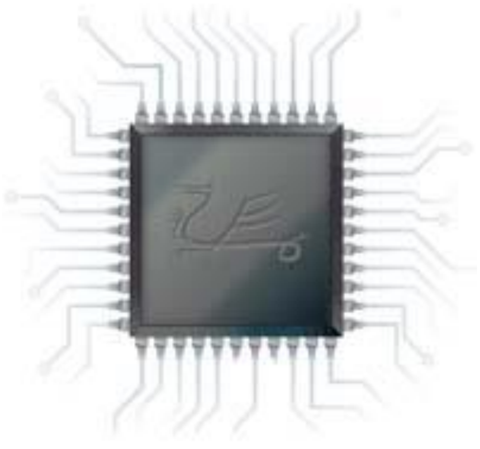
Reference Manual for the *tcpIpPg* 10GbE TCP/IPv4 XGMII packet generator

(version 1.0.3)



Simon Southwell

August 2021

(last updated 27th July 2024)

Simon Southwell (simon@anita-simulators.org.uk)

Cambridge, UK, January 2024

Contents

Introduction

The `tcp_ip_pg` project is a set of verification IP for generating and receiving 10GbE TCP/IPv4 Ethernet packets over an XGMII interface in a Verilog test environment. The generation environment is a set of C++ classes, to generate packets into a buffer and then send that buffer over the Verilog XGMII interface, and to simultaneously receive packets sent from a remote connection. The connection between the Verilog and the C++ domain is done using the Virtual Processor, VProc [1]—a piece of VIP that allows C and C++ code, compiled for the local machine, to run and access the Verilog simulation environment, and VProc is freely available on github.

The intent for this packet generator is to allow ease of test vector generation when verifying 10G Ethernet logic IP, such as a MAC, and/or a server or client for TCP and IPv4 protocols. The bulk of the functionality is defined in the provided C++ classes, making it easily extensible to allow support for other protocols such as UDP and IPv6. It is also meant to allow exploration of how these protocols function, as an educational vehicle.

An example test environment is provided, for ModelSim, with two packet generators instantiated, connected to one another—one acting as a client and one acting as a server. Connection establishment and disconnection software is provided in the test code to illustrate how packet generation is done, and how to easily build up mode complex and useful patterns of packets. Formatted output of received packets can be displayed during the simulation. Support for Questa and Icarus Verilog is also provided.

Features

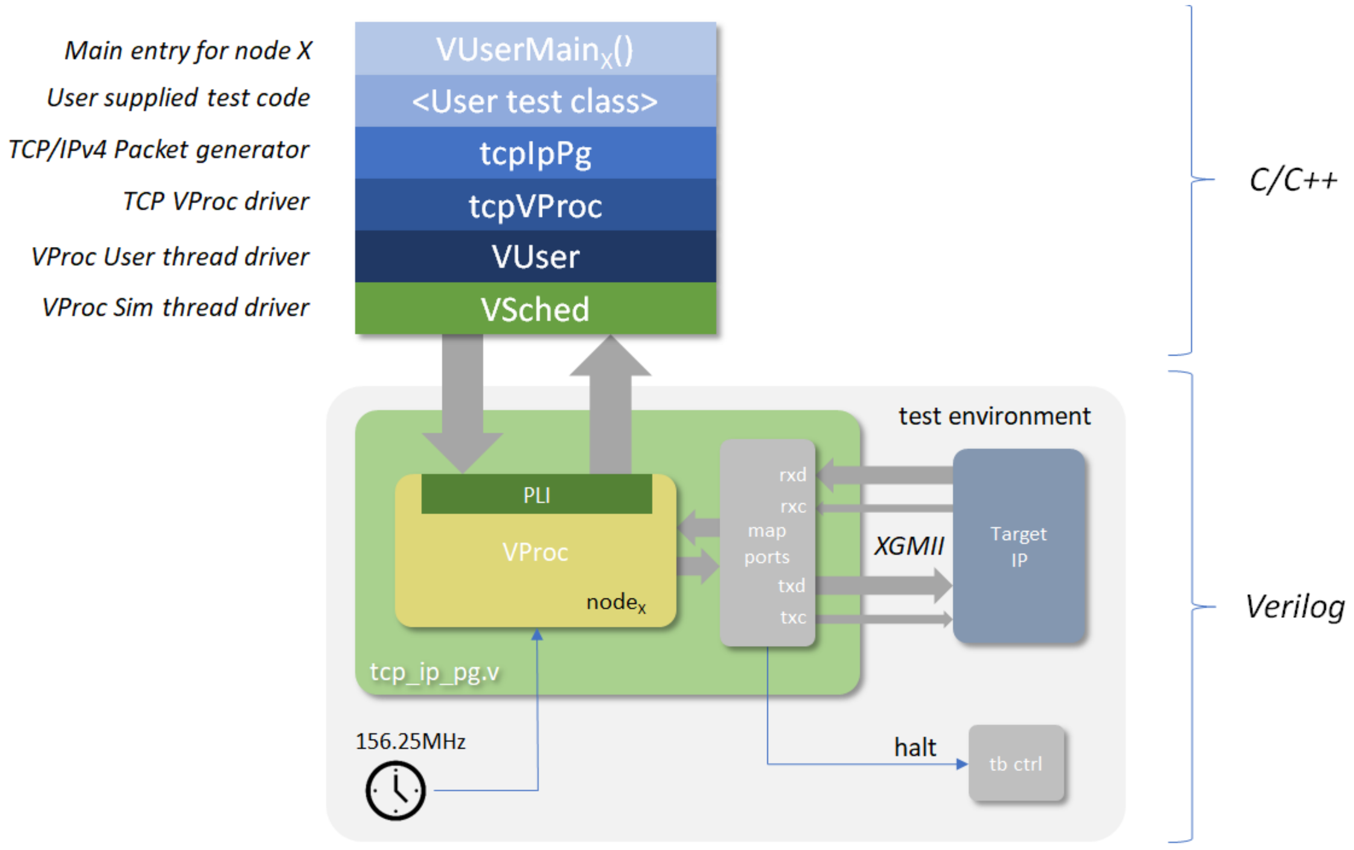
The basic functionality provided is as listed below:

