2009 CASPER Workshop

Tutorial 2: 10GbE Interface

Author: Jason Manley and Andrew Martens

Expected completion time: 2hrs

SKA SOUTH AFRICA
SQUARE KILOMETRE ARRAY

Contents:

1 Introduction

In this tutorial, you will create a simple Simulink design which uses the ROACH's 10GbE ports to send data at high speeds to another port. This could just as easily be another ROACH board or a computer with a 10GbE network interface card. In addition, we will learn to control the design remotely, using a supplied Python library for KATCP.

2 Background

ROACH boards have four CX-4 ports. There are two 156.25MHz crystals on the board. Each one clocks two ports, 0 & 1 and 2 & 3. This clock is then multiplied up on the FPGA by a factor of 20. Each port has 4 channels running in parallel (hence the digit in CX-4). Thus, the speed on the wire is actually 4 x 156.25MHz x 20 = 12.5Gbps.

However, 10GbE uses 8/10 encoding, which means that for every byte sent, 10 bits are actually transmitted. This is to ensure proper clocking, since the receiver recovers and locks-on to the transmitter's clock and requires edges in the data. Imagine transmitting a string of 0xFF or 0b11111111... which would otherwise generate a DC level on the line, now an extra two bits are introduced which includes a zero bit which the receiver can use to recover the clock and byte endings. See http://en.wikipedia.org/wiki/8b/10b_encoding for more information.

For this reason, we actually get 12.5Gbps * 8/10 = 10Gbps usable datarate.

CASPER's 10GbE Simulink core sends and receives UDP over IPv4 packets. These IP packets are wrapped in Ethernet frames.

Each Ethernet frame requires a 38 byte header, IPv4 requires another 20 bytes and UDP a further 16. So, for each packet of data you send, you will incur a cost of at least 74 bytes. I say at least, because the core will zero-pad some headers to be on a 64-bit boundary. You will thus never achieve 10Gbps of usable throughput. The best we have managed without loss is ~9.5Gbps. It pays to send larger packets if you are trying to get higher throughputs.

The maximum payload length of the CASPER 10GbE_v2 core is 8192 bytes (implemented in BRAM) plus another 512 (implemented in distributed RAM) which is useful for an application header.

These ports (and hence part of the10 GbE cores) run at 156.25MHz, while the interface to your design runs at the FPGA clock rate (*sys_clk*, *adcX_clk* etc). The interface is asynchronous, and buffers are required at the clock boundary. For this reason, even if you send data between two ROACH boards which are running off the same hard-wired clock, there will be jitter in the data.

A second consideration is how often you clock values into the core when you try to send data. If your FPGA is running faster than the core, and you try and clock data in on every clock cycle, the buffers will eventually overflow. Likewise for receiving, if you send too much data to a board and cannot clock it out of the receive buffer fast enough, the receive buffers will overflow and you will lose data.

In our design, we are clocking the FPGA at 200MHz, with the cores running at 156.25MHz. We will thus not be able to clock data into the TX buffer continuously for very long before it overflows. If this doesn't make much sense to you now, don't worry, it will become clear after you've tried it.

<<<< DIAGRAMS OF BUFFERS illustrating different input/output rates would ease readability>>>>

3 Setup

The lab at the workshop is pre-configured with the CASPER libraries, Matlab and Xilinx tools. Simply double-click the mlib_devel_10_1 icon on the desktop to start Matlab with all required libraries.

4 Creating Your Design

In this tutorial, a counter will be transmitted through one CX-4 port and back into another. This will allow a test of the communications link in terms of performance and reliability. This test can be used to test the link between boards and the effect of different cable lengths on communications quality.

4.1 Create a new model

Start Matlab and open Simulink (either by typing '`simulink'` on the Matlab command line, or by clicking on the Simulink icon in the taskbar). Create a new model and add the*Xilinx System Generator* and *XSG core config* blocks as before in Tutorial 1.

Specify *sys_clk_2x* (200MHz clock derived from 100MHz onboard clock) as the clock source in the *XSG core config* block.

4.2 Add reset logic

A very important piece of logic to consider when designing your system is how, when and what happens during reset. In this example we shall control our resets via a software register. We shall have two independent resets, one for the 10Ge cores which shall be used initially, and one to reset the user logic which may be used more often to restart the user part of the system. Construct reset circuitry as shown below.

