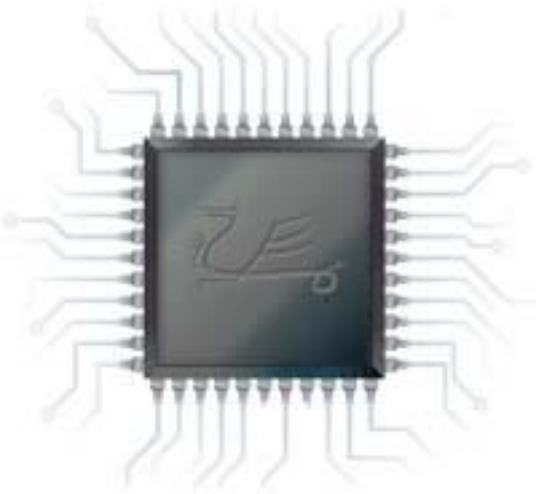
The *VProc* Virtual Processor

(version 1.5.1)



Simon Southwell

19th June 2010

(last updated 10th April 2024)

Copyright

Simon Southwell
Cambridge, UK
10th April 2024

Abstract

The VProc co-simulation element is documented here, which abstracts away the communication between a user C interface and a Verilog, SystemVerilog or VHDL task-based interface, wrapped in a simple behavioural HDL module, over the PLI, VPI, DPI, VHPIDIRECT or FLI of a simulator as a building block for a processing element whose code is a normal C or C++ program written and compiled for the host computer, or a Python program, controlling bus transactions in the simulation environment. This 'virtual processor' (VProc) element has a generic memory mapped type interface in the HDL domain, and a simple API in the C domain, allowing read and write operations and also support for burst transactions and nested vectored interrupts. The interfaces are kept as simple as possible, but documented here is the means to build on top of this fundamental co-simulation element any arbitrary complexity of data transfer interface (e.g. AHB, AXI, Avalon or PCI-Express) which can then be controlled by a user written program compiled for the host computer. A Python interface is also added on top of the C so that VProc may be driven from Python code. Support is provided for various simulators, with example code for ModelSim, Questa, Vivado Xsim, Icarus Verilog, NVC and GHDL. Code running on the virtual processor can be debugged using gdb and hence an IDE such as Eclipse, and an example of how to use these two tools with VProc is described in the document.

Contents

Introduction

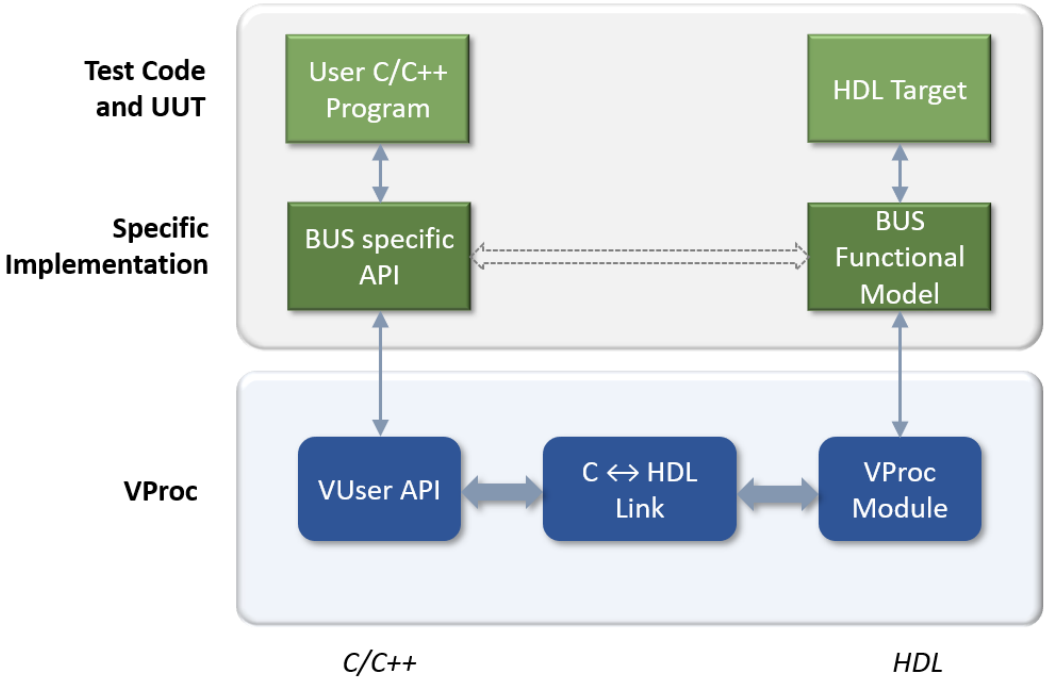
The concept of a virtual processor is far from new, as is the concept of hardware/software co-simulation. Tools such as V-CPU from Summit or Seamless from Mentor Graphics, give a professional way of running software targeted for a processor or system under development early in the design cycle. Although these tools are mainly aimed at running specifically targeted 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 provides 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 proprietary 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 into 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 utilise 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.

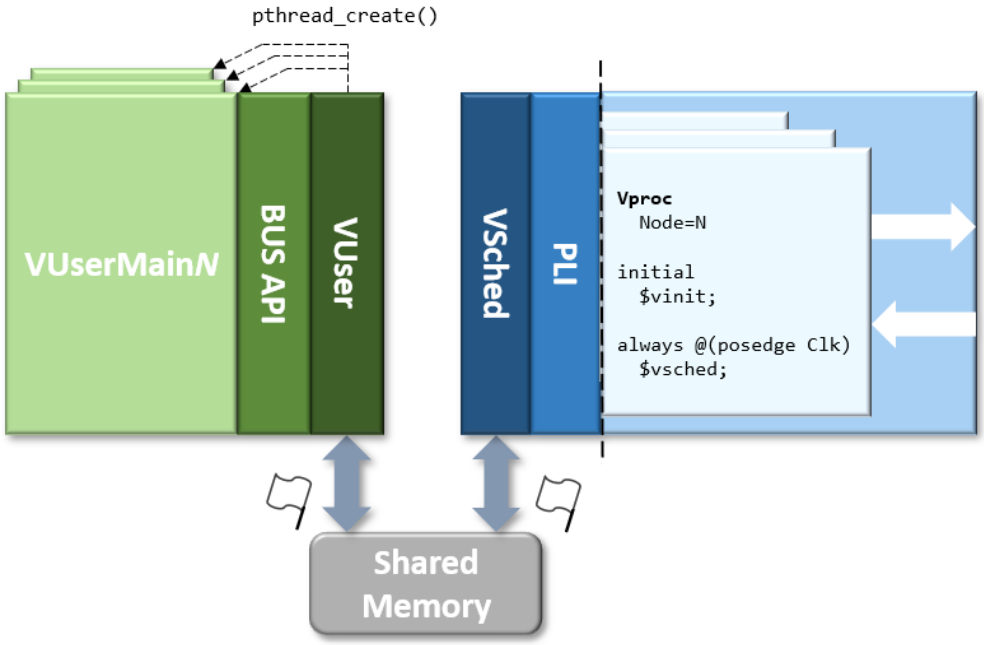


The above diagram shows what a typical virtual processor system set of layers might be. The bottom layers are provided with this system. On top of that a BFM in the HDL domain and an equivalent Bus API in the C domain are constructed to initiate bus transactions for the given targeted protocol. This 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 and VPI interfaces or the VHDL foreign model (FLI) and VHPIDIRECT interfaces, supporting it in both HDL 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). Currently ModelSim, Questa, Icarus Verilog and NVC VHDL simulators are supported.

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.



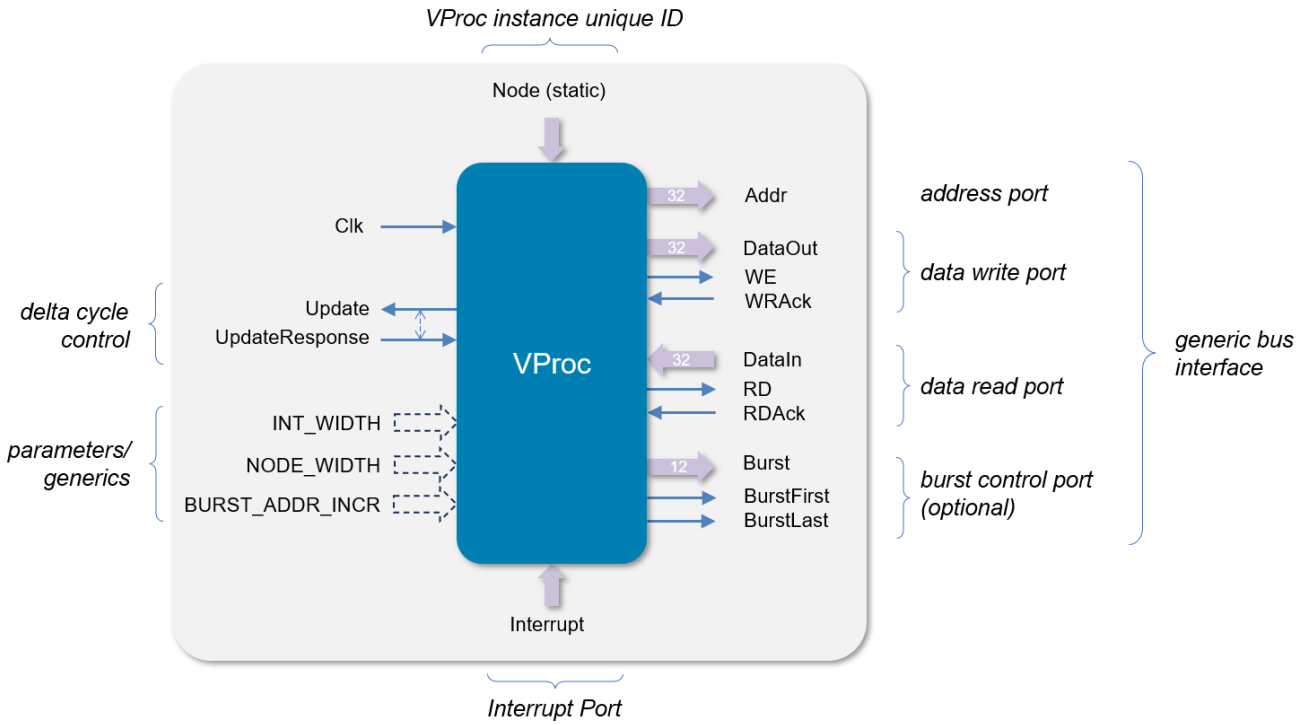
Bus Structure

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 separately 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 port which causes a vector wide call to a `$virq PLI` task whenever the interrupt value changes, independent of calls to `$vsched`. If a user has registered a callback function, then this is called with the state of the `Interrupt` input.

