# Pointers and the Power of Callback Functions

Simon Southwell

June 2025

## Preface

This document is a PDF version of an article written in June 2025 and published on LinkedIn, on callback functions and their use in modelling SoC systems and in co-simulation.

Simon Southwell
Cambridge, UK
June 2025

## Contents

## Introduction

In this document I want to talk about "callback" functions and the powerful things you can do with them. I don't want to assume too much prior knowledge and will build up in steps for clarity. I personally still have to look up basic things after nearly 40 years, so in this document we'll start at the beginning and tell the story. In this sense I will cover some aspects of C and C++ that may be unfamiliar, discussing the concept of pointers-to-functions as a fundamental concept needed to understand callback functions.

Once we have understood what callback functions are, I will give two examples of their use related to modelling processors in C++ and in co-simulation of software and HDL in a logic simulation. These can even be married up to integrate these two systems, as we'll see.

So, to start the journey, let's have a reminder of what a pointer is in, in C and C++.

## Pointer Revision

It is likely that you know, already, what a pointer is, and you can freely skip this part if you are comfortable with these concepts.

### Pointer Basics

Fundamentally, a pointer is a variable that holds an address in memory. Everything in a program—the program instructions, the initialised data, the heap, the stack, and so on—has an address somewhere in memory. A pointer can be set to any of these addresses. It, itself, will reside somewhere in memory and have an associated address. It doesn't have to, but it is most useful is if a pointer is assigned an address of some actual 'thing' such as the location of a variable.

```
#include <stdio.h>
int main (int argc, char *argv[]) {
    int variable = 123;
    int *ptr     = &variable;

    printf("variable = %d, *ptr = %d\n", variable, *ptr);
    printf("&variable = %p, ptr = %p, &ptr = %p\n", &variable, ptr, &ptr);
}
```

In the above C code snippet, an integer variable was declared (`variable`) and assigned a value, then a pointer to an integer was declared (`ptr`) and assigned the address of the variable (using the & symbol in front of `variable`). The first `printf` line displays the value of the variable and displays the value of the 'thing' pointed to by ptr, using * in front of ptr to "dereference" to the address contained in ptr. Of course, this will be the same value, as ptr is pointing to variable. The second line is, perhaps, more indicative of what's going on, printing the address of variable, the contents of ptr and then the address of ptr. The output, when the program is run, looks like the following (the exact addresses will be different each time it is run).

```
variable = 123, *ptr = 123
&variable = 0000008200dffb5c, ptr = 0000008200dffb5c, &ptr = 0000008200dffb50
```

You'll notice that the address of variable is the same as the *value* of ptr, whilst the *address* of ptr is another location in memory.

And this is the fundamental aspect of pointers. The example shows a pointer to an integer, but this works for floats and double, to literal strings and all the basic built in types of C or C++. More complex things can be constructed from the fundamental types, and we can use pointers for these as well.

## Pointers to Other Things

The first constructed element that we can point to is an array. If we have an array of integers (or whatever), we can point to the beginning of the array. The elements of the array are guaranteed to be contiguous in memory (that is, virtual memory—the system will take care of physical memory, and this need not concern us). Some quirks of the syntax means that the name of the array is also its address value (i.e. myArray is the same as &myArray[0]). We can dereference a pointer to the array as if it were the array itself (i.e. int *pMyArray = myArray; pMyArray[10];). This is really just an extension of what we saw before.

The basic types can also be gathered into structures and, as these also reside in memory, we can point to these as well. In C++ structures are extended to allow more than just variable types, with data manipulating functions also declared within the structure. In fact, in C++, a structure is *almost* identical to a class. The only difference is that a structures defaults to having everything public, whilst a class defaults to having everything private. This was done to allow migration of C struct declarations to be ported to C++ without having to modify them. These days one is encouraged to always use classes to get data hiding properties as the default.

So, even with the most complex struct or class, even with nested structures containing other structures etc., they all reside in memory and can all have a pointer assigned to them. The pointer will have match the type of the thing to which it can be assigned an address in order to be able to access the internal elements. As this gets more complex, things can be made easier using the typedef features.