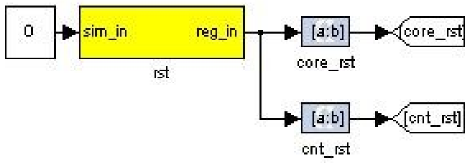


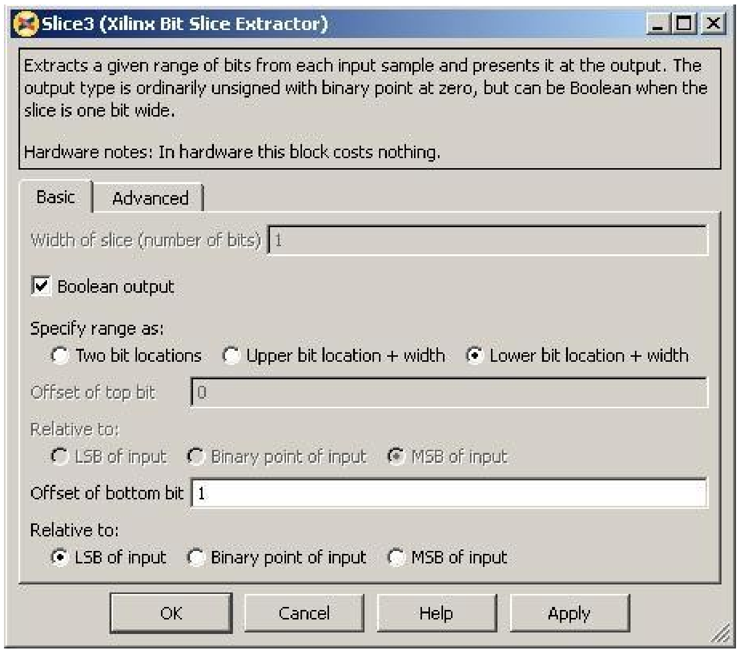
Illustration 1: Reset logic

4.2.1 Add a software register

Use a software register yellow block from the *BEE_XPS System Blockset* for the rst block. Rename it to rst. Configure the *I/O direction* to be *From Processor.* Attach a *Constant* block from the *Simulink->Sources* section of the Simulink Library Browser to the input of the software register and make the value 0.

4.2.2 Add *Slice* blocks

Add two *Slice* blocks from the *Xilinx Blockset.* Configure the one to be used to reset the 10Ge core to use the 2nd least significant bit as shown below.


Slice3 (Xilinx Bit Slice Extractor)

Extracts a given range of bits from each input sample and presents it at the output. The output type is ordinarily unsigned with binary point at zero, but can be Boolean when the slice is one bit wide.

Hardware notes: In hardware this block costs nothing.

Basic | Advanced

Width of slice (number of bits): 1

[✓] Boolean output

Specify range as: Two bit locations / Upper bit location + width / [✓ Lower bit location + width]

Offset of top bit: 0

Relative to: LSB of input / Binary point of input / [✓ MSB of input]

Offset of bottom bit: 1

Relative to: [✓ LSB of input] / Binary point of input / MSB of input

Illustration 2: core_rst setup

The *Slice* block to use the least significant bit is configured in a similar way except that the *Offset of bottom bit* should be 0.

4.2.3 Add *Goto* blocks

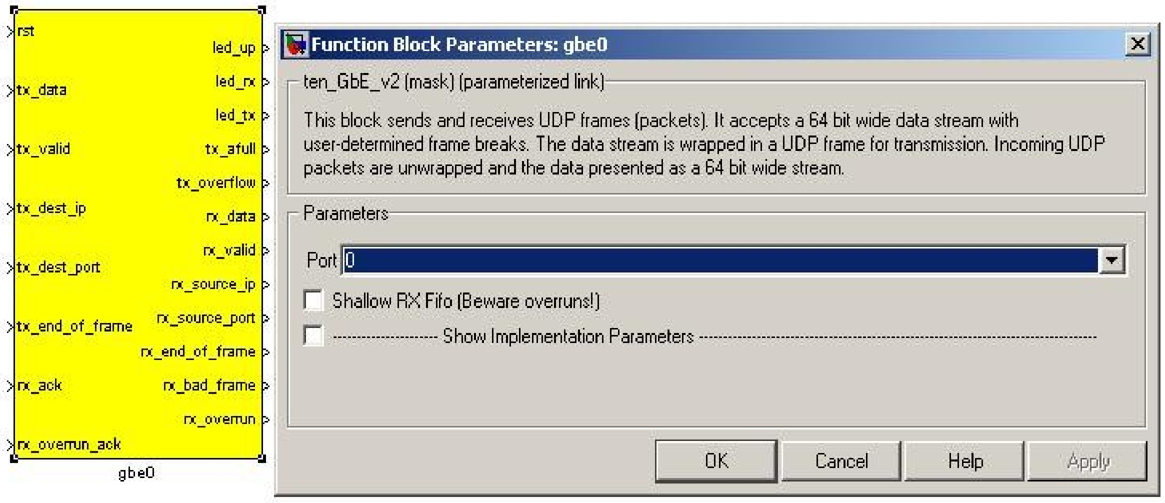
Add two *Goto* blocks from *Simulink->Signal Routing*. Configure them to have the tags as shown (*core_rst* and *cnt_rst*). These tags will be used by associated *From* (also found in *Simulink->Signal* Routing) blocks in other parts of the design. These help to reduce clutter in your design and are useful for control signals that are routed to many destinations. They should not be used a lot for data signals as it reduces the ease with which data flow can be seen through the system.

4.3 Add 10Ge and associated registers for data transmission

We will now add the 10Ge block to transmit a counter at a programmable rate.

