Reference Manual for the *udpIpPg* 1GbE UDP/IPv4 GMII packet generator

(version 1.0.4)

Simon Southwell

December 2025

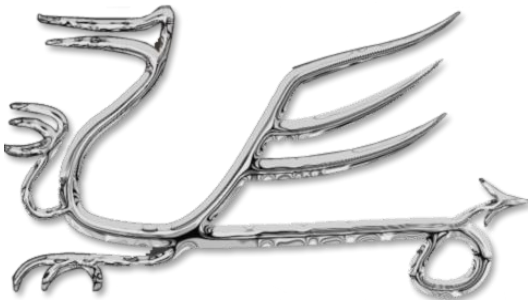
(last updated 8th July 2025)

Copyright

Disclaimers

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

Cambridge, UK, January 2025



Contents

Introduction

The `udp_ip_pg` project is a set of verification IP for generating and receiving 1GbE UDP/IPv4 Ethernet packets over a GMII 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 GMII interface, and to simultaneously receive packets sent from a remote device. 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 1G Ethernet logic IP, such as a MAC, and/or a server or client for UDP and IPv4 protocols. The bulk of the functionality is defined in the provided C++ classes, making it 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 Vivado and Verilator, with two packet generators instantiated, connected to one another—one acting as a client and one acting as a server. Test software is provided in the provided 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.

Features

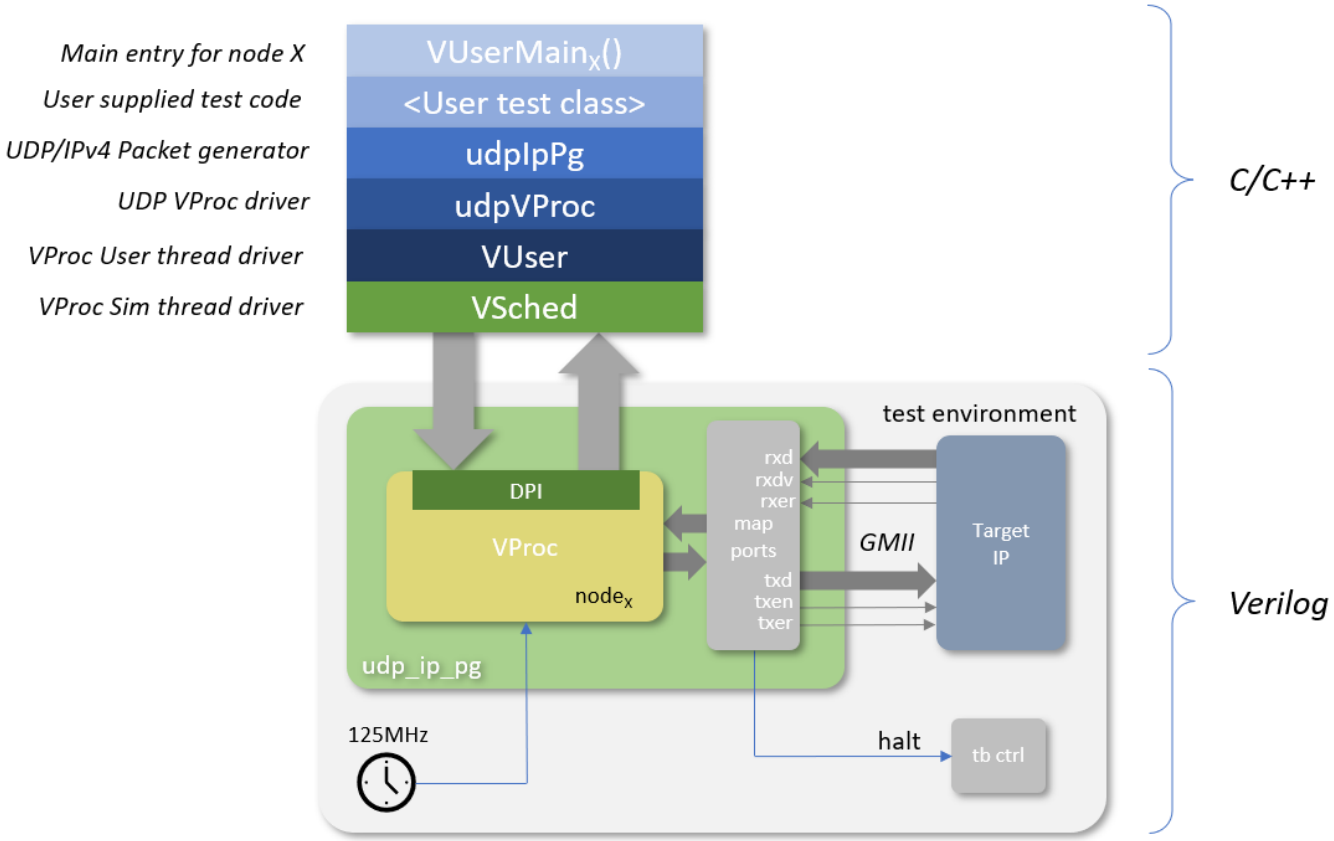
The basic functionality provided is as listed below:

- A Verilog module `tudp_ip_pg`
  - Clock input, nominally 125MHz ($1\times10^9 \div 8$)
  - GMII interface, with TX and RX data and control signals
    - A GMII/RGMII convertor to support RGMII MACs
  - A `halt` output for use in test bench control
- A class to generate a UDP/IPv4 packet into a buffer
- A class to send a generated packet over the GMII interface
- A means to receive UDP/IPv4 packets over the GMII interface and buffer them
- A means to display, in a formatted manner, received packets
- 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 udp_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 DPI interface of SystemVerilog. The details of VProc will not be discussed at length in this document, as the provided udp_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 udp_ip_pg component, with the VProc logic wrapped in a Verilog module, and then the VProc interface software providing access to a specific UDP 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, `VUserMainX`, 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 `udp_ip_pg.v` in the `verilog/` folder. It has clock input (`clk`) which nominally needs to be 125MHz to meet the 1GbE 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 GMII signals, with 8-bit data busses and two control bits for each direction (`txd`, `txen`, `txer` and `rxd`, `rxdv`, `rxer`). The busses are organised with the control and data bits on separate ports for flexibility, though some interfaces may have a single 10-bit port, with 8 bits of data then the corresponding control bits.

The `udp_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 `udp_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 GMII interface can now be used to connect to, say, a 1GbE MAC with an GMII interface.

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 DPI (`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 `udpVProc`. This gets inherited by the packet generator class to provide a low-level UDP/IP oriented API. The following lists the prototypes for the public methods:

- `udpVProc(int node)` : the constructor with the VProc node number specified
- `uint32_t UdpVpSendIdle(uint32_t ticks)` : A method to 'pause' for a given number of cycles
- `uint32_t UdpVpSendRawEthFrame(uint32_t* frame, uint32_t len)` : A method to send a constructed packet out over the transmission interface.
- `void UdpVpSetHalt(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 UDP/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 `udp_ip_pg` Verilog component.

The `UdpVpSendIdle()` 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 GMII 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 no packet is being transmitted, time won't advance automatically, and the `UdpVpSendIdle()` 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 `UdpVpSendRawFrame()` method is the means to send the pre-constructed UDP/IP packet out over the GMII 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 `UdpVpSetHalt()` method sets or clears the value of the `halt` output pin, based on the value of the arguments 0^{th} 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, `udpIpPg`, is where the UDP/IPv4 packets are generated. Without the `udpVProc` 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 `udpIpPg` class (in addition to the inherited `udpVProc` class methods) is as follows:

- `udpIpPg (uint32_t node, uint32_t ipv4Addr, uint64_t macAddr, uint32_t udpPort)` : the constructor specifying a unique node, an IPv4 address for the node, a MAC address for the node and a source UDP port number to use.

- `uint32_t genUdpIpPkt (udpConfig_t &cfg, uint32_t* frm_buf, uint32_t* payload, uint32_t payload_len)` : A method to construct a UDP/IPv4 packet suitable for transmission on the `udp_ip_pg` GMII 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 `udp_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 UDP port number is specified.

