FPGA-based QPU interface unit for fast qubit control and readout

Module description document

Christian Križan[1,3] and Mats Tholén[2,3]

[1] Quantum Technology Laboratory, MC2, Chalmers University of Technology
`krizan@chalmers.se`

[2] Intermodulation Products AB

[3] WACQT, Wallenberg Centre for Quantum Technology

Abstract. This document provides additional specifications to all modules present in the 2020 DAT096 project 'FPGA-based QPU interface unit for fast qubit control and readout.'

1 How to read this document

The intention of this document is to specify the VHDL modules expected to be present in the final device, although this does not set a limit for including more modules into the design should your design process deem this a valid strategy. Some parts of the specification are stricter in this regard than others. The Ethernet communications module is for instance more defined as to how it should look like and act, whereas the central stream controller will look very different between the different groups implementing it. Many intra-device ports in turn also feature specifications where this is desired, such as the AXI buses.

The design purposely lacks detail where design decisions are best made by the implementer (you), meaning that there has been room left over for creative solutions. However, remember that you are not supposed to be left alone without adequate specifications to complete your task.

Requested changes to the already made specifications themselves should preferably be communicated to the product owner (`krizan@chalmers.se`). Necessary changes to the overall specification will be informed to all groups, please inform the product owner whether a change impacts some feature of your already implemented design.

The rest of this section will outline details for the HDL modules present in the map, as well as the dataflow overview. *Do note that all specified port names are suggestions.*

2 Ethernet communications module

Device-to-host communication will be achieved via a custom Ethernet communications module, specifically designed to handle streaming from and to a host device. In the interest of rapid transmission, the specification has been limited as to what package type is expected when receiving a data stream packet. It is **adamant** that this Ethernet interface should be UDP-based and not be host-CPU dependent. The interface should preferably be able to handle corrupt and/or lost frames. Frames not containing UDP content should not simply be thrown away, but instead sent onwards to the other_data_tx bus as will be repeated shortly.

Incoming relevant UDP packets containing data for the central stream controller will be addressed in the port range 30000 to 30255. The port numbers will correspond to what FIFO is targeted in the central stream controller. Everything else should be considered as 'other data,' for instance control words used to set various modules in your system (for instance, the upconversion NCO frequency). You are allowed to assume that all data received in this port range can be considered as *valid* stream data.

The last paragraph has an additional implication: a given output DAC channel is targeted by writing a sample to the FIFO given by the received port number. Ie. writing a data sample to the very first FIFO (data stemmed from port 30000) will put this sample into the very first output register in the DAC.

Control words are received on port 29999; these differ syntactically to the datastream words, and are given in detail in subsection 5.3.

With the particular development board in mind, you have been requested to coform to an MII (Media-independent interface). More specifically, RMII (Reduced media-independent interface). Documentation relating to this interface, such as signals and clocking requirements, is broadly available online. You are not expected to implement every detail of a full standard into your module - your goal should be to strive for a sleek solution.

The connection of other_data_rx to the UDP packet stream uplink block is a suggestion, it might be more useful to simply send whatever data is present on this line straight onto the Ethernet frame transmitter. Although, it is highly recommended to stick to UDP on the uplink as well.

The Ethernet communications module is expected to feature ARP functionality. This implies that the Ethernet frame receiver / transmitter is expected to forward ARP requests to the ARP resolver and back. The reason for including ARP is that assigning MAC addresses to static IP's in some operating systems require elevated access rights for the user.

The UDP packet streaming blocks will send data onwards onto the central stream controller using an AXI bus, this bus should utilise *at least* the AXI signals TDATA, TVALID, TLAST and TREADY.

Stream package content information is available in section 5.1.

