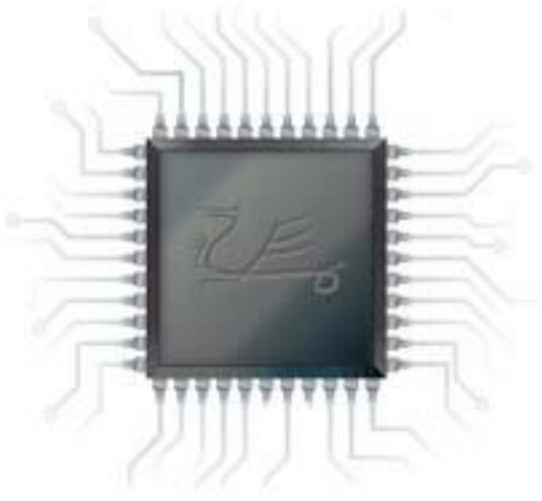
# The *VProc* Virtual Processor Verification IP



Simon Southwell

January 2024

## Preface

This document is a PDF version of an article written in January 2024 and published on LinkedIn, on the VProc virtual processor verification component.

Simon Southwell
Cambridge, UK
January 2024

## Contents

## Introduction

I have written about the VProc virtual processor verification IP in a previous article, which was the second in a set on the programming interfaces of hardware description languages running on logic simulators. In that article the main focus was on the use of the different simulation programming interfaces, with the second article focussing on a virtual processor as a real-world example. This was, necessarily, a summary with only a few details on the internal workings of the component.

In this article, I want to elaborate on the design and go into more detail of how VProc actually works, the issues that had to be solved and the particular solution used and why these choices were made (whether rightly or wrongly). The idea is to give an insight on how, by creatively mixing some basic software and HDL, and with a little bit of knowledge of the programming interfaces, allows the creation of (I believe) a powerful solution that can lead to verification abilities that would be much more complex to do in pure HDL (and probably slower) and even some capabilities unavailable in pure HDL, such as running real embedded code in the logic simulation.

You may be asking, if you've done all the hard work already, why can't we just use VProc as a black box solution, and not worry about how it works inside? That is a perfectly valid point, and you can read the VProc manual and do just that. But, I would argue, understanding how VProc works may give people some new ideas of how to use the programming interfaces that go beyond the VProc design in new ways that have not yet been thought of. For example, earlier last year (2023), I add co-simulation capabilities to OSVVM. This uses the same basic principles as VProc to connect the software domain to the logic simulation domain and synchronise the two, though it is not a virtual processor. The code borrows from VProc but is largely rewritten, and rather than map a memory-mapped bus to the user code, it exposes procedures for a protocol independent transaction layer model, which can then be mapped to protocol specific signalling, such as AXI, in the VHDL domain using the provided OSVVM verification components, or ones written by the user. The point is that the central ideas are transferable to other problems and not just to a virtual processor, and by understanding the mechanisms you might come up with ways of adapting them to new solutions; ones that I haven't even thought of. In fact, I really hope you do.

The article I referred to earlier does cover some principles of VProc but I don't want to make any assumptions and make it necessary to have read that article in order to understand this one, and I want to keep this article as stand-alone document. If you have read the previous article, forgive any repetition.

## What Is a Virtual Processor?

Before answering the question of what a virtual processor is, let's first ask what is a processor (in particular a single core) from the viewpoint of its external interfaces. Whatever the internal design and architecture, a processor core interacts with the outside world with just two basic interfaces; a master memory mapped bus interface, and interrupt input interface. (I'm talking about CPU cores such as ARM and RISC-V, rather than microcontrollers, DSPs, or GPUs.) The bus interface might be split in two for separate data and instruction buses (a Harvard architecture) and the interrupt inputs may be a single input (implying an external interrupt controller) or have several separate interrupt inputs. None-the-less, these are just engineering optimisations. The internal details of a core are of no consequence to the external logic, and it will just expect a bus master (or two) and interrupt inputs. Thus our virtual processor need only provide a component with these ports.

Another feature of a processor core is that it runs software, usually compiled from a high-level language such as C++ or C—especially if targeting a processor in an embedded system. Our logic simulator will run on a PC or a workstation, and we can compile and run programs for this 'host' computer to run on its processor(s). What would be useful to do is to be able to write a program, compiled for our host computer, that can read and write over the bus of our HDL processor component running in a logic simulation and to process any interrupt inputs. Our program would then be 'running' on that simulated HDL processor component in the logic simulation and thus is a 'virtual' program, running on the 'virtual' processor. It's virtual because the programming isn't being processed internally by the HDL processor component logic but is running on the host computer. From the rest of the logic simulation's point of view, though, they just see normal bus transactions and can generate interrupts, interfacing with the processor component's ports.

So this is the definition of a virtual processor in the VProc case. So how can we do that and what problems need solving? The list below is roughly the list I originally started thinking about (as near as I can remember, nearly twenty years on) when thinking about designing such a component.

1. We need an HDL module (in Verilog or VHDL—VProc supports both) that looks like a generic processor core.
    - A clock input.
    - A generic memory mapped master bus.
    - A vectored interrupt input.

