# Nested Vectored Interrupts and Co-simulation

Simon Southwell

January 2025

# Preface

This document is from an article written in January 2025 and published on LinkedIn, covering how to model nested vectored interrupts in a *VProc* based co-simulation environment.

Simon Southwell
Cambridge, UK
January 2025

# Contents

# Introduction

In general, co-simulation, in the context discussed in this document, is the ability to simulate in both the native HDL language (or languages) supported by a logic simulator and another non-native language such as C. The HDL languages themselves specify, as standard, various programming logic interfaces, such as VPI and DPI-C for Verilog and SystemVerilog, and VHPI for VHDL. These are interfaces specifically to C code, and the various logic simulators support one or more of these to varying degrees. The programming interfaces provide a lot of different features, but a common useful feature is the ability to call a C function as a system task (Verilog), a VHDL foreign task or even directly (SystemVerilog and DPI-C). Input arguments can be passed in, and data returned in output arguments or as a returned value. The limitation of this is that it is a call and return system. The C function is just adding to the tasks/functions of the HDL but written in a foreign language. The C code has access to all the features of code compiled for the host machine that the simulator is running on and, if a complex function, can run much faster than an equivalent written in HDL. So this is still very useful. What it isn't, though, is a C program running as an independent block concurrently with the HDL instantiated modules/components.

The definition of co-simulation we have so far is quite broad—any foreign code running within the logic simulation is co-simulation. We might narrow this down a bit by saying co-simulation is a *program* running concurrently with the logic blocks and interacting with them. This kind of sounds like the function of a processor. Suppose we had an HDL module that had the basic interfaces of a generic processor—one that isn't any specific implementation—and has the basic interfaces of such a processor core. At its simplest, a processor core has a master memory mapped bus or interconnect interface to load and store to memory mapped locations (glossing over potential IO space or different memory spaces). In addition, it has one or more interrupt inputs. And that's really all.  If we had a means to run a C program, and by extension a C++ program, that could drive the bus and respond to interrupts we would be co-simulating a *program* with the logic.

The *VProc* virtual processor is a piece of verification IP (VIP) that can do just this. It has a generic memory mapped master address bus and a multi-bit interrupt input. It is this interrupt that is the subject of this document—how can we model interrupting our program "running" on the virtual processor. I have written about *VProc*, how it all works and can be used, in another article, including how a user supplied function can be called whenever the interrupt input port of the HDL component changes value.

This, in itself, won't interrupt the user program but, with some additional code, it can be made to do so.

I should point out that the methods described in this document use *VProc* as an example but are as easily applicable to other environments, such as a pure C++ model of an SoC, where state can be updated with interrupt changes that will alter the flow of running code modelling the application software. Such an example system might be for developing embedded software in a software SoC test harness model that generates appropriate responses to memory mapped accesses for the various peripherals, used before real hardware is available. Other verification environments, such as OSVVM support co-simulation and use similar techniques (as I contributed these myself).

## How Interrupts Work in a Processor

Before diving into discussing how to model interrupts in a co-simulation setup it might be worth recapping on how interrupts work in an actual processor at the logic level. This obviously varies between processors, but the same principle is used in all of them. If you are familiar with these concepts ten you can skip to the next section.

In the normal course of events a processor core will read instructions, keeping tabs of which address to find the next instruction in a Program Counter (PC). Let's assume, to keep things simple, that the processor is pipelined and can read an instruction every clock cycle, ignoring wait states on memory and latencies on branches etc. Most instructions will cause the PC to increment by the number of bytes the instruction takes (e.g., 32-bits, or 4 bytes). Some instructions, such as branch or jump, will alter this steady increment and force the PC to be a different value—perhaps relative to its current value, or to some absolute address. The program thus proceeds as a single thread of execution.