Packet Generation

To generate a UDP/IPv4 packet `genUdpIpPkt` is called. It takes a configuration input, defined as a structure in the `udpIpPg` 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 of 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, `udpConfig_t`, passed in to `genUdpIpPkt` is defined within the class as follows:

```
typedef class {
public:
    uint32_t dst_port;
    uint32_t ip_dst_addr;
    uint64_t mac_dst_addr;
} udpConfig_t;
```

The configuration specifies all that's required to send out a packet for UDP. Firstly the destination port number to be used (or being used) is specified.

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 if 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 `udpIpPg` buffers will be reused and the data lost. In addition, the callback cannot make calls to access the `udpIpPg` 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 udp_src_port;
    uint32_t udp_dst_port;
    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 `udpIpPg` receiver code will reject packets that do not match the MAC and IP addresses and will print a warning that such a packet was received, but it will not call the callback function, and discards the packet.

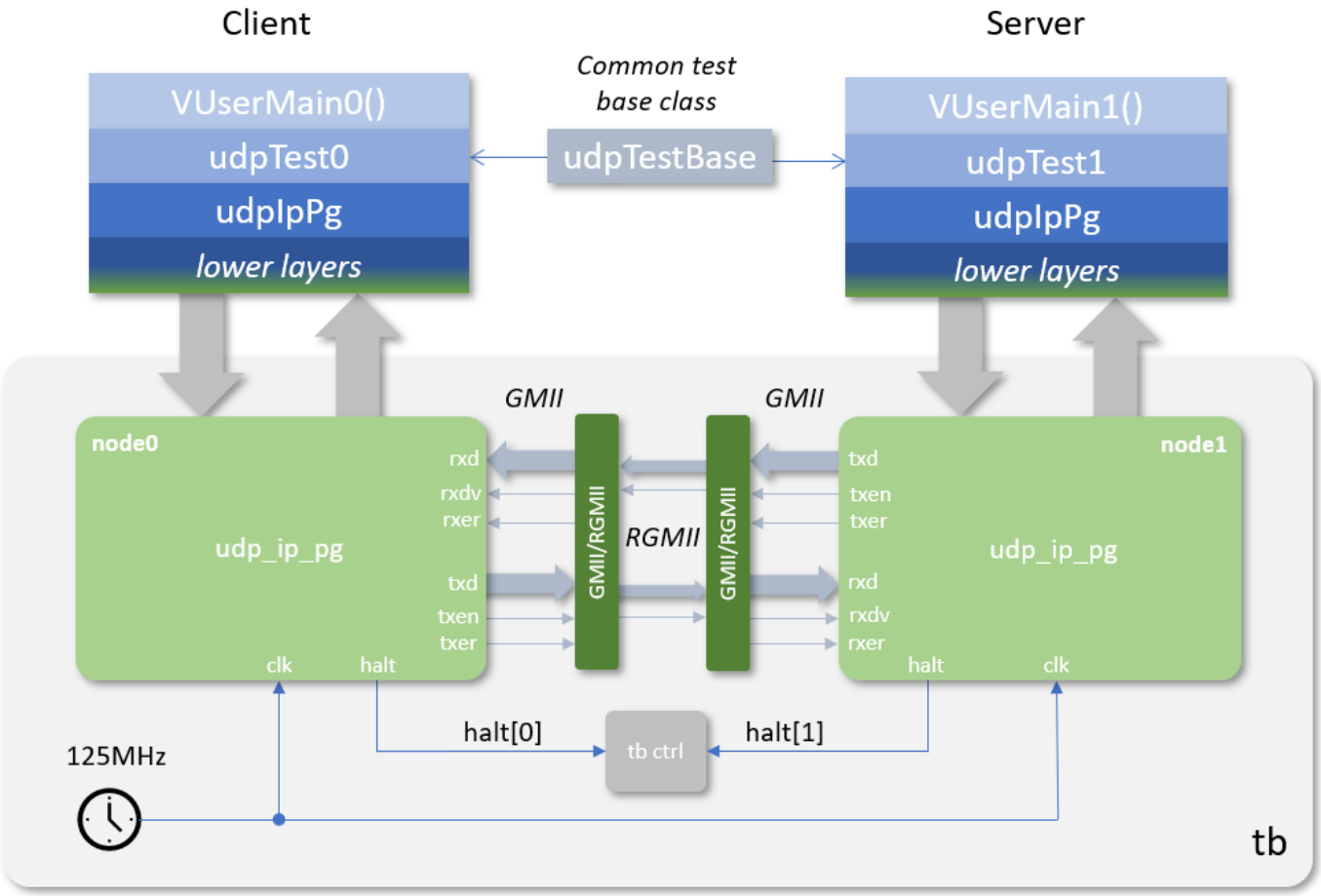
If a payload is present, the `rx_len` field indicates its size, and the `rx_payload` buffer has the data. Unlike transmission, as this is only user payload data, with no control, this buffer is a byte buffer, with one byte per array entry.