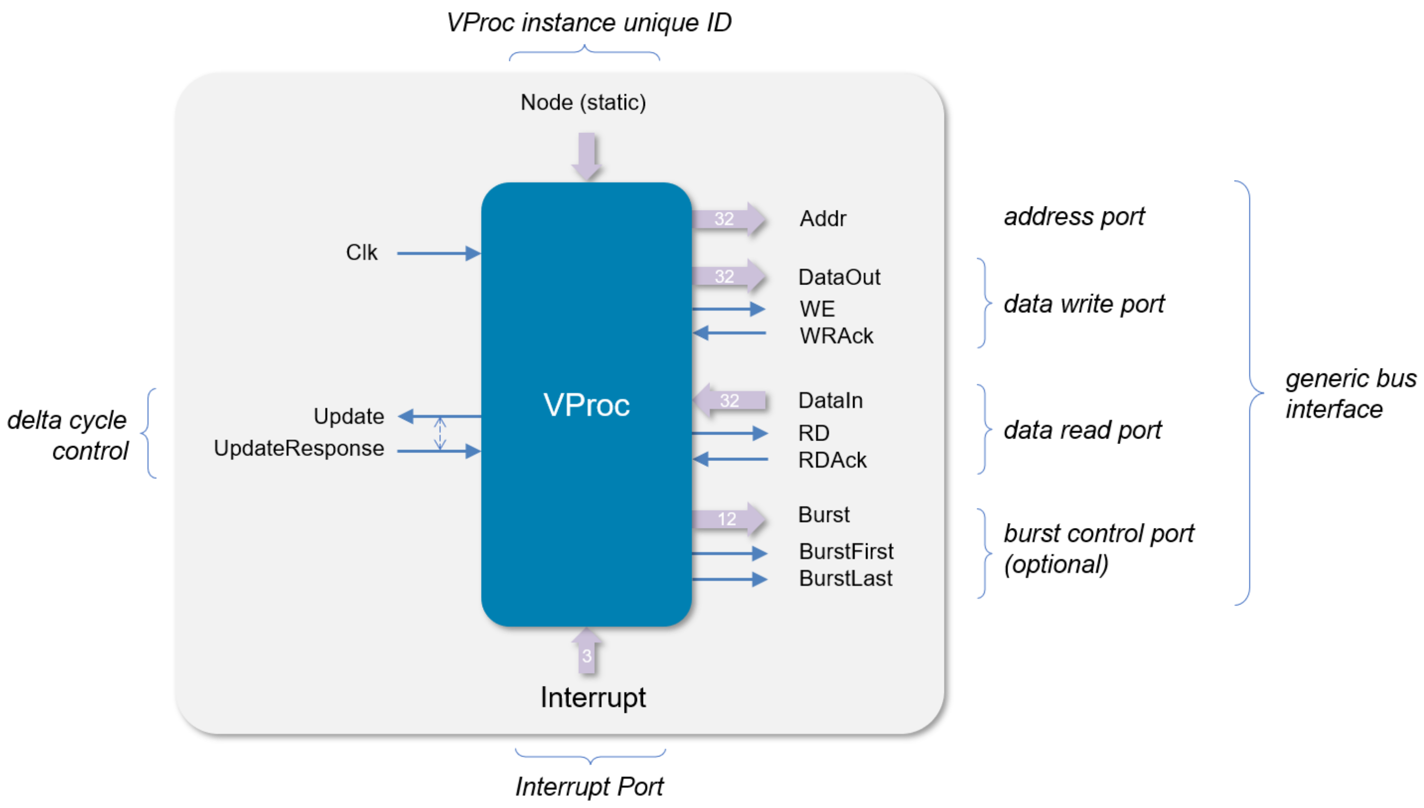
2. We need to use a standard programming interface to cross to the C/C++ software domain.
    - Verilog has PLI and VPI.
    - VHDL has VHPI (but didn't when VProc was first conceived, so the similar ModelSim FLI was used).
3. The user software needs to run freely, like normal a program.
    - Can't be a series of call-and-return functions, invoked from HDL over the PLI
    - Needs the equivalent of a 'main' entry point.
4. A simple API needs to be provided to the user program and must include, as a minimum:
    - Memory Read function.
    - Memory Write function.
    - Interrupt processing.
5. Must be able to instantiate multiple virtual processors, running different programs.
    - Need to be able to model a multi-core systems (which was true of the first real-world VProc use case)

In the rest of this article we will look at these requirements in turn to see what solutions were chosen and why, bearing in mind that other solutions may have been possible, or even more desirable.

## The HDL Component

The virtual processor component has two aspects to it. The first is the ports that are visible to the rest of the HDL solution. The second is the internal behavioural logic.

For the ports, these were chosen to be as generic as possible, rather than be tied down to a specific standard protocol, such as AXI or Avalon memory mapped protocols. The idea is that a 'thin' bus functional mode (BFM) layer would convert from the generic bus signalling to a protocol specific signalling. This, as it turned out, also allowed other usage models for the VProc component, as we shall see later. The interrupt input is a small vector input to allow for some modelling of multiple interrupt sources. The diagram below shows the current VProc component and its ports. There are also some additional ports from the requirement just stated, and we shall briefly discuss these shortly.



We can see from the diagram that the component has a clock input but there is no reset input as the code initialises itself. This does mean that it isn't possible to do a hard reset on the virtual processor and have its code restart from scratch. Such behaviour can be emulated in software by monitoring a memory mapped reset signal, or wiring an interrupt level to a reset, but this hasn't been needed so far in any VProc deployments, though my usbModel monitors for reset removal before allowing transactions to start.

The generic memory mapped bus is geared up for reads and writes of single 32-bit words in a 32-bit address space. It is not specified whether the address bus is a byte or a word address, and this is an 'agreement' between the software calling the API and the external logic. This flexibility allows for simple read-modify-write style accesses with word address values, to the generation of byte enables from the lower address bit when a byte address, for byte or half-word style accesses. 32-bit was state-of-the-art when VProc was first constructed, but future proofing was built into the design to allow larger address spaces and larger word accesses, with just a little bit of extra work by the user logic and software. This can be achieved with 'delta-cycle updates' which we will discuss in a later section.

The generic bus itself consists of an address output, a read data port and a write data port. Each of the data ports has a strobe output (`RD` or `WE`) and an acknowledge input

(RDAck and WRAck) to allow for wait states on accesses. The write port has a matching 32-bit data output (DataOut) whilst the read port has a 32-bit input (DataIn). The two data ports are mutually exclusive, so only one can be active at any one time, and the bus is idle when both strobe outputs are at 0. If it is known that all accesses will have no wait states, the acknowledge inputs can be tied directly to their counterpart strobe outputs.

There are also some optional burst access output signals that have been recently added, but these are just useful extensions and were not in the original design—and are *not strictly necessary* to a virtual processor component. A burst access uses the read or write port signals but is active when the Burst signal is a non-zero value, specifying the burst length. The first and last transfers are flagged with the BurstFirst and BurstLast outputs. The address will increment by one at each transfer, implying word addressing, but if this is not the case some conversion can be done in the behavioural logic (e.g., to increment by four from the initial start address if word aligned).

