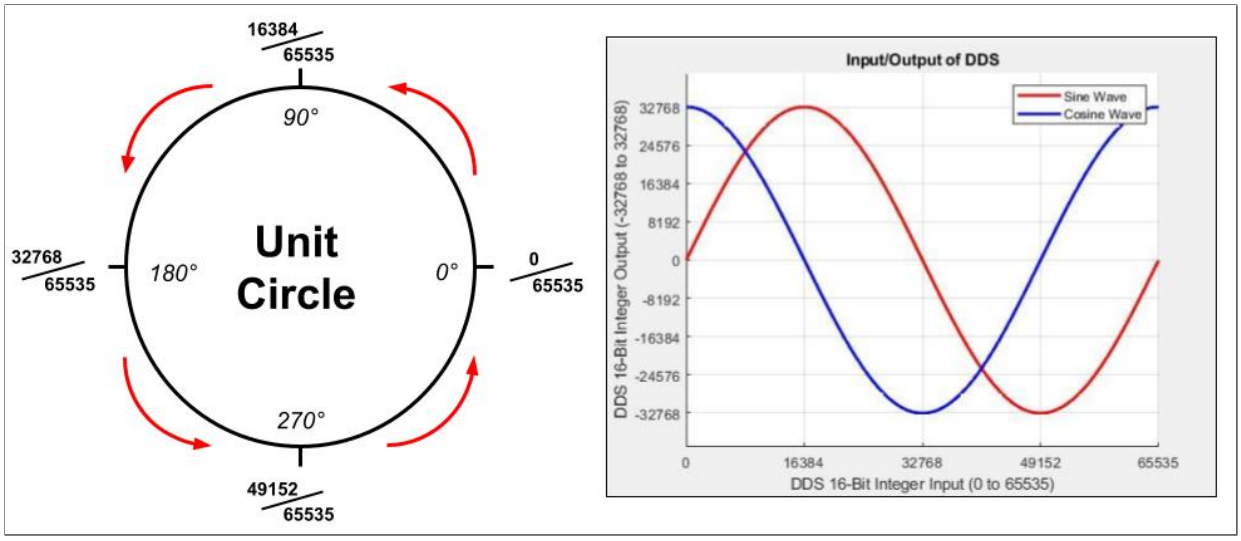

Figure X: DDS Unit Circle Representation & Input/Output Characteristics

5 Reed-Solomon State Machine

The Reed-Solomon State Machine is based on the RS Codec Core found here. The main VHDL component of the core is found in the `rs_decoder.vhd` file, with IO ports corresponding to the following diagram. The parameters of the Reed-Solomon system (which in this case are RS(255,239)) are set as generics in the VHDL file.

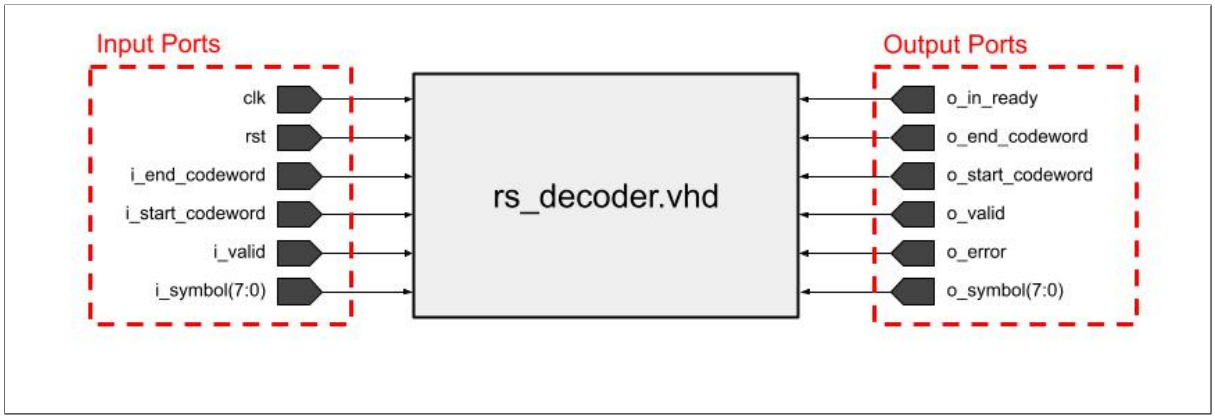

Figure X: `rs_decoder.vhd` *Block Diagram*

To begin decoding, `i_valid` is set HIGH, and `i_start_codeword` is toggled HIGH for a single clock cycle. The array of bytes to be decoded is sequentially entered into the `i_symbol` port, with one symbol (byte) per clock period. As the last input byte is fed into the decoder, `i_end_codeword` is toggled HIGH for a single clock cycle and `i_valid` is set to LOW. Then the decoder will take an unspecified amount of clock cycles depending on how many errors need to be corrected. Once the `rs_decoder` has finished correcting errors, `o_valid` is set to HIGH and `o_start_codeword` is toggled HIGH for a single clock cycle. Once this happens, `rs_decoder.vhd` will begin to output error-corrected bytes on the `o_symbol` port. After all the corrected data is output, `o_end_codeword` is toggled HIGH for a single clock cycle and `o_valid` is set LOW. If any errors are encountered throughout this process,

6 MicroBlaze Soft Processor