Arbitrary Protocol Modelling Using the *VProc* Co-simulation IP

Simon Southwell

May 2025

Preface

This document is a PDF version of an article written in May 2025 and published on LinkedIn, on modelling arbitrary protocols using the *VProc* virtual processor.

Simon Southwell
Cambridge, UK
May 2025

Contents

Introduction

In this document I want to discuss how to use the *VProc* virtual processor to model any protocol, whether standard or custom, in a co-simulation environment so that it can be driven from a C or C++ program to run in a logic simulation. Now, I want to stress, right up front, that this IP will not create the protocol model for you and you have to put the work in to create the model of the protocol yourself. What is discussed here is a technique of using *VProc* to generate any arbitrary signalling you may want, even though *VProc* has a generic memory mapped interface on its HDL component. Now *VProc* comes with a set of bus functional model (BFM) wrappers for converting between the generic memory mapped bus to more standard protocols, such as AXI, AHB and Avalon, but these are simple conversions between one memory mapped address bus protocol to another, and don't require the methods described here. Even custom address bus protocols have had *VProc* BFM wrappers, such as the "soc_if" address bus interface of Chili.CHIPS*ba, used in their Wireguard FPGA project, just require some small amount of logic to convert. Affectively, these all do the same thing—read and write over a bus with, potentially, some burst accesses, and require no special techniques to do this.

The method described here is tried and tested and has been used in various verification environments, including in commercial developments. Indeed, examples are available in my github repository for the following co-simulation models, all of which have a *VProc* virtual processor at their core.

- *tcpIpPg*: a TCP/IPv4 pattern generator with XGMII.
- *udpIpPg*: a UDP version of *tcpIpPg* with GMII and RGMII support.
- *pcievhost*: a GEN1 and GEN2 PCIe root-complex model with support for some endpoint features.
- *usbModel*: a USB1.1 model with hooks for USB2.0 support

It is the above protocols, and others like them, that can be modelled using the techniques I want to talk about here, and that are used in the listed verification IP. Beyond this, in fact, any arbitrary signalling can be modelled, if it is desired that this signalling is best driven from within a C or C++ program.

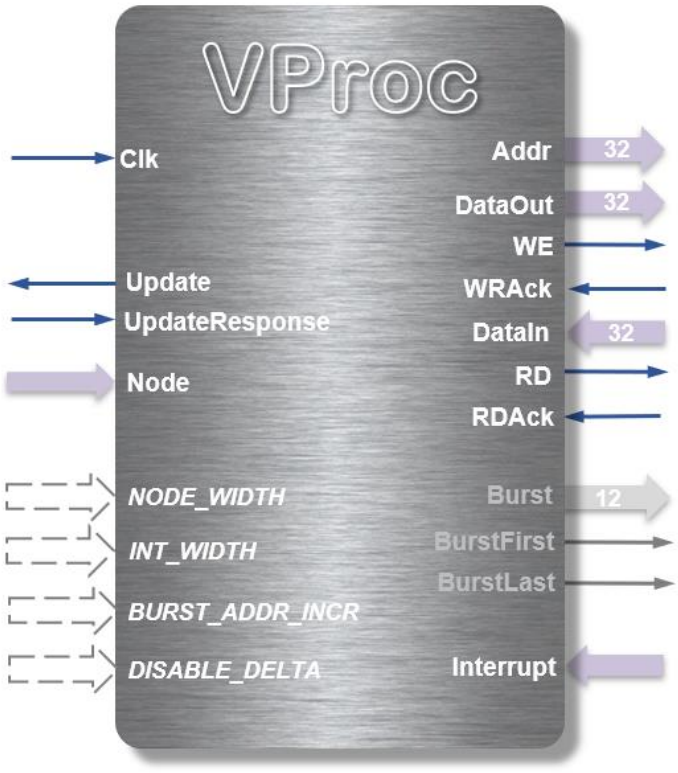
The techniques we'll look at rely on a feature of *VProc* known as delta-cycle updates. I have discussed *VProc*, how it works, and its general usage, in a previous article, including delta-cycles, but it is worth, for those who are unfamiliar with *VProc*, having a summary section on the virtual processor before the main subject matter and we'll emphasise the features relevant to this document.