- A Verilog module `tcp_ip_pg`
  - Clock input, nominally 156.25MHz ($10\times10^9 \div 64$)
  - XGMII interface, with TX and RX data and control ports
  - A `halt` output for use in test bench control
- A class to generate a TCP/IPv4 packet into a buffer
- A class to send a generated packet over the XGMII interface
- *A means to receive TCP/IPv4 packets over the XGMII interface and buffer them*
- A means to display, in a formatted manner, received packets
- Connection state machine not part of the packet generation class, but examples provided as part of the test environment (not complete).
- A means to request a halt of the simulation (when no more test data to send)
- A means to read a clock tick counter from the software

Architecture

The `tcp_ip_pg` verification IP is split across two domains; namely the Verilog HDL domain and the C/C++ domain. The interface between these two domains is via the Virtual Processor (VProc) using the PLI interface of Verilog—which may be configured for either 1.0 (tf interface) or 2.0 (VPI). The details of VProc will not be discussed at length in this document, as the provided `tcp_ip_pg` logic and software hides most of this away, and more details can be found in the VProc manual [1]. Suffice it to say that in the Verilog domain a generic processing element is instantiated, allowing the reading, and writing of data over a memory-mapped bus, and in the C/C++ domain, and API is provided to read and write over this bus.

The diagram below shows the complete stack for the `tcp_ip_pg` component, with the VProc logic wrapped in a Verilog module, and then the VProc interface software providing access to a specific TCP driver, which provides an API to the packet generation software. The user test code is written on top of this, using the packet generation class to build and then send data. It also provides a means for receiving packets and presenting them to the test routines. A VProc entry point, `VUserMain`*X*, replaces `main()` (much like `WinMain` or `DllMain` for Windows GUI and dynamic linked library software), with *X* being the VProc node number.



The Verilog Environment

The Verilog module is defined in `tcp_ip_pg.v` in the `verilog/` folder. It has clock input (`clk`) which nominally needs to be 156.25MHz to meet the 10GbE standards. It also has a `halt` output, which can be controlled by test software to indicate a request to end the simulation when all tests have completed.

The only other ports are the transmitter and receiver data and control. These are XGMII signals, with 64-bit data busses and 8-bit control busses for each direction (`txd`/`txc` and `rxd`/`rxc`). The busses are organised with the control and data bits on separate ports for flexibility, though some interfaces may have a single 72-bit port, with 8 bits of data then the corresponding control bit and so on for all byte lanes. Since this is just wiring, this is left to be done externally.

The `tcp_ip_pg` model has a single parameter, `NODE`, specifying a node number which is passed on to the VProc component. This defaults to 0, but if multiple `tcp_ip_pg` components are instantiated, or any other components with a VProc module, each must have a unique node number. The node number determines the `VUserMain`*X* entry point, and software accessing the virtual processors use this number to address the correct processor.

This is all that is required in the Verilog domain. The XGMII interface can now be used to connect to, say, a 10GbE MAC with an XGMII interface, or perhaps a XAUI PHY.

The VHDL Environment

An equivalent VHDL model is also provided for those wishing to use that HLD language, with the model found in the directory `vhdl/tcp_ip_pg.vhd`. In every other respect its use is identical to the Verilog model, with matching generics for parameters and matching ports.

The Software Stack

The packet generation software can be found in the `src/` folder of the repository.

VProc layers

The software layer starts with the API provided by VProc, which has a set of software attached to the PLI (`VSched`) and running in the same thread as the simulator. This communicates with a code running in a 'user' thread (`VUser`) that provides an API to the application. Fortunately, the details of this are hidden from the higher layers via and interface class called `tcpVProc`. This gets inherited by the packet generator class to provide a low-level TCP/IP oriented API. The following lists the prototypes for the public methods:

- `tcpVProc(int node)` : the constructor with the VProc node number specified

- `uint32_t TcpVpSendIdle(uint32_t ticks)` : A method to 'pause' for a given number of cycles

- `uint32_t TcpVpSendRawEthFrame(uint32_t* frame, uint32_t len)` : A method to send a constructed packet out over the transmission interface.

- `void TcpVpSetHalt(uint32_t val)` : A method to set or clear the halt port.

As mentioned above, this will be inherited by the packet generator class described later. This effectively means this API is available in that class directly. The separation of the access to the VProc component and the generation of the TCP/IP packets is done to allow ease of reuse in other environments. The higher layer software could interface to a different simulation or modelling environment by simply replacing this access layer, which is agnostic to the details of the packet data protocol. Or, conversely, this access layer can be inherited by another protocol packet generator to allow that to be used with the `tcp_ip_pg` Verilog component.

