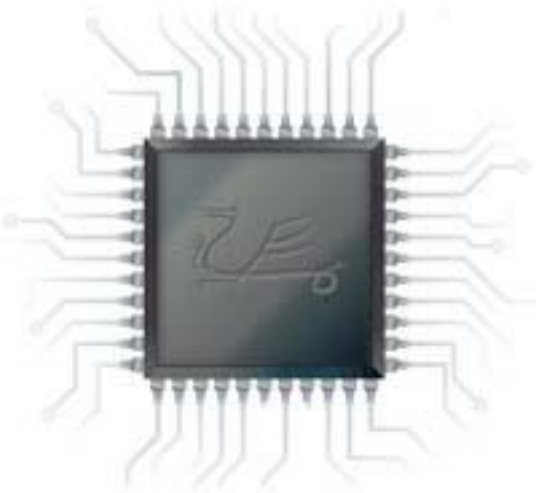
Modelling Interrupts with OSVVM Co-simulation



Simon Southwell

April 2023

Simon Southwell
Cambridge, UK
April 2023

Contents

Introduction

In this article I want to give some thoughts on how to model interrupts within the OSVVM co-simulation environment. The manual details how to register an interrupt callback function with the software and this is called whenever the interrupt request input (`IntReq`) to the `CoSimTrans` VHDL procedure changes with the new state. This makes the interrupt request state available to the software but doesn't model interrupt behaviour itself.

There are various ways of dealing with this and this article aims to look at these. Fortunately, this is likely to be straight forward as either one can pass the state to a software model that internally has interrupt control capabilities modelled within it or, when writing our own code, we can use methods that do not involve any complex methods or operating system calls. Before we look at this, though, it'll be instructive to review how interrupts work in general in a processor's internals.

How Interrupts work in a Processor

Before diving into discussing how to model interrupts in OSVVM co-simulation software 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.

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 (let's call it `IntReq`). 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.

How to Model Interrupts in an ISS

So, before we look at interrupt modelling in OSVVM co-simulation (we'll get there eventually), I want to briefly summarise how an instruction set simulator program (ISS) might model interrupts as that will be instructive to what follows.

An ISS is architected in a similar fashion to the classic stages for a processor design (well mine are at any rate): fetch, decode, execute, memory, writeback. Some of these stages might be combined. These stage functions will be in a loop, executing instructions from a memory model, updating the PC, and running until some condition is met to break out of the loop. All the while the model of the PC register is updated as per the instructions executed. To model exceptions, either at the end of the writeback when the PC is to be updated, or before the next fetch, a function can be inserted to inspect all exception states and decide if the PC is to be updated to an exception address and update any other state that the logic would do, if an exception is to be taken. The loop then continues as before, running from the exception address.

An interrupt, as we've seen, is an external signal that causes an exception, and so some means to get hold of external interrupt state is required for the interrupt processing function to inspect. One means of doing this is discussed in the next section, and we will detail it there. None-the-less, using nothing but a function to inspect state at each iteration of the instruction loop, polling the interrupt input

state as would the logic, we can model interrupts (and exceptions) with ordinary single threaded programming code, mapping to the actual operation of a processor core. An example of this is the open-source rv32 RISC-V ISS. This is meant to be structured for ease of understanding and its internals well documented, so if you're interested in diving deeper into ISS interrupt modelling architecture, this is a resource you can reference.

RISC-V ISS example in OSVVM

We're one step closer to discussing interrupt modelling in OSVVM co-simulation but I want to recap how the above mentioned RISC-V processor ISS is integrated into this environment as an example of how to use the co-simulation interrupt features if you already have a software model that includes handling of interrupts. In this case things are much easier and all that needs to happen is to communicate the interrupts state to the ISS. More details of this can be found in a previous article and details of the C++ and VHDL interfaces we'll be discussing are contained in the OSVVM co-simulation documentation.