The *VProc* Virtual Processor

The *VProc* virtual processor tries to answer the question "*how can I write an ordinary C or C++ program for my PC or workstation that can do reads and writes on a bus in an HDL logic simulation so I can, say, read and write to peripherals' registers?*" I was actually asked this by a verification manager who wanted such a means (I was working on other, different, co-simulation software then), and it got me thinking. At the time, commercial tools already existed to model specific processors which could have programs compiled for their particular instruction set architecture, but these were not free tools, the programs had to be cross-compiled for the ISA and exist in a specific embedded environment. They were also relatively slow. What was wanted was a program compiled for the host machine. Circumstances at the particular company overtook all this, and I didn't get the chance to do this work for the manager, but this means that I started my own development on these lines as an open-source project.

The Logic

So, from an HDL point of view, we want a component that we can instantiate in our logic simulation that looks like a generic processor core. At its simplest, this would have an address bus of some sort, a clock and, to be a complete core, an interrupt input (whether single or a vector). The diagram below shows the current *VProc* HDL component.



Here we see the clock input on the top left, with the generic address bus signals on the top right. An optional set of burst signalling is underneath this and then the interrupt input vector. As for the other signals on the left, the `Node` input value is a handle or ID (or whatever you want to call it) used to give a unique number to a particular instantiation of a *VProc* component, as we can instantiate more than one if we wish. Each instantiation must have a unique number. Above the `Node` signal, the `Update` and `UpdateResponse` signals are used in delta-cycle updates, so I will skip these for now as we will look at this in more detail later.

The parameters/generics shown on the bottom left of the diagram allow some configuration of the components. The width of the `Node` input is set by `NODE_WIDTH` and allows for an environment with many *VProc* instantiations. The default width is 4, allowing up to 15 *VProc* devices in a simulation—usually adequate for most situations. The `INT_WIDTH` parameter does the same for the `Interrupt` input and defaults to 3 but can be up to 32 bits. The `BURST_ADDR_INCR` is for the optional burst interface, defining the increment of the `Addr` output in a burst. This defaults to 1 but could be set to, say, 4. This would be for if the address is a word address or a byte address. Lastly, the `DISABLE_DELTA` parameter was added at the request of a user. This removes the internal code associated with delta cycle updates, for when these are not being used in a particular test bench. It has a minor speed advantage.

I won't go into the internal details of this component here (see the article I wrote previously for these details) but, suffice it to say, it is fairly lightweight and makes calls over the logic programming interface to connect to a C environment. Which particular programming interface is employed will depend on the simulator being used and the language (VHDL, Verilog, SystemVerilog). Support for VPI, DPI, FLI, and VHPIDIRECT is available. If none of those acronyms mean anything to you then don't worry, you don't need to know about them for this discussion. In a *VProc* context, they allow, from HDL, a call to a function written in C, passing in parameter values and having parameters updated on return. I have written a series of articles on some of these interfaces previously, documented here.

So, from a logic test bench point of view we simply compile in the *VProc* Verilog or VHDL (as appropriate) and instantiate it where it's needed. If a particular bus protocol is required, one of the provided BFM wrappers can be used or a custom one made. Now we just need to be able to write our program to run on it.

The Software

Normally, when we write a C or C++ program for our PC or Linux workstation we firstly write a `main()` function. This is the entry point for our code—the first function that will be called when the program runs. Actually, this isn't quite true in all cases. For example, if writing a Windows graphical program `WinMain` (or `wWinMain`) is our entry point. Borrowing from this concept, a *VProc* program will have an entry point called `VUserMain<n>` where `<n>` is the node number for the particular *VProc* instantiated component—e.g. `VUserMain0()`. This allows us to have different programs running on different *VProc* components but still compiled into a single program that can be loaded into the simulator. For now, let's assume we only have one *VProc* component in the test bench that has a node number of 0 and the program entry point is `VUserMain0()`. Treating this like `main()` we can now do anything that we can do for any normal program to be compiled for the machine were running on as we're are going to compile this code on our machine.