The `TcpVpSendIdle()` method will yield control back to the simulator for the specified number of ticks. It is crucial that, when not sending packets out over the XGMII interface that this method is used in the intervening time, for two reasons. Firstly, all VProc programs execute whilst the simulation has halted, waiting for new access calls from the program. When sending a packet, this is fine as time advances whilst the data is sent over the interface. But if n packet is being transmitted, time won't advance automatically, and the `TcpVpSendIdle()` method allows time to advance in these periods. Think of it, from a simulation point of view, as the software runs infinitely fast and simulation time stands still. So a `while(1);` loop would hang the simulation. The second reason is that, when time does advance, the code is monitoring for received packet data. Again, when sending a frame, the receive inputs are monitored as data is transmitted. When no data is transmitted, the idle methods allows the receive ports to still be monitored for arrive packets.

The `TcpVpSendRawFrame()` method is the means to send the pre-constructed TCP/IP packet out over the XGMII interface. It takes a pointer to the frame buffer as an argument, and a length for the entire packet. It is expected that the frame buffer has already had the packet data placed in it before calling the method (see below).

Finally the `TcpVpSetHalt()` method sets or clears the value of the `halt` output pin, based on the value of the arguments 0th bit. Note that one would not normally need to clear the halt bit, as setting it was a request to finish the simulation.

Packet Generator Class

The packet generator class, `tcpIpPg`, is where the TCP/IPv4 packets are generated. Without the `tcpVProc` access class, it just generated packets, optionally with payloads, into a local buffer. Therefore, it could be used as a standalone class in, say, a C++ model or in a SystemC environment, or whatever.

It is responsible for not only constructing the packets, but also to provide a means, where desired, for receiving packets and presenting them to the higher layer test software for processing. This is achieved by allowing the higher layer test code to register a callback function to be called whenever a packet is (correctly) received, passing in the packet information and payload (if present).

The public API presented by the `tcpIpPg` class (in addition to the inherited `tcpVProc` class methods) is as follows:

- `tcpIpPg (uint32_t node, uint32_t ipv4Addr, uint64_t macAddr, uint32_t tcpPort)` : the constructor specifying a unique node, an IPv4 address for the node, a MAC address for the node and a connection TCP port number to use.

- `uint32_t genTcpIpPkt (tcpConfig_t &cfg, uint32_t* frm_buf, uint32_t* payload, uint32_t payload_len)` : A method to construct a TCP/IPv4 packet suitable for transmission on the `tcp_ip_pg` XGMII interface.

- `void registerUsrRxCbFunc (pUsrRxCbFunc_t pFunc, void* hdl)` : a method for registering a callback function on reception of a validated packet.

Construction

The constructor generates a packet generator object and requires some parameters. The node number must be unique and match the node number specified for the `tcp_ip_pg` Verilog component parameter. An IPv4 address is specified for the packet generator along with a MAC address. These must be different from any other components' addresses in the test environment. Finally, a TCP port number is specified. Currently only one TCP connection is supported. It may be possible to instantiate multiple tcpIpPg objects, each with different port numbers (or even different IP and MAC addresses), addressing to the same `tcp_ip_pg` node, but this has not been tested.

Packet Generation

To generate a TCP/IPv4 packet `genTcpIpPkt` is called. It takes a configuration input, defined as a structure in the `tcpIpPg` class and a pointer to a buffer to place the frame data. This buffer must be large enough to contain the data. Since the maximum transmit unit is 1500 bytes, and overhead data 18 bytes, then 2048 is sufficient. However, to accommodate control bits, the data type of the required buffer is `uint32_t`, with one 'byte' occupying one 32 bit value. So, the buffer must be > 1518 `uint32_t` in size.

If a data payload is required then a pointer to a payload buffer is passed in, with a specified payload length. This, too, must be an array of `uint32_t`, with a byte occupying a single entry. If no payload is required, then the payload pointer may be NULL, but the `payload_len` value must be 0 in this case.

The configuration structure, `tcpConfig_t`, passed in to `genTcpIpPkt` is defined within the class as follows:

```
    typedef class {
    public:
        uint32_t dst_port;
        uint32_t seq_num;
        uint32_t ack_num;
        bool     ack;
        bool     rst_conn;
        bool     sync_seq;
        bool     finish;
        uint32_t win_size;
        uint32_t ip_dst_addr;
        uint64_t mac_dst_addr;
    } tcpConfig_t;
```

The configuration specifies all that's required to send out a packet. Firstly the port number to be used (or being used) is specified. This would normally be the same as the port number specified at construction, but need not be (if, say, testing for rejection of a non-open port).

The sequence number and ACK number are the values to be transmitted. These are not managed by the `tcpIpPg` class, and it is expected that this is done as part of the test code—again, potentially to send 'invalid' values to test certain cases. The provided test bench example does manage this values to a certain degree as an example (see later).

The four Boolean values are the flags that can be set in the packet. Currently these concentrate on connection and disconnection, with other flags not yet supported. The class allows invalid combinations (to test), but correct usage in the test code allows connection, transmission, and disconnection protocols to be followed.

The windows size parameter is used to advertise space available and thus flow control the reception of data. As the model, if data received correctly via the callback function, will not run out of space, this may be fixed at some suitable value. If flow control testing is required, though, then this may be changed according to needs.

The packet generation needs foreknowledge of the destination IP and MAC addresses to communicate with a remote piece of IP, and this are provided in the last two fields. Once more, this may be set to any value if testing of invalid/missing addresses is required.