The Interrupt input is a 3-bit vector. This actually works such that it is idle when 0 is input, and is active when non-zero, giving 7 possible interrupt levels. In this sense this has been constructed to be geared to interrupt 'events' such as the changing of an interrupt signal level for a level triggered interrupt or to an edge triggered interrupt pulse—this method supports both. Some minimal logic may be required to flag events and generate a non-zero value on the Interrupt input for a single cycle, but the ports functionality does not assume what interrupt source signally will be like and is a means simply to call different user supplied code when active. As we shall see, for every cycle that the Interrupt is non-zero, it will call a user registered function (if one was supplied) for the particular active level. Note that it is possible to have an interrupt event whilst the memory mapped bus is active and the user interrupt code will be called in that active cycle to register the event, without disrupting the memory mapped access.

So we have now met the requirements for item 1 in our specification list, and even have some extra functionality in the burst signalling. There are two other ports in the diagram that we need to discuss. The first is the Node input. This there to meet requirement 5. If we are to have multiple instantiations of VProc, and each one run a different program, then we need to uniquely identify each one. Each VProc instantiation must have a unique Node number. I chose the name 'node', but this could easily be ID, handle or whatever. It all means the same thing. It is the number that is particular to the instantiation and will connect it to its own user program to run. If multiple instantiations of VProc are present and two or more share the same node number, then this won't work as one might expect.

The final port is the 'delta cycle' port, consisting of an `Update` output signal, and an `UpdateResponse` input. This port wouldn't be necessary if only using VProc as memory mapped bus processor restricted to a 32-bit bus. In fact, if this is the case, then the `UpdateResponse` input should be connected to the `Update` output and can then be safely forgotten about. What it can be used for is to do 'memory mapped accesses' in delta time—i.e., in 0 time units. So, imagine, for example, that we want a 64-bit address space and 64-bit data words. We only have a 32-bit bus, but if addresses in the 32-bit memory space, not used by the system, have some 32-bit registers that can be written (and/or read), then upper address/data values can be loaded in a delta cycle to be used as the upper 32 bits of a 64 bit address and data value, and then a subsequent non-delta cycle access will do a 64-bit access, combining the save upper values with the non-delta access values. On reads, the upper bits of a returned data value can be registered, and this read in a delta cycle to construct the full double word. With some simple abstraction code, this can all be hidden behind 64-bit extensions to the basic API read and write functions. In fact, this delta-cycle update feature allows for any mapping of signals to be updated as a series of zero-time updates, allowing the mapping of non-memory mapped interconnects, such as streaming interfaces (e.g., PCIe). The interface signals are simply memory mapped into VProc's memory space so they can be updated and read in zero time until the final update, when time is allowed to advance with a non-delta access.

These, then, are the external ports for the component. We might get away without the delta-cycle port and can definitely dispense with the burst port (if making things slightly less convenient) giving a quite simple generic processor core. So what's happening on the inside?

A goal for VProc was definitely to keep the amount of HDL code to a minimum and do most of the processing in software, which is more convenient, more flexible, and much faster. Speed may not be a consideration for a given deployment but certainly was originally for VProc. ASIC development is time consuming and expensive, and having a fabricated chip return with a fatal defect in the design can ruin a project (or even a company). Therefore, running as many effective vectors through a design as possible to minimise risk is very desirable, and a suite of regression tests to ensure previously working functionality doesn't disappear with new updates is key. At each check-in of new functionality a large number of vectors may need to be run before the update can be folded onto the main code branch, and this can take a long time.

Minimising the simulation times shortens this cycle, and thus the development time. VProc, originally, was only part of a much larger verification system, but the goal was still to make it as fast as possible (after all, we were developing supercomputers, where this was also true).

## Inside the VProc Component

Up until now, I've been talking as if the development of VProc was a linear process, with well specified functionality that was then implemented without issue. Well, I can tell you that this was not the case, but it would not be useful to document all the blind alleys and experimentation that occurred (even if I could remember that far back). What it does mean, though, is that in order to discuss the VProc HDL internals we need to know about the PLI tasks (in Verilog speak, or foreign procedures for VHDL), even though that interface is yet to be discussed in detail, which it will be. I will stick to the Verilog PLI tasks (as VProc was originally only for Verilog), but the VProc manual gives details for the VHDL foreign procedures. In this section I want to describe making calls to the VProc PLI tasks and then driving the ports. We will delay the details of the choices made with the PLI interface until later.

Firstly, let's meet the PLI tasks available to the VProc HDL:

```
$vinit     (node)
$vsched    (node, Interrupt, DataIn, DataOut, Addr, RW, Ticks)
$vprocuser (node, Value)
$vaccess   (node, Idx, DataIn, DataOut)
```

I should first say upfront that these tasks, when VProc software is compiled and loaded in the simulation, are available to any piece of code in the simulation, and the VProc component doesn't have exclusive access. So these could be used in ways other than in a virtual processor if desired. They are just the HDL side API to the user code. All of the arguments are integers, and only the PLI functionality to map C functions to system task was used. This makes the PLI code simpler and gives maximum compatibility with all the different simulators. The original VProc code was developed on early versions of Icarus Verilog, and sticking to C function mapping and integer arguments meant that no PLI support issues were encountered, even though its actual first deployment was on a commercial simulator (it was also ported to Verilator at that time as well).

I will only briefly mention `$vprocuser` and `$vaccess` as they aren't strictly required for a Virtual processor implementation. The `$vprocuser` task is a convenience task and simply allows the ability to pass a value from the simulation to a user function if one has been registered for such a purpose, in a traditional PLI call to a C function manner. The task can be called anytime and from anywhere. The VProc HDL does not use this task. The `$vaccess` function is concerned with burst accesses and is used to fetch data from a C buffer or to place read data in that buffer, using an index to access the buffer array. We will not concern ourselves with burst transfers as, again, this is not essential for a virtual processor implementation.

This leaves us with just two tasks: `$vinit` and `$vsched`. The `$vinit` task is required to initialise the software side for the specified node (passed in as its only argument) and must be called once, and only once, for a given node before any calls are made to `$vsched` for that same node. In the VProc HDL it is called from an initial process, having ensured that the Node input to the VProc module has been set. This just leaves `$vsched`.

