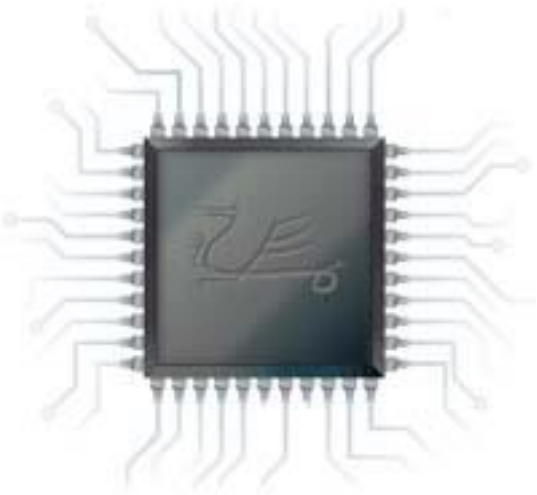
Ethernet and TCP/IP



Simon Southwell

2nd August 2024

Preface

This document is a PDF version of an article written in August 2024 and uploaded to LinkedIn, on Ethernet and TCP/IP

Simon Southwell
Cambridge, UK
2nd August 2024

Contents

Introduction

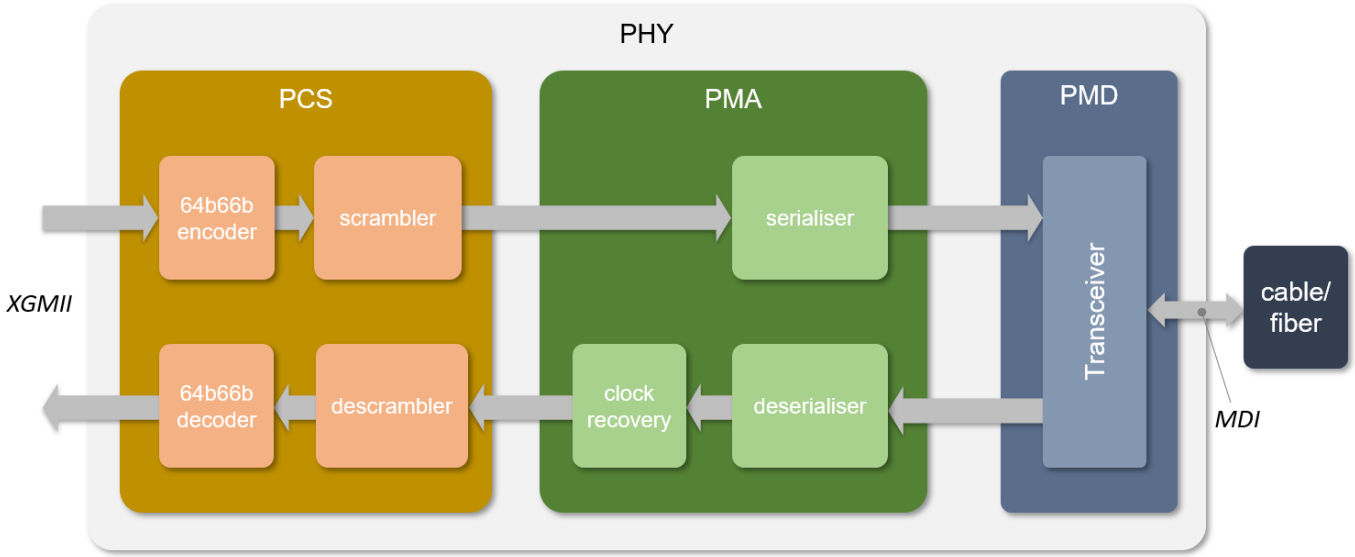
I have had a few requests to cover something about ethernet over the last few months and so I've finally put pen to paper (as it were). Ethernet is a large topic and has been around for some time, with its origins back in the 70s and standardization being obtained in the 80s. I don't want this article to be a full history of ethernet and I like to keep my subjects up-to-date, relevant and practical (as much as possible). Therefore, in this article, I want to talk about ethernet with reference to current technology and will concentrate on 10 gigabit ethernet (10Gbe), the XGMII interface and, because many applications wish to connect to the internet, TCP/IP (thinking of IoT). Also, I have some experience in these areas. This is isn't all there is to ethernet, by any means, but there is much overlap with other standards still in use such as UDP in place of TCP and 1Gbe/RGMII in place of 10Gbe/XGMII and the case study of 10Gbe will hopefully broaden into understanding of the related alternatives. What this article isn't about is the actual wires and fiber-optic cables, network topologies or wireless, but is about the sort of thing you might come across in, say, an FPGA with high speed transceivers and hard PHY layer implementations suitable for ethernet.

We will take a bottom up approach, starting with the physical layer before moving up through the media access control layer (MAC), internet protocol layer (IP—looking at version 4) and to the transmission control protocol (TCP). On the way we will look at the XGMII interface, where the 'XG' part stands for 10 gigabit and 'MII' is for 'media independent interface' and have a brief look at XAUI ("zowee"), standing for 10 (Gbe) and 'attachment unit interface'.

There is a working simulation model to accompany this article on github—the TCP/IP pattern generator (tcpIpPg) and we will briefly look at what this does with reference to the article's subject matter though, of course, more detail can be found in the manual.

PHY Layer

The physical layer (PHY) is the lowest level ethernet layer which includes the actual media that data is transferred across. This might be twisted pair wires or fiber optic cables, for example. Before any data can be transmitted and successfully recovered at a receiver it must be conditioned so that this is possible. There are three sublayers in the PHY layer. The diagram below shows a simplified illustration of these layers.