Packet Reception

Packet reception, if required, is provided to the user code via the means of a callback function. This registered function must be of type `pUsrRxCbFunc_t`, that is, have a prototype of:

```
    void cbFunc(rxInfo_t rx_info, void *hdl)
```

To register this function, `registerUsrRxCbFunc()` is used, passing in the function pointer and a void pointer 'handle'. This handle will be passed into the callback function when called, and can be a pointer to anything, or even NULL. The intended use is to allow the received packet to be processed before returning from the callback function. So it might be a pointer to a queue to push the packet on to, or a pointer to a function to process the data. In the example test bench (see below), it is a pointer to the test class (the 'this' pointer) in which the callback function is defined as a static method. This gives it access to the other public (non-static) methods and members of the class to which it belongs to help process the received packet. However it is defined, the received packet must be processed and/or saved before returning from the callback function, as the `tcpIpPg` buffers will be reused and the data lost. In addition, the callback cannot make calls to access the `tcpIpPg` methods that advance simulation time, as this becomes recursive. Instead, it is expected that the callback does the minimum to process the packet into a buffer or queue, which the main test program can then inspect and process.

The `rx_info` argument passed into the callback contains the received packet information. The `rxInfo_t` is defined as:

```
    typedef struct {
        uint64_t mac_src_addr;
        uint32_t ipv4_src_addr;
        uint32_t tcp_src_port;
        uint32_t tcp_seq_num;
        uint32_t tcp_ack_num;
        uint32_t tcp_flags;
        uint32_t tcp_win_size;
        uint8_t  rx_payload[ETH_MTU];
        uint32_t rx_len;
    } rxInfo_t;
```

This contains similar information as for the transmitted packet configuration structure, but for the packets sent from the remote device. Note that the `tcpIpPg` receiver code will reject packets that do not match the MAC and IP addresses, and the open TCP port number configured at construction, and will print a warning that such a packet was received, but it will not call the callback function, and discards the packet.

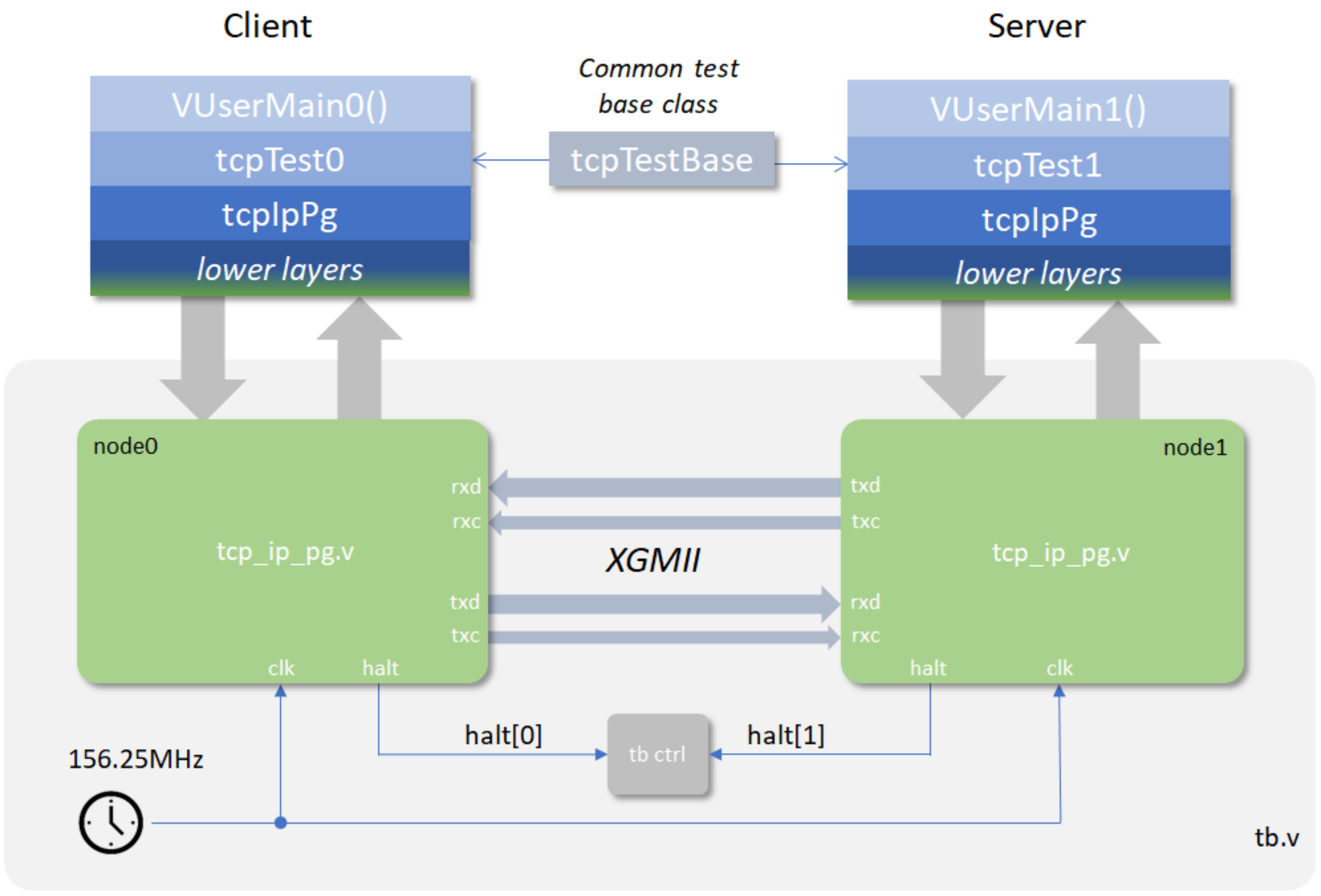
If a payload is present, the `rx_len` field indicates its size, and the `rx_payload` buffer has the data. Unlike transmission, as this is only user payload data, with no control, this buffer is a byte buffer, with one byte per array entry.

