# Virtual Processor (VProc)

by Simon Southwell
19th June 2010
(updated 5th May 2021)

---

## abstract

A co-simulation element is presented which abstracts away the communication between a user C interface and a verilog or VHDL task based interface, wrapped in a simple behavioural HDL module, over the PLI of a simulator, as a building block for a processing element, whose code is a normal C program written and compiled for the host computer, controlling bus type operations in the simulation environment. This 'virtual processor' (VProc) element has a basic memory type interface in the HDL domain, and a simple API in the C domain, allowing read and write operations and support for simple interrupts. The interfaces are kept as simple as possible, but descibed is the means to build, on top of this fundamental co-simulation element, any arbitrary complexity of bus interface (e.g. AHB or PCI-Express) which can then be controlled by a user written program, compiled for the host computer.

---

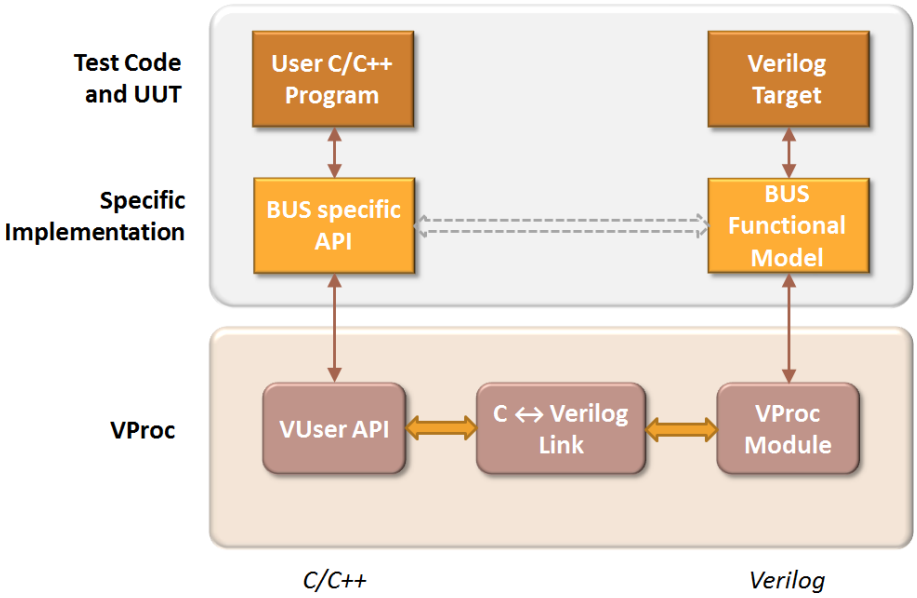
# Introduction

The concept of a virtual processor is far from new, as is the concept of hardware/software co-simulation. Tools such as V-CPU from Summit or Seamless from Mentor Graphics, give a professional way of running software targetted for a processor or system under development early in the design cycle. Although these tools are mainly aimed at running specifically targetted processors (ARM, Mips, etc.), some have the ability to run in 'host' mode (such as V-CPU), as a true 'virtual processor'.

In this context, a virtual processor is a means to run host compiled programs as normal programs in the OS environment, with all its facilities available, which are the control source for a bus of an associated instantiated HDL module in a simulation environment running on the same machine. This has the advantage of not relying on cross-compilers in order to introduce a processing element in a simulation; either to act as a test harness stimulus, or to replace an, as yet, non-existing embedded processor. With the software on the virtual processor written in C or C++ (or with suitable C style linkage) and a suitable I/O access API layer, early development of code in this environment feeds forward directly to the actual target processor with minimal changes.

The 'VProc' package is a thin *'veneer'* to enable the development of a 'host' virtual processor system and provide a basic co-simulation environment. It isn't necessarily meant to be an end in itself, but hides the co-simulation communication complexities for easily and quickly constructing a virtual processor environment, with the bus functional capabilities desired. Enough has been done to set up a link between one or more virtual processors and user written programs, compiled into the simulation executable. A very basic API is provided to the user code to allow control of the HDL environment. What's not provided in VProc is the actual bus functional models (BFMs) for specific protocols and their corresponding API software layers. This is left for the developer. But this also means that any arbitrary protocols can be supported, including proprietry ones. Enough functionality is provided at the C/HDL boundary to support (it is believed) any arbitrary protocol that one may conceive.

Where the boundary is chosen between functionality in C and BFM support in HDL is not restricted by this model. At one extreme, the pins of an HDL component can be mapped directly in to the VProc address space and the user C code controls the pins directly each cycle. At the other extreme, a complex HDL model for a given bus has, say, control registers mapped into the VProc, which simply reads and writes them to affect complex bus behaviour. The choice is left to the implementer, and maybe affected by what's already available in terms of source code and HDL models, as much as by the preference of the designer, or speed requirements of the model. For example, a PCIe model might simply have its 10 bit wide, pre-serialised, lanes directly addressable by the virtual processor. Only serialisation is left to do in HDL (if required)—all the rest of the PCIe protocol could be handled by the user program associated with the VProc. On the other hand, a Virtual AMBA Bus Processor used in a test harness might utilse an already existing HDL interface block to connect to a transactional bus model which was then controlled by mapping much simpler input signals into VProc.