The `$vsched` task is the 'one size fits all' task that does all of the interfacing to the C code from the HDL. Putting interrupts aside for a moment, this task is called once for each read or write transaction, or for a set delay without a transaction. It is called `$vsched`, as it is for scheduling a new transaction to be generated once an access is completed or a delay expired and will block until the software has a new transaction to generate. It has a node argument (as do all the tasks), followed by an `Interrupt` input argument (more later). The next four arguments are for the memory mapped transactions, with a `DataIn` input argument for returning read data, a `DataOut` output argument for write data, an `Addr` address output value, and a `RW` read-write output value. The read-write output bottom bits are the read and write strobe values, with bit 0 for writes and bit 1 for reads. Once this task is called, the output ports of the VProc module can be updated with these values. The `Ticks` output argument is used to allow simulation time to advance without doing a transaction. When it is greater than zero the HDL code will not schedule a new transaction update until that many clock cycles have expired. This can be after a scheduled read or write, or it can be called with both the read and write strobes at 0, and simply add a delay before scheduling a new transaction. A simplified version of the synchronous process in the VProc module is shown below:

```
always @(posedge Clk)
begin
    RdAckSamp          = RDAck;
    WRAckSamp          = WRAck;
    DataInSamp         = DataIn;
    IntSamp            = {1'b0, Interrupt};

    if ((RD === 1'b0 && WE        === 1'b0 && TickCount === 0) ||
        (RD === 1'b1 && RdAckSamp === 1'b1)                    ||
        (WE === 1'b1 && WRAckSamp === 1'b1))
    begin

        $vsched(Node, IntSamp, DataInSamp, VPDataOut, VPAddr, VPRW, VPTicks);

        WE             = VPRW[`WEBIT];
        RD             = VPRW[`RDBIT];
        Addr           = VPAddr;
        DataOut        = VPDataOut;

        if (VPTicks >= 0)
            TickCount = VPTicks;

        Update         = ~Update;
        @(UpdateResponse)

    end
    else
        TickCount      = (TickCount > 0) ? TickCount - 1 : 0;
end
```

This is all there is to it as far as the reads, writes and delays are concerned. The process, synchronous on `Clk`, will sample the input ports, converting to integers suitable for `$vsched`, and then call this procedure if there is no outstanding access, or if an access is about to complete. When `$vsched` returns, the output ports are updated with the new values, as is a tick count register (`TickCount`) if the tick value returned by `$vsched` was not a negative number. The toggling of the `Update` output and waiting on the `UpdateResponse` to change is to do with delta-cycle updates. If they are tied together, then this is effectively a no-operation. If `TickCount` does get set to a value, then the process decrements this until zero, before it can call `$vsched` again.

Something slightly unobvious happens for reads. When $vsched is called with a new address and read strobe, the sampled input data is not that for the new read access. This will not be ready until `RDAck` is 1 which will be at least a clock cycle. When `$vsched` is called again to wait for a new access, the input data is valid for the previous read access. The underlying software will fetch this returned data input and return it to the read API call. Thus reads are one $vsched out of sync, but the underlying code takes care of this (as we will see later) and on the very first call to `$vsched` it handles that fact that returned read data is not part of any scheduled read and discards it.

We now need to deal with interrupts, processing these even if a memory access is in progress. The code fragment below shows the additional code to do this.

```
always @(posedge Clk)
begin
    RdAckSamp           = RDAck;
    WRAckSamp           = WRAck;
    DataInSamp          = DataIn;
    IntSamp             = {1'b0, Interrupt};

    if (IntSamp > 0)
    begin
        $vsched(NodeI, IntSamp, DataInSamp, VPDataOut, VPAddr, VPRW, VPTicks);
        IntSamp         = 0;
        if (VPTicks > 0)
            TickCount                = VPTicks;
    end

    // ------------------------------------
    // Read-write-delay code from before...
end
```

So to handle interrupts, above the code we saw previously, a test for a non-zero value on the sampled interrupt input (`IntSamp`) is made and, if it is, a new call to `$vsched` is made to pass in the new interrupt value. This does not cancel the outstanding access, but the underlying code knows that, since the interrupt argument is non-zero it must simply call the relevant registered user interrupt function (if one was registered for the active interrupt level) and then re-wait for the original transaction to complete.

Now we have to deal with delta cycle updates. Since the process is triggered on the clock rising edge, we will have to deal with delta cycle updates within the process before completing it, and this implies a loop. A delta cycle update is flagged when the `Ticks` output from the `$vsched` call is -1. Therefore, if we have a loop around the `$vsched` call which loops until the `Ticks` output is not a negative number, then we will get successive updates in the same clock cycle. The code is modified as shown below:

```
  always @(posedge Clk)
  begin
      RdAckSamp          = RDAck;
      WRAckSamp          = WRAck;
      IntSamp            = {1'b0, Interrupt};

      if ((RD === 1'b0 && WE        === 1'b0 && TickCount === 0) ||
          (RD === 1'b1 && RdAckSamp === 1'b1)                    ||
          (WE === 1'b1 && WRAckSamp === 1'b1))
      begin

          while (VPTicks < 0)
          begin

              DataInSamp    = DataIn;
              $vsched(Node, IntSamp, DataInSamp, VPDataOut, VPAddr, VPRW, VPTicks);

              WE            = VPRW[`WEBIT];
              RD            = VPRW[`RDBIT];
              Addr          = VPAddr;
              DataOut       = VPDataOut;

              if (VPTicks >= 0)
                  TickCount = VPTicks;

              Update        = ~Update;
              @(UpdateResponse)
          end loop;
      end
      else
          TickCount     = (TickCount > 0) ? TickCount - 1 : 0;
  end
```

A couple of things to note here. Firstly, the `DataInSamp` assignment has been moved to be inside the loop as it will need to be updated at each call to `$vsched` to return any read data at each delta cycle access. Secondly, `VPTicks` is initialised to be -1 (`` `DELTACYCLE ``) in the initial block so that the loop is entered at the first clock edge. It is updated at each call to `$vsched` from then on.