The OSVVM co-simulation features provide interrupt requesting from the VHDL side via the calls to the `CoSimTrans` procedure which generates the address bus transactions for the logic simulation. It has an `IntReq` input as an integer type, and values can be set at each call to indicate interrupt state, with each bit, for example, being able to map to a single `IntReq` signal from a device-under-test (DUT).

```
------------------------------------------------------------------
procedure CoSimTrans (
------------------------------------------------------------------
  signal   ManagerRec       : inout  AddressBusRecType ;
  variable Done             : inout  integer ;
  variable Error            : inout  integer ;
  variable IntReq           : in     integer := 0 ;
  variable NodeNum          : in     integer := 0
) ;
```

From the C++ side, in order to get the `IntReq` input state, the code must register a function as a 'callback' function. If you're not sure what a callback function is then let me explain (software engineers, remember, logic engineers will be reading this too, but you can skip forward). A function resides in memory, like all the rest of the code. Therefore, it has an address associated with it to the start of its code. So, you can get a pointer to that address which is thus a pointer to that function. Functions also have parameters and return values associated with them, and the type for the

pointer to a function is defined to give information about this. A type definition for such a pointer is shown below:

```
typedef int (*pVUserInt_t) (int);
```

So, we have defined a new type `pVUserInt_t` which is a pointer to a function that returns an `int` value and has one argument of type `int`. If we define a function that matches this prototype, then we can create a pointer to it and assign that pointer to point to that function by using the unadorned function name:

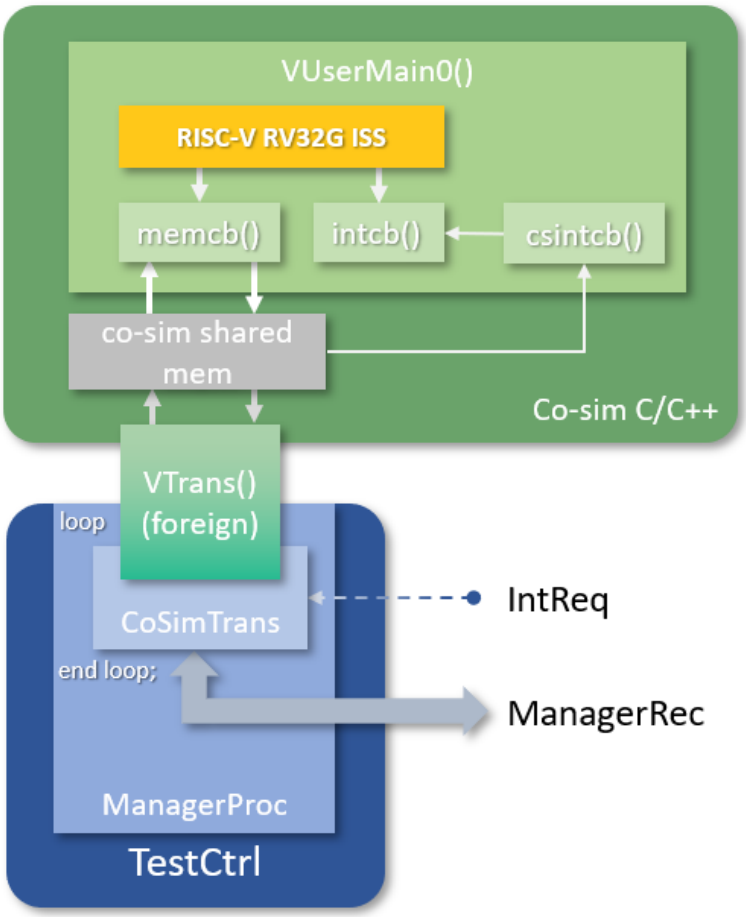
```
int csintcb (int int_req);

pVUserInt_t p_csintcb = csintcb;
```

Now we have a pointer to a function we can pass that pointer to another piece of code that can use it to call that function—e.g., `rval = (*p_csintcb)(IntReq)`. This is the callback function, so called as the main code supplies a function which is to called back when something happens—in this case when the interrupt state changes. The OSVVM co-simulation API provides a function to register a callback function:

```
------------------------------------------------------------------
void regInterruptCB (
------------------------------------------------------------------
    pVUserInt_t func
);
```

Whenever the IntReq input to the VHDL CoSimTrans procedure changes, the user's function will be called and the interrupt state passed in via the single argument. The callback function itself can then use some means to indicate to the main program the currents state. In the case of the ISS it just needs to be saved off somewhere in the program. The ISS also has callback functionality, one for calling on each memory access and one for inspecting interrupt state. The `VUserMain`*n* program registers an interrupt callback function with the ISS that simply passes on the current interrupt state that the interrupt callback to the co-simulation had saved whenever it's called. The situation is summarised in the diagram below.