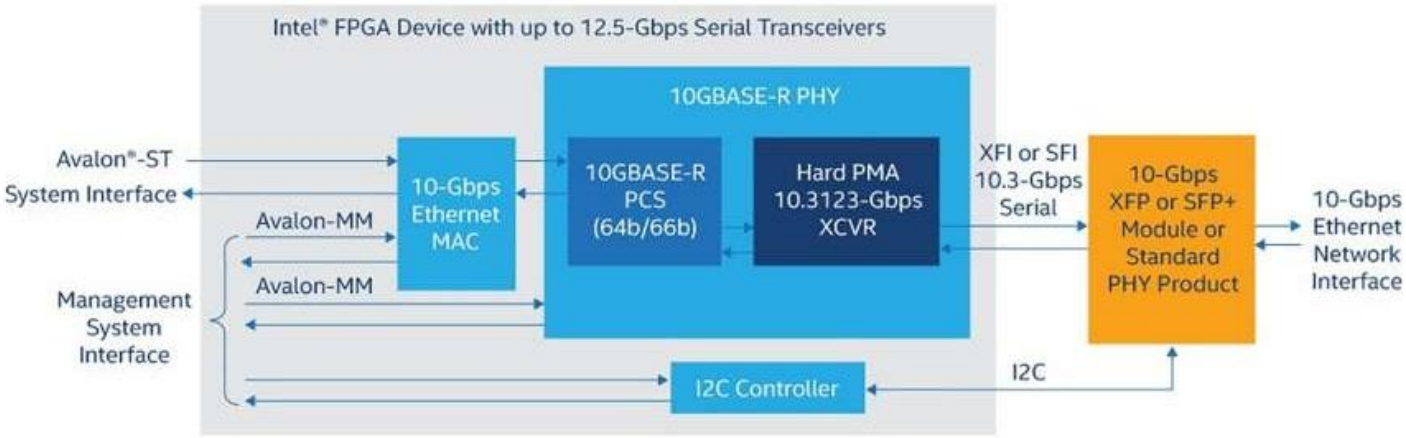


Firstly, a physical coding sublayer (PCS) is responsible for encoding/coding data, which for 10Gbe is usually a 64b66b encoding but can be 8b10b or even 64b65b, which allows sufficient transitions in the data for a clock to be recovered from the data at the receiver. In addition, it 'scrambles' the data with a pseudo-random binary sequence to attempt to spread the spectrum of frequencies generated by the data patterns which, if not scrambled, might have a repetitive pattern with certain frequencies dominating for a significant period. The interface to the upper layers is via a media independent interface which, for 10Gbe, is XGMII, and we will be looking at this later. The use of a standard interface such as XGMII abstracts away the specifics of the PHY IP, allowing MAC layer IP to be reused with a variety of physical layers and vice versa.

The physical medium attachment sublayer (PMA) performs serialisation and deserialisation of the scrambled data and, on the receiver side, performs clock recovery and synchronisation of byte boundaries, detecting special symbols for synchronisation to boundaries.

The physical medium dependent sublayer (PMD) is where the transceiver, specific to the actual media, lives. For example, a 10G SFP+ transceiver (enhanced small form factor pluggable) supporting up to 16Gbits/s. The actual connection to the media is through a media dependent interface (MDI), specifying, both electrically and physically, the connection the cables are plugged into.

As a real-world example, the diagram below is taken from Altera documentation, showing their 10GBASE-R PHY FPGA offering, in context with a MAC and PMA, where the PCS is a soft core and the PMA a hard macro implementation.



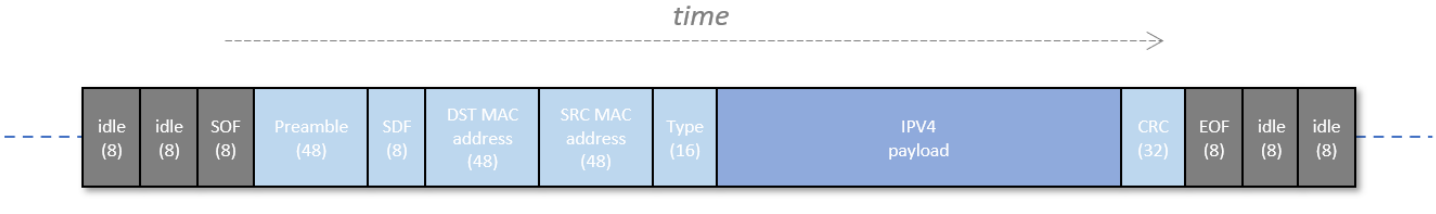
Ethernet Protocols

Now we have a connection to the physical media, with enough coding to recover a clock and the raw data, the actual protocols to be used need laying on top of this. Some of you may be familiar with the OSI model of networking, with its seven layers from the physical layer through to the application layer. Since we are concentrating on the TCP/IP model it is worth pointing out that there is some correlation between the TCP/IP model and the OSI model, but it is not identical. The OSI physical and data layers are merged into a network access layer, the network layer has an internet layer equivalent, there is a transport layer for both, and the top three OSI layers (session, presentation and application) are replaced by a single application layer. On the face of it these differences are minor, but whilst the OSI model is a generic networking model, the TCP/IP is geared specifically for the internet, and that's the main difference.

Before we can talk about the TCP/IP we need a basic and generic framing protocol in which to transfer higher level protocols.

Ethernet Frame

Data is transferred over the ethernet media as frames. The diagram below shows the construction of an ethernet frame with the different components:



Something must be transmitted all the time over ethernet, and there are various control symbols, each of which are 8 bits, transmitted least significant bit first, but also flagged as control to differentiate them from packet data. One of these symbols is an 'idle' symbol which is transmitted when no frame is in progress. A list of the control symbols is given below:

- IDLE: `0x07` : Idle symbol
- SOF: `0xFB` : Start of frame symbol
- EOF: `0xFD`: End of frame symbol

When a frame is to be transmitted, an SOF symbol marks the start of the frame, and it is terminated with an EOF symbol. The frame we want to transmit is actually a type II ethernet frame, which can carry various payload types, such ARP and (for our purposes) IP. A type II frame starts with a preamble, which is a set of six `0x55` bytes (note that these preamble bytes are not control symbols). After the preamble a "start of frame delimiter" (SFD = `0xD5`) is transmitted followed by a destination and source MAC address, each of which is 6 bytes in length (transmitted most significant byte first, with each byte transmitted lsb first).

A MAC (media access control) address is a unique identifier on a network for each device that's connected. It is usually associated with a network interface card (NIC) and is usually hard wired. A computer may have more than one interface, such as an additional Wi-Fi adapter, in which case both interfaces will have their own unique MAC addresses. You can find your own system's MAC addresses if you want. On Windows, in a console, typing `ipconfig /all` will list all the adapters, and the 'Physical Address' fields are the MAC addresses for each adapter. On Linux you can type `ifconfig -a` and the numbers next to the 'ether' field is the six byte MAC address. So, the ethernet frame specifies the destination for the frame, and also provides of the address of the adapter that transmitted it for the routing of any responses.

The header part of the frame is finished off with a 'type' of 2 bytes. This specifies the protocol of the payload. The ethernet frame, as mentioned before, can carry various protocols, and each one is assigned a unique type code by the IEEE Registration Authority. For IPv4, the type code is `0x0800`, whilst for ARP it's `0x0806`. A list of other type codes can be found here.

The payload of the frame then follows. There are restrictions on the size of the payload, which has a maximum of 1500 bytes (and minimum of 42—the smallest amount of header), though some implementations allow larger frames, known as jumbo frames, but the IEEE 802.3 specification defines the limit as 1500.