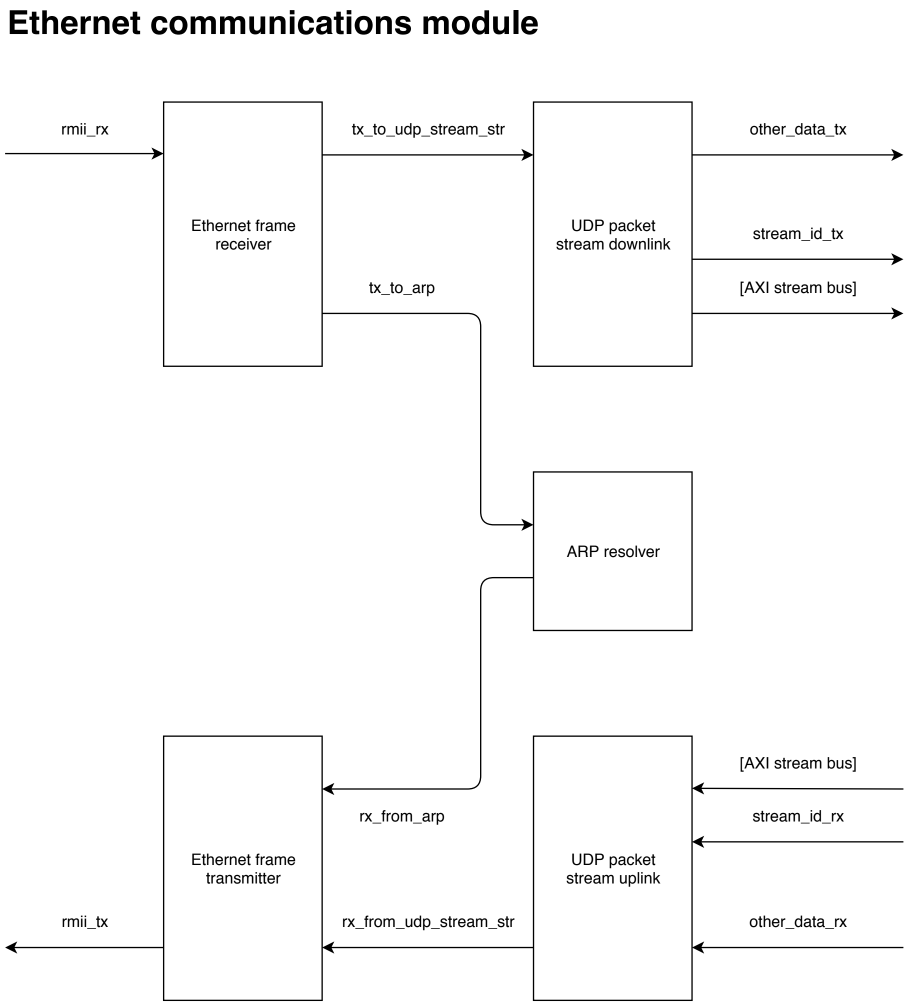

Fig. 1. An overall view of the Ethernet communications module, available as a separate graphic.

3 Central stream controller

Managing the high data rates expected in this device is best handled by a central stream controller. The stream controller will have to store large amounts of data for efficient stream handling. Data going to pretty much whatever destination can arrive at any time, meaning that the data rate going in is highly inconsistent. Keep in mind that this does not change the sampling rate of the data contained within the packet going in - or out for that matter.

For this datastream management, it is recommended that you utilise the capabilities of the Nexys 4 board. A suggestion is to investigate the capabilities of the included BRAM, as well as the on-PCB DDR memory. This module and its implementation is largely uncharted territory, its implementation will depend greatly on your findings.

The officially recommended method of implementing the Nexys 4 DDR is via the Memory Interface Generator (MIG). Tutorials and settings required are available online. Moreover, the UCF file containing the DDR mapping is available at Digilent's website. This file cuts down a lot on the complexity when using the MIG. You are not recommended to implement your own, custom DDR controller for this project. Do note that the interface generated by the MIG is likely an advanced superset of the AXI bus, which is far more complicated than what is recommended here for the datastream signal path. This will pose another engineering trade-off for you to explore.

A requested feature is that the operator should be able to set and change windowing on the received waveform on-the-fly. This is likely best solved in the downconversion stage, using the skip/store block. Whether you wish to implement windowing in the central stream controller is of course up to you depending on your implementation.

The up- and downconversion lanes will typically feature many stages. These will all separately be interfaced to via AXI buses. It is recommended to include a 'FIFO_not_full' signal, in order to indicate to the datastream manager that the FIFO may still be written to. The 'FIFO_not_full' signal in the upconversion means that the FIFO can now be written to by the central stream controller, ie. one slice of the DRAM buffer can be extracted and put in the requesting FIFO. Your lead goal should always be that all FIFOs are kept as full as possible, effectively placing a specified constraint in the dataflow management of your system.

Writing to an already full FIFO should by your design not cause a package drop, the handling of this case is best done as per your specific implementation. One recommendation is, as mentioned, to use a DRAM memory to your advantage where so is needed.

Reading from the downconversion stage FIFOs are best done when the 'FIFO_not_full' signal goes low. Meaning that one slice can now safely be extracted and put in the stream control memory.