Test Bench

A test bench is provided in the repository to validate the model and to provide as an example of the usage of the packet generator—both logic and software.

Structure

The test bench consists of the instantiation of two `tcp_ip_pg` components, one acting as a client (configured as node 0) and one acting as a server (configured as node 1). The two components then have their XGMII interface connected together (tx to rx and vice versa), and a 156.25MHz clock routed to them. The halt outputs are routed to some test bench control logic to stop the simulation when a test is completed. The diagram below shows the test bench arrangement.



The test bench can be found in the `test/` folder of the repository, which is where the test bench verilog can be found (`tb.v`), along with the scripts and a `makefile` to allow compilation (with a `makefile.icarus` version for Icarus Verilog). There is a VHDL version of the `tcpIpPg` model provided, as mentioned previously, and this has an accompanying VHDL test bench. To compile these instead of the Verilog, an argument can be given to the make command—i.e., `make HDL=VHDL`. All the test software can be found in the `src/` folder, which is a sub-folder of the `test/` folder.

With reference to the Architecture section, the test bench provides user test code via two separate classes for the client (`tcpTest0`) and the server (`tcpTest1`).

The Test Base Class

The code between the two test classes differ in some respects, but much is common, and a `tcpTestBase` class is defined for this common code. This is a virtual base class with all the functionality defined, except a virtual method, `runTest()`, which must be provided by the two derived classes, where the specific functionality is implemented. The class inherits a utility class, `tcpPrintPkt`, which adds formatting capability for received packets. This functionality was separated out so that it might be easily reused in another context.

The constructor for the base class just requires a node number, which is passed on to the `tcpIpPg` class, and also used in the formatted output to identify the node.

A couple of utility methods are defined to aid simulation control. The `haltSim()` method is called when all testing has completed and the node wishes to end the simulation. It is effectively a request to halt to the underling test logic, via the `tcpIpPg` class. The simulation may not finish straight away, but control must be given back to the simulation when idle, and so a `sleepForever()` method is provided for calling as the last thing in a test, which will continue to tick the simulation indefinitely until the simulation is complete.

The base method provides a common receive callback method to be registered with the tcpIpPg class. In this method, a call is made to a method to display a received packet in a formatted manner to the simulation output console. The packet is then added to an internal vector queue, making it available to the `runTest()` code provided by the derived test class. T is expected that the test code will monitor, periodically, the status of the queue and process packets as and when it is able, popping each packet from the front of the queue when it’s not empty. This allows as many packets to be received as required, subject only to the flow control set by the test program.

Connection and Disconnection Utility Class

As part of the test software, a class is defined, `tcpConnect`, to implement connection and disconnection states. This follows the TCP protocols but is example code only and not considered complete. The `tcpTestBase` class has a `tcpConnect` object defined as a member variable for access to this functionality in the tests. The connection class has two pairs of methods; a pair for connection and a pair for disconnection.

To go through a connection cycle, the client uses the `initiateConnect()` method, whilst the server uses the `listenConnect()` method. For both methods a node number is required, along with a reference to a `tcpIpPg` object and to a vector of `rxInfo_t` types. This allows the class to create, send and receive packets during the connection process. Also both have initial window size and initial sequence number parameters, with the later defaulting to 0. The `initiateConnect` method also requires a destination TCP port number and destination addresses for the IPv4 and MAC.

For disconnection, things are similar to connection. The two methods provided by the class for disconnection are `initiateTermination` and `waitForTermination`. Like the other methods a node number, `tcpIpPg` object reference and a receive queue reference are required. Also, both require a window size and sequence number. The `initiateTermination` method also needs the TCP port number and the IP and MAC addresses of the destination. The `waitForTermination` method also takes a TCP port number as an 'open port' to specify the port number for the open connection and will reject disconnection for a non-matching port number. The `initiateTermination` method must also be passed the `ACK` value needed at the point the disconnection is started, based on the traffic so far transmitted and received. Finally, the `waitForTermination` method has a couple of Boolean flags. The first of these, `finAlreadyReceived`, flags whether, on entering the method a received packet already had a `FIN` flag set. This defaults to false, meaning the method will first wait for a packet with the `FIN` flag set, before continuing the disconnection protocol. If it is set to true, then the method assumes a `FIN` packet has been received (but not acknowledged) and will skip waiting for a packet with this flag. This allows a test to be written where either the order of the packets where a `FIN` packet's position is known in advance, or a test where a `FIN` packet can arrive at an arbitrary point.

Whilst waiting for a `FIN` packet, the `waitForTermination` method can still process received packets if the `processPkts` argument is set to true (default is false). In this case the code will process received packets and acknowledge them. The test bench can generate packets with data payloads, but these are (for this case) null terminated strings messages which can be printed to the simulation screen. Whilst wait for a `FIN` packet, any payload of a received packet will have the message printed to the screen (as well as the normal formatted header output).

The Test Code

The two test classes simply inherit the base class and provide the `runTest()` method to perform a test. As mentioned before, the node 0 code (`tcpTest0`) acts as a client, whilst the node 1 code (`tcpTest1`) acts as a server. The main difference being that the client will instigate a connection and close it down, whilst the server will only respond.