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 VProc component is shown in the diagram below:



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 transitions, 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* section 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.

The optional burst interface ports are only present if `VPROC_BURST_IF` is defined in order to maintain backwards compatibility with older versions of VProc. When present, a 12-bit `Burst` port indicates a burst length when a burst is active, and the `BurstFirst` and `BurstLast` signals indicate the first and last data transfers. The address during a burst will increment by the `BURST_ADDR_INCR` parameter value, which defaults to 1 for a word address and can be overridden for other alignments, such as 4 for byte addressing.

PLI Syntax

The actual syntax of the PLI (or VPI) 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, LegacyInterruptIn,
            DataIn, DataOut, AddrOut, RWOut,
            TicksOut)
$virq      (NodeIn, InterruptIn)
$vaccess   (NodeIn, Idx, DataIn, DataOut)
$vprocuser (NodeIn, ValueIn)
```

The VHDL has FLI equivalents of the verilog tasks as procedures `VInit`, `VSched`, `VIrq`, `VAccess` and `VProcUser`, defined in the `f_vproc_pkg.vhd` file, with equivalents for the NVC and GHDL simulators which use VHPIDIRECT (but have different syntax). 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 `$vinit` task is usually called within an initial block. The `Node` input needs to be a unique number for the `VProc` instantiated. If called with a constant or a module parameter, then this can be at time 0 within the initial block. If it is connected to a port or wire, even if ultimately connected to a constant, then a small delay must be introduced before calling `$vinit`, as the call to the PLI function and the wire assignment ordering is indeterminate.

The main active task call is to `$vsched`. It is this call that delivers commands and data, and has new input returned. It too has a node input and, in addition, a data input. `$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. A value of less than 0 can be returned, 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 `$vaccess` task is for accessing block data from a buffer via an index or placing/replacing data in that buffer. It requires an index argument and a data input and data output arguments. Presence of a buffer for burst reads or writes is

highlighted from a call to `$vsched` where the `RWOut` argument has a non-zero value in bits 13:2. This is the length of the 32-bit words of the access, with the buffer containing write data for write bursts, or the length of data to be updated for read bursts. Note that data is always written to the buffer, even on writes, so the buffer in the C domain will not retain its contents.

The `$vprocuser` syntax is a straightforward 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 function call methods in the different PLI interfaces are used in the 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 true for the VHDL FLI and VHPIDIRECT 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 `VuserMainN()` 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 addr, unsigned  data, int delta, unsigned node);
int  VRead         (unsigned addr, unsigned *data, int delta, unsigned node);
int  VBurstWrite   (unsigned addr, void* data, unsigned len,  unsigned node);
int  VBurstWrite   (unsigned addr, void* data, unsigned len,  unsigned node);
int  VTick         (unsigned cycles, int node);
```

```
void VRegIrq       (pVUserIrqCB_t func, unsigned node);
void VRegUser      (pVUserCB_t func, unsigned 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 on 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 `$vsched` 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* section 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 for writes, allowing wide data reads without advancing simulation time, or waiting on an acknowledge.

Two burst functions are provided for reads and writes. These require an address, a pointer to a 32-bit word buffer, either containing data for writes, or space to receive data on reads. The length of the burst, in words, is passed in, and then the node number (as for all the functions). The burst transfer will start accesses from the address and, by default, increment this by 1 for each subsequent access. Thus the address argument supplied is a *word* address. A parameter/generic is supplied (`BURST_ADDR_INCR`) which can be overridden to increment by other constants, such as 4 for byte addressing etc.

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). For example, at each call to a read or write some arbitrary tick count might be called first to model processing time of the virtual processor and to vary delays between transactions from no delay to several cycles to give better usage coverage.

Interrupts

Interrupts are handled using callback functions, where the user registers their own function using an API method, and this is called whenever the `Interrupt` input on the VProc component changes, passing in the interrupt vector value.

The method uses a single user registered callback function, registered using the `VRegIrq()` API function, which gets called whenever the interrupt input *changes* value. The callback function must be of type `pVUserIrqCB_t` which takes a single integer argument for the current interrupt value, returning an integer (usually 0). The new interrupt value is passed into the user callback for it to process and will also be called if the interrupt port changes to a passive (all zeros) state. Thus the callback must maintain state and update it when the interrupt changes. The advantage is that a single wide vector can be handled, and multiple interrupt changes of state will be logged in a single call. By default the `Interrupt` port of the VProc component is 3 bits wide, but the `INT_WIDTH` parameter/generic can be set to widen this to be up to a maximum of 32 bits.

In this release, the user interrupt callback routine cannot make calls to the VProc API functions (as this will break the scheduling). They are intended only to flag events, which, if IO is required, the main program can service. It is 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 before calling the VProc API function, and a call to a handler made if required, based on status updated by a call to the interrupt callback. This handler would now be part of the main thread and can safely make API accesses. This implies that simulation API calls are atomic and won't be interrupted.

As an example, suppose we construct a C++ class to wrap up the API calls and check for interrupt state changes. Let’s call it `vprocIrqClass`. In this class, as well as having wrapper methods for the VProc API functions we will have a `processIrq()` method

which will check `irq` state and make calls to user registered functions as appropriate. A sketch classis shown below:

```
#include "VUser.h"

class vprocIrqClass
{
public:

    // constructor
    vprocIrqClass(const int nodeIn = 0) :
        node(nodeIn),
        interrupt(0),
        isr_enable(0),
        irq(0) {};

    void tick (const int ticks) {
        processIrq();
        VTick(ticks, node);
    }

    void write (const uint32_t addr, const uint8_t data, const int delta = 0) {
         processIrq();
         return VWrite(addr, data, delta, node);
    }

    void read (const uint32_t addr, uint8_t *data, const int delta = 0) {
        processIrq();
        VRead(addr, data, delta, node);
    }
    // ...and so on for the rest of the API functions


    // Interrupt API methods
    void enableInterrupts  (void) {interrupt_enable = true;}
    void disableInterrupts (void) {interrupt_enable = false;}
    void enableIsr         (const unsigned intNum) {isr_enable |=  (1 << intNum);}
    void disableIsr        (const unsigned intNum) {isr_enable &= ~(1 << intNum);}
    void udateIrqState     (const uint32_t newirq) {irq = newirq;}
    void registerIsr       (const pIntFunc_t isrFunc, const uint32_t level)
                           {isr[level] = isrFunc;}

protected:
    void processIrq();

private:

    int node;
    bool interrupt_enable;         // master interrupt enables
    uint32_t isr_enable;           // vector irq enables
    uint32_t irq;                  // irq state
    pIntFunc_t isr[MAXINTERRUPTS]; // pointers to ISR functions
}
```

In this class, each of the VProc API functions are wrapped in a matching method, but an internal method, `processIrq()`, is called before each call to the API function. Some internal state gives control of enabling or disabling interrupts at a global level

or for each of the 32 individual IRQs of the vector input via the shown methods. User code can register one or more interrupt service routine (ISR) functions using the `registerIsr()` method, mapping to the desired IRQ level. The `updateIrqState()` method is used to log the current state of the interrupt requests. It is this method that would be called by the VProc registered user IRQ callback to update the IRQ state each time it changed.

The `processIrq()` method is shown as protected as one might construct this class as a base class and the user can derive a new class with this method overloaded in order to implement their own interrupt handling characteristics. At a minimum, when called, the method inspects the current `irq` state and calls any user registered function for an active request, so long as interrupts are enabled, and the particular level is also enabled. Hierarchical interrupts can be modelled by selecting the 'highest' priority ISR when multiple active and enabled interrupts are present, where highest could be lowest numbered IRQ or highest, or any other scheme required. Nested interrupts can also be handled, so long as the ISR functions use this class to access the VProc API. When an ISR function does a read or write (or any other API call) `processIrq()` will be called again. If a new, higher priority IRQ has become active whilst running the current ISR, it can call the higher priority ISR instead, which will continue to run (assuming no irq at an even higher level) until it clears its interrupt and returns. The outstanding, lower priority, ISR will then continue from the point of the API call it made when it was interrupted, until all ISRs complete and the main program continues.

A much more elaborate example interrupt wrapper class is provided in the file `code/VprocIrqClass.h` which supports nested vectored interrupts and configuring for edge triggered events. Some example test code is in the `test/usercodeIrq` directory. To run, set the `USRCDIR` make variable to point to the `usercodeIrq` directory. E.g. `make USRCDIR=usercodeIrq run`.

User Callback

As well as registering interrupt callback functions, a general-purpose callback function may be registered using `VRegUser()`, and is similar to `VRegIrq()`, 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 on where the task is called

from. 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.

A C++ API Class

In order to make interfacing VProc code to C++ easier an API class wrapper is provided in a file `VProcClass.h` which defines a VProc class. This class provides access to the VProc API functionality as methods within the class. The class is shown below, abbreviated to the available public methods:

```
class VProc
{
public:
          // Constructor
          VProc (const unsigned nodeIn) : node(nodeIn);