So, this is the easy case, where interrupt state passes through from the logic simulation and is handed to the ISS model which already has interrupt modelling within it (as we saw earlier) and the link is complete for modelling interrupt functionality. Note, however, that the OSVVM co-simulation environment uses the OSVVM transaction models and the `CoSimTrans` procedure is only called when a transaction completes. This means that there is a latency now from an interrupt being asserted on the `IntReq` input to when the ISS sees it compared to just the ISS on its own which can see state changes for every instruction. This is not usually a problem and is the best we can do in a TLM environment.

Transaction Layer Modelling Interrupts

What if we are writing co-simulation code from scratch (a test program, say) and not using a pre-existing model with interrupt functionality as discussed above? Indeed, what if we just want to write some test code to drive some DUT IP that generates interrupts? In this section we will look at how we might model interrupts in this case (at last!).

Use and Limitations of the Interrupt Callback

It has been stated in both the co-simulation manual and the article giving a co-simulation overview that the interrupt callback function registered with the co-simulation software can't be used as a function that has access to the transaction methods of the `OsvvmCosim` object as it is part of the simulation thread and is called while an outstanding transaction is in progress. Its purpose is just to register any updated interrupt state changes in a manner accessible to the main program, when it is called by the software. Indeed, attempting to call a transaction method from within the callback would have undefined behaviour.

We need, therefore, a means to alter the main program's flow based on this updated interrupt state.

Interrupting the Main Program

As stated in the manual, the `OsvvmCosim` class has a simple API for generating transactions in the logic simulation or allowing the logic simulation to advance. For example, for word/sub-word writes and reads, and for advancing time by some tick count, the manual defines the following methods (amongst others):

```
------------------------------------------------------------------
uint<nn>_t transWrite (
------------------------------------------------------------------
  const uint<nn>_t addr,
  const uint<nn>_t data,
  const int        prot = 0 );

------------------------------------------------------------------
void transRead (
------------------------------------------------------------------
  const uint<nn>_t addr,
  const uint<nn>_t *data,
  const int        prot = 0);

------------------------------------------------------------------
int tick (
------------------------------------------------------------------
  const int ticks,
  const bool done  = false,
  const bool error = false
);
```

There are also burst transaction methods similarly defined. We saw in the section on processor interrupts that the logic will poll the interrupt state and decide whether to update the PC to instigate an interrupt. Similarly, when discussing the ISS modelling, we defined a function to do the same that's called once per

iteration—let's say before the next instruction is fetched. The OSVVM co-simulation environment is transaction based rather than instruction based, and so the level of interrupt processing—its granularity—is at the transaction level. This is the same, though, as the situation with the ISS integrated into the OSVVM co-simulation. The incoming interrupt can only be updated on the calls to the `CoSimTrans` VHDL procedure. So, we want to insert a call to an interrupt processing function before each atomic transaction just like we did for each atomic instruction on the ISS, and we will explore one possible way of doing this.

The `OsvvmCosim` class could be used as a base class which can be inherited by a derived class to extend its functionality—call it `OsvvmClassInt`. An example of such an interrupt class, available from the OSVVM release 2023.04 is provided in the C++ header file `OsvvmLibraries/CoSim/code/OsvvmCosimInt.h`. In this new class a new method is defined, `processInt()`, to carry out the detection of new interrupts and act according. To call this function before each transaction and tick methods, these methods are overloaded by the new class. The new methods call `processInt()` first and then call the base class's original method. An abbreviated outline class is shown below:

```
#include "OsvvmCosim.h"
class OsvvmCosimInt : public OsvvmCosim
{
public:
    uint8_t transWrite (const uint32_t addr, const uint8_t  data, const int prot = 0) {
         processInt();
         return OsvvmCosim::transWrite(addr, data, prot);
    }

    void transRead (const uint32_t addr, uint8_t *data, const int prot = 0) {
        processInt();
        OsvvmCosim::transRead(addr, data, prot);
    }
    // ...and so on for the rest of the transaction methods.

    void tick (const int ticks, const bool done = false, const bool error = false) {
        processInt();
        OsvvmCosim::tick(ticks, done, error);
    }

private:
    void processInt();
}
```