General hint: the FIFOs including the not-full-signal and their behaviour, were specified the way they were due to more causes than those inferred by this document...

Central stream controller

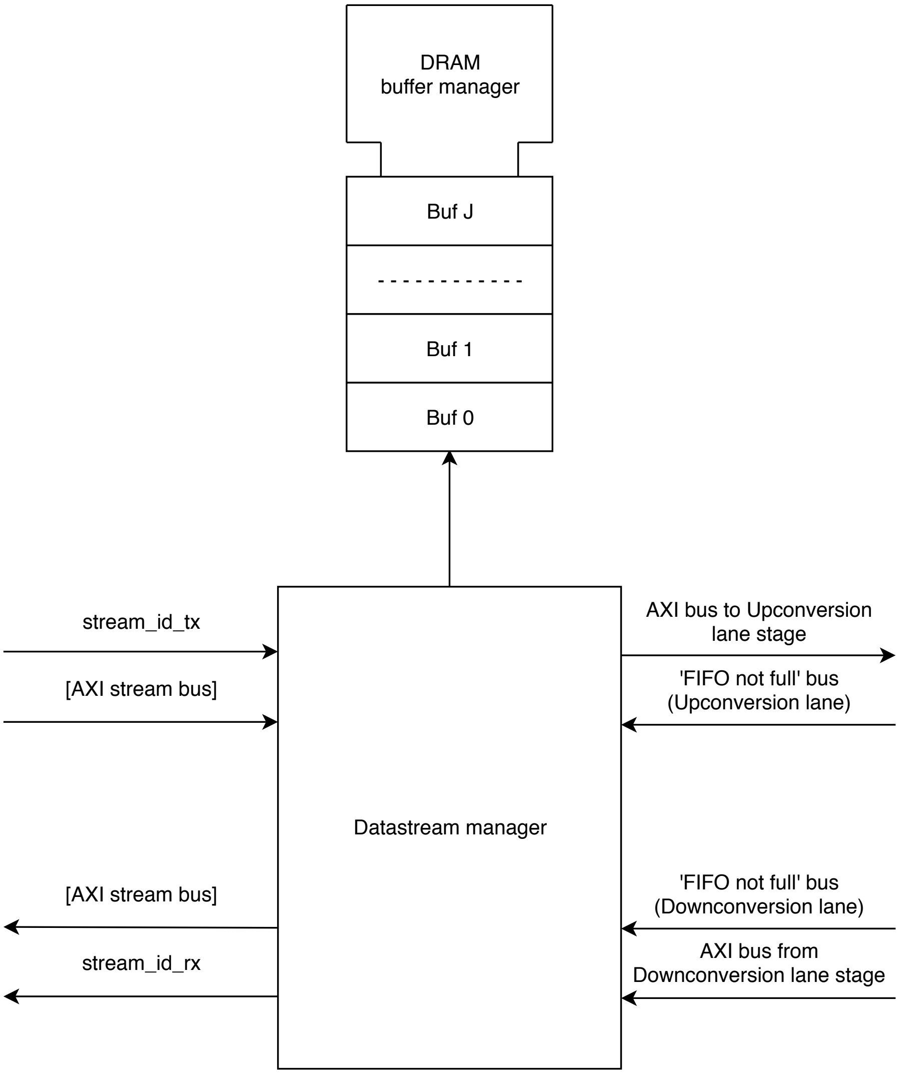
DRAM
buffer manager
Buf J
- - - - - - - - - - - -
Buf 1
Buf 0
stream_id_tx
[AXI stream bus]
Datastream manager
[AXI stream bus]
stream_id_rx
AXI bus to Upconversion lane stage
'FIFO not full' bus (Upconversion lane)
'FIFO not full' bus (Downconversion lane)
AXI bus from Downconversion lane stage

Fig. 2. An overall view of the central stream controller module, available as a separate graphic.

4 Up- and downconversion lanes

During normal operation, the stream controller will have sent some amount of data to the lane stage's initial FIFO. This FIFO is expected to output a data sample at some steady clock rate. The intended final data output from the DAC is 4 GSa/s, which is unreasonable on the Nexys 4 board. A currently very vague preliminary decision has been taken that the FIFO should output as much data as possible per time. Do note that the FIFO output clock will have to be re-adjustable, typically using generics, on the target FPGA.

A very important detail regarding the up- and downconversion as to how the GSa-output is established: the central clock on the target FPGA is 500 MHz. To output 4 GSa/s, the DAC has eight parallel registers which it reads from at a much higher pace than the central clock, one at a time. For every strike of the main system clock, you are expected to have prepared and set the next eight samples in the DAC channel registries. Meaning that you will have to run eight output channels in parallel in order to keep up with the final DAC output pace of 4 GSa/s. See figure 4 for further reference.