The only thing that's missing is a means to drive the *VProc* component's signals. *VProc* provides a simple API for either C or C++ as appropriate. In C, this is made available by including the header `VUser.h` and for C++, `VProcClass.h`. When we compile are code, we also compile in a couple of small *VProc* source files (compiled as C): `VSched.c` and `VUser.c`. These provide the code for the connection to the simulator and the user API respectively. The API is very simple, providing means to do writes and reads over the bus, with burst variations of these basic functions. There is also a function to advance the simulation without doing any read or write transactions. A few other functions are available to do with registering callback functions on various events, such as an interrupts, but these need not concern us here. The table below shows a summary of the most basic API functions and methods.

C function	Class method(s)	Description
N/A	`VProc (unsigned node);`	Constructor
`int VWrite (unsigned addr, unsigned data, int delta, unsigned node);`	`int write (unsigned addr, unsigned data, int delta=0);`	Write 32-bit word
`int VWriteBE (unsigned addr, unsigned data, unsigned be, int delta, unsigned node);`	`int writeByte | writeHword | writeWord (unsigned addr, unsigned data, int delta=0);`	Write word with byte enables
`int VRead (unsigned addr, unsigned *data, int delta, unsigned node);`	`int read | readByte | readHword | readWord (unsigned addr, unsigned data, int delta=0);`	Read words, half-words and bytes
`int VTick (unsigned ticks, unsigned node);`	`int tick (unsigned ticks);`	Advance simulation by a number of clocks ticks.

With these functions or methods, along with the others provided, we can now do reads and writes in the logic simulation over the *VProc* memory mapped address bus.

We now have the code we need to write a C or C++ program and an API to initiate transactions in the logic simulation. We still need to compile our program and then "somehow" connect it to the simulator we will use to simulate out logic test bench.

Compiling the Software and Connection to the Simulator

Compilation of our program is straightforward, and it is detailed in the *VProc* manual, and make file examples are provide for all the supported simulators, but we'll summarise it here. I'll assume we will compile our code in two steps, generating object files first and then linking everything together, and that we're on Linux.

The user program we've written can simply be compiled with a `gcc -c` type command. In addition, we will have to make this "position independent" code, for reasons I'll discuss shortly. This just means adding `-fPIC` to the `gcc` (or `g++`) command. The *VProc* headers included in the user program code make reference to headers that the simulation tool provides, so a `-I <sim include dir>` will be needed. This will be different for different simulators. For Questa, as an example this is:

```
-I <path to quest installation>/include
```

Something similar will be needed for other tools, and their manuals will need to be consulted. The provide make files will also have examples. The *VProc* headers will also need to be compiled with some definitions set to select the appropriate code for the programming interface that's appropriate and whether using Verilog or VHDL, employing the `-D` option. For example, if Questa is the simulator and VHDL is the language used, we use the two define options:

```
-DVPROC_VHDL -DMODELSIM
```

The *VProc* manual lists the various options for the various simulator and language combinations. Any other options that the user code requires can, of course, be added. As an example, if we have a single user source code file, `VUserMain0.c`, and our compiling for Questa and VHDL, then we can generate the object files needed as:

```
gcc-c -fPIC -I<path to questa installation>/include \
            -I<path to vproc>/code                  \
            -DVPROC_VHDL -DMODELSIM                 \
            VUserMain0.c                            \
            <path to vproc>/VSched.c                \
            <path to vproc>/VUser.c
```

We now need to link these object files together, along with some library code provided by the simulator. Normally we would be compiling all this into an executable which we can run. However, we need to connect to the simulator, and we must compile into something that it can load and run. For most simulators, this simply means compiling into a "shared object" (or "dynamically linked library" in Windows parlance). With `gcc` this just means adding the `-shared` option which produces a `.so` file. A shared object is *almost* an executable but has no `main()` function and is meant to be loaded by a program (that does have a `main()` function, like a simulator) at run-time. Since it is loaded at run time its memory location isn't defined and so it must be made from "position independent code"—hence the use of the `-fPIC` flag when compiling the object files.