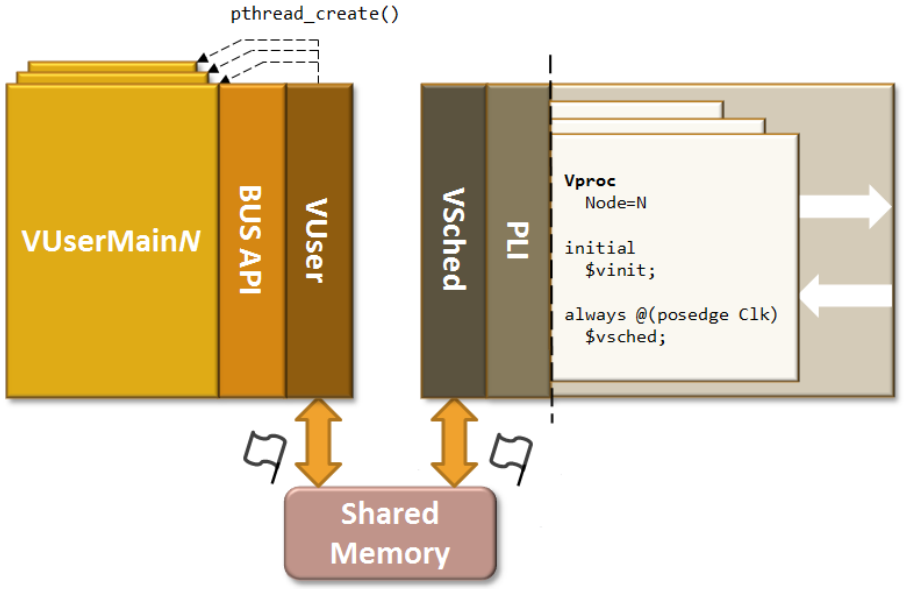



The above diagram shows what a typical virtual processor system set of layers might be. The bottom layers are provided with this system. On top of that a BFM in the HDL domain and an equivalent Bus API in the C domain are constructed to initiate bus transactions for the given targetted protocol. This system can then be used to run a user program which communicates with the target simulation.

Currently with this system up to 64 virtual processors may be instantiated in a single simulation. The link between the user programs and the simulator is implemented as messages passed via semaphore protected shared memory. The interface between the HDL domain and the C environment is via the standard verilog PLI interface or the VHDL foreign model interface (FLI), supporting it both in languages. It could just as easily have been via VCS's DirectC, or some other method provided by the simulator (e.g. direct instantiation in SystemVerilog).

Each verilog module has access to two main 'tasks' in C (still within the simulation's process). `$vinit` is used to initialise a node specific message area and initiate communication with user code for that node in a new thread. Each VProc module calls `$vinit` once at time 0, passing in a node number. The VHDL equivalent calls are to `VInit` and `VSched`. For the rest of the document the verilog is discussed, with the VHDL equivalent calls over the FLI implied, unless mentioned specifically.

Communication between the HDL and the user code is done using calls to `$vsched`. The bus status is sent with the node number, and routed to the relevant user thread's message area and a semaphore set. The simulation code then waits for a return semaphore. The user code, waiting for a simulation message, responds with a message in the other direction, with a node number and a new bus state plus a tick count, clearing the incoming semaphore and setting its outgoing semaphore. Verilog state is updated via the PLI, and the module waits for the number of ticks to have passed, or an IO transaction is acknowledged, before calling $vsched again.



# Bus Structure

In the bundled VProc package (see download section) a simple memory bus module is provided to illustrate the use of the VProc elements. This can be used as the starting point for connection to a BFM, but the VProc tasks can be used seperately in a user defined module wrapper if desired. The verilog code is `f_VProc.v` in the top level VProc directory, with the software source in `code/`, and the VHDL is `f_vproc.vhd` and `f_vproc_pkg.vhd`.

The bus interface of the provided VProc component is kept as simple as possible. Any complexity required for a particular protocol must be added as a wrapper around this simple interface, or a new wrapper created. The IO consists of a data write and data read bus. The width of the data busses is 32 bits. A 32 bit address is provided for the read and write accesses, with associated write and read strobes (`WE` and `RD`). Acknowledgement inputs for the accesses are `WRAck` and `RDAck`, which will hold off returning to the user code until active (which could be immediately).