So we can loop to get multiple accesses per clock cycle, but how does this update external state without advancing time? This is where the `Update`/`UpdateResponse` ports are used. We've seen already that we toggle the `Update` output for each iteration around the loop and then wait for `UpdateResponse` to change state. If, in the code outside of the VProc component (BFM code, for example) has a process that has a sensitivity list that is just on `Update` (`always (@Update)`), then in that process the outputs can be used to update state, and the `DataIn` updated with any read state. At the end of that process, the `UpdateResponse` is toggled, to indicate that the delta cycle update is completed.

```
reg UpdateResponse;
reg [31:0] DataIn;
initial
    UpdateResponse  = 1'b1;

always @(Update)
begin
    // Extract VProc output signals

    // Do delta cycle access

    // Update data input (DataIn)

    UpdateResponse = ~UpdateResponse;
end
```

As I've mentioned before, this may not be necessary if delta cycle updates are not required. In VProc's first deployment, standing in for a set of 'as-yet-to-be-completed' processors to drive a switching interconnect block, this was not needed. It's second use case, though, was for a software model of a PCIe root/endpoint to drive a PCIe endpoint design (and, indeed, to co-simulate with the kernel software to drive it and the highspeed network beyond) and this did use the delta cycle updates to update the link lane output signals and return the link lane input line states. The original version of this PCIe model is open-source and available on github. (The one mentioned had project specific modifications to ease integration with the kernel software, and these are not available, but not really useful outside of that project in any case.) The open source PCIe model can serve as an example of the use of delta-cycle updates. The same techniques are also used for my usbModel (USB software model) and tcpIpPg (10GbE XGMII TCP/IPv4 packet generator) projects.

So that's the HDL/logic simulation side covered, and we've crossed off items 1 and 5 (mostly) and the HDL side of item 2. For item 2, we have defined the HDL side PLI system tasks, but we now need to cross into the software domain using the programming interface, whether PLI, VHPI, FLI or whatever.

## The Software

In this section we have moved from the HDL domain to the software domain. The first hurdle is using the PLI (or other standard programming interface) to bridge the chasm between these domains. Once we have done this, we need a way of connecting a free running user program with this interface in a meaningful and consistent way (the simulations need to run identically each time for the same program). We can then produce a simple API for the user code to interact with the simulation. So, let's build that HDL to software bridge.

## The PLI

VProc, as at the time of writing, supports both the PLI 1.0 and the VPI (PLI 2.0) for Verilog, and uses the ModelSim FLI for VHDL. I have documented the mapping of C functions to system tasks or foreign procedures in a previous article, using these interfaces, and so I won't go into detail here as we are concentrating on how to put together a virtual processor. So, we will stick to Verilog and PLI 1.0 as being the simplest (and the original implementation)  and summarise how this is used in VProc.

All of the programming interfaces are C based, and so all of the VProc software is also C based. It needn't have been as only C linkage to the functions mapped to the system tasks is required but, as mentioned before, early deployment was being surrogate embedded processors co-simulating C code, or co-simulating with Kernel code (which is necessarily C). So that was the decision. This doesn't stop user code being C++, and the only requirement is that the user code entry point function has C linkage.

As we have already defined the Verilog system tasks above, we have a specification for our PLI C functions as there needs to be a one-to-one correspondence. As before, we only want to concentrate on the `$vinit` and `$vsched` tasks (the mapping process is the same for `$vaccess` and `$vprocuser`) and these need equivalent C functions. For PLI 1.0 these have the following prototypes:

```
int VInit (void);
int Vsched(void);
```

You'll notice that the functions don't have any arguments matching the arguments of the Verilog system tasks, but we can get hold of these using provided functions from the PLI. The system tasks arguments are indexed from left to write, starting from 1 (with index 0 being the task name itself). Two functions are provided to read and write these integer arguments. Namely `tf_getp` and `tf_putp`. The first takes a single index argument and returns the value of the system task argument at that position. The second takes an index argument followed by an integer value and updates the system task argument in that position with the new value. So, we now have a template for constructing C functions to map to Verilog system tasks and accessing their arguments.

```
int myPliFunction(void)
{
   int arg1 = tf_getp(1);
   int arg2 = tf_getp(2);

   // Do some stuff

   tf_putp(3, newVal);
}
```

We still have some magic to do to associate the C functions with the tasks. A table is constructed with entries of a PLI defined type, `s_tfcell`. The PLI table for VProc looks something like that shown below:

```
s_tfcell veriusertfs[] =
{
    {usertask, 0, NULL, 0, VInit,     VHalt, "$vinit",     1},
    {usertask, 0, NULL, 0, VSched,    NULL,  "$vsched",    1},
    {usertask, 0, NULL, 0, VAccess,   NULL,  "$vaccess",   1},
    {usertask, 0, NULL, 0, VProcUser, NULL,  "$vprocuser", 1},
    {0}
};
```

The table has an entry for each system task and is terminated with a null entry. Each is defined as a user task (there are other types for functions), and after a few fields we don't care about, the C function is listed. The next fields we can gloss over (`VHalt` is called when the simulation terminates), and then a string associates the system task with the C function. The final argument is actually a don't care and can be ignored.

Having constructed the table it needs to be registered with the simulator in a function which must be called `bootstrap`, which returns the pointer to the table.

```
p_tfcell bootstrap
{
    return veriusertfs;
}
```

If the table and this function is compiled in with our software, then the mapping between the Verilog system tasks and our C functions will take place. All of the definitions for the PLI types and functions are accessible by including a header `veriuser.h` which will be provided by the simulator along with a PLI library. We now have our mapped C functions and can interact with the arguments of the system tasks.

Points 1 and 2 of the original rough specification are now fully completed. We must now put functionality in the mapped C functions to complete the rest of the specification.

## The Simulation Side C functions

Point 3 of the specification states that a user program must run freely and so our mapped C functions `VInit` and `VSched` don't meet this criterion as they stand, as they are called from HDL and return at each call of the system task and always start from the beginning of the function once more. One way we can have free running user code is to run the code in a separate thread. How we then communicate between this concurrent user thread and the simulation PLI C functions we will discuss later. For now we need to be able to start a new thread with the user code. Point 5 of our specification states that we must be able to have multiple instantiations of the VProc component and each run their own program. So we can't have a 'main' entry point for our user code as each virtual processor must have a different entry point and, in any case, the simulator executable will have a main function. Thus a choice was made to have an entry point name `VUserMain`*N*, where *N* is the node number for the particular virtual processor. So a VProc with a `Node` input of 0 will run a 'main' function `VUserMain0`, which is supplied by the user. The use of different 'main' entry points is not unique to VProc. For windows graphical programs, for example, the entry point is called `WinMain`.