     // API methods
     int  write (const unsigned addr, const unsigned data, const int delta = 0);
     int  read  (const unsigned addr, unsigned  *data, const int delta = 0);
     int  burstWrite (const unsigned addr, void *data, const unsigned length);
     int  burstRead  (const unsigned addr, void *data, const unsigned length);
     int  tick         (const unsigned ticks) ;
     void regIrq       (const pVUserIrqCB_t func);
     void regUser      (const pVUserCB_t func);
};
```

Including the `VProcClass.h` header gives access to all the VProc API features. The constructor has a single node input argument to associate the created object with the particular VProc instance, and thus the API methods do not require to have a node argument. The API methods have a one-to-one correspondence with their C counterparts (as defined in `VUser.h`) with the same argument lists minus the node argument. The delta argument for the write and read methods defaults to being a normal access (`delta` equal 0) and so can be omitted when delta updates are not required.

Using Python with VProc

Prerequisites

In order to use the python features of the VProc virtual processor python3 must be installed—VProc was tested with python version 3.10. Also, the matching python development libraries and headers must also be installed (e.g., `sudo apt-get install python3-dev`)

Usage

VProc is supplied with a Python API interface and support code to allow scripts to be written in that language as an alternative to C or C++. It provides the same sort of interface as the C API, with additional C code being compiled into a separate shared object, `PyVProc.so`, that is suitable be imported into a script as a module. Like the user C code, each node must have a `VUserMain`*N* python function defined as the entry point for that node's code and must be in a script file called `VUserMain`*N*`.py`. The scripts can't be run from the command line as they will be called from the VProc

code when the simulation is run. For example, python code for node 0 might look like the following:

```
from ctypes import *
import pyvproc   # Assumes PYTHONPATH set up

# Define a global vproc API object handle
vpapi = None

def VUserMain0() :
  # Get access to global variable
  global vpapi

  # This is node 0
  node  = 0

  # Create an API object for node 0
  vpapi = pyvproc.PyVProcClass(node)

  # --- REST OF USER CODE HERE ---
```

The python code creates an API object using `PyVProcClass` from the `pyvproc` module (in `python/modules`), passing in the node number to attach to that VProc instantiation. The rest of the Python script then has access to the API methods of that class:

- `vpapi.write(addr, data, delta=0)`
- `vpapi.read(addr, delta=0)`
- `vpapi.uread(addr, delta=0)`
- `vpapi.burstWrite(addr, datalist, length)`
- `vpapi.burstRead(addr, length)`
- `vpapi.tick(ticks)`
- `vpapi.regIrq(irqCb)`
- `vpapi.VPrint(printstr)`

These all have direct counterparts in the C/C++ APIs. The only real difference is that the `read` Python method returns the read data, rather than updates an argument passed in as a pointer. Similarly for `burstRead`, the data is returned as a list of values.

The `regIrq` method takes a Python function definition that expects a single argument (the IRQ vector) and returns a status (0 being success). For example, a function where IRQ[0] is connected to power-on-reset:

```
def irqCb (irq) :
    global seenreset, vpapi     # Get access to global variables

    # If irq[0] is 1, flag that seen a reset deassertion
    if irq & 0x1 :
      vpapi.VPrint("  Seen reset deasserted!\n")
      seenreset = 1

    return 0
```

The above function can then be registered for calling whenever the IRQ vector changes using `vpapi.regIrq(irqCb)` in the main code.

The `VPrint` method is an alternative to the Python print, as some simulators on some platforms don't seem to send the Python output to the console (e.g., Questa on Linux). This method takes a single string input and passes this over to the C domain for printing. It is not an exact replacement for Python's print, which can take a set of comma separated arguments, but must have a single string argument. A string can be concatenated using + and formatting functions (e.g., `str` or `hex` etc.) so this is not much of a limitation. If your simulator does print the python output, then use `print` for maximum portability.

A note on the returned type from `vapi.read`. This will be a C type `int32_t`. Python will, by default, interpret the number as a python `int`, and values returned with the top bit set will be interpreted as negative (as expected) but also sign extended. If a value (from the C domain) is returned into `rdata` as, say, `0xffffffff`, then the read value will be -1 (as expected) and print as such. If `hex(rdata)` is printed, though, it will be `-0x1` which may not be as expected. If you wish to interpret as unsigned numbers then you can use the `uread` API call which returns a `c_uint32` value. E.g.

```
rdata = vpapi.uread(addr))
```

Alternatively, if you wish the value to be interpreted as a signed number, but wish to print the original hex value, then the `c_uint32` can be used to cast the value in the print:

```
print(c_uint32(rdata).value)
```

Some demonstration test code is located in `python/test` that demonstrates all these features, along with some make files for compiling for ModelSim, Questa and Icarus Verilog. A VHDL make file is also provided which defaults to ModelSim but can be used with Questa by adding the argument `ARCHFLAG=-m64` to the call to `make`. When compiled and additional shared object (over and above `VProc.so`) is compiled called `PyVProc.so` which is the loadable foreign module that's the interface to the

VProc C/C++ domain. User code is in a `usercode/` directory which must be on the `PYTHONPATH`. I.e., `export PYTHONPATH=$PYTHONPATH:./usercode`.

Limitations of VProc with Python

Unlike for the C++ domain, multiple instantiations of VProc or not supported at this time and only a single VProc can be used in a simulation environment. (The code gives a run time error if a second instantiation is initialised.) This is due to the limitations of the Python/C interface which does not appear to support multiple outstanding calls across it, so new transactions from other nodes can't be started if one node already has an active transaction. A solution for this issue is currently under development and will be added in a future release.

In all other respects, for a single node, the python API can be used just as for C and C++, supporting all transaction types and support for interrupts.

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.

ModelSim

ModelSim compiles the Verilog or VHDL, and runs the simulation separately, 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 functions 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-wholearchive 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.

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-wholearchive` 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`.

Compilation Options

There are various example `make` files for `VProc` to support different simulators in the `test/` directory, which contains a demonstration test bench and code. The default file (`makefile`) is for ModelSim, but other simulators are supported. The list below shows the supported simulators for the Verilog version of VProc, and their respective make files.

`makefile`	: ModelSim Verilog PLI/VPI
`makefile.vhd`	: ModelSim/Questa VHDL FLI

`makefile.questa` : Questa Verilog
`makefile.ica` : Icarus Verilog
`makefile.nvc` : NVC VHDL simulator
`makefile.nc` : NC-Sim (unsupported)
`makefile.vcs` : VCS (unsupported)
`makefile.cver` : GPL CVer (unsupported)

When using the VHDL version of VProc then `makefile.vhd` is used. This supports ModelSim by default but can be used with Questa by setting the `ARCHFLAG` make file variable to `-m64`, either by updating the make file or on the command line. E.g.

```
make ARCHFLAG=-m64 -f makefile.vhd
```

By default, the verilog compilations use the PLI task/function API (PLI 1.0). VProc supports use of the VPI (PLI 2.0) and can be compiled to use this if `VPROC_PLI_VPI` is defined when compiling the code. The make files can be updated to define this internally, but a variable (`USRFLAGS`) may be set when calling make to set the `VPROC_PLI_VPI` definition. E.g.

```
make USRFLAGS=-DVPROC_PLI_VPI -f makefile.ica
```

Compilation Definitions

The VProc code supports various simulators and programming interfaces, and each have their particular syntaxes and specific requirements. In order to have a single set of source code, various compilation definitions are used to select for the particular needs of the interfaces or simulators. The make files in the test directories serve as examples for the various supported platforms, but the list below summaries what the available definitions are, and their use, for those wishing to construct their own compilation scrips or make files.

- C/C++ programming interface definitions
  - **`VPROC_VHDL`** : define when compiling the VHDL VProc
  - **`VPROC_VHDL_VHPI`** : select VHPI over VHPIDIRECT interface for VHDL VProc (experimental).
  - **`VPROC_PLI_VPI`** : select Verilog's VPI instead of PLI 1.0 interface, when compiling Verilog VProc
  - **`VPROC_SV`** : define when compiling SystemVerilog VProc with DPI interface

- C/C++ simulator definitions
  - **`NVC`** : define when using NVC VHDL simulator

    - **GHDL** : define when using GHDL VHDL simulator
    - **ICARUS** : define when using Icarus Verilog simulator

- Verilog simulator definitions
    - **VPROC_BURST_IF** : define with Verilog VProc to enable burst interface

Debugging User Programs

It is possible to debug user programs using the normal debug tools such as gdb. This is done by attaching to a running simulation's kernel process. In the discussion that follows we shall look at ModelSim as an example for gdb, but this should be adaptable for other simulators.

Using gdb

Using the supplied demonstration test code as an example, typing `make sim` will compile all the code and start the simulator, but without running the logic simulation. At this point the simulation kernel process, `vsimk` for ModelSim, will be running and its process ID is required. Under Linux this can be found by running:

```
ps -e | grep vsimk
```

This will give an output something like that shown below:

```
3200 pts/0    00:00:00 vsimk
```

Thus the process ID is 3200. On Windows the task manager can be used to find the same information. Right click on the task bar and choose the Task Manager from the selections. If the opened window has "More details" displayed in the bottom left corner, then expand this to get a set of tabs. Choosing the "Details" tab gives a list of all the running processes. Search for `vsimk` and note its PID number.

Once the PID for the simulation kernel process is known, from the directory where `VProc.so` resides, we can attach a gdb session to it using:

```
gdb -p <PID>
```

You may have to be root user on Linux for this to work. There will be a load of warnings that there are no debugging symbols for the system libraries being used, but this is not an issue so long as there are none regarding `VProc.so`. Once the gdb session starts, you can list code (e.g. `list VUserMain0`) and set breakpoints (e.g. `b 66`). At this point the simulation is paused by gdb, so a `continue` command is

needed to start the simulation process once more. On the simulator command line, then the simulation can be run (e.g. `run -all`) and will continue until a break point in the C/C++ code is hit, or the simulation ends. If at a breakpoint, gdb will have a command prompt once again and state can be inspected and any other gdb debug command used as normal. And so debugging of user code can proceed. When at a gdb prompt, the simulation process will be paused and simulator command lines and window buttons etc. will not respond to inputs.

Of course, the simulation may stop if, say, run for a set time (e.g. `run 100 us`) or any other criteria, and then waveforms and state can be inspected. At this point, gdb will still be 'running' waiting for a breakpoint, and so cannot take new command inputs.

Some simulators that aren't immediately run also delay loading the `VProc.so` shared object. For example, Icarus Verilog when using the `vvp -s` command. Therefore when gdb is attached to the process it cannot find the symbols for the user code. A way around this is to stop the simulation with a call to $stop very soon after the simulation is run, for example in an initial process:

```
initial
   #0 $stop;
```

At this point the shared object is loaded and `gdb` can attach and setup breakpoints, list code etc. The demonstration test bench will do this if the top-level `test` module's parameter, `DEBUG_STOP`, is set to a non-zero value and the Icarus make file can be run as `make -f makefile.ica debug`, and it will stop at time zero.

Setting Up Eclipse

In this section is an example for setting up the Eclipse IDE, assuming `gdb` is already available. For this Eclipse example we'll use the Questa simulator and the VProc test code supplied with the repository as the software to be debugged, but the procedure should be able to be adapted for the other simulators and code.

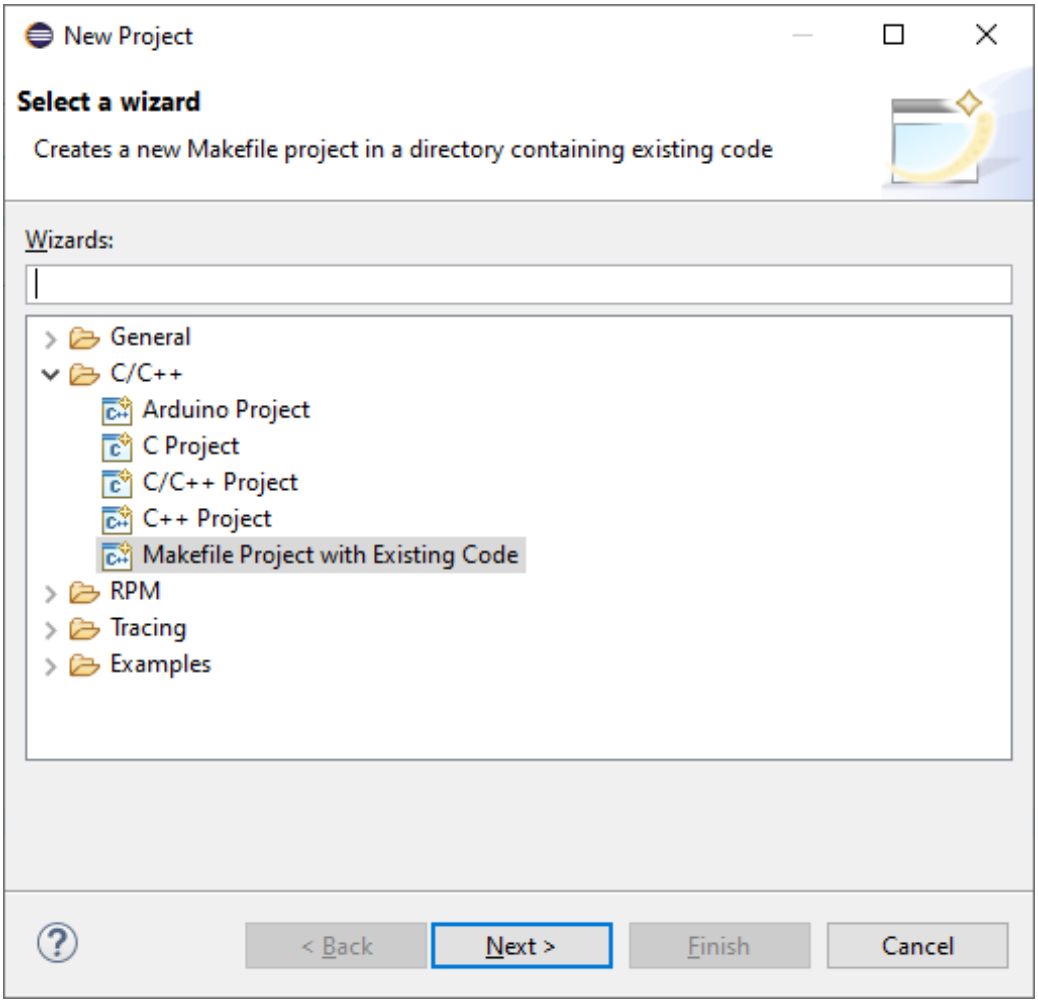
Although the use of Eclipse is not necessary when developing vectors on the virtual processor test environment, the advantages are such that it is highly recommended, as the full power of an integrated IDE for development, building and debugging can't be underestimated. There are three steps to setting up Eclipse, and it is assumed that an Eclipse tool has already been installed on the system, and a workspace set up. The steps are:

- Create a project to use the make setup we already have
- Set up the build, using the make files of the project

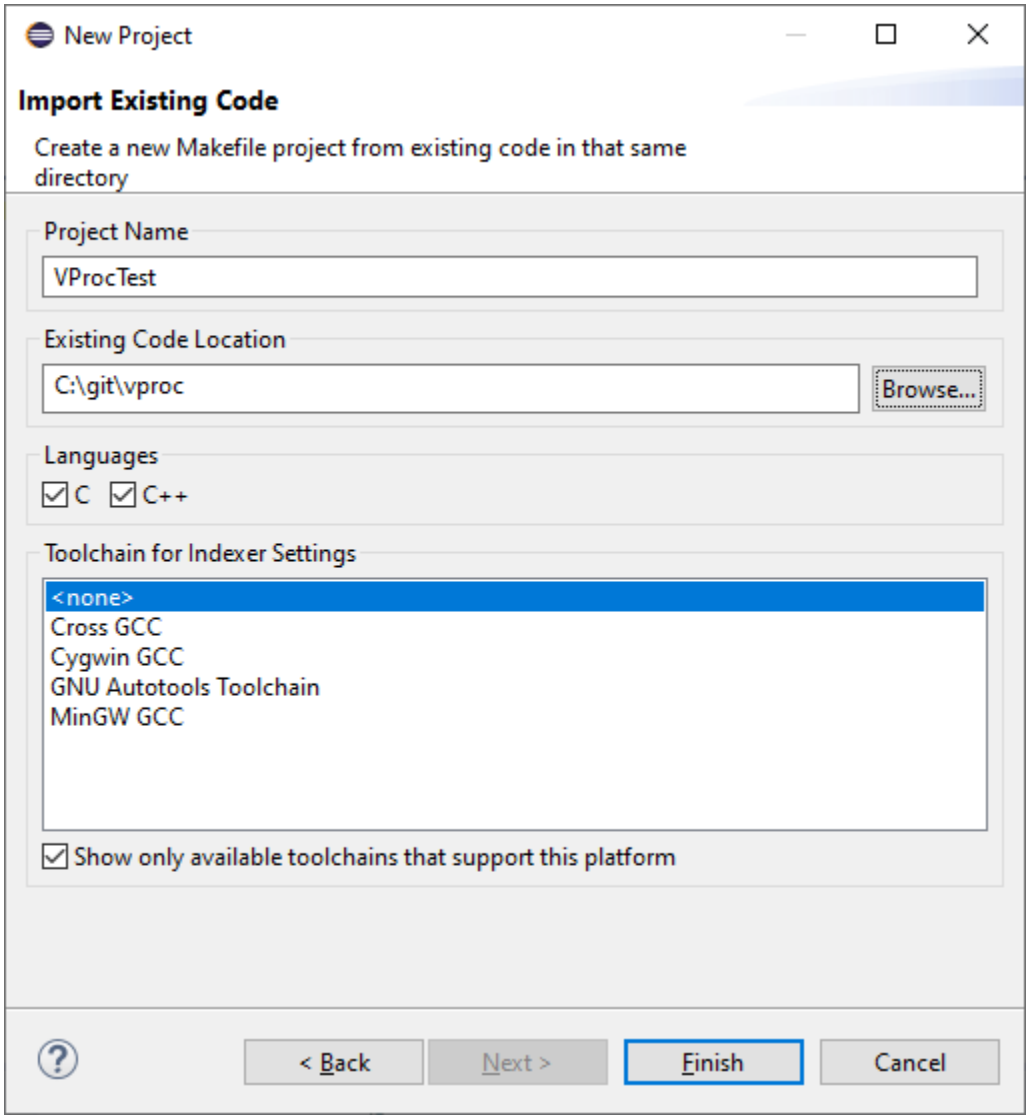
- Set up a debug configuration

Creating a Project

With Eclipse open, a new project is created using: ***File->New->Project***. This will open a window like the following:

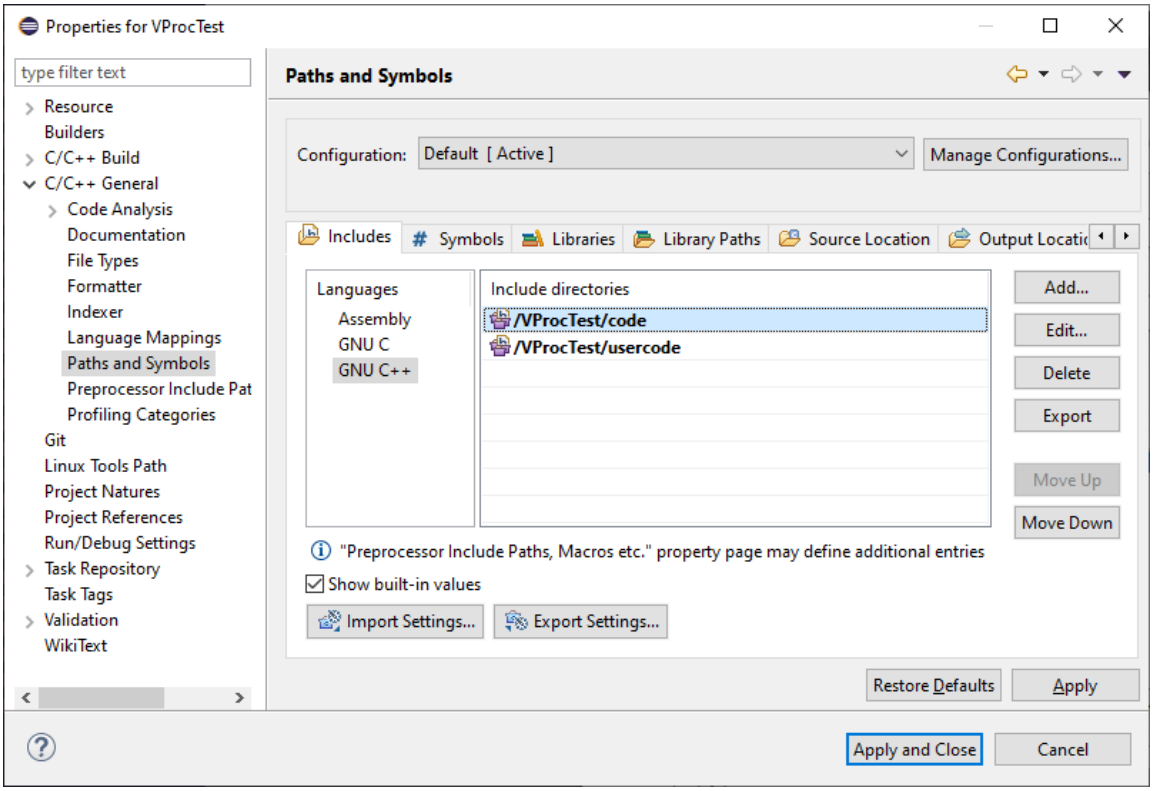


Select ***Makefile Project with Existing Code*** and press the ***Next >*** button. This will open a new window that looks something like the following:



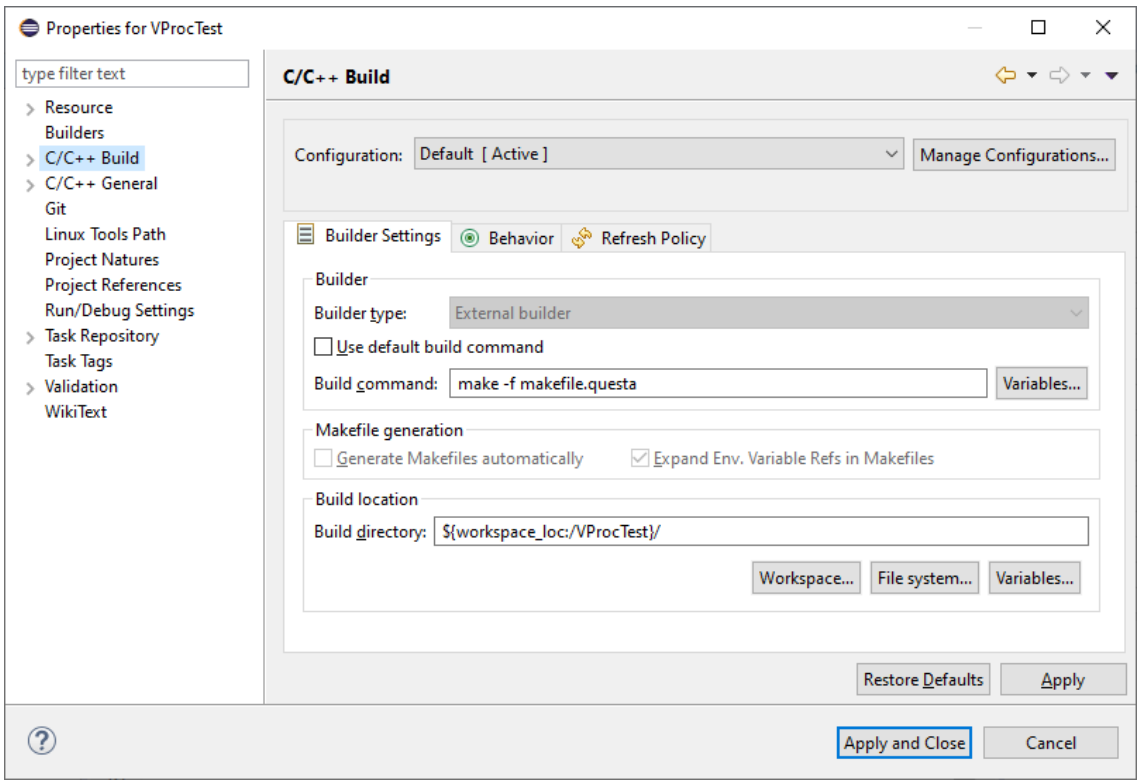
The Project Name field should be filled in with a suitable name, and the Browse button used to navigate to the test folder of the project, where the `makefile` is located. The ***Languages*** tick boxes should both be ticked**,** and the ***Toolchain for Indexer Settings*** box should have *<none>* selected. This is important later for setting up a build configuration using our make file process, and not an internal tool.

After the ***Finish*** button is pressed the new project will appear in the Project Explorer pane on the left of the IDE window. Paths need to be set up for the IDE to know about include files etc. that aren't part of the main source code. This is done with ***Project->Properties*** and selecting ***C/C++ General->Paths and Symbols***. Paths can be added with ***Add...***, which brings up a small window, and the file system button used to find a folder. Note that the ***Add to all configurations*** and the ***Add to all languages*** tick boxes should be checked. For example**,** the folders below pick up the VProc Code folder and the example test user code. Since both are under the project folder they were selected as workspace folder, where external folders would be selected as file system folders when adding the include directories:



Setting up a Build Configuration

A build configuration is what tells eclipse how to compile the code. In this case we want to use the test code's own make files. To create a new build configuration use: ***Project->Properties***. A Properties box then appears that looks something like the following once the C/C++ Build entry on the left is selected:

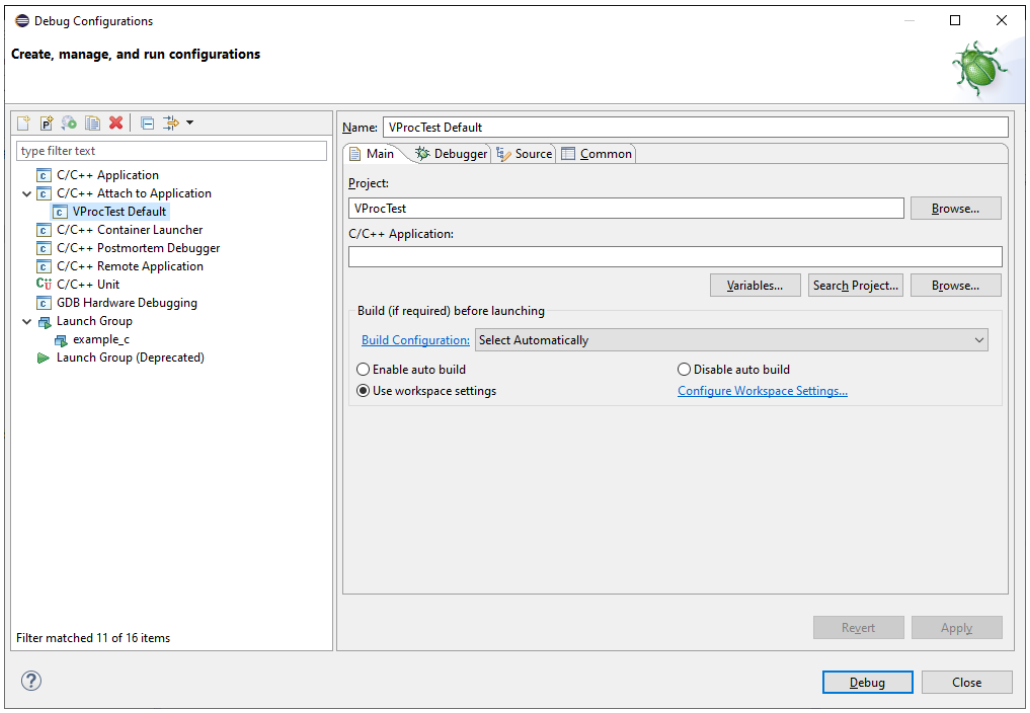


By default the ***Use default build command*** box will be ticked. This should be unticked and the ***Build command*** changed from `make` to `make -f makefile.questa`, as this example is for the Questa simulator, as shown above. It might be useful to select the ***Behavior*** tab and check that the ***Build (incremental build)*** tick box is checked, and the adjacent box is *all*. Similarly for the ***Clean*** box, check that this is ticked and the text box is set to *clean*. You can test if the configuration works by selecting ***Project->Build Project***.

Setting up a Debug Configuration

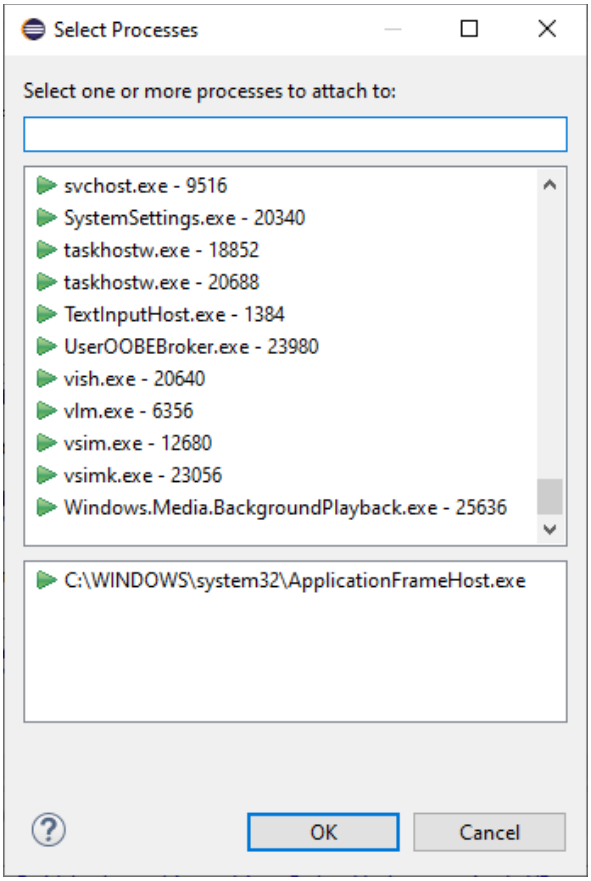
Because the VProc code is compiled as a shared object loaded into the Questa simulator tool, we can't simply debug as for an executable. The approach to be taken is to run the simulation separately and then attach the debugger to a particular Questa process. Since, by default when running the simulations (with `make run`) the simulation will start running the code more or less straight away, it is worth having a debug build where the test code will stop before running. The debugger can then be attached, any breakpoints setup and the simulation resumed. The make file for the VProc example test code has a target 'sim' which will do everything that `make run` will do, except it will not run the simulation but wait at the command line for a *run -all* command.

To add a new configuration use ***Run->Debug Configurations***. Select the ***C/C++ Attach to Application*** and press the ***New launch configuration*** icon in the top left. The right hand pane will change with a new set of tabs and fields like that shown below:



For the ***Main*** tab, all that needs changing is to select ***Disable auto build***. None of the other settings for any of the tabs need changing from their default settings—though of course they can be if necessary for project specific reasons. In particular, because we will be attaching to a running process, the ***C/C++ Application*** field does not need to be set, as the debugger will not be responsible for running the program. When the changes are made, the ***Apply*** button is clicked and the window then closed. To run a debug session the following needs to be done:

- Set any debug points in the source code that are required
    - It might be a good idea to set one in `VUserMain0()` before the main tests are called to ensure the debugger is attached properly.
    - If setting a debug point gives an error about a launch configuration required, ensure that the break point type is set to C/C++ breakpoints using ***Run->Breakpoint Types***.
- From a separate command window execute the simulation for debugging. I.e. `make sim`
    - This will get the simulator running but the simulation will be paused at time 0.
- Run the debug session using ***Run->Debug Configurations***, selecting the configuration created before and then click ***Debug***. A window like that shown below should appear:



Find the `vsimk` or `vsimk.exe` process, select it and press ***OK***. Don't pick `vsim/vsim.exe`, even though this is the main called program, as our shared object is not loaded by this process but by the main simulation kernel process, `vsimk/ vsimk.exe`.

The debugger should attach to the process and pause it. The debugger should then be Resumed (F8) and then `run -all` (or `run <time>`) at the simulation command prompt that is running the simulation, to start the logic simulation execution. At this point the simulation will run until the next break point or until the end of the simulation (or reached the time specification of the run command if specified), if no breakpoint was hit. Once at a break point, debugging of the C/C++ code can proceed as for any normal code. Note that the logic simulation, when the debugger is at a breakpoint, is 'frozen' and will not respond to commands or other input.

Delivered Files

For reference, the source following files are listed below with a brief description.

`f_VProc.v`	: Virtual processor verilog
`extradefs.v`	: Common global verilog definitions
`vprocdefs.vh`	: VProc verilog definitions (includes `extradefs.v`)
`f_VProc.sv`	: Virtual Processor SystemVerilog
`vprocdpi.vh`	: DPI-C import definitions for SystemVerilog
`f_vproc.vhd`	: Virtual processor VHDL
`f_vproc_pkg.vhd`	: Virtual processor VHDL package defining FLI tasks
bfm/axi4bfm.v	: Experimental AXI4 BFM VProc wrapper
`code/VSched.c`	: Simulation (server) side C code
`code/VSched_pli.h`	: Common PLI definitions and prototypes
`code/veriuser.c`	: PLI table (NC-Verilog)
`code/VProc.h`	: VProc layer definitions
`code/VUser.c`	: User (client) side C code
`code/VUser.h`	: User (client) side header file
`code/VProcClass.h`	: C++ API class definition
`code/VUserMainT.c`	: Template for user code
`python/src/PythonVProc.c`	: Python interface code
`python/src/PythonVProc.h`	: Header for Python interface code
`python/src/VUserMainPy.c`	: Substitute C VUserMain entry points for Python
`python/modules/pyvproc.py`	: Python API class definition

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 make files 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 make files, 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 make files, scripts, and test bench. Note that the only recent testing has been done with ModelSim on Linux.

Building more Complex Virtual Processors

The VProc, as described 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, if very useful, scope as a verification tool if that was the extent 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 AXI 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 AXI 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 AXI 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 AXI. (Actually, a prototype AXI BFM wrapper can be found in the `bfm/` directory.) This BFM maps the VProc generic memory mapped bus into AXI transactions for reading and writing and also has a vectored interrupt input.

Controlling the BFM from C code is now simply a matter of sequences of read and write calls over the `VProc` API to instigate AXI bus traffic, completely transparently to the user code. An experimental Verilog AXI4 BFM is provided in the `bfm/` directory as `axi4bfm.v`.

PCIe 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) into 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. This kind of PCIe model with VProc has already been done and can be found on github. On a similar vein, their also exists a USB model and a TCP/IP model co-simulating in a logic simulator via VProc.

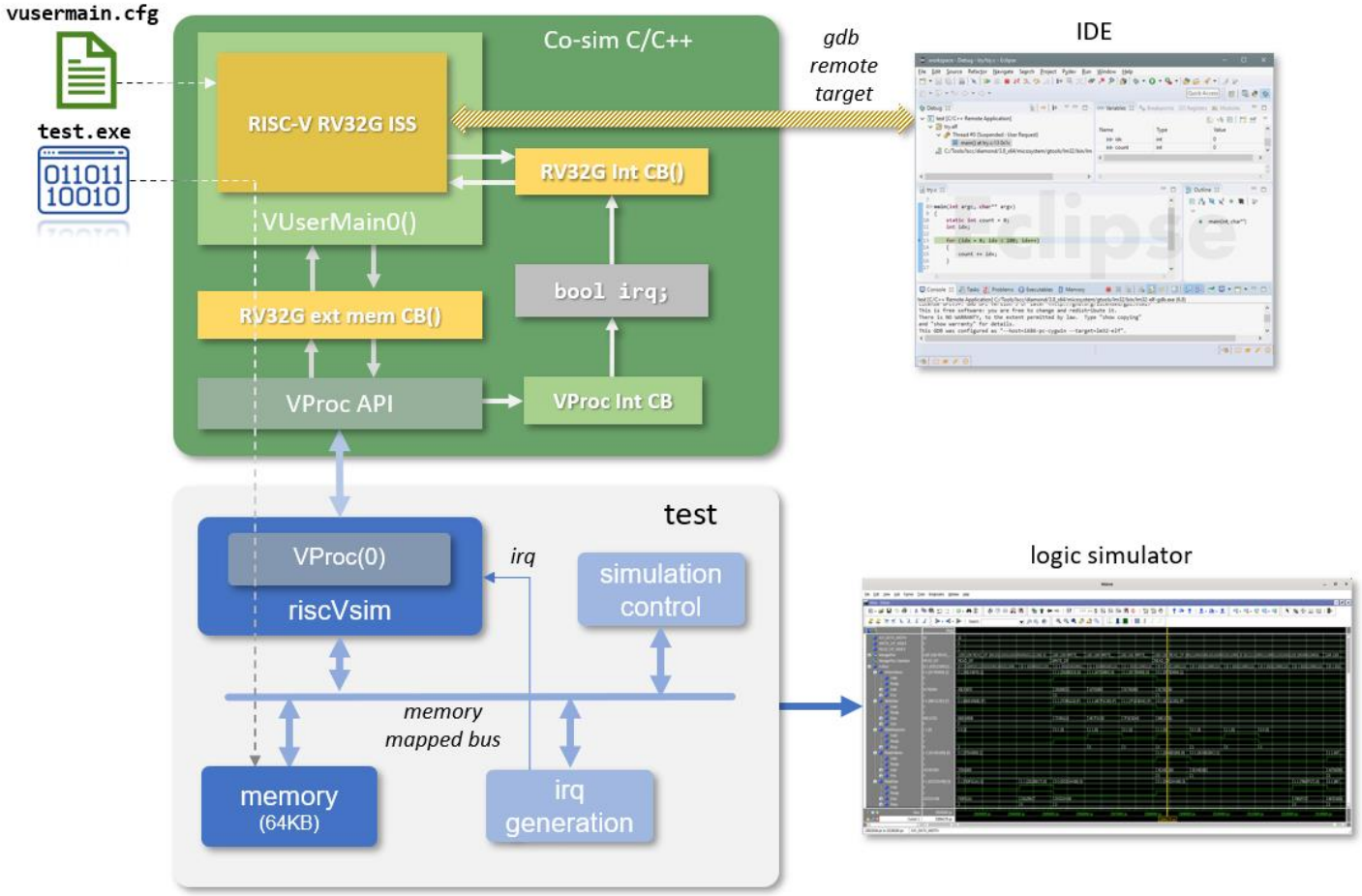
RISC-V Software Model Co-simulation

The VProc virtual processor can run a natively compiled program using the API to do read and write transactions over the VProc memory mapped bus, and to receive interrupt signalling. This program can be anything, though, and is not limited to running test code in C++. For example, the code that's run could be a model of an actual processor core, such as a RISC-V core.