4.3.1 Add a 10Ge block for data transmission

Add a *ten_GbE_V2* yellow block from the *BEE_XPS System Blockset.* It will be used to transmit data and we shall add another later to receive data. Rename it gbe0. Double click on the block to configure it and set it to be associated with CX-4 port 0. If your application can guarantee that it will be able to use received data straight away (as our application can), shallow receive buffers can be used to save resources. This optimisation is not necessary in this case as we will use a small fraction of resources in the FPGA.



ten_GbE_v2 (mask) (parameterized link)

This block sends and receives UDP frames (packets). It accepts a 64 bit wide data stream with user-determined frame breaks. The data stream is wrapped in a UDP frame for transmission. Incoming UDP packets are unwrapped and the data presented as a 64 bit wide stream.

Parameters

Port 0

☐ Shallow RX Fifo (Beware overruns!)

☐ ------------------- Show Implementation Parameters -------------------

Illustration 3: 10Ge transmit block configuration

4.3.2 Add registers to provide the target IP address and port number

Add two yellow-block software registers to provide the destination IP address and port number for transmission with the data. Name one dest_ip and the other dest_port. The registers should be configured to receive their values from the processor. Connect them to the appropriate inputs of the gbe0 10Ge block as shown. A *Slice* block is required to use the lower 16 bits of data from the dest_port register. *Constant* blocks from *Simulink->Sources* with 0 values are attached to the simulation inputs of the software registers. The destination port and IP address are not important in this system as it is a loopback example.Add a *From* block from *Simulink->Signal Routing* and set the tag to use *core_rst,* this enables one to reset the block.

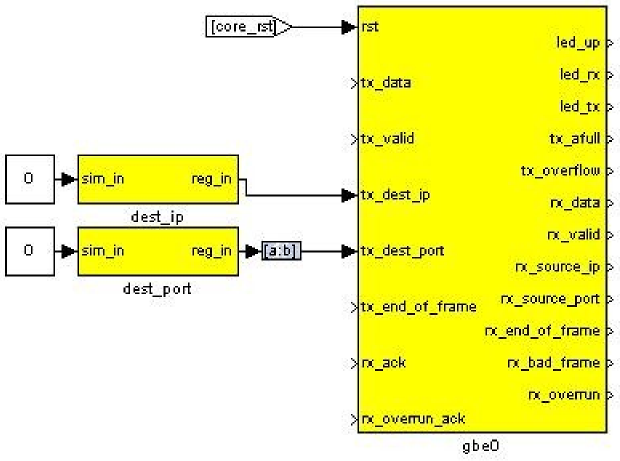


Illustration 4: 10Ge transmission block and registers

4.4 Create a subsystem to generate a counter to transmit as data

We will now implement logic to generate a counter to transmit as data.

4.5 Construct a subsystem for data generation logic

It is often useful to group related functionality and hide the details. This reduces drawing space and complexity of the logic on the screen, making it easier to understand what is happening. Simulink allows the creation of *Subsystems* to accomplish this.

These can be copied to places where the same functionality is required or even placed in a library for use in other projects and by other people.

To create a subsystem, one can highlight the logical elements to be encapsulated, then right-click and choose *Create Subsystem* from the list of options. You can also simply add a *Subsystem* block from *Simulink->Ports & Subsystems.*

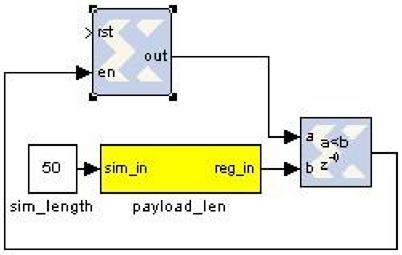
Subsystems inherit variables from their parent system. Simulink allows one to create a variable whose scope is only a particular subsystem. To do this, right-click on a subsystem and choose the *Create Mask* option. The mask created for that particular subsystem allows one to add parameters that appear when you double-click on the icon associated with the subsystem.

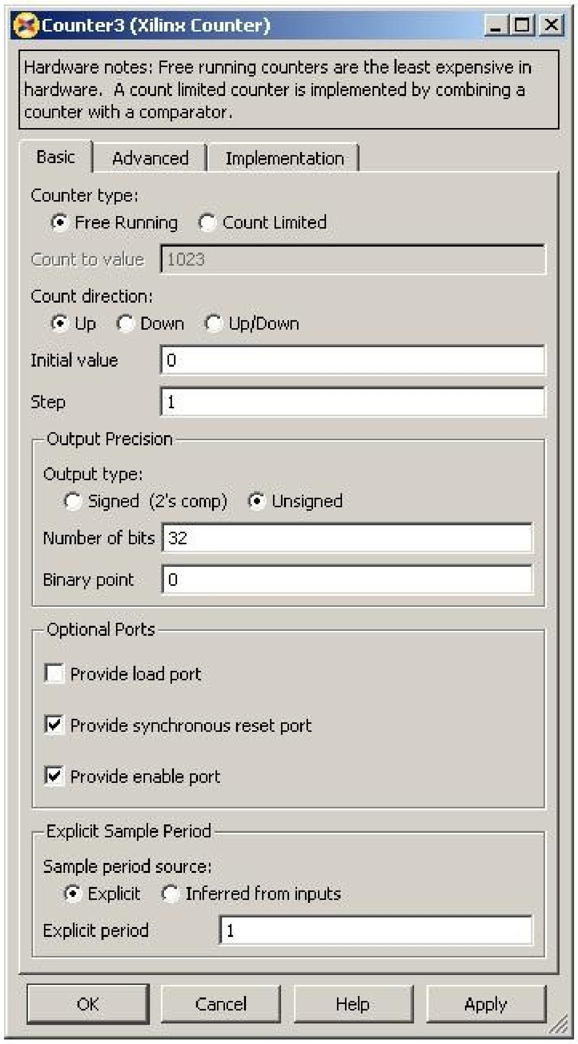
The mask also allows you to associate an initialisation script with a particular subsystem. This script is called every time a mask parameter is modified and the *Apply* button clicked. It is especially useful if the internal structure of a subsystem must change based on mask parameters. Most of the interesting blocks in the CASPER library use these initialisation scripts.