There is an interrupt input which causes an early call to `$vsched` when active. Normally `$vsched` is only called when the tick count has expired, (set when the previous $vsched call returned) or when an IO call is acknowledged. An interrupt causes `$vsched` to be called, flagging the interrupt as the reason for the call, and allowing an interrupt message to be sent to the relevant user thread. The return message would not normally update the tick count (though it can), so that the original scheduled call still gets invoked.

When a simulation 'finishes' (i.e. `$finish` is called, or the simulation is quit), then `$vinit` is effectively called in the background. At this stage, the user threads are shut down and the simulation cleanly exited.

The node number is also passed in to uniquely identify each VProc. If two nodes have the same node number, then undefined behaviour will result. The module definition is shown below:

```
module VProc (Clk, Addr, WE, RD, DataOut, DataIn,
              WRAck, RDAck, Interrupt, Update, UpdateResponse, Node);

input         Clk, RDAck, WRAck, UpdateResponse;
input  [2:0]  Node, Interrupt;
input  [31:0] DataIn;
output [31:0] Addr;
output [31:0] DataOut;
output        WE, RD, Update;
```

The Update output port is a special signal which allows multiple reads and writes within a single delta cycle; useful when trying to construct or read vectors wider than 32 bits. The port changes state (1 to 0 or 0 to 1) whenever the outputs change. By waiting on this event, and updating local register values each time it transistions, large vectors may be processed in a single delta cycle. For example, suppose the VProc module is instantiated with a wire '`Update`' on the port, then an '`always @(Update)`' can be used to instigate a decode of the new address and strobes and update or return sub-words of a large vector. Control of whether multiple `Update` events occur in a single delta cycle is affected by a parameter into the access C procedures (see Writing User Code below). The Update signal needs a response to flag that the Updating is complete, via the `UpdateResponse` input. This would normally be an external register that is, like `Update`, toggled 1 to 0 or 0 to 1, at the end of the update—say at the end of the `Update` always block. It is not necessary to use the `Update` event signal if delta time access of wide vectors is unnecessary. A normal inspection of the strobes on an 'always @(posedge Clk)' (say) will work just fine, but then `VWrite()` and `VRead()` must never be called with `Delta` set to 1 (see below for details of this)). If delta updating is not required, then the `UpdateResponse` input should be directly connected to the `Update` output as the VProc module will suspend until the response comes back.

## PLI syntax

The actual syntax of the PLI task calls are not important if using the provided memory bus BFM (see above), as this is taken care of by the verilog in the VProc module. So this section can be skipped if the memory bus BFM is to be used unmodified, but if wishing to create one's own wrapper module, then the PLI tasks of VProc are as follows:

```
$vinit     (NodeIn)
$vsched    (NodeIn, InterruptIn, DataIn, DataOut, AddrOut, RWOut, TicksOut)
$vprocuser (NodeIn, ValueIn)
```

The VHDL has FLI equivalents of `VInit`, `VSched` and `VProcUser`, defined in the `f_vproc_pkg.vhd` file. All the arguments to the PLI tasks are `integer` type, for simplicity of interfacing to C. Vectors of verilog signals can still be passed in (including padding with 0's), but care must be taken as an x or z on even a single bit causes the whole returned value to be 0. This can be hard to debug, and so checks should be made in the verilog code. For VHDL, the arguments must be converted to integers first, which is done in f_vproc.vhd.

The `$vinit` task is usually called within an `initial` block. The `Node` input needs to be a unique number for the VProc instantiated. If called with a constant or a module parameter, then this can be at time 0 within the initial block. If it is connected to a port or wire, even if ultimately connected to a constant, then a small delay must be introduced before callin `$vinit`, as the call to the PLI function and the wire assignment ordering is indeterminate.

The main active task call is to `$vsched`. It is this call that delivers commands and data, and has new input returned. It too has a node input and, in addition, an interrupt and data input. The interrupt is an integer value, but the valid range is from 0 (no interrupts) to 7. The data input is a single 32 bit value. `$vsched` is called for every input update required, or expired tick. On return, updated DataOut and AddrOut values are returned, along with a directional RWOut. These map easily to the memory style interface of the VProc module, but can be interpreted in any way. E.g. AddrOut is a pin number, the DataOut value is the pin value (including x or z, say), and the RWOut determines whether the pin value is just read, or both read and updated.

The `TicksOut` output is crucial to the timing of the `$vsched` calls. In normal usage this should update a timer counter, which then ticks down until 0, when `$vsched` is called once more. So usually a value equal or greater than zero is expected. For a usage where communication is required every cycle, then this would be zero (i.e. no additional cycles), but it can be an enormous value (up to 2^31 - 1) if the software wishes to go to sleep. If `$vsched` was called on an interrupt, then a new value of TickOut is returned, where a value greater than zero can be used to override the current tick count, or leave alone if 0. This allows a sleeping VProc to wake up on interrupt. A value of less than 0 can be returned (for non-interrupt calls), indicating that the call is for a "delta cycle" update, and another call is expected before advancing time in the simulation. This allows multiple commands/updates to affect state before advancing the clock. Only when a TickOut value of 0 or more is returned should the code cease calling `$vsched` and processing the commands.