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 `udp_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 GMII interface connected together (tx to rx and vice versa) via two GMII/RGMII convertor modules to validate support for the RGMII interface, along with a 125MHz 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.verilator` to allow compilation on the Verilator simulator (with a `makefile.vivado` version for Vivado xsim).

With reference to the Architecture section, the test bench provides user test code via two separate classes for the client (`udpTest0`) and the server (`udpTest1`).

The Test Base Class

The code between the two test classes differ in some respects, but much is common, and a `udpTestBase` 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, `udpPrintPkt`, 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 `udpIpPg` class, and also used in the formatted output to identify the node.

The base method provides a common receive callback method to be registered with the `udpIpPg` 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. It 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.

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 (`udpTest0`) acts as a client, whilst the node 1 code (`udpTest1`) 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 `udpIpPg` object, providing the client IP and MAC addresses and a UDP 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 sends a couple of packets with payload
    a. The payload has a string message

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

1. Creates a `udpIpPg` object, providing the server IP and MAC addresses and a UDP 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 loops waiting for the receive queue to have a packet, calling the `UdpSendIdle()` method when none to process

4. When a packet is in the queue, it pops it out, deletes the packet from the queue and prints the message.

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 `udpIpPg` 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 make files 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.vivado` and `makefile.verilator` allows for compilation and execution of the example simulation test. It has some prerequisites before being able to be run. The following tools are expected to be installed beyond the logic simulator:

- 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 `udp_ip_pg` repository folder. See [1] in References section for link to github.

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++ and HDL code without running simulation
make sim           Build and run command line interactive (sim not started)
make run           Build and run batch simulation
make rungui/gui    Build and run GUI simulation (sim not started)
make clean         clean previous build artefacts
```

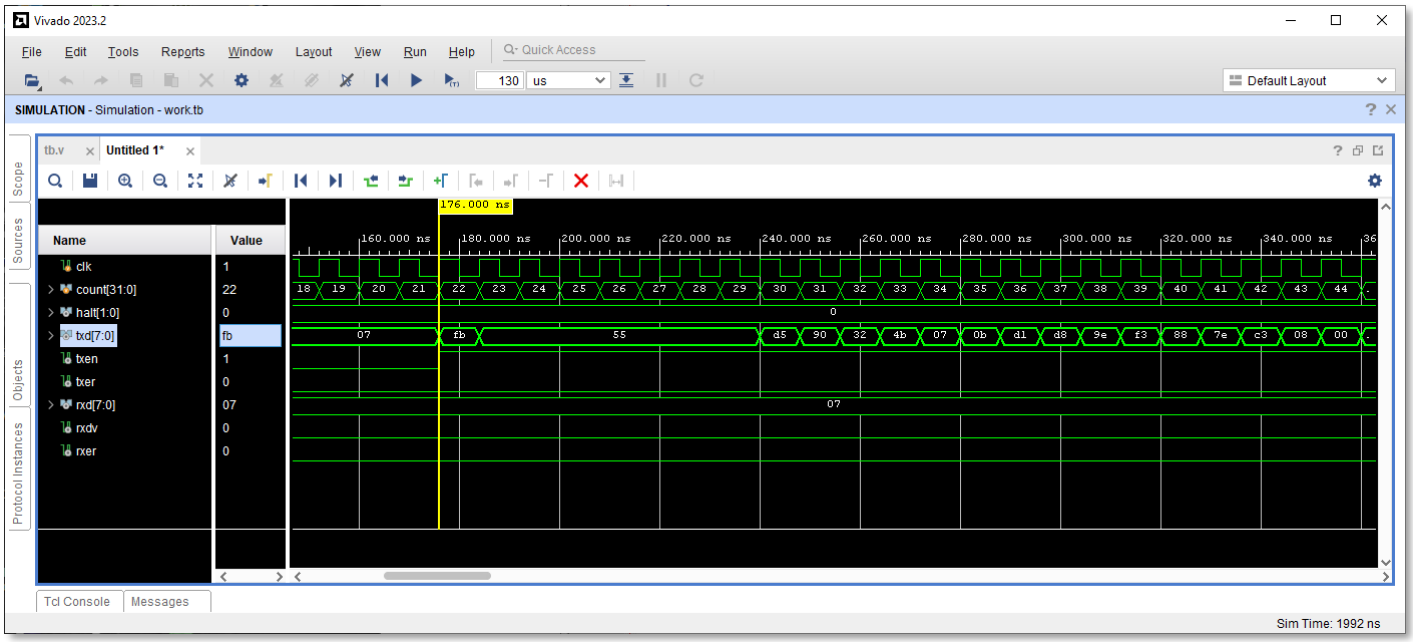
Giving no arguments compiles the C/C++ code. The `compile` argument compiles the Verilog. The various execution modes are for batch and GUI. The `make clean` command deletes all the artefacts from previous builds to ensure the next compilation starts from a fresh state.

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.