Drop a subsystem block into your design and rename it *pkt_sim.* Then double-click on it to add logic.

4.6 Add a counter to generate a certain amount of data

Add a *Counter* block from *Xilinx Blockset->Basic Elements* and configure it to be unsigned, free-running, 32-bits, incrementing by 1 as shown. Add a *Logical* block, *software register* and *Constant* block as shown. In simulation this circuit will generate a counter from 0 to 49 and then stop counting. This will allow us to generate 50 data elements before stopping.





Counter3 (Xilinx Counter)

Hardware notes: Free running counters are the least expensive in hardware. A count limited counter is implemented by combining a counter with a comparator.

Basic | Advanced | Implementation

Counter type: [✓ Free Running] Count Limited

Count to value: 1023

Count direction: [✓ Up] Down Up/Down

Initial value: 0

Step: 1

Output Precision

Output type: Signed (2's comp) [✓ Unsigned]

Number of bits: 32

Binary point: 0

Optional Ports

Provide load port

[✓] Provide synchronous reset port

[✓] Provide enable port

Explicit Sample Period

Sample period source: [✓ Explicit] Inferred from inputs

Explicit period: 1

OK | Cancel | Help | Apply

Illustration 5: Data count configuration

4.7 Add a counter to limit the data rate

As mentioned earlier in this tutorial, it is impossible to supply data to the 10Ge transmission block at the full clock rate. This would mean transmitting a 64-bit word at 200MHz, and the 10Ge standard only supports up to 156.25MHz data transmission. We thus want to generate data in bursts such that the transmission FIFOs do not overflow. We thus add circuitry to limit the data rate as shown below. The logic that we have added on the left generates a reset at a fixed period determined by the software register. This will trigger the generation of a new packet of data as before. In simulation this allows us to limit the data rate to (50/200=1/4 * 200MHz =) 50MHz. Using these values in actual hardware would limit the data rate to (50/(8/10*156.25)) = 4Gbps.

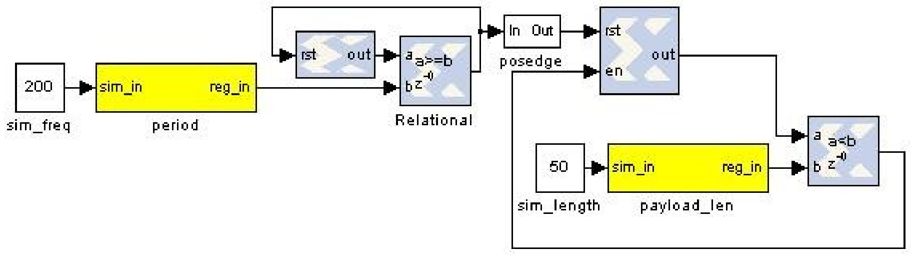


Illustration 6: Payload counter and period counter

4.8 Finalise logic including counter to be used as data

We will now finalise the data generation logic as shown below. To save time, use the existing logic provided with the tutorial. Counter1 in the illustration generates the actual data to be transmitted and the enable register allows this data stream to the transmitting 10Ge core to be turned off and on. Logic linked to the eof output port provides an indication to the 10Ge core that the final data word for the frame is being sent. This will trigger the core to begin transmission of the frame of data using the IP address and port number specified.