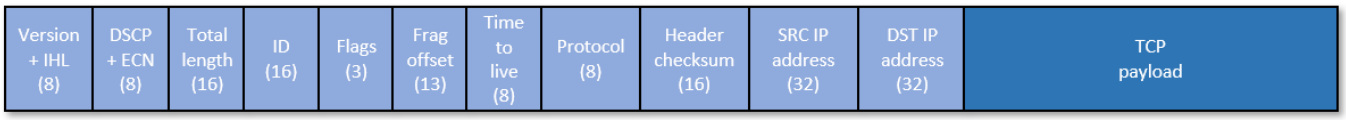
After the payload a 32-bit (4 bytes) CRC is transmitted, sometimes known as the frame check sequence (FCS). The CRC is calculated over the MAC addresses, type and payload. The polynomial for the CRC (see my article on error correction and detection) is specified as `0x04C11DB7` and complemented, but also, confusingly, to be transmitted most significant bit first, unlike the rest of the packet. This has caught out many an implementer—myself included. If one bit reverses the polynomial (`0xEDB88320` and still complemented) and shifts as per the rest of the packet data, then the same bits get transmitted and is easier to implement.

One final thing to thing to discuss is 'inter packet gap' (IPG). A minimum amount of time may be required between transmission of frames to allow a receiver to prepare for reception of a new packet. For example, for 10Gbe, a 9.6ns gap is a minimum requirement, equivalent to 12 bytes of idle. A receiver will need to be tolerant of a smaller IPG to allow for network delay variations and clock tolerances: e.g., 4ns for 10Gbe.

IPv4

Above the ethernet layer (which, as mentioned before, is part of the network access layer) is the IP layer, which is part of the network layer. For this article we will be looking at the IP version 4 protocol. The basic format for an IPv4 packet is shown below:

IPv4



In the diagram I have amalgamated some fields into a single byte, but the protocol has fields which vary in bit length which aren't necessarily 8-bit aligned. We will go through the fields in a little more detail and highlight the most essential fields.

The first field is the **version** field, which is a 4-bit value to specify the IP protocol version. For IPv4 this is always 4. This is followed by another 4-bit field, the internet header length (**IHL**). This value is variable as there are some optional fields that can be added to the header and its units are in 32-bit words. With no optional fields (as in our example), this has a value of 5.

The differentiated serviced code point (**DSCP**) is 6 bits and is involved in defining which 'class' the packet is to specify the type of packet data for such things a quality of service (QoS) for, say, voice or video. This allows different routing and prioritising through the network based on that class. The code for normal traffic is 0. The explicit

congestion notification (**ECN**) field is an optional field of two bits. This is used by two ECN capable endpoints to notify of impending network congestion in place of dropping packets. Normally, this field is 0 for non-ECN devices. The DSCP and ECN fields make up the packet's Traffic Class.

The **Total Length** field is 16 bits and gives the total length of the entire packet, including the header and data, in units of bytes. The minimum size this can be is 20 bytes (5 words), as this is the size of the smallest header with no data. The maximum size is 65535 bytes.

The 16-bit identification (**ID**) is used to group data that has been fragmented into separate packets and are then assigned the same ID number. The ID is followed by a 3-bit **Flags** field, involved with controlling fragmentation of data. The 3 bits are defined as follows:

- bit 2: more fragments
- bit 1: don't fragment
- bit 0: reserved (0)

If bit 1 is set, then a packet will be dropped if it requires fragmentation. This might be set if sending to a device that can't reassemble fragmented data. If a packet is fragmented, then bit 2 is set on all but the last fragment sent to indicate that more data is expected. The Flags are followed by a **Fragment Offset** of 13 bits. This gives the offset of the fragment data, in bytes, relative to the beginning. Thus the first fragment is always 0.

The time to live (**TTL**) byte nominally gives a value in seconds for the packet before it is dropped. In actuality the value is decremented at each router and when it reaches 0 the packet is dropped. Thus it is really a hop count mechanism.

The **Protocol** byte specifies the protocol that the payload uses. For TCP this value is `0x06` whereas UDP it's `0x11`.

The next field is a **Header Checksum**. Confusingly, it does include the two IP addresses that follow it but, of course, does not include itself (equivalent of 0 at these byte positions). The checksum adds all the 16-bit header fields. If there is a carry value in bits 19 down to 16 of the sum, this nibble is added to the sum. The sum is then inverted and the bottom 16 bits are the checksum value. Note that, since the TTL field can change, the checksum is recalculated at each hop in the network.

The final two 32-bit fields are the source and destination **IPv4 addresses**. The 32-bit IPv4 addresses are usually represented in a dot-notation, with decimal values for

each byte. E.g. `192.160.0.56`. The different values of the first two most significant bytes represent different classes of network, so that, for example, the 192.68.x.x is for local data transfer on a private network, with up to 65536 addressed components.

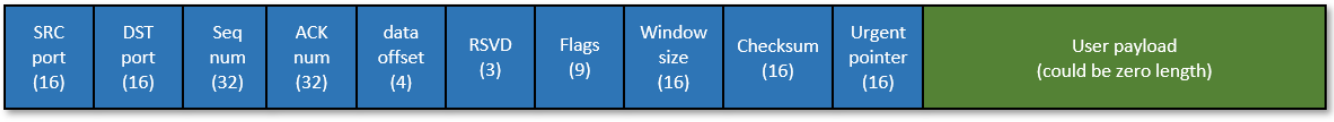
These addresses are usually dynamically allocated on a network and use a dynamic host configuration protocol (DHPC) service to do the allocation as devices are added and removed from a network. This allows IP addresses to be reused and avoid clashes with duplicate static IP addresses or running out of IP addresses to statically allocate, even though not every device would be connected to the network. Once again, just as for the MAC addresses, you can inspect the IPv4 addresses of your devices with `ipconfig` or `ifconfig`. If you ping a website it will usually display the IP address, though these days this will likely be an IPv6 address. But, at the time of writing, the following was done:

```
$ ping bristol.ac.uk

Pinging bristol.ac.uk [137.222.1.237] with 32 bytes of data:
Reply from 137.222.1.237: bytes=32 time=8ms TTL=246
Reply from 137.222.1.237: bytes=32 time=9ms TTL=246
Reply from 137.222.1.237: bytes=32 time=9ms TTL=246
Reply from 137.222.1.237: bytes=32 time=9ms TTL=246

Ping statistics for 137.222.1.237:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 8ms, Maximum = 9ms, Average = 8ms
```

TCP

The transmission control protocol (TCP) is responsible for providing reliable communications over an IP network between applications running on different computers, with ordering and error detection. In TCP, data is constructed into segments that are encapsulated by the IP packets—i.e., it is the IP payload. The TCP segment format is shown in the diagram below:

TCP



A TCP segment consists of a header followed by a payload, which may be zero in length. The first two 16-bit half-words are the source and destination **Ports**. A 'socket' allows connection from some system (a client) to another that's running some TCP 'server' service. This server can support multiple connections and each will have a unique port number for that server. If you type (for Windows) `netstat -a`

and you will get a list of connections with IP addresses, followed (after a ':') by the port number being used. The source and destination ports being used, then, are these port numbers.