The samples in the upconversion lane will be run length encoded (RLE), the format of which will likely be decided for you in order to provide sufficient test vectors. The reason for including RLE is to not send 150 packets of zeroes into the device just because you can, but to easen the traffic congestion in the central stream controller. It then makes more sense to simply send one control signal telling something (hint: the RLE block) to send 150 words of zeroes onto the upconversion stage the next 150 strikes of the main system clock.

Once the sample has been decoded by the RLE, it will consist of two 16-bit words describing the I and Q components of some signal's sample sent by the operator.[4] This microwave pulse will as expected carry some frequency content. Before sending this pulse to a qubit, it has to undergo frequency upconversion since the differences are typically in the order of kHz versus GHz. The upconversion is expected to utilise IQ mixing, as we do not want to taint the quantum system with unwanted sideband spurs if these can be avoided. Analysis of received signal sidebands is a key part of investigating the returned QPU readout signal, thus we should not infer that our QPU does / does not carry particular characteristics.

Do note that IQ mixing brings with it various challenges in sampling and filtering. Your design might just as likely stem from an academic paper as it

[4] Example clarifying this previous paragraph: imagine a three sample long operator-specified signal waveform with samples W1, W2 and W3. On the host-PC, these samples will be split into I and Q components, resulting in two vectors I1 I2 I3 and Q1 Q2 Q3. These will be combined to constitute three data stream package *payloads*, I1Q1, I2Q2, I3Q3. Thus, packages must be split in order to do your IQ mixing.

might stem from a book on DSP design.

To add to that, this frequency should be changeable without having to recompile the FPGA. Typically this implies that your NCO block, the module generating the sinusoids required for the IQ mixer, is settable. To clarify: the operator should be able to specify that the NCO generates sinusoids with some frequency F. In turn, this setting will steam from some host block that you configure using the other_data_tx signal, see the previous section describing the Ethernet communications module.

The 'Add N' module is a sum block. The operator should be able to specify that an arbitrary amount of channels H through G should be combined sample-by-sample and put on this channel output. This allows for clever tricks in quantum experiment design. The summation is done by element-wise summation, including adding negative elements (subtracting). The result of the summation will be sent to the first channel, aka. channel H. All other channels from H+1 through G should go dormant, we are not interested in processing these further.

The target waveform played from a channel on the upconversion DAC will typically consist of some cosine or gaussian envelopes encapsulating some carrier frequency. These waveforms can very easily be faked at will, please contact the product owner (`krizan@chalmers.se`) in case you wish to procure typical qubit control pulse data for testing purposes. Dissecting such a waveform in Matlab could for instance be useful to determine whether your mixer is operating as expected.

The downconversion stage is very similar in function to a reversed upconversion stage. However, the RLE has been replaced with a skip / store block. This can be used to your advantage when considering how you wish to implement windowing on the received waveform, or to manage stream control at an early stage. The skip/store block will be your primary tool in applying basic windowing to the received waveform. The operator should be able to define that some interval in the received waveform should be kept, while throwing away all other data samples. In a sense, this is a windowing operation with an ideally square pulse, and it is conceivable that denoting this as a windowing block might be slightly deceptive.