## Typedef and Pointers

When, say, a structure is declared it uses the struct keyword followed by the name for the structure: struct simpleStruct_s {…};. A variable can be created from this structure with struct simpleStruct_s s; A pointer can be declared using the struct keyword again: struct simpleStruct_s *p = &s;. This is all well and good, but we can simplify things a little by declaring the structure with a typedef.

```
typedef struct {
        .
        .
        .
} sim_struct_t;
```

Here a new type has been declared, which will be the same as the simpleStruct_s from before but aliased as sim_struct_t. Using this type name we can drop the struct keyword, as the type has already been declared as a struct:

```
sim_struct_t s;
sim_struct_t *p = &s;
```

This might not seem like a big improvement, but I want to introduce the concept of typedef as it will come in vary handy in what we look at next to simplify, and thus clarify, the code we'll look at later.

## Pointers to Functions

We have seen pointers to data objects, but the code also resides in memory and so can be pointed to with a pointer variable. In particular, a function can be pointed to.

For the data pointers we still had to match the pointer to the type of object that it is to address, but functions don't have a data type as such. What they do have is a return type (possibly void) and zero or more parameters. A pointer to a function must match these attributes so the compiler knows about them when dereferencing through the pointer. We can use typedef to declare a new type for a function pointer with a particular set of parameters and return type.

```
typedef int (*pFunc_t) (int, int);
```

In the above code fragment, a new type is declared with typedef as pFunc_t. As it's a pointer, it is preceded with *, and then this combination is wrapped up in a pair of parentheses. The expected return type is then specified before this, and the parameter types after, just as if this were a function prototype declaration. We can now declare pointers of this type and assign them to functions that match the declaration—in this case a function that takes two integer arguments and returns an integer.

```
#include <stdio.h>

typedef int (*pFunc_t)(int, int);

int func (int a, int b) {
    return a * b;
}

int main (int argc, char *argv[]) {
    pFunct_t p = func;

    int accum = 1;
    accum = p(accum, 2);
}
```

In the above code, the function pointer type is declared and a multiply function, with the same parameters and return type, is defined. In the main code, a pointer is declared of type **`pFunc_t`** and assigned to the address of the function (like for arrays, the name of the function is its address). We can now use the pointer to call the function, just as if it were the function itself.

We are a step nearer to callback functions, but before this it is worth looking at pointers to functions for C++ class methods.

## Pointers to Class Methods

The pointers to functions we've looked at so far have been a very C like but, nonetheless, very useful. If we're working in C++, however, the question that immediately rises is "can we point to C++ methods?". The answer is, of course, yes.

### Static Methods

A static method, declared with the **`static`** keyword preceding its declaration, is a method that will be common to all instances of a given class and can even be called without reference to a class object but directly through the class name. In that sense it really belongs to the class itself. They have restrictions because of this; such is not having access to member variables which will have copies per object created all with different state.

A method would be declared static within a class if it is, say, a "pure" function. That is, it only references state from its passed in arguments, and returns (possibly) with a value, rather than updating internal object state. Thus, a single copy of a static methods is shared between all objects. This is not unlike a C function, with a single copy referenced from various places, and this is how we reference a static function.

```
#include <stdio.h>

typedef int (*pFunc_t)(int);

class tryClass {
public:
    static int increment (int value) {
        return value + 1;
    }
};

int main (int argc, char **argv) {

    pFunc_t meth_ptr = &tryClass::increment;
    printf("===> meth_ptr(%d) = %d\n", 10, meth_ptr(10));

    return 0;
}
```

In the above example, a type is defined, **`pFunc_t`**, that points to a function with a single integer argument and that returns an integer. A simple class, **`tryClass`**, defines a static method with this prototype and in the main program we can point to it without the need to construct a class object but by dereferencing through the class name, using **`tryclass::increment`**. This is almost identical to the C style but just using the class name to identify the function. To call the method, we just use the function pointer name, just as for the C style function pointer.

Since static methods have limitations, then it would be useful to point to non-static methods.

## Non-static Methods

It is possible to point to non-static methods, but a slightly more involved syntax is required. Since non-static methods are associated with the object instance of a class, we will have to dereference them through the object and not the class name.

```
#include <stdio.h>

class tryClass {
public:
    int increment (int value) {
        return value + 1;
    }
};

typedef int (tryClass::*pFunc_t)(int);

int main (int argc, char **argv) {
    tryClass *tryObj = new tryClass;

    pFunc_t meth_ptr = tryObj->increment;

    printf("===> meth_ptr(%d) = %d\n", 10, (tryObj->*meth_ptr)(10));

    return 0;
}
```

In the above example, the `typedef` declaration is almost the same as before, but now a reference to the class needs to be prepended to the front of the `*pFunc_t` pointer type name. The class now declares its method as a non-static and the main program declares a pointer to an object of this class and creates a new one to which it's assigned.