The 32-bit values that follow are for the **Sequence Number** and the **Acknowledge Number**. This is the mechanism for acknowledging the correct reception of payload data. The sequence number is dual purposed and is involved with establishing a connection (when the SYN flag is set), but we will discuss this in a separate section to follow. Normally it will be the 'sequence number' of the first byte of data in the payload, where the sequence number represents a relative number of bytes since connection established. So, for example, if at establishment of a connection, the sequence number is 0 (it doesn't have to be—see later), and before our current packet 752 bytes have been sent, then the sequence number for the new packet will be 752 (`0x2f0`). To acknowledge data correctly received, a returned packet will set the acknowledge value (when ACK flag set) to be the sequence number it is expecting to see next. So, if a returned acknowledge packet has an acknowledge value of 701, then it has successfully received 701 bytes (again assuming a sequence start value of 0).

The 4-bit **Data Offset** field is the offset from the beginning of the TCP segment to the start of the data payload, in units of 32-bit words. This is followed by 4 bits of reserved bits, set to 0, which precedes a byte of single bit flags with various functions. A list of these flags is shown below:

- **CWR**: congestion window reduced: This is involved in explicit congestion notification, as mentioned in the IPv4 section. Set by a host if seen an ECE bit set.
- **ECE**: ECN-echo. Also for ECN notification, set when ECN bits in an IP header of 11b have been seen.
- **URG**: Urgent: Set if the urgent pointer field is to be used.
- **ACK**: acknowledgment: the acknowledge field has a valid acknowledgement value
- **PSH**: push: request to push buffered data to receiver, to flush data towards the destination.
- **RST**: reset: reset the connection
- **SYN**: synchronise the sequence: used to set the initial sequence number during connection (see below).
- **FIN**: finish: flags last packet from sender, used during connection closure (see below).

The **Window Size** 16-bit value is used to indicate how much data the receiver can get before it has to acknowledge any. This prevents unnecessary ACK traffic on the network if every packet had to be acknowledged. Since each packet will have a window size of its own, this is a dynamic window, and packets that require a more urgent acknowledgement can have a smaller window size. The window size units default to bytes, but window scaling can be used to alter this using optional header fields.

Following the window size field is a 16-bit **Checksum**. The TCP checksum is somewhat odd in that includes fields from the IPv4 header, as well as being computed over the TCP header. Specifically, the IPv4 source and destination addresses, the protocol field and the TCP length. With these IPv4 fields, the TCP header is included in the sum along with the data payload. It is computed in the same way as the IPv4 checksum, with carry values added to the bottom 16-bits and the inverted bottom 16 bits used as the result.

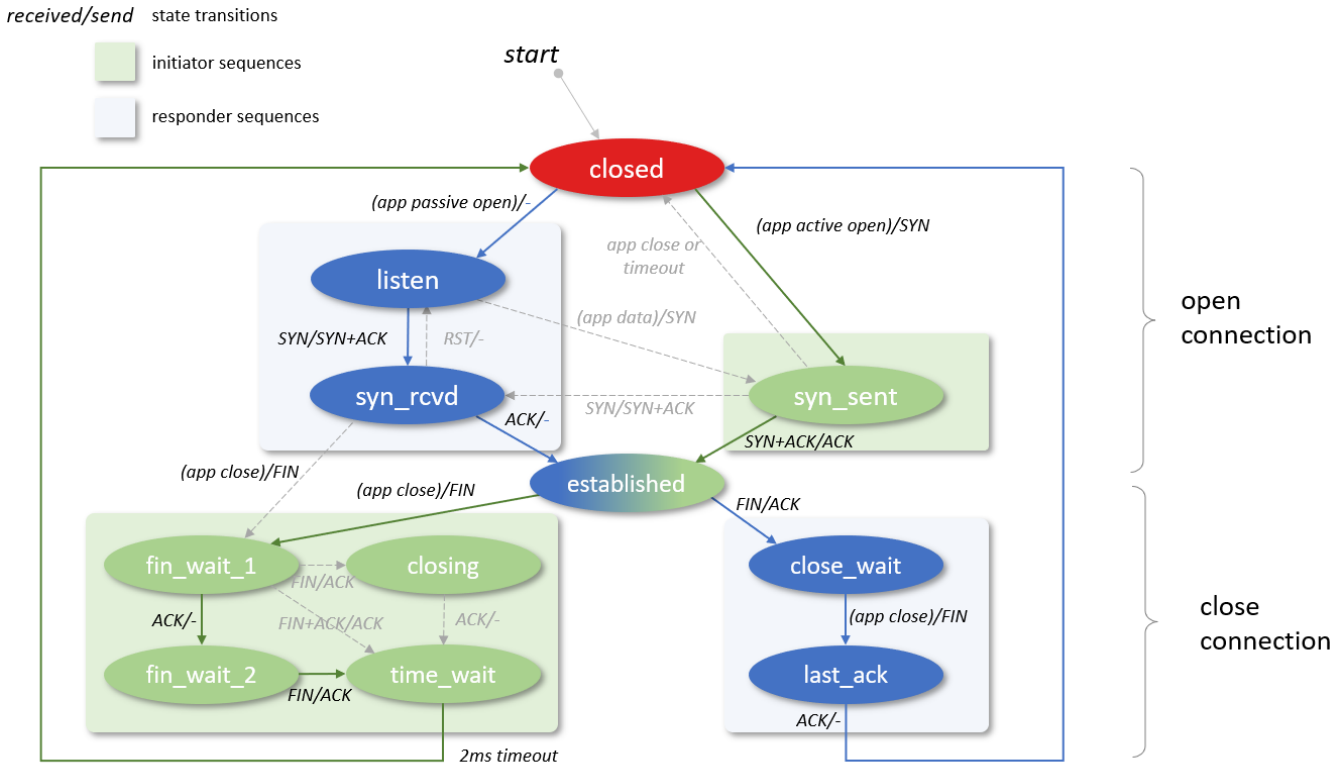
Lastly, the **Urgent Pointer** (16 bits), valid if the URG flag is set, points to the last byte of urgent data as an offset from the sequence number. This is for the use of the receiver, to upgrade priority of data for processing immediately, possible by placing in a higher priority queue to bypass other already queued data.