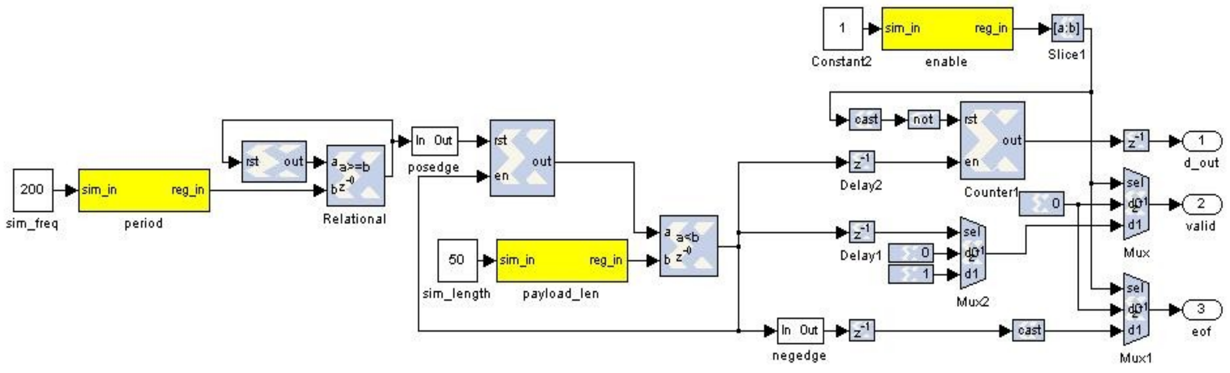


Illustration 7: Full data generation logic

To limit verbosity, major concepts and specific blocks of Tutorial 2 will be discussed in the following pages.

4.9 Receive blocks and logic

The receive logic is is composed of another 10Ge yellow block with the transmission interface inputs all tied to 0 as no transmission is to be done, however Simulink requires all inputs to be connected. Connecting them to 0 should ensure that during synthesis the transmission logic for this 10Ge block is removed.

4.10 Buffers to capture received and transmitted data

The transmission logic and receive interface of the 10Ge block receiving data are both connected to *snap64* blocks (located in *CASPER DSP Blockset->scopes*) that have been modified to capture status information associated along with the data. These blocks (snap_gbe0_tx and snap_gbe3_rx) are identical internally. Using these blocks we can capture data as it is transmitted and compare it to the data we receive.

Snap and *snap64* are a standard way of capturing snapshots of data in the CASPER tool-set. A s*nap* block contains a single shared BRAM allowing capture of 32-bit words while a s*nap64* contains two shared BRAM blocks supporting capture of 64-bit data. In this example, bram_oob along with the data path supplying it have been added to allow the capture of status information associated with the data.

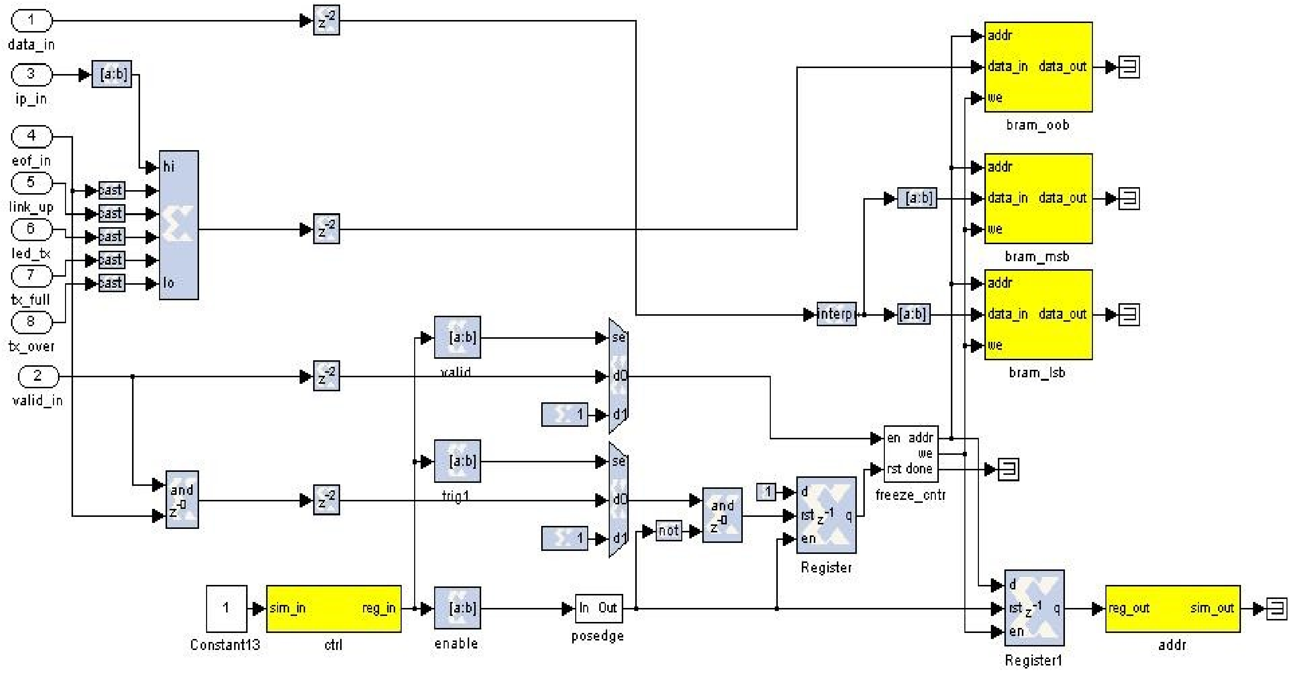


Illustration 8: Modified snap64

The *ctrl* register in a snap block allows control of the capture. The least significant bit enables the capture. Writing a rising edge to this bit primes the snap block for capture. The 2nd least most significant bit allows the the choice of a trigger source. The trigger can come from an external source or be internal and immediately. The 3rd most least significant bit allows you to choose the source of the valid signal associated with the data. This may also be supplied externally or be immediately enabled.