Interrupts are a source of external 'exceptions'. There are other sources of exceptions, including illegal instructions, memory access faults, and even specific instructions to cause an exception (e.g., break). We'll focus on interrupts though, but the mechanism is the same for all of them. An interrupt will be an external signal connected to a port on a processor core's top level as an interrupt request. This might be a vector of inputs, if the core has within itself logic for interrupt controller functions. As often as not the core all only have a single input signal and rely on an external interrupt controller—for example, RISC-V has a single external interrupt and utilizes an external PLIC. On the Other hand ARM's Cortex-M3 has its nested vectored interrupt controller (NVIC) as part of the processor.

There will be means to enable or disable interrupts, perhaps a master enable along with individual masks for particular interrupts (and other exceptions). Now, at each update of the PC, the processor logic will inspect the interrupt input, along with the enables, to determine whether there is a pending interrupt request and whether it should act upon it. If all the relevant enables are set, when an interrupt request input goes active the logic will override the instruction PC calculation and will set the PC to a particular address, known as the interrupt or exception vector. It will also save off the address of the instruction it would otherwise have executed. Other information needs to be saved, either programmatically or via logic, to restore the state when the interrupt returns, but we'll focus on the PC.

The interrupt vector will be some fixed location (though perhaps can be moved with a write to a configuration register). Different interrupts and exceptions may make the PC jump to the same location, or each type may be a particular offset from the base interrupt vector address. Code will then start executing from this new location, and this code is known as an interrupt service routine (ISR). It will continue to flow through this code, possibly branching off to some other program locations, until it reaches a particular instruction to 'return from interrupt'—on the RISC-V processor this is mret (if in machine mode). At that point the PC is set to the address saved when the interrupt was taken, the other saved state restored, and the original program continues as before.

With one interrupt input, that's all there is to it at the logic level. If there are multiple interrupt request inputs, then these may have a priority set between them, with some taking higher over others. If a lower priority interrupt is active and a new higher priority interrupt is activated (and enabled), then the ISR of the lower priority is, itself, interrupted (and the PC address saved, in addition to the original PC of the main code). This allows for 'nested' interrupts. Similarly, if a higher priority interrupt is being serviced and a new lower priority request comes in, this will be flagged as pending but will not cause the running ISR to be interrupted. When that ISR completes, though, the flow will not return to the main code's saved address, but a new interrupt call made for the pending lower priority and only when that ISR completes will flow return to original PC address of the main code (assuming no new interrupts or exceptions).

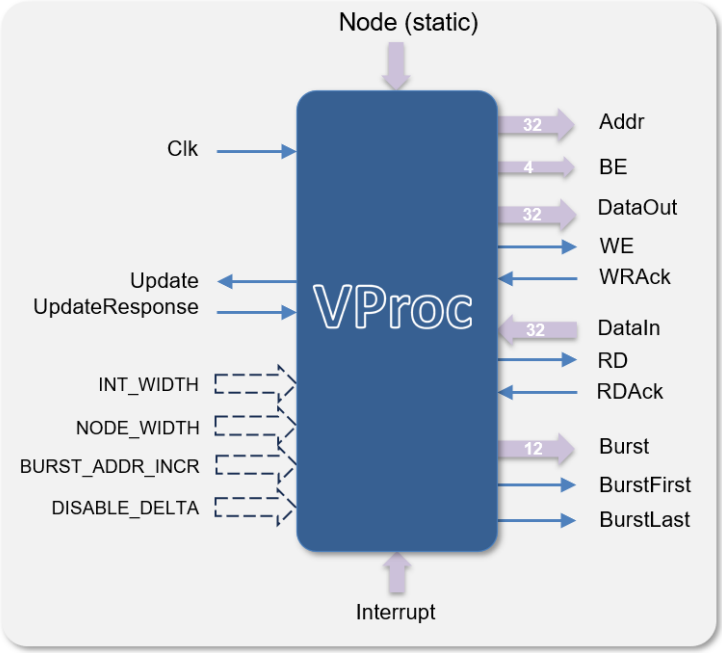
Before completing this overview, I want to mention that despite an external signal causing the PC value to change, an interrupt is equivalent to a jump instruction and an interrupt return is equivalent to a subroutine return. Whilst running ISR code, the processor core is doing exactly what it was doing before. It is in no special state and only the interrupt logic is keeping tabs on the active and pending interrupt state. In

other words, there is still only one thread of code being executed. In addition, the logic is checking for possible exceptions at every PC update, effectively polling the state at this resolution. This is important to bear in mind when we look at modelling interrupts and that we can do so without resorting to complex program control—actually, really interrupting user application code is not normally available to ordinary programs and is all hidden within the operating system which then provides services.