The new class will also need some state to keep track of the interrupt status, such as enables and active state. Most important of all, though, is defining some functions to be the interrupt service routines. In this scheme the new class will have an array

of function pointers (`pVUserInt_t isr[max_interrupts]`)—one for each interrupt level. By default, these will be initialised to `null`. A method will be provided (`registerIsr(pVUserInt_t, level)`) to allow the registering a user function for each possible interrupt level and the pointers to these functions are stored in the table. It is up to an implmenter whether a received interrupt without an associated function is an error or just ignored. The OSVVM class does the latter. The `processInt()` method, when called at each transaction, will inspect the interrupt state updated by the co-simulation interrupt callback, and call the appropriate ISR function if one is registered. The internal interrupt state can be updated with another new method, `updateIntReq(uint32_t intReq)`.

Taking the simplest case of a single interrupt, which will cover many use cases, a user program might initialise for interrupts by registering a callback function with the `OsvvmCosimInt` object (`regInterruptCB()` inherited from the `OsvvmCosim` base class). This callback function, when called, will simply call the `updateIntReq()` method with the new interrupt request state. In addition, it will register a function as an ISR using the new `registerIsr()` method, which will take a function pointer (`pVUserInt_t`, as for the co-simulation callback) and a level argument—in this case level 0. Some additional enable methods are provided to turn on/off interrupts (e.g., `enableMasterInterrupt()`, `enableIsr(int_num)` and their disable counterparts) and then the main code is ready to go, generating transactions as would any other test program.

When an interrupt becomes active, the co-simulation callback will be called which will update the interrupt state. At the next call to a transaction or tick method the `processInt()` method will be called which, enables allowing, will call the registered ISR function. Effectively, the main program is now stalled on the call to the transaction method whilst the ISR function is running. The ISR function is free to generate new transactions. This would normally be to access registers to service the interrupt and clear it. When it returns, `processInt()` will then return and the main code's call to the transaction/tick method will be unblocked and it will continue as before from where it was interrupted.

The particulars of the interrupts scheme are encapsulated in the `processInt()` method, and an implementer is free to define whatever model is required for handling interrupts, all the way to a full interrupt controller model. Typically, when an interrupt is activated, interrupts are immediately disabled. This prevents a new interrupt being generated for the existing active interrupt request on the ISR

(which, itself, will be making calls to the transaction methods, which will inspect for interrupts). As part of a typical ISR, the interrupt is cleared (if a level, rather than edge triggered, interrupt) before re-enabling new interrupts for the `IntReq`.

It should be noted that the `OsvvmCosim` transaction API parent class has no internal state (except its node number) and multiple instances, for a given node number, are allowed with which to access the same co-simulation node in VHDL. This is not true of the `OsvvmCosimInt` class, as an instance of this will have its own internal interrupt state. Software using this class must use the same instance throughout the code. An example test is provided in OSVVM where a local static is declared that is a pointer to an `OsvvmClassInt` object. The main program creates the object and accesses the normal transaction and time methods, as for any normal test program, and the interrupt callback function can then access the `updateIntReq()` method to update interrupt request state. ISR functions, also with transaction and time method calls, are registered with the `OsvvmCosimInt` object which are then called if the interrupts state and interrupt enables warrant, as determined by the internal `processInt()` method. A more complete (though not completed) definition of the `OsvvmCosimInt` class is shown below.

```
#include "OsvvmCosim.h"
class OsvvmCosimInt : public OsvvmCosim {
public:
    // Constructor
    OsvvmCsimInt(int nodeIn=0, std::string test_name="") : OsvvmCosim(nodeIn,test_name) {
        // initialise interrupt state...
    }

    uint8_t transWrite (const uint32_t addr, const uint8_t  data, const int prot = 0) {
         processInt();
         return OsvvmCosim::transWrite(addr, data, prot);
    }

    void transRead (const uint32_t addr, uint8_t *data, const int prot = 0) {
        processInt();
        OsvvmCosim::transRead(addr, data, prot);
    }
    // ...and so on for the rest of the transaction methods.

    void tick (const int ticks, const bool done = false, const bool error = false) {
        processInt();
        OsvvmCosim::tick(ticks, done, error);
    }

    // Interrupt enable/disable methods
    void enableMasterInterrupt  (void);
    void disableMasterInterrupt (void);
    void enableIsr              (const int int_num);
    void disableIsr             (const int int_num);

    // External interrupt state update method
    int  updateIntReq           (const uint32_t intReq);

    // ISR function callback registration
    void registerIsr            (const pVUserInt_t isrFunc, const unsigned level);

private:
    void processInt();              // Interrupt control functionality

    pVUserInt_t isr[max_interrupts]; // Function pointers for ISRs
    uint32_t   int_active;          // Interrupt status vector
    uint32_t   int_enabled;         // Interrupt enable vector
    bool       int_master_enable;   // Interrupt master enable
    uint32_t   int_req;             // Interrupts request input state
}
```