To point to the non-static function, a pointer of the type we defined is created and the member method is dereferenced by the class object instance we created. Not unlike before, but we have to use an object instance to get to a *particular* copy of the method. If we created another instance of `tryClass`, its method would have a different address, and we would reference it through the its object instance.

To actually call the method we can't just use the pointer name as before. The object instance must be used again, prepended, along with *; i.e. `tryObj->*meth_ptr`.

You can also point to methods within a class, referencing its own methods. I won't go into details but, instead of using an object name, the dereferencing is done with the 'this' pointer, which points to the particular instance of the class. I.e. `this` in place of the object name.

## Callback Functions

Having understood pointers to functions and methods the concept of a callback function can be introduced, and we will need pointers to functions for this. There is nothing special about a callback function itself and it is constructed just like any other function. A callback function is just the way it is going to be used.

Imagine we have a black box library where we have no access to the internal code. If it declares a type for a pointer to a function (in a header file, either standalone or as part of a class) we can create a pointer with this type in our own code. If, in addition, we can pass in this pointer to the library, having assigned it to point to a function that we've created, the library can then call our function from within its own code. This is a form of "injected dependency" which has many advantages for decoupling code. When the library calls this function it is said to "call back" the user function. To use an analogy, it is like giving someone at the reception of a company a phone number to pass on to an individual, and then they can "call back" when they are available.

Callbacks have many useful applications—too many to discuss here—but a couple of ways to use them is to call when some event occurs or some particular operation happens. For the rest of the section, I will discuss two examples of this to show what can be done and even bring these two things together.

## ISS Memory Access Callback

An instruction set simulator (ISS) is a software model of a processor that can run the instructions meant for a hardware implementation of the processor, but as a software model. The *rv32* ISS is just such a model for a RISC-V processor. A characteristic of the RISC-V processor, and other RISC architecture processors, is the separation of memory access instructions from the rest of the instructions. So, for RISC-V there are the basic load and store instructions and, if extensions are added, other type of memory access instructions such as for the 'A' atomic operation extensions.

An ISS might have an internal memory where the program and data is stored but, since it is only a model of a processor, a means is required to interface with external code to add memory mapped peripherals without the need to incorporate into the ISS source code. This is where a callback function is really useful.

The *rv32* ISS defines a pointer to a function type, as shown below:

```
typedef int (*p_rv32i_memcallback_t) (const uint32_t     byte_addr,
                                            uint32_t     &data,
                                      const int          type,
                                      const uint64_t     time);
```

Internally, *rv32* has a function pointer of this type which, by default is set to NULL. When the ISS executes a memory access instruction, such as a load or a store, before executing the memory access operations it checks this pointer. If it is still NULL then it will continue on with internal operations. If it is not null, then it will call the function first. The ISS also provides an API function to register a memory access callback function defined as shown below:

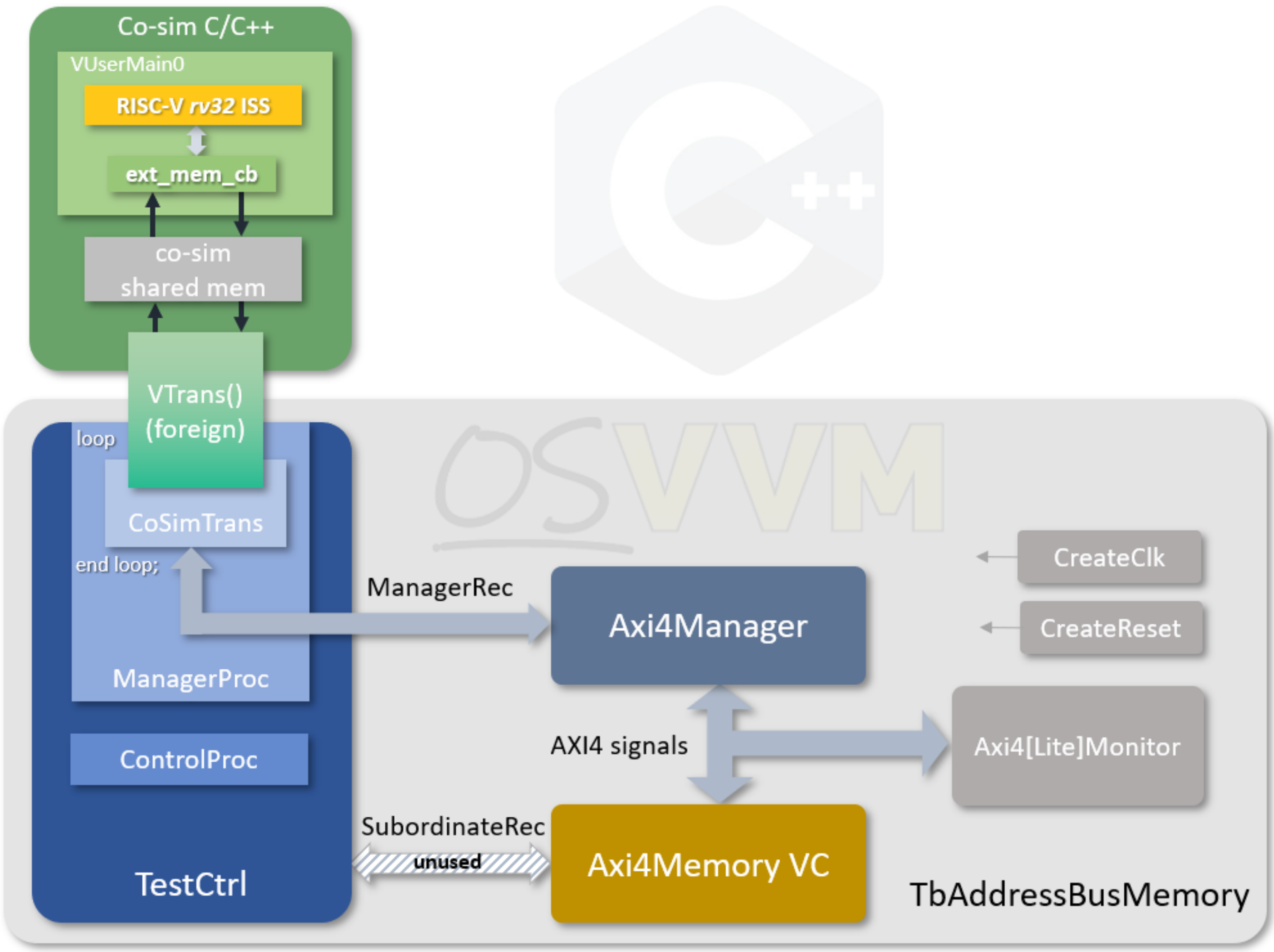
```
    void register_ext_mem_callback (p_rv32i_memcallback_t callback_func);
```

If this method is called, with a pointer to our own function, before the ISS's **run** method is invoked, then at each memory access our callback function will be called. The byte address is passed in along with a means to pass in data or send it back out. The type of the memory access is also passed in. This defines whether the access is a read or a write and of what size (byte, 16-bit half-word or 32-bit word). Finally, a cycle count is passed in. The ISS has an internal timing model, based on the number of cycles each instruction takes to execute, and accumulates a cycle count based on this. It is this value that is passed to the callback function, so it has a sense of time from the model.

The callback now has two choices of what it can do. It can do a decode of the address and decide it is not in the range of addresses that it can process. It can then return a value of `RV32I_EXT_MEM_NOT_PROCESSED` to indicate to the ISS that it didn't process the access. The ISS can then handle the access internally (just like when the function pointer was NULL). If it did process the access, then it can return a zero or positive number to indicate the number of wait states that the operation took. These will be added internally by the ISS to its cycle count.

Using this callback feature it is, hopefully, easy to see that the function can do address decoding and farm out accesses for processing by other software models of peripheral devices, and even build a hierarchy of decoding to reflect an SoC architecture.  I have written about this in more detail in a previous article. It can also be used to interface with some other software to communicate with a different modelling environment if that software has an API with read and write accesses that map to the types of accesses that the callback can handle.

The *rv32* ISS has been integrated to the co-simulation features of OSVVM using the callback features of the model. The diagram below gives an overview of the OSVVM integration of the *rv32* ISS. The model runs under the user **VUserMain0** user C++ code with an external callback registered (**ext_mem_cb**), and this makes calls to the provided OSVVM co-simulation API to instigate transactions in a logic simulation via common structure.