The `$vprocuser` syntax is a straight forward set of a node number and single integer value. The value is not interpreted and is passed straight on to the registered user function (if one exists).

For those interested in following this up further with an example usage, then look at the `f_VProc.v` verilog and `Pli.tab` (or `VSched_pli.c`) code provided for the memory interface BFM, or the `f_vproc.vhd` and `f_vproc_pkg.vhd` VHDL equivalents. Only the PLI 1.0 task/function (TF) routines are used in the verilog virtual processor to simplify and speed up the interface. I.e. all communications between C and verilog are only via the arguments to the PLI tasks. The same is tru for the VHDL FLI calls.

Of course, the PLI tasks are only one side of the interface, and they are connected indirectly to C API functions. These are described in the next section.

# Writing User Code

The user code that's 'run' on a virtual processor has an entry point to a function whose prototype is as follows:

```
void VUserMainN(void);
```

The *N* indicates the node number of the virtual processor the user code is run on. At start up, the initialisation code in the API will attempt to call a user function of this form. So, for instance, if there is instantiated a VProc with node number 8, VUserMain8() is called. There must be one such function for each instantiated VProc, otherwise a runtime error is produced, and, of course, each instantiated node must have a different node number from all the rest.

When in a VuserMain*N*() function, the user code thus has access to some functions for communication to and from the VProc bus in the simulation, which appear as if defined as below:

```
int  VWrite        (unsigned int addr, unsigned int  data, int delta, int node);
int  VRead         (unsigned int addr, unsigned int *data, int delta, int node);
int  VTick         (unsigned int cycles, int node);
void VRegInterrupt (int level, pVUserInt_t func, int node);
void VRegUser      (pVUserCB_t func, int node);
void VPrint        (char *format, ...);
```

In order to access these functions, `VUser.h` must be included at the head of the user code file.

## Input and Output

If the user code needs to write to the VProc's bus, then `VWrite()` is called with data and address. The function will return after the simulation has advanced some cycles depending when the write acknowledge is returned in the VP module input and a status is returned. The status is actually the `DataIn` value on the `$vcsched` parameters, and so is implementation dependant, depending on the address decoding arrangements outside of VProc. The Delta flag can override the advance of simulation time or the waiting on an acknowledgement and performs the write in the current simulation delta time. This can be employed when it is required to write to a

number of separate registers to form, say, a wide, bus specific, transaction which may need to be issued once per clock cycle. Setting `Delta` to 1 for all the writes except the last one allows words greater than 32 bits to be constructed in a single cycle. Care must be taken that `VWrite()` (or `VRead()` for that matter) is not always called with `Delta` set to 1, as this may result in registers being overwritten without simulation time being advanced. It is also necessary to have support for this feature in the verilog wrapper around VProc, using the 'delta' event to instigate external register accesses instead of the more normal clock edge (see Verilog Bus Structure above).

Similarly a read is invoked with `VRead()`, which returns an arbitrary number of cycles later since the simulation is waiting on a read acknowledge. The 32 bit read value is returned into the variable pointed to by the second argument. A `Delta` input acts in the same way as writes, allowing wide data reads without advancing simulation time, or waiting on an acknowledge.

## Advancing Time

If the user code isn't expecting to need to process any IO data for some time, it can effectively go to sleep for a number of clock ticks by calling `VTick()`, passing in a cycle count. This should be used liberally to allow the simulator to advance time without the need for heavy message traffic polling on a register status (say). Interrupts can be used to flag events and wake up the main process if the wait time is arbitrary or indeterminate (see below).

## Interrupts

There are (currently) seven levels of interrupt. For every cycle that a VProc's `Interrupt` input is non-zero, a call is made to a previously registered function. To register a function `VRegInterrupt()` is call with the interrupt level (1 to 7) and a pointer to a function. The function pointer must be of type `pVUserInt_t` (i.e. a pointer to a function returning an `int`, with `void` argument list). The user interrupt function must return an integer, which is normally 0, but may be > 0 if the interrupt function decides it wants to override the schedule count of the outstanding IO or Tick call. A runtime error is generated if the interrupt level is one which has no registered interrupt function.

In this release, the Interrupt routines cannot make calls to the IO routines (as this will break the scheduling). They are intended only to flag events, which, if IO status is required, the main routine can service. It should still possible to emulate full interrupt service routines, even with this limitation. For example, if the `VRead()` or `VWrite()` calls are wrapped inside another procedure, interrupt event status can be inspected each time they return, and a call to a handler made if required, based on status updated by a call to the inetrrupt handler. This handler would now be part of the main thread, and can safely make IO accesses. This implies that simulation IO calls are atomic, and won't be interrupted (unless the interrupt function changes the pending tick count, which affectively cancels the IO, which would need re-issuing). By adding this slight complication in the user code, the VProc implementation becomes much simpler.