Like IPv4 there are optional header fields. This introduction will not go into detail of these fields, but these include things such as window scaling, timestamps, and maximum segment size (MSS). If a header is not a multiple of 32-bits, then padding bytes are added to the end.

The data payload follows the header and can be zero bytes. Before we can transfer data, though, an end-to-end connection has to be established and, when all communications have finished, the connection closed. A procedure is defined with various states for this purpose, and this will be looked at in the following section.

TCP Connection

When communication is required between two peers a connection has to be established between them and, at some point, closed once more. Usually there will be an initiator (e.g., a TCP client) and a responder (e.g., a TCP server), but connection may be possible from either peer, with the scenario where both ends try and establish a connection simultaneously. For our purposes we will restrict ourselves to a normal initiator/responder establishment and closure procedure. The diagram below shows a state diagram for connection and closure:



In the above diagram the main flow through the states are shown in solid colour, and the exception transitions shown in grey for reference. The blue states and transitions are for a responder whilst the green states and transitions are for an initiator. The transition labels represent what is received and what is sent in reply (if anything).

Starting from a closed connection, an application will instigate to open a connection, and will cause a TCP packet with the SYN flag set to be sent. The sequence number will be valid and will be set to the value from which the responder should start its Acknowledge numbering. The initiator is now in the `SYN_SENT` state. The responder (as a server) will automatically move to the `LISTEN` state waiting for an initiator to send a SYN packet. When this is received, it responds with a packet with both SYN and ACK flags set (a SYN+ACK packet). The sequence number sent is the value that the responder requires acknowledge values to start from the initiator, and the Acknowledge value will be the sequence number that was sent in the SYN packet of the initiator. The responder is then in the `SYN_RCVD` state. When the initiator receives the SYN+ACK packet it acknowledges this by sending an ACK, with the sequence number sent by the responder, and then is in the `ESTABLISHED` state. The responder gets the ACK and moves to the `ESTABLISHED` state as well, and data may now be exchanged. In fact, the initiator can send first data in the ACK packet for the received SYN+ACK, if it is able.

From this point onwards, data can be exchanged with the sequence and acknowledge protocol, with the sequence number indicating the sequence of the first data byte of the payload, and the acknowledgements returned giving the sequence number *expected* for the next data byte. Once a connection is finished with, there is a close mechanism. Closing of a connection can be initiated from either side, but let's walk through the initiator closing the connection.

When the initiator's application closes the connection, with no more data to be sent, a packet is sent with the FIN flag set and it moves to the `FIN_WAIT_1` state. On receiving the FIN packet, the responder acknowledges that packet with an ACK packet and moves to the `CLOSE_WAIT` state. Note that the returned sequence will be incremented by the reception of the FIN packet. When the initiator receives the ACK it moves to `FIN_WAIT_2`. At some point the responder's application closes the connection, possibly having sent final data, and sends a FIN packet and moves to the `LAST_ACK` state. The initiator, on receiving the FIN packet, sends an ACK and moves to `TIME_WAIT`, which will then move to `CLOSED` after a timeout period, to allow all transfers to complete. The responder, on receiving the ACK, will then also move to `CLOSED`.

So we now know all the protocol we need to send data across a network, using an established connection, encoded as TCP/IPv4, and to cleanly close this connection afterwards. As I've mentioned before, there are more cases involved than we have space to cover here in this introduction, and we have only had a peek at the higher layers but, fundamentally, we have enough information to understand a working implementation. When putting a logic implementation together using IP blocks, there is some help that can abstract and standardise connection between blocks and we will look at this in the next section.

10Gbe and XGMII

In a typical system (your mileage may vary), one might have separate blocks for the PHY layer implementation (and we saw the Altera offering earlier), then a MAC block to implement the IP protocol and then a TCP client or server block, though the MAC and client/server components might be a single piece of IP. Between the MAC and the PHY implementation a media independent interface (MII) is likely to exist which abstracts away the details of the PHY into a simple data transfer interface with some control. Different variations of the interface have been specified over time, such as GMII, RGMII, MII and RMII but, for our purposes, sticking to 10Gbe as our case study, we will look a XGMII.

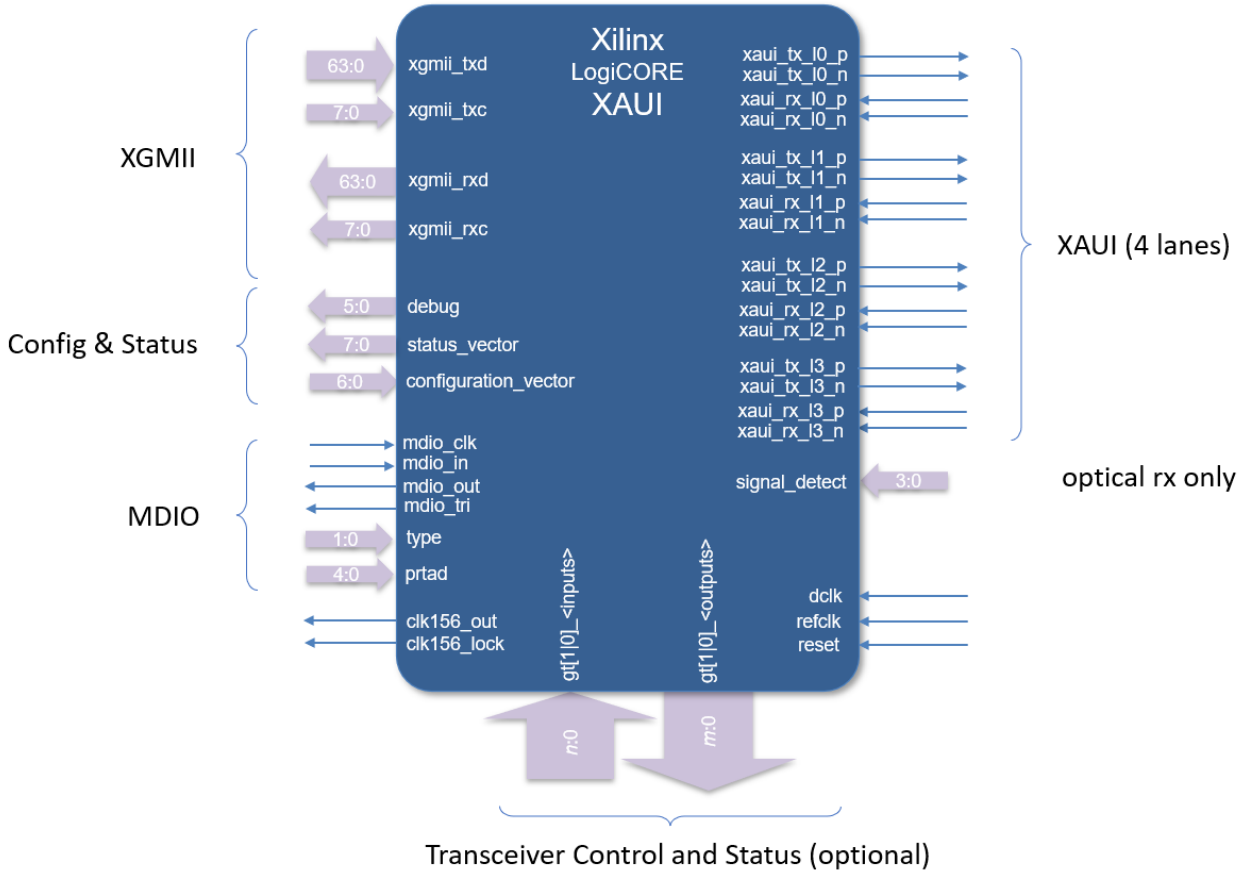
The XGMII interface is incredibly simple, with a transmit interface and a receive interface. The interfaces have a data bus (`txd/rxd`) which is 64 bits wide (8 bytes) and an 8-bit control (`txc/rxc`), with a control bit for each of the bytes in the data bus. The control bits differentiate each byte as ethernet control symbols from just ordinary data, as was discussed in the PHY Layer section earlier. The clock rate is 156.25MHz, which is 10GHz (for 10Gbe) divided by 64, the width of the bus. This explains why we have a wide bus. Clocking logic at 156.25MHz should be fairly straight forward compared to higher frequencies with a narrower bus.