## *VProc* Summary

We now have an idea of what we want to model and have mentioned that our example modelling platform will be the *VProc* virtual processor. So before diving into interrupts let's summarise *VProc*'s main features.

*VProc*, at the logic level, is an HDL component with a generic 32-bit bus and an `Interrupt` input port, along with various configuration parameters and a clock input. There are also features to allow for "delta-cycle" updates, but this is not relevant to this document's subject.




The above diagram shows the *VProc* module that can be instantiated as Verilog or VHDL in a logic simulation. The burst signals and the byte enable signals are optional and the `Node` input is used to differentiate between multiply instantiated *VProc* components, which must all have unique node IDs. In the normal scheme of this, the basic *VProc* component would be wrapped in a bus-functional-model module to translate the generic bus to a particular protocol, be it a standard such as AXI, or proprietary, and some such are supplied in *VProc*'s github repository. So this takes care of the HDL side.

For each *VProc* instantiated, it expects some software to be provided for it to run. For many logic simulators the programming interface code is compiled into a shared object (or dynamically linked library in Windows terminology) that is loaded at run-time. Some others, such as *Verilator*, can have the source code compiled directly, or supplied as a statically linked library. The details are not important. The *VProc* software supplied takes care of the connection to the simulator and provides a simple API to instigate reads and writes on the bus of the HDL component, for use by the user program to be run on the virtual processor.

The user program is compiled along with the *VProc* software layer and each "node" is expecting the user code to supply a C function named `VUserMain<n>`, where `<n>` is the node number. So, if a single *VProc* component is instantiated and has its node input set to 0, the user code must include a "main" entry point function as `VUserMain0()`. The user code can be compiled as C++ so long as the `VUserMain<n>` functions have C linkage using `extern "C"`. From the `VUserMain` entry point, any structure of a normal C or C++ program can be used and the *VProc* API called from anywhere in the hierarchy. Both a low-level C and a C++ API are supplied and abbreviated prototypes are shown below.

C API

```
VWrite   (addr,  data,     delta, node);
VWriteBE(addr,  data, be, delta, node);
VRead    (addr, *data,     delta, node);
VTick    (ticks,                  node);
```

C++ API

```
vp->writeByte  (addr,  data, delta=0);
vp->writeHword (addr,  data, delta=0);
vp->writeWord  (addr,  data, delta=0);
vp->readByte   (addr, *data, delta=0);
vp->readHword  (addr, *data, delta=0);
vp->readWord   (addr, *data, delta=0);
vp->tick       (ticks);
```

Basic read and write function and methods are supplied. There are "delta" parameters, but these can be ignored for this discussion and should be 0 (the default in the C++ methods). The node parameters of the C functions should also match the node of the *VProc* component that the program is running on. For C++ this was set at construction of the C++ API class (called `VProc`).

An additional "tick" function/method is supplied. This allows simulation time to advance, without instigating a transaction, in units of clock cycles. The running program executes infinitely fast with regard to simulation time and so, if the program wishes to do nothing for a period without a read or write on the bus, the tick API must be called to allow the simulation to progress.

This takes care of the normal flow of the program running on a *VProc* component, but what about interrupts?