At the start of the simulation the user thread needs to be started, and this is what calling `$vinit` in an initial process, with a node number argument, does. The `VInit` C function is called, gets the node number, and initialises some state for the particular node. The details aren't important just yet, but a table of node state is created (with entries of type `SchedState_t`). This contains state for synchronisation (more later), some structures for passing information between the user thread and the PLI C functions and tables for the interrupt and user callback functions. Once this is created and initialised it calls a `VUser` internal function which does the actual thread creation. It is passed the node number and completes some initialisation of the specific node state structure. It then creates a new thread using an internal function `VUserInit` as the code to start running in the new thread. As the code is in C, this uses the `pthread_create` function (include `pthread.h`). This would all be easier in C++, but for reasons discussed before we chose C, and so this is what we need to do. The call looks like that shown below:

```
pthread_t thread;
pthread_create(&thread, NULL, (pThreadFunc_t)VUserInit, (void *)((long)node);
```

The last argument allows all sorts to be passed to the thread, but the node number is cast here so that it is passed to the `VUserInit` function. The `VUserInit` function isn't our user code but an internal function, and it is responsible for calling the user code. Jumping a head a little, to run C code (including our PLI functions) in a simulator, these must be compiled into a shared object (or dynamic linked library in Windows speak). What this means is that they are loaded at run time. So, when we compile the VProc software, the user entry functions aren't available as static symbols and need to be 'looked up' at run time. Under Linux, the `dlfcn.h` header provides access to functions for dynamically loading shared objects and for looking up function names which can then be called. It is different under Windows but the VProc software maps the relevant windows functions to the Linux equivalents, so the code looks like Linux code, and that's what we will discuss.

The `VUserInit` function is now running in a separate thread from the simulator. All the other functions mentioned previously are part of the main simulator thread, and we need to bear this in mind. When `VUserInit` is started it uses a function `dlsym` to look up the user entry point function. The `dlsym` function takes a pointer to a handle (but we can use a default and pass in `RTLD_DEFAULT`), and a string with the name of the function we want. It then returns a pointer to that function. So `VUserInit` constructs a string as "VUserMain" plus the passed in node number as a string. E.g., "0", giving "VuserMain0". This is looked up using `dlsym` to get its pointer and we are now able call this function. Just before this it waits for the first message from the simulator using the synchronisation method discussed later. This is done to firstly ensure that the simulation side has reached a certain point and to handle the initial read data offset. I mentioned above that reads are one `$vsched` call offset from writes, as the read has to happen to return the data and is done on the next call. The first read data is garbage, and so is discarded here. Now the user program is called (e.g. `VUserMain0`), and so is now running in the thread created in `VUser`, via `VUserInit`. This will happen for each VProc instance, with separate threads for each user program and separates node state.

People will tell you of the dangers of threads and communication between them, and non-thread safe functions etc. etc. If you are a software engineer and reading this, you will probably think that just with some knowledge and care and this is all fine. If you're a logic engineer, then think of this like clock domain crossing. It's a hazard, but simply avoiding it may not be an option. Some knowledge and care allows this to be done safely, and so it is with threads in VProc. Firstly, though, we must provide an API to the user code so it can interact with the simulation.

## The User Side API

The basic API provided to the user code consists of a small set of C functions as shown below, with some `const` keywords missing to declutter the prototypes:

```
int  VWrite       (unsigned addr,  unsigned    data, int      delta, unsigned node);
int  VRead        (unsigned addr,  unsigned   *data, int      delta, unsigned node);
int  VTick        (unsigned ticks, unsigned    node);
void VRegInterrupt (int      level, pVUserInt_t func, unsigned node);
```

There are some other API functions not listed for burst reads and writes, and to register a callback function for the `$vprocuser` system task. The first two listed are fairly self-explanatory I hope, with a write function to pass in an address and some write data. The read function is also supplied an address and a pointer to a word variable in which to return the read data. Common to both are a `delta` argument and a node number. The delta argument is either 0 for a normal accesses or `DELTA_CYCLE` (-1) for a delta cycle access. The node number must match the node of the VProc instantiation in which the code is virtually executing.

### *Advancing Time*

Since time will only move on in the simulation when an access is active (the simulation is blocked when it calls a `$vsched` task until a new access is ready) all the read and write accesses would be back-to-back with no gaps between them. Therefore there is also a `VTick` function which allows an 'access' that does not read or write but ticks for the specified number of clock cycles. Recall from the HDL that the $vsched can update a VPTicks integer, and this will then create a count down, by-passing the next call to `$vsched` until it reaches 0. The `VPTicks` API function will set the ticks value and have the read and write strobes at idle. The `VPTicks` function is like a sleep function (in units of clock cycles) but can also be used to insert random delays between accesses. Imagine writing some wrapper functions around the read and write API functions that calls `VTick` with some random value over a range (from 0 to 20 cycles, say) before calling the appropriate read or write API function. This would then emulate 'processing time' that you might expect a real processor to have between loads and stores. And this is an important point—the code running on the virtual processor, apart from the API calls, is running infinitely fast with respect to simulation time. Time must be advanced with API calls or the simulation will lock. If your code had a `while(1);` line of code then in a real processor it would still execute going around this loop, but in the virtual processor this will lock the simulation as time is not advancing. (I mean you can do something this daft in behavioural HDL as well.)

### *Modelling Interrupts*

The final API function is the `VRegInterrupt`, which takes a `level` argument and a pointer to a function, as well as node number. This associates the function with the interrupt level. This function will be called if the VProc's component Interrupt port is at that level in any given clock cycle. It is not necessary to provide a function for all possible interrupt levels, or even any at all. The VProc code will simply return having called no user function for that interrupt level. The functions that can be registered with the interrupt level have a prototype of `int myIntFunc (void)`. The function can return a non-zero value if it wants to override any tick countdown value outstanding or set up a new one—though should be used with care.