And that's all there is to the XGMII interface. There are no valid signals as (as we saw earlier) when no data is transmitted, `IDLE` symbols are sent, so something is sent or received every cycle. There is a potential problem though. The XGMII interface, excluding the clock, is 144 bits wide. Within an ASIC or FPGA, routing this many pins is probably not an issue, but if one were utilizing an external component to implement all the PHY functions, then these signals would need to be routed across a PCB or even a backplane. The solution to this is XAUI.

XAUI

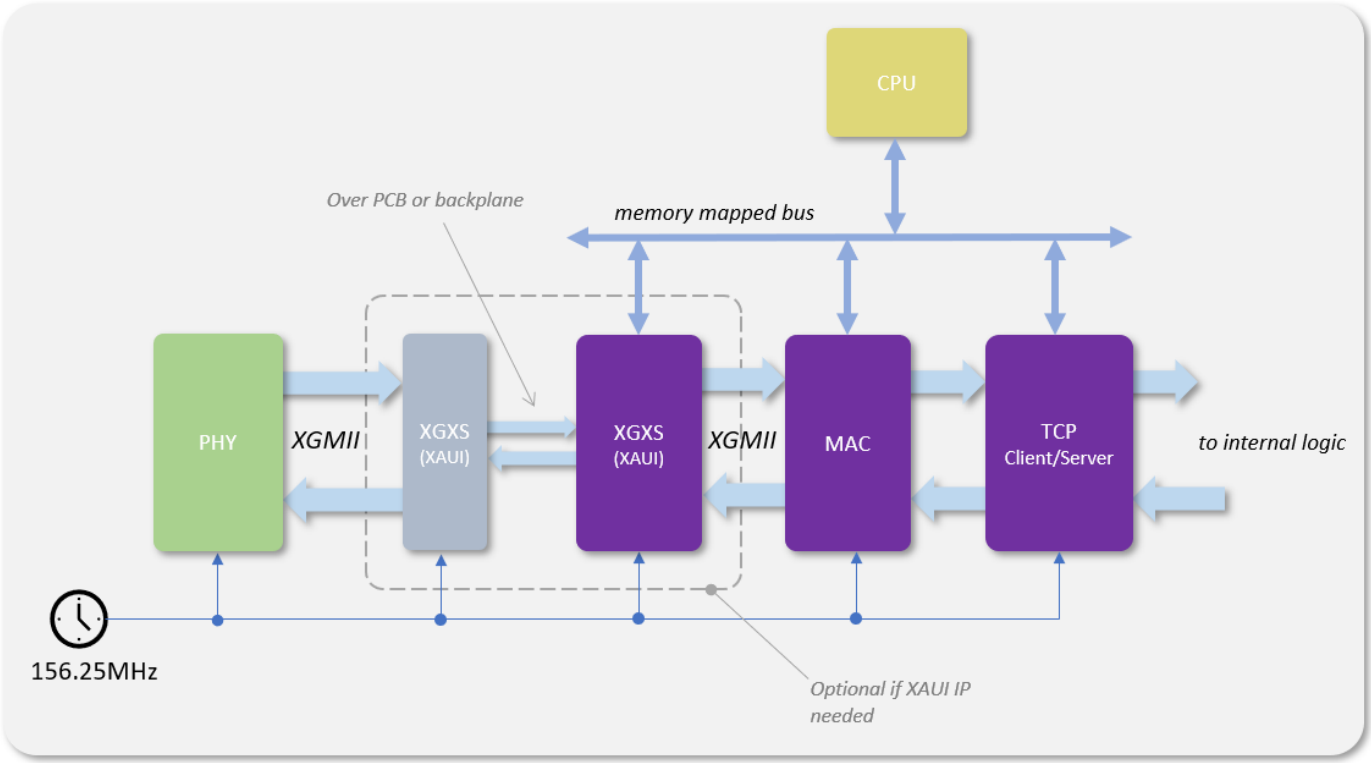
XAUI stands for the 10Gb (X) 'attachment unit interface'. Its purpose is to reduce the pin count of the XGMII interface, from 144 to 32 (16 in each direction) and extend the length of routing that the signalling can reach. It does this by encoding the data with 8b10b and serialising into four differential signal 'lanes' for each direction at 3.125Gb/s.

From within the 8b10b control symbols it defines control codes used to synchronise and align data to recover the bytes, along with a clock. From a system implementer's point of view, though, a pair of XGXS (XGMII Extender Sublayer) blocks, with XGMII to XAUI conversion, at either end of the PHY/MAC connection, seamlessly interfaces the components as if a direct XGMII connection was in place. A typical XAUI IP core is shown in the diagram below.