To pull in the required libraries from the simulator the use of `-L<tool lib path>` and `-l<lib name>` are required. The *VProc* code makes use of posix threads, so this library is also needed, using `-lpthread`. Some additional flags are also needed to ensure the simulator can see the functions it needs to call and for the linker to not delete these functions because nothing references them (though they will be referenced from the simulator at run-time). So, an example link command might look like the following:

```
gcc -shared -lpthread               \
    -rdynamic                       \
    -Wl,-whole-archive              \
    -L<path to questa lib> -lmtipli \ (Note: not needed for VHDL)
    VUserMain.o VSched.o VUser.o    \
    -Wl,-no-whole-archive           \
    -o VProc.so
```

When the simulator is run, for VHDL, it will automatically pick up the `VProc.so` file when run. If using Verilog, then a `-pli VProc.so` command line option is needed when running `vsim`. It is a similar situation for other simulators.

So, there we have it. We can now natively compile and run software on the virtual processor, instigating memory mapped transactions on the *VProc* component's bus. This is fine if we want to use *VProc* as an address bus master, but how can we use for other types of interface?

Creating Non-Memory Mapped Components

Once approach might be to write a new replacement HDL component with ports for the protocol we would like. The only issue here is that the programming interface tasks or foreign procedures (for VHDL) have arguments oriented towards a memory

mapped bus. We could write new PLI tasks or foreign procedures, but then we are starting to write a brand new virtual component and would have to do this for every protocol we might want to support. A more generic way is to build a wrapper, much like I mentioned for the bus protocols such as AXI or Avalon. But, instead of doing a simple conversion from the generic bus to a standard bus (which is essentially doing the same thing) we need to be a bit more imaginative and use the features of *VProc* in a clever way. In particular, we will use the delta-cycle update feature. So let's have a look at this.

The delta-cycle Features of VProc

In a logic simulation—at least, a fully event driven logic simulation (which most are)—a delta cycle is a slot where the simulator resolves signals due to changes, but simulation time does not move forward. Imagine at a clock edge a load of signals get updated from registers. The connected combinatorial logic inputs change and their outputs *might* change, which change more combinatorial inputs and so on. In this delta time slot, initially the logic connected to the register outputs are put in an (imaginary) list to be resolved. The simulator works through this list and any that have their outputs change that are connected to other combinatorial logic will have the connected logic added to the end of the list for resolution. The simulator keeps processing the list and, eventually, no changes in that delay time slot occurred and the list is empty. Note that any changes that occur in the future, such as a delayed assignment, will be added to a *new* list at that simulation time slot. When a delta time slot's list is empty, the simulator moves to the next list, which may be at an intermediate time before the next clock edge, or at the next clock edge when registers will be updated. This is an oversimplification, but it gives you an idea of what's going on.

Now we can change the state of a signal in a combinatorial process and then change it back again using, say, two blocking assignments. On the waveforms that signal will look like it didn't change, but it did, and anything connected to it will see that change. This is a delta-cycle update. The connected logic will be put on the delta time slot list and processed, just as we described before. We can use this to our advantage

The *VProc* component has the `Update` and `UpdateResponse` signals to allow delta-cycle updates external to the component. In normal operation, the *VProc* HDL code is running in a loop, waiting on its clock input, making calls to the C code for new transactions. When one arrives it instigates that transaction, over one or more clock cycles, and then calls the C code again for a new transaction or a set of clock cycle

'ticks' to pass, and so on. This is all synchronous. If you remember, the API supplied has functions and methods with a delta argument. For normal operation this is set to 0, but if it is set to `DELTA_CYCLE`, then almost the same thing happens as before, except the clock is not waited for to iterate the internal loop. In addition, the `Update` signal changes state, from 1 to 0 or 0 to 1, depending on its current state. It will, in this case, wait on the `UpdateResponse` to change state before moving on. Since no delays were added in this transaction, the transaction happened in a delta time slot. We can call any number of API functions in delta time to do as many reads and writes as required without advancing simulation time. When we're done, on the last function/method call, we don't select delta-time and time advances as normal.