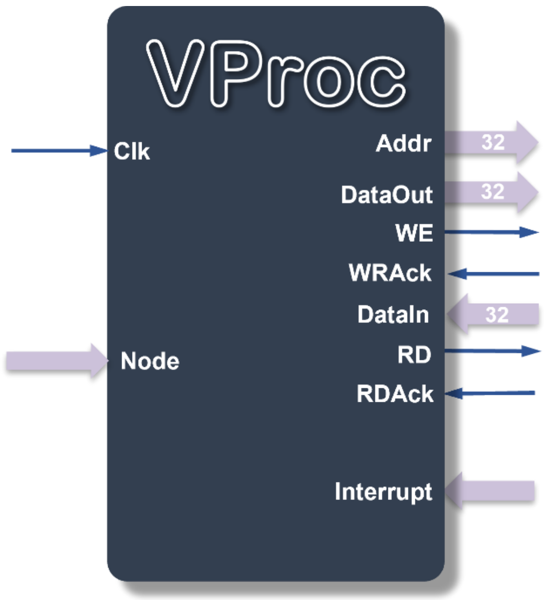


The OSVVM co-simulation features are based on the technology behind the *VProc* virtual processor, and the *rv32* ISS has been integrated with this in a very similar manner and is being used in this way on the Wireguard FPGA project out of Chili.CHIPS*ba.

As well as the memory access callbacks, the *rv32* also has callbacks for passing in interrupt status and for handling unimplemented instructions. The former is called regularly to read external state to reflect input interrupt values. The latter allows extension of the model to handle additional instructions not supported by the ISS to be added, either standard extensions or for custom and prototype instructions, without the need to modify the source code.

## VProc and Interrupt Callbacks

The *VProc* virtual processor allows a user written program, compiled for the host PC or workstation, to 'run' on an HDL component instantiated in a test environment running on a logic simulator, instigating reads and writes on a memory mapped bus in the simulation by making calls to C or C++ API functions provide for the user code. *VProc* also supports interrupts to allow for a complete model of a basic processor core functionality. A simplified diagram of the *VProc* HDL component is shown below.



The main interface is the generic 32-bit memory address bus, but it also has an interrupt vector input (with a configurable width). We somehow need to get the values on this input port to user code for processing.

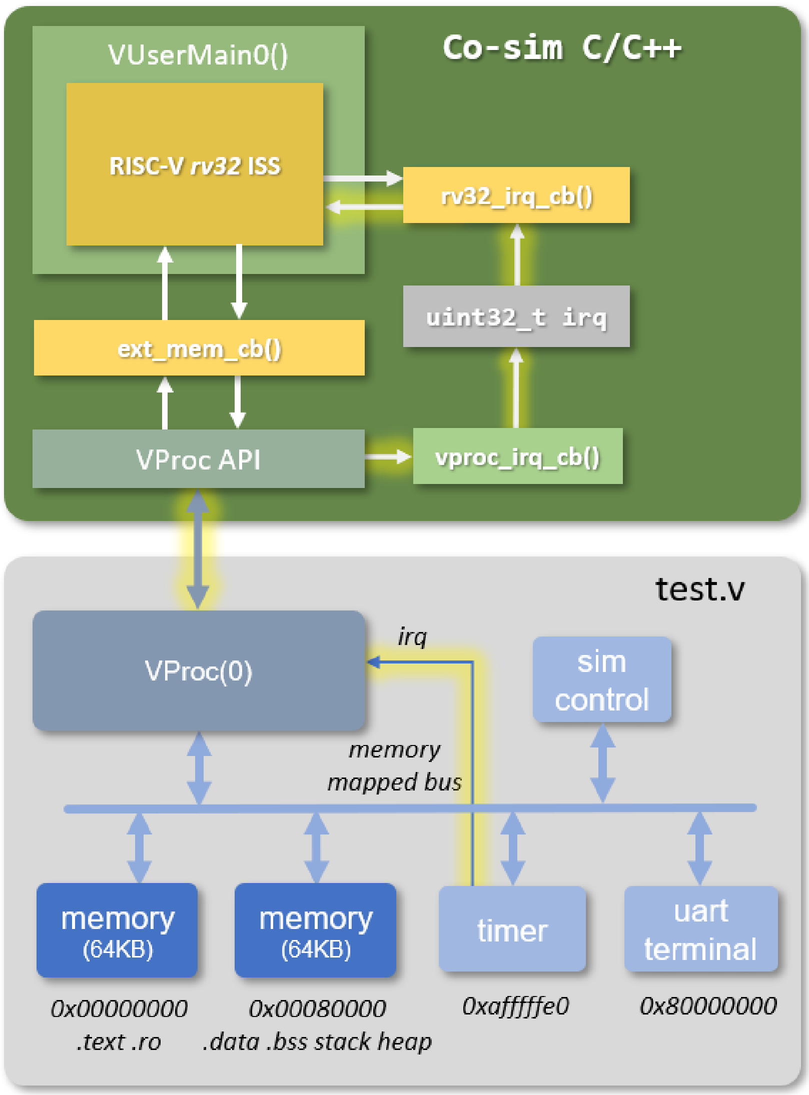
Internally, the value gets passed to the *VProc* C code whenever the input port values changes. This code monitors for changes and, if a user registered callback has been set, will call it and pass in the new interrupt state. From the callback function, the new interrupt state can be saved off for processing from the main user code. I have previously written about how to use this feature to model fully nested vectored interrupts without any complex coding methods, so I won't go into more details here, but a *VProc* C code fragment is shown below to give the idea.

```
static int      node    = 0;
static uint32_t irq_vec = 0;

static int VInterrupt (int irq) {
   irq_vec = irq;
   printf("VInterrupt received irq = 0x%08x\n", irq_vec);
}
// Main entry point for user code
void VUserMain0() {
    VRegIrq(VInterrupt, node); // Register function for interrupt

    // Tick forever
    while (1)
        VTick(1, node);
}
```