## Getting Interrupt State from Logic to Software

The user program can register a function to be called every time the `Interrupt` port of the HDL component changes state. The code fragment below shows the function pointer definition and the C++ API method to register the user interrupt "callback" function.

```
typedef int (*pVUserIrqCB_t) (int);

void regIrq (const pVUserIrqCB_t func);
```

The user function has a simple prototype of a single integer input and returning an integer status (usually 0). The parameter passed in is the new value of the `Interrupt` port. *VProc* is agnostic to whether the signals are active low or active high and simply passes in a new value on each change on the port. It is expected that the user callback will store away the input state in some convenient storage for use by the running program. What it won't do is interrupt the running program at the point of the call to the user interrupt function. The callback function is a simple call and return PLI function and executes in zero simulation time. It can't run interrupt code directly from there (in fact, for various reasons, it can't make API calls without consequences) but, as we shall see, this is not a problem. Assuming we stored away the interrupt state when the callback was executed, we can model interrupting the main program code with some simple trickery.

## Responding to Changes in Interrupt State

I said before that program code runs infinitely fast with reference to simulation time. For the interrupt signal, that changes at cycle boundaries, we can't get a look in at any particular instruction or line of code. Except simulation time does advance if we call an API method for reads and writes or for ticks. This is where we can do something to change behaviour of the program flow.

Imagine we write a derived class as a child of the *VProc* C++ API class, `VProc`. In it we wrap up all the API read/write and tick methods in new methods with the same names and parameters, which call the equivalent methods of the base class. (We can emulate all this with the C API as well.) Before we call the base class methods, though, we can call another method first—let's call it `processIrq()`. This method will inspect the state of the `Interrupt` port as saved by the callback and, if other state enables an interrupt for the given state, a function will be called that implements the appropriate interrupt service routine (ISR), if one exists. A class is supplied with the *VProc* software that does exactly this, called `VProcIrqClass`.

# The *VProcIrqClass*

The `VProcIrqClass` has a one to one correspondence with the all the read, write and tick methods of the `VProc` class. The other methods of `VProc` are inherited directly, such as the interrupt callback registering method and so on. In addition to these wrappers, and the `processIrq()` method, which we shall come to, it adds other API methods to allow full nested vectored interrupt modelling as well as edge and level trigger interrupts. These additional methods are summarised below:

```
void enableInterrupts      (void);
void disableInterrupts     (void);
void enableIsr             (const unsigned int_num);
void disableIsr            (const unsigned int_num);

void updateIrqState        (const uint32_t newirq);

void registerIsr           (const pIntFunc_t isrFunc, const uint32_t level,
                            const bool enable = false);

void setIrqAsEdgeTriggered (const uint32_t mask);
void clearEdgeTriggeredIrq (const uint32_t level);
```

These methods allow user code to control enabling and disabling of interrupts, with master control (`enableInterrupts`, `disableInterrupts`) and individual level control (`enableIsr`, `disableIsr`). The `int_num` is an index for the bits of the `Interrupt` input port vector of the HDL component, so each bit is controlled individually. In the `VProcIrqClass`, bit 0 is the highest priority, with priority going down as the index increases.

Changes to interrupts state are stored away internally with a call the `updateIrqState` method. This is usually called from within the interrupt callback function which will need access to the instantiated `VProcIrqClass` object in order to do so.

For each of the `Interrupt` bits a user function can be registered as the interrupt service routine (ISR) using the `registerIsr` method. The callback prototype is a simple `void` function with a single input parameter where the current state of the interrupts is passed in.

Finally, the `Interrupt` port bits can be flagged as edge triggered with the `setIrqAsEdgeTriggered` configuration method and a single `mask` parameter, with high bits indicating which are edge triggered. This allows the model to make the bits sticky until processed, as the input may disappear. A `clearEdgeTriggeredIrq` method is provided to clear the sticky state. This would normally be cleared on

entering the matching ISR for that particular edge triggered bit. This does mean that a new edge triggered pulse might be missed if the matching ISR has not been called since the first interrupt, but this is true of real interrupts as well if an active edge triggered interrupt is not serviced due to active higher priority interrupts when a new edge interrupt occurs.

So we now have control over interrupts, a means to set up ISR function to be called and to flag any edge triggered interrupts. We return now to the afore-mentioned `processIrq()` method to bring this all together to model nested vectored interrupts with priority.