In the `examples/riscv` directory just such a scenario is demonstrated using the rv32 RISC-V ISS model that is part of the riscV project. The manual for the ISS details its interface and usage in detail, but a short summary here will set the scenario. It is a compliant RISC-V model for RV32G standard, with support for RV32E and RV32C capabilities (configurable) and has the minimum Zicsr extensions. It has a single hart and support machine privilege mode only. It has support for interrupts via registering a callback function to read interrupt state, and also an external memory callback registering capability to allow external models to intercept all loads and stores and map in other models into the core's address space. RISC-V executables can be loaded by the model to memory (including externally mapped address space). It supports debugging via gdb remote target access (which is separate from debugging the native VProc program which, in this case, is the ISS model itself) and can output run-time disassembled code.

The provided example runs this model from the VProc's `VUserMain0` entry point (for node 0), registering a callback function for the model's external memory accesses and registering another callback function for its interrupts status reads. A separate interrupt callback function is registered with the VProc API to be called when the interrupts status on VProc's `Interrupt` input port changes. This updates internal IRQ

state which can then be returned to the model when the ISS calls its registered callback function. The `VUserMain0` program reads a configuration file (`vusermain.cfg`), if present, which sets various 'command line' options for the model (add the `-h` option to the configuration file to display the other available options). The test bench environment, in simulation, looks something like the following diagram:



VProc is wrapped up in a RISC-V component with an instruction read interface, a data read/write interface, and a 32-bit vectored interrupt input. A simple dual-port 64Kbyte RAM component serves for both program and data. When the model loads the RISC-V test program this is sent to the registered external memory callback, which does a series of writes to the HDL memory model via the VProc API. The ISS is then run, executing code out of the HDL memory model, via the external memory callback function and, where necessary, doing loads and stores to that same memory. Some code in the test bench is memory mapped to set and clear an interrupt signal to exercise the IRQ functionality, and some other code memory maps test control to

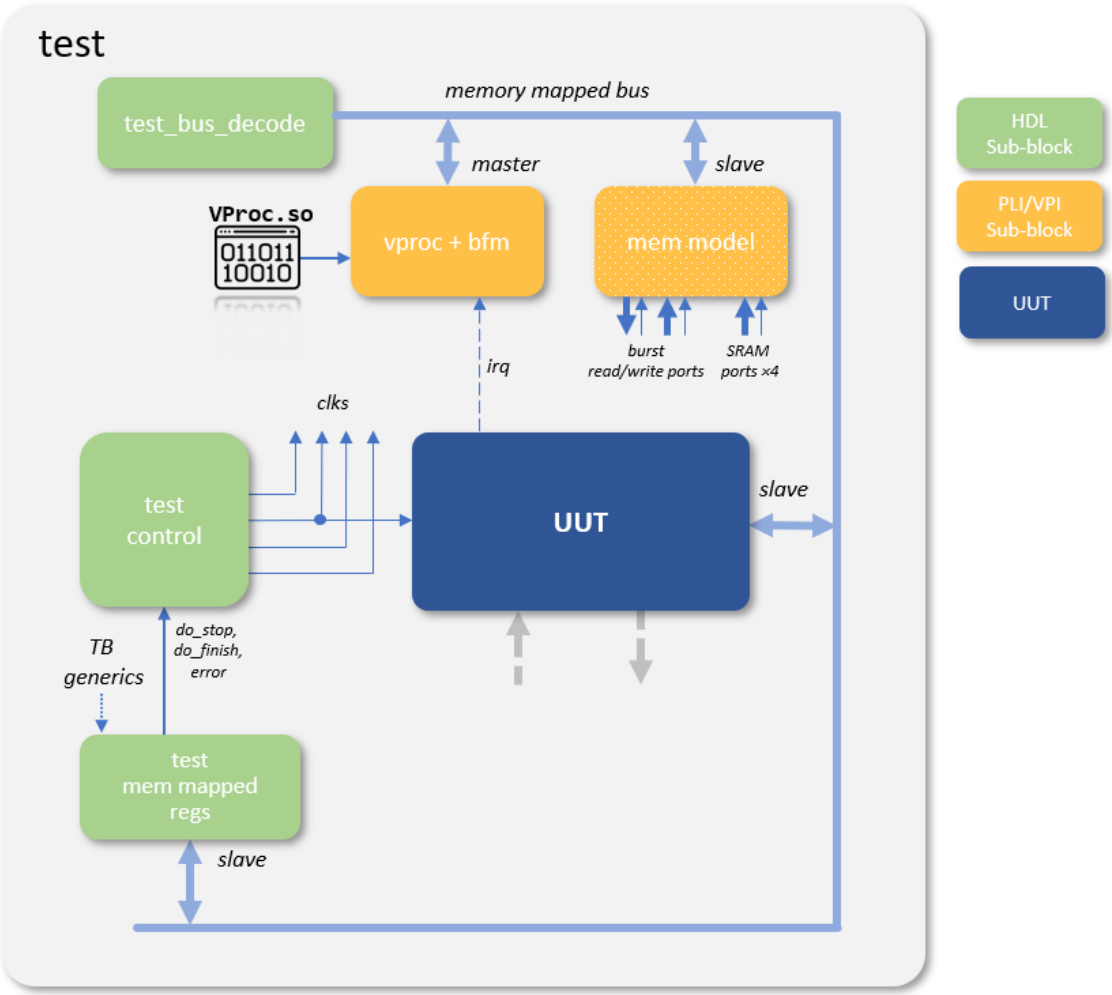
allow the RISC-V program to stop or finish the simulation. In the VProc repository, the RISC-V rv32 ISS model is provided as a set of precompiled libraries in `examples/riscv/lib` for the various possible platforms and architectures. The test make files will choose the correct library to use. The headers for the model are in `examples/riscv/include`. A simple precompiled RISC-V test program is provided, along with the original source code, in `example/riscv/interrupt/test.exe`. This loads the program, and then runs this test to set up generating an interrupt, which calls an ISR, and then checks it functioned correctly. It then stops the simulation, with register state to indicate a pass or fail.

The example is simple to allow the basic principles to be demonstrated without obfuscation with complexity, but this basic setup is expandable for more elaborate scenarios with more complex bus/interconnect infrastructure, more memory mapped devices, caches, MMUs or whatever is appropriate for the system requirements. Both HDL implementations of devices or software models of devices can be mixed, with the external memory callback function using the VProc API when targeting devices mapped in HDL or calling model APIs functions when modelled in software.

This, then, gives the basis for a co-design environment where embedded software is compiled for the intended target processor and can drive a model that could, in the limit, be all modelled in software, but transition to replacing these with HDL implementations when available or as appropriate. VProc allows multiple instantiations of the component and so multi-core environments are also possible with this setup.

VProc and HDL Test Benches

One or more VProc HDL components are meant to be instantiated in a test bench environment. Typically, the VProc component will be wrapped in a bus-functional model (BFM) to translate the generic bus signals into a protocol specific signalling, such as AXI for example. The diagram below shows a simple test bench that uses VProc, wrapped in such a BFM:



Here a VProc component is wrapped with a BFM which has a master/manager memory mapped bus port. On that bus, in this example, is an optional memory model PLI component and the unit-under-test (UUT). This could be a single module for unit testing with a VProc test program, or the top level of a complex hierarchy where VProc is running embedded software. Depending on the UUT, it may be connected to other test bench components to drive its other interfaces. Also mapped onto the bus are test bench registers that can be read and written so that the VProc program can control the simulation, such as stopping the simulation at the end of a test. These might go to a test control block to action the controls set in the registers,

or to funnel status back to the registers for reading by the VProc program. A test address decode block maps the test and UUT modules into the memory space. If the UUT generates interrupts, then these can be routed to the VProc component.

Variations on this basic theme are, of course, possible, with multiple VProcs, UUTs with master bus interfaces (and thus bus arbitration), and other verification IP blocks.

Using Multiple VProc Nodes

What is a VProc Node?

In short, a VProc node is an instantiation of a VProc component that acts as source of memory mapped transactions that has a unique node number, as set by the node input port, acting independently of any other VProc instantiation, and running its own virtual program. This allows VProc to be used to model a system with multiple cores.

The node number is critical in delineating the various instantiations of the VProc HDL component. All VProc components take a Node input, and this connects the instantiation with the user software that is driving the generation of memory mapped transactions. Each VProc instantiation that drives a given memory mapped bus must have a unique node number, and must also be static for a given instantiation—no dynamically changing the `Node` input port during the simulation. This ensures that the user software running for a given node is the only source for transactions to a particular VProc.

User Node Programs

The user programs, as we established in the last section, have a given entry point for each node but, from then on, they can have any complexity of hierarchy, with sub-routines, classes etc. to organise the flow of the program. The API (via either the V*xxx* functions of the C API or the `VProcClass` C++ API) can be called from anywhere within the user source hierarchy. So, we can have multiple nodes, each running a unique user program, as if running on its own independent processor.

Communication Between User Programs

Now it happens that the synchronisation mechanisms that have been put in place in the VProc software (not coincidently) ensure only one of the user program threads of the simulation are running at any given time, whilst the others are blocked at an API call. As well as ensuring safe communication between the user program threads and

the simulation, it also has the benefit in that it allows safe communication and synchronisation between the user programs on different nodes as well, and without the need to use thread synchronisation methods in the user source code. There are some things we must take care of, but these are just programming requirements, and do not require calls to OS features.

Exchanging data

In a normal multi-threaded program, care must be taken when exchanging information between these free running threads. If data from one thread is only partly constructed when the thread is de-scheduled and the receiver thread is activated, the program can fail to act as intended. This is still true of the VProc user code threads—accept now that has already been done by the underlying VProc software. Therefore, any data can be constructed for reading by another VProc's user program safely. Not until a new call to a VProc API class method is made by the user code that's generating the data destined for use by another VProc's user program, will that user thread be de-scheduled, and the target program have a chance to be activated and read the exchanged data.

The method of data exchange is entirely a choice of the programmer, be it a single variable, or a complex data structure. The main point is that its generation is 'atomic' and requires no further action by the programmer.

Synchronisation

As well as exchanging data between user programs, it is useful if they can be synchronised to co-ordinate that data exchange.

Since we already know that data exchange is safe between user programs, we can use a data variable as a 'barrier'. A simple integer variable, initialised to 0, can be used as such a 'barrier'. The responder increments the barrier variable whenever it wants the transactor to advance to its next transaction generation. The transactor 'waits' on the barrier variable to increment before advancing.

So, what do we mean by 'wait'? A user program cannot simply loop internally on some program variable state that it relies on to be updated from the simulation, otherwise the simulation will hang as no simulation time passes. Given that other user programs will, therefore, not be advanced, waiting on any state change from them will also cause a hang. A user program can loop waiting on a barrier variable, so long as it allows simulation time to advance. The simplest way to do that is to tick for a cycle at each iteration of the loop. For example:

```
extern int barrier;
static int last_barrier = 0;

static void waitOnBarrier(void) {
    VProc  vp(node);

    while (barrier <= last_barrier)
        vp.tick(1);

    last_barrier = barrier;
}
```

This is the simplest form of sychronisation. A dual synchronization can be achieved using the barrier variable for 'acknowledging'. For example, if the code waiting on a barrier is released when incremented, it can do some processing and then decrement the barrier variable to indicate it has completed an action. The barrier incrementing code can then wait on the barrier returning to zero to know that the other user program has completed some process. Thus, the two programs can be locked-stepped.

The barrier variable works a lot like a semaphore, and this is because the user programs are synchronised on semaphores in VProc under the hood. Therefore, any action that can be done with a semaphore can be emulated with this kind of use of barrier variables between the separate nodes' programs.

Using Multiple Threads within a Single Node

Up until now, although each VProc node's user program is actually running in a separate thread, we have assumed that each user program for a given node is a single thread of execution. It may have any level of hierarchy, but the assumption has been that it is a single thread. This, though, is not a limitation required by the co-simulation environment.

The co-simulation layer provides for the situation where the software running on a given node is driven by multiple threads. The handling of this might have been passed on to the user code which would then have to take steps to manage the co-simulation API interface as a shared resource, but this is taken care of within the VProc software layer. At the heart of the VProc software is an exchange function that manages the data between user code and the logic simulation. At the point of exchange, where a user thread is blocked and the logic simulation is released, the code is guarded using semaphores so that it becomes an atomic operation. There is a semaphore setup for each node so that each node does not interfere with other nodes program flow but, within a node, access to the co-simulation features, via the

API classes, is thread safe, and each thread can drive the interface without further coding requirements.

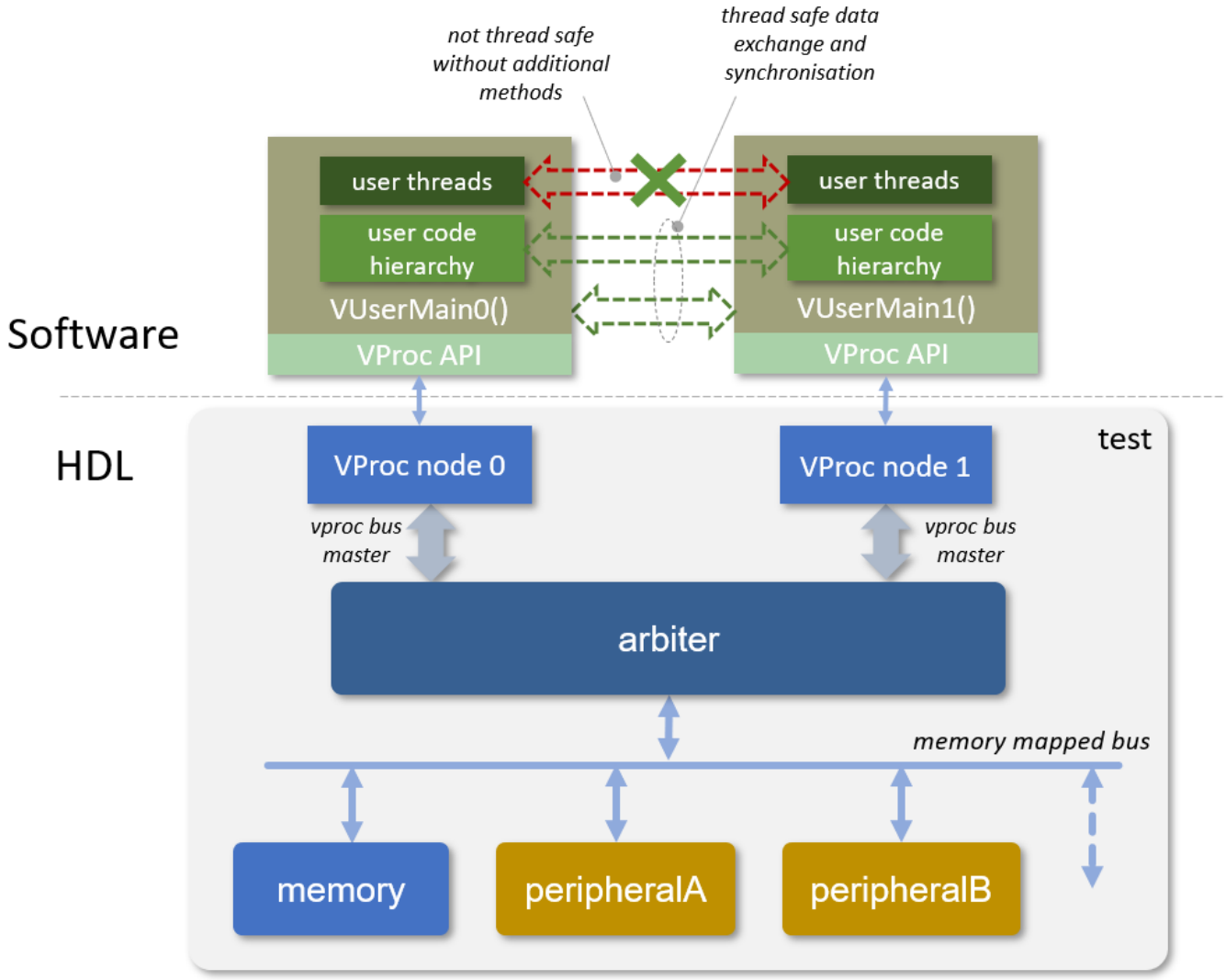
A use case for such a requirement might be that the code running on a node is retargeted multi-threaded embedded software, cross-compiled for the VProc co-simulation platform, making read and write accesses over a memory mapped bus. With memory access to certain address regions intercepted and sent to the simulation via the APIs, a true co-simulation environment is achieved with embedded software co-simulating with its target logic hardware in simulation. Since the embedded software can be multi-threaded, the APIs need to be thread safe—and they are.

Modelling an Event Based Software Architecture

Using the vectored interrupt input and a user registered callback, as described earlier in the document, an event based software architecture can be modelled that has an event loop. The user interrupt callback function is constructed to simply receive the changes in interrupt state and make it available to other parts of the program. It can also use a barrier, as described earlier, which it updates on changes to the interrupt state, to allow synchronisation from external functions.

A user software program running on a given VProc node can then be constructed where the code is a loop waiting on the barrier from the interrupt node, in a manner we looked at earlier, ensuring advancement of simulation time. When released, the code can then process the new state to call various functions to perform the required actions for the current interrupt state, which will be, ultimately, the memory mapped transaction calls to handle the interrupts, accessing the registers in the HDL to inspect status and clear state. This main program loop, then, is the event loop servicing the incoming events on the interrupt lines.

The subject of this section on multiple instantiated VProc nodes is summarised in the diagram below which shows a simple SoC environment with two VProc cores, memory, and several peripherals. In this example an arbiter arbitrates between the cores onto a single memory mapped bus:



Limitation 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 achieved—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.

Currently, the VProc API for Python can only support one instantiation of the component due to the limitations of the Python/C interface, but a solution is being developed.

Conclusions

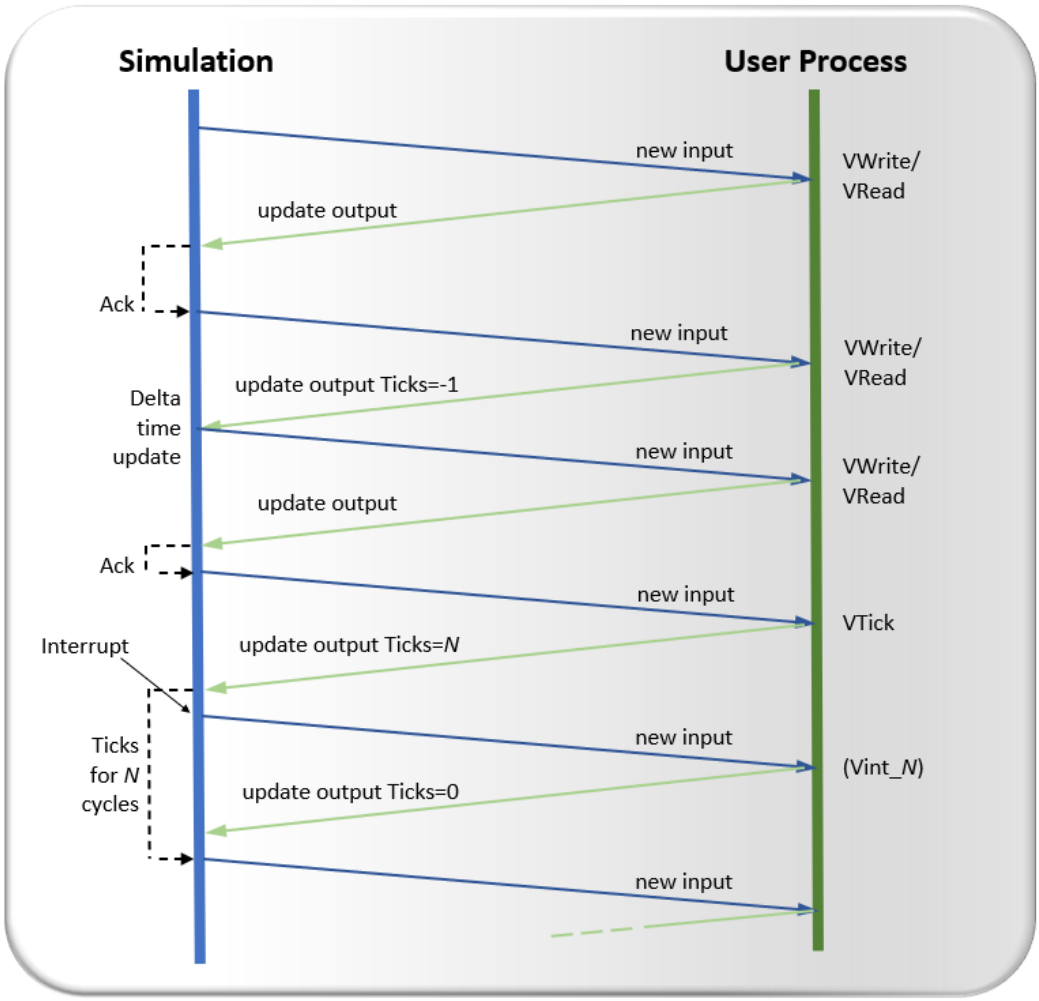
A fundamental co-simulation element, `VProc`, has been described which virtualises away the C and simulation interface to give a basic processing element, controllable by host compiled code. This basic element provides enough functionality such that any arbitrary processing element with a given bus can 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 from github 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.

More information on the virtual processor can be found in two articles. Part 2 of the set of articles on the programming interfaces of logic simulators describes VProc as a real-world use case example. Another article on the VProc Virtual Processor describes in more detail its internal workings.

Appendix A: 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 minimum, 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 above), 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 arbitrary 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.