In the particular example test provided, the client `runTest()` code:

1. Creates a `tcpIpPg` object, providing the client IP and MAC addresses and a TCP port
2. It registers a receive callback function, using the method provide by the `testBaseClass`, and uses its 'this' pointer as the handle
3. It then initiates a connection using the `tcpConnect` method.
    a. The method sends a `SYN` packet
    b. waits for a `SYN+ACK` return
    c. `ACK`s this with a message payload

4. It then sends a normal packet with a payload
    a. The payload has a string message
    b. It then waits for an `ACK`
5. A check is made that the `ACK` has all packets acknowledged and then initiates a termination for the connection, using the `tcpConnect` object.

The server test follows a similar, but more passive flow:

1. Creates a `tcpIpPg` object, providing the server IP and MAC addresses and a TCP port
2. It registers a receive callback function, using the method provide by the `testBaseClass`, and uses its 'this' pointer as the handle
3. It then calls the `listenForConnect` method of the `tcpConnect` object.
    a. It waits for a first packet and checks it's a `SYN` packet
    b. It sends a `SYN+ACK` packet in return
    c. It then waits for an `ACK` packet back and processes any payload—i.e. it prints the string message.
4. It then calls `waitForTermination` using the `tcpConnect` object, with the `processPkts` argument set to true to allow any packets sent by the client to be processed before it sends a `FIN` packet.

At this point, this is the limit of what this code does. It is hoped that there is enough functionality implemented to demonstrate how to use the tcpIpPg software and elaborate more complex patterns of packets and more complex functionality.

Compiling and Running the Test Bench

The test bench provides various scripts and a make file to compile and run the code, with GUI or batch simulation runs. It makes assumptions about the tools installed and the prerequisites a defined in the next section. It is expected that these can be adapted for other simulation environments. The VProc [1] repository has many more scripts for other simulator support to act as examples.

Prerequisites

The provided `makefile` allows for compilation and execution of the example simulation test. As mentioned above it is only for ModelSim at this time and has some prerequisites before being able to be run. The following tools are expected to be installed:

- ModelSim Intel FPGA Starter Edition. Version used is 10.5b, bundled with Quartus Prime 18.1 Lite Edition. The `MODEL_TECH` environment variable must be set to point to the simulator binaries/libraries directory (e.g. `win32aloem` or `linuxaloem`)
- mingw64 and MSYS 2.0: For the tool chain and for the utility programs (e.g. `make`). The `PATH` environment variable is expected to include paths to these two tools' executables

- VProc: checked out to `vproc/` in the same directory as the `tcp_ip_pg` repository folder. See [1] in References section for link to github.

The versions mentioned above are those that have been used to verify the environment, but different versions will likely work just as well. If `MODEL_TECH` environment variable is set correctly, and MinGW and Msys2 `bin` folders added to the `PATH`, then the `makefile` should just use whichever versions are installed. Note that the test code must be compiled for 32 bit for ModelSim, which the simulation test `makefile` does. This is required for ModelSim PLI code and may be different for other simulators. So, if compiling PLI code independently of the VProc `makefile` then remember to add the `–m32` flag. For other supported simulators, these will be 64-bit applications with a 64-bit compilation.

Running the Test

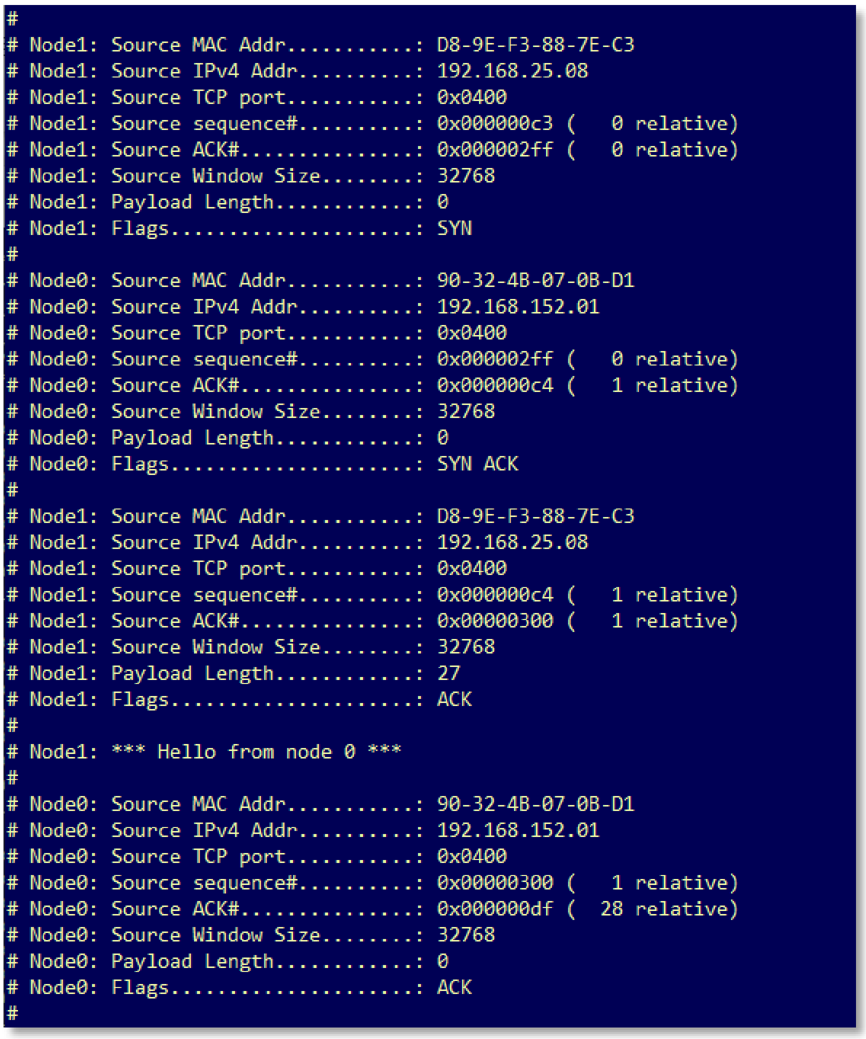
The entire body of code needs to be compiled to shared object (DLL, in Windows parlance), so that it may be loaded into the simulator. The VProc code, packet generator code and user code (`VUserMain`*X*, and associated test source) are catered for using a `makefile` in `test/`. The test makefile can both compile and run C/C++ source and also run the simulations. Typing `make help` displays the following usage message:

```
make help           Display this message
make                Build C/C++ code without running simulation
make compile        Build HDL code without running simulation
make run/sim        Build and run batch simulation
make rungui/gui     Build and run GUI simulation
make runlog/log     Build and run batch simulation with signal logging
make waves          Run wave view in free starter ModelSim (to view runlog signals)
make clean          clean previous build artefacts
```