An important point to note here is that the interrupt function called when its associated interrupt level is active *remains in the simulation thread*. It is a traditional PLI function that can be called from HDL and then return. Therefore there is an important restriction on what these interrupt functions can do. They cannot make calls to the read, write or tick API functions. This seems like a dire restriction, as it would be useful if they acted like interrupt service routines (ISRs) which would definitely need to do reads and writes. So why was it done this way?

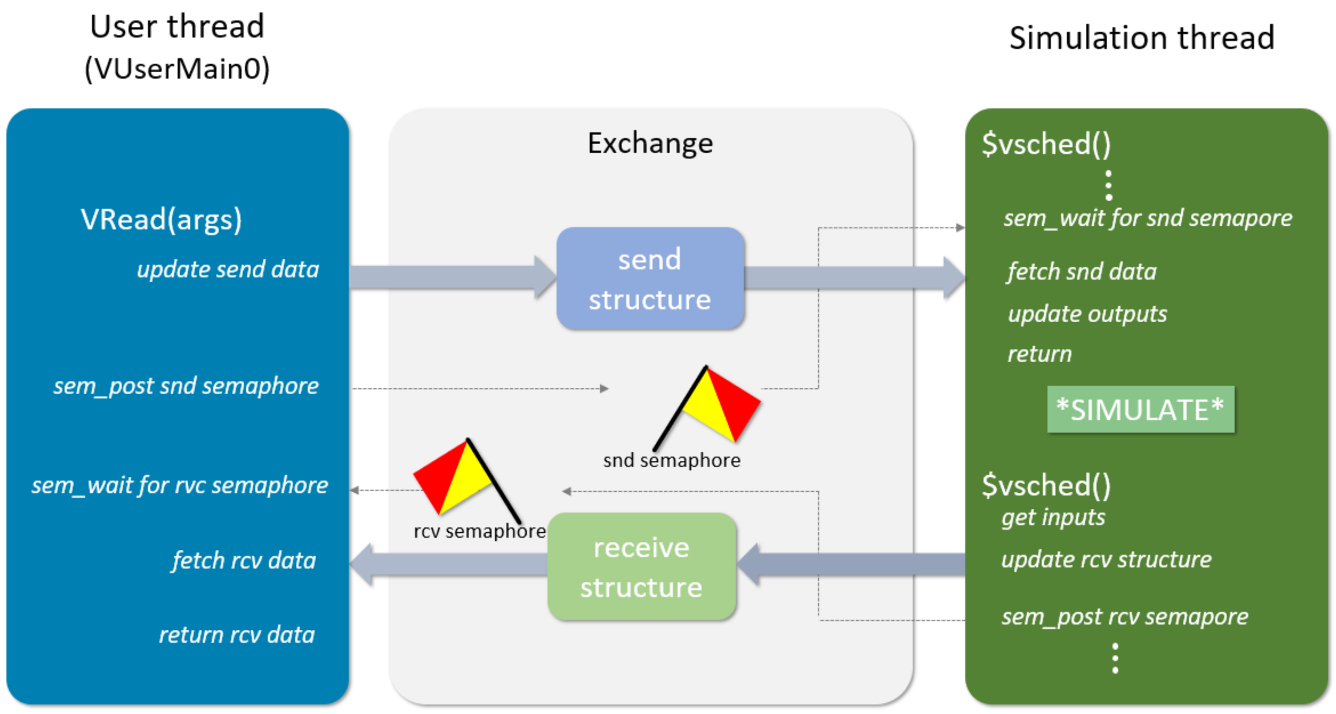
Well, in order to allow interrupts to happen in any cycle (and an edge triggered interrupt might be active for just one cycle), even whilst a transaction is in progress, we must impose this restriction otherwise the active transaction would be overridden by any new accesses made in the interrupt function. The way this is meant to be used is that the interrupt function simply stores the fact that an interrupt has happened (the 'event'), perhaps via a queue or some other state, which is available for the main user program to inspect at regular intervals—perhaps before any call to a read, write or tick API function—and then call its own ISR functions from there. It could also model interrupt enables and priorities in any way that it sees fit. I discuss this in detail in a blog with interrupts for the OSVVM co-simulation features which, though not a virtual processor, details how user code might be constructed to handle interrupts in just the same way (the underlying technology is derived from VProc).

If thread alarm bells are ringing for you, then you probably spotted that I said that the interrupt functions are running from the simulation thread, but that they pass information to the user code for processing, which runs in its own thread. Indeed, multiple VProc instances would mean that there can be many threads, and it may be useful to exchange information between them as well. The synchronisation code of VProc is constructed to handle all these problems at once in such a way that the user code does not have to worry about the fact that threads are being used at all. The next section will discuss this mechanism.

## Synchronising Between User and Simulation Spaces

The synchronisation mechanism between the user code threads and the simulation PLI functions is really the key technology for VProc (and OSVVM co-simulation). Up until now we have spoken about behavioural HDL, mapping C functions to system tasks via the PLI and of user code running in threads. In any capable embedded system R&D lab all these individual systems would be known via the experience of logic, verification, and embedded software engineers. The key is putting all this together so that it reliably and consistently works. As one (very capable) engineer put it to me, when I showed him VProc running in a simulation environment "*Hang on! On the one hand the simulation looks like it's the master and on the other hand the user code looks like it's the master. How does this work*?". Well, we've been leading up to this and I want to show you how, as sharing techniques and methodologies with others is what I like to do (and always have) as it might be useful for them, or spawn new ideas and suggest improvements beyond what I have imagined. What I'm hoping is that, once I've explained it (if I've done my job correctly) you say to yourself "*Oh yeah, it's obvious when you think about it.*" I'm not claiming I got to this solution straight away or it's the only way but, from the start, I wanted the simplest method possible. After experimenting for a while I concluded I needed something that means that only one thread is ever unblocked at any one time. This will be true for the main simulation thread as well. If VProc can pull this off, it is safe not only to exchange data between the PLI functions and the user code, but also the interrupt functions and even between user threads.