## User Callback

As well as registering interrupt callback functions, a general purpose callback function may be registered using `VRegUser()`. It is similar to `VRegInterrupt()`, but requires no 'level' argument. Registering a function in this manner does not automatically attach it to an event, but instead attaches it to a defined verilog task '`$vprocuser(node, val)`'. To invoke the registered function $vprocuser is called with the node number corresponding to the VProc instantiation running the user code. A value is also specified and is passed in as an integer argument to the user function. This could be used to select between different functions, depending from where the task is called. Example uses might be to call the function at regular intervals to dump debug state, or to call when about to finish to do some tidying up in the user code etc. It should be noted that the registered function is synchronous to the simulator thread, and *not* the user thread. Therefore, if the function is to communicate with the main user thread it must do so in a thread safe manner.

The registered user function must be of type `pVUserCB_t`, which is to say a function returning void, with a single integer argument. E.g.:

```
void VUserCB (int value);
```

If `$vprocuser` is invoked for a given node before a callback function has been registered, then no effect is seen and the task exits. It is therefore safe to invoke the task even if the user code never registers a function. However, it is not safe to invoke the task for a non-existent node, and undefined behaviour will result.

## Log File Messages

The `VPrint()` function (actually a macro) allows normal `printf` type formatted output, but sends to the simulation log output (and thus to any log file being generated) instead of `stdout`. This makes correlation

between verilog log data and user code messages much easier, as they are sent to the same output stream, and appear in the correct relative order.

# Adding C functions to Simulators

Each of the simulators treats PLI code slightly differently, in the way it is linked as a shared object/DLL. Three simulator examples are documented here.

## VCS

The internal C functions called by the VProc modules' invocation of `$vinit` and `$vsched` (`VInit()`, `VSched()` and `VHalt()`) are compiled into the verilog during normal simulation compilation, along with all the user code and any BFM support code Simply add references to the list of `.c` files somewhere in the vcs compile command. When compiling `VSched.c` for VCS, then 'VCS' must be defined, which will usually be the case if compiled directly from the VCS command line. If compiling to an object first, the use `-DVCS`. This contains the PLI code as well, which the simulator must be informed about with a PLI table file (`Pli.tab` provided), and indicated in the command line with the `-P` option. E.g.:

```
vcs  *.c -P Pli.tab -Xstrict=0x01
```

Needless to say, the user code, `VSched.c` and `Pli.tab` files need to be in the compilation directory, or the above modified to reference the files remotely. The `-Xstrict=0x01` is a VCS requirement for using threads in PLI code, and the `pthread` and `rt` libraries must be compiled in (e.g. use the `-syslib` option).

If `VProc`is being used as a library component in a system which has additional PLI functionality, then the system's `Pli.tab` must be extended with the entries in that supplied by `VProc`.

## NC-Verilog

Adding C functions to NC-Verilog is slightly different to VCS. Firstly the PLI code must be compiled as a shared object. In our case, this includes all the user code as well, compiled into a file `VSched.so` (e.g. use `-fpic -shared` options with `gcc`). The PLI table, unlike for VCS, is compiled in as an array, so no extra table file is required. To access this code the `+ncloadpli1` command line option is used at compile time. E.g.

```
ncverilog <normal compile options> +ncloadpli1=VSched:bootstrap
```

The pthread and rt libraries must also be linked with the shared object.

If `VProc` is being used as a library component in a system which has additional PLI functionality, there can only be one set of '`veriuser`' tables and boot functions. The system using `VProc` must extend its own `veriusertfs` table to include the `VProc` entries. The `VSched_pli.h` header includes definitions for these entries as `VPROC_TF_TBL`, along with the number of entries (`VPROC_TF_TBL_SIZE`). The new system's table can then look something like the following:

```
s_tfcell veriusertfs[VPROC_TF_TBL_SIZE + MY_TBL_SIZE] =
{
    VPROC_TF_TBL,
    <my table entries>,
    {0}
};
```

When building `VProc`, a static library of the functions is generated as `libvproc.a`, for linking with the new system. This does not contain a compiled object for `veriuser.c`. The new system must provide this functionality if adding additional PLI functionality, extended as just mentioned. If no new functionality is to be added, the `VProc` code can be used, and `veriuser.c` from `VProc` compiled into a shared object, along with the code from `libvproc.a`.

Note: when linking library functions into a shared object (with `ld`) the `-whole-archive/-no-whole-archive` pair must bracket the library references in order to have the shared object contain the library functions. When using `gcc` to compile and link in a single step , the options become `-Wl,-whole-archive/-Wl,-whole-archive`.