Giving no arguments compiles the C/C++ code. The `compile` argument compiles the Verilog. The various execution modes are for batch, GUI, and batch with logging. The batch run with logging assumes there is a `wave.do` file (generated from the waveform viewer) that it processes into a `batch.do` with signal logging commands. When run, a `vsim.wlf` file is produced by the simulator, just as it would when running in GUI mode, with data for the logged signals. The `make wave` command can then be used to display the logged data without running a new simulation. Note that the `batch.do` generation is fairly simple at this point, and if the `wave.do` is constructed with complex mappings the translation may not work. The `make clean` command deletes all the artefacts from previous builds to ensure the next compilation starts from a fresh state.

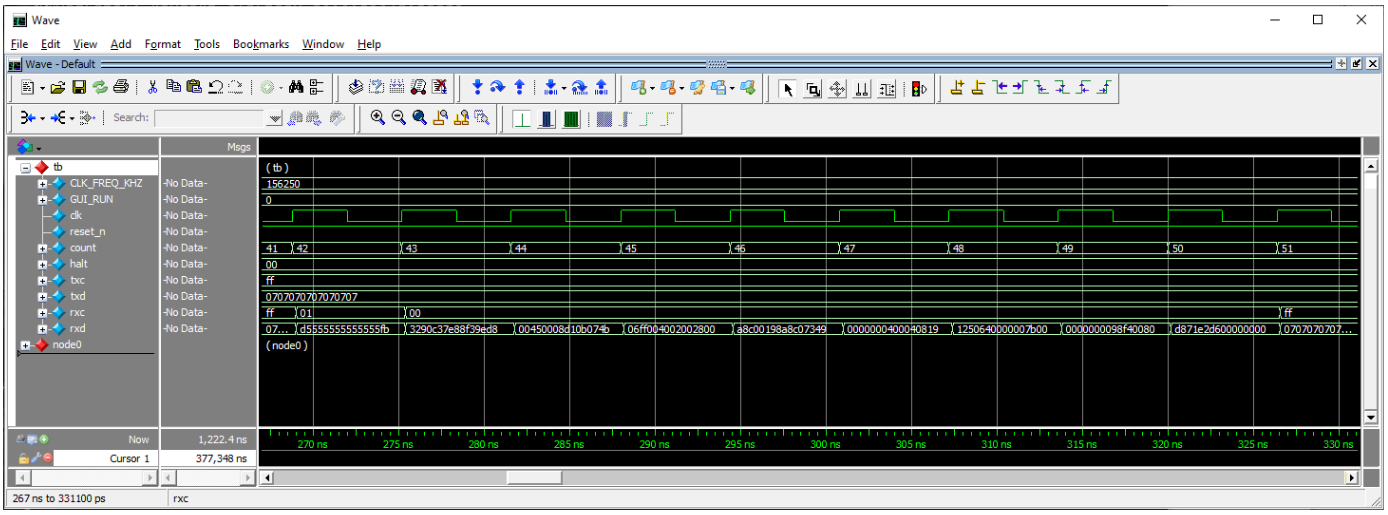
Output

When run, the test produces printed out of the received packets at each node, along with any string payload messages. The following diagram is a fragment of the output produced during simulation.



```
#
# 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
#
```