The software engineers amongst you will already be thinking in terms of mutexes and semaphores. These are the bricks for synchronising between threads and for thread safe code. I chose to stick with just semaphores to keep things simple. The state for each virtual processor node contains two semaphores and two structures for exchanging data, a semaphore and structure pair for sending data and a similar pair for receiving data. The diagram below summarises what's going on.



When an API function is called from the user code, such as `VRead`, the software updates the send data structure for that node with the details (address, data out, ticks etc.), via an internal function called `VExch`. It will then 'post' to the send semaphore. Posting to a semaphore allows any code that's 'waiting' on that semaphore to proceed. The post and wait functions (defined by including the header `semaphore.h`) are:

- `sem_post(sem_t semaphore)`
- `sem_wait(sem_t semaphore)`.

For the send direction, the `VSched` PLI function waits on this send semaphore when `$vsched` is called from the HDL. The user side API function call then 'waits' on a receive semaphore and will block until the simulation code posts to it. When the simulation side `VSched` function is unblocked on the send semaphore posting, the `$vsched` outputs are updated with the data from the send structure and VSched, and hence $vsched, returns and the simulation starts running again to simulate the bus access. During this simulation time the API function called from the user code is blocked on the receive semaphore.

When the transaction is completed, `$vsched` is called once again. The value of the data input port of VProc is passed in to return any read data from a previous read access, and `VSched` will update the receive structure with the input data and any interrupt state, and then will post to the receive semaphore. It will then, once again, wait on the send semaphore. Since the receive semaphore was posted to by `VSched`, the user API call waiting on it is unblocked and it can fetch the input data and return it (if a read access). The interrupt data isn't returned, but any user registered functions are called for the appropriate active level in `VExch`, and the send semaphore posted without a new API call.

Hopefully you can see, for one VProc instance, that the semaphores guarantee that either the simulator is blocked, on the send semaphore, or the user thread is blocked, on the receive semaphore. So how does this map with multiple VProc instances? Well, whenever the simulation calls a system task it naturally blocks until that call returns. With multiple VProc instances, in a given cycle, it will have to process all calls to the `$vsched` tasks before it can advance simulation time to the next time slot. There's no guarantee which order it will call the system task for each VProc, but it will definitely call all of them, and will block on each one, so that no other one is active. This means that all the threads are inactive accept the VProc instance where `$vsched` was called, and the VProc synchronisation methods ensures that only the simulation side or user side code is active for that VProc node. Therefore, it is safe to use shared memory and structures between all the various components in the system without the need to manage the inter-thread communications with mutexes and semaphores etc. Of course, if within a particular virtual processor's user code, threads are spawned and need to communicate with each other, then this must be managed as it would for any multi-threaded software.

## Notes on Compiling and Running Code

I won't dwell on this too long, but some things to note on compiling the code for VProc—both the user code and the VProc software. A normal program might be compiled into an executable and run from the command line. Since the executable we will be running will be the simulator, we can't do this for the user programs. Instead, all the code must be compiled into a shared object which the simulator will load at run time. For `gcc` this is done using the `-shared` command line option. Libraries will also need to be linked, such as the PLI library supplied by the simulator, and as the VProc software uses posix threads and real-time dynamic linking loader libraries, these all need linking too. Make files are provided with VProc, and the resultant compiled code is generated into a `VProc.so` shared object file, which includes all the VProc software as well as the user code.

The simulator needs to be told that it needs to load this shared object which is usually done with a command line option and the name of the shared object file. For ModelSim this looks something like:

```
vsim -pli VProc.so test
```

Thus we can compile our code and when the simulation is run this will be loaded and run on the VProc virtual processor instances.

## Conclusions

Does the virtual processor meet the original requirements that were stated? We have an HDL component that looks like a processor core with a generic memory mapped bus master and an interrupt input. We've seen how we can use the PLI to bridge the HDL software divide, but VPI and FLI are also available. We used threads to spawn freely running user code with entry points for each VProc instantiation. A simple API was defined to allow reads and write over the generic bus and to pass simulation time, and a means to have a user function called on an active interrupt. We can have multiple VProc instantiations, as each has a unique node number with its own state structure, calling the specific user code entry function and a mechanism to synchronise between the user code and the simulation such that thread safety is taken care of by the VProc software and data exchange between user code and interrupt code is possible without further intervention. So, I believe all requirements are met with this.

In this article I have looked at how VProc works. The series of articles I wrote on co-simulation (gathered into a PDF here) show how the basic ideas presented here can be taken much further to produce software based VIP models, co-simulate real embedded software in a logic simulation and even use the ideas to extend a verification library, like OSVVM, to give these same possibilities whilst in a fully-fledged, widely used and mature verification methodology with all the features that it brings.

Is VProc the only method to do this kind of thing? Not at all. At a transaction level one can use SystemC and its TLM specification to model a logic system and co-simulate with software. Verilator can compile Verilog to C++ or SystemC, and thus can be hooked up to embedded software as well. (VProc code has been embedded in Verilator code in the past, as it happens.) The cocotb co-routine system allows python code to run with a logic simulator, and it doesn't take much imagination to envisage a virtual processor component running with this, with python driving a memory mapped bus. So, what does VProc bring that is any different? Perhaps nothing, depending on what problem is to be solved. What it does do is provide a solution (and a concept) that is fast to execute, only uses standard interfacing to the simulator giving maximum compatibility with both commercial and open-source simulators without the need to use any other simulation kernel and has a clear path to full co-simulation with embedded software with all the facilities that EDA tools can bring. Not covered here, but VProc also has a C++ API wrapper class, allowing user code in C++ with easier interfacing. In addition, the software running on the virtual processors can be debugged using gdb, and hence via an IDE such as Eclipse. In fact, dual debugging of software with an IDE and logic in a simulator is possible, if a little tricky to get your head around.

The main aim of this article, though, isn't really to promote VProc as an alternative to other methods and tools, but a tutorial on what can be done with a little knowledge of behavioural logic, the programming interfaces, and some minimal software knowledge, and how these can be synthesized together into something much greater than the sum of its parts. I hope you've learnt something from this article and that some of the ideas presented here can be useful to you, perhaps in new and interesting ways well beyond what VProc originally set out to do.