UDP interface over GbE on FPGA

Andrej Seljak

To my best understanding this core was taken from opencores.org website and modified by a number of people. It still uses Xilinx IP emac core.

I patched the code to fix send/receive IP addresses and ports.
Overview

- Motivation:

  More channels, more data, more, more....... A fast data bus is needed.

  A core to communicate with the external GbE controller, not using Xilinx IP proprietary components is desired.

- UDP Basic description

- Block diagram

- Core structure

- Integration into the readout firmware

- How to read packets on the OS through python
UDP (User Datagram Protocol) is an alternative communications protocol to Transmission Control Protocol (TCP) used primarily for establishing low-latency and loss tolerating connections between applications on the Internet.

Like the Transmission Control Protocol, UDP uses the Internet Protocol to actually get a data unit (called a datagram) from one computer to another. Unlike TCP, however, UDP does not provide the service of dividing a message into packets (datagrams) and reassembling it at the other end.

Wikipedia

Block diagram

Good news that is fast, however unreliable. If a message gets corrupted somehow, you will never know it.
The entry ports to the UDP_integration file are made from FIFOs for sending and transmitting. These are 16 bit, parallel port style with additional flag bits.

The UDP_integration core manages header generation (IP, port number) and low level signals.

Code compiles using ISE 14.7.
CORE structure

FEATURES
- Implements UDP, IPv4, ARP protocols
- Zero latency between UDP and MAC layer
  (combinatorial transfer during user data phase)
- Allows full control of UDP src & dst ports on TX.
- Provides access to UDP src & dst ports on RX (user filtering)
- Couples directly to Xilinx Tri-Mode eth Mac via AXI interface
- Separate building blocks to create custom stacks
- Easy to tap into the IP layer directly
- Supports TX and RX with IP layer broadcast address
- Separate clock domains for tx & rx paths
- Tested for 1Gbit Ethernet, but applicable to 100M and 10M

LIMITATIONS:
- Does not handle segmentation and reassembly
  Assumes packets offered for transmission will fit in a single ethernet frame
  Discards packets received if they require reassembly
- Currently implementing only one ARP resolution slot
  means only realistic to use for pt-pt cxns (but can easily extend ARP layer to manage an array of address mappings
- Doesn't currently double register signals where they cross between tx & rx clock domain in a couple of places.
CORE structure - Customized setup
“the tweak”

Setup the IP and MAC of the board:

UDP_integration_example.vhd

```vhdl
-- set up our local addresses and default controls
our_ip <= x"c0a80509"; -- 192.168.5.9
our_mac <= x"002320212223";
```

TX Port setting:
UDP_TX.vhd

```vhdl
when x"0002" => tx_data <= "11011101"; --udp_txi.hdr.dst_port (15 downto 8); -- dst port
when x"0003" => tx_data <= "10101011"; --udp_txi.hdr.dst_port (7 downto 0);
```

If the port is not fixed, it would change value at every transmission. This makes it impossible to fetch data in software.
Firmware

PC/software

Ethernet PISO IN/OUT

CPU

Requested data

DAQ

Runtime parameters:
- mode of operation
- Pretrigger value
- No. of Windows to read

Calibration mode options:
- Select preferred channel
- Select window to read

64 bit read/write bus controlling drivers.

HG2 HG2 DDS AD
CSA CSA TMP DAC

DAQ drivers

Trigger logic

Physical connection to chips

andrejs@hawaii.edu 8/22/2016
How to: OS and Python

Wireshark example for a single data packet.

You need to make a bit of your own data format.
How to: OS and Python

Define sending command for GbE buffer:

```python
import socket
import codecs
import os
import sys

UDP_IP = "192.168.5.9"
LOCAL_IP = "192.168.5.7"
UDP_MAC = "00:23:20:21:22:23"
UDP_PORT = 9071

class interface(object):
    def __init__(self, IP = UDP_IP, PORT = UDP_PORT, LOCAL_IP = LOCAL_IP):
        self.IP = IP
        self.PORT = PORT
        self.LOCAL_IP = LOCAL_IP

    def send(self, header, id_daq, id_sc, reg, data):
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, 0)
        self.socket.connect((self.IP, self.PORT))
        os.system('arp -s '+UDP_IP+' '+UDP_MAC)
        data1 = codecs.decode(header, "hex_codec")
        data2 = codecs.decode(id_daq + id_sc, "hex_codec")
        data3 = codecs.decode(reg, "hex_codec")
        data4 = codecs.decode(data, "hex_codec")
        self.socket.send(data1)
        self.socket.send(data2)
        self.socket.send(data3)
        self.socket.send(data4)
        self.socket.close()
```

Write buffer:

```python
udp = interface()
udp.send("0000", "0700", "0000", "F000") # manual mode
```

Reading from GbE buffer:

Settings:

```python
UDP_IP = "192.168.5.7"
UDP_PORT = 56747

sock = socket.socket(socket.AF_INET, # Internet
    socket.SOCK_DGRAM) # UDP

sock.setsockopt(socket.SOL_SOCKET, 25,"eth1")
sock.bind((UDP_IP, UDP_PORT))
sock.setblocking(True)

udp = interface()
```

Reading a packet:

```python
data, addr = sock.recvfrom(128)
my_hex = binascii.b2a_hex(data)
sys_id = my_hex[0:2]
my_id = int(my_hex[0:2],16)
DAQ_mode = hex(int(my_hex[2:4],16) & 0x0F)
DAQ_chip = (int(my_hex[4],16) & 0xF) # & 0xF
DAQ_channel = (int(my_hex[5],16) & 0xF)
DAQ_address = (int(my_hex[8:10],16) & 0xFF)
DAQ_sample = (int(my_hex[10:12],16) & 0xFF)
DAQ_data = (int(my_hex[12:16],16) & 0X0FFF)
```