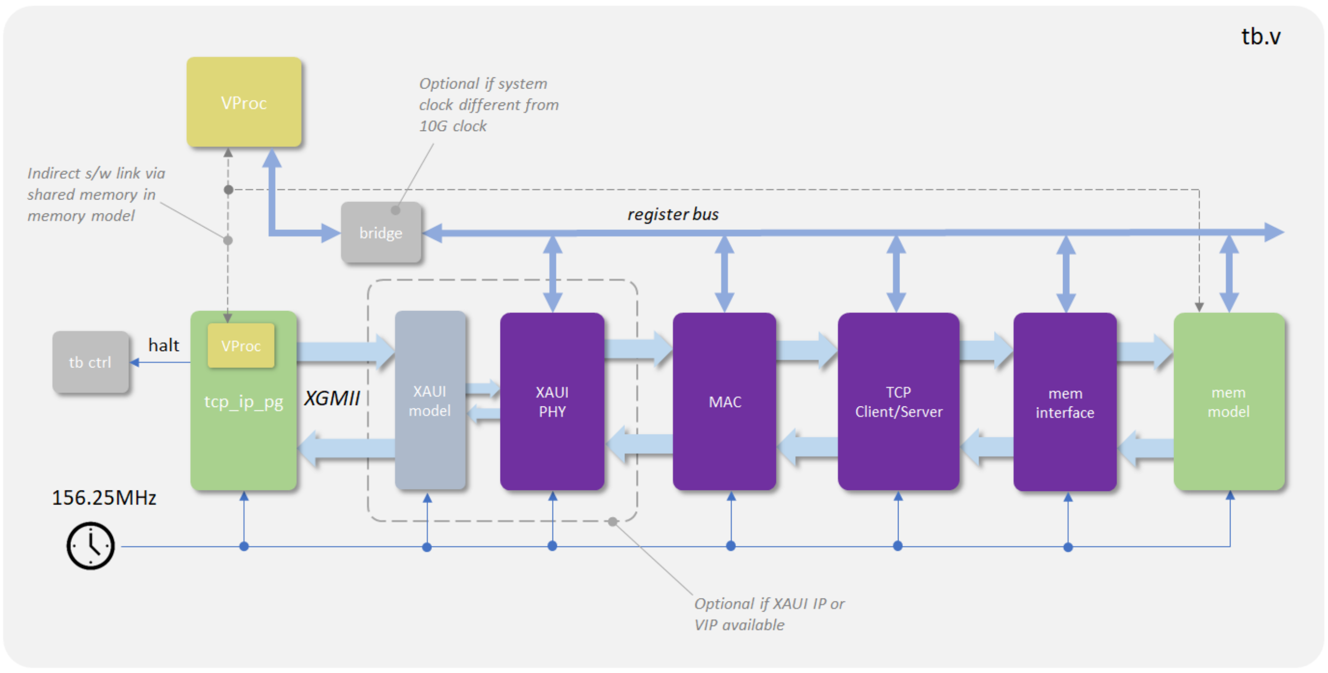
From a simulation point of view, the XGMII interface has the valid signals corresponding to the packets displayed on the console output, as shown below.



Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Search:
Msgs
tb
CLK_FREQ_KHZ -No Data- 156250
GUI_RUN -No Data- 0
clk -No Data-
reset_n -No Data-
count -No Data- 41 42 43 44 45 46 47 48 49 50 51
halt -No Data- 00
txc -No Data- ff
txd -No Data- 0707070707070707
rxc -No Data- ff 01 00 ff
rxd -No Data- 07... d5555555555555fb 3290c37e88f39ed8 00450008d10b074b 06ff004002002800 a8c00198a8c07349 0000000400040819 1250640000007b00 0000000098f40080 d871e2d600000000 070707070 7...
(tb)
(node0)
node0
Now 1,222.4 ns
Cursor 1 377,348 ns
270 ns 275 ns 280 ns 285 ns 290 ns 295 ns 300 ns 305 ns 310 ns 315 ns 320 ns 325 ns 330 ns
267 ns to 331100 ps
rxc

Usage

The test bench provided is just an example to illustrate the usage of the packet generator VIP. A more realistic scenario is where the VIP is used to test various other 10GbE IP. In this section will be briefly discussed an intended use case. The diagram below shows a test environment using the `tcp_ip_pg` component, connected to some 10GbE IP, along with a virtual processor acting as a processing element for configuration and control, and a memory model based on the wyvernSemi memory model VIP[2].


tb.v
VProc
Optional if system clock different from 10G clock
Indirect s/w link via shared memory in memory model
bridge
register bus
tb ctrl
halt
VProc
tcp_ip_pg
XGMII
XAUI model
XAUI PHY
MAC
TCP Client/Server
mem interface
mem model
156.25MHz
Optional if XAUI IP or VIP available

In this use case the IP under test is a 10GbE data path chain with (optionally) a XAUI PHY block, a MAC, a TCP client or server and a memory interface to read and write payload data (possibly with an interrupt to indicate packet reception). The memory model serves as the buffer space for the TX and RX payload and has the advantage that the programs running on the virtual processors (including that on the `tcp_ip_pg` block) can have direct software access to the memory via an API (see [2]) if required.

If the path being tested does have a XAUI PHY, then an external XAUI must be instantiated between the `tcp_ip_pg` XGMII interface and the IP under test. Possibly this can be another instantiation of the XAUI IP being tested, or some other model. If not, the XGMII interface might connect directly to the MAC.

In this scenario, the VProc component acts as the system processor, and is intended to control the test flow. As the test program running on the `tcp_ip_pg` is also a VProc program, and because it can have direct access to the memory, communication can be established between the processor program and the packet generator program via a shared memory interface. Note that the programs on the two virtual processors are running in separate threads but, due to the nature of the VProc design, only one thread can be active at any given time. Therefore, exchanging data through shared memory requires no additional control to make thread safe. It is envisaged that the processor software and the `tcp_ip_pg` software would co-ordinate to keep their programs in synchronisation, exchanging status information, and also that, if required, the `tcp_ip_pg` software can request updates to system registers by the processor as it has no access itself.

A likely scenario is that the processor will be running on a different clock to that of the 10GbE logic. The test environment may have other pieces of IP that need to be controlled by the processor software running on a system clock. In this case a bridge would be required for register access across the two domains, but this is likely to be a piece of IP already available or being developed and tested as part of the system implementation. Different clock domains do not affect the direct memory access API as this is independent of the simulation environment.

References

[1] *Virtual Processor (VProc)*, https://github.com/wyvernSemi/vproc, Simon Southwell, June 2010

[2] *Reference Manual for the Verilog Memory Model Simulation Component*, https://github.com/wyvernSemi/mem_model , Simon Southwell, August 2021