In this case it shows the pinout of the Xilinx LogiCORE XAUI IP. On one side is an XGMII interface, and on the other the 4 XAUI lanes with differential signalling. In addition, control of the block is provided by an MDIO interface, with some additional configuration and status signals separated to their own pins. A `signal_detect` input, when using optical fiber (else they are tied high), alerts the IP when an optical connection is present, and there are clock and reset inputs, along with a synchronised clock output and lock signal. Not shown in detail, but there are some, quite wide, optional transceiver control and status pins as well.

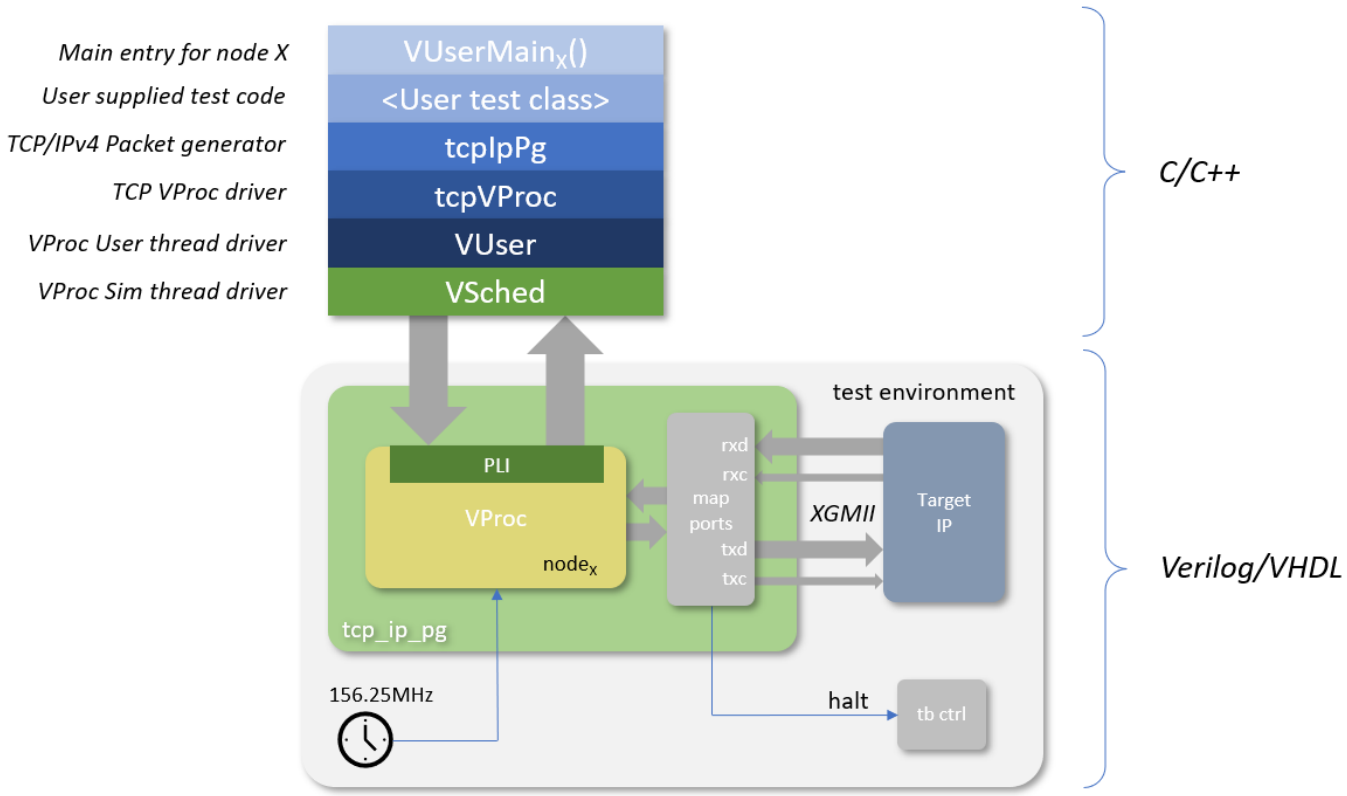
To sum up what we've talked about so far, then, the diagram below shows a typical implementation of a 10Gbe system with TCP/IPv4. It shows a PHY, as per the PHY Layer section above, with an XGMII interface, an optional pair of XGXS blocks, if the PHY is off chip, a MAC with an XGMII interface and a TCP client or server, with an interface to the rest of the system's logic. A common clock is shown but, obviously, if the PHY is remote, the PHY and its XGXS will be clocked from their own clock source. If the PHY is local, and there is no XAUI connection, then the PHY is likely to be on the same memory mapped bus as the other components. As we have seen, a typical system these days, such as an FPGA with hard macro components, will likely be all encompassing, down to the physical layer's PMA, and XAUI is not needed.



tcpIpPg Model

As with many of my articles, there is verification IP or models to accompany the material so that exploration of the subject can be undertaken using working examples that run on a logic simulator. The tcpIpPg (TCP/IPv4 pattern generator) component is just such a model, utilizing the VProc virtual processor, which I've written about in a previous article. This allows modelling of protocols in software, and these models to be driven by user written test code.