## ModelSim

ModelSim compiles the verilog or VHDL, and runs the simulation seperately, with the commands `vlog` (or `vcom`) and `vsim` respectively. The PLI/FLI code is referenced at the running of the simulation. So, assuming the PLI/FLI C code was compiled as `VProc.so`, the `vsim` command is structed as shown below:

```
vsim <normal compile options> -pli VProc.so
```

As with NC-Verilog, the pthread and rt libraries must be linked with the shared object.

If the verilog `VProc` is being used as a library component in a system which has additional PLI functionality, there can only be one set of '`veriuser`' tables and boot functions. The system using `VProc` must extend its own `veriusertfs` table to include the `VProc` entries. The `VSched_pli.h` header includes definitions for these entries as `VPROC_TF_TBL`, along with the number of entries (`VPROC_TF_TBL_SIZE`). The new system's table can then look something like that shown for NC-Verilog, above. For VHDL the connections to the functins are defined in the `f_vhdl_pkg.vhd` file, and a separate table is not necessary.

When building `VProc`, a static library of the functions is generated as `libvproc.a`, for linking with the new system. This does not contain a compiled object for `veriuser.c`. The new system must provide this functionality if adding additional PLI functionality, extended as just mentioned. If no new functionality is to be added, the `VProc` code can be used, and `veriuser.c` from `VProc` compiled into a shared object, along with the code from `libvproc.a`.

Note: when linking library functions into a shared object (with `ld`) the `-whole-archive/-no-whole-archive` pair must bracket the library references in order to have the shared object contain the library functions. When using `gcc` to compile and link in a single step , the options become `-Wl,-whole-archive/-Wl,-whole-archive`.

# Delivered Files

In order to use the virtual processor, the following files are used.

| | |
|---|---|
| `f_VProc.v` | : Virtual processor verilog |
| `f_vproc.vhd` | : Virtual processor VHDL |
| `f_vproc_pkg.vhd` | : Virtual processor VHDL package defining FLI tasks |
| `VSched.c` | : Simulation (server) side C code |
| `VSched_pli.h` | : Common PLI definitions and prototypes |
| `veriuser.c` | : PLI table (NC-Verilog) |
| `Pli.tab` | : PLI table (VCS only) |
| `VProc.h` | : VProc layer definitions |
| `VUser.c` | : User (client) side C code |
| `VUser.h` | : User (client) side header file |
| `VUserMainT.c` | : Template for user code |

Note: If using NC-Verilog, the compiled shared object, `VSched.so`, must be available in the invocation directory.

VProc is released under the GNU General Public License (version 3). See `LICENSE.txt` in the downloadable package for details (see VProc Download section to access package).

Along with the above files are delivered example makefiles for compiling the C and verilog. These have been tested in a specific environment and are for reference only. Adaptations will need to be made to the local host environment, and no guarantees are given on their validity. A simple test example is also bundled, with a basic random memory access from one VProc, and an interrupt generation from another. Again, as for the makefiles, this is for reference only.