In this example (which doesn't do much), the `VInterrupt` function is registered with the *VProc* code via the `VRegIrq` API function and will be called if the `Interrupt` input changes on the HDL component in the simulation. It assigns it to a shared variable, `irq_vec`, which can then be processed by the main code (in this case, though, it isn't and the simulation loops forever, 'ticking' to allow the simulation to advance and generate new interrupts).

## Integrating the ISS and Co-simulation Together with Callbacks

Now we can bring together the *rv32* ISS, with its external memory callback and interrupt callback features with *VProc* with its transaction API and it interrupt callback. The diagram below summaries the situation of a simple SoC test environment, with a *VProc* instantiated as the processing element, running the *rv32* ISS, connected in the logic to a bus with some peripherals for memory, a timer, a UART and some test bench simulation control.



Here the timer is a source of interrupts and sends IRQ signals to the *VProc* component's **`Interrupt`** input. This gets passed through the logic programming interface and the API layer calls the registered *VProc* interrupt callback function (labelled **`vproc_irq_tb`** in the diagram). This updates some internal variable, **`irq`**, with the current interrupt status. The running *rv32* ISS, meanwhile, regularly calls its registered interrupt callback function, labelled **`rv32_irq_cb`**, which retrieves and passes to the ISS the state of the internal **`irq`** variable. This path is highlighted in the diagram.

The ISS also has a registered memory access callback, labelled **`ext_mem_cb()`**, which converts the memory access types passed in, to calls to the *VProc* transaction API to instigate transactions over the bus in the logic simulation. The callback could choose to process all accesses, and the programs code (text) and data could all be run from logic, or it could choose to only process addresses that the peripherals are mapped to, and the programme and its data would all be run out of the ISS's internal memory, speeding up the simulation—but not visible in the simulation waveforms.

So here we have an example of the use of callbacks to integrate two separate systems, each of which can run independently, but which are easily integrated both for memory accesses and for interrupt handling to allow co-simulation of a software model of a RISC-V processor to drive a logic simulation using *VProc* co-simulation.

## Conclusions

This document started off with a revision of C and C++ pointers and moved towards the concepts and syntax for pointers to functions, and even C++ methods. Once function pointers were available, the concept of a callback function was discussed were an external function can be 'injected' into a black-box set of code with no internal visibility. This black box code can then call back the user function for given situations.

Following on from this a couple of real-use examples were explored. Firstly, a RISC-V instruction set simulator, *rv32*, was integrated into the co-simulation features of OSVVM using the memory callback features of the ISS. Secondly, interrupt input to a *VProc* HDL component was routed to an IRQ callback, registered with *VProc*, to update internal state, and another IRQ callback was registered with the ISS to return the internal interrupt state when called by model. Indeed, the second example also utilised the memory callback from the ISS to make *VProc* read and write API calls, in a similar way to OSVVM. OSVVM co-simulation also supports interrupts much like

*VProc*, which is not so surprising when you know that they both share the underlying technology.

Callback functions can’t solve all the needs of a software architecture, but they are powerful when used correctly and, as I hope the examples show, can be very powerful and allow complex things to be modelled with very simple code.