External to the *VProc* instantiated component, then, we can have a process that's sensitive to the `Update` signal changing, process any of the bus signals as required, and then flip the `UpdateResponse` signal (assumed initialised in an `initial` block).

```
always @(Update)
begin
    DataIn = 32'h00000000;

    if (WE === 1'b1 || RD === 1'b1)
    begin
      case (Addr)
        // *************************
        // *** Do processing here ***
        // *************************
      endcase
    end

    // Finished processing, so flag to VProc
    UpdateResponse <= ~UpdateResponse;
end
```

With this simple process we can test the address bus signal outputs for reads and writes and even return read data, all without advancing time. So multiple transactions can be processed before time is allowed to move forward.

Constructing an Arbitrary Model

How does this help us? Well, imagine the signals we need for the interface we want to drive has a lot of bits. For example, say we want a PCIe PIPE interface of 16 lanes. At the very least we will have sixteen 9-bit ports for the data, in both the input and output directions, and there is likely to be set of control and status signals. The *VProc* component only has a 32-bit data bus, so can't update and read these all as a single

transaction, and it can't use several clock cycles to do it if the protocol requires single cycle updates (which the PIPE does). But, by using the delta-cycle features, we could update and read these ports in delta time. *VProc* is still an address bus component, so the way to do this is to memory map the input and output signals. So long as each is 32-bits or less, this is easily done. Even if a port is larger vector, it can be address mapped in 32-bit sub-sections. We now have a means to read every port and, for outputs, write the ports, having declared then as `reg`, in Verilog parlance. From a software point of view we just use the read and write API calls with the appropriate address to access these signals. In the models I have constructed to date, the addresses are defined in a Verilog header and a C header is automatically generated from this via an `awk` script, so that the software and HDL address definitions match. The code below shows a simplified and abbreviated version of the update process for the *pcievhost* model, where the link lane outputs are 10-bit encoded data symbols.

```
`include "pcie_vhost.vh"

reg    [9:0] LinkOut [0:15]; // Assume driving the output ports
wire   [9:0] LinkIn  [0:15]; // Assume connected to the input ports

always @(Update)
begin

    if (WE === 1'b1 || RD === 1'b1)
    begin
      case (Addr)
        `LINKADDR0,  `LINKADDR1,  `LINKADDR2,  `LINKADDR3,
        `LINKADDR4,  `LINKADDR5,  `LINKADDR6,  `LINKADDR7,
        `LINKADDR8,  `LINKADDR9,  `LINKADDR10, `LINKADDR11,
        `LINKADDR12, `LINKADDR13, `LINKADDR14, `LINKADDR15:
        begin
            if (WE === 1'b1)
                LinkOut[Addr%16] = DataOut[9:0];
            DataIn = {22'h000000, LinkIn[Addr%16]};
        end
        `NODENUMADDR: DataIn = NodeNum;
        `LANESADDR:   DataIn = LinkWidth;
        `EP_ADDR:     DataIn = EndPoint;
      endcase
    end

    // Finished processing, so flag to VProc
    UpdateResponse <= ~UpdateResponse;
end
```

As well as being able to write to the lane output signals and read the lane inputs (at the same addresses), various other state values can be read, such as the values of the module's parameters. In the real model there are also controls for stopping or finishing the simulation, as well as some other settings.

In the above example, we now have control of all the PCIe ports from software. We can have a function that's called every cycle to update all the lane outputs and read the lane inputs:

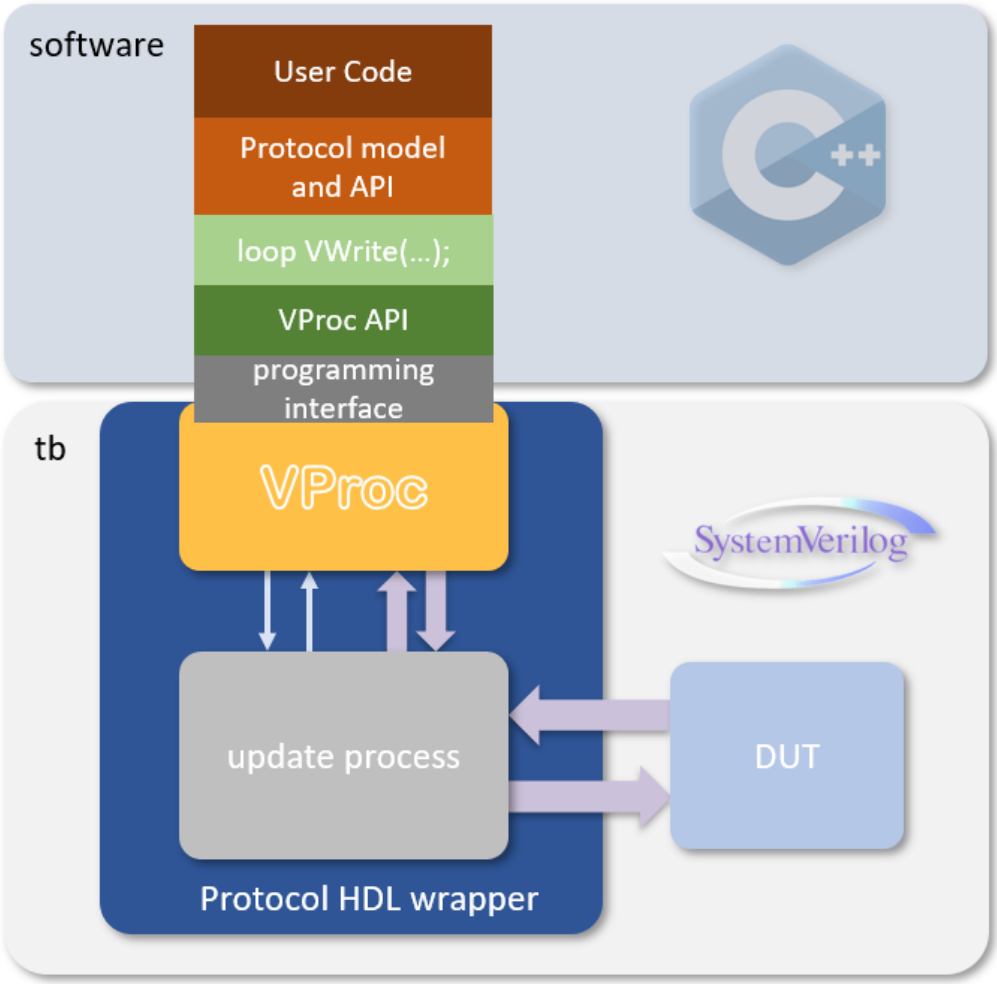
```
void Update (const int node)
{
                        ⋮

    for (lanes = 0; lanes < LinkWidth; lanes++)
    {
        LinkIn[lanes] = VWrite(LINKADDR0+lanes, LinkOut[lanes],
                               lanes != LinkWidth-1, node);
    }

                        ⋮
}
```

Here, in an `Update` function, some processing is done and then a loop around all the active lanes does delta cycle writes to each lane in turn, with the lane input values returned (in this case) by the call to `VWrite` (if you check the HDL code from before, you'll see that `DataIn` is updated whether a write or a read. The third parameter (for the `delta` flag) is non-zero for all but the last lane, so simulation time is advanced at the update of the top lane, having updated all the others in delta-time.

Of course, all we have done here is mapped the ports of the PCIe wrapper to addresses in the C domain. It does not give us any means to know what values should be put on those lanes. There is no easy path to this, and a C or C++ model must be written to generate the patterns to be written based on received input and desired transactions to be sent. All the models I have constructed so far follow the same basic pattern, with a layered approach to construction. The diagram below summarises this stack.



We have already discussed the layers from a wrapper module with the appropriate ports, driven from an update process in delta-cycle time. This is connected to the *VProc* component's update signals and the address bus. Connection to the C and C++ domains happens over a programming interface such as DPI or VPI etc., provided by the HDL language and the simulator running the test bench. The *VProc* software provides the connection to the simulation, presenting an API to higher layer software to do reads and writes via, for example, `VWrite()`, which can be wrapped in some user written low level function, such as the `Update()` function we saw earlier, that is aware of the addresses for the memory mapped ports, as defined in an HDL and (auto-generated) C header. At this point it is now up to the user to provide a model that generates the values that we need to place on the HDL component's output ports and process the values on the input ports. Potentially, this model provides its own API to higher level user code, having virtualised away the details of driving the simulation. These would be much higher layer functions or methods, with a PCIe example being functions to generate memory, configuration or I/O reads and writes, or send PCIe messages etc.