The basic principle of the snap block is that it is primed by the user and then waits for a trigger at which point it captures a block of data and then waits to be primed again. Once primed the addr output register returns an address of 0 and will increment as data is written into the BRAMs. Upon completion the addr register will contain the final address. Reading this value will show that the capture has completed and the results may be extracted from the shared BRAMs.

4.11 LEDs and status registers

We shall now look at some registers and LEDs to monitor the progress of our data transfer. We shall be able to check if the link on each 10Ge ports is up, whether transmission is taking place, if our buffers have overflowed etc via registers.

4.11.1 Transmission registers and LEDs

- *gbe0_tx_cnt* is attached to a counter that increments when the end-of-frame signal is raised on the input to the transmitting 10Ge block. It keeps a count of the number of frames transmitted.
- *led0_gbe0_pulse_tx* is attached to the 16th bit of this counter and will give a visual indication of data transmission.
- *gbe0_linkup* is a register that allows us to check if the transmitting 10Ge block is available for data transfer.
- *gbe0_rx* is a register that we can poll to see if the transmitting core is receiving any data, which should not be the case in this design except for house-keeping traffic.
- *gbe0_tx* can be polled to check for data transmission.
- *gbe0_tx_full* is a useful register allowing us to see when the FIFOs in the transmitting core are almost full, indicating that we are close to overflowing our transmission buffers.
- *gbe0_tx_over* indicates to us that transmission FIFOs have in fact overflowed and we have lost data.
- *led1_gbe0_tx_over* is a visual indication for the user of overflows in the transmission FIFOs.

4.11.2 Receiving registers and LEDs

- *gbe3_linkup* is a register that allows one to determine if the receiving 10Ge block is attached to a viable communications medium.
- *gbe3_rx* allows monitoring of whether the receiving 10Ge block is receiving data.
- *gbe3_tx* allows you to check whether the receiving 10Ge is transmitting data. This should be rare and should only occur during house-keeping operations such as ARP.
- *gbe3_tx_full* allows you to check whether the transmit FIFOs are almost full.
- *gbe3_tx_over* reflects whether the the transmission FIFOs hae overflowed.
- *led3_gbe3_rx_err* is an LED that reflects whether the 10Ge receiving block detected any errors during frame reception. A pulse extender block ensures that the LED turns on for long enough that a user can see when errors occur.
- *gbe3_rx_frame_err* is a register that allows you to check for receive errors via software.
- *gbe3_rx_frame_cnt* is attached to a counter that counts the number of frames received by the receiving 10Ge block.
- *led2_gbe3_pulse_rx* is an LED that gives a visual indication of the rate of frame reception in a similar way to *led_gbe0_pulse_tx.*

4.12 Compilation

Compiling this design takes approximately one hour and five minutes. A pre-compiled binary (.bof file) is made available to save time. This is already present on the ROACHs' filesystem as *tut2_2009_Sep_28_1407.bof.*

5 Control

In this next section we shall run tests using our compiled design. We shall learn how the 10Ge interface is managed via tgtap processes. We shall learn about the katcp python library that provides routines to access the ROACH board via the katcp server.

5.1 Recap of lessons from Tutorial 1

You should remember from Tutorial 1 that the end of the compilation process for our gateware produces a .bof file. This file can be executed on the ROACH like any other Linux executable by the custom BORPH-enabled kernel running on the PPC. Registers, shared BRAMs etc are availble for reading and writing as files in the */proc* filesystem.

A special telnet-like server called the 'katcp server' provides a simple interface to the outside world. You can telnet to port 7147 on your ROACH and give it commands in the KATCP protocol (try ?help for a list of available commands). This interface allows you to list available .bof files (via ?listbof) execute a .bof file (via ?progdev <filename>), list available registers, memory regions etc for access (via ?listdev), and write and read from these (via ?wordwrite <offset> <value> and ?wordread <offset> <size>).

5.2 Introduction to Python library for use with KATCP server

5.2.1 Getting the required packages:

These are pre-installed on the server in the workshop and you do not need to do any further configuration. Python and C APIs are available for connecting to KATCP servers. A further Python wrapper is available for simplifying communications with the KATCP server on ROACH. The KATCP package is available from the Python Package Index (PyPI) here: http://pypi.python.org/pypi/katcp and the wrapper is bundled in the CASPER Packetised Correlator control package, corr, found in CASPER SVN here: http://casper.berkeley.edu/svn/trunk/projects/packetized_correlator/corr-0.4.0/