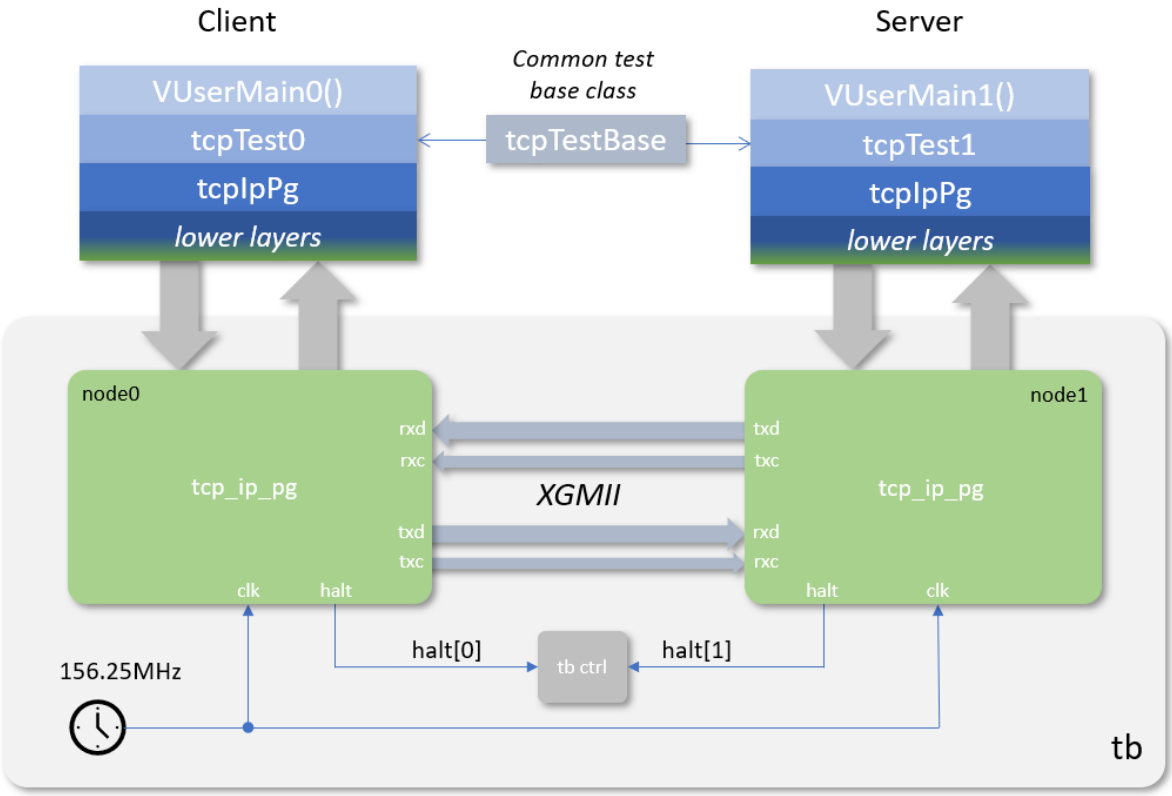
The tcpIpPg model has a both a Verilog and VHDL component with an XGMII interface, a clock input and a 'halt' output that software can set to indicate externally that the program has finished. The TCP/IP model is implemented in a class (`tcpIpPg`) with a very simple API. The constructor arguments define the source IPv4 address, MAC address and TCP port numbers that the instance will use (as well as the VProc node number). A `genTcpIpPkt` method allows the generation and transmission of data in a TCP packet. For receiving packets, a callback function can be registered with `registerUsrRxCbFunc`, and this user function is called whenever a data packet is received. More details can, of course, be found in the manual. A typical use case scenario is shown in the diagram below, showing the component instantiation and various software layers:



Note that the `tcpIpPg` class does not include connection establishment functionality, which needs to be implemented on top of the `tcpIpPg` class. However, an example test environment is provided in the repository, and this does implement a simplified connection state machine. We will briefly look at this example environment next.

Example Test Environment

An example test environment has been constructed as part of tcpIpPg which instantiates two tcpIpPg components, one as a client and one as a server, using VProc node's 0 and 1 respectively. A `tcpTestBase` class provides a means to interface with the simulation to 'sleep forever' (if finished but not halting) and to 'halt simulator'. A basic receiver callback function is also provided which prints output about received packets (via `tcpPrintPkt`) and places data into a queue (which is accessible from classes that inherit the `tcpTestClass`). The diagram below shows the setup of the test environment.



The two `VUserMain` VProc entry points just call the two test classes (appropriate to their node), though `VUserMain0` will request to halt the simulator when complete. The two test classes, `tcpTest0` and `tcpTest1` act as client and server respectively. A simple connection state machine is implemented with `tcpConnect` (though not exhaustive) and is used by both test classes.

When the simulation is run, the two models will connect (with `tcpTest0` the initiator) and an initial data transfer takes place. The client will then send more data from its test code. In this test, the data is in the form of strings and this is printed to the screen. When the client is finished, the connection is closed and the simulation halted.

A fragment of the displayed output is shown in the diagram below, showing *received* traffic from the start of connection establishment and first data transfer:

```
#
# Node1: Source MAC Addr..........: D8-9E-F3-88-7E-C3
# Node1: Source IPv4 Addr..........: 192.168.25.08
# Node1: Source TCP port...........: 0x0400
# Node1: Source sequence#..........: 0x000000c3 (   0 relative)
# Node1: Source ACK#...............: 0x000002ff (   0 relative)
# Node1: Source Window Size........: 32768
# Node1: Payload Length............: 0
# Node1: Flags.....................: SYN
#
# Node0: Source MAC Addr..........: 90-32-4B-07-0B-D1
# Node0: Source IPv4 Addr..........: 192.168.152.01
# Node0: Source TCP port...........: 0x0400
# Node0: Source sequence#..........: 0x000002ff (   0 relative)
# Node0: Source ACK#...............: 0x000000c4 (   1 relative)
# Node0: Source Window Size........: 32768
# Node0: Payload Length............: 0
# Node0: Flags.....................: SYN ACK
#
# Node1: Source MAC Addr..........: D8-9E-F3-88-7E-C3
# Node1: Source IPv4 Addr..........: 192.168.25.08
# Node1: Source TCP port...........: 0x0400
# Node1: Source sequence#..........: 0x000000c4 (   1 relative)
# Node1: Source ACK#...............: 0x00000300 (   1 relative)
# Node1: Source Window Size........: 32768
# Node1: Payload Length............: 27
# Node1: Flags.....................: ACK
#
# Node1: *** Hello from node 0 ***
#
# Node0: Source MAC Addr..........: 90-32-4B-07-0B-D1
# Node0: Source IPv4 Addr..........: 192.168.152.01
# Node0: Source TCP port...........: 0x0400
# Node0: Source sequence#..........: 0x00000300 (   1 relative)
# Node0: Source ACK#...............: 0x000000df (  28 relative)
# Node0: Source Window Size........: 32768
# Node0: Payload Length............: 0
# Node0: Flags.....................: ACK
#
```

Conclusions

We have looked at ethernet protocols, in some detail, from the perspective of 10Gbe and TCP/IPv4 as case studies, and from a logic and system implementer's point of view. We have seen the PHY, MAC, IP and TCP IP components as a typical system, and looked at the XGMII interface, with optional use of XGXS components implementing XAUI to extend the reach of XGMII if required. Finally, we briefly looked at a provided working model, `tcpIpPg`, which attempts to bring all these concepts together in a logic simulation for further exploration.

There is way more to ethernet, networking, transport protocols and the internet than covered here, and even covering the first few layers (from the bottom), we still had to restrict ourselves to single case study of 10Gbe and TCP/IPv4. None-the-less, I hope that this is a steppingstone for understanding other currently used systems, both

older and slower, like 1Gbe, and future faster systems and for the higher layers above.