## Processing Interrupts

We have seen that things have been setup to call a `processIrq` method before each API call to the logic simulation, and all the functionality can be captured in here to model nested vectored interrupts and to call user registered ISR callback functions. This is not as difficult as you might think and the bulk of the work is deciding if an active, enabled and higher priority interrupt has asserted itself and then calling the ISR associated with it, if one has been registered. An outline of this method is shown below.

```
void processIrq()  {
    if (interrupt_enable) {
        uint32_t int_new_int = isr_enable & ~int_active & irq;

        for (int isr_idx = 0; isr_idx < MAXINTERRUPTS; isr_idx++) {
            bool higher_active = int_active &
                                    (0xffffffffULL >> (MAXINTERRUPTS-isr_idx));

            uint32_t int_unary = 1 << isr_idx;

            if (int_active & ~irq & ~edgeTriggered & int_unary)
                int_active &= ~int_unary;

            if ((int_new_int & int_unary) && !higher_active) {
                int_active |= int_unary;

                if (isr[isr_idx] != NULL)
                    (*(isr[isr_idx]))(irq);
            }
        }
    }
}
```

The first thing to notice is that the whole functionality is bracketed by whether a master enable is set (`interrupt_enable`) or not. It is this internal state that is set or cleared by the `enable_interrupts` and `disable_interrupts` API methods.

Next we need to detect any new interrupts (i.e. become active that weren't until now) that are also enabled, and this gets set in the local `int_new_int` variable, which is a unary bitmap of flags for each interrupt bit. The `isr_enable` interrupt bitmap variable is the one controlled by the `enableIsr` and `disableIsr` API methods. The `int_active` bitmap variable indicates which interrupts are pending, with `irq` reflecting the interrupt input port state and set by the `updateIrqState` API method.

After this, a loop goes through each interrupt from highest priority to lowest with, in this case, the highest priority being 0. Within the loop, a calculation is done to flag if a higher priority interrupt is active than the interrupt of current level in the loop. This can never be true for level 0 but, for simplicity, the calculation is done for this index anyway. After this, the current interrupt index is turned into a bitmap unary value for use with the other bitmap values.

Following this, a test is done to see if the interrupt input port bit for the looped interrupt index is now low, but the interrupt is flagged as active. This will clear the active state so long as it's not an edge triggered interrupt, where this bit is sticky and will be cleared by an external call to the `clearEdgeTriggeredIrq` API method. The `edgeTriggered` bitmap variable is that which is set at configuration with a call to the `setIrqAsEdgeTriggered` API method.

After this, a test is done to see if, for the indexed interrupt, it is a new interrupt and no higher interrupt is active, in which case it is marked as active and, if an ISR was registered for this level, the callback function is called.

## How this Manifests as nested Interrupts

It might not be obvious from the description above how this all comes together to produce a flow of main program and then nested interrupt ISR execution. So let's have a walkthrough of some cases and what might happen. We'll assume that only two interrupts are enabled—level 0 (highest) and level 1 (lowest) and that they are enabled, the master interrupt is enabled, and both have ISR functions registered. Let's also assume that, at the start, no interrupts are pending and the main program (as started running from `VUserMain0`) is executing.

The main flow of user code will be making calls to read, write and tick methods from the *VProc* API of the `VProcIrqClass` object it created, and `processIrq` will be called

at each of these. If, say, the `Interrupt` port's bit 0 goes from 0 to 1, then this will go from inactive to active at the next API call and (by definition in this case) will be the highest priority. Thus, the registered ISR0 routine will be called effectively interrupting the main program. The ISR0 function will itself execute its code, including calls to the transaction routines, where new interrupts will be monitored for in `processIrq`.

If, whilst running ISR0, `Interrupt` bit 1 goes high, this will be ignored, filtered on the fact that it's not the highest active priority interrupt. However, as soon as ISR0 returns, ISR1 will be called as the highest outstanding priority new interrupt. When it returns, and no outstanding new interrupts, the main program's call to the API method will return and the main program flow will resume. If we consider that interrupt bit 1 was set first, then ISR1 is called. If bit 0 goes active whilst in ISR1 then this will be interrupted and ISR0 called. When ISR0 returns, ISR1 resumes, and when ISR1 returns, the main program resumes. And so on, extending up to 32 active interrupts supported.