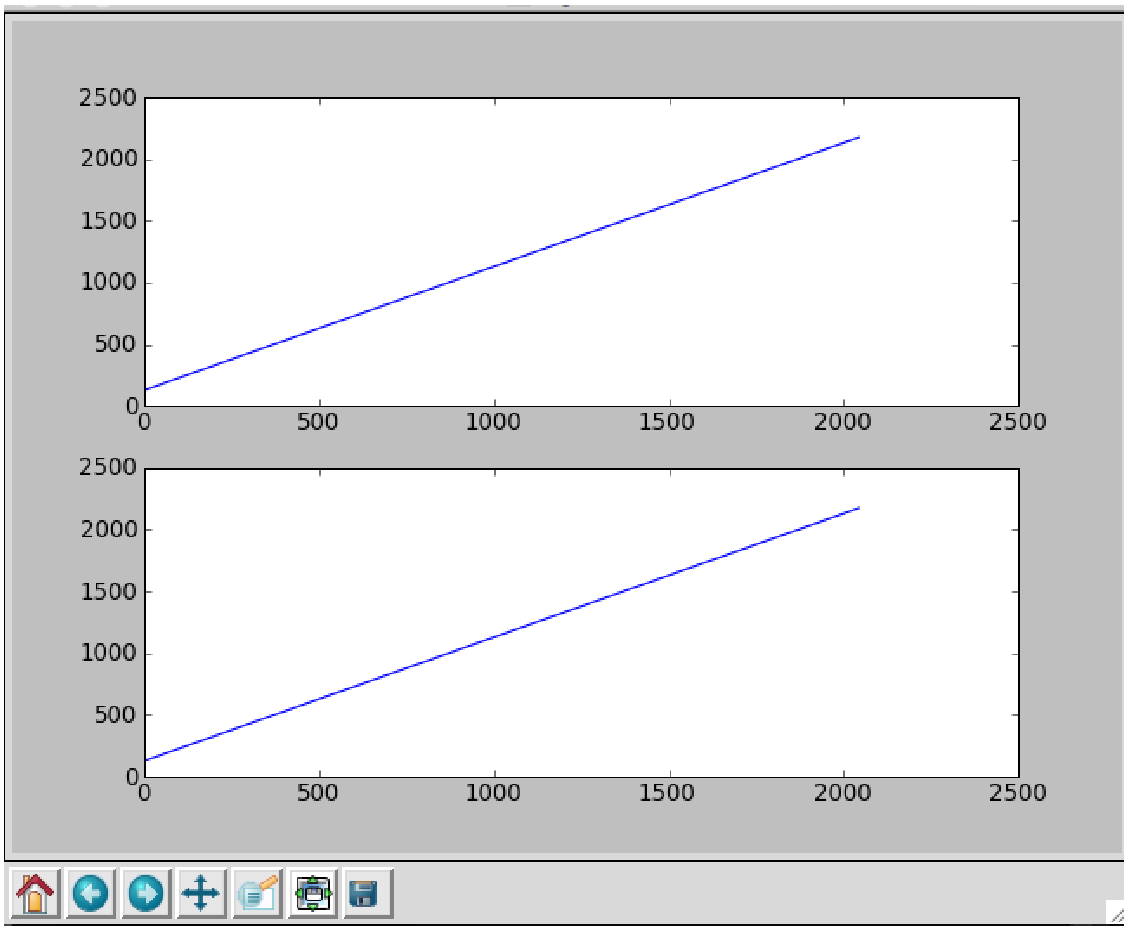
5.3 Testing our design using the provided demonstration python script.

Once we have compiled our design we are ready to test it in hardware. If you boot your ROACH, interrupt the boot process via the serial port and make it 'run netboot' (as in the first tutorial), it should have the bof file available in the */boffiles/* directory. The pre-compiled binary is called *tut2_2009_Sep_28_1407.bof.*

You can run the demonstration test script, *tut2.py* on the workshop Linux server computer, *wsserv* as follows:

SSH into wsserv. Double-click on the *xserver.bat* icon on your desktop. This will start an X11 server and then open a VT terminal. Type *ssh* *tutee@wsserv* and press enter. It will prompt for a password which is *tutee.*

Run the demonstration script. Type *./tut2.py roach020219 -p* and press enter. You should be presented with a crude figure containing two curves as follows:



The top curve represents the counter that you transmitted. The lower curve represents the contents of the first 2048 datawords received. Seeing as how you used a loopback cable, we expect these to be the same! If not, call a CASPERite over to help you debug.

5.4 What's going on behind the scenes?

Execute *./tut2.py* again, this time with the *-s* flag. This will suppress all the output lines decoding the snap block contents, allowing you to see the initilisation process. You should see something like this:

```
tutee@wsserv:~$ ./tut2.py roach020110 -s
Connecting to server roach020110 on port 7147...  ok

------------------------
Programming FPGA... ok
---------------------------
Port 0 linkup:  True
Port 3 linkup:  True
---------------------------
Configuring receiver core... done
Configuring transmitter core... done
---------------------------
Setting-up packet source... done
Setting-up destination addresses... done
Resetting cores and counters... done
Sent 0 packets already.
Received 0 packets already.
------------------------
Triggering snap captures... done
Enabling output... done
Reading 2048 values from bram snap_gbe0_tx_bram_msb... ok
Reading 2048 values from bram snap_gbe0_tx_bram_lsb... ok
Reading 2048 values from bram snap_gbe0_tx_bram_oob... ok
Reading 2048 values from bram snap_gbe3_rx_bram_msb... ok
Reading 2048 values from bram snap_gbe3_rx_bram_lsb... ok
Reading 2048 values from bram snap_gbe3_rx_bram_oob... ok
Unpacking TX packet stream...
Unpacking RX packet stream...
```

The script has some Python specific stuff where it defines some variables, the command-line options and some initialisation stuff. As part of this, it will establish a connection to your ROACH of choice using KATCP.