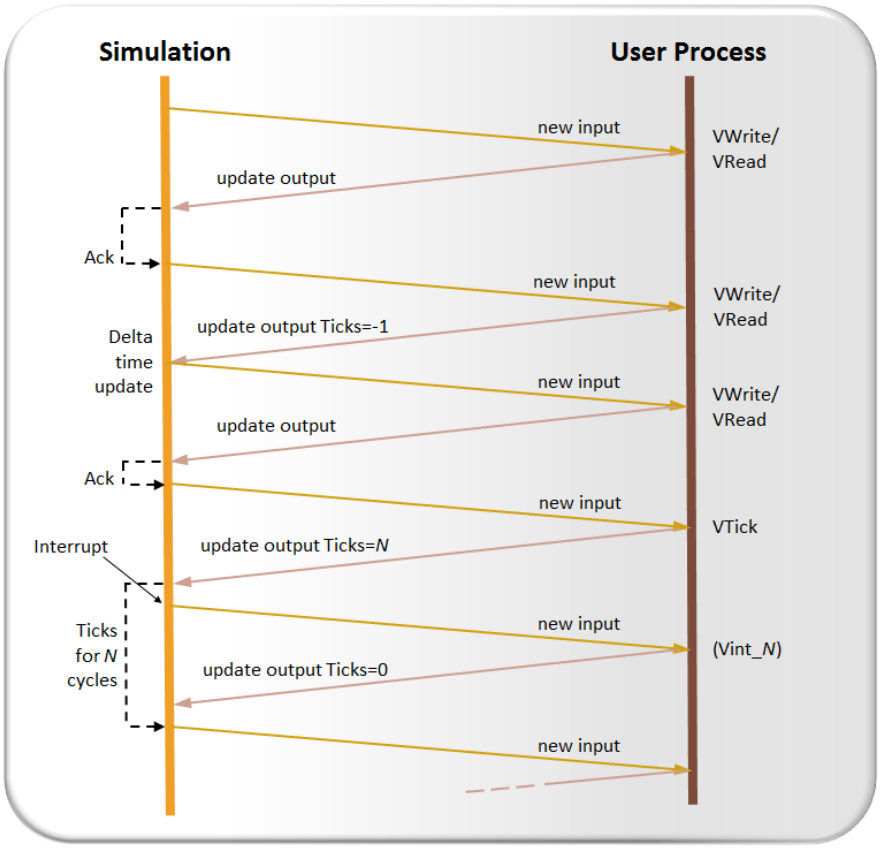
### VProc Download

The VProc package can be downloaded from github. This contains all the files needed to use VProc, along with *example* makefiles, scripts and test bench. Note that the only recent testing has been done with ModelSim on Linux.

# Appendices

## Message Passing

The figure below shows a typical exchange of messages between the HDL simulation and one of the user threads. There can, of course, be multiple user threads (one for each VProc) with similar interactions with the simulator.




As can be seen, there are normally two types of messages exchanged between user thread and simulation. The simulation, at time 0, always sends the first message. The simulation message to the user thread (a 'receive' message) includes the value of the `DataIn` port value, as well as an interrupt status flag. Once `$vsched` is called and a receive message sent, the simulation is effectively paused. Running in VUserMain*N*(), the simulation will not advance until `VWrite()`, `VRead()` or `VTick()` is invoked. Any amount of processing can occur in the user process before this happens, but it effectively happens in zero (delta) time as far as the simulation is concerned. When, say, a `VWrite()` is eventually called a 'send' message is sent back to the simulation with update data. In addition a Tick value is returned which is usually 0, meaning that the simulation will not call `$vsched` again until the write (or read) has been acknowledged externally. However it can be -1, in which case `$vsched` is called again after the update without waiting for an acknowledge, or waiting for the next clock edge. This allows a delta time update, enabling vectors wider than 32 bits be written (or read) before allowing the simulation to act upon the updated data. This is shown in the second exchange in the above figure. Simulation time can also be advanced without the need to perform a read or a write. Calling `VTick()` from the user process sends a message with a positive value for the Ticks. This simply delays the recalling of `$vsched` by the number of cycles specified. Effectively the user process can go to sleep for a set number of cycles; useful if waiting for an event that is known to take a minumum, but significant, time.

The send and receive messages are always paired like the exchanges in the above figure—one never sees two or more consecutive messages in the same direction. This gives full synchronisation between the simulation process and the user thread, and is controlled with send and receive semaphores—a pair for each VProc instantiated. If an interrupt occurs, a receive message is still delivered (see figure), but potentially earlier than expected. This will send the interrupt level in the message, and the appropriate registered function is called. When the function returns, a new send message is sent back, but the update values are ignored. Only the tick value is of significance. Normally a tick value of 0 is sent back. In this case the original state for the outstanding access is retained, and so the expected receive message for the original `VRead`/`VWrite` is still generated, when it would have been, if there had been no interrupt. A tick value greater than zero can be returned by the interrupt function, in which case the current outstanding tick count can be overridden, and the next receive message invoked earlier or later than originally set. This is useful for cancelling or extending a VTick() call. Say one has set off some action in the simulation which must complete before the user code can continue, but it is of abitrary length in time. By calling `VTick()` with a very large number, and by arranging the external verilog to invoke an interrupt when the action is completed, the interrupt routine called can clear the tick count to zero, and the user code will fall through the `VTick()` at just the right time, without, for example, the need to continually poll a register status, generating large amounts of message traffic and slowing the simulation down.

## Building more complex Virtual Processors

The VProc, as presented above, presents a simple environment, which has a memory style interface in the verilog domain, and a very simple read/write style API for the user C program to use, with simple support for interrupt handling. This would have limited scope as a useful verification tool if that was the limit of its capabilities. The main purpose of VProc is to hide away the verilog/C interface complexities, and allow a user to have an environment where a more useful and realistic virtual processor may be built. As such, two scenarios are described below, where VProc could be used to create more practical test elements.

### An AHB processing element (ARM substitute).

Suppose a test environment for an ARM based chip is created which is using an ARM model or netlist in the simulation. The test code for the processor needs to be cross-compiled to target the ARM, with the limitations on embedded ROM and RAM, and the compilation setup for the different areas of memory etc. relevant to the processor in the simulation. It may be that for greater than 80% of the tests it is not important that the code is running exactly as would be on the silicon implementation of the processor, but only bus transactions, on the AHB bus from the processor, are valid to configure the chip, instigate operations, monitor status, and log information to the simulation log. VProc can be the base to replace the ARM model in these situations, giving additional facilities of computation (all host libraries are available), checking, logging etc., that aren't possible in the actual processor model.