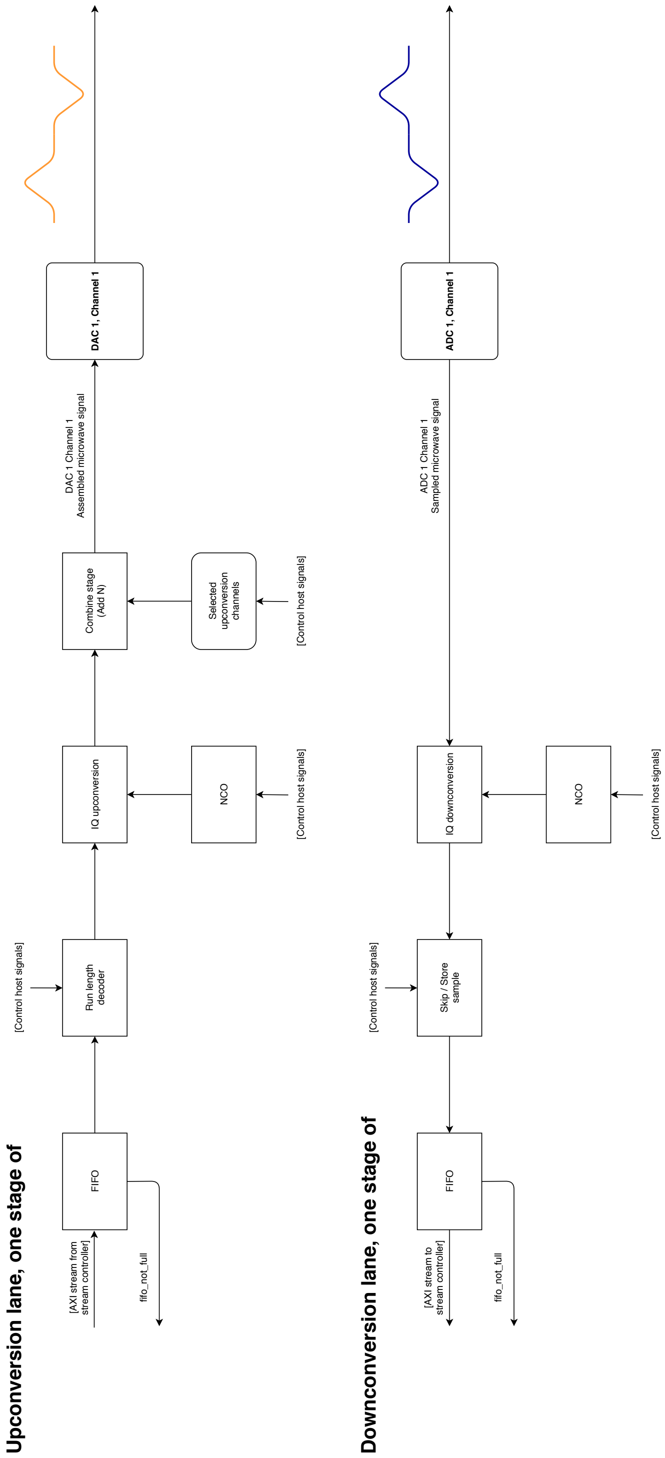

Fig. 3. An overall view of one declared stage of the up- and downconversion lane. This image is available as a separate graphic.

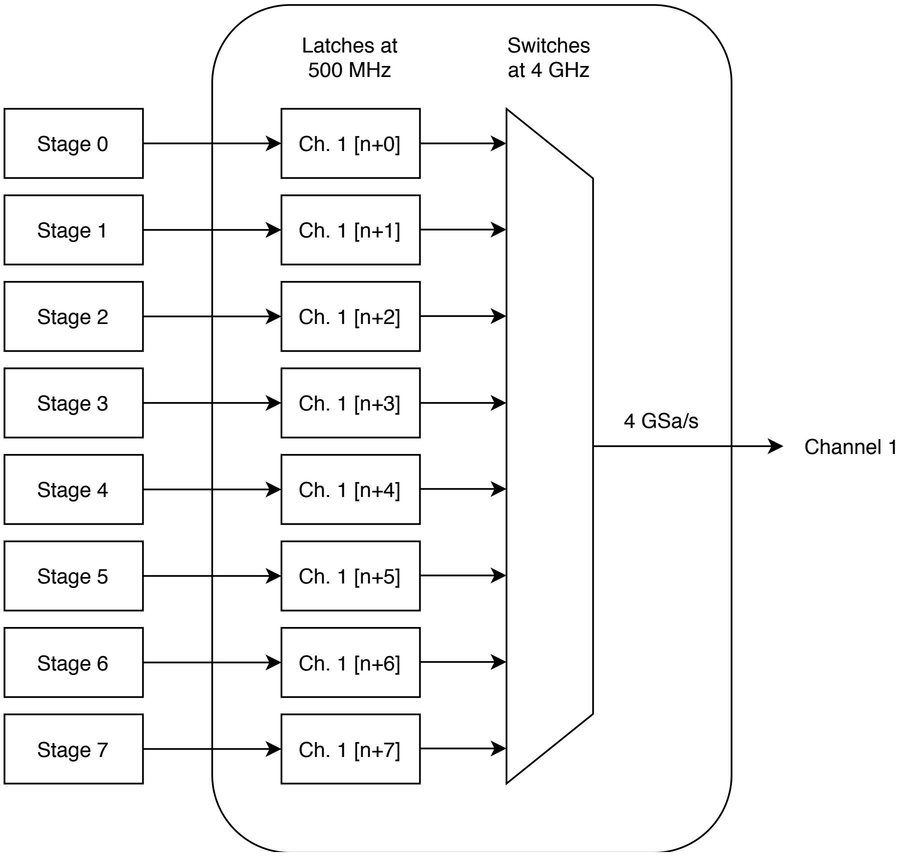

Fig. 4. Illustration of how eight parallel stages are clocked at 500 MHz by the target FPGA DAC, although output at a rate of 4 GSa/s. This is achieved by reading the content of the DAC input registers at a higher pace than what is latched at said register inputs.