```
source xsim.dir/work.tb/xsim_script.tcl
# xsim {work.tb} -autoloadwcfg -runall
Time resolution is 1 ps
run -all
VInit(0): initialising DPI-C interface
  VProc version 1.12.3. Copyright (c) 2004-2024 Simon Southwell.
VInit(1): initialising DPI-C interface
  VProc version 1.12.3. Copyright (c) 2004-2024 Simon Southwell.

****************************
*   Wyvern Semiconductors   *
* Virtual Processor (VProc) *
*    udp_ip_pg (node 1)     *
*    Copyright (c) 2025     *
****************************


****************************
*   Wyvern Semiconductors   *
* Virtual Processor (VProc) *
*    udp_ip_pg (node 0)     *
*    Copyright (c) 2025     *
****************************


udpIpPg version 1.0.0

Node1: Source MAC Addr...........: D8-9E-F3-88-7E-C3
Node1: Source IPv4 Addr..........: 192.168.25.08
Node1: Source UDP port...........: 0x0400
Node1: Destination UDP port......: 0x0401

Node1: *** Data Packet #1 from node 0 ***


Node1: Source MAC Addr...........: D8-9E-F3-88-7E-C3
Node1: Source IPv4 Addr..........: 192.168.25.08
Node1: Source UDP port...........: 0x0400
Node1: Destination UDP port......: 0x0401

Node1: *** Data Packet #2 from node 0 ***


$finish called at time : 1992 ns : File "C:/git/udpIpPg/test/tb.v" Line 132
exit
INFO: [Common 17-206] Exiting xsim at Fri Jan 10 08:43:40 2025...
```

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


Vivado 2023.2
File Edit Tools Reports Window Layout View Run Help
Quick Access
130 us
Default Layout
SIMULATION - Simulation - work.tb
tb.v
Untitled 1*
Scope
Sources
Objects
Protocol Instances
176.000 ns
Name
Value
clk 1
count[31:0] 22
halt[1:0] 0
txd[7:0] fb
txen 1
txer 0
rxd[7:0] 07
rxdv 0
rxer 0
160.000 ns
180.000 ns
200.000 ns
220.000 ns
240.000 ns
260.000 ns
280.000 ns
300.000 ns
320.000 ns
340.000 ns
18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
0
07 fb 55 d5 90 32 4b 07 0b d1 d8 9e f3 88 7e c3 08 00
07
Tcl Console
Messages
Sim Time: 1992 ns

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