By doing it this way, from the perspective of program flow, it looks like we have interrupt capability with priority nesting but, actually, we have a single thread of execution. The details are abstracted away with the `VProcIrqClass` and its `processIrq` method. It does mean, though, that interrupts are only processed at the API methods identified earlier, rather than at general lines of code or even at instruction level. This adds a "granularity" and latency in servicing interrupts but, remember, non-API call code runs infinitely fast with respect to the simulation, so this isn't really much of a restriction. One consideration does have to be taken into account though with regards to delays.

I said previously that the tick methods allow simulation time to be advanced when the code wants, say, a delay. Imagine a `usleep(delay)` function is constructed to pause the code for a set amount of time in microseconds. Suppose internally, it has knowledge of the clock rate and can calculate the number of clock cycles that matched the specified microsecond delay and will call the tick method with that number of cycles. The simulation will then advance that many cycles before returning and the tick method unblocking. Job done. But if an interrupt goes active at the beginning of that period it won't get serviced until the delay is completed. This might be acceptable but is not ideal for all cases. The way around this is, if using interrupts in the system being simulated, it would be better to model this as something like a for-loop for the number of ticks around a call to `vp->tick(1)`. This is not as efficient as there will be more calls and returns across the programming interface, but it does

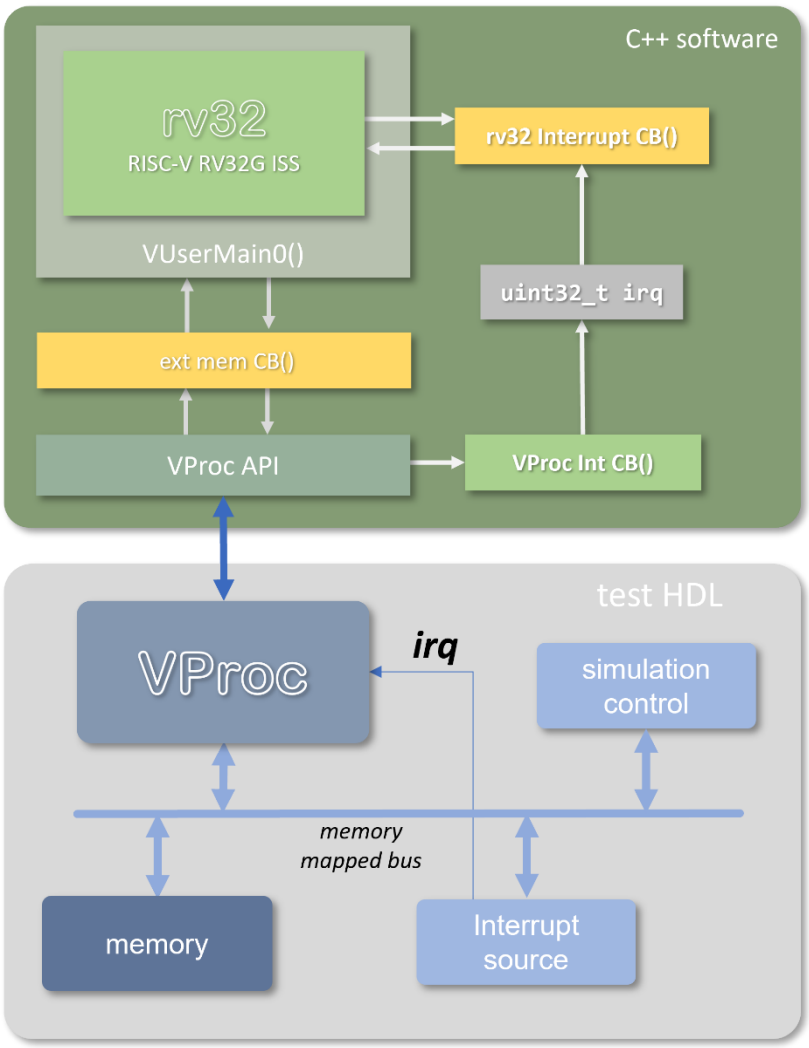
mean that an interrupt going active will be serviced immediately, whilst giving the correct behaviour for the `usleep` function.