5 Datastream

This section outlines a further description of the data sent to/from the PC host.

5.1 Stream package content

The data stream packages are defined using a custom format.

Bytes (LITTLE ENDIAN)	**Content (Starting at 0)**
0 - 3	Packet sequence number
4 - 7	Address of first stream data in packet
[start of A0]	
8 - 11	Address of last stream data in packet
12 - 13	Data at A0
14 - 15	Data at A0 + 2
...	

Do note, one set of 'Data' comprises 2 bytes of a signal sample's I-component, followed by 2 bytes of its Q-component.

The packet sequence number is a running 16-bit number starting at 16b0. Packet 16b0 is thus the very first packet generated by the test signal generator, and should thus be the first packet expected to be received. Its main purpose is to detect dropped packages; how this is handled further down the signal path is up to you. The 16b0 value is expected to overflow after packet $2^{16} - 1$ has been sent, and simply go back to 16b0 again.

The signal data is represented as NBCD, using a 2's complement-signed fixed point number format, 16 bits in total including the sign. This is what was previously used in for instance DAT093. The unit is 'full scale;' the operator sets a range voltage for the output DAC, the output voltage from the device is 'data value' $\cdot$ range (so, at range setting ±3 V, output range ϵ [-3 V, 2.9999 V]).

5.2 Control and readout pulses

When designing your filters, you might be interested in knowing the useful bandwidth of a typical control pulse.

Interestingly enough, your experiments will run on some of the best qubits on planet Earth; the interesting signal spectrum will fall within merely 200 Hz spectral bandwidth (the rest of the band being noise). In the ideal case, the qubits would return infinitely thin Dirac-pulses when viewed in the frequency plane such as when running an FFT. However, no filter is perfect, but your target platform's qubits are pretty close.

In a more normal qubit spectroscopy, a scoped qubit resonator's spike in the frequency spectrum will be closer to 20 MHz wide. Keep this in mind, as it is unknown as of yet what qubits might be unavailable when you reach your final verification stage.

Control pulses for the QPU encoded in the input data are ideally as thin as possible spectrum-wise. Remember that control and readout pulses normally contain some frequency content from the signal envelope, such as a (near) DC signal from a flat readout pulse, while also containing the carrier frequency.

5.3 RLE and word syntax for the control host

The syntax for the control word host has been chosen to be the following.

Packages received on the other_data_tx bus on port 29999 is considered a control word. This data may be considered valid for now, although a realistic IRL implementation of this spec. will likely add an authenticity bitfield, in order not to set various functionality within the interface when port 29999 is polled by some rogue process. This is beyond the scope of this project.

Control word	Function	Description
00000	Control word error	Is returned to the host PC when a control word was misunderstood. Similar to an exception.
10000	Reset FIFO	
01000	Blast stream controller memory	
11000	(Not used)	
00100	Force NCO oscillator reset	
10100	Set lane stage I-phase	
01100	Set lane stage Q-phase	
11100	Set NCO oscillation frequency	
00010	(Not used)	
10010	Set lane stage I-scaling	
01010	Set lane stage Q-scaling	
11010	Set RLE	Run length decoding, see [5.3]
00110	Add stage to combiner	Where 0 is FIFO 0 (port 30000)
10110	(Not used)	
01110	(Not used)	
11110	Set initial sample skip amount	
00001	Local echo	Return the received UDP datagram to the host PC.
10001	(Not used)	
01001	(Not used)	
11001	(Not used)	
00101	(Not used)	
10101	Get NCO I-phase offset	
01101	Get NCO Q-phase offset	
11101	Get NCO oscillation frequency	
00011	(Not used)	
10011	Get lane stage I-scaling	
01011	Get lane stage Q-scaling	
11011	Get RLE	Run length decoding, see [5.3]
00111	Remove stage from combiner	Where 0 is FIFO 0 (port 30000)
10111	(Not used)	
01111	(Not used)	
11111	Get initial sample skip amount	

Run length encoding syntax

Table 1. Work in progress