In order to do this VProc needs to be wrapped in some code to turn its memory based interface into an AHB interface, and the C API extended to wrap up the basic VProc API. In this example, let's assume that there is available a bus functional model (BFM) in behavioural verilog for AHB. This BFM, say, is controlled by reading and writing internal registers via a memory mapped interface. In that case, it is a simple matter to connect VProc's memory style interface to the register interface of the BFM, and wrap the whole in another module to hide away the details.

Controlling the BFM from C code is now simply a matter of sequences of read and write calls over the VProc API to configure the registers and instigate bus traffic. It is likely that for any given bus type transaction multiple register accesses are likley, so the normal thing to do would be to extend the VProc API with fundamental AHB bus API operations, such that all supported BFM transactions are a single C function call. This new AHB API is then the interface for verification tests to be written.

As an aside, one could imagine that if an instruction set simulator (ISS) existed for the processor being modelled (in this case an ARM processor), then this could be interfaced to the API described above to provide a full instruction capable model for verilog simulation. In this ARM scenario, this defeats the object of replacing the original ARM model, but suppose a new processing element or microcode engine is being designed, and an ISS is available long before the RTL design is ready and verified, then this could allow early testing of the rest of the RTL, and even as a platform for embedded firmware test and development.

### PCI Express Host Model

This scenario is for a model to drive a PCI Express (PCIe) interface. The PCIe interface on the chip under test is the element that is under verification, and needs a transactor to drive it. In this scenario, let's assume, unlike for the AHB model, that there is no existing BFM model in verilog. Whatever code drives the bus (actually link in the case of PCIe) must be written from scratch. We want to use VProc to allow a C program to deliver transactions over the bus, and receive and process returned data etc.

Because there is no BFM, and we are going to have to write a C API for the PCIe transactions anyway, let's decide that an absolute minimum of verilog is going to be written, and that we will model almost everything in C. Thus the verilog will consist only of mapping each of the PCIe lanes (the serial input and output ports) in to locations in the VProc memory map. In this case each lane will be a 10 bit value, with a behavioural serialiser/deserialiser in verilog. Thus lane 0 is mapped at address 0, lane 1 at address 1 etc. To update all lanes (anything from 1 to 32 can be used at once), delta updates to write to all the lanes (bar the last, to increment the clock) are done, returning the read value for the said lane on return. In this scenario, a single cycle always elapses for each lane set update, and the lanes are updated every cycle.

This is about as fundamental a C to verilog mapping that is possible with VProc. Each IO pin is simply mapped to memory and update/sampled for every clock. The C API then has to extend the basic VProc API in to all the PCIe transaction types. This is a much more complex task than for the AHB scenario above, but is made here simply because this functionality must be coded somewhere (there is no BFM, remember), and so C was the choice made. The C code would need to provide some basic conversions for the PCIe standard, such as 8/10 encoding/decoding, inline data pattern generation such as ordered sets, data link transactions like flow control and the data transactions such as memory, IO and configure reads and writes. A transmit queueing system is required, and the ability to handle split completions etc. This code should be written to hide these details and present an extended API to a user which allows single call access to the bus for every type of transaction. As you may appreciate by now, this is a non-trivial task, and VProc does not, of itself, solve these problems—it just enables the ability to do this. Under different circumstances it might be better to place more functionality in verilog behavioural code, and simplify the C code. It will depend on local circumstances.

## Limitations of VProc

There are few limitations to the model, which compiles and runs on a variety of simulators (4 have been tested), and platforms (Linux, Solaris). There is one flaw however, regarding the use of save and restore (or checkpointing, as it is sometimes known). Although support for checkpointing has been implemented in the model, inconsistent results are acheived—some simulation runs have worked, others behave differently, even by simply saving a checkpoint, and not just after a restart. Currently there is no fix for this in the published version of VProc, and the use of checkpointing is not yet supported.

## Pending Enhancements

- Transfer of data *blocks* over API
- Common entry point for all nodes (VUserMain()), to allow common code on all nodes.
- Fix checkpointing
- Pointer memory accesses (i.e. not read/write functional calls, but pointer references)

## Conclusions

A fundamental co-simulation element, VProc, has been described which virtualises away the C and simulation interface to give a basic processing element, controllable by host compiled code. This basic element provides enough functionality such that any arbitrary processing element with a given bus can be constructed, and two scenarios given (AHB and PCIe) with two very different approaches, showing the ultimate flexibility of the VProc element. The VProc code is available for download and is released under the GNU GPL version 3. This code, it is hoped, will allow engineers to construct highly flexible test elements, where none already exists, with the bus functionality they require combined with the power and flexibility of a full host programming environment. The two bus scenarios described were based on real examples using VProc, but it is hoped that VProc will be used in even more ways than this or originally envisaged.

---


*simon@anita-simulators.org.uk*

Return to homepage.