The granularity of interrupts at the transaction API calls is a minor restriction, but is there a way to get interrupts at the granularity of instructions themselves? The short answer is yes—sort of.

## *VProc* Interrupts and Instruction Set Simulators

I said at the beginning of this document that the *VProc* VIP allows natively compiled programs to "run" on an HDL component instantiated in a logic simulation, as if it were running on the virtual processor component itself. Since the programs have been compiled for the host computer they can be anything we like that we can imagine for running on that computer. This includes an Instruction Set Simulator (ISS). An ISS is a C or C++ model implementation of a particular processor's instruction set architecture (ISA) that can execute programs compiled for that ISA, for example RISC-V.

If we have an ISS model and if we want to run this as the *VProc* program we need a means to interface it to the API calls. The *rv32* RISC-V ISS has some features that make this fairly straight forward. We've already seen that the *VProc* API has means for reading and writing and these match the load and store instructions of the RISC-V ISA. We also saw that we can get interrupt state from the HDL and we can pass this to the RISC-V interrupt inputs. The *rv32* model allows two callback functions to be registered, one of which is called on memory access instruction execution (such as load and store instructions) and another to be called when it wants to read the latest interrupt input state. Consider the following diagram:




Here we see a system set up with a *VProc* HDL component instantiated and a memory mapped peripheral that is a source of interrupt to the virtual processor. Running on the virtual processor is the *rv32* ISS and it will call an external memory access callback on all load and store (and other memory access) instructions. That callback is written to make calls to the *VProc* API read and write methods to instigate transactions on the memory mapped bus as appropriate to the type of access. When an interrupt is generated in the logic simulation, then the *VProc* registered interrupt callback function is invoked, and this can save off the new value of the `Interrupt` port. An *rv32* interrupt callback function is also registered and is called by the ISS to inspect the IRQ state, as updated by the *VProc* callback. This will, in fact, be at every instruction boundary of the RISC-V program running on the model, and so interrupt state is inspected at every instruction.

Note that the HDL and software are simply processing the state of the interrupt signalling and is agnostic to what this actually means, or whether edge- or level-triggered etc. This is all taken care of in the ISS modelling. Also, since the model is

implementing interrupt functionality, the software does not need to use the `VProcIrqClass` for its API and can use the plain `VProc` class (or low level C API).

So we can have interrupt granularity at the instructions level, albeit by running an ISS model of a particular processor ISA as the program executing on the *VProc* component.

# Conclusions

In this document we tackled how to model interrupts in a software/logic-simulation co-simulation environment, with specifics for *VProc*. We did this is a clever way so that interrupting program flow looked like it was interrupted but, in fact, was a single thread of program execution, simplifying what was required to do so. An API 'wrapper' class was described to abstract the details away and to implement the enabling/disabling, prioritizing and nesting to give emulation of full nested vectored interrupts. The granularity of interrupts was restricted by the interrupt points being at the calls to read, write and tick API calls but since the rest of the code is running infinitely fast compared to the logic simulation, this was a minor restriction. If this is still a problem, we got to the point of modelling interrupts at an instruction granularity by making the *VProc* program running be an ISS. With some simple code, this was integrated with the *VProc* API and interrupts modelled as for a real processor.

These methods aren't restricted to a *VProc* based environment. For other systems with co-simulation features, these can work just as well. The OSVVM library and verification methodology has co-simulation support (as I added this) and has interrupt modelling in a similar manner. A pure C/C++ SoC modelling environment might also make use of the techniques described here. The main thrust is to model interrupts but without complex methods and as a single thread of execution.