Let's try to replicate some of this script's functions manually. Start interactive python by running

ipython

Now import the correlator control library. This will automatically pull-in the KATCP library and any other required communications libraries.

import corr

5.4.1 Connecting to the board

To connect to the roach board, we define a new object. Let's call it *fpga.* The wrapper's fpgaclient initiator requires two arguments, the IP address or hostname of the roach board, and the KATCP port. KATCP defaults to port 7147.

fpga=corr.katcp_wrapper.FpgaClient('roach020110',7147)

The first thing we do is configure the FPGA. This happens on line 134 of the script.

fpga.progdev('tut2_2009_Sep_28_1407.bof')

5.4.2 Reading from software registers

Then we check to make sure that there is actually a cable plugged in, else it's pointless continuing. The 10GbE cores require some time to initialise and establish a link. So we sleep for 4 seconds to allow it to settle. Then we read our two software registers, *gbe0_linkup* and *gbe3_linkup.* These should both read *1.*

Reading and writing registers is easy with the python wrapper. It automatically packs and unpacks the binary data for you. There are a few functions available. For the 32-bit software registers, use *read_int* and *read_uint* to unpack signed and unsigned numbers respectively. Since these boolean numbers are only ever zero or one, it makes no difference which function you use. Let's try

In [3]:   fpga.read_uint('gbe0_linkup')
Out[3]: 1

5.4.3 Writing to software registers

This is very similar to reading from registers. Python will automatically figure out if you have a negative number that you need a signed number, and otherwise assumes an unsigned number. For example, to set the software register that configures the destination UDP port:

fpga.write_int('dest_port',60000)

5.4.4 Configuring the 10GbE cores

There is a special KATCP command to start the 10GbE userspace tap driver on ROACH.

The call is as follows: *fpga.tap_start(device_name, mac, ip, port)*. The device name is the name of your core as defined in Simulink. The MAC address is a 48-bit integer. Unless you have purchased a MAC address, we recommend you use a MAC address reserved for private networks (02:xx:xx:xx:xx:xx).
The IP address is the integer representation of the 32-bit address. The port defines the fabric UDP port. Any data received by the core not destined for this port will be redirected to the PPC for processing. This is how ARP and pings are handled (ie not by the FPGA fabric itself). If you disable the CPU interface by un-checking the boxes in the Simulink mask, obviously you will lose this functionality.

If you run tut2.py with the -a command-line switch, it will decode the 10GbE core's ARP table so that you can see how it has been configured and if it has correctly discovered other nodes on the network.

```
===============================
10GbE Transmitter core details:
===============================
------------------------s
GBE0 Configuration...
My MAC:  02 02 0A 00 00 14
Gateway:    0   0   0  20
This IP:   10   0   0  20
Gateware Port:  60000
Fabric interface is currently:  Enabled
XAUI Status:  7E
      Lane sync 0: 1
      Lane sync 1: 1
      Lane sync 2: 1
      Lane sync 3: 1
      Chan bond  : 1
XAUI PHY config:
      RX_eq_mix:  0
      RX_eq_pol:  0
      TX_pre-emph:  0
      TX_diff_ctrl:  4
ARP Table:
IP:  10.  0.  0.  0: MAC: FF FF FF FF FF FF
IP:  10.  0.  0.  1: MAC: FF FF FF FF FF FF
IP:  10.  0.  0.  2: MAC: FF FF FF FF FF FF
```

5.4.5 Reading and writing RAM

In much the same way as you would read and write registers, there is a function to read or write generic blocks of memory. Unsurprisingly, they are called *read* and *write*. The write function will automatically verify that the writes happened. There are also a *blindread* and *blindwrite* functions, which do not perform this verification which means you can obtain higher performance.

For example, to read one of the brams in the snap blocks, you could do

fpga.read(bram_name,tx_size)

Where *tx_size* is in bytes.

Python returns binary data in a string, so you need to unpack it into integers, floats, characters etc as required. There is a library called *struct* to do this. Do an *import struct* in iPython and then you can do packing or unpacking of binary data. There is onboard documentation, try *print struct.__doc__*.

For example, to unpack a big-endian 32-bit unsigned integer, we could do:

struct.unpack('>L',string_to_unpack[start_index:index_end_of_four_bytes])

5.4.6 Other notes

- iPython includes tab-completion. In case you forget which function to use, try typing *library_name.<tab><tab>*
- There is also onboard help. Try typing *library_name.function?*
- Many libraries have onboard documentation stored in the string *library_name.__doc__*
- *KATCP* in Python supports asynchronous communications. This means you can send a command, register a callback for the response and continue to issue new commands even before you receive the response. You can achieve much greater speeds this way. The Python wrapper in the *corr* package does not currently make use of this and all calls are blocking. Feel free to use the lower layers to implement this yourself if you need it!