A full example of such a class can be found in the OSVVM co-simulation code, which includes a working definition of a `processInt()` implementation. See `OsvvmLibraries/CoSim/tests/interruptClass/VUsermain0.cpp` for its usage.

Priority and Nested Interrupts

The previous situation took the basic case of a single interrupt, but this method works for nested interrupts as well. In this case the user will register multiple ISR

functions, each associated with a different interrupt bit (they do not have to be consecutive). Which has priority is really up to how the `processInt()` method is coded. This could be with the lower (or higher) bit as highest priority, to having configurable priorities for each bit, some of which could be the same.

Now, when an ISR is active and a higher priority ISR needs to be called, the lower ISR will be interrupted at its next call to a transaction or tick methods. When the new ISR returns the lower priority ISR continues until it returns, when the main program continues again. Hopefully you can see that any depth of nested interrupts can be modelled in this way, limited just by the number of interrupt line available.

Interrupts and Ticks

We have spoken about granularity of interrupts, from instructions to transactions. The `OsvvmCosim` tick method is a different beast to the transaction level methods. It is possible, even desirable, to call the tick methods with a very large number to allow the simulation to advance some considerable time without any new transactions being generated. For programs that don't need interrupts this is not an issue. However, when using interrupts, care needs to be taken if ticking for a long period. The OSVVM co-simulation software will register interrupt state whilst the simulation is ticking but, with the method described above, since no new calls to any `OsvvmCosimInt` method is being done there will be no new calls to `processInt()` and hence no interrupts. The way to deal with this is to decide what granularity of interrupt is required for the modelling being implemented, and wrap calls to the tick methods in another function which divides the long calls into smaller calls at that granularity—which may be making multiple calls of just one tick. Now the main program will 'sleep' but can still be interrupted. The ISRs will add to the advancement of simulation time, and if that's important, then tabs will need to be kept on their execution (e.g., number of transactions issued or some such).

Conclusions

So, as I hope you can now see, the simple features of the OSVVM co-simulation API, regards interrupts, of just allowing a callback function to receive interrupt request state, is not a limitation but allows the greatest flexibility. The state can either be sent to a model that already has functionality to process it or, for new test code, a simple extension to the basic API class allows for custom interrupt processing up to and including multiple nested interrupts.

An example derived class is provided with OSVVM which can be used directly, or customized and adapted to a user's own needs. From a coding point of view a user then just provides ISR functions, which are registered as callbacks, and will be executed at the appropriate interrupt state. The granularity of the interrupt detection matches that of the co-simulation environment, which is transaction based, and therefore can only interrupt at transaction boundaries. When 'ticking' we saw that care must be taken not to introduce single call long delays which would add large latencies to processing interrupts, though they will still be registered within the model.

The OSVVM co-simulation example we have studied is specific to that environment, but the general methodology is applicable to whatever language in which you wish to model interrupt functionality—even if you won't be using co-simulation features and code just in behavioural HDL. Procedures and functions in VHDL (or the equivalent tasks and functions in Verilog/SystemVerilog) can be used in just the same way as that described in this article.