Motivation for this Approach

Having described the method of driving an arbitrary protocol's signals on an HDL component, it seems I've left out the most complicated part. That is, writing a model in C or C++ to generate the value to place on the ports. Why, then, use this method?

It may be that you already have access to some verification IP to drive the desired protocol, or even some synthesisable logic IP that can be driven from some HDL model of a processor, that can do the same thing. This method of using *VProc* does not replace these things and using VIP or existing logic may be the best approach for a given project.

The original motivation for using *VProc* in this way, for a PCIe endpoint development, was because I didn't have access to VIP (for most of the development) or any existing logic, the specification was new, and a very important aspect was that it had to run fast to maximise the number of vectors passed through the logic (not limited to just the endpoint design).

Because the specification was new, and I knew nothing about it, I started to construct a C model to generate packets that I could inspect that met my understanding of the specification. As this matured, I'd met one of my criteria for having access to a model in the C domain. *VProc* itself is inherently fast as it runs the C code as dynamically linked library code and effectively becomes part of the simulation, and each user program is run as a separate thread. Other methods I've seen connect to the simulator via TCP socket, running as a completely separate process. This requires a lot of calls through the operating system and the heavyweight transporting of TCP packets back and forth. Running code directly as part of the simulation is much faster. Also, a C model is running as normal code on the host machine, rather than as HDL in the simulation, and is thus much faster that an equivalent HDL model.

I have also found that writing a model in C or C++ a much simpler task than doing the equivalent task in an HDL language. There are no timing considerations or parallel processes etc., and in all the models I've done so far fall along the lines of encoding into buffers and sending, and reading received data into buffers and decoding. This is particularly advantageous if a new protocol is to be modelled, whether a new standard or some custom protocol, and no IP of any kind already exists. To knock up a model in C or C++ can be relatively quick and getting something running to the point of driving signals in a simulation allows for experimentation and testing early in the design process—just as it did for my PCIe development. In fact, this model was later taken by the software team and extended to include their Linux kernel driver development running on QEMU to produce a full

co-development and co-simulation environment, running sufficiently fast to be able to interact with it, in real-time, in a terminal.

An Alternative but Related Approach

It is worth finishing this section with a look at an alternative approach that is related to *VProc* and may be more appropriate to a given project. The OSVVM methodology and verification VHDL library now has co-simulation support. The underlying co-simulation technology of *VProc* was used to do this, exposing OSVVM's API for model independent transactions (MIT) for both address bus and stream protocols, allowing it to drive Verification Components (VCs), for interfaces such as SPI, UART, Wishbone, VideoBus, AXI and Ethernet. It has also had the *rv32* RISC-V instruction set simulator integrated on top of the co-simulation.

If a VC already exists for the protocol required, with new ones being added all the time, then this might be the approach for you. If not, then a new VC needs to be constructed, and it is extensively documented how to do this, though now the choice is whether it is better to model an OSVVM VC, or write it in C or C++ and use the techniques documented here. At some point someone has to model the protocol in some form.

Conclusions

In this document I introduced the *VProc* virtual processor as a co-simulation verification IP that allows user written C or C++ code to be compiled natively for the host machine and have that program 'run' on a generic processor component instantiated in test HDL executing in a logic simulator. The component is ostensibly for modelling a processor core, having a memory mapped address bus as its main interface.

In addition, though, we looked at the delta-cycle features of *VProc* and discovered that these could be used to model any arbitrary protocol and its signals. The process for doing this was explained with examples for just such a model, and four existing models were referenced for Ethernet's UDP and TCP protocols (with IPv4), USB and PCIe.

The technique gets to the point where a C or C++ model to generate the data and signal patterns can be integrated for co-simulation, but the method doesn't provide these models. We finished with a discussion on why this method might be used over other approaches, with speed and ease of initial model development amongst the

advantages. An alternative approach using OSVVM's co-simulation capabilities and verification components was also